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

```python
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"$\log_{10}\left(\left|\left\langle\frac{dZ_i}{dX_j}\right\rangle\right|\ri
ght)$" + 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"$\log_{10}\left(\left\langle\left\vert\frac{dY_i}{dX_j}\right\vert\right\r
angle\right)$" + 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)
```