TSSL Lab 1 - Autoregressive models

We load a few packages that are useful for solvign this lab assignment.

In [1]:
```
import pandas  # Loading data / handling data frames
import numpy as np
import matplotlib.pyplot as plt
from sklearn import linear_model as lm  # Used for solving linear regressio
from sklearn.neural_network import MLPRegressor # Used for NAR model

from tssltools_lab1 import acf, acfplot # Module available in LISAM - Used
```

1.1 Loading, plotting and detrending data

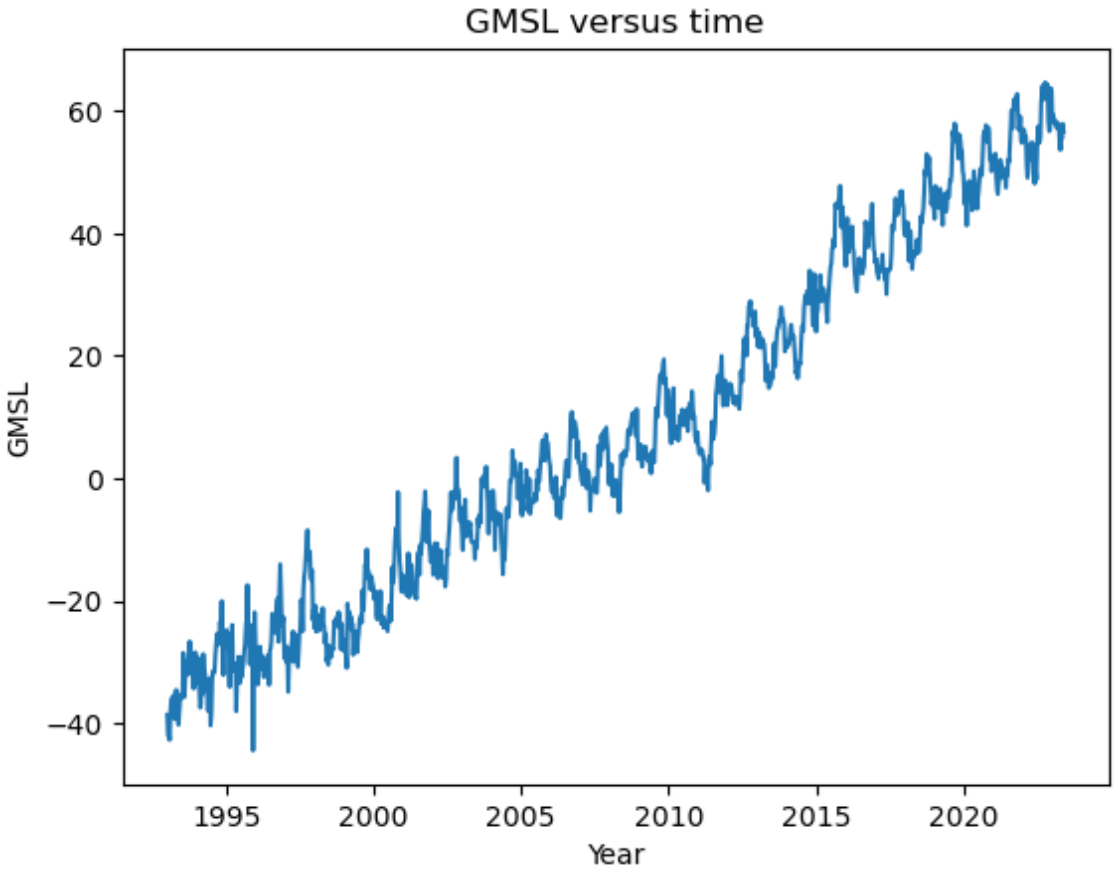
In this lab we will build autoregressive models for a data set corresponding to the Global Mean Sea Level (GMSL) over the past few decades. The data is taken from https://climate.nasa.gov/vital-signs/sea-level/ (https://climate.nasa.gov/vital-signs/sea-level/) and is available on LISAM in the file `sealevel.csv` .

Q1: Load the data and plot the GMSL versus time. How many observations are there in total in this data set?

Hint: With pandas you can use the function `pandas.read_csv` to read the csv file into a data frame. Plotting the time series can be done using `pyplot` . Note that the sea level data is stored in the 'GMSL' column and the time when each data point was recorded is stored in the column 'Year'.

A1: There are 1119 observations in total in this data set.

In [2]:
```
sealevel=pandas.read_csv("sealevel.csv")
#print(sealevel)

plt.plot(sealevel.Year,sealevel.GMSL)
plt.title('GMSL versus time')
plt.xlabel('Year')
plt.ylabel('GMSL')
plt.show()
```



Q2: The data has a clear upward trend. Before fitting an AR model to this data need to remove this trend. Explain, using one or two sentences, why this is necessary.

A2: The AR model assumes stationarity, the upward trend time series is not stationary, which will lead to the larger and larger mean and variance as time proceed, leading to inaccurate forecasting results.

Q3 Detrend the data following these steps:

1. Fit a straight line, $\mu_t = \theta_0 + \theta_1 u_t$ to the data based on the method of least squares. Here, u_t is the time point when obervation t was recorded.

   Hint: You can use `lm.LinearRegression().fit(...)` from scikit-learn. Note that the inputs need to be passed as a 2D array.

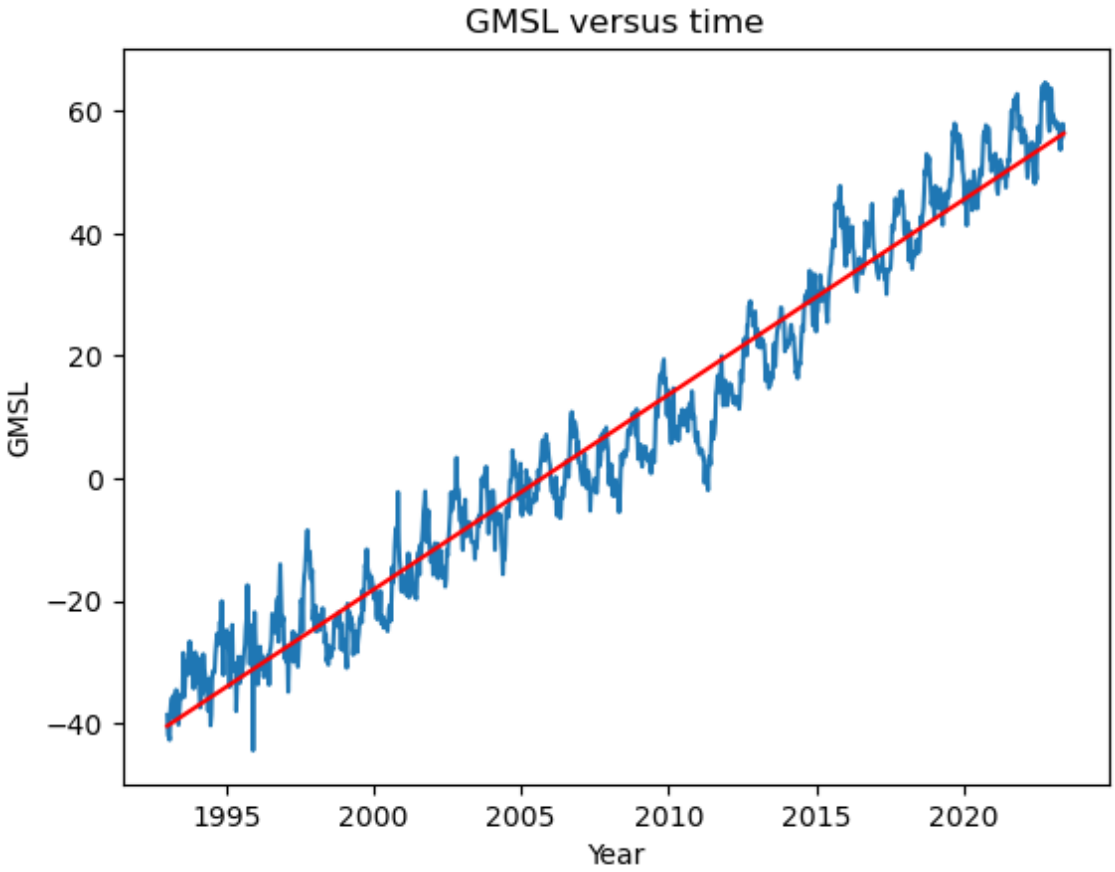
   Before going on to the next step, plot your fitted line and the data in one figure.
2. Subtract the fitted line from y_t for the whole data series and plot the deviations from the straight line.

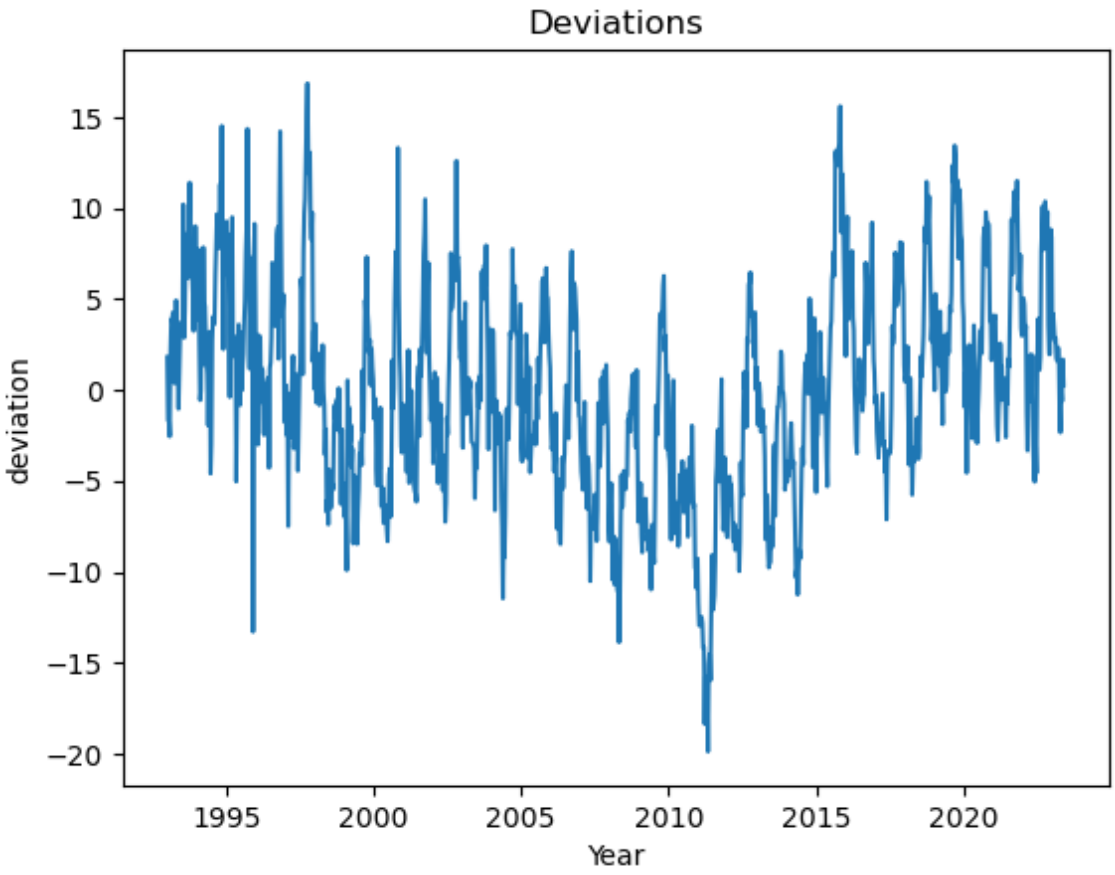
From now, we will use the detrended data in all parts of the lab.

Note: The GMSL data is recorded at regular time intervals, so that $u_{t+1} - u_t = \text{const}$. Therefore, you can just as well use t directly in the linear regression function if you prefer, $\mu_t = \theta_0 + \theta_1 t$.

A3:

In [3]:
```
# Fit the straight line
X=np.asarray(sealevel.Year).reshape(-1, 1)
Y=np.asarray(sealevel.GMSL).reshape(-1, 1)
linreg=lm.LinearRegression().fit(X,Y)
predict_line=linreg.predict(X)

# Plot
plt.plot(X,sealevel.GMSL)
plt.plot(X,predict_line,c='r')
plt.title('GMSL versus time')
plt.xlabel('Year')
plt.ylabel('GMSL')
plt.show()

# Subtract the fitted line from y_t
new_Y=Y-predict_line

# Plot the deviations
plt.plot(X,new_Y)
plt.title('Deviations')
plt.xlabel('Year')
plt.ylabel('deviation')
plt.show()
```





Q4: Split the (detrended) time series into training and validation sets. Use the values from the beginning up to the 700th time point (i.e. y_t for $t = 1$ to $t = 700$) as your training data, and the rest of the values as your validation data. Plot the two data sets.

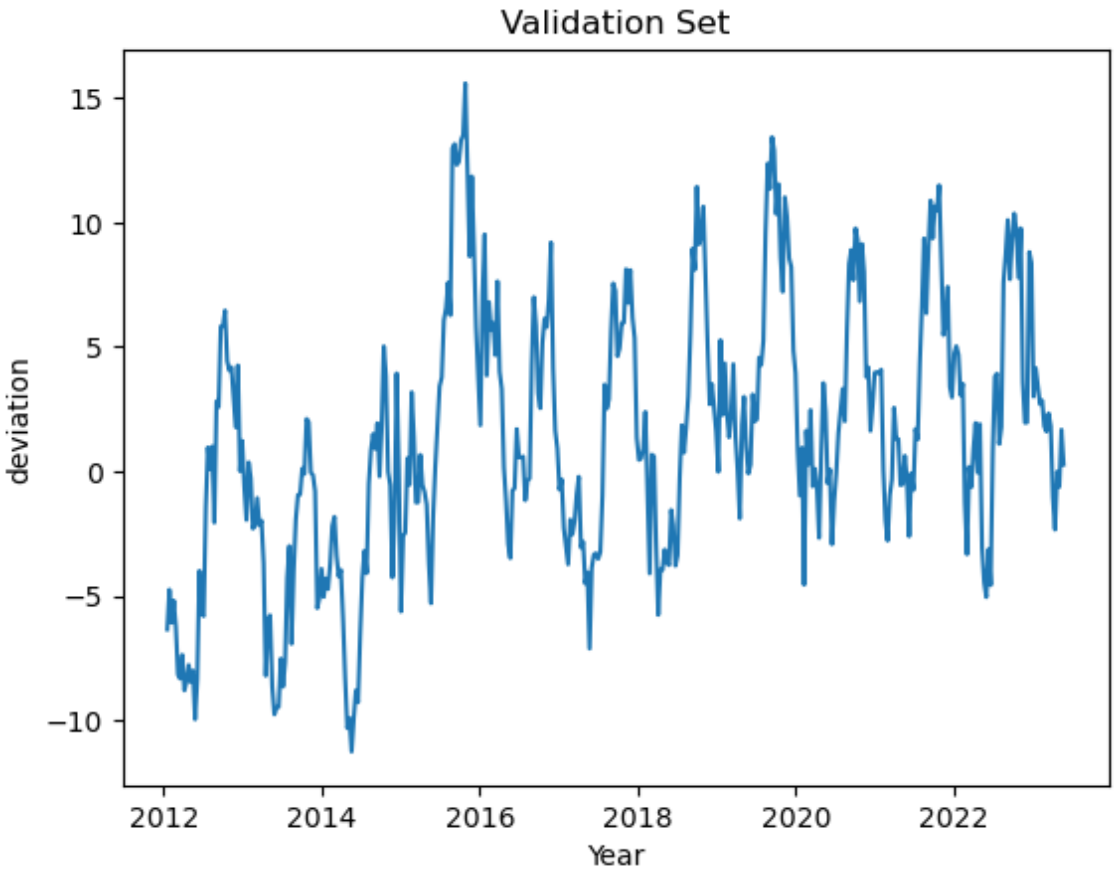
Note: In the above, we have allowed ourselves to use all the available data (train + validation) when detrending. An alternative would be to use only the training data also when detrending the model. The latter approach is more suitable if, either:

- we view the linear detrending as part of the model choice. Perhaps we wish to compare different polynomial trend models, and evaluate their performance on the validation data, or
- we wish to use the second chunk of observations to estimate the performance of the final model on unseen data (in that case it is often referred to as "test data" instead of "validation data"), in which case we should not use these observations when fitting the model, including the detrending step.

In this laboration we consider the linear detrending as a predetermined preprocessing step and therefore allow ourselves to use the validation data when computing the linear trend.

A4:

In [4]:
```
# Split the (detrended) time series into training and validation sets.
train_X=X[0:700]
train_Y=new_Y[0:700]
valid_X=X[700:]
valid_Y=new_Y[700:]

# Plot the training set
plt.plot(train_X,train_Y)
plt.title('Training Set')
plt.xlabel('Year')
plt.ylabel('deviation')
plt.show()

# Plot the Validation set
plt.plot(valid_X,valid_Y)
plt.title('Validation Set')
plt.xlabel('Year')
plt.ylabel('deviation')
plt.show()
```



1.2 Fit an autoregressive model

We will now fit an AR(p) model to the training data for a given value of the model order p.

Q5: Create a function that fits an AR(p) model for an arbitrary value of p. Use this function to fit a model of order $p = 10$ to the training data and write out (or plot) the coefficients.

Hint: Since fitting an AR model is essentially just a standard linear regression we can make use of `lm.LinearRegression().fit(...)` similarly to above. You may use the template below and simply fill in the missing code.

A5:

In [122]:
```
def fit_ar(y, p):
    """Fits an AR(p) model. The loss function is the sum of squared errors

    :param y: array (n,), training data points
    :param p: int, AR model order
    :return theta: array (p,), learnt AR coefficients
    """

    # Number of training data points
    n = y.shape[0]

    # Construct the regression matrix
    Phi = np.zeros(((n-p),p))
    for j in range(p):
        Phi[:,j] =y[(p-j-1):(n-j-1)]

    # Drop the first p values from the target vector y
    yy = y[p:]  # yy = (y_{t+p+1}, ..., y_n)

    # Here we use fit_intercept=False since we do not want to include an in
    regr = lm.LinearRegression(fit_intercept=False)
    regr.fit(Phi,yy)

    return regr.coef_
```

In [126]:
```
p=10
theta=fit_ar(train_Y.flatten(),p)
print(theta)
```

```
[ 0.63068183  0.1231388   0.12558768  0.17683292 -0.02284342 -0.07140349
 -0.05693816  0.0479181  -0.0893176   0.0251526 ]
```

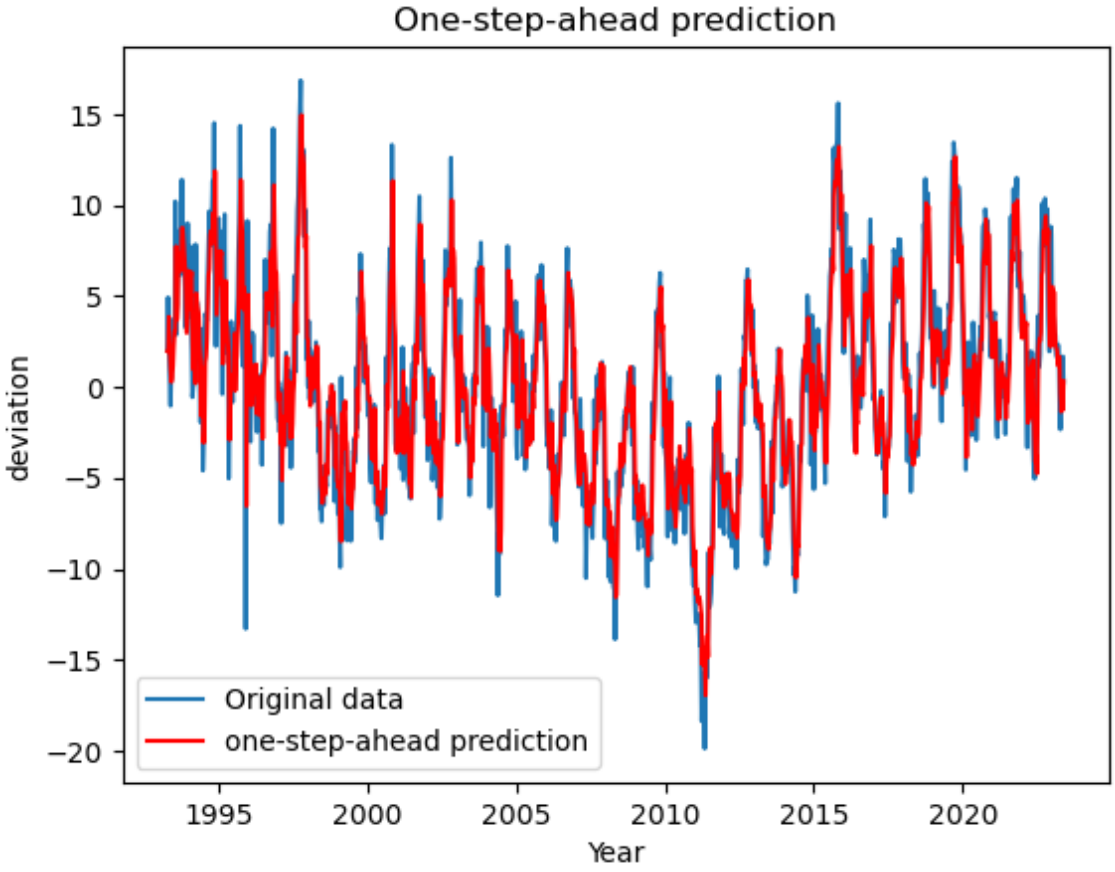
Q6: Next, write a function that computes the one-step-ahead prediction of your fitted model. 'One-step-ahead' here means that in order to predict y_t at $t = t_0$, we use the actual values of y_t for $t < t_0$ from the data. Use your function to compute the predictions for both *training data* and *validation data*. Plot the predictions together with the data (you can plot both training and validation data in the same figure). Also plot the *residuals*.

Hint: It is enought to call the predict function once, for both training and validation data at the same time.

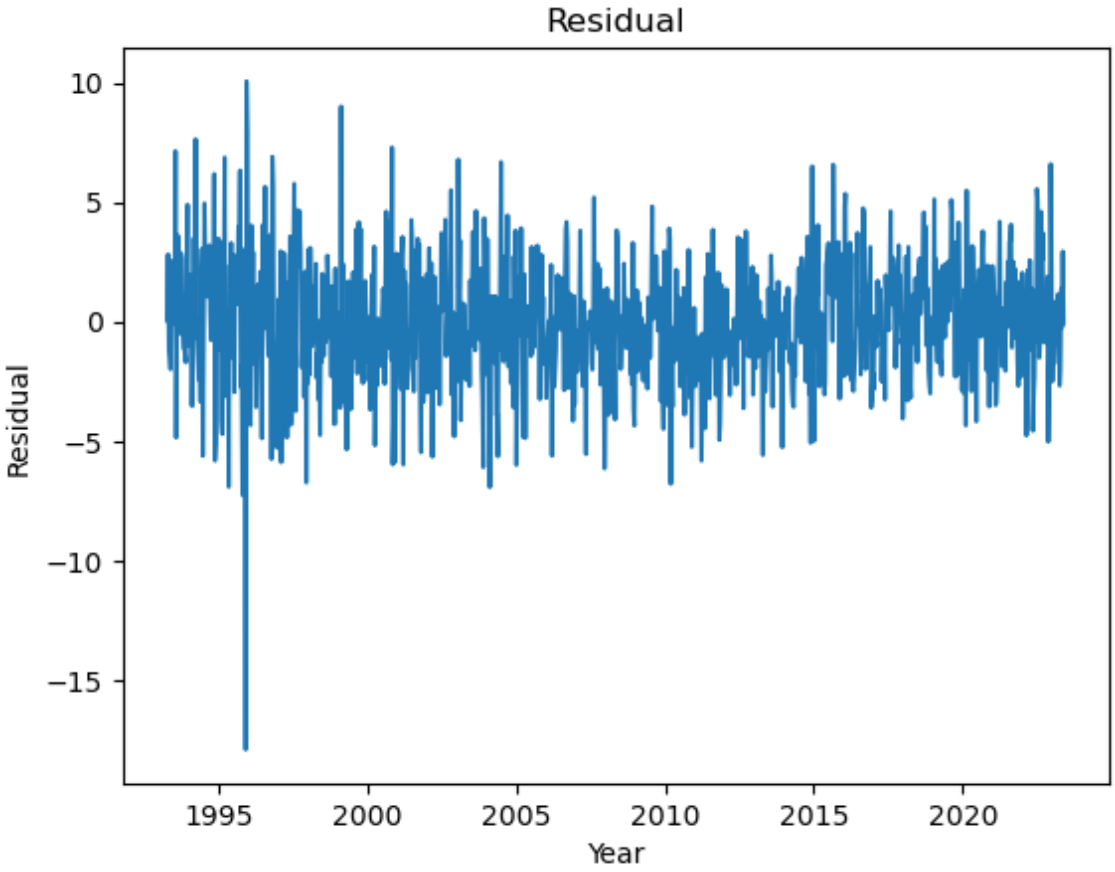
A6:

In [127]:
```
def predict_ar_1step(theta, y_target):
    """Predicts the value y_t for t = p+1, ..., n, for an AR(p) model, base
    one-step-ahead prediction.

    :param theta: array (p,), AR coefficients, theta=(a1,a2,...,ap).
    :param y_target: array (n,), the data points used to compute the predic
    :return y_pred: array (n-p,), the one-step predictions (\hat y_{p+1}, .
    """

    n = len(y_target)
    p = len(theta)

    # Number of steps in prediction
    m = n-p
    y_pred = np.zeros(m)

    for i in range(m):
        # <COMPLETE THIS CODE BLOCK>
        y_pred[i] =np.dot(np.flip(y_target[i:(i+p)]),theta).sum()

    return y_pred
```

In [128]:
```
# Predict
oneStepAheadPredict=predict_ar_1step(theta, new_Y.flatten())

# Plot the prediction
plt.plot(X[p:],new_Y[p:],label="Original data")
plt.plot(X.flatten()[p:],oneStepAheadPredict,c='r',label="one-step-ahead pr
plt.title('One-step-ahead prediction')
plt.xlabel('Year')
plt.ylabel('deviation')
plt.legend()
plt.show()
```



In [9]:
```
# Plot the residual
plt.plot(X.flatten()[p:],new_Y.flatten()[p:]-oneStepAheadPredict)
plt.title('Residual')
plt.xlabel('Year')
plt.ylabel('Residual')
plt.show()
```



Q7: Compute and plot the autocorrelation function (ACF) of the *residuals* only for the *validation data*. What conclusions can you draw from the ACF plot?

Hint: You can use the function `acfplot` from the `tssltools` module, available on the course web page.

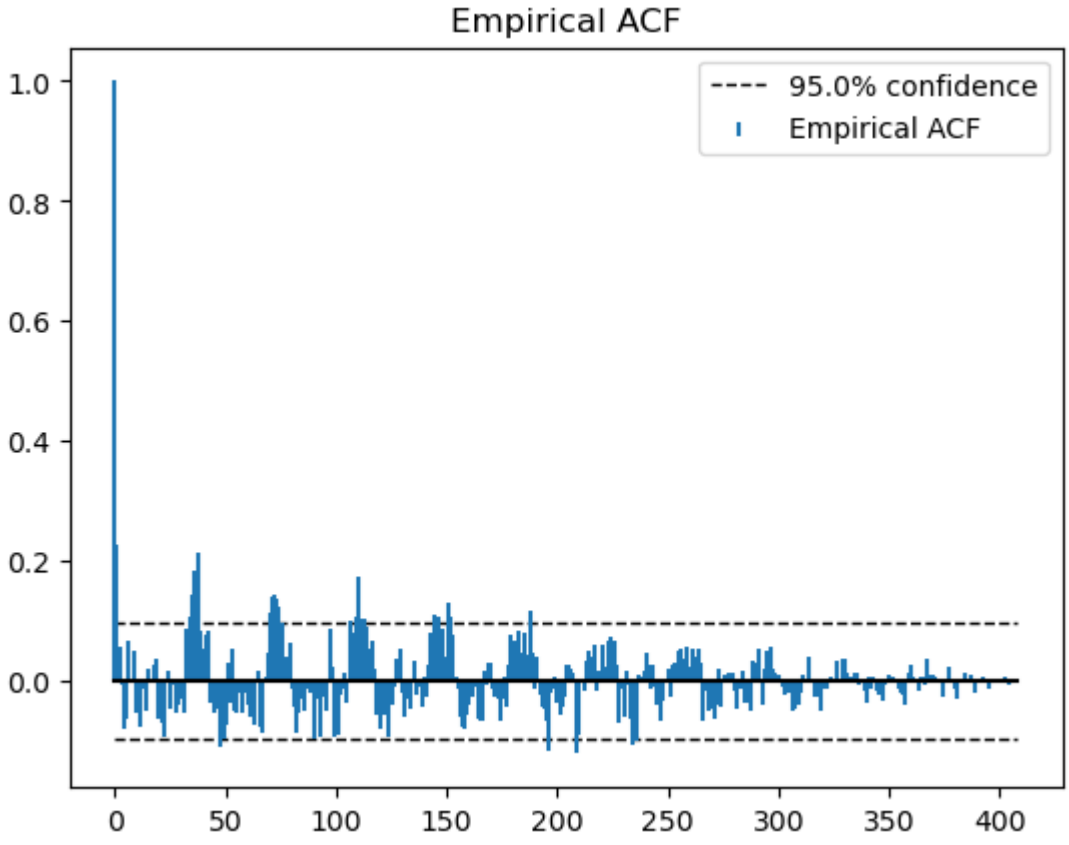
A7: ACF (AutoCorrelationFunction) describes the correlation between different time points. If the residuals of the AR model show significant autocorrelation at certain lag orders, this may indicate that the model still has some information that has not been captured and needs further improvement.

As can be seen from the above figure, the model needs to go through a certain lag and the fitting will be better.

In [10]:
```
help(acfplot)
```

```
Help on function acfplot in module tssltools_lab1:

acfplot(x, lags=None, conf=0.95)
    Plots the empirical autocorralation function.

    :param x: array (n,), sequence of data points
    :param lags: int, maximum lag to compute the ACF for. If None, this is
set to n-1. Default is None.
    :param conf: float, number in the interval [0,1] which specifies the c
onfidence level (based on a central limit
                 theorem under a white noise assumption) for two dashed li
nes drawn in the plot. Default is 0.95.
    :return:
```

In [11]:
```
oneStepAheadPredictValid=predict_ar_1step(theta, valid_Y.flatten())
residuals=valid_Y.flatten()[10:]-oneStepAheadPredictValid
acfplot(residuals)
```



Type *Markdown* and LaTeX: α^2

1.3 Model validation and order selection

Above we set the model order $p = 10$ quite arbitrarily. In this section we will try to find an appropriate order by validation.

Q8: Write a loop in which AR-models of orders from $p = 2$ to $p = 150$ are fitted to the data above. Plot the training and validation mean-squared errors for the one-step-ahead predictions versus the model order.

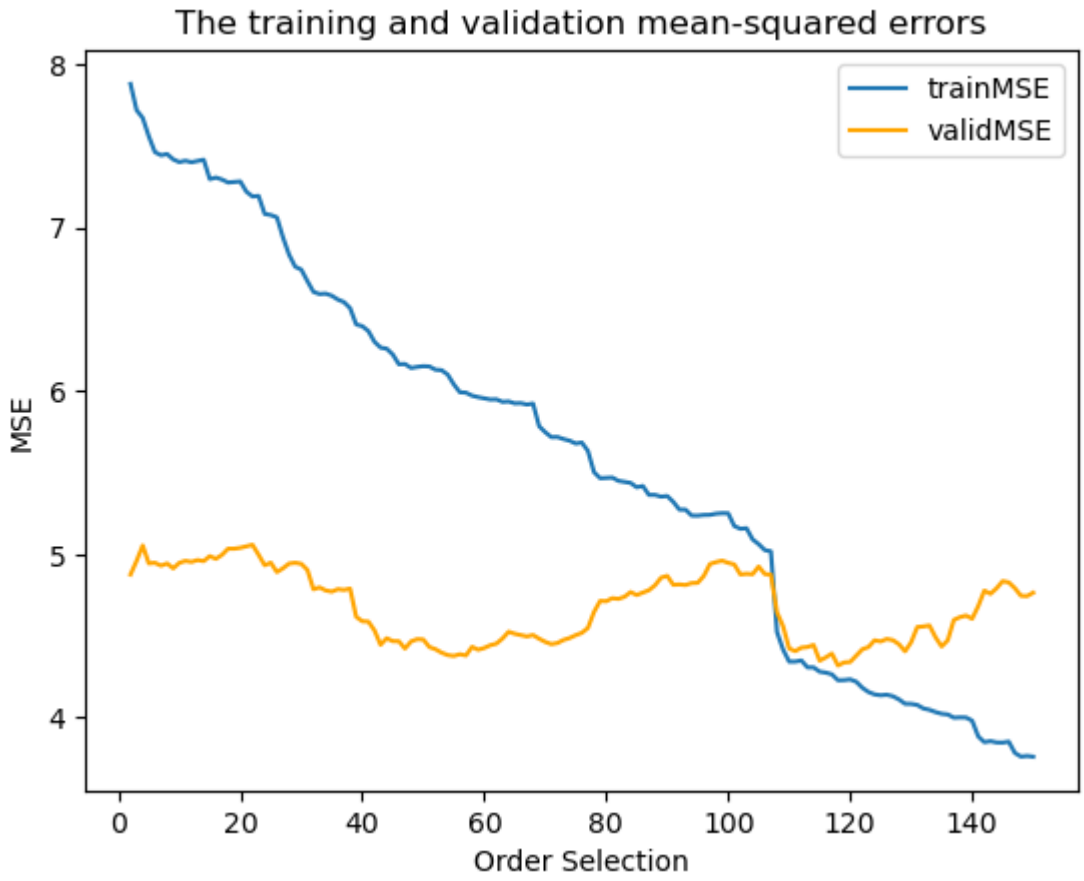
Based on your results:

- What is the main difference between the changes in training error and validation error as the order increases?
- Based on these results, which model order would you suggest to use and why?

Note: There is no obvious "correct answer" to the second question, but you still need to pick an order an motivate your choice!

A8: I suggest to use 118 as model order, because we get the minimum validation MSE.

In [12]:
```
orderSelection=np.arange(2, 151, 1)
trainMSE=np.zeros(149)
validMSE=np.zeros(149)

for i in range(149):
    p=i+2
    theta=fit_ar(train_Y.flatten(),p)

    trainOneStepAheadPredict=predict_ar_1step(theta, train_Y.flatten())
    trainResidual=train_Y.flatten()[p:]-trainOneStepAheadPredict
    trainMSE[i]=np.mean((trainResidual)**2)

    validOneStepAheadPredict=predict_ar_1step(theta, valid_Y.flatten())
    validResidual=valid_Y.flatten()[p:]-validOneStepAheadPredict
    validMSE[i]=np.mean((validResidual)**2)

plt.plot(orderSelection,trainMSE,label='trainMSE')
plt.plot(orderSelection,validMSE,label='validMSE',color='orange')
plt.title('The training and validation mean-squared errors')
plt.xlabel('Order Selection')
plt.ylabel('MSE')
plt.legend()
plt.show()

best_order=np.argmin(validMSE)+2
print("The best model order:",best_order)
```




```
The best model order: 118
```

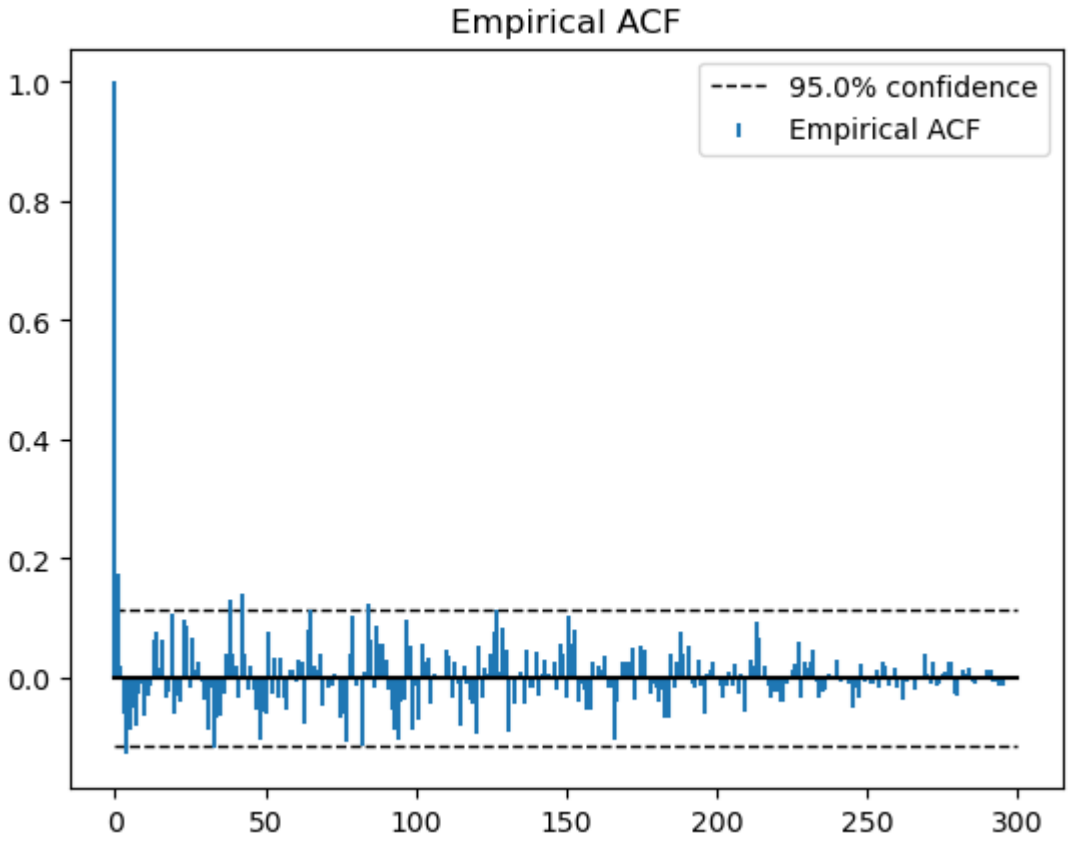
In [ ]:

Type *Markdown* and LaTeX: α^2

Q9: Based on the chosen model order, compute the residuals of the one-step-ahead predictions on the *validation data*. Plot the autocorrelation function of the residuals. What conclusions can you draw? Compare to the ACF plot generated above for p=10.

A9: Compare to the ACF plot generated for p=10,the ACF when p=118 has smaller Empirical ACF in general, which shows the model for p=118 fit the data better.

In [13]:
```
bestTheta=fit_ar(train_Y.flatten(),118)

validOneStepAheadPredict=predict_ar_1step(bestTheta, valid_Y.flatten())
validResidual=valid_Y.flatten()[118:]-validOneStepAheadPredict

acfplot(validResidual)
```



Type *Markdown* and LaTeX: α^2

1.4 Long-range predictions

So far we have only considered one-step-ahead predictions. However, in many practical applications it is of interest to use the model to predict further into the future. For intance, for the sea level data studied in this laboration, it is more interesting to predict the level one year from now, and not just 10 days ahead (10 days = 1 time step in this data).

Q10: Write a function that simulates the value of an AR(p) model m steps into the future, conditionally on an initial sequence of data points. Specifically, given $y_{1:n}$ with $n \geq p$ the function/code should predict the values

$$\hat{y}_{t|n} = \mathbb{E}[y_t | y_{1:n}], \qquad t = n+1, \ldots, n+m.$$

Use this to predict the values for the validation data ($y_{701:997}$) conditionally on the training data ($y_{1:700}$) and plot the result.

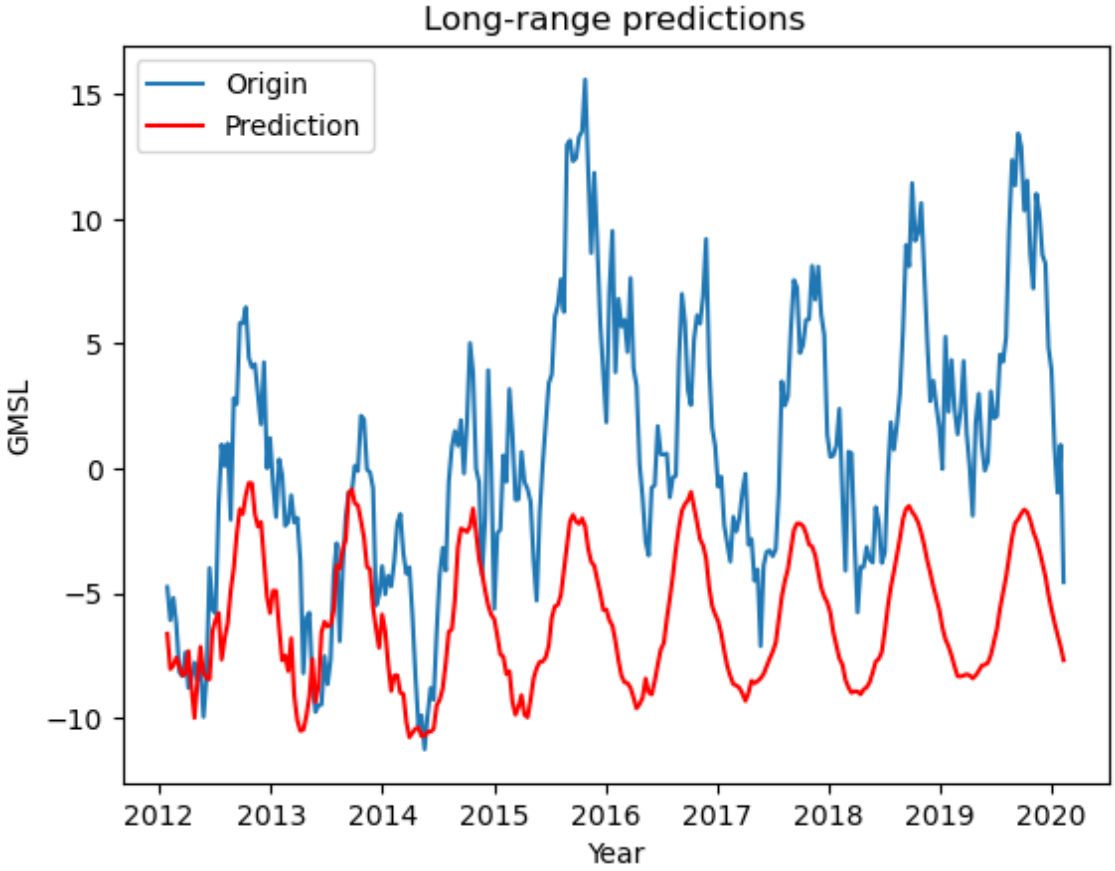
Hint: Use the pseudo-code derived at the first pen-and-paper session.

A10:

In [62]:
```
def simulate_ar(y, theta, m):
    """Simulates an AR(p) model for m steps, with initial condition given b

    :param y: array (n,) with n>=p. The last p values are used to initializ
    :param theta: array (p,). AR model parameters,
    :param m: int, number of time steps to simulate the model for.
    """

    p = len(theta)
    y_sim = np.zeros(m)
    phi = np.flip(y[-p:].copy()) # (y_{n-1}, ..., y_{n-p})^T - note that y[
    phi = np.squeeze(np.asarray(phi))

    for i in range(m):
        y_sim[i] = (phi*theta).sum()
        # <COMPLETE THIS CODE BLOCK>
        phi=np.append(np.flip(phi),y_sim[i])
        phi=np.flip(phi[1:])
    return y_sim
```

In [63]:
```
theta=fit_ar(train_Y.flatten(),118)
sim_Y=simulate_ar(train_Y,theta,297)
print(sim_Y.shape)
plt.plot(X[701:998],new_Y[701:998],label='Origin')
plt.plot(X[701:998],sim_Y,label='Prediction',color='r')
plt.title('Long-range predictions')
plt.xlabel('Year')
plt.ylabel('GMSL')
plt.legend()
plt.show()
```

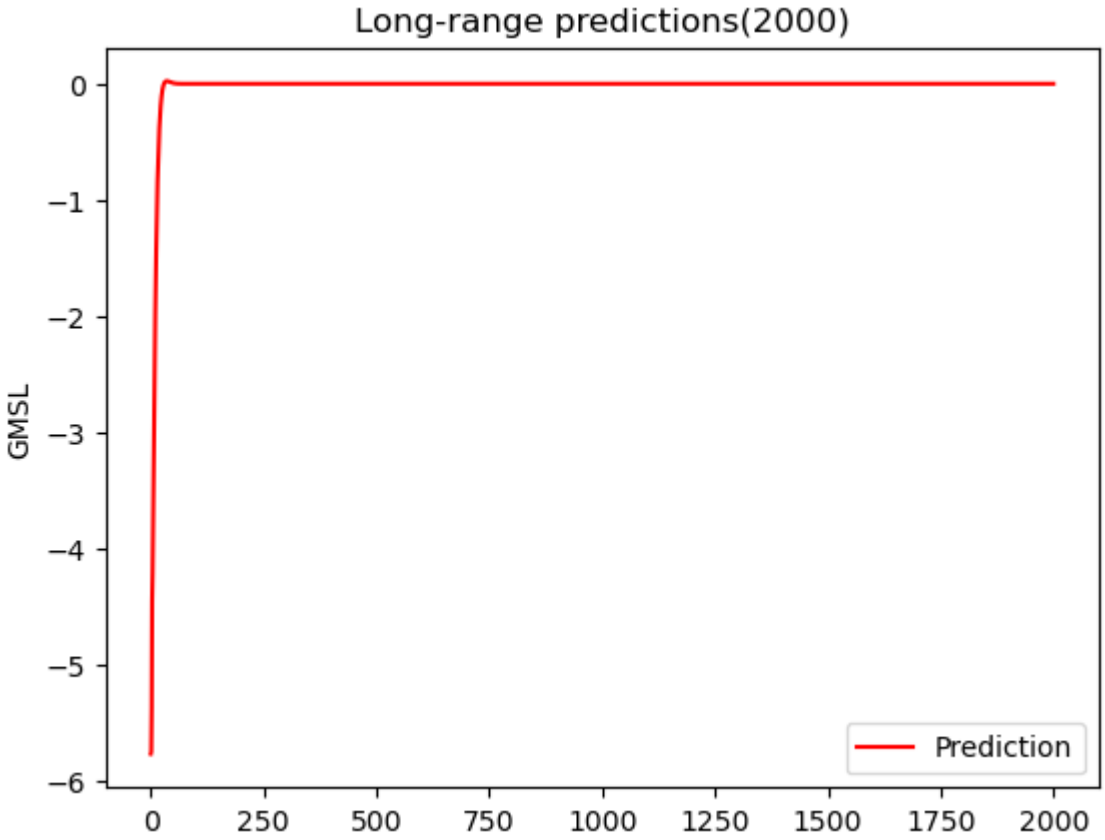
```
(297,)
```



Q11: Using the same function as above, try to simulate the process for a large number of time steps (say, $m = 2000$). You should see that the predicted values eventually converge to a constant prediction of zero. Is this something that you would expect to see in general? Explain the result.

A11: for the stationary AR model, |a|<1, which means when t increase, the mean value converge to 0 (the variance may be a constant). So this is what we expect to see in general.

In [64]:
```
theta=fit_ar(train_Y.flatten(),10)
sim_Y_2000=simulate_ar(train_Y,theta,2000)
plt.plot(sim_Y_2000,label='Prediction',color='r')
plt.title('Long-range predictions(2000)')
plt.ylabel('GMSL')
plt.legend()
plt.show()
```



1.5 Nonlinear AR model

In this part, we switch to a nonlinear autoregressive (NAR) model, which is based on a feedforward neural network. This means that in this model the recursive equation for making predictions is still in the form $\hat{y}_t = f_\theta(y_{t-1}, \ldots, y_{t-p})$, but this time f is a nonlinear function learned by the neural network. Fortunately almost all of the work for implementing the neural network and training it is handled by the `scikit-learn` package with a few lines of code, and we just need to choose the right structure, and prepare the input-output data.

Q12: Construct a NAR(p) model with a feedforward (MLP) network, by using the `MLPRegressor` class from `scikit-learn`. Set p to the same value as you chose for the linear AR model above. Initially, you can use an MLP with a single hidden layer consisting of 10 hidden neurons. Train it using the same training data as above and plot the one-step-ahead predictions as well as the residuals, on both the training and validation data.

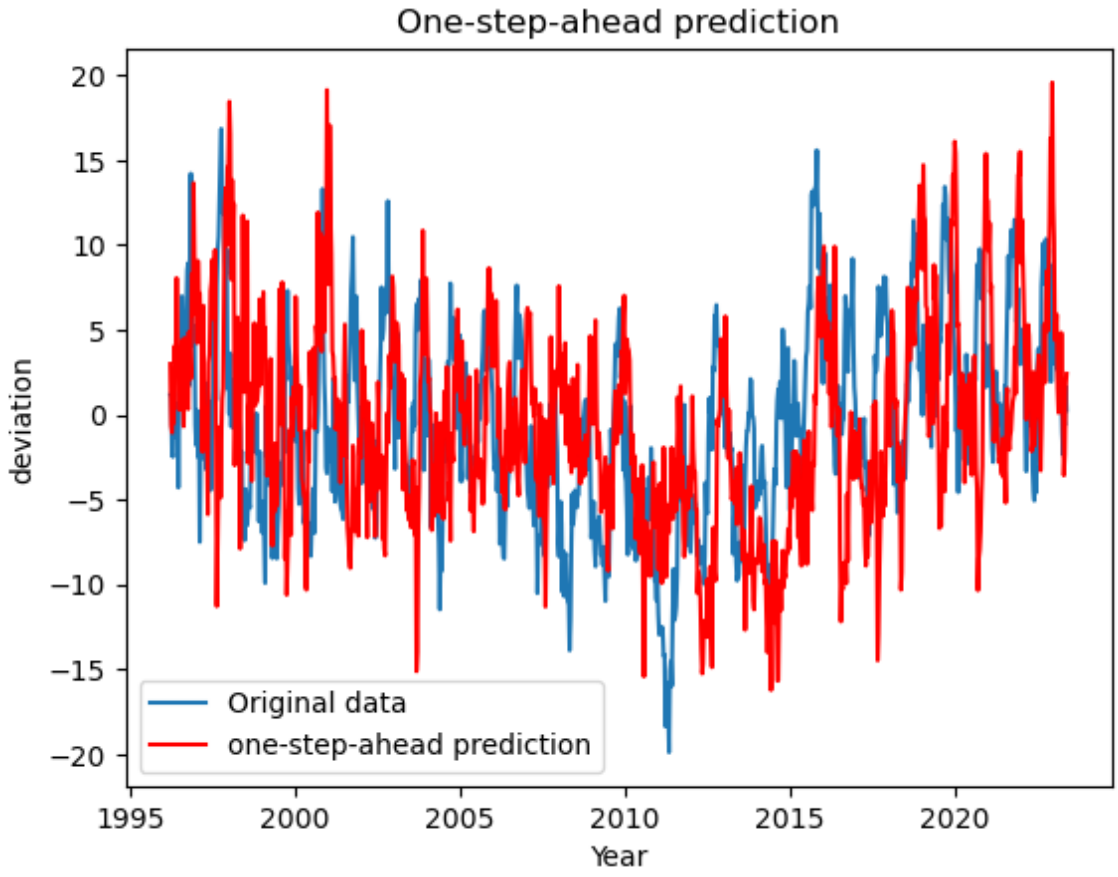
Hint: You will need the methods `fit` and `predict` of `MLPRegressor`. Read the user guide of `scikit-learn` for more details. Recall that a NAR model is conceptuall very similar to an AR model, so you can reuse part of the code from above.

A12:

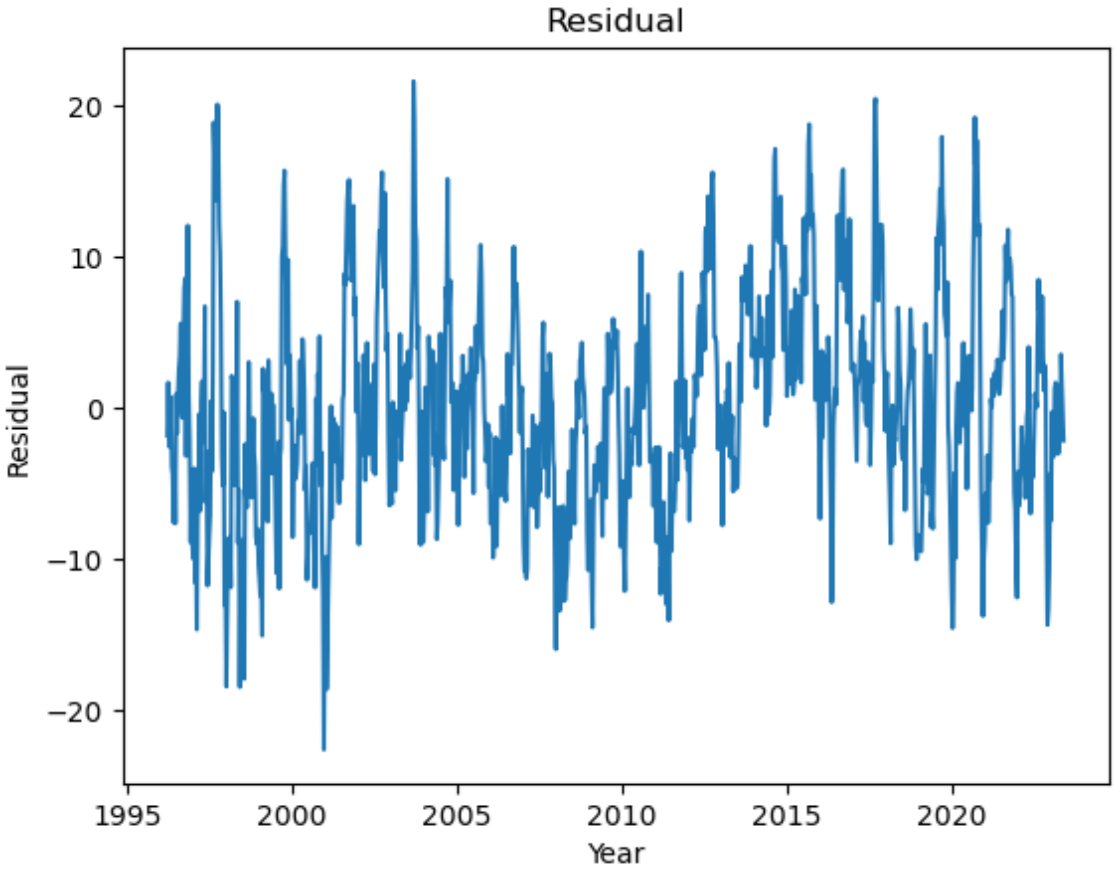
In [89]:
```
def fit_nar(y, p,hidden_layer_sizes=10,activation='relu',max_iter=3000,solv
    """Fits an NAR(p) model. The loss function is the sum of squared errors

    :param y: array (n,), training data points
    :param p: int, NAR model order
    :return regr: learnt NAR moodel
    """

    # Number of training data points
    n = y.shape[0]

    # Construct the regression matrix
    Phi = np.zeros(((n-p),p))
    for j in range(p):
        Phi[:,j] =y[(p-j-1):(n-j-1)]

    # Drop the first p values from the target vector y
    yy = y[p:]  # yy = (y_{t+p+1}, ..., y_n)

    regr = MLPRegressor(hidden_layer_sizes=hidden_layer_sizes,
                        activation=activation, max_iter=max_iter,
                        solver=solver)
    regr.fit(Phi, yy)


    return regr


def predict_nar_1step(regr, y_target,p):
    """Predicts the value y_t for t = p+1, ..., n, for an NAR(p) model, bas
    one-step-ahead prediction.

    :param regr: the trained NN Regressor model.
    :param y_target: array (n,), the data points used to compute the predic
    :return p: int, NAR model order
    """
    n = len(y_target)

    # Number of steps in prediction
    m = n-p
    y_pred = np.zeros(m)

    for i in range(m):
        # <COMPLETE THIS CODE BLOCK>
        y_pred[i] =regr.predict(y_target[i:(i+p)].reshape(1, -1))

    return y_pred
```

In [96]:
```
p=118

regr=fit_nar(train_Y.flatten(),p)

# Make Prediction
res=predict_nar_1step(regr,new_Y.flatten(),p)

# Residual
residual=new_Y.flatten()[p:]-res
```

In [97]:
```
# Plot the one-step-ahead predictions
plt.plot(X[p:],new_Y[p:],label="Original data")
plt.plot(X.flatten()[p:],res,c='r',label="one-step-ahead prediction")
plt.title('One-step-ahead prediction')
plt.xlabel('Year')
plt.ylabel('deviation')
plt.legend()
plt.show()
```



In [98]:
```
# Plot the residual
plt.plot(X.flatten()[p:],residual)
plt.title('Residual')
plt.xlabel('Year')
plt.ylabel('Residual')
plt.show()
```



Q13: Try to expirement with different choices for the hyperparameters of the network (e.g. number of hidden layers and units per layer, activation function, etc.) and the optimizer (e.g. `solver` and `max_iter`).

Are you satisfied with the results? Why/why not? Discuss what the limitations of this approach might be.

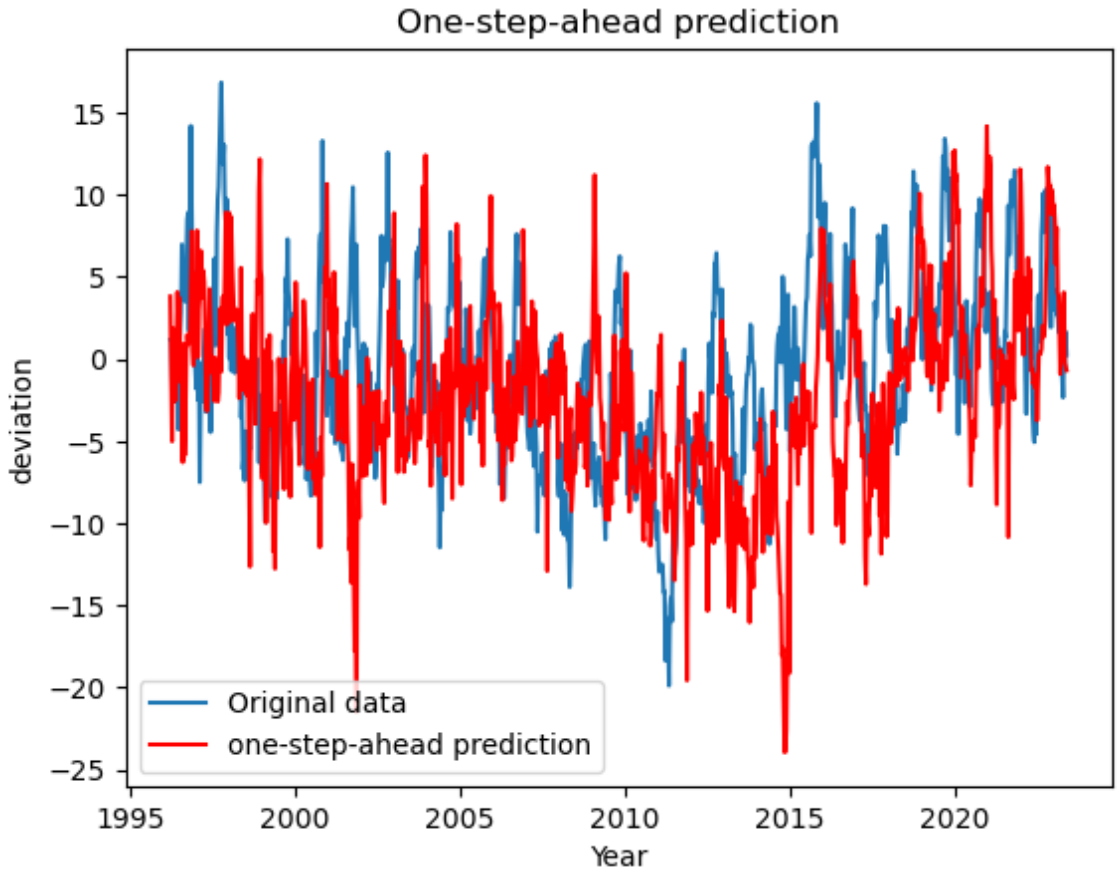
A13: In this choices of parameters, we get an worse MSE than the linear model.Interestingly, if we choose p=2, the fitting will be much better, this might because p=118 is the best hyperparameter of linear AR, but not non-linear AR model.

In this case, there are overfitting problem. But if we choose p=118, the training error won't be better, nor validation error. And if we try p=10,20,50,100,..., the predicted data trend is a bit like the trend of the original data delayed by p time points, this is weird.

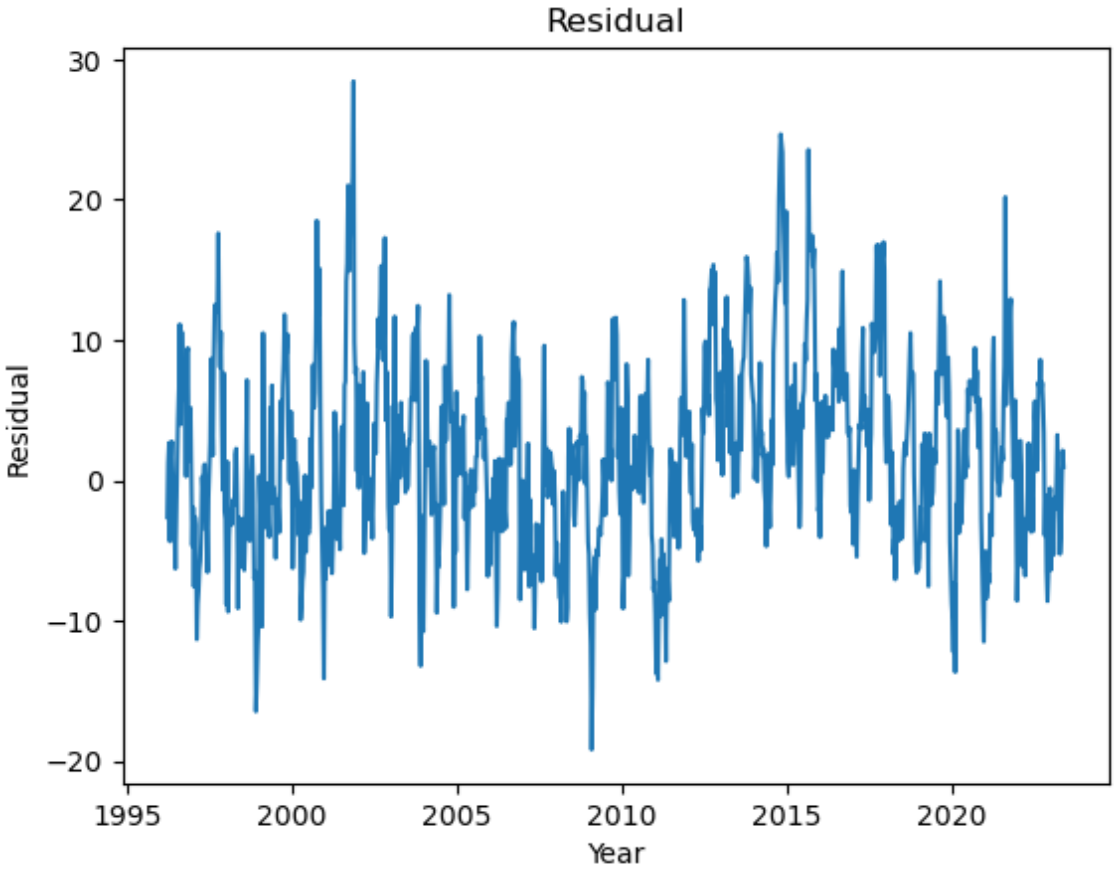
Learning an NN requires nonlinear optimization, the model may trapped into local optimal. More hyper-parameters need to be tune, grid search is expensive.

In [132]:
```
p=118

regr=fit_nar(train_Y.flatten(),p,hidden_layer_sizes=(10,10),activation='rel

# Make Prediction
res=predict_nar_1step(regr,new_Y.flatten(),p)

# Residual
residual=new_Y.flatten()[p:]-res
```

In [133]:
```
# Plot the one-step-ahead predictions
plt.plot(X[p:],new_Y[p:],label="Original data")
plt.plot(X.flatten()[p:],res,c='r',label="one-step-ahead prediction")
plt.title('One-step-ahead prediction')
plt.xlabel('Year')
plt.ylabel('deviation')
plt.legend()
plt.show()
```



In [134]:
```
# Plot the residual
plt.plot(X.flatten()[p:],residual)
plt.title('Residual')
plt.xlabel('Year')
plt.ylabel('Residual')
plt.show()
```



In [135]:
```
print(np.mean(residual**2))
```

```
47.83735186130395
```

In [ ]:
```

```