# Simplest rules

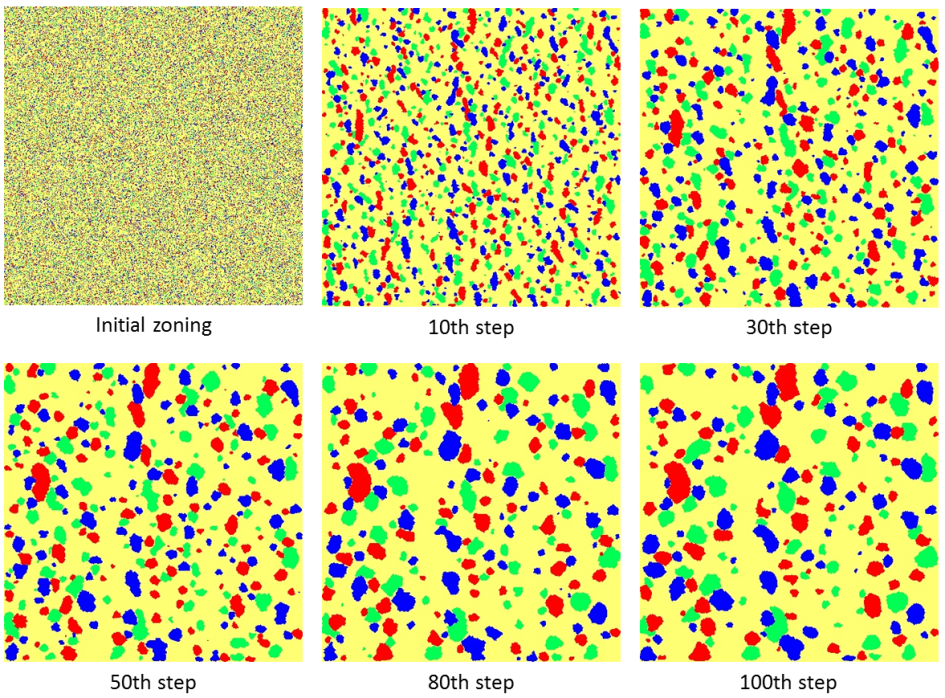
最もシンプルなルールで、proceduralにゾーニングを生成してみよう。ゾーンタイプは、residential (70%)、commercial (10%)、industrial (10%)、park (10%)の4種類を使用した。ゾーンの比率は、デザイナが指定したものということにする。また、Cityサイズは、グリッドのサイズは、を使用した。最初に、上の比率に基づいてランダムにゾーンを生成した後、以下のアルゴリズムによる更新を一定回数（実験では100回）繰り返す。

**ルール：**

|  |  |
| --- | --- |
| 条件 | 変更後のゾーンタイプ |
|  | industrial |
|  | commercial |
|  | park |
| Otherwise | residential |

ただし、は、隣接8セルの中で、ゾーンタイプがのセルの数、は、比率に基づくゾーンタイプのセル数の期待値に対する不足数を表す。例えば、の場合、このタイプのセルが期待値より1個少ないことを示す。このパラメータに基づくロジスティック関数により、期待値より少ないタイプに変わりやすいルールとなり、ゾーンタイプの分布が期待値に近づくようバランスを取ることができる。

**結果：**以下に、初期ゾーン、10、30、50、80、100回目のゾーンを示す。

****

上記のルールのせいで、基本的にはクラスタを形成する方向に進化していくことが分かる。これはこれで、面白い結果だ。

# Simplest rules and Accessibility

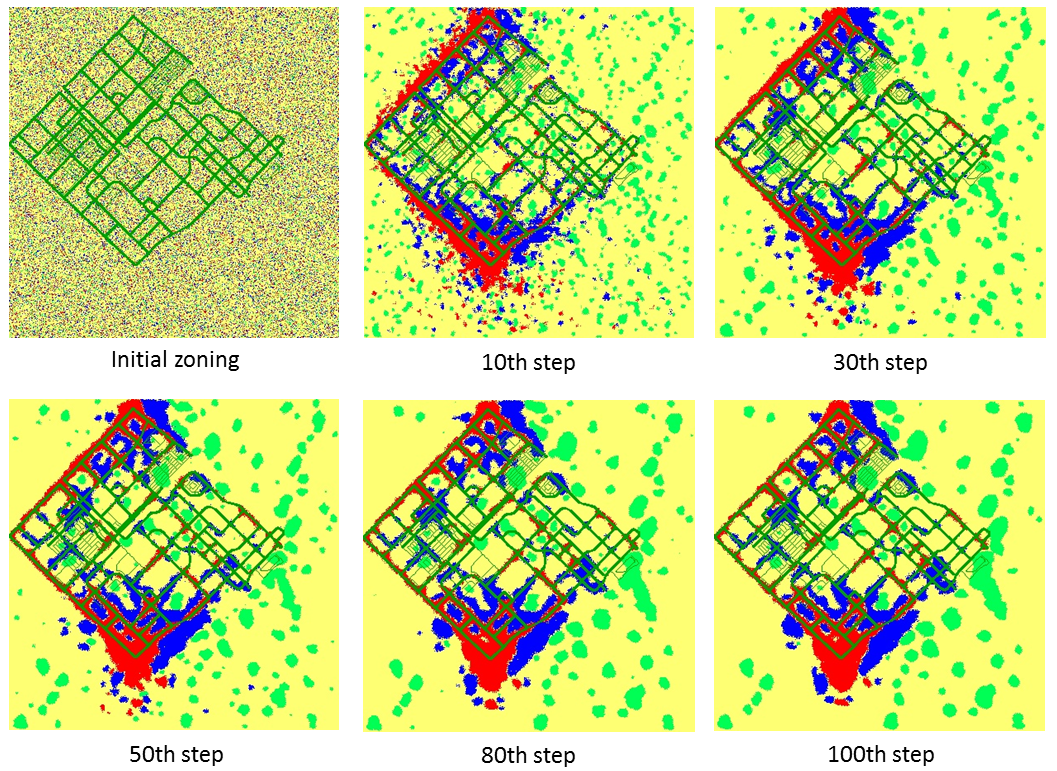
「Real and virtual urban design」のスライドでも述べられている通り、メイン道路へのアクセスは、店をオープンする場所を決定する上で重要な要素となる。というわけで、以下のようにルールを少し変更してみた。

**ルール：**

|  |  |
| --- | --- |
| 条件 | 変更後のゾーンタイプ |
|  | industrial |
|  | commercial |
|  | park |
| Otherwise | residential |

ただし、は、メイン道路へのaccessbilityで、以下の式で算出している（）。

**結果：**以下に、初期ゾーン、10、30、50、80、100回目のゾーンを示す。

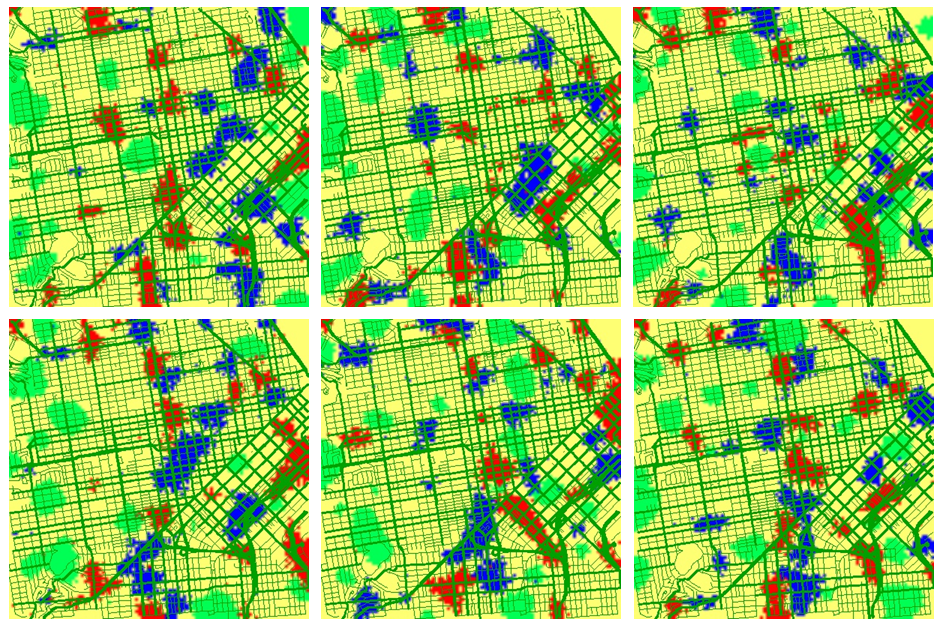


とりあえず、道路沿いにcommercial、industrialゾーンが集中するというパターンは作れた。しかし、画面中央に変な境界が見える。バグか？

また、もっとランダム性が必要だ。さもないと、多様なゾーンパターンを探索できない。

# Variation

異なる初期乱数シードにより、様々なゾーニングが生成されることが望ましい。ある程度ルールに従っていながら、且つ、バラエティに富んだアウトプットが理想だ。100個のゾーニングを生成してみた。似ているとはいえ、それなりにバラエティがあるかな。



# People Allocation

ユーザをクラスタリングしなかった時は、people allocationのgreedyアルゴリズムは以下の通り。

1 num\_cells 住宅ゾーンの数

2 **while** num\_cells > 0

3 **for** user u

4 **while** occupied[all\_scores[u][pointer[u]].cell]

5 pointer[u]++;

6 **end while**

7 occupied[all\_scores[u][pointer[u]].cell] = true

8 score += all\_scores[u][pointer[u]].score

9 num\_cells--

10 **end** **for**

11 **end** **while**

しかし、ユーザをクラスタリングした場合、各クラスタの重みを考慮しながらpeople allocationを実施する必要がある。というわけで、アルゴリズムを以下のように修正した。

1 num\_cells 住宅ゾーンの数

2 **while** num\_cells > 0

3 **for** user u

4 **while** occupied[all\_scores[u][pointer[u]].cell] >= 1

5 pointer[u]++;

6 **end while**

7 occupied[all\_scores[u][pointer[u]].cell] += weight[u]

8 score += all\_scores[u][pointer[u]].score \* weight[u]

9 **if** occupied[all\_scores[u][pointer[u]].cell] >= 1

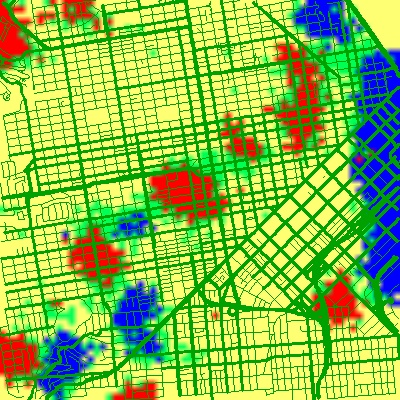
10 num\_cells—

11 **end if**

12 **end** **for**

13 **end** **while**

というわけで、**500個**のゾーニング（64x64グリッド）を生成し、それぞれpeople allocationしてスコアを計算し、ベストを探してみた。



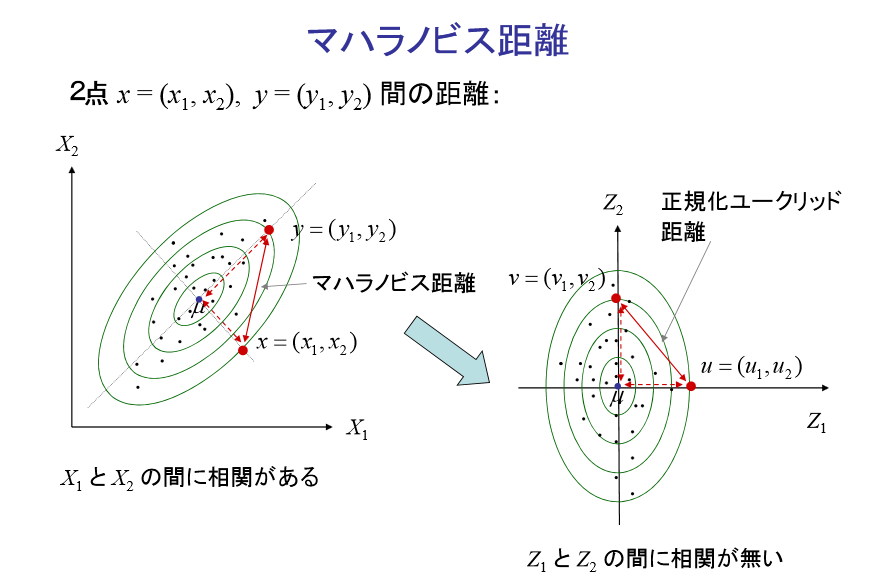
なお、性能測定結果は以下の通り。意外なことに、people allocationには時間を要していない。一方、propertyベクトルの計算に時間がかかっている。Distance mapの計算に時間がかかっているということだ。差分のみを計算するというわけにいかないので、この時間を短縮する方法は今のところ思いつかない。CUDAでやるには、priorityキューをCUDAで実装できないため、無理だということは、以前調べた通りだ。後、PMによるゾーニング生成も時間がかかっている。これについても、要検討だ。

|  |  |
| --- | --- |
| 処理内容 | 合計時間 |
| 初期化 | 0 秒 |
| PMによるゾーニング生成（それぞれ40ステップ） | 15.586 秒 |
| 各セルのpropertyベクトルの計算 | 12.781 秒 |
| People allocationとスコア計算 | 0.5 秒 |

# ユーザのClustering

ユーザのpreferenceベクトルをクラスタリングする。各要素ごとに、分散が大きく異なる可能性があるため、単純なベクトルの距離を使ったクラスタリングは良くないと思う。おそらく、マハラノビス距離を使ってクラスタリングするのが良いだろう。クラスタリング自体は、K-meansでOkだろう。ただし、K-meansは、クラスタ数とクラスタの初期位置に大きく依存するため、何回か試して、最も***BIC(Bayesian information criterion)***が低いものを選ぶなどの工夫が必要になる。

マハラノビス距離のイメージは、以下の感じ。正規化ユークリッドに似ているが、各要素間に相関がある場合、マハラノビス距離は直感的により良い距離を与える。



クラスタリングのアルゴリズムは以下の通り。

1 ユークリッド距離で個にクラスタリング

2 **while** true

3 **for** **each** cluster

4 cv::calcCoverMatrix(に属するポイント, )

5 invert()

6 **end** **for**

7 **for** **each** point

8 min\_dist

8 **for** **each** cluster

9 dist cv::Mahalanobis(, , )

10 if dist < min\_dist

11 min\_dist dist

12 min\_c

13 **end** **for**

14をクラスタ**min\_c**に属させる

15 **end for**

16 もし、クラスタに更新がなければ、終了

17 **end while**