Spplementary material of the paper: "Active inference and functional parametrisation: differential flatness and smooth random realisations"

Hugues Mounier[a,1,*], Thomas Parr[b], Karl Friston[c,d]

[a] Laboratoire des Signaux et Systèmes, Université Paris-Saclay, CNRS, CentraleSupélec, A3, rue Joliot Curie, Gif sur Yvette, 91192, France
[b] Nuffield Department of Clinical Neurosciences, University of Oxford, UK, Level 6, West Wing, John Radcliffe Hospital, Oxford, OX3 9DU, United Kingdom
[c] Wellcome Centre for Human Neuroimaging, Queen Square Institute of Neurology, University College London, 12 Queen Square, London, WC1N 3AR, United Kingdom
[d] VERSES Research Lab, Los Angeles, CA, USA, 5877 Obama Blvd, Los Angeles, 90016, CA, United States of America

## Abstract

This document is the supplementary material of the paper attempting to marry constructive non-linear control theory techniques with active inference.

*Keywords:* Differential flatness, active inference, periodic smooth random functions, pathwise formulations.

## Contents



[*] Corresponding author

*Email addresses:* hugues.mounier@universite-paris-saclay.fr (Hugues Mounier), thomas.parr@ndcn.ox.ac.uk (Thomas Parr), k.friston@ucl.ac.uk (Karl Friston)

*URL:* https://l2s.centralesupelec.fr/u/mounier-hugues/ (Hugues Mounier), https://tejparr.github.io/ (Thomas Parr), https://www.fil.ion.ucl.ac.uk/~karl/ (Karl Friston)

[1] This work was partially supported by ANR-17-CE40-0005 MindMadeClear (Mathematical Investigation of Neuroscience Dynamics for Meditative Model Identification, Control, Estimation And Observation).

## Supplementary material A. Eye movement models

## Supplementary material B. A very brief recall on differential geometric notions

Consider an affine input system

$$\dot{\boldsymbol{x}} = f(\boldsymbol{x}) + \sum_{i=1}^{n-1} g_i(\boldsymbol{x})u_i = f(\boldsymbol{x}) + g(\boldsymbol{x})\boldsymbol{u}$$

where $f, g_i$ are smooth vector fields on a domain $D \subset \mathbb{R}^n$, $\boldsymbol{x} \in D$, $\boldsymbol{u} \in \mathbb{R}^m$.

**Definition 1.** *Let $r \geqslant 0$ be an integer. A $C^r$* vector field *on $\mathbb{R}^n$ is a mapping $f : D \to \mathbb{R}^n$ of class $C^r$ from an open set $D \subset \mathbb{R}^n$ to $\mathbb{R}^n$. A* smooth vector field *is a mapping $f : D \to \mathbb{R}^n$ of class $C^\infty$.*

Let $h(\boldsymbol{x})$ be a smooth vector field on a domain $D \in \mathbb{R}^n$. The *Lie derivative* of $h$ along $f$, denoted as $L_f h(\boldsymbol{x})$ can be defined (in local coordinates) as

$$\frac{\partial h(\boldsymbol{x})}{\partial \boldsymbol{x}} f(\boldsymbol{x}) = \sum_{i=1}^{n} \frac{\partial h(\boldsymbol{x})}{\partial x_i} f(\boldsymbol{x})$$

since it is a smooth vector field, a Lie derivative operator can be applied to it. Set

$$L_f^i = L_f L_f^{i-1}$$

The *Lie bracket* of $f$ and $g$ can be defined (in local coordinates) as

$$[f, g](\boldsymbol{x}) = \frac{\partial g}{\partial \boldsymbol{x}} f(\boldsymbol{x}) - \frac{\partial f}{\partial \boldsymbol{x}} g(\boldsymbol{x})$$

Iterated lie brackets are denoted as $\mathrm{ad}_f^i g$:

$$\mathrm{ad}_f g = [f, g]$$
$$\mathrm{ad}_f^i g = [f, \mathrm{ad}_f^{i-1} g]$$

Let $f_1, \ldots, f_\eta$ be some vector fields on $D \subset \mathbb{R}^n$. The *distribution* $\Delta$ spanned the vector fields $f_1, \ldots, f_\eta$ is the collection of vector spaces

$$\Delta(\boldsymbol{x}) = \mathrm{span}_{\mathbb{R}^n}\{f_1(\boldsymbol{x}), f_2(\boldsymbol{x}), \ldots, f_\eta(\boldsymbol{x})\}$$

for all $x \in D$. We denote

$$\Delta = \mathrm{span}_{\mathbb{R}^n}\{f_1, f_2, \ldots, f_\eta\}$$

A distribution $\Delta$ is *involutive* if

$$\forall g_1, g_2 \in \Delta, [g_1, g_2] \in \Delta$$

## Supplementary material C. Some flatness simple criteria

There does not exist, at the time of this writing, a general criterion for checking flatness, neither for building flat outputs in a constructive manner. Nevertheless, some peculiar cases are to be noticed.

### *Supplementary material C.1. Necessary and sufficient conditions in peculiar cases*

**Proposition 1.** *Any static state feedback linearizable system is flat.*

See below (Subsection ??, p. ??) for a static state feedback linearizability criterion for affine input systems.

**Proposition 2** (Charlet, Levine and Marino, 1989)**.** *For systems with a single input, dynamic feedback linearization implies static feedback linearization.*

**Proposition 3** (Charlet, Levine and Marino, 1989)**.** *A dynamics affine in the input with $n$ states and $n-1$ inputs is flat as soon as it is controllable (strongly accessible).*

Recall that a dynamics is called affine in the input if it is of the form

$$\dot{\boldsymbol{x}} = f_0(\boldsymbol{x}) + \sum_{i=1}^{n-1} g_i(\boldsymbol{x})\boldsymbol{u}_i$$

A dynamics with $\boldsymbol{x} \in \mathcal{X} \subseteq \mathbb{R}^n$ is strongly accessible if, for all $x \in \mathcal{X}$, there exists a $T > 0$ such that

$$\mathrm{int}\mathsf{R}(\boldsymbol{x}) \neq \varnothing$$

where $\mathrm{int}\mathsf{S}$ denotes the interior of the set $\mathsf{S}$ and $\mathsf{R}(\boldsymbol{x})$ is the reachable set of $\boldsymbol{x}$.

### *Supplementary material C.2. A necessary condition*

**Proposition 4** (Ruled variety criterion, Rouchon, 1995)**.** *Suppose the dynamics $\dot{\boldsymbol{x}} = f(\boldsymbol{x}, \boldsymbol{u})$ is flat. The projection of the sub variety $\boldsymbol{p} = f(\boldsymbol{x}, \boldsymbol{u})$ in the $(\boldsymbol{p}, \boldsymbol{u})$-space ($\boldsymbol{x}$ is here a parameter) onto the $\boldsymbol{p}$-space is a ruled variety for all $\boldsymbol{x}$.*

This criterion means that the elimination of $\boldsymbol{u}$ from the $n$ equations $\dot{\boldsymbol{x}} = f(\boldsymbol{x}, \boldsymbol{u})$ yields $n-m$ equations $F(\boldsymbol{x}, \dot{\boldsymbol{x}}) = 0$ with the following property: for all $(\boldsymbol{x}, \boldsymbol{p})$ such that $F(\boldsymbol{x}, \boldsymbol{p}) = 0$, there exists $\boldsymbol{a} \in \mathbb{R}^n$, $\boldsymbol{a} \neq 0$ such that

$$\forall \lambda \in \mathbb{R}, \quad F(\boldsymbol{x}, \boldsymbol{p} + \lambda \boldsymbol{a}) = 0$$

The variety $F(\boldsymbol{x}, \boldsymbol{p})$ is thus ruled since it contains the line passing through $\boldsymbol{p}$ with direction $\boldsymbol{a}$.

### *Supplementary material C.3. Static state feedback linearizability criterion*

Consider an affine input system

$$\dot{\boldsymbol{x}} = f(\boldsymbol{x}) + \sum_{i=1}^{n-1} g_i(\boldsymbol{x}) u_i = f(\boldsymbol{x}) + g(\boldsymbol{x})\boldsymbol{u}$$

where $f, g_i$ are smooth vector fields on a domain $D \subset \mathbb{R}^n$, $\boldsymbol{x} \in D$, $\boldsymbol{u} \in \mathbb{R}^m$.

**Proposition 5** ((?), Subsect. 8.2 p. 91)**.** *The system with dyanmics $\dot{\boldsymbol{x}} = f(\boldsymbol{x}) + g(\boldsymbol{x})\boldsymbol{u}$ is static state feedback linearisable if, and only if, there is a domain $D_0 \subset D$ such that the following two conditions are staisfied:*

1. *The matrix $\left[g, ad_f g, \ldots, ad_f^{n-1} g\right]$ has rank $n$ for all $x \in D_0$.*
2. *The distribution $\left\{g, ad_f g, \ldots, ad_f^{n-2} g\right\}$ is involutive in $D_0$.*

**Supplementary material D. Flatness definition in a differential geomteric setting**

The original definitions of a system, or model, in our setting, and of differential flatness have been made in the mathematical frameworks of differential algebra and differential geometry of infinite jets. In a differential algebraic setting (see (**??**)), a system, or a model, is a finitely generated differential extension $\mathscr{D}/\mathbb{R}$, and in an infinite jet differential geometry setting (see (**?**)), a system is a pair $(\mathcal{M}, F)$ where $\mathcal{M}$ is a smooth manifold, possibly of infinite dimension, and $F$ is a smooth vector field on $\mathcal{M}$ (see below).

In the differential algebraic setting, a system $\mathscr{D}$ is flat with flat output $\boldsymbol{\omega}$ if $\mathscr{D}/\mathbb{R}\langle\boldsymbol{\omega}\rangle$ is differentially algebraic, and thus, it is a *differential transcendence basis*. In the jet differential geometry setting, a system $(\mathcal{M}, F)$ is flat if it is equivalent to a trivial system, i.e. a system made of a finite number of arbitrary lengths chains of integrators.

Now, let us define a system as *a pair* $(\mathcal{M}, \mathcal{F})$ where $\mathcal{M}$ is a smooth manifold and $F$ is a smooth vector field on $\mathcal{M}$.
Let $\mathbb{R}^\infty_m = \mathbb{R}^m \times \mathbb{R}^m \times \ldots$ be the product of a countable number of copies of $\mathbb{R}^m$.
Let the trivial system $(\mathbb{R}^\infty_m, F_m)$ be the system with coordinates $(\zeta, \zeta^1, \zeta^2, \ldots)$ and vector field

$$F_m(\zeta, \zeta^1, \zeta^2, \ldots) = (\zeta^1, \zeta^2, \ldots)$$

which can model any system made of $m$ chains of integrators of arbitrary lengths.

Let us now precise the notion of equivalence of systems. Take two systems $(\mathcal{M}, F)$ and $(\mathfrak{N}, \mathcal{G})$; they will be called *equivalent* if there exists an invertible transformation exchanging their trajectories. More precisely, let $\psi : \mathcal{M} \to \mathcal{N}$ be an invertible smooth mapping. Then if $t \mapsto \xi(t)$ is a trajectory of $(\mathcal{M}, F)$, i.e.

$$\forall\, \xi, \quad \dot{\xi}(t) = F(\xi(t))$$

then

$$\dot{\zeta}(t) = \mathcal{G}(\psi(\xi(t))) = (\zeta(t))$$

meaning that $t \mapsto \zeta(t)$ is a trajectory of $(\mathcal{N}, \mathcal{G})$. The mapping $\psi$ is called an *endogenous transformation.*
We can then state

**Definition 2.** *The system* $(\mathcal{M}, F)$ *is* flat *if it is equivalent to a trivial system.*

Consider a nonlinear control system of the form

$$\dot{\boldsymbol{x}} = f(\boldsymbol{x}, \boldsymbol{u})$$

with $\boldsymbol{x}(t) \in \mathbb{R}^n$, and $\boldsymbol{u}(t) \in \mathbb{R}^m$. Let this system be flat; it is then equivalent to the trivial system $(\mathbb{R}^\infty_s, F_s)$ where the endogenous transformation takes the form

$$\psi(\boldsymbol{x}, \boldsymbol{u}, \boldsymbol{u}^1, \ldots) = (h(\boldsymbol{x}, \boldsymbol{u}, \boldsymbol{u}^1, \ldots), \dot{h}(\boldsymbol{x}, \boldsymbol{u}, \boldsymbol{u}^1, \ldots), \ldots)$$

Then $\boldsymbol{\omega} = h(\boldsymbol{x}, \boldsymbol{u}, \boldsymbol{u}^1, \ldots)$ is called a *flat* output of the system.
Supposing the flat output takes the form

$$\boldsymbol{\omega} = h(\boldsymbol{x}, \boldsymbol{u}, \boldsymbol{u}^1, \ldots, \boldsymbol{u}^r)$$

all the trajectories $(\boldsymbol{x}(t), \boldsymbol{u}(t))$ can be expressed in terms of the flat output and its derivatives:

$$\begin{aligned} \boldsymbol{x} &= \alpha(\boldsymbol{\omega}, \dot{\boldsymbol{\omega}}, \ldots, \boldsymbol{\omega}^{(\rho_x)}) \\ \boldsymbol{u} &= \beta(\boldsymbol{\omega}, \dot{\boldsymbol{\omega}}, \ldots, \boldsymbol{\omega}^{(\rho_u)}) \end{aligned}$$

And parametrizing the flat output through a basis of known functions reduces a rest to rest steering problem to a system of linear equations (the initial and final conditions on the components of $\boldsymbol{\omega}$ and its derivatives up to order $\max(\rho_x, \rho_u)$).

### Supplementary material E. Smooth random function simulation code

The code for the various plots of this paper, concerning the smooth random functions examples, is reproduced below.

```
#
# Oculomotor movement tracking simulation -- H Mounier -- July 2024
#
import matplotlib.pyplot as plt
import matplotlib as matpl
import numpy as np
import scipy.integrate as spy
import sys
from   dataclasses import dataclass
from   abc import ABC, abstractmethod
from   os import chdir, getcwd, path, makedirs
from   datetime import date, datetime

# use latex fonts
# matpl.style.use('text.usetex') # style not found
matpl.rcParams['text.usetex'] = True
plt.rcParams.update({"text.usetex": True, "font.family": "Computer Modern Roman"})


# Workaround the bug from which we are not in the current working directory
chdir(path.abspath(path.dirname(__file__)))

########################
## Utilitary functions
########################

def printf(format, *args):
    sys.stdout.write(format % args)

def fprintf(stream, format_spec, *args):
    stream.write(format_spec % args)


###############################################
##  parameters
###############################################

class Pars(ABC):
    @abstractmethod
    def get(self):
        pass
```

```
    def writeOnFile(self, fileid):
        pass

@dataclass
class FluctSmoothRandFunPars(Pars):
    L:      float
    lmbda:  float
    scale:  float
    trajName:   str
    def get(self):
        return [self.L, self.lmbda, self.scale]
    def printPars(self):
        printf('\nSmooth Random Function parameters: L = %.5g  $\\lambda$ = %.5g  scale = %.5g\n\n',
                self.L, self.lmbda, self.scale)
    def writeOnFile(self, filename):
        stream = open(filename, "w+")
        firstLine = "\nSmooth Random function Fluctuation Trajectory of " + self.trajName + " parameters\n"
        fprintf(stream, firstLine); fprintf(stream, "-" * len(firstLine) + "\n")
        fprintf(stream, "Smooth random function length L                    : %.5g\n", self.L)
        fprintf(stream, "Smooth random function wavelength $\\lambda$       : %.5g\n", self.lmbda)
        fprintf(stream, "Smooth random function scale                       : %.5g\n", self.scale )
        stream.close()


@dataclass
class ZeroTrajPars:
    def get(self):
        return
    def writeOnFile(self, fileid):
        return


##########################################
## Reference and fluctuation trajectories
##########################################

class Traj(ABC):
    @abstractmethod
    def traj(self, t):
        pass


@dataclass
class SmoothRandFunTraj(Traj):
    L:      float
    lmbda:  float
    scale:  float
    r:      float
    a:      np.ndarray[float, np.dtype[np.float64]]
    b:      np.ndarray[float, np.dtype[np.float64]]
    def __init__(self, L, lmbda, scale):
        self.L = L; self.lmbda = lmbda; self.scale = scale
        r = int(np.floor(self.L/self.lmbda)); self.r = r
        var = 1/(2*r+1)
        self.a = np.empty([r+1], dtype = float)
        self.b = np.empty([r+1], dtype = float)
```

```
        for i in range(1, r+1):
            self.a[i] = np.random.normal(0, var)
            self.b[i] = np.random.normal(0, var)
    def traj(self, t):
        L = self.L; scale = self.scale
        a = self.a; b = self.b; r = self.r
        pi = np.pi
        zeta = a[0]; dotzeta = 0; ddotzeta = 0
        for j in range(1, r):
            nuj = 2*pi*j/L
            zeta     = zeta     + a[j] * np.cos(nuj*t)        + b[j] * np.sin(nuj*t)
            dotzeta  = dotzeta  - a[j]*nuj * np.sin(nuj*t)    + b[j]*nuj * np.cos(nuj*t)
            ddotzeta = ddotzeta - a[j]*nuj**2 * np.cos(nuj*t) - b[j]*nuj**2 * np.sin(nuj*t)
        return [scale*zeta, scale*dotzeta, scale*ddotzeta]

@dataclass
class ZeroTraj(Traj):
    def traj(self, t):
        return [0*t, 0*t, 0*t]


######################
## Plotting functions
######################

def indiviualPlotAndSave(fid, ttoPlot, xToPlot,
                  xLabelStr, yLabelStr, titleStr, saveFigStr, plotsDirectory, figNameEnd,
                  firstLast = "first&last", lineThickness = 1.5, lineColor = "blue"):
        plt.figure(fid);
        if ("first" in firstLast):
            plt.cla(); plt.clf();
        plt.plot(ttoPlot, xToPlot, linewidth = lineThickness, color = lineColor)
        if ("last" in firstLast):
            plt.xlabel(xLabelStr); plt.ylabel(yLabelStr);
            plt.title(titleStr);
            figName = plotsDirectory + '/' + saveFigStr % (figNameEnd)
            plt.grid(); plt.savefig(figName, format="pdf");
            printf('Figure %s saved\n', saveFigStr % (figNameEnd))
        fid = fid + 1;
        return fid


def createFluctuationTrajs(trajType):
    # default parameter values
    L = 8; lmbda = 2; scale = 1e-5;   # fluctPars
    match trajType:
            case s if 'SRFTiSc' in s:  # Smmoth Random Function Tiny Scale
            # r = floor(L/lmbda) - 2*pi*j/L = N*2*pi <=> L = j/N ; j = 1, ..., r
                L = 8; lmbda = 2; scale = 2e-6;   # fluctPars
            case s if 'SRFSmSc' in s:  # Smmoth Random Function Small Scale
                L = 8; lmbda = 2; scale = 1e-1;   # fluctPars
            case s if 'SRFMdSc' in s:  # Smmoth Random Function Medium Scale
                L = 8; lmbda = 2; scale = 1;   # fluctPars
            case s if 'SRFBgSc' in s:    # Smmoth Random Function Big Scale
                L = 8; lmbda = 2; scale = 5;   # fluctPars
            case s if 'SRFVBgSc' in s:  # Smmoth Random Function Very Big Scale
```

```
            L = 8; lmbda = 2; scale = 10;    # fluctPars
        case s if 'SRFHgSc' in s:     # Smmoth Random Function Huge Scale
            L = 8; lmbda = 2; scale = 100;    # fluctPars
        case s if 'SRFLowLambda' in s:
            L = 8; lmbda = 0.1; scale = 1;    # fluctPars
        case default:
            print(trajType)
            print('No matching fluctuation type; default values taken')
    printf('Trajectory type : %s -- L = %.5g  lmbda = %.5g  scale = %.5g\n', trajType, L, lmbda, scale)
    fluctTrajPars = FluctSmoothRandFunPars(L, lmbda, scale, "abscissa x");
    fluctTraj     = SmoothRandFunTraj(L, lmbda, scale)
    fluctTrajPars.printPars()
    return [fluctTrajPars, fluctTraj]


def performTask(whatToDo, refTrajType, refFluctType, saveFigures):
    # directory and file names $$$
    plotsDirectory = './PlotsFromSimulations'
    if not path.exists(plotsDirectory):
        makedirs('./PlotsFromSimulations')
    t = np.arange(0, 50, 0.01); fid = 1


    # Plot no 1
    refFluctType = 'SRFMdSc'
    figNameEnd    = 'smoothRandFun-' +  whatToDo  + '-' + refFluctType
    paramsFilename = plotsDirectory + '/pars-' + figNameEnd + ".txt"
    firstLast = "first&last"; lineThick = 1.5; lineColor = "blue"

    [fluctTrajPars, fluctTraj] = createFluctuationTrajs(refFluctType)
    [xFluct, dotxFluct, ddotxFluct] = fluctTraj.traj(t);
    fid = indiviualPlotAndSave(fid, t, xFluct, 'Time (s)', '$\\zeta (t)$', 'Fluctuation trajectory',
                            "zeta-%s.pdf", plotsDirectory, figNameEnd, firstLast, lineThick, lineColor)
    fluctTrajPars.writeOnFile(paramsFilename)

    # Plot no 2
    refFluctType = 'SRFLowLambda'
    figNameEnd    = 'smoothRandFun-' +  whatToDo + '-' + refFluctType
    paramsFilename = plotsDirectory + '/pars-' + figNameEnd + ".txt"
    [fluctTrajPars, fluctTraj] = createFluctuationTrajs(refFluctType)
    [xFluct, dotxFluct, ddotxFluct] = fluctTraj.traj(t);
    fid = indiviualPlotAndSave(fid, t, xFluct, 'Time (s)', '$\\zeta (t)$', 'Fluctuation trajectory',
                            "zeta-%s.pdf", plotsDirectory, figNameEnd, firstLast, lineThick, lineColor)
    fluctTrajPars.writeOnFile(paramsFilename)




def main():
    # reference traj, fluctuation and action law choice
    saveFigures  = True                 # Save or plot
    taskId = 1
    ctrlType = 'flatCtrlLaw'; refTrajType  = 'tanhTr';
    # Open loop control computation and plot, no fluctuation
    whatToDo = 'Task-' + str(taskId) + '-plotFluct';  refFluctType = 'SRFSmSc';
    performTask(whatToDo, refTrajType, refFluctType, saveFigures)
```

```
    taskId = taskId + 1
    return

main()
```

## Supplementary material F. Eye movement simulation code

The simulation code for the various plots of this paper, concerning the oculomotor example, is reproduced below.

```
#
# Oculomotor movement tracking simulation -- H Mounier -- July 2024
#
import matplotlib.pyplot as plt
import matplotlib as matpl
import numpy as np
import scipy.integrate as spy
import sys
from   dataclasses import dataclass
from   abc import ABC, abstractmethod
from   os import chdir, getcwd, path, makedirs
from   datetime import date, datetime

# use latex fonts
# matpl.style.use('text.usetex') # style not found
matpl.rcParams['text.usetex'] = True
plt.rcParams.update({"text.usetex": True, "font.family": "Computer Modern Roman",
"axes.labelsize": 14, "font.size": 14, "xtick.labelsize": 12, "ytick.labelsize": 12})

# Workaround the bug from which we are not in the current working directory
chdir(path.abspath(path.dirname(__file__)))

#######################
## Utilitary functions
#######################

def printf(format, *args):
    sys.stdout.write(format % args)

def fprintf(stream, format_spec, *args):
    stream.write(format_spec % args)


##############################################
## Physical, time and feedback gain parameters
##############################################

class Pars(ABC):
    @abstractmethod
    def get(self):
        pass
    def writeOnFile(self, fileid):
        pass
```

```
@dataclass
class PhysPars(Pars):
    me:     float = 7e-3        # Eye mass in kg
    re:     float = 12e-3       # Eye radius in m
    Ie:     float = 14e-3 * 12e-3**2 / 5 # Eye interia (supposed spherical)
    d:      float = 10           # Distance of eye from the visual scene (m)
    x0:     float = 0            # Abscissa of eye's projection on visual scene (m)
    y0:     float = 0            # Ordinate of eye's projection on visual scene (m)
    def get(self):
        self.Ie = 2*self.me * self.re**2 / 5
        return [self.me, self.re, self.Ie, self.d, self.x0, self.y0]
    def writeOnFile(self, fileid):
        fprintf(fileid, "\nPhysical parameters\n")
        fprintf(fileid, "-------------------\n")
        fprintf(fileid, "Eye mass                                  : %.5g kg\n", self.me)
        fprintf(fileid, "Eye radius                                : %.5g m\n", self.re)
        fprintf(fileid, "Eye inertia                               : %.5g kg.m^2\n", self.Ie)
        fprintf(fileid, "Distance from visual scene                : %.5g m\n", self.d)
        fprintf(fileid, "Eye's projection abscissa on visual scene : %.5g m\n", self.x0)
        fprintf(fileid, "Eye's projection ordinate on visual scene : %.5g m\n", self.y0)

@dataclass
class TimePars(Pars):
    tini:   float
    tend:   float
    dt:     float
    Xi:     float
    Yi:     float
    vPsii:  float
    vPhii:  float
    def get(self):
        return [self.tini, self.tend, self.dt, self.Xi, self.Yi, self.vPsii, self.vPhii]
    def writeOnFile(self, fileid):
        fprintf(fileid, "\nSimulation time parameters\n")
        fprintf(fileid, "--------------------------\n")
        fprintf(fileid, "Initial simulation time                   : %.5g s\n", self.tini)
        fprintf(fileid, "Final simulation time                     : %.5g s\n", self.tend)
        fprintf(fileid, "sampling interval                         : %.5g s\n", self.dt)
        fprintf(fileid, "Initial abscissa x                        : %.5g s\n", self.Xi)
        fprintf(fileid, "Initial ordinate y                        : %.5g s\n", self.Yi)
        fprintf(fileid, "Initial orientation (yaw) rate $\\psi$     : %.5g s\n", self.vPsii)
        fprintf(fileid, "Initial elevation (pitch) rate $\\phi$     : %.5g s\n", self.vPhii)

@dataclass
class GainPars(Pars):
    lambda0x:    float
    lambda1x:    float
    lambda0y:    float
    lambda1y:    float
    def get(self):
        return [self.lambda0x, self.lambda1x, self.lambda0y, self.lambda1y]
    def writeOnFile(self, fileid):
        fprintf(fileid, "\nFeedback gain parameters\n")
        fprintf(fileid, "------------------------\n")
        fprintf(fileid, "x error feedback gain $\\lambda_{0x}$           : %.5g\n", self.lambda0x)
        fprintf(fileid, "x error derivative feedback gain $\\lambda_{1x}$   : %.5g\n", self.lambda1x)
```

```python
        fprintf(fileid, "y error feedback gain $\\lambda_{0y}$              : %.5g\n", self.lambda0y)
        fprintf(fileid, "y error derivative feedback gain $\\lambda_{1y}$   : %.5g\n", self.lambda1y)

@dataclass
class TanhRefTrajPars(Pars):
    lowVal:     float
    highVal:    float
    stiff:      float
    tRaise:     float
    trajName:   str
    def get(self):
        return [self.lowVal, self.highVal, self.stiff, self.tRaise]
    def writeOnFile(self, fileid):
        firstLine = "\nTanh() Trajectory of " + self.trajName + " parameters\n"
        fprintf(fileid, firstLine); fprintf(fileid, "-" * len(firstLine) + "\n")
        fprintf(fileid, "Initial tanh() reference trajectory value         : %.5g\n", self.lowVal)
        fprintf(fileid, "Final tanh() reference trajectory value           : %.5g\n", self.highVal)
        fprintf(fileid, "tanh() reference trajectory stiffness             : %.5g\n", self.stiff)
        fprintf(fileid, "tanh() reference trajectory mid value time        : %.5g\n", self.tRaise)

@dataclass
class TanhHatRefTrajPars(Pars):
    lowVal:     float
    highVal:    float
    stiffR:     float
    stiffD:     float
    tRaise:     float
    tGoDown:    float
    trajName:   str
    def get(self):
        return [self.lowVal, self.highVal, self.stiffR, self.stiffD, self.tRaise, self.tGoDown]
    def writeOnFile(self, fileid):
        firstLine = "\nTanhHat() Trajectory of " + self.trajName + " parameters\n"
        fprintf(fileid, firstLine); fprintf(fileid, "-" * len(firstLine) + "\n")
        fprintf(fileid, "Initial tanhHat() ref traj value                  : %.5g\n", self.lowVal)
        fprintf(fileid, "Final tanhHat() ref traj value                    : %.5g\n", self.highVal)
        fprintf(fileid, "tanhHat() ref traj begin stiffness                : %.5g\n", self.stiffR)
        fprintf(fileid, "tanhHat() ref traj end stiffness                  : %.5g\n", self.stiffD)
        fprintf(fileid, "tanhHat() ref traj beginning mid value time       : %.5g\n", self.tRaise)
        fprintf(fileid, "tanhHat() ref traj end mid value time             : %.5g\n", self.tGoDown)

@dataclass
class QuaterfoilTrajPars(Pars):
    a:          float
    trajName:   str
    def get(self):
        return [self.a]
    def writeOnFile(self, fileid):
        firstLine = "\nQuaterfoil() Trajectory of " + self.trajName + " parameters\n"
        fprintf(fileid, firstLine); fprintf(fileid, "-" * len(firstLine) + "\n")
        fprintf(fileid, "Quaterfoil ref traj a parameter value            : %.5g\n", self.a)

@dataclass
class HypocycloidTrajPars(Pars):
    a:     float
    b:     float
```

```
    trajName:   str
    def get(self):
        return [self.a, self.b]
    def writeOnFile(self, fileid):
        firstLine = "\nHypocycloid() Trajectory of " + self.trajName + " parameters\n"
        fprintf(fileid, firstLine); fprintf(fileid, "-" * len(firstLine) + "\n")
        fprintf(fileid, "Hypocycloid ref traj a parameter value            : %.5g\n", self.a)
        fprintf(fileid, "Hypocycloid ref traj b parameter value            : %.5g\n", self.b)

@dataclass
class HypotrochoidTrajPars(Pars):
    a:    float
    b:    float
    c:    float
    trajName:   str
    def get(self):
        return [self.a, self.b, self.c]
    def writeOnFile(self, fileid):
        firstLine = "\nHypocycloid() Trajectory of " + self.trajName + " parameters\n"
        fprintf(fileid, firstLine); fprintf(fileid, "-" * len(firstLine) + "\n")
        fprintf(fileid, "Hypotrochoid ref traj a parameter value           : %.5g\n", self.a)
        fprintf(fileid, "Hypotrochoid ref traj b parameter value           : %.5g\n", self.b)
        fprintf(fileid, "Hypotrochoid ref traj c parameter value           : %.5g\n", self.c)


@dataclass
class FluctSmoothRandFunPars(Pars):
    L:      float
    lmbda:  float
    scale:  float
    trajName:   str
    def get(self):
        return [self.L, self.lmbda, self.scale]
    def writeOnFile(self, fileid):
        firstLine = "\nSmooth Random function Fluctuation Trajectory of " + self.trajName + " parameters\n"
        fprintf(fileid, firstLine); fprintf(fileid, "-" * len(firstLine) + "\n")
        fprintf(fileid, "Smooth random function length L                   : %.5g\n", self.L)
        fprintf(fileid, "Smooth random function wavelength $\\lambda$       : %.5g\n", self.lmbda)
        fprintf(fileid, "Smooth random function scale                      : %.5g\n", self.scale )

@dataclass
class PertXYUPars:
    pertXPars:      Pars
    pertYPars:      Pars
    pertUPsiPars:   Pars
    pertUPhiPars:   Pars
    def get(self):
        return [self.pertXPars, self.pertYPars, self.pertUPsiPars, self.pertUPhiPars]
    def writeOnFile(self, fileid):
        self.pertXPars.writeOnFile(self, fileid)
        self.pertYPars.writeOnFile(self, fileid)
        self.pertUPsiPars.writeOnFile(self, fileid)
        self.pertUPhiPars.writeOnFile(self, fileid)

@dataclass
class AllPars(Pars):
```

```
    physPars:           PhysPars
    timePars:           TimePars
    gainPars:           GainPars
    refTrajXPars:       Pars
    refTrajYPars:       Pars
    fluctTrajXPars:     Pars
    fluctTrajYPars:     Pars
    fluctTrajUPsiPars:  Pars
    fluctTrajUPhiPars:  Pars
    def get(self):
        return [self.physPars, self.timePars, self.gainPars, self.refTrajXPars, self.refTrajYPars,
                self.fluctTrajXPars, self.fluctTrajYPars, self.fluctTrajUPsiPars, self.fluctTrajUPhiPars]
    def writeOnFile(self, filename):
        stream = open(filename, "w+")
        fprintf(stream, 'Oculomotor simulation parameters\n')
        fprintf(stream, '--------------------------------\n')
        self.physPars.writeOnFile(stream)
        self.timePars.writeOnFile(stream)
        self.gainPars.writeOnFile(stream)
        self.refTrajXPars.writeOnFile(stream)
        self.refTrajYPars.writeOnFile(stream)
        self.fluctTrajXPars.writeOnFile(stream)
        self.fluctTrajYPars.writeOnFile(stream)
        self.fluctTrajUPsiPars.writeOnFile(stream)
        self.fluctTrajUPhiPars.writeOnFile(stream)
        stream.close()


@dataclass
class ZeroTrajPars:
    def get(self):
        return
    def writeOnFile(self, fileid):
        return


###########################################
## Reference and fluctuation trajectories
###########################################

class Traj(ABC):
    @abstractmethod
    def traj(self, t):
        pass

@dataclass
class TanhRefTraj(Traj):
    lowVal:     float
    highVal:    float
    stiff:      float
    tRaise:     float
    def traj(self, t):
      aR    = (self.highVal-self.lowVal)/2;
      phi   =  np.tanh(self.stiff*(t-self.tRaise));
      y     =  aR * (1+ phi) + self.lowVal;
      doty  =  aR*self.stiff * (1-phi**2);
```

```
        ddoty = -2*aR*self.stiff**2 * phi*(1-phi**2);
        return [y, doty, ddoty]

@dataclass
class TanhHatRefTraj(Traj):
    lowVal:     float
    highVal:    float
    stiffR:     float
    stiffD:     float
    tRaise:     float
    tGoDown:    float
    def traj(self, t):
        stR = self.stiffR; stD = self.stiffD;
        lowVal = self.lowVal; highVal = self.highVal
        a = 0.5*(highVal-lowVal);
        phi = np.tanh(stR*(t-self.tRaise));
        psi = np.tanh(-stD*(t-self.tGoDown));
        y   = a * (phi + psi) + lowVal;
        doty = a * (stR * (1-phi**2) - stD * (1-psi**2));
        ddoty = -2*a * (stR**2.*phi * (1-phi**2) - stD**2.*psi * (1-psi**2))
        dot3y = -2*a * ( stR**3 * (1-phi**2) * (1-3*phi**2) -
                    stD**3 * (1-psi**2) * (1-3*psi**2) )
        return [y, doty, ddoty]

@dataclass
class QuaterfoilTrajX(Traj):
    a:    float
    def traj(self, t):
        a = self.a; cos = np.cos(t); sin = np.sin(t);
        x     =  2*a * sin**2 * cos
        dotx  = -2*a*sin**3 + 4*a*sin*cos**2
        ddotx = -2*a*(7*sin**2 - 2*cos**2)*cos
        return [x, dotx, ddotx]

@dataclass
class QuaterfoilTrajY(Traj):
    a:    float
    def traj(self, t):
        a = self.a; cos = np.cos(t); sin = np.sin(t);
        y     =  2*a * cos**2 * sin
        doty  = -4*a*sin**2*cos + 2*a*cos**3
        ddoty =  2*a*(2*sin**2 - 7*cos**2)*sin
        return [y, doty, ddoty]

@dataclass
class HypocycloidTrajX(Traj):
    a:    float
    b:    float
    def traj(self, t):
        a = self.a; b = self.b;
        cos = np.cos(t); sin = np.sin(t);
        cosab = np.cos((a-b)*t/b); sinab = np.sin((a-b)*t/b);
        x     =  (a-b) * cos + b * cosab
        dotx  = -(a-b) * sin - (a-b) * sinab
        ddotx = -(a-b) * cos - (a-b)**2 * cosab / b
        return [x, dotx, ddotx]
```

```
@dataclass
class HypocycloidTrajY(Traj):
    a:    float
    b:    float
    def traj(self, t):
        a = self.a; b = self.b;
        cos = np.cos(t); sin = np.sin(t);
        cosab = np.cos((a-b)*t/b); sinab = np.sin((a-b)*t/b);
        y     =  (a-b) * sin - b * sinab
        doty  =  (a-b) * cos - (a-b) * cosab
        ddoty = -(a-b) * sin - (a-b)**2 * sinab / b
        return [y, doty, ddoty]


@dataclass
class HypotrochoidTrajX(Traj):
    a:    float
    b:    float
    c:    float
    def traj(self, t):
        a = self.a; b = self.b; c = self.c
        cos = np.cos(t); sin = np.sin(t);
        cosab = np.cos((a-b)*t/b); sinab = np.sin((a-b)*t/b);
        x     =  (a-b) * cos + c * cosab
        dotx  = -(a-b) * sin - c*(a-b)/b * sinab
        ddotx = -(a-b) * cos - c*((a-b)/b)**2 * cosab
        return [x, dotx, ddotx]

@dataclass
class HypotrochoidTrajY(Traj):
    a:    float
    b:    float
    c:    float
    def traj(self, t):
        a = self.a; b = self.b; c = self.c
        cos = np.cos(t); sin = np.sin(t);
        cosab = np.cos((a-b)*t/b); sinab = np.sin((a-b)*t/b);
        y     =  (a-b) * sin - c * sinab
        doty  =  (a-b) * cos - c*(a-b)/b * cosab
        ddoty = -(a-b) * sin - c*((a-b)/b)**2 * sinab
        return [y, doty, ddoty]


@dataclass
class SmoothRandFunTraj(Traj):
    L:      float
    lmbda:  float
    scale:  float
    r:      float
    a:      np.ndarray[float, np.dtype[np.float64]]
    b:      np.ndarray[float, np.dtype[np.float64]]
    def __init__(self, L, lmbda, scale):
        self.L = L; self.lmbda = lmbda; self.scale = scale
        r = int(np.floor(self.L/self.lmbda)); self.r = r
        var = 1/(2*r+1)
```

```
        self.a = np.empty([r+1], dtype = float)
        self.b = np.empty([r+1], dtype = float)
        for i in range(1, r+1):
            self.a[i] = np.random.normal(0, var)
            self.b[i] = np.random.normal(0, var)
    def traj(self, t):
        L = self.L; scale = self.scale
        a = self.a; b = self.b; r = self.r
        pi = np.pi
        zeta = a[0]; dotzeta = 0; ddotzeta = 0
        for j in range(1, r):
            nuj = 2*pi*j/L
            zeta     = zeta     + a[j] * np.cos(nuj*t)        + b[j] * np.sin(nuj*t)
            dotzeta  = dotzeta  - a[j]*nuj * np.sin(nuj*t)    + b[j]*nuj * np.cos(nuj*t)
            ddotzeta = ddotzeta - a[j]*nuj**2 * np.cos(nuj*t) - b[j]*nuj**2 * np.sin(nuj*t)
        return [scale*zeta, scale*dotzeta, scale*ddotzeta]

@dataclass
class ZeroTraj(Traj):
    def traj(self, t):
        return [0*t, 0*t, 0*t]

@dataclass
class PertXYU:
    pertX:      Traj
    pertY:      Traj
    pertUPsi:   Traj
    pertUPhi:   Traj
    def pert(self, t):
        [dx, dotdx, ddotdx]          = self.pertX.traj(t)
        [dy, dotdy, ddotdy]          = self.pertY.traj(t)
        [dupsi, dotdupsi, ddotdupsi] = self.pertUPsi.traj(t)
        [duphi, dotduphi, ddotduphi] = self.pertUPsi.traj(t)
        return [dx, dotdx, ddotdx, dy, dotdy, ddotdy, dupsi, duphi]


#########################
## Tracking control laws
#########################

class TrackCtrlElem(ABC):
    @abstractmethod
    def ctrl(self, e, dote):
        pass

@dataclass
class TrackCtrlPD(TrackCtrlElem):
    Kp:     float
    Kd:     float
    def ctrl(self, ddotXr, e, dote):
        return (ddotXr -self.Kp * e - self.Kd * dote);

@dataclass
class TrackCtrlFlat:
    physPars:       PhysPars
    refTrajX:       Traj
```

```
    refTrajY:       Traj
    fluctXYU:       Traj
    elemTrackCtrlX: TrackCtrlElem
    elemTrackCtrlY: TrackCtrlElem
    def ctrl(self, t, X):
        if np.isscalar(t) == True:
            x = X[0];  y = X[1]; vPsi = X[2]; vPhi = X[3]
        else:
            x = X[:,0]; y = X[:,1]; vPsi = X[:,2]; vPhi = X[:,3];
        me, re, Ie, d, x0, y0 = self.physPars.get()
        xr, dotxr, ddotxr = self.refTrajX.traj(t)
        yr, dotyr, ddotyr = self.refTrajY.traj(t)
        zetx, dotzetx, ddotzetx, zety, dotzety, \
              ddotzety, zetupsi, zetuphi = self.fluctXYU.pert(t)
        dotx   = vPsi * (d**2 + (x-x0 - zetx)**2) / d + dotzetx
        doty   = vPhi * (d**2 + (y-y0 - zety)**2) / d + dotzety
        # error computations
        ex     = x - xr; dotex = dotx - dotxr;
        ey     = y - yr; dotey = doty - dotyr;
        # elementary tracking feedback laws
        vx = self.elemTrackCtrlX.ctrl(ddotxr, ex, dotex);
        vy = self.elemTrackCtrlY.ctrl(ddotyr, ey, dotey);
        deltax = d**2 + (x-x0-zetx)**2; deltay = d**2 + (y-y0-zety)**2
        # action (input control) laws computation
        uPsi = (2*d*Ie/deltax) * (-2*vPsi*(dotx-dotzetx)/d - ddotzetx + vx) - zetupsi
        uPhi = (d*Ie/deltay)   * (-2*vPhi*(doty-dotzety)/d - ddotzety + vy) - zetuphi
        return [uPsi, uPhi]

@dataclass
class TrackCtrlFlatBlind:
    physPars:       PhysPars
    refTrajX:       Traj
    refTrajY:       Traj
    fluctXYU:        Traj
    elemTrackCtrlX: TrackCtrlElem
    elemTrackCtrlY: TrackCtrlElem
    def ctrl(self, t, X):
        if np.isscalar(t) == True:
            x = X[0];  y = X[1]; vPsi = X[2]; vPhi = X[3]
        else:
            x = X[:,0]; y = X[:,1]; vPsi = X[:,2]; vPhi = X[:,3];
        me, re, Ie, d, x0, y0 = self.physPars.get()
        xr, dotxr, ddotxr = self.refTrajX.traj(t)
        yr, dotyr, ddotyr = self.refTrajY.traj(t)
        zetx, dotzetx, ddotzetx, zety, dotzety, \
                ddotzety, zetupsi, zetuphi = [0, 0, 0, 0, 0, 0, 0, 0]
        dotx   = vPsi * (d**2 + (x-x0 - zetx)**2) / d + dotzetx
        doty   = vPhi * (d**2 + (y-y0 - zety)**2) / d + dotzety
        # error computations
        ex     = x - xr; dotex = dotx - dotxr;
        ey     = y - yr; dotey = doty - dotyr;
        # elementary tracking feedback laws
        vx = self.elemTrackCtrlX.ctrl(ddotxr, ex, dotex);
        vy = self.elemTrackCtrlY.ctrl(ddotyr, ey, dotey);
        deltax = d**2 + (x-x0-zetx)**2; deltay = d**2 + (y-y0-zety)**2
        # action (input control) laws computation
```

```python
        uPsi = (2*d*Ie/deltax) * (-2*vPsi*(dotx-dotzetx)/d - ddotzetx + vx) - zetupsi
        uPhi = (d*Ie/deltay)   * (-2*vPhi*(doty-dotzety)/d - ddotzety + vy) - zetuphi
        return [uPsi, uPhi]

@dataclass
class OpenLoopCtrlFlat:
    physPars:       PhysPars
    refTrajX:       Traj
    refTrajY:       Traj
    fluctXYU:       Traj
    def ctrl(self, t):
        me, re, Ie, d, x0, y0 = self.physPars.get()
        xr, dotxr, ddotxr = self.refTrajX.traj(t)
        yr, dotyr, ddotyr = self.refTrajY.traj(t)
        zetx, dotzetx, ddotzetx, zety, dotzety, \
                ddotzety, zetupsi, zetuphi = self.fluctXYU.pert(t)
        deltax = d**2 + (xr-x0-zetx)**2; deltay = d**2 + (yr-y0-zety)**2
        dotxzetxr  = dotxr - dotzetx;    dotyzetyr = dotyr - dotzety;
        ddotxzetxr = ddotxr - ddotzetx; ddotyzetyr = ddotyr - ddotzety;
        # reference state variables
        vPsir = d * dotxzetxr / deltax;
        vPhir = d * dotyzetyr / deltay;
        # action (input control) laws computation
        uPsir = 2*Ie * (d*ddotxzetxr*deltax - 2*d * (xr-x0-zetx) * dotxzetxr**2) - zetupsi
        uPhir =   Ie * (d*ddotyzetyr*deltay - 2*d * (yr-y0-zety) * dotyzetyr**2) - zetuphi
        return [vPsir, vPhir, uPsir, uPhir]


######################
## Plotting functions
######################
def oculomotorPlot(t, X, xr, yr, uPsi, uPhi, fluctXYU):
    x = X[:,0]; y = X[:,1]; vPsi = X[:,2]; vPhi = X[:,3];
    zetx, dotzetx, ddotzetx, zety, dotzety, \
                 ddotzety, zetupsi, zetuphi = fluctXYU.pert(t)
    fig, ((ax1, ax2), (ax3, ax4), (ax5, ax6)) = plt.subplots(3, 2);
    ax1.plot(t,x,"blue", label = "$x$");
    ax1.plot(t,xr,"red", label = "$x_r$");
    ax1.set_ylabel("x, $x_r$"); ax1.grid()
    ax2.plot(t,y,"blue", label = "$y$")
    ax2.plot(t,yr,"red", label = "$y_r$")
    ax2.set_ylabel("$y$, $y_r$"); ax2.grid()
    ax3.plot(t,uPsi,"blue", label = '$u_\\psi$');
    ax3.set_ylabel("$u_\\psi$"); ax3.grid()
    ax4.plot(t,uPhi,"blue", label = '$u_\\phi$')
    ax4.set_ylabel("$u_\\phi$"); ax4.grid()
    ax5.plot(t,zetx,"blue", label = '$\\zeta_x$')
    ax5.set_ylabel("$\\zeta_x$"); ax5.grid()
    ax6.plot(t,zety,"blue", label = '$\\zeta_y$')
    ax6.set_ylabel("$\\zeta_y$"); ax6.grid()
    ax3.set_xlabel("Time (s)"); ax4.set_xlabel("Time (s)");
    fig.suptitle("Oculomotor movement")
    ax1.legend(); ax2.legend(); ax3.legend();
    ax4.legend();
    plt.figure(2)
```

```
    plt.plot(xr, yr, "red", label = "$x_r, y_r$")
    plt.plot(x, y, "blue", label = "$x, y$")
    plt.xlabel('x (m)');    plt.ylabel('y (m)');
    plt.title('Evolution curve $x(t), y(t)$')
    plt.legend(); plt.grid(); plt.show()

def openLoopPlot(t, XRefr, Ur, F):
    [uPsir, uPhir] = Ur; [zetx, zety, zetupsi, zetuphi] = F; [xr, yr] = XRefr
    fig, ((ax1, ax2), (ax3, ax4), (ax5, ax6)) = plt.subplots(3, 2);
    ax1.plot(t,xr,"red", label = "$x_r$");
    ax1.set_ylabel("$x_r$"); ax1.grid()
    ax2.plot(t,yr,"red", label = "$y_r$")
    ax2.set_ylabel("$y_r$"); ax2.grid()
    ax1.set_xlabel("Time (s)"); ax2.set_xlabel("Time (s)");
    ax3.plot(t,uPsir,"blue", label = '$u_\\psi$');
    ax3.set_ylabel("$u_{\\psi r}$"); ax3.grid()
    ax4.plot(t,uPhir,"blue", label = '$u_\\phi$')
    ax4.set_ylabel("$u_{\\phi r}$"); ax4.grid()
    ax3.set_xlabel("Time (s)"); ax4.set_xlabel("Time (s)");
    fig.suptitle("Oculomotor movement open loop action law")
    ax1.legend(); ax2.legend(); ax3.legend();
    ax4.legend();  plt.grid(); plt.show()


def indiviualPlotAndSave(figTab, fid, ttoPlot, xToPlot,
                xLabelStr, yLabelStr, titleStr, saveFigStr, plotsDirectory, figNameEnd,
                firstLast = "first&last", lineThickness = 1.5, lineColor = "blue"):
        figTab[fid] = plt.figure(fid);
        if ("first" in firstLast):
            plt.cla(); plt.clf();
        plt.plot(ttoPlot, xToPlot, linewidth = lineThickness, color = lineColor)
        if ("last" in firstLast):
            plt.xlabel(xLabelStr); plt.ylabel(yLabelStr);
            plt.title(titleStr);
            figName = plotsDirectory + '/' + saveFigStr % (figNameEnd)
            plt.grid(); plt.savefig(figName, format="pdf");
            printf('Figure %s saved\n', saveFigStr % (figNameEnd))
        fid = fid + 1;
        return fid

def indiviualZoomedPlotAndSave(figTab, fid, ttoPlot, xToPlot,
                xLabelStr, yLabelStr, titleStr, saveFigStr, plotsDirectory, figNameEnd,
                firstLast = "first&last", lineThickness = 1.5, lineColor = "blue", zoomLevel = 8):
        figTab[fid] = plt.figure(fid);
        if ("first" in firstLast):
            plt.cla(); plt.clf();
        tini = ttoPlot[0]; tend = ttoPlot[-1]
        zoomedtrange = (tend-tini) / zoomLevel
        tzoomlast = tini + zoomedtrange
        tZoomedToPlot = np.extract(ttoPlot < tzoomlast, ttoPlot)
        xZoomedToPlot = xToPlot[0:len(tZoomedToPlot)]
        plt.plot(tZoomedToPlot, xZoomedToPlot, linewidth = lineThickness, color = lineColor)
        if ("last" in firstLast):
            plt.xlabel(xLabelStr); plt.ylabel(yLabelStr);
            plt.title(titleStr);
            plt.grid(); plt.savefig(plotsDirectory + '/' + saveFigStr % (figNameEnd), format="pdf");
```

```
        printf('Figure %s saved\n', saveFigStr % (figNameEnd))
    fid = fid + 1;
    return fid

def indiviualPlotPlusRefAndSave(figTab, fid, ttoPlot, xToPlot, xrToPlot,
                 xLabelStr, yLabelStr, titleStr, saveFigStr, plotsDirectory, figNameEnd):
    figTab[fid] = plt.figure(fid);
    plt.cla(); plt.clf();
    plt.plot(ttoPlot, xToPlot, "blue");
    plt.plot(ttoPlot, xrToPlot, "red");
    plt.xlabel(xLabelStr); plt.ylabel(yLabelStr);
    plt.title(titleStr);
    plt.grid(); plt.savefig(plotsDirectory + '/' + saveFigStr % (figNameEnd), format="pdf");
    printf('Figure %s saved\n', saveFigStr % (figNameEnd))
    fid = fid + 1;
    return fid

def curvePlotPlusRefAndSave(figTab, fid, xtoPlot, yToPlot, xrToPlot, yrToPlot,
                 xLabelStr, yLabelStr, titleStr, saveFigStr, plotsDirectory, figNameEnd):
    figTab[fid] = plt.figure(fid);
    plt.cla(); plt.clf();
    plt.plot(xtoPlot, yToPlot, "blue");
    plt.plot(xrToPlot, yrToPlot, "red");
    plt.xlabel(xLabelStr); plt.ylabel(yLabelStr);
    plt.title(titleStr);
    plt.grid(); plt.savefig(plotsDirectory + '/' + saveFigStr % (figNameEnd), format="pdf");
    printf('Figure %s saved\n', saveFigStr % (figNameEnd))
    fid = fid + 1;
    return fid

def plotSaveAllIndividualPlots(X, XRef, U, F, figTab, t, plotsDirectory, figNameEnd):
  x = X[:,0]; y = X[:,1]; vPsi = X[:,2]; vPhi = X[:,3];
  [xRef, yRef] = XRef; [uPsi, uPhi] = U; [flx, fly, fluPsi, fluPhi] = F
  fid = 1;
  printf("plotSaveAllIndividualPlots(): plot x, y\n")
  # Plot the flat output
  fid = indiviualPlotPlusRefAndSave(figTab, fid, t, x, xRef, 'Time (s)',
                                    '$x_{\\normalfont\\footnotesize\\textsc{x}}$, \
                                     $x_{{\\normalfont\\footnotesize\\textsc{x}}r}$ (m)',
                                     'Abscissa, actual and reference',
                                    "x-%s.pdf", plotsDirectory, figNameEnd)
  fid = indiviualPlotPlusRefAndSave(figTab, fid, t, y, yRef, 'Time (s)',
                                    '$x_{\\normalfont\\footnotesize\\textsc{y}}$, \
                                     $x_{{\\normalfont\\footnotesize\\textsc{y}}r}$ (m)',
                                    'Ordinate, actual and reference',
                                    "y-%s.pdf", plotsDirectory, figNameEnd)
  # Plot the (x,y) and (x_r, y_r) curves
  fid = curvePlotPlusRefAndSave(figTab, fid, x, y, xRef, yRef,
                                    '$x_{\\normalfont\\footnotesize\\textsc{x}}$, \
                                    $x_{{\\normalfont\\footnotesize\\textsc{x}}r}$ (m)',
                                    '$x_{\\normalfont\\footnotesize\\textsc{y}}$, \
                                     $x_{{\\normalfont\\footnotesize\\textsc{y}}r}$ (m)',
                                    'Actual and reference curves',
                                    "XY-%s.pdf", plotsDirectory, figNameEnd)
  # Plot the rest of the state
  fid = indiviualPlotAndSave(figTab, fid, t, vPsi, 'Time (s)',
```

```
                                   '$v_\\psi$ (rad/s)', 'Angular orientation speed',
                                   "vPsi-%s.pdf", plotsDirectory, figNameEnd)
    fid = indiviualPlotAndSave(figTab, fid, t, vPhi, 'Time (s)',
                                   '$v_\\phi$ (rad/s)', 'Angular elevation speed',
                                   "vPhi-%s.pdf", plotsDirectory, figNameEnd)
    # Plot the control laws
    fid = indiviualPlotAndSave(figTab, fid, t, uPsi, 'Time (s)',
                                   '$u_\\psi$ (N/m)', 'Orientation action',
                                   "uPsi-%s.pdf",plotsDirectory, figNameEnd)
    fid = indiviualPlotAndSave(figTab, fid, t, uPhi, 'Time (s)',
                                   '$u_\\phi$ (N/m)', 'Elevation action',
                                   "uPhi-%s.pdf", plotsDirectory, figNameEnd)
    # Plot the fluctuations
    fid = indiviualPlotAndSave(figTab, fid, t, flx, 'Time (s)',
                                   '$\\zeta_x$ (m)', 'Abscissa fluctuation',
                                   "zetaX-%s.pdf", plotsDirectory, figNameEnd)
    fid = indiviualPlotAndSave(figTab, fid, t, fly, 'Time (s)',
                                   '$\\zeta_y$ (m)', 'Ordinate fluctuation',
                                   "zetaY-%s.pdf", plotsDirectory, figNameEnd)
    fid = indiviualPlotAndSave(figTab, fid, t, fluPsi, 'Time (s)',
                            '$\\zeta_{\\psi}$ (m)', 'Orientation acceleration fluctuation',
                            "zetaUPsi-%s.pdf", plotsDirectory, figNameEnd)
    fid = indiviualPlotAndSave(figTab, fid, t, fluPhi, 'Time (s)',
                            '$\\zeta_{\\phi}$ (m)', 'Elevation acceleration fluctuation',
                            "zetaUPhi-%s.pdf", plotsDirectory, figNameEnd)

def plotSaveOpenLoopZoomedPlots(t, XRefr, Ur, F, figTab, plotsDirectory, figNameEnd, fid = 1,
                           firstLast = "first&last", lineThick = 1.5, lineColor = "blue", zoomLevel = 8):
    [uPsir, uPhir] = Ur; [zetx, zety, zetupsi, zetuphi] = F; [xr, yr, vPsir, vPhir] = XRefr
    # Plot the flat output
    fid = indiviualZoomedPlotAndSave(figTab, fid, t, xr, 'Time (s)',
                                   '$x_{{\\normalfont\\footnotesize\\textsc{x}}r}$ (m)',
                                   'Reference abscissa',
                                   "xr-zoom-%s.pdf", plotsDirectory, figNameEnd, firstLast,
                                   lineThick, lineColor, zoomLevel)
    fid = indiviualZoomedPlotAndSave(figTab, fid, t, yr, 'Time (s)',
                                   '$x_{{\\normalfont\\footnotesize\\textsc{y}}r}$ (m)',
                                   'Reference ordinate',
                                   "yr-zoom-%s.pdf", plotsDirectory, figNameEnd, firstLast,
                                   lineThick, lineColor, zoomLevel)
    # Plot the reference state
    fid = indiviualZoomedPlotAndSave(figTab, fid, t, vPsir, 'Time (s)',
                               '$v_{\\psi r}$ (rad/s)', 'Reference angular orientation speed',
                               "vPsir-zoom-%s.pdf", plotsDirectory, figNameEnd, firstLast,
                               lineThick, lineColor, zoomLevel)
    fid = indiviualZoomedPlotAndSave(figTab, fid, t, vPhir, 'Time (s)',
                               '$v_{\\phi r}$ (rad/s)', 'Reference angular elevation speed',
                               "vPhir-zoom-%s.pdf", plotsDirectory, figNameEnd, firstLast,
                               lineThick, lineColor, zoomLevel)
    # Plot the control laws
    fid = indiviualZoomedPlotAndSave(figTab, fid, t, uPsir, 'Time (s)',
                               '$u_{\\psi r}$ (N/m)', 'Reference orientation action',
                               "uPsir-zoom-%s.pdf", plotsDirectory, figNameEnd, firstLast,
                               lineThick, lineColor, zoomLevel)
    fid = indiviualZoomedPlotAndSave(figTab, fid, t, uPhir, 'Time (s)',
                               '$u_{\\phi r}$ (N/m)', 'Reference elevation action',
```

```
                                  "uPhir-zoom-%s.pdf", plotsDirectory, figNameEnd, firstLast,
                                  lineThick, lineColor, zoomLevel)
    # Plot the fluctuations
    fid = indiviualZoomedPlotAndSave(figTab, fid, t, zetx, 'Time (s)',
                                          '$\\zeta_x$ (m)', 'Abscissa dynamics fluctuation',
                                          "zetx-zoom-%s.pdf", plotsDirectory, figNameEnd, firstLast,
                                          lineThick, lineColor, zoomLevel)
    fid = indiviualZoomedPlotAndSave(figTab, fid, t, zety, 'Time (s)',
                                          '$\\zeta_y$ (m)', 'Ordinate dynamics fluctuation',
                                          "zety-zoom-%s.pdf", plotsDirectory, figNameEnd, firstLast,
                                          lineThick, lineColor, zoomLevel)
    fid = indiviualZoomedPlotAndSave(figTab, fid, t, zetupsi, 'Time (s)',
                                          '$\\zeta_\\psi$ (m)', 'Orientation dynamics fluctuation',
                                          "zetupsi-zoom-%s.pdf", plotsDirectory, figNameEnd, firstLast,
                                          lineThick, lineColor, zoomLevel)
    fid = indiviualZoomedPlotAndSave(figTab, fid, t, zetuphi, 'Time (s)',
                                          '$\\zeta_\\phi$ (m)', 'Elevation dynamics fluctuation',
                                          "zetuphi-zoom-%s.pdf", plotsDirectory, figNameEnd, firstLast,
                                          lineThick, lineColor, zoomLevel)


def plotSaveOpenLoopPlots(t, XRefr, Ur, F, figTab, plotsDirectory, figNameEnd, fid = 1,
                          firstLast = "first&last", lineThick = 1.5, lineColor = "blue"):
    [uPsir, uPhir] = Ur; [zetx, zety, zetupsi, zetuphi] = F; [xr, yr, vPsir, vPhir] = XRefr
    # Plot the flat output
    fid = indiviualPlotAndSave(figTab, fid, t, xr, 'Time (s)',
                                          '$x_{{\\normalfont\\footnotesize\\textsc{x}}r}$ (m)',
                                          'Reference abscissa',
                                          "xr-%s.pdf", plotsDirectory, figNameEnd, firstLast,
                                          lineThick, lineColor)
    fid = indiviualPlotAndSave(figTab, fid, t, yr, 'Time (s)',
                                          '$x_{{\\normalfont\\footnotesize\\textsc{y}}r}$ (m)',
                                          'Reference ordinate',
                                          "yr-%s.pdf", plotsDirectory, figNameEnd, firstLast,
                                          lineThick, lineColor)
    # Plot the reference state
    fid = indiviualPlotAndSave(figTab, fid, t, vPsir, 'Time (s)',
                                  '$v_{\\psi r}$ (rad/s)', 'Reference angular orientation speed',
                                  "vPsir-%s.pdf", plotsDirectory, figNameEnd, firstLast,
                                  lineThick, lineColor)
    fid = indiviualPlotAndSave(figTab, fid, t, vPhir, 'Time (s)',
                                  '$v_{\\phi r}$ (rad/s)', 'Reference angular elevation speed',
                                  "vPhir-%s.pdf", plotsDirectory, figNameEnd, firstLast,
                                  lineThick, lineColor)
    # Plot the control laws
    fid = indiviualPlotAndSave(figTab, fid, t, uPsir, 'Time (s)',
                                  '$u_{\\psi r}$ (N/m)', 'Reference orientation action',
                                  "uPsir-%s.pdf", plotsDirectory, figNameEnd, firstLast,
                                  lineThick, lineColor)
    fid = indiviualPlotAndSave(figTab, fid, t, uPhir, 'Time (s)',
                                  '$u_{\\phi r}$ (N/m)', 'Reference elevation action',
                                  "uPhir-%s.pdf", plotsDirectory, figNameEnd, firstLast,
                                  lineThick, lineColor)
    # Plot the fluctuations
    fid = indiviualPlotAndSave(figTab, fid, t, zetx, 'Time (s)',
                                          '$\\zeta_x$ (m)', 'Abscissa dynamics fluctuation',
```

```
                                                "zetx-%s.pdf", plotsDirectory, figNameEnd, firstLast,
                                                lineThick, lineColor)
    fid = indiviualPlotAndSave(figTab, fid, t, zety, 'Time (s)',
                                                '$\\zeta_y$ (m)', 'Ordinate dynamics fluctuation',
                                                "zety-%s.pdf", plotsDirectory, figNameEnd, firstLast,
                                                lineThick, lineColor)
    fid = indiviualPlotAndSave(figTab, fid, t, zetupsi, 'Time (s)',
                                                '$\\zeta_\\psi$ (m)', 'Orientation dynamics fluctuation',
                                                "zetupsi-%s.pdf", plotsDirectory, figNameEnd, firstLast,
                                                lineThick, lineColor)
    fid = indiviualPlotAndSave(figTab, fid, t, zetuphi, 'Time (s)',
                                                '$\\zeta_\\phi$ (m)', 'Elevation dynamics fluctuation',
                                                "zetuphi-%s.pdf", plotsDirectory, figNameEnd, firstLast,
                                                lineThick, lineColor)



#####################################################
## Oculomotor dynamics function (called by odeint())
#####################################################
def oculomotorDyn(X, t, physPars, trackCtrl, fluctXYU):
    # State space components gathering
    x = X[0];  y = X[1]; vPsi = X[2]; vPhi = X[3]
    # Physical variables gathering
    me, re, Ie, d, x0, y0 = physPars.get()
    # Fluctuations computation
    zetx, dotzetx, ddotzetx, zety, dotzety, \
                ddotzety, zetupsi, zetuphi = fluctXYU.pert(t)
    # Action computation
    uPsi, uPhi       = trackCtrl.ctrl(t, X)
    # Equations of motion
    dotX   = vPsi * (d**2 + (x-x0 - zetx)**2) / d + dotzetx
    dotY   = vPhi * (d**2 + (y-y0 - zety)**2) / d + dotzety
    dotVPsi = (uPsi + zetupsi) / (2*Ie)
    dotVPhi = (uPhi + zetuphi) / Ie
    return [dotX, dotY, dotVPsi, dotVPhi]

###########################
## Simulation and plotting
###########################
def simulateAndPlot(physPars, timePars, trackCtrl,
                    refTrajX, refTrajY, fluctXYU, plotsDirectory, figNameEnd, saveFigures):
    # Simulation
    [tini, tend, dt, xi, yi, vPsii, vPhii] = timePars.get()
    t = np.arange(tini,tend,dt)
    Xi = [xi, yi, vPsii, vPhii]
    X = spy.odeint(oculomotorDyn, Xi, t,
                   args = (physPars, trackCtrl, fluctXYU))
    # Results plotting
    uPsi, uPhi = trackCtrl.ctrl(t, X);
    U = [uPsi, uPhi]
    xr, dotxr, ddotxr = refTrajX.traj(t)
    yr, dotyr, ddotyr = refTrajY.traj(t)
    zetx, dotzetx, ddotzetx, zety, dotzety, \
                    ddotzety, zetupsi, zetuphi = fluctXYU.pert(t)
    F = [zetx, zety, zetupsi, zetuphi]
```

```
    XRef = [xr, yr]
    if (saveFigures == True):
        figTab = [plt.Figure() for i in range(100)];
        fid = 1;
        plotSaveAllIndividualPlots(X, XRef, U, F, figTab, t, plotsDirectory, figNameEnd)
    else:
        oculomotorPlot(t, X, xr, yr, uPsi, uPhi, fluctXYU)
    return

def computeOpenLoopAndPlot(physPars, timePars, openLoopCtrl, refTrajX,
                           refTrajY, fluctXYU, plotsDirectory, figNameEnd, saveFigures):
    # Simulation
    [tini, tend, dt, xi, yi, vPsii, vPhii] = timePars.get()
    t = np.arange(tini,tend,dt)
    # Open loop computation
    xr, dotxr, ddotxr = refTrajX.traj(t)
    yr, dotyr, ddotyr = refTrajY.traj(t)
    vPsir, vPhir, uPsir, uPhir = openLoopCtrl.ctrl(t);
    zetx, dotzetx, ddotzetx, zety, dotzety, \
                ddotzety, zetupsi, zetuphi = fluctXYU.pert(t)
    # Results gathering
    Ur    = [uPsir, uPhir]
    F     = [zetx, zety, zetupsi, zetuphi]
    XRefr = [xr, yr, vPsir, vPhir]
    # Results plotting
    if (saveFigures == True):
        figTab = [plt.Figure() for i in range(100)];
        fid = 1;
        plotSaveOpenLoopPlots(t, XRefr, Ur, F, figTab, plotsDirectory, figNameEnd)
    else:
        openLoopPlot(t, XRefr, Ur, F)
    return


def computeOpenLoopPlusFluctAndPlot(physPars, timePars, refTrajX,
                           refTrajY, fluctXYUList, plotsDirectory, figNameEnd, saveFigures):
    # Simulation
    [tini, tend, dt, xi, yi, vPsii, vPhii] = timePars.get()
    t = np.arange(tini,tend,dt)
    figTab = [plt.Figure() for i in range(100)];
    figTabZoom = [plt.Figure() for i in range(200)];
    fid = 1; fidZoom = 101
    # Open loop computation
    xr, dotxr, ddotxr = refTrajX.traj(t); yr, dotyr, ddotyr = refTrajY.traj(t)
    # First plot with fluctuations
    firstLast = "first"; thickness = 0.7; color = "gray"
    zetx, dotzetx, ddotzetx, zety, dotzety, \
                ddotzety, zetupsi, zetuphi = fluctXYUList[0].pert(t)
    openLoopCtrlFluct = OpenLoopCtrlFlat(physPars, refTrajX, refTrajY, fluctXYUList[0])
    vPsir, vPhir, uPsir, uPhir = openLoopCtrlFluct.ctrl(t);
    XRefr = [xr, yr, vPsir, vPhir]
    Ur    = [uPsir, uPhir]; F = [zetx, zety, zetupsi, zetuphi];  XRefr = [xr, yr, vPsir, vPhir]
    if (saveFigures == True):
        zoomlevel = 8
        plotSaveOpenLoopPlots(t, XRefr, Ur, F, figTab, plotsDirectory,
                              figNameEnd, fid, firstLast, thickness, color)
```

```
        plotSaveOpenLoopZoomedPlots(t, XRefr, Ur, F, figTabZoom, plotsDirectory,
                                    figNameEnd, fidZoom, firstLast, thickness, color, zoomlevel)
    else:
        openLoopPlot(t, XRefr, Ur, F)
    # All plots with fluctuations
    firstLast = "middle"; thickness = 0.7; color = "gray"
    for i in range(1, len(fluctXYUList)):
        zetx, dotzetx, ddotzetx, zety, dotzety, \
                ddotzety, zetupsi, zetuphi = fluctXYUList[i].pert(t)
        openLoopCtrlFluct = OpenLoopCtrlFlat(physPars, refTrajX, refTrajY, fluctXYUList[i])
        vPsir, vPhir, uPsir, uPhir = openLoopCtrlFluct.ctrl(t);
        Ur    = [uPsir, uPhir]; F = [zetx, zety, zetupsi, zetuphi]; XRefr = [xr, yr, vPsir, vPhir]
        if (saveFigures == True):
            plotSaveOpenLoopPlots(t, XRefr, Ur, F, figTab, plotsDirectory,
                                  figNameEnd, fid, firstLast, thickness, color)
            plotSaveOpenLoopZoomedPlots(t, XRefr, Ur, F, figTabZoom, plotsDirectory,
                                        figNameEnd, fidZoom, firstLast, thickness, color)
        else:
            openLoopPlot(t, XRefr, Ur, F, figTab)
    # Last plot: open loop without fluctiations
    zeroTraj = ZeroTraj()
    fluctXYUZero = PertXYU(zeroTraj, zeroTraj, zeroTraj, zeroTraj)
    openLoopCtrlNoFluct = OpenLoopCtrlFlat(physPars, refTrajX, refTrajY, fluctXYUZero)
    zetx, dotzetx, ddotzetx, zety, dotzety, \
                ddotzety, zetupsi, zetuphi = fluctXYUZero.pert(t)
    vPsir, vPhir, uPsir, uPhir = openLoopCtrlFluct.ctrl(t);
    Ur    = [uPsir, uPhir]; F0 = [zetx, zety, zetupsi, zetuphi];  XRefr = [xr, yr, vPsir, vPhir]
    firstLast = "last"; thickness = 2; color = "red"
    if (saveFigures == True):
        plotSaveOpenLoopPlots(t, XRefr, Ur, F0, figTab, plotsDirectory,
                              figNameEnd, fid, firstLast, thickness, color)
        plotSaveOpenLoopZoomedPlots(t, XRefr, Ur, F, figTabZoom, plotsDirectory,
                                    figNameEnd, fidZoom, firstLast, thickness, color)
    else:
        openLoopPlot(t, XRefr, Ur, F, figTab)
    return


def createFluctuationTrajs(trajType, timePars):
    [tini, tend, dt, xi, yi, vPsii, vPhii] = timePars.get()
    if (trajType == 'tanhHatTr'):
        lVal = 0; hValX = 0.2; hValY = 0.2; hValUPsi = 0.2; hValUPhi = 0.2; # trajPars
        stR = 4; stD = stR;
        tRaise = tini + (tend-tini)/8; tGoDown = tRaise + (tend-tini)/4
        fluctTrajXPars = TanhHatRefTrajPars(lVal, hValX, stR, stD, tRaise, tGoDown, "abscissa x")
        fluctTrajYPars = TanhHatRefTrajPars(lVal, hValY, stR, stD, tRaise, tGoDown, "ordinate y")
        fluctTrajUPsiPars = TanhHatRefTrajPars(lVal, hValUPsi, stR, stD, tRaise, tGoDown, "action $u_\\psi$")
        fluctTrajUPhiPars = TanhHatRefTrajPars(lVal, hValUPhi, stR, stD, tRaise, tGoDown, "action $u_\\phi$")
        fluctTrajX    = TanhHatRefTraj(lVal, hValX, stR, stD, tRaise, tGoDown)
        fluctTrajY    = TanhHatRefTraj(lVal, hValY, stR, stD, tRaise, tGoDown)
        fluctTrajUPsi = TanhHatRefTraj(lVal, hValUPsi, stR, stD, tRaise, tGoDown)
        fluctTrajUPhi = TanhHatRefTraj(lVal, hValUPhi, stR, stD, tRaise, tGoDown)
    elif (trajType == 'zero'):
        fluctTrajXPars = ZeroTrajPars; fluctTrajYPars = ZeroTrajPars;
        fluctTrajUPsiPars = ZeroTrajPars; fluctTrajUPhiPars = ZeroTrajPars;
```

```
        fluctTrajX = ZeroTraj; fluctTrajY = ZeroTraj;
        fluctTrajUPsi = ZeroTraj; fluctTrajUPhi = ZeroTraj;
    #elif ('smoothRandomFun' in trajType):
    elif (trajType.find('SRF') != -1):
        # default parameter values
        L = 8; lmbda = 2; scale = 1e-5;    # fluctPars
        match trajType:
            case s if 'SRFTiSc' in s:  # Smmoth Random Function Tiny Scale
            # r = floor(L/lmbda) - 2*pi*j/L = N*2*pi <=> L = j/N ; j = 1, ..., r
                L = 8; lmbda = 2; scale = 2e-6;    # fluctPars
            case s if 'SRFSmSc' in s:  # Smmoth Random Function Small Scale
                L = 8; lmbda = 2; scale = 1e-5;    # fluctPars
            case s if 'SRFMdSc' in s:  # Smmoth Random Function Medium Scale
                L = 8; lmbda = 2; scale = 1e-4;    # fluctPars
            case s if 'SRFBgSc' in s:    # Smmoth Random Function Big Scale
                L = 8; lmbda = 2; scale = 1e-3;    # fluctPars
            case s if 'SRFVBgSc' in s:  # Smmoth Random Function Very Big Scale
                L = 8; lmbda = 2; scale = 1e-1;    # fluctPars
            case s if 'SRFHgSc' in s:    # Smmoth Random Function Huge Scale
                L = 8; lmbda = 2; scale = 10;    # fluctPars
            case default:
                print(trajType)
                print('No matching fluctuation type; default values taken')
#        L = 0.5; lmbda = 0.25; scale = 0.5; # Horrible
        fluctTrajXPars = FluctSmoothRandFunPars(L, lmbda, scale, "abscissa x");
        fluctTrajYPars = FluctSmoothRandFunPars(L, lmbda, scale, "ordinate y");
        fluctTrajUPsiPars = FluctSmoothRandFunPars(L, lmbda, scale, "action $u_\\psi$");
        fluctTrajUPhiPars = FluctSmoothRandFunPars(L, lmbda, scale, "action $u_\\phi$");
        fluctTrajX    = SmoothRandFunTraj(L, lmbda, scale)
        fluctTrajY    = SmoothRandFunTraj(L, lmbda, scale)
        fluctTrajUPsi = SmoothRandFunTraj(L, lmbda, scale)
        fluctTrajUPhi = SmoothRandFunTraj(L, lmbda, scale)
    else:
        print('Perturbation trajectory type unknown')
        return
    pertXYU = PertXYU(fluctTrajX, fluctTrajY, fluctTrajUPsi, fluctTrajUPhi)
    pertXYUPars = PertXYUPars(fluctTrajXPars, fluctTrajYPars, fluctTrajUPsiPars, fluctTrajUPhiPars)
    return [pertXYU, pertXYUPars]

# create the reference trajectories
def createReferenceTrajs(refTrajType, timePars):
    [tini, tend, dt, xi, yi, vPsii, vPhii] = timePars.get()
    if (refTrajType == 'tanhTr'):
        tRaise = tini + (tend-tini)/4; lVal = 1; hVal = 4; st = 0.2;
        refTrajXPars = TanhRefTrajPars(hVal, lVal, st, tRaise, "abscissa x")
        refTrajYPars = TanhRefTrajPars(hVal, lVal, st, tRaise, "ordinate y")
        refTrajX = TanhRefTraj(lVal, hVal, st, tRaise);
        refTrajY = TanhRefTraj(lVal, hVal, st, tRaise);
    elif (refTrajType == 'quaterfoil'):
        a = 2;
        refTrajXPars = QuaterfoilTrajPars(a, "abscissa x")
        refTrajYPars = QuaterfoilTrajPars(a, "ordinate y")
        refTrajX = QuaterfoilTrajX(a);
        refTrajY = QuaterfoilTrajY(a);
    elif (refTrajType == 'hypocycloid'):
        a = 2; b = 3*a/5
```

```
        refTrajXPars = HypocycloidTrajPars(a, b, "abscissa x")
        refTrajYPars = HypocycloidTrajPars(a, b, "ordinate y")
        refTrajX = HypocycloidTrajX(a, b);
        refTrajY = HypocycloidTrajY(a, b);
    elif (refTrajType == 'hypotrochoid'):
        a = 5; b = 3; c = 3.5;
        refTrajXPars = HypotrochoidTrajPars(a, b, c, "abscissa x")
        refTrajYPars = HypotrochoidTrajPars(a, b, c, "ordinate y")
        refTrajX = HypotrochoidTrajX(a, b, c);
        refTrajY = HypotrochoidTrajY(a, b, c);
    else:
        print('Reference trajectory type unknown')
    return [refTrajXPars, refTrajYPars, refTrajX, refTrajY]

# Create the reference trajectories, the fluctuations and the action feedback law
def createObjects(refTrajType, refPertType, ctrlType):
    # Physical parameters
    physPars = PhysPars()
    # Time, initial state and refs trajectories
    tini = 0; dt = 0.01;
    xi = 0.1; yi = 0.2; vPsii = 0.; vPhii = 0.;
    if (refTrajType == 'tanhTr'):
        tend = 80;
    elif (refTrajType == 'quarterfoil'):
        tend = 2*np.pi;
    elif (refTrajType == 'hypocycloid'):
        tend = 6*np.pi;
    elif (refTrajType == 'hypotrochoid'):
        tend = 6*np.pi;
    else:
        tend = 10;
    timePars = TimePars(tini, tend, dt, xi, yi, vPsii, vPhii);
    # Reference trajectories
    [refTrajXPars, refTrajYPars, refTrajX, refTrajY] = createReferenceTrajs(refTrajType, timePars)
    # Fluctuations
    pertXYU, pertXYUPars = createFluctuationTrajs(refPertType, timePars)
    fluctTrajXPars, fluctTrajYPars, fluctTrajUPsiPars, fluctTrajUPhiPars = pertXYUPars.get()
    # Gains and elementary tracking control laws
    gainx = 120; gainy = 120;
    gainPars = GainPars(gainx, gainx, gainy, gainy)
    elemTrackCtrlX = TrackCtrlPD(gainx, gainx)
    elemTrackCtrlY = TrackCtrlPD(gainy, gainy)
    # Tracking control law
    if (ctrlType == 'flatCtrlLaw'):
        trackCtrl = TrackCtrlFlat(physPars, refTrajX, refTrajY, pertXYU,
                                  elemTrackCtrlX, elemTrackCtrlY)
    elif (ctrlType == 'blindFlatCtrlLaw'):
        trackCtrl = TrackCtrlFlatBlind(physPars, refTrajX, refTrajY, pertXYU,
                                  elemTrackCtrlX, elemTrackCtrlY)
    else:
        print('CtrlType undefined')
    allPars = AllPars(physPars, timePars, gainPars, refTrajXPars, refTrajYPars, fluctTrajXPars,
                        fluctTrajYPars, fluctTrajUPsiPars, fluctTrajUPhiPars)
    return [allPars, trackCtrl, refTrajX, refTrajY, pertXYU]
```

```python
def performCompleteSimulation(whatToDo, refTrajType, refFluctType, ctrlType, saveFigures):
    # directory and file names $$$
    plotsDirectory = './PlotsFromSimulations'
    if not path.exists(plotsDirectory):
        makedirs('./PlotsFromSimulations')
    # today = datetime.now(); dateHour = today.strftime("%Y-%b-%d-%Hh-%Mmin-%Ssec")
    figNameEnd   = 'oculom-' +  whatToDo + '-' + refTrajType + '-' + refFluctType # + '-' + dateHour
    paramsFilename = plotsDirectory + '/pars-' + figNameEnd + ".txt"

    # create reference trajectories, fluctuations
    [allPars, trackCtrl, refTrajX, refTrajY, fluctXYU] = createObjects(refTrajType, refFluctType, ctrlType);
    [physPars, timePars, gainPars, refTrajXPars, refTrajYPars, fluctTrajXPars,
                        fluctTrajYPars, fluctTrajUPsiPars, fluctTrajUPhiPars] = allPars.get()
    allPars.writeOnFile(paramsFilename)

    if ('simAndPlot' in whatToDo):
        # simulate the model and plot the results
        simulateAndPlot(physPars, timePars, trackCtrl,
                        refTrajX, refTrajY, fluctXYU, plotsDirectory, figNameEnd, saveFigures)
    elif ('compOLPlFlAndPlot' in whatToDo):
        fluctXYUList = []; nbFluctTrajs = 12
        for i in range(1, nbFluctTrajs + 1):
            pertXYUi, pertXYUParsi = createFluctuationTrajs(refFluctType, timePars)
            fluctXYUList.append(pertXYUi)
        computeOpenLoopPlusFluctAndPlot(physPars, timePars, refTrajX,
                                        refTrajY, fluctXYUList, plotsDirectory, figNameEnd, saveFigures)
    elif ('compOLAndPlot' in whatToDo):
        openLoopCtrl = OpenLoopCtrlFlat(physPars, refTrajX, refTrajY, fluctXYU)
        computeOpenLoopAndPlot(physPars, timePars, openLoopCtrl, refTrajX,
                               refTrajY, fluctXYU, plotsDirectory, figNameEnd, saveFigures)


def main():
    # reference traj, fluctuation and action law choice
    saveFigures  = True                # Save or plot
    taskId = 1
    # SRF: Smooth Random Function
    ctrlType = 'flatCtrlLaw'; refTrajType  = 'tanhTr';
    # Open loop control computation and plot, no fluctuation
    whatToDo = 'Task-' + str(taskId) + '-compOLAndPlot';  refFluctType = 'SRFSmSc';   # Small scale
    performCompleteSimulation(whatToDo, refTrajType, refFluctType, ctrlType, saveFigures)
    taskId = taskId + 1
    # Open loop control computation and plot, tiny fluctuation
    whatToDo = 'Task-' + str(taskId) + '-compOLPlFlAndPlot';  refFluctType = 'SRFTiSc';  # Tiny scale
    performCompleteSimulation(whatToDo, refTrajType, refFluctType, ctrlType, saveFigures)
    taskId = taskId + 1
    # Open loop control computation and plot, small fluctuation
    whatToDo = 'Task-' + str(taskId) + '-compOLPlFlAndPlot';  refFluctType = 'SRFSmSc';
    performCompleteSimulation(whatToDo, refTrajType, refFluctType, ctrlType, saveFigures)
    taskId = taskId + 1
    # Open loop control computation and plot, medium fluctuation
    whatToDo = 'Task-' + str(taskId) + '-compOLPlFlAndPlot'; refFluctType = 'SRFMdSc';  # Medium scale
    performCompleteSimulation(whatToDo, refTrajType, refFluctType, ctrlType, saveFigures)
    taskId = taskId + 1
    # 1st simulation
    refTrajType  = 'quaterfoil'; refFluctType = 'SRFSmSc';
```

```
    whatToDo = 'Task-' + str(taskId) + '-simAndPlot'
    performCompleteSimulation(whatToDo, refTrajType, refFluctType, ctrlType, saveFigures)
    taskId = taskId + 1
    # 2nd simulation
    refTrajType  = 'hypotrochoid'; refFluctType = 'SRFSmSc';
    whatToDo = 'Task-' + str(taskId) + '-simAndPlot'
    performCompleteSimulation(whatToDo, refTrajType, refFluctType, ctrlType, saveFigures)
    taskId = taskId + 1
    # 3rd simulation
    refTrajType  = 'hypocycloid'; refFluctType = 'SRFSmSc';
    whatToDo = 'Task-' + str(taskId) + '-simAndPlot'
    performCompleteSimulation(whatToDo, refTrajType, refFluctType, ctrlType, saveFigures)
    taskId = taskId + 1
    # 4th simulation
    refTrajType  = 'quaterfoil'; refFluctType = 'SRFHgSc'; # Huge scale
    whatToDo = 'Task-' + str(taskId) + '-simAndPlot'
    performCompleteSimulation(whatToDo, refTrajType, refFluctType, ctrlType, saveFigures)
    taskId = taskId + 1
#     ctrlType     = 'blindFlatCtrlLaw' # scale 1e-7 (1e-6 is too big)
    return

main()
```