~~ネットワークモデル~~の基礎
利用者均衡配分

井料隆雅
東京大学工学系研究科（客員）
東北大学情報科学研究科（本務）

講師紹介

- 井料隆雅（いりょう　たかまさ）
  - 東京大学大学院工学系研究科
    特定客員大講座担当教授（兼務：今日はこっち）
  - 東北大学大学院情報科学研究科
    人間社会情報科学専攻 教授（本務：日頃はこれ）
  - 神戸大学大学院工学研究科客員教授（たまに）
    - R2/3まで：神戸大学大学院工学研究科市民工学専攻 教授

交通量配分

- 所与の起終点交通量を所与のネットワーク上に割り当てる（＝配分する）
- 交通需要を交通ネットワークがどのように処理するのかを計算する.
- さまざまな施策の効果を評価できる.
  - 最適化問題の制約条件にすることにより（MPEC/Bi-level）最適な施策を計算できる.

交通量配分の計算のポイント

- ドライバー（利用者）の行動は交通状況に依存する.
- 交通状況はドライバーの行動に依存する.
- ∴ドライバーの行動と交通状況は相互作用する.
  - –ドライバー同士の行動が相互作用する，ともいえる.

この「相互作用」をどうモデルかし，どう解くかが
交通量配分の最大のポイントである.

※ 相互作用があるシステムでは，需要と交通行動がわかっても，
正しい将来予測や施策評価はできない.

相互作用の逐次計算

- 交通シミュレーションの例
  - 1台1台の車に，ネットワーク上での挙動のルールを与えて，そのとおりに動かす．
  - 挙動のルールが「過去から今現在までの交通状況のみに依存する」であれば，各車両の挙動を，時間軸方向に沿って，逐次計算すればよい．
    - ある時間までの車両の挙動の計算結果が，その時間における各車両の挙動を決定し，それによって，直後の時間までの交通状況を計算できる．
    - 数学的に言えば常微分方程式を解いていることになる．

ドライバーの将来予測？

- ドライバーは，過去の経験に基づいて，交通状況の将来予測が可能であろう.
  - 頻繁に道路を使っていれば，どの経路がより早く目的地に着くかについて，経験に基づく戦略を持つことになる，と考えられる.
  - 予測を強いられることもある：到着時刻制約があれば，所要時間を予測して出発時刻を決めなくてはならない.
- 将来予測が可能である／必要とすれば，時系列に沿った逐次的な計算はできなくなる.

Day-to-dayの繰り返し計算

- 繰り返し道路を使用するドライバーの学習過程もシミュレーションしてしまえばよい？
  - 複数日のシミュレーションを繰り返す.
  - ドライバーは，過去の日の混雑状況を参照する.
  - 今日≒昨日としてもよいし，学習モデルを考えてもよい.
- この考え方自体は正しい.
- 繰り返し計算を行った「結果」だけを先回りして計算することはできないか？

均衡状態

- どのドライバーも，他のドライバーの行動が不変であれば，現在の選択よりもよい選択肢を持たない．
  - この状態が一旦成立すれば，これ以降，どのドライバーも，選択肢を変える動機付けを持たないので，その状態が長期的に安定して実現することが”期待”できる．
- 均衡状態
  - 上記の定義は「Nash均衡」に相当する．
  - Wardropの第1原理も概ね同じ意味（厳密には異なる）
  - 均衡状態を条件とする交通量配分：利用者均衡配分

均衡状態の計算

- Day-to-dayの繰り返し計算をそのまま実行
  - もっとも素朴な方法
  - 計算時間がかかる.
- 不動点問題で定式化し，それに対する解法を適用
  - 比較的多くの一般的な問題に適用可能
- 等価最適化問題で定式化する
  - もっとも利便性が高い.
  - 適用可能な問題は限られる.

最適化問題での定式化

- 静的交通量配分
  - 交通流を連続流として扱い，時間変化を考慮しない．
- リンク旅行時間は，リンク交通量の単調増加関数として示せる．BPR関数が有名．
- 均衡状態を示す条件と等価な最適化問題が存在

等価最適化問題

この最適化問題の
KKT条件が，
Wardropの第1原理と
等価となる

$$\text{Min.} \sum_{l \in E} \int_0^{x_l} t_l(w)\mathrm{d}w$$

リンク旅行時間

$$\text{subject to:}$$

OD交通量 → $$q_i = \sum_{r \in R_i} f_r \quad \text{for all } i \in \Omega$$

リンク交通量 → $$x_l = \sum_{i \in \Omega} \sum_{r \in R_i} e_{lr} f_r \quad \text{for all } l \in E$$

経路交通量 → $$f_r \geq 0 \quad \text{for all } r \in R_i, i \in \Omega$$

Beckmann et al. 1956 イェール大のWebサイトで全文公開されている
https://cowles.yale.edu/sites/default/files/files/pub/misc/specpub-beckmann-mcguire-winsten.pdf

等価最適化問題の解釈

- 目的関数は「ポテンシャル関数」と解釈できる.
  - 勾配(grad, ∇)をとったら何が出てくるか？
- ポテンシャル関数が最小となる点が均衡状態
  - 交通量保存則を制約条件とした最適化問題
- ゲーム理論では「ポテンシャルゲーム」と呼ぶ.
  - いろいろと性質がよい.
  - 均衡状態の「安定性」が保証される
    Day-to-dayの繰り返し計算が必ず収束する.
    (例えば, Sandholmの教科書(2010))を参照.

最適化問題の解法

- Frank-Wolfe法を用いる方法が一般的
  - 最速ではないが，実用性があり，簡単に実装できる.
  - LeBlanc et al. (1975)
- 最短経路探索（一般にはDijkstra法を用いる）と1次元探索（黄金分割法が多用される）を組み合わせる.
  - 利用者均衡配分問題では，FW法のLP近似問題の部分を，最短経路探索問題に置き換えられる.
    これによって計算が高速化されている.

Frank-Wolfe法

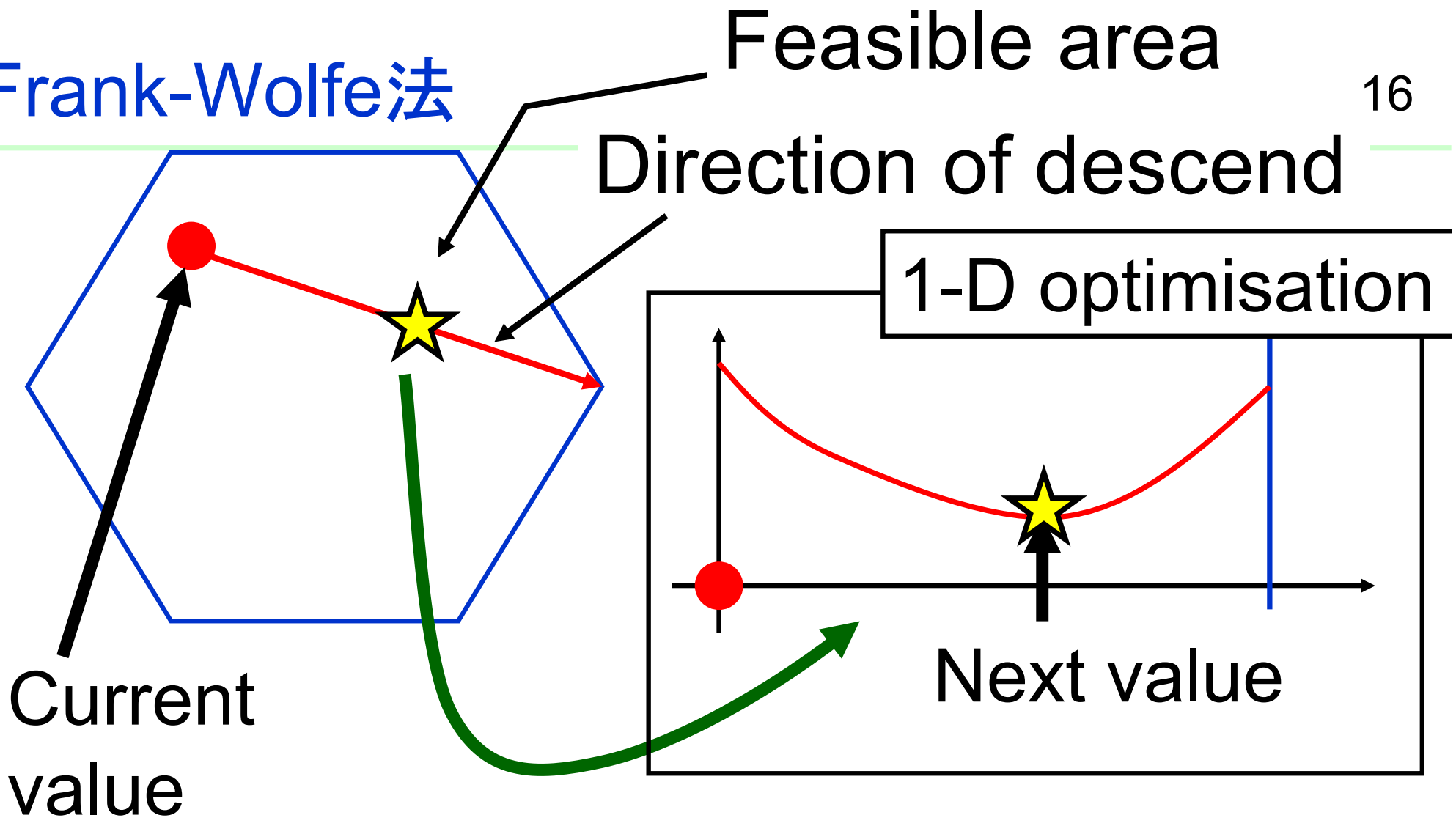
- $\mathbf{x}_n$:n回目の繰り返し計算でのリンク交通量パターン
- Step 1:　All-or-nothing配分
  - OD間の最短経路を計算し，各ODのOD交通量の全量をその最短経路に配分し，そのときのリンク交通量パターン $\mathbf{y}$ を計算する.
    - 最短経路探索にはDijkstra法を用いることが多い．Dijkstra法は1つの起点から全ノードまでの最短経路をまとめて計算するので，All-or-nothing配分は，起点ごとにまとめて計算するのがよい.
    - 最短経路を記憶する必要がないことに注意！　各ODのOD交通量を各リンクに配分したら，そのODの最短経路は忘れて良い.

Frank-Wolfe法

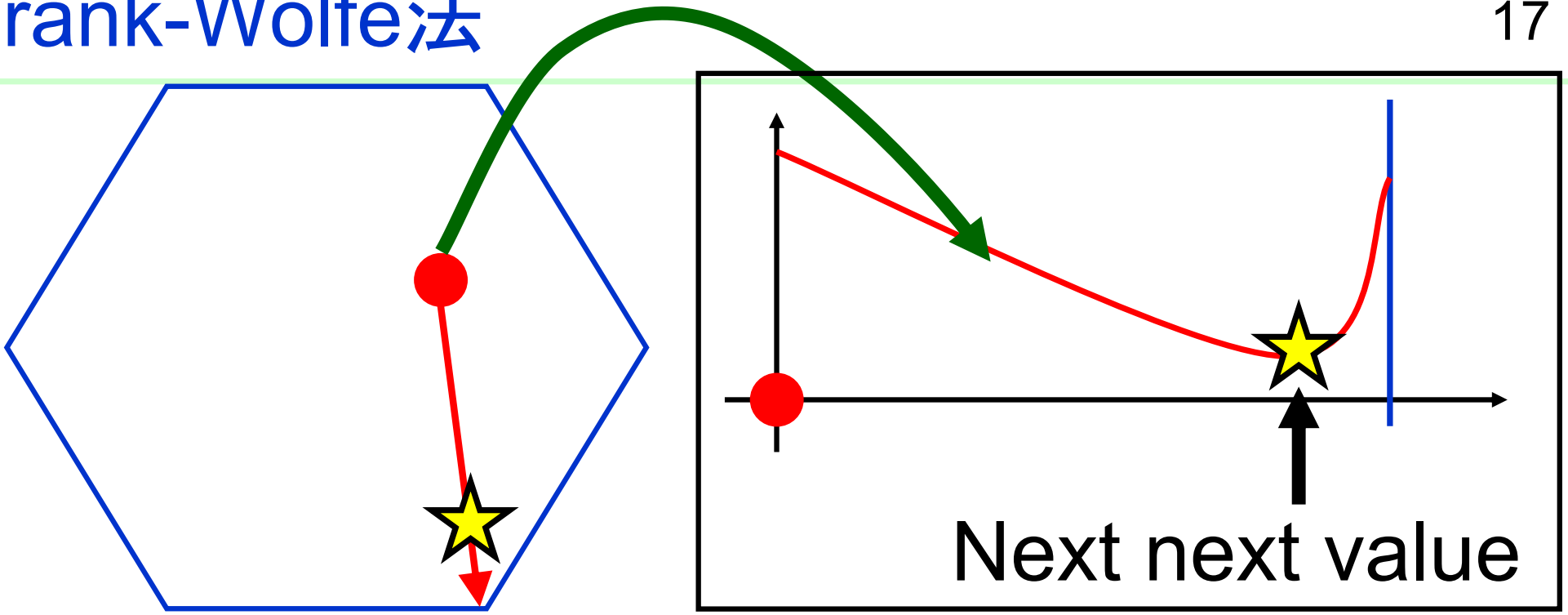
- Step 2:1次元探索
  - $\mathbf{x}_{n+1}=(1-\alpha)\,\mathbf{x}_n+\alpha\mathbf{y}$ とし，最適化問題の目的関数 $Z(\mathbf{x}_{n+1})$を最小化する 0以上1以下のαを計算する.
  - 1次元探索の方法は何でも構わないが，黄金分割法を用いれば効率的に計算できる.

Step1と2を繰り返す.

Frank-Wolfe法



Frank-Wolfe法



Frank-Wolfeの実装


```
68  # FWをとりあえず5回繰り返す FW is going to repeat for five times here.
69  for k in range(1,5):
70      # 経路旅行時間を計算（本来のFWであれば，リンク旅行時間を計算してから最短経路探索をする）
71      # Calculate route travel time, note that it must be replaced by the shortest path search
72      # with link travel times calculated.
73      route_TT = calcRouteTT(link_traf_x)
74      minTT = 10000000
75      minRoute = ''
76      for route in route_TT:
77          if minTT > route_TT[route]:
78              minTT = route_TT[route]
79              minRoute = route
80      print ("Shortest route: ",minRoute,minTT)
81
82      # All-or-nothing 配分
83      link_traf_y = calcLinkTraf(
84              {'a':OD_traf*('a'==minRoute),'b':OD_traf*('b'==minRoute),'c':OD_traf*('c'==minRoute)}
85          )
86      print ("All-or-nothing assigmnent: ",link_traf_y)
```

Step 1

最短経路探索は
経路列挙で行なっている

Frank-Wolfeの実装


```
88    # 一次元探索（単純な方法による）
89    # One-dimension search (Naive method)
90    minObj = 100000000
91    minAl = 0
92    for alpha in range(0,30001):
93        al = alpha / 30000
94        link_mix = {}
95        for link in links:
96            link_mix[link] = al * link_traf_y[link] + (1-al) * link_traf_x[link]
97        obj = calcObjUE(link_mix)
98        if (minObj > obj):
99            minObj = obj
100           minAl = al
101
102   print ("1D search alpha = ",minAl," Obj = ",minObj)
103
104   # 一次元探索の最適解を用いて，リンク交通量を更新
105   # Update link traffic volume pattern using the optimal solution of 1-D search.
106   for link in links:
107           link_traf_x[link] = minAl * link_traf_y[link] + (1-minAl) * link_traf_x[link]
108
109   print ("Current link traffic volume: ",link_traf_x)
110   print ()
```

Step 2

1次元探索は
線形探索で行なっている

MSA

- Method of Successive Average
- Frank-Wolfeのステップ2の α を，目的関数の評価を行うことなく「適当に」求める.
  - 適当というが，いちおう一定の方法はある.
  - $\alpha = 1/n$ と設定することが多い　詳細はSheffiの教科書でも
  - α を十分小さい値に固定してもよい.
    これは，Best response dynamicsと呼ばれるDay-to-dayのドライバーの行動調整モデルに相当する.

MSA

- 目的関数の計算が難しい場合に利用できる.
  - SUEの目的関数は経路交通量が入る
    - 経路なしの定式化も可能ではある（赤松他1991）
  - 最適化問題に変換できない問題でもいちおう使える.
    - この場合は収束性が保証されないことに注意.

均衡状態の一意性と安定性

Day-to-dayの繰り返し計算 8

- 繰り返し道路を使用するドライバーの学習過程もシミュレーションしてしまえばよい？
  - 複数日のシミュレーションを繰り返す.
  - ドライバーは，過去の日の混雑状況を参照する.
- この考え方自体は正しい.
- 繰り返し計算を行った「結果」だけを先回りして計算することはできないか？

ここは正しい
（学習過程のモデルが正しければ）

ここは，本当に
それでいいのか？

均衡状態の一意性と安定性

- 均衡状態が複数ある場合は，どこに収束するかがわからない．計算の初期状態に依存する．
- 安定な均衡状態がなければ，そもそも均衡状態を将来実現する状態とすることはできない．

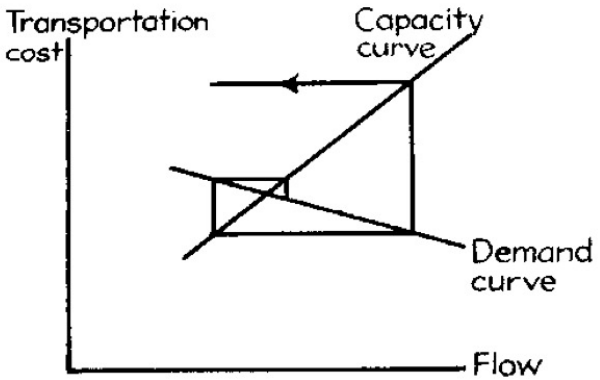


Figure 3.1. Unstable Case

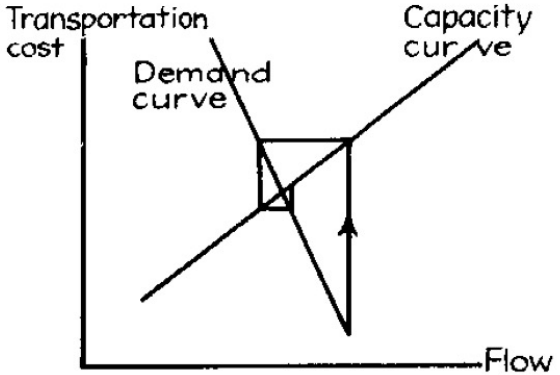
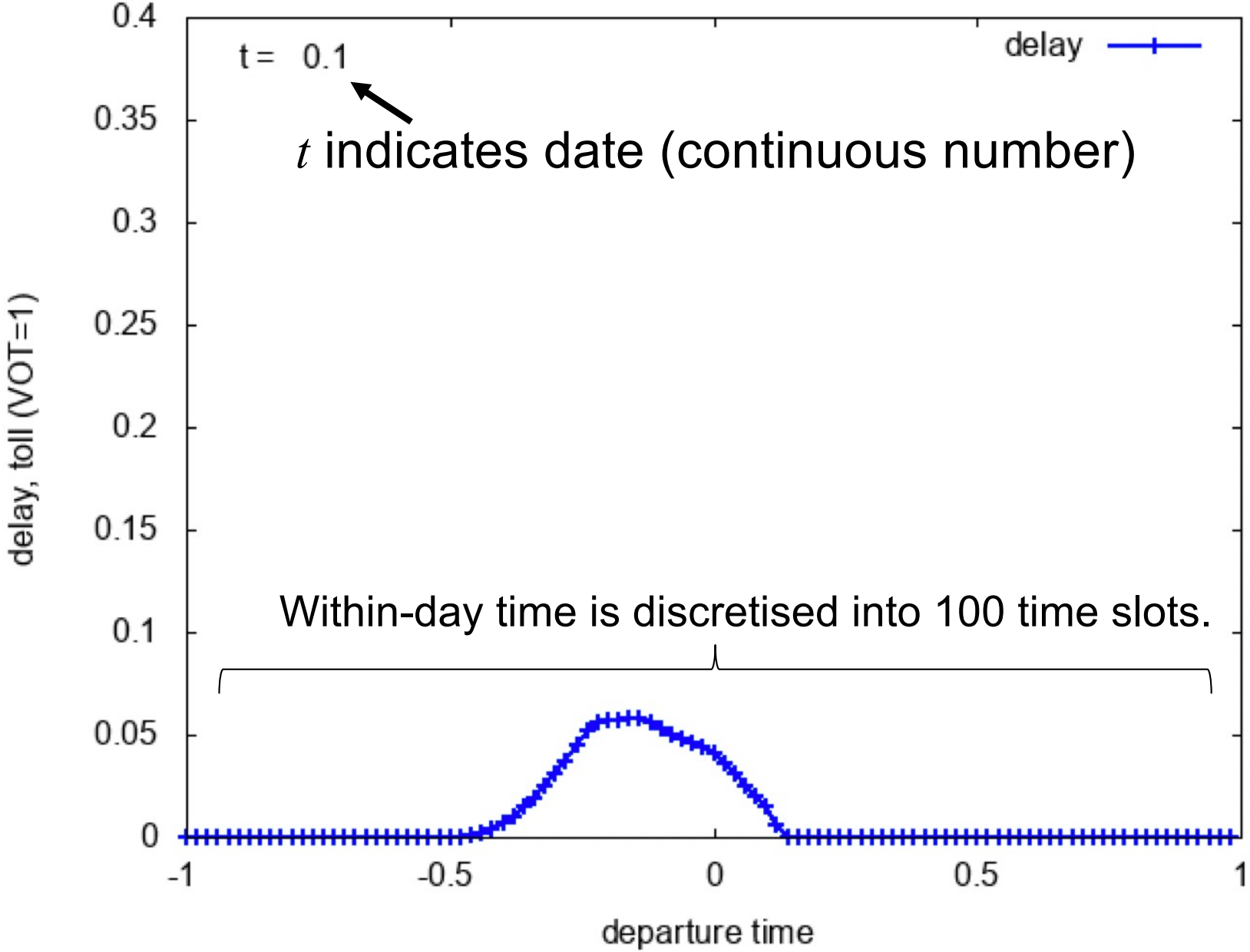


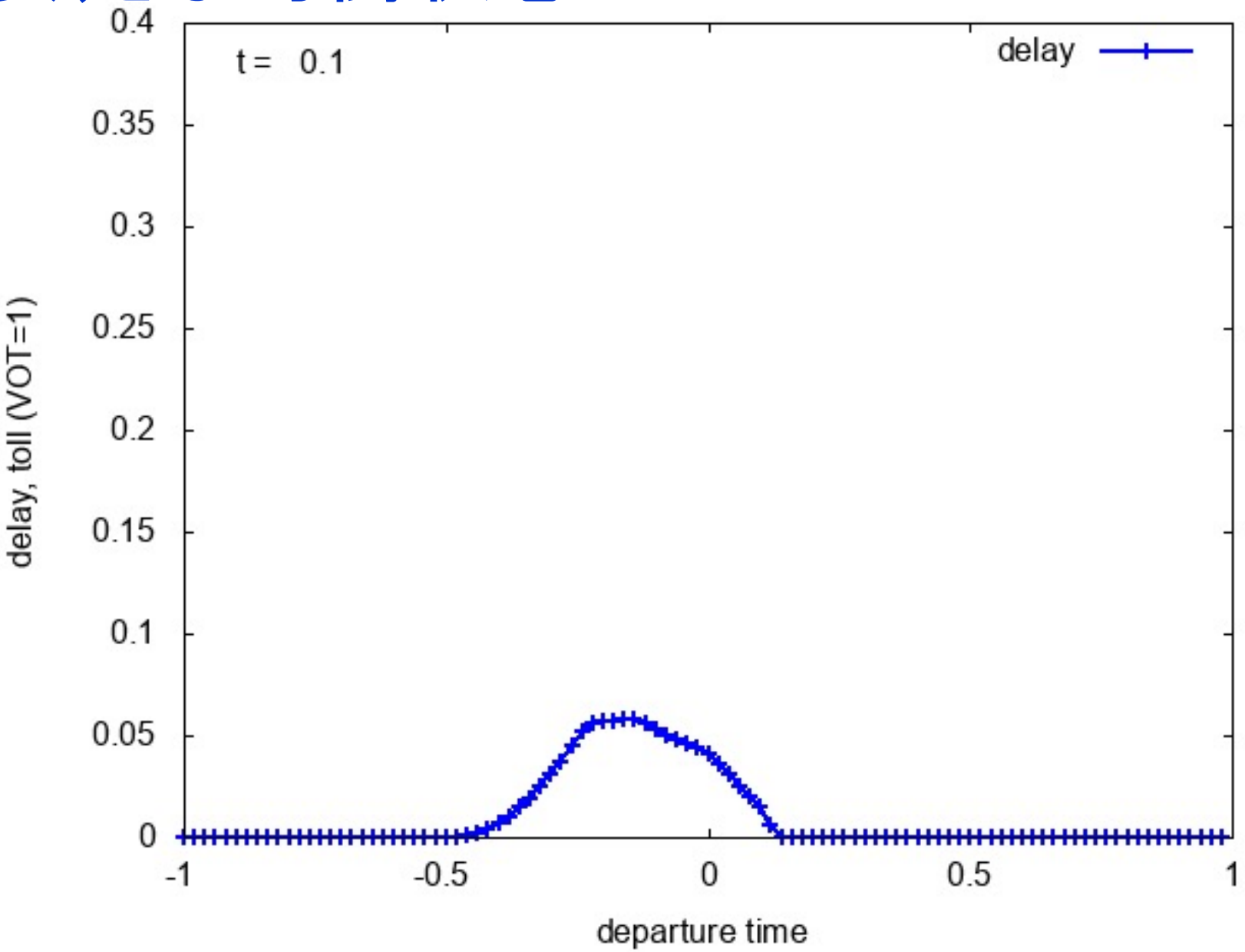
Figure 3.2. Stable Case

Beckmann et al. 1956　https://cowles.yale.edu/sites/default/files/files/pub/misc/specpub-beckmann-mcguire-winsten.pdf

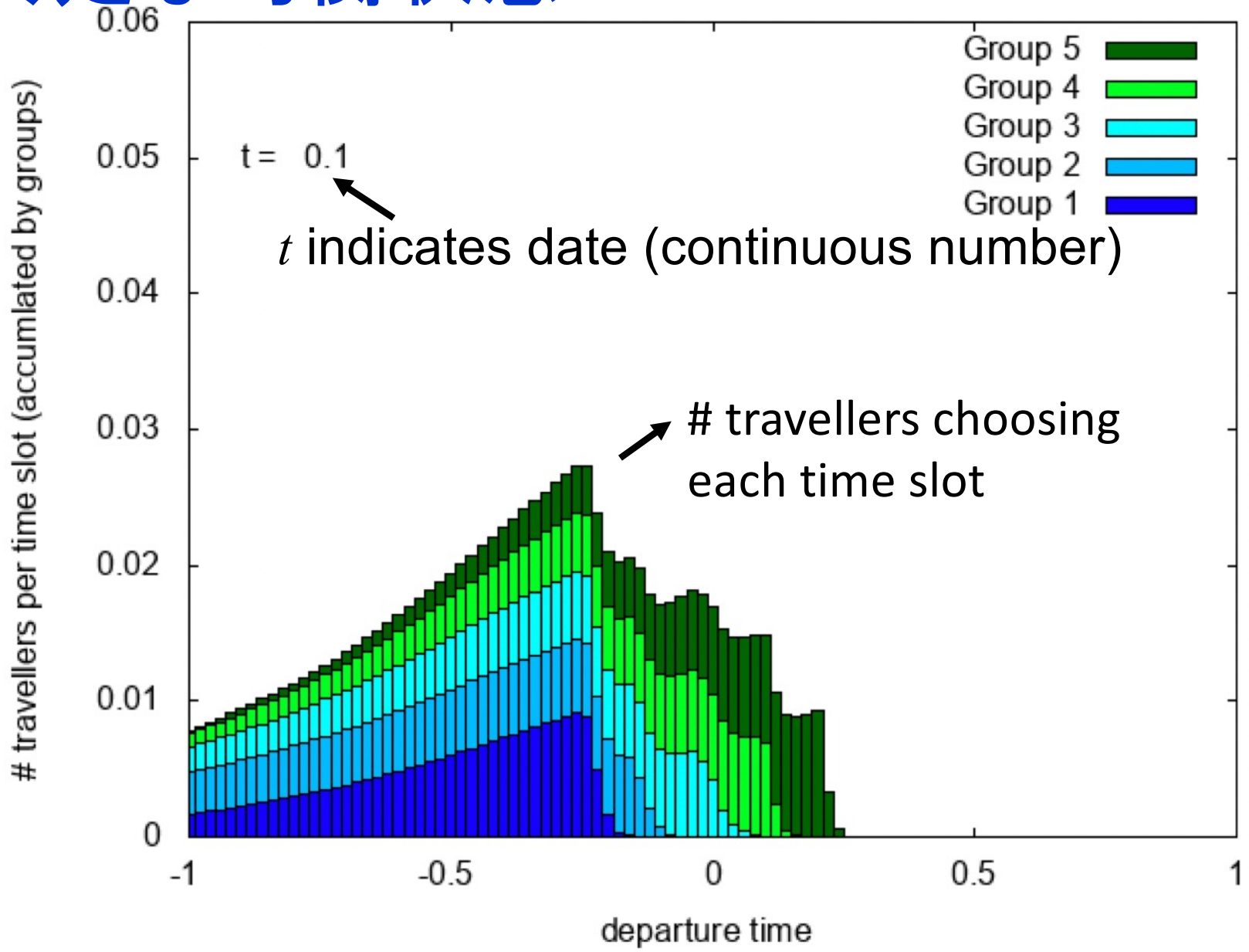
不安定な均衡状態


t = 0.1
delay
t indicates date (continuous number)
Within-day time is discretised into 100 time slots.
delay, toll (VOT=1)
departure time
0.4
0.35
0.3
0.25
0.2
0.15
0.1
0.05
0
-1
-0.5
0
0.5
1

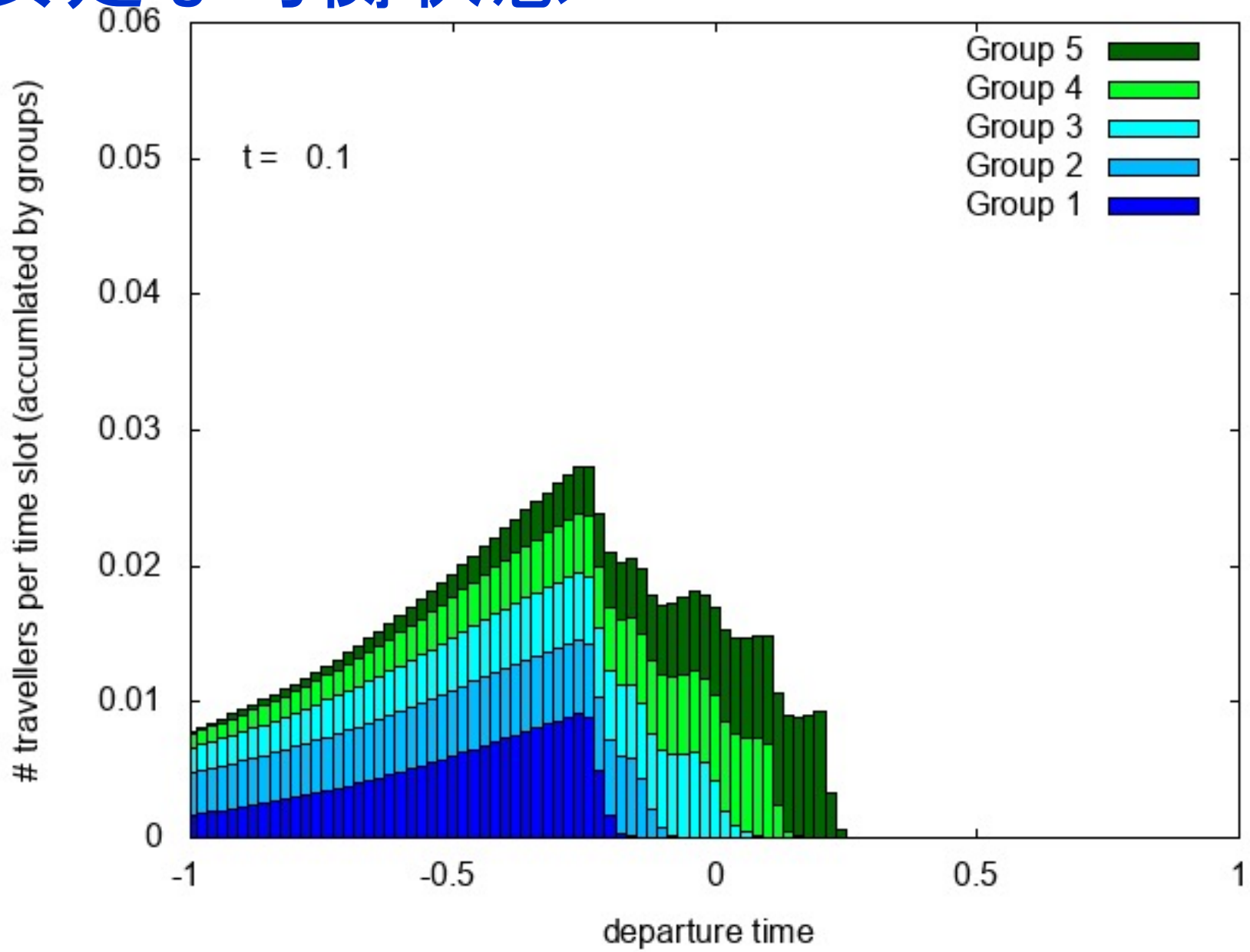
不安定な均衡状態


t = 0.1
delay
delay, toll (VOT=1)
departure time
0.4
0.35
0.3
0.25
0.2
0.15
0.1
0.05
0
-1
-0.5
0
0.5
1

不安定な均衡状態


0.06
0.05
0.04
0.03
0.02
0.01
0
# travellers per time slot (accumulated by groups)
t = 0.1
Group 5
Group 4
Group 3
Group 2
Group 1
t indicates date (continuous number)
travellers choosing each time slot
-1
-0.5
0
0.5
1
departure time

不安定な均衡状態


0.06
0.05
0.04
0.03
0.02
0.01
0
t = 0.1
Group 5
Group 4
Group 3
Group 2
Group 1
travellers per time slot (accumlated by groups)
-1
-0.5
0
0.5
1
departure time

複雑な相互作用の均衡

Transportation Research Part B 126 (2019) 87–114

Contents lists available at ScienceDirect

Transportation Research Part B

journal homepage: www.elsevier.com/locate/trb

Properties of equilibria in transport problems with complex interactions between users

Takamasa Iryo[a,*], David Watling[b]

[a] Graduate School of Engineering, Kobe University, 1-1, Rokkodai, Nada, Kobe 657-8501, Japan
[b] Institute for Transport Studies, University of Leeds, Leeds LS2 9JT, United Kingdom

ARTICLE INFO

Article history:
Received 28 June 2018
Revised 23 January 2019
Accepted 13 May 2019
Available online 7 June 2019

Keywords:
Uniqueness
Stability
Evolutionary dynamics
Asymmetric interactions
Positive interactions
Social interactions

ABSTRACT

It is well known that uniqueness and stability are guaranteed properties of traffic equilibria in static user-equilibrium traffic assignment problems, if the link travel utilities are assumed to be strictly monotonically decreasing with respect to the link traffic volumes. However, these preferable properties may not necessarily hold in a wide range of transport problems with complex interactions, e.g. asymmetric interactions (including dynamic traffic assignment), social interactions, or with economies of scale. This study aims to investigate such solution properties of transport models with complex interactions between users. Generic formulations of models are considered in this study, both for utility functions and for the evolutionary dynamics relevant to the stability analysis. Such an analysis for a generic formulation is mathematically challenging due to the potential non-differentiability of the dynamical system, precluding the application of standard analyses for smooth systems. To address this issue, this study proposes a transport system with two alternatives and two user groups. While it is a simple model whose dynamics can be depicted on a plane, it also includes the core components of transport models, i.e. multiple choices and user-classes. This study classifies all possible formulations into nine cases with respect to the signs (i.e. positive or negative) of interactions between users. Then, the evolutionary dynamics of each case is mathematically analysed to examine stability of equilibria. Finally, the solution properties of each case is revealed. Multiple equilibria exist in many cases. In addition, cases with no stable equilibrium are also found, yet even in such cases we are able to characterise the circumstances in which the different kinds of unstable behaviour may arise.

© 2019 The Authors. Published by Elsevier Ltd.
This is an open access article under the CC BY license.
(http://creativecommons.org/licenses/by/4.0/)

It is well known that uniqueness and stability are guaranteed properties of traffic equilibria in static user-equilibrium traffic assignment problems, if the link travel utilities are assumed to be strictly monotonically decreasing with respect to the link traffic volumes. However, these preferable properties may not necessarily hold in a wide range of transport problems with complex interactions, e.g. asymmetric interactions (including dynamic traffic assignment), social interactions, or with economies of scale.

複雑な相互作用の動学

- … the existence of positive interactions. In the traditional context of transport studies, interactions between users are considered to be mostly originated by congestion phenomena, which creates negative interactions (snip). In contrast, positive interaction is characterised by a situation in which the increased patronage of a transport facility increases its attractiveness for other users.

複雑な相互作用の動学

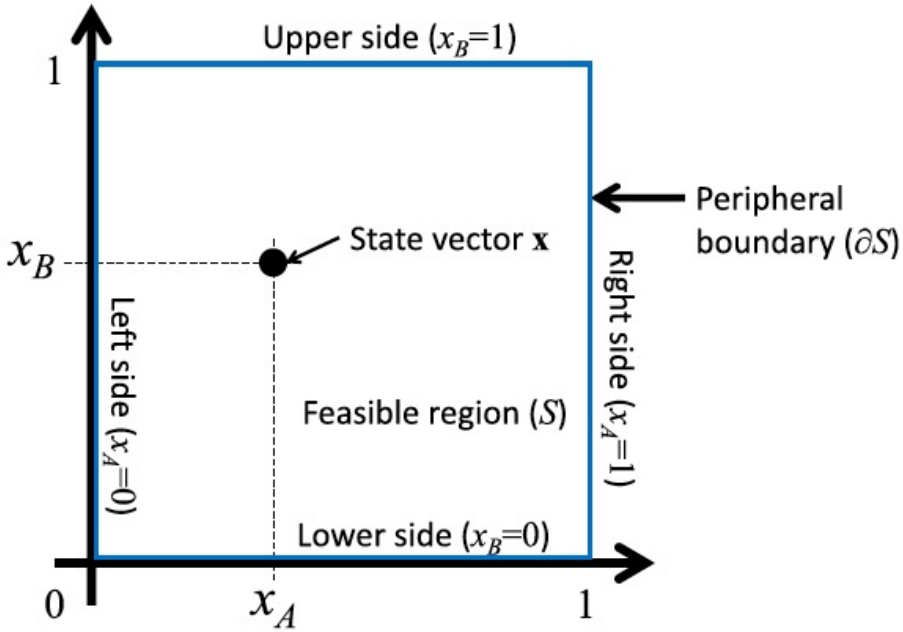
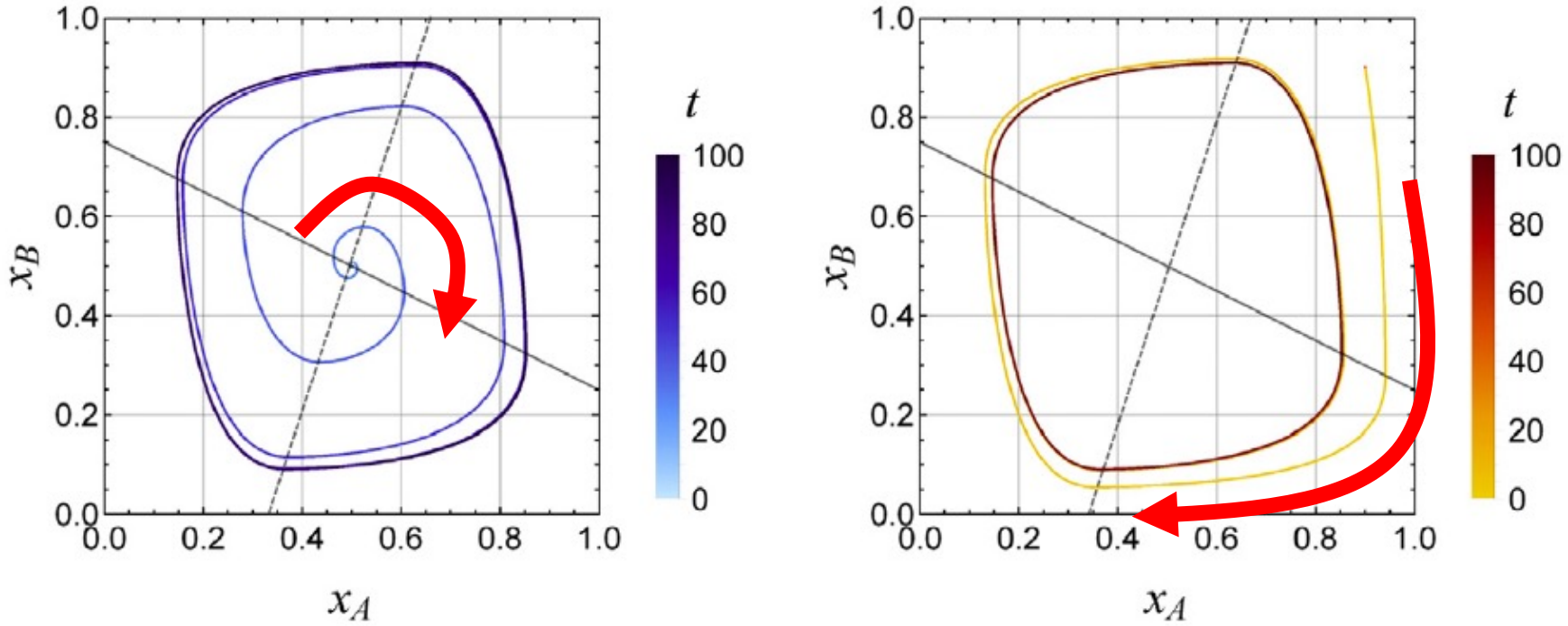


Fig. 1. Feasible region and state vector on phase diagram.

two alternatives and two user groups
two-dimensional problem

複雑な相互作用の動学



複雑な相互作用の動学

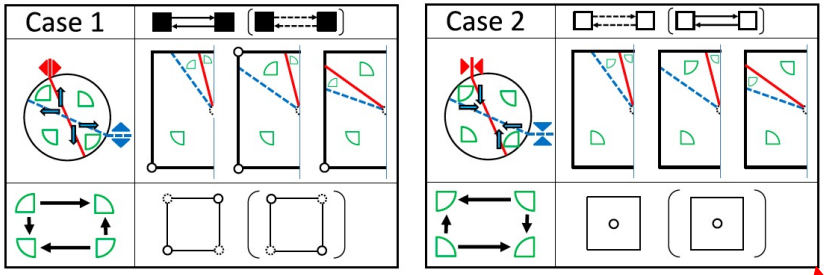


Fig. 19. Summary charts of Cases 1 and 2.

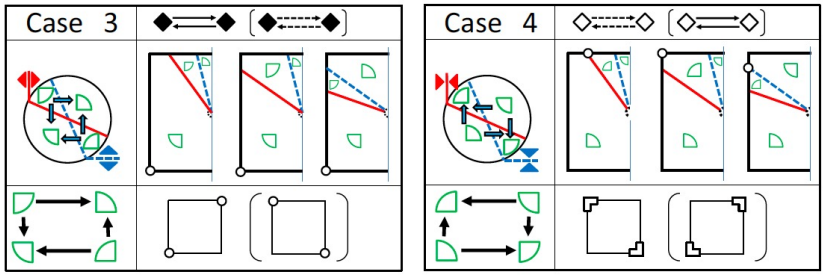


Fig. 20. Summary charts of Cases 3 and 4.

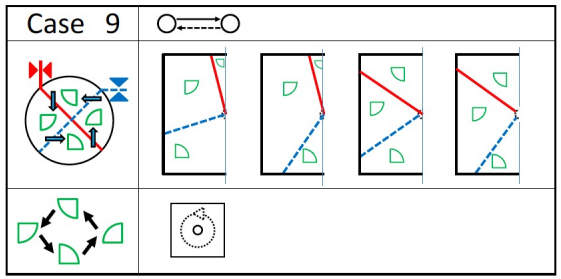


Fig. 24. Summary chart of Case 9.

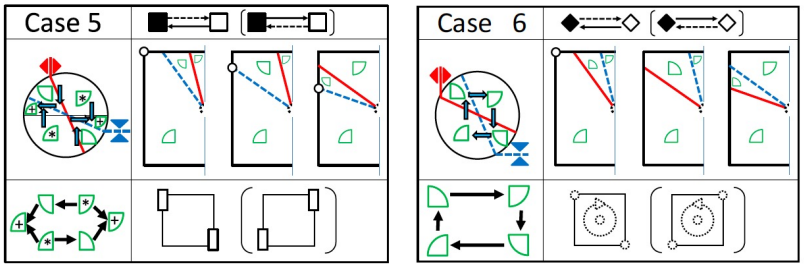


Fig. 21. Summary charts of Cases 5 and 6.

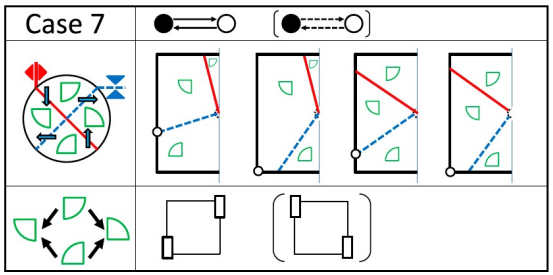


Fig. 22. Summary chart of Case 7.

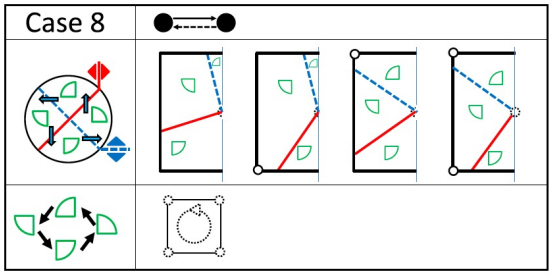
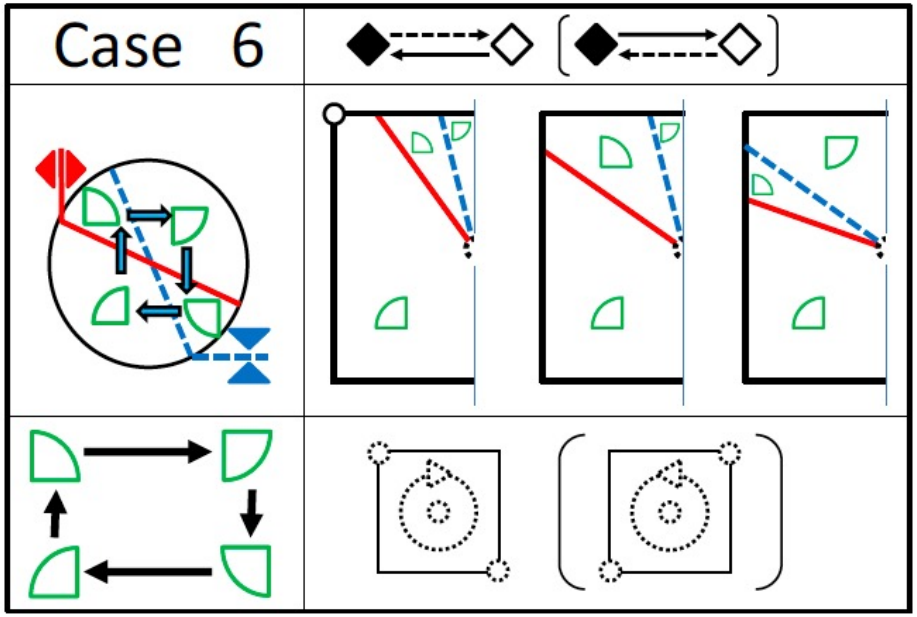


Fig. 23. Summary chart of Case 8.

可能性のある全ケースを
9種類に場合わけして性質を特定

一意な安定均衡解がある
ケースは1個しかない！

複雑な相互作用の動学



There exist two groups of travellers, called leaders and followers in a city. Followers always observe leaders behaviour because they want to imitate it and leaders also have slight social interactions between them.

Leader-follower model
端点解か，内点解か，不安定解かは
Day-to-dayの動学の特性に依存する.

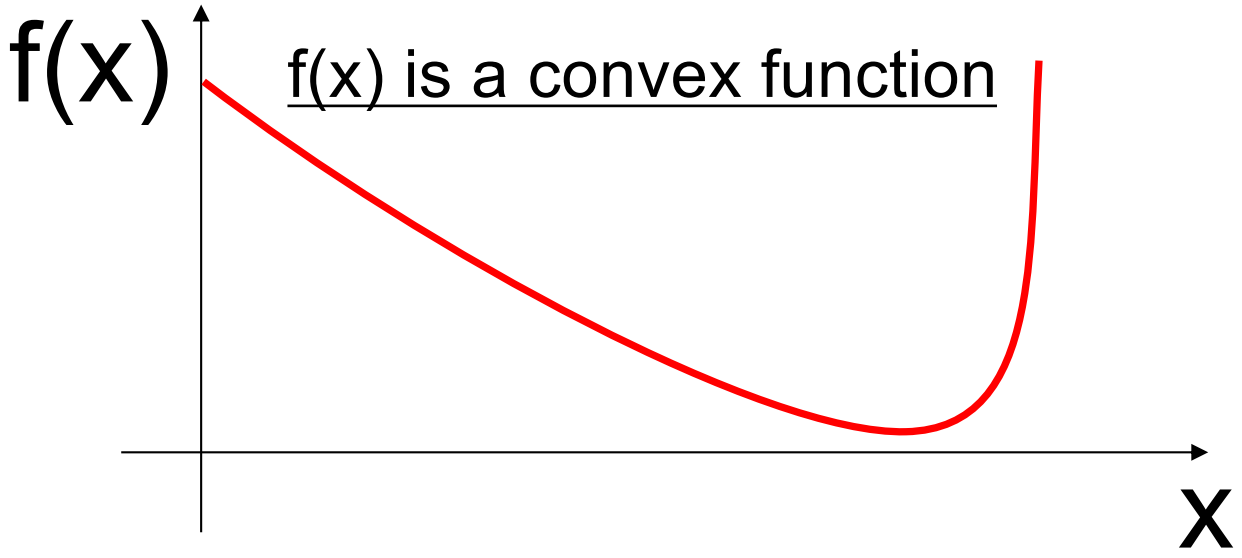
まとめ

- 交通システムの将来予測や施策立案のためには，システム内での利用者の相互作用の影響を考えなくてはならない．
- 均衡状態：相互作用の結果，最終的に安定して実現する（しそう）な状態．
  - それを条件とした交通量配分：利用者均衡配分
- 等価最適化問題による解法：Frank-Wolfe法
- 均衡解の一意性と安定性の問題
  - 複雑な相互作用を考えると，これらが必ずしも成立しなくなる．
  - Day-to-dayの選択行動調整過程をどうモデリングするか？

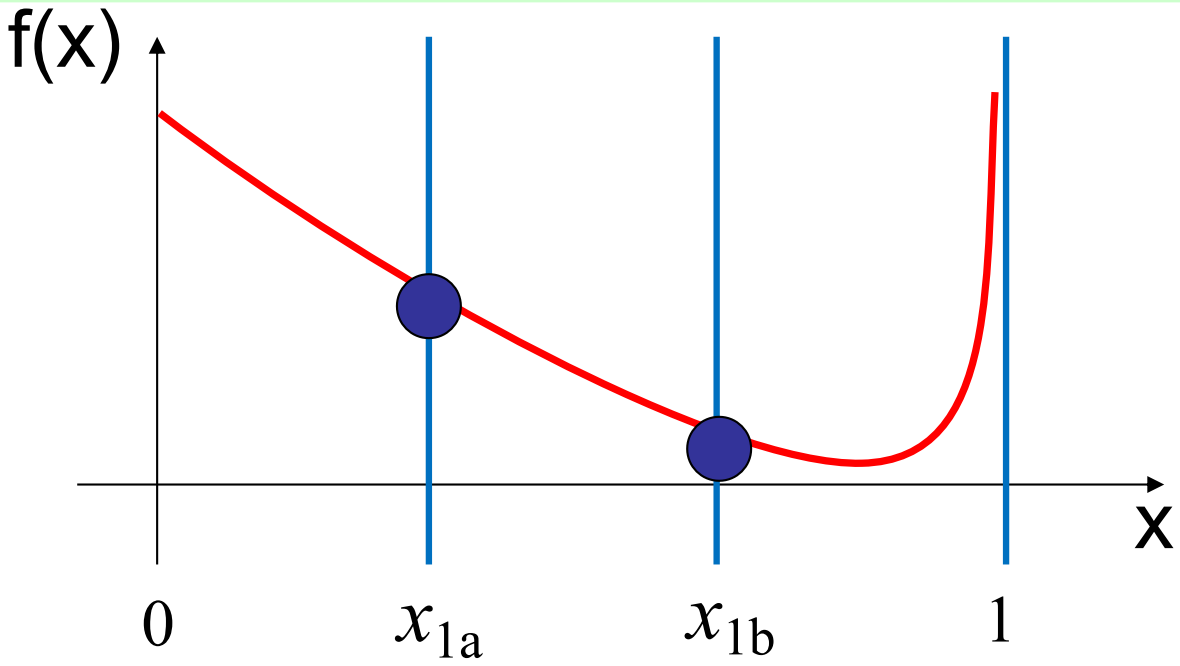
付録:1次元探索

How to solve 1D optimisation problem

- The golden-section search is a well-known method for a convex (or concave) function.
- Consider: $\min f(x)$ sub. to: $0 \leq x \leq 1$.

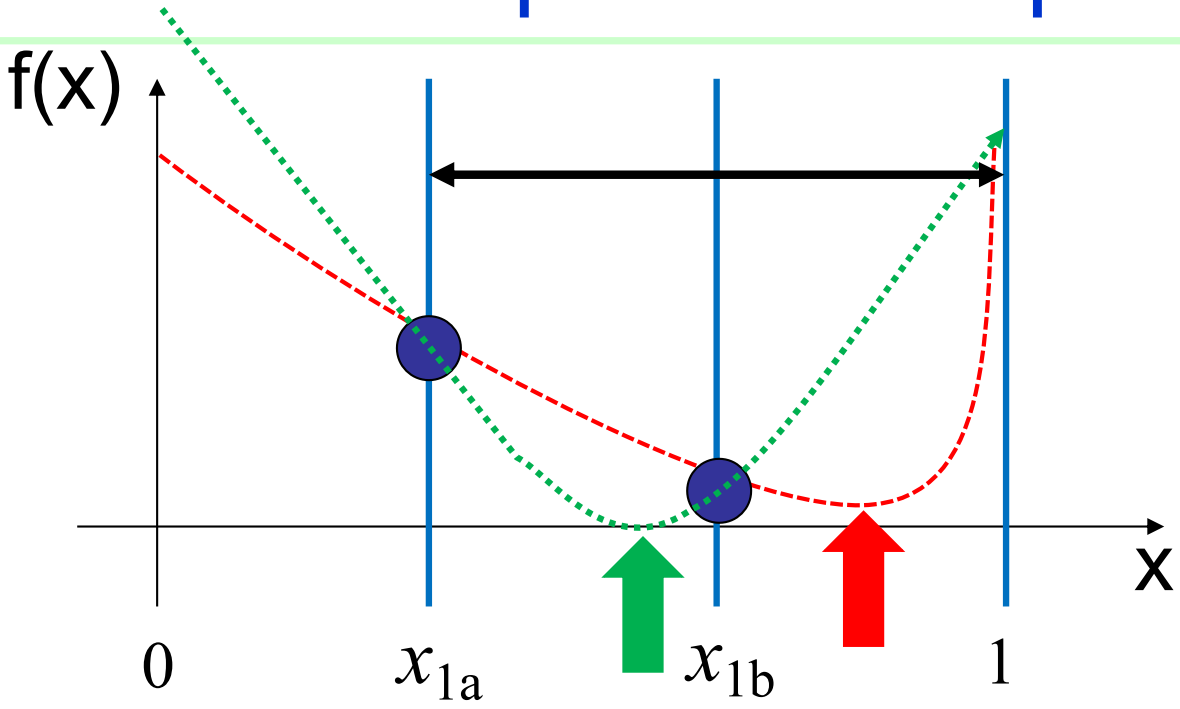


How to solve 1D optimisation problem



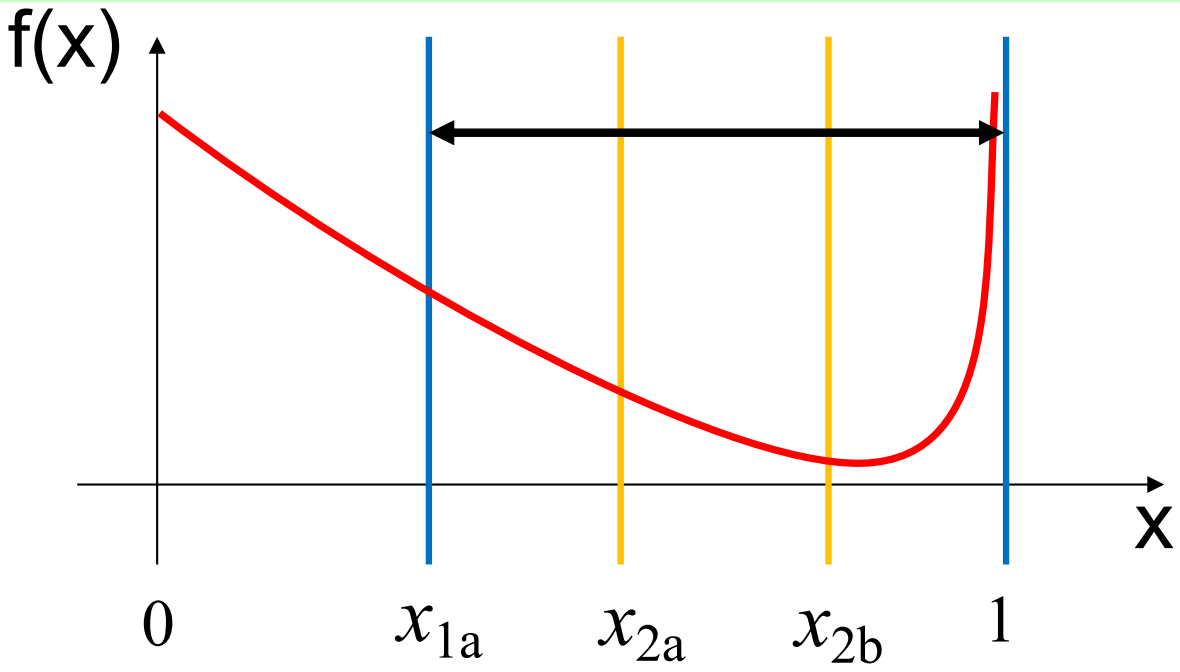
Set two intermediate points, denoted by x_{1a} and x_{1b} and calculate $f(x_{1a})$ and $f(x_{1b})$.

How to solve 1D optimisation problem



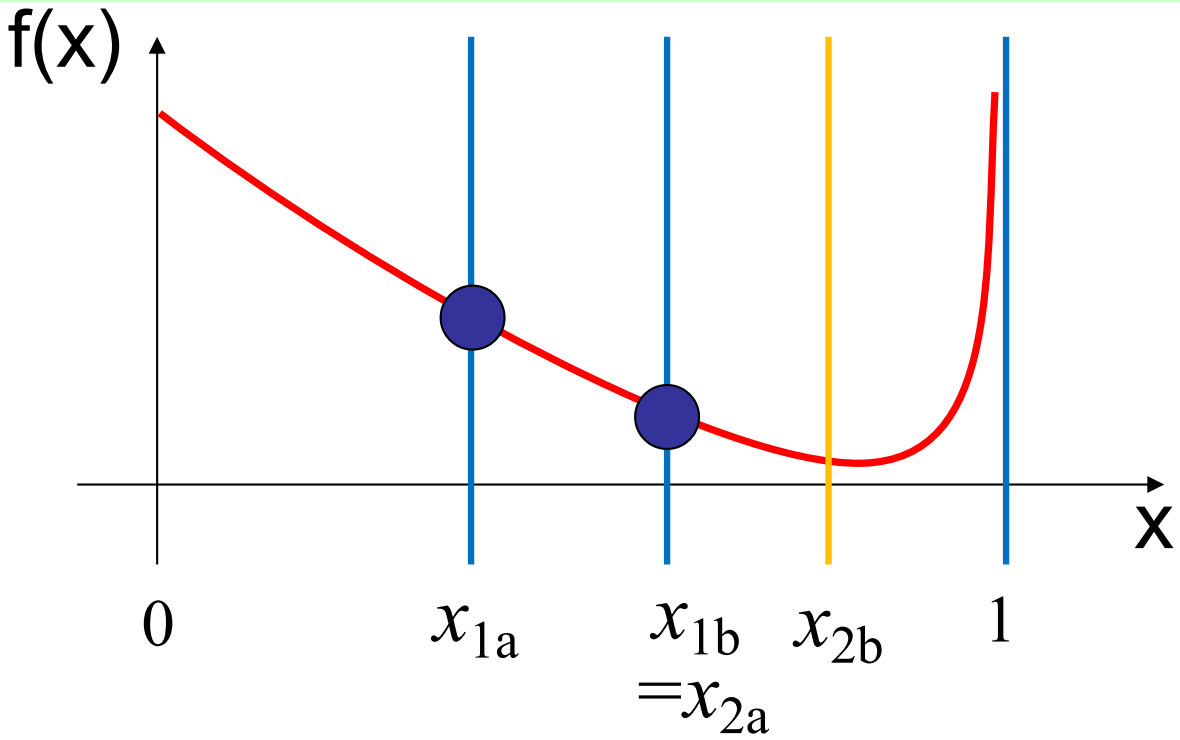
If $f(x_{1a}) > f(x_{1b})$, the optimal point is between x_{1a} and 1 at all times because the function is convex.

How to solve 1D optimisation problem



Do the same thing between x_1 and 1 instead of 0 and 1. Repeating this process, the section where the optimal point must exist is getting shorter.

Golden section search

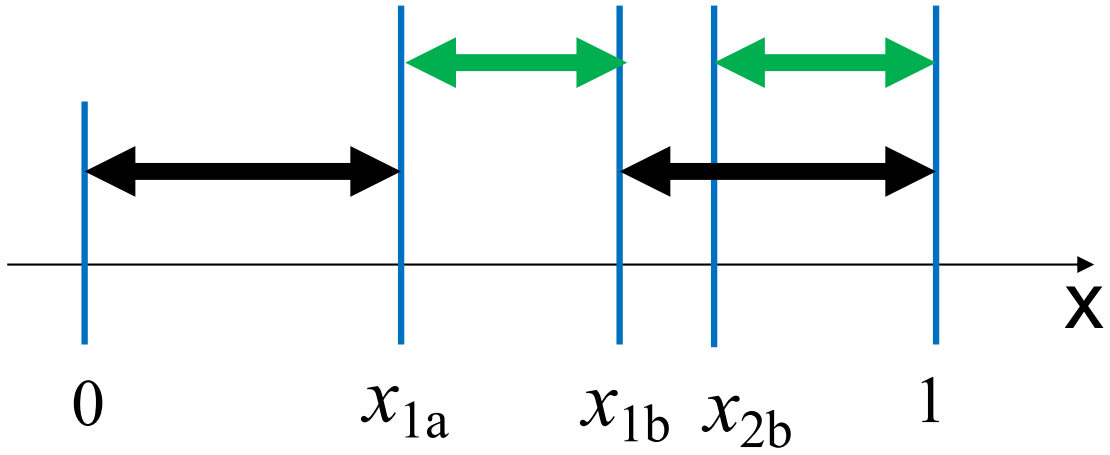


Now find x_{1a} and x_{1b} so that $x_{1b} = x_{2a}$. If we can do so, we can reduce the number of calculations of f(x).

Golden section search

- $\frac{x_{1b}-x_{1a}}{1-x_{1a}} = x_{1a}$
- $x_{1b} = 1 - x_{1a}$

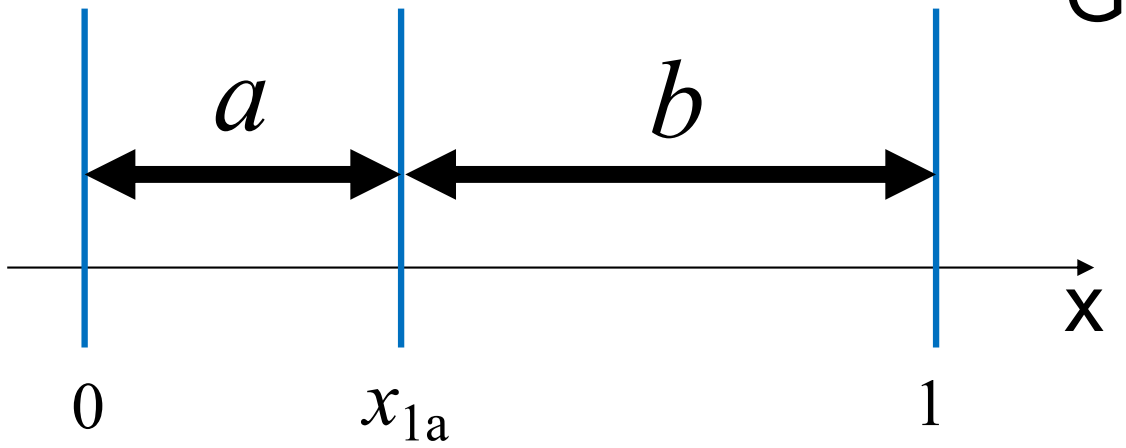
$$x_{1a} = \frac{3-\sqrt{5}}{2}$$



Golden section search

$$a:b = \frac{3-\sqrt{5}}{2} : 1 - \frac{3-\sqrt{5}}{2} = 1 : \frac{1+\sqrt{5}}{2}$$

Golden ratio



Golden section search: Algorithm


```
1   import math
2
3   def func(x):
4       return (x-0.7)**2
5
6   def goldenSearch(func,xa,xb,tolerance,isExact,isPrint):
7       # 分割比（区間全体を1とした値）Golden ratio (entire section is set to 1)
8       gr = (3-math.sqrt(5))/2
9       # 区間の幅 With of the entire section
10      wd = xb - xa
11      # 左側分割位置の初期値 Initial valuel of the left-hand side dividing point
12      xa_int = xa + wd*gr
13      fa = func(xa_int)
14      # 右側分割位置の初期値 Initial valuel of the right-hand side dividing point
15      xb_int = xb - wd*gr
16      fb = func(xb_int)
```

```
18      while True:
19
20          # 計算結果の出力 Printing the calculation process
21          if isPrint:
22              print ("(%6.4f,%6.4f,%6.4f,%6.4f) (%8.6f,%8.6f)" % (xa,xa_int,xb_int,xb,fa,fb))
23
24          # 2つの分割間で右下がりであれば，左側分割より左側を切り落とす
25          if (fa > fb):
26              xa = xa_int
27              wd = xb - xa
28              # 新しい左側分割の計算 Caculating new left-hand side dividing point
29              # 黄金分割の特性を用いずに，あえて計算しなおしている（チェック用）
30              # The following calculation does not utilse the advantahe of the golden section method,
31              # for checking purpose only.
32              if (isExact):
33                  xa_int = xa + wd*gr
34                  fa = func(xa_int)
35              # 黄金分割による計算 Calculation based on the golden section method.
36              else:
37                  xa_int = xb_int
38                  fa = fb
39               # 新しい右側分割の計算 Caculating new right-hand side dividing point
40              xb_int = xb - wd*gr
41              fb = func(xb_int)
```

```
42        else:
43            xb = xb_int
44            wd = xb - xa
45            if (isExact):
46                xb_int = xb - wd*gr
47                fb = func(xb_int)
48            else:
49                xb_int = xa_int
50                fb = fa
51            xa_int = xa + wd*gr
52            fa = func(xa_int)
53
54        # 収束判定 Convergence check
55        if (xb_int - xa_int) < tolerance:
56            break
57
58    return [0.5*(xb_int + xa_int),0.5*(fa+fb)]
```

```
print ("Answer:", goldenSearch(func,0,1,0.01,True,True))
print()
print ("Answer:", goldenSearch(func,0,1,0.01,False,True))
```

```
(0.0000,0.3820,0.6180,1.0000) (0.101146,0.006718)
(0.3820,0.6180,0.7639,1.0000) (0.006718,0.004087)
(0.6180,0.7639,0.8541,1.0000) (0.004087,0.023747)
(0.6180,0.7082,0.7639,0.8541) (0.000067,0.004087)
(0.6180,0.6738,0.7082,0.7639) (0.000688,0.000067)
(0.6738,0.7082,0.7295,0.7639) (0.000067,0.000870)
(0.6738,0.6950,0.7082,0.7295) (0.000025,0.000067)
Answer: [0.690983005625052 5, 9.783292789580766e-05]

(0.0000,0.3820,0.6180,1.0000) (0.101146,0.006718)
(0.3820,0.6180,0.7639,1.0000) (0.006718,0.004087)
(0.6180,0.7639,0.8541,1.0000) (0.004087,0.023747)
(0.6180,0.7082,0.7639,0.8541) (0.000067,0.004087)
(0.6180,0.6738,0.7082,0.7639) (0.000688,0.000067)
(0.6738,0.7082,0.7295,0.7639) (0.000067,0.000870)
(0.6738,0.6950,0.7082,0.7295) (0.000025,0.000067)
Answer: [0.690983005625052 5, 9.783292789580766e-05]
```

```
# FW（1ODペア専用，最短経路は経路列挙による簡便な手法）
# Frank-Wolfe algorithm, with shortest path search enumerating all routes.


import numpy as np
import pandas as pd
from scipy.optimize import minimize
import matplotlib.pyplot as plt

# 経路交通量からOD交通量を計算する Calculating OD traffic volume from route traffic volume
def calcODTraf(route_traf):
    od_traf = 0
    for route in routes:
        od_traf += route_traf[route]
    return od_traf

# 経路交通量からリンク交通量を計算する Calculating link traffic volume from route traffic volume
def calcLinkTraf(route_traf):
    link_traf = {}
    for link in links:
        link_traf[link] = 0
    for route in routes:
        for link in routes[route]:
            link_traf[link] += route_traf[route]
    return link_traf

# リンク交通量から利用者均衡配分用の目的関数を計算する
# Calculating the objective function from link traffic volume
def calcObjUE(link_traf):
    obj = 0
    for link in links:
        obj += 0.5*links[link][1]*link_traf[link]**2 ¥
            +links[link][0]*link_traf[link]
    return obj

# リンク交通量から各経路の旅行時間を計算する Calculating route travel time from link traffic volume
def calcRouteTT(link_traf):
    route_TT={}
    for route in routes:
        route_TT[route] = 0
    for route in routes:
        for link in routes[route]:
            route_TT[route] = route_TT[route] ¥
                + links[link][1]*link_traf[link]+links[link][0]
    return route_TT

# ここからメインルーチン Main routine

# すべてのリンクとそのリンク旅行時間パラメータ([自由流旅行時間，交通量との比例係数])
# All links and their link travel time parameters ([free-flow travel time, coeff of traffic volume])
links = {'A':[10,10],'B':[60,1],'C':[60,1],'D':[10,10],'E':[10,1]}

# すべての経路とそれに属するリンク
# All routes and links included by each route.
routes={'a':['A','B'], 'b':['C','D'],  'c':['A','E','D']}

# 初期リンク交通量パターン．とりあえず経路aにOD交通量6を全部置く
# 本来のFWであれば，自由流旅行時間に基づいて最短経路探索して経路を決めるべき
# Initial link traffic volume pattern. Assign all OD traffic to route 'a' here.
# It must be replaced by the shortest path search with free-flow link travel times given.
OD_traf = 6
link_traf_x = calcLinkTraf({'a':OD_traf,'b':0,'c':0})

print ("Initial link traffic volume: ",link_traf_x)
print ()

# FWをとりあえず5回繰り返す FW is going to repeat for five times here.
for k in range(1,5):
    # 経路旅行時間を計算（本来のFWであれば，リンク旅行時間を計算してから最短経路探索をする）
    # Calculate route travel time, note that it must be replaced by the shortest path search
    # with link travel times calculated.
    route_TT = calcRouteTT(link_traf_x)
    minTT = 10000000
    minRoute = ''
    for route in route_TT:
        if minTT > route_TT[route]:
            minTT = route_TT[route]
            minRoute = route
    print ("Shortest route: ",minRoute,minTT)

    # All-or-nothing 配分
    link_traf_y = calcLinkTraf(
        {'a':OD_traf*('a'==minRoute),'b':OD_traf*('b'==minRoute),'c':OD_traf*('c'==minRoute)}
        )
    print ("All-or-nothing assignment: ",link_traf_y)

    # 一次元探索（単純な方法による）
    # One-dimension search (Naive method)
    minObj = 100000000
    minAl = 0
    for alpha in range(0,30001):
        al = alpha / 30000
        link_mix = {}
        for link in links:
            link_mix[link] = al * link_traf_y[link] + (1-al) * link_traf_x[link]
        obj = calcObjUE(link_mix)
        if (minObj > obj):
            minObj = obj
            minAl = al

    print ("1D search alpha = ",minAl," Obj = ",minObj)

    # 一次元探索の最適解を用いて，リンク交通量を更新
    # Update link traffic volume pattern using the optimal solution of 1-D search.
    for link in links:
        link_traf_x[link] = minAl * link_traf_y[link] + (1-minAl) * link_traf_x[link]

    print ("Current link traffic volume: ",link_traf_x)
    print ()
```

```
import math

def func(x):
    return (x-0.7)**2

def goldenSearch(func,xa,xb,tolerance,isExact,isPrint):
    # 分割比（区間全体を1とした値）Golden ratio (entire section is set to 1)
    gr = (3-math.sqrt(5))/2
    # 区間の幅 With of the entire section
    wd = xb - xa
    # 左側分割位置の初期値 Initial valuel of the left-hand side dividing point
    xa_int = xa + wd*gr
    fa = func(xa_int)
    # 右側分割位置の初期値 Initial valuel of the right-hand side dividing point
    xb_int = xb - wd*gr
    fb = func(xb_int)

    while True:

        # 計算結果の出力 Printing the calculation process
        if isPrint:
            print ("(%6.4f,%6.4f,%6.4f,%6.4f) (%8.6f,%8.6f)" % (xa,xa_int,xb_int,xb,fa,fb))

        # 2つの分割間で右下がりであれば，左側分割より左側を切り落とす
        if (fa > fb):
            xa = xa_int
            wd = xb - xa
            # 新しい左側分割の計算 Caculating new left-hand side dividing point
            # 黄金分割の特性を用いずに，あえて計算しなおしている（チェック用）
            # The following calculation does not utilse the advantahe of the golden section method,
            # for checking purpose only.
            if (isExact):
                xa_int = xa + wd*gr
                fa = func(xa_int)
            # 黄金分割による計算 Calculation based on the golden section method.
            else:
                xa_int = xb_int
                fa = fb
            # 新しい右側分割の計算 Caculating new right-hand side dividing point
            xb_int = xb - wd*gr
            fb = func(xb_int)
        else:
            xb = xb_int
            wd = xb - xa
            if (isExact):
                xb_int = xb - wd*gr
                fb = func(xb_int)
            else:
                xb_int = xa_int
                fb = fa
            xa_int = xa + wd*gr
            fa = func(xa_int)

        # 収束判定 Convergence check
        if (xb_int - xa_int) < tolerance:
            break

    return [0.5*(xb_int + xa_int),0.5*(fa+fb)]



print ("Answer:", goldenSearch(func,0,1,0.01,True,True))
print()
print ("Answer:", goldenSearch(func,0,1,0.01,False,True))
```