November 2024

Introduction to C++: Workshop Six

Dr. Alexander Hill

a.d.hill@liverpool.ac.uk

LIV.INNO

Last Week

- Monte Carlo Basics
- Generating random numbers in C++

Aim of Workshop Six

- Homework recap
- Markov Chains
- Group Project

Resources

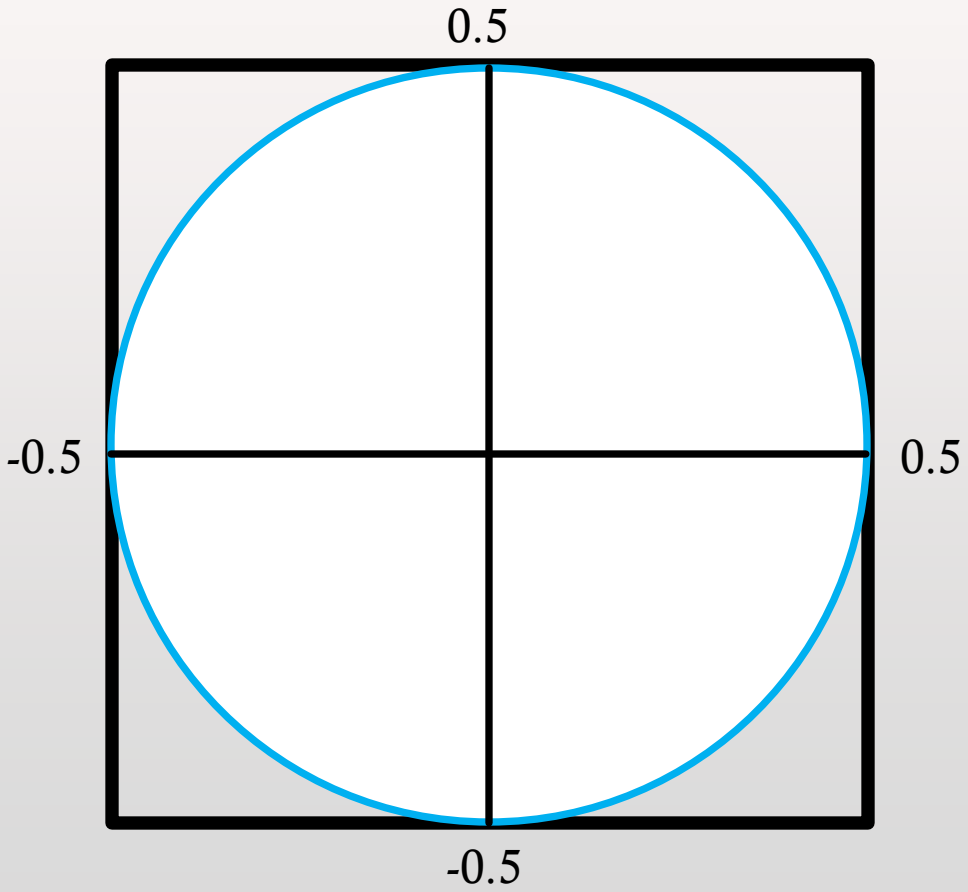
- alex-hill94.github.io/#WS6 for lecture slides
- alex-hill94.github.io/#Proj for project description and a potentially useful C++ script
- Previous lecture slides for details on random numbers, classes, functions, etc.
- https://cplusplus.com/reference/random

Challenge Ten (Homework)

- For the circle described by $x^2 + y^2 = 0.5^2$, compute its area using a Monte Carlo method

- i.e. draw random x and y between -0.5 and 0.5, and compute the fraction of draws that satisfy $x^2 + y^2 < 0.5^2$

- How many draws do you need to get ~1% error on πr^2 ?



```cpp
double getRadius(double x, double y) {
return sqrt(x * x + y * y);
}

int main(void) {
// Specify number of steps
int steps;
cout << "Number of steps: ";
cin >> steps;

// Define variables
double x;
double y;
double r;
int count = 0;
double area;
double error;

const double start = -0.5;
const double end = 0.5;
const double radius = (end - start) / 2;
// Random number generator
random_device rand_dev;
mt19937 generator(rand_dev());
uniform_real_distribution<double> distribution(start, end);
```

Nicely set up RNG

```cpp
// Monte Carlo simulation
for (int i = 0; i < steps; ++i) {
x = distribution(generator);
y = distribution(generator);
r = getRadius(x, y);
if (r < radius) {
count++;
}
}

// Calculate estimated area and error
area = double(count) / steps;
error = 100*(1 - area/M_PI_4);

// Print results
cout << "Estimated area: " << area << endl;
cout << "Target area: " << M_PI_4 << endl;
cout << "Error: " << error << "%" << endl;
return 0;
}
```

Area of square = 1, so fraction of hits does equal area estimate

M_PI_$ = PI * 0.5**2

```
$ ./a
Number of steps: 10
Estimated area: 0.9
Target area: 0.785398
Error: -14.5916%
$ ./a
Number of steps: 1000
Estimated area: 0.798
Target area: 0.785398
Error: -1.60452%
$ ./a
Number of steps: 10000000
Estimated area: 0.785342
Target area: 0.785398
Error: 0.00718914%
```

Naomi

```
double randomiser(mt19937& generator, const double& range_from,
const double& range_to) {
uniform_real_distribution<double> distr(range_from, range_to);
return distr(generator);
}
int main() {
double min = -5;
double max = 5;
double area = 0;
double true_area = 0.25 * M_PI;
double n = 1000;
vector<double> x;
vector<double> y;
random_device rand_dev;
mt19937 generator(rand_dev());

// Seed the generator to see how things change! for some reason it's the
same for all values unless you stick 1 in the generator?
int seed;
cout << "Enter a seed value: ";
cin >> seed;
generator.seed(seed);
```

```
while (abs((area / n) - true_area) / true_area > 0.01) {
int new_points = n * 2;
x.reserve(x.size() + new_points);
y.reserve(y.size() + new_points);
for (int i = 0; i < new_points; ++i) {
double new_x = randomiser(generator, min, max);
double new_y = randomiser(generator, min, max);
x.push_back(new_x);
y.push_back(new_y);
if (pow(new_x, 2) + pow(new_y, 2) <= 25) {
area += 1;
}
}
n += new_points;
}

write_out("data.py", "x", x, true).writ
write_out("data.py", "y", y, false).write();
cout << "The area of the circle is: " << (area / n)<< " units\u00B2" << endl;
cout << "The true area of the circle is: " << true_area << " units\u00B2" << endl;
cout << "number of iterations: " << n << endl;
return 0;
}
```

Something going wrong with the while statement and n+=new_points I think

```
$ ./a
Enter a seed value: 404
The area of the circle is: 0.782115 units²
The true area of the circle is: 0.785398 units²
number of iterations: 243000
$ ./a
Enter a seed value: 8
The area of the circle is: 0.781938 units²
The true area of the circle is: 0.785398 units²
number of iterations: 243000
```

```cpp
int main(){
int n_sim_runs = 10;
const double radius = 0.5;
double sim_frac;
double error = 100.0;
double error_thresh = 1.0;

circle my_circle(radius);
while (error > error_thresh) {
vector<double> x_inputs(n_sim_runs);
vector<double> y_inputs(n_sim_runs);
vector<double> outputs(n_sim_runs);
for (auto i = 0; i < x_inputs.size(); ++i){
random_num(x_inputs.at(i), radius);
random_num(y_inputs.at(i), radius);
}

my_circle.calculate_area(x_inputs, y_inputs, outputs);
sim_frac = my_circle.calculate_fraction(outputs, radius);
error = my_circle.calculate_error(sim_frac);
n_sim_runs = n_sim_runs*2;
}


cout << "Final number of runs = " << n_sim_runs << endl;
cout << "Fraction = " << sim_frac << endl;
cout << "Final error = " << error << "%" <<endl;
cout << "Actual area: " << my_circle.actual_area << endl;
return 0;
}
```

```
$ ./a
Final number of runs = 2560
Fraction = 0.78125
Final error = 0.528161%
Actual area: 0.785398
```

Inefficient to bin the 10, 20, 40,… computations before convergence

```
class circle {
public:
double radius;
double actual_area;
circle(double r){
radius = r;
actual_area = M_PI*pow(r,2);}

void calculate_area(const vector<double>& x_inputs, const vector<double>& y_inputs, vector<double>& outputs){
for (auto i = 0; i < x_inputs.size(); ++i){
outputs.at(i) = pow(x_inputs.at(i), 2) + pow(y_inputs.at(i), 2);}}


double calculate_fraction(const vector<double>& areas, const double radius){
int count = 0;
double fraction;

for (double a : areas) {
if (a < pow(radius,2)) {
count++;}}
fraction = static_cast<double>(count) / areas.size();
return fraction;}


double calculate_error(double sim_frac){
double error = (abs(actual_area - sim_frac) / actual_area) * 100.0;
return error;}};
```

Not actually calculating the area, more a point's position on a plot

This only works with r = 0.5, as in this case sim_frac == estimated area

Matthew

```cpp
int main() {
float square_area = 4 * radius * radius;

// Random number generator
random_device rand_dev;
mt19937 generator(rand_dev());
uniform_real_distribution<double> distr(-radius, radius);

vector<double> x_init(num_elements), y_init(num_elements), x_final, y_final;

// Fill vectors with random values
for (int i = 0; i < num_elements; ++i) {
x_init[i] = distr(generator);
y_init[i] = distr(generator);
}

// Discard points outside the circle
for (int i = 0; i < num_elements; ++i) {
if (x_init[i] * x_init[i] + y_init[i] * y_init[i] <= radius * radius) {
x_final.push_back(x_init[i]);
y_final.push_back(y_init[i]);
}
}

// Calculate the areas
double estimated_area = circle_area_estimation(x_final, y_final, square_area);
double actual_area = true_circle_area(radius);

cout << "Estimated Area = " << estimated_area << endl;
cout << "True Area = " << actual_area << endl;
cout << "Error = " << calculate_error(estimated_area, actual_area) << "%" << endl;

return 0;
}
```

```cpp
const int num_elements = 1500;


// Function to calculate the estimated area based on the Monte Carlo simulation
double circle_area_estimation(const vector<double>& x_inside, const vector<double>&
y_inside, float square_area) {
double ratio = static_cast<double>(x_inside.size()) / num_elements;
return ratio * square_area;
}
```

Multiplying fraction by square_area makes this work

Would have even been better to show decreasing error with increasing draws

```
$ ./a
Estimated Area = 0.793333
True Area = 0.785398
Error = 1.01034%
```

```
double random_number(const double min, const double max) {
//srand(time(0));
double rn = ((double)rand()) / RAND_MAX;
double rnir = min + ((max - min)*rn);
return rnir;}

int circle(const double radius) {
double x = random_number((-0.5), 0.5);
double y = random_number((-1*radius), radius);
double eq = pow(x,2) + pow(y,2);
//cout <<x <<y;
if(eq < pow((radius), 2)) {
return 1;}
else {return 0;}}

int main() {
double r = 0.5;
double actual_area = pow(r, 2)*M_PI;
double square_area = pow((2*r),2);
double sample_size = 3300;
double points_in_circle = 0;
for(double i = 0; i < sample_size-1; ++i) {
if(circle(r) == 1){
++points_in_circle;}}

double in_vs_out = points_in_circle/sample_size;
double MC_area = in_vs_out*square_area;
double error = abs(((actual_area - MC_area)/actual_area)*100);
cout <<"actual area: " << actual_area << "\n"<< "MC area: " << MC_area << "\n"<< "error: " <<
error <<"%";
return 0; }
```

Potential for error up here

Nice and concise

Rosie

```
$ ./a
actual area: 0.785398
MC area: 0.785152
error: 0.0314042%
```

```cpp
int main(){
int n_sim_runs =1e4;
double range_from=-0.5;
double range_to = 0.5;
double radius = 0.5;
double total_area = 1.0;
vector<double> x_val(n_sim_runs);
vector<double> y_val(n_sim_runs);
for (auto i = 0; i < x_val.size(); ++i){
rnum(x_val.at(i), range_from, range_to);
rnum(y_val.at(i), range_from, range_to);
}
int inside_circle = draw(x_val, y_val);
double area= total_area*inside_circle/n_sim_runs;
double diff=(2*100*abs(M_PI*pow(radius, 2)-area))/(total_area+area);
if (diff < 1) {
cout << "Total draws = " << n_sim_runs << "\n";
cout <<"Passed draws = " << inside_circle << "\n";
cout << "Area of Circle = " << area << "\n";
cout << "Percentage Diff = " << diff << "%";}
else{
cout << "Try again or increase number of iterations " << "\n" ;
}
return 0;
}
```

```
Total draws = 10000
Passed draws = 7789
Area of Circle = 0.7789
Percentage Diff = 0.730582%
```

Shoaib

```cpp
// function to generate random number
void rnum(double& input, const double ini_val, const double fin_val){

random_device rand_dev;
mt19937 gen(rand_dev());
uniform_real_distribution<double> distr(ini_val, fin_val);
input = distr(gen);
}
// function to calculate draws that satisfy give condition
int draw(const vector<double>& input1, const vector<double>& input2){
int count=0;
for (int i=0; i< input1.size(); ++i){
if (pow(input1.at(i),2)+pow(input2.at(i),2) < pow(0.5,2)) {
count=count+1;
}}
return count;
}
```

Consider passing radius as an argument

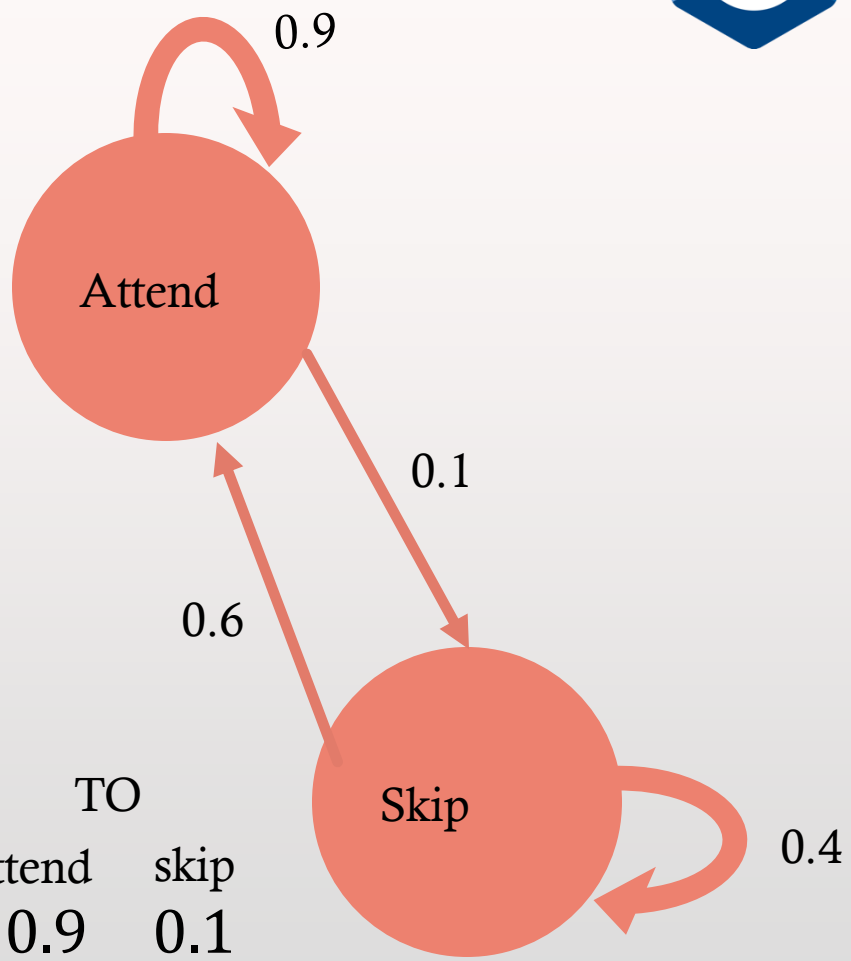
Markov Chain Monte Carlo (MCMC)

- Monte Carlo: estimate the expected value or probability density of some unknown space by drawing independent random values
- For high-dimension probabilistic models, Monte Carlo sampling may not be effective, as volume of sample space grows exponentially with additional parameters
- MCMCs try to sample more intelligently, the next random draw depends on the current one

Andrey Markov

Markov Chains

- A simple Markov Chain uses stochastic processes to determine the evolving state of a system
- Consider this system, it describes whether someone attends class given their previous attendance
- E.g. if you attend class one week, there's a 90% chance you will the next

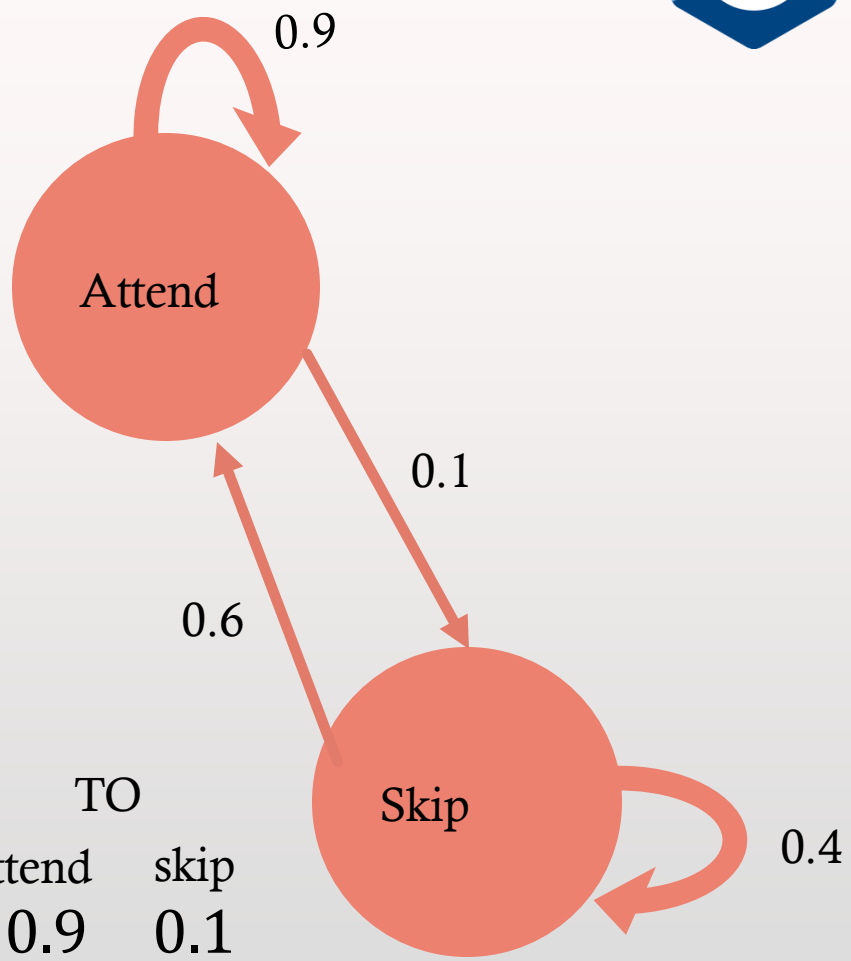


		TO attend	skip
FROM	attend	0.9	0.1
	skip	0.6	0.4

Transition matrix

Markov Chains

- Start with initial state of attendance: $X_0 = [1,0]$
- $X_1 = X_0 T = [0.9, 0.1]$
- $X_2 = X_1 T = [0.87, 0.13]$
- In the long-run, you approach a **steady state**, i.e. $X_{n+1} = X_n$

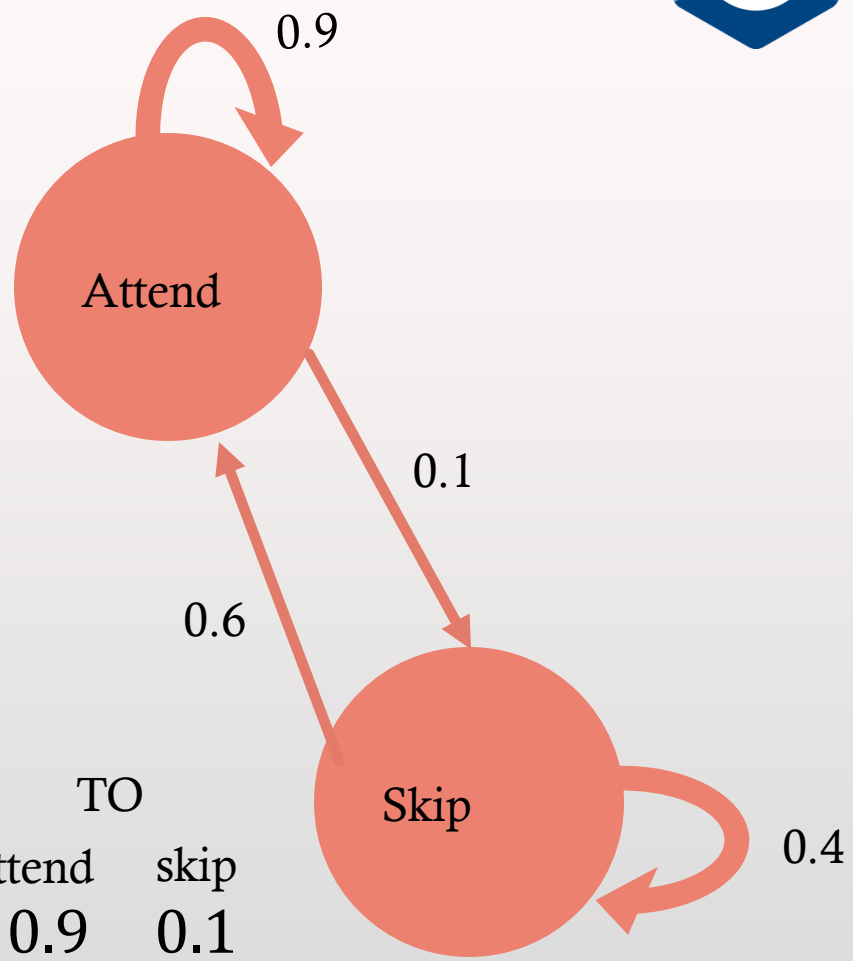


		TO attend	skip
FROM	attend	0.9	0.1
	skip	0.6	0.4

Transition matrix

Markov Chains

- $X_s - X_s T = 0$
- $X_s(I - T) = 0$
- $[x, y]\begin{pmatrix} 0.1 & -0.1 \\ -0.6 & 0.6 \end{pmatrix} = 0$
- $0.1x - 0.6y = 0$ and $x + y = 1$
- $X_s = \left[\frac{6}{7}, \frac{1}{7}\right]$



		TO	
		attend	skip
FROM	attend	0.9	0.1
	skip	0.6	0.4

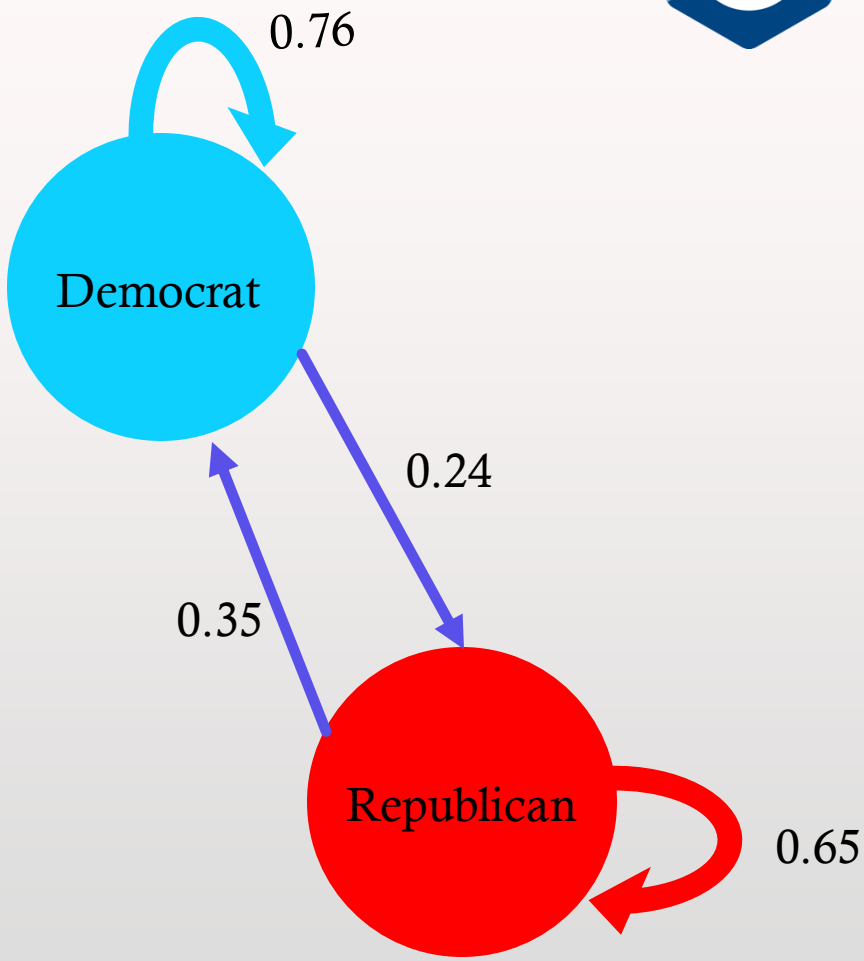
Transition matrix

Markov Chains

- Markov Chains can also be used to generate a sequence of random variables where the current value is dependent on the value of the prior value
- An example of this is a number line, where possible moves are -1 and 1 (chosen with equal probability)
- MCMCs are Monte Carlo methods where a Markov chain is used to draw samples
- The idea is that the chain will settle (find equilibrium) on the desired quantity we are inferring

Challenge Eleven

- Create a class that generates random numbers from a uniform distribution
- Create a Markov Chain class that predicts which US party will win the next election (lookup matrices, matrix mutlipliaction, if statements etc.)
- Assume initially a Dem is in power X0 = [1,0], create a method in the MC class that calculates numerically the steady state vector, i.e. the probability that in a given year
- Create another method that uses random draws from the random number class to stochastically predict who will be in power for each of the next 20 cycles
- Create a figure showing how the holder of office changed over the 20 cycles

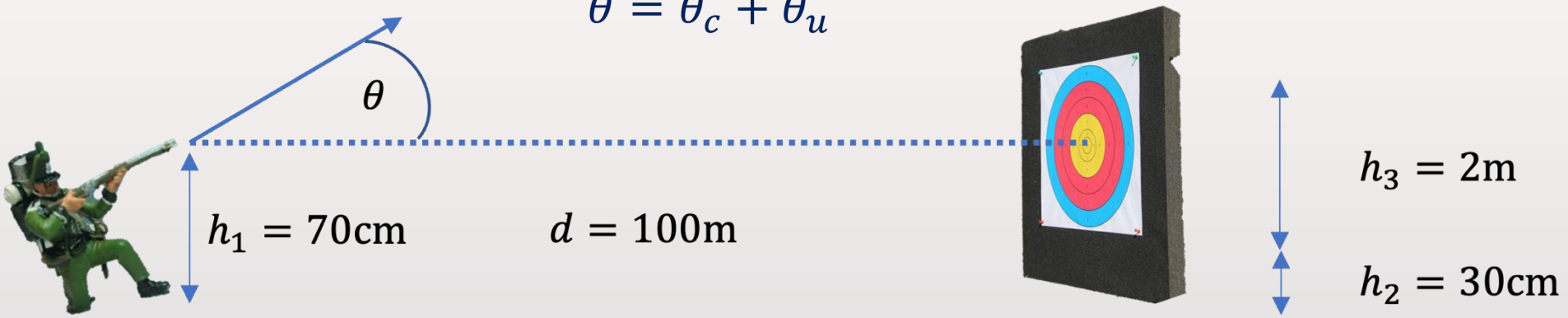


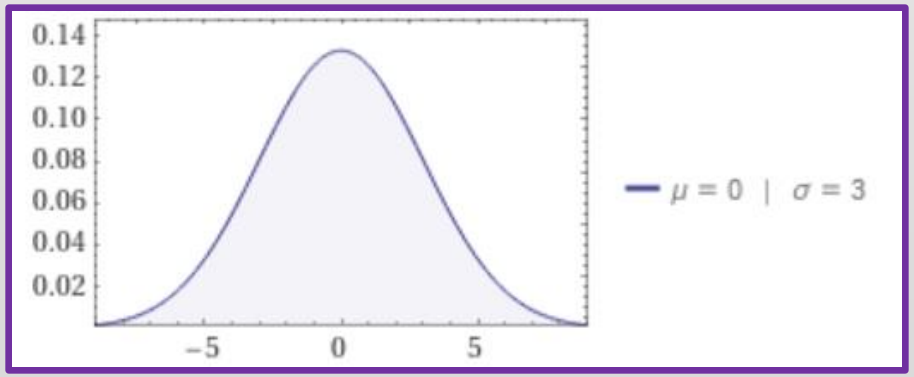
Group Project

- Context: ballistics in the 1800s
- Aim: determine the accuracy of various weapons at a given distance

Project Description

$$\theta = \theta_c + \theta_u$$



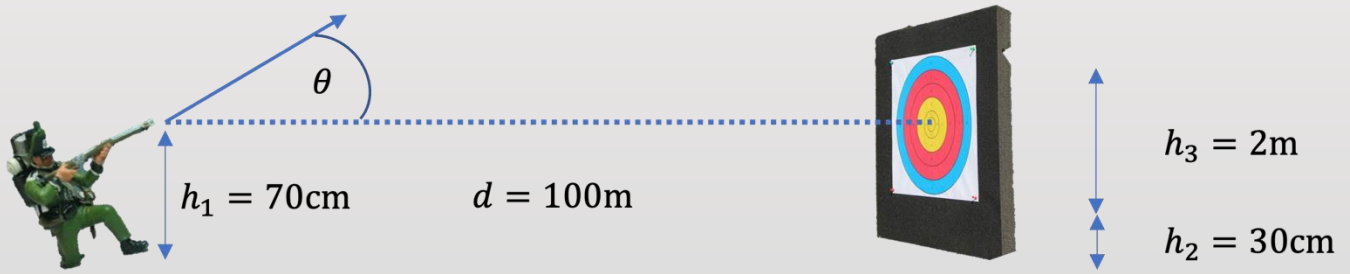


θ_u is sampled from a normal distribution

Project Description

- **Step one**: find the optimal angle required to hit the bullseye.
- **Step two**: set this to be θ_c. Sample θ_u from a normal distribution and add to θ_c to find θ.
- **Step three**: the soldier fires three times per minute. Simulate one 'trial' as being five minutes of firing. How many times does he hit the target?
- **Step four**: run 1000, 10,000, 100,000 trials. What is the distribution of the number of hits?

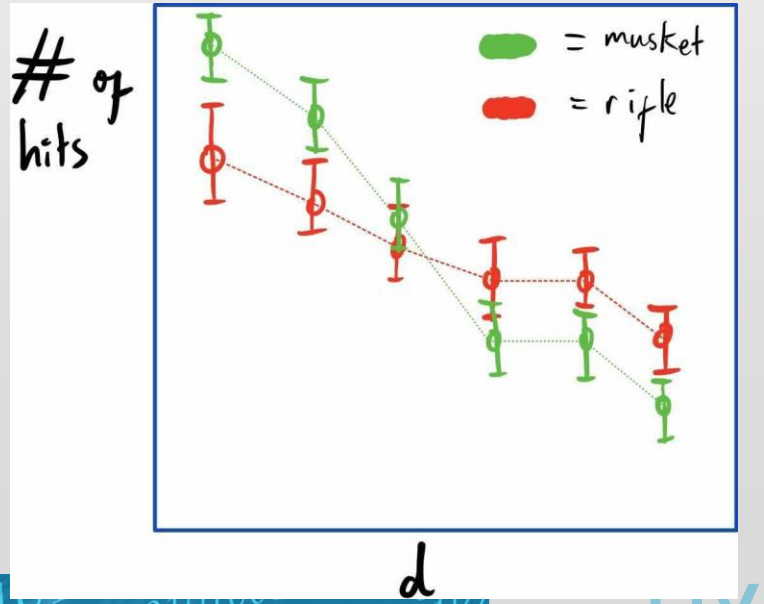
$$\theta = \theta_c + \theta_u$$



Project Description

- Step five: compare the performance of a rifle vs a musket at 100m (details on the main document)

- Step six: run the experiment for muskets and rifles at various distances. At what distance does it become better to use one over the other?



Project Description

- Live demonstration on my laptop, will it compile and run first time?
  - Produce data stream, save data, produce plots…

Tips

- DRY – don't repeat yourself!
- Generalise things as much as possible, and consider where it would be useful to use classes
- Plan your approach before you start coding
- Communicate via Slack, or book study rooms in the teaching hub (502)
- Try using GitHub if the project becomes complex

Your Project

- Group One
  - John
  - Naomi
  - Rosie
  - Shoaib
- Group Two
  - Ben
  - Jak
  - Shirsendu
  - Matthew
- Group Three
  - Pawan
  - Stephen
  - Salma
  - Archie
  - Liam

Room:


READY STEADY COOK