Machine Learning for Multiple Yield Curve Markets: Fast Calibration in the Gaussian Affine Framework Sandrine Gumbel and Thorsten Schmidt

Dylan Riboulet

May 3, 2024

Introduction

Calibration is a highly complex task, particularly in the context of multiple yield curves. This paper presents an initial approach using machine learning methods to address this challenge. The coexistence of multiple yield curves associated with different tenor structures was a contributing factor to the 2007-2009 financial crisis. At that time, spreads between different yield curves peaked beyond 200 basis points. Since then, these spreads have remained at a significant level. The most important curves to consider in the current economic environment are overnight-indexed swap (OIS) rates and interbank offered rates (Ibor), such as the London Interbank Offered Rate (Libor), across various tenors. In the European market, these correspond to OIS rates based on Eonia and Euribor rates, respectively.

First Problem

Here, we are faced with limited data and a high-dimensional prediction problem, such as a complete curve (the term structure). In the multi-curve market, multiple curves must be calibrated and predicted simultaneously.

Proposed Approach

The idea of the paper to efficiently solve this problem is to incorporate historical information and a Bayesian approach. Therefore, Gaussian Process Regression (GPR) is chosen as the machine learning method to ensure rapid calibration (according to De Spegeleer et al. (2018)). It is a non-parametric Bayesian method capable of capturing non-linear relationships between variables. We operate within the Vasicek model, which ensures a normal distribution for r_t and consequently a normal distribution for log-bonds according to theory. By calculating the expectation and covariance of log-bond prices, we obtain the mean function and kernel of our Gaussian process for performing Gaussian Process Regression. Initially, we implemented the hyperparameter values, simulated 1000 samples of log-bond prices over 252 days (equivalent to 1 year of history) see Appendix 1.

Regression and Calibration

Consider a regression problem with additive Gaussian noise: we observe $y_1, \dots, y_n$ with variables $x_1, \dots, x_n$ and assume that $y_i = f(x_i, \theta) + \epsilon_i$, $i = 1, \dots, n$,

where $f = \log$ and $x_i = \log B(t_i, T, \delta)$. We obtain $n = 252$ log-bond price values for each sample.

Part One: Single Yield Curve Modeling

Initially, we define the parameters of the Vasicek model, the time step (1/252), and the Riccati equations for the deterministic functions $A(t,T)$ and $C(t,T)$ in vector form. We then implement the mean and covariance functions, taking into account Gaussian white noise with variance $\hat{\sigma}^2$ in the covariance matrix. We use the `numpy` module and avoid loops by using predefined functions to improve computation time during simulation. Next, we implement the 95% confidence interval and plot the Prior Mean, the 95% confidence interval for an initial illustration of the paper's results.

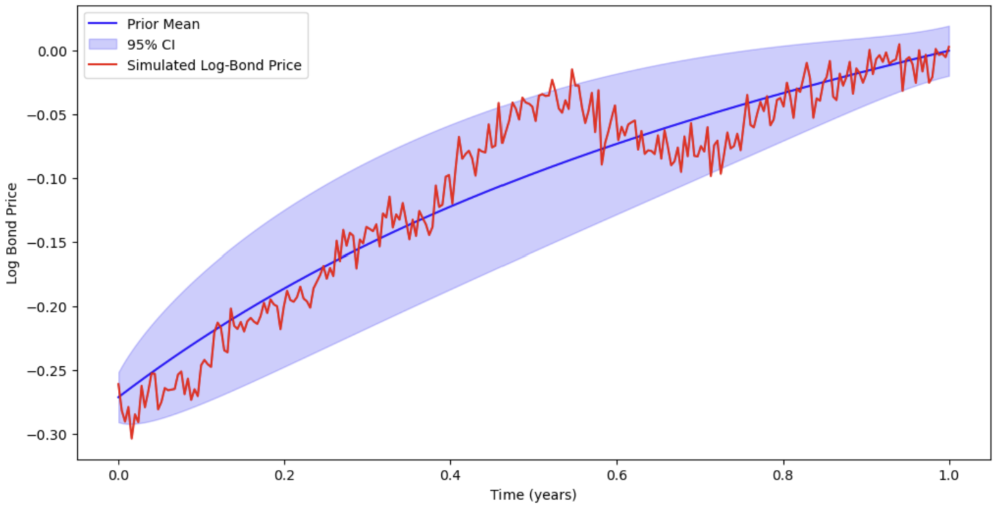


Figure 1: Illustration of a GP prior. Our objective is to predict the unobserved log-bond prices (red dashed line) with a maturity of $T = 1$ year. The blue line represents the prior mean, and the shaded area represents the 95% confidence interval. Note that the final value $B(T,T) = 1$ implies $\log B(T,T) = 0$.

The shape of the log bond prices, predictions, and confidence intervals is similar to what is found in the paper, confirming the implementation of the previous functions.

We ensure that the covariance matrix is symmetric and positive definite using an implemented function to test it. To apply our machine learning model, we split our dataset of simulated zero-coupon bond prices over 252 days into 70% training data (176 data points) and 30% test data (76 data points) as suggested in the paper. Then, we apply the theory of the conditional Gaussian law by formally posing the question: assuming we have training data $y = (y(t_1,T),\ldots,y(t_n,T))$, we aim to predict the vector $\tilde{y} = (y(s_1,T),\ldots,y(s_m,T))$.

$$\begin{pmatrix} y \\ \tilde{y} \end{pmatrix} \sim \mathcal{N}\left(\begin{pmatrix} \mu_y \\ \mu_{\tilde{y}} \end{pmatrix}, \begin{pmatrix} \Sigma_{yy} & \Sigma_{y\tilde{y}} \\ \Sigma_{\tilde{y}y} & \Sigma_{\tilde{y}\tilde{y}} \end{pmatrix}\right)$$

$\tilde{y}|y \sim \mathcal{N}\left(\mu_{\tilde{y}} + \Sigma_{\tilde{y}y}\Sigma_{yy}^{-1}(y-\mu_y), \Sigma_{\tilde{y}\tilde{y}} - \Sigma_{\tilde{y}y}\Sigma_{yy}^{-1}\Sigma_{y\tilde{y}}\right)$

We implement the matrices $\Sigma_{\tilde{y}y}$ and $\Sigma_{y\tilde{y}}$, invert the matrix Σ_{yy} which has a cost of $O(n^3)$ via Cholesky decomposition, and directly apply the `inv` function from the numpy module for this. We verify the size of each matrix and ensure they are symmetric and positive definite, then perform the matrix multiplication in Python to obtain the estimators. Thus, we have our Bayesian method estimator for the mean, which allows us to predict new data, and the conditional variance helps us construct the 95% confidence interval.

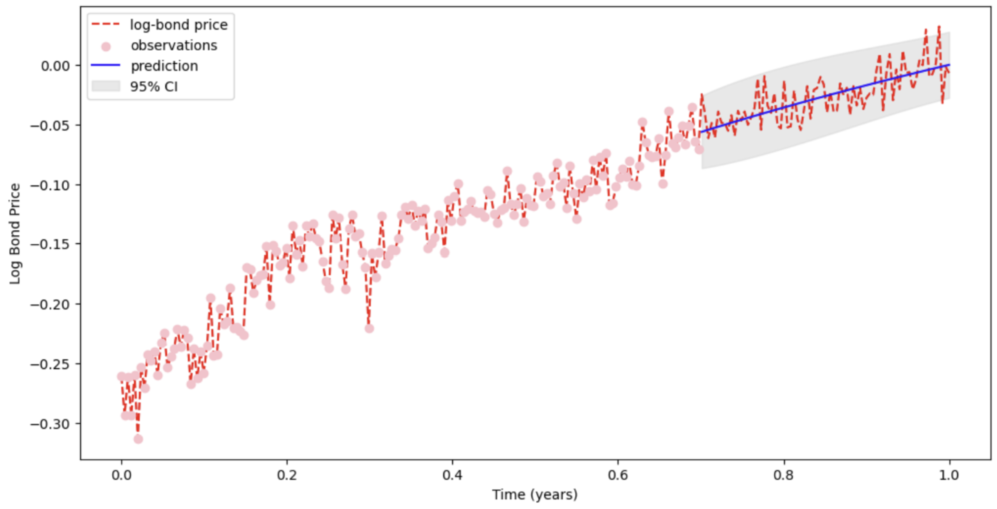


Figure 2: Illustration of a posterior distribution, i.e., the prior conditioned on 70% training data. We aim to predict the log-bond prices (red dashed line). The red points are observations of log-bond prices. The blue line is the posterior mean, which we take as the prediction. The shaded area represents the 95% confidence interval.

Observations

The posterior distributions, that is, the prior conditioned on training data with 10% observations respectively, are illustrated in Figure 2. The red dashed line represents a simulated trajectory of log-bond prices, while the red points constitute the observed training data. The blue line is the posterior mean, which we take as our prediction, and the shaded areas indicate the 95% confidence interval. We observe that the more training data we have, the more precise our prediction becomes. While in Figure 4 the confidence interval is quite wide, a few additional training data points significantly reduce the confidence interval.

Importance of Observation Times (t_i)

Significantly, the choice of observation times (t_i) plays a crucial role. In most practical cases, the (t_i) cannot be freely chosen and arrive sequentially, as illustrated in Figures 1-3: here, we aim to predict the log-bond prices of a bond with a maturity of one year that we have observed daily up to day $t_n = 176$, then $t_n = 227$, and $t_n = 25$. Figures 1-3 describe the situation in which we have no information about future prices of the one-year maturity bond, over one to several months.

Performance Metrics

To evaluate the model's performance, we use metrics commonly employed in time series analysis such as Mean Squared Error (MSE) and Standardized Mean Squared Error (SMSE). The SMSE considers the mean of the squared residuals between the predicted mean and the test set target, then standardizes it by the variance of the test cases' targets.

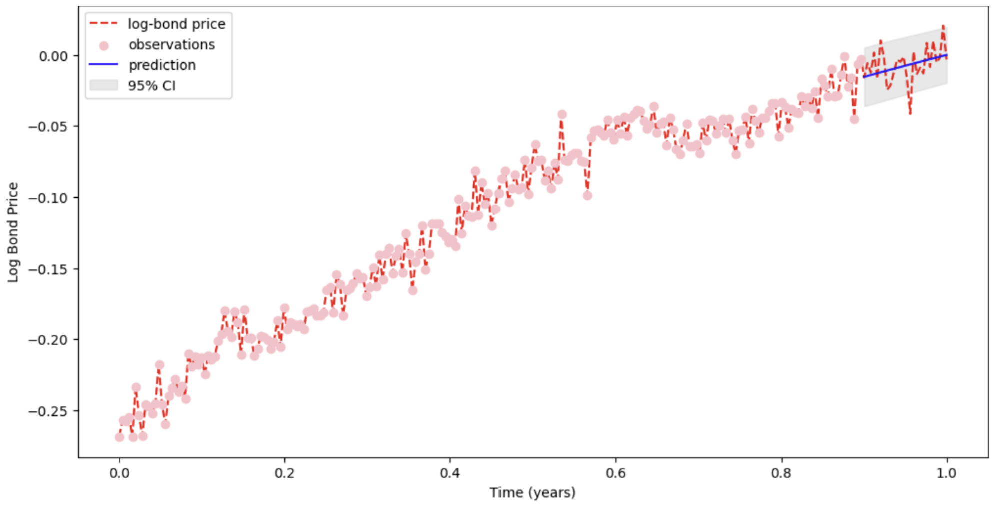


Figure 3: Posterior distribution with 90% training data. We want to predict log-bond prices (red dotted line). The red points constitute observations of log-bond prices. The blue line is the posterior mean, which we take as the prediction. The shaded area represents the 95% confidence interval.

Calibration

The 1000 simulations of log-bond prices will be used for calibration (see Appendix 1). For each of the 1000 samples, we seek the optimal choice of hyperparameters $\Theta = \{r_0, \kappa, \theta, \sigma\}$ based on the simulated trajectory of 252 log-bond prices. To this end, we minimize the negative marginal log-likelihood using the mean and covariance functions from the paper, respectively, through two optimization methods: the non-linear conjugate gradient (CG) algorithm and the adaptive moment estimation (Adam) optimization algorithm. We use the `minimize` function from `scipy` for the CG optimization algorithm and the `TensorFlow` module for the Adam optimization algorithm. After random initialization of the parameters, we use the algorithms already implemented in the `minimize` functions and retrieve the optimal hyperparameter vectors in a list, then calculate the mean and variance for our 1000 results.

Comparison of CG and Adam Algorithms for Calibration

Advantages and Disadvantages

We observe that:

1. The calibration results using the Conjugate Gradient (CG) optimization algorithm are very satisfactory. In the figure, the learned parameters r_0, κ, θ, and σ from 1000 simulated log-bond prices are represented in histograms. The orange in each subplot indicates the true model parameters. The mean and standard deviation of the learned parameters are summarized in the two graphs for the CG and Adam optimization algorithms. We observe that the mean of the learned parameters closely approaches the true parameters while the standard deviation is reasonable.

2. The calibration results using the Adam algorithm are satisfactory, although we note a shift in the mean reversion parameter κ and the volatility parameter σ. The learned parameters r_0, κ, θ, and σ from 1000 simulated log-bond prices are summarized with their mean and standard deviation in Figure 6.

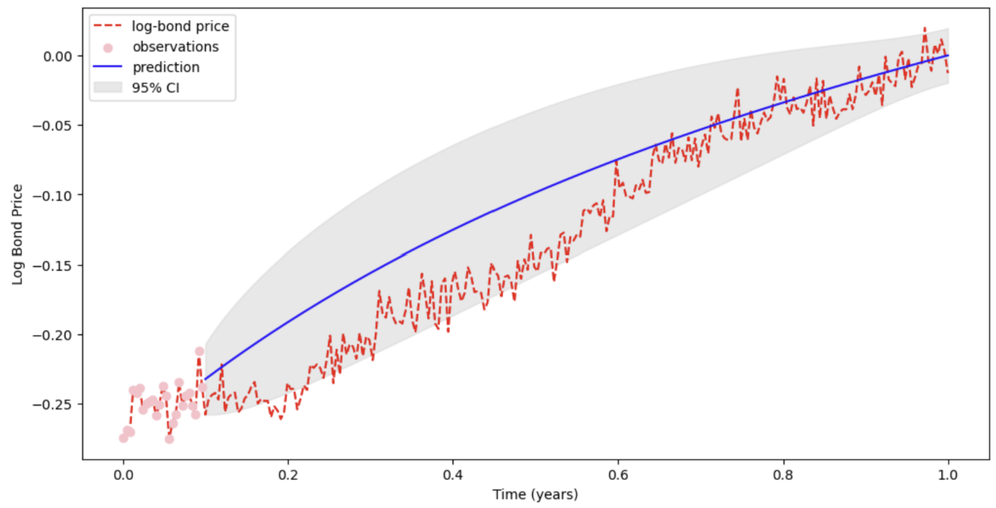
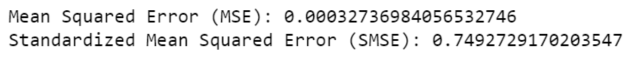


Figure 4: Posterior distribution with 10% training data. We want to predict log-bond prices (red dotted line). The red points constitute observations of log-bond prices. The blue line is the posterior mean, which we take as the prediction. The shaded area represents the 95% confidence interval.



```
Mean Squared Error (MSE): 0.00032736984056532746
Standardized Mean Squared Error (SMSE): 0.7492729170203547
```

Figure 5: Performance evaluation metrics of GPR

> The Adam algorithm gradually reduces the learning rate over time to accelerate the learning algorithm. However, it is necessary to specify an appropriate learning rate. If the learning rate is small, training is more reliable, but the optimization time to find a minimum can increase rapidly. If the learning rate is too large, the optimizer may overshoot a minimum. We tried different learning rates of 0.0001, 0.001, 0.01, 0.05, and 0.1. Finally, we opted for a learning rate of 0.05.

We conclude that in the specification of the single Vasicek curve, both optimization algorithms provide reliable results. Optimization via the CG algorithm outperforms the Adam algorithm, see Figure 6. However, there is hope that the results obtained by the Adam algorithm can be improved by choosing a smaller learning rate.

Part Two - Multi-Curve Modeling

Theory

We generalize the Vasicek model to two processes, r_1 and r_2. We consider log delta bonds where, in the case $\delta = 0$, we recover the theory from the first part. For implementation, we now have 8 hyperparameters instead of 4, where r_1 plays the role of r in the previous section.

The value $B(t, T, \delta)$ is given by:

$$B(t, T, \delta) = E_Q \left[e^{-\int_t^T (r_{1,s} - r_{2,s})\, ds} \mid \mathcal{F}_t \right]$$

Since r is an affine process, this expectation can be calculated explicitly. Using the theory for affine rate models from the paper, we find that log delta bonds follow a Gaussian distribution, allowing us to use GPR.

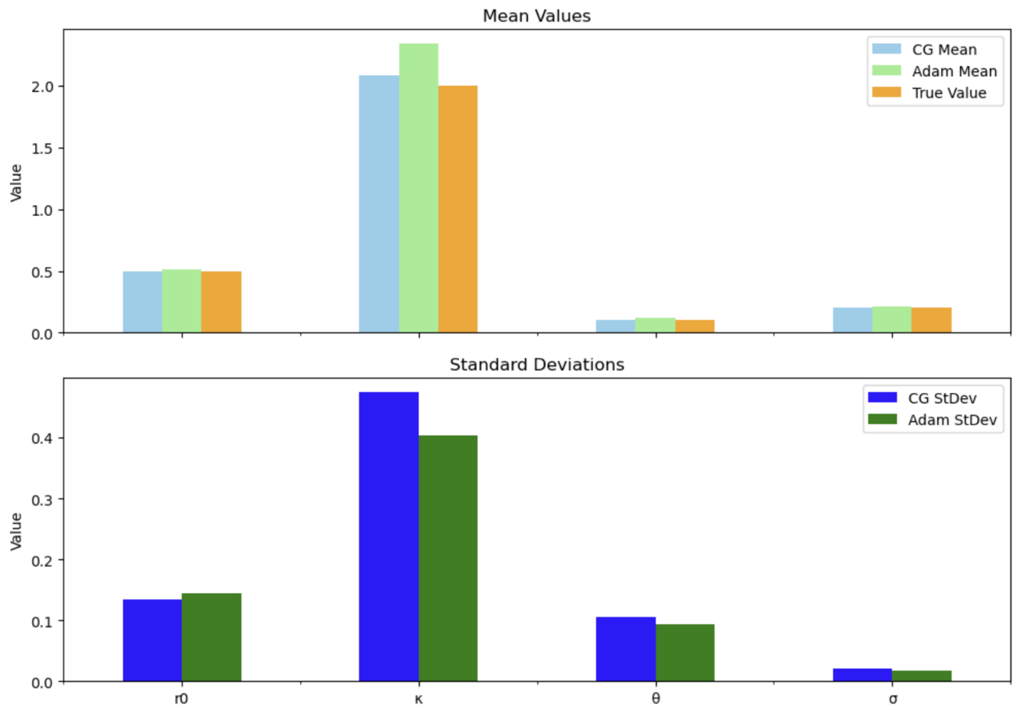


Figure 6: Calibration results for the single curve with means and standard deviations (StDev) of the learned parameters from 1000 simulated log-bond prices as well as the true Vasicek parameters.

Model Implementation for Prediction

First, we implement the functions ϕ, ψ_1, and ψ_2 from the Riccati equations obtained in the paper. Thanks to Proposition 2 of the paper, we implement the mean and kernel functions to apply GPR. According to the theory, log delta bond prices also follow Gaussian laws in the case where W_1 and W_2 are Brownian motions with correlation ρ. Thus, by considering the function $f = \log$, $x_i = \log B(t_i, T, \delta)$, and $\Theta = \{r_{01}, r_{02}, \dots\}$, we use the Gaussian Process Regression theory previously used in the pre-crisis framework (single yield curve modeling) and simulate 1000 log delta bond prices (see Appendix 2). The predictive mean and conditional variance (for confidence intervals) are implemented in the same manner as in Part I.

We denote $y = (y_0, y_\delta)^\top$ where $y_0 = (y(t_1, T, 0), \dots, y(t_n, T, 0))$ and $y_\delta = (y(t_1, T, \delta), \dots, y(t_n, T, \delta))$. At each time t_i, we now observe two bond prices: $P(t_i, T, 0)$ and $P(t_i, T, \delta)$. The vector y is normally distributed, $y \sim \mathcal{N}(\mu_y, \Sigma_y)$ with $\mu_y := \begin{pmatrix} \mu(\cdot, T, 0) \\ \mu(\cdot, T, \delta) \end{pmatrix}$, $\Sigma_y := \begin{pmatrix} \Sigma_{00} & \Sigma_{0\delta} \\ \Sigma_{\delta 0} & \Sigma_{\delta\delta} \end{pmatrix}$. We calculate these parameters explicitly and, analogous to the single-curve setup, the calibration methodology with Gaussian Process Regression follows. We split the dataset into 70% training data and 30% test data as in the first part.

Performance Metrics

We use the same metrics as in the single curve case and obtain the following results:

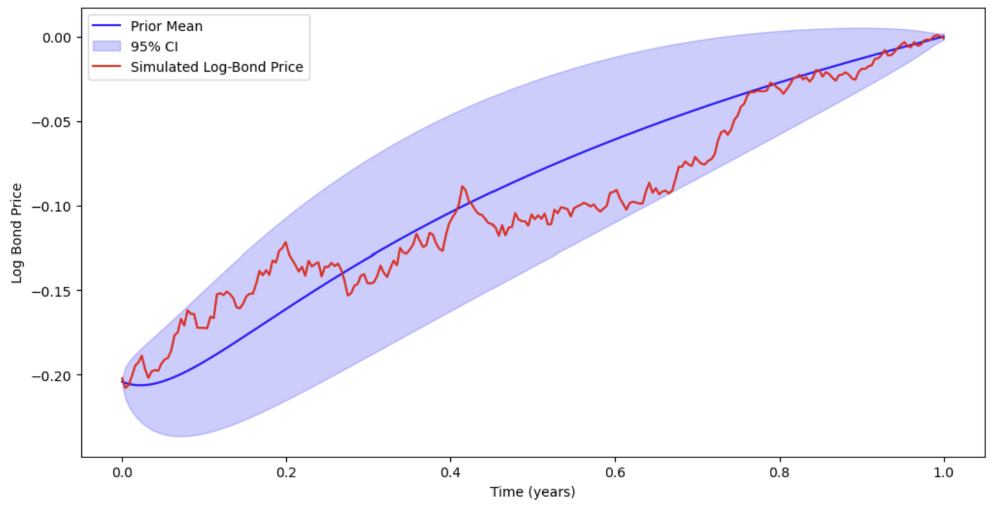


Figure 7: Illustration of a Gaussian process prior. Our task is to predict the unobserved log-delta-bond prices (red dotted line) with a maturity of $T = 252$. The blue line represents the prior mean, and the shaded area represents the 95% confidence interval. Note that the initial value at time 0 is known, as well as the final value $P(T, T) = 1$ implying $\log P(T, T) = 0$.

```
Mean Squared Error (MSE): 3.984432558808832e-05
Standardized Mean Squared Error (SMSE): 0.2091143291090524
```

Calibration

We consider 126 values to maintain similar computational complexity in both pre- and post-crisis approaches for calibration and parallelization. As in the single curve case, we seek to minimize the negative log-likelihood. We implement it without the constant to simplify the computational cost in the optimization iterations using the algorithms.

Observations

1. Considering the calibration results using the CG optimization algorithm, several facts are noteworthy. The mean and standard deviation of the calibrated parameters are found in Table 2. Figure 5 shows the learned parameters of the processes r_1 and r_2 in histograms with 50 bins for 1000 simulations. The red dashed line in each subplot indicates the true model parameter value. Firstly, except for the long-term mean, the volatility for r_2 is higher than that for r_1, implying more difficulty in estimating parameters for r_2, which is clearly visible in the results. For r_1, we are able to estimate the parameters well (on average), with the greatest difficulty in estimating κ_1 which shows a high standard deviation. For the estimation of r_2 parameters, we face more challenges as expected. The standard deviation of θ_2 is very high – it is known from filtering and statistical theory that the mean is difficult to estimate, which is reflected here. Similarly, it seems difficult to estimate the speed of mean reversion parameter κ_2, and we observe a peak around 0.02 in κ_2. This could be due to a local minimum where the optimizer gets stuck.

2. For the calibration results using the Adam algorithm, we note the following. The learned parameters are illustrated in a histogram with 50 bins, see Figure 6, and the mean and standard deviation of each parameter are indicated in Table 2. After trying several learning rates of 0.0001, 0.001, 0.01, 0.05, and 0.1, we opted for a learning rate of 0.05. Although most of the learned parameters' mean values are not as close

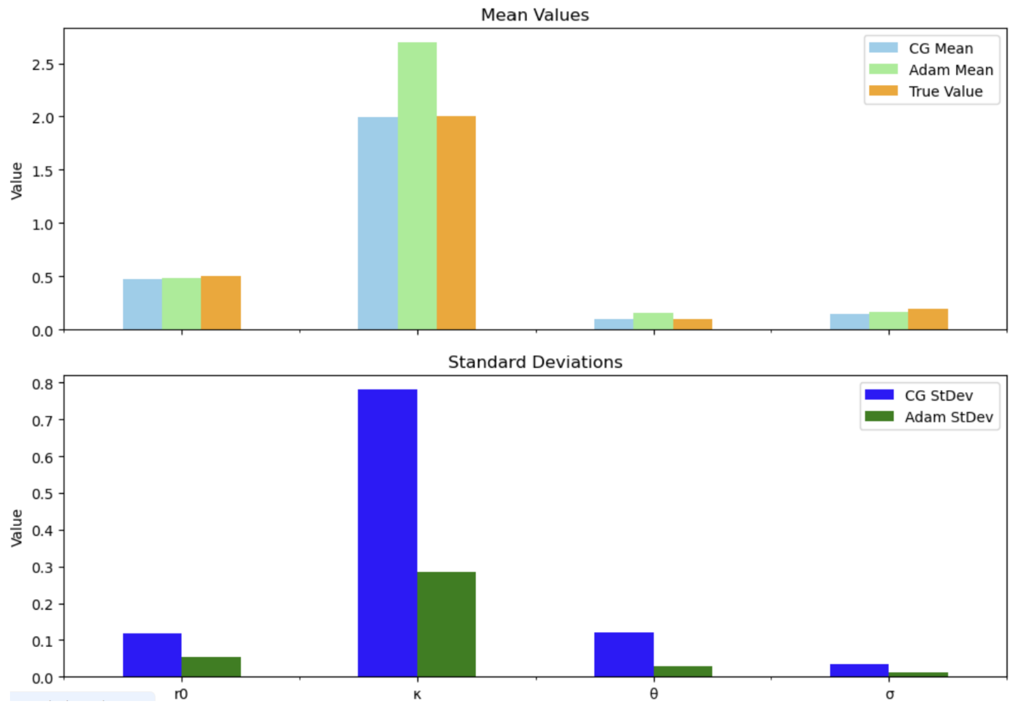


Figure 8: Calibration results with means and standard deviations (StDev) of the learned parameters from 1000 simulated log-bond and log tenor-δ bond prices, as well as the true Vasicek parameters.

to the true values as those learned by the CG algorithm, we notice that the standard deviation of the learned parameters is smaller compared to the standard deviation of the CG algorithm's learned parameters. In particular, comparing the values of θ_2 in Figure 9, we observe a very significant difference.

Conclusion

In concluding the simulation results, we can affirm that calibration in the multi-curve framework is a more complex task compared to calibration in the single-curve framework, as we need to find 8 parameters instead of 4. In particular, to maintain computational complexity similar to the single-curve framework, we chose 126 training data points, resulting in a covariance matrix of the same dimension as in the single-curve framework. We are confident that doubling the training data points would improve the results at the cost of increased computation time based on the obtained prediction results. Moreover, the GPR approach naturally comes with the posterior distribution, which contains much more information compared to simple prediction (which contains only the mean). In summary, calibrating multiple yield curves is a challenging task, and we hope to stimulate future research with this initial study showing promising results on one hand and numerous future challenges on the other.

Future Work (Other Possible Approaches)

A first step to extend the presented approach could involve studying other optimization techniques such as the adaptive gradient algorithm (AdaGrad) or its extension Adadelta, Root Mean Square Propagation

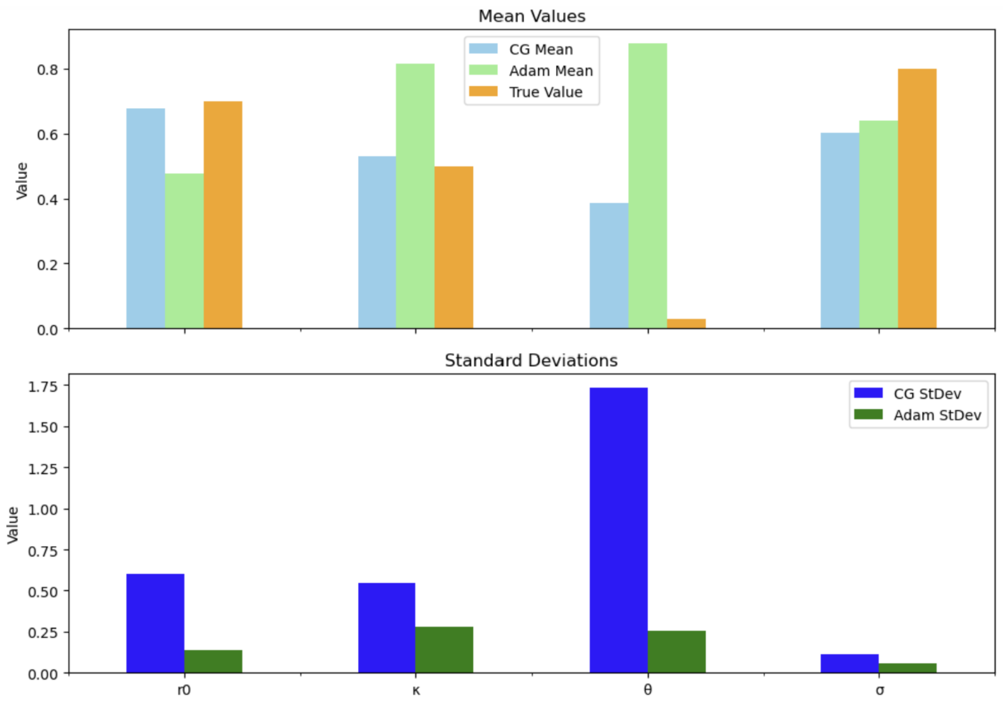


Figure 9: Calibration results with means and standard deviations (StDev) of the learned parameters from 1000 simulated log-bond and log tenor-δ bond prices, as well as the true Vasicek parameters.

(RMSProp), and Nesterov's Accelerated Gradient (NAG). One could also further investigate the prediction of log-bond prices with the learned parameters to provide support for decision-making in purchasing zero-coupon bond options and use the underlying exercise prices as worst-case scenarios. A next step could be the development of other short-rate models such as Cox–Ingersoll–Ross, the extended Vasicek model by Hull–White, or the Cox–Ingersoll–Ross framework. An interesting application is the calibration of interest rate models including jumps. Beyond that, studying the calibration of interest rate markets using Bayesian neural networks seems very promising. It would also be interesting to analyze how other estimators perform compared to the results presented, such as classical maximum likelihood estimators. Since it is already known that estimating variance is difficult with ML techniques, it could also be very interesting to combine classical approaches with ML approaches. Furthermore, in the case of data scarcity, we can use time series-specific resampling methods such as the `TimeSeriesSplit` function in Python to improve prediction performance before training the machine learning model on the data.

Appendix

Appendix 1

Appendix 2

Python Code

```
from numpy.linalg import inv
```

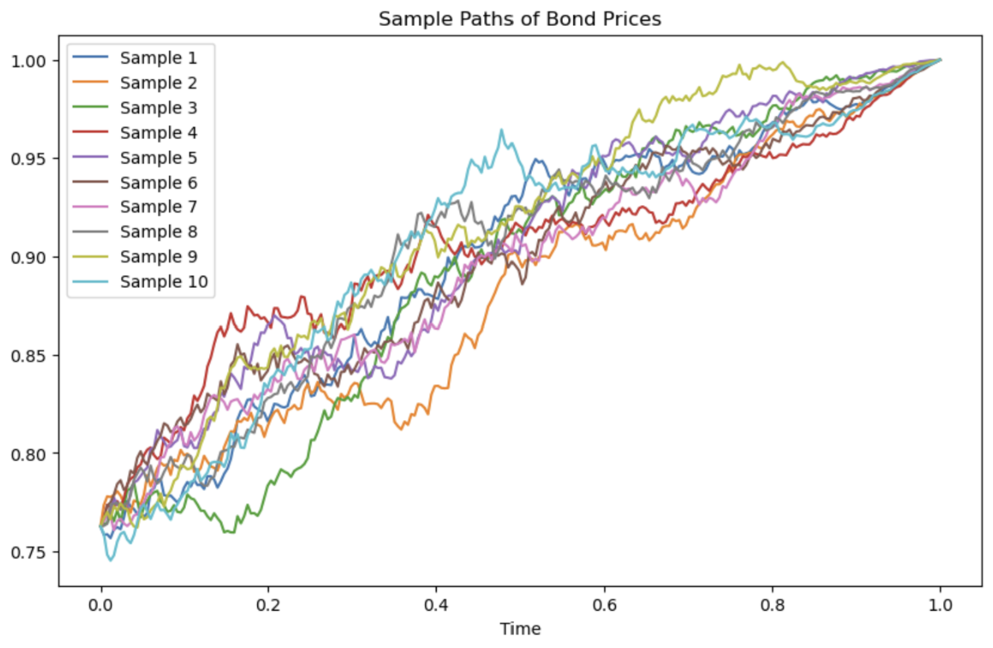


Figure 10: Simulation of 10 log bond prices.

```python
from scipy.spatial.distance import cdist
import numpy as np
from scipy.spatial.distance import pdist, squareform
from scipy.stats import multivariate_normal
import matplotlib.pyplot as plt
from scipy.optimize import minimize
```

Listing 1: Modules Used

```python
# Definition of parameters
kappa =  2
theta = 0.1
sigma = 0.2
T = 1
r0 = 0.5
sigma_noise = 0 # noise

# Definition of A and B functions
def compute_A_B(kappa, theta, sigma, T, t):
    A = (theta / kappa) * (np.exp(-kappa * (T-t)) + kappa * (T-t) - 1) + (sigma**2 / (4 *
    kappa**3)) * (np.exp(-2 * kappa * (T-t)) - 4 * np.exp(-kappa * (T-t)) - 2 * kappa * (T-t
    ) + 3)
    B = 1 / kappa * (1 - np.exp(-kappa * (T-t)))
    return A, B

# Time points for evaluation
times = np.linspace(0, T, 252)

# Calculate the mean vector
mean_vector = np.array([-compute_A_B(kappa, theta, sigma, T, t)[0] - compute_A_B(kappa,
    theta, sigma, T, t)[1] * (r0 * np.exp(-kappa * t) + theta * (1 - np.exp(-kappa * t)))
```

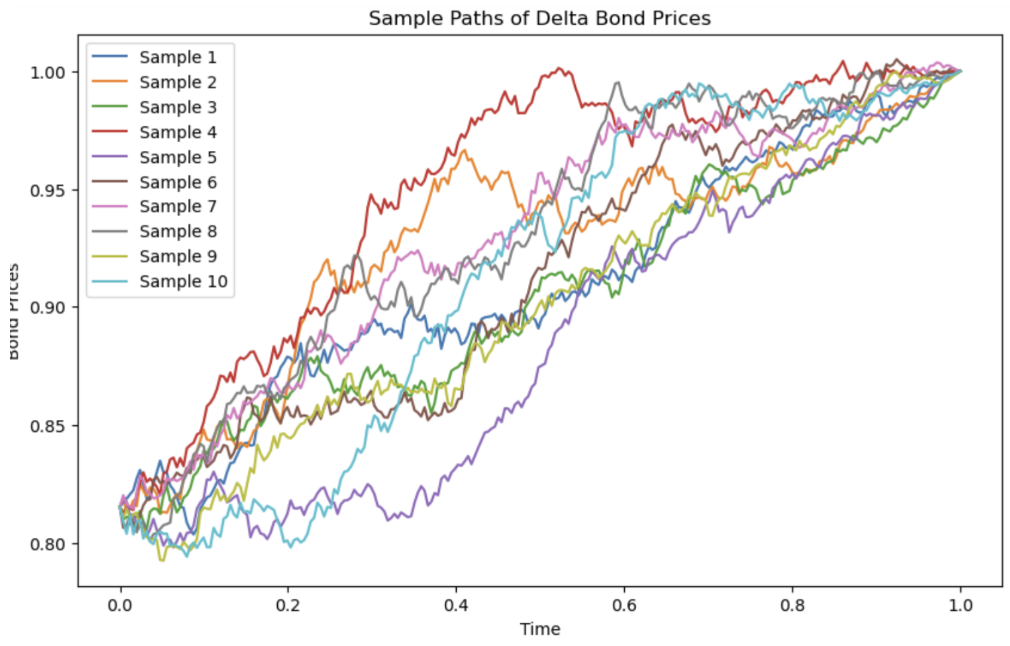


Figure 11: Simulation of 10 log delta bond prices.

```
    for t in times])

def mean_function(times, r0, kappa, sigma, theta):
    return np.array([-compute_A_B(kappa, theta, sigma, T, t)[0] - compute_A_B(kappa, theta,
    sigma, T, t)[1] * (r0 * np.exp(-kappa * t) + theta * (1 - np.exp(-kappa * t))) for t in
    times])

# Calculate the covariance matrix
def covariance_matrix(times, kappa, sigma, T):
    B_values = np.array([compute_A_B(kappa, theta, sigma, T, t)[1] for t in times])
    exp_term = np.exp(-kappa * (times[:, None] + times[None, :]))  # e^{-kappa(s+t)}
    min_matrix = np.minimum.outer(times, times)  # min(s, t) for each pair (s, t)
    exp_min_term = np.exp(2 * kappa * min_matrix) - 1  # e^{2*kappa*min(s,t)} - 1

    cov_matrix = (B_values[:, None] * B_values[None, :]) * (sigma**2 / (2 * kappa)) *
    exp_term * exp_min_term
    return cov_matrix + np.eye(len(times)) * sigma_noise**2

def covariance_matrix_cross(t_train, t_predict, kappa, sigma, T):
    # Calculate B for both time sets
    B_train = np.array([compute_A_B(kappa, theta, sigma, T, t)[1] for t in t_train])
    B_predict = np.array([compute_A_B(kappa, theta, sigma, T, t)[1] for t in t_predict])

    # Calculate e^{-kappa(s+t)} for each pair (s,t)
    st_sum = np.add.outer(t_train, t_predict)
    exp_term = np.exp(-kappa * st_sum)

    # Calculate e^{2*kappa*min(s,t)} - 1 for each pair (s,t)
    min_matrix = np.minimum.outer(t_train, t_predict)
    exp_min_term = np.exp(2 * kappa * min_matrix) - 1

```

```python
    # Calculate the cross covariance matrix
    cov_matrix = np.outer(B_train, B_predict) * (sigma**2 / (2 * kappa)) * exp_term *
    exp_min_term

    return cov_matrix

# Construction of the covariance matrix
cov_matrix = covariance_matrix(times, kappa, sigma, T)

# Sampling the Gaussian process
samples = multivariate_normal.rvs(mean=mean_vector, cov=cov_matrix, size=1000)

# Plot
plt.figure(figsize=(10, 6))
for i in range(10):  # Plot 10 sample paths
    plt.plot(times, np.exp(samples[i]), label=f'Sample {i+1}')
plt.xlabel('Time')
plt.ylabel('Log Bond Prices')
plt.title('Sample Paths of Bond Prices')
plt.legend()
plt.show()
```

Listing 2: Python Code for Prediction

```python
# Calculate mean and covariance
mean_vec = mean_function(times)
cov_mat = covariance_matrix(times, kappa, sigma, T) + np.eye(len(times)) * sigma_noise**2

# Sample from the Gaussian process
sample_path = np.random.multivariate_normal(mean_vec, cov_mat)

# Calculate the 95% confidence interval
std_dev = np.sqrt(np.diag(cov_mat))
ci_upper = mean_vec + 1.96 * std_dev
ci_lower = mean_vec - 1.96 * std_dev

# Plot
plt.figure(figsize=(12, 6))
plt.plot(times, mean_vec, 'b', label='Prior Mean')
plt.fill_between(times, ci_lower, ci_upper, color='blue', alpha=0.2, label='95% CI')
plt.plot(times, sample_path, 'r', label='Simulated Log-Bond Price')
plt.title('Prior Distribution and Simulated Log-Bond Price for a 1-Year Maturity Bond')
plt.xlabel('Time (years)')
plt.ylabel('Log Bond Price')
plt.legend()
plt.show()
```

Listing 3: Prior Distribution and Result Observation

```python
t_log_bond_prices = np.linspace(0, 1, 252)

# Calculate the mean vector
mean_train = mean_function(t_log_bond_prices, r0, kappa, sigma, theta)

# Calculate the covariance matrix
Sigma_yy = covariance_matrix(t_log_bond_prices, kappa, sigma, T) + np.eye(len(
    t_log_bond_prices)) * sigma_noise**2
```

```python

# Simulate log bond prices
log_bond_prices = np.random.multivariate_normal(mean_train, Sigma_yy) # simulated

data_indices = np.arange(0, 252)  # 252 days of data

# Split into training and validation data (70% training, 30% validation)
train_size = int(0.7 * len(data_indices))
train_indices = data_indices[:train_size]
validation_indices = data_indices[train_size:]

# Extract training and test data
t_train = times[train_indices]
y_train = log_bond_prices[train_indices]
t_validation = times[validation_indices]
y_validation = log_bond_prices[validation_indices]

# Calculate mean and covariance functions for training data
mean_train = mean_function(t_train, r0, kappa, sigma, theta)
cov_train = covariance_matrix(t_train, kappa, sigma, T) + np.eye(len(t_train)) * sigma_noise
    **2

# Calculate mean and covariance for test data
mean_validation = mean_function(t_validation, r0, kappa, sigma, theta)
cov_validation_cross = covariance_matrix_cross(t_train, t_validation, kappa, sigma, T)
cov_validation = covariance_matrix(t_validation, kappa, sigma, T) + np.eye(len(t_validation)
    ) * sigma_noise**2

# Prediction using GP
mu_validation_given_train = mean_validation + cov_validation_cross.T @ inv(cov_train) @ (
    y_train - mean_train)
Sigma_validation_given_train = cov_validation - cov_validation_cross.T @ inv(cov_train) @
    cov_validation_cross

# Predicted values
y_validation_predicted = np.random.multivariate_normal(mu_validation_given_train,
    Sigma_validation_given_train)
```

Listing 4: Simulation and Validation of Log Bond Prices

```python
plt.figure(figsize=(12, 6))

plt.plot(t_log_bond_prices, log_bond_prices, 'r--', label='log-bond price')

plt.scatter(t_train, y_train, color='pink', label='observations', zorder=3) # Training data

plt.plot(t_validation, mu_validation_given_train, 'b', label='prediction')  # Posterior mean

ci_upper = mu_validation_given_train + 1.96 * np.sqrt(np.diag(Sigma_validation_given_train))
ci_lower = mu_validation_given_train - 1.96 * np.sqrt(np.diag(Sigma_validation_given_train))
plt.fill_between(t_validation, ci_lower, ci_upper, color='lightgrey', alpha=0.5, label='95%
    CI')  # 'lightgrey' for CI

plt.title('Posterior Given 176 Training Examples')
plt.xlabel('Time (years)')
plt.ylabel('Log Bond Prices')

plt.legend()

plt.show()
```

Listing 5: Visualization of Prediction and Validation Results

```python
# Performance metrics
sigma_y_tilde_squared = np.var(y_validation)

mse = np.mean((y_validation - y_validation_predicted) ** 2)

smse = mse / sigma_y_tilde_squared

print(f"Mean Squared Error (MSE): {mse}")
print(f"Standardized Mean Squared Error (SMSE): {smse}")
```

Listing 6: Calculation of Performance Metrics for Validation

```python
def neg_log_likelihood(params, prices, times):
    r0, kappa, theta, sigma = params
    # Calculate the mean vector and covariance matrix using the defined functions
    mean_vector = mean_function(times, r0, kappa, theta, sigma)
    cov_matrix = covariance_matrix(times, kappa, sigma, T)
    # Calculate the conditional log-likelihood
    residual = prices - mean_vector
    try:
        log_likelihood = -0.5 * residual.T @ inv(cov_matrix) @ residual
        #log_likelihood -= 0.5 * np.linalg.slogdet(cov_matrix)[1]  # Log determinant of
    covariance matrix
        #log_likelihood -= 0.5 * len(prices) * np.log(2 * np.pi)
        return log_likelihood  # Return the negative log-likelihood
    except np.linalg.LinAlgError:
        return np.inf
```

Listing 7: Negative Log-Likelihood Function for Parameter Estimation

```python
from scipy.optimize import minimize

# List for optimization parameters
optimized_params = []

for sample in samples:
    initial_params = [np.random.rand(), 2 + np.random.rand(), 0.1 + 0.1 * np.random.rand(),
    0.2 + 0.1 * np.random.rand()]
    result = minimize(neg_log_likelihood, initial_params, args=(sample, times), method='CG')
    optimized_params.append(result.x)

optimized_params = np.array(optimized_params)
```

Listing 8: Optimization of Model Parameters via Negative Log-Likelihood Minimization

```python
params_mean = np.mean(optimized_params, axis=0)
params_std = np.std(optimized_params, axis=0)

print("Mean of optimal parameters: ", params_mean)
print("Standard deviation of optimal parameters: ", params_std)

# Define the number of bins
```

```python
bins = 50

# Create a figure and subplots
fig, axs = plt.subplots(2, 2, figsize=(10, 10))

# Histogram for r0
axs[0, 0].hist(optimized_params[:,0], bins=bins, color='blue', alpha=0.7)
axs[0, 0].set_title('Histogram of $r_0$')
axs[0, 0].set_xlabel('Values of $r_0$')
axs[0, 0].set_ylabel('Frequency')

# Histogram for kappa
axs[0, 1].hist(optimized_params[:,1], bins=bins, color='green', alpha=0.7)
axs[0, 1].set_title('Histogram of $\\kappa$')
axs[0, 1].set_xlabel('Values of $\\kappa$')

# Histogram for theta
axs[1, 0].hist(optimized_params[:,2], bins=bins, color='red', alpha=0.7)
axs[1, 0].set_title('Histogram of $\\theta$')
axs[1, 0].set_xlabel('Values of $\\theta$')
axs[1, 0].set_ylabel('Frequency')

# Histogram for sigma
axs[1, 1].hist(optimized_params[:,3], bins=bins, color='purple', alpha=0.7)
axs[1, 1].set_title('Histogram of $\\sigma$')
axs[1, 1].set_xlabel('Values of $\\sigma$')

# Display the results
plt.tight_layout()
plt.show()

# Calculate the mean and standard deviation for CG and Adam
stats_cg = {
    "Mean": np.mean(optimized_params_cg, axis=0),
    "StDev": np.std(optimized_params_cg, axis=0)
}
stats_adam = {
    "Mean": np.mean(optimized_params_adam, axis=0),
    "StDev": np.std(optimized_params_adam, axis=0)
}

# Define parameter names
params_names = ['r0', ' ', ' ', ' ']
true_values = [0.5, 2, 0.1, 0.2]

# Create dataframes for CG and Adam
df_cg = pd.DataFrame(stats_cg, index=params_names)
df_adam = pd.DataFrame(stats_adam, index=params_names)

# Combine the data into a single DataFrame
df_combined = pd.concat([df_cg, df_adam], axis=1, keys=['CG', 'Adam'])

# Add a row for the true values
df_combined.loc['True Value'] = pd.Series({('CG', 'Mean'): true_values, ('Adam', 'Mean'): true_values})

# Reorder to add the true values
```

```
df_combined = pd.concat([df_combined.loc[['True Value']], df_combined.drop('True Value')])

print(df_combined)
```

Listing 9: Statistical Analysis and Visualization of Optimal Parameter Distributions

```
# Define model parameters as TensorFlow variables
r0 = tf.Variable(0.5, dtype=tf.float32, name='r0')
kappa = tf.Variable(0.1, dtype=tf.float32, name='kappa')
theta = tf.Variable(0.01, dtype=tf.float32, name='theta')
sigma = tf.Variable(0.01, dtype=tf.float32, name='sigma')

def tf_neg_log_likelihood(prices, times, r0, kappa, theta, sigma):
    # Implement the mean function and covariance function with TensorFlow
    mean_vector = tf_mean_function(times, r0, kappa, theta, sigma)
    cov_matrix = tf_covariance_matrix(times, kappa, sigma, theta) + tf.eye(tf.size(times)) *
     sigma_noise

    # Calculate the conditional log-likelihood using TensorFlow
    residual = prices - mean_vector
    cov_matrix_inv = tf.linalg.inv(cov_matrix)
    log_likelihood = -0.5 * tf.matmul(tf.matmul(tf.transpose(residual), cov_matrix_inv),
    residual)
    log_likelihood -= 0.5 * tf.linalg.slogdet(cov_matrix)[1]
    log_likelihood -= 0.5 * tf.cast(tf.size(prices), tf.float32) * tf.math.log(2.0 * tf.
    constant(np.pi, dtype=tf.float32))
    return -log_likelihood

optimizer = tf.optimizers.Adam(learning_rate=0.005)

sample_prices = tf.constant(sample, dtype=tf.float32)
times_tensor = tf.constant(times, dtype=tf.float32)

# Define the optimization step as a TensorFlow function
@tf.function
def optimization_step():
    with tf.GradientTape() as tape:
        loss = tf_neg_log_likelihood(sample_prices, times_tensor, r0, kappa, theta, sigma)
    gradients = tape.gradient(loss, [r0, kappa, theta, sigma])
    optimizer.apply_gradients(zip(gradients, [r0, kappa, theta, sigma]))

# Perform optimization
for _ in range(1000):
    optimization_step()
```

Listing 10: Optimization of Model Parameters Using TensorFlow

```
# Parameters for modeling
kappa1 = 2
theta1 = 0.1
sigma1 = 0.2
T = 1
r01 = 0.5
kappa2 = 18
theta2 = 0.03
sigma2 = 0.8
r02 = 0.7
```

```
sigma_noise = 0
rho = 0.5

# Definitions of deterministic functions from the Riccati Equations
def Psi_1(T, t, kappa1):
    return -1 / kappa1 * (1 - np.exp(-kappa1 * (T - t)))

def Psi_2(T, t, kappa2):
    return 1 / kappa2 * (1 - np.exp(-kappa2 * (T - t)))

def Phi(T, t, theta1, theta2, kappa1, kappa2, sigma1, sigma2, rho):
    term1 = -(theta1 - theta2) * (T - t)
    term2 = - (theta1 / kappa1) * (np.exp(-kappa1 * (T - t)) - 1)
    term3 = (theta2 / kappa2) * (np.exp(-kappa2 * (T - t)) - 1)
    term4 = sigma1**2 / (2 * kappa1**2) * ((T - t) + (2/kappa1) * np.exp(-kappa1 * (T - t))
    - (1/(2 * kappa1)) * np.exp(-2 * kappa1 * (T - t)) - 3/(2 * kappa1))
    term5 = sigma2**2 / (2 * kappa2**2) * ((T - t) + (2/kappa2) * np.exp(-kappa2 * (T - t))
    - (1/(2 * kappa2)) * np.exp(-2 * kappa2 * (T - t)) - 3/(2 * kappa2))
    term6 = -rho * sigma1 * sigma2 / (kappa1 * kappa2) * ((T - t) + (1/kappa1) * (np.exp(-
    kappa1 * (T - t)) - 1) + (1/kappa2) * (np.exp(-kappa2 * (T - t)) - 1) - (1/(kappa1 +
    kappa2)) * (np.exp(-(kappa1 + kappa2) * (T - t)) - 1))

    return term1 + term2 + term3 + term4 + term5 + term6
```

Listing 11: Definition of Deterministic Functions for Multi-Curve Model

```
# Definition of mean and covariance functions
def mean_function_delta(times, r01, r02, kappa1, kappa2, sigma1, sigma2, theta1, theta2):
    return np.array([Phi(T, t, theta1, theta2, kappa1, kappa2, sigma1, sigma2, rho) + Psi_1(
    T, t, kappa1) * (r01 * np.exp(-kappa1 * t) + theta1 * (1 - np.exp(-kappa1 * t))) + Psi_2
    (T, t, kappa2) * (r02 * np.exp(-kappa2 * t) + theta2 * (1 - np.exp(-kappa2 * t))) for t
    in times])

def covariance_matrix_delta(times, kappa1, kappa2, sigma1, sigma2, rho, T):
    Psi1_values = np.array([Psi_1(T, t, kappa1) for t in times])
    Psi2_values = np.array([Psi_2(T, t, kappa2) for t in times])
    exp_term1 = np.exp(-kappa1 * (times[:, None] + times[None, :]))
    exp_term2 = np.exp(-kappa2 * (times[:, None] + times[None, :]))
    min_matrix = np.minimum.outer(times, times)
    exp_min_term1 = np.exp(2 * kappa1 * min_matrix) - 1
    exp_min_term2 = np.exp(2 * kappa2 * min_matrix) - 1

    term1 = (Psi1_values[:, None] * Psi1_values[None, :]) * (sigma1**2 / (2 * kappa1)) *
    exp_term1 * exp_min_term1
    term2 = (Psi1_values[:, None] * Psi2_values[None, :]) * ((rho * sigma1 * sigma2) / (
    kappa1 + kappa2)) * np.exp(-(kappa1 * times[:, None] + kappa2 * times[None, :])) * (np.
    exp((kappa1 + kappa2) * min_matrix) - 1)
    term3 = (Psi1_values[None, :] * Psi2_values[:, None]) * ((rho * sigma1 * sigma2) / (
    kappa1 + kappa2)) * np.exp(-(kappa1 * times[None, :] + kappa2 * times[:, None])) * (np.
    exp((kappa1 + kappa2) * min_matrix) - 1)
    term4 = (Psi2_values[:, None] * Psi2_values[None, :]) * (sigma2**2 / (2 * kappa2)) *
    exp_term2 * exp_min_term2

    cov_matrix = term1 + term2 + term3 + term4
    return cov_matrix + np.eye(len(times))*sigma_noise**2
```

Listing 12: Mean and Covariance Functions for a Multi-Curve Model

```python
def is_positive_definite(A):
    # Check symmetry
    if not np.allclose(A, A.T):
        return False

    # Check if positive definite via eigenvalues
    eigenvalues = np.linalg.eigvalsh(A)
    return np.all(eigenvalues > 0)

def nearest_positive_definite(mat):
    B = (mat + mat.T) / 2  # ensure symmetry
    _, s, V = np.linalg.svd(B)
    H = V @ np.diag(np.maximum(s, 0)) @ V.T
    A2 = (B + H) / 2
    A2 = (A2 + A2.T) / 2  # symmetry

    # Ensure all eigenvalues are positive
    p = 1
    while np.min(np.linalg.eigvalsh(A2)) <= 0:
        A2 += np.eye(mat.shape[0]) * p
        p *= 10

    return A2
```

Listing 13: Verification and Correction to Obtain Positive Definite Matrices

```python
cov_matrix_deltadelta = covariance_matrix_delta(times, kappa1, kappa2, sigma1, sigma2, rho,
    T)
cov_matrix_00 = covariance_matrix(times, kappa1, sigma1, T)
cov_matrix_d0 = cross_covariance_sigmad0(times, kappa1, kappa2, sigma1, sigma2, rho, T)
cov_matrix_0d = cross_covariance_sigma0d(times, kappa1, kappa2, sigma1, sigma2, rho, T)

samples = multivariate_normal.rvs(mean=mean_vector, cov=cov_matrix_00, size=1000)
samples_delta = multivariate_normal.rvs(mean=mean_vector_delta, cov=cov_matrix_deltadelta,
    size=1000)

data_indices = np.arange(0, 252)
times = np.linspace(0, T, 252)

log_delta_bond_prices = samples_delta[8]

# Split into training and validation dataset (70% training, 30% validation)
train_size = int(0.7 * len(data_indices))
train_indices = data_indices[:train_size]
validation_indices = data_indices[train_size:]

# Extract training and validation data
t_train = times[train_indices]
y_train = log_delta_bond_prices[train_indices]
t_validation = times[validation_indices]
t_predict = t_validation
y_validation_delta = log_delta_bond_prices[validation_indices]

# Calculate the mean and kernel for training data
mean_train = mean_function_delta(t_train, r01, r02, kappa1, kappa2, sigma1, sigma2, theta1,
    theta2)
cov_train = covariance_matrix_delta(t_train, kappa1, kappa2, sigma1, sigma2, rho, T) + np.
```

```
    eye(len(t_train)) * sigma_noise**2

# Calculate the mean and kernel for validation data
mean_validation = mean_function_delta(t_validation, r01, r02, kappa1, kappa2, sigma1, sigma2
    , theta1, theta2)
cov_validation_cross = cross_covariance_matrix_delta(t_train, t_predict, kappa1, kappa2,
    sigma1, sigma2, rho, T)
cov_validation = covariance_matrix_delta(t_validation, kappa1, kappa2, sigma1, sigma2, rho,
    T) + np.eye(len(t_validation)) * sigma_noise**2

# Prediction using GP
mu_validation_given_train = mean_validation + cov_validation_cross.T @ inv(cov_train) @ (
    y_train - mean_train)
Sigma_validation_given_train = cov_validation - cov_validation_cross.T @ inv(cov_train) @
    cov_validation_cross

# Predicted values
y_validation_predicted_delta = np.random.multivariate_normal(mu_validation_given_train,
    Sigma_validation_given_train)
```

Listing 14: Simulation and Prediction in a Multi-Curve Framework

```
# Calculate mean and covariance
mean_vec = mean_function_delta(times, r01, r02, kappa1, kappa2, sigma1, sigma2, theta1,
    theta2)
cov_mat = covariance_matrix_delta(times, kappa1, kappa2, sigma1, sigma2, rho, T)

# Sample from the Gaussian process
sample_path = np.random.multivariate_normal(mean_vec, cov_mat)

# Calculate 95% confidence intervals
std_dev = np.sqrt(np.diag(cov_mat))
ci_upper = mean_vec + 1.96 * std_dev
ci_lower = mean_vec - 1.96 * std_dev

# Plot
plt.figure(figsize=(12, 6))
plt.plot(times, mean_vec, 'b', label='Prior Mean')
plt.fill_between(times, ci_lower, ci_upper, color='blue', alpha=0.2, label='95% CI')
plt.plot(times, sample_path, 'r', label='Simulated Log-Bond Price')
plt.title('Predictive Distribution and Simulated Bond Prices for a 1-Year Maturity')
plt.xlabel('Time (years)')
plt.ylabel('Log Bond Prices')
plt.legend()
plt.show()
```

Listing 15: Simulation and Visualization of Predictive Distribution and Simulated Bond Prices

```
# Calculate MSE and SMSE
sigma_y_tilde_squared = np.var(y_validation_delta)

mse = np.mean((y_validation_delta - y_validation_predicted_delta) ** 2)

smse = mse / sigma_y_tilde_squared

print(f"Mean Squared Error (MSE): {mse}")
print(f"Standardized Mean Squared Error (SMSE): {smse}")
```

Listing 16: Calculation of MSE and SMSE for Prediction Evaluation