Enhancing Sensor Efficiency with Monte Carlo Subsampling
Jeremy Rand
Team SNARC (Sooners / Norman Advanced Robotics Coalition)
jeremy@veclabs.net

Enhancing Sensor Efficiency with Monte Carlo Subsampling

1 Introduction

Botball now features many high-resolution, high-bandwidth sensors for students to experiment with, but the CPU speed of the Link controller can often appear to be a limiting factor. Simply taking the centroid (center of mass) of a region of the Asus Xtion's point cloud can take 700-800 milliseconds – a crippling frame rate for many tasks. Botballers are seemingly left with three options: wait for a faster controller, decrease the resolution of the data, or choose simpler processing tasks. But what if there were a way to simplify processing without decreasing the resolution of the data? Monte Carlo subsampling offers a potential solution.

This paper will discuss the idea behind Monte Carlo, and then give an example of its usage with the Asus Xtion. It will focus more on the basics rather than going into advanced statistics theory (and is therefore likely to contain some simplifications).

2 Statistical Background (Representation of Distributions)

Many operations which we might want to do with a dataset operate on a *distribution* of data. A distribution is a mathematical object, different from a number. Probably the most famous distribution is the “bell curve” (also called the Gaussian distribution) which is used to model a large number of real-world situations. A distribution can be described in many ways. For example, many distributions can be described by an “average” (a typical value) and a “spread” (how much variation from the typical value occurs). There are two critical observations here:

(1) A distribution can be described either in terms of average and spread, or in terms of a random set of sample data which fits the distribution. These two representations are actually the same thing conceptually, just represented differently (subject to the error introduced by the sampling process).

(2) By extension, two different random samples of the same distribution are equivalent conceptually (subject to the error introduced by both sampling processes).

This means that we can convert between different samples, and (subject to sampling error) end up with equivalent results.

3 Statistical Background (Sampling)

Statistically, it can be demonstrated that a relatively small sample of a distribution can still accurately represent it. In fact, for determining a mean, a sample size of only 30 usually yields good results.

For this property to hold, the sampling should be an SRS, or Simple Random Sample. This means that all possible selections of the sample should be equally likely, all possible values in

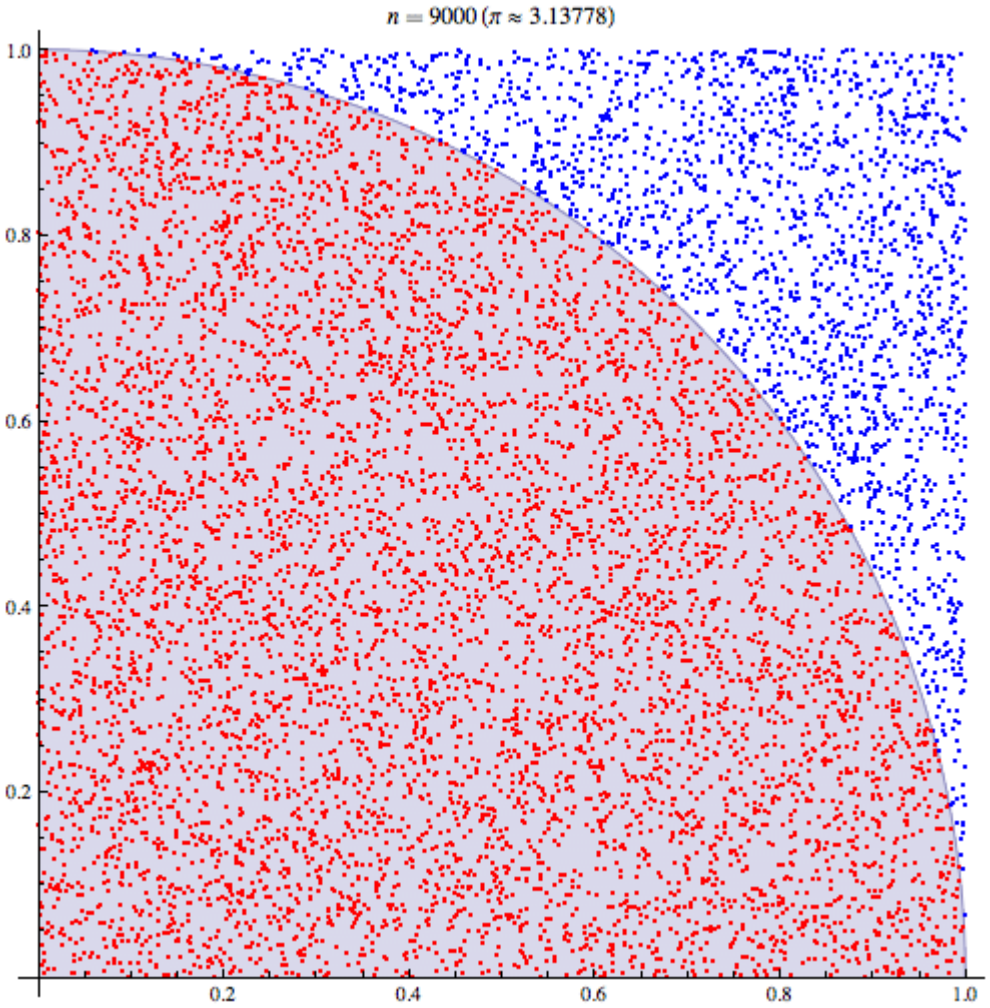
the sample have an equal chance of being chosen, and the choice of one value in the sample does not influence the rest of the sample.

We can take advantage of this.

4 Monte Carlo: Brute-Forcing a Random Sample

The Monte Carlo method is derived from the principle that many properties of a distribution which we might want to compute exactly, can more easily be approximated by taking a random sample of the data and applying the calculation to that sample.

For example, let's say we want to know the value of pi. If we were good at calculus, we might try calculating it with a Taylor series. But many problems are very difficult to solve with calculus, so let's look at using a Monte Carlo method instead.



(Image courtesy CaitlinJo via Wikimedia Commons [1])

This algorithm consists of drawing a quarter circle inscribed in a square, and then randomly picking points inside the square. After a while, the fraction of total points that lie inside the quarter circle converges toward the ratio of the quarter circle's area and the square's area. The definition of area tells us that this value is pi/4; multiplying by 4 yields pi. In this particular

example, 9000 points gives us 3.13778, which is sufficiently close to the true value of pi that for many purposes it would be usable. And of course, as the sample size increases, the accuracy improves.

5 The Xtion Centroid Problem

Let's say that there's a region of space which the Xtion's depth map provides, for which we want to calculate its centroid (center of mass). The following code might work for you:

```
int main()
{
  long col_iteration;
  long row_iteration;
  long long point_count;
  long long col_tally;
  long long row_tally;

  if(! depth_open())
  {
    printf("ERROR: Failed to open Xtion!\n");
    return 1;
  }

  set_depth_resolution(DEPTH_RESOLUTION_640_480);

  while(1)
  {
    // Get a new image
    while(!depth_update()) msleep(5);

    if(get_depth_image_height() != 480)
    {
      continue;
    }

    row_tally = 0;
    col_tally = 0;
    point_count = 0;

    for(row_iteration = 0; row_iteration < 480; row_iteration++)
    {
      for(col_iteration = 0; col_iteration < 640; col_iteration++)
      {
        if(get_depth_value(row_iteration, col_iteration) > 1000 &&
get_depth_value(row_iteration, col_iteration) < 2000)
        {
          row_tally += row_iteration;
          col_tally += col_iteration;
          point_count++;
        }
      }
    }

    if(point_count)
```

```
    {
      printf("%d ms: Row = %d, Col = %d\n", (int)(seconds() * 1000.0), (int)
(row_tally/point_count), (int)(col_tally/point_count));
    }
    else
    {
      printf("No blobs found\n");
    }

    msleep(5);
  }
}
```

This is pretty simple code; basically it loops through all points in the depth map, looks for points between 1000 and 2000 millimeters away, and adds them to a tally. It then divides that tally by the number of suitable points it found, to get the average. The average of both the row and column dimensions yields the centroid.

However, this code has a serious problem. It can take on the order of 800ms to process a single frame. This rules it out from real-time problems such as sometimes occur in Botball competitions.

6 Undesirable Ways of Improving Speed

We could, of course, simply reduce the horizontal and vertical resolutions of the for loop, e.g. by replacing this:

```
for(row_iteration = 0; row_iteration < 480; row_iteration++)
```

With this:

```
for(row_iteration = 0; row_iteration < 480; row_iteration+=5)
```

However, this causes problems because we have now introduced poorly-behaved (nonrandom) noise. For example, what happens if the entire scene shifts left by one pixel? The resulting shift will not follow an ideal Gaussian distribution, because the pixels will “jump” 5 pixels at a time, while staying still in between jumps.

How can we do this better?

7 Improving Speed with Monte Carlo Subsampling

Rather than sampling points via a 5-by-5 grid, we can sample points randomly. This works better because when a large object moves in any direction by even 1 pixel, it will be almost guaranteed to affect some sampled pixels.

First, we need random numbers. While the Link can generate these for us on the fly, that uses extra CPU time, which is the opposite of what we want. We can instead precompute some random point coordinates in a giant lookup table. For this, I really like Random.org. [2] Random.org generates scientifically accurate random numbers (using analog atmospheric noise),

and is so good that it is frequently used by lotteries. It's also free for most uses.

To get a nice set of random numbers from Random.org, go to the “Numbers -> Integers” page from their menu. Enter a nice large count of numbers (I used 10000), and min/max values (for Xtion width, 0 and 639 would be used since the Xtion's image width is 640 pixels). Format in 1 column for easy processing. Click “Get Numbers” and you'll have your random numbers.

Now you need to convert this to a C array. I just loaded it into a text editor, and did a mass search/replace from “\n” to “,\n”. The rest is just putting in an array name, curly braces, and a semicolon.

Do this for both the width and height of the Xtion image, and save them as .h files.

8 Putting It Together

Here's the changed code:

```
#include "/kovan/media/sda1/xtion_col_lut.h"
#include "/kovan/media/sda1/xtion_row_lut.h"

int main()
{
  long iteration;
  long long point_count;
  long long col_tally;
  long long row_tally;

  if(! depth_open())
  {
    printf("ERROR: Failed to open Xtion!\n");
    return 1;
  }

  set_depth_resolution(DEPTH_RESOLUTION_640_480);

  while(1)
  {
    // Get a new image
    while(!depth_update()) msleep(5);

    if(get_depth_image_height() != 480)
    {
      continue;
    }

    row_tally = 0;
    col_tally = 0;
    point_count = 0;

    for(iteration=0; iteration < 1000; iteration++)
    {
      if(get_depth_value(row_lut[iteration], col_lut[iteration]) > 1000 &&
get_depth_value(row_lut[iteration], col_lut[iteration]) < 2000)
      {
        row_tally += row_lut[iteration];
```

```
        col_tally += col_lut[iteration];
        point_count++;
      }
    }

    if(point_count)
    {
      printf("%d ms: Row = %d, Col = %d\n", (int)(seconds() * 1000.0), (int)
(row_tally/point_count), (int)(col_tally/point_count));
    }
    else
    {
      printf("No blobs found\n");
    }

    msleep(5);
  }
}
```

Note that we now #include the lookup tables we generated in the previous step, and we also now only have a single for loop instead of a nested for loop. Instead of using row and column iteration, we now calculate them both from a lookup table based on a single counter. Also notice that our sample size is only 1000 points. By contrast, the previous method used 640*480 = 307,200 points. That's a compression of over 300 times!

9 So How Well Does It Work?

Testing against a person standing 1.5 meters away, the Monte Carlo subsampling method was easily able to see movements as small as we could reliably produce with walking. And the calculations were so fast that the Xtion sensor's native framerate became the bottleneck rather than the CPU usage of processing frames.

To compress by 307.2 times with grid subsampling, you would need to cut each dimension by the square root of 307.2, which would yield a resolution of just 37 columns and 27 rows. This would drastically reduce the quality of the sample.

10 Conclusion

Monte Carlo subsampling appears quite usable for reducing CPU usage of data processing in Botball.

It's worth noting that we could also cheat and use compiler optimizations, which might improve speed to some (potentially large) extent with no accuracy costs. We didn't do this because the Link doesn't handle it natively, meaning hacking is necessary to get this feature. (But, if you're a hacker, feel free to try that!)

We are looking into practical applications of this technique, which you may or may not see at GCER.

Happy Tracking!

11 References

[1] Wikipedia Contributors. *Monte Carlo method*. https://en.wikipedia.org/wiki/Monte_Carlo_method

[2] Randomness and Integrity Services Ltd. Random.org. https://www.random.org/