Arbitrary-Precision Samplers for the Sum or Ratio of Uniform Random Numbers

This version of the document is dated 2020-08-23.

Peter Occil

This page presents new algorithms to sample the sum of uniform(0, 1) random numbers and the ratio of two uniform(0, 1) random numbers, with the help of **partially-sampled random numbers**, with arbitrary precision and without relying on floating-point arithmetic. See that page for more information on some of the algorithms made use of here, including **SampleGeometricBag** and **FillGeometricBag**.

1 About the Uniform Sum Distribution

The sum of n uniform(0, 1) random numbers has the following density function (see **MathWorld**):

$f(x) = (\Sigma_{k = 0, ..., n} (-1)^k * \text{choose}(n, k) * (x - k)^{n - 1} * \text{sign}(x - k)) / (2*(n-1)!)$.

For n uniform numbers, the distribution can take on values in the interval [0, n]. Note also that the density expresses a polynomial of degree $n - 1$.

The samplers given below for the uniform sum logically work as follows:

1. The distribution is divided into pieces that are each 1 unit wide (thus, for example, if n is 4, there will be four pieces).

2. An integer in [0, n) is chosen uniformly at random, call it i, then the piece identified by i is chosen. There are **many algorithms to choose an integer** this way, but an algorithm that is "optimal" in terms of the number of bits it uses, as well as unbiased, should be chosen.

3. The density function at [i, i + 1] is simulated. This is done by shifting the density so the desired piece of the density is at [0, 1] rather than its usual place. More specifically, the density is now as follows:

   - $f(x) = (\Sigma_{k = 0, ..., n} (-1)^k * \text{choose}(n, k) * ((x + i) - k)^{n - 1} * \text{sign}(i - k)) / (2*(n-1)!)$,

   where x is a real number in [0, 1]. Since f is a polynomial, it can be converted to a Bernstein polynomial whose control points describe the shape of the curve drawn out by f. A Bernstein polynomial has the form—

   - $\Sigma_{k = 0, ..., m} \text{choose}(m, k) * x^k * (1 - x)^{m - k} * a[k]$,

   where $a[k]$ are the control points and m is the polynomial's degree (here, $n - 1$). In this case, there will be n control points, which together trace out a 1-dimensional Bézier curve. Moreover, this polynomial can be simulated because it is continuous,

returns only numbers in [0, 1], and doesn't touch 0 or 1 anywhere inside the domain (except possibly at 0 and/or 1) (Keane and O'Brien 1994)[1].

4. The sampler creates a "coin" made up of a uniform partially-sampled random number (PSRN) whose contents are built up on demand using an algorithm called **SampleGeometricBag**. It flips this "coin" $n - 1$ times and counts the number of times the coin returned 1 this way, call it j. (The "coin" will return 1 with probability equal to the to-be-determined uniform random number. See (Goyal and Sigman 2012)[2].)

5. Based on j, the sampler accepts the PSRN with probability equal to the control point $a[j]$.

6. If the PSRN is accepted, it fills it up with uniform random digits, and returns i plus the finished PSRN. If the PSRN is not accepted, the sampler starts over from step 2.

2 Finding Parameters

Using the uniform sum sampler for an arbitrary n requires finding the Bernstein control points for each of the n pieces of the uniform sum density. This can be found, for example, with the Python code below, which uses the SymPy computer algebra library. In the code:

- `unifsum(x,n,v)` calculates the density function of the sum of `n` uniform random numbers when the variable `x` is shifted by `v` units.
- `find_control_points` prints out the control points for each piece of the density for the sum of `n` uniform random numbers, starting with piece 0.
- `find_areas` prints out the areas for each piece of that density. This can be useful to implement a variant of the sampler above, as detailed later in this section.

```
def unifsum(x,n,v):
    # Builds up the density at x (with offset v)
    # of the sum of n uniform random numbers
    ret=0
    x=x+v # v is an offset
    for k in range(n+1):
        s=(-1)**k*binomial(n,k)*(x-k)**(n-1)
        if k>v: ret-=s
        else: ret+=s
    return ret/(2*factorial(n-1))

def find_areas(n):
  x=symbols('x', real=True)
  areas=[integrate(unifsum(x,n,i),(x,0,1)) for i in range(n)]
  print(areas)

def find_control_points(n):
 x=symbols('x', real=True)
 for i in range(n):
  # Find the "usual" coefficients of the uniform
  # sum polynomial at offset i.
  poly=Poly(unifsum(x, n, i))
  coeffs=[poly.coeff_monomial(x**i) for i in range(n)]
  # Build coefficient vector
  coeffs=Matrix(coeffs)
  # Build power-to-Bernstein basis matrix
  mat=[[0 for _ in range(n)] for _ in range(n)]
  for j in range(n):
```

```
    for k in range(n):
        if k==0 or j==n-1:
          mat[j][k]=1
        elif k<=j:
          mat[j][k]=binomial(j, j-k) / binomial(n-1, k)
        else:
          mat[j][k]=0
  mat=Matrix(mat)
  # Print out the Bernstein control points
  print(mat*coeffs)
```

The basis matrix is found, for example, as Equation 42 of (Ray and Nataraj 2012)[3].

For example, if $n = 4$ (so a sum of four uniform random numbers is desired), the following control points are used for each piece of the density function:

Piece	Control Points
0	0, 0, 0, 1/6
1	1/6, 1/3, 2/3, 2/3
2	2/3, 2/3, 1/3, 1/6
3	1/6, 0, 0, 0

For more efficient results, all these control points could be scaled so that the highest control point is equal to 1. In the example above, after multiplying by 3/2 (the reciprocal of the highest control point, which is 2/3), the table would now look like this:

Piece	Control Points
0	0, 0, 0, 1/4
1	1/4, 1/2, 1, 1
2	1, 1, 1/2, 1/4
3	1/4, 0, 0, 0

Notice that all these control points are rational numbers, and the sampler may have to determine whether an event is true with probability equal to a control point. For rational numbers like these, it is possible to determine this exactly (using only random bits), using the **ZeroOrOne** method given in my **article on randomization and sampling methods**.

If the areas of the density's pieces are known in advance (and SymPy makes them easy to find as the `find_areas` method shows), then the sampler could be modified as follows, since each piece is now chosen with probability proportional to the chance that a random number there will be sampled:

- Step 2 is changed to read: "An integer in [0, n) is chosen with probability proportional to the corresponding piece's area, call the integer i, then the piece identified by i is chosen. There are many **algorithms to choose an integer** this way, but it's recommended to use one that takes integers rather than floating-point numbers as weights, and perhaps one that is economical in terms of the number of random bits it uses. In this sense, the Fast Loaded Dice Roller (Saad et al. 2020)[4] comes within 6 bits of the optimal number of random bits used on average."
- The last sentence in step 6 is changed to read: "If the PSRN is not accepted, the sampler starts over from step 3." With this, the same piece is sampled again.

If the sampler is modified this way, then the control points should be scaled so that the

highest control point of *each* piece is equal to 1. However, not doing this additional step does not affect the sampler's correctness. The following table is an example:

Piece Control Points

Piece	Control Points
0	0, 0, 0, 1
1	1/4, 1/2, 1, 1
2	1, 1, 1/2, 1/4
3	1, 0, 0, 0

3 Sum of Two Uniform Random Numbers

The following algorithm samples the sum of two uniform random numbers.

1. Create an empty uniform PSRN (partially-sampled random number), call it *ret*.
2. Generate an unbiased random bit.
3. Remove all digits from *ret*. (This algorithm, and others on this page, works for digits of any base, including base 10 or decimal, or base 2 or binary.)
4. Call the **SampleGeometricBag** algorithm on *ret*, then generate an unbiased random bit.
5. If the bit generated in step 2 is 1 and the result of **SampleGeometricBag** is 1, fill *ret* with uniform random digits as necessary to give its fractional part the desired number of digits (similarly to **FillGeometricBag**), and return *ret*.
6. If the bit generated in step 2 is 0 and the result of **SampleGeometricBag** is 0, fill *ret* as in step 4, and return 1 + *ret*.
7. Go to step 3.

And here is Python code that implements this algorithm. It uses floating-point arithmetic only at the end, to convert the result to a convenient form, and that it relies on methods from *randomgen.py* and *bernoulli.py*.

```
def sum_of_uniform(bern, precision=53):
    """ Exact simulation of the sum of two uniform
        random numbers. """
    bag=[]
    rb=bern.randbit()
    while True:
       bag.clear()
       if rb==1:
          # 0 to 1
          if bern.geometric_bag(bag)==1:
             return bern.fill_geometric_bag(bag, precision)
       else:
          # 1 to 2
          if bern.geometric_bag(bag)==0:
             return 1.0 + bern.fill_geometric_bag(bag, precision)
```

4 Sum of Three Uniform Random Numbers

The following algorithm samples the sum of three uniform random numbers.

1. Create an empty uniform PSRN, call it *ret*.
2. Choose an integer in 0, 6), uniformly at random. (With this, the left piece is chosen at a 1/3 chance, the right piece at 1/6 and the middle piece at 2/3, corresponding to the relative areas occupied by the three pieces.)

3. Remove all digits from *ret*.
4. If 0 was chosen by step 2, we will sample from the left piece of the function for the sum of three uniform random numbers. This piece runs along the interval [0, 1) and is a Bernstein polynomial with control points (0, 0, 1/2). (Bernstein polynomials are the backbone of the well-known Bézier curve. In this case, the density's value at 0 is 0, or the first control point, and its value at 1 is 1/2, or the third control point.) Thus, we can simulate this polynomial with the algorithm of Goyal and Sigman (2012)[(2)], using a partially-sampled uniform random number, as follows: Call the **SampleGeometricBag** algorithm twice on *ret*. If both of these calls return 1, then accept *ret*. (Alternatively, "...accept *ret* with probability 1/2", since 1/2 is the third control point and corresponds to both calls returning 1, but the step as given is still correct since it merely moves the upper bound closer to the top of the left piece.) If *ret* was accepted, fill *ret* with uniform random digits as necessary to give its fractional part the desired number of digits (similarly to **FillGeometricBag**), and return *ret*.
5. If 2, 3, 4, or 5 was chosen by step 2, we will sample from the middle piece of the density, which runs along the interval [1, 2) and is a Bernstein polynomial with control points (1/2, 1, 1/2). Call the **SampleGeometricBag** algorithm twice on *ret*. If neither or both of these calls return 1, then accept *ret*. Otherwise, if one of these calls returns 1 and the other 0, then accept *ret* with probability 1/2. If *ret* was accepted, fill *ret* as given in step 4 and return 1 + *ret*.
6. If 1 was chosen by step 2, we will sample from the right piece of the density, which runs along the interval [2, 1) and is a Bernstein polynomial with control points (1/2, 0, 0). Call the **SampleGeometricBag** algorithm twice on *ret*. If both of these calls return 0, then accept *ret*. (Alternatively, "...accept *ret* with probability 1/2", 1/2 is the first control point and corresponds to both calls returning 0, but the step is given is still correct.) If *ret* was accepted, fill *ret* as given in step 4 and return 2 + *ret*.
7. Go to step 3.

And here is Python code that implements this algorithm.

```python
def sum_of_uniform3(bern):
    """ Exact simulation of the sum of three uniform
        random numbers. """
    r=6
    while r>=6:
        r=bern.randbit() + bern.randbit() * 2 + bern.randbit() * 4
    while True:
        # Choose a piece of the PDF uniformly (but
        # not by area).
        bag=[]
        if r==0:
            # Left piece
            if bern.geometric_bag(bag) == 1 and \
               bern.geometric_bag(bag) == 1:
                # Accepted
                return bern.fill_geometric_bag(bag)
        elif r<=4:
            # Middle piece
            ones=bern.geometric_bag(bag) + bern.geometric_bag(bag)
            if (ones == 0 or ones == 2) and bern.randbit() == 0:
                # Accepted
                return 1.0 + bern.fill_geometric_bag(bag)
            if ones == 1:
                # Accepted
                return 1.0 + bern.fill_geometric_bag(bag)
        else:
            # Right piece
            if bern.randbit() == 0 and \
```

```
            bern.geometric_bag(bag) == 0 and \
            bern.geometric_bag(bag) == 0:
             # Accepted
            return 2.0 + bern.fill_geometric_bag(bag)
```

5 Ratio of Two Uniform Random Numbers

The ratio of two uniform(0,1) random numbers has the following density function (see **MathWorld**):

- 1/2 if $x >= 0$ and $x <= 1$,
- $(1/ x^2) / 2$ if $x > 1$, and
- 0 otherwise.

The following algorithm simulates this density.

1. With probability 1/2, return either an empty uniform PSRN or a uniform random number in [0, 1) whose fractional part contains the desired number of digits.
2. At this point, the result will be 1 or greater. Set *intval* to 1 and set *size* to 1.
3. With probability 1/2, add *size* to *intval*, then multiply *size* by 2, then repeat this step. (This step chooses an interval beyond 1, taking advantage of the fact that the area under the density between 1 and 2 is 1/4, between 2 and 4 is 1/8, between 4 and 8 is 1/16, and so on, so that an appropriate interval is chosen with the correct probability.)
4. Generate an integer in the interval [*intval*, *intval* + *size*) uniformly at random, call it *i*.
5. Create an empty uniform PSRN, *ret*.
6. Call the **sub-algorithm** below with $d = intval$ and $c = i$. If the call returns 0, go to step 4. (Here we simulate *intval*/(*i*+λ) rather than 1/(*i*+λ) in order to increase acceptance rates in this step. This is possible without affecting the algorithm's correctness.)
7. Call the **sub-algorithm** below with $d = 1$ and $c = i$. If the call returns 0, go to step 4.
8. The PSRN *ret* was accepted, so fill it with uniform random digits as necessary to give its fractional part the desired number of digits (similarly to **FillGeometricBag**), and return $i + ret$.

The algorithm above uses a sub-algorithm that simulates the probability $d / (c + λ)$, where λ is the probability built up by the uniform PSRN, as follows:

1. With probability $c / (1 + c)$, return a number that is 1 with probability d/c and 0 otherwise.
2. Call **SampleGeometricBag** on *ret* (the uniform PSRN). If the call returns 1, return 0. Otherwise, go to step 1.

And the following Python code implements this algorithm.

```python
def numerator_div(bern, numerator, intpart, bag):
    # Simulates numerator/(intpart+bag)
    while True:
      if bern.zero_or_one(intpart,1+intpart)==1:
          return bern.zero_or_one(numerator,intpart)
      if bern.geometric_bag(bag)==1: return 0

def ratio_of_uniform(bern):
    """ Exact simulation of the ratio of uniform random numbers."""
    # First, simulate the integer part
```

```
if bern.randbit():
    # This is 0 to 1, which follows a uniform distribution
    bag=[]
    return bern.fill_geometric_bag(bag)
else:
    # This is 1 or greater
    intval=1
    size=1
    count=0
    # Determine which range of integer parts to draw
    while True:
        if bern.randbit()==1:
            break
        intval+=size
        size*=2
        count+=1
    while True:
      # Draw the integer part
      intpart=random.randint(0, size-1) + intval
      bag=[]
      # Note: Density at [intval,intval+size) is multiplied
      # by intval, to increase acceptance rates
      if numerator_div(bern,intval,intpart,bag)==1 and \
         numerator_div(bern,1,intpart,bag)==1:
           return intpart + bern.fill_geometric_bag(bag)
```

6 Notes

[1] Keane, M. S., and O'Brien, G. L., "A Bernoulli factory", *ACM Transactions on Modeling and Computer Simulation* 4(2), 1994

[2] Goyal, V. and Sigman, K., 2012. On simulating a class of Bernstein polynomials. ACM Transactions on Modeling and Computer Simulation (TOMACS), 22(2), pp.1-5.

[3] S. Ray, P.S.V. Nataraj, "A Matrix Method for Efficient Computation of Bernstein Coefficients", *Reliable Computing* 17(1), 2012.

[4] Saad, F.A., Freer C.E., et al., "**The Fast Loaded Dice Roller: A Near-Optimal Exact Sampler for Discrete Probability Distributions**", arXiv:2003.03830v2 [stat.CO], also in AISTATS 2020: Proceedings of the 23rd International Conference on Artificial Intelligence and Statistics, Proceedings of Machine Learning Research 108, Palermo, Sicily, Italy, 2020.

7 License