TRABALHO PRÁTICO

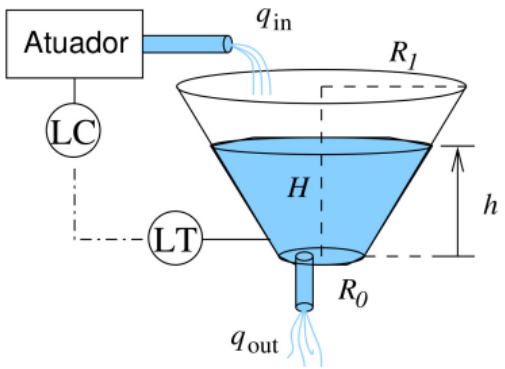
Helen da Silva Ikeda – 2019021360
Matheus José Fonseca Franklin – 2018014212

1. INTRODUÇÃO

O presente trabalho prático é referente à aplicação dos assuntos abordados na disciplina de Sistemas Distribuídos para Automação. O objetivo do trabalho é aplicar os conceitos aprendidos na disciplina a um sistema composto por uma tanque industrial que deve ser simulado e controlado, além de enviar informações para um sistema supervisório responsável pela teleoperação do controle do tanque.

2. DESCRIÇÃO DO PROBLEMA

Temos um tanque industrial que possui uma vazão de entrada q_{in} controlável, uma saída q_{out} não controlável e uma altura $h(t)$ variável. O desenho do sistema está logo a seguir:

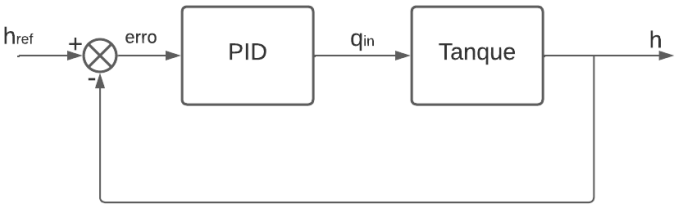


As características construtivas do tanque foram definidas iguais a:

- R1 (raio superior do tanque) = 8 metros
- R0 (raio inferior do tanque) = 5 metros
- H (altura total do tanque) = 10 metros
- Cv (coeficiente de descarga da saída do tanque) = 0.75

É necessário, portanto, criar os programas integrantes do sistema. Um dos programas será o *tanque_conico*, responsável por simular o comportamento dinâmico do tanque apresentado acima. O outro programa será chamado de *CLP* e deverá conter um cliente OPC para ler e atuar no processo via servidor, além de um servidor TCP/IP, responsável por receber conexões de clientes TCP/IP que desejam controlar o processo. O processo será o *synoptic_process*, responsável por garantir o interfaceamento com o usuário ao receber um valor de referência de altura do teclado, armazenar os dados em um arquivo *.txt* e comunicar através de TCP/IP com o *tanque_conico*.

Para facilitar no entendimento do sistema e começar a planejar como e em que ordem o código seria feito, o seguinte diagrama de blocos foi projetado:



Vemos que na prática estaremos iterativamente calculando o erro através da diferença da saída do nosso tanque e da altura desejada e escolhida pelo usuário. Esse erro passará pelo controlador PID que, sintonizado corretamente, deve retornar um valor de q_{in} que diminua o erro cada vez mais, até que esteja suficientemente próximo de zero.

3. DESENVOLVIMENTO DO CÓDIGO

Para iniciar o desenvolvimento do código, a estratégia de dividir o código em 3 partes em arquivos diferentes foi adotada para realizar testes modulares e garantir o funcionamento individual dos programas antes de uní-los. As partes foram planta, controle e comunicação TCP/IP, e foram desenvolvidas nessa ordem. Cada parte será melhor explicada ao decorrer dos próximos tópicos.

As bibliotecas utilizadas ao longo de todo o desenvolvimento do código estão reunidas abaixo:

```
import numpy as np
from scipy.integrate import odeint
from opcua import Client
import time
from simple_pid import PID
import socket
import threading
```

3.1. PLANTA

O requisito do desenvolvimento do programa responsável por simular o comportamento dinâmico do tanque industrial é utilizar o método de integração por Runge-Kutta.

3.1.1. RUNGE-KUTTA

Os métodos de Runge-Kutta formam uma família de métodos numéricos utilizados para resolver equações diferenciais ordinárias. O método mais utilizado é o

método de Runge-Kutta de quarta ordem, que foi o selecionado para realizar a aproximação numérica que desejamos.

A dinâmica não linear que representa o comportamento da planta é dada por

$$\dot{h}(t) = \frac{-C_v\sqrt{h(t)}}{\pi[R_0 + \alpha h(t)]^2} + \frac{1}{\pi[R_0 + \alpha h(t)]^2}u(t)$$

O código começa, portanto, com a definição dessa equação:

```
# Definição da EDO
def edo(h, t, cv, r0, a, u):
    return (-cv*np.sqrt(h))/(np.pi*(r0 + a*h)**2) + u/(np.pi*(r0 + a*h)**2)
```

As variáveis do processo foram declaradas e estão a seguir:

```
# Definição das variáveis da dinâmica do tanque
cv = 0.75           # coeficiente de descarga da saída do tanque
r0 = 5              # raio da base do tanque (menor raio)
r1 = 8              # raio do topo do tanque (raio maior)
H = 10              # altura do tanque
a = (r1 - r0)/H     # alpha
```

Para resolver uma equação diferencial ordinária, é necessário definir suas condições iniciais. Portanto, para este problema, consideramos uma condição inicial nula:

```
# Solução da EDO para condição inicial
h0 = 0
t = np.linspace(0, 10, 1001)
sol = odeint(edo, h0, t, args=(cv, r0, a, u))
```

O método Runge-Kutta consiste de fato em estimar um valor da função em vários pontos intermediários. O ponto escolhido será a média ponderada entre os pontos intermediários citados. Ele é baseado na série de Taylor e sua ordem é definida de acordo com a ordem da série de Taylor.

Temos, portanto:

$$
\begin{aligned}
k_1 &= f\left(t_n, y_n\right) \\
k_2 &= f\left(t_n + \frac{h}{2}, y_n + \frac{h}{2}k_1\right) \\
k_3 &= f\left(t_n + \frac{h}{2}, y_n + \frac{h}{2}k_2\right) \\
k_4 &= f\left(t_n + h, y_n + hk_3\right)
\end{aligned}
$$

Sendo k_1 o termo inicial, k_2 e k_3 os termos intermediários e k_4 o termo final. A cada iteração, y será calculado por:

$$y_{n+1} = y_n + \frac{h}{6}\left(k_1 + 2k_2 + 2k_3 + k_4\right)$$

O código desenvolvido para este algoritmo ficou da seguinte forma:

```
# Método de integração de Runge-Kutta
def rungekutta4(f, h0, t, args=()):
    n = len(t)
    h = np.zeros((n, 1))
    h[0] = h0
    for i in range(n - 1):
        height = t[i+1] - t[i]
        k1 = f(h[i], t[i], *args)
        k2 = f(h[i] + k1 * height / 2., t[i] + height / 2., *args)
        k3 = f(h[i] + k2 * height / 2., t[i] + height / 2., *args)
        k4 = f(h[i] + k3 * height, t[i] + height, *args)
        h[i+1] = h[i] + (height / 6.) * (k1 + 2*k2 + 2*k3 + k4)
    return h[i+1]
```

Temos então uma função *tanque* que tem como argumento a vazão de entrada do tanque e que engloba todo o código explanado acima, com esse adicional ao final da função:

```
# Solução pelo método de Runge-Kutta
t = np.linspace(0, 10, 1001)

sol = rungekutta4(edo, h0, t, args=(cv, r0, a, u))
sol = np.round(sol, 4)

qout = cv*np.sqrt(sol)
qout = np.round(qout, 4)

u = np.round(u, 4)

return sol, u, qout
```

Dentro da própria função, utilizamos a função *rungekutta4* para resolver a EDO a partir do argumento passado para a nossa função *tanque*. A função *round()* é utilizada para arredondar as casas decimais do valor e, como podemos ver, nossa função *tanque* retornará os valores de altura atual, vazão de entrada do tanque e vazão de saída. Essas saídas serão essenciais para as próximas etapas.

3.2. CONTROLE

O controle foi realizado separadamente para que pudesse ser testado com a planta antes de utilizá-lo na comunicação TCP/IP. O controlador escolhido foi o PID e seu desenvolvimento está nos tópicos a seguir.

3.2.1. PID

Para desenvolver o controlador PID, foi utilizada a biblioteca *simple_pid*. Foi utilizada somente a função *PID()*, que tem como argumento os ganhos *kp, ki* e *kd* e o valor de referência, chamado de *setpoint*.

```
pid = PID(3, 9, 0.05, setpoint = href)
```

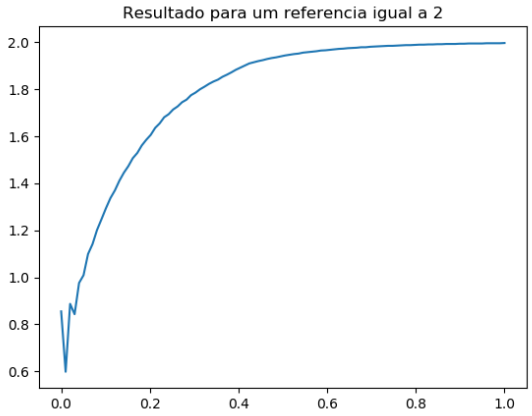
Os valores foram determinados de forma empírica, através do método de tentativa e erro. O *setpoint* até então foi definido como constante para testes, afinal esse *setpoint* posteriormente será definido pelo usuário através de uma entrada do teclado.

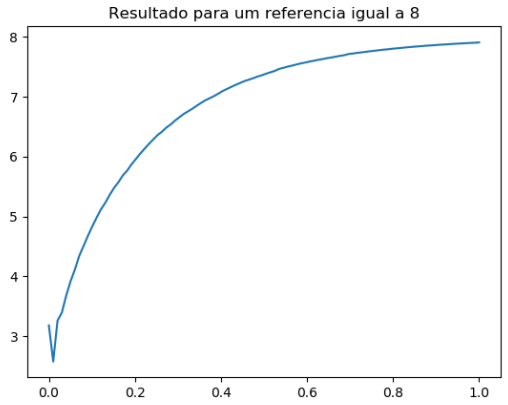
O código completo é bem simples. Consiste em manter em loop o controle atuando no processo, considerando os valores de forma iterativa.

```
pid = PID(3, 9, 0.05, setpoint=href)
h, qin, qout = tanque(0)
while True:
    href = node_h_ref.get_value()
    #a finalidade do 'for' é para não enviar muitas requisições ao servidor
    #opc desnecessariamente
    for _ in range(5):
        pid.setpoint = href
        control = pid(h)
        h, qin, qout = tanque(control)
    node_q_in.set_value(qin[0])
    node_h.set_value(h[0])
    time.sleep(0.5)
```

3.2.2. TESTE

O teste foi realizado diretamente com a função *tanque* e foi gerado um gráfico para permitir uma melhor visualização e entender se a resposta atinge a referência. Segue o resultado obtido pelo teste:





Podemos ver que para ambas as referências, o resultado rastreou corretamente a referência definida.

3.3. TCP/IP

A comunicação via TCP/IP entre o *CLP* e o *synoptic_process* foi feita utilizando a biblioteca *socket*. O *synoptic_process* deve simular um sistema supervisório para realizar a teleoperação do controle do tanque. Podemos pensar que seria a tela e os controles que ficam dispostos dentro de uma sala de controle. Ou seja, o *synoptic_process* terá interfaceamento com o usuário, além de solicitar informações ao *CLP*. Portanto, nosso *synoptic_process* será o cliente TCP/IP e o *CLP* será o servidor TCP/IP e o cliente OPC.

3.3.1. CLIENTE

A comunicação entre cliente e servidor foi desenvolvida para ficar o mais simples possível, portanto o cliente e o servidor se comunicam através de uma porta fixa, definida em “1024”, afinal portas abaixo de 1023 são reservadas.

Para uma melhor visualização, as linhas de código referentes à *print* e *write* foram suprimidas em arquivo externo. Esses detalhes serão vistos mais adiante. A estrutura do código cliente inicia da seguinte forma:

```
HOST = "127.0.0.1"      # The server's hostname or IP adress
PORT = 1024             # The port used by the server

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.connect((HOST, PORT))

print("\nInsira a altura desejada: ")
ref = input()
```

Um socket *s* é inicializado e se conecta ao *host* e *porta* definidos. Além disso, a variável *ref* recebe uma entrada do teclado. Precisamos enviar essa referência continuamente ao servidor para que ele possa calcular e retornar os valores de controle. Temos o seguinte loop para isso:

```
while True:
    s.sendall(ref.encode())
    hist = s.recv(1024)

    h, qin, alarme = hist.split(b',')
    h = h.decode()
    qin = qin.decode()
    alarme = alarme.decode()

    h = str(h).replace("[", " ")
    h = str(h).replace("]", " ")

    qin = str(qin).replace("[", " ")
    qin = str(qin).replace("]", " ")

    alarme = str(alarme).replace("[", " ")
    alarme = str(alarme).replace("]", " ")
```

O *while* envia a variável *ref* e recebe em uma variável *hist* a resposta do servidor. Assim que recebe esse dado, é necessário conferir se o dado chegou de fato, pois se não chegou, encerramos o *while* para não dar nenhum erro. Com o *hist* entrando no loop, utilizamos a função *split()* para separar o dado de acordo com as vírgulas e o *decode()* para transformar de *bytes* para *string*. As linhas abaixo apenas tratam os dados ao retirar os caracteres desnecessários para mostrar ao usuário.

Além disso, foi solicitado como requisito de projeto que o operador seja capaz de receber sinais de aviso quando o nível do tanque estiver muito alto ou muito baixo. Com isso, os requisitos foram implementados da seguinte forma:

```
if alarme == '1':
    print("PERIGO: NIVEL BAIXO\n")
elif alarme == '2':
    print("PERIGO: NÍVEL ALTO\n")
```

3.3.2. SERVIDOR

Assim como no código do cliente, o *host* e a *porta* foram definidos de forma a serem fixos e conectarem sem maiores problemas com o cliente. Um socket *s* é inicializado, dá um *bind* e fica escutando até que o cliente conecte. O servidor aceita a comunicação e entra em um *while*, no qual receberá a referência do cliente. Temos aqui a criação do cliente OPC, o qual recebe o caminho de acesso e utiliza a função *connect* para realizar a conexão. Nós são criados através do *get_node* para as variáveis de altura atual, altura de referência e vazão de entrada.

```
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.bind((HOST, PORT))

s.listen()
print("Esperando conexoes...")

conn, addr = s.accept()
with conn:
    print(f"Conectado por {addr}")

    # cria o client
    client = Client("opc.tcp://localhost:52520/OPCUA/SimulationServer")

    client.connect()

    node_h = client.get_node("ns=3;i=1008")
    node_q_in = client.get_node("ns=3;i=1009")
    node_h_ref = client.get_node("ns=3;i=1010")

    mutex.acquire()
    while True:
        href = conn.recv(1024)
        href = float(href.decode())
        node_h_ref.set_value(href)

        h = node_h.get_value()
        q_in = node_q_in.get_value()
```

Para realizar as condições de acionamento do alarme, o seguinte código foi implementado:

```
#condições para acionamento do alarme
#a altura do tanque é 10m
h_tanque = 10
if (h/h_tanque)<0.05:
    alarme = 1
elif (h/h_tanque)>0.95:
    alarme = 2
else:
    alarme = 0
```

Após isso, as saídas obtidas do *tanque_conico* serão concatenadas com vírgulas entre si para serem enviadas ao cliente, que como visto anteriormente, irá separar e tratar esses dados. O código está a seguir:

```
    aux = str(h)+str(v)+str(q_in)+str(v)+str(alarme)
    data = aux.encode()
    conn.sendall(data)
```

3.3.3. VISUALIZAÇÃO

É necessário gravar os dados de altura atual, vazão de entrada e vazão de saída do tanque em um arquivo *.txt* nomeado de *historiador*. Uma variável *arquivo* foi utilizada para abrir o arquivo e escrever nele. Note pelo trecho a seguir que foram adicionados alguns elementos para que o arquivo fique o mais legível possível:

```
with open('historiador.txt','w') as arquivo:
    arquivo.write("-------------------------------------\n")
    arquivo.write("            HISTORICO DA\n")
    arquivo.write("     TELA DE CONTROLE DO TANQUE\n")
```

Não há necessidade de incluir o resto do código neste momento. No entanto, ao longo de sua extensão, existem elementos de impressão que permitem ao usuário interagir com informações de forma suficientemente clara e tornam o arquivo de saída agradável e com uma interface simples, sem a necessidade de bibliotecas complementares, ou seja, eliminando uma possível complexidade desnecessária.

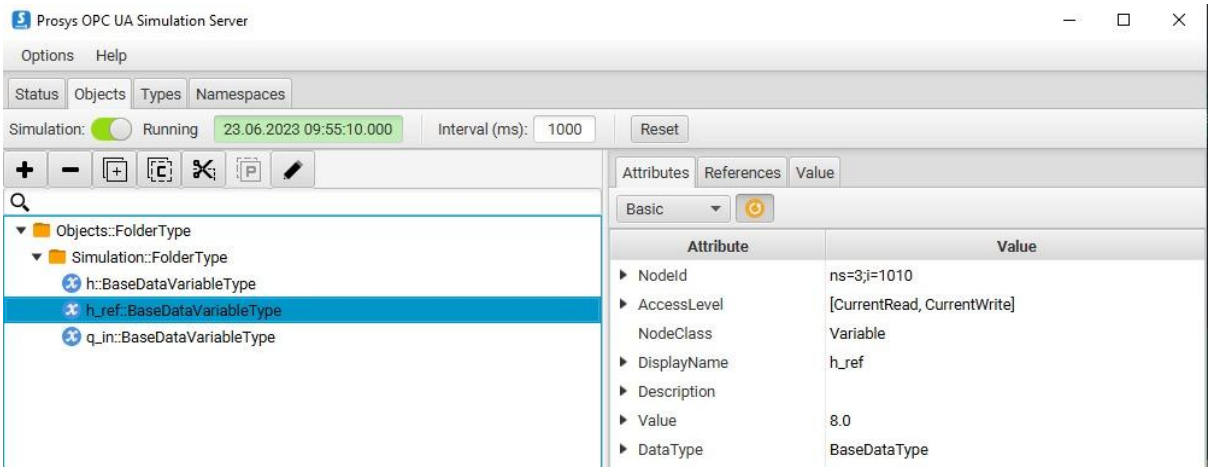
4. PROSYS

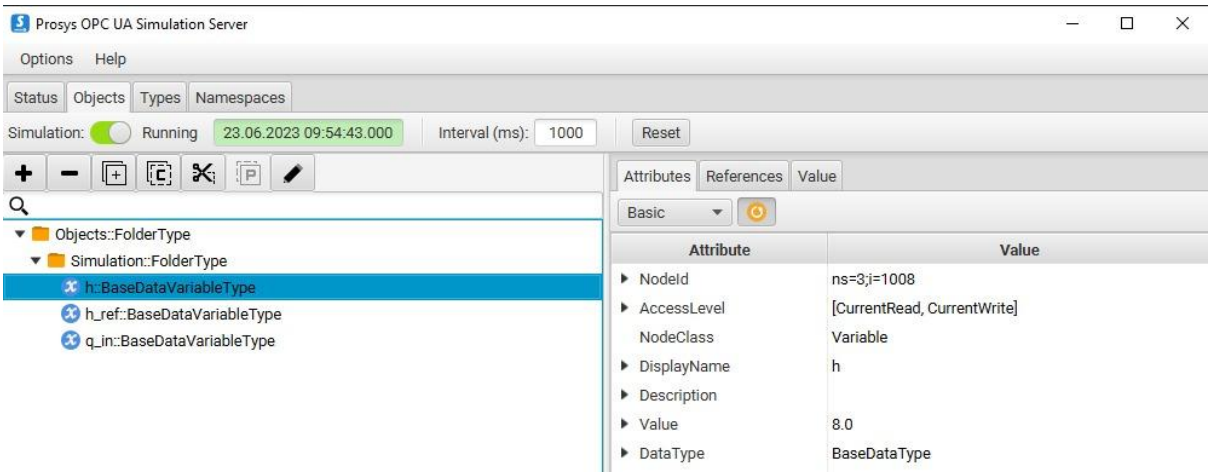
Para operar como servidor OPC, foi usado o software Prosys OPC UA Simulation Server. Na aba "Objects" foram criados 3 nós:

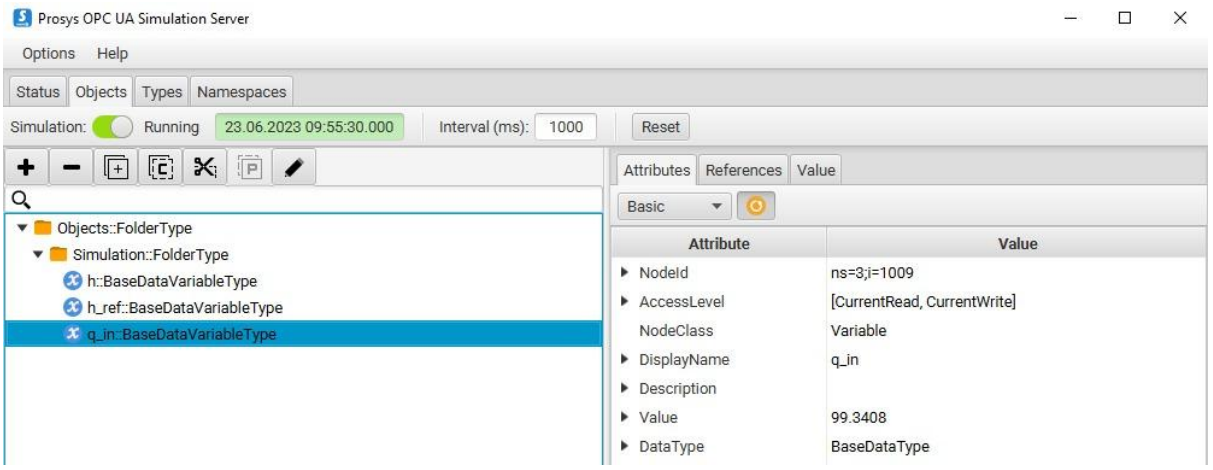
- h - indica o nível atual do tanque medido pelo sensor

- q_in - indica a vazão de entrada do tanque imposta pelo controlador
- h_ref - indica o setpoint de nível passado para o controlador

Dessa forma, é possível o controlador ler o setpoint de nível e o sensor enviar dados de supervisão via OPC.







5. RESULTADOS

A seguir, vemos o prompt de comando do cliente, o *synoptic_process,* e os dados enviados ao arquivo *historiador.txt*.

```
Insira a altura desejada:
8

Iniciando controle, aguarde...

Processo finalizado.
Altura final alcancada atraves do controle:  7.9591
Dados armazenados no arquivo historiador.txt
```

A referência utilizada para este teste foi igual a 8 metros. No prompt de comando, a última altura medida é mostrada na tela.

```
1   ----------------------------------------
2               HISTORICO DA
3       TELA DE CONTROLE DO TANQUE
4   ----------------------------------------
5   Altura desejada: 8 metros
6   ----------------------------------------
7   Dados do controle realizado:
8   ----------------------------------------
9   Altura atual =9.0
10  Vazao de entrada =117.3525
```

```
152 ----------------------------------------
153 Altura atual =8.0
154 Vazao de entrada =99.3408
```

A tela acima é o *historiador.txt*, abrangendo somente a parte inicial e final do resultado de controle para melhor visualização, afinal são muitos valores impressos no arquivo. A cada execução, os valores são sobrescritos nos valores antigos.

6. CONCLUSÃO

O desenvolvimento deste trabalho prático foi essencial para a fixação dos conceitos e métodos aprendidos durante toda a disciplina de Sistemas Distribuídos para Automação. Além disso, a multidisciplinariedade que enfrentamos entre as áreas de Controle, Redes de Computadores e Automação tornaram o trabalho ainda mais interessante.

No fim, o problema proposto foi solucionado e o controle do tanque funcionou com sucesso, assim como a comunicação com o sinótico por OPC e TCP/IP.

7. REFERÊNCIAS

https://perso.crans.org/besson/publis/notebooks/Runge-Kutta_methods_for_ODE_integration_in_Python.html

https://www.bogotobogo.com/python/Multithread/python_multithreading_Synchronization_Lock_Objects_Acquire_Release.php

https://superfastpython.com/timer-thread-in-python/

https://www.codingem.com/python-convert-bytes-to-string/