修 士 論 文

高齢者に代表される弱者を包摂するコミュニティ形成のマルチエージェントシミュレーション

1610415　張　玥（ZHANG　YUE）

主指導教員	橋本 敬
審査委員主査	橋本 敬
審査委員	池田 満
	内平 直志
	白肌 邦生

北陸先端科学技術大学院大学
先端科学技術研究科［知識科学］

平成３１年２月

Multi-Agent Model for Formation of Inclusive Community through Social Intercourse for the Weak represented by Elderly

ZHANG Yue

School of Knowledge Science,
Japan Advanced Institute of Science and Technology
March 2019

Keywords: Inclusive community, indirect reciprocity, the weak, multi-agent model

Due to the increase of single elderly households, living difficulties and lonely death due to isolation is becoming social issues. The purpose of this research is to clarify the condition for holding up the local community which is inclusive of elderly people who are at high risk of social exclusion. We used agent-based simulation to find out the conditions for maintaining a stable social inclusion.

We believe that the cooperative relationships are formed through social intercourse in the local community, and such relationships should be the basis for creating social inclusion.

Therefore, we modeled a community that carries out two different kinds of cooperative games: an interactive game based on direct reciprocity and an inclusive game based on indirect reciprocity. The results showed that the conditions for realizing social inclusion are: a） the number of interactive games exceeds the number of inclusive games, and b） social learning progresses gradually.

Therefore, we suggested that the establishment of cooperative relations based on direct reciprocity serves as the basis for creating social inclusion.

Instead of thinking in the direct reciprocal way to get the benefits matched the cost, it can be said that it is effective to think in the indirect reciprocal way as “someone may see my act of helping, not only the person whom I helped, so I might be helped someday” and to recommend taking a “socially (collectivistic)” idea to maintain a community that becomes the basis for regularizing the relationship that provides such indirect cooperation.

For those who only moved by directly reciprocity, it is important to continue telling stories that have good results from indirect reciprocity. To tell a story that indirect reciprocity has good results may also be related to slowly learning as in education, rather than instant imitation that jumps to immediate profit. It is difficult to change quickly, but in the end, the imitation probability of everyone may be lowered, leading to the increase in cooperation in the inclusive community.

目　次

図　目　次

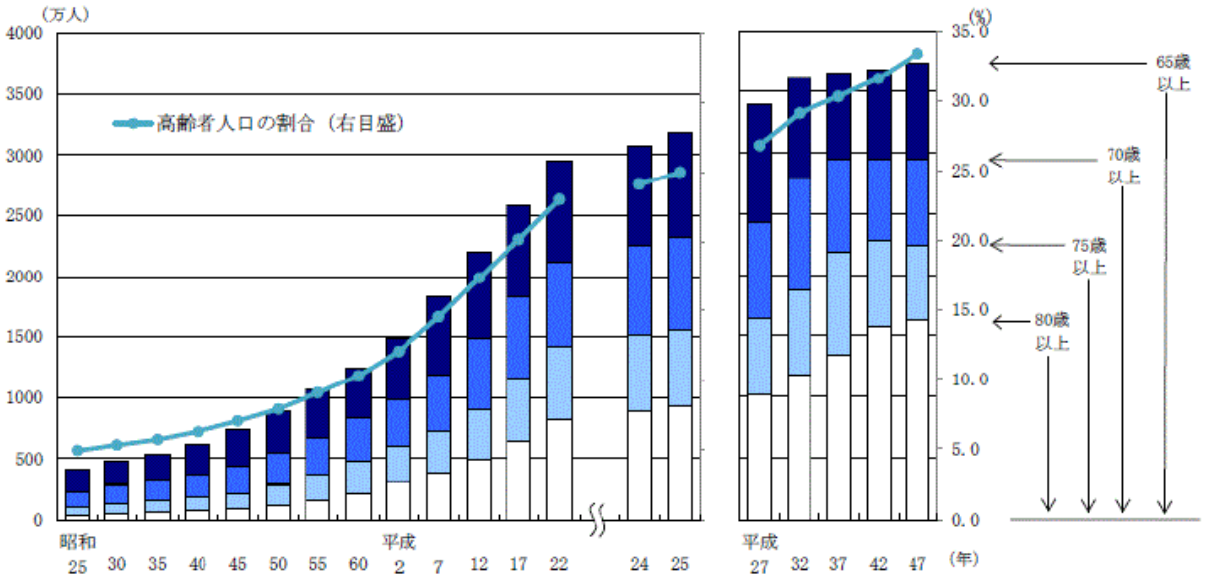
第１章　序論

1.1　はじめに

本研究では，シミュレーションを用いて高齢者に代表される弱者を包摂するコミュニティの形成のメカニズムを検討する．本章では，弱者の包摂において重要な概念である間接互恵，が先行研究でどのように扱われてきたのかについて述べる．その上で，本研究がどのようなモチベーションで包摂的なコミュニティを取り上げ，地域のコミュニティがいかにして包摂的なコミュニティになれるのかを目的として論じる．（←間接互恵と地域のコミュニティがどういう関係にあるのか（ありそうなのか）を記述するべきでは？）

1.2　背景

近年，日本では高齢化社会が進み，65 歳以上の（以下，「高齢者」と呼ぶ）人口は 3186 万人で過去最多になった．総人口に占める割合は 25.0％で過去最高となり，4 人に 1 人が高齢者という状況になっている．（総務省統計局 2013）



資料：昭和25年～平成22年は「国勢調査」、平成24年及び25年は「人口推計」
平成27年以降は「日本の将来推計人口（平成24年１月推計）」出生（中位）死亡（中位）推計（国立社会保障・人口問題研究所）から作成
注）平成24年及び25年は９月15日現在、その他の年は10月１日現在

図　1.1 高齢者人口及び割合の推移

高齢化率の上昇により高齢者を支える人が不足しており（内閣府，2016），高齢者の中でも元気な人が弱い高齢者を支えることが期待されている．しかし，他人を助けることができない弱者を支えるということは，見返りを期待せずに弱者を助けるということである．本研究では，弱い高齢者のように身体能力が低く，他の人に対して体力を使った手助けをしたくてもできない人を弱者と定義する．

多くの協力関係は，相手を助けたら相手からもいずれ助けてもらえるという相互の見返りを期待する．しかし，他者を助けられない弱者を助けるということは，その人からの助けが返って来ることは期待できない．そのような人を助けようと思わないのであれば，他者を助けられない弱者が排除されるような望ましくない社会になってしまうだろう．しかし，実際には，高齢や貧困など様々な困難を持つ弱者を支援し包摂できる社会の構築が求められている（厚生労働省，2000）　．

弱者を包摂する，すなわち，弱者からの直接の見返りを期待せずに相手に協力するためには，間接互恵が有効である．Nowak and Sigmund（1998a）は評判の概念を導

入することで互恵性理論を発展させ，間接互恵が達成できることを示した．彼らのモデルでは，イメージスコアという評判を示す値によってグループ内の全員が継続的に評価され，再評価される仕組みを導入した．そして「相手に協力行動をすれば，周囲からの自身に対する評価が上がり，相手以外の他の誰かから協力行動を返してもらえる（＝返報)」という間接的互恵関係の仕組みによって，協力行動が進化的に安定になることをシミュレーションによって説明した．

間接互恵が成立するNowak and Sigmund（1998a）の研究では協力をしないフリーライダーは排除される．一方，我々が目指したい社会は協力したくてもできない人，行動としてはフリーライダーにならざるを得ない人を助けられる包摂的な社会である．そのような社会をいかにして作るかということに関しては，Nowak and Sigmund（1998a） の仕組みだけでは不十分であると考えられる．

見守りコミュニティや震災後の助け合いなどの見返りを期待しない協力は人間社会には存在し，それらの成立過程に関する事例研究がある．長谷川・小川（2015）は，サークルなどの活動から不特定多数の高齢者が交流するコミュニティができ，そのコミュニティ内での交流を通じて，生活問題解決のための見守りを行うコミュニティがスムーズに形成できたことを報告している．すなわち，見守りコミュニティのような包摂的なコミュニティはいきなりできるのではなく，普段からの交流が地域のコミュニティに根付いて，そこからいざという時の助け合いである見守りコミュニティへと段階的に作られる．

この事例より，交流のコミュニティを直接互恵的なコミュニティと考えると，そこでの人間関係が包摂的なコミュニティに伝播し，間接互恵的な協力社会が成り立つという仕組みが予想される．しかし，このような仕組みはNowak and Sigmund（1998a）のモデルには含まれておらず，間接互恵的な協力社会の成立メカニズムは不明である．直接互恵と間接互恵が両方存在するモデル（Gilbert，2008）では，会ったことがある相手に対して，自分の経験を信じて直接互恵的なゲームをするのか，あるいは噂を信じて間接互恵的なゲームをするのかを選ぶことが想定されている．しかし，この研究

でも自分に対しては協力してもらうが，相手に対しては協力しないフリーライダーを排除することが主題となっている．

本研究では，一つの地域コミュニティを対象に，2 種類の局面を経ることを想定している．それらは前述で予想した，交流のコミュニティと包摂的なコミュニティである．交流のコミュニティとは，コミュニティサロン・コミュニティカフェや町内会での会話などの負担が小さい交流を目的とし，相手に話しかけたら話してくれるという見返りがあるか，それが期待できるような特徴を持つ直接互恵で成り立つ組織である．

包摂的なコミュニティとは，交流よりも負担の大きい物理的・直接的な助けを一方的に与えることを目的とし，相手からの手助けという見返りを期待しないような特徴をもつ組織である．すなわち，他者を手助けできない弱者を助ける際は，助ける側は見返りを期待することができない．このようなコミュニティは間接互恵で成り立つと考えられる．

間接互恵に関しては Nowak and Sigmund（1998a）が導入している「評判」を取り入れ，つきあいを判断する基準もモデルに取り入れる．本研究では，つきあいを判断する基準を不寛容度と呼ぶ．間接互恵か直接互恵かの 2 種類のゲームがあり，エージェントが両方のゲームを順にプレーする．直接互恵ではフリーライダーを排除するべきで，間接互恵では他の人を助けられない弱者をフリーライダーとして扱わないが，他者を助けられるにも関わらず実際には助けないような人をフリーライダーとして扱い，排除するべきであろう．本研究では，直接互恵の方でうまく協力しているという情報が間接互恵の方にも噂としてうまく伝わり，直接互恵的なゲームでうまくいくということが，間接互恵のゲームで他者から助けてもらえることに伝搬（?）するモデルを構築した．

1.3　研究の目的

本研究は，他者を助けることができないような高齢者に代表される弱者が，フリーライダーとして排除されるのではなく助けられるような包摂的なコミュニティが作られる要因を示すことを目指している．そのために，交流のコミュニティから見守り

コミュニティができる事例（長谷川，小川，2015） にヒントを得て，この2種類の協力的なコミュニティが安定に成立する条件を，エージェント・ベース・シミュレーションを用いて調べた．すなわち，交流のコミュニティで協力関係が形成され，その関係が弱者を包摂するコミュニティに展開される制度設計のヒントをエージェントモデルで示すことを目的とした．

1.4　研究の方法

本論文では研究手法として，エージェント・ベース・モデル（Agent-Based Model，以下 ABM と略す）及びシミュレーションを用いる．

Sqazzoni（2010）（←参考文献にありません）は，ABM を以下の五つのタイプに分類している．人工社会（Artificial socisitcs），抽象モデル（Absutract models），ミドルレンジモデル（Middle-range models），ケースベースモデル（Case-based models），そして応用シミュレーション（Applied simulation）である．本研究で扱う ABM は，抽象モデルである．抽象モデルとは，社会現象についての理論モデルであり理論の構築と発展を目的とする，特定の経験的事例を反映するものではない（山田，2017）．本シミュレーションは，理論的な研究をベースに事例（長谷川，小川，2015）から得たヒントを組み合わせた．

さらに，ABM は人間の多様性を考慮でき，個々の主体間に見られる動的な関係性（相互作用）を取り扱うことができる（島，武藤，2002）という利点を持ち，近年では，社会現象の分析や理解のために，経済学や心理学などの社会科学分野で導入されている．

本論文では ABM を用い，交流のコミュニティと包摂的なコミュニティが安定して成立するための条件について洞察を得るため，シミュレーションを行った．

1.5　本論文の構成

第 1 章では，本論文の主題である弱者の包摂に関する議論を概観し，本論文の論点

と目的を述べた.

第 2 章では，高齢者包摂，コミュニティ，直接互恵や間接互恵などに関する先行研究を紹介し，本研究の位置付けを明らかにする.

第 3 章では，本研究で扱うモデルを詳細に説明する.

第 4 章では, シミュレーションの結果と考察を述べ, 制度設計のヒントを提示する.

第 5 章では，全体に対する考察を述べ，理論的の発展と現実的における意味を提示する.

第 6 章では，以上の研究をまとめ，本研究の結論を述べた上で，本論文で提起した問いに答えを提示し，今後の課題を述べる.

第２章 先行研究レビュー

この章では，本研究と関連する先行研究をいくつかの分野を用いて紹介する．まずは，直接互恵と間接互恵に関する理論的な研究を挙げ，本研究の理論的な位置付けを示す．そして，政策課題にもなっている社会的包摂の中の高齢者の包摂に注目する．さらに，交流のコミュニティという局面を想定しているサードプレイスやコミュニティサロンの現実の事例研究を紹介する．最後に，本研究の研究方法である社会現象の理解と社会システムの手段として使われている社会シミュレーションについて言及する．

2.1　直接互恵と間接互恵

人はなぜ利他的にふるまうのであろうか．利他行動とは外的報酬を期待せず他者のために自発的に行う行為自体が目的の行動と定義される（Bar-Tal, Sharabany, &Reviv,1982）．

利他的行動の動機の促進・抑制要因の解明が中心的な焦点とされてきた（明田，岡本，奥田，外山，山口,1994; 大坊，安藤，池田,1989）．しかし，なぜ人間がこのような一見非合理的な属性を持っているのかに対しては答えられていない．協力的な社会の形成は社会科学の大きな目的の一つといえるが，人々が自己の損失を顧みずに見返りが期待できない協力を行うことは単純には説明しがたい．人々が自らの目先の利益を捨て，他人や公共のために自己犠牲を行うことを説明する理論が，これまでいくつか提案されてきた．協力的な社会的交換を促進するとされる社会関係資本は，社会的ネットワーク，信頼，互酬性の規範から構成されると言われている．（Putnam，1993）

生物学的にコストになる協力行動の進化を解明する Nowak（2006）の理論的研究では協力の進化のための五つのメカニズムを明らかにした．それらは，血縁淘汰（Kin selection），直接互恵性（direct reciprocity），間接互恵性（indirect reciprocity），ネットワーク互恵性（network reciprocity），集団淘汰（group selection）の五つに分類

される.

Nowak and Sigmund（1998ab）は，世間の評判などを抽象化したイメージスコアと呼ばれる概念を導入することにより，再び出会うことがない相手への一方的な協力行動というギビングゲームにおいて，説明が不十分だった繰り返し相手と出会わないような状況や，血縁的な結びつきがない状況での協力行動が成立する可能性を示した.

ギビングゲームでは，一定の人数（100 名程度）の集団内において，協力の渡し手と受け手がランダムに選ばれてペアリングされ，協力の渡し手が一方的な協力をするかどうかの意思決定を行う．協力行動にはコストがかかり，協力すると決めれば協力の渡し手が不利益になり，協力の受け手の利益が増える．しかし，協力の渡し手の世間の評判を表すイメージスコアが上昇することになり，協力しないことを選んだ場合はイメージスコアが下がる．Nowak and Sigmund（1998ab）は，ギビングゲームにおいて，協力の渡し手が自分の他者を判断する基準を持ち，毎回相手のイメージスコアと自分の判断基準を比べて，相手に協力するかどうかを決める．このようにイメージスコアを参照することによって，協力的な個人が集団の中でより助けてもらえやすくなり，それを他の人々が模倣することで協力的な集団が発生する可能性があることをシミュレーション及び数値解析により示した.

上記のギビングゲーム的な状況を，より理論的に解釈した協力の進化の研究（松井ら 2009）では，直接の見返りがない状況において，個人の協力的な行動により，結果として集団内で相互に協力しあっている状態に至るタイプの協力を，間接互恵性と呼んだ.

間接互恵で一定のコストを支払い，他者に利益をもたらす行動を向社会的行動と呼ぶ. 集団で生活を営む人間の社会において，向社会的行動は重要な役割を担い（松井，鳥居，村上，寺野，2009），多くの領域でその進化的な説明原理が検討されてきた（Rand and Nowak, 2013）．その中でも，直接互恵行為と間接互恵行為の重要さが特に強調されている．直接互恵行為とは，向社会的行動の送り手と，その行為の見返りの対象が同じ二者間での互恵性を指す（Trivers, 1971）．つまり，協力したら協力してもらえる，裏切りをしたら裏切り返されるか，またはその可能性があるということである．直接互恵行為が成立する場面では，相手からの将来的な返報が予測される場合，向社会的行動は適応的となりうる．直接互恵に対して，間接互恵行為とは，行動の出し手と，その返報の対象が異なる三者関係での互恵性を指す（真島，高橋 2005b）.

間接互恵性の概念は"情けは人の為ならず"という諺で端的に表現することができる．すなわち，助けた相手から協力が直接に来なくても，回りに回って別の相手が自分を助けてくれるという仕組みを間接互恵性と呼ぶ．

Leimar and Hammerstein（2001）と Panchanathan and Boyd（2003）は，Nowak and Sigmund（1998a，b）のイメージスコアによる結論がごく限られた条件でしか成立しないと指摘し，間接互恵性の成立を可能にする新たなスタンディング戦略（standing 戦略）という解決策を提唱した（Sugden,1986）．スタンディング戦略とは「評判の悪い相手に対する非提供」を悪い行為とみなさない戦略である．一方，本研究では協力できない弱者が評判を上げられないので，弱者を助けないことを悪い行為とみなさない社会は望ましくないと考える．

一方，真島・高橋（2005b）は評判の良い相手への資源提供のみを「いい行為」とみなし，評判の悪い相手への提供とすべての非提供を「悪い行為」とみなす厳密識別（strict discriminator：以下 SD）のみが間接互恵性を可能にすると述べた．SD 規範が イメージスコアと異なるのは「評判の悪い相手への提供」を「悪い行為」とみなす点であり，これに関しては非寛容な規範であるといえる（松井ら，2009）．さらに，真島・高橋（2005a）は，SD 規範だけでなく，「評判の良い相手への資源提供と， 評判の悪い相手への非提供を「いい行為」とみなし，評判の良い相手への非提供と，評判の悪い相手への提供を「悪い行為」とみなす」（extra-standing）によっても間接互恵性の成立が可能であると指摘している．

Nowak and Sigmund（1998）は間接的互恵関係という仕組みによって，協力行動の進化的安定をシミュレーションによって説明した．しかし，世代交代による進化ではなく，人間関係の変化を捉える際，一世代での創発を考えるべきである（岡崎，赤井，西野，2014）という議論もある．

間接互恵はなぜ「評判」を使って協力するかに対して答えを上げたが，間接互恵で想定しているのは助ける者と助けられる者が二度は会わないような状況である．しかし，人間も動物もグループの中では繰り返し会うことが一般的で，間接互恵の前提は現実の状況にそぐわないところがある．Gilbert（2008）は「個体がいつ出会っても，協力するかどうかを選択するときに直接互恵を選ぶか，または間接互恵を選ぶかを決定する」と論じている．

2.2　高齢者の包摂

本研究では高齢者をどうやって包摂するのかを研究するわけだが，なぜ高齢者包摂が必要なのかについて先行研究を紹介する．

単独高齢者世帯の増加によって，孤立による生活困難や孤独死が社会的課題となっている．そのような中，高齢者の近隣高齢者との日常的な相互行為（気遣い合いの日常交流）は，加齢による変化という現状を日常的に共有することを通して，高齢者が自ら積極的に今を生きようとすることを助けている(大森, 2007)という言及もある．

福嶋（2014）は，本研究で想定していない交流のコミュニティの形成について調べた．高齢者が体操に関する自主グループ設立に至るまでの，段階的な気持ちや認識の変化とその変化に関連する要因を明らかにするために，参加者に電話調査を行い，調査を通じて体操に関するグループ活動の不参加の理由を調べた．高齢者の地域コミュニティへの参加の促進，地域の健康維持に関する講座の開催などを通じて，体操に関する自主グループの設立準備への支援を進めることで，高齢者の自主グループ設立が効果的に行われることができると考えられている．

しかし，人口の流出と自然減により過疎化が進んでしてしまう一方，高齢者の定住意欲はきわめて高い（金子,2009）．そのため，高齢者が安心して定住地で生活できる継続的な仕組み作りが求められている（岩原,2010）．「身近な地域において誰もが安心して生活を維持できるよう，地域住民相互の支え合いによる共助の取り組みを通じて，高齢者を含め，支援が必要な人を地域全体で支える基盤を構築する必要がある」（内閣府,2016）．

様々な問題や困難を抱えた人々が社会から排除されている状況を踏まえて，特別なニーズ を持っている人たちへ必要な支援を行い，社会参加を促進して社会へ包括しようとする社会的包摂は政策課題でもある（上野，金城,，植村，畑下，2010）．

2.3　サードプレイスとコミュニティサロン

本研究で想定している交流のコミュニティはサードプレイスやコミュニティサロンなどであるため，それらに関する研究に言及する．

小林，山田（2014）はコミュニティサロンやコミュニティカフェなどの形成についてそのメカニズムを調べた．その結果，利用者の地域への愛着を高め，利用者の地域関与を生み出すには交流型とマイプレイス型の両機能が実現するサードプレイスが有用であると述べられている．サードプレイスは，自宅（ファーストプレイス）や 職場・学校（セカンドプレイス）とは異なる場所として定義される．サードプレイスは地域の交流拠点としての機能を有する．しかし，小林らの研究においては，コミュニティカフェが非常設型カフェであり，カフェの継続的実施がなされていない．そのため，サードプレイスなどの場所は，いざという時に協力が生まれる基盤となりうるが，サードプレイスで築いた関係性がより大きな協力に発展していけるかどうかはまだ考えられていない．

コミュニティカフェやコミュニティサロンなどの場所では，利用者が交流を通じて関係性を築きやすいが，利用者の固定化や，多様性に乏しい排他的なコミュニティになりやすい問題が残っている（大分大学福祉科学研究センター，2011）．

身体的に自立できる立場でありながら，閉じこもりがちな高齢者への対応は，地域福祉の課題である．中村（2009）はこの課題に対して，孤立しがちな高齢者の地域における居場所として，あるいは地域のあり方に関心のある住民の活動拠点としてのサロンに注目し，地域コミュニティとしてサロンを評価することで，持続性あるサロンのあり方を考えた．その結果，非常時にサロンの主催者や仲間を頼りたいとする高齢者は多く，サロンに参加できない高齢者も，その存在がサロン関係者の間で情報として共有され，日常の見守りと非常時の対応のための情報ネットワークに組み込まれていることを示している．情報拠点としてのサロンの存在価値は大きく，「心のよりどころ」でもある．しかし，このようなサロンが必ず地域に存在するではないので，コミュニティサロンの形成を考えるべきである．

サードプレイスを始め，高齢者の居住地で生活を支援する地域拠点としてのコミュニティカフェやコミュニティサロンなどに関する研究は，大阪府の 「街かどデイハウス」がサロン活動を通じて介護予防事業を支援する事例があり（今井ら 2005），在宅高齢者の自立生活を支援する住宅改善に関して，「街かどデイハウス」 という施設の利用者の属性や生活状況について分析した研究もある（田中ら 2006）．ただし，「街かどデイハウス」が非常時の対応というコミュニティサロンが担う重要な役割を負えているのかどうか，仮にできているとしたらいかにして成立しているのかについては

分析されておらず，検討する必要がある．

2.4　社会シミュレーション

社会，経済や文化など，従来実験が困難であった問題，あるいは人間の意志決定が中心となる問題に対しての接近法として計算機を用いた研究が行われている．このようなマルチエージェントシミュレーションは，社会シミュレーション（Gilbert and Troitzsch，1999）と総称され，社会現象や社会システムの理解ための手段として使われている（石田ら，2007）．

エージェントを用いたシミュレーションには，二つの目的がある．第一は複雑な社会現象を解明するためのものである．エージェントは可能な限り単純化してモデル化され，エージェント相互のインタラクションによって複雑な現象が生まれることを観察する．第二の目的は，複雑な人間の挙動に関して，現実に近い形での再現を目指していることである．エージェントは現実に近い形でモデル化され，利用者に疑似体験を与える（石田ら，2007）．本研究は，エージェントの相互作用によって生まれた結果の分析を通じて，複雑な社会現象を一旦抽象化しモデル化することで，現実に返り，制度設計のヒントを与えることを目指している．

実験を伴うものであれば，それを再現できることが要請される．実験の再現という困難を克服する手段としてのシミュレーションの役割が注目されている（寺野 2010）．

社会シミュレーションの分析の目標は，必ずしも未来の社会現象を「予測」することではなく，「可能性」 を可視化することである（高橋 2013）． 本研究でも同様に，予測ではなく，制度設計のヒントの提示という可能性を示すために社会シミュレーションを行う．

第３章 モデル

本章では，本研究で用いるモデルを解説する．まず，モデル化の対象としている人とコミュニティを説明し，それらを抽象化したモデルの概要について述べる．そして，交流のコミュニティと包摂的なコミュニティについて解説する．そして，評判や不寛容度といった概念がどのようにしてシミュレーションに組み込まれているかについて述べる．

3.1　モデル化の対象

本モデルでは，2 種類の人を想定する．一つは他者と交流や手助けなどの協力ができる一般人，もう一つは他者と交流はできるが，他人に対して体力を使う手助けなどの協力ができない高齢者に代表される弱者である．

公民館やコミュニティカフェなどの場所において，人が集まる機会があり，皆でお茶を飲むなどの毎週または毎日行われているイベント等を，本研究では交流のコミュニティとして扱う．

包摂的なコミュニティでは，助けが必要な相手のところに行き，負担が大きい物理的・直接的な見返りの無い手助けを行うことを目的とする．例えば，雪国において，生活の問題として発生する雪かきなどの手助けが考えられる．

3.2　モデルの概要

本研究では，交流のコミュニティと包摂的なコミュニティという 2 種類の局面を考える．交流のコミュニティは，たとえばコミュニティカフェや町内会などのところで

の会話などの負担が小さい交流を目的とし，互いに楽しく話し合うような見返りが期待できる特徴を持つ．したがって，こちらを直接互恵によりモデル化する（「交流ゲーム」と称する）．包摂的なコミュニティとは，助ける相手のところに行き，交流のコミュニティと比べて負担が大きい物理的・直接的な助けを一方的にしてあげることを目的とし，相手に手助けをしてあげても，相手が弱者であれば手助けを期待できない，相手が弱者ではなくても手助けの見返りを期待しないような特徴をもつコミュニティとする．したがって，こちらは間接互恵のゲームとしてモデル化する（「包摂ゲーム」と名付ける）．そして，交流ゲームの関係が後者の包摂ゲームの関係に影響を与えるとする．

これらの二つのコミュニティにおけるエージェント間の相互作用を，Nowak and Sigmunt（1998a），および Gilbert（2008）の 2 つの研究をベースに，高齢者に代表される弱者とそうでない人を区別するようにエージェントをモデリングした．

シミュレーションでは，「不寛容度k」，「経験スコアe」と「イメージスコアi」を持つ主体（エージェント）が，直接互恵による「交流ゲーム」と，間接互恵で成り立つコミュニティを表す「包摂ゲーム」で相互作用する．

交流ゲームでは，相手に対する経験スコアeと自身の判断基準kの差および判断基準kに応じた参加コストβにより，利得pと経験スコアeが変わる．包摂ゲームでは主体（エージェント）はランダムに協力の提供者か受容者となり，提供者は受容者に返報を求めない協力をするか否かを，判断基準とイメージスコアから決定する．協力すると利得が下がるがイメージスコアが上がる．何回かのゲームの後，各主体（エージェント）は他の主体（エージェント）の判断基準を確率的に模倣する．

3.3　エージェント

Nowak and Sigmunt（1998a）と同様に各エージェントは，他者とのつきあいを判断する基準，および経験に応じた他エージェントへの評価値を持つ．本研究では「不寛容度k」「経験スコアe」でそれぞれ表す．不寛容度k（$1 \leqq k \leqq 11$の整数を取る）は各エージェントが他者と交流・協力するかどうかを決める基準であり，この値が小さいほど寛容的で交流も協力もする傾向にある．経験スコアe（こちらも$1 \leqq e \leqq 11$実数）

は，各エージェントが持つ他のそれぞれのエージェントに対する評価であり，高いほど評価が高い．また，各エージェントは全エージェントが持つ自身に対する経験スコアから構成される「イメージスコア i」によっても評価される．これが評判である．この Nowak and Sigmunt（1998a）と共通する 3 つの要素に加えて，本シミュレーションにおけるエージェントは弱者かそうでないかの違いを持つ．この性質の違いはシミュレーションを通じて変化しない．

エージェントは直接互恵による「交流ゲーム」と間接互恵による「包摂ゲーム」で相互作用し，交流ゲームでは不寛容度kと経験スコアe，包摂ゲームにおいては不寛容度kとイメージスコアiをもとに意志決定する．これらがどのように用いられるか，また変化するかは，次に示す各ゲームの中で説明する．

3.4　不寛容度と参加確率

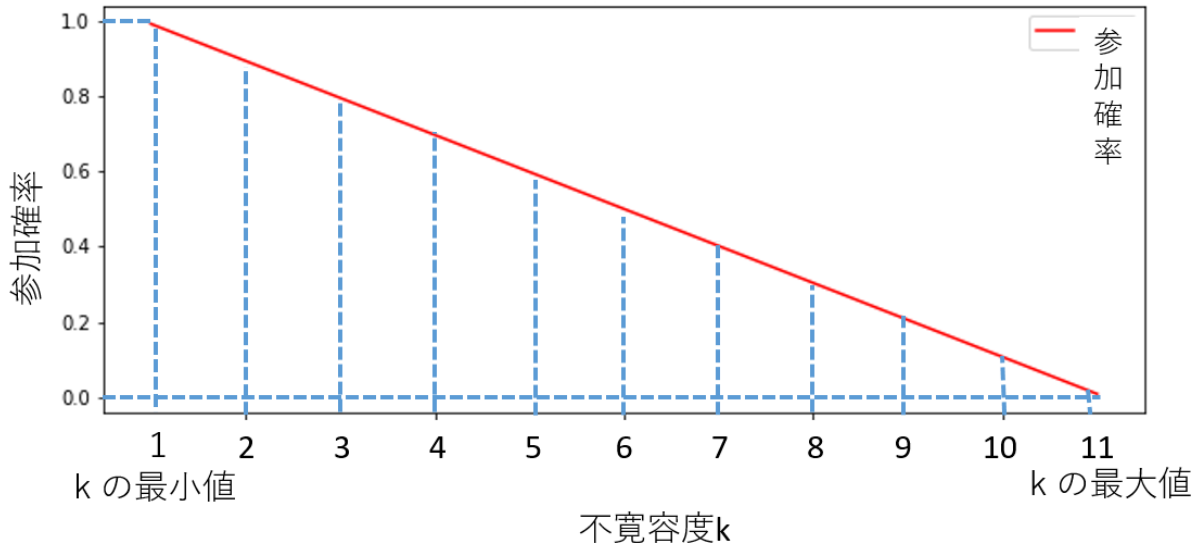


図　3.1　参加確率

公民館や町内会活動などの場所でイベントがあると知った場合に，参加者は参加するかどうかを選択できるが，行くにはコストがかかるとする．そこに行くと誰かがいて，誰かと話ができる（交流できる）．もし交流が楽しかった場合は，自分の利益になり，もし交流が楽しくない場合は自分の損となる．すなわち，参加コストと利得の差

が利益や損になる．交流のコミュニティでは，弱者でもできる程度の交流がなされると想定しているため，一般人と弱者を区別しない．

一般人と弱者，この 2 種類の人が交流のコミュニティに行ったとしても，必ず交流がうまくいくとは限らない．これを判断する基準として，他者とのつきあいを判断する基準を不寛容度k，または非フレンドリーさと呼ぶ．不寛容度kは，エージェントに固有であり，ある人物がそのような性格であることを表す．不寛容度kが大きいほど，不寛容的で交流も協力もしない人物であり，不寛容度kが小さいほど寛容的で交流も協力もする人である．不寛容度kは個人で固有なものであるため，誰とでも協力する人もいれば，誰とでも協力しない人もいる．そういう人物は，相手の経験スコアeが高くても交流がうまくいかなくなる．

また，交流のコミュニティに行くか行かないかは確率となっており（図 3.1），それもそのエージェントの不寛容度kによって違う．不寛容度kが大きいほど交流のコミュニティに行く確率が低くなり，不寛容度kが小さいほど交流のコミュニティに行く確率が高くなる．

3.5　経験スコアとイメージスコア

一般人も弱者も，他者への 2 種類の評価値を持つ，経験スコアeとそこから作られるイメージスコアiである．

経験スコアeは，あるエージェントaがエージェント b に対する経験スコアe_{ab}，エージェント c に対する経験スコアe_{ac}，エージェント d に対する経験スコアe_{ad}というように，個人がそれぞれに対して全員が持っている評価値である．自分（a）がエージェント b と交流がうまくいったら，エージェント b がいい人物であると思い，自分（a）が持つエージェント b の経験スコアe_{ab}がアップする．エージェント c との交流がうまくいかない場合は，エージェント c が良くない人物であると評価され，自分（a）が持つエージェント c の経験スコアe_{ac}が下がる．

イメージスコアiは，噂により個人の評判が広まることで，全員が共通して認識し，イメージスコアの値は人それぞれ異なり，全てのエージェントが共通で知りえる．イメージスコアiを正確に知りえる確率は，人それぞれ異なっており，交流ゲームに参加

すればするほど，正確に知る確率が高くなる．

世間的な評価と個人的な評価は一致しない，誰から見てもイメージスコアが一緒の場合，エージェントaのイメージは良く，エージェント b のイメージは良くないということはあり得る．また世間から見てエージェント b のイメージは良くないが，エージェントaからみるエージェント b はいい人ということはあり得る．

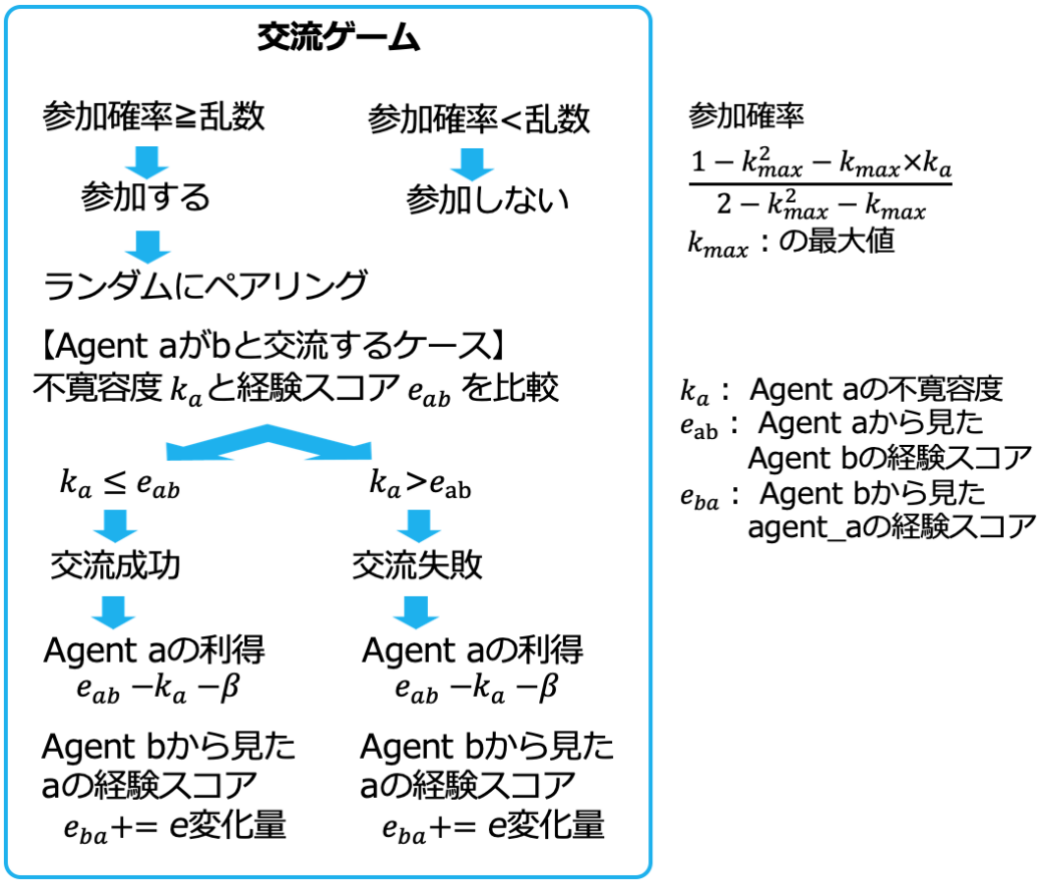


図　3.2　交流ゲーム

3.6　交流ゲーム

エージェントが交流ゲーム（図 3.2）に参加するかどうかは，不寛容度kに応じた確率で決まる．不寛容度kが大きいほど参加確率が低く，$k = 1$ではほとんど毎回参加し，$k = 11$ではほとんど毎回参加しない．参加するエージェントは，ランダムに他エージ

ェントとペアリングされる．

そして，自身の不寛容度kと自身が持つ出会った相手の経験スコアeを比較し，$k_{\mathrm{a}} \leq e_{\mathrm{ab}}$ のときに交流が成功だと感じる．ここで，aを自身，bを相手のエージェントとし，k_{a}はエージェントa自身の不寛容度，e_{ab}はエージェントa自身が持つ相手のエージェントbの経験スコアである．そして，交流がうまくいったら，相手（b）がいい人だと思いe_{ab}が上がる．逆の条件（$k_{\mathrm{a}} > e_{\mathrm{ab}}$）では交流がうまくいかなかったと感じ，相手の経験スコアを下げる．どちらの場合も，エージェントaは $e_{\mathrm{ab}} - k_{\mathrm{a}} - \beta$ の利得を得る．

$$p_{\mathrm{a}} = e_{\mathrm{ab}} - k_{\mathrm{a}} - \beta \qquad \text{（式３.1）}$$

3.7　経験スコアからイメージスコアへの変換

あるエージェント（ここではyとする）のイメージスコアi_y'は，出会ったエージェントが持つ自分に対する経験スコアの平均と現在のイメージスコアを合わせるように，（3.2）式で算出した．

$$i_y' = (1-\alpha) i_y + \alpha \frac{1}{M} \sum_x e_{xy} \qquad \text{（式３.2）}$$

- i_y'：今回の y のイメージスコア
- α：交流ゲームでの評価が包摂ゲームでの評判に変換される割合
- i_y：前回の y のイメージスコア
- M：y が会ったエージェントの数
- x：会ったエージェント
- y：エージェント自身
- e_{xy}：出会った他のエージェントxから見るyの経験スコア

ここでxの和はエージェントyが出会った相手全てに渡って取り，Mはその総数である．αは経験スコアがどの程度反映されるかを決めるパラメータである．本研究では，イメージスコアは評判として広まっていて，基本的には全員が共通して知り得るもの

とする．しかし，実際にその値を正確に知る確率は，提供者が交流ゲームへの参加程度に依存し，よく参加するほど正確に知る確率が高くなる．正確に知れない場合は，1～11 の中間の$i = 6$であるとみなす．

3.8　包摂ゲーム

間接互恵で成り立つコミュニティを表す「包摂ゲーム」（図 3.3）では，エージェントはランダムに協力の提供者か受容者となる．提供者が弱者でない場合は，受容者に返報を求めない協力をするか否かを判断する．

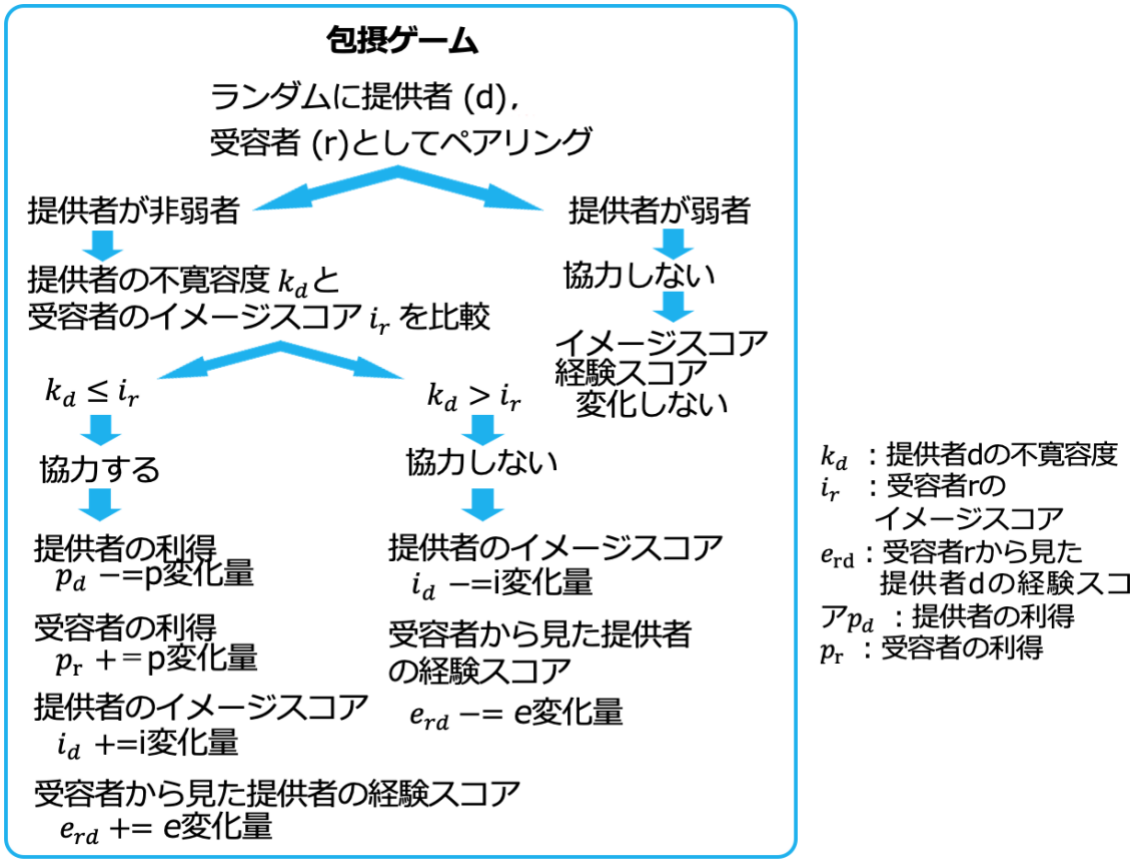


図　3.3　包摂ゲーム

自身（提供者 d）の不寛容度k_dと相手（受容者 r）のイメージスコアi_rを比べ，$k_d \leq i_r$ であれば協力する．協力により自身の利得がp_dが減り，イメージスコアが上がり，相手の利得がp_r増加する．また，相手が持つ自分に対する経験スコアe_{rd}も上がる．逆

の条件（$k_d > i_r$）のときは相手には協力せず，利得の変化はないが，自身のイメージスコア，相手が持つ自身の経験スコアe_{rd}がともに下がる．

一方，提供者が弱者の場合は，不寛容度，イメージスコアの値にかかわらず協力しない．しかし，イメージスコア，相手の持つ経験ともに下がらない．

3.9　利得

交流ゲームでは, 利得 = 経験スコア − 不寛容度 − 参加コスト β(式 3.1), つまり，参加から得られる利益−参加に払うコストである．

包摂ゲームでは，相手に協力をしたら，自分の利得が下がり，助けた相手の利得が上がる．これは，自分が相手に対して体力や時間を使うことで，相手がその便益を受けることを抽象化したものである．

3.10 不寛容度の模倣

交流ゲームと包摂ゲームが終わった後に，不寛容度の模倣を行う．

この種のモデルをシミュレーションで行う時に，ほとんどの人が最高利得のエージェントを模倣することになっている．しかし現実的に考えると，これは過度な模倣であると考え，もっと模倣を減らそうという意図で，モデルを変更した．会ったことがある人の中で一番利得が低い人の不寛容度が自分の不寛容度と同じなら，会ったことがある人の不寛容度をランダムに模倣して自分の不寛容度を変える．つまり，現在と前ステップで出会ったエージェントの中からランダムに模倣対象を選ぶ．結果として，会ったことがある人の中で一番利得が低い人の不寛容度が自分の不寛容度と同じではない場合，自分の不寛容度を変更しないというようなモデルとなり，これをモデル 1（即時模倣）と呼ぶ．

モデル 1 に対し，更に現実的な状況として，社会学習が起こらないようにモデルを変え，模倣したい人の不寛容度kにいきなり変化するのではなく，そのkに 1 だけ近づくようにした．これをモデル 2（漸近模倣）と呼ぶ．

モデル 1 では，世代交代だと不寛容度の高い人のコピーができるため，不寛容度の高い人はそのまま増える．一世代における学習では，何度も頻繁に不寛容度kが変わ

るわけではないため，モデル 2 では，世代交代的なものでない高齢者による学習はがあまり急激に学習しないようにモデリングしている．

つまり，つぎの 2 つの方法を比べている．

・モデル 1（即時模倣）：対象の不寛容度に変更する

・モデル 2（漸近模倣）：対象の不寛容度に 1 だけ近づく

3.11 モデルのまとめ

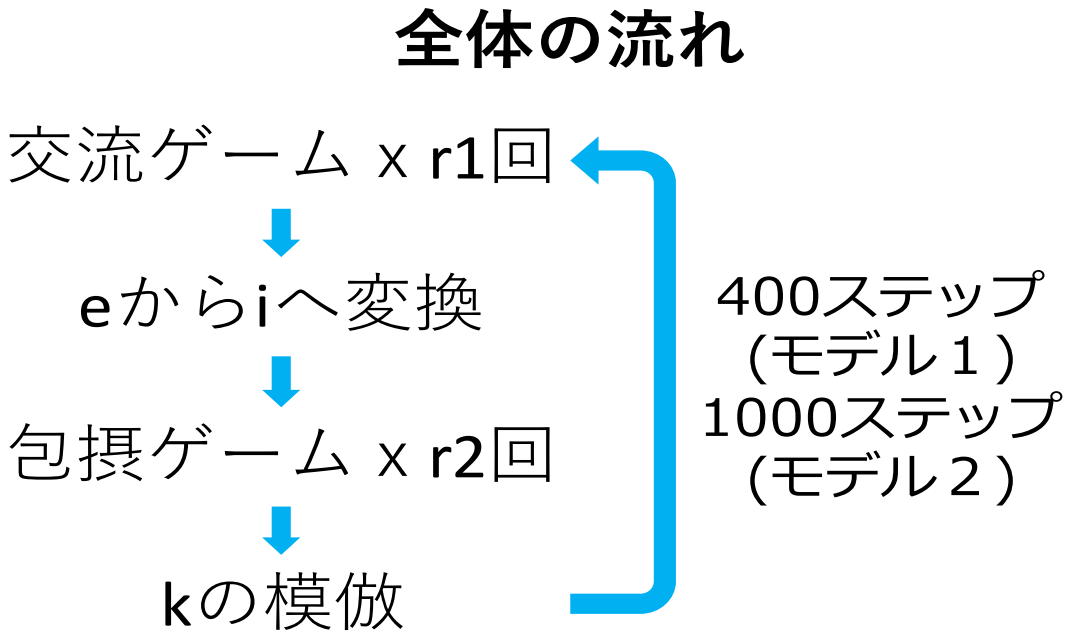


図 3.4 モデル全体の流れ

図 3.4 に示すように，全エージェントが交流ゲームを$r1$回行い，その交流で作られた経験スコアからイメージスコアが計算される．つぎに包摂ゲームを$r2$回行う．その後，各エージェントは他の不寛容度を確率的に模倣する[1]．この一連の流れを「1 ステップ」と呼ぶ．

過去に交流が上手くいき協力もする人は，他のエージェントがその人物をいい人だ

[1] 進化ゲームモデルでは，上位利得の戦略が増え下位利得の戦略が少なくなるという適応的進化（あるいは上位利得の戦略を模倣する社会学習）が使われる場合が多い．しかし本研究では，人間（特に高齢者）はそのように柔軟に自分の戦略を変更できたり上位利得の個体を見つけられたりするとは考えず，より穏当な社会学習方法を採用し，摸倣確率を小さくしている．

と思うてめ，交流と協力はどんどん上手くいきやすくなる．

自分が弱者の場合だと，交流ゲームで仲良くして経験スコアを上げておくと，包摂ゲームでは助けられやすくなる．

弱者ではなくても，協力の受容者がランダムに決まり，弱者ではない人でも助けてもらいたい場合があった際，交流ゲームで経験スコアを上げると後で得することになる．

第４章 シミュレーションの結果と考察

本章では，協力に影響するパラメータを紹介し，モデル別でシミュレーションの結果と結果の説明，及び結果から得られる考察を述べる．

4.1 パラメータ設定

表 1 パラメータ設定

パラメータ	数値	パラメータ	数値
総エージェント数	500	e 変化量（$e_u = e_d$）	0.5
交流ゲームの回数（r1）	3	i 変化量（$i_u = i_d$）	1
包摂ゲームの回数（r2）	2	p 変化量 ($p_u = p_d$)	1
参加コスト（β）	5	α	0.8
記憶維持できる ステップ数（S_m）	2	模倣確率	0.5
不寛容度の最大値（k_{max}）	11		

本研究で用いているパラメータ設定を表 1 に示す．以降の分析では，このうち一部を変更して調べているが，明示しない限りこの表の値を用いている．包摂ゲームでの協力は相対的に負担が大きいと想定しているので，イメージスコアの変化量は経験スコアの変化量よりも大きく設定している．初期値は$k = [1,11]$で一様分布，eは全て 6 である．本稿で示したパラメータ依存性以外のパラメータについては，結果は本質的に変化しないことが確かめられている．

シミュレーションは，モデル 1 は 400 ステップ，モデル 2 は 1000 ステップまで行

い，最後の 100 ステップの平均の 10run の平均値を図に用いている（時系列以外）

4.2　モデル１の結果と考察

4.2.1　参加コスト

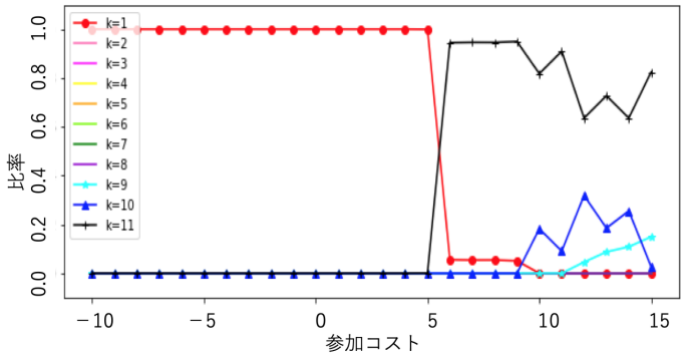


図　4.1　参加コストによる不寛容度の分布の違い

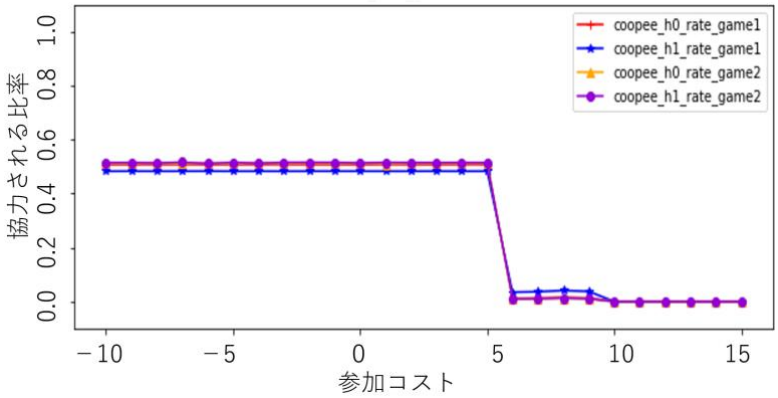


図　4.2　参加コストによる被協力率の違い

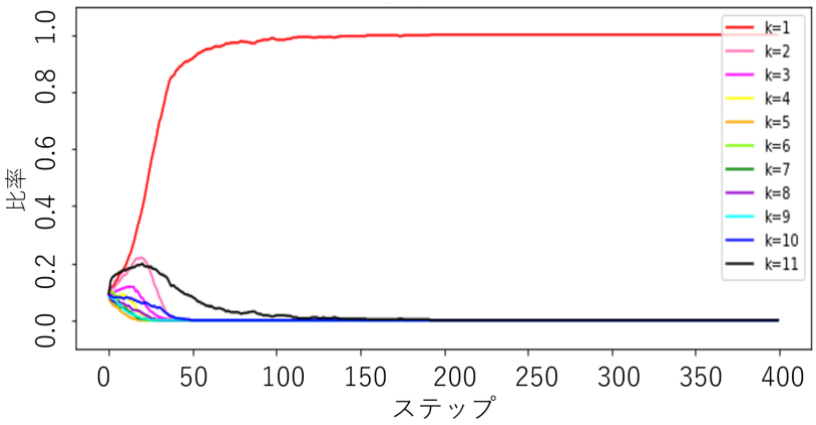


図　4.3　不寛容度の分布の時間変化

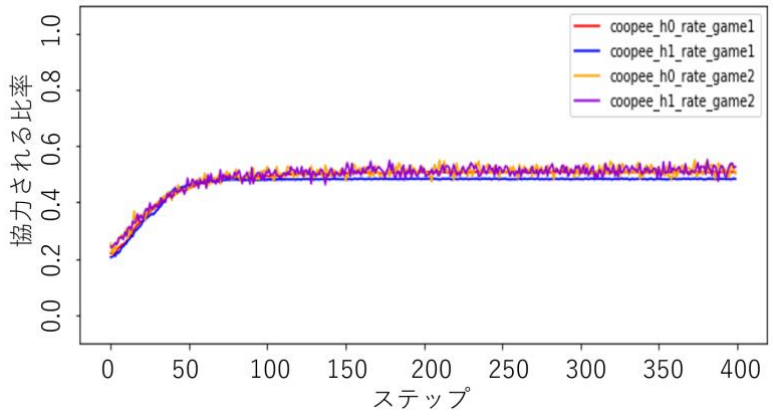


図　4.4　被協力率の分布の時間変化

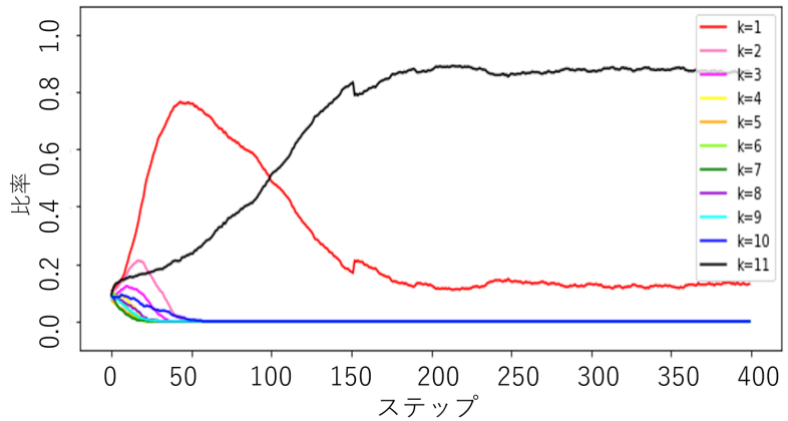


図　4.5　不寛容度の分布の時間変化

（β ＝5.05）

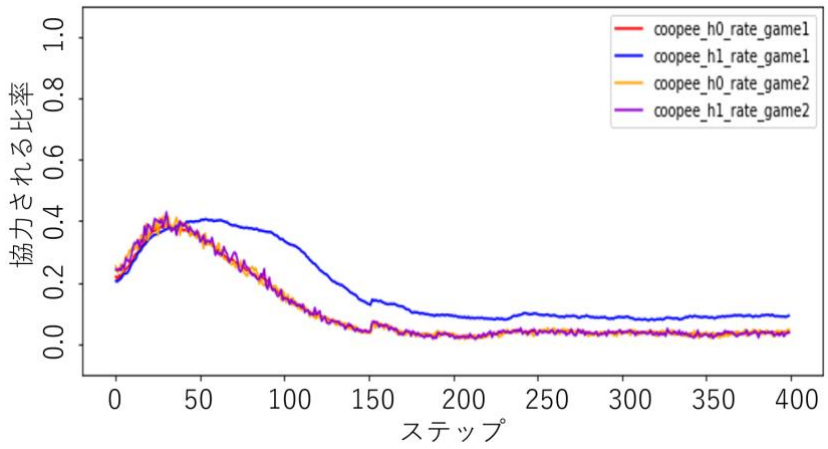


図 4.6 被協力率の分布の時間変化（β =5.05）

ここでの参加コストは交流のコミュニティを維持するために参加者が支払うべき費用と考え，便益を享受できるコミュニティを維持できる設計を検討するために，まず参加コストに対する依存性を確認した．

その結果，交流ゲームに参加するコストの影響が参加から得られる利益より小さいと，包摂ゲームでの協力がうまくいく．

図 4.1 は横軸に参加コストβをとり，各βにおいて各不寛容度kの値（$k = 1$～11）を取るエージェントがどの比率で安定になるかを示す．$\beta \leq 5$で全員が完全寛容の$k = 1$になる協力的な社会になる．逆にコストがこれより大きいと多くのエージェントが完全不寛容$k = 11$になるため，交流ゲームにはほとんど誰も参加しない．$\beta = 5$という閾値は，交流ゲームに参加することで利益が得られる最小の値である（e が初期値の 6，不寛容度$k = 1$とすると，参加コスト$\beta = 5$で利得が 0 となる）．

包摂ゲームにおいて弱者がどの程度協力されているかを「弱者被協力率」とする．この参加コストによる違いを図 4.2 に示した．赤の線が交流ゲームで一般人の被協力率，青の線が弱者の交流ゲームでの被協力率，黄色の線が包摂ゲームで一般人の被協力率，紫の線が包摂ゲームで弱者の被協力率．一番注目すべきなのは，包摂的なコミュニティで協力が必要とする弱者が実際に協力された比率（紫の線）である．以降被協力率の分布の図の説明は同じく．

図 4.3 と図 4.4 は図 4.1 の参加コストが 5 の時の不寛容度の分布の時間変化と被協力率の分布の時間変化を示し，全エージェントがすぐに k＝ 1 になり（図 4.3），協力

も起こっている（図 4.4）．図 4.5 と図 4.6 は図 4.1 の参加コストが 5.05 の時の不寛容度の分布の時間変化と被協力率の分布の時間変化を示し，k＝1 のエージェントがしばらく一番多いが，最終的には k＝11 が多くなり k＝1 も少し共存する（図 4.5），協力が起こっている状態から段々減っていき，最終的には弱者の被協力率はほぼ 0 になる（図 4.6）．

交流ゲームに参加するコストが参加から得られる利益より小さいと，包摂ゲームでは弱者は協力してもらえる．コストが高く交流ゲームに参加しない状況では弱者被協力率もほぼ 0 になる．弱者被協力率が 0.5 程度に留まるのは，ランダムに提供者・受容者が決まるため，提供者も弱者である可能性が半分あるからである．

今のモデルは町内会のような排他的コミュニティ，個人の便益と関係なくコミュニティに参加することが求められる．“みんなのためになるので町内会活動に行く”ということは，個人にとっては利益がなく，むしろマイナスとなってしまう．規範意識が高い人は行くことになるが，規範意識が低い人は行かずに，フリーライダーとして存在することになる

今回のモデル（←モデル１？）では，コミュニティカフェなど人が出入りするような場合をモデリングの対象としていない．一度でも損する場合，あるいは面白くないと思った場合は，かなり大きなメリットがないと参加しなくなる．

また，つまらなく不便益だとしても行くというのは古典的な町内会活動である．完全寛容な人（k=1）は，たとえ参加することの利得が低く個人的に不利益を得るとしても，集団の利益が高まると考えて行く人のことである．

そして，参加から得られる利益が参加コストより上回ると，参加の動機付けになる．

自分が想定している交流のコミュニティでは，利得が低いから行かないという設定はないが，交流ゲームに何回も利得がマイナスの場合，もう交流ゲームに参加しないという形式にすると，出入りの自由度が高いコミュニティカフェのモデルとして考えることができる．

4.2.2　交流のコミュニティの回数と包摂的なコミュニティの

回数の比率

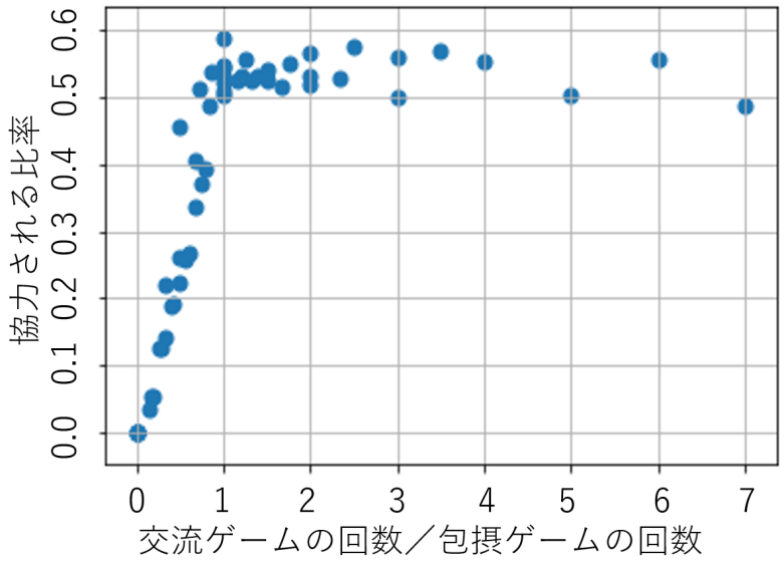


図　4.7　ゲーム回数の比による弱者被協力率の違い

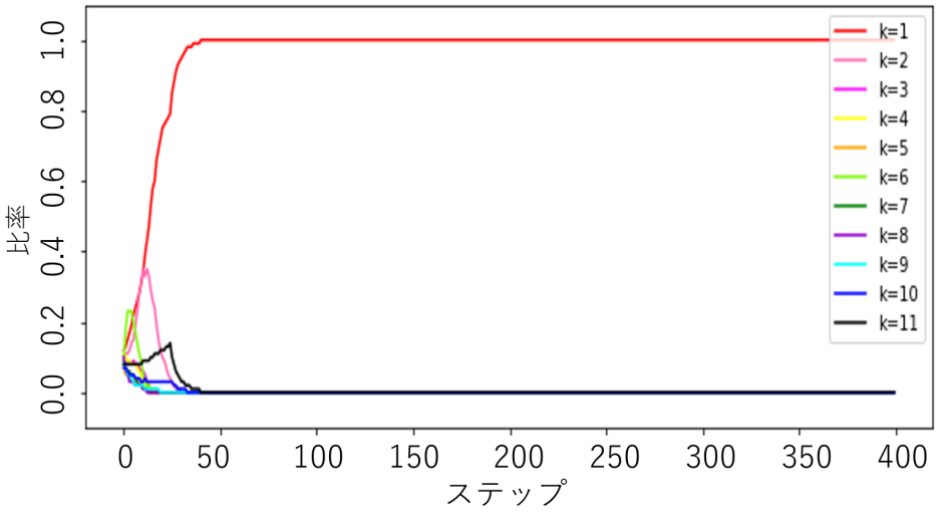


図　4.8　不寛容度の分布の時間変化

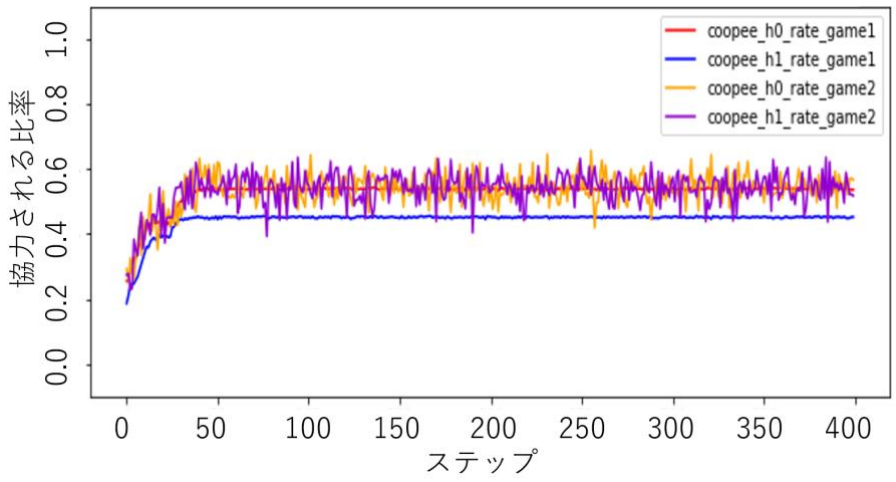


図 4.9 被協力率の分布の時間変化

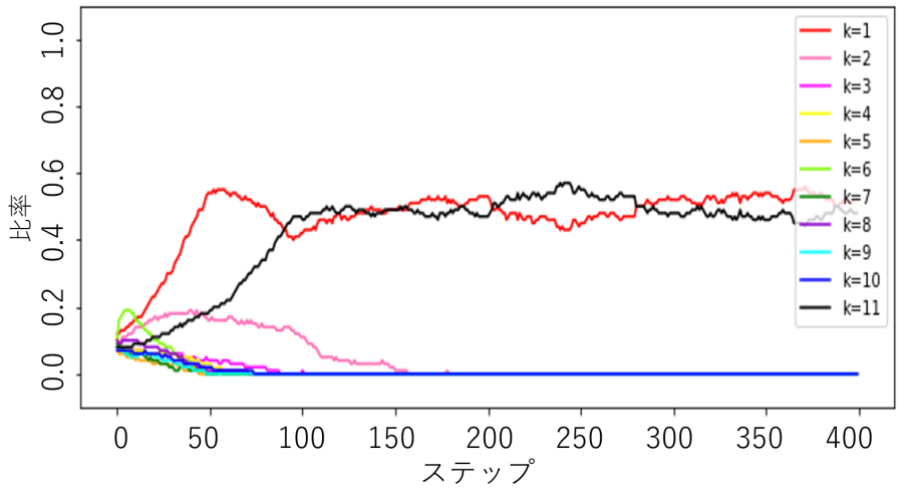


図 4.10 不寛容度の分布の時間変化

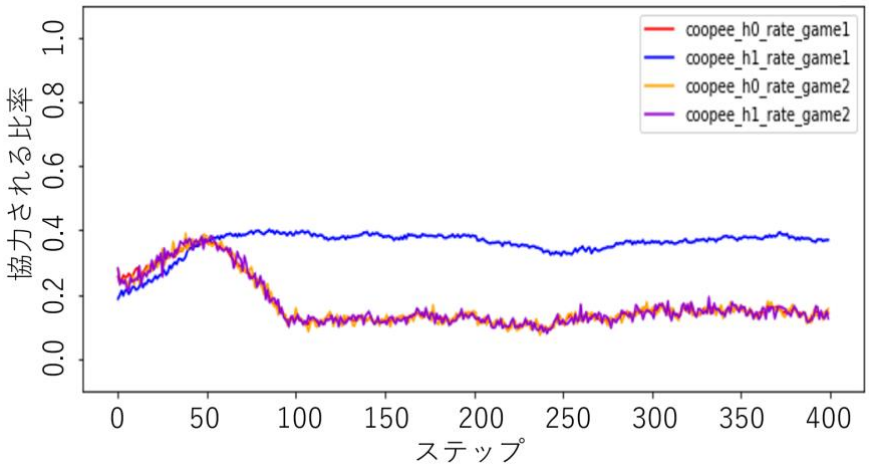


図 4.11 被協力率の分布の時間変化

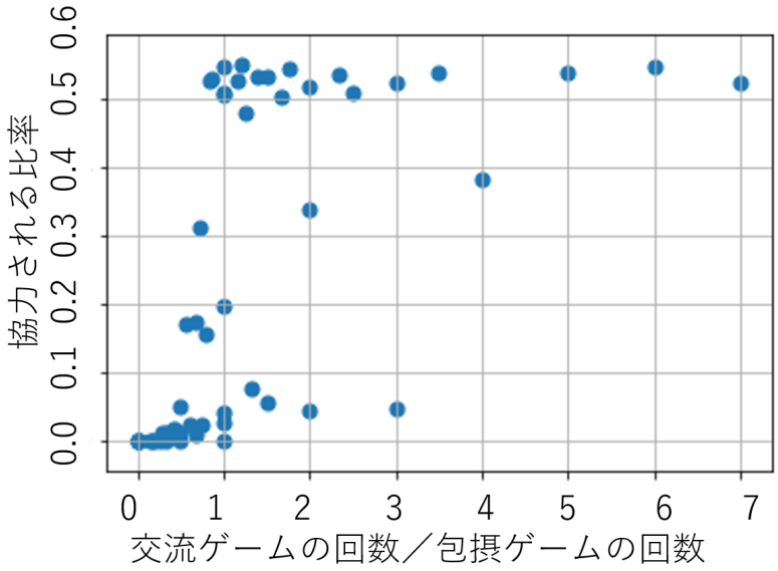


図 4.12 ゲーム回数の比による弱者被協力率の違い（β=5.05）

交流ゲームが包摂ゲームにどのような影響を及ぼすのかを見るために，交流ゲーム・包摂ゲームの回数を変えた場合の弱者被協力率を図 4.7 に示す．図 4.7 の横軸が交流ゲームの回数と包摂ゲームの回数の比率，縦軸がゲーム回数の比による弱者被協力率．交流ゲームを行う回数が包摂ゲームを行う回数より多いと，弱者がより助けられて，協力がうまくいく（図 4.7）．包摂ゲームのみ（横軸=0 の点）では，弱者はまったく協力されず，どのエージェントも協力行動をせず，間接互恵が成立していない．交流ゲームの回数が相対的に多くなると，包摂ゲームでの協力がうまく行くようになり，包摂ゲームの回数より多いと（横軸が 1 を超える領域），弱者が協力される．

交流ゲームでは不寛容度kが低い人の利得が上がりやすいが，包摂ゲームではkが低い人はよく協力をするので利得が低くなりやすい．交流ゲームでは参加コストβが低い場合は，回数が多ければ多いほどプラスになるため，結果的には利得がプラスになりやすい．したがって，交流ゲームが多いほど低い不寛容度kが模倣されやすくなる．包摂ゲームの回数が相対的に多くなると，協力によるコストが多くなるため，協力がうまくいかなくなる．

例えば，図 4.8（横軸がステップ数，縦軸が不寛容度の分布，図 4.10 も同じ）より，交流ゲームを 2 回，包摂ゲームを 1 回行う場合は，完全寛容$k = 1$のエージェントに支配される．図 4.9 の紫の線は 2 つのゲームで協力も多く行われていることを示す．図 4.10 より，交流ゲームを 2 回，包摂ゲームを 7 回行う場合は，完全不寛容$k = 11$

が多く，完全寛容$k=1$のエージェントと共存をしているが，完全不寛容$k=11$のエージェントがフリーライダーとして存在している．図 4.11 からは 2 つのゲームで協力が行われていないことが分かる．

図 4.12 より，参加コスト β（β＝10.10）が少し高く，交流ゲームを行う回数が包摂ゲームを行う回数より多い場合，弱者がより助けられて，協力がうまくいくことは変わらない．

包摂ゲームの回数が相対的に多くなる場合は，交流ゲームの影響が小さくなり，包摂ゲームのみ行われる状態に近いため，包摂ゲームでの協力が上手くいかなくなる，と考えられる．

また交流ゲームが多い場合，不寛容度kが低い人の利得が上がりやすいため，一番利得が上がりやすい不寛容度$k=1$が模倣されやすくなる．包摂ゲームでは，不寛容度kが低い人はよく協力をするため，利得が低くなりやすい．しかし交流ゲームでは，参加コスト βが低い場合は回数が多ければ多いほどプラスになりやすく，結果的には利得がプラスになりやすい．包摂ゲームの回数が相対的に多くなると，不寛容度kが低いほど協力に払うコストが多くなるため，利得が低くなりやすく，協力が上手くいかなくなると考えられる．

これらの結果より，交流のコミュニティを多く行い，さらにそこで高いメリットが得られ，噂が十分広がるようにコミュニティを作るべきだと考える．

4.2.3　模倣確率

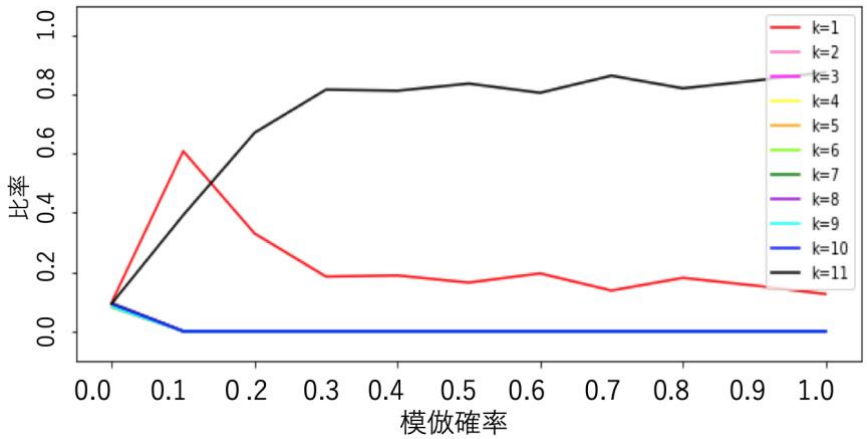


図　4.13　模倣確率による不寛容度の分布の違い

($\beta = 5.04$)

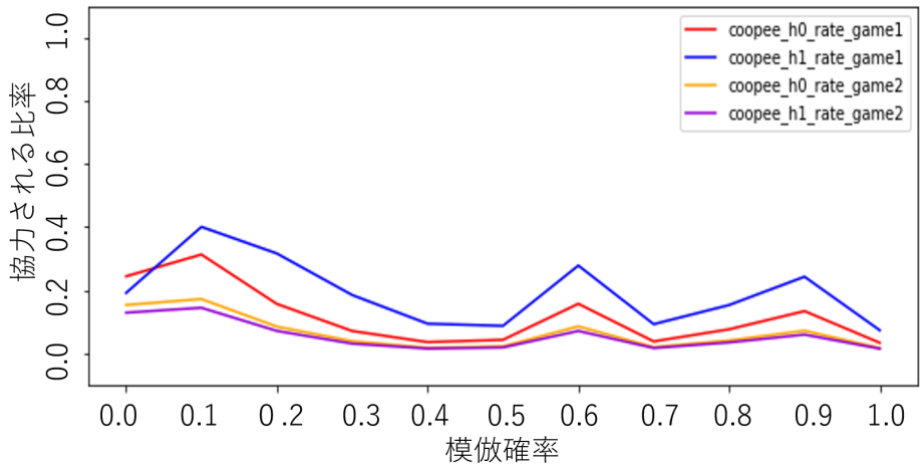


図 4.14 模倣確率による被協力率の違い（$\beta = 5.04$）

他者を模倣できる確率がどのように協力に影響するのかを調べる．そして，参加コストがやや高い，より厳しい状態で協力的社会になれるかどうかを観察するため，参加コストβを協力的社会ができない状態である 5.04 に設定した．図 4.13 の横軸が模倣確率，縦軸が模倣確率による不寛容度の分布である．図 4.13 が示すように，このコストの値では，不寛容度$k = 1$と$k = 11$が共存するものの，他のエージェントの不寛容度kを真似る可能性が 10%以下なら，$k = 1$のエージェントが一番多い．図 4.14 に示すように，模倣する可能性が 10%以下でも協力は起きていることがわかる．

模倣確率が低いというのは，不寛容度k が変わりにくいということである．参加コストβが低い場合は一番利益を得られそうな不寛容度$k = 1$に会う確率が高く，不寛容度$k = 1$を真似する確率が増える．包摂ゲームでは，不寛容度$k = 11$が協力せずに，協力される一方なので利得が増える，模倣できる確率が低い場合はすぐ変わらないので，$k = 11$に会える確率も増える，時間が経つと$k = 11$も模倣されて，$k = 11$が増える．

しかし，交流ゲームの方の利益が高いため，k=1の方が人数多くいるようになる．

すぐに学習しないという結果は，目先の利益に飛びついて学習しないほうが，より包摂的な社会になることを意味する．しかし，これは現実的に操作するのが難しい．

4.2.4　総人数

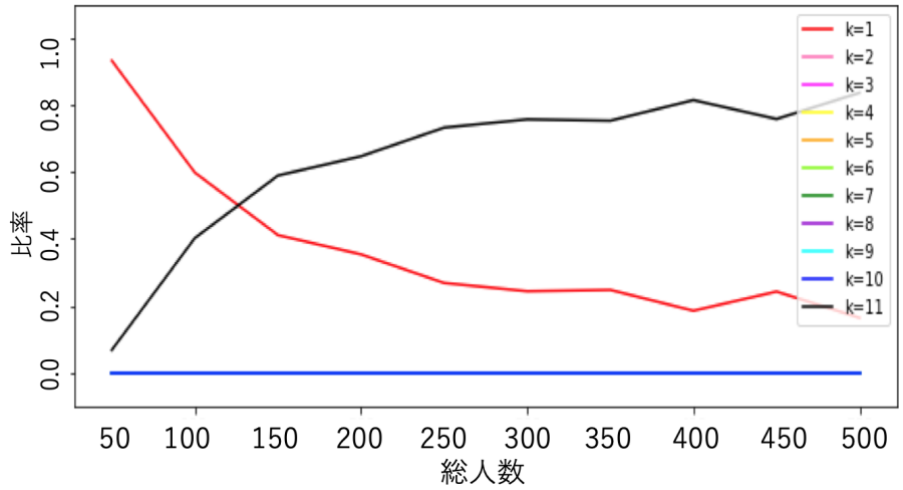


図　4.15　総エージェント数による不寛容度の分布の違い（$\beta = 5.04$）

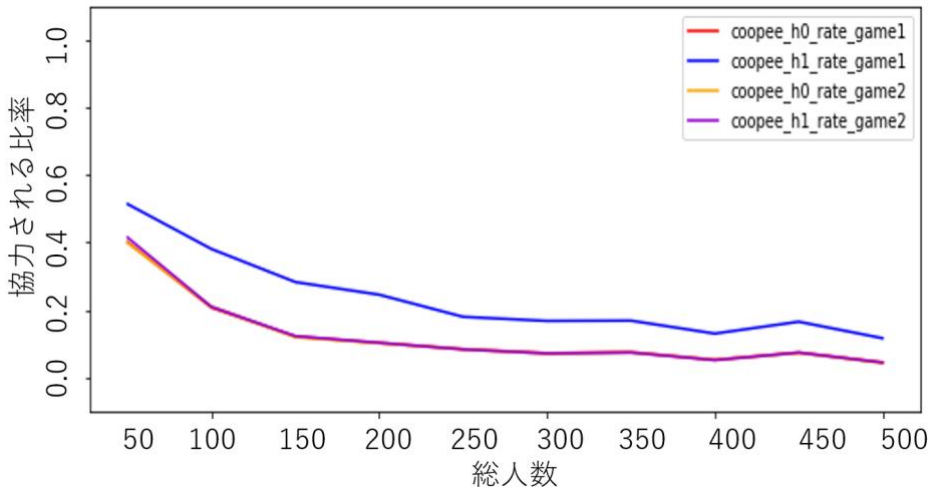


図　4.16　総エージェント人数による被協力率の違い（$\beta = 5.04$ ）

コミュニティの規模と包摂の成立の関係を調べる．図 4.15 の横軸が総人数，縦軸が総エージェント数による不寛容度の分布．図 4.15 が示すように，総数が 100 以下では参加コストがやや大きい場合でも$k = 1$が一番多くなり，参加コストの影響が少し大きくても完全寛容$k = 1$の人が一番多く，$k = 11$も少し共存する．図 4.16 からは総人数が 100 人以下であれば，参加コストの影響が少し大きくても，協力が行われていることがわかる．

今回のモデルでは，交流の相手をランダムマッチングするため，総数が少ないと会

ったことがある人と会いやすくなる傾向にある．

参加コストの低い場合には利得が高くなりやすく，$k=1$と出会いやすくなり，それが模倣されやすい一方，参加コストが高い場合は逆に$k=11$が模倣されやすい．

総人数が多いと$k=1$の人に会えないかもしれないが，ゲームの回数を増やすと会える確率は高まる．これは参加から得られる利得が高いため，模倣されやすい．

会ったことある人が$k=1$か$k=11$しかいないと，kがブレやすいが，多様な人がいれば模倣する対象が多様になるため，$k=1$の人に会う可能性が減る．

制度設計にはコミュニティの人数を少なくすると良いのではないかと考えられる．

4.3　モデル２の結果と考察

4.3.1　参加コスト

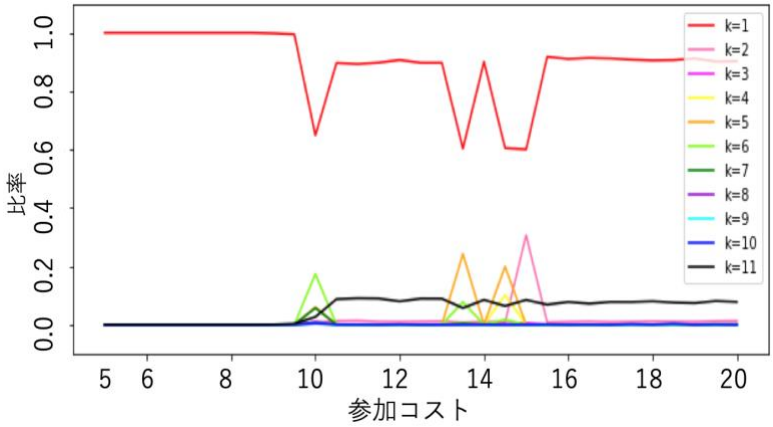


図　4.17　参加コストによる不寛容度の分布の違い

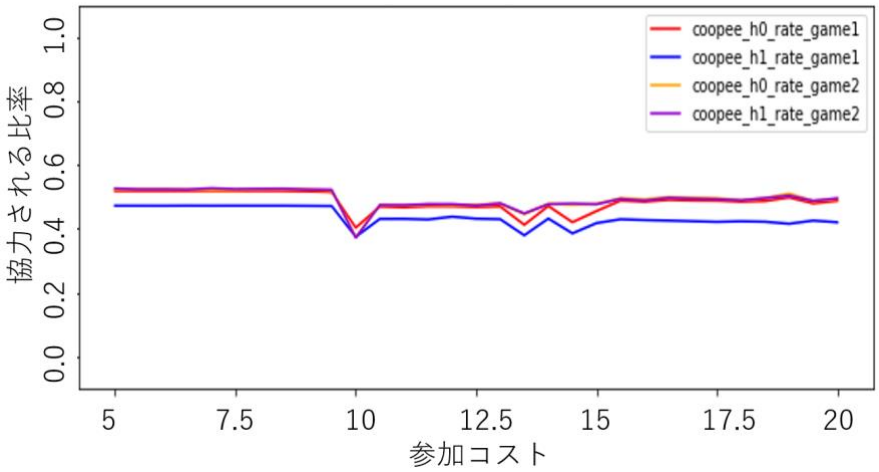


図 4.18 参加コストによる被協力率の違い

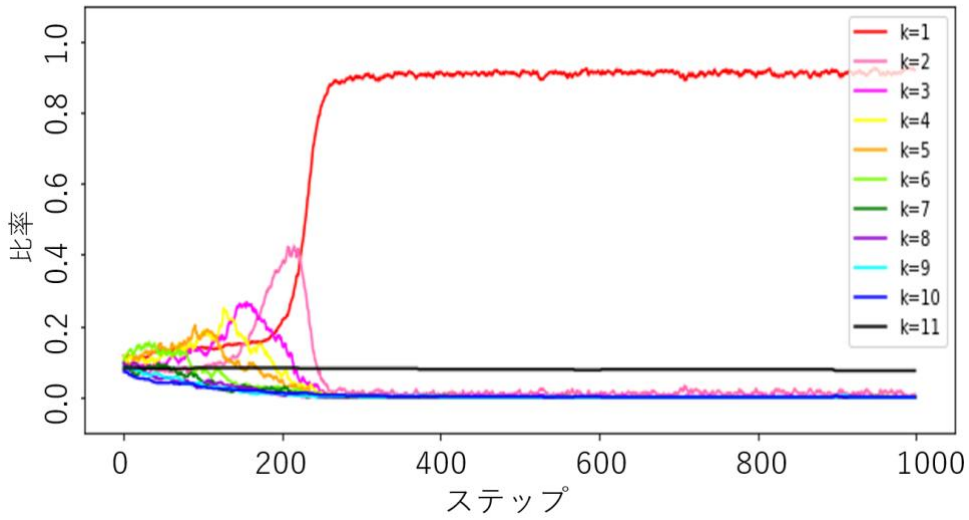


図 4.19 不寛容度の分布の時間変化

モデル 2（漸近模倣）の結果を示す．このモデルでは，模倣対象をすぐにまねるのではなく，その不寛容度に 1 だけ近づくようになる．包摂ゲームでの協力が参加するコストからほぼ影響されない．

図 4.17 の横軸が参加コスト，縦軸が参加コストによる不寛容度の分布．

図 4.17 と図 4.18 と図 4.19 では模倣確率を 0.5 にしているが，モデル 1 にくらべて実質的に模倣確率が低い状態になっている．モデル 2 の場合，多様な不寛容度のエージェントがいる状態が長期間過渡的に生じている．だとすると，毎回交流に参加して交流を楽しみ包摂ゲームでも協力する人と，まったく参加せず協力もしないという二極化するのではなく，たまに参加して交流したり，参加しても交流がうまくいかな

かったりという中間的であるが協力的な模倣対象が増える（図 4.19）．それらが媒体になってkが小さい方に近づいていくものと考えられる．

図 4.17 が示すように，参加コストβが 9.5 以上だと，完全寛容（k＝1）のエージェントが一番多いが，完全不寛容（k＝11）のエージェントもやや共存していることがわかる．多様なkがいることは，協力しないフリーライダーに罰を与えるエージェントがいるということであり，　kが高いエージェントが多くなることはない，しかし，完全寛容のk = 1という誰とでも協力するようなエージェントがいるので，完全不寛容のk = 11がフリーライダーとして少し共存する．図 4.18 は参加コストに関係なく，２つのゲームで協力が多く行われていることを示している．

4.3.2　模倣確率

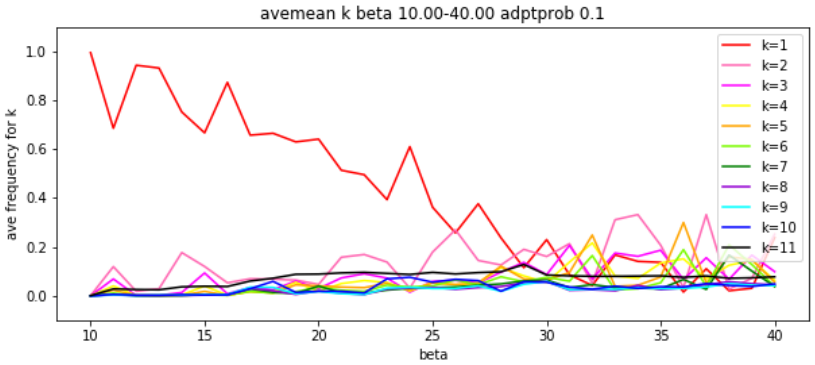


図　4.20　参加コストによる不寛容度の分布の違い（模倣確率＝0.1）

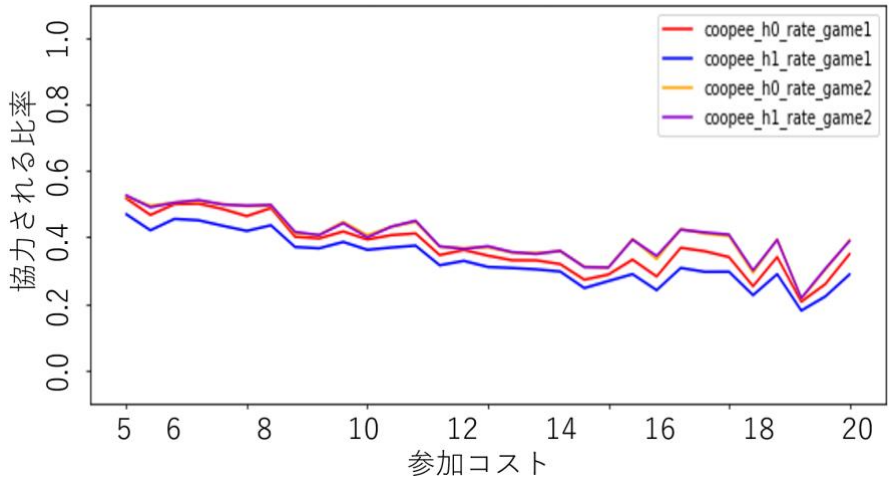


図　4.21　参加コストによる被協力率の違い（模倣確率＝0.1）

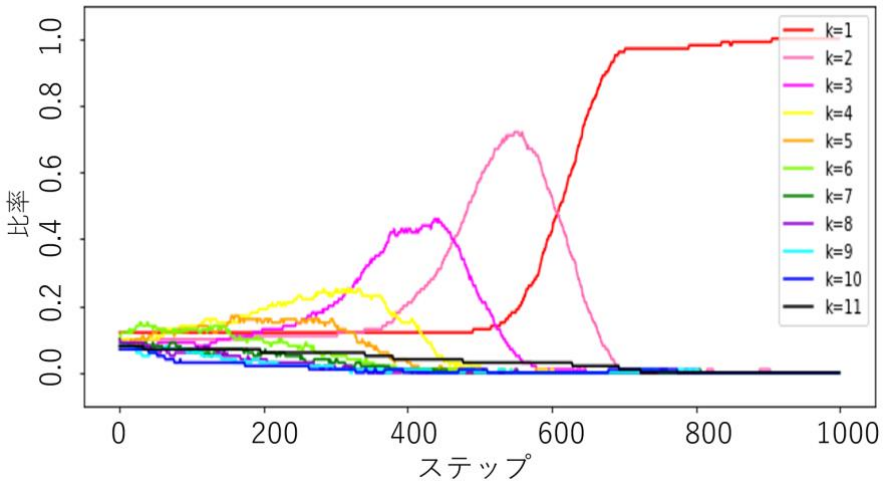


図　4.22　不寛容度の分布の時間変化（模倣確率＝0.1）

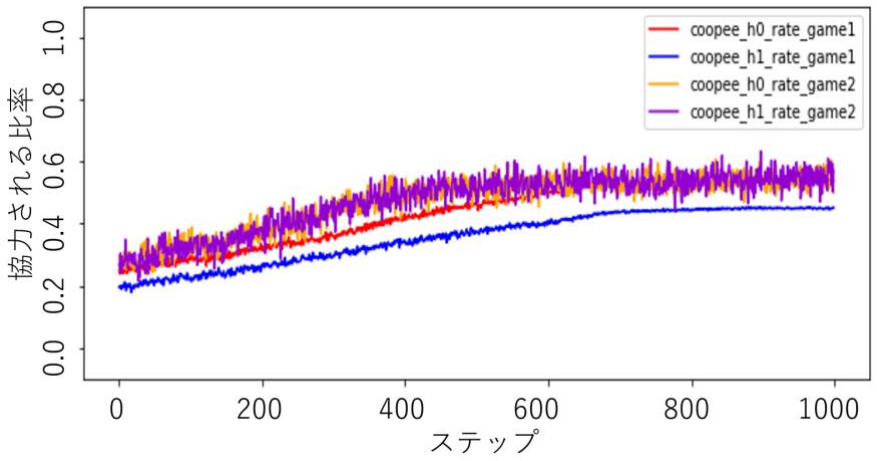


図　4.23　一般人と被協力率の時間変化（模倣確率＝0.1）

モデル 2 は，模倣対象をすぐにまねるのではなく，その不寛容度に 1 だけ近づくようにしている．この条件から自明であるが，図 4.22 が示すようにゆっくりと変化し収束が遅くなる（図 4.22 の横軸が参加コスト，縦軸が参加コストによる不寛容度の分布の違い）．

最終的には$k = 1$が占めるようになるが，それまでに様々な k が低いエージェントが共存する状態がある（ここでは過渡的変化がわかりやすいように模倣確率を 0.1 に設定している．0.5 の場合は収束が早くなるがダイナミクスは定性的に同じである．）．図 4.23 が示すように協力も少しずつ増える．模倣確率が 0.5 の場合でも，図 4.20 と図 4.21 で示すように，この模倣方法の場合は，参加コストが 5 より大きくなっても

不寛容度と弱者被協力率への影響が少なく，協力的な社会ができている．

これらの結果より，他者の不寛容度をすぐに模倣しない方が，より包摂的な社会になるということが分かった．なぜなら，kの種類が沢山あり，多様な人がいるというのは結果に影響する．つまり，$k=1$と$k=11$の間の人達が媒体になって，1 に近づいたり，11 に近づいたりすることを通して，$k=1$か$k=11$が増えていくことになるからである．最終状態は$k=1$に収束するが，その過程にはいろんな人物が存在していることは重要である．

4.3.3　総人数

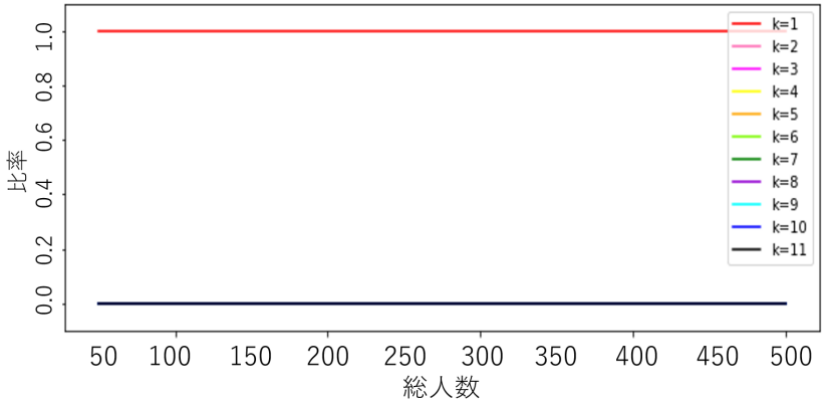


図　4.24　総人数による不寛容度の分布の違い

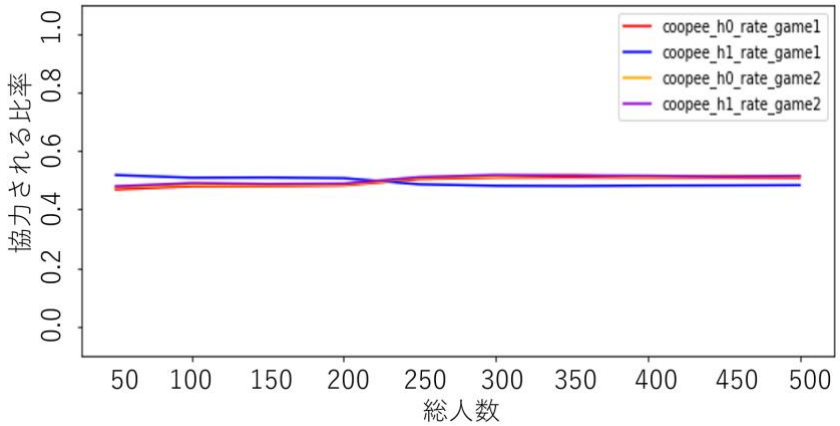


図　4.25　総人数による被協力率の違い

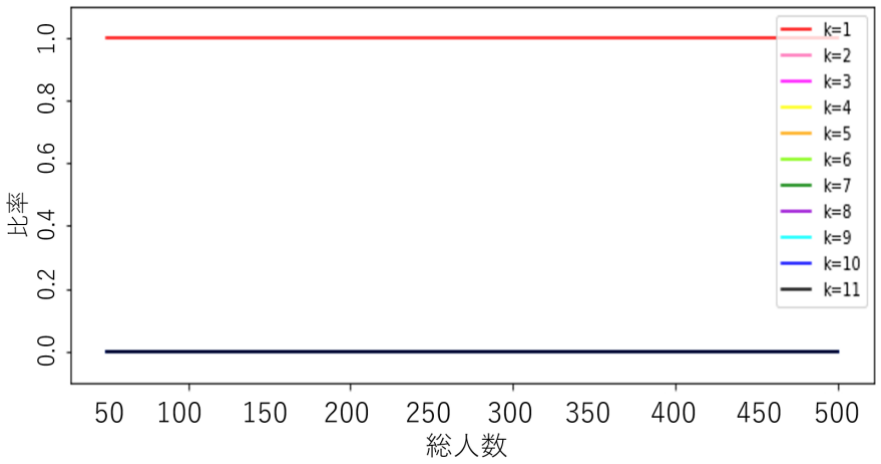


図 4.26　総人数による不寛容度の分布の違い　（$\beta = 6$）

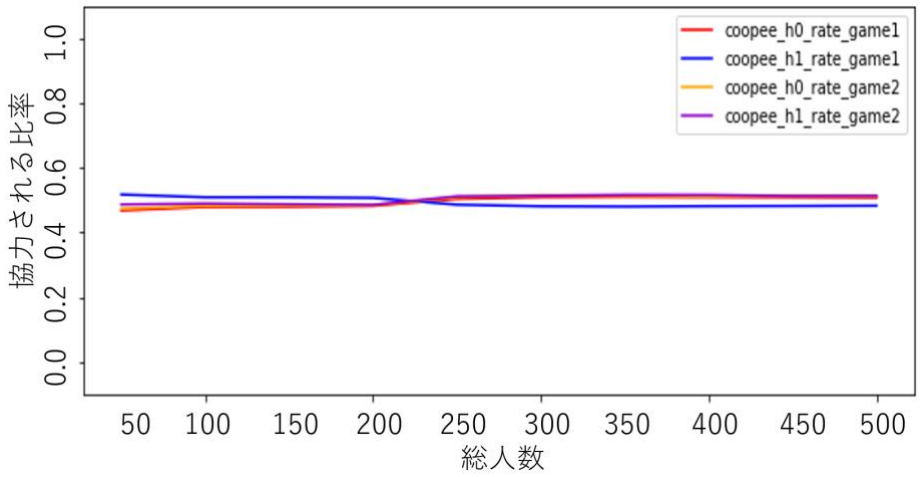


図 4.27　総人数による被協力率の違い　（$\beta = 6$）

図 4.24（$\beta = 5$）の横軸が総人数，縦軸が総人数による不寛容度の分布，図 4.24 が示すように，総人数が多くて 500 人でも，参加コストが少し大きくても（図 4.26，$\beta = 6$）完全寛容（$k = 1$）に支配され，他のkと共存しない．図 4.25 と図 4.27 より，総人数が 50 から 500 までの場合，総人数と関係なく協力が行われていることがわかる．

総人数が多くてもモデル 2 のような模倣の仕方では，様々な種類の k がモデル 1 と比べて長くいられる．そうすると多様な模倣対象がいる中で，交流ゲームで高い利得を得られやすい低いkを持つエージェントが増え，その中で一番高い利得が得られる可能性がある$k = 1$のエージェントが最終的には一番多くいることになる．

4.3.4　交流・包摂ゲーム回数の影響

図 4.28 が示すように，包摂ゲームのみ（横軸=0 の点）では，弱者はまったく協力されず，どのエージェントも協力行動をせず，間接互恵が成立していない．交流ゲームの回数が 1，包摂ゲームの回数が 7 未満，つまり交流ゲームの回数が少なすぎなければ，協力が起きる．

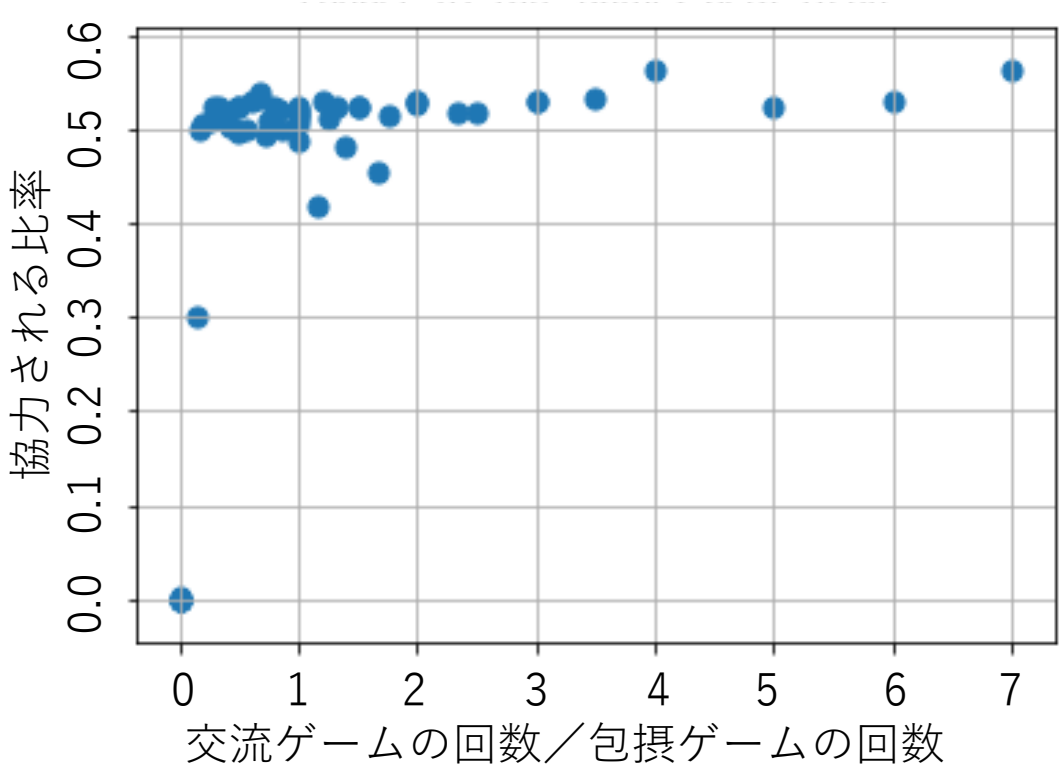


図　4.28 ゲーム回数の比による弱者被協力率の違い

4.4　協力に影響しないパラメータ

本研究では，以下のパラメータについても観察したが，協力に影響を及ぼさないことが分かった．そのパラメータは，経験スコアeの変化量，イメージスコアiの変化量，利得pの変化量，経験スコアがどの程度反映されるかを決めるパラメータα及び，記憶できるステップ数である．

参加確率は不寛容度kと連動して，kが小さいほど参加確率が高く，kが大きいほど参加確率が低い．交流ゲームには経験スコアeやイメージスコアiや利得と関係なく，

kが小さければ交流ゲームに参加する．そして，交流ゲームで出会った人はkが低い可能性が大きい. 包摂ゲームでkが高い人と出会っても, 交流ゲームの回数が多いので，模倣をするときは小さなkをまた模倣する可能性が大きい．交流ゲームに参加しているkが低い人たちはよく協力をし，eの変化量と関係なくeが相対的に高い．交流ゲームではkが低い同士が協力しあうことになる. 包摂ゲームが終わったあとにkの模倣をするので，交流ゲームであった人数が多いことから，低いkを模倣する確率が高い．包摂ゲームでiと利得pが変わっても，会った人の中でkが低い人が多いことは変わらない.

つまり，参加確率によってどのような k の人に会うかがおおよそ決まり，模倣の対象が決まる．よって，経験スコアeの変化量，イメージスコアiの変化量，利得pの変化量は協力に影響しない.

第５章　　議論

本章は，結果から得られたモデルに対する考察を述べ，現実の例を挙げその例がなぜそうなっているのかの理由を解明する．さらに，制度設計のヒントを提示する．

5.1 モデルに対する考察

モデ 1 とモデル 2 を比較して得られたことは, 模倣確率が協力率を左右するのではないということである．モデル 2 では，β が大きくても協力はされている．kは単なるラベルではなく，その変化の過程に意味があるということであり，kが徐々にしか近づけない，悪い人になろうとしてもなりきれず，いい人の枠が広がっていく．モデル 1 では，kの低い人の方が多いけども，kの高い人の方が得をしており，一気に変われるから，kの低い人が一気に減ってしまう．モデル 2 は，二極化してすぐ真似られるではなく，どっちの方に行こうとしても徐々にしか行けない性質が効いている．

協力行動が起こっている集団の参加するメンバーが変わらないことと, モデル 1 では 50～150 ステップで収束する. モデル 2 では 250～700 ステップで収束することから, 関係の閉鎖性や長期性が必要だと考える. もしメンバーの流動性があるとしても, 出ると同時に入ってくるメンバーが必要であろう．

Nowak and Sigmund（1998a, b）に代表される生物学者は，資源の提供者と被提供者がランダムにマッチングされる状況を想定しており，それ自体は相手を選べないことを意味している．本研究が想定している包摂的なコミュニティでも，いざという時に助けてもらえる相手は選べず，誰がいつ協力を必要とするのは予測できないといった点で，ランダムマッチングの妥当性はあると考えられる．

本モデルはランダムマッチングを設定しているが，知り合いとよく合うなどのマッチングの仕方もある．知り合いとよく会う場合には，ランダムマッチングより協力が維持されやすいと予想される．ランダムマッチングは実際には現実的ではないため，かなり厳しい状況を設定している．ランダムマッチングでは会った人は必ずしもいい人ではなく，つまり交流ゲームで会うのはkが低い人．ランダムマッチングでない場合，kが低い人とよく会うことになり，いい人同士が固まりやすく，いい人同士で真似しやすくなり，より協力が維持されやすい．

本研究においては利得を相対的な形でのみ使用しており，それは現実的にどうなっているのか，人はそのような相対的な比較だけで生きているのか，を検討するために利得を見た．

参加コストが非常に高い場合でも，人々は協力的になり（図 5.1），協力が起こって

いることがわかる（図 5.2）．しかし，参加コストが低くない時（図 5.4）の利得と比べて，利得を見ると最低利得が非常に低くなっていることがわかる（図 5.3）．

大変な不利益を負わせても，現実的には人々は協力をするのかは疑問である．モデルで抽象化していた，利得がコミュニティの中での相対的な比較，そしてコミュニティの外に目を向けた場合，もっと楽に過ごしている人たちがいるはずであり，その人々を見てコミュニティ内の人々は住むところを移動するだろう．移動の自由がない公権的な社会，村社会では本モデルが成立するが，現代では移動は自由であるため，コミュニティ内の協力の均衡は破綻すると予想できる．

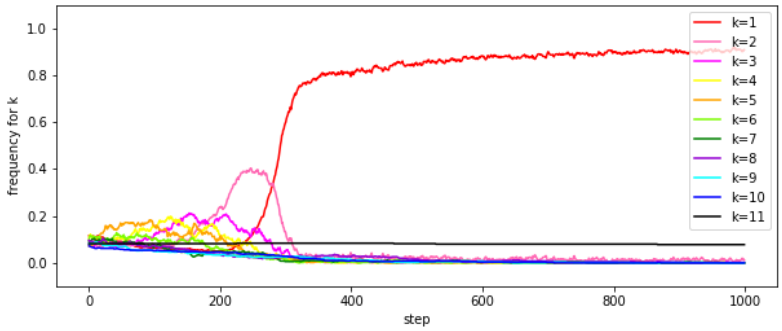


図　5.1　不寛容度の分布の時間変化　（β =20）

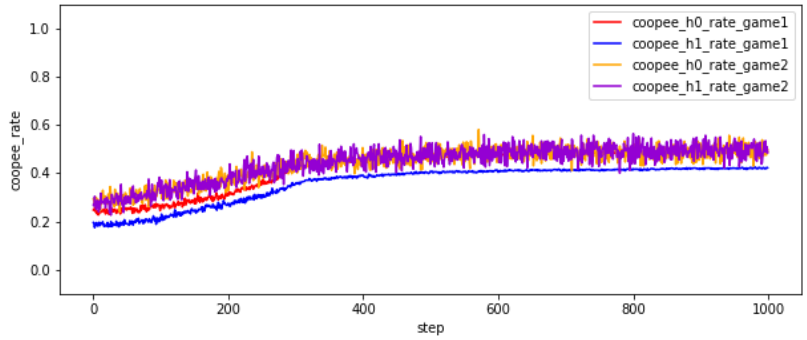


図　5.2　一般人と弱者被協力率の時間変化　（β =20）

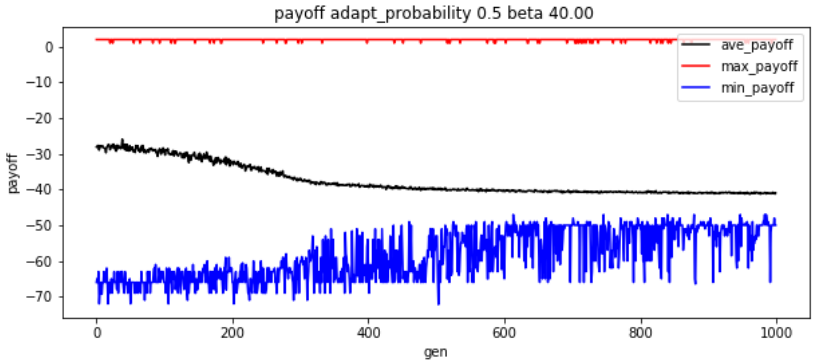


図　5.3　利得の分布の時間変化　（β ＝20）

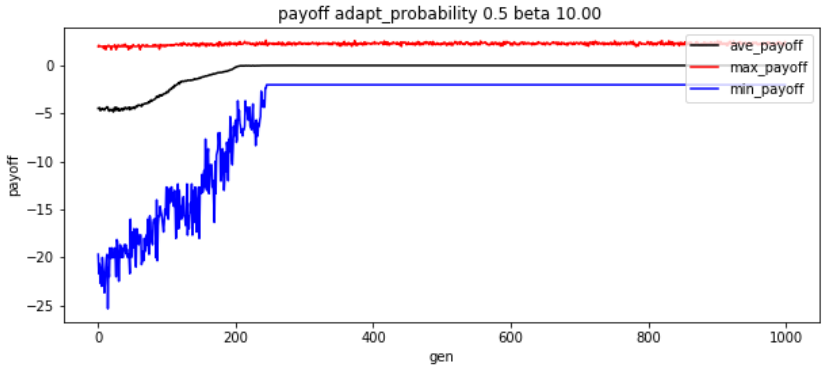


図　5.4　利得の分布の時間変化　（β ＝5）

5.2 現実との対応

自治会入会金と参加費が高く，でも参加者が参加し続けるという記事がある．（西日本新聞 2019/01/30）記事の中に「釈然としないが，支払いを断れば集落に居づらくなるから」という参加者の声があった．本モデルに当てはめて，なぜそうなるかを考えると，すぐに参加しない人の模倣をするのではなく，k＝11に近づきたくても，しばらくはずっとkが低い良い人で居続けなければならない．この近づいていく過程で，k が低い人は参加確率が高いため，交流ゲームに参加し続ける．模倣をする場合も，参加ゲームで出会ったk の低い人たちを模倣する確率が高く，結局はk が低い状態で居続けることになる．

本研究では，行動の主体が何らかのコストを負わせて，協力を成り立たせるにはど

うすればいいのかという理論的な問いに対して，すぐに模倣できない，模倣対象に少しずつにしか近づけないと協力が起こるということを示した. また, Gilbert（2008）のように，直接互恵か間接互恵かどちらかを選ぶではなく，直接互恵と間接互恵を組み合わせることによって，間接互恵での協力が起こることも分かった.

5.3 制度設計のヒント

事例研究（長谷川・小川 2015）では，交流のコミュニティを踏まえることで包摂的なコミュニティができたが，本研究でも交流ゲームがあることで包摂ゲームだけでは生じなかった間接互恵が成り立ち，行動だけではフリーライダーである弱者も協力される包摂的なコミュニティができることが示された．そして，包摂的なコミュニティが成立にいたる過程では不寛容度が低めの多様なエージェントが存在することが有効であり，そのためには模倣対象をすぐに模倣できるのではなく，模倣対象に少し近づくというゆっくりとした社会学習がよいことが示唆された.

全体の人数が少ないと，前に会ったことある人に会う頻度が増えて，包摂的なコミュニティになりやすいことも示された．会う頻度が増えると，交流回数や相手の変化に気づく回数が増えるかもしれない．相手の異変に気付くと，より確実な包摂に繋がるだろう．また，交流ゲームの回数が包摂ゲームの回数より多いと協力を増やしやすい．現実には，交流が行われる施設や場所が必要となり，その場の運営と支援が必要となる.

一方，すぐに模倣できる場合（モデル 1）は，交流ゲームへの参加コストが全体の状態をもっとも左右する．モデル 1 は町内会のような規範的コミュニティになっていると解釈できるだろう．すなわち，個々の便益と関係なく参加が決められている．町内会活動は全体の利益のために参加するが，個人にとっての利益がない，もしくはマイナスになるような人もいる．規範意識が高い人は参加するが，規範意識が低い人は行かずに，フリーライダーとなる.

規範意識の高い人は参加コストが高く自分が損をしても，全体の利益のためになると考えて交流のコミュニティに参加するが，そのような規範意識の高い人はフリーライダーから搾取されることになる．そのような社会は望ましくないため，行政が参加

コストを下げるような制度を作った方が望ましいが，それができない場合はコミュニティ自身で作らなければならず，それがさらなるコストを生み出す．

このような場合，直接的に参加コストを下げるのではなく，別の手段で寛容度を上げることを考える必要がある．すなわち，直接互恵でしか動かない人に対して，間接互恵によって良い結果をもたらした話を伝え続けることが重要になる．例えば，以前あった災害時に助け合った話を語り継ぐ，あるいは，困ったときにこんなに良いことがあったなどと言い続けることが該当する．間接互恵が良い結果をもたらした話を言い続けることが目先の利益に飛びつくような即時的な模倣ではなく，じっくりと学習する教育にも繋がるかもしれない．すぐに変えるのは難しいが，いつかはみんなの模倣確率が下がるかもしれず，そうすると包摂的なコミュニティで協力が増えることにも繋がる．

第 6 章 結論

本章は，今までの述べてきた目的やそれを実現するモデルやそこから得られた結果と結論をまとめ，今後の課題を述べる．

6.1 まとめ

本研究は，社会的排除のリスクが高い高齢者を包摂する地域コミュニティが成立する条件を明らかにするために，直接互恵による交流ゲームと間接互恵による包摂ゲームの 2 種類の協力ゲームを行うコミュニティをモデル化した．モデルでは一般人と弱者，そして，交流のコミュニティと包摂的なコミュニティという 2 種類の局面を考えた．モデルでは，他者とのつきあいを判断する基準と他者への評価値（経験スコアとそこから作られる印象スコア）を持つ主体が，直接互恵による「交流ゲーム」と，間接互恵で成り立つコミュニティを表す「包摂ゲーム」で相互作用する．交流ゲームでは，相手に対する経験スコアと自身の判断基準の差および判断基準に応じた参加コストにより，利得と経験スコアが変わる．包摂ゲームでは主体はランダムに協力の提供者か受容者となり，提供者は受容者に返報を求めない協力をするか否かを，判断基準と印象スコアから決定する．協力すると利得が下がるが印象スコアが上がる．何回かのゲームの後，各主体は最大利得主体の判断基準を確率的に模倣する．

そして，２種類の社会学習方法を採用した 2 つのモデルを比較することを通して，社会的包摂が安定的に維持される条件をシミュレーションによって探索した．

結果から，参加コストが小さく，参加人数が少なくすること，交流ゲームの回数が包摂ゲームのそれを上回ること，そして社会的学習が緩やかに進む条件で社会的包摂が実現されていることを示した．

6.2　結論

本研究の結果から，交流のコミュニティで十分の利益を得る，噂は十分出回る．そして社会学習がゆっくり進行することで社会的包摂が実現されることが分かった．払ったコスト分だけのメリットをもらうという直接互恵的な考え方ではなく，「自分が助けた人だけじゃなく，助ける行為を誰かが見ているので，いつか自分も協力してもらえるかもしれない」という間接互恵的な考え，そして，そういう協力を差し出す関係が回り回るベースとなるコミュニティが維持されるという「向社会的（集団主義的）」な考えを取ることを勧めることが有効だと言える．

間接互恵関係は現代的・個人主義的な考え方からすると納得されにくいかもしれない．しかし，個人主義をつきつめていくと利他行動を実現する手段として直接互恵しか残らなくなってしまう．それどころか，自分の利益のみを追求する，利他行動をしない集団に陥りかねないという現代社会の問題に直面する．

そうした風潮に対するアンチテーゼとして，コストをある程度高く払ってでも協力コミュニティを維持することが，包摂的なコミュニティに繋がるということを，このモデルのシミュレーション解析で実際に示したことになる．

直接互恵でしか動かない人に対して，間接互恵によって良い結果をもたらした話を伝え続けることが重要になる．間接互恵が良い結果をもたらした話を言い続けることが目先の利益に飛びつくような即時的な模倣ではなくゆっくり学習する教育にも繋がるかもしれない．すぐに変えるのは難しいが，いつかはみんなの模倣確率が下がるかもしれず，そうすると包摂コミュニティで協力が増えることにも繋がる．

6.3　今後の課題

本研究では，直接互恵による交流ゲームと間接互恵による包摂ゲームこの2種類のコミュニティをモデル化し，社会的包摂が安定的に維持される条件について，シミュレーションを用いて解析した．しかし，抽象化をしていくうちに，本来考えなければならない要素や現実的の要素を捨象してしまった可能性が大きい．この節では，本来考えなければならないことや現実的の要素について触れたい．

今のモデルは町内会活動という一種類のコミュニティ形式を解析したが他の交流のコミュニティをモデリングする必要もある．例えば，コミュニティカフェなど行くことが強制されていない，楽しくないなら行かないようなコミュニティをモデリングし，今の町内会活動のようなコミュニティと比べることで，制度設計としてより多様なコミュニティを導入することが有効なことを示せるかもしれない．

本研究では，人の出入りがないというように設定しているが，コミュニティカフェやコミュニティサロンをモデリングすると，人の出入りが自由という点を加える必要がある．モデル 2 の結果から，途中には様々な人がいることが大事だと分かったが，収束に連れて全員が最終的には$\mathrm{k}=1$になる．もし多様な不寛容度kがいることに意味があるなら，終始同じ人ではなく，様々な不寛容度を持つ人が参加することで結果が変わる可能性がある．例えば，いろいろな高齢化率の進行状態をみるために，高齢者と一般人の人口の比率を変えるなどが考えられる．

現実から離れた極端な状況を想定すると，規範意識が高い人とフリーライダーの人しかいないとどうなるのかを観察するために，いろいろな不寛容度がいる状態から始めるのではなく，$\mathrm{k}=1$と$\mathrm{k}=11$のみから始まるシミュレーションが必要であろう．不寛容度が一種類の k のみからはじまるなど，様々な人がいない状態から始まるとどうなるのかも興味深い．

高齢者に代表される弱者がフリーライダーとして排除されないように，協力しなくても評判は下がらないようにしているが，協力しなかったら弱者も評判が落ちるように設定し，どのくらいに評判が落ちても排除されないのかを解析する必要がある．

包摂ゲームでは，イメージスコアがkより大きければ必ず助けるというように設定しているが，相手がいい人であっても必ず助けてあげるというのは現実的ではない，もっと現実的にするためにはイメージスコアが，kより大きくてもたまには助けないという設定をする．この上で，どのくらいの頻度で相手を助ければ，包摂的なコミュニティで弱者が協力されるのかを解析する必要がある．

本研究で行ったゆっくりとした社会学習が，普通の進化のモデルにおいても同じ結果が見られるのかは興味深い．

抽象化した社会学習の仕方は，現実の人間の場合どうなっているだろう．社会心理学的なことで模倣の仕方が現実にもできるのかどうかを見る必要がある．

いざという時に助け合いをする包摂的なコミュニティ自体は設計できないが，交流

コミュニティのメリットを高めることができる，交流のコミュニティでは噂が出回るようにするしかない．さらに，交流コミュニティで得られるメリットが包摂的なコミュニティで支払うデメリットコストよりも大きい設定になってないといけない．噂が出回るようにするには，評判のメカニズムを IT で実現することができる．

謝辞

本研究を遂行するにあたり，指導教員としてご指導，ご意見を下さった北陸先端科学技術大学院大学知識マネジメント領域の橋本敬教授に深く感謝の意を申し上げます．研究に対して何もわからなかった私に対し，丁寧にご指導頂きましたこと，深く感謝申し上げます． 論文審査におきましては，審査委員である池田満教授，内平直志教授，白肌邦生准教授から研究に関する貴重なご意見を頂きまして心より御礼申し上げます． 北陸先端科学技術大学院大学知識マネジメント領域の小林重人講師は橋本研究室のゼミやミーティングで，いつも議論に付き合っていただき多くの有益なアドバイスを頂き，本研究をより充実させることができました．深く感謝致します．最後に，研究だけでなく精神的にも多くのサポートを頂いた橋本研究室の皆様に深く感謝の意を表します．特に，先輩であるドクターの李冠宏さん，外谷弦太さん，藤原正幸さんからはゼミや発表でアドバイスを頂き，多くの面で助けて頂きまして，本当にありがとうございました．

橋本研究室にいた 3 年間が大変有意義な時間を過ごせて，人生の中でかけがえのない時間になりました．ここで学んだことは研究だけでなく，これからの人生にも役立つものだと思います．今後は一社会人として，ここで学んだことを活用しながら，進んでいきたいと思っております．

参 考 文 献

Bar-Tal, D., Sharabany, R., Raviv, A.（1982）. Cognitive basis for the development of altruistic behavior. In V. J. Derlega &J. Grzelak（Eds.）, Cooperation and helping behavior 7-henry and research. New York: Academic Press, 377- 396.

Gilbert, R.（2008）. Evolution of direct and indirect reciprocity. Proceedings of the Royal Society of London. Series B. 275, 173-179. Doi:10.1098/rspb.2007.1134

Gilbert, N. and Troitzsch, K.G.（1999）. Simulation for the Social Scientist, Open University Press

Leimar, O., Hammerstein, P.（2001）. Evolution of cooperation through indirect reciprocity. Proceedings of the Royal Society of London. Series B. Biological Sciences,268,745-753.

Nowak, M.A.& Sigmund, K.（1998a）. Evolution of indirect reciprocity by image scoring. Nature,393, 573-577.

Nowak, M.A.& Sigmund, K.（1998b）. The dynamics of indirect reciprocity. Journal of Theoretical BiolOgy,194,561—574.

Nowak, M.A. Sigmund, K. &E1-Sedy, E.（1995）. Automata, repeated games, and noise. Journal of Mathematical Biology,33,703-732.

Nowak, M. A.（2006）. Five rules for the evolution of cooperation. Science, 314（5805）, 1560– 1563.

Panchanathan, K., Boyd, R.（2003）. A tale of two defectors: The importance of standing in the evolution of indirect reciprocity. Journal of Theoretical Biology,224,115-126.

Putnam, R.D.（1993）.Making democracy work: Civic traditions in modern Italy. Princeton University Press.

Rand, D.G. Nowak, M. A.（2013）. Human cooperation. Trends in Cognitive Sciences. 17, 8, 413-425.

Squazzoni, F.（2012）. *Agent-Based Computational Sociology*, UK: Wiley.

Sugden, R.（1986）. The economics of rights, cooperation and welfare. Oxford, UK; Basil Blackwell

Trivers, R. L.（1971）. The evolution of reciprocal altruism. Quarterly Review of Biology, 46（1）, 35-57.

石田亨，鳥居大祐，村上陽平，寺野隆雄（2007）「社会に向き合うエージェントシステム：７．社会シミュレーションと参加型デザイン」，情報処理，48（3），271-277

上野善子, 金城八津子, 植村直子, 畑下博世.（2010）「地域包括支援センターの役割と可能性：高齢者の地域生活とソーシャル・インクルージョン」. 滋賀医科大学看護学ジャーナル, 8（1）, 4-8.

大森純子.（2007）「前期高齢女性の近隣他者との交流関係と健康関連 QOL との関連」. 日本公衆衛生雑誌，9（54），605-614

河田潤一（訳）.（2001）.『哲学する民主主義―伝統と改革の市民公的構造』, NTT 出版

真島理恵，高橋伸幸.（2005a）.「敵の味方は敵？：間接互恵性における二次情報の効果に対する理論的・ 実証的検討」.理論と方法，20，177-195.

真島理恵，高橋伸幸.（2005b）.「間接互恵性の成立：非寛容な選別主義に基づく利他行動の適応的基盤」. 心理学研究，76（5），436-444.

福嶋篤.（2014）「地域在住高齢者における自主参加型体操グループへの参加中止に関連する要因と要因の経年変化」. 日本公衛誌，1

島弘樹，武藤佳恭.（2002）「エージェントベース社会シミュレーションのための人間の行動原則の観測システムに関する提案 シミュレーション世界と現実世界をリンクするための方法論」，情報処理学会研究報告知能と複雑系，127（15），111-117,

松井博史, 大良宏樹，井出野尚, 酒折文武, 高橋英彦, 竹村和久.（2009）「ギビングゲーム状況における協力行動の意思決定」. 日本認知心理学会発表論文集, 日本認知心理学会第 7 回大会, 2-5

白木優馬.（2017）「制御焦点が間接互恵行為に与える影響：価値・コスト の認知的評価の弁別に着目した検討」. 対人社会心理学研究, 17, 1-6

大分大学福祉科学研究センター.（2011）「コミュニティカフェの実態に関する調査結果』.

内閣府.（2016a）高齢社会対策 ＜ http://www8.cao.go.jp/kourei/whitepaper/w2016/gaiyou/28pdf_indexg.html>（2017/04/05 閲覧）

内閣府.（2016b）「平成 28 年版高齢社会白書（概要版）」.＜ https://www8.cao.go.jp/kourei/whitepaper/w-2016/gaiyou/28pdf_indexg.html＞（2018/3/27 閲覧）

長谷川裕紀, 小川正光.（2015）「高齢者の生活様式と地域のコミュニティの支援構造に関する研究」. 愛知教育大学家政教育講座研究紀要, 44, 117-132.

厚生労働省.（2000）「社会的な援護を要する人々に対する社会福祉のあり方に関する検討会」報告書, ＜ https://www.mhlw.go.jp/www1/shingi/s0012/s1208-2_16.html＞（2018/11/20 閲覧）

厚生労働省.（2011）「社会的包摂政策を進めるための基本的考え方」.＜ https://www.mhlw.go.jp/stf/shingi/2r9852000001ngpw-att/2r9852000001ngxn.pdf＞ （2019/02/02 閲覧）

岡崎幹, 赤井研樹, 西野成昭.（2014）「非均質 Image Score に基づくネットワーク形成過程における個体差と戦略模倣の影響分析」. 人工知能学会全国大会論文集, 第 28 回全国大会

金子勝司.（2009）「豪雪過疎地域における冬期の社会参加事業の検討」. 共栄学園短期大学研究紀要, 25, 177-196.

岩原 昭彦.（2010）「大学生による介入が高齢者の生きがいの向上に及ぼす効果」, 人間環境学研究 8（1）, 89-95.

小林重人, 山田広明.（2014）「マイプレイス志向と交流志向が共存するサードプレイス形成モデルの研究：石川県能美市の非常設型「ひょっこりカフェ」を事例として」. 地域活性研究, 5, 3-12

中村久美.（2009）「地域コミュニティとしての「ふれあい・いきいきサロン」の評価」. 日本家政学会誌，60（1），25-37

明田芳久，岡本浩一，奥田秀宇，外山みどり，山口勧.（1994）.「ベーシック現代心理学」，社会心理学，有斐閣，7

大坊郁夫，安藤清志，池田謙一（1989）.『社会心理学パースペクティブ』，誠信書房

高橋 真吾（2013）.「社会システムの研究動向 3－評価・分析手法（1）－モデルの解像度と妥当性評価」. 計測と制御，52（7），582-587

総務省統計局.（2013）.「高齢者の人口」，＜https://www.stat.go.jp/data/topics/topi1131.html＞（2019/02/02 閲覧）

付録 A　ソースコード

```
# -*- coding: utf-8 -*-
"""
Created on Mon Dec 17 22:48:27 2018

@author: Zhang Yue
"""


"""
Created on Wed Oct 10 16:37:46 2018

@author: Zhang Yue

これは本プログラムの本体
説明：直接互恵による「交流ゲーム」と，間接互恵で成り立つコミュニティを表す「包摂ゲーム」.
交流ゲームでは，相手に対する経験スコアと自身の判断基準の差および判断基準に応じた参加コストにより，利得と経験スコアが変わる.
包摂ゲームでは主体はランダムに協力の提供者か受容者となり，提供者は受容者に返報を求めない協力をするか否かを，判断基準と印象スコアから決定する.
協力すると利得が下がるが印象スコアが上がる．何回かのゲームの後，各主体は最大利得主体の判断基準を確率的に模倣する.
"""

import numpy as np
import sys
#from sys import stdout
import argparse
#import matplotlib.pyplot as plt

"定数"
COOP = 1              #ゲーム１とゲーム２で協力する
DEFECT = 0            #ゲーム１で協力しない、とゲーム２でｈ＝０の人が協力しない
DEFECT_h1 = -1    #ゲーム２でｈ＝1の人が協力しない
GO = 1                #ゲーム１に参加する
PASS = 0              #ゲーム１に参加しない

"初期値＆初期化"
num_all = 500              #総エージェント数      !偶数     4
max_step_game1 = 3      #ゲーム１の回す回数
max_step_game2 = 2      #ゲーム２の回す回数
max_gen = 1000            #世代数         00
SEED = 13                 #シード値
```

```
beta = 10              #ゲーム 1 に参加するコストの大きさ
decrease_e = 0.5       #ゲーム 1 で player1 が非協力を選んだ時に、player2 から見る player1 の経験スコア（e）が減少する
increase_e = 0.5       #ゲーム 1 で player1 が協力を選んだ時に、player2 から見る player1 の経験スコア（e）が増加する
delta = 0.05           #e が delta 分減少して、k_median に近づく

decrease_payoff_unit2 = 1    #ゲーム 2 で doner が協力を選んだ時に、doner の payoff が減少する
increase_payoff_unit2 = 1    #ゲーム 2 で doner が協力を選んだ時に、recipient の payoff が増加する
decrease_i = 1               #ゲーム 2 で doner が非協力を選んだ時に、recipient から見る doner の経験スコア（i）が減少する
increase_i = 1               #ゲーム 2 で doner が協力を選んだ時に、recipient から見る doner の経験スコア（i）が増加する
alpha = 0.8                  #前回のゲームの i の（１－alpha）分今回の i になる
adapt_probability = 0.5      #確率 q で k をまねる
k_max = 11                   #k の範囲　k=k_min～k_max　!k_max は奇数
k_min = 1
k_median = (k_max+k_min)/2   #k の中間値

agent_h0_num = 0
go_game1_num_list = []       #各 gen ゲーム 1 に参加する人数のリスト
pass_game1_num_list = []     #各 gen ゲーム 1 に参加しない人数のリスト
num_habu_game1 = 0           #g ゲーム 1 に参加したが、はぶられた人数
h0_doner_game2 = 0           #ゲーム 2 でｈ 0 が doner になった人数
h1_doner_game2 = 0           #ゲーム 2 でｈ 1 が doner になった人数
h0_recipient_game2 = 0       #ゲーム 2 でｈ 0 が recipient になった人数
h1_recipient_game2 = 0       #ゲーム 2 でｈ1 が recipient になった人数

"協力関係を記録する二次元配列"
#ゲーム 1
coop_mtx_game1 = np.zeros((num_all,num_all))          #ゲーム 1 で　協力（交流が楽しかった・うまくいった）
defect_mtx_game1 = np.zeros((num_all,num_all))        #ゲーム 1 で　非協力（交流が楽しくなかった・うまくいかなかった）
#ゲーム 2
coop_mtx_game2 = np.zeros((num_all,num_all))          #ゲーム 2 で　協力
defect_h0_mtx_game2 = np.zeros((num_all,num_all))     #ゲーム 2 で　h=0 の人が　非協力
defect_h1_mtx_game2 = np.zeros((num_all,num_all))     #ゲーム 2 で　h=1 の人が　非協力
#全体
coop_mtx_all = np.zeros((num_all,num_all))            #全体の　協力
defect_mtx_all = np.zeros((num_all,num_all))          #全体の　非協力
met_mtx_all = np.zeros((num_all,num_all))             #全体の　協力と非協力　つまり、会ったことあるかどうか

A = 1 #0.1**15
memory_gen_num = 2
b = 1.5
constant_cost_go_game1 = 0.5  #game1 の参加コスト、全員同じ一定の数値　（0.5 にすると、今までのやつと一緒）

"""
```

```
Agent クラスを定義
"""
class Agent:

    def __init__( self, name, k, h ):
        self.name = name
        self.h = h   #弱者かどうか h ＝ 1 は弱者

        self.k = k   #doner が他のエージェントを評価する基準
        self.temp_k = k      #max_step が終わった後に k を模倣するので、それまでに temp_k に一時的に預かる
        self.adpt_k = k
        self.actually_imitate = 0

        self.cost_attend_game1 = self.costfunc()    #ゲーム 1 に参加する確率

        self.payoff = 0    #エイジェントの利得
        self.met_partner_payoff_list = []    #出会ったエージェントの payoff のリスト

        #self.maxpayoff_index_list = []    #met_partner_payoff_list の中で一番高い payoff のエージェントの位置をリストに入れる
                                           #（同じ一番高い payoff は何人もいるかもしれないので、リストにしている）
        #self.maxpayoff_index = 0    #maxpayoff_index_list の中から 1 つの maxpayoff の位置を取る
                                    #（この位置は　met_partner_list_game1　で記録された出会ったエージェントの位置と一致する
        #self.maxpayoff_agent = 0    #出会ったエージェントの中で一番 payoff が高いエージェントの name を取り出す

        self.memory_list = np.zeros((memory_gen_num,(max_step_game1+max_step_game2) ))
#k を真似できるエージェントのリスト
        self.memory_list_flatten = []       #memory_list を 1 次元にしたリスト
        self.memory_payoff_list = []     #記録できる範囲内の payoff のリスト

        self.minpayoff_index_list = []    #memory_list の中で一番低い payoff のエージェントの位置をリストに入れる
                                           #（同じ一番低い payoff は何人もいるかもしれないので、リストにしている）
        self.minpayoff_index = 0    #minpayoff_index_list の中から 1 つの minpayoff の位置を取る
                                    #（この位置は　memory　で記録された出会ったエージェントの位置と一致する
        self.minpayoff_name = 0    #出会ったエージェントの中で一番 payoff が低いエージェントを探し出す


        self.partner_e = np.full( num_all, k_median )  #エージェントから見たパートナーの評判
        self.partner_e_gen = np.zeros(num_all) #どの世代の e なのかを記録するリスト
        self.own_e_list_game1 = []       #あるエージェントに対する game1 で会ったエージェントが持つそのエージェントの e をリストに集める
        self.own_e_list_all = []          #                    game1 と game2 で
```

```
        self.image_score = k_median        #エージェントのイメージスコア、@ゲーム２

        self.own_go_game1_num = 0          #memory_gen_num 回の世代の中で、ゲーム１に行った回数
        self.own_go_game1_num_list = np.zeros(memory_gen_num)  #memory_gen_num 回の各世代においてゲーム１に行った回数
        self.know_partneri_prob = self.partneri_func() #ゲーム２で　相手の imagescore を知る確率、確率を計算する関数　　new!

        self.g_or_p = 0                    #ゲーム 1 go or pass
        self.c_or_d = 0                    #cooperate or defect

        self.met_partner_list_game1 = []   #エージェントがゲーム１で会った人のリスト
                                           #imitate_maxpayoff_partner のところで重複無しのリストになる
        self.met_partner_name_list_all = []          #エージェントがあった全部の人名前のリスト　（k をまねる時に使う）


    """リセット"""
    "新しい gen が始まったら、以下のものが初期値になる"
    def reset(self):
        self.actually_imitate = 0
        self.payoff = 0
        self.met_partner_payoff_list = []
        self.minpayoff_index_list = []
        self.minpayoff_index = 0
        self.minpayoff_name = 0
        self.own_e_list_game1 = []
        self.own_e_list_all = []
        self.g_or_p = 0
        self.c_or_d = 0
        self.met_partner_list_game1 = []
        self.met_partner_name_list_all = []
        self.memory_list_flatten = []
        self.memory_payoff_list = []


    """e が忘れられる　k_median に徐々に近づく"""
    def decay_e( self ):
        #自分の partner_e リストを順番に見る
        for partner_name in range(0, num_all):
            #自分から見るほかのエージェントの e が k_median を超えたかどうかを判断
            if self.partner_e[partner_name] > k_median:
                #超えたら －delta
                self.partner_e[partner_name] -= delta
                #delta を引いた後、k_median より小さいかどうかを判断
                if self.partner_e[partner_name] < k_median:
                    #小さいだったら、k_median になる（k_median 以下にならないように）
                    self.partner_e[partner_name] = k_median
            elif self.partner_e[partner_name] < k_median:
```

```
                    #こえてなかったら、+delta
                    self.partner_e[partner_name] += delta
                    #+delta 後、k_median を超えたら
                    if self.partner_e[partner_name] > k_median:
                        #k_median になる（k_median 以上にならないように）
                        self.partner_e[partner_name] = k_median

    """ ゲーム 1 に参加するコスト（＝ゲーム 1 に参加する確率）を計算 """
    def costfunc( self ):
        #return 1-(1/(k_max+1))*self.k
        #return 1-(k_max/(k_max**2+1))*self.k
        return -(k_max*self.k-k_max**2-1) / (k_max**2-k_max+2)

    """ゲーム 1 ＜参加するかどうかを判断＞"""
    def go_or_pass_game1( self ):

        ran = np.random.rand()

        if( self.cost_attend_game1 >= ran ):
            #print(gens, self.name, self.k, "GOPASS: ", self.own_go_game1_num_list[0],
self.own_go_game1_num_list[1])
            self.g_or_p = GO
            self.own_go_game1_num_list[ int(gens % memory_gen_num) ] +=1
            #print("    ", self.g_or_p, self.own_go_game1_num_list[0],
self.own_go_game1_num_list[1])

        else:
            self.g_or_p = PASS
            self.memory_list[ int(gens % memory_gen_num) ][ step_game1 ] = -1



    """ゲーム 1＜うまくいったかどうか＞"""
    def coop_or_defect_game1( self, partner ):

        global coop_mtx_game1, defect_mtx_game1

        self.memory_list[ gens % memory_gen_num ][ step_game1 ] = partner.name
        #self.memory_list.append(partner)
        if( self.k <= self.partner_e[partner.name] ):
            self.c_or_d = COOP
            #coop_mtx_game1 二次元配列に回数＋ 1
            coop_mtx_game1[self.name][partner.name] += 1
        else:
            self.c_or_d = DEFECT
            defect_mtx_game1[self.name][partner.name] += 1

    """ゲーム 1 とゲーム 2 の中で e を変化させるところ"""
    def change_e( self, partner_player):

        self.partner_e_gen[partner_player.name] = gens
        #partner_player.partner_e_gen[self.name] = gens
```

```
        for partnernames in range (0, num_all):
            if self.partner_e_gen[partnernames] - gens + 1> memory_gen_num:
                self.partner_e[partnernames] = k_median

        #協力したら
        if ( self.c_or_d == COOP ):
            #player_b から見る player_a の e が上がる
            partner_player.partner_e[ self.name ] += increase_e
            #e は k_max に越えないように
            if ( partner_player.partner_e[ self.name ] > k_max ):
                partner_player.partner_e[ self.name ] = k_max
        elif ( self.c_or_d == DEFECT ):
            partner_player.partner_e[ self.name ] -= decrease_e
            if ( partner_player.partner_e[ self.name ] < k_min ):
                partner_player.partner_e[ self.name ] = k_min

    """ゲーム 1 の中で payoff を変化させるところ"""
    def change_payoff_game1( self, partner_player):
        self.payoff ¥
            += A*(partner_player.partner_e[self.name] - self.k ¥
                - beta*constant_cost_go_game1)                          #change
beta*self.cost_attend_game1 to a Constant value
                #- beta*self.cost_attend_game1)

    """ゲーム 2 で相手の i を知る確率を計算する"""
    def partneri_func( self ):
        #print("OWNGO", self.own_go_game1_num)
        if max_step_game1 != 0:
            owngg1n = b**(self.own_go_game1_num)
            allgg1n = b**(max_step_game1*memory_gen_num)
            return (   owngg1n- 1 ) / (   allgg1n- 1 )
        else:
            return 0.5

    """ゲーム 2＜協力するかどうかを判断＞"""
    def coop_or_defect_game2( self, recipient ):
        global coop_mtx_game2, defect_h0_mtx_game2, defect_h1_mtx_game2, ¥
                h0_doner_game2, h1_doner_game2, h0_recipient_game2, h1_recipient_game2
        self.memory_list[ gens % memory_gen_num ][ max_step_game1 + step_game2 ] =
recipient.name
        "相手のイメージスコアを知るかどうかを判断"
        ran = np.random.rand()
        if( self.know_partneri_prob >= ran ):
            partner_image_score = recipient.image_score
        else:
            partner_image_score = k_median
        #print("Prob", self.know_partneri_prob, ran, partner_image_score, recipient.image_score)

        "協力するかどうかを判断"
        if self.h == 0 :
            h0_doner_game2 += 1
            if self.k <= partner_image_score :
```

```
                self.c_or_d = COOP
                coop_mtx_game2[self.name][recipient.name] += 1
            elif self.k > recipient.image_score：
                self.c_or_d = DEFECT
                defect_h0_mtx_game2[self.name][recipient.name] += 1
        elif self.h == 1：
            h1_doner_game2 += 1
            self.c_or_d = DEFECT_h1
            defect_h1_mtx_game2[self.name][recipient.name] += 1

        if recipient.h == 0:
            h0_recipient_game2 += 1
            #print(h0_recipient_game2)
        elif recipient.h == 1:
            h1_recipient_game2 += 1
            #print(h1_recipient_game2)

    """ゲーム２の中で image_score を変化させるところ"""
    def change_i_payoff_game2( self, recipient ):
        if ( self.c_or_d == COOP ):
            #doner と recipient の payoff を変更
            self.payoff -= decrease_payoff_unit2
            recipient.payoff += increase_payoff_unit2
            self.image_score += increase_i
            if self.image_score > k_max：
                self.image_score = k_max
        #doner が協力しないを選んだ　＆＆ h=0 の場合
        elif ( self.c_or_d == DEFECT ):
            self.image_score -= decrease_i
            if self.image_score < k_min：
                self.image_score = k_min

    """e を i に変換する"""
    def calculate_image_score( self ):
        #リストの中身の重複をなくす：list(set(リスト名))
        self.met_partner_list_game1 = list(set( self.met_partner_list_game1 ))
        if len( self.met_partner_list_game1 ) > 0:
            for met_agent in self.met_partner_list_game1:
                #出会ったエージェントの自分に対する partner_e をリストアップ
                self.own_e_list_game1.append(met_agent.partner_e[self.name])
            #イメージスコアに変換計算
            self.image_score = ((1 - alpha)*self.image_score) ¥
                                + (alpha/len(self.own_e_list_game1)*sum(self.own_e_list_game1))


    def create_own_e_list_all(self):
        for met_agent in range( 0,num_all ):
            if met_mtx_all[self.name][met_agent] != 0:
                self.own_e_list_all.append(agent_list[met_agent].partner_e[self.name])


    """
```

```
        K の模倣
        出会った相手の中の最小 payoff を持つエージェントの k と別の k を持つ
        2018/08/24
    """
    def imitate_minpayoff_partner( self , agent_list):
        self.memory_list_flatten = self.memory_list.flatten()
        for memo_n in range(0, len(self.memory_list_flatten)):
            memo_name = int(self.memory_list_flatten[memo_n])
            if memo_name != 0:
                memo_p = agent_list[memo_name].payoff
                self.memory_payoff_list.append(memo_p)
            else:
                self.memory_payoff_list.append(0)
        "あったことある人の中から一番 payoff 低い人を探し出す"
        if sum( self.memory_list_flatten ) > 0:

            #一番低い payoff のエージェントの位置をリストアップ
            self.minpayoff_index_list =[y for y, x in enumerate(self.memory_payoff_list)
                                        if x == min(self.memory_payoff_list)]
            #複数いるかもしれないので、1 つだけをランダムに取る
            self.minpayoff_index = np.random.choice(self.minpayoff_index_list)
            #一番低い payoff を持つエージェントを探す
            self.minpayoff_name = self.memory_list_flatten[self.minpayoff_index]
            #print(self.name, "before: k", self.k, "temp_k", self.temp_k, "adpt_k", self.adpt_k)
            "一番 payoff 低いエージェントの k と同じか自分の方が下だったら、他のあったことあ
る人の k をランダムに真似する"
            if (agent_list[int(self.minpayoff_name)].payoff >= self.payoff ):
                self.temp_k = agent_list[int(np.random.choice(self.memory_list_flatten))].k
                self.actually_imitate = 1
            """
            if (agent_list[int(self.minpayoff_name)].payoff >= self.payoff ):
                self.adpt_k = agent_list[int(np.random.choice(self.memory_list_flatten))].k
                self.actually_imitate = 1
                if self.adpt_k > self.k:
                    self.temp_k += 1
                    if self.temp_k > k_max:
                        self.temp_k = k_max
                elif self.adpt_k < self.k:
                    self.temp_k -= 1
                    if self.temp_k < k_min:
                        self.temp_k = k_min
            #print(self.name, "after: k", self.k, "temp_k", self.temp_k, "adpt_k", self.adpt_k)
            """

"""
リセット

"""
def reset_lastgen(agent_list):
    #agent をリセットする
    for agent in agent_list:
        agent.reset()
```

```
def reset_observer():
    #グローバル変数たち　と　グローバルリストたちをリセットする
    global coop_mtx_game1, defect_mtx_game1, ¥
    coop_mtx_game2, defect_h0_mtx_game2, defect_h1_mtx_game2, ¥
    coop_mtx_all, defect_mtx_all, met_mtx_all,¥
    go_game1_num_list, pass_game1_num_list, ¥
    num_habu_game1, h0_doner_game2, h1_doner_game2, ¥
    h0_recipient_game2, h1_recipient_game2

    #配列を全部０にリセットする　（https://note.nkmk.me/python-numpy-zeros-ones-full/）
    coop_mtx_game1 = np.zeros((num_all,num_all))
    defect_mtx_game1 = np.zeros((num_all,num_all))
    coop_mtx_game2 = np.zeros((num_all,num_all))
    defect_h0_mtx_game2 = np.zeros((num_all,num_all))
    defect_h1_mtx_game2 = np.zeros((num_all,num_all))
    coop_mtx_all = np.zeros((num_all,num_all))
    defect_mtx_all = np.zeros((num_all,num_all))
    met_mtx_all = np.zeros((num_all,num_all))
    go_game1_num_list = []
    pass_game1_num_list = []
    num_habu_game1 = 0
    h0_doner_game2 = 0
    h1_doner_game2 = 0
    h0_recipient_game2 = 0
    h1_recipient_game2 = 0

"""
game1

"""
def game1( agent_list ):
    global go_game1_num_list, pass_game1_num_list

    agent_list_go_game1 = []
    num_agent_go_game1 = 0
    agent_list_pass_game1 = []
    num_agent_pass_game1 = 0

    "ゲーム１に参加するエージェントをリストアップ"
    for agent in agent_list:
        # 忘れられる、e が少しずつ減る
        agent.decay_e()
        #ゲーム１に行くかどうかを決める
        agent.go_or_pass_game1()
        if agent.g_or_p == GO:
            agent_list_go_game1.append(agent)
        else：
            agent_list_pass_game1.append(agent)

    num_agent_go_game1 = len(agent_list_go_game1)
    go_game1_num_list.append(num_agent_go_game1)
```

```
    num_agent_pass_game1 = len(agent_list_pass_game1)
    pass_game1_num_list.append(num_agent_pass_game1)

    #
    # player1 と player2 をランダムにペアリング
    # 関数からの戻り値として，それぞれのリストを受け取る
    #
    if( num_agent_go_game1 > 0):
        list_player1_game1 = []
        list_player2_game1 = []
        num_player1_game1, list_player1_game1, list_player2_game1 ¥
            = paring_game1( agent_list_go_game1, num_agent_go_game1 )

        for player_i in range( num_player1_game1 ):
            "player1 組と player2 組それぞれ 1 つを取り出す"
            cur_player1 = list_player1_game1[player_i]
            cur_player2 = list_player2_game1[player_i]
            "出会ったエージェントを met_partner_list_game1 に記録する"
            cur_player1.met_partner_list_game1.append( cur_player2 )
            cur_player2.met_partner_list_game1.append( cur_player1 )

            "payoff の変化"
            cur_player1.change_payoff_game1(cur_player2)
            cur_player2.change_payoff_game1(cur_player1)

            "COOP と DEFECT の場合、具体的に"
            cur_player1.coop_or_defect_game1( cur_player2 )
            cur_player2.coop_or_defect_game1( cur_player1 )

            "e の変化"
            cur_player1.change_e( cur_player2 )
            cur_player2.change_e( cur_player1 )

"""
ペアリング　ゲーム 1

"""
def paring_game1( agent_list_go_game1, num_agent_go_game1 ):
    global num_habu_game1

    #２で割って num_player1_game1 に、残りを num_player2_game1 にする
    ##num_agent_go_game1 が奇数でも、int(num_agent_go_game1/2)したら、
num_player1_game1 が偶数になる
    num_player1_game1 = int(num_agent_go_game1/2)
    #ゲーム 1 に参加したけど、奇数の時はぶられた人の数
    num_habu_game1 += num_agent_go_game1%2
    #重複なく要素をランダムに抽出し２組に分ける <choice-ランダムに抽出（元のリスト、抽出す
る数、重複無し）>
    list_all_game1 = np.random.choice( agent_list_go_game1, num_agent_go_game1,
replace=False )
    #０から num_player1_game1 個までは player1 組
    list_player1_game1 = list_all_game1[0:num_player1_game1]
```

```
    #その残りの数は player2 組
    list_player2_game1 = list_all_game1[num_player1_game1:num_player1_game1*2]

    #それぞれを戻り値として返す
    return num_player1_game1, list_player1_game1, list_player2_game1

"""
ゲーム 1 の参加回数を集計し，i を知る確率を再計算
"""
def update_game1_gopass( agent_list ):

    for agent in agent_list:
        agent.own_go_game1_num = sum(agent.own_go_game1_num_list)
        agent.know_partneri_prob = agent.partneri_func()
        #print(gens, agent.name, "UPDATE_GOPASS  ", agent.own_go_game1_num,
agent.own_go_game1_num_list[0], agent.own_go_game1_num_list[1], agent.know_partneri_prob)
        agent.own_go_game1_num_list[ (gens+1) % memory_gen_num ] = 0 #前の世代の参加回数
をリセット
        #print(gens, agent.name, "UPDATE_GOPASS  ", agent.own_go_game1_num,
agent.own_go_game1_num_list[0], agent.own_go_game1_num_list[1], agent.know_partneri_prob)

"""
ゲーム 1 の e をゲーム 2 の i に変換する

"""
def transform_e_to_i( agent_list ):
    for agent in agent_list:
        if( len(agent.met_partner_list_game1) >0 ):
            agent.calculate_image_score()

"""
game2

"""
def game2( agent_list ):
    #
    # doner と recipient をランダムにペアリング
    # 関数からの戻り値として，それぞれのリストを受け取る
    #
    list_doner_game2 = []
    list_recipient_game2 = []
    num_doner_game2, list_doner_game2, list_recipient_game2 = paring_game2( agent_list )

    for player_i in range( num_doner_game2 ):
        """doner 組と recipient 組それぞれ 1 つを取り出す"""
        cur_doner = list_doner_game2[player_i]
        cur_recipient = list_recipient_game2[player_i]
        """COOP と DEFECT の場合、具体的に"""
        cur_doner.coop_or_defect_game2( cur_recipient )
        cur_doner.change_i_payoff_game2(cur_recipient)
        cur_doner.change_e( cur_recipient )
```

```
"""
ペアリング　ゲーム 2

"""
def paring_game2( agent_list ):
    #num_all の半分が doner 半分が recipient
    num_doner_game2　=　int(num_all/2)
    #重複なく要素をランダムに抽出し 2 組に分ける <choice-ランダムに抽出（元のリスト、抽出する数、重複無し）>
    list_all_game2 = np.random.choice( agent_list, num_all, replace=False )
    # 0 から num_player1 個までは doner 組
    list_doner_game2 = list_all_game2[0:num_doner_game2]
    #その残りの数は recipient 組
    list_recipient_game2 = list_all_game2[num_doner_game2:num_all]
    #それぞれを戻り値として返す
    return num_doner_game2, list_doner_game2, list_recipient_game2



"""
distribute_payoff_e_i_k          #20181024

"""
def calculate_p_e_i_k(agent_list):

    k_list = []
    #count_k_list = []
    frequency_k_list =[]
    payoff_list = []
    ave_payoff = 0
    max_payoff = 0
    min_payoff = 0
    e_list = []
    ave_e = k_median
    max_e = k_median
    min_e = k_median
    i_list = []
    ave_i = k_median
    max_i = k_median
    min_i = k_median
    num_go_game1 = 0
    num_pass_game1 = 0
    #count_adapted_k = 0

    for agent in agent_list:
        "own_e_list_all を集計"
        agent.create_own_e_list_all()
        "payoff に関する計算"
        payoff_list.append(agent.payoff)
        "e に関する計算"
        if sum(agent.own_e_list_all) > 0:
```

```
                e_list.append(sum(agent.own_e_list_all)/len(agent.own_e_list_all))
            "i に関する計算"
            i_list.append(agent.image_score)
            "k の分布"
            k_list.append(agent.k)

        #"e に関する計算"
        if len(e_list) > 0:
            ave_e = sum(e_list)/len(e_list)
            max_e = max(e_list)
            min_e = min(e_list)
        #payoff に関する計算
        ave_payoff = sum(payoff_list)/num_all
        max_payoff = max(payoff_list)
        min_payoff = min(payoff_list)
        #"i に関する計算"
        ave_i = sum(i_list)/num_all
        max_i = max(i_list)
        min_i = min(i_list)
        #"k の分布"
        for ks in range( k_min, k_max+1 ):
            frequency_k_list.append( k_list.count(ks)/num_all )
        #ゲーム 1 に参加するとしない人の数
        num_go_game1 = sum(go_game1_num_list)
        num_pass_game1 = sum(pass_game1_num_list)

        return ave_payoff, max_payoff, min_payoff, ¥
                    ave_e, max_e, min_e, ¥
                    ave_i, max_i, min_i, ¥
                    frequency_k_list, ¥
                    num_go_game1, num_pass_game1

"""
二次元配列から rate を算出する

"""
def allmtx(agent_list):

        global coop_mtx_all, defect_mtx_all, met_mtx_all

        "マトリックスに記録する"
        coop_mtx_all = coop_mtx_game1 + coop_mtx_game2
        defect_mtx_all = defect_mtx_game1 + defect_h0_mtx_game2 + defect_h1_mtx_game2
        met_mtx_all = coop_mtx_all + defect_mtx_all

        coop_count_all = np.sum(coop_mtx_all)
        defect_count_all = np.sum(defect_mtx_all)
        met_count_all = np.sum(met_mtx_all)

        return coop_count_all, defect_count_all, met_count_all

def rate_game1(agent_list):
```

```
if max_step_game1 != 0:
    num_games1 = max_step_game1*num_all
    pass_rate_game1 = sum(pass_game1_num_list)/num_games1

    #
    #ゲーム 1 の COOP と DEFECT の数とレートを計算する
    #
    coop_count_game1 = np.sum(coop_mtx_game1)
    coop_rate_game1 = coop_count_game1/num_games1
    defect_count_game1 = np.sum(defect_mtx_game1)
    defect_rate_game1 = defect_count_game1/num_games1

    #
    # game1 の cooper, defecter, cooper, dectee の数・率
    #
    #協力する人-cooper と協力された人ーcoopee の rate を計算する
    cooper_h0_game1 = 0
    cooper_h1_game1 = 0
    defecter_h0_game1 = 0
    defecter_h1_game1 = 0

    coopee_h0_game1 = 0
    coopee_h1_game1 = 0
    defectee_h0_game1 = 0
    defectee_h1_game1 = 0

    for agent_name in range(num_all) :
        # Cooper, Defcter
        if (sum(coop_mtx_game1[agent_name, :]) != 0) :
            if (agent_list[agent_name].h == 0):
                cooper_h0_game1 += sum(coop_mtx_game1[agent_name, :])
            elif (agent_list[agent_name].h == 1):
                cooper_h1_game1 += sum(coop_mtx_game1[agent_name, :])
        if (sum(defect_mtx_game1[agent_name, :]) != 0):
            if (agent_list[agent_name].h == 0):
                defecter_h0_game1 += sum(defect_mtx_game1[agent_name, :])
            elif (agent_list[agent_name].h == 1):
                defecter_h1_game1 += sum(defect_mtx_game1[agent_name, :])

        # Coopee, Defctee
        if (sum(coop_mtx_game1[:, agent_name]) != 0) :
            if (agent_list[agent_name].h == 0):
                coopee_h0_game1 += sum(coop_mtx_game1[:, agent_name])
            elif ( agent_list[agent_name].h == 1):
                coopee_h1_game1 += sum(coop_mtx_game1[:, agent_name])
        if (sum(defect_mtx_game1[:, agent_name]) != 0):
            if (agent_list[agent_name].h == 0):
                defectee_h0_game1 += sum(defect_mtx_game1[:, agent_name])
            elif (agent_list[agent_name].h == 1):
                defectee_h1_game1 += sum(defect_mtx_game1[:, agent_name])
```

```
        cooper_h0_rate_game1 = cooper_h0_game1/num_games1
        cooper_h1_rate_game1 = cooper_h1_game1/num_games1
        defecter_h0_rate_game1 = defecter_h0_game1/num_games1
        defecter_h1_rate_game1 = defecter_h1_game1/num_games1

        coopee_h0_rate_game1 = coopee_h0_game1/num_games1
        coopee_h1_rate_game1 = coopee_h1_game1/num_games1
        defectee_h0_rate_game1 = defectee_h0_game1/num_games1
        defectee_h1_rate_game1 = defectee_h1_game1/num_games1

        return pass_rate_game1, coop_rate_game1, defect_rate_game1, ¥
                cooper_h0_rate_game1, cooper_h1_rate_game1, defecter_h0_rate_game1,
defecter_h1_rate_game1,¥
                coopee_h0_rate_game1, coopee_h1_rate_game1, defectee_h0_rate_game1,
defectee_h1_rate_game1

    else:
        pass_rate_game1=0
        coop_rate_game1=0
        defect_rate_game1=0
        cooper_h0_rate_game1=0
        cooper_h1_rate_game1=0
        defecter_h0_rate_game1=0
        defecter_h1_rate_game1=0
        coopee_h0_rate_game1=0
        coopee_h1_rate_game1=0
        defectee_h0_rate_game1=0
        defectee_h1_rate_game1=0

        return pass_rate_game1, coop_rate_game1, defect_rate_game1, ¥
                cooper_h0_rate_game1, cooper_h1_rate_game1, defecter_h0_rate_game1,
defecter_h1_rate_game1,¥
                coopee_h0_rate_game1, coopee_h1_rate_game1, defectee_h0_rate_game1,
defectee_h1_rate_game1
    #""" """
def rate_game2(agent_list):
    if max_step_game2 != 0:
        #
        #ゲーム 1 の COOP と DEFECT の数とレートを計算する
        #
        coop_count_game2 = np.sum(coop_mtx_game2)
        coop_rate_game2 = coop_count_game2/h0_doner_game2
        defect_h0_count_game2 = np.sum(defect_h0_mtx_game2)
        defect_h0_rate_game2 = defect_h0_count_game2/h0_doner_game2
        defect_h1_count_game2 = np.sum(defect_h1_mtx_game2)
        defect_h1_rate_game2 = defect_h1_count_game2/h0_doner_game2
        #
        # game2 の cooper, defecter, cooper, dectee の数・率
        #
        cooper_h0_game2 = 0
        cooper_h1_game2 = 0
        defecter_h0_game2 = 0
```

```
defecter_h1_game2 = 0

coopee_h0_game2 = 0
coopee_h1_game2 = 0
defectee_h0_byh0_game2 = 0
defectee_h1_byh0_game2 = 0
defectee_h0_byh1_game2 = 0
defectee_h1_byh1_game2 = 0

for agent_name in range(num_all) :
    # Cooper, Defecter
    if (sum(coop_mtx_game2[agent_name, :]) != 0) :
        if (agent_list[agent_name].h == 0):
            cooper_h0_game2 += sum(coop_mtx_game2[agent_name, :])
        elif ( agent_list[agent_name].h == 1):
            cooper_h1_game2 += sum(coop_mtx_game2[agent_name, :])
    if (sum(defect_h0_mtx_game2[agent_name, :]) != 0):
        defecter_h0_game2 += sum(defect_h0_mtx_game2[agent_name, :])
    if (sum(defect_h1_mtx_game2[:, agent_name]) != 0):
        defecter_h1_game2 += sum(defect_h1_mtx_game2[agent_name, :])

    # Coopee, Defectee
    if (sum(coop_mtx_game2[:, agent_name]) != 0) :
        if (agent_list[agent_name].h == 0):
            coopee_h0_game2 += sum(coop_mtx_game2[:, agent_name])
        elif ( agent_list[agent_name].h == 1):
            coopee_h1_game2 += sum(coop_mtx_game2[:, agent_name])
    if (sum(defect_h0_mtx_game2[:, agent_name]) != 0):
        if (agent_list[agent_name].h == 0):
            defectee_h0_byh0_game2 += sum(defect_h0_mtx_game2[:, agent_name])
        elif (agent_list[agent_name].h == 1):
            defectee_h1_byh0_game2 += sum(defect_h0_mtx_game2[:, agent_name])
    if (sum(defect_h1_mtx_game2[:, agent_name]) != 0):
        if (agent_list[agent_name].h == 0):
            defectee_h0_byh1_game2 += sum(defect_h1_mtx_game2[:, agent_name])
        elif (agent_list[agent_name].h == 1):
            defectee_h1_byh1_game2 += sum(defect_h1_mtx_game2[:, agent_name])

num_games2 = max_step_game2*num_all/2

cooper_h0_rate_game2_parall = cooper_h0_game2/num_games2
#if h0_doner_game2 != 0:
cooper_h0_rate_game2 = cooper_h0_game2/h0_doner_game2
defecter_h0_rate_game2 = defecter_h0_game2/h0_doner_game2
#else:
#       cooper_h0_rate_game2 = 0
#       defecter_h0_rate_game2 = 0
# if h1_recipient_game2 != 0:
coopee_h1_rate_game2 = coopee_h1_game2/h1_recipient_game2
defectee_h1_byh0_rate_game2 = defectee_h0_byh0_game2/h1_recipient_game2
defectee_h1_byh1_rate_game2 = defectee_h1_byh1_game2/h1_recipient_game2
#else:
```

```
        #       coopee_h1_rate_game2 = 0
        #       defectee_h1_byh0_rate_game2 = 0
        #       defectee_h1_byh1_rate_game2 = 0

        cooper_h1_rate_game2_parall = cooper_h1_game2/num_games2
        cooper_h1_rate_game2 = cooper_h1_game2/h1_doner_game2

        defecter_h0_rate_game2_parall = defecter_h0_game2/num_games2

        defecter_h1_rate_game2_parall = defecter_h1_game2/num_games2
        defecter_h1_rate_game2 = defecter_h1_game2/h1_doner_game2

        coopee_h0_rate_game2_parall = coopee_h0_game2/num_games2
        coopee_h0_rate_game2 = coopee_h0_game2/h0_recipient_game2
        coopee_h1_rate_game2_parall = coopee_h1_game2/num_games2


        defectee_h0_byh0_rate_game2_parall = defectee_h0_byh0_game2/num_games2
        defectee_h0_byh0_rate_game2 = defectee_h0_byh0_game2/h0_recipient_game2
        defectee_h1_byh0_rate_game2_parall = defectee_h1_byh0_game2/num_games2


        defectee_h0_byh1_rate_game2_parall = defectee_h0_byh1_game2/num_games2
        defectee_h0_byh1_rate_game2 = defectee_h0_byh1_game2/h0_recipient_game2
        defectee_h1_byh1_rate_game2_parall = defectee_h1_byh1_game2/num_games2



        return coop_rate_game2, defect_h0_rate_game2, defect_h1_rate_game2, ¥
                cooper_h0_rate_game2_parall, cooper_h0_rate_game2,
cooper_h1_rate_game2_parall, cooper_h1_rate_game2, ¥
                defecter_h0_rate_game2_parall, defecter_h0_rate_game2,
defecter_h1_rate_game2_parall, defecter_h1_rate_game2, ¥
                coopee_h0_rate_game2_parall, coopee_h0_rate_game2,
coopee_h1_rate_game2_parall, coopee_h1_rate_game2, ¥
                defectee_h0_byh0_rate_game2_parall, defectee_h0_byh0_rate_game2, ¥
                defectee_h1_byh0_rate_game2_parall, defectee_h1_byh0_rate_game2, ¥
                defectee_h0_byh1_rate_game2_parall, defectee_h0_byh1_rate_game2, ¥
                defectee_h1_byh1_rate_game2_parall, defectee_h1_byh1_rate_game2
    else:
        coop_rate_game2=0
        defect_h0_rate_game2=0
        defect_h1_rate_game2=0
        cooper_h0_rate_game2_parall=0
        cooper_h0_rate_game2=0
        cooper_h1_rate_game2_parall=0
        cooper_h1_rate_game2=0
        defecter_h0_rate_game2_parall=0
        defecter_h0_rate_game2=0
        defecter_h1_rate_game2_parall=0
        defecter_h1_rate_game2=0
        coopee_h0_rate_game2_parall=0
```

```
        coopee_h0_rate_game2=0
        coopee_h1_rate_game2_parall=0
        coopee_h1_rate_game2=0
        defectee_h0_byh0_rate_game2_parall=0
        defectee_h0_byh0_rate_game2=0
        defectee_h1_byh0_rate_game2_parall=0
        defectee_h1_byh0_rate_game2=0
        defectee_h0_byh1_rate_game2_parall=0
        defectee_h0_byh1_rate_game2=0
        defectee_h1_byh1_rate_game2_parall=0
        defectee_h1_byh1_rate_game2=0

        return coop_rate_game2, defect_h0_rate_game2, defect_h1_rate_game2, ¥
                cooper_h0_rate_game2_parall, cooper_h0_rate_game2,
cooper_h1_rate_game2_parall, cooper_h1_rate_game2, ¥
                defecter_h0_rate_game2_parall, defecter_h0_rate_game2,
defecter_h1_rate_game2_parall, defecter_h1_rate_game2, ¥
                coopee_h0_rate_game2_parall, coopee_h0_rate_game2,
coopee_h1_rate_game2_parall, coopee_h1_rate_game2, ¥
                defectee_h0_byh0_rate_game2_parall, defectee_h0_byh0_rate_game2, ¥
                defectee_h1_byh0_rate_game2_parall, defectee_h1_byh0_rate_game2, ¥
                defectee_h0_byh1_rate_game2_parall, defectee_h0_byh1_rate_game2, ¥
                defectee_h1_byh1_rate_game2_parall, defectee_h1_byh1_rate_game2
"""
k の模倣

"""
def adaptation_k_game1cost(agent_list):
    "あったことあるエージェントの中に payoff が一番高い人の k をまねる"
    for agent in agent_list:

        #adapt_probability の確率で k を模倣できる
        p = np.random.rand()
        if   p <= adapt_probability:
            agent.imitate_minpayoff_partner(agent_list)
        # k を模倣する
        if agent.actually_imitate == 1:
            agent.k = agent.temp_k
            #エージェントがゲーム 1 に参加する確率
            agent.cost_attend_game1 = agent.costfunc()

"""
初期化

"""
"agent_list の中で agent を num_all 個初期生成する    Agent( name ,k ,h )"
def initalize(num_all):
    global agent_h0_num

    agent_list = []

    for name in range( num_all ):
```

```
            # Agent( ID, k, h )
            agent_list.append(Agent(name, np.random.randint(k_min,k_max+1),
np.random.randint(0,2)))


        #h = 0 の人数を数える
        for agent in agent_list:
            if agent.h == 0:
                agent_h0_num += 1
        #print("agent_h0_num", agent_h0_num)

        return agent_list


"""
コマンドライン引数
20181009

"""
def analyze_options():

        parser = argparse.ArgumentParser( description='game')

        parser.add_argument('-SEED', required=True, type=int,
            help="parameter SEED")
        parser.add_argument('-beta', required=True, type=float,
            help="parameter beta")
        parser.add_argument('-o', '-fo', dest='fo', #nargs=1,
            type=argparse.FileType('w'), default=sys.stdout,
            help="The output file [Standard out]")
            # For using FileType, the option "narps" should not be diven
            # in order to avoid that an argment is stored as a list
        """
        parser.add_argument('-oo', '-foo', dest='foo', #nargs=1,
            type=argparse.FileType('w'), default=sys.stdout,
            help="The output file [Standard out]")
            # For using FileType, the option "narps" should not be diven
            # in order to avoid that an argment is stored as a list
        """
        args = parser.parse_args() # parse command line argments

        return args

"""
main part

"""
if __name__ == '__main__':

        """------------------------------データ前処理----------------------------------------"""
        """
           パラメータの条件チェック
```

```
    """
    args = analyze_options()
    args.fo.write("#D game1&game2&adapt_k ¥n")
    beta = args.beta
    SEED = args.SEED
    np.random.seed(SEED)

    args.fo.write("#P num_all=%d agent_h0_num=%d max_step_game1=%d
max_step_game2=%d max_gen=%d SEED=%d ¥
 beta=%.2f delta=%.2f decrease_i=%.1f increase_i=%.1f ¥
 decrease_payoff_unit1=%.1f increase_payoff_unit1=%.1f decrease_e=%.1f increase_e=%.1f ¥
 k_max=%d k_min=%d k_median=%d ¥
 alpha=%.2f adapt_probability=%.1f¥n" ¥
%(num_all, agent_h0_num, max_step_game1, max_step_game2, max_gen, SEED, ¥
beta, delta, decrease_e, increase_e, ¥
decrease_payoff_unit2, increase_payoff_unit2, decrease_i, increase_i, ¥
k_min, k_max, k_median, ¥
alpha, adapt_probability ))

    args.fo.write("#L gen, ¥
 ave_payoff ,max_payoff, min_payoff, ¥
 ave_e, max_e, min_e, ¥
 ave_i, max_i, min_i, ¥
 num_go_game1, num_pass_game1, ¥
 frequency_k_list(1,k_max), ¥
 coop_rate_game1, defect_rate_game1, pass_rate_game1, ¥
 cooper_h0_rate_game1, cooper_h1_rate_game1, defecter_h0_rate_game1,
defecter_h1_rate_game1, ¥
 coopee_h0_rate_game1, coopee_h1_rate_game1, defectee_h0_rate_game1,
defectee_h1_rate_game1, ¥
 coop_rate_game2, defect_h0_rate_game2, defect_h1_rate_game2¥
 cooper_h0_rate_game2_parall, cooper_h0_rate_game2, cooper_h1_rate_game2_parall,
cooper_h1_rate_game2, ¥
 defecter_h0_rate_game2_parall, defecter_h0_rate_game2, defecter_h1_rate_game2_parall,
defecter_h1_rate_game2, ¥
 coopee_h0_rate_game2_parall, coopee_h0_rate_game2, coopee_h1_rate_game2_parall,
coopee_h1_rate_game2, ¥
 defectee_h0_byh0_rate_game2_parall, defectee_h0_byh0_rate_game2, ¥
 defectee_h1_byh0_rate_game2_parall, defectee_h1_byh0_rate_game2, ¥
 defectee_h0_byh1_rate_game2_parall, defectee_h0_byh1_rate_game2, ¥
 defectee_h1_byh1_rate_game2_parall, defectee_h1_byh1_rate_game2 ¥n")

    """
    args.foo.write("#P num_all=%d agent_h0_num=%d max_step_game1=%d
max_step_game2=%d max_gen=%d SEED=%d ¥
 beta=%.2f delta=%.2f decrease_i=%.1f increase_i=%.1f ¥
 decrease_payoff_unit1=%.1f increase_payoff_unit1=%.1f decrease_e=%.1f increase_e=%.1f ¥
 k_max=%d k_min=%d k_median=%d ¥
 alpha=%.2f adapt_probability=%.1f¥n" ¥
%(num_all, agent_h0_num, max_step_game1, max_step_game2, max_gen, SEED, ¥
beta, delta, decrease_e, increase_e, ¥
decrease_payoff_unit2, increase_payoff_unit2, decrease_i, increase_i, ¥
```

```
k_min, k_max, k_median, ¥
alpha, adapt_probability ))

    args.foo.write("#L coop_mtx_all coop_mtx_game1 coop_mtx_game2 defect_mtx_all
defect_mtx_game1 defect_h0_mtx_game2 defect_h1_mtx_game2 met_mtx_all ¥n")
    """
    """---------------------------------- メイン----------------------------------------"""
    """agent_list を初期生成する"""
    agent_list = initalize(num_all)

    for gens in range(max_gen):
        #print("==== GEN:", gens, "====")
        """リセット"""
        reset_lastgen(agent_list)
        reset_observer()

        """GAME1"""
        for step_game1 in range( max_step_game1 ):
            #print("===== GAME1 STEP:", step, "=====" )
            game1( agent_list )

        """game1 の参加回数を集計"""
        update_game1_gopass( agent_list )
        """e を i に変換する"""
        transform_e_to_i(agent_list)

        """GAME2"""
        for step_game2 in range( max_step_game2 ):
            #print("===== GAME2 STEP:", step, "=====" )
            game2( agent_list )

        """mtx で全体の coop, defect, met を数える"""
        coop_count_all, defect_count_all, met_count_all¥
            = allmtx( agent_list )
        #print(met_mtx_all)

        """計算( payoff,e,i の平均、最大値、最小値,k の分布 )"""
        ave_payoff, max_payoff, min_payoff, ave_e, max_e, min_e, ¥
          ave_i, max_i, min_i, frequency_k_list, ¥
            num_go_game1, num_pass_game1 ¥
                = calculate_p_e_i_k(agent_list)



        """ゲーム 1 の rate を mtx から計算する"""

        pass_rate_game1, coop_rate_game1, defect_rate_game1, ¥
          cooper_h0_rate_game1, cooper_h1_rate_game1, defecter_h0_rate_game1,
defecter_h1_rate_game1, ¥
            coopee_h0_rate_game1, coopee_h1_rate_game1, defectee_h0_rate_game1,
defectee_h1_rate_game1 ¥
                = rate_game1(agent_list)
```

```
        """ゲーム 2 の rate を mtx から計算する"""
        coop_rate_game2, defect_h0_rate_game2, defect_h1_rate_game2, ¥
          cooper_h0_rate_game2_parall, cooper_h0_rate_game2, cooper_h1_rate_game2_parall,
cooper_h1_rate_game2, ¥
            defecter_h0_rate_game2_parall, defecter_h0_rate_game2,
defecter_h1_rate_game2_parall, defecter_h1_rate_game2, ¥
              coopee_h0_rate_game2_parall, coopee_h0_rate_game2, coopee_h1_rate_game2_parall,
coopee_h1_rate_game2, ¥
                defectee_h0_byh0_rate_game2_parall, defectee_h0_byh0_rate_game2, ¥
                 defectee_h1_byh0_rate_game2_parall, defectee_h1_byh0_rate_game2, ¥
                  defectee_h0_byh1_rate_game2_parall, defectee_h0_byh1_rate_game2, ¥
                   defectee_h1_byh1_rate_game2_parall, defectee_h1_byh1_rate_game2 ¥
                       = rate_game2(agent_list)

        """k をアダプテーション"""
        adaptation_k_game1cost(agent_list)

        """------------------------------出力-----------------------------------------"""
        """計算したデータをパラメータ名のついたファイルに出力する"""
        #ここで出したい結果を書く
        args.fo.write("%d " %(gens))
        args.fo.write("    %.2f %.2f %.2f " %(ave_payoff, max_payoff, min_payoff)) # data related to
payoff
        args.fo.write("    %.2f %.2f %.2f " %(ave_e, max_e, min_e)) # data related to e
        args.fo.write("    %.2f %.2f %.2f " %(ave_i, max_i, min_i)) # data related to i
        args.fo.write("    %d %d " %(num_go_game1, num_pass_game1)) # data related to the
number of agents who coop or defect at game1 per gen
        for ks in range( k_min-1, k_max ):
            args.fo.write(" %.4f"%(frequency_k_list[ks]) )
        args.fo.write("    %.4f %.4f %.4f     %.4f %.4f %.4f %.4f     %.4f %.4f %.4f %.4f    " ¥
            %(coop_rate_game1, defect_rate_game1, pass_rate_game1, ¥
              cooper_h0_rate_game1, cooper_h1_rate_game1, defecter_h0_rate_game1,
defecter_h1_rate_game1, ¥
                coopee_h0_rate_game1, coopee_h1_rate_game1, defectee_h0_rate_game1,
defectee_h1_rate_game1 )) # data related to game1_rate

args.fo.write("    %.4f %.4f %.4f     %.4f %.4f %.4f %.4f     %.4f %.4f %.4f %.4f     %.4f %.4f %.4f %.4f
%.4f %.4f %.4f %.4f     %.4f %.4f %.4f %.4f    " ¥
              %(coop_rate_game2, defect_h0_rate_game2, defect_h1_rate_game2, ¥
                cooper_h0_rate_game2_parall, cooper_h0_rate_game2,
cooper_h1_rate_game2_parall, cooper_h1_rate_game2, ¥
                  defecter_h0_rate_game2_parall, defecter_h0_rate_game2,
defecter_h1_rate_game2_parall, defecter_h1_rate_game2, ¥
                    coopee_h0_rate_game2_parall, coopee_h0_rate_game2,
coopee_h1_rate_game2_parall, coopee_h1_rate_game2, ¥
                      defectee_h0_byh0_rate_game2_parall, defectee_h0_byh0_rate_game2, ¥
                       defectee_h1_byh0_rate_game2_parall, defectee_h1_byh0_rate_game2, ¥
                        defectee_h0_byh1_rate_game2_parall, defectee_h0_byh1_rate_game2, ¥
                         defectee_h1_byh1_rate_game2_parall, defectee_h1_byh1_rate_game2 )) #
data related to the mtxs
```

```
        args.fo.write("¥n")

        """マトリックスをパラメータ名のついたファイルに出力する
        #ここで出したい結果を書く
        args.foo.write("%d " %(gens))

        for coops_all in range( 0, num_all**2 ):
            args.foo.write(" %d"%(np.ravel(coop_mtx_all)[coops_all]) )
        for coops_game1 in range( 0, num_all**2 ):
            args.foo.write(" %d"%(np.ravel(coop_mtx_game1)[coops_game1]) )
        for coops_game2 in range( 0, num_all**2 ):
            args.foo.write(" %d"%(np.ravel(coop_mtx_game2)[coops_game2]) )

        for defects_all in range( 0, num_all**2 ):
            args.foo.write(" %d"%(np.ravel(defect_mtx_all)[defects_all]) )
        for defects_game1 in range( 0, num_all**2 ):
            args.foo.write(" %d"%(np.ravel(defect_mtx_game1)[defects_game1]) )
        for defects_game2_h0 in range( 0, num_all**2 ):
            args.foo.write(" %d"%(np.ravel(defect_h0_mtx_game2)[defects_game2_h0]) )
        for defects_game2_h1 in range( 0, num_all**2 ):
            args.foo.write(" %d"%(np.ravel(defect_h1_mtx_game2)[defects_game2_h1]) )

        for mets in range( 0, num_all**2 ):
            args.foo.write(" %d"%(np.ravel(met_mtx_all)[mets]) )

        args.foo.write("¥n")
        """
    args.fo.close()
    #args.foo.close()
```