TSSL Lab 1 - Autoregressive models

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

In [1]:

```
import pandas as pd # 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
```

Lab submitted by Akshay Gurudath (aksgu350) Keshav Padiyar (kespa139)

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'.

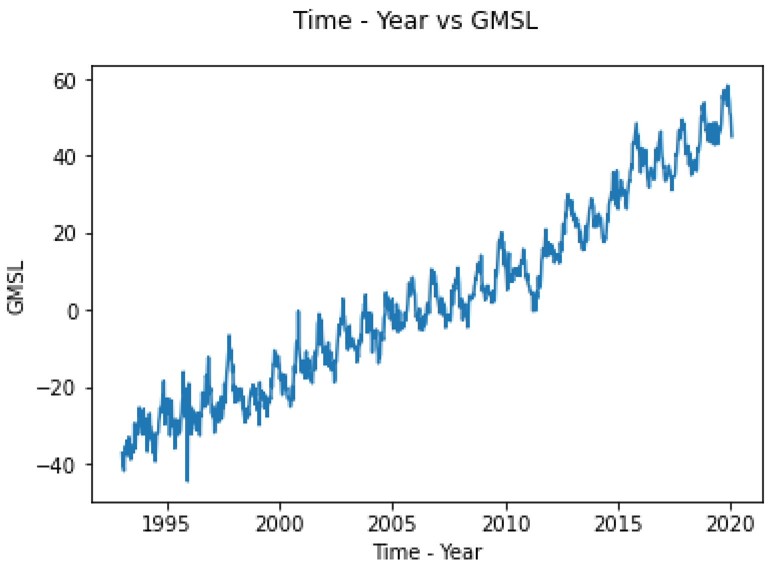
A1:

In [2]:

```
sealevel=pd.read_csv("sealevel.csv")

plt.plot("Year","GMSL",data=sealevel)
plt.xlabel('Time - Year')
plt.ylabel('GMSL')
plt.suptitle('Time - Year vs GMSL')
```

Out[2]:

```
Text(0.5, 0.98, 'Time - Year vs GMSL')
```



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 trend line is subtracted from the data to make the process have a constant mean (of zero in this case) and in turn convert it into a stationary process. Estimating the coefficients become simpler when we turn it into a stationary process

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.

1. Subtract the fitted line from y_t for the whole data series and plot the deviations from the straight line.

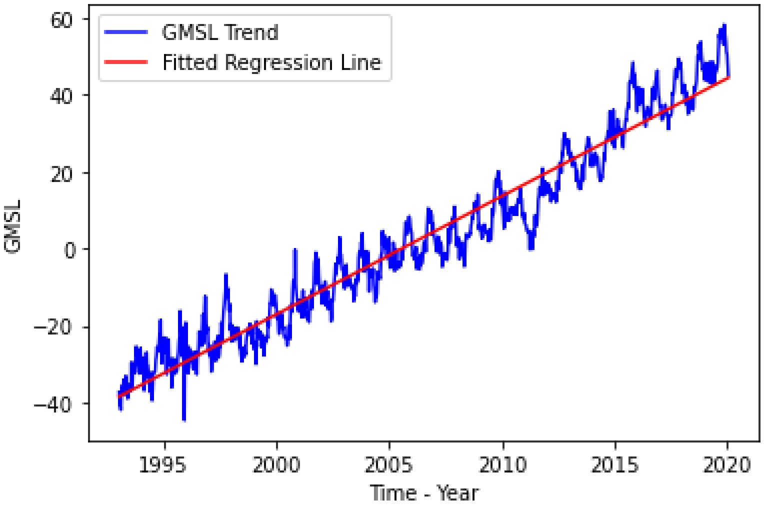
From now, we will use the detrended data in all parts of the lab.

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, $\mu_t = \theta_0 + \theta_1 t$.

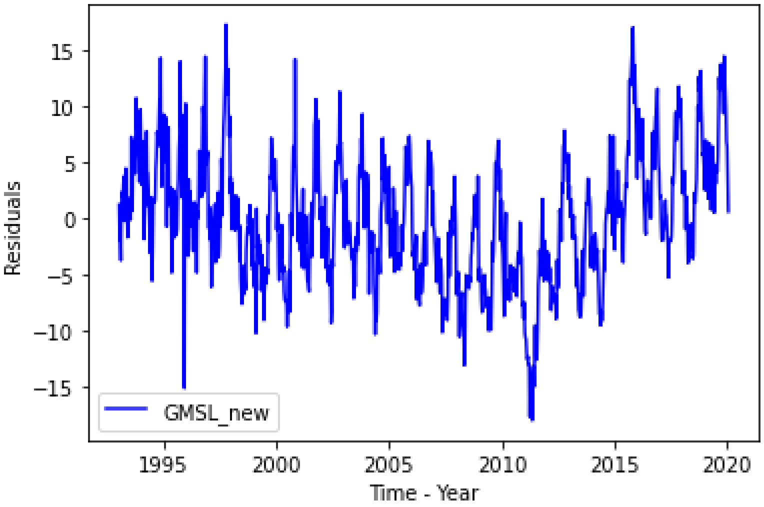
A3:

In [3]:

```
x1=sealevel["Year"].values.reshape(-1,1)
y1=sealevel["GMSL"].values

model=lm.LinearRegression().fit(X=x1,y=y1)

plt.plot("Year","GMSL",data=sealevel,color="b",label="GMSL Trend")
plt.plot(sealevel["Year"],model.predict(x1),color='r',label="Fitted Regression Line")
plt.xlabel("Time - Year")
plt.ylabel("GMSL")
plt.legend()
plt.show()
```



In [4]:
```
GMSL_new=y1-model.predict(x1)

sealevel["GMSL_new"]=GMSL_new

plt.plot("Year","GMSL_new",data=sealevel,color="b")
plt.xlabel("Time - Year")
plt.ylabel("Residuals")
plt.legend()
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

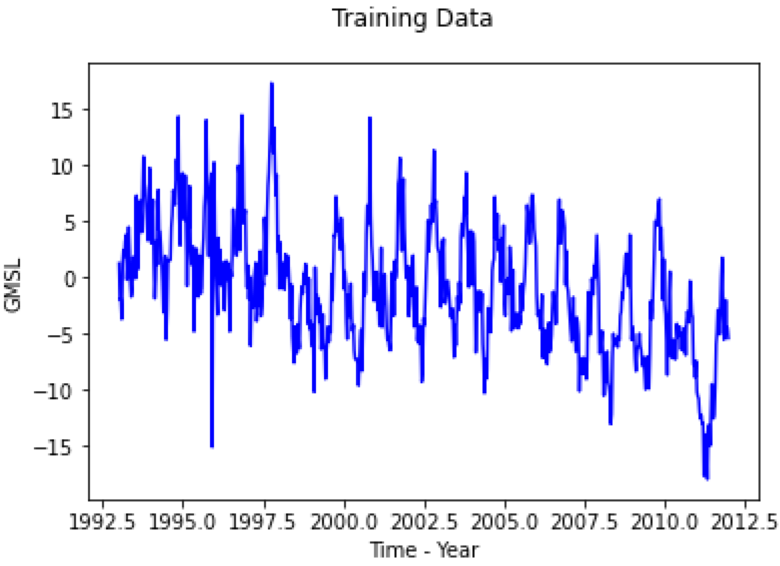
data"), in which case we should not use these observations when fitting the model, including the detrending step.

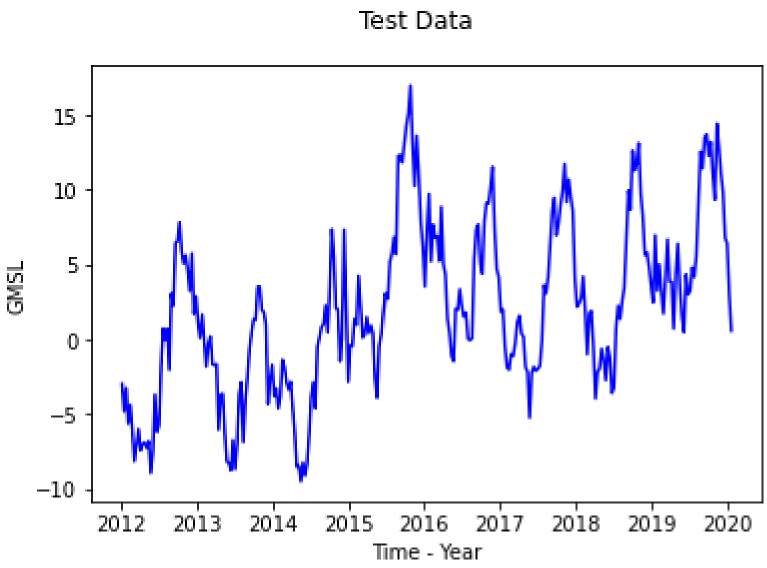
In this laboration we consider the linear detrending as a predetermined preprocessing step and therefore allow ourselves to use the validation data when computing the linear trend.

A4:

In [5]:

```
train=sealevel[0:700]
test=sealevel[700:]

plt.plot("Year","GMSL_new",data=train,color="b")
plt.xlabel("Time - Year")
plt.ylabel("GMSL")
plt.suptitle("Training Data")
plt.show()


plt.plot("Year","GMSL_new",data=test,color="b")
plt.xlabel("Time - Year")
plt.ylabel("GMSL")
plt.suptitle("Test Data")
plt.show()


n_training=train.shape[0]
n_test = test.shape[0]
n=sealevel.shape[0]
```





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 [6]:

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

    :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>

    # 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
    regr = lm.LinearRegression(fit_intercept=False)
    regr.fit(Phi,yy)

    return regr.coef_
```

In [7]:

```
coef=fit_ar(train["GMSL_new"],p=10)
```

```
coef
```

Out[7]:

```
array([ 0.62156052,  0.10763277,  0.15104657,  0.1745703 , -0.02184709,
       -0.05955406, -0.09578106,  0.07585221, -0.11175939,  0.02305208])
```

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.

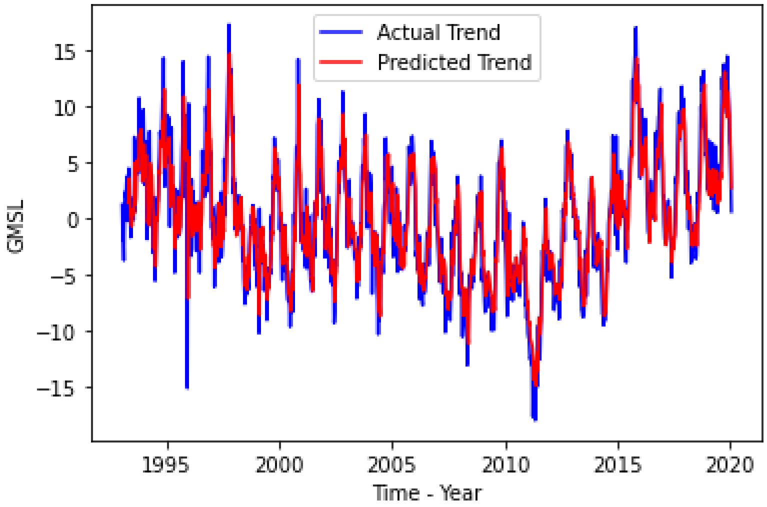
A6:

In [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 dat
    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 = 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(theta,np.flip(y_target[i:p+i])) # <COMPLETE THIS LINE>

    return y_pred
```

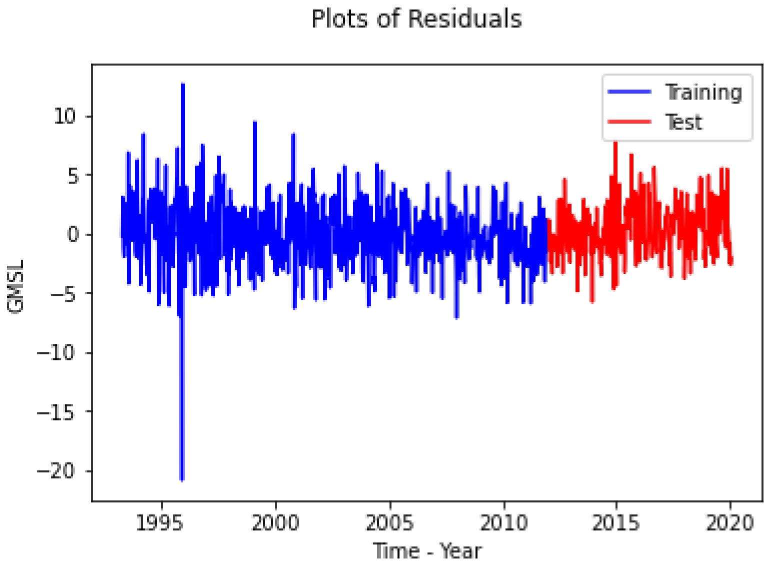
In [9]:

```
y_pred=predict_ar_1step(coef,sealevel["GMSL_new"])

plt.plot("Year","GMSL_new",data=sealevel,color="b",label="Actual Trend")
plt.xlabel("Time - Year")
plt.ylabel("GMSL")

plt.plot(sealevel["Year"][10:],y_pred,color="r",label="Predicted Trend")
plt.xlabel("Time - Year")
plt.ylabel("GMSL")
plt.legend()


plt.show()
```



It is very important to note here that a part of the training data is used to predict for validation. Because of this, we do not really lose p observations from the validation data. If we want to keep the training data seperate from the validation data, then we can also do that by ignoring the first p terms. In this report, we have *not* ignored the first p terms during validation

In [10]:
```
res=sealevel["GMSL_new"][10:]-y_pred

plt.plot(sealevel["Year"][10:700],res[:690],color="b",label="Training")
plt.xlabel("Time - Year")
plt.ylabel("GMSL")

plt.plot(sealevel["Year"][700:],res[690:],color="r",label="Test")
plt.xlabel("Time - Year")
plt.ylabel("GMSL")

plt.suptitle("Plots of Residuals")

plt.legend()

plt.show()
```



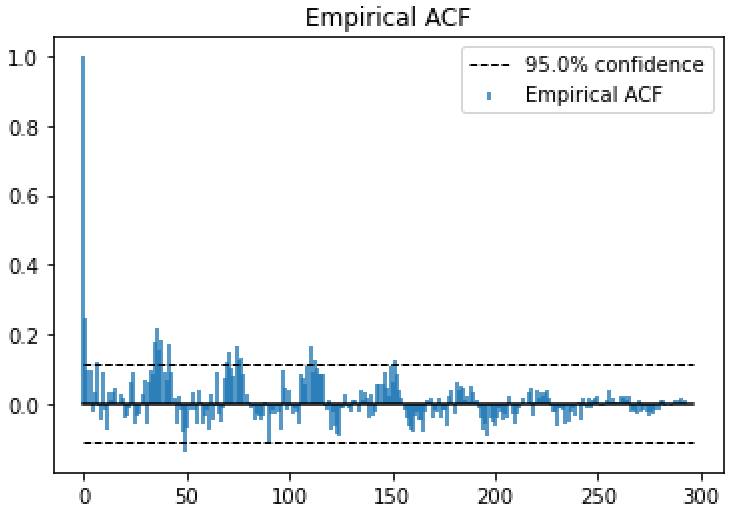
Q7: Compute and plot the autocorrelation function (ACF) of the *residuals* only for the *validation data*. What conclusions can you draw from the ACF plot?

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

A7:

```
In [11]: acfplot(res[690:])
         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. D
efault is None.
    :param conf: float, number in the interval [0,1] which specifies the confidence leve
l (based on a central limit
                 theorem under a white noise assumption) for two dashed lines drawn in t
he plot. Default is 0.95.
    :return:
```



We can see here that as the lag increases, ACF decreases and mostly the ACF values are within the confidence interval. However we see that arpund the value of 45 the ACF is pretty high. This might mean that two data points at a lag of 45 are correlated and therefore we may be looking at a model with higher order.

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:

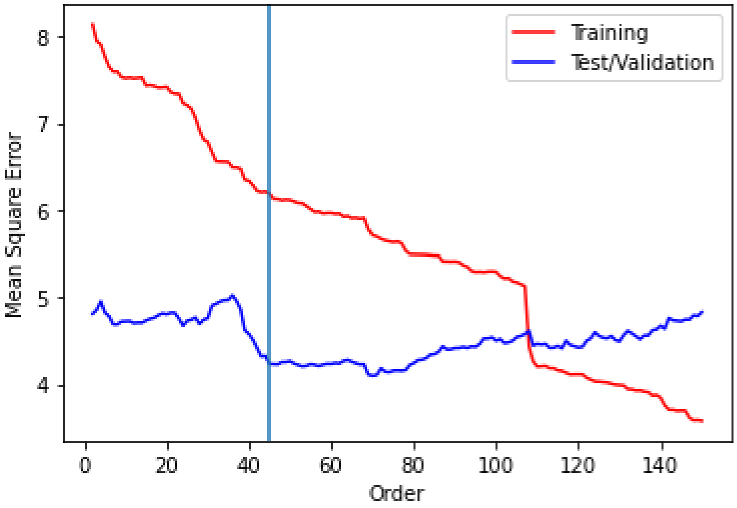
- What is the main difference between the changes in training error and validation error as the order increases?
- Based on these results, which model order would you suggest to use and why?

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

A8:

In [12]:

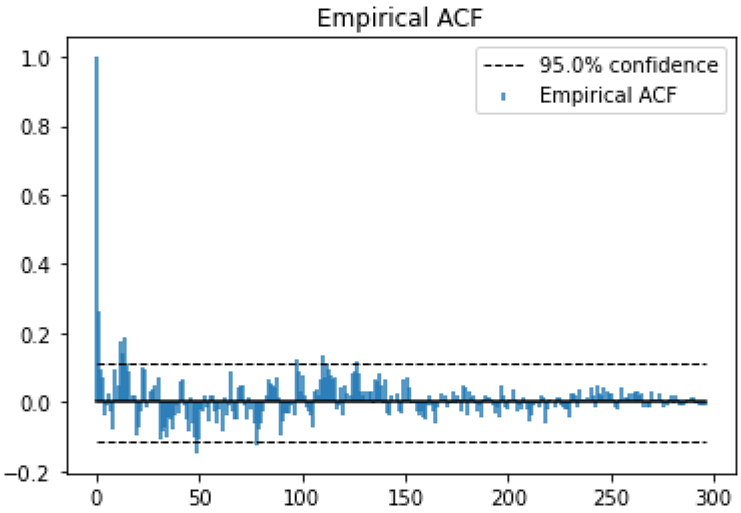
```python
train_error=[]
test_error=[]

for i in range(2,151):
    split=700-i
    y_pred1=predict_ar_1step(fit_ar(train["GMSL_new"],p=i),sealevel["GMSL_new"])
    train_error.append(sum((y_pred1[:split]-train["GMSL_new"][i:])**2)/(n_training-i))
    test_error.append(sum((y_pred1[split:]-test["GMSL_new"])**2)/n_test)

plt.plot(range(2,151),train_error,color="r",label="Training")
plt.xlabel("Order")
plt.ylabel("Mean Square Error")

plt.plot(range(2,151),test_error,color="b",label="Test/Validation")
plt.legend()

plt.axvline(x=45)
plt.show()
```



As the order increases, the training error falls sharply, however the validation error drops and after a point stays constant.The training error may not be an indication of the model as it is natural for the MSE to decrease with the order increases. We would suggest an order of 45 as the test/validation error is very low at that point. Suggesting a higher model order may lead low test/validation error but it will also compromise more data points at the start and the marginal MSE decrease is not worth it.

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.

In [13]:
```
order=45

y_pred1=predict_ar_1step(fit_ar(train["GMSL_new"],p=order),sealevel["GMSL_new"])

test_pred=y_pred1[655:]

residuals=test["GMSL_new"]-test_pred
acfplot(residuals)
```



We see now that most of the ACF is within the confidence interval and there is some variation because of noise in the data.

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.

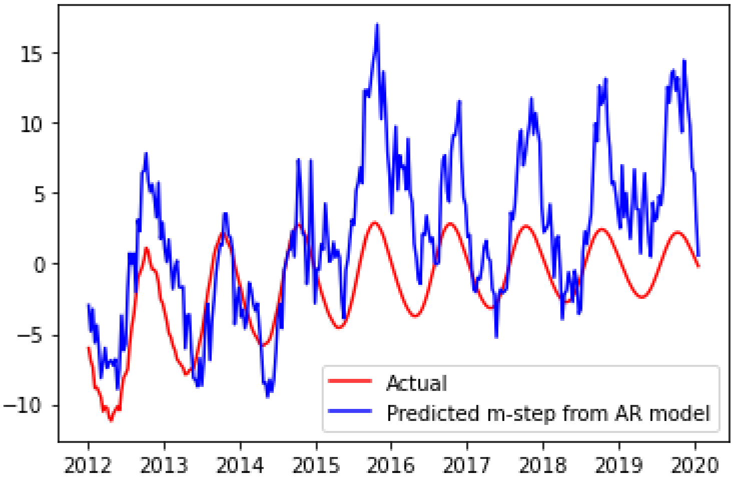
Hint: Use the pseudo-code derived at the first pen-and-paper session.

A10:

In [14]:
```
def simulate_ar(y, theta, m):
    """Simulates an AR(p) model for m steps, with initial condition given by the last p

    :param y: array (n,) with n>=p. The last p values are used to initialize the simula
    :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

    for i in range(m):
        if(p-i>0):
            y_sim[i]=np.dot(np.append(np.flip(y_sim[0:i]),phi[:p-i]),theta)
        else:
            y_sim[i]=np.dot(np.flip(y_sim[i-p:i]),theta)

    return y_sim
```

In [15]:

```
p=45

y_pred2=simulate_ar(train["GMSL_new"],fit_ar(train["GMSL_new"],p=order),m=n_test)

plt.plot(test["Year"],y_pred2,color="r",label="Actual")
plt.plot(test["Year"],test["GMSL_new"],color='b',label="Predicted m-step from AR model"
plt.legend()
plt.show()
```

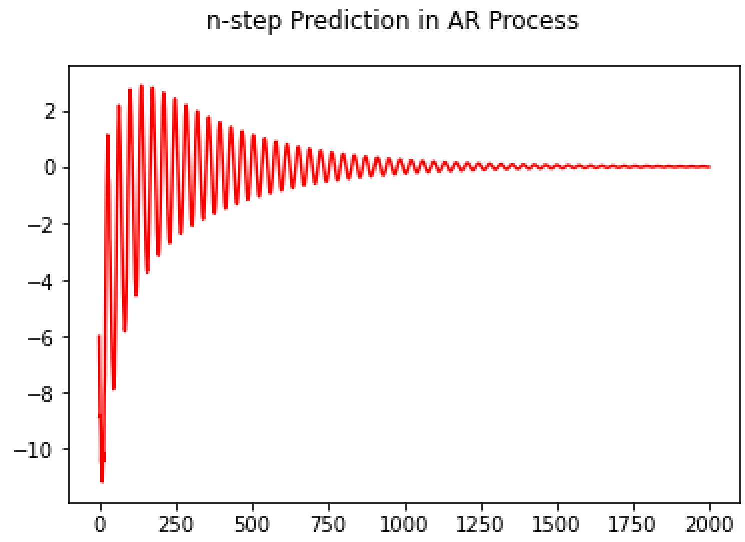


Q11: Using the same function as above, try to simulate the process for a large number of time steps (say, $m = 2000$). You should see that the predicted values eventually converge to a constant prediction of zero. Is this something that you would expect to see in general? Explain the result.

A11:

In [16]:

```
p=45

y_pred2=simulate_ar(train["GMSL_new"],fit_ar(train["GMSL_new"],p=order),m=2000)

plt.plot(range(len(y_pred2)),y_pred2,color="r")
plt.suptitle("n-step Prediction in AR Process")
plt.show()
```



We see that this converges to 0 and this is because we know that if the process is stationary with zero mean then finally it will converge to 0. This is because, after some point we will just have white noise whose expected value is 0

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 [17]:

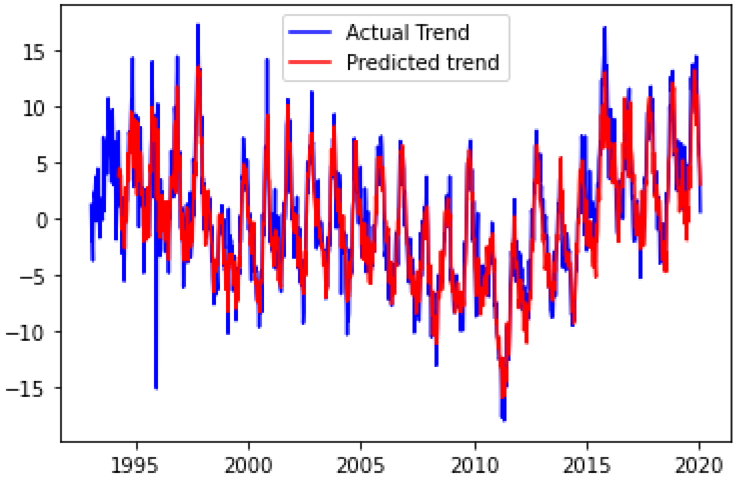
```
y1=train["GMSL_new"]
n = len(y1) # <COMPLETE THIS LINE>
p=45
    # Construct the regression matrix
Phi = np.zeros((n-p,p))
for j in range(p):
    Phi[:,j] = y1[p-j-1:n-j-1]

# Drop the first p values from the target vector y
yy = y1[p:]  # yy = (y_{t+p+1}, ..., y_n)
```

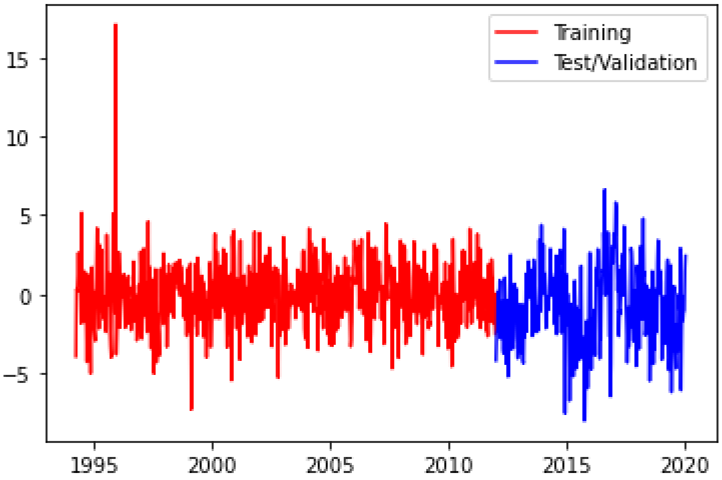
```
regr = MLPRegressor(hidden_layer_sizes=(10,),random_state=1, max_iter=500).fit(Phi, yy)
```

```
C:\ProgramData\Anaconda3\lib\site-packages\sklearn\neural_network\_multilayer_perceptro
n.py:582: ConvergenceWarning: Stochastic Optimizer: Maximum iterations (500) reached and
the optimization hasn't converged yet.
  warnings.warn(
```

In [18]:

```
y2=sealevel["GMSL_new"]

    # Number of steps in prediction
m = sealevel["GMSL_new"].shape[0]-p
y_pred = np.zeros(m)


for i in range(m):
    y_pred[i] = regr.predict((np.flip(y2[i:p+i])).values.reshape(1,-1)) # <COMPLETE THI

plt.plot("Year","GMSL_new",data=sealevel,color="b",label="Actual Trend")
plt.plot(sealevel["Year"][p:],y_pred,color="r",label="Predicted trend")
plt.legend()

plt.show()


residuals=y_pred-sealevel["GMSL_new"][p:]
plt.plot(sealevel["Year"][p:700],residuals[:700-p],color="r",label="Training")
plt.plot(sealevel["Year"][700:],residuals[700-p:],color="b",label="Test/Validation")
plt.legend()
plt.show()
```





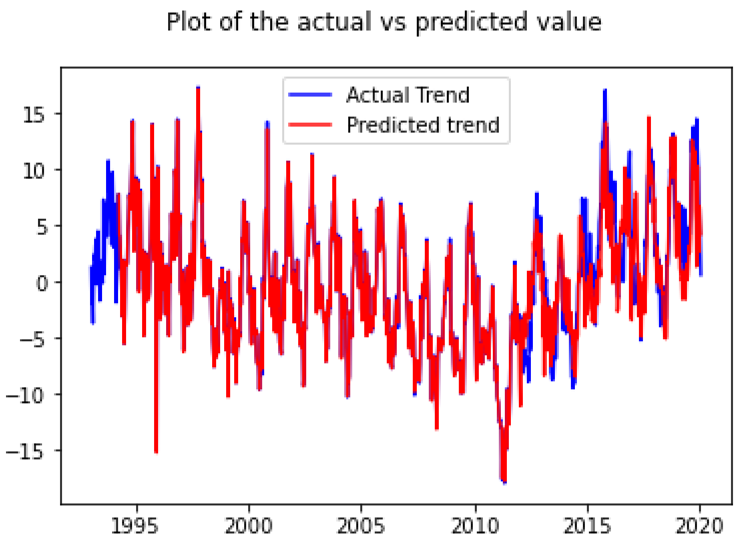
Q13: Try to expirement with different choices for the hyperparameters of the network (e.g. number of hidden layers and units per layer, activation function, etc.) and the optimizer (e.g. `solver` and `max_iter`).

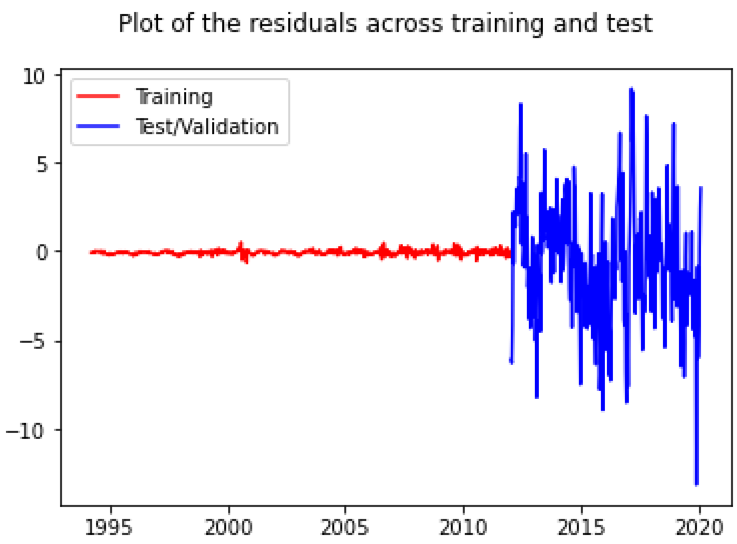
Are you satisfied with the results? Why/why not? Discuss what the limitations of this approach might be.

A13:

In [19]:
```
y1=train["GMSL_new"]
n = len(y1) # <COMPLETE THIS LINE>
p=45
    # Construct the regression matrix
Phi = np.zeros((n-p,p))
for j in range(p):
    Phi[:,j] = y1[p-j-1:n-j-1]

# Drop the first p values from the target vector y
yy = y1[p:]  # yy = (y_{t+p+1}, ..., y_n)

regr = MLPRegressor(hidden_layer_sizes=(50,50),random_state=1, max_iter=500,activation=
```

In [20]:
```
y2=sealevel["GMSL_new"]

    # Number of steps in prediction
m = sealevel["GMSL_new"].shape[0]-p
y_pred = np.zeros(m)


for i in range(m):
    y_pred[i] = regr.predict((np.flip(y2[i:p+i])).values.reshape(1,-1)) # <COMPLETE THI

plt.plot("Year","GMSL_new",data=sealevel,color="b",label="Actual Trend")
plt.plot(sealevel["Year"][p:],y_pred,color="r",label="Predicted trend")
plt.legend()
plt.suptitle("Plot of the actual vs predicted value")
plt.show()


residuals=y_pred-sealevel["GMSL_new"][p:]
plt.plot(sealevel["Year"][p:700],residuals[:700-p],color="r",label="Training")
plt.plot(sealevel["Year"][700:],residuals[700-p:],color="b",label="Test/Validation")
plt.legend()
plt.suptitle("Plot of the residuals across training and test")
plt.show()
```





After tuning some hyperparamters such as number of hidden layers=2 and 50 units in each hidden layer, we notice that there is a clear overfitting problem where the training data is perfectly fit whereas the validation data has high error. Therefore, we need to be careful with this model to not overfit our data. And in this respect trying to keep lesser parameters (such as hidden layers and units per hidden layer) is beneficial for our model.