Welcome to Safran Lab 2

Every day, more than 80,000 commercial flights take place around the world, operated by hundreds of airlines. For all aircraft take-off weight exceeding 27 tons, a regulatory constraint requires companies to systematically record and analyse all flight data, for the purpose of improving the safety of flights. Flight Data Monitoring strives to detect and prioritize deviations from standards set by the aircraft manufacturers, the authorities of civil aviation in the country, or even companies themselves. Such deviations, called events, are used to populate a database that enables companies to identify and monitor the risks inherent to these operations.

This notebook is designed to let you manipulate real aeronautical data, provided by the Safran Group. It is divided in two parts: the first part deals with data exploration, data visualization, the use case, analysis of distributions and simple linear models to predict the fuel consumption of a flight. The second part deals with more complex models, optimization of parameters, interpretation of the results, and models to predict the fuel consumption of each flight phase and finally conclude by the objective of giving pieces of advice to the pilot so that he optimizes the consumption of fuel the next time.

Load the cell below for overall set up

In [2]:
```
# set up
BASE_DIR = "/mnt/safran/TP2/data/"

import time

import glob

import matplotlib as mpl
import matplotlib.pylab as plt
%matplotlib inline
import matplotlib.dates as mdates
mpl.rcParams['axes.grid'] = True

import numpy as np
import scipy as sp
import pandas as pd

pd.options.display.max_columns = 23

from datetime import datetime

from sklearn import tree, linear_model, model_selection, preprocessing,
decomposition
from sklearn.metrics import mean_squared_error


# read_pickle
dfs = []
files = glob.glob(BASE_DIR + "flights/*.pkl")

p = 0
```

```
for idx, file in enumerate(files):
    if idx % int(len(files) / 10) == 0:
        print(str(p * 10) + "%: [" + "#" * p + " " * (10 - p) + "]", end
="\r")
        p += 1
    dfs.append(pd.read_pickle(file))


# load global values data, Summarising global values in a dataframe

file_features_df = BASE_DIR + "features_df.pkl"
file_output_df = BASE_DIR + "output_df.pkl"

features_df = pd.read_pickle(file_features_df)
output_df = pd.read_pickle(file_output_df)

# flight_phases

flight_phases = ['APPROACH', 'CLIMB', 'CRUISE', 'DESCENT', 'ENG START',
'FINAL APP',
                 'FLARE', 'INIT CLIMB', 'LANDING', 'TAKE OFF', 'TAXI IN'
, 'TAXI OUT', 'TOUCH N GO',
                 'LVL CHANGE', 'GO AROUND']
```

```
100%: [##########]
```

1 Know & understand the data

Context

You are provided with nearly 3000 flights operating different routes.

Each flight data is a collection of time series resumed in a dataframe, the sample rate is 1Hz, the columns variables are described in the schema below:

Schema

VAR	DESCRIPTION	UNIT
ORIGIN	Flight departure airport	NA
RUNWAY_TO	Flight origin runway	NA
DESTINATION	Flight arrival airport	NA
RUNWAY_LD	Flight destination runway	NA
DATE_HF	High rate date computation	%d/%m/%y (UTC)
TIME_HF	High rate time computation	%H:%M:%S (UTC)
FLIGHT_PHASE	Current flight-phase	NA
ALT_STD_C	Standard altitude corrected	feet
HEAD_MAG	Magnetic heading	deg
IAS_C	Indicated air speed corrected	knot
RALTC	Radio altitude computed from different sources	feet

PITCH_C	Pitch attitude corrected	deg
ROLL_C	Bank angle corrected	deg
FOB	Fuel on board	kg
TORQ1_C	Torque corrected (engine 1)	%
TORQ2_C	Torque corrected (engine 2)	%
NH1_C	NH corrected (engine 1)	%
NH2_C	NH corrected (engine 2)	%
NL1	NL Left from frequency input	%
NL2	NL Right from frequency input (NL2)	%
GW_C	Gross weight corrected	kg

Here are some links to get expertise on some variables:

- torque
- aicraft fuel consumption
- about FOB and fuel flow

Question 1.1

- What variables are categorical?
- What variables are numerical?

In [6]:
```
df.describe()
```

Out[6]:

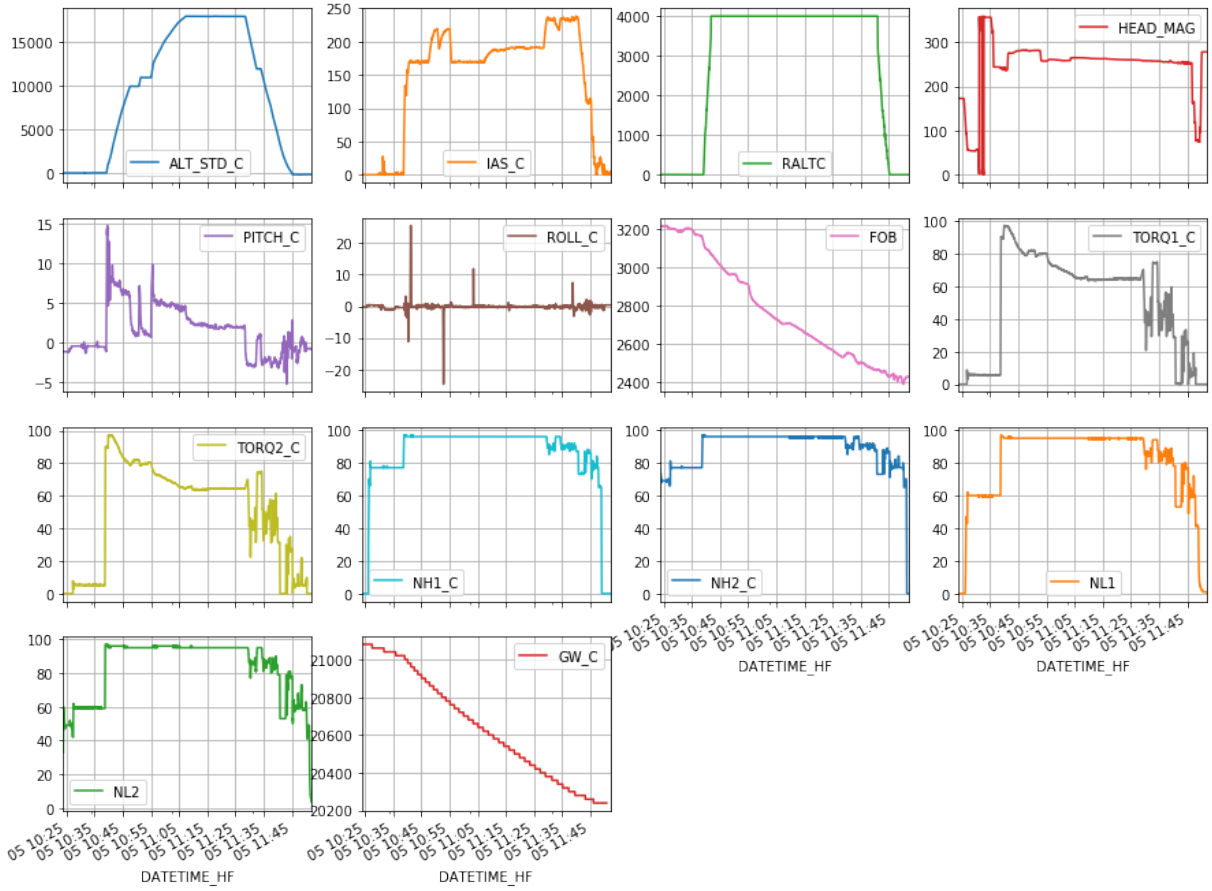
	ALT_STD_C	IAS_C	RALTC	HEAD_MAG	PITCH_C	ROLL_C	
count	5300.000000	5300.000000	5300.000000	5300.000000	5300.000000	5300.00000	530
mean	9380.041698	143.439434	2805.375472	244.704509	1.589113	-0.12283	27
std	7313.233291	83.138209	1767.863233	63.837697	3.019048	1.62885	265
min	-169.000000	0.000000	-4.300000	0.000000	-5.200000	-24.70000	239
25%	42.750000	104.500000	0.000000	253.900000	-0.600000	-0.30000	254
50%	10406.000000	170.000000	4000.000000	259.800000	1.700000	-0.10000	270
75%	17722.250000	191.000000	4000.000000	264.500000	3.200000	0.10000	299
max	17962.000000	238.000000	4000.000000	358.900000	14.800000	25.60000	321

COMMENT
- *CATEGORICAL*: ORIGIN, RUNWAY_TO, DESTINATION, RUNWAY_LD, FLIGHT_PHASE, DATE_HF, TIME_HF
- *NUMBERICAL*: , ALT_STD_C, HEAD_MAG, IAS_C, RALTC, PITCH_C, ROLL_C, FOB, TORQ1_C, TORQ2_C, NH1_C, NH2_C, NL1, NL2, GW_C

Visualize all time series for one flight

Let's visualize all the numerical time series for one flight, for example `dfs[0]`, to have a sense of how the time series vary.

In [4]:
```
# Give an alias to dfs[0] for convenience
df = dfs[0]

df.plot(kind="line", subplots=True, layout=(4, 4), figsize=(15, 12));
```

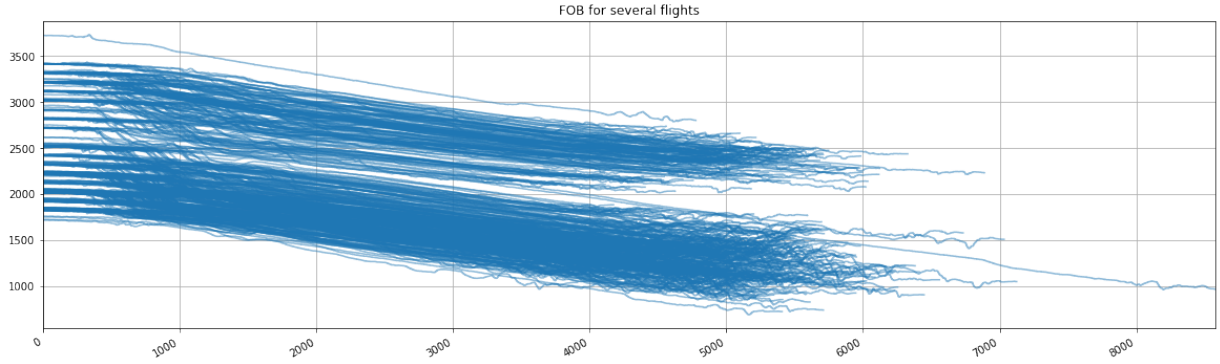


Question 1.2

Comment this visualization of FOB for several flights.

- Is there a pattern?
- Do you notice anything strange?

In [10]:
```
# Create a figure and one subplot
fig, ax = plt.subplots(figsize=(20, 6))

for df in dfs[:500]:
    df.FOB.plot(use_index=False, color="C0", ax=ax, title="FOB for sever
al flights", alpha=0.5)
```



COMMENT

The linear decay of most of the flights seem to have a similar shape for all of them. Also, it seems like there are two separate parts on this data: flights starting with up to 2500kg and those starting wiht fuel above that value. Out guess is that the planes are refilled once every two flights (maybe go and return?), since the ending of the top part of the graph matches the beginning of the lower part of the graph. This would make sense considering that the flights take in average less tha 2h, meaning that they are short distance flights

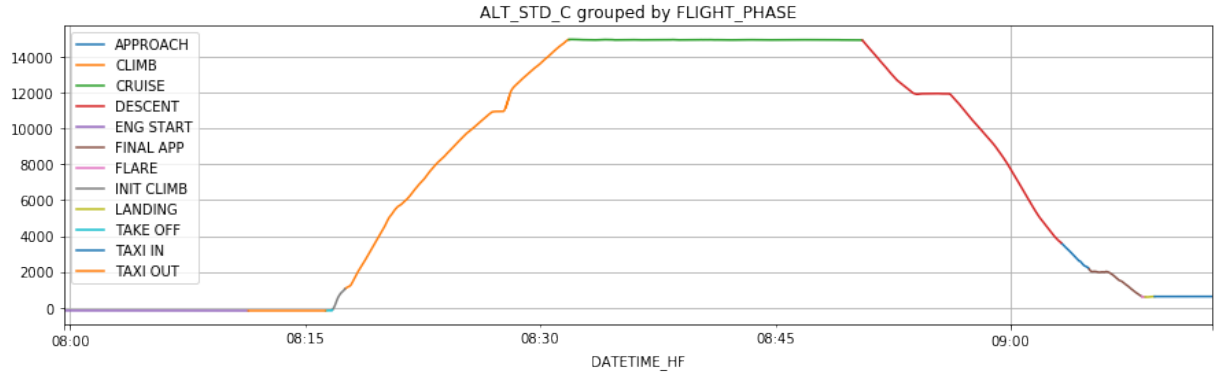
It's also worth noting that all flights seem to start with a fixed value of fuel that resembles a scale (because of the steps at the beginning of the graph).

About FLIGHT_PHASE column

Observe how ALT_STD_C varies for each phase

In [9]:
```
# Create a figure and one subplot
fig, ax = plt.subplots(figsize=(15, 4))

# Give an alias to dfs[0]
df = dfs[0]

df.groupby("FLIGHT_PHASE").ALT_STD_C.plot(title="ALT_STD_C grouped by FL
IGHT_PHASE", ax=ax);
ax.legend()
```

Out[9]:
```
<matplotlib.legend.Legend at 0x2294de048>
```



Typical chronology of cruise phases:

- ENG START

- TAXI OUT
- TAKE OFF
- INIT CLIMB
- CLIMB
- CRUISE
- DESCENT
- APPROACH
- FINAL APP
- FLARE
- LANDING
- TAXI IN

Phases that exist only for some flights:

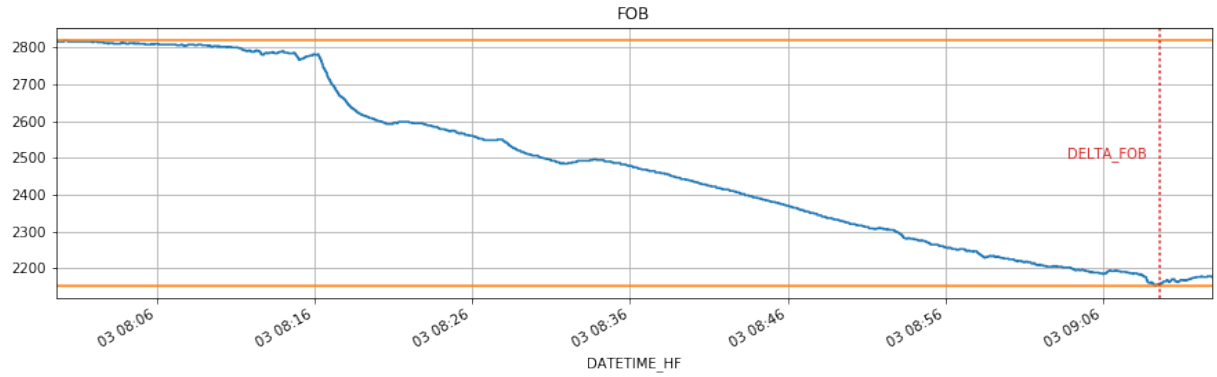
- LVL CHANGE
- GO AROUND
- TOUCH N GO

About DELTA_FOB

- MAX_FOB: FOB at the beginning of the flight
- MIN_FOB: FOB at the end of the flight
- DELTA_FOB: MAX_FOB - MIN_FOB the amount of fuel consumed during the whole flight

In [8]:
```
# Create a figure and one subplot
fig, ax = plt.subplots(figsize=(15, 4))

# Give an alias to dfs[0]
df = dfs[0]

df.FOB.plot(title="FOB")
plt.axhline(df.FOB.max(), color="C1")
plt.axhline(df.FOB.min(), color="C1")
plt.axvline(df.FOB.index[-200], linestyle=":", color="C3")
plt.text(df.FOB.index[-550], 2500, "DELTA_FOB", color="C3")
```

Out[8]:
```
<matplotlib.text.Text at 0x2294de5c0>
```



From this FOB signal we will only be interested in deltas, that is the amount of fuel consumed over a certain time period. The first approach will focus on predicting the delta of all the flight, the second approach will be predicting the deltas of each flight phase

2 Use case

The story

A company asks you to help them optimize the fuel consumption of their fleet. They've been collecting data from their flights for 2 years, operating on four different routes. They do not understand why sometimes the pilots consume 800 kilograms of fuel and sometimes 600 kg for the same route. The company provided you with all their flights.

Your objective is two-fold:

- Create a model of the quantity of fuel consumed
- Tell the company how their pilots should fly the aircraft to optimize their fuel consumption

In this part we will only use some summarised features of each flight, not the whole time series. Here are the features we will be using:

features_df

VAR	DESCRIPTION
DATE_HF	date
DESTINATION	destination airport
ORIGIN	origin airport
RUNWAY_LD	runway landing
RUNWAY_TO	runway take-off
MAX_FOB	FOB at origin
MAX_GW	Gross weigth at origin
TIME_APPROACH	Approach length
TIME_CLIMB	Climb length
TIME_CRUISE	Cruise length
TIME_DESCENT	Descent length
TIME_ENG START	Eng start length
TIME_FINAL APP	Final app length
TIME_FLARE	Flare length
TIME_GO AROUND	Go around length
TIME_INIT CLIMB	Init climb length
TIME_LANDING	Landing length
TIME_LVL CHANGE	Lvl change length
TIME_TAKE OFF	Take off length
TIME_TAXI IN	Taxi in length
TIME_TAXI OUT	Taxi out length

TIME_TOUCH N GO	Touch n go length
TIME_TOTAL	Total length

From these features we want to predict the amount of fuel consumed during the whole flight, that is `DELTA_FOB`. Imagine that at the end of flight the sensor that measures FOB broke and that we cannot know how much fuel we have left.

output_df

VAR	DESCRIPTION
DELTA_FOB	FOB conso

The `features_df` containing all the features described above has been computed for you, as well as `output_df`.

In [13]:
```
features_df = pd.read_pickle(file_features_df)
features_df.head()
```

Out[13]:

	DATE_HF	ORIGIN	DESTINATION	RUNWAY_TO	RUNWAY_LD	MAX_FOB	MAX_GW	T
0	03/12/15	ARPT0	ARPT1	08	26	2820	20620.0	1
1	03/12/15	ARPT0	ARPT1	08	26	2752	18480.0	2
2	22/06/15	ARPT0	ARPT3	24	28	2026	17780.0	1
3	10/08/15	ARPT0	ARPT3	08	28	1930	18300.0	3
4	21/08/16	ARPT0	ARPT4	20	33	2054	21540.0	6

In [14]:
```
output_df = pd.read_pickle(file_output_df)
output_df.head()
```

Out[14]:

	DELTA_FOB
0	668
1	630
2	1072
3	1000
4	926

Question 2.1

In this question we compute some statistics about the population of flights:

- How many different origin airports?
- How many different destination airports?
- How many routes? How many flights per route?

In [14]:
```
features_df['ORIGIN'].value_counts()
```

Out[14]:
```
ARPT0    2742
Name: ORIGIN, dtype: int64
```

In [13]:
```
features_df['DESTINATION'].value_counts()
```

Out[13]:
```
ARPT2    956
ARPT3    929
ARPT4    442
ARPT1    415
Name: DESTINATION, dtype: int64
```

COMMENT

- There is one single origin airport.
- There are 4 different destination airports.
- There are 4 different routes (one for each pair [ORIGIN, DESTINATION]) and the following flights per route: [956, 929, 442, 415].

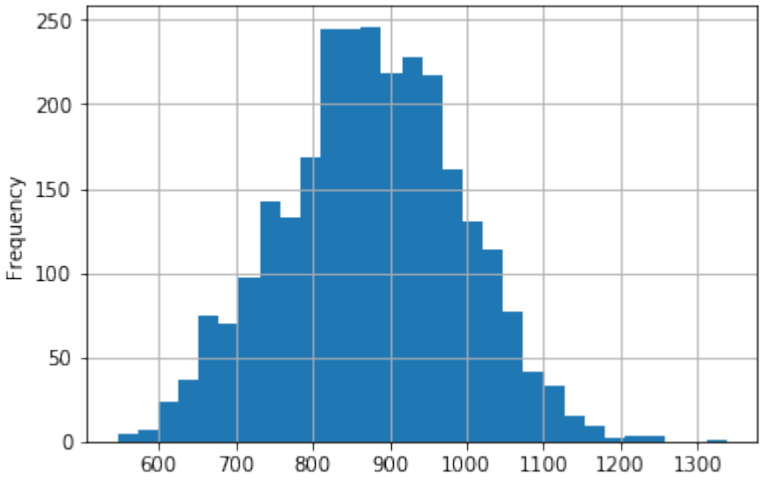
Question 2.2

In this question we focus on the output we want to predict: `DELTA_FOB` in dataframe `output_df`

- Plot the DELTA_FOB distribution and comment.
- What influences the most DELTA_FOB according to you? There is no right or wrong answer.

In [16]:
```
output_df.DELTA_FOB.plot.hist(bins=30)
```

Out[16]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x7f1048fc3668>
```



COMMENT

The minimum values are arround 600kg, the mean in arround 900kg and the maximum arroung 1300kg. Standard deviation is arround 100kg.

In our oppinion, the features affecting the most to the fuel consumption are: DESTINATION (further airports demand more consumption), MAX_GW (heavier planes consume more),

TIME_CLIMB and TIME_INIT_CLIMB (because it's when the fule is consumed at a higher rate) and TIME_TOTAL (the longer the flight, the more consumption)

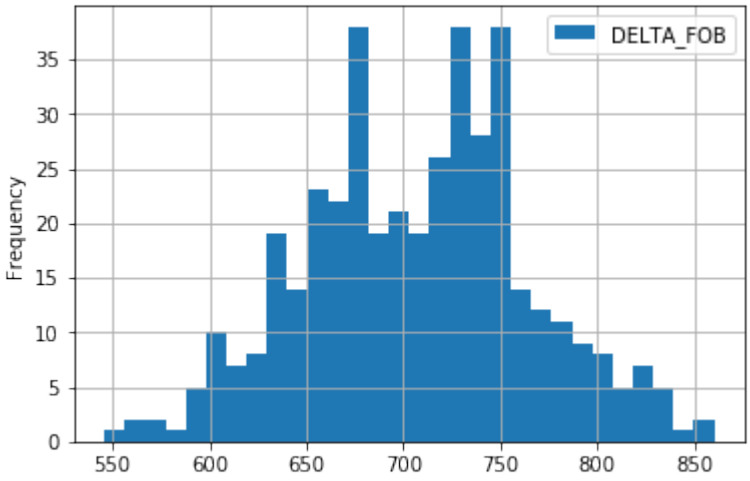
Question 2.3

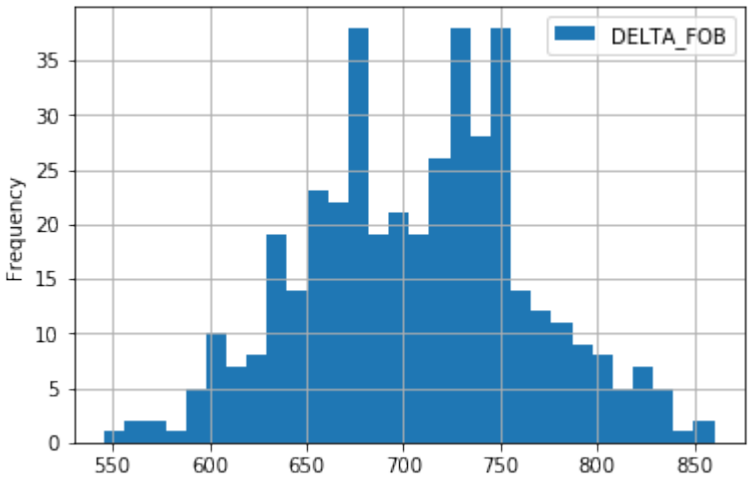
In this question we work on the distributions of `DELTA_FOB` conditionned on `DESTINATION` and (optional) `RUNWAY_LD`

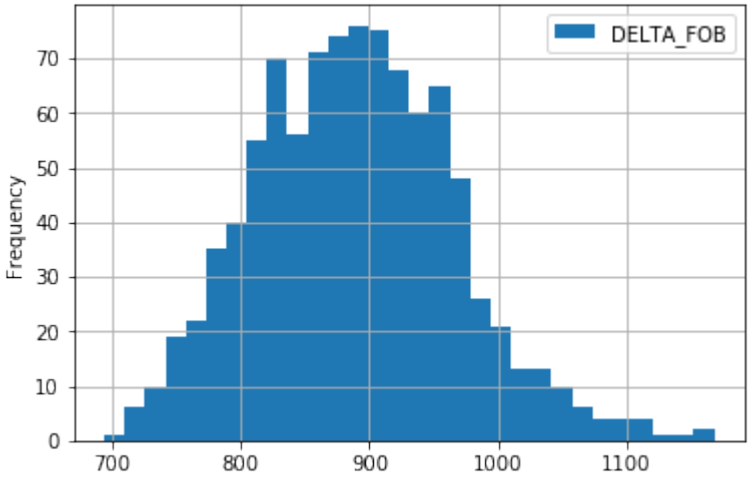
- Plot `DELTA_FOB` distribution conditionned on `DESTINATION` airport and comment.
- (optional) Plot `DELTA_FOB` distribution conditionned on `DESTINATION` airport and `RUNWAY_LD` and comment.

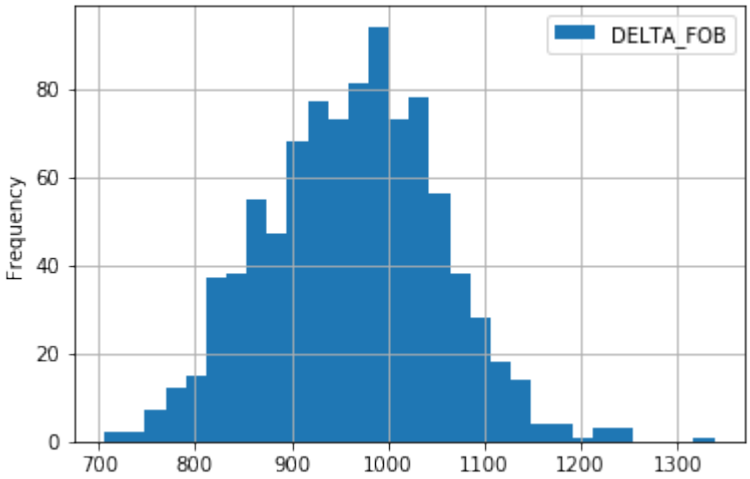
In [42]:
```
df2 = pd.concat([output_df.DELTA_FOB, features_df.DESTINATION], axis=1)
df2.groupby('DESTINATION').plot.hist(bins=30)
```

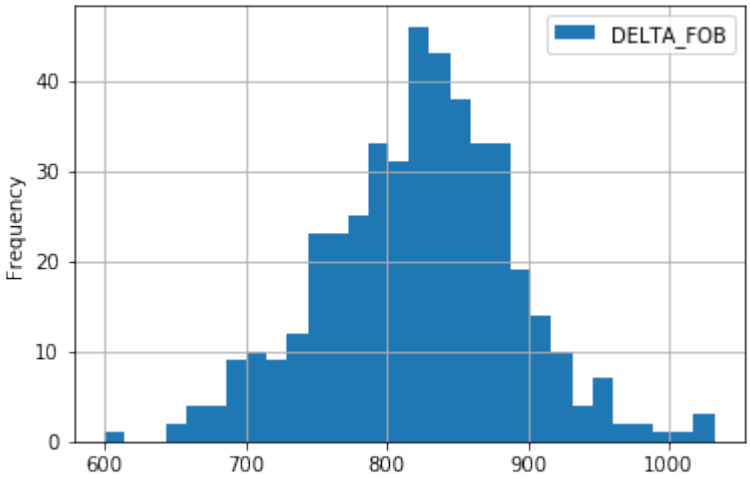
Out[42]:
```
DESTINATION
ARPT1    Axes(0.125,0.125;0.775x0.755)
ARPT2    Axes(0.125,0.125;0.775x0.755)
ARPT3    Axes(0.125,0.125;0.775x0.755)
ARPT4    Axes(0.125,0.125;0.775x0.755)
dtype: object
```











COMMENT

The DELTA_FOB changes greatly depending on the DESTINATION. The consumption is: ARPT1 < ARPT4 < ARPT2 < ARPT3 (distances whould behave accordingly)

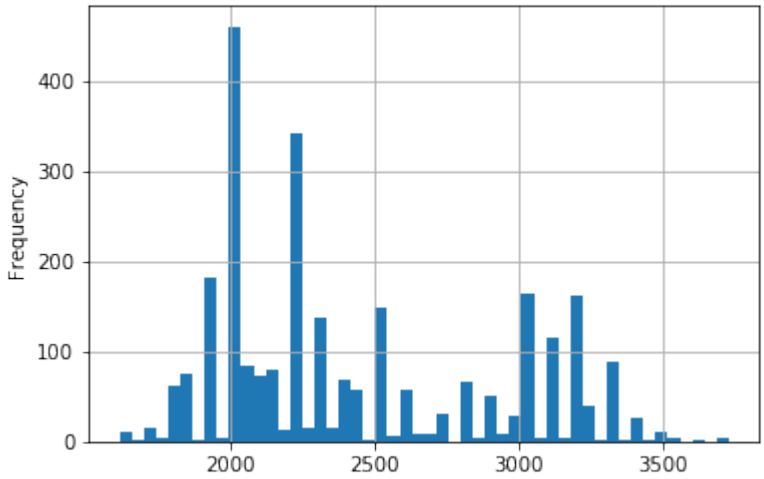
Question 2.4

Finally let's work with two important features: the total duration of the flight `TOTAL_TIME` and the quantity of fuel at the beginning of the flight `MAX_FOB`.

- Plot the distributions of the following variables:
  - MAX_FOB
  - TIME_TOTAL
- Comment these distributions

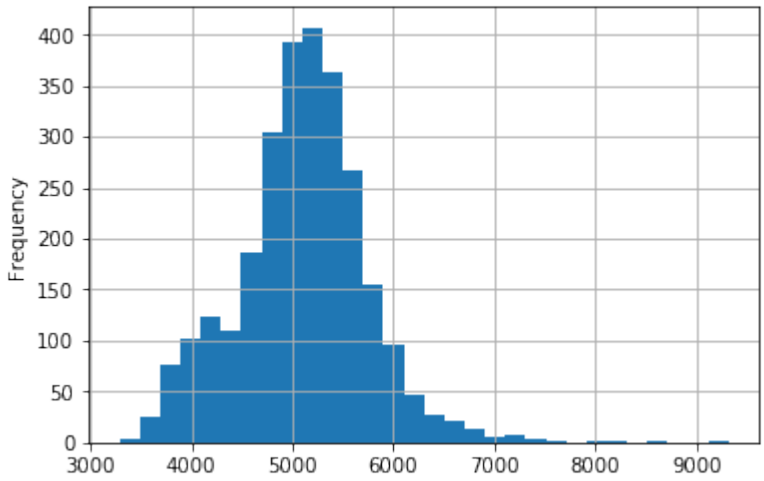
In [48]:
```
features_df.MAX_FOB.plot.hist(bins=50)
features_df.MAX_FOB.quantile([0.1, 0.25, 0.5, 0.75, 0.9])
```

Out[48]:
```
0.10    1926.0
0.25    2028.0
0.50    2243.0
0.75    2922.0
0.90    3218.0
Name: MAX_FOB, dtype: float64
```



In [46]:
```
features_df.TIME_TOTAL.plot.hist(bins=30)
features_df.TIME_TOTAL.quantile([0.1, 0.25, 0.5, 0.75, 0.9])
```

Out[46]:
```
0.10    4189.10
0.25    4741.25
0.50    5119.00
0.75    5467.75
0.90    5827.90
Name: TIME_TOTAL, dtype: float64
```



COMMENT

The histogram of MAX_FOB It matches the previous plot in question 1.2. Its shape is full of

steps, as expected from the previous figure since the initial value of FOB was full of steps/gaps. The mean is 2243, and it has a high standard deviation.

The histogram of TIME_TOTAL shows a gaussian distribution, with mean in 5100 and a standard deviation of 800.

3 Model DELTA_FOB

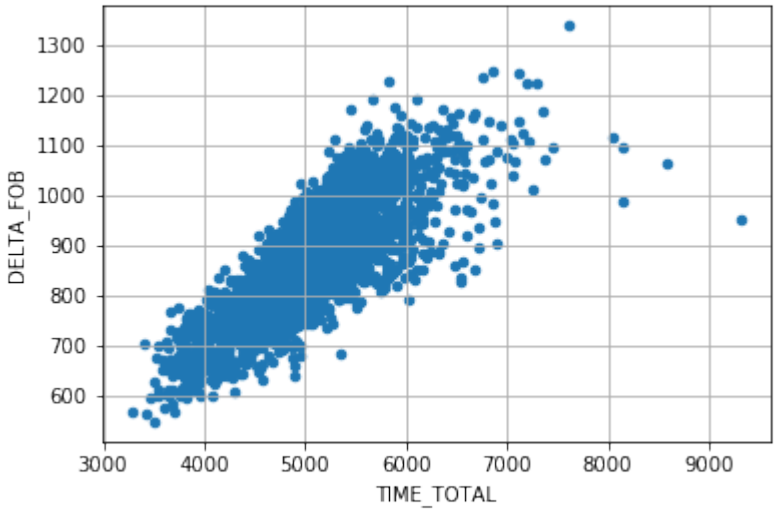
We use sklearn for models.

Question 3.1

- Scatter plot `TIME_TOTAL` against `DELTA_FOB`.
- Is there any relationship?

In [51]:
```
df3 = pd.concat([output_df.DELTA_FOB, features_df.TIME_TOTAL], axis=1)
df3.plot.scatter('TIME_TOTAL', 'DELTA_FOB')
```

Out[51]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x7f104593efd0>
```



COMMENT
There is a clear linear correlation between total time and delta fob, as expected. We can also see some outliers on this plot, as well as the main cluster of points.

Let's explore the predictive power of `TIME_TOTAL` on `DELTA_FOB`. Our first model is the following linear regression:

- algorithm: `LinearRegression` from sklearn package `linear_model` (reference)
- input: `TIME_TOTAL`
- output: `DELTA_FOB`

Follow the code below, in the following questions you will extensively use this template code and adapt it.

Question 3.2

Comment briefly the results, the following questions can help you:

- What is the score method implemented by sklearn?
- How do you relate it to the mse?
- What is best to interprete the quality of the estimator?

In [181]:
```
# This is template code for setting up a sklearn regression model
def modelTraining(featureMtrx=["TIME_TOTAL"], featureMtrx2=["DELTA_FOB"]
, test_size = 0.2, random_state = 42,
             estimator = linear_model.LinearRegression(), reg=False):

    X = features_df[featureMtrx].as_matrix()
    y = output_df[featureMtrx2].as_matrix()



    # Split the features and output in training and test sets
    X_train, X_test, y_train, y_test = model_selection.train_test_split(
X, y, test_size=test_size, random_state=random_state)

    if(reg):
        X_train, X_test = regularize(numeric_features)

    # Train the model using the training sets
    estimator.fit(X_train, y_train)

    # Test the model on tests sets
    print("score (R2)", estimator.score(X_test, y_test))
```

In [103]:
```
modelTraining()
```

```
score (R2) 0.606144364551
```

COMMENT

The score method used is R^2, which is a normalized version of Mean Squared Error $$MSE = (y-yPred)**2.sum()$$ $$R2 = 1-MSE/((y-mean(y))**2.sum())$$ The R2 is a better way to interpret the performance of an estimator because it has a relative value up to 1 (where a value above 0 is considered as better than constant values), whereas MSE doesn't hold any information by itself (you need information about the data in order to understand it).

Question 3.3

- Adapt the code above for the following model
  - algorithm: `LinearRegression`
  - input: `TIME_TOTAL`, `MAX_FOB`
  - output: `DELTA_FOB`
- Compare the score with the results of the previous regression
  - is it getting better?

which variable has the most predictive power among `TIME_TOTAL` and `MAX_FOB`?

In [105]:
```
modelTraining(featureMtrx=['TIME_TOTAL', 'MAX_FOB'])
modelTraining(featureMtrx=['MAX_FOB'])
```

```
score (R2) 0.708710764451
score (R2) 0.0797671623189
```

COMMENT

It's clearly getting better (0.1 more in R2). When comparing TIME_TOTAL with MAX_FOB, we find TIME_TOTAL to be much more powerfull towards prediction of DELTA_FOB.

Before moving on and adding more features to our models, let's compare our linear regressions to a simple non-linear model:

- algorithm: `DecisionTreeRegressor` from sklearn package `tree` (reference)
- input: `TIME_TOTAL`, `MAX_FOB`
- output: `DELTA_FOB`

Question 3.4

- Adapt the code for the model described above.
- What is the default behavior for `max_depth` parameter?
- With a default behavior, is the score better than with linear regression?
- Split in 10 validation sets and optimize `max_depth` parameter (you can use the template code below)
  - plot `max_depth` parameters against the score
  - why is it not OK to optimize parameters only with train and test sets?
- What attributes can you use to you interprete the tree?

In [106]:
```
modelTraining(featureMtrx=['TIME_TOTAL', 'MAX_FOB'], estimator=tree.Deci
sionTreeRegressor())
```

```
score (R2) 0.594651328213
```

COMMENT

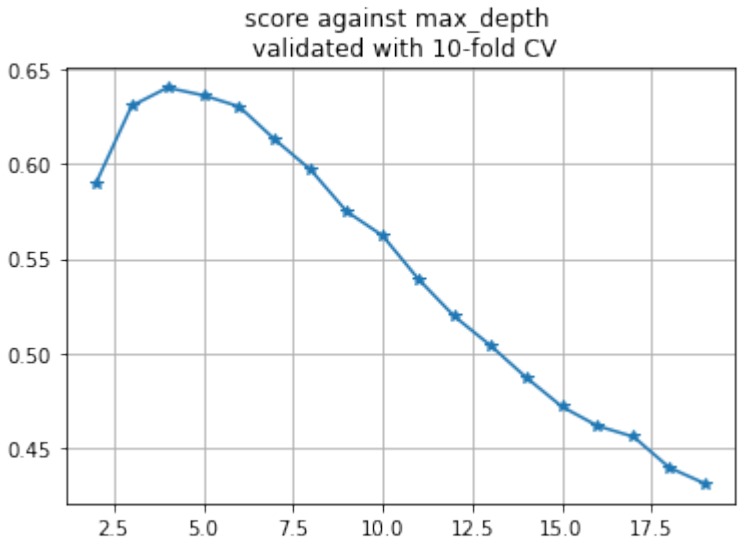
- the default max_depth is None, meaning there is no restriction to the maximum depth of the tree. This could lead to overfitting.
- The score with the default behavior is worse than the linear regression model with those two parameters (and very close to the linear regression only with TIME_TOTAL)

In [80]:
```
max_depths = range(2,20)

kFold = model_selection.KFold(n_splits=10)
scores = []
```

```
for max_depth in max_depths:
    estimator=tree.DecisionTreeRegressor()
    estimator.set_params(max_depth=max_depth)
    score = []
    folds = kFold.split(X)
    for fold in folds:
        # Unpack train and test indices
        train, test = fold
        # Split in train and test sets
        X_train, X_test, y_train, y_test = X[train], X[test], y[train],
y[test]
        estimator.fit(X_train, y_train)
        score.append(estimator.score(X_test, y_test))
    scores.append(np.mean(score))
    print("NodeCount for tree %d: %d"%(max_depth, estimator.tree_.node_c
ount))
plt.plot(max_depths, scores, marker="*")
plt.title("score against max_depth \n validated with 10-fold CV")
```

```
NodeCount for tree 2: 7
NodeCount for tree 3: 15
NodeCount for tree 4: 31
NodeCount for tree 5: 61
NodeCount for tree 6: 119
NodeCount for tree 7: 221
NodeCount for tree 8: 369
NodeCount for tree 9: 569
NodeCount for tree 10: 817
NodeCount for tree 11: 1081
NodeCount for tree 12: 1381
NodeCount for tree 13: 1661
NodeCount for tree 14: 1903
NodeCount for tree 15: 2127
NodeCount for tree 16: 2327
NodeCount for tree 17: 2479
NodeCount for tree 18: 2581
NodeCount for tree 19: 2659
```

Out[80]: <matplotlib.text.Text at 0x7f1044011358>



COMMENT

- the default max_depth is None, meaning there is no restriction to the maximum depth of the tree. This could lead to overfitting.
- The score with the default behavior is worse than the linear regression model with those two parameters (and very close to the linear regression only with TIME_TOTAL)
- We should use a **validation set** because when tuning the hyperparameters we are fitting those hyperparameters to the test set, without knowing how it would behave with other datasets. Fitting a hyperparameter such as max_depth should be done using a validation set always.
- We have the estimator.tree_, which can be used to obtain data from the tree suchas node_counts(how populated the tree is), or thresholds (what decissions is the tree making). There is alsoa package to print the tree.

Let's add all the other numeric features.

In [81]:
```
numeric_features = ['MAX_FOB',
                    'MAX_GW',
                    'TIME_APPROACH',
                    'TIME_CLIMB',
                    'TIME_CRUISE',
                    'TIME_DESCENT',
                    'TIME_ENG START',
                    'TIME_FINAL APP',
                    'TIME_FLARE',
                    'TIME_GO AROUND',
                    'TIME_INIT CLIMB',
                    'TIME_LANDING',
                    'TIME_LVL CHANGE',
                    'TIME_TAKE OFF',
                    'TIME_TAXI IN',
                    'TIME_TAXI OUT',
                    'TIME_TOUCH N GO',
                    'TIME_TOTAL']
```

Question 3.5

In Question 3.5 we will use regularized regression. In this question we take a preliminary step and rescale the features.

We use the `StandardScaler` from sklearn `preprocessing` package (reference).

- Give one reason for rescaling the features before doing regularized regression
- Fill in the blanks in the code below and observe the results printed out
- Why the first feature of `X_test_transformed` does not have a std of 1?

In [182]:
```
def regularize(num_features=numeric_features, out_features=["DELTA_FOB"]
, dataframex=features_df, dataframey=output_df):
    X = dataframex[num_features].as_matrix()

    y = dataframey[out_features].as_matrix()

    test_size = 0.2
    random_state = 42
```

```
    X_train, X_test, y_train, y_test = model_selection.train_test_split(
X, y, test_size=test_size, random_state=random_state)


    scaler = preprocessing.StandardScaler()

    scaler.fit(X_train, y_train)

    X_train_transformed = scaler.transform(X_train)
    X_test_transformed = scaler.transform(X_test)

    print("X_train_transformed first feature,", "mean:", X_train_transfo
rmed[:, 0].mean(), "std:", X_train_transformed[:, 0].std())
    print("X_test_transformed first feature,", "mean:", X_test_transform
ed[:, 0].mean(), "std:", X_test_transformed[:, 0].std())
    return X_train_transformed, X_test_transformed
```

In [89]:
```
X_train_transformed, X_test_transformed = regularize(numeric_features)
```

```
X_train_transformed first feature, mean: -1.36892068335e-16 std: 1.0
X_test_transformed first feature, mean: 0.0510960710978 std: 1.038384287
34
```

COMMENT

Regularization is important to ensure the model weights all feature values equally (specially useful for representation), obtaining a normalized scale for all the features and avoiding numerical problems that may happen due to different ranges.

The X_test is regularized using the standard deviation and mean from the training data, which is why we obtain a std different from 1.

The last question of this part focus on regularized linear regression.

- algorithm: `Ridge` for sklearn package `linear_model` (reference)
- input: `numeric_features` of `features_df`
- output: `DELTA_FOB`

Question 3.6

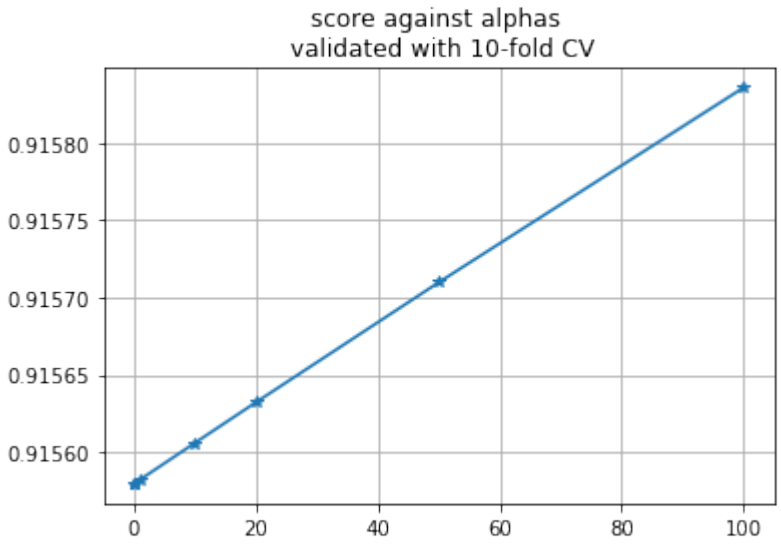
- Use the previous template codes and adapt it to set up the models described above and rescale the features
- Split in validation sets and optimize `alpha` parameter
- Interprete the coefficients of your best estimator

In [100]:
```
# INPUT
X_train_transformed, X_test_transformed = regularize(numeric_features)

# HYPERPARAMETER
alphas = [0.001, 0.01, 0.1, 1, 10, 20, 50, 100]

kFold = model_selection.KFold(n_splits=10)
scores = []
```

```
for alpha in alphas:
    estimator=linear_model.Ridge(alpha=alpha)
    score = []
    folds = kFold.split(X_train_transformed)
    for fold in folds:
        # Unpack train and validation indices
        train, val = fold
        # Split in train and validation sets
        X_train, X_val, y_train, y_val = X[train], X[val], y[train], y[v
al]
        estimator.fit(X_train, y_train)
        score.append(estimator.score(X_val, y_val))
    scores.append(np.mean(score))
plt.plot(alphas, scores, marker="*")
plt.title("score against alphas \n validated with 10-fold CV")
```

```
X_train_transformed first feature, mean: -1.36892068335e-16 std: 1.0
X_test_transformed first feature, mean: 0.0510960710978 std: 1.038384287
34
```

Out[100]: <matplotlib.text.Text at 0x7f103caadba8>



In [109]:
```
modelTraining(numeric_features, estimator=linear_model.Ridge(alpha=1), r
eg=True)
```

```
X_train_transformed first feature, mean: -1.36892068335e-16 std: 1.0
X_test_transformed first feature, mean: 0.0510960710978 std: 1.038384287
34
score (R2) 0.919968073452
```

COMMENT

Alpha has almost no impact on the performance of the estimator, since we always get almost the same score. We have decided to use alpha=1, the default value. Our final score in the Test set is **R2=0.92**

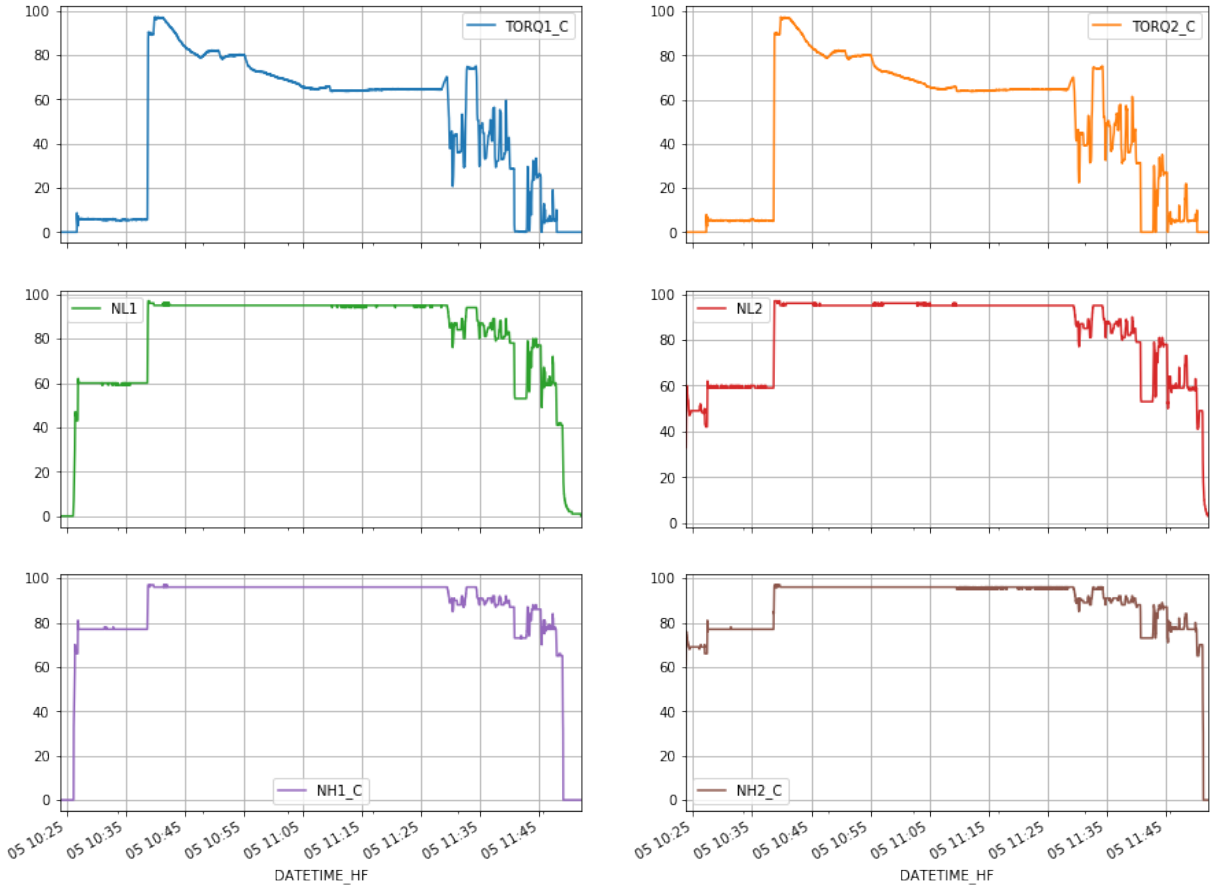
4 Feature engineering & model delta_fob for each phase

We will add a lot of information with engine-related time series:

- `TORQ1_C`, `TORQ2_C`
- `NL1`, `NL2`
- `NH1_C`, `NH2_C`

Visualization of engine-related time series for one flight

In [110]:
```
# Five an alias to dfs[0]
df = dfs[0]

engine_features = ['TORQ1_C', 'TORQ2_C',
                   'NL1', 'NL2',
                   'NH1_C', 'NH2_C']

df[engine_features].plot(subplots=True, layout=(3, 2), figsize=(15, 12))
;
```



Question 4.1

We will extract few numbers out of each phase for all engine variables (`TORQ1_C`, `TORQ2_C`, `NL1`, `NL2`, `NH1_C`, `NH2_C`)

- Feature engineering
  - get `TORQ1_C` of one phase of one flight, do a linear regression by index and plot the result
  - is it OK to resume the signal by a straight line? what other features would you

add?

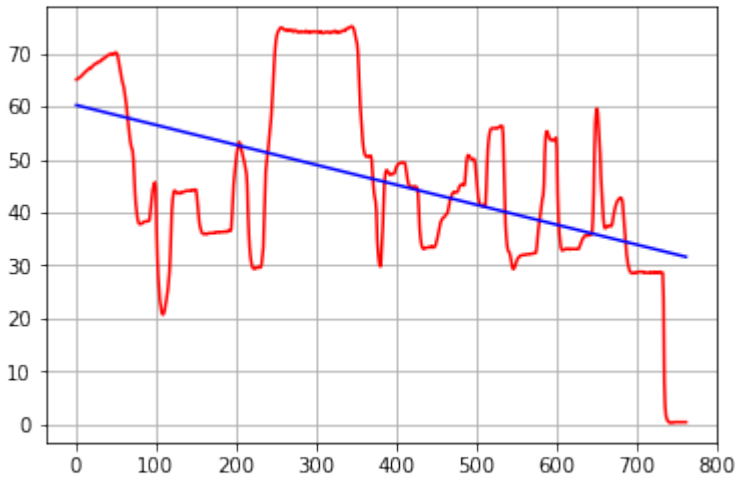
In [151]:
```
len(y_train)
```

Out[151]:
```
608
```

In [156]:
```
y = df[df.FLIGHT_PHASE=='DESCENT'].TORQ1_C.as_matrix()
X = np.arange(1, len(X)+1)
X = X.reshape(-1, 1)

test_size = 0.2
random_state = 42
estimator = linear_model.LinearRegression()

# Split the features and output in training and test sets
X_train, X_test, y_train, y_test = model_selection.train_test_split(X, y
, test_size=test_size, random_state=random_state)

# Train the model using the training sets
estimator.fit(X_train, y_train)

# Test the model on tests sets
print("score (R2)", estimator.score(X_test, y_test))
```

```
score (R2) 0.2926616323
```

In [158]:
```
pred = estimator.predict(X)
plt.plot(X, y, 'r')
plt.plot(X, pred, 'b')
plt.show()
```



COMMENT

Clearly a linear estimation is not the best we could do for this phase (DESCENT). I would probably add the NL1, NL2, NH1_C, NH2_C information (due to the high correlation between engine related features).

These features have been created for you in the `features2_df` dataframe.

Recap features2_df

In [159]:
```
file_features2 = BASE_DIR + "features2_df.pkl"

features2_df = pd.read_pickle(file_features2)

numeric_features2 = list(features2_df.columns.values)
numeric_features2.remove("DATE_HF")
numeric_features2.remove("ORIGIN")
numeric_features2.remove("DESTINATION")
numeric_features2.remove("RUNWAY_TO")
numeric_features2.remove("RUNWAY_LD")

features2_df.describe()
```

Out[159]:

	MAX_FOB	MAX_GW	TIME_APPROACH	TIME_CLIMB	TIME_CRUISE	TIME_
count	2742.000000	2742.000000	2742.000000	2742.000000	2742.000000	2742.0
mean	2439.663020	20495.557987	143.109044	1190.589351	1568.312181	745.98
std	482.857964	1161.616565	94.942888	258.253887	581.133001	182.28
min	1616.000000	17300.000000	1.000000	85.000000	9.000000	230.00
25%	2028.000000	19640.000000	83.000000	1012.000000	1102.250000	613.00
50%	2243.000000	20540.000000	126.000000	1160.000000	1687.500000	715.00
75%	2922.000000	21400.000000	187.000000	1340.000000	1995.750000	854.00
max	3732.000000	23440.000000	1560.000000	2445.000000	3467.000000	1892.0

8 rows × 468 columns

Let's be more ambitious and predict the fuel consumed by each phase. It would help a lot more to model the fuel consumption behavior for each phase when it will come to interpretation. The dataframe `output2_df` containing the fuel consumed by each phase for each flight has been created for you.

In [180]:
```
file_output2 = BASE_DIR + "output2_df.pkl"
output2_df = pd.read_pickle(file_output2)

output2_df.describe()
```

Out[180]:

	APPROACH	CLIMB	CRUISE	DESCENT	ENG START	FINAL APP	
count	2742.000000	2742.000000	2742.000000	2742.000000	2742.000000	2742.000000	274
mean	44.630197	245.240700	293.458789	148.687819	21.590810	43.513494	3.2
std	31.565958	57.213462	115.312441	50.315759	24.553695	24.024636	2.8
min	0.000000	54.000000	2.000000	16.000000	4.000000	4.000000	0.0
25%	22.000000	202.000000	204.000000	116.000000	14.000000	28.000000	2.0

50%	42.000000	240.000000	306.000000	154.000000	18.000000	42.000000	2.0
75%	62.000000	282.000000	376.000000	180.000000	26.000000	56.000000	4.0
max	712.000000	512.000000	754.000000	416.000000	1170.000000	410.000000	24.

Question 4.3

In this question we make use of `output2_df` to determine the phase that consumes the most fuel per second in average.

- What flight phase consumes the most in average?
- What flight phase consumes the most per second in average?

In [162]:
```
# This is template code to extract the phases durations out of features_df and rename the columns to make them identical to output2_df

phases_durations_df = features2_df[["TIME_" + fp for fp in flight_phases]]
phases_durations_df.columns = flight_phases
phases_durations_df.head()
```

Out[162]:

	APPROACH	CLIMB	CRUISE	DESCENT	ENG START	FINAL APP	FLARE	INIT CLIMB	LANDING	TAI O
0	106	857	1123	762	705	203	12	49	36	26
1	219	1129	649	933	174	151	7	39	53	22
2	152	1139	2418	366	354	188	10	47	36	23
3	32	749	2676	619	249	243	9	34	32	21
4	62	1038	1040	1210	302	161	5	34	24	27

In [172]:
```
# FLIGHT CONSUMPTION PER PHASE
output2_df.mean()
```

Out[172]:
```
APPROACH       44.630197
CLIMB         245.240700
CRUISE        293.458789
DESCENT       148.687819
ENG START      21.590810
FINAL APP      43.513494
FLARE           3.267688
GO AROUND       0.868709
INIT CLIMB     42.369803
LANDING        48.817651
LVL CHANGE      2.668855
TAKE OFF       24.814004
TAXI IN        55.407002
TAXI OUT       37.649161
TOUCH N GO      0.024799
dtype: float64
```

In [173]:
```
# FLIGHT CONSUMPTION RATE PER PHASE
```

```
output2_df.mean()/phases_durations_df.mean()
```

Out[173]:
```
APPROACH      0.311861
CLIMB         0.205983
CRUISE        0.187118
DESCENT       0.199317
ENG START     0.043524
FINAL APP     0.224860
FLARE         0.378730
GO AROUND     0.250552
INIT CLIMB    0.881011
LANDING       0.991254
LVL CHANGE    0.357185
TAKE OFF      0.975064
TAXI IN       0.200860
TAXI OUT      0.110724
TOUCH N GO    0.601770
dtype: float64
```

COMMENT
CRUISE consumes the most on average: 293.458789
LANDING consumes the most fuel on average per second: 0.991254

Question 4.4

This last question aims to let you work with `features2_df` and `output2_df`. You can make use of any pieces of code we have worked with so far: especially to set up your models, rescale the features, optimize and validate your models... The expected outcome is to explain to the company what parameters influence the most the fuel consumption of their fleet and give them advice to consume less.

- The algorithm:
  - algorithm: you choose
  - input: `features2_df`, output: `output2_df`
- Interprete the coefficients, explain the causes of fuel consumption for each phase, give adive to the company to consume less in their following flights

In [191]:
```
# This is template code for setting up a sklearn regression model
def modelTraining(featureMtrx=["TIME_TOTAL"], featureMtrx2=["DELTA_FOB"]
, test_size = 0.2, random_state = 42,
             estimator = linear_model.Ridge(), reg=False):

    X = features2_df[featureMtrx].as_matrix()
    X = X.reshape(-1, 1)
    y = output2_df[featureMtrx2].as_matrix()



    # Split the features and output in training and test sets
    X_train, X_test, y_train, y_test = model_selection.train_test_split(
X, y, test_size=test_size, random_state=random_state)

    if(reg):
        X_train, X_test = regularize(numeric_features)
```

```
    # Train the model using the training sets
    estimator.fit(X_train, y_train)

    # Test the model on tests sets
    # print("score (R2)", estimator.score(X_test, y_test))
    return estimator.score(X_test, y_test)
```

In [192]:
```
mtrx = []
for col in output2_df.columns:
    m=[]
    for row in numeric_features2:
        m.append(modelTraining(row, col))
    mtrx.append(m)
```

Answer

Your answer goes here

COMMENT

For each phase we analyzed each attribute by itself in order to modelize with a ridge the fuel consumption. The final idea would be to plot/tablize the mtrx and filter out all the low values. The rest of the values would help us decide which attributes are more important for which phase. In short, whe're missing some filtering and some plotting.