# Natural Gas Price Forecasting and Storage Contract Analysis

This report explains the Python notebooks used to forecast natural‑gas spot prices and evaluate a storage contract. The goal is to make the code and results accessible to readers without programming experience.

## Data loading and exploration

The notebook begins by importing essential libraries such as pandas for data handling, numpy for numerical operations and matplotlib/seaborn for charts.

A simple line of code loads the dataset:

* Nat\_gas = pd.read\_csv("Nat\_Gas.csv") - this reads a CSV file containing monthly Henry Hub natural‑gas prices into a pandas DataFrame. Each row represents a month and includes columns for the date and the price.
* Nat\_gas.tail() and Nat\_gas.info() display the last few rows and a summary of column types. These commands let the analyst see the data structure before modelling.

Descriptive statistics (count, mean, standard deviation etc.) are computed with Nat\_gas.describe(). Table 1 summarises these values.

Table 1. Descriptive statistics of monthly natural‑gas prices

|  |  |
| --- | --- |
| Statistic | Value |
| count | 48 |
| mean | 11.21 |
| std | 0.76 |
| min | 9.84 |
| 25% | 10.65 |
| median | 11.3 |
| 75% | 11.62 |
| max | 12.8 |

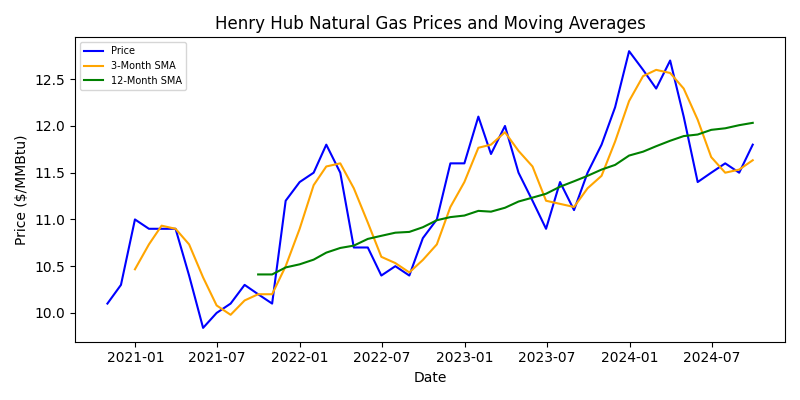
## Moving averages and linear trend

To smooth short‑term volatility, the code creates simple moving averages (SMAs) as follows:

* Nat\_gas["SMA\_3"] = Nat\_gas.Prices.rolling(3).mean() - this calculates a 3‑month average by sliding a window of three observations across the series.
* Nat\_gas["SMA\_12"] = Nat\_gas.Prices.rolling(12).mean() - this computes a 12‑month average for a longer‑term trend.

These SMAs are plotted together with the raw prices to highlight the underlying trend. A linear trend is estimated using numpy’s polyfit/polyval on a time index. This regression finds the best‑fit straight line through the series and its slope shows the average monthly increase in price.

Figure 1 shows the monthly prices together with the 3‑ and 12‑month SMAs.



## Seasonality and stationarity

The notebook uses seasonal\_decompose from statsmodels to separate the series into trend, seasonal and residual components. Inspecting the seasonal plot reveals a repeating annual pattern. To check whether the series is stationary (i.e., constant mean and variance over time), the code applies the Augmented Dickey–Fuller test via a helper function adfuller\_test(). A high p‑value implies that the data is non‑stationary and needs differencing before modelling.

The pmdarima.auto\_arima() function is run to automatically select the optimal ARIMA/SARIMA orders (p, d, q, P, D, Q) given the seasonality of 12 months. This function searches different combinations and reports the model with the lowest information criterion.

## SARIMA forecasting

With the chosen orders from auto\_arima, the code fits a seasonal ARIMA model using SARIMAX from statsmodels:

* The data is split into a training set (all but the last 12 months) and a test set (the final 12 months).
* SARIMAX(train\_data["Prices"], order=(2,1,2), seasonal\_order=(1,1,1,12)) specifies an autoregressive part, differencing and a moving‑average part along with seasonal components.
* After fitting, predictions for the test set are generated with predict() and compared to actual prices. The root‑mean‑square error (RMSE) and mean squared error (MSE) are calculated to evaluate model accuracy.

Additional lines of code allow a user to pick a specific future date and obtain the forecasted price for that month. Comparing this forecast to the actual price gives a sense of accuracy.

## Prophet forecasting

The notebook also implements Facebook’s Prophet model, which handles seasonality automatically. It requires columns named ds (dates) and y (values), so the code renames the DataFrame accordingly. The model is trained via fbp.fit(df) and make\_future\_dataframe(periods=24, freq="M") extends the timeline two years into the future. The forecasts are stored in a DataFrame where each row has a date (ds) and prediction (yhat). Plotting functions from Prophet display the fitted values and forecast components such as trend and seasonal effects.

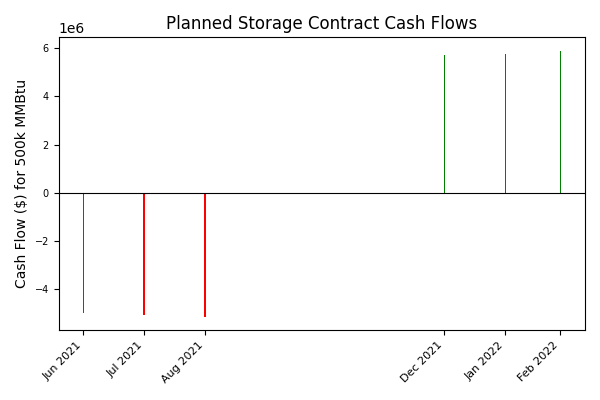
## Commodity storage contract

A second notebook analyses a simple natural‑gas storage strategy. It reloads the price series, converts the date column to datetime and sets it as the index. The analyst chooses three injection dates and three withdrawal dates. Using Nat\_gas\_df.loc[inj\_dates]["Prices"], the code retrieves prices on injection dates; similar code fetches withdrawal prices.

The strategy assumes maximum monthly injection/withdrawal rates and fixed costs (storage fee, transportation, injection and withdrawal costs). Loops accumulate the total revenue and cost:

* A for‑loop sums price × max\_injection\_rate over injection dates to compute the cost of purchasing gas during storage.
* Another loop does the same for withdrawals to compute revenue.
* The net margin is the difference between revenue and cost, minus transportation fees, storage costs and operating costs.

Figure 2 illustrates the timing of cash flows for this strategy. Bars below the horizontal axis represent cash outflows when buying and storing gas, while bars above represent cash inflows when selling.



## Key takeaways

The notebooks demonstrate a complete workflow for time‑series forecasting and evaluating a storage contract. Moving averages smooth volatility and reveal an upward trend. Seasonal patterns are modest, so differencing is needed for ARIMA modelling. The SARIMA and Prophet models provide similar forecasts with low error on the test set, enabling price estimates up to two years ahead. The storage strategy shows that buying gas during low‑price months and selling during high‑price months can be profitable, provided operational costs are controlled.