In [1]:

```
from scipy.integrate import odeint
import matplotlib.pyplot as plt
import numpy as np
import random
import scipy.stats as ss
import scipy.special as sp
import pickle
import pandas as pd
import seaborn as sns
import time
from numba import jit
import matplotlib

%matplotlib inline

# import os
# import imageio # to make gif of heatmaps in time

matplotlib.rcParams["pdf.fonttype"] = 42
matplotlib.rcParams["ps.fonttype"] = 42
rng = np.random.default_rng(2021)  # create a new random number generator
```

In [2]:

```
# General parameters for this set of experiments:
bacteria = 2                    # Number of bacterial strains
repeats = 3                     # Number of experiment repeats
days = 100                      # Duration of experiment (days), T_tot
passage = 0.001                 # Dilution factor, D
d_S_antimicrobial = 40          # Antimicrobial-induced decrease in the growth rate of susceptible bacteria, d_Smax
mix_amount = 0.10               # Mixing factor, M

# Parameters for initial conditions:
r_S = 2.1*24                    # Intrinsic growth rate of susceptible bacteria, m_Smax
K_S = 1e9                       # Carrying capacity, K
K_R = 1e9                       # Carrying capacity, K
a_SR = 1                        # Competition coefficient, a
a_RS = 1                        # Competition coefficient, a
total = 1e6                     # Initial concentration of bacterial inoculum (sum of both strains) at the beginning of the sim
```

In [3]:

```
wells = 100                     # Number of hosts
S_wells = 0                     # Number of hosts with only S bacteria at beginning of simulation
p0 = np.array([1.0] * S_wells + [0.5] * (wells-S_wells))
rng.shuffle(p0)
```

In [4]:

```
# # TIME for ode:
steps = 24
t = np.linspace(0, 1, steps)

# TIME for experiment:
d = 1
b = 0
d2 = d
t_list = []
for k in range(0, days):
    t_k = np.linspace(b, d2, steps)
    b = d2
    d2 += d
    t_list += t_k.tolist()

# Fixing the time:
t_array = np.array(t_list)
t_array_long = np.insert(t_array, 0, t_array[0])
t_array_steps_long = t_array_long[::steps]
```

In [5]:

```
# Parameters for experiments:
mixing_frequency = np.linspace(0, 19, 20).astype(int)        # Mixing intervals, I_trans
antimicrobial_frequency = np.linspace(0, 19, 20).astype(int) # Treatment intervals, I_antim
```

In [6]:

```
# # LOTKA-VOLTERA COMPETITION FUNCTION:
@jit
def Lotka_Voltera(z, t, d_Smax, d_Rmax, r_S, r_R, K_S, K_R, a_SR, a_RS):
    S, R = z
    dSdt = S * (r_S-d_Smax) * (1 - ((S + a_SR * R) / K_S))
    dRdt = R * (r_R-d_Rmax) * (1 - ((R + a_RS * S) / K_R))
    dzdt = [dSdt, dRdt]
    return dzdt
```

In [7]:

```
class Experiment:

    # instructions: ALWAYS call 'run_experiment()' first, and then choose the one/some of the other methods for output.


    def __init__(self, antimicrobial_mu, mix_freq):
```

```
        self.mix_pop = 1.0
        self.antimicrobial_mu = antimicrobial_mu
        self.mix_freq = mix_freq
        self.r_R = 0.7 * r_S
        self.d_R_antimicrobial = 0.05*self.r_R


    def run_experiment(self):
        # Create array to store ALL solutions:
        z = np.empty((repeats, days, wells, steps, bacteria))
        coexist = np.empty((repeats,days*steps+1))


        for r in range(0,repeats):
            S0 = p0 * total           # Initial number of susceptible bacteria, S_0
            R0 = total - S0           # Initial number of resistant bacteria, R_0
            z0 = np.array([S0, R0]).T
            z0_mix = np.empty((np.shape(z0)))

            # ANTIBIOTIC:
            if self.antimicrobial_mu == 0:
                antimicrobial_day = np.full(wells, (days+1))
            else:
                antimicrobial_day = rng.integers(0, self.antimicrobial_mu, size=wells) # Choosing a random first day for antibi

            # MIXING:
            pairs = []
            if self.mix_freq == 0:
                mixing_days = list(range(days+1, days+1))
            else:
                mixing_days = list(range(self.mix_freq, days+1))[::self.mix_freq]
                for k in range(0, days):             # make random pairs for mixing
                    if k in mixing_days:
                        lst = list(range(1, wells+1))
                        day_pairs = []
                        for j in range(0, int(self.mix_pop*(wells//2))):
                            pair = tuple(rng.choice(lst, size = 2, replace = False))
                            day_pairs.append(pair)
                            lst = [x for x in lst if x not in pair]
                        pairs.append(day_pairs)
                    else:
                        pairs.append([None])


            for k in range(0, days): # k: keeps track of days, every loop is one day

                for w in range(0, wells): # w: keeps track of wells, every loop is one well

                    # Set decrease in intrinsic growth rate by antibiotic (d_Smax, d_Rmax) if it is an antibiotic day
                    if k == antimicrobial_day[w]:
                        d_Smax = d_S_antimicrobial
                        d_Rmax = self.d_R_antimicrobial
                    else:
                        d_Smax = 0
                        d_Rmax = 0

                    # Solve ODE
                    par = (d_Smax, d_Rmax, r_S, self.r_R, K_S, K_R, a_SR, a_RS)
                    z[r][k][w] = odeint(Lotka_Voltera, z0[w], t, args=par)
                    z0[w] = z[r][k][w][-1]*passage # Passage to the next day
                    z0[w][z0[w] < 1] = 0

                    if k == antimicrobial_day[w]: # If treatment happened, choose day for next treatment
                        antimicrobial_day[w] += self.antimicrobial_mu


                if k in mixing_days:
                    l = len(pairs[k])
                    for p in range(0, l):
                        p1 = pairs[k][p][0]
                        p2 = pairs[k][p][1]

                        z0_mix[p1-1] = (z0[p1-1] * (1 - mix_amount)) + (z0[p2-1] * mix_amount)
                        z0_mix[p2-1] = (z0[p2-1] * (1 - mix_amount)) + (z0[p1-1] * mix_amount)

                        z0[p1-1] = z0_mix[p1-1]
                        z0[p2-1] = z0_mix[p2-1]

                        z0[p1-1][z0[p1-1] < 1] = 0
                        z0[p2-1][z0[p2-1] < 1] = 0

        z_T_flat = z.transpose(0, 2, 4, 1, 3).reshape(repeats, wells, bacteria, days*steps)
        z_sol = np.insert(z_T_flat, 0, z_T_flat[:,:,:,0], axis=3)

        S = z_sol[:,:,0]
        R = z_sol[:,:,1]

        S_fraction = S / (S + R)
        R_fraction = R / (S + R)
```

```
        for r in range(0,repeats):
            for k in range(0, days*steps+1):
                coexist[r][k] = ((0.01 < S_fraction[r,:,k]) & (S_fraction[r,:,k] < 0.96)).sum()

        # Arithmetic mean
        # arithmetic mean of the wells of each repeat
        S_fractions_amean_w = np.mean(S_fraction, axis = 1)
        R_fractions_amean_w = np.mean(R_fraction, axis = 1)
        # arithmetic mean of the ameans of each repeats of each experiment
        S_fractions_amean_wr = np.mean(S_fractions_amean_w, axis = 0)
        R_fractions_amean_wr = np.mean(R_fractions_amean_w, axis = 0)
        # standard deviation of repeats of each experiment
        S_fractions_std_wr = np.std(S_fractions_amean_w, axis = 0)
        R_fractions_std_wr = np.std(R_fractions_amean_w, axis = 0)

        coexist_amean = np.mean(coexist, axis = 0)

        # to use in other methods
        self.S = S
        self.R = R
        self.S_fraction = S_fraction
        self.R_fraction = R_fraction
        # new amean
        self.S_fractions_amean_w = S_fractions_amean_w
        self.R_fractions_amean_w = R_fractions_amean_w
        self.S_fractions_amean_wr = S_fractions_amean_wr
        self.R_fractions_amean_wr = R_fractions_amean_wr
        self.S_fractions_std_wr = S_fractions_std_wr
        self.R_fractions_std_wr = R_fractions_std_wr

        self.coexist_amean = coexist_amean

    def ODE_solutions(self):
        return self.S, self.R

    def fractions(self):
        return self.S_fraction, self.R_fraction

    def amean_w(self):
        return self.S_fractions_amean_w, self.R_fractions_amean_w

    def amean_wr(self):
        return self.S_fractions_amean_wr, self.R_fractions_amean_wr

    def amean_SD_wr(self):
        return self.S_fractions_std_wr, self.R_fractions_std_wr

    def amean_step_wr(self):
        return self.S_fractions_amean_wr[::steps], self.R_fractions_amean_wr[::steps]

    def amean_SD_step_wr(self):
        return self.S_fractions_std_wr[::steps], self.R_fractions_std_wr[::steps]

    def coexistence(self):
        return self.coexist_amean

    def coexistence_step(self):
        return self.coexist_amean[::steps]


# The methods return tuples. To use the tuples:
        # either use indexing to choose the values needed: e.g. exp1.amean()[0] to get self.S_fractions_amean_r
        # or unpack it: e.g. S_amean, R_amean = exp1.amean()
```

In [8]:

```
# Make dataframes to save parameters and results
df_param = pd.DataFrame(columns=('antim_mu', 'mix_freq'))
df_amean_S = pd.DataFrame(columns=(t_array_steps_long))
df_amean_R = pd.DataFrame(columns=(t_array_steps_long))
df_ameanSD_S = pd.DataFrame(columns=(t_array_steps_long))
df_ameanSD_R = pd.DataFrame(columns=(t_array_steps_long))
df_coexist_S = pd.DataFrame(columns=(t_array_steps_long))
```

In [9]:

```
start_time = time.time()

row = 0

for af in antimicrobial_frequency:
    for mf in mixing_frequency:
        df_param.loc[row] = [af,mf]
        exp = Experiment(af,mf)
        exp.run_experiment()
        df_amean_S.loc[row], df_amean_R.loc[row] = exp.amean_step_wr()
        df_ameanSD_S.loc[row], df_ameanSD_R.loc[row] = exp.amean_SD_step_wr()
        df_coexist_S.loc[row] = exp.coexistence_step()
        row += 1

print(f"My program, with {wells} wells, {days} days, {repeats} repeats, {row} experiments, took {time.time() - start_time} seco
```

```
My program, with 100 wells, 100 days, 3 repeats, 400 experiments, took 5801.8328948020935 seconds to run
```

In [10]:

```
# Save in the same folder
df_param.to_pickle("./df_param.pkl")
df_amean_S.to_pickle("./df_amean_S.pkl")
df_amean_R.to_pickle("./df_amean_R.pkl")
df_ameanSD_S.to_pickle("./df_ameanSD_S.pkl")
df_ameanSD_R.to_pickle("./df_ameanSD_R.pkl")
df_coexist_S.to_pickle("./df_coexist_S.pkl")
```

In [11]:

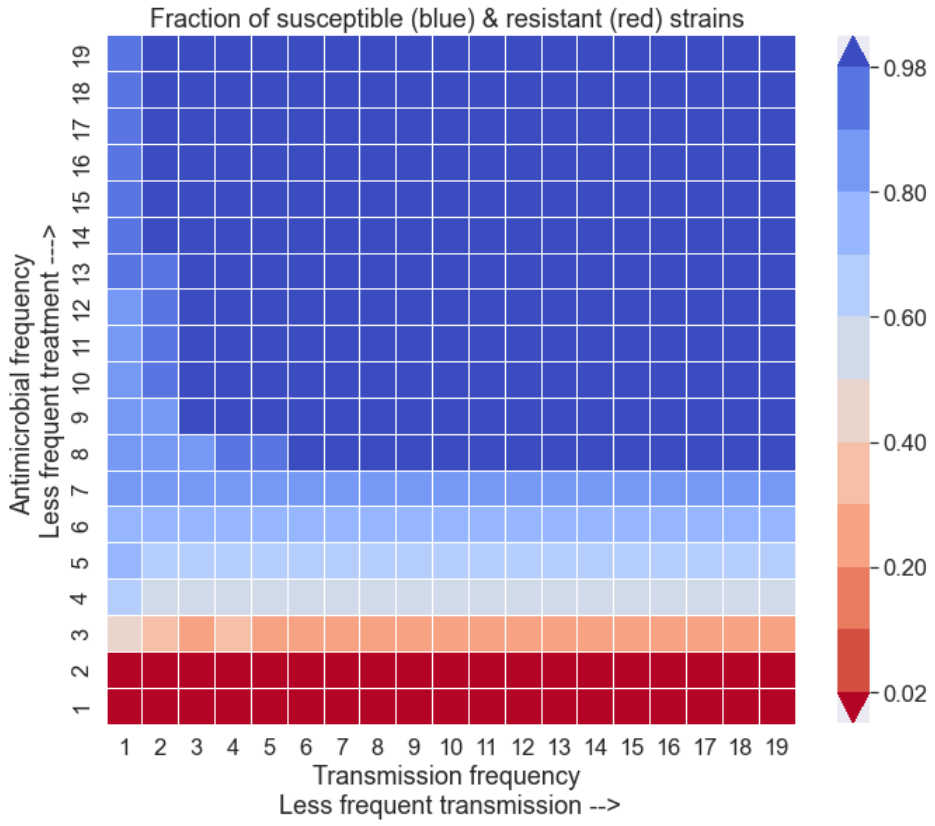
```
# # Retrieve from same folder
df_amean_S = pd.read_pickle("./df_amean_S.pkl")
df_amean_R = pd.read_pickle("./df_amean_R.pkl")
df_ameanSD_S = pd.read_pickle("./df_ameanSD_S.pkl")
df_ameanSD_R = pd.read_pickle("./df_ameanSD_R.pkl")
df_param = pd.read_pickle("./df_param.pkl")
df_coexist_S = pd.read_pickle("./df_coexist_S.pkl")
```

In [12]:

```
sns.set(font_scale = 1.5)
df_S_indx = df_amean_S.set_index(pd.MultiIndex.from_frame(df_param))
df_S_indx['mean'] = df_S_indx.iloc[:, 50:100].mean(axis=1)
S_af_rf= df_S_indx['mean'].unstack(level=-1).drop(labels=0, axis=0).drop(labels=0, axis=1)

cmap = matplotlib.cm.coolwarm_r
bounds = [0.02, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 0.98]
norm = matplotlib.colors.BoundaryNorm(bounds, cmap.N, extend='both')

fig, ax = plt.subplots(figsize=(10,10), tight_layout=True)
ax = sns.heatmap(S_af_rf, cmap=cmap, norm=norm, xticklabels = True, yticklabels=True,
                 vmin=0, vmax=1, square=True, linewidths=.5, cbar_kws={"shrink": .72})
ax.invert_yaxis()
ax.set_ylabel('Antimicrobial frequency \nLess frequent treatment --->')
ax.set_xlabel('Transmission frequency \nLess frequent transmission -->')
plt.title('Fraction of susceptible (blue) & resistant (red) strains ')
plt.savefig('heatmap1_average.pdf',  transparent=True)
plt.savefig('heatmap1_average.png',  transparent=True)
plt.show()
```



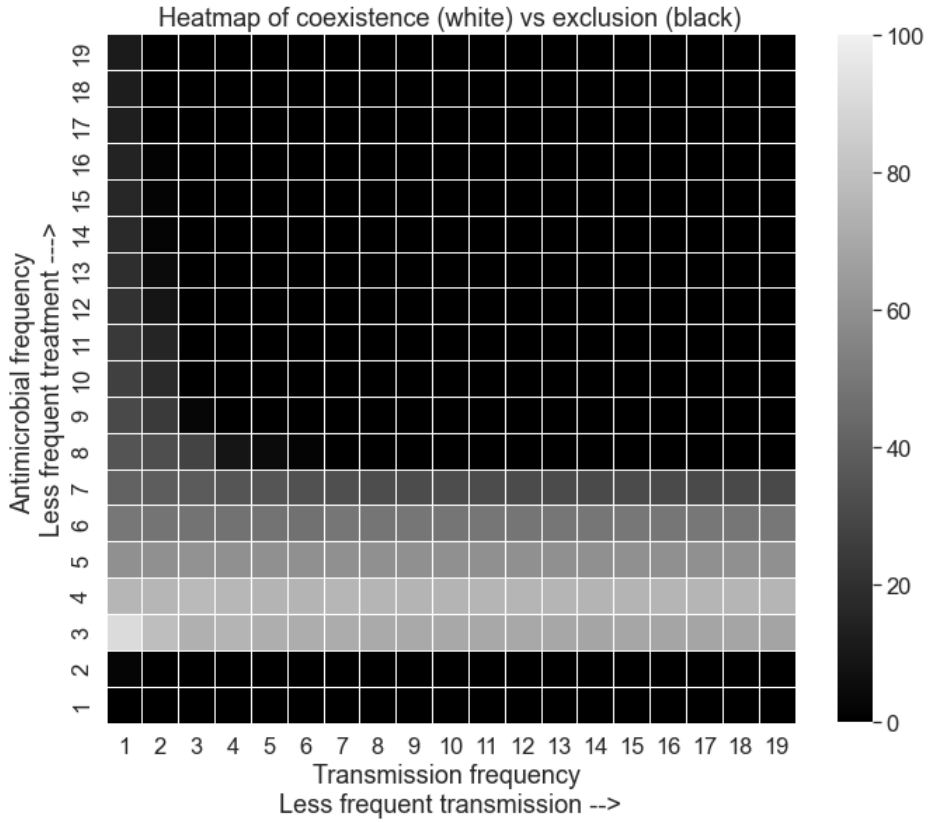
In [13]:

```
df_coex_indx = df_coexist_S.set_index(pd.MultiIndex.from_frame(df_param))
df_coex_indx['mean'] = df_coex_indx.iloc[:, 50:100].mean(axis=1)
coex_af_rf= df_coex_indx['mean'].unstack(level=-1).drop(labels=0, axis=0).drop(labels=0, axis=1)

cmap = sns.light_palette("black", reverse = True, as_cmap=True)

fig, ax = plt.subplots(figsize=(10,10), tight_layout=True)
ax= sns.heatmap(coex_af_rf, cmap=cmap, xticklabels = True,
                vmin=0, vmax=100, yticklabels=True, square=True, linewidths=.5, cbar_kws={"shrink": .72})
ax.invert_yaxis()
ax.set_ylabel('Antimicrobial frequency \nLess frequent treatment --->')
ax.set_xlabel('Transmission frequency \nLess frequent transmission -->')
plt.title('Heatmap of coexistence (white) vs exclusion (black)')
```

```
plt.savefig('heatmap2_average.pdf',  transparent=True)
plt.savefig('heatmap2_average.png',  transparent=True)
plt.show()
```



In [14]:

```
sns.set_style("white")
fig, ax = plt.subplots(1,1, figsize=(15,7), tight_layout=True, dpi=100)

for i in range (0,400):
    ax.plot(t_array_steps_long, df_amean_S.iloc[i])
ax.set_xlabel('Time (days)')
ax.set_ylabel('Ratio of susceptible')
ax.grid()
#ax.set_ylim(0,1)
plt.savefig('timeplot.pdf',  transparent=True)
plt.savefig('timeplot.png',  transparent=True)
```