Modelling and Model Calibration Coursework - Elliot Jones

April 20, 2025

0.1 Academic integrity statement

This coursework is declared as my own work and any re-used work or the use of work of others has been appropriately referenced.

0.2 Question 1 i)

The Vasicek model is define as

$$dr_t = (b - a \cdot r_t)dt + \sigma dW_t \tag{1}$$

this is loosely interpreted as, for δt small,

$$r_{t+\delta t} = r_t + (b - a \cdot r_t)\delta t + \sigma\sqrt{\delta t} \cdot Z \tag{2}$$

where Z is standard normal.

in the Vasicek model, the price of a zero-coupon bond with maturity T is as follows:

$$p(t,T) = \mathbb{E}^{\mathbb{Q}}[e^{-\int_t^T r_s ds}|\mathcal{F}_t] = exp(A(t,T) - B(t,T)r_t) \tag{3}$$

where

$$B(t,T) = \frac{1}{a}[1 - e^{-a(T-t)}] \tag{4}$$

$$A(t,T) = \frac{[B(t,T) - (T-t)](ab - \frac{1}{2}\sigma^2)}{a^2} - \frac{\sigma^2 B(t,T)^2}{4a} \tag{5}$$

we now want to calibrate this model for the three parameters (a, b, σ) to do this we have a "function" that takes the parameters of the Vasicek model and gives us Bond prices at given maturities:

$$(a, b, \sigma) \mapsto P^{VAS}(t, T_i)(a, b, \sigma; r_t) \tag{6}$$

We want to define an error function to use in the Levenberg-Marquardt algorithm to calibrate (a, b, σ), the LM algorithem interpolates between the Gauss-Newton algorithm and the method of gradient descent. We want to choose parameters (a, b, σ) to get as close as possible to the bond prices implied by the price data given:

$$minimise_{a,b,\sigma} \sum_i^I (P^{VAS}(t, T_i)(a, b, \sigma; r_t) - P^{data}(t, T_i))^2 \tag{7}$$

Notice that this is a nonlinear squares optimisation problem, perfect for the LM algorithm. Starting with an intial guess for (a, b, σ) we would then compute the jacobian matrix and update the parameters using

$$(\mathbf{J}^T\mathbf{J} + \lambda I)\delta = \mathbf{J}^T r \tag{8}$$

where J is the jacobian matrix, r are the residual errors and λ is the damping factor adjusting between Gauss-Newton and gradient descent. This will then iterate until convergence.

To determine the performance of some of our predictions our through this coursework we will use the Root Mean Square Error (RMSE) multiple times. This is a commonly used metric that measures the average difference between predicted values and true values and is defined as follows,

$$RSME = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(y_i^{pred} - y_i^{true})^2} \tag{9}$$

where - y_i^{true} is the true values - y_i^{pred} is the prediction values - n is the number of samples

This will allow us to empirically evaluate our predictions and indicate how far our model's predictions are off on average.

[3]:
```
pip install Quantlib
```

```
Requirement already satisfied: Quantlib in /opt/conda/lib/python3.11/site-
packages (1.37)
Note: you may need to restart the kernel to use updated packages.
```

[4]:
```
pip install tensorflow
```

```
Collecting tensorflow
  Using cached tensorflow-2.19.0-cp311-cp311-
manylinux_2_17_x86_64.manylinux2014_x86_64.whl.metadata (4.1 kB)
Collecting absl-py>=1.0.0 (from tensorflow)
  Using cached absl_py-2.2.2-py3-none-any.whl.metadata (2.6 kB)
Collecting astunparse>=1.6.0 (from tensorflow)
  Using cached astunparse-1.6.3-py2.py3-none-any.whl.metadata (4.4 kB)
Collecting flatbuffers>=24.3.25 (from tensorflow)
  Using cached flatbuffers-25.2.10-py2.py3-none-any.whl.metadata (875 bytes)
Collecting gast!=0.5.0,!=0.5.1,!=0.5.2,>=0.2.1 (from tensorflow)
  Using cached gast-0.6.0-py3-none-any.whl.metadata (1.3 kB)
Collecting google-pasta>=0.1.1 (from tensorflow)
  Using cached google_pasta-0.2.0-py3-none-any.whl.metadata (814 bytes)
Collecting libclang>=13.0.0 (from tensorflow)
  Using cached libclang-18.1.1-py2.py3-none-manylinux2010_x86_64.whl.metadata
(5.2 kB)
Collecting opt-einsum>=2.3.2 (from tensorflow)
  Using cached opt_einsum-3.4.0-py3-none-any.whl.metadata (6.3 kB)
Requirement already satisfied: packaging in /opt/conda/lib/python3.11/site-
packages (from tensorflow) (24.0)
Requirement already satisfied:
```

```
protobuf!=4.21.0,!=4.21.1,!=4.21.2,!=4.21.3,!=4.21.4,!=4.21.5,<6.0.0dev,>=3.20.3
in /opt/conda/lib/python3.11/site-packages (from tensorflow) (4.25.3)
Requirement already satisfied: requests<3,>=2.21.0 in
/opt/conda/lib/python3.11/site-packages (from tensorflow) (2.32.3)
Requirement already satisfied: setuptools in /opt/conda/lib/python3.11/site-
packages (from tensorflow) (69.5.1)
Requirement already satisfied: six>=1.12.0 in /opt/conda/lib/python3.11/site-
packages (from tensorflow) (1.16.0)
Collecting termcolor>=1.1.0 (from tensorflow)
  Using cached termcolor-3.0.1-py3-none-any.whl.metadata (6.1 kB)
Requirement already satisfied: typing-extensions>=3.6.6 in
/opt/conda/lib/python3.11/site-packages (from tensorflow) (4.11.0)
Collecting wrapt>=1.11.0 (from tensorflow)
  Using cached wrapt-1.17.2-cp311-cp311-
manylinux_2_5_x86_64.manylinux1_x86_64.manylinux_2_17_x86_64.manylinux2014_x86_6
4.whl.metadata (6.4 kB)
Requirement already satisfied: grpcio<2.0,>=1.24.3 in
/opt/conda/lib/python3.11/site-packages (from tensorflow) (1.62.2)
Collecting tensorboard~=2.19.0 (from tensorflow)
  Using cached tensorboard-2.19.0-py3-none-any.whl.metadata (1.8 kB)
Collecting keras>=3.5.0 (from tensorflow)
  Using cached keras-3.9.2-py3-none-any.whl.metadata (6.1 kB)
Requirement already satisfied: numpy<2.2.0,>=1.26.0 in
/opt/conda/lib/python3.11/site-packages (from tensorflow) (1.26.4)
Requirement already satisfied: h5py>=3.11.0 in /opt/conda/lib/python3.11/site-
packages (from tensorflow) (3.11.0)
Collecting ml-dtypes<1.0.0,>=0.5.1 (from tensorflow)
  Using cached ml_dtypes-0.5.1-cp311-cp311-
manylinux_2_17_x86_64.manylinux2014_x86_64.whl.metadata (21 kB)
Collecting tensorflow-io-gcs-filesystem>=0.23.1 (from tensorflow)
  Using cached tensorflow_io_gcs_filesystem-0.37.1-cp311-cp311-
manylinux_2_17_x86_64.manylinux2014_x86_64.whl.metadata (14 kB)
Requirement already satisfied: wheel<1.0,>=0.23.0 in
/opt/conda/lib/python3.11/site-packages (from astunparse>=1.6.0->tensorflow)
(0.43.0)
Requirement already satisfied: rich in /opt/conda/lib/python3.11/site-packages
(from keras>=3.5.0->tensorflow) (13.9.3)
Collecting namex (from keras>=3.5.0->tensorflow)
  Using cached namex-0.0.8-py3-none-any.whl.metadata (246 bytes)
Collecting optree (from keras>=3.5.0->tensorflow)
  Using cached optree-0.15.0-cp311-cp311-
manylinux_2_17_x86_64.manylinux2014_x86_64.whl.metadata (48 kB)
Requirement already satisfied: charset-normalizer<4,>=2 in
/opt/conda/lib/python3.11/site-packages (from requests<3,>=2.21.0->tensorflow)
(3.3.2)
Requirement already satisfied: idna<4,>=2.5 in /opt/conda/lib/python3.11/site-
packages (from requests<3,>=2.21.0->tensorflow) (3.6)
Requirement already satisfied: urllib3<3,>=1.21.1 in
```

```
/opt/conda/lib/python3.11/site-packages (from requests<3,>=2.21.0->tensorflow)
(2.2.3)
Requirement already satisfied: certifi>=2017.4.17 in
/opt/conda/lib/python3.11/site-packages (from requests<3,>=2.21.0->tensorflow)
(2024.8.30)
Collecting markdown>=2.6.8 (from tensorboard~=2.19.0->tensorflow)
  Using cached markdown-3.8-py3-none-any.whl.metadata (5.1 kB)
Collecting tensorboard-data-server<0.8.0,>=0.7.0 (from
tensorboard~=2.19.0->tensorflow)
  Using cached tensorboard_data_server-0.7.2-py3-none-
manylinux_2_31_x86_64.whl.metadata (1.1 kB)
Collecting werkzeug>=1.0.1 (from tensorboard~=2.19.0->tensorflow)
  Using cached werkzeug-3.1.3-py3-none-any.whl.metadata (3.7 kB)
Requirement already satisfied: MarkupSafe>=2.1.1 in
/opt/conda/lib/python3.11/site-packages (from
werkzeug>=1.0.1->tensorboard~=2.19.0->tensorflow) (2.1.5)
Requirement already satisfied: markdown-it-py>=2.2.0 in
/opt/conda/lib/python3.11/site-packages (from rich->keras>=3.5.0->tensorflow)
(3.0.0)
Requirement already satisfied: pygments<3.0.0,>=2.13.0 in
/opt/conda/lib/python3.11/site-packages (from rich->keras>=3.5.0->tensorflow)
(2.17.2)
Requirement already satisfied: mdurl~=0.1 in /opt/conda/lib/python3.11/site-
packages (from markdown-it-py>=2.2.0->rich->keras>=3.5.0->tensorflow) (0.1.2)
Using cached
tensorflow-2.19.0-cp311-cp311-manylinux_2_17_x86_64.manylinux2014_x86_64.whl
(644.9 MB)
Using cached absl_py-2.2.2-py3-none-any.whl (135 kB)
Using cached astunparse-1.6.3-py2.py3-none-any.whl (12 kB)
Using cached flatbuffers-25.2.10-py2.py3-none-any.whl (30 kB)
Using cached gast-0.6.0-py3-none-any.whl (21 kB)
Using cached google_pasta-0.2.0-py3-none-any.whl (57 kB)
Using cached keras-3.9.2-py3-none-any.whl (1.3 MB)
Using cached libclang-18.1.1-py2.py3-none-manylinux2010_x86_64.whl (24.5 MB)
Using cached
ml_dtypes-0.5.1-cp311-cp311-manylinux_2_17_x86_64.manylinux2014_x86_64.whl (4.7
MB)
Using cached opt_einsum-3.4.0-py3-none-any.whl (71 kB)
Using cached tensorboard-2.19.0-py3-none-any.whl (5.5 MB)
Using cached tensorflow_io_gcs_filesystem-0.37.1-cp311-cp311-
manylinux_2_17_x86_64.manylinux2014_x86_64.whl (5.1 MB)
Using cached termcolor-3.0.1-py3-none-any.whl (7.2 kB)
Using cached wrapt-1.17.2-cp311-cp311-
manylinux_2_5_x86_64.manylinux1_x86_64.manylinux_2_17_x86_64.manylinux2014_x86_6
4.whl (83 kB)
Using cached markdown-3.8-py3-none-any.whl (106 kB)
Using cached tensorboard_data_server-0.7.2-py3-none-manylinux_2_31_x86_64.whl
(6.6 MB)
```

```
Using cached werkzeug-3.1.3-py3-none-any.whl (224 kB)
Using cached namex-0.0.8-py3-none-any.whl (5.8 kB)
Using cached
optree-0.15.0-cp311-cp311-manylinux_2_17_x86_64.manylinux2014_x86_64.whl (410
kB)
Installing collected packages: namex, libclang, flatbuffers, wrapt, werkzeug,
termcolor, tensorflow-io-gcs-filesystem, tensorboard-data-server, optree, opt-
einsum, ml-dtypes, markdown, google-pasta, gast, astunparse, absl-py,
tensorboard, keras, tensorflow
Successfully installed absl-py-2.2.2 astunparse-1.6.3 flatbuffers-25.2.10
gast-0.6.0 google-pasta-0.2.0 keras-3.9.2 libclang-18.1.1 markdown-3.8 ml-
dtypes-0.5.1 namex-0.0.8 opt-einsum-3.4.0 optree-0.15.0 tensorboard-2.19.0
tensorboard-data-server-0.7.2 tensorflow-2.19.0 tensorflow-io-gcs-
filesystem-0.37.1 termcolor-3.0.1 werkzeug-3.1.3 wrapt-1.17.2
Note: you may need to restart the kernel to use updated packages.
```

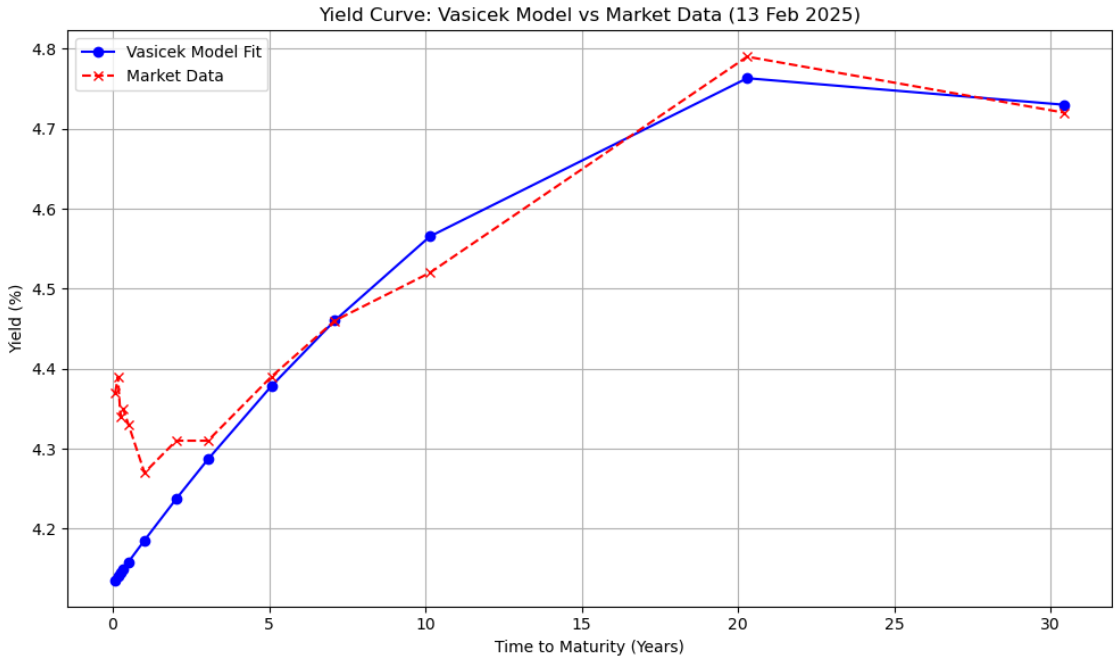
[20]:
```python
import numpy as np
import matplotlib.pyplot as plt
import QuantLib as ql # See https://www.quantlib.org/install.shtml
import pandas as pd
from scipy.optimize import least_squares
import scipy.interpolate as spi
from scipy.interpolate import interp1d
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import train_test_split
from sklearn.decomposition import PCA
from tensorflow.keras import models, layers, callbacks
```

[11]:
```python
# Load data
dat = pd.read_csv('https://people.bath.ac.uk/mapamgc/AT/data/
 ↪daily-treasury-rates-ql.csv', index_col='Date')

# Select the rates for a specified day (Feb 13th 2025)
latest_rates = dat.iloc[0]

# Set evaluation date
today = ql.Date(13, 2, 2025)
ql.Settings.instance().evaluationDate = today
dayCounter = ql.Actual360()

# Extract maturities from column names
maturity_labels = latest_rates.index
maturities = []
for label in maturity_labels:
    num, unit = int(label[:-1]), label[-1].lower()
    if unit == 'm':
        maturities.append(ql.Period(num, ql.Months))
```

```
    elif unit == 'y':
        maturities.append(ql.Period(num, ql.Years))

# Convert to QuantLib Dates
maturity_dates = [today + p for p in maturities]

# Convert rates to numpy array
observed_rates = latest_rates.values.astype(float)

# Compute time to maturities
times = np.array([dayCounter.yearFraction(today, m) for m in maturity_dates])

# Convert observed rates to discount factors
observed_discount_factors = np.exp(-observed_rates / 100 * times)

# Define the Vasicek model functions
def B_vas(tau, a):
    return (1 - np.exp(-a * tau)) / a

def A_vas(tau, a, b, sigma):
    B = B_vas(tau, a)
    term1 = (b - (sigma**2) / (2 * a**2)) * (B - tau)
    term2 = (sigma**2 * B**2) / (4 * a)
    return np.exp(term1 - term2)

def P_vas(tau, r0, a, b, sigma):
    B = B_vas(tau, a)
    A = A_vas(tau, a, b, sigma)
    return A * np.exp(-B * r0)

# Objective function to model prices
def objective(params):
    b, a, sigma, r0 = params
    model_prices = np.array([P_vas(t, r0, a, b, sigma) for t in times])
    return model_prices - observed_discount_factors

# Initial guess for parameters (b, a, sigma, r0)
initial_guess = [0.05, 0.1, 0.02, 0.01]

# Least squares optimisation (LM algorithm) with bounds
result = least_squares(objective, initial_guess, method='lm')

# Extract optimised parameters
b_opt, a_opt, sigma_opt, r0_opt = result.x

# Compute bond prices and yield curve using calibrated parameters
```

```
optimized_prices = np.array([P_vas(t, r0_opt, a_opt, b_opt, sigma_opt) for t in␣
 ↪times])
yields = -np.log(optimized_prices) / times

# Output optimised parameters and RMSE
print(f"Optimized Vasicek parameters:")
print(f" b = {b_opt:.6f}")
print(f" a = {a_opt:.6f}")
print(f" sigma = {sigma_opt:.6f}")
print(f" r0 = {r0_opt:.6f}")
rmse = np.sqrt(np.mean(result.fun**2))
print(f" RMSE = {rmse:.6f}")

# Plot model vs market
plt.figure(figsize=(10, 6))
plt.plot(times, yields * 100, label="Vasicek Model Fit", marker='o',␣
 ↪color='blue')
plt.plot(times, observed_rates, label="Market Data", linestyle='dashed',␣
 ↪marker='x', color='red')
plt.xlabel("Time to Maturity (Years)")
plt.ylabel("Yield (%)")
plt.title("Yield Curve: Vasicek Model vs Market Data (13 Feb 2025)")
plt.legend()
plt.grid(True)
plt.tight_layout()
plt.show()
```

```
Optimized Vasicek parameters:
 b = 1.111071
 a = 0.001021
 sigma = 0.008329
 r0 = 0.041314
 RMSE = 0.001174
```



0.2.1 Analysis of results

The Vasicek model above shows a strong fit to the observed yield curve and close alignment between the Vasicek Model yields and market yield across most maturities. The low Root Mean Square Error (RMSE) value is generally extremely low and indicates that this model has captures the general structure of the curve. However, there is a slight underestimation in the shorter maturities. This is a common limitation for mean-reverting models such as the Vasicek model. Overall these results suggest that the Vasicek model is only effective when fitting yields curves with medium to long-term maturities.

0.3 Question 1 ii)

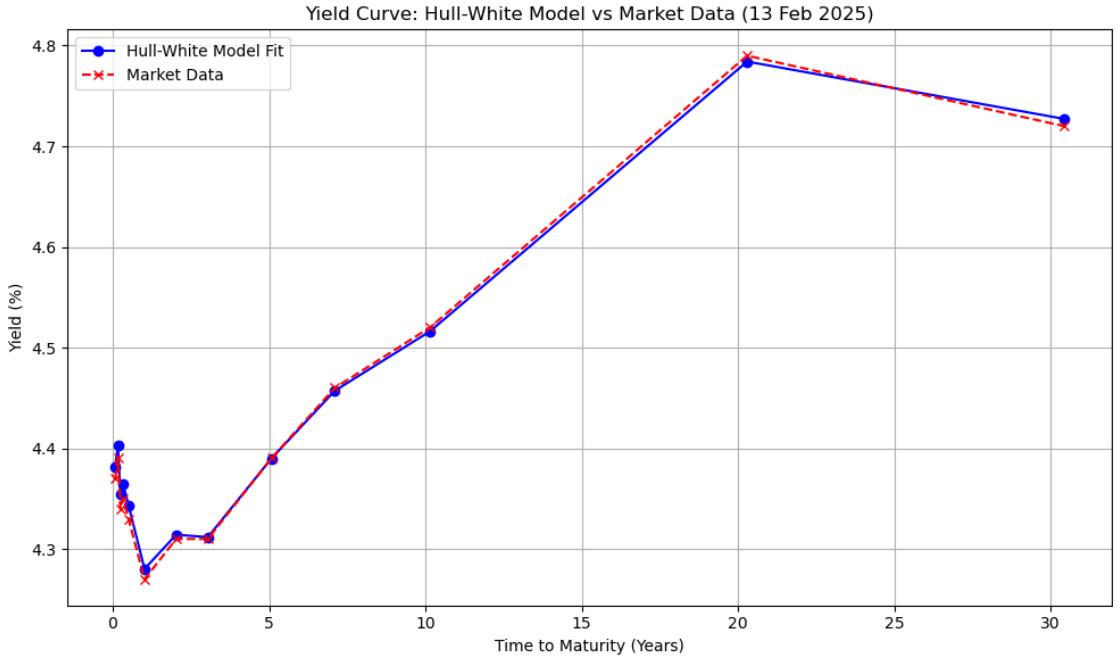
[12]:
```
# Load data
dat = pd.read_csv('https://people.bath.ac.uk/mapamgc/AT/data/
 ↪daily-treasury-rates-ql.csv', index_col='Date')
latest_rates = dat.iloc[0]

# Set evaluation date
today = ql.Date(13, 2, 2025)
ql.Settings.instance().evaluationDate = today
dayCounter = ql.Actual360()

# Parse maturities
maturity_labels = latest_rates.index
maturities = []
```

```
for label in maturity_labels:
    num, unit = int(label[:-1]), label[-1].lower()
    if unit == 'm':
        maturities.append(ql.Period(num, ql.Months))
    elif unit == 'y':
        maturities.append(ql.Period(num, ql.Years))
maturity_dates = [today + p for p in maturities]

# Times to maturity
times = np.array([dayCounter.yearFraction(today, m) for m in maturity_dates])
observed_rates = latest_rates.values.astype(float)
observed_discount_factors = np.exp(-observed_rates / 100 * times)

# Interpolate forward curve using numerical approximation as we are not allowed␣
 ↪to use Quantlib
zero_curve = interp1d(times, observed_rates / 100, kind='cubic',␣
 ↪fill_value="extrapolate")
def forward_rate(t):
    dt = 1e-5
    return (zero_curve(t + dt) * (t + dt) - zero_curve(t) * t) / dt

# Define the Hull-White model
def theta_hw(t, a, sigma):
    f0 = forward_rate(t)
    df0 = (forward_rate(t + 1e-5) - forward_rate(t)) / 1e-5 # approximate of␣
 ↪the derivative
    return df0 + a * f0 + (sigma**2 / (2 * a)) * (1 - np.exp(-2 * a * t))

def B_hw(tau, a):
    return (1 - np.exp(-a * tau)) / a

def A_hw(t, T, a, sigma):
    tau = T - t
    B = B_hw(tau, a)
    integral_theta_B = np.array([theta_hw(s, a, sigma) * B_hw(T - s, a) for s␣
 ↪in np.linspace(t, T, 100)])
    dt = (T - t) / 100
    int_theta_B = np.sum(integral_theta_B) * dt
    term1 = -int_theta_B
    term2 = (B - (T - t)) * (sigma**2 / (2 * a**2))
    term3 = (sigma**2 * B**2) / (4 * a)
    return np.exp(term1 - term2 - term3)

def P_hw(t, T, r0, a, sigma):
    B = B_hw(T - t, a)
    A = A_hw(t, T, a, sigma)
    return A * np.exp(-B * r0)
```

```python
# Objective function for Hull-White model
def objective_hw(params):
    a, sigma, r0 = params
    model_prices = np.array([P_hw(0, T, r0, a, sigma) for T in times])
    return model_prices - observed_discount_factors

# Initial parameters/guess (a, sigma, r)
initial_guess = [0.1, 0.01, 0.01]

# Fit the Hull-White model
result = least_squares(objective_hw, initial_guess, method='lm')

# Extract optimised parameters
a_opt, sigma_opt, r0_opt = result.x

# Final bond prices and model yields
optimized_prices = np.array([P_hw(0, T, r0_opt, a_opt, sigma_opt) for T in␣
 ↪times])
yields = -np.log(optimized_prices) / times

# Output results
print(f"Optimised Hull-White parameters:")
print(f" a = {a_opt:.6f}")
print(f" sigma = {sigma_opt:.6f}")
print(f" r0 = {r0_opt:.6f}")
rmse = np.sqrt(np.mean(result.fun**2))
print(f" RMSE = {rmse:.6f}")

# Plot model vs market
plt.figure(figsize=(10, 6))
plt.plot(times, yields * 100, label="Hull-White Model Fit", marker='o',␣
 ↪color='blue')
plt.plot(times, observed_rates, label="Market Data", linestyle='dashed',␣
 ↪marker='x', color='red')
plt.xlabel("Time to Maturity (Years)")
plt.ylabel("Yield (%)")
plt.title("Yield Curve: Hull-White Model vs Market Data (13 Feb 2025)")
plt.legend()
plt.grid(True)
plt.tight_layout()
plt.show()
```

```
Optimised Hull-White parameters:
 a = 3.043980
 sigma = -0.001077
 r0 = 0.041374
```

```
RMSE = 0.000211
```



In the Hull-White model we have the short rate given by the Stochastic Differential Equation:

$$dr_t = (\theta(t) - ar_t)dt + \sigma dW_t \tag{10}$$

$\theta(t)$ is a deterministic function that acts as a time-dependent mean-reversion level. This allows the model to exactly match an observed yield curve. Since this is a given function and not random, we must choose how to represent $\theta(t)$, this can be done in a number of ways,

0.3.1 Method 1 - Exact Analytical Solution From Market Data.

Within this method, we use the fact that the Hull-White model can reproduce any initial yield curve exactly by directly deriving $\theta(t)$ from it. That is, we assume that there exists a $\theta(t)$ that perfectly fits the observed yield curve without needing to parametrise or estimate it directly. From this method we gain:

$$\theta(t) = \frac{\partial f(0,t)}{\partial t} + af(0,t) + \frac{\sigma^2}{2a}(1 - e^{-2at}) \tag{11}$$

where $f(0,t)$ is the instant forward rate curve computed from market data, $f(0,t) = \frac{d}{dt}[t \cdot z(t)]$. The pros to this method are that there are no extra parameters, it is guaranteed to match the yield curve and it is the most accurate calibration to today's market. However, the cons are that it requires a smooth and differentiable forward rate curve. It can also be unstable if the market data is noisy or there is a limited amount of market information.

0.3.2 Method 2 - Piecewise Constant

The idea of this method is to divide the time into discrete intervals and assume that $\theta(t)$ is constant on each. e.g. $\theta(t) = \theta_i$ for $t \in [t_i, t_{t+1})$

The pros of this method is that it is computationally simple and intuitive, it grants more flexibility as the number of intervals increases, and it can be fitted to our data using standard optimisation. However, the model can introduce a lot of parameters leading to a risk of overfitting and potential discontinuities in the short rate.

0.3.3 Method 3 - Parametric Functions (e.g. Polynomial, Exponential, Linear)

Using a parametric function such as a linear function ($ (t) = + t), $Polynomial$((t)= ^n_{k=0} _kt^k), $orExponential$((t) = e^{- t}+ $) would be beneficial as the function would be extremely compact, only using a few parameters. Using one of these functions will also ensure that theta is smooth and differentiable, allowing it to be computationally easy to calibrate. However, using these simpler functions will lead to a less flexible model, leading to the function not fitting the market curves exactly.

0.3.4 Method 4 - Cubic Spline Interpolation

We can also define $\theta(t)$ as a set of nodes and interpolate this using splines. The spline method fits piecewise polynomials between each pair of data points instead of using a single high-degree polynomial for the entire data set. This method would lead to much more flexibility and smoothness and fewer parameters than the piecewise constant method. However, this method requires careful placement of the nodes as it can produce overfitting of $\theta(t)$ if it is not regularised properly.

0.3.5 Implementation of Method 1

In our implementation of the Hull-White model, we decided to use the exact analytical solution approach, where the time-dependent mean reversion level $\theta(t)$ is not parameterised directly but instead it is inferred analytically from the initial interest rate using the model's bond pricing formula. This allowed us to exactly calibrate a, σ and r_0 to fit the best observed market bond prices. The advantages of this model allowed us to exactly fit the initial yield curve, ensuring that the Hull-White model can produce the observed market yield curve at time $t = 0$. This method also allowed our optimisation problem to become less computationally complex as there were fewer parameters (a, σ, r_0) to calibrate. However, this method meant that we did not have an explicit form of $\theta(t)$, making it potentially harder to interpret the meaning of the drift. Additionally, since the model is calibrated to fit today's curve (13/02/2025) exactly and $\theta(t)$ is implied by the current term, the future rates are determined purely by the parameters a, σ and r_0. This could potentially lead to future shifts in curve shape not being captured fully. This method can also be very sensitive to noisy or irregular market data, resulting in an unrealistic model. For our code we deemed this method optimal as it provided a high accuracy and efficiency at the cost of potential overfitting to noisy curve data.

0.3.6 Results of the model

From our implementation of the Hull-White model we can see that it is extremely close to the market data given. This is as expected as one of the key features of this model is that it can be fully calibrated to match the entire term structure as that is what $\theta(t)$ is designed to do. We can also

observe that our implementation of $\theta(t)$ provided an extremely small Root Mean Squared Error (RSME), indicating an extremely accurate result in predicting the market prices and indicating that we chose a sensible method to represent $\theta(t)$.

0.4 Question 2

[34]:
```
# Load data
dat = pd.read_csv('https://people.bath.ac.uk/mapamgc/AT/data/
 ↪daily-treasury-rates-ql.csv', index_col='Date')
latest_rates = dat.iloc[0]

# Set evaluation date (13th Feb 2025)
today = ql.Date(13, 2, 2025)
ql.Settings.instance().evaluationDate = today
dayCounter = ql.Actual360()

# Parse maturities from labels
maturity_labels = latest_rates.index
maturities = []
for label in maturity_labels:
    num, unit = int(label[:-1]), label[-1].lower()
    if unit == 'm':
        maturities.append(ql.Period(num, ql.Months))
    elif unit == 'y':
        maturities.append(ql.Period(num, ql.Years))
maturity_dates = [today + p for p in maturities]

# Compute time to maturity and extract observed rates.
times = np.array([dayCounter.yearFraction(today, m) for m in maturity_dates])
observed_rates = latest_rates.values.astype(float)
observed_discount_factors = np.exp(-observed_rates / 100 * times)

# Interpolate forward curve using numerical approximation as we are not allowed␣
 ↪to use Quantlib
zero_curve = interp1d(times, observed_rates / 100, kind='cubic',␣
 ↪fill_value="extrapolate")
def forward_rate(t):
    dt = 1e-5
    return (zero_curve(t + dt) * (t + dt) - zero_curve(t) * t) / dt

# Define Hull-White model ensuring to Numerically integrate over the interval␣
 ↪(t,T) in A_hw.
def theta_hw(t, a, sigma):
    f0 = forward_rate(t)
    df0 = (forward_rate(t + 1e-5) - forward_rate(t)) / 1e-5
    return df0 + a * f0 + (sigma**2 / (2 * a)) * (1 - np.exp(-2 * a * t))
```

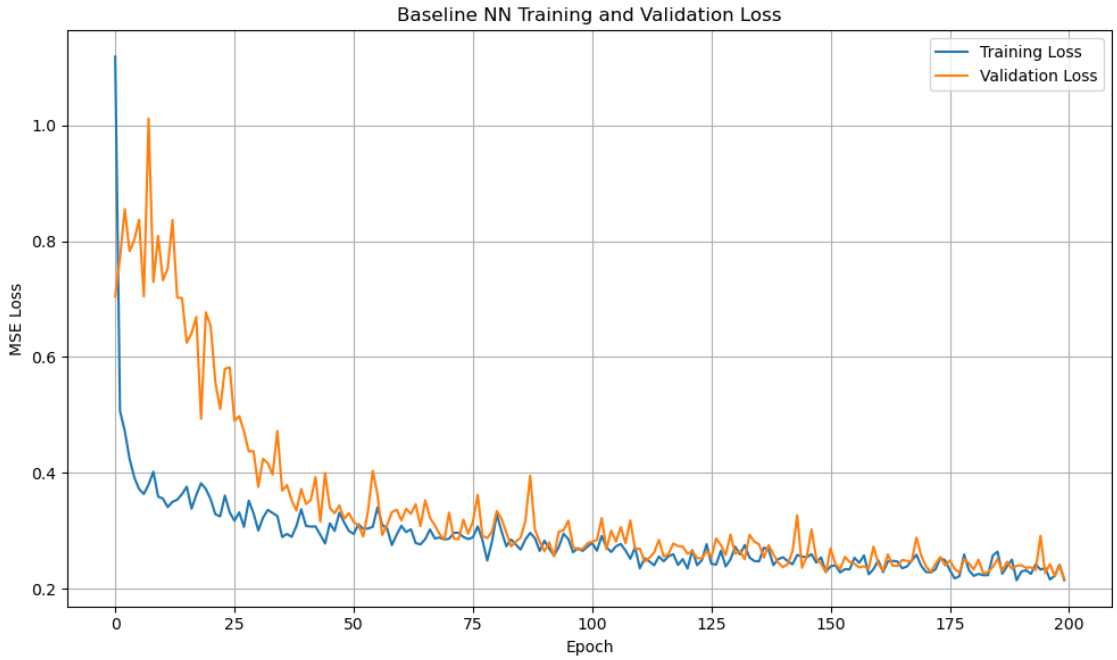
```
def B_hw(tau, a):
    return (1 - np.exp(-a * tau)) / a

def A_hw(t, T, a, sigma):
    tau = T - t
    B = B_hw(tau, a)
    integral_theta_B = np.array([theta_hw(s, a, sigma) * B_hw(T - s, a) for s␣
 ↪in np.linspace(t, T, 100)])
    dt = (T - t) / 100
    int_theta_B = np.sum(integral_theta_B) * dt
    term1 = -int_theta_B
    term2 = (B - (T - t)) * (sigma**2 / (2 * a**2))
    term3 = (sigma**2 * B**2) / (4 * a)
    return np.exp(term1 - term2 - term3)

def P_hw(t, T, r0, a, sigma):
    B = B_hw(T - t, a)
    A = A_hw(t, T, a, sigma)
    return A * np.exp(-B * r0)


# Generate synthetic training data for the Neural Network (Uniform Sampling)
num_samples = 1200

# Define plausible/realistic parameter ranges for the Hull-White model as␣
 ↪described in Hernandez (2016).
a_vals = np.random.uniform(0.001, 3.5, num_samples)        # Mean reversion
sigma_vals = np.random.uniform(0.0001, 1, num_samples)       # Volatility
r0_vals = np.random.uniform(0.001, 0.1, num_samples)         # Initial short␣
 ↪rate

# Stack parameters together.
params = np.column_stack((a_vals, sigma_vals, r0_vals))

def compute_yields(params):
    """
    For each set of Hull-White parameters we compute the corresponding yield␣
 ↪curve.
    The yields are calculated as y = -log(price) / T as we assume continuous␣
 ↪compounding.
    """
    yields = []
    for a, sigma, r0 in params:
        prices = np.array([P_hw(0, T, r0, a, sigma) for T in times])
        y = -np.log(prices) / times
        yields.append(y)
    return np.array(yields)
```

```
# Generate synthetic yield curves using the Hull-White model.
X_syn = compute_yields(params)

# Apply a log transformation to parameters to ensure positivity.
y_syn = np.log(params)


# Standardise the data and split into training and validation sets

scaler_X = StandardScaler()
scaler_y = StandardScaler()
X_scaled = scaler_X.fit_transform(X_syn)
y_scaled = scaler_y.fit_transform(y_syn)

# Split the synthetic dataset.
X_train, X_val, y_train, y_val = train_test_split(X_scaled, y_scaled,␣
 ↪test_size=0.2)

# Build and test the feed-forward Neural Network.
model_baseline = models.Sequential([
    layers.Input(shape=(X_train.shape[1],)),
    layers.Dense(256, activation='elu'),
    layers.BatchNormalization(),
    layers.Dense(256, activation='elu'),
#    layers.Dropout(0.1),
    layers.Dense(128, activation='elu'),
    layers.Dense(3)])

model_baseline.compile(optimizer='adam', loss='mse')
early_stop = callbacks.EarlyStopping(monitor='val_loss', patience=40,␣
 ↪restore_best_weights=True)

history_baseline = model_baseline.fit(X_train, y_train,
                                      validation_data=(X_val, y_val),
                                      epochs=200,
                                      batch_size=80,
                                      callbacks=[early_stop],
                                      verbose=0)

# Plotting of Neural validation and training loss
plt.figure(figsize=(10, 6))
plt.plot(history_baseline.history['loss'], label='Training Loss')
plt.plot(history_baseline.history['val_loss'], label='Validation Loss')
plt.xlabel("Epoch")
plt.ylabel("MSE Loss")
plt.title("Baseline NN Training and Validation Loss")
```

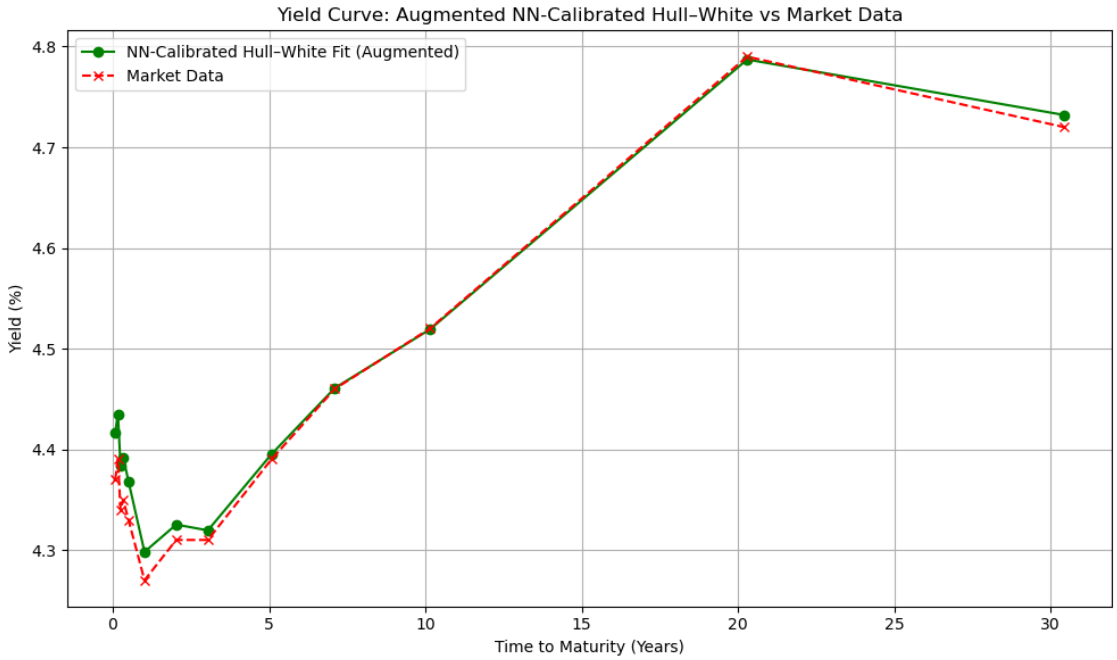
```
plt.legend()
plt.grid(True)
plt.tight_layout()
plt.show()


# Prepare the observed yield curve as the network input.
observed_yields = (observed_rates / 100).reshape(1, -1)
observed_scaled = scaler_X.transform(observed_yields)

# Predict the log-parameters, invert-scale, then apply exponential
pred_base_scaled = model_baseline.predict(observed_scaled)
pred_base_log    = scaler_y.inverse_transform(pred_base_scaled)
a_base, sigma_base, r0_base = np.exp(pred_base_log[0])

print("NN-Calibrated Hull-White parameters (Augmented):")
print(f" a = {a_base:.6f}")
print(f" sigma = {sigma_base:.6f}")
print(f" r0 = {r0_base:.6f}")

prices_base = np.array([P_hw(0, T, r0_base, a_base, sigma_base) for T in times])
yields_base = -np.log(prices_base) / times

# Plotting for the augmented Hull-White
plt.figure(figsize=(10, 6))
plt.plot(times, yields_base * 100, label="NN-Calibrated Hull-White Fit␣
 ↪(Augmented)", marker='o', color='green')
plt.plot(times, observed_rates, label="Market Data", linestyle='dashed',␣
 ↪marker='x', color='red')
plt.xlabel("Time to Maturity (Years)")
plt.ylabel("Yield (%)")
plt.title("Yield Curve: Augmented NN-Calibrated Hull-White vs Market Data")
plt.legend()
plt.grid(True)
plt.tight_layout()
plt.show()

# Apply PCA with a 99.9% variance threshold:
pca = PCA(n_components=0.999, svd_solver='full')
X_train_pca = pca.fit_transform(X_train)

# Print out the number of components retained
print(f"Retained {X_train_pca.shape[1]} components out of {X_train.shape[1]}␣
 ↪features.")

# Concatenate PCA scores with the standardised log-parameters.
combined_train = np.hstack([X_train_pca, y_train])
```

```
# Compute covariance matrix and mean vector of the combined training data.
cov_matrix = np.cov(combined_train, rowvar=False)
mean_combined = np.mean(combined_train, axis=0)

# Generate new synthetic samples from a multivariate normal distribution.
num_augmented = X_train.shape[0]
synthetic_combined = np.random.multivariate_normal(mean_combined, cov_matrix,␣
 ↪size=num_augmented)

# Split the synthetic combined samples.
num_pc = X_train_pca.shape[1]
X_synth_pca = synthetic_combined[:, :num_pc]  # Synthetic PCA scores for yield␣
 ↪curves.
y_synth_aug = synthetic_combined[:, num_pc:]  # Synthetic standardized␣
 ↪log-parameters.

# Inverse transform the synthetic PCA scores to retrieve the synthetic yield␣
 ↪curves.
X_synth_scaled_aug = pca.inverse_transform(X_synth_pca)
X_train_aug = np.vstack([X_train, X_synth_scaled_aug])
y_train_aug = np.vstack([y_train, y_synth_aug])

# Train the Neural Network using the augmented data
model_augmented = models.Sequential([
    layers.Input(shape=(X_train_aug.shape[1],)),
    layers.Dense(256, activation='elu'),
    layers.BatchNormalization(),
    layers.Dense(256, activation='elu'),
#   layers.Dropout(0.1),
    layers.Dense(128, activation='elu'),
    layers.Dense(3)])

model_augmented.compile(optimizer='adam', loss='mse')
early_stop_aug = callbacks.EarlyStopping(monitor='val_loss', patience=40,␣
 ↪restore_best_weights=True)

history_augmented = model_augmented.fit(X_train_aug, y_train_aug,
                                        validation_data=(X_val, y_val),
                                        epochs=200,
                                        batch_size=80,
                                        callbacks=[early_stop_aug],
                                        verbose=0)

# Plotting of augmented Neural Network training and validation loss
plt.figure(figsize=(10, 6))
plt.plot(history_augmented.history['loss'], label='Training Loss')
```

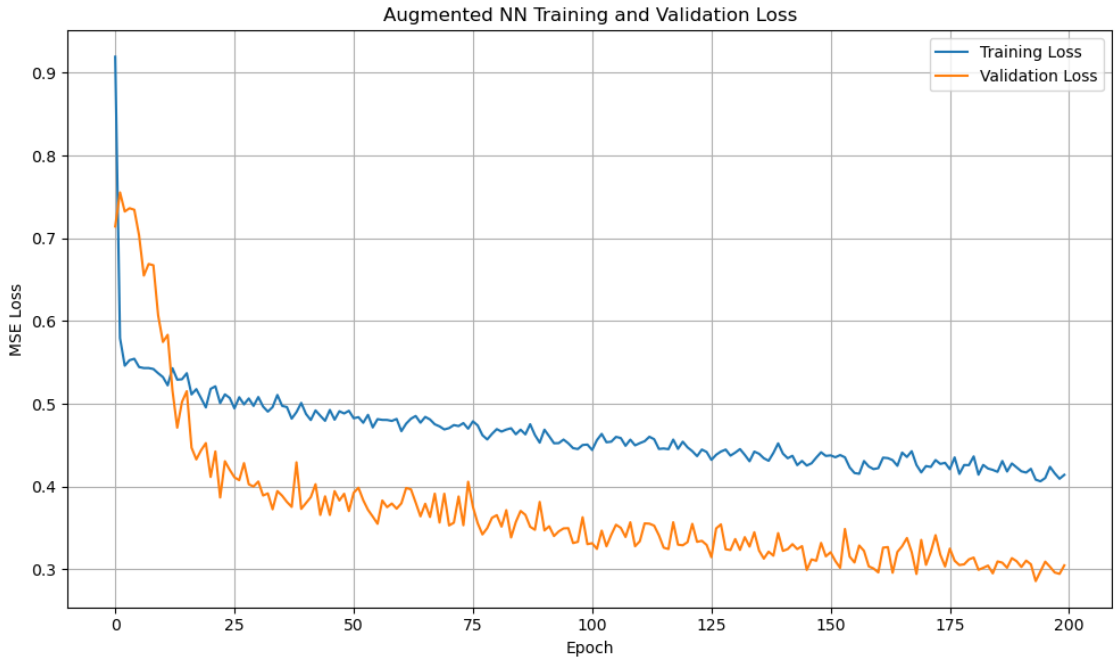
```
plt.plot(history_augmented.history['val_loss'], label='Validation Loss')
plt.xlabel("Epoch")
plt.ylabel("MSE Loss")
plt.title("Augmented NN Training and Validation Loss")
plt.legend()
plt.grid(True)
plt.tight_layout()
plt.show()

# Prepare the observed yield curve as the network input.
observed_yields = (observed_rates / 100)
observed_yields = observed_yields.reshape(1, -1)
observed_scaled = scaler_X.transform(observed_yields)

# Use the augmented neural network to predict log-parameters.
pred_scaled_aug = model_augmented.predict(observed_scaled)
pred_log_params_aug = scaler_y.inverse_transform(pred_scaled_aug)
pred_params_aug = np.exp(pred_log_params_aug)  # Exponentiate to retrieve␣
 ↪original parameter scale.
a_nn_aug, sigma_nn_aug, r0_nn_aug = pred_params_aug[0]

print("NN-Calibrated Hull-White parameters (Augmented):")
print(f" a = {a_nn_aug:.6f}")
print(f" sigma = {sigma_nn_aug:.6f}")
print(f" r0 = {r0_nn_aug:.6f}")

# Use the augmented calibrated parameters to compute bond prices and the␣
 ↪resulting yield curve.
model_prices_nn_aug = np.array([P_hw(0, T, r0_nn_aug, a_nn_aug, sigma_nn_aug)␣
 ↪for T in times])
yields_nn_aug = -np.log(model_prices_nn_aug) / times

# Plotting for the augmented Neural Network calibrated Hull-White model
plt.figure(figsize=(10, 6))
plt.plot(times, yields_nn_aug * 100, label="NN-Calibrated Hull-White Fit␣
 ↪(Augmented)", marker='o', color='green')
plt.plot(times, observed_rates, label="Market Data", linestyle='dashed',␣
 ↪marker='x', color='red')
plt.xlabel("Time to Maturity (Years)")
plt.ylabel("Yield (%)")
plt.title("Yield Curve: Augmented NN-Calibrated Hull-White vs Market Data")
plt.legend()
plt.grid(True)
plt.tight_layout()
plt.show()
```



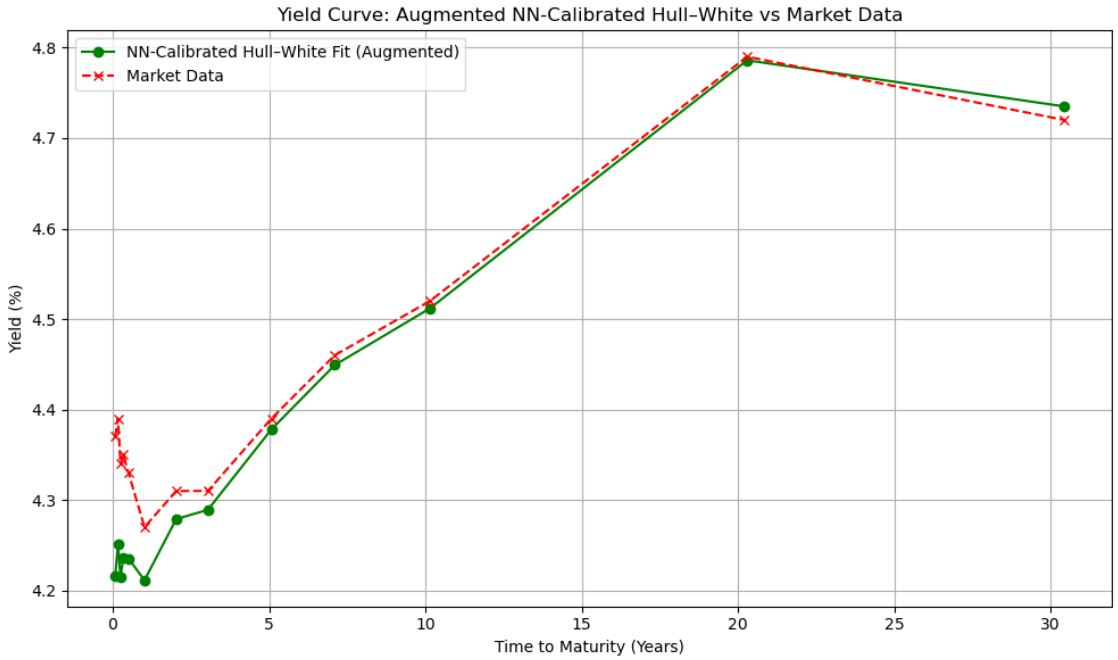

```
1/1             0s 69ms/step
NN-Calibrated Hull-White parameters (Augmented):
 a = 2.439728
 sigma = 0.229988
 r0 = 0.041757
```



```
Retained 4 components out of 13 features.
```




```
1/1             0s 70ms/step
NN-Calibrated Hull-White parameters (Augmented):
 a = 2.013613
 sigma = 0.250633
 r0 = 0.039572
```



0.5 Explanation of model and assumptions

Using the methods as described in Hernandez (2016) we were able to train a neural network to calibrate and fit the Hull-White model. In the first section of our code, we first prepared the treasury rate data and to ensured the time to maturities and rates were in a easier format to handle while coding. We chose our start date as 13/02/2025, we did this as this would allow us to compare our neural network model to our calibrated pricing model in the previous question. We then implemented the same Hull-White pricing model as in question 1 ii) and began to generate synthetic training data for our neural network. To train our neural network we needed to generate synthetic yield curves by sampling parameters, a, σ and r_0 in the Hull-White model and mapping them to their corresponding yield curves with the pricing model. Once we had the zero-coupon bond price $P(0,T)$ from the Hull-White formula, we converted it into a continuously compounded yield using,

$$y(T) = -\frac{1}{T} ln(P(0,T)) \tag{12}$$

To do this we used a uniform distribution to sample these parameters and limited them to specific ranges to ensure our neural network was being trained on realistic market conditions. For our mean reversion parameters we sampled $a \sim U[0.001-3.5]$ which allowed our neural network to cover both very slow and relatively fast mean reversion scenarios. For our volatility parameter we sampled $\sigma \sim U[0.0001-1]$, this was broad enough to capture both extremely low and fairly high levels of uncertainty in the movement of the treasury rates. Finally, we chose $r_0 \sim U[0.01-0.1]$ for our initial short-term rate r_0. We made sure that these ranges we sampled from were realistically plausible, ensuring we directly followed the sampling methodology as mentioned in Hernandez (2016). Before

training our feed-forward neural network model we applied some pre-processing techniques. We first performed a log transformation of the parameters (a, σ, r_0) to ensure positivity, ensuring to apply the exponential later on to retrieve the original parameters before plotting. We also used Z-score standardisation to both the yield vectors and the log-parameters to ensure all inputs and outputs had mean 0 and variance 1. After this we ran the data through our Feed-forward ELU neural network, this consisted of,

- an input layer of dimension N
- Two hidden layers of 256 units with ELU activation and Batch normalisation between them
- A 128 unit ELU activation layer
- Final dense layer with the output of three parameters $log(a), log(\sigma), log(r_0)$

We originally included dropout layers as a regularisation measure, but since our training set is noise-free and virtually unlimited, we found overfitting to be negligible and therefore removed dropout to streamline training. When using the dropout however we did experience a loss of data and found that it made our model less accurate. This was in line with what Hernandez (2016) found, as lower dropout rates yielded the best results. After trialling different activation units such as ReLU and LeakyReLU we found that ELU provided the most improvement and also found the Adam optimiser with the original learning rate of 0.001 to provide the best results for our model. This was consistent to the assumptions Hernandez (2016) made about his model, having the same activation unit and learning rate. We also found that fine-tuning our model to have a 25% validation split, an early stopping with a patience of 40, batch size of 80, and maximum epochs of 200 allowed us to see the best fit for our model.

After we managed to produce the Feed-forward Neural Network following the same methodology as outlined in Hernandez (2016), we further decided to try and implement Principl Component Analysis (PCA) as discussed in his paper. We found that we only needed to retain a number of 4 components to explain 99.9% of variance of the synthetic yields using PCA. Although PCA would improve the overall time our neural network would take to learn, we did find that there was notably higher training loss due to this and as such, the predicted yield curve was slightly worse. This calibrated model was validated by reconstructing the yield curve for comparison with the market data and plotted as shown above. From our Feed-forward Neural network we can that it was very effective in fitting the Hull-white model to a given yield curve further confirming that the method as outlined in Hernandez (2016) shows "good behaviour"

1 Question 2ii)

To evaluate the capabilities of our neural network trained on Hull-White data, we tested and compared its performance on data generated by the Cox-Ingersoll-Ross (CIR) model.

The CIR model defines the short rate as

$$dr_t = a(b - r_t)dt + \sigma\sqrt{r_t}dW_t \tag{13}$$

where W_t is a Wiener process. The price $P(t, T)$ of a zero-coupon bond under this model at time t maturing at time T is given by

$$P(t, T) = A(T - t) \cdot e^{-B(T-t)r_t} \tag{14}$$

where $A(\tau)$ and $B(\tau)$ are deterministic functions of time to maturity $\tau = (T - t)$ and,

$$h = \sqrt{a^2 + 2\sigma^2} \tag{15}$$

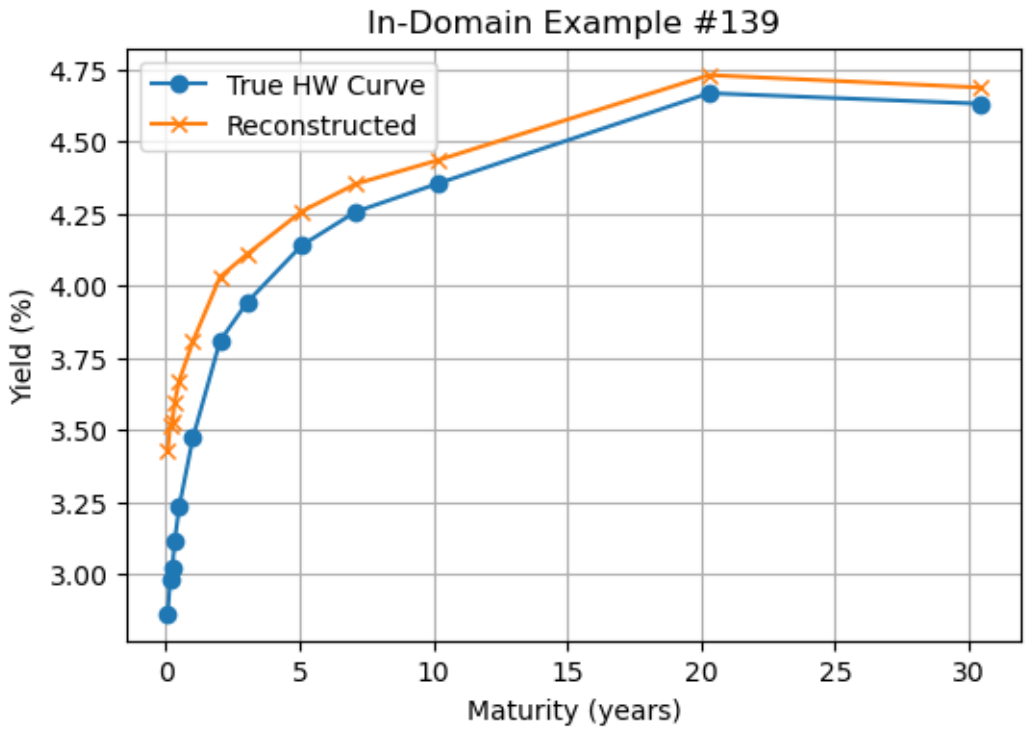
$$A(\tau) = \left[\frac{2h \cdot (e^{(a+h)\tau/2})}{2h + (a+h)(e^{h\tau} - 1)}\right]^{\frac{2ab}{\sigma^2}} \tag{16}$$

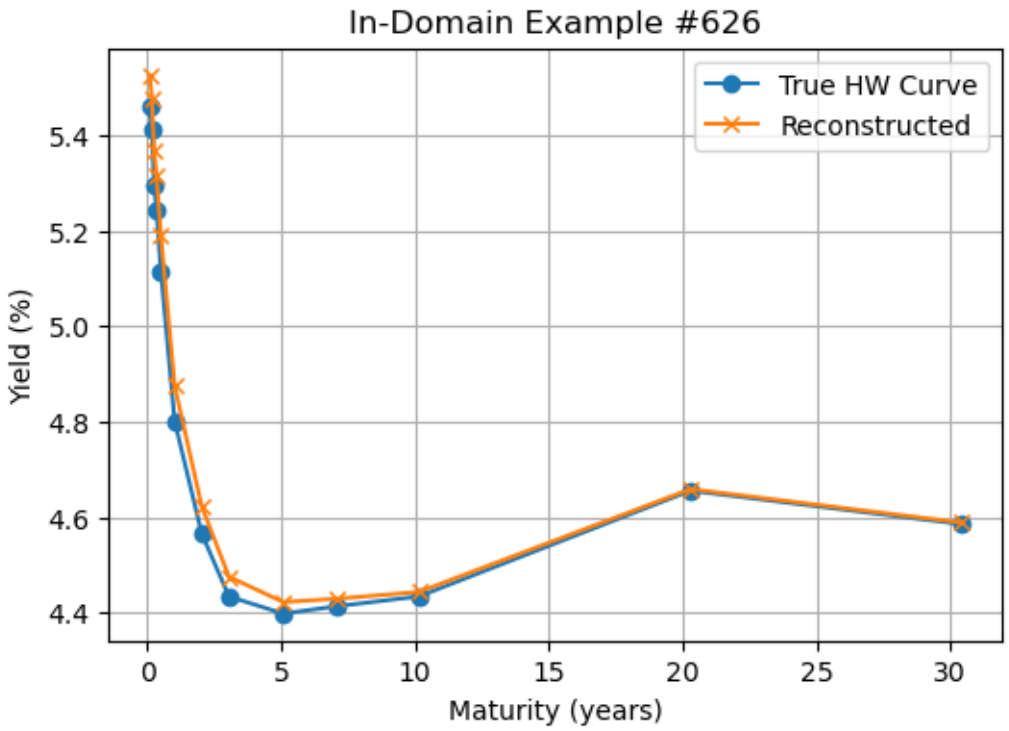
$$B(\tau) = \frac{2(e^{h\tau} - 1)}{2h + (a+h)(e^{h\tau} - 1)} \tag{17}$$

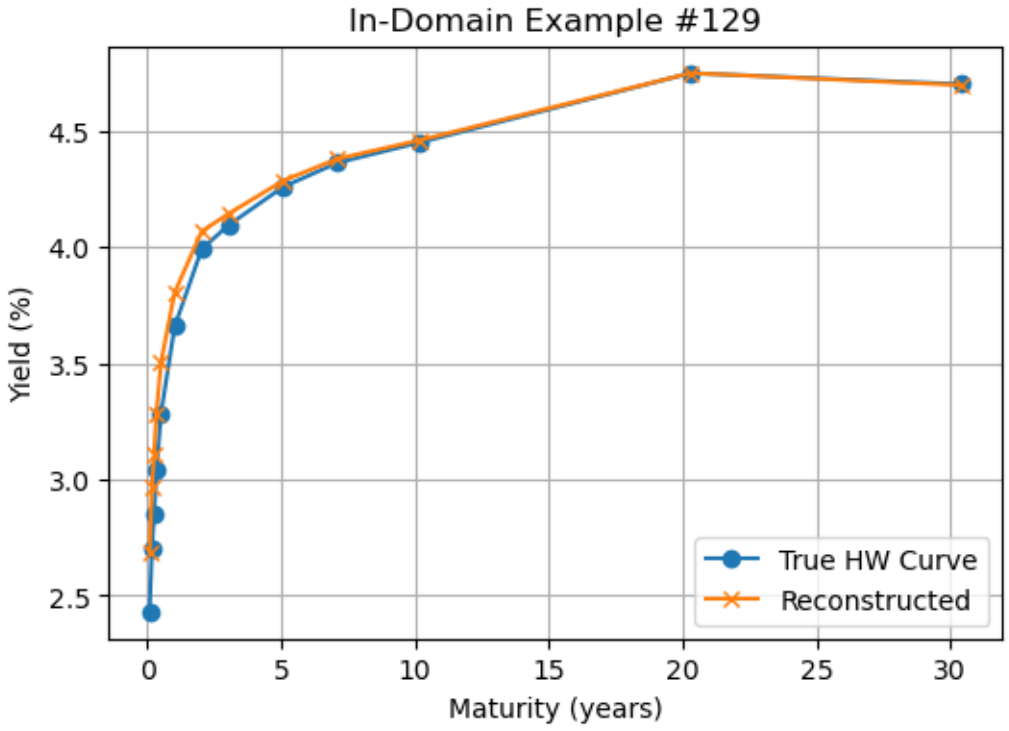
[35]:
```
# In-Domain evaluation
def evaluate_in_domain(model, scaler_X, scaler_y, generator, times,␣
 ↪n_samples=1000):

    # Sample true parameters
    a_test = np.random.uniform(0.001, 3.5,   n_samples)
    sigma_test = np.random.uniform(0.0001, 1.0,  n_samples)
    r0_test = np.random.uniform(0.001, 0.1,   n_samples)
    params_test = np.vstack([a_test, sigma_test, r0_test]).T

    # Build true curves
    X_test = generator(params_test)

    # Scale and predict
    Xs = scaler_X.transform(X_test)
    pred_scaled = model.predict(Xs)
    pred_log = scaler_y.inverse_transform(pred_scaled)
    pred_params = np.exp(pred_log)

    # RMSE of parameter recovery
    param_rmse = np.sqrt(np.mean((pred_params - params_test)**2, axis=0))

    # Reconstruct curves from predicted parameters
    curves_hat = generator(pred_params)

    # RMSE of the curve
    curve_rmse = np.sqrt(np.mean((curves_hat - X_test)**2))
    return param_rmse, curve_rmse, X_test, curves_hat

# Run In-Domain check
param_rmse, curve_rmse, X_true, X_hat = evaluate_in_domain(model_baseline,␣
 ↪scaler_X, scaler_y, compute_yields, times, n_samples=1000)
print("In-Domain parameter RMSE (a, sigma, r0):", param_rmse)
print("In-Domain curve RMSE:", curve_rmse)

# Plotting of random yield curves for visual interpretation
for i in np.random.choice(len(X_true), size=3, replace=False):
    plt.figure(figsize=(6,4))
    plt.plot(times, X_true[i]*100, label="True HW Curve", marker='o')
    plt.plot(times, X_hat[i]*100, label="Reconstructed", marker='x')
    plt.xlabel("Maturity (years)")
```

```
    plt.ylabel("Yield (%)")
    plt.title(f"In-Domain Example #{i}")
    plt.legend()
    plt.grid(True)
    plt.show()
```

```
32/32              0s 2ms/step
In-Domain parameter RMSE (a, sigma, r0): [0.36802301 0.18156418 0.00505129]
In-Domain curve RMSE: 0.019200816869749098
```



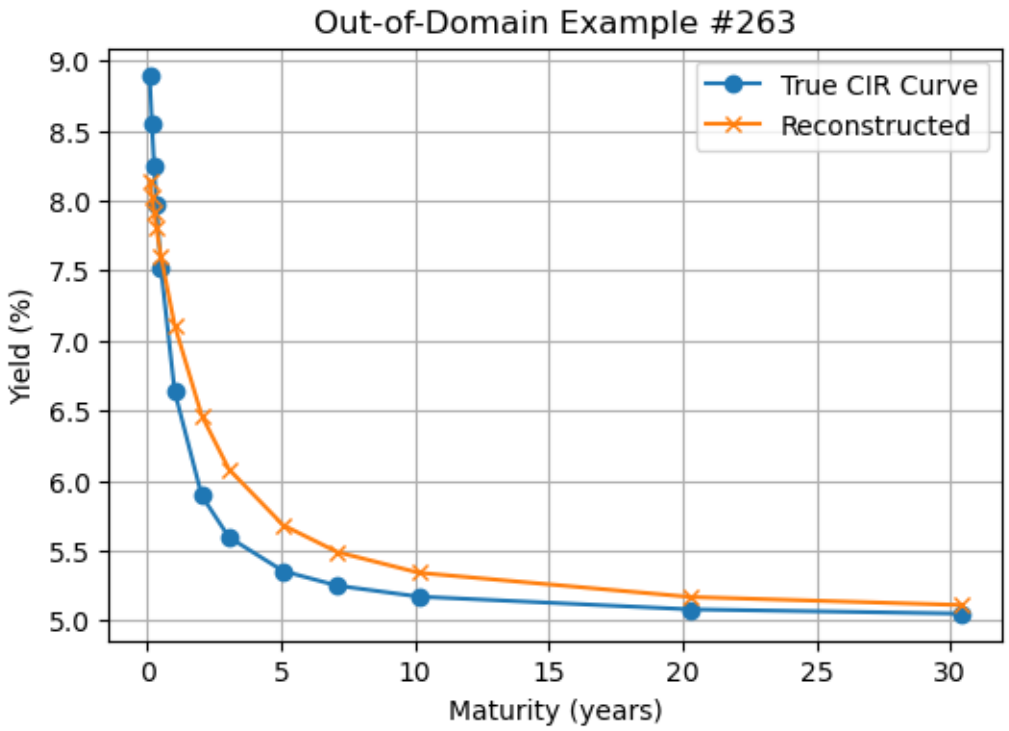

In-Domain Example #626
True HW Curve
Reconstructed
Yield (%)
Maturity (years)
5.4
5.2
5.0
4.8
4.6
4.4
0
5
10
15
20
25
30

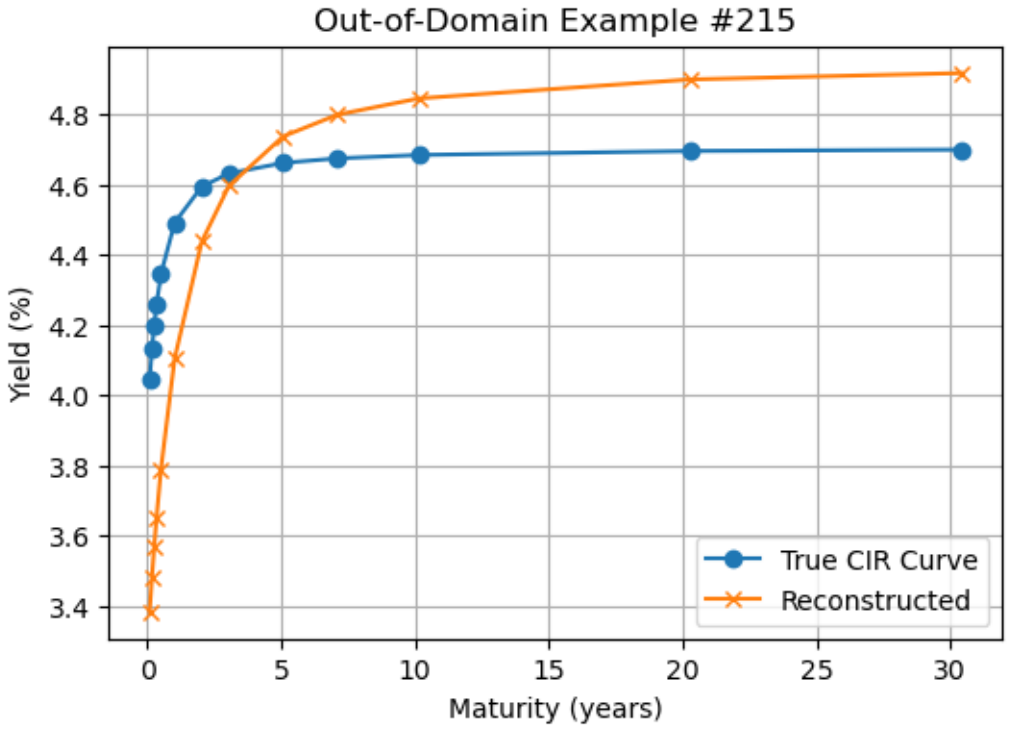

In-Domain Example #129
True HW Curve
Reconstructed
Yield (%)
Maturity (years)
4.5
4.0
3.5
3.0
2.5
0
5
10
15
20
25
30

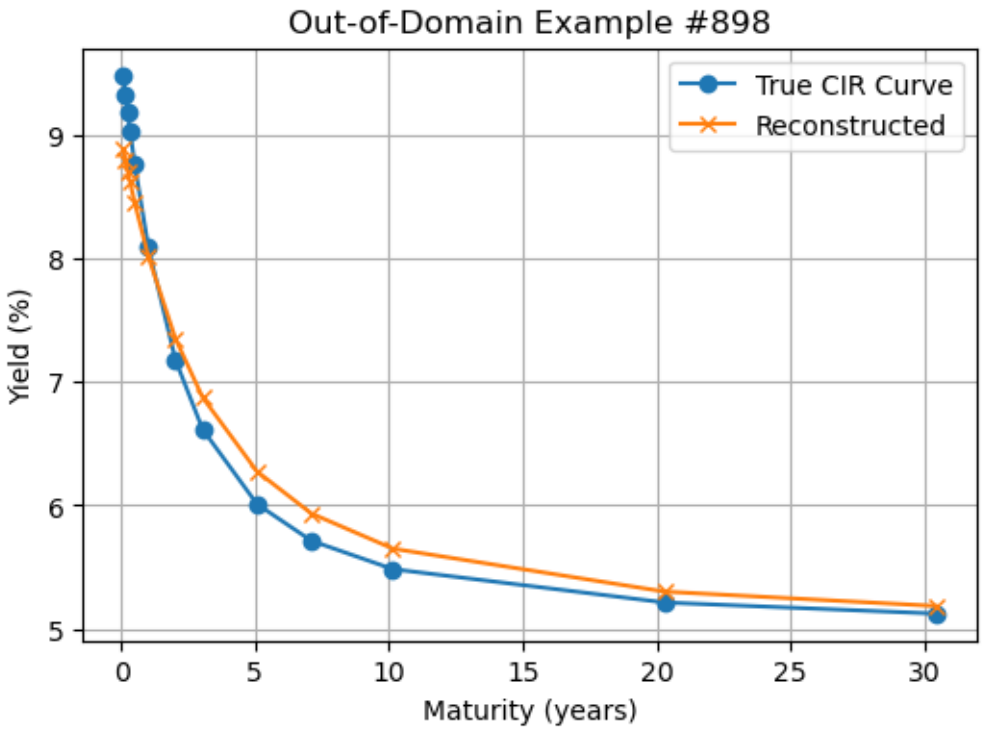
[36]:
```
# Define the Cox-Ingersoll-Ross (CIR) Model
def B_CIR(tau, a, sigma):
    h = np.sqrt(a**2 + 2 * sigma**2)
    num = 2 * (np.exp(h * tau) - 1)
    denom = (a + h) * (np.exp(h * tau) - 1) + 2 * h
    return num / denom

def A_CIR(tau, a, b, sigma):
    h = np.sqrt(a**2 + 2 * sigma**2)
    num = 2 * h * np.exp((a + h) * tau / 2)
    denom = 2 * h + (a + h) * (np.exp(h * tau) - 1)
    exp = (2 * a * b) / sigma**2
    return (num / denom) ** exp

def P_CIR(tau, r0, a, b, sigma):
    A = A_CIR(tau, a, b, sigma)
    B = B_CIR(tau, a, sigma)
    return A * np.exp(-B * r0)

# CIR generator with fixed b = 0.05
def CIR_generator(params):
    b = 0.05
    curves = []
    for a, sigma, r0 in params:
        prices = np.array([P_CIR(t, r0, a, b, sigma) for t in times])
        yields = -np.log(prices) / times
        curves.append(yields)
    return np.array(curves)

# Out-Of-Domain evaluation using the CIR model
def evaluate_out_of_domain(model, scaler_X, scaler_y, cir_generator, times,␣
 ↪n_samples=1000):

    # Sample true parameters
    a_test     = np.random.uniform(0.001, 3.5,   n_samples)
    sigma_test = np.random.uniform(0.0001, 1.0,  n_samples)
    r0_test    = np.random.uniform(0.001, 0.1,   n_samples)
    params_test = np.vstack([a_test, sigma_test, r0_test]).T

    # Build true curves using CIR model
    X_test = CIR_generator(params_test)

    # Scale and predict
    Xs = scaler_X.transform(X_test)
    pred_scaled = model.predict(Xs)
    pred_log    = scaler_y.inverse_transform(pred_scaled)
    pred_params = np.exp(pred_log)
```

```
    # RMSE of the parameter recovery and the curve
    param_rmse = np.sqrt(np.mean((pred_params - params_test)**2, axis=0))
    curves_hat = CIR_generator(pred_params)
    curve_rmse = np.sqrt(np.mean((curves_hat - X_test)**2))
    return param_rmse, curve_rmse, X_test, curves_hat

# Run Out-Of-Domain check
param_rmse, curve_rmse, X_true, X_hat = evaluate_out_of_domain(model_baseline,␣
 ↪scaler_X, scaler_y, CIR_generator, times, n_samples=1000)
print("Out-Of-Domain parameter RMSE (a, sigma, r0):", param_rmse)
print("Out-Of-Domain curve RMSE:", curve_rmse)

# Plotting of some random yield curves for visual interpretation
for i in np.random.choice(len(X_true), size=3, replace=False):
    plt.figure(figsize=(6,4))
    plt.plot(times, X_true[i]*100, label="True CIR Curve", marker='o')
    plt.plot(times, X_hat[i]*100,  label="Reconstructed",    marker='x')
    plt.xlabel("Maturity (years)")
    plt.ylabel("Yield (%)")
    plt.title(f"Out-of-Domain Example #{i}")
    plt.legend()
    plt.grid(True)
    plt.show()
```

```
32/32             0s 1ms/step
Out-Of-Domain parameter RMSE (a, sigma, r0): [1.24400082 0.42102426 0.00774505]
Out-Of-Domain curve RMSE: 0.00576288334348822
```


Out-of-Domain Example #263
True CIR Curve
Reconstructed
Yield (%)
Maturity (years)
9.0
8.5
8.0
7.5
7.0
6.5
6.0
5.5
5.0
0
5
10
15
20
25
30


Out-of-Domain Example #215
True CIR Curve
Reconstructed
Yield (%)
Maturity (years)
4.8
4.6
4.4
4.2
4.0
3.8
3.6
3.4
0
5
10
15
20
25
30



1.0.1 Performance of model

When tested on Hull-White data (In-Domain), the Neural Network performed very well achieving a low Root Mean Squared Error (RMSE) in both recovery of parameters and reconstruction of the yield curve. This confirmed the accuracy of our network's ability to learn the mapping between curve shapes and parameters when the data-generating process matched the training model. The key strength of this approach lies in its speed and accuracy, once trained the network calibrates almost instantaneously. However, this method still requires a well-designed training set and careful data pre-processing.

In contrast, when we evaluated our Neural Network on CIR data (Out-Of-Domain), parameter recovery deteriorated significantly, with much higher RMSE values. This indicated that our network struggled to infer the correct parameters when the data originated from a model with fundamentally different dynamics. The CIR model differs from Hull-White in that it requires a strictly positive short rate r_0, exhibits non-Gaussian dynamics, and shows different sensitivities to parameters such as volatility (σ). While our network generalises well in terms of curve shape, it lacks robustness in interpreting parameters when our data-generating process differs. Overall, we found that this method was well-suited for fast and flexible curve fitting but produced unreliable parameter estimates, especially for a and σ, due to the model mismatch. Below we have outlined the strength and weakness of using this Out-Of-Domain approach,

1.0.2 -Strengths of using Out-Of-Domain data

- The Neural network was able to generalise to unseen yield curves from different models providing potentially more use-cases.
- The Neural network was able to maintain a low yield curve reconstruction RMSE, making it somewhat reliable to various types of data.
- The Neural network is suitable for fast and flexible yield curve fitting, making it preferable when speed is favoured over parameter interpretability.

1.0.3 -Weaknesses of using Out-Of-Domain data

- The Neural network faces poor parameter recover when the data-generating process differs from the training process
- There is a risk of parameter inferences being misleading and therefore the recovery could be potentially considered useless.

In conclusion we found that our Neural Network trained upon the Hull-White model can approximate Cox-Ingersoll-Ross yield curves fairly effectively but can leads to compromises in parameter interpretability.