ClimatePoliciesUnderWealthInequality

December 31, 2024

0.1 Climate Policies Under Wealth Inequality

This notebook is meant to reproduce the results shown in the paper "Climate policies under wealth inequality", linked here: https://www.pnas.org/doi/10.1073/pnas.1323479111.

Abstract of the paper: (verbatim) Taming the planet's climate requires cooperation. Previous failures to reach consensus in climate summits have been attributed, among other factors, to conflicting policies between rich and poor countries, which disagree on the implementation of mitigation measures. Here we implement wealth inequality in a threshold public goods dilemma of cooperation in which players also face the risk of potential future losses. We consider a population exhibiting an asymmetric distribution of rich and poor players that reflects the present-day status of nations and study the behavioral interplay between rich and poor in time, regarding their willingness to cooperate. Individuals are also allowed to exhibit a variable degree of homophily, which acts to limit those that constitute one's sphere of influence. Under the premises of our model, and in the absence of homophily, comparison between scenarios with wealth inequality and without wealth inequality shows that the former leads to more global cooperation than the latter. Furthermore, we find that the rich generally contribute more than the poor and will often compensate for the lower contribution of the latter. Contributions from the poor, which are crucial to overcome the climate change dilemma, are shown to be very sensitive to homophily, which, if prevalent, can lead to a collapse of their overall contribution. In such cases, however, we also find that obstinate cooperative behavior by a few poor may largely compensate for homophilic behavior.

To achieve the results of the paper, we have created a Public Goods Game (PGG) to model the climate policies game.

[2]:
```
#!pip install numpy matplotlib > /dev/null 2>&1
```

[1]:
```
import numpy as np
import random
import matplotlib.pyplot as plt
from math import comb
from scipy.stats import multivariate_hypergeom
from egttools.utils import calculate_stationary_distribution
```

[2]:
```
class ClimateThresholdPublicGoodsGame:
    def __init__(self,
                 population_size = 200,
                 rich_fraction = 0.2,
                 endowment_rich = 2.5,
```

```
                 endowment_poor = 0.625,
                 group_size = 6,
                 threshold = 3,
                 risk = 0.6,
                 contribution_factor = 0.1,
                 homophily = 0.4,
                 beta = 5,
                 mu = 0.1,
                 verbose = False,
                 pi_max = float('inf'),
                 gos_max = float('inf')
                ):
        self.Z = population_size

        self.rich_fraction = rich_fraction
        self.endowment_rich = endowment_rich
        self.endowment_poor = endowment_poor
        self.N = group_size

        self.threshold = threshold
        self.risk = risk
        self.contribution_factor = contribution_factor
        self.homophily = homophily
        self.beta = beta
        self.mu = mu

        self.verbose = verbose

        # We can calculate these right away!
        self.Z_R = int(rich_fraction * population_size)
        self.Z_P = population_size - self.Z_R

        assert endowment_rich >= endowment_poor, "The Marxists are back! Rise␣
↪of the proletariat!"

        # Average endowment is used to calculate the threshold for success
        self.average_endowment = (self.Z_R * endowment_rich + self.Z_P *␣
↪endowment_poor) / population_size

        if self.verbose:
            print("Average Endowment", self.average_endowment)
            print("Mcb", self.average_endowment * self.threshold * self.
↪contribution_factor)

        # Create/clear the cache dict for storing fitness
        self.fitness_dict = dict()
        self.stationary_distribution = None
```

```python
        self.transition_matrix = None

        self.pi_max = pi_max
        self.gos_max = gos_max

        self.generate_states()

    def transform_to_scalar(self, x, y):
        """
        Transform a 2D value (x, y) into a scalar using modulo transformation␣
↪with Z
        """
        scalar = x * self.Z + y
        return scalar

    def transform_to_2d(self, scalar):
        """
        Transform a scalar back into a 2D value (x, y) using modulo␣
↪transformation with Z
        """
        x = scalar // self.Z
        y = scalar % self.Z
        return x, y

    def generate_states(self):
        """
        Generate all possible states (C_R, D_R, C_P, D_P)

        Could also make this a generator that yields states!

        Parameters (implicit):
        Z_R : int - Total number of rich individuals
        Z_P : int - Total number of poor individuals

        Returns:
        dict - Dict of all possible states (C_R, C_P) = (index, scalar(C_R,␣
↪C_P))
        """
        self.states = []
        for C_R in range(self.Z_R + 1):
            for C_P in range(self.Z_P + 1):
                # D_R and D_P can be derived from these since Z_R and Z_P are␣
↪constants
                # if C_R >= self.O_R and C_P >= self.O_P:
                self.states.append((C_R, C_P))
```

```
        # Now build 2D-state <-> scalar representation so we can switch back␣
↪and forth easily
        self.indexed_states = dict()
        total = self.Z
        for index, (C_R, C_P) in enumerate(self.states):
            #if C_R >= self.O_R and C_P >= self.O_P:
            # Format: (C_R, C_P) = (index, scalar(C_R, C_P))
            self.indexed_states[(C_R, C_P)] = (index, self.
↪transform_to_scalar(C_R, C_P))

        # Print everything for sanity
        # for (C_R, C_P), (index, scalar) in self.indexed_states.items():
        #     print(f"C_R, C_P = ({C_R}, {C_P}) \t=>\t Index: {index},\t Scalar␣
↪(Mod Z): {scalar} \t 2D: {self.transform_to_2d(scalar)}")

        return self.states

    def transition_probability(self,
                               i_k_X,
                               i_k_Y,
                               i_l_X,
                               i_l_Y,
                               Z_k,
                               f_k_X,
                               f_k_Y,
                               f_l_X,
                               f_l_Y):
        """
        Calculate the transition probability T_k^{X -> Y} for a subpopulation k

        This function replicates the equation for transition T_k^{X -> Y}
        mentioned on page 2 of the SI text!

        The pairwise comparison is built into this function. Also the use of␣
↪homophily to determine evolution!

        Parameters:
        i_k_X : int - Number of individuals with strategy X in subpopulation k
        i_k_Y : int - Number of individuals with strategy Y in subpopulation k
        i_l_X : int - Number of individuals with strategy X in subpopulation l
        i_l_Y : int - Number of individuals with strategy Y in subpopulation l
        Z_k : int - Total size of subpopulation k. We can get Z_l from this
        f_k_X : float - Fitness of individuals with strategy X in subpopulation␣
↪k
        f_k_Y : float - Fitness of individuals with strategy Y in subpopulation␣
↪k
```

```
        f_l_X : float - Fitness of individuals with strategy X in subpopulation␣
↪l
        f_l_Y : float - Fitness of individuals with strategy Y in subpopulation␣
↪l

        Implicit params:
        h : float - Homophily parameter
        mu : float - Mutation probability
        beta : float - Intensity of selection

        Returns:
        float - Transition probability T_k^{X -> Y}
        """
        # Create a unique key for memoization
        key = (i_k_X, i_k_Y, i_l_X, i_l_Y, Z_k, f_k_X, f_k_Y, f_l_X, f_l_Y)

        # Check if the result is already cached
        # if key in self.cache:
        #     return self.cache[key]

        Z_l = self.Z - Z_k

        # Non-mutation probability
        non_mutation_prob = (1 - self.mu) * (
            (i_k_Y / (Z_k - 1 + (1 - self.homophily) * Z_l)) * (1 / (1 + np.
↪exp(self.beta * (f_k_X - f_k_Y))))
            + ((1 - self.homophily) * i_l_Y / (Z_k - 1 + (1 - self.homophily) *␣
↪Z_l)) * (1 / (1 + np.exp(self.beta * (f_k_X - f_l_Y))))
        )
        # The second exp arg is:
        # (1 + np.exp(beta * (f_k_X - f_l_Y)))
        # Because we are comparing "our" strategy to the same strategy in the␣
↪other population (rich/poor)
        # based on homophily, so if h=1, then that whole part is "cancelled" out

        # Mutation probability
        mutation_prob = self.mu

        # Total transition probability
        result = i_k_X / Z_k * (non_mutation_prob + mutation_prob)

        # Store in cache, so we don't calculate this again
        #self.cache[key] = result

        return result

    def calculate_fitness_rich_C(self, i_R, i_P):
```

```
        """
        Calculate the fitness of rich cooperators (f_R^C) for a given␣
↪configuration

        This and the below functions implement the fitness equations on page 1␣
↪in the SI text!

        Parameters:
        i_R : int - Number of rich cooperators
        i_P : int - Number of poor cooperators

        Implicit params:
        Z : int - Total population size
        N : int - Group size
        payoff_function : function - Function to compute payoff Pi_R^C(j_R + 1,␣
↪j_P)

        Returns:
        float - Fitness of rich cooperators
        """
        # Return cached result if it exists
        # key = (1, 1, i_R, i_P) # 1, 1 = Rich C
        # if key in self.fitness_dict:
        #     return self.fitness_dict[key]

        # Total ways to sample a group of size N - 1 from Z - 1 population
        normalization_factor = comb(self.Z - 1, self.N - 1)
        # total_ways = 0

        fitness = 0.0
        # Iterate over all possible group compositions
        for j_R in range(min(i_R, self.N) + 1):  # Rich cooperators in the group
            for j_P in range(min(i_P, self.N - j_R) + 1):  # Poor cooperators␣
↪in the group
                if self.N - 1 - j_R - j_P < 0 or self.N - 1 - j_R - j_P > self.
↪Z - i_R - i_P or i_R - 1 < 0:  # Invalid group composition
                    continue

                # Compute hypergeometric probabilities!
                prob = (comb(i_R - 1, j_R) *
                        comb(i_P, j_P) *
                        comb(self.Z - i_R - i_P, self.N - 1 - j_R - j_P)) /␣
↪normalization_factor

                # total_ways += ways
```

```python
                # Compute payoff for a rich cooperator in this group
                payoff = self.payoff_function(1, 1, j_R + 1, j_P) # 1-rich, 1-C

                # Add contribution to fitness
                fitness += prob * payoff

        # if total_ways > 0:
        #     fitness /= total_ways

        # self.fitness_dict[key] = fitness

        return fitness

    def calculate_fitness_rich_D(self, i_R, i_P):
        """
        Calculate the fitness of rich defectors (f_R^D) for a given␣
↪configuration

        Parameters:
        i_R : int - Number of rich cooperators
        i_P : int - Number of poor cooperators
        Z : int - Total population size
        N : int - Group size
        payoff_function : function - Function to compute payoff Pi_R^D(j_R, j_P)

        Returns:
        float - Fitness of rich defectors
        """
        # Return cached result if it exists
        # key = (1, 0, i_R, i_P) # 1, 0 = Rich D
        # if key in self.fitness_dict:
        #     return self.fitness_dict[key]

        # Total ways to sample a group of size N - 1
        normalization_factor = comb(self.Z - 1, self.N - 1)
        # total_ways = 0

        fitness = 0.0
        # Iterate over all possible group compositions
        for j_R in range(min(i_R, self.N) + 1):  # Rich cooperators in the group
            for j_P in range(min(i_P, self.N - j_R) + 1):  # Poor cooperators␣
↪in the group
                if self.N - 1 - j_R - j_P < 0 or self.N - 1 - j_R - j_P > self.
↪Z - 1 - i_R - i_P:  # Invalid group composition
                    continue

                # Compute hypergeometric probabilities
```

```
                prob = (comb(i_R, j_R) * comb(i_P, j_P) *
                        comb(self.Z - 1 - i_R - i_P, self.N - 1 - j_R - j_P)) /␣
↪normalization_factor

                # total_ways += ways

                # Compute payoff for a rich defector in this group
                payoff = self.payoff_function(1, 0, j_R, j_P) # 1-rich, 0-D

                # Add contribution to fitness
                fitness += prob * payoff

        # if total_ways > 0:
        #     fitness /= total_ways

        # self.fitness_dict[key] = fitness

        return fitness

    def calculate_fitness_poor_C(self, i_R, i_P):
        """
        Calculate the fitness of poor cooperators (f_P^C) for a given␣
↪configuration

        Parameters:
        i_R : int - Number of rich cooperators
        i_P : int - Number of poor cooperators
        Z : int - Total population size
        N : int - Group size
        payoff_function : function - Function to compute payoff Pi_P^C(j_R, j_P␣
↪+ 1)

        Returns:
        float - Fitness of poor cooperators
        """
        # Return cached result if it exists
        # key = (0, 1, i_R, i_P) # 0, 1 = Poor C
        # if key in self.fitness_dict:
        #     return self.fitness_dict[key]

        # Total ways to sample a group of size N - 1
        normalization_factor = comb(self.Z - 1, self.N - 1)
        # total_ways = 0

        fitness = 0.0
        # Iterate over all possible group compositions
        for j_R in range(min(i_R, self.N) + 1):  # Rich cooperators in the group
```

```python
            for j_P in range(min(i_P, self.N - j_R) + 1):  # Poor cooperators␣
↪in the group
                # Number of defectors in the group
                j_D = self.N - 1 - j_R - j_P
                if j_D < 0 or j_D > self.Z - i_R - i_P or i_P - 1 < 0:  #␣
↪Invalid group composition
                    continue

                # Compute hypergeometric probabilities
                prob = (comb(i_R, j_R) * comb(i_P - 1, j_P) *
                        comb(self.Z - i_R - i_P, j_D)) / normalization_factor

                # total_ways += ways

                # Compute payoff for a poor cooperator in this group
                payoff = self.payoff_function(0, 1, j_R, j_P + 1) # 0-poor, 1-C

                # Add contribution to fitness
                fitness += prob * payoff

        # if total_ways > 0:
        #     fitness /= total_ways

        # self.fitness_dict[key] = fitness

        return fitness

    def calculate_fitness_poor_D(self, i_R, i_P):
        """
        Calculate the fitness of poor defectors (f_P^D) for a given␣
↪configuration

        Parameters:
        i_R : int - Number of rich cooperators
        i_P : int - Number of poor cooperators
        Z : int - Total population size
        N : int - Group size
        payoff_function : function - Function to compute payoff Pi_P^D(j_R, j_P)

        Returns:
        float - Fitness of poor defectors
        """
        # Return cached result if it exists
        # key = (0, 0, i_R, i_P) # 0, 0 = Poor D
        # if key in self.fitness_dict:
        #     return self.fitness_dict[key]
```

```python
        # Total ways to sample a group of size N - 1
        normalization_factor = comb(self.Z - 1, self.N - 1)
        # total_ways = 0

        fitness = 0.0
        # Iterate over all possible group compositions
        for j_R in range(min(i_R, self.N) + 1):  # Rich cooperators in the group
            for j_P in range(min(i_P, self.N - j_R) + 1):  # Poor cooperators in the group
                # Number of defectors in the group
                j_D = self.N - 1 - j_R - j_P
                if j_D < 0 or j_D > self.Z - 1 - i_R - i_P:  # Invalid group composition
                    continue

                # Compute hypergeometric probabilities
                prob = (comb(i_R, j_R) * comb(i_P, j_P) *
                        comb(self.Z - 1 - i_R - i_P, j_D)) / normalization_factor

                # total_ways += ways

                # Compute payoff for a poor defector in this group
                payoff = self.payoff_function(0, 0, j_R, j_P) # 0-poor, 0-D

                # Add contribution to fitness
                fitness += prob * payoff

        # if total_ways > 0:
        #     fitness /= total_ways

        # self.fitness_dict[key] = fitness

        return fitness

    def payoff_function(self, individual_type, strategy, j_R, j_P):
        """
        Compute the payoff for an individual based on their type and strategy in a group

        Parameters:
        - individual_type : bool - 1/0 for rich/poor, the type of the individual
        - strategy : bool - 1/0 for C/D, the strategy of the individual
        - j_R : int - Number of rich cooperators in the group
        - j_P : int - Number of poor cooperators in the group
        - endowment_rich : float - Endowment of rich individuals
        - endowment_poor : float - Endowment of poor individuals
```

```
        - threshold : float - Contribution threshold for group success
        - contribution_factor : float - Fraction of endowment contributed by␣
↪cooperators

        Returns:
            The payoff for the individual
        """
        # Determine the endowment of the individual
        endowment = self.endowment_rich if individual_type else self.
↪endowment_poor

        # Total contributions in the group
        contributions = j_R * self.endowment_rich * self.contribution_factor +␣
↪j_P * self.endowment_poor * self.contribution_factor
        required_contribution = self.threshold * self.contribution_factor *␣
↪self.average_endowment

        # Payoff calculation based on strategy
        if strategy:  # Cooperator
            cooperation_cost = endowment * self.contribution_factor
            if contributions >= required_contribution:  # Group succeeds
                return endowment - cooperation_cost
            else:  # Group fails
                return endowment * (1 - self.risk) - cooperation_cost
        elif not strategy:  # Defector
            if contributions >= required_contribution:  # Group succeeds
                return endowment
            else:  # Group fails
                return endowment * (1 - self.risk)

    def build_transition_matrix_full(self):
        """
        UNUSED since we don't build the whole transition matrix, just along the␣
↪diagonal

        Build the full transition matrix

        Transition matrix is essentially a representation of the population
        going "from" certain states "to" other states (within the system)

        So we have (Z_R + 1) * (Z_P + 1) on both x and y axes, and we map
        how each state (C_R_i, C_D_i) "transitions" to another state (C_R_j,␣
↪C_D_j)

        Once we have this massive transition matrix, we can solve for the␣
↪stationary distribution
```

```
        Returns:
            Square transition matrix of size (Z_R + 1) * (Z_P + 1) x (Z_R + 1)␣
↪* (Z_P + 1)
        """
        self.generate_states() # Just in case!
        num_states = len(self.states) # Should be (Z_R + 1) * (Z_P + 1)
        transition_matrix = np.zeros((num_states, num_states))

        # Iterate over all state pairs using the mapping we created
        for (C_R_i, C_P_i), (i, scalar_i) in self.indexed_states.items():  #␣
↪Current state
            # Calculate the fitness of every population in this configuration
            f_R_C = self.calculate_fitness_rich_C(C_R_i, C_P_i)
            f_R_D = self.calculate_fitness_rich_D(C_R_i, C_P_i)
            f_P_C = self.calculate_fitness_poor_C(C_R_i, C_P_i)
            f_P_D = self.calculate_fitness_poor_D(C_R_i, C_P_i)

            for (C_R_j, C_P_j), (j, scalar_j) in self.indexed_states.items(): ␣
↪# Target state
                # print(f"Computing T[{i, j}]")
                # Skip diagonal for now; handle transitions
                if i == j:
                    continue

                # Delta in strategies for both populations
                delta_C_R = C_R_j - C_R_i
                delta_C_P = C_P_j - C_P_i

                # Skip impossible transitions, could happen because we're␣
↪iterating over everything
                # We allow delta to be negative, it just means a net decrease
                if abs(delta_C_R) > self.Z_R or abs(delta_C_P) > self.Z_P:
                    continue

                # Compute the transition probability
                prob = self.compute_joint_transition_probability(
                    C_R_i, C_P_i, C_R_j, C_P_j, delta_C_R, delta_C_P, f_R_C,␣
↪f_R_D, f_P_C, f_P_D
                )

                # Populate the matrix
                transition_matrix[i][j] = prob

        # Set diagonal elements to ensure rows sum to 1, that's the probability␣
↪that the state won't change
```

```
        for i in range(num_states):
            transition_matrix[i][i] = 1 - np.sum(transition_matrix[i])

        return transition_matrix

    def get_transition_matrix(self):
        """
        Helper function to not recompute transition matrix if it was already
        calculated for this game config
        """
        if self.transition_matrix is not None:
            return self.transition_matrix
        self.transition_matrix = self.build_transition_matrix()
        return self.transition_matrix

    def build_transition_matrix(self):
        """
        Build the partial transition matrix along the diagonal with neighboring␣
↪states

        Transition matrix is essentially a representation of the population
        going "from" certain states "to" other states (within the system)

        So we have (Z_R + 1) * (Z_P + 1) on both x and y axes, and we map
        how each state (C_R_i, C_D_i) "transitions" to another state (C_R_j,␣
↪C_D_j)

        This is because (C_R, C_P) can actually fully represent the state of␣
↪the whole system
        We can derive (D_R, D_P) from these, so we use the scalar␣
↪transformations of these tuples
        to plot them on the x and y axis (or matrix indices, in our case)

        To save some time (and our computers), we do this for neighboring␣
↪states,
        so the values are filled along the diagonal! Check if this is fine!

        Once we have this massive transition matrix, W, we can solve for the␣
↪stationary distribution, pi:
        W.pi = pi, where W is the transposition of W, or W.T

        Returns:
            Square transition matrix of size (Z_R + 1) * (Z_P + 1) ^ 2
        """
        self.generate_states() # ensure states and indexing are up-to-date
        num_states = len(self.states)  # (Z_R + 1) * (Z_P + 1)
```

```python
        transition_matrix = np.zeros((num_states, num_states))

        # Now build the transition matrix row by row
        for (C_R_i, C_P_i), (i, scalar_i) in self.indexed_states.items():
            f_R_C = self.calculate_fitness_rich_C(C_R_i, C_P_i)
            f_R_D = self.calculate_fitness_rich_D(C_R_i, C_P_i)
            f_P_C = self.calculate_fitness_poor_C(C_R_i, C_P_i)
            f_P_D = self.calculate_fitness_poor_D(C_R_i, C_P_i)

            if self.verbose:
                print(f"For (C_R, C_P)={C_R_i}, {C_P_i}, \t (f_R_C, f_R_D,␣
↪f_P_C, f_P_D) = ({f_R_C}, {f_R_D}, {f_P_C}, {f_P_D})")

            # Possible neighbors differ by exactly one individual's strategy
            neighbors = [
                (C_R_i + 1, C_P_i),
                (C_R_i - 1, C_P_i),
                (C_R_i, C_P_i + 1),
                (C_R_i, C_P_i - 1)
            ]

            # Filter out invalid neighbors (out of bounds)
            neighbors = [(C_R_j, C_P_j) for (C_R_j, C_P_j) in neighbors
                         if 0 <= C_R_j <= self.Z_R and 0 <= C_P_j <= self.Z_P]

            # neighbors = [(C_R_j, C_P_j) for (C_R_j, C_P_j) in neighbors
            #              if self.O_R <= C_R_j <= self.Z_R and self.O_P <=␣
↪C_P_j <= self.Z_P]

            total_out_prob = 0.0
            for (C_R_j, C_P_j) in neighbors:
                j, scalar_j = self.indexed_states[(C_R_j, C_P_j)]

                # Determine the deltas
                delta_C_R = C_R_j - C_R_i
                delta_C_P = C_P_j - C_P_i

                # Check for obstinate behaviour in our state
                # if C_R_j < self.O_R or C_P_j < self.O_P:
                #     # Leave these values as 0
                #     continue

                # Compute the probability of transitioning from state i to j
                prob = self.compute_joint_transition_probability(
                    C_R_i, C_P_i, C_R_j, C_P_j, delta_C_R, delta_C_P, f_R_C,␣
↪f_R_D, f_P_C, f_P_D
                )
```

```
                if self.verbose:
                    print(f"Transition from {C_R_i, C_P_i} to {C_R_j, C_P_j}:␣
↪{prob}")
                transition_matrix[i, j] = prob
                total_out_prob += prob

            # The diagonal entry ensures that rows sum up to 1
            transition_matrix[i, i] = 1.0 - total_out_prob

        return transition_matrix

    def compute_joint_transition_probability(self,
                                             C_R_i,
                                             C_P_i,
                                             C_R_j,
                                             C_P_j,
                                             delta_C_R,
                                             delta_C_P,
                                             f_R_C,
                                             f_R_D,
                                             f_P_C,
                                             f_P_D):
        """
        Compute the joint probability of transitioning between states with
        given changes in strategies

        Parameters:
        - C_R_i, C_P_i: Current state
        - C_R_j, C_P_j: Target state
        - delta: Changes in the number of rich and poor cooperators, can be␣
↪positive or negative
        - Fitness for each strategy and population

        Returns:
            Probability of transitioning from (C_R_i, C_P_i) to (C_R_j, C_P_j)
        """
        prob = 1.0

        if delta_C_R > 0:  # Rich D -> C
            for _ in range(delta_C_R):
                prob *= self.transition_probability(
                    i_k_X=self.Z_R - C_R_i,  # Rich defectors
                    i_k_Y=C_R_i,             # Rich cooperators
                    i_l_X=self.Z_P - C_P_i,  # Poor defectors
                    i_l_Y=C_P_i,             # Poor cooperators
                    Z_k=self.Z_R,
                    f_k_X=f_R_D, f_k_Y=f_R_C,
```

```
                f_l_X=f_P_D, f_l_Y=f_P_C
            )
            C_R_i += 1
    elif delta_C_R < 0:  # Rich C -> D
        for _ in range(-delta_C_R):
            prob *= self.transition_probability(
                i_k_X=C_R_i,            # Rich cooperators
                i_k_Y=self.Z_R - C_R_i, # Rich defectors
                i_l_X=C_P_i,            # Poor cooperators
                i_l_Y=self.Z_P - C_P_i, # Poor defectors
                Z_k=self.Z_R,
                f_k_X=f_R_C, f_k_Y=f_R_D,
                f_l_X=f_P_C, f_l_Y=f_P_D
            )
            C_R_i -= 1

    if delta_C_P > 0:  # Poor D -> C
        for _ in range(delta_C_P):
            prob *= self.transition_probability(
                i_k_X=self.Z_P - C_P_i,  # Poor defectors
                i_k_Y=C_P_i,             # Poor cooperators
                i_l_X=self.Z_R - C_R_i,  # Rich defectors
                i_l_Y=C_R_i,             # Rich cooperators
                Z_k=self.Z_P,
                f_k_X=f_P_D, f_k_Y=f_P_C,
                f_l_X=f_R_D, f_l_Y=f_R_C
            )
            C_P_i += 1
    elif delta_C_P < 0:  # Poor C -> D
        for _ in range(-delta_C_P):
            prob *= self.transition_probability(
                i_k_X=C_P_i,            # Poor cooperators
                i_k_Y=self.Z_P - C_P_i, # Poor defectors
                i_l_X=C_R_i,            # Rich defectors
                i_l_Y=self.Z_R - C_R_i, # Rich cooperators
                Z_k=self.Z_P,
                f_k_X=f_P_C, f_k_Y=f_P_D,
                f_l_X=f_R_C, f_l_Y=f_R_D
            )
            C_P_i -= 1

    return prob

def get_stationary_distribution(self):
    """
    Helper function to not recompute stationary distribution if it was␣
↪already
```

```
        calculated for this game config
        """
        if self.stationary_distribution is not None:
            return self.stationary_distribution
        self.stationary_distribution = self.compute_stationary_distribution()

        # Clip the values if there is a max param for stationary distribution
        if self.pi_max != float('inf'):
            self.stationary_distribution = np.clip(self.
↪stationary_distribution, None, self.pi_max)
        return self.stationary_distribution

    def compute_stationary_distribution(self):
        """
        Compute the stationary distribution using the transition matrix

        Thisis essentially an eigenvector search problem, i.e. we need to find␣
↪the eigenvalue
        associated with the eigenvector 1. Stationary distribution does not␣
↪affect the direction
        of the transition matrix entries, it just affects the amplitude or␣
↪magnitude.

        To do this we use the transposed transition matrix (W.T), we need to␣
↪find pi in
        the equation W.T * pi = pi, i.e. W should not affect the stationary␣
↪distribution if
        applied to it again and again. This means that the stationary␣
↪distribution (pi)
        should then describe the "long-term" behaviour of the Markov chain,
        or the state where the system's probabilities converge/stabilise!

        It's the "steady-state behavior of the Markov chain"

        References:
        https://stackoverflow.com/questions/31791728/
↪python-code-explanation-for-stationary-distribution-of-a-markov-chain
        https://en.wikipedia.org/wiki/Stationary_distribution
        https://www.youtube.com/watch?v=PFDu9oVAE-g
        https://www.geeksforgeeks.org/transition-probability-matrix/
        """
        transition_matrix = self.get_transition_matrix()

        # Use egttools.utils.calculate_stationary_distribution
        # stationary_distribution =␣
↪calculate_stationary_distribution(transition_matrix)
```

```
        eigenvalues, eigenvectors = np.linalg.eig(transition_matrix.T)
        eigenvector1 = eigenvectors[:,np.isclose(eigenvalues, 1.0)]

        # Since np.isclose will return an array, we've indexed with an array
        # so we still have our 2nd axis, we should get rid of it, since it's␣
↪only size 1
        eigenvector1 = eigenvector1[:,0]

        stationary_distribution = eigenvector1 / eigenvector1.sum()

        # This contains complex eigenvalues and eigenvectors, but we want the␣
↪real part
        stationary_distribution = stationary_distribution.real

        return stationary_distribution

    def calculate_average_groups_reaching_threshold(self, C_R, C_P):
        """
        Calculate the average fraction of groups that successfully achieve the␣
↪public good.
        Uses multivariate hypergeometric sampling to calculate success␣
↪probabilities.
        """
        def group_contributions(rich_cooperators, poor_cooperators):
            return (
                rich_cooperators * self.contribution_factor * self.
↪endowment_rich +
                poor_cooperators * self.contribution_factor * self.
↪endowment_poor
            )

        # Number of defectors
        D_R, D_P = self.Z_R - C_R, self.Z_P - C_P
        weighted_total = 0.0
        successes = 0.0

        # Total number of ways to form a group of size N
        total_states = comb(self.Z + 1, self.N + 1)

        for j_Rc in range(max(0, self.N - (D_R + C_P + D_P)), min(C_R, self.N)␣
↪+ 1):
            for j_Pc in range(max(0, self.N - (j_Rc + D_R + D_P)), min(C_P,␣
↪self.N - j_Rc) + 1):
                j_Rd = self.N - (j_Rc + j_Pc)
                if j_Rd > D_R:
```

```
                    continue
                j_Pd = self.N - (j_Rc + j_Rd + j_Pc)
                if j_Pd > D_P:
                    continue

                assert j_Rc + j_Rd + j_Pc + j_Pd == self.N, "Invalid group,␣
↪should never happen if we get to this point"

                # Probability weight of this configuration
                prob = (
                    comb(C_R, j_Rc) * comb(D_R, j_Rd) *
                    comb(C_P, j_Pc) * comb(D_P, j_Pd)
                ) / total_states

                weighted_total += prob

                # Check success condition
                if group_contributions(j_Rc, j_Pc) >= self.threshold * self.
↪average_endowment * self.contribution_factor:
                    successes += prob
                    if self.verbose:
                        print(f"Group with {j_Rc} rich and {j_Pc} poor Cs␣
↪reached {group_contributions(j_Rc, j_Pc)}")
                else:
                    if self.verbose:
                        print(f"Group with {j_Rc} rich and {j_Pc} poor Cs DID␣
↪NOT reach {group_contributions(j_Rc, j_Pc)}")

        # print(successes)
        # Calculate and return the weighted fraction
        return successes / weighted_total

    def compute_average_group_achievement_matrix(self):
        """
        Precompute the average group achievement (a_G) for all configurations␣
↪(C_R, C_P).
        """
        a_G_matrix = np.zeros((self.Z_R + 1, self.Z_P + 1))
        for C_R in range(self.Z_R + 1):
            for C_P in range(self.Z_P + 1):
                a_G_matrix[C_R, C_P] = self.
↪calculate_average_groups_reaching_threshold(C_R, C_P)
        return a_G_matrix

    def compute_eta_G(self):
        """
```

```
        Compute the average group achievement (eta_G) weighted by the␣
↪stationary distribution.
        """
        # Get the stationary distribution and reshape it into a 2D matrix
        stationary_distribution = self.get_stationary_distribution()
        pi_matrix = stationary_distribution.reshape((self.Z_R + 1, self.Z_P +␣
↪1))

        # Compute the average group achievement matrix
        a_G_matrix = self.compute_average_group_achievement_matrix()

        # Compute  _G as the weighted sum of a_G by the stationary distribution
        weighted_group_achievement = pi_matrix * a_G_matrix
        eta_G = np.sum(weighted_group_achievement)

        return eta_G

    def compute_gradient_of_selection(self):
        """
        Compute the gradient of selection (GoS) for each configuration.

        Returns:
        X, Y: Meshgrid of states (i_R, i_P)
        U, V: Gradients of selection for i_R and i_P
        stationary_distribution: Matrix of stationary probabilities
        """
        # Get stationary distribution
        stationary_distribution = self.get_stationary_distribution()
        stationary_distribution = stationary_distribution.reshape((self.Z_R +␣
↪1, self.Z_P + 1))

        # Get transition matrix
        transition_matrix = self.get_transition_matrix()

        # Initialize gradient arrays
        U = np.zeros((self.Z_R + 1, self.Z_P + 1))  # Gradient in i_R direction
        V = np.zeros((self.Z_R + 1, self.Z_P + 1))  # Gradient in i_P direction

        for (C_R_i, C_P_i), (i, scalar_i) in self.indexed_states.items():
            # Get T_{i,R}+ and T_{i,R}-
            T_R_plus = transition_matrix[i, self.indexed_states.get((C_R_i + 1,␣
↪C_P_i), (None, None))[0]] if C_R_i + 1 <= self.Z_R else 0
            T_R_minus = transition_matrix[i, self.indexed_states.get((C_R_i -␣
↪1, C_P_i), (None, None))[0]] if C_R_i - 1 >= 0 else 0

            # Get T_{i,P}+ and T_{i,P}-
```

```
            T_P_plus = transition_matrix[i, self.indexed_states.get((C_R_i,␣
↪C_P_i + 1), (None, None))[0]] if C_P_i + 1 <= self.Z_P else 0
            T_P_minus = transition_matrix[i, self.indexed_states.get((C_R_i,␣
↪C_P_i - 1), (None, None))[0]] if C_P_i - 1 >= 0 else 0

            # Compute the gradients
            U[C_R_i, C_P_i] = T_R_plus - T_R_minus

            V[C_R_i, C_P_i] = T_P_plus - T_P_minus

        # Create a meshgrid for plotting
        X, Y = np.meshgrid(range(self.Z_R + 1), range(self.Z_P + 1),␣
↪indexing='ij')

        if self.gos_max != float('inf'):
            U = np.clip(U, None, self.gos_max)
            V = np.clip(V, None, self.gos_max)

        return X, Y, U, V, stationary_distribution
```

[3]:

```
def plot_gradient_of_selection(X, Y, U_, V_, stationary_distribution,␣
↪title=None, eta_G=0, arrow_density=3):
    """
    Standalone function to plot the gradient of selection using matplotlib.
↪quiver and stationary distribution.

    Usage:
        X, Y, U, V, stationary_distribution = game.
↪compute_gradient_of_selection()
        eta_G = game.compute_eta_G()
        plot_gradient_of_selection(X, Y, U, V, stationary_distribution, eta_G)
    """
    # Amplify gradients because they're too small otherwise
    scaling_factor = 100 # Large, but we can play around with this
    U = U_ * scaling_factor
    V = V_ * scaling_factor

    # Need to scale the gradients for color mapping otherwise it complains␣
↪about different sizes
    magnitude = np.sqrt(U**2 + V**2)

    # Downsample the whole matrix, so we can actually see what's happening␣
↪instead of absolute chaos
    X = X[::arrow_density, ::arrow_density]
    Y = Y[::arrow_density, ::arrow_density]
    U = U[::arrow_density, ::arrow_density]
```

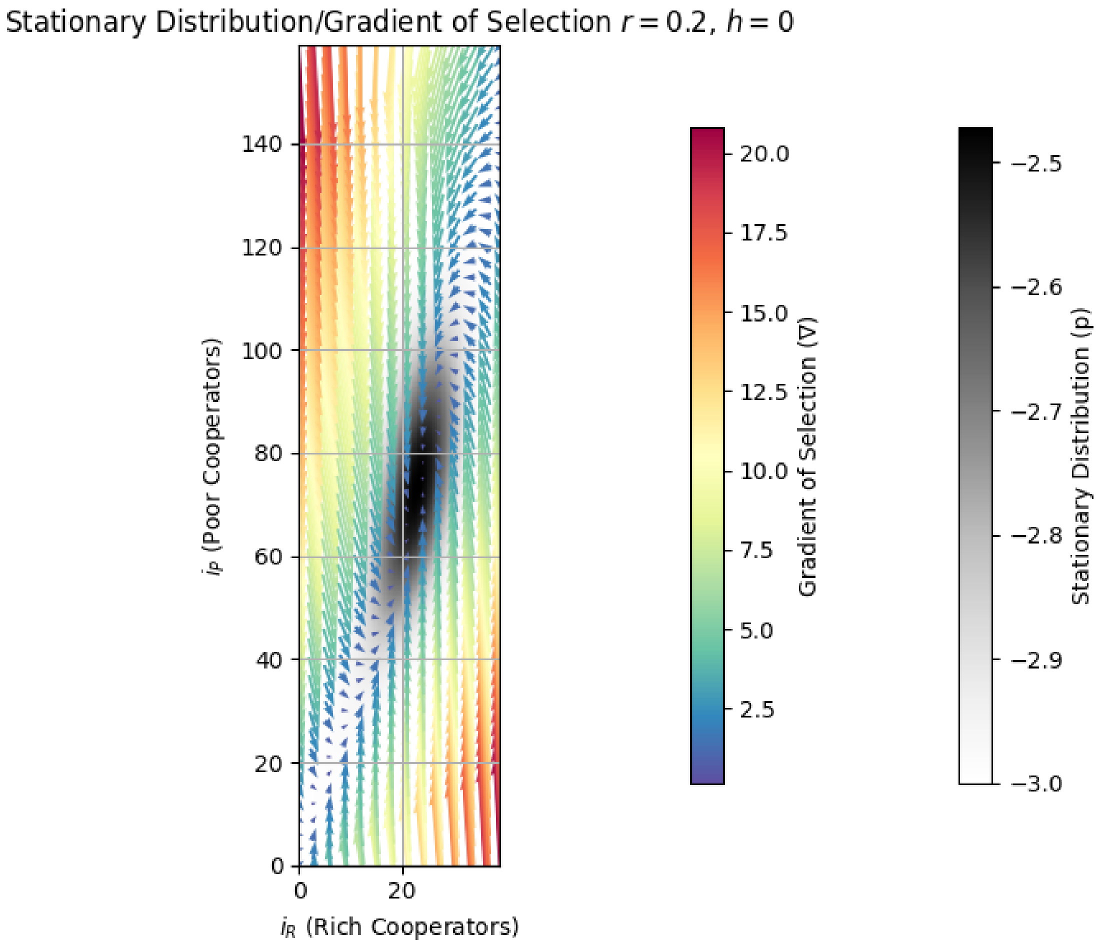
```python
    V = V[::arrow_density, ::arrow_density]
    magnitude = magnitude[::arrow_density, ::arrow_density]

    # Flatten X, Y, U, V to match the number of arrow positions
    X = X.flatten()
    Y = Y.flatten()
    U = U.flatten()
    V = V.flatten()

    fig, ax = plt.subplots(figsize=(8, 6))

    # Plot stationary distribution as a heatmap
    im = ax.imshow(
        np.log10(stationary_distribution.T + 1e-3),  # Log scale for better contrast! Otherwise also can't fully see what's happening
        origin='lower',
        cmap='Greys',
        extent=[0, X.max(), 0, Y.max()]
    )

    # Add quiver plot with colormap based on magnitude of gradients!
    quiver = ax.quiver(
        X, Y, U, V, magnitude,  # magnitude is used for arrow colors
        scale=0.4,
        scale_units='xy',
        pivot='middle',
        width=0.015,
        cmap='Spectral_r' # Could also use viridis, inferno, cividis, plasma, Spectral
        # Spectral_r is reversed Spectral, which is the closest to the colours used in the paper
    )

    # Show the bars on the side about stationary distribution and gradient of selection
    cbar1 = fig.colorbar(im, ax=ax, location='right', pad=0.1, shrink=0.8)
    cbar1.set_label("Stationary Distribution (p)", fontsize=10)

    cbar2 = fig.colorbar(quiver, ax=ax, location='right', pad=0.2, shrink=0.8)
    cbar2.set_label("Gradient of Selection ( )", fontsize=10)

    if eta_G != 0:
        ax.set_title(f"Stationary Distribution/Gradient of Selection ( G = {eta_G:.3f})", fontsize=12)
    else:
        ax.set_title(f"Stationary Distribution/Gradient of Selection", fontsize=12)
```

```python
    if title is not None:
        ax.set_title(f"Stationary Distribution/Gradient of Selection {title}")

    ax.set_xlabel("$i_R$ (Rich Cooperators)", fontsize=10)
    ax.set_ylabel("$i_P$ (Poor Cooperators)", fontsize=10)
    ax.grid()

    plt.tight_layout()
    plt.show()
```

[4]:
```python
# Default parameters
population_size = 200
rich_fraction = 0.2
endowment_rich = 2.5
endowment_poor = 0.625
group_size = 6
threshold = 3
contribution_factor = 0.1
risk = 0
homophily = 0
beta = 5
mu = 0.01
```

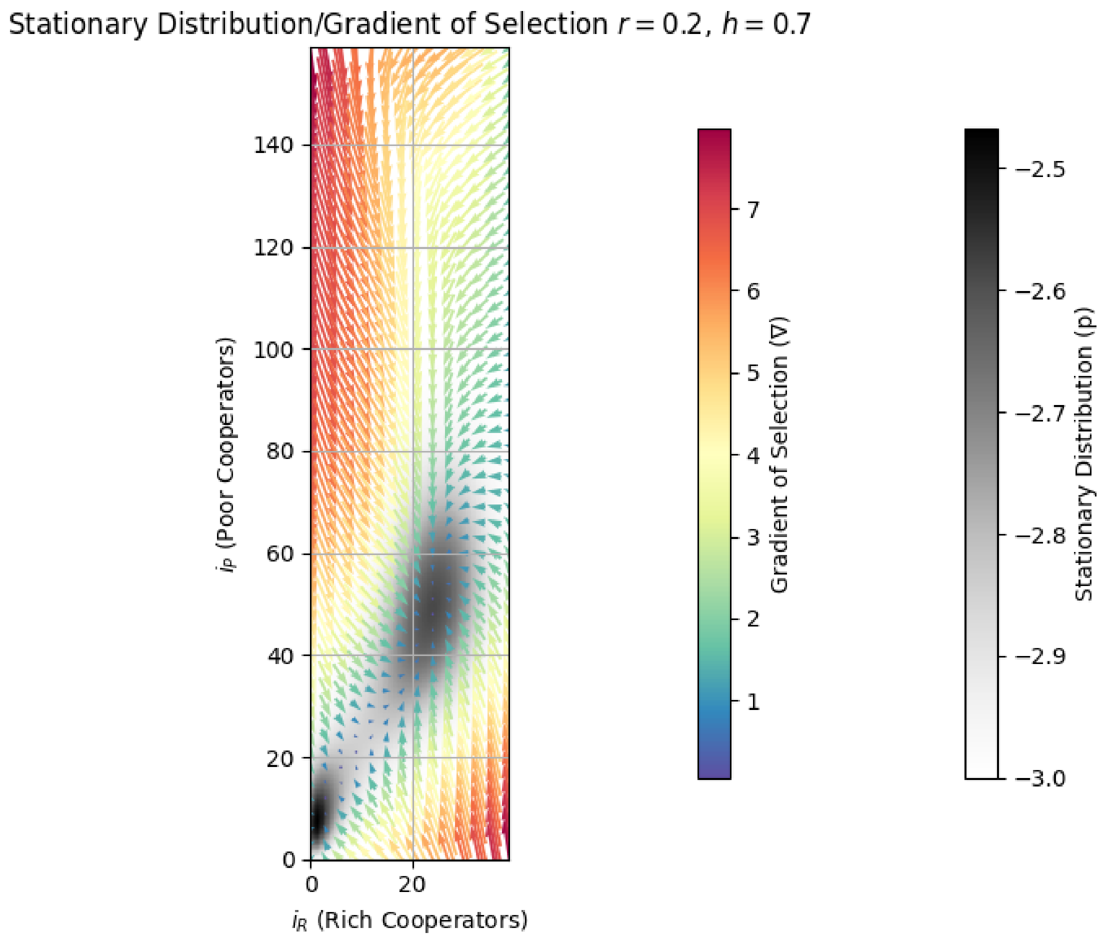
1 Stationary distribution and gradient of selection for different values of r and h, within the population of Z_R x Z_P

[7]:
```python
# Figure 2 - A
game = ClimateThresholdPublicGoodsGame(
    population_size=population_size,
    rich_fraction=rich_fraction,
    endowment_rich=endowment_rich,
    endowment_poor=endowment_poor,
    group_size=group_size,
    threshold=threshold,
    risk=0.2,
    contribution_factor=contribution_factor,
    homophily=0,
    beta=beta,
    mu=mu
)

X, Y, U, V, stationary_distribution = game.compute_gradient_of_selection()
plot_gradient_of_selection(X, Y, U, V, stationary_distribution, "$r=0.2$, 
 ↪$h=0$")
```



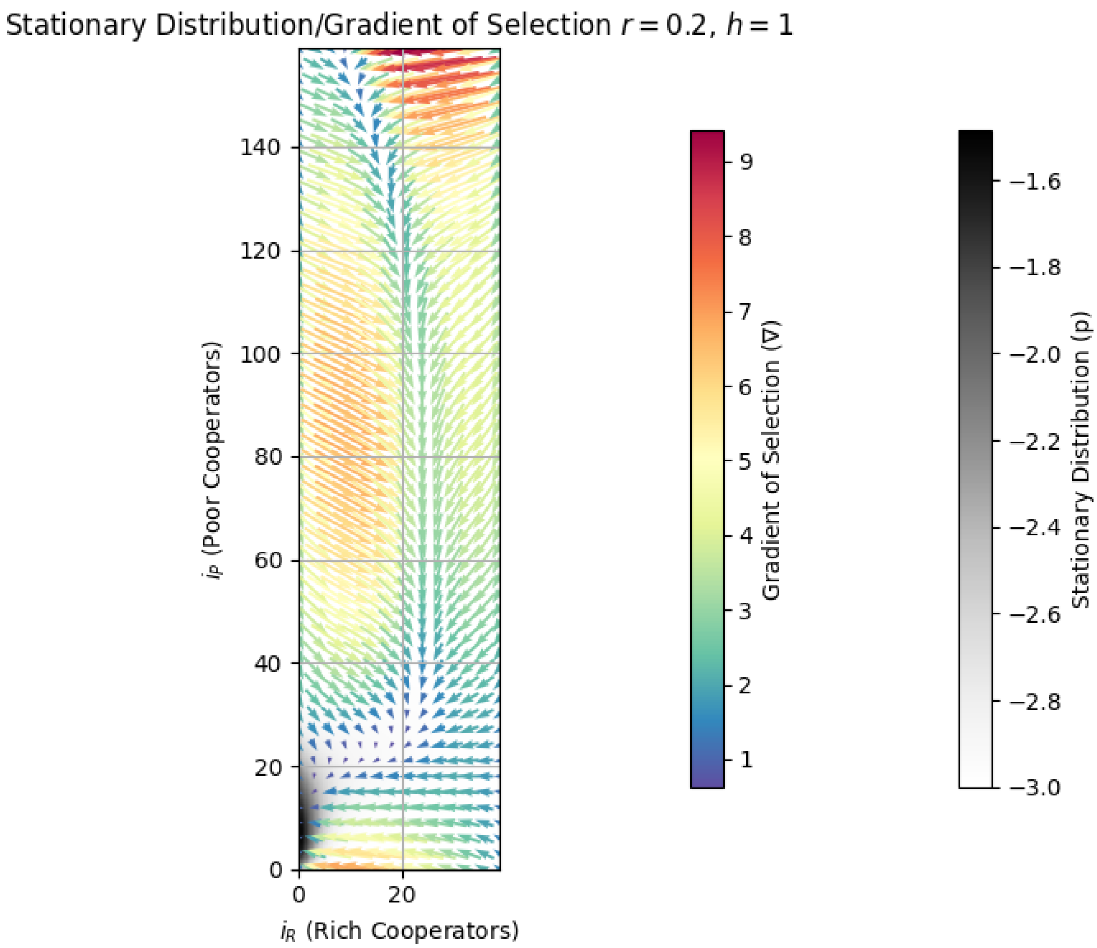
[8]:
```
# Figure 2 - B
game = ClimateThresholdPublicGoodsGame(
    population_size=population_size,
    rich_fraction=rich_fraction,
    endowment_rich=endowment_rich,
    endowment_poor=endowment_poor,
    group_size=group_size,
    threshold=threshold,
    risk=0.2,
    contribution_factor=contribution_factor,
    homophily=0.7,
    beta=beta,
    mu=mu
)

X, Y, U, V, stationary_distribution = game.compute_gradient_of_selection()
```

```
plot_gradient_of_selection(X, Y, U, V, stationary_distribution, "$r=0.2$, $h=0.
↪7$")
```



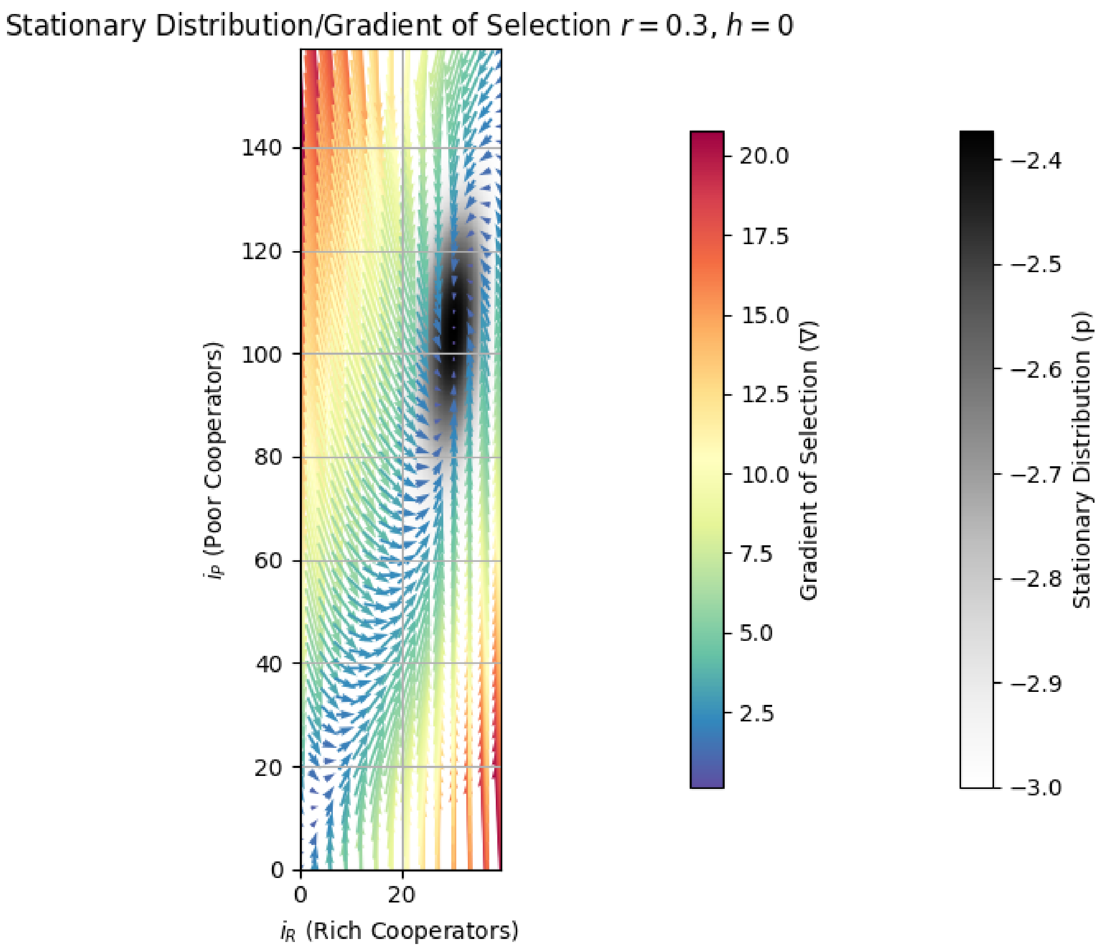

```
[9]: # Figure 2 - C
game = ClimateThresholdPublicGoodsGame(
    population_size=population_size,
    rich_fraction=rich_fraction,
    endowment_rich=endowment_rich,
    endowment_poor=endowment_poor,
    group_size=group_size,
    threshold=threshold,
    risk=0.2,
    contribution_factor=contribution_factor,
    homophily=1,
    beta=beta,
    mu=mu
)
```

```
X, Y, U, V, stationary_distribution = game.compute_gradient_of_selection()
plot_gradient_of_selection(X, Y, U, V, stationary_distribution, "$r=0.2$,␣
↪$h=1$")
```



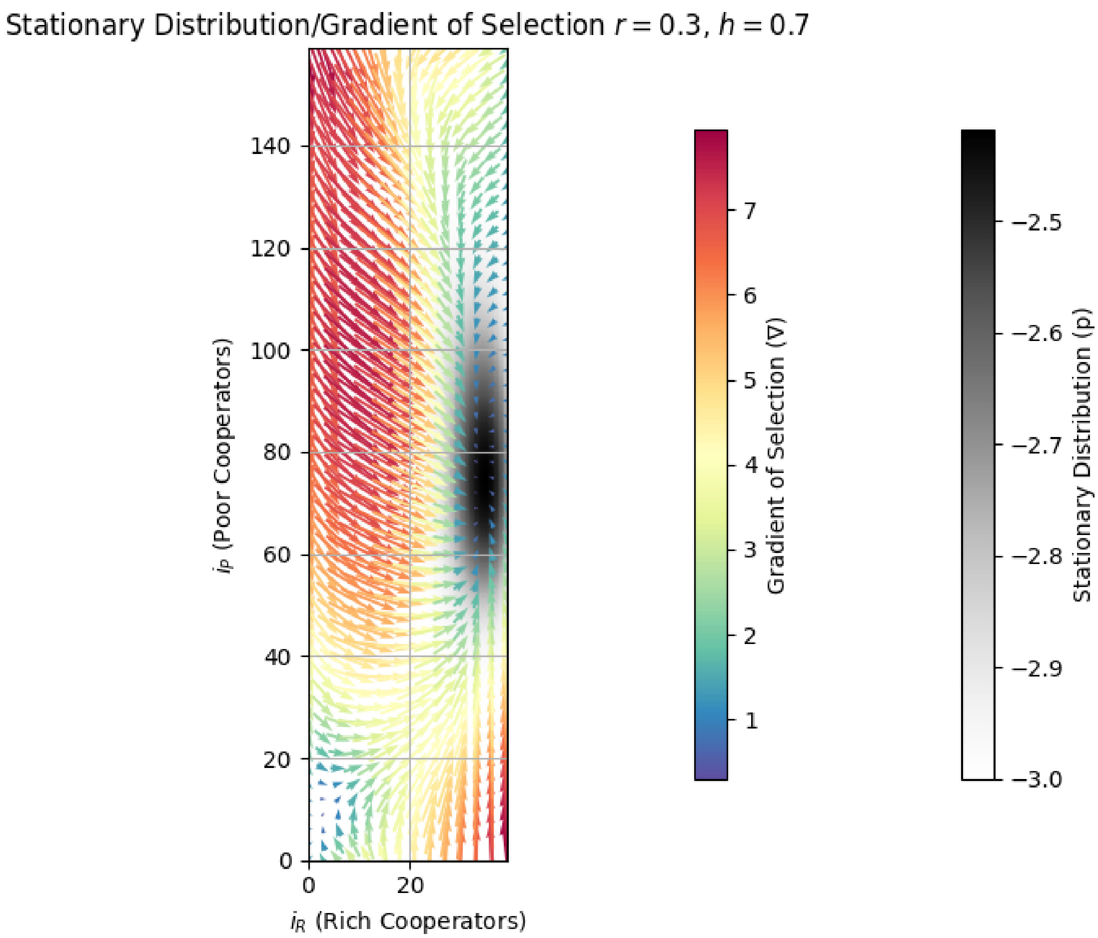
[10]:
```
# Figure 2 - D
game = ClimateThresholdPublicGoodsGame(
    population_size=population_size,
    rich_fraction=rich_fraction,
    endowment_rich=endowment_rich,
    endowment_poor=endowment_poor,
    group_size=group_size,
    threshold=threshold,
    risk=0.3,
    contribution_factor=contribution_factor,
    homophily=0,
    beta=beta,
```

```
    mu=mu
)

X, Y, U, V, stationary_distribution = game.compute_gradient_of_selection()
plot_gradient_of_selection(X, Y, U, V, stationary_distribution, "$r=0.3$,␣
↪$h=0$")
```



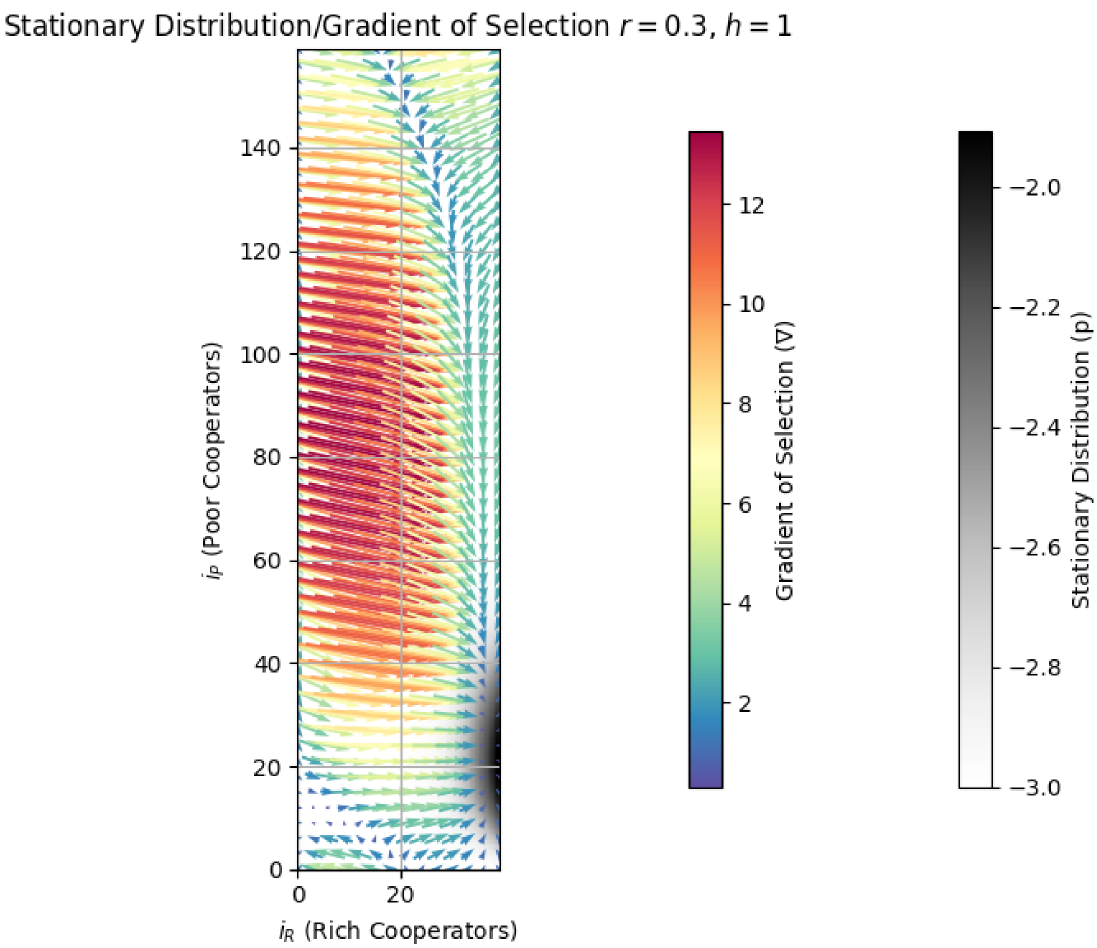
[11]:
```
# Figure 2 - E
game = ClimateThresholdPublicGoodsGame(
    population_size=population_size,
    rich_fraction=rich_fraction,
    endowment_rich=endowment_rich,
    endowment_poor=endowment_poor,
    group_size=group_size,
    threshold=threshold,
    risk=0.3,
    contribution_factor=contribution_factor,
```

```
    homophily=0.7,
    beta=beta,
    mu=mu
)

X, Y, U, V, stationary_distribution = game.compute_gradient_of_selection()
plot_gradient_of_selection(X, Y, U, V, stationary_distribution, "$r=0.3$, $h=0.
↪7$")
```



[13]:
```
# Figure 2 - F
game = ClimateThresholdPublicGoodsGame(
    population_size=population_size,
    rich_fraction=rich_fraction,
    endowment_rich=endowment_rich,
    endowment_poor=endowment_poor,
    group_size=group_size,
    threshold=threshold,
```

```
    risk=0.3,
    contribution_factor=contribution_factor,
    homophily=1,
    beta=beta,
    mu=mu
)

X, Y, U, V, stationary_distribution = game.compute_gradient_of_selection()
plot_gradient_of_selection(X, Y, U, V, stationary_distribution, "$r=0.3$,␣
↪$h=1$")
```

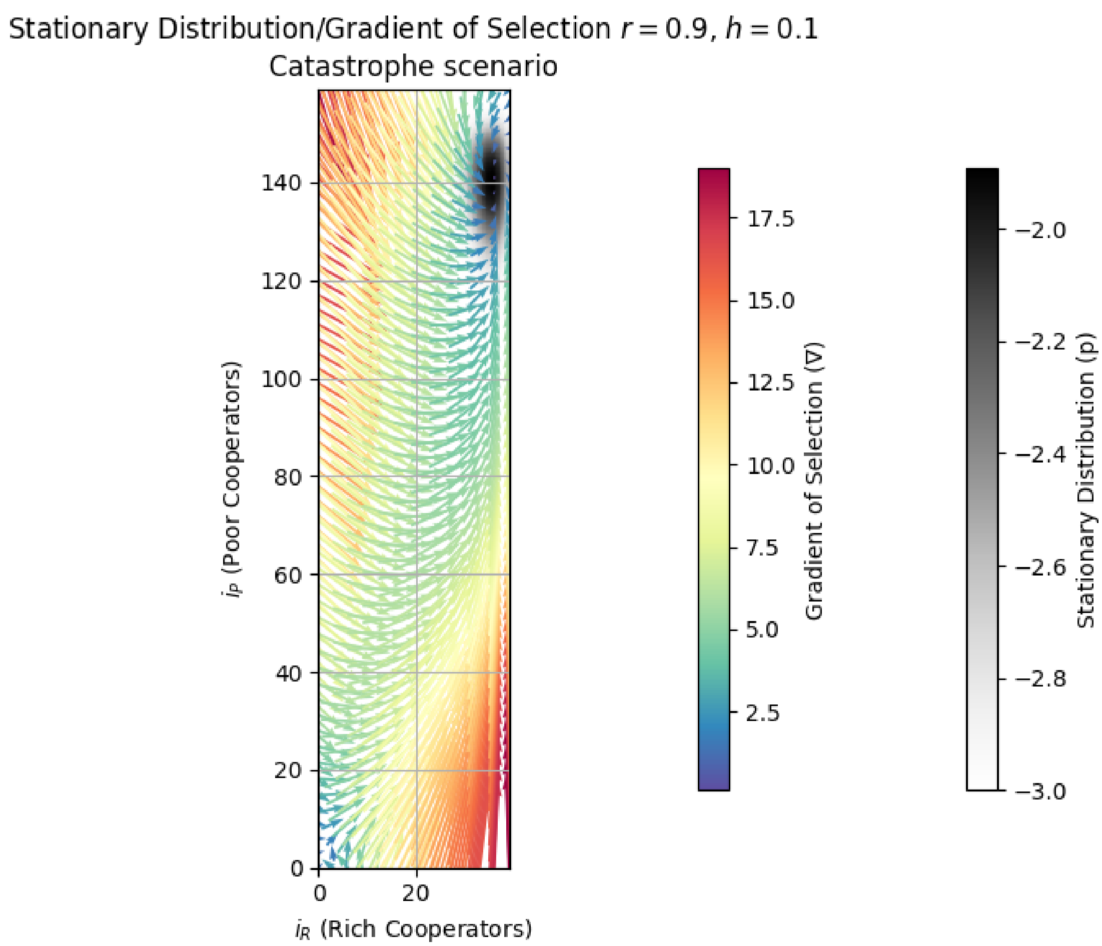


1.1 Additional figure with high risk, $r = 0.9$, low homophily, $h = 0.1$

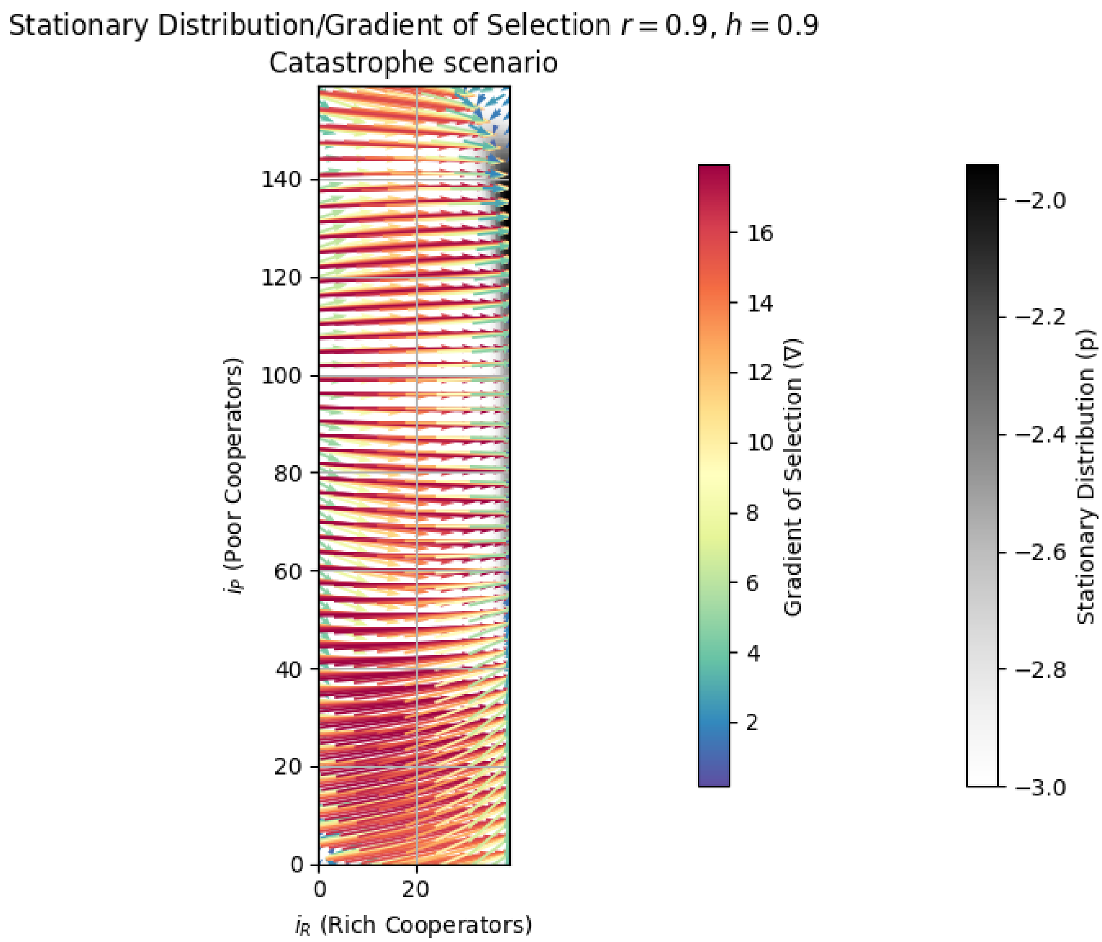
Ideal imitation strategy under climate pressure, for when the end is nigh!

[6]:
```
# Additional figure with high risk and low homophily
game = ClimateThresholdPublicGoodsGame(
    population_size=population_size,
```

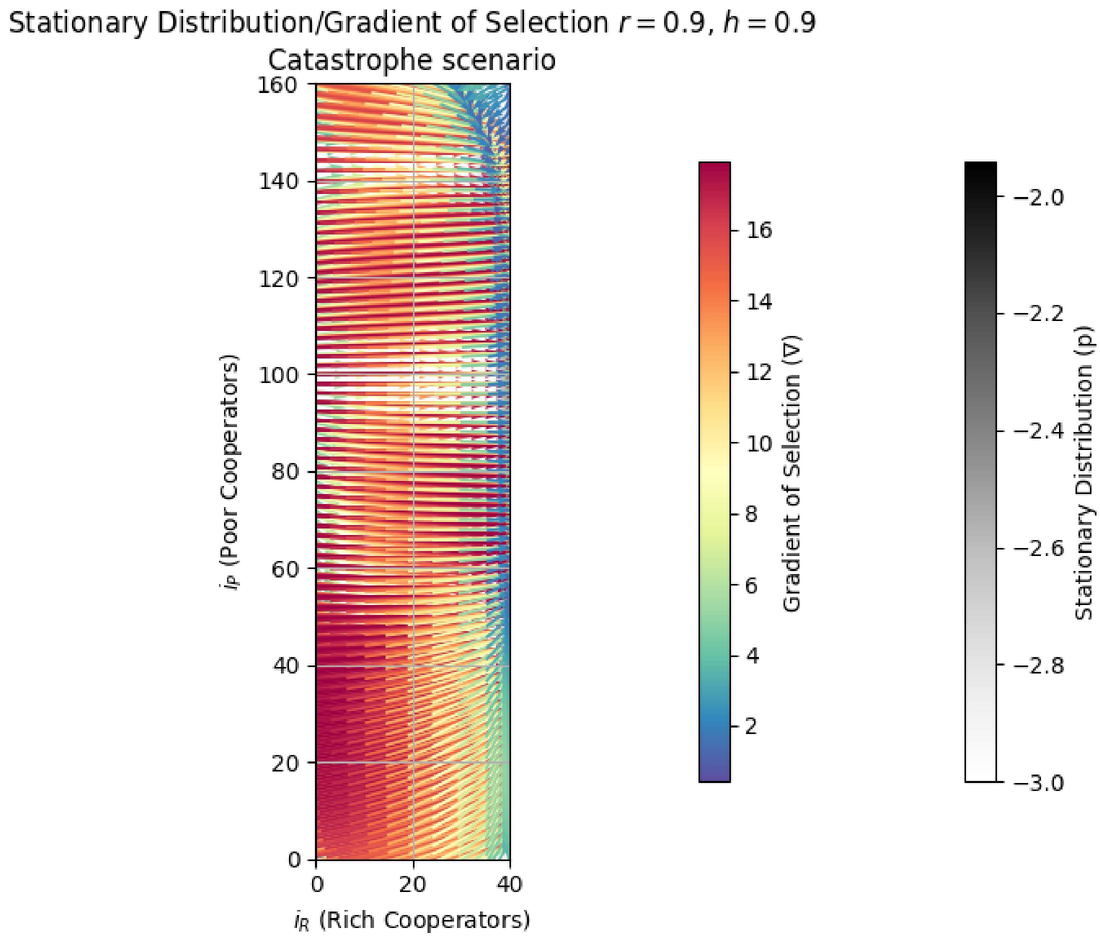
```
    rich_fraction=rich_fraction,
    endowment_rich=endowment_rich,
    endowment_poor=endowment_poor,
    group_size=group_size,
    threshold=threshold,
    risk=0.9,
    contribution_factor=contribution_factor,
    homophily=0.1,
    beta=beta,
    mu=mu
)

X, Y, U, V, stationary_distribution = game.compute_gradient_of_selection()
plot_gradient_of_selection(X, Y, U, V, stationary_distribution, "$r=0.9$, $h=0.
↪1$\nCatastrophe scenario")
```



[51]:
```python
# Additional figure with high risk and high homophily
game = ClimateThresholdPublicGoodsGame(
    population_size=population_size,
    rich_fraction=rich_fraction,
    endowment_rich=endowment_rich,
    endowment_poor=endowment_poor,
    group_size=group_size,
    threshold=threshold,
    risk=0.9,
    contribution_factor=contribution_factor,
    homophily=0.9,
    beta=beta,
    mu=mu
)

X, Y, U, V, stationary_distribution = game.compute_gradient_of_selection()
plot_gradient_of_selection(X, Y, U, V, stationary_distribution, "$r=0.9$, $h=0.
↪9$\nCatastrophe scenario")
```



```
[58]: plot_gradient_of_selection(X, Y, U, V, stationary_distribution, "$r=0.9$, $h=0.
      ↪9$\nCatastrophe scenario", arrow_density=2)
```

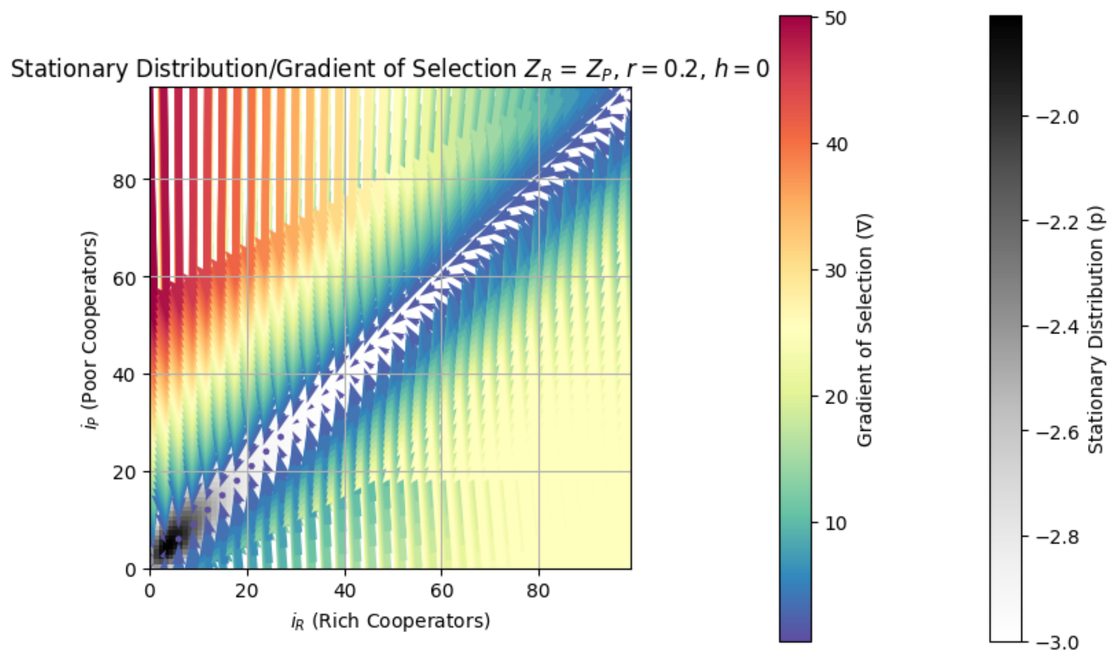


2 Figures from the SI Text with $Z_R = Z_P$

```
[15]: # Redefine default params for this experiment
rich_fraction = 0.5
endowment_rich = 1.7
endowment_poor = 0.3
```

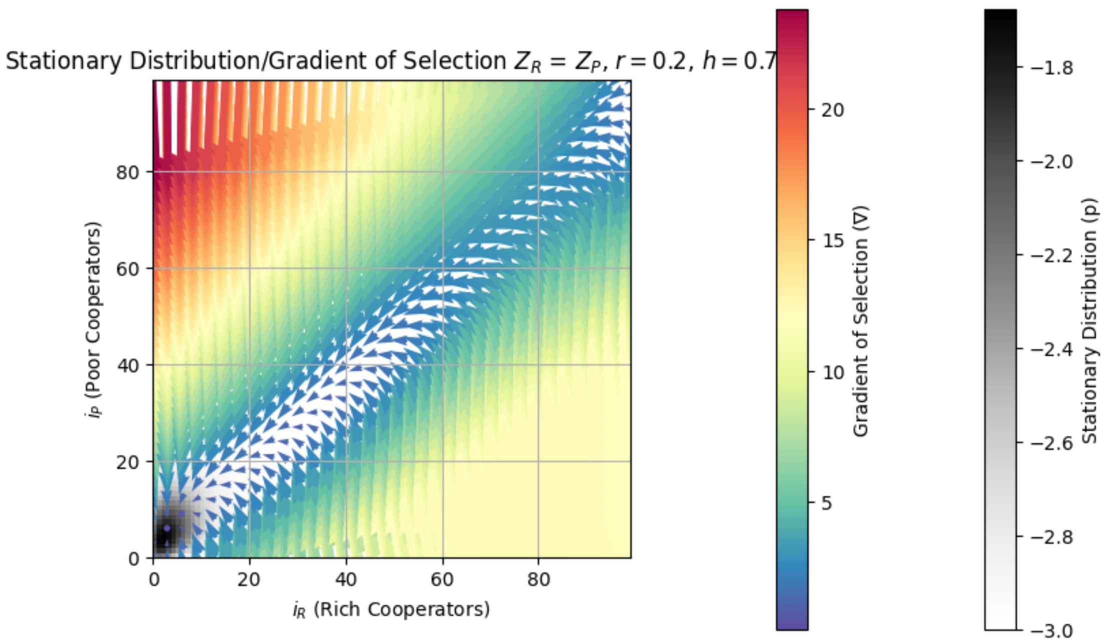
```
[18]: # Figure S2 - A
game = ClimateThresholdPublicGoodsGame(
    population_size=population_size,
    rich_fraction=rich_fraction,
    endowment_rich=1.7,
```

```
    endowment_poor=0.3,
    group_size=group_size,
    threshold=threshold,
    risk=0.2,
    contribution_factor=contribution_factor,
    homophily=0,
    beta=5,
    mu=mu,
    pi_max=4.2e-2,
    gos_max=0.25
)

X, Y, U, V, stationary_distribution = game.compute_gradient_of_selection()
plot_gradient_of_selection(X, Y, U, V, stationary_distribution, "$Z_R$ = $Z_P$,␣
 ↪$r=0.2$, $h=0$")
```



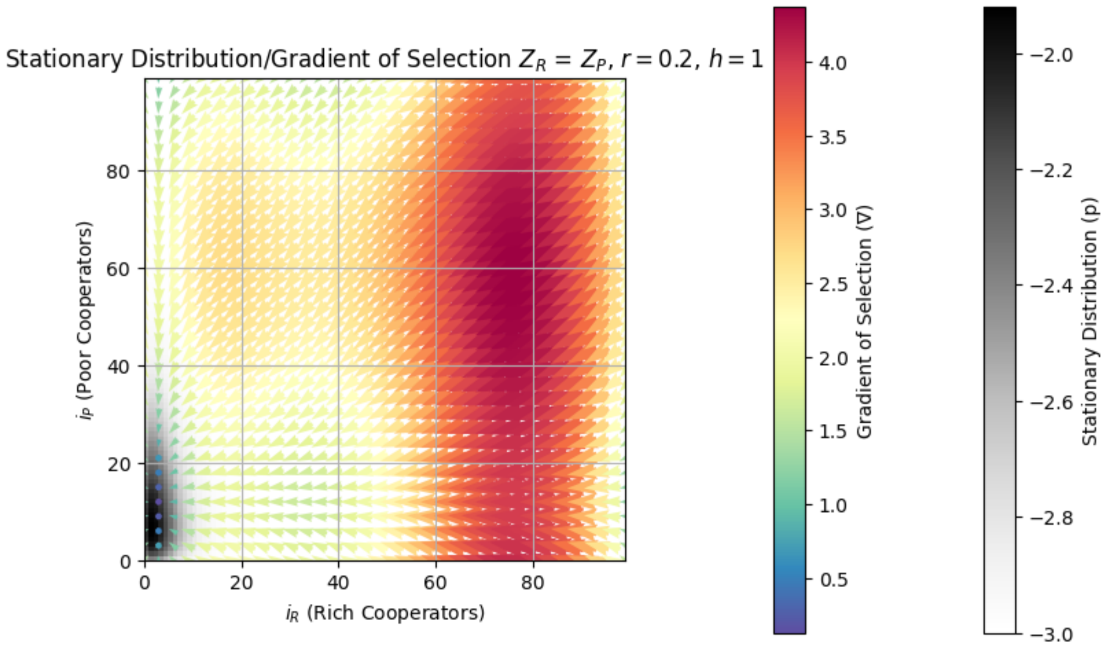
[17]:
```
# Figure S2 - B
game = ClimateThresholdPublicGoodsGame(
    population_size=population_size,
    rich_fraction=rich_fraction,
    endowment_rich=endowment_rich,
    endowment_poor=endowment_poor,
    group_size=group_size,
    threshold=threshold,
    risk=0.2,
```

```
    contribution_factor=contribution_factor,
    homophily=0.7,
    beta=5,
    mu=mu,
    pi_max=5.3e-2,
    gos_max=0.12
)

X, Y, U, V, stationary_distribution = game.compute_gradient_of_selection()
plot_gradient_of_selection(X, Y, U, V, stationary_distribution, "$Z_R$ = $Z_P$,␣
  ↪$r=0.2$, $h=0.7$")
```



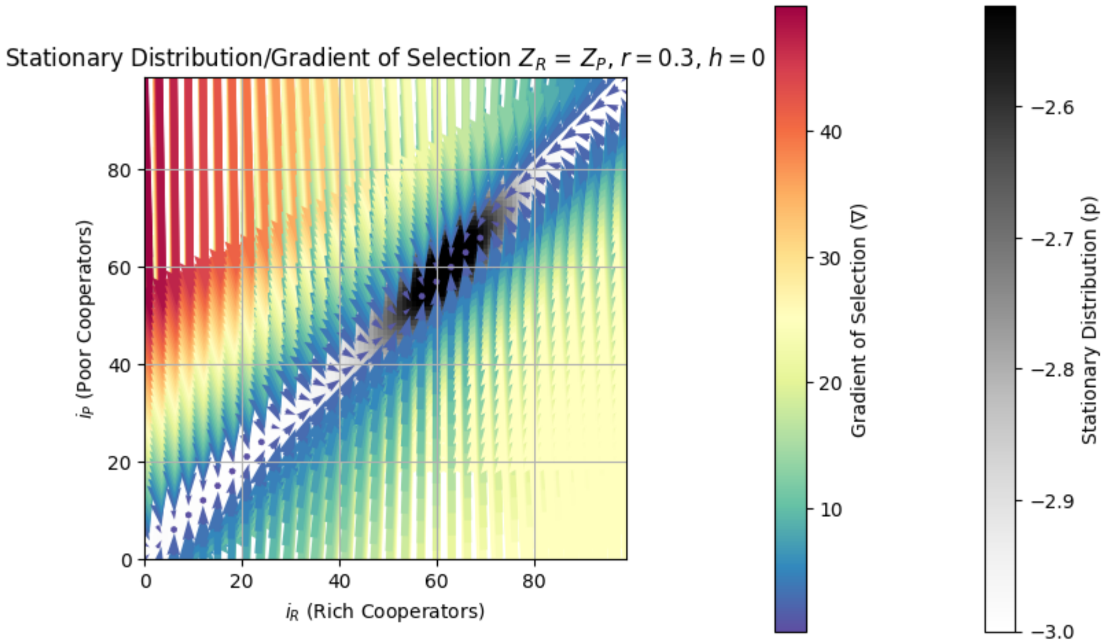
[19]:
```
# Figure S2 - C
game = ClimateThresholdPublicGoodsGame(
    population_size=population_size,
    rich_fraction=rich_fraction,
    endowment_rich=endowment_rich,
    endowment_poor=endowment_poor,
    group_size=group_size,
    threshold=threshold,
    risk=0.2,
    contribution_factor=contribution_factor,
    homophily=1,
    beta=5,
    mu=mu,
```

```
    pi_max=3.4e-2,
    gos_max=0.02
)

X, Y, U, V, stationary_distribution = game.compute_gradient_of_selection()
plot_gradient_of_selection(X, Y, U, V, stationary_distribution, "$Z_R$ = $Z_P$,␣
↪$r=0.2$, $h=1$")
```

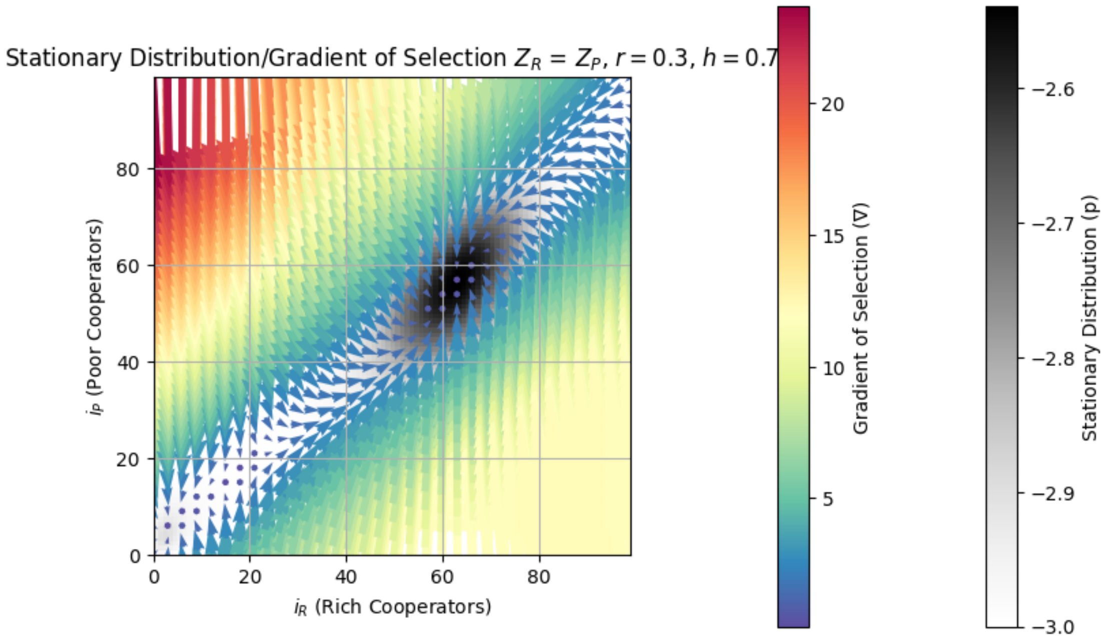


[20]:
```
# Figure S2 - D
game = ClimateThresholdPublicGoodsGame(
    population_size=population_size,
    rich_fraction=rich_fraction,
    endowment_rich=endowment_rich,
    endowment_poor=endowment_poor,
    group_size=group_size,
    threshold=threshold,
    risk=0.3,
    contribution_factor=contribution_factor,
    homophily=0,
    beta=5,
    mu=mu,
    pi_max=0.2e-2,
    gos_max=0.25
)
```

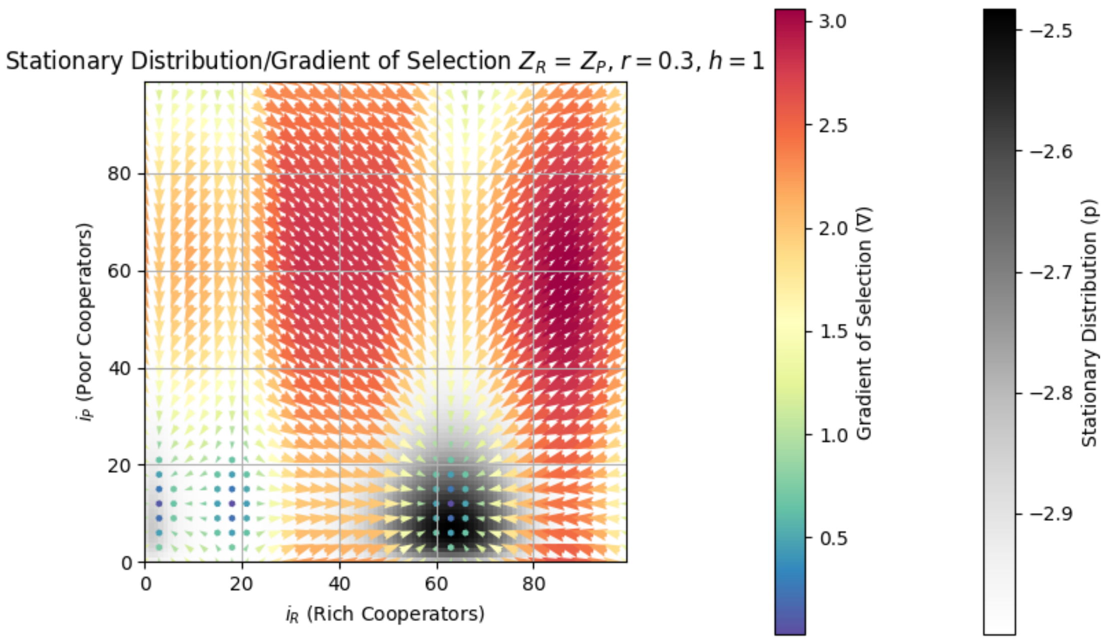
```
X, Y, U, V, stationary_distribution = game.compute_gradient_of_selection()
plot_gradient_of_selection(X, Y, U, V, stationary_distribution, "$Z_R$ = $Z_P$,␣
↪$r=0.3$, $h=0$")
```



[21]:
```
# Figure S2 - E
game = ClimateThresholdPublicGoodsGame(
    population_size=population_size,
    rich_fraction=rich_fraction,
    endowment_rich=endowment_rich,
    endowment_poor=endowment_poor,
    group_size=group_size,
    threshold=threshold,
    risk=0.3,
    contribution_factor=contribution_factor,
    homophily=0.7,
    beta=5,
    mu=mu,
    pi_max=0.7e-2,
    gos_max=0.12
)

X, Y, U, V, stationary_distribution = game.compute_gradient_of_selection()
plot_gradient_of_selection(X, Y, U, V, stationary_distribution, "$Z_R$ = $Z_P$,␣
↪$r=0.3$, $h=0.7$")
```



[22]:
```
# Figure S2 - F
game = ClimateThresholdPublicGoodsGame(
    population_size=population_size,
    rich_fraction=rich_fraction,
    endowment_rich=endowment_rich,
    endowment_poor=endowment_poor,
    group_size=group_size,
    threshold=threshold,
    risk=0.3,
    contribution_factor=contribution_factor,
    homophily=1,
    beta=5,
    mu=mu,
    pi_max=1.6e-2,
    gos_max=0.02
)

X, Y, U, V, stationary_distribution = game.compute_gradient_of_selection()
plot_gradient_of_selection(X, Y, U, V, stationary_distribution, "$Z_R$ = $Z_P$, 
    ↪$r=0.3$, $h=1$")
```



3 Average group achievement as a function of risk

[17]:
```
# Default parameters again
population_size = 160 # NOT 200!
rich_fraction = 0.2
endowment_rich = 2.5
endowment_poor = 0.625
group_size = 6
threshold = 3
contribution_factor = 0.1
risk = 0
homophily = 0
beta = 3
mu = 0.01
```

[18]:
```
# Range of risk values
risks = np.linspace(0, 1, 9) # Reduced because this cell takes very long to␣
↪compute

# Compute group achievement for each scenario
eta_no_inequality = []
eta_with_inequality_h0 = []
eta_with_inequality_h1 = []

# Case 1: No inequality
```

```python
endowment_rich = 1
endowment_poor = 1
homophily = 0

print("Case 1")
for r in risks:
    print(f"Risk: {r}")
    game_1 = ClimateThresholdPublicGoodsGame(
        population_size=population_size,
        rich_fraction=rich_fraction,
        endowment_rich=endowment_rich,
        endowment_poor=endowment_poor,
        group_size=group_size,
        threshold=threshold,
        risk=r,
        contribution_factor=contribution_factor,
        homophily=homophily,
        beta=beta,
        mu=mu
    )

    eta_G = game_1.compute_eta_G()
    print(eta_G)
    eta_no_inequality.append(eta_G)

print(eta_no_inequality)

# Case 2: With inequality, homophily = 0
endowment_rich = 2.5
endowment_poor = 0.625
homophily = 0

print("Case 2")
for r in risks:
    print(f"Risk: {r}")
    game_2 = ClimateThresholdPublicGoodsGame(
        population_size=population_size,
        rich_fraction=rich_fraction,
        endowment_rich=endowment_rich,
        endowment_poor=endowment_poor,
        group_size=group_size,
        threshold=threshold,
        risk=r,
        contribution_factor=contribution_factor,
        homophily=homophily,
        beta=beta,
        mu=mu
```

```
    )

    eta_G = game_2.compute_eta_G()
    print(eta_G)
    eta_with_inequality_h0.append(eta_G)

print(eta_with_inequality_h0)

# Case 3: With inequality, homophily = 1
homophily = 1

print("Case 3")
for r in risks:
    print(f"Risk: {r}")
    game_3 = ClimateThresholdPublicGoodsGame(
        population_size=population_size,
        rich_fraction=rich_fraction,
        endowment_rich=endowment_rich,
        endowment_poor=endowment_poor,
        group_size=group_size,
        threshold=threshold,
        risk=r,
        contribution_factor=contribution_factor,
        homophily=homophily,
        beta=beta,
        mu=mu
    )

    eta_G = game_3.compute_eta_G()
    print(eta_G)
    eta_with_inequality_h1.append(eta_G)

print(eta_with_inequality_h1)
```

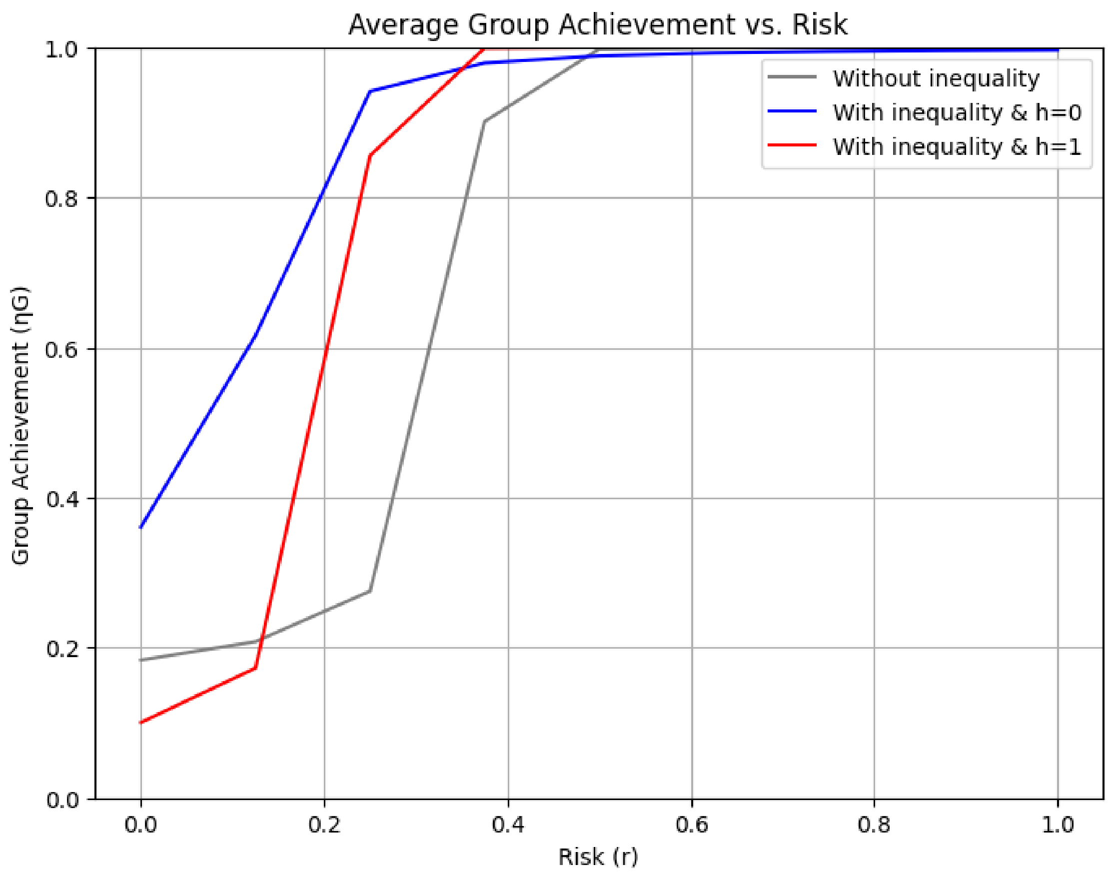
```
Case 1
Risk: 0.0
0.18344936758455913
Risk: 0.125
0.20804626281626493
Risk: 0.25
0.2753007281082099
Risk: 0.375
0.9007244391196239
Risk: 0.5
0.9975174300780394
Risk: 0.625
0.9988159466703199
Risk: 0.75
```

```
0.9992677097810718
Risk: 0.875
0.99949359978387
Risk: 1.0
0.9996246236480638
[np.float64(0.18344936758455913), np.float64(0.20804626281626493),
np.float64(0.2753007281082099), np.float64(0.9007244391196239),
np.float64(0.9975174300780394), np.float64(0.9988159466703199),
np.float64(0.9992677097810718), np.float64(0.99949359978387),
np.float64(0.9996246236480638)]
Case 2
Risk: 0.0
0.36082845575665895
Risk: 0.125
0.6157595556331266
Risk: 0.25
0.941093932860475
Risk: 0.375
0.9788798945042947
Risk: 0.5
0.9883779023322715
Risk: 0.625
0.9922637337685742
Risk: 0.75
0.9942871353183634
Risk: 0.875
0.9955029982884523
Risk: 1.0
0.9963052944180141
[np.float64(0.36082845575665895), np.float64(0.6157595556331266),
np.float64(0.941093932860475), np.float64(0.9788798945042947),
np.float64(0.9883779023322715), np.float64(0.9922637337685742),
np.float64(0.9942871353183634), np.float64(0.9955029982884523),
np.float64(0.9963052944180141)]
Case 3
Risk: 0.0
0.10047390047954155
Risk: 0.125
0.172803201794725
Risk: 0.25
0.8554594115182029
Risk: 0.375
0.9987162567938156
Risk: 0.5
0.9996084818110151
Risk: 0.625
0.999790212505055
Risk: 0.75
```

```
0.9998529277810636
Risk: 0.875
0.999864666978211
Risk: 1.0
0.999829837341715
[np.float64(0.10047390047954155), np.float64(0.172803201794725),
np.float64(0.8554594115182029), np.float64(0.9987162567938156),
np.float64(0.9996084818110151), np.float64(0.999790212505055),
np.float64(0.9998529277810636), np.float64(0.999864666978211),
np.float64(0.999829837341715)]
```

[19]:

```
# Plot the results
plt.figure(figsize=(8, 6))
plt.plot(risks, eta_no_inequality, label="Without inequality", color="gray")
plt.plot(risks, eta_with_inequality_h0, label="With inequality & h=0",␣
 ↪color="blue")
plt.plot(risks, eta_with_inequality_h1, label="With inequality & h=1",␣
 ↪color="red")
plt.xlabel("Risk (r)")
plt.ylabel("Group Achievement ( G)")
plt.title("Average Group Achievement vs. Risk")
plt.ylim(0, 1)
plt.legend()
plt.grid(True)
plt.show()
```




```
[ ]:
```

4 Evolving Rich and Poor subpopulations while freezing the other

This is done with $r = 0.3$, but we also plot results for $r = 0.9$. For both cases, we look at when $4 \times Z_R = Z_P$ (our primary case, with a 20-80 split) and when $Z_P = Z_R$. Homophily, h, is kept at 0 throughout this experiment.

```
[5]: class ClimateThresholdPublicGoodsFrozenGame(ClimateThresholdPublicGoodsGame):
         def __init__(self,
                      population_size=200,
                      rich_fraction=0.2,
                      endowment_rich=2.5,
                      endowment_poor=0.625,
                      group_size=6,
                      threshold=3,
                      risk=0.6,
                      contribution_factor=0.1,
                      homophily=0.4,
```

```
                 beta=5,
                 mu=0.1,
                 verbose=False,
                 fixed_C_R=None,  # Fixed rich cooperators
                 fixed_C_P=None   # Fixed poor cooperators
                ):
        self.Z = population_size

        self.rich_fraction = rich_fraction
        self.endowment_rich = endowment_rich
        self.endowment_poor = endowment_poor
        self.N = group_size

        self.threshold = threshold
        self.risk = risk
        self.contribution_factor = contribution_factor
        self.homophily = homophily
        self.beta = beta
        self.mu = mu

        self.verbose = verbose

        self.Z_R = int(rich_fraction * population_size)
        self.Z_P = population_size - self.Z_R

        assert endowment_rich >= endowment_poor, "The Marxists are back! Rise␣
↪of the proletariat!" # Again!

        # Average endowment is used to calculate the threshold for success
        self.average_endowment = (self.Z_R * endowment_rich + self.Z_P *␣
↪endowment_poor) / population_size

        if self.verbose:
            print("Average Endowment", self.average_endowment)
            print("Mcb", self.average_endowment * self.threshold * self.
↪contribution_factor)

        # Create/clear the cache dict for storing fitness
        self.fitness_dict = dict()
        self.stationary_distribution = None
        self.transition_matrix = None

        self.fixed_C_R = fixed_C_R
        self.fixed_C_P = fixed_C_P

        self.stationary_distribution = None # Not used in this class since␣
↪we're just dealing with gradients
```

```
        self.transition_matrix = None

        self.generate_states()

    def generate_states(self):
        """
        Generate all possible states (C_R, C_P) or enforce fixed values if␣
↪provided!
        """
        self.states = []
        if self.fixed_C_R is not None and self.fixed_C_P is not None:
            # If both are fixed, we have exactly one state
            self.states.append((self.fixed_C_R, self.fixed_C_P))
        elif self.fixed_C_R is not None:
            # If rich cooperators are fixed, vary poor cooperators
            for C_P in range(self.Z_P + 1):
                self.states.append((self.fixed_C_R, C_P))
        elif self.fixed_C_P is not None:
            # If poor cooperators are fixed, vary rich cooperators
            for C_R in range(self.Z_R + 1):
                self.states.append((C_R, self.fixed_C_P))
        else:
            # Default: vary both
            for C_R in range(self.Z_R + 1):
                for C_P in range(self.Z_P + 1):
                    self.states.append((C_R, C_P))

        if self.verbose:
            print(self.states)

        # Now build 2D-state <-> scalar representation for easy mapping
        self.indexed_states = dict()
        for index, (C_R, C_P) in enumerate(self.states):
            self.indexed_states[(C_R, C_P)] = (index, self.
↪transform_to_scalar(C_R, C_P))

        return self.states

    def build_transition_matrix(self):
        """
        NOTE: Copied from parent class, adapted to the case where we freeze one␣
↪subpopulation!

        Build the partial transition matrix along the diagonal with neighboring␣
↪states,
        respecting the frozen subpopulation if applicable.
        """
```

```
        self.generate_states()  # Ensure states and indexing are up-to-date
        num_states = len(self.states)  # Number of valid states
        transition_matrix = np.zeros((num_states, num_states))

        # Iterate over all states
        for (C_R_i, C_P_i), (i, scalar_i) in self.indexed_states.items():
            # Compute fitness for the current state
            f_R_C = self.calculate_fitness_rich_C(C_R_i, C_P_i)
            f_R_D = self.calculate_fitness_rich_D(C_R_i, C_P_i)
            f_P_C = self.calculate_fitness_poor_C(C_R_i, C_P_i)
            f_P_D = self.calculate_fitness_poor_D(C_R_i, C_P_i)

            if self.verbose:
                print(f"For (C_R, C_P)={C_R_i}, {C_P_i}, \t (f_R_C, f_R_D,␣
↪f_P_C, f_P_D) = ({f_R_C}, {f_R_D}, {f_P_C}, {f_P_D})")

            # Determine neighbors based on the frozen subpopulation
            neighbors = []
            if self.fixed_C_P is not None:  # Rich subpopulation can vary
                neighbors += [(C_R_i + 1, self.fixed_C_P), (C_R_i - 1, self.
↪fixed_C_P)]
            if self.fixed_C_R is not None:  # Poor subpopulation can vary
                neighbors += [(self.fixed_C_R, C_P_i + 1), (self.fixed_C_R,␣
↪C_P_i - 1)]

            # Filter neighbors to ensure valid transitions within bounds
            neighbors = [(C_R_j, C_P_j) for (C_R_j, C_P_j) in neighbors
                         if 0 <= C_R_j <= self.Z_R and 0 <= C_P_j <= self.Z_P]

            total_out_prob = 0.0
            for (C_R_j, C_P_j) in neighbors:
                j, scalar_j = self.indexed_states[(C_R_j, C_P_j)]

                # Determine the deltas
                delta_C_R = C_R_j - C_R_i
                delta_C_P = C_P_j - C_P_i

                # Compute the probability of transitioning from state i to j
                prob = self.compute_joint_transition_probability(
                    C_R_i, C_P_i, C_R_j, C_P_j, delta_C_R, delta_C_P, f_R_C,␣
↪f_R_D, f_P_C, f_P_D
                )
                if self.verbose:
                    print(f"Transition from {C_R_i, C_P_i} to {C_R_j, C_P_j}:␣
↪{prob}")
                transition_matrix[i, j] = prob
                total_out_prob += prob
```

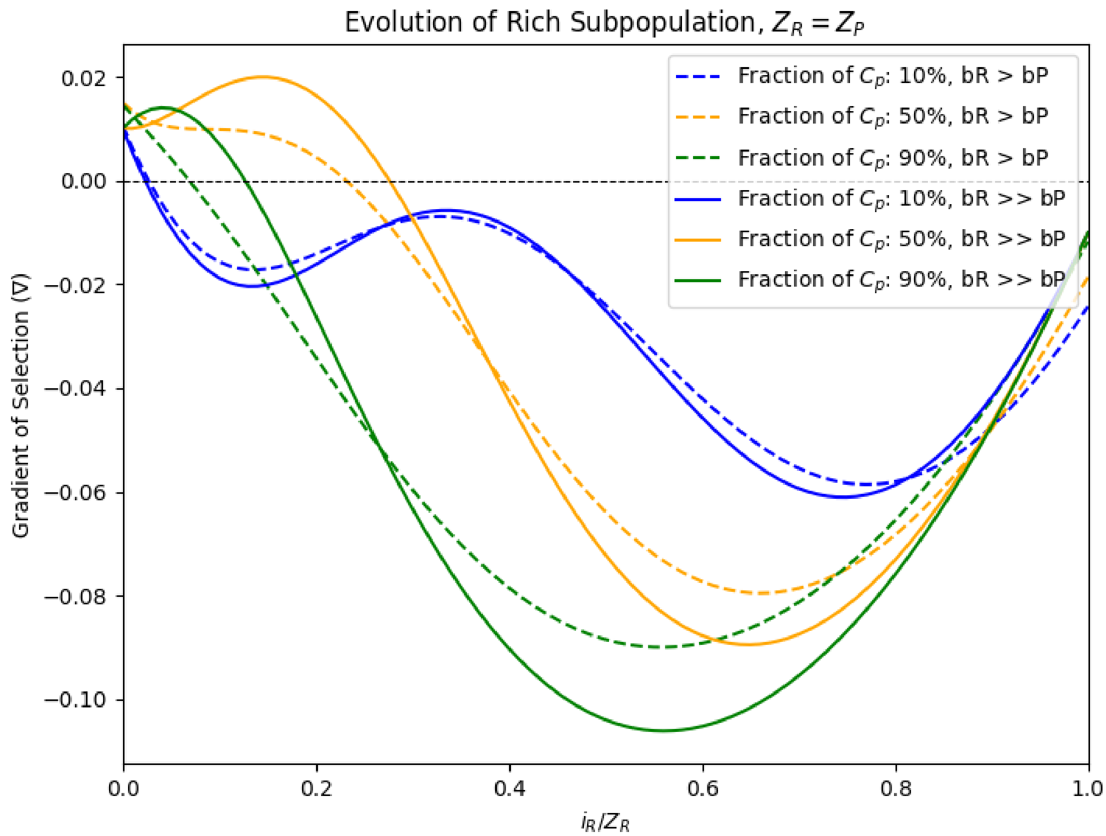
```
            # The diagonal entry ensures that rows sum to 1
            transition_matrix[i, i] = 1.0 - total_out_prob

        return transition_matrix

    def compute_gradient_of_selection(self):
        """
        Compute the gradient of selection (GoS) for each configuration with the␣
↪frozen subpopulation!
        """
        # Get transition matrix
        transition_matrix = self.get_transition_matrix()

        # Initialize gradient arrays
        U = np.zeros((self.Z_R + 1, self.Z_P + 1))  # Gradient in i_R direction
        V = np.zeros((self.Z_R + 1, self.Z_P + 1))  # Gradient in i_P direction

        for (C_R_i, C_P_i), (i, scalar_i) in self.indexed_states.items():
            if self.fixed_C_P is not None: # If C_P is fixed, C_R can change
                # Get T_{i,R}+ and T_{i,R}-
                T_R_plus = transition_matrix[i, self.indexed_states.get((C_R_i␣
↪+ 1, self.fixed_C_P), (None, None))[0]] if C_R_i + 1 <= self.Z_R else 0
                T_R_minus = transition_matrix[i, self.indexed_states.get((C_R_i␣
↪- 1, self.fixed_C_P), (None, None))[0]] if C_R_i - 1 >= 0 else 0
                U[C_R_i, C_P_i] = T_R_plus - T_R_minus
            elif self.fixed_C_R is not None: # If C_R is fixed, C_P can change
                # Get T_{i,P}+ and T_{i,P}-
                T_P_plus = transition_matrix[i, self.indexed_states.get((self.
↪fixed_C_R, C_P_i + 1), (None, None))[0]] if C_P_i + 1 <= self.Z_P else 0
                T_P_minus = transition_matrix[i, self.indexed_states.get((self.
↪fixed_C_R, C_P_i - 1), (None, None))[0]] if C_P_i - 1 >= 0 else 0
                V[C_R_i, C_P_i] = T_P_plus - T_P_minus

        return U, V # U is for GoS_R & V is for GoS_P
```

[6]:
```
# Default parameters again
population_size = 200
rich_fraction = 0.5
endowment_rich = 2.5
endowment_poor = 0.625
group_size = 6
threshold = 3
contribution_factor = 0.1
risk = 0
homophily = 0
beta = 3
```

```
mu = 0.01
```

[46]:
```
# Figure S1 - A from SI Text

# Parameters for the game
fractions_C_poor = [0.1, 0.5, 0.9]  # Fixed fractions of poor cooperators
benefit_ratios = ["bR > bP", "bR >> bP"]  # Different benefit scenarios

# Colors and linestyles for plotting
colors = ["blue", "orange", "green"]
linestyles = ["--", "-"]

# Initialize a dictionary to store results
results_gos_rich = {}

# Population sizes
Z_R = rich_fraction * population_size
Z_P = population_size - Z_R

# Iterate over different benefit ratios
for benefit_ratio in benefit_ratios:
    gos_rich_values = []

    # Iterate over different fixed fractions of poor cooperators
    for fraction_C_poor in fractions_C_poor:
        fixed_C_P = int(fraction_C_poor * Z_P)

        # Initialize the game with the specific configuration
        game = ClimateThresholdPublicGoodsFrozenGame(
            population_size=200,
            rich_fraction=0.5,
            endowment_rich=1.35 if benefit_ratio == "bR > bP" else 1.75,
            endowment_poor=0.9125 if benefit_ratio == "bR > bP" else 0.8125,
            group_size=10,
            threshold=3,
            risk=0.3,
            contribution_factor=0.1,
            beta=10,
            mu=0.01,
            verbose=False,
            fixed_C_P=fixed_C_P  # Fix the poor cooperators
        )

        # Compute the gradients
        U, _ = game.compute_gradient_of_selection()

        # Extract GoS for the rich subpopulation for all i_R values
```

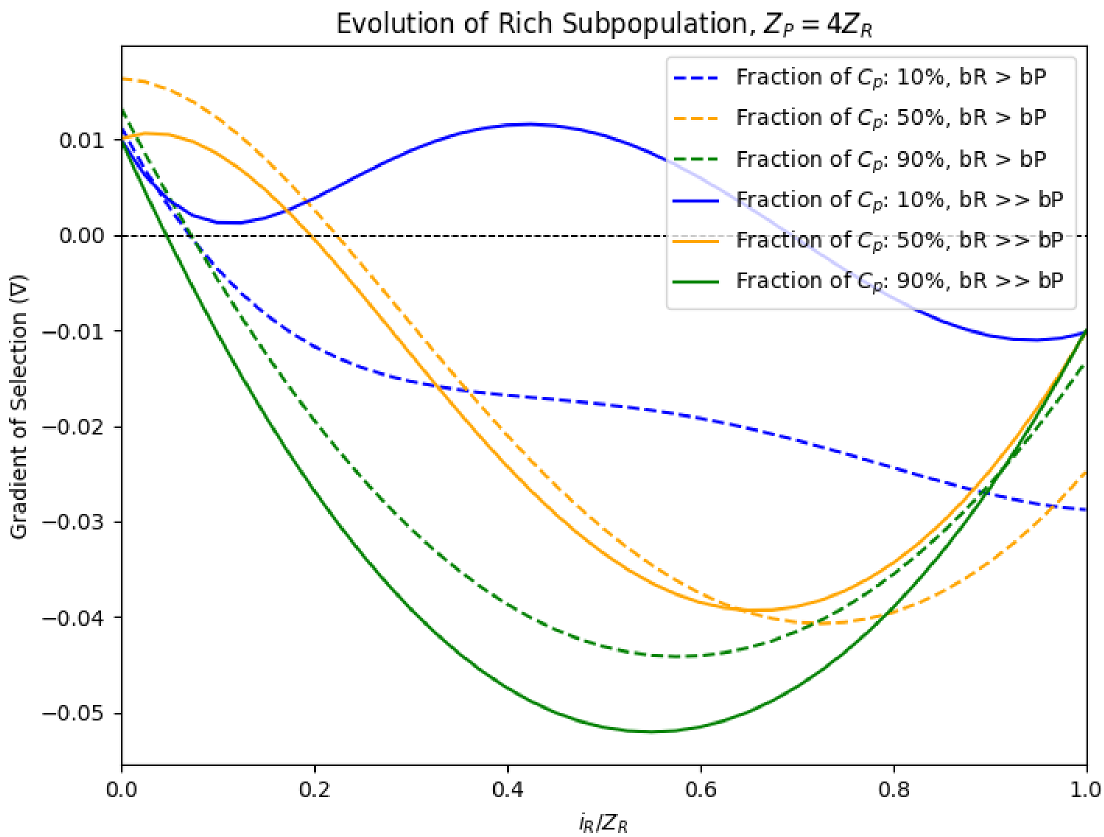
```
        gos_rich = [U[i_R, fixed_C_P] for i_R in range(int(Z_R) + 1)]
        gos_rich_values.append(gos_rich)

    # Store the results
    results_gos_rich[benefit_ratio] = gos_rich_values

# Plotting the results
plt.figure(figsize=(8, 6))
plt.xlim(0, 1)

x_rich = np.linspace(0, 1, int(Z_R) + 1)  # x-axis is i_R / Z_R

# Iterate over benefit ratios and plot
for benefit_idx, benefit_ratio in enumerate(benefit_ratios):
    for i, fraction_C_poor in enumerate(fractions_C_poor):
        label = f"Fraction of $C_p$: {int(fraction_C_poor*100)}%, {benefit_ratio}"
        plt.plot(x_rich, results_gos_rich[benefit_ratio][i], color=colors[i], linestyle=linestyles[benefit_idx], label=label)

# Add labels and legend
plt.axhline(0, color='black', linestyle='--', linewidth=0.8)
plt.xlabel("$i_R / Z_R$")
plt.ylabel("Gradient of Selection ( )")
plt.title("Evolution of Rich Subpopulation, $Z_R = Z_P$")
plt.legend()
plt.show()
```



[42]:
```
# Figure S1 - B from SI Text

# Population sizes
rich_fraction = 0.2
Z_R = rich_fraction * population_size
Z_P = population_size - Z_R

# Parameters for the game
fractions_C_poor = [0.1, 0.5, 0.9]  # Fixed fractions of poor cooperators
benefit_ratios = ["bR > bP", "bR >> bP"]  # Different benefit scenarios

# Colors and linestyles for plotting
colors = ["blue", "orange", "green"]
linestyles = ["--", "-"]

# Initialize a dictionary to store results
results_gos_rich = {}

# Iterate over different benefit ratios
for benefit_ratio in benefit_ratios:
```

```
    gos_rich_values = []

    # Iterate over different fixed fractions of poor cooperators
    for fraction_C_poor in fractions_C_poor:
        fixed_C_P = int(fraction_C_poor * Z_P)

        # Initialize the game with the specific configuration
        game = ClimateThresholdPublicGoodsFrozenGame(
            population_size=200,
            rich_fraction=0.2,
            endowment_rich=1.35 if benefit_ratio == "bR > bP" else 1.75,
            endowment_poor=0.9125 if benefit_ratio == "bR > bP" else 0.8125,
            group_size=10,
            threshold=3,
            risk=0.3,
            contribution_factor=0.1,
            beta=10,
            mu=0.01,
            verbose=False,
            fixed_C_P=fixed_C_P  # Fix the poor cooperators
        )

        # Compute the gradients
        U, _ = game.compute_gradient_of_selection()

        # Extract GoS for the rich subpopulation for all i_R values
        gos_rich = [U[i_R, fixed_C_P] for i_R in range(int(Z_R) + 1)]
        gos_rich_values.append(gos_rich)

    # Store the results
    results_gos_rich[benefit_ratio] = gos_rich_values

# Plotting the results
plt.figure(figsize=(8, 6))
plt.xlim(0, 1)

x_rich = np.linspace(0, 1, int(Z_R) + 1)  # x-axis is i_R / Z_R

# Iterate over benefit ratios and plot
for benefit_idx, benefit_ratio in enumerate(benefit_ratios):
    for i, fraction_C_poor in enumerate(fractions_C_poor):
        label = f"Fraction of $C_p$: {int(fraction_C_poor*100)}%,␣
↪{benefit_ratio}"
        plt.plot(x_rich, results_gos_rich[benefit_ratio][i], color=colors[i],␣
↪linestyle=linestyles[benefit_idx], label=label)

# Add labels and legend
```

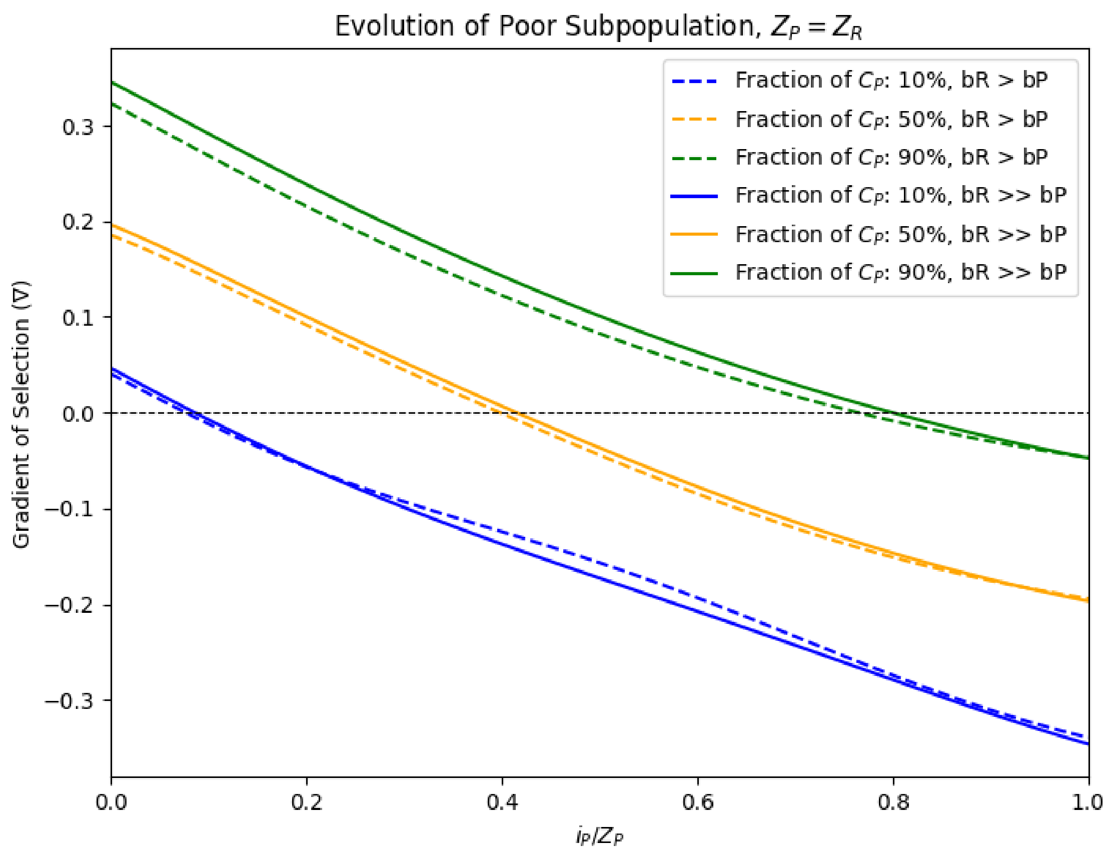
```python
plt.axhline(0, color='black', linestyle='--', linewidth=0.8)
plt.xlabel("$i_R / Z_R$")
plt.ylabel("Gradient of Selection ( )")
plt.title("Evolution of Rich Subpopulation, $Z_P = 4Z_R$")
plt.legend()
plt.show()
```



[7]:
```python
# Figure S1 - C from SI Text

# Population sizes
rich_fraction = 0.5
Z_R = rich_fraction * population_size
Z_P = population_size - Z_R

# Parameters for the game
fractions_C_rich = [0.1, 0.5, 0.9]  # Fixed fractions of poor cooperators
benefit_ratios = ["bR > bP", "bR >> bP"]  # Different benefit scenarios

# Colors and linestyles for plotting
colors = ["blue", "orange", "green"]
```

```python
linestyles = ["--", "-"]

# Initialize a dictionary to store results
results_gos_poor = {}

# Iterate over different benefit ratios
for benefit_ratio in benefit_ratios:
    gos_poor_values = []

    # Iterate over different fixed fractions of poor cooperators
    for fraction_C_rich in fractions_C_rich:
        fixed_C_R = int(fraction_C_rich * population_size * rich_fraction)

        # Initialize the game with the specific configuration
        game = ClimateThresholdPublicGoodsFrozenGame(
            population_size=200,
            rich_fraction=0.5,
            endowment_rich=1.35 if benefit_ratio == "bR > bP" else 1.75,
            endowment_poor=0.9125 if benefit_ratio == "bR > bP" else 0.8125,
            group_size=10,
            threshold=3,
            risk=0.3,
            contribution_factor=0.1,
            beta=8,
            mu=0.01,
            verbose=False,
            fixed_C_R=fixed_C_R,  # Fix the rich cooperators
            fixed_C_P=None
        )

        # Compute the gradients
        _, V = game.compute_gradient_of_selection()

        # Extract GoS for the rich subpopulation for all i_R values
        gos_poor = [V[fixed_C_R, i_P] for i_P in range(game.Z_P + 1)]
        gos_poor_values.append(gos_poor)

    # Store the results
    results_gos_poor[benefit_ratio] = gos_poor_values

# Plotting the results
plt.figure(figsize=(8, 6))
plt.xlim(0, 1)

x_poor = np.linspace(0, 1, game.Z_P + 1)  # x-axis is i_P / Z_P

# Iterate over benefit ratios and plot
```

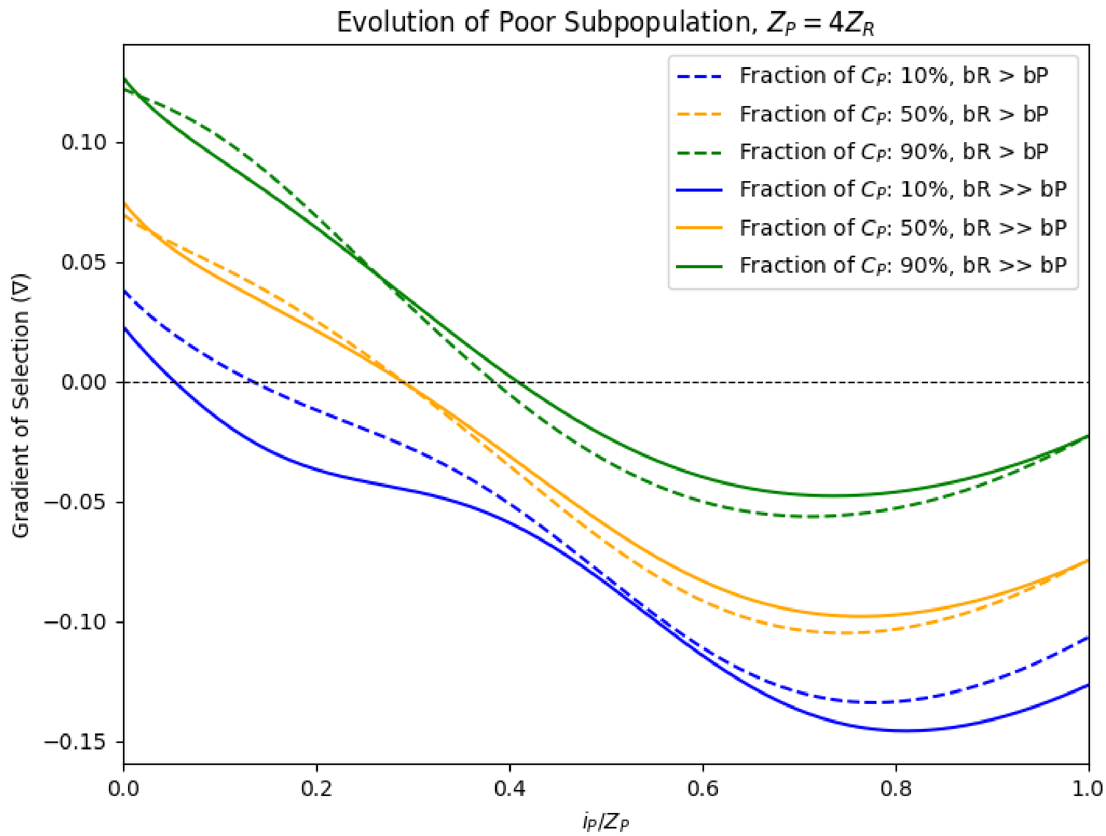
```python
for benefit_idx, benefit_ratio in enumerate(benefit_ratios):
    for i, fraction_C_rich in enumerate(fractions_C_rich):
        label = f"Fraction of $C_P$: {int(fraction_C_rich*100)}%,␣
↪{benefit_ratio}"
        plt.plot(x_poor, results_gos_poor[benefit_ratio][i], color=colors[i],␣
↪linestyle=linestyles[benefit_idx], label=label)

# Add labels and legend
plt.axhline(0, color='black', linestyle='--', linewidth=0.8)
plt.xlabel("$i_P / Z_P$")
plt.ylabel("Gradient of Selection ( )")
plt.title("Evolution of Poor Subpopulation, $Z_P = Z_R$")
plt.legend()
plt.show()
```



[8]:
```python
# Figure S1 - D from SI Text

# Population sizes
rich_fraction = 0.2
```

```python
Z_R = rich_fraction * population_size
Z_P = population_size - Z_R

# Parameters for the game
fractions_C_rich = [0.1, 0.5, 0.9]  # Fixed fractions of poor cooperators
benefit_ratios = ["bR > bP", "bR >> bP"]  # Different benefit scenarios

# Colors and linestyles for plotting
colors = ["blue", "orange", "green"]
linestyles = ["--", "-"]

# Initialize a dictionary to store results
results_gos_poor = {}

# Iterate over different benefit ratios
for benefit_ratio in benefit_ratios:
    gos_poor_values = []

    # Iterate over different fixed fractions of poor cooperators
    for fraction_C_rich in fractions_C_rich:
        fixed_C_R = int(fraction_C_rich * game.Z_R)

        # Initialize the game with the specific configuration
        game = ClimateThresholdPublicGoodsFrozenGame(
            population_size=200,
            rich_fraction=0.2,
            endowment_rich=1.35 if benefit_ratio == "bR > bP" else 1.75,
            endowment_poor=0.9125 if benefit_ratio == "bR > bP" else 0.8125,
            group_size=10,
            threshold=3,
            risk=0.3,
            contribution_factor=0.1,
            beta=10,
            mu=0.01,
            verbose=False,
            fixed_C_R=fixed_C_R  # Fix the poor cooperators
        )

        # Compute the gradients
        _, V = game.compute_gradient_of_selection()

        # Extract GoS for the rich subpopulation for all i_R values
        gos_poor = [V[fixed_C_R, i_P] for i_P in range(game.Z_P + 1)]
        gos_poor_values.append(gos_poor)

    # Store the results
    results_gos_poor[benefit_ratio] = gos_poor_values
```

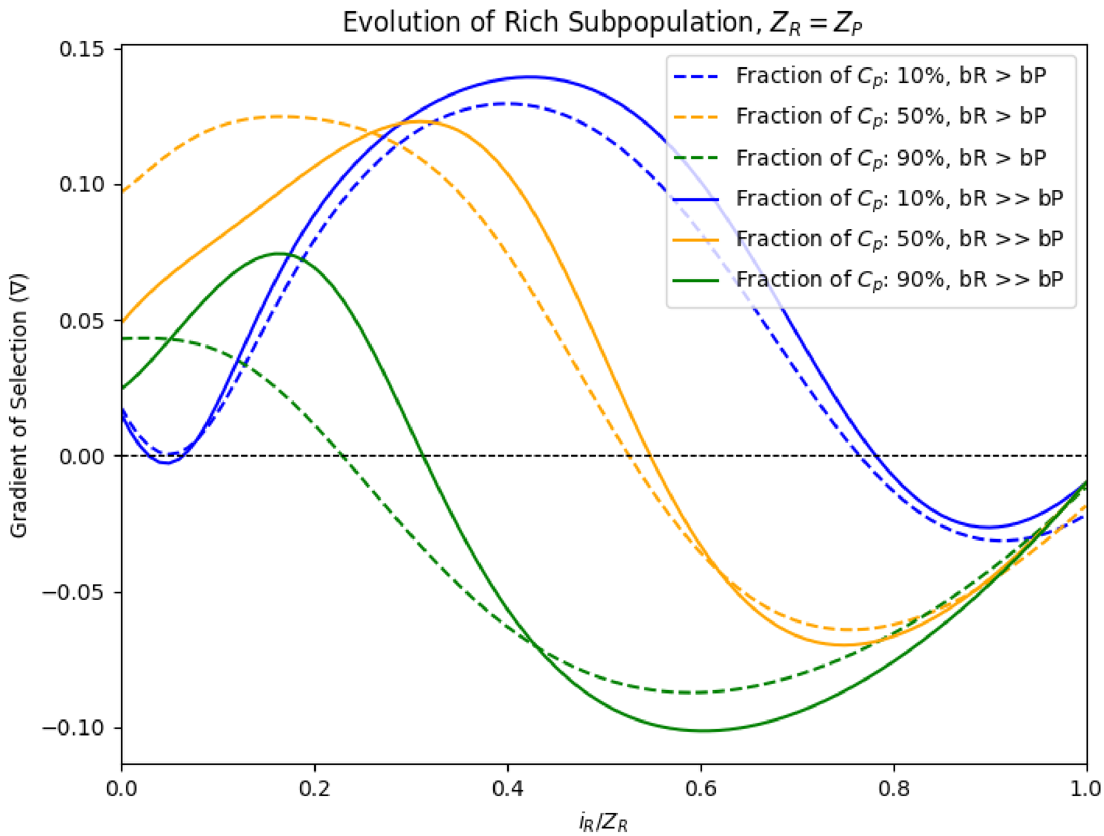
```
# Plotting the results
plt.figure(figsize=(8, 6))
plt.xlim(0, 1)

x_poor = np.linspace(0, 1, game.Z_P + 1)  # x-axis is i_P / Z_P

# Iterate over benefit ratios and plot
for benefit_idx, benefit_ratio in enumerate(benefit_ratios):
    for i, fraction_C_rich in enumerate(fractions_C_rich):
        label = f"Fraction of $C_P$: {int(fraction_C_rich*100)}%,␣
↪{benefit_ratio}"
        plt.plot(x_poor, results_gos_poor[benefit_ratio][i], color=colors[i],␣
↪linestyle=linestyles[benefit_idx], label=label)

# Add labels and legend
plt.axhline(0, color='black', linestyle='--', linewidth=0.8)
plt.xlabel("$i_P / Z_P$")
plt.ylabel("Gradient of Selection ( )")
plt.title("Evolution of Poor Subpopulation, $Z_P = 4Z_R$")
plt.legend()
plt.show()
```



[43]:

```python
# Testing some additional params (high risk, 0.9) with Z_R = Z_P

# Parameters for the game
fractions_C_poor = [0.1, 0.5, 0.9]  # Fixed fractions of poor cooperators
benefit_ratios = ["bR > bP", "bR >> bP"]  # Different benefit scenarios

# Colors and linestyles for plotting
colors = ["blue", "orange", "green"]
linestyles = ["--", "-"]

# Initialize a dictionary to store results
results_gos_rich = {}

# Population sizes
Z_R = rich_fraction * population_size
Z_P = population_size - Z_R

# Iterate over different benefit ratios
for benefit_ratio in benefit_ratios:
    gos_rich_values = []

    # Iterate over different fixed fractions of poor cooperators
    for fraction_C_poor in fractions_C_poor:
        fixed_C_P = int(fraction_C_poor * Z_P)

        # Initialize the game with the specific configuration
        game = ClimateThresholdPublicGoodsFrozenGame(
            population_size=200,
            rich_fraction=0.5,
            endowment_rich=1.35 if benefit_ratio == "bR > bP" else 1.75,
            endowment_poor=0.9125 if benefit_ratio == "bR > bP" else 0.8125,
            group_size=10,
            threshold=3,
            risk=0.9,
            contribution_factor=0.1,
            beta=10,
            mu=0.01,
            verbose=False,
            fixed_C_P=fixed_C_P  # Fix the poor cooperators
        )

        # Compute the gradients
        U, _ = game.compute_gradient_of_selection()
```

```
        # Extract GoS for the rich subpopulation for all i_R values
        gos_rich = [U[i_R, fixed_C_P] for i_R in range(int(Z_R) + 1)]
        gos_rich_values.append(gos_rich)

    # Store the results
    results_gos_rich[benefit_ratio] = gos_rich_values

# Plotting the results
plt.figure(figsize=(8, 6))
plt.xlim(0, 1)

x_rich = np.linspace(0, 1, int(Z_R) + 1)  # x-axis is i_R / Z_R

# Iterate over benefit ratios and plot
for benefit_idx, benefit_ratio in enumerate(benefit_ratios):
    for i, fraction_C_poor in enumerate(fractions_C_poor):
        label = f"Fraction of $C_p$: {int(fraction_C_poor*100)}%,␣
 ↪{benefit_ratio}"
        plt.plot(x_rich, results_gos_rich[benefit_ratio][i], color=colors[i],␣
 ↪linestyle=linestyles[benefit_idx], label=label)

# Add labels and legend
plt.axhline(0, color='black', linestyle='--', linewidth=0.8)
plt.xlabel("$i_R / Z_R$")
plt.ylabel("Gradient of Selection ( )")
plt.title("Evolution of Rich Subpopulation, $Z_R = Z_P$")
plt.legend()
plt.show()
```

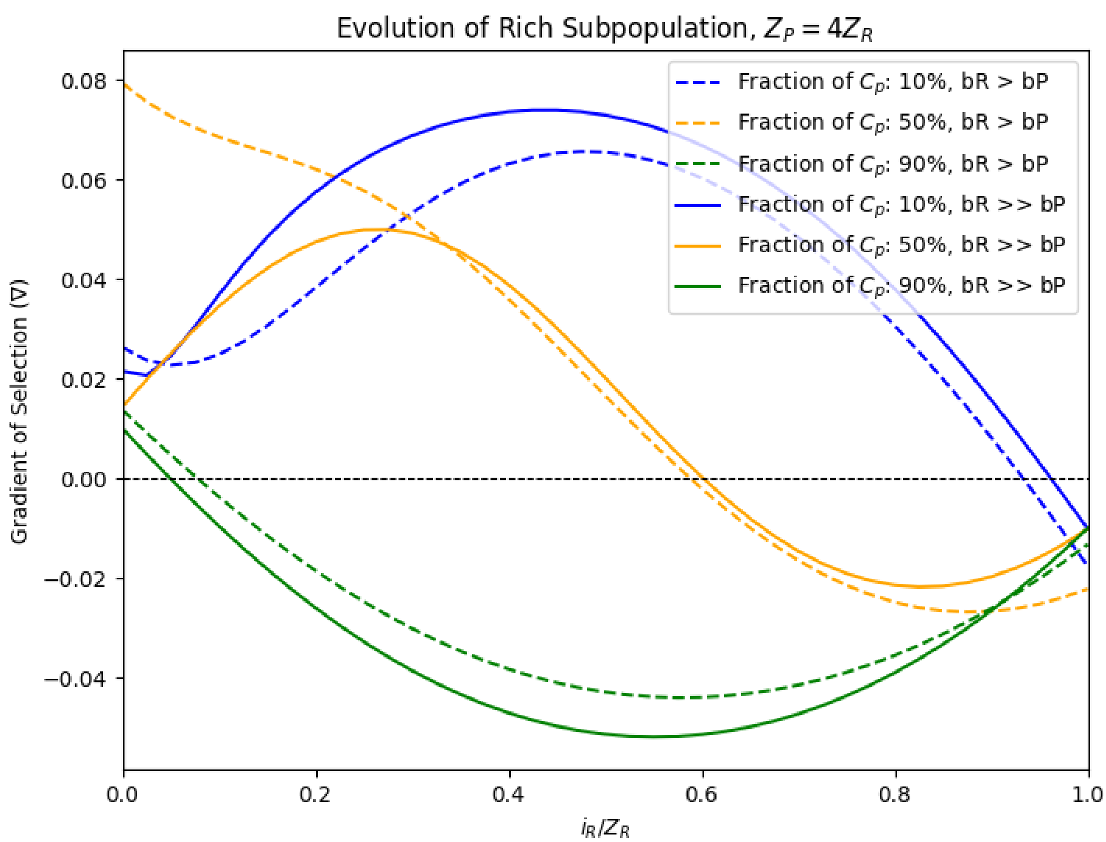


[8]:

```
# Testing some additional params (high risk, 0.9) with 4 * Z_R = Z_P

# Population sizes
rich_fraction = 0.2
Z_R = rich_fraction * population_size
Z_P = population_size - Z_R

# Parameters for the game
fractions_C_poor = [0.1, 0.5, 0.9]  # Fixed fractions of poor cooperators
benefit_ratios = ["bR > bP", "bR >> bP"]  # Different benefit scenarios

# Colors and linestyles for plotting
colors = ["blue", "orange", "green"]
linestyles = ["--", "-"]

# Initialize a dictionary to store results
results_gos_rich = {}

# Iterate over different benefit ratios
for benefit_ratio in benefit_ratios:
```

```
    gos_rich_values = []

    # Iterate over different fixed fractions of poor cooperators
    for fraction_C_poor in fractions_C_poor:
        fixed_C_P = int(fraction_C_poor * Z_P)

        # Initialize the game with the specific configuration
        game = ClimateThresholdPublicGoodsFrozenGame(
            population_size=200,
            rich_fraction=0.2,
            endowment_rich=1.35 if benefit_ratio == "bR > bP" else 1.75,
            endowment_poor=0.9125 if benefit_ratio == "bR > bP" else 0.8125,
            group_size=10,
            threshold=3,
            risk=0.9,
            contribution_factor=0.1,
            beta=10,
            mu=0.01,
            verbose=False,
            fixed_C_P=fixed_C_P  # Fix the poor cooperators
        )

        # Compute the gradients
        U, _ = game.compute_gradient_of_selection()

        # Extract GoS for the rich subpopulation for all i_R values
        gos_rich = [U[i_R, fixed_C_P] for i_R in range(int(Z_R) + 1)]
        gos_rich_values.append(gos_rich)

    # Store the results
    results_gos_rich[benefit_ratio] = gos_rich_values

# Plotting the results
plt.figure(figsize=(8, 6))
plt.xlim(0, 1)

x_rich = np.linspace(0, 1, int(Z_R) + 1)  # x-axis is i_R / Z_R

# Iterate over benefit ratios and plot
for benefit_idx, benefit_ratio in enumerate(benefit_ratios):
    for i, fraction_C_poor in enumerate(fractions_C_poor):
        label = f"Fraction of $C_p$: {int(fraction_C_poor*100)}%,␣
 ↪{benefit_ratio}"
        plt.plot(x_rich, results_gos_rich[benefit_ratio][i], color=colors[i],␣
 ↪linestyle=linestyles[benefit_idx], label=label)

# Add labels and legend
```

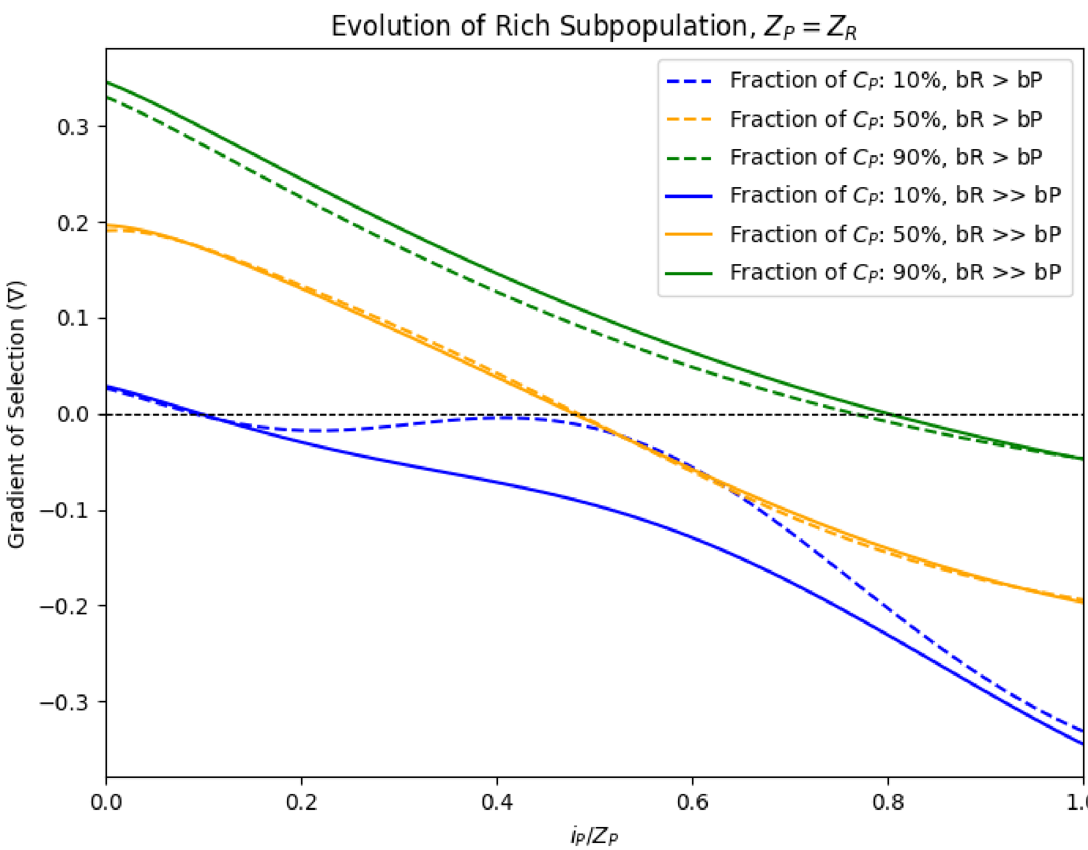
```
plt.axhline(0, color='black', linestyle='--', linewidth=0.8)
plt.xlabel("$i_R / Z_R$")
plt.ylabel("Gradient of Selection ( )")
plt.title("Evolution of Rich Subpopulation, $Z_P = 4Z_R$")
plt.legend()
plt.show()
```



[9]:
```
# Testing some additional params (high risk, 0.9) with Z_R = Z_P

# Population sizes
rich_fraction = 0.5
Z_R = rich_fraction * population_size
Z_P = population_size - Z_R

# Parameters for the game
fractions_C_rich = [0.1, 0.5, 0.9]  # Fixed fractions of poor cooperators
benefit_ratios = ["bR > bP", "bR >> bP"]  # Different benefit scenarios

# Colors and linestyles for plotting
colors = ["blue", "orange", "green"]
```

```
linestyles = ["--", "-"]

# Initialize a dictionary to store results
results_gos_poor = {}

# Iterate over different benefit ratios
for benefit_ratio in benefit_ratios:
    gos_poor_values = []

    # Iterate over different fixed fractions of poor cooperators
    for fraction_C_rich in fractions_C_rich:
        fixed_C_R = int(fraction_C_rich * population_size * rich_fraction)

        # Initialize the game with the specific configuration
        game = ClimateThresholdPublicGoodsFrozenGame(
            population_size=200,
            rich_fraction=0.5,
            endowment_rich=1.35 if benefit_ratio == "bR > bP" else 1.75,
            endowment_poor=0.9125 if benefit_ratio == "bR > bP" else 0.8125,
            group_size=10,
            threshold=3,
            risk=0.9,
            contribution_factor=0.1,
            beta=8,
            mu=0.01,
            verbose=False,
            fixed_C_R=fixed_C_R,  # Fix the rich cooperators
            fixed_C_P=None
        )

        # Compute the gradients
        _, V = game.compute_gradient_of_selection()

        # Extract GoS for the rich subpopulation for all i_R values
        gos_poor = [V[fixed_C_R, i_P] for i_P in range(game.Z_P + 1)]
        gos_poor_values.append(gos_poor)

    # Store the results
    results_gos_poor[benefit_ratio] = gos_poor_values

# Plotting the results
plt.figure(figsize=(8, 6))
plt.xlim(0, 1)

x_poor = np.linspace(0, 1, game.Z_P + 1)  # x-axis is i_P / Z_P

# Iterate over benefit ratios and plot
```

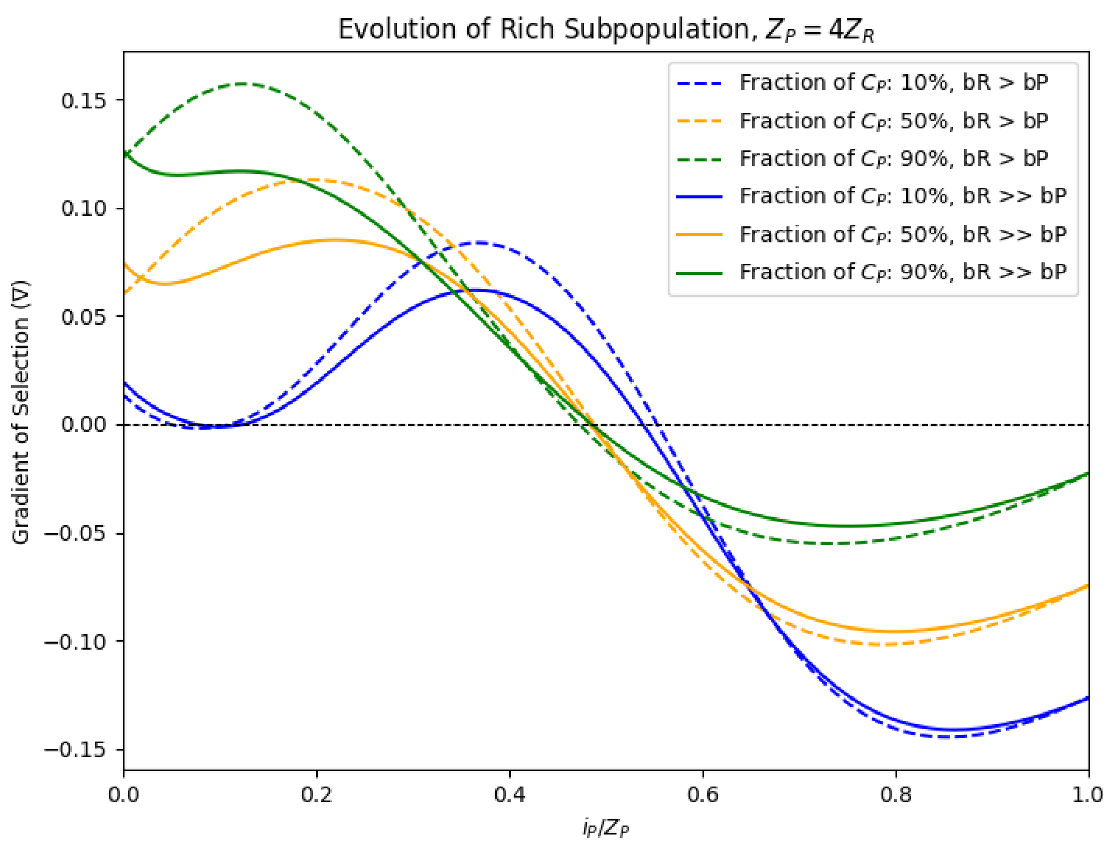
```
for benefit_idx, benefit_ratio in enumerate(benefit_ratios):
    for i, fraction_C_rich in enumerate(fractions_C_rich):
        label = f"Fraction of $C_P$: {int(fraction_C_rich*100)}%,␣
↪{benefit_ratio}"
        plt.plot(x_poor, results_gos_poor[benefit_ratio][i], color=colors[i],␣
↪linestyle=linestyles[benefit_idx], label=label)

# Add labels and legend
plt.axhline(0, color='black', linestyle='--', linewidth=0.8)
plt.xlabel("$i_P / Z_P$")
plt.ylabel("Gradient of Selection ( )")
plt.title("Evolution of Rich Subpopulation, $Z_P = Z_R$")
plt.legend()
plt.show()
```



[11]:
```
# Testing some additional params (high risk, 0.9) with 4 * Z_R = Z_P

# Population sizes
rich_fraction = 0.2
```

```
Z_R = rich_fraction * population_size
Z_P = population_size - Z_R

# Parameters for the game
fractions_C_rich = [0.1, 0.5, 0.9]  # Fixed fractions of poor cooperators
benefit_ratios = ["bR > bP", "bR >> bP"]  # Different benefit scenarios

# Colors and linestyles for plotting
colors = ["blue", "orange", "green"]
linestyles = ["--", "-"]

# Initialize a dictionary to store results
results_gos_poor = {}

# Iterate over different benefit ratios
for benefit_ratio in benefit_ratios:
    gos_poor_values = []

    # Iterate over different fixed fractions of poor cooperators
    for fraction_C_rich in fractions_C_rich:
        fixed_C_R = int(fraction_C_rich * game.Z_R)

        # Initialize the game with the specific configuration
        game = ClimateThresholdPublicGoodsFrozenGame(
            population_size=200,
            rich_fraction=0.2,
            endowment_rich=1.35 if benefit_ratio == "bR > bP" else 1.75,
            endowment_poor=0.9125 if benefit_ratio == "bR > bP" else 0.8125,
            group_size=10,
            threshold=3,
            risk=0.9,
            contribution_factor=0.1,
            beta=10,
            mu=0.01,
            verbose=False,
            fixed_C_R=fixed_C_R  # Fix the poor cooperators
        )

        # Compute the gradients
        _, V = game.compute_gradient_of_selection()

        # Extract GoS for the rich subpopulation for all i_R values
        gos_poor = [V[fixed_C_R, i_P] for i_P in range(game.Z_P + 1)]
        gos_poor_values.append(gos_poor)

    # Store the results
    results_gos_poor[benefit_ratio] = gos_poor_values
```

```python
# Plotting the results
plt.figure(figsize=(8, 6))
plt.xlim(0, 1)

x_poor = np.linspace(0, 1, game.Z_P + 1)  # x-axis is i_P / Z_P

# Iterate over benefit ratios and plot
for benefit_idx, benefit_ratio in enumerate(benefit_ratios):
    for i, fraction_C_rich in enumerate(fractions_C_rich):
        label = f"Fraction of $C_P$: {int(fraction_C_rich*100)}%,␣
↪{benefit_ratio}"
        plt.plot(x_poor, results_gos_poor[benefit_ratio][i], color=colors[i],␣
↪linestyle=linestyles[benefit_idx], label=label)

# Add labels and legend
plt.axhline(0, color='black', linestyle='--', linewidth=0.8)
plt.xlabel("$i_P / Z_P$")
plt.ylabel("Gradient of Selection ( )")
plt.title("Evolution of Rich Subpopulation, $Z_P = 4Z_R$")
plt.legend()
plt.show()
```



[ ]:

```
```