# DIFFUSION MODELS AND CONSTRAINED GAUSSIAN PROCESSES

PROBABILISTIC MACHINE LEARNING

**Björn Thor Stefánsson**
University of Copenhagen
qrx747@alumni.ku.dk

**Oliver Sanchez**
University of Copenhagen
fml137@alumni.ku.dk

**Viktor Már Guðmundsson**
University of Copenhagen
smp884@alumni.ku.dk

January 17, 2025

## 1 Introduction

Diffusion models and Gaussian Processes are powerful tools in probabilistic machine learning. This report examines two specific applications: extending diffusion-based generative models for the MNIST dataset and fitting functions under integral constraints using Gaussian Processes.

For the first part, we implemented a baseline Denoising Diffusion Probabilistic Model (DDPM) and extended it with four variations to explore improvements in image generation quality. In the second part, we investigated kernel selection, prior design, and optimization for GPs to satisfy integral constraints. The report is structured to provide theoretical insights, experimental results, and discussions.

## 2 Part A: Diffusion-Based Generative Models

In this part, we created DDPMs based on the architecture proposed by Ho et al, 2020, for generating 28x28 greyscale images of handwritten digits from 0 to 9 using the MNIST dataset. We started with the simple diffusion model from the `mnist_ddpm_solutions.ipynb` notebook and extended it with 4 variants. We compare the models' performance visually and quantitatively. The visual plots are in Appendix B.1

### 2.1 Extensions/Variations of the MNIST Diffusion Model

#### 2.1.1 Baseline Model

The baseline model is a standard DDPM. It utilizes a U-Net-like neural network to predict the noise $\epsilon$ added to the clean data during the diffusion process. The forward diffusion step corrupts the input $x_0$ by gradually adding Gaussian noise over $T$ timesteps, while the reverse process iteratively removes this noise to generate data samples. The U-Net incorporates time embeddings to condition its predictions on the diffusion timestep $t$. Training is guided by a simplified Evidence Lower Bound (ELBO), implemented as the mean squared error (MSE) between the predicted and actual noise.

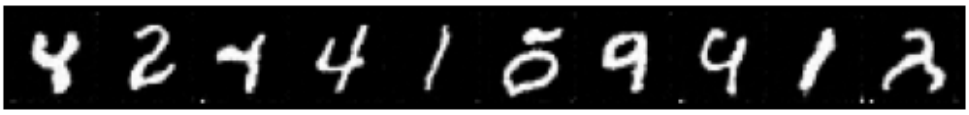

Figure 1: Generated samples from the Baseline model. Step size $T = 1000$

#### 2.1.2 Variant 1: Predicting the Original Clean Image $x_0$ Instead of Noise $\epsilon$

In this variation, we modified the standard DDPM to predict the original clean image $x_0$ directly, instead of the noise $\epsilon$ added during the diffusion process. The reverse diffusion step was adjusted accordingly to use the predicted $x_0$ to compute $x_{t-1}$ (see Appendix A.1 for derivation). Specifically, the MSE loss between the predicted and true noise was

replaced with an MSE loss between the predicted and true clean image. The visual performance was worse compared to the baseline, showing less diverse and more uniform outputs (often repeated ones).

#### 2.1.3 Variant 2: Low-Discrepancy Sampling to Reduce ELBO Variance

When training with uniform sampling of the timestep we noticed great variability in the loss, which is an approximation of the VLB. To reduce this variability, and achieve a smoother training loss we implement two alternative sampling methods. First we implemented low-discrepancy sampling in discrete form, inspired by the continuous form method described in Kingma et al, 2021. In this method we sample the timesteps for the entire batch at once, spreading the timestep values spread out over its range, while previously we sampled each sample's timestep randomly.

#### 2.1.4 Variant 3: Importance Sampling to Reduce ELBO Variance

The second sampling method we explored was importance sampling, first described in Nichol & Dhariwal 2021. In this method we sampled the timestep from a probability distribution based on how well the model had predicted for each timestep in its last ten attempts. The motivation behind this method is for the model to exposed to more iterations of timesteps its struggling to learn the most, possibly leading to faster convergence.

#### 2.1.5 Variant 4: Classifier-Free Diffusion Guidance

This variant enhances the diffusion model with classifier-free guidance, which uses the class label in the prediction. The network embeds the class label similarly to the timestep embedding and concatenates the two before passing them through the network. To encourage robustness to unconditioned samples, we set the class label embedding to zero for 10% of training steps. During reverse diffusion, the conditional and unconditional predictions are combined as:

$$\text{pred} = (1 + w) \cdot \text{pred}_{\text{cond}} - w \cdot \text{pred}_{\text{uncond}},$$

where $w$ controls the balance between diversity and fidelity. Read more about it in this article that we followed.

### 2.2 Quantitative Comparisons

To quantitatively evaluate model performance, we used modifications of Fréchet Inception Distance (FID), and likelihood scores, which measure the quality, diversity, and alignment of generated samples with the data distribution.

**Fréchet Inception Distance (FID):** FID measures the similarity between the distribution of generated and real images by calculating the Fréchet distance between their feature embeddings extracted from a pre-trained network. A lower FID score indicates that generated images are more realistic and better aligned with the real data distribution. However, implementing FID for MNIST required modifications, as the standard Inception network is trained on ImageNet and expects 3-channel images. Inspired by a GitHub discussion, we trained a variational autoencoder (VAE) on the MNIST test set to extract latent feature embeddings. We validated the VAE's representativity using t-SNE and PCA, then computed FID by comparing the latent feature distributions of the generated samples and the MNIST test set.

**Likelihood Scores:** Likelihood measures how well the model predicts the real data distribution, but it poses challenges for diffusion models. Exact computations are infeasible due to high dimensionality in latent spaces and the iterative nature of the diffusion process. Instead, likelihood is approximated using the Evidence Lower Bound (ELBO), which consists of reconstruction accuracy (MSE between predicted and true noise) and a regularization term (KL divergence between the posterior and prior distributions).

While these approximations provide insights into training dynamics, likelihood scores often poorly correlate with perceptual quality, as they primarily measure the model's ability to reverse the diffusion process rather than the realism or diversity of the generated samples. (Belongs in discussion?)

**Discussion:** FID strongly correlate with human-perceived quality, while likelihood scores offer limited visual insights. Therefore, we prioritize FID, using likelihoods for complementary insights into model behavior.

Table 2.2 summarizes the FID, and likelihood results for each model variant.

### 2.3 Results and Discussion

The quantitative results suggest that variant 4, guided diffusion, generates the images that resembles the training set the most. The baseline was surprisingly performant when compared with variants 1-3. Variants 2 and 3, which used alternate sampling methods, had very similar MSE and ELBO score as to the Baseline. The FID score, however suggests that Variant 2 was significantly worse at covering the distribution of the training data. We hypothesize that this

| Model | FID | MSE | ELBO |
|---|---|---|---|
| Baseline | 42651.1 | $1.15 \times 10^{-4}$ | $5.80 \times 10^{-2}$ |
| Variant 1 | 42784.0 | $2.6 \times 10^{-2}$ | $8.19 \times 10^{-2}$ |
| Variant 2 | 43158.7 | $1.20 \times 10^{-4}$ | $5.69 \times 10^{-2}$ |
| Variant 3 | 42710.1 | $1.19 \times 10^{-4}$ | $5.78 \times 10^{-2}$ |
| Variant 4 | 41739.6 | $5.18 \times 10^{-5}$ | $2.87 \times 10^{-2}$ |

Table 1: FID, and likelihood scores for the baseline and model variants.

is due to the lower discrepancy sampling producing sampled images that are more random than for example the priority sampling. And possibly due to some randomness in the FID score that this specific score was high. Variant 1 produces an acceptable FID score but its MSE and ELBO scores are a lot higher than expected. This shows that while it does not reduce its ELBO and MSE error as much, showing that its likelihood cannot be as minimized it is able to generate surprisingly realistic images. Guided diffusion, variant 4, gets the best scores in all metrics, and shows that when the model is given its target, guiding it towards a single number, it can produce better MNIST images. We hypothesis that this can be ascertained to be due to the certainty the model has throughout its inference, resulting in no mixing between different numbers. The other models have a mix of multiple numbers and in the space of their samples the space is less divided, less clear distinctions in the space of all possible generated images.

Similarly, the visual results in Appendix B.1 show that Variant 4 performed the best, due to its use of classifier-free guidance, which provides class label information to the network. This extra information allows the model to better distinguish between classes and produce more accurate samples. Variants 2 and 3, which implemented advanced sampling methods to reduce ELBO variance, performed comparably, with Variant 2 slightly ahead in generating more realistic examples. The baseline followed, while Variant 1 performed the worst, producing less diverse and less realistic outputs.

Variant 4's superior performance can be attributed to the inclusion of class labels, which provides additional context to guide the diffusion process. This approach reduces the burden on the model to learn the class distinctions solely from noisy input, resulting in clearer and more accurate samples.

## 3 Part B: Function Fitting with Constraints (GPs)

We assume that our data is given by

$$y_i = g(x_i) + \varepsilon_i \quad \text{with} \quad g(x) = -\left(\sin(6\pi x)\right)^2 + 6x^2 - 5x^4 + \frac{3}{2}$$

where $\varepsilon_i \sim \mathcal{N}(0, 0.01)$ and $x_i = \frac{i}{29}$ for $i = 0, ..., 29$. See Figure 2 for a plot of the generated data.

Our goal is to approximate the function $g(x)$ using MAP and other inference methods. Note that since we assume that our noise is has a fixed parameter $\sigma_y^2 = 0.01$; then our model does not require a prior distribution for the parameter $\sigma_y^2$.

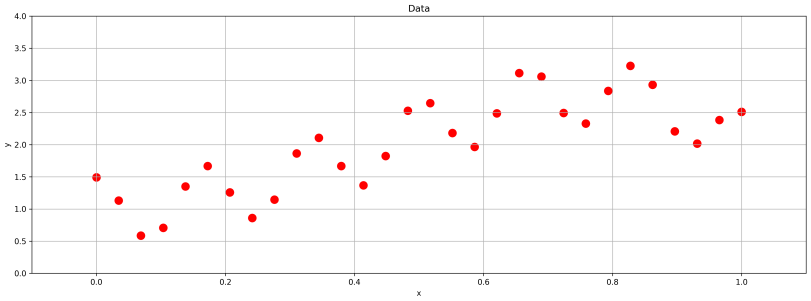


Figure 2: Generated Data $y_i = g(x_i) + \varepsilon_i$

### 3.1 Kernel Selection

For the kernel, we considered several viable options based on the observed characteristics of the data. When plotting the data, it is evident that there is a periodic trend, and the mean values of nearby data points appear to increase linearly across most of the dataset. However, the data points near the edges deviate slightly, suggesting that the mean increase might be nonlinear or influenced by noise. Based on these observations, we tested the following kernel combinations:

i) Periodic $\times$ RBF ii) Periodic $+$ RBF iii) Periodic $\times$ Linear iv) Periodic $+$ Linear

The results (see Appendix **??**) indicated that the **Periodic** $+$ **RBF** kernel performed the best across various performance metrics, optimization methods, and train-test splits (e.g., Leave-One-Out Cross Validation). This result aligns with our expectations from data visualization: the periodic magnitude remains fairly consistent throughout the dataset, except possibly at the edges. While the unusual ends of the data might suggest some irregularities, testing revealed that these could be better modeled with an additive RBF kernel rather than a multiplicative one. Furthermore, the RBF kernel captures potential nonlinearities more effectively than the Linear kernel.

### 3.2 Prior Selection for the Combined Kernel (Periodic + RBF)

The kernel parameters–**lengthscale** $(l)$, **output variance** $(\sigma^2)$, and **period** $(p)$ require carefully chosen priors to encode our assumptions about their plausible values. The chosen priors are justified as follows:

- $l \sim$ **LogNormal**$(-1.0, 0.7)$.

  **Role of Lengthscale**: The lengthscale governs the smoothness of the function, both for the periodic and non-periodic components. Smaller values allow the function to change more rapidly, while larger values enforce smoother variations.

  The **LogNormal** distribution is strictly positive and focuses its probability mass on smaller values. Our parameter selection of $\mu = -1.0$ and $\sigma^2 = 0.7$ reflects the belief that the periodic pattern is not very extreme. However, the distribution tail makes the model able to retain the ability to explore state spaces which are sufficient to adapt to the periodic pattern if they turn out to be stronger than anticipated, i.e. our belief that the patterns will not require such extremities.

- $\sigma^2 \sim$ **LogNormal**$(-1.0, 0.7)$.

  **Role of Output Variance**: The output variance determines the amplitude of the function, representing the overall scale of variability in the modelled outputs.

  The **LogNormal** distribution ensures strictly positive values and offers a broad range of plausible amplitudes. Our choice of $\mu = -1.0$ and $\sigma^2 = 0.7$ provides flexibility for capturing both moderate and higher variability in the data. Again, as per the lengthscale, the tail probability mass allows the model to retain possible state spaces to a certain degree that we believe it to be less likely, i.e. the more extreme values.

- $p \sim$ **Uniform**$(0.3, 1.0)$.

  **Role of Period**: The period determines the distance over which the periodic component of the function repeats itself. This parameter is critical for capturing the periodic nature of the data.

  By restricting the prior to $[0.3, 1.0]$, we encode the assumption that the period of the data lies within this range, which is reasonable given the interval $[0, 1]$. However, this is an uninformative prior since we do not think one is more likely than another. Initially, we had thought that the periodic parameter would be the actual parameter of the periodic pattern but the relationship between the kernels seems to be more complex hence we opted to forgo our initial thoughts and decided on an uninformative bounded prior.

### 3.3 Optimization

First, we normalized the y values as the GP expects a mean of zero and used Adam for the optimizer due its results from the kernel selection testing, see figures in Appendix B.2 and B.3. To find the MAP estimate, we select $\theta^*$ as:

$$\theta^* = \underset{\theta}{\operatorname{argmax}} \frac{p(\theta, y|X)}{p(y|X)} = \underset{\theta}{\operatorname{argmax}}\, p(\theta|y, X) = \underset{\theta}{\operatorname{argmax}} \frac{p(y|\theta, X)p(\theta)}{p(y|X)} = \underset{\theta}{\operatorname{argmax}}\, p(y|\theta, X) \cdot p(\theta)$$

This corresponds to minimizing the loss: $\text{Loss}(\theta, y, X) = -\log p(\theta, y \mid X)$, where the joint distribution is:

$$p(\theta, y \mid X) = p\Big(y \mid \big(l_{\text{per}}, \sigma^2_{\text{per}}, p,\ l_{\text{rbf}}, \sigma^2_{\text{rbf}}\big),\ X\Big) \times \underbrace{\frac{1}{l_{\text{per}}\sqrt{2\pi}}\ \exp\!\left(-\frac{\big(\ln\ l_{\text{per}}+1\big)^2}{2}\right)}_{\substack{\text{LogNormal prior for } l_{\text{per}}>0 \\ \text{with } \mu=-1,\ \sigma^2=0.7 \text{ (omitted factor)}}} \times \underbrace{\frac{1}{\sigma^2_{\text{per}}\sqrt{2\pi}}\ \exp\!\left(-\frac{\big(\ln\ \sigma^2_{\text{per}}+1\big)^2}{2}\right)}_{\substack{\text{LogNormal prior for } \sigma^2_{\text{per}}>0 \\ \text{with } \mu=-1,\ \sigma^2=0.7}}$$

$$\times \underbrace{\mathbf{1}_{\{0.3<p<1.0\}}}_{\text{Uniform prior on } p\in(0.3,1.0)} \times \underbrace{\frac{1}{l_{\text{rbf}}\sqrt{2\pi}}\ \exp\!\left(-\frac{\big(\ln\ l_{\text{rbf}}+1\big)^2}{2}\right)}_{\substack{\text{LogNormal prior for } l_{\text{rbf}}>0 \\ \text{with } \mu=-1,\ \sigma^2=0.7}} \times \underbrace{\frac{1}{\sigma^2_{\text{rbf}}\sqrt{2\pi}}\ \exp\!\left(-\frac{\big(\ln\ \sigma^2_{\text{rbf}}+1\big)^2}{2}\right)}_{\substack{\text{LogNormal prior for } \sigma^2_{\text{rbf}}>0 \\ \text{with } \mu=-1,\ \sigma^2=0.7}}.$$

(See more definitions and formulas in Appendix C.2 for details.)

### 3.4 Posterior Samples

MCMC diagnostics are essential for evaluating whether a Markov Chain Monte Carlo (MCMC) sample accurately represents the target distribution. Key aspects include verifying that most of the sample aligns with the target distribution, ensuring the sample size is adequate, and confirming that the chains have converged. Our primary diagnostic tools are trace plots (Figure 11) and autocorrelation plots (Figure 12).

We ran four independent chains, each with 500 samples and a 200-sample warm-up phase. 500 samples is a bit low but considering that we get a fair effective sample size then we will make 700 samples total do. Sampling multiple chains helps identify convergence issues and assess consistency across chains. The autocorrelation plots indicate that the samples exhibit low autocorrelation within acceptable error ranges, suggesting effective mixing and independent sampling.

More diagnostic statistics are in Tables 5 and 6. The potential scale reduction factors ($\hat{R}$) for all parameters are close to 1.00, indicating good convergence of the chains. Additionally, the Effective Sample Sizes (ESS) for the bulk parameters range from approximately 967 to 1,593, while the ESS for the tail parameters range from 517 to 778. These ESS values suggest that the sampling has adequately explored the posterior distributions, ensuring reliable parameter estimates.

Moreover, the low Monte Carlo Standard Errors (MCSE) for both the mean and standard deviation further confirm the precision and stability of the estimates obtained from the MCMC simulations. Overall, the diagnostics indicate that the MCMC sampling process was effective, with satisfactory convergence and sufficient sampling efficiency to support robust inferences about the model parameters.

Approximate posterior likelihood estimates the likelihood of the data under the posterior distribution when exact calculations are too complex. NUTS (No-U-Turn Sampler) is an advanced MCMC algorithm used to efficiently draw samples from the posterior without requiring manual tuning of parameters. By using NUTS to generate these posterior samples, we can accurately approximate the posterior likelihood through averaging the log-likelihoods across the sampled parameter values.

### 3.5 Test Log Likelihood of MAP and Sampling

To further compare the two approaches, we repeated the entire procedure on 20 independently generated datasets. Here you can see the mean and standard deviation of the test log-likelihood over these runs:

$$\text{MAP:} \quad -4.549 \pm 3.803 \qquad \text{NUTS:} \quad -5.829 \pm 1.412$$

Although the MAP approach yields a *higher average* test log-likelihood, its performance also exhibits greater variability across different seeds (as reflected by the larger standard deviation). One interpretation is that MAP can sometimes "lock onto" hyperparameters that happen to fit a particular dataset very well but may fail more dramatically for others; meanwhile, the fully Bayesian (NUTS) approach is more stable but occasionally misses the best-fitting parameters for a given dataset, leading to a slightly lower *overall* mean log-likelihood. From a modeling perspective, if one values consistent performance and robust uncertainty quantification, the NUTS-based posterior sampling may be preferable, whereas MAP can achieve stronger fits in certain cases but is also more sensitive to variation in the training data.

### 3.6 Integral Constraint

This section extends the Gaussian Process (GP) model to include an integral constraint on the function $f$. Unlike the unconstrained case, where $f$ is modeled purely by a GP prior, we now enforce that the integral of $f$ over a predefined

domain approximates a specified value. This constraint is incorporated using the Trapezoidal rule, which approximates the integral as a weighted sum of $f(x_i)$ at discrete points $x_i$. The modified model requires deriving the joint distribution of the integral approximation $\hat{q}$ and the function values $f$, and the conditional distribution of $f$ given $\hat{q}$. These derivations provide the foundation for integrating the constraint into the GP framework and adjusting the prior accordingly.

### 3.6.1 The probability distribution of $(\hat{q}, f) \mid X$

The main idea is that $\hat{q}$ is a linear combination of the discretized values of $f$, and since $f$ is jointly Gaussian at these points, any linear combination (and stacking) remains Gaussian. Recall the definition of $\hat{q}$

$$\hat{q} \;=\; \sum_{i=1}^{\ell} w_i\, f(x_i) \;=\; \mathbf{w}^{\mathsf{T}}\mathbf{f} \quad \text{where} \quad \mathbf{w} \;=\; \begin{pmatrix} w_1 \\ \vdots \\ w_\ell \end{pmatrix} \quad \text{and} \quad \mathbf{f} \;=\; \begin{pmatrix} f(x_1) \\ \vdots \\ f(x_\ell) \end{pmatrix} \in \mathbb{R}^{\ell}.$$

From the GP prior $f \sim GP(0,\, k(\cdot,\cdot))$, it follows that $\mathbf{f} \;\sim\; \mathcal{N}\big(\mathbf{0},\, \mathbf{K}\big)$, where $\mathbf{K} \in \mathbb{R}^{\ell\times\ell}$ is the covariance matrix with entries $K_{ij} \;=\; k(x_i,\, x_j)$. By a standard result about linear transformations of multivariate Gaussians (see Oswin's notes), $\mathbf{f}$ remains Gaussian under linear transformations, then it follows that

$$\begin{pmatrix} \hat{q} \\ \mathbf{f} \end{pmatrix} \;=\; \underbrace{\begin{pmatrix} \mathbf{w}^{\mathsf{T}} \\ \mathbf{I}_\ell \end{pmatrix}}_{\mathbf{T}} \mathbf{f} \;\sim\; \mathcal{N}\Big(\mathbf{T}\,\mathbf{0},\; \mathbf{T}\,\mathbf{K}\,\mathbf{T}^{\mathsf{T}}\Big) \;=\; \mathcal{N}\left(\begin{pmatrix} 0 \\ \mathbf{0} \end{pmatrix},\; \begin{pmatrix} \mathbf{w}^{\mathsf{T}}\mathbf{K}\,\mathbf{w} & \mathbf{w}^{\mathsf{T}}\mathbf{K} \\ \mathbf{K}\,\mathbf{w} & \mathbf{K} \end{pmatrix}\right).$$

where $\mathbf{I}_\ell$ is the $\ell \times \ell$ identity matrix and $\mathbf{T}$ is $(\ell+1)\times\ell$. Thus, we have shown $\big(\hat{q},\; f\big) \;\big|\; X$ is indeed jointly normal.

### 3.6.2 The probability distribution of $f \;\big|\; X, \hat{q}$

We assume that $k(\cdot,\cdot)$ is a universal kernel which produces a strictly positive definite matrix when evaluated at distinct points according to the definition in Oswin's notes. Hence, $\mathbf{K}$ is positive definite and, as a result, invertible, since its null space contains only the zero vector, ensuring full rank.

By the standard Gaussian conditioning formula (see Oswin's notes) and the result of the previous subsection 3.6.1,

$$\mathbf{f} \,\big|\, \hat{q} \;\sim\; \mathcal{N}\Big(\boldsymbol{\mu},\, \boldsymbol{\Sigma}\Big),$$

where

$$\boldsymbol{\mu} \;=\; \mathbf{K}\,\mathbf{w}\Big(\mathbf{w}^{\mathsf{T}}\mathbf{K}\,\mathbf{w}\Big)^{-1}\hat{q} \quad \text{and} \quad \boldsymbol{\Sigma} \;=\; \mathbf{K} \;-\; \mathbf{K}\,\mathbf{w}\Big(\mathbf{w}^{\mathsf{T}}\mathbf{K}\,\mathbf{w}\Big)^{-1}\mathbf{w}^{\mathsf{T}}\,\mathbf{K}.$$

To find the rank of $\boldsymbol{\Sigma}$, we rewrite it as

$$\boldsymbol{\Sigma} \;=\; \underbrace{\mathbf{K}}_{\text{invertible}} \;-\; (\mathbf{K}\,\mathbf{w})\,\underbrace{\Big(\mathbf{w}^{\mathsf{T}}\mathbf{K}\,\mathbf{w}\Big)^{-1}}_{\text{scalar}}\,(\mathbf{K}\,w)^{\mathsf{T}}$$

The second term has rank 1 as it is the outer product of the vectors $\mathbf{K}\,\mathbf{w}$ and $\big(\mathbf{K}\,\mathbf{w}\big)^{\mathsf{T}}$, with all row vectors parallel. Subtracting a rank-1 update from an invertible matrix reduces its rank by 1 (see Appendix C.1). Thus, $\boldsymbol{\Sigma}$ has rank $\ell - 1$. Equivalently, fixing $\hat{q}$ removes one linear degree of freedom in $\mathbf{f}$, yielding a degenerate (rank-$\ell - 1$) covariance.

This completes the derivation and shows that $f \;\big|\; X,\, \hat{q}$ is Gaussian with a rank-deficient covariance if $k$ is universal.

## 3.7 Constrained GP Results

This section presents the results of applying an integral constraint to the Gaussian Process (GP) model. The difference between constrained and unconstrained approaches is illustrated through both sampled functions (no data points fixed) and posterior distributions after observing a small dataset $\mathcal{D}$.

**Samples from $\mathbf{f} \mid X, \hat{q}$:** Figure 3 shows five samples from the conditional distribution $\mathbf{f} \mid X, \hat{q}$ for three integral constraint values, $\hat{q} \in \{0, 5, 10\}$. We overlay a *mean* (black line) and *95% confidence interval* to highlight how the function mean and uncertainty change with $\hat{q}$. In accordance with the Gaussian conditioning formula (see Section 3.6.1), the covariance matrix remains independent of $\hat{q}$. Consequently, only the mean vector scales up (or down) to achieve the specified integral value.

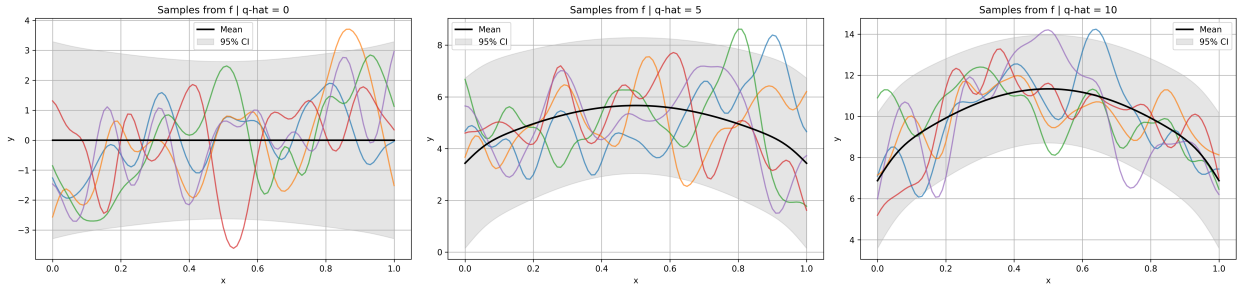


Figure 3: Samples from $f \mid X, \hat{q}$ for different values of $\hat{q}$. Each panel shows five random samples and the corresponding mean function (black line) along with confidence intervals.

As $\hat{q}$ increases, the mean shifts upward so the total area under the curve matches $\hat{q}$. One might initially think the variance shrinks for higher $\hat{q}$, but careful inspection reveals that the confidence band is simply offset to a higher $y$-range. Indeed, if we subtract out the new mean, the shape of the individual draws (and thus the covariance) is unchanged. The broader vertical extent in Figure 3 makes the confidence interval appear relatively tighter for higher $\hat{q}$, though the actual scale of the function has simply shifted. In practice, without any fixed data points, the GP has considerable freedom, so the samples still exhibit large wiggles to reflect high uncertainty away from observed constraints.

**Comparison of Posteriors:** Figure 4 contrast the posterior distributions $\mathbf{f} \mid \mathcal{D}$ (unconstrained) versus $\mathbf{f} \mid \mathcal{D}, \hat{q}$ (constrained, with $\hat{q} = 2$). Here, $\mathcal{D} = \{(0, 1.46), (0.25, 0.93), (0.5, 2.76)\}$ and the true function $g(x)$ is overlaid (dashed black line). We also plot the mean functions, the $\pm 1.95\sqrt{v(x)}$ confidence bands, and the data points.

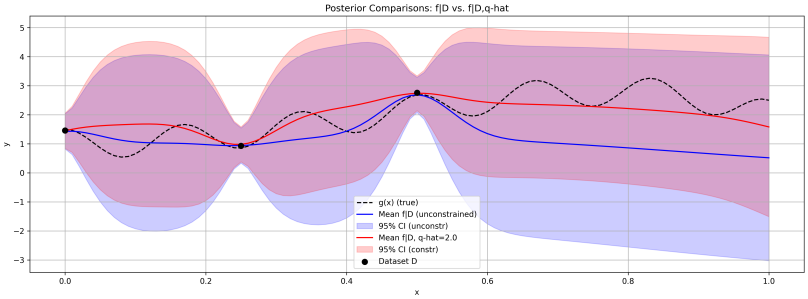


Figure 4: Comparison of posterior distributions with and without the integral constraint. Observed data points are shown as black dots, with the true function $g(x)$ overlaid as a dashed line. Shaded areas represent $m(X) \pm 1.95\sqrt{v(X)}$.

We see that both posteriors fit the data points in the immediate vicinities of $x = 0$, $x = 0.25$, and $x = 0.5$. However, the constrained model's mean curve is visibly adjusted to achieve an integral near 2.0, which leads it to remain closer to the true function $g(x)$ on average (notably around $x > 0.6$). This can reduce overall posterior variance where the integral constraint exerts the most influence, though we still see considerable uncertainty because only three data points are observed. In addition, we rely on the MAP kernel hyperparameters found earlier (Section 3.1), which were originally optimized for a 20-point dataset; thus, with only three points, the GP may not achieve an ideal fit but does highlight how additional integral constraints help shape the posterior.

**Discussion:** Incorporating an integral constraint enables the GP to capture extra global information about the function. When $\hat{q}$ is reasonably accurate (e.g. around 2, while the true integral is closer to 3), the constraint guides the mean function closer to the true curve. Without any fixed data points, the GP is still highly uncertain, but the integral constraint alone can shift the entire function upward or downward to match a target area. When combined with even a small dataset $\mathcal{D}$, this approach can significantly alter the posterior mean to achieve a better compromise between local observations and global integrals. As expected, with only three data points, the model's overall fit remains imperfect. Nonetheless, these results illustrate how the integral constraint can be a powerful way to incorporate prior knowledge about function integrals into GP regression.

## 4 Conclusion

This report explored the implementation and evaluation of diffusion-based generative models for MNIST and Gaussian Processes (GPs) with integral constraints. The diffusion model analysis demonstrated that incorporating classifier-free guidance significantly improved image quality and diversity. In the GP study, the Periodic + RBF kernel produced the best function-fitting results, and adding integral constraints effectively guided the model toward the true function.

# A Implementation Details

## A.1 Adjustments to the Reverse Diffusion Step in Variant 1

The reverse diffusion step was modified to reflect the change in parameterization, i.e., predicting the original image $x_0$ instead of the noise $\epsilon$. The general formula for the mean of the reverse process in DDPMs is:

$$\mu_t = \frac{\sqrt{\alpha_{t-1}}\beta_t}{1-\bar{\alpha}_t}x_0 + \frac{\sqrt{\alpha_t}(1-\bar{\alpha}_{t-1})}{1-\bar{\alpha}_t}x_t$$

In the standard DDPM, the model predicts $\epsilon$, and $x_0$ is derived as:

$$\hat{x}_0 = \frac{x_t - \sqrt{1-\bar{\alpha}_t}\hat{\epsilon}}{\sqrt{\bar{\alpha}_t}}.$$

Substituting this into the general formula gives:

$$\mu_t = \frac{1}{\sqrt{\alpha_t}}\left(x_t - \frac{\beta_t}{\sqrt{1-\bar{\alpha}_t}}\hat{\epsilon}\right).$$

For this variant, the model predicts $x_0$ directly. Substituting $\hat{x}_0$ into the general mean formula results in:

$$\mu_t = \sqrt{\bar{\alpha}_t} \cdot \hat{x}_0 + \sqrt{1-\bar{\alpha}_t} \cdot \text{noise}.$$

This modification allows the reverse diffusion process to directly combine the predicted clean image with a noise component scaled by the timestep-dependent variance.

## A.2 Code Changes for Variant 1

The implementation for Variant 1 modifies the standard DDPM model to predict the original image $x_0$ instead of the noise $\epsilon$. The changes primarily affect the reverse diffusion process and the ELBO loss computation.

**Key Modifications:**

1. `reverse_diffusion`: The original implementation computes the mean for $\mathbf{x}_{t-1}$ based on the predicted noise $\epsilon$. In Variant 1, this step is adjusted to use the predicted $x_0$:

```
if self.predict_x0:
    x0_pred = self.network(xt, t)
    mean = torch.sqrt(self.alpha_bar[t]) * x0_pred + \
           torch.sqrt(1 - self.alpha_bar[t]) * torch.randn_like(xt)
else:
    epsilon_pred = self.network(xt, t)
    mean = (1. / torch.sqrt(self.alpha[t]) *
            (xt - self.beta[t] / torch.sqrt(1 - self.alpha_bar[t]) * epsilon_pred))
```

2. `elbo_simple`: The loss function is updated to reflect the new prediction target ($x_0$ instead of $\epsilon$). If the model predicts $x_0$, the loss directly compares the predicted and true $x_0$:

```
if self.predict_x0:
    x0_pred = self.network(xt, t)
    return -nn.MSELoss()(x0_pred, x0)
else:
    epsilon_pred = self.network(xt, t)
    return -nn.MSELoss()(epsilon_pred, epsilon)
```

3. `predict_x0` Flag: A boolean flag `self.predict_x0` is added to toggle between the two modes (predicting $x_0$ or $\epsilon$). This ensures compatibility with the baseline model.

```
    self.predict_x0 = predict_x0
```

**Impact of Changes:** By predicting $x_0$, the reverse diffusion step integrates the predicted clean image directly into the mean computation. This modification aligns the training objective with the end goal of generating $x_0$. However, it didn't improve the performance as can be seen in 2.2 and plots in B.1. This is likely due to the training being less stable similar to how residual networks simplify the training objective.

### A.3 Code changes for Variant 2

When implementing low-discrepancy sampling we use the same overall idea as outlined in Appendix I.1 in the Kingma et al, 2021 paper. The idea is to instead of simply sampling every timestep at random we sample one timestep in the batch at random and then choose the remaining timesteps in the batch based on this intial time step. The remaining timesteps are sampled to be spread in the space [1,T] with increments equal to the amount of samples in the batch. The following formula is used: $t_i = \left\lfloor \text{mod}\left(u_0 + \frac{i}{k}, 1\right) \cdot T \right\rfloor + 1$

with $u_i$ being the random timestep, $i$ being the number of the current sample in this batch, $k$ being the amount of samples in the batch and $T$ being our range. This way we train our model on timestep samples from more different timesteps than if we use random sampling since then each sample is in fact random and for any given batch the timesteps samples could be anywhere in the range $[1, T]$ and could be very close together. Exposing the model to more different timesteps in each batch could lead to more stabile training and reduce variance in our ELBO estimate. This is mostly a result of having very different sampled timesteps from epoch to epoch will greatly change the gradients and what part of the denoising process the model learns best. In terms of code we will see changes matching the explanation above and utilizing the formula mentioned in code.

```
def elbo_simple(self, x0):
    batch_size = x0.shape[0]
    with torch.no_grad():
        u0 = torch.rand(1).to(x0.device)
        i  = torch.arange(batch_size, device=x0.device, dtype=torch.float32)

        new_t  = (u0 + i / batch_size) % 1.0
        t_vals = torch.floor(new_t * self.T) + 1
        t_vals = t_vals.clamp(1, self.T).long()
        t_vals = t_vals.unsqueeze(-1)

    epsilon = torch.randn_like(x0)
    xt = self.forward_diffusion(x0, t_vals, epsilon)
    return -nn.MSELoss(reduction='mean')(epsilon, self.network(xt, t_vals))
```

For clarity we can contrast the elbo_simple function to the original function as defined in the solution given:

```
def elbo_simple(self, x0):
    t = torch.randint(1, self.T, (x0.shape[0],1)).to(x0.device)
    epsilon = torch.randn_like(x0)
    xt = self.forward_diffusion(x0, t, epsilon)
    return -nn.MSELoss(reduction='mean')(epsilon, self.network(xt, t))
```

As we can see clearly now previously we simply assign t a random integer from the range [1,T] while in the new code for the low-discrepancy sampling code above we are only randomly choosing a starting point for the batch before sampling the rest spaced out in accordance with the amount of samples in the batch.

### A.4 Code Changes for Variant 3

When implementing importance sampling we followed the explanation given in Section 3.3 in Nichol & Dhariwal 2021. The simple notion behind this method is the idea that the timesteps at which the model has done a poor job at predicting previously are timesteps it should be exposed to more. And this makes a lot of sense intuitively as some timesteps might be more difficult for the model and hence training on these more often than easier ones makes a lot of sense. The way this is implemented is by introducing a memory of previous sampled timesteps and the ensuing loss of the model. We do this with normal random sampling until we have 10 samples of every timestep and the corresponding losses that the model had when it was trained on these timesteps. Now we average these 10 losses for each timestep and use these values as weights for sampling future time steps, making sure to keep this loss memory updated throughout continued training. The formula for the probability for choosing a specific timestep was expressed in the original paper like this:

$$L_{\text{vlb}} = \mathbb{E}_{t\sim p_t}\left[\frac{L_t}{p_t}\right], \quad \text{where} \quad p_t \propto \sqrt{\mathbb{E}[L_t^2]} \quad \text{and} \quad \sum p_t = 1$$

Here we have first redefine the variational lower bound (our loss), $L_{\text{VLB}}$, to be scaled inversely with the frequency at which it is sampled such that we compensate for not having uniform sampling. The second equation is saying that the probability of sampling a given timestep, $p_t$, is proportional to the square root of the squared loss.

Looking at the code itself the key changes are how we sample t. We first and foremost have to draw t from the distribution we created based on previous losses:

```
sampled_t = np.random.choice(T+1, size=(x.shape[0],), p=p)
t_tensor = torch.tensor(sampled_t, device=device).unsqueeze(1)
t_tensor = torch.clamp(t_tensor, 1, T)
```

Then we also have to create this distribution and we do this by using this update distribution function:

```
def update_distribution():
    avg_l2 = np.array([np.mean(lt) for lt in loss_track])
    avg_l2[0] = 0.0 #ignore since we won't use t0
    w = np.sqrt(avg_l2)
    w /= w.sum()
    return w
```

We collect the previous loss values in our training script with a deque:

```
loss_track = [deque([1e-2]*hist_len, maxlen=hist_len) for _ in range(T+1)]
```

We track the losses using this for loop inside of the enumerate(dataloader) loop:

```
for idx, tval in enumerate(sampled_t):
    loss_track[tval].append(per_sample_loss[idx].item()**2)
```

This still required us to have a large enough dequeu of previous losses but we do not have this when we start training our model and we therefore use random sampling of t values like previously like we did in the original code, but only until we have enough loss values to create the timestep distribution we want to sample from:

```
    def elbo_simple(self, x0, t=None):
        if t is None:
            t = torch.randint(1, self.T, (x0.shape[0],1), device=x0.device)

        epsilon = torch.randn_like(x0)
```

```
        xt = self.forward_diffusion(x0, t, epsilon)
        mse = F.mse_loss(self.network(xt, t), epsilon, reduction='mean')
        return mse
```

### A.5 Code Changes for Variant 4

This variant implements *Classifier-Free Diffusion Guidance* (CFG), which combines conditional and unconditional predictions to improve diversity and fidelity in generated samples. Key modifications include conditioning the network on class labels, applying label dropout during training, and dynamically combining predictions during inference.

**Key Modifications:**

1. `reverse_diffusion`: The reverse diffusion process was updated to compute both conditional and unconditional predictions. These predictions are combined using the CFG formula:

```
    pred_cond = self.network(xt, t, label)
    label_uncond = -torch.ones_like(label)  # -1 for ignored class labels
    pred_uncond = self.network(xt, t, label_uncond)
    pred = (1 + w) * pred_cond - w * pred_uncond # w = 2.0
```

   This ensures the generated images can balance diversity (via `w`) and alignment with the class label.

2. `sample`: Added functionality for conditional sampling. If no label is provided, the model defaults to unconditional sampling by setting the label to `-1`.

```
    label = -torch.ones((shape[0],), device=self.beta.device) if label is None else label
```

3. `elbo_simple`: The ELBO loss function was modified to incorporate labels:

```
    xt = self.forward_diffusion(x0, t, epsilon)
    return -nn.MSELoss()(epsilon, self.network(xt, t, label))
```

4. `train`: During training, labels are set to $-1$ for a fraction (`puncond`) of training steps to simulate unconditioned predictions and improve robustness:

```
    use_condition = torch.rand(x.shape[0], device=x.device) < 1 - puncond # 1 - 0.1 = 0.9
    label = torch.where(use_condition, label, -torch.ones_like(label))
```

### A.6 Implementation of FID

We validated our VAE's representative latent space using a t-sne plot which can be seen in figure 5. Here is the coding implementation.

```
def compute_fid_vae(vae, real_loader, diffusion_model, device, num_gen=10000):
    vae.eval()
    diffusion_model.eval()

    real_features = []
    for x, _ in real_loader:
        feats = get_vae_features(vae, x, device=device)
        real_features.append(feats)
```

```
    real_features = np.concatenate(real_features, axis=0)

    gen_features = []
    batch_size = 128
    rounds = num_gen // batch_size
    for _ in range(rounds):
        with torch.no_grad():
            samples = diffusion_model.sample((batch_size, 28*28)).to(device)
            samples = (samples + 1)/2
            samples = samples.clamp(0,1).view(batch_size, 1, 28, 28)
        feats = get_vae_features(vae, samples, device=device)
        gen_features.append(feats)
    gen_features = np.concatenate(gen_features, axis=0)

    mu_r = np.mean(real_features, axis=0)
    sigma_r = np.cov(real_features, rowvar=False)

    mu_g = np.mean(gen_features, axis=0)
    sigma_g = np.cov(gen_features, rowvar=False)

    eps = 1e-6
    diff = mu_r - mu_g
    diff_squared = diff.dot(diff)

    return diff_squared, sigma_g, sigma_r

diff_squared, sigma_r, sigma_g = compute_fid_vae(
    vae=vae_model,
    real_loader=dataloader_test,
    diffusion_model=model,
    device=device_vae,
    num_gen=10000
)
covmean = scipy.linalg.sqrtm(sigma_r.dot(sigma_g))
if np.iscomplexobj(covmean):
    covmean = covmean.real

fid_val = diff_squared + np.trace(sigma_r + sigma_g - 2 * covmean)

eps = 1e-6
sigma_r_add = sigma_r + eps * np.eye(sigma_r.shape[0])
sigma_g_add = sigma_g + eps * np.eye(sigma_g.shape[0])
covmean_add = scipy.linalg.sqrtm(sigma_r_add.dot(sigma_g_add))
if np.iscomplexobj(covmean):
    covmean_add = covmean_add.real
fid_val_add = diff_squared + np.trace(sigma_r_add + sigma_g_add - 2 * covmean_add)
```

**Changes to the Neural Network:**

To handle labels, the network was updated with an embedding layer for class labels. The embeddings are concatenated with time embeddings:

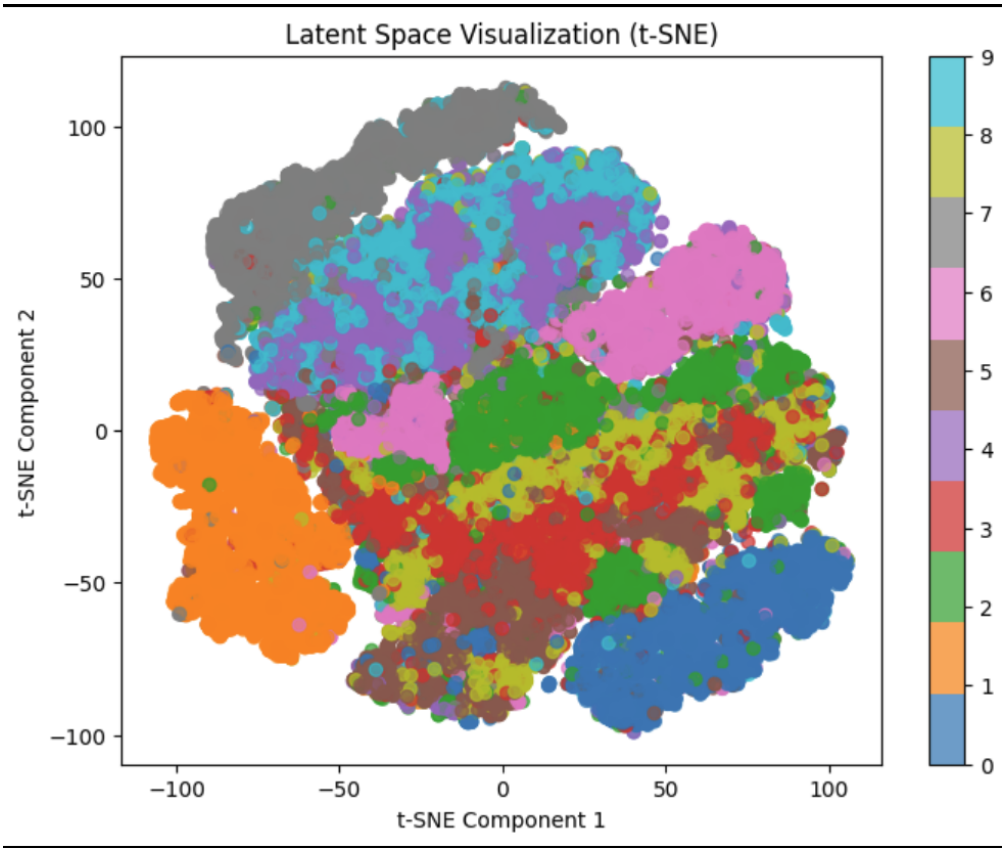


Figure 5: T-sne plot of our VAE trained on the training set which was used to train the DDPMs

```
self.label_embed = nn.Embedding(num_labels, embed_dim)
embed = torch.cat([embed_t, self.label_embed(label)], dim=-1)
```

A mask was added to zero out embeddings for `-1` labels during unconditioned sampling:

```
label_mask = (label.unsqueeze(-1) != -1).float()
label_embedding = label_embedding * label_mask
```

**Dynamic Thresholding:** During sampling, pixel values were dynamically thresholded to improve the visual quality of generated images:

```
percentile = torch.quantile(samples.flatten(start_dim=1), 0.995, dim=1, keepdim=True)
samples = torch.clamp(samples, -percentile, percentile)
samples = (samples - samples.min()) / (samples.max() - samples.min() + 1e-8)
```

**Future Work:** Future improvements could explore techniques like *CADS* (Condition Annealed Diffusion Sampling), which improves generation diversity. However, it might be more relevant for a more complex generative task like text-to-image generation or ImageNet class generation.

# B Additional Figures and Results

## B.1 Diffusion Generated Images

This section includes figures showing 10 generated samples for each model variation after training for 100 epochs.

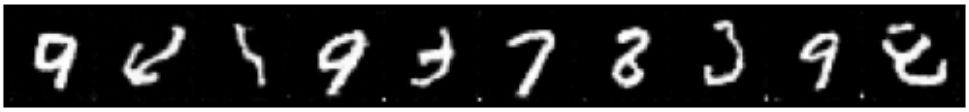

Figure 6: Generated samples from the Baseline model. Step size $T = 1000$

Figure 7: Generated samples from Variant 1: Predicting original image. Step size $T = 1000$

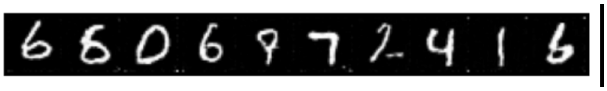

Figure 8: Generated samples from Variant 2: Lower Discrepancy sampling of timesteps $t$. Step size $T = 1000$

Figure 9: Generated samples from Variant 3: Importance Sampling of timesteps. Step size $T = 1000$

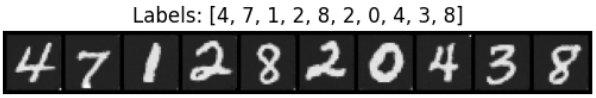


Figure 10: Generated samples from Variant 4: Classifier-free guidance. The samples are conditioned on class labels. Step size $T = 1000$

Variant 4 performed the best, likely because it had more information in the network's input—it knew the class label it was supposed to predict. This additional context helps the model distinguish between images. For instance, having nine separate models, each specialized for one class, would make training easier compared to a single generative model tasked with generating all classes.

Variant 1 performed the worst, producing the least diverse and least realistic examples. Variants 2, 3, and the baseline were comparable in quality, though we believe Variant 2 performed the best, producing more realistic examples (despite sampling four instances of the digit "6"). Variant 3 followed closely, while the baseline came last among these three, as it only produced four realistic examples, three of which were "9s."

**Performance Ranking (Visual Quality):**

1. Variant 4
2. Variant 2
3. Variant 3
4. Baseline
5. Variant 1

## B.2 Kernel Selection

In this section, we summarize the kernel selection process for Gaussian Process (GP) models using four kernel combinations: Periodic + RBF, Periodic × RBF, Periodic + Linear, and Periodic×Linear. The models were trained using different optimization methods: Adam, Adam with resets, grid search, and random search. The evaluation metrics include log-likelihood (LL), mean squared error (MSE), and root mean squared error (RMSE).

The results for each optimization method are presented below:

As we can see the Periodic + RBF achieves the best performance, with the highest log likelihood value, lowest MSE and RMSE values, and it strangely happened when using Adam without any resets. Our Adam resets code selected the parameters that had the lowest last iteration loss, maybe we should have selected the lowest loss from all iterations instead? However, if we compare the optimization methods we can see that regular Adam seemed to be the best one. Even though the course slides suggested gradient descent gets often stuck, but Adam might just be good enough for this dataset. That's why we will use it for the rest of part B or Section 3 of this report.

| Kernel | Adam | | | Adam+Resets | | |
|---|---|---|---|---|---|---|
| | LL | MSE | RMSE | LL | MSE | RMSE |
| Periodic + RBF | **-1.223** | **0.0561** | **0.2368** | **-7.340** | **0.1639** | **0.4049** |
| Periodic × RBF | -13.162 | 0.6883 | 0.8297 | -10.893 | 0.5751 | 0.7583 |
| Periodic + Linear | -8.296 | 0.3082 | 0.5552 | -14.073 | 0.3800 | 0.6165 |
| Periodic × Linear | -18.551 | 0.4858 | 0.6970 | -21.313 | 0.5394 | 0.7344 |

Table 2: Kernel selection results using Adam and Adam with resets.

| Kernel | Grid Search | | | Random Search | | |
|---|---|---|---|---|---|---|
| | LL | MSE | RMSE | LL | MSE | RMSE |
| Periodic + RBF | -73.722 | 0.2557 | 0.5057 | **-17.404** | 0.5490 | 0.7409 |
| Periodic × RBF | **-63.044** | **0.2112** | **0.4596** | -21.518 | 0.6723 | 0.8199 |
| Periodic + Linear | -74.492 | 0.2568 | 0.5068 | -75.648 | **0.4730** | **0.6877** |
| Periodic × Linear | -126.221 | 0.3192 | 0.5650 | -59.473 | 0.6723 | 0.8199 |

Table 3: Kernel selection results using grid search and random search.

### B.3 Cross-Validation

Cross-validation was performed with three optimization methods: Adam, Adam with resets (5 resets), and random search. The data was split into 30 folds (Leave-One-Out-Cross-Validation), and the average results are shown below:

| Kernel | Adam | | | Adam+Resets | | | Random Search | | |
|---|---|---|---|---|---|---|---|---|---|
| | LL | MSE | RMSE | LL | MSE | RMSE | LL | MSE | RMSE |
| Periodic + RBF | -0.744 | 0.1742 | **0.3141** | -1.175 | 0.1970 | **0.3486** | **-3.233** | **0.3090** | **0.4402** |
| Periodic × RBF | -1.014 | 0.3426 | 0.4534 | **-0.973** | **0.1952** | 0.3491 | -8.427 | 0.4165 | 0.5408 |
| Periodic + Linear | **-0.654** | **0.1490** | 0.3253 | -1.349 | 0.2478 | 0.3879 | -4.070 | 0.3486 | 0.4763 |
| Periodic × Linear | -1.589 | 0.4421 | 0.5596 | -2.864 | 0.4348 | 0.5416 | -16.096 | 0.4254 | 0.5474 |

Table 4: Cross-validation results for 30 folds.

The results demonstrate that Adam optimization generally achieves better log-likelihood, MSE and RMSE compared to other methods. Adam with resets occasionally outperforms for specific kernels, but at the cost of increased computation.

### B.4 NUTS run

| Parameter | Mean | SD | HDI_3% | HDI_97% | MCSE_Mean | MCSE_SD | ESS_Bulk | ESS_Tail | $\hat{R}$ |
|---|---|---|---|---|---|---|---|---|---|
| kernel.kern0.lengthscale | 0.378 | 0.017 | 0.346 | 0.409 | 0.000 | 0.000 | 1354.0 | 722.0 | 1.01 |
| kernel.kern0.period | 0.667 | 0.003 | 0.661 | 0.673 | 0.000 | 0.000 | 1593.0 | 778.0 | 1.02 |
| kernel.kern0.variance | 0.297 | 0.106 | 0.121 | 0.485 | 0.003 | 0.003 | 1100.0 | 716.0 | 1.00 |
| kernel.kern1.lengthscale | 0.270 | 0.065 | 0.152 | 0.394 | 0.002 | 0.002 | 982.0 | 657.0 | 1.01 |
| kernel.kern1.variance | 0.664 | 0.389 | 0.158 | 1.316 | 0.015 | 0.012 | 967.0 | 517.0 | 1.00 |

Table 5: Posterior Summary

Table 6: Effective Sample Size (ESS)

| Parameter | ESS |
|---|---|
| kernel.kern0.lengthscale | $1.354 \times 10^3$ |
| kernel.kern0.period | $1.593 \times 10^3$ |
| kernel.kern0.variance | $1.100 \times 10^3$ |
| kernel.kern1.lengthscale | 981.6 |
| kernel.kern1.variance | 966.7 |

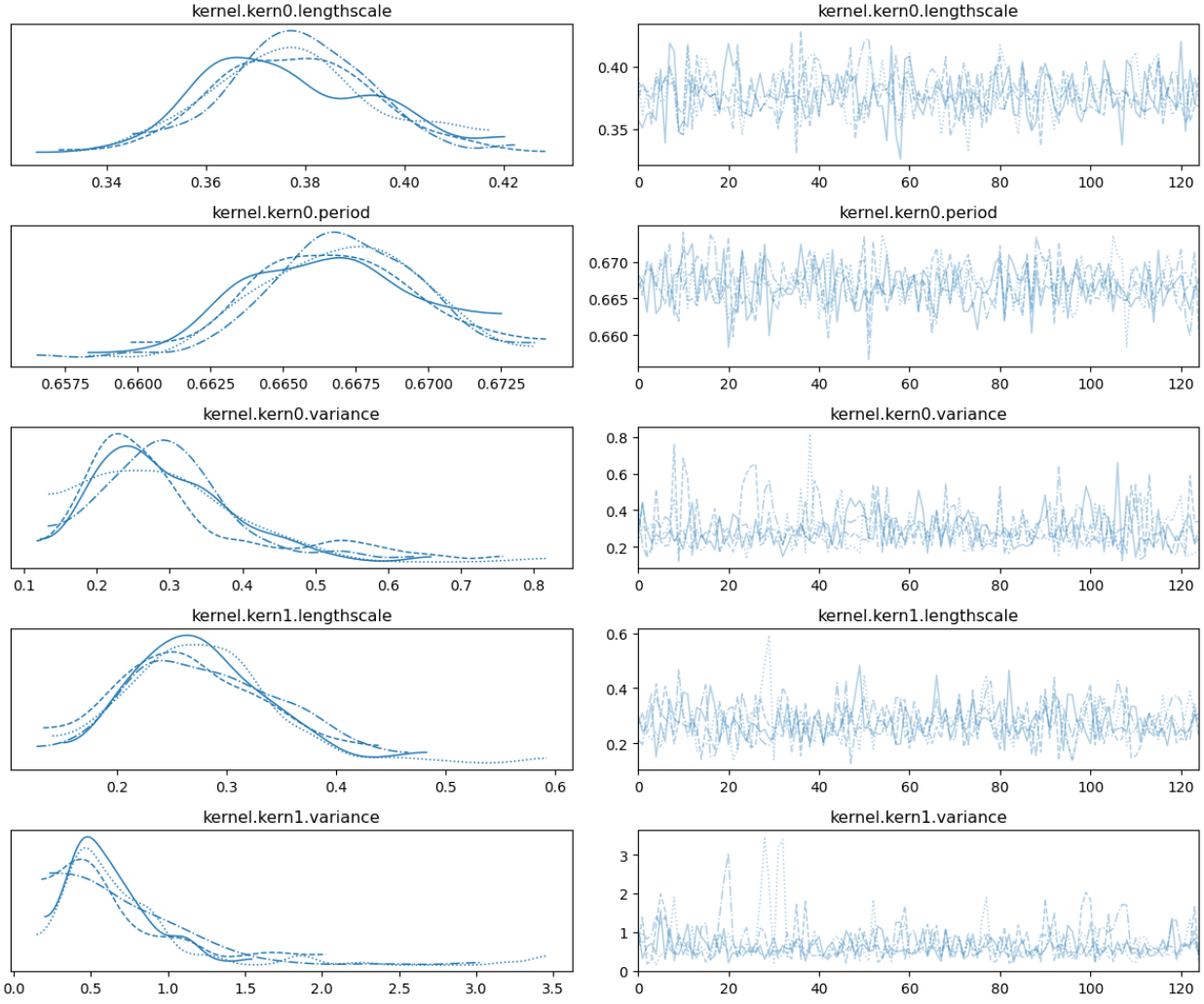


Figure 11: Trace Plots from Markov Chain Sampling B.1

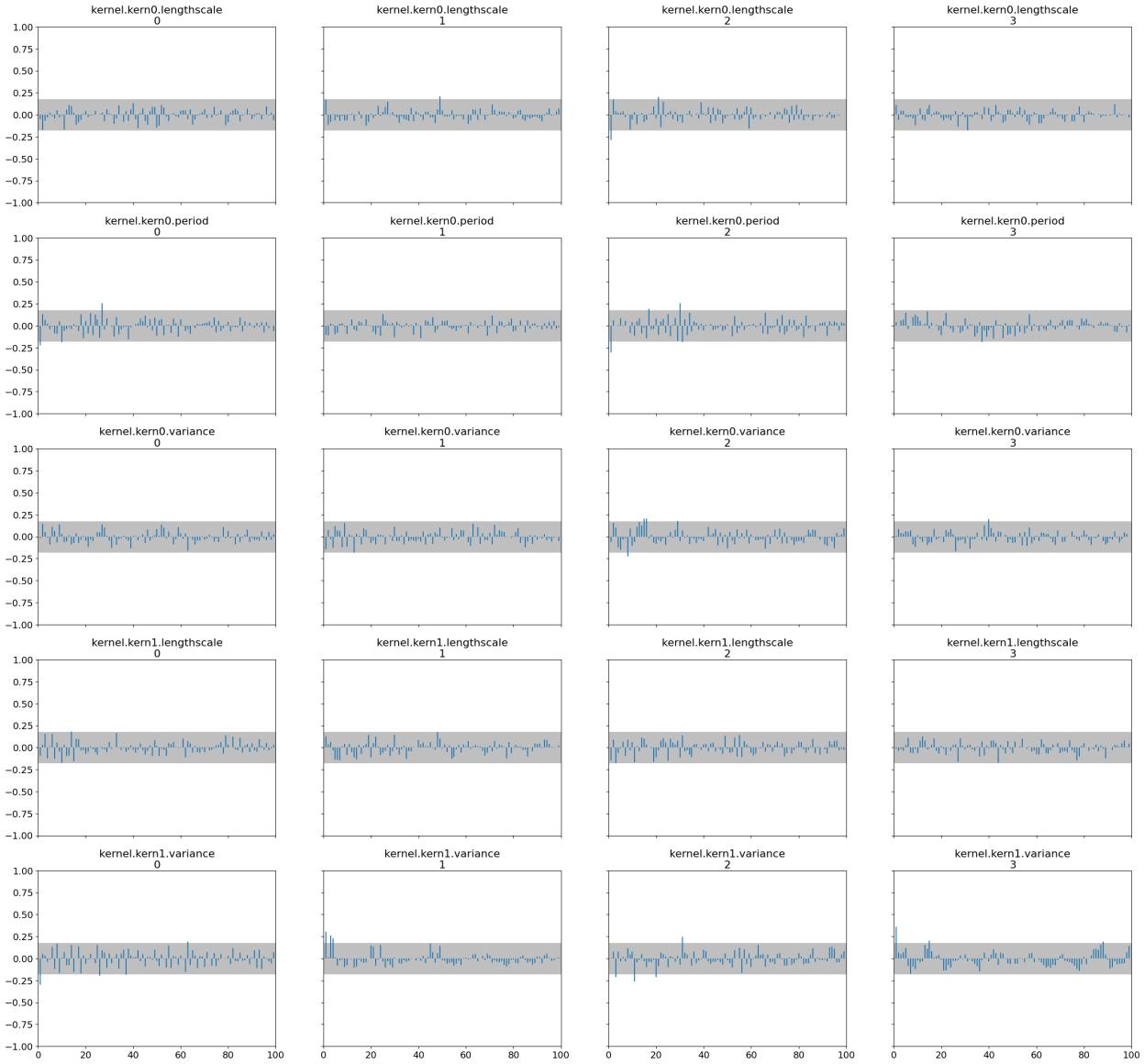


Figure 12: Autocorrelation Plots from Markov Chain Sampling B.1

# C Mathematics

## C.1 Proof of Rank n - 1 = Rank n - Rank 1

Let $\mathbf{K} \in \mathbb{R}^{\ell \times \ell}$ be invertible, and let $\mathbf{u} \in \mathbb{R}^{\ell}$ be nonzero. Consider

$$\mathbf{B} \;=\; \mathbf{K} \;-\; \alpha\,\mathbf{u}\,\mathbf{u}^{\mathsf{T}} \quad (\alpha \in \mathbb{R} \setminus \{0\}).$$

We claim $\operatorname{rank}(\mathbf{B}) = \ell - 1$.

i) **$\mathbf{B}$ is not invertible:** We will show $\mathbf{B}$ has a nontrivial null vector. Since $\mathbf{K}$ is invertible, define

$$\mathbf{v} \;=\; \mathbf{K}^{-1}\mathbf{u}.$$

Then

$$\mathbf{B}\,\mathbf{v} \;=\; \Big(\mathbf{K} - \alpha\,\mathbf{u}\,\mathbf{u}^{\mathsf{T}}\Big)\,\mathbf{v} \;=\; \underbrace{\mathbf{K}\mathbf{v}}_{=\mathbf{u}} \;-\; \alpha\,\mathbf{u}\,\mathbf{u}^{\mathsf{T}}\mathbf{v}.$$

Observe that $\mathbf{u}^{\mathsf{T}}\mathbf{v} = \mathbf{u}^{\mathsf{T}}(\mathbf{K}^{-1}\mathbf{u})$ is just a scalar. If we choose $\alpha$ to make $\alpha\,\mathbf{u}^{\mathsf{T}}\mathbf{v} = 1$, then $\mathbf{B}\,\mathbf{v} = \mathbf{u} - \mathbf{u} = \mathbf{0}$. Thus $\mathbf{v} \neq \mathbf{0}$ is in the kernel of $\mathbf{B}$, making $\mathbf{B}$ singular.

ii) **The kernel is exactly 1-dimensional:** Because $\mathbf{K}$ was invertible (rank $\ell$), subtracting one "direction" $\mathbf{u}\mathbf{u}^{\mathsf{T}}$ drops the rank by at most 1. Having found one explicit nonzero null vector of $\mathbf{B}$ shows that the rank really does drop by 1 (not 0). Also if $\operatorname{rank}(\mathbf{B}) \;<\; \ell - 1$. then we get a contradiction from the rank-sum inequality:

$$\ell = \operatorname{rank}(\mathbf{K}) = \operatorname{rank}\Big(\mathbf{B} \;+\; \alpha\,\mathbf{u}\,\mathbf{u}^{\mathsf{T}}\Big) \leq \operatorname{rank}\big(\mathbf{B}\big) + \operatorname{rank}\Big(\alpha\,\mathbf{u}\,\mathbf{u}^{\mathsf{T}}\Big) < (\ell - 1) + 1 = \ell.$$

Hence

$$\operatorname{rank}(\mathbf{B}) \;=\; \ell - 1.$$

Applying to $\mathbf{\Sigma} = \mathbf{K} - \mathbf{K}\mathbf{w}\big(\mathbf{w}^{\mathsf{T}}\mathbf{K}\mathbf{w}\big)^{-1}\mathbf{w}^{\mathsf{T}}\mathbf{K}$. Set $\mathbf{u} = \mathbf{K}\mathbf{w}$ and $\alpha = \big(\mathbf{w}^{\mathsf{T}}\mathbf{K}\mathbf{w}\big)^{-1}$.

i) $\mathbf{K}$ is invertible (because $k$ is a universal kernel, implying $\mathbf{K}$ is positive definite for distinct $x_i$).

ii) $\mathbf{u} \neq \mathbf{0}$ as $\mathbf{w}^{\mathsf{T}}\mathbf{K}\mathbf{w} \neq 0$ because $\mathbf{K}$ is positive definite and $\mathbf{w} \neq \mathbf{0}$.

Therefore $\mathbf{\Sigma}$ has rank $\ell - 1$. This completes the proof.

## C.2 Gaussian Process Definitions and formulas

$$\textbf{Periodic Kernel:} \quad K_{ij} = \sigma^2 \exp\Big[-\frac{2\,\sin^2\big(\frac{\pi|x_i - x_j|}{p}\big)}{l^2}\Big],$$

$$\textbf{RBF Kernel:} \quad K_{ij} = \sigma^2 \exp\Big[-\frac{\big(x_i - x_j\big)^2}{2l^2}\Big],$$

$$\mathbf{K} = \begin{pmatrix} K_{11} & K_{12} & \cdots & K_{1N} \\ K_{21} & K_{22} & \cdots & K_{2N} \\ \vdots & \vdots & \ddots & \vdots \\ K_{N1} & K_{N2} & \cdots & K_{NN} \end{pmatrix}.$$

$$\textbf{GP Likelihood:} \quad p(y \mid \theta, X) = \frac{1}{(2\pi)^{N/2}\,|\mathbf{K}|^{1/2}} \exp\Big(-\tfrac{1}{2}\,y^{\top}\mathbf{K}^{-1}y\Big).$$

$$\textbf{Independent Priors:} \quad \begin{aligned} p(l) &= \frac{1}{l\sqrt{2\pi}} \exp\Big[-\tfrac{(\ln l + 1)^2}{2}\Big] \quad (l > 0), \\ p(\sigma^2) &= (\sigma^2)^{-2} \exp\Big[-\tfrac{1}{\sigma^2}\Big] \quad (\sigma^2 > 0), \\ p(p) &= 1 \quad (0 < p < 1), \quad 0 \text{ otherwise.} \end{aligned}$$

$$\textbf{Joint Prior:} \quad p(\theta) = p(l, \sigma^2, p) = p(l)\,p(\sigma^2)\,p(p).$$

**Final Joint:**

$$p(\theta, y \mid X) = \underbrace{\frac{1}{(2\pi)^{N/2}|\mathbf{K}|^{1/2}} \exp\Big(-\tfrac{1}{2} y^\top \mathbf{K}^{-1} y\Big)}_{\text{GP Likelihood}} \times \underbrace{\frac{1}{l_{\text{per}}\sqrt{2\pi}} \exp\left(-\frac{\left(\ln l_{\text{per}}+1\right)^2}{2}\right)}_{\substack{\text{LogNormal prior for } l_{\text{per}}>0 \\ \text{with } \mu=-1,\ \sigma^2=0.7 \text{ (omitted factor)}}}$$

$$\times \underbrace{\frac{1}{\sigma^2_{\text{per}}\sqrt{2\pi}} \exp\left(-\frac{\left(\ln \sigma^2_{\text{per}}+1\right)^2}{2}\right)}_{\substack{\text{LogNormal prior for } \sigma^2_{\text{per}}>0 \\ \text{with } \mu=-1,\ \sigma^2=0.7}} \times \underbrace{\mathbf{1}_{\{0.3<p<1.0\}}}_{\substack{\text{Uniform prior on } p\in(0.3,1.0)}}$$

$$\times \underbrace{\frac{1}{l_{\text{rbf}}\sqrt{2\pi}} \exp\left(-\frac{\left(\ln l_{\text{rbf}}+1\right)^2}{2}\right)}_{\substack{\text{LogNormal prior for } l_{\text{rbf}}>0 \\ \text{with } \mu=-1,\ \sigma^2=0.7}} \times \underbrace{\frac{1}{\sigma^2_{\text{rbf}}\sqrt{2\pi}} \exp\left(-\frac{\left(\ln \sigma^2_{\text{rbf}}+1\right)^2}{2}\right)}_{\substack{\text{LogNormal prior for } \sigma^2_{\text{rbf}}>0 \\ \text{with } \mu=-1,\ \sigma^2=0.7}}.$$

# D Contributions

All authors contributed and weighted in on most parts of the project but focused on their respective strengths. Discussing ideas and the problems we encountered as a team was essential for solving them and progressing.

**Björn Thor Stefnánsson** Björn Thor Stefnánsson contributed primarily to Part B of the project by overseeing the modeling, optimization, and sampling of the Gaussian Processes models. However, he also had a large part in the results and discussions of part A 2.3. He was responsible for Section 3.1 (Kernel selection), Section 3.2, and Section 3.3. In addition to his work on the main sections, Björn authored Appendix B.4, which details the NUTS diagnostics, and Appendix C.2, which outlines the definitions and formulas used throughout the project. Furthermore, Björn assisted in writing the code used for the Gaussian Processes modelling with respect to certain constraints (Section 3.7).

**Oliver Sanchez** focused on improving the diffusion models and their evaluation. He implemented Variant 2 (Low-Discrepancy Sampling, Section 2.1.3) and Variant 3 (Importance Sampling, Section 2.1.4). He was also responsible for the quantitative evaluation of the diffusion models (Section 2.2) and the formulation and implementation of the integral constraint for Gaussian Processes (Section 3.6). Additionally, Oliver authored the corresponding appendices: Appendix A.3, Appendix A.6 and Appendix A.4, detailing the code changes for Variants 2 and 3.

**Viktor Már Guðmundsson** implemented Variant 1 (original image prediction, Section 2.1.2) and Variant 4 (classifier-free guidance, Section 2.1.5) for diffusion modeling. He also performed the quantitative comparison of diffusion models (Section 2.2), diffusion results and discussion (Section 2.3), kernel selection (Section 3.1), and provided mathematical proofs for the probability distributions with the integral constraint (Sections 3.6.1 and 3.6.2). Lastly, he created the plots for the constrained GP results (Section 3.7).

In the appendices, Viktor authored several sections: Appendix A.1, detailing adjustments to the reverse diffusion step; Appendix A.2 and Appendix A.5 about code changes for Variant 1 and 4; Appendix B.1, presenting diffusion generated results; Appendix B.2, presenting kernel selection results; Appendix B.3, covering cross-validation results; and Appendix C.1, providing the proof that the rank is $\ell - 1$;