numerical integration; Game of Life

Ben Bolker

17 November 2019

numerical integration

In first year calculus the definite integral of a function $f(x)$ over the interval $[a, b]$ is defined to be the limit of a sequence of *Riemann sums*:

$$\int_a^b f(x)\,dx = \lim_{n\to\infty} \sum_{i=0}^{n-1} f(x_i)\,\Delta x$$

where $\Delta x = (b-a)/n$ and $x_i = a + i \cdot \Delta x$ - So the definite integral can be approximated using Riemann sums for suitably large values of n

coding Riemann sums in numpy

```
import numpy as np
def num_int(f, a, b, n):
   dx = (b-a)/n
   x = np.arange(a, b, step = dx)
   y = f(x)
   return y.sum() * dx
```

Note this takes a *function* `f` as input.

example

Try approximating $\int_0^1 x^2\,dx$ with $n = \{10, 50, 5000\}$

```
def x_squared(x):
   return x ** 2
approx1 = num_int(x_squared, 0, 1, 10)
approx2 = num_int(x_squared, 0, 1, 50)
approx3 = num_int(x_squared, 0, 1, 5000)
print(approx1, approx2, approx3)
```

```
## 0.2850000000000001 0.3234 0.33323334
```

We could create an *adaptive* version of this that keeps trying larger values of `n` until the results are "close enough"

refinements

- `num_int` uses the *left endpoint* rule to compute Riemann sums for a function $f(x)$

- In Section 7.7 of Dr. Stewart's Calculus textbook, several other methods are presented.
- The *right endpoint* and *midpoint* rules are simple variations of the left endpoint rule:

$$\int_a^b f(x)\,dx = \lim_{n\to\infty} \sum_{i=0}^{n-1} f(x_{i+1})\,\Delta x \qquad \text{right endpoint}$$

$$\int_a^b f(x)\,dx = \lim_{n\to\infty} \sum_{i=0}^{n-1} f\left((x_i + x_{i+1})/2\right)\,\Delta x \qquad \text{midpoint}$$

other rules

- **trapezoid rule**: use the average of the approximations obtained by using the left and right endpoint rules
- **Simpson's method**: use quadratic functions (parabolas) instead of linear functions to interpolate

$$\int_a^b f(x)\,dx \approx \sum (\Delta x/6)\left(f(x_i) + 4f((x_i + x_{i+1})/2) + f(x_{i+1})\right)$$

We'll write a function that implements any of these rules, and check it with $\int_0^1 x^2\,dx$ and $\int_0^1 x^3\,dx$.

area of a circle

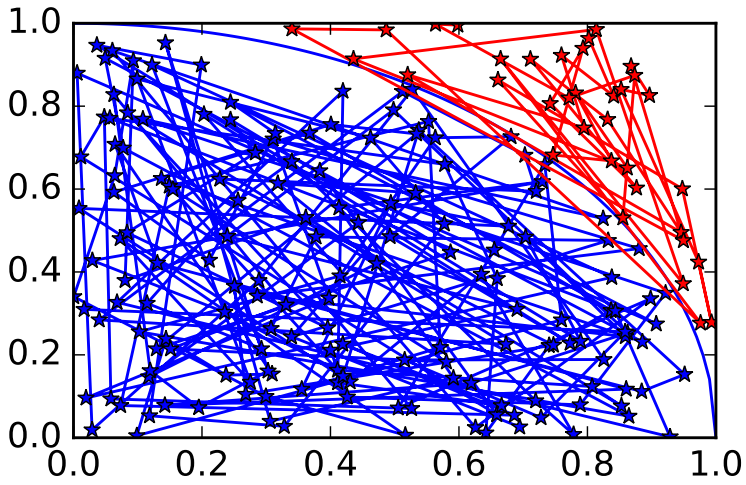
- what if we want to figure out the area of a circle with radius 1?
- (we already know it's 1)
- because $x^2 + y^2 = 1$, one quadrant is traced out by the function $y = \sqrt{1 - x^2}$ from $x = 0$ to 1
- (we could do this symbolically: $= 1/2\left(x\sqrt{1-x^2} + \sin^{-1}(x)\right)_0^1$)
- let's try it with our integrator from above

Monte Carlo integration

- "Monte Carlo" in general refers to **any** algorithm that uses (pseudo) random numbers
- a Monte Carlo method to approximate π:
- draw a square of area 1
- inscribe a quarter circle (with radius = 1)
- the area A of the quarter-circle is equal to $\pi/4$
- randomly throw darts in the square and count the number of them that fall within the circle (i.e. $x^2 + y^2 < 1$)
- the ratio of the number that fall into the circle to the total number thrown should be close to the ratio of the area of the circle to the area of the square

- the more darts thrown, the better the approximation.

```
import matplotlib.pyplot as plt
import numpy.random as npr
x = np.linspace(0,1,101)
y = np.sqrt(1-x**2)
fig, ax = plt.subplots()
ax.plot(x,y)
xr = npr.uniform(size=200)
yr = npr.uniform(size=200)
incirc = xr**2+yr**2<1
plt.plot(xr[incirc],yr[incirc],"b-*")
plt.plot(xr[np.logical_not(incirc)],yr[np.logical_not(incirc)],"r-*")
```




```
import matplotlib.pyplot as plt
import numpy as np
x = np.linspace(0,1,101)
y = np.sqrt(1-x**2)
plt.figure(figsize=(4,4))
plt.plot(x,y)
plt.savefig("pix/qtrcirc.png")
```

another example

Overlap of circles at (1.5,2.5) (radius 1), (4,3) (radius 3), (1,2) (radius 2)

```
## (-2, 8)
```

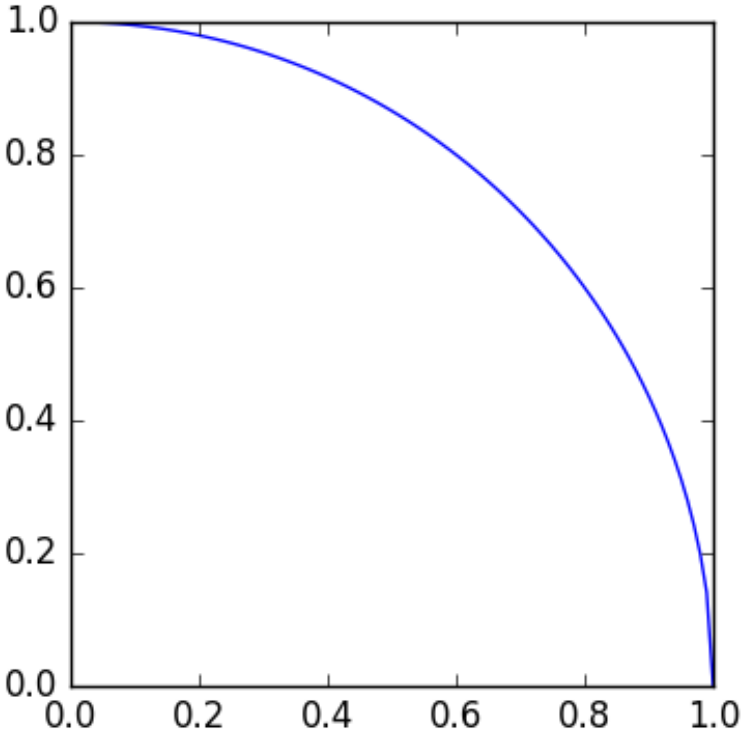
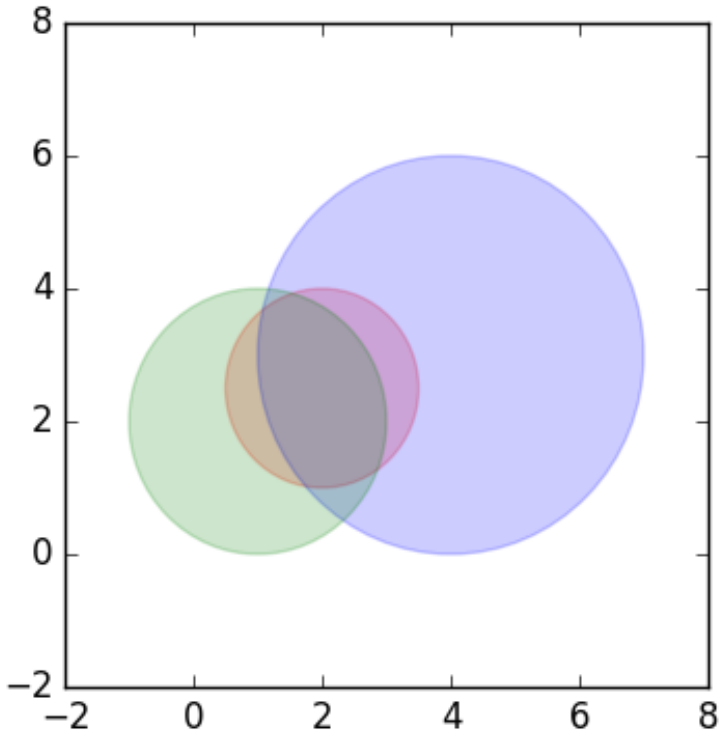


Figure 1: quarter circle

```
## (-2, 8)
```

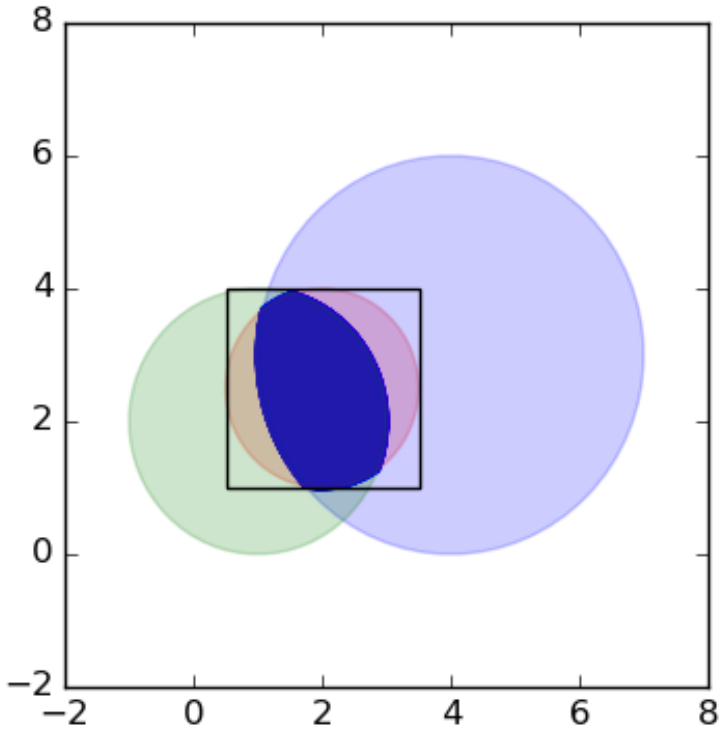



```
import numpy.random as npr
import numpy as np
c = ((2,2.5),(4,3),(1,2))
r = (1.5,3,2)
npr.seed(101)
x = npr.uniform(c[0][0]-r[0],c[0][0]+r[0],size=1000000)
y = npr.uniform(c[0][1]-r[0],c[0][1]+r[0],size=1000000)
tests = np.tile(True,10**6)
def incirc(x,y,ctr,radius):
    dsq = (x-ctr[0])**2+(y-ctr[1])**2
    return(dsq<radius**2)
for c0,r0 in zip(c,r):
    tests = tests & incirc(x,y,c0,r0)
    print(np.mean(tests))
```

```
## 0.78613
## 0.658081
## 0.48976
```

```
print(r[0]**2*np.mean(tests))
```

```
## 1.10196
```



Conway's game of life

Rules:

- set up a rectangular array of cells
- set some cells to 1 ("alive") and some to 0 ("dead")
- live cells with exactly 2 or 3 neighbours live; others die
- dead cells with exactly 2 neighbours become living; others stay dead

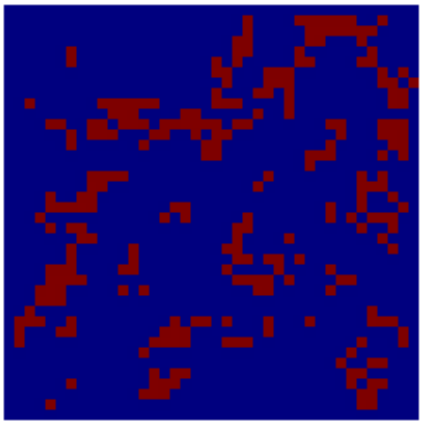
pieces

- `life_init(size, init_dens)`: set up a `size*size` 2D array of zeros; set a density `init_dens` to 1
- `life_step(w,nw)`: run the Conway rules on an array `w` and return the new array `nw`
- `count_nbr(w,i,j)`: count the number of neighbours in the 3x3 square around `w[i,j]`
- `life_show(w, ax)`: plot an image of the current array `w` on axes `ax`

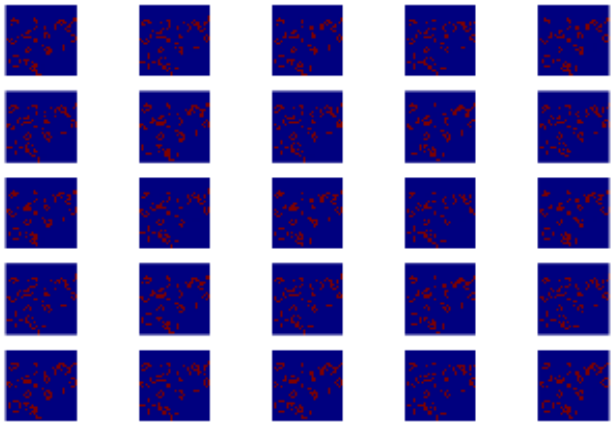
examples

```
import os
import matplotlib.pyplot as plt
os.chdir("../code")
from life import *
os.chdir("../notes")
w = life_init()
fig, ax = plt.subplots()
life_show(w, ax)
fig.savefig("pix/life1.png")
```

```
nw = w.copy()
(nw, w) = life_step(w,nw)
life_show(w, ax)
fig.savefig("pix/life2.png")
```



```
fig, ax = plt.subplots(5,5)
for i in range(ax.shape[0]):
    for j in range(ax.shape[1]):
        nw, w = life_step(w,nw)
        life_show(w, ax[i][j])
fig.savefig("pix/life3.png")
```



```
fig, ax = plt.subplots(5,5)
for i in range(ax.shape[0]):
    for j in range(ax.shape[1]):
        for k in range(10):
            life_show(w, ax[i][j])
            nw, w = life_step(w,nw)
fig.savefig("pix/life4.png")
```