Computational Physics

Adaptive, Multi-Dimensional, & Monte Carlo Integration

Feb 21, 2019

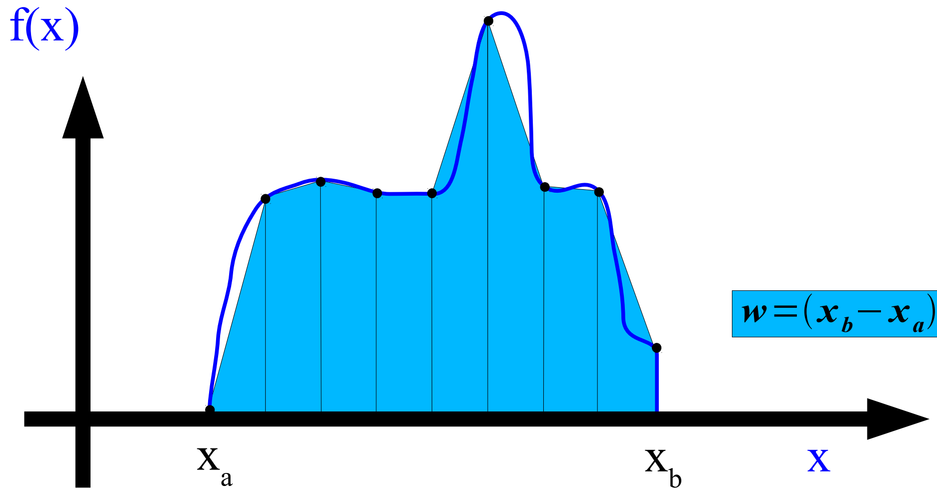
http://hadron.physics.fsu.edu/~eugenio/comphy/

eugenio@fsu.edu

Series Integration

review from last lecture

trapezoidal rule



$w = (x_b - x_a)/N$

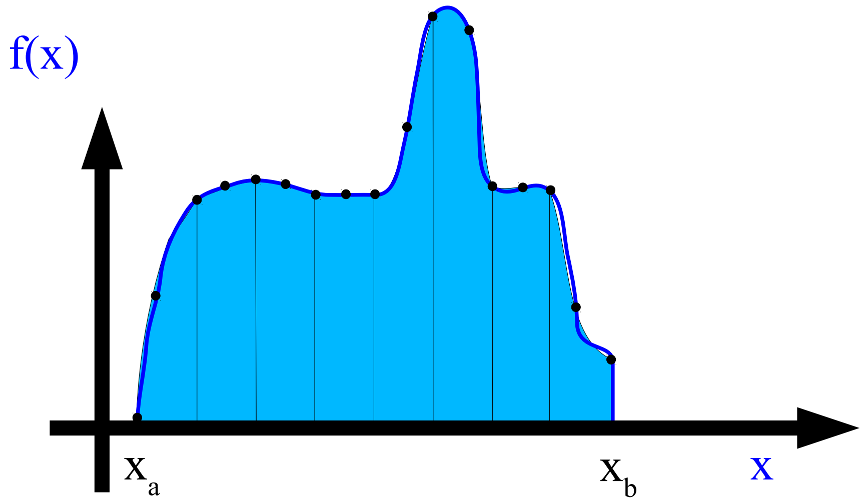
$$\int_{x_a}^{x_b} f(x)\,dx = I_N + O(w^2)$$

$$I_N = \frac{w}{2} * \left(f(x_a) + f(x_b) + 2 \sum_{k=1}^{N-1} f(x_a + k w) \right)$$

$$\epsilon_2 = \frac{1}{3} \left| I_{N_2} - I_{N_1} \right|$$

error on 2nd integration

Simpson's rule



$$\int_{x_a}^{x_b} f(x)\,dx = I_N + O(w^4)$$

$$I_N = \frac{w}{6} \Big[f(x_a) + f(x_b) + 2 \sum_{k=1}^{N-1} f(x_a + k w) + 4 \sum_{k=0}^{N-1} f(x_a + w(k + 1/2)) \Big]$$

$$\epsilon_2 = \frac{1}{15} \left| I_{N_2} - I_{N_1} \right|$$

error on 2nd integration

Adaptive Integration

Integration with just enough steps, N, to achieve the accuracy we want

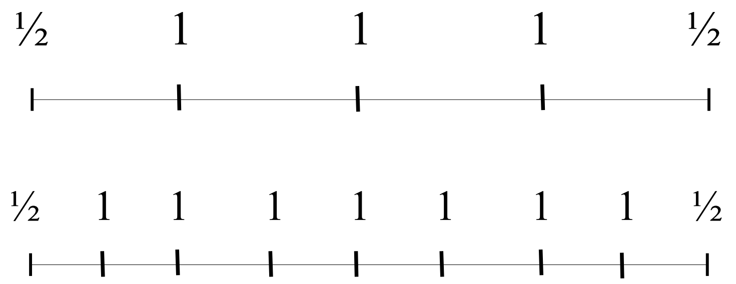
- Decide on the integration accuracy
- Evaluate the integral with a small number of steps N_1
- Then double the number $N_2 = 2N_1$, & evaluate integral again
- Calculate the error on I_2
    - use

$$\epsilon_2 = \frac{1}{3}|I_2 - I_1|_{trapezodial} \quad \textbf{or} \quad \epsilon_2 = \frac{1}{15}|I_2 - I_1|_{Simpson}$$

- If error is to within accuracy then we are finished, otherwise repeat doubling of integration steps until desired accuracy is reached

Adaptive Integration

Doubling the number of integration steps with half the calculations

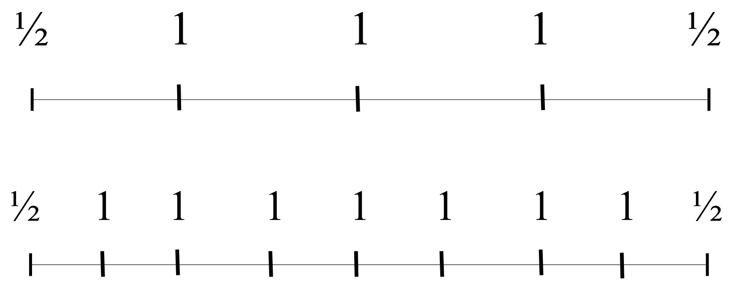


doubling effectively adds an additional set of points halfway between the previous points

$$I_N = w\left(\frac{1}{2}f(x_a)+\frac{1}{2}f(x_b)+\sum_{k=1}^{N-1}[f(x_a+kw)]\right)$$

Adaptive Integration

Doubling the number of integration steps with half the calculations



doubling effectively adds an additional set of points halfway between the previous points

$$I_N = w\left(\frac{1}{2}f(x_a)+\frac{1}{2}f(x_b)+\sum_{k=1}^{N-1}[f(x_a+kw)]\right)$$

adaptive trapezoidal rule integration

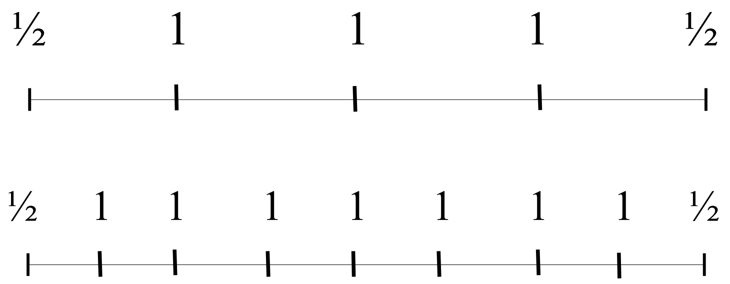
(extra points are the odd points)

$$I_2=\frac{1}{2}I_1+w_2*\sum_{\substack{k\,odd\\1\ldots N_2-1}} f(x_a+k\,w_2)$$

$$\mathbf{N_2=2N_1} \qquad w_2=\frac{w_1}{2} \qquad \epsilon_2 = \frac{1}{3}|I_2-I_1|$$

Adaptive Integration

Doubling the number of integration steps with half the calculations



doubling effectively adds an additional set of points halfway between the previous points

$$I_N = \frac{w}{6}\left[f(x_a)+f(x_b)+2\sum_{k=1}^{N-1} f(x_a+kw)+4\sum_{k=0}^{N-1} f(x_a+w(k+1/2))\right]$$

$$I_N = w\left(\frac{1}{2}f(x_a)+\frac{1}{2}f(x_b)+\sum_{k=1}^{N-1}[f(x_a+kw)]\right)$$

adaptive trapezoidal rule integration

(extra points are the odd points)

$$I_2=\frac{1}{2}I_1+w_2*\sum_{\substack{k\,odd\\ 1\ldots N_2-1}} f(x_a+kw_2)$$

$$N_2=2N_1 \qquad w_2=\frac{w_1}{2} \qquad \epsilon_2 = \frac{1}{3}|I_2-I_1|$$

adaptive Simpson's rule integration

(extra points are the new mid points minus odd endpoint point values)

$$I_2 = \frac{1}{2}I_1 - \frac{1}{3}w_2\sum_{\substack{k\,odd\\ k=1}}^{N_2-1} f(x_a+w_2 k) + \frac{2}{3}w_2\sum_{k=0}^{N_2-1} f(x_a+w_2(k+1/2))$$

$$N_2=2N_1 \qquad w_2=\frac{w_1}{2} \qquad \epsilon_2 = \frac{1}{15}|I_2-I_1|$$

Integrals over infinite ranges

$$\int_0^\infty f(x)\,dx$$

Solve by changing variables: $z = \frac{x}{1+x} \rightarrow x = \frac{z}{(1-z)}$ & $dx = \frac{dz}{(1-z)^2}$

$$\int_0^\infty f(x)\,dx = \int_0^1 \frac{1}{(1-z)^2} f\left(\frac{z}{1-z}\right) dz$$

Integrals over infinite ranges

$$\int_0^\infty f(x)\,dx$$

solution is to change variables: $z=\frac{x}{1+x} \rightarrow x=\frac{z}{(1-z)}$ & $dx=\frac{dz}{(1-z)^2}$

$$\int_0^\infty f(x)\,dx = \int_0^1 \frac{1}{(1-z)^2} f\left(\frac{z}{1-z}\right) dz$$

We can make two changes of variables: y = x - a & z = y/(1+y)

to calculate:
$$\int_a^\infty f(x)\,dx = \int_0^1 \frac{1}{(1-z)^2} f\left(\frac{z}{1-z}+a\right) dz$$

Multi-Dimensional Integration

“Divide and Conquer”

$$\begin{tikzcd}
\int_{x_a}^{x_b}\int_{y_a}^{y_b} f(x,y)\,dy\,dx \arrow[dr] & & f(x)=\int_{y_a}^{y_b} f(x,y)\,dy \\
& \int_{x_a}^{x_b} f(x)\,dx \arrow[ur] &
\end{tikzcd}$$

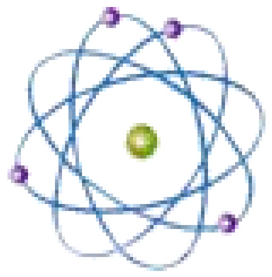
Solve by Series Integration

Trapezoidal
Simpson's Rule

For N points in each integral calculation there are N^2 calculations

Multi-Dimensional Integration

Example: Atomic Physics



$$I=\int_0^1 dx_1\int_0^1 dx_2\ldots\int_0^1 dx_{12}\, f(x_1, x_2, \ldots x_{12})$$

^{4}Be

3 Dimension/electron * 4 electrons = 12 Dimensions

For 100 steps in each integration there are $100^{12} = 10^{24}$ calculations

Assuming 1 Giga evaluations/sec

It would take over 10^7 years!!!!

Monte Carlo Integration

Using Random Numbers to Solve Integrals

Monte Carlo methods provide an alternative method of calculating an integral.

Random Numbers

Pseudo-Random Numbers

The numbers are pseudo-random in the sense that they are generated deterministically from a seed number, but are distributed in what has statistical similarities to random fashion.

Random Numbers

Pseudo-Random Numbers

The numbers are pseudo-random in the sense that they are generated deterministically from a seed number, but are distributed in what has statistical similarities to random fashion.

Python modules for generating random numbers:

- **import random:**
    - a smaller set of functions for random numbers
- **import numpy.random:**
    - a more complete set of utilities with many generating functions along with array manipulation capabilities

Random Numbers

Pseudo-Random Numbers

The numbers are pseudo-random in the sense that they are generated deterministically from a seed number, but are distributed in what has statistical similarities to random fashion.

Python modules for generating random numbers:

- **import random:**
    - a smaller set of functions for random numbers
- **import numpy.random:**
    - a more complete set of utilities with many generating functions along with array manipulation capabilities

Numpy.random and random modules uses a *Mersenne Twister* algorithm to generate pseudorandom numbers which has become the generator of choice for serious physics calculations.

Let's always use the NumPy random module

NumPy's random module

```
import numpy as np

>>> np.random.seed(136)
```

The seed is an integer value. Any program that starts with the same seed will generate exactly the same sequence of random numbers each time it is run. This is useful for debugging programs but otherwise **not needed as the seed is uniquely set each time the program executes**.

Note: one could also import via:

```
import numpy.random as rnd
>>> rnd.seed(136)
```

NumPy's random module

Generate random integers in the range [min, max)

```
>>> np.random.randint(5,10)
8
```

Generates a single random number in [0.0, 1.0)

```
>>> np.random.random()
0.70110427435769551
```

Generate an array of random numbers in the interval [0.0, 1.0)

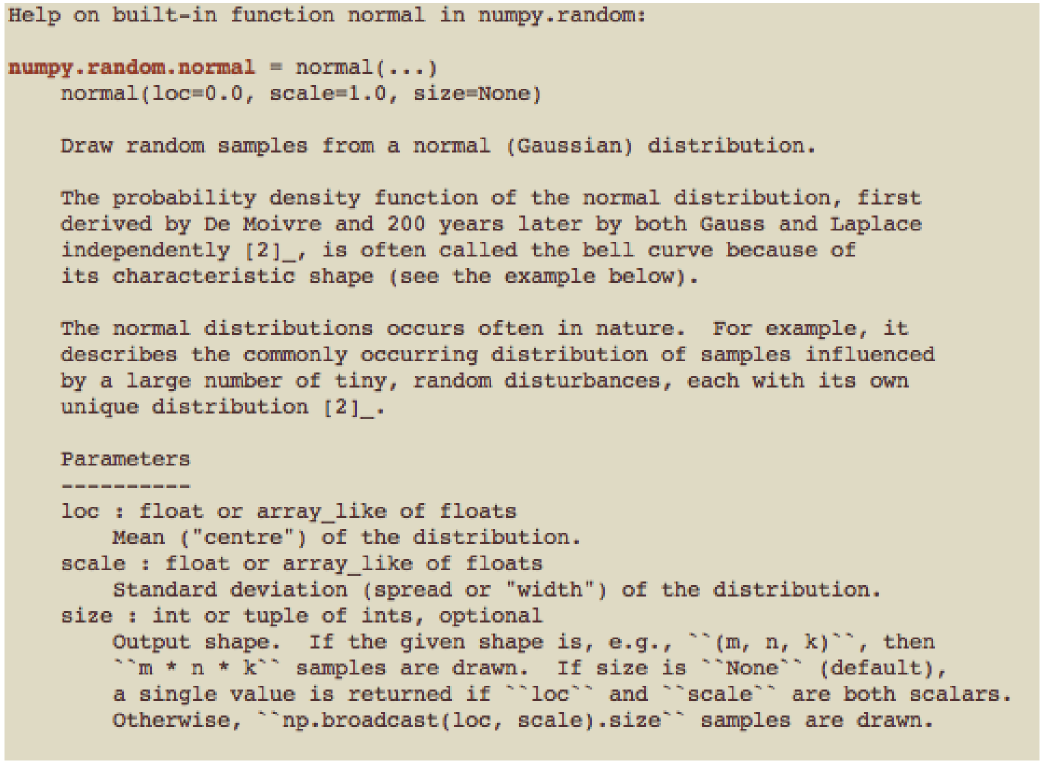
```
>>> np.random.rand(1)
array([ 0.73549029])
>>> np.random.rand(5)
array([ 0.6652181 ,  0.58861746,  0.8514131 ,  0.68607923,
0.8785746 ])
>>> np.random.rand(2,3)
array([[ 0.81698429,  0.632073  ,  0.10512043],
       [ 0.88226248,  0.47654622,  0.45082853]])
```

NumPy's random module

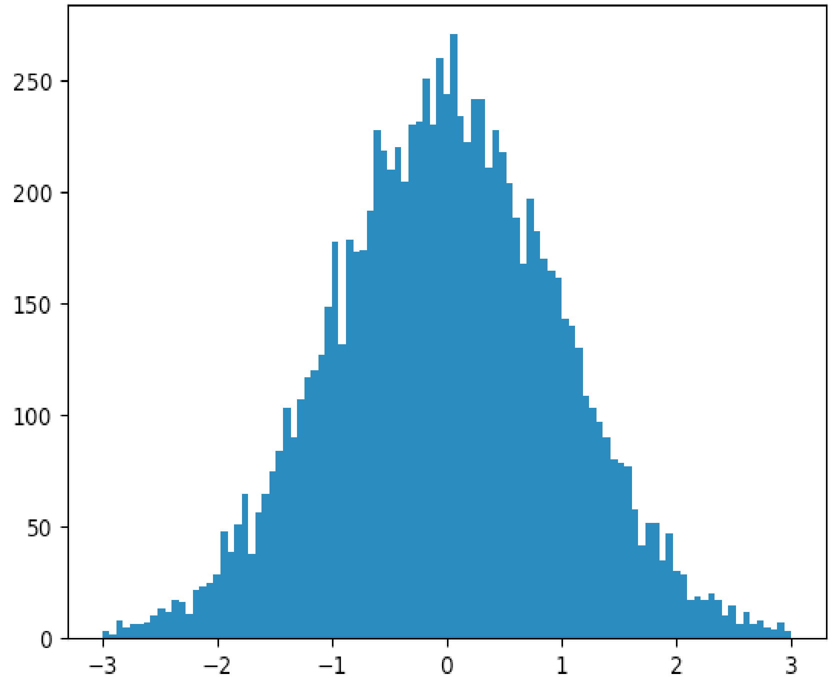
```
hpc-login% pydoc numpy.random
 ...
 ...
DESCRIPTION
    ========================
    Random Number Generation
    ========================

    ...
    ...

    ==================== =========================================================
    Univariate distributions
    ==============================================================================
    beta                 Beta distribution over ``[0, 1]``.
    binomial             Binomial distribution.
    chisquare            :math:`\chi^2` distribution.
    exponential          Exponential distribution.
    f                    F (Fisher-Snedecor) distribution.
    gamma                Gamma distribution.
    geometric            Geometric distribution.
    gumbel               Gumbel distribution.
    hypergeometric       Hypergeometric distribution.
    laplace              Laplace distribution.
    logistic             Logistic distribution.
    lognormal            Log-normal distribution.
    logseries            Logarithmic series distribution.
    negative_binomial    Negative binomial distribution.
    noncentral_chisquare Non-central chi-square distribution.
    noncentral_f         Non-central F distribution.
    normal               Normal / Gaussian distribution.
    pareto               Pareto distribution.
    poisson              Poisson distribution.

     ...
     ...
```

Generating random distributions

hpc-login-24 % **pydoc numpy.random.normal**



```
Help on built-in function normal in numpy.random:

numpy.random.normal = normal(...)
    normal(loc=0.0, scale=1.0, size=None)

    Draw random samples from a normal (Gaussian) distribution.

    The probability density function of the normal distribution, first
    derived by De Moivre and 200 years later by both Gauss and Laplace
    independently [2]_, is often called the bell curve because of
    its characteristic shape (see the example below).

    The normal distributions occurs often in nature.  For example, it
    describes the commonly occurring distribution of samples influenced
    by a large number of tiny, random disturbances, each with its own
    unique distribution [2]_.

    Parameters
    ----------
    loc : float or array_like of floats
        Mean ("centre") of the distribution.
    scale : float or array_like of floats
        Standard deviation (spread or "width") of the distribution.
    size : int or tuple of ints, optional
        Output shape.  If the given shape is, e.g., ``(m, n, k)``, then
        ``m * n * k`` samples are drawn.  If size is ``None`` (default),
        a single value is returned if ``loc`` and ``scale`` are both scalars.
        Otherwise, ``np.broadcast(loc, scale).size`` samples are drawn.
```



```
>>> import numpy as np
>>> import matplotlib.pyplot as plt
>>>
>>> plt.hist(numpy.random.normal(size=10000), bins=numpy.linspace(-3,3,100))
>>> plt.show()
```

Monte Carlo Integration

- Two of the most common ways to employ Monte Carlo techniques are
    - Monte Carlo Sampling method
    - Monte Carlo Mean-Value method

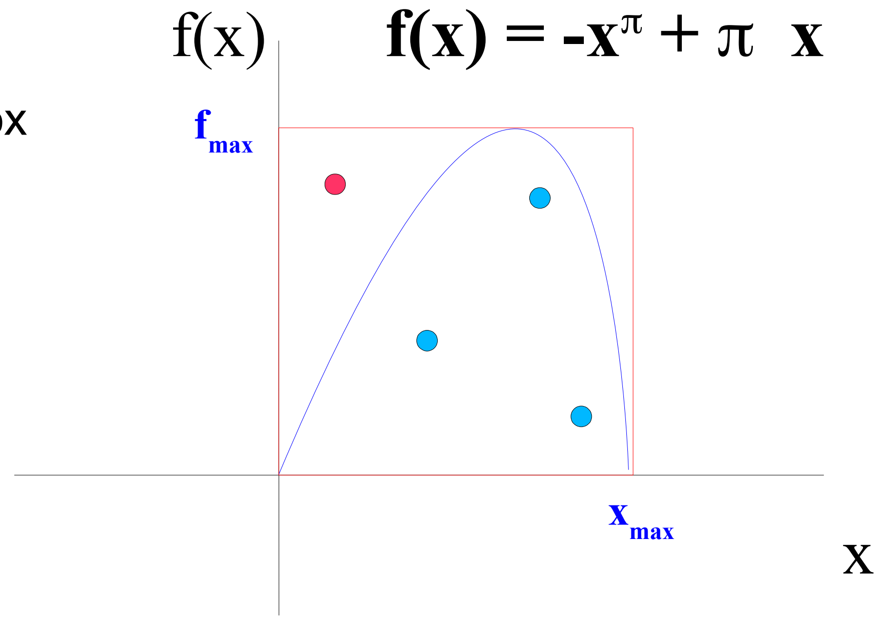
Monte Carlo Sampling Method

"random marks"

- Box off the region of integration
  - calculate the area of the box
- Randomly place points in the box
- Count # of points in the box vs # under the function
- $\text{Area}_{f(x)} = \text{Area}_{box} * N_{f(x)}/N_{box}$

$N_{f(x)} = \#$ ●

$N_{box} = \#$ ● $+ \#$ ●



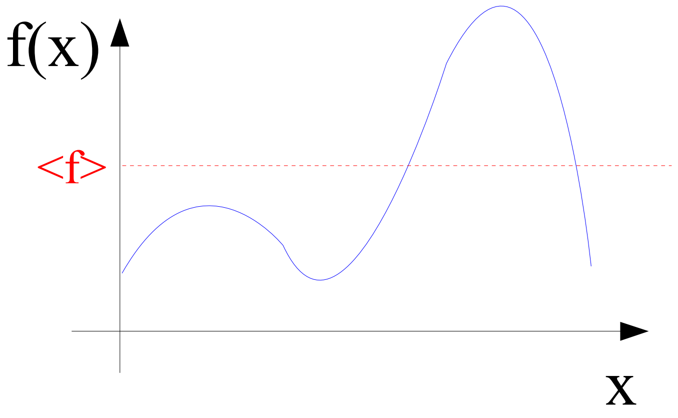
Monte Carlo Mean-Value Method

"The Work Horse"

Integration using the Monte Carlo method is done by averaging the value of the function at randomly selected points within the integration interval

$$\boxed{I = \int_a^b dx\, f(x) = (b-a)\langle f \rangle}$$

$$\langle f \rangle \simeq \frac{1}{N} \sum_{i=0}^{N} f(x_i)$$



Statistical Error

$$\delta I \sim \sigma_{\bar{f}}$$

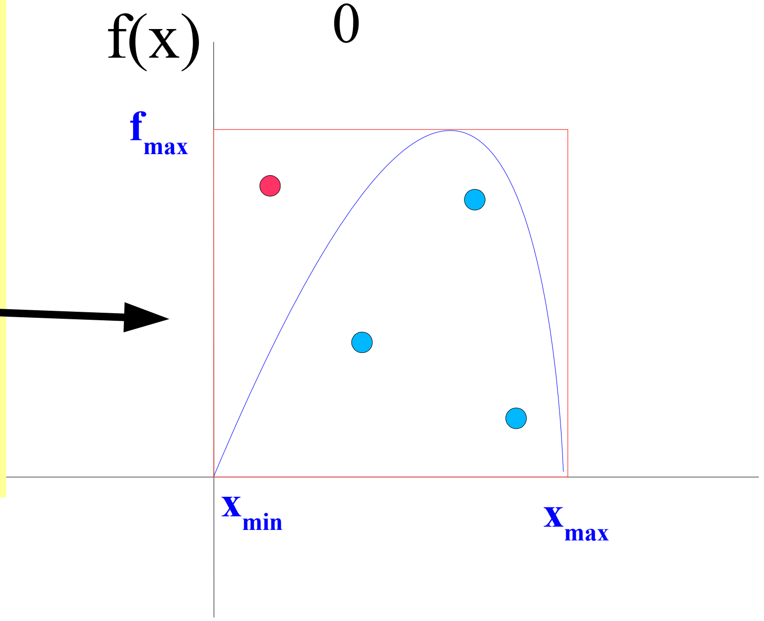
standard deviation of mean

$$\sigma_{\bar{f}} = \sigma_f / \sqrt{N}$$

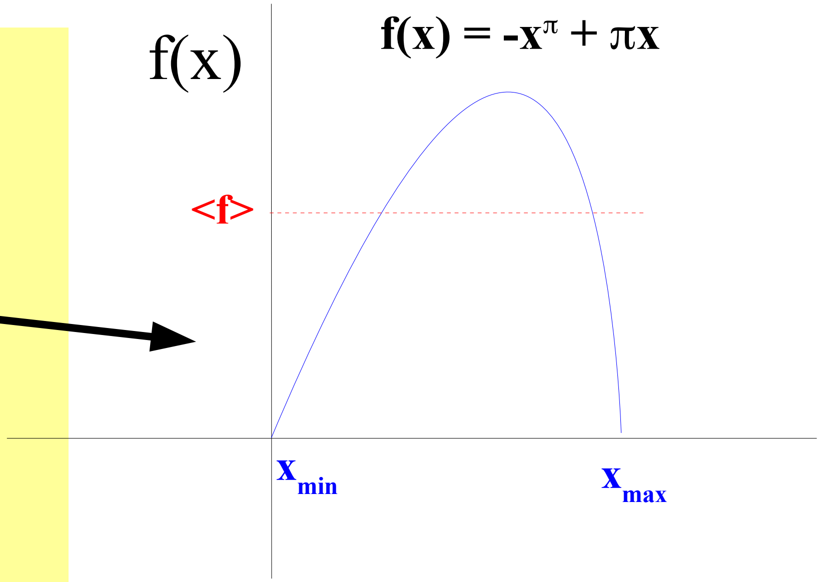
Example: MC integration

$$\int_0^{e^{\frac{\ln(\pi)}{(\pi-1)}}} \pi x - x^\pi$$

```
import numpy.random as rnd
nUnder = 0
for _ in range(nPoints):
    x = xMin + (xMax - xMin)*rnd.random()
    fRand = fMin + (fMax - fMin)*rnd.random()
    if fRand < f( x ):
        nUnder += 1
 print(“Integration by Samples = ”, \
      fMax*(xMax - xMin) * nUnder/nPoints)
```



```
import numpy.random as rnd
sum = 0
for _ in range(nPoints):
    x = xMin + (xMax - xMin)*rnd.random()
    sum += f( x )

print(“Integration by Mean-Value = ”, \
      (xMax – xMin) * sum/nPoints )
```



Multi-Dimensional Monte Carlo

- Easy to generalize mean-value integration to many dimensions

$$I=\int_a^b dx\int_c^d dy\, f(x,y)\simeq(b-a)(d-c)*\frac{1}{N}\sum_i^N f(x_i,y_i)$$

Multi-Dimensional Monte Carlo

- Easy to generalize mean-value integration to many dimensions

$$I=\int_a^b dx\int_c^d dy\, f(x,y)\simeq(b-a)(d-c)*\frac{1}{N}\sum_i^N f(x_i,y_i)$$

```
import numpy.random as rnd
def integrateMC(func, dim, limit, N=100):
    I,sum = 1/N,0
    for n in range(dim):
        I *= (limit[n][1] - limit[n][0])
    for k in range(N):
        x = []
        for n in range(dim):
            x += [limit[n][0] + (limit[n][1] - limit[n][0])*rnd.random()]
        sum += func(x)
    return I*sum
```

multi-dimensional Monte Carlo integration implementation

```
def f(x):
    return np.sin(x[0] * x[1]**2)
```

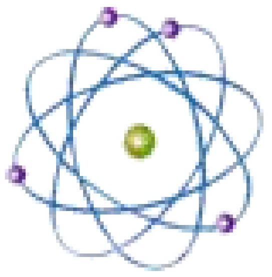
integrand function $f(x,y)=\sin(xy^2)$

```
dim,limit = 2,[[0,np.pi],[0,np.pi]]
print(integrateMC(f, dim, limit))
```

main part of program

Multi-Dimensional Integration
via MC mean value

Example: Atomic Physics



$$I = \int_0^1 dx_1 \int_0^1 dx_2 \ldots \int_0^1 dx_{12} \, f(x_1, x_2, \ldots x_{12})$$

3 Dimension/electron * 4 electrons = 12 Dimensions

^{4}Be

$$\simeq (1-0)^{12} * \frac{1}{N} \sum_i^N f(x_{1,}^{\ i} \ \ x_{2,}^{\ i} \ \ \ldots \ \ x_{12}^{\ i})$$

For N=10^6 random points in the MC integration there are ~10^6 calculations

Assuming 1 Giga evaluations/sec

It would take ~ 10^{-3} sec

compare to 10^7yrs

Monte Carlo Error

- Monte Carlo error is Statistical
  - Error decreases as $\mathbf{1/\sqrt{N}}$

Mean-Value Integration is an ordinary statistical mean

$$\bar{x} = \frac{1}{N}\sum_{i=1}^{N} x_i$$

The error on the mean is the standard deviation of the mean (SDOM)

$$\sigma_{\bar{x}} = \frac{\sigma_x}{\sqrt{N}}$$

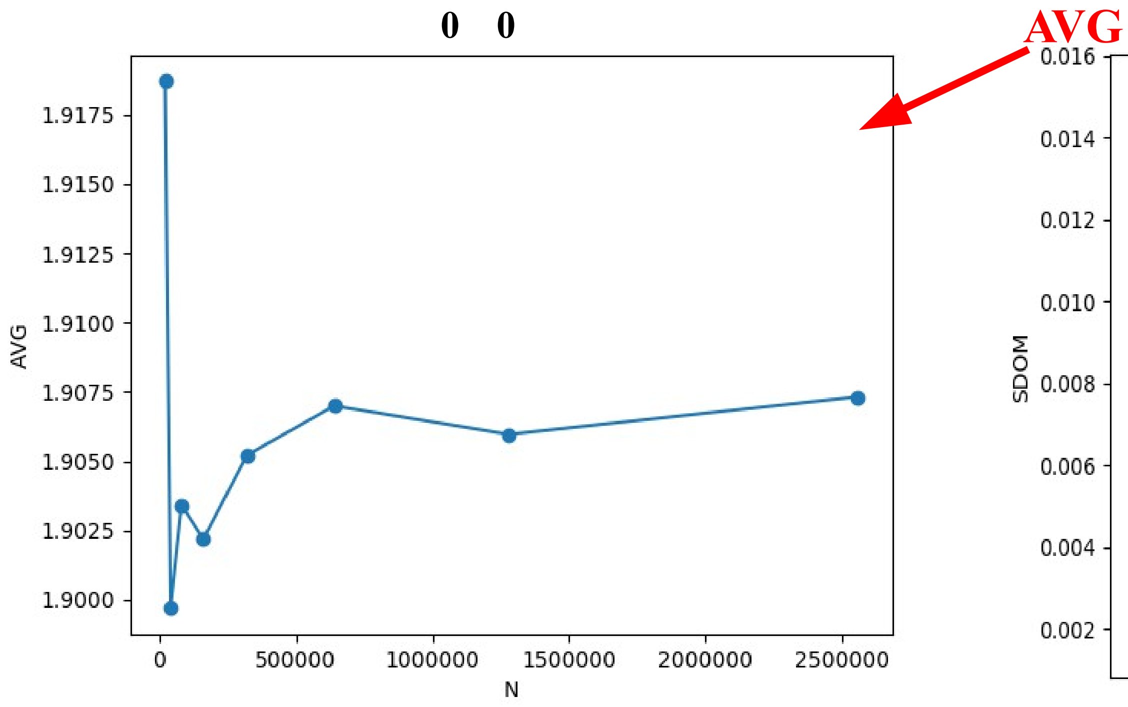
With the standard deviation σ_ξ defined as usual

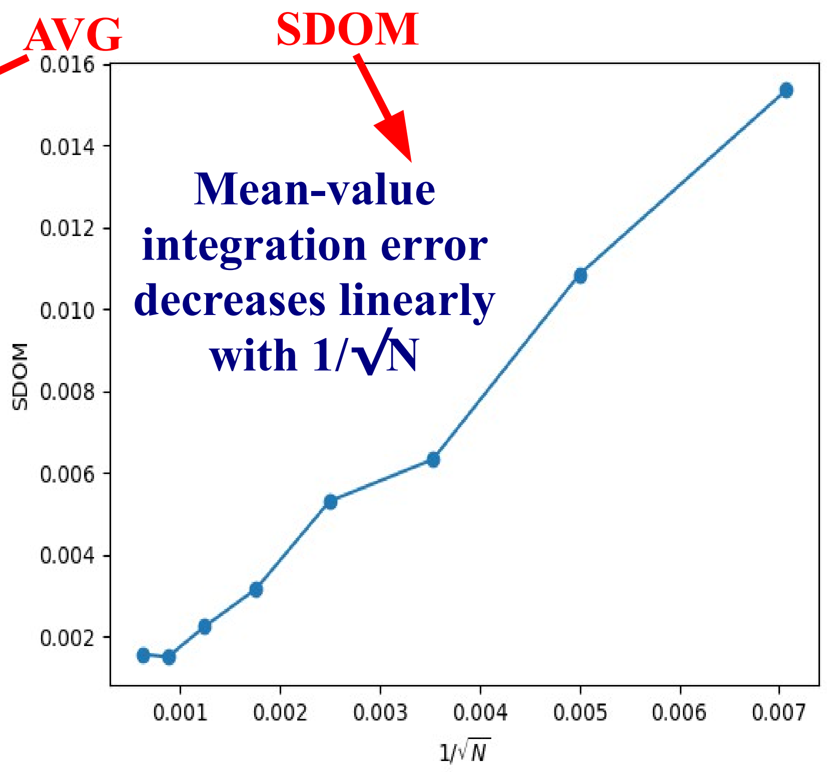
$$\sigma_x = \sqrt{\frac{1}{N-1}\sum (x_i - x_b)^2}$$

Note: $\sum[(x_i-\bar{x})^2] = [\sum (x_i)^2] - N\bar{x}^2$ **is very useful in computing SDOM!**

Monte Carlo Error

$$\int_0^{\pi}\int_0^{\pi} \sin(x*y^2)\,dx\,dy = 1.9051 \pm 0.0006$$





```
nSamples, n = 10000, 20
While nSamples < MaxSamples
    vSum, vSumSq = 0.0, 0.0
    for _ in range(n):
        v = integrateMC(func, nDim, limits, nSamples)
        vSum += v
        vSumSq += v*v
     avg = vSum/n
     SDOM = sqrt(1/n) * sqrt( 1/(n - 1) * (vSumSq - n*avg*avg) )
     nSamples *= 2
```

Averaged over 20 samples for each value of number of Monte Carlo samples

See python code at
http://hadron.physics.fsu.edu/~eugenio/comphy/examples/mcint.py

Let's get working on #5