Fuzzy and Artificial Neural Networks-Based Intelligent Control Systems Using Python

Article · June 2019

DOI: 10.11594/nstp.2019.0221

2 authors:

Basuki Rahmat
Universitas Pembangunan Nasional Veteran Jawa Timur, Surabaya, Indonesia
49 PUBLICATIONS **73** CITATIONS
SEE PROFILE

Bangkit Nugroho
Lampung University
4 PUBLICATIONS **11** CITATIONS
SEE PROFILE

NST PROCEEDINGS
OPEN ACCESS

Conference Paper

Fuzzy and Artificial Neural Networks-Based Intelligent Control Systems Using Python

B. Rahmat*, B. Nugroho

Informatics Engineering Department, Universitas Pembangunan Nasional "Veteran" Jawa Timur, Surabaya, East Java, Indonesia

Abstract

This research proposes intelligent system programming based on Fuzzy and Artificial Neural Networks (ANN). Programming is built using the Python Programming Language. The control system is used based on Proportional Integral Derivative (PID) Controller, where the gain tuning uses Fuzzy and ANN. The research stages include the preparation of plant models, fuzzy and ANN programming libraries using Python. And then how to show the performance of the Intelligent Control System that was built in the form of a simulation.

Keywords: ANN, Control, Fuzzy, Intelligent, Python

INTRODUCTION

Now intelligent control systems play an important role in the industrial world. It is undeniable that two of the intelligent system algorithms that are often used today are fuzzy systems and artificial neural networks. So that fuzzy-based and neural network-based intelligent control systems are often the solution to current intelligent system control. Meanwhile the Proportional Integral Derivative (PID) based control system remains the mainstay of the industry up to now. In turn, so that the development of intelligent system technology can contribute to the development of current control technology, PID controllers are combined with intelligent systems. In this case, especially fuzzy-based and neural networks. The method that allows for the incorporation of PID with an artificial intelligence system, is during the process of selecting the best gain from PID. In the term control this is called tuning or generally called PID gain tuning.

In the current development, research on PID gain tuning based on fuzzy and neural networks has been done a lot. But it is still interesting to develop. It is proven that until now there are still many studies on this matter. Some of them can be mentioned as follows. PID gain tuning uses a fuzzy system as described in the following papers: (Rodríguez-Castellanos, Grisales-Palacio, & Cote-Ballesteros, 2018), (El-samahy & Shamseldin, 2018), (Dettori, Iannino, Colla, & Signorini, 2018), (Khan, Pasupuleti, & Jidin, 2018), (Asgharnia, Shahnazi, & Jamali, 2018), (Gheisarnejad, 2018), (Verma, Manik, & Jain, 2018), (Huang et al., 2018), (Huang et al., 2018). PID gain tuning uses artificial neural networks, among others, as described in the following papers: (Yu, 2018), (Anh, 2010), (Milovanović et al., 2016), (Attaran, Yusof, & Selamat, 2016), (Fang, Zhuo, & Lee, 2010), (Chu & Teng, 1999).

Furthermore, to realize how fuzzy algorithms and neural networks can be implemented in hardware controls, especially those using PID controllers, the Python programming language can be one of the best alternative choices. At least from its open source nature and very complete and reliable library support. So this study proposes the use of the Python programming language for the implementation of fuzzy and neural algorithms used for PID gain tuning.

* Corresponding author

Email address: basukirahmat.if@upnjatim.ac.id

METHODS

The intelligent control system designed is PID based. The PID controller in its work automatically adjusts the control output based on the difference between the set point (SP) and the measured process variable (PV), as a control error e(t). The controller output value u(t) is transferred as the system input. Each relationship is used as shown in Equations (1) and (2).

$$e(t) = SP - PV \tag{1}$$

$$u(t) = u_{bias} + K_c\, e(t) + \frac{K_c}{\tau_I}\int_0^t e(t)\,dt - K_c\,\tau_D\,\frac{d(PV)}{dt} \tag{2}$$

The term ubias is a constant that is usually set to the value u(t) when the first controller switches from manual mode to automatic mode. This gives a "bumpless" transfer if the error is zero when the controller is turned on. The three tuning values for the PID controller are controller gain, Kc, integral time constant, τI, and derivative time constant, τD. The value of Kc is a multiplier on proportional errors and the terms integral and higher values make the controller more aggressive in response to errors from the set point. The integral time constant, τI, (also known as integral reset time) must be positive and have a unit of time. Because τI is getting smaller, the term integral is bigger because τI is in the denominator. The constant time derivative, τD also has a unit of time and must be positive. The set point (SP) is the target value and the process variable (PV) is a measured value that may deviate from the desired value. The error of the set point is the difference between SP and PV and is defined as e(t) = SP - PV.

Furthermore, for implementation purposes Discrete PID Controller is used. The digital controller is implemented with a discrete sampling period and a separate form of the PID equation is needed to estimate the integral of errors and derivatives. This modification replaces the continuous form of the integral with the sum of errors and uses Δt as the time between sample sampling and nt as the number of samples taken. It also replaces derivatives with derivative versions or other methods filtered to estimate instant slope (PV). Equation (2) if stated in digital form as shown in Equation (3).

$$u(t) = u_{bias} + K_c\, e(t) + \frac{K_c}{\tau_I}\sum_{i=1}^{n_t} e_i(t)\,\Delta t - K_c\,\tau_D\,\frac{PV_{n_t} - PV_{n_t-1}}{\Delta t} \tag{3}$$

From Equation (3) it can be seen 3 determinants of the success of the control process, namely Kc, τI and τD. The search process or setting to obtain the best gain from Kc, τI and τD is called the gain tuning process. In this study, the tuning process is proposed using fuzzy and artificial neural network. The PID gain tuning process uses fuzzy and artificial neural networks that we propose as shown in Figure 1 and Figure 2.

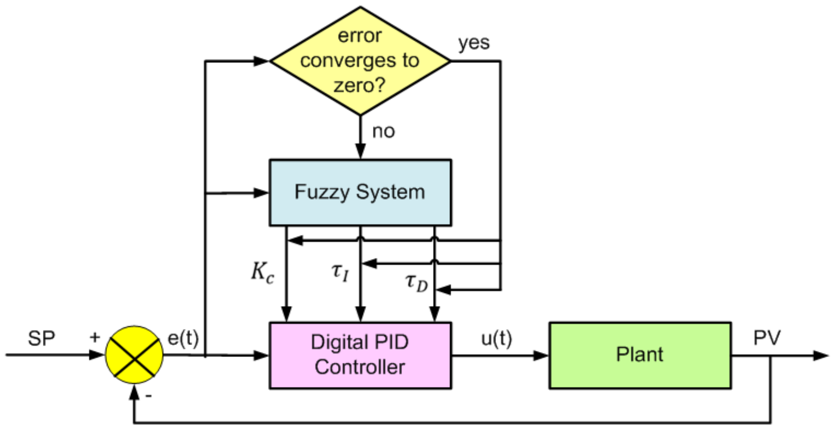

Figure 1. PID gain tuning using Fuzzy

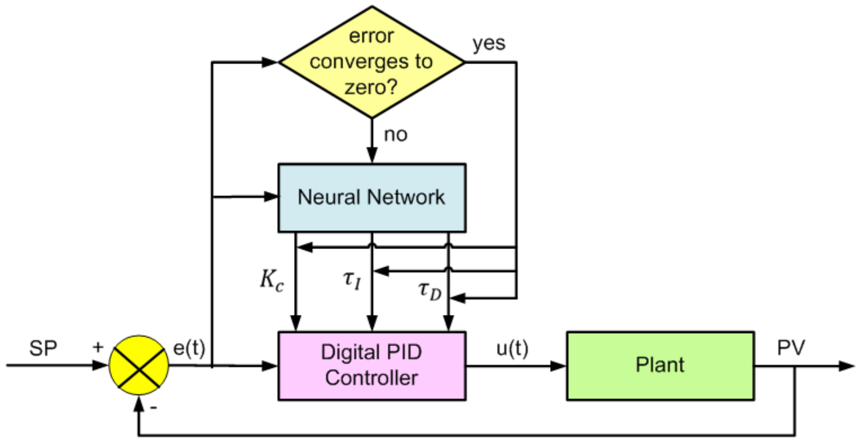

Figure 2. PID gain tuning using artificial neural network

The fuzzy system design for the PID gain tuning process as shown in Figure-1 requires at least a mechanism for how to make gain adjustments from PID (Kc, τI and τD), based on reading errors e(t) and delta error Δe(t). A reading of errors and delta error is used to decide whether or not to change the PID gain. If the error has converged towards zero, then the existing gain is maintained. But if it is still far from converging towards zero, it is necessary to change the gain following the rule that has been designed. The fuzzy system for this purpose is shown in Figure-3. Fuzzy system rules for changing gain are shown in Table 1.

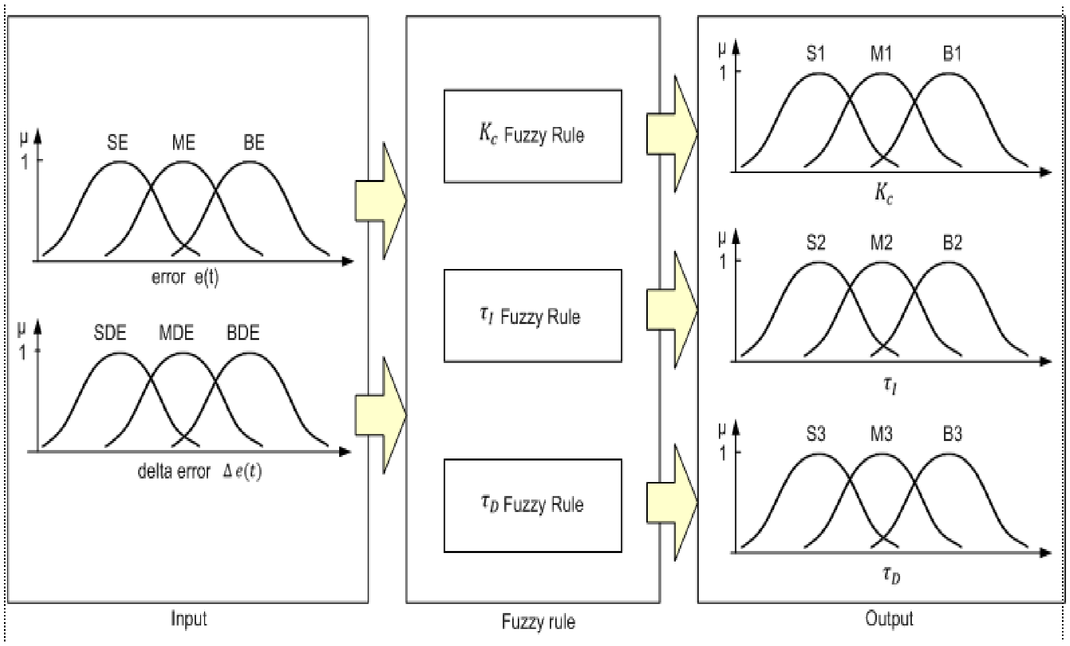

Figure 3. Fuzzy System Design

Table 1. Fuzzy Rule System

Rule	Input		Output		
	error	delta_error	K_c	τ_I	τ_D
1	SE	SDE	S1	S2	S3
2	SE	MDE	S1	S2	S3
3	SE	BDE	S1	S2	S3
4	ME	SDE	M1	M2	M3
5	ME	MDE	M1	M2	M3
6	ME	BDE	M1	M2	M3
7	BE	SDE	B1	B2	B3
8	BE	MDE	B1	B2	B3
9	BE	BDE	B1	B2	B3

The artificial neural network design for the PID gain tuning process, as shown in Figure-2, requires a mechanism for making gain adjustments from PID (Kc, τI and τD), based on reading errors e(t) and delta error Δe(t). A reading of errors and delta error is used to decide whether or not to change the PID gain. If the error has converged towards zero, then the existing gain is maintained. But if it is still far from converging towards zero, it is necessary to change the gain by generating artificial neural network weights. The artificial neural network architecture for this purpose is shown in Figure 4.

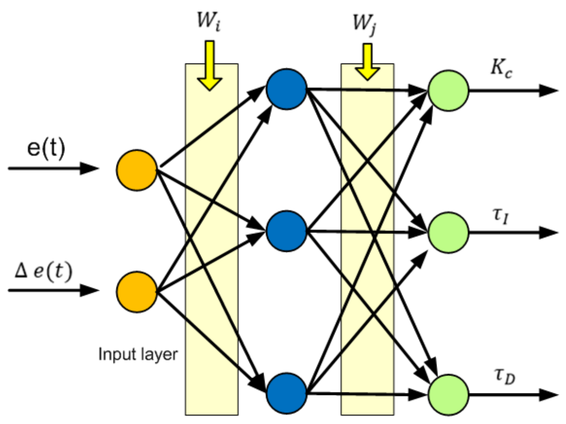

Figure 4. Artificial neural network architecture for PID gain tuning

RESULT AND DISCUSSION

To realize how the Python implementation for the PID gain tuning process, Python simulation is first made for the control system using PID, with a manual tuning process. If the manual tuning process has been successful, the task of the fuzzy system and the artificial neural network is to imitate the tuning process and choose PID gain which has the best control effect. Python programming for manually controlling simulation with PID controller, with a linear plant model, is shown in the following program.

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.integrate import odeint
import ipywidgets as wg
from IPython.display import display
n = 100 # time points to plot
tf = 50.0 # final time
SP_start = 2.0 # time of set point change

def process(y,t,u):
    Kp = 4.0
    taup = 3.0
    thetap = 1.0
    if t<(thetap+SP_start):
        dydt = 0.0  # time delay
    else:
        dydt = (1.0/taup) * (-y + Kp * u)
    return dydt

def pidPlot(Kc,tauI,tauD):
    t = np.linspace(0,tf,n) # create time vector
    P= np.zeros(n)          # initialize proportional term
    I = np.zeros(n)         # initialize integral term
    D = np.zeros(n)         # initialize derivative term
    e = np.zeros(n)         # initialize error
    OP = np.zeros(n)        # initialize controller output
    PV = np.zeros(n)        # initialize process variable
    SP = np.zeros(n)        # initialize setpoint
    SP_step = int(SP_start/(tf/(n-1))+1) # setpoint start
    SP[0:SP_step] = 0.0     # define setpoint
    SP[SP_step:n] = 4.0     # step up
    y0 = 0.0                # initial condition
    # loop through all time steps
    for i in range(1,n):
        # simulate process for one time step
        ts = [t[i-1],t[i]]         # time interval
        y = odeint(process,y0,ts,args=(OP[i-1],))  # compute next step
        y0 = y[1]                  # record new initial condition
        # calculate new OP with PID
        PV[i] = y[1]                # record PV
        e[i] = SP[i] - PV[i]        # calculate error = SP - PV
        dt = t[i] - t[i-1]          # calculate time step
        P[i] = Kc * e[i]            # calculate proportional term
        I[i] = I[i-1] + (Kc/tauI) * e[i] * dt  # calculate integral term
        D[i] = -Kc * tauD * (PV[i]-PV[i-1])/dt # calculate derivative term
        OP[i] = P[i] + I[i] + D[i] # calculate new controller output

    # plot PID response
    plt.figure(1,figsize=(15,7))
    plt.subplot(2,2,1)
    plt.plot(t,SP,'k-',linewidth=2,label='Setpoint (SP)')
    plt.plot(t,PV,'r:',linewidth=2,label='Process Variable (PV)')
    plt.legend(loc='best')
    plt.subplot(2,2,2)
    plt.plot(t,P,'g.-',linewidth=2,label=r'Proportional = $K_c \; e(t)$')
    plt.plot(t,I,'b-',linewidth=2,label=r'Integral = $\frac{K_c}{\tau_I} \int_{i=0}^{n_t} e(t) \; dt $')
    plt.plot(t,D,'r--',linewidth=2,label=r'Derivative = $-K_c \tau_D \frac{d(PV)}{dt}$')
    plt.legend(loc='best')
    plt.subplot(2,2,3)
    plt.plot(t,e,'m--',linewidth=2,label='Error (e=SP-PV)')
    plt.legend(loc='best')
    plt.subplot(2,2,4)
    plt.plot(t,OP,'b--',linewidth=2,label='Controller Output (OP)')
    plt.legend(loc='best')
    plt.xlabel('time')

Kc_slide = wg.FloatSlider(value=0.1,min=-0.2,max=1.0,step=0.05)
tauI_slide = wg.FloatSlider(value=4.0,min=0.01,max=5.0,step=0.1)
tauD_slide = wg.FloatSlider(value=0.0,min=0.0,max=1.0,step=0.1)
wg.interact(pidPlot, Kc=Kc_slide, tauI=tauI_slide, tauD=tauD_slide)
```

The purpose of using fuzzy systems and artificial neural networks is how to do the PID gain tuning process, which is to find the appropriate Kc, τI and τD values, so that the best control results are obtained. Good control results are indicated by a significant decrease in error. An example of implementing fuzzy based PID gain tuning programming using the Python programming language is partly shown in the following program script.

```
# Import NumPy and scikit-fuzzy

import numpy as np

import skfuzzy as fuzz


# Generate universe functions

ERROR       = np.arange(0, 5, 0.01)

DELTA_ERROR = np.arange(0, 5, 0.01)



Kc   = np.arange(-0.2, 1.0, 0.01)

tauI = np.arange(0.01, 5.0, 0.01)

tauD = np.arange(0.0, 1.0, 0.01)



# Membership functions for ERROR

SE = fuzz.gaussmf(ERROR, 0.01, 0.8495)

ME = fuzz.gaussmf(ERROR, 2.5, 0.8495)

BE = fuzz.gaussmf(ERROR, 5, 0.8495)



# Membership functions for DELTA_EROR

SDE = fuzz.gaussmf(DELTA_ERROR, 0.01, 0.8495)

MDE = fuzz.gaussmf(DELTA_ERROR, 2.5, 0.8495)

BDE = fuzz.gaussmf(DELTA_ERROR, 5, 0.8495)
```

```
# Membership functions for OUTPUT_Kc, OUTPUT_tauI, OUTPUT_tauD

S1 = fuzz.gaussmf(Kc, -0.2, 0.2039)

M1 = fuzz.gaussmf(Kc, 0.4, 0.2039)

B1 = fuzz.gaussmf(Kc, 1, 0.2039)



S2 = fuzz.gaussmf(tauI, 0.01, 0.8476)

M2 = fuzz.gaussmf(tauI, 2.505, 0.8476)

B2 = fuzz.gaussmf(tauI, 5, 0.8476)



S3 = fuzz.gaussmf(tauD, 6.939e-18, 0.1699)

M3 = fuzz.gaussmf(tauD, 0.5, 0.1699)

B3 = fuzz.gaussmf(tauD, 1, 0.1699)


def ERROR_category(ERROR_in = 5):

    ERROR_cat_SMALL   = fuzz.interp_membership(ERROR, SE, ERROR_in)

    ERROR_cat_MEDIUM  = fuzz.interp_membership(ERROR, ME, ERROR_in)

    ERROR_cat_BIG     = fuzz.interp_membership(ERROR, BE, ERROR_in)

    return dict(SMALL_ERROR = ERROR_cat_SMALL, MEDIUM_ERROR = ERROR_cat_MEDIUM,
BIG_ERROR = ERROR_cat_BIG)
```

```
def DELTA_ERROR_category(DELTA_ERROR_in = 5):

    DELTA_ERROR_cat_SMALL   = fuzz.interp_membership(DELTA_ERROR, SDE, DELTA_ERROR_in)

    DELTA_ERROR_cat_MEDIUM  = fuzz.interp_membership(DELTA_ERROR, MDE, DELTA_ERROR_in)

    DELTA_ERROR_cat_BIG     = fuzz.interp_membership(DELTA_ERROR, BDE, DELTA_ERROR_in)

    return dict(SMALL_DELTA_ERROR = DELTA_ERROR_cat_SMALL, MEDIUM_DELTA_ERROR = DELTA_ERROR_cat_MEDIUM, BIG_DELTA_ERROR = DELTA_ERROR_cat_BIG)


# RULE for OUTPUT_Kc

rule1 = np.fmax(ERROR_in['SMALL_ERROR'], DELTA_ERROR_in['SMALL_DELTA_ERROR'])

rule2 = np.fmax(ERROR_in['SMALL_ERROR'], DELTA_ERROR_in['MEDIUM_DELTA_ERROR'])

rule3 = np.fmax(ERROR_in['SMALL_ERROR'], DELTA_ERROR_in['BIG_DELTA_ERROR'])

rule4 = np.fmax(ERROR_in['MEDIUM_ERROR'], DELTA_ERROR_in['SMALL_DELTA_ERROR'])

rule5 = np.fmax(ERROR_in['MEDIUM_ERROR'], DELTA_ERROR_in['MEDIUM_DELTA_ERROR'])

rule6 = np.fmax(ERROR_in['MEDIUM_ERROR'], DELTA_ERROR_in['BIG_DELTA_ERROR'])

rule7 = np.fmax(ERROR_in['BIG_ERROR'], DELTA_ERROR_in['SMALL_DELTA_ERROR'])

rule8 = np.fmax(ERROR_in['BIG_ERROR'], DELTA_ERROR_in['MEDIUM_DELTA_ERROR'])

rule9 = np.fmax(ERROR_in['BIG_ERROR'], DELTA_ERROR_in['BIG_DELTA_ERROR'])



# RULE for tau_I

rule10 = np.fmax(ERROR_in['SMALL_ERROR'], DELTA_ERROR_in['SMALL_DELTA_ERROR'])

rule11 = np.fmax(ERROR_in['SMALL_ERROR'], DELTA_ERROR_in['MEDIUM_DELTA_ERROR'])
```

```
rule12 = np.fmax(ERROR_in['SMALL_ERROR'], DELTA_ERROR_in['BIG_DELTA_ERROR'])

rule13 = np.fmax(ERROR_in['MEDIUM_ERROR'], DELTA_ERROR_in['SMALL_DELTA_ERROR'])

rule14 = np.fmax(ERROR_in['MEDIUM_ERROR'], DELTA_ERROR_in['MEDIUM_DELTA_ERROR'])

rule15 = np.fmax(ERROR_in['MEDIUM_ERROR'], DELTA_ERROR_in['BIG_DELTA_ERROR'])

rule16 = np.fmax(ERROR_in['BIG_ERROR'], DELTA_ERROR_in['SMALL_DELTA_ERROR'])

rule17 = np.fmax(ERROR_in['BIG_ERROR'], DELTA_ERROR_in['MEDIUM_DELTA_ERROR'])

rule18 = np.fmax(ERROR_in['BIG_ERROR'], DELTA_ERROR_in['BIG_DELTA_ERROR'])



# RULE for tau_D

rule19 = np.fmax(ERROR_in['SMALL_ERROR'], DELTA_ERROR_in['SMALL_DELTA_ERROR'])

rule20 = np.fmax(ERROR_in['SMALL_ERROR'], DELTA_ERROR_in['MEDIUM_DELTA_ERROR'])

rule21 = np.fmax(ERROR_in['SMALL_ERROR'], DELTA_ERROR_in['BIG_DELTA_ERROR'])

rule22 = np.fmax(ERROR_in['MEDIUM_ERROR'], DELTA_ERROR_in['SMALL_DELTA_ERROR'])

rule23 = np.fmax(ERROR_in['MEDIUM_ERROR'], DELTA_ERROR_in['MEDIUM_DELTA_ERROR'])

rule24 = np.fmax(ERROR_in['MEDIUM_ERROR'], DELTA_ERROR_in['BIG_DELTA_ERROR'])

rule25 = np.fmax(ERROR_in['BIG_ERROR'], DELTA_ERROR_in['SMALL_DELTA_ERROR'])

rule26 = np.fmax(ERROR_in['BIG_ERROR'], DELTA_ERROR_in['MEDIUM_DELTA_ERROR'])

rule27 = np.fmax(ERROR_in['BIG_ERROR'], DELTA_ERROR_in['BIG_DELTA_ERROR'])


# IMPLICATION for Kc

imp1 = np.fmax(rule1, S1)
```

```
imp2 = np.fmax(rule2, S1)

imp3 = np.fmax(rule3, S1)

imp4 = np.fmax(rule4, M1)

imp5 = np.fmax(rule5, M1)

imp6 = np.fmax(rule6, M1)

imp7 = np.fmax(rule7, B1)

imp8 = np.fmax(rule7, B1)

imp9 = np.fmax(rule7, B1)



# IMPLICATION for tauI

imp10 = np.fmax(rule10, S2)

imp11 = np.fmax(rule11, S2)

imp12 = np.fmax(rule12, S2)

imp13 = np.fmax(rule13, M2)

imp14 = np.fmax(rule14, M2)

imp15 = np.fmax(rule15, M2)

imp16 = np.fmax(rule16, B2)

imp17 = np.fmax(rule17, B2)

imp18 = np.fmax(rule18, B2)



# IMPLICATION for tauD
```

```
imp19 = np.fmax(rule19, S3)

imp20 = np.fmax(rule20, S3)

imp21 = np.fmax(rule21, S3)

imp22 = np.fmax(rule22, M3)

imp23 = np.fmax(rule23, M3)

imp24 = np.fmax(rule24, M3)

imp25 = np.fmax(rule25, B3)

imp26 = np.fmax(rule26, B3)

imp27 = np.fmax(rule27, B3)


# Aggregate all output - min

aggregate_membership1 = np.fmax(imp1, np.fmax(imp2, np.fmax(imp3, np.fmax(imp4, np.fmax(imp5,
np.fmax(imp6, np.fmax(imp7, np.fmax(imp8,imp9))))))))

aggregate_membership2 = np.fmax(imp10, np.fmax(imp11, np.fmax(imp12, np.fmax(imp13, np.fmax(imp14,
np.fmax(imp15, np.fmax(imp16, np.fmax(imp17,imp18))))))))

aggregate_membership3 = np.fmax(imp19, np.fmax(imp20, np.fmax(imp21, np.fmax(imp22, np.fmax(imp23,
np.fmax(imp24, np.fmax(imp25, np.fmax(imp26,imp27))))))))


# Defuzzification

result_Kc   = fuzz.defuzz(Kc, aggregate_membership1 , 'centroid')

result_tauI = fuzz.defuzz(tauI, aggregate_membership2 , 'centroid')

result_tauD = fuzz.defuzz(tauD, aggregate_membership3 , 'centroid')

print (result_Kc)
print (result_tauI)

print (result_tauD)
```

From the PID gain tuning example using this fuzzy system, the results of Kc = 0.39331459212203296, τI = 2.4983242909359484, and τD = 0.49331226163799125, as shown in Figure 5.

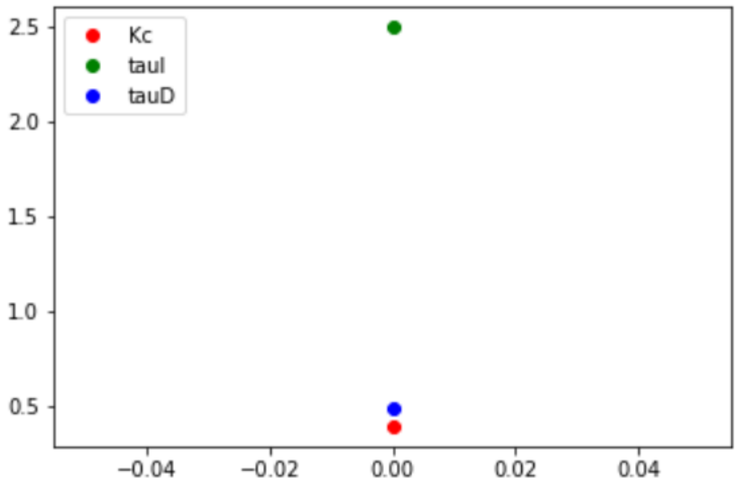

Figure 5. PID gain tuning using Fuzzy

The effect of the PID gain tuning results from the fuzzy system mechanism, then seen on the success of the control process. The results of the control process with the PID controller using the fuzzy tuning gain process are shown in Figure 6.

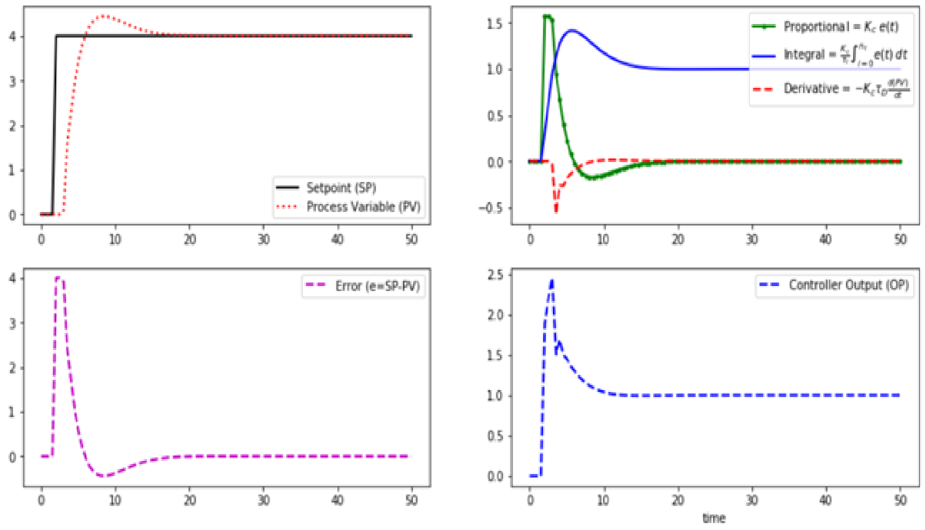

Figure 6. PID gain tuning control results using Fuzzy

Examples of PID programming gain tuning based on artificial neural networks using the Python programming language are partially shown in the following program script.

```
import numpy as np # For matrix math

import matplotlib.pyplot as plt # For plotting

import sys # For printing


num_i_units = 2 # Number of Input units

num_h_units = 3 # Number of Hidden units

num_o_units = 3 # Number of Output units


# The learning rate for Gradient Descent.

learning_rate = 0.01

# The parameter to help with overfitting.

reg_param = 0

# Maximum iterations for Gradient Descent.

max_iter = 100

# Number of training examples

m = 4

#Generating the Weights and Biases

np.random.seed(1)

W1 = np.random.normal(0, 1, (num_h_units, num_i_units)) # 3x2

W2 = np.random.normal(0, 1, (num_o_units, num_h_units)) # 3x3
```

```
B1 = np.random.random((num_h_units, 1)) # 3x1

B2 = np.random.random((num_o_units, 1)) # 3x1


def train(_W1, _W2, _B1, _B2): # Neural Network Training

    for i in range(max_iter):

        c = 0

        dW1 = 0

        dW2 = 0

        dB1 = 0

        dB2 = 0



        for j in range(m):

            sys.stdout.write("\rIteration: {} and {}".format(i + 1, j + 1))



            # Forward Prop.

            a0 = X[j].reshape(X[j].shape[0], 1) # 2x1



            z1 = _W1.dot(a0) + _B1 # 3x2 * 2x1 + 3x1 = 3x1

            a1 = sigmoid(z1) # 3x1
            z2 = __W2.dot(a1) + __B2 # 3x2 * 2x1 + 3x1 = 3x1

            a2 = sigmoid(z2) # 3x1



            # Back prop.

            dz2 = a2 - y[j].reshape(y[j].shape[0], 1) # 3x1

            dW2 += dz2 * a1.T # 3x1 .* 1x3 = 3x3



            dz1 = np.multiply((__W2.T).dot(dz2), sigmoid(a1, derv=True)) # (3x3 * 3x1) .* 3x1 = 3x1

            dW1 += dz1.dot(a0.T) # 3x1 * 1x2 = 3x2



            dB1 += dz1 # 3x1

            dB2 += dz2 # 3x1



            c = c + (-(y[j].reshape(y[j].shape[0], 1) * np.log(a2)) - ((1 - y[j].reshape(y[j].shape[0], 1)) * np.log(1 - a2)))

            sys.stdout.flush() # Updating the text.



        __W1 = __W1 - learning_rate * (dW1 / m) + ( (reg_param / m) * __W1)

        __W2 = __W2 - learning_rate * (dW2 / m) + ( (reg_param / m) * __W2)



        __B1 = __B1 - learning_rate * (dB1 / m)

        __B2 = __B2 - learning_rate * (dB2 / m)
```

```
 return (_W1, _W2, _B1, _B2)


# Testing

 for j in range(1):

        a0 = coba[j].reshape(coba[j].shape[0], 1) # 2x1

        z1 = W1.dot(a0) + B1 # 3x2 * 2x1 + 3x1 = 3x1

        a1 = sigmoid(z1) # 3x1

        z2 = W2.dot(a1) + B2 # 3x2 * 2x1 + 3x1 = 3x1

        outNN = sigmoid(z2) # 3x1
```

From the PID gain tuning example using this artificial neural network, Kc = 0.43504464, τI = 0.99994792, and τD = 0.27455905, as shown in Figure 7.

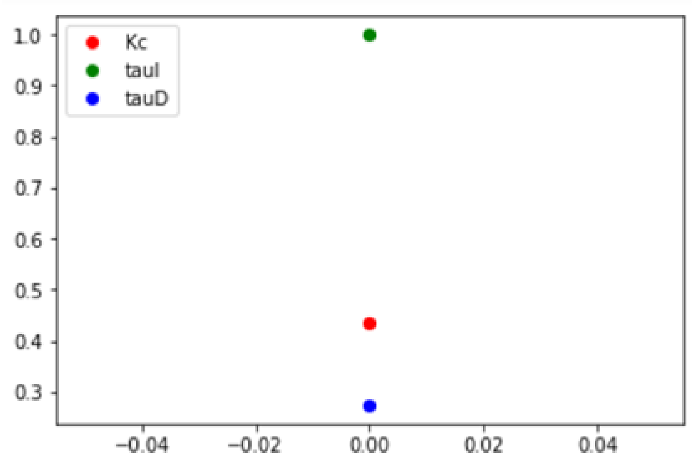

Figure 7. PID gain tuning using Artificial Neural Network

PID gain tuning obtained from artificial neural network systems, then the effect is seen on the success of control. The results of the control process with PID controllers with the gain tuning process using the artificial neural network are shown in Figure 8.

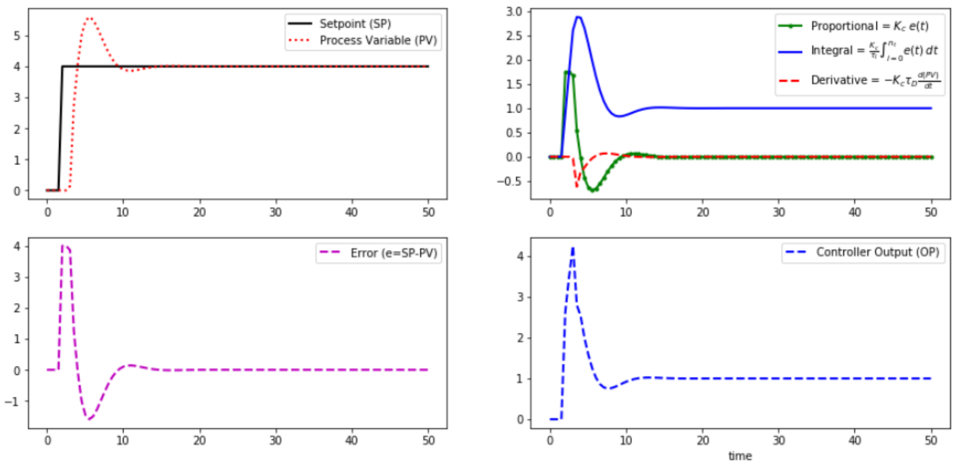

Figure 8. PID gain tuning control results using Artificial Neural Network

CONCLUSION

From the experimental results it can be shown that the Python programming language can be used as an alternative to realize a simulation of Fuzzy and Artificial Neural Networks (ANN)-based intelligent control systems. Where most intelligent control systems currently use the Proportional Integral Derivative (PID) Controller, fuzzy and neural are mostly used for the tuning process rather than the PID gain. The simulation results show the effect of gain changes on the success of the control process.

REFERENCES

Anh, H.P.H. (2010). Online tuning gain scheduling MIMO neural PID control of the 2-axes pneumatic artificial muscle (PAM) robot arm. *Expert Systems with Applications, 37(9)*, 6547–6560.

Asgharnia, A., Shahnazi, R., and Jamali, A. (2018). Performance and robustness of optimal fractional fuzzy PID controllers for pitch control of a wind turbine using chaotic optimization algorithms. *ISA Transactions, 79,* 27–44.

Attaran, S. M., Yusof, R., and Selamat, H. (2016). A novel optimization algorithm based on epsilon constraint-RBF neural network for tuning PID controller in decoupled HVAC system. *Applied Thermal Engineering, 99,* 613–624.

Chu, S.-Y., and Teng, C.-C. (1999). Tuning of PID controllers based on gain and phase margin specifications using fuzzy neural network. *Fuzzy Sets and Systems, 101(1),* 21–30.

Dettori, S., Iannino, V., Colla, V., and Signorini, A. (2018). An adaptive Fuzzy logic-based approach to PID control of steam turbines in solar applications. *Applied Energy, 227,* 655–664.

El-samahy, A. A., and Shamseldin, M. A. (2018). Brushless DC motor tracking control using self-tuning fuzzy PID control and model reference adaptive control. *Ain Shams Engineering Journal, 9(3),* 341–352.

Fang, M.-C., Zhuo, Y.-Z., and Lee, Z.-Y. (2010). The application of the self-tuning neural network PID controller on the ship roll reduction in random waves. *Ocean Engineering, 37(7),* 529–538.

Gheisarnejad, M. (2018). An effective hybrid harmony search and cuckoo optimization algorithm based fuzzy PID controller for load frequency control. *Applied Soft Computing, 65*, 121–138.

Huang, H., Zhang, S., Yang, Z., Tian, Y., Zhao, X., Yuan, Z., ... Wei, Y. (2018). Modified Smith fuzzy PID temperature control in an oil-replenishing device for deep-sea hydraulic system. *Ocean Engineering, 149,* 14–22.

Khan, M. R. B., Pasupuleti, J., and Jidin, R. (2018). Load frequency control for mini-hydropower system: A new approach based on self-tuning fuzzy proportional-derivative scheme. *Sustainable Energy Technologies and Assessments, 30*, 253–262.

Milovanović, M. B., Antić, D. S., Milojković, M. T., Nikolić, S. S., Perić, S. L., and Spasić, M. D. (2016). Adaptive PID control based on orthogonal endocrine neural networks. *Neural Networks, 84,* 80–90.

Rodríguez-Castellanos, J. E., Grisales-Palacio, V. H., and Cote-Ballesteros, J. E. (2018). A tuning proposal for direct fuzzy PID controllers oriented to industrial continuous processes. *IFAC-PapersOnLine, 51(4),* 657–662.

Verma, O. P., Manik, G., and Jain, V. K. (2018). Simulation and control of a complex nonlinear dynamic behavior of multi-stage evaporator using PID and Fuzzy-PID controllers. *Journal of Computational Science, 25*, 238–251.

Yu, W. (2018). *PID Control with Intelligent Compensation for Exoskeleton Robots*. Cambridge: Academic Press.