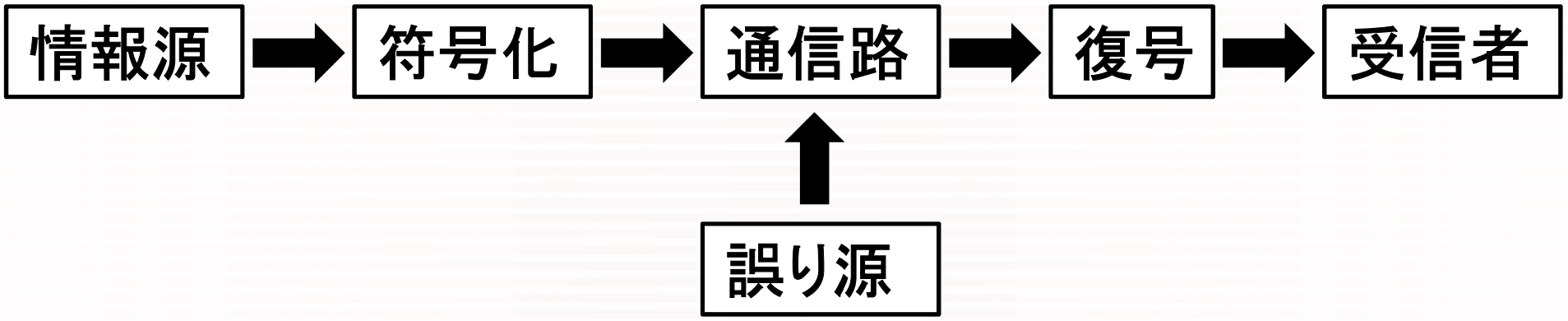
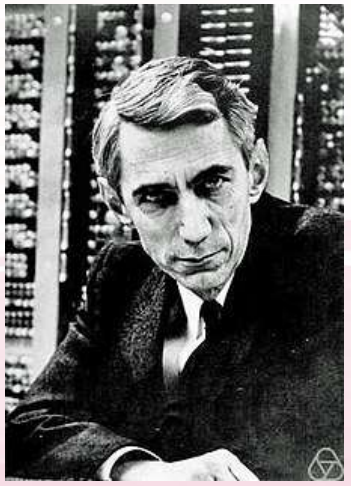
情報源符号化(データ圧縮)

通信における情報伝達のモデル

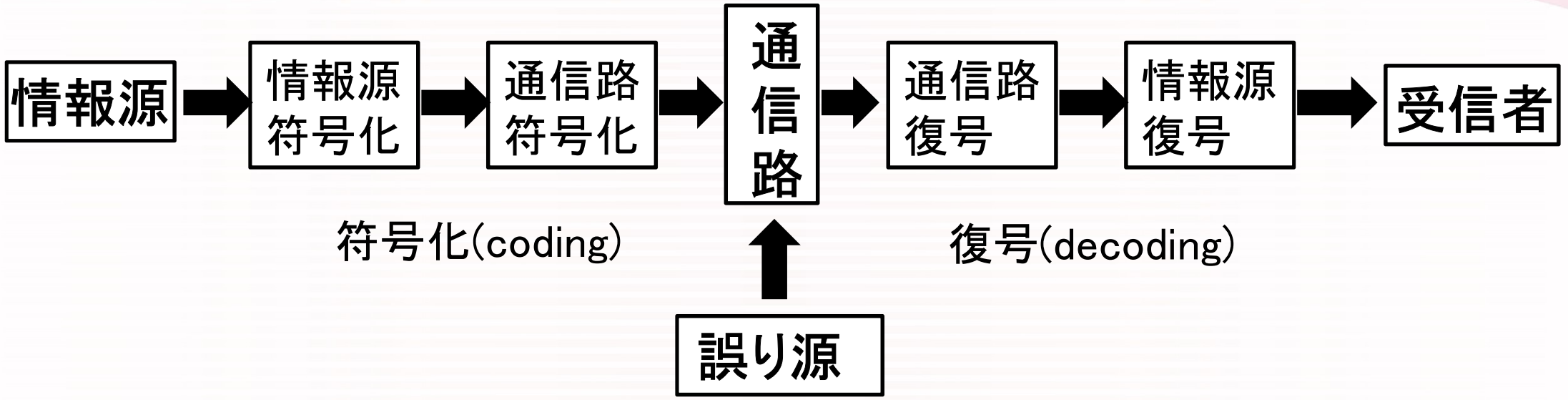


クロード・シャノン
「通信の数学的理論」1948年



Wikipedia

通信における情報伝達のモデル



情報源符号化：伝送効率向上のための符号化
（できるだけ少ない記号で）

通信路符号化：信頼性確保のための符号化
（通信路の誤りの影響を抑える冗長性確保）

情報の定量化

通報(事象)の生起確率が p のとき、その情報量 $f(p)$ を定義する

1. 確率が小さい事象が起こったという通報のほうが
   確率が大きい事象の通報より情報量が大きい。
   すなわち、$f(p)$ は p の減少関数である。

2. 独立事象E_1, E_2の生起確率をp_1, p_2とすると、E_1, E_2 が
   同時に起こる確率は $p_1 p_2$ であるから、
   $$f(p_1 p_2) = f(p_1) + f(p_2)$$

3. $f(p)$ は p の連続関数である。近い確率の事象の情報量は近い。

   $\Rightarrow f(p)$ は $-\log p$ の正の定数倍

情報の定量化

関数方程式 $f(p_1 p_2) = f(p_1) + f(p_2)$ を解く。

$p_1 = 2^{-q_1}$, $p_2 = 2^{-q_2}$とおくと、q_1, q_2は0以上の実数値をとる。
そこで、

$$g(x) = f(2^{-x})$$

とおけば、

$$g(q_1 + q_2) = g(q_1) + g(q_2)$$

を得る。

情報の定量化

定義: 事象 E の生起確率が p であるとき、
その情報量を $-\log_2 p$ と定める。(単位: ビット)

定義: M個の独立な通報 (事象) $a_1, \ldots, a_M$があり、各通報が送られる確率(事象の生起確率)が$p_1, \ldots, p_M$であるとする $(p_1 + \cdots + p_M = 1)$。このとき、
1通報(事象)あたりの平均情報量 (エントロピー) $H(A)$ を

$$H(A) = -\sum_{i=1}^{M} p_i \log_2 p_i$$

と定める。(= 独立生起情報源のエントロピー)

情報の定量化

例題: 4つの通報a_1, a_2, a_3, a_4をそれぞれ確率0.6, 0.2, 0.1, 0.1で発生する独立生起情報源がある。この情報源から発生する通報のエントロピーを求めよ。

解: 定義に従って、

$$-(0.6\log_2 0.6 + 0.2\log_2 0.2 + 0.1\log_2 0.1 + 0.1\log_2 0.1)$$

= 1.57 ビット。

エントロピーの性質

(1) エントロピーは0以上である。

証明:情報量が0以上であることから従う。

(2) すべての事象の生起確率が等しいとき、
エントロピーは最大になる。

証明:$\log_e x \leq x - 1 \quad (x > 0)$ を用いる
(等号成立は$x = 1$のとき)。

エントロピーの性質

(2) すべての事象の生起確率が等しいとき、
エントロピーは最大になる。

情報源符号化

符号化:　情報源記号系列からなる通報を
通信路記号系列に1対1に割り当てる

以下簡単のため、0と1のみからなる二元符号を扱う
(多元符号に一般化可能)

通報	符号I	符号II	符号III	符号IV
a_1	0	0	1	00
a_2	1	01	01	01
a_3	01	011	001	10
a_4	10	0111	0001	11

符号語: 割り当てられた通信路記号系列

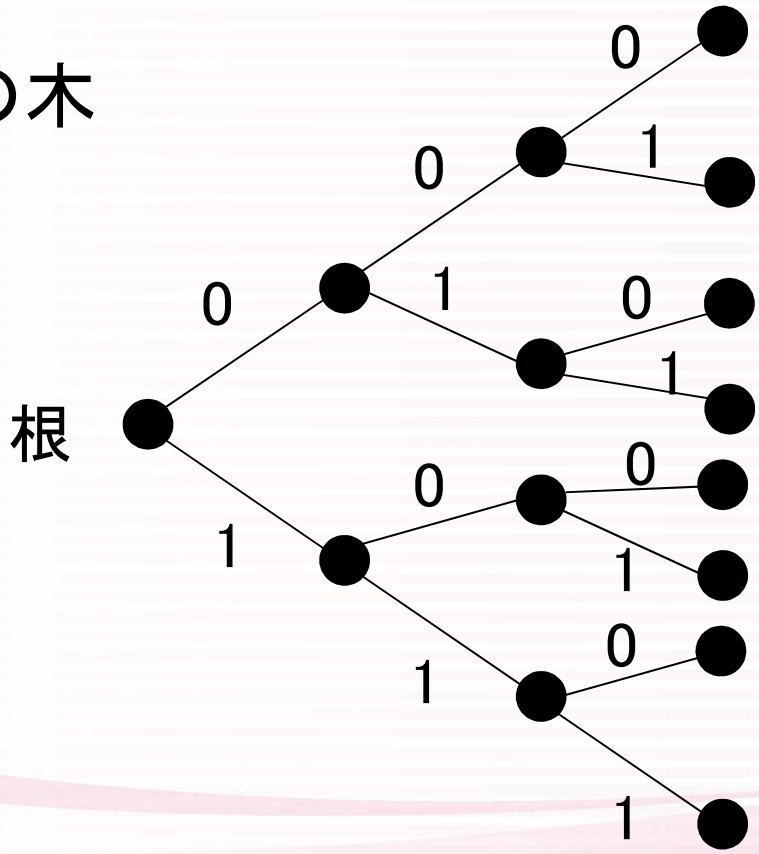
情報源符号化

通報	符号I	符号II	符号III	符号IV
a_1	0	0	1	00
a_2	1	01	01	01
a_3	01	011	001	10
a_4	10	0111	0001	11
一意復号可能?	×	○	○	○
瞬時復号可能?	×	×	○	○

木

木 (tree): 連結で、サイクルを持たない無向グラフ

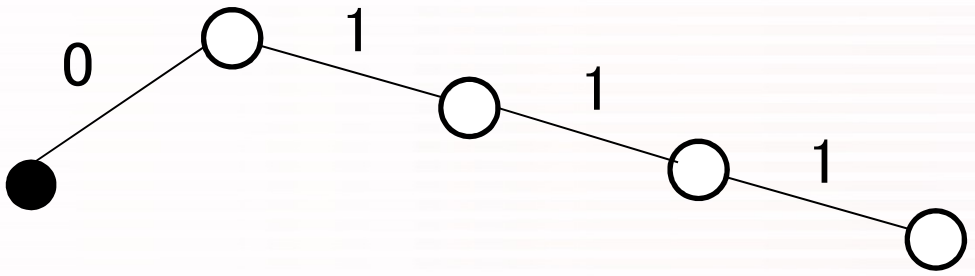
符号の木



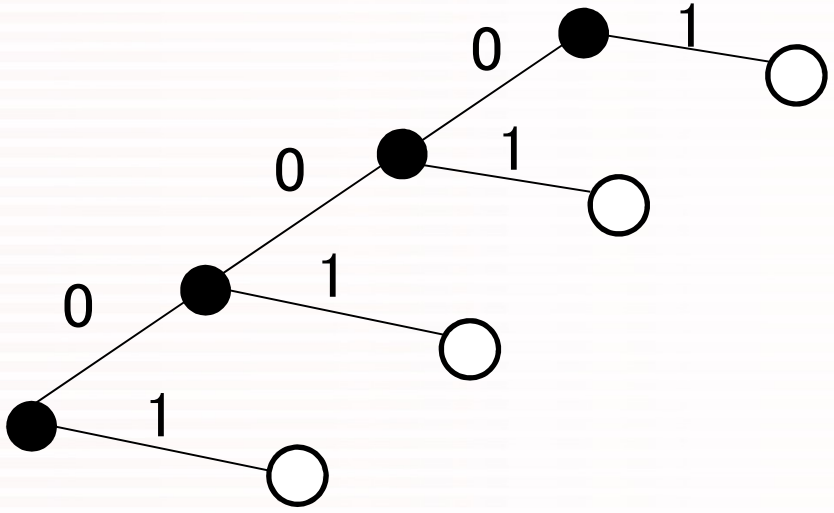
葉：根以外の頂点のうち、
接続する辺が1本しかないもの

符号の木

符号IIの木



符号IIIの木



符号が瞬時復号可能であるための必要十分条件は、
符号の木において、
すべての符号語○が葉に対応づけられること。

クラフトの不等式

符号語長が $l_1, \ldots, l_M$ のM個の符号語からなる、
瞬時復号可能な2元符号が構成できるための必要十分条件は

$$\sum_{i=1}^{M} 2^{-l_j} \leq 1$$

が成り立つことである。

(証明の方針) 符号の木を $\max(l_1, \ldots, l_m)$ 次まで伸ばして、
葉の個数を数える。
(注)「瞬時復号可能」を「一意復号可能」に変えても
成り立つことが知られている(マクミランの不等式)。

平均符号長

通報 $a_1, \ldots, a_M$ の生起確率をそれぞれ $p_1, \ldots, p_M$、
符号長をそれぞれ $l_1, \ldots, l_M$とすると、

1通報あたりの平均符号長Lは

$$L = \sum_{i=1}^{M} p_i l_i$$

で与えられる。

平均符号長

例題: 4つの通報a_1, a_2, a_3, a_4をそれぞれ確率0.6, 0.2, 0.1, 0.1で発生する独立生起情報源がある。この情報源を符号IIIで符号化したときの平均符号長Lを求めよ。

解: 定義に従って、
$L = 1 \times 0.6 + 2 \times 0.2 + 3 \times 0.1 + 4 \times 0.1 = 1.7$ ビット。

エントロピーが$H = 1.57$ビットだったことを思い出そう。

情報源符号化定理

一般に、瞬時復号可能な2元符号に対して

$$H \leq L$$

が成り立つ(注1)。また、

$$H \leq L \leq H + 1$$

をみたすような符号化が存在する(注2)。

(注1) 平均符号長 L はエントロピー H より小さくなれない。

(注2) 通報a_iに対応する符号語の長さl_iを、 $-\log_2 p_i$ 以上の最小の整数にとればよい(確率の高い通報の符号長を短く)

ハフマン符号

与えられた独立生起情報源からの通報を符号化するとき、
平均符号長を最小にする符号を最短符号(compact code)という。

最短符号の構成法として、ハフマン符号が知られ、
JPEGやZIPなどの圧縮フォーマットで用いられている。

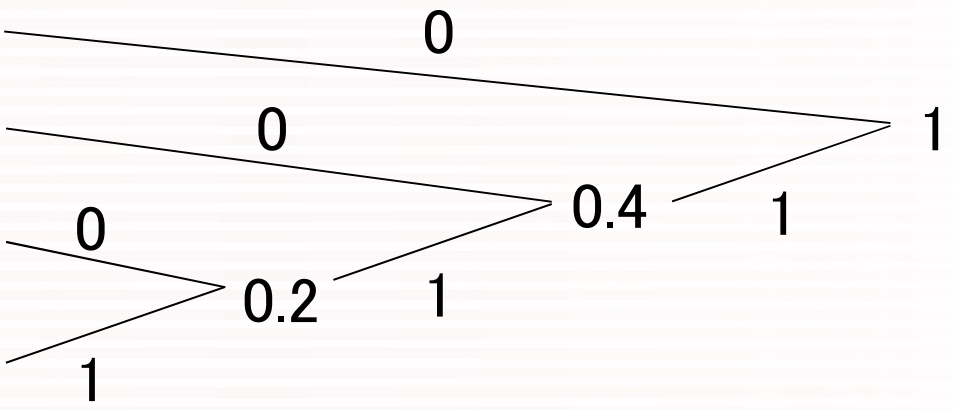
構成法：

1. *M*個の通報に対応する葉を作る。生起確率の最も小さい葉を2つ選び、枝で結んで頂点を作る。それぞれの枝に0, 1を割り当てる。
2. この頂点を新しい葉とみなし、もとの2つの葉の生起確率の和を新たな葉の生起確率とする。
3. 葉が1枚になるまで繰り返し、そこからもとの葉に至るまでの枝に割り当てられた数字を順に読む

ハフマン符号

例題: 4つの通報a_1, a_2, a_3, a_4をそれぞれ確率0.6, 0.2, 0.1, 0.1で発生する独立生起情報源がある。この通報を2元ハフマン符号で符号化せよ。

符号語	通報	確率
0	a_1	0.6
10	a_2	0.2
110	a_3	0.1
111	a_4	0.1



この符号の平均符号長は$1 \times 0.6 + 2 \times 0.2 + 3 \times 0.1 + 3 \times 0.1$ = 1.6ビットである。

データ構造（リスト、配列、木構造）

アルゴリズムとデータ構造

アルゴリズム

計算機を用いて何らかの問題を解く際の手続き(解き方)

プログラム

その解き方を計算機上で実際に実行可能な命令列として
表現したもの

データ構造

計算機を用いて計算や処理を効率的に行うのに適した
データの保持方法

コンピュータ(PC)のハードウェア構成

CPU (中央演算処理装置, Central Processing Unit)

命令を高速で処理する
クロック周波数：1秒間に何回命令処理ができるか
(GHz = 1秒間に10億回)

メインメモリ

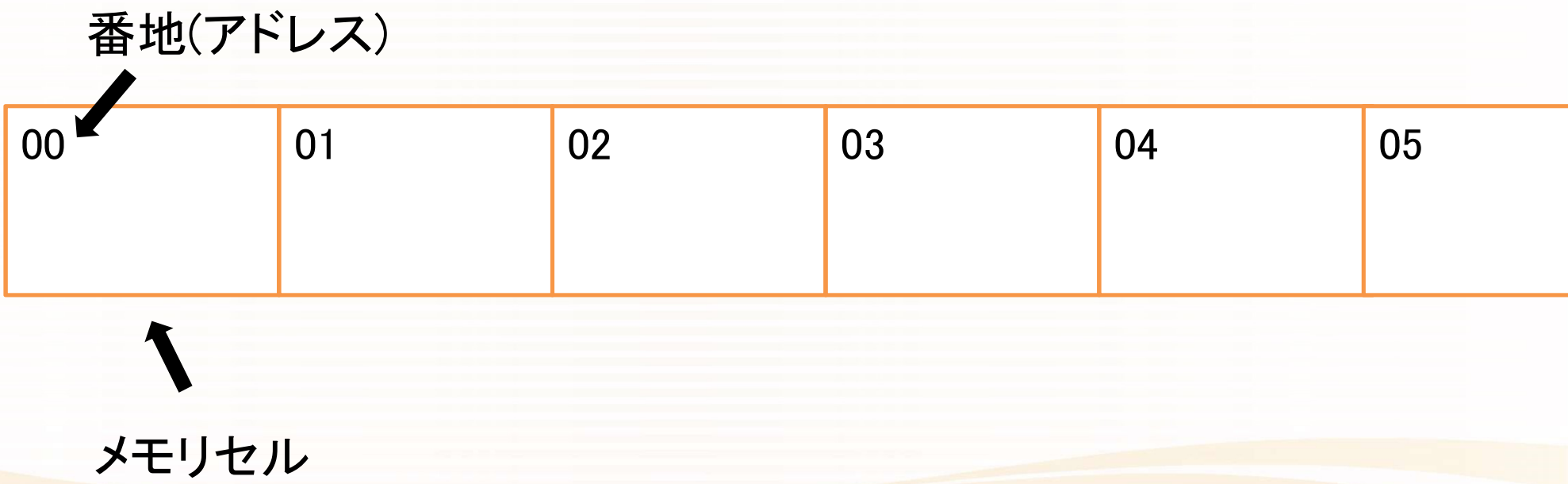
データの書き込みと読み込みに特化。CPUとはバスで通信。
電源を切ると内容が失われる(Dynamic RAM)

外部記憶装置・周辺機器(ディスプレイ・キーボードなど)

コントローラを介してバスに接続

メモリの構造の模式図

コンピュータで実行されるプログラムや、プログラムが
扱うデータを一時的に保持



データ構造とクエリ処理

・要素 x をデータ構造に挿入する

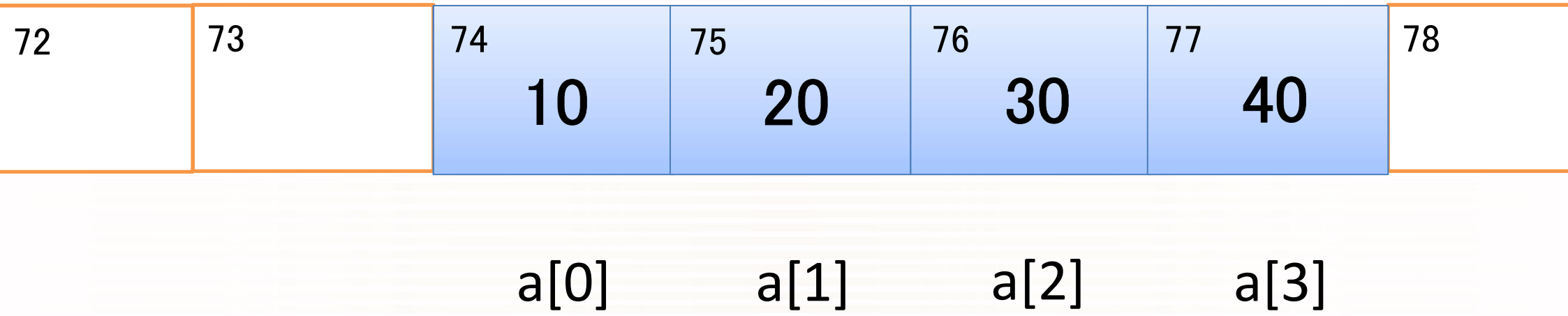
・要素 x をデータ構造から削除する

・要素 x がデータ構造に含まれるかどうかを判定する

⇒ 使用するデータ構造によって計算時間に差が生じる

配列(array)

連続するメモリ領域を必要なだけ確保することで、
順番を保持してメモリに記録できる



C++では std::vector、Pythonではlist

配列(array)の計算量

配列のサイズをNとする。

i 番目の要素へのアクセス : $O(1)$

要素 x を最後尾に挿入 : $O(1)$

要素 x を特定の要素 y の直後に挿入 : $O(N)$

要素 x を削除 : $O(N)$

要素 x を検索 : $O(N)$

ビッグ・オー記法

$O(1)$: Nによらない
定数

$O(N)$: Nの
1次式(以下)

ビッグ･オー記法

アルゴリズムの計算量を、入力サイズnの関数として表すとき、その漸近的上界を示す。

関数$f(x), g(x)$に対し、十分大きなnについて

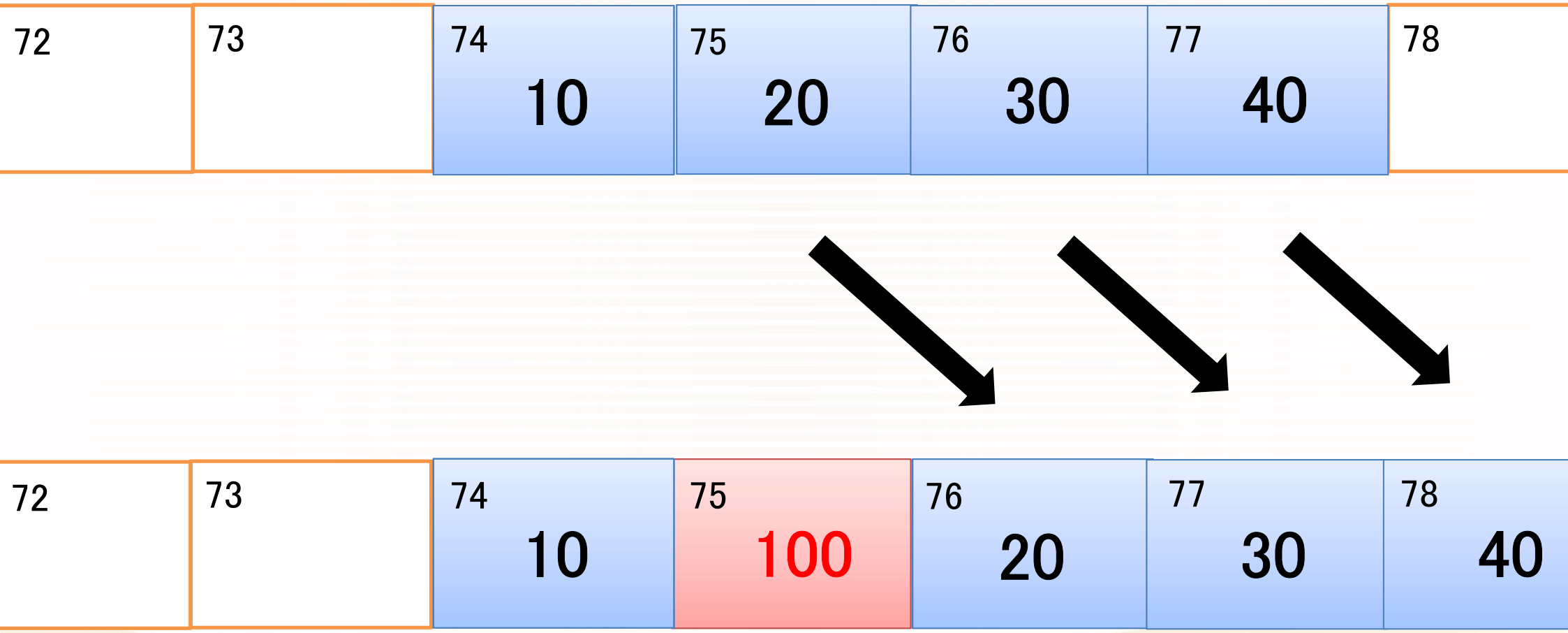
$$f(n) \leq c\, g(n)$$

となる定数cが存在するならば、

$$f(n) = O\big(g(n)\big)$$

と表し、$f(n)$のオーダーは$g(n)$であるという。

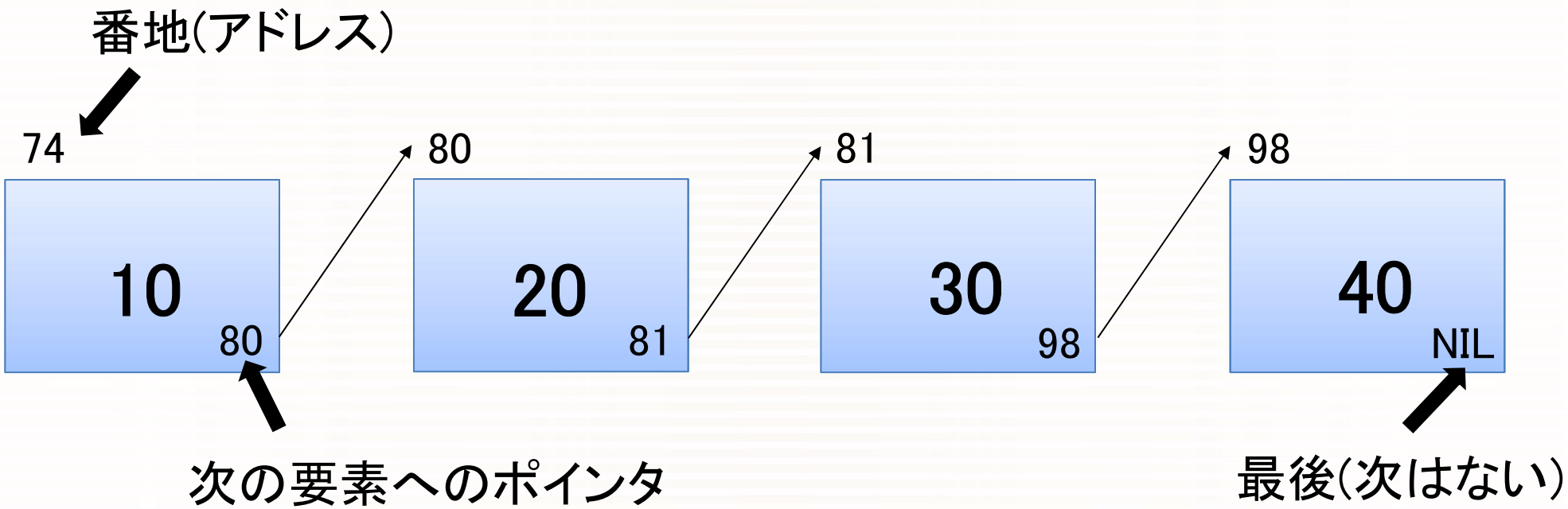
配列における要素の挿入



連結リスト(linked list)

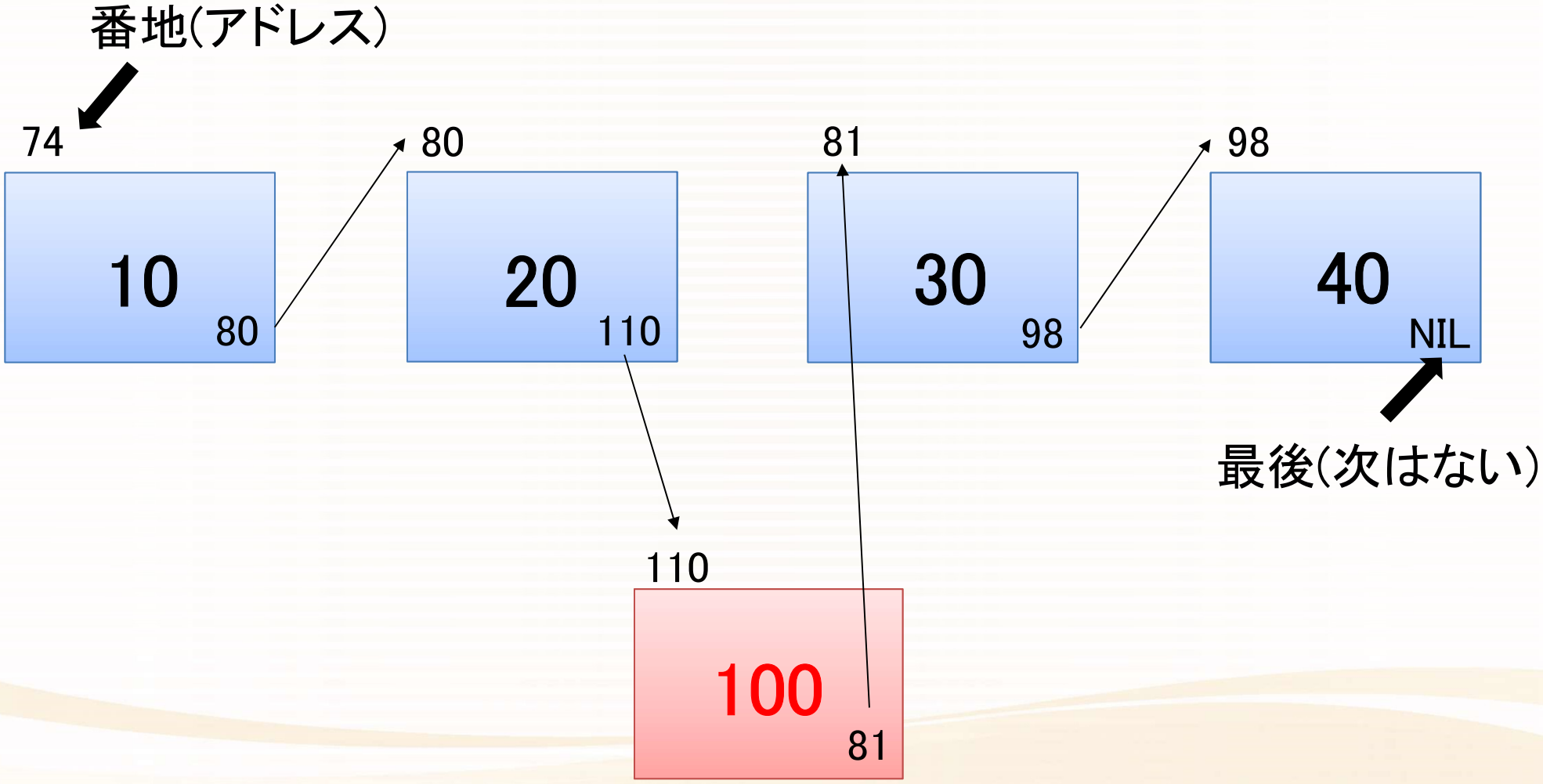
配列の弱点である挿入・削除クエリに強い

C++では
std::list



図：単方向連結リストの場合。双方向にもできる
(次の要素と前の要素へのポインタをもつ)

連結リストにおける要素の挿入



連結リスト(linked list)の計算量

配列のサイズをNとする。

i 番目の要素へのアクセス: $O(N)$

要素 x を最後尾に挿入: $O(1)$

要素 x を特定の要素 y の直後に挿入: $O(1)$

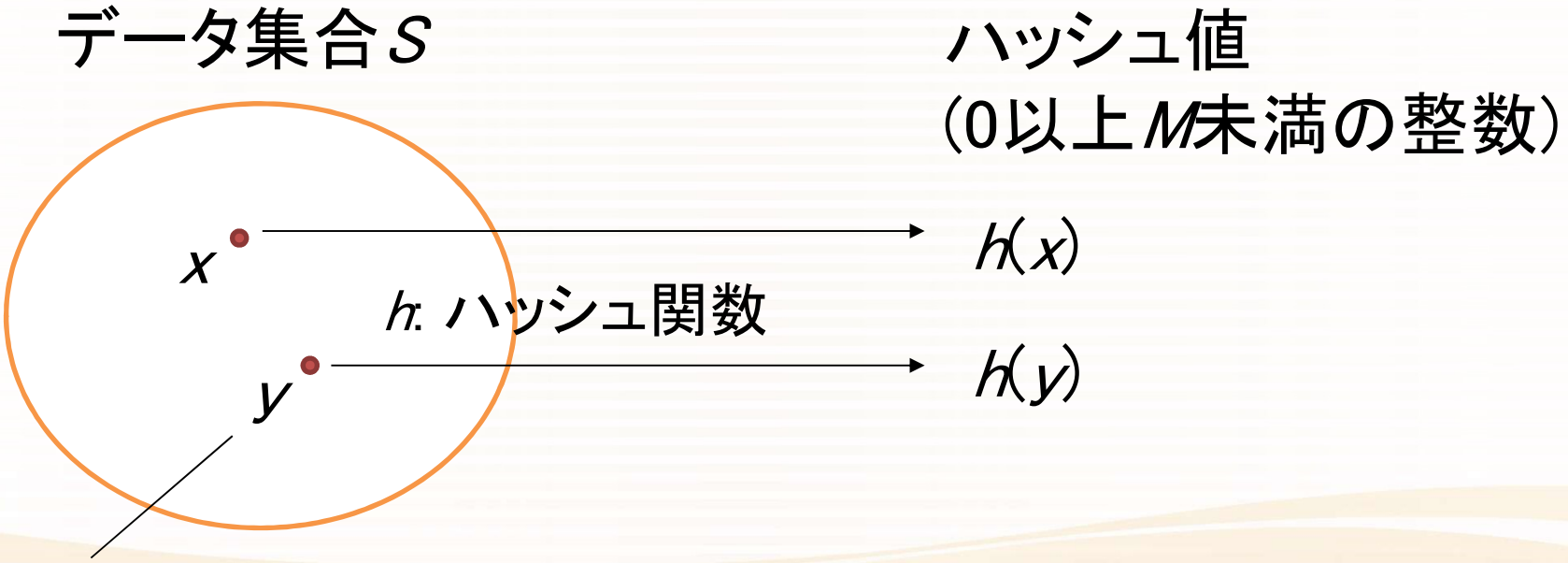
要素 x を削除: $O(1)$

要素 x を検索: $O(N)$

ハッシュテーブル

要素の検索が$O(1)$の計算量で
できるが、各要素間の順序に
関する情報をもたない

C++では std::unordered_set、
Pythonではset



ハッシュテーブル

配列 T を用意して、

要素 x の挿入: $T[h(x)] \leftarrow$ true

要素 x の削除: $T[h(x)] \leftarrow$ false

要素 x の検索: $T[h(x)]$ が true かどうか

いずれも計算量は平均的に $O(1)$。

ただし、ハッシュ値が等しい $h(x) = h(y)$ ものがある場合は、
それらで連結リストを作り、$T[h(x)]$ にその連結リストの
先頭を指すポインタを入れる。

スタックとキュー

配列や連結リストを用いて実現可能

クエリ
push(x): 要素 x をデータ構造に挿入する
pop(): データ構造から要素を1つ取り出す
isEmpty(): データ構造が空かどうかを調べる

C++ では std::stack, std::queue

スタックとキュー

pop() でデータ構造から要素を1つ取り出す際に、

スタックの場合は、last-in first-out (LIFO)、すなわち
最後に push された要素を取り出す。

例: Webブラウザの訪問履歴 (戻るボタンがpop)、
テキストエディタのUndo

配列での実現：左側が閉じているイメージ、
行き止まりのトンネルに要素を突っ込むイメージ

スタックとキュー

pop() でデータ構造から要素を1つ取り出す際に、

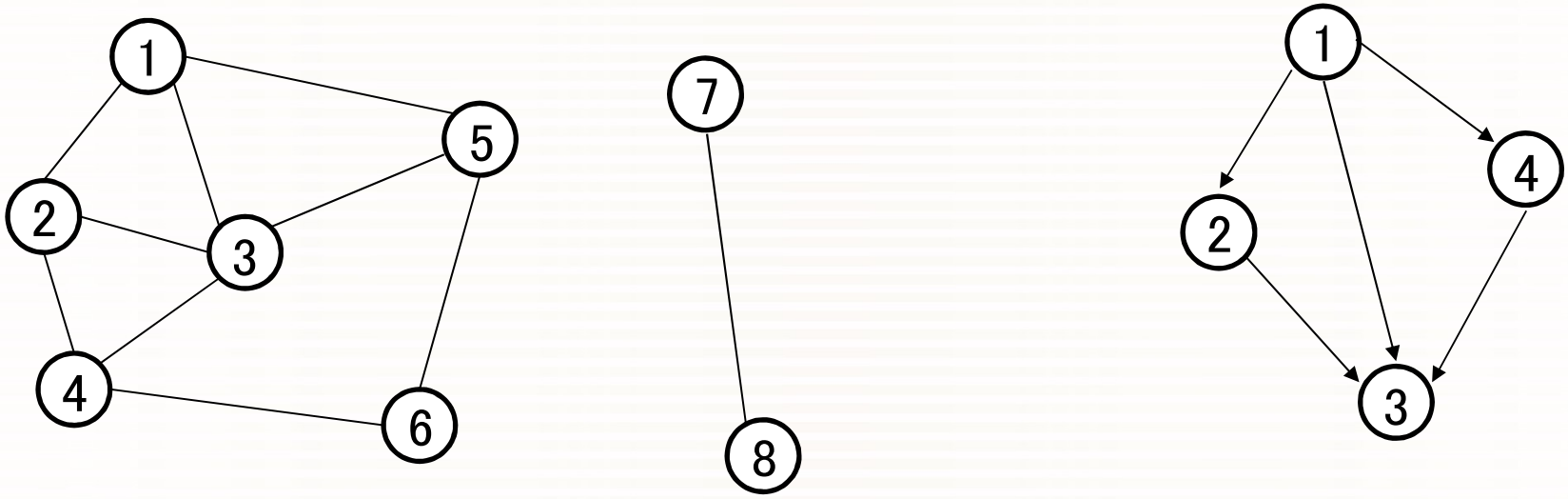
キューの場合は、first-in first-out (FIFO) 、すなわち
最初に push された要素を取り出す。

例: 航空券予約のキャンセル待ち処理、印刷機の
ジョブスケジューリング

配列での実現：両端が開いているイメージ。
右からenqueueして、左からdequeueする。

グラフと木

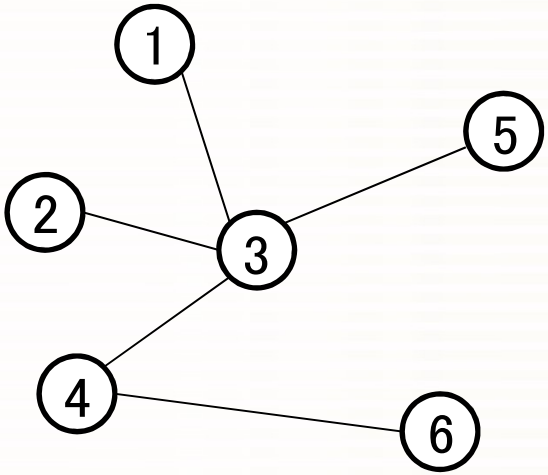
無向グラフ：頂点の集合 V と辺の集合 E の組
有向グラフ：グラフの各辺に向きがあるもの

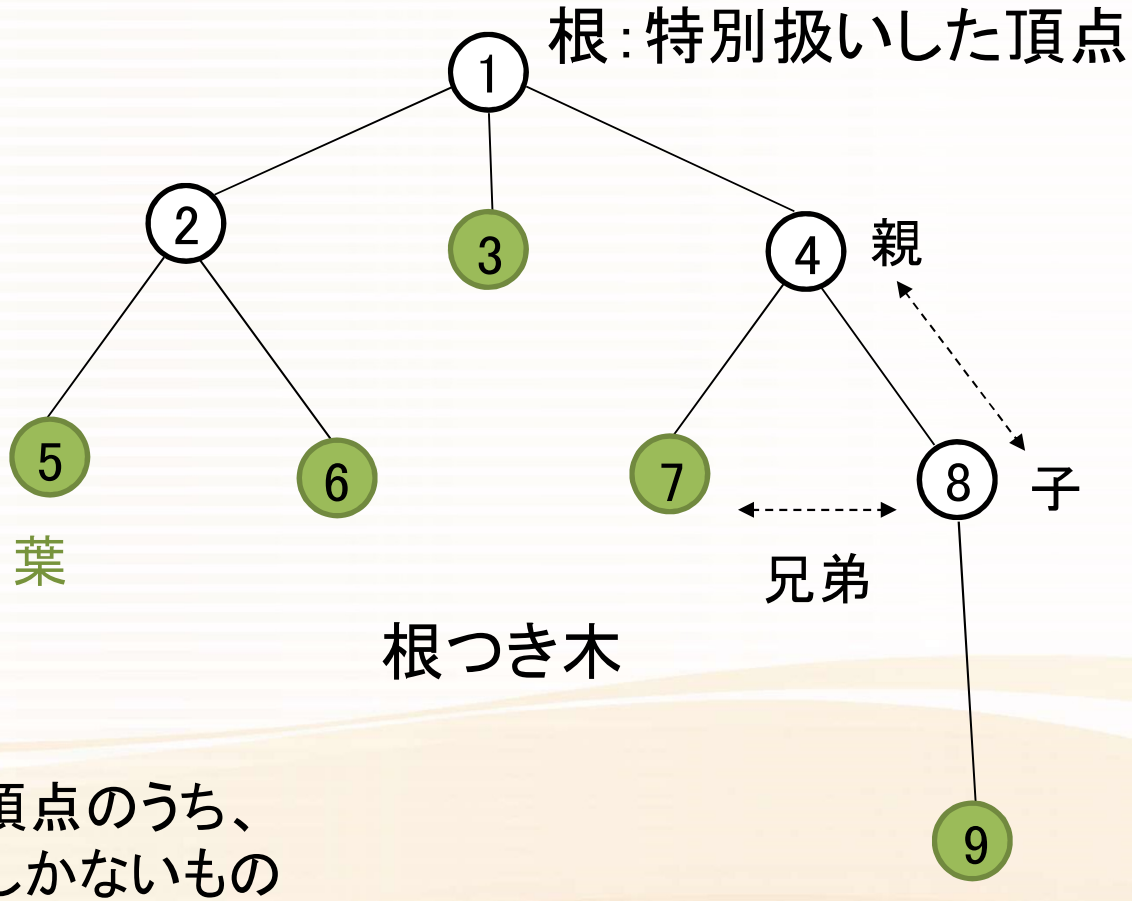


無向グラフの例: ソーシャルネットワーク、交通ネットワーク
有向グラフの例: タスクの依存関係、ゲームの局面遷移

グラフと木

木 (tree): 連結で、サイクルを持たない無向グラフ



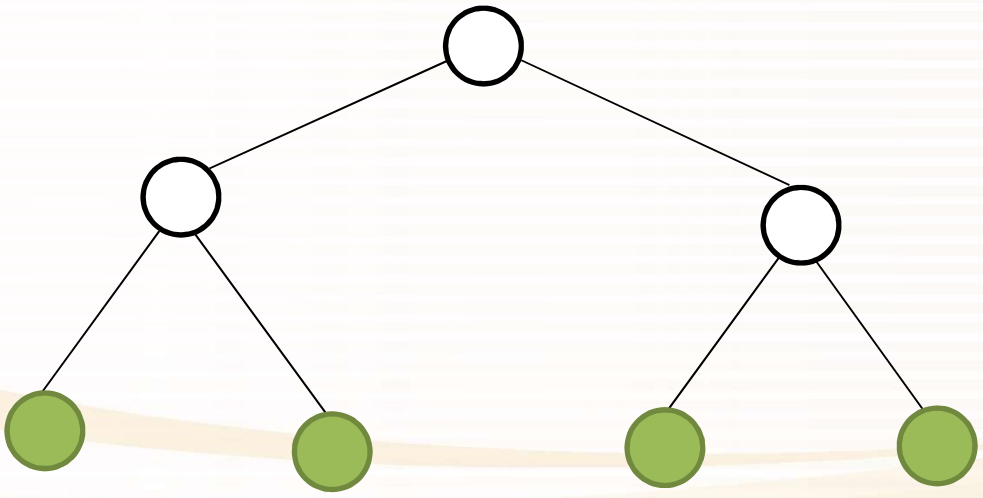


葉：根つき木において、根以外の頂点のうち、
その頂点に接続している辺が1本しかないもの

二分木 (binary tree)

順序木: 根つき木において、各頂点の子頂点の順序を考慮したもの。兄弟間で兄と弟の区別がつく

二分木: 順序木において、すべての頂点が高々 2 個の子頂点をもつもの。



葉

深さ: 根と頂点を結ぶパスの長さ
高さ: 深さの最大値

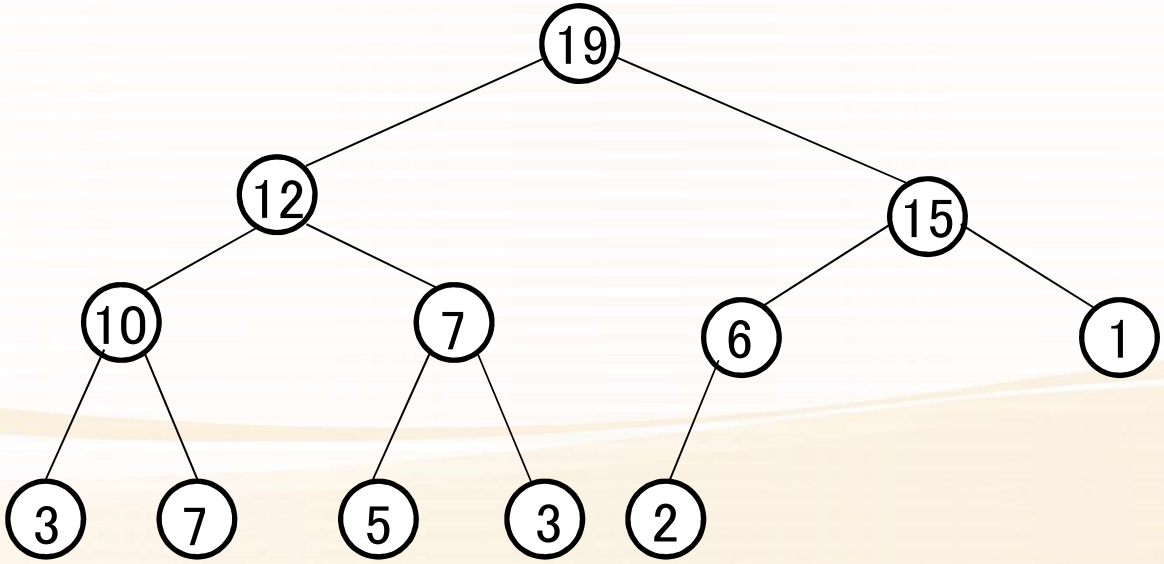
完全二分木: すべての葉の深さが等しい (左例ではすべて2)

完全二分木では、頂点数がNのとき深さは $O(\log N)$

二分ヒープ

各頂点 v がキーとよばれる値 key[v] をもつ二分木で、
以下の条件をみたすものを(二分)ヒープという。

・頂点 v の親頂点 p に対して、key[p] ≥ key[v]
・木の高さを h とすると、木の深さ h−1 以下の部分は
　完全二分木であり、木の深さ h の部分は左詰め

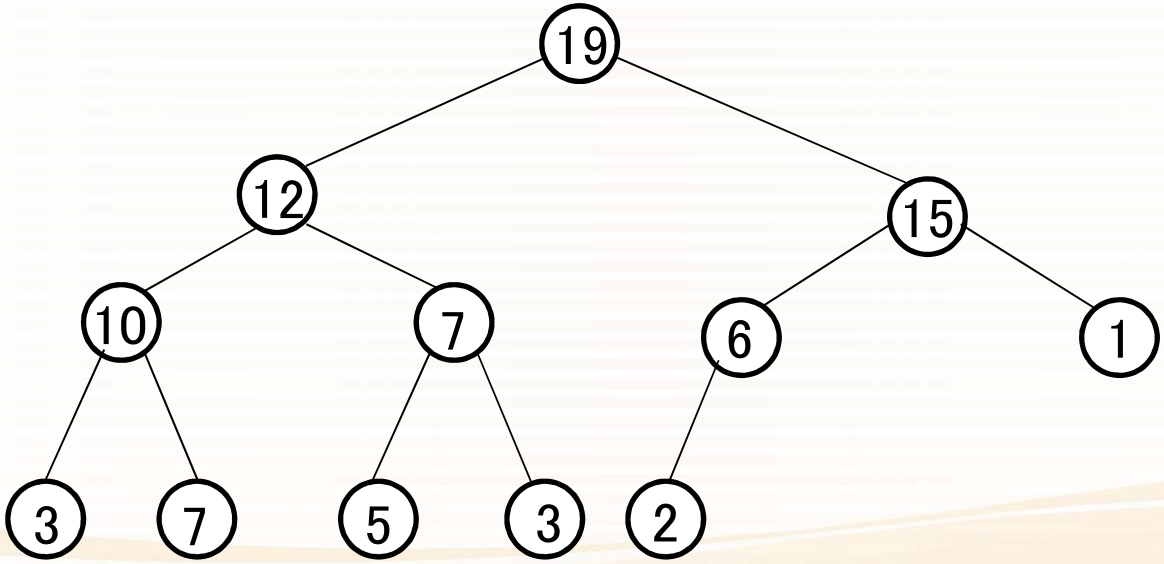


ヒープのクエリ処理

値 x を挿入する: 挿入した後、形を整える。$O(\log N)$

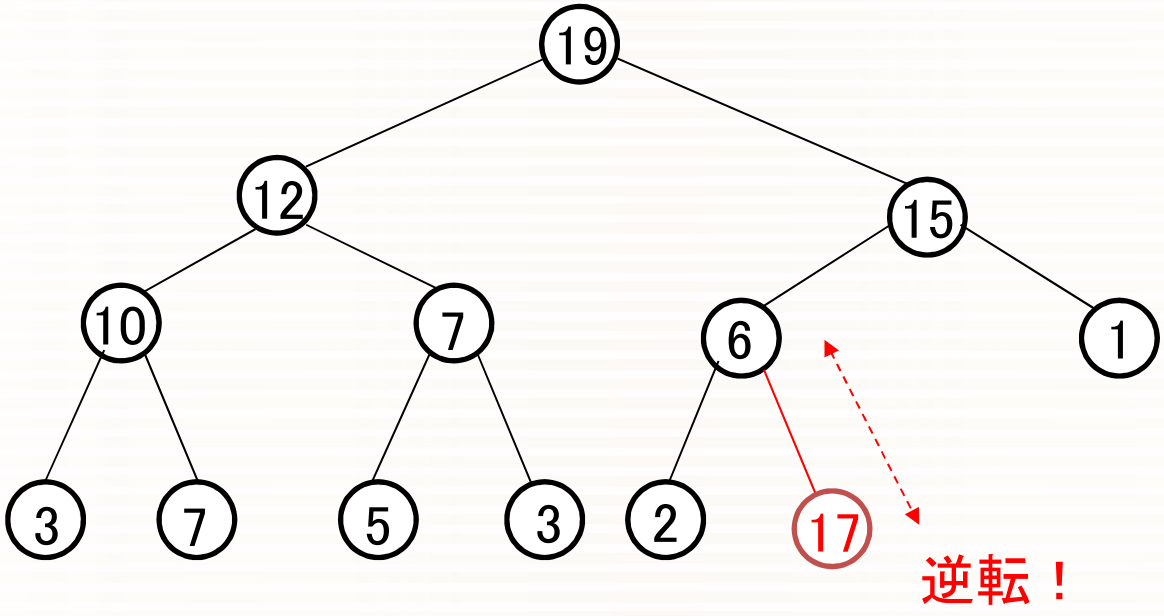
最大値を取得する: 根の値を取得する。$O(1)$

最大値を削除する: 根を削除後、形を整える。$O(\log N)$



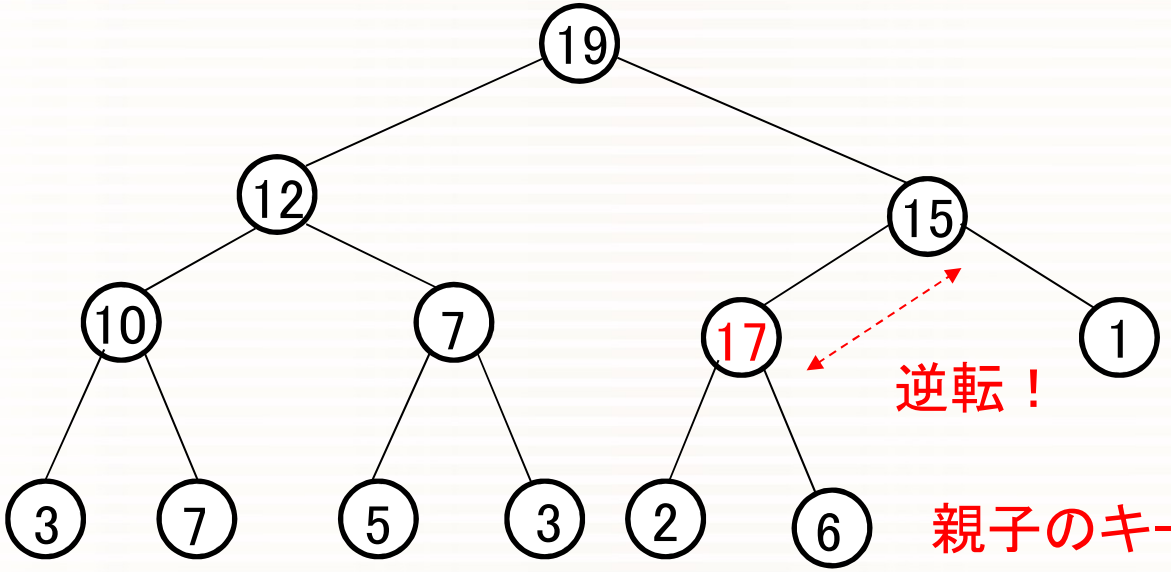
配列による実現: 19, 12, 15, 10, 7, 6, 1, 3, 7, 5, 3, 2

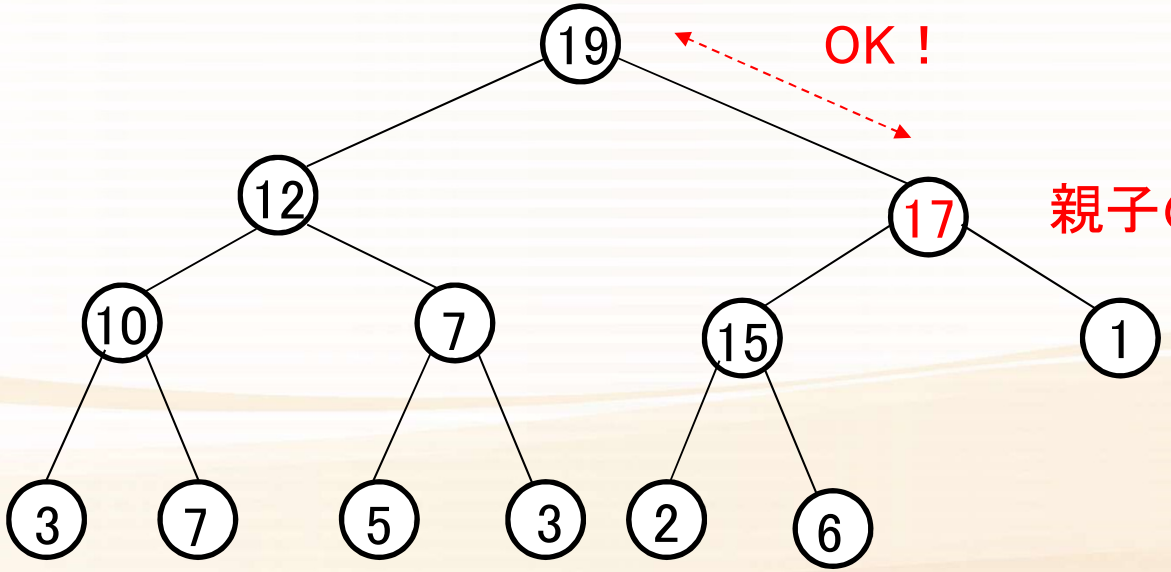
ヒープに値を挿入する



ヒープの最後尾に挿入

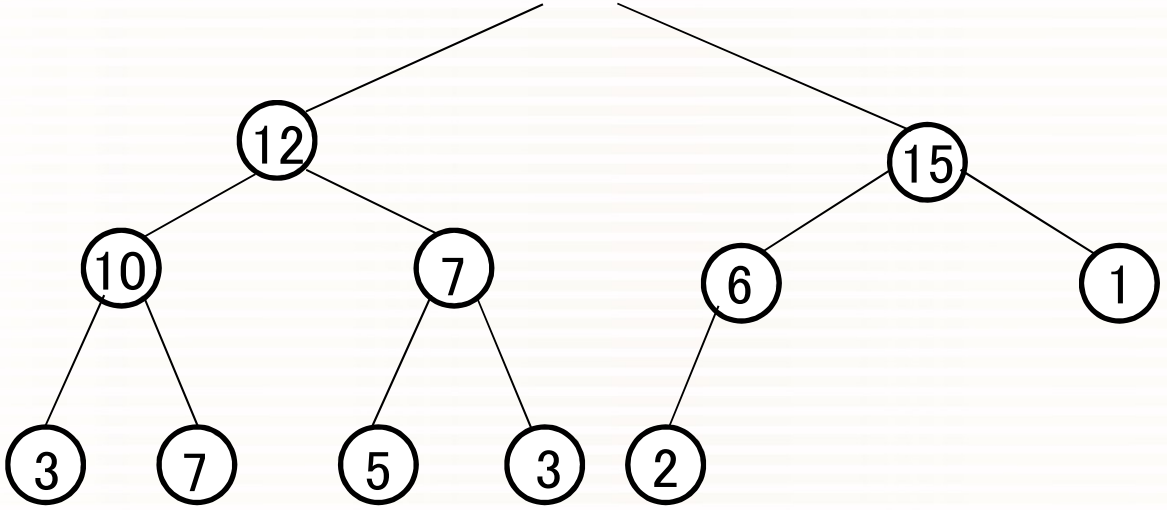
ヒープに値を挿入する

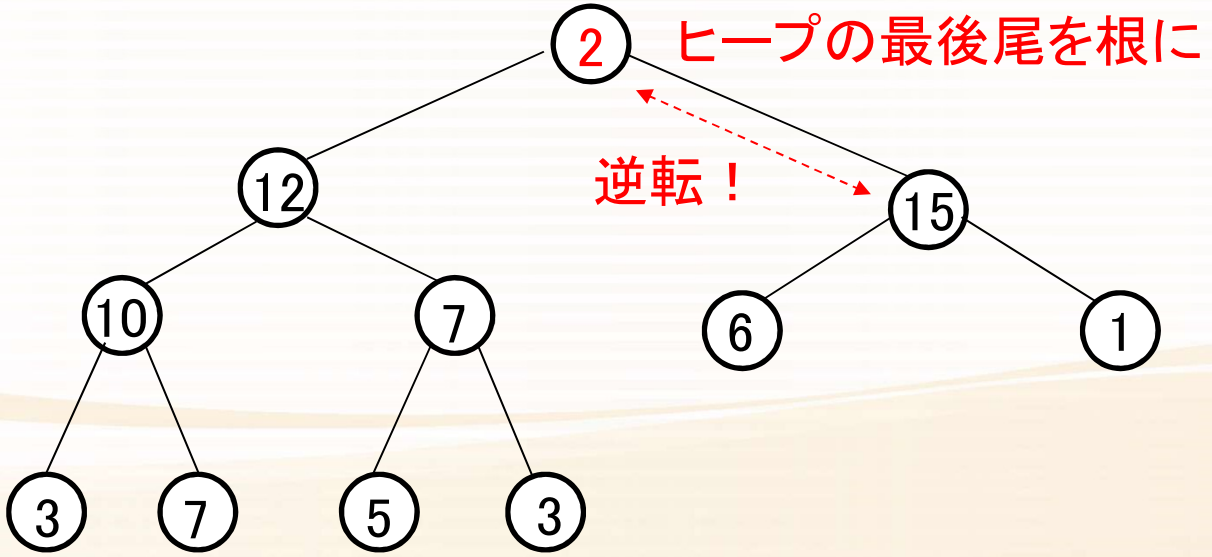




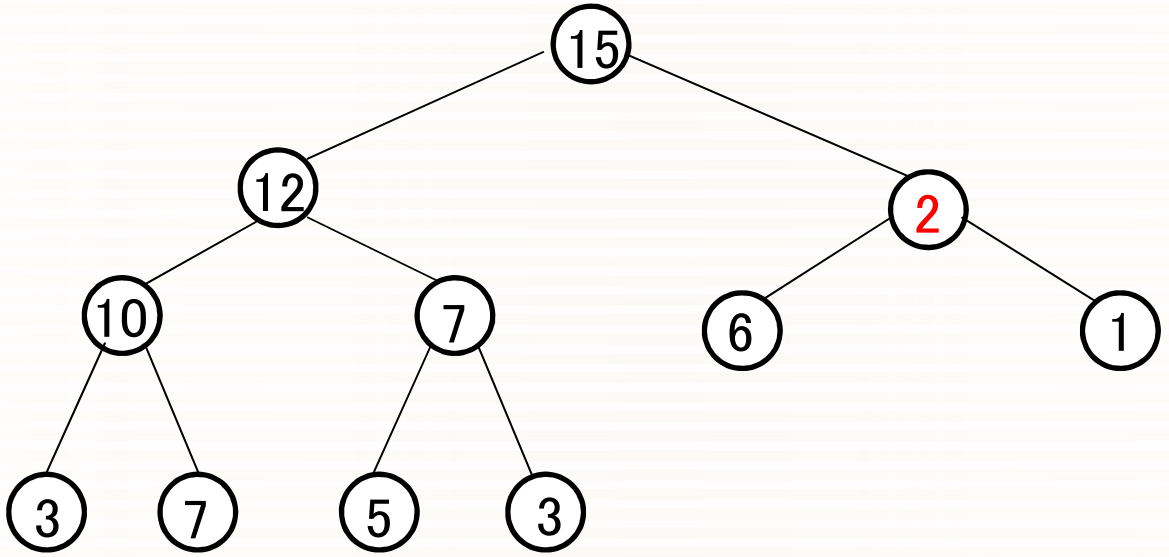
計算量は
木の高さ $O(\log N)$

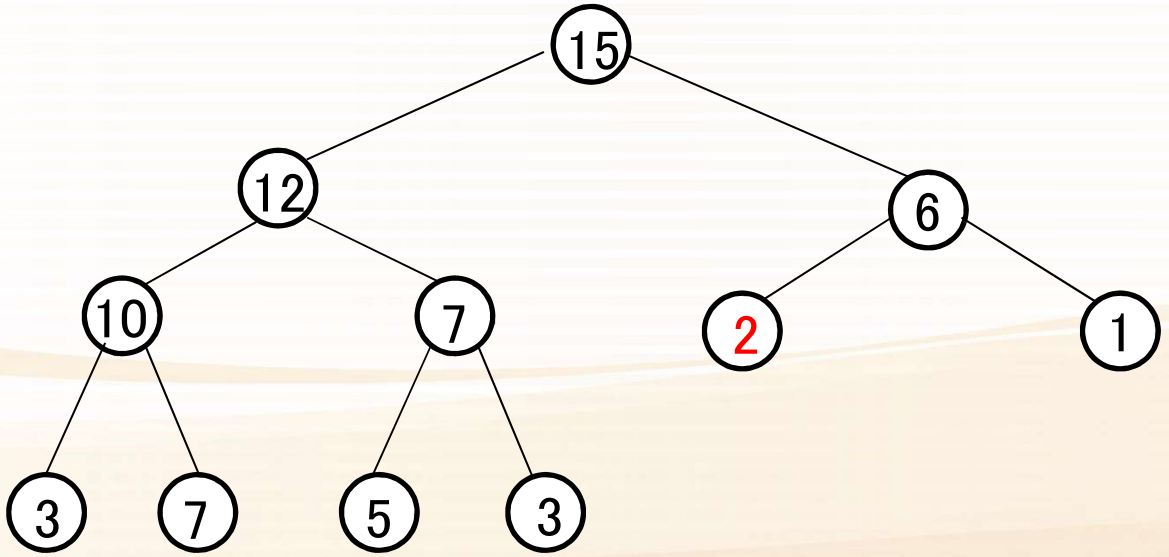
ヒープから最大値を削除する





ヒープから最大値を削除する



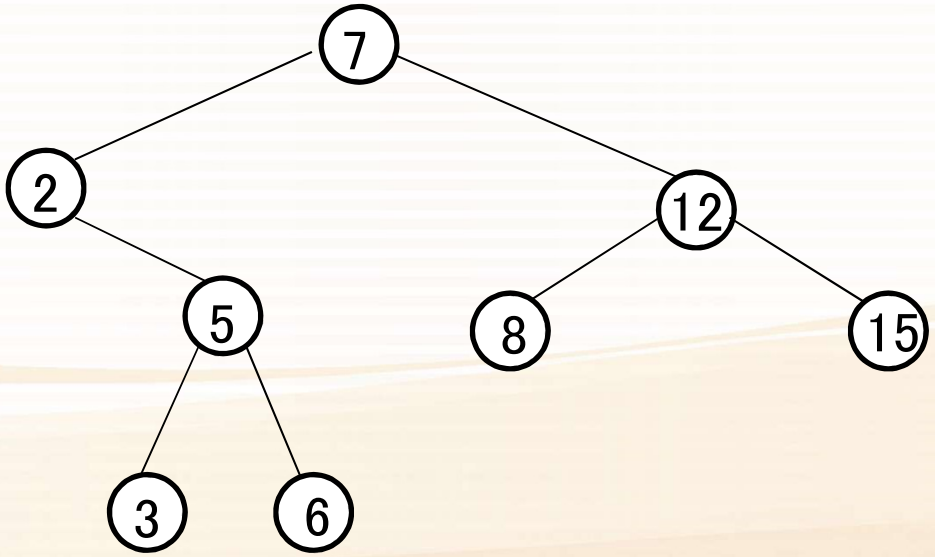


計算量は
木の高さ $O(\log N)$

二分探索木

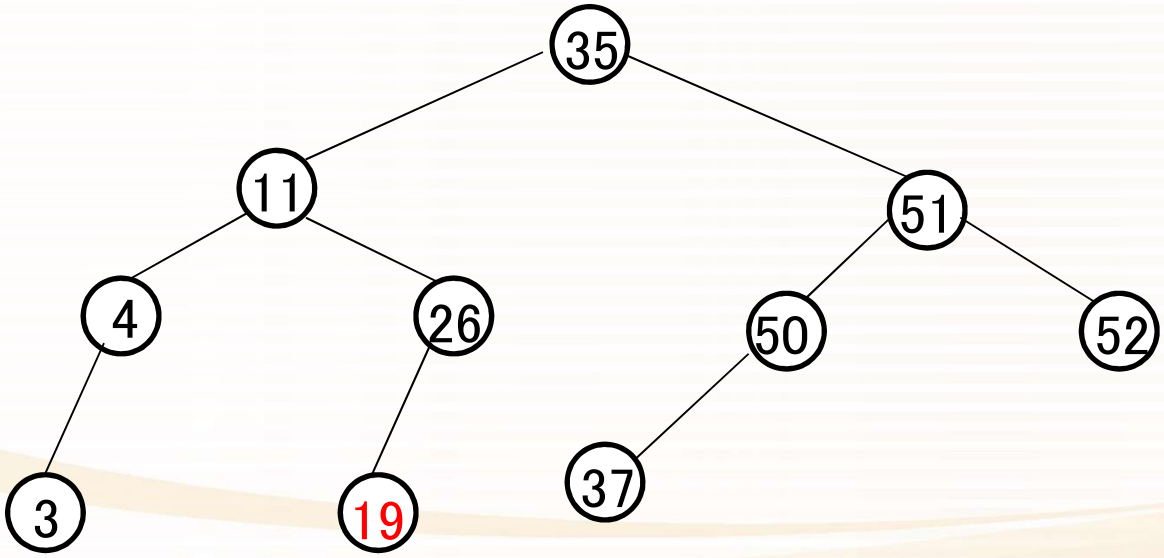
各頂点 v がキーとよばれる値 key[v] をもつ二分木で、
以下の条件をみたすものを二分探索木という。

任意の頂点 v に対し、
・v の左部分木に含まれるすべての頂点 v’ に対して
key[v] ≥ key[v’] が成立し、
・v の右部分木に含まれるすべての頂点 v’ に対して
key[v] ≤ key[v’] が成立する。



二分探索木のクエリ処理

クエリ
- 要素 x をデータ構造に挿入する
- 要素 x をデータ構造から削除する
- 要素 x がデータ構造に含まれるかどうかを判定する



二分探索木

3, 4, 11, 19, 26, 35, 37, 50, 51, 52

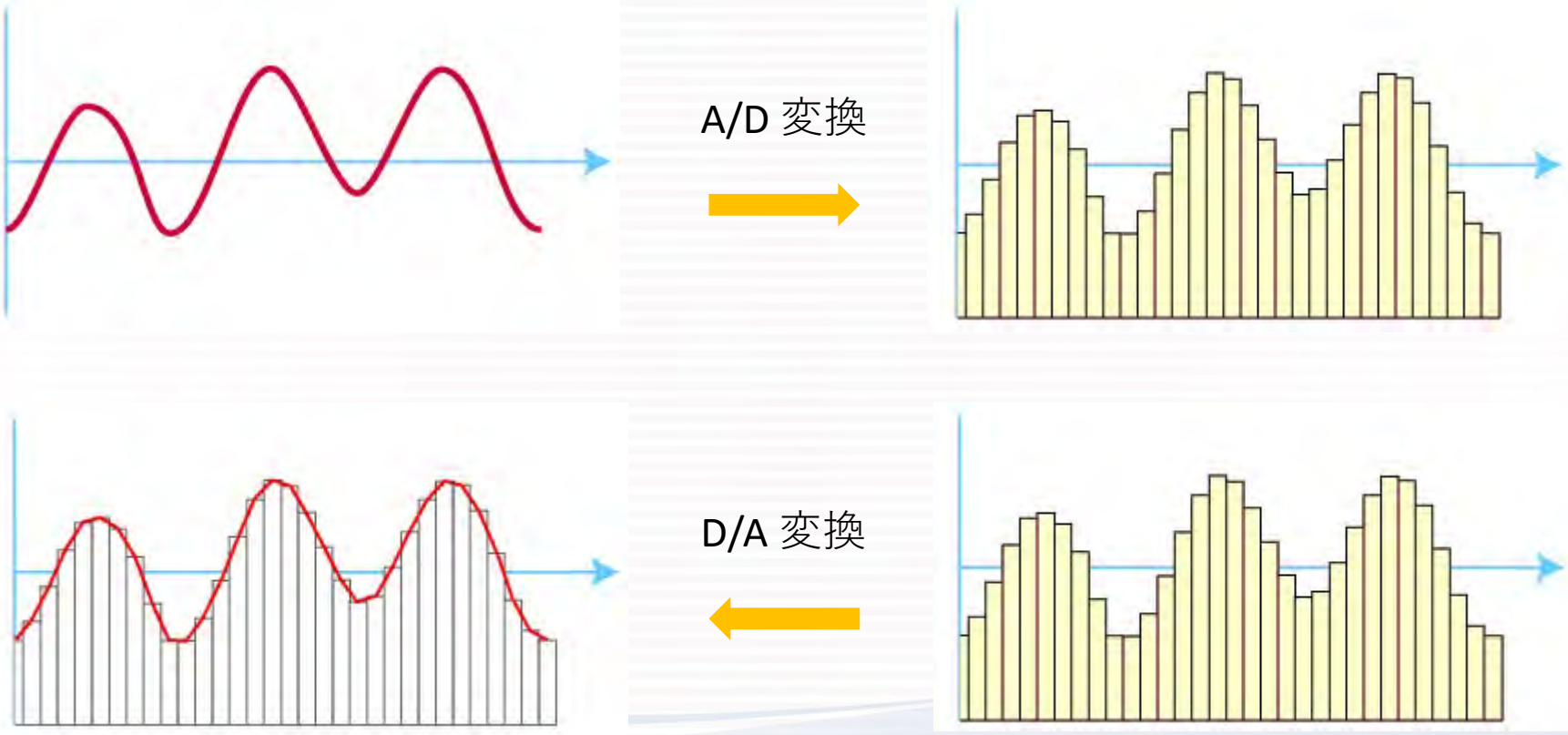
（ソート済み配列に対する）
二分探索アルゴリズム

データサイエンス応用コース
標本化・量子化（A/D,D/A変換）

下川　和郎
大阪大学

標本化・量子化（AD/DA 変換）

現実世界のアナログ情報と、コンピュータ等で扱いやすいデジタル情報を相互変換する

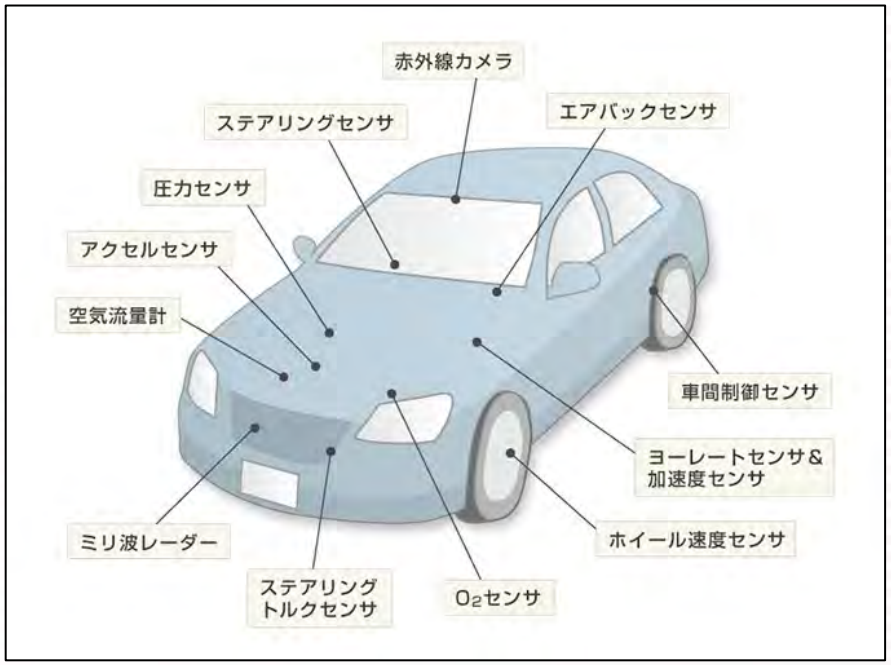


https://ja.wikipedia.org/wiki/デジタル-アナログ変換回路

AD/DA コンバーター

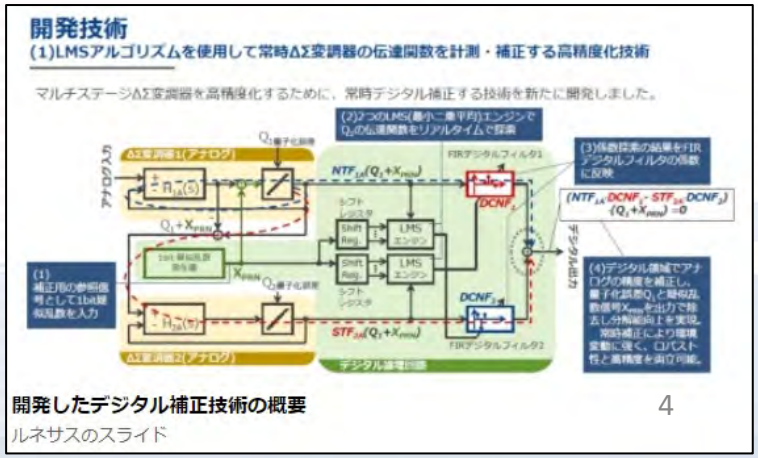
- 用途
  - 体温計、音楽プレイヤー、ビデオカード
  - 有線/無線通信
  - 自動車（自動運転技術）
    - 加速度・ジャイロセンサ、GPS、超音波センサ、ミリ波レーダ
  - 家（セキュリティー技術）

AD/DA コンバーター

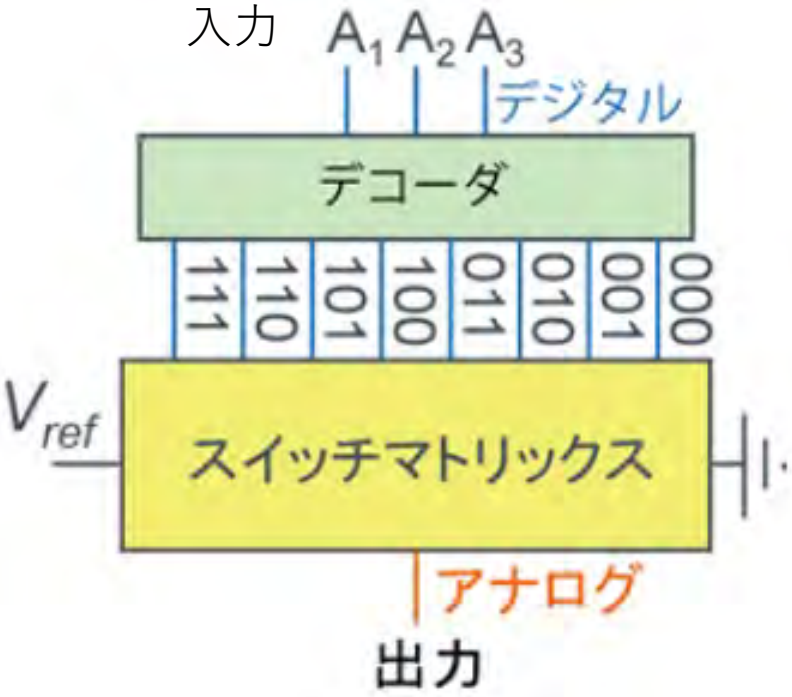


新エネルギー・産業技術総合開発機構　MEMSプロジェクト　HPより

MEMS （Micro Electro Mechanical Systems/微小電気機械システム）



D/A コンバーター

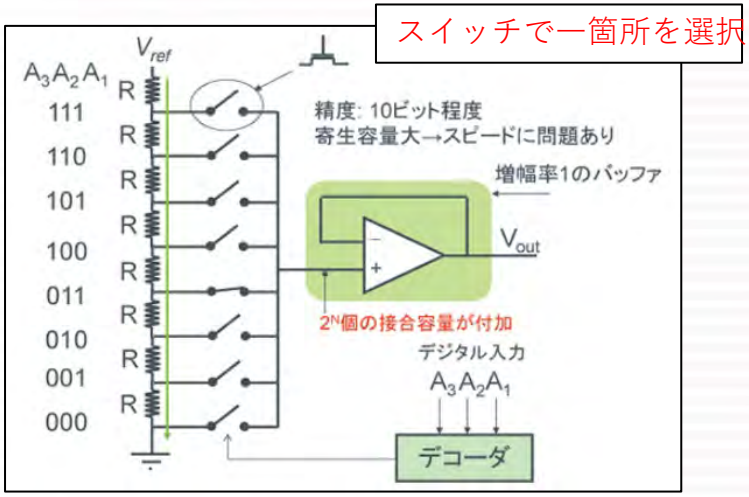


デジタル信号をアナログ信号に変換

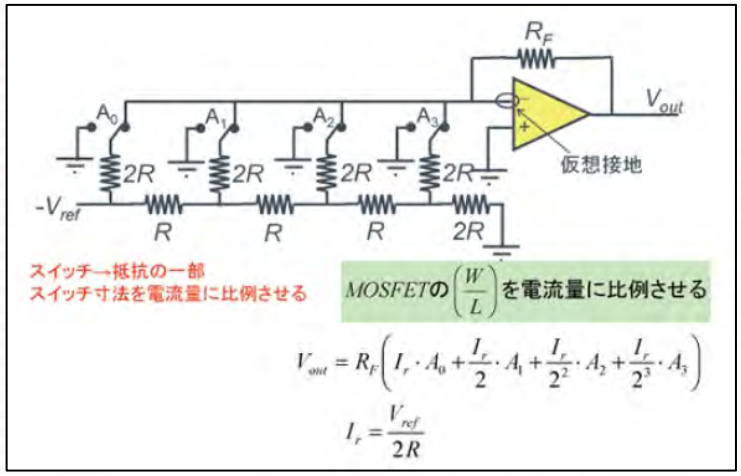
- デコーダー方式
  - 抵抗分圧方式
- バイナリー方式
  - 抵抗ラダー回路
  - キャパシター回路
- デルタシグマ（ΔΣ） 型

ROHM エレクトロニクス豆知識 HP より

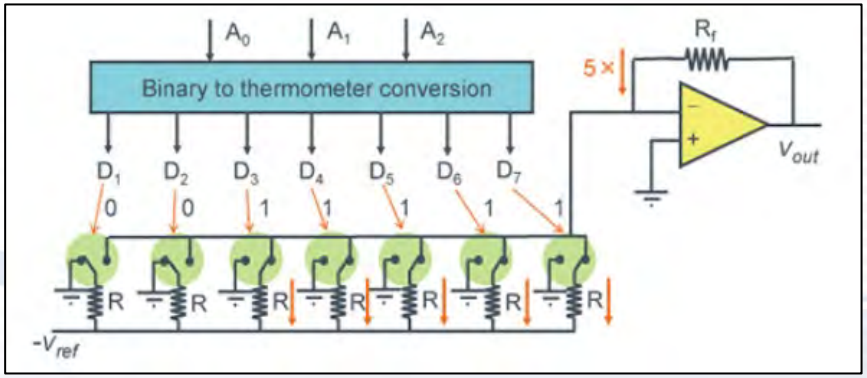
D/A コンバーター



抵抗分圧方式（N Bit を実現するのに 2^N 個のSWが必要）



R-2R ラダー方式（Rと2Rのペア性が重要）

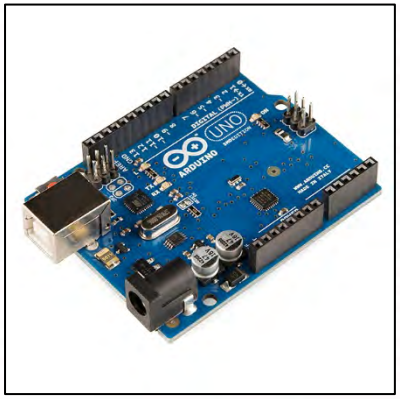


温度計コード方式（データの切り変わりで雑音が出ない）

線形性に優れ、原理上、単調増加性

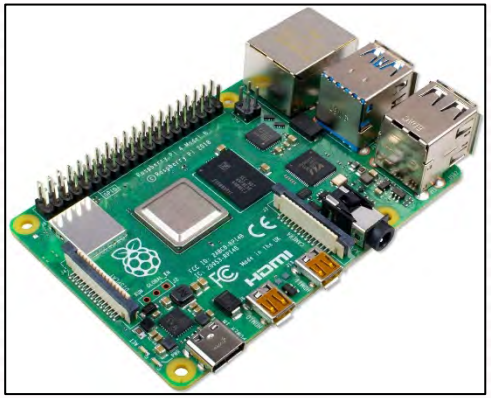
ROHM エレクトロニクス豆知識 HP より

簡単な実装系



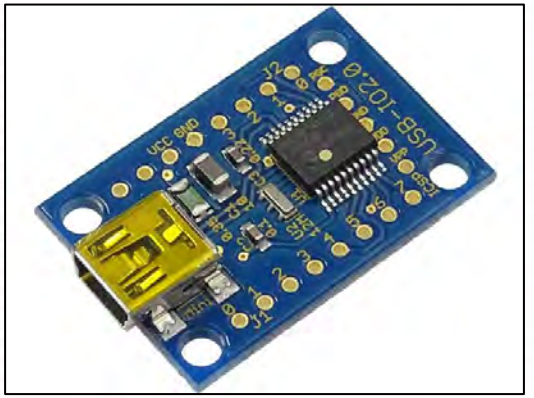
Arduino-Uno

https://www.arduino.cc/



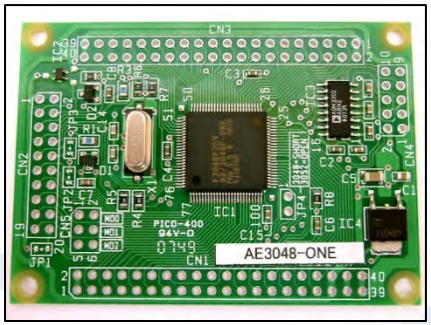
Raspberry-Pi

https://www.raspberrypi.org/



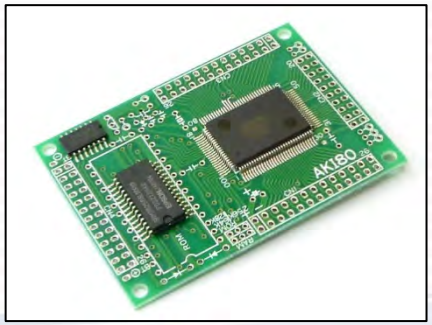
PIC Microcontrollers

https://www.microchip.com/



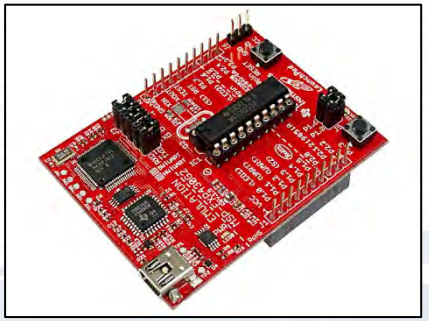
H8 Microcontrollers

https://www.renesas.com/jp/ja/



Z80 Microcontrollers

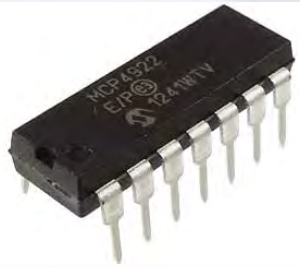
http://www.zilog.com



MSP430 Microcontrollers

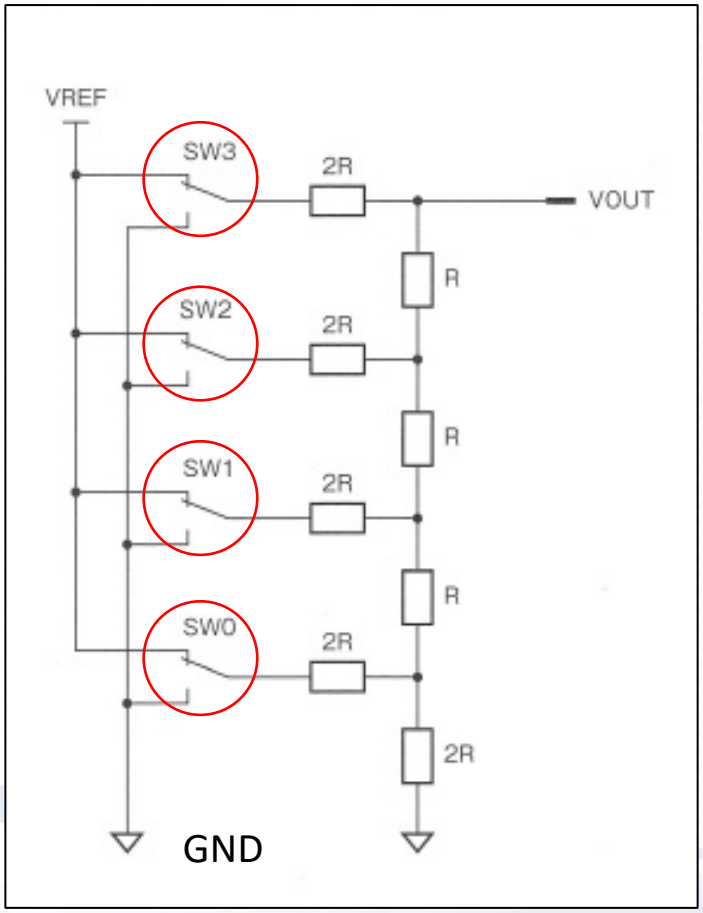
https://www.ti.com/

D/Aコンバーター MCP4922



Microchip 社

R-2R ラダー抵抗回路



電気回路の試作等でよく用いられる D/A コンバーターの内部動作を見る.

SW0 ～ SW3 がデジタル入力に相当

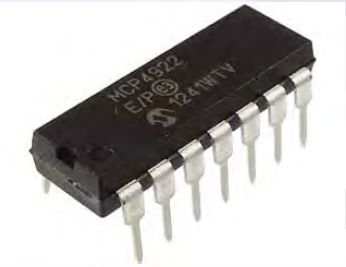
各スイッチが ON または OFF になることで

V_{ref} 又は GND に接続される.

これによって回路が変化し、合成抵抗が変わることによって V_{out} の電位が変わる.

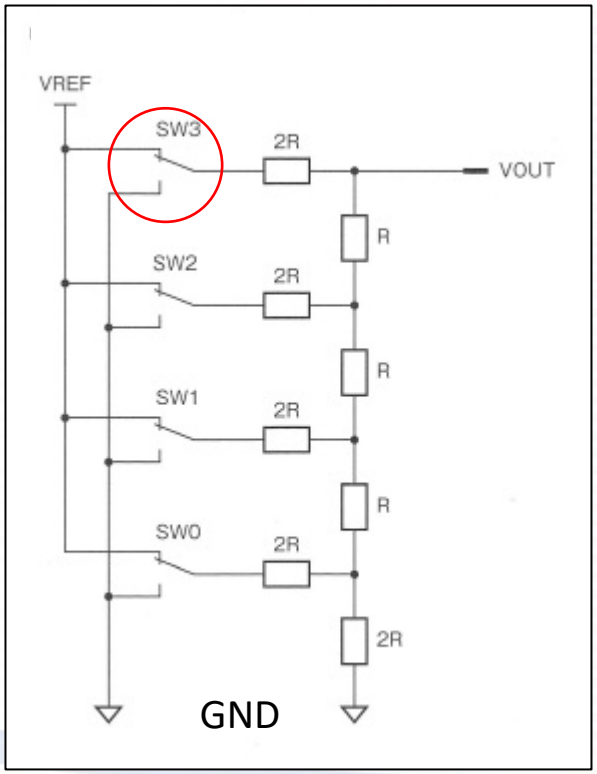
「Raspberry Pi 徹底入門」　菊池達也 著　技術評論社 より

D/Aコンバーター MCP4922

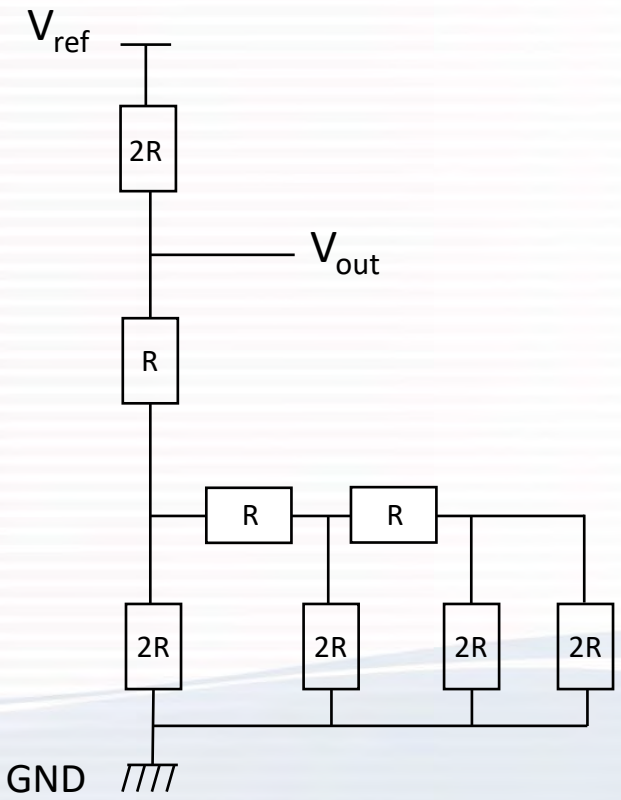


Microchip 社

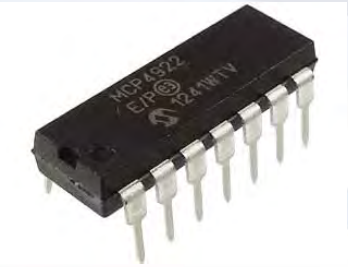
R-2R ラダー抵抗回路



SW3 のみ ON のときの等価回路



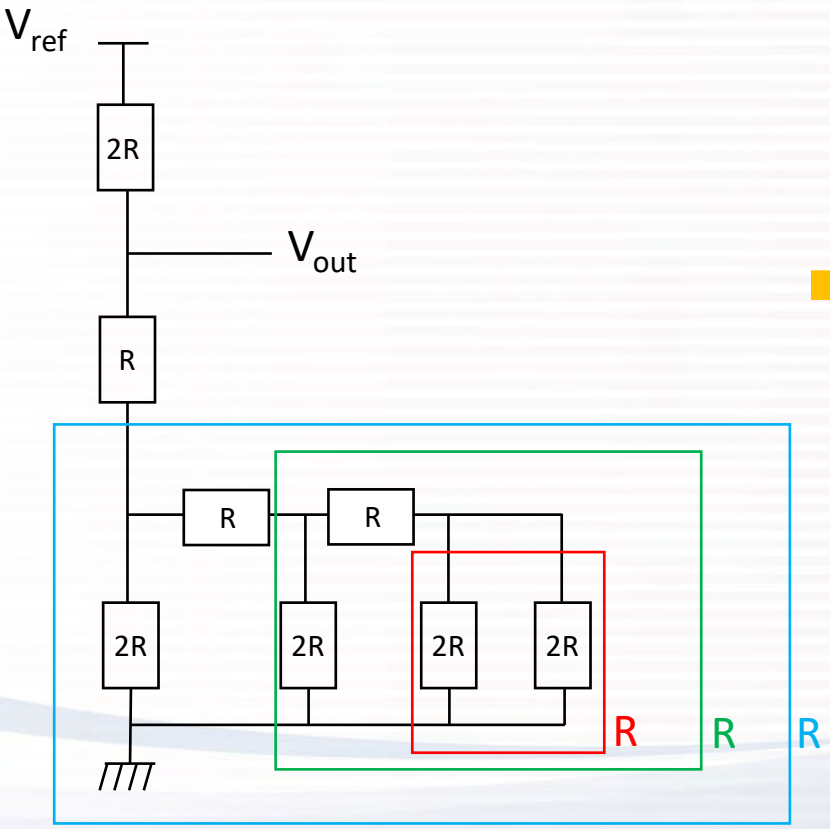
D/Aコンバータ MCP4922

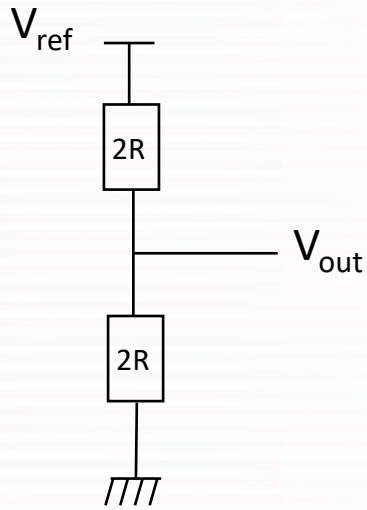


Microchip 社

SW3 のみ ON のときの等価回路 $= (1000)_2 = (8)_{10}$

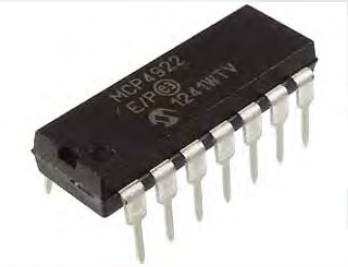
二進数　　十進数





$$V_{out} = \frac{1}{2} V_{ref}$$

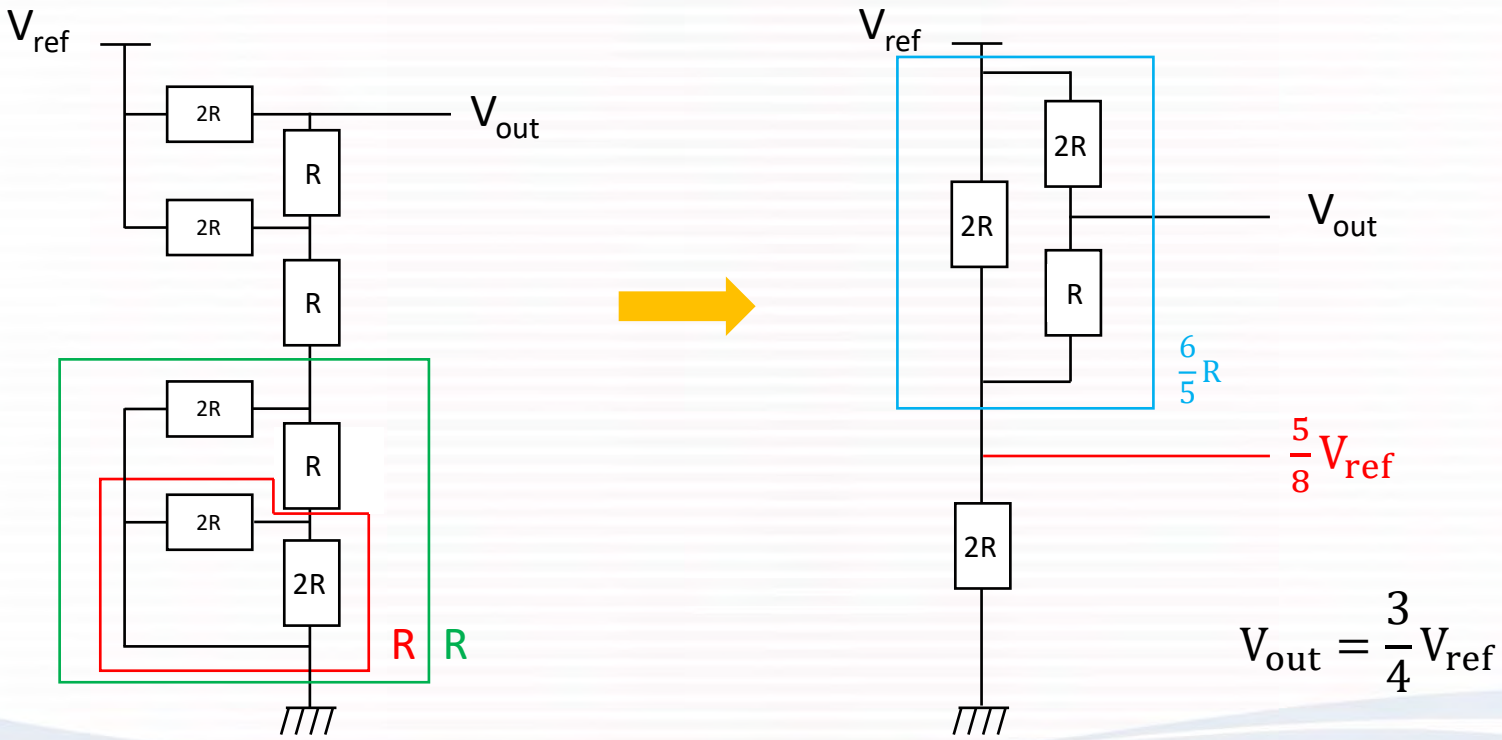
D/Aコンバータ MCP4922



Microchip 社

SW2, SW3 が ON のときの等価回路 = $(1100)_2 = (12)_{10}$

二進数　　十進数



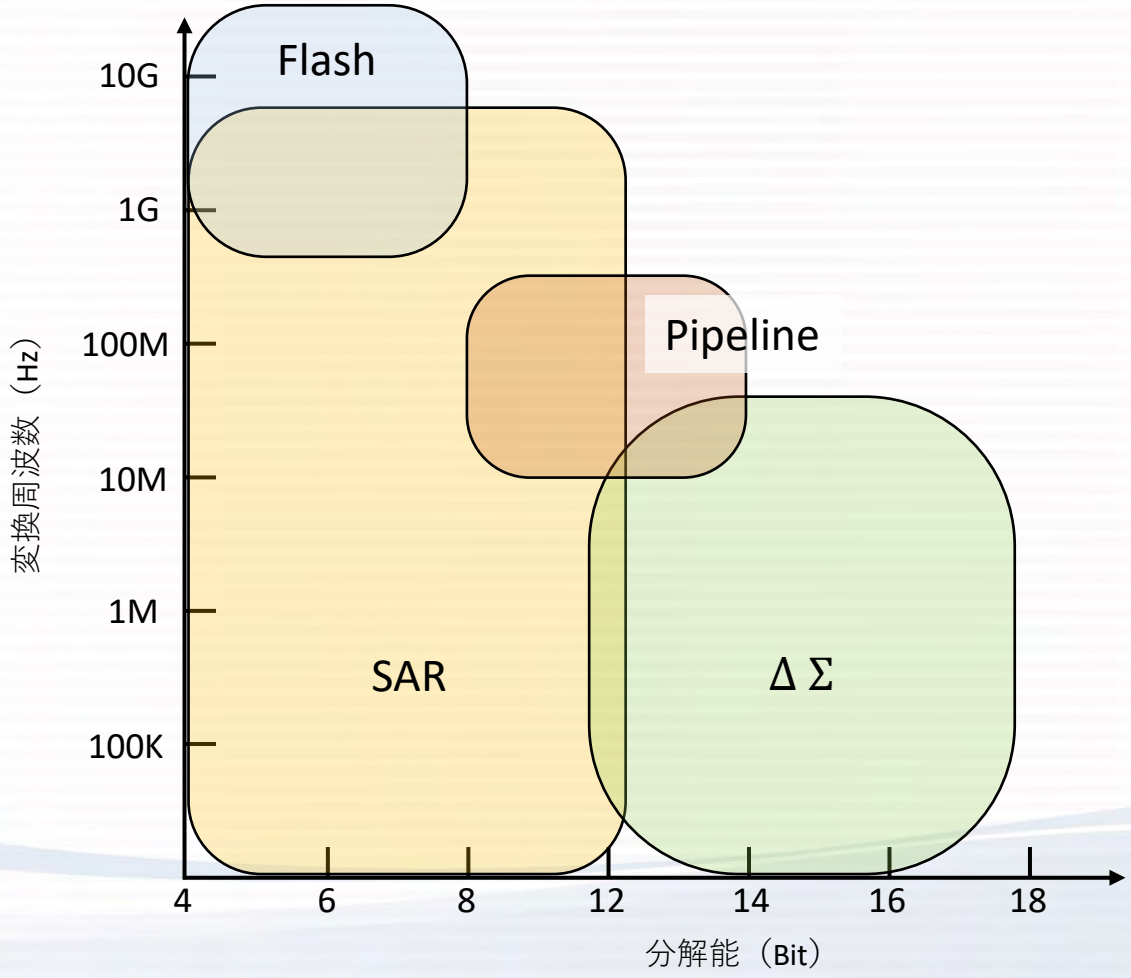
$$V_{out} = \frac{3}{4} V_{ref}$$

A/D コンバーター

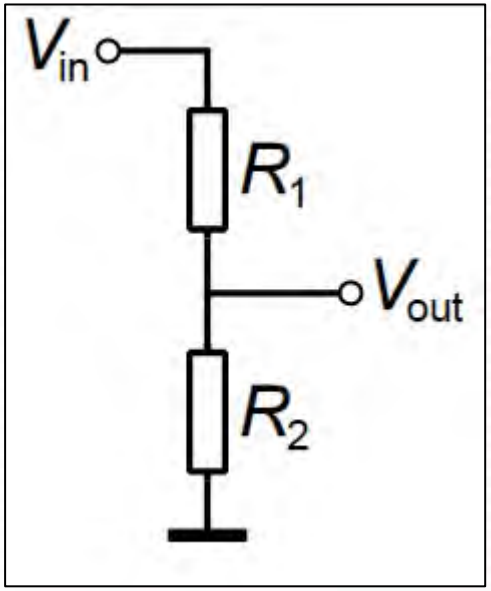
- 基本的な変換方式
  - フラッシュ型（並列型）
  - 逐次比較 (Successive Approximation) 型
    - （SAR型、D/A コンバーターと比較する）
  - パイプライン型
  - デルタシグマ（ΔΣ）型
    - 交流信号しか扱えない.

A/D コンバーター

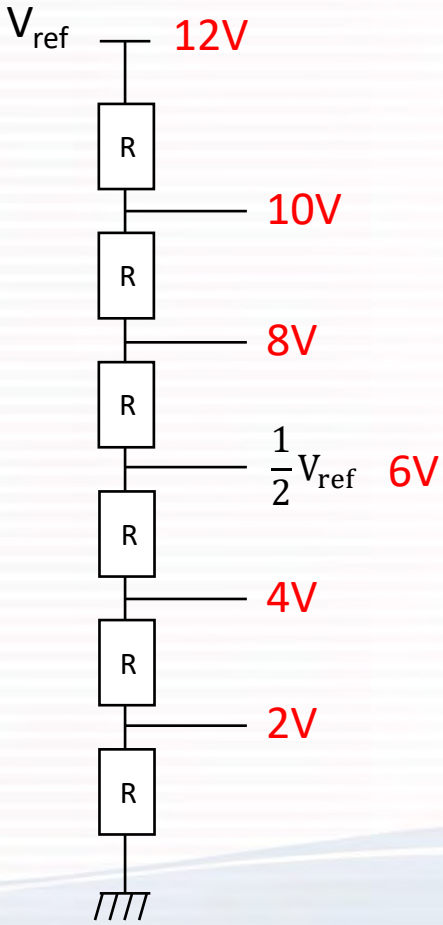
変換方式の違いと分解能・変換周波数



A/D コンバーター 回路図の補足：抵抗分圧



$$V_{\text{out}} = \frac{R_2}{R_1 + R_2} \cdot V_{\text{in}}$$

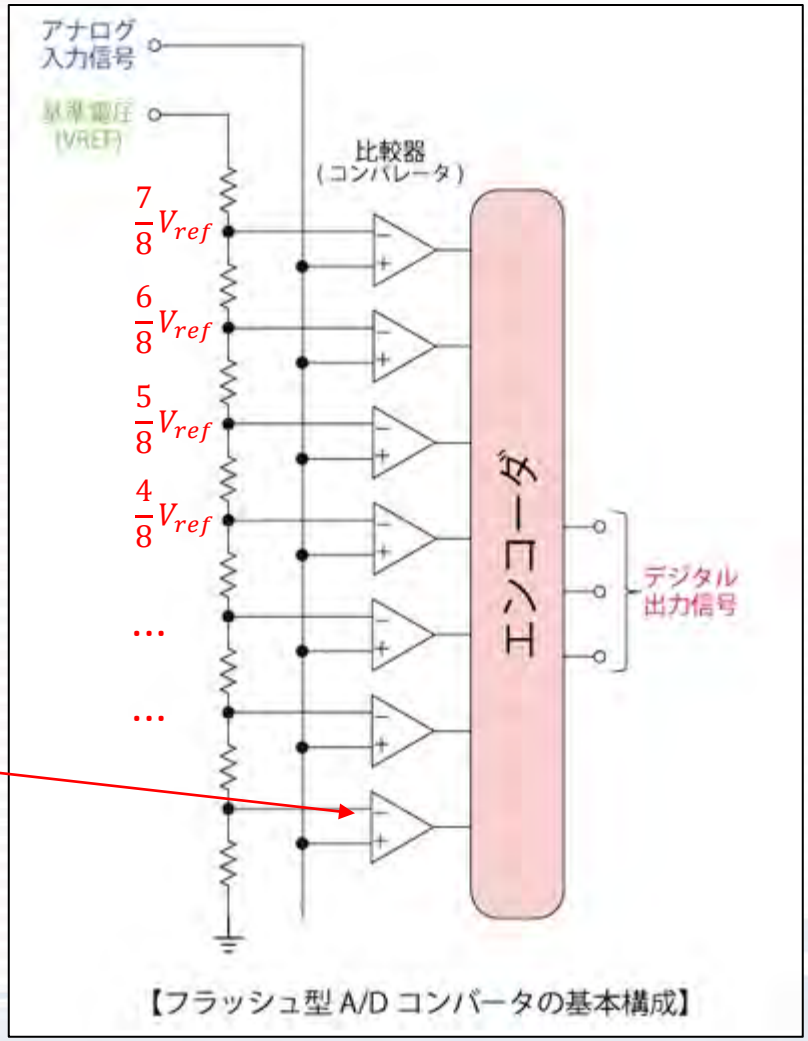


A/D コンバーター フラッシュ型（並列型）

基準電圧の異なる比較器と比べ、
それをデジタルエンコードする
ことによってデジタル出力を得る.

比較器の数
N bit なら 2^N-1 個必要

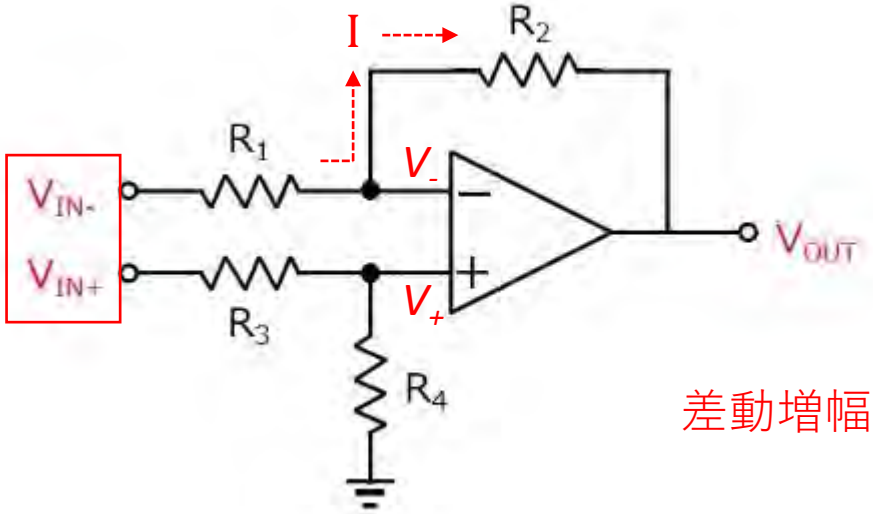
3 bit なら　7 個必要
8 bit なら 255 個必要



【フラッシュ型 A/D コンバータの基本構成】

ROHM エレクトロニクス豆知識 HP より

A/D コンバーター 回路図の補足：差動増幅器



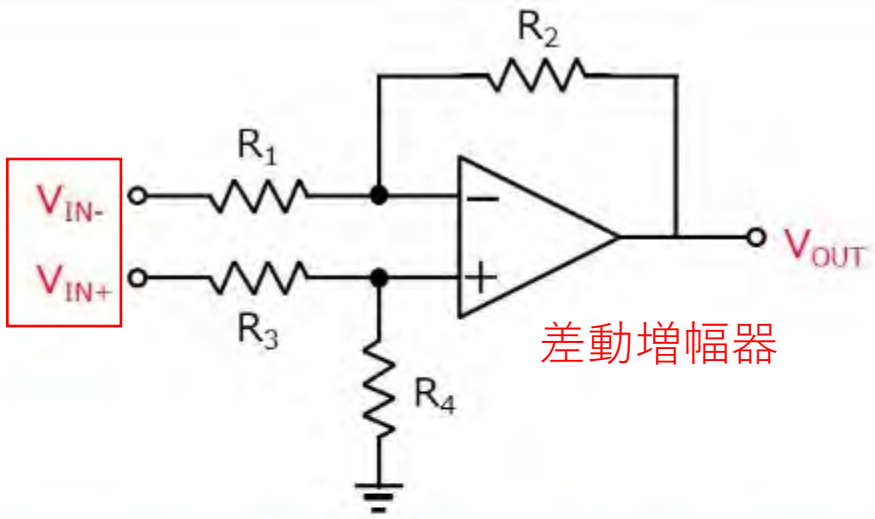
差動増幅器

抵抗の分圧回路以外には
この式くらいで計算可能

$$V = I \cdot R$$

$$\begin{cases} V_+ = \dfrac{R_4}{R_3 + R_4} V_{IN+} \\ I = \dfrac{V_{IN-} - V_-}{R_1} = \dfrac{V_- - V_{out}}{R_2} \end{cases}$$

A/D コンバーター 回路図の補足：差動増幅器



先の式から導かれる

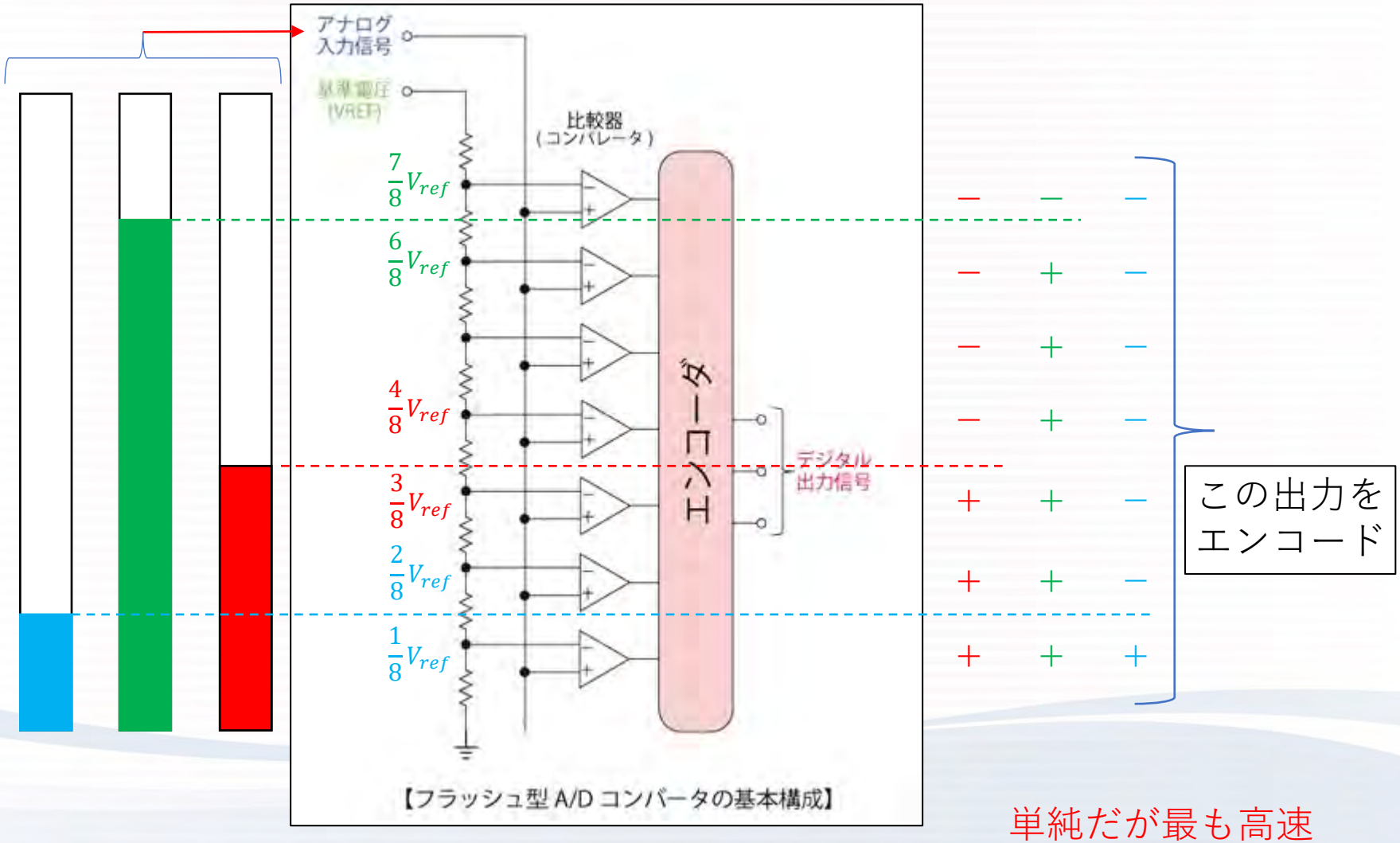
$$V_{OUT} = \left(\frac{R_1 + R_2}{R_1}\right)\left(\frac{R_4}{R_3 + R_4}\right)V_{IN+} - \frac{R_2}{R_1}V_{IN-}$$

$$R_1 = R_3, R_2 = R_4$$

$$V_{OUT} = \frac{R_2}{R_1}(V_{IN+} - V_{IN-})$$

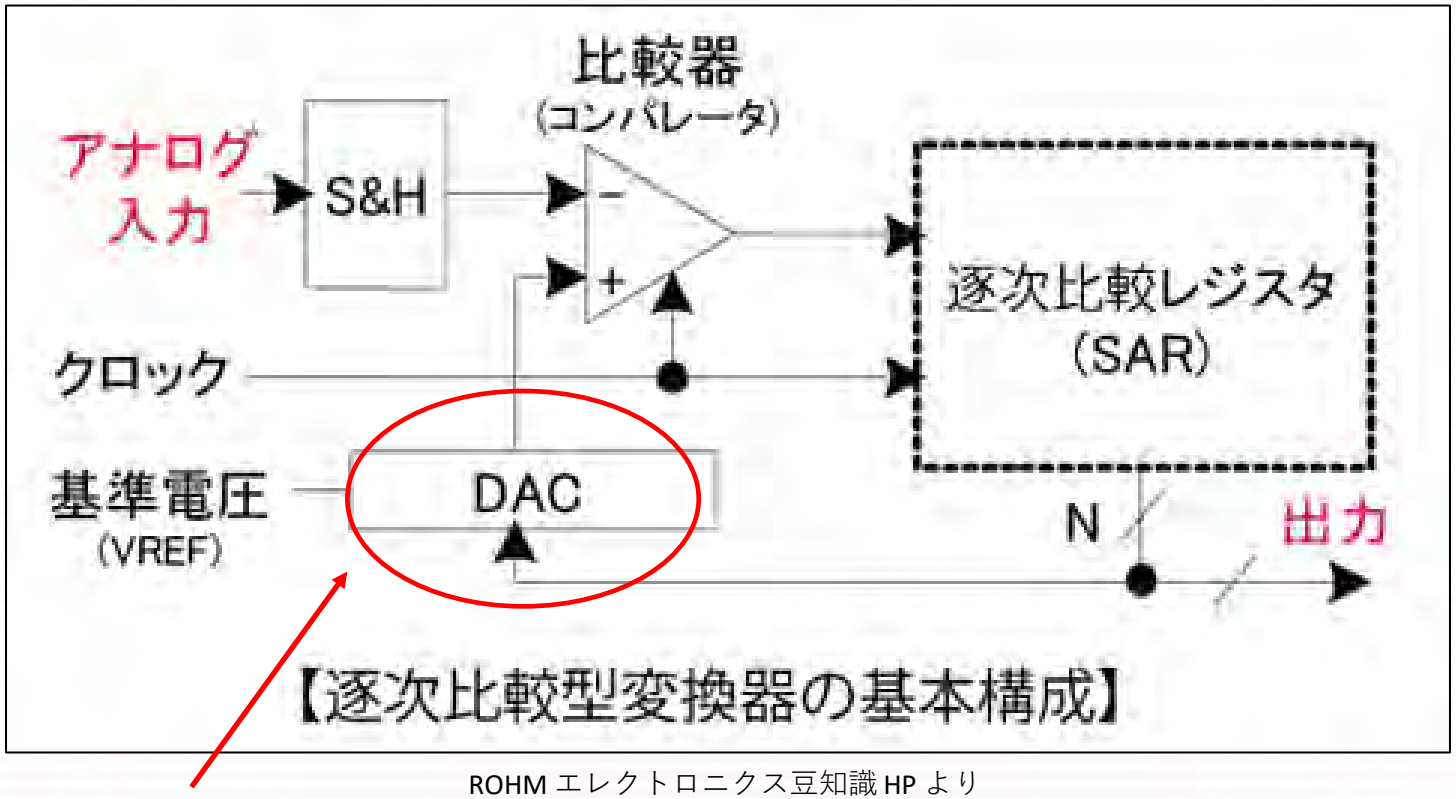
二つの入力の
引き算が出力に出る.

A/D コンバーター フラッシュ型（並列型）



単純だが最も高速

A/D コンバーター 逐次比較（SAR）型

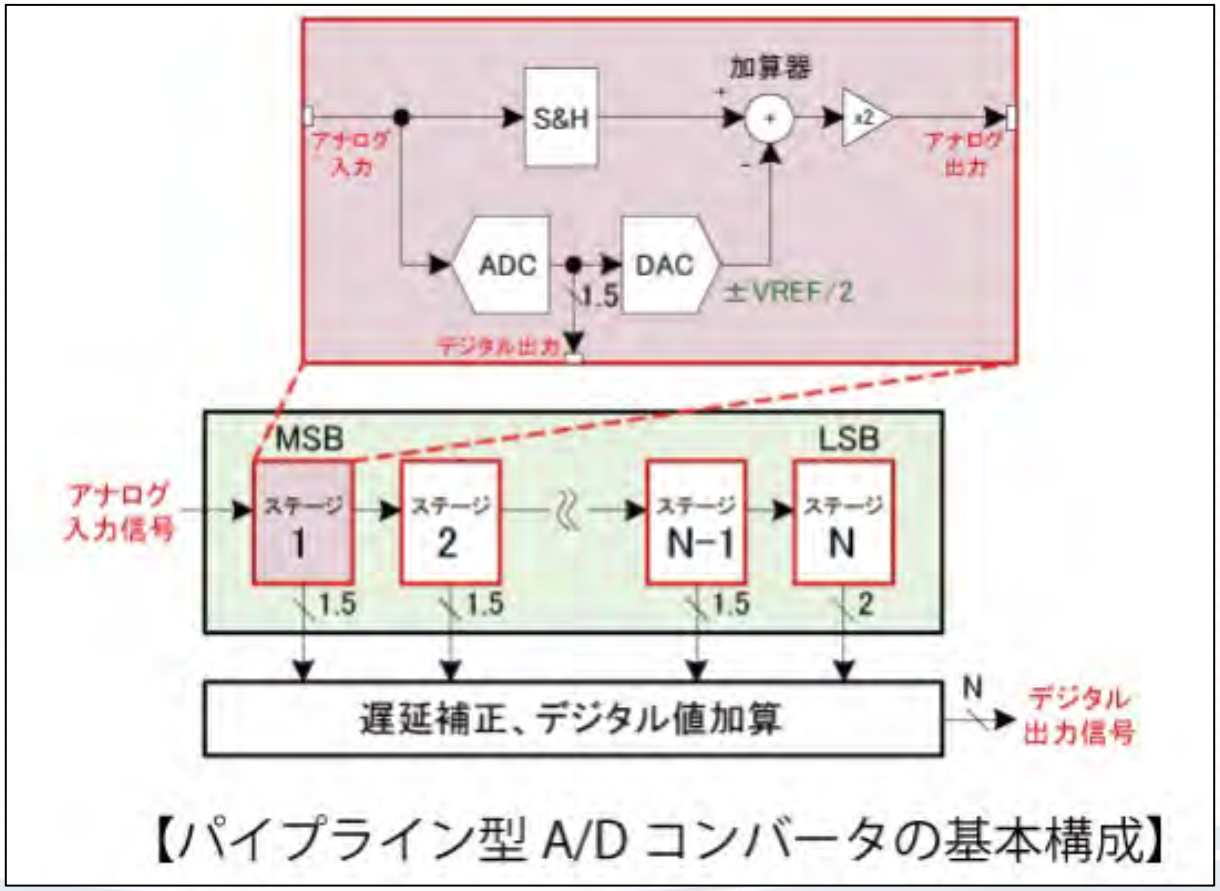


ROHM エレクトロニクス豆知識 HP より

DAC と比較することによってデジタル出力を得る.

S&H, SHA ： サンプル＆ホールド

A/D コンバーター パイプライン型

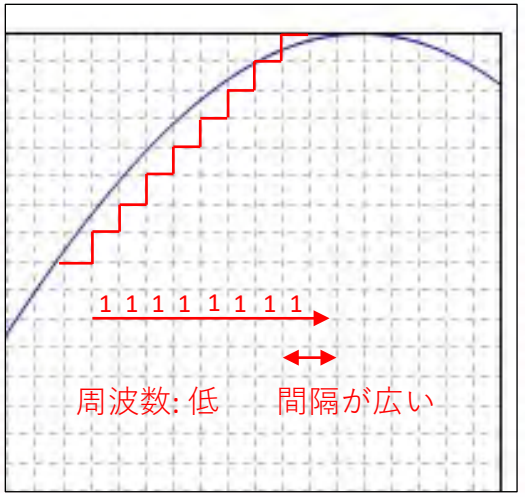


ROHM エレクトロニクス豆知識 HP より

A/D コンバーター　デルタシグマ（ΔΣ）型

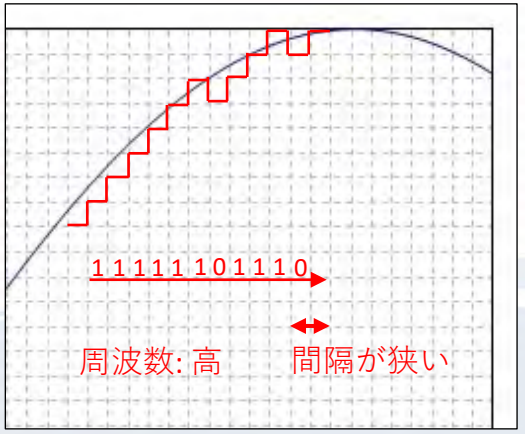
最も高分解能

- 入力信号が一つ前のΔ時間の出力電圧より高いか低いかを1 Bitで補正（± 1）して出力する.

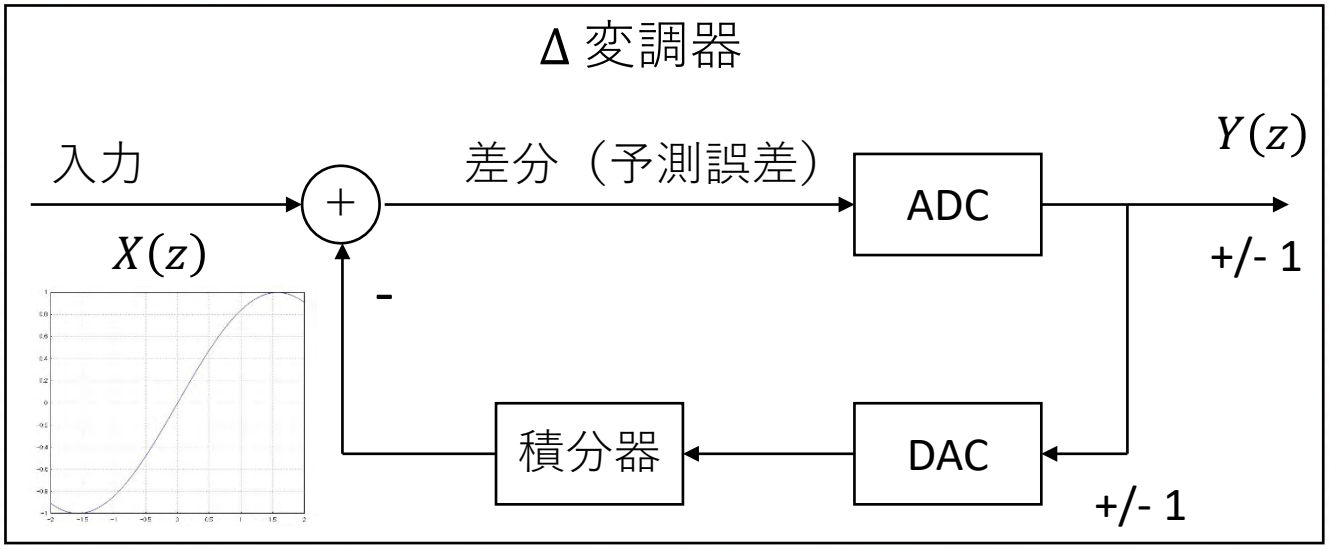


サンプリング周波数が低いと、
入力電圧の変化に追従できない.

- 入力信号に対して十分高い周波数でサンプリングする必要がある.遅い場合には右図のように入力信号に出力が追いつかない場合が出る.



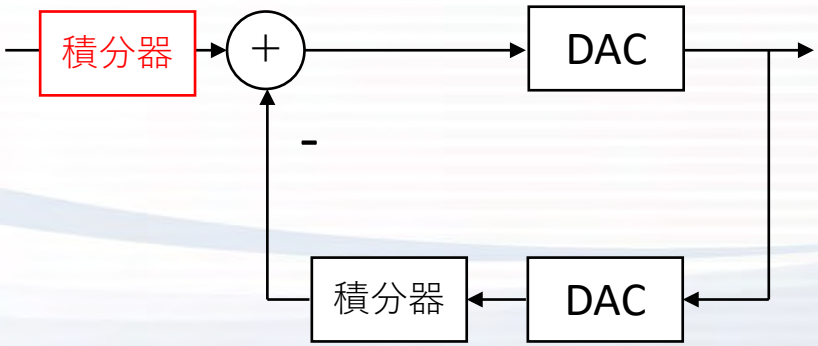
A/D コンバーター　デルタシグマ（ΔΣ）型



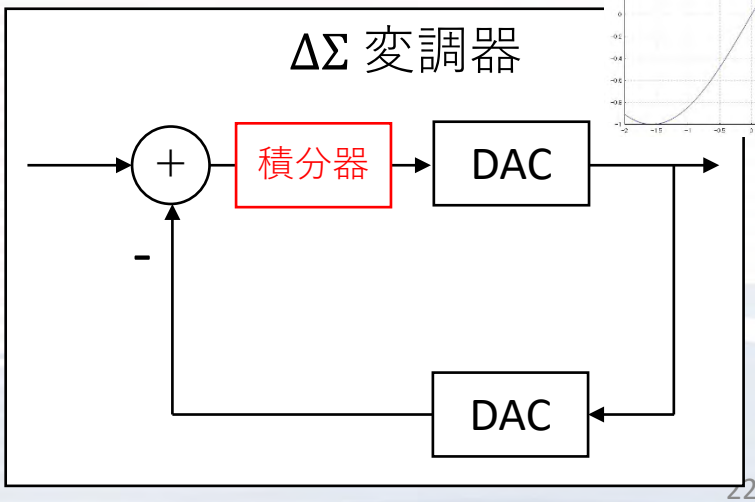
111010011

このままでは出力が
01 の符号になる.

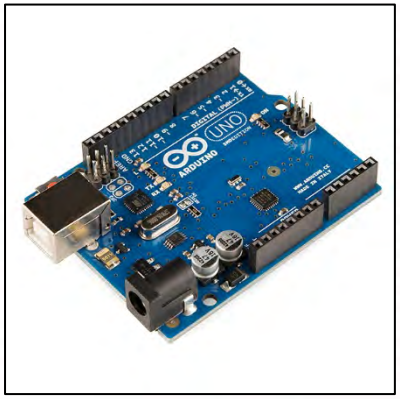
入力信号を積分することにより最終出力が
01 ではなく本来信号になるように変形する



等価
回路

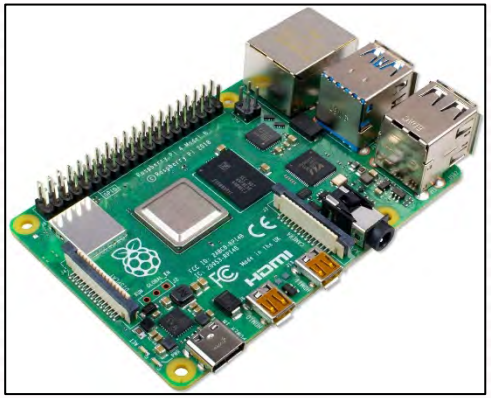


簡単な実装系



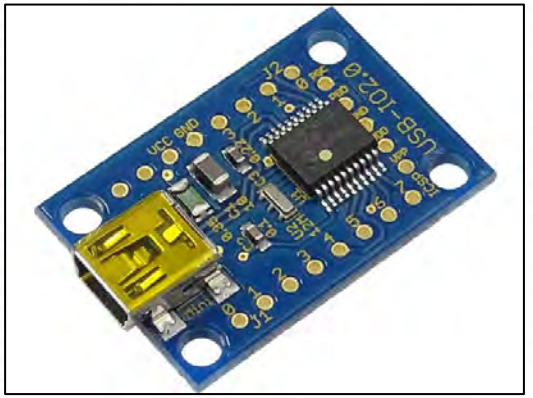
Arduino-Uno

https://www.arduino.cc/



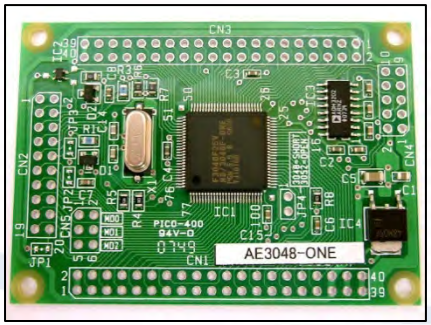
Raspberry-Pi

https://www.raspberrypi.org/



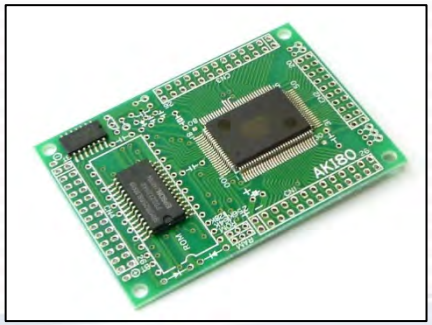
PIC Microcontrollers

https://www.microchip.com/



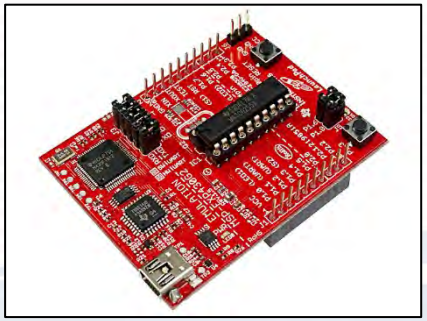
H8 Microcontrollers

https://www.renesas.com/jp/ja/



Z80 Microcontrollers

http://www.zilog.com

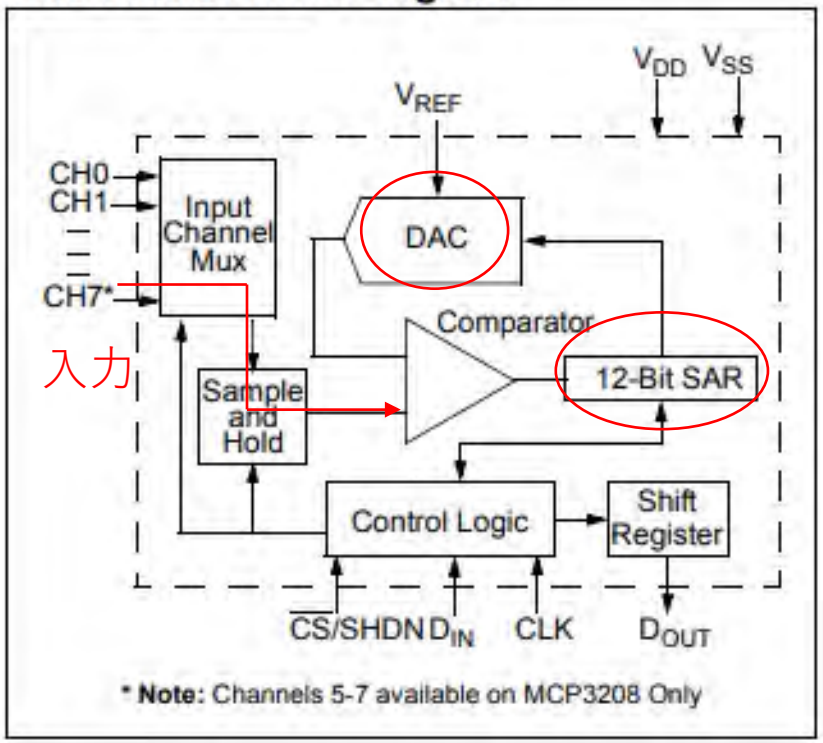


MSP430 Microcontrollers

https://www.ti.com/

A/Dコンバーター MCP3208

Functional Block Diagram



入力

電気回路の試作等でよく用いられる
A/D コンバーターの内部動作を見る.

このコンバーターの入力は

CH0 ～ CH7 の 8 チャンネル

入力信号と DAC との比較を行うことに
よってデーコードを行っている.

→ 逐次比較型の A/D コンバーター

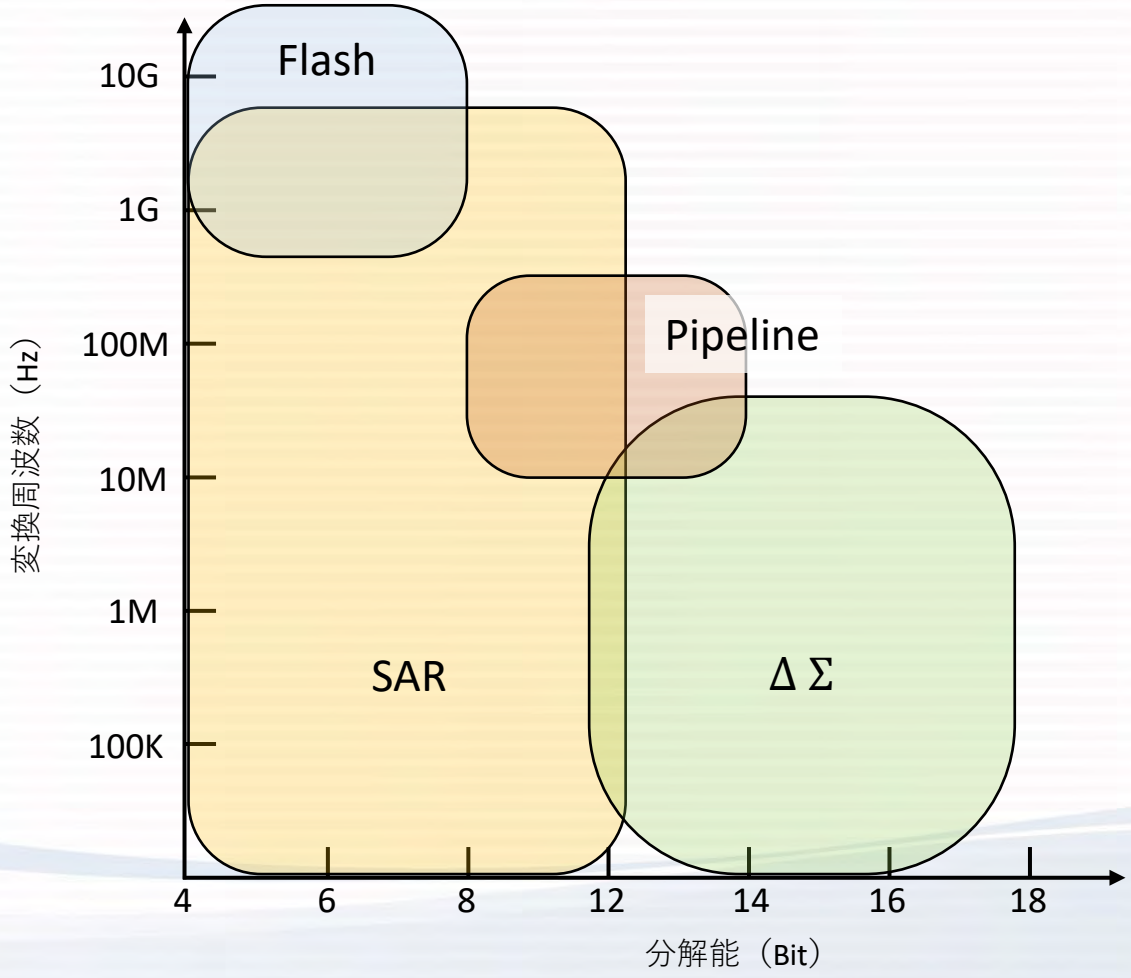
MICROCHIP 社 MCP3208 Data Sheet より

性能評価 A/D,D/A コンバーター

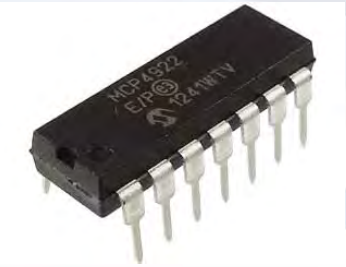
- 分解能
- サンプリングレート
- チャンネル数
- 直線性誤差
- 微分直線性誤差（DNL）
- 積分非直線性誤差（INL）
- Spurious Free Dynamic Range（SFDR）

A/D コンバーター

変換方式の違いと分解能・変換周波数



微分非線形誤差 (DNL)



1Bit 当たりの理想的な電圧幅と実際の電圧幅との誤差

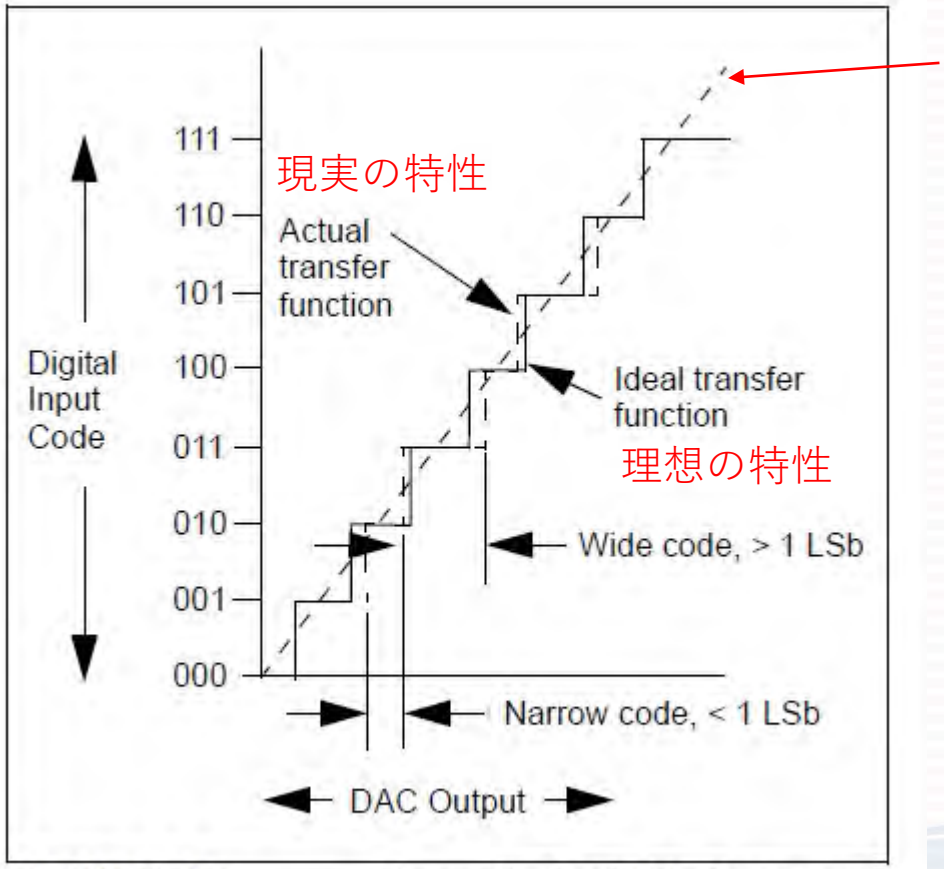
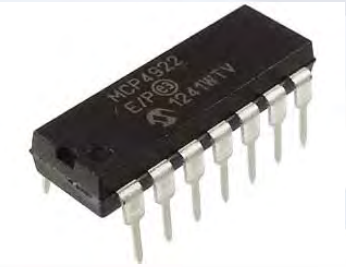


FIGURE 4-2: *Example for DNL Accuracy.*

MICROCHIP 社 MCP4902 Data Sheet より

積分非線形誤差 (INL)



オフセット誤差とゲイン誤差を取り除いた後の、理想的な出力との誤差の最大値

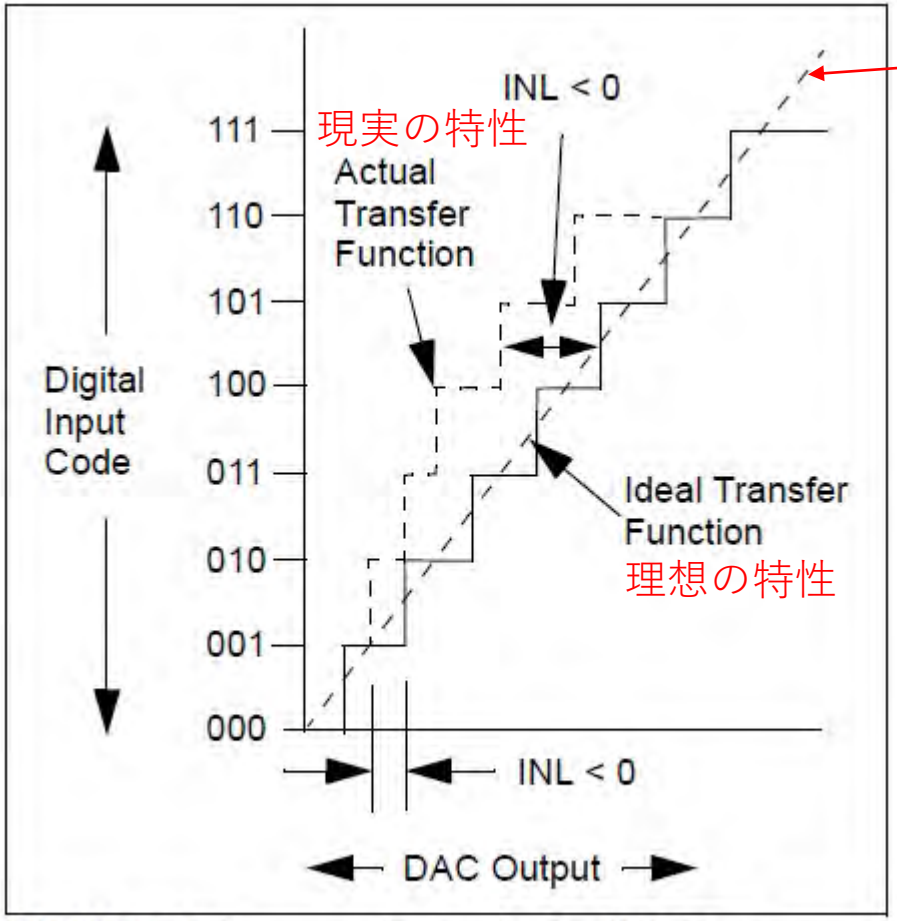


FIGURE 4-1: *Example for INL Error.*

MICROCHIP 社 MCP4902 Data Sheet より

Spurious　（スプリアス）

- 「偽の」、「見せかけの」の意
- 高調波、低調波、寄生振動などによって発生する目的外の電波
- 電子回路の内部で作られたノイズ等.

データサイエンス応用コース
形式言語，形式手法

下川　和郎
大阪大学

形式言語

- 1950年代、言語学者チョムスキー（Avram Noam Chomsky）によって、言語を生成するシステムとしての形式文法が定式化された.
- 形式文法として定義されるのは句構造文法、文脈依存文法、文脈自由文法、正規文法の 4 種類. これら 4 種の言語の生成能力の違いを研究するのが形式言語理論.
- 形式文法によって生成される言語は形式言語と呼ばれる. すなわち形式文法は生成機械、生成システムである.

オートマトン（自動機械）

- コンピューターの数学的なモデルとして研究された.
  - プッシュダウン・オートマトン
    ＝有限オートマトン＋無限容量のスタック.
  - プッシュダウン・オートマトンはプログラミング言語のコンパイラの基礎となっている.
- オートマトン理論
  - オートマトンの能力の違い（何ができるか、できないか）を研究する学問
- 形式言語で記述された文を識別するための仮想機械として利用.

形式言語理論とオートマトン理論

（1950年代）　　（1930年代）

正規文法 ＝ 有限オートマトン

文脈自由文法 ＝ プッシュダウンオートマトン

文脈依存文法 ＝ 線形拘束オートマトン

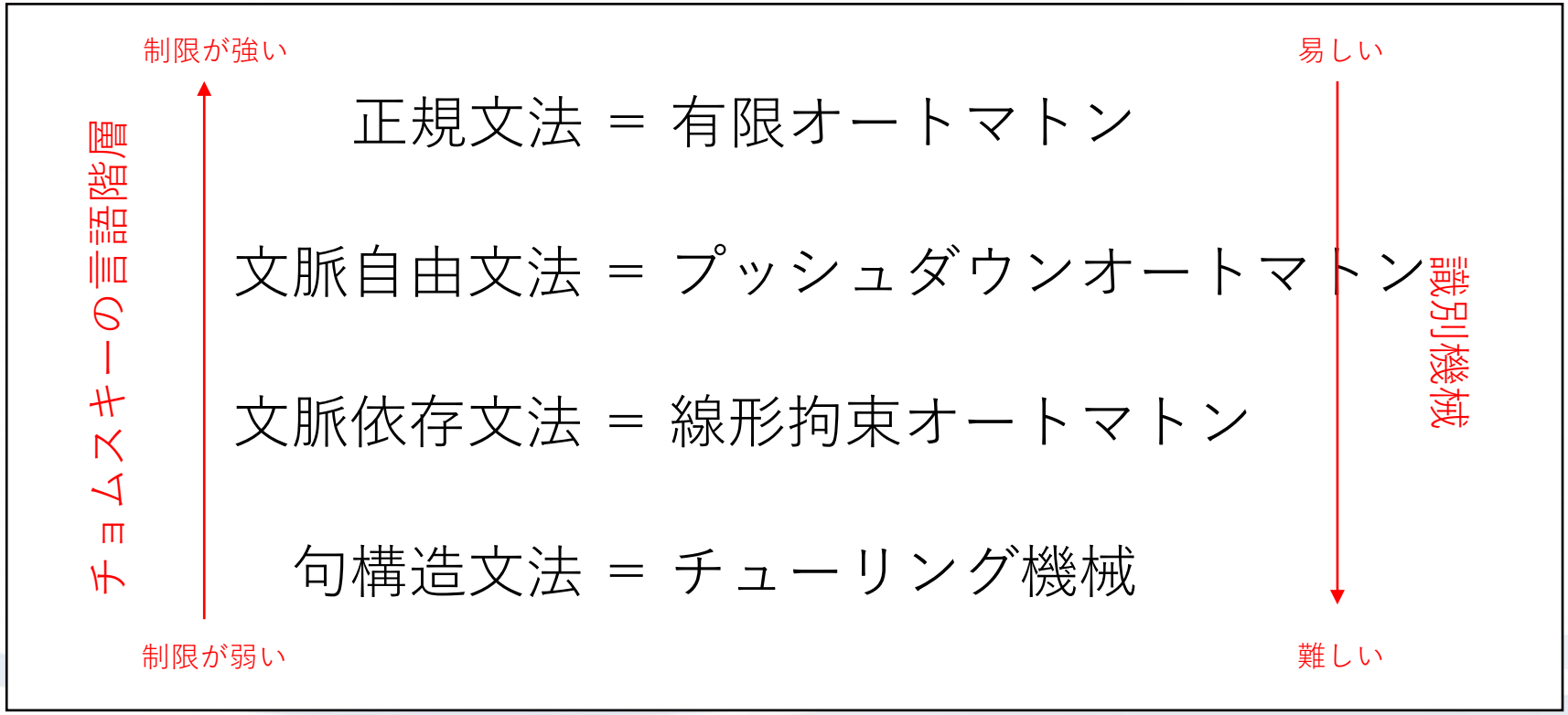
句構造文法 ＝ チューリング機械

＝ コンパイラの基礎

＝ 現在のコンピューターと同じ

形式言語理論とオートマトン理論

（1950年代）　　（1930年代）



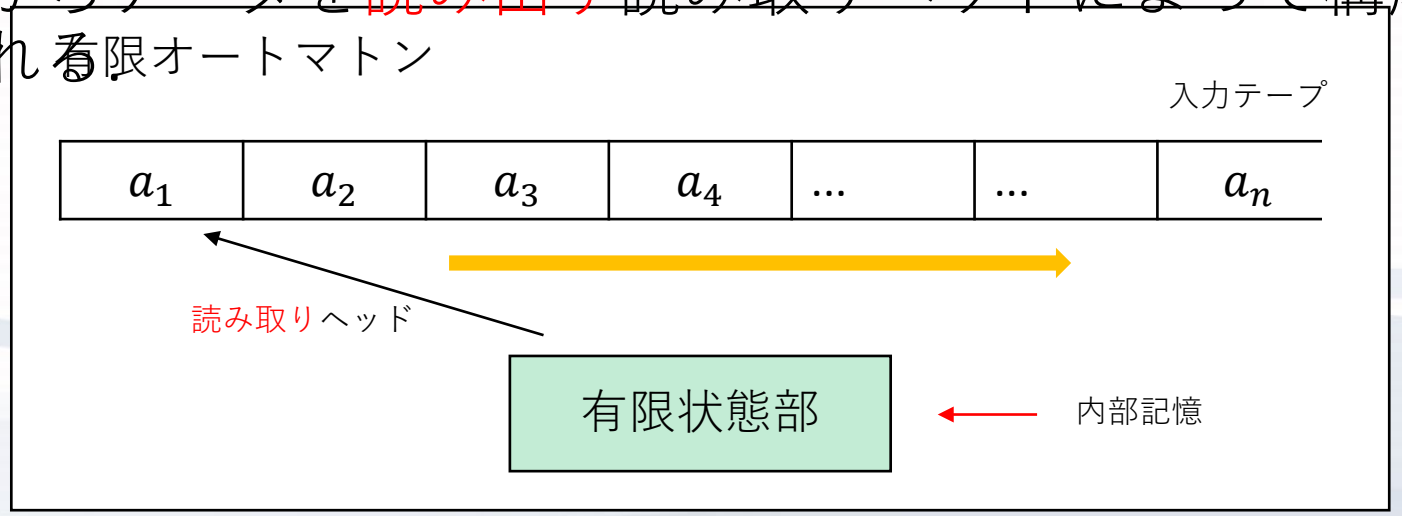
形式文法により定義される言語は…

- 言語が文法上正しく使われているかを識別する. 文章の意味内容については問題にしていない.
- 形式文法を用いて自然言語を完全に定義することはできす、自然言語に対する完全なモデルには至らなかった.
- コンピュータ上で使われる数式やプログラミング言語を容易に定義することができ、情報科学の分野では広く用いられる概念となった.

有限オートマトンの概要

- 有限オートマトンは状態数が有限な認識機械
- 一連の入力列に対してその入力に応じて状態を変化させ

  入力終了時の状態に応じて「Yes」or「No」を出力する.
- 有限オートマトンは状態制御部、入力用テープとテープからデータを読み出す読み取りヘッドによって構成される.



有限オートマトンの定義

- 有限オートマトンMは、三つの集合 Q, Σ, F と、特別な状態
  $q_0 \in Q$および関数を指定することにより定まる計算のモデルであり、以下のように表される.

$$M = (Q, \Sigma, \delta, q_0, F)$$

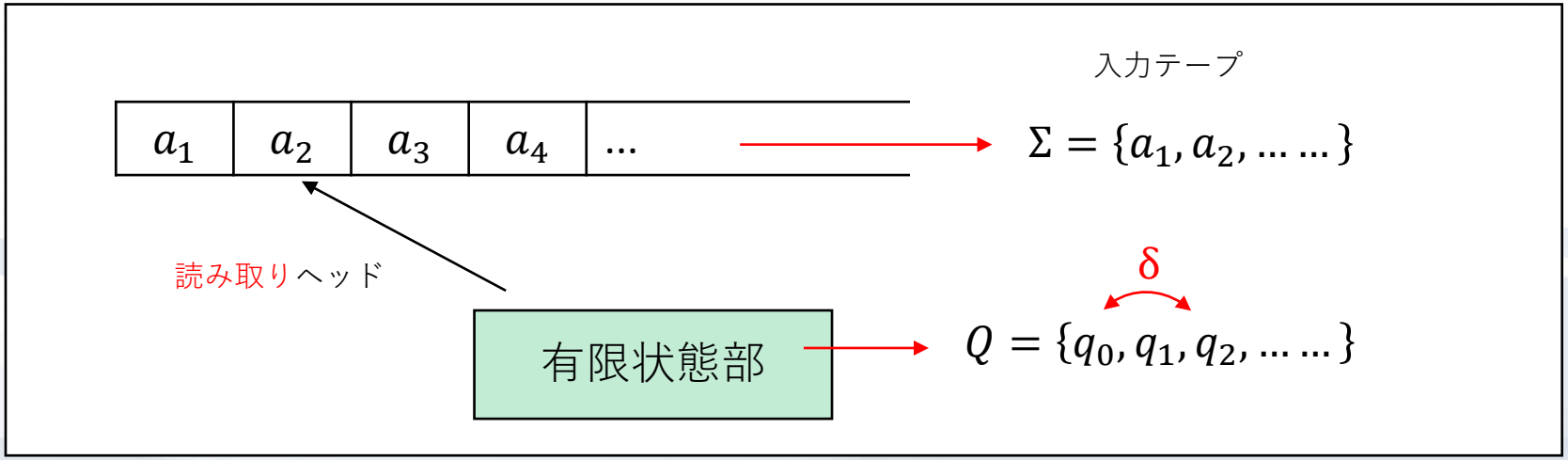
- また各要素は以下の通り
  - Q：状態の集合
  - Σ：入力記号の集合
  - δ：状態遷移関数
  - q_0：初期状態
  - F：受理状態の集合

有限オートマトンの定義

- 有限オートマトンの状態遷移関数は、状態 $q_0 \in Q$ と入力
  $a \in \Sigma$ に対して次の状態 $q \in Q$ を定める関数であり、
  以下のように表される.

前ステップのqの値

$$\delta(p, a) = q$$



二進数を認識する有限オートマトン

- 二進数

- 例1. 1101
- 例2. 1000100

- 二進数ではない記号列（0 から始まることはない）

- 例3. 0010110
- 例4. 0110111

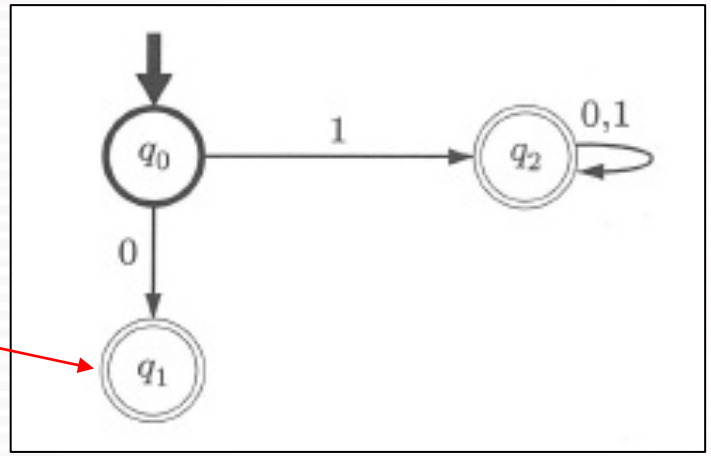
-

「はじめて学ぶオートマトンと言語理論」　藤原暁宏 著　　森北出版社 p.21

二進数を認識する有限オートマトン

- $M = (Q, \Sigma, \delta, q_0, F)$
- $Q = \{q_0, q_1, q_2\}$
- $\Sigma = \{0,1\}$
- $F = \{q_1, q_2\}$
- $\delta(q_0, 0) = q_1$
- $\delta(q_0, 1) = q_2$
- $\delta(q_2, 0) = q_2$
- $\delta(q_2, 1) = q_2$

二進数ではない!
終了



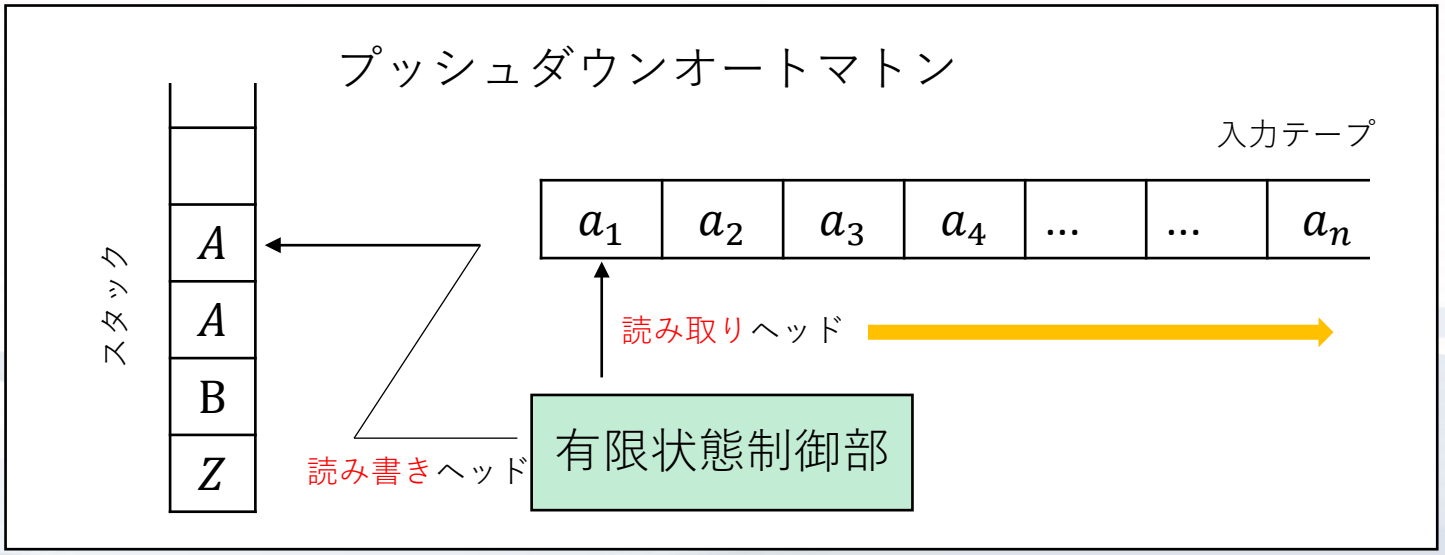
- これだと意味がわかりにくいので

最初に入力として 0 が入力されるとそれ以上の入力を受理しない.
最初に入力として 1 が入力されるとに遷移し、それ以上は 0,1 のどの記号でも入力を受ける.

プッシュダウンオートマトンの概要

- 有限オートマトンにプッシュダウンテープを取り付けた機械

- プッシュダウンテープ：
  - 半無限長のテープで、スタックメモリ（先入れ後出し）として働く



プッシュダウンオートマトンの定義

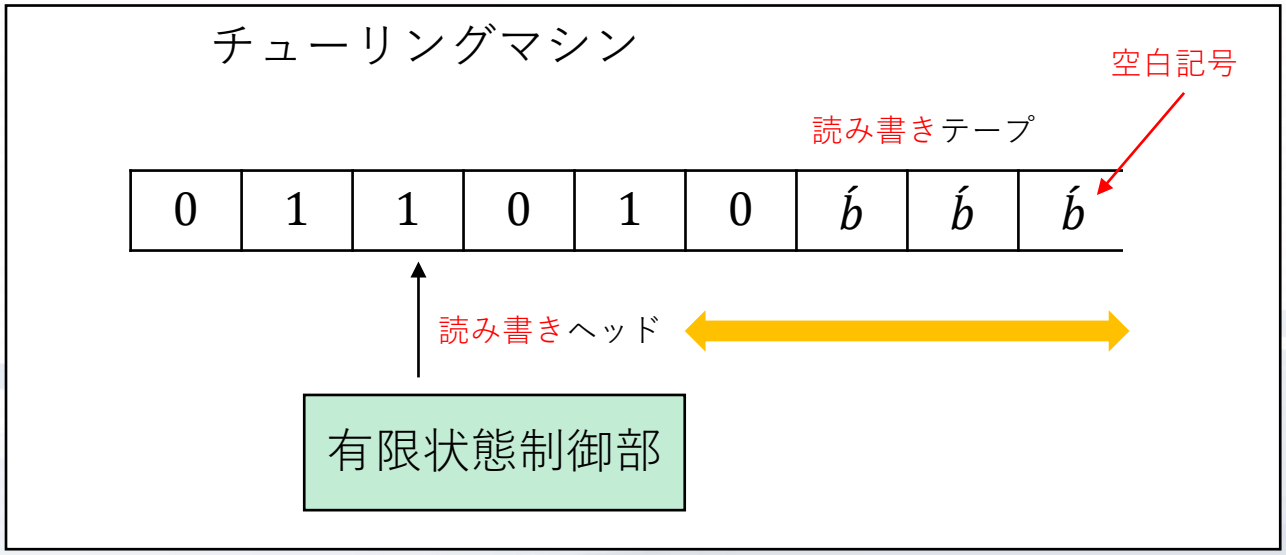
- プッシュダウンオートマトンオートマトンMは、四つの集合
  Q, Σ, Γ, F と、初期状態 $q_0 \in Q$ 、初期スタック記号 $Z_0 \in \Gamma$
  および状態遷移関数 δ を指定することにより定まる計算
  のモデルであり、以下のように表される.

$$M = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$$

- また各要素は以下の通り
  - Q：状態の集合
  - Σ：入力記号の集合
  - Γ：入力記号の集合
  - δ：状態遷移関数

チューリング機械の概要

- 半無限長の読み書きが自由に出来るテープを用いた有限状態機械
- 人間が論理的に考えることのできる言語は全て識別できる
- 線形拘束オートマトン：
  - 読み書きテープの使用できる範囲が、入力記号の書かれていた範囲あるいはその定数倍に限定.



チューリング機械の定義

- チューリング機械 M は、四つの集合 Q, Σ, Γ, F と、初期状態
  $q_0 \in Q$、空テープ記号 $\sqcup \in \Gamma$ および状態遷移関数 δ を指定することにより定まる計算のモデルであり、以下の
  ように表される.

$$M = (Q, \Sigma, \Gamma, \delta, q_0, \sqcup, F)$$

- また各要素は以下の通り
  - Q：状態の集合
  - Σ：入力記号の集合
  - Γ：テープ記号の集合であり、$\Sigma \subseteq \Gamma$ である
  - δ：状態遷移関数
  - F：受理状態の集合

チューリング機械の例

- 入力記号の個数を数え、奇数個なら1、偶数個なら0 を出力する
- 例: 入力記号 "AAAAA="
- 出力として、入力記号”A”の個数が偶数なら、記号 =
  の右側に 0 を書き込み、奇数なら 1 を書き込む
- 書き込み後は書き込んだ位置で停止する

「はじめて学ぶオートマトンと言語理論」 藤原暁宏 著　森北出版社 p.90

チューリング機械の例

- $M = (Q, \Sigma, \Gamma, \delta, q_0, \sqcup, F)$
- $Q = \{q_0, q_1, q_f\}, \Sigma = \{A, =\},$
- $\Gamma = \{A, =, 0, 1, \sqcup\}, F = \{q_f\}$

- $\delta(q_0, A) = (q_1, A, R), \delta(q_0, =) = (q_0, =, R),$
- $\delta(q_0, \sqcup) = (q_f, 0, S), \delta(q_1, A) = (q_0, A, R),$
- $\delta(q_1, =) = (q_1, =, R), \delta(q_1, \sqcup) = (q_f, 1, S)$

偶数

奇数　空テープ記号　ストップ

「はじめて学ぶオートマトンと言語理論」　藤原暁宏 著　森北出版社 p.90

バッカス記法（Backus Normal Form：BNF）

- 文脈自由文法の簡潔な記述
- プログラミング言語 ALGOL60 の構文記述用に開発された.
- Backus：世界初のコンパイラ FORTRAN 作成指導を行い、後にALGOL の開発･指導に当たった.
- 記号
  - ＜○○＞　　○○という概念を示す.
  - ：：＝　　左辺の記号が右辺で定義される.
  - ｜　　もう一つの定義、「または」

BNF 例（文脈自由文法）

- 整数を表す BNF
  - ＜整数＞：：＝“０”｜＜負整数＞＜正整数＞
  - ＜負整数＞ ：：＝“－”＜非零整数＞＜数字列＞
  - ＜正整数＞：：＝[“＋”] ＜非零整数＞＜数字列＞
  - ＜数字列＞：：＝ {＜数字＞}
  - ＜非零数字＞：：＝“１”｜“２”｜“３”｜“４”｜“５”｜“６”｜
    “７”｜“８”｜“９”
  - ＜数字＞：：＝“０”｜＜非零整数＞

BNF 例（文脈自由文法）

- メールアドレスを表す BNF
  - ＜メールアドレス＞：：＝＜ユーザ名＞“@”＜ドメイン＞
  - ＜ユーザ名＞ ：：＝＜英数字＞ {＜記号入り英数字＞}
  - ＜ドメイン＞：：＝ ＜英数字列＞ {“.”＜英数字列＞
  - ＜英数字列＞：：＝＜英数字＞ {＜英数字＞}
  - ＜記号入り英数字＞：：＝“ー”｜“_”｜ ＜英数字＞
  - ＜英数字＞：：＝“０”｜“１”｜“２”｜…｜“９”｜“a”｜“b”｜…
    ｜“ｘ”｜“ｙ”｜“ｚ”

「はじめて学ぶオートマトンと言語理論」 藤原暁宏 著　森北出版社 p.141

データサイエンス応用コース：応用Ⅰ ⑦-⑧

担当：中澤嵩（大阪大学MMDS数理科学ユニット・准教授）

1. はじめに

- このコンテンツの目的は，ウェブベースの計算環境である Jupyter Notebook（このウェブページを形作っているもの）を利用して，アルゴリズム(Python の基本操作)，数値計算(Numpy, PyTorch)，自然言語処理(正規表現)，分散・並列コンピューティング(GPU)に関する基本的な技術を習得することです．
- 主に大学院生，社会人が日々の研究や業務で活用できるように，テクニカルな内容については極力触れず，「使えるようになる」ことを前提として，データサイエンスの初学者向きに教材を作成しています．

2. Python の基本的な使用方法

2-1. Python とは

- Python は，科学技術界隈，特にデータ科学に関する分野において昨今，最も流行っていると言って過言ではないプログラミング言語です．これを習得することで今後，プログラミング技術なしでは為し得なかったような大量データの解析ができるようになります．
- Python は C言語とか Fortran とかの伝統的なプログラミング言語と比べて簡単に学ぶことができるプログラミング言語です．ものすごく人気がある言語であり，たくさんのライブラリ (補助ツール) が存在しているので，色々なことができます．計算の速さもそこそこで，少なくとも悪くはありません．

2. Python の基本的な使用方法

2-2. Python の実行方法

- もし以下のようなプログラムを書いて，それを実行するとターミナルウィンドウ等に「Hello world」という表示が出力されます．

```
def main():
        print("Hello world")
if __name__ == "__main__":
main()
```

2. Python の基本的な使用方法

2-3. Python の基本操作

2-3-1. 四則計算

```
def main():
        print(3 + 15) # addition
if __name__ == "__main__":
        main()
```

2. Python の基本的な使用方法

2-3. Python の基本操作

2-3-1. 四則計算

```
def main():
        print(10 - 2) # subtraction
        print(90 / 3) # division
        print(51 * 4) # multiplication
if __name__ == "__main__":
        main()
```

2. Python の基本的な使用方法

2-3. Python の基本操作

2-3-2. 変数

```
def main():
        a = 3 # assign a value '3' to a variable 'a‘
        b = 15
        print(a + b) # addition
        print(a - b) # subtraction
        print(b / a) # division
        print(a * b) # multiplication
if __name__ == "__main__":
        main()
```

2. Python の基本的な使用方法

2-3. Python の基本操作

2-3-2. 変数

```
def main():
        greeting = "Hello world"
        print(greeting)
 if __name__ == "__main__":
        main()
```

2. Python の基本的な使用方法

2-3. Python の基本操作

2-3-3. リストとディクショナリ

```
def main():
        list_a = [10, 4, "aaa"]
        print(list_a[0]) # access first element of list_a
        print(list_a[1]) # access second element of list_a
        print(list_a[2]) # access third element of list_a
if __name__ == "__main__":
        main()
```

2. Python の基本的な使用方法

2-3. Python の基本操作

2-3-3. リストとディクショナリ

```
def main():
        dict_a = {'January' : '1', 'February' : '2'}
        print(dict_a['January'])
        print(dict_a['February'])
if __name__ == '__main__':
        main()
```

2. Python の基本的な使用方法

2-3. Python の基本操作

2-3-3. リストとディクショナリ

```
def main():
        list_a = [10, 4, "aaa"]
        for w in list_a:
                print(w)
if __name__ == "__main__":
        main()
```

2. Python の基本的な使用方法

2-3. Python の基本操作

2-3-3. リストとディクショナリ

```
def main():
        dict_a={'January' : '1', 'February' : '2'}
        for k in dict_a.keys():
        print(k, dict_a[k])
if __name__ == "__main__":
        main()
```

2. Python の基本的な使用方法

2-4. アルゴリズム

2-4-1. 繰り返し処理

```
def main():
        for i in range(5):
                print(i)
if __name__ == "__main__":
        main()
```

2. Python の基本的な使用方法

2-4. アルゴリズム

2-4-1. 繰り返し処理

```
def main():
        i = 0
        while i != 10:
                print(i)
                i = i + 1
if __name__ == '__main__':
        main()
```

2. Python の基本的な使用方法

2-4. アルゴリズム

2-4-2. 条件分岐

```
def main():
        dict_a={'January' : '1', 'February' : '2', 'May' : '5'}
        for k in dict_a.keys():
                if k == 'January':
                        print(dict_a[k])
                elif dict_a[k] == '2':
                        print(k,"corresponds to",dict_a[k],"in dict_a.")
                else:
                        print(k,"is not January or February.")
if __name__ == "__main__":
        main()
```

2. Python の基本的な使用方法

2-4. アルゴリズム

2-4-3. 関数

```
def main():
        str_1 = "Python"
        func_1(str_1);
def func_1(arg_1):
        print("Hello", arg_1)
if __name__ == "__main__":
        main()
```

2. Python の基本的な使用方法

2-4. アルゴリズム

2-4-3. 関数

```
def main():
        numbers = [2, 4, 5, -2, 3]
        result_1 = summation(numbers)
        print(result_1)
def summation(arg_1):
        sumvalue = 0
        for number in arg_1:
                sumvalue = sumvalue + number
                return sumvalue
if __name__ == "__main__":
        main()
```

演習1（15分）

- これまで学習した，Pythonプログラミングの内容を各自で復習して下さい.
  - 四則演算
  - 変数
  - リストとディクショナリ
  - アルゴリズム
    - 繰り返し処理
    - 条件分岐
    - 関数

2. Python の基本的な使用方法

2-5. 自然言語処理

2-5-1. 正規表現

- プログラミング言語は「正規表現（regular expression）」と呼ばれる文字列パターン表現方法を持ちます.
- プログラミングで文字列を操作するための非常に強力なツールです.
- Python で正規表現を使用する場合は、「re」ライブラリを利用します.

2. Python の基本的な使用方法

2-5. 自然言語処理

2-5-2. テキストの抽出 re.findall

```
import re
def main():
        text = "The Volatility Index Japan (VXJ) provides a measure of how volatile the Japanese stock market would be over the next month and it is based on Nikkei225 index options. The VXJ index is presented herein, not so much as a tradable class of assets, but as a guide for firms in the dynamic hedging of exposures to equity risk, and for financial regulators in the supervision of banks' compliance with capital requirements through Value-at-
Risk modelling. This volatility index can be also useful in the examination of the reaction of volatility expectations to macroeconomic information. Empirical studies suggest that implied volatility indices provide reliable forecasts of short-
term volatility, and the "fear gauge" appellation is widely accepted in light of the negative dynamic relationship with market returns."
        liresult = re.findall("¥d+", text)
        print(liresult)
if __name__ == "__main__":
        main()
```

2. Python の基本的な使用方法

2-5. 自然言語処理

2-5-2. テキストの抽出 re.findall

```
import re
def main():
        text = "The Volatility Index Japan (VXJ) provides a measure of how volatile the Japanese stock market would be over the next month and it is based on Nikkei225 index options. The VXJ index is presented herein, not so much as a tradable class of assets, but as a guide for firms in the dynamic hedging of exposures to equity risk, and for financial regulators in the supervision of banks' compliance with capital requirements through Value-at-
Risk modelling. This volatility index can be also useful in the examination of the reaction of volatility expectations to macroeconomic information. Empirical studies suggest that implied volatility indices provide reliable forecasts of short-
term volatility, and the "fear gauge" appellation is widely accepted in light of the negative dynamic relationship with market returns."
        liresult = re.findall("[A-Z][a-z]+", text)
        print(liresult)
if __name__ == "__main__":
        main()
```

2. Python の基本的な使用方法

2-5. 自然言語処理

2-5-3. テキストの判定 re.search()

```
import re
def main():
        text = "The Volatility Index Japan (VXJ) provides a measure of how volatile the Japanese stock market would be over the next month and it is based on Nikkei225 index options. The VXJ index is presented herein, not so much as a tradable class of assets, but as a guide for firms in the dynamic hedging of exposures to equity risk, and for financial regulators in the supervision of banks' compliance with capital requirements through Value-at-
Risk modelling. This volatility index can be also useful in the examination of the reaction of volatility expectations to macroeconomic information. Empirical studies suggest that implied volatility indices provide reliable forecasts of short-
term volatility, and the "fear gauge" appellation is widely accepted in light of the negative dynamic relationship with market returns."
        if re.search("^¥d", text):
                print("Yes")
        else:
                print("No")
if __name__ == "__main__":
        main()
```

2. Python の基本的な使用方法

2-5. 自然言語処理

2-5-4. テキストの置換 re.sub

```
import re
def main():
        text = "The Volatility Index Japan (VXJ) provides a measure of how volatile the Japanese stock market would be over the next month and it is based on Nik
kei225 index options. The VXJ index is presented herein, not so much as a tradable class of assets, but as a guide for firms in the dynamic hedging of exposures to equity risk, and for fi
nancial regulators in the supervision of banks' compliance with capital requirements through Value-at-
Risk modelling. This volatility index can be also useful in the examination of the reaction of volatility expectations to macroeconomic information. Empirical studies suggest that implie
d volatility indices provide reliable forecasts of short-
term volatility, and the fear gauge appellation is widely accepted in light of the negative dynamic relationship with market returns."
        replaced_text = re.sub("¥(¥w+¥)¥s", "", text)
        print(replaced_text)
if __name__ == "__main__":
        main()
```

2. Python の基本的な使用方法

2-5. 自然言語処理

2-5-5. テキストの分割 re.split

```
import re
def main():
        text = "The Volatility Index Japan (VXJ) provides a measure of how volatile the Japanese stock market would be over the next month and it is based on Nikkei225 index options. The VXJ index is presented herein, not so much as a tradable class of assets, but as a guide for firms in the dynamic hedging of exposures to equity risk, and for financial regulators in the supervision of banks' compliance with capital requirements through Value-at-
Risk modelling. This volatility index can be also useful in the examination of the reaction of volatility expectations to macroeconomic information. Empirical studies suggest that implied volatility indices provide reliable forecasts of short-
term volatility, and the fear gauge appellation is widely accepted in light of the negative dynamic relationship with market returns."
        liresult = re.split("¥s+", text)
        print(liresult)
if __name__ == "__main__":
        main()
```

2. Python の基本的な使用方法

2-6. ライブラリのインポート

```
import statistics
def main():
        pass
if __name__ == "__main__":
        main()
```

2. Python の基本的な使用方法

2-6. ライブラリのインポート

```
import statistics
def main():
        pass
if __name__ == "__main__":
        main()

import statistics
import sys
def main():
        pass
if __name__ == "__main__":
        main()
```

2. Python の基本的な使用方法

2-6. ライブラリのインポート

2-6-1. 統計モジュール statistics

```
import statistics
def main():
        linumber=[1, 2, 3, 4, 5, 6, 7, 8]
        print("mean:", statistics.mean(linumber))
        print("sd:", statistics.stdev(linumber))
if __name__ == "__main__":
        main()
```

2. Python の基本的な使用方法

2-6. ライブラリのインポート

2-6-2. 数学モジュールmath

```
import math
def main():
        x=10
        print("log_10:", math.log10(x))
        print("log_2:", math.log2(x))
        print("log_e:", math.log(x))
        x=math.radians(180)
        print(math.sin(x))
        print(math.cos(x))
        print(math.tan(x))
if __name__ == "__main__":
        main()
```

2. Python の基本的な使用方法

2-6. ライブラリのインポート

2-6-3. 外部モジュール

- 外部のモジュールはインストールしなければなりません．そのためにはコマンドライン上で「pip3」というコマンドを用います．

```
! pip3 install numpy
```

- 現在使っている Python にインストールされているライブラリを確認するには以下のように打ちます．

```
! pip3 list
```

2. Python の基本的な使用方法

2-7. NumPy の基本的な使用方法

2-7-1. NumPy のインポート

- NumPy とは Python で利用可能な数値計算のライブラリです．さまざまな計算をコマンド一発で行うことができます．NumPy は以下のようにしてインポートします．読み込んだ NumPy には np という略称を与えることが普通です．

```
import numpy as np
def main():
        pass
if __name__ == "__main__":
        main()
```

2. Python の基本的な使用方法

2-7. NumPy の基本的な使用方法

2-7-2. ベクトルの基本的な計算

```
import numpy as np
def main():
        print(np.array([1, 2, 3]))
if __name__ == "__main__":
        main()
```

2. Python の基本的な使用方法

2-7. NumPy の基本的な使用方法

2-7-2. ベクトルの基本的な計算

```
import numpy as np
def main():
        na = np.array([1, 2, 3])
        print(na[0])
if __name__ == "__main__":
        main()
```

2. Python の基本的な使用方法

2-7. NumPy の基本的な使用方法

2-7-2. ベクトルの基本的な計算

```
import numpy as np
def main():
        na = np.array([1, 2, 3])
        nb = np.array([5, 7, 9])
        print(na + nb)
        print(na - nb)
        print(na * nb)
        print(nb / na)
if __name__ == "__main__":
        main()
```

2. Python の基本的な使用方法

2-7. NumPy の基本的な使用方法

2-7-2. ベクトルの基本的な計算

```
import numpy as np
def main():
        na = np.array([1, 1])
        nb = na.copy()
        print(na, nb)
        na[0] = 2
        nb[0] = 3
        print(na, nb)
if __name__ == "__main__":
        main()
```

2. Python の基本的な使用方法

2-7. NumPy の基本的な使用方法

2-7-2. ベクトルの基本的な計算

```
import numpy as np
def main():
        na = np.array([1, 1])
        nb = na
        print(na, nb)
        na[0] = 2
        nb[0] = 3
        print(na, nb)
if __name__ == "__main__":
        main()
```

2. Python の基本的な使用方法

2-7. NumPy の基本的な使用方法

2-7-3. 行列の基本的な計算

```
import numpy as np
 def main():
        na = np.array([[1, 2], [3, 4]])
        print(na)
        print(na.shape)
if __name__ == "__main__":
        main()
```

2. Python の基本的な使用方法

2-7. NumPy の基本的な使用方法

2-7-3. 行列の基本的な計算

```
import numpy as np
        def main():
        na = np.array([[1, 2], [3, 4]])
        print(na[0,:]) # all elements of row 1
        print(na[:,1]) # all elements of column 2
if __name__ == "__main__":
        main()
```

2. Python の基本的な使用方法

2-7. NumPy の基本的な使用方法

2-7-3. 行列の基本的な計算

```
import numpy as np
def main():
        na = np.array([[1, 2], [3, 4]])
        print(na.max())
        print(na.min())
        print(na.sum())
        print(na.mean())
        print(na.var())
        print(na.std())
if __name__ == "__main__":
        main()
```

2. Python の基本的な使用方法

2-7. NumPy の基本的な使用方法

2-7-3. 行列の基本的な計算

```
import numpy as np
def main():
        print(np.zeros((3,3)))
        print(np.ones((4,4)))
if __name__ == "__main__":
        main()
```

2. Python の基本的な使用方法

2-7. NumPy の基本的な使用方法

2-7-3. 行列の基本的な計算

```
import numpy as np
def main():
        na = np.array([[1, 2], [3, 4]])
        nb = np.array([[5, 6], [7, 8]])
        print(na + nb)
        print(nb - na)
        print(na * nb)
        print(nb / na)
if __name__ == "__main__":
        main()
```

2. Python の基本的な使用方法

2-7. NumPy の基本的な使用方法

2-7-3. 行列の基本的な計算

```
import numpy as np
        def main():
        na = np.array([[1, 2], [3, 4]])
        nb = np.array([[5, 6], [7, 8]])
        print(np.dot(na, nb))
if __name__ == "__main__":
        main()
```

2. Python の基本的な使用方法

2-7. NumPy の基本的な使用方法

2-7-3. 行列の基本的な計算

```
import numpy as np
np.random.seed(0)
def main():
        print(np.random.rand(3, 3))
if __name__ == "__main__":
        main()
```

2. Python の基本的な使用方法

2-7. NumPy の基本的な使用方法

2-7-3. 行列の基本的な計算

```
import numpy as np
def main():
        na = np.array([[4, 15, 4], [-11, 5, 6], [2, 4, 8]])
        print(np.linalg.det(na))
        print(np.linalg.inv(na))
if __name__ == "__main__":
        main()
```

2. Python の基本的な使用方法

2-7. NumPy の基本的な使用方法

2-7-3. 行列の基本的な計算

```
import numpy as np
def main():
        na = np.array([[3, 4, 1, 4], [1, 2, 1, 1], [1, 1, 2, 1], [1, 1, 1, 2]])
        eigenvalue, eigenvector = np.linalg.eig(na)
        print("eigenvalue",eigenvalue)
        print("eigenvector",eigenvector)
if __name__ == "__main__":
        main()
```

2. Python の基本的な使用方法

2-7. NumPy の基本的な使用方法

2-7-3. 行列の基本的な計算

```
import numpy as np
def main():
        na = np.array([[3, 4, 1, 4], [1, 2, 1, 1], [1, 1, 2, 1], [1, 1, 1, 2]])
        print(na ** 2)
if __name__ == "__main__":
        main()
```

2. Python の基本的な使用方法

2-7. NumPy の基本的な使用方法

2-7-3. 行列の基本的な計算

```
import numpy as np
def main():
        na = np.array([[3, 4, 1, 4], [1, 2, 1, 1], [1, 1, 2, 1], [1, 1, 1, 2]])
        print(np.linalg.matrix_power(na, 2))
if __name__ == "__main__":
        main()
```

2. Python の基本的な使用方法

2-7. NumPy の基本的な使用方法

2-7-3. 行列の基本的な計算

```
import numpy as np
def main():
        na = np.array([[3, 4, 1, 4], [1, 2, 1, 1], [1, 1, 2, 1], [1, 1, 1, 2]])
        print(na + 2)
if __name__ == "__main__":
        main()
```

2. Python の基本的な使用方法

2-7. NumPy の基本的な使用方法

2-7-4. 特殊な操作

```
import numpy as np
def main():
        na = np.array([1, 2, 3, 4, 5, 6, 7])
        print(na[::-1])
if __name__ == "__main__":
        main()
```

2. Python の基本的な使用方法

2-7. NumPy の基本的な使用方法

2-7-4. 特殊な操作

```
import numpy as np
def main():
        na = np.array([1, 2, 3, 4, 5, 6, 7])
        print(na > 3)
if __name__ == "__main__":
        main()
```

2. Python の基本的な使用方法

2-7. NumPy の基本的な使用方法

2-7-4. 特殊な操作

```
import numpy as np
def main():
        na = np.array([1, 2, 3, 4, 5, 6, 7])
        print(na[na > 3])
if __name__ == "__main__":
        main()
```

演習2（15分）

- これまで学習した，Pythonプログラミングの内容を各自で復習して下さい.
  - 正規表現
  - ライブラリのインポート
  - Numpyの基本的な使用方法
  - 余裕のある方は，PythorchによるMLPの実装を確認すること

データサイエンス応用コース：応用Ⅰ⑦-⑧

担当：中澤嵩（大阪大学MMDS数理科学ユニット・准教授）

内容：アルゴリズム，数値計算，自然言語処理，分散・並列コンピューティング

1　はじめに

1-1. このコンテンツで学ぶこと

このコンテンツの目的は，ウェブベースの計算環境である Jupyter Notebook（このウェブページを形作っているもの）を利用して，アルゴリズム(Python の基本操作)，数値計算(Numpy, PyTorch)，自然言語処理(正規表現)，分散・並列コンピューティング(GPU)に関する基本的な技術を習得することです．受講者（主に大学院生，社会人）が日々の研究や業務で活用できるように，テクニカルな内容については極力触れず，「使えるようになる」ことを前提として，データサイエンスの初学者向きに教材を作成しています．

より高度なプログラミングを希望される方は，昨年度末に大阪大学MMDSが開催していたGPUプログラミング講習会のURLをご覧下さい．大阪大学サイバーメディアセンターが所有するGPUクラスターOCTOPUSを活用して実習を行いました．

イベント web site：https://koikezlemma.github.io/gpu/

OCTOPUS web site: http://www.hpc.cmc.osaka-u.ac.jp/octopus/

1-2. この環境について

1-2-1. Jupyter Notebook

Jupyter Notebook は Python を実行するための環境です．メモを取りながら Python のコードを実行することができます．この環境は，Python プログラムが実行される実際の環境とは少し異なるのですが，Python プログラムがどのように動くかということを簡単に確認しながら学習することができます． ご自身の研究や仕事の業務で使う場合には，anacondaをインストールしてJupyter Notebookを使ってください．Google Colabはあくまでも演習用の教材として準備しています．また，Google Colbは利用時間(ランタイム)に制限があり，GPUが使えるからと言って膨大な学習時間を必要とする深層学習の研究・業務には向いていないかもしれません．あくまで，「お手軽感が最も重要なアプリケーション」という認識が正しいかもしれません．

anaconda web site: https://www.anaconda.com/

1-2-2. GPU の利用方法

この環境（Google Colab）で GPU を利用するには上のメニューの「ランタイム」から「ランタイムのタイプを変更」と進み，「ハードウェアアクセラレータ」の「GPU」を選択します．ただし，ご自身のPCでGPU計算を行う際には，NVIDIAが提供するライブラリのCUDAを呼び出して実行する必要があります．詳細については，GPUプログラミング講習会にご参加ください．また，GPUのアーキテクチャに関する資料については下記を参照してください．

https://koikezlemma.github.io/gpu/GPU01.pdf

1-3. 開始前に行うこと（重要）

1-4. 利用方法

1-4-1. 進め方

上から順番に読み進めます．Python のコードが書かれている場合は実行ボタンをクリックして実行します．コード内の値を変えたり，関数や条件分岐を変えたりして挙動を確かめてみてください．その上で，データサイエンスで求められるアルゴリズムや数値計算，モジュールのインポート，GPUを用いた並列計算等について学習します．勿論，全てを網羅することは不可能ですので初級～中級レベルを想定しつつ，講義後に不明な点が発生した場合の問題解決方法について体験して頂ければと思います． 中級～上級レベルを希望される方はGPUプログラミング講習会のURLをご覧下さい．

1-4-2. コードセル

コードセルとは，Python のコードを書き込み実行するためのセルです．実行するためには以下のような灰色のボックスです．ここにコードを書きます．実行は，コードセルの左に表示される「実行ボタン」をクリックするか，コードセルを選択した状態で Ctrl + Enter を押します．環境によっては行番号が表示されていると思いますので注意してください．

```
print("MMDS aims to develop a cross-disciplinary educational program to enable new innovation using mathema
```

```
MMDS aims to develop a cross-disciplinary educational program to enable new innovation using mathema
```

2. Python の基本的な使用方法

2-1. Python とは

Python は，科学技術界隈，特にデータ科学に関する分野において昨今，最も流行っていると言って過言ではないプログラミング言語です．これを習得することで今後，プログラミング技術なしでは為し得なかったような大量データの解析ができるようになります．Python は C言語とか Fortran とかの伝統的なプログラミング言語と比べて簡単に学ぶことができるプログラミング言語です．ものすごく人気がある言語であり，たくさんのライブラリ (補助ツール) が存在しているので，色々なことができます．計算の速さもそこそこで，少なくとも悪くはありません．ここでは，Python のバージョン3（Python3）の使い方を学びます．ご自身のPCでPythonを使いたい場合には，anacondaをインストールすることになります．

anaconda web site: https://www.anaconda.com/

2-2. Python の実行方法

もし以下のようなプログラムを書いて，それを実行するとターミナルウィンドウ等に「Hello world」という表示が出力されます．以下のプログラムにおいて，1〜3と5〜7行は Python にとって常に必要になる表記です．これは，いかなるときにも取り敢えず書いておくと良いでしょう．なので，以下のプログラムの本体は実質，4行目だけです．これは，「Hello world」という文を画面に表示しなさいという命令です． `print()` という表記の前に存在する空白は，キーボードの `Tab` を押すと入力できる空白文字です．Pythonではこのように空白文字をきちんと入れる必要があります． `def main():` 以下に書いた，タブの右側の表記だけがプログラムとして認識されます．

```
#!/usr/bin/env python3

def main():
    print("Hello world")

if __name__ == "__main__":
    main()
```

```
Hello world
```

2-3. Python の基本操作

編集するにはダブルクリックするか Enter キーを押してください

以降，実際にプログラムを書いてそれらを実行してみます．

2-3-1. 四則計算

以下のように書くと足し算ができます．プログラム中の「#」はコメントの宣言のための記号で，これが書いてある部分から右側はプログラムとして認識されません．

```
#!/usr/bin/env python3

def main():
    print(3 + 15) # addition

if __name__ == "__main__":
    main()
```

```
18
```

その他の四則計算は以下のように書きます．

```
#!/usr/bin/env python3

def main():
    print(10 - 2) # subtraction
    print(90 / 3) # division
    print(51 * 4) # multiplication

if __name__ == "__main__":
    main()
```

```
8
30.0
204
```

2-3-2. 変数

上では，実際の数値を入力してプログラムを書きました．しかし，これではただの電卓です．以下のようにすると，変数を定義することができます．例えば，4行目で「a」という変数を生成し，これに「3」という数値を代入しています．ここで注意しなければならないことは，プログラミング言語において「=」は「その右側と左側が等しい」という意味ではなく，「右側を左側に代入する」という意味で用いられる点です．このように変数というものを使うことでプログラムは色々な処理を一般化することができます．

```
#!/usr/bin/env python3
```

```
def main():
    a = 3 # assign a value '3' to a variable 'a'
    b = 15
    print(a + b) # addition
    print(a - b) # subtraction
    print(b / a) # division
    print(a * b) # multiplication

if __name__ == "__main__":
    main()
```

```
18
-12
5.0
45
```

変数として使うことができるものはアルファベット一文字だけではありません．以下のように文字列を使用することができます．

```
#!/usr/bin/env python3

def main():
    aaa = 5
    print(aaa)

if __name__ == "__main__":
    main()
```

```
5
```

変数として使用できる文字は以下のものを含みます．

- アルファベット
- 数字

しかし以下のようなことは禁止事項です．

- 変数の最初の文字が数字であること．
- 予約語と等しい文字列．例えば，「print」という変数を使おうとしても使えません．これは Python が「print」という関数を備えているためです．

変数には数値だけでなく，文字列や文字も代入することができます．以下のようにすると，「greeting」という変数が「print()」という関数に処理されて画面に表示されますが，「greeting」に代入された値は「Hello world」なので，画面に表示される文字列は同じく「Hello world」となります．

```
#!/usr/bin/env python3

def main():
    greeting = "Hello world"
```

```
    print(greeting)

if __name__ == "__main__":
    main()
```

```
Hello world
```

2-3-3. リストとディクショナリ

複数個の値を代入することができる変数があります．これのことをただの変数ではなく，Python ではリスト（list）と呼びます．以下のように生成した変数「list_a」の要素にアクセスするには「list_a[0]」のような感じでリスト変数に添字を付けて書きます．こうすることで「list_a」に格納されている最初の値である「10」にアクセスできます．多くのプログラミング言語ではリストの一番最初を示すインデックスは「0」です（1ではありません．）．こういうシステムのことはゼロオリジンと呼称します．

```
#!/usr/bin/env python3

def main():
    list_a = [10, 4, "aaa"] # declaration of a variable, list_a, this list variable contains three values
    print(list_a[0]) # access first element of list_a
    print(list_a[1]) # access second element of list_a
    print(list_a[2]) # access third element of list_a

if __name__ == "__main__":
    main()
```

```
10
4
aaa
```

Python のリストでは上のように数値と文字（列）をおなじリストに入れることができます．同様のリストを使うことが可能な多くのプログラミング言語ではこのような操作はできない場合が多いです．

また，以下のような特別な変数も存在します．これをディクショナリ（dictionary）と呼びます．ディクショナリ変数には，キー（key）とそれに対応する値（value）を一組にして代入します．その後，キーの値を使って以下のようにしてディクショナリ変数にアクセスすることで，そのキーに対応する値を取得することができます．

```
#!/usr/bin/env python3

def main():
    dict_a = {'January' : '1', 'February' : '2'} # declaration of dictionary, dict_a. this dictionary conta
    print(dict_a['January']) # access a value corresponding to a key 'January'
    print(dict_a['February']) # access a value corresponding to a key 'January'

if __name__ == '__main__':
```

```
main()
```

```
1
2
```

以下のように書くとリストに格納されているデータ全てにひとつずつアクセスすることができます．この「for」というのは繰り返し処理をするためのシステムです．プログラムではこの「for」を様々な局面で利用することによって人間ではできないような同じことの繰り返し作業を実現します．この繰り返し処理の方法と後に出る条件分岐の方法（if）さえ用いればどのようなプログラムでも実現することができます．繰り返し操作が終わった次の行からは，インデントを再びアウトデントするのを忘れないよう注意しましょう．

```
#!/usr/bin/env python3

def main():
    list_a = [10, 4, "aaa"] # declaration of variable list_a
    for w in list_a: # repetitive access of list_a. w represents an element.
        print(w) # when you start a 'function' you have to insert [tab] (and when you finish the function,

if __name__ == "__main__":
    main()
```

```
10
4
aaa
```

ディクショナリに対する繰り返しのアクセスは以下のようにします．

```
#!/usr/bin/env python3

def main():
    dict_a={'January' : '1', 'February' : '2'}
    for k in dict_a.keys(): # dict_a.keys() is a list containing only keys of 'dict_a'. or you can write si
        print(k, dict_a[k]) # k is a key and dict_a[k] is a value corresponding to the key.

if __name__ == "__main__":
    main()
```

```
January 1
February 2
```

2-3-4. CSVファイルの入出力

CSVファイルの入出力については下記のサイトを参照してください．

https://note.nkmk.me/python-csv-reader-writer/

2-4. アルゴリズム

2-4-1. 繰り返し処理

既に上で出てきましたが，Python で繰り返し処理をするには「for」を使います．以下のように書くと，range(5) に格納されている「0，1，2，3，4」という値が一行ごとに表示されます．

```
#!/usr/bin/env python3

def main():
    for i in range(5):
        print(i)

if __name__ == "__main__":
    main()
```

```
0
1
2
3
4
```

繰り返し処理をする方法にはその他に，「while」があります．以下のようにして使います．「while」の右側に書いた条件が満たされる場合のみ，それ以下の処理を繰り返すというものです．この場合，iという変数が10より小さいとき処理が繰り返されます．そして，この「i」の値は「while」が1回実行されるたびに1ずつインクリメントされています．その表記が，「i = i + 1」です．これは，「i」に「iに1を足した値」を代入しろという意味です．なので，この表記の前のiに比べて，表記の後のiは1だけ大きい値です．

```
#!/usr/bin/env python3

def main():
    i = 0
    while i < 10: # If the condition 'i < 10' is satisfied,
        print(i)
        i = i + 1 # these two lines are processed.

if __name__ == "__main__":
    main()
```

```
0
1
2
3
4
5
```

```
6
7
8
9
```

また，while は以下のようにして用いることもできます．この「!=」は「この記号の右側と左側の値が等しくない」ということを意味します．

```
#!/usr/bin/env python3

def main():
    i = 0
    while i != 10:
        print(i)
        i = i + 1

if __name__ == '__main__':
    main()
```

```
0
1
2
3
4
5
6
7
8
9
```

2-4-2. 条件分岐

プログラミング言語にとって重要な関数として「if」があります．これは条件分岐のためのシステムです．以下のように書くと，もしキーが「January」であった場合，「1」が表示されます．それ以外でもし，「ディクショナリ変数の値が2」であった場合，February corresponds to 2 in dict_a. と表示されます．それら以外の全ての場合おいては，key is not January or February.と表示されます．ここで，== という記号は「記号の右側と左側の値が等しい」ということを意味します．現実世界の「=」と同義です．

```
#!/usr/bin/env python3

def main():
    dict_a={'January' : '1', 'February' : '2', 'May' : '5'}
    for k in dict_a.keys():
        if k == 'January': # a condition means if variable 'k' is same as 'January'
            print(dict_a[k]) # If the above condition is satisfied, this line is executed. do not forget ar
        elif dict_a[k] == '2': # If 'k' is not 'January' and 'dict_a[k]' is same as '2',
            print(k,"corresponds to",dict_a[k],"in dict_a.") # this line is executed.
        else: # In all situation except above two condition
```

```
        else: # In all situation except above two condition,
            print(k,"is not January or February.") # this line is executed.

if __name__ == "__main__":
    main()
```

```
1
February corresponds to 2 in dict_a.
May is not January or February.
```

2-4-3. 関数

プログラミングをするにあたり，関数というものはとても大事な概念です．関数はあるプロセスの一群をまとめるためのシステムです．書いたプロセスを関数化することで，ソースコードの可読性が上がります．また，処理に汎用性を与えることができるようになります．同じ処理を変数の値だけを変えて何度も行いたい場合は同じ処理を何度も書くより，関数化して，必要なときにその関数を呼び出す，といったやり方が効率的です．関数は， def というものを使って生成します．これまでに書いてきたプログラムにも def main() という表記がありました．これは， main() という関数を定義するものです．これまでは，この関数の一部としてプログラムを書いていたのです．そして，その main() という関数をソースコードの一番下，if **name** == "**main**":で呼び出して使っていたということです．この main() 以外にも，プログラマーは自由に関数を定義することができます．以下のように func_1() という関数を作って，それを main() の中で実行することができます．

```
#!/usr/bin/env python3

def main():
    func_1(); # The function 'func_1()' is executed in a function 'main()'.

def func_1(): # Declaration of a novel function 'func_1()'.
    print("Hello") # substance of 'func_1()'

if __name__ == "__main__":
    main() # The function 'main()' is executed (and 'main()' execute 'func_1()' and 'func_1()' execute 'pri
```

```
Hello
```

関数を作成する際に () が付属していますが，これは，引数（ひきすう，パラメータ）の受け渡しに使うためのものです．以下のように書くと，Python という文字列が格納された変数 str_1 を関数 func_1() のパラメータとして関数が実行されます．その後，func_1() 内では，パラメータ str_1 == Python を func_1() の中でのみ定義されるパラメータ変数である arg_1 で受け取りそれを利用して新たな処理が実行されます．

```
#!/usr/bin/env python3

def main():
    str_1 = "Python"
```

```
    func_1(str_1); # pass the variable 'str_1' to the function 'func_1'

def func_1(arg_1): # To get argument from external call, you need to set a variable to receive the argument
    print("Hello", arg_1) # print "Hello" and arg_1 (= str_1 in main() = "Python").

if __name__ == "__main__":
    main()
```

```
Hello Python
```

また，関数はその関数内で処理された結果の値を関数の呼び出し元に返す (呼び出し元の関数内で使うことができるように値を渡す) ことができます．例えば，以下の書くと，summation() の呼び出し元に対して，summation() で計算した結果（リスト内の数値の和）が返ってきます．その帰ってきた値を変数 result_1 に格納してその後の処理に利用することができます．

```
#!/usr/bin/env python3

def main():
    numbers = [2, 4, 5, -2, 3] # a list to be calculated
    result_1 = summation(numbers) # execution of summation(). Numbers is passed to the function. And the re
    print(result_1)

def summation(arg_1): # arg_1 is a variable to contain the argument (numbers in main()).
    sumvalue = 0 # A variable to contain a result of summation.
    for number in arg_1:
        sumvalue = sumvalue + number # this means 'sum is renewed by the value of sum + number'
    return sumvalue # return the result of above calculation to the function caller by the word 'return'

if __name__ == "__main__":
    main()
```

```
12
```

2-5. 自然言語処理

ここでは，自然言語処理で求められる必要最低限のプログラミング知識として正規表現を主に学習します．これに「2-3-3. リストとディクショナリ」や「3-3. MLP」を転用することで，機械学習における自然言語処理が可能となります．因みに，応用コースⅡ25-27を担当する上坂先生は自然言語処理が専門です．

2-5-1. 正規表現

プログラミング言語は「正規表現（regular expression）」と呼ばれる文字列パターン表現方法を持ちます．プログラミングで文字列を操作するための非常に強力なツールです．Python で正規表現を使用

する場合は、「re」 ライブラリを利用します．ここでは，以下のような文を用いて正規表現を学びます．

2-5-2. テキストの抽出 re.findall

The Volatility Index Japan (VXJ) provides a measure of how volatile the Japanese stock market would be over the next month and it is based on Nikkei225 index options. The VXJ index is presented herein, not so much as a tradable class of assets, but as a guide for firms in the dynamic hedging of exposures to equity risk, and for financial regulators in the supervision of banks' compliance with capital requirements through Value-at-Risk modelling. This volatility index can be also useful in the examination of the reaction of volatility expectations to macroeconomic information. Empirical studies suggest that implied volatility indices provide reliable forecasts of short-term volatility, and the "fear gauge" appellation is widely accepted in light of the negative dynamic relationship with market returns.

この文に存在するアラビア数字を抜き出したいとき，以下のようなプログラムを書きます．

```
#!/usr/bin/env python3

import re

def main():
    text = "The Volatility Index Japan (VXJ) provides a measure of how volatile the Japanese stock market w
    liresult = re.findall("¥d+", text)
    print(liresult)

if __name__ == "__main__":
    main()
```

```
['225']
```

このコードにおいて7行目の \d+ が正規表現です．\d が「アラビア数字の任意の1文字」を表し，その後の + は「前の表現の1回以上の繰り返すこと」を意味しています．すなわち，\d+ は「1つ以上のアラビア数字が連続した文字列」を意味します．関数 re.findall() は最初の引数に指定されたパターン（正規表現）を2番目ので指定された文字列より検索して抽出するための役割を果たします．

次に，以下のプログラムを実行します．これによって得られるリストには，「大文字から始まる文字列」が含まれます．使用する正規表現において，[A-Z] は大文字，[a-z] は小文字を意味します．

```
#!/usr/bin/env python3

import re

def main():
    text = "The Volatility Index Japan (VXJ) provides a measure of how volatile the Japanese stock market w
```

```
    text = "The Volatility Index Japan (VXJ) provides a measure of how volatile the Japanese stock market w
    liresult = re.findall("[A-Z][a-z]+", text)
    print(liresult)

if __name__ == "__main__":
    main()
```

```
['he', 'olatility', 'ndex', 'apan', 'provides', 'a', 'measure', 'of', 'how', 'volatile', 'the', 'apa
```

2-5-3. テキストの判定 re.search()

以下のように，関数 re.search() を用いると，文の最初（^）にアラビア数字が含まれるかどうかを判定できます．

```
#!/usr/bin/env python3

import re

def main():
    text = "The Volatility Index Japan (VXJ) provides a measure of how volatile the Japanese stock market w
    if re.search("^¥d", text):
        print("Yes")
    else:
      print("No")

if __name__ == "__main__":
    main()
```

```
Yes
```

2-5-4. テキストの置換 re.sub

以下のように書くと，「(から始まり，何らかの文字が1個以上連続し，) で終了する文字列」を全て削除（何もないものに置換）することができます．関数 re.sub() は第3引数の文字列から，第1引数の文字列を同定し，それを第2引数の値で置換するためのものです．この場合，(および) をそれぞれ (および) のように表記していますが，これは (および) が正規表現のための記号として用いられるものであるため，正規表現としてそれらを利用したい場合には，そのための特別な表記としなければならないためです．この操作は，「エスケープする」というように表現します．また，\w は「何らかの文字」，\s は「空白文字（スペース等）」を意味します．

```
#!/usr/bin/env python3

import re

def main():
```

```
    text = "The Volatility Index Japan (VXJ) provides a measure of how volatile the Japanese stock market w
    replaced_text = re.sub("¥(¥w+¥)¥s", "", text)
    print(replaced_text)

if __name__ == "__main__":
    main()
```

```
The Volatility Index Japan provides a measure of how volatile the Japanese stock market would be ove
```

以下の関数 re.split() を利用すると「文字列を空白文字がひとつ以上連続した文字列（\s+）」を区切り文字として分割することができます．これは文字列処理をする場合によく用いる関数です．

2-5-5. テキストの分割 re.split

```
#!/usr/bin/env python3

import re

def main():
  text = "The Volatility Index Japan (VXJ) provides a measure of how volatile the Japanese stock market wou
  liresult = re.split("¥s+", text)
  print(liresult)

if __name__ == "__main__":
    main()
```

```
['The', 'Volatility', 'Index', 'Japan', '(VXJ)', 'provides', 'a', 'measure', 'of', 'how', 'volatile',
```

2-5-6. Python の正規表現

Python の正規表現をまとめたウェブサイトには以下のようなものがあります．

- https://www.debuggex.com/cheatsheet/regex/python
- https://docs.python.org/3/howto/regex.html

2-5-7. ファイルの入出力

Pythonでテキストファイルの入出力をまとめたサイトが下記にあります．

https://note.nkmk.me/python-file-io-open-with/

2-6. ライブラリのインポート

上では関数を作ってプログラミングを便利にしましたが，世界には様々な便利な関数をまとめたライブラリが存在します．これを使うと様々な便利なことをすごく簡単な記述で実現可能です．そういった関数の集まりをライブラリと言いますが，ここではそれらライブラリの使い方を学びます．ライブラリは以下のように，import という表記によりプログラム中で使うことを宣言します（利用する直前に呼び出しても良い）．以下では，「statistics」というモジュールを使うことを明示しています．以下は実行しても何もしない（ライブラリをインポートすることだけをする）プログラムです．

```
#!/usr/bin/env python3

import statistics

def main():
    pass

if __name__ == "__main__":
    main()
```

複数個のモジュールを呼び込むときは以下のように書くことができます．

```
#!/usr/bin/env python3

import statistics
import sys

def main():
    pass

if __name__ == "__main__":
    main()
```

2-6-1. 統計モジュール statistics

例えば，モジュール「statistics」を用いれば上で行なった平均や標準偏差の計算は1行で可能です．モジュールに入っている関数を使うときは「モジュール名.関数名()」という形式で書きます．

```
#!/usr/bin/env python3

import statistics

def main():
    linumber=[1, 2, 3, 4, 5, 6, 7, 8]

    print("mean:", statistics.mean(linumber))
    print("sd:", statistics.stdev(linumber))
```

```
if __name__ == "__main__":
    main()
```

```
  mean: 4.5
  sd: 2.449489742783178
```

詳細は下記のサイトを参照してください.

https://docs.python.org/ja/3/library/statistics.html

2-6-2. 数学モジュールmath

また，モジュール「math」を用いれば数学で用いられる様々な関数 (数学における関数) を簡単に記述することができます.

```
#!/usr/bin/env python3

import math

def main():
    x=10
    print("log_10:", math.log10(x))
    print("log_2:", math.log2(x))
    print("log_e:", math.log(x))

    x=math.radians(180) #This is radian of an angle of 90 degrees
    print(math.sin(x))
    print(math.cos(x))
    print(math.tan(x))

if __name__ == "__main__":
    main()
```

```
  log_10: 1.0
  log_2: 3.321928094887362
  log_e: 2.302585092994046
  1.2246467991473532e-16
  -1.0
  -1.2246467991473532e-16
```

詳細は下記のサイトを参照してください.

https://docs.python.org/ja/3/library/math.html

2-6-3. 外部モジュール

これまでは Python にデフォルトで組み込まれているモジュールのみを使ってきましたが，外部で用意されているモジュールを使うことも可能です．外部のモジュールはインストールしなければなりません．そのためにはコマンドライン上で「pip3」というコマンドを用います．例えば，数値計算を行うためのモジュール「NumPy」は以下のようにすることでインストールすることができます．現在使っている Python（正確には Python3）と pip3 が紐付いており，pip3 で外部からインストールしたライブラリは Python3 から呼び出すことができるのです．

```
! pip3 install numpy
```

現在使っている Python にインストールされているライブラリを確認するには以下のように打ちます．

```
! pip3 list
```

2-7. NumPy の基本的な使用方法

2-7-1. NumPy のインポート

NumPy とは Python で利用可能な数値計算のライブラリです．さまざまな計算をコマンド一発で行うことができます．NumPy は以下のようにしてインポートします．読み込んだ NumPy には np という略称を与えることが普通です．

```
#!/usr/bin/env python3

import numpy as np

def main():
    pass

if __name__ == "__main__":
    main()
```

2-7-2. ベクトルの基本的な計算

ベクトルは以下のように生成します．

```
#!/usr/bin/env python3

import numpy as np

def main():
```

```
    print(np.array([1, 2, 3]))

if __name__ == "__main__":
    main()
```

要素の参照は普通の Python 配列と同じようにできます．もちろんゼロオリジンです．

```
#!/usr/bin/env python3

import numpy as np

def main():
    na = np.array([1, 2, 3])
    print(na[2])

if __name__ == "__main__":
    main()
```

ベクトルの四則計算は以下のようにします．NumPy は基本的に要素ごとに（element-wise）値を計算します．

```
#!/usr/bin/env python3

import numpy as np

def main():
    na = np.array([1, 2, 3])
    nb = np.array([5, 7, 9])
    print(na + nb)
    print(na - nb)
    print(na * nb)
    print(nb / na)

if __name__ == "__main__":
    main()
```

コピーをする際は気を使わなければならない点があります．あるベクトルから別のベクトル変数を生成，つまり，コピーでベクトルを生成した場合，その生成したベクトルを元のベクトルと別のものとして扱いたい場合は以下のようにしなければなりません．以下では，9行目と10行目で元のベクトルとコピーで生成されたベクトルの要素をそれぞれ別の値で変更していますが，それぞれ別の値にて要素が置換されていることがわかります．

```
#!/usr/bin/env python3

import numpy as np

def main():
```

```
    na = np.array([1, 1])
    nb = na.copy()
    print(na, nb)
    na[0] = 2
    nb[0] = 3
    print(na, nb)

if __name__ == "__main__":
    main()
```

一方で，以下のように = を使ってコピーをすると生成されたベクトルは元のベクトルの参照となってしまい，（この場合の）意図している操作は実現されません．上の挙動とこの挙動は把握していないと結構危険です．

```
#!/usr/bin/env python3

import numpy as np

def main():
    na = np.array([1, 1])
    nb = na
    print(na, nb)
    na[0] = 2
    nb[0] = 5
    print(na, nb)

if __name__ == "__main__":
    main()
```

その他のベクトル計算は以下を参照して下さい．

https://stats.biopapyrus.jp/python/vector.html

2-7-3. 行列の基本的な計算

行列を生成するためにも，np.array() を利用します．さらに，行列のサイズは .shape によって確認することができます．

```
#!/usr/bin/env python3

import numpy as np

def main():
    na = np.array([[1, 2], [3, 4]])
    print(na)
    print(na.shape)

if __name__ == "__main__":
```

```
if __name__ == "__main__":
    main()
```

NumPy 行列は以下のようなアクセスの方法があります．行ごとまたは列ごとのアクセスです．これは多用します．

```
#!/usr/bin/env python3

import numpy as np

def main():
    na = np.array([[1, 2], [3, 4]])
    print(na[0,:]) # all elements of row 1
    print(na[:,1]) # all elements of column 2

if __name__ == "__main__":
    main()
```

以下のようにすると行列に関する様々な統計値を得ることができます．

```
#!/usr/bin/env python3

import numpy as np

def main():
    na = np.array([[1, 2], [3, 4]])
    print(na.max())
    print(na.min())
    print(na.sum())
    print(na.mean())
    print(na.var())
    print(na.std())

if __name__ == "__main__":
    main()
```

以下の np.zeros() や np.ones() を用いると引数で指定したサイズの，全要素が0または1の行列を生成することができます．

```
#!/usr/bin/env python3

import numpy as np

def main():
    print(np.zeros((3,3)))
    print(np.ones((4,4)))

if __name__ == "__main__":
    main()
```

四則計算は以下のようにします．これもやはり，element-wise な計算です．

```
#!/usr/bin/env python3

import numpy as np

def main():
    na = np.array([[1, 2], [3, 4]])
    nb = np.array([[5, 6], [7, 8]])
    print(na + nb)
    print(nb - na)
    print(na * nb)
    print(nb / na)

if __name__ == "__main__":
    main()
```

行列の掛け算は以下のようにします．

```
#!/usr/bin/env python3

import numpy as np

def main():
    na = np.array([[1, 2], [3, 4]])
    nb = np.array([[5, 6], [7, 8]])
    print(np.dot(na, nb))

if __name__ == "__main__":
    main()
```

以下のようにすると一様分布に従う乱数を生成することができます．以下の例は一様分布のものですが，NumPy には一様分布以外にもたくさんの分布が用意されています．引数で指定するのは行列のサイズです．Python や NumPy に限らず計算機実験をする際に気を付けなければならないことに「乱数のタネを固定する」ということがあります．計算機実験の再現性を得るためにとても重要なので絶対に忘れないようにすべきです．乱数のタネは4行目で行っています．ここでは 0 を設定しています．

```
#!/usr/bin/env python3

import numpy as np
np.random.seed(0)

def main():
    print(np.random.rand(3, 3))

if __name__ == "__main__":
    main()
```

```
    main()
```

np.random.rand(3, 3)は3×3の行列要素に0以上1未満の乱数を与えるためのコマンドとなっています．
その他の乱数生成コマンドについては下記を参照してください．

https://note.nkmk.me/python-numpy-random/

以下のようにすると行列式と逆行列を計算することができます．

```
#!/usr/bin/env python3

import numpy as np

def main():
    na = np.array([[4, 15, 4], [-11, 5, 6], [2, 4, 8]])
    print(np.linalg.det(na)) # determinant of matrix
    print(np.linalg.inv(na)) # inverse matrix

if __name__ == "__main__":
    main()
```

当然，逆行列を求める際には行列式が非零である必要があります．ところで，膨大なサンプリング数が前提となる場合，大規模な行列をメモリーに格納する必要がある．そこで，低次元化を施すことが可能な疑似逆行列を活用することが一般的のようである．

https://techacademy.jp/magazine/33550

行と列が同じ要素数の行列に関する固有値分解は以下のようにします．

```
#!/usr/bin/env python3

import numpy as np

def main():
    na = np.array([[3, 4, 1, 4], [1, 2, 1, 1], [1, 1, 2, 1], [1, 1, 1, 2]])
    eigenvalue, eigenvector = np.linalg.eig(na)
    print("eigenvalue",eigenvalue)
    print("eigenvector",eigenvector)

if __name__ == "__main__":
    main()
```

因みに，固有値計算は代表的な教師なし学習である主成分分析の基本的な数値計算アルゴリズムです．ただし，疑似逆行列を求める際のように，行と列の要素数が異なる場合，特異値分解が必要となり，詳細は下記のサイトを参照してください．

https://ohke.hateblo.jp/entry/2017/12/14/230500

以下のようにすると「行列の要素の冪乗」を計算できます．

```
#!/usr/bin/env python3

import numpy as np

def main():
    na = np.array([[3, 4, 1, 4], [1, 2, 1, 1], [1, 1, 2, 1], [1, 1, 1, 2]])
    print(na ** 2)

if __name__ == "__main__":
    main()
```

一方で，「行列の冪乗」は以下のように計算します．

```
#!/usr/bin/env python3

import numpy as np

def main():
    na = np.array([[3, 4, 1, 4], [1, 2, 1, 1], [1, 1, 2, 1], [1, 1, 1, 2]])
    print(np.linalg.matrix_power(na, 2))

if __name__ == "__main__":
    main()
```

NumPy は「行列にスカラを足す」，このような異様な計算をしても結果を返してくれます．以下の6行目では，最初に生成した4行4列の行列と同じサイズの，全要素が2からなる行列を自動で生成し，その行列と最初の4行4列の行列の和を計算しています．このような，対象となる行列のサイズに合せて，スカラから行列を生成することを「ブロードキャスト」と言います．この機能は非常に便利で様々な局面で使用することがあります．

```
#!/usr/bin/env python3

import numpy as np

def main():
    na = np.array([[3, 4, 1, 4], [1, 2, 1, 1], [1, 1, 2, 1], [1, 1, 1, 2]])
    print(na + 2)

if __name__ == "__main__":
    main()
```

2-7-4. 特殊な操作

以下のようにすると配列の順番を逆向きにして用いることができます．魔法ですね．

```
#!/usr/bin/env python3

import numpy as np

def main():
    na = np.array([1, 2, 3, 4, 5, 6, 7])
    print(na[::-1])

if __name__ == "__main__":
    main()
```

以下のようにすると指定した条件に対する bool 配列を得ることができます．

```
#!/usr/bin/env python3

import numpy as np

def main():
    na = np.array([1, 2, 3, 4, 5, 6, 7])
    print(na > 3)

if __name__ == "__main__":
    main()
```

これを利用すると条件に合う要素のみに選択的にアクセスすることができます．以下では条件に合う要素を単に出力しているだけですが，例えば，条件に合う値のみを0に置き換えるとか，そのような操作ができます．

```
#!/usr/bin/env python3

import numpy as np

def main():
   na = np.array([1, 2, 3, 4, 5, 6, 7])
   print(na[na > 3])

if __name__ == "__main__":
    main()
```

その他の行列計算は下記を参照してください．

https://note.nkmk.me/python-numpy-matrix/

3. PyTorch による MLP の実装

3-1. PyTorch の基本的な使用方法

テンソル

PyTorch では「テンソル」と呼ばれる NumPy の多次元配列に類似したデータ構造を用います．3行目で PyTorch をインポートし，4行目では乱数のタネを0に固定しています．7行目のテンソルを生成するためのコマンドは torch.empty() で，引数の生成されるテンソルの次元数を指定します．また，一様分布に従うテンソル，全要素が0のテンソルはそれぞれ torch.rand() および torch.zeros() にて生成します．

```
#!/usr/bin/env python3

import torch
torch.manual_seed(1)

def main():
    tx = torch.empty(3, 3)
    print(tx)

if __name__ == "__main__":
    main()
```

Python 配列より変換することもできます．以下のようにします．以下では dtype = torch.float を指定していますが，これはデータを32ビットで扱うための指定です．

```
#!/usr/bin/env python3

import torch
torch.manual_seed(0)

def main():
    tx = torch.tensor([2, 4], dtype=torch.float)
    print(tx)

if __name__ == "__main__":
    main()
```

深層学習はテンソルの掛け算をたくさん行います．そのため，テンソルのサイズを確認したいときがあるはずですが，それは以下のようにやります．NumPy の場合は .shape という属性を用いていましたが，テンソルではメソッド .size() を用います．

```
#!/usr/bin/env python3
```

```
#!/usr/bin/env python3

import torch
torch.manual_seed(0)

def main():
    tx = torch.tensor([[2, 4], [6, 8]], dtype=torch.float)
    print(tx.size())

if __name__ == "__main__":
    main()
```

3-1-2. 四則計算

テンソルの四則計算は以下のように行います．NumPy と同じようにやはり element-wise な計算です．

```
#!/usr/bin/env python3

import torch
torch.manual_seed(0)

def main():
    tx = torch.tensor([[2, 4], [6, 8]], dtype=torch.float)
    ty = torch.tensor([[1, 2], [3, 4]], dtype=torch.float)
    print(tx + ty)
    print(tx - ty)
    print(tx * ty)
    print(tx / ty)

if __name__ == "__main__":
    main()
```

行列の積は以下のように torch.matmul() を利用するのが楽です．

```
#!/usr/bin/env python3

import torch
torch.manual_seed(0)

def main():
    tx = torch.tensor([[2, 4], [6, 8]], dtype=torch.float)
    ty = torch.tensor([[1, 2], [3, 4]], dtype=torch.float)
    print(torch.matmul(tx, ty))

if __name__ == "__main__":
    main()
```

3-1-3. 特殊な操作

以下のようなスライスの実装も NumPy と同じです．

```
#!/usr/bin/env python3

import torch
torch.manual_seed(0)

def main():
    tx = torch.tensor([[2, 4], [6, 8]], dtype=torch.float)
    print(tx[0,:])

if __name__ == "__main__":
    main()
```

テンソルのサイズの変更には view() を利用します．

```
#!/usr/bin/env python3

import torch
torch.manual_seed(0)

def main():
    tx = torch.rand(4, 5)
    print(tx)
    print(tx.view(20))
    print(tx.view(-1, 4))

if __name__ == "__main__":
    main()
```

以上のプログラムの7行目では4行4列の行列が生成されています．これを，1行20列の行列に変換するのが9行目の記述です．また，10行目の記述では5行4列の行列が生成されます．ここでは，view() の最初の引数に -1 が指定されていますが，これのように書くと自動でその値が推測されます．この場合，5であると推測されています．

3-1-4. 変数の変換

PyTorch では「NumPy 配列」，「テンソル」，「GPU 上のテンソル」，この3つの変数のタイプを自由に変換することができます．NumPy 配列から直接 GPU 上のテンソルへの変換はできませんが．

```
#!/usr/bin/env python3

import torch
torch.manual_seed(0)
```

```
import numpy as np

def main():
    na = np.ones(5)
    print("NumPy:", na)
    ta = torch.tensor(na, dtype=torch.float)
    print("Torch:", ta)
    na = ta.numpy()
    print("NumPy:", na)
    device = torch.device("cuda")
    ca = ta.to(device)
    print("CUDA:", ca)
    ta = ca.to("cpu", dtype=torch.float)
    print("Torch:", ta)

if __name__ == "__main__":
    main()
```

以上のプログラムにおいて，8行目は NumPy 配列を生成します．10行目はその NumPy 配列をテンソルに変換します．さらに，NumPy 配列に戻すためには12行目のように .numpy() を利用します．14行目では CUDA のデバイスを定義しています．この環境で GPU を利用するには上のメニューの「ランタイム」から「ランタイムのタイプを変更」と進み，「ハードウェアアクセラレータ」の「GPU」を選択します．定義したデバイスに変数を送るには .to() を利用します．

3-1-5. 勾配の計算

深層学習に関するニューラルネットワークやディープラーニングの詳細なアルゴリズムについては下記のサイトを参照して頂きつつ，応用コースⅠ15-18でも講義予定です．ここでは，Pythonを用いた演算に焦点を当てて講義を進めます．

https://koikezlemma.github.io/gpu/neural-network.pdf

深層学習の最も基本的な構成要素は行列の掛け算と微分です．PyTorch はこれを行うライブラリです．自動微分機能を提供します．微分をしたい変数の生成は以下のように行います．テンソル型の変数を生成する際に，requires_grad=True を追加するだけです．

```
#!/usr/bin/env python3

import torch

def main():
    tx = torch.tensor(5, dtype=torch.float, requires_grad=True)
    print(tx)

if __name__ == "__main__":
    main()
```

```
tensor(5., requires_grad=True)
```

ここで，以下の式を考えます．

$y = x^2 + 2$

これに対して以下の偏微分を考えることができます．

$\frac{\partial y}{\partial x} = 2x$

よって $x = 5$ のときの偏微分の値は以下のように計算できます．

$\left.\frac{\partial y}{\partial x}\right|_{x=5} = 10$

これを PyTroch で実装すると以下のようになります．微分は8行目のように backward() によって行います．

```
#!/usr/bin/env python3

import torch

def main():
    tx = torch.tensor(5, dtype=torch.float, requires_grad=True)
    ty = tx**2 + 2
    ty.backward()
    print(tx.grad)

if __name__ == "__main__":
    main()
```

```
tensor(10.)
```

上の程度の微分だとこの自動微分機能はさほど有難くないかもしれませんが，以下のような計算となると，そこそこ有難くなってきます．以下では，(1,2) の行列と (2,2) の行列 tt と (2,2) の行列 tu と (2,1) の行列を順に掛けることで，最終的に (1,1) の行列の値，スカラー値を得ますが，それを tt および tu で微分した値を計算しています．

```
#!/usr/bin/env python3

import torch

def main():
    # Definition
    ts = torch.tensor([[1, 1]], dtype=torch.float)
    tt = torch.tensor([[2, 4], [6, 8]], dtype=torch.float, requires_grad=True)
    tu = torch.tensor([[1, 2], [3, 4]], dtype=torch.float, requires_grad=True)
    tv = torch.tensor([[1], [1]], dtype=torch.float)
    # Calculation
    ta = torch.matmul(ts, tt)
    tb = torch.matmul(ta, tu)
```

```
    tc = torch.matmul(tb, tu)
    ty = torch.matmul(tc, tv)
    ty.backward()
    print(tt.grad)
    print(tu.grad)

if __name__ == "__main__":
    main()
```

```
tensor([[17., 37.],
        [17., 37.]])
tensor([[ 68., 100.],
        [100., 148.]])
```

なぜ微分を求めたいかというと，勾配降下法でパラメータをアップデートしたいからです．以下では勾配降下法を実装してみます．勾配降下法は関数の最適化法です．ある関数に対して極小値を計算するためのものです．以下のような手順で計算が進みます．

1. 初期パラメータ（θ_0）をランダムに生成します．
2. もしパラメータ（θ_t）が最適値または，最適値に近いなら計算をやめます．ここで，t は以下の繰り返しにおける t 番目のパラメータです．
3. パラメータを以下の式によって更新し，かつ，t の値を 1 だけ増やします．ここで，α は学習率と呼ばれる更新の大きさを決める値で，g_t は t のときの目的の関数の勾配です．
   $\theta_{t+1} = \theta_t - \alpha g_t$
4. ステップ2と3を繰り返します．

ここでは以下の関数を考えます．

$y = (x+1)^2 + 2$

初期パラメータを以下のように決めます．

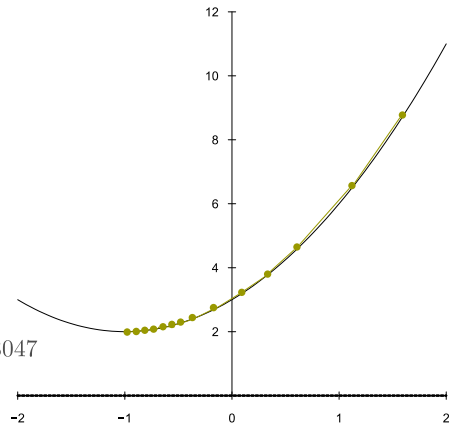
$x_0 = 1.6$

この関数の極小値を見つけたいのです．これは解析的に解くのはとても簡単で，括弧の中が0になる値，すなわち x が -1 のとき，極小値 $y = 2$ です．

勾配降下法で解くと，以下の図のようになります．勾配降下法は解析的に解くことが難しい問題を正解の方向へ少しずつ反復的に動かしていく方法です．

勾配降下法

$$
\begin{aligned}
x_1 &= x_0 - \alpha g_0 \\
&= 1.6 - 0.1 \cdot 5.2 \\
&= 1.08 \\
x_2 &= x_1 - \alpha g_1 \\
&= 1.08 - 0.1 \cdot 4.16 \\
&= 0.664 \\
&\vdots \\
x_{30} &= x_{29} - \alpha g_{29} \\
&= -0.99598 - 0.1 \cdot 0.008047 \\
&= -0.99678
\end{aligned}
$$



これを PyTorch を用いて実装すると以下のようになります．出力中，Objective は目的関数の値，Solution はその時点での解です．最終的に $x = -0.9968 \simeq -1$ のとき，最適値 $y = 2$ が出力されています．

```
#!/usr/bin/env python3

import torch

def main():
    tx = torch.tensor(1.6, dtype=torch.float, requires_grad=True)
    epoch, update_value, lr = 1, 5, 0.1
    while abs(update_value) > 0.001:
        ty = (tx + 1)**2 + 2
        ty.backward()
        with torch.no_grad():
            update_value = lr * tx.grad
            tx = tx - update_value
            print("Epoch {:4d}:¥tObjective = {:5.3f}¥tSolution = {:7.4f}".format(epoch, ty, tx.numpy()))
            tx.requires_grad = True
        epoch = epoch + 1

if __name__ == "__main__":
    main()
```

6行目で最初のパラメータを発生させています．通常は乱数によってこの値を決めますが，ここでは上の図に合わせて1.6とします．次の7行目では，最初のエポック，更新値，学習率を定義します．エポックとは（ここでは）パラメータの更新回数のことを言います．8行目は終了条件です．以上のような凸関数においては勾配の値が0になる点が本当の最適値ではありますが，計算機的にはパラメータを更新

する値が大体0になったところで計算を打ち切ります．この場合，「大体0」を「0.001」としました．9行目は目的の関数，10行目で微分をしています．11行目に with torch.no_grad(): という記述がありますが，この記述の下のインデント中の記述では勾配計算のための計算の記録がされません．ここで計算する値についての勾配を計算する必要がないためです．13行目の計算で x をアップデートします．これが上述の勾配降下法の式です．11行目はその更新値を計算するためのものです．また，値の更新によって requires_grad=True が消えてしまうため（PyTorch の仕様），14行目で再度追加します．

3-2. 扱うデータの紹介

3-2-1. MNIST について

ここでは，機械学習界隈で最も有名なデータセットである MNIST（Mixed National Institute of Standards and Technology database）を解析対象に用います．MNIST は縦横28ピクセル，合計784ピクセルよりなる画像データです．画像には手書きの一桁の数字（0から9）が含まれています．公式ウェブサイトでは，学習データセット6万個とテストデータセット1万個，全部で7万個の画像からなるデータセットが無償で提供されています．

編集するにはダブルクリックするか Enter キーを押してください

3-2-2. ダウンロードと正規化

公式サイトよりダウンロードしてきても良いのですが，PyTorch がダウンロードするためのユーティリティを準備してくれているため，それを用います．以下の torchvision.datasets.MNIST() を用いることで可能となります．

```
#!/usr/bin/env python3

import torch
import torchvision

def main():
    learn_dataset = torchvision.datasets.MNIST(root="dataset", train=True, download=True, transform=torchvi
    test_dataset = torchvision.datasets.MNIST(root="dataset", train=False, download=True, transform=torchvi

if __name__ == "__main__":
  main()
```

以上のプログラムを実行することで MNIST をダウンロードすることができました．以下のようにするとダウンロードディレクトリである dataset が生成されていることを確認することができます．これは

以上の7行目および8行目で指定したディレクトリです．

```
! ls
```

```
dataset  mnist_mlp.pt  sample_data
```

7行目および8行目のオプションである train は学習データセットを扱うための値です．また，download = True となっていますが，上のプログラムを再度実行したとしても既にデータが存在しているなら再度のダウンロードは実行されません．

最後のオプション transform は，読んで字のごとく，データの変換を行うためのものです．ここでは，torchvision.transforms.Compose([torchvision.transforms.ToTensor(), torchvision.transforms.Normalize((0.1307,), (0.3081,))]) を指定しています．この torchvision.transforms.Compose() はその引数に指定された配列型の変換を同時に指定するだけの単純な関数です．そこで指定した配列型データの最初の要素である torchvision.transforms.ToTensor() はデータを PyTroch のテンソル型に変換するためのものです．また，torchvision.transforms.Normalize() はデータの正規化をするためのものです．元の MNIST の構成要素の平均値は0.1307で，標準偏差は0.3081です．この値を用いて標準化を行っています．タプルで指定されていますが，これは複数チャネルに対応しているためです（MNIST は1チャネル）．

3-2-3. データの可視化

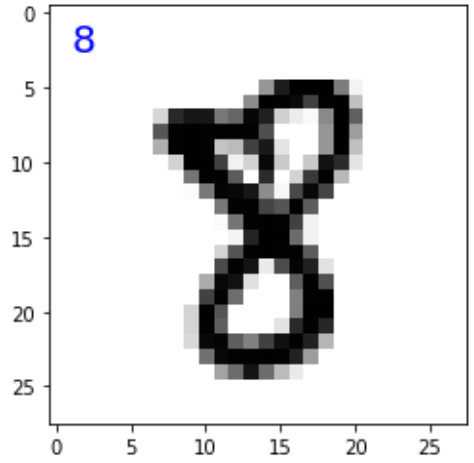
前述のとおり，MNIST は縦28横28ピクセルからなる画像データです．そのうちのひとつを可視化してみます．以下のプログラムで可視化できます．

```
#!/usr/bin/env python3
import torch
torch.manual_seed(0)
import torchvision
import matplotlib.pyplot as plt
%matplotlib inline

def main():
    MINIBATCHSIZE=200
    learn_dataset = torchvision.datasets.MNIST(root="dataset", train=True, download=True, transform=torchvi
    test_dataset = torchvision.datasets.MNIST(root="dataset", train=False, download=True, transform=torchvi
    learnloader = torch.utils.data.DataLoader(learn_dataset, batch_size=MINIBATCHSIZE, shuffle=True)
    testloader = torch.utils.data.DataLoader(test_dataset, shuffle=False)
    lix, lit = iter(learnloader).next()
    npimg = lix[100].numpy()
    npimg = npimg.reshape((28, 28))
    plt.imshow(npimg, cmap="gray_r")
    plt.text(1, 2.5, int(lit[100]), fontsize=20, color="blue")

if __name__ == "__main__":
```

```
main()
```



上のプログラムで，10行目と11行目はデータをダウンロードして PyTorch 用に変換，正規化するためのものでひとつ前のプログラムと同じ記述です．

12行目と13行目では，「データローダー」を定義しています．12行目が学習データセットのデータローダーで13行目がテストデータセットのデータローダーです．深層学習では（多くの機械学習でも），最初にランダムに生成した性能の良くない予測器に対して，与えられたデータを何度も何度も繰り返し学習させることでその予測器を成長させ，高性能の予測器を良くします．データローダーとは与えられたデータを呼び出し，予測器に提示する役割をするものです．データローダーは torch.utils.data.DataLoader() で生成できます．以上のプログラムでは10行目および11行目で MNIST をダウンロードしてそれを PyTorch 用のテンソルデータに変換していますが，もし自分でデータを用意した場合にもそれと同じ手順でデータをテンソルデータに変換して，データローダーを用いて学習を進めます．12行目では10行目で生成した学習データセット learn_dataset を最初の引数に設定しています．3番目の引数はデータをシャッフルするかどうか，つまり，並び替えて使うかどうかを指定するためのオプションです．ここでは乱数を使うため，乱数のタネを指定しなければなりません．それを行っているのが3行目の記述 torch.manual_seed(0) です．この場合，乱数のタネを0に設定しています．2番目の引数には MINIBATCHSIZE という9行目で定義した整数の値が指定されていますが，これはミニバッチ学習の際のミニバッチのサイズを決定するハイパーパラメータ（機械学習の過程によって値が決定されるパラメータを単にパラメータ，学習前に解析者が決めなければならない学習に依らないパラメータをハイパーパラメータと呼びます）です．機械学習をする際には繰り返し与えられたデータを読みこむことを上述しましたが，ミニバッチ学習はそのうちのひとつのテクニックです．予測器に1個ずつデータを読ませる方法を逐次更新法とかオンライン学習と言います．逆に，一度に全部のデータを読ませ，パラメータを更新する方法を一括更新法とかバッチ学習法と言います．多くの開発者の経験上，逐次更新法は局所解に陥りやすい，一括更新法は最適解に到達しにくい，という弱点がありますが，その中間をとって解析者があらかじめ決めたデータサイズ分のデータをランダムに選びそれを予測器に読ませることをするのがミニバッチ学習法です．この場合，ミニバッチサイズは200なので，データを200ずつ選択し，それに基づいて予測器のパラメータを更新することとなります．

14行目はデータローダーからひとまとまりのデータを抽出するための記述です．この場合，ひとまとまりはミニバッチのサイズです．iter() はイテレータのための記述です．イテレータとは「次の要素に

繰り返しアクセスする」ことを実現するインターフェイスのことです．.next() によって次のひとまとまりのデータにアクセスすることができます．15行目は抽出したデータの最初の要素を抜き出すための記述です．今回の場合，ミニバッチのサイズは200であるため，ひとまとまりには200個のデータが含まれています．よって，lix[199] とすると200番目のデータにアクセスできます．lix[200] とするとエラーが出るはずです．16行目はデータを縦28横28の画像に変換するための記述，17と18行目はインポートしたライブラリ matplotlib で描画するための記述です．

プログラムを実行することで表示されたように，MNIST で予測するのはこの手書きの数字が0から9のどのクラスに属するのかを予測することです．

3-3. MLP の実装

この教材では，Google Colabを用いて計算しますが，GPUプログラミング講習会では実際にOCTOPUSを用いてMLPを行いました．ご興味のある方はGPUプログラミング講習会URLをご覧ください．

3-3-1. 実行と出力結果について

以上のデータセットの分類（0～9のクラス分類）を行うための予測器を MLP にて構築します．そのためのプログラムが以下です．29行目の USECUDA という変数で GPU を利用するかどうかを選択しますが，これを現在設定しているのランタイムのタイプに合わせて実行してみてください．

```
#!/usr/bin/env python3

import torch
torch.manual_seed(0)
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
import torchvision
import time

def main():

    # Hyperparameter setting
    MINIBATCHSIZE = 1000
    TOTALEPOCH = 10
    USECUDA = True

    # Dataset preparation
    learn_dataset = torchvision.datasets.MNIST(root="dataset", train=True, download=True, transform=torchvi
    test_dataset = torchvision.datasets.MNIST(root="dataset", train=False, download=True, transform=torchvi

    # Making training and validation dataset from the learning dataset
    train_dataset, valid_dataset = torch.utils.data.random_split(learn_dataset, [int(len(learn_dataset) * 0
```

```
# Usage of CPU or GPU
device = torch.device("cuda" if USECUDA else "cpu")
kwargs = {'num_workers': 1, 'pin_memory': True} if USECUDA else {}

# Definition of data loader
trainloader = torch.utils.data.DataLoader(train_dataset, batch_size=MINIBATCHSIZE, shuffle=True, **kwar
validloader = torch.utils.data.DataLoader(valid_dataset, batch_size=len(valid_dataset), shuffle=False,
testloader = torch.utils.data.DataLoader(test_dataset, batch_size=len(test_dataset), shuffle=False, **k

# Generating a model
model = Network().to(device)
optimizer = optim.Adam(model.parameters())

# Learning process
for epoch in range(1, TOTALEPOCH + 1):
    # Training
    model.train() # training mode
    traincostsum, minibatchnumber = 0, 0
    for input, target in trainloader:
        input, target = input.to(device), target.to(device)
        optimizer.zero_grad()
        output = model(input)
        traincost = F.nll_loss(output, target)
        traincost.backward()
        optimizer.step()
        traincostsum += traincost.item()
        minibatchnumber += 1
    traincostavg = traincostsum / minibatchnumber
    # Validation
    model.eval() # evaluation mode to stop dropout
    validcost, correct = 0, 0
    with torch.no_grad():
        for input, target in validloader:
            input, target = input.to(device), target.to(device)
            output = model(input)
            validcost = F.nll_loss(output, target).item()
            prediction = output.argmax(dim=1, keepdim=True)
            correct = prediction.eq(target.view_as(prediction)).sum().item()
    print("Epoch {:5d}: Training cost= {:.4f}, Validation cost= {:.4f}, Validation ACC= {:.4f}".format(

# Test process (It should not be included in the program of learning)
model.eval() # evaluation mode to stop dropout
testcost, correct = 0, 0
with torch.no_grad():
    for input, target in testloader:
        input, target = input.to(device), target.to(device)
        output = model(input)
        testcost = F.nll_loss(output, target).item()
        prediction = output.argmax(dim=1, keepdim=True)
        correct = prediction.eq(target.view_as(prediction)).sum().item()
testcost /= len(testloader.dataset)
print("Test cost= {:.4f}, Test ACC: {:.4f}".format(testcost, correct / len(testloader.dataset)))

# Saving parameter
```

```
    torch.save(model.state_dict(), "mnist_mlp.pt")

class Network(nn.Module):
    def __init__(self):
        super(Network, self).__init__()
        self.fc1 = nn.Linear(28 * 28, 128)
        self.fc2 = nn.Linear(128, 10)
        self.dropout = nn.Dropout2d(0.5)
    def forward(self, x):
        x = torch.flatten(x, 1)
        x = self.fc1(x)
        x = F.relu(x)
        x = self.dropout(x)
        x = self.fc2(x)
        output = F.log_softmax(x, dim=1)
        return output

if __name__ == "__main__":
    main()
```

```
Downloading http://yann.lecun.com/exdb/mnist/train-images-idx3-ubyte.gz to dataset/MNIST/raw/train-ir
                                                                9920512/? [00:20<00:00, 21699082.46it/s]
Extracting dataset/MNIST/raw/train-images-idx3-ubyte.gz to dataset/MNIST/raw
Downloading http://yann.lecun.com/exdb/mnist/train-labels-idx1-ubyte.gz to dataset/MNIST/raw/train-la
                                                                32768/? [00:00<00:00, 387538.43it/s]
Extracting dataset/MNIST/raw/train-labels-idx1-ubyte.gz to dataset/MNIST/raw
Downloading http://yann.lecun.com/exdb/mnist/t10k-images-idx3-ubyte.gz to dataset/MNIST/raw/t10k-imag


                                                                1654784/? [00:19<00:00, 139873.02it/s]
Extracting dataset/MNIST/raw/t10k-images-idx3-ubyte.gz to dataset/MNIST/raw
Downloading http://yann.lecun.com/exdb/mnist/t10k-labels-idx1-ubyte.gz to dataset/MNIST/raw/t10k-labe
                                                                0/? [00:00<?, ?it/s]
Extracting dataset/MNIST/raw/t10k-labels-idx1-ubyte.gz to dataset/MNIST/raw
Processing...
Done!
/usr/local/lib/python3.6/dist-packages/torchvision/datasets/mnist.py:480: UserWarning: The given NumI
  return torch.from_numpy(parsed.astype(m[2], copy=False)).view(*s)
Epoch     1: Training cost= 0.8504, Validation cost= 0.3540, Validation ACC= 0.8996
Epoch     2: Training cost= 0.3816, Validation cost= 0.2650, Validation ACC= 0.9252
Epoch     3: Training cost= 0.3102, Validation cost= 0.2219, Validation ACC= 0.9374
Epoch     4: Training cost= 0.2748, Validation cost= 0.1959, Validation ACC= 0.9434
Epoch     5: Training cost= 0.2445, Validation cost= 0.1739, Validation ACC= 0.9497
Epoch     6: Training cost= 0.2263, Validation cost= 0.1611, Validation ACC= 0.9525
Epoch     7: Training cost= 0.2094, Validation cost= 0.1498, Validation ACC= 0.9554
Epoch     8: Training cost= 0.1936, Validation cost= 0.1402, Validation ACC= 0.9587
Epoch     9: Training cost= 0.1840, Validation cost= 0.1332, Validation ACC= 0.9617
Epoch    10: Training cost= 0.1752, Validation cost= 0.1263, Validation ACC= 0.9625
Test cost= 0.0000, Test ACC: 0.9635
```

結果として，エポック1から10までのトレーニングのコスト，バリデーションのコストと正確度（ACC），最後の行にテストデータセットにおけるコストと ACC が表示されました．ここで，1エポックの完了は学習データセットを全部なめ終わることを意味します． エポックを経るに従い，トレーニングおよびバリデーションデータセットにおけるコストは減少し，ACC が増加しているのがわかります．すなわち，学習が正常に進んでいることがわかります． 最後に，学習セット（トレーニングセットとバリデーションセット）とは異なるデータセットであるテストセットにてその性能を評価していますが，ACC はほぼ100%であり，高い性能で予測が行われていることがわかります

3-3-2. プログラムの解説

最初に，10行目からのハイパーパラメータの設定ですが，ここで設定しているのは，ミニバッチのサイズと最大エポックサイズと GPU を学習の際に用いるかどうかです．

```
# Hyperparameter setting
MINIBATCHSIZE = 1000
TOTALEPOCH = 10
USECUDA = False
```

ミニバッチの意味が解っていないのか，多くの解説サイトにバリデーションやテストにもミニバッチを設定している人がとても多くいますが，基本的（サイズが大きすぎてメモリに載らないとか，Theano のようにミニバッチのサイズをトレーニングと揃えなければならないフレームワークを使う場合以外）にはバリデーションやテストの際にミニバッチを設定する必要はありません．

次に，以下の16および17行目の記述ですが，これは上で説明したとおりです．

```
learn_dataset = torchvision.datasets.MNIST(root="dataset", train=True, download=True, transform=torc
test_dataset = torchvision.datasets.MNIST(root="dataset", train=False, download=True, transform=torc
```

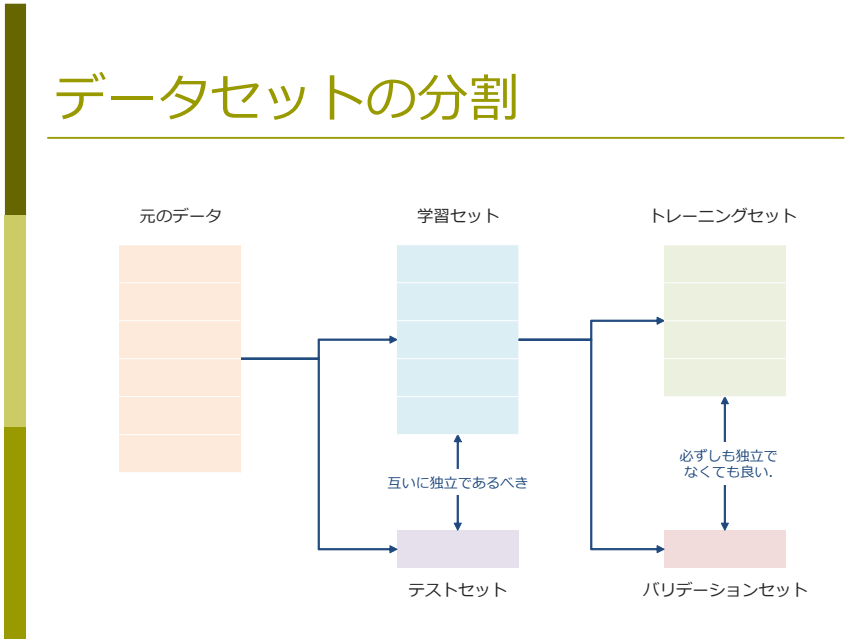
20行目は機械学習をする際に最も重要なプロセスのひとつです．ここでは，データセットを以下の3つに分けています．

- トレーニングセット（学習セット）
- バリデーションセット（学習セット）
- テストセット

トレーニングセットは予測器構築のためのパラメータを決めるのに使うデータセットです．これを何度も何度も読み込んで，勾配降下法を繰り返し，最終的なパラメータが決まります．バリデーションセットは，学習が変なことになっていないか（過学習）を確認するために用いる疑似テストセットです．テストセットは構築した予測器（機械学習の最終産物）の性能を測るためのデータセットです．学習セットとは独立でなければなりません．

以下の記述によって学習セットをトレーニングセットとバリデーションセットに分けています．トレーニングセットの割合は学習セットの0.8，バリデーションセットの割合は0.2です．

```
# Making training and validation dataset from the learning dataset
train_dataset, valid_dataset = torch.utils.data.random_split(learn_dataset, [int(len(learn_dataset)
```



学習の際には学習セットだけの結果を観察します．学習の際に一瞬でもテストセットにおける予測器の性能を観察するべきではありません．また，「独立であること」もしっかり定義すべきです．データが互いに独立であるとは，機械学習の文脈においては「互いに同じ分布に従うことが予想されるが，その生成過程が異なること」でしょうか．MNIST の学習セットとテストセットには，同一の手書き数字の提供者が同時に含まれている可能性があり，この観点からするとそれらは互いに独立なデータセットではないのかもしれません．

次に，23行目の記述ですが，もし，GPU を利用する場合，`device = torch.device("cuda")` なる変数を定義します（PyTorchの仕様）．

```
device = torch.device("cuda" if USECUDA else "cpu")
kwargs = {'num_workers': 1, 'pin_memory': True} if USECUDA else {}
```

24行目には，GPU を利用する場合には，`num_workers` を 1 に，`pin_memory` を `True` に設定するための記述をしています．`num_workers` はデータの読み込みの際に使用するスレッドの数を指定するものですが，GPU を利用する場合にはこれをシングルに限定します．これは，以下のような PyTorch 公式ウェ

ブサイトの warning に依ります．ここで，pin_memory とは，ガベージコレクター（動的確保したメモリを解放しようとするシステム）に掃除させないように取得したデータをメモリ上に固定（pinned）するための記述です．

> It is generally not recommended to return CUDA tensors in multi-process loading because of many subtleties in using CUDA and sharing CUDA tensors in multiprocessing (see CUDA in multiprocessing). Instead, we recommend using automatic memory pinning (i.e., setting pin_memory=True), which enables fast data transfer to CUDA-enabled GPUs.

27から29行目の記述はデータを読み込むためのシステム，データローダーを定義するためのものです．

```
trainloader = torch.utils.data.DataLoader(train_dataset, batch_size=MINIBATCHSIZE, shuffle=True, **k
validloader = torch.utils.data.DataLoader(valid_dataset, batch_size=len(valid_dataset), shuffle=Fals
testloader = torch.utils.data.DataLoader(test_dataset, batch_size=len(test_dataset), shuffle=False,
```

32行目の記述はこのプログラムで学習させるネットワーク（MLP）を生成させるための記述です．Network() にて生成した MLP を device に送っています．

```
model = Network().to(device)
```

これの本体は79から92行目に書かれているクラスです．

```
class Network(nn.Module):
    def __init__(self):
        super(Network, self).__init__()
        self.fc1 = nn.Linear(28*28, 128)
        self.fc2 = nn.Linear(128, 10)
        self.dropout = nn.Dropout2d(0.5)
    def forward(self, x):
        x = torch.flatten(x, 1)
        x = self.fc1(x)
        x = F.relu(x)
        x = self.dropout(x)
        x = self.fc2(x)
        output = F.log_softmax(x, dim=1)
        return output
```

self.fc1 は全結合層で，784次元（28*28）のベクトルを入力に128次元のベクトルを出力するための記述です（実際は784行128列の行列）． self.fc2 は128次元のベクトルを10次元のベクトルに変換す

るための記述です．MNIST のクラスが0から9の10クラスからなるためです．これらの結合は nn.Linear によって生成されますが，これはプログラム中，4行目でインポートされているものです．また， self.dropout にてドロップアウトを設定しています．ドロップアウトとはニューラルネットワークの正則化（過学習を抑制する仕組み）のひとつです．このように， def __init__(self): 以下に設定したいネットワークで用いるパラメータの記述を行います．

次に， def forward(self, x): 以下にネットワークの繋がり，順伝播の計算を記述します．入力は x です．これを1次元のベクトルに変換後，これに対して，上で定義した行列を作用させるのが x = self.fc1(x) の記述です．さらに下の行で線形正規化関数（ReLU）を作用させています．これは活性化関数の一種であり，ニューラルネットワークに非線形な関係を表現する能力を与えるものです．その後，ドロップアウトを効かせ，さらに次の全結合層の計算を行い，最終的にソフトマックス関数を作用させます．ソフトマックス関数とは，出力ベクトルの各要素の和が「1」でかつ，各要素の値域が [0,1] になるようにそのベクトルの値を変換させる関数です．これをすることで，ベクトルの各要素を確率として解釈することができるようになります．

```
[0.1, 0.5, 0.0, 0.1, 0.1, 0.0, 0.0, 0.1, 0.1, 0.0,]
```

このようなベクトルは合計が「1」です．このうち最大の値を示す要素は2番目の要素です．MNIST の予測をした場合，このような出力が返ってきたなら，その時の予測の数字は「1」であると出力されるわけです（2ではなくて1なのは，1番目の要素には数字の「0」が紐付いているから）．

33行目の記述は勾配降下法の最適化法を決める記述です．ここで設定している Adam という最適化法は，学習開始からの勾配情報を記録してそれらに対する平均値と分散の値を計算します．分散が大きい（ことが予想される）極小値付近では更新値を小さく，そうでないときにはグングン更新するという仕組みで最適値へと到達する速さを高めている方法です．

36行目のループが学習プロセス，学習ループと呼ばれるプロセスです．以下の記述はモデルをトレーニングモードにするためのものです．正則化，ドロップアウトが働きます．

```
    model.train() # training mode
```

データローダーは以下のように用いることで，データを一単位ずつ読み出すことができます．

```
    for input, target in trainloader:
```

以下の記述は勾配を初期化するための記述です．

```
        optimizer.zero_grad()
```

以下の記述でコスト（勾配を計算する対象）を計算して，それに対する勾配を計算します．

```
            output = model(input)
            traincost = F.nll_loss(output, target)
            traincost.backward()
```

以下の記述が勾配降下法の更新式に相当する部分です．

```
            optimizer.step()
```

50から59行目はバリデーションに関する記述です．ドロップアウトを起動させないために model.eval() を設定し，また，パラメータ更新をしないため with torch.no_grad(): を設定します．最後に60行目で学習の過程を出力します．
62行目から73行目までの記述はテストセットにおける性能評価のためのものです．本来は，この計算を学習を行うためのプログラムと同じプログラムとして同居させることは行儀が良くないことであると考えられます．普通，ハイパーパラメータを変化させながら色々な予測器を構築し，そのバリデーションの結果を見ながら最終的な予測器を作り上げるのですが，そこにテストセットでの結果が見えているなら，そのテストセットはバリデーションセットと思われても仕方ないような．．．

main() の最終行の以下の記述は現在のパラメータとアーキテクチャの情報を保存するための記述です．これが機械学習の最終産物である予測器の本体です．

```
    torch.save(model.state_dict(), "mnist_mlp.pt")
```

別のファイルからこれを読み出して，アーキテクチャを再構成し，予測器として利用することができます．

以上で PyTorch による MLP の実装は終了ですが，PyTorch は非常に拡張性が高く，上で定義したクラスを少し変更するだけで，CNN 等の複雑なネットワークを一瞬で実装することができます．例えば以下のようにクラスを改変することで畳み込み層がふたつ，プーリング層がひとつ，全結合層がふたつからなる CNN を構築することができます．

3-3-3. アーキテクチャの改変

```
#!/usr/bin/env python3

import torch
torch.manual_seed(0)
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
import torchvision

def main():
```

```
def main():
    # Hyperparameter setting
    MINIBATCHSIZE = 1000
    TOTALEPOCH = 10
    USECUDA = False

    # Dataset preparation
    learn_dataset = torchvision.datasets.MNIST(root="dataset", train=True, download=True, transform=torchvi
    test_dataset = torchvision.datasets.MNIST(root="dataset", train=False, download=True, transform=torchvi

    # Making training and validation dataset from the learning dataset
    train_dataset, valid_dataset = torch.utils.data.random_split(learn_dataset, [int(len(learn_dataset) * 0

    # Usage of CPU or GPU
    device = torch.device("cuda" if USECUDA else "cpu")
    kwargs = {'num_workers': 1, 'pin_memory': True} if USECUDA else {}

    # Definition of data loader
    trainloader = torch.utils.data.DataLoader(train_dataset, batch_size=MINIBATCHSIZE, shuffle=True, **kwar
    validloader = torch.utils.data.DataLoader(valid_dataset, batch_size=len(valid_dataset), shuffle=False,
    testloader = torch.utils.data.DataLoader(test_dataset, batch_size=len(test_dataset), shuffle=False, **k

    # Generating a model
    model = Network().to(device)
    optimizer = optim.Adam(model.parameters())

    # Learning process
    for epoch in range(1, TOTALEPOCH + 1):
        # Training
        model.train() # training mode
        traincostsum, minibatchnumber = 0, 0
        for input, target in trainloader:
            input, target = input.to(device), target.to(device)
            optimizer.zero_grad()
            output = model(input)
            traincost = F.nll_loss(output, target)
            traincost.backward()
            optimizer.step()
            traincostsum += traincost.item()
            minibatchnumber += 1
        traincostavg = traincostsum / minibatchnumber
        # Validation
        model.eval() # evaluation mode to stop dropout
        validcost, correct = 0, 0
        with torch.no_grad():
            for input, target in validloader:
                input, target = input.to(device), target.to(device)
                output = model(input)
                validcost = F.nll_loss(output, target).item()
                prediction = output.argmax(dim=1, keepdim=True)
                correct = prediction.eq(target.view_as(prediction)).sum().item()
        print("Epoch {:5d}: Training cost= {:.4f}, Validation cost= {:.4f}, Validation ACC= {:.4f}".format(

    # Test process (It should not be included in the program of learning)
```

```
    model.eval() # evaluation mode to stop dropout
    testcost, correct = 0, 0
    with torch.no_grad():
        for input, target in testloader:
            input, target = input.to(device), target.to(device)
            output = model(input)
            testcost += F.nll_loss(output, target).item()
            prediction = output.argmax(dim=1, keepdim=True)
            correct += prediction.eq(target.view_as(prediction)).sum().item()
    testcost /= len(testloader.dataset)
    print("Test cost= {:.4f}, Test ACC: {:.4f}".format(testcost, correct / len(testloader.dataset)))

    # Saving parameter
    torch.save(model.state_dict(), "mnist_cnn.pt")

class Network(nn.Module):
    def __init__(self):
        super(Network, self).__init__()
        self.conv1 = nn.Conv2d(1, 32, 3, 1)
        self.conv2 = nn.Conv2d(32, 64, 3, 1)
        self.fc1 = nn.Linear(9216, 128)
        self.fc2 = nn.Linear(128, 10)
        self.dropout = nn.Dropout2d(0.5)
    def forward(self, x):
        x = self.conv1(x)
        x = F.relu(x)
        x = self.conv2(x)
        x = F.max_pool2d(x, 2)
        x = self.dropout(x)
        x = torch.flatten(x, 1)
        x = self.fc1(x)
        x = F.relu(x)
        x = self.dropout(x)
        x = self.fc2(x)
        output = F.log_softmax(x, dim=1)
        return output

if __name__ == "__main__":
    main()
```