POLITECNICO
MILANO 1863

SCUOLA DI INGEGNERIA INDUSTRIALE
E DELL'INFORMAZIONE

Active Exploration Strategy for Tumor Localization in Robot-Assisted Surgery Using Bayesian Optimization

PACS Project Report

Author: **Raphael Ullrich**

Student ID: 10946123
Advisor: Prof. Formaggia
Co-advisors: Junling Fu
Academic Year: 2023

Contents

Contents i

1 Introduction 1

2 Computational Methods 3
- 2.1 Tumor Modelling . . . 3
- 2.2 Bayesian Optimization . . . 7
- 2.3 Mean Shift on Multivariate Gaussian Distribution . . . 13
- 2.4 K-means and Tumor Stiffness Classification . . . 14
- 2.5 Contour Approximation . . . 15

3 Implementation 19
- 3.1 Tumor Model . . . 20
  - 3.1.1 PolarPolygon Class . . . 21
  - 3.1.2 Triangle Class . . . 24
- 3.2 Tumor Location and Tumor Segmentation . . . 25
- 3.3 Parameters and Parameter Tuning . . . 38
- 3.4 Visualization . . . 41
- 3.5 Evaluation . . . 43
- 3.6 Processing Data Organization . . . 45
- 3.7 Installation, Compilation and Program Execution . . . 47

4 Experiments 51
- 4.1 Clustering of Tumor Errors . . . 52
- 4.2 Segmentation of Tumor . . . 52

5 Conclusion and Future Developments 63

Bibliography **65**

A Appendix **69**

A.1 Polar Straight Line 69

A.2 Parameter setup 70

A.3 Parameter classes 72

A.4 CMakeLists.txt 73

A.5 Dockerfile 76

List of Figures **79**

List of Tables **83**

1 | Introduction

Palpation is a fundamental diagnostic technique in medicine, where healthcare professionals use their hands or fingers to feel the body's surface and underlying tissues. In the process tactile feedback plays a crucial role in diagnosing conditions and informing treatment strategies. For instance, cancerous tissues often present a higher stiffness compared to the softer tissues surrounding them, such as in cases involving the prostate or breast [22],[16]. This difference in stiffness allows surgeons to detect abnormalities through palpation, aiding in the identification and removal of tumors.

Robots excel at performing repetitive tasks with high precision [9] and consistency. They can operate continuously without fatigue, maintaining a constant level of performance. In medical applications, this translates to uniformity in diagnostic procedures, which is vital for reliable and accurate results. During the last years, robot-assisted minimally invasive surgery (RMIS) has found its way into the operating theatre for cancer resection [20] due to its advantages in minimizing tissue damage, reducing infection risks, and facilitating quicker recovery times compared to traditional minimally invasive techniques [21]. However, one significant disadvantage of RMIS in clinical settings is its inability to provide surgeons with adequate tactile feedback, making it difficult for the surgeon to sense tissue mechanics and therefore detect hard inclusions efficiently. Tactile sensors can be utilized to measure tissue stiffness. These sensors measure the resistance of the tissue against the applied force. The data collected by the sensors is then analyzed to determine tissue stiffness. The integration of tactile sensors with surgical robots was discussed in multiple sources [1], [3], [15], [14] and it's deployment has proven beneficial in decreasing peak forces applied to tissue and task completion time. The crucial step in tumor detection involves creating a tactile display, or stiffness map, of the tissue. This allows for quick identification of tumors by targeting areas of increased stiffness during palpation [24]. A common strategy to collect data for a stiffness map is discrete probing [23] where a tactile sensor is mounted at the tip of a robot to perform an exhaustive point by point sampling. A more informed and guided method to obtain a stiffness map is to formulate the tumor detection as a Bayesian Optimization problem. Several works suggest this approach [24], [18], [6] and [2].

Within this project a Bayesian Optimization tumor localization strategy is implemented and augmented with a tumor segmentation strategy which was inspired by [24]. Testing and evaluation is done on a custom tumor model which implements four different tumor shapes that test different aspects of the localization and segmentation strategy.

2 | Computational Methods

In this chapter the theoretical background of deployed computational methods is presented.First, the foundations for tumor model are given and after a mathematical outline of the proposed algorithm. Components of the algorithm can be divided in a tumor localization part and a segmentation part. Localization is done by probing a tumor model and approximating a stiffness map with the help of Bayesian optimization. Subsequently, the approximated stiffness map is used to compute modes which are later explored to find tumor contour points. Approximating contour points with a curve finally yields the segmented tumor outline.

2.1. Tumor Modelling

The main goal of the presented algorithm is to detect tumorous tissue which is known to be of higher stiffness than its surrounding tissue. In the literature several models [19] and tissue phantoms [24] are being discussed. While tissue phantoms mimic the physical properties of tissue with hard inclusions well, the evaluation of a stiffness measurement is tied to a long processing time. For each query, a robotic manipulator needs to navigate its tactile sensor to a desired point and measure the tissue stiffness. To reduce processing time required for every measurement and enable rapid algorithm testing a tumor model is introduced.

In order to test the algorithm's capacity to explore complex shapes, the tumor model is comprised of a planar shape , a stiffness function mapping a stiffness value to each (x,y) coordinate and a noise model.

Shape

Two different base shapes are considered in this work. A circular tumor shape is investigated and a polygon shape. More precisely, a triangular and a rectangular base shape are used. As shown in figure 2.1, a polygon can be constructed by sampling points along the circumference of a bounding circle with radius r_b. Dividing the full circumference of

2π by the amount of desired desired vertices N yields the angular distance between each polygon vertex and the size of the polygon is defined through the radius of the bounding circle.

$$\alpha = \frac{2\pi}{N} \tag{2.1}$$

The polar coordinates of a vertex in a polygon can be expressed in polar coordinates as:

$$\mathbf{v}_i = (r_b, \alpha i) \quad i = (0..N-1) \tag{2.2}$$

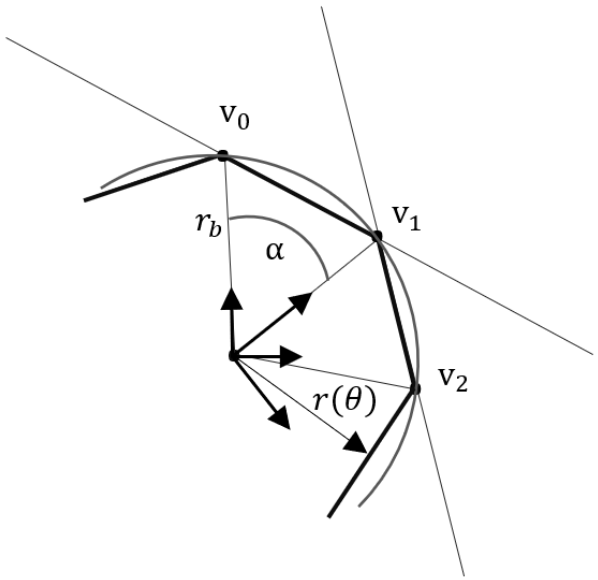

Figure 2.1: Construction of a polygon contour through straight line segments with the help of a bounding circle r_b and angle α.

The contour function of the polygon in polar coordinates can then be defined by connecting neighboring vertices through straight line segments. A polygon can therefore be represented by N line segments connecting neighboring vertices $\mathbf{v}_i = (x_i, y_i)$ and $\mathbf{v}_{i+1} = (x_{i+1}, y_{i+1})$ for $i = (0..N-1)$, where the $N-1$th vertex is defined as the starting vertex 0. The function of a straight line segment between two vertices in polar coordinates is given by:

$$r(\theta, \mathbf{v}_i, \mathbf{v}_{i+1}) = -\frac{\frac{y_i x_{i+1} - x_i y_{i+1}}{x_{i+1} - x_i}}{\frac{y_{i+1} - y_i}{x_{i+1} - x_i} cos(\theta) - sin(\theta)}, \quad \theta \in [\alpha i, \alpha(i+1)) \tag{2.3}$$

A detailed derivation is found in appendix A.1. By applying a coordinate transformation

to vertices given in 2.2 with

$$\begin{aligned} x &= rcos(\varphi) \\ y &= rsin(\varphi) \end{aligned} \tag{2.4}$$

Equation 2.3 can be used to calculate the contour function of a polygon in polar coordinates.

Stiffness Function

After defining the base shape for the tumor, stiffness values are assigned to the tumor and the surrounding region. Ishihara and Haga [8] claim that tissue matrix stiffness is critical for progression of various cancer types and cancer tissues are stiffer than surrounding non affected tissue. In the case of mammary cancer tissue is around 4 kPa stiffer than normal mammary tissue which measures stiffness values of around 0.2 kPa. In the following these values will be assumed as high and low stiffness values.

A naive approach would be to simply assign a high stiffness value if an observation falls within a base shape and a low stiffness value if the observations lies outside of the base shape. However, to add a more realistic character to the model, transition regions where high an low stiffness values meet are interpolated by a smooth function. Smoothness is guaranteed by using a third degree polynomial.

$$p(x) = ax^3 + bx^2 + cx + d \tag{2.5}$$

Coefficients are computed by using information about high and low stiffness values and their planar coordinates as well as zero slope information. Figure 2.2 depicts a stiffness function that is interpolated between $x = 1$ and $x = 2$ by a third degree polynomial.

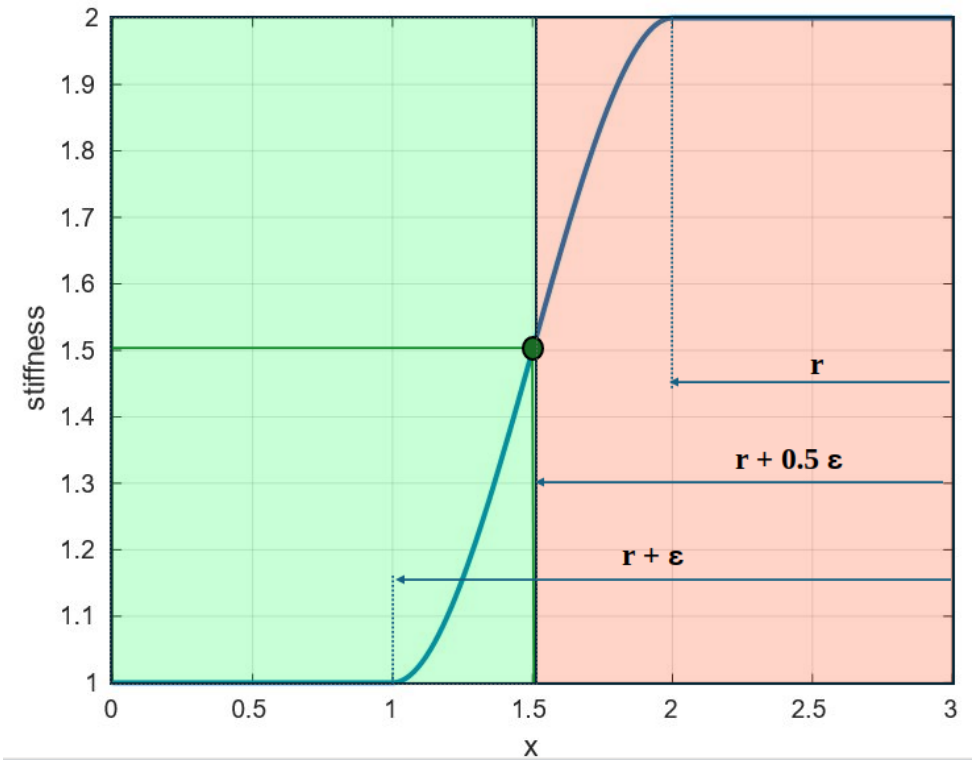

Figure 2.2: One dimensional cross-section of two dimensional smooth step function. Third degree polynomial (blue) interpolating high and low stiffness values. Smoothness of interpolation is defined by parameter ε. Ground truth stiffness is at $r + 0.5\varepsilon$ (green dot) separating tumor (red) and non-tumor (green) regions.

The polar function $r(\theta)$ given in equation 2.3 defines the contour of a tumor. By introducing a parameter ε a transition region can be defined where high and low stiffness values are interpolated by a smooth function. As can be seen in in Figure 2.2 the ground truth of the tumor contour is defined as $r(\theta) + 0.5\varepsilon$. An example of a stiffness map of a tumor model with a circular base shape is shown in Figure 2.3. As can be seen high and low stiffness areas are interpolated by smooth third degree polynomial.

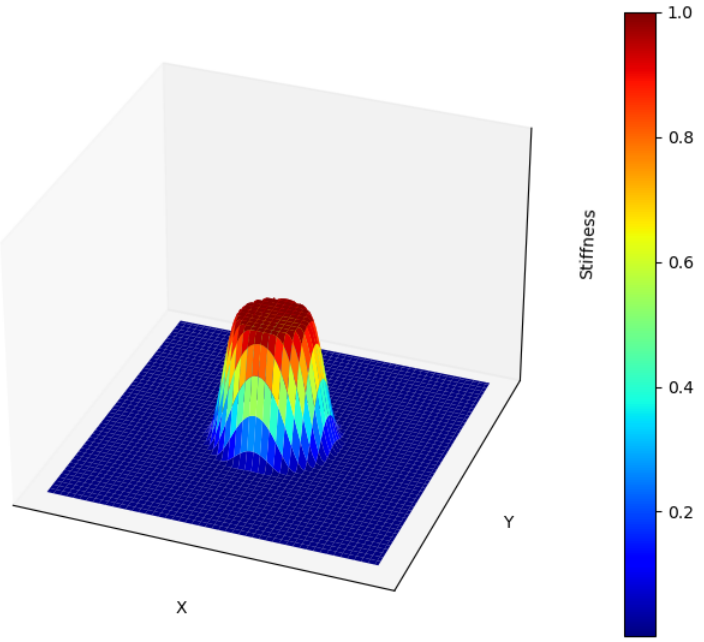

Figure 2.3: Circle tumor model. Tumor defined through circular base shape of radius r with two dimensional smooth-step.

Noise

Real world measurements are subject to noise. For this reason the presented tumor models measurement uncertainty with a Gaussian distribution. The global stiffness function f of the tumor model is added with zero mean Gaussian Noise and becomes:

$$\hat{f}(x) = f(x) + \mathcal{N}(\mu = 0, \sigma^2) \tag{2.6}$$

2.2. Bayesian Optimization

Bayesian optimization is used for black box optimization [7] where little information about the problem characteristics are known. It can be applied to problems where no closed-form representation for the objective function exists but where observation can be taken that are possibly noisy [4]. Assumptions such as that the problem is convex, gradients are easily computable and objective function values are cheap to evaluate can be relaxed when deploying Bayesian optimization (BO). These relaxations make the Bayesian optimization

approach particularly interesting when trying to detect the shape and location of a tumor with unknown topological properties. Moreover, its characteristic to require a low number of function evaluations make it especially powerful when collecting observations from an objective function by probing tissue with a robotic tactile sensor. In this work the optimization problem can be stated as follows: We have a real valued objective function that maps a stiffness value to a two dimensional domain. The function is considered to be smooth. Namely, $f : R^2 \mapsto R$ where f maps a real valued and positive stiffness value to a location $\mathbf{x} \in R^2$. The goal of the optimization is to systematically search the domain for a point $\mathbf{x}^\star$ such that f is globally maximal.

$$\max_{\mathbf{x} \in R^2} f(\mathbf{x}) \tag{2.7}$$

It is noted that the problem above can arbitrarily stated as a minimization or maximization. In the following $\mathbf{x_i}$ is denoted as the i-th sample and $f(\mathbf{x_i})$ as the observation of the objective function at $\mathbf{x_i}$. The collection of previously seen data at iteration t is defined as $D_{1:t} = (\mathbf{x_{1:t}}, f(\mathbf{x_{1:t}}))$. $P(A)$ describes the probability of A and $P(A|B)$ the conditional probability of A given B.

BO relies on the concept of probability and Bayesian inference to reason about uncertain quantities that are encountered during optimization. The optimization strategy for Bayesian optimization algorithms (BOA) stems from the principles of Bayes's theorem, which postulates that the posterior probability of a model $P(f|D_{1:t})$, given observed data $D_{1:t}$ is proportional to the prior probability of f, multiplied by the likelihood of $D_{1:t}$, given f:

$$P(f|D_{1:t}) \propto P(D_{1:t}|f)P(f) \tag{2.8}$$

The prior stiffness distribution $P(f)$ is described through a Gaussian Process (GP). The inference step can be viewed as estimating the objective function with a surrogate function.

Gaussian Process

A Gaussian Process (GP) expands the concept of a multivariate Gaussian distribution into an infinite-dimensional stochastic process and it was shown by Močkus [13] that it is well suited to serve as a surrogate model in Bayesian optimization. Parallel to how a Gaussian distribution over a random variable is completely specified by its mean and covariance, the process is defined by two functions: the mean function $\mu(\mathbf{x})$ and the

covariance function $k(\mathbf{x}, \mathbf{x}')$, denoted as

$$f(x) \sim GP(\mu(\mathbf{x}), k(\mathbf{x}, \mathbf{x}')) \tag{2.9}$$

It can be helpful to understand a GP as a function, but instead of returning a scalar $f(\mathbf{x})$ for a $\mathbf{x}$, it returns the mean and variance of a normal distribution over the possible values of f in $\mathbf{x}$. Frequently, for ease of calculation, the prior mean function is set to zero.

$$\mu = 0 \tag{2.10}$$

A commonly used covariance function is the squared exponential function, also known as the radial-basis function (RBF). This function is characterized by its smoothness and capacity for infinite differentiability, making it an appropriate choice for representing the stiffness distribution within a tumor.

$$k(x_i, x_j) = \sigma_s^2 \exp\left(\frac{-\|x_i - x_j\|^2}{2l^2}\right) + \sigma_n^2 \delta_{ij} \tag{2.11}$$

In this context, σ_s^2 represents the kernel variance or sometimes called signal variance, while l is the hyperparameter that determines the kernel's width. σ_n is know as observation noise variance, sometimes referred to as nugget, and it is problem specific. Here δ_{ij} is the Kronecker delta which is 1 for $i = j$ and 0 for $i \neq j$. A Bayesian optimization algorithm (BOA) can be divided into two key ideas: First, a sampling policy that is comprised of an initialization routine and the optimization of an acquisition function. Second, an inference step in which a Gaussian process (GP) is updated in the light of new data. An outline of a BOA is shown in Algorithm 2.1. To build a suitable surrogate function a preliminary set of samples is required. Different strategies are discussed in literature such as latin hypercube sampling, sobol sequences or uniform sampling, each with the goal of filling the optimization domain [11]. The algorithm is initialized by computing a set of preliminary samples. Next, the main loop is entered and an acquisition function is optimized to generate a new sampling point $\mathbf{x}$. Once a new point is determined, the objective function is evaluated at $\mathbf{x}$ to produce another observation. In a final step the new observation is updated to the Gaussian process to create a posterior distribution. The process involving the optimization of the acquisition function, the evaluation of the objective function, and the update of the Gaussian process is repeated for a total of $\boldsymbol{N}$ iterations.

Algorithm 2.1 Bayesian Optimization

1: Compute initial Samples
2: **for** $t = 1, 2..\boldsymbol{N}$ **do**
3: Find $\mathbf{x_t}$ by optimizing the acquisition function over the Gaussian Process: $\mathbf{x_i} = argmax_x u(\mathbf{x}|D_{1:t-1})$
4: Evaluate the objective function: $y_t = f(\mathbf{x_t}) + \epsilon_t$ (y_t denotes observation of f at $\mathbf{x_t}$ that is subject to noise ϵ_t)
5: Add observation to data $D_{1:t} = \{D_{1:t-1}, (\mathbf{x_t}, y_t)\}$ and update the Gaussian Process
6: **end for**

Given observations $\{\mathbf{x_{1:t}}, f_{1:t}\}$, where $\mathbf{x_{1:t}} = [x_1, x_2, ...x_t]$ denotes the point coordinates and $f_{1:t} = [f(x_1), f(x_2), ..., f(x_t)]^T$ the respective stiffness measurements observed at $\mathbf{x_{1:t}}$. Using a new input x_{t+1} computed by the acquisition function, the predictive distribution of f_{t+1} can be derived with the Sherman-Morrison-Woodbury formula [17]:

$$\begin{bmatrix} f_{1:t} \\ f_{t+1} \end{bmatrix} \sim N\left(\mathbf{0}, \begin{bmatrix} \mathbf{K} & \mathbf{k} \\ \mathbf{k}^T & k(\mathbf{x_{t+1}}, \mathbf{x_{t+1}}) \end{bmatrix}\right) \tag{2.12}$$

where

$$\mathbf{k} = [k(\mathbf{x_{t+1}}, \mathbf{x_1}), k(\mathbf{x_{t+1}}, \mathbf{x_2})...k(\mathbf{x_{t+1}}, \mathbf{x_t})] \tag{2.13}$$

and

$$\mathbf{K} = \begin{bmatrix} k(\mathbf{x_1}, \mathbf{x_1}) & \cdots & k(\mathbf{x_1}, \mathbf{x_t}) \\ \vdots & \ddots & \vdots \\ k(\mathbf{x_t}, \mathbf{x_1}) & \cdots & k(\mathbf{x_t}, \mathbf{x_t}) \end{bmatrix} \tag{2.14}$$

And finally a predictive distribution can be computed by solving:

$$P(f_{t+1}|D_{1:t}, \mathbf{x_{t+1}}) = N(\mu_t(\mathbf{x_{t+1}}), \sigma_t^2(\mathbf{x_{t+1}})) \tag{2.15}$$

where

$$\begin{aligned} \mu_t(\mathbf{x_{t+1}}) &= \mathbf{k}^T\mathbf{K}^{-1}f_{1:t} \\ \sigma_t^2(x_{t+1}) &= k(\mathbf{x_{t+1}}, \mathbf{x_{t+1}}) - \mathbf{k}^T\mathbf{K}^{-1}\mathbf{k} \end{aligned} \tag{2.16}$$

Figure 2.4 demonstrates the computation of four iterations of the Bayesian optimization algorithm 2.1. The plots show a one dimensional Gaussian process, an objective function that is being approximated, an acquisition function that computes the sample for the next iteration and samples. It can be seen that the posterior has highest certainty in observed data and as we move away from observations, the uncertainty increases. Below

the Gaussian process the respective graph of the acquisition function is depicted. One can observe that the maximum of the acquisition function is used as the sampling location for the subsequent iteration.

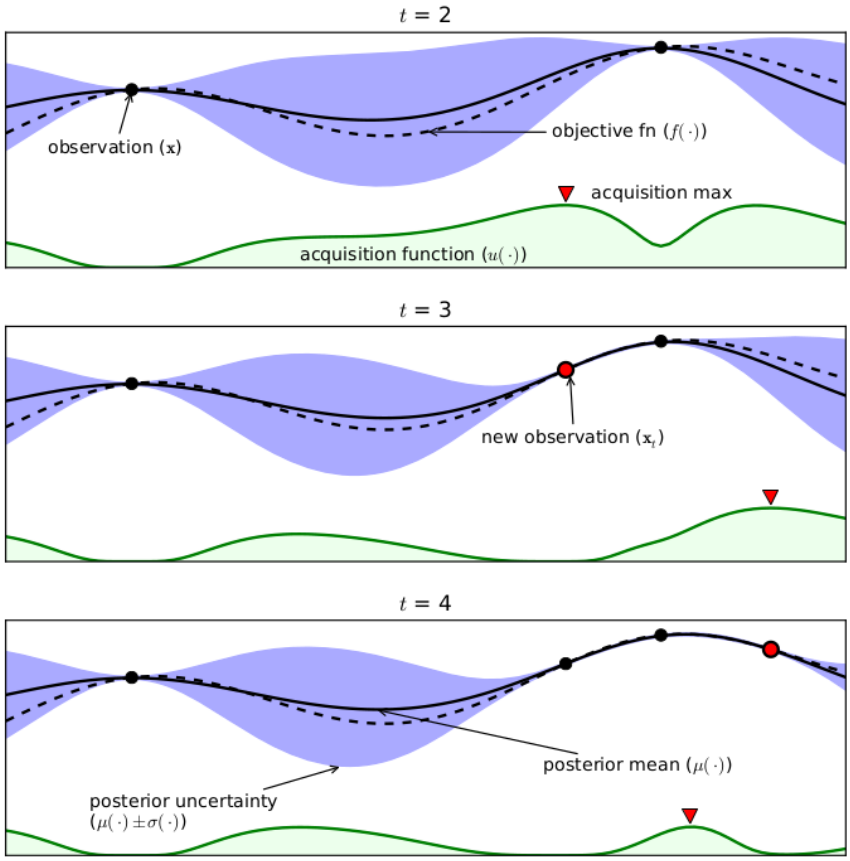

Figure 2.4: An example of a Bayesian optimization on a one dimensional objective function [4]

Acquisition Function

Bayesian optimization employs an acquisition function to efficiently select the subsequent sampling point, $\mathbf{x}_{t+1}$. This approach balances exploration (targeting areas with high uncertainty in the objective function) and exploitation (focusing on regions where the objective function is anticipated to be high). This behaviour can be witnessed in the one dimensional example of a Bayesian optimization in figure 2.4. A key advantage of this method is its focus on reducing the number of evaluations of the objective function. Additionally, it is effective in scenarios where the objective function presents several local maxima. Different strategies to find new sampling points have been proposed [11] [7] but

this body of work uses the Expected Improvement algorithm (EI) since it proved to be successful in a related work [24]. The EI function can formulated as:

$$I(x) = \max\{0, f_{t+1}(x) - f(x^+)\}. \tag{2.17}$$

$$x = \arg\max_x \mathbb{E}(\max\{0, f_{t+1}(x) - f(x^+)\}|D_t). \tag{2.18}$$

$$PDF = \frac{1}{\sqrt{2\pi}\sigma(\mathbf{x})}\exp\Big(-\frac{(\mu(\mathbf{x}) - f(\mathbf{x}^+) - \mathbf{I})^2}{2\sigma^2(\mathbf{x})}\Big) \tag{2.19}$$

Expected Improvement

$$\begin{aligned}\mathbb{E}(I) &= \int_0^\infty \mathbf{I}\frac{1}{\sqrt{2\pi}\sigma(\mathbf{x})}\exp\Big(-\frac{(\mu(\mathbf{x}) - f(\mathbf{x}^+) - \mathbf{I})^2}{2\sigma^2(\mathbf{x})}\Big)\mathrm{d}I \\ &= \sigma(\mathbf{x})\left[\frac{\mu(\mathbf{x}) - f(\mathbf{x}^+)}{\sigma(\mathbf{x})}\Phi\Big(\frac{\mu(\mathbf{x}) - f(\mathbf{x}^+)}{\sigma(\mathbf{x})}\Big) + \phi\Big(\frac{\mu(\mathbf{x}) - f(\mathbf{x}^+)}{\sigma(\mathbf{x})}\Big)\right]\end{aligned} \tag{2.20}$$

Where $\phi(.)$ is probability density function PDF and $\Phi(.)$ is the cumulative distribution function CDF. The expected improvement can be evaluated analytically with the following equation:

$$\begin{aligned}\mathbf{EI} &= \begin{cases}(\mu(\mathbf{x}) - f(\mathbf{x}^+))\Phi(Z) + \sigma(\mathbf{x})\phi(Z) & \text{if } \sigma(\mathbf{x}) > 0 \\ 0 & \text{if } \sigma(\mathbf{x}) = 0\end{cases} \\ Z &= \frac{\mu(\mathbf{x}) - f(\mathbf{x}^+)}{\sigma(\mathbf{x})}\end{aligned} \tag{2.21}$$

The expected improvement (EI) function, when calculated with respect to the predictive distribution of a Gaussian process, facilitates an optimal balance between exploration and exploitation strategies. For exploration, it is advantageous to select points with a higher surrogate variance, indicating less certainty and more potential for discovery. Conversely, for exploitation, the focus shifts to points with a higher surrogate mean, suggesting areas with known high values. To effectively manage the balance between conducting a global search and focusing on local optimization—navigating the exploration/exploitation dichotomy, it is beneficial to utilize a generalized version of the EI function. Lizotte [10] proposes incorporating a parameter ε(where $\varepsilon \geq 0$), which serves to adjust this balance,

tailoring the search strategy to specific optimization needs.

$$\mathbf{EI} = \begin{cases} (\mu(\mathbf{x}) - f(\mathbf{x}^+) - \varepsilon)\Phi(Z) + \sigma(\mathbf{x})\phi(Z) & \text{if } \sigma(\mathbf{x}) > 0 \\ 0 & \text{if } \sigma(\mathbf{x}) = 0 \end{cases}$$
$$Z = \frac{\mu(\mathbf{x}) - f(\mathbf{x}^+) - \varepsilon}{\sigma(\mathbf{x})} \tag{2.22}$$

2.3. Mean Shift on Multivariate Gaussian Distribution

Locating tumors in an underlying posterior distribution requires the computation of modes. A mode defines a high density region of data points. Such modes can be found with the help of mean shift. Mean shift is an unsupervised learning algoŕithm that is a non-parametric, iterative and used for data clustering. It seeks to identify centroids or points of maximum density in a given space, effectively locating the centers of clusters. The algorithm starts with an initial estimate for the centroid, then iteratively shifts this estimate towards regions of higher density by averaging the points within a given radius. This process is repeated until convergence, with the centroid no longer moving significantly. Mean shift does not require specifying the number of clusters beforehand, making it particularly useful in applications where the number of clusters is unknown, such as the localization of tumors. Its ability to adapt to the data's underlying structure allows for the identification of clusters of different shapes and sizes, providing a versatile tool for data analysis and pattern recognition [5]. Figure 2.5 illustrates the mechanism of means shift on a two dimensional scattered data set. An initial window with a defined radius that is referred to as "bandwidth" is given by N. The center of the initial window is defined by x. The center of mass of $N(x)$ is computed by a weighting the neighboring samples in with a Gaussian kernel function $K(x_i - x) = e^{-c\|x_i - x\|^2}$:

$$m(x) = \frac{\sum_{x_i \in N(x)} K(x_i - x)x_i}{\sum_{x_i \in N(x)} K(x_i - x)} \tag{2.23}$$

The newly computed point $m(x)$ is the center of mass of N and it is used to update the algorithm $x \leftarrow m(x)$. The algorithm converges if the euclidean norm of the mean shift vector $\|m(x) - x\|$ falls below a certain threshold.

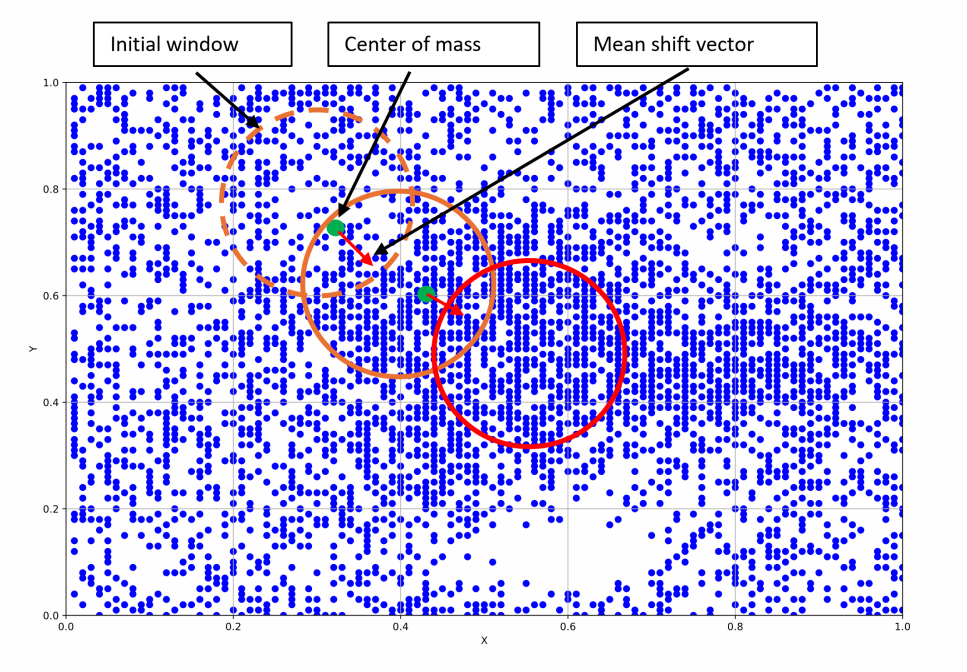

Figure 2.5: Convergence of mean shift toward high density regions. Circular windows depict the regions used to compute a center and mean shift vector that points toward direction of maximum increase in density. The size of the window is defined through bandwidth N.

2.4. K-means and Tumor Stiffness Classification

K-means

K-means is a clustering algorithm that belongs to the group of unsupervised learning techniques. The algorithm assigns previously unseen data points to their according clusters. Unlike Means Shift, k-means requires knowledge about existing labels in the data set and the number of clusters K has to be specified in advance. K-means aims to partition n observations into K clusters in which each observation belongs to the cluster with the nearest mean, serving as a prototype of the cluster. The objective is to find:

$$\arg\min_{S} \sum_{i=1}^{K} \sum_{\mathbf{x} \in S_i} \|\mathbf{x} - \mu_i\|^2 \tag{2.24}$$

where $S = \{S_1, S_2, \ldots, S_K\}$ is the partition of n observations into K clusters, $\mathbf{x}$ is an observation, μ_i is the mean of points in S_i. Convergence is achieved when the assignments no longer change.

Stiffness Measurement Classification

For this work the clustering capability of k-means is leveraged to classify stiffness measurements. It is assumed that measurements either correspond to tumorous on non-tumorous tissue. Therefore, the algorithm is trained on a one-dimensional dataset with two labels. Given that high stiffness values correlate with tumorous tissue, the cluster with the larger values is selected and its mean value and uncertainty calculated. Based on these measures, the threshold multiplier β will define how many standard deviations away from the mean value, measurements are categorized as tumor tissue. Figure 2.6 demonstrates how the threshold value is determined based on the mean value μ of the cluster and its standard deviation σ.

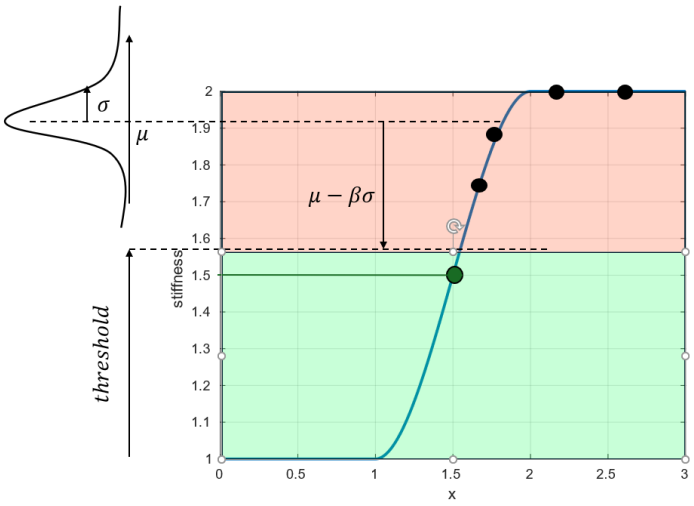

Figure 2.6: Computation of stiffness value based on in-cluster variance and mean value. The threshold is calculated by subtracting the threshold multiplier times the in-cluster standard deviation $\beta\sigma$ from the mean value of the cluster μ.

2.5. Contour Approximation

Contour Points Computation

The outline of a tumor is defined by a set of contour points. These characteristic points are found by exploring the earlier computed tumor center. With the information of where a tumor center is located, contour points are identified by exploring the centroid in equiangular radial directions 2.7. In a real-world scenario, the exploration is done by linearly sliding the tactile sensor over the tissue, starting at the center of the tumor and ending when the stiffness measurement falls below the earlier defined criterion 2.6.

In simulation the sliding operation is replicated by taking equidistant samples along a straight path.

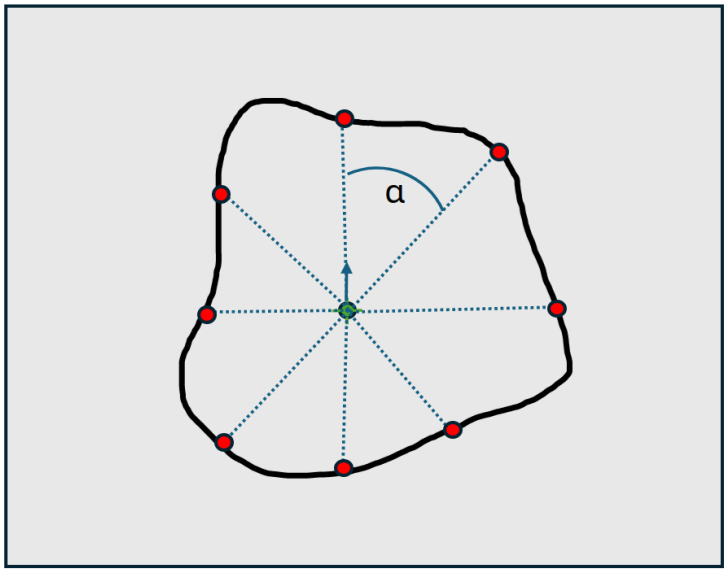

Figure 2.7: Exploration of contour points. The tumor centroid (green dot) radially explored in $n = \frac{360}{\alpha}$ directions (blue dashed line) until the tumor outline (black line) is detected. The search for a contour point in each direction is stopped and a contour point identified (red dot) when stiffness measurement falls below stiffness threshold 2.6.

Cubic Spline Interpolation

After successfully identifying tumor contour points, the next step is to approximate a smooth and differentiable mathematical representation of the tumor contour. For this purpose contour points are interpolated with a cubic spline. Cubic splines are a sequence of third order polynomials connected at the the endpoints with boundary conditions. For each interior point connecting two spline segments, first and second order derivatives are required to be equal. A two dimensional contour C can be expressed as a parametric function $\mathbf{s}(t) = (s_x(t), s_y(t)), t \in [0, 1]$ where s_x and s_y are cubic splines.

Given a set of contour points $(t_0, \mathbf{f_0} = \mathbf{f}(t_0)), (t_1, \mathbf{f_1} = \mathbf{f}(t_1)), \ldots, (t_n, \mathbf{f_n} = \mathbf{f}(t_n))$, the cubic spline $\mathbf{s}(t)$ on each interval $[t_i, t_{i+1}]$ is defined by:

$$\begin{bmatrix} s_x(t) \\ s_y(t) \end{bmatrix} = \begin{bmatrix} a_{xi}(t-t_i)^3 + b_{xi}(t-t_i)^2 + c_{xi}(t-t_i) + d_{xi} \\ a_{yi}(t-t_i)^3 + b_{yi}(t-t_i)^2 + c_{yi}(t-t_i) + d_{yi} \end{bmatrix} \tag{2.25}$$

where $i = 0, 1, \ldots, n-1$. The conditions for the cubic spline interpolation are:

1. $\mathbf{s}_i(t_i) = \mathbf{f}_i$ for all i.
2. $\mathbf{s}_i(t_{i+1}) = \mathbf{s}_{i+1}(t_{i+1})$ for all i.
3. $\mathbf{s}_i'(x_{i+1}) = \mathbf{s}_{i+1}'(x_{i+1})$ and $\mathbf{s}_i''(x_{i+1}) = \mathbf{s}_{i+1}''(x_{i+1})$ for all i.

By formulating a system of equations with contour points and conditions 1-3 spline coefficients $\mathbf{a}_i$ and $\mathbf{b}_i$ can be computed. In order to obtain a closed coutour representation start and endpoint need to be equal $\mathbf{s}_0(t_0) = \mathbf{s}_{n-1}(t_{n-1})$. Figure 2.8 demonstrates the interpolation of contour points with a closed parametric spline curve. It can be seen that conditions 1-3 hold as function values coincide with contour points, slopes at interior points match and curvatures at interior points is the same.

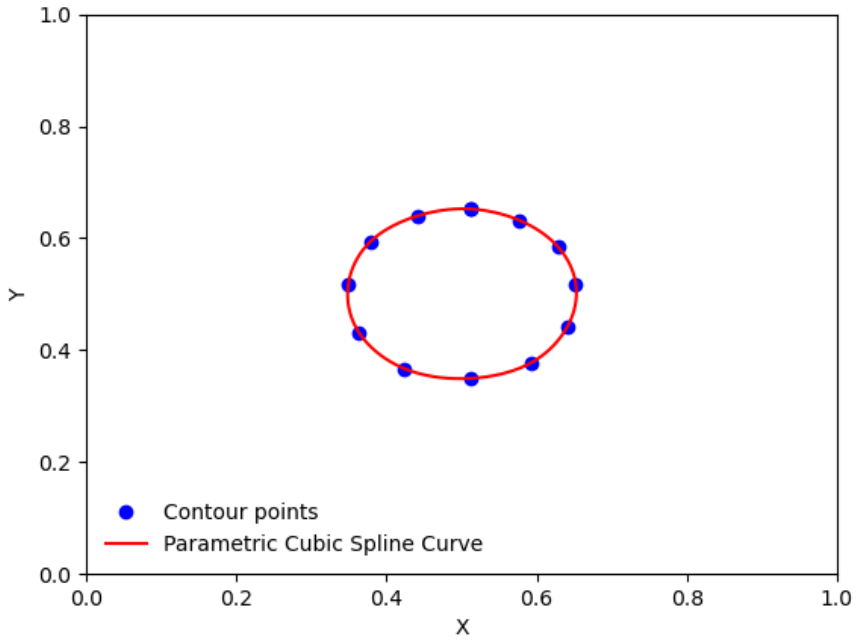

Figure 2.8: Contour approximation through interpolation of twelve contour points (blue) with a parametric cubic spline (red).

3 | Implementation

In this chapter the implementation of the proposed tumor localization and segmentation algorithm is analyzed.

The developed program is written and compiled with C++17 and relies on several external libraries:

1. BayesOpt library for Bayesian optimization algorithms
2. mlpack library to use machine learning algorithms
3. alglib library to perform curve fitting operations
4. CGAL library for operations with two dimensional polygons
5. Matplot++ library for visualization

The latest version, installation instructions as well documentation can be found and downloaded from the GitLab repository (main branch). Further, inside the README provided in the project directory, instructions for the compilation of additional Doxygen documentation are found. The project was developed as a simulation environment to demonstrate the effectiveness of the Bayesian optimization algorithm for the segmentation of tumors. Critical parameters required for the shape of the tumor model as well as parameters related to the segmentation algorithm itself can be tuned easily without the need of recompilation. Simulation results are organized in dedicated directories that allow later analysis and comparison among experiments. Since acquiring real tumor measurements is costly the simulation of the tumor segmentation algorithm relies on a tumor model. Nevertheless, functionality and classes are structured in a way that allow future works to deploy the algorithm with small modifications to a real-world scenario.

This work comprises three compilation targets with different purposes:

1. The first compilation target is called *compute_contour*. It runs the core algorithm and reads parameters from dedicated files and produces simulation results.
2. Second, *display_gp* extends the target above by a visualization suite that gives

insight into the process.

3. Lastly, *evaluate* was developed to augment *compute_contour* by the calculation of metrics that give rise to quality of the segmentation.

3.1. Tumor Model

As described above, three different geometries were implemented as shape primitives to serve as a tumor model. Additionally, a fourth model was developed to test the algorithms capabilities to detect two tumors. Along with the tumor model class, *Class RoboticPalpation* was conceptualized. The class is not implemented in the current state but future works might use this class as a starting to run the algorithm with a robotic system that produces stiffness stiffness measurements. Four classes, *Class Circle*, *Class Rectangle*, *Class Triangle* and *Class TwoCircles* denote the different tumor models that the simulation can be run on. Each class inherits from a parent *Class Shape* that holds variables and functions relevant for child classes. Additionally, *Class Shape* inherits from *bayesopt::ContinuousModel* class that is used to interface the a custom model with bayesopt library. *bayesopt::ContinuousModel* itself inerits from *bayesopt::BaseOptBase*. The class relationship can be seen in Figure 3.1

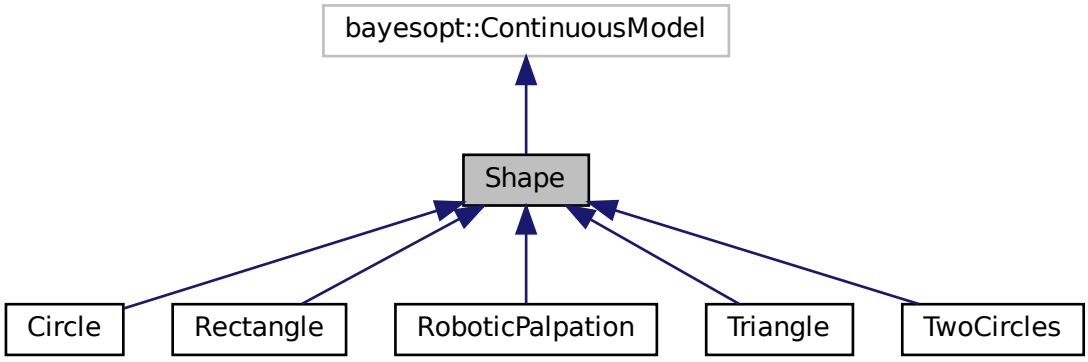

Figure 3.1: Inheritance structure of tumor models.

In the following the functionality of the tumor model is further outlined with the Triangle Class presented in Figure 3.2.

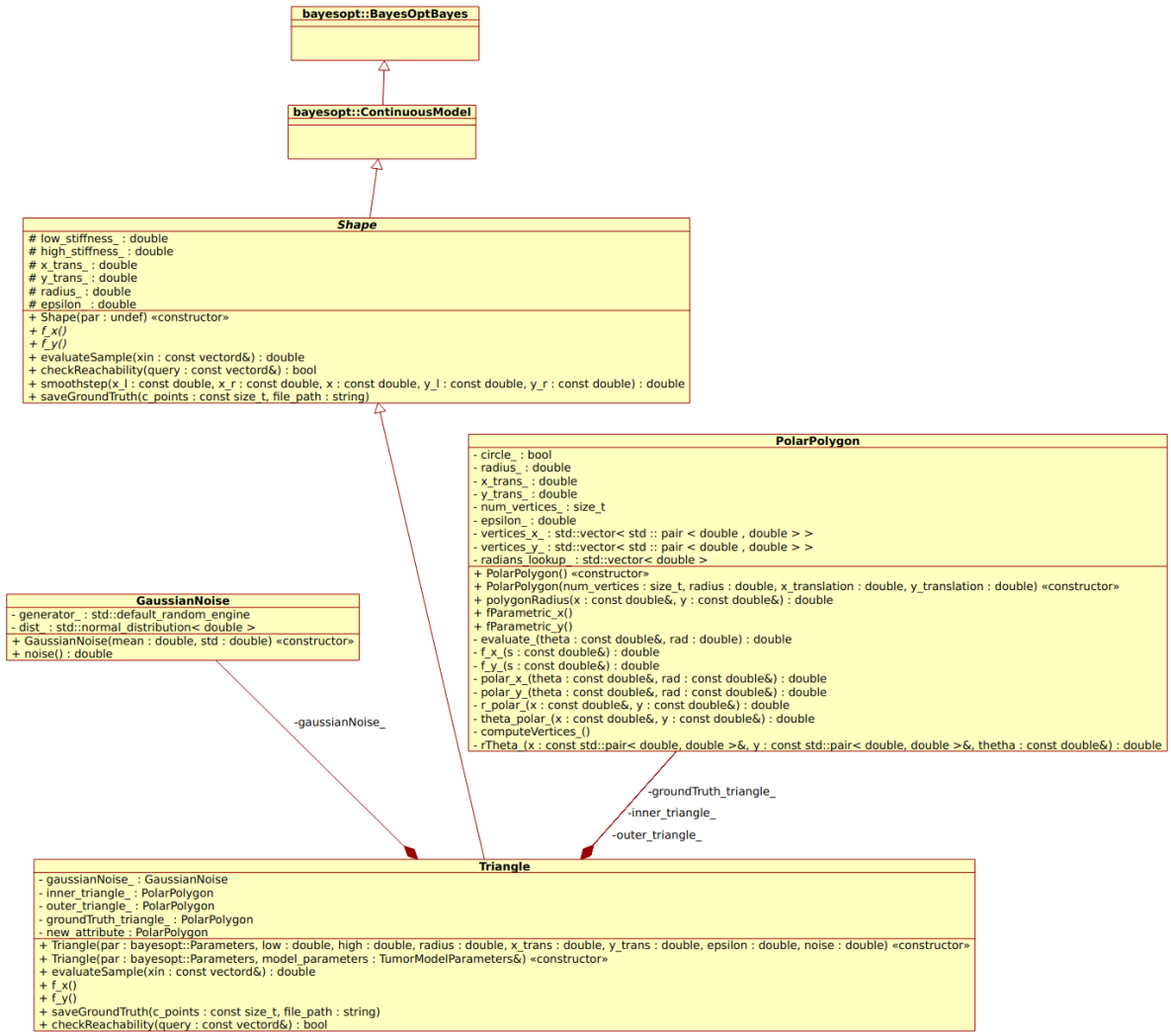

Figure 3.2: UML class diagram for Triangle tumor model.

3.1.1. PolarPolygon Class

PolarPolygon Class defines a polygon shape in polar coordinates. The planar coordinates of the polygon's vertices are defined through a bounding circle. Depending on the number of vertices, same-sized angle segments are computed that divide a full circle and constrain the vertices to the circumference of a circle with a pre-defined radius.

Upon creation of a polygon object the number of vertices, its radius and the the relative coordinate offset in x an y need to be passed. A circle is treated as a special type of polygon by this class and can also be instantiated by passing a value for vertices that is below three.

Central to *PolarPolygon Class* is the function *rTheta_* defined in figure 3.3. The function

calculates values for the polar coordinate r of a line segments connecting two points in Cartesian coordinates for a given angle theta.

```
1  double PolarPolygon::rTheta_(const std::pair<double,double> &x,
2                               const std::pair<double,double> &y,const double &theta )
3  {
4    return ((x.second*y.first- x.first*y.second)/(x.second-x.first))
5    *1/( sin(theta) - ((y.second-y.first)/(x.second-x.first))*cos(theta) );
6  }
7
8
```

Figure 3.3: rTheta_ function

The function is later used internally by *evalaute_* in figure 3.4 which calls the functions on the polygon's vertices in Cartesian coordinates. *evalaute_* parametrizes the outline of a polygon and returns its radius in polar coordinates for a given angle theta. For this, the function internally leverages a lookup table (*radians_lookup_*) created by *computeVertices_* at the beginning of the objects lifetime. By comparing the queried angle *theta* with angle segments generated in the lookup table, *rTheta_* can be called with the according vertex coordinates. For this, polar coordinates are transformed to Cartesian space by functions *polar_x_* and *polar_y_*.

```
1  void PolarPolygon::computeVertices_()
2  {
3    for(size_t i= 0; i<num_vertices_; i++)
4    {
5      vertices_x_.push_back(std::make_pair(cos((M_PI*2/num_vertices_)*i), cos( (M_PI*2/num_vertices_)*(i+1))));
6      vertices_y_.push_back(std::make_pair( sin((M_PI*2/num_vertices_)*i), sin( (M_PI*2/num_vertices_)*(i+1))));
7      radians_lookup_.push_back((M_PI*2/num_vertices_)*i);
8    }
9    radians_lookup_.push_back(M_PI*2);
10 }
11 double PolarPolygon::polar_x_(const double &theta, const double &rad )
12 {
13   return  rad * cos(theta);
14 }
15 double PolarPolygon::polar_y_(const double &theta, const double &rad )
16 {
17   return rad * sin(theta);
18 }
19 double PolarPolygon::evaluate_(const double & theta, const double & rad)
20 {
21   int idx = 0;
22   //deals with undefined behaviour for theta = 0
23   if(theta == 0){
24     return rad;
25   }
26   for (size_t i = 0; i < radians_lookup_.size(); i++)
27   {
28
29       if (radians_lookup_[i] >= theta)
30       {
31         idx = i;
32         break;
33       }
34
35   }
36   double rt = rTheta_(std::make_pair( polar_x_(radians_lookup_[idx-1], rad)
37                                       ,polar_x_(radians_lookup_[idx],rad)),
38                       std::make_pair( polar_y_(radians_lookup_[idx-1],rad),
39                                       polar_y_(radians_lookup_[idx],rad) ), theta);
40   return rt;
41 }
42
```

Figure 3.4: *Polar Class* functions

In order to query whether a point is contained within a polygon or not the class implements a function named *polygonRadius* displayed in figure 3.5 to calculate the radius of the polygon's contour for a given point in Cartesian coordinates. The function is later used by *Triangle Class* shown in section 3.1.2 to compare whether a given point lies within the polygon contour or not.

```
1  double PolarPolygon::polygonRadius(const double &x, const double & y)
2  {
3    if(!circle_)
4    {
5      double theta = theta_polar_(x-x_trans_,y-y_trans_);
6      if(isnan(theta))
7      {
8       return 0.0;
9      }
10     return abs(evaluate_(theta,radius_));
11   }
12   else return radius_;
13 }
14
```

Figure 3.5: Caption

3.1.2. Triangle Class

Triangle class is instantiated by passing an object of type *bayesopt::Parameters* that contains the necessary parameters related to the BOA and an instance of class *TumorModelParameters* that stores tumor model parameters. Central to *Triangle Class* is the method *evaluateSample* illustrated in figure 3.6 which implements the cost function for the optimization procedure. The function takes a two dimensional coordinate as vector type object and returns a decimal value which in this case represents the stiffness of the tumor. As explained in section 2.1 each shape is represented in polar coordinates with an inner and an outer shape that are interpolated by a third-degree polynomial. To achieve this, *Triangle Class* instantiates two *PolarPolygon Class* objects, where one shape has a smaller diameter than the other. In total, this separates the exploration domain in three sections where each section defines a different cost function. The inner shape assigns constant stiffness values correlating with the stiffness of tumorous tissue. Every point that is not included within the set of the outer shape is considered non-tumours tissue and is assigned a lower constant stiffness value. For every point that falls within the set of the outer shape but not in the set of the inner shape, a stiffness value is calculated by a smooth-step function as explained above. Lines 13-22 in figure 3.6 demonstrate how this is implemented.

```
double Triangle::evaluateSample( const vectord& xin)
{
 if (xin.size() != 2)
  {
    std::cout << "WARNING: This only works for 2D inputs." << std::endl
    << "WARNING: Using only first two components." << std::endl;
  }
  double x = xin(0);
  double y = xin(1);
  double r = sqrt((x - x_trans_)*(x - x_trans_) + (y - y_trans_)*(y - y_trans_));
  double radius_inner = inner_triangle_.polygonRadius(x,y);
  double radius_outer = outer_triangle_.polygonRadius(x,y);
  if(r < radius_inner){
    return -high_stiffness_+ gaussianNoise_.noise();
  }
  else if (r >= radius_outer)
  {
    return -low_stiffness_+ gaussianNoise_.noise();
  } else
  {
    // smoothstep to interpolate between 1 and 0 for the edge of the circle
    return smoothstep(radius_inner, radius_outer, r, -high_stiffness_,-low_stiffness_);
  }
}
```

Figure 3.6: *evaluateSample* function.

As can be seen in figure 3.6 each sample is subject to Gaussian noise. A class called *GaussianNoise* presented in 3.7 computes zero mean Gaussian noise and its member *noise* is used to add noise to the stiffness measurements.

```
class GaussianNoise{
  public:
    GaussianNoise(double mean, double std);
    double noise();
  private:
    std::default_random_engine generator_;
    std::normal_distribution<double> dist_;

};
```

Figure 3.7: *Class GaussianNoise*

3.2. Tumor Location and Tumor Segmentation

Contour Class

Contour Class from figure 3.8 was developed with the idea in mind to organize core functionality for tumor localization and tumor segmentation. While serving as a standalone class to simulate the algorithm on a tumor model (a minimal working example is shown in figure 3.9) the class can also be used as a collection of functions and containers to process and store the current state of the algorithm. A selection of getter functions and the option to step run (*stepRunGaussianProcess*, figure 3.8) the Gaussian process provide access and functionality to run the tumor localization and segmentation algorithm in an external state machine (figure 3.10) such as a viusalization loop.

Figure 3.8: Inheritance structure.

```
bayesopt::Parameters par; //Parameters used for Bayesian Optimization
ContourParameters contour_parameters; //Parameters used by Contour Class
TumorModelParameters model_parameters; //Parameters used for tumor model
std::string dirName="path/where/results/are/stored";
//Instantiate tumor model that algorithm is run on
std::unique_ptr<Shape> shape = std::make_unique<Triangle>(par,model_parameters);
//Create Contour Class object with tumor model and
Contour contour(shape.get(),contour_parameters, dirName);
//Run bayesian optimization
contour.runGaussianProcess();
//Call means shift algorithm to find tumor centers
contour.computeCluster();
//Search for contour points
contour.exploreContour();
//Approximate Contour
contour.approximateContour();
```

Figure 3.9: First, three sets of parameters are created, one that defines the behaviour of the Bayesian optimization routine, another one that controls how the tumor contour is found and finally a set that defines the tumor model. Next the *Contour* object is created by passing in a tumor model, contour parameters and a dedicated directory to store results in. Lastly, the routines *runGaussianProcess*, *computeCluster*, *exploreContour* and *approximateContour* will perform the necessary computations to locate and segment the tumor.

```
 enum RunningStatus{RUN, STOP, CENTROIDS, CONTOUR_POINTS, CONTOUR_APPX};
 //Get Total number of runs
std::vector<double> lx(contour->getTotalNumberOfSamples());
std::vector<double> ly(contour->getTotalNumberOfSamples());
RunningStatus status = RUN;
Contour *contour;
//get granularity of Gaussian process (GP)
c_points = contour->getCPoints();
size_t state_ii = 0;
//Initialize GP
contour->prepareGaussianProcess();
//Get samples from initalization routine
contour->getInitialSamples(lx,ly);
while(1)
{
    if(status == RUN)
    {
        state_ii++;
        contour->stepRunGaussianProcess();
        const vectord last = contour->getLastSample();
        lx.push_back(last(0));
        ly.push_back(last(1));
        //Contains data such as posterior, standard deviation, aquisition function ..
        bayesopt::ProbabilityDistribution* pd = contour->getPredictionGaussianProcess(q);
        if (state_ii == contour->getNumberOfRuns())
        {
            status = CENTROIDS;
        }
    }
    if(status == CENTROIDS)
    {
        contour->computeCluster();
        computeClusters=false;
        //Do something with computed clusters, e.g visualization
        std::vector<Point> clusters = contour->getClusters();
        status = CONTOUR_POINTS;
    }
    if(status == CONTOUR_POINTS)
    {
        contour_->exploreContour();
        //Do something with computed points, e.g visualization
        std::vector<Point> cpoints = contour->getContourPoints();
        status = CONTOUR_APPX;
    }
    if(status == CONTOUR_APPX)
    {
        contour->approximateContour();
        //Do something with computed spline interplolant, e.g visualization
        SplineInterpolant_ptr_pair_vec spline_pairs = contour->getSplineInterpolant();
        status = STOP;
    }
    if(status==STOP) break;
}
```

Figure 3.10: Example code for a state machine using Contour class.

Bayesian Optimization

BayesOpt is an efficient implementation of Bayesian optimization methodology for non-linear optimization [12] written in C++. The library is well documented and an array of example programs proved helpful for the development process. The *Contour class* described above wraps around the object *bayesopt::BayesOptBase* bopt_model* which has access to the cost function (*evaluateSample*, figure 3.6) and the object provides control and information over the underlaying Gaussian process. *Contour class* receives a pointer to the *bayesopt::BayesOptBase* bopt_model* upon creation in the constructor. 3.11

```
1 Contour(bayesopt::BayesOptBase* bopt_model,ContourParameters cp, std::string experiment_path)
2 {/.../}
```

Figure 3.11: Contor Class constructor.

The optimization is initialized by running *initializeOptimization* from figure 3.12 which internally creates the posterior surrogate model from provided parameters and it is called before every optimization. Further, it runs an algorithm to collect initial samples, stores them and updates the surrogate model with the initial samples.

```
1 bopt_model_->initializeOptimization();
```

Figure 3.12: Initialization of Bayesian optimization

In order to perform an update step to the posterior model, *Contour Class* calls *stepOptimization* in figure 3.13. Internally, the function computes the next sample by querying the cost function. Next, the internal surrogate model is updated according to equation 2.16. Lastly, the calculated posterior distribution is used to compute the optimum of the expected improvement function (EI) shown in equation 2.22. The implementation of bayesopt library offers several surrogate models and acquisition functions as well as methods to relearn kernel parameters. This work implements the bayesian optimization routine with a Gaussian Process with a constant mean function 2.10 and a Gaussian Kernel function 2.11 with constant parameters (no relearning of kernel parameters). Further, expected improvment (EI) is used as the critera to determine new sampling points throughout this work. The corresponding parameters setting is detailed in section 3.3.

```
1 bopt_model_->stepOptimization();
```

Figure 3.13: Update surrogate posterior model with new sample

Tumor Localization

After approximating a stiffness function with a Gaussian process the next step is to find modes in the distribution. Modes in this context are equivalent to points of high stiffness. Whereas there can only be a single maximum point, there can be multiple modes of varying value. In this context we are interested in modes since high stiffness regions correlate potentially with tumorous tissue. In this work, modes are computed by running a means shift clustering algorithm on the posterior distribution as shown in figure 3.14. In comparison to other clustering algorithms like k-means, means shift is able to determine the number of clusters by itself. Hence, the number of tumors does not need to be known beforehand.

```
1 void Contour::computeCluster()
2 {
3     mean_shift_ = MeanShift(posterior_, bandwidth_, samples_, experiment_path_);
4     mean_shift_.meanshift_mlpack();
5    /*..*/
6 }
```

Figure 3.14: Function call to perform means shift clustering on posterior distribution.

Class MeanShift given in figure 3.15 contains an efficient implementation of mean shift from the machine learning C++ library mlpack.

```
class MeanShift
{
private:
    std::string experiment_path_;
    std::vector<std::vector<double>> data_;
    std::vector<Point> scattered_points_;
    double bandwidth_;
    arma::mat centroids_;
    arma::Row<size_t> assignments_;
    std::string working_dir_path_;
    bool pointNoPoint_(float probabilityOfPoint);
    void scatterData_(std::vector<Point>& points_);
    bool saveNormalizedDataToCSV_();
    bool savePointsToCSV_(std::string name, std::vector<Point>& points_);
    void loadDataFP_(std::string filename, std::vector<std::vector<double>> &data);
public:
    MeanShift();
    MeanShift(std::vector<std::vector<double>> data, double bandwidth, std::string experiment_path);
    MeanShift(std::string file, double bandwidth, std::string experiment_path);
    ~MeanShift();
    void printClusters();
    void meanshift_mlpack();
    std::vector<std::vector<double>> getCentroids();
    void saveResultsToFile();
};
```

Figure 3.15: Function call to perform means shift clustering on posterior distribution.

The class operates on a two dimensional distribution that is passed to the constructor. The distribution is a $n \times n$ square matrix where each entry corresponds to a probability value. Along the data that mean shift is run on, mean shift depends on a bandwidth parameter. The parameter determines the size of the radius that encircles adjacent points as explained in previous section 2.3. Before mean shift is able to compute modes, *scatterData_* Figure 3.16 is run upon instantiation. Inside, the function samples data points from the two dimensional probability distribution. This is done by normalizing the data before sampling it (*pointNoPoint_*, figure 3.16).

```
bool MeanShift::pointNoPoint_(float probabilityOfPoint)
{
  return rand()%100 < (probabilityOfPoint * 100);
}
void MeanShift::scatterData_(std::vector<Point>& scattered_points)
{
    double x = 0.0;
    double y = 0.0;
    int num_data_points = data_[0].size();
    double increment_x = 1.0/(double) num_data_points;
    double increment_y = 1.0/(double) num_data_points;
    for(size_t i=0; i<data_.size(); ++i)
        {
        for (size_t j=0; j<num_data_points; ++j)
        {
            x += increment_x;

            if(pointNoPoint_(data_[i][j]))
            {
                Point p(x,y);
                scattered_points.push_back(p);
            }
        }
        x=0;
        y += increment_y;
        }
}
```

Figure 3.16: *scatterData_* and *pointNoPoint_* functions used to obtain a scattered points representation from probability distribution.

As shown in figure 3.14 the clustering algorithm is run by calling the method *meanshift_mlpack* shown in figure 3.17 which creates a MeanShift object from the mlpack library, assigns the bandwidth and runs it on the scattered data (*data*). The resulting centroids (*centroids_*) and assignments for each data point(*assignments_*) are stored inside the class.

```
void MeanShift::meanshift_mlpack()
{
    /*...*/
    mlpack::meanshift::MeanShift<> meanShift;
    meanShift.Radius(bandwidth_);
    meanShift.Cluster(data, assignments_, centroids_, forceConvergence, true);
}
```

Figure 3.17: Function that accesses mlpack's mean shift algorithm.

Contour Points Search

K_means Class figure 3.18 is a custom class that wraps mlpack's k-means implementation. It takes a one dimensional vector of stiffness measurements and two initial guesses associated with high and low tumor stiffness values. The class is designed to classify a one-dimensional vector into two categories: values attributed to tumor tissue and values attributed to non-tumorous tissue.

```
1  class K_means
2  {
3  public:
4      K_means(const std::vector<double> &vals,const double low_stiffness,const double high_stifness);
5      void cluster();
6      std::vector<double> getCentroids();
7      std::vector<std::pair<double,size_t>> getAssignments();
8
9  private:
10     arma::mat data_;
11     size_t clusters_;
12     arma::Row<size_t> assignments_;
13     mlpack::kmeans::KMeans<> kmeans_;
14     arma::mat centroids_;
15 };
```

Figure 3.18: *K_means Class*

Before contour points can be searched along radial directions from tumor centroids as detailed in figure 2.7 k-means clustering algorithm is employed to distinguish stiffness measurements belonging to tumorous tissue from surrounding non-tumorous regions. For this, all stiffness values of observed data are collected (figure 3.19, lines 5-8) and separated into two categories.

```
void Contour::labelData_()
{
    //collect all sampled stiffness values from gaussian process
    bayesopt::Dataset* data = const_cast<bayesopt::Dataset*>(bopt_model_->getData());
    for(size_t i = 0; i<total_number_of_iterations_; i++)
    {
        y_values_.push_back(data->getSampleY(i));
    }
    //cluster them to filter stiffness values associated with tumor
    K_means kmeans(y_values_, tumor_stiffness_guess_low_,tumor_stiffness_guess_high_);
    kmeans.cluster();
    //identify label for centroid with lowest stiffness values
    std::vector<double> c = kmeans.getCentroids();
    //Assumes that centroid index corresponds to label
    std::vector<double>::iterator min_idx_it = std::min_element(c.begin(), c.end());
    size_t min_idx = std::distance(c.begin(), min_idx_it);
    //retrieve cluster center of tumor stiffness
    tumor_stiffness_mean_ = c[min_idx];
    std::vector<std::pair<double,size_t>> labels = kmeans.getAssignments();
    for(auto &a : labels)
    {
        if(a.second == min_idx)
        {
            tumor_stiffness_vec_.push_back(a.first);
        }
    }
}
```

Figure 3.19: *labelData_* member function of *Contour Class*. Clusters stiffness measurements, tumor stiffness values and non tumor stiffness values. Stores measurements belonging to tumor in *tumor_stiffness_vec_* mean value of tumor measurements in *tumor_stiffness_vec_*.

In a next step, the mean value of measurements and the measurements itself belonging to tumor tissue are stored in *tumor_stiffness_vec_* and *tumor_stiffness_vec_*, respectively. Both are used to compute a threshold value as can be seen in figure 3.20 that is used to decide whether probes collected during the search for contour points belong to tumor tissue or surrounding tissue.

```
void Contour::computeStiffnessThreshold_()
{
    threshold_ = multiplier_*stdDev(tumor_stiffness_vec_)+tumor_stiffness_mean_;
}
```

Figure 3.20: Computation of threshold value for contour point search.

Once a threshold was successfully determined the algorithm begins by exploring cluster centers in radial directions as described in section 2.5. The related code is presented in figure 3.21 . An outer loop iterates over all clusters that were computed earlier (figure 3.21, lines 5-39). For each cluster radial exploration directions are calculated by dividing 360° by the the number of directions we want to explore (figure 3.21, line 7). Next, starting from the center itself each direction is incrementally explored (figure 3.21, lines 20-25) and measurements are taken until a limit of steps is reached (figure 3.21, lines 18) or until the measured stiffness value falls below the previously computed threshold (figure 3.21,line 26). The function *conoturPoint_* compares *new_stiffness* against the threshold and returns true if the current measurement is below the threshold. This indicates that a contour point was reached and it is stored inside *contour_vec* (figure 3.21, line 28). Lastly, for each center that is explored, *contour_vec* containing all point coordinates of contour points is collected within the class member variable *contour_* (figure 3.21, line 38).

```
void Contour::exploreContour()
{
    labelData_();
    computeStiffnessThreshold_();
    for(auto &p1 : clusters_)
    {
        double delta_theta = 360 /static_cast<double>(n_directions_);
        double theta = 0.0;
        std::vector<Point> contour_vec{n_directions_+1};

        for (size_t i = 0; i < n_directions_+1; i++)
        {
            theta = delta_theta* i;
            double delta_x = stepsize_ * sin(theta*M_PI/180.0);
            double delta_y = stepsize_ * cos(theta*M_PI/180.0);
            int j = 0;
            Point p = p1;
            while(j < lim_steps_)
            {
                p.x +=  delta_x;
                p.y +=  delta_y;
                vectord q(2);
                q(0) = p.x;
                q(1) = p.y;
                double new_stiffness = bopt_model_->evaluateSample(q);
                if (contourPoint_(new_stiffness))
                {
                    contour_vec[i] = p;
                    j = lim_steps_ +1; //exits while loop
                }
                j++;
                if(j == lim_steps_)
                {
                    std::cout<<"No contour point for theta at "<< theta<<std::endl;
                }
            }
        }
        contours_.push_back(contour_vec);
    }
}
```

Figure 3.21: *exploreContour* function that searches for contour points in radial directions from tumor centers.

Contour Approximation

The contour curve is approximated by a parametric cubic spline function as explained in section 2.5. alglib is a powerful numerical analysis and data processing library and its cubic spline interpolation *alglib::spline1dbuildcubic* was used throughout this work. The function *approximateContour* from figure 3.22 separates x and y coordinates and introduces the parameter t that is used to traverse a parametric function in x and y.

$(x(t), y(t))$ describes the parametric contour curve of the tumor. Figure 3.22 lines 5-11 show how a data sets in x $[(t_0, x_0), (t_1, x_1)..(t_n, x_n)]$ and y $[(t_0, y_0), (t_1, y_1)..(t_n, y_n)]$ with $n = n\_samples$ are created for approximation.Figure 3.22 lines 13-21 perform the approximation of $(x(t), y(t))$ and both spline functions are stored as a *alglib::spline1dinterpolant* type. Each centroid results a set of contour points and all contour points for each centroid are kept within the private member *contours_*. Since all contour points belonging to a tumor should be approximated by a parametric curve, the function *approximateContour* performs a spline approximation for each set of contour points by iterating over *contours_*. The results of contour curve approximations for all tumors are stored within *spline_ contours_* .

```
1  void Contour::approximateContour() {
2      int file_num = 1;
3
4      for (const auto& contour : contours_) {
5          std::vector<double> t(n_samples_), f_1(n_samples_), f_2(n_samples_);
6
7          for (size_t i = 0; i < n_samples_; ++i) {
8              t[i] = static_cast<double>(i) / (n_samples_ - 1);
9              f_1[i] = contour[i].x;
10             f_2[i] = contour[i].y;
11         }
12
13         alglib::real_1d_array alg_t, alg_x, alg_y;
14         alg_t.setcontent(t.size(), t.data());
15         alg_x.setcontent(f_1.size(), f_1.data());
16         alg_y.setcontent(f_2.size(), f_2.data());
17
18         alglib::spline1dinterpolant s1, s2;
19         alglib::spline1dbuildcubic(alg_t, alg_x, s1);
20         alglib::spline1dbuildcubic(alg_t, alg_y, s2);
21         spline_contours_.emplace_back(std::make_shared<alglib::spline1dinterpolant>(s1),
22                 std::make_shared<alglib::spline1dinterpolant>(s2));
23
24     }
25 }
```

Figure 3.22: Cubic spline approximation of contour points.

3.3. Parameters and Parameter Tuning

Special attention was placed on parameter handling. The tumor localization and segmentation algorithm has three sets of parameters and adjusting them from within the code is cumbersome. For this reason, all parameters required to run the algorithm were extracted from the code and placed inside text files within a dedicated directory. Adjustments to

the behaviour of components can be done without the need for recompilation.

To achieve this parameters are organized inside config/ directory in three different files (figure, 3.23).

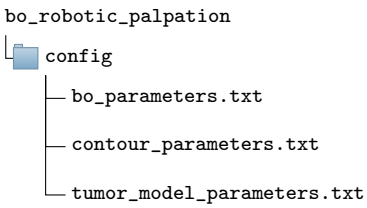

Figure 3.23: Parameter files directory structure

A struct called *TumorModelParameters* appended to this document A.3 stores all parameters necessary to define a tumor finding problem that the algorithm can be tested on. It has methods to read and save parameters from tumor_model_parameters.txt. Each tumor model is defined by the following parameters:

1. <Shape>_low and <Shape>_high define low and high stiffness values.
2. <Shape>_radius either the radius for circular shape or the outline on which vertices are placed for polygons.
3. <Shape>_x_trans and <Shape>_y_trans determine translation of origin of shape defining circle.
4. <Shape>_epsilon is added to <Shape>_radius and defines the size of the region that is interpolated by the smooth step.
5. <Shape>_noise steers the level of noise present in measurements. Here the parameter determines the standard deviation of Gaussian noise that is added to each observation.

contour_paramters.txt are parsed and saved by *struct ContourParameters*. Its implementation is attached to this report in appendix A.3. It offers the same methods as *struct TumorModelParameters* but it is responsible to hold parameters that control the behaviour of the tumor localization and segmentation algorithm.

1. n_exploration_directions defines the amount of contour points that are searched.
2. c_points sets granularity of Gaussian process.
3. means_shift_bandwidth controls the bandwidth for mean shift bandwidth described above.

4. lim_steps number of measurements that are taken along each exploration direction in the contour search. The search will terminate after the limit of step is reached and no contour point is found.
5. stepsize will adjust the size of each step for the contour point search.
6. threshold_multiplier determines the stiffness threshold. To do so, all previously seen sample stiffness observations are clustered by a k-means algorithm. Given that high stiffness values correlate with tumorous tissue, the cluster with the highest values is selected and its mean value and uncertainty calculated. Based on these measurements, the threshold_multiplier parameter will define how many standard deviations from the mean value of the tumor stiffness cluster are categorized as tumor tissue.
7. tumor_stiffness_guess_low and tumor_stiffness_guess_high

Finally, in order to tune the behaviour of the Bayesian optimization, relevant parameters are placed in bo_parameters.txt. A detailed explanation of all possible optimization strategies is given in the official bayesopt library documentation. Here only the parameters used for this work will be explained:

1. n_iterations defines the amount of samples that are taken.
2. n_inner_iterations gives the maximum number of iterations per dimension to optimize the acquisition function (criteria).
3. n_init_ depicts samples number of samples that are computed by initialization routine.
4. init_method method that is used for finding initial samples, in this work uniform sampling was leveraged.
5. random_seed fixes probabilistic behaviours of algorithm to ensure reproducible.
6. surr_name name of surrogate model that is used to perform update steps on. Here sGaussianProcess was selected.
7. sigma_s signal variance
8. noise noise level for stochastic functions, where multiple evaluations of the same point yield different outcomes, the noise-to-signal ratio should closely align with the noise variance relative to the signal variance. Excessive noise can lead to slow convergence, whereas insufficient noise may prevent convergence altogether.

9. l_type defines the learning type of hyperparametes. In this work hyperparameters are kept constant and therefore the parameters is set to L_FIXED
10. epsilon Random sample strategy. Probability value between 0 and 1 for the sampling policy to place a random sample. High values correlate with a forced exploration whereas low values imply that the algorithm leans toward the exploration/exploitation strategy imposed by the criterion (expected improvement).
11. kernel.name sets the type of kernel used for the Gaussian process. In this context a Gaussian kernel (kSEISO) function was selected.
12. kernel.hp_mean array of values defining means of kernels. This parameter is referred to as length scale and inputs are converted to log scale.
13. kernel.hp_std array of values defining standard deviation of kernels.
14. mean.name name of prior mean function. Here a constant mean function was selected (mConst).
15. mean.coef_mean value of mean function.
16. crit_name name of acquisition function. As described above, expected improvement is used here which is set with cEI.
17. crit_params controls trade-off between exploration and exploitation of the acquisition function.

3.4. Visualization

A target called *display_gp*, as described above offers the possibility to monitor the state of the Gaussian process for each iteration. Further, it visualizes the computed tumor center, the respective contour points and approximated contour curves. The program is embedded within matplotpp's visualization framework. Matplot++ is a C++ graphics library for data visualization that is inspired by MATLAB's plotting interface and Matplotlib, a popular Python plotting library. To interface Matplot++'s plotting capabilities and combine them with the tumor segmentation algorithm, *class DisplayHeatMap2D* illustrated in figure 3.24 was conceptualized.

```
enum RunningStatus
{
        RUN, STEP, STOP, NOT_READY, PLOT_CENTROIDS, PLOT_CONTOUR_POINTS, PLOT_CONTOUR_APPX
};

    class DisplayHeatMap2D :public MatPlot
    {
    private:

      RunningStatus status; //enum to set current state
      size_t state_ii; // current iteration
      Contour *contour_; // pointer to contour class object
      std::vector<double> lx,ly; //last point
      std::vector<double> cx, cy; //temp variable to store last point
      std::vector<double> cX,cY;
      std::vector<std::vector<double>> z_, c_, std_; //containers for posterior, eval criterion, standard deviation
      size_t c_points; //c_points x c_points defines size of domain computation is performed on
      std::vector<std::vector<double> > cZ; //container to store ground truth
      double gTHigh, gTLow, PHigh, PLow, StdHigh, StdLow, CVHigh, CVLow;
      void setHighLow(double &low, double &high, double val);
      //Cluster plot variables
      std::vector<double> clusterx_, clsutery_;
      bool computeClusters;
      //Contour points plot variables
      bool computeContourPoints;
      std::vector<double> cpointsx_, cpointsy_;
      //Contour approximation plot variables
      bool computeSplinePoints;
      std::vector<std::vector<double>> splinex_, spliney_;
    public:
      DisplayHeatMap2D();
      void loadGroundTruth();
      void init(Contour *contour,size_t dim);
      void setSTEP();
      void toogleRUN();
      void updateData();
      void plotData();
      void DISPLAY();
    };
```

Figure 3.24: *class DisplayHeatMap2D*

The class implements a state machine similar to the one presented in figure 3.10. The state machine is implemented within *DISPLAY* which itself is a virtual function defined in the *MatPlot* parent class. The function is continuously called until the program is terminated. Figure 3.25 depicts *DISPLAY* and it can be seen that the application is continuously updating data in a first step and plotting data in a second step. Inside *updateData* functions from *Contour class* are utilized to perform the following steps:

1. Do an optimization step, collect data from posterior, standard deviation, acquisition

functions and current sample.

2. Run tumor localization algorithm and save computed centroids
3. Perform contour points search and store all computed contour points
4. Approximate contour points with cubic spline and store results.

After updating of data was done the application proceeds by calling *plotData* which is responsible for plotting the acquired data with Matplot++'s plotting utilities.

```
1 void DisplayHeatMap2D::DISPLAY()
2 {
3     if (status != NOT_READY)
4     {
5         updateData();
6         plotData();
7     }
8 }
```

Figure 3.25: *DISPLAY* function

3.5. Evaluation

Part of the project is the target *evaluate* which compares approximated tumor contour curves against ground truths. Yan and Pan [24] present *sensitivity* and *specificity* metrics to reason about the quality of a segmented tumor contours. The *sensitivity* metric depicted in equation 3.1 defines the proportion of true positives that are found and *specificity* given in equation 3.2 calculates the proportion of true negatives that are correctly identified.

$$\text{Sensitivity} = \frac{\text{Area of True Positive}}{\text{Area of True Positive}+\text{Area of False Negative}} \tag{3.1}$$

$$\text{Specificity} = \frac{\text{Area of True Negative}}{\text{Area of True Negative}+\text{Area of False Positive}} \tag{3.2}$$

Figure 3.26 illustrates true negative, true positive, false negative and false positive sets and how each is defined.

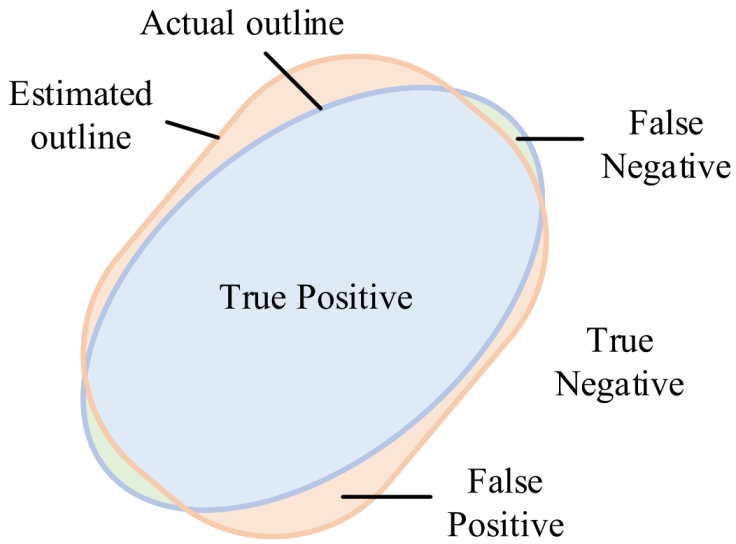

Figure 3.26: Visualization of approximated tumor contour (blue), actual tumor contour (orange) and evaluation metrics used to compute sensitivity and specificity.Illustration source: [24]

The computation of *sensitivity* and *specificity* metrics is stemmed by *class ContourPairAnalyzer* shown in figure 3.27. The class is designed to compare an approximated contour curve against a ground truth curve. Methods *computeFalseNegative*, *computeFalsePositive*, *computeTruePositive* and *computeTrueNegative* compute the required areas to determine *sensitivity* and *specificity* metrics. The underlying mathematical operations are performed by *computeDifferenceArea*, *computeJoinArea*, *computeIntersectArea* and *computeArea* functions which internally exploit CGAL's Polygon_with_holes_2 and Polygon_2 objects to perform calculations.

```
class ContourPairAnalyzer
{
private:
    SplineInterpolant_ptr_pair f_parametric_A_,f_parametric_B_; //parametric Spline functions of Contour A and B
    double domain_area_;
    std::string experiment_path_;
public:
    ContourPairAnalyzer(SplineInterpolant_ptr_pair &f_parametric_A,
                        SplineInterpolant_ptr_pair &f_parametric_B,
                        double domain_area, std::string experiment_path_);
    void analyzeContours(const size_t contour_number);
    bool polygonizeSpline( SplineInterpolant_ptr_pair &spline_pair, Polygon_2 &P,size_t num_vertices);
    double computeDifferenceArea(Polygon_2 &A, Polygon_2 &B, bool verbose);
    double computeJoinArea(Polygon_2 &A, Polygon_2 &B, bool verbose);
    double computeIntersectArea(Polygon_2 &A, Polygon_2 &B, bool verbose);
    double computeFalseNegative(Polygon_2 &GroundTruth, Polygon_2 &Approximation);
    double computeFalsePositive(Polygon_2 &GroundTruth, Polygon_2 &Approximation);
    double computeTruePositive(Polygon_2 &GroundTruth, Polygon_2 &Approximation);
    double computeTrueNegative(double domainArea, Polygon_2 &GroundTruth, Polygon_2 &Approximation);
    double computeArea(Polygon_2 &A);

    double computeSpecificity(double domainArea, Polygon_2 &GroundTruth, Polygon_2 &Approximation);
    double computeSensitivity(Polygon_2 &GroundTruth, Polygon_2 &Approximation);
    ~ContourPairAnalyzer();
};
```

Figure 3.27: *Class ContourPairAnalyzer*

The class is designed to receive two parametric spline contour curves upon creation of the object which are later converted to CGAL polygon types. A custom type definition was introduced for to store a contour curve as a pair of splines, as can be seen in figure 3.28.

```
typedef std::pair<std::shared_ptr<alglib::spline1dinterpolant>,
                                  std::shared_ptr<alglib::spline1dinterpolant>>
                                  SplineInterpolant_ptr_pair;
```

Figure 3.28: *Class ContourPairAnalyzer*

3.6. Processing Data Organization

Postprocessing and analysis steps require data collection and for this several files are written during the execution of the application. Figure 3.29 shows the output of a single experiment where the localization and segmentation algorithm was run on a Triangle tumor model. Upon launching the simulation, an experiment directory with sub-directories is created inside the /data directory. Inside the experiment directory (here Triangle1) sub

directories /log, /parameters and /results are used to store processing data and parameter configuration.

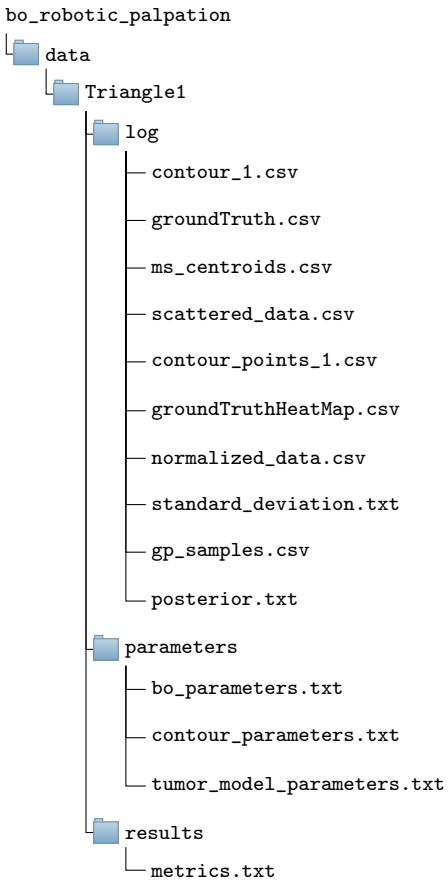

Figure 3.29: data directory with example

The following files are stored inside /log, /parameters, and /results directories:

1. contour_1.csv stores x and y coordinates of the approximated contour.
2. groundTruth.csv saves x and y coordinates of ground truth contour.
3. ms_centroids.csv contains point coordinates of centroids.
4. scattered_data.csv collects sampled posterior distribution.
5. contour_points_1.csv saves computed contour points.
6. groundTruthHeatMap.csv, posterior.txt, normalized_data.csv and standard_deviation.txt contain two dimensional data to display ground truth, final posterior, final posterior normalized and standard deviation, respectively.
7. gp_samples.csv stores all samples that were computed.

8. bo_parameters.txt, contour_parameters.txt and tumor_model_parameters.txt save the parameter configuration used to run the experiment. Detailed information is found in section 3.3 and appendix A.

9. metrics.txt contains specificity and sensitivity metrics.

3.7. Installation, Compilation and Program Execution

The sources for this application are provided with the public Gitlab repositories. Inside the repository, a README.md file is found which contains a detailed description on how to install and run the program. Program compilation and linking of external libraries is needed to obtain an executable. In particular the program depends on bayesopt library, mlpack library, alglib,CGAL library and Matplot++ plotting utilites. The build process is managed by CMake which is a cross-platform and open source build tool for C++ code. Steps required for the compilation of the program are listed in a CMakeLists.txt A.4 which which is parsed and compiled by CMake and an executable is created. Besides the conventional approach of installing libraries locally and compiling the program form sources a Docker image was created that allows rapid deployment of the application. Docker is a set of platfrom as service products which use operating system virtualization to deliver software in so called containers. The containers are portable and encapsulate all the software needed to run them. This means no libraries need to be installed locally. All necessary environment setup steps are listed in a Dockerfile A.5.

Compilation

Compilation on your local machine requires the installation of the the above listed libraries. After successful installation of third party libraries and dependencies the application can be build by running the following commnds.

```
$ git clone https://git.tu-berlin.de/raphael/bo_robotic_palpation
$ cd bo_robotic_palpation/ && mkdir build && cd build
$ cmake ..
$ make
```

Program execution

After successfully building the project with the commands listed above the targets display_gp, evaluate and compute_contours are found within the build directory. The main

function of each of these targets is designed to take an argument along the execution command which selects the experiment that is performed. The application display_gp which offers a graphical interface can be launched by running one of the following commands.

```
$ ./display_gp Triangle
$ ./display_gp Rectangle
$ ./display_gp Circle
$ ./display_gp TwoCircles
```

A window will open displaying the current states of the Gaussian process and a single step can be done by pressing the key 's'. By pressing the key 'r' the whole simulation is run in a continuous fashion. The evaluation target that produces performance metrics is run with the following commands:

```
$ ./evaluate Triangle
$ ./evaluate Rectangle
$ ./evaluate Circle
$ ./evaluate TwoCircles
```

and the target compute_contours which only runs the algorithm without visualization nor evaluation is launch with on of the following prompts:

```
$ ./compute_contour Triangle
$ ./compute_contour Rectangle
$ ./compute_contour Circle
$ ./compute_contour TwoCircles
```

Each target will log process data and results within the /data directory that is found within the repository.

Docker container

Environment setup and installation of libraries can be a cumbersome and time consuming endeavor. For that reason a docker image was created. More precisely the image is created from a Dockerfile A.5 that is given in the /images directory in the project folder. The Dockerfile contains all instructions to setup a container with the correct operating system and it will install all required libraries internally. Further, it compiles the code and executables can later be found inside. Docker offers the possibility to define mount points which are shared folders that allow file exchange and access between docker container and host operating system. The container is defined in a way that the experiment is launched from within the container and data that is produced along the process is written inside the mounted data/ directory. This means that all results are later accessible in the host operating system. The container is set up with the following commands. First the a build/ directory has to be created inside the project directory bo_robotic_palpation/.

Note: the build/ directory needs to be empty. No files from previous build steps can be inside.

```
$ mkdir build
$ export BO_PALPATION_WS=$(pwd)
```

Next the container is build by running:

```
$ sudo docker build -f "$BO_PALPATION_WS"/images/Dockerfile
                    -t bo_robotic_palpation "$BO_PALPATION_WS"
```

and finally the container is accessed in interactive mode with:

```
$ sudo docker run --mount type=bind,source="$BO_PALPATION_WS"/data,
                   target=/usr/src/bo_robotic_palpation/data
                   --mount type=bind,source="$BO_PALPATION_WS"/config,
                   target=/usr/src/bo_robotic_palpation/config
                   -it -t  bo_robotic_palpation
```

Once a command prompt opened inside the container, the same execution commands as listed above can be used to run the program. Important: At this development stage, no graphical interface is provided within the docker container and therefor only the targets evaluate and compute_contour can be launched.

4 | Experiments

Multiple experiments were run to demonstrate the effectiveness of the localization and segmentation algorithm. Computations were done on a standard laptop with the following specifications:

- Chip: 11th Gen Intel® Core™ i7-1165G7 @ 2.80GHz with 8 cores
- Memory: 16GB
- Operating system: Ubuntu 20.04.6 LTS

Experiments were simulated with four different tumor models to test different aspects of the algorithm:

1. Triangle, tests the ability to deal with acute angles in tumor shapes.
2. Rectangle, tests the ability to approximate a more complicated contour curve.
3. Circle, demonstrates the ability to approximate a smooth contour curve
4. Two circles, to test whether the algorithm is capable to detect two tumors

Stiffness values in the underlying tumor models were adjusted according to the work done by Ishihara et al. [8] which suggests that tumorous tissue posses a stiffness of 4.0kPa and unaffected regions a stiffness of 0.2kPa. The kernel variance σ_s was set to 1 as suggested in the literature [11],[24]. Ideal results were found for a length scale $l = log(-2)$ after numerous trials. A good balance between exploitation and exploration in the computation of the sample for the next iteration was given for $\epsilon = 0.5$. The signal to noise ratio denoted σ_n was set to match the noise level present in the tumor model which is $\sigma_s = 0.001$. The localization was successful for a mean shift bandwidth of 0.4 for all experiments and in total the algorithm searched for 10 tumor contour points which were subsequently used to approximate a contour curve.

4.1. Clustering of Tumor Errors

The optimal number of samples that is required to perform a successful localization was determined by taking the localization error d and plotting it over the number of samples taken. The localization error d was computed by taking the euclidean distance between true tumor center and approximated tumor center. Later a visual inspection yielded that the optimal number of iterations required is in the range of 25 to 30 samples. Figure 4.1 shows this correlation. The clustering error was calculated over the number of observations taken.

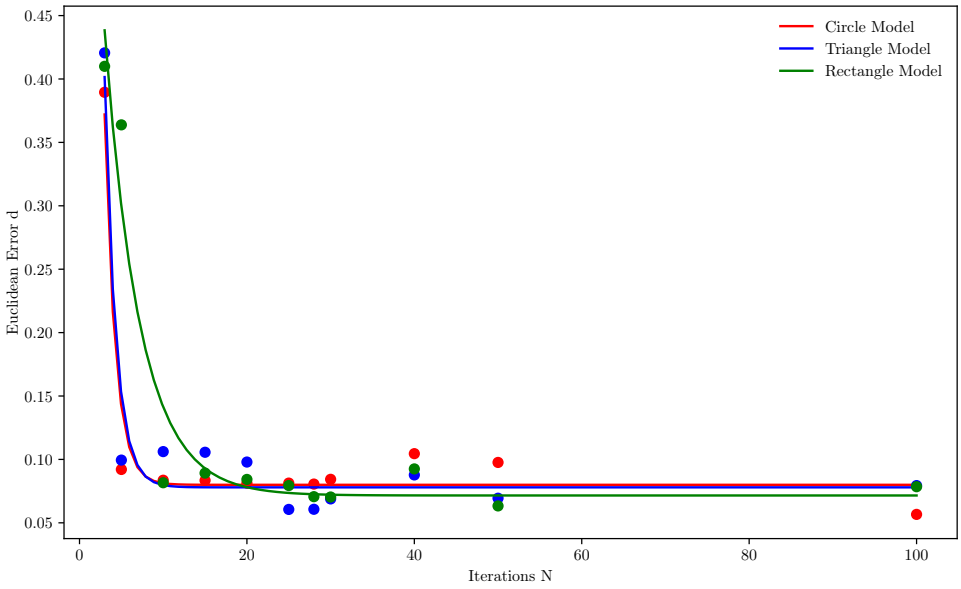

Figure 4.1: Euclidean distance d between true tumor centers and detected tumor centers over number of number of samples taken for circular, rectangular and triangular tumor models. As expected the error is decreasing for increasing number of interactions N

4.2. Segmentation of Tumor

The capacities of the proposed algorithm were tested on four different tumor models. Experiments were done for a circular (Circle) ,triangular (Triangle), rectangular (Rectangle) shaped tumor model as well as tumor model with two circular shapes (TwoCircles). All Parameters were kept constant throughout the experiments to demonstrate the robustness of the algorithm. As identified in previous section 4.1 the optimal range of iterations lies between 25 and 30 and for the following experiments the iteration count was limited

to 30. The parameter setup to reproduce the results can be found in the appendix A.2. Sensitivity and specificity performance metrics from section 3.5 were used to measure the accuracy of the produced contour curves.

Circle

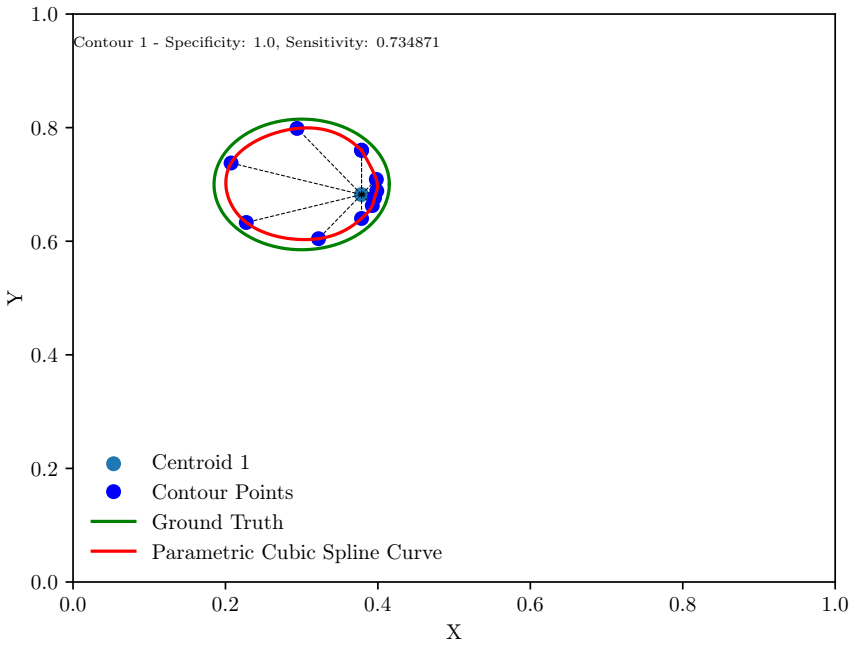

Figure 4.2: Comparison of Approximated contour and ground truth tumor.

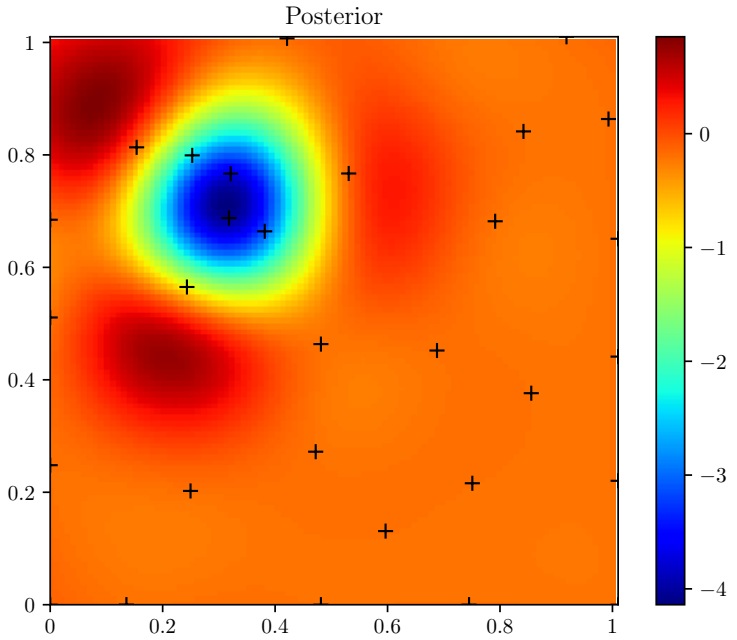

Figure 4.3: Posterior distribution after 30 iterations.

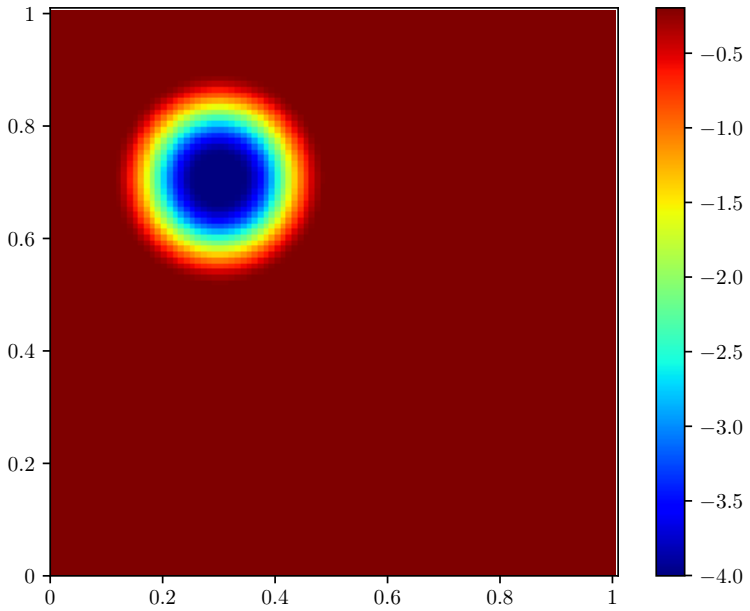

Figure 4.4: Tumor model used for localization and segmentation. Stiffness values are inverted to formulate the optimization as a minimization problem.

Triangle

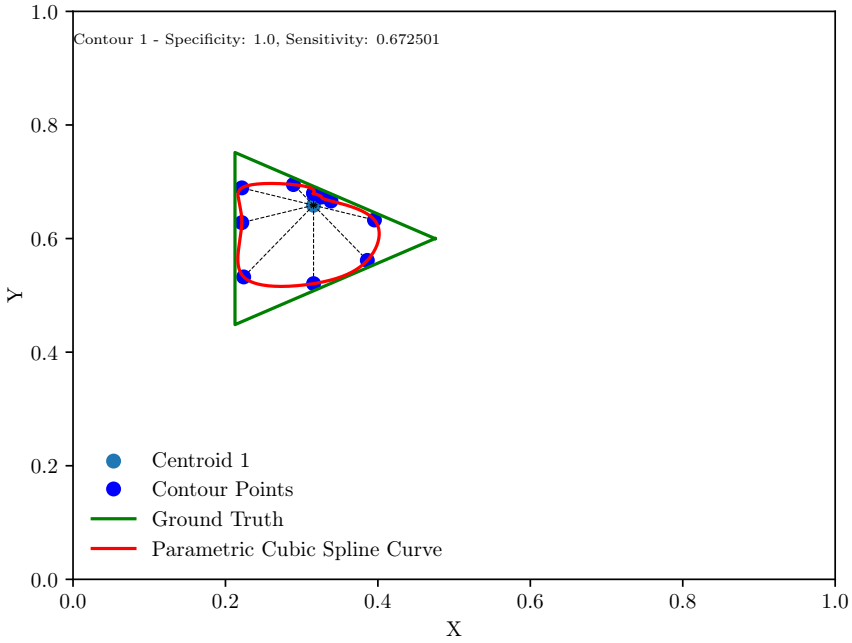

Figure 4.5: Comparison of Approximated contour and ground truth tumor.

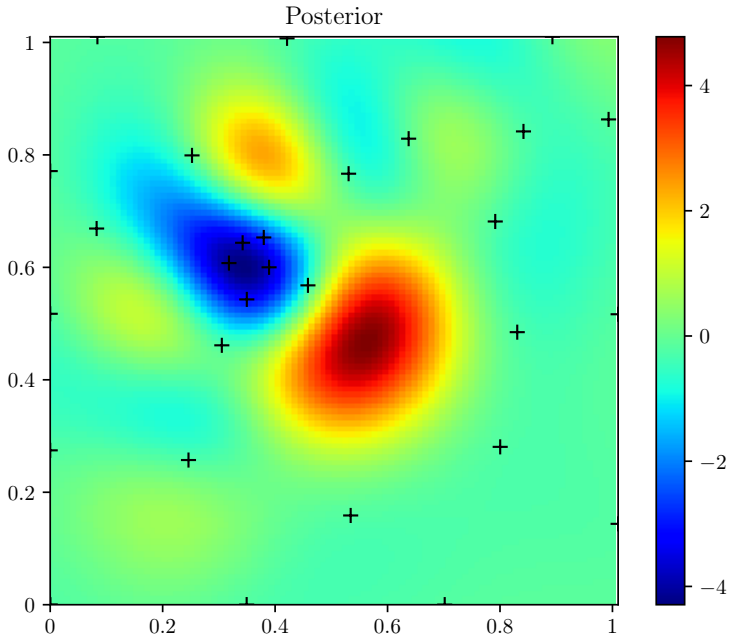

Figure 4.6: Posterior distribution after 30 iterations.

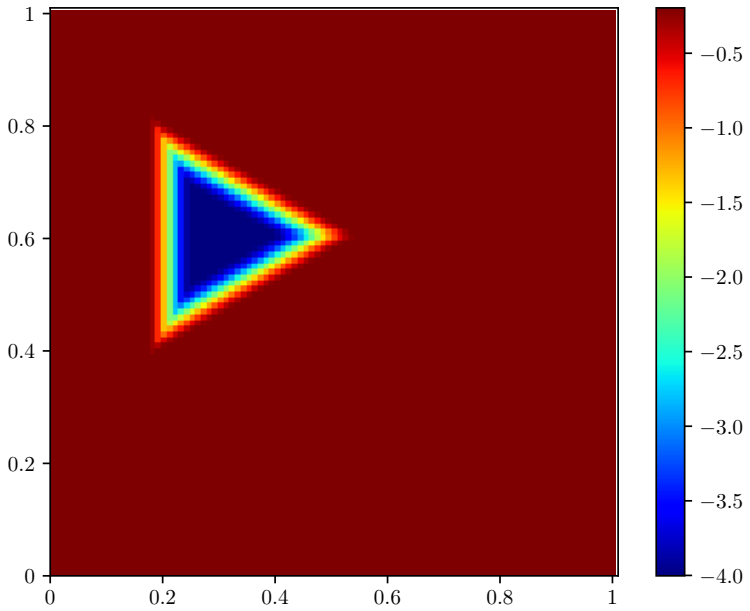

Figure 4.7: Tumor model used for localization and segmentation. Stiffness values are inverted to formulate the optimization as a minimization problem.

Rectangle

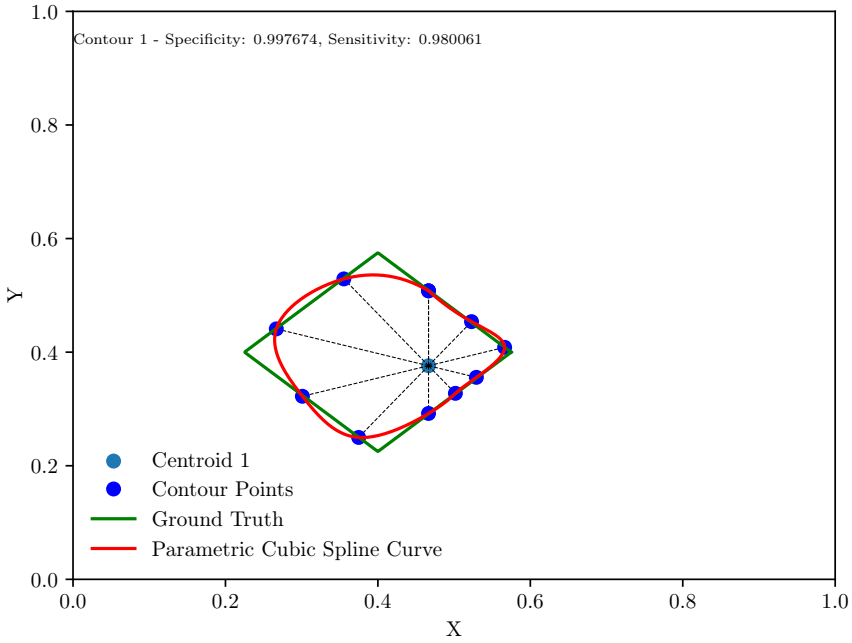

Figure 4.8: Comparison of Approximated contour and ground truth tumor.

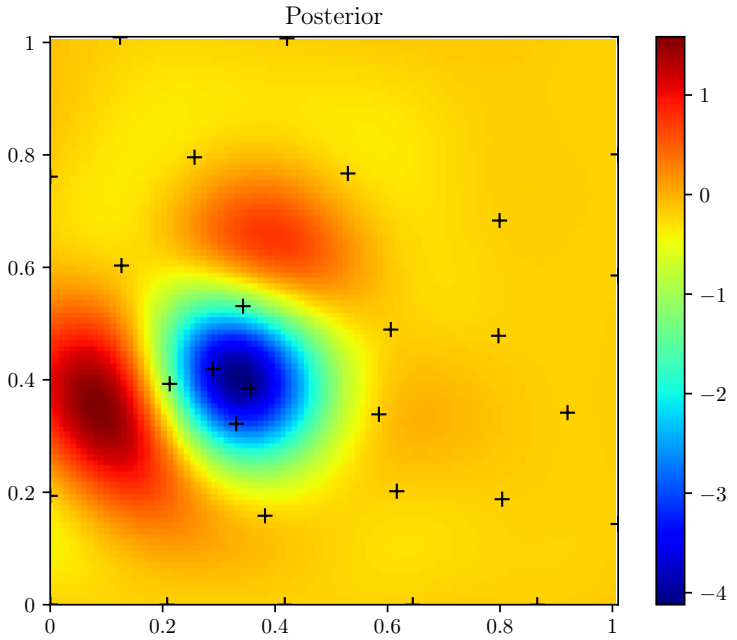

Figure 4.9: Posterior distribution after 30 iterations.

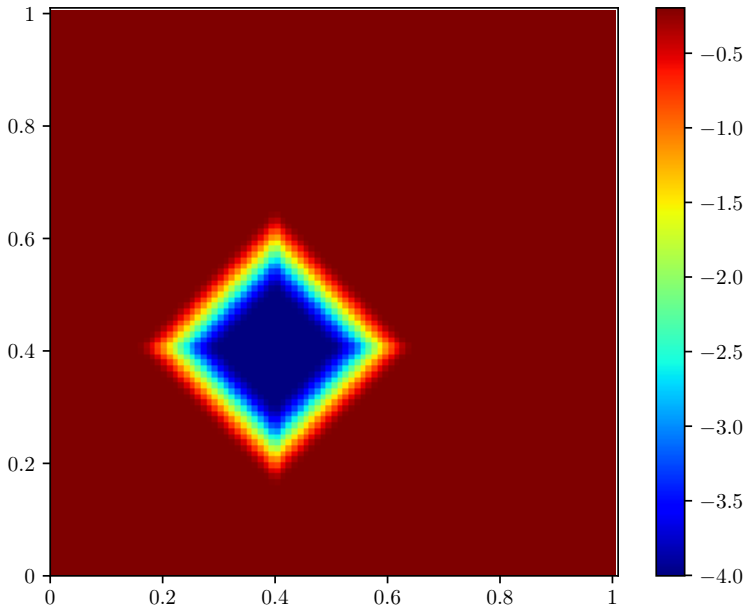

Figure 4.10: Tumor model used for localization and segmentation. Stiffness values are inverted to formulate the optimization as a minimization problem.

Two Circles

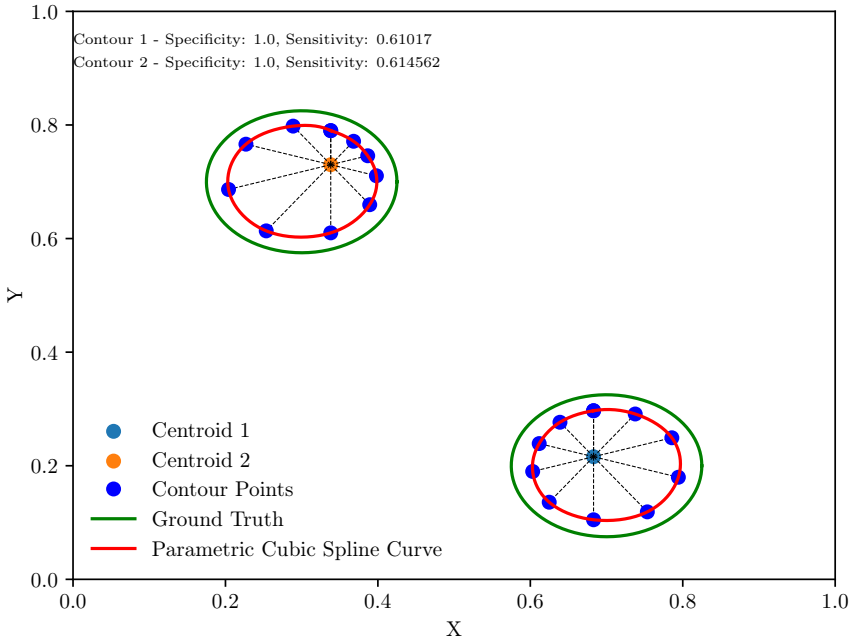

Figure 4.11: Comparison of Approximated contour and ground truth tumor.

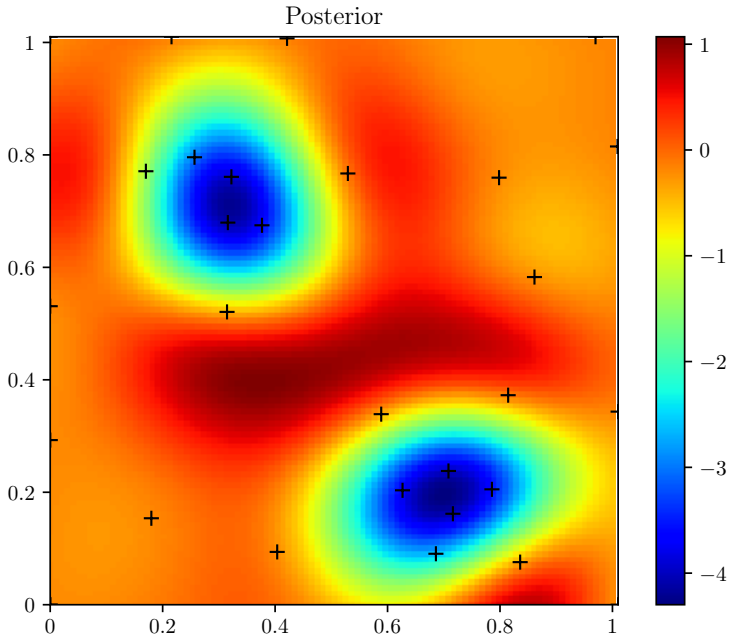

Figure 4.12: Posterior distribution after 30 iterations.

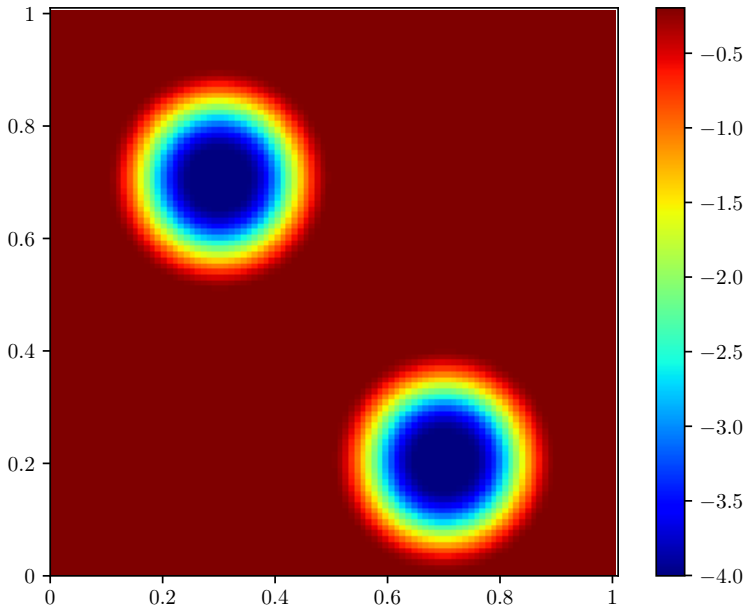

Figure 4.13: Tumor model used for localization and segmentation. Stiffness values are inverted to formulate the optimization as a minimization problem.

Experiment	Sensitivity	Specificity
Circle	0.734871	1
Triangle	0.672501	1
Rectangle	0.980061	0.997674
TwoCircles	0.61017 / 0.614562	1 / 1

Figure 4.14: Sensitivity and Specificity for Circle, Triangle, Rectangle and TwoCircles experiments

5 | Conclusion and Future Developments

In this project an algorithm was implemented that successfully localizes and segments differently shaped tumors from a tumor model. Not more than 30 observations are required to localize the tumor and segmentation is done with a contour point search that finds 10 contour points. Figures 4.2, 4.5, 4.8 and 4.11 show the approximated contour curve displayed in red and the ground truth contour marked in green. Made observations and the final posterior distributions are depicted in heat-map plots 4.3, 4.6, 4.9 and 4.12. Discovered contour points and tumor centroids are graphed in blue and light blue, respectively. Results listed in table 4.14 give evidence to evidence to high values for specificity for all shapes. This indicates that false positive areas are either zero or close to zero which means that most of the approximated contour curve is contained within the ground truth contour. Sensitivity values on the other hand range from 0.61 for TwoCircles experiment to 0.98 for Rectangle experiment hinting on the fact that some approximations fall behind in covering the full surface of the tumor. As can be seen in the final posterior distributions, tumorous regions are correctly captured but sometimes the computed tumor centers do not align with the minimum value of the underlying mode. Essentially, this means that the mean shift clustering method is not precisely registering extremal points from the posterior distribution. The behaviour of mean shift is controlled by the bandwidth parameter and its value is critical for how many clusters are detected and how precise the tumor localization will be. In this work, while tuning the bandwidth emphasis was placed on detecting the correct number of tumors. Larger values correlated with the successful identification of the number of tumors. For a smaller bandwidth parameter some of the detected clusters were incorrect but the for the correctly identified tumor centers showed better accuracy. In a future work, a strategy could be implemented that compares the stiffness values in detected clusters and filters them. A key factor influencing the quality of the contour curve is the number of contour points that are searched for. While evident that more contour points lead to better approximation of the tumor model the number was kept at 10 to reduce the total number of queries.

Bibliography

[1] A. A. Abushagur, N. Arsad, M. Ibne Reaz, A. Ashrif, and A. Bakar. Advances in bio-tactile sensors for minimally invasive surgery using the fibre bragg grating force sensor technique: A survey. *Sensors*, 14(4):6633–6665, 2014.

[2] E. Ayvali, R. A. Srivatsan, L. Wang, R. Roy, N. Simaan, and H. Choset. Using bayesian optimization to guide probing of a flexible environment for simultaneous registration and stiffness mapping. In *2016 IEEE International Conference on Robotics and Automation (ICRA)*, pages 931–936. IEEE, 2016.

[3] J. Back, P. Dasgupta, L. Seneviratne, K. Althoefer, and H. Liu. Feasibility study-novel optical soft tactile array sensing for minimally invasive surgery. In *2015 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 1528–1533. IEEE, 2015.

[4] E. Brochu, V. M. Cora, and N. De Freitas. A tutorial on bayesian optimization of expensive cost functions, with application to active user modeling and hierarchical reinforcement learning. *arXiv preprint arXiv:1012.2599*, 2010.

[5] D. Comaniciu and P. Meer. Mean shift: a robust approach toward feature space analysis. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 24(5): 603–619, 2002. doi: 10.1109/34.1000236.

[6] A. Garg, S. Sen, R. Kapadia, Y. Jen, S. McKinley, L. Miller, and K. Goldberg. Tumor localization using automated palpation with gaussian process adaptive sampling. In *2016 IEEE International Conference on Automation Science and Engineering (CASE)*, pages 194–200. IEEE, 2016.

[7] R. Garnett. *Bayesian optimization*. Cambridge University Press, 2023.

[8] S. Ishihara and H. Haga. Matrix stiffness contributes to cancer progression by regulating transcription factors. *Cancers*, 14(4):1049, 2022.

[9] J. R. Joseph, B. W. Smith, X. Liu, and P. Park. Current applications of robotics in

spine surgery: a systematic review of the literature. *Neurosurgical focus*, 42(5):E2, 2017.

[10] D. J. Lizotte. Practical bayesian optimization. 2008.

[11] R. Martinez-Cantin. BayesOpt: A Bayesian optimization library . http://rmcantin.github.io/bayesopt/html/boptttheory.html#modbopt, 2011-2020. [Online; accessed 6-February-2024].

[12] R. Martinez-Cantin. Bayesopt: a bayesian optimization library for nonlinear optimization, experimental design and bandits. *J. Mach. Learn. Res.*, 15(1):3735–3739, 2014.

[13] J. Mockus. Application of bayesian approach to numerical methods of global and stochastic optimization. *Journal of Global Optimization*, 4:347–365, 1994.

[14] A. M. Okamura, L. N. Verner, C. E. Reiley, and M. Mahvash. Haptics for robot-assisted minimally invasive surgery. In *Robotics Research: The 13th International Symposium ISRR*, pages 361–372. Springer, 2011.

[15] J. Peirs, J. Clijnen, D. Reynaerts, H. Van Brussel, P. Herijgers, B. Corteville, and S. Boone. A micro optical force sensor for force feedback during minimally invasive robotic surgery. *Sensors and Actuators A: Physical*, 115(2-3):447–455, 2004.

[16] M. Plodinec, M. Loparic, C. A. Monnier, E. C. Obermann, R. Zanetti-Dallenbach, P. Oertle, J. T. Hyotyla, U. Aebi, M. Bentires-Alj, R. Y. Lim, et al. The nanomechanical signature of breast cancer. *Nature nanotechnology*, 7(11):757–765, 2012.

[17] C. E. Rasmussen and C. Williams. Gaussian processes for machine learning the mit press. *Cambridge, MA*, 32:68, 2006.

[18] H. Salman, E. Ayvali, R. A. Srivatsan, Y. Ma, N. Zevallos, R. Yasin, L. Wang, N. Simaan, and H. Choset. Trajectory-optimized sensing for active search of tissue abnormalities in robotic surgery. In *2018 IEEE International Conference on Robotics and Automation (ICRA)*, pages 5356–5363. IEEE, 2018.

[19] A. Shafikov, T. Tsoy, R. Lavrenov, E. Magid, H. Li, E. Maslak, and N. Schiefermeier-Mach. Medical palpation autonomous robotic system modeling and simulation in ros/gazebo. In *2020 13th International Conference on Developments in eSystems Engineering (DeSE)*, pages 200–205. IEEE, 2020.

[20] R. Tozzi, C. Köhler, A. Ferrara, and A. Schneider. Laparoscopic treatment of early

ovarian cancer: surgical and survival outcomes. *Gynecologic oncology*, 93(1):199–203, 2004.

[21] A. L. Trejos, J. Jayender, M. Perri, M. D. Naish, R. V. Patel, and R. Malthaner. Robot-assisted tactile sensing for minimally invasive tumor localization. *The International Journal of Robotics Research*, 28(9):1118–1133, 2009.

[22] P. Wellman, R. D. Howe, E. Dalton, and K. A. Kern. Breast tissue stiffness in compression is correlated to histological diagnosis. *Harvard BioRobotics Laboratory Technical Report*, 1, 1999.

[23] T. Yamamoto, B. Vagvolgyi, K. Balaji, L. L. Whitcomb, and A. M. Okamura. Tissue property estimation and graphical display for teleoperated robot-assisted surgery. In *2009 IEEE International Conference on Robotics and Automation*, pages 4239–4245. IEEE, 2009.

[24] Y. Yan and J. Pan. Fast localization and segmentation of tissue abnormalities by autonomous robotic palpation. *IEEE Robotics and Automation Letters*, 6(2):1707–1714, 2021.

A | Appendix

A.1. Polar Straight Line

The polar representation of a straight line segment between two points is given by:

$$\begin{aligned} y &= mx + b \\ x &= rcos(\theta) \\ y &= rsin(\theta) \\ r &= -\frac{b}{mcos(\theta) - sin(\theta)} \end{aligned} \tag{A.1}$$

In the next step a straight line is constructed that connects two points $\mathbf{P}_i = (x_i, y_i)$ and $\mathbf{P}_{i+1} = (x_{i+1}, y_{i+1})$

$$\begin{aligned} \mathbf{P}_i : y_i &= mx_i + b \\ \mathbf{P}_{i+1} : y_{i+1} &= mx_{i+1} + b \\ b &= y_i - mx_i \\ y_2 &= mx_{i+1} + y_i - mx_i = m(x_{i+1} - x_i) + y_i \\ m &= \frac{y_{i+1} - y_i}{x_{i+1} - x_i} \\ b &= y_i - x_i \frac{y_{i+1} - y_i}{x_{i+1} - x_i} \\ r(\theta) &= -\frac{\frac{y_i x_{i+1} - x_i y_{i+1}}{x_{i+1} - x_i}}{\frac{y_{i+1} - y_i}{x_{i+1} - x_i} cos(\theta) - sin(\theta)} \end{aligned} \tag{A.2}$$

A.2. Parameter setup

```
n_iterations=28
n_inner_iterations=500
n_init_samples=2
n_iter_relearn=0
init_method=0
random_seed=1
verbose_level=1
log_filename=
load_save_flag=2
load_filename=
save_filename=
surr_name=sGaussianProcess
sigma_s=1
noise=0.001
alpha=0
beta=0
sc_type=
l_type=L_FIXED
l_all=0
epsilon=0.0
force_jump=0
kernel.name=kSEISO
kernel.hp_mean=-2
kernel.hp_std=1
mean.name=mConst
mean.coef_mean=[1](0)
mean.coef_std=[0]()
crit_name=cBEI
crit_params=[2](1,0.5)

```

Figure A.1: bo_paramerts.txt file that contains Bayesian optimization parameters.

```
n_exploration_directions=10
c_points=100
means_shift_bandwidth=0.40
lim_steps=1000
threshold_multiplier=1
tumor_stiffness_guess_low=1
tumor_stiffness_guess_high=2
n_exploration_stepsize=0.003
```

Figure A.2: contour_parameters.txt file

```
triangle_low=0.2
triangle_high=4
triangle_radius=0.1
triangle_x_trans=0.3
triangle_y_trans=0.6
triangle_epsilon=0.15
triangle_noise=0.001
rectangle_low=0.2
rectangle_high=4
rectangle_radius=0.1
rectangle_x_trans=0.4
rectangle_y_trans=0.4
rectangle_epsilon=0.15
rectangle_noise=0.001
circle_low=0.2
circle_high=4
circle_radius=0.04
circle_x_trans=0.3
circle_y_trans=0.7
circle_epsilon=0.15
circle_noise=0.001
two_circles_low=0.2
two_circles_high=4
two_circles_radius_1=0.05
two_circles_radius_2=0.05
two_circles_x_trans_1=0.7
two_circles_x_trans_2=0.3
two_circles_y_trans_1=0.2
two_circles_y_trans_2=0.7
two_circles_epsilon=0.15
two_circles_noise=0.001
```

Figure A.3: tumor_model_parameters.txt file

A.3. Parameter classes

```
struct TumorModelParameters
{
    //Triangle Parameters
    double triangle_low=1;
    double triangle_high=2;
    double triangle_radius=0.1;
    double triangle_x_trans=0.5;
    double triangle_y_trans=0.5;
    double triangle_epsilon=0.1;
    double triangle_noise = 0.01;
    //Rectangle Parameters
    double rectangle_low=1;
    double rectangle_high=2;
    double rectangle_radius=0.15;
    double rectangle_x_trans=0.5;
    double rectangle_y_trans=0.5;
    double rectangle_epsilon=0.2;
    double rectangle_noise = 0.01;
    //Circle Parameters
    double circle_low=1;
    double circle_high=2;
    double circle_radius=0.1;
    double circle_x_trans=0.5;
    double circle_y_trans=0.5;
    double circle_epsilon=0.1;
    double circle_noise = 0.01;
    //Two Circle Parameters
    double two_circles_low=1;
    double two_circles_high=2;
    double two_circles_radius_1=0.05;
    double two_circles_radius_2=0.1;
    double two_circles_x_trans_1=0.1;
    double two_circles_x_trans_2=0.2;
    double two_circles_y_trans_1=0.7;
    double two_circles_y_trans_2=0.8;
    double two_circles_epsilon=0.1;
    double two_circles_noise=0.01;
    load
    void loadModel(bayesopt::utils::FileParser &fp, TumorModelParameters &cp);
    bool loadModelParameters(std::string filename, TumorModelParameters &cp);
    void printParameters();
};


```

Figure A.4: *TumorModelParameters struct* that holds tumor model parameters. It sets default values in case no parameters are loaded. *loadModelParameters* will parse tumor_model_parameters.txt and save parameters.

```
struct ContourParameters{
    size_t n_exploration_directions=10;
    size_t c_points=100;
    double means_shift_bandwidth=0.05;
    size_t lim_steps=1000;
    double threshold_multiplier=3.0;
    double tumor_stiffness_guess_low = 0.1;
    double tumor_stiffness_guess_high = 0.9;
    double stepsize = 0.003;

    void loadContour(bayesopt::utils::FileParser &fp, ContourParameters &cp);
    bool loadContourParameters(std::string filename, ContourParameters &cp);
    void PrintParameters();

};

```

Figure A.5: *ContourParameters struct* that holds tumor localization and segmentation parameters. It sets default values in case no parameters are loaded. *loadContourParameters* will parse contour_parameters.txt and save parameters.

A.4. CMakeLists.txt

Listing 1: CMkaeLists.txt

```
cmake_minimum_required(VERSION 3.16)
#set (CMAKE_CXX_FLAGS "-lstdc++fs -std=c++17")
set (CMAKE_CXX_FLAGS " -std=c++17")

cmake_minimum_required(VERSION 3.16)
#set (CMAKE_CXX_FLAGS "-lstdc++fs -std=c++17")
set (CMAKE_CXX_FLAGS " -std=c++17")

project(display_gp)
#https://stackoverflow.com/questions/8774593/cmake-link-to-external-library
#add_library(bayesopt_tut_libs main.cpp)
find_package(OpenMP REQUIRED)
if(OpenMP_CXX_FOUND)
    set(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} ${OpenMP_CXX_FLAGS}")
endif()
find_library(BAYESOPT_LIB bayesopt REQUIRED)
find_library(NLOPT_LIB nlopt REQUIRED)

find_library(MLPACK_LIB
  NAMES mlpack
  PATHS "$ENV{ProgramFiles}/mlpack/" /usr/lib64/ /usr/lib/ /usr/local/lib64/ /usr/local/
  REQUIRED
)
find_package(Boost REQUIRED)
find_package(CGAL REQUIRED)
message(OpenMPlibrarypath="${OpenMP_CXX_LIBRARIES}")
message(bayeslibrarypath="${BAYESOPT_LIB}")
message(nloptlibrarypath="${NLOPT_LIB}")
message(cgullibrarypath="${CGAL_DIR}" )
message(mlpacklibrary= "${MLPACK_LIB}")

enable_language(C)  #For OpenGL and bo_display demo


```

```
35  set ( ALGLIB_HEADER_FILES
36  libs/alglibinternal.h
37  libs/alglibmisc.h
38  libs/ap.h
39  libs/dataanalysis.h
40  libs/diffequations.h
41  libs/fasttransforms.h
42  libs/integration.h
43  libs/interpolation.h
44  libs/kernels_avx2.h
45  libs/kernels_fma.h
46  libs/kernels_sse2.h
47  libs/linalg.h
48  libs/optimization.h
49  libs/solvers.h
50  libs/specialfunctions.h
51  libs/statistics.h
52  libs/stdafx.h
53  )
54  set( ALGLIB_SOURCE_FILES
55  libs/alglibinternal.cpp
56  libs/alglibmisc.cpp
57  libs/ap.cpp
58  libs/dataanalysis.cpp
59  libs/diffequations.cpp
60  libs/fasttransforms.cpp
61  libs/integration.cpp
62  libs/interpolation.cpp
63  libs/kernels_avx2.cpp
64  libs/kernels_fma.cpp
65  libs/kernels_sse2.cpp
66  libs/linalg.cpp
67  libs/optimization.cpp
68  libs/solvers.cpp
69  libs/specialfunctions.cpp
70  libs/statistics.cpp
71  )
72
73
74  set(LIBS_PATH "${CMAKE_CURRENT_SOURCE_DIR}/libs")
75  message(libs_path=${LIBS_PATH})
76  message(ALGLIB_HEADER_FILES=${ALGLIB_HEADER_FILES})
77  add_library(alglib ${ALGLIB_SOURCE_FILES} ${ALGLIB_HEADER_FILES} )
78  target_include_directories(alglib PUBLIC ${LIBS_PATH})
79  #Display test
80  find_package(GLUT REQUIRED)
81  message(GLUT_library "${GLUT_LIBRARY}")
82  find_package(OpenGL REQUIRED)
83  if(OPENGL_FOUND AND GLUT_FOUND)
84
85
86
87    if(GLUT_LIBRARY MATCHES freeglut_static.lib)
88        add_definitions(-DFREEGLUT_STATIC)
89  endif()
90
91
92    INCLUDE_DIRECTORIES(${CMAKE_SOURCE_DIR}/matplotpp
93                        ${GLUT_INCLUDE_DIRS}
94                        ${GLUT_INCLUDE_DIR}
95                        ${OpenGL_INCLUDE_DIRS}
96            ${Boost_INCLUDE_DIRS})
97
98    link_directories(${GLUT_LIBRARY_DIRS}
99                     ${OpenGL_LIBRARY_DIRS})
100
101   add_definitions(${GLUT_DEFINITIONS}
102                   ${OpenGL_DEFINITIONS})
103
104   ADD_LIBRARY(matplotpp STATIC
105     matplotpp/matplotpp.cc
106     matplotpp/gl2ps.c
107     )
108
109   TARGET_LINK_LIBRARIES(matplotpp
110     ${GLUT_LIBRARY} ${OPENGL_LIBRARY})
111
112 set(UTILS_SOURCES
```

```
113         utils/displaygp.cpp
114         utils/fileparser.cpp
115         utils/param_loader.cpp
116         utils/parser.cpp
117         utils/ublas_extra.cpp
118
119        )
120     set(UTILS_HEADERS
121         utils/boundingbox.hpp
122         utils/displaygp.hpp
123         utils/fileparser.hpp
124         utils/gridsampling.hpp
125         utils/indexvector.hpp
126         utils/lhs.hpp
127         utils/log.hpp
128         utils/param_loader.hpp
129         utils/parser.hpp
130         utils/testfunctions.hpp
131         utils/ublas_cholesky.hpp
132         utils/ublas_cholesky.hpp
133         utils/ublas_elementwise.hpp
134         utils/ublas_elementwise.hpp
135         utils/ublas_extra.hpp
136         utils/ublas_trace.hpp
137         utils/prob_distribution.hpp
138         utils/randgen.hpp
139         utils/helper.hpp
140
141     )
142
143     set(SRCS
144         src/display2dgp.cpp
145         src/tumorModel.cpp
146         src/meanShift.cpp
147         src/contour.cpp
148         src/helper.cpp
149         src/parameters.cpp
150         )
151     set(HEADERS
152         include/display2dgp.hpp
153         include/meanShift.hpp
154         include/contour.hpp
155         include/tumorModel.hpp
156         include/evaluation.hpp
157         include/helper.hpp
158         include/parameters.hpp
159         )
160
161
162
163     include_directories(${CMAKE_CURRENT_SOURCE_DIR}/include )
164
165     add_library(utils_lib ${UTILS_SOURCES})
166     target_include_directories(utils_lib PUBLIC ${CMAKE_CURRENT_SOURCE_DIR}/utils)
167
168
169     add_library(headers ${SRCS} )
170
171     target_link_libraries(headers ${BAYESOPT_LIB} ${NLOPT_LIB} matplotpp ${MLPACK_LIB}
172                         ${GLUT_LIBRARY} ${OPENGL_LIBRARY}  alglib utils_lib)
173
174     add_executable(display_gp src/main.cpp )
175     add_executable(compute_contour src/contour_main.cpp)
176     add_executable(evaluate src/evaluation.cpp)
177
178
179     target_link_libraries(compute_contour alglib  ${OpenMP_CXX_LIBRARIES}
180     ${BAYESOPT_LIB} ${NLOPT_LIB} ${MLPACK_LIB} CGAL::CGAL ${Boost_LIBRARIES} headers utils_lib  )
181
182     target_link_libraries(display_gp ${OpenMP_CXX_LIBRARIES}
183     ${BAYESOPT_LIB} ${NLOPT_LIB} matplotpp alglib ${MLPACK_LIB}
184     ${GLUT_LIBRARY} ${OPENGL_LIBRARY} headers utils_lib)
185
186
187     target_link_libraries(evaluate alglib  ${OpenMP_CXX_LIBRARIES} ${BAYESOPT_LIB} ${NLOPT_LIB}
188     ${MLPACK_LIB} CGAL::CGAL ${Boost_LIBRARIES} headers utils_lib  )
189     target_compile_options(evaluate PRIVATE -fopenmp)
```

```
190 endif()CMakeLists
```

A.5. Dockerfile

Listing 2: Dockerfile

```
1
2  # Use Ubuntu as the base image
3  FROM ubuntu:focal
4  # Fix timezone issue
5  ENV TZ=Europe/London
6  RUN ln -snf /usr/share/zoneinfo/$TZ /etc/localtime && echo $TZ > /etc/timezone
7  # Update the system and install dependencies
8  RUN apt-get update && apt-get install -y \
9      build-essential \
10     libboost-all-dev \
11     libnlopt-dev \
12     libcgal-dev \
13     libomp-dev \
14     libarmadillo-dev \
15     libboost-math-dev \
16     libboost-program-options-dev \
17     libboost-test-dev \
18     libboost-serialization-dev \
19     git \
20     wget \
21     cmake \
22     && rm -rf /var/lib/apt/lists/*
23
24
25 RUN /bin/bash -c 'git clone https://github.com/stevengj/nlopt.git && cd nlopt && mkdir build && cd build && cmake .. && make && make install'
26 RUN git clone https://github.com/rmcantin/bayesopt.git \
27     && cd bayesopt \
28     && mkdir build \
29     && cd build \
30     && cmake .. \
31     && make \
32     && make install
33
34 RUN apt-get update && apt-get install -y libensmallen-dev
35
36 RUN wget https://github.com/USCiLab/cereal/archive/refs/tags/v1.1.2.tar.gz \
37     && tar -xzf v1.1.2.tar.gz \
38     && mkdir -p /usr/local/include/cereal \
39     && cp -r cereal-1.1.2/include/cereal/* /usr/local/include/cereal/
40
41 #RUN  git clone  https://github.com/mlpack/mlpack.git \
42 #   && cd mlpack \
43 #   && git checkout tags/3.2.2  \
44 #   && mkdir build \
45 #   && cd build \
46 #   && cmake .. \
47 #   &&  make install
48
49 # Set the working directory in the container
50 WORKDIR /usr/src/bo_robotic_palpation
51
52 RUN apt-get update && apt-get install -y \
53     libglu1-mesa-dev \
54     freeglut3-dev \
55     mesa-common-dev \
56     pkg-config \
57     libstb-dev \
58     && rm -rf /var/lib/apt/lists/*
59
60 RUN apt-get update && apt-get install -y \
61     libmlpack-dev \
62     && apt-cache policy libmlpack-dev
63 # Copy your project files into the container
```

```
64  COPY . .
65
66
67  # Compile your project
68  # Replace this with the actual build commands for your project
69  RUN cd build \
70      && cmake .. \
71      && make
72
73
74  # Setzen Sie den Befehl, der beim Starten des Containers ausgeführt wird
75  CMD ["bash"]
```

List of Figures

2.1 Construction of a polygon contour through straight line segments with the help of a bounding circle r_b and angle α. 4
2.2 One dimensional cross-section of two dimensional smooth step function. Third degree polynomial (blue) interpolating high and low stiffness values. Smoothness of interpolation is defined by parameter ε. Ground truth stiffness is at $r + 0.5\varepsilon$ (green dot) separating tumor (red) and non-tumor (green) regions. 6
2.3 Circle tumor model. Tumor defined through circular base shape of radius r with two dimensional smooth-step. 7
2.4 An example of a Bayesian optimization on a one dimensional objective function [4] 11
2.5 Convergence of mean shift toward high density regions. Circular windows depict the regions used to compute a center and mean shift vector that points toward direction of maximum increase in density. The size of the window is defined through bandwidth N. 14
2.6 Computation of stiffness value based on in-cluster variance and mean value. The threshold is calculated by subtracting the threshold multiplier times the in-cluster standard deviation $\beta\sigma$ from the mean value of the cluster μ. 15
2.7 Exploration of contour points. The tumor centroid (green dot) radially explored in $n = \frac{360}{\alpha}$ directions (blue dashed line) until the tumor outline (black line) is detected. The search for a contour point in each direction is stopped and a contour point identified (red dot) when stiffness measurement falls below stiffness threshold 2.6. 16
2.8 Contour approximation through interpolation of twelve contour points (blue) with a parametric cubic spline (red). 17

3.1 Inheritance structure of tumor models. 20
3.2 UML class diagram for Triangle tumor model. 21
3.3 rTheta_ function 22
3.4 *Polar Class* functions 23

3.5 Caption . . . 24
3.6 *evaluateSample* function. . . . 25
3.7 *Class GaussianNoise* . . . 25
3.8 Inheritance structure. . . . 27
3.9 First, three sets of parameters are created, one that defines the behaviour of the Bayesian optimization routine, another one that controls how the tumor contour is found and finally a set that defines the tumor model. Next the *Contour* object is created by passing in a tumor model, contour parameters and a dedicated directory to store results in. Lastly, the routines *runGaussianProcess*, *computeCluster*, *exploreContour* and *approximateContour* will perform the necessary computations to locate and segment the tumor. . . . 28
3.10 Example code for a state machine using Contour class. . . . 29
3.11 Contor Class constructor. . . . 30
3.12 Initialization of Bayesian optimization . . . 30
3.13 Update surrogate posterior model with new sample . . . 31
3.14 Function call to perform means shift clustering on posterior distribution. . . . 31
3.15 Function call to perform means shift clustering on posterior distribution. . . . 32
3.16 *scatterData_* and *pointNoPoint_* functions used to obtain a scattered points representation from probability distribution. . . . 33
3.17 Function that accesses mlpack's mean shift algorithm. . . . 33
3.18 *K_means Class* . . . 34
3.19 *labelData_* member function of *Contour Class*. Clusters stiffness measurements, tumor stiffness values and non tumor stiffness values. Stores measurements belonging to tumor in *tumor_stiffness_vec_* mean value of tumor measurements in *tumor_stiffness_vec_*. . . . 35
3.20 Computation of threshold value for contour point search. . . . 35
3.21 *exploreContour* function that searches for contour points in radial directions from tumor centers. . . . 37
3.22 Cubic spline approximation of contour points. . . . 38
3.23 Parameter files directory structure . . . 39
3.24 *class DisplayHeatMap2D* . . . 42
3.25 *DISPLAY* function . . . 43
3.26 Visualization of approximated tumor contour (blue), actual tumor contour (orange) and evaluation metrics used to compute sensitivity and specificity.Illustration source: [24] . . . 44
3.27 *Class ContourPairAnalyzer* . . . 45
3.28 *Class ContourPairAnalyzer* . . . 45

3.29 data directory with example 46

4.1 Euclidean distance d between true tumor centers and detected tumor centers over number of number of samples taken for circular, rectangular and triangular tumor models. As expected the error is decreasing for increasing number of interactions N 52
4.2 Comparison of Approximated contour and ground truth tumor. 53
4.3 Posterior distribution after 30 iterations. 54
4.4 Tumor model used for localization and segmentation. Stiffness values are inverted to formulate the optimization as a minimization problem. 54
4.5 Comparison of Approximated contour and ground truth tumor. 55
4.6 Posterior distribution after 30 iterations. 56
4.7 Tumor model used for localization and segmentation. Stiffness values are inverted to formulate the optimization as a minimization problem. 56
4.8 Comparison of Approximated contour and ground truth tumor. 57
4.9 Posterior distribution after 30 iterations. 58
4.10 Tumor model used for localization and segmentation. Stiffness values are inverted to formulate the optimization as a minimization problem. 58
4.11 Comparison of Approximated contour and ground truth tumor. 59
4.12 Posterior distribution after 30 iterations. 60
4.13 Tumor model used for localization and segmentation. Stiffness values are inverted to formulate the optimization as a minimization problem. 60
4.14 Sensitivity and Specificity for Circle, Triangle, Rectangle and TwoCircles experiments 61

A.1 bo_paramerts.txt file that contains Bayesian optimization parameters. . . 70
A.2 contour_parameters.txt file 70
A.3 tumor_model_parameters.txt file 71
A.4 *TumorModelParameters struct* that holds tumor model parameters. It sets default values in case no parameters are loaded. *loadModelParameters* will parse tumor_model_parameters.txt and save parameters. 72
A.5 *ContourParameters struct* that holds tumor localization and segmentation parameters. It sets default values in case no parameters are loaded. *loadContourParameters* will parse contour_parameters.txt and save parameters. 73

List of Tables