ソーシャルフォースモデルを基にした
スーベニアショップの混雑シミュレーション

指導教員名：森口　聡子　准教授

学修番号：13159176

氏名：福岡　由惟

枚数：35 枚

目次

第 1 章　はじめに 2
第 2 章　スーベニアショップにおける混雑の問題について 3
2-1.　スーベニアショップにおける混雑の問題 3
2-2.　混雑時における客の動きを検証する意味 3
第 3 章　過去の研究事例 5
3-1.　M/M/1 型と M/M/s 型の比較 5
3-2.　マルチエージェントを使用したモデル 5
3-3.　ソーシャルフォースモデルを用いた避難シミュレーション 5
第 4 章　本研究におけるシミュレーション 7
4-1.　シミュレーションとシミュレーター 7
4-2.　使用したシミュレーターの紹介 7
4-3.　ソーシャルフォースモデル理論 8
4-4.　モデルの設定 12
4-5.　シミュレーションの実行 15
第 5 章　結果の分析と考察 20
第 6 章　研究のまとめと今後の課題 22
謝辞 23
参考文献 24
付録 25

第1章 はじめに

レジャー施設にとってスーベニアショップは欠かせない収入源となっている。レジャー施設利用者の多くは、レジャー施設で遊んだ後、スーベニアショップを利用する。そのため、閉園間際にはスーベニアショップの店内は大変混雑してしまう。

店の利用者で、混雑していない時間帯に買い物をし、コインロッカーを利用する人が増加すれば理想的であるが、コインロッカーも利用料金がかかり、利用者の負担となるため、実現は困難である。また、店自体を拡大することも、土地や工事費を考えると難しい。

本研究ではスーベニアショップに代表される、店の中にレジが点在しており、客が商品をレジまで持って行き清算した後、退店するタイプの店において、店内の客の動きを検証する。先に述べたように、レジャー施設にとってスーベニアショップは大きな収入源になっているため、店内混雑による客の満足度や購買意欲の低下は、大きな問題である。店内の客の行動を検証することは、特に混雑している場所やデッドスペースなどが分かり、混雑解消の手がかりとなるため、意義のあるものである。

実験には S4 Simulation System のソーシャルフォースモデルを用い、店内における客の行動シミュレーターを制作する。このシミュレーターを制作することにより、店内の混雑を可視化、混雑の差を数値化し、店内の混雑を考察できるようにする。

第2章 スーベニアショップにおける混雑の問題について

2-1. スーベニアショップにおける混雑の問題

レジャー施設において、スーベニアショップは大きな収入源となっている。例えば、表1からわかるようにディズニーリゾートを運営する(株)オリエンタルランドでは、利益の約3割が商品販売収入である。

表1 (株)オリエンタルランドのセグメント別売上高内訳
出典 [2] p.5

(百万円)

	'11/3	'12/3	'13/3	'14/3	'15/3	'16/3
■テーマパーク事業	¥ 290,478	¥ 297,891	¥ 329,814	¥ 390,912	¥ 387,622	¥ 384,602
アトラクション・ショー収入	127,698	130,190	143,696	165,695	169,590	175,559
商品販売収入	104,294	108,692	119,946	148,265	142,361	134,586
飲食販売収入	55,238	55,930	62,201	71,835	70,786	69,140
その他の収入	3,264	3,077	3,969	5,115	4,883	5,316
■ホテル事業	44,004	42,210	48,924	65,933	61,066	63,173
東京ディズニーランドホテル	13,723	13,413	15,416	17,309	16,674	17,933
東京ディズニーシー・ホテルミラコスタ	14,694	15,122	16,841	17,373	16,080	16,540
ディズニーアンバサダーホテル	11,919	11,759	12,690	15,278	14,466	14,433
その他	3,667	1,914	3,976	14,971	13,845	14,266
■その他の事業	21,697	19,959	16,787	17,727	17,603	17,576
イクスピアリ事業	8,649	8,591	8,633	8,519	8,683	8,788
シアトリカル事業	4,749	3,670	—	—	—	—
モノレール事業	3,449	3,396	3,829	4,262	4,147	4,351
その他	4,848	4,301	4,324	4,944	4,772	4,437
合計	¥356,180	¥360,060	¥395,526	¥473,572	¥466,291	¥465,353

注： 1. ホテル事業において、2013年3月29日付で（株）ブライトンコーポレーションの全株式を取得しました。
2. シアトリカル事業のシルク・ドゥ・ソレイユ「ZED™（ゼッド）」は、2011年12月31日付で終了しました。

もし店内が混雑していると、利用者の中には、レジャー施設に対する満足度が低下したり、購買意欲を無くしたりする者が出てきてしまう。これは、スーベニアショップが大きな収入源となっているレジャー施設において大きな損失である。しかし、利用者を減らさずに店内の混雑を緩和することは難しく、問題となっている。

2-2. 混雑時における客の動きを検証する意味

研究するにあたって、まず一般的なスーパーマーケットやコンビニエンスストアのような店舗とスーベニアショップの違いを論じる。

一般的な店舗では、レジが一か所に集まっており、客が自分で空いているレジを選択することが可能である。さらに、一般店舗において、混雑時にレジが行列となることはあっても、

店内が混雑して動きにくいということはあまり見受けられない。

しかし、スーベニアショップはレジが店内に点在しており、客は現在地から一番近いレジで精算することが多い。また、スーベニアショップは、レジャー施設利用客の多くが利用するものであり、遠方から来ている客も多い。そのため、一人で複数個土産を買う者も少なくない。その結果、荷物を持って歩くのも大変であり、コインロッカーを利用するにも客の負担が増えるため、閉園間際に利用する客が多くなる。そのため、閉園間際のスーベニアショップは大変混雑してしまうのである。

単純に店舗を拡大すれば混雑は減るであろうが、土地も資金も限られているため、現在のスペースで運営せざるを得ない。その結果混雑が発生する。もし、店内の混雑を解消することができれば、顧客満足度の上昇とより多くの客の利用を見込むことができ、利益の増加が見込める。そして、店内の混雑を解消するためには、店内のどの場所が特に混雑しているのか知る必要がある。これを知ることができれば、陳列棚の移動により通路を広くしたり、人気の商品を複数個所に分けたりすることにより混雑解消につながると思われる。

第3章 過去の研究事例

3-1. M/M/1 型と M/M/s 型の比較

森　弘隆による「瀬戸キャンパスにおける食堂の待ち行列」[11]は、食堂において食事を受け取るカウンターとレジの待ち行列を解消しようとする研究である。この研究では、待ち行列理論の分野である M/M/1 型と M/M/s 型を比較している。M/M/1 型とは、レジ 1 つに対しそれぞれ列が発生するものであり、M/M/s 型とは、複数の列に 1 つだけ列を作り、客が空いたレジに随時入っていくものである。そして、食券機導入による客の滞在時間を現状のデータと比較している。結果として M/M/s 型、食券機導入による混雑解消の可能性が認められた。

この研究は、混雑を解消するために窓口の数を変化させている。結果として、混雑解消が認められているが、窓口を増やすためのスペースの問題には触れられておらず、すでにある施設への導入は難しい。

3-2. マルチエージェントを使用したモデル

芹澤　良による「マルチエージェントを使用したレジにおける混雑解消法の検証」[7]は、スーパーマーケットにおいてレジの待ち行列を解消しようとする研究である。この研究では、M/M/1 型のスーパーマーケットのレジを、商品数の少ない客専用のレジ、商品客の多い客専用のレジの 2 つに分け、マルチエージェントシミュレーターを使用し、それぞれの適切なレジ数を導き出している。結果として、客の購入する商品の数によって並ぶ列を変えるという改善案が、有効に働く可能性が高い場合の条件を導き出すことができている。

この研究はレジにフォーカスが当てられており、店内の混雑については考えられていない。また、レジは 1 か所に並んでいるものが想定されているため、スーベニアショップのようなレジが点在している店舗への導入は難しい。

3-3. ソーシャルフォースモデルを用いた避難シミュレーション

磯崎勝吾・中辻隆による「Social force model を基にした歩行者の避難シミュレーションモデルに関する研究」[1]では、ソーシャルフォースモデルをベースとした避難シミュレーションモデルの有効性について論じられている。歩行者同士の影響によりどのように動きが変わるかといったことに目を向けたソーシャルフォースモデル、実際に行った実験、マルチエージェントモデルの 3 つを比較している。その結果、マルチエージェントモデルよりもソーシャルフォースモデルのほうが実験との誤差が少ないため、局所的かつ歩行者密集と

いう状況下においてソーシャルフォースモデルの再現性はかなり高精度であることが分かっている。

谷口豊による「Social force model を用いた災害時地下歩行空間における歩行者の避難挙動分析」[8]では、ソーシャルフォースモデルを用いて避難挙動を再現している。また、実際の避難実験と比較検証することで、ソーシャルフォースモデルと再現性の検討も行っている。その結果、避難実験とシミュレーションで人々の同様な挙動が見られたので、再現性の高いシミュレーションを行うことが出来たということが分かっている。

この研究のようにソーシャルフォースモデルは様々な避難シミュレーションで活用されている。歩行者が密集しているということは、店舗が混雑している状況にも言えることであるため、店舗混雑のシミュレーションにも大変有効なのではないかと考えられる。

第4章 本研究におけるシミュレーション

4-1. シミュレーションとシミュレーター

シミュレーションとは現実の複雑なシステムをモデル化することによって、その振る舞いを予測・分析する問題解決方法である。実際に存在しないようなシステムや、実行するにあたり時間・コストが掛かり過ぎてしまうようなシステムにシミュレーションは非常に有効である。例えば、収益予測やリスク分析などの問題は計算を行うには不確定要素が多くシステムが複雑であるため、結果を予測することは難しい。このような時にシミュレーションが役立つのである。また、複数の解を比較できるという点もシミュレーションの大きなメリットであり、様々に条件を変化して得られた解の優劣を容易につけることが出来る。シミュレーションには様々な種類がある。離散型シミュレーション(待ち行列などを数値的に取り扱うシミュレーション)や、確率的シミュレーション(システムを確率的に変動させ結果が確率的に変動するシミュレーション)などその他多数存在する[4]。

本研究では、ソーシャルフォースモデルを研究対象とする。これは歩行者の経路選択などといったことよりも、歩行者同士がいかに影響して挙動が変わるかということに主眼を置いたモデルである。

上記で述べたような様々なシミュレーションを行うことができるシミュレーターを紹介する。

例えば構造計画研究所の Visual SLAM 汎用シミュレーションモデル構築ツ―ルである。これは、日本国内で最も使用されている汎用シミュレーション言語である。速度、柔軟性に優れており、幅広い対象システムの分析や ユーザ独自の問題を解決するための専用シミュレーター構築が効率的に実現できる[5]。構造計画研究所では他にも artisoc マルチエージェント・シミュレーションプラットフォームというものもある。これは、社会現象など人間の意思決定に基づいたシステムを分析するアプローチである「マルチエージェント・シミュレーション」を構築するプラットフォームである[6]。

4-2. 使用したシミュレーターの紹介

実際にシミュレーションを実行するにあたり、使用したシミュレーターは S4 Simulation System(エスクワトロ・シミュレーション・システム)という(株)NTT データ数理システムが開発した製品である。これは汎用的な離散イベントシミュレーションシステムである。工場などの生産システム、サプライチェーンなどの流通システム、銀行の窓口や ATM、通信システム、交通システム、コールセンターなど、確率的な振る舞いをするものを対象とする

ような様々な領域のシミュレーションを行うことが出来る。シミュレーションに必要となる基本的な部品は用意されており、簡単なGUI(グラフィカルユーザーインターフェース)操作でそれらを組み合わせることにより、問題を手軽にかつ本格的にシミュレーションすることが出来る。また、分析を同時に行うことが出来るのも一つの特徴である。S4 Simulation System は、簡単にシミュレーションモデルの作成と分析が出来る操作性と、自由度を合わせ持つシミュレーションシステムである[4]。

4-3. ソーシャルフォースモデル理論

本研究で用いるソーシャルフォースモデルを紹介する[1, 9]。Dirk Helbing・Peter Molnar が提唱したソーシャルフォースモデル[10]は、群集行動の力学ベースモデルのひとつである。まずこのモデルでは各歩行者は質量(歩行者の体重)を持つ質点として表され、平面内で運動する粒子とみなす。各歩行者は、目的地を持つが、他の歩行者や障害物から相互に干渉を受けながら、それぞれが運動するようなモデルである。

$F = ma$ (1)

その質点(歩行者)ごとに運動方程式(1)を当てはめることで、ある時刻の歩行者の位置から、次の時刻への進む方向、歩行速度などが算出されるというものである。つまり運動方程式における外力 F を他者や、障害物からの見かけ上の力として仮定することで算出するということになる。この F は大きく分けて以下の 2 通りに分類することが出来る。

I. 目的地へ進もうとする力

図 1 に示すように歩行者が他の影響により当初考えているコース(基本は目的地への最短経路)からずれてしまった場合に他の目的地の方向へ進行方向と曲げるように発生する力。式(2)のように表すことが出来る。シミュレーション内でまずこの項が働かなければ歩行者は目的地に進まない。

$$f_i(t) = \frac{v_i^0(t) e_i^0(t) - v_i(t)}{\tau_i} \quad (2)$$

ここで v_i^0(t):歩行最適速度、$e_i^0(t)$:目的地に向かうベクトル、$v_i(t)$:現在の速度、τ_i:加速時間とする。

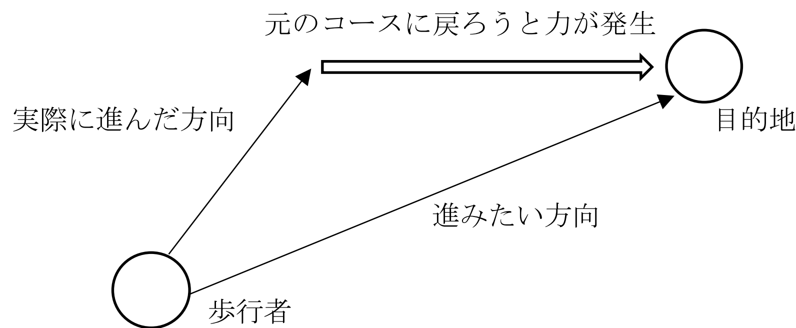

図 1　要因 1 模式図、出典　[1] p.1

II. 他者、障害物から受ける反発力(避ける力)

図 2 に示すように近くにいる歩行者は互いに相手とぶつからないように避け合う性質があると考えられる。これを再現するのが式(3)で表される 2 つ目の要因である。これは他の歩行者 j との相互作用f_{ij}と壁 W との相互作用の和で表される。

$$f_i(t) = \sum_{j(\neq i)} \left[f_{ij}^{soc}(t) + f_{ij}^{alt}(t) \right] \quad (3)$$

また、この性質を相手から斥力を受けていると考え、距離に応じて指数関数的に減少するポテンシャルと仮定することが多い。身体が接触して働く物理的な力ではなく、社会心理学的な効果によるものであるため、ソーシャルフォースと呼ばれる。そして、歩行者との衝突を避ける性質は式(4)のように表される。

$$f_{ij} = A_i \exp\left(-\frac{d_{ij}}{B_i}\right) n_{ij} \quad (4)$$

A_i, B_iはそれぞれ相互作用の強さと範囲、$d_{ij} = \left|\vec{d_i} - \vec{d_j}\right|$は歩行者の間の距離、$n_{ij}$は i から j 方向の単位ベクトルである。

同様に、壁との衝突を避ける性質も壁から斥力を受けていると考えて、式(5)のように書く。

$$f_{iW} = A_i \exp\left(-\frac{d_{iW}}{B_i}\right) n_{iW} \quad (5)$$

d_{iW}は壁までの距離、n_{iW}は壁への法線方向の単位ベクトルを表す。

このソーシャルフォースは主に視覚情報にもとづく歩行者の判断に起因するものと考えられる。人間の視界は前方に限られているため、自分の前方にいる人からは大きなソーシャルフォースを感じる一方、後方にいる人から受ける力は小さいとするのが自

然である。そこで、適当な重み関数をかけあわせることによって視界の効果を取り入れる手法もしばしばとられる。

また、注目している空間が空いているときはソーシャルフォースのみで人の動きを考えることができるが、混雑しているときには歩行者どうしの物理的な接触も無視できなくなる。そのためソーシャルフォースに加えて、摩擦のある剛体の間に働く相互作用に似た形を持つ物理的な力の項を加えることもある。接触の法線方向成分n_{ij}, n_{iW}と接線方向成分t_{ij}, t_{iW}それぞれに力を受けて、

$$f_{ij} = \left\{A_i exp\left[\frac{(r_{ij}-d_{ij})}{B_i}\right] + k\theta(r_{ij} - d_{ij})\right\}n_{ij} + K\theta(r_{ij} - d_{ij})(v_i \cdot t_{ij})t_{ij} \quad (6)$$

$$f_{iW} = \left\{A_i exp\left[\frac{(r_i-d_{iW})}{B_i}\right] + k\theta(r_i - d_{iW})\right\}n_{iW} + K\theta(r_i - d_{iW})(v_i \cdot t_{iW})t_{iW} \quad (7)$$

ここでr_{ij}は i と j の影響範囲の和、k と K は弾性と散逸の係数である。また、 θ はランプ関数で以下とする。

$$\theta\,(x) = \begin{cases} x & \text{for } x \geq 0 \\ 0 & \text{otherwise} \end{cases}$$

以上に挙げた式の r, d, θ の関係を図 3 に示す。

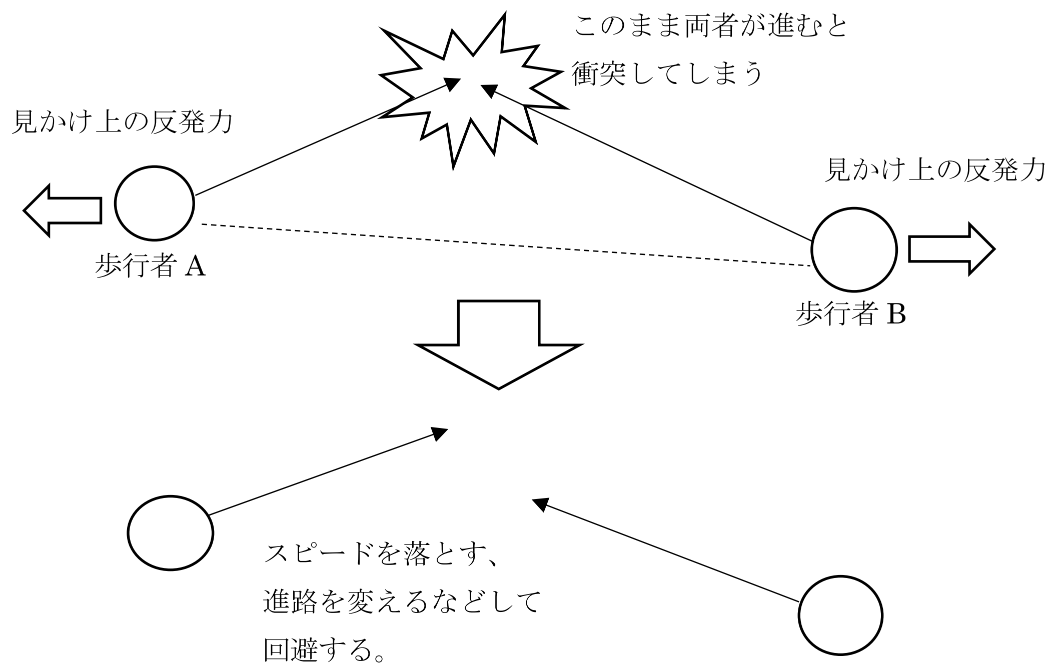

図 2　要因 2 模式図、出典　[1] p.2

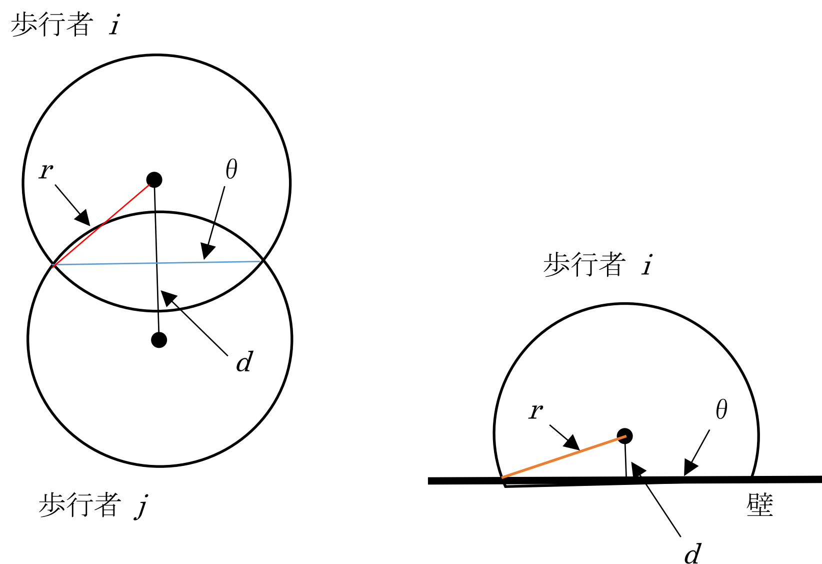

図 3　接触時の模式、出典　[1] p.2 (一部修正)

4-4. モデルの設定

本研究で使用した S4 Simulation System では、質量m_iを持つ歩行者は下記の運動方程式に基づくものとして設定されている。

$$m_i \frac{d\overrightarrow{v_t}}{dt} = m_i \frac{v_{0i}\overrightarrow{e_i}(t) - \overrightarrow{v_i}(t)}{\tau_i} + R(c, \sum_{j(\neq i)} \overrightarrow{f_{ij}} + \sum_{W} \overrightarrow{f_{iW}})$$

ここで、$\overrightarrow{e_i}$:目的地に向かうベクトル、$\overrightarrow{v_i}(t)$:現在の速度、v_{0i}:歩行者の最適な速度、τ_i:加速時間とする。

$R(c,p)$は、平均 p、分散共分散共分散行列が$\begin{pmatrix} \sigma^2 & 0 \\ 0 & \sigma^2 \end{pmatrix}, \sigma = c\|p\|$の多変量正規分布に従う乱数である。$\overrightarrow{f_{ij}}$は歩行者 jから歩行者 iに与える外力であり、以下のように表される。

$$\overrightarrow{f_{ij}} = \left\{A_i exp\left[\frac{r_i + r_j - d_{ij}}{B_i} + kg(r_i + r_j - d_{ij})\right]\overrightarrow{n_{ij}}\right\} + Kg(r_i + r_j - d_{ij})\left\{\left(\overrightarrow{v_j} - \overrightarrow{v_i}\right)\cdot\overrightarrow{t_{ij}}\right\}\overrightarrow{t_{ij}}$$

ここで、r_i, r_j:歩行者の半径、d_{ij}:歩行者 iと歩行者 jの距離、k:弾性係数、K:散逸係数、n_{ij}:歩行者 jから iに向かう単位ベクトル、$\overrightarrow{t_{ij}}$:$\left(-n_{ij}^2, n_{ij}^1\right)$とし、また、$g$は以下とする。

$$g(x) = \begin{cases} x & x \geq 0 \\ 0 & \text{otherwise} \end{cases}$$

$\overrightarrow{f_{iW}}$は障害物 Wから歩行者 iに与える外力であり、以下のように表される。

$$\overrightarrow{f_{iW}} = \left\{A_i exp\left[\frac{r_i - d_{iW}}{B_i} + kg\left(r_i - d_{iW}\right)\right]\overrightarrow{n_{iW}}\right\} + Kg\left(r_i - d_{iW}\right)\left\{\overrightarrow{v_i}\cdot\overrightarrow{t_{iW}}\right\}\overrightarrow{t_{iW}}$$

ここで、r_i:壁までの距離、$\overrightarrow{n_{iW}}$:壁への法線方向の単位ベクトル、$\overrightarrow{t_{iW}}$:$\left(-n_{iW}^2, n_{iW}^1\right)$、$A_i$:歩行者 iの相互作用の強さ、B_i:歩行者 iの相互作用の範囲とする。

S4 Simulation System において、シミュレーション空間は 2 次元の平面であり、左下の座標(x_0, y_0)と、右上の座標(x_1, y_1)で定められる。その中に任意の障害物を配置させること

が出来る。また、ソーシャルフォースモデルでは、各エージェントが(可視な)単一の目的地を持つような場合に、エージェント間、障害物間との相互干渉をモデリングする。つまりソーシャルフォースモデルだけでは、目的地が可視でない場合、エージェントがスタックしてしまうような現象が容易に発生する。そこで、S4 Simulation System では、ソーシャルフォースモデルとは別に、複数の経路ポイントを経由した歩行者の行動もサポートするように設計されている。エージェントの通過する可能性のある地点を経路地点と呼ぶ、経路地点は面積を持つ円である。互いに経路地点の中心を視認出来る経路地点の組はエッジで結ばれる。そのようにして作成された無向(もしくは有向)グラフを経路グラフと呼ぶ。ソーシャルフォースモデルでは、障害物を越えて目的地に到達することが出来ない。視認できない目的地が設定された場合は、経路グラフを元に途中の経路が選択される[3]。

以上のことを踏まえ、モデルを設定する。モデルの設定については、エージェントと環境の二つに分け、詳細を付録に掲載した。まず今回のシミュレーションにおける仮想のスーベニアショップを図 4 のように設定する(マップ設定のプログラムについては付録 2 を参照)。

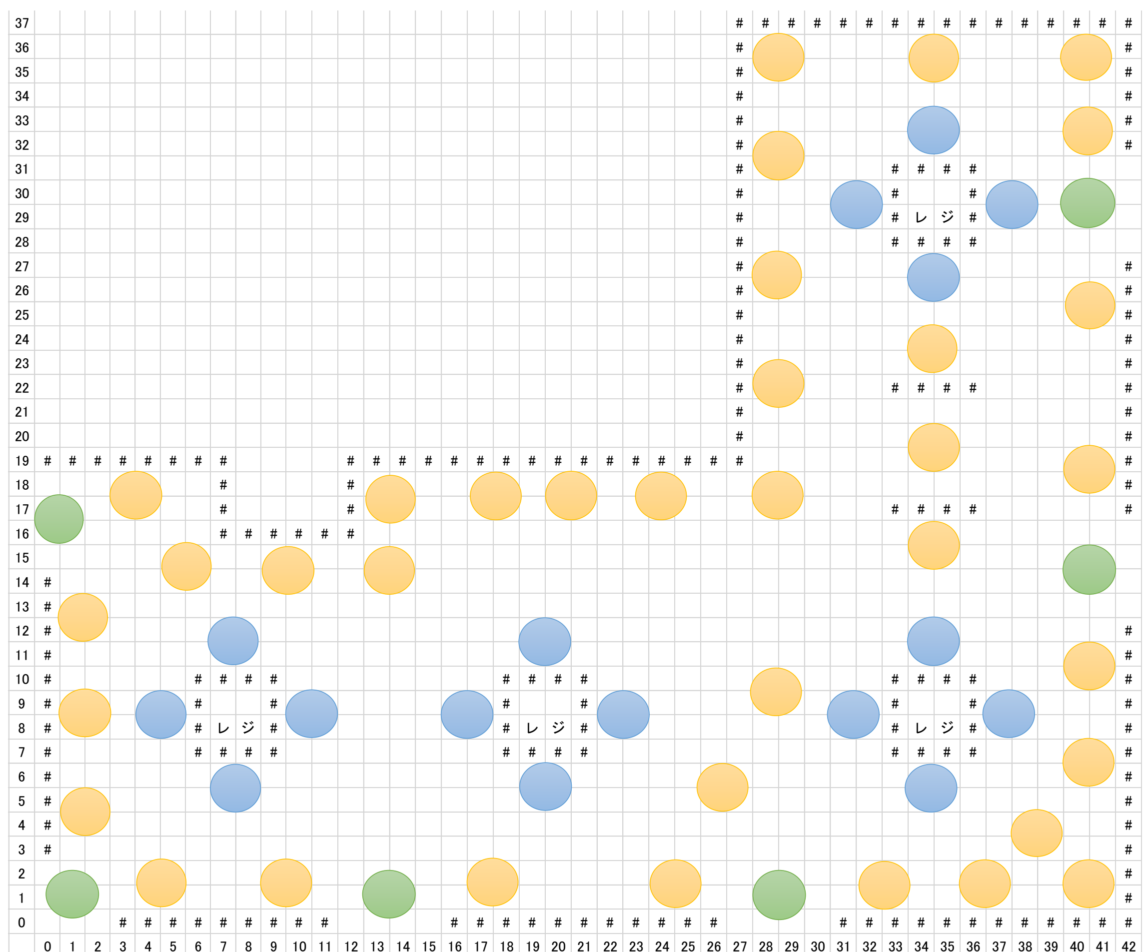

図 4　仮想店舗のマップ

図 4 において、#は壁や棚、レジのような障害物を示すものである。そして、円が経路地点である。緑色の円が出入り口、青色の円がレジ、黄色の円が店内で客が商品を見る可能性のある地点である。これらを設定すると図 5 のようなマップが出来る。

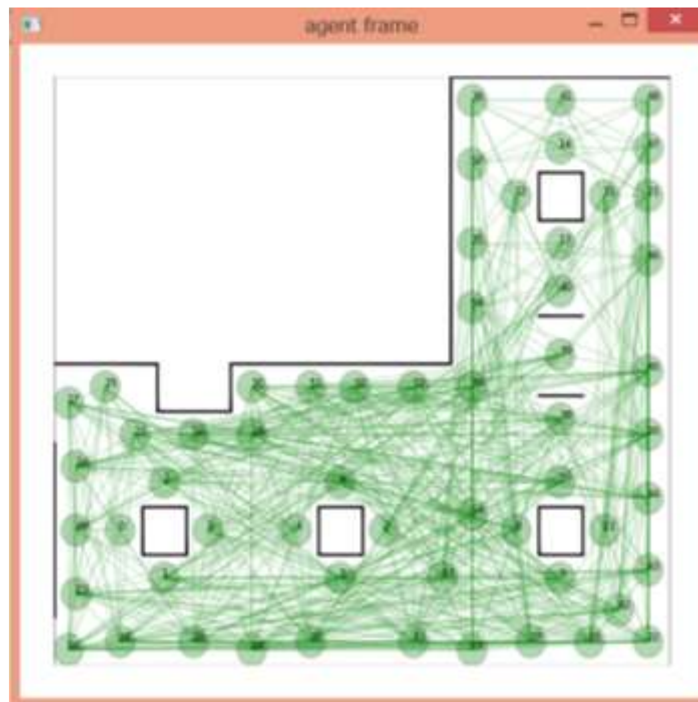

図 5　シミュレーション時のマップ

次に、本研究におけるエージェント移動のルールを示す(エージェント設定のプログラムについては付録 1 を参照)。

① 出入り口からランダムに客が来店する。

発生確率はプログラム内の self.genP において設定することができ、ここでは 0.02 とする。

② 入店したお客は各売り場の代表点を経由してレジに向かう。

これは、図 5 の経路ポイントの番号をもとに、予め 16 個のパターンを設定しておき、入店時にランダムにエージェントに振り分けるようにする。パターンは以下の通りである。

ルート 1:出入り口 16→26→31→54→32→35→41→46→レジ 15→出入り口 21

ルート 2:出入り口 16→10→25→50→32→54→42→39→レジ 13→出入り口 21

ルート 3:出入り口 17→25→50→32→35→39→42→54→レジ 8→出入り口 19

ルート 4:出入り口 17→10→26→31→54→39→41→35→レジ 12→出入り口 21

ルート 5:出入り口 18→31→42→46→39→32→50→25→レジ 2→出入り口 17

ルート 6:出入り口 18→26→10→50→32→54→42→39→レジ 10→出入り口 19

ルート 7:出入り口 18→50→32→39→46→41→35→42→レジ 11→出入り口 20
ルート 8:出入り口 19→54→32→50→25→10→26→31→レジ 5→出入り口 18
ルート 9:出入り口 19→31→50→32→35→41→46→54→レジ 7→出入り口 18
ルート 10:出入り口 19→42→39→32→50→31→26→10→レジ 0→出入り口 16
ルート 11:出入り口 20→39→32→54→31→26→10→25→レジ 3→出入り口 17
ルート 12:出入り口 20→46→41→35→32→50→31→26→レジ 1→出入り口 16
ルート 13:出入り口 20→42→54→32→39→46→41→35→レジ 14→出入り口 21
ルート 14:出入り口 21→46→39→32→32→50→10→26→レジ 4→出入り口 16
ルート 15:出入り口 21→41→35→39→42→31→50→32→レジ 6→出入り口 18
ルート 16:出入り口 21→41→35→32→50→26→31→42→レジ 9→出入り口 19

このとき、なるべくすべての出入り口から行くことのできる全方向に向かうルートを作成し、また、人気の棚(ここでは 32 とする)を必ず通るようにする。

③ レジを出たお客は出入り口から退店する

最後に様々な数値を設定する。

$(x_0, y_0) = (0,42), (x_1, y_1) = (0,37)$

これは図 4 をもとに設定する。

最適速度(m/s)=0.6、最高速度(m/s)=1.5

これは平均的な歩行速度を最高速度とし、店内を見て回ることを考え最適速度を 0.6 とした。

歩行者の半径(m)=0.2、体重(kg)=50

これらは、一般的な人を考え、設定した。

4-5. シミュレーションの実行

上記の設定のもと、シミュレーションを行い、シミュレーション時間 100 ごとに結果を図 6 から図 25 に示す。このときマップ上にある赤い点が客を表している。また、100 ごとに、各経路ポイントに到達した客数の合計をグラフに表す。

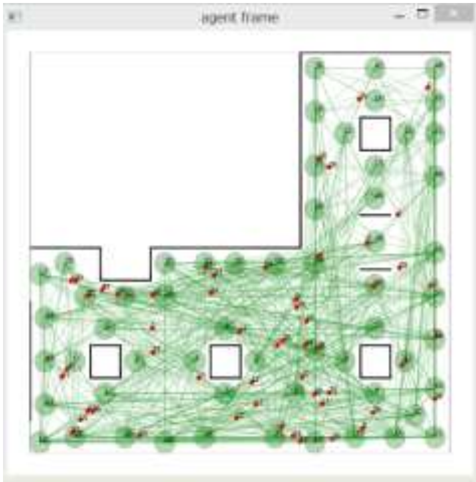

図 6　100 時点でのマップ

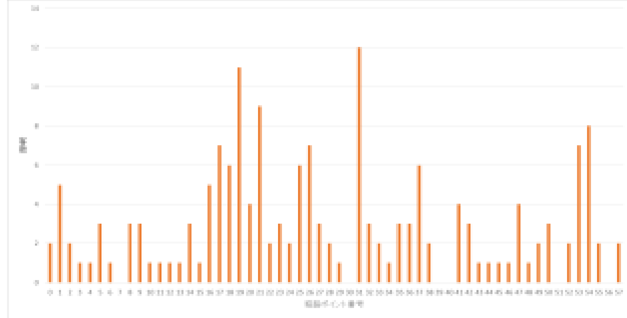

図 7　0～100 までの客数の合計

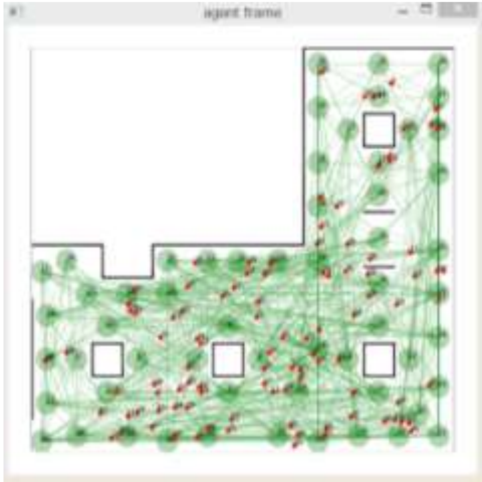

図 8　200 時点でのマップ

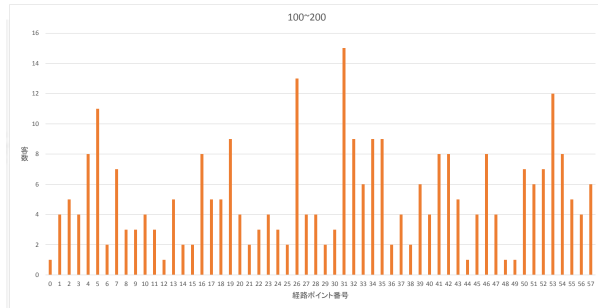

図 9　100～200 までの客数の合計

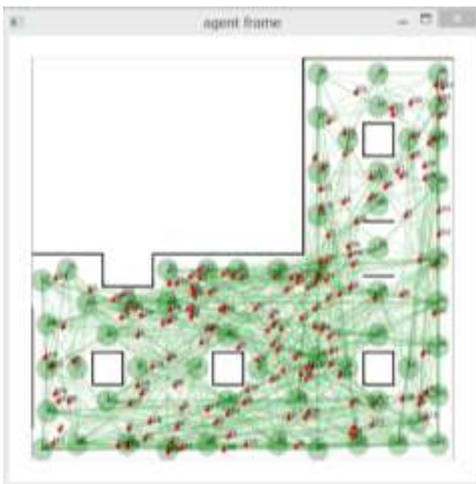

図 10　300 時点でのマップ

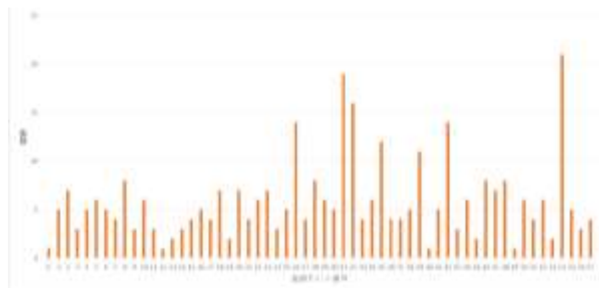

図 11　200～300 までの客数の合計

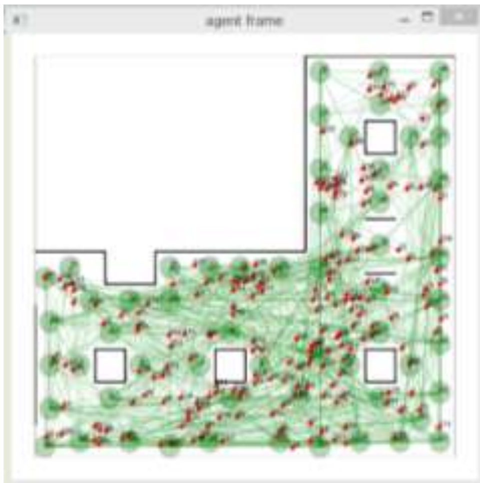

図 12　400 時点でのマップ

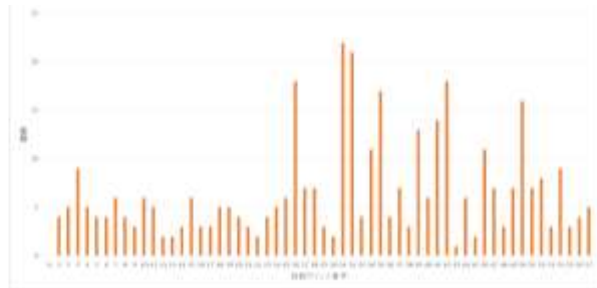

図 13　300～400 までの客数の合計

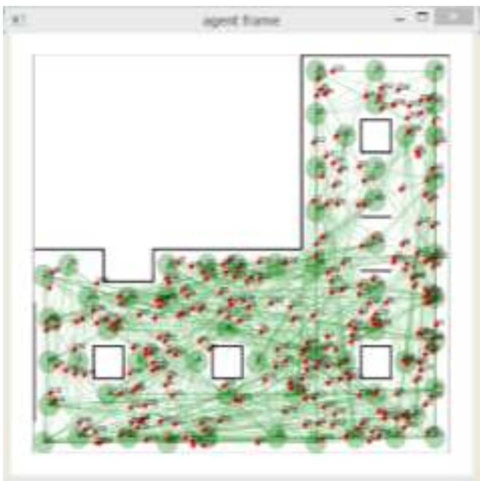

図 14　500 時点でのマップ

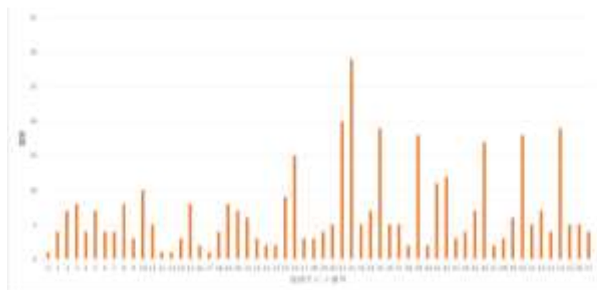

図 15　400～500 までの客数の合計

図 16　600 時点でのマップ

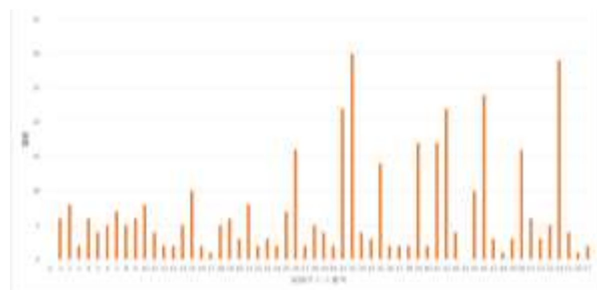

図 17　500～600 までの客数の合計

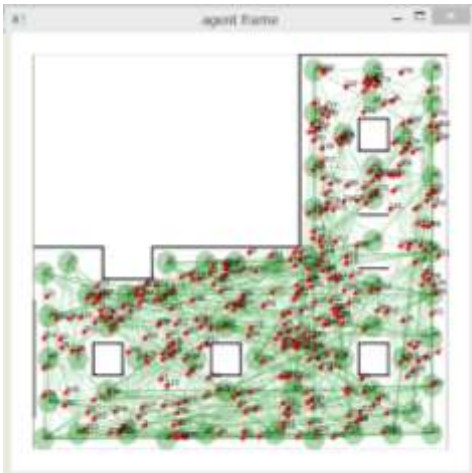

図 18　700 時点でのマップ

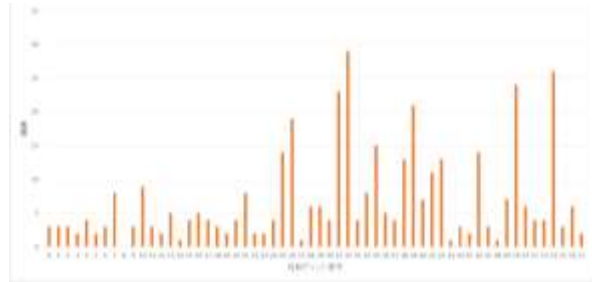

図 19　600～700 までの客数の合計

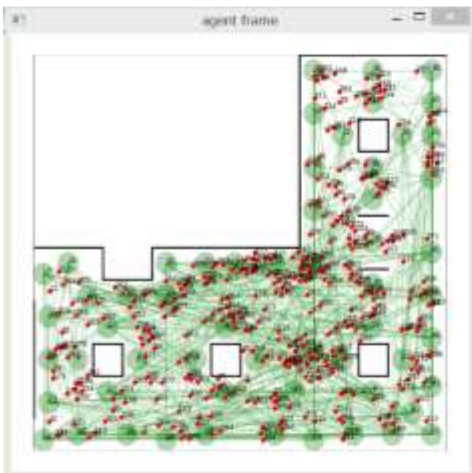

図 20　800 時点でのマップ

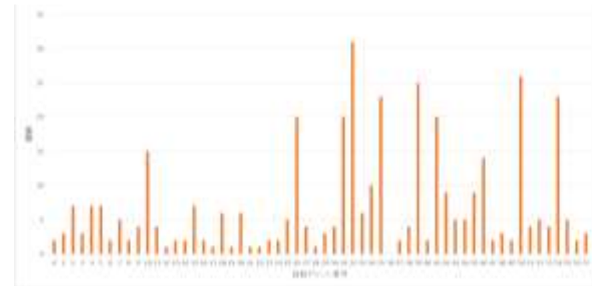

図 21　700～800 までの客数の合計

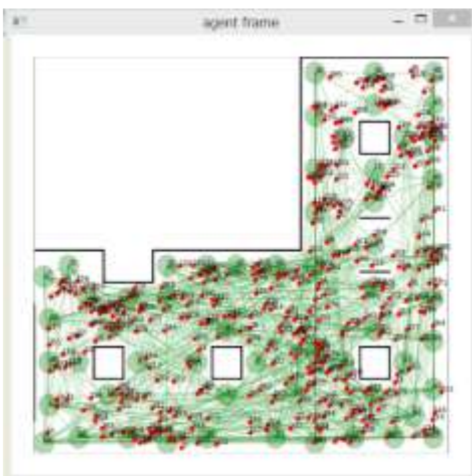

図 22　900 時点でのマップ

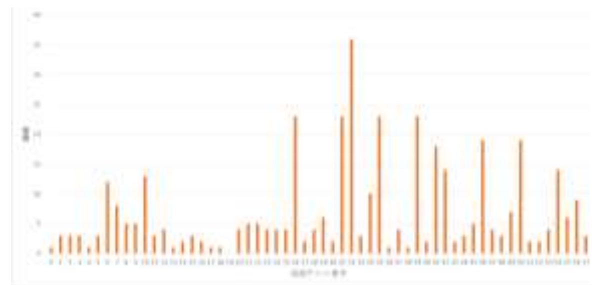

図 23　800～900 までの客数の合計

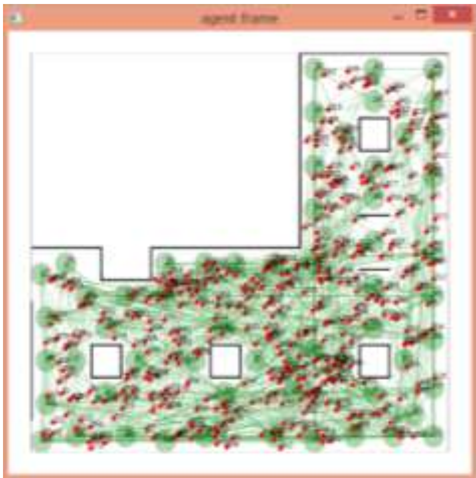

図 24　1000 時点でのマップ

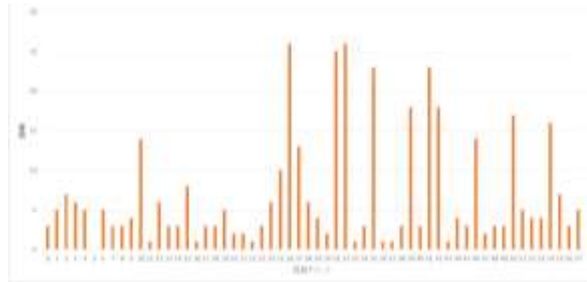

図 25　900～1000 までの客数の合計

第5章 結果の分析と考察

ソーシャルフォースモデルをベースとしたシミュレーションを行うことによって、店内の混雑を可視化することが出来た。経過時間ごとにマップとグラフを比較すると、シミュレーション開始後 100 の時点では、まだ客はまばらで、経路ポイントにより客数の差はあるもののそれほど大きくはないことが分かる(図 6、7)。開始後 200 の時点では、まだ混雑は確認できず､開始からそれほど経っていないためか､人気のポイントである 32 付近よりも、入り口に近い 26、31、53 に訪れた客数が多い(図 8、9)。開始後 300 の時点では、200 の時点より少し混雑が見られるようになった(図 10)。時間が経過し、入り口付近だけではなく、32、42、54 といった入り口から少し入った、あるいは奥まったところの客数の伸びが見られる(図 11)。開始後 400 の時点では、マップから見られる混雑度合は 300 の時点とあまり変わらない(図 12)。しかし、グラフより 35、50 がとても伸びていることが分かる。これは入り口から人気のポイントである 32 に向かう客が通っているからである(図 13)。開始後 500、600 の時点では、店内全体で混雑が見られるようになった(図 14、16)。グラフから見て取れる客数は多いポイントは 400 までと変わらない(図 15、17)。開始後 700、800 の時点では、店内にかなりの混雑が見られる(図 18、20)。またこの時点では混雑しているポイントと混雑していないポイントの差がマップからも分かるようになった。実際、グラフと比較すると、36・57 や 47・48、37・55 といった、人気の棚から離れているところの客数が少ない(図 19、21)。開始後 900、1000 の時点では、店内がより混雑していることが分かる(図 22、24)。また、マップからもグラフからも混雑しているポイントと混雑していないポイントとの差が大きいことが分かる(図 23、25)。

このように、図 6 ではまばらであった人が、図 24 ではかなり混雑していることが見てとれた。また、経路ポイントを通った客数を数値化することにより、人気のある、人が集まりやすいポイントとそうではないポイントとの差がより明らかになった。例えば通る客数が一番多いポイントである 32 と一番少ないポイントである 43 を比べてグラフにすると以下のようになる。

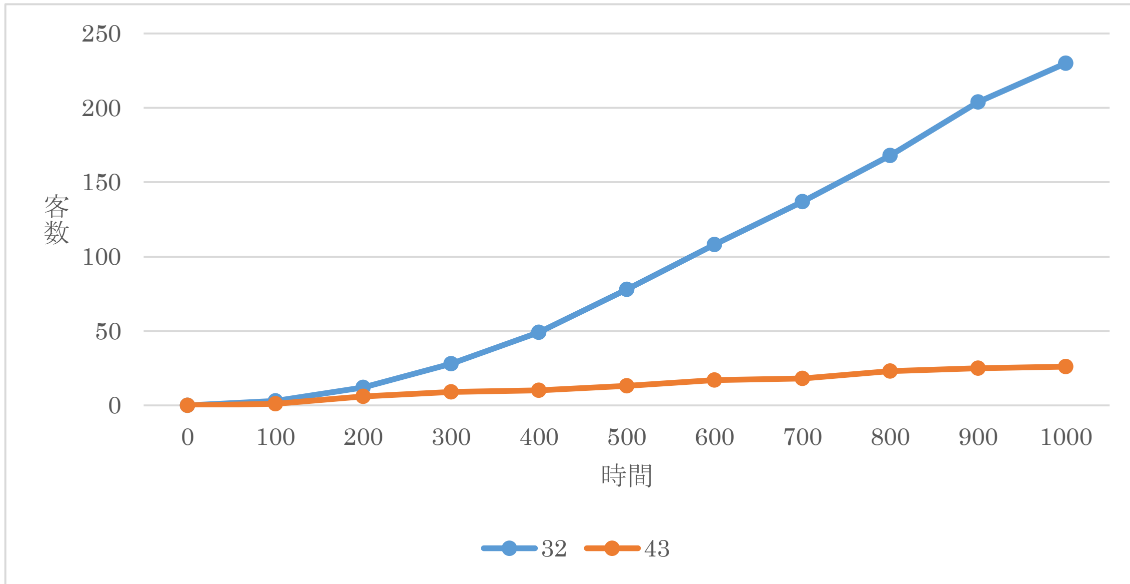

図 26　時間ごとの客数の合計

図 26 から分かるように差は歴然である。混雑するポイントが可視化できたため、混雑する人気の棚を分散させたり、場所を移動させたりすることで混雑の緩和につながる。また、実際に棚を移動して本当に混雑が緩和されるかどうか検証するシミュレーションを行うことができるのではないかという予測ができる。また、ソーシャルフォースモデルは物理力モデルであるため、混雑時に人と人とが押し合うような挙動を再現することができ、これまで混雑緩和に使われていたエージェントシミュレーションよりも現実に近いシミュレーションができた。

第6章 研究のまとめと今後の課題

最終的に混雑の可視化や混雑するポイントと混雑していないポイントの比較を行うことはできたが、より現実に近いモデルをつくることが課題としてあげられる。例えば本研究において入店率は混雑している状態をつくることができる数値に設定した。これを実際の統計に基づき、入店率を設定することでよりそれぞれの店舗にあったモデルを作成することができる。また、客が必ず商品を購入する、つまりレジを通るように設定されている。これについても実際の統計に基づき、買い上げ率を加えることで現実に近いモデルを作成することができる。

本研究において、ソーシャルフォースモデルは避難モデルだけではなく、混雑緩和に活用できることが分かった。つまり、客を避難口から出るように設定すれば、混雑だけでなく避難をシミュレーションすることもできることが考えられる。避難モデルと混雑緩和、両方のシミュレーションができるのである。

また、本研究ではレジの混雑は考えられていない。レジは一般的に 1 列に並ぶため、人と人との押し合いが少ない場所である。よって、 [3]でも使用されていたようにレジの混雑にはマルチエージェントモデルが適していると考えられる。そのため、使用する状況を踏まえたうえで欲しい結果に応じて、ソーシャルフォースモデルとマルチエージェントモデルを使い分けることが重要である。両方を使い分けることによって、より混雑緩和をする方法を細かく考えることができるのではないかと予測する。

今後の課題として、ソーシャルフォースモデルの他分野への応用、ソーシャルフォースモデルとマルチエージェントモデルの使い分けの境界などまだ研究の発展が考えられる。

謝辞

本論文執筆にあたりお世話になりました森口聡子准教授に感謝の意を示したいと思います。熱心にご指導頂いたことや研究に必要なシミュレーターをご提供頂いたこと、本論文をご精読頂き多くの有用なアドバイスを頂いたことを深く心より御礼申し上げます。また、シミュレーターを扱う上で、様々な技術的アドバイスをして下さった(株)NTT データ数理システムの嶋田佳明様にも深く感謝しております。

参考文献

[1] 磯崎勝吾・中辻隆、「Social force model を基にした歩行者の避難シミュレーションモデルに関する研究」、北海道大学大学院工学研究科、平成 21 年度土木学会北海道支部論文報告集第 66 号 D-3

[2] 株式会社オリエンタルランド、「ファクトブック 2016」、2016 年 3 月期

[3] 株式会社 NTT データ数理システム、「S4 Simulation System　Version 4.2 psim 言語リファレンスマニュアル」、2016 年

[4] 株式会社 NTT データ数理システム、「S4 Simulation System　Version 4.2 操作マニュアル」、2016 年

[5] 構造計画研究所 HP(最終アクセス 2017 年 1 月 15 日)
(http://www4.kke.co.jp/orsim/VisualSLAM_jirei.html)

[6] 構造計画研究所 HP(最終アクセス 2017 年 1 月 15 日)
(http://www.kke.co.jp/solution/theme/artisoc.html)

[7] 芹澤良、「マルチエージェントを使用したレジにおける混雑解消法の検証」東京工科大学メディア学部モデリング＆シミュレーションプロジェクト、指導教員 : 藤澤公也、2006 年 3 月

[8] 谷口豊、「Social force model を用いた災害時地下歩行空間における歩行者の避難挙動分析」、北海道大学大学院工学研究院交通インテリジェンス研究室、平成 23 年度卒業論文

[9] 平岡喬之、「自己駆動粒子系の動力学 : 群集運動を中心に」、東京大学工学系研究科、ワーキングペーパーシリーズ人工社会研究 No.44、2014 年 3 月

[10] Dirk Heling・Peter Molnar、「Social force model for pedestrian dynamics」、Physical Review E51 4282、1995

[11] 森弘隆、「瀬戸キャンパスにおける食堂の待ち行列」、南山大学数理情報学部数理科学科、指導教員 : 澤木勝茂、平成 15 年度卒業論文

付録

付録 1：エージェントの設定

≪エージェントの次の経路地点決定処理≫

```
u"""経路地点 G へ向かうための次の経路地点を返す。
次の経路地点は視野内になくてはならない。
p が指定されている場合は、その座標点を起点とする。
経路地点が存在しない場合は None を返す。"""
if p is None:
    p = self.p
g = self.agentset.env.pathgraph
paths = self.agentset.env.paths()
# mu: スケールパラメータ
# mu が大きくなると、ほぼ最適解を選ぶようになる
mu = 0.1
cs = [] # [(重み, 経路地点), ...]
# ある経路エリア内にいる場合、
for v in g.nodes_iter():
    if self.inArea(v, p):
        self.agentset.monitor.observe(now(),  self.agentid, v)

        # そこから可視な近傍経路地点が候補となる。
        for n in g.neighbors(v):
            if v == n:
                continue
            if not self.inSight(n, p):
                continue
            try:
                # v から n の距離に、n から G の最短経路を足す
                d = g[v][n]["d"] + paths[n][G]
                # (重み, 経路地点)のタプルを追加
                cs.append((numpy.exp(- mu * d), n))
            except Exception, e:
                pass
        if len(cs) == 0:
            # 経路地点が存在しない
```

```
            return None
        else:
            # 経路地点を重み付きサンプリング
            return empiricalDistribution(cs).next()
# 経路エリア外にいる場合、
for v, data in g.nodes_iter(data = True):
    # そこから可視な経路地点が候補となる。
    if self.inSight(v, p, r = self.r):
        try:
            # p から v の中心点の距離に、v から G の最短経路を足す
            s = p - data["p"]
            d = numpy.sqrt(s.dot(s)) + paths[v][G]
            # (重み, 経路地点)のタプルを追加
            cs.append((numpy.exp(- mu * d), v))
        except Exception, e:
            pass
if len(cs) == 0:
    # 経路地点が存在しない
    return None
else:
    # 経路地点を重み付きサンプリング
    return empiricalDistribution(cs).next()
```

≪エージェントのステップ処理≫

```
u"""エージェントのステップ処理を記述する。SFM に従った基本的な動作を行った後に呼
ばれる。"""

if self.isStopping(): # エージェントが停止状態(目的地に到着)
    # 次の目的地を設定する:
    if len(self.route) > 0: # ルート上の全ての地点を回ったか
        # 自分に設定されているルートから次の目的地を設定
        nextPoint = self.route.pop(0)
        self.setDestination(nextPoint)
    else:
        # エージェントを削除する: (出口から退場)
        self.agentset.remove(self)
```

```
elif self.isInErrorState(): # エラーが発生(目的地に到達不可など)
    print self.agentid, self.state.message
    # エージェントを削除する:
    self.agentset.remove(self)
else: # それ以外の場合は、目的地に移動中
    pass
#if self.agentid == 0:
#    print now(), self.state, self.p
```

≪エージェント集合の初期化処理≫

```
# 最適速度(m/s)
self.gv0 = normalDistribution(0.6, 0.6 * 0.01)

# 最高速度(m/s)
self.gv1 = normalDistribution(1.5, 1.5 * 0.01)

# 歩行者の半径(m)
self.gr = normalDistribution(0.2, 0.2 * 0.01)

# 加速時間(s)
self.gtau = normalDistribution(0.5, 0.5 * 0.01)

# 体重(kg)
self.gm = normalDistribution(50, 50 * 0.01)

# エージェントの色
self.col='r'

# 各出入口におけるエージェントの発生確率
self.genP = 0.02

# 経路地点モニター
self.monitor = TimeMonitor([u"agent",u"経路地点"], ["i","i"],name = u"経由地点")
self.simulator.addMonitor(self.monitor)
```

≪エージェント集合のステップ処理≫

```
u"""エージェント集合のステップ処理を行う。"""
# 全ての SFM エージェントを SFM のルールに従って動かす。

for i in range(0,len(self.env.gate)): # 全ての入口に対して
    if random.random() < self.genP: # 生成確率によって入店を決める

        # スタート地点を環境部品から取得
        idx = self.env.gate[i]
        start = self.env.pathgraph.node[idx]['p']

        # ルートを環境部品から取得し、目的地を取得
        route = copy.deepcopy(self.env.route[idx])
        route = random.choice(route)
        goal = route.pop(0)

        # エージェントを生成
        self.generateAgents(1,
                            # スタート地点
                            p = start,
                            # 目標経路ポイント
                            goal = goal,
                            # 最適速度(m/s)
                            v0 = self.gv0.next(),
                            # 最高速度(m/s)
                            v1 = self.gv1.next(),
                            # 歩行者の半径(m)
                            r = self.gr.next(),
                            # 加速時間(s)
                            tau = self.gtau.next(),
                            # 体重(kg)
                            m = self.gm.next(),
                            # 表示色
                            color = self.col)

        # 生成したエージェントにルートを設定
        self.agents[len(self.agents)-1].route = route
```

```
return SFMAgentSetBase.step(self)
```

≪エージェント集合の起動処理≫

```
u"""同期エージェント集合の動作を開始する。"""
# エージェント集合のプロセス
def proc():
    try:
        while True:
            # ステップ処理
            self.step()
            yield pause(self.interval)
    except Exception, e:
        fatal(u"[%s]エージェントエラー: エージェント集合のプロセスでエラーが発生しました。" % self.name,
              "error in agentset process", tb = True)
# エージェント集合のプロセスを起動
activate(proc)()
```

≪エージェント集合の可視化≫

```
u"""エージェントの描画を開始する。"""
interval = 1.0 # 表示間隔
screen = self.getAgentScreen(interval = interval,
                             xlim = (self.env.x0, self.env.x1),
                             ylim = (self.env.y0, self.env.y1))
screen.addAgentSet(self)
screen.start()
```

付録 2：環境の設定

≪環境の初期化後の処理≫

```
# 壁の厚み
d = 0.1

# 横線
lines = [
        (27,37, 42,37),
```

```
             (33,31, 36,31),
             (33,28, 36,28),
             (33,22, 36,22),
             (0,19, 7,19),
             (12,19, 27,19),
             (33,17, 36,17),
             (7,16, 12,16),
             (6,10, 9,10),
             (18,10, 21,10),
             (33,10, 36,10),
             (6,7, 9,7),
             (18,7, 21,7),
             (33,7, 36,7),
             (3,0, 11,0),
             (16,0, 26,0),
             (31,0, 42,0)
             ]
for (x0, y0, x1, y1) in lines:
      self.addObstacle(Polygon.Polygon([(x0, y0), (x0, y0-d), (x1+d, y1-d), (x1+d, y1)]))

# 縦線
lines = [(0,3, 0,14),
             (6,7, 6,10),
             (7,16, 7,19),
             (9,7, 9,10),
             (12,16, 12,19),
             (18,7, 18,10),
             (21,7, 21,10),
             (27,19, 27,37),
             (33,7, 33,10),
             (33,28, 33,31),
             (36,7, 36,10),
             (36,28, 36,31),
             (42,0, 42,12),
             (42,17, 42,27),
             (42,32, 42,37)
```

```
        ]

for (x0, y0, x1, y1) in lines:
    self.addObstacle(Polygon.Polygon([(x0+d, y0), (x0, y0), (x1, y1), (x1+d, y1)]))

# 経路ポイント作成
self.pathPoint = []

self.pathPoint.append(self.addPathPoint(4.5, 8.5, 1))
self.pathPoint.append(self.addPathPoint(7.5, 5.5, 1))
self.pathPoint.append(self.addPathPoint(7.5, 11.5, 1))
self.pathPoint.append(self.addPathPoint(10.5, 8.5, 1))
self.pathPoint.append(self.addPathPoint(16.5, 8.5, 1))
self.pathPoint.append(self.addPathPoint(19.5, 5.5, 1))
self.pathPoint.append(self.addPathPoint(19.5, 11.5, 1))
self.pathPoint.append(self.addPathPoint(22.5, 8.5, 1))
self.pathPoint.append(self.addPathPoint(31.5, 8.5, 1))
self.pathPoint.append(self.addPathPoint(34.5, 5.5, 1))
self.pathPoint.append(self.addPathPoint(34.5, 11.5, 1))
self.pathPoint.append(self.addPathPoint(37.5, 8.5, 1))
self.pathPoint.append(self.addPathPoint(31.5, 29.5, 1))
self.pathPoint.append(self.addPathPoint(34.5, 26.5, 1))
self.pathPoint.append(self.addPathPoint(34.5, 32.5, 1))
self.pathPoint.append(self.addPathPoint(37.5, 29.5, 1))
self.pathPoint.append(self.addPathPoint(1, 1, 1))
self.pathPoint.append(self.addPathPoint(1, 16.5, 1))
self.pathPoint.append(self.addPathPoint(13.5, 1, 1))
self.pathPoint.append(self.addPathPoint(28.5, 1, 1))
self.pathPoint.append(self.addPathPoint(40.5, 14.5, 1))
self.pathPoint.append(self.addPathPoint(40.5, 29.5, 1))
self.pathPoint.append(self.addPathPoint(40.5, 1.5, 1))
self.pathPoint.append(self.addPathPoint(1.5, 4.5, 1))
self.pathPoint.append(self.addPathPoint(1.5, 12.5, 1))
self.pathPoint.append(self.addPathPoint(3.5, 17.5, 1))
self.pathPoint.append(self.addPathPoint(4.5, 1.5, 1))
self.pathPoint.append(self.addPathPoint(5.5, 14.5, 1))
```

```
self.pathPoint.append(self.addPathPoint(9.5, 1.5, 1))
self.pathPoint.append(self.addPathPoint(13.5, 14.5, 1))
self.pathPoint.append(self.addPathPoint(13.5, 17.5, 1))
self.pathPoint.append(self.addPathPoint(17.5, 1.5, 1))
self.pathPoint.append(self.addPathPoint(20.5, 17.5, 1))
self.pathPoint.append(self.addPathPoint(24.5, 1.5, 1))
self.pathPoint.append(self.addPathPoint(28.5, 17.5, 1))
self.pathPoint.append(self.addPathPoint(28.5, 26.5, 1))
self.pathPoint.append(self.addPathPoint(28.5, 35.5, 1))
self.pathPoint.append(self.addPathPoint(32.5, 1.5, 1))
self.pathPoint.append(self.addPathPoint(34.5, 15.5, 1))
self.pathPoint.append(self.addPathPoint(34.5, 19.5, 1))
self.pathPoint.append(self.addPathPoint(34.5, 23.5, 1))
self.pathPoint.append(self.addPathPoint(34.5, 35.5, 1))
self.pathPoint.append(self.addPathPoint(38.5, 3.5, 1))
self.pathPoint.append(self.addPathPoint(40.5, 6, 1))
self.pathPoint.append(self.addPathPoint(40.5, 10.5, 1))
self.pathPoint.append(self.addPathPoint(40.5, 18.5, 1))
self.pathPoint.append(self.addPathPoint(40.5, 25.5, 1))
self.pathPoint.append(self.addPathPoint(40.5, 32.5, 1))
self.pathPoint.append(self.addPathPoint(40.5, 35.5, 1))
self.pathPoint.append(self.addPathPoint(1.5, 8.5, 1))
self.pathPoint.append(self.addPathPoint(9.5, 14.5, 1))
self.pathPoint.append(self.addPathPoint(17.5, 17.5, 1))
self.pathPoint.append(self.addPathPoint(24.5, 17.5, 1))
self.pathPoint.append(self.addPathPoint(26.5, 5.5, 1))
self.pathPoint.append(self.addPathPoint(28.5, 9.5, 1))
self.pathPoint.append(self.addPathPoint(36.5, 1.5, 1))
self.pathPoint.append(self.addPathPoint(28.5, 22.5, 1))
self.pathPoint.append(self.addPathPoint(28.5, 31.5, 1))

# 出入口経路 ID
self.gate = [16,17,18,19,20,21]

# 売り場の代表点となる経路ポイント ID
self.buyPoint = [25,26,31,32,35,39,41,42,46,49,50,54]
```

```
# レジとなる経路ポイント ID
self.register = [0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15]

# 経路候補
self.route = {}
self.route[16] = [[26,31,54,32,35,41,46,15,21],[26,31,54,32,35,41,46,15,21]]
self.route[17] = [[25,50,32,35,39,42,54,8,19],[10,26,31,54,39,41,35,12,21]]
self.route[18] =
[[31,42,46,39,32,50,25,2,17],[26,10,50,32,54,42,39,10,19],[50,32,39,46,41,35,42,11,20]]
self.route[19] =
[[54,32,50,25,10,26,31,5,18],[31,50,32,35,41,46,54,7,18],[42,39,32,50,31,26,10,0,16]]
self.route[20] =
[[39,32,54,31,26,10,25,3,17],[46,41,35,32,50,31,26,1,16],[42,54,32,39,46,41,35,14,21]]
self.route[21] =
[[46,39,32,32,50,10,26,4,16],[41,35,39,42,31,50,32,6,18],[41,35,32,50,26,31,42,9,19]]

# 可視な経路ポイント同士を自動接続
self.connectPathGraphAuto()
```

≪環境の描写処理≫

```
screen = panel.screen
backscreen = panel.backscreen
if now() == 0 or self.backscreen_redraw:
    backscreen.clear()
    backscreen.set_axis_off()
    (xmin, xmax, ymin, ymax) = self.space.boundingBox()
    backscreen.lines(((xmin, ymin),
                      (xmin, ymax),
                      (xmax, ymax),
                      (xmax, ymin),
                      (xmin, ymin)),
                     color = "k")
    bb = Polygon.Shapes.Rectangle(xmax - xmin, ymax - ymin)
    bb.shift(xmin, ymin)
    for tristrips in (bb - self.space).triStrip():
```

```
        for i in xrange(len(tristrips) - 2):
            backscreen.polygon(tristrips[i:(i+3)],
                               closed = True, color = "k")
    # 経路グラフ表示
    if self.dispP:
        for (v, data) in self.pathgraph.nodes_iter(data = True):
            backscreen.ellipse(data["x"], data["y"],
                               data["r"] * 2, data["r"] * 2,
                               color = "g", alpha = 0.3)
            backscreen.text(data["x"], data["y"], "%d" % v)
        for (u, v) in self.pathgraph.edges_iter():
            p0 = self.pathgraph.node[u]["p"]
            p1 = self.pathgraph.node[v]["p"]
            backscreen.line(p0[0], p0[1], p1[0], p1[1],
                            color = "g", alpha = 0.3)
    backscreen.set_xlim((self.x0, self.x1))
    backscreen.set_ylim((self.y0, self.y1))
    backscreen.draw()
    self.backscreen_redraw = False
```

≪環境上のエージェントの描写処理≫

```
screen = panel.screen
# 歩行者半径表示
if self.dispR:
    for agent in agents:
        screen.ellipse(agent.p[0], agent.p[1],
                       agent.r * 2, agent.r * 2,
                       color = agent.screenColor,
                       edgecolor = agent.screenEdgeColor,
                       alpha = agent.screenAlpha)
        # 歩行者目的地表示
        goal = agent.getDestination()
        if goal is not None:
            #screen.text(agent.p[0], agent.p[1],
            #            "%s" % goal)
            screen.text(agent.p[0], agent.p[1],
```

```
                            "%s" % agent.agentid)
                # エージェント番号を表示したい場合の例:
                # "%s" % agent.agentid
# 速度ベクトル表示
if self.dispV:
    for agent in agents:
        v = agent.v * 2
        screen.arrow(agent.p[0], agent.p[1],
                     v[0], v[1],
                     color = "r",
                     edgecolor = "r")
# 歩行者間外力表示
if self.dispF:
    for agent in agents:
        if len(agent.fs) > 0:
            f = sum(agent.fs)
        else:
            f = numpy.array((numpy.nan, numpy.nan))
        screen.arrow(agent.p[0], agent.p[1],
                     f[0], f[1],
                     color = "y",
                     edgecolor = "y")
# 障害物外力表示
if self.dispW:
    for agent in agents:
        if len(agent.fwalls) > 0:
            w = sum(agent.fwalls)
        else:
            w = numpy.array((numpy.nan, numpy.nan))
        screen.arrow(agent.p[0], agent.p[1],
                     w[0], w[1],
                     color = "c",
                     edgecolor = "c")
```