CODE

PARAMETERS

```
class _Froe(object):
    def __init__(self, args):
        self.nE = args.ne if args.ne != None else 1
        self.nU = args.nu if args.nu != None else 1
        self.nY= args.ny if args.ny != None else 1
        self.polDegree = args.deg if args.deg != None else 1
        self.inclBias = args.incbias if args.incbias != None else True
```

These parameters are used for the model initialization, the first three represents the maximum lags for:

- "nE": Moving Average part
- "nU": Exogenous part
- "nY": Auto Regressive Part

The "polDegree" parameter is used to set the polynomial degree of the model and it is possible to include or not the bias term with the "incBias" parameter.

```
self.varianceToExplain1 = args.var1 if args.var1 != None else 0.990366
self.varianceToExplain2 = args.var2 if args.var2 != None else 0.9905
self.convergenceThresholdNARMAX = args.convth if args.convth != None else 0.05
self.maxIterationsNARMAX = args.maxit if args.maxit != None else 150
```

This group of parameters includes:

- "varianceToExplain1": stopping condition of the forward selection algorithm for NARX or first NARMAX iteration
- "varianceToExplain2": stopping condition of the forward selection algorithm for following NARMAX iterations
- "convergenceThresholdNARMAX": convergence condition for the NARMAX extension of FROE
- "maxIterationsNARMAX": limit to the number of iteration when the other convergence condition is hard to meet

Each parameter can be set by changing the default value in the code or when launching the script in this way:

```
usage: NARMAX_FROE.py [-h] [--ny NY] [--nu NU] [--ne NE] [--deg DEG]
                      [--incbias INCBIAS] [--var1 VAR1] [--var2 VAR2]
                      [--convth CONVTH] [--maxit MAXIT]

Performs identification and validation on narx/narmax models.

optional arguments:
  -h, --help          show this help message and exit
  --ny NY             The maximum lag for y (integer)
  --nu NU             The maximum lag for u (integer)
  --ne NE             The maximum lag for e (integer)
  --deg DEG           The polynomial degree of the model (integer)
  --incbias INCBIAS   Add or not Bias to the polynomial (boolean)
  --var1 VAR1         The variance to be explained for NARX or fist NARMAX
                      iteration (float)
  --var2 VAR2         The variance to be explained for following NARMAX
                      iterations (float)
  --convth CONVTH     Convergence threshold in the NARMAX case (float)
  --maxit MAXIT       Iterations limit in the NARMAX case (integer)
```

EXECUTION

DATA LOADING

```
#Read Data
dataunif = sio.loadmat('NonLinearData/DATAUNIF.mat')
u11 , u12 = np.array(dataunif["u11"]).flatten() , np.array(dataunif["u12"]).flatten()
z11 , z12 = np.array(dataunif["z11"]).flatten() , np.array(dataunif["z12"]).flatten()

dataprbs = sio.loadmat('NonLinearData/DATAPRBS.mat')
u1 , u2 , u3 = np.array(dataprbs["u1"]).flatten() , np.array(dataprbs["u2"]).flatten() , np.array(dataprbs["u3"]).flatten()
z1 , z2 , z3 = np.array(dataprbs["z1"]).flatten() , np.array(dataprbs["z2"]).flatten() , np.array(dataprbs["z3"]).flatten()

#Training Set
U, Y = u12 + u1, z12 + z1

#Validation Set
Uval1, Yval1 = u11, z11
Uval2, Yval2 = u2, z2
```

In this part of the code, the provided data are loaded and the sets for the identification part and the validation part are chosen; the choice can be modified to test the performances with other data.

IDENTIFICATION

```
print("\nStarting model with the following time delays: U", nU, "Y", nY, "E", nE, "\n")

ThetaSelected , Model = self.froe(nU, nY, nE, U, Y, np.zeros(shape = (yDimIde), dtype = np.float64), polDegree, yDimIde, inclBias,
varianceToExplain1, varianceToExplain2, 0, convergenceThresholdNARMAX, maxIterationsNARMAX)

print("Model Obtained:\n")
print(Model,"\n")

print("Results on the identification set\n")

#Prediction results on identification set
YhatPredictionIde = self.generateYhat(0, U, Y, np.zeros(shape = (yDimIde), dtype = np.float64), ThetaSelected, polDegree, inclBias, yDimIde, nU, nY, nE)
if (not YhatPredictionIde is None):
    residualsPredictionIde = self.calcResiduals(Y, YhatPredictionIde, yDimIde)
    msePredictionIde = self.calcMSE(residualsPredictionIde, yDimIde)

    #Simulation results on identification set
    YhatSimulationIde = self.generateYhat(1, U, Y, residualsPredictionIde, ThetaSelected, polDegree, inclBias, yDimIde, nU, nY, nE)
    if (not YhatSimulationIde is None):
        residualsSimulationIde = self.calcResiduals(Y, YhatSimulationIde, yDimIde)
        mseSimulationIde = self.calcMSE(residualsSimulationIde, yDimIde)

if (not YhatPredictionIde is None):
    print("MSPE" , msePredictionIde, "\n")
if (not YhatSimulationIde is None):
    print("MSSE" , mseSimulationIde, "\n")
```

First, the FROE algorithm is executed on the identification set, returning the estimated parameters of the model and the obtained model itself.

Once obtained the model, the results in terms of simulation and prediction are calculated and evaluated in terms of MSE both with the identification set and the validation sets (not shown in the figure, same code).

The real, predicted and simulated outputs are plotted.

VALIDATION

```
confidence1 = np.full(shape = (yDimVal1), fill_value = (1.96 / np.sqrt(yDimVal1)) , dtype = np.float64)
confidence2 = np.full(shape = (yDimVal1), fill_value = (1.96 / np.sqrt(yDimVal2)) , dtype = np.float64)
if (not YhatPredictionVal1 is None):
    self.plotValidationTest(confidence1, residualsPredictionVal1 ,Uval1, yDimVal1, "Prediction", "1")
if (not YhatSimulationVal1 is None):
    self.plotValidationTest(confidence1, residualsSimulationVal1 ,Uval1, yDimVal1, "Simulation", "1")
if (not YhatPredictionVal2 is None):
    self.plotValidationTest(confidence2, residualsPredictionVal2 ,Uval2, yDimVal2, "Prediction", "2")
if (not YhatSimulationVal2 is None):
    self.plotValidationTest(confidence2, residualsSimulationVal2 ,Uval2, yDimVal2, "Simulation", "2")
```

The model is validated performing the “Billings and Voon” test both on the prediction and the simulation results obtained from the validation sets. The model is considered valid if the constraints are satisfied with a 95% of confidence.

FUNCTIONS EXPLANATION

FROE

This is the core function of the whole program, where the model is identified from the dataset.

```
def froe(self, nU, nY, nE, U, Y, E, polDegree, yDim, inclBias, varianceToExplain1, varianceToExplain2, K, th, maxIt):
    if(K == 0):
        Phi = self.createPhi(U, Y, E, polDegree, inclBias, np.mean(U), np.mean(Y), yDim, nU, nY, 0)
    else:
        Phi = self.createPhi(U, Y, E, polDegree, inclBias, np.mean(U), np.mean(Y), yDim, nU, nY, nE)
    regDim = Phi.shape[1]
    Theta = np.zeros(shape = (regDim,), dtype = np.float64)
    A = np.zeros(shape = (regDim,regDim), dtype = np.float64)
    np.fill_diagonal(A, 1)
    sum_y_pow_2 = Y.T.dot(Y)
    regressorToKeep = np.array([], dtype = int)
    varianceExplained = 0
    W = np.zeros(shape = (yDim, regDim), dtype = np.float64)
    G = np.array([], dtype=np.float64)
```

In the first part ϕ is initialised (in the case [K = 0] the function is dealing with the first iteration of the NARMAX identification or with NARX identification since the MA part is not needed) along with all the needed variables

```
for j in range(regDim):
    w = np.zeros(shape=(yDim,regDim), dtype = np.float64)
    err = np.zeros(shape=(regDim), dtype = np.float64)
    g = np.zeros(shape=(regDim), dtype = np.float64)

    for i in range(regDim):
        if i not in regressorToKeep:
            wi = Phi[:,i].copy()
            for k in range(j):
                den = (W[:,k].T.dot(W[:,k]))
                akj = 0
                if (den != 0.0):
                    akj = W[:,k].T.dot(Phi[:,i]) / den
                wi -= akj * W[:,k]
                A[k,j] = akj
            w[:,i] = wi
            den_1 = (wi.T.dot(wi))
            g[i] =0
            if(den_1 != 0):
                g[i] = wi.T.dot(Y) / den_1
            err[i] = np.power(g[i], 2) * den_1 / sum_y_pow_2
    bestRegressor=np.argmax(err)
    regressorToKeep = np.append(regressorToKeep, bestRegressor)
    varianceExplained += err[bestRegressor]
    for k in range(j):
        den = (W[:,k].T.dot(W[:,k]))
        akj = 0
        if (den != 0.0):
            akj = W[:,k].T.dot(Phi[:,bestRegressor]) / den
        A[k,j] = akj
    W[:,j] = w[:,bestRegressor]
    G = np.append(G,g[bestRegressor])
    if (((1 - varianceExplained) < (1 - varianceToExplain1)) or (j == regDim - 1)):
        if (nE != 0):
            print("Iteration",K)
        print("taken" , len(regressorToKeep) , "/" , regDim , "parameters explaining" , varianceExplained, "variance\n")
        break
```

In this part, the regressors are selected based on the Error Reduction Ratio index:

- The external loop runs until the required amount of variance has been explained or all regressors have been selected; at every iteration, it chooses the regressor with the highest ERR and updates accordingly A, W and G.
- The internal loop cycles through all the non-chosen regressors, and calculates their ERR.

```
ThetaTemp = np.zeros(len(G))
for i in reversed(range(len(G))):
    if i == len(G):
        ThetaTemp[i] = G[i]
    else:
        temp = 0
        for k in range(i+1, len(G)):
            temp += A[i,k] * ThetaTemp[k]
        ThetaTemp[i] = G[i] - temp
Theta[regressorToKeep] = ThetaTemp

model = self.writeModel(Theta,regDim, nU, nY, nE, polDegree, inclBias)
```

Here the values of Θ are calculated, keeping at 0 the ones of the non-selected regressors.

Using the function “writeModel” the actual model to print is obtained.

```
if (nE != 0):
    if(K == 0):
        Yhat = W[:,0:len(G)].dot(G)

    else:
        Yhat = self.generateYhat(0, U, Y, np.zeros(shape = (yDim), dtype = np.float64), Theta, polDegree, inclBias, yDim, nU, nY, nE)
    if (Yhat is None):
        print("Impossible to identify the residuals for the NARMAX\n")
        sys.exit()
    actualResiduals = self.calcResiduals(Y, Yhat, yDim)
    if (K == 0):
        return self.froe(nU, nY, nE, U, Y, actualResiduals, polDegree, yDim, inclBias, varianceToExplain2, varianceToExplain2, K + 1, th, maxIt)
    else:
        if(self.convergence(actualResiduals, E, yDim, K, th, maxIt)):
            return Theta , model
        else:
            return self.froe(nU, nY, nE, U, Y, actualResiduals, polDegree, yDim, inclBias, varianceToExplain2, varianceToExplain2, K + 1, th, maxIt)

return Theta , model
```

When dealing with NARMAX identification, the function becomes recursive, the residuals ($\varepsilon = Y - \hat{Y}$) are calculated and if the convergence condition is not satisfied or [K = 0] the function calls itself passing them as parameters and switching from "varianceToExplain1" to "varianceToExplain2", otherwise it returns the obtained Θ and the model.

In the simple NARX case only the final return is executed.

CALC RESIDUALS

```
def calcResiduals(self, out, outhat, yDim):
    residualsTemp = np.zeros(shape=(yDim,), dtype = np.float64)
    for i in range(yDim):
        residualsTemp[i] = out[i]-outhat[i]
    return residualsTemp
```

This is how the residuals are calculated where the parameter "out" represents the real output and "outhat" the predicted/simulated one.

CONVERGENCE

```
def convergence(self, actual, previous, yDim, K, th, maxIt):

    diff = sum(np.abs(self.calcResiduals(actual, previous, yDim))) / yDim

    return diff <= th or K >= maxIt
```

The converge condition in the NARMAX case is met when the return of this function is "True":

- If the actual number of iterations has surpassed the maximum number of iterations set in the parameters or
- $\frac{\sum_{i=0}^{yDim-1} |\varepsilon_i^K - \varepsilon_i^{K-1}|}{yDim} \leq th$ i.e. the average difference between the residuals of the actual and the previous steps is within the threshold set in the parameters.

CREATE PHI

```
def createPhi(self, inpt, outpt, resids, polDeg, incB, inMean, outMean, yDim , nU, nY, nE):

    PhiTempBasic= np.zeros(shape = (yDim, (nU + nY + nE)), dtype = np.float64)
    if (nE != 0):
        resMean = np.mean(resids)
        resVar = np.var(resids)

    for i in range(yDim):
        for u in range(nU):
            if (i - u - 1) < 0:
                PhiTempBasic[i,u] = inMean
            else:
                PhiTempBasic[i,u] = inpt[i - u - 1]
        for y in range(nY):
            if (i - y - 1) < 0:
                PhiTempBasic[i,y + nU] = outMean
            else:
                PhiTempBasic[i,y + nU] = outpt[i - y - 1]
        for e in range(nE):
            if (i - e - 1) < 0:
                PhiTempBasic[i,e + nU + nY] = resMean
            else:
                PhiTempBasic[i,e + nU + nY] = resids[i - e - 1]

    if(not np.isfinite(PhiTempBasic).all()):
        return None

    PhiTemp = pf(polDeg , include_bias = incB).fit_transform(PhiTempBasic)
    return PhiTemp.copy()
```

With this function, the matrix ϕ is created.

The basic version contains only the 1st degree elements; once this version is filled, every row of the matrix is expanded to the desired polynomial degree.

CALC MSE

```
def calcMSE(self, res, yDim):
    MSE=0
    if (not np.isfinite(res).all()):
        return np.inf
    for i in range(yDim):
        MSE += np.power(res[i],2)
    MSE /= yDim
    return MSE
```

If all the residuals are finite, they are used to calculate the MSE, otherwise it is assumed to be infinite.

GENERATE YHAT

```
def generateYhat(self, type, u, y, e, theta, polDegree, inclBias, yDim, nU, nY, nE):
    if (type == 0 and nE == 0):
        Phi = self.createPhi(u, y, e, polDegree, inclBias, np.mean(u), np.mean(y), yDim, nU, nY, nE)
        Yhat= Phi.dot(theta)
    else:
        if(nE != 0 and type != 0):
            resMean = np.mean(e)
            resStdDev = np.std(e)
        Yhat = np.zeros(shape = (yDim), dtype = np.float64)

        Phi = self.createPhi(u, np.zeros(shape = (yDim), dtype = np.float64), np.zeros(shape = (yDim), dtype = np.float64),
         1, False, np.mean(u), 0.0, yDim, nU, nY, nE)
        for i in range(yDim):
            try:
                Yhat[i] = pf(polDegree , include_bias = inclBias).fit_transform(Phi[i,:].reshape(1,-1)).dot(theta)
                if(nE != 0 and type != 0):
                    res = np.random.normal(resMean, resStdDev)
                for j in range(nY):
                    if((i + j + 1) > (yDim - 1)):
                        break
                    if (type != 0):
                        Phi[i + j + 1,nU + j] = Yhat[i]
                    else:
                        Phi[i + j + 1,nU + j] = y[i]
                for k in range(nE):
                    if((i + k + 1) > (yDim - 1)):
                        break
                    if (type == 0):
                        Phi[i + k + 1, nU + nY + k] = y[i] - Yhat[i]
                    else:
                        Phi[i + k + 1, nU + nY + k] = res
            except ValueError:
                if (type == 0):
                    print("Model Divergent in prediction\n")
                    return None
                else:
                    print("Model Divergent in simulation\n")
                    return None
    return Yhat
```

This function is in charge to generate predicted [type = 0] or simulated outputs.

In the simplest case (NARX prediction) $\hat{Y} = \phi\hat{\Theta}$.

Otherwise the calculation is accomplished iteratively:

1. A basic ϕ with only the inputs is created
2. For every time instant (row of the ϕ matrix)
   2.1. $\hat{Y}$ of the current time instant is calculated
   2.2. ϕ is filled with the correct values for the next time instants:
      2.2.1. Prediction case: the past Y are equals to the real outputs and the residuals are calculated as $Y[i] - \hat{Y}[i]$
      2.2.2. Simulation case: the past Y are equals to the previously simulated outputs and the residuals are generated by a GWN process with mean and standard deviation obtained by the residuals from the prediction case
3. The possibility of divergence is handled

WRITE MODEL

```
def writeModel(self, Theta, regDim, nU, nY, nE, polDegree, inclBias):
    tempModelBasic = list()
    for i in range(nU + nY + nE):
        if (i < nU):
            tempModelBasic.append("u(t - {})".format(i + 1))
        elif (i < nU + nY):
            tempModelBasic.append("y(t - {})".format(i - nU + 1))
        else:
            tempModelBasic.append("e(t - {})".format(i - (nU + nY) + 1))

    tempModel = list()
    currentPos = nU + nY + nE
    startPos = 0
    if(inclBias):
        tempModel.append("1")
        currentPos +=1
        startPos = 1

    tempModel[startPos:currentPos] = tempModelBasic

    for i in range(2, polDegree + 1):
        temp = list(it.combinations_with_replacement(tempModelBasic, i))
        temp = np.array([reduce(add, x) for x in temp])
        nextPos = currentPos + len(temp)
        tempModel[currentPos:nextPos] = temp
        currentPos = nextPos

    for i in range(regDim):
        tempModel[i] = "{}*{}".format(Theta[i],tempModel[i])
    remove_idx = list()
    for i in range(regDim):
        if(Theta[i] == 0.0):
            remove_idx.append(i)
    model = [i for j, i in enumerate(tempModel) if j not in remove_idx]

    return model
```

Here, accordingly to the parameters found the model is composed to be easily readable.

CODE MODIFICATIONS FOR SRR CRITERION

Since, as mentioned before, it's has been hard to find a good robust simulation model, the froe function has been modified in order to use the SRR criterion instead of the ERR in this way:

- The variable "actualMSSE" has been added for the SRR calculation
- The outer loop has now an array "msse" containing all MSSEs of new potential models (i.e. adding one of the missing potential regressors) and it selects a new regressor only if its SRR is positive
- The inner loop will now calculate the SRR for each new potential regressor:
  - ```
    Atemp = A.copy()
    Gtemp = G.copy()
    Wtemp = W.copy()
    regressorToKeepTemp = regressorToKeep.copy()
    ```
    these copied variables will be used to simulate the addition of the potential regressor.
  - ```
    regressorToKeepTemp = np.append(regressorToKeepTemp, i)
    for k in range(j): #updates A based on selected regressor
        den = (Wtemp[:,k].T.dot(Wtemp[:,k]))
        akj = 0
        if (den != 0.0):
            akj = Wtemp[:,k].T.dot(Phi[:,i]) / den
        Atemp[k,j] = akj
    Wtemp[:,j] = w[:,i]
    Gtemp = np.append(Gtemp,g[i])
    ThetaTemp = self.calcTheta(Gtemp, Atemp, regressorToKeepTemp, regDim)
    ```
    Instead of calculating the ERR, now it calculates the parameters that one would obtain keeping the current regressor
  - ```
    tempResids = np.zeros(shape = (yDim), dtype = np.float64)
    YhatPred = None
    if (K != 0):
        YhatPred = self.generateYhat(0, U, Y, tempResids, ThetaTemp, polDegree, inclBias, yDim, nU, nY, nE, True)
        if (YhatPred is not None):
            tempResids= self.calcResiduals(Y,YhatPred,yDim)

    if (K == 0 or( K != 0 and YhatPred is not None)):
        nrE = 0
        if (K != 0):
            nrE = nE

        YhatSim = self.generateYhat(1, U, Y, tempResids, ThetaTemp, polDegree, inclBias, yDim, nU, nY, nrE, True)

        if (YhatSim is not None):
            res = self.calcResiduals(Y, YhatSim, yDim)
            MSSEi = self.calcMSE(res, yDim)
            msse[i] = MSSEi
            srr[i] = (actualMSSE - MSSEi) / (sum_y_pow_2 / yDim)
        else:
            srr[i] = -999999.0
    else:
        srr[i] = -999999.0
    ```
    The obtained parameters are used to simulate the output and to calculate the MSSE and the resulting SRR.
    - For the NARMAX case, before the simulation, also the predicted output is calculated and its residuals will be used to estimate the parameters of the GWN feeding the simulation process

The Internal loop has been modified in order run in parallel, since the computational time is problematic and needs a speed-up; inside the FROE function, where the "Parallel" function is called, the number of cores to use can be set by changing the "n_jobs" value.
    ```