# Group_lab1_modified

September 27, 2022

## 1 TSSL Lab 1 - Autoregressive models

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

[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 regression
 ↪problems
from sklearn.neural_network import MLPRegressor # Used for NAR model

from tssltools_lab1 import acf, acfplot # Module available in LISAM - Used for
 ↪plotting ACF
```

### 1.1 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/ 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?

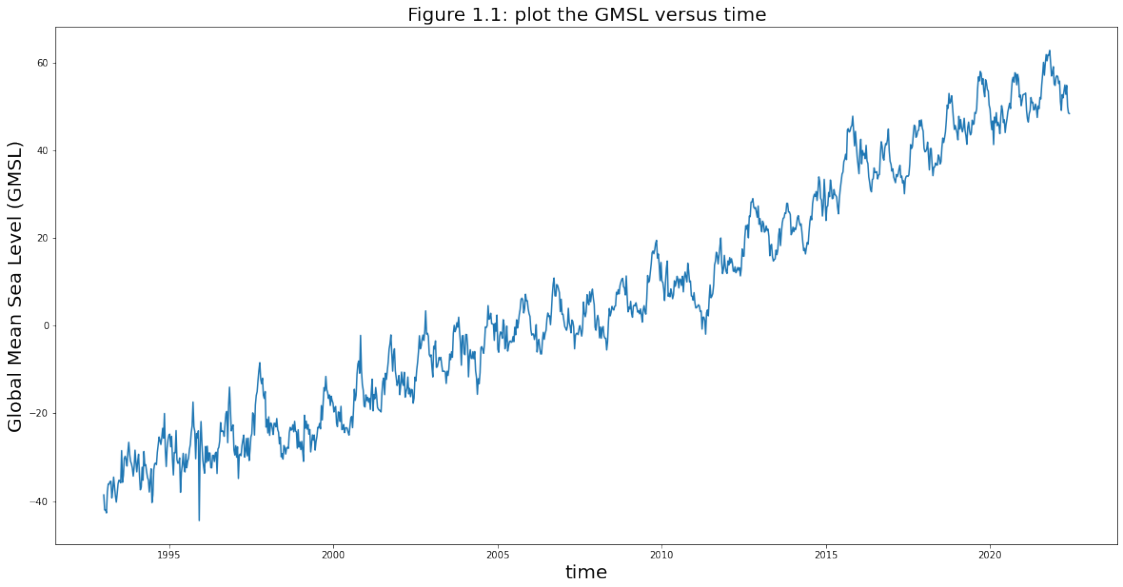
*Hint:* With pandas you can use the function `pandas.read_csv` to read the csv file into a data frame. Plotting the time series can be done using `pyplot`. Note that the sea level data is stored in the 'GMSL' column and the time when each data point was recorded is stored in the column 'Year'.

**A1**: The code below shows there is 1083 observations.

[2]:
```
df = pandas.read_csv('sealevel.csv')
print(df.shape[0])

# plt
fontsize = 20
plt.title('Figure 1.1: plot the GMSL versus time', fontsize = fontsize)
plt.xlabel('time', fontsize = fontsize)
plt.ylabel('Global Mean Sea Level (GMSL)', fontsize = fontsize)
plt.plot(df["Year"], df["GMSL"])
plt.gcf().set_size_inches(20, 10)
```

```
plt.show()
```

```
1083
```




**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:** 1. If the values of the data keep increasing over time, some statistical properties such as sample mean or variance will grow with the size of the sample and they will always underestimate the mean or variance in future periods. This is very detrimental to our ability to analyze and predict future series.

2. For time series modeling methods, such as AR, MA, ARMA, etc., the series are required to be stationary.

**Q3** Detrend the data following these steps: 1. Fit a straight line, $ _t= _0 + _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 ne`

`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 = $ const. Therefore, you can just as well use $t$ directly in the linear regression function if you prefer, $ _t= _0 + _1 t $.

**A3:** code and results are shwon below

[3]:
```
Y =  df["GMSL"]
X = np.asarray(df["Year"]).reshape(-1,1)
reg = lm.LinearRegression().fit(X,Y)
fitted_line = reg.predict(X)
new_Y = Y-fitted_line

# plot lm
fontsize = 20
plt.title('Figure 3.1: Linear regression', fontsize = fontsize)
plt.xlabel('time', fontsize = fontsize)
plt.ylabel('GMSL', fontsize = fontsize)
plt.plot(X, Y)
plt.plot(X, fitted_line, c='r')
plt.gcf().set_size_inches(20, 10)
plt.show()

# plot deviations
fontsize = 20
plt.title('Figure 3.2: Deviations (detrend data)', fontsize = fontsize)
plt.xlabel('time', fontsize = fontsize)
plt.ylabel('deviation', fontsize = fontsize)
plt.plot(X, new_Y)
plt.gcf().set_size_inches(20, 10)
plt.show()
```

Figure 3.1: Linear regression

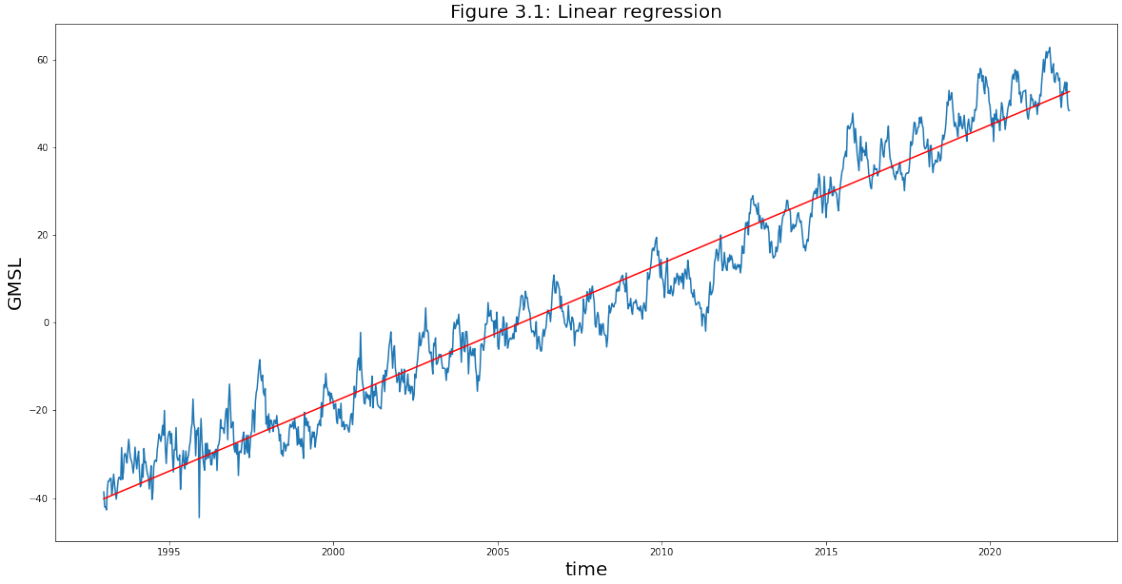
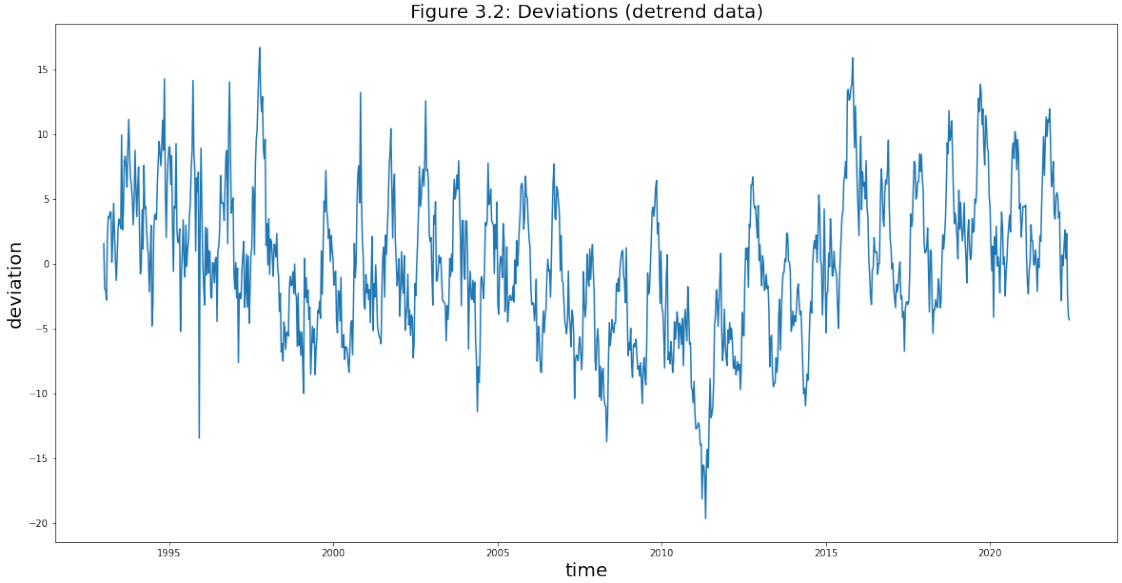


Figure 3.2: Deviations (detrend data)




[4]:
```
len(new_Y)
```

[4]: 1083

**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:**

[5]:
```
train_x = df.iloc[0:700]
valid_x  = df.iloc[701:]
train_y = np.asarray(new_Y.iloc[0:700])
valid_y  = np.asarray(new_Y.iloc[701:])

# plot training data (detrend)
fontsize = 20
plt.title('Figure 4.1: training', fontsize = fontsize)
```

```
plt.xlabel('time', fontsize = fontsize)
plt.ylabel('value', fontsize = fontsize)
plt.plot(df["Year"][0:700], train_y, label = 'train')
plt.legend(fontsize = fontsize)
plt.gcf().set_size_inches(20, 10)
plt.show()


# plot validation data (detrend)
fontsize = 20
plt.title('Figure 4.2: validation', fontsize = fontsize)
plt.xlabel('time', fontsize = fontsize)
plt.ylabel('value', fontsize = fontsize)
plt.plot(df["Year"][701:],valid_y, label = 'valid')
plt.legend(fontsize = fontsize)
plt.gcf().set_size_inches(20, 10)
plt.show()
```

Figure 4.1: training

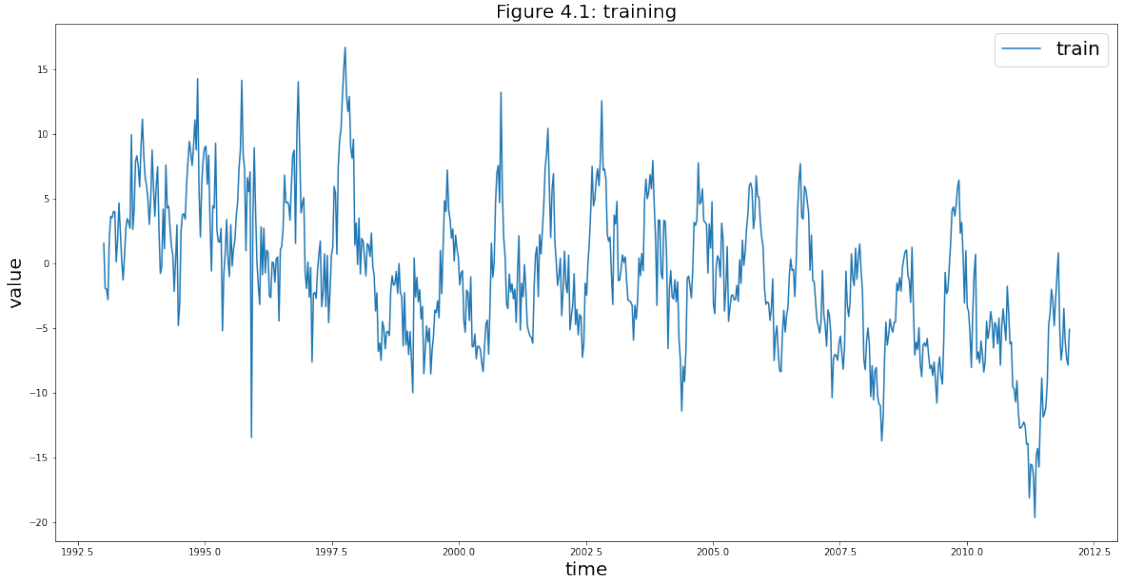
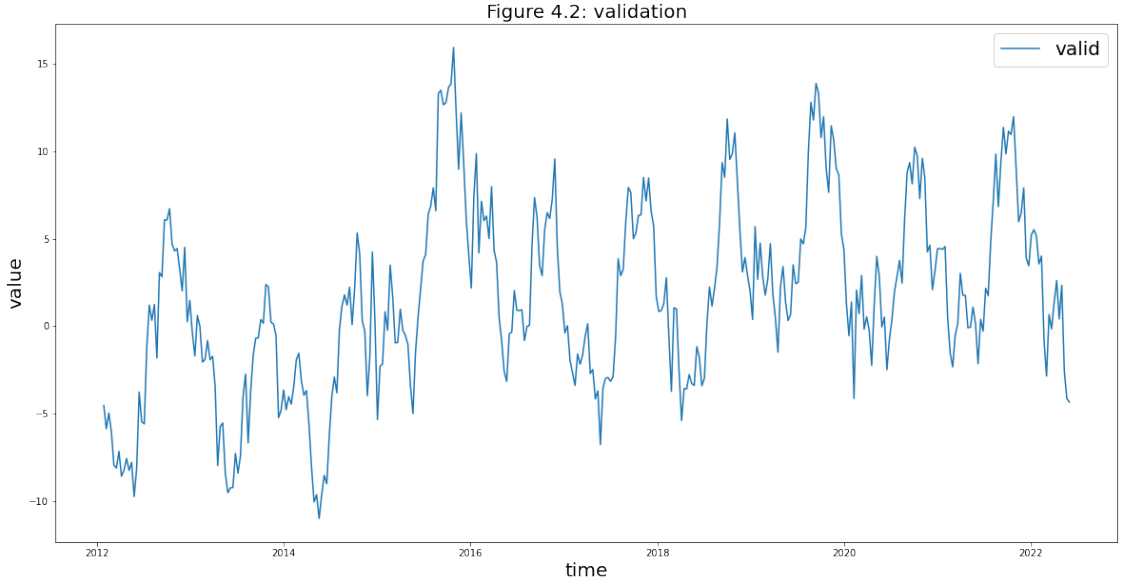


Figure 4.2: validation




## 1.2 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:**

[6]:
```
def fit_ar(y, p):
    """Fits an AR(p) model. The loss function is the sum of squared errors from␣
 ↪t=p+1 to t=n.

    :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 = len(y) # <COMPLETE THIS LINE>

    # Construct the regression matrix
    Phi = np.zeros((n-p,p)) # <COMPLETE THIS LINE>
    for j in range(p):
        Phi[:,j] = y[(p-j-1):(n-j-1)] # <COMPLETE THIS LINE> seminar1+2 page 7
```

```
    # 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␣
 ↪intercept term in the AR model
    regr = lm.LinearRegression(fit_intercept=False)
    regr.fit(Phi,yy)

    return regr.coef_
```

[7]:
```
theta = fit_ar(train_y,10)
print("the coefficients are: ",theta)
```

```
the coefficients are:  [ 0.62917098  0.1226964   0.12524737  0.17675316
-0.02258214 -0.0711463
 -0.05700401  0.04760027 -0.08974475  0.02365747]
```

**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.

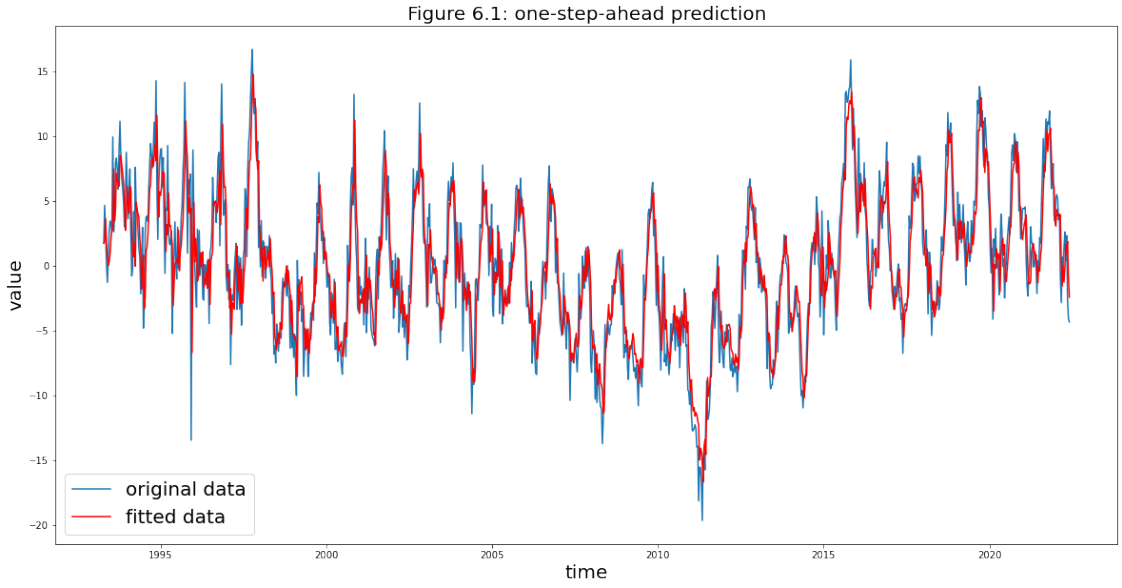
**A6:**

[8]:
```
def predict_ar_1step(theta, y_target):
    """Predicts the value y_t for t = p+1, ..., n, for an AR(p) model, based on␣
 ↪the data in y_target using
    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␣
 ↪predictions.
    :return y_pred: array (n-p,), the one-step predictions (\hat y_{p+1}, ....,␣
 ↪\hat y_n)
    """
#     theta = np.flip(theta)
    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.flip(y_target[i:(i+p)])*theta).sum()
    return y_pred
```

[9]:
```
res = predict_ar_1step(theta, np.asarray(new_Y))
print(len(res))
print(len(new_Y))

# plot AR (the first 10 data points are missing)
fontsize = 20
plt.title('Figure 6.1: one-step-ahead prediction', fontsize = fontsize)
plt.xlabel('time', fontsize = fontsize)
plt.ylabel('value', fontsize = fontsize)
plt.plot(df["Year"][10:],np.asarray(new_Y)[10:], label = 'original data') #␣
 ↪plot new_Y[10:]
plt.plot(df["Year"][10:],res, c='r',label = 'fitted data')
plt.gcf().set_size_inches(20, 10)
plt.legend(fontsize = fontsize)
plt.show()
```

```
1073
1083
```

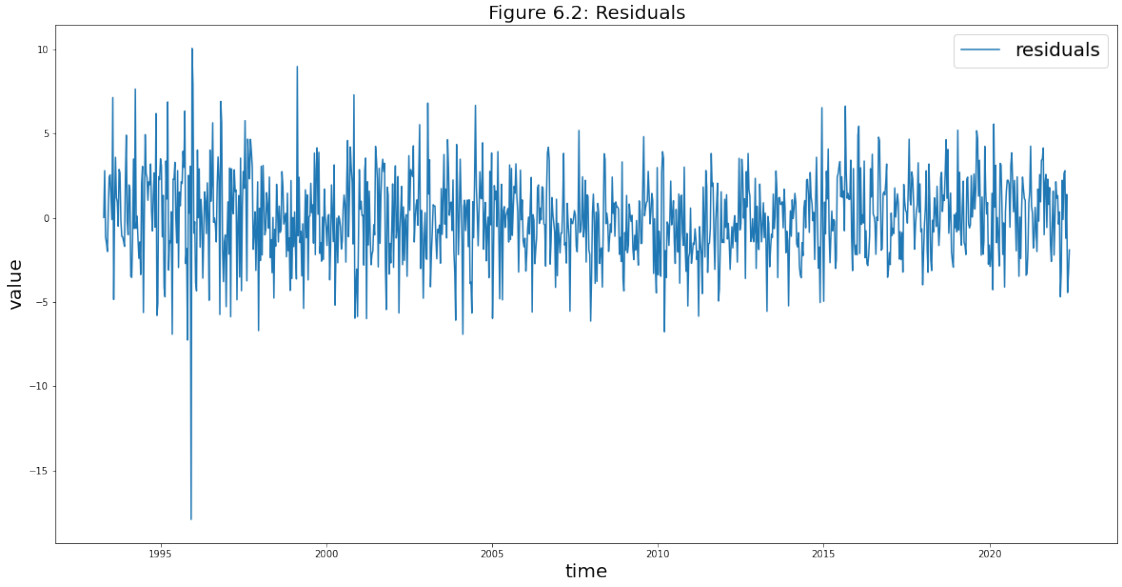



[10]:
```
print(len(new_Y))
print(len(res))
```

```
1083
```

1073

[11]:
```
# res_valid = predict_ar_1step(theta,np.asarray(new_Y))
# residuals = new_y[10:] - res_valid
residuals = np.asarray(new_Y[10:]) - res

# plot residuals (all data points: 1073)
fontsize = 20
plt.title('Figure 6.2: Residuals', fontsize = fontsize)
plt.xlabel('time', fontsize = fontsize)
plt.ylabel('value', fontsize = fontsize)
plt.plot(df["Year"][10:], residuals, label = 'residuals')
plt.legend(fontsize = fontsize)
plt.gcf().set_size_inches(20, 10)
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?

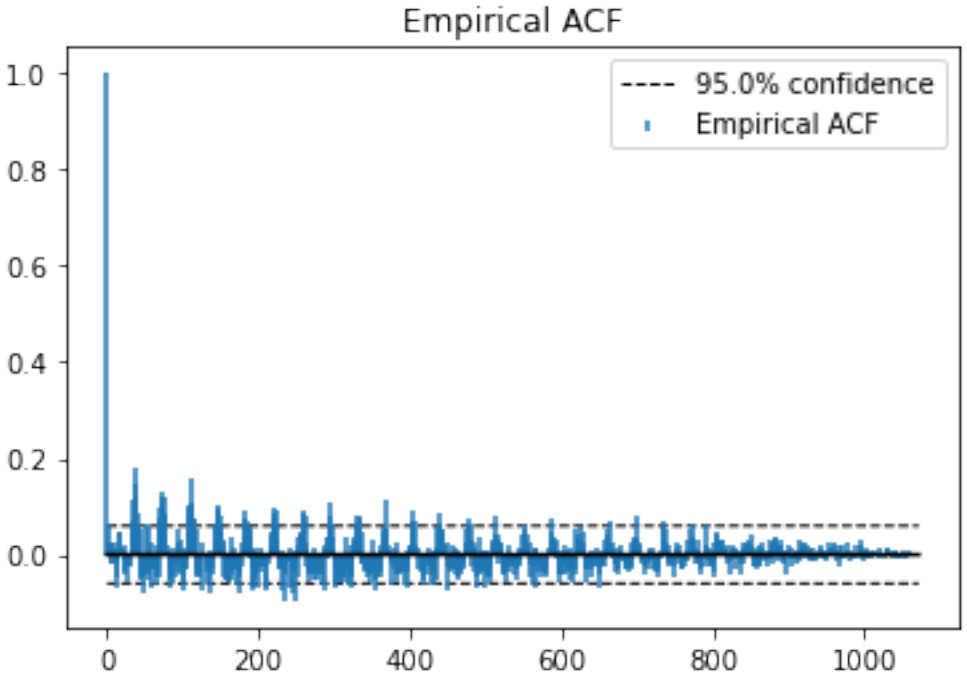
*Hint:* You can use the function `acfplot` from the `tssltools` module, available on the course web page.

**A7:**

[12]:
```
help(acfplot)
acfplot(residuals)
```

```
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
confidence level (based on a central limit
                 theorem under a white noise assumption) for two dashed lines
drawn in the plot. Default is 0.95.
    :return:
```




It seems the model needs some time to meet convergence.

### 1.3 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?

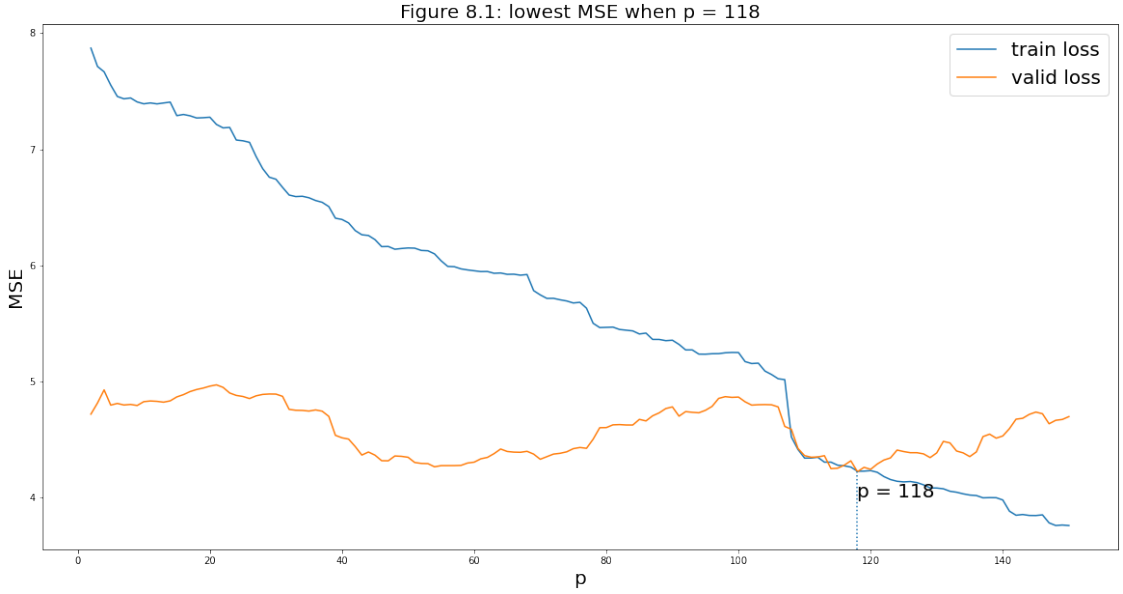
*Note:* There is no obvious "correct answer" to the second question, but you still need to pick an order an motivate your choice!

**A8:**

[13]:
```
start = 2
end = 150
train_loss = np.zeros(end-start+1)
valid_loss = np.zeros(end-start+1)

for i in range(end-start+1):
    p = i + start
    theta = fit_ar(train_y, p)

    res_train = predict_ar_1step(theta, train_y)
    residuals = train_y[p:] - res_train
    train_loss[i] = np.mean(residuals**2)

    res_valid = predict_ar_1step(theta, valid_y)
    residuals = valid_y[p:] - res_valid
    valid_loss[i] = np.mean((residuals)**2)
```

[14]:
```
x_axis = np.arange(start,end+1)
best_p = np.argmin(valid_loss)+start

fontsize = 20
plt.plot(x_axis,train_loss, label = "train loss")
plt.plot(x_axis,valid_loss, label = "valid loss")
plt.ylabel("MSE", fontsize = fontsize)
plt.xlabel("p", fontsize = fontsize)
plt.axvline(x = best_p, ymin=0, ymax=0.15, linestyle=':')
plt.text(118, 4, 'p = 118', fontsize = fontsize)
plt.title("Figure 8.1: lowest MSE when p = {}".format(best_p), fontsize =␣
 ↪fontsize)
plt.gcf().set_size_inches(20, 10)
plt.legend(fontsize = fontsize)
plt.show()
```

Figure 8.1: lowest MSE when p = 118




**a8**: As the order increase, the training loss generally keep reducing while valid loss fluctuate. The lowest MSE loss can be seen when p = 2. So we determine that p = 2 is the best result.

**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.

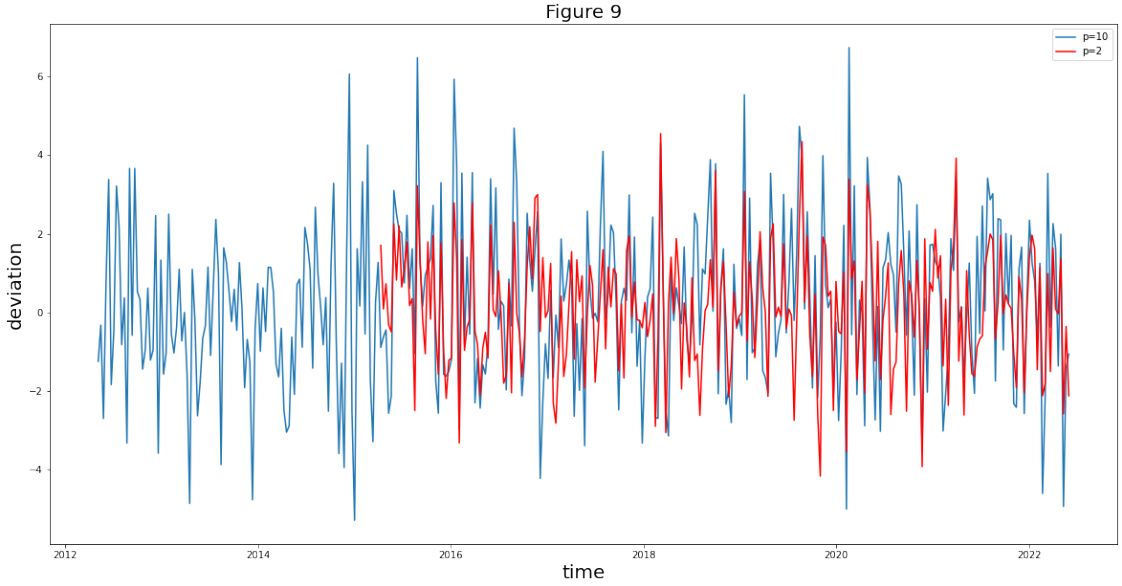
[15]:
```
theta = fit_ar(valid_y,best_p)
res_best_p = predict_ar_1step(theta,valid_y)
theta = fit_ar(valid_y,10)
res_10 = predict_ar_1step(theta,valid_y)
```

[16]:
```
print(len(res_best_p))
print(len(res_10))
```

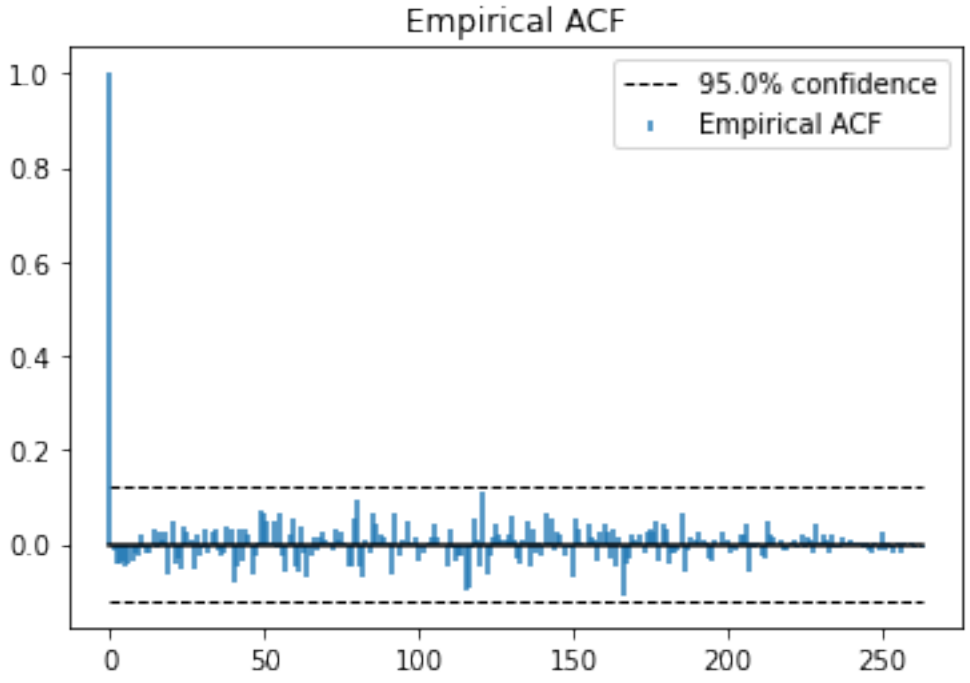
```
264
372
```

[17]:
```
# plot residuals (all data points: 1073)
fontsize = 20
plt.title('Figure 9', fontsize = fontsize)
plt.xlabel('time', fontsize = fontsize)
plt.ylabel('deviation', fontsize = fontsize)
plt.plot(valid_x["Year"][10:],valid_y[10:] - res_10, label = 'p=10')
plt.plot(valid_x["Year"][best_p:],valid_y[best_p:] - res_best_p, c='r',label =␣
↪'p=2')
plt.gcf().set_size_inches(20, 10)
plt.legend()
plt.show()
```

Figure 9




[18]:
```
acfplot(valid_y[best_p:] - res_best_p)
```




**A9** :From the residuals plot, we can see that p=2 can offer much less residuals. Comparing the ACF plot, it's also clear that model with p=2 converge much more quickly.

## 1.4 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

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

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.

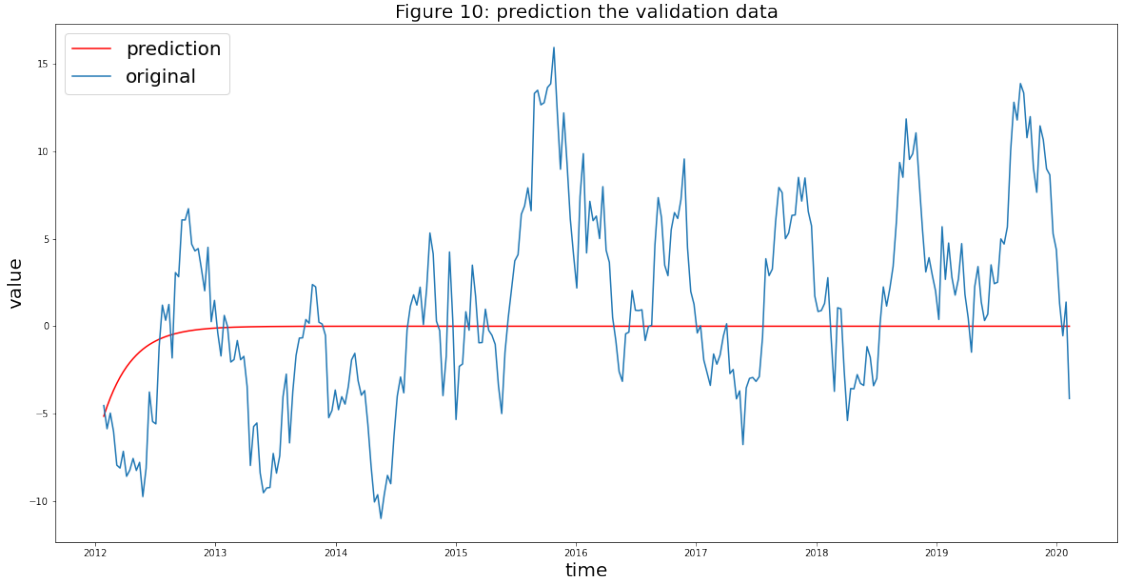
**A10:**

In simulate_ar function for long-range prediction, I think you need to check the line where you append the last prediction to phi . If it is assumed that the last (newest) output is the first element of phi, then you cannot append it at the end.

[19]:
```
def simulate_ar(y, theta, m):
    """Simulates an AR(p) model for m steps, with initial condition given by␣
 ↪the last p values of y

    :param y: array (n,) with n>=p. The last p values are used to initialize␣
 ↪the simulation.
    :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[ntrain-1] is the last training data point
    for i in range(m):
        pre = phi*theta
        y_sim[i] = pre.sum() # <COMPLETE THIS LINE>
        phi = np.flip(np.append(np.flip(np.array(phi)),pre.sum())[-p:])
        phi = np.append(phi,y_sim[i])[-1:]
#         phi = np.insert(phi, 0, y_sim[i])[0:p]
        # <COMPLETE THIS CODE BLOCK>

    return y_sim
```

[20]:
```
theta = fit_ar(train_y,2)
res = simulate_ar(train_y, theta, 297)

# plot
fontsize = 20
plt.title('Figure 10: prediction the validation data', fontsize = fontsize)
plt.xlabel('time', fontsize = fontsize)
plt.ylabel('value', fontsize = fontsize)
plt.plot(df["Year"][701:998],res, label = 'prediction',c='r')
plt.plot(df["Year"][701:998],valid_y[:297],label = 'original')
plt.gcf().set_size_inches(20, 10)

plt.legend(fontsize = fontsize)
plt.show()
```

Figure 10: prediction the validation data



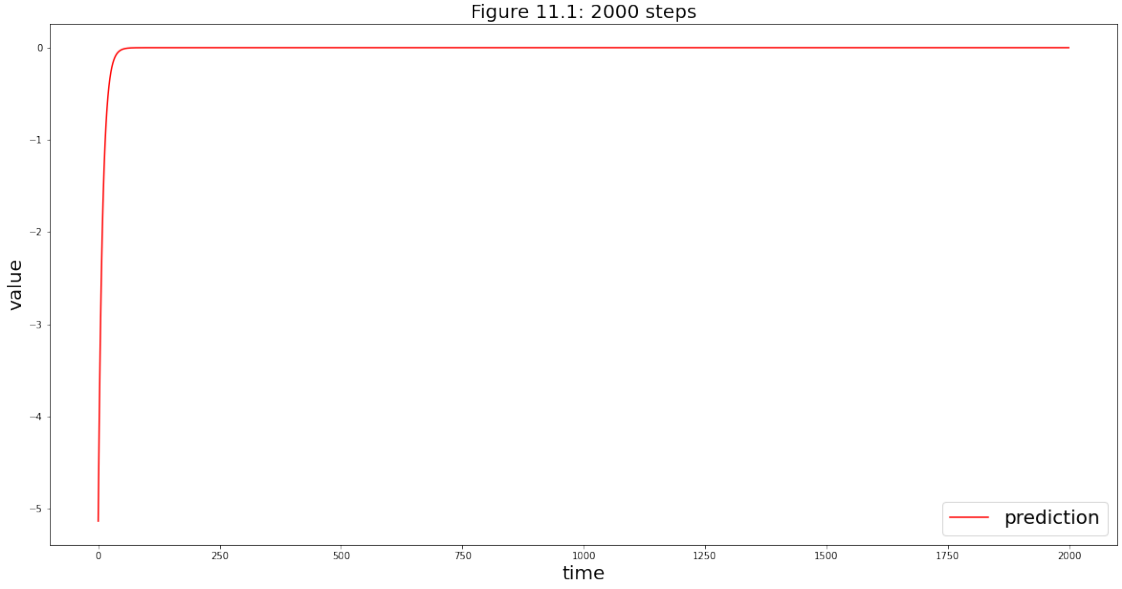

**Q11:** Using the same function as above, try to simulate the process for a large number of time steps (say, $m = 2000$). You should see that the predicted values eventually converge to a constant prediction of zero. Is this something that you would expect to see in general? Explain the result.

**A11:**

[21]:
```
m = 2000
theta = fit_ar(train_y,2)
res = simulate_ar(train_y, theta, m)

fontsize = 20
plt.title('Figure 11.1: 2000 steps', fontsize = fontsize)
plt.xlabel('time', fontsize = fontsize)
```

```
plt.ylabel('value', fontsize = fontsize)
plt.plot(res, label = 'prediction',c='r')
plt.gcf().set_size_inches(20, 10)

plt.legend(fontsize = fontsize)
plt.show()
```

Figure 11.1: 2000 steps




**A11**: For the stationary AR(p) model, when the number of m steps tends to infinity, the predicted result tends to the mean of the series, which is called mean reversion. Multi-step prediction is basically based on the mean value, where the mean value of the sequence is 0, and the mean value of the random item is also 0, so the final prediction result is stable around 0

## 1.5 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 $\hat{y}_t = f_\theta(y_{t-1}, ..., y_{t-p})$, 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:**

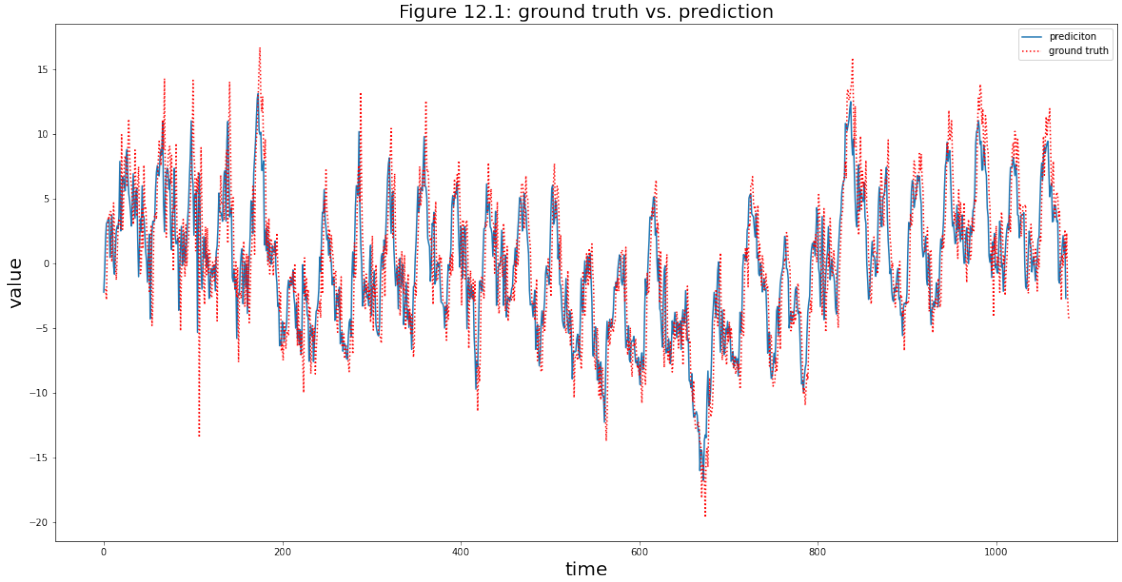
[22]:
```
def fit_nar(y, p,model=None):
    """Fits an AR(p) model. The loss function is the sum of squared errors from␣
 ↪t=p+1 to t=n.

    :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 = len(y) # <COMPLETE THIS LINE>

    # Construct the regression matrix
    Phi = np.zeros((n-p,p)) # <COMPLETE THIS LINE>
    for j in range(p):
        Phi[:,j] = y[(p-j-1):(n-j-1)] # <COMPLETE THIS LINE> seminar1+2 page 7

    # 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␣
 ↪intercept term in the AR model
    if model == None:
        regr = MLPRegressor(hidden_layer_sizes=10,max_iter=5000)
    else:
        regr = model
    regr.fit(Phi,yy)

    return regr

def predict_nar_1step(m, y_target, p):
    """Predicts the value y_t for t = p+1, ..., n, for an AR(p) model, based on␣
 ↪the data in y_target using
    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␣
 ↪predictions.
    :return y_pred: array (n-p,), the one-step predictions (\hat y_{p+1}, ....,␣
 ↪\hat y_n)
    """

    n = len(y_target)
```

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

    for i in range(m):
        data = y_target[i:(i+p)].reshape(1,-1)
        y_pred[i] = model.predict(data)

    return y_pred
```

[23]:
```
model = fit_nar(train_y, 2)
res = predict_nar_1step(model, np.asarray(new_Y).reshape(-1,1), 2)
residuals = res-np.asarray(new_Y[2:])
```

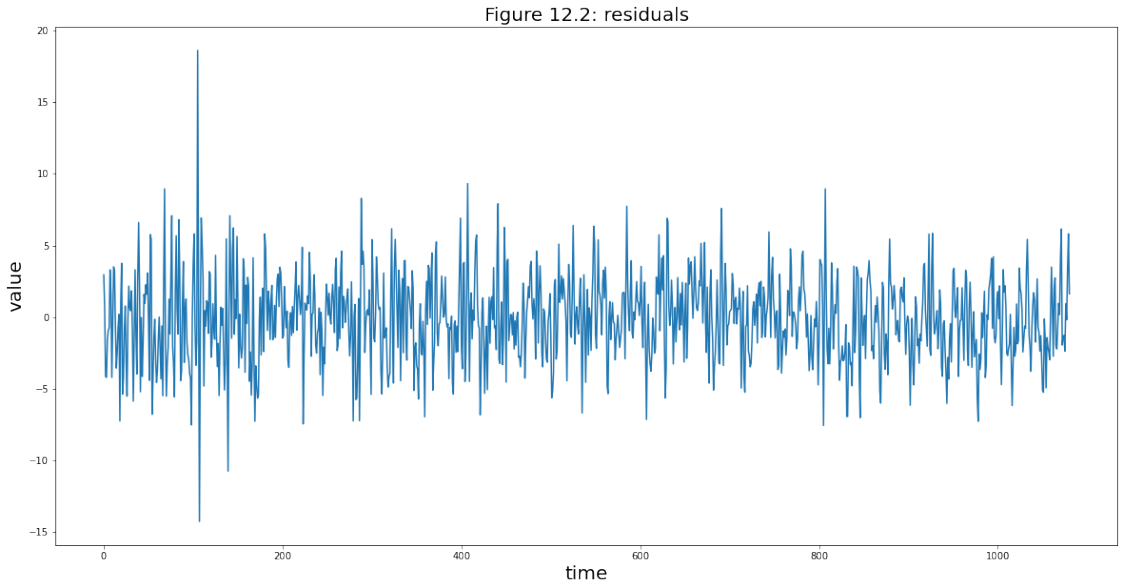
[24]:
```
# plot prediction (res)
fontsize = 20
plt.plot(res[2:],label = "prediciton")
plt.title("Figure 12.1: ground truth vs. prediction", fontsize = fontsize)
plt.plot(new_Y[2:],label = "ground truth",c='r', linestyle = ":")
plt.gcf().set_size_inches(20, 10)
plt.legend()
plt.xlabel('time', fontsize = fontsize)
plt.ylabel('value', fontsize = fontsize)
plt.show()
```

Figure 12.1: ground truth vs. prediction




[25]:
```
# plot residuals (residuals)
plt.plot(residuals)
```

```
plt.title("Figure 12.2: residuals", fontsize = fontsize)
plt.gcf().set_size_inches(20, 10)
plt.xlabel('time', fontsize = fontsize)
plt.ylabel('value', fontsize = fontsize)
plt.show()
```




[26]:
```
len(new_Y[2:])
len(residuals)
```

[26]: 1081

[27]:
```
def mse(pre,true):
    return np.mean(pre-true)**2
```

**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:**

[28]:
```
model = MLPRegressor(hidden_layer_sizes=(20,), activation='relu',␣
↪solver='adam', alpha=0.0001, batch_size='auto', learning_rate='invscalling',␣
↪learning_rate_init=0.002,
                     power_t=0.5, max_iter=500, shuffle=True,␣
↪random_state=None, tol=0.0001, verbose=False, warm_start=False, momentum=0.
↪85,
```

```
                    nesterovs_momentum=False, early_stopping=True,␣
  ↪validation_fraction=0.1, beta_1=0.9, beta_2=0.999, epsilon=1e-08,␣
  ↪n_iter_no_change=10, max_fun=15000)
mse1 = []
for i in range(30):
    model = fit_nar(train_y, 2)
    res = predict_nar_1step(model, valid_y.reshape(-1,1), 2)
    mse1.append(mse(res,valid_y[2:]))
print(np.mean(np.asarray(mse1)))
```

```
0.13043903574336507
```

For this method the results are quite good. In this method, we have only previous value without all other information, it's like we are encoding those features(like those in the original dataset) directly into a single value. This sinple encoding will lose information and thus adversely affecting the result.