Pythonにおける動的グラフィックス

経営学部教授　有澤健治

はじめに

ここではPythonによる動的グラフィックスを実現するプログラミングの方法を解説する。プログラムはLaser5 Linux 6.4およびWindows2000上のPython 2.0で動作の確認を行っている。動的グラフィックスを実現する上で中心的な役割を果たしているのはCanvasクラスのafterメソッドである。

afterメソッドに関してはJ.E.Graysonも紹介しているが(文献[6],p.433)、残念ながらその解説は余りにも簡単に済ませられている。むしろ、解説はされていないものの、UNIX版の配布プログラムに添付されているデモプログラムの方が役に立つ。配布プログラムではグラフィックスを使用したデモプログラムは以下の2つのディレクトリの中に納められている。

```
Python-2.0/Demo/tkinter/guido/
Python-2.0/Demo/tkinter/matt/
```

これらの中でafterメソッドを使用したプログラムは以下の5つである。

```
guido/hanoi.py
guido/sortvisu.py
matt/animation-simple.py
matt/animation-w-velocity-ctrl.py
matt/pong-demo-1.py
```

ここに、hanoi.pyはハノイの塔のリング移動を示すアニメーションである。sortvisu.pyは種々のソートアルゴリズムの動作を示すアニメーションである。これらは共に完成度の高いプログラムなので入門用としては適さない。入門用としてはmattの下にあるMatt Conwayの作品が役に立つ。animation-simple.pyは小さな正方形をキャンバス内で移動させるだけの簡単なプログラムである。このプログラムはafterメソッドの使い方を示すためのものである。animation-w-velocity-ctrl.pyはanimation-simple.pyを発展させたもので、Scaleウィジェットによって正方形の移動速度を制御している。

pong-demo-1.pyはanimation-w-velocity-ctrl.pyをさらに発展させたもので、ボールが壁に跳ね返りながら運動するアニメーションである。

ところでMattのプログラムはいずれも全てを品良くクラスオブジェクトとして纏めている。そのためにプログラムの理解のためにはクラス定義法についての知識を必要とし、さらにafterメソッドの本質の理解を困難にしている。ここでは読者に動的グラフィックスのエッセンスだけを理解して

貰うためにあえてクラス定義法を使用しないで初等的なプログラミングスタイルをとることとした。

読者は以下の前提知識を有していると想定されている。

1. Python の文法を知っている事。
2. Canvasクラスの使い方を知っている事。

Python の文法は文献[3]または文献[10]に丁寧に纏められている。
Canvasクラスの使い方は文献[11]に纏められている。また文献[11]の内容は文献[10]にそのまま載っている。

筆者は片仮名と英単語、さらに英単語の表示におけるフォントを次ぎの様に使い分けている。「キャンバス」のように片仮名で書いているのは普通名詞を意味している。他方 Canvas のように英単語をそのまま書いているのは固有名詞あるいは商標登録されている名詞である。さらに Canvas のように固定幅フォントで表示されていれば、それはプログラムの中で使用される文字列であり、それがプログラムの中に現れた時にはその文字列をそのまま使用しなくてはならないことを意味している。

この記事を書き終えた頃に Python 2.1 がリリースされた(2001年4月)。幸い、2.1 版の改訂内容は今回の記事には影響を与えなかった様である。

目次

第1節　afterメソッド

Pythonの動的グラフィックスのプログラムではCanvasクラスのafterメソッドが中心的な役割を演じている。この節の目的はafterメソッドがタイマーを設定している事を明らかにしながら、afterメソッドの使い方を示す事にある。

cをキャンバスとするとき、afterメソッドは

```
c.after(t,f,a1,a2,...)
```

の形で使用する。ここにtは1/1000秒を単位とした時間であり、fは関数名、a1,a2,...は関数fに与える引数である。(引数を必要としない関数ではa1,a2,...を省略できる。)

このメソッドによって、これが実行されたt/1000秒後に関数fが実行されるように、キャンバスcにタイマーが仕掛けられる。

afterメソッドはタイマーである

afterメソッドがsleepのような関数、即ち、ある時間の間プログラムの進行を停止させるような関数ではない事は譜1aの実験プログラムによって示される。

```
from Tkinter import *
def draw():
    print 'moved'
    c.move(p,10,0)
c = Canvas(width=500,height=300)
c.pack()
c.create_text(250,150,text="Animation Demo",font="Times 60 italic",fill="red")
p=c.create_oval(50,50,200,250)
print 'start'
for n in range(0,30):
    c.after(1000*n,draw)
print 'done'
mainloop()
```

譜1a. 楕円を動かすプログラム

注意: 実際にはこのようなafterメソッドの使い方はしない

このプログラムは

```
for n in range(0,30):
    c.after(1000*n,draw)
print 'done'
```

によって、0秒後、1秒後、2秒後、...、29秒後に関数drawが実行されるようにタイマーを仕掛けている。

プログラムを実行すると、図1aの画面が得られるが、コンソールにはこの前後のprint文

```
print 'start'
```

と

```
print 'done'
```

による出力を観測できる。実験すれば分かるように、'done' は 'start' が出力された後、直ちに出力される。つまり after メソッドはプログラムの進行を停止させるような関数ではないのである。

さて、仕掛けられたタイマーによって、関数 draw は1秒間隔で30回実行される。そのたびに関数 draw の中の

```
print 'moved'
```

が実行され、その出力をコンソールウィンドウに見ることができる。そして 'moved' が出力されるたびに、キャンバス上の楕円が少しずつ右方向に移動する(図1a)。この様子もプログラムを実行すれば直ちに確認できる。

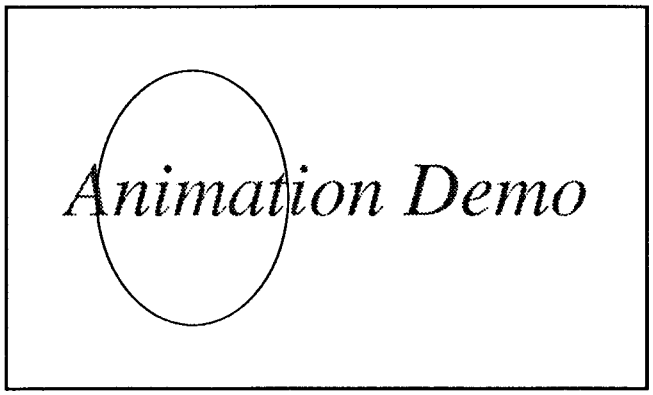

図1a. この図の楕円が右方向に動く
Windows 環境ではこの図よりも大きな文字が表示される
であろう。(この問題については第4節を見よ。)
Linux で画面の文字がギザギザする場合にはアウトラインフォント
を設定する必要がある。設定に関しては文献[20]を参照せよ。

after メソッドの使い方

譜1のプログラムはあくまで実験のためのプログラムである。実際のプログラムでは予定される全ての時刻に対して、このようにあらかじめタイマーを設定するようなな方式は採られないであろう。なぜならこの方法は Python のタイマー管理ルーチンに対して余計な負荷をかけるばかりかタイマーによって起動された関数が何らかの理由で次の予約時刻までに終了しない場合に問題が発生する可能性があるからである。

実際のプログラムではむしろ図1bに示すように予約時刻は1個に留め、関数の実行が完了した時点で次の予約をとる。この方法では図1b から分かる様に関数の実行に必要な時間だけの遅れが生じる。しかしながら関数の実行に多くの時間が取られても多重に関数が実行される恐れはないので安全な方法と言える。

この考えに基づいて譜1aのプログラムを書き換えると譜1bのようになる。

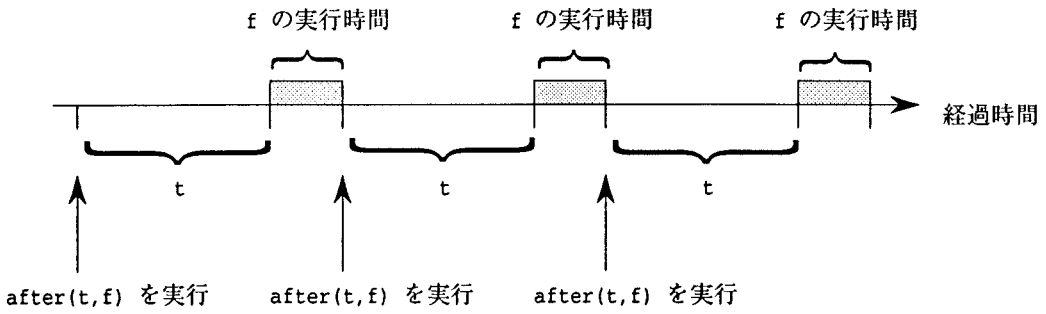

図1b. after メソッドの使い方に伴う遅延を示す概念図

```
from Tkinter import *
def animate(n):
    if n <= 0: return
    print 'moved'
    c.move(p,10,0)
    c.after(1000,animate,n-1)
c = Canvas(width=500,height=300)
c.pack()
c.create_text(250,150,text="Animation Demo",font="times 60 italic",fill="red")
p=c.create_oval(50,50,200,250)
animate(30)
mainloop()
```

譜1b. 楕円を動かすプログラム
画面を更新するごとに after メソッドを実行する

関数 animate が実行されると、その最後の

```
c.after(1000,animate,n-1)
```

でタイマーが仕掛けられ、直ちに戻る。そして

```
mainloop()
```

の中でタイマーが起動されるのを待つのである。

なお譜1bのプログラムでは関数 animate を30回実行させるために引数を渡している。しかしながらアニメーションを扱う問題では前もって回数を指定する方法は好まれないであろう。むしろ他の方法、例えばこの問題では、動いているアイテムが画像から消えた時にアニメーションを終了させるような方法の方が好まれるであろう。

第2節　アイテム識別子とタグ

アイテム

キャンバスに描かれる個々の基本図形をアイテムと言う。Tkinterには基本図形を描く以下のメソッドが存在し、これらのメソッドによって描かれる個々の図形がアイテムである。ここに挙げたメソッドをアイテム生成メソッドと言う。

create_line	(Line)	線分、折れ線、曲線
create_rectangle	(Rectangle)	矩形
create_polygon	(Polygon)	多角形
create_oval	(Oval)	楕円
create_arc	(Arc)	円弧
create_text	(CanvasText)	文字列
create_bitmap	(Bitmap)	画像(2色)
create_image	(ImageItem)	画像(カラー)
create_window	(Window)	ウィンドウ

ここに（）の中はCanvas モジュールを利用した場合のアイテムを生成する命令である。Canvasモジュールを使用したグラフィックスは文献[10]あるいは[11]に解説されている。Python-2.0のLib/lib-tk/Canvas.pyの冒頭のコメントによると、Canvas モジュールは廃止される予定であるので、今後はTkinterの中で定義されているCanvasクラスの生成メソッドを使用する必要がある(注1)。

注1: Tkinterへの書き換えは次のように行えばよい。
cをキャンバスとするとき、Line(c,....)は単にc.create_line(....)と書き換えればよい。ここに....は座標やオプションの並びであり、この部分は書き換えにおいて不変である。
Rectangleなど他の命令についても同様である。もちろん

```
From Canvas import *
```

は不要となる。

アイテム識別子

アイテムがキャンバスに生成された順序に従って、アイテムは1,2,3,...と自然数によって番号付けられる。この番号をアイテム識別子と言う。アイテム生成メソッドはアイテム識別子を返す(注2)。

注2: Canvasモジュールの場合にはアイテムを生成する命令はインスタンスを返す。Tkinterの Canvasクラスのメソッドは、このインスタンスをアイテム識別子の代わりに使用してもよいように設計されている。

ディスプレイリスト

キャンバス上の任意の2つのアイテムが重なると一方のアイテムは他方のアイテムの下に隠れる。

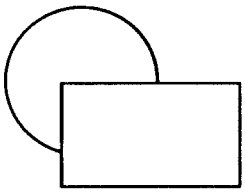

図2. この図では矩形が円の上位にある。

このようにキャンバスのアイテムには表示における上下関係が存在する。キャンバスの全てのアイテムを下にあるものから順に並べたリストをディスプレイリストと言う。後に生成されたアイテムは上に重ねられるので、ディスプレイリストは一般にアイテム識別子(1, 2, 3, ...)の順である。しかしながらこのリストは変更可能である。現在の実際のディスプレイリストは、cをキャンバスとするとき、

```
c.find_all()
```

で得られる。このメソッドが返すアイテムの一覧は、下位のアイテムから上位のアイテムの順に並べられている。

タグ

アイテムの集合に名前を与える事ができる。この名前をタグ(注1)と言う。タグを使用する事によって複数のアイテムをまとめて移動したり、一様に大きさを変更したり、色等の属性を同時に変更したり、まとめて削除したりできる。

注1: タグとは付箋とか名札の意味である。従って「アイテムにはタグを付け、タグに名前を付ける」ことができると言うべきであろう。

タグは個々のアイテムに付ける。同じ名前を与えられたアイテムが1つの集合を形成すると看做される。また1つのアイテムに複数のタグを付けることもできる。

いろいろな方法でタグを付ける事ができるが、初等的にはアイテムの生成メソッドの tags オプションを使用する。例えばcをキャンバスとすると、

```
c.create_rectangle(50,100,200,250,tags='rect')
```

によって、この矩形に 'rect' という名前のタグが付く。

複数の名前、例えば 'a' と 'b' と 'c' を与えるには

```
tags='a b c'
```

のように1つの文字列の中に空白で区切って名前を指定するか、あるいは

```
tags=('a', 'b', 'c')
```

のように、タプル形式で名前を指定する。前者の場合には名前の中に空白文字を含めることはでき

ないが、後者のようにすれば可能である。

定義済タグ 'all' と 'current' が存在する。'all' はキャンバスの全てのアイテムの集合を表す。'current' はマウスカーソルで指されているアイテムを表す。

譜2にタグを利用したプログラム例を載せる。このプログラムを実行すると画面は図2aから図2b へ変化する。このプログラムでは2つの矩形に対してタグ 'rect' を与え、一緒に動かしている。

```
from Tkinter import *
def animate():
    if len(c.find_overlapping(0,0,500,300)) < 2: c.quit()
    c.move(p,1,0)
    c.move('rect',2,0)
    c.after(50,animate)
c = Canvas(width=500,height=300)
c.pack()
c.create_text(250,150,text="Animation Demo",font="times 60 italic",fill="red")
p=c.create_oval(50,50,200,200)
c.create_rectangle(50,100,200,250,tags='rect')
c.create_rectangle(60,110,210,260,tags='rect')
animate()
mainloop()
```

譜2. タグを利用したプログラム

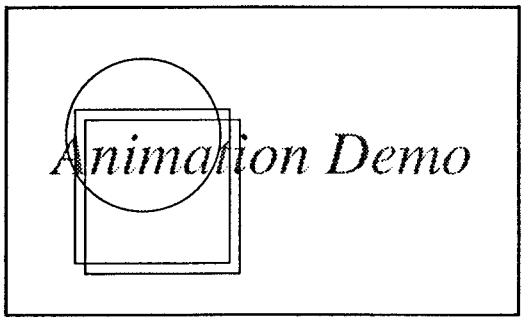

図2a. 譜2の実行画面(最初の頃)

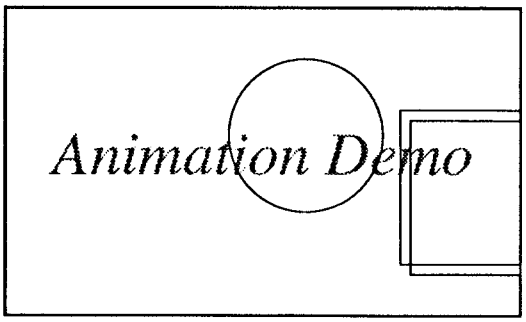

図2b. 譜2の実行画面(時間が経過した画面)

move は図形を平行移動するメソッドであるが、対象とする図形は

```
c.move(p,1,0)
```

のようにアイテム識別子を指定しても、あるいは

```
c.move('rect',2,0)
```

のようにタグを指定してもどちらでも構わない。

なお譜2の

```
if len(c.find_overlapping(0,0,500,300)) < 2: c.quit()
```

はキャンバスに見えるアイテムが1つになったら(つまり文字列 "Animation Demo" だけになったら)終了する事を指示している。

第3節　移動とスケール変換

長さの単位

Tkinterにおける長さの基本単位は画素(Pixel)であるが、数字の後に以下の1文字を添える事によって他の単位を利用する事もできる。

- m　ミリメートル
- c　センチメートル
- i　インチ
- p　ポイント

これらの単位は文字列形式で指定する必要がある。例えば、インチを単位とする場合には

```
c=Canvas(width='3i', height='2i')
```

のように指定する。なお、1ポイントは1/72インチである。

画面の画素密度(単位長さ当たりのピクセル数)はWindowsでは96、UNIXでは72であると想定されている。従って画素と他の単位を混在した場合には画面に表示される図形たちの相互の大きさの関係はOSによって異なる。この問題の解決法については第4節で議論する。

PythonのTkinterはPostScriptプリンタを利用する。PostScriptでは画面の画素数は72であると想定されている。従ってWindowsの場合には印刷結果に問題が生じる。この問題についても第4節で議論する。

座標系

キャンバスの座標系は、キャンバスの左上隅を原点とし、右方向をxの正の方向、下方向をyの正の方向と定められている(図3)。

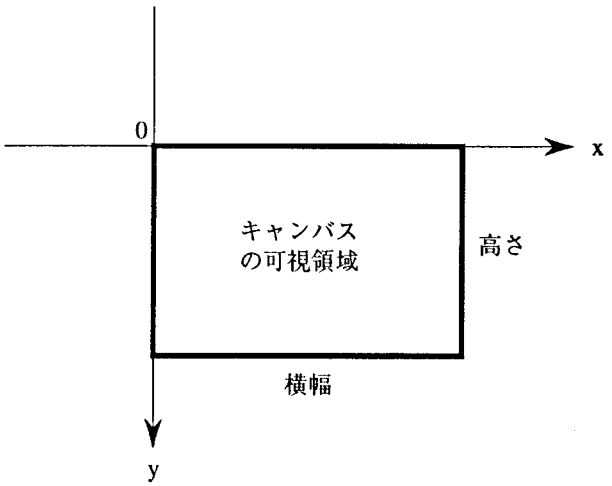

図3. Tkinterにおけるキャンバス平面の座標系と可視領域

キャンバスは無限に広がった平面であり、キャンバスに対する描画メソッドはこの平面上のどこにでも図形を描く事ができる。キャンバス平面には可視領域と呼ばれる領域が存在する。この領域は (0,0) と (w,h) を対角とする矩形である。ここに w はキャンバスウィジェットの横幅、h は高さである。可視領域はキャンバスウィジェットに写影されるので、キャンバス平面に描かれた図形のうち可視領域の部分だけが目に見える。

平行移動

キャンバス c に対する操作

```
c.move(t,x0,y0)
```

はタグ t を構成する全てのアイテムの座標点 (x,y) を (x+x0,y+y0) に移動する。この操作は平行移動を表している。

スケール変換

Tkinter において、動的グラフィックスを実現する上で役に立つもう一つのメソッドが scale であり、操作

```
c.scale(t,x0,y0,a,b)
```

はタグ t を構成する全てのアイテムの座標点 (x,y) を (a*(x-x0)+x0, b*(y-y0)+y0) に移動する。この操作によって図形は x 方向に a 倍に拡大され、y 方向に b 倍に拡大される。点 (x0,y0) は移動しないことに注意する。

注意: これは

```
c.move(t,-x0,-y0)
c.scale(t,0,0,a,b)
c.move(t,x0,y0)
```

と等価である。

注意: 論理座標系(1の長さのピクセル数と座標原点の位置や座標の向きを自由に設定できる座標系)を欲しいと願う読者にとっては scale メソッドが座標系を定義する命令のように思えるかも知れないが、scale はあくまで移動操作であることを心得るべきである。もしも Tkinter において論理座標系が定義されるとすれば、メソッドとしてではなく、Canvas のオプションとしてであろう。

以下に図3a を基に scale メソッドの効果を解説する。

図3aの太線の矩形はキャンバス c である。キャンバスの中の図形に着目 しよう。この図形は一辺が100の正方形と、十字の線、及び円から構成されており、中心の座標は (200,150) である。これらにタグ 'a' が付けられているとする。即ち、

```
x0=200
y0=150
d=50
```

```
c.create_rectangle(x0-d,y0-d,x0+d,y0+d,tags='a')
c.create_line(x0-d,y0,x0+d,y0,tags='a')
c.create_line(x0,y0-d,x0,y0+d,tags='a')
c.create_oval(x0-d/2,y0-d/2,x0+d/2,y0+d/2,tags='a')
```

この下で、

```
c.scale('a',200,150,2,1.5)
```

を実行すると図3bに示すようにタグ 'a' の図形は変形を受ける。図形がx方向に2倍に、y方向に1.5倍に拡大された事、そして点(200,150)は動かない事に注目しよう。

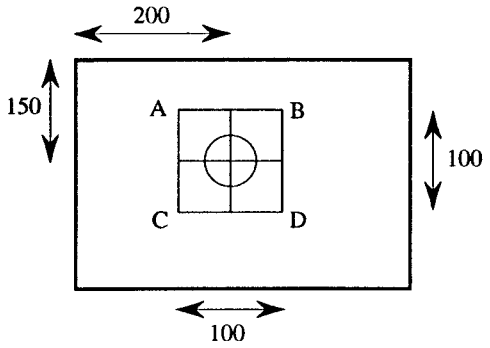

図3a

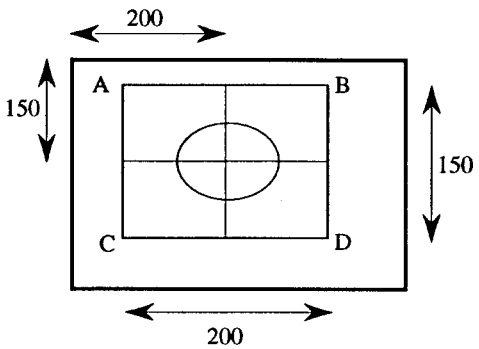

図3b

拡大率が負の場合にもscaleによる操作は意味を持っている。この場合には図形が反転する。

例1. x方向の拡大率が負の場合

```
c.scale('a',200,150,-2,1.5)
```

によって図3bに比べて左右に反転した図3cになる。(文字A,B,C,Dが点の対応関係を表している。)

例2. y方向の拡大率が負の場合

```
c.scale('a',200,150,2,-1.5)
```

によって図3bに比べて上下に反転した図3dになる。(文字A,B,C,Dが点の対応関係を表している。)

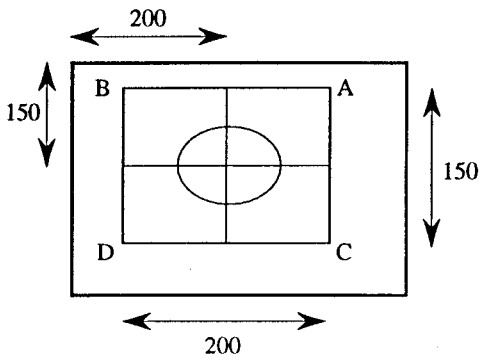

図3c

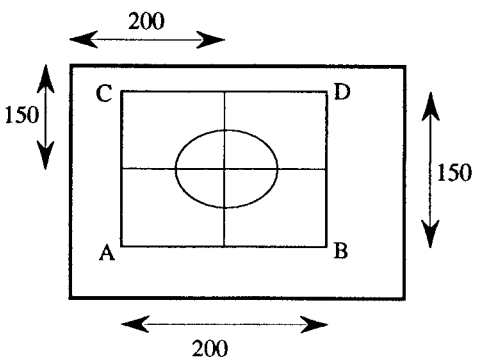

図3d

scaleメソッドによって影響を受けるのはアイテムの配置座標だけであって、文字が反転されない

事はもちろんの事、文字サイズやアイテムの線幅も影響を受けない事に注意しよう。

プログラム例

scale メソッドを活用したプログラムを譜3 に示す。このプログラムの実行画面は図3e に載せてある。但しこの図はウィンドウの全てではない。ボタンを省略しキャンバスだけを取り出している。このプログラムが行っている事は点(150,100)の周りに小さな円を次々と発生させ、それらを scale メソッドを用いて拡大している。実行画面は泡が湧き出ているように見える。

```
from Tkinter import *
from random import *
from math import *
import sys
canvas_width=325
canvas_height=225
canvas_rect=(0,0,canvas_width,canvas_height)
(x0,y0)=(150,100) # scaling center
def pr():
    d=c.postscript(file="a.eps")
def and_rect(r1,r2):
    if r1[0]>r1[2]: (r1[0],r1[2])=(r1[2],r1[0])
    if r1[1]>r1[3]: (r1[1],r1[3])=(r1[3],r1[1])
    if r2[0]>r2[2]: (r2[0],r2[2])=(r2[2],r2[0])
    if r2[1]>r2[3]: (r2[1],r2[3])=(r2[3],r2[1])
    (ax,ay)=(max(r1[0],r2[0]),max(r1[1],r2[1]))
    (bx,by)=(min(r1[2],r2[2]),min(r1[3],r2[3]))
    if (ax>bx) or (ay>by): return None
    return (ax,ay,bx,by)
def animate():
    c.scale('all',x0,y0,1.1,1.1)
    r=1 # radius of circle
    angle=2*pi*random()  # random angle
    x=2*cos(angle)+x0    # (x,y) is the center of circle
    y=2*sin(angle)+y0    # uniformly distributed on circle
    c.create_oval(x-r,y-r,x+r,y+r)
    for t in c.find_all():
        if and_rect(c.coords(t),canvas_rect)!=None: break
        c.delete(t)
    c.after(100,animate)
f=Frame()
f.pack(fill=X)
Button(f,text="write to a.eps",command=pr).pack(side=LEFT,fill=X,expand=YES)
Button(f,text="quit",command=sys.exit).pack()
c = Canvas(width=canvas_width,height=canvas_height)
c.pack()
animate()
mainloop()
```

譜3. scale メソッドを用いたサンプルプログラム

このプログラムの中にただ1つ存在する scale メソッド

```
c.scale('all',x0,y0,1.1,1.1)
```

のお陰で画面上の全ての円が自動的に 0.1 秒間隔で 1.1 倍になる。scale メソッドはこのような問題

では実にありがたい存在である。

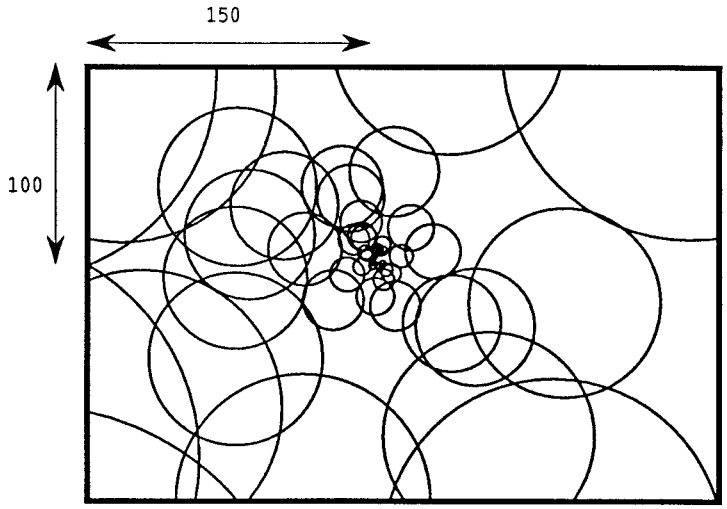

図3e. 譜3の実行画面のスナップショット。泡が湧き出ているように見える。

ゴミ問題とその対策

実はこのプログラムにはプログラミング初心者には気が付きにくい問題が含まれている。

多数の円が発生する。これらが後にスケール変換可能であるためには、それらの座標が Tkinter モジュールの内部で記憶されている必要がある。この事が初心者には気が付きにくいのだ。

画面からはみ出した円を消さずに放置しておいたらどのようになるであろうか?

Tkinter は膨大な数の円をスケール変換し、そしてそれらを描き直す。それらの殆どは意味の無い操作なのである。そしてやがて負荷が極限にまで達し、コンピュータが動かなくなるであろう。

このプログラムではこの問題に対して次のように対処している。

生成される円の中にスケール変換の中心点 (150,100) が含まれないようにする。そうすれば生成された円はやがてキャンバスの外に移動するのが保障される。そのもとでキャンバスの外に移動した円を削除する。

プログラムの中の

```
r=1 # radius of circle
angle=2*pi*random()  # random angle
x=2*cos(angle)+x0    # (x,y) is the center of circle
y=2*sin(angle)+y0    # uniformly distributed on circle
c.create_oval(x-r,y-r,x+r,y+r)
```

によって点 (150,100) を中心とする半径 2 の円周上(下図の破線で示されている)に半径 1 の円(下図の実践で示されている)を描く。これによって生成された円の内部には点 (150,100) は含まれない。

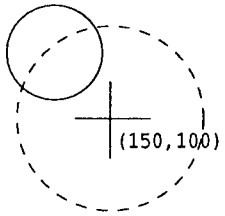

図3f.

キャンバスの外に出てしまった円を消しているのが

```
for t in c.find_all():
    if and_rect(c.coords(t),canvas_rect)!=None: break
    c.delete(t)
```

である。c.find_all() はキャンバス上の全てのアイテムを、それらが生成された順序でタプル形式で返す。この問題では時間が経過したアイテムほどキャンバスの視界から消えている可能性が圧倒的に高い。従ってここではタプルの先頭から調べ、キャンバスの視界に入っている円が見つかった段階で消去操作を終えている。

キャンバスの視界から消えている全ての円を消していない事に留意すべきである。この段階で消去されなくてもやがては消去されるのだ。見えないゴミが多少含まれていようが気にする必要は無い。大切な事はゴミの量を一定水準に抑える事である。

視界からはみ出したアイテムを見つけるのに、このプログラムでは関数 and_rect(r1,r2) を定義している。このような基本的な関数は初めから Tkinter に備わっていても良さそうなのだが、残念ながら備わっていない。

関数 and_rect(r1,r2) は2つの矩形領域 r1 と r2 が交わってできる矩形の座標を返す(下図)。そしてもしも交わり部分が無ければ None を返す。

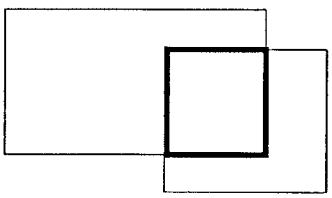

図3g. 2つの矩形が交わってできる矩形
太線で示されている。

第4節　Windowsにおけるフォントサイズ問題の解決法

画面の1インチ当たりのピクセル数はWindowsでは96、UNIXでは72であると想定されている。従ってピクセルと他の単位を混在した場合には画面に表示される図形の相互の大きさの関係はOSによって異なる事になる。画面上に表示される文字の大きさがWindowsではUNIXよりも大きく表示される。この原因もここから派生している。なぜならは1ポイントは1/72インチであると定められているからである。

Pythonが描く図形はGhostScriptやGhostViewなどのポストスクリプトプロセッサによって処理される。ポストスクリプトでは伝統的に画面の1インチ当たりのピクセル数を72と見做したモデルを使用してきている(文献[16])。そのためにWindows環境ではPythonは画面上のイメージを正しく印刷しない。

Windows環境のPythonが何故Windows の設定である1インチ当たりのピクセル数96をそのまま引き継いでいるかは不明であるが、ここでは72に設定し直す方法を紹介する。こう設定する事によってWindows環境とUNIX環境では共通の画面イメージが得られるばかりか、Windows環境でも印刷イメージが画面イメージに一致する。

PythonにはTcl/Tkと会話するためのPython/Tkのメソッド`tk.call`が備わっている。そしてTcl/Tkには現在設定されている 1インチ当たりのピクセル数を知る命令とそれを変更する命令が備わっている(文献[18]の "Tk Built-in Commands" を見よ)。これらをPythonで利用するには

```
from Tkinter import *
tk=Tk()
print tk.tk.call('tk','scaling','-displayof',tk._w)
tk.tk.call('tk','scaling','-displayof',tk._w,1.0)
```

のようにする。

即ち、`tk`を`Tk`のインスタンスとするとき、

```
tk.tk.call('tk','scaling','-displayof',tk._w)
```

によって、現在の設定値が得られ、(Windowsの場合、この値は96.0/72.0 = 1.3333になっているであろう)

```
tk.tk.call('tk','scaling','-displayof',tk._w,1.0)
```

で、この値を1.0に設定する。

読者はここに述べた事を実際に譜1b「楕円を動かすプログラム」に適用してみるがよい。その場合には譜1bに代わって次ぎの譜4aを実行する事になる。但し、ここでは

```
print tk.tk.call('tk','scaling','-displayof',tk._w)
```

は本質的ではないので省いてある。Windows環境のPythonでは文字の大きさが小さくなり、ウィンドウ内でのその大きさが図1で示されている文字の大きさになったのが観察されるであろう。

```
from Tkinter import *
tk=Tk(); tk.tk.call('tk','scaling','-displayof',tk._w,1.0)
def animate(n):
    if n <= 0: return
    print 'moved'
    c.move(p,10,0)
    c.after(1000,animate,n-1)
c = Canvas(width=500,height=300)
c.pack()
c.create_text(250,150,text="Animation Demo",font="times 60 italic",fill="red")
p=c.create_oval(50,50,200,250)
animate(30)
mainloop()
```

譜4a.「楕円を動かすプログラム」

従って、Tkinter を使用する場合には

```
from Tkinter import *
```

の直後に行

```
tk=Tk(); tk.tk.call('tk','scaling','-displayof',tk._w,1.0)
```

を挿入すればよい。しかしながら Tkinter モジュールの Tk クラスの定義を読めば判明するように、プログラムの中に直接この行を挿入しなくても、Tkinter が呼び出される場合に自動的にこれが実行される機能が Python に存在する。そのためには譜4b の内容を書いたファイル

```
.Tk.py
```

を(注1)、実行しようとしている Python のプログラムと同一のディレクトリに準備する(注2)。

注1: Windows では「マイコンピュータ」のメニューからファイルの名前をピリオドで始まる名前に変更しょうとしても拒否されるが、「コマンドプロンプト」からは rename コマンドで変更できる。また「メモ帳」からもピリオドで始まるファイルに保存できる。

注2: あるいは環境変数 HOME が存在すれば、HOME で指定されるディレクトリに置いてもよい。".Tk.py" が両方に存在すれば HOME に置かれた ".Tk.py" だけが読み取られる。Windows 環境では環境変数 HOME が自動的には作成されないので ".Tk.py" は Tkinter を使用するプログラムと同一のディレクトリに置くのがよいであろう。

```
self.tk.call('tk','scaling','-displayof',self._w,1.0)
```

譜4b. ファイル ".Tk.py" の内容

第5節　シミュレーションへの応用

動的グラフィックスの応用として確率的事象のシミュレーションを採りあげる。ここで扱う問題はその中で最も基本的な問題、即ち、ガウスによる中心極限定理である。

中心極限定理

有限区間の分布関数を $f(x)$ とする。このもとで、$X_1, X_2, .., X_n$ を 分布 $f(x)$ に従う確率変数とする。その時、

$$X = \frac{(X_1 + X_2 + .. + X_n)}{\sqrt{n}} \tag{1}$$

の分布は n が大きいと(即ち、$n \to \infty$ の極限で)

$$g(x) = \frac{1}{\sqrt{2\pi}\sigma} \exp\left\{-\frac{1}{2}\left(\frac{x-\mu}{\sigma}\right)^2\right\} \tag{2}$$

の形をとるのだと。ここに μ と σは $f(x)$ によって次ぎの様に定まる定数である。

$$\mu = \int x f(x) dx \tag{3}$$

$$\sigma^2 = \int (x-\mu)^2 f(x) dx \tag{4}$$

この定理は驚くべき事を主張している。多数の確率変数の和をとった場合、もとの分布関数 $f(x)$ の持っていた個性がたった2つ(平均と分散)を除いて消失してしまい、和はある普遍的な分布 $g(x)$ に従うのだと。この分布を正規分布(あるいはガウス分布)と言う。

中心極限定理は統計学の分野の最も基本的な定理なので大抵の統計学の教科書には解説されているはずである。証明に興味のある読者は例えば文献[19]を参照されたい。

目的

この節では $f(x)$ として区間 [-1,1] の一様分布を考える。即ち、

$$f(x) = \begin{cases} 1/2 & (|x| \le 1) \\ 0 & (|x| > 1) \end{cases} \tag{5}$$

この場合、容易に μと σが計算できて

$$\mu = 0 \tag{6}$$

$$\sigma = \frac{1}{\sqrt{3}} \tag{7}$$

となり $g(x)$ が定まる。

他方、この $f(x)$ の下で乱数 $X_1, X_2, .., X_n$ を発生させ、これらの乱数から X の値を式(1)から計算する。この計算を何回も何回も繰り返し、そうして得られた X の値の分布をヒストグラムで表す。

このヒストグラムを $g(x)$と比較して中心極限定理の主張を実験的に検証するのに役立つプログラムを作成するのがここでのテーマである。

以下のプログラムでは X の計算式を(1)ではなく、次ぎの(8)によって行う。式(8)の m の値は利用者が自由に設定できるパラメータとする。$m=\sqrt{n}$ に設定すればこの式は(1)に還元する。

$$X=\frac{(X_1+X_2+..+X_n)}{m} \tag{8}$$

このような自由度を認めるのは m=1 の場合や m=n の場合のシミュレーションをも可能にするためである。式(8)の下では X の分布は大きな n に対して同様に式(2)の形をとるが、 σは(4)に代わって

$$\sigma^2=\frac{n}{m^2}\int(x-\mu)^2 f(x)dx \tag{9}$$

で与えられる。式(5)の $f(x)$ に対してはこの計算結果は

$$\sigma=\frac{1}{m}\sqrt{\frac{n}{3}} \tag{10}$$

となる。

プログラムの仕様

以下にプログラムの基本使用と利用者インターフェースの仕様を挙げる。

基本仕様

1. パラメータとしては以下の4つを設定できる様にする事。(括弧内は初期値)

和をとる一様乱数の個数 n	(16)
割り算の分母 m	(4)
ヒストグラムの横軸のメッシュ	(0.1)
ヒストグラムの縦軸のスケール	(100)

2. X の値が計算されるごとにヒストグラムが自動的に更新される事。
3. 途中で止めて、理論曲線と比較できる様にする事。また、再開できる事。
4. 結果をプリンタに出力できる様にする事。

利用者インターフェースの仕様

1. プログラムの実行と共にシミュレーションが自動的に実行される事。この時のパラメータは基本仕様の1に従う初期値とする。この様子を図5aに示す。図の横軸は X の値、縦軸 f は発生頻度である。初期設定では、横軸は 0.1 刻み、発生頻度は 100 が単位になっている。
2. pause ボタンを押したら実行を停止し、それとともに自動的にヒストグラムと理論曲線が比較される事。またこの時、pause ボタンの表示が continue に変化する事(図5b)。
3. continue ボタンを押したら実行を再開し、continue ボタンの表示が pause に変化する事。
4. preference ボタンを押したら preference ウィンドウが開き、パラメータ を再設定する入力欄が現れる事(図5c)。
5. preference のパラメータを変更した時にヒストグラムは初期化される事。

6. write to a.eps ボタンを押した時に描画結果を EPS 形式のファイル a.eps に出力する事。
7. quit ボタンでプログラムが終了する事。

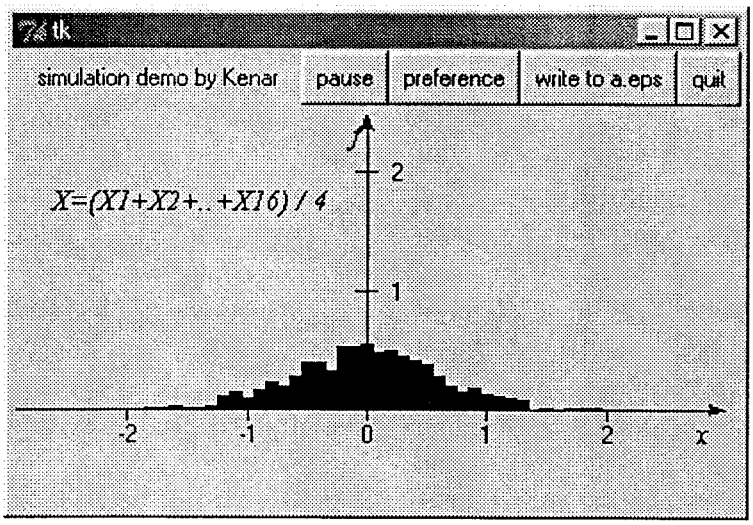

図5a. プログラムの実行画面(pause を押す前)

区間 [-1,+1] の一様乱数を n 個生成し、それらの和を m で割った値を x としたときの分布を表すヒストグラム。横軸 x を等間隔に区切って発生する回数を縦軸にとる。なお、左上の標題の *Kenar* は筆者がフリーのプログラムに添えているペンネームである。

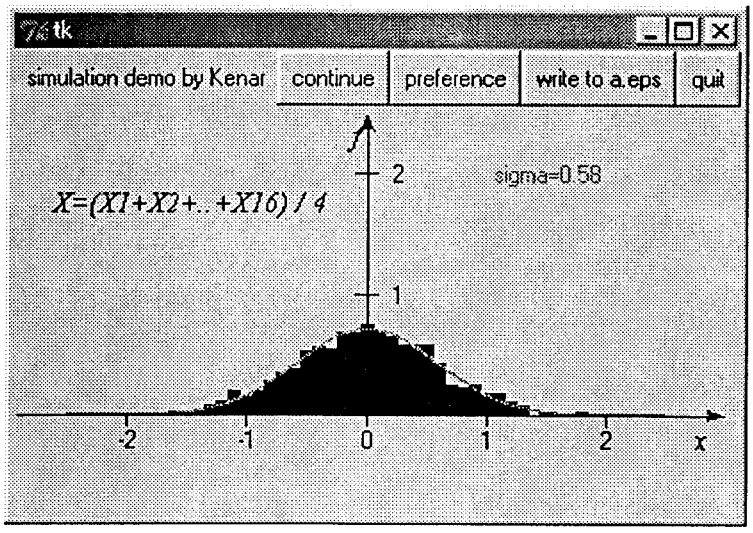

図5b. プログラムの実行画面(pause を押した時)

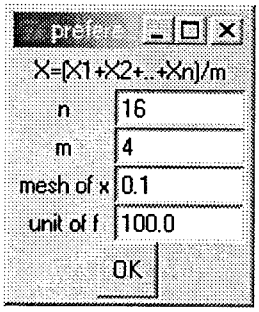

図5c. preference ウィンドウ

pauseボタンの表示が continue に変化し、理論分布が表示される。
曲線の右肩には "sigma=" によってその時の σ の値も示される。

プログラム作成における注意点

以下にプログラム作成における注意点を挙げる。これらは当然の要求であるが、注意を怠るとこれらの要求は満たされない可能性が高い。また、これらの要求の実現には技術力が要求される。

1. 画面が更新される際にゴミを作らない事。
2. preference ウィンドウを二重に出さない事。
3. preference ウィンドウを消去した後にも、再度 preference ウィンドウを生成できる事。

プログラム

以下に中心極限定理をシミュレーションするプログラムを載せる。プログラム名は gauss.py となっていが、読者は任意の名前を付けることができる。

なお、このプログラムは筆者の FTP サーバ

ftp://ar.aichi-u.ac.jp/Python/animation/gauss.py

にも置かれている。

```
from Tkinter import *
from random import *
import sys
from math import *

# set the display scaling to be 1.0 (72 dpi display)
tk=Tk(); tk.tk.call('tk','scaling','-displayof',tk._w,1.0)

# default data
sum_num=16
dev_num=4
mesh=0.1
y_unit=100.0

font="Times 14 italic"

cid=None   # callback id to cancel the timer
w=None # preference window

def pr():
    # print canvas to a file named "a.eps"
    c.postscript(file="a.eps")
def title(n,m):
    # draw a title.
    # n: the number of samples
    # m: avarage by m
    # example: (X1+X2+..+Xn)/m
    t=("X1 / ","(X1+X2) / ","(X1+X2+X3) / ","(X1+X2+X3+X4) / ",
    "(X1+X2+..+X%d) / ")
    if n < 5:
        s="X="+t[n-1]+str(m)
    else:
        s="X="+t[4]%n+str(m)
    a=c.create_text(20,40,text=s,anchor='nw',tags='title',font=font)
def drawcurve():
    # draw a Gauss curve using sum_num, dev_num
```

```
    sig2=(1.0/3)*sum_num/(dev_num*dev_num)
    sig=sqrt(sig2)
    norm=1.0/(sqrt(2*pi)*sig)
    # count*mesh/y_unit is the square measure of histogram
    # our curve must be normarized using this square measure
    norm=norm*count*mesh/y_unit
    coord=[]
    for n in range(-25,26):
        x=0.1*n
        z=x/sig
        y=norm*exp(-z*z/2)
        coord=coord+[(x,y)]
    c.delete('curve')
    c.create_line(coord,fill='red',smooth=YES,tags='curve')
    c.create_text(1.5,2,text="sigma=%4.2f"%sig,fill='red',tags='curve')
    c.scale('curve',-3,2.5,60,-60)
def pause():
    # pause/continue the simulation
    global cid
    if cid==None:
        animate()
        text='pause'
    else:
        c.after_cancel(cid)
        cid=None
        text='continue'
        drawcurve()
    pause_button.configure(text=text) # change the text on the pause button
def labeledEntry(w,title,init=None):
    # create single labeled entry
    # w: widget to attach this
    # title: the name of label
    # init: initial value of the entry
    f=Frame(w)
    f.pack(expand=YES,fill=X)
    Label(f,text=title).pack(side=LEFT,expand=YES)
    e=Entry(f,width=10)
    e.pack(side=RIGHT)
    e.delete(0,END)
    e.insert(0,init)
    return e
def pref():
    # reflect our preference to simulation parameters
    global sum_num,dev_num,mesh,y_unit,dic,count
    sum_num=int(e1.get())
    dev_num=float(e2.get())
    mesh=float(e3.get())
    y_unit=float(e4.get())
    dic={}
    count=0
    c.delete('rects')
    c.delete('title')
    title(sum_num,dev_num)
def preference(event=None):
    # configure preference window
    # if this window is destroyed, event will not be None
    global e1,e2,e3,e4,w
    if event: w=None; return
    if w: return
    w=Toplevel()
    w.title("preference")
```

```
    w.bind("<Destroy>",preference)
    Label(w,text="X=(X1+X2+..+Xn)/m").pack()
    e1=labeledEntry(w,"n",sum_num)
    e2=labeledEntry(w,"m",dev_num)
    e3=labeledEntry(w,"mesh of x",mesh)
    e4=labeledEntry(w,"unit of f",y_unit)
    Button(w,text="OK",command=pref).pack()

dic={}
count=0
def animate():
    global cid,dic,count
    x=0
    for i in range(0,sum_num):
        x=x + 2*random() - 1
    x=x/dev_num
    x=round(x/mesh)*mesh
    if dic.has_key(x):
        f=dic[x][0]+1
    else:
        f=1
    y=f/y_unit
    if f==1:
        r=c.create_rectangle(x-mesh/2,y,x+mesh/2,0,fill='black',tags='rects')
        dic[x]=(1,r)
    else:
        r=dic[x][1]
        c.coords(r,x-mesh/2,y,x+mesh/2,0)
        dic[x]=(f,r)
    c.scale(r,-3,2.5,60,-60)
    count=count+1
    cid=c.after(10,animate)

# configure main window
f=Frame()
f.pack(fill=X)
Label(f,text="simulation demo by Kenar").pack(side=LEFT,fill=X,expand=YES)
pause_button=Button(f,text='pause',command=pause)
pause_button.pack(side=LEFT)
Button(f,text='preference',command=preference).pack(side=LEFT)
Button(f,text='write to a.eps',command=pr).pack(side=LEFT)
Button(f,text='quit',command=sys.exit).pack(side=LEFT)

# create canvas and draw basic items on it
c = Canvas(width=360,height=200)
c.pack()
c.create_line(-3,0,+3,0,arrow=LAST)
c.create_line(0,0,0,2.5,arrow=LAST)
for n in -2,-1,0,1,2:
    c.create_line(n,0,n,-0.1)
    c.create_text(n,-0.2,text=n)
for n in 1,2:
    c.create_line(-0.1,n,0.1,n)
    c.create_text(0.2,n,text=n,anchor='w')
c.create_text(-0.1,2.3,text="f",font=font)
c.create_text(2.8,-0.2,text="x",font=font)
c.scale('all',-3,2.5,60,-60)
title(sum_num,dev_num)
animate()
mainloop()
```

プログラムの解説

「プログラム作成における注意点」で述べた事がどのようにプログラムに反映されているかを解説する。

ゴミの問題は、ヒストグラムにおける出現頻度を図示する矩形のアイテム ID と出現回数を辞書に登録する事によって解決されている。辞書のキーには階級値(矩形の中心の x 座標)が採られている。出現頻度の変更があった時には、既に辞書に登録されている矩形に関しては新たに矩形を生成しないで、単にその矩形の形を変更する。これらの事は関数 animate の中の

```
x=round(x/mesh)*mesh
if dic.has_key(x):
    f=dic[x][0]+1
else:
    f=1
y=f/y_unit
if f==1:
    r=c.create_rectangle(x-mesh/2,y,x+mesh/2,0,fill='black',tags='rects')
    dic[x]=(1,r)
else:
    r=dic[x][1]
    c.coords(r,x-mesh/2,y,x+mesh/2,0)
    dic[x]=(f,r)
```

によって行われている。

preference ウィンドウの問題は関数 preference の引数の event によって処理されている。preference ウィンドウをマウスで消去すると event に None でない値が渡される。preference ウィンドウはプログラムの中では w で表わされているので、消去された場合には w を None に設定する。逆に w が None でない場合には新たに preference ウィンドウを生成しない様にする。即ち

```
def preference(event=None):
    global e1,e2,e3,e4,w
    if event: w=None; return
    if w: return
    w=Toplevel()
    ....
```

でコントロールが行われている。

付録A　アニメーション作りに役立つCanvas のメソッド一覧

ここでは Canvas のメソッドを個別に解説する。ただし全てのメソッドが網羅的に解説されていはいない。ここで採りあげるのは動的グラフィックスの作成に役に立ちそうなメソッドのみである。Canvas のメソッドは、その引数や返す値に関して、以下の統一的なルールを持っているので、個別の解説の中ではいちいち断らない。

1. メソッドの引数のアイテム識別子

　　タグも許されるが、その場合にタグ中の最下位ののアイテムで置き換えられる。

2. メソッドの引数のタグ

　　アイテム識別子も許される

3. メソッドの引数の実数

　　整数も許される

4. メソッドが返す座標系

　　Canvas 座標系でピクセルを単位とする

ここにルール 1 の意味は、個別解説の中では引数はアイテム識別子であると説明されていても、実際にはタグも許され、タグを使用した場合にはタグ中の最下位のアイテムで置き換えて解釈される事を言う。ルール 2, 3 も個別解説の引数がどのように拡張されて解釈されるかを述べている。ルール 4はメソッドが座標系を返す場合の座標の単位を述べている。

Canvas のメソッドのうち、ここでは採り挙げなかったメソッドは以下の通りである。

```
forcus, icursor, index, insert,  select_clear, select_from, select_item, select_to
postscript, scan_dragto, scan_mark, xview_moveto, xview_scroll, yview_moveto,
yview_scroll
```

これらの内、xview と yview はスクロールバーを備えたキャンバスが必要になるときに使用される。使い方は Listbox や Text ウィジェットのそれと本質的な違いはない。詳しくは文献[14]の第10節 Listbox を参照するがよい。

☐ addtag_above(s,t)

引数 s,t: タグ

戻り値: None

機能: タグ t の中の最上位のアイテムより1つ上のアイテムにタグ s を付ける。

☐ addtag_below(s,t)

引数 s,t: タグ

戻り値: None

機能: タグ t の中の最下位のアイテムより1つ下のアイテムにタグ s を付ける。

☐ addtag_withtag(s,t)

引数 s,t: タグ

戻り値: None

機能: タグ t の中の全てのアイテムに新たにタグ s を付ける。

☐ after(t,f,a1,a2,...)

引数 t: 非負整数 (時間を 1/1000 秒単位で与える)

引数 f: 関数

引数 a1,a2,...: f に与える引数 (省略可能)

戻り値: after_cancel メソッドに渡す識別子

機能: t/1000 秒後に実行する関数を定める。

☐ after_cancel(i)

引数 i: after メソッドが返す識別子

引数 f: 関数

戻り値: None

機能: after メソッドによるタイマー設定を取り消す。

☐ bbox(t)

引数 t: タグ

戻り値: 実数のリストまたは None

機能: bbox とは bounding box の意味である。i をアイテムとするとき bbox(i) はマウスカーソルがアイテム i を指す領域を意味しており、その領域はアイテム i を囲む矩形である。タグ t が複数のアイテムを含む場合、bbox(t) はタグ t に含まれる全てのアイテムの bbox を含む最小の矩形を返す。下図に例を示す。t には2つのアイテム(楕円と文字列)が含まれている。各々の bbox はそれらを囲む薄い矩形で示されている。タグ t の bbox はそれらを囲む最小の矩形である。

タグ t が存在しない時には None を返す。

☐ coords(i)

引数 i: アイテム識別子

戻り値: 実数のリストまたは空のリスト

機能: アイテム i の座標を返す。アイテム i が存在しない時には空のリストを返す。

☐ coords(i,x0,y0,x1,y1,...)

引数 i: アイテム識別子

引数 x0,y0,x1,y1,...: 実数(座標系)

戻り値: [](空のリスト)

機能: アイテム i を再配置する。

☐ create_line(x1,y1,x2,y2,...,o=v,...)

☐ create_rectangle(x1,y1,x2,y2,o=v,...)

☐ create_polygon(x1,y1,x2,y2,...,o=v,...)

☐ create_oval(x1,y1,x2,y2,o=v,...)

☐ create_arc(x1,y1,x2,y2,o=v,...)

☐ create_text(x,y,o=v,...)

☐ create_bitmap(x,y,o=v,...)

☐ create_image(x,y,o=v,...)

☐ create_window(x,y,o=v,...)

引数 x,y,x1,y1,x2,y2: 座標

引数 o=v: オプション

戻り値: アイテム識別子

機能: これらについては第2節を見よ。

☐ delete(t)

引数 t: タグ

戻り値: None

機能: タグ t の全てのアイテムを削除する。

☐ dtag(i)

引数 i: アイテム識別子

戻り値: None

機能: アイテム i から全てのタグを削除する。

☐ dtag(i,t1,t2,...)

引数 i: アイテム識別子

引数 t1,t2,...: タグ

戻り値: None

機能: アイテム i からタグ t1,t2,... を削除する。

☐ find_above(t)
引数 t: タグ
戻り値: アイテム識別子
機能: ディスプレイリストの中の t の 1 つ上のアイテム識別子を返す。

☐ find_all()
戻り値: 自然数から構成されるタプル
機能: 全てのアイテムの一覧を返す。一覧の順序はディスプレイリストの順である。
find_withtag('all') と同じ

☐ find_below(t)
引数 t: タグ
戻り値: アイテム識別子
機能: ディスプレイリストの中の t の 1 つ下のアイテム識別子を返す。

☐ find_enclosed(x1,y1,x2,y2)
引数 x1,y1,x2,y2: 実数
戻り値: 実数から構成されるタプル
機能: 矩形領域(x1,y1,x2,y2) で囲まれているアイテムの一覧を返す。

☐ find_overlapping(x1,y1,x2,y2)
引数 x1,y1,x2,y2: 実数
戻り値: 実数から構成されるタプル
機能: 矩形領域 (x1,y1,x2,y2) と重なるアイテムの一覧を返す。

☐ find_withtag(t)
引数 t: タグ
戻り値: 自然数から構成されるタプル
機能: タグ t が付けられているアイテムの一覧を返す。

☐ gettags(i)
引数 i: アイテム識別子
戻り値: 自然数から構成されるタプル
機能: アイテム i に付けられているタグの一覧を返す。(定義済タグ 'all' はこの一覧には含まれない。)

☐ itemcget(i,o)

引数 i: アイテム識別子

引数 o: オプション (文字列)

戻り値: 文字列 または 数字 または None

機能: アイテム i の属性 o の現在の値を返す。

☐ itemconfigure(i)

引数 i: アイテム識別子

戻り値: 集合の値

機能: アイテム i の属性の一覧を返す。

☐ itemconfigure(t,o=v,...)

引数 t: タグ

引数 o=v: オプション

戻り値: None

機能: タグ t に属するアイテムの属性 o を v に定める。

☐ move(t,x,y)

引数 t: タグ

引数 x,y: 実数

戻り値: None

機能: タグ t で指定される図形を (x,y) だけ移動する。

☐ scale(t,x0,y0,a,b)

引数 t: タグ

引数 x0,y0: 実数

引数 a,b: 実数

戻り値: None

機能: タグ t を構成する図形の座標点 (x,y) を点 (a*(x-x0)+x0, b*(y-y0)+y0) に移動する。その結果タグ t を構成する図形は左右に a 倍、上下に b 倍に拡大される。

☐ type(i)

引数 i: アイテム識別子

戻り値: 文字列

機能: アイテムの種類(rectangle や line など)を返す。

☐ tag_bind(t)

引数 t: タグ

戻り値: 文字列(イベントシーケンス)のタプル

機能: タグ t にバインドされているイベントシーケンスの一覧を返す。

☐ tag_bind(t,s,f)

引数 t: タグ

引数 s: 文字列(イベントシーケンス)

引数 f: 関数

戻り値: tag_unbind に渡す値

機能: タグ t にイベント s をバインドする。f にイベントが発生した時に実行される関数を指定する。戻り値はここで指定したバインドを取り消すのに使用される。
オプションとして第4引数に '+' を与える事ができるが、解説は省略する。

☐ tag_lower(t,s)

引数 t,s: タグ

戻り値: None

機能: タグ t をタグ s の下位に持っていく。以下に tag_lower の効果を2つばかり例で示す。数字はアイテム識別子を表し、その下の a と b はタグを表している。見やすくするために1つの行に1つのタグを書いている。矢印 ⇒ の左には tag_lower(a,b) を実行する前のディスプレイリストを、右側に実行後のディスプレイリストを示す。

```
          1  2  3  4  5  6  7                1  3  6  2  4  5  7
例1       a     a        a        ⇒         a  a  a
             b        b                                b     b

          1  2  3  4  5  6  7                1  3  6  2  4  5  7
例2       a     a        a        ⇒         a  a  a
             b  b     b                        b     b     b
```

タグ a と b に共通のアイテム識別子が存在する場合(例2のアイテム3)にはこの結果はいささかルーズである。正しくは

```
1  6  3  2  4  5  7
a  a  a
      b  b     b
```

になるべきであると考えたくなるが、そのようにはなっていない。アイテム 3 のタグ b の存在が単に無視されるのである。

☐ tag_raise(t,s)

引数 t,s: タグ

戻り値: None

機能: タグ t をタグ s の上位に持っていく。

☐ tag_unbind(t,s)

引数 t: タグ

引数 s: 文字列(イベントシーケンス)

戻り値: None

機能: タグ t からイベント s のバインドを取り消す。

☐ tag_unbind(t,s,f)

引数 t: タグ

引数 s: 文字列 (イベントシーケンス)

引数 f: tag_bind によって返された値

戻り値: None

機能: tag_bind の効果を取り消す。

参考文献

[1] Mark Lutz著/飯坂剛一監訳、村山敏夫、戸田英子共訳:「Python 入門」(O'Reilly Japan, 1998)

[2] Mark Lutz著/飯坂剛一監訳、村山敏夫、戸田英子共訳:「Python プログラミング」
(O'Reilly Japan, 1998)

[3] Mark Lutz, David Ascher著/紀太章訳:「初めてのPython」(O'Reilly Japan, 2000)

[4] David M. Beazley著/習志野弥治朗訳:
「Python テクニカルリファレンス — 言語仕様とライブラリ —」
(ピアソン・エデュケーション, 2000)

[5] Mark Lutz:"Programming Python",(O'Reilly & Associates,Inc. 1996)

[6] John E.Grayson : "Python and Tkinter Programming", (Manning, 2000)

[7] John K.Ousterhout著/ 西中芳幸、石曾根信共訳:
「Tcl&Tk ツールキット」(ソフトバンク、1995)

[8] 宮田重明、芳賀敏彦:「Tcl/Tk プログラミング入門」(オーム社,1995)

[9] V.Quercia,T.O'Reily/ 大木敦雄監訳:「X ウィンドウ・システム・ユーザ・ガイド」
(ソフトバンク、1993)

[10] 有澤健治:「Python によるプログラミング入門 (第2版)」(講義テキスト, 2000)

[11] 有澤健治:「Python によるグラフィックス」
(愛知大学情報処理センター紀要「Com」第17号, 1999)

[12] 有澤健治:「Python のすすめ」(愛知大学情報処理センター紀要「Com」第17号, 1999)

[13] 有澤健治:「Python におけるGUIの構築法 I — ウィジェットの配置 —」
(愛知大学情報処理センター紀要「Com」第18号, 2000)

[14] 有澤健治:「Python におけるGUIの構築法 II — ウィジェット各論 —」
(愛知大学情報処理センター紀要「Com」第19号, 2001)

[15] 有澤健治:「Python におけるGUIの構築法 III — Text ウィジェット —」
(愛知大学情報処理センター紀要「Com」第20号, 2001)

[16] Adobe Systems:「POSTSCRIPT リファレンス・マニュアル (第2版)」(アスキー出版局, 1991)

[17] "Python 2.0 Documentation" (Python2.0 配布ファイル `Python20/Doc/index.html`)

[18] "Tcl/Tk Reference Manual" (Tcl/Tk8.0 配布ファイル `Tcl/doc/tcl80.hlp, v8.0.4`)

[19] 竹内啓:「数理統計学 — データ解析の方法 —」(東洋経済、1963)

[20] 海上忍:「Linux 各種設定×活用 徹底ガイド」(技術評論社、2000)