Globals.py

import random

from mpl\_toolkits import mplot3d

from matplotlib import patches

from matplotlib import cm

from matplotlib.colors import ListedColormap

from mpl\_toolkits.axes\_grid1 import make\_axes\_locatable

import visvis as vv

import autograd.numpy as np

from matplotlib.pyplot import ion, draw, Rectangle, Line2D

import matplotlib.pyplot as plt

import os

plt.rcParams["savefig.directory"] = os.chdir(os.path.dirname(\_\_file\_\_) + "/figs")

Cartpole.py

"""

fork from python-rl and pybrain for visualization

"""

from globals import \*

#import numpy as np

import autograd.numpy as np

from matplotlib.pyplot import ion, draw, Rectangle, Line2D

import matplotlib.pyplot as plt

# If theta has gone past our conceptual limits of [-pi,pi]

# map it onto the equivalent angle that is in the accepted range (by adding or subtracting 2pi)

def remap\_angle(theta):

return \_remap\_angle(theta)

remap\_angle\_v = np.vectorize(remap\_angle)

def \_remap\_angle(theta):

while theta < -np.pi:

theta += 2. \* np.pi

while theta > np.pi:

theta -= 2. \* np.pi

return theta

## loss function given a state vector. the elements of the state vector are

## [cart location, cart velocity, pole angle, pole angular velocity]

def \_loss(state):

sig = 0.5

return 1-np.exp(-np.dot(state,state)/(2.0 \* sig\*\*2))

def loss(state):

return \_loss(state)

class CartPole:

"""Cart Pole environment. This implementation allows multiple poles,

noisy action, and random starts. It has been checked repeatedly for

'correctness', specifically the direction of gravity. Some implementations of

cart pole on the internet have the gravity constant inverted. The way to check is to

limit the force to be zero, start from a valid random start state and watch how long

it takes for the pole to fall. If the pole falls almost immediately, you're all set. If it takes

tens or hundreds of steps then you have gravity inverted. It will tend to still fall because

of round off errors that cause the oscillations to grow until it eventually falls.

"""

def \_\_init\_\_(self, delta\_time=.2, visual=False):

self.cart\_location = 0.0

self.cart\_velocity = 0.0

self.pole\_angle = np.pi # angle is defined to be zero when the pole is upright, pi when hanging vertically down

self.pole\_velocity = 0.0

self.visual = visual

# Setup pole lengths and masses based on scale of each pole

# (Papers using multi-poles tend to have them either same lengths/masses

# or they vary by some scalar from the other poles)

self.pole\_length = 0.5

self.pole\_mass = 0.5

self.frictionless = False

self.mu\_c = 0.001 # # friction coefficient of the cart

self.mu\_p = 0.001 # # friction coefficient of the pole

if self.frictionless:

self.mu\_c = 0

self.mu\_p = 0

self.sim\_steps = 50 #50 # number of Euler integration steps to perform in one go

self.delta\_time = delta\_time #.2 # time step of the Euler integrator

self.max\_force = 20.

self.gravity = 9.8

self.cart\_mass = 0.5

# for plotting

self.cartwidth = 1.0

self.cartheight = 0.2

if self.visual:

self.drawPlot()

def setState(self, state):

self.cart\_location = state[0]

self.cart\_velocity = state[1]

self.pole\_angle = state[2]

self.pole\_velocity = state[3]

def getEnergy(self):

state = self.getState()

V = 0.5 \* self.pole\_length \* self.gravity \* self.pole\_mass \* (np.cos(state[2]) - 1)

T = 0.5 \* (self.cart\_mass + self.pole\_mass) \* state[1]\*\*2

T += 0.5 \* self.pole\_mass \* self.pole\_length \* state[3] \* state[1] \* np.cos(state[2])

T += (0.5 \* self.pole\_mass / 3) \* (self.pole\_length \* state[3])\*\*2

return [T, V]

def getState(self, energy=False):

if not energy:

return np.array([self.cart\_location,self.cart\_velocity,self.pole\_angle,self.pole\_velocity])

T, V = self.getEnergy()

return np.array([self.cart\_location,self.cart\_velocity,self.pole\_angle,self.pole\_velocity, T, V])

# reset the state vector to the initial state (down-hanging pole)

def reset(self):

self.cart\_location = 0.0

self.cart\_velocity = 0.0

self.pole\_angle = np.pi

self.pole\_velocity = 0.0

# This is where the equations of motion are implemented

def performAction(self, action = 0.0):

# prevent the force from being too large

force = self.max\_force \* np.tanh(action/self.max\_force)

# integrate forward the equations of motion using the Euler method

for step in range(self.sim\_steps):

s = np.sin(self.pole\_angle)

c = np.cos(self.pole\_angle)

m = 4.0\*(self.cart\_mass+self.pole\_mass)-3.0\*self.pole\_mass\*(c\*\*2)

cart\_accel = (2.0\*(self.pole\_length\*self.pole\_mass\*(self.pole\_velocity\*\*2)\*s+2\*(force-self.mu\_c\*self.cart\_velocity))\

-3.0\*self.pole\_mass\*self.gravity\*c\*s + 6.0\*self.mu\_p\*self.pole\_velocity\*c/self.pole\_length)/m

pole\_accel = (-3.0\*c\*2.0/self.pole\_length\*(self.pole\_length/2.0\*self.pole\_mass\*(self.pole\_velocity\*\*2)\*s + force-self.mu\_c\*self.cart\_velocity)+\

6.0\*(self.cart\_mass+self.pole\_mass)/(self.pole\_mass\*self.pole\_length)\*\

(self.pole\_mass\*self.gravity\*s - 2.0/self.pole\_length\*self.mu\_p\*self.pole\_velocity) \

)/m

# Update state variables

dt = (self.delta\_time / float(self.sim\_steps))

# Do the updates in this order, so that we get semi-implicit Euler that is simplectic rather than forward-Euler which is not.

self.cart\_velocity += dt \* cart\_accel

self.pole\_velocity += dt \* pole\_accel

self.pole\_angle += dt \* self.pole\_velocity

self.cart\_location += dt \* self.cart\_velocity

if self.visual:

self.\_render()

# remapping as a member function

def remap\_angle(self):

self.pole\_angle = \_remap\_angle(self.pole\_angle)

# the loss function that the policy will try to optimise (lower) as a member function

def loss(self):

return \_loss(self.getState())

# def terminate(self):

# """Indicates whether or not the episode should terminate.

# Returns:

# A boolean, true indicating the end of an episode and false indicating the episode should continue.

# False is returned if either the cart location or

# the pole angle is beyond the allowed range.

# """

# return np.abs(self.cart\_location) > self.state\_range[0, 1] or \

# (np.abs(self.pole\_angle) > self.state\_range[2, 1]).any()

# the following are graphics routines

def drawPlot(self):

ion()

self.fig = plt.figure()

# draw cart

self.axes = self.fig.add\_subplot(111, aspect='equal')

self.box = Rectangle(xy=(self.cart\_location - self.cartwidth / 2.0, -self.cartheight),

width=self.cartwidth, height=self.cartheight)

self.axes.add\_artist(self.box)

self.box.set\_clip\_box(self.axes.bbox)

# draw pole

self.pole = Line2D([self.cart\_location, self.cart\_location + np.sin(self.pole\_angle)],

[0, np.cos(self.pole\_angle)], linewidth=3, color='black')

self.axes.add\_artist(self.pole)

self.pole.set\_clip\_box(self.axes.bbox)

# set axes limits

self.axes.set\_xlim(-10, 10)

self.axes.set\_ylim(-0.5, 2)

def \_render(self):

self.box.set\_x(self.cart\_location - self.cartwidth / 2.0)

self.pole.set\_xdata([self.cart\_location, self.cart\_location + np.sin(self.pole\_angle)])

self.pole.set\_ydata([0, np.cos(self.pole\_angle)])

self.fig.show()

plt.pause(0.05)

class Pendulum:

"""Cart Pole environment. This implementation allows multiple poles,

noisy action, and random starts. It has been checked repeatedly for

'correctness', specifically the direction of gravity. Some implementations of

cart pole on the internet have the gravity constant inverted. The way to check is to

limit the force to be zero, start from a valid random start state and watch how long

it takes for the pole to fall. If the pole falls almost immediately, you're all set. If it takes

tens or hundreds of steps then you have gravity inverted. It will tend to still fall because

of round off errors that cause the oscillations to grow until it eventually falls.

"""

def \_\_init\_\_(self, delta\_time=.2, visual=False):

self.pole\_angle = np.pi # angle is defined to be zero when the pole is upright, pi when hanging vertically down

self.pole\_velocity = 0.0

self.visual = visual

# Setup pole lengths and masses based on scale of each pole

# (Papers using multi-poles tend to have them either same lengths/masses

# or they vary by some scalar from the other poles)

self.pole\_length = 0.5

self.pole\_mass = 0.5

self.mu\_p = 0.001 # # friction coefficient of the pole

self.sim\_steps = 50 #50 # number of Euler integration steps to perform in one go

self.delta\_time = delta\_time #.2 # time step of the Euler integrator

self.max\_force = 20.

self.gravity = 9.8

# for plotting

self.cartwidth = 1.0

self.cartheight = 0.2

if self.visual:

self.drawPlot()

def setState(self, state):

self.pole\_angle = state[0]

self.pole\_velocity = state[1]

def getState(self):

return np.array([self.pole\_angle,self.pole\_velocity])

def getEnergy(self):

state = self.getState()

V = 0.5 \* self.pole\_length \* self.gravity \* self.pole\_mass \* (np.cos(state[2]) - 1)

T += (self.pole\_mass / 3) \* (self.pole\_length \* state[3])\*\*2

return [T, V]

# reset the state vector to the initial state (down-hanging pole)

def reset(self):

self.pole\_angle = np.pi

self.pole\_velocity = 0.0

# This is where the equations of motion are implemented

def performAction(self, action = 0.0):

# prevent the force from being too large

force = self.max\_force \* np.tanh(action/self.max\_force)

# integrate forward the equations of motion using the Euler method

for step in range(self.sim\_steps):

s = np.sin(self.pole\_angle)

c = np.cos(self.pole\_angle)

m = 4.0\*(self.cart\_mass+self.pole\_mass)-3.0\*self.pole\_mass\*(c\*\*2)

cart\_accel = (2.0\*(self.pole\_length\*self.pole\_mass\*(self.pole\_velocity\*\*2)\*s+2\*(force-self.mu\_c\*self.cart\_velocity))\

-3.0\*self.pole\_mass\*self.gravity\*c\*s + 6.0\*self.mu\_p\*self.pole\_velocity\*c/self.pole\_length)/m

pole\_accel = (-3.0\*c\*2.0/self.pole\_length\*(self.pole\_length/2.0\*self.pole\_mass\*(self.pole\_velocity\*\*2)\*s + force-self.mu\_c\*self.cart\_velocity)+\

6.0\*(self.cart\_mass+self.pole\_mass)/(self.pole\_mass\*self.pole\_length)\*\

(self.pole\_mass\*self.gravity\*s - 2.0/self.pole\_length\*self.mu\_p\*self.pole\_velocity) \

)/m

# Update state variables

dt = (self.delta\_time / float(self.sim\_steps))

# Do the updates in this order, so that we get semi-implicit Euler that is simplectic rather than forward-Euler which is not.

self.cart\_velocity += dt \* cart\_accel

self.pole\_velocity += dt \* pole\_accel

self.pole\_angle += dt \* self.pole\_velocity

self.cart\_location += dt \* self.cart\_velocity

if self.visual:

self.\_render()

# remapping as a member function

def remap\_angle(self):

self.pole\_angle = \_remap\_angle(self.pole\_angle)

# the loss function that the policy will try to optimise (lower) as a member function

def loss(self):

return \_loss(self.getState())

# def terminate(self):

# """Indicates whether or not the episode should terminate.

# Returns:

# A boolean, true indicating the end of an episode and false indicating the episode should continue.

# False is returned if either the cart location or

# the pole angle is beyond the allowed range.

# """

# return np.abs(self.cart\_location) > self.state\_range[0, 1] or \

# (np.abs(self.pole\_angle) > self.state\_range[2, 1]).any()

# the following are graphics routines

def drawPlot(self):

ion()

self.fig = plt.figure()

# draw cart

self.axes = self.fig.add\_subplot(111, aspect='equal')

self.box = Rectangle(xy=(self.cart\_location - self.cartwidth / 2.0, -self.cartheight),

width=self.cartwidth, height=self.cartheight)

self.axes.add\_artist(self.box)

self.box.set\_clip\_box(self.axes.bbox)

# draw pole

self.pole = Line2D([self.cart\_location, self.cart\_location + np.sin(self.pole\_angle)],

[0, np.cos(self.pole\_angle)], linewidth=3, color='black')

self.axes.add\_artist(self.pole)

self.pole.set\_clip\_box(self.axes.bbox)

# set axes limits

self.axes.set\_xlim(-10, 10)

self.axes.set\_ylim(-0.5, 2)

def \_render(self):

self.box.set\_x(self.cart\_location - self.cartwidth / 2.0)

self.pole.set\_xdata([self.cart\_location, self.cart\_location + np.sin(self.pole\_angle)])

self.pole.set\_ydata([0, np.cos(self.pole\_angle)])

self.fig.show()

plt.pause(0.05)

utils.py

from globals import \*

# import sobol\_seq

# from os import urandom

# seed = urandom(16)

# seed = 0

# for i in range(10):

# vec, seed = sobol\_seq.i4\_sobol(4, seed)

# print(vec)

P\_RANGE = np.array([15, 10, np.pi, 15])

P\_BOUNDS = np.ones((4, 2))

P\_BOUNDS \*= P\_RANGE[:, np.newaxis]

def rand\_state(bounds=None):

if bounds is None:

bounds = [15, 10, np.pi, 15]

bounds = np.array(bounds)

state = np.random.random(4) \* 2 - 1

return state \* bounds

VAR\_STR = [r"$x$", r"$\dot{x}$", r"$\theta$", r"$\dot{\theta}$"]

# https://stackoverflow.com/questions/40642061/how-to-set-axis-ticks-in-multiples-of-pi-python-matplotlib

def multiple\_formatter(denominator=2, number=np.pi, latex='\pi'):

def gcd(a, b):

while b:

a, b = b, a%b

return a

def \_multiple\_formatter(x, pos):

den = denominator

num = np.int(np.rint(den\*x/number))

com = gcd(num,den)

(num,den) = (int(num/com),int(den/com))

if den==1:

if num==0:

return r'$0$'

if num==1:

return r'$%s$'%latex

elif num==-1:

return r'$-%s$'%latex

else:

return r'$%s%s$'%(num,latex)

else:

if num==1:

return r'$\frac{%s}{%s}$'%(latex,den)

elif num==-1:

return r'$-\frac{%s}{%s}$'%(latex,den)

else:

return r'$\frac{%s%s}{%s}$'%(num,latex,den)

return \_multiple\_formatter

class Multiple:

def \_\_init\_\_(self, denominator=2, number=np.pi, latex='\pi'):

self.denominator = denominator

self.number = number

self.latex = latex

def locator(self):

return plt.MultipleLocator(self.number / self.denominator)

def formatter(self):

return plt.FuncFormatter(multiple\_formatter(self.denominator, self.number, self.latex))

def axis\_pi\_multiples(axis\_obj):

axis\_obj.set\_major\_locator(plt.MultipleLocator(np.pi / 2))

axis\_obj.set\_minor\_locator(plt.MultipleLocator(np.pi / 12))

axis\_obj.set\_major\_formatter(plt.FuncFormatter(multiple\_formatter()))

rollouts.py

from globals import \*

from model import \*

from utils import \*

def plot\_energy():

sys = CartPole(0.02)

sys.setState([0, 0, 0.2, 0])

states = []

Es = []

T = [x for x in range(100)]

for t in T:

print(sum(sys.getEnergy()))

sys.performAction(0.)

states.append(sys.getState())

Es.append(sys.getEnergy())

plt.plot(T, [x[0] for x in Es], label="T")

plt.plot(T, [x[1] for x in Es], label="V")

plt.plot(T, [sum(x) for x in Es], label="T+V")

plt.plot(T, [x[0] for x in states], label="x")

plt.plot(T, [x[2] for x in states], label="theta")

plt.legend()

plt.show()

def sigmoid(z):

return 1/(1 + np.exp(-z))

def format\_IC(IC):

if IC[2] == np.pi:

IC[2] = r"$\pi$"

if IC[2] == -np.pi:

IC[2] = r"$-\pi$"

if IC[2] == np.pi/2:

IC[2] = r"$-\pi/2$"

if IC[2] == -np.pi/2:

IC[2] = r"$-\pi/2$"

return f"[{IC[0]}, {IC[1]}, {IC[2]}, {IC[3]}]"

def f1\_IC(rollout\_fn, IC, N=200, remap=False):

for j in range(4):

for t\_step in [0.02, 0.2]:

N\_steps = N if t\_step==0.02 else int(round(0.1\*N))

T = np.arange(N\_steps)\*t\_step

states = rollout\_fn(IC, N\_steps, t\_step)

if remap:

remapped = remap\_angle\_v(states[:,2])

for i in range(1, len(T)):

if remapped[i] < -(np.pi-1) and remapped[i-1] > (np.pi-1):

remapped[i] = np.nan

elif remapped[i] > (np.pi-1) and remapped[i-1] < -(np.pi-1):

remapped[i] = np.nan

states[:,2] = remapped

if t\_step == 0.02:

p = plt.plot(T, states[:,j], label=VAR\_STR[j], lw=1.5)

else:

plt.plot(T, states[:,j], ls="--", lw=1, c=p[0].get\_color(), marker="s", ms=3, mew=1, mec="k", mfc=p[0].get\_color())

plt.gcf().set\_size\_inches((4.8, 4.0))

plt.title(r"State variable evolution for I.C. " + format\_IC(IC))

plt.xlabel("Time (s)", labelpad=2.0)

plt.ylabel("State variable value", va="top")

plt.grid(which="both", alpha=0.2)

plt.legend(loc="upper right")

plt.show()

def pseudo\_pendulum\_rollout(rollout\_fn=None):

if rollout\_fn is None:

rollout\_fn = rollout

# ICs = [[0, 0, np.pi, 5], [0, 0, np.pi, 12.2], [0, 0, np.pi, 12.3], [0, 0, np.pi, 15]]

ICs1 = [[0, 0, -np.pi, x] for x in [3, 7, 11, 13.7]]

ICs2 = [[0, 0, np.pi, x] for x in [3, 7, 11, 13.7]]

ICs3 = [[0, 0, -np.pi\*3, x] for x in [14, 15, 18]]

ICs4 = [[0, 0, np.pi\*3, -x] for x in [14, 15, 18]]

ICs5 = [[0, 0, np.pi\*3, x] for x in [14]]

ICs6 = [[0, 0, -np.pi\*3, x] for x in [14]]

ICsa, ICsb, ICsc = ICs1.copy(), ICs3.copy(), ICs5.copy()

ICsa.extend(ICs2)

ICsb.extend(ICs4)

ICsc.extend(ICs6)

if False:

for i, IC in enumerate(ICs):

states = rollout\_fn(IC, 150, 0.02)

states[:,2][np.abs(states[:,2])>2.2\*np.pi] = np.nan

plt.plot(states[:,2], states[:,3], c="tab:blue", alpha=0.9, lw=1)

for i, IC in enumerate(ICs3):

states = rollout\_fn(IC, 150, 0.02)

states[:,2][np.abs(states[:,2])>2.2\*np.pi] = np.nan

plt.plot(states[:,2], states[:,3], c="tab:green", alpha=0.9, lw=1)

for i, IC in enumerate(ICs5):

states = rollout\_fn(IC, 150, 0.02)

states[:,2][np.abs(states[:,2])>2.2\*np.pi] = np.nan

plt.plot(states[:,2], states[:,3], c="tab:orange", lw=2)

plt.show()

for IC in [[0, 0, np.pi, x] for x in [1, 10]]:

f1\_IC(rollout\_fn, IC, 150)

for IC in [[0, 0, np.pi, x] for x in [15]]:

f1\_IC(rollout\_fn, IC, 150, True)

for IC in [[0, 0, np.pi, x] for x in [13.9]]:

f1\_IC(rollout\_fn, IC, 150)

# for i, IC in enumerate(ICs3):

# states = rollout(IC, 200, 0.02)

# states[:,2][np.abs(states[:,2])>2.2\*np.pi] = np.nan

# plt.plot(states[:,2], states[:,3], c="tab:green", alpha=0.9, lw=1)

# for i, IC in enumerate(ICs5):

# states = rollout(IC, 200, 0.02)

# states[:,2][np.abs(states[:,2])>2.2\*np.pi] = np.nan

# plt.plot(states[:,2], states[:,3], c="tab:orange", lw=2)

def pseudo\_pendulum\_rollout\_2():

# ICs = [[0, 0, np.pi, 5], [0, 0, np.pi, 12.2], [0, 0, np.pi, 12.3], [0, 0, np.pi, 15]]

ICs = [[0, -5, -np.pi, x] for x in [3, 5, 7, 9, 11, 12.2]]

ICs2 = [[0, -5, np.pi, x] for x in [3, 5, 7, 9, 11, 12.2]]

ICs3 = [[0, -5, -np.pi\*3, x] for x in [12.3, 13, 15]]

ICs4 = [[0, -5, np.pi\*3, -x] for x in [12.3, 13, 15]]

ICs.extend(ICs2)

ICs.extend(ICs3)

ICs.extend(ICs4)

for IC in ICs:

states = rollout(IC, 100, 0.02)

states[:,2][np.abs(states[:,2])>2.2\*np.pi] = np.nan

plt.plot(states[:,2], states[:,3], lw=1)

plt.show()

def cart\_induced\_oscillation():

# ICs = [[0, 0, np.pi, 5], [0, 0, np.pi, 12.2], [0, 0, np.pi, 12.3], [0, 0, np.pi, 15]]

ICs = [[0, x, -np.pi, 0] for x in [1, 2, 4, 8, 10, 50]]

ICs2 = [[0, x, -np.pi+0.0001, 0] for x in [1, 2, 4, 8, 10, 50]]

ICs.extend(ICs2)

for IC in ICs:

states = rollout(IC, 100, 0.02)

plt.plot(states[:,2], states[:,3], lw=1)

plt.show()

n = np.arange(len(states[:,0]))

plt.plot(n, states[:,0])

plt.plot(n, states[:,1])

plt.plot(n, states[:,2])

plt.plot(n, states[:,3])

plt.show()

# # ICs = [[0, 0, np.pi, 5], [0, 0, np.pi, 12.2], [0, 0, np.pi, 12.3], [0, 0, np.pi, 15]]

# ICs = [[0, x, -np.pi, x] for x in [1]]

# for IC in ICs:

# states = rollout(IC, 100, 0.02)

# states[:,2][np.abs(states[:,2])>2.2\*np.pi] = np.nan

# plt.plot(states[:,2], states[:,3], lw=1)

# plt.show()

def pseudo\_pendulum\_cart\_rollout():

# ICs = [[0, 0, np.pi, 5], [0, 0, np.pi, 12.2], [0, 0, np.pi, 12.3], [0, 0, np.pi, 15]]

ICs = [[0, .1\*x, np.pi, -x] for x in [3,5, 9, 13.5, 13.8]]

ICs3 = [[0, .1\*x, np.pi\*3, -x] for x in [13.9, 15]]

for IC in ICs:

states = rollout(IC, 200, 0.01)

states[:,2][np.abs(states[:,2])>5] = np.nan

plt.plot(np.arange(200), states[:,1], "r", lw=1)

for IC in ICs3:

states = rollout(IC, 200, 0.01)

states[:,2][np.abs(states[:,2])>5] = np.nan

plt.plot(np.arange(200), states[:,1], "b", lw=1)

plt.show()

def energy\_ellipse\_params(E, theta):

RHS = E + (9.8/8)\*(1-np.cos(theta))

a = 0.5

b = .0625 \* np.cos(theta)

c = 1/48

conditions = (a\*c - b\*b > 0, RHS / (a + c) > 0)

# print(conditions)

if not all(conditions):

return

sqrt\_term = np.sqrt((a-c)\*\*2 + 4 \* b\*\*2)

major = np.sqrt(2\*RHS / (a + c - sqrt\_term))

minor = np.sqrt(2\*RHS / (a + c + sqrt\_term))

angle = .5\*np.pi + .5\*np.arctan2(2\*b, (a-c))

return major, minor, angle

# print(major, minor, 180/np.pi \* (angle-0.5\*np.pi))

def ellipse\_fn(angle, pos, a, b, tilt\_angle):

x\_ = a \* np.cos(angle)

z\_ = b \* np.sin(angle)

x = x\_\*np.cos(tilt\_angle) - z\_\*np.sin(tilt\_angle)

y = np.zeros\_like(angle) + pos

z = x\_\*np.sin(tilt\_angle) + z\_\*np.cos(tilt\_angle)

return x, y, z

def get\_IC(E, theta, phi):

ret = energy\_ellipse\_params(E, theta)

if ret is None:

return None

major, minor, tilt\_angle = ret

x, y, z = ellipse\_fn(phi - tilt\_angle, theta, major, minor, tilt\_angle)

return [0, x, y, z]

def plot\_energy\_ellipse(E, theta):

major, minor, tilt\_angle = energy\_ellipse\_params(E, theta)

angle = np.linspace(0, 2\*np.pi, 100)

x,y,z = ellipse\_fn(t, theta, major, minor, tilt\_angle)

vv.plot(x, y, z, lc=(0, 0, 1), alpha=0.5, lw=3)

def isosurface\_rings(ax, E, theta\_range, phi\_range=(-np.pi, np.pi), N\_thetas=50, N\_phis=50):

# thetas = np.linspace(-5.5\*np.pi, 2.5\*np.pi, 50)

thetas = np.linspace(theta\_range[0], theta\_range[1], N\_thetas)

phis = np.linspace(phi\_range[0], phi\_range[1], N\_phis)

x\_grid = np.zeros(thetas.shape + phis.shape)

y\_grid = np.zeros(thetas.shape + phis.shape)

z\_grid = np.zeros(thetas.shape + phis.shape)

fc\_grid = np.ones(thetas.shape + phis.shape + (4,))

# Thetas, Phis = np.meshgrid(thetas, phis, sparse=False, indexing='ij')

params = np.array([np.array(energy\_ellipse\_params(E, theta)) for theta in thetas])

# print(params)

cmap = cm.get\_cmap('summer', 256)

for i, theta in enumerate(thetas):

major, minor, tilt\_angle = params[i]

x, y, z = ellipse\_fn(phis, theta, major, minor, tilt\_angle)

x\_grid[i,:] = x

y\_grid[i,:] = y

z\_grid[i,:] = z

c = cmap(0.5 + 3\*(tilt\_angle - .5\*np.pi))

fc\_grid[i,:,:] = (c[0], c[1], c[2], .99)

vv.xlabel(r"cart velocity")

vv.ylabel(r"pole angle")

vv.zlabel(r"pole angular velocity")

vv.axis("off")

vv.surf(x\_grid, y\_grid\*3, z\_grid, fc\_grid, axes\_adjust=True)

# ax.plot\_surface(x\_grid, y\_grid, z\_grid, facecolors=fc\_grid, shade=True, rstride=1, cstride=1)

# for i, theta in enumerate(thetas):

# vv.plot(grid[i,:,0],grid[i,:,1], grid[i,:,2], alpha=1, lw=3)

# for i, phi in enumerate(phis):

# vv.plot(grid[:,i,0],grid[:,i,1], grid[:,i,2], alpha=1, lw=3)

# for i, phi in enumerate(phis):

# major, minor, tilt\_angle = params[i]

# x,y,z = ellipse\_fn\_2(phi, thetas, major, minor, tilt\_angle)

# vv.plot(x, y, z, lc=(0, 0, 1), alpha=0.5, lw=3)

def draw\_energy\_trajectories(E=1):

app = vv.use()

f = vv.clf()

ax = vv.cla()

# isosurface\_rings(ax, E, (-0.1\*np.pi, 2.1\*np.pi)) #E>0

# isosurface\_rings(ax, E, (0.01\*np.pi, 1.99\*np.pi)) #E=0

# isosurface\_rings(ax, E, (0.3\*np.pi, 1.7\*np.pi)) #E=-0.5

isosurface\_rings(ax, E, ((1-.28195)\*np.pi, (1+.28195)\*np.pi)) #E=-2

# isosurface\_rings(ax, E, (0.95\*np.pi, 1.05\*np.pi)) #E=-2.4349

# isosurface\_rings(ax, E, ((1-.0912)\*np.pi, (1+.0912)\*np.pi)) #E=-2.5

ICs1 = [get\_IC(E, 0, 0.5\*np.pi),

get\_IC(E, 0, 0.3\*np.pi),

get\_IC(E, 0, 0.17\*np.pi)]

# ICs2 = [get\_IC(E, 1.01\*np.pi, 0),

# get\_IC(E, 1.05\*np.pi, 0),

# get\_IC(E, 1.08\*np.pi, 0),

# get\_IC(E, 1.0911\*np.pi, 0)]

ICs2 = [get\_IC(E, 1.1\*np.pi, 0),

get\_IC(E, 1.2\*np.pi, 0),

get\_IC(E, 1.28195\*np.pi, 0)] #E = -2

# ICs2 = [get\_IC(E, 1.95\*np.pi, 0),

# get\_IC(E, 1.5\*np.pi, 0),

# get\_IC(E, 1.25\*np.pi, 0)] #E>0

ICs1 = [x for x in ICs1 if x is not None]

ICs2 = [x for x in ICs2 if x is not None]

states1\_slow = [rollout(IC, 20, 0.2) for IC in ICs1]

states1\_fast = [rollout(IC, 200, 0.02) for IC in ICs1]

# states2\_slow = [rollout(IC, 20, 0.2) for IC in ICs2]

# states2\_fast = [rollout(IC, 200, 0.02) for IC in ICs2] # E>0

states2\_slow = [rollout(IC, 10, 0.2) for IC in ICs2]

states2\_fast = [rollout(IC, 100, 0.02) for IC in ICs2] #E=-2

for states\_list in [states2\_slow, states2\_fast, states1\_slow, states1\_fast]:

for states in states\_list:

states[:,2][np.abs(states[:,2]) > 2.1\*np.pi] = np.nan

full\_plot = True

sf = 3

c1 = (138/255,43/255,226/255)

c2 = "r"

if full\_plot:

for states in states1\_slow:

vv.plot(states[:,1], states[:,2]\*sf, states[:,3], lc=c1, ls="--", alpha=0.99)

for states in states2\_slow:

vv.plot(states[:,1], states[:,2]\*sf, states[:,3], lc=c2, ls="--", alpha=0.99)

for states in states1\_fast:

vv.plot(states[:,1], states[:,2]\*sf, states[:,3], lc=c1, lw=5, alpha=0.99)

for states in states2\_fast:

vv.plot(states[:,1], states[:,2]\*sf, states[:,3], lc=c2, lw=5, alpha=0.99)

for states in states1\_slow:

vv.plot(states[:,1], states[:,2]\*sf, states[:,3], mc=c1, mw=5, mew=3, mec="k", ms="o", ls="", alpha=0.99)

for states in states2\_slow:

vv.plot(states[:,1], states[:,2]\*sf, states[:,3], mc=c2, mw=5, mew=3, mec="k", ms="o", ls="", alpha=0.99)

else:

for states in states2\_fast:

vv.plot(states[:,1], states[:,2]\*sf, states[:,3], lc=c1, lw=5, alpha=0.99)

for states in states1\_fast:

vv.plot(states[:,1], states[:,2]\*sf, states[:,3], lc=c2, lw=5, alpha=0.99)

# isosurface\_rings(1)

app.Run()

# pseudo\_pendulum\_rollout()

# pseudo\_pendulum\_cart\_rollout()

# pseudo\_pendulum\_rollout\_2()

# cart\_induced\_oscillation()

# plot\_energy()

# draw\_energy\_trajectories(E=-2)

# from main import \*

C = np.load("../lin\_model.npy")

print(C)

exit()

# C = np.load("../lin\_model1.npy")

def rollout1(IC, N):

T = np.arange(0, N)

states = np.zeros((len(T), 4))

states[0] = IC

state = IC

for t in T[1:]:

state = state + C @ state

state[2] = remap\_angle(state[2])

states[t] = state

return states

def f2\_IC(IC, N=50, remap=True, t\_step=0.2):

T = np.arange(N)\*t\_step

T3 = np.arange(N\*10)\*t\_step/10

states1 = rollout(IC, N, t\_step)

states2 = rollout1(IC, N)

states3 = rollout(IC, N\*10, t\_step/10)

for j in range(0,4):

if remap:

for states in [states1, states2, states3]:

remapped = remap\_angle\_v(states[:,2] - np.pi) + np.pi

# for i in range(1, len(T)):

# if remapped[i] < -(np.pi-1) and remapped[i-1] > (np.pi-1):

# remapped[i] = np.nan

# elif remapped[i] > (np.pi-1) and remapped[i-1] < -(np.pi-1):

# remapped[i] = np.nan

states[:,2] = remapped

p=plt.plot(T, states2[:,j], label=VAR\_STR[j])

# plt.plot(T3, states3[:,j], ls="--", lw=0.7, c=p[0].get\_color())#, marker="s", ms=3, mew=1, mec="k", mfc=p[0].get\_color())

# plt.plot(T, states1[:,j], lw=0, c=p[0].get\_color(), marker="s", ms=3, mew=1, mec="k", mfc=p[0].get\_color())

plt.gcf().set\_size\_inches((4.8, 4.0))

plt.title(r"Modelled trajectory for random I.C.")

plt.xlabel("Time (s)", labelpad=2.0)

plt.ylabel("State variable value", va="top")

plt.grid(which="both", alpha=0.2)

plt.legend(loc="upper right")

plt.show()

while True:

f2\_IC(rand\_state(), 500, t\_step=0.2)

f2\_IC([0, 0, np.pi, 1], 20, t\_step=0.2)

f2\_IC([0, 0, 0.1, 0], 20, t\_step=0.2)

f2\_IC([0, 0, 0, 1], 20, t\_step=0.2)

f2\_IC([0, 0, 0, 3], 20, t\_step=0.2)

def pseudo\_pendulum\_rollout5(rollout\_fn=None):

if rollout\_fn is None:

rollout\_fn = rollout

# ICs = [[0, 0, np.pi, 5], [0, 0, np.pi, 12.2], [0, 0, np.pi, 12.3], [0, 0, np.pi, 15]]

ICs1 = [[0, 0, -np.pi, x] for x in [3, 7, 11, 13.7]]

ICs2 = [[0, 0, np.pi, x] for x in [3, 7, 11, 13.7]]

ICs3 = [[0, 0, -np.pi\*3, x] for x in [14, 15, 18]]

ICs4 = [[0, 0, np.pi\*3, -x] for x in [14, 15, 18]]

ICs5 = [[0, 0, np.pi\*3, x] for x in [14]]

ICs6 = [[0, 0, -np.pi\*3, x] for x in [14]]

ICsa, ICsb, ICsc = ICs1.copy(), ICs3.copy(), ICs5.copy()

ICsa.extend(ICs2)

ICsb.extend(ICs4)

ICsc.extend(ICs6)

if False:

for i, IC in enumerate(ICs):

states = rollout\_fn(IC, 150, 0.02)

states[:,2][np.abs(states[:,2])>2.2\*np.pi] = np.nan

plt.plot(states[:,2], states[:,3], c="tab:blue", alpha=0.9, lw=1)

for i, IC in enumerate(ICs3):

states = rollout\_fn(IC, 150, 0.02)

states[:,2][np.abs(states[:,2])>2.2\*np.pi] = np.nan

plt.plot(states[:,2], states[:,3], c="tab:green", alpha=0.9, lw=1)

for i, IC in enumerate(ICs5):

states = rollout\_fn(IC, 150, 0.02)

states[:,2][np.abs(states[:,2])>2.2\*np.pi] = np.nan

plt.plot(states[:,2], states[:,3], c="tab:orange", lw=2)

plt.show()

for IC in [[0, 0, np.pi, x] for x in [1, 10]]:

f1\_IC(rollout\_fn, IC, 150)

for IC in [[0, 0, np.pi, x] for x in [15]]:

f1\_IC(rollout\_fn, IC, 150, True)

for IC in [[0, 0, np.pi, x] for x in [13.9]]:

f1\_IC(rollout\_fn, IC, 150)

# for i, IC in enumerate(ICs3):

# states = rollout(IC, 200, 0.02)

# states[:,2][np.abs(states[:,2])>2.2\*np.pi] = np.nan

# plt.plot(states[:,2], states[:,3], c="tab:green", alpha=0.9, lw=1)

# for i, IC in enumerate(ICs5):

# states = rollout(IC, 200, 0.02)

# states[:,2][np.abs(states[:,2])>2.2\*np.pi] = np.nan

# plt.plot(states[:,2], states[:,3], c="tab:orange", lw=2)

Model.py

from globals import \*

from CartPole import \*

sys = CartPole(0.2, False)

def rollout(IC, N, delta\_time=0.2):

global sys

if sys.delta\_time != delta\_time:

sys = CartPole(delta\_time, False)

sys.setState(IC)

T = np.arange(0, N)

states = np.zeros((len(T), 4))

states[0] = sys.getState()

for t in T[1:]:

sys.performAction(0.)

states[t] = sys.getState()

return states

def single\_action(state, delta\_time=0.2):

global sys

if sys.delta\_time != delta\_time:

sys = CartPole(delta\_time, False)

sys.setState(state)

sys.performAction(0.)

return np.array(sys.getState())

def modelf(state, delta\_time=0.2):

global sys

if sys.delta\_time != delta\_time:

sys = CartPole(delta\_time, False)

sys.setState(state)

sys.performAction(0.)

return np.array(sys.getState()) - np.array(state)

main.py

from globals import \*

from model import \*

from utils import \*

def train\_model(N=500, t\_step=0.2):

X = np.zeros((N,4))

Y = np.zeros((N,4))

for i in range(N):

X[i,:] = rand\_state([10, 10, np.pi, 15])

Y[i,:] = modelf(X[i,:], t\_step)

CT, res, rank, s = np.linalg.lstsq(X, Y, rcond=None)

return CT.T

# return C, s, res

def junk\_lin\_reg():

X = [10, 20, 30, 40, 50, 70, 80, 100]#, 200, 500, 1000, 2000, 3000]#, 5000, 7000, 10000]

Y1 = []

Y2 = []

Y6 = []

i = 0

for N in X:

C, s, res = train\_model(N)

if i!=0:

print((C1-C)[0])

n1 = np.linalg.norm(C1-C, 'fro')

n2 = np.linalg.norm(s)

Y1.append(n1)

Y2.append(s)

Y6.append(res/N)

C1 = C

i=1

print(C)

# plt.semilogx()

# plt.plot(X[1:], Y1)

# plt.show()

# plt.loglog()

# plt.plot(X[1:], Y2)

# plt.show()

plt.loglog()

plt.plot(X[1:], Y6)

plt.show()

def show\_matrix(M, zero\_range = 0, title="", x="", y="", axes=True, div=True, cmap\_str=None):

if cmap\_str is not None:

cmap = cm.get\_cmap(cmap\_str, 256)

newcolors = cmap(np.linspace(0, 1, 256))

else:

if div:

cmap = cm.get\_cmap("RdYlBu", 256)

newcolors = cmap(np.linspace(1, 0, 256))

else:

cmap = cm.get\_cmap("viridis", 256)

newcolors = cmap(np.linspace(0, 1, 256))

newcolors[:zero\_range, :3] = 0

newcmp = ListedColormap(newcolors)

if axes:

plt.gca().set\_xticks([0,1,2,3])

plt.gca().set\_xticklabels(VAR\_STR)

plt.gca().set\_yticks([0,1,2,3])

plt.gca().set\_yticklabels(VAR\_STR)

plt.xlabel(x, labelpad=2.0)

plt.ylabel(y)

else:

plt.gca().set\_xticks([])

plt.gca().set\_yticks([])

plt.title(title)

plt.imshow(M, cmap = newcmp)

plt.colorbar()

if div:

maxelem = np.max(np.abs(M[np.isnan(M)==False]))

plt.clim(-maxelem, maxelem)

print(M)

plt.show()

# USE SOBEL SEQUENCE?

def task1\_2a():

RR = np.zeros((4, 4))

MM = np.zeros((4, 4))

DD = np.zeros((4, 4))

NK = 100

for k in range(NK):

print(k)

R = np.zeros((4,4))

M = np.zeros((4,4))

D = np.zeros((4,4))

x = rand\_state()

for i in range(4):

for j in range(4):

DXi = P\_RANGE[i] \* np.linspace(-1, 1, 10) - x[i]

z = []

for dxi in DXi:

x1 = x.copy()

x1[i] += dxi

z.append(modelf(x1, 0.2)[j])

z1 = np.sum(np.abs(z-z[0]))

X = DXi + x[i]

slope, intercept = np.polyfit(X, z, 1)

# slope1= np.sum((DXi + x[i])\*z)/np.sum(np.square(DXi + x[i]))

# slope2= z[-1]/(DXi + x[i])[-1]

# print(slope,slope1, slope2, intercept, np.linalg.norm(z - ((DXi + x[i])\*slope + intercept)), np.linalg.norm(z - ((DXi + x[i])\*slope2)))

M[j,i] = slope

if z1 == 0:

R[j,i] = np.nan

else:

R[j,i] = np.corrcoef(X, z)[0,1]

# D[j,i] = np.linalg.norm(z - ((X)\*slope + intercept))

# if i < 2:

# p = plt.plot(X, z)

# p = plt.plot(X, X\*slope + intercept, ls="--", c=p[0].get\_color())

# plt.show()

RR += R/NK

MM += M/NK

DD += D/NK

# print(RR)

# print(MM)

# RR = np.abs(RR)

MM = np.abs(MM)

# RR[np.abs(RR)<1e-15] = np.nan

MM[np.abs(MM)<1e-12] = np.nan

DD[np.abs(DD)<1e-12] = np.nan

# show\_matrix(np.log10(DD), 0,

# r"$\log\_{10}$ " + f"mean square deviation of best-fit line\nfrom scans over {NK} random initial states",

# "Component of X scanned",

# "Component of Z", div=False)

show\_matrix(RR, 0,

r"$\langle (Z\_i, X\_j)$ correlation coefficient$\rangle$ for" + f"\nscans over {NK} random initial states",

"Component of X scanned",

"Component of Z", div=True)

show\_matrix(np.log10(MM), 0,

r"$\log\_{10}\left(\left|\left\langle\frac{dZ\_i}{dX\_j}\right\rangle\right|\right)$" + f" of best-fit line to\n scans over {NK} random initial states",

"Component of X scanned",

"Component of Z", div=False)

# show\_matrix(np.log10(MM), 0,

# r"$\log\_{10}\left(\left\langle\left\vert\frac{dY\_i}{dX\_j}\right\vert\right\rangle\right)$" + f" evaluated at {NK} random \n initial states, normalised to max value",

# "Component of X varied",

# "Component of Y", div=False)

def task1\_2b():

M = np.zeros((4, 4))

for i in range(4):

for j in range(4):

for k in range(200):

x = rand\_state()

y1 = modelf(x, 0.2)

x[i] += P\_RANGE[i]\*0.0001

y2 = modelf(x, 0.2)

delta = np.abs(y1[j]-y2[j])

M[j,i] += delta

#adjust this to match above

M /= np.max(M)

M[np.abs(M)<1e-12] = np.nan

print(M)

show\_matrix(np.log10(M), 0,

"Approximate dependencies of system\ntime evolution on state variables",

"Parameter varied",

"Mean gradient of output ", div=False)

def task1\_3a():

for n in [500, 5000]:

C = train\_model(n)

show\_matrix(np.log10(np.abs(C)), 0,

r"$\log\_{10}(|$Linear model matrix components$|)$" + f"\n with {n} training pairs", axes=False, div=False)

# C = train\_model(10000)

# show\_matrix(np.log10(C))

# task1\_2a()

# task1\_3a()

C = np.load("../lin\_model.npy")

def task1\_3c():

colours = ["tab:blue", "tab:orange", "tab:green", "tab:red"]

for i in range(4):

a = np.linspace(-5, 5, 2)

plt.plot(a,a, "k--", alpha=0.5, lw=1, label="perfect prediction")

NK = 100

for k in range(NK):

print(k)

x = rand\_state()

for j in range(0,4):

DXi = P\_RANGE[i] \* np.linspace(-1, 1, 10) - x[i]

z = []

z1 = []

for dxi in DXi:

x1 = x.copy()

x1[i] += dxi

z.append(modelf(x1, 0.2)[j])

z1.append((C @ x1)[j])

lw = 2

if i == 1 and j == 0:

lw=0.5

if i==2 or i ==3:

lw=1

if k == 0:

p = plt.plot(z, z1, c=colours[j], lw=lw, label=VAR\_STR[j])

else:

p = plt.plot(z, z1, c=colours[j], lw=lw)

plt.ylabel(r"$CX$")

plt.xlabel(r"$f(X)$")

plt.title(f"Modelled state evolution vs. actual evolution\nfor {NK} random I.C.s, scanned over {VAR\_STR[i]}")

plt.legend()

plt.show()

task1\_3c()

exit()

def task\_3b():

X = np.power(10, np.linspace(1, 3.5, 50))

C = np.zeros((len(X), 16))

y = np.zeros(len(X))

print(X)

for i, n in enumerate(X):

C = train\_model(int(n))

# print(C)

y[i] = np.linalg.norm(C, "fro")

print(i, y[i])

print(X, y)

plt.semilogx()

# plt.gca().set\_xticks(X)

plt.scatter(X, y, marker="x")

plt.plot([500, 500], [0, max(y)], "r--")

plt.title("Frobenius norm of C for N\nrandomly drawn training states")

plt.ylabel("Norm")

plt.xlabel("N", labelpad=2.0)

plt.show()

# C = train\_model(500, 0.2)

# np.save("../lin\_model2", C)

# print(C)

C = np.load("../lin\_model.npy")

# fig, ax = plt.subplots(3, 3)

# ax = ax.flatten()

def fn1():

si = 1

ai = 2

bi = 3

# for i, scan\_var in enumerate(np.linspace(-10, 10, 9)):

scan\_var = 0

X = []

Y = []

Z = []

for a in np.linspace(-np.pi, np.pi, 50):

for b in np.linspace(-15, 15, 50):

state = np.array([0., 0., 0., 0.])

state[si] = scan\_var

state[ai] = a

state[bi] = b

# state = rand\_state([10, 0, np.pi, 15]) + np.array([0, scan\_var, 0, 0])

x = state[ai]

y = state[bi]

# pred = C @ state

# print(pred)

actual = modelf(state, 0.1)

# pred[2] = remap\_angle(pred[2])

# actual[2] = remap\_angle(actual[2])

# error = np.linalg.norm(pred - actual)

X.append(x)

Y.append(y)

Z.append(actual[2])

# ax[i].tricontourf(X, Y, Z)

# ax[i].set\_title(scan\_var)

plt.tricontourf(X, Y, Z)

plt.title("theta")

plt.colorbar()

plt.show()

def fn2():

si = 1

xi = 2

yi = 3

NS = 5

NX = 50

NY = 50

x = np.linspace(-P\_RANGE[xi], P\_RANGE[xi], NX)

y = np.linspace(-P\_RANGE[yi], P\_RANGE[yi], NY)

X, Y = np.meshgrid(x, y)

scan\_range = np.linspace(0, 10, NS)

scan\_var = 0

for k in range(4):

ax = plt.gca()

Z = np.zeros\_like(X)

# if l == 0:

# Z0 = Z.copy()

for i, x\_val in enumerate(x):

for j, y\_val in enumerate(y):

state = np.array([0., 0., 0., 0.])

state[si] = scan\_var

state[xi] = x\_val

state[yi] = y\_val

pred = C @ state

actual = modelf(state, 0.2)

# pred[2] = remap\_angle(pred[2])

# actual[2] = remap\_angle(actual[2])

# error = np.linalg.norm(pred[k] - actual[k])

error = actual[k]-pred[k]

Z[j, i] = error #actual[k] #np.linalg.norm(actual)

# plt.tricontourf(X, Y, Z)

# Z -= np.mean(Z)

# plt.xlabel("Time (s)", labelpad=2.0)

# plt.ylabel("State variable value", va="top")

im = ax.contourf(X, Y, Z)

ax.set\_xlabel(VAR\_STR[xi])

ax.set\_ylabel(VAR\_STR[yi])

ax.set\_title(r"Error in prediction of " + VAR\_STR[k])

divider = make\_axes\_locatable(ax)

cax = divider.append\_axes("right", size="5%", pad=0.05)

cb = plt.colorbar(im, cax=cax)

if xi == 2: axis\_pi\_multiples(ax.xaxis)

if yi == 2: axis\_pi\_multiples(ax.yaxis)

# if k == 2: axis\_pi\_multiples(cb.ax.yaxis)

# Z0 = Z

plt.show()

# plt.plot(X[0,:], Z[0,:])

# plt.show()

def fn3():

si = 1

xi = 2

yi = 3

NS = 5

NX = 50

NY = 50

x = np.linspace(-P\_RANGE[xi], P\_RANGE[xi], NX)

y = np.linspace(-P\_RANGE[yi], P\_RANGE[yi], NY)

X, Y = np.meshgrid(x, y)

scan\_range = np.linspace(0, 10, NS)

for k in range(4):

ZZ = np.zeros((NS, NX, NY))

ax = plt.gca()

for l, scan\_var in enumerate(scan\_range):

print(l)

Z = np.zeros\_like(X)

# if l == 0:

# Z0 = Z.copy()

for i, x\_val in enumerate(x):

for j, y\_val in enumerate(y):

state = np.array([0., 0., 0., 0.])

state[si] = scan\_var

state[xi] = x\_val

state[yi] = y\_val

# pred = C @ state

actual = modelf(state, 0.2)

# pred[2] = remap\_angle(pred[2])

# actual[2] = remap\_angle(actual[2])

# error = np.linalg.norm(pred - actual)

Z[j, i] = actual[k] #np.linalg.norm(actual)

# plt.tricontourf(X, Y, Z)

Z -= np.mean(Z)

ZZ[l,:,:] = Z

# ZZ -= np.mean(ZZ, axis=0)

# stds = np.sum(np.abs(ZZ), axis=0)

stds = np.std(ZZ, axis=0)

im = ax.contourf(X, Y, stds, cmap="inferno")

ax.set\_xlabel(VAR\_STR[xi])

ax.set\_ylabel(VAR\_STR[yi])

# ax.set\_title(r"Evolution of " + VAR\_STR[k])

divider = make\_axes\_locatable(ax)

cax = divider.append\_axes("right", size="5%", pad=0.05)

cb = plt.colorbar(im, cax=cax)

if xi == 2: axis\_pi\_multiples(ax.xaxis)

if yi == 2: axis\_pi\_multiples(ax.yaxis)

# if k == 2: axis\_pi\_multiples(cb.ax.yaxis)

plt.show()

# task1\_2()

# fn1()

fn2()

# fn3()

# pseudo\_pendulum\_rollout(rollout1)