SINES & SQUARES 2016 A Celebration of Analogue and Modular Synthesis 18th-20th November, Manchester UK

Arduino random generators for Eurorack Synths

Michael Spicer

Singapore Polytechnic, Singapore, Singapore
Mixspicer@gmail.com

Abstract.

Random techniques have been used in modular synthesizers from their earliest incarnations. Computer music composers have a developed a wide repertoire of other pseudo-random techniques over the years. Many of these approaches can easily be implemented on the user programmable microcontroller based modules that are now available such as those that are designed around the Arduino architecture. This paper demonstrates how to implement a variety of classic random processes in the C++ programming language, and shows how they can be integrated into the control voltage environment of a eurorack modular. Three example Algorithms are discussed include the Gaussian probability distribution, the Uniform probability distribution, the Triangle probability distribution and the Logistic Function. Object-oriented programming concepts, implemented in C++, will help to simplify issues such as memory management, code readability and code maintenance.

Keywords: Arduino, random generator modules, object-oriented programming.

Introduction:

Random techniques have been used in modular synthesizers from their earliest incarnations. The technology used to implement random behaviour is frequently based on either some combination of noise generators, filters and sample and hold modules, or on shift registers. These have provided a variety of useful random methods but there are many other types of random process that are potentially useful in modular synthesis systems. Since 1957, computer music composers have drawn on the work of mathematicians and have a developed a wide repertoire of pseudorandom techniques. These fall into two basic categories. In one category, the past has some influence on what happens in the future, and in the other category, it doesn't. Many of these approaches can easily be implemented on any of the user programmable microcontroller based modules that are currently available. While there are many different microcontrollers based modules, the cheapest, and easiest to program are designed around the Arduino architecture. Examples of this kind of module include the Circuit Abbey Euro-Duino and the Snazzy FX – ARDcore. This paper explains how to implement a variety of classic random processes in the C++ programming language, and shows how they can be integrated into the control voltage environment of a eurorack modular. The examples are designed to run on the Euro-Duino module but the code can be easily adapted to other modules. This collection of programs enables the EuroDuino module to output a random number every time a gate or clock signal is applied to the digital input of the module.

Program structure

Software that enables the control of hardware systems all have a simple two-part structure. One part initialises the system, and the other part is a control loop that runs infinitely (or until the power is turned off). There are numerous examples of code available online, demonstrating how to implement the functionality of common synthesizer modules, such as LFO, etc. Usually, one program will implement the functionality of a number of modules, which makes the software more complex. In addition, many of the software examples are written in a relatively cryptic programming style and are highly optimised in terms of speed and memory requirements. While this approach is very efficient from the point of view of runtime performance, I find that coding in this way can be error prone, so it can take some time for the programmer to get everything working correctly. I have adopted a different code structure for my work and use an object-oriented approach; similar to the way I have created computer games over the last twenty years. Most of my Arduino programs use the same core code and the differences between programs are isolated inside specific software containers, referred to as **objects,** which are created by specifying a particular software design, called a **class**. This conceptual isolation makes creating new software quite simple.

The basic program structure is superficially the same as traditional Arduino programs:

- *Initialise the system*
- *Run the infinite loop*

The infinite program loop is also the same as most Arduino programs:

- *Read the switches/control voltage input/ knobs/clock*
- *Respond to the switches by creating a new object, if required.*
- *Update the controls for the objects, based on the control voltage input/ knobs*
- *Check/calculate a new output if it is time to do so.*

The basic program loop

When translated into code, using an object-oriented approach the complete loop is much simpler than most Arduino programs:

```
void loop() {
  readControlInput();                // read the hardware
  handleSwitches();                  // create new random object if switch changes

  pDist1->setParm1(max(cvIn1Value, pot1Value)/1024.0);  // set parameter 1
  pDist1->setParm2(max( cvIn2Value, pot2Value)/1024.0); // set parameter 2
  digitalWrite(DigitalOut1Pin, 0);                      // turn off trigger output

  if (digitalIn1Value != currentDigitalIn1State){      // calculate a new output
    if (digitalIn1Value == 1){                         // clock turns on so…
        digitalWrite(DigitalOut1Pin, 1);               // turn on the trigger signal
        currentDigitalIn1State=1;                      // remember current state
        analogWrite(analogOut1Pin, 255- pDist1->gen()); // output new random value
    }
    else {                                             // clock turns off so…
      currentDigitalIn1State = 0;                      // remember current state
      digitalWrite(DigitalOut1Pin, 0);                 // turn off trigger output
    }
  }
  delay(10);                                           // wait for hardware to settle!
}
```

While this initially looks a bit cryptic for non-programmers, it is very short and easily understood. It is an ordered sequence of actions. The parts that do the actual work to create the random numbers are those that contain the text *pDist1->* . This is where the custom behaviour for each module is implemented. In the 4th and 5th line, the internal states of the software object are updated to reflect what has happened in the hardware, and the call to pDist1->*gen()* in line 11, is where the new random value is generated. (The text "*255 –*" is there due to a design quirk of the EuroDuino module and is probably not needed in other implementations). Currently, I have a collection of objects that create their values via various pseudorandom algorithms, but all of them use this code. In conventional programming, allowing for many options makes the code much more complicated. Some sort of decision has to be made each iteration of the loop work out which parts of the code to execute. In the object-oriented approach, each algorithm is contained within an object, and only one object exists a time so no choice needs to be made as to which part of the loop needs to execute. The code for each of the different random algorithms is completely isolated within their own type of object. This approach makes the code simpler, much more readable and easier to maintain.

The remaining part of the program loop involves the creation of the random generator objects themselves. This occurs inside the *handleSwitches()* function on line 3 which determines if either of the two switches has changed since the last iteration of the loop. If they have changed, the current random object is destroyed, and a new one is created and initialised, according to the specific combination of switch values. Because the EuroDuino module has two three position switches, design the software so that the user can select one of nine different random generators.

The basic random distribution object blueprint

In my software, I create random generator objects that are based on the following template. This is a simple C++ class definition, and besides having the ability to generate random numbers having a uniform distribution (like white noise),it outlines the template that can be extended to create the more complex random generators.

```
class CDistribution{
  protected:
    float parm1 = 0.5;        // in the range 0..1.0
    float parm2 = 0.00;       // in the range 0..1.0
    float clip(float v){      // useful function to clip values to be between 0 and 1.0
      if (v > 1.0)
        v = 1.0;
      else if (v < 0)
        v = 0.0;
      return v;
    }
  public:
    float frand() {return random(32767) / 32767.0;}   // basic random generator in the
range of 0..1.0
    virtual void init() {};
    virtual byte gen(){ return frand()*255.0;}
    virtual void setParm1(float v) {parm1 = v;}
    virtual void setParm2(float v) {parm2 = v;}
};
```

The class specifies elements that will be present in all of the generators. It contains two memory locations, called *param1* and *param2,* which are used to shape the output of the various random algorithms. These memory locations are sometimes referred to by programmers as **attributes**. There are also four behaviours (usually called **methods**) that each random object can exhibit. These are:

- Create a Uniform distribution random value – the plain “vanilla” random generator - *frand()*
- Initialise the object – *init()*
- Set the two parameter values that control the random generator – *setParm1()* and *setParm2()*
- Generate a random number – *gen()*

The init() method is called immediately after a random object is created, *setParm1()* and *setParm2()* are called every iteration of the program loop, and *gen()* is called when a new random value needs to be created. The CDistribution class specifies what a random generator object looks like to the main program. We can easily make specialised random generators by making different versions of the *gen()* method. (We also often make specialised versions the *init()* method.)

Creating specialised objects to create specific types of randomness

C++ is designed to make it easy extend the basic behaviour of a class by extending its behaviour. This uses a mechanism called **inheritance**, where replacing and or adding some of methods and attributes can create a new variation of the class. I have created new objects by adding a new method, which contains the specific random algorithm, and use it in a rewritten version of the *gen()* method that is specific to the new object. An example, of another type of random object to create a triangular shaped probability distribution is shown. The class CTriangle, is a variant of CDistribution.

```
class CTriangle : public CDistribution
{
  float tri(){
    return (frand() + frand()) * 0.5;
  }

public:
  byte gen(){ return tri()*255.0;}
};
```

The output of this produces a probability density function, which indicates how frequently particular outputs occur, is shown in Figure 1. A plot of the output is on the left, and a graph indicating how often an output occurs is on the right.

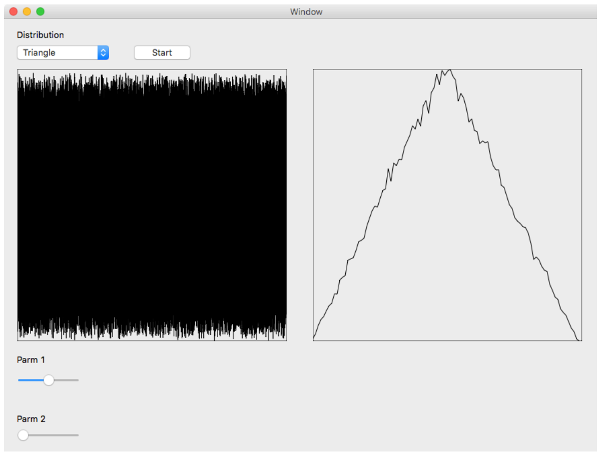
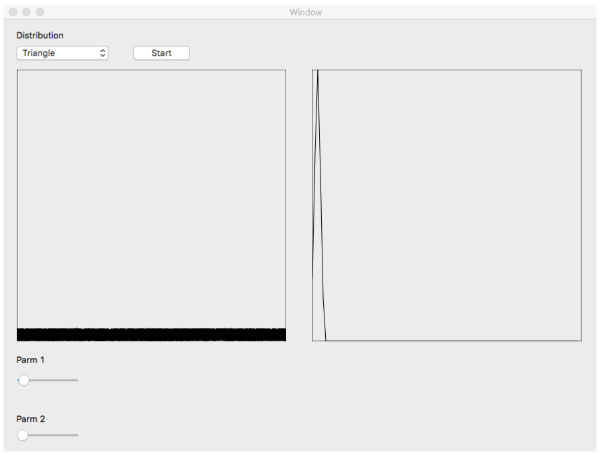


Figure 1. The output of the CTriangle object with a 0.5 scaling factor (500,000 samples shown)

Replacing the 0.5 scaling factor by a variable, determined by a hardware knob or a control voltage, the calculation int the tri() method becomes:

Return (frand() + frand()) * parm1;

This yields a useful variety of random distributions, such as those shown in Figure 2 and Figure 3.



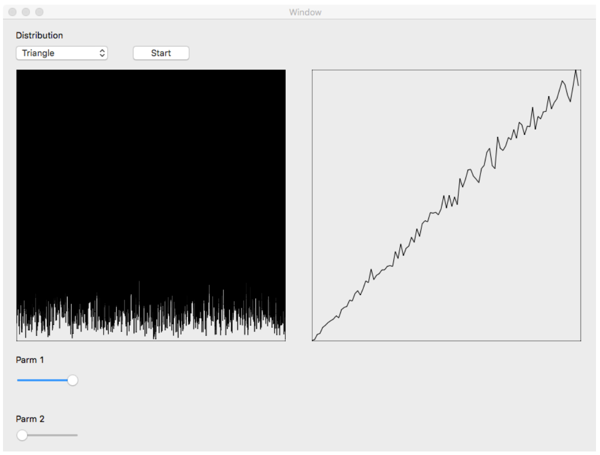
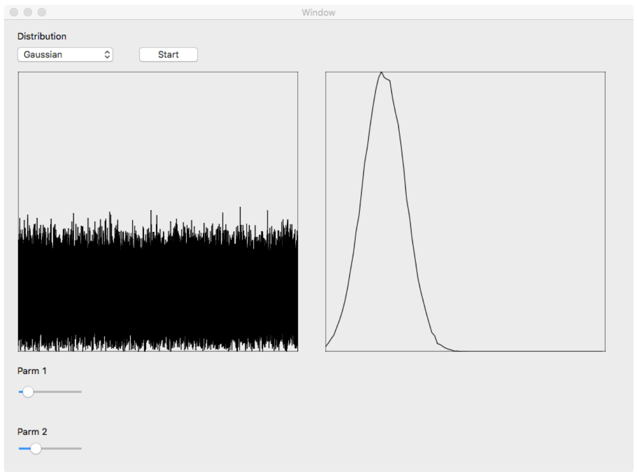


Figure 2. Triangle object output with low parm1 value.

Figure 3. Triangle object output with high parm1 value

One distribution I find useful is the Gaussian distribution. Figures 5 and 6 illustrate so of its possibilities.

```
class CGauss : public CDistribution
{
  float gauss() {
    int N  = 12;
    float halfN = N * 0.5;//6 ;
    float scale = 1 / N ;
    float sum = 0;
    float r = 0;
    for (int k=0; k < N; k++) {
       r = frand();//random(255);
     sum= sum+r;
      }
      return clip(parm2 * scale * (sum - halfN) + parm1);
  }
  public:
  byte gen(){ return gauss() * 255;}
};
```



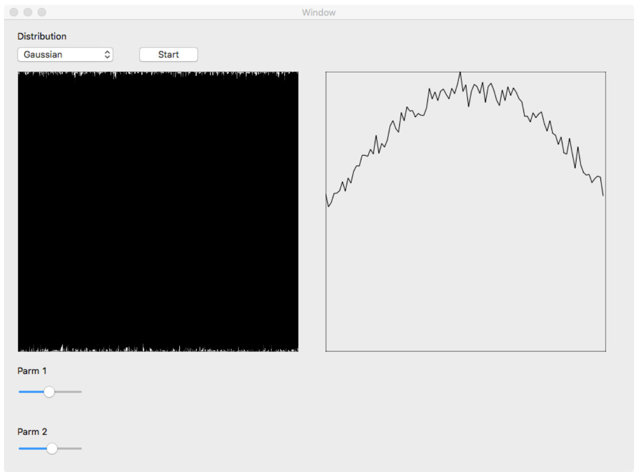


Figure 4. Gauss with low mean

Figure 5. Gauss with wider spread

Conclusion

The program structure outlined here has proven to be a very useful framework for quickly creating a variety of random number generators. Many algorithms can be found in (Lorrain, 1980) and (Dodge & Jerse, 1997), as well as many other math references. The approach taken to produce the Triangle and Gaussian distribution examples shown above can easily be adapted to produce Poisson, Weibull, Linear, Beta distributions, and other types of generators such as 1/f noise and the Logistic function can also be easily implemented. Examples can be found online.

Bibliography

Lorrain, Dennis, ‘A panoply of stochastic'cannons', *Computer Music Journal* , (1980) 53-81.

Strange, Allen. *Electronic music: systems, techniques, and controls* (2 ed.). (Dubuque, IA: Wm. C. Brown Company Publishers, 1983).

Dodge, Charles., & Jerse, Thomas. A. *Computer Music Synthesis, Composition, and Performance* (2 ed.). (New York, NY: Schirmer Books, 1997).