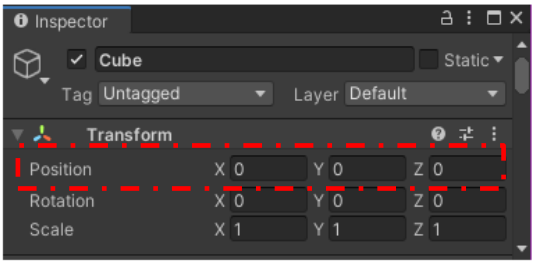
ベクトル・行列応用② ～平行移動～

■回転操作以外の変換行列（Transformation Matrix）

「ベクトル・行列応用①」ではベクトルの回転（座標系の回転）について調べ、正規直交基底ベクトルの組合せが座標系になっており、さらに回転行列にもなっていることがわかりました。ここでは、あるベクトルを別のベクトルに変換する行列について、回転操作以外の変換行列を紹介します。

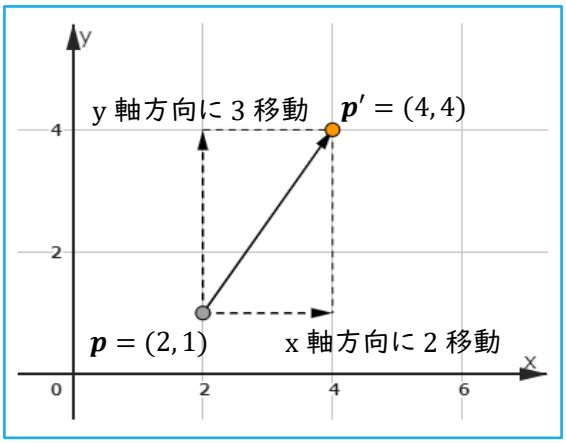
■平行移動（トランスレーション）行列

Unity でゲームオブジェクトのインスペクターを見てみると、右図のように【Transform】というコンポーネントがあります。じつは内部では行列のデータになっています。このセクションでは平行移動（右図の【Position】）の部分を見ていきたいと思います。まずは Unity 上で動きと行列データの確認をしてみましょう。平行移動は親オブジェクトの座標系上で移動していることがわかります。



※Position は「位置」という意味ですが、
座標系の「原点から平行移動した場所」と同義です

【Fig1. x 軸方向に 2, y 軸方向に 3 移動の図】



さて、我々は座標系を行列で表現できること、任意のベクトルの回転操作は行列との積で可能であることを知っています。そこで、回転操作以外の座標変換も行列との積で可能にしたいと考えます。その理由としては、

$$\boldsymbol{M = S_1 R_3 T_2 R_2 T_1 R_1} \quad \left(\boldsymbol{R_1, R_2, R_3, T_1, T_2, S_1} \in \text{変換行列}\right)$$

のように、複数の座標変換を事前に行列の積で求めておけば、大量のポリゴン頂点の座標変換コストを抑えることが可能にできるからです。また、行列との演算で統一することで、処理の流れをシンプルにすることができるからです。

```
for(int i = 0; i < 10000; ++i) {
    M = S1R3T2R2T1R1;
    displayPos = M * vertexPos;
}
```

よりも

```
M = S1R3T2R2T1R1;
for(int i = 0; i < 10000; ++i) {
    displayPos = M * vertexPos;
}
```

のほうが
計算コストが低い。

それでは、すべての平行移動の動きをベクトルと行列との積で表現してみましょう。

任意の点 P の位置ベクトル $\boldsymbol{p}=(p_x,\ p_y)$、x 軸、y 軸方向の移動量をそれぞれ $t_x,\ t_y$ とすると、
移動後の点P′の位置ベクトル $\boldsymbol{p}'=(p_x',\ p_y')$ は $\boldsymbol{p}'=(p_x+t_x,\ p_y+t_y)$ となります。

ここで、平行移動行列を $\boldsymbol{T}=\begin{pmatrix}a & b\\ c & d\end{pmatrix}$ とおいて、$\boldsymbol{p}'=\boldsymbol{T}\cdot\boldsymbol{p}$ となるような a,b,c,d を見つけます。

$$\boldsymbol{p}'=\boldsymbol{T}\cdot\boldsymbol{p}=\begin{pmatrix}a & b\\ c & d\end{pmatrix}\begin{pmatrix}p_x\\ p_y\end{pmatrix}=\begin{pmatrix}ap_x+bp_y\\ cp_x+dp_y\end{pmatrix}$$

よって

$$\begin{cases}ap_x+bp_y=p_x+t_x\\ cp_x+dp_y=p_y+t_y\end{cases}$$

を常に満たす a,b,c,d があればいいのですが・・・　どうやら決められそうにありません。

しかしながら、いろいろな座標変換を行列とベクトルとの積で計算させることが目的なので、どうにかしたいところです。

★同次座標系（Homogeneous Coordinates）に拡張

さきほどの計算式をもう一度見てみましょう。

$$\boldsymbol{p}'=\boldsymbol{T}\cdot\boldsymbol{p}=\begin{pmatrix}a & b\\ c & d\end{pmatrix}\begin{pmatrix}p_x\\ p_y\end{pmatrix}=\begin{pmatrix}ap_x+bp_y\\ cp_x+dp_y\end{pmatrix}$$

ここで、$a=1,b=0,c=0,d=1$ とおいてみると

$$\begin{cases}1\cdot p_x+0\cdot p_y=p_x & \cdots(1)\\ 0\cdot p_x+1\cdot p_y=p_y & \cdots(2)\end{cases}$$

となり、(1), (2) 式のそれぞれの両辺に $t_x,\ t_y$ を足せば、右辺が $p_x+t_x,\ p_y+t_y$ の形になります。

$$\begin{cases}1\cdot p_x+0\cdot p_y+t_x=p_x+t_x & \cdots(1)'\\ 0\cdot p_x+1\cdot p_y+t_y=p_y+t_y & \cdots(2)'\end{cases}$$

何か見えてきませんか？ 左辺だけに着目してみましょう。

$$\begin{cases}1\cdot p_x+0\cdot p_y+t_x\cdot 1 & \cdots(1)''\\ 0\cdot p_x+1\cdot p_y+t_y\cdot 1 & \cdots(2)''\end{cases}$$

なんと！ (1)″, (2)″ 式は

$$\boldsymbol{T} = \boxed{}, \quad {}^{h}\boldsymbol{u} = \boxed{}, \quad {}^{h}\boldsymbol{v} = \boxed{}, \quad {}^{h}\boldsymbol{p} = \boxed{}$$

とおいたときの $\boldsymbol{T} \cdot {}^{h}\boldsymbol{p}$ の x 成分、y 成分になっていることがわかります。

$$\boldsymbol{T} \cdot {}^{h}\boldsymbol{p} = \boxed{}$$

$$= \begin{pmatrix} p_x + t_x \\ p_y + t_y \end{pmatrix} \quad \cdots (3)$$

どうやら、次元数を 1 つ上げてあげれば計算可能なようです。その次元数を上げた空間を

【同次座標系（Homogeneous Coordinates）】とよび、

ゲームエンジンやグラフィクスライブラリの中身では、この同次座標系で処理がおこなわれています。

操作したい位置ベクトルや方向ベクトルの次元数を上げた成分は、基本的に 1 にします。

ほかの値にすることもできますが、ここでは取り上げません。興味がある方は下記リンク先を参考にしてください。

- https://xr-hub.com/archives/12124
- http://marupeke296.com/DXG_No55_WhatIsW.html

ところで、(3) 式に着目してみると、次元数が 1 落ちてしまっていることに気づきます。これでは問題があります。どのような問題があるのでしょうか。入力は 3 次元ベクトルなのですが、出力が 2 次元ベクトルになっていますので、つぎの座標変換の計算ができません。

（例）異なる平行移動を 2 回おこなう場合

$$\boldsymbol{T_2} \cdot \boldsymbol{T_1} \cdot \boldsymbol{p} = \begin{pmatrix} 1 & 0 & t_{2x} \\ 0 & 1 & t_{2y} \end{pmatrix} \cdot \begin{pmatrix} 1 & 0 & t_{1x} \\ 0 & 1 & t_{1y} \end{pmatrix} \cdot \begin{pmatrix} p_x \\ p_y \\ 1 \end{pmatrix}$$

$$= \begin{pmatrix} 1 & 0 & t_{2x} \\ 0 & 1 & t_{2y} \end{pmatrix} \cdot \begin{pmatrix} p_x + t_x \\ p_y + t_y \end{pmatrix} \quad \cdots \text{計算できない!!}$$

計算可能にするにはどうしたらいいでしょうか。
じつは、平行移動行列 $\boldsymbol{T}$ を 3 次の正方行列に拡張すれば問題なく計算できるようになります。

あらためて ${}^h\boldsymbol{T}=\begin{pmatrix}1 & 0 & t_x\\ 0 & 1 & t_y\\ a & b & c\end{pmatrix}$, ${}^h\boldsymbol{p}=\begin{pmatrix}p_x\\ p_y\\ 1\end{pmatrix}$ とおいて、${}^h\boldsymbol{p}=\begin{pmatrix}p_x+t_x\\ p_y+t_y\\ 1\end{pmatrix}$ となる a,b,c を見つけます。

$$
{}^h\boldsymbol{T}\cdot{}^h\boldsymbol{p}=\begin{pmatrix}1 & 0 & t_x\\ 0 & 1 & t_y\\ a & b & c\end{pmatrix}\cdot\begin{pmatrix}p_x\\ p_y\\ 1\end{pmatrix}
$$

$$
=\begin{pmatrix}p_x+t_x\\ p_y+t_y\\ ap_x+bp_y+c\end{pmatrix}
$$

$ap_x+bp_y+c=1$ を常に満たすには、$a=\quad,\ b=\quad,\ c=\quad$ であればよいので

${}^h\boldsymbol{T}=$ ［　　］ … 同次座標系におけるへ平行移動行列

となりました。

※今後は同次座標系で考えるので、基本的に $\boldsymbol{h}$ 記号は省略します

■回転行列も同次座標系へ拡張

平行移動の操作は、対象のベクトルと変換行列を同次座標系に拡張すればよいことがわかりました。そして、回転行列についても同次座標系に拡張する必要があります。

（例）これまでの回転行列では計算できない

$$
\boldsymbol{R}\cdot{}^h\boldsymbol{p}=\begin{pmatrix}cos\theta & -sin\theta\\ sin\theta & cos\theta\end{pmatrix}\cdot\begin{pmatrix}p_x\\ p_y\\ 1\end{pmatrix}
$$

… 計算できない!!

結果的に以下のような回転行列になります。

$$
{}^h\boldsymbol{R}=\begin{pmatrix}cos\theta & -\sin\theta & 0\\ sin\theta & cos\theta & 0\\ 0 & 0 & 1\end{pmatrix}
$$

… 同次座標系におけるへ回転行列

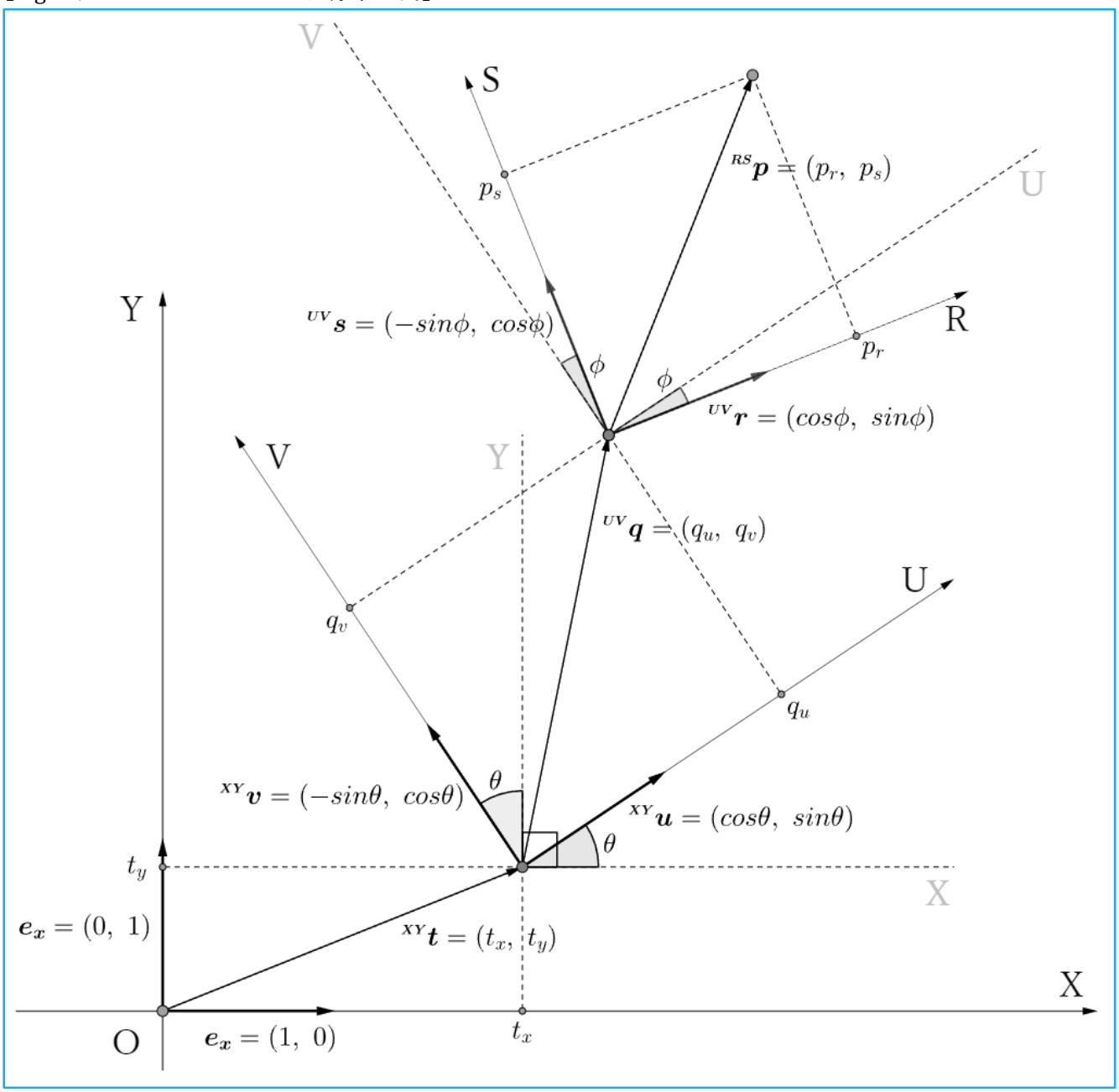
■ローカル座標系に位置情報を追加しよう

最初に Unity 上で動きと行列データの確認をしました。ゲームオブジェクトのローカル座標系は、そのオブジェクトの

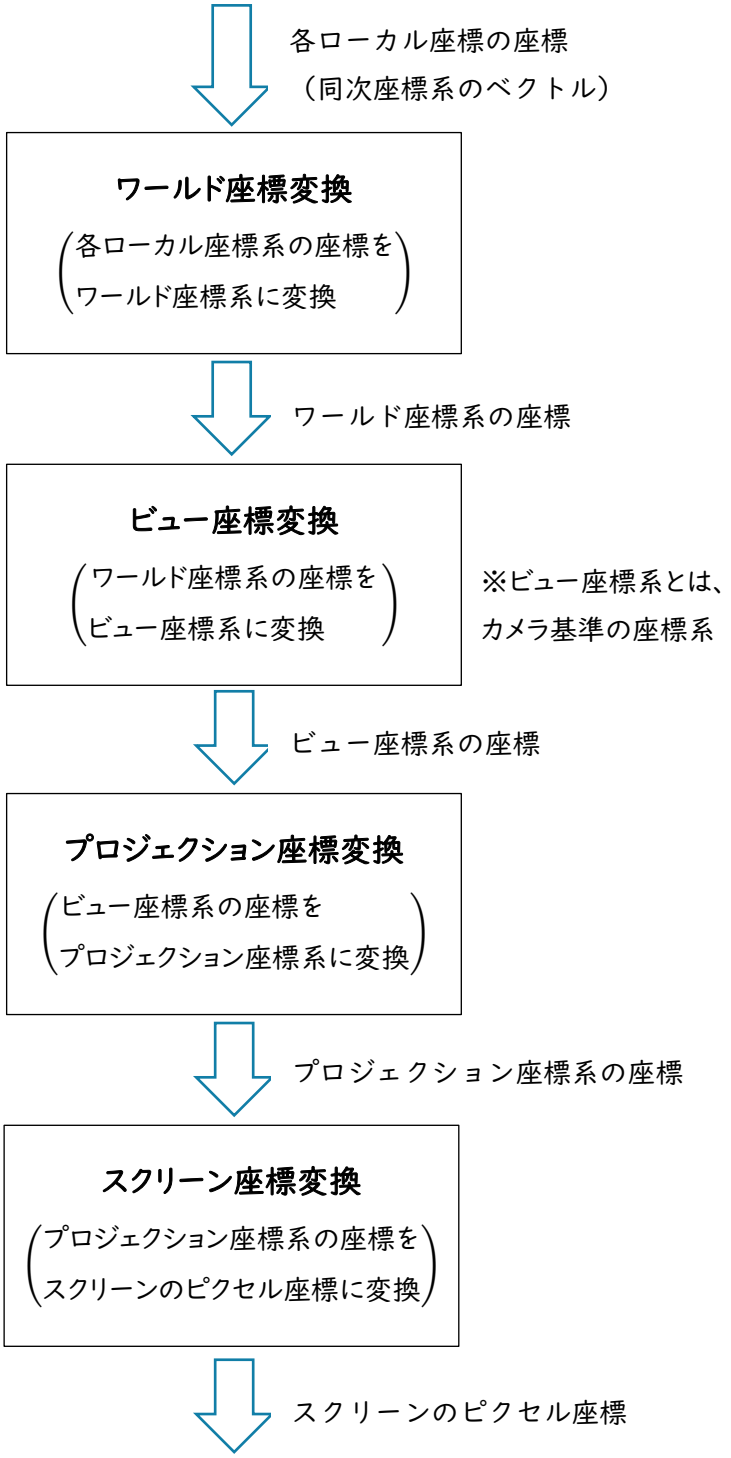
- 親座標系上における姿勢（基底ベクトルの向き）
- 親座標系上における位置

を決定します。下図のような状態です。

【Fig2. ネストされたローカル座標系の例】



ところで、ゲームエンジンやグラフィクスライブラリでは、それぞれのローカル座標系におけるベクトルを最終的にはワールド座標系に変換する必要があります。なぜなら、画面に表示するために以下の流れで座標変換をおこなうからです。



ここで Fig2 を例にして、RS 座標系の位置ベクトル ${}^{RS}\boldsymbol{p}$ をワールド座標系の位置ベクトル ${}^{XY}\boldsymbol{p}$ に変換することを考えます。RS 座標系の基底ベクトル ${}^{UV}\boldsymbol{r}, {}^{UV}\boldsymbol{s}$, および RS 座標系の原点の位置ベクトル ${}^{UV}\boldsymbol{q}$ は UV 座標系におけるベクトルなので、まず ${}^{RS}\boldsymbol{p}$ を UV 座標系におけるベクトル ${}^{UV}\boldsymbol{p}$ に変換します。

① RS 座標系の原点が UV 座標系の原点と一致している場合（その座標系をR′S′座標系とする）を考え、
${}^{RS}\boldsymbol{p}$ を ϕ だけ回転させる変換行列を ${}^{UV}\boldsymbol{R}_{RS}$ とすると、回転後の位置ベクトル ${}^{UV}\boldsymbol{p}'$ は次式で求められます。

$$ {}^{UV}\boldsymbol{p}' = {}^{UV}\boldsymbol{R}_{RS} \cdot {}^{RS}\boldsymbol{p} $$

ただし、$ {}^{UV}\boldsymbol{R}_{RS} = \begin{pmatrix} cos\phi & -\sin\phi & 0 \\ sin\phi & cos\phi & 0 \\ 0 & 0 & 1 \end{pmatrix} $

② ①で求めた ${}^{UV}\boldsymbol{p}'$ を ${}^{UV}\boldsymbol{q}$ だけ平行移動させれば、求める ${}^{UV}\boldsymbol{p}$ となることがわかります。
平行移動行列を ${}^{UV}\boldsymbol{T}$ とおいて

$$ {}^{UV}\boldsymbol{p} = {}^{UV}\boldsymbol{T} \cdot {}^{UV}\boldsymbol{p}' $$

ただし、$ {}^{UV}\boldsymbol{T} = \begin{pmatrix} 1 & 0 & q_u \\ 0 & 1 & q_v \\ 0 & 0 & 1 \end{pmatrix} $

①、② より

$$ {}^{UV}\boldsymbol{p} = {}^{UV}\boldsymbol{T} \cdot {}^{UV}\boldsymbol{R}_{RS} \cdot {}^{RS}\boldsymbol{p} $$

したがって、UV 座標系からワールド座標系の変換についても

$$ {}^{XY}\boldsymbol{p} = {}^{XY}\boldsymbol{T} \cdot {}^{XY}\boldsymbol{R}_{UV} \cdot {}^{UV}\boldsymbol{p} $$

$$ = \boxed{} $$

と表現できることがわかりました。じつはこの ${}^{XY}\boldsymbol{T} \cdot {}^{XY}\boldsymbol{R}_{UV}$ や ${}^{UV}\boldsymbol{T} \cdot {}^{UV}\boldsymbol{R}_{RS}$ が、Unity におけるゲームオブジェクトの Transform 内部の 4 x 4 行列データとなっているのです（今回は 2 次元で考えているので 3 x 3 行列）。一般に【モデル行列】とよばれています。

ここで ${}^{UV}\boldsymbol{T} \cdot {}^{UV}\boldsymbol{R}_{\boldsymbol{RS}}$ を計算してひとつの行列にまとめます。

RS 座標系から UV 座標系に変換する **モデル行列** ${}^{UV}\boldsymbol{M}_{\boldsymbol{RS}}$ は

$$
\begin{aligned}
{}^{UV}\boldsymbol{M}_{\boldsymbol{RS}} &= {}^{UV}\boldsymbol{T} \cdot {}^{UV}\boldsymbol{R}_{\boldsymbol{RS}} \\
&= \begin{pmatrix} 1 & 0 & q_u \\ 0 & 1 & q_v \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} cos\phi & -\sin\phi & 0 \\ sin\phi & cos\phi & 0 \\ 0 & 0 & 1 \end{pmatrix} \\
&= \begin{pmatrix} cos\phi & -\sin\phi & q_u \\ sin\phi & cos\phi & q_v \\ 0 & 0 & 1 \end{pmatrix} \\
&= ({}^{UV}\boldsymbol{r} \quad {}^{UV}\boldsymbol{s} \quad {}^{UV}\boldsymbol{p})
\end{aligned}
$$

他によくみる表現としては $\boldsymbol{M} = \begin{pmatrix} \boldsymbol{R} & \boldsymbol{T} \\ \boldsymbol{0} & 1 \end{pmatrix}$ があります。

ただし

$\boldsymbol{R}$ は通常次元の回転行列

$\boldsymbol{T}$ は通常次元の平行移動ベクトル（列）

$\boldsymbol{0}$ は通常次元のゼロベクトル（行）

1 は実数の 1