## Statkraft Case Study Task

## TTF Futures Background

In gas trading, we commonly trade and analyse TTF futures. TTF (Title Transfer Facility) is Europe’s leading gas benchmark, representing the price of natural gas traded in the Dutch virtual hub. Futures contracts exist for various delivery months in the future, each with two key dates: delivery and expiry. Delivery marks when the contract must be settled (physically or financially), while expiry is the last day it can be traded.  
  
The various futures contracts that make up the forward curve for TTF can be referred to in the following ways (each has its own purpose):

1. **Specific**: Uses a product symbol followed by a letter representing the month and last two digits of the year. For example, "TFM\H25" represents the TTF contract for March 2025. The month codes are: F (January), G (February), H (March), J (April), K (May), M (June), N (July), Q (August), U (September), V (October), X (November), Z (December).
2. **Generic**: Uses a product symbol followed by a number indicating the sequence. For example, "TFM1" refers to the first active TTF contract as of a specific date. “TFMn” refers to the n-th closest TTF contract. As of 05/03/2025, TFM1 refers to the April 2025 contract, or TFM\J25. TFM2 refers to the May 2025 contract (TFM\K25) etc. When TFM\J25 expires (i.e. it **cannot** be traded) then the TFM\K25 contract becomes TFM1 etc. Different products can have different rules about what the closest active delivery is and when expiry takes place.
3. **Monthly Generic**: Refers to the first active TTF contract for a specific month. For example, "TFMAPR1" refers to the first active April TTF contract. As of 05/03/2025, TFMAPR1 refers to the April contract, or TFM\J25. Expiry and the roll over from one contract to the next works the same as the mechanism described for generics: TFMAPR2 refers to TFM\J26, and this becomes TFMAPR1 after TFM\J25 expires.

On a trading desk, it is important to quickly pull in data for various products in different notations. To do this efficiently and intuitively, we need a good data infrastructure. One common solution for this is only storing the specifics and building an API that allows for querying the specifics as generics or monthly generics. In Part 1 of this case study, you will implement a simplified version of this interface for outright futures contracts. In Part 2, you will utilise this interface to implement an application to visualise intraday price data.

## Part 1

**Objective**

You must create a Python module capable of constructing and querying a data structure that maps any one of the security types (contract naming conventions) to the remaining two for any point in time or point in time range within the date range of the provided expiry calendars.

**Data**

The provided *ttf\_calendar.csv* file contains all data that is needed for this part.

**Requirements**

* **Input**: The script must accept an expiry calendar. This calendar includes the delivery month and the expiry month for each specific contract. The provided file *ttf\_calendar.csv* contains the data for TTF contracts, but the code should support any given expiry calendar in this format. Specific contracts are always denoted in the format: [product\_code]\[1 letter][2 digits].
* **Data Structure**: Design a data structure that can be serialized for disk storage and efficiently accessed in memory (millisecond order of magnitude query time).
* **Query Interface**: Develop functionality that allows querying this data structure with 3 parameters:
  + security (e.g., "TFM1", " TFM\J25", " TFMAPR1")
  + security type (specific, generic, monthly generic)
  + point-in-time date (range)
* **Scalability**: Think about which components we might need to scale and make sure that your solution allows for this. There are plenty of solutions to the problem that work in our example case, but that might not pass in practice.

**Deliverables**

* **.py script:** That generates the data structure and serialises it and writes to disk.
* **Serialised data structure**
* **.py script:** That loads the data structure from disk and allows for querying with the parameters provided in the requirements section.
* **requirements.txt**
* **Instructions on running**

**!!!!Bonus Points, attempt after finishing part 2!!!!**

In addition to outright futures, spreads are also traded on the gas trading desk. A spread is the difference between two or more outright contracts. Examples include the DEC-JUN spread and the DEC-DEC spread. There is a naming convention to convert these into generics. For instance, TFMDECJUN1 refers to the first DEC contract, minus the first JUN contract that follows this specific DEC contract (not just the next JUN contract). To complicate things further and prove the above statement that these are not monthly generics: This is written as TFMDEC1JUN2 as a monthly generic on 05/03/2025, but as TFMDEC1JUN1 between the expiry of TFM\M25 and the expiry of TFM\Z25. TFMDECDEC1 refers to the first DEC minus the next DEC after. The monthly generic version of TFMDECDEC1 will always be TFMDEC1DEC2.

Can you add a module to the data structure generator that also adds these time spreads? The input for this would be a list of tuples where each tuple contains the two months to make the spread with. You only need to provide the first active spread for each tuple. It is fine to test the code with just DEC-JUN and DEC-DEC, but make sure that your solution scales to providing all month combinations as well.

## Part 2

**Objective**

You are now required to use the data structure created in Part 1 within a Dash application to visualise intraday price movements. The application should include at least one page displaying an intraday plot for a specified date range. The plot below displays the intraday price change of a given contract from the previous close for the period from 14/01/2025 to 21/01/2025 (i.e., the given date is 21/01/2025 and includes the 5 days prior).

A graph with colorful lines

Description automatically generated

For illustrative purposes only.

**Data**

Data is provided in intraday bars in the *30\_min\_bars.csv* file. This file contains OHLC prices, and volume for 30-minute bars from 07:00 to 17:00 London time. The timestamp on each bar indicates the end time of the bar. The OPEN column of a row with timestamp 17:00 refers to the price of first trade after 16:30 and the CLOSE column of the same row refers to the price of the last trade before 17:00. Assume that the exchange opens at 07:00 and closes at 17:00.

The data is provided for specific contracts, but the plot must support input options for specific, generic, and monthly generic contracts. This relies on the data structure from Part 1. Note: When comparing futures prices across multiple days, it is essential to use the same specific contracts. The intraday plot should load data for the specific contract corresponding to a (monthly) generic on the point in time date, for all other days in the plot.

**Requirements**

* **Visualization & front-end:** An intraday price visualization page with input options for selecting specific, generic, and monthly generic contracts, as well as a date and a number of days prior to that date.
* **Deployment:** We must be able to run the dash app on a localhost server.
* **Reproducibility:** We must be able to delete the data structure from part 1 and regenerate it with your provided code and re-deploy the dash app without problems.

**Deliverables**

* **.py script:** That runs the dash app on a localhost server.
* **requirements.txt.**
* **Instruction on running.**

**!!!!More Bonus Points, only if you have finished everything else!!!!**

There are many other visualizations that could enhance a trader’s daily workflow. For example, intraday cumulative volume vs. historical average volume, z-scores of price moves for the front contract (which would require an additional API component to load "stitched together" specific contracts as the first active), cumulative price movements within a time window over the past few days or on specific days of the week, and more. Feel free to add any visualisations or models that could assist a trader. If a longer or more granular data sample is needed, let us know.

## Tips

* Parts 1 and 2 have the highest priority before the bonus points of each, If you build all the infrastructure properly, you should be able to integrate the updated data structure with the bonus logic seamlessly. If you finish everything, you can add more visualisations, modelling, or other fun stuff you can come up with for the bonus points in part 2.
* Make sure that all code is clear and modular. Write comments where code is ambiguous only but try and avoid ambiguous code. Include docstrings for public methods and functions.
* If you are new to dash, start simple and build and test page components one by one. It is simple but perhaps a bit fiddly at the start. Use [Dash in 20 Minutes Tutorial | Dash for Python Documentation | Plotly](https://dash.plotly.com/tutorial) for a brief introduction and [A Minimal Dash App | Dash for Python Documentation | Plotly](https://dash.plotly.com/minimal-app) can be used as a starting point for your app.
* For now, the focus should be on functionality rather than aesthetics of the app. A solution that addresses the bonus points is preferred over front-end design.

**Logistics**

* Please prepare a small presentation explaining the design, further models and if you have done any bonus work.
* Please submit your entire solution as a .zip file at least 3 hours prior to the meeting.