**Óbuda University**

# FPTControl Toolbox Guide

The example driven approach

Dr. Krisztián Kósi

Institute of Applied Mathematics

Budapest, 27th April 2016

# CONTENTS

# ACKNOWLEDGEMENT

I would like to thank all of the support to Prof. József Kázmér Tar.

I also want to express my thank to Óbuda University

Last, but not least I want say thank you to that awesome people, who wrote Julia language, it is an amazing language.

# CHAPTER 1 INTRODUCTION

My dear friend,

The story of Robust Fixed Point Transformation (RFPT) started in 2009 [1]. This method using iterative techniques, which widely used for finding the solutions to typically nonlinear problems.

This Toolbox helps you to create your own simulation. First I will show the basics of RFPT method. Then I will explain the functions, which included in the toolbox. Finally, I will guide you through the process to create a Simulations.

CHAPTER 2

# PRINCIPLES OF ROBUST FIXED POINT TRANSFORMATION BASED METHODS

First I want to show the control cycle for Fixed Point Transformation (Fig. 2.1).

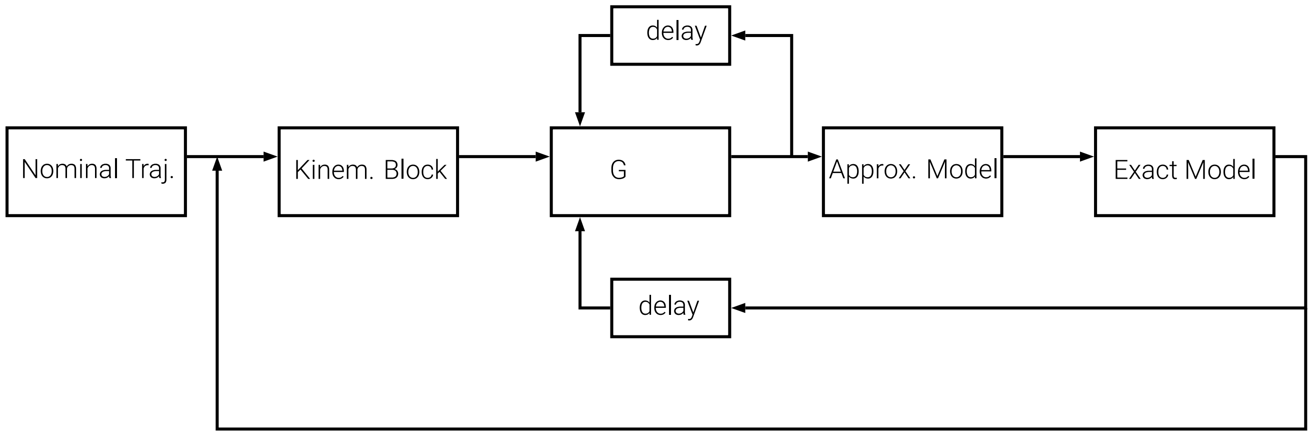

Figure 2.1: The control cycle for RFPT

In the following sections I will explain what is inside the blocks

## 2.1 Nominal Trajectory

It is the path, which the System needs to follow. In RFPT it is needs to be a continuous path, step functions are not recommended.

You have to create the derivatives of the path. If it is a math function, like $sin()$ which I usually use, derivation in closed form is easy. You can also use some Symbolic Math Programs, like the awesome SymPy package, or use some numerical technique.

## 2.2 Kinematic Block

You can use PID or PD type kinematic description, to produce the desired signal. This Block is not limited to PID, or PD, you can use any description.

Definition of the Error:

$$h \overset{def}{=} q^N - q \tag{2.1}$$

The Derivation of the error from the definition. For example the first time derivative:

$$\dot{h} = \dot{q}^N - \dot{q} \tag{2.2}$$

Error Integral:

$$h\_integral \overset{def}{=} \int_{t_0}^{t} q^N(\xi) - q(\xi) d\xi \tag{2.3}$$

PID:

$$(\frac{d}{dt} + \Lambda)^{n+1} h\_integral \tag{2.4}$$

PD:

$$(\frac{d}{dt} + \Lambda)^{n} h \tag{2.5}$$

where $\Lambda > 0$ is the tuning parameter, $n$ is the order of the system

The practical usage will be explained in section 3.1.

## 2.3 G

This is the Deformation Block. The G function is different in SISO, and MIMO case.
SISO case:

$$G \overset{def}{=} (r + K)[1 + B\sigma(A(f(r) - r^d))] - K \tag{2.6}$$

where $K, B \overset{def}{=} \pm 1, A$ are control parameter and $\sigma()$ is a sigmoid function. You can use any sigmoid type function, like $tanh()$. $r^d$ is the desired signal, came from the Kinematic Block). $r$ is the past input, came from the output of G, through delay block (previous cycle). $f(r)$ is the past response, the System's answer from the previous cycle, came from the Exact Model, trough a delay block.
MIMO case:

$$\vec{h} \overset{def}{=} \vec{f}(\vec{x}_n) - \vec{x}^d \tag{2.7}$$

$$\vec{e} \overset{def}{=} \frac{\vec{h}}{||\vec{h}||} \tag{2.8}$$

$$\tilde{B} = \sigma(A||\vec{h}||) \tag{2.9}$$

$$\vec{x}_{n+1} \overset{def}{=} (1 + \tilde{B})\vec{x}_n + \tilde{B}K\vec{e} \tag{2.10}$$

## 2.4 Approximate Model

This is some kind of "inverse" model. The only different is: The model don't have to be precise. There are not exact rule, which parts are needed yet, but you can experiments with that. It is really robust for that kind of approximations. The control parameters can be approximated too.

## 2.5 Exact Model

This is the system. It needed for simulations. In real world it is a physical system (for example a robot).

# CHAPTER 3 EXPLAINING THE FUNCTIONS IN FPTCONTROL TOOLBOX

## 3.1 KB

This is the kinematic block.

```
KB(n, lambda, errors, nominal)
```

where $n$ is the proper power of $(\frac{d}{dt}+\Lambda)$, $errors$ is the error vector, $nominal$ is the nominal trajectory in the proper order.

To understand it let me show you an example. Let's say, we have a second order system, and we choose PID. (we will use it when we building a simulation in the next chapter)

So we have to compute $(\frac{d}{dt}+\Lambda)^3 h\_int0$ (h_int is the error integral). We will use the binomial expansion:

$$(a+b)^n = \sum_{l=0}^{n} \frac{n!}{l!(n-l)!} a^l b^{n-l} \tag{3.1}$$

I have to note, that this $n$ in the binomial expression is different than the $n$ in PID.
In our case $a$ will be $\frac{d}{dt}$, and $b$ will be $\Lambda$. We will get the following expression:

$$\Lambda^3 h\_int + 3\Lambda^2 h + 3\Lambda \dot{h} + \ddot{h} \tag{3.2}$$

We know, the $\ddot{h} = \ddot{q}^N - \ddot{q}^{Des}$. If we plug that formula in, and arrange the equation for $\ddot{q}^{Des}$, we will get the following equation, which will be our Kinematic Block:

$$\ddot{q}^{Des} = \Lambda^3 h\_int + 3\Lambda^2 h + 3\Lambda \dot{h} + \ddot{q}^N \tag{3.3}$$

If we see in Julia code, the "for" loop produce the expression before the nominal part, after the loop we simply add the nominal part to the expression.

```
content.function KB(n, lambda, errors, nominal)
out=0
for l=0:n-1
out=out+(factorial(n)/(factorial(l)*factorial(n-l))*errors[l+1]*lambda^(n-l))
end
out=out+nominal
return out
end
```

## 3.2 Integ

This is just a simple Euler integral. The rectangle method. It is fast, and many case, it is enough. Of course you can use any numerical integrator.

```
function Integ(Integral ,step ,input)
out=Integral+step*input
return out
end
```

## 3.3 G function

The functions was explained in section 2.3, nothing special.

```
function G_SISO(past_input ,past_response ,desired ,Kc,Bc,Ac)
out=(Kc+past_input)*(1 + Bc * tanh(Ac * (past_response - desired))) - Kc
return out
end

function G_MIMO(past_input ,past_response ,desired ,error_limit ,Kc,Bc,Ac) # Inputs and Ou
#  need control parameters  K B A
error_norm=vecnorm(past_response - desired ,2)
# If the norm of the error is greater then the limnit compute the deformation
# (it is not near the Fixed Point)
if error_norm>error_limit
e_direction=(past_response-desired)/error_norm
B_factor= Bc* sigmoid(Ac *error_norm)
out=(1+B_factor)*past_input +B_factor*Kc*e_direction
else
out=past_input # Almost in the Fixed Point
end
return out
end
```

# CHAPTER 4 THE SIMULATION

I will show the process, how to build your simulation. We will use the Duffing Oscillator (also called Harmonic Oscillator) as an example.

The model of the Duffing Oscillator is (the Exact Model block):

$$\ddot{q} = \alpha q - \delta\dot{q} - \beta q^3 + u(t) \tag{4.1}$$

where $\alpha, \delta, \beta$ are the model parameters and $u(t)$ is our control signal.

We can create the "inverse" model from it, just arrange the equation to $u(t)$:

$$u(t) = \ddot{q} - \alpha q + \delta\dot{q} + \beta q^3 \tag{4.2}$$

You can use that, or you can Approximate it. As I said before, there are no rule, how to approximate, and what will work or not. In that case you can use the following approximation:

$$u(t) = \ddot{q} \tag{4.3}$$

(Note: In programming you can place a # symbol after the acceleration, so you can easily test, what is the changes.)

In section 3.1, discussed the method to create the Kinematic Block. Because it is an $2^{nd}$ order system, we can use that kinematic block, or we can use the $KB()$ function. All another things will be discussed in the "comments" of the code.
If you did not install FPTControl toolbox befor use Pkg.add() in Julia

```
Pkg.add("FPTControl")
```

First we have to load the FPTControl toolbox into Julia.

```
using FPTControl
```

Than we need to set up some variables For the simulations.

```
Adaptive=1
LONG=Int(1e4)
δt=1e-3
```

$Adaptive = 1$ means, that we will "switch on" Fixed Point Transformation, if this value set to $0$, it will be just a simple PID controller. $LONG$ is the simulation length, measured in cycles. It has to be an integer, because "for" loop and indexing array needs integer value. $\delta t$ is the cycle time. If you create a real world system, you have to taking account the speed of the sensors.

```
#PID tune parameter
Λ=3

#Nominal Trajectory
ω=5
Ampl=0.4

#Controller Parameters
K=1e5
B=-1
A=1e-5

# Exact model parameters
α_e=1
δ_e=0.2
β_e=1

# Approx model parameters
α_a=0.8
δ_a=0.1
β_a=0.9
```

We will plan a simple $A\ \ sin(\omega t)$ nominal trajectory.
You can play with the "Approx. model parameters" too.
Creating two functions for models, which described before.

```
function exact(q,q_p,u)
out=α_e*q+δ_e*q_p+β_e*q^3+u
return out #q_pp
end

function approx(q,q_p,q_pp)
out=q_pp-α_a*q-δ_a*q_p-β_a*q^3
return out #u
end
```

Have to create some Arrays to store variables for plotting, and have to take some initial conditions.

```
qN=zeros(LONG)
qN_p=zeros(LONG)
qN_pp=zeros(LONG)

qDes_pp=zeros(LONG)

qDef_pp=zeros(LONG)
u=zeros(LONG)

q=zeros(LONG)
q_p=zeros(LONG)
q_pp=zeros(LONG)
```

```
t=zeros(LONG)

h_int=0
past_input=0
past_response=0
```

We can start our simulation. We have to run it until $LONG-1$.
We storing the time into Array "t".

```
for i=1:LONG-1
#store time
t[i]=δt*i
```

Create a simple nominal trajectory, and the time derivatives in closed form.

```
#Nominal Trajectory
qN[i]=Ampl*sin(ω*t[i])
qN_p[i]=Ampl*ω*cos(ω*t[i])
qN_pp[i]=-Ampl*ω^2*sin(ω*t[i])
```

Build the error and the time derivative of the error. Putting together the error vector, and call KB() to produce the Kinematic Block.

```
h=qN[i]-q[i]
h_p=qN_p[i]-q_p[i]
errors=[h_int,h,h_p]
qDes_pp[i]=KB(3,Λ,errors,qN_pp[i])
```

The G function, we usually wait some cycle before turn it on.

```
if Adaptive==1 && i>10
qDef_pp[i]=G_SISO(past_input,past_response,qDes_pp[i],K,B,A)
else
qDef_pp[i]=qDes_pp[i]
end #if
past_input=qDef_pp[i]
```

Produce the control signal:

```
u[i]=approx(q[i],q_p[i],qDef_pp[i])
```

Get the system response:

```
q_pp[i]=exact(q[i],q_p[i],u[i])
past_response=q_pp[i]
```

Create the integrals for the next cycle:

```
h_int=Integ(h_int,δt,h)
q_p[i+1]=Integ(q_p[i],δt,q_pp[i])
q[i+1]=Integ(q[i],δt,q_p[i])
end #for
```

Compute the last step:

```
t[LONG]=δt*LONG
qN[LONG]=Ampl*sin(ω*t[LONG])
qN_p[LONG]=Ampl*ω*cos(ω*t[LONG])
```

```
qN_pp[LONG]=−Ampl*ω^2*sin(ω*t[LONG])
h=qN[LONG]−q[LONG]
h_p=qN_p[LONG]−q_p[LONG]
errors=[error_int,h,h_p]
qDes_pp[LONG]=KB(3,Λ,errors,qN_pp[LONG])
if Adaptive==1
qDef_pp[LONG]=G_SISO(past_input,past_response,qDes_pp[LONG],K,B,A)
else
qDef_pp[LONG]=qDes_pp[LONG]
end #if
u[LONG]=approx(q[LONG],q_p[LONG],qDef_pp[LONG])
q_pp[LONG]=exact(q[LONG],q_p[LONG],u[LONG])
```

Finally, we can create figures. I usually use PyPlot, but many another great packages available.

```
using PyPlot
figure(1)
grid("on")
title("Nominal and realized traj. Nominal: red Realized: blue")
xlabel("Time")
ylabel("q")
plot(t,qN,color="red")
plot(t,q,color="blue","r−−")

figure(2)
grid("on")
title("the control signal")
plot(t,u,color="orange")

figure(3)
grid("on")
title("Tracking Error")
plot(t,qN−q,color="blue")

figure(4)
grid("on")
title("Trajectory Phase Space")
plot(qN,qN_p,color="red")
plot(q,q_p,color="blue","r−−")

figure(5)
grid("on")
title("Accelerations: Nominal: red, Desired: orange, Realized: Blue")
plot(t,qN_pp,color="red")
plot(t,qDes_pp,color="orange","r−.")
plot(t,q_pp,color="blue","r−−")
```

# BIBLIOGRAPHY

[1] J.K. Tar, J.F. Bitó, L. Nádai, and J.A. Tenreiro Machado. Robust Fixed Point Transformations in adaptive control using local basin of attraction. *Acta Polytechnica Hungarica*, 6(1):21–37, 2009.