

14. オブジェクト指向

プログラミング・データサイエンス I

2024/7/11

1 今日の目的

今日の目的

- オブジェクト指向
- フラクタル図形
- アニメーション

今回は、「Python を使うとこんなこともできる」という例を見ていきましょう。後で、自分で Python プログラミングをする際の参考にしてください。

プログラムする対象を扱う考え方は、様々なものがあります。その考え方にうまく対応したプログラミング言語を使うことで、考え方を素直にプログラムとして書くことが出来ます。オブジェクト指向 (object-oriented) もそのような考え方の一つです。現在使われているプログラミング言語の多くに、オブジェクト指向を支援する機能が備わっています。Python も例外ではありません。今日は、Python のオブジェクト指向機能を見ていきます。

オブジェクト指向プログラミングの例としてフラクタル (fractals) を扱うことにします。Google で "fractals" というキーワードで画像を検索して見てください。非常に複雑な図形がたくさん出てきます。特定のモノを差していないことがわかります。これらの図形には、何かが共通しているのです。こうした図形をプログラムを使って描く、基本的な手法についてみていきましょう。

フラクタル図形として出来上がったものを見るのは楽しいですが、フラクタルになっていく過程を見ることができれば、もっと楽しいでしょう。そこで、最後にアニメーション、つまり図形を次々に示す方法を見ていきましょう。

<https://github.com/first-programming-saga/fractals>

2 オブジェクト指向プログラミング

オブジェクト指向プログラミング (OOP, Object Oriented Programming) とは、プログラムを書く際に、データの塊を対象 (object) として捉え、その対象を操作あるいは対象の運動としてプログラム全体を書いていく方法、あるいはそのようなプログラミング手法のことです。オブジェクトには、その属性 (attributes) と操作 (methods) があります。オブジェクトの類型・型のことをクラス (class) といい、具体的値が入ったオブジェクトをインスタンス (instance) と言います。

Student
name
ID
creditEarned
registerCredit()

図1 Student クラスイメージ

例として、学生のクラス **Student** を考えましょう (図1)。このクラスには、学籍番号や氏名、取得した科目等がデータとして含まれているとします。このクラスの操作として、取得した単位を追加することを考えましょう。

個々の学生の情報は、**Student** クラスのインスタンスです。そのため、学生が入学すると **Student** クラスのインスタンスを生成し、名前と学籍番号を登録します。

Student クラスを Python で書くと、ソースコード 2.1 のようになります。5 行目の `__init__()` は、コンストラクタ (constructor) と言い、インスタンスを生成する際に使う特別なメソッドです。16 行目のメソッド `registerCredit()` は取得単位を登録するメソッドです。`self` とは自分自身のインスタンスを表しています。変数 `self._name` は、自インスタンスに属する変数を表しています。変数の前の `_` は、クラスの外部からその変数が見えないようにするための対策です。

Student クラスのインスタンスを使う例がソースコード 2.2 です。二つのインスタンス `bob` と `alice` を生成し、それぞれに取得単位を登録しています。

この例からわかることは、インスタンスの後ろにピリオドをつけて、メソッドや属性を使うことです。これまでの講義でも、ピリオドがしばしば登場していました。オブジェクトに付随した、メソッドや属性を表していたのです。

ソースコード 2.1 Student クラス: 一部省略

```
1 class Student:
2     """
3     学生のクラス
4     """
5     def __init__(self, name:str, id:str):
6         self._name: str = name
7         self._id: str = id
8         self._creditEarned:list[tuple[str,int]] = list()
9
10    def registerCredit(self, lecture:str, unit:int):
11        """
12        取得単位の登録
13        """
14        self._creditEarned.append((lecture, unit))
15
16    @property
17    def name(self) -> str:
18        return self._name
19
20    @property
21    def creditEarned(self) -> list[tuple[str, int]]:
22        return self._creditEarned
```

ソースコード 2.2 Student クラスの利用

```
1 bob = Student('bob',str(1))
2 alice = Student('alice',str(2))
3 bob.registerCredit('English',2)
4 bob.registerCredit('Math',2)
5 alice.registerCredit('French',2)
6 alice.registerCredit('Sci',2)
7 print(f"{bob.name}'s credit earned")
8 for c in bob.creditEarned:
9     print(c)
10 print(f"{alice.name}'s credit earned")
11 for c in alice.creditEarned:
12     print(c)
```

もう一つ簡単なクラスの例を見ましょう。2次元のベクトルを表すクラスです(ソースコード 2.3)。2次元のベクトルは、 x 成分と y 成分の二つの数値で表されます。ベクトルの操作として、ベクトル同士の和や差、スカラー倍を計算するメソッド定義します。

メソッド `__add__()` と `__sub__()` は、ベクトルの和と差を計算するメソッドです。他

ソースコード 2.3 Vector クラス

```
1 class Vector:#2次元ベクトル
2     def __init__(self, x:float, y:float):
3         self.x:float = x
4         self.y:float = y
5
6     def __add__(self, other)->'Vector':#ベクトルの加算
7         return Vector(self.x + other.x, self.y + other.y)
8
9     def __sub__(self, other)->'Vector':#ベクトルの減算
10        return Vector(self.x - other.x, self.y - other.y)
11
12    def __neg__(self)->'Vector':#ベクトルの符号反転
13        return Vector(-self.x, -self.y)
14
15    def __mul__(self, scalar:float)->'Vector':#ベクトルのスカラー倍
16        ↪ vector*scalar
17        return Vector(self.x * scalar, self.y * scalar)
18
19    def __rmul__(self, scalar:float)->'Vector':#ベクトルのスカラー倍
20        ↪ scalar*vector
21        return Vector(self.x * scalar, self.y * scalar)
22
23    def __str__(self)->str:#ベクトルの文字列表現
24        return f'({self.x}, {self.y})'
```

のメソッドの意味はコメントに記述しています。このようにすることで、以下のように、ベクトル同士の演算やスカラー倍を簡単に計算することができます。

ソースコード 2.4 Vector クラスの利用

```
1 v1 = Vector(0.1,0.5)
2 v2 = Vector(0.2,-0.3)
3 v3 = v1 + v2
4 print(v3)
5 print(3*v1)
6 print(-v2)
```

3 フラクタル図形

フラクタルのイメージを見ていると感じる「複雑さ」とは何でしょうか。一つの共通の性質は、あるパターンが大きさを変えて、繰り返し現れていることです。自然界にあるフラクタル図形では、正確に同じ形の繰り返しではなく、ほぼ同じ形が繰り返し現れます。

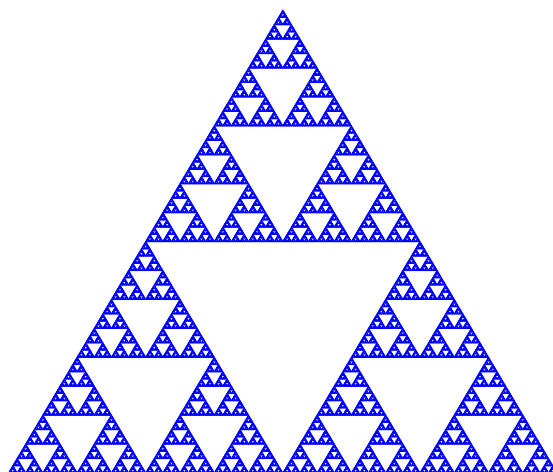


図2 Sierpinski ガスケット

図2は、Sierpinski ガスケットと言うフラクタル図形です。全体と同じ図形、つまり正三角形から中心をくり抜き、残った正三角形の中心をくり抜くを繰り返した図形が、無数に繰り返されています。このように全体と同じ性質が部分にも現れる性質を自己相似 (self-similar) と言います。このような自己相似性を生み出す仕組みを見ていきましょう。



図3 Sierpinski に対応する変換

図3を見てください。左の正三角形に対して、中心をくり抜いたものが右になります。

次のステップは、右の各正三角形に同じ操作を行います。この操作を無限回繰り返すと Sierpinski ガスケットができます。私たちが見ている画面は、有限の解像度しかありません。実際には、適当な回数の繰り返しで図 2 のような図形となります。

別の作り方も示しましょう。図 4 をみてください。左の 1×1 の平面部分を $1/2$ に縮小し、原点をずらして三つ張り合わせたものが中央の図です。次のステップは、中央の三つの正三角形の全体、つまり各点に対して 1×1 の平面部分を再び $1/2$ に縮小し、三つ貼り合わせるのです。これを繰り返します。

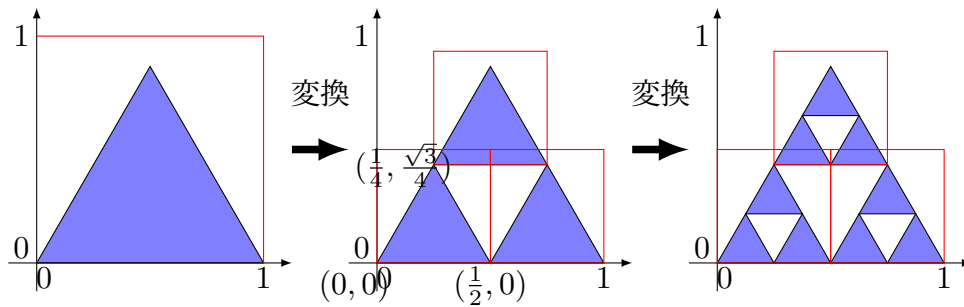


図 4 Sierpinski に対応する変換

後者は以下のような変換に相当します。 x 及び y の値が $[0, 1)$ の範囲の点 (x, y) を考えます。図 4 の変換は以下のような線形変換で表すことができます。

$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} 1/2 & 0 \\ 0 & 1/2 \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} + \begin{pmatrix} 0 \\ 0 \end{pmatrix} \quad (3.1)$$

$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} 1/2 & 0 \\ 0 & 1/2 \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} + \begin{pmatrix} 1/2 \\ 0 \end{pmatrix} \quad (3.2)$$

$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} 1/2 & 0 \\ 0 & 1/2 \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} + \begin{pmatrix} 1/4 \\ \sqrt{3}/4 \end{pmatrix} \quad (3.3)$$

式 (3.1)、(3.2)、(3.3) がそれぞれ、左下隅、右下隅、中央上の小さな三角形への変換を表します。この変換では、それぞれの図形を $1/2$ に縮小するだけで、回転や捻りを入れていませんから、行列の成分は対角成分は $1/2$ だけです。一方、並行移動をしますから、それに対応したベクトルを加えています。

これらの変換は、以下の Affine 変換の特殊な場合です。各パラメタの意味を図 5 に示します。 ϕ と ψ は、 x 軸及び y 軸からの回転を表しています。 $\phi \neq \psi$ の場合には、捻じれが生じることを表しています。 r と s は、 x 軸及び y 軸方向の縮小です。縮小回転の後に、原点を (e, f) へ移動します。

$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} r \cos \phi & -s \sin \psi \\ r \sin \phi & s \cos \psi \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} + \begin{pmatrix} e \\ f \end{pmatrix} \quad (3.4)$$

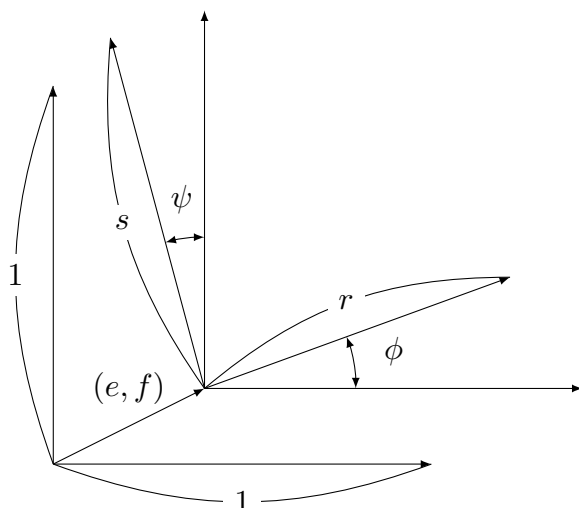


図5 Affine 変換のパラメタ

角度は3時の方向から反時計回りに測ることにします。また、Python の三角関数の引数となる角度は、radian です。0° から 360° が、0 から 2π に対応します。つまり、半径 1 の扇型の円周部分の長さで、角度を表すものが radian です。

4 フラクタルクラス

いよいよフラクタルのクラスを定義しましょう。行うべきことは二つです。第一は、Affine 変換のクラスを作っておくことです。Affine 変換を保持し、それに基づいて指定した座標を変換します。第二は、Affine 変換を使って、フラクタル図形を生成するクラスです。

4.1 Affine 変換クラス

Affine 変換を行うには、行列とベクトルの演算が必要になります。幸い、Python の `numpy.array` を行列とベクトルのように使うことが出来ます。

Affine 変換のクラスをソースコード 4.1 に示します。コンストラクタでは、Affine 変換のパラメタ (r, s, ϕ, ψ, e, f) を与えます。メソッド `translate()` は、引数で与えた 2 次元面内の座標を変換します。座標は、要素を二つ持つタプルです。11 行目の `@` は行列とベク

ソースコード 4.1 Affine 変換のクラス: 一部

```
1 class AF:
2     """
3     Affine Transformation
4     """
5     def __init__(self, r:float, s:float, phi:float, psi:float,
6                 e:float, f:float):
7         self.__m = np.array([
8             [r*math.cos(phi), -s*math.sin(psi)],
9             [r*math.sin(phi), s*math.cos(psi)]]
10        self.__t = np.array([[e], [f]])
11
12    def translate(self, v:tuple[float, float])>tuple[float,float]:
13        """
14        Translate one point
15        """
16        vv = np.array([[v[0]], [v[1]]])
17        r = self.__m @ vv + self.__t
18        return r[0][0], r[1][0]
19
20    def translate_list(self, vl:list[tuple[float,float]]):
21        """
22        Translate list of points
23        """
24        result = list()
25        for v in vl:
26            result.append(self.translate(v))
27        return result
```

トルの積を表しています。メソッド `translateList()` は、引数で与えた 2 次元面内の座標のリストに対して、それぞれを変換します。

4.2 Fractal クラス

Sierpinski ガスケットでは、最初の正三角形が、一辺の長さが $1/2$ の 3 つの正三角形に変換されます。次のステップでは、三つの正三角形がそれぞれ三つに分かれ、一辺の長さが $1/4$ の 9 個の正三角形になります。一回の処理で、三角形の数が 3 倍になります。実は、最初が正三角形である必要はありません。図 4 で示したように、元の点の集合を $1/2$ に縮小し、原点を移動して貼り付けるだけで十分です。

それでは、Fractal クラスをみていきましょう (ソースコード 4.2)。Fractal クラスのコンストラクタでは、Affine 変換のリストと、初期の図形を表す頂点リストを引数とするこ

ソースコード 4.2 Fractal クラス

```
1 class Fractal:
2     """
3     Fractal class
4     """
5     def __init__(self, afList: list[AF], xy = [(0, 0), (0, 1), (1, 1),
6     ↪ (1, 0)]):
7         self.__afList = afList
8         self.__shapes = [xy]
9
10    def iterate(self):
11        """
12        Iterate one step
13        """
14        sp = list()
15        for xy in self.__shapes:
16            for af in self.__afList:
17                sp.append(af.translate_list(xy))
18        self.__shapes.clear()
19        self.__shapes = list(sp)
20
21    def getShapes(self) -> list[pt.Polygon]:
22        """
23        Returns list of shapes as patch
24        """
25        sp = list()
26        for xy in self.__shapes:
27            sp.append(pt.Polygon(xy, fill=True, color='b'))
28        return sp
```

とにします。しかし、Fractal 図形は、初期の図形には依存しないため、デフォルト値として、 1×1 の正方形に対応する頂点リストを設定しておきます。

一回の変換処理を表す `iterate()` では、現在の図形を表す頂点リストに対して、Affine 変換を行い、新たな図形に対応した頂点リストを生成します。

作図のために図形を取り出す際には、`getShapes()` メソッドを使うことにします。このメソッドは、図形を `matplotlib.patch` として返します。

具体的なフラクタル図形を定義するためには、Affine 変換と初期図形を与える必要があります。ソースコード 4.3 は、初期に正三角形を与える、Sierpinski ガスケットの例です。

ソースコード 4.3 Sierpinski ガスケットを定義する

```

1  def Sierpinski():
2      p = math.pi/3
3      xy = [(0, 0), (1, 0), (1./2, math.sin(p))]
4      r = s = 0.5
5      phi = psi = 0.
6      af = [
7          AF(r, s, phi, psi, 0, 0),
8          AF(r, s, phi, psi, 1./2, 0),
9          AF(r, s, phi, psi, 1./4, math.sin(p)/2)
10     ]
11     return Fractal(af, xy)

```

3 行目では初期の三角形を定義しています。Affine 変換のパラメタは、

$$r = s = \frac{1}{2} \quad (4.1)$$

$$\phi = \psi = 0 \quad (4.2)$$

です。それらを使って、3 個の Affine 変換を以下のように定義しています。

7 行目 1/2 に縮小するだけの変換であり、左下の図形に対応

8 行目 1/2 に縮小し、右に 1/2 移動する変換であり、右下の図形に対応

9 行目 1/2 に縮小し、 $(1/3, \sin(\pi/3)/2)$ 移動する変換であり、上中央の図形に対応

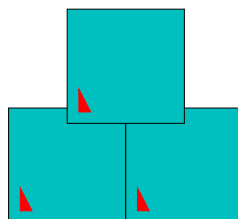


図 6 Sierpinski ガスケットに対応する写像

ソースコード 4.4 は、フラクタル図形を描く部分です。showMap を True とすると、Affine 変換そのものを表示します。図 6 に、ソースコード 4.3 の場合の写像を示します。図 6 では、回転をしていますが、赤い三角形の向きで、回転があれば、その角度がわかります。

ソースコード 4.4 フラクタル図形を描くメイン部分

```
1 tMax = 8
2 isAnimation = False
3 showMap = True
4 fractal = Sierpinski()
5
6 fig= plt.figure(figsize = (10, 10),facecolor = 'w')
7 ax = fig.subplots()
8 if showMap:
9     ax = fig.subplots()
10    ax.set_xlim(0, 1)
11    ax.set_ylim(0, 1)
12    ax.axis("off")
13    for p in fractal.getMap():
14        ax.add_patch(p)
15    plt.savefig("map.pdf")
16    plt.show()
```

showMap を False とすると、繰り返した結果を表示します。例では、8 回繰り返した後の図形を描いています。

初期図形を 1×1 の正方形とする場合をソースコード 4.6 に示します。繰り返しの最初は、初期図形の影響があります。しかし、繰り返し数が増えると、Sierpinski ガスケットになります。

5 Sierpinski ガスケットの拡張

Sierpinski ガスケットの特性を考えると、ソースコード 4.6 のように、3 つの正方形を並べることはあまり意味がないことがわかります。一番大事な点は、 $1/2$ の大きさの図形を 3 個配置するところなのです。そこで、 $1/2$ の大きさの図形を、 1×1 の正方形を 4 等分した領域のいずれかに配置することにしましょう。Fractal クラスのコンストラクタで、2 番目の引数を省略すると、初期図形が 1×1 の正方形となります。

課題 5.1 fractals.ipynb 内の Sierpinski2() で行っていること、特に Affine 変換について理解しなさい。showMap の値を True として写像を確認するとともに、False として、結果を確認しなさい。

一辺の長さを $1/2$ にするとともに、回転を加えると、様々な複雑な図形を生成すること

ソースコード 4.5 フラクタル図形を描くメイン部分: つづき

```
1 elif isAnimation:
2     imgs = []
3     for i in range(tMax):
4         ax = fig.subplots()
5         ax.clear()
6         ax.set_xlim(0, 1)
7         ax.set_ylim(0, 1)
8         ax.axis("off")
9         for p in fractal.getShapes():
10             ax.add_patch(p)
11             imgs.append(ax.get_children())
12             fractal.iterate()
13     ani = animation.ArtistAnimation(fig, imgs, interval = 1000)
14     display(HTML(ani.to_jshtml()))
15 else:
16     ax.set_xlim(0, 1)
17     ax.set_ylim(0, 1)
18     ax.axis("off")
19     for i in range(tMax):
20         fractal.iterate()
21     for p in fractal.getShapes():
22         ax.add_patch(p)
23     plt.savefig('fractal.pdf')
24     plt.show()
```

ソースコード 4.6 Sierpinski ガスケットを定義する: 初期図形は正方形

```
1 def Sierpinski1():
2     p = math.pi/3
3     r = s = 0.5
4     phi = psi = 0.
5     af = [
6         AF(r, s, phi, psi, 0, 0),
7         AF(r, s, phi, psi, 1./2, 0),
8         AF(r, s, phi, psi, 1./4, math.sin(p)/2)
9     ]
10     return Fractal(af)
```

ができます。また、回転を入れると、原点の移動に注意が必要になります。

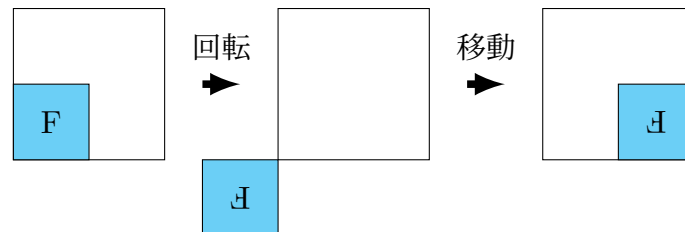


図 7 回転の影響: π 回転した場合

図 7 は、 $1/2$ に縮尺して図形を π 回転させた場合を表しています。左の図で、 $1/2$ した図形を、まず π 回転したものが中央の図です。さらに原点を $(1, 1/2)$ 移動し、右の図となります。

課題 5.2 `fractals.ipynb` 内の `Sierpinski3()` で行っている変換を図 8 に示します。どのような変換かを理解しなさい。

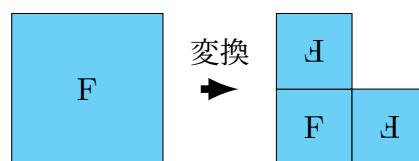


図 8 `Sierpinski3()` に相当する変換

課題 5.3 確認テスト: 図 9 の回転に対するフラクタル図形を定義し、動作を確認しなさい。

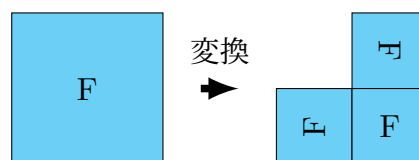


図 9 `Exercise()` に相当する変換

6 アニメーション

Python において、アニメーションとは、紙芝居のようにイメージを次々に示すものを指しています。ソースコード 4.5 において、`isAnimation` を `True` とすることで表示します。

`Axes` クラスのインスタンス `ax` を毎回作成し、そこに作図をしています。その作図結果を、`imgs` という配列に、作図結果を追加しておきます。最後にアニメーションとして再生します。

付録 A ベクトルと行列

2次元の空間を考えます。ベクトル

$$\vec{v} = \begin{pmatrix} x \\ y \end{pmatrix} \quad (\text{付録 A.1})$$

とは、原点から座標 (x, y) への矢印、つまり長さや向きのある量として定義します。

2×2 の行列として

$$A = \begin{pmatrix} a & b \\ c & d \end{pmatrix} \quad (\text{付録 A.2})$$

を考えます。ベクトルと行列の積を定義します。

$$A\vec{x} = \begin{pmatrix} a & b \\ c & d \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} ax + by \\ cx + dy \end{pmatrix} \quad (\text{付録 A.3})$$

積の順序が重要であることを指摘しておきます。

行列 A はベクトルの変換と考えることができます。例えば

$$A = \begin{pmatrix} a & 0 \\ 0 & b \end{pmatrix} \quad (\text{付録 A.4})$$

とすると、この行列は x 方向に a 倍、 y 方向に b 倍することを表しています。

$$A\vec{x} = \begin{pmatrix} ax \\ by \end{pmatrix} \quad (\text{付録 A.5})$$

また、以下は角度 ϕ の回転を表しています。

$$A = \begin{pmatrix} \cos \phi & -\sin \phi \\ \sin \phi & \cos \phi \end{pmatrix} \quad (\text{付録 A.6})$$

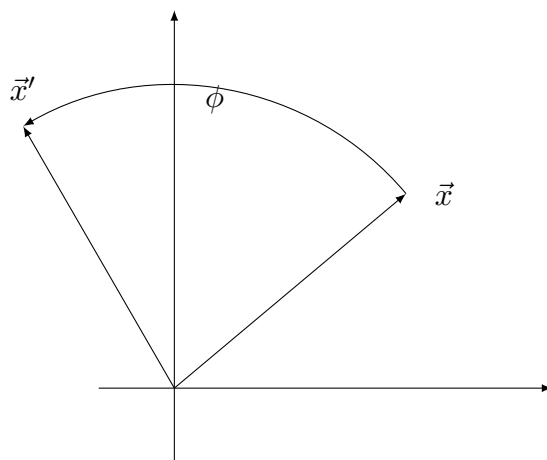


図 10 ベクトルの回転

$$\vec{x}' = A\vec{x} = \begin{pmatrix} x \cos \phi - y \sin \phi \\ x \sin \phi + y \cos \phi \end{pmatrix} \quad (\text{付録 A.7})$$

ベクトルと行列は、「線形代数」と言う理工系の学科では初年次必修の科目の内容です。空間内のベクトルの変換、線形連立方程式の解法、線形な組合せ最適化問題など、理工学のあらゆる分野で使われます。もちろん、データサイエンスや機械学習を理解する上でも必須の知識・技術です。高校数学では、「数学 C」に行列の初歩が含まれていますから、基礎の部分はそれほど難しくありません。

付録 B 三角関数

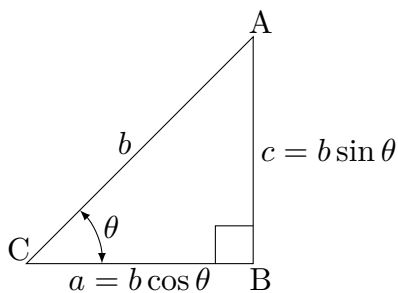


図 11 三角関数

三角関数は、角度と辺の長さを結びつけます。図 11 を見てください。 $\angle ACB = \theta$ である直角三角形を考えます。辺 AC の長さを b とすると、直角を挟む二辺の長さは、角度 θ

の三角関数で表すことができます。

$$a = b \cos \theta \quad (\text{付録 B.1})$$

$$c = b \sin \theta \quad (\text{付録 B.2})$$

$\sin \theta$ を正弦、 $\cos \theta$ を余弦と日本語では言います。直角を挟む二辺の長さの比は

$$\frac{c}{a} = \frac{\sin \theta}{\cos \theta} = \tan \theta \quad (\text{付録 B.3})$$

となり、正接と呼びます。