

Computational Materials Science (計算材料学特論)

Lecture materials updated (this morning, 8:24)

<http://conf.msl.titech.ac.jp/Lecture/ComputationalMaterialsScience/index-numericalanalysis.html>

2024年度Q2 計算材料学特論 (資料: 英語 + 日本語版)

Computational Materials Science 2023 Q2

数値解析に関する講義資料・pythonプログラム (神谷担当分)

Lecture materials on numerical analysis (by Kamiya)

講義で使うプレゼン資料は、Other related programsの下にあります

Lecture presentation slides will be found after the python tips section.

Update News:

- June 11, 8:24 Lecture materials on June 11 have been updated. ([20240611ComputerAndErrorSources.zip](#))
- June 07, 9:23 Lecture materials on June 11 have been uploaded.

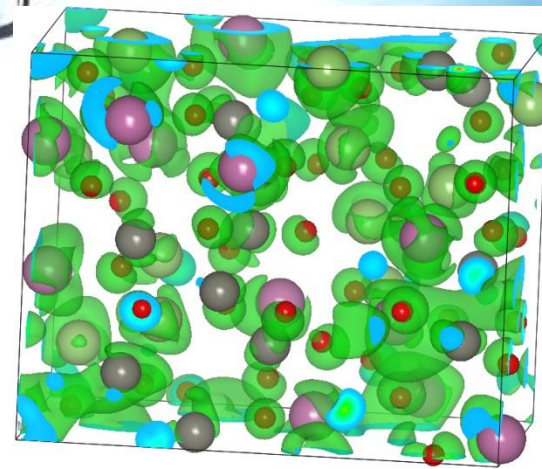
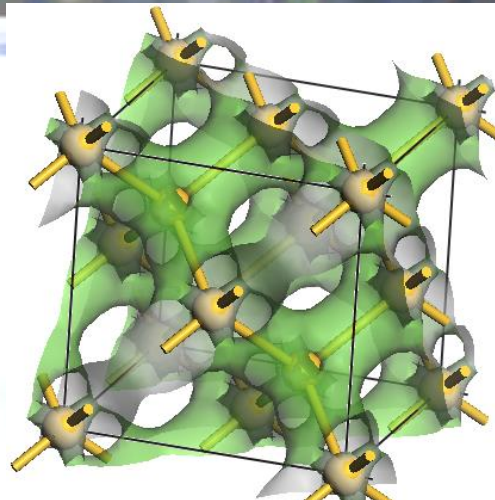
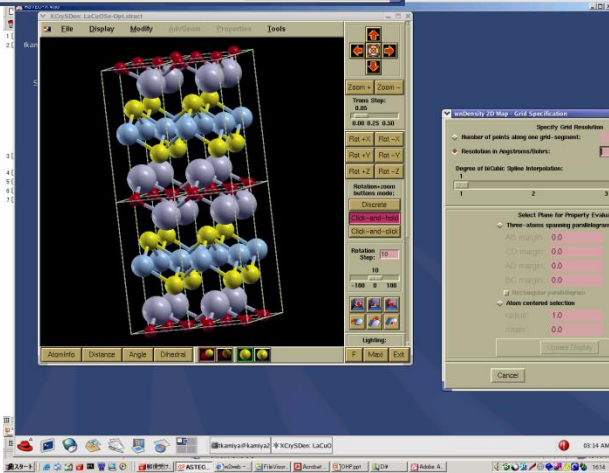
python ノート

- [起動と対話モード](#)
- [基本変数型](#)
- [変数型\(上級者向\)](#)

神

今日

神谷利夫



Class Schedule

Lecture materials (Kamiya's part): <http://conf.msl.titech.ac.jp/Lecture/>

<http://conf.msl.titech.ac.jp/Lecture/ComputationalMaterialsScience/index-numericalanalysis.html>

- #01 June 11 (Tue) Kamiya (Fundamental of computer, Sources of errors (コンピュータの基礎、誤差))
- #02 June 14 (Fri) Kamiya (Numerical differentiation/integration (数値微分/積分),
Differential equation (微分方程式))
- #03 June 18 (Tue) Kamiya (Differential equation (微分方程式), Molecular dynamics (分子動力学法),
Interpolation (補間), Smoothing (平滑化))
- #04 June 21 (Fri) Kamiya (Linear least-squares method (線形最小二乗法), Optimization (最適化),
Numerical solutions of equations (方程式の数値解法), Nonlinear optimization (非線形最適化))
- #05 June 25 (Tue) Kamiya (Nonlinear optimization (非線形最適化),
Fourier transformation (フーリエ変換))
- #06 June 28 (Fri) Kamiya, Matrix (行列)
- July 2 (Tue) No lecture (休講)**
- #07 July 5 (Fri) Kamiya, Review (復習)
- #08 July 9 (Tue) Sasagawa (Review of quantum theory 1: 量子論おさらい1)
- #09 July 12 (Fri) Sasagawa (Review of quantum theory 2: 量子論おさらい2)
- #10 July 16 (Tue) Sasagawa (First principles calculations: basics 1 第一原理計算: 基礎1)
- #11 July 19 (Fri) Sasagawa (First principles calculations: basics 2 第一原理計算: 基礎2)
- #12 July 23 (Tue) Sasagawa (First principles calc.: applications 1 第一原理計算: 応用1)
- #13 July 26 (Tue) Sasagawa (First principles calc.: applications 2 第一原理計算: 応用2)
- #14 Sasagawa (Classical and Quantum Computers 古典および量子コンピュータ)

English textbooks

Search by ‘numerical analysis’, ‘numerical simulation’, ‘数值解析’ etc.

1. *Introduction to Applied Numerical Analysis*

Richard W. Hamming

Dover publications, inc., New York (1989)

~340 pages

2. *A First Course in Numerical Analysis*

Anthony Ralston and Philip Rabinowitz

Dover publications, inc., New York (1978)

~600 pages

For practical programming: Numerical Recipes series

1. **Numerical Recipes in C**

2. **Numerical Recipes Example Book (FORTRAN)**

3. **Numerical Recipes Source Code**

Second Edition: C, Fortran77, Fortran 90

Third Edition: C++

Policy

Evaluation:

- 1. Assignment is given in each class**
- 2. Term-end assignment**

You can use AI like ChatGPT, but your answers must include your thought and improvements.

Absence of class

- 1. If you can't join a class, let me know prior to the class.**

Zoom record

- 1. Classes will be recorded, used only for students who request watching it.**

Numerical analysis web

<http://conf.msl.titech.ac.jp/Lecture/ComputationalMaterialsScience/index-numericalanalysis.html>

2024年度Q2 計算材料学特論 (資料: 英語 + 日本語版)

Computational Materials Science 2023 Q2

数値解析に関する講義資料・pythonプログラム (神谷担当分)

Lecture materials on numerical analysis (by Kamiya)

講義で使うプレゼン資料は、Other related programsの下にあります

Lecture presentation slides will be found after the python tips section.

Update News:

- June 07, 9:23 Lecture materials on June 11 have been uploaded

python ノート

- [起動と対話モード](#)
- [基本変数型](#)
- [変数型\(上級者向\)](#)
オブジェクト、クラス、インスタンス
値、ポインタ、参照
イミュータブル、ミュータブル
- [Tips 基礎編](#)
- [pythonによる最小二乗法・最適化問題 GUIプログラミング \(Japanese\)](#)
- [Tips 開発者向け](#)

Other related programs

- [D2MatE拠点開発プログラム \(Japanese\)](#)

機械学習 数値解析のpythonプログラムをGUIインターフェースで配布

Note: Getting Started with python

python is not a requirement for this class, but it will help your understanding about the algorithms to be learned and also assist your future research.

注: pythonプログラミングを始める前に

本講義では、pythonは必須ではありませんが、アルゴリズムの理解と今後の研究に役に立ちますので、余裕のある人は試してみてください。

Other open programs [Japanese]

http://conf.msl.titech.ac.jp/D2MatE/D2MatE_programs.html

- [Top page](#)
- [共通ファイル・単位など](#)
[神谷・片瀬研共通単位表](#)
- [標準ディレクトリ構成](#)
- [Launcherプログラミング](#)
- ▶ [tkProg python ライブラリ: tklib](#)
- [データ読み込みplugin仕様](#)
[plug-inリスト](#)
[tklib.tkfilter](#)
- [plotly-Dash-Flaskメモ](#)
- [pythonでGPIB制御メモ](#)
- [Excel=>python移植支援](#)
- [SILVACO ATLAS tips](#)
- ▶ [python Tips](#)
- ▶ [python 高度な内容](#)
- ▶ [プログラミングメモ](#)
- ▶ [Linuxサーバ設定](#)
- ▶ [神谷・片瀬研メモ](#)
- ▶ [講義資料](#)
- [D2MatE拠点固定ページ](#)
- [更新履歴](#)

公開プログラム

ラウンチャプログラム Launcher.py から各プログラムを実行できるように パッケージを配布しています

・パッケージ記号

A: 一般(all)
C: 神谷・片瀬研

D: D²MatE

・その他記号

pl: perlが必要
L: Linuxのみ動作保証

注: インストールトラブル、エラーやバグを見つけた場合は、[このページ](#)に従って報告をお願いします。

・インストール方法

1. [python](#)
2. [pythonモジュール](#)
3. [tkProg](#)
4. [その他のモジュール](#)
5. [PHYSBOモジュールがインストールされなかった場合](#)

(必要な人のみ)

智慧とデータが拓くエレクトロニクス新材料開発拠点 公開・非公開プログラム情報 (Data Driven Materials Research Institute for Electronics)

質問、要望、バグ報告などの連絡先: 神谷 利夫 tkamiya@msl.tech.ac.jp
東京工業大学 [国際先駆研究センター](#) [元素戦略MDX研究センター](#) 教授

News (ユーザ)

- **New!** 2024/05/17 17:29 [チュートリアル: 第一原理計算で何がわかるか](#) の録画を公開しました
- 2024/05/17 16:50 5/17講義資料を更新しました。チュートリアル: [第一原理計算で何がわかるか](#)
- 2024/05/12 14:30 [チュートリアル: 実空間像から理解するバンド理論](#) の録画を公開しました
- 2024/03/15 10:14 Excelの複雑な数式をpythonに移植する[支援プログラム](#)を更新しました
- 2024/01/24 17:14 2024/1/24 [チュートリアル: 学生と教員のためのpythonとChatGPT活用法](#) の講義資料を最終版に更新しました
- 2023/9/26 12:02 一般向けパッケージを更新しました。
 - ・ GPIB、filterの追加
 - ・ spectrum => decay.pyの更新

Launcher.pyを起動したら、"External programs/外部プログラム" メニューの "Install modules" ボタンをクリックすると、必要なモジュールをすべてインストールします。インストールするモジュールは以下の通りです。

numpy scipy matplotlib seaborn scikit-learn chardet openpyxl pandas pymatgen
physbo
pygments msoffcrypto python-pptx

News (開発者)

▶ **New!** 詳細

▶ チュートリアル・講演会等 (録画、資料を公開しているものあり)

- 材料計算科学・データ解析チュートリアルコース 2024年度
2024/05/17 [チュートリアル: 第一原理計算で何がわかるか](#)
2024/05/10 [チュートリアル: 実空間像から理解するバンド理論](#)
- 材料計算科学・データ解析チュートリアルコース 2023年度
2024/2/8 放射光を用いた薄膜材料構造解析 (D2MatE拠点内部限定)
2024/1/24 [チュートリアル: 学生と教員のためのpythonとChatGPT活用法](#)
2023/12/27 [VASP6機械学習利用の計算加速法に関するセミナー](#)
2023/7/31 Notion活用事例紹介 (D2MatE拠点内部限定)
2023/7/28 [東工大関係者向けチュートリアル: 「どのように第一原理バンド計算の条件を決めるか」](#)
2023/6/5 [MATLANTIS紹介講演会](#)
- 2023/12/1 [チュートリアル: 光電デバイスの原理と評価](#)
- [材料計算科学・データ解析チュートリアルコース 2022年度](#)

Python: A Light Weight Language (LWL)

Install: <http://conf.msl.titech.ac.jp/Lecture/InstallPython/InstallPython.html>

- **Interpreter language** (インタプリタ言語 – 逐次解釈)
 ⇔ Compiled language (コンパイル言語 – 機械語翻訳)
 Slower execution, but faster development
- Only **interpreter** and **editor** are required
- Free or public domain versions available
- Grammar similar to C, C++, perl, php, ...
- Native **Object-Oriented** (オブジェクト指向) language
- Efficient functions and libraries
 - Text processing: **Regular expression** (正規表現),
 csv, html, xml, json etc
 - Science: numpy, scipy, scikit-learn** etc
 - Network: ...
 - Graph plotting: matplotlib etc
 - GUI: tkinter, pygtk etc

Editor vs Word processor

| | Editor | Word processor |
|-------------------------|--|---|
| Startup time (起動時間) | Shorter | Longer |
| Processing speed (実行速度) | Faster | Slower |
| Memory | Light | Heavy |
| Text style / format | Usually none | Required |
| File format | Basically text-based | Application specific |
| Others | Specialized for specific program languages. Macro (small program languages) | Print (WYSIWYG): What You See is What You Get |
| Examples | Linux : vi, emacs Windows: TeraPad, Sakura Editor Multi : Visual Studio Code, Sublime text, Atom | MS-Word |

Recommendation:

Microsoft Visual Studio Code: <https://code.visualstudio.com/>

- Multiplatform (Windows, MacOS, Linux)
- Multilanguage
- Integrated Development Editor (IDE)

If interested in python and AI-based programming

<http://conf.msl.titech.ac.jp/D2MatE/2023Tutorial/tutorial2023-python-ChatGPT.html> [Japanese]b

2023年度 チュートリアル

主催: D²MatE

共催:

元素戦略MDX研究センター 講演会
第172回フロンティア材料研究所学術講演会
出島コンソーシアム・チュートリアル講座
物質・情報卓越教育院講演会

チュートリアル題目: チュートリアル: 学生と教員のためのpythonとChatGPT活用法

開催日時: 2024年1月24日(水) 15:00~16:30 (この時間後も個別の質問を受け付けます)

開催方法: Zoom Webinar

参加対象: 制限はありません

参加費: 無料

事前登録 (修了しました):

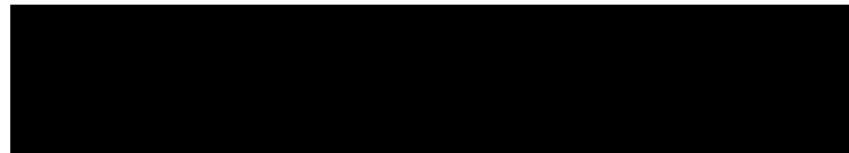
プログラミング・コンピュータ環境:

講義中には時間がないので、実習は行いません。

興味のある方は、python (numpy, scipy, openpyxl, pandas), Visual Studio Code (python plug-in推奨), JupyterおよびChatGPTが使える環境があれば、講義中にも確認できると思います。

講義資料: [python-tutorial2023-V5.zip](#) (2024/1/24 17:12)

録画:



PROBLEM, June 11

- **Submit electronic file(s) via T2SCHOLAR until the midnight of June 12**
(If T2SCHOLAR doesn't work, send the files to kamiya.t.aa@m.titech.ac.jp.
In this case, file name must include your **STUDENT ID** and **FULL NAME**)

Choose one of the following PROBLEM 1 or PROBLEM 2

PROBLEM 1:

- (i) Convert 110011_2 to base 10
- (ii) Convert 5323_{10} to base 16

PROBLEM 2:

Choose one of the python programs given today (sum_error.py, sum.py, base.py).

- Explain what each block of the source code does,
- or
- list up the source code parts that you cannot understand what they do or why they are needed.

今日配布したプログラム (sum_error-plt.py, sum.py, base.py) から1つを選び、
以下のいずれかを答えよ

- ソースコードのそれぞれの部分が何をしているかを説明する
- ソースコードの中で理解できない部分、あるいは なぜそれが必要かわからない部分を述べよ

Fundamental of computer

コンピュータの基礎

Numeric representation

(数の表現)

Base 10 $1975 = 1 \times 1000 + 9 \times 100 + 7 \times 10 + 5 \times 1$
(decimal) $= 1 \times 10^3 + 9 \times 10^2 + 7 \times 10^1 + 5 \times 10^0$
(10進数) the 1000's place
 (1000の位)

All data in computer are represented by **0 or 1** (binary) : **bit (b)**

Base 2 $(11011)_2 = 1 \times 2^4 + 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0$
(binary) $= 1 \times (16)_{10} + 1 \times (8)_{10} + 0 \times (4)_{10} + 1 \times (2)_{10} + 1 \times (1)_{10}$
(2進数) $= (27)_{10}$

Base r $N = a_n r^n + a_{n-1} r^{n-1} + \cdots + a_3 r^3 + a_2 r^2 + a_1 r^1 + a_0 r^0$
(r 進数) $= (a_n a_{n-1} \cdots a_3 a_2 a_1 a_0)_r$

Numeric representation

(数の表現)

Base 8 (octal) (8進数)

(01234567)

2 digits: $0 \sim 8^2 - 1 = 63$

$$\mathbf{00: } 0 \times 8^1 + 0 \times 8^0 = 0$$

$$\mathbf{53: } 5 \times 8^1 + 3 \times 8^0 = 43$$

$$\mathbf{77: } 7 \times 8^1 + 7 \times 8^0 = 63$$

Base 16 (hexadecimal) (16進数)

(0123456789ABCDEF) = (0 ~ 15)

2 digits: $0 \sim 16^2 - 1 = 255$

$$\mathbf{00: } 0 \times 16^1 + 0 \times 16^0 = 0$$

$$\mathbf{9F: } 9 \times 16^1 + 15 \times 16^0 = 159$$

$$\mathbf{FF: } 15 \times 16^1 + 15 \times 16^0 = 255$$

(ABCDEFGHIJKLMNOPQRSTUVWXYZ

abcdefghijklmnopqrstuvwxyz

0123456789+/-) = (0 ~ 63)

Correspondence relations (対応関係)

| Base 10 | Base 2 | Base 8 | Base 16 |
|---------|--------|--------|---------|
| 0 | 0000 | 00 | 0 |
| 1 | 0001 | 01 | 1 |
| 2 | 0010 | 02 | 2 |
| 3 | 0011 | 03 | 3 |
| 4 | 0100 | 04 | 4 |
| 5 | 0101 | 05 | 5 |
| 6 | 0110 | 06 | 6 |
| 7 | 0111 | 07 | 7 |
| 8 | 1000 | 10 | 8 |
| 9 | 1001 | 11 | 9 |
| 10 | 1010 | 12 | A |
| 11 | 1011 | 13 | B |
| 12 | 1100 | 14 | C |
| 13 | 1101 | 15 | D |
| 14 | 1110 | 16 | E |
| 15 | 1111 | 17 | F |
| 16 | 10000 | 20 | 10 |

Convert Base (基数の変換)

Base r to Base 10

$$N_r = (a_n a_{n-1} \cdots a_3 a_2 a_1 a_0)_r$$

$$N_{10} = a_0 r^0 + a_1 r^1 + a_2 r^2 + a_3 r^3 + \cdots + a_{n-1} r^{n-1} + a_n r^n$$

$$\text{Ex. } 1101_2 = 1 \times 2^0 + 0 \times 2^1 + 1 \times 2^2 + 1 \times 2^3 = 13_{10}$$

Base 10 to Base r

$$N_{10} = (b_n b_{n-1} \cdots b_3 b_2 b_1 b_0)_{10} = (c_n c_{n-1} \cdots c_2 c_1 c_0)_r$$

$$= c_0 r^0 + c_1 r^1 + c_2 r^2 + \cdots + c_{n-1} r^{n-1} + c_n r^n$$

$$= c_0 + r(c_1 + c_2 r^1 + \cdots + c_{n-1} r^{n-2} + c_n r^{n-1})$$

$$= c_0 + r(c_1 + r(c_2 + c_3 r + \cdots + c_{n-1} r^{n-3} + c_n r^{n-2})) =$$

$$\underbrace{N_{10}^{(1)}}_{N_{10}^{(2)}}$$

$$(1) N_{10}^{(0)} = N_{10} = N_{10}^{(1)} * r + c_0 \quad \text{where } 0 \leq c_0 < r$$

$$(2) N_{10}^{(1)} = N_{10}^{(2)} * r + c_1 \quad \text{where } 0 \leq c_1 < r$$

... repeat until $N_{10}^{(n+1)} = 0$

$$\Rightarrow N_r = (c_n c_{n-1} \cdots c_2 c_1 c_0)_r$$

Ex. Base 10 to Base 8

$$302_{10} = 8 \times 37 + 6$$

$$37_{10} = 8 \times 4 + 5$$

$$4_{10} = 8 \times 0 + 4$$

$$302_{10} = 456_8$$

Python program: base.py

Program: base.py

Usage: python base.py value base_source base_target

Ex.

COMMAND:

python base.py FA 16 8

Convert FA in base 16 to base 8

OUTPUT:

Convert FA in base 16 to base 10

1st digit = 10: $+ 10 * 16^0 \Rightarrow + 10_{10} \Rightarrow 10_{10}$

2nd digit = 15: $+ 15 * 16^1 \Rightarrow + 240_{10} \Rightarrow 250_{10}$

Convert 250 in base 10 to base 8

$250_{10} = 31 * 8 + 2$: base_8 $\Rightarrow 2$

$31_{10} = 3 * 8 + 7$: base_8 $\Rightarrow 72$

$3_{10} = 0 * 8 + 3$: base_8 $\Rightarrow 372_8$ result

Units of data processed in computers

(コンピュータ内のデータ単位)

bit (b): binary: **0 or 1**

In computer: **8 bits data** is treated as a fundamental unit

byte (B): $0 \sim 2^8 - 1 = 255$

$$1 \text{ kB} = 2^{10} \text{ B} = 1,024 \text{ B}$$

$$1 \text{ MB} = 1024 \text{ kB} = 1,048,576 \text{ B}$$

$$1 \text{ TB} = 1024 \text{ GB} = 1024^2 \text{ MB} = 1024^3 \text{ kB} = 1024^4 \text{ B}$$

Numeric representation: Integer (整数型)

Integer type: Based on the CPU bit (CPUのbit数が基本)

16bit for 16bit CPU

unsigned int (符号無し整数型) $0 \sim 2^{16} - 1 = 65,535$

signed int (符号付き整数型) $-32,768 \sim +32,767$

32bit for 32bit CPU

unsigned int (符号無し整数型) $0 \sim 4,294,967,295$

signed int (符号付き整数型) $-2,147,483,648 \sim +2,147,483,647$

For all CPUs:

int : depends on CPU bits

short int : 16 bit

long int : 32 bit

long long int: 64 bit

Numeric representation: Floating point, Real

(浮動小数点型, 実数)

Floating point type: Minimum 32bit (except half precision)

The range of available value depends on computer architectures, programming language etc.

C language (C言語)

float : 32 bit $3.4\text{E}-38 \sim 3.4\text{E}+38$
double : 64 bit $1.7\text{E}-308 \sim 1.7\text{E}+308$
long double: 64 bit

Fortran

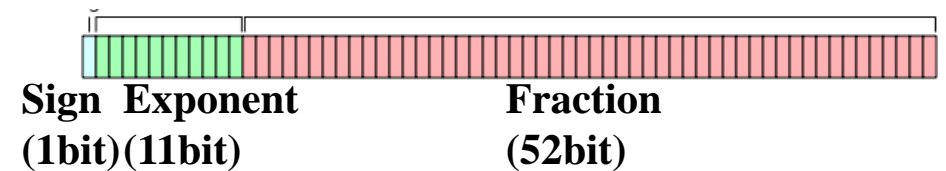
Single precision (単精度) FP (REAL) : 32 bit
Double precision (倍精度) FP (DOUBLE) : 64 bit, 16 digits (桁) in decimal
Quadruple precision (4倍精度) FP (REAL*16) : 128 bit

Definition of IEEE 754 (binary32, binary64):

Sign : 1 bit
Exponent: 8 bits (REAL, $-128 \sim +127$) 11 bits (DOUBLE, $-1024 \sim +1023$)
Fraction : 23 bits (REAL) 52 bits (DOUBLE)
8,388,608: 7 digits 4,503,599,627,370,495: 16 digits

$$\begin{array}{c} \text{Sign} \\ \text{(符号)} \end{array} \text{--} \text{1.} \begin{array}{c} \text{Fraction} \\ \text{(仮数部)} \end{array} \text{2} \times 2^{\begin{array}{c} \text{Exponent} \\ \text{(指数部)} \end{array}}$$

The diagram shows the IEEE 754 floating-point format. The sign is a single bit (blue). The fraction is the part after the decimal point (red). The exponent is the power of 2 (green). The fraction is also referred to as the mantissa (仮数部).



Required variable sizes: Integer types

unsigned int (16 bit): 65,536

16 bit CPU can handle only 64 kB of memory
(アドレスバスが16bitだと、64 kBのメモリーしか扱えない)

unsigned int (32 bit): 4,294,967,295

32bit CPU can handle 4 GB memory
(アドレスバスが32bitだと、4 GBのメモリーを扱える)

GDP of Japan: ~5 trillion US\$ = 500,000,000,000,000 JYen
(requires **16 digits**)

cf. unsigned long long int (64 bit): ~1.8E+19 (**18 digits**)

The ratio of the circumference of a circle (円周率):

Significant figure: **50 trillion digits (as of Jan, 2020)**

Need to use multi-fold calculation (多倍長計算)

Implemented based on software

Required sizes: FP types for quantum calculations

1s orbital energy level:

H atom : 13.6 eV

heavy atoms: >> keV

Energies related to physical properties

Thermal energy at room temperature: 26 meV

Magnetism: several meV

Quantum simulations of physical properties require the precision for the meV – MeV range (over 9 digits precision)

Definition of standard FP: IEEE 754

| | | |
|---------------------------|-----------|----------|
| Fraction: 23 bit (single) | 8,388,608 | 7 digits |
|---------------------------|-----------|----------|

| | | |
|--------------------------|-----------------------|-----------|
| Fraction: 52bit (double) | 4,503,599,627,370,495 | 16 digits |
|--------------------------|-----------------------|-----------|

Required sizes: FP types for semiconductor simulation

Boltzmann factor: $\exp(-E_g / k_B T)$

$$E_g = 1.1 \text{ eV}$$

$$k_B T = 0.026 \text{ eV } (T = 300 \text{ K}) \Rightarrow \exp(-42) \sim 10^{-19}$$

$$E_g = 4.0 \text{ eV}$$

$$k_B T = 0.026 \text{ eV } (T = 300 \text{ K}) \Rightarrow \exp(-154) \sim 10^{-67}$$

$$k_B T = 0.00026 \text{ eV } (T = 3 \text{ K}) \Rightarrow \exp(-15400) \sim 10^{-5141}$$

Double precision (64bit): **Fraction: 16 digits**

Exponent: $-1024 \sim +1023$ ($2^{-1024} \sim 10^{-308}$)

Quad precision : 128 bit

Octuple precision (8倍精度): 256 bit

Error of floating point variables (浮動小数点型の誤差)

Representation of floating point in computer:

$$-1. \mathbf{011101}_2 \times 2^{-\mathbf{015}_{10}} \quad (\text{in binary})$$

Errors arise from converting Base 10 to Base 2.

- Some values do not have errors between Base 10 and Base 2 if fraction equals to 2^n

$$1.0 = (1.0)_2 \times 2^0$$

$$0.5 = (1.0)_2 \times 2^{-1}$$

$$0.125 = (1.0)_2 \times 2^{-3}$$

$$0.0390625 = 1.25 \times 2^{-5} = (1.01)_2 \times 2^{-5}$$

$$1.75 = 1 \times 2^0 + 1 \times 2^{-1} + 1 \times 2^{-2} = (1.11)_2$$

$$0.65625 = 1 \times 2^{-1} + 0 \times 2^{-2} + 1 \times 2^{-3} + 0 \times 2^{-4} + 1 \times 2^{-5} = (0.10101)_2$$

$$100.0 = 1.5625 \times 64 = (1 + 2^{-1} + 2^{-4}) \times 2^6 = (1.1001)_2 \times 2^4$$

- Other values have errors

even if it is represented by a simple figure in Base 10:

$$\mathbf{0.1 = (1.1001100110011001 \cdots)_2 \times 2^{-3}}$$

Program (roundoff error): sum_error.py

Usage: python sum_error.py *h n iPrintStep*

Summing up *h* for *n* times with different precision interger types. Output every *iPrintStep* steps.

python sum_error.py 0.1 100 20

| exact: | sum16 (error) | sum32 (error) | sum64 (error) |
|---------|----------------------------------|----------------------------------|-----------------------------------|
| 0.1000: | 0.099975585937500000 (+2.44e-05) | 0.100000001490116119 (-1.49e-09) | 0.100000000000000006 (+0.00e+00) |
| 2.1000: | 2.095703125000000000 (+4.30e-03) | 2.100000143051147461 (-1.43e-07) | 2.1000000000000000533 (-4.44e-16) |
| 4.1000: | 4.089843750000000000 (+1.02e-02) | 4.099998474121093750 (+1.53e-06) | 4.1000000000000001421 (-8.88e-16) |
| 6.1000: | 6.121093750000000000 (-2.11e-02) | 6.099996566772460938 (+3.43e-06) | 6.099999999999994316 (+6.22e-15) |
| 8.1000: | 8.148437500000000000 (-4.84e-02) | 8.099994659423828125 (+5.34e-06) | 8.099999999999987210 (+1.24e-14) |

python sum_error.py 0.125 100 20

| exact: | sum16 (error) | sum32 (error) | sum64 (error) |
|---------|-----------------------------------|-----------------------------------|-----------------------------------|
| 0.1250: | 0.125000000000000000 (+0.00e+00) | 0.125000000000000000 (+0.00e+00) | 0.125000000000000000 (+0.00e+00) |
| 2.6250: | 2.625000000000000000 (+0.00e+00) | 2.625000000000000000 (+0.00e+00) | 2.625000000000000000 (+0.00e+00) |
| 5.1250: | 5.125000000000000000 (+0.00e+00) | 5.125000000000000000 (+0.00e+00) | 5.125000000000000000 (+0.00e+00) |
| 7.6250: | 7.625000000000000000 (+0.00e+00) | 7.625000000000000000 (+0.00e+00) | 7.625000000000000000 (+0.00e+00) |
| 10.125: | 10.125000000000000000 (+0.00e+00) | 10.125000000000000000 (+0.00e+00) | 10.125000000000000000 (+0.00e+00) |

python sum_error.py 0.0390625 100 20

| exact: | sum16 (error) | sum32 (error) | sum64 (error) |
|---------|----------------------------------|----------------------------------|----------------------------------|
| 0.0391: | 0.039062500000000000 (+0.00e+00) | 0.039062500000000000 (+0.00e+00) | 0.039062500000000000 (+0.00e+00) |
| 0.8203: | 0.820312500000000000 (+0.00e+00) | 0.820312500000000000 (+0.00e+00) | 0.820312500000000000 (+0.00e+00) |
| 1.6016: | 1.601562500000000000 (+0.00e+00) | 1.601562500000000000 (+0.00e+00) | 1.601562500000000000 (+0.00e+00) |
| 2.3828: | 2.382812500000000000 (+0.00e+00) | 2.382812500000000000 (+0.00e+00) | 2.382812500000000000 (+0.00e+00) |
| 3.1641: | 3.164062500000000000 (+0.00e+00) | 3.164062500000000000 (+0.00e+00) | 3.164062500000000000 (+0.00e+00) |

Roundoff error (桁落ち誤差)

Summing small value h for many times N

Calc by summation

```
x = 0.0;
for i in range(N)
    x = x + h
```

Error is accumulated by each summation

Calc by multiplication

```
x0 = 0.0;
for i in range (N)
    x = x0 + i * h
```

Typically multiplication is slower than summation, but **the total error originates from only one multiplication operation**

Result of sum_error.py (compare different precision FP types): $h = 0.01$, $N = 101$

Program: sum_error.py

| | float16: half precision, 16 bit | float32: single precision, 32 bit | float64: half precision, 64 bit |
|---------|----------------------------------|-----------------------------------|------------------------------------|
| Exact: | float16 (error) | float32 (error) | float64 (error) |
| 0.0100: | 0.010002136230468750 (-2.14e-06) | 0.009999999776482582 (+2.24e-10) | 0.010000000000000000 (+0.00e+00) |
| 0.1100: | 0.110046386718750000 (-4.64e-05) | 0.1099999984502792358 (+1.55e-08) | 0.1099999999999999987 (+1.39e-17) |
| 0.2100: | 0.210083007812500000 (-8.30e-05) | 0.2100000023245811462 (-2.32e-08) | 0.2100000000000000048 (-5.55e-17) |
| 0.3100: | 0.310058593750000000 (-5.86e-05) | 0.309999972581863403 (+2.74e-08) | 0.31000000000000000109 (-1.11e-16) |
| 0.4100: | 0.410156250000000000 (-1.56e-04) | 0.409999877214431763 (+1.23e-07) | 0.41000000000000000198 (-1.67e-16) |
| 0.5100: | 0.509765625000000000 (+2.34e-04) | 0.509999811649322510 (+1.88e-07) | 0.51000000000000000231 (-2.22e-16) |
| ... | | | |
| 0.8100: | 0.802734375000000000 (+7.27e-03) | 0.809999525547027588 (+4.74e-07) | 0.81000000000000000497 (-4.44e-16) |
| 0.9100: | 0.900390625000000000 (+9.61e-03) | 0.909999430179595947 (+5.70e-07) | 0.91000000000000000586 (-5.55e-16) |
| 1.0100: | 0.998046875000000000 (+1.20e-02) | 1.009999394416809082 (+6.06e-07) | 1.01000000000000000675 (-6.66e-16) |

Python program: sum.py

Program: sum.py

Usage: `python sum.py h N`

Summing small value `h` for many times `N`

Example command: `python sum.py 0.1 10`

OUTPUT:

0: $0.0 + 0.1 \Rightarrow 0.1$

1: $0.1 + 0.1 \Rightarrow 0.2$

2: $0.2 + 0.1 \Rightarrow 0.300000000000000000000004$

3: $0.300000000000000000000004 + 0.1 \Rightarrow 0.4$

4: $0.4 + 0.1 \Rightarrow 0.5$

...

7: $0.7 + 0.1 \Rightarrow 0.799999999999999999999999$

...

9: $0.899999999999999999999999 + 0.1 \Rightarrow 0.999999999999999999999999$

Caution for conditional branch (条件分岐における注意)

Calculation of integers does not produce errors.

=> Conditional judgement only using integers works properly

整数変数のみでの条件判断は誤差が出ないので問題ない

```
if i * 10 == 30:
```

```
    print("i == 30") # executed if i == 30
```

Calculation of floating points can produce roundoff error.

=> Strict conditional judgement often does not work properly: **Must not be used!!**

実数計算では丸め誤差が発生するので、厳格な条件判断は使ってはいけない

```
if x * 10.0 == 30.0:
```

```
    print("x == 3.0") # expected to execute if x == 3.0, but may be not
```

[Important!!] Consider possible errors:

【重要】起こりうる誤差 (epsilon: eps) を考慮した条件判断をする

```
eps = 1.0e-30 # epsilon: small, but a bit larger value than possible error
```

```
if abs(x * 10.0 - 30.0) < eps:
```

```
    print("x == 3.0") # executed if x is practically equal to 3.0
```

Program: bad_if.py

Usage: python bad_if.py h n answer

Check the condition $h * n == \text{answer}$

python bad_if.py 0.1 1 0.1 **Confirm $\sum_{i=1}^1 0.1 == 0.1$**

Summing up 0.1 for 1 times: $v = 0.1$

$v == 0.1$?: **True**

$|v - 0.1| < 1e-10$?: **True**

python bad_if.py 0.1 2 0.2 **Confirm $\sum_{i=1}^2 0.1 == 0.2$**

Summing up 0.1 for 2 times: $v = 0.2$

$v == 0.2$?: **True**

$|v - 0.2| < 1e-10$?: **True**

python bad_if.py 0.1 3 0.3 **Confirm $\sum_{i=1}^3 0.1 == 0.3$: Failed**

Summing up 0.1 for 3 times: $v = 0.30000000000000000004$

$v == 0.3$?: **False** **$\sum_{i=1}^3 0.1 == 0.3$ では判断を間違える**

$|v - 0.3| < 1e-10$?: **True** **$|\sum_{i=1}^3 0.1 - 0.3| < \text{eps}$ (eps = 10^{-10}) では正しく判断できる**

How to use conditional branch, if (条件分岐の判断)

Bad (悪い例):

if $x * 10.0 == 30.0$:

DO NOT use the strict comparison '==' for floating values
(浮動小数点の比較には、厳密な比較 == は使わない)

Good (良い例):

eps = 1.0e-30 # epsilon:

A value satisfactory smaller than minimum expected value
(想定される誤差よりも十分大きい、
なるべく小さい値を設定する)

if **$\text{abs}(x * 10.0 - 30.0) < \text{eps}$**

Program: bad_int.py

Usage: python bad_int.py *h n*

Check interger conversion of the summation of *h* for *n* times

python bad_int.py 0.1 100

Summing up 0.1 for 100 times: $v = 9.999999999999998$

$\text{int}(9.999999999999998) = 9$

10でなければいけない

$\text{int}(9.999999999999998 + 1e-10) = 10$

eps (10^{-10}) を加えてから $\text{int}()$ を取ることで**正しい解**

python bad_int.py 0.4 20

Summing up 0.4 for 20 times: $v = 8.0000000000000002$

$\text{int}(8.0000000000000002) = 8$

正しい解だが、 v には誤差があることに注意

$\text{int}(8.0000000000000002 + 1e-10) = 8$

eps (10^{-10}) を加えてから $\text{int}()$ を取っても**正しい解**

python bad_int.py 1.2 20

Summing up 1.2 for 20 times: $v = 23.999999999999993$

$\text{int}(23.999999999999993) = 23$

24でなければいけない

$\text{int}(23.999999999999993 + 1e-10) = 24$

eps (10^{-10}) を加えてから $\text{int}()$ を取ることで**正しい解**

Case for floating point to integer conversion (浮動小数点 => 整数変換):

Bad (悪い例):

$n = \text{int}(v)$

Good (良い例):

$\text{eps} = 1.0e-6$

$n = \text{int}(v + \text{eps})$

Typical cases for FP calculations with care

Evaluate possible errors every time for FP floating point calculations

- Error originates from the limited length of FP type: underflow, overflow

Representation range of 64bit FP (IEEE 754 standard)

Exponent: 11 bit -1024 ~ +1023

Fraction : 23 bit 4,503,599,627,370,495: 16 digits

- **FP type in computer cannot represent accurate values for most of integers**

$$100.0_{10} = 1.5625 \times 64 = (1 + 2^{-1} + 2^{-4}) \times 2^6 = (1.1001)_2 \times 2^4$$

- **Most of FP values in computer include errors**

$$1.0/3.0 = 0.3333...333 \text{ (16 digits)} \quad \text{Error} \sim 10^{-16} \text{ should be included}$$

Conditional branch:

Bad: if $x * 10.0 == 30.0$: **No guarantee to get the correct judge 'true'** even if $x = 3.0$

Good: $\text{eps} = 1.0\text{e-}30$ # epsilon: A value satisfactory smaller than expected values

if $\text{abs}(x * 10.0 - 30.0) < \text{eps}$: **Gives the correct judge within the error of eps**

FP => integer conversion:

How to calculate the number of division in the range xmin – xmax at xstep step

Bad: $n = \text{int}((\text{xmax} - \text{xmin}) / \text{xstep})$: The value in int() can include error.

Even if the correct value is $n = 3.0$,

you will get $n = 2$ if int() becomes 2.99999... due to error,.

Good:

$$\text{eps} = 1.0\text{e-}6$$

$$n = \text{int}((\text{xmax} - \text{xmin}) / \text{xstep} + \text{eps})$$

Even if $(\text{xmax} - \text{xmin}) / \text{xstep}$ becomes smaller than the expected integer due to error,

you can receive the correct value as long as the error is smaller than eps.

数値演算プログラムの一般的な注意

浮動小数点型の演算では、常に誤差を意識すること

- ・ 変数長の制限による誤差: underflow, overflow

IEEE 754の標準で、64bit浮動小数点の範囲は

指数部: 11 bit $-1024 \sim +1023$

仮数部: 23 bit 4,503,599,627,370,495: 16桁

- ・ 浮動小数点では、整数を“正確に”表現できない

$$100.0_{10} = 1.5625 \times 64 = (1 + 2^{-1} + 2^{-4}) \times 2^6 = (1.1001)_2 \times 2^4$$

- ・ 有限の桁数の浮動小数点の表現は、ほぼすべての場合に誤差を含む

$$1.0/3.0 = 0.3333...333 \text{ (小数点以下16桁)} \quad 10^{-16} \text{ 程度の誤差が発生する}$$

条件分岐の判断:

悪い例: `if x * 10.0 == 30.0:` `x = 3.0` であっても、**true と判断される保証はない**

良い例: `eps = 1.0e-30` # epsilon: 想定される誤差よりも十分大きい、なるべく小さい値を設定する。

`if abs(x * 10.0 - 30.0) < eps:` **誤差 eps 以内で必ず実行される**

浮動小数点 => 整数変換: `xmin ~ xmax` の範囲を `xstep` 毎の幅で分割したときの分点の数

悪い例:

`n = int((xmax - xmin) / xstep):`

`(xmax - xmin) / xstep` が誤差により `2.99999...` となった場合、

本来は `int() = 3` となって欲しいのに、`2` になってしまう

良い例:

`eps = 1.0e-6`

`n = int((xmax - xmin) / xstep + eps):`

`(xmax - xmin) / xstep` が誤差により期待する整数値より小さくなくても、

誤差が `eps` より小さければ、本来期待している整数値が得られる

Precision and errors in computer

Data bit width (データ長): Determine the upper limit of precision

=> Roundoff (rounding) error (丸め誤差)

Other error sources

- **Overflow (積み残し誤差, 桁あふれ):**

e.g. by summation between large integers (有効桁数を超える整数の和・積)

⇔ underflow

(overflow and underflow can be detected by CPU / software

but may deteriorate calculation speed)

- **Roundoff error (桁落ち誤差): By subtracting very similar values**

ex: for 4 digits calculation:

$$5\sqrt{41} - 32 \sim 5 * 6.403 - 32.00 = 32.015 - 32.00 = 32.02 - 32.00 = 0.02$$

The given values have 4 significant digits

but the result has only 1 significant digits

Avoid subtraction between similar large values

- **Loss of trailing digits (情報落ち):**

by summing / subtracting between largely-different values

ex: $1000 + 1.456 = 1001$ (The initial significant value of .456 is lost)

Errors in calculation process

- Overflow (summing large values)
- Underflow (huge numbers of summing up small values)
- Rounding off error
- **Information buried** (情報埋没)
- **Truncation error** (打ち切り誤差)

To sum up values slowly approaching to zero:

- **Taylor expansion** (テーラー展開)
- **Summation of Coulomb energy** (Coulombエネルギーの和)

Need to terminate the summation if calculation time has limitation or the result reaches the required precision

(計算時間と必要な精度に応じて、どこかで計算を打ち切る)

- **Convergence error** (収束誤差)
The required precision (often expressed EPS) is given to judge the termination of iterative convergence calculations
- **Errors originating from physical model** (物理モデルの誤差)

Information buried (情報埋没)

Program: python information_buried.py

e.g., calculate $\exp(-40)$ by $\exp(x) = \sum_{n=0} x^n/n!$

Summing up large values with
opposite signs results in
significant errors (正負が交番する
大きな数の和を取るために誤差が大きくなる)



Better to add positive values
only

$A = \sum_{n=0}^N (-x)^n/n!$ (if $x < 0$)
, and take $\frac{1}{A} = \exp(-40)$

Exact value $4.24835425529159 \times 10^{-18}$

$N : \sum_{n=0}^N x^n/n! \quad 1.0 / \sum_{n=0}^N (-x)^n/n!$

| | | |
|------|------------------|-----------------|
| 0 : | 1 | 1 |
| 1 : | -39 | 0.024390244 |
| 2 : | 761 | 0.0011890606 |
| 18 : | $7.3620174e+12$ | $5.290335e-14$ |
| 19 : | $-1.5234693e+13$ | $2.4096905e-14$ |
| 20 : | $2.9958728e+13$ | $1.153502e-14$ |
| 21 : | $-5.6123978e+13$ | $5.7878667e-15$ |
| 22 : | $1.0039003e+14$ | $3.0368438e-15$ |
| 23 : | $-1.7180825e+14$ | $1.6625449e-15$ |

Sum up large +/- values

| | | |
|------|-------------------------------|-----------------------------------|
| 79 : | $-1.3651644e+09$ | $4.2483543e-18$ |
| 115: | 5.8811462 | $4.2483543e-18$ |
| 116: | 5.8811665 | $4.2483543e-18$ |

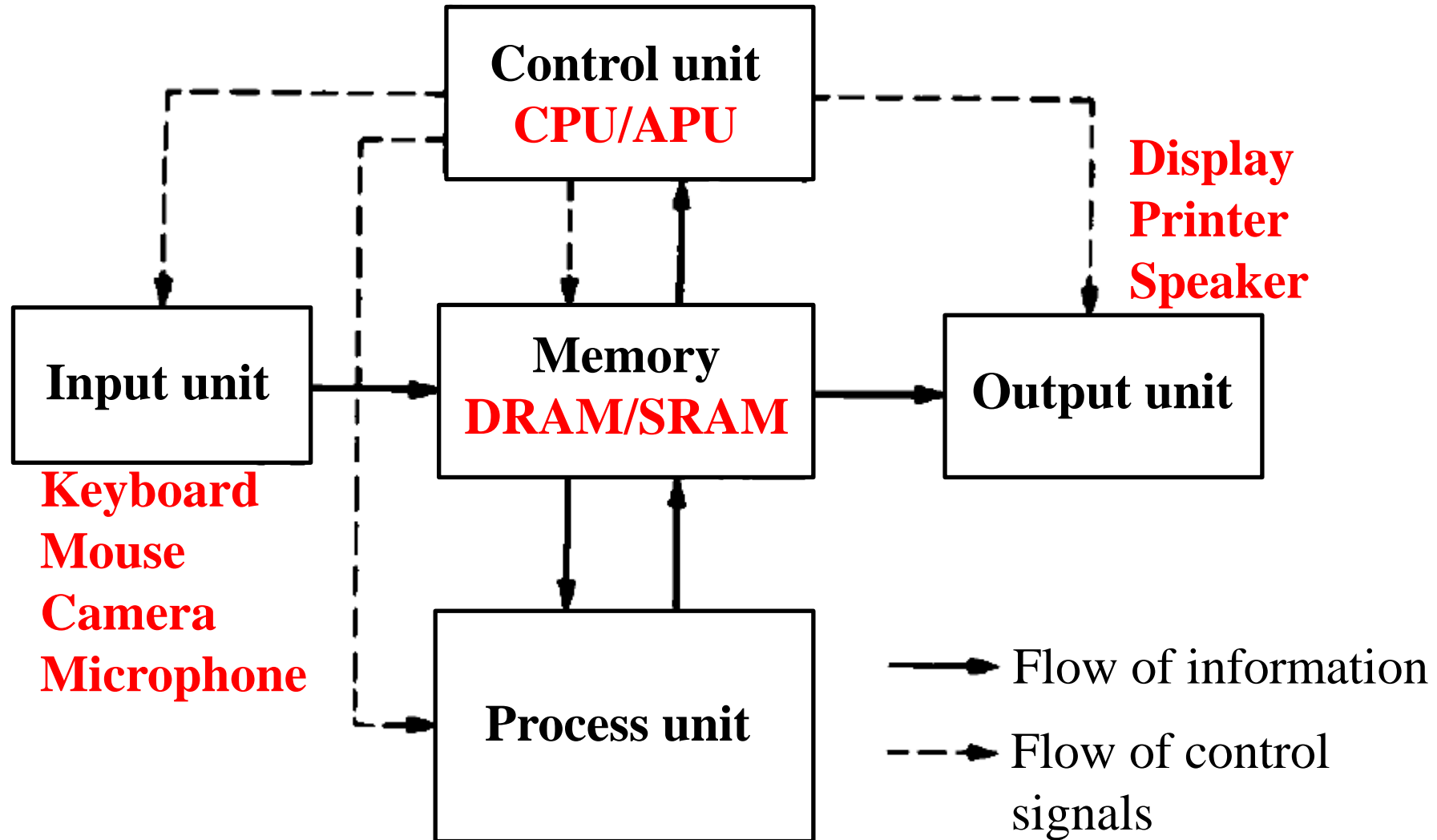
Well converged,
but 18 digits of error!!

Supplementary materials

Structure of typical computer

(基本的な計算機の構成)

大河内他、基礎 電子計算機、実教出版



Computer architectures

4bit CPU: Intel 4004 (1971) data 4bit, address 12bit
8bit CPU: 8008 (1972) data 8bit, address 14bit
16bit CPU: 8086 (1978) data 16bit, address 20bit
32bit CPU: 80386SX (1985) External data/address 32bit
Internal data 16bit, address 24bit
80486 (1989) data 32bit, address 32bit
Pentium,,,
64bit CPU: Pentium Pro(?), Itanium, Core i,, ...

Pentium Pro:

Processor 32bit: Operation (命令)・Process (データ処理) in CPU

External data bus (外部データバス)

64bit: Data transfer with memory / external units

Floating point operation (浮動小数点演算)

80bit

Logical operations (bitwise operations)

(論理演算, ビット演算)

Logical NOT (Bitwise inversion) (論理否定, ビット反転)

$$\text{NOT } 0 = 1; \text{NOT } 1 = 0$$

Logical AND (論理積)

$$0 \text{ AND } 0 = 0; 1 \text{ AND } 0 = 0$$

$$0 \text{ AND } 1 = 0; 1 \text{ AND } 1 = 1$$

Logical OR (論理和)

$$0 \text{ OR } 0 = 0; 1 \text{ OR } 0 = 1$$

$$0 \text{ OR } 1 = 1; 1 \text{ OR } 1 = 1$$

Logical Exclusive OR (排他的論理和)

$$0 \text{ XOR } 0 = 0; 1 \text{ XOR } 0 = 1$$

$$0 \text{ XOR } 1 = 1; 1 \text{ XOR } 1 = 0$$

Required data size: Character type

Alphanumeric (英数字文字):

0~9, A~Z, a~z,

Control chars (制御文字) etc

ASCII code: 7 bit (0 ~ 127)

Extended ASCII code

Add non-English chars,
symbols etc: 8 bit

Japanese

ASCII+half-width Kana (半角カナ): **8bit**

Kanji・Kana (Full-width Kana, 全角文字): **16 bit**

Shift-JIS (SJIS), JIS, EUC-JP

Universal character codes

(全世界共通文字コード)

Unicode: Started from 2 Bytes (Ver1.0.0)

Extended to 1 – 4 Bytes (UCS, Unicode / UTF-7/8/16 etc)

| 制御 文字 | 10 進 | 16 進 | 文字 | コード | 10 進 | 16 進 | 文字 | 10 進 | 16 進 | 文字 | 10 進 | 16 進 | 文字 |
|----------|------|------|----|-----|------|------|----|------|------|----|------|------|----|
| ^@ | 0 | 00 | | NUL | 32 | 20 | ! | 64 | 40 | @ | 96 | 60 | ' |
| ^A | 1 | 01 | | SOH | 33 | 21 | ! | 65 | 41 | A | 97 | 61 | a |
| ^B | 2 | 02 | | STX | 34 | 22 | .. | 66 | 42 | B | 98 | 62 | b |
| ^C | 3 | 03 | | ETX | 35 | 23 | # | 67 | 43 | C | 99 | 63 | c |
| ^D | 4 | 04 | | EOT | 36 | 24 | \$ | 68 | 44 | D | 100 | 64 | d |
| ^E | 5 | 05 | | ENQ | 37 | 25 | % | 69 | 45 | E | 101 | 65 | e |
| ^F | 6 | 06 | | ACK | 38 | 26 | & | 70 | 46 | F | 102 | 66 | f |
| ^G | 7 | 07 | | BEL | 39 | 27 | , | 71 | 47 | G | 103 | 67 | g |
| ^H | 8 | 08 | | BS | 40 | 28 | (| 72 | 48 | H | 104 | 68 | h |
| ^I | 9 | 09 | | HT | 41 | 29 |) | 73 | 49 | I | 105 | 69 | i |
| ^J | 10 | 0A | | LF | 42 | 2A | * | 74 | 4A | J | 106 | 6A | j |
| ^K | 11 | 0B | | VT | 43 | 2B | + | 75 | 4B | K | 107 | 6B | k |
| ^L | 12 | 0C | | FF | 44 | 2C | , | 76 | 4C | L | 108 | 6C | l |
| ^M | 13 | 0D | | CR | 45 | 2D | - | 77 | 4D | M | 109 | 6D | m |
| ^N | 14 | 0E | | SO | 46 | 2E | . | 78 | 4E | N | 110 | 6E | n |
| ^O | 15 | 0F | | SI | 47 | 2F | / | 79 | 4F | O | 111 | 6F | o |
| ^P | 16 | 10 | | DLE | 48 | 30 | 0 | 80 | 50 | P | 112 | 70 | p |
| ^Q | 17 | 11 | | DC1 | 49 | 31 | 1 | 81 | 51 | Q | 113 | 71 | q |
| ^R | 18 | 12 | | DC2 | 50 | 32 | 2 | 82 | 52 | R | 114 | 72 | r |
| ^S | 19 | 13 | | DC3 | 51 | 33 | 3 | 83 | 53 | S | 115 | 73 | s |
| ^T | 20 | 14 | | DC4 | 52 | 34 | 4 | 84 | 54 | T | 116 | 74 | t |
| ^U | 21 | 15 | | NAK | 53 | 35 | 5 | 85 | 55 | U | 117 | 75 | u |
| ^V | 22 | 16 | | SYN | 54 | 36 | 6 | 86 | 56 | V | 118 | 76 | v |
| ^W | 23 | 17 | | ETB | 55 | 37 | 7 | 87 | 57 | W | 119 | 77 | w |
| ^X | 24 | 18 | | CAN | 56 | 38 | 8 | 88 | 58 | X | 120 | 78 | x |
| ^Y | 25 | 19 | | EM | 57 | 39 | 9 | 89 | 59 | Y | 121 | 79 | y |
| ^Z | 26 | 1A | | SUB | 58 | 3A | : | 90 | 5A | Z | 122 | 7A | z |
| ^[| 27 | 1B | | ESC | 59 | 3B | : | 91 | 5B | [| 123 | 7B | { |
| ^\ | 28 | 1C | | FS | 60 | 3C | < | 92 | 5C | \ | 124 | 7C | |
| ^] | 29 | 1D | | GS | 61 | 3D | = | 93 | 5D |] | 125 | 7D | } |
| ^^ | 30 | 1E | ▲ | RS | 62 | 3E | > | 94 | 5E | ^ | 126 | 7E | ~ |
| ^- | 31 | 1F | ▼ | US | 63 | 3F | ? | 95 | 5F | - | 127 | 7F | ° |