# 範囲の写像

## はじめに

2 点で決定される範囲を他の範囲に写像することがある．2 点内(あるいはその外側)にある点をどこに写像するのかという問題に由来する．大方は単純な線形問題と思われるかもしれない．だが，侮るなかれ，これには大きな落とし穴が潜んでいる．これを知らずにプログラミングしているとすれば，計測機器としては欠陥プログラムである．そして現実には，そのようなプログラムが大量生産されている，ということを筆者は知っている．本稿では一般に語られない(簡単と思われている)写像の闇を解説する．

## 範囲の写像

まず，範囲の写像を**図 1** に示す．

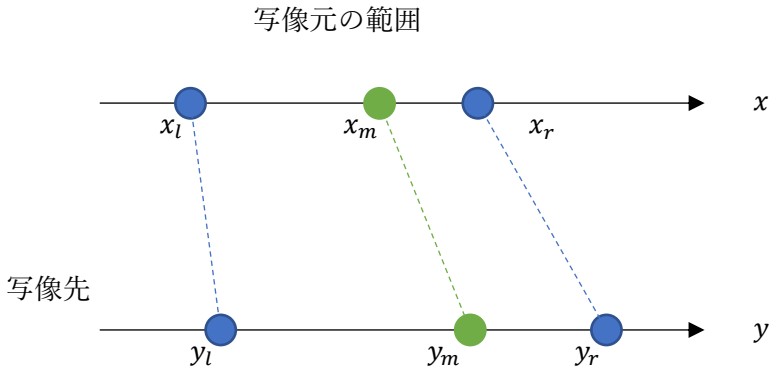


**図 1　範囲の写像**

ここで$y_m$を求めるには(1)である．

$$y_m = y_l + (y_r - y_l)\frac{x_m - x_l}{x_r - x_l} \quad (1)$$

一見，問題ない．だが，問題ないのは写像元と写像先がいずれも浮動小数(実数)の場合である．次節からは 2 つの場合を解説する．

## 整数と整数の写像

整数と整数の間の写像が**図 2** である．

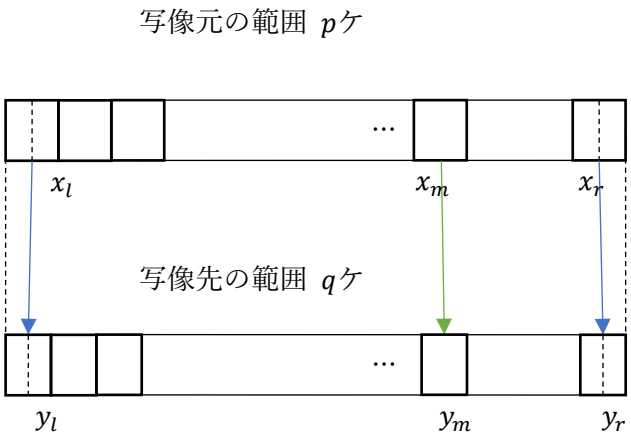


**図 2　整数と整数の写像**

主にグラフィックを扱う場面で問題は生ずる．例えば，マウスクリック位置と画像の画素との対応づけ，あるいは画像サイズを変更するといった場面である．まず，(2)，(3)である．

$$p = x_r - x_l + 1 \quad (2)$$

$$q = y_r - y_l + 1 \quad (3)$$

画素数をカウントすれば上記で間違いない．これを(1)へ代入すれば(4)である．

$$y_l + (q-1)\frac{x_m - x_l}{p-1} \quad (4)$$

全体としては$p$，$q$ケの画素数であるはずなのに，$p-1$，$q-1$でスケーリングされる．これは異常事態である．よくよく眺めてみると，この原因は両端の画素の外側 0.5 画素分を考慮していないことにある．つまり，画素の厚みを考えないと，両端の画素とそれ以外の内側の画素では画素あたりの幅が異なるのである．このままでは正確な写像とならない． 修正すると(5)である．

$$y_m = \mathrm{Round}\left((y_l - 0.5) + q\frac{x_m - (x_l - 0.5)}{p}\right) \quad (5)$$

Round: 四捨五入

## 浮動小数と整数の写像

整数と整数の写像の問題が理解できれば，一方が浮動小数であっても同様である．基本的には整数側の両端で，いずれの数値が浮動小数側の数値と対応しているのか，である(**図3**)．

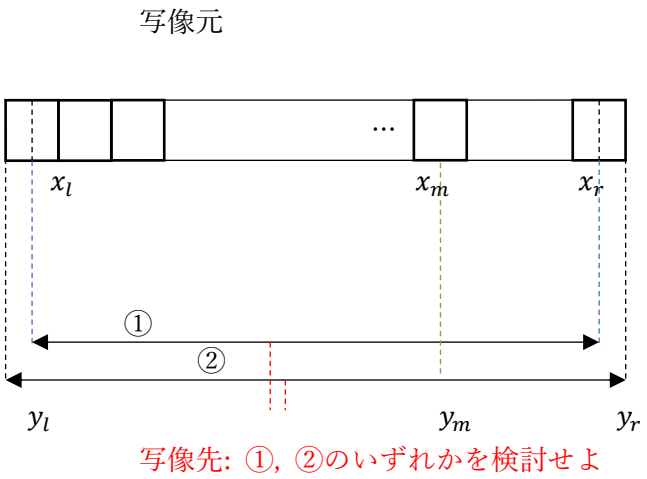


**図3　浮動小数と整数の写像**

以上，これを実現するのにC#やC++で範囲をクラスとして設計するのが賢いやり方であるし，筆者も実施している．一点，注意がある．2点の関係で，左側が小さい数値，右側が大きい数値に限定しないで設計したいと思うこともある．具体的には小さい方の数値を大きい方の数値に対応づけるやり方である．筆者としては，あまり，お勧めしない．理由として2つのやり方でわずかな差異が生ずることがあるからだ．

例　　写像元の範囲(整数):　　(2, 10)
　　写像先の範囲(倍精度浮動小数):　　(1.1, 5.34)
　　[図3では厚みを考える②を選択]

| 写像元 | ⇒ | 写像先 |
|---|---|---|
| 3 | | 1.8066666666666666 |

写像先を入れ換えて算出すると下記である．

　　写像先の範囲(倍精度浮動小数):　　(5.34, 1.1)

| 写像元 | ⇒ | 写像先 |
| --- | --- | --- |
| 3 | | (1.1 + 5.34 - 4.6333333333333329) |
| | | = 1.8066666666666671 |

グラフィックでは座標系の向きを反転する場面がしばしばあり，このようなことは日常茶飯事である．筆者としてはわずかな差(有効桁の末尾 2 桁程度)でも，やり方によって変化するのは気になる．同一の問題に対しては，統一するのがよいかと思う．本件は数値を離散値への丸めと計算の累積による差である．筆者の場合，小さいもの同士，大きいもの同士で統一している．こうすることで入力時の不正入力防止にもなる．とはいっても，五月蠅すぎるとの批判もありそうな件である．この程度は気にならないということなら，それも構わない．

## 四捨五入

四捨五入も若干の注意が必要である．四捨五入の最も単純なプログラムは**図 4** である．

```
int Round(double v)
{
        int x = static_cast<int>(v + 0.5);
        return x;
}
```

**図 4　四捨五入のプログラム(C++)**

だが，このプログラムには欠陥がある．例えば下記である．

-2.4 ⇒ -1
-2.5 ⇒ -2
-2.6 ⇒ -2

負数を考慮しておらず，プログラムを修正する(**図 5**)．

```
int Round(double v)
{
        int x = (v >= 0)? static_cast<int>(v + 0.5): static_cast<int>(v - 0.5)
        return x;
}
```

**図 5　［修正版］四捨五入のプログラム(C++)**

これにより下記となる．

-2.4 ⇒ -2
-2.5 ⇒ -3
-2.6 ⇒ -3

C ランタイムの round 関数はこれと同じ動作である．だが，気になることがある(**図 6**)．

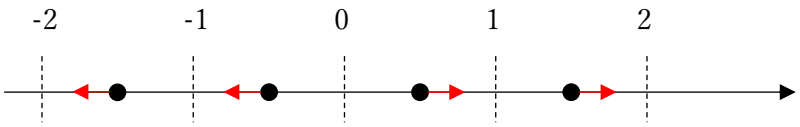


**図 6　0.5 の切り上げる方向**

0.5 の切り上げる方向がどちらかという問題である．場合によっては，軸の正負に関わらず，0.5 の切り上げ方向を一定に保ちたいことがある．例えば，温度を計測して整数に丸める場合，どちらかというと上記よりも，切り上げ方向が一定方向であることが望ましい．他に，データの範囲をまとめてシフトする(オフセットの変更)ような場合を考えてみる．元のデータが正負にまたがっていた場合，その状態で整数に丸めてから正数側にシフトするのと，正数側にシフトしてから丸めるのでは異なる．以上，数値計算には厳密さを必要とする場面がある．

## まとめ

まとめとして，写像元と写像先のいずれかが整数であった場合，扱う対象を十分に吟味する必要がある．また，範囲に関しては広い範囲で定義した方が高精度となるのは当然である．外挿はできるだけ避け，2 点間の内挿となるように 2 点を与えるのが望ましい．

本件は筆者がグラフィックのプログラミングを初めて間もないころに直面した問題である．当時，マウスクリック位置での画像の輝度値を表示するプログラムで時折，不可解な数値が表示された．このデバッグと修正に実に 100 時間以上も消費した．プログラミングで洗礼を受けた初めての件である．