HOMEWORK 2

1a. Lotka-Volterra model

[i]. Explain why this is called the predator-prey model.

It's called the predator prey model because of the implicit assumptions made by the model.

In particular

1. Specie 1 (y1) population increases at all times provided previous population exists;
2. Species 2 population size increases only at interaction with species 1 with a concomittant reduction in species 1 population
3. Species 2 population thus depends directly in species 1 while species 1 population is jointly dependent on self proliferation and interaction with species 1.

Statement 2 is best described as a predator-prey relationship.

1a. Lotka-Volterra model

[ii]. What is the corresponding ODE model?

ODE equations

the deterministic dynamics, which describe the population of the species, are described by the following ODEs

$$\frac{\mathrm{d}[y_1]}{\mathrm{d}t} = c_1[y_1] - c_2[y_1][y_2],$$

$$\frac{\mathrm{d}[y_2]}{\mathrm{d}t} = c_2[y_1][y_2] - [c_3][y_3].$$

1a. Lotka-Volterra model

[iii]. Explain whether species 1 or species 2 can exist in isolation?

Species 1 can exist in isolation of species 2 since negative rate (death rate) only accrues from its interaction with species 2

Species 1 cannot exist in isolation of species 1 since it's postive rate (growth rate) only accrues from its interaction with species 1.

1b. Use the Gillepsie algorithm to simulate trajectories from the model. use c1 = 1, c2 = 0.005, c3 = 0.6

I implemented the SSA initially using a simple code which I developed from the gillepsie method described in the Singham paper. The results did not seem right so I consulted a gillepsie template from http://be150.caltech.edu/2019/handouts/12_stochastic_simulation_all_code.html (http://be150.caltech.edu/2019/handouts/12_stochastic_simulation_all_code.html)

I have both solutions here.

In [1]:

```
import multiprocessing
import tqdm

import numpy as np
import scipy
import scipy.stats as st
import numba

# Plotting modules
import matplotlib.pyplot as plt
import bokeh.io
import bokeh.plotting

bokeh.io.output_notebook()

# Line profiler (can install with conda install line_profiler)
%load_ext line_profiler
```

(https://bokeh.org) Loading BokehJS ...

In [2]:

```
# Simple solution

c = np.zeros(3)
y = np.zeros(2)
a = np.zeros(3)
v = np.zeros((2,3))
c[0] = 1.0
c[1] = 0.5
c[2] = 0.005
y[0] = 100.
y[1] = 5.
ytraj = []
ttraj = []


# Stoichometric matrix

v[0] = [1, -1, 0]
v[1] = [0, 1, -1]

t = 0
tfinal = 2

while t < tfinal:
    a[0] = c[0]*y[0]
    a[1] = c[1]*y[0]*y[1]
    a[2] = c[2]*y[1]
    asum = sum(a)
    next_ind = np.random.random()<np.array(np.cumsum(a/asum))
    next_indices = [i for i, x in enumerate(next_ind) if x]
    j = next_indices.index(min(next_indices))
    tau = np.log(1/np.random.random())/asum
    y = y + v[:,j]
    ytraj.append(y)
    t = t + tau
    ttraj.append(t)
```

In [3]:

```
# Other Solution

simple_update = np.array([[1, 0],    # give birth to prey
                          [-1, 0],    # prey meets predator
                          [0, 1],     # prey meets predator
                          [0, -1]],    # predator meets death
                          dtype = np.int)

def simple_propensity(propensities, population, t, c1, c2, c3):
    """Updates an array of propensities given a set of parameters
    and an array of populations.
    """
    # Unpack population
    y1, y2 = population

    # Update propensities
    propensities[0] = c1*y1         # gave birth to prey?
    propensities[1] = c2*y1*y2      # prey meets predator?
    propensities[2] = c2*y1*y2      # prey meets predator?
    propensities[3] = c3*y2        # predator meets death?


def sample_discrete(probs):
    """Randomly sample an index with probability given by probs."""
    # Generate random number
    q = np.random.rand()

    # Find index
    i = 0
    p_sum = 0.0
    while p_sum < q:
        p_sum += probs[i]
        i += 1
    return i - 1


def gillespie_draw(propensity_func, propensities, population, t, args=()):
    """
    Draws a reaction and the time it took to do that reaction.

    Parameters
    ----------
    propensity_func : function
        Function with call signature propensity_func(population, t, *args)
        used for computing propensities. This function must return
        an array of propensities.
    population : ndarray
        Current population of particles
    t : float
        Value of the current time.
    args : tuple, default ()
        Arguments to be passed to `propensity_func`.

    Returns
    -------
    rxn : int
        Index of reaction that occured.
    time : float
        Time it took for the reaction to occur.
```

```
    """

    # Compute propensities
    propensity_func(propensities, population, t, *args)

    # Sum of propensities
    props_sum = propensities.sum()

    # Compute next time
    time = np.random.exponential(1.0 / props_sum)

    # Compute discrete probabilities of each reaction
    rxn_probs = propensities / props_sum

    # Draw reaction from this distribution
    rxn = sample_discrete(rxn_probs)

    return rxn, time


def gillespie_ssa(propensity_func, update, population_0, time_points, args=()):
    """
    Uses the Gillespie stochastic simulation algorithm to sample
    from probability distribution of particle counts over time.

    Parameters
    ----------
    propensity_func : function
        Function of the form f(params, t, population) that takes the current
        population of particle counts and return an array of propensities
        for each reaction.
    update : ndarray, shape (num_reactions, num_chemical_species)
        Entry i, j gives the change in particle counts of species j
        for chemical reaction i.
    population_0 : array_like, shape (num_chemical_species)
        Array of initial populations of all chemical species.
    time_points : array_like, shape (num_time_points,)
        Array of points in time for which to sample the probability
        distribution.
    args : tuple, default ()
        The set of parameters to be passed to propensity_func.

    Returns
    -------
    sample : ndarray, shape (num_time_points, num_chemical_species)
        Entry i, j is the count of chemical species j at time
        time_points[i].
    """

    # Initialize output
    pop_out = np.empty((len(time_points), update.shape[1]), dtype=np.int)

    # Initialize and perform simulation
    i_time = 1
    i = 0
    t = time_points[0]
    population = population_0.copy()
    pop_out[0,:] = population
    propensities = np.zeros(update.shape[0])
    while i < len(time_points):
        while t < time_points[i_time]:
            # draw the event and time step
            event, dt = gillespie_draw(propensity_func, propensities, population, t, args)

            # Update the population
            population_previous = population.copy()
            population += update[event,:]

            # Increment time
            t += dt

        # Update the index
        i = np.searchsorted(time_points > t, True)

        # Update the population
        pop_out[i_time:min(i,len(time_points))] = population_previous

        # Increment index
        i_time = i

    return pop_out
```

In [4]:

```
# Specify parameters for calculation
c1 = 1.0
c2 = 0.005
c3 = 0.6
args = (c1, c2, c3)
time_points = np.linspace(0, 30, 101)
population_0 = np.array([100, 10], dtype=int)
n_traj = 40

# Seed random number generator for reproducibility
from random import seed
some_seed = 43210
np.random.seed = some_seed

# Initialize output array
samples = np.empty((n_traj, len(time_points), 2), dtype=int)

# Run the calculations
for i in tqdm.tqdm_notebook(range(n_traj)):
    samples[i,:,:] = gillespie_ssa(simple_propensity, simple_update,
                                   population_0, time_points, args=args)
```

```
/home/dcajuzie/.virtualenvs/dl4cv/lib/python3.6/site-packages/ipykernel_launcher.py:19: TqdmDeprecationWarning: This functio
n will be removed in tqdm==5.0.0
Please use `tqdm.notebook.tqdm` instead of `tqdm.tqdm_notebook`


/home/dcajuzie/.virtualenvs/dl4cv/lib/python3.6/site-packages/ipykernel_launcher.py:68: RuntimeWarning: divide by zero encou
ntered in double_scalars
/home/dcajuzie/.virtualenvs/dl4cv/lib/python3.6/site-packages/ipykernel_launcher.py:71: RuntimeWarning: invalid value encoun
tered in true_divide
```

We now have our samples, so we can plot the trajectories. For visualization, we will plot every trajectory as a thin blue line, and then the average of the trajectories as a thick orange line.

In [6]:

```
# Set up plots
plots = [bokeh.plotting.figure(plot_width=300,
                               plot_height=200,
                               x_axis_label='time',
                               y_axis_label='prey population'),
         bokeh.plotting.figure(plot_width=300,
                               plot_height=200,
                               x_axis_label='time',
                               y_axis_label='predator population')]

# Plot trajectories and mean
for i in [0, 1]:
    for x in samples[:,:,i]:
        plots[i].line(time_points, x, line_width=0.5,
                      alpha=0.2, line_join='bevel')
    plots[i].line(time_points, samples[:,:,i].mean(axis=0),
                  line_width=3, color='orange', line_join='bevel')

# Link axes
plots[0].x_range = plots[1].x_range

bokeh.io.show(bokeh.layouts.gridplot(plots, ncols=2))
```

1c. Changing which parameter will make it more likely for species 1 to go extinct? What about species 2? Change these parameters gradually to verify your hypothesis in simulations.

Species 1 goes extinct: changing parameters c2 and c1.

Increasing c2 will increase the propensity for prey-predator interactions which leads to reduction in the population of species 1.

Decreasing c1 will reduce the propensity for procreation of species 1. This will tilt the result towards an increased chance of extinction

Species 2 goes extinct: changing parameters c3, c2 and c1.

Increasing c3 will increase the death rate. If this outcome is more probable than interaction with prey, then c3 is more likely to go extinct.

Decreasing c2 will reduce the propensity for predator-prey interactions and hence species 2 birth rate.

Decreasing c1 will reduce species 1 abundance and species 2 abundance is directly tied to species 1 abundance. Decreasing c2 and c1 signficantly could tilt the outcome towards extinction.

I will now attempt to simulate prey extinction

In [8]:

```
# Specify parameters for calculation
c1 = 1.0

for c2 in [0.005, 0.02, 0.4]:
    for c3 in [0.6, 0.06, 0.006]:
        args = (c1, c2, c3)
        time_points = np.linspace(0, 30, 101)
        population_0 = np.array([100, 10], dtype=int)
        n_traj = 5

        # Seed random number generator for reproducibility
        from random import seed
        some_seed = 43210
        np.random.seed = some_seed

        # Initialize output array
        samples = np.empty((n_traj, len(time_points), 2), dtype=int)

        # Run the calculations
        for i in range(n_traj):
            samples[i,:,:] = gillespie_ssa(simple_propensity, simple_update,
                                           population_0, time_points, args=args)

        # Set up plots
        plots = [bokeh.plotting.figure(plot_width=300,
                                       plot_height=200,
                                       x_axis_label='time',
                                       y_axis_label='prey population'),
                 bokeh.plotting.figure(plot_width=300,
                                       plot_height=200,
                                       x_axis_label='time',
                                       y_axis_label='predator population')]

        # Plot trajectories and mean
        for i in [0, 1]:
            for x in samples[:,:,i]:
                plots[i].line(time_points, x, line_width=0.3,
                              alpha=0.2, line_join='bevel')
            plots[i].line(time_points, samples[:,:,i].mean(axis=0),
                          line_width=3, color='orange', line_join='bevel')

        # Link axes
        plots[0].x_range = plots[1].x_range

        bokeh.io.show(bokeh.layouts.gridplot(plots, ncols=2))
```

```
/home/dcajuzie/.virtualenvs/dl4cv/lib/python3.6/site-packages/ipykernel_launcher.py:68: RuntimeWarning: divide by zero encou
ntered in double_scalars
/home/dcajuzie/.virtualenvs/dl4cv/lib/python3.6/site-packages/ipykernel_launcher.py:71: RuntimeWarning: invalid value encoun
tered in true_divide
```

Case: Prey Extinction

In the figure above, I held c1 constant and gradually changed c2 and c3 as species 1 tends towards extinction.

It is intituive that the predator population may likely suffer the same fate as the prey: extinction.

I will now attempt to simulate predator extinction.

In [9]:

```
# Specify parameters for calculation
c3 = 0.6

for c2 in [0.005, 0.010, 0.02]:
    for c1 in [1.0, 0.5, 0.15]:
        args = (c1, c2, c3)
        time_points = np.linspace(0, 30, 101)
        population_0 = np.array([100, 10], dtype=int)
        n_traj = 5

        # Seed random number generator for reproducibility
        from random import seed
        some_seed = 43210
        np.random.seed = some_seed

        # Initialize output array
        samples = np.empty((n_traj, len(time_points), 2), dtype=int)

        # Run the calculations
        for i in range(n_traj):
            samples[i,:,:] = gillespie_ssa(simple_propensity, simple_update,
                                           population_0, time_points, args=args)

        # Set up plots
        plots = [bokeh.plotting.figure(plot_width=300,
                                       plot_height=200,
                                       x_axis_label='time',
                                       y_axis_label='prey population'),
                 bokeh.plotting.figure(plot_width=300,
                                       plot_height=200,
                                       x_axis_label='time',
                                       y_axis_label='predator population')]

        # Plot trajectories and mean
        for i in [0, 1]:
            for x in samples[:,:,i]:
                plots[i].line(time_points, x, line_width=0.3,
                              alpha=0.5, line_join='bevel')
            plots[i].line(time_points, samples[:,:,i].mean(axis=0),
                          line_width=3, color='orange', line_join='bevel')

        # Link axes
        plots[0].x_range = plots[1].x_range

        bokeh.io.show(bokeh.layouts.gridplot(plots, ncols=2))
```

```
/home/dcajuzie/.virtualenvs/dl4cv/lib/python3.6/site-packages/ipykernel_launcher.py:68: RuntimeWarning: divide by zero encou
ntered in double_scalars
/home/dcajuzie/.virtualenvs/dl4cv/lib/python3.6/site-packages/ipykernel_launcher.py:71: RuntimeWarning: invalid value encoun
tered in true_divide
```

Case: Predator extinction

Specifically, notice that though the prey continues to proliferate, the predator population continues to dwindle. Perhaps, one of the factors dictated by c2 is the ability of the predator to convert prey to meal. It could also be determined by the mean age of the predator population.

Is it likely that the predator population increases at time t>30?

2a. Unknown model

[i]. Find the fixed points of the system

Fixed point

To find the fixed point of the model, we solve for x with $\dot{x} = 0$ and for y with $\dot{y} = 0$. We get that

$$0 = k - \alpha_1[X] - k_a[X][Y],$$

$$0 = k - \alpha_2[Y] - k_a[X][Y],$$

Simplifying,

$$0 = k - [X](\alpha_1 + k_a[Y])$$

$$0 = k - [Y](\alpha_2 + k_a[X])$$

We have for the fixed points,

$$X = \frac{k}{(\alpha_1 + k_a[Y])}$$

$$Y = \frac{k}{(\alpha_2 + k_a[X])}$$

This reduces to a root finding problem.

$$X = \frac{k}{(\alpha_1 + k_a \frac{k}{(\alpha_2 + k_a X)})}$$

$$Y = \frac{k}{(\alpha_2 + k_a \frac{k}{(\alpha_1 + k_a Y)})}$$

We then have:

$$\alpha_1 k_a X^2 + \alpha_1 \alpha_2 X - k\alpha_2 = 0$$

$$\alpha_2 k_a Y^2 + \alpha_1 \alpha_2 Y - k\alpha_1 = 0$$

The roots of the quadratic above:

X, Y

$$= \frac{\alpha_2 \alpha_1 \pm \sqrt{\alpha_1^2 {\alpha_2}^2 + 4(\alpha_1 \alpha_2 k_a k)}}{2\alpha_1 k_a},$$

$$= \frac{\alpha_2 \alpha 2 \pm \sqrt{\alpha_1^2 {\alpha_2}^2 + 4(\alpha_1 \alpha_2 k_a k)}}{2\alpha_2 k_a}$$

In [25]:

```
# Parameters
k, alpha1, alpha2, ka = 10, 10e-6, 10e-5, 10e-5


# Make composition of functions
y = np.linspace(0, 10, 200)
x = y

# f(y) g(x)
f1= k / (alpha1 + ka*y)
g1= k / (alpha2 + ka*x)

# Show plot
p = bokeh.plotting.figure(height=300, width=350, x_axis_label='x, y', y_axis_label='g,f')
p.line(y, f, line_width=2, legend_label= 'f(y)')
p.line(x, g, line_width=2, color='orange', legend_label='g(x)')
bokeh.io.show(p)

k, alpha1, alpha2, ka = 10e3, 10e-4, 10e-3, 10e-3
f2= k / (alpha1 + ka*y)
g2= k / (alpha2 + ka*x)

# Show plot
p = bokeh.plotting.figure(height=300, width=350, x_axis_label='x, y', y_axis_label='g,f')
p.line(y, f, line_width=2, legend_label= 'f(y)')
p.line(x, g, line_width=2, color='orange', legend_label='g(x)')
bokeh.io.show(p)
```

2a. Unknown model

[i]. Run the Gillepsie Algorithn and show that the behavior is very different in the two cases

In [80]:

```
# Other Solution
```

```
simple_update = np.array([[1, 0, 0],    # make x
                          [0, 1, 0],    # make y
                          [-1, 0, 0],     # degrade x
                          [0, -1, 0],   # degrade y
                          [-1, -1, 1]],    # make c
                          dtype = np.int)

def simple_propensity_q2(propensities, conc, t, k, alpha1, alpha2, ka):
    """Updates an array of propensities given a set of parameters
    and an array of populations.
    """
    # Unpack population
    x, y, c = conc

    # Update propensities
    propensities[0] = k    # make x
    propensities[1] = k    # make y
    propensities[2] = alpha1*x    # degrade x
    propensities[3] = alpha2*y    # degrade y
    propensities[4] = ka*x*y    # make c


def sample_discrete_q2(probs):
    """Randomly sample an index with probability given by probs."""
    # Generate random number
    q = np.random.rand()

    # Find index
    i = 0
    p_sum = 0.0
    while p_sum < q:
        p_sum += probs[i]
        i += 1
    return i - 1


def gillespie_draw_q2(propensity_func, propensities, conc, t, args=()):
    """
    Draws a reaction and the time it took to do that reaction.

    Parameters
    ----------
    propensity_func : function
        Function with call signature propensity_func(population, t, *args)
        used for computing propensities. This function must return
        an array of propensities.
    population : ndarray
        Current population of particles
    t : float
        Value of the current time.
    args : tuple, default ()
        Arguments to be passed to `propensity_func`.

    Returns
    -------
    rxn : int
        Index of reaction that occured.
    time : float
        Time it took for the reaction to occur.
    """
    # Compute propensities
    propensity_func_q2(propensities, conc, t, *args)

    # Sum of propensities
    props_sum = propensities.sum()

    # Compute next time
    time = np.random.exponential(1.0 / props_sum)

    # Compute discrete probabilities of each reaction
    rxn_probs = propensities / props_sum

    # Draw reaction from this distribution
    rxn = sample_discrete_q2(rxn_probs)

    return rxn, time


def gillespie_ssa_q2(propensity_func, update, conc_0, time_points, args=()):
    """
    Uses the Gillespie stochastic simulation algorithm to sample
    from probability distribution of particle counts over time.

    Parameters
    ----------
    propensity_func : function
        Function of the form f(params, t, population) that takes the current
        population of particle counts and return an array of propensities
        for each reaction.
    update : ndarray, shape (num_reactions, num_chemical_species)
        Entry i, j gives the change in particle counts of species j
        for chemical reaction i.
    population_0 : array_like, shape (num_chemical_species)
        Array of initial populations of all chemical species.
    time_points : array_like, shape (num_time_points,)
        Array of points in time for which to sample the probability
        distribution.
    args : tuple, default ()
        The set of parameters to be passed to propensity_func.
```

```
    Returns

    -------
    sample : ndarray, shape (num_time_points, num_chemical_species)
        Entry i, j is the count of chemical species j at time
        time_points[i].
    """

    # Initialize output
    conc_out = np.empty((len(time_points), update.shape[1]), dtype=np.int)

    # Initialize and perform simulation
    i_time = 1
    i = 0
    t = time_points[0]
    conc = conc_0.copy()
    conc_out[0,:] = conc
    propensities = np.zeros(update.shape[0])
    while i < len(time_points):
        while t < time_points[i_time]:
            # draw the event and time step
            event, dt = gillespie_draw(propensity_func, propensities, conc, t, args)

            # Update the population
            conc_previous = conc.copy()
            conc += update[event,:]

            # Increment time
            t += dt

        # Update the index
        i = np.searchsorted(time_points > t, True)

        # Update the population
        conc_out[i_time:min(i,len(time_points))] = conc_previous

        # Increment index
        i_time = i

    return conc_out
```

In [92]:

```
# Specify parameters for calculation
k = 10
alpha1 = 10e-6
alpha2 = 10e-5
ka = 10e-5

args = (k, alpha1, alpha2, ka)
time_points = np.linspace(0, 1000, 21)
conc_0 = np.array([0, 0, 0], dtype=int)
n_traj = 50

# Seed random number generator for reproducibility
from random import seed
some_seed = 43210
np.random.seed = some_seed

# Initialize output array
results = np.empty((n_traj, len(time_points), 3), dtype=int)

# Run the calculations
for i in tqdm.tqdm_notebook(range(n_traj)):
    results[i,:,:] = gillespie_ssa_q2(simple_propensity, simple_update,
                                conc_0, time_points, args=args)
```

```
/home/dcajuzie/.virtualenvs/dl4cv/lib/python3.6/site-packages/ipykernel_launcher.py:21: TqdmDeprecationWarning: This functio
n will be removed in tqdm==5.0.0
Please use `tqdm.notebook.tqdm` instead of `tqdm.tqdm_notebook`
```

In [84]:

```
k = 10e3
alpha1 = 10e-4
alpha2 = 10e-3
ka = 10e-3

args = (k, alpha1, alpha2, ka)
time_points = np.linspace(0, 5, 21)
conc_0 = np.array([0, 0, 0], dtype=int)
n_traj = 50

# Seed random number generator for reproducibility
from random import seed
some_seed = 43210
np.random.seed = some_seed

# Initialize output array
results2 = np.empty((n_traj, len(time_points), 3), dtype=int)

# Run the calculations
for i in tqdm.tqdm_notebook(range(n_traj)):
    results2[i,:,:] = gillespie_ssa_q2(simple_propensity, simple_update,
                                conc_0, time_points, args=args)
```

```
/home/dcajuzie/.virtualenvs/dl4cv/lib/python3.6/site-packages/ipykernel_launcher.py:20: TqdmDeprecationWarning: This functio
n will be removed in tqdm==5.0.0
Please use `tqdm.notebook.tqdm` instead of `tqdm.tqdm_notebook`
```

In [85]:

```
def plot_q2b(result):

    results = result

    # Set up plots
    plots = [bokeh.plotting.figure(plot_width=200,
                                   plot_height=200,
                                   x_axis_label='time',
                                   y_axis_label='[X]'),
             bokeh.plotting.figure(plot_width=200,
                                   plot_height=200,
                                   x_axis_label='time',
                                   y_axis_label='[Y]'),
             bokeh.plotting.figure(plot_width=200,
                                   plot_height=200,
                                   x_axis_label='time',
                                   y_axis_label='[C]')]

    # Plot trajectories and mean
    for i in range(3):
        for x in results[:,:,i]:
            plots[i].line(time_points, x, line_width=0.3,
                          alpha=0.2, line_join='bevel')
        plots[i].line(time_points, results[:,:,i].mean(axis=0),
                      line_width=3, color='orange', line_join='bevel')

    # Link axes
    plots[1].x_range = plots[2].x_range
    plots[0].x_range = plots[1].x_range

    bokeh.io.show(bokeh.layouts.gridplot(plots, ncols=3))
```

In [93]:

```
plot_q2b(results)
```

In [87]:

```
plot_q2b(results2)
```

ii) Can you give an argument why the behavior is different in the two cases?

Though the ratio of k to α_1 , α_2, and k_a is equivalent in both cases, the time to the fixed point is very different for both systems. It is thus the magnitude of these contributions to the time constant of the respective reactions that constitute the differences seen in the stochastic simulation.

These show the stochasticity or discreteness of the system with respect to time.

In [104]:

```
print('X mean copy number (CASE 1) =', results[:,-20:,0].mean())
print('X mean copy number (CASE 2)=', results2[:,-20:,0].mean())
print('Y mean copy number (CASE 1) =', results[:,-20:,1].mean())
print('Y mean copy number (CASE 2)=', results2[:,-20:,1].mean())

print('\n[X] variance (CASE 1)=', results[:,-20:,0].std()**2)
print('[X] variance (CASE 2)=', results2[:,-20:,0].std()**2)
print('[Y] variance (CASE 1)=', results[:,-20:,1].std()**2)
print('[Y] variance (CASE 2)=', results2[:,-20:,1].std()**2)

print('\n[X] noise (CASE 1)=', results[:,-20:,0].std() / results[:,-50:,0].mean())
print('[X] noise (CASE 2)=', results2[:,-20:,0].std() / results2[:,-50:,0].mean())
print('[Y] noise (CASE 1)=', results[:,-20:,1].std() / results[:,-50:,0].mean())
print('[Y] noise (CASE 2)=', results2[:,-20:,1].std() / results2[:,-50:,0].mean())
```

```
X mean copy number (CASE 1) = 322.346
X mean copy number (CASE 2)= 1006.94
Y mean copy number (CASE 1) = 316.901
Y mean copy number (CASE 2)= 1000.928

[X] variance (CASE 1)= 3353.3562839999995
[X] variance (CASE 2)= 11317.7504
[Y] variance (CASE 1)= 3492.573199
[Y] variance (CASE 2)= 11434.154816

[X] noise (CASE 1)= 0.18862830526804344
[X] noise (CASE 2)= 0.11093427814493584
[Y] noise (CASE 1)= 0.19250400626085762
[Y] noise (CASE 2)= 0.11150330491766601
```

3a. Autoregulation model

[i]. Write down the transition matrix for the Markov process describing the system given in Q3 equation 1

First we label each species with an index, which will corresponds to its position in the array of populations in the simulation.

Next, we can set up a table of updates and propensities for the moves we allow in the Gillespie simulation. Then I will provide the transition matrix

index	description	update	propensity
0	transcription of gene to mRNA	r → r + 1	$k_1 + \phi$
1	degradation of mRNA	r → r - 1	$\gamma_r * r$
2	translation of mRNA	p → p + 1	$k_p * r$
3	degradation of protein	p → p - 1	$\gamma_p * p$

In [35]:

```
circuit_update = np.array([
    # 0   1
    [ 1,   0],  # 0
    [ -1,  0],  # 1
    [ 0,   1],  # 2
    [ 0,  -1],  # 3
    ], dtype=int)
```

transition_matrix =

$$\begin{bmatrix} k_1 + \phi & -\gamma_r r & 0 & 0 \\ 0 & 0 & k_p & -\gamma_p \end{bmatrix}$$

b. Consider the case of positive autoregulation:

$$\phi(p) = \frac{k_0 + (\frac{p}{K})^n}{1 + (\frac{p}{K})^n}$$

Given the following parameters, determine the number of fixed points and stability of the fixed points.

parameter	value
k_1	0
γ_p	1
γ_r	1
γ_p	1
k_p	1
k_0	1
K	0.5

Fixed point

To find the fixed point of the model, we solve for r with $\dot{r} = 0$ and for p with $\dot{p} = 0$. We get that

$$0 = k_l + \phi(p) - \gamma_r r,$$

$$0 = rk_p - \gamma_p p,$$

Simplifying,

$$r = \frac{k_l + \phi_p}{\gamma_r}$$

$$p = \frac{rk_p}{\gamma_p}$$

but,

$$\phi(p) = \frac{k_0(\frac{p}{K})^n}{1 + (\frac{p}{K})^n}$$

This gives us:

$$r = \frac{k_l + (\frac{k_0(\frac{p}{K})^n}{1 + (\frac{p}{K})^n})}{\gamma_r}$$

$$p = \frac{k_l + \frac{k_0(\frac{p}{K})^n}{1 + (\frac{p}{K})^n}}{\frac{\gamma_r \gamma_p}{k_p}}$$

We see that there are two parts to the fixed point:

The constant $\frac{k_l}{(\frac{\gamma_r \gamma_p}{k_p})}$ reduces to zero given $k_l = 0$

and the second part which can be written conveniently as a **function**. Specifically,

$$p = f(p)$$

where

$$f(p) = \frac{1}{1 + (\frac{p}{k})^n}.$$

Again,

$$f(p) = \frac{1}{1 + 2p^n}.$$

Since f(p) is monotonically decreasing, f'(p)<0; This means that f(p) is monotonically decreasing. Since p is increasing, there is a single fixed point with p=f(p).

Thus we will have linear stabilility and a singular fixed point for n=1 and n=10.

I have illustrated this in the following plots

In [132]:

```
# Parameters

def plot_fixed_point(n):

    k0 = 1

    # f(p)
    f = lambda p: k0 / (1 + 2*p**n)

    # Make composition of functions
    p = np.linspace(0, 10, 200)
    f = f(p)

    # Show plot
    pplot = bokeh.plotting.figure(title='n = %d' % n, height=300, width=350, x_axis_label='p')
    pplot.line(p, p, line_width=2, legend_label='p')
    pplot.line(p, f, line_width=2, color='orange', legend_label='f(p)')
    pplot.legend.location = 'center_right'

    bokeh.io.show(pplot)
```

In [133]:

```
plot_fixed_point(1)
```

In [134]:

```
plot_fixed_point(10)
```

In [148]:

```
def fixed_point(k0, n):
    return scipy.optimize.brentq(lambda p: k0 - p*(1+2*p**n), 0, k0)
```

Solving for the fixed point

We have shown that there is a single fixed point, p with

$$k_0 = p_0(1 + p_0^n).$$

This devolves into a root finding problem really.I use here a **Brent's method (https://en.wikipedia.org/wiki/Brent%27s_method)** It takes as an argument the function $f(x)$ whose root is to be found, and the left and right bounds for the root. We can write a function to find the fixed point for given k_0 and n.

In [155]:

```
# Compute the fixed point for various k0 given n=10
k0 = np.linspace(0.1, 4, 200)
p_fp = [fixed_point(b, 10) for b in k0]

# Make the plot
bplt = bokeh.plotting.figure(title='finding fixed point, p0 over k0=[0,4] for n =10',width=400, height=400,
                             x_axis_label='k0',
                             y_axis_label='fixed point, p0')
bplt.line(k0, p_fp, line_width=2)

bokeh.io.show(bplt)
```

In [151]:

```
# Other Solution

simple_update = circuit_update

def simple_propensity_q3(propensities, conc, t, kl, kappa_r, kappa_p, gamma_p, gamma_r, k):
    """Updates an array of propensities given a set of parameters
    and an array of populations.
    """
    # Unpack population
    r, p = conc

    phi = .01/(1+(p/k)**10)

    # Update propensities
    propensities[0] = kl+phi       # make r
    propensities[1] = gamma_r*r    # degrade r
    propensities[2] = kappa_p*r    # make p
    propensities[3] = gamma_p*p    # degrade p

def getSystemParameters():
    return ()


def sample_discrete_q3(probs):
    """Randomly sample an index with probability given by probs."""
    # Generate random number
    q = np.random.rand()

    # Find index
    i = 0
    p_sum = 0.0
    while p_sum < q:
        p_sum += probs[i]
        i += 1
    return i - 1


def gillespie_draw_q3(propensity_func_q3, propensities, conc, t, args=()):

    # Compute propensities
    propensity_func_q3(propensities, conc, t, *args)

    # Sum of propensities
    props_sum = propensities.sum()

    # Compute next time
    time = np.random.exponential(1.0 / props_sum)

    # Compute discrete probabilities of each reaction
    rxn_probs = propensities / props_sum

    # Draw reaction from this distribution
    rxn = sample_discrete_q3(rxn_probs)

    return rxn, time


def gillespie_ssa_q3(propensity_func_q3, update, conc_0, time_points, args=()):

    # Initialize output
    conc_out = np.empty((len(time_points), update.shape[1]), dtype=np.int)

    # Initialize and perform simulation
    i_time = 1
    i = 0
    t = time_points[0]
    conc = conc_0.copy()
```

```
    conc = conc_0.copy()
    conc_out[0,:] = conc
    propensities = np.zeros(update.shape[0])
    while i < len(time_points):
        while t < time_points[i_time]:
            # draw the event and time step
            event, dt = gillespie_draw_q3(propensity_func_q3, propensities, conc, t, args)

            # Update the population
            conc_previous = conc.copy()
            conc += update[event,:]

            # Increment time
            t += dt

        # Update the index
        i = np.searchsorted(time_points > t, True)

        # Update the population
        conc_out[i_time:min(i,len(time_points))] = conc_previous

        # Increment index
        i_time = i

    return conc_out

def exactStationaryDistribution(propensity_func_q3, update, conc_0, time_points, args=()):

    # Initialize output
    conc_out = np.empty((len(time_points), update.shape[1]), dtype=np.int)

    # Initialize and perform simulation
    i_time = 1
    i = 0
    t = time_points[0]
    conc = conc_0.copy()
    conc_out[0,:] = conc
    propensities = np.zeros(update.shape[0])
    while i < len(time_points):
        while t < time_points[i_time]:
            # draw the event and time step
            event, dt = gillespie_draw_q3(propensity_func_q3, propensities, conc, t, args)

            # Update the population
            conc_previous = conc.copy()
            conc += update[event,:]

            # Increment time
            t += dt

        # Update the index
        i = np.searchsorted(time_points > t, True)

        # Update the population
        conc_out[i_time:min(i,len(time_points))] = conc_previous

        # Increment index
        i_time = i


    return (1 - propensities.reshape(2,2)) * propensities.reshape(2,2) ** update[event,:]
```

In [86]:

```
def plot_q3results(q3results):

    results = q3results

    # Set up plots
    plots = [bokeh.plotting.figure(plot_width=300,
                                   plot_height=200,
                                   x_axis_label='time [minutes]',
                                   y_axis_label='mRNA Abundance'),
             bokeh.plotting.figure(plot_width=300,
                                   plot_height=200,
                                   x_axis_label='time [minutes]',
                                   y_axis_label='protein Abundance')]

    # Plot trajectories and mean
    for i in [0, 1]:
        for x in results[:,:,i]:
            plots[i].line(time_points, x, line_width=0.5,
                          alpha=0.2, line_join='bevel')
        plots[i].line(time_points, results[:,:,i].mean(axis=0),
                      line_width=3, color='orange', line_join='bevel')

    # Link axes
    plots[0].x_range = plots[1].x_range

    bokeh.io.show(bokeh.layouts.gridplot(plots, ncols=2))
```

Case1: Strong Regulation, k = 100

In [26]:

```
plot_q3results(q3results)
```

Case 2: Weak Regulation, k = 10000

In [27]:

```
#### Case2: Weak Regulation, k = 10000
k = 10000

args = (kl, kr, kp, yp, yr, k)
time_points = np.linspace(0, 60*8, 60)
conc_0 = np.array([0, 0], dtype=int)
n_traj = 50

# Seed random number generator for reproducibility
from random import seed
some_seed = 43210
np.random.seed = some_seed

# Initialize output array
q3resultsWeak = np.empty((n_traj, len(time_points), 2), dtype=int)

# Run the calculations
for i in range(n_traj):
    q3resultsWeak[i,:,:] = gillespie_ssa_q3(simple_propensity_q3, simple_update,
                                conc_0, time_points, args=args)
plot_q3results(q3resultsWeak)
```

In [28]:

```
q3resultsWeak.shape
```

Out[28]:

```
(50, 60, 2)
```

In [33]:

```
print('mean protein copy number (CASE tight) =', q3results[:,-60:,1].mean())
print('mean protein copy number (CASE weak)=', q3resultsWeak[:,-60:,1].mean())

print('\n[p] variance (CASE tight)=', q3results[:,-60:,1].std()**2)
print('[p] variance (CASE weak)=', q3resultsWeak[:,-60:,1].std()**2)

print('\n[X] noise (CASE tight)=', q3results[:,-60:,1].std() / q3results[:,-60:,1].mean())
print('[X] noise (CASE weak)=', q3resultsWeak[:,-60:,1].std() / q3resultsWeak[:,-60:,1].mean())
```

```
mean protein copy number (CASE tight) = 35.935
mean protein copy number (CASE weak)= 32.98433333333333

[p] variance (CASE tight)= 1468.0801083333333
[p] variance (CASE weak)= 1437.664087888889

[X] noise (CASE tight)= 1.066245540902045
[X] noise (CASE weak)= 1.1495318308340219
```