Příloha 1 - Kódové provedení simulátoru

August 10, 2022

1 Model for further simulation

The goal is to create a model for failure simulation. For parameters adjustment exploit the existing model created by Vit Pawlik with the gradient descend method.

Requirements: 1. Based on deterministic physical model 2. Ability to change inputs (weigth, temperature, voltage, etc.) 3. Ability to change parameters of assembly parts in time (change of resistance with temperature) 4. Save the models as .h5 that contains columns of inputs and outputs

Importing libraries

[ ]:
```
import numpy as np
import pandas as pd
import matplotlib.pylab as plt

from scipy import io
from scipy import signal

import time as tm
import datetime as dt

import h5py

from IPython.display import clear_output
```

Re-write hand-written state-space function to code function. To fulfil the requirement of change in constants (as k1, k2 etc.) during the simulation we have to conclude extra inputs into the ODE. Those inputs would cause the appearance of nonlinear relations such as x1.u2. We would have to solve nonlinear equations, which we solve by linearization and if these equations are linearized, they are returned to the initial state before implementing extra inputs. I.e. it seems impossible and we must find another way.

[ ]:
```
def generate_ABCD(coeficients):
    """
    Generates space-state matricies for the simplified shaker model for given␣
↪data.

    Parameters
    -----------
```

```
    coeficients : dictionary
        Dictionary of all needed coefficients that consists of:
        R : float
            Resistance of electrical resistor of electromagnetic circuit

        L : float
            Inductance of electrical coil of electromagnetic circuit

        k1 : float
            Constant representing loss of induction

        k2 : float
            Constant representing back EMF

        cc : float
            Damper constant for damping between coil and armature in Ns/m

        kc : float
            Spring constant for springing between coil and armature in N/m

        cs : float
            Damper constant for damping between armature and body in Ns/m

        ks : float
            Spring constant for springing between armature and body in N/m

        cb : float
            Damper constant for damping between body and ground in Ns/m

        kb : float
            Spring constant for springing between body and ground in N/m

        mc : float
            Mass of the armature coil in kg

        mt : float
            Mass of the armature in kg

        md : float
            Mass of the load in kg

        mb : float
            Mass of the body in kg

    Returns
    ----------
    A : list
```

```
        Calculated space state representation matrix A
    B : list
        Calculated space state representation matrix B
    C : list
        Calculated space state representation matrix C
    D : list
        Calculated space state representation matrix D. Always zero.
    """
    R=coeficients['R']; L=coeficients['L']
    k1=coeficients['k1']; k2=coeficients['k2']
    cc=coeficients['cc']; kc=coeficients['kc']
    cs=coeficients['cs']; ks=coeficients['ks']
    cb=coeficients['cb']; kb=coeficients['kb']
    mc=coeficients['mc']; mt=coeficients['mt']
    md=coeficients['md']; mb=coeficients['mb']

    A = [[0, 1, 0, 0, 0, 0, 0],
         [-kc/mc, -cc/mc, kc/mc, cc/mc, 0, 0, k1/mc],
         [0, 0, 0, 1, 0, 0, 0],
         [kc/(mt+md), cc/(mt+md), -(ks+kc)/(mt+md), -(cs+cc)/(mt+md), ks/
↪(mt+md), cs/(mt+md), 0],
         [0, 0, 0, 0, 0, 1, 0],
         [0, 0, ks/mb, cs/mb, -(kb+ks)/mb, -(cs+cb)/mb, -k1/mb],
         [0, -k2/L, 0, 0, 0, k2/L, -R/L]]

    B = [[0], [0], [0], [0], [0], [0], [1/L]]

    C = A[2:4]

    D = [[0.0]]*len(C)

    return A, B, C, D
```

1.1 MatLab model

Load MatLab model by Vit Pawlik

[ ]:
```
model_data=io.loadmat('MatLab stuff\LDS_model_20st_stred.mat')
matA = model_data['A']
matB = model_data['B']
matC = model_data['C']
matD = model_data['D']
matsys = signal.StateSpace(matA,matB,matC,matD)
```

1.2 Physical model

Physical model based on handwritten State-Space matricies. Shown parameters are adjusted based on comparation with MatLab model. Designation of every variable is based on the simplified mechanical and electrical model shown abow.

[ ]:
```
burden_weight = 0 #in kg

coef = {
    # Weights in kg
    'mc' : 0.5,
    'mt' : 2,
    'md' : burden_weight,
    'mb' : 630,

    # Springs in N/m (norm value 1e6 for car, much weaker for epoxy)
    'kc' : 1.5,
    'ks' : 1.6 ,
    'kb' : 1e6 ,

    # Dampers in Ns/m (usual range of dambers 300-2000 Ns/m)
    'cc' : 8,
    'cs' : 10 ,
    'cb' : 300,

    # Electrical stuff
    'k1' : 8 ,
    'k2' : .015 ,
    'L' : 1e-4 ,
    'R' : 2e-5
}

# State-Space equations
A, B, C, D = generate_ABCD(coef)

# Model
sys = signal.StateSpace(A,B,C,D)
```

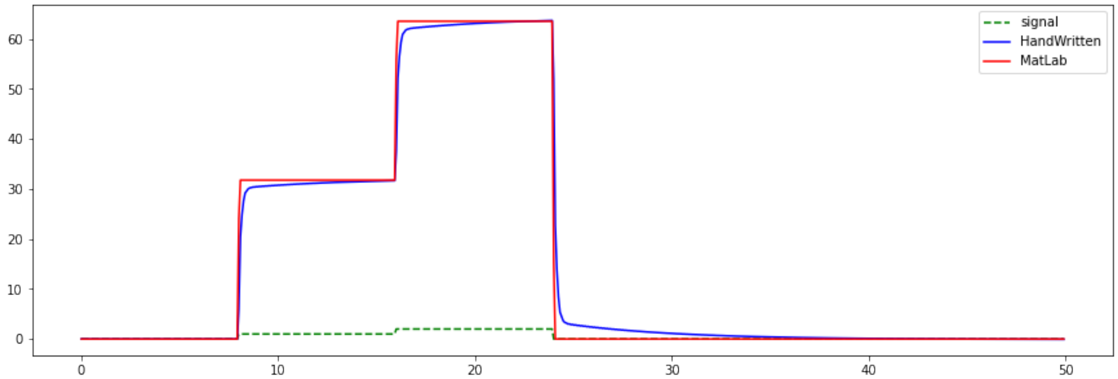
1.3 Models comparation

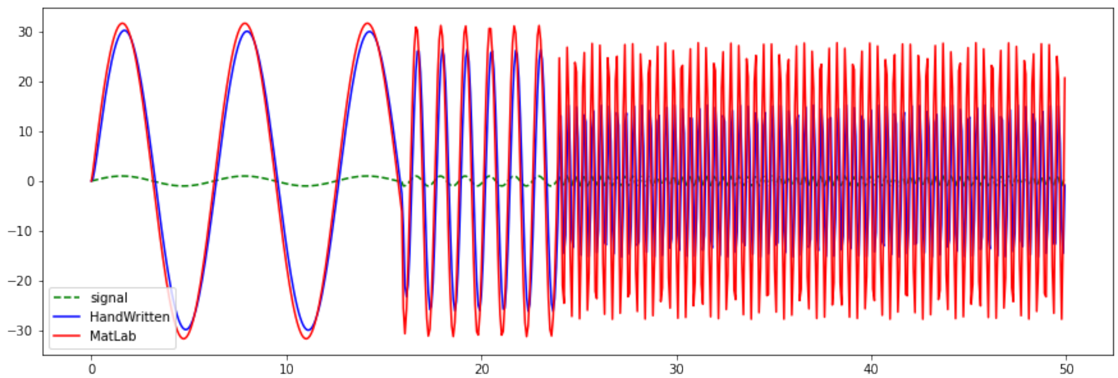
The goal is to compare both models and adjust physical parameters of physical model. We will do that by inputing different signals and comparing the output.

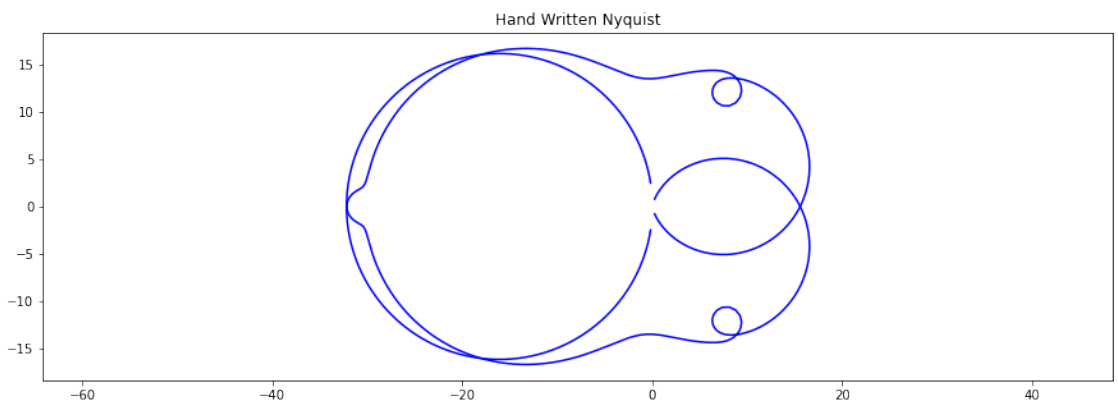
[ ]:
```
time = np.arange(0, 50, 0.08)

#Creating series of inputs
inputs = np.zeros((2, len(time)))
inputs[0] = np.zeros(len(time))
```

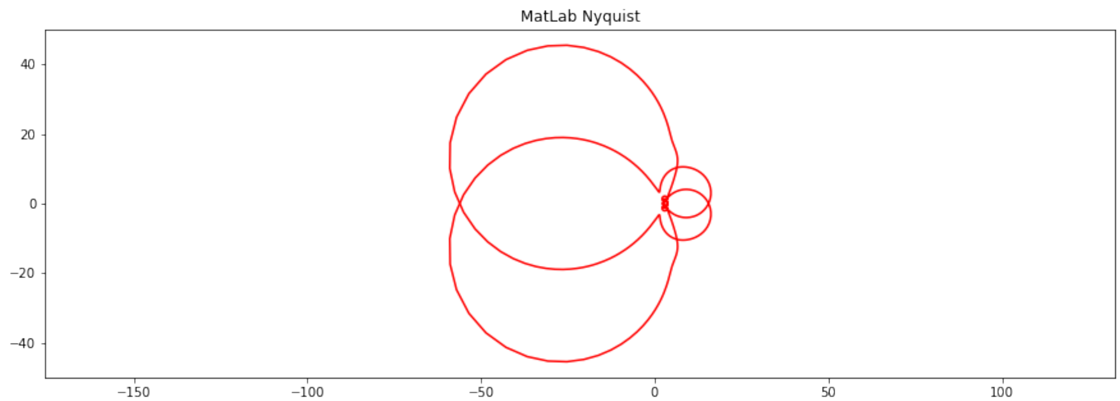
```python
inputs[0][100:]=1
inputs[0][200:]=2
inputs[0][300:]=0

inputs[1] = np.sin(time)
inputs[1][200:] = np.sin(time[200:]*5)
inputs[1][300:] = np.sin(time[300:]*15)

for u in inputs:
    t, y, x = signal.lsim(sys, u, time)
    matt, maty, matx = signal.lsim(matsys, u, time)

    plt.figure(figsize=(15,5))
    plt.plot(t, u, label='signal', linestyle='--', color='green')
    plt.plot(t, y[:,0], label='HandWritten', color='blue')
    plt.plot(matt, -maty, label='MatLab', color='red')
    plt.legend()



# Basic fucntions
w, H = signal.freqresp(signal.StateSpace(A,B,C[0],D[0]))
matw, matH = signal.freqresp(matsys)
plt.figure(figsize=(15,5))
plt.title('Hand Written Nyquist')
plt.plot(-H.real, H.imag, label='HandWritten', color='blue')
plt.plot(-H.real, -H.imag, label='HandWritten', color='blue')
plt.axis('equal')

plt.figure(figsize=(15,5))
plt.title('MatLab Nyquist')
plt.plot(matH.real, matH.imag, label='MatLab', color='red')
plt.plot(matH.real, -matH.imag, label='MatLab', color='red')
plt.axis('equal')


plt.show()
```

```
c:\ProgramData\Anaconda3\lib\site-packages\scipy\signal\filter_design.py:1631:
BadCoefficients: Badly conditioned filter coefficients (numerator): the results
may be meaningless
  warnings.warn("Badly conditioned filter coefficients (numerator): the "
```


signal
HandWritten
MatLab


signal
HandWritten
MatLab


Hand Written Nyquist



1.3.1 Conclusion

Nyquist characteristics of the created model differ significantly from the MatLab model and the main difference can be seen on higher input frequencies, where the magnitude of output drops significantly. For purposes of this work - to create tools for signal description and to predict system aging - differences in the model dynamics are not substantial and thus can be neglected for further processing.

1.4 Signal generation

The goal is to create signal that will be changed over time due to aging of parts. Following models must be created: 1. Clear model without failure or noise (main.h5) 2. Model loaded with noise (lwn_main.h5) 3. Model loaded with failure (lwf_[name of the failure].h5)

Terms: 1. Real-life sampling rate from provided sensor is used, including dispersion 2. Timestamp is used as time period 3. Data are separated into aprox. 1 hour periods and 100 hours of data are generated for every model 4. Every change of parameters will appears first 10 hours and reaches its maximum after 60 hours of work 5. Large-data format is used for saving 6. Saved signal contains - Inputs: signal, weigth. Outputs: position, speed. Trained wegths. 7. Final name of model is 'in-' + noi* + '_' + nof*

Changable parameters: R, kb, cb, ks, cs, kc, cc, k1, k2, L

Failures: 1. Up to 5% change with the sigmoid function for every changable parameter (parameter) (nof* - sigmoid_5) 1. Up to 10% change with the sigmoid function for every parameter (nof* - sigmoid_20) 1. Up to 50% change with the sigmoid function for every parameter (nof* - sigmoid_20) 1. White-noise will appear of max 10% change in every parameter with linear growth. (nof* - whitenoise_10) 1. Uniform-noise will appear of max 10% change in every parameter with linear growth. (nof* - uniformnoise_10)

Inputs: 1. Low frequency sine-input for max duration (noi* - sin) 2. Random agresive input for max duration (noi* - gauss) 3. Every hour different weight of load (noi* - weight) 5. White noise input (noi* - whitenoise) 6. Sine input with slowly changable frequency for max duration (noi* - freq) 4. Every hour different weight of load and different input frequency (white, sine with changed freq, sine, etc) (noi* - all) *nof - name of the failure* noi - name of the input

First of all we need to define change strategies of parameters

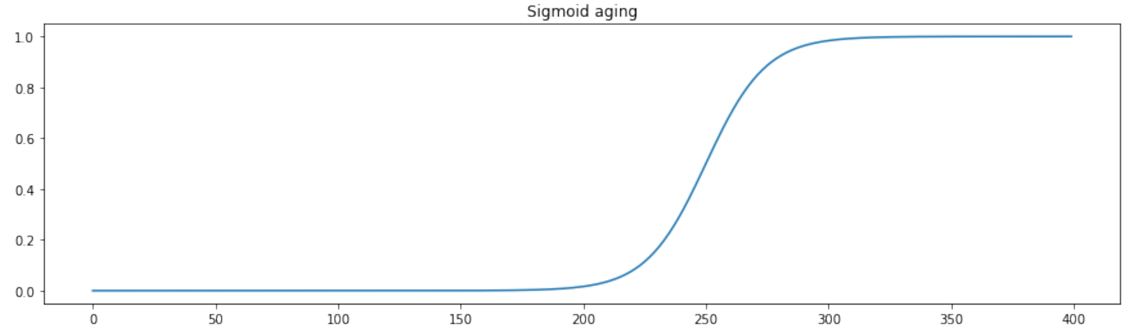
[ ]:
```
def whitenoise_aging(max_val, i, max_i, min_i=0):
    """
    Aging strategy that reduces/increases the value based on whitenoise with␣
↪linear growth
    of standard deviation (std).

    Note: Only std of white-noise is changed not mean

    Parameters
    -------------
    max_val : int, float
        Coresponds to the two times standard deviation (std).

        Separates outputed values into two groups. Group of
        numbers that are lower than max_val are achieved 95%
        of the time. Group of numbers that are higher than
        max_val are achieved 5% of the time.

    i : int, float
        Actual step

    max_i : int, float
        Step at which maximum value is achieved

    min_i: int, float, Optional
        Step at which minimum value starts to grow

    Returns
    -------------
    aged_val : float
        Returns value generated by white_noise


    """
    if i < min_i:
        gain = 0
    elif i > max_i:
        gain = 1
    else:
        m = 1/(max_i-min_i)
        b = -min_i*m
        gain = i*m+b

    mean = 0
```

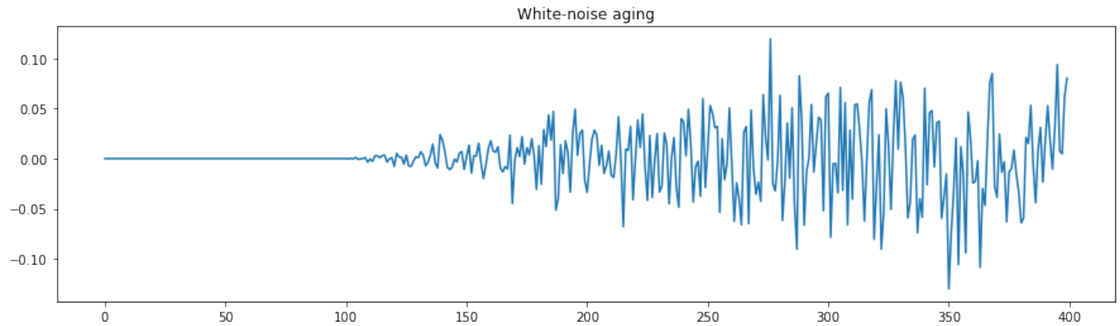
```
    std = max_val/2
    return gain*np.random.normal(mean, std)


def sigmoid_aging(max_val, i, max_i, min_i=0):
    """
    Aging strategy that reduces maximum value based on modified sigmoid␣
 ↪function.
    Sigmoid function limit goes to 0, when 'i' goes to -inf, and goes to 1 when␣
 ↪'i' goes to +inf.

    This sigmoid function is modified that the function is equal to 0, when 'i'␣
 ↪<= 0 and function is
    equal to max_val, when 'i' = max_i.

    Sigmoid can be separated into three parts. Slow growth rate at the␣
 ↪beginning, linear course most of
    the time and slow decrease in growth at the end.

    Parameters
    -------------
    max_val : int, float
        Maximum value that will be achieved

    i : int, float
        Actual step

    max_i : int, float
        Step at which maximum value is achieved

    min_i: int, float, Optional
        Step at which minimum value starts to grow

    Returns
    -------------
    aged_val : float
        Returns max_val multiplied by modified sigmoid function

    Example
    ------------
    We want to set maximum value to 10, actual interval is 5, and interval when␣
 ↪the value is at its maximum
    is 10. The actual interval is in the middle of it's maximum so 50th␣
 ↪percentil will be returned.

    In:
```

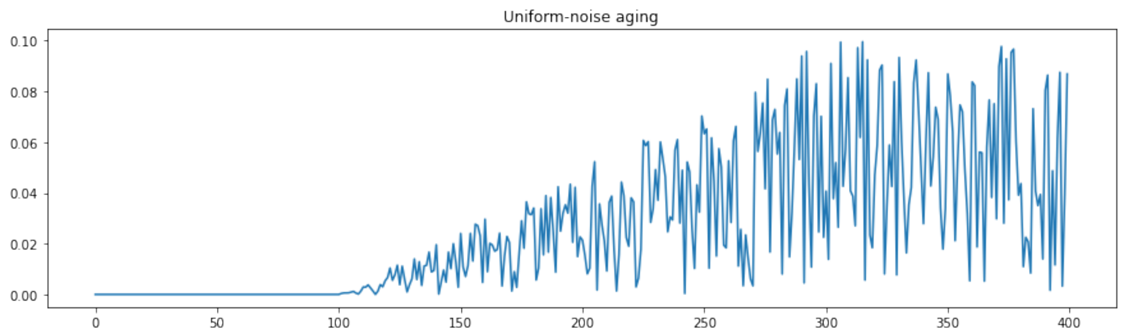
```
    >> sigmoid(max_val=10, i=5, max_i=10)

    Out:
    >> 5


    """

    if i < min_i:
        sigm = 0
    else:
        treshold = 12.3
        x = (i-min_i)/(max_i-min_i)*treshold*2-treshold
        sigm = 1/(1+np.exp(-x))

    return max_val*round(sigm, 5)


def uniformnoise_aging(max_val, i, max_i, min_i=0):
    """
    Aging strategy that reduces/increases the value based on uniform noise with␣
 ↪linear growth
    of standard deviation (std).


    Parameters
    -------------
    max_val : int, float
        ...

    i : int, float
        Actual step

    max_i : int, float
        Step at which maximum value is achieved

    min_i: int, float, Optional
        Step at which minimum value starts to grow

    Returns
    -------------
    aged_val : float
        Returns value generated by uniform noise


    """
    if i < min_i:
```

```
        gain = 0
    elif i > max_i:
        gain = 1
    else:
        m = 1/(max_i-min_i)
        b = -min_i*m
        gain = i*m+b

    mean = 0
    std = max_val/2
    return max_val*gain*np.random.rand()
```

[ ]:
```
length=400
a = np.zeros(length)

for step in range(length):
    a[step]=sigmoid_aging(1,step,400,100)

plt.figure(figsize=(15,4))
plt.title('Sigmoid aging')
plt.plot(a)
plt.show()

for step in range(length):
    a[step]=whitenoise_aging(.1,step,300,100)

plt.figure(figsize=(15,4))
plt.title('White-noise aging')
plt.plot(a)
plt.show()

for step in range(length):
    a[step]=uniformnoise_aging(.1,step,300,100)

plt.figure(figsize=(15,4))
plt.title('Uniform-noise aging')
plt.plot(a)
plt.show()
```







Now create function that will reduce parameters based on the strategy

[ ]:
```
def age_param(param, max_change, i, max_iter, min_iter=0, polarity='minus',␣
 ↪strategy='sigmoid'):
    """
    Funciton that simulates failure/aging of given parameter based on
    chosen strategy.

    Parameters
    ------------
```

```
    param : float
        Parameter / value to be changed

    max_change : float
        Maximum change of parameter, that will be achieved.
        For example: 0.1 = 10% of change will be achieved.

    i : int
        Actual iteration

    max_iter : int
        Number of iterations when maximum change is achieved.

    min_iter : int, Optional
        Number of iterations when change stars to appear.

    polarity : str, Optional
        Choose between addition or substraction as cause of aging.
        'plus' : addition is applied
        'minus': substraction is applied

    strategy : str, Optional
        Chosen strategy that will be used to age parameter.
        'sigmoid' : sigmoid_aging is called, modified sigmoid function

        (See more in the description of individual strategy functions)

    Returns
    ----------
    aged_param : float
        Inputed parameter decreased by chosen strategy

    """
    sign = -1 if polarity=='minus' else 1
    if strategy == 'sigmoid':
        aged_param = param*(1+sign*sigmoid_aging(max_change, i, max_iter,␣
↪min_iter))
    elif strategy == 'whitenoise':
        aged_param = param*(1+sign*whitenoise_aging(max_change, i, max_iter,␣
↪min_iter))
    elif strategy == 'uniform':
        aged_param = param*(1+sign*uniformnoise_aging(max_change, i, max_iter,␣
↪min_iter))

    return aged_param
```

And now is time to create a function that will generate the aged signal. First of all, let's define control strategy.

```
[ ]: np.linalg.eig(A)[0]
```

```
[ ]: array([-7.94220461e+00+47.75108239j, -7.94220461e+00-47.75108239j,
            -9.06267343e+00 +0.j        , -1.28853143e-04 +0.j        ,
            -1.75650398e-01 +0.j        , -2.84600791e-01+39.81348584j,
            -2.84600791e-01-39.81348584j])
```

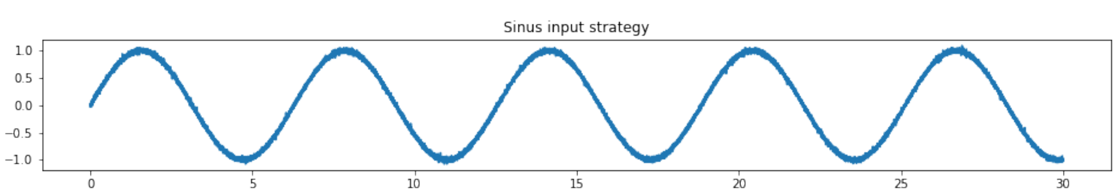
We can see that all eigenvalues of the A matrix are on the left side of s-plane, which means, the system is stable. For a stable system, we can use simple open-loop control. Which practically means we can simply determine input u without feedback.

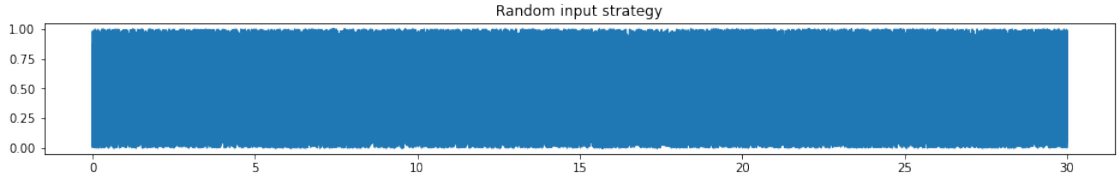
1. Low frequency sine-input for max duration (noi* - sin)
2. Random agresive input for max duration (noi* - gauss)
3. Every hour different weight of load (noi* - weight)
4. White noise input (noi* - whitenoise)
5. Sine input with slowly changable frequency for max duration (noi* - freq)
6. Every hour different weight of load and different input frequency (white, sine with changed freq, sine, etc) (noi* - all)

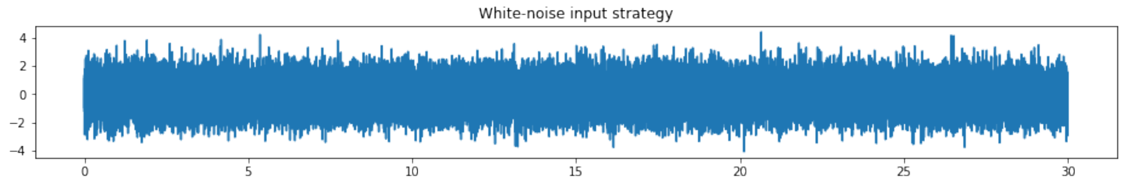
```
[ ]: def generate_inputs_strat(time):
         u_names = ['Sinus', 'Random', 'White-noise', 'Sinus with different␣
      ↪frequency']
         U = np.zeros((len(time), 4))
         U[:,0] = np.sin(time) + np.random.logistic(0, 0.01, size=len(time))
         U[:,1] = np.random.rand(len(time))
         U[:,2] = np.random.normal(0, 1, len(time))
         U[:,3] = np.cos(2*np.pi*time*0.4+2*np.sin(2*np.pi*0.1*time)) + np.random.
      ↪logistic(0, 0.005, size=len(time))
         return U, u_names
```

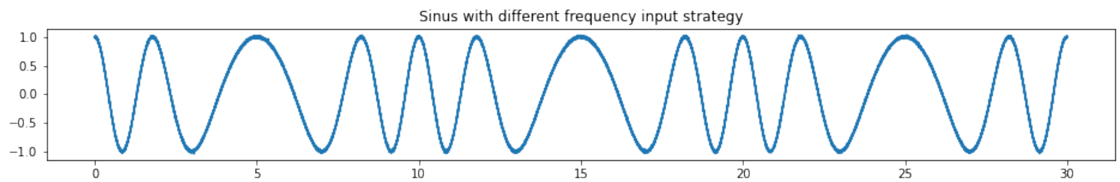
```
[ ]: time = np.arange(0, 30, 1/2000)
     U, nms = generate_inputs_strat(time)
```

```
[ ]: show = 100000
     for k in range(U.shape[1]):
         u = U[:, k]
         plt.figure(figsize=(16,2))
         plt.title(nms[k]+' input strategy')
         plt.plot(time[:show], u[:show])
         plt.show()
```









In order to meet the 2nd and 3rd requirements we specified at the beginning, we need to change parameters in the model. Library scipy signal StateSpace does not allow to change parameters after creating model nor during the simulation or to input matrices A,B,C,D with the time domain. One of the possibilities left is something I will refer to as micro-simulation. **Micro-simulation.** The scipy model will be created and simulated for a short period of time. After simulation State Space parameters A,B,C,D are changed based on the aging strategy and the new scipy model is created. The process repeats until required time. To the whole simulation process, I refer to it as a **makro-simulation**.

Note: Micro-simulation is used as an analogy to the cosimulations, where we distinguish micro- and makro- integration

[ ]:
```
def time2step(act_time, time_domain, steps, unit='s'):
    """
    Calculate step index that coresponds to the specific time in time domain.

    Note: This function works with steps, that are refer to batch.
    Note: This serves as auxilary function for signal simulation function.

    Parameters
    --------------
```

```
    act_time : float
        Time that we want to translate into step

    time_domain : array
        Array of time in seconds with i.e. np.arange(0,160,0.1)

    steps : int
        Number of steps that are utilize

    unit : str
        Unit of inputed act_time.


    """

    if unit == 's':
        pass
    elif unit == 'm':
        act_time*=60
    elif unit == 'h':
        act_time*=60**2
    else:
        raise ValueError(f"Unit is expected to be: 's', 'm', 'h'. Your input:␣
 ↪{unit}")

    T = time_domain[1]-time_domain[0]
    return act_time*steps/(len(time_domain)*T)


def step2timestep(act_step, time_domain, steps):
        """
        Calculate time that coresponds to the specific step in time domain.

        Inverse function to time2step with translation to timestamp.

        Parameters
        --------------
        act_step : float
            Step that we want to translate into time

        time_domain : array
            Array of time in seconds with i.e. np.arange(0,160,0.1)

        steps : int
            Number of steps that are utilize
```

```
        """

        init_timestamp = 1640991600.00
        T = time_domain[1]-time_domain[0]
        act_time = act_step*len(time_domain)*T/steps
        return init_timestamp+act_time

class Timer:
    def __init__(self):
        self.sta = 0
        self.sto = 0

    def start(self):
        self.sta = tm.perf_counter()

    def stop(self):
        self.sto = tm.perf_counter()
        print(f'Processed in {round(self.sta-self.sto,3)} seconds')
```

[ ]:
```
def simulator(u, time, coeficients, change_strat, change_period=3,␣
↪extra_step=None):
    """
    State Space model simulator based on scipy.signal that allows change of␣
↪parameters during simulation.

    Parameters
    -------------
    u : array-like
        Generated control signal

    time : array-like
        Time domain that coresponds to the control signal u

    coeficients : dictionary
        Dictionary of coeficients that is used for generate
        Space State matricies through the generate_ABCD function
        and to simulate aging.

    change_strat : 2D list
        List that contains parameters of coef. change strategy.
        Every row in list contains:
            parameter_name - name of the parameter must exist inside␣
↪coeficients dictionary
            maximum_change - maximu % change of the parameter
            peak_time - time point when the change will achieve its maximum
            start_time - time point when the change takes effect
            time_unit - units of peak_time and start_time 's', 'm', 'h'
```

```
            polarity - decides if parameter value will decrease or increase␣
↪'plus'/'minus'
            strategy name - name of the strategy that must exist in the␣
↪age_param function

        Example of change_strat:
            'sigmoid' strategy that will decrease parameter 'k1' after reaching␣
↪simulating
            time 100s and it will grow until reaching maximum of .9% at 120s.

            [
            ['k2', .9, 120, 100, 's', 'minus', 'sigmoid']
            ]

    change_period : int, Optional
        How often are parameteres changed during the simulation.
        Minimum period of change is once every three time steps␣
↪(change_period=3)

    extra_step : int, Optional

    Returns
    ----------------
    y : np.array
        Value of the function

    t : np.array
        Time domain of running simulation
        Careful: Due to change_period outputed time 't' might not
                 migth not be the same as inputed time 'time'

    timesteps : np.array
        Time domain converted into timesteps

    Requirements
    ----------------
    libraries
        from scipy import io
        from scipy import signal

    functions
        time2step : for translation between time and step
        age_param : to change parameters based on specific strategy
            + additional strategy functions
        generate_ABCD : creates State Spaces matricies for fixed model
    """
```

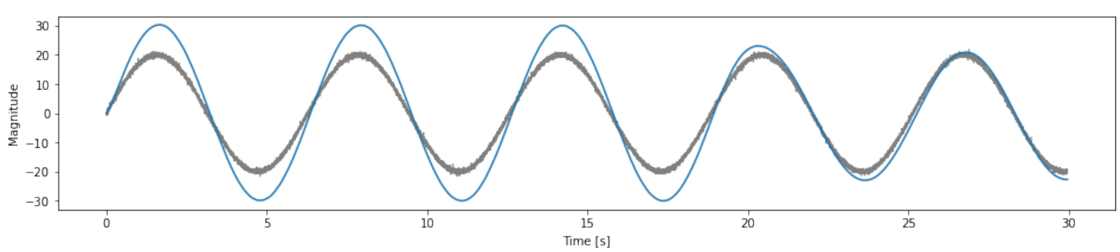
```
    #Time&step description
    number_of_steps = int(len(time)/change_period)
    real_signal_len = number_of_steps*change_period
    T = (time[1]-time[0])

    #Initialization
    changed_coef = coeficients.copy()
    A, B, C, D = generate_ABCD(coeficients)
    y = np.zeros((real_signal_len, len(C)))
    x_sim = [0]*len(A)

    #Macro-Simulation
    for step in range(number_of_steps):
        beg = step*change_period
        end = (step+1)*change_period+1 #+1 is needed to slight overlap

        #Batch
        u_batched = u[beg:end]
        period = time[beg:end]-time[beg:end][0]

        #Aging strategy
        iteration = step if extra_step == None else number_of_steps*extra_step␣
↪+ step

        for cmd in change_strat:
            max_step = time2step(cmd[2], time, number_of_steps, cmd[4])
            min_step = time2step(cmd[3], time, number_of_steps, cmd[4])

            changed_coef[cmd[0]] = age_param(param=coef[cmd[0]],␣
↪max_change=cmd[1],
                                             i=iteration, max_iter=max_step,␣
↪min_iter=min_step,
                                             polarity=cmd[5], strategy=cmd[6])

        #Micro-Simulation
        A, B, C, D = generate_ABCD(changed_coef)
        sys = signal.StateSpace(A,B,C,D)
        _, y_sim, x_sim = signal.lsim(sys, u_batched, period, X0=x_sim[-1])
        if len(y[beg:end]) < len(y_sim):
            y[beg:end] = y_sim[:-1]
        else:
            y[beg:end] = y_sim

    timesteps = np.array([1640991600.00 + time_s for time_s in np.arange(0,␣
↪real_signal_len*T, T)])
    return y, np.arange(0, real_signal_len*T, T), timesteps
```

The simulator is created and now it's time to test it out. Expectations are that the simulation will last much longer due to repeated StateSpace model generation and micro-simulation. In order to represent simulator usage, I will describe the meaning of the 'strategy' list, because it might be the most confusing part. I want to simulate the increasing of coefficient 'k2' which would mean that the counter-electromotive force (back EMF) of the model will increase due to changes in the electrical circuit or the surrounding heat. The coefficient will change up to 30% (0.3) and the change will take place at 25 sec and it will reach its maximum at 175 sec of simulation. The aging strategy that we will use is 'sigmoid'.

```
[ ]: timer = Timer()
     timer.start()
     in_u = U[:,0]

     strategy = [['R',500,25,15,'s','plus','sigmoid']]
     y, t, _ = simulator(in_u, time, coef, strategy)

     timer.stop()
```

```
Processed in -7.423 seconds
```

```
[ ]: #Print the stuff
     shift = 100
     magnifier = 20
     plt.figure(figsize=(16, 3))
     plt.plot(time[:-shift], magnifier*in_u[:-shift], linewidth=1, color='grey',␣
      ↪linestyle='--')
     plt.plot(t[:-shift], y[shift:,0])
     plt.ylabel('Magnitude')
     plt.xlabel('Time [s]')
     plt.show()
```



The test showed that simulation is working as it was required and it also gives inside in simulation duration. 200 seconds of simulation takes 0.26 to 0.27 seconds of real-time to process. The initial goal is to create 5 different inputs and 5 different failures and 10 parameters = 250 combinations and simulate it for 100 hours with 1-hour batches. Every start of the simulator takes 0.1 seconds. To simulate 25 combs * 100 hours = 2500 hours will take...

```
[ ]: combs = 250
     hours = 100
     to_simulate = (combs*hours)*60**2
     batch = 0.1
     simulated = 200
     simul_time = 0.263

     needed_time = to_simulate/simulated*simul_time+combs*hours*batch
     print(f'It will take {round(needed_time/60/60,1)} hours to simulate')
```

```
It will take 33.6 hours to simulate
```

In order to lower the time it takes to create simulated data, we can reduce the duration of the simulation and compensate it with a different aging rate. Or we could create samples for a duration of 30-60 sec every X minutes/hours. The main goal of the simulation duration is to provide enough data to observe changes in the signal. ### Conclusion Terms of signal generator #1 #3 and #4 are not met. The real sampling frequency is expected to be at 100 kHz which would tremendously elongate the signal generating process and linear neuron learning. This sampling frequency could be generated for practice reasons in the future after calculations of the real requirements of signal time that is needed to predict aging. For further simulations, only 30s samples with a 2 kHz frequency every 10 minutes will be generated (corresponding to the real measurements from another thesis). In order to adapt to term #4, the aging will start in the 30s and reach its maximum after 3 hours.

1.5 Data generation

Generating and saving required data. Before we start generating every of 250 combinations, the first so-called Minimum Value Product (MVP) must be created. This smaller set of signals is going to serve as a probe sample on which parameters of further signal generation will be adjusted. MVP is contained of following signals: 1. cc_sin_sigmoid_20 1. cc_sin_whitenoise_20 1. L_freq_sigmoid_20 2. k2_freq_sigmoid_20 3. cc_freq_sigmoid_20 4. kc_freq_sigmoid_20 5. cs_freq_sigmoid_20 6. ks_freq_sigmoid_20

1.5.1 Generator

First, let's create a function that could generate realistic signal sampling for a specified duration. We need to be capable to create 30s sampling every 10th minute for 3 hours.

```
[ ]: def higher_simulator(simulation_duration, step_size, sample_size, freq,␣
      ↪coeficients, change_strat, input_index):
         """
         Dedicated simulator that uses simulator() function in loops.
         This function allows to create sampling of a specified duration, every Xth␣
      ↪minute
         for a specifed simulation duration.

         Parameters
         --------------
         simulation_duration : float
```

```
        Duration of whole simulation in seconds

    step_size : float
        How often is sample taken in seconds

    sample_size : float
        Duration of one sample in seconds

    freq : float
        Sampling frequency in kHz

    coeficients : dictionary
        Dictionary of coeficients that is used for generate
        Space State matricies through the generate_ABCD function
        and to simulate aging.

    change_strat : 2D list
        List that contains parameters of coef. change strategy.
        Every row in list contains:
            parameter_name - name of the parameter must exist inside␣
↪coeficients dictionary
            maximum_change - maximu % change of the parameter
            peak_time - time point when the change will achieve its maximum
            start_time - time point when the change takes effect
            time_unit - units of peak_time and start_time 's', 'm', 'h'
            polarity - decides if parameter value will decrease or increase␣
↪'plus'/'minus'
            strategy name - name of the strategy that must exist in the␣
↪age_param function

        Example of change_strat:
            'sigmoid' strategy that will decrease parameter 'k1' after reaching␣
↪simulating
            time 100s and it will grow until reaching maximum of .9% at 120s.

            [
            ['k2', .9, 120, 100, 's', 'minus', 'sigmoid']
            ]

    input_index : int
        Index of input created by generate_inputs_strat()


    Requirements
    ---------------
    functions
```

```
        generate_inputs_strat() : to generate strategies of input with slight␣
↪offset
        simulator() : main simulation function


    """

    #num_of_steps, one_step,  sim_duration,one_step
    time = np.arange(0,sample_size,1/(freq*1e3))
    time_len = len(time)

    num_of_steps = int(simulation_duration/step_size)

    # Input initialisation
    y = np.zeros((time_len*num_of_steps,2))
    t = np.zeros(time_len*num_of_steps)
    u = np.zeros(time_len*num_of_steps)

    for stp in range(num_of_steps):
        srt = stp*time_len
        end = (stp+1)*time_len
        # Input initialization (added offset randomness)
        rnd_factor = np.random.randint(0, 10)
        U, _ = generate_inputs_strat(np.arange(rnd_factor,␣
↪rnd_factor+sample_size, 1/(freq*1e3)))
        u[srt:end] = U[:,input_index]

        # Simulation process
        spec_stp = stp*step_size/simulation_duration
        print(spec_stp)
        y[srt:end,:], _, t[srt:end] = simulator(u[srt:end], time, coeficients,␣
↪change_strat=change_strat, extra_step=spec_stp)

        clear_output(wait=True)
        print(f'Ready {stp+1}-th simulation of total {num_of_steps}␣
↪simulations')

    clear_output(wait=True)
    print('Simulation completed')
    return t, y, u
```

```
[ ]: #Aging parameters & Aging strategy
aging_start = 30
aging_end = 150

strategies = [['cc',0.2,aging_end,aging_start,'s','plus','sigmoid'],
              ['cc',0.2,aging_end,aging_start,'s','plus','whitenoise'],
```

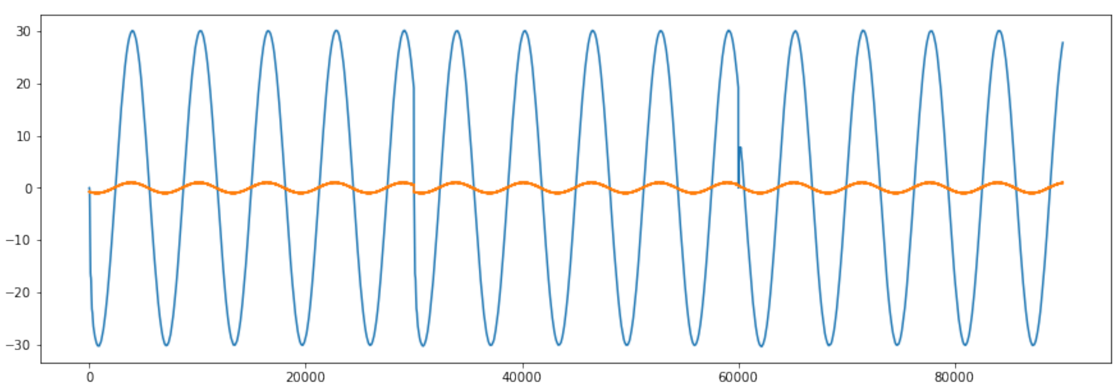
```
              ['L',0.2,aging_end,aging_start,'s','plus','sigmoid'],
              ['k2',0.2,aging_end,aging_start,'s','plus','sigmoid'],
              ['cc',0.2,aging_end,aging_start,'s','plus','sigmoid'],
              ['kc',0.2,aging_end,aging_start,'s','plus','sigmoid'],
              ['cs',0.2,aging_end,aging_start,'s','plus','sigmoid'],
              ['ks',0.2,aging_end,aging_start,'s','plus','sigmoid']]


groups = ['cc_sin_sigmoid_20',
          'cc_sin_whitenoise_20',
          'L_freq_sigmoid_20',
          'k2_freq_sigmoid_20',
          'cc_freq_sigmoid_20',
          'kc_freq_sigmoid_20',
          'cs_freq_sigmoid_20',
          'ks_freq_sigmoid_20']

# 30 sec, every 10 minutes, for 3 hours
# Bubble Trouble: Parametry se ovlivňují pouze s pohledem na 30 vteřin ne na␣
 ↪generaci
```

Now let's test it out.

[ ]:
```
timer = Timer()
timer.start()
t, y, u = higher_simulator(200, 60, 30, 1, coef, [strategies[0]], 0)
timer.stop()
```

```
Simulation completed
Processed in -11.695 seconds
```

[ ]:
```
plt.figure(figsize=(15,5))
plt.plot(y[:,0])
plt.plot(u[:])
plt.show()
```



Disruptions in the signal are created purposely. It simulates time samples at different times. So what we see on the plot is not one simultaneous signal, it is 6 samples of 30 s duration.

1.5.2 Data saving

The goal is to create a file with the name samples_package_0.h5 that will contain mentioned signals and those are made of matrices of timestamp, inputs and outputs. Inputs are the mass of the load and signal, outputs are position and velocity.

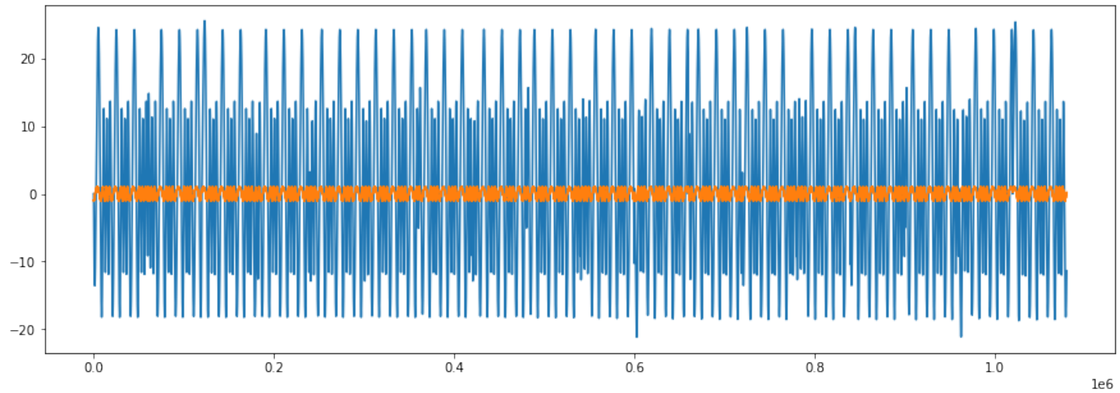
[ ]:
```
def save_it(data, file_name, group_name):
    with h5py.File(file_name + ".h5", 'w') as file:
        group = file.create_group(group_name)
        for col in data:
            dataset = group.create_dataset(col, shape=data[col].shape,␣
↪maxshape=(None,))
            dataset[:] = data[col].astype(float)
    print(f'Data {group_name} are saved.')
```

1.5.3 Generate and save

Following datasets are being generated and saved. 1. cc_sin_sigmoid_20 1. cc_sin_whitenoise_20 1. L_freq_sigmoid_20 2. k2_freq_sigmoid_20 3. cc_freq_sigmoid_20 4. kc_freq_sigmoid_20 5. cs_freq_sigmoid_20 6. ks_freq_sigmoid_20

[ ]:
```
file_name = "samples_package_0"

strateg_id = 7
input_id = -1

simulation_time = 3*60**2
sampling_period = 10*60
sampling_duration = 30
frequency = 2
group_name = groups[strateg_id]

# Create
t, y, u = higher_simulator(simulation_time, sampling_period, sampling_duration,␣
↪frequency, coef,
                           [strategies[strateg_id]], input_id)

#Visualise
plt.figure(figsize=(15,5))
plt.plot(y[:,0])
plt.plot(u[:])
plt.show()
```

```
Simulation completed
```




```
[ ]: #Create dataframe
     data = pd.DataFrame({'timestamp':t, 'signal':u, 'weight':burden_weight,␣
      ↪'position': y[:,0], 'velocity': y[:,1]})
     data
```

```
[ ]:             timestamp    signal  weight   position   velocity
     0        1.640992e+09 -0.998221      10   0.000000   0.000000
     1        1.640992e+09 -0.973596      10  -0.000002  -0.013148
     2        1.640992e+09 -0.991718      10  -0.000017  -0.052184
     3        1.640992e+09 -1.005101      10  -0.000059  -0.117131
     4        1.640992e+09 -1.002746      10  -0.000139  -0.208028
     …                 …         …       …          …          …
     1079995  1.640992e+09  0.230857      10 -11.415573  25.062190
     1079996  1.640992e+09  0.226976      10 -11.403038  25.076487
     1079997  1.640992e+09  0.225463      10 -11.390497  25.090761
     1079998  1.640992e+09  0.217203      10 -11.377948  25.104915
     1079999  1.640992e+09  0.232354      10 -11.365392  25.118845

     [1080000 rows x 5 columns]
```

```
[ ]: #Save
     save_it(data, file_name, group_name)
```

```
Data ks_freq_sigmoid_20 are saved.
```

1.6 Overall conclusion

This work brings to the main thesis data generator by State Space model that is based on a physical model. The generator is equipped with the option to change inputs and the option to create complex aging strategies of one or more parameters. For further testing purposes, 8 signals are generated and saved as large data format .h5 for simpler loading of longer signals in the future. With this sentence, all of the requirements are completed.