Imperial College London

MEng Group Project

Imperial College London

Department of Computing

Designing Sounds by Drawing Them

Authors:
Alexander Goncharov,
Codrin Cotarlan, Daria
Petca, Hyunhoi Koo,
Jeewoo Kim, Karan Obhrai,
Olivian Cretu

Supervisor:
Dr Mark van der Wilk

January 10, 2022

Contents

Chapter 1

Introduction

For the Software Engineering Group Project, we've built a sophisticated sound generation tool. Our software allows users to easily generate and hear a sound, simply by plotting points on a built-in graph display. After sound generation, a visualisation of the sound wave generated will be displayed to the user, allowing them to modify it until their desired sound is reached. If required, users can save the final sound produced directly to their devices.

To provide a great user experience, we created a simple, intuitive design with several sound modification options available for the user to choose from. Our tool is quick and responsive, as we have heavily researched and optimised the algorithms used both in the backend and frontend to generate a sound.

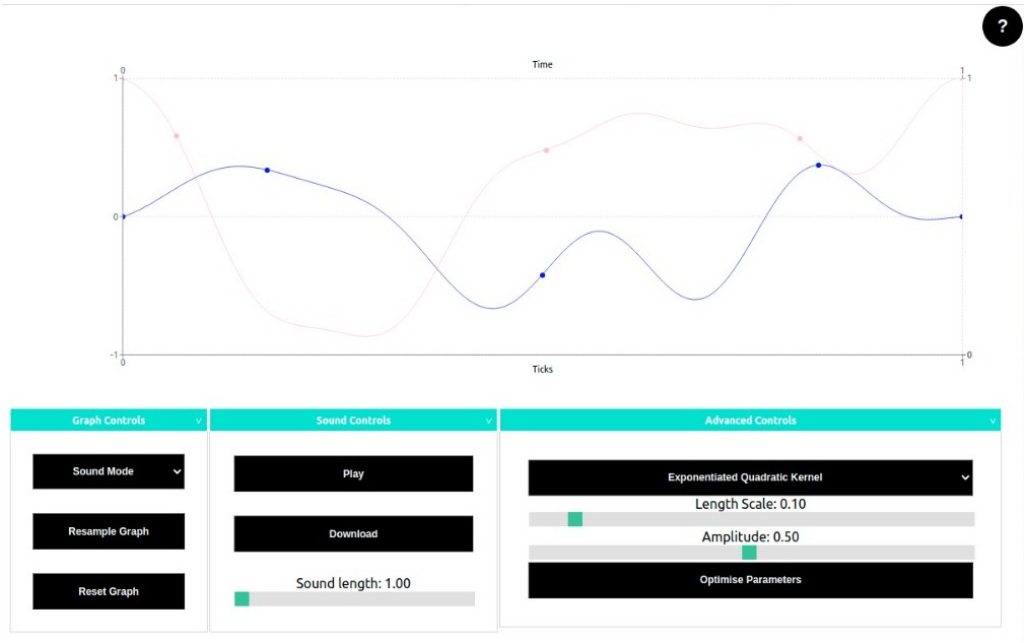

Figure 1.1: A screenshot of our tool.

Our software could be highly beneficial for organisations working within the entertainment industry, especially for those in the video game industry. This is due to the vast amount of specific sound effects needed to build products on a daily basis. Not only could our software help the industry, but could also help scientists who conduct academic research in the field of acoustics due to the precision our software provides.

Chapter 2

Design and Implementation

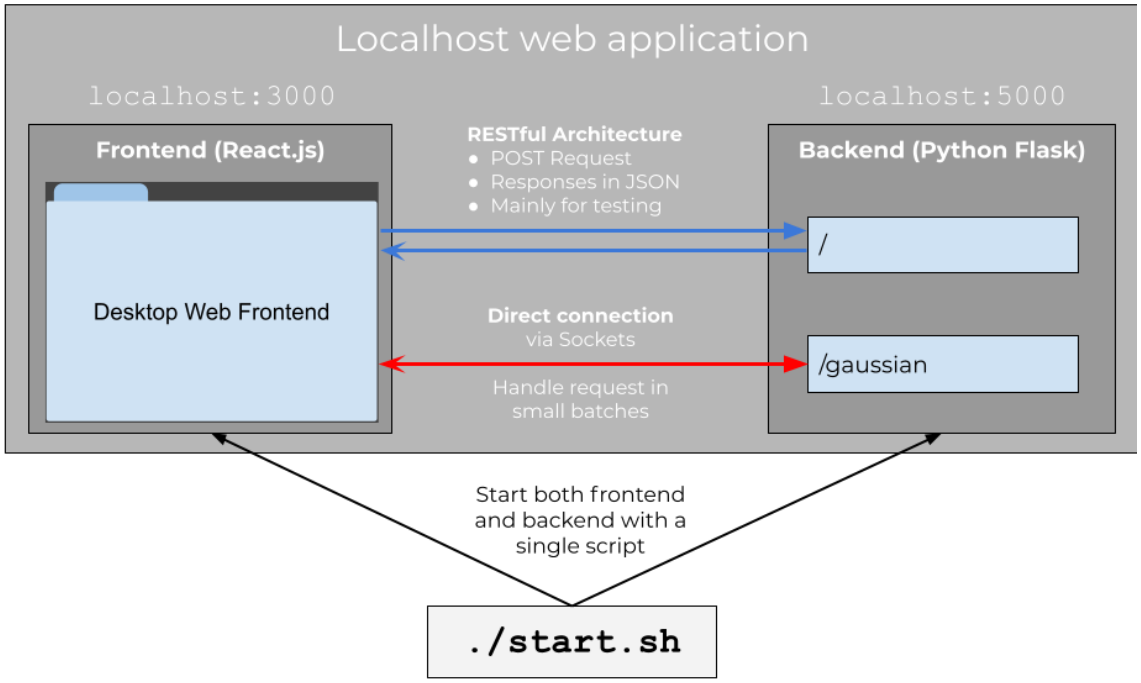

Figure 2.1: Our system architecture diagram

For our architectural design, we considered several possible options and chose the one which we believed to provide the best user experience. The core of our product is allowing the user to plot various points, where a Gaussian process is then run on these points to generate a sound wave, allowing the user to hear the sound. As a result, we aimed for a design where data is processed fast in the Gaussian process, to develop a quick and responsive user interface.

We initially intended to develop our tool on a mobile device. However, a mobile device display would have been too small for users to easily use the software, and would only be effective on large display tablet devices. As a result, since we require a large display to provide the optimal user experience, we decided to create either a web or native desktop application instead.

As we did not have much experience creating native desktop apps, we decided to develop a web application. The added benefit of this is that users who would like to use the tool on a mobile or tablet device are still able to do this.

After we agreed on creating a web application, we then needed to choose whether we should host our application, or have it run locally. Whilst a hosted website is much more convenient and easier to be accessed by non-tech savvy users, it can be very server heavy due to highly computationally expensive algorithms used within the Gaussian process. Additionally, sending large amounts of data over the internet can cause a delay in our application, where minimising latency is important for us to provide a quick and responsive user interface.

Therefore, we decided to develop our application as a local website that users would be able to run on their personal computer. Via LAN, users could also access the tool on their mobile or tablet if required. Following this decision, it was thereafter a logical step to host the backend locally.

After deciding this, we had to choose the tech stack we would use for both the frontend and backend. Our options for the frontend involved a range of JavaScript-based languages, as well as HTML-hosted Python servers. In the end, we decided to use TypeScript (1), as most of us had prior experience using it and provided us with the possibility to use many useful modules for sound generation, which will be elaborated upon later in this report.

For our backend, to decide on our stack, we evaluated our needs. Our backend was not required to perform any database or file writes, and instead, its main purpose was to perform complex matrix calculations. This meant that Python (2) was the best choice for us to use. This is because it balanced the speed of calculation with a high level of abstraction in matrix calculations, which would allow us to more easily implement the complex algorithms involved in the Gaussian Process. Additionally, Python has many libraries dedicated to Machine Learning that would be useful for us to use (e.g. GPflow(3), GPyTorch (4), etc.).

2.1 Frontend

Since we were developing a website in TypeScript, which is compatible with JavaScript libraries, we opted to use React (5). As a result, users can perform many different actions in our tool, which can all be efficiently handled by React.

Regarding the user interaction, we began by implementing some simple SVG graphics, including those for the sound graph, to develop a minimum viable product quickly and to ensure that a connection to the server was successful.

After developing the minimum viable product, we quickly realized that the SVG graphics were too basic, and so we decided to use a plotting library, Recharts (6), for our graph. We also briefly looked at potentially using Chart.js, which is in the same family as Recharts, but it was much more client intensive and therefore plotting many points took slightly longer than we wanted. Instead, we found the Recharts library easy to use and behave exactly how we wanted it to.

However, one of the main challenges we came across when using Recharts, was that the recharts library did not expose an API for the user to be able to plot their own points. To overcome this, we created a custom solution: when a user clicked on an area in the graph component, it would perform a calculation that converts the coordinate points of the user click to the coordinate on the graph based on the axis' size and the component's size in pixels (as well as including other factors such as the axis width, which was factored into the component size). Using this, we were

able to create a custom function that accurately mapped user clicks to points on the graph.

To style our webpage, we used SASS (7), a CSS extension. For our purposes, regular CSS wouldn't have behaved differently than SASS but was clumsier to use; we didn't experience any issues writing our styling in SASS.

The sound generation in the frontend was extremely complex. Initially, we used a library called ToneJS (8); it's an incredibly powerful API that utilises the intrinsic audio components included within the Audio Web API in default JavaScript, to create a range of sounds including amongst other things, instruments and complex timing functions.

Initially, we used ToneJS to take the points plotted by a user, and effectively generate a sound that corresponded to the points as if it were a pitch-time graph. Whilst this was a good step at the time as a proof of concept (despite being incorrect in terms of the final aim), the duration it took for a sound to be generated was very long and the process would have to be reperformed each time a user plotted a new point.

Therefore, we decided it would be best to no longer use this library and utilise more low-level Web APIs provided in the frontend JavaScript languages: we directly accessed the audio context of the client, so that whenever the user plotted a point, we would perform a backend fetch, then update the generated sound in the state of the frontend (displaying it on the graph).

The raw values generated would then be loaded into a buffer array of size, 700 values, and sent directly to the user's speakers. We kept a fixed sampling rate of 42000 Hz, as based on the Nyquist Theorem (9), should provide a tone of good quality.

Our frontend receives 700 points back from the backend, including the anchor points at 0 as the first and last point of the wave. These points in the buffer are then played in a loop until the time specified is elapsed or the user plots a new point on the graph, creating a new sound. This worked effectively as we were guaranteed to create a sound wave at a minimum of 60Hz (being a frequency range that could be easily heard by a human (10)).

Whilst, in theory, we could allow the user to plot each individual part of the wave - unless they plot a large number of points, it will be unlikely that they will generate a usable sound. Hence, we repeat their generated sound wave to provide a better user experience. Moreover, generating 700 points is near-instant to the human perception, whereas generating 42,000 points every time the user plotted a point on the graph would result in very high latency, and therefore a poor user experience.

2.2 Backend

The problem that we aimed to solve in the backend was to generate a waveform given a user's data points - this can be classified as a regression problem. Moreover, the regression model used to solve this problem should be capable of robust extrapolation, as the ideal soundwave would be a repetition of the sound wave plotted by the user.

We chose to use Python as the main language to build the backend, as well as the Flask framework (11) to host the server. We chose to use Python, as it is

the most widely used language for Numerical Computation and Machine Learning - there exists a huge community of people using these libraries that proved to be very helpful when trying to solve errors we came across during development by asking for help in community forums.

To generate a sound wave, we decided to use a Gaussian process (12) as the core algorithm. This is because a Gaussian process is a powerful algorithm for solving regression and classification problems, especially those that are robust with extrapolation when using periodic or spectral mixture kernels (13).

We implemented the Gaussian process from scratch using the NumPy (14) library, however, we soon encountered some technical challenges: to generate a waveform that passes through the given data points, we were required to sample from the posterior distribution, which has a time complexity of $O(n^3)$ due to the Cholesky decomposition (12).

Moreover, to generate a high-quality sound, the sample rate should be at least 44.1kHz, and for it to be 3 seconds long, we needed to sample 132300 data points and run a Cholesky decomposition of a 132300 by 132300 matrix. Even if we sampled only 10000 data points, a computer with 8GB of RAM will crash with an 'out-of-memory' error.

We decided to use an open-source library GPyTorch, as it provides us with a few advantages. Firstly, it provides us with support for speed and resource utilisation, allowing us to easily apply optimisation techniques to rapidly speed up the time it takes to sample from the posterior. Also, the modular design of GPyTorch enables us to maintain a clean code structure. For 1-2 weeks, we migrated the new version of our code for Gaussian Process with GPyTorch and ran several benchmarks in order to evaluate the effect of optimisation techniques. Amongst many other optimisation techniques such as the BlackBox Matrix-Matrix Inference (BBMM) (15), and LancZos Variance Estimates (LOVE) (16), we decided to implement LOVE for our optimisation.

The main idea of the LOVE algorithm is as follows: efficiently decompose a matrix in near linear time using the Lanczos algorithm and with such upfront computation, due to the special structure of the decomposed matrices, sampling can be completed in constant time (16). However, LOVE did not perform as we expected. We believe that this is likely due to the repeated upfront computations completed whenever the user adds a new data point, which is in effect the same as creating a new training dataset.

With the help of our supervisor, we found a recent academic research paper about optimising the speed of sampling from the posterior using Matheron's rule (17). The main idea presented in this research paper is to update the prior as realised in terms of the sample paths, instead of conditioning it as a distribution. This, as a result, allows us to accurately sample from the posterior in linear time.

Using this, we discovered an open-source repository created by the author of this paper, where we were then able to easily implement an optimisation based on their implementation, even though we had to re-implement Gaussian Process system with GPflow. After implementing this optimisation, our algorithm now takes roughly 0.5-0.6 seconds to sample 132300 data points, which is a significant improvement compared to our previous result.

2.3 Data Flow

After a user had finished plotting their points on the graph, we had to find the best solution of sending these points from the frontend to the backend and thereafter, sending a large number of generated points back to the frontend and plotting them on the graph, all with minimal latency.

Our initial product used a RESTFUL architecture, where the frontend would send an HTTP POST request to the backend and in turn, send back a JSON structured list of points. Whilst this was successful in the very early stages of development when we only received back a few points, we found that this API was unreliable and that the latency would heavily fluctuate, despite being local.

Ultimately, we decided to migrate to using WebSockets instead. Using these meant that port collisions were very unlikely to occur, as we only had one connection per backend. Also, it resulted in the overhead of establishing a connection being performed on load, instead of on every occasion a point was plotted by the user. This provided a huge increase in the speed of the program and the rate at which we received data from the backend. Moreover, it allowed us to create an overlay over the graph which would display a message if the connection to the backend was lost, providing the user with better visibility if their locally run server was blocked for any reason.

Latency on the frontend is something we have always been very aware of - in fact, for our initial checkpoint, the graph and points drawn were implemented manually using SVG graphics. However, we discovered that when we attempted to plot 100+ points using this method, it had resulted in a massive impact on user performance. Hence, we pivoted to using a library instead, even though we would need to create our own API for a user to plot points.

For the frontend, however, perceived latency is just as important to user experience as actual latency. This is the reason why we created an API on our backend, which would allow the frontend to specify 'batches' - batches are effectively an array list of points that we would like the backend to return. For example, if batches were set to be the array, [100, 2000, 7000], then once a request was made, the web socket would first return 100 points, then 2000, then 7000. As a result, the user received instant feedback on any action performed as the first 100 points would be generated near-instantly, where then the graph would slowly build and form closer towards the actual wave as larger batches were generated. The aim of this was to ensure that the user wouldn't have to wait for any indication of success after plotting a point, but still ultimately received the same precision. Our user experience was also aided by our graph library choice, Recharts, as there is an intrinsic animation function that provides a smooth transition between graphs on re-render.

However, we did encounter a small technical challenge with this in the form of a concurrency issue: for example, if a user plotted a single point, and then a second point whilst the batches of the first point were still processing, then it would be likely that a set of generated points pertaining to the first point, but not the new, second point, would be returned to the frontend, which could be drastically different.

Therefore to provide a smooth transition, each point is tagged, and each tag is sent to the backend which returns the tag with the data. This means that if a tag returned by the backend is different to the current tag for the data it is expecting, it discards the data.

The points are not the only thing of interest to us when creating sound waves. The Gaussian process that generates the points is defined by a kernel function and a set of parameters for it. The kernel function gives the wave an overall type (such as periodic), while the parameters affect the particular shape of the wave (such as it's amplitude or length scale).

We initially started with the exponentiated quadratic kernel as the only kernel used in our Gaussian processes because it's simplicity and to ensure the process runs correctly. Once we were satisfied with our results, we expanded the types of kernels we used to include the rational quadratic kernel, the periodic kernel and the local periodic kernel and explored how different kernels affect the created waves.

With the kernels established, our next objective was to allow the user to have more control over the parameters we mentioned previously. As a result our website offers the ability to choose the kernel type and all it's parameters before requesting a full sound wave. The points would be sent through the web socket together with the kernel type and the parameters so that the server would know how to process the request. Furthermore, in order to find a wave that matches the inputted points as best as possible, the user can also choose to 'optimise hyperparameters', in which case the server would return not only the wave, but also the optimal hyperparameters used to create the wave.

2.4 Sound Amplitude Modulation

For our project, we thought that an interesting extension would be to implement an option for amplitude modulation of the sound generated, as it would allow the user to produce a much larger range of sound by adding a completely different dimension. This was a good modular extension for us, as it built upon the majority of the previous work we had already developed: the Gaussian process was entirely reusable in terms of generating a graph to fit the sound modulation, and a large amount of the frontend algorithmic work such as calculations to plot points were already built.

The first thing we had to consider was the type of user interface we should provide. One option was to create a separate graph for the user to use, which would have fewer technical challenges as we would just have to effectively duplicate the current graph component we have and alter the effect it has on sound generation. However, we decided that this would not provide the best user experience, as the amplitude modulation with the sound wave generation would not be properly unified. Therefore, we decided that we would display both graphs on the same axis and that the type of graph the user was editing would depend on the mode that they had selected in the options table.

The aim with this however was still to reduce and optimise the code. Hence, in the frontend, we effectively moved any graph specific states under a 'sound' state, and duplicated all those values under an 'amplitude' state which would then allow for the program to automatically store different values, including user plotted points, generated points, and kernel parameters automatically, as well being able to switch automatically between them on the user interface by updating the value of our kernel inputs to be dependant on the selected state. As a result, this enabled us to provide a good and reliable user interface.

The next step for us was to think about how different lines on the graph would be

represented. This is because we had to ensure that the user could easily differentiate between the graphs, and as will be explained later in the report, the graphs could have different axis values and ranges. Therefore, we decided that each line on the graph (amplitude or sound) would be in different colours, specifically red and blue respectively.

To further help the user distinguish between them, the line related to the mode in which the user was currently not in, would be faded to a lighter version of its respective colour. This meant that the user could more easily identify which waveform it was currently editing. Similarly, the axes for the graphs would become a solid colour depending on the mode in which the user was currently in, where the Y and X axes for the sound wave were on the left and bottom respectively, whereas the Y and X axes for the amplitude wave were on the top and right respectively.

Furthermore, we then decided that it would be beneficial to add a variable sound length for amplitude modulation, rather than previously where the sound length was fixed at 1 second. This is because a 1-second tone is sufficient to hear a sound, whereas, for amplitude modulation to be more effective and noticeable for a user, a longer time frame is required. Therefore, the user would still plot a single tick for the sound wave, but the tone would now be played back for a specified duration (anywhere between 1-10 seconds). We designed it so that the X-axis value for the amplitude corresponds to the length of the sound, and the user can alter the amplitude of the wave over a range of periods.

Adding a second data function posed a similar problem to the one mentioned with asynchronous batch fetches. For example, if a user plots a point in sound mode, but then switches and plots a point in amplitude mode before all the batches in sound mode are complete, then a later returning batch for the sound wave may be plotted on the amplitude graph.

To fix this, we first added another data tag, this one specifically for amplitude, and the original data tag was configured specifically to the sound wave, to ensure that the tag for each wave was independent. However, we still had a potential issue where inconsistent data may be plotted if, by chance, both tags are at the same current value. Therefore, we also added the mode of the user in the API to the backend, which is returned with the data to ensure that the correct generated point graph is plotted, as well as ensuring that a late batch of data returning for the mode the user is currently not in, can still be updated in the background.

Finally, it is important to discuss how we altered the sound generation. This was simple on the Web API, as it exposes an object called a 'GainNode', that allows us to perform linear ramping of amplitude between two amplitude values and two timestamps, which we calculate by distributing the amplitude points evenly into the specified sound length.

2.5 Risks Anticipated / Mitigated

We aimed to deliver a product that fulfils a clear goal, that is to generate a sound out of a drawn wave. As such, one important risk in our project was a user having difficulties understanding our UI. To mitigate this, we implemented all the essential features of our tool and took extra care to ensure that the UI was as simple and clear as possible, whilst still looking esthetically pleasing.

To generate sounds, we run a Gaussian process on the user's plotted points.

Therefore, the main risk of our project consisted in not being able to implement this process correctly. If that were to happen, then we wouldn't be able to generate a correct graph plot and thus neither a sound. Hence, throughout the project, we often did pair programming and code reviews/testing within both sub-teams (backend and frontend). This was because the intuition needed to implement most features required a large amount of prior reading and studying to be done beforehand, which could have led to more errors arising if a topic was not understood entirely correctly before implementation.

Also, to mitigate any other unforeseen risks, we agreed on a strict development schedule. In particular, alongside daily sub-team meetings, we met as a whole team twice per week to discuss what we implemented, any issues that occurred, and what would be our next goals.

Chapter 3

Evaluation

To ensure that our final produced code was of high quality, we implemented multiple unit tests for the backend.

Also, to ensure scalability, we conducted multiple tests to determine the best number of data points that should be returned on each request (shown in figure 3.1). After analysing our tests, we chose 700 data points to be returned to maximise efficiency and minimise the lag produced, which, as a result, increased the user experience.

In our final product, we include several beneficial features that can be used by users to modify the sound they generated, which are listed below:

- Add new data points to the sound wave.
- Change the amplitude of the sound wave.
- Adjust hyperparameters of the kernel to fit current plotted data points.
- Choose which kernel to use from a selection of different options available.

As a result, developing many different modification features into our tool has provided users with great flexibility and precision to modify the sound wave generated until their desired sound is reached.

Moreover, to reduce the latency in our application, we heavily optimised the rate at which points are sampled from posterior in the backend using Matheron's rule (our analysis is shown in figure 3.2), and required the tool to have to be downloaded to a user's device to be used. This helped our application to become more responsive and appear to be more in 'real-time' for the user, which increased the user experience too. Also, we noticed that for 50,000 and 100,000 number of data points, running LOVE and Exact GP with Cholesky decomposition caused an out-of-memory error to occur.

Regarding the weaknesses of our tool, there will always be a delay in sound generation due to the highly computationally expensive algorithm used in the backend to generate the data points of the new sound wave. As a result of this, we decided to not host our application due to it being expensive and causing the latency to increase. This reduced the ease of access to the tool and our user target audience.

To take this project further, it would be beneficial to have our tool included in plugin libraries of commonly used software in the industry (e.g. Audacity), to enable users not having to switch often between different applications when building products, thereby increasing productivity. Also, it would be useful to develop

different visualisations of a generated sound to allow for users to uncover new ideas in which it could be modified.

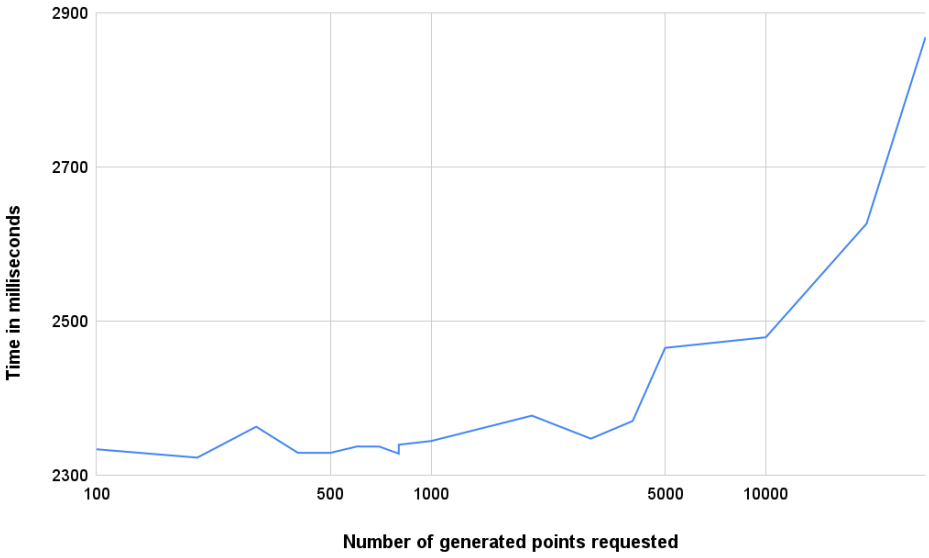

Figure 3.1: A graph to show how the response time of a HTTP request to our backend varied when generating a varying number of points for the exponentiated quadratic kernel.

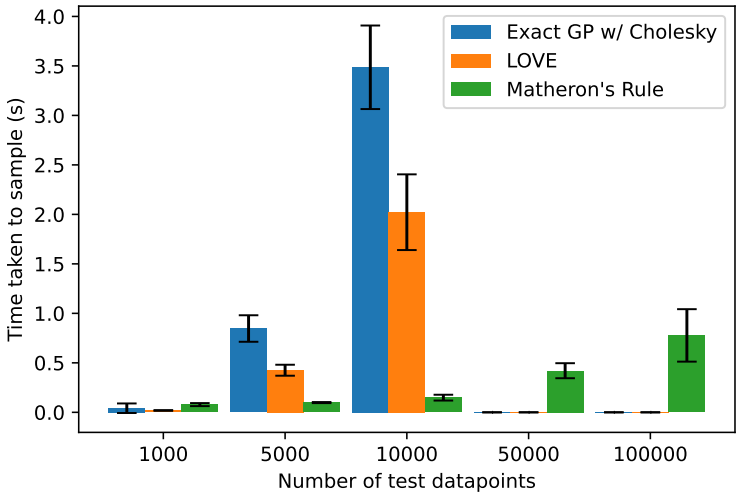

Figure 3.2: A graph to show the time taken to sample based on the number of test data points, both with and without optimisation.

Chapter 4

Ethical Considerations

Fortunately, due to the nature of our project, there are no major ethical issues. However, there are a few ethical topics that we should take into consideration.

First of all, user interaction is necessary for our project (a user has to plot points on the graph for any sound to be generated). For this, we do not collect and/or process any type of personal data, thus being both GDPR and COPPA compliant.

Another area of interest in which our product may be used is the creation of new sounds and their distribution. For example, our tool could be used to generate sounds for a music track or video game. In these situations, users should be aware that the sound they generate may be similar to a copyrighted composition, and therefore subjected to any laws concerning that, such as the DMCA.

Bibliography

[1] Microsoft. *TypeScript is a superset of JavaScript that compiles to clean JavaScript output.* Available from: https://www.typescriptlang.org/. [Accessed 5th January 2022].

[2] Python Software Foundation. *Python is a programming language that lets you work quickly and integrate system more effectively.* Available from: https://www.python.org/. [Accessed 5th January 2022].

[3] The GPflow Contributors. *Gaussian processes in TensorFlow.* Available from: https://gpflow.readthedocs.io/en/master/. [Accessed 5th January 2022].

[4] Gardner JR, Pleiss G, Bindel D, Weinberger KQ, Wilson AG. *GPyTorch: Blackbox Matrix-Matrix Gaussian Process Inference with GPU Acceleration.* Available from: https://gpytorch.ai/. [Accessed 5th January 2022].

[5] Meta Platforms, Inc. *React – A JavaScript library for building user interfaces.* Available from: https://reactjs.org/. [Accessed 5th January 2022].

[6] Recharts Group. *Recharts - A composable charting library build on React components.* Available from: https://recharts.org/en-US/. [Accessed 5th January 2022].

[7] Sass. *Sass: Syntactically Awesome Style Sheets.* Available from: https://sass-lang.com/. [Accessed 5th January 2022].

[8] Mann Y. *A Web Audio framework for making interactive music in the browser.* Available from: tonejs.org. [Accessed 5th January 2022].

[9] Elsevier B.V. *Nyquist Theorem - an overview.* Available from: https://www.sciencedirect.com/topics/engineering/nyquist-theorem. [Accessed 5th January 2022].

[10] Amplifon. *The Human Hearing Range.* Available from: https://www.amplifon.com/uk/audiology-magazine/human-hearing-range. [Accessed 5th January 2022].

[11] Pallets. *The Python micro framework for building web applications.* Available from: https://flask.palletsprojects.com/en/2.0.x/. [Accessed 5th January 2022].

[12] Rasmussen CE, Williams CKI. *Gaussian Processes for Machine Learning.* Massachusetts: MIT Press; 2001.

[13] Görtler J, Kehlbeck R, Deussen O. *A Visual Exploration of Gaussian Processes*. Available from: https://distill.pub/2019/visual-exploration-gaussian-processes/. [Accessed 5th January 2022].

[14] NumPy. *NumPy: The fundamental package for scientific computing with Python*. Available from: https://numpy.org/. [Accessed 5th January 2022].

[15] Gardner JR, Pleiss G, Bindel D, Weinberger KQ, Wilson AG. *GPyTorch: Blackbox Matrix-Matrix Gaussian Process Inference with GPU Acceleration*. Available from: https://arxiv.org/pdf/1809.11165.pdf. [Accessed 5th January 2022].

[16] Pleiss G, Gardner JR, Weinberger KQ, Wilson AG. *Constant-Time Predictive Distributions for Gaussian Processes*. Available from: https://arxiv.org/pdf/1803.06058.pdf. [Accessed 5th January 2022].

[17] Wilson J, Borovitskiy V, Terenin A, Mostowsky P, Deisenroth M. *Efficiently sampling functions from Gaussian process posteriors*. Available from: https://arxiv.org/pdf/2002.09309.pdf. [Accessed 5th January 2022].