Chapter 5

表示空間(平面)と2次元座標変換

5.1 座標系とアフィン変換

常に `Canvas` の大きさを意識しながら，図形を描くのは適切ではない．たとえば 600×600 ピクセル程度の大きさに合うようにプログラムの数値を書き換えるのではなく，プログラムの側でウィンドウにあった大きさに変換すればよい．表示ウィンドウに合わせた座標系は，**ウィンドウ座標系** (window coordinate system) ないし**スクリーン座標系** (screen coordinate system) と呼ばれる．これに対して，図形を扱う座標系は**ワールド座標系** (world coordinate system) と呼ばれる．

一方，位置の表現である点 $\boldsymbol{p}$ を変換する方法に，**アフィン変換** (affine transformation) がある．アフィン変換 $\mathcal{A}$ は，正方行列 M とベクトル $\boldsymbol{v}$ によって，$\mathcal{A}\boldsymbol{p} = M\boldsymbol{p} + \boldsymbol{v}$ という形式で表現される変換である．アフィン変換は閉じた変換である．すなわち，

$$\mathcal{A}_2\left(\mathcal{A}_1\boldsymbol{p}\right) = M_2\left(M_1\boldsymbol{p} + \boldsymbol{v}_1\right) + \boldsymbol{v}_2 = \left(M_2M_1\right)\boldsymbol{p} + \left(M_2\boldsymbol{v}_1 + \boldsymbol{v}_2\right) = M'\boldsymbol{p} + \boldsymbol{v}'$$

というように，アフィン変換の合成変換はアフィン変換になる．

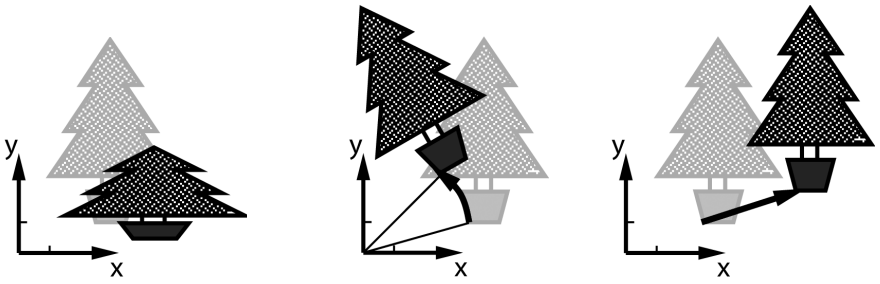

Figure 5.1: アフィン変換の基本変換 - 左からスケール変換，回転変換，平行移動

アフィン変換の基本変換には，図 5.1 に示す**スケール変換** (scaling) と**回転変換** (rotation)，**平行移動** (translation) などがある．多くのアフィン変換は，この 3 種類の基本変換の合成変換として表現できる[1]．2 次元の場合，これらの基本変換は，以下のような行列ならびにベクトルとなる．

- スケール変換：　x, y 成分を s_x, s_y 倍する．

$$M = \begin{bmatrix} s_x & 0 \\ 0 & s_y \end{bmatrix}, \quad \boldsymbol{v} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

- 回転変換：　原点を中心に θ 回転する．

$$M = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix}, \quad \boldsymbol{v} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

[1] 2 次元空間のアフィン変換であれば，これに加えてせん断 (shearing) と呼ばれる基本変換がある．

- 平行移動： $\boldsymbol{v} = [\ t_x \quad t_y\ ]^{\mathsf{T}}$ だけ平行移動する．

$$M = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}, \quad \boldsymbol{v} = \begin{bmatrix} t_x \\ t_y \end{bmatrix}$$

5.2 2 次元ベクトルと変換行列

点とベクトルとは，異なるものとして扱うべき場面もある．たとえば，ベクトル同士の和は合成ベクトルとなるが，2 点の座標値の和をとることには，数学的な意味はないと言える．しかし，プログラムのレベルでは，2 次元の点とベクトルはともに 2 つの数値の組として表現し，加減算などの演算を用意すると非常に便利である．本節のプログラムは，2 次元ベクトルと 2×2 行列，とそれらの間の演算を提供する．実際の 2 次元ベクトルや 2×2 行列の計算には，NumPy として知られる numpy モジュールの array クラスを利用する．このプログラムは単体で実行されるものではなく，次節以降のプログラムで利用される．

numpy.array クラス： 実数配列のクラス

ベクトルや行列の計算に広く利用されるクラス．NumPy には行列専用の matrix クラスなどもあるが，array クラスで基本的なベクトルや行列の演算ができる．

+, -, *, / 四則演算

基本的に要素ごとの四則演算を行う．+や-などの計算は，ベクトルの加算や減算となる．

dot 内積メソッド

同じ長さの配列の間で，ベクトルの内積に相当する計算を行う．行列とベクトルの積や行列同士の積などにも利用できる．

例１： 2 次元ベクトルの定義 — vectorMatrix.py

ワールド座標系は実数の組で表現されるものとする．さらにここでは，今後のプログラムのために，ベクトルのユークリッドノルム（L2 ノルム）を求める norm 関数，回転行列と拡大縮小行列を生成する rotMatrix 関数と scaleMatrix 関数も定義している．

```
import math                                 # math モジュールの import
import numpy as np                          # numpy モジュールの import (np で)

def norm(v):                                # ベクトルのノルム計算
  '''
  v - ベクトル
  ベクトルの大きさを返す
  '''
  v = np.array(v)                           # numpy.array への変換
  return (v.dot(v))**0.5                    # 内積の平方根

def rotMatrix(t):                           # 回転行列の作成
  '''
  t - 回転角度 (radian 単位)
  原点を中心として t だけ回転する 2x2 行列を作成して返す
  '''
  return np.array(((math.cos(t), -math.sin(t)), (math.sin(t), math.cos(t))))
                                            # numpy.array の回転行列
def scaleMatrix(s):                         # 拡大縮小行列の作成
  '''
  s - 拡大縮小の係数
  x,y 軸方向ともに s 倍に拡大縮小する 2x2 行列を作成して返す
  '''
  return np.array(((s, 0), (0, s)))         # numpy.array の拡大縮小行列

def main():                                 # main 関数
  vec1 = np.array((1, -1))                  # ベクトル (1, -1)
  vec2 = np.array((3, 4))                   # ベクトル (3, 4)
  rot = rotMatrix(math.pi/2.0)              # pi/2 回転の回転行列
  inv = rotMatrix(-math.pi/2.0)             # -pi/2 回転の回転行列
```

```
  scl = scaleMatrix(2.0)                  # 2 倍の拡大縮小行列
  print('|', vec2, '| =', norm(vec2))     # (3, 4) のノルム
  print(vec1, '+', vec2, '=', vec1+vec2) # (1, -1) + (3, 4)
  print(vec2, '-', vec1, '=', vec2-vec1) # (3, 4) - (1, -1)
  print(vec2, '* 2 =', vec2*2)            # (3, 4) * 2
  print(vec1, '*', vec2, '=', vec1*vec2) # (1, -1) * (3, 4) (要素積)
  print(vec1, '.', vec2, '=', vec1.dot(vec2)) # (1, -1) . (3, 4) (内積)
  print('rotate   (', vec2, ') =', rot.dot(vec2)) # (3, 4) の pi/2 回転
  print('scale&rot(', vec2, ') =', scl.dot(rot).dot(vec2))
                                          # (3, 4) の pi/2 回転 & 2 倍拡大
  print('inv&rot  (', vec2, ') =', inv.dot(rot).dot(vec2))
                                          # (3, 4) の pi/2 回転 & -pi/2 回転
if __name__ == '__main__':                # 起動の確認 (コマンドラインからの起動)
  main()                                  # main 関数の呼出
```

例２： 仮想の表示空間を持つクラス MyCanvas — myCanvas.py

MyCanvas を用いることで，任意のワールド座標系における描画プログラムを簡潔に書ける．通常の Canvas と同様にコールバック関数を登録する bind メソッドや，ルートウィンドウのメインループを起動する mainloop メソッドがある．標準では，600×600 のウィンドウ内に，ワールド座標系において原点を中心にした $-1 \leq x, y \leq 1$ の範囲の描画を行なう．原点位置や描画範囲を変更するメソッドとして，setOrigin メソッドと setRange メソッドがある．ワールド座標系からウィンドウ座標系への変換は，x メソッドと y メソッドによって求められる．逆のウィンドウ座標系からワールド座標系への変換は，point メソッドによって計算できる．これは，マウスイベントを利用する際に，ウィンドウ内のカーソル位置をワールド座標系に変換するために必要となる．また，背景を白で塗りつぶす clear メソッド，円を描画する drawCircle メソッド，線分や折れ線を描画する drawPolyine メソッド，多角形を描画する drawPolygon メソッド，マーカ（正方形）を描画する drawMarker メソッドなどが用意されている．

```
from tkinter import *                     # tkinter モジュールの import
import numpy as np                        # numpy モジュールの import (np で)

class MyCanvas(object):                   # MyCanvas クラスの定義
  def __init__(self, width = 600, height = 600, xo = 300, yo = 300, r = 2.0):
                                          # 初期化メソッド
    '''
    width, height - canvas の幅と高さ , 省略時 600, 600
    xo, yo        - 描画原点の canvas (スクリーン) 上の位置 , 省略時 300, 300
    r             - 描画領域 (世界座標系で) の範囲 , 省略時 2.0
    canvas の作成と初期化を行う
    '''
    self.w, self.h = (width, height)      # canvas の幅と高さ
    self.xo, self.yo = (xo, yo)           # 描画原点の canvas (スクリーン) 上の位置
    self.r = r                            # 描画領域 (世界座標系で) の範囲
    self.s = min(self.w, self.h) / self.r # 描画領域の拡大縮小率
    self.mr = 2                           # marker (制御点用) の半径
    self.root = Tk()                      # ルートフレームの作成
    self.canvas = Canvas(self.root, width = self.w, height = self.h)
    self.canvas.pack()                    # canvas の作成と配置確定

  def bind(self, event, func):            # コールバック関数の登録メソッド
    '''
    event - イベント
    func  - コールバック関数
    イベントに対するコールバック関数を登録する
    '''
    self.canvas.bind(event, func)         # event のコールバック関数

  def mainloop(self):                     # 実行ループの開始メソッド
    '''
    実行ループを開始する
    '''
    self.root.mainloop()                  # ルートフレームの実行ループ開始

  def setOrigin(self, x, y):              # 描画原点の指定メソッド
```

```
    '''
    x, y - 描画原点の canvas（スクリーン）上の位置
    描画原点の位置を記録する
    '''
    self.xo, self.yo = (x, y)            # 描画原点の位置の記録

  def setRange(self, r):                 # 描画領域の範囲指定メソッド
    '''
    r - 描画領域（世界座標系で）の範囲
    描画領域の範囲（長さ）を指定する
    '''
    self.r = r                           # 描画領域（世界座標系で）の範囲
    self.s = min(self.w, self.h) / self.r # 描画領域の拡大縮小率

  def point(self, x, y):                 # 点データの作成メソッド
    '''
    x, y - スクリーン上の座標値
    対応する世界座標系での点データ（numpy.array）を作成して返す
    '''
    return np.array(((x-self.xo) / self.s, (self.yo-y) / self.s))

  def x(self, p):                        # 点のスクリーン上での x 座標計算メソッド
    '''
    p - 世界座標系での点データ
    対応するスクリーン上での x 座標値を返す
    '''
    return self.s*p[0] + self.xo         # スクリーン上での x 座標

  def y(self, p):                        # 点のスクリーン上での y 座標計算メソッド
    '''
    p - 世界座標系での点データ
    対応するスクリーン上での y 座標値を返す
    '''
    return -self.s*p[1] + self.yo        # スクリーン上での y 座標

  def inside(self, p):                   # 描画領域内かの判定メソッド
    '''
    p - 世界座標系での点データ
    描画領域内にあるか否かを bool 型で返す
    '''
    return 0 <= self.x(p) <= self.w and 0 <= self.y(p) <= self.h

  def clear(self):                       # クリアメソッド
    '''
    背景を白長方形でで描画（クリア）する
    '''
    self.canvas.create_rectangle((2, 2), (self.w+3, self.h+3),
                                 outline='', fill='white')

  def drawCircle(self, c, r, fill='', outline='black'): # 円の描画メソッド
    '''
    c             - 円の中心
    r             - 円の半径
    fill, outline - 内部の色, 輪郭の色, 省略時 なし, 黒
    円の円周を描画する
    '''
    dr = self.s * r                      # スクリーン上での半径
    xc, yc = (self.x(c), self.y(c))      # スクリーン上での中心
    self.canvas.create_oval((xc-dr, yc-dr), (xc+dr+1, yc+dr+1),
                            fill=fill, outline=outline) # 円の描画

  def drawPolyline(self, ps, color='black'): # 折れ線の描画メソッド
    '''
    ps    - 折れ線の点列
    color - 描画色, 省略時 黒
    点列を結んだ折れ線（開いた線）を描画する
    '''
    points = []                          # スクリーン上での点列リストの初期化
    for i in range(len(ps)):             # 点の変換の反復
```

```
      points.append((self.x(ps[i]), self.y(ps[i]))) # 変換した点の追加
    self.canvas.create_line(tuple(points), fill=color) # 折れ線の描画

  def drawPolygon(self, ps, fill='gray90', outline='black'): # 多角形の描画メソッド
    '''
    ps             - 多角形の点列
    fill, outline - 内部の色，輪郭の色，省略時 灰色，黒
    点列を結んだ多角形（閉じた線と内部）を描画する
    '''
    points = []                              # スクリーン上での点列リストの初期化
    for i in range(len(ps)):                 # 点の変換の反復
      points.append((self.x(ps[i]), self.y(ps[i]))) # 変換した点の追加
    self.canvas.create_polygon(tuple(points), fill=fill, outline=outline)
                                             # 多角形の描画

  def drawMarker(self, p, fill='black', outline=''): # marker の描画メソッド
    '''
    p       - marker の描画位置の点
    fill, outline - 内部の色，輪郭の色，省略時 黒，なし
    指定された位置に marker を描画する
    '''
    x, y = (self.x(p), self.y(p))            # スクリーン上での座標
    self.canvas.create_rectangle((x-self.mr, y-self.mr),
                                 (x+self.mr+1, y+self.mr+1),
                                 fill=fill, outline=outline) # marker の描画
```

5.3 MyCanvas を用いたプログラム

例１： 円の描画 — myCircle.py

myCanvas モジュールの MyCanvas クラスを利用することで，ワールド座標系における描画プログラムが（2.2 節の例 3 に比べて）簡潔に記述できる．このプログラムでは，原点を中心として，半径 0.8 の円を描いている．

```
import numpy as np                           # numpy モジュールの import（np で）
import sys                                   # sys モジュールの import
import math                                  # math モジュールの import
from myCanvas import MyCanvas                # myCanvas モジュールの import

def circle(cen = (0, 0), r = 0.8):
  '''
  cen - 中心座標，省略時 (0, 0)
  r   - 半径，省略時 0.8
  円周上（正多角形）の n 頂点座標（タプル）を返す
  '''
  if len(sys.argv) > 1:                      # シェル引数がある場合
    num = sys.argv[1]                        # 第 1 引数を頂点数の文字列
  else:                                      # シェル引数がない場合
    num = input('# of points -> ')           # 頂点数の文字列を入力
  n = int(num)                               # 頂点数
  p = []                                     # 点列リストの初期化（空リスト）
  for i in range(n):                         # 頂点座標計算の反復
    t = 2 * math.pi * i / n                  # 2 pi の n 等分
    p.append(np.array((r*math.cos(t)+cen[0], r*math.sin(t)+cen[1])))
  return tuple(p)                            # タプルにして return

def display(canvas, points):                 # （円の）描画関数
  '''
  canvas - 描画する canvas
  points - 円周上の頂点列
  円を描画する
  '''
  canvas.drawPolygon(points, fill='')        # 多角形描画メソッドの呼出

def main():                                  # main 関数
```

```
  canvas = MyCanvas()                    # myCanvas の作成
  points = circle()                      # 頂点作成関数（circle）の呼出
  display(canvas, points)                # 描画関数（display）の呼出
  canvas.mainloop()                      # ルートフレームの実行ループ開始

if __name__ == '__main__':               # 起動の確認（コマンドラインからの起動）
  main()                                 # main 関数の呼出
```

例２： 円によるカージオイドの描画 — myCardioidCircle.py

例１の myCircle モジュールの circle 関数を利用して，円でカージオイドを包絡する．vectorMatrix モジュールのノルム計算（norm 関数）によって，(2.2 節の例５に比べて）プログラムが簡潔になる．

```
from myCanvas import MyCanvas            # myCanvas モジュールの import
from vectorMatrix import norm            # vectorMatrix モジュールの norm 関数の import
import myCircle                          # myCircle モジュールの import

def display(canvas, points):             # (cardioid の）描画関数
  '''
  canvas - 描画する canvas
  points - 円周上の頂点列
  基円の上に円で包絡された cardioid を描画する
  '''
  for i in range(1, len(points)):        # 円描画の反復（頂点数-1）
    r = norm(points[i] - points[0])      # 半径（0 番頂点との距離）
    canvas.drawCircle(points[i], r)      # 円の描画

def main():                              # main 関数
  canvas = MyCanvas()                    # myCanvas の作成
  points = myCircle.circle((0.25, 0), 0.33) # 頂点作成関数（myCircle.circle）の呼出
  myCircle.display(canvas, points)       # 基円の描画関数（myCircle.display）の呼出
  display(canvas, points)                # 描画関数（display）の呼出
  canvas.mainloop()                      # ルートフレームの実行ループ開始

if __name__ == '__main__':               # 起動の確認（コマンドラインからの起動）
  main()                                 # main 関数の呼出
```

例３： マウスによる線分の描画 — myRubberband.py

4.3 節の例３の rubberband のプログラムも MyCanvas クラスを利用することで，ワールド座標系におけるプログラムとなる．マウスカーソルの位置を，ウィンドウ座標系からワールド座標系に point メソッドで変換していることに注意せよ．

```
from myCanvas import MyCanvas            # myCanvas モジュールの import

def pressed(event):                      # Button1 pressed コールバック関数
  global canvas, start                   # 大域変数 canvas, start
  start = canvas.point(event.x, event.y) # press 時の点を記録

def dragged(event):                      # Button1 dragged コールバック関数
  global canvas, start                   # 大域変数 canvas, start
  canvas.clear()                         # canvas の背景クリア
  canvas.drawPolyline((start, canvas.point(event.x, event.y))) # 線分の描画

def main():                              # main 関数
  global canvas                          # 大域変数 canvas
  canvas = MyCanvas(width=400, height=400) # canvas の作成
  canvas.bind('<Button-1>', pressed)     # Button1 pressed コールバック関数
  canvas.bind('<B1-Motion>', dragged)    # Button1 dragged コールバック関数
  canvas.mainloop()                      # ルートフレームの実行ループ開始

if __name__ == '__main__':               # 起動の確認（コマンドラインからの起動）
  main()                                 # main 関数の呼出
```

章末課題

ダイアモンドパターンやネフロイドの描画

第 2 章の章末問題と同様に myCircle モジュールを利用して，ダイアモンドパターンやネフロイドを描画するプログラムを作成してみよ．当然のことながら，vectorMatrix モジュールを利用し，ワールド座標系における計算・描画を行うこと．

マウスによる円の描画

第 4 章の章末問題にあったマウスによって円を描くプログラムも，MyCanvas クラスを用いることによって，ワールド座標系でのプログラムにすることが可能である．マウスボタンを押した位置を中心とし，マウスボタンを離した点を通る円を描画するプログラムである．第 4 章の章末問題の解答に加えて，5.3 節の例 3「マーカの描画」を参考にするとよい．さらに，図 5.2 のように半径が 1/2 倍と 3/2 倍の 3 つの円を同時に描画するプログラムを考えると，MyCanvas クラスや vectorMatrix モジュールの有用性が感じられるだろう．

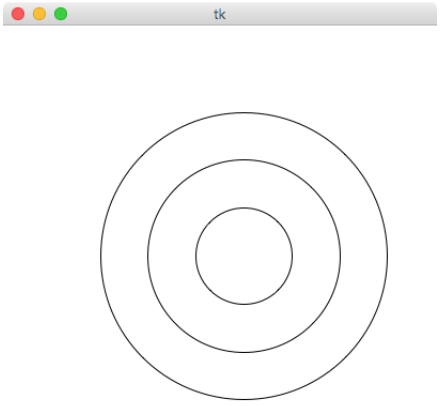

Figure 5.2: マウスによる円の描画