PART I. The Monte Carlo Method

Monte Carlo methods are a widely used class of computational algorithms (or methods). In its simplest form, the Monte Carlo algorithm can be used for simple computations such as approximating a scalar, an area or volume. More complicated variants of the algorithm can be used in areas such as computational physics, chemistry, applied mathematics, for example.

Monte Carlo methods were originally called statistical sampling methods due to the use of randomness.

Historically these methods were developed by John von Neumann, Enrico Fermi, Stanislaw Ulam and Nicholas Metropolis.

To describe the Monte Carlo method in its simplest form consider the following problem. Suppose your neighbor has put in a swimming pool and because you are unfortunately afflicted with the syndrome "keeping up with the Joneses" you desperately want to determine how big his pool is so that you can put in a larger one. Unfortunately, your neighbor has erected a tall fence around his entire backyard and you are unable to see over it. Since your lots are the same size you know the total area of his back yard but not the size of his pool. One day, when you know your neighbor is not home, you collect a lot of ice cubes. You throw each ice cube in a random manner into your neighbor's back yard. Every time you hear an ice cube land in the pool you record this. After you have thrown all the ice cubes you take the fraction of the total number of ice cubes that landed in the pool times the area of his backyard. This will give you an estimate for the area of the pool.

$$\frac{\text{number of ice cubes landing in pool}}{\text{total number of ice cubes thrown}} \times \text{area of backyard}$$

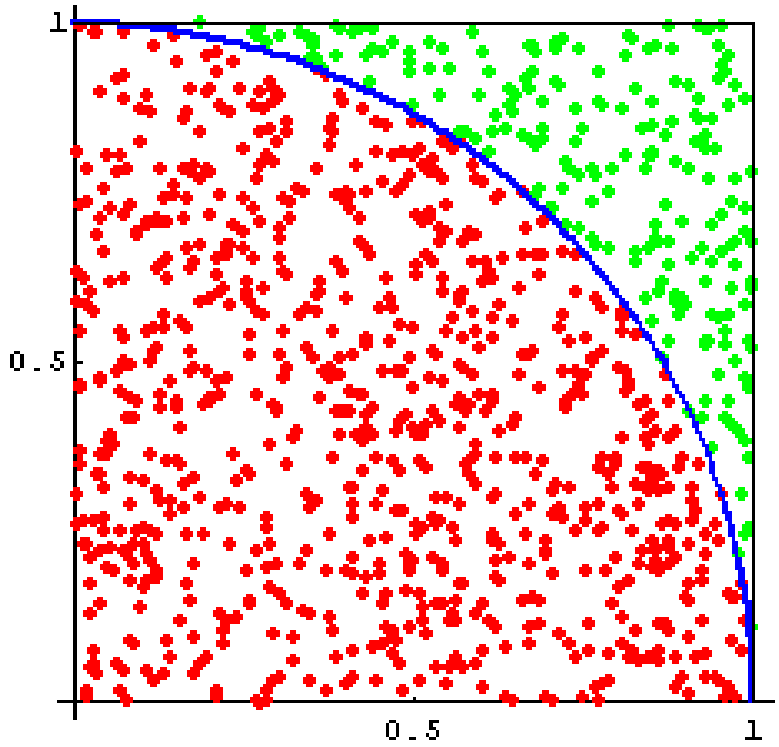
As we throw more and more ice cubes, we expect to get a better and better approximation to the size of the pool.

Approximating π using a simple Monte Carlo approach

- We know that the area of the $unit$ circle (with radius 1) is just π. So the area of the portion of the unit circle in the first quadrant is just $\pi/4$.
- If we can choose random numbers x, y where $0 \leq x \leq 1$, $0 \leq y \leq 1$ then the point (x, y) will be our "ice cube".
- If the point (x, y) lies in or on the circle (i.e., $x^2 + y^2 \leq 1$) then we record this as a "hit".
- The area of the circle (π) is then approximated by

$$4 \times \frac{\text{points satisfying } x^2 + y^2 \leq 1}{N}$$

where N is the total number of random points (x, y) that we generate.



- This method is easily implemented in a computer program since Fortran has a built-in random number generator which we can use and there are constructs for repetitive calculation and conditions.
- However, we will see that we need a huge number of points to get even 4 digits of accuracy in our approximation. Thus, we need to implement this algorithm into a computer program.

- So to write the body of our code (after `program, implicit none` and declaration statements) we need to
  - specify the number of random points to be generated
  - generate random numbers x, y between zero and one
  - check to see if $x^2 + y^2 \leq 1$; if it is, increment counter
  - after we have generated all random points then we compute our approximation to π from the formula

$$4\left(\frac{\text{points satisfying } x^2+y^2\leq 1}{N}\right)$$

    and output results

Note that we will need

- a `do` loop
- a conditional (`if` statement) for checking if $x^2 + y^2 \leq 1$

Generating a Random Number

- The last thing we need before we can write our program to approximate π using a simple Monte Carlo approach is to determine how we can generate a random number.
- We don't want to get into the details of how the intrinsic procedure in fortran for generating a random number works. At this point, we are willing to accept the fact that it generates a sufficiently random number (actually a pseudorandom number).
- The random number intrinsic procedure is more complicated than say a trig intrinsic function; in fact it is what we call a intrinsic subroutine. For now I will just tell you how to use it.

```
call random_number ( output variable )
```

- This command generates a single random number between zero and one and stores this number in the output variable you specify.
- For example

```
call random_number (x )
```

  generates a random number and stores it in the variable `x`.
- What if we want a random number between 0 and 100? We first generate a random number between 0 and 1 and then scale to the interval (0,100)
- For example, if we have generated 0.56777 between 0 and 1, then between 0 and 100 it becomes 56.777
- Fortran statements:

```
call random_number (x)
x = x * 100.0
```

- What if we want a random number between 1 and 2. We first generate a random number between 0 and 1 and then translate to the interval (1,2)
- For example, if we generate 0.56777 between 0 and 1, then between 1 and 2 it just becomes 1.5677

- Fortran statements:

```
call random_number (x)
x = x + 1.0
```

- What if we want a random number between 1 and 100? Then we translate and scale. Specifically, we use the linear mapping $y = 1+99x$ which satisfies $y(0) = 1$, $y(1) = 100$
- For example, if we generate 0.56777 between 0 and 1, we can first scale it to (0,99) by $\hat{x} = 99 * x$ and then translate it to (1,100) by adding 1 to it; i.e., $\hat{x} = 99x + 1$.
- Fortran statements:

```
call random_number (x)
x = 99.0* x + 1.0
```

- If we want a random number in the interval $[a, b]$ and we have a random number in the interval $[0, 1]$ we simply use the linear transformation

$$\hat{x} = (b - a)x + a$$

- Note that when $x = 0$, $\hat{x} = a$, the left endpoint, and when $x = 1$, $\hat{x} = b$, the right endpoint and since the mapping is linear it maps e.g., the midpoint of $[0, 1]$ to the midpoint of $[a, b]$, etc.
- For classwork you will modify a code which generates a random number in [0,1] to generate a random number in a given interval.
- If we want a random number (x, y) in the plane with each coordinate between 0 and 1, we simply call the routine twice.

```
call random_number (x)
call random_number (y)
```

Do we want to be able to reproduce our results?

- The problem with using the call to this intrinsic routine is that if we compile the code, and then run it two separate times we may get different answers! This is actually compiler dependent. Two Fortran compilers can handle the way the random number generator is initialized differently.
- It is useful to be able to reproduce our results.
- To do this, we have to tell the random number routine where to (randomly) start (i.e., set the "seed"). This can be done in many ways. For now, just simply include this call in your program. At a later time you may investigate these routines further if they are of interest to you.
- The routine that we call is named `random_seed` and its argument will be a random number or a vector of random numbers.
- Note that we simply set the random seed at the first of our program.
- Now we will run a code to generate some random numbers: `generate_random_numbers.f90` . For classwork you will be asked to modify this to scale and translate the number.

Outline of Code for Approximating π

The basic structure of the code is

- program statement
- comments describing what program does and algorithm for doing it
- `implicit none` statement
- declarations
- body of code to carry out algorithm
- output the results
- end program statement

Of course the core part of the code is the implementation of the Monte Carlo algorithm for determining π. We will look at that in more detail now. First let's look at an example.

- First we will generate a random point (x, y); for example

$$(x, y) = (.655815, .513207)$$

- We now want to determine if this point is inside the unit circle (in the first quadrant). To do this we determine

$$z = x^2 + y^2 = .693475$$

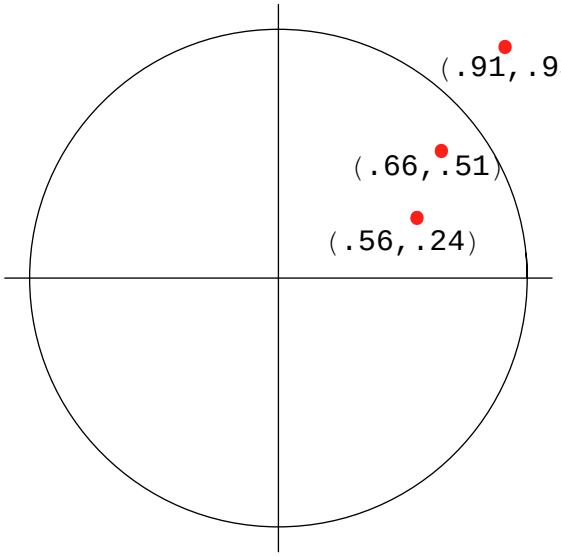
- So in this case the point is inside the circle since $.693475 \leq 1.0$ so increment the counter for the number of hits by 1.
- We repeat the process and select another random number

$$(x, y) = (.556763, .244819)$$

- In this case $x^2 + y^2 = .369922$ so we increment the counter for the number of hits again so its new value is 2.
- We repeat the process and select another random number

$$(x, y) = (.910643, .93098)$$

- In this case $x^2 + y^2 = 1.69599 > 1$ so we do nothing.
- We continue this procedure until we have generated a specific number of random points and we have a counter for the number of points inside the circle.
- We then approximate $\pi/4$ by the fraction of the total points that are in the given region. Thus our approximation to π is 4 times this fraction.
- In this example with only 3 points, our approximation to π is rather crude, $\pi \approx 4\left(\frac{2}{3}\right) = 2.667$.


(.91,.9
(.66,.51)
(.56,.24)

First, let's outline what we would have to do to implement the method and then we can translate that into fortran.

1. We need to know how many random points we are going to generate, call it `n_points`
2. For 1 to `n_points` we must
   - generate a random point (x, y)
   - test to see if $x^2 + y^2 \leq 1$
     - if $x^2 + y^2 \leq 1$ then we increment the number of points in the circle by 1
     - if $x^2 + y^2 > 1$ we do nothing
3. Compute the approximation to π by determining the fraction of the number of points that were in the circle and then multiplying by 4 (because we are only considering the first quadrant)

- Clearly we are going to need a `do` loop for step (2).
- Also we are going to need a conditional to test if $x^2 + y^2 \leq 1$.
- We will use the intrinsic routine `random_number` to generate a random (x, y) and `random_seed` to set the seed.
- We will read in the value of `n_points`
- We will set a variable, say `n_pts_inside` to be the counter for the number of points that lie inside the circle. So if $x^2 + y^2 \leq 1$ then

  ```
  n_pts_inside = n_pts_inside +1
  ```

  - This leads us to the interesting question of what happens the first time we get to this statement.
  - If `n_pts_inside` is not initialized to some value, then the compiler picks a value. Most machines will choose 0, but not all!
  - Consequently, it is imperative that you initialize variables like this to 0 first. Otherwise when you run your code on two different compilers you may get drastically different results!

Let's look at the do loop which is the heart of the code

```
n_pts_inside = 0      ! initialize points inside circle

do n = 1, n_points      ! loop over number of random points

    call random_number (x)      ! generate random x

    call random_number (y)      ! generate random y

    z = x * x + y * y

    if ( z <= 1.0 )  n_pts_inside = n_pts_inside + 1

end do
```

Note that we could also used an `if then` statement

```
if ( z <= 1.0 ) then
  n_pts_inside = n_pts_inside + 1
end if
```

Also, we didn't have to define `z` we could have simply said

```
if ( x**2 + y**2 <= 1.0 )  n_pts_inside = n_pts_inside + 1
```

There is not a unique way to write a code.

- Finally we need to approximate π which is the just the fraction of the points inside the region times 4.
- What is wrong with the following statement?

```
approx_pi = 4 * n_pts_inside / n_points
```

Because we want the actual fraction of points inside the region (not 0 or 1) and we don't want to use mixed arithmetic, the statement we need is

```
approx_pi = 4.0 * float ( n_pts_inside ) / float (n_points)
```

- Note that the number of points (`n_points`) and the counter for the number of points we found inside the circle (`n_pts_inside`) are both integers.

Sample Results

Now let's run a code to implement and get approximations as the number of points increases. Recall that $\pi = 3.141592654$

number of points used	our approximation	$\lvert\pi$-approximation$\rvert$
10	3.6	0.4584
100	2.96	0.2184
1000	3.136	.0055928
10,000	3.144	0.002407
100,000	3.13264	0.0089528

Using Monte Carlo to Approximate an Integral

- Suppose we want to evaluate $\int_a^b f(x)\,dx$
- If $f(x) \geq 0$ for $a \leq x \leq b$ then we know that this integral represents the area under the curve $y = f(x)$ and above the $x-$axis.
- Standard deterministic numerical integration rules approximate this integral by evaluating the integrand $f(x)$ at a set number of points and multiplying by appropriate weights.
  - For example, the midpoint rule is

$$\int_a^b f(x)\,dx \approx f\left(\frac{a+b}{2}\right)(b-a)$$

  - Simpson's rule is

$$\int_a^b f(x)\,dx \approx \frac{b-a}{6}\left[f(a) + 4f\left(\frac{a+b}{2}\right) + f(b)\right]$$

To approximate the integral using Monte Carlo which is a probablistic approach then we

- choose a simpler region (such as a rectangle) which includes the area you want to determine
- generate a random point in the simpler region
- determine if random point is in desired region
- take area as fraction of area of simpler region

Note that some interpreted languages such as $Mathematica$ have a built-in command to perform a numerical integration using the Monte Carlo method.

- Lets consider a specific integral.
- We know from calculus that if $f(x) \geq 0$ on $[a, b]$ then

$$\int_a^b f(x)\, dx$$

  represents the area under the curve $y = f(x)$, above the x-axis, and between $x = a$ and $x = b$.
- Suppose we know that $0 \leq f(x) \leq M$ on the interval $[a, b]$
- For example, if $f(x) = 2\cos x$ and the interval is $[0, \frac{\pi}{2}]$ then we know the maximum value that $f(x)$ takes on is 2.
- We then select a random point (x, y) where $a \leq x \leq b$ and $0 \leq y \leq M$
- In the previous problem we checked to see if the point was in the circle; now we check to see if the point lies on or below the function $f(x)$ at the point (x, y); i.e., if it is in the desired area. How do we do this?

– First we generate two random points, call them (x, y). We do this as before.

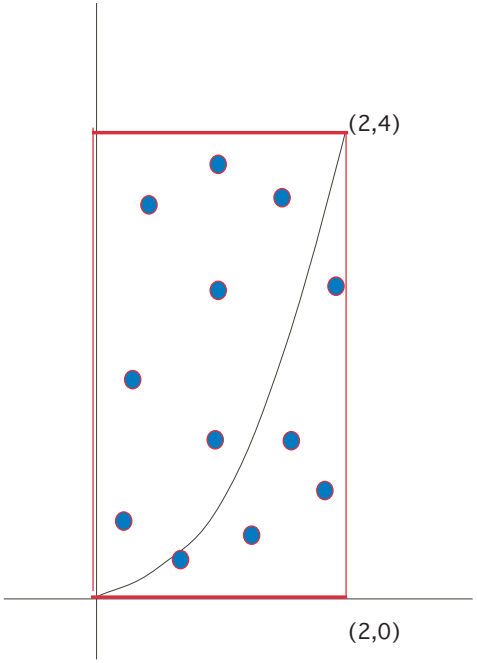
– Now at this given x-point we want to see if the random point y is below the curve. What must we test? We just want

$$y \leq f(x) \quad \text{where } x \text{ is the random point}$$

- As before, if the point is in the region we increment the counter for the number of "hits"
- Our approximation to the area, i.e., the value of the integral, is simply the usual fraction of the area of the testing region (whose area is $M(b-a)$).

$$\frac{\texttt{number of hits}}{\texttt{number of points}} \times M(b-a)$$

- Writing a code to approximate $\displaystyle\int_0^\pi \sin x \; dx$ will be your homework assignment.



12 random points generated and
5 in the desired region

Here we are approximating $\int_0^2 x^2 \, dx = \frac{x^3}{3}\Big|_0^2 = \frac{8}{3} = 2.67$ using Monte Carlo. Using 12 points we have the approximation is $\frac{5}{12}(8) = \frac{40}{12} = 3.33$ where 8 is the area of the box we are sampling in.

How do you choose the bounding box? It is not unique. We could have chosen

a larger box, say with height 6.

v

Classwork

- Modify the code `generate_random_numbers.f90` to do the following:
  - Define real variables `a` and `b` which will be the interval $[a, b]$ where you want the random number to be generated in.
  - Add statements to read these values from the screen.
  - Add a print statement to write out the statement like "We are generating random numbers in the interval" and print out interval
  - After the random number is generated, modify it by scaling and translating it.
  - Run your code to generate 10 random numbers on [50,60].
- Run the code `monte_carlo_pi.f90` for several increasing values of points to see the results.

Homework

- Use a template program `mc_integral.f90` to write a program to approximate the integral

$$\int_0^{\pi} \sin x \; dx$$

using a Monte Carlo approach. As written, this code will not compile but is a framework of how the code should be written. After your code is working, make a copy of the code and modify it to compute the integral

$$\int_1^{2} e^x \; dx$$

See Homework I.2