第２０話　数値計算のポイントは精度と安定性と処理スピード

今回、数値計算の重要ポイントについてお話しする。ここでの数値計算はコンピュータを使用した計算法であり、理論とプログラミング、使用法などが含まれ、連立方程式の解法、固有値問題、微分方程式の解法、数値微分・積分など多岐に渡っている。全体を通して**最も重要なのは、計算精度と安定性と処理スピード**である。建築構造を学ぶ学生には、このような話はあまり興味ないかもしれない。しかし、構造設計や構造解析を行うとき、コンピュータを用いたソフトを必ず使用する。ここで学ぶことを通して、ソフトを書く人達の苦労を少しは知って欲しい。

まず、精度について考える。理論では無限の数値を前提にしている。それは 10 進法でも 2 進法でも同じであり、無限に続く数列を扱うことができる。一方、コンピュータは有限の数列で処理することになる。一般的な内部表現は 2 進法であり、整数と実数、複素数に分かれている。今日の PC は、32 ビット及び 64 ビットマシンであるため、基準の数値は 16，32、64、128 ビットである。整数は 32 ビットで表され、10 進法では約 9 桁が表現可能。単精度実数は 32 ビットで表され、内訳は符号 1 ビット、指数部 8 ビットと仮数部 23 ビットに分かれており、各々 ±128 乗と 7 桁が有効。倍精度実数は符号 1 ビット、指数部 11 ビット・±1024 乗、仮数部 52 ビット・16 桁が有効である。プログラマは扱える数値の有効桁数を常に考慮し、プログラミングコードを記述する。

有効桁数を持った数値での四則演算では、丸目誤差、桁落ち、オーバーフローやアンダーフローなどに注意を払う。例えば、絶対値の近い数値の加減算は有効桁が減少し、桁落ちが発生する。ゼロに近い数値で割り算を行うとオーバーフローを起こし易い。大きな数値と小さな数値を加えると小さな数値の持つ情報が失われる。このような現象を避けるため、プログラマは必ずしも理論通りのアルゴリズムを用いず、式を変形するなど工夫し、計算精度を保つよう心がける。また、現在の数値計算では、桁落ちを考慮してほとんど倍精度実数を用いている。

有限要素法では、極端な扁平要素や大きさが大きく異なる要素を使用すると桁落ちを起こし、精度が悪くなる。骨組構造でも同様で、長さが極端に違う部材を用いると、剛性の数値が大きく異なり桁落ちが生じる。また、異種構造の解析でも同様の現象が生じる場合がある。例えば、骨組と地盤の連成振動などが挙げられる。使用者側も、これらの情報や他で得た経験を活用し、計算精度を保つよう注意することが肝心である。

コンピュータで使用される数値やコードの基礎をここでまとめておこう。

ビット：1 つの情報単位で、0 か 1 を表す。IC メモリひとつに対応する。

バイト；8 ビットで構成され、256 の ASKII 文字を表すことができる。この 1 バイトで文字が表すことできるとしていたが、漢字のように多くの文字を表すためには 2 バイトコードが用いられている。

16 ビットと 32 ビットマシン：両者の違いは、ダイレクトにアクセスできるメモリ空間であり、前者では 1 M バイト、後者では 4 G バイトになる。64 ビットマシンでは事実上無限に近く、物理的に搭載可能なメモリをすべて使用することができる。

オーバーフロー及びアンダーフローとは？

単精度で指数部は 8 ビット、培精度では 11 ビットであり、10 進表現では、前者は 128 乗から-127 乗まで、後者は 1024 乗から-1023 乗までとなる。この値を超えるとアンダーフロー/オーバーフローとなる。

反復計算などで解が発散する場合や、振動方程式や非線形解析など、解が安定せず、予想外の値を採ることがある。多くの場合、計算アルゴリズムの選択やパラメータの設定ミスによるところが大きい。これら計算の安定性については後日話すことにしよう。

PC の処理能力の向上や内部メモリの飛躍的な増加、数値計算処理技術の進歩により、3 次元モデルの非線形振動解析を実施可能となる。ここでは、精度や安定性は無論のこと、特に処理スピードが問題となる。

今回は、処理速度向上技術の一つであるキャシュを如何に効果的に活用するかについて考えてみよう。キャシュは、CPU と DRAM の速度差を補うために、高速の SRAM を中間に配置し、CPU の待ち時間を少なくする。ただし、SRAM は DRAM に比較してコスト高で、大量に組み込めない。現在、キャシュは CPU の中に 1 次、2 次、3 次キャシュまであり、数字が若いほど高速であるが容量は少ない。プログラムもデータもこのキャシュを経由して CPU 内に取り込まれる。実際のプログラムも元データも DRAM 内にあるが、OS が気を利かして、つまり現在必要としているプログラムやデータがキャシュ内にないと判断すると、DRAM 内の該当するデータ群を取り込む。これをスワップという。プログラムもデータも必要な部分を含む領域全体をスワップするが、1 次キャシュの容量は小さいので、その領域は小さい。処理が進み、キャシュ内に必要なコードがない場合や必要なデータがない場合、再度スワップが発生する。当然、スワップは OS 任せである。

このようなキャシュ内のプログラムやデータのスワップ状況を考慮して、如何にプログラムの処理速度を向上させるかが、プログラマの腕の見せ所である。キャシュの動きによって、処理速度が大幅に変わる場合があり、データのアクセスを局所的にすることで頻繁にスワップを起こさせないようにする。例として、次に示す行列とベクトルの掛け算について考える。

$$x(i) = \sum_{j=1}^{n} A(i,j) * y(j); \quad i = 1 \sim n$$

ここで n が大きな値であるとき、言語 C や C++によるコードでは局所的にアクセスしているが、Fortran では離れたメモリにアクセスするため頻繁にスワップする可能性がある。この種の計算は他のアルゴリズムにも多数存在し、プログラマは常に局所的アクセスとなるよう工夫する。

今回はキャシュについてのみお話ししたが、処理速度を向上させる方法は他にもある。例えば、計算のベクトル処理、並列計算があるが、これらはスーパーコンピュータで用いられた手法であり、今では、PC に使われている。これらを使いこなすための話は、後日としよう。

二次元配列 $A(100,100)$ を定義するとき、言語によってメモリの割り付け順が異なる。FORTRAN では、先に列方向から以下のように割り付ける。

$A(1,1), A(2,1), A(3,1)\cdots$
$A(n-1,1), A(n,1),$
$A(1,2), A(2,2)\cdots$
$A(n-1,n), A(n,n)$

一方、C や C++では、先に行方向から以下のように割り付ける。

$A(1,1), A(1,2), A(1,3)\cdots$
$A(1,n-1), A(1,n),$
$A(2,1), A(2,2)\cdots$
$A(n,n-1), A(n,n)$

従って、FORTRAN で左の積和を実行すると配列 **A** のメモリ位置を大きく移動しながらアクセスすることになり、配列が大きいと頻繁にスワップを繰り返すことになる。そこで、次のように計算順序を変更すれば、順番にアクセスすることになり、スワップが生じない。

```
do j=1,n
do i=1,n
x(i)  += A(i,j)*y(j)
enddo
enddo
```