ECB Project - Coding Appendix

Contents

Preliminaries

Setup

```r
# --------- i. Clear environment

rm(list=ls())

# --------- ii. Load here package to enable finding script loading other packages

# First, define a function that checks if a package is installed.
# If not, it installs it. Then it loads it.
package_loader <- function(pkg) {
  if (!require(pkg, character.only = TRUE)) {
    message(paste("Installing missing package:", pkg))
    install.packages(pkg, dependencies = TRUE)
    library(pkg, character.only = TRUE)
  }
}
package_loader("here")

# --------- iii. Load packages
source(here("scripts/00a_packages.R"))

# --------- iv. Load helper functions ONLY for plots and tables
# Note: These will NOT be explained in the companion paper
source(here("helpers/raw_plotter.R"))
source(here("helpers/stationarity_tables.R"))
source(here("helpers/taylor_tables.R"))
source(here("helpers/struct_breaks_tables.R"))
source(here("helpers/roll_TR_plotter.R"))
source(here("helpers/pseudo_outofsample_tables.R"))
source(here("helpers/pseudo_outofsample_plots.R"))
source(here("helpers/actual_forecast_displays.R"))

# --------- v. Load API keys for data (only FRED required)
fredr_set_key(Sys.getenv("FRED_API_KEY"))

# --------- vi. Dates (to automatically get the latest data from API calls)
start_date <- "1999-01-01"
end_date <- Sys.Date()
```

Interactive Option Selection

- Use Hamilton Filter:
  - TRUE: Selects Hamilton method for output gap estimation
  - FALSE: Selects Hodrick-Prescott method for output gap estimation
- Use Inflation Expectations:
  - TRUE: The models used for forecasting will use 12-month ahead inflation expectations from the ECB survey of professional forecasts (average).

  - FALSE: The models used for forecasting will use realised inflation

- Use Formula
  - Formula 1: Actual interest rate regressed on inflation and output gaps

  - Formula 2: Shadow interest rate regressed on inflation and output gaps
  - Formula 3: Actual interest rate regressed on the one-quarter lag of the interest rate and on inflation and output gaps
  - Formula 4: Shadow interest rate regressed on the one-quarter lag of the shadow interest rate and on inflation and output gaps

- Format:
  - html: For printing tables in html (to view in console)
  - latex: For saving and knitting tables to LaTeX and pdf

- Save figures:
  - TRUE: All figures (plots and tables) are saved to the figures/ folder
  - FALSE: No figures are saved, useful for running the code in console

```r
# --------- 1. Related to Analysis ---------

USE_HAMILTON_FILTER <- TRUE
USE_INFLATION_EXPECTATIONS <- FALSE
USE_FORMULA <- "Formula 3"

# --------- 2. Related to Document Output ---------

format <- "html"
format <- "latex"
save_figures <- TRUE

#and add window R choice and horizon H choice removed from pseudo estimation
```

```r
# Note: All 4 formulas are used in the pseudo out-of-sample estimation exercise
#        as well as in the absolute and relative performance evaluation.
#       However, for the rest of the code, specifically spaghetti plots, errors
#        evaluation, and actual future forecast, we only use 1 formula which is
#         determined in the "options_config.R" script.
#       These are also NOT used in the initial taylor rule estimation exercise
#        on the full sample, as there we want to really view the various options
#         and combinations regardless of the selected options config.

# --------- 1. Specifications ---------
# (using either current inflation or inflation expectations
#  according to configuration, same with HP vs Hamilton)
formula_1 <- rate ~ inflation_gap + output_gap
formula_2 <- shadowrate ~ inflation_gap + output_gap
formula_3 <- rate ~ rate_lag + inflation_gap + output_gap
formula_4 <- shadowrate ~ shadowrate_lag + inflation_gap + output_gap

# --------- 2. Select the main model based on select initial option ---------
# Again, used in spaghetti plots, errors evaluation, and actual future forecast
if (USE_FORMULA == "Formula 1") {
  model <- "F_TR_FORMULA_1"
  model_formula <- formula_1
  model_name <- "Taylor Rule Formula 1"
} else if (USE_FORMULA == "Formula 2") {
  model <- "F_TR_FORMULA_2"
  model_formula <- formula_2
  model_name <- "Taylor Rule Formula 2"
} else if (USE_FORMULA == "Formula 3") {
  model <- "F_TR_FORMULA_3"
  model_formula <- formula_3
  model_name <- "Taylor Rule Formula 3"
} else if (USE_FORMULA == "Formula 4") {
  model <- "F_TR_FORMULA_4"
  model_formula <- formula_4
  model_name <- "Taylor Rule Formula 4" }
```

Data

Sources & Explanations

- FRED Data Source: European Central Bank, ECB Deposit Facility Rate for Euro Area [ECBDFR], retrieved from FRED, Federal Reserve Bank of St. Louis; https://fred.stlouisfed.org/series/ECBDFR. The data is the Deposit Facility and is directly reprinted from the ECB. It's in percent and not seasonally adjusted. The ECB Monetary Policy is steered through this rate.

- Shadow Interest Rate: The Shadow rate was developed by **wuTimeVaryingLowerBound2017** and does quantify a hypothetical removal of the ZLB. The rate does not track 1:1 on the deposit facility in the data before the ZLB, instead being closer to the refinancing rate. (maybe need to adjust, oui je crois faux, surtout la partie sur refinancing rate)

- GDP: The GDP Data is quarterly real GDP in 2010 Euros in million retrieved from FRED via Eurostat. The data is seasonally adjusted.

- Potential GDP: Either estimated with the Hodrick-Prescott filter or the Hamilton filter, based on the formulas described in equation (1):

$$\begin{aligned} \text{HP Filter:} \quad & \min_{\tau} \left(\sum_{t=1}^{T} (y_t - \tau_t)^2 + \lambda \sum_{t=2}^{T-1} \left[(\tau_{t+1} - \tau_t) - (\tau_t - \tau_{t-1}) \right]^2 \right) \\ \text{Hamilton:} \quad & y_t = \beta_0 + \sum_{j=1}^{p} \beta_j y_{t-h-j+1} + v_t \quad (\text{where } v_t \text{ is the cycle}) \end{aligned} \tag{1}$$

- Inflation:
  - Realised: The Inflation data is from Eurostat and measures HICP monthly data (annual rate of change). It's the index that the ECB uses for Inflation and is not seasonally adjusted, but the fact that it represents the year-on-year change in prices implies there is no seasonality.
  - Expectations: Expected Inflation is the ECB's survey of professional forecasters. It tracks professional forecasts of HICP inflation 12 months in advance.

Loading & Preparation Data

To get this data, we run the following code using API calls to FRED, EuroStat, and DBnomics. We convert each variable to the quarterly average and then store them all to one dataframe named "data". It includes a date column, the deposit rate, the shadow rate, inflation, inflation gap (which is just inflation minus the ECB target, which is set at 2%), expected inflation, expected inflation gap, real GDP, GDP output gap computed with the HP filter and GDP output gap computed with the Hamilton filter.

```r
# --------- 1. ECB Deposit Facility Rate & Shadow Rate ---------
ecb_rate_daily <- fredr(
                    series_id = "ECBDFR",
                    observation_start = as.Date(start_date))
ecb_rate_q <- ecb_rate_daily %>%
  mutate(quarter = as.yearqtr(date)) %>%
  group_by(quarter) %>%
  summarise(rate = last(value)) %>%
  mutate(date = as.Date(quarter))

# Wu-Xia Shadow Rate
```

```r
shadow_rate_daily = as.data.frame(readMat("data/shadowrate_ECB.mat"))
colnames(shadow_rate_daily) <- c("DATE", "shadowrate")
shadow_rate_daily$DATE <- as.Date(paste0(shadow_rate_daily$DATE, "01"),
                                  format="%Y%m%d")
shadow_rate_daily$quarter <- as.yearqtr(as.Date(shadow_rate_daily$DATE))
shadow_rate_daily$month <- as.yearmon(as.Date(shadow_rate_daily$DATE))
quarterly_shadow = aggregate(shadowrate ~ quarter,
                             data=shadow_rate_daily,
                             FUN=mean,
                             na.rm=T)
monthly_shadow = aggregate(shadowrate ~ month,
                           data=shadow_rate_daily,
                           FUN=mean, na.rm=T)


# --------- 2. HICP Inflation (Euro Area) ---------
inflation_data <- get_eurostat(
                    "prc_hicp_manr",
                    filters = list(geo = "EA", coicop = "CP00"),
                    type = "label")
inflation_q <- inflation_data %>%
  filter(time >= start_date) %>%
  dplyr::select(date = time, inflation = values) %>%
  mutate(quarter = as.yearqtr(date)) %>%
  group_by(quarter) %>%
  summarise(inflation = mean(inflation, na.rm = TRUE)) %>%
  mutate(date = as.Date(quarter))

# Inflation expectations
inflation_exp <- rdb(ids = "ECB/SPF/M.U2.HICP.POINT.P12M.Q.AVG")
# Inflation_exp <- rdb(ids = "ECB/SPF/M.U2.HICP.POINT.P24M.Q.AVG")
inflation_exp_q <- inflation_exp %>%
  mutate(quarter = as.yearqtr(period)) %>%
  group_by(quarter) %>%
  summarise(exp_inflation = last(original_value)) %>%
  mutate(date = as.Date(quarter))

#P12M : 12-month ahead forecasts
inflation_q$exp_inflation = c(rep(NA,3),as.numeric(inflation_exp_q$exp_inflation),NA)
#P24M : 24-month ahead forecasts
#inflation_q$exp_inflation = c(rep(NA,7),as.numeric(inflation_exp_q$exp_inflation[1:101]))

# --------- 3. Real GDP and Estimated Output Gap ---------
# a) Real GDP for the Euro Area. The series ID is CLVMNACSCAB1GQE_A.
gdp_q <- fredr(
  series_id = "CLVMEURSCAB1GQEA19",
  observation_start = as.Date(start_date)) %>%
  mutate(quarter = as.yearqtr(date)) %>%
  dplyr::select(quarter, real_gdp = value) %>%
  mutate(log_real_gdp = log(real_gdp))

# b) Estimate Potential GDP (the trend) using the HP Filter on the log of real GDP.
# The lambda value of 1600 is standard for quarterly data.
```

```r
hp_gdp <- hpfilter(gdp_q$log_real_gdp, freq = 1600)
gdp_q$potential_gdp_log <- as.numeric(hp_gdp$trend)
ham_gdp_cycle <- filter_hamilton(gdp_q$log_real_gdp, p = 4, horizon = 8)
gdp_q$potential_gdp_log_ham <- gdp_q$log_real_gdp - ham_gdp_cycle


# --------- 4. Combine all data into a single data frame ---------
data <- ecb_rate_q %>%
  dplyr::select(quarter, rate) %>%
  left_join(inflation_q, by = "quarter") %>%
  left_join(gdp_q, by = "quarter") %>%
  left_join(quarterly_shadow, by = "quarter")

# --------- 5. Create new variables ---------
data <- data %>%
  mutate(
    realised_inflation_gap = inflation - 2.0,
    exp_inflation_gap = exp_inflation -2.0,
    output_gap_hp = 100 * (log_real_gdp - potential_gdp_log),
    output_gap_ham = 100 * (log_real_gdp - potential_gdp_log_ham),
    rate_lag = lag(rate, 1),
    shadowrate = case_when(
      quarter < "2012 Q3" | quarter >= "2022 Q3" ~ rate,
      TRUE ~ shadowrate),
    shadowrate_lag = lag(shadowrate, 1))

# Remove last row since no output data
data = subset(data, quarter < "2025 Q4")

# Clean environment
rm(gdp_q, hp_gdp, ecb_rate_daily, ecb_rate_q, inflation_data, inflation_q,
   inflation_exp, inflation_exp_q, monthly_shadow, quarterly_shadow, shadow_rate_daily)
```

Options Configuration

We then implement the chosen options on using inflation expectations or not, and on using either the HP filter or the Hamilton filter for output gap.

```r
# --------- 1. Filter selection for output gap estimation ----------

# TRUE  = Use Hamilton Filter (newer, arguably more robust)
# FALSE = Use HP Filter (classic approach)

# Applying selection
if (USE_HAMILTON_FILTER) {
  data$output_gap <- data$output_gap_ham
  cat("* CONFIGURATION: Using Hamilton Filter for output gap estimation.\n")
} else {
  data$output_gap <- data$output_gap_hp
  cat("* CONFIGURATION: Using HP Filter for output gap estimation.\n") }
```

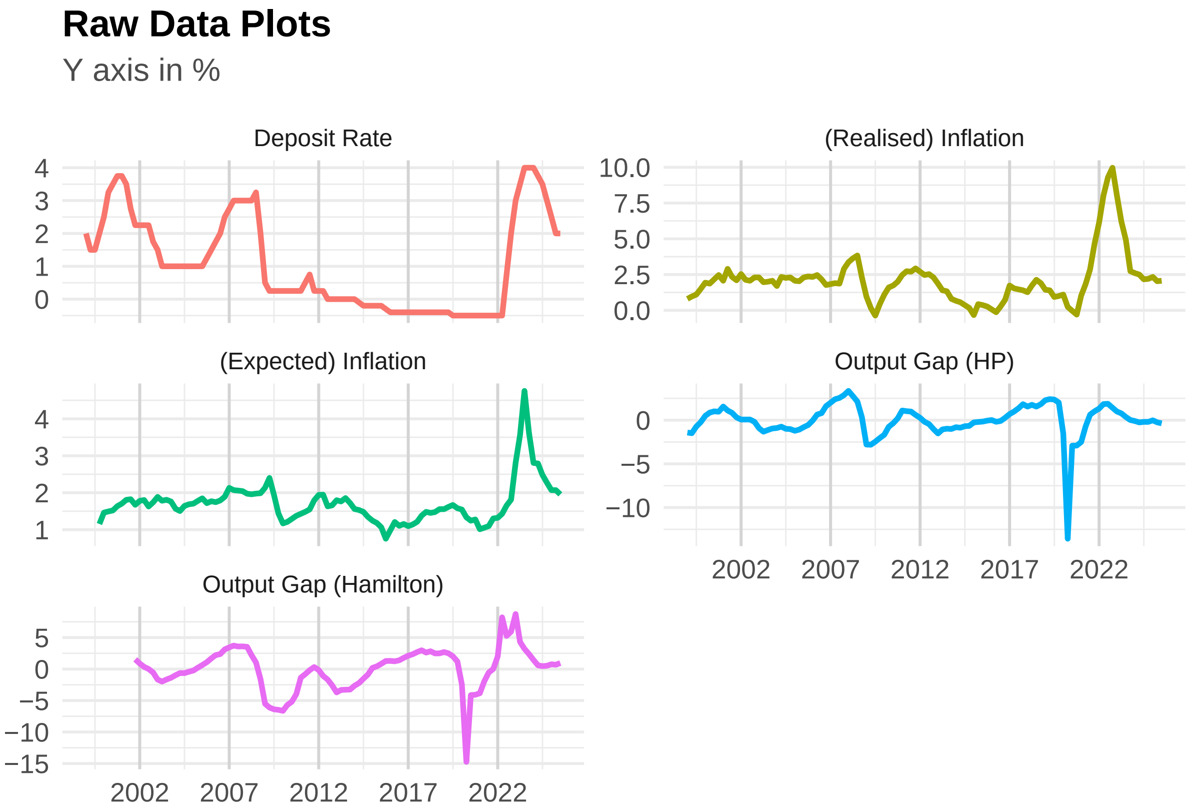
- CONFIGURATION: Using Hamilton Filter for output gap estimation.

```r
# --------- 2. Inflation expectations choice ----------

# TRUE  = Use inflation expectations from ECB survey of professional forecasts
# FALSE = Use realised inflation

# Applying selection
if (USE_INFLATION_EXPECTATIONS) {
  data$inflation_gap <- data$exp_inflation_gap
  cat("* CONFIGURATION: Using inflation expectations in Taylor Rule forecasting.\n")
} else {
  data$inflation_gap <- data$realised_inflation_gap
  cat("* CONFIGURATION: Using realised inflation in Taylor Rule forecasting.\n")
}
```

- CONFIGURATION: Using realised inflation in Taylor Rule forecasting.

Raw Data Plots

```r
raw_data_plotter(data = data)
```



Data Properties

In this section, we run various tests to understand the properties of our main variables, interest rate, inflation and output gap. These include both ADF and KPSS stationarity tests (using the tseries package) and a cointegration test (using the aTSA package) to see if the interest rate is cointegrated with inflation. We wrap these in custom functions that automatically report the results and intepret them based on the relevant p-value.

```r
# Helper function to run ADF and KPSS tests and return dataframe fit for table

check_stationarity <- function(y, var_name) {

  # Running tests and getting pvalues
  adf_result <- tseries::adf.test(y, alternative = "stationary")
  kpss_result <- tseries::kpss.test(y, null = "Level")
  adf_p <- adf_result$p.value
  kpss_p <- kpss_result$p.value

  # Automatically assign interpretation (yes, modular!)
  interpretation <- ""
  if (adf_p < 0.05 && kpss_p > 0.05) {
    interpretation <- "Stationary I(0): ADF rejects unit root, KPSS confirms stationarity."
  } else if (adf_p > 0.05 && kpss_p < 0.05) {
    interpretation <- "Non-Stationary I(1): ADF confirms unit root, KPSS rejects stationarity."
  } else if (adf_p > 0.05 && kpss_p > 0.05) {
    interpretation <- "Conflicting Results: ADF confirms unit root, while KPSS suggests stationarity."
  } else { # adf_p < 0.05 && kpss_p < 0.05
    interpretation <- "Conflicting Results: ADF rejects unit root, while KPSS rejects stationarity."
  }

  # Export results
  data.frame(Variable = var_name,
             `ADF p-value` = adf_p,
             `KPSS p-value` = kpss_p,
             Result = interpretation, check.names = FALSE)
}
```

```r
# Similar helper function, but for cointegration tests
check_coint <- function(y1,y2, var_name1, var_name2){

  # Run test and get pvalue
  coint_test <- aTSA::coint.test(y1, y2, output = FALSE)
  p_value <- coint_test[1, 3] #type 1

  # Again, modular(!) interpretation
  interpretation <- ""
  if (p_value < 0.05) {
    interpretation <- "Cointegrated"}
  else {
    interpretation <- "Not Cointegrated" }

  # Export results
  data.frame(Variables = paste(var_name1, "&", var_name2),
             `p-value` = p_value,
```

Table 1: Summary of Stationarity Tests (ADF & KPSS)

Variable	ADF p-value	KPSS p-value	Result
Interest Rate	0.4350	0.0359	Non-Stationary I(1): ADF confirms unit root, KPSS rejects stationarity.
Inflation	0.2257	0.1000	Conflicting Results: ADF confirms unit root, while KPSS suggests stationarity.
Output Gap	0.0131	0.1000	Stationary I(0): ADF rejects unit root, KPSS confirms stationarity.
Interest Rate (1st Diff)	0.0100	0.1000	Stationary I(0): ADF rejects unit root, KPSS confirms stationarity.
Inflation (1st Diff)	0.0100	0.1000	Stationary I(0): ADF rejects unit root, KPSS confirms stationarity.

```
             Result = interpretation, check.names = FALSE)
}
```

```
# --------- 1. Run Tests ---------

# Use helper to run stationarity checks for our main variables
test_rate <- check_stationarity(data$rate, "Interest Rate")
test_inflation <- check_stationarity(data$inflation, "Inflation")
test_output_gap <- check_stationarity(na.omit(data$output_gap), "Output Gap")

# Given results for rate and inflation, run tests on 1st diffs
test_rate_diff <- check_stationarity(diff(data$rate), "Interest Rate (1st Diff)")
test_inflation_diff <- check_stationarity(diff(data$inflation), "Inflation (1st Diff)")

# Since rate and inflation are I(1), run a cointegration test
coint_test = check_coint(data$rate, data$inflation,
                         var_name1 = "Interest Rate", var_name2 = "Inflation")


# --------- 2. Print Results ---------

# Combine stationarity results
all_stationarity_results <- rbind(test_rate,
                                  test_inflation,
                                  test_output_gap,
                                  test_rate_diff,
                                  test_inflation_diff)

# Tables
generate_stat_tests_table(stat_tests = all_stationarity_results)
```

```
generate_coint_tests_table(coint_test = coint_test)
```

Table 2: Cointegration Test Results

Variables	p-value	Result
Interest Rate & Inflation	0.0585	Not Cointegrated

```
# Cleanup
rm(all_stationarity_results, coint_test, test_inflation,
   test_inflation_diff, test_output_gap, test_rate, test_rate_diff)
```

Taylor Rule Estimation

We now run basic Taylor Rule estimations using OLS on the complete dataset in order to get some information on the historical Taylor coefficients.

Without Lags

The “No Lag” Taylor Rule specifications follow the regression expressed in equation (2). We run 6 variations total of this, starting with the base case using the actual deposit rate, realised inflation (gap) and the output gap. We then mix cases where we use the shadow rate, expected inflation (gap) and finally including both expected and realised inflation (gap).

$$i_t = \pi^* + \beta(\pi_t - \pi^*) + \gamma(y_t - \bar{y_t}) \tag{2}$$

```r
# --------- 1. Simple Taylor Rule without lag ---------

# Fit models
TR <- lm(rate ~ realised_inflation_gap + output_gap, data = data)
TRsr <- lm(shadowrate ~ realised_inflation_gap + output_gap, data = data)

# List them and print them as tables using our helper function
models <- list(TR, TRsr)
taylor_regression_to_table(models, caption = "No Lag, No Expectations")
```

Table 3: No Lag, No Expectations

	TR	TR w/ SR
(Intercept)	0.8460	-0.8295
	(0.8940)	(2.6541)
realised_inflation_gap	0.1944	0.6807
	(0.4320)	(0.6057)
output_gap	0.0840	-0.0524
	(0.1633)	(0.2764)
N	96	96
R2	0.1723	0.1146

*** p < 0.001; ** p < 0.01; * p < 0.05.

```r
# --------- 2. Taylor Rule without lag but with inflation expectations  ---------

# Fit models
TR_e <- lm(rate ~ exp_inflation_gap + output_gap, data = data)
TRsr_e <- lm(shadowrate ~ exp_inflation_gap + output_gap, data = data)
TR_ie <- lm(rate ~ realised_inflation_gap + exp_inflation_gap +
              output_gap, data = data)
TRsr_ie <- lm(shadowrate ~ realised_inflation_gap + exp_inflation_gap +
                output_gap, data = data)

# List them and print them as tables using our helper function
models <- list(TR_e, TRsr_e, TR_ie, TRsr_ie)
taylor_regression_to_table(models, caption = "No Lag, with
                                              Inflation Expectations")
```

Table 4: No Lag, with Inflation Expectations

	TR	TR w/ SR	TR	TR w/ SR
(Intercept)	1.3522 ***	0.3097	1.3470 ***	0.1985
	(0.3168)	(1.5707)	(0.3706)	(1.6118)
exp_inflation_gap	1.7281 ***	3.7703 **	1.7164 ***	3.5221 **
	(0.3555)	(1.3952)	(0.3482)	(1.1797)
output_gap	0.0712	-0.0013	0.0672	-0.0870
	(0.0469)	(0.1873)	(0.0811)	(0.2132)
realised_inflation_gap			0.0146	0.3117
			(0.1278)	(0.3424)
N	96	96	96	96
R2	0.6180	0.3914	0.6182	0.4089

*** p < 0.001; ** p < 0.01; * p < 0.05.

Lagged Models

The lagged Taylor Rule specifications follow the regression expressed in equation (3) which is the same as (2) but also including a one-quarter lag for the interest rate. We then run the same 6 variations as for the models without the lag.

$$i_t = \pi^* + \phi i_{t-1} + \beta(\pi_t - \pi^*) + \gamma(y_t - \bar{y}_t) \qquad (3)$$

```
# --------- 1. Simple Taylor Rule without lag ---------

# Fit models
lTR <- lm(rate ~ rate_lag + realised_inflation_gap + output_gap, data = data)
lTRsr <- lm(shadowrate ~ shadowrate_lag + realised_inflation_gap +
              output_gap, data = data)

# List them and print them as tables using our helper function
models <- list(lTR, lTRsr)
```

```r
taylor_regression_to_table(models, caption = "Interest Rate Lag,
                                              No Expectations")
```

Table 5: Interest Rate Lag, No Expectations

	TR	TR w/ SR
(Intercept)	0.0512	-0.0756
	(0.0406)	(0.0592)
rate_lag	0.9194 ***	
	(0.0381)	
realised_inflation_gap	0.0877 *	0.2493 ***
	(0.0348)	(0.0343)
output_gap	0.0219	-0.0113
	(0.0163)	(0.0184)
shadowrate_lag		0.9535 ***
		(0.0173)
N	96	96
R2	0.9597	0.9828

*** p < 0.001; ** p < 0.01; * p < 0.05.

```r
# --------- 2. Taylor Rule without lag but with inflation expectations  ---------

# Fit models
lTR_e <- lm(rate ~ rate_lag + exp_inflation_gap + output_gap, data = data)
lTRsr_e <- lm(shadowrate ~ shadowrate_lag + exp_inflation_gap +
                output_gap, data = data)
lTR_ie <- lm(rate ~ rate_lag + realised_inflation_gap + exp_inflation_gap +
               output_gap, data = data)
lTRsr_ie <- lm(shadowrate ~ shadowrate_lag + realised_inflation_gap +
                 exp_inflation_gap + output_gap, data = data)

# List them and print them as tables using our helper function
models <- list(lTR_e, lTRsr_e, lTR_ie, lTRsr_ie)
taylor_regression_to_table(models, caption = "Interest Rate Lag, with
                                              Inflation Expectations")
```

Checking for structural breaks

The previous Taylor Rule estimation are most likely flawed if there have been structural breaks in the economic system that would influence the way the ECB reacts to the economy. Given this, we run both a

Table 6: Interest Rate Lag, with Inflation Expectations

	TR	TR w/ SR	TR	TR w/ SR
(Intercept)	0.1805 *	0.0012	0.1343 ***	-0.0876
	(0.0791)	(0.0852)	(0.0376)	(0.0594)
rate_lag	0.8612 ***		0.8750 ***	
	(0.0437)		(0.0381)	
exp_inflation_gap	0.2381 **	0.1295	0.1530 ***	-0.0548
	(0.0734)	(0.1038)	(0.0434)	(0.0618)
output_gap	0.0449 *	0.0592	0.0234	-0.0106
	(0.0218)	(0.0474)	(0.0165)	(0.0181)
shadowrate_lag		0.9630 ***		0.9586 ***
		(0.0287)		(0.0187)
realised_inflation_gap			0.0768 *	0.2528 ***
			(0.0378)	(0.0346)
N	96	96	96	96
R2	0.9547	0.9714	0.9614	0.9829

*** $p < 0.001$; ** $p < 0.01$; * $p < 0.05$.

Chow test and a Bai-Perron test in order to see if there are any using the “strucchange” package. The Chow test is run on two dates where we suspect there have been breaks which are in the third quarter of 2012, representing the start of the Zero Lower Bound, and in the first quarter of 2020, representing the start of the COVID-19 pandemic. The Bai-Perron test, however, works by finding any structural breaks present in the overall sample which minimise the Aikake Information Criterion.

```
# This first function runs all the steps necessary for a Chow test to check
#  for structural breaks in key moments. It runs it for 2 dates at a time
#   and uses the formula we set in options

chow_tests <- function(data, break1, break2, events_name) {

  # Formula to test for breaks (uses formula selected in options config)
  break_formula = model_formula

  # Chow test (rejecting the null means there are structural breaks)
  chow_test1 <- sctest(break_formula, type = "Chow",
                       point = break1, data = data)
  chow_test2 <- sctest(break_formula, type = "Chow",
                       point = break2, data = data)

  # Get dates of the breakpoints
```

Table 7: Chow tests for suspected structural breaks

Event	Date	p-value
ZLB Start	2012 Q3	0.0018
COVID-19 Start	2020 Q1	0.1213

```
  breakpoint1_date <- data$quarter[break1]
  breakpoint2_date <- data$quarter[break2]

  table_output <- generate_chow_table(test1 = chow_test1,
                                      test2 = chow_test2,
                                      date1 = breakpoint1_date,
                                      date2 = breakpoint2_date,
                                      events_name = events_name)
  return(table_output)
}
```

```
# --------- 1. Chow test ---------

# Run test using helper function
chow_tests(data, break1 = 55, break2 = 85,
           events_name <- c("ZLB Start", "COVID-19 Start"))
```

```
# This second function runs the estimation for the Bai-Perron test which
#  detects structural breaks wherever in the data

bp_tests <- function(data) {

  # Formula to test for breaks (uses formula selected in options config)
  break_formula = model_formula

  # Estimate Bai-Perron test & output results
  BP_test = breakpoints(break_formula, data = data)
  BP_test_res = summary(BP_test)

  # Optimal values (modular!)
  bic_values <- BP_test_res$RSS[2, ]
  optimal_m <- as.numeric(names(bic_values)[which.min(bic_values)])

  table_output <- generate_bp_table(base_data = data,
                                    BP_test_res = BP_test_res,
                                    optimal_m = optimal_m)
  return(table_output)
}
```

```
# --------- 2. Bai-Perron test ---------
# Run test using helper function
bp_tests(data)
```

Table 8: Bai-Perron test for multiple breaks

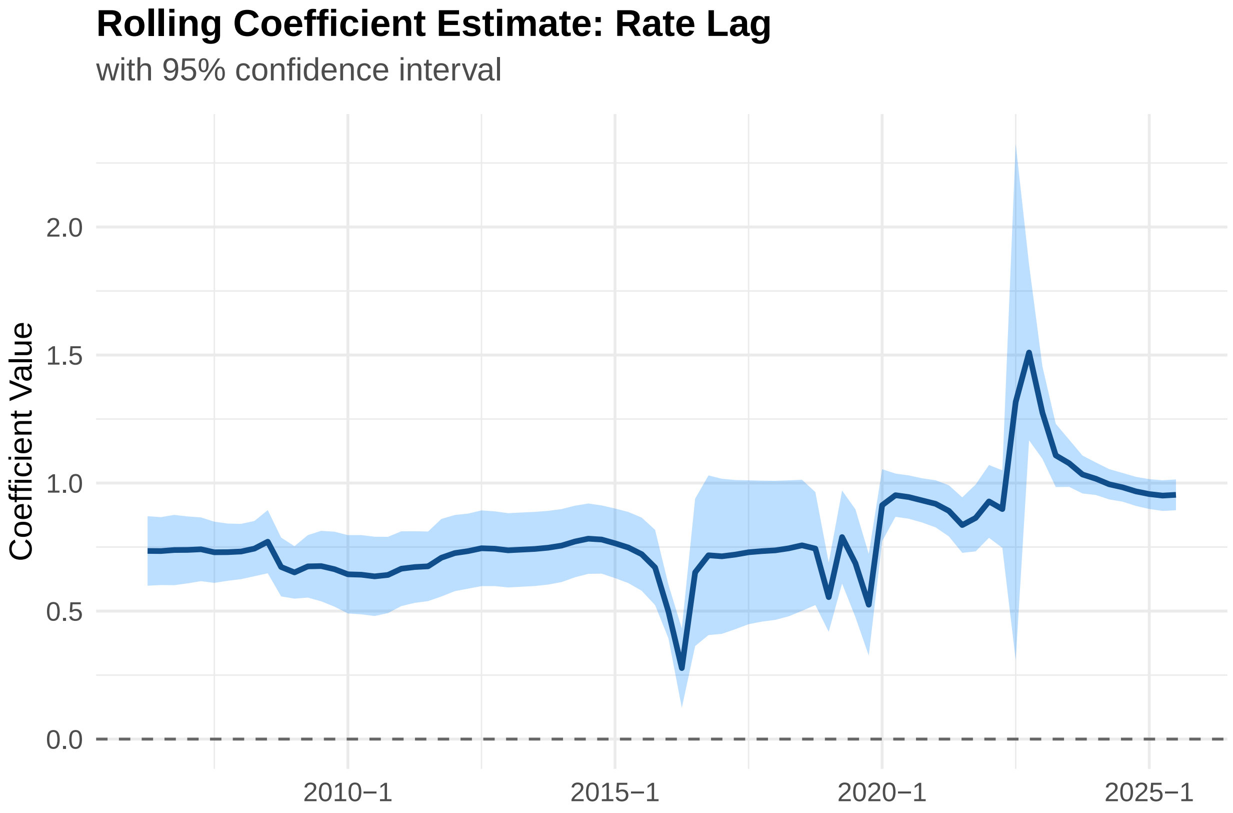
Detected Breaks
2006 Q2
2019 Q2

Rolling Estimation

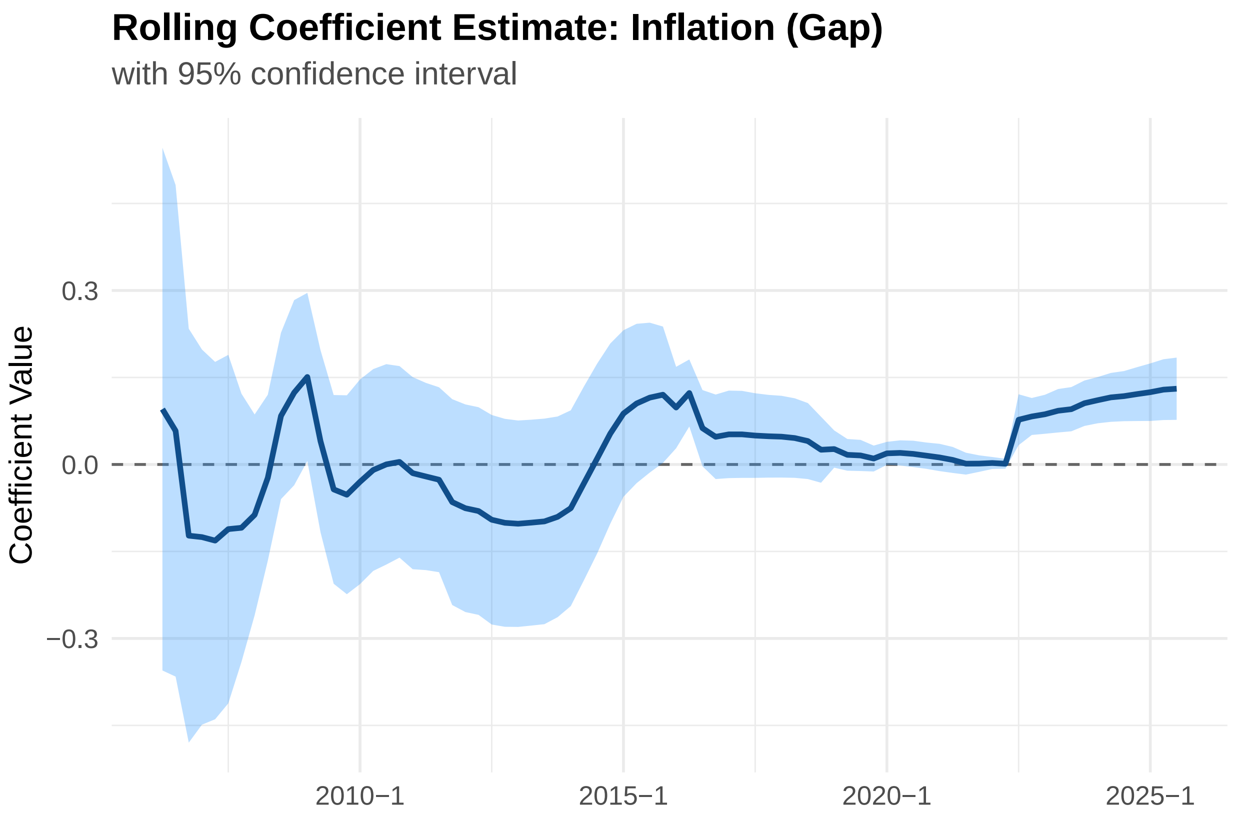
Given that there have been structural breaks (according to our tests), we decide to run a rolling estimation scheme for the Taylor Rule formula selected in the setup options configuration. To do this, we use a custom function which runs the chosen model firstly on the first data "window" (which can be selected to be however many quarters desired) and then moves this window forward in time by one quarter at a time, running the estimation for each loop. We then use another custom function to plot the resulting coefficients through time, including confidence intervals.

```r
# This function run a rolling sample estimation loop for the main TR formula

estimate_rolling_TR <- function(data, formula, W) {

  # Based on the quarter rolling window W parameter, automatically output
  #  the Looping Parameter
  L = nrow(data) - W + 1

  # Preparation of result data, dates, var names, and confidence intervals
  var_names <- attr(terms(formula), "term.labels")
  window_end_dates <- data$quarter[W:nrow(data)] # First window [1:W] ends at data$date[W]
  TR_roll <- data.frame(date = window_end_dates)
  TR_roll[var_names] <- NA
  lower_col_names <- paste0(var_names, "_lower")
  upper_col_names <- paste0(var_names, "_upper")

  # Looped estimation of TR, outputs coefficients and CIs
  for (l in 1:L) {

    # 1. Define splits (with rolling scheme)
    rolled_data <- data[l:(W + l - 1), ]

    # 2. Estimate TR on split data, using whatever formula is desired
    TR_estimate <- lm(formula, data = rolled_data)

    # 3. Pull out coefficients & compute confidence intervals
    all_coefs <- coef(TR_estimate)
    all_cis <- confint(TR_estimate)

    TR_roll[l, var_names] <- all_coefs[var_names]
    TR_roll[l, lower_col_names] <- all_cis[var_names, 1]
    TR_roll[l, upper_col_names] <- all_cis[var_names, 2]
  }
  return(TR_roll)
}
```

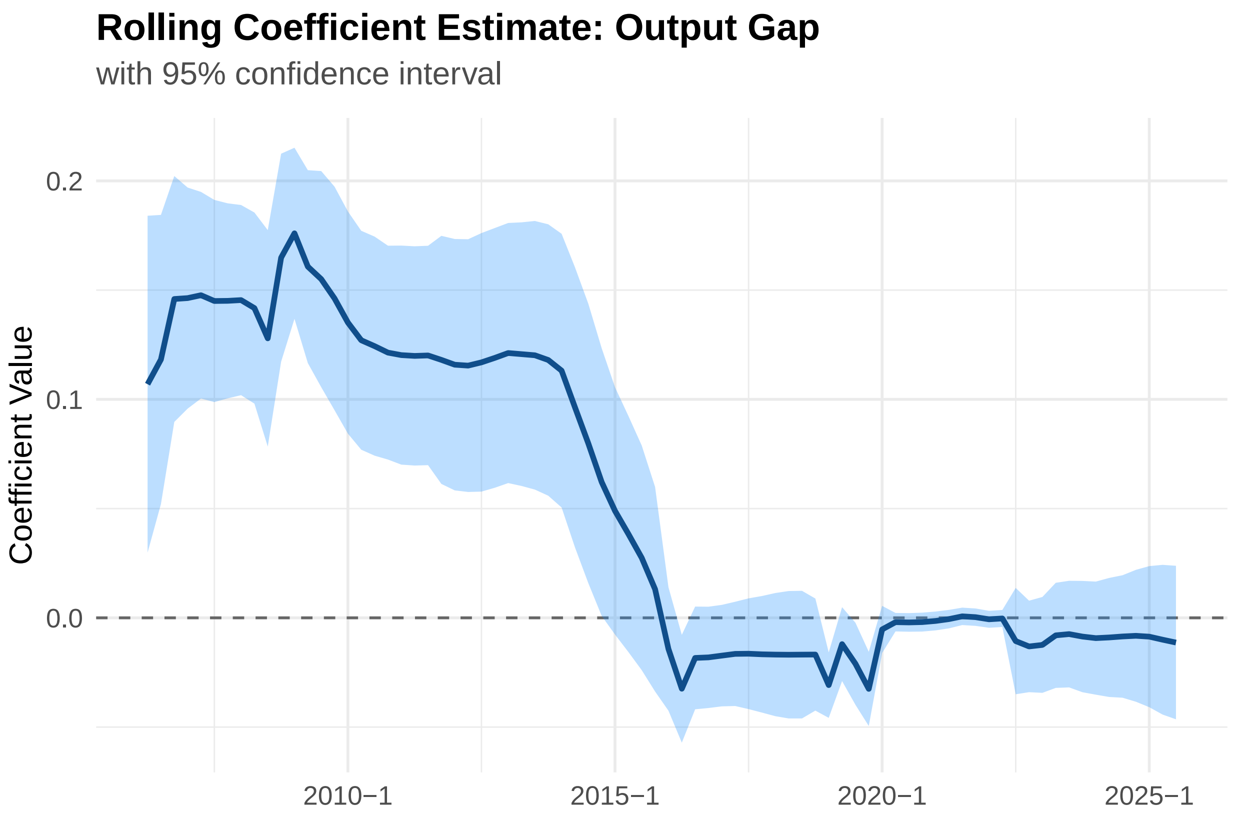
```r
# --------- 1. Estimation ---------

# Estimate a rolling-window (W in quarters) Taylor Rule specification
TR_roll <- estimate_rolling_TR(data, model_formula, W = 30)

# --------- 1. Plots ---------

# Note: this part is not modular
plot_rolling_coefs(TR_roll, "rate_lag", var_name_title="Rate Lag")
```



```r
plot_rolling_coefs(TR_roll, "inflation_gap", var_name_title="Inflation (Gap)")
```



```
plot_rolling_coefs(TR_roll, "output_gap", var_name_title="Output Gap")
```


Rolling Coefficient Estimate: Output Gap
with 95% confidence interval
Coefficient Value
0.2
0.1
0.0
2010−1
2015−1
2020−1
2025−1

Forecasting Model Evaluation

Methodology

Our method of forecasting consists of two steps. First, we run our custom auto ARIMA function on the inputs of the Taylor Rule (inflation and output gap). This function replicates the one in the “forecast” package, though does not loop over the differentiation parameter as we want to set that value manually based on the stationarity tests run above. We additionally run this function on the interest rate in order to serve as our benchmark model for relative forecasting performance evaluation. In order to compute the forecasts based on the fitted ARIMA models, we use a custom function that replicates the same use-case in the “forecast” package. At the same time as the ARIMA fits, we use OLS to compute the Taylor Rule coefficients. The second step is then to do the actual forecasting, where we predict future values of the interest rate based on the forecasted values of the inputs using the fitted ARIMA models, weighted by the coefficients computed with OLS.

```r
# This first small function just enables easy automatic interpretations of
#   the final selected models.
explain_arima <- function(model_object, var_name) {

  var_name = var_name

  # Extract the resulting model
  vec <- model_object$arma

  # Get the specification for p,d,q and put in string
  p <- vec[1]
  q <- vec[2]
  d <- vec[6]
  spec_str <- paste0("ARIMA(", p, ", ", d, ", ", q, ")")

  # Make it legible
  result <- paste0("The ARIMA model fitted to the '", var_name,
                   "' variable is specified as ", spec_str, ". \n\n")
  return(result) }
```

```r
# This function searches for the best p and q for a fixed d (unnecessary
#   to loop over d in our project since we either set d=0 for output gap
#    or d=1 for interest and inflation)

my.auto.arima <- function(x, max.p, max.q, d, var_name="x") {

  # Get the length of the time series
  # We use n = length(x) - d as the effective sample size for BIC/AICc
  # Note: arima()$nobs will give the exact number used.
  T_orig <- length(x)

  # Initialize matrices to store the criteria values
  matrix.AIC <- matrix(NA, max.p + 1, max.q + 1)
  matrix.BIC <- matrix(NA, max.p + 1, max.q + 1)
  matrix.AICc <- matrix(NA, max.p + 1, max.q + 1)

  # Add row and column names for clarity
  rownames(matrix.AICc) <- 0:max.p
```

```r
colnames(matrix.AICc) <- 0:max.q
rownames(matrix.AIC) <- 0:max.p
colnames(matrix.AIC) <- 0:max.q
rownames(matrix.BIC) <- 0:max.p
colnames(matrix.BIC) <- 0:max.q


# Loop over all p from 0 to max.p
for(p in 0:max.p) {
  # Loop over all q from 0 to max.q
  for(q in 0:max.q) {

    # tryCatch to handle errors
    fit.arima <- try(
      arima(x, order = c(p, d, q), method = "ML"),
      silent = TRUE)

    # Check if the fit was successful
    if(!inherits(fit.arima, "try-error")) {

      # Get log-likelihood
      loglik <- fit.arima$loglik

      # Get number of parameters.
      # This includes p, q, and the mean (if d=0), plus the variance (sigma^2).
      k <- length(fit.arima$coef) + 1

      # Get effective number of observations
      n <- fit.arima$nobs

      # Calculate AIC
      aic_val <- -2 * loglik + 2 * k
      matrix.AIC[p + 1, q + 1] <- aic_val

      # Calculate BIC
      matrix.BIC[p + 1, q + 1] <- -2 * loglik + k * log(n)

      # Calculate AICc (AIC corrected for small samples)
      # We add a check for n - k - 1 > 0 to avoid division by zero
      if (n - k - 1 > 0) {
        matrix.AICc[p + 1, q + 1] <- aic_val + (2 * k^2 + 2 * k) / (n - k - 1)
      }}}}

# Find the p and q that minimize the AICc
# 'na.rm=TRUE' jut in case of fails
min_AICc <- min(matrix.AICc, na.rm = TRUE)
position_AICc <- which(matrix.AICc == min_AICc, arr.ind = TRUE)

# Get the first match if there are ties
opt.p <- position_AICc[1, 1] - 1  # -1 to convert from 1-based index to p-value
opt.q <- position_AICc[1, 2] - 1  # -1 to convert from 1-based index to q-value

# Store the optimal parameters
```

```r
  parameters <- c(opt.p, d, opt.q)

  final.model <- arima(x, order = parameters, method = "ML")

  # Print the model summary
  print <- explain_arima(final.model, var_name = var_name)
  cat(print)

  # Return a list with all the results
  return(final.model)}
```

```r
# This function produces forecasts from the my.auto.arima output

my.forecast = function(model, h = 1) {
  prediction_output = predict(object = model, n.ahead = h)
  values <- as.vector(prediction_output$pred)
  return(values)  }
```

```r
# This function replicates the dm.test function from the forecast package

dm_test <- function(e1, e2, alternative = c("two.sided", "less", "greater"),
                    h = 1) {

  # 1. Setup and Arguments
  alternative <- match.arg(alternative)
  h <- as.integer(h)

  # 2. Calculate Loss Differential
  d <- e1^2 - e2^2
  P <- length(d)

  # 3. Calculate Variance-Covariance Matrix
  d.cov <- acf(d, na.action = na.omit, lag.max = h - 1,
               type = "covariance", plot = FALSE)$acf[, , 1]

  # Variance estimator formula
  d.var <- sum(c(d.cov[1], 2 * d.cov[-1])) / P

  # 4. Logic for Negative Variance
  # If variance is negative/zero for h>1, fall back to h=1
  # We couldn't manage to run our code without including this part
  # (taken from the base package)
  # Probable reason is that our R starts quite "late"
  if (d.var > 0) {
    STATISTIC <- mean(d, na.rm = TRUE) / sqrt(d.var)
  } else if (h == 1) {
    stop("Variance of DM statistic is zero")
  } else {
    warning("Variance is negative. Proceeding with horizon h=1.")
    return(dm.test(e1, e2, alternative = alternative, h = 1))
  }

  # 5. Apply Harvey-Leybourne-Newbold correction (like in slides)
  #     (corrects for small samples, useful for us)
  k <- ((P + 1 - 2 * h + (h / P) * (h - 1)) / P)^(1/2)
  STATISTIC <- STATISTIC * k

  # 6. pvalue with student distribution (as seen in class)
  #     use (P-1) degrees of freedom
  if (alternative == "two.sided") {
    PVAL <- 2 * pt(-abs(STATISTIC), df = P - 1)
  } else if (alternative == "less") {
    PVAL <- pt(STATISTIC, df = P - 1)
  } else if (alternative == "greater") {
    PVAL <- pt(STATISTIC, df = P - 1, lower.tail = FALSE)
  }

  # 7. Output
  structure(list(
    alternative = alternative,
    p.value = PVAL,
    method = "Diebold-Mariano Test",
    data.name = c(deparse(substitute(e1)), deparse(substitute(e2)))
  ), class = "htest")
}
```

Pseudo Out of Sample Estimation

In order to asses the performance of this methodology, we use a pseudo out-of-sample rolling estimation scheme of direct forecasts for all Taylor Rule formulas and for the benchmark ARIMA. We forecast up to a horizon of 10 quarters ahead. To do this, we loop over quarters starting just before the evaluation sample, where we then implement our methodology to forecast up to the farthest horizon. We then store these values and the loop starts again, adding one more quarter from the evaluation sample, and dropping the very first quarter in the overall sample. For the lagged models, we additionally need to include an iterative forecasting step, where we loop over each horizon in order to use the previous point forecast to compute the next one. In order to speed up the process, we include a simple parallelization procedure which runs these loops in parallel on a fourth the the users' CPU cores.

Parameters

```
# Rolling window size
R = 85 # Chow: Structural breaks at R=55 and R=85
cat("Evaluation sample starts after ",as.character(data$quarter[R]),".",sep="")
```

```
## Evaluation sample starts after 2020 Q1.
```

```
# Start of evaluation sample
P = nrow(data) - R # Will effectively be: P = T-h-R

# Number of different horizons (takes 10 to go until 2028 Q1)
H = 10
```

Pre-allocate storage for all results

```r
# We need 4 lists for the TR models, 1 list for the shared benchmark
init_storage_list <- function(H, P) {
  storage <- vector("list", length = H)
  for (h in 1:H) {
    storage[[h]] <- rep(NA_real_, P)}
  return(storage)}

# Storage for realised values
Actuals <- init_storage_list(H, P)

# Storage for Forecasts
F_TR_1 <- init_storage_list(H, P) # Model 1: shadowrate, no lag
F_TR_2 <- init_storage_list(H, P) # Model 2: rate, no lag
F_TR_3 <- init_storage_list(H, P) # Model 3: shadowrate, with lag
F_TR_4 <- init_storage_list(H, P) # Model 4: rate, with lag
F_BM   <- init_storage_list(H, P) # Benchmark: ARIMA

# Storage for Forecast Errors
FE_TR_1 <- init_storage_list(H, P) # Model 1: shadowrate, no lag
FE_TR_2 <- init_storage_list(H, P) # Model 2: rate, no lag
FE_TR_3 <- init_storage_list(H, P) # Model 3: shadowrate, with lag
FE_TR_4 <- init_storage_list(H, P) # Model 4: rate, with lag
FE_BM   <- init_storage_list(H, P) # Benchmark: ARIMA
```

Setup & run the parallel "backtesting" (rolling) loop

```r
num_cores <- detectCores() / 4
cl <- makeCluster(num_cores)
registerDoParallel(cl)

# .export sends read-only objects to each core
# .packages loads libraries on each core
worker_results <- foreach(
  p = P:1,
  .packages = c("forecast", "stats", "dplyr"),
  .export = c("data", "H", "formula_1", "formula_2", "formula_3", "formula_4")
) %dopar% {

  # 1. Define splits (with rolling scheme)
  training <- data[(1 + nrow(data) - R - p):(nrow(data) - p), ]
  testing <- data[(nrow(data) - (p - 1)):nrow(data), ]

  # --- 2. Fit common models only once ---
  # note: d=1 for interest and inflation as non-stationary
  inflation_arma <- my.auto.arima(training$inflation_gap, max.p=4, max.q=4, d=1)
  outputgap_arma <- my.auto.arima(training$output_gap, max.p=4, max.q=4, d=0)
  interest_arma <- my.auto.arima(training$rate, max.p=4, max.q=4, d=1) # Benchmark

  # --- 3. Get common forecasts only once (all H horizons) ---
  inflation_forecasts <- my.forecast(inflation_arma, h = H)
  outputgap_forecasts <- my.forecast(outputgap_arma, h = H)
  BMpredicted_rates <- my.forecast(interest_arma, h = H)

  # --- 4. Fit the 4 TR models ---
  TR_model_1 <- lm(formula_1, data = training)
  TR_model_2 <- lm(formula_2, data = training)
  TR_model_3 <- lm(formula_3, data = training)
  TR_model_4 <- lm(formula_4, data = training)

  # --- 5. Build forecast input data & get forecasts for non-lagged models ---
  # These are direct forecasts
  new_data_base <- data.frame(
    inflation_gap = inflation_forecasts,
    output_gap = outputgap_forecasts)

  TR_preds_1 <- round(pmax(predict(TR_model_1, new_data_base), min(data$rate)) / 0.25) * 0.25
  TR_preds_2 <- round(pmax(predict(TR_model_2, new_data_base), min(data$rate)) / 0.25) * 0.25
  BM_preds <- round(pmax(BMpredicted_rates, min(data$rate)) / 0.25) * 0.25

  # --- 6. Get forecasts for lagged models via iteration ---
  # We must loop 1 step at a time, feeding forecasts back in.

  # a) Pre-allocate storage for H forecasts
  TR_preds_3 <- numeric(H)
  TR_preds_4 <- numeric(H)

  # b) Get the last known lag from the training set (lag for h=1 forecast)
  current_rate_lag  <- last(training$rate)
  current_shadowrate_lag <- last(training$shadowrate)

  # Loop for iterative forecasting
  for (h in 1:H) {
    # --- Prepare dataset for predictions ---
    new_data_3_h <- data.frame(
      inflation_gap = inflation_forecasts[h],
      output_gap = outputgap_forecasts[h],
      rate_lag = current_rate_lag)
    new_data_4_h <- data.frame(
      inflation_gap = inflation_forecasts[h],
      output_gap = outputgap_forecasts[h],
      shadowrate_lag = current_shadowrate_lag )

    # Get the forecast values (keep for lag, and then round for actual prediction)
    pred_3_h <- predict(TR_model_3, new_data_3_h)
    TR_preds_3[h] <- round(pmax(pred_3_h, min(data$rate)) / 0.25) * 0.25
    pred_4_h <- predict(TR_model_4, new_data_4_h)
    TR_preds_4[h] <- round(pmax(pred_4_h, min(data$rate)) / 0.25) * 0.25

    # Update lag for h+1
    current_shadowrate_lag <- pred_4_h
    current_rate_lag <- pred_3_h }
  # --- 7. Get actual values in evaluation sample  ---
  actual_rates <- testing$rate[1:H]

  # --- 8. Return all forecasts and actuals from the worker ---
  list(F_TR_FORMULA_1 = TR_preds_1,
       F_TR_FORMULA_2 = TR_preds_2,
       F_TR_FORMULA_3 = TR_preds_3,
       F_TR_FORMULA_4 = TR_preds_4,
       F_BM  = BM_preds,
       actuals = actual_rates) }

# --- Stop the Cluster ---
stopCluster(cl)
rm(cl)
```

Unpack parallel results into storage lists

```
# 'worker_results' is a list of P lists. We need to re-organize it.
for (i in 1:P) {
  # i=1 corresponds to p=P, i=2 to p=P-1, ... i=P to p=1
  # This 'storage_index' matches the loop order
  storage_index <- i
  p_results <- worker_results[[i]]

  for (h in 1:H) {
    # Get the raw values for this h
    actual_val <- p_results$actuals[h]
    f_tr1_val  <- p_results$F_TR_FORMULA_1[h]
    f_tr2_val  <- p_results$F_TR_FORMULA_2[h]
    f_tr3_val  <- p_results$F_TR_FORMULA_3[h]
    f_tr4_val  <- p_results$F_TR_FORMULA_4[h]
    f_bm_val   <- p_results$F_BM[h]

    # Store Actuals (for MZ)
    Actuals[[h]][storage_index] <- actual_val

    # Store Forecasts (for MZ)
    F_TR_1[[h]][storage_index] <- f_tr1_val
    F_TR_2[[h]][storage_index] <- f_tr2_val
    F_TR_3[[h]][storage_index] <- f_tr3_val
    F_TR_4[[h]][storage_index] <- f_tr4_val
    F_BM[[h]][storage_index]   <- f_bm_val

    # Calculate and Store Errors (for MSFE/DM)
    FE_TR_1[[h]][storage_index] <- f_tr1_val - actual_val
    FE_TR_2[[h]][storage_index] <- f_tr2_val - actual_val
    FE_TR_3[[h]][storage_index] <- f_tr3_val - actual_val
    FE_TR_4[[h]][storage_index] <- f_tr4_val - actual_val
    FE_BM[[h]][storage_index]   <- f_bm_val  - actual_val } }
```

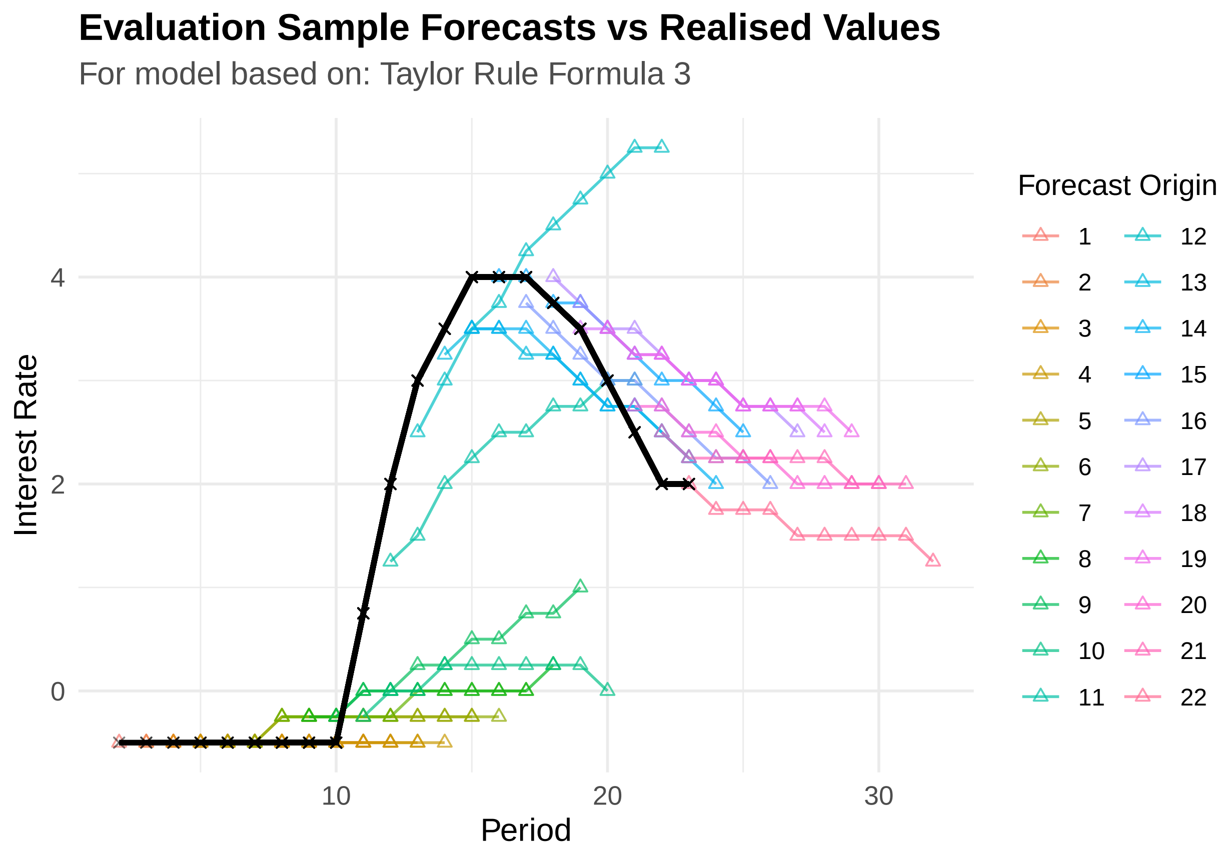
Render results more intuitive for further analysis

```r
# Convert the forecast lists (F_TR_x) into single dataframes
forecast_to_df <- function(forecast_list, period) {
  # Convert each element to numeric (benchmark is ts object, which is bad)
  numeric_list <- lapply(forecast_list, function(x) as.numeric(x)) #just make each list inside numeric
  df <- as.data.frame(numeric_list)
  # Add period and horizon
  df$period <- period #first list is all horizon 1 forecasts, gives this to all observations
  df$horizon <- 1:nrow(df) #counts rows and gives each the horizon corresponding to it
  df}

# Apply to all forecasting models
eval_all_models <- do.call(rbind, lapply(seq_along(worker_results), function(i) {
  forecast_to_df(worker_results[[i]], period = i) }))
eval_all_models[] <- lapply(eval_all_models, function(x) as.numeric(x))

# Compute date_of_forecast
# period = date the forecast was made
# date_of_forecast = the future date we are predicting
eval_all_models$date_of_forecast <- eval_all_models$period +
  eval_all_models$horizon

# Compute forecast error
eval_all_models$forecast_error <- eval_all_models[[model]] -
  eval_all_models$actuals

# Compute variance of forecast errors for the h step ahead forecast
#  (also computes mean)
var_by_horizon <- eval_all_models %>%
  group_by(horizon) %>%
  summarize(
    mean_fe = mean(forecast_error, na.rm=T),
    var_fe = sd(forecast_error, na.rm=T)^2, n = n() )
```

Spaghetti Plots

In order to visualize the pseudo out-of-sample evaluation exercise, we use spaghetti plots of the point forecasts which allow to compare how often the model "misses" the realised values of the interest rate between different horizons and dates. We do the plot only on the on the Taylor Rule formula selected in the options configuration in order to avoid clutter.

```r
spaghetti_plotter(evals = eval_all_models,
                  model = model,
                  model_name = model_name)
```

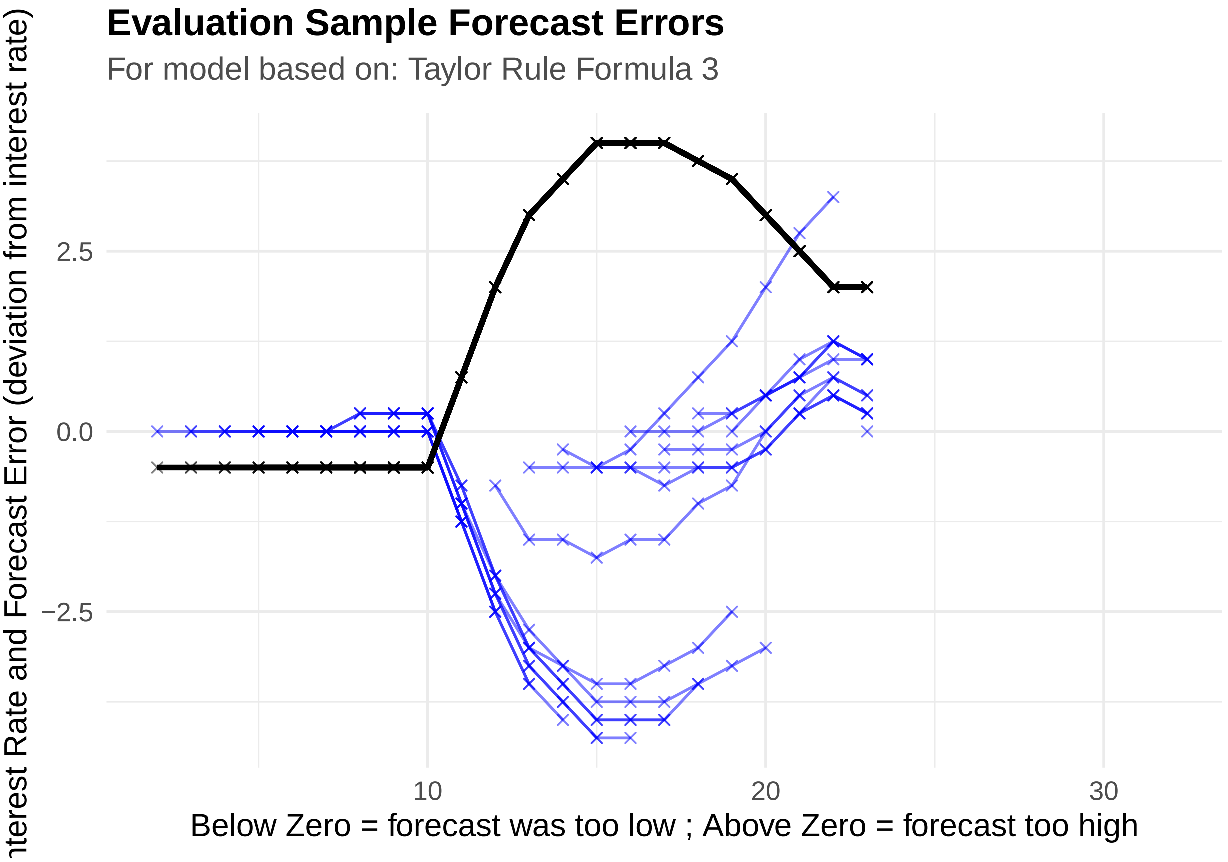


Forecast Errors

Plots of FE

Similar to the spaghetti plot of the point forecasts, we essentially use the same plot but computing the forecast errors as the difference between our point forecast and the realised value.

```
FE_spaghetti_plotter(evals = eval_all_models)
```



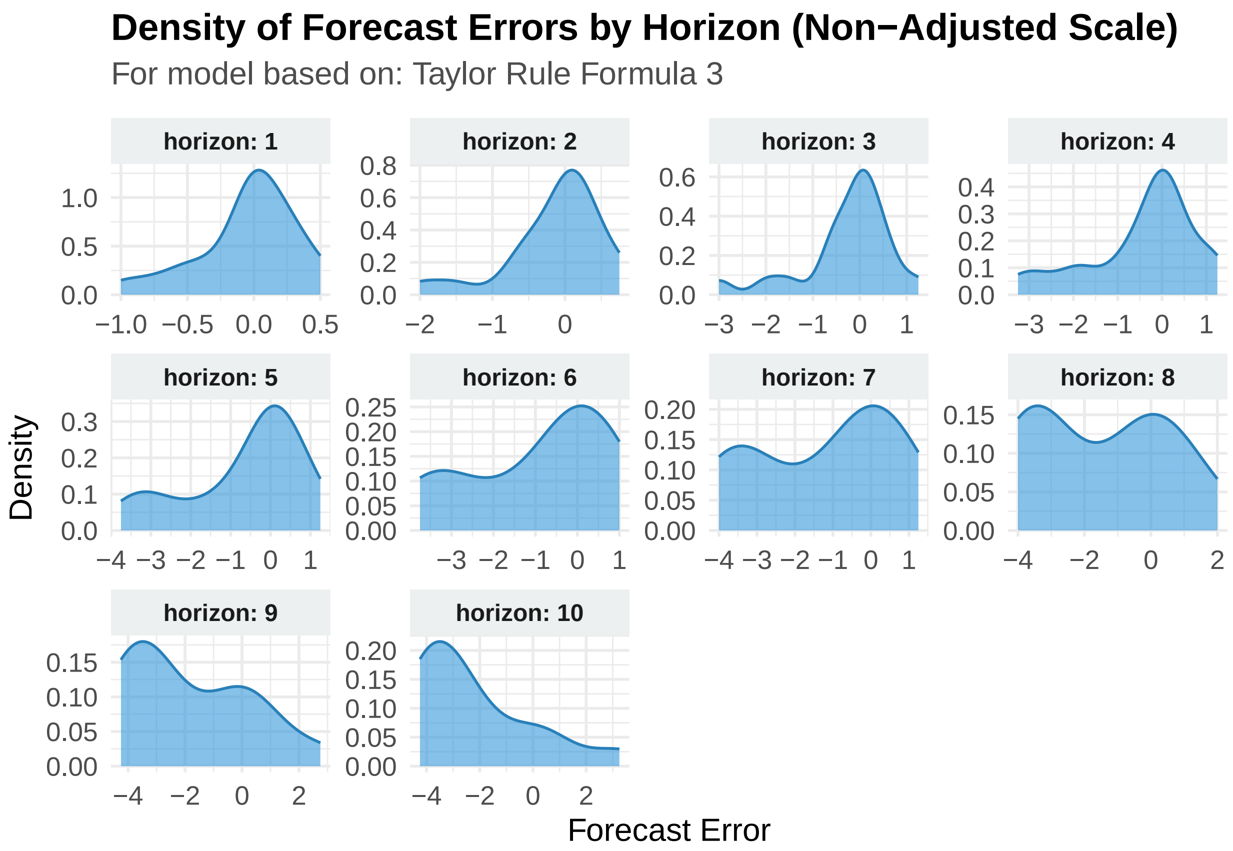
Density of FE

We can then plot the distribution of these forecast errors using a density plot. We include 3 versions: one that is unscaled in order to compare the absolute values, one that is scaled in order to compare relative differences, and one that plots density in a way to compare both.

```
FE_density_plotter_unscaled(evals = eval_all_models)
```

```
## Saving 6.5 x 4.5 in image
```

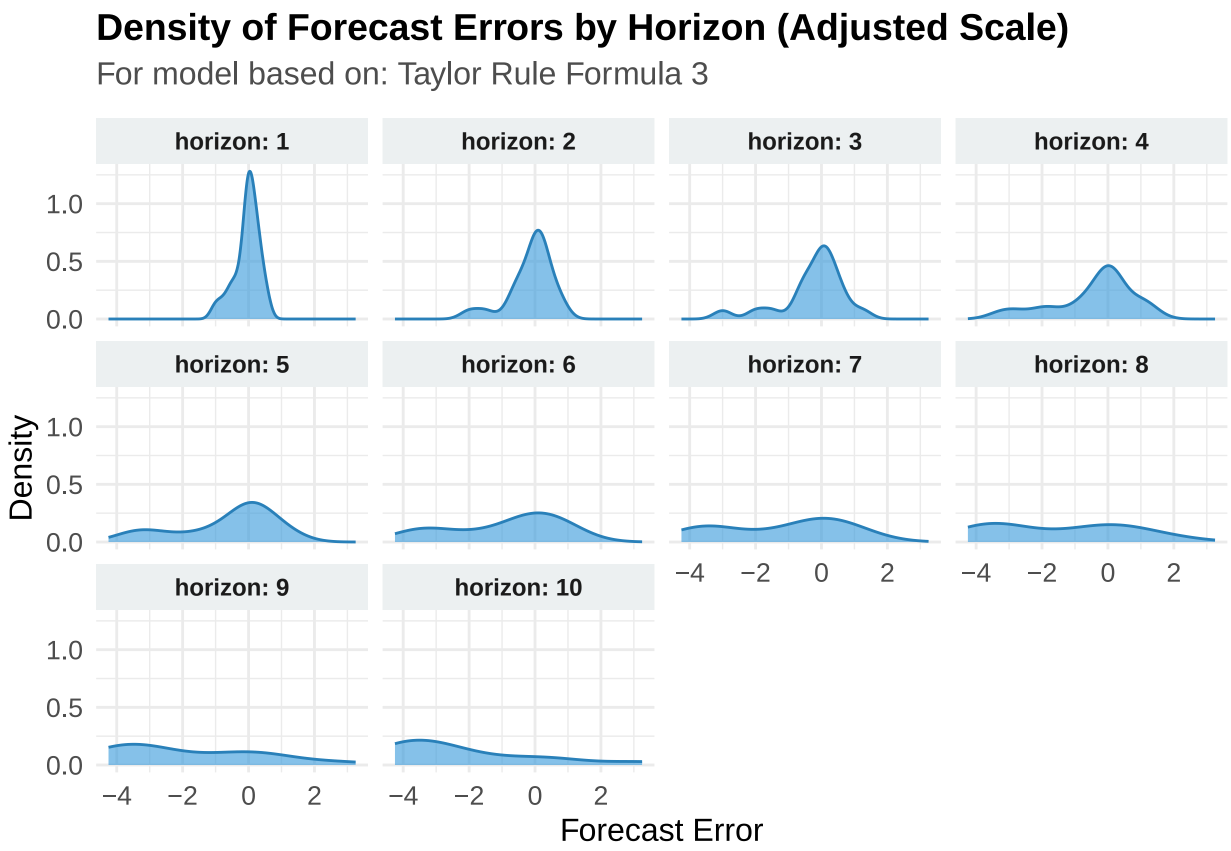
```
## Warning: Removed 45 rows containing non-finite outside the scale range
## (`stat_density()`).
## Removed 45 rows containing non-finite outside the scale range
## (`stat_density()`).
```



```
FE_density_plotter_scaled(evals = eval_all_models)
```

```
## Saving 6.5 x 4.5 in image
```

```
## Warning: Removed 45 rows containing non-finite outside the scale range
## (`stat_density()`).
## Removed 45 rows containing non-finite outside the scale range
## (`stat_density()`).
```

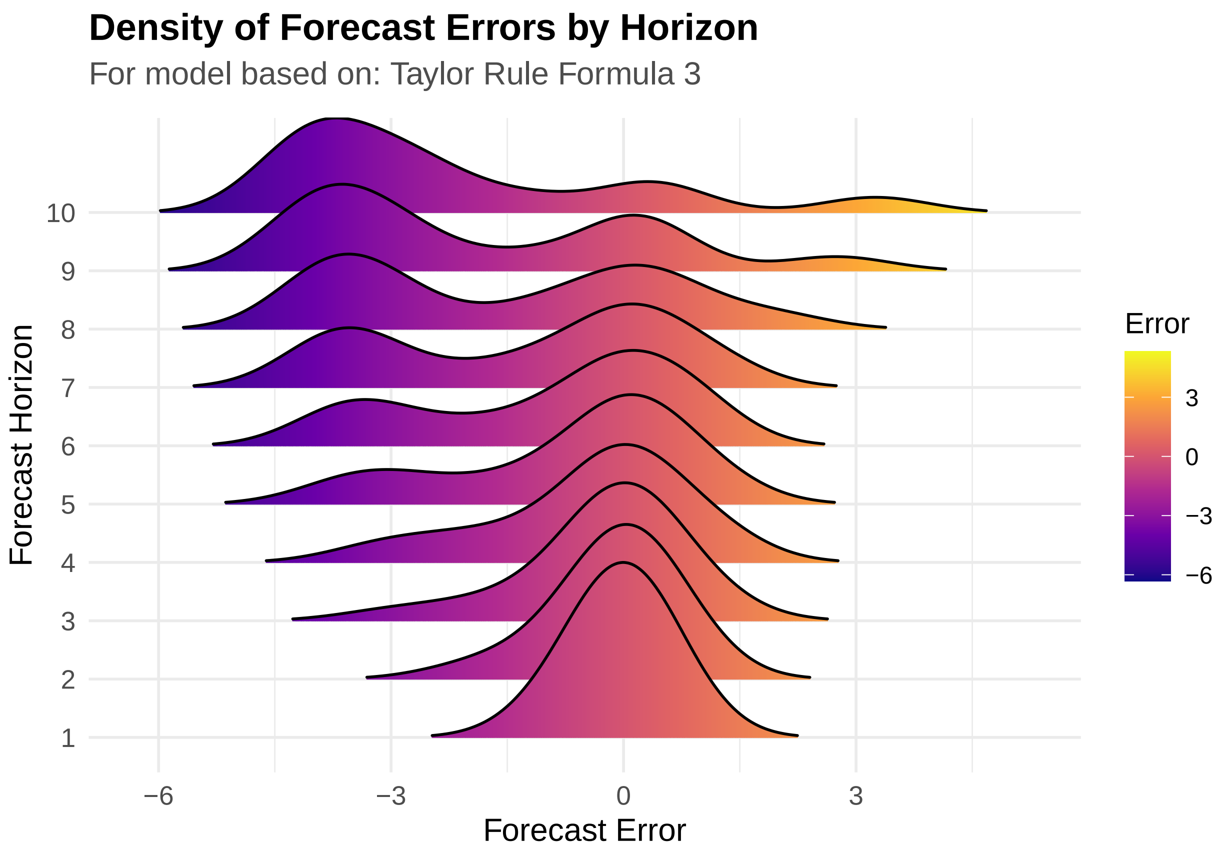


```
FE_density_plotter_ridges(evals = eval_all_models)
```

```
## Saving 6.5 x 4.5 in image
```

```
## Warning: `stat(x)` was deprecated in ggplot2 3.4.0.
## i Please use `after_stat(x)` instead.
## This warning is displayed once every 8 hours.
## Call `lifecycle::last_lifecycle_warnings()` to see where this warning was
## generated.
```

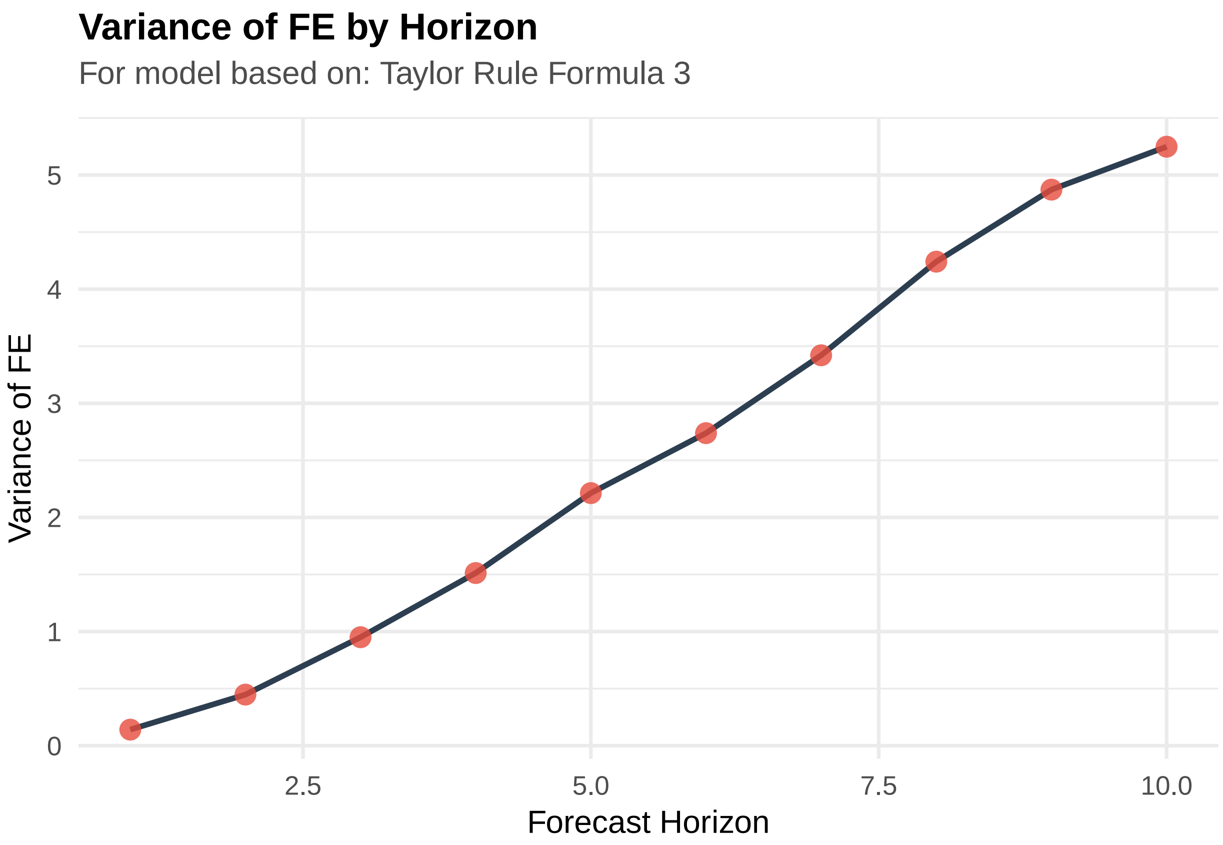
```
## Picking joint bandwidth of 0.69
## Picking joint bandwidth of 0.69
```



Variance of FE

```
FE_variance_plotter(var_by_horizon)
```

```
## Saving 6.5 x 4.5 in image
```



Horizon 1 Autocorrelation of FE

Model 1, 2, 3 are autocorrelated at h=1. Include results in paper, not here in the appendix. Well maybe not idk.

```
# This function runs the Durbin-Watson tests on forecast errors

generate_dw_tests <- function(eval_all_models, formula_cols) {

  dw_results_list <- list()

  for (e_model_name in formula_cols) {
    # Compute forecast errors
    all_errors <- eval_all_models[["actuals"]] - eval_all_models[[e_model_name]]
    h1_errors <- all_errors[eval_all_models$horizon == 1]

    # Run Durbin-Watson test on h=1 errors
    temp_model <- lm(h1_errors ~ 1)
    dw_test_result <- durbinWatsonTest(temp_model)

    # Store result in tibble
    dw_results_list[[e_model_name]] <- tibble(
      horizon = 1,
      dw_stat = dw_test_result$dw,
      p_value = dw_test_result$p
```

Table 9: Durbin Watson Tests

Model	Horizon	DW Statistic	p-value
Formula 1	1	0.1465	0.000 ***
Formula 2	1	0.1799	0.000 ***
Formula 3	1	1.0577	0.018 **
Formula 4	1	2.1154	0.800

Note: This test checks the first order autocorrelation of forecasting errors. A test statistic around 2 indicates no autocorrelation.
DW Test Alternative Hypotheses (H_A):
* ’Greater 2’ means negative autocorrelations.
† ’Lesser 2’ means positive autocorrelation.
‡ Signif. codes: ’***’ 0.01, ’**’ 0.05, ’*’ 0.1

```
    )
  }

  return(dw_results_list)
}
```

```
# Tests for first autocorrelation at h=1

# Select Forecast Columns
formula_cols <- grep("^F_TR_FORMULA_", names(eval_all_models), value = TRUE)

# Automatically set max horizon for loop
max_h <- max(eval_all_models$horizon)

#Run the Table function, the formula is inside
generate_dw_table(eval_all_models,
                  formula_cols,
                  model_caption = "Durbin Watson Tests",
                  format = format)
```

Overall Autocorrelation of FE

We don’t reject H0 -> h+1 are uncorrelated 1,2, are autocorrelated, 3 and 4 are not.

```
# This function runs the Ljung-Box test

generate_ljung_box_test <- function(eval_all_models, formula_cols, max_h = max_h){
  lb_results_list <- list()

  for (model_name in formula_cols) {

    all_errors <- eval_all_models[["actuals"]] - eval_all_models[[model_name]]

    # Collect results for each horizon
    model_results <- tibble(
```

Table 10: Ljung-Box Tests

Horizon	Formula 1	Formula 2	Formula 3	Formula 4
1	47.97 (0.000 ***)	43.55 (0.000 ***)	8.67 (0.070 *)	1.82 (0.770)
2	45.55 (0.000 ***)	31.45 (0.000 ***)	15.23 (0.004 ***)	9.02 (0.061 *)
3	44.9 (0.000 ***)	18.25 (0.001 ***)	15.87 (0.003 ***)	11.89 (0.018 **)
4	40.67 (0.000 ***)	5.79 (0.215)	18.41 (0.001 ***)	12.86 (0.012 **)
5	27.72 (0.000 ***)	2.22 (0.695)	17.36 (0.002 ***)	11.97 (0.018 **)
6	16.42 (0.003 ***)	1.52 (0.824)	16.86 (0.002 ***)	11.28 (0.024 **)
7	10.25 (0.036 **)	1.55 (0.819)	16.02 (0.003 ***)	11.98 (0.017 **)
8	9.62 (0.047 **)	1.09 (0.896)	14.72 (0.005 ***)	12.3 (0.015 **)
9	7.04 (0.134)	0.95 (0.918)	12.3 (0.015 **)	12.41 (0.015 **)
10	8.8 (0.066 *)	0.74 (0.947)	8.83 (0.065 *)	11.81 (0.019 **)

Note: This test checks residual are randomly distributed up to a certain lag.
Ljung-Box Test Alternative Hypotheses (H_A):
* ’There is autocorrelation in the errors at one (or more) lags.

```
      Horizon = 1:max_h,
      Q_stat = numeric(max_h),
      p_value = numeric(max_h)
    )

    for (h in 1:max_h) {
      h_errors <- all_errors[eval_all_models$horizon == h]
      max_lag <- 4 # default quarterly
      lb_test <- Box.test(h_errors, lag = max_lag, type = "Ljung-Box")
      model_results$Q_stat[h] <- lb_test$statistic
      model_results$p_value[h] <- lb_test$p.value
    }

    lb_results_list[[model_name]] <- model_results
  }
  return(lb_results_list)
}
```

```
# With Columns & max lags set in Durbin Watson code chunk

# Loop through each TR
generate_ljung_box_table(eval_all_models,
                         formula_cols,
                         max_h = max_h,
                         format = format,
                         model_caption = "Ljung-Box Tests")
```

Errors Normally Distributed

Run the Jarque-Bera test using a custom function that explain loop and package where the actual test function is made

```r
# These Are all functions needed to generate the jarque bera test to test
#  for normality of errors in each horizon

# Helper function where formula cols is the list of Models, for each element
#  in formula cols (model_name) produce the caption
create_jb_specs <- function(model_vector) {
  lapply(model_vector, function(model_name) {
    list(
      model_name = model_name,
      caption = paste("Jarque-Bera Test Results for Model:", model_name)
    )
  })
}

# This function generates the jarques-bera test
# it creates it for 1 set of errors for each horizon
# Input is a model name and the dataframe where all the results and actual values are stored
# max_h is defined globally for all tests
generate_jb_report <- function(model_name, eval_all_models, max_h) {

  # for the model get all errors
  all_errors <- eval_all_models[["actuals"]] - eval_all_models[[model_name]]

  # generate dataframe for storage
  model_results <- data.frame(
    horizon = integer(),
    X_squared = numeric(),
    p_value = numeric(),
    stringsAsFactors = FALSE
  )

  # loop pver each horizon
  for (h in 1:max_h) {

    # select horizon error
    h_errors <- all_errors[eval_all_models$horizon == h]

    # drop nans
    h_errors <- h_errors[!is.na(h_errors)]

    # below 3 observation really not useful so skip
    if (length(h_errors) < 3) {
      model_results <- dplyr::bind_rows(
        model_results,
        data.frame(horizon = h, X_squared = NA, p_value = NA)
      )
      next # break and return back to start of loop
    }

    # do test
    jb <- jarque.bera.test(h_errors)

    # extract and store results in df
    model_results <- dplyr::bind_rows(
      model_results,
      data.frame(
        horizon = h,
        X_squared = jb$statistic,
        p_value = jb$p.value
      )
    )
  }

  # resturn results
  return(model_results)
}


# This  function uses the previous one and collects them into a nice tavle

generate_all_jb_reports <- function(formula_cols, eval_all_models, max_h, format = format) {

  jb_specs <- create_jb_specs(formula_cols)

  jb_results <- list()

  for (i in seq_along(jb_specs)) {
    spec <- jb_specs[[i]]

    jb_results[[spec$model_name]] <- tryCatch({

      # Generate JB report for this model using previous function
      results <- generate_jb_report(
        model_name = spec$model_name,
        eval_all_models = eval_all_models,
        max_h = max_h
      )

      # Return the results
      results

    }, error = function(e) {
      message("Error generating JB report for ", spec$model_name, ": ", e$message)
      message("Skipping this report and continuing...")
      return(NULL)
    })
  }

  # Use table generator helper function
  table_output <- format_jb_table(jb_results, format = format)
  return(table_output)
}
```

```
# Run with helper
# missing benchmark
generate_all_jb_reports(
```

Table 11: Jarque–Bera Test Results

horizon	Formula 1	Formula 2	Formula 3	Formula 4
1	1.3015 (0.522)	1.4536 (0.483)	3.1539 (0.207)	17.8432 (0.000 ***)
2	1.5416 (0.463)	1.3227 (0.516)	8 (0.018 **)	2.9683 (0.227)
3	1.7654 (0.414)	1.6514 (0.438)	7.5554 (0.023 **)	2.5318 (0.282)
4	1.4723 (0.479)	30.1444 (0.000 ***)	2.5102 (0.285)	2.4614 (0.292)
5	1.4886 (0.475)	58.129 (0.000 ***)	2.3399 (0.310)	2.194 (0.334)
6	1.5736 (0.455)	67.3941 (0.000 ***)	1.8789 (0.391)	2.4866 (0.288)
7	0.9188 (0.632)	64.089 (0.000 ***)	1.6653 (0.435)	3.159 (0.206)
8	0.3858 (0.825)	56.1628 (0.000 ***)	1.3626 (0.506)	2.9565 (0.228)
9	0.3242 (0.850)	42.1115 (0.000 ***)	1.3005 (0.522)	3.2125 (0.201)
10	1.5089 (0.470)	30.9638 (0.000 ***)	3.1801 (0.204)	3.0005 (0.223)

Note: This test checks if the errors are normally distributed.
Jarque-Berra Test Alternative Hypothesis (H_A): * ’Errors aren’t normally distributed
† Signif. codes: ’***’ 0.01, ’**’ 0.05, ’*’ 0.1

```
  formula_cols =  formula_cols,
  eval_all_models = eval_all_models,
  max_h = max_h,
  format = format)
```

Absolute Performance: Efficiency & Bias

In order to evaluate the absolute performance of our model with the results from the pseudo out-of-sample estimation, we use Mincer-Zarnowitz regressions to check for both efficiency and bias. The method is to regress the actual realised values of the interest rate on the forecasted ones, as seen in equation (4). We then run a joint test, on the null hypothesis that $\alpha = 0$ (model is unbiased) and $\beta = 1$ (model is efficient). We run this with a custom function that loops over each horizon, in order to assess the performance of our models in the shorter vs longer term, and do so on all Taylor Rule formulas as well as on the benchmark

$$i_t^{actual} = \alpha + \beta i_t^{forecast} + \epsilon_t \tag{4}$$

```
# This function runs the Mincer-Zarnowitz regression (Actuals ~ Forecasts)
# for each horizon h and tests the joint null hypothesis H0: (alpha, beta) = (0, 1)
# and the uses the table generator helper function to make it into a table

generate_absolute_performance_report <- function(F_model,
                                   Actual_values,
                                   H,
                                   model_caption,
                                   format = "html") {
  # Pre-allocate storage for results
  mz_results <- data.frame(
    Horizon = 1:H,
    Alpha = numeric(H),
    Beta = numeric(H),
    P_Value_Joint_Test = numeric(H))

  for (h in 1:H) {
    # 1. Create a clean data frame for this horizon
    #    This pairs the forecasts and actual values and removes any NAs,
    #    ensuring they remain perfectly aligned.
    df_h <- data.frame(
      actuals = Actual_values[[h]],
      forecasts = F_model[[h]] ) %>%
      na.omit()

    # Check if we have enough data to run the regression (at least 2 obs)
    if (nrow(df_h) > 2) {
      # 2. Run MZ regression
      mz_reg <- lm(actuals ~ forecasts, data = df_h)

      # 3. Get coefficients
      coeffs <- summary(mz_reg)$coefficients
      mz_results$Alpha[h] <- coeffs[1, 1]
      mz_results$Beta[h]  <- coeffs[2, 1]

      # Using NW errors as seen in class, with lag selection h-1
      v_matrix <-
        if (h == 1) {
          # h=1: No autocorrelation, use standard "White" (HC) errors
          sandwich::vcovHC(mz_reg, type = "HC3")
        } else {
          # h>1: Use Newey-West, manually setting lag = h-1
          sandwich::NeweyWest(mz_reg, lag = h - 1)
        }

      # 4. Test Joint Hypothesis H0: Alpha = 0 AND Beta = 1 and store pvalues
      test_joint <- linearHypothesis(mz_reg,
                                     c("(Intercept) = 0", "forecasts = 1"),
                                     vcov. = v_matrix)
      mz_results$P_Value_Joint_Test[h] <- test_joint$"Pr(>F)"[2]

    } else {
      # Not enough data to run regression for this horizon
      mz_results$Alpha[h] <- NA_real_
      mz_results$Beta[h]  <- NA_real_
      mz_results$P_Value_Joint_Test[h] <- NA_real_
    }
  }

  # Generate table with helper
  table_output <- generate_absolute_performance_table(mz_results=mz_results,
                                                  model_caption=model_caption,
                                                  format=format)

  return(table_output)
}
```

```
# Call MZ-test helper function 4 times.
#  Note: These reports is wrapped in trycatch as it sometimes fails
#         If it does fail, simply decrease R in order to have more
```

```r
#          observations, removing potential multicolinearity.

# MZ Report 1: Actual Rate, No Lag
mz_report_1 <- tryCatch({
  generate_absolute_performance_report(
    F_model = F_TR_1,
    Actual_values = Actuals,
    H = H,
    model_caption = "Mincer-Zarnowitz Test: Actual Rate, No Lag",
    format = format)}, error = function(e) {
      message("Error generating MZ Report (Actual Rate, No Lag): ", e$message)
      message("Skipping this report and continuing...")
      return(NULL)})

# MZ Report 2: Shadow Rate, No Lag (
mz_report_2 <- tryCatch({
  generate_absolute_performance_report(
    F_model = F_TR_2,
    Actual_values = Actuals,
    H = H,
    model_caption = "Mincer-Zarnowitz Test: Shadow Rate, No Lag",
    format = format)}, error = function(e) {
      message("Error generating MZ Report (Shadow Rate, No Lag): ", e$message)
      message("Skipping this report and continuing...")
      return(NULL)})

# MZ Report 3: Actual Rate, with Lag
mz_report_3 <- tryCatch({
  generate_absolute_performance_report(
    F_model = F_TR_3,
    Actual_values = Actuals,
    H = H,
    model_caption = "Mincer-Zarnowitz Test: Actual Rate, with Lag",
    format = format)}, error = function(e) {
      message("Error generating MZ Report (Actual Rate, with Lag): ", e$message)
      message("Skipping this report and continuing...")
      return(NULL)})

# MZ Report 4: Shadow Rate, with Lag
mz_report_4 <- tryCatch({
  generate_absolute_performance_report(
    F_model = F_TR_4,
    Actual_values = Actuals,
    H = H,
    model_caption = "Mincer-Zarnowitz Test: Shadow Rate, with Lag",
    format = format)}, error = function(e) {
      message("Error generating MZ Report (Shadow Rate, with Lag): ", e$message)
      message("Skipping this report and continuing...")
      return(NULL)})

# MZ Report 5: Benchmark
mz_report_BM <- tryCatch({
  generate_absolute_performance_report(
```

Table 12: Mincer-Zarnowitz Test: Actual Rate, No Lag

h	Alpha	Beta	pv(Joint)
1	1.2426	0.2801	0.172
2	1.2531	0.3795	0.756
3	1.0120	0.7922	0.841
4	0.8109	1.2440	0.218
5	0.8868	1.4210	0.007 ***
6	1.0360	1.4710	0.000 ***
7	1.2366	1.3372	0.000 ***
8	1.5904	1.0953	0.000 ***
9	2.1531	0.6397	0.000 ***
10	2.8582	0.0750	0.004 ***

Note: pv(Joint) is the p-value for the joint hypothesis H_0: (Alpha, Beta) = (0, 1). A high p-value means we fail to reject the null hypothesis of an unbiased, efficient forecast. * Signif. codes: ’***’ 0.01, ’**’ 0.05, ’*’ 0.1

```
    F_model = F_BM,
    Actual_values = Actuals,
    H = H,
    model_caption = "Mincer-Zarnowitz Test: Benchmark ARIMA",
    format = format)}, error = function(e) {
      message("Error generating MZ Report (Benchmark ARIMA): ", e$message)
      message("Skipping this report and continuing...")
      return(NULL)})

list(mz_report_1, mz_report_2, mz_report_3, mz_report_4, mz_report_BM)
```

[[1]]

[[2]]

[[3]]

[[4]]

[[5]]

Relative Performance (against benchmark)

In order to evaluate the relative performance of our model against the benchmark ARIMA, we first compute the Mean Squared Forecast Errors (MSFE) for each horizon. This allows us to see the average performance of each estimation. For a deeper comparison, we implement Diebold-Mariano tests on each horizon. This test compares the forecast errors of both models by computing a loss differential, which is the difference of the squared forecast errors between the main model and the benchmark. The test statistic is as reported in equation (5) and assumes that $\widehat{d_{h,t}}$ is covariance stationary. We include three versions of the test, one where the alternative hypothesis is that the two models bring about different losses, one where the alternative hypothesis is that the tested model is greater than the benchmark, and one where the alternative hypothesis

Table 13: Mincer-Zarnowitz Test: Shadow Rate, No Lag

h	Alpha	Beta	pv(Joint)
1	1.8200	-0.3633	0.000 ***
2	1.8598	-0.2299	0.000 ***
3	1.7793	-0.0457	0.000 ***
4	1.8299	0.0095	0.000 ***
5	1.9916	-0.0130	0.000 ***
6	2.1898	-0.0409	0.000 ***
7	2.4185	-0.0665	0.000 ***
8	2.6798	-0.0894	0.000 ***
9	2.8483	-0.0617	0.000 ***
10	3.0489	-0.0389	0.000 ***

Note:

pv(Joint) is the p-value for the joint hypothesis H_0: (Alpha, Beta) = (0, 1). A high p-value means we fail to reject the null hypothesis of an unbiased, efficient forecast.

[*] Signif. codes: ’***’ 0.01, ’**’ 0.05, ’*’ 0.1

Table 14: Mincer-Zarnowitz Test: Actual Rate, with Lag

h	Alpha	Beta	pv(Joint)
1	0.0564	0.9926	0.855
2	0.2077	0.9484	0.793
3	0.4454	0.8750	0.731
4	0.7409	0.7822	0.655
5	1.1190	0.6535	0.492
6	1.4765	0.5449	0.372
7	1.8810	0.3823	0.108
8	2.2834	0.1895	0.023 **
9	2.6784	0.0002	0.001 ***
10	3.0516	-0.1807	0.000 ***

Note:

pv(Joint) is the p-value for the joint hypothesis H_0: (Alpha, Beta) = (0, 1). A high p-value means we fail to reject the null hypothesis of an unbiased, efficient forecast.

[*] Signif. codes: ’***’ 0.01, ’**’ 0.05, ’*’ 0.1

Table 15: Mincer-Zarnowitz Test: Shadow Rate, with Lag

h	Alpha	Beta	pv(Joint)
1	0.0681	0.9143	0.087 *
2	0.1930	0.7986	0.186
3	0.4392	0.6389	0.007 ***
4	0.7667	0.4865	0.000 ***
5	1.1300	0.3526	0.000 ***
6	1.4534	0.2523	0.000 ***
7	1.8018	0.1659	0.000 ***
8	2.1810	0.0875	0.000 ***
9	2.6095	0.0196	0.000 ***
10	3.0870	-0.0420	0.000 ***

Note:

pv(Joint) is the p-value for the joint hypothesis H_0: (Alpha, Beta) = (0, 1). A high p-value means we fail to reject the null hypothesis of an unbiased, efficient forecast.

[*] Signif. codes: '***' 0.01, '**' 0.05, '*' 0.1

Table 16: Mincer-Zarnowitz Test: Benchmark ARIMA

h	Alpha	Beta	pv(Joint)
1	0.1487	0.9447	0.382
2	0.3943	0.8503	0.193
3	0.7096	0.7125	0.197
4	1.0656	0.5675	0.290
5	1.4463	0.4072	0.146
6	1.8131	0.2622	0.144
7	2.1639	0.1192	0.006 ***
8	2.4781	-0.0143	0.001 ***
9	2.7540	-0.1508	0.000 ***
10	2.9915	-0.2735	0.000 ***

Note:

pv(Joint) is the p-value for the joint hypothesis H_0: (Alpha, Beta) = (0, 1). A high p-value means we fail to reject the null hypothesis of an unbiased, efficient forecast.

[*] Signif. codes: '***' 0.01, '**' 0.05, '*' 0.1

is that the tested model is worse than the benchmark (greater loss). The null hypothesis is the same for all three, being that the two models bring about the same loss.

$$\begin{aligned} DM &= \frac{\bar{\hat{d}}_h}{\hat{\sigma}^2_{\bar{\hat{d}}_h}} \\ &= \frac{\frac{1}{P}\sum_{t=R}^{T-h} \hat{d}_{h,t}}{\hat{\sigma}^2_{\bar{\hat{d}}_h}} \\ \text{where } \widehat{d_{h,t}} &= \widehat{e^2_{1,t+h|t}} - \widehat{e^2_{2,t+h|t}} \end{aligned} \tag{5}$$

```r
# This function runs Diebold-Mariano tests and automatically turns the output
#  into a kable table

generate_relative_performance_report <- function(FE_TR_model,
                                                 FE_BM_model,
                                                 H,
                                                 model_caption,
                                                 format = "html") {
  # Preallocate storage for MSFEs and output data
  MSFE_TR = numeric(H)
  MSFE_BM = numeric(H)
  dm_results <- data.frame(
    Horizon = 1:H,
    MSFE_TR = MSFE_TR,
    MSFE_BM = MSFE_BM,
    Ratio_TR_vs_BM = numeric(H))

  # Calculate MSFEs
  for (h in 1:H) {
    # Ensure errors are cleaned of NAs
    fe1 <- na.omit(FE_TR_model[[h]])
    fe2 <- na.omit(FE_BM_model[[h]])

    MSFE_TR[h] = mean((fe1)^2)
    MSFE_BM[h] = mean((fe2)^2)
  }

  # Run DM Tests
  DMpvalues = matrix(, nrow = H, ncol = 3)
  colnames(DMpvalues) <- c("DM_Two_Sided", "DM_Greater", "DM_Lesser")
  for (h in 1:H){
    # Note: dm.test needs the full (un-omitted) error vectors
    #       to align them properly, hence using the original list inputs
    x1 = dm_test(e1 = FE_BM_model[[h]], e2 = FE_TR_model[[h]], h = h)
    x2 = dm_test(e1 = FE_BM_model[[h]], e2 = FE_TR_model[[h]], h = h,
                 alternative = "greater")
    x3 = dm_test(e1 = FE_BM_model[[h]], e2 = FE_TR_model[[h]], h = h,
                 alternative = "less")
    DMpvalues[h, 1] = round(x1$p.value, digits = 4)
    DMpvalues[h, 2] = round(x2$p.value, digits = 4)
    DMpvalues[h, 3] = round(x3$p.value, digits = 4)
  }
  # Put results in a dataframe
  dm_results$MSFE_TR = MSFE_TR
  dm_results$MSFE_BM = MSFE_BM
  dm_results$Ratio_TR_vs_BM = MSFE_TR / MSFE_BM
  dm_results <- bind_cols(dm_results, as.data.frame(DMpvalues))

  # Generate table with helper
  table_output <- generate_relative_performance_table(dm_results=dm_results,
                                                     model_caption=model_caption,
                                                     format=format)
  return(table_output)
}
```

```r
# Call DM-test helper function 4 times.

# Report 1: Actual Rate, No Lag
report_1 <- generate_relative_performance_report(
  FE_TR_model = FE_TR_1,
  FE_BM_model = FE_BM,
  H = H,
  model_caption = "MSFE Comparison, Trained on Actual Rate, No Lag",
  format = format)

# Report 2: Shadow Rate, No Lag
report_2 <- generate_relative_performance_report(
  FE_TR_model = FE_TR_2,
  FE_BM_model = FE_BM,
  H = H,
  model_caption = "MSFE Comparison, Trained on Shadow Rate, No Lag",
  format = format)

# Report 3: Actual Rate, with Lag
report_3 <- generate_relative_performance_report(
  FE_TR_model = FE_TR_3,
  FE_BM_model = FE_BM,
  H = H,
  model_caption = "MSFE Comparison, Trained on Actual Rate, with Lag",
  format = format)

# Report 4: Shadow Rate, with Lag
report_4 <- generate_relative_performance_report(
  FE_TR_model = FE_TR_4,
  FE_BM_model = FE_BM,
  H = H,
  model_caption = "MSFE Comparison, Trained on Shadow Rate, with Lag",
  format = format)

list(report_1, report_2, report_3, report_4)
```

```
[[1]]

[[2]]

[[3]]
```

Table 17: MSFE Comparison, Trained on Actual Rate, No Lag

h	MSFE TR	MSFE BM	Ratio	DM Two-Sided	DM Greater	DM Lesser
1	4.0170	0.1591	25.2500	0.000 ***	1.000	0.000 ***
2	3.9732	0.6458	6.1521	0.032 **	0.984	0.016 **
3	3.4813	1.6062	2.1673	0.377	0.811	0.188
4	3.1086	2.8224	1.1014	0.915	0.542	0.458
5	3.0938	4.3368	0.7134	0.603	0.302	0.698
6	3.2794	5.8787	0.5578	0.124	0.062 *	0.938
7	3.5234	7.4570	0.4725	0.000 ***	0.000 ***	1.000
8	4.0667	9.0333	0.4502	0.001 ***	0.000 ***	1.000
9	4.8839	10.1786	0.4798	0.000 ***	0.000 ***	1.000
10	5.8798	11.5721	0.5081	0.000 ***	0.000 ***	1.000

Note: TR refers to the forecast made with an estimated Taylor Rule. BM refers to a benchmark of the interest rate using an ARIMA model. Ratio < 1 indicates that the TR model has lower MSFE.
DM Test Alternative Hypotheses (H_A): * 'DM Greater' tests if the TR model is significantly more accurate than the BM model. † 'DM Lesser' tests if the TR model is significantly less accurate than the BM model.

Table 18: MSFE Comparison, Trained on Shadow Rate, No Lag

h	MSFE TR	MSFE BM	Ratio	DM Two-Sided	DM Greater	DM Lesser
1	8.7557	0.1591	55.0357	0.000 ***	1.000	0.000 ***
2	9.4762	0.6458	14.6728	0.002 ***	0.999	0.001 ***
3	9.4969	1.6062	5.9125	0.030 **	0.985	0.015 **
4	12.0493	2.8224	4.2692	0.209	0.895	0.105
5	16.4410	4.3368	3.7910	0.313	0.844	0.156
6	22.9412	5.8787	3.9024	0.363	0.819	0.182
7	30.3125	7.4570	4.0650	0.384	0.808	0.192
8	40.6792	9.0333	4.5032	0.336	0.832	0.168
9	48.6339	10.1786	4.7781	0.256	0.872	0.128
10	59.1154	11.5721	5.1084	0.144	0.928	0.072 *

Note: TR refers to the forecast made with an estimated Taylor Rule. BM refers to a benchmark of the interest rate using an ARIMA model. Ratio < 1 indicates that the TR model has lower MSFE.
DM Test Alternative Hypotheses (H_A): * 'DM Greater' tests if the TR model is significantly more accurate than the BM model. † 'DM Lesser' tests if the TR model is significantly less accurate than the BM model.

Table 19: MSFE Comparison, Trained on Actual Rate, with Lag

h	MSFE TR	MSFE BM	Ratio	DM Two-Sided	DM Greater	DM Lesser
1	0.1364	0.1591	0.8571	0.540	0.270	0.730
2	0.4435	0.6458	0.6866	0.196	0.098 *	0.902
3	0.9719	1.6062	0.6051	0.136	0.068 *	0.932
4	1.6217	2.8224	0.5746	0.113	0.056 *	0.944
5	2.5347	4.3368	0.5845	0.037 **	0.018 **	0.982
6	3.4632	5.8787	0.5891	0.008 ***	0.004 ***	0.996
7	4.7305	7.4570	0.6344	0.009 ***	0.004 ***	0.996
8	6.2083	9.0333	0.6873	0.002 ***	0.001 ***	0.999
9	7.9732	10.1786	0.7833	0.013 **	0.006 ***	0.994
10	9.7356	11.5721	0.8413	0.080 *	0.040 **	0.960

Note: TR refers to the forecast made with an estimated Taylor Rule. BM refers to a benchmark of the interest rate using an ARIMA model. Ratio < 1 indicates that the TR model has lower MSFE.

DM Test Alternative Hypotheses (H_A): * 'DM Greater' tests if the TR model is significantly more accurate than the BM model.

† 'DM Lesser' tests if the TR model is significantly less accurate than the BM model.

[[4]]

Table 20: MSFE Comparison, Trained on Shadow Rate, with Lag

h	MSFE TR	MSFE BM	Ratio	DM Two-Sided	DM Greater	DM Lesser
1	0.1591	0.1591	1.0000	1.000	0.500	0.500
2	0.5536	0.6458	0.8571	0.715	0.357	0.643
3	1.5969	1.6062	0.9942	0.946	0.473	0.527
4	3.5526	2.8224	1.2587	0.400	0.800	0.200
5	6.7917	4.3368	1.5661	0.230	0.885	0.115
6	11.0846	5.8787	1.8856	0.076 *	0.962	0.038 **
7	17.4023	7.4570	2.3337	0.079 *	0.960	0.040 **
8	26.3667	9.0333	2.9188	0.100	0.950	0.050 *
9	37.3973	10.1786	3.6741	0.096 *	0.952	0.048 **
10	51.5048	11.5721	4.4508	0.056 *	0.972	0.028 **

Note: TR refers to the forecast made with an estimated Taylor Rule. BM refers to a benchmark of the interest rate using an ARIMA model. Ratio < 1 indicates that the TR model has lower MSFE.

DM Test Alternative Hypotheses (H_A): * 'DM Greater' tests if the TR model is significantly more accurate than the BM model.

† 'DM Lesser' tests if the TR model is significantly less accurate than the BM model.

Actual Forecast Model

As described earlier, our methodology consists of two steps where we first fit ARIMA models to the Taylor Rule inputs as well as estimate the coefficients, and then use the forecasted inputs and the coefficients to compute our point forecast for the interest rate. In this section however, we run this method on the entire sample (Q1 1999 to Q3 2025) making forecasts up to 10 quarters ahead (Q1 2028).

Forecasting

In order to do this, we use a custom function that first fits the ARIMA models for the inputs, uses these to make forecasts of said inputs, estimates the Taylor Rule coefficients and then computes the point forecast for the interest rate. Note that for the lagged models, we then use these point forecasts to compute the next forecast iteratively. It also fits an ARIMA model for the interest rate, so that we have a simple benchmark model to see if there are large differences in estimates.

```r
#--------------------------------------------------------------------------------
#-----------------                     1                        -----------------
#--------------------------------------------------------------------------------

# Function to make our actual forecast, including forecasts of our inputs,
#  and also outputting the ARIMA coefficients and models used

our_predict <- function(data,formula,H){

  # --- 1. Fit inputs and benchmark models  ---
  inflation_arma <- my.auto.arima(data$inflation_gap, max.p=4, max.q=4, d=1, var_name="Inflation Gap")
  outputgap_arma <- my.auto.arima(data$output_gap, max.p=4, max.q=4, d=0, var_name="Output Gap")
  interest_arma <- my.auto.arima(data$rate, max.p=4, max.q=4, d=1, var_name="Interest Rate")

  # --- 2. Get forecasts of inputs (all H horizons) ---
  inflation_forecasts <- my.forecast(inflation_arma, h = H)
  outputgap_forecasts <- my.forecast(outputgap_arma, h = H)
  BMpredicted_rates <- my.forecast(interest_arma, h = H)

  # --- 3. Fit TR model
  TR_model <- lm(formula, data = data)

  # --- 4. Build forecast input data frame (iteratively for lags) ---

  # Allocate storage for full horizon
  TR_preds <- numeric(H)

  # Get last known lags (starting point for lagged models)
  current_shadowrate_lag <- last(data$shadowrate)
  current_rate_lag <- last(data$rate)

  for (h in 1:H) {
    new_data_h <- data.frame(
      inflation_gap = inflation_forecasts[h],
      output_gap = outputgap_forecasts[h],
      shadowrate_lag = current_shadowrate_lag,
      rate_lag = current_rate_lag)
    # Get forecasted values
    pred_h <- predict(TR_model, new_data_h)
    TR_preds[h] <- round(pmax(pred_h, min(data$rate)) / 0.25) * 0.25

    # Update the lag for h+1
    current_rate_lag <- pred_h }

  # --- 5. Compute forecast for BM ---
  BM_preds <- round(pmax(BMpredicted_rates, min(data$rate)) / 0.25) * 0.25

  return(list(TR_Forecast = TR_preds,
              BM_Forecast = BM_preds,
              Inflation_Forecast = inflation_forecasts + 2, #to add back target
              OutputGap_Forecast = outputgap_forecasts,
              inflation_gap_arma_coef = inflation_arma$coef,
              output_gap_arma_coef = outputgap_arma$coef,
              benchmark_arma_coef = interest_arma$coef,
              inflation_gap_arma_var = inflation_arma$sigma2,
              output_gap_arma_var = outputgap_arma$sigma2,
              benchmark_arma_var = interest_arma$sigma2))}
```

```r
# --------- 1. Compute forecast ---------

# Uses our helper function
final_forecasts <- our_predict(data = data, formula = model_formula, H = H)
```

The ARIMA model fitted to the ‘Inflation Gap’ variable is specified as ARIMA(1, 1, 4).

The ARIMA model fitted to the ‘Output Gap’ variable is specified as ARIMA(2, 0, 3).

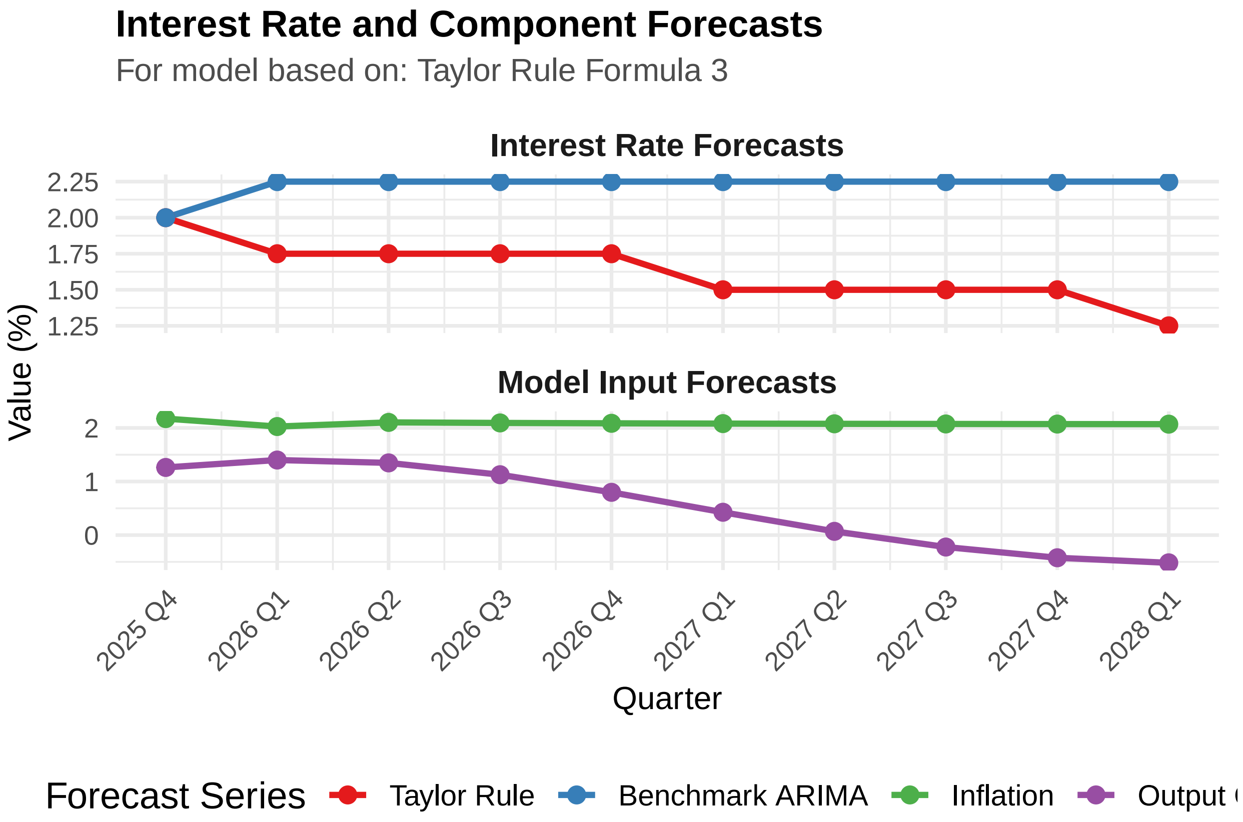
The ARIMA model fitted to the ‘Interest Rate’ variable is specified as ARIMA(2, 1, 0).

```r
# --------- 2. Table ---------

# Uses a helper to display the results in a table
display_forecasts(final_forecasts,
                  caption = paste("For model based on:", model_name),
                  format = format)
```

```r
# --------- 3. Plot ---------

# Uses a helper to display the results in a plot
plot_forecasts(final_forecasts)
```

Table 21: For model based on: Taylor Rule Formula 3

Horizon: Quarter	Taylor Rule Forecast	Benchmark Forecast	Inflation Forecast	Output Gap Forecast
1: 2025 Q4	2.00	2.00	2.17	1.26
2: 2026 Q1	1.75	2.25	2.03	1.40
3: 2026 Q2	1.75	2.25	2.10	1.35
4: 2026 Q3	1.75	2.25	2.09	1.13
5: 2026 Q4	1.75	2.25	2.09	0.80
6: 2027 Q1	1.50	2.25	2.08	0.43
7: 2027 Q2	1.50	2.25	2.08	0.07
8: 2027 Q3	1.50	2.25	2.07	-0.22
9: 2027 Q4	1.50	2.25	2.07	-0.42
10: 2028 Q1	1.25	2.25	2.07	-0.52



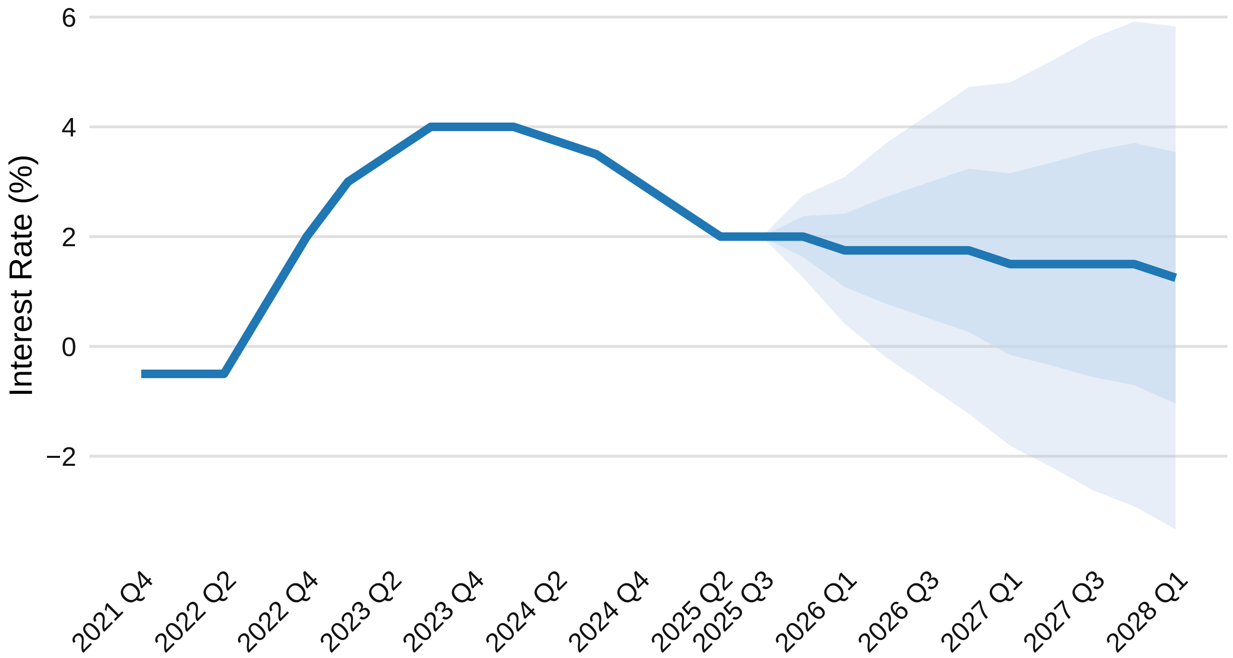
Prediction Intervals

To compute the prediction intervals for our point forecasts, we use the estimated variance of errors computed in the pseudo out-of-sample evaluation exercise assuming that the distribution of errors will be the same for our actual future forecast. We include two intervals, one being one standard deviation away from the point forecasts, and the other being two standard deviations away. This is computed as in equation (6).

$$
\begin{aligned}
PI_1^{upper} &= i_t^{forecast} + \sigma_e \\
PI_1^{lower} &= i_t^{forecast} - \sigma_e \\
PI_2^{upper} &= i_t^{forecast} + 2\sigma_e \\
PI_2^{lower} &= i_t^{forecast} - 2\sigma_e
\end{aligned}
\tag{6}
$$

```r
# Function to compute the prediction intervals based on the actual forecasted
#  values computed from the previous function, and the forecast error variance
#   estimated in the pseudo out-of-sample evaluation exercise

our_predict_intervals <- function(estimated_variance, forecast) {

  # Preparing data with point forecasts and variance
  prediction <- estimated_variance
  final_interval <- forecast[c(1,2,3,4)]
  prediction$sd_fe <- sqrt(prediction$var_fe)

  # Computing confidence intervals
  final_interval$sd <- prediction$sd_fe
  final_interval$upper_1_sd <- final_interval$sd + final_interval$TR_Forecast
  final_interval$lower_1_sd <- final_interval$sd*(-1) + final_interval$TR_Forecast

  final_interval$upper_2_sd <- final_interval$sd*2 + final_interval$TR_Forecast
  final_interval$lower_2_sd <- final_interval$sd*2*(-1) + final_interval$TR_Forecast

  final_interval <- as.data.frame(final_interval)

  return(final_interval)
}
```

```r
# --------- 4. Compute Prediction Intervals ---------

final_interval <- our_predict_intervals(estimated_variance = var_by_horizon,
                                        forecast = final_forecasts)

# --------- 5. Plots Prediction Intervals ---------

plot_forecasts_pred_int(data, intervals = final_interval)
```



ECB Deposit Facility Rate Forecast

Model: Taylor Rule Formula 3

Shaded areas: ±1 S.D. (dark) / ±2 S.D. (light)