User Manual for FunctionalSmoothingSpline

Yuriy A. Korablev

1 INSTALLATION

Copy FunctionalSmoothingSpline.py and MyLemke.py files to your project directory (https://github.com/YuAKorablev/FunctionalSmoothingSpline). Import the function from this module (or just copy it directly to your code).

```
from FunctionalSmoothingSpline import *
```

For the numerical solution, in case this optimization method is chosen, the minimize method from the `scipy` package is used. In this case, make sure that `scipy` package is installed, `%pip install scipy`.

To solve a quadratic programming problem, three methods are used to choose from: Lemke's algorithm implemented in pure Python; the same Lemke Algorithm, but optimized using numba library; CVXOPT optimizer, which is based on the interior point algorithm.

By default, the Lemke algorithm is used to calculate a spline with the positivity condition. File MyLemke.py should be located in the same directory. Also, the implementation of Lemke's algorithm is accelerated by compiling on the fly using the numba library. In MyLemke.py module the numba is imported (`from numba import njit`). If you do not already have this library installed, you should install it with `%pip install numba`. When using the Lemke algorithm optimized using the numba library, the first function call may take quite a long time, it is compiled on the fly using the numba library, but subsequent calls will be very fast. If you use the CVXOPT optimizer, you must first install the CVXOPT package using `%pip install cvxopt`. The CVXOPT optimizer displays additional messages; to disable the output of these messages, you can add the following line before calling the function.

```
cvxopt.solvers.options['show_progress'] = False
```

Standard libraries numpy, pandas, matplotlib must be installed as well.

As data files from the presented examples, copy files Sales1.csv, DiscrSignals.csv to the project directory. Open test.ipynb file in your Jupyter Notebook, or Example *.py files in your Python, or implement the below code.

2 IMPLMENTATION

Function FunctionalSmoothingSpline has following arguments:

- t_f – array of moments of observed function values (t_i);
- values_f – array of function values (y_i);
- weights_f – array of observation weights for function values (w_i^f);
- t_df – array of moments of observed first derivative values (t_j);
- values_df – array of first derivative values (y'_j);
- weights_df – array of observation weights for first derivatives (w_j^{df});

- coef_df – group weight of the first derivative observations (μ, 1 by default);
- t_d2f – array of moments of observed second derivative values (t_l);
- values_d2f – array of second derivative values (y_l'');
- weights_d2f – array of observation weights for second derivatives ($w_l^{d^2f}$);
- coef_d2f – group weight of the second derivative observations (ν, 1 by default);
- t_int_a – array of integrals start moments (t_u^a);
- t_int_b – array of integrals end moments (t_u^b);
- values_int – array of integral values (Y_u);
- weights_int – array of observation weights for integrals (w_u^{int});
- coef_int – group weight of the integrals observations (ψ, 1 by default);
- knots – spline knots (s_k);
- knots_number – knots number (m);
- alpha – smoothing coefficient (α, 1 by default);
- x – array of coordinates in which the spline values will be calculated and returned (by default it varies from the first to the last observation among all groups, step size 1);
- All_Positive – True or False, to solve as a non negative spline using Lemke's algorithm (by default False);
- method – method for solving quadratic programming problems, one of "Lemke", "Lemke_njit", "cvxopt" or "exp" ("Lemke" by default).
- output– list of the form ["y", "dy", "d2y", "integral"] indicating which values to calculate (by default ["y"] to output only values y), to output integral function $F(t)$ output should contain "integral";
- info – True or False to return addition information (by default False);
- add_knots - True or False, to add extra knots where spline becomes negative (False by default);
- add_condition_without_knots - True or False, to add extra conditions where spline becomes negative (False by default), if this is True, no new knots are added;
- new_knots_tol - tolerance for adding new knots or new conditions, i.e. a knot or condition is added if the absolute negative value is greater than the tolerance (0.0001 by default);
- max_added_knots - maximum number of knots or conditions to add (10 by default).

Any function argument can be omitted. The number of observations must be at least 2 for successful function restoration.

If info is False, then the function returns the spline values calculated at points x.

If info is True, then the function returns the following dictionary:

- x – points where spline values are calculated;
- y – spline values at points x (if output contains "y", it does by default);
- dy – derivative of the spline at points x (if output contains "dy");
- d2y – second derivative of the spline at points x (if output contains "d2y");
- integral – integral function $F(t)$ at points x (if output contains "integral");
- g – array of spline parameters $g = (g_1, \ldots, g_m)^T$, where $g_k = g(s_k)$ values of spline at m spline knots $s_1 < s_2 < \cdots < s_m$;

- gamma – array of spline parameters $\gamma = (\gamma_1, \dots, \gamma_m)^T$, where $\gamma_k = g''(s_k)$ values of the second derivatives at m spline knots $s_1 < s_2 < \cdots < s_m$, where $\gamma_1 = \gamma_m = 0$;
- knots – array of spline knots $s_1 < s_2 < \cdots < s_m$;
- error_total – value of the minimized functional $S(g) =$ error_f $+ \mu \cdot$ error_df $+ \nu \cdot$ error_d2f $+ \psi \cdot$ error_int $+ \alpha \cdot$ error_penalty, where μ, ν, ψ are group weights (see main article);
- error_f – sum of squared deviations of values $\sum_{i=1}^{n_f} w_i^f \left(y_i - g(t_i)\right)^2$;
- error_df – sum of squared deviations of first derivatives $\sum_{j=1}^{n_{df}} w_j^{df} \left(y_j' - g'(t_j)\right)^2$;
- error_d2f – sum of squared deviations of second derivatives $\sum_{l=1}^{n_{d^2f}} w_l^{d^2f} \left(y_l'' - g''(t_l)\right)^2$;
- error_int – sum of squared deviations of integrals $\sum_{u=1}^{n_{int}} w_u^{int} \left(Y_u - \int_{t_u^a}^{t_u^b} g(t)dt\right)^2$;
- error_penalty – penalty (regularization) on the curvature measure (roughness penalty) $\int_{s_1}^{s_m} \left(g''(t)\right)^2 dt$;
- fraction_error_f – proportion of value error in the total error, equal to error_f/error_total;
- fraction_error_df – proportion of first derivatives error in the total error, equal to $\mu \cdot$ error_df/ error_total;
- fraction_error_d2f – proportion of second derivatives error in the total error, equal $\nu \cdot$ error_df/ error_total;
- fraction_error_int – proportion of integrals error in the total error, equal to $\psi \cdot$ error_int/error_total;
- fraction_penalty – proportion of penalty error in the total error, equal to $\alpha \cdot$ error_penalty/error_total;
- relative_sqr_error_f – standard deviation (relative) of values $\sqrt{\sum_{i=1}^{n_f} w_i^f \left(\frac{y_i - g(t_i)}{y_i}\right)^2}$;
- relative_sqr_error_df – standard deviation (relative) of first derivatives $\sqrt{\sum_{j=1}^{n_{df}} w_j^{df} \left(\frac{y_j' - g'(t_j)}{y_j'}\right)^2}$;
- relative_sqr_error_d2f – standard deviation (relative) of second derivatives $\sqrt{\sum_{l=1}^{n_{d^2f}} w_l^{d^2f} \left(\frac{y_l'' - g''(t_l)}{y_l''}\right)^2}$;
- relative_sqr_error_int – standard deviation (relative) of second integrals $\sqrt{\sum_{u=1}^{n_{int}} w_u^{int} \left(\frac{Y_u - \int_{t_u^a}^{t_u^b} g(t)dt}{Y_u}\right)^2}$;
- relative_abs_error_f – mean absolute deviation (relative) of values $\frac{1}{n_f}\sum_{i=1}^{n_f} w_i^f \left|\frac{y_i - g(t_i)}{y_i}\right|$;
- relative_abs_error_df – mean absolute deviation (relative) of first derivatives $\frac{1}{n_{df}}\sum_{j=1}^{n_{df}} w_j^{df} \left|\frac{y_j' - g'(t_j)}{y_j'}\right|$;
- relative_abs_error_df – mean absolute deviation (relative) of second derivatives $\frac{1}{n_{d^2f}}\sum_{l=1}^{n_{d^2f}} w_l^{d^2f} \left|\frac{y_l'' - g''(t_l)}{y_l''}\right|$;
- relative_abs_error_int – mean absolute deviation (relative) of integrals $\frac{1}{n_{int}}\sum_{u=1}^{n_{int}} w_u^{int} \left|\frac{Y_u - \int_{t_u^a}^{t_u^b} g(t)dt}{Y_u}\right|$.

These relative errors are convenient because they make it easy to understand how much the data has been smoothed (toned down). If the relative error is 0.05, this means that the corresponding group of observations was smoothed by 5%. However, if there are zero values among the data, then a non-numeric value (nan or inf) will be returned (because of division by zero).

3 APPLICATION

3.1 Example 1. Reconstructing a function from its values.

Import standard Python libraries and our FunctionalSmoothingSpline function.

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from FunctionalSmoothingSpline import *
```

Data input. Let's assume that the dataset is presented in a Data1.csv file and has only 2 columns, Figure 1.

	A	B
1	t_f	y_f
2	20.02.2021	100
3	05.03.2021	200
4	20.03.2021	10
5	20.04.2021	10
6	20.05.2021	100
7	20.06.2021	10
8	20.07.2021	100

Figure 1: Data1.csv file

The data from this csv file can be read using the following lines of code:

```
filename = "./ Data1.csv"
MyData = pd.read_csv(filename, sep = ";", decimal=',')
print(MyData)
```

Preparing input arrays. Dates in datetime format are converted to days relative to the very first date.

```
t_f = pd.to_datetime(MyData.t_f.dropna(), format='%d.%m.%Y')
t_start = min(t_f[0],t_df[0],t_d2f[0],t_int_a[0])
t_f = np.array([(x-t_start).days for x in t_f])
y_f = MyData.y_f.dropna().to_numpy()
```

The number of spline knots can be several times greater than the number of observation points; in any case, the roughness penalty will prevent the function from being too smooth.

```
m = round(3 * len(t_f))
```

Calling a function.

```
r = FunctionalSmoothingSpline(t_f = t_f,
                              values_f = y_f,
                              knots_number = m,
                              alpha = 10**2,
                              All_Positive = False,
                              info = True)
```

Since the info argument was set to True when the function was called, the return value is a dictionary. To obtain x and y coordinates, we extract the corresponding data from the dictionary. Plotting a graph of a restored function, Figure 2.

```
x = r['x']
y = r['y']
plt.plot(x, y, color="red")
plt.xticks(ticks=t, labels=[(t_start + pd.Timedelta(tx, "d")).date() for tx in t_f],
               rotation='vertical')
```

```
plt.scatter(t_f,y_f)
plt.show()
```

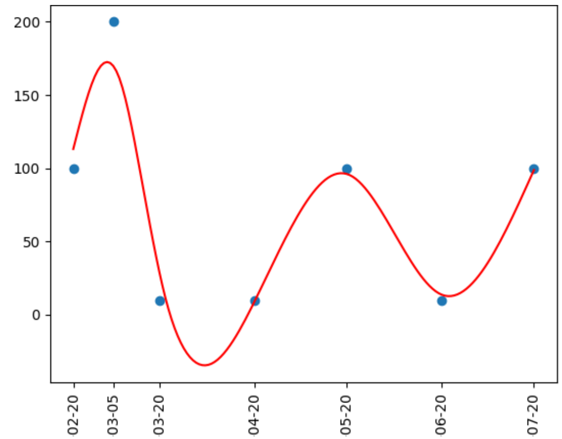


Figure 2: Restored function without the requirement for non-negativity

3.1.1. Providing non-negativity by taking the exponent from the spline (linearization). For this purpose it is enough to supply logarithms of observations. We reduced the smoothing coefficient by a factor of 2.3 because the logarithm of 200 is less than the value of 200 by a factor of 2.3 (logarithmization reduces errors). It is sufficient to make the following changes.

```
r = FunctionalSmoothingSpline(t_f = t_f,
                              values_f = np.log(y_f),
                              knots_number = m,
                              alpha = 10**2 / 2.3,
                              All_Positive = False,
                              info = True)
plt.plot(x, np.exp(y), color="red")
```

As a result, we have Figure 3.

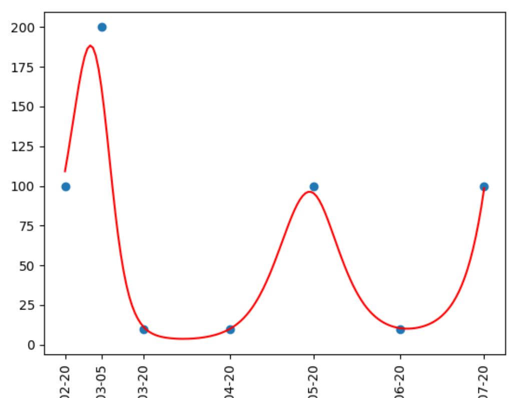


Figure 3: Restoration of non-negative function by taking exponent from spline and linearization

3.1.2. Providing non-negativity by taking the exponent from the spline (numerical optimization).

To calculate a spline with non-negative values, we have to change `All_Positive` argument from `False` to `True`, and select `method` as `"exp"`. Make the following changes in the code (the input is normal values, not logarithms, also plot graph by values, not by exponential from values). The result is shown in Figure 4.

```
r = FunctionalSmoothingSpline(t_f = t_f,
                              values_f = y_f,
                              knots_number = m,
                              alpha = 10**2,
                              All_Positive = True,
                              method = "exp",
                              info = True)
x = r['x']
y = r['y']
plt.plot(x, y, color="red")
plt.xticks(ticks=t, labels=[(t_start + pd.Timedelta(tx, "d")).date() for tx in t_f],
               rotation='vertical')
plt.scatter(t_f,y_f)
plt.show()
```

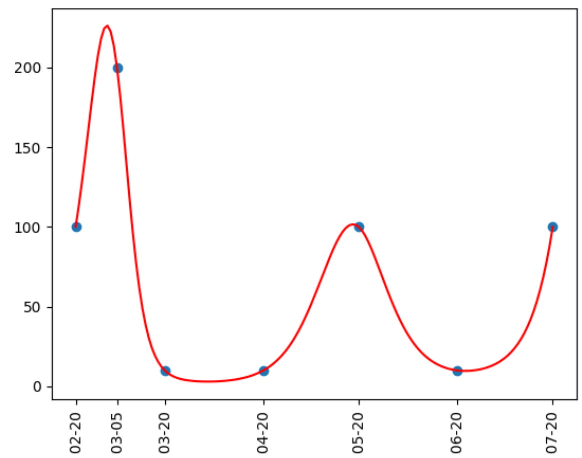


Figure 4: Restoration of non-negative function by taking exponent from spline and numerical optimization

3.1.3. Quadratic programming with non-negativity condition at knots

Set argument `All_Positive` to `True`, choose `method` one of the following `"Lemke"`, `"Lemke_njit"`, `"cvxopt"` (or do not specify argument `method` at all, the default will be `"Lemke"`). Make the following code changes.

```
r = FunctionalSmoothingSpline(t_f = t_f,
                              values_f = y_f,
                              knots_number = m,
                              alpha = 10**2,
                              All_Positive = True,
                              method = "Lemke",
                              info = True)
```

The result is shown in Figure 5.

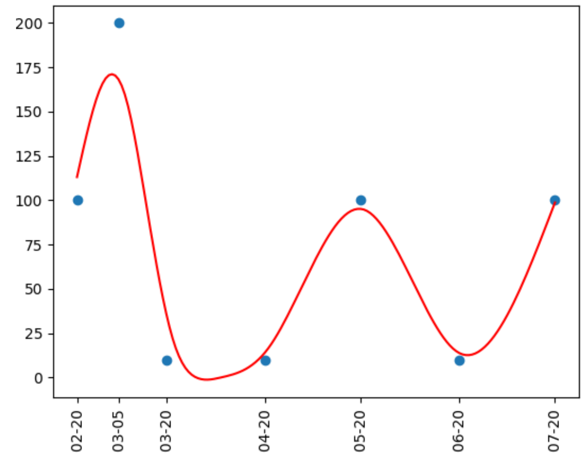


Figure 5: Restoration of non-negative function using quadratic programming

In the case of method `"Lemke_njit"`, the first call of the function will be long because the Lemke method is precompiled, but subsequent calls will be very fast because the precompiled machine code is executed. If method `"cvxopt"` is specified, the computation is also fast (package cvxopt uses an interior point algorithm to solve quadratic programming problems). In all 3 cases the reconstructed functions coincide.

3.1.4. Providing non-negativity using quadratic programming, adding extra knots

Set argument `add_knots` to `True`, `new_knots_tol` to 10^{-4}, max_added_knots to 10. Make the following code changes.

```
r = FunctionalSmoothingSpline(t_f = t_f,
                              values_f = y_f,
                              knots_number = m,
                              alpha = 10**2,
                              All_Positive = True,
                              method = "Lemke",
                              info = True,
                              add_knots = True,
                              new_knots_tol = 10 ** (-4),
                              max_added_knots = 10)
x = r['x']
y = r['y']
plt.plot(x, y, color="red")
t = t_f
plt.xticks(ticks=t, labels=[(t_start + pd.Timedelta(tx, "d")).date() for tx in t],
               rotation='vertical')
plt.scatter(t_f,y_f)
new_knots = r['new_knots']
old_knots = np.setdiff1d(r['knots'],new_knots)
print("new_knots = ", new_knots)
print("len(new_knots) = ", len(new_knots))
```

```
plt.scatter(old_knots, np.zeros(len(old_knots)), marker="+", c = "red")
plt.scatter(new_knots,np.zeros(len(new_knots)), marker = "+", c ="blue")
plt.show()
```

The result is shown in Figure 6.

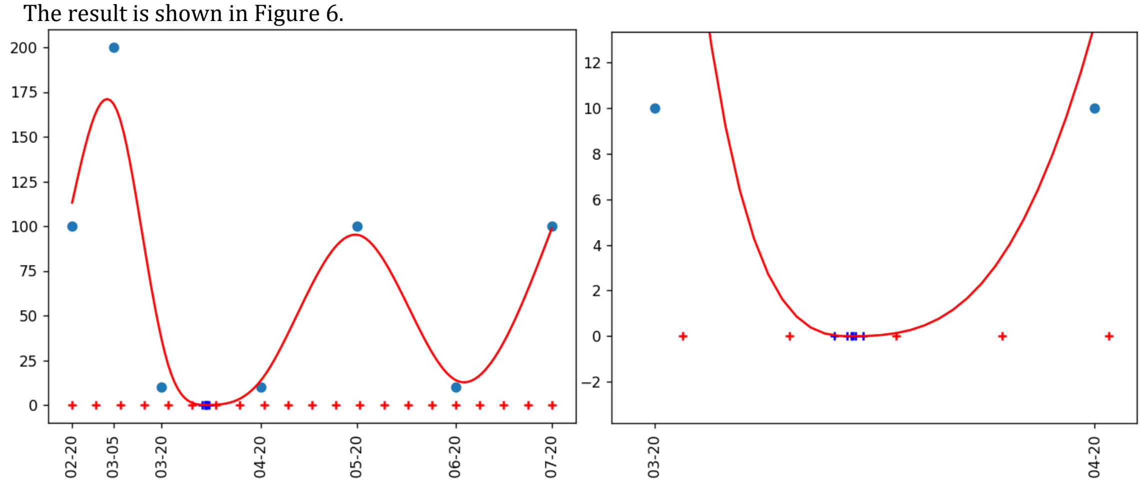


Figure 6: Restoration of non-negative function using quadratic programming with additional knots

3.1.5. Providing non-negativity using quadratic programming, adding extra conditions without knots

Set argument `add_condition_without_knots` to `True`, `new_knots_tol` to 10^{-4}, max_added_knots to 10. Make the following code changes.

```
r = FunctionalSmoothingSpline(t_f = t_f,
                              values_f = y_f,
                              knots_number = m,
                              alpha = 10**2,
                              All_Positive = True,
                              method = "Lemke",
                              info = True,
                              add_condition_without_knots = True,
                              new_knots_tol = 10 ** (-4),
                              max_added_knots = 10)
x = r['x']
y = r['y']
plt.plot(x, y, color="red")
t = t_f
plt.xticks(ticks=t, labels=[(t_start + pd.Timedelta(tx, "d")).date() for tx in t],
               rotation='vertical')
plt.scatter(t_f,y_f)
new_knots = r['new_knots']
old_knots = np.setdiff1d(r['knots'],new_knots)
print("new_knots = ", new_knots)
print("len(new_knots) = ", len(new_knots))
```

```
plt.scatter(old_knots, np.zeros(len(old_knots)), marker="+", c = "red")
plt.scatter(new_knots,np.zeros(len(new_knots)), marker = "+", c ="blue")
plt.show()
```

The result is shown in Figure 7.

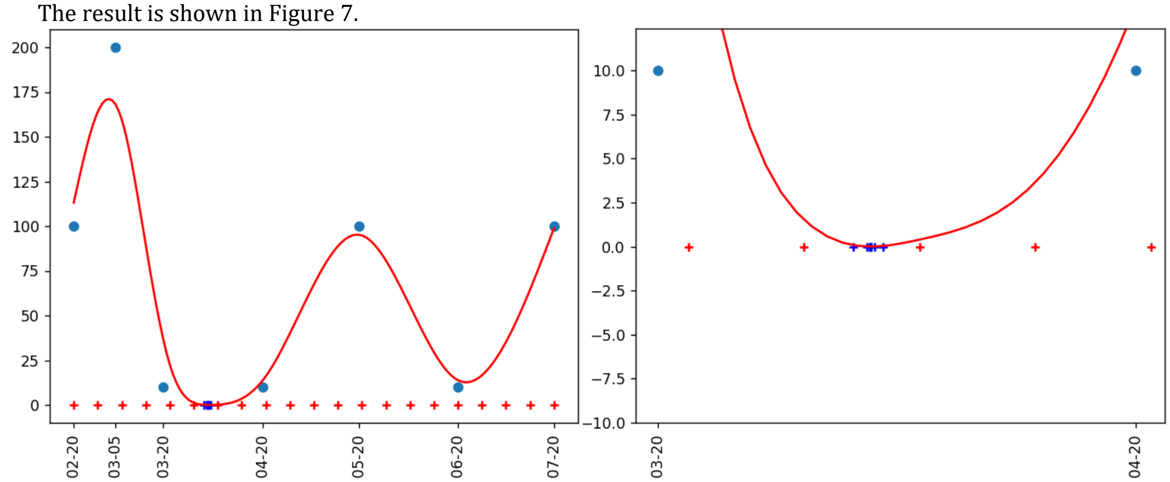


Figure 7: Restoration of non-negative function using quadratic programming with additional conditions between knows (without additional knots)

3.1.6. Returning values of first derivative, second derivative, and integral function

Set argument `output` to `["y", "dy", "d2y", "integral"]`, and make the following code changes.

```
r = FunctionalSmoothingSpline(t_f = t_f,
                              values_f = y_f,
                              knots_number = m,
                              alpha = 10**2,
                              All_Positive = True,
                              method = "Lemke",
                              info = True,
                              add_condition_without_knots = True,
                              new_knots_tol = 10 ** (-4),
                              max_added_knots = 10,
                              output=["y", "dy", "d2y", "integral"]

)
x = r['x']
y = r['y']
dy = r['dy']
d2y = r['d2y']
F = r['integral']
old_knots = r['knots']
new_knots = r['new_knots']
```

```
old_knots = np.setdiff1d(old_knots, new_knots)
plt.plot(x, y, color="red")
t = t_f
plt.xticks(ticks=t, labels=[(t_start + pd.Timedelta(tx, "d")).date() for tx in t],
           rotation='vertical')
plt.scatter(t_f, y_f)
plt.scatter(old_knots, np.zeros(len(old_knots)), marker="+", c="red")
plt.scatter(new_knots, np.zeros(len(new_knots)), marker="+", c="blue")
plt.show()

plt.plot(x, dy, color="orange")
plt.xticks(ticks=t, labels=[(t_start + pd.Timedelta(tx, "d")).date() for tx in t],
           rotation='vertical')
plt.scatter(old_knots, np.zeros(len(old_knots)), marker="+", c="red")
plt.scatter(new_knots, np.zeros(len(new_knots)), marker="+", c="blue")
plt.show()

plt.plot(x, d2y, color="magenta")
plt.xticks(ticks=t, labels=[(t_start + pd.Timedelta(tx, "d")).date() for tx in t],
           rotation='vertical')
plt.scatter(old_knots, np.zeros(len(old_knots)), marker="+", c="red")
plt.scatter(new_knots, np.zeros(len(new_knots)), marker="+", c="blue")
plt.show()

plt.plot(x, F, color="red")
plt.xticks(ticks=t, labels=[(t_start + pd.Timedelta(tx, "d")).date() for tx in t],
           rotation='vertical')
plt.scatter(old_knots, np.zeros(len(old_knots)), marker="+", c="red")
plt.scatter(new_knots, np.zeros(len(new_knots)), marker="+", c="blue")
plt.show()

print("new_knots = ", new_knots)
print("len(new_knots) = ", len(new_knots))
```

The result is shown in Figure 8.

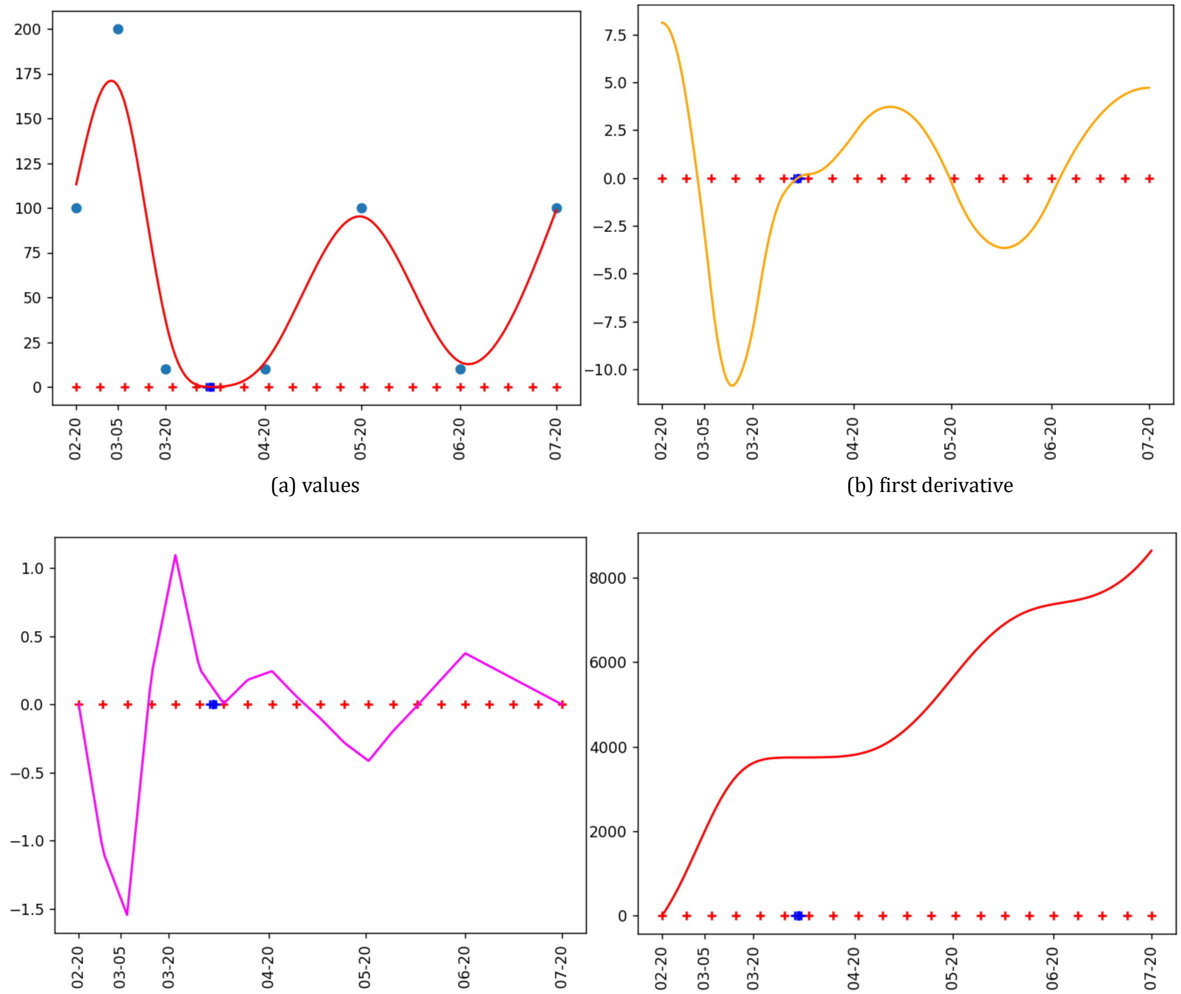


Figure 8: Restoration of non-negative function using quadratic programming

3.2 Example 2. Reconstructing a function by values, values of the first and second derivative.

In this example the function is reconstructed simultaneously by values, first and second derivatives simultaneously. We will pay more attention to the values of the first and second derivatives and assign them more weight (since their values are smaller than the values of the function)

Import standard Python libraries.

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
```

Data input.

Let's assume that the dataset is presented in a Data2.csv file and has all the required columns, Figure 9.

	A	B	C	D	E	F
1	t_f	y_f	t_df	y_df	t_d2f	y_d2f
2	20.02.2021	40	31.03.2021	5	19.05.2021	0
3	08.12.2021	20	07.05.2021	-15	01.09.2021	0
4	01.01.2022	0	06.07.2021	0		
5			11.08.2021	5		
6			29.09.2021	0		

Figure 9: Data2.csv file

The data from this csv file can be read using the following lines of code:

```
filename = "./Data2.csv"
MyData = pd.read_csv(filename, sep = ";", decimal=',')
print(MyData)
```

As a result, the following table (DataFrame) will be read, Figure 10.

```
          t_f   y_f         t_df  y_df       t_d2f  y_d2f
0  20.02.2021  40.0  31.03.2021     5  19.05.2021    0.0
1  08.12.2021  20.0  07.05.2021   -15  01.09.2021    0.0
2  01.01.2022   0.0  06.07.2021     0         NaN    NaN
3         NaN   NaN  11.08.2021     5         NaN    NaN
4         NaN   NaN  29.09.2021     0         NaN    NaN
```

Figure 10: Data read into the dataframe

Preparing input arrays. Dates in datetime format are converted to days relative to the very first date.

```
t_f = pd.to_datetime(MyData.t_f.dropna(), format='%d.%m.%Y')
t_df = pd.to_datetime(MyData.t_df.dropna(), format='%d.%m.%Y')
t_d2f = pd.to_datetime(MyData.t_d2f.dropna(), format='%d.%m.%Y')
t_start = min(min(t_f), min(t_df), min(t_d2f) )
t_f = np.array([(x-t_start).days for x in t_f])
t_df = np.array([(x-t_start).days for x in t_df])
t_d2f = np.array([(x-t_start).days for x in t_d2f])

y_f = MyData.y_f.dropna().to_numpy()
y_df = MyData.y_df.dropna().to_numpy()
y_d2f = MyData.y_d2f.dropna().to_numpy()
```

The number of spline knots can be several times greater than the number of observation points; in any case, the roughness penalty will prevent the function from being too smooth.

```
m = round(3*(len(t_f) + len(t_df) + len(t_d2f) ))
```

Calling a function. Here we increase the weight of the values of the first derivative by a factor of 10 and the values of the second derivative by a factor of 20.

```
r = FunctionalSmoothingSpline(t_f = t_f,
                              values_f = y_f,
                              t_df = t_df,
```

```
                              values_df = y_df,
                              coef_df = 10,
                              t_d2f = t_d2f,
                              values_d2f = y_d2f,
                              coef_d2f = 20,
                              knots_number = m,
                              alpha = 10**2,
                              All_Positive = False,
                              info = True)
```

Since the info argument was set to True when the function was called, the return value is a dictionary. To obtain x and y coordinates, we extract the corresponding data from the dictionary. Plotting a graph of a restored function, Figure 11.

```
x = r['x']
y = r['y']
plt.plot(x, y, color="red")
t = np.concatenate ((t_f, t_df , t_d2f))
plt.xticks(ticks=t, labels=[(t_start + pd.Timedelta(tx, "d")).date() for tx in t],
               rotation='vertical')
plt.scatter(t_f,y_f)
plt.scatter(t_df, y_df,marker="v", color="red")
plt.scatter(t_d2f, y_d2f,marker="d", color="magenta")
plt.show()
```

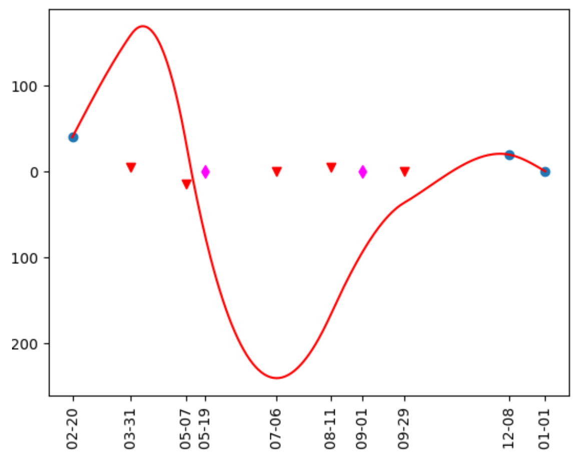


Figure 11: Restored function

The values of the function are known only at the very beginning and end. In the middle, the function is defined only through the values of the first and second derivative. As a result, the function may fall below zero, which is not always desirable in practical problems.

3.2.1. Providing non-negativity by taking the exponent from the spline (numerical optimization).

To calculate a spline with non-negative values, all we have to change `All_Positive` argument from `False` to `True`, and select `method` as `"exp"`. Make the following changes in the code.

```
r = FunctionalSmoothingSpline(t_f = t_f,
                              values_f = y_f,
                              t_df = t_df,
                              values_df = y_df,
                              coef_df = 10,
                              t_d2f = t_d2f,
                              values_d2f = y_d2f,
                              coef_d2f = 20,
                              knots_number = m,
                              alpha = 10**2,
                              All_Positive = False,
                              method="exp",
                              info = True)
```

The result is shown in Figure 12.

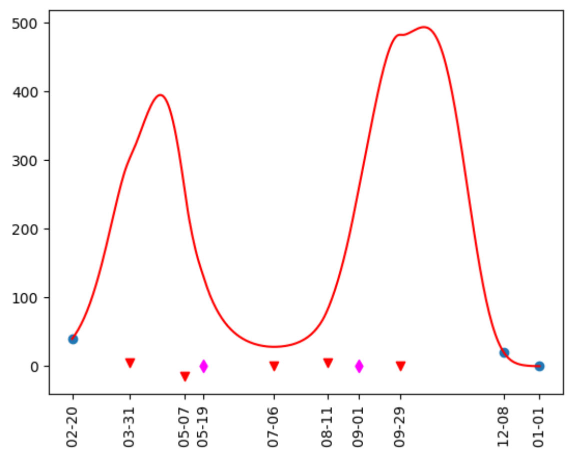


Figure 12: Restoration of non-negative function by taking exponent from spline and numerical optimization

3.2.2. Quadratic programming with non-negativity condition at knots

Set argument `All_Positive` to `True`, choose `method` one of the following `"Lemke"`, `"Lemke_njit"`, `"cvxopt"` (or do not specify argument `method` at all, the default will be `"Lemke"`). Make the following code changes.

```
r = FunctionalSmoothingSpline(t_f = t_f,
                              values_f = y_f,
                              t_df = t_df,
                              values_df = y_df,
                              coef_df = 10,
                              t_d2f = t_d2f,
                              values_d2f = y_d2f,
                              coef_d2f = 20,
```

```
                              knots_number = m,
                              alpha = 10**2,
                              All_Positive = True,
                              method="Lemke",
                              info = True)
```

The result is shown in Figure 13.

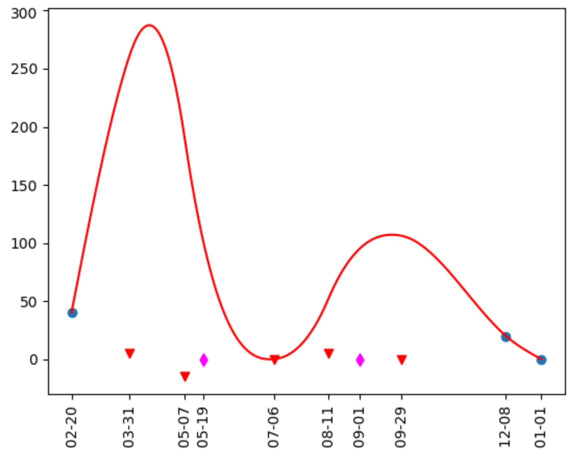


Figure 13: Restoration of non-negative function using quadratic programming

3.2.3. Providing non-negativity using quadratic programming, adding extra knots

Set argument `add_knots` to `True`, if needed add `new_knots_tol` or `max_added_knots`. Make the following code changes.

```
r = FunctionalSmoothingSpline(t_f = t_f,
                              values_f = y_f,
                              t_df = t_df,
                              values_df = y_df,
                              coef_df = 10,
                              t_d2f = t_d2f,
                              values_d2f = y_d2f,
                              coef_d2f = 20,
                              knots_number = m,
                              alpha = 10**2,
                              All_Positive = True,
                              method = "Lemke",
                              info = True,
                              add_knots = True
                              new_knots_tol = 10 ** (-4),
                              max_added_knots = 10,
)
```

Add these lines to visualize the knots

```
new_knots = r['new_knots']
```

```
old_knots = np.setdiff1d(r['knots'],new_knots)
print("new_knots = ", new_knots)
print("len(new_knots) = ", len(new_knots))
plt.scatter(old_knots, np.zeros(len(old_knots)), marker="+", c = "red")
plt.scatter(new_knots,np.zeros(len(new_knots)), marker = "+", c ="blue")
```

The result is shown in Figure 14.

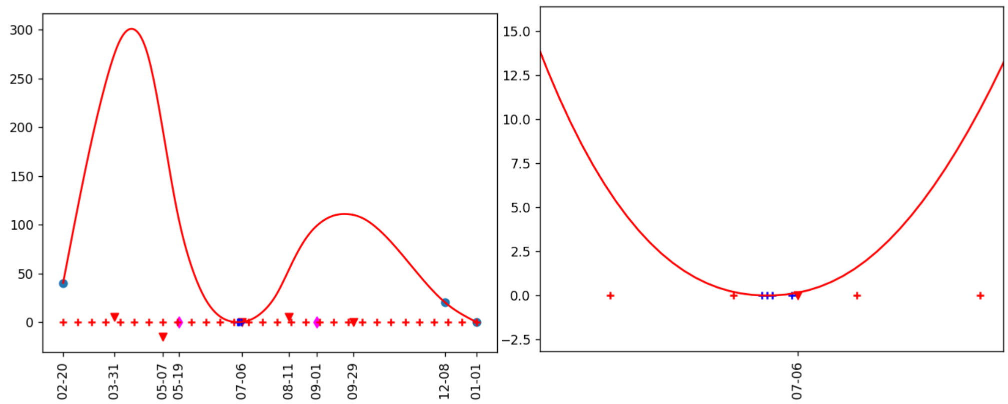


Figure 14: Restoration of non-negative function using quadratic programming with additional knots

3.2.4. Providing non-negativity using quadratic programming, adding extra conditions without knots

Set argument `add_condition_without_knots` to `True`, if needed add `new_knots_tol` or `max_added_knots`. Make the following code changes.

```
r = FunctionalSmoothingSpline(t_f = t_f,
                              values_f = y_f,
                              t_df = t_df,
                              values_df = y_df,
                              coef_df = 10,
                              t_d2f = t_d2f,
                              values_d2f = y_d2f,
                              coef_d2f = 20,
                              knots_number = m,
                              alpha = 10**2,
                              All_Positive = True,
                              method="Lemke",
                              info = True,
                              add_condition_without_knots = True
                              new_knots_tol = 10 ** (-4),
                              max_added_knots = 10,
)
```

The result is shown in Figure 15 (blue + indicates new condition positions, no new knots added).

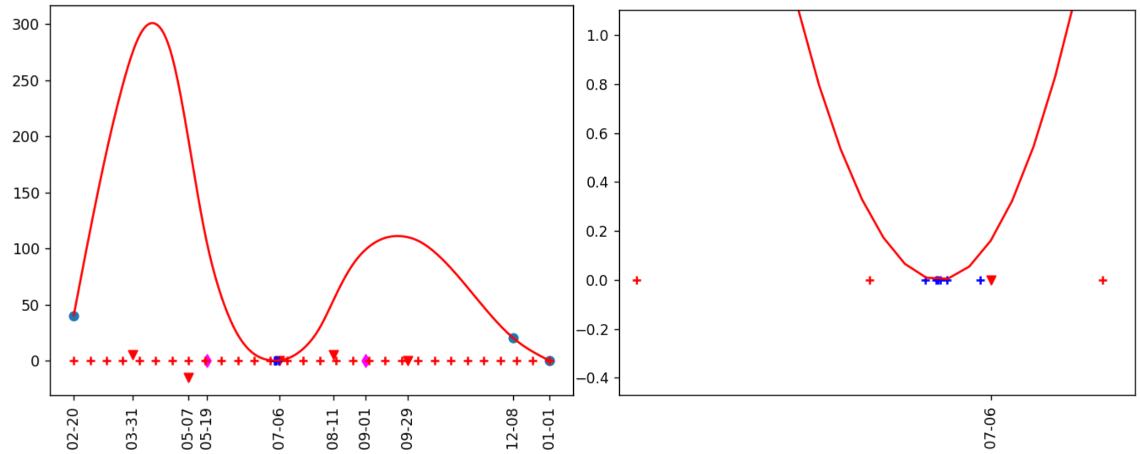


Figure 15: Restoration of non-negative function using quadratic programming with additional conditions between knows (without additional knots)

3.3 Example 3. Reconstructing a function by values, values of the first and second derivative and values of definite integrals.

In this example, the original function $f(t)$ itself is positive. The data allow us to restore this function as positive without additional non-negativity conditions. If we use the same data to reconstruct the function with additional non-negativity conditions, the reconstructed function coincides on the graph with the reconstructed function without these additional conditions (the difference does not exceed 0.03 in absolute values, can't be noticed on the graph). Since information about integrals is used no other approaches discussed above, except for the approach based on quadratic programming, are suitable.

Import standard Python libraries.

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
```

Data input.

Let's assume that the data set is presented in a Data3.csv file and has all the required columns, Figure 16.

	A	B	C	D	E	F	G	H	I
1	t_f	y_f	t_df	y_df	t_d2f	y_d2f	t_int_a	t_int_b	y_int
2	20.02.2021	100	31.03.2021	0	19.05.2021	0	25.03.2021	24.04.2021	3999,916
3	08.12.2021	20	07.05.2021	-1,74967			21.10.2021	20.11.2021	2281,669
4	01.01.2022	0	06.07.2021	0					
5			11.08.2021	1,552171					
6			29.09.2021	0					

Figure 16: Data2.csv file

The data from this csv file can be read using the following lines of code:

```
filename = "./Data3.csv"
MyData = pd.read_csv(filename, sep = ";", decimal=',')
with pd.option_context('display.max_rows', None, 'display.max_columns', None):
   print(MyData)
```

As a result, the following table (DataFrame) will be read, Figure 17.

	t_f	y_f	t_df	y_df	t_d2f	y_d2f	t_int_a	t_int_b	y_int
0	20.02.2021	100.0	31.03.2021	0.000000	19.05.2021	0.0	25.03.2021	24.04.2021	3999.916
1	08.12.2021	20.0	07.05.2021	-1.749674	NaN	NaN	21.10.2021	20.11.2021	2281.669
2	01.01.2022	0.0	06.07.2021	0.000000	NaN	NaN	NaN	NaN	NaN
3	NaN	NaN	11.08.2021	1.552171	NaN	NaN	NaN	NaN	NaN
4	NaN	NaN	29.09.2021	0.000000	NaN	NaN	NaN	NaN	NaN

Figure 17: Data read into the dataframe

Preparing input arrays. Dates in datetime format are converted to days relative to the very first date.

```
t_f = pd.to_datetime(MyData.t_f.dropna(), format='%d.%m.%Y')
t_df = pd.to_datetime(MyData.t_df.dropna(), format='%d.%m.%Y')
t_d2f = pd.to_datetime(MyData.t_d2f.dropna(), format='%d.%m.%Y')
t_int_a = pd.to_datetime(MyData.t_int_a.dropna(), format='%d.%m.%Y')
t_int_b = pd.to_datetime(MyData.t_int_b.dropna(), format='%d.%m.%Y')
t_start = min(min(t_f), min(t_df), min(t_d2f), min(t_int_a))
t_f = np.array([(x-t_start).days for x in t_f])
t_df = np.array([(x-t_start).days for x in t_df])
t_d2f = np.array([(x-t_start).days for x in t_d2f])
t_int_a = np.array([(x-t_start).days for x in t_int_a])
t_int_b = np.array([(x-t_start).days for x in t_int_b])

y_f = MyData.y_f.dropna().to_numpy()
y_df = MyData.y_df.dropna().to_numpy()
y_d2f = MyData.y_d2f.dropna().to_numpy()
y_int = MyData.y_int.dropna().to_numpy()
```

The number of spline knots can be several times greater than the number of observation points; in any case, the roughness penalty will prevent the function from being too smooth.

```
m = round(3*(len(t_f) + len(t_df) + len(t_d2f) + len(t_int_a)))
```

Calling a function.

```
r = FunctionalSmoothingSpline(t_f = t_f,
                              values_f = y_f,
                              t_df = t_df,
                              values_df = y_df,
                              t_d2f = t_d2f,
                              values_d2f = y_d2f,
                              t_int_a = t_int_a,
                              t_int_b = t_int_b,
                              values_int = y_int,
                              knots_number = m,
                              alpha = 10**1,
                              All_Positive = False,
                              info = True)
```

Plotting a graph of a restored function, Figure 18.

```
x = r['x']
y = r['y']
plt.plot(x, y, color="red")
t = np.concatenate((t_f, t_df, t_d2f,t_int_a,t_int_b))
plt.xticks(ticks=t, labels=[(t_start + pd.Timedelta(tx, "d")).date() for tx in t],
               rotation='vertical')
plt.scatter(t_f,y_f)
plt.scatter(t_df, y_df,marker="v", color="red")
plt.scatter(t_d2f, y_d2f,marker="d", color="magenta")
plt.plot([t_int_a[0],t_int_b[0]], [0,0], '-g', linewidth=3)
plt.plot([t_int_a[1],t_int_b[1]], [0,0], '-g', linewidth=3)
plt.show()
```

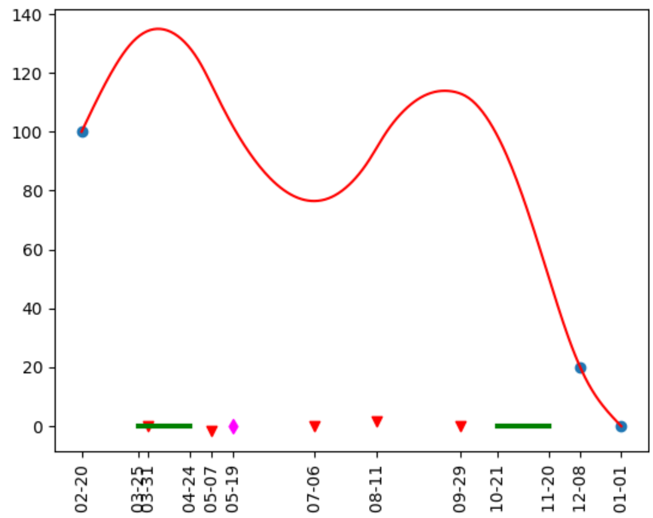


Figure 18: Restored function

3.4 Practical example. Sequences of integrals.

Data input.

Now we have online shopping purchases data expressed in dollars, file Sales1.csv contains only integrals data (sequences of integrals), Figure 19.

```
filename = "./Sales1.csv"
MyData = pd.read_csv(filename, sep = ";", decimal=',')
print(MyData)
```

	A	B
1	t	Values
2	01.12.2009	21,95
3	17.12.2009	5,7
4	18.01.2010	18,75
5	05.03.2010	15,25
6	23.03.2010	8,5
7	07.06.2010	7,95
8	13.10.2010	19,3
9	12.12.2010	31,3
10	21.01.2011	8,25
11	31.03.2011	3,9
12	17.04.2011	2,1
13	28.04.2011	15,27
14	07.07.2011	9,2

Figure 19: Integrals data

Preparing input arrays. Since this is a sequence of integrals, for the last value there is no time point for the upper limit of integration. We discard the last value of the integral. We again select the number of spline knots to be several times greater than the number of observations.

```
t = pd.to_datetime(MyData.t.dropna(), format='%d.%m.%Y')
t_start = min(t)
t = np.array([(x-t_start).days for x in t])
Y = MyData.Values.dropna().to_numpy()
n = len(t)
Y = Y[0:(n-1)]
m = round(3*n)
```

Function call.

```
y = FunctionalSmoothingSpline(t_int_a = t[0:(n-1)],
                              t_int_b = t[1:n],
                              values_int = Y,
                              knots_number = m,
                              alpha = 10**5,
                              All_Positive = False,
                              info = False)
```

Since the info argument is False, an array of spline values is returned, not a dictionary. Therefore, to plot a graph, we need to prepare an array of x points. For clarity, we will also depict a step function calculated as the average values of the function $y_i/(t_{i+1} - t_i)$, Figure 20.

```
x = np.append(np.arange(t[0],t[-1],1), t[-1])
plt.plot(x, y, color = "red")
x2 = np.array([])
y2 = np.array([])
for i in range(n-1):
    x2 = np.append(x2,[ t[i], t[i+1] ])
    y2 = np.append(y2, [ Y[i]/(t[i+1]-t[i]), Y[i]/(t[i+1]-t[i]) ] )
plt.plot(x2,y2,color="grey")
```

```
plt.xticks(ticks=t, labels=[(t_start+pd.Timedelta(tx,"d")).date() for tx in t],
           rotation='vertical')
plt.show()
```

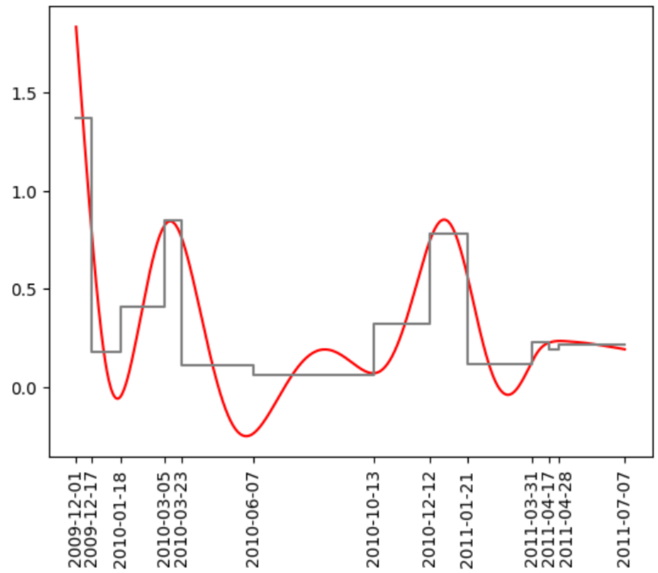


Figure 20: Integrals data (All_Positive = False)

To calculate a spline with non-negative values, it is enough to change `All_Positive` argument as True (`add_condition_without_knots` as True for good result). Executing the previously presented code will produce the following output, Figure 21.

```
y = FunctionalSmoothingSpline(t_int_a = t[0:(n-1)],
                              t_int_b = t[1:n],
                              values_int = Y,
                              knots_number = m,
                              alpha = 10**5,
                              All_Positive = False,
                              add_condition_without_knots = True,
                              info = False)
```

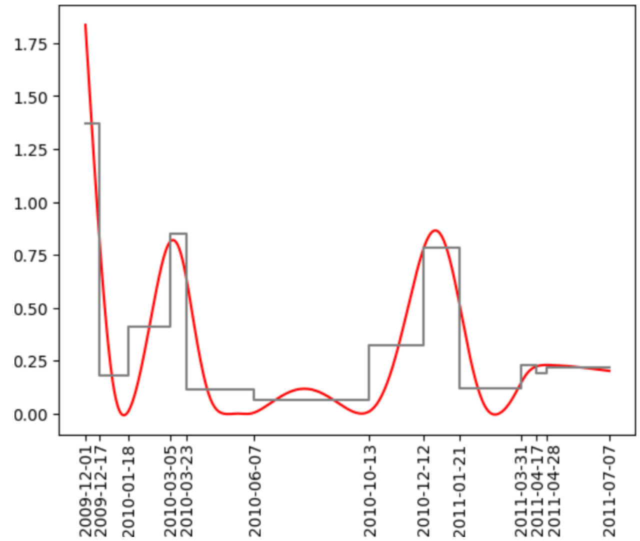


Figure 21: Integrals data (All_Positive = True)

To calculate the integral function $F(x)$, it is enough to set argument `output` as `"integral"`. The result, along with changing `All_Positive` argument from False to True, is shown in Figures 22 and 23.

```
y = FunctionalSmoothingSpline(t_int_a = t[0:(n-1)],
                              t_int_b = t[1:n],
                              values_int = Y,
                              knots_number = m,
                              alpha = 10**5,
                              All_Positive = False,
                              output = "integral",
                              info = False)
```

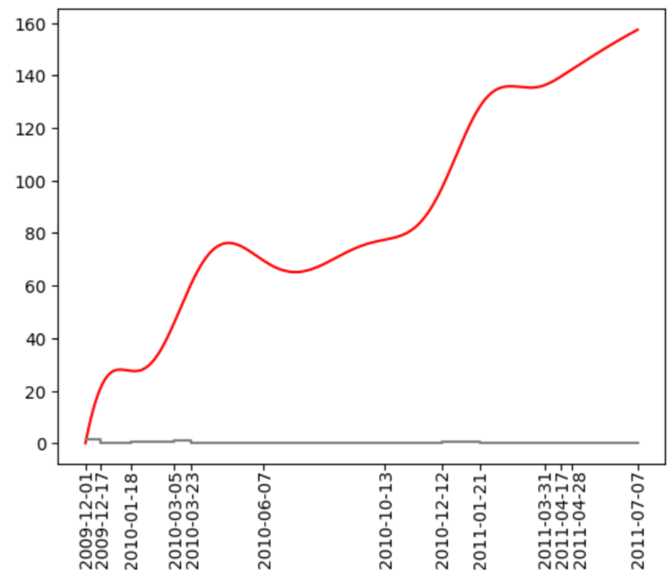


Figure 22: Integrals data (All_Positive = False, output = "integral")

```
y = FunctionalSmoothingSpline(t_int_a = t[0:(n-1)],
                              t_int_b = t[1:n],
                              values_int = Y,
                              knots_number = m,
                              alpha = 10**5,
                              All_Positive = True,
                              output = "integral",
                              info = False,
                              add_condition_without_knots = True,
)
```

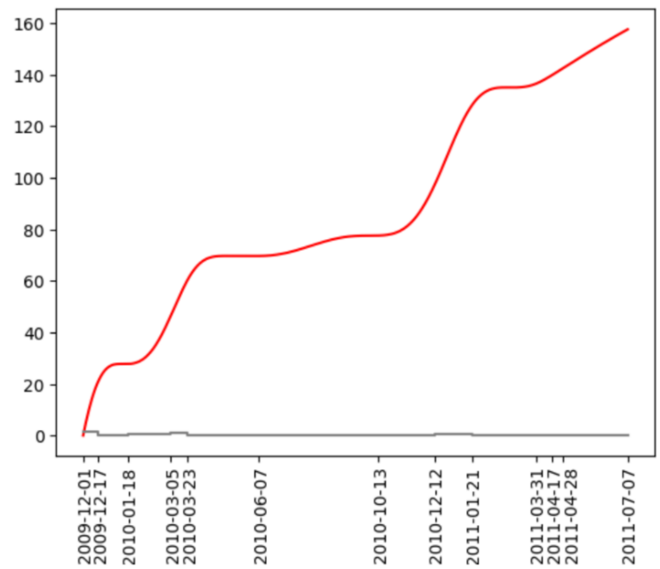
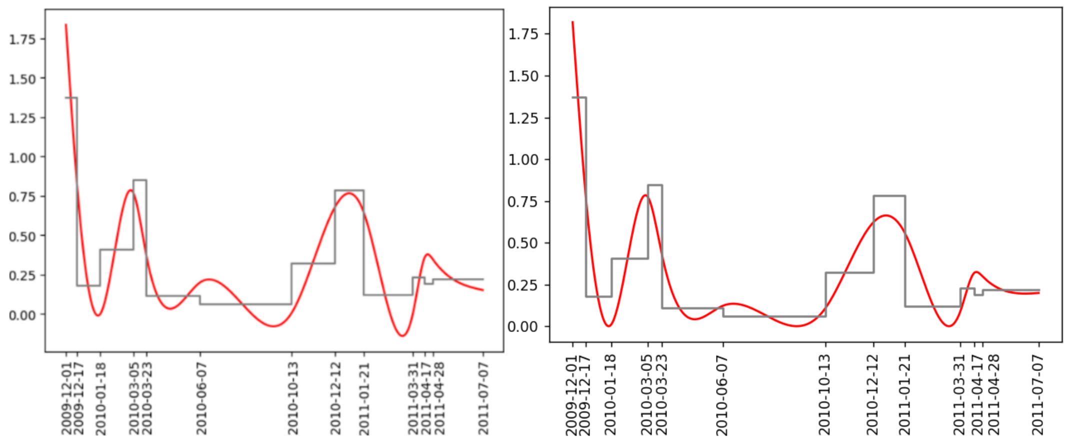


Figure 23: Integrals data (All_Positive = True, output = "integral")

To specify spline knots ourselves, we must specify argument `knots` as an array of values. In this case, the argument `knots_number` is ignored and determined by the length of the array `knots`. An example where spline knots are specified at observation points `t` is shown in Figure 24a. Please note that the positivity condition applies only to function at spline knots, so the function can take negative values between knots (that's why we previously specified a larger number of knots than observation points, or added additional knots or conditions without knots). With additional conditions, the function remains non-negative between knots, Figure 24b.

```
y = FunctionalSmoothingSpline(t_int_a = t[0:(n-1)], t_int_b = t[1:n], values_int = Y,
                              knots = t,
                              alpha = 10**5,
                              All_Positive = True,
                              info = False,
                              add_condition_without_knots = False)
```



(a) (b)

Figure 24: Integrals data (knots = t, All_Positive = True). The knots coincide with observations.
a) add_condition_without_knots = False, between knots function can take negative values;
b) add_condition_without_knots = True, function is non-negative between knots;

To assign weights to specific observations, simply define a one-dimensional array of weights and pass it as a function argument. For the example just given above, we can give the central peak a small weight, thereby significantly changing the form of the function, Figure 25.

```
W = np.ones(n-1)
W[7] = 0.01
y = FunctionalSmoothingSpline(t_int_a = t[0:(n-1)], t_int_b = t[1:n], values_int = Y,
                              knots = t,
                              weights_int = W,
                              alpha = 10**5,
                              All_Positive = True,
                              info = False,
                              add_condition_without_knots = True
                              )
```

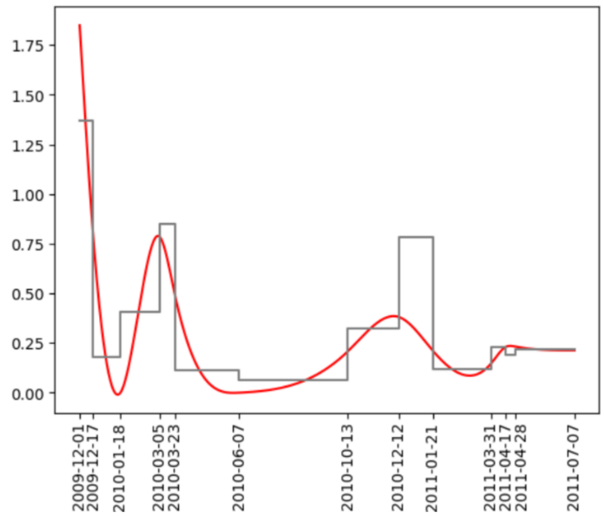


Figure 25: Integrals data, small weight for the 7th observation (knots = t, All_Positive = True)

As mentioned earlier, by default, spline values are calculated at every point between the first and last observations. To calculate a spline at other points, it is only necessary to define the vector x, Figure 26.

```
x = np.append(np.arange(t[0],t[-1],17), t[-1])
y = FunctionalSmoothingSpline(t_int_a = t[0:(n-1)],
                              t_int_b = t[1:n],
                              values_int = Y,
                              knots = t,
                              weights_int = W,
                              x = x,
                              alpha = 10**5,
                              All_Positive = True,
                              info = False,
                              add_condition_without_knots = True)
```

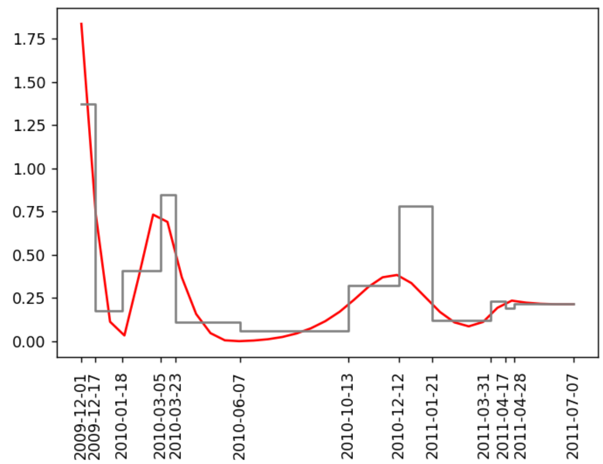


Figure 26: Calculating spline at every 17 points (knots = t, All_Positive = True)