半導体工学における数値計算の基礎

第 1 章 講義の概要と導入

1.1 講義の進め方と評価

本講義では、コンピュータを用いた解析やシミュレーションプログラムを理解し、開発するために必要な**数値解析（Numerical Analysis）**の基礎について解説します。私（担当教員）が前半の第 1 回から第 7 回までを担当し、コンピュータシミュレーションの基本と計算における誤差の発生源について説明します。

- **評価方法:** 期末試験は実施せず、**期末課題**を提出していただきます。
- **AI ツールの利用:** ChatGPT のような生成 AI の利用は許可しますが、提出物には**皆さん自身の考察や AI の回答からの改善点**を必ず含めてください。AI の回答をそのまま提出した場合は評価の対象外とします。
- **欠席時の対応:** 欠席する場合は事前に連絡をお願いします。各講義は録画しており、後から視聴が必要な場合はメールまたは Slack でご連絡ください。

1.2 参考資料と教科書

数値解析や数値シミュレーションに関する英語の教科書は多数存在します。「Numerical Analysis」や「Numerical Simulation」といったキーワードで検索すると見つけることができます。日本語では「数値解析」で検索してください。

実用的なアルゴリズムやプログラミングに興味がある方には、『Numerical Recipes』シリーズを強くお勧めします。これは様々なプログラミング言語版が提供されており、非常に有名な教科書です。本講義でも適宜そのエッセンスを紹介します。

1.3 開発環境の推奨

プログラム開発には、Word のようなワープロソフトは適していません。いわゆる**テキストエディタ**を使用する必要があります。特定のテキストエディタをお持ちでない場合は、**Microsoft Visual Studio Code** の使用を推奨します。

近年、生成 AI の発展は目覚ましく、非常に有用なツールとなっています。AI を活用したチュートリアルビデオや講義スライドなども多数見つかりますので、参考にしてください（ただし、多くは日本語で提供されている可能性があります）。

1.4 本日の課題

本日出題される課題は、**2 日以内**に LMS（Learning Management System）を通じて提出してください。提出期限は **6 月 11 日（水）の深夜 23 時 59 分**です。LMS にアクセスできない場合は、メールで課題ファイルを送付してください。その際、ファイル名に**学籍番号と氏名**を含めるようお願いします。

本日の課題は以下の 2 問です。

1. **基数変換**:
    - 2 進数 `101001` を 10 進数に変換してください。
    - 10 進数 `4251` を 16 進数に変換してください。 この変換方法は、本日の講義で詳しく解説します。
2. **Python プログラムの解析**:
    - 本日提供される講義資料の ZIP ファイルに含まれる Python プログラムの中から **1 つ**を選び、そのソースコードの各ブロックが何をしているのかを説明してください。
    - もし理解できない部分があれば、「この部分は理解できませんでしたが、何のために必要なのでしょうか」といった形で疑問点を挙げてください。

- プログラムの内容が全く理解できなくても、「何も理解できませんでした」という回答でも構いません。重要なのは、プログラムに触れてみることです。

第 2 章 コンピュータにおける数値表現の基礎

2.1 コンピュータの基本と 2 進数

コンピュータの CPU やメモリといったハードウェアは、基本的に**バイナリ（2 進数）論理回路**で構成されています。これは、電気信号の「オン」と「オフ」、あるいは「高電圧」と「低電圧」といった **2 つの状態**しか区別できないためです。したがって、コンピュータ内部における最も基本的な数値表現は **2 進数（Base 2）**となります。

人間にとって 2 進数は扱いにくいため、通常は **10 進数（Base 10）**や **16 進数（Base 16）**などが用いられますが、コンピュータ内部ではすべて 2 進数に変換されて処理されます。

2.2 基数変換の原理

ある基数（例えば 10 進数）で表現された数を、別の基数（例えば 2 進数）で表現し直すことを**基数変換**と呼びます。

2.2.1 R *進数から 10 進数への変換*

私たちが普段使う 10 進数は、例えば `1975` という数は以下のように表現されます。$1 \times 10^3 + 9 \times 10^2 + 7 \times 10^1 + 5 \times 10^0$ ここで「10」が**基数（Base）**です。

同様に、2 進数で表現された数も、基数を「2」とすることで 10 進数に変換できます。例えば、2 進数 `11011` は以下のようになります。 $1 \times 2^4 + 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 = 16 + 8 + 0 + 2 + 1 = 27$ したがって、2 進数 `11011` は 10 進数で `27` に対応します。

一般化された変換式: R 進数で表現された数 $(a_n a_{n-1} \dots a_1 a_0)_R$ は、10 進数にすると以下のようになります。 $(a_n a_{n-1} \dots a_1 a_0)_R = a_n \times R^n + a_{n-1} \times R^{n-1} + \dots + a_1 \times R^1 + a_0 \times R^0$

2.2.2 10 *進数から R 進数への変換*

R 進数から 10 進数への変換は上記の式で比較的容易ですが、その逆、つまり 10 進数から R 進数への変換は少し複雑です。これは**繰り返し除算**の原理を用いて行います。

例として、10 進数 374 を 8 進数（基数 8）に変換する手順を見てみましょう。

1. 374 を 8 で割ります。 `374 = 8 × 46 + 6` (余り 6) この余り 6 が 8 進数の**最下位桁（右端）**になります。
2. 商の 46 を再び 8 で割ります。 `46 = 8 × 5 + 6` (余り 6) この余り 6 が 8 進数の 2 桁目になります。
3. 商の 5 を再び 8 で割ります。 `5 = 8 × 0 + 5` (余り 5) この余り 5 が 8 進数の 3 桁目になります。商が `0` になったので計算は終了です。

結果として、余りを下から順に並べると 566 となります。 したがって、10 進数 374 は 8 進数で 566 です。

2.3 8 進数と 16 進数

2 進数は桁数が多くなりがちで、人間が直接理解するには不便です。そのため、しばしば 2 進数をまとめて表現する **8 進数（Octal）**や **16 進数（Hexadecimal）**が利用されます。

- **8 進数（基数 8）**: 1 桁で 8 種類の数字（0 から 7）を表現します。2 進数の 3 桁が 8 進数の 1 桁に対応します ($2^3 = 8$ のため)。 例: 8 進数 53 は、$5 \times 8^1 + 3 \times 8^0 = 40 + 3 = 43$ (10 進数) となります。
- **16 進数（基数 16）**: 1 桁で 16 種類の文字を表現します。0 から 9 の数字に加え、**A から F のアルファベット**が用いられます。A は 10、B は 11、…、

F は 15 に対応します。2 進数の 4 桁が 16 進数の 1 桁に対応します ($2^4 = 16$ のため)。 例: 16 進数 2F は、$2 \times 16^1 + 15 \times 16^0 = 32 + 15 = 47$ (10 進数) となります。 2 桁の 16 進数 FF は、$15 \times 16^1 + 15 \times 16^0 = 240 + 15 = 255$ (10 進数) となり、これは 8 ビットで表現できる最大値に相当します。

Base64 エンコーディング: インターネット通信などで用いられる Base64 エンコーディングは、64 種類の文字（英数字、一部記号）を用いてバイナリデータをテキストデータに変換するものです。これは特定の基数変換とは少し異なりますが、バイナリデータを人間が扱える形式に変換するという点で関連性があります。

2.3.1 *基数変換プログラムの利用*

本日提供される Python プログラム `base.py` は、様々な基数間の変換をサポートしています。 例えば、16 進数 `fa` を 8 進数に変換するには、コマンドプロンプトで以下のように実行します。 `python base.py fa 16 8` このプログラムは、まず入力値を 10 進数に変換し、その後目標の基数に変換します。例えば `fa` (16 進数) は `250` (10 進数) に、そして `372` (8 進数) に変換されることがわかります。

第 3 章 コンピュータにおけるデータ単位と数値型

3.1 データ単位: ビットとバイト

コンピュータ内部では、最も基本的なデータ単位は**ビット（bit）**であり、これは 2 つの状態（0 または 1）を表します。しかし、多くのコンピュータシステムでは、8 ビットをひとまとめにして扱います。この 8 ビットの塊を**バイト（Byte）**と呼びます。

- **1 バイト = 8 ビット**: 1 バイトで表現できる値の範囲は、$2^8 = 256$ 種類の状態、つまり 0 から 255 までの整数です。
- **KB (キロバイト) と KiB (キビバイト)**:
    - コンピュータの文脈で「KB（キロバイト）」と表記された場合、通常は 2^{10} バイト、つまり **1024 バイト** を指します。これは国際単位系

（SI）のキロ（1000）とは異なります。正確には **KiB（キビバイト）** と表記されるべきですが、一般的なコンピュータの教科書では大文字の B がバイトを、小文字の b がビットを表し、K、M、G、T は 2 の冪乗を意味することが多いです。

- 1KB = 2^{10} バイト = 1024 バイト
- 1MB = 2^{20} バイト = 1,048,576 バイト
- 1GB = 2^{30} バイト = 1,073,741,824 バイト
- 1TB = 2^{40} バイト = 1,099,511,627,776 バイト SI 接頭辞（1000 倍）と 2 の冪乗（1024 倍）の違いは、容量が大きくなるほど顕著になります。

3.2 数値型

コンピュータプログラムでは、様々な種類の数値を扱うために、**数値型（Data Type）**を使い分けます。主なものに**整数型**と**浮動小数点型**があります。

3.2.1 *整数型* (Integer Type)

整数型は小数点以下の値を持たない数（例: 1, 100, -5）を扱います。整数型は、割り当てられるビット長によって表現できる数値の範囲が異なります。また、負の数を扱うか否か（符号の有無）によっても分類されます。

- **16 ビット整数型**:
  - **符号なし整数 (Unsigned Integer)**: 0 から $2^{16} - 1$ までの値を表現できます。つまり、0 から 65535 までです。
  - **符号あり整数 (Signed Integer)**: 負の数も表現でき、-2^{15} から $2^{15} - 1$ までの値を表現します。つまり、-32768 から 32767 までです。
- **32 ビット整数型**:
  - **符号なし整数**: 0 から $2^{32} - 1$ (約 4.29×10^9) までの値を表現できます。
  - **符号あり整数**: -2^{31} から $2^{31} - 1$ (約 $\pm 2.14 \times 10^9$) までの値を表現できます。
- **64 ビット整数型**:

- 現在の CPU の主流は 64 ビットであり、一般的に 64 ビット整数型が標準的に使用されます。
- **符号なし整数 (Unsigned Long Long Integer)**: 0 から $2^{64} - 1$ (約 1.84×10^{19}) までの値を表現できます。
- **符号あり整数 (Signed Long Long Integer)**: -2^{63} から $2^{63} - 1$ (約 $\pm 9.22 \times 10^{18}$) までの値を表現できます。 日本の GDP（約 5 兆 US ドル）のような巨大な金額を正確に扱うには、16 桁程度の精度が必要となり、64 ビット整数型が適切です。円周率の桁数計算のような、さらに巨大な整数を扱う場合は、標準の整数型では表現しきれないため、**多倍長演算**といったソフトウェア的な処理が必要になります。

3.2.2 *浮動小数点型* (Floating-Point Type)

浮動小数点型は、小数点以下の値を持つ実数（例: 3.14, -0.001）を扱います。コンピュータでは、実数を「符号部」「指数部」「仮数部」に分けて表現します。

- **IEEE 754 規格**: 浮動小数点数の表現は、**IEEE 754** という国際標準で定められています。主なものに以下の 2 種類があります。
  - **単精度浮動小数点数 (Binary32)**: 32 ビットを使用。
  - **倍精度浮動小数点数 (Binary64)**: 64 ビットを使用。
  - より高い精度が必要な場合は、**四倍精度 (Binary128)** なども存在します。
- **Binary64 (倍精度)** の構成例:
  - **符号部 (Sign)**: 1 ビット（正負を表す）。
  - **指数部 (Exponent)**: 11 ビット（数値の大きさのオーダーを表す）。
  - **仮数部 (Fraction/Mantissa)**: 52 ビット（数値の有効数字部分を表す）。 これにより、約 2^{-1022} から 2^{1023} までの範囲の数を表現でき、有効桁数は約 15〜17 桁に相当します ($2^{52} \approx 4.5 \times 10^{15}$)。
- **精度の重要性**:

- 原子のエネルギー（水素原子の 1s 軌道エネルギーは 13.6 eV）から、室温での熱エネルギー（約 62 meV）、磁気エネルギー（数 meV）といった物理現象のエネルギーは、非常に広い範囲にわたります。
- これらを統一的に扱う場合、meV（ミリ電子ボルト）から MeV（メガ電子ボルト）までの 9 桁以上の精度が必要になります。
- 単精度（32 ビット）浮動小数点数では有効桁数が約 7 桁しかないため、この要求を満たすことができません。
- そのため、少なくとも**倍精度（64 ビット）浮動小数点数**を使用することが不可欠であり、計算によっては**四倍精度**や**多倍長演算**が必要となる場合もあります。

例: ボルツマン因子 シリコンのバンドギャップ 1.1 eV、室温での熱エネルギー $kT \approx 62$ meV の場合、ボルツマン因子 $e^{-E/kT}$ は約 10^{-19} となります。 ワイドバンドギャップ材料（バンドギャップ 4 eV）であれば約 10^{-67}、極低温（3 K）であれば 10^{-5141} といった極めて小さな値になります。 倍精度浮動小数点数で表現できる最小値は約 10^{-308} ですから、10^{-5141} のような値は倍精度では扱えません。このような場合に四倍精度（約 10^{-4932} まで対応）やそれ以上の精度が必要となります。しかし、これらの高精度演算は CPU ハードウェアで直接サポートされていないことが多く、ソフトウェアで実装されるため、計算速度は大幅に低下します。

第 4 章 数値計算における誤差

4.1 浮動小数点数表現の限界と丸め誤差

実数には無限の桁数を持つものがありますが、コンピュータの浮動小数点型変数は**有限のビット長**しか持たないため、すべての実数を正確に表現することは不可能です。これにより生じる誤差を**丸め誤差 (Rounding Error)** と呼びます。

コンピュータ内部ではすべての数値が 2 進数で表現されます。浮動小数点数は一般的に $\pm 1.F \times 2^E$ の形式で表現されます。

- **正確に表現できる例**:
  - $1.0 = 1.0_2 \times 2^0$
  - $0.5 = 1.0_2 \times 2^{-1}$
  - $0.125 = 1.0_2 \times 2^{-3}$ これらは 2 進数で有限桁の仮数部で表現できるため、コンピュータ上で正確な値として扱われます。
- **正確に表現できない例**:
  - 0.1 (10 進数) は、2 進数で表現すると $0.0001100110011\ldots_2$ となり、無限に続く循環小数になります。 このため、コンピュータは 0.1 を有限桁数で近似して表現することになり、わずかな誤差（丸め誤差）を含みます。

4.1.1 *丸め誤差の累積*

`0.1` のような近似値を使って計算を繰り返すと、丸め誤差が累積し、最終結果に大きな影響を与えることがあります。 例えば、`0.1` を 100 回足し合わせる計算を考えてみましょう。 数学的には $0.1 \times 100 = 10.0$ となりますが、浮動小数点数でこれを計算すると、わずかに 10.0 からずれた値になることがあります。 一方、`0.125` のように正確に表現できる数をいくら足し合わせても、誤差は生じません。このことから、**浮動小数点数の計算結果には常に誤差が含まれる可能性がある**ことを認識しておく必要があります。

4.2 その他の誤差の発生源と対策

丸め誤差以外にも、コンピュータシミュレーションには様々な誤差の発生源があります。

4.2.1 *オーバーフロー* (Overflow) *とアンダーフロー* (Underflow)

- **オーバーフロー**: 変数型で表現できる**最大値を超える**ような大きな数値を扱おうとすると発生します。
- **アンダーフロー**: 変数型で表現できる**最小値（0 に近い極めて小さな数）を下回る**ような数値を扱おうとすると発生します。この場合、値が 0 として扱われてしまい、情報が失われます。

これらの誤差は、計算結果が予期せぬ大きな値や小さな値になった場合に生じます。

4.2.2 *桁落ち* (Loss of Significance)

ほぼ等しい数値同士の引き算を行うと、有効桁数が大幅に失われる現象を**桁落ち**と呼びます。 例: $5\sqrt{41}-32$ の計算を考えます。 $\sqrt{41}\approx 6.40312423743$ $5\sqrt{41}\approx 32.01562118715$ もし計算が 4 桁の有効数字で行われるとします。 $5\sqrt{41}\approx 32.02$ $32.02-32.00=0.02$ この結果は、もとの値が持っていた有効桁の多くを失い、1 桁しか有効数字が残っていません。このような計算は避けるべきです。 対策としては、数式を変形して引き算を回避するなどの方法があります。 例えば、$\sqrt{x^2+1}-x$ のような式は、$\frac{(\sqrt{x^2+1}-x)(\sqrt{x^2+1}+x)}{\sqrt{x^2+1}+x}=\frac{(x^2+1)-x^2}{\sqrt{x^2+1}+x}=\frac{1}{\sqrt{x^2+1}+x}$ と変形することで桁落ちを回避できます。

4.2.3 *情報落ち* (Catastrophic Cancellation / Loss of Information)

絶対値が非常に異なる数値同士の加算や減算を行うと、絶対値の小さい方の情報が失われる現象を**情報落ち**と呼びます。 例: 1000 に 1.456 を加える計算を考えます。 もしコンピュータが 4 桁の有効数字でしか計算できないとすると、$1000+1.456\approx 1001.456\rightarrow 1001$ (4 桁に丸め) この結果では、元の 0.456 という情報が失われてしまいます。 幸い、現在のコンピュータでは 64 ビット浮動小数点数が主流であり、約 15〜17 桁の有効桁数があるため、このような極端な情報落ちはほとんど発生しません。しかし、多数の計算を繰り返す場合には影響が出る可能性があるので、注意が必要です。

4.2.4 *打ち切り誤差* (Truncation Error)

無限級数や連続的な関数を**有限の項で打ち切って近似**することで生じる誤差を**打ち切り誤差**と呼びます。 例: 関数をテイラー展開で近似する場合。 $f(x)=f(a)+f'(a)(x-a)+\frac{f''(a)}{2!}(x-a)^2+\cdots+\frac{f^{(n)}(a)}{n!}(x-a)^n+R_{n+1}$ テイラー展開は無限項の和ですが、コンピュータで計算する際には有限項で打ち切らざるを得ません。この打ち切った部分が打ち切り誤差となります。

4.2.5 *収束誤差* (Convergence Error)

繰り返し計算（イテレーション）によって解を求める**自己無撞着計算（Self-Consistent Field calculation）**などにおいて、計算を**有限回で打ち切る**ことで生じる誤差を**収束誤差**と呼びます。理想的には無限回繰り返せば厳密な解が得られますが、実際には計算コストの制約から、ある程度の許容誤差に達した時点で計算を終了します。

4.2.6 *モデル誤差* (Model Error)

物理現象をシミュレートする際に、**物理モデルそのものに近似**が含まれている場合に生じる誤差です。これは数値計算による誤差とは異なり、モデルの不完全性や単純化に起因します。

4.3 誤差を回避するためのプログラミングのヒント

4.3.1 *浮動小数点数の等価比較*

浮動小数点数は丸め誤差を含むため、**厳密な等価比較 (==) は避けるべき**です。 例えば、`x * 10.0 == 30.0` のような比較は、`x * 10.0` の計算結果が 30.0 にわずかに足りなかったり、超えたりすることがあるため、期待通りに動作しないことがあります。 代わりに、**許容誤差 (epsilon, ϵ)** を用いて比較します。 `|` (計算結果) - (期待値) `|` `<` `epsilon` 例: `abs(x * 10.0 - 30.0) < 1e-10` のように、絶対値の差が非常に小さな値 (ϵ) より小さいかどうかで判断します。

4.3.2 *浮動小数点数から整数への型変換*

浮動小数点数を整数型に変換する際にも注意が必要です。 例えば、`value` が 10.0 であるはずが、コンピュータ内部では丸め誤差によってわずかに 9.999999999999999 のような値になっていることがあります。このとき、`int(value)` のような関数を使用すると、結果が 9 になってしまう可能性があります。 これを回避するためには、やはり許容誤差 ϵ を利用します。 `int(value + epsilon)` このようにわずかに正の値を加えることで、期待される値が 10 であれば

10 に、それより小さければ 9 にと、意図した通りの変換結果が得られやすくなります。

4.3.3 *数値的に不安定な計算の回避*

テイラー展開における情報落ちの例: e^{-x} の値をテイラー展開で計算する場合を考えます。 $e^{-x} = \sum_{n=0}^{\infty} \frac{(-x)^n}{n!} = 1 - x + \frac{x^2}{2!} - \frac{x^3}{3!} + \cdots$ もし x が大きな正の値（例: $x = 39$）の場合、偶数項は非常に大きな正の値に、奇数項は非常に大きな負の値になります。そして、これらの大きな値同士を足し引きすることで、**桁落ちや情報落ち**が発生し、最終的な結果の精度が著しく損なわれることがあります。 例えば、e^{-39} は数学的には 4.25×10^{-18} という極めて小さな値ですが、上記のテイラー展開を直接計算すると、Python でも 5.8811665 のような全く異なる値に収束してしまうことがあります。これは、途中で大きな正負の数が相殺し合う際に有効桁が失われたためです。

このような場合は、**数値的に安定な計算方法**を選択することが重要です。 $e^{-x} = \frac{1}{e^x}$ と変形し、e^x をテイラー展開で計算します。 $e^x = \sum_{n=0}^{\infty} \frac{x^n}{n!} = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \cdots$ この場合、全ての項が正の値となるため、大きな正負の数同士の引き算による桁落ちや情報落ちは発生しません。結果として、e^{-39} の計算でも正確な 4.25×10^{-18} に収束することが期待できます。

コンピュータシミュレーションプログラムを開発する際には、常にこれらの誤差の発生源を意識し、適切な対策を講じることが重要です。

第 5 章 まとめと課題

本日の講義では、コンピュータにおける数値表現の基礎から、データ単位、数値型、そして数値計算に潜む様々な誤差について学習しました。特に浮動小数点数の表現限界、丸め誤差、桁落ち、情報落ち、打ち切り誤差、収束誤差といった概念は、シミュレーションの精度に直結する非常に重要なポイントです。

皆さんは本日出題された課題に取り組み、これらの概念を実践的に理解してください。

- **課題 1**: 基数変換 (2 進数から 10 進数、10 進数から 16 進数)
    - 計算過程を自身で導き出してください。
    - 提供された `base.py` プログラムは、答え合わせのために利用してください。
- **課題 2**: Python プログラムの解析
    - 提供された Python プログラムのいずれかを選び、その機能やコードの役割を理解する努力をしてください。
    - 理解できた範囲で構いませんし、不明点があればそれを明確にしてください。

課題提出は、**6 月 11 日（水）の深夜 23 時 59 分まで**に LMS を通じてお願いします。

以上で本日の講義を終了します。質問があれば随時受け付けます。