Nature of Code

Patrick Dwyer
Fall 2005
Week 3 - September 20th

Probability and Outcomes

Probability is the measurement of the likelihood of a desired outcome out of all possible outcomes. If our desired outcome is flipping a coin and having it come out heads, and we flip the coin a single time, our probability of a successful outcome is 1 (heads) out of 2 (heads or tails), or 50%.

Single Event Probability

Flipping a coin once is a single event probability; we have a single desired outcome (heads) from a single instance of an event (a single coin flip). Determining the probability of an outcome from a single event is as simple as finding the number of different possible outcomes that can be considered successful, and dividing that number by the total quantity of possible outcomes.

Consider a deck of cards; What is the probability of choosing an Ace from the deck? With 52 cards and 4 aces, the probability is:

> *4 possible successful outcomes / 52 total possible outcomes = 4 / 52 or approximately 7.6%*

Likewise, the probability of drawing a Club is:

> *13 possible successful outcomes / 52 total possible outcomes = 13 / 52 or 25%*

Multiple Event Probability

When evaluating more than a single event, the probability of a desired outcome is the product of all of the individual event outcomes. For instance; the likelihood of drawing two aces in a row (two separate events) is:

> *First Ace: 4 / 52 = 7.6%*
>
> *Second Ace: 3 / 51 = 5.8%*
>
> *Overall Likelihood: 7.6% * 5.8% = 0.4%*

Note that the likelihood of drawing a second Ace is different than drawing the first, as the first event changed the circumstances of the second event, resulting in a lower probability for the second event.

The probability of drawing a 5 card flush (all cards the same suit) is:

> *(52 / 52) * (12 / 51) * (11 / 50) * (10 / 49) * (9 / 48) = 0.2%*

And the likelihood of drawing a straight?

> *(36 / 52) * (4 / 51) * (4 / 50) * (4 / 49) * (4 / 48) = 0.0042%*

Random Numbers

When simulating natural systems it is quite often necessary to introduce some amount of random "noise" into the system. Using non-random algorithms to model events creates highly predictable simulations that don't "feel" right; they lack the disconnection between human control and system response that typifies nature. We have no direct control over how wind effects leaves blowing across the ground, and we expect them to behave in an unpredictable manner.

By definition it is very hard for computers to create truly random numbers; in fact most random number generators for computers are called pseudo-random number generators. Depending on the method of generating the numbers we can expect different results. While Processing has built in functionality for generating random numbers, the number it creates fall into a "Uniform" distribution.

Uniform Distribution

Generally when we work with random numbers we want to get a random value between a minimum and a maximum; in Processing we can use the `random()` method:

```
float myVal = random(0, 100);
```

This will return a random value between 0 and 100 as a floating point number. Uniformly distributed random numbers have an equal chance of being generated; over a sample of random values each possible number has a near equal probability of occurring (See Examples 1 or 2 for the week).

Gaussian Distribution

There are cases where we don't want a uniform distribution of numbers; we might need some values to occur more frequently than others. Gaussian and Custom distributions give us a set of tools for creating non-uniform distributions of numbers.

For instance, if we needed the number 1 to occur 20% of the time, 2 to occur 40% of the time, and the number 3 to occur the other 40% of the time, we could create a simple custom distribution like the following:

```
int[] custRand = new int[5];
custRand[0] = 1;
custRand[1] = 2;
custRand[2] = 2;
custRand[3] = 3;
custRand[4] = 3;
int randIndex = int(random(custRand.length));
println(custRand[randIndex]);
```

This example would create a small generator that would have the probabilities listed above for generating 1, 2 or 3.

Sometimes the values we want to generate fall into a pattern called a Gaussian distribution, or a Bell-Curve. In a Gaussian distribution the "Mean" or average value has the highest probability of occurring, with all values less than or greater than the average occurring in relation to their distance from the average. We can quickly simulate a Gaussian distribution using the CustomRandom class[1]:

[1] First used as a class in the 4th example for the week, in the previous examples it is included in the Processing code.

```
CustomRandom rand = new CustomRandom();
for (int i = 0; i < 100; i++) {
      float randNum = rand.getRandom(100.0);
      println(randNum);
}
```

This code snippit will generate a series of 100 random numbers between the values of 0.0 and 100.0, the mean value of 50.0 will occur most often, with the remainder of the number spreading out on a bell-curve.

Custom Distribution

The distribution of numbers that we need may not fit a Uniform or Gaussian distribution; in these cases we can generate a custom distribution of numbers. Any method that favors a range of number over another can be considered a custom generator. Consider the following:

```
float myGenerator() {
      float i = random(100.0);
      if (i > 60.0) {
            return 2;
      } else {
            return 1;
      }
}
```

Here we've created a simple number generator that has a 60% chance of returning 2 and a 40% chance of returning 1. To aid in more complex distributions we can use the CustomRandom class again, this time supplying our own distribution model[2]:

```
CustomRandom rand = new CustomRandom();
float[] distrib = {0.4, 0.01, 0.01, 0.4, 0.4, 0.01, 0.01, 0.4, 0.4,
0.01};
rand.useDistribution(distrib);
for (int i = 0; i < 100; i++) {
      float randNum = rand.getRandom(100.0);
      pritnln(randNum);
}
```

This code defines a custom number distribution pattern in the array of floats `distrib` and then sets the custom distribution used by our CustomRandom object using the `useDistribution` method. The array of floating point number we create defines a pattern that will be used to choose random numbers. It works by first choosing a random number in the range we choose when the `getRandom` method is called. This random value is then scaled to fit the length of our array of floating point numbers. If the random number lies under the line connecting all of the floating point numbers together, we return this random number to the user, if not, we generate a new random number and test it against the array of floating point numbers. This method is commonly called a Monte-Carlo system. This system will generate fairly random number conforming to our distribution pattern. On the down side this method is somewhat slow, and has a small chance of needing to generate a long stream of numbers before finding one that fits our pattern[3].

---

[2] The CustomRandom class initializes itself with a model for generating a Gaussian Distribution by default.

[3] We could calculate the probability of this method needing more than 1 try to generate a number using the probability math we discussed at the beginning, and figuring out how likely it is that our number will be under the line of our distribution. If we guess that our Gaussian distribution has a 30% chance of fitting under our line, then there is a 0.2% chance it will take 5 tries to find a suitable random number

Perlin Noise

Our approach to random number to this point hasn't taken into account the relationship between one random number and the next. With all of the methods above, each random number is disconnected from the previous one; there is no distinct between the numbers in a random series[4]. Sometimes, though, we desire random numbers that have some relationship over the series; we want numbers that transition smoothly between one and the next. In the 1980s Ken Perlin invented an algorithm for generating N-dimensional smooth noise (called Perlin Noise). This means that we can created an ordered, natural series of random values with as many dimensions as factors as we want. Processing is capable of generating 1, 2 or 3 dimensional Perlin Noise for us:

```
for (int i = 0; i < 5; i++) {
      float nOff = 0.0;
      float noiseValue = noise(nOff);
      println(noiseValue);
}
```

If you run this code you'll notice that the value of our `noiseValue` variable is the same each time; this is because we are calculating it for the same place in 1-dimensional space each time. Consider instead:

```
float nOff = 0.0;
for (int i = 0; i < 5; i++) {
      nOff += 0.01;
      float noiseValue = noise(nOff);
      println(noiseValue);
}
```

Because we slowly increment the value passed to the `noise` method, our resulting `noiseValue` changes at each pass. How we alter the value passed to the `noise` method affects the "feel" of the value returned. Creating noise in 2 and 3 dimensions is similar:

```
float twoNoise = noise(0.01, 0.4);
float threeNoise = noise(10.0, 0.04, 2.1);
```

Again; what we use for the parameters of the `noise` method, and how we change those values, affects the outcome.

Assignment

- Create a visual system that is guided completely by random numbers (random shape, location, size, color). Now, recreate the system you've designed without using any of the methods we've discussed for creating randomness; try and find new ways to simulate controlled randomness.

- Modify a previous assignment to include randomness or Perlin noise.

---

[4] Or so we hope; random number generation hinges on the idea that each number is distinct and cannot be used to determine anything about the next step in the sequence. The more disconnected a random number is from the conditions surrounding it (initial conditions, previous random values, state of computer) the stronger the generation method is considered to be.