

Appendix

Default Controller

```
import luyaODE as luya
reload(luya)

## Initialise our Physics World
def init(g):
    g.w = luya.World(ground=True) # By default there is no ground plane.

## Step through our physics world
def run(g):
    g.w.step(0.05)

## End function
def end(g):
    pass
```

Fuzzy Chain Controller

```
#General Modules
import luyaUtils as lu
from mayaUtils import *
import luyaODE as luya ; reload(luya)
from math import pi,degrees,radians
from maya.api import MVector,MQuaternion
from maya.mel import *

## Our Fuzzy Logic Modules
import pyFuzzy; reload(pyFuzzy)
from pyFuzzy import *
from fuzzyRules import interpret

## Given a Current Rotation and and Reference position Compute an Angular Velocity.
class jointTracker:
    def __init__( self, strength=180 ):
        #Define the Angle Input Variable.
        scale = 90
        u = Stepper(-scale*3, scale*3,[ -3*scale, -scale, scale, 3*scale ])
        self.Angle = InputVariable (u.lower,u.upper,(
            ("Low" , Linear(u(0),u(1)), Linear(u(1),u(2))),
            ("High", Linear(u(1),u(2)), Linear(u(2),u(3)))
        ))

        #Define the Angular Speed Output Value
        u = Stepper(-strength*3, strength*3,[ -3*strength, -strength, strength, 3*strength ])
        self.AngularSpeed = OutputVariable (u.lower,u.upper,(
            ("Low" , Linear(u(0),u(1)), Linear(u(1),u(2))),
            ("High", Linear(u(1),u(2)), Linear(u(2),u(3)))
        ))

        #Define and interpret our ruleset
        self.ruleSet = interpret(\
        """
            if Angle is High : AngularSpeed = High
            if Angle is Low : AngularSpeed = Low
        """)

    ## Do the computation
    def __call__( self, Current, Reference=MVector(0,1,0)):
        Angle = self.Angle ; AngularSpeed = self.AngularSpeed

        #Normalize the two input vectors"""
        Reference = Reference.normal()
        Current = Current.normal()

        #Compute the rotation axis.
        #The rotation axis flips when Current changes which side of Reference it is on.
        #This is what we use to determin the Direction part of the Angular Velocity.
        Axis = ( Reference ^ Current ).normal()

        #Work out the current Error value.
        #This is the angle between Current and Reference
        Error = degrees( Current.angle( Reference ) )

        #Fuzzify the Error value
        Angle.fuzz(Error)
```

```

#Prepare the output variable for the computation of the Fuzzy Ruleset.
#Any accumulated membership values are cleared out.
AngularSpeed.begin()

#Execute the fuzzy rules.
#This computes an accumulative membership value for each set in the output variable AngularSpeed.
exec(self.ruleSet)

#Defuzz the Angular velocity.
#Get a "crisp" value from the AngularSpeed Linguistic Variable.
Solution = AngularSpeed.defuzz()

#Return thr angular velocity as a Quaternion.
return MQuaternion( radians( Solution ),Axis)

```

```

N= None
halfPi = pi / 2

```

```

## Initialise our behaviour
def init( g ):
    #Create an instance of our physics class
    g.w = luya.World( ground = True )
    #Create an instance of our jointTracker class
    g.w.tracker = Tracker.jointTracker()

## Run our Fuzzy Feedback Controller
def run( g ):
    #Loop through each joint in the chain
    for name in g.w.joints.keys():
        #Retrieve the joint
        joint = g.w.joints[name]
        #Retrieve the motor for the joint
        motor = joint.motor
        #Retrieve the current direction of the joint
        Current = Translate(joint.child) - Translate(name)
        #Use our fuzzy joint tracker class to compute an Angular Velocity
        Euler = g.w.tracker( Current ).asEulerRotation()

        #Multiply this velocity by a Constant
        kP = 2
        motor.setParam( luya.ParamVel, kP*Euler.x )
        motor.setParam( luya.ParamVel2, kP*Euler.y )
        motor.setParam( luya.ParamVel3, kP*Euler.z )

    #Step the simulation
    g.w.step(0.05)

def end( g ):
    pass

```

Fuzzy Inverse Pendulum Controller

[illegible]

```

### Set up our physics world
g.w = luya.World( ground=True )

g.lastError = 0 ## Record the last error so that we can compute the angular velocity

## Run fuzzy logic feedback controller
def run( g ):
    ## Housekeeping, Make local Declarations of our Linguistic variables for neatness
    Torque = g.Torque
    Angle = g.Angle
    AngularVelocity = g.AngularVelocity

    ## Grab transform information and turn it into something usable
    vThi = (Translate("CurrentTip") - Translate("CurrentBase")).normal()
    yAxis = (Translate("ReferenceTip") - Translate("ReferenceBase")).normal()
    zAxis = ( yAxis ^ MVector(0,1,0) ).normal()
    xAxis = ( zAxis ^ yAxis).normal()

    ## Work out the current error values
    Error = 90 - degrees( vThi.angle(xAxis) )
    ErrorChange = Error - g.lastError

    ## Fuzzify the error values
    Angle.fuzz(Error)
    AngularVelocity.fuzz(ErrorChange)

    ## Clean the output Variable
    Torque.begin()
    ## Evaluate the fuzzy rules
    exec(g.ruleSet)

    ## Defuzz the torque
    torque = Torque.defuzz()

    ## Convert from a quaternion to an Euler rotation. We lose information here.
    lEuler = MQuaternion(radians( torque ),zAxis).asEulerRotation()
    ## Because of the loss of information it is important to magnify the rotation by a P constant
    kP = -1

    ## Declare a local copy of motor for clarity
    motor = g.w.dynamics.CurrentBase.motor
    motor.setParam( luya.ParamVel, lEuler.x*kP )
    motor.setParam( luya.ParamVel2, lEuler.y*kP )
    motor.setParam( luya.ParamVel3, lEuler.z*kP )

    ## Store the last error as a global
    g.lastError = Error
    g.w.step(0.1)

def end( g ):
    pass

```

PID Chain Controller

#Modules needed by the current module. These can be thought of as includes and libraries in one thing.

```
import luyaODE as luya ; reload(luya)
```

```
from maya.mel import *
```

```
from mayaUtils import *
```

```
from luyaUtils import *
```

```
class jointPID:
```

```
    def __init__( self, w, joint, kP, kI, kD ):
```

```
        self.kP = kP
```

```
        self.kI = kI
```

```
        self.kD = kD
```

```
        self.w = w
```

```
        self.lastErrorX = 0
```

```
        self.lastErrorY = 0
```

```
        self.lastErrorZ = 0
```

```
        self.joint = joint
```

```
        #del joint.motor
```

```
    def __call__(self):
```

```
        joint = self.joint
```

```
        #Retrieve the Current Direction of the Inverse Pendulum.
```

```
        Current = Translate( joint.child ) - Translate( joint.name )
```

```
        #Retrieve the Direction of where we want to be.
```

```
        Reference = MVector( 0, 1, 0 )
```

```
        # The Axis is the cross product of the Reference and Current vectors.
```

```
        # when Current moves to the other side of Reference the axis will flip,
```

```
        # this will flip the direction of rotation. A side effect of the design.
```

```
        # All we need to worry about is the AngularVelocity.
```

```
        Axis = ( Reference ^ Current ).normal()
```

```
        # The current error. If this is 0 then we are in the right place.
```

```
        Error = Current.angle( Reference )
```

```
        # Using a quaternion allows us to perform an neat Axis Angle rotation.
```

```
        EError = MQuaternion( Error, Axis ) . asEulerRotation()
```

```
        # Both of these constants have to be tweaked by hand
```

```
        kP = self.kP    # The Proportional Constant
```

```
        kD = self.kD    # The Derivative Constant
```

```
        #Performing the PID Controll all all 3 Axis of Rotation
```

```
        Px = EError.x
```

```
        Dx = EError.x - self.lastErrorX
```

```
        SolutionX = kP*Px + kD*Dx
```

```
        Py = EError.y
```

```
        Dy = EError.y - self.lastErrorY
```

```
        SolutionY = kP*Py + kD*Dy
```

```
        Pz = EError.z
```

```
        Dz = EError.z - self.lastErrorZ
```

```
        SolutionZ = kP*Pz + kD*Dz
```

```

#Apply the computed torque to the capsule body
#self.w.active[joint.child].body.addTorque( (-SolutionX, -SolutionY, -SolutionZ) )
joint.motor.setParam( luya.ParamVel, SolutionX )
joint.motor.setParam( luya.ParamVel2, SolutionY )
joint.motor.setParam( luya.ParamVel3, SolutionZ )

#Record the last Error
self.lastErrorX = EError.x
self.lastErrorY = EError.y
self.lastErrorZ =EError.z

## Initialize our PD Controller
def init(g):
    g.w = luya.World( ground=True )
    g.jointPIDs = []

    for i in g.w.joints.values():
        g.jointPIDs.append( jointPID( g.w, i, 1, 0, 1 ) )

## Run Our PD Controller
def run(g):
    for pid in g.jointPIDs:
        pid()

    g.w.step( 0.05 )

def end(g):
    pass

```


PID Controller

#Modules needed by the current module. These can be thought of as includes and libraries in one thing.

```
import luyaODE as luya ; reload(luya)
```

```
from maya.mel import *
```

```
from mayaUtils import *
```

```
from luyaUtils import *
```

Initialize our PD Controller

```
def init(g):
```

```
    #Initialise a new Physics World
```

```
    g.w = luya.World( ground=True )
```

```
    #Create a last Error variable to store the rate of change in Error
```

```
    g.lastErrorX = 0
```

```
    g.lastErrorY = 0
```

```
    g.lastErrorZ = 0
```

```
    #By default motors are added to joints in luyaODE.
```

```
    #For the purpose of this experiment we will remove them.
```

```
    del g.w.dynamics.CurrentBase.motor
```

Run Our PD Controller

```
def run(g):
```

```
    #Retrieve the Current Direction of the Inverse Pendulum.
```

```
    Current = Translate("CurrentTip") - Translate("CurrentBase")
```

```
    #Retrieve the Direction of where we want to be.
```

```
    Reference = Translate("ReferenceTip") - Translate("ReferenceBase")
```

```
    # The Axis is the cross product of the Reference and Current vectors.
```

```
    # when Current moves to the other side of Reference the axis will flip,
```

```
    # this will flip the direction of rotation. A side effect of the design.
```

```
    # All we need to worry about is the AngularSpeed.
```

```
    Axis = ( Reference ^ Current ).normal()
```

```
    # The current error. If this is 0 then we are in the right place.
```

```
    Error = Current.angle( Reference )
```

```
    # Using a quaternion allows us to perform an neat Axis Angle rotation.
```

```
    EError = MQuaternion( Error, Axis ) . asEulerRotation()
```

```
    # Both of these constants have to be tweaked by hand
```

```
    kP = 10000    # The Proportional Constant
```

```
    kD = 140000 # The Derivative Constant
```

```
    #Performing the PID Controll all all 3 Axis of Rotation
```

```
    Px = EError.x
```

```
    Dx = EError.x - g.lastErrorX
```

```
    SolutionX = kP*Px + kD*Dx
```

```
    Py = EError.y
```

```
    Dy = EError.y - g.lastErrorY
```

```
    SolutionY = kP*Py + kD*Dy
```

```
    Pz = EError.z
```

```
    Dz = EError.z - g.lastErrorZ
```

```
    SolutionZ = kP*Pz + kD*Dz
```

```
    #Apply the computed torque to the capsule body
```

```
g.w.dynamics.CurrentTip.body.addTorque( (-SolutionX, -SolutionY, -SolutionZ) )
```

```
#Increment the simulation timestep
```

```
g.w.step( 0.05 )
```

```
#Record the last Error
```

```
g.lastErrorX = EError.x
```

```
g.lastErrorY = EError.y
```

```
g.lastErrorZ =EError.z
```

```
def end(g):
```

```
    pass
```

PID Double Inverse Pendulum Controller

#Modules needed by the current module. These can be thought of as includes and libraries in one thing.

```
from maya.api import MVector,MQuaternion
from math import acos,degrees,radians
from mayaUtils import *
from luyaUtils import *
import luyaODE as luya
```

```
halfPi = luya.pi / 2
```

```
def calcTorque(torque,upV,thiV,motor):
```

```
    upV.normalize()
    thiV.normalize()
```

```
    lAxis = (upV ^ thiV)
```

```
    lA = upV ^ lAxis
```

```
    lError = thiV.angle(lA) - halfPi
```

```
    lEuler = MQuaternion(torque(lError),lAxis)
```

```
    strength = -1
```

```
    #In the below example ODE velocities can only be set as Euler rotations.
```

```
    motor.setParam( luya.ParamVel, lEuler.x*strength )
```

```
    motor.setParam( luya.ParamVel2, lEuler.y*strength )
```

```
    motor.setParam( luya.ParamVel3, lEuler.z*strength )
```

```
def equation( balance ):
```

```
    return cos(balance*(halfPi))
```

```
def init( g ):
```

```
    #w.dynamics.shin.body.addForce((10000,0,0))
```

```
    g.torque = PID(10,0,0)
```

```
    g.bSwitch = True
```

```
    g.w=luya.World( ground=True )
```

```
def run( g ):
```

```
    base = Translate("b")
```

```
    knee = Translate("knee") - base
```

```
    shin = Translate("shin") - base
```

```
    thi = Translate("thi") - knee
```

```
    cogX = MVector( - knee.x, 0, - knee.z ) * 2
```

```
    balance = cogX.length() / shin.length()
```

```
    print "Frame %s : %s" % ( g.time, balance)
```

```
    if balance < 1:
```

```
        cogY = MVector( 0, sqrt( pow(shin.length(),2) - pow(cogX.length(),2) ), 0 )
```

```
        cog = cogY + cogX
```

```
        oX = cogX.normal()
```

```
        oY = MVector(0,1,0)
```

```
        offset = balance
```

```
        setPoint = (cog.normal()*offset) + (oY*(1-offset))
```

```
        Translate("cog",cog+knee+base) ; setKeyframe("cog")
```

```
        Translate("setPoint",setPoint+knee+base) ; setKeyframe("setPoint")
```

```
        Translate("x",oX+knee+base) ; setKeyframe("x")
```

```
        Translate("y",oY+knee+base) ; setKeyframe("y")
```

```
        calcTorque(g.torque,setPoint,thi,g.w.dynamics.knee.motor)
    else:
        if g.bSwitch:
            print "Bale out: %s" % g.time
            g.bSwitch = False

    g.w.step(0.01)

def end(g):
    pass
```

Ragdoll Controller

```
import luyaODE as luya
reload(luya)
import time
from fuzzyRules import *

from maya.mel import *
from mayaUtils import *
from math import pi

N = None

def lockStops( world, jointName, mstop=0.0000001 ):
    motor = world.dynamics.__dict__[jointName].motor
    strength = 10000
    motor.setParam(luya.ParamFMax,strength)
    motor.setParam(luya.ParamLoStop,-mstop)
    motor.setParam(luya.ParamHiStop,mstop)
    motor.setParam(luya.ParamFMax2,strength)
    motor.setParam(luya.ParamLoStop2,-mstop)
    motor.setParam(luya.ParamHiStop2,mstop)
    motor.setParam(luya.ParamFMax3,strength)
    motor.setParam(luya.ParamLoStop3,-mstop)
    motor.setParam(luya.ParamHiStop3,mstop)

def init(g):
    ###The Physics Part
    g.startT = time.time()
    g.w = luya.World(ground=True)
    lockStops( g.w, "Physics_SpineB" )
    lockStops( g.w, "Physics_SpineC" )
    lockStops( g.w, "Physics_LHip" )
    lockStops( g.w, "Physics_RHip" )

def run(g):
    g.w.step(0.05)

def end(g):
    print ( time.time() - g.startT )
```

Simple Flocking Controller

```
import luyaODE as luya ; reload(luya)
import fuzzyRules as rules ; reload(rules)
from pyFuzzy import *
from maya.mel import *
from mayaUtils import *

def init(g):
    g.w = luya.World( gravity=(0,0,0), ground=False)
    g.Followers = returnByName("FollowerCollide")

    #The Distance input variable
    u = Stepper(-50, 10000,[ -1000000 , -10, 0, 10, 20, 10000 ])
    g.Distance = InputVariable (u.lower,u.upper,(
        ("TooNear",   Linear(u(0),u(2)), Linear(u(2),u(3)) ),
        ("Perfect",   Linear(u(2),u(3)), Linear(u(3),u(4)) ),
        ("TooFar",    Linear(u(3),u(4)), Linear(u(4),u(5)) )
    ))

    #The Force output variable
    u = Stepper(-200, 200,[ -2, -2, 0, 10, 20 ])
    g.Force = OutputVariable (u.lower,u.upper,(
        ("Reverse",   Linear(u(0),u(1)), Linear(u(1),u(2)) ),
        ("Stop",      Linear(u(1),u(2)), Linear(u(2),u(3)) ),
        ("Speedup",   Linear(u(2),u(3)), Linear(u(3),u(4)) )
    ))
    g.rules = rules.interpret\
    ("""
    if Distance is TooNear : Force = Reverse
    if Distance is Perfect  : Force = Stop
    if Distance is TooFar   : Force = Speedup
    """)

def run(g):
    Distance = g.Distance
    Force = g.Force

    k = 0
    for F in g.Followers :
        Distance.fuzz( ( Translate(F) - Translate("Leader") ).length() )
        Force.begin() ; exec( g.rules )
        setattr( "Push%s.strength" % k, Force.defuzz() )
        k+=1

    g.w.step(0.05)

def end(g):
    pass
```