# Monte Carlo Algorithm Report

**David Hwang**

GitHub

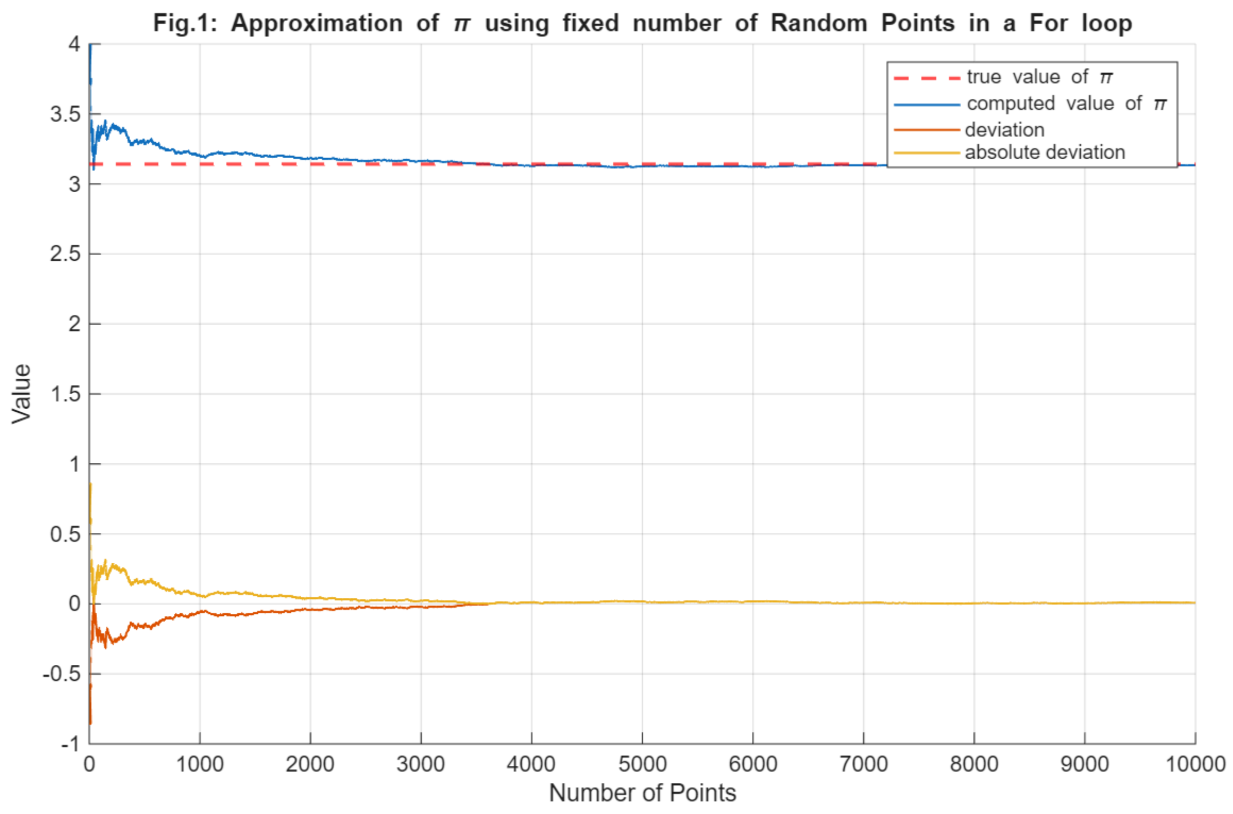
Through the Monte Carlo algorithm, we can estimate the value of π. By using the ratio of the number of random dots that fall within a unit square and a quarter of a circle in the quadrant with radius 1 centered on (0,0), we can approximate the value of π. Considering the area of the quarter circle is π/4 and the area of a unit square is 1, we can say that the number of random dots falling within a quarter of the circle divided by the number of random dots falling within the unit square is an approximation of π/4. Thus, we can estimate the value of π by multiplying 4 by the value that we earned in the last step.

- Task 1

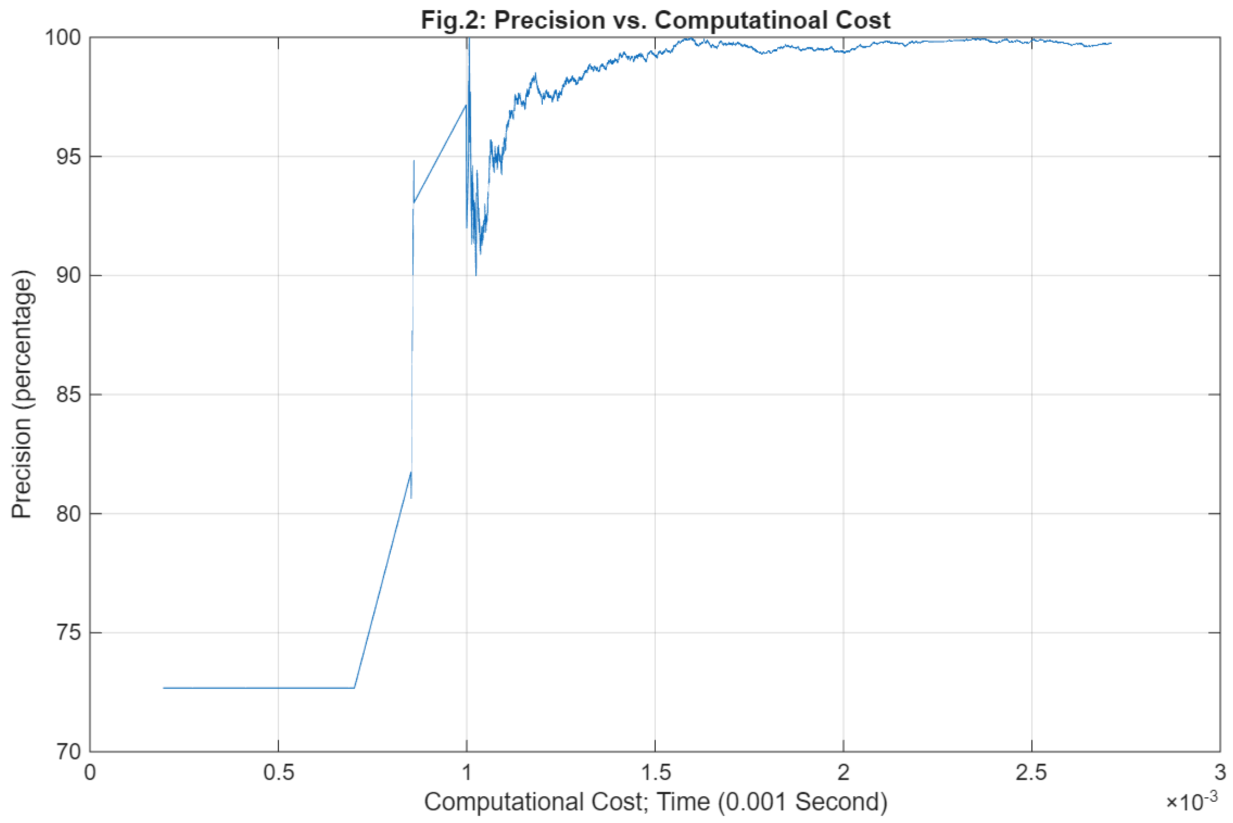
For task 1, I used a "for" loop to compute π with 1000 fixed random points. Using the for loop, every for loop step, I generated a random point inside the unit square, [0,1]x[0,1]. Also, by using an "if" function to confirm the distance of the generated dot from the origin (0,0) every step, I counted how many points went inside the area of a quarter circle with radius 1 centered at the origin. Since dots that have a distance from the origin larger than 1, which is the radius of a unit circle, are located outside of the quarter, I used the distance formula $d = \sqrt{(x - 0)^2 + (y - 0)^2}$ in the IF function to count the points inside the circle.
In every step, I calculated the computed value of π. Since the ratio of the area of the unit square and the quarter circle is 1: $\frac{\pi}{4}$, I divide the number of generated random points inside the circle by the number of whole random points and multiply by 4 to approximately compute the value of π in every step.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% Task 1: Using "for" loop
%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% parameter
N=10000; % fixed number of random points
count=0; % counter % the number of points inside the circle with radius 1
com_pi=zeros(1,N); % vector for computed value of π
Num=1:1:N; % Number vector % vector of number point
time=zeros(1,N); % excution time vector
T=0; % T is a number to save the excution time for each step.
dots=rand(2,N); % generate N-coupled random dots
x=dots(1,:); y=dots(2,:); % use 1st row as x-coordinate and 2nd row as
y-coordinate
% main equation
```

```
for i=Num
   tic;
   if sqrt(x(i)^2+y(i)^2)<=1 % the condition that random points are inside the
circle.
       count = count+1;
   end
   com_pi(i)=count/i*4; % the area of the circle is π/4. the area of the squre
is 1. then to get π, we need to muliply 4 here.
   T=T+toc;
   time(i)=T; % measure the execution time for each point counts
end
error=pi-com_pi; % deviation vector
abs_er=abs(error); % absolute deviation vector
precision=zeros(1,N);
for i = Num
   % The percentage precision is 100 minus the percentage error
   precision(i) = (1 - (abs_er(i)/pi))*100;
end
% plot
figure(1); clf; hold on;
yline(pi,'r--', 'LineWidth', 1.5); plot(Num,com_pi, 'LineWidth',1);
plot(Num, error, 'LineWidth', 1); plot(Num, abs_er, 'LineWidth', 1);
xlabel('Number of Points'); ylabel('Value');
legend('true value of \pi','computed value of \pi', 'deviation', 'absolute
deviation')
title('Fig.1: Approximation of \pi using fixed number of Random Points in a
For loop');
grid on; hold off;
% I am not quite sure the meaning of "create a plot comparing the precision to
the computational cost"
figure(2);
plot(time, precision);
xlabel('Computational Cost; Time (0.001 Second)'); ylabel('Precision
(percentage)')
title('Fig.2: Precision vs. Computatinoal Cost')
grid on;
```



In Fig.1, I plotted the computed value of π and its deviation from the true value as the number of points increases. From the figure, I could confirm that as the number of random points increases, the computed value of π converges to the true value of π. Also, sometimes, the randomly generated dots can make the computed value of π bigger or smaller than the true value of π. To see the absolute distance from the true value clearly, I displayed an absolute deviation on the figure. As a result of the simulation, we can say that this algorithm actually well approximates the true value of π.



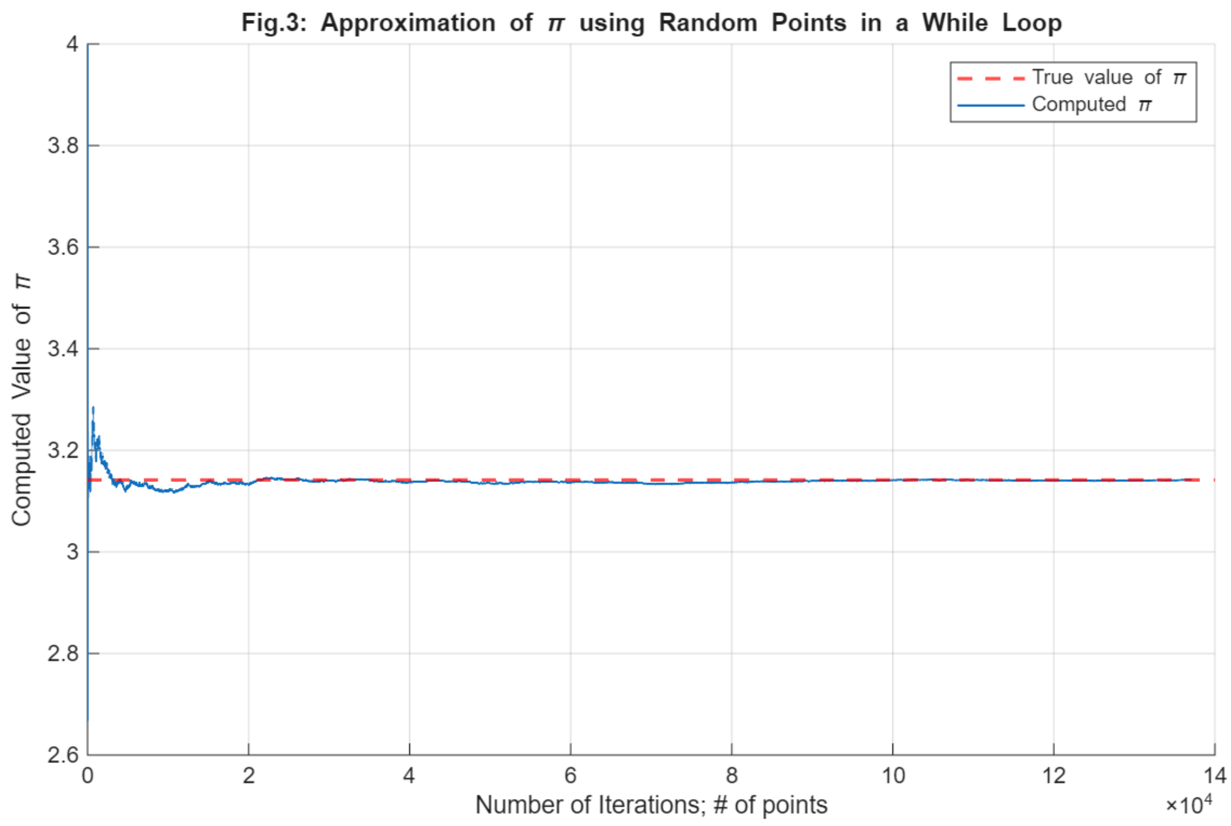
In Fig.2, I measured the execution time of the code for different point counts and created a plot comparing the precision to the computational cost. I used the "tic-toc" function in MATLAB to get the simulation(execution) time of the code in each step of the for loop, and compiled it to a 'time' vector variable. Also, I used the mathematical equation below using absolute deviation to get a percentage of precision in each step. Since the execution time means the computational cost, I plotted a line showing the precision percentage according to the increasing computational cost (time).

$$Precentage\ Precision = \left(1 - \frac{|Computed\ Value - True\ Value|}{True\ Value}\right) \times 100$$

- Task 2

For task 2, I used a "while" loop to compute π to a specified level of precision. To set the precision level without using the true value of π, I applied the "law of large numbers" to my program. According to the law, if a large number of random points are generated inside the unit square, the ratio of points inside the quarter of a circle and the square follows the ratio of their area. Thus, for the first criterion, I use an "if" function inside the while loop, and set the minimum number of points generated to 50 to reach the second condition that terminates the loop. For the second criterion, I use the "elseif" function to check how stable the computed value is. After a sufficient number of points have been generated, even if some additional points are biased, it will not significantly affect the calculated value. So I collect the 50 recent steps and check the gap between the maximum and minimum of the computed value. If the gap is below the degree that I set, 0.00005, then the program assumes that the computed value accomplished the specified level of precision and terminates the loop.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% Task 2: Using "while" loop
%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% parameter
v=[]; % vector to contain random dots
count=0; % counter
N=0; % counter to record the number of iteration; the number of created random
dots
bool=true; % boolean to run the while loop
com_pi=[]; % vector of computed value of pi
% main equation
while bool==true
   N=N+1;
   dot=rand(2,1); % generate one-coupled random dot
   if sqrt(dot(1)^2+dot(2)^2)<=1
       count=count+1;
   end
   com_pi=[com_pi; count/N*4]; % add a computed pi to the vector
   if N<51 % first condition of precision to use the law of large numbers
   elseif max(com_pi(N-50:N))-min(com_pi(N-50:N))<=0.00005 % second condition
of precision
       bool=false;
   end
end
fprintf("%d iterations are completed",N) % we can check how many iterations
are required
% plot
figure(3); clf; hold on;
yline(pi,'r--', 'LineWidth', 1.5); plot(com_pi, 'LineWidth', 1); hold off;
xlabel('Number of Iterations; # of points'); ylabel('Computed Value of \pi');
legend('True value of \pi','Computed \pi');
title('Fig.3: Approximation of \pi using Random Points in a While Loop');
grid on;
```



In Fig.3, we can see that as the number of iterations increases, the computed value of π converges to the true value of π. From this simulation, I got the result that 137090 iterations are needed to pass the final level of precision. And of course, to pass the first condition, 50 iterations are needed. From the result of the simulation, we can say that this algorithm actually well approximates the true value of π.

- Task 3

  For task 3, I modified the code for task 2 and made a function with 4 features.

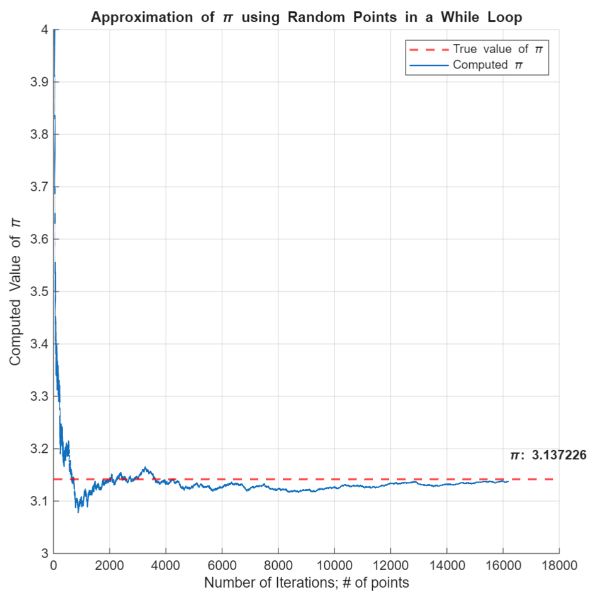
1. This function takes a user-defined level of precision as input. I set this level of precision by using the gap between the maximum and minimum values of the last 50 computed values of π. If this gap is less than or equal to the user-defined precision (e.g., 0.00005), the program assumes the computed value of π has become stable and has achieved the specified level of precision, at which point the loop terminates. Notice that you need to input the level of precision, not for the precision percentage but for the absolute difference between computed values. So the outcome would be accurate as you put the smaller number because it forces the computed value of π to converge to a tighter range before the simulation stops.

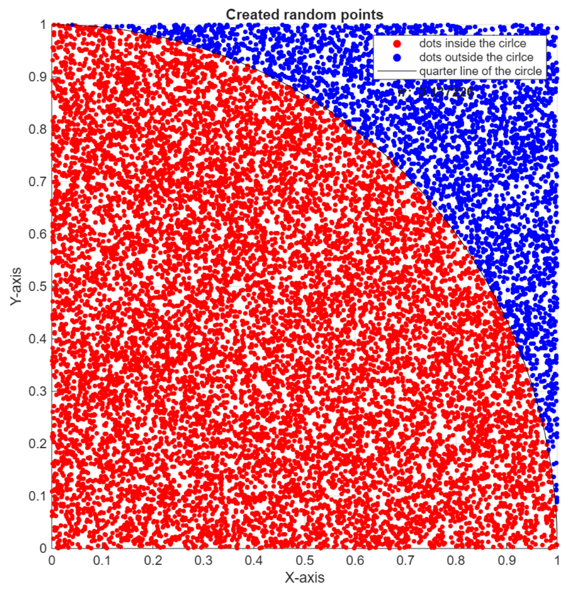
2. Running this function allows us to see all generated random points on the x-y plane, and to make the difference between the points inside and the outside of the circle, I differentiate the color of the points.

3. Also, when we use the function with user-specified precision, the final computed value of π is both displayed in the command window and printed on the plot.

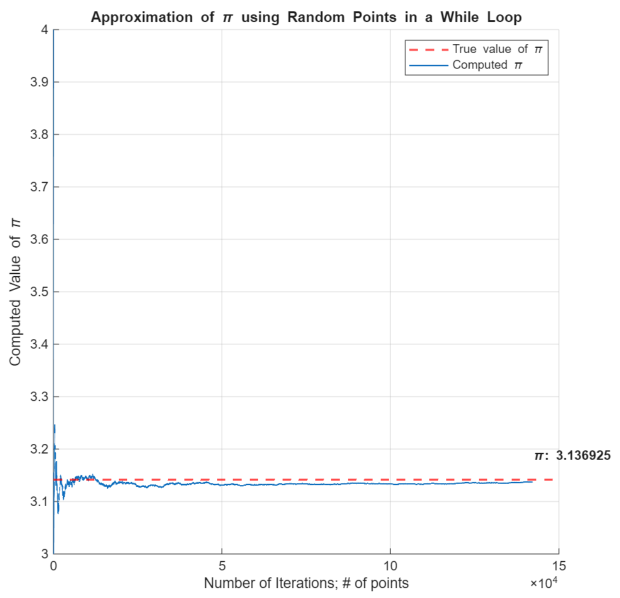
4. The computed value of π is returned. In the code, the computed value is restored in the "Result" variable.

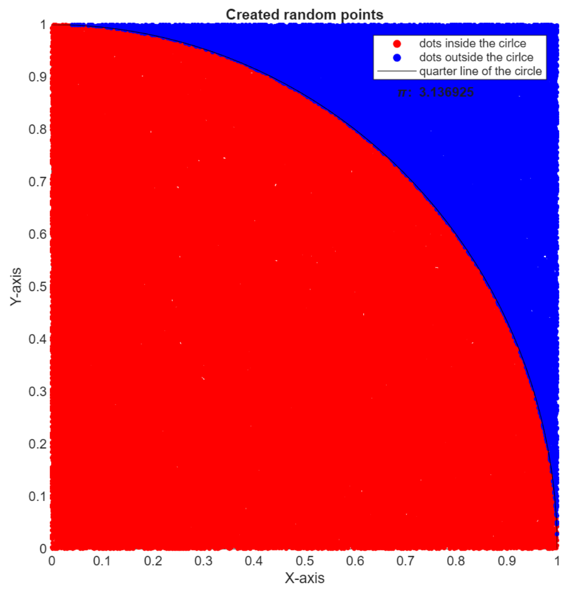
```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% Task 3
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function final_pi=compute_pi()

   % parameter
   v=[]; % vector to contain random dots
   count=0; % counter
   N=0; % counter to recodr the number of iteration; the number of created
random dots
   bool=true; % boolean to run the while loop
   com_pi=[]; % vector of computed value of pi
   dot_v1=[]; % vector of randomly created dots inside the circle
   dot_v2=[]; % vector of randomly created dots outside the circle
   precision=input('Enter the level of precision: '); % need to revise
   r=1; % radius
   theta=linspace(0,pi/2,50); % we are going to draw a quarter of the circle;
we only need from 0 to pi/2 radian


   % main equation
   while bool==true
       N=N+1;
       dot=rand(2,1); % generate one-coupled random dot
       if sqrt(dot(1)^2+dot(2)^2)<=1
           count=count+1; dot_v1=[dot_v1,dot]; % add a randomly created dot to
the vector; maintaining the size of matrix as 2 by N
       else; dot_v2=[dot_v2,dot];
       end
       com_pi=[com_pi; count/N*4]; % add a computed pi to the vector
       if N<51
       elseif max(com_pi(N-50:N))-min(com_pi(N-50:N))<=precision
           bool=false;
       end
   end
   fprintf('The computed value of π is: %f\n', com_pi(end));
   txt=sprintf('\\pi: %f',com_pi(end));
   final_pi=com_pi(end); % assign the final computed value to the output

   % plot
   figure(4); clf; subplot(1,2,1); hold on;
   yline(pi,'r--', 'LineWidth', 1.5); plot(com_pi, 'LineWidth', 1);
   text(length(com_pi),com_pi(end)+0.05,txt,'FontWeight', 'bold'); hold off;
   xlabel('Number of Iterations; # of points'); ylabel('Computed Value of 
\pi');
   legend('True value of \pi','Computed \pi');
   title('Approximation of \pi using Random Points in a While Loop');
   grid on;

   subplot(1,2,2); hold on;
   scatter(dot_v1(1,:),dot_v1(2,:),12,'filled','r');
   scatter(dot_v2(1,:),dot_v2(2,:),12,'filled','b');
   plot(r*cos(theta),r*sin(theta),'k');
   text(0.68,0.87,txt,'FontWeight', 'bold');
   hold off;
   xlabel('X-axis');ylabel('Y-axis');
   legend('dots inside the cirlce','dots outside the cirlce','quarter line of 
the circle')
   title('Created random points');
   grid on;
end
Result=compute_pi();
```





The above figure is the result of setting the low precision level, and the below figure is the result of the high precision level. We can see the difference in the number of generated dots on the figures. We can see the final computed value of π is displayed on the plot. Although it is hidden by the generated random dots, not only on the left side figure, but also on the right figure, the computed value is displayed(below the legend). Also, from the result of the simulation, we can say that this algorithm actually well approximates the true value of π.


Approximation of $\pi$ using Random Points in a While Loop
True value of $\pi$
Computed $\pi$
$\pi$: 3.136925
Computed Value of $\pi$
Number of Iterations; # of points
$\times 10^4$


Created random points
dots inside the cirlce
dots outside the cirlce
quarter line of the circle
$\pi$: 3.136925
Y-axis
X-axis