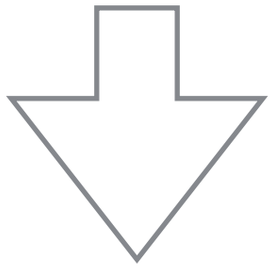
科学技術計算

科学技術計算に必要なこと

- 数値計算 (行列演算、モンテカルロ法)
- 設計 (オブジェクト指向)
- バージョン管理システム
- 並列化

論理、データ構造を定義・実装



コンピュータ上で自動処理

データ処理に使うメリット

- データを更新したときに再処理を自動化
- 人と共有

実行方式

- ☑ 実行が高速 (コンパイル時に最適化)
- ☐ 環境毎に、コードの変更毎に再コンパイル

例 C, C++, Fortran

- ☑ コンパイル不要
- ☐ 実行が比較的低速

例 Python, Javascript

プログラミング言語

- Fortran　科学技術計算に特化
- C/C++　汎用。OSの実装から科学技術計算まで。
- Python　豊富なライブラリ (信号処理、行列計算、文章処理など)

グルー言語 (glue language)　異なるプログラムを結合

Julia, Java, *etc*.

Python

- マルチプラットフォーム (Windows, Mac, Unix, *etc*.)
- 整理された(比較的)簡単な構文　入門言語
- 行列演算、フーリエ変換、疑似乱数、グラフ出力
- 並列コンピューティング

数値

精度は使うシステムに依存

浮動小数点数　科学技術計算で普通使うのは倍精度 (64 bits=2^{64}状態)

Pythonではfloat, C/C++ではdoubleと呼ばれる

$1.234567891011 \times 10^{300}$

IEEE754: 符号 (1bit)、指数部 (11bits), 仮数部 (52bits)

$(-1)^{sign} (1.b_1b_2\ldots b_{52}) \times 2^{e-1023}$

十進数換算で有効桁数は十数桁

値域は大体10^{-308}〜10^{+308}

符号付き整数　32bitの場合、-2^{31} 〜 $2^{31}-1$

$2^{31}-1=2,147,483,647 \sim 2\times10^9$

落とし穴

整数の割り算　　3/4 = 0

5/4 = 1

cf. 3.0/4 = 0.75

情報落ち　　1.0 + 1E-19 = 1.0

桁落ち　　大きな数 - 大きな数 = 小さな数

> 1.00000000003 - 1.00000000002

> 1.000000082740371e-11

型システム

- 動的型付け = 実行時に変数が格納する型が決定

i = 0	整数
i = 0.0	倍精度浮動小数点
i = “a”	文字列

コードが柔軟 (場合によって型を変更)、実行時まで型が確定しない

- 静的型付け (C++) = コンパイル時に型が決定

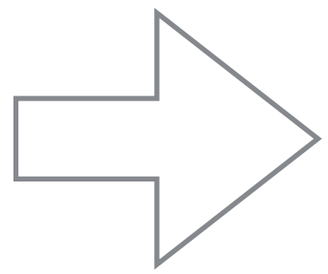
```
int i = 0; //OK
std::string i = 0; //NG
```

コンパイラによる厳密な型チェック

制御構造

条件分岐 (if文)

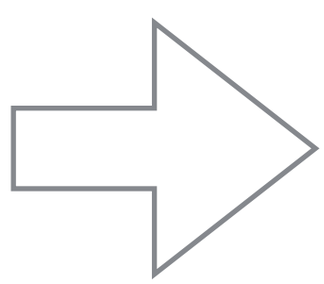
```
i = 2
if i > 2:
    print "A"
else:
    print "B"
```



```
B
```

反復(for文)

```
for i in xrange(4):
    print i
```



```
0
1
2
3
```

関数

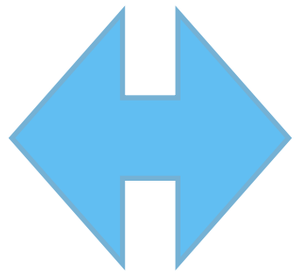
```
if i > 0:
    if j < 0:
        jの値による複雑な処理 (100行)
    else:
        jの値による複雑な処理 (100行)
else:
    複雑な処理
```

```
if i > 0:
    if j < 0:
        function_A(j)
    else:
        function_A(j)
else:
    function_B()
```

クラス

ユーザ定義型データ構造＋データに対する操作を担う関数

メインルーチン



クラス

グラフ出力を担う

外部からは見えないデータ構造

プログラムのコンポーネント化、データのカプセル化、継承

これからの予定

・Python言語の基本的な文法
・オブジェクト指向
・数値計算ライブラリ
・サンプルコード作成 (数値計算、グラフプロット)

Python

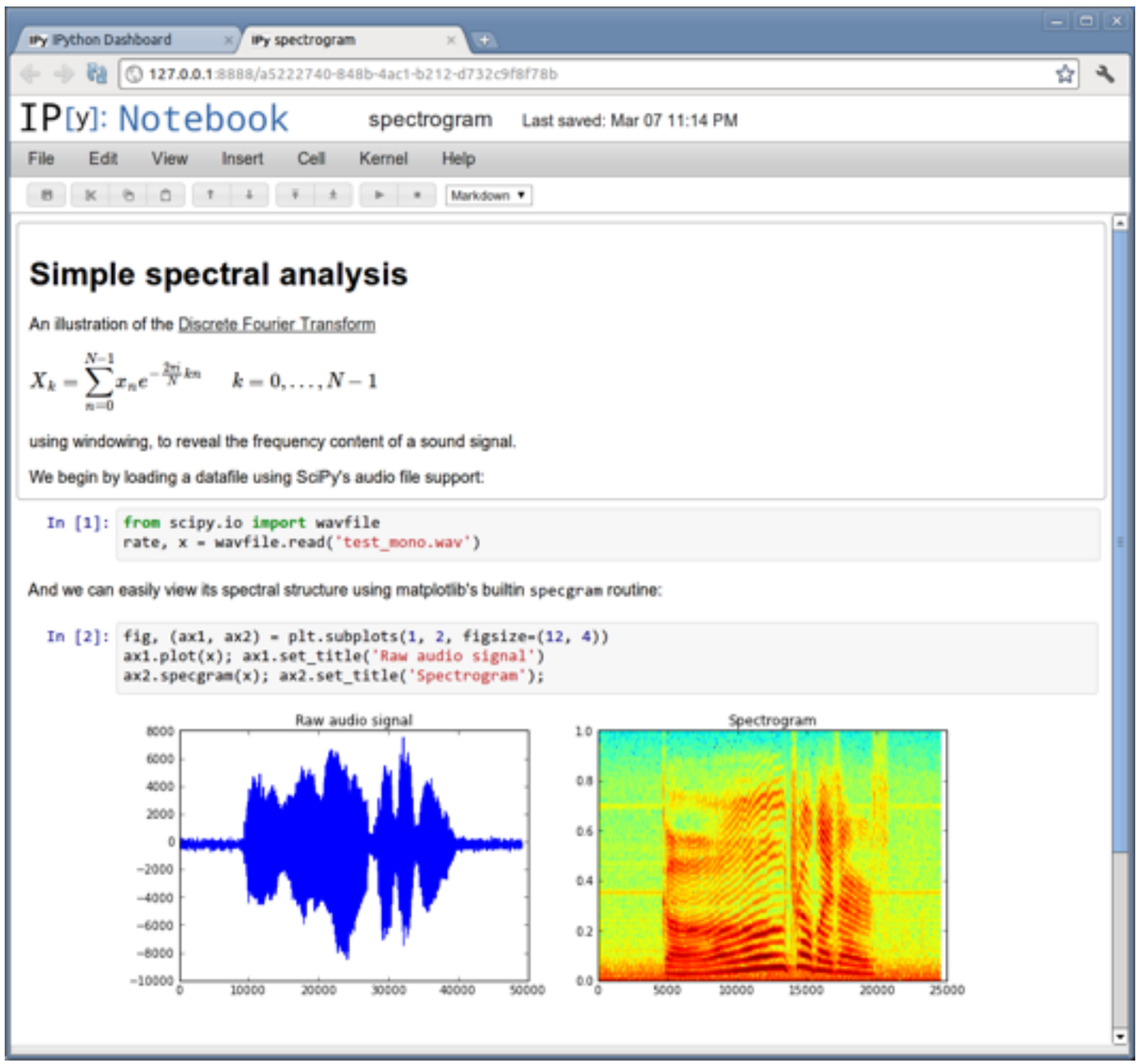
- マルチプラットフォーム (Windows, Mac, Unix, *etc*.)
- 整理された(比較的)簡単な構文　入門言語
- 行列演算、フーリエ変換、疑似乱数、グラフ出力
- 並列コンピューティング

2つの系列: Python2.7, Python3.5

http://docs.python.jp/3/tutorial/

Pythonを起動する

- 対話的に利用 (標準インタープリター)
- IPython/Jupyter Notebook
- プログラムとして利用 (例 python program.py)



http://pgbovine.net/ipython-notebook-first-impressions.htm

ソースコードの記述

- 文字コード　UFT-8が使える。実際にはASCIIにとどめるのが良い。
- 基本的にはテキストエディタで編集

  emacs, vim, PyCharm (統合開発環境)

型 (クラス)

データ+データに対する手続き

- 組み込み型: 浮動小数点数(float)、整数(int)、真偽値(bool)、文字型 (string), リスト型 (list)、複素数 (complex)

float: 1.0, 2.0, …
int: 1, 2, …
bool: True, False

```
>>> print(type(1.0))
>>> <class 'float'>
```

- ユーザ定義型

ndarray in numpy, etc.

モジュール

関数、クラスをまとめた物

```
>>> import numpy as np
>>> np.exp(1.0)
>>> 2.7182818284590451
```

npという別名でnumpyをインポート
numpyの中の関数expを実行

標準: random

非標準: numpy, scipy

numpy (numerical python), scipy

著者: *Adrien Chauve, Andre Espaze, Emmanuelle Gouillart, Gaël Varoquaux, Ralf Gommers*

Scipy

`scipy` パッケージは科学技術計算での共通の問題のための多様なツールボックスがあります。サブモジュール毎に応用範囲が異なっています。応用範囲は例えば、補完、積分、最適化、画像処理、統計、特殊関数等。

`scipy` は GSL (GNU Scientific Library for C and C++) や Matlab のツールボックスのような他の標準的な科学技術計算ライブラリと比較されます。 `scipy` は Python での科学技術計算ルーチンの中核となるパッケージです; これは `numpy` の配列を効率良く扱っているということで、numpy と scipy は密接に協力して動作しています。

ルーチンを実装する前に、望んでいるデータ処理が Scipy で既に実装されているかを確認した方がいいでしょう。プロフェッショナルでないプログラマや科学者は **車輪の再開発** をしたがります、これはバグが多く最適でなく、共有が難しく、メンテナンスできないコードに陥りがちです。対照的に `Scipy` のルーチンは最適化され、テストされていてるので、利用できるときには利用すべきです。

http://www.turbare.net/transl/scipy-lecture-notes/intro/scipy.html

行列演算: blasやlapackを呼び出す実装→高性能

numpy.ndarray

N次元配列クラス

```
# 1x2配列 (0初期化)
>>> A = np.zeros((1,2), dtype=float)
>>> print(A)
>>> [[ 0.  0.]]

# 長さ4の一次元配列 (0初期化、複素数)
>>> A = np.zeros((4,), dtype=complex)
>>> print(A)
>>> [ 0.+0.j  0.+0.j  0.+0.j  0.+0.j]
```

例: 4次元配列の要素へのアクセス A[i,j,k,l]

例: strides, A[:,j,k,l] →部分配列への参照の取り出し

```
A[:,j,k,l] = 1.0
```

Row-major(C) or Column-major(Fortran)

Row-major: A[0,0], A[0,1], A[0,2], A[1,0], A[1,2], …

Column-major: A[0,0], A[1,0], A[2,0], A[0,1], A[2,1], …

現代のCPUはメモリアクセスが低速。メモリを連続的にアクセスする場合には、キャッシュから恩恵

参照の代入

```
>>> A = np.zeros((4,4,4), dtype=complex)
>>> B = np.zeros((4,4,4), dtype=complex)
>>> A = B (Bが指す配列への参照がAへ代入)
>>> A[:,:,:] = 1.0
>>> print(B)
>>> [[[ 1.+0.j  1.+0.j  1.+0.j  1.+0.j]
  [ 1.+0.j  1.+0.j  1.+0.j  1.+0.j]
  [ 1.+0.j  1.+0.j  1.+0.j  1.+0.j]
  [ 1.+0.j  1.+0.j  1.+0.j  1.+0.j]]

 [[ 1.+0.j  1.+0.j  1.+0.j  1.+0.j]
  [ 1.+0.j  1.+0.j  1.+0.j  1.+0.j]
  [ 1.+0.j  1.+0.j  1.+0.j  1.+0.j]
  [ 1.+0.j  1.+0.j  1.+0.j  1.+0.j]]

[[ 1.+0.j  1.+0.j  1.+0.j  1.+0.j]
 [ 1.+0.j  1.+0.j  1.+0.j  1.+0.j]
 [ 1.+0.j  1.+0.j  1.+0.j  1.+0.j]
 [ 1.+0.j  1.+0.j  1.+0.j  1.+0.j]]

[[ 1.+0.j  1.+0.j  1.+0.j  1.+0.j]
 [ 1.+0.j  1.+0.j  1.+0.j  1.+0.j]
 [ 1.+0.j  1.+0.j  1.+0.j  1.+0.j]
 [ 1.+0.j  1.+0.j  1.+0.j  1.+0.j]]]
```

実は、AとBが同じ配列を参照している

配列のコピー

```
>>> A = np.zeros((4,4,4), dtype=complex)
>>> B = np.zeros((4,4,4), dtype=complex)
>>> A = np.array(B)     (Bが指す配列のコピーを作成して、Aへその参照を代入)
>>> A[:,:,:] = 1.0
>>> print(B)
>>> [[[ 0.+0.j  0.+0.j  0.+0.j  0.+0.j]
  [ 0.+0.j  0.+0.j  0.+0.j  0.+0.j]
  [ 0.+0.j  0.+0.j  0.+0.j  0.+0.j]
  [ 0.+0.j  0.+0.j  0.+0.j  0.+0.j]]

 [[ 0.+0.j  0.+0.j  0.+0.j  0.+0.j]
  [ 0.+0.j  0.+0.j  0.+0.j  0.+0.j]
  [ 0.+0.j  0.+0.j  0.+0.j  0.+0.j]
  [ 0.+0.j  0.+0.j  0.+0.j  0.+0.j]]

 [[ 0.+0.j  0.+0.j  0.+0.j  0.+0.j]
  [ 0.+0.j  0.+0.j  0.+0.j  0.+0.j]
  [ 0.+0.j  0.+0.j  0.+0.j  0.+0.j]
  [ 0.+0.j  0.+0.j  0.+0.j  0.+0.j]]

 [[ 0.+0.j  0.+0.j  0.+0.j  0.+0.j]
  [ 0.+0.j  0.+0.j  0.+0.j  0.+0.j]
  [ 0.+0.j  0.+0.j  0.+0.j  0.+0.j]
  [ 0.+0.j  0.+0.j  0.+0.j  0.+0.j]]]
```

配列とリストの違い

list: 任意の型のオブジェクトの並び。汎用的だが遅い

```
>>> A = [0.0, 0.0, 0.0]
>>> B = np.array(A)
>>> print(B)
>>> [ 0.  0.  0.]
```

配列のリストを多次元配列へ変換

```
>>> A1 = np.zeros((4,4))
>>> A2 = np.zeros((4,4))
>>> A = np.array([A1, A2])
>>> print(A.shape)
>>> (2, 4, 4)
```

配列演算

```
>>> A = np.linspace(1, 10, 10)
>>> print(A)
>>> [  1.   2.   3.   4.   5.   6.   7.   8.   9.  10.]

>>> A *= 2
>>> print(A)
>>> [  2.   4.   6.   8.  10.  12.  14.  16.  18.  20.]

>>> A = A.reshape((2,5))
>>> print(A)
>>> [[  2.   4.   6.   8.  10.]
 [ 12.  14.  16.  18.  20.]]

>>> print(A.shape)
>>> (2, 5)

#NG
>>> B = np.zeros_like(A)
>>> B = 1.0   #1.0というfloatのオブジェクトを作成して、Bに参照を代入
>>> print(B)
>>> 1.0

#OK
>>> B = np.zeros_like(A)
>>> B[:,:] = 1.0
>>> print(B)
>>> [[ 1.  1.  1.  1.  1.]
 [ 1.  1.  1.  1.  1.]]

>>> A = A+B
>>> print(A)
>>> [[  3.   5.   7.   9.  11.]
 [ 13.  15.  17.  19.  21.]]
```

統計処理

```
>>> import numpy as np
>>> np.average([0.0, 1.0, 2.0])   #平均
>>> 1.0
>>> np.std([0.0, 1.0, 2.0])       #標準偏差
>>> 0.8164965809277260 3
```

疑似乱数

「ランダム」な数字列、一般的には、方式と過去の出力が既知であれば、未来の出力を予測可能。初期値seedを与える

いろいろなアルゴリズム

- 線形合同法 (古い、相関が強い) 古いCの実装で使われていることも
- メルセンヌ・ツイスタ (松本眞と西村拓士により開発)

周期2^{19937}-1, 高次元空間(623次元)で均等分布, Python randomモジュール, C++11

```
>>> import random
>>> random.seed(100)
>>> random.random()
>>> 0.1456692551041303
>>> random.random()
>>> 0.4549270045140213 5

>>> random.seed(100)
>>> random.random()
>>> 0.1456692551041303

>>> random.randint(0,100)
>>> 58
```

疑似乱数

「ランダム」な数字列、一般的には、方式と過去の出力が既知であれば、未来の出力を予測可能。初期値seedを与える

いろいろなアルゴリズム

- 線形合同法 (古い、相関が強い) 古いCの実装で使われていることも
- メルセンヌ・ツイスタ (松本眞と西村拓士により開発)

周期2^{19937}-1, 高次元空間(623次元)で均等分布, Python randomモジュール, C++11

```
>>> import random
>>> random.seed(100)
>>> random.random()
>>> 0.1456692551041303
>>> random.random()
>>> 0.4549270045140213 5

>>> random.seed(100)
>>> random.random()
>>> 0.1456692551041303

>>> random.randint(0,100)
>>> 58
```

古典スピン模型

$$
\begin{aligned}
Z &= \sum_c e^{-\beta E_c} \\
c &= \{S_1, S_2, \cdots, S_N\}, S_i = \pm 1 \\
\mathcal{H} &= \frac{J_{ij}}{2} \sum^N S_i \cdot S_j + h \sum^N S_i \\
& \quad ij \qquad\qquad\qquad\quad i \\
\langle O \rangle &= \frac{\sum_c O_c e^{-\beta E_c}}{\sum_c e^{-\beta E_c}}
\end{aligned}
$$

参考書: 物性物理におけるモンテカルロ法(第52回物性若手夏の学 校(2007年度),講義ノート)

マルコフ連鎖モンテカルロ法

物性物理におけるモンテカルロ法(第52回物性若手夏の学 校(2007年度),講義ノート)

重み付きサンプリング

$$c_1 \to c_2 \to c_3 \to \cdots$$

$$p_c \propto e^{-\beta E_c} \ (> 0)$$

$$\langle O \rangle = \frac{\sum_c O_c e^{-\beta E_c}}{\sum_c e^{-\beta E_c}} = \lim_{k \to +\infty} \frac{1}{N_k} \sum_k O_{c_k}$$

マルコフ過程

時間に依存した一種の確率過程。未来の挙動が現在の挙動だけで決まる。
単純マルコフ過程とは、単一の状態から未来の挙動が決まるもの。

$$\cdots \rightarrow c_{k-1} \rightarrow c_k \rightarrow c_{k+1} \rightarrow \cdots$$

c_k のみから(確率的に)決まる

実用的には、任意の2つの状態間の遷移確率 $Pc \rightarrow c'$ がcのみに依存するように設計する。

マルコフ過程の設計方針

与えられた出現確率に従うマルコフ過程をどう設計すればいいか？

- Balance condition
- Ergodicity

Balance condition

$$\sum_j p_i P_{i\to j} = \sum_j p_j P_{j\to i}$$

状態iから出る確率流　　状態iに入る確率流

一般に設計するのは難しい→よりきつい条件を考える

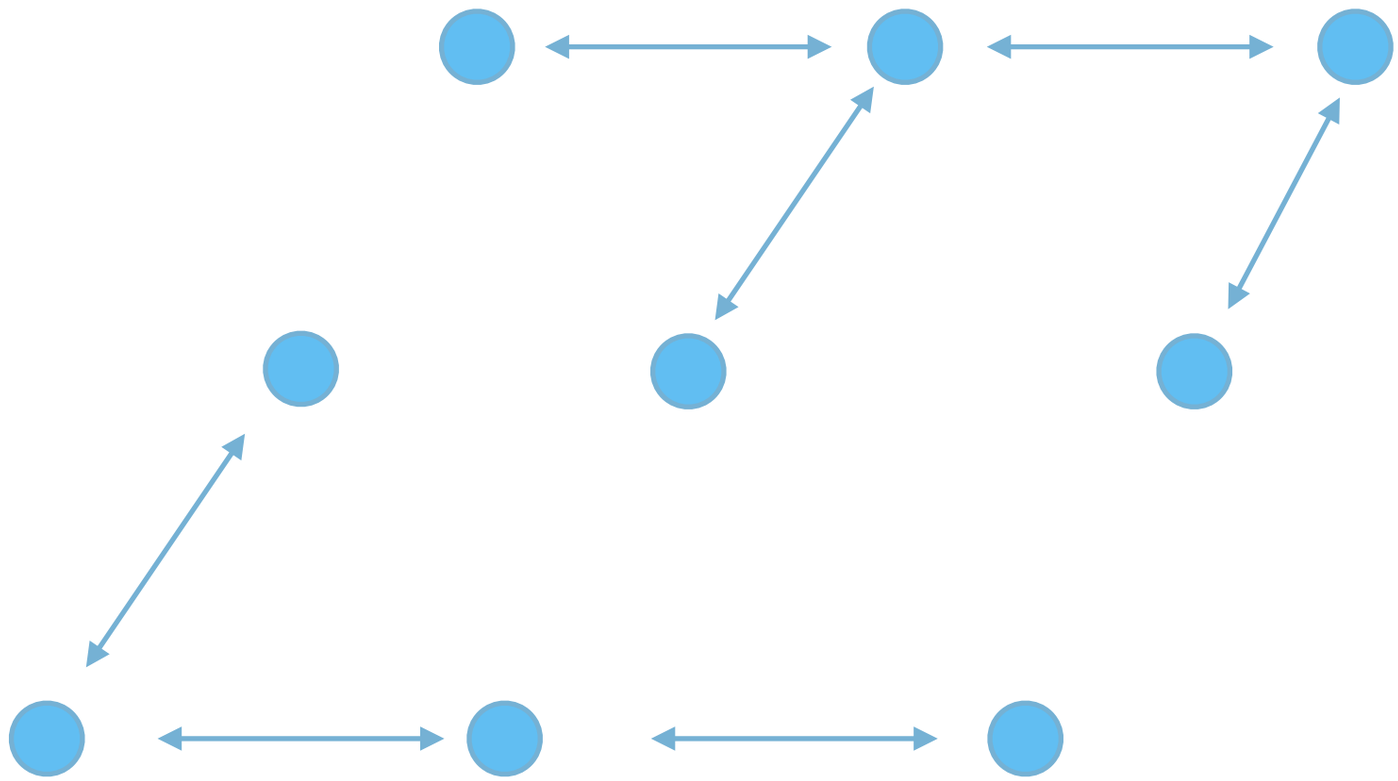
Detailed balance condition

任意のペアi, jに対して、以下の条件の成立を要求すると、
自動的にbalance conditionが満たされる。

$$p_i P_{i \to j} = p_j P_{j \to i}$$

Ergodicity

任意の状態iから状態jへ有限回の遷移で到達可能。以下だめな例



具体例: 古典スピン模型

$$
\begin{aligned}
Z &= \sum_c e^{-\beta E_c} \\
c &= \{S_1, S_2, \cdots, S_N\}, S_i = \pm 1 \\
\mathcal{H} &= \frac{J_{ij}}{2} \sum_{ij}^{N} S_i \cdot S_j + h \sum_{i}^{N} S_i \\
\langle O \rangle &= \frac{\sum_c O_c e^{-\beta E_c}}{\sum_c e^{-\beta E_c}}
\end{aligned}
$$

シングルスピンフリップ

1.ランダムに一つスピンを選ぶ
2.スピンが上向き、下向きの状態のエネルギーを計算
3.以下の確率で、どちらかの状態を選ぶ。
4.1に戻る

$$P_{\uparrow} = e^{-\beta E_{\uparrow}}/(e^{-\beta E_{\uparrow}} + e^{-\beta E_{\downarrow}})$$
$$P_{\downarrow} = e^{-\beta E_{\downarrow}}/(e^{-\beta E_{\uparrow}} + e^{-\beta E_{\downarrow}})$$

最近接相互作用の場合、確率の計算量はO(1)である。

帯磁率の計算法(1)

$$\frac{\partial \langle m_z \rangle}{\partial h} = \beta N(\langle m_z^2 \rangle - \langle m_z \rangle^2)$$

$$m_z = \sum_i S_i / N$$