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

This version of the document is dated 2022-11-07.

Peter Occil

2020 Mathematics Subject Classification: 68W20, 60-08.

This page presents new algorithms to sample the sum of uniform(0, 1) random variates and the ratio of two uniform(0, 1) random variates, with the help of **partially-sampled random numbers** (PSRNs), 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**.

The algorithms on this page work no matter what base the digits of the partially-sampled number are stored in (such as base 2 for decimal or base 10 for binary), unless noted otherwise.

1 Contents

2 About This Document

This is an open-source document; for an updated version, see the source code or its rendering on GitHub. You can send comments on this document on the GitHub issues page.

My audience for this article is **computer programmers with mathematics knowledge, but little or no familiarity with calculus**.

I encourage readers to implement any of the algorithms given in this page, and report their implementation experiences. In particular, **I seek comments on the following aspects**:

- Are the algorithms in this article easy to implement? Is each algorithm written so that someone could write code for that algorithm after reading the article?
- Does this article have errors that should be corrected?
- Are there ways to make this article more useful to the target audience?

Comments on other aspects of this document are welcome.

3 About the Uniform Sum Distribution

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

$$f(x)=\left(\sum_{k=0}^n (-1)^k {n\choose k} (x-k)^{n-1} \text{sign}(x-k)\right)/(2(n-1)!),$$

where ${n\choose k}$ is a *binomial coefficient*, or the number of ways to choose k out of n labeled items, and sign(x) is 1 if x is greater than 0, or 0 if x is 0, or -1 is less than 0.[1]

For n uniform numbers, the distribution can take on values in the interval [0, n]. Note also that the PDF 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 long (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 PDF at [i, i + 1] is simulated. This is done by shifting the PDF so the desired piece of the PDF is at [0, 1] rather than its usual place. More specifically, the PDF is now as follows: $$f(x)=\left(\sum_{k=0}^n (-1)^k {n\choose k} ((x+i)-k)^{n-1} \text{sign}((x+i)-k)\right)/(2(n-1)!),$$ where x is a real number in [0, 1]. Since f is a polynomial, it can be rewritten in Bernstein form, so that it has *Bernstein coefficients*, which are equivalent to control points describing the shape of the curve drawn out by f. (The Bernstein coefficients are the backbone of the well-known Bézier curve.) A polynomial can be written in Bernstein form as— $$\sum_{k=0}^m {m\choose k} x^k (1-x)^{m-k} a[k],$$ 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. For example, given control points 0.2, 0.3, and 0.6, the curve is at 0.2 when x = 0, and 0.6 when x = 1. (Note that the curve is not at 0.3 when x = 1/2; in general, Bézier curves do not cross their control points other than the first and the last.)

   Moreover, this polynomial can be simulated because its Bernstein coefficients all lie in [0, 1] (Goyal and Sigman 2012)[2].

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 variate.)

5. Based on j, the sampler accepts the PSRN with probability equal to the control point $a[j]$. (See (Goyal and Sigman 2012)[3].)

6. If the PSRN is accepted, the sampler optionally fills it up with uniform random digits, then sets the PSRN's integer part to *i*, then the sampler returns the finished PSRN. If the PSRN is not accepted, the sampler starts over from step 2.

4 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 PDF. 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 PDF of the sum of `n` uniform random variates when the variable `x` is shifted by `v` units.
- `find_control_points` returns the control points for each piece of the PDF for the sum of `n` uniform random variates, starting with piece 0.
- `find_areas` returns the relative areas for each piece of that PDF. 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 PDF at x (with offset v)
    # of the sum of n uniform random variates
    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)
           # Equivalent to k>x+v since x is limited
           # to [0, 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)]
   g=prod([v.q for v in areas])
   areas=[int(v*g) for v in areas]
   g=gcd(areas)
   areas=[v/int(g) for v in areas]
```

```
   return areas

def find_control_points(n, scale_pieces=False):
 x=symbols('x', real=True)
 controls=[]
 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)
  # Get the Bernstein control points
  mv = mat*coeffs
  mvc = [Rational(mv[i]) for i in range(n)]
  maxcoeff = max(mvc)
  # If requested, scale up control points to raise acceptance rate
  if scale_pieces:
     mvc = [v/maxcoeff for v in mvc]
  mv = [[v.p, v.q] for v in mvc]
  controls.append(mv)
 return controls
```

The basis matrix is found, for example, as Equation 42 of (Ray and Nataraj 2012)[4].

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

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. This doesn't affect the algorithm's correctness because scaling a Bézier curve's control points scales the curve accordingly, as is well known. 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 the following:

- 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**.
- The first and last piece of the PDF have a predictable set of control points. Namely the control points are as follows:
  - Piece 0: 0, 0, ..., 0, $1/((n-1)!)$, where $(n-1)! = 1*2*3*...*(n-1)$.
  - Piece $n-1$: $1/((n-1)!)$, 0, 0, ..., 0.

If the areas of the PDF'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 variate 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."

- 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.
- The following are additional modifications that should be done to the sampler. However, not applying them does not affect the sampler's correctness.
    - The control points should be scaled so that the highest control point of *each* piece is equal to 1. See the table below for an example.
    - If piece 0 is being sampled and the PSRN's digits are binary (base 2), the "coin" described in step 4 uses a modified version of **SampleGeometricBag** in which a 1 (rather than any other digit) is sampled from the PSRN when it reads from or writes to that PSRN. Moreover, the PSRN is always accepted regardless of the result of the "coin" flip.
    - If piece $n-1$ is being sampled and the PSRN's digits are binary (base 2), the "coin" uses a modified version of **SampleGeometricBag** in which a 0 (rather than any other digit) is sampled, and the PSRN is always accepted.

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

5 Sum of Two Uniform Random Variates

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

1. Create a positive-sign zero-integer-part uniform PSRN (partially-sampled random number), call it *ret*.
2. Generate an unbiased random bit (that is, either 0 or 1, chosen with equal probability).
3. Remove all digits from *ret*. (This algorithm works for digits of any base, including base 10 for decimal, or base 2 for 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, optionally fill *ret* with uniform random digits as necessary to give its fractional part the desired number of digits (similarly to **FillGeometricBag**), then return *ret*.
6. If the bit generated in step 2 is 0 and the result of **SampleGeometricBag** is 0, optionally fill *ret* as in step 5, then set *ret*'s integer part to 1, then return *ret*.
7. Go to step 3.

For base 2, the following algorithm also works, using certain "tricks" described in the next section.

1. Generate an unbiased random bit (that is, either 0 or 1, chosen with equal probability), call it *d*.
2. Generate unbiased random bits until 0 is generated this way. Set *g* to the number of one-bits generated by this step.
3. Create a positive-sign zero-integer-part uniform PSRN (partially-sampled random number), call it *ret*. Then, set the digit at position *g* of the PSRN's fractional part to *d* (positions start at 0 in the PSRN).
4. Optionally, fill *ret* with uniform random digits as necessary to give its fractional part the desired number of digits (similarly to **FillGeometricBag**). Then set *ret*'s integer part to $(1 - d)$, then return *ret*

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 variates. """
    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)

def sum_of_uniform_base2(bern, precision=53):
    """ Exact simulation of the sum of two uniform
          random variates (base 2). """
    if bern.randbit()==1:
      g=0
      while bern.randbit()==0:
          g+=1
      bag=[None for i in range(g+1)]
      bag[g]=1
      return bern.fill_geometric_bag(bag)
    else:
      g=0
      while bern.randbit()==0:
          g+=1
      bag=[None for i in range(g+1)]
      bag[g]=0
      return bern.fill_geometric_bag(bag) + 1.0
```

6 Sum of Three Uniform Random Variates

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

1. Create a positive-sign zero-integer-part uniform PSRN, call it *ret*.
2. Choose an integer in [0, 6), uniformly at random. (With this, the left piece is chosen at a 1/6 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 variates. This piece runs along the interval [0, 1) and is a polynomial with Bernstein

coefficients of (0, 1, 1/2) (and is thus a Bézier curve with those control points). Due to the particular form of the control points, the piece can be sampled in one of the following ways:

- Call the **SampleGeometricBag** algorithm twice on *ret*. If both of these calls return 1, then accept *ret* with probability 1/2. This is the most "naïve" approach.
- Call the **SampleGeometricBag** algorithm twice on *ret*. If both of these calls return 1, then accept *ret*. This version of the step is still correct since it merely scales the polynomial so its upper bound is closer to 1, which is the top of the left piece, thus improving the acceptance rate of this step.
- Base-2 only: Call a modified version of **SampleGeometricBag** twice on *ret*; in this modified algorithm, a 1 (rather than any other digit) is sampled from *ret* when that algorithm reads or writes a digit in *ret*. Then *ret* is accepted. This version will always accept *ret* on the first try, without rejection, and is still correct because *ret* would be accepted by this step only if **SampleGeometricBag** succeeds both times, which will happen only if that algorithm reads or writes out a 1 each time (because otherwise the control point is 0, meaning that *ret* is accepted with probability 0).

If *ret* was accepted, optionally fill *ret* with uniform random digits as necessary to give its fractional part the desired number of digits (similarly to **FillGeometricBag**), then return *ret*.

5. If 2, 3, 4, or 5 was chosen by step 2, we will sample from the middle piece of the PDF, which runs along the interval [1, 2) and is a polynomial with Bernstein coefficients (control points) of (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, optionally fill *ret* as given in step 4, then set *ret*'s integer part to 1, then return *ret*.

6. If 1 was chosen by step 2, we will sample from the right piece of the PDF, which runs along the interval [2, 3) and is a polynomial with Bernstein coefficients (control points) of (1/2, 0, 0). Due to the particular form of the control points, the piece can be sampled in one of the following ways:

   - Call the **SampleGeometricBag** algorithm twice on *ret*. If both

    of these calls return 0, then accept *ret* with probability 1/2. This is the most "naïve" approach.
  - Call the **SampleGeometricBag** algorithm twice on *ret*. If both of these calls return 0, then accept *ret*. This version is correct for a similar reason as in step 4.
  - Base-2 only: Call a modified version of **SampleGeometricBag** twice on *ret*; in this modified algorithm, a 0 (rather than any other digit) is sampled from *ret* when that algorithm reads or writes a digit in *ret*. Then *ret* is accepted. This version is correct for a similar reason as in step 4.

   If *ret* was accepted, optionally fill *ret* as given in step 4, then set *ret*'s integer part to 2, then return *ret*.

7. Go to step 3.

And here is Python code that implements this algorithm.

```
def sum_of_uniform3(bern):
    """ Exact simulation of the sum of three uniform
          random variates. """
    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)
```

7 Ratio of Two Uniform Random Variates

The ratio of two uniform(0,1) random variates has the following PDF (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 PDF.

1. Generate an unbiased random bit. If that bit is 1 (which happens with probability 1/2), we have a uniform(0, 1) random variate. Create a positive-sign zero-integer-part uniform PSRN, then optionally fill the PSRN with uniform random digits as necessary to give the number's fractional part the desired number of digits (similarly to **FillGeometricBag**), then return the PSRN.
2. At this point, the result will be 1 or greater. Set *intval* to 1 and set *size* to 1.
3. Generate an unbiased random bit. If that bit is 1 (which happens with probability 1/2), go to step 4. Otherwise, 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 PDF 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 a positive-sign zero-integer-part 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 set *ret*'s integer part to i, then optionally fill *ret* with uniform random digits as necessary to give its fractional part the desired number of digits (similarly to **FillGeometricBag**), then return *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.

```
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 variates."""
    # 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
       # Determine which range of integer parts to draw
       while True:
           if bern.randbit()==1:
                break
           intval+=size
           size*=2
       while True:
```

```
        # Draw the integer part
        intpart=bern.rndintexc(size) + 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)
```

8 Reciprocal of Uniform Random Variate

The reciprocal of a uniform(0, 1) random variate has the PDF—

- 1 / x^2 if x > 1, and
- 0 otherwise.

The algorithm to simulate this PDF is the same as the algorithm for the ratio of two uniform random variates, except step 1 is omitted.

9 Notes

10 License

---

1. choose(n, k) = (1*2*3*...*n)/((1*...*k)*(1*...*(n−k))) = n!/(k! * (n − k)!) = ${n\choose k}$ is a *binomial coefficient*, or the number of ways to choose k out of n labeled items. It can be calculated, for example, by calculating i/(n−i+1) for each integer i in [n−k+1, n], then multiplying the results (Yannis Manolopoulos. 2002. "Binomial coefficient computation: recursion or iteration?", SIGCSE Bull. 34, 4 (December 2002), 65–67. DOI: **https://doi.org/10.1145/820127.820168**). Note that for every m>0, choose(m, 0) = choose(m, m) = 1 and choose(m, 1) = choose(m, m−1) = m; also, in this document, choose(n, k) is 0 when k is less than 0 or greater than n.

$n! = 1*2*3*...*n$ is also known as n factorial; in this document, $(0!) = 1$.
Summation notation, involving the Greek capital sigma (Σ), is a way to write the sum of one or more terms of similar form. For example, $\sum_{k=0}^n g(k)$ means $g(0)+g(1)+...+g(n)$, and $\sum_{k\ge 0} g(k)$ means $g(0)+g(1)+...$.↩

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. 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.↩

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