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

This version of the document is dated 2020-09-20.

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** (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 the Uniform Sum Distribution

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

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

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:

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

    where x is a real number in [0, 1], and choose(n, k) = n!/(k! * (n − k)!) is the binomial coefficient. 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. (Bernstein polynomials are the backbone of the well-known Bézier curve.) A Bernstein polynomial has the form—

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

    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. 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 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.

3 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 numbers 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 numbers, 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 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)
           # 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)[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 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, $1/((n-1)!)$.
    - 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 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.

- 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

4 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 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, 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.

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

1. Generate an unbiased random bit, 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 an empty 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. Fill *ret* with uniform random digits as necessary to give its fractional part the desired number of digits (similarly to **FillGeometricBag**), and return $(1 - d) + ret$.

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 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)

def sum_of_uniform_base2(bern, precision=53):
    """ Exact simulation of the sum of two uniform
          random numbers (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
```

5 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 1/6 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 numbers. This piece runs along the interval [0, 1) and is a Bernstein polynomial (and Bézier curve) with control points (0, 0, 1/2). 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, 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 PDF, 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 PDF, which runs along the interval [2, 3) and is a Bernstein polynomial with control points (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, 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.

```
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)
```

6 Ratio of Two Uniform Random Numbers

The ratio of two uniform(0,1) random numbers 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. With probability 1/2, we have a uniform(0, 1) random number. Create an empty uniform PSRN, then either return that PSRN as is or fill it with uniform random digits as necessary to give the number's fractional part the desired number of digits (similarly to **FillGeometricBag**) and return the resulting number. 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 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 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.

```
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=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)
```

7 Addition and Subtraction of Two PSRNs

The following shows how to add or subtract two uniform PSRNs (**a** and **b**) that store digits of the same base, and get a uniform PSRN as a result. The input PSRNs may be negative or non-negative, and it is assumed that their integer parts and signs were sampled.

1. If **a** has unsampled digits before the last sampled digit in its fractional part, set each of those unsampled digits to a digit chosen uniformly at random. Do the same for **b**.
2. If **a** has fewer digits in its fractional part than **b** (or vice versa), sample enough digits (by setting them to uniform random digits) so that both PSRNs' fractional parts have the same number of digits. Now, let *digitcount* be the number of digits in **a**'s fractional part.
3. Add or subtract **a** and **b** to form a new PSRN, **c**. (For example, if **a** is 0.12344... and **b** is 0.38925..., their fractional parts are added to form **c** = 0.51269.... In this example, **c** is the smallest result of applying interval addition to the PSRNs [0.12344, 0.12345] and [0.38925, 0.38926], respectively, namely [0.51269, 0.51271]. However, the PSRN **c** is not uniformly distributed in this latter interval, and this is what the rest of this algorithm will solve, since in fact, the distribution of numbers in the interval resembles the distribution of the sum of two uniform random numbers.)
4. If the base of **a**'s and **b**'s digits is 2, and if **c** is non-negative:
    1. Generate an unbiased random bit, 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 substep.
    3. Set the digit at position (g + $digitcount$) of **c**'s fractional part to *d* (positions start at 0 in the PSRN, and digits between the positions ($digitcount$ − 1) and (g + $digitcount$) will be unsampled).
    4. If *d* is 0, sum the constant $2^{-digitcount}$ into **c**.
5. Otherwise:

1. Generate an unbiased random bit, call it *d*. Create an empty uniform PSRN, call it _ret.
2. Remove all digits from *ret*.
3. Call the **SampleGeometricBag** algorithm on *ret*. If the result of the call (which can be 0 or 1) is not equal to *d*, go to the previous substep.
4. Append the digits (both sampled and unsampled) of *ret*'s fractional part into **c**'s fractional part.
5. If *d* is 0 and **c** is non-negative, or if *d* is 1 and **c** is negative, sum the constant $2^{-digitcount}$ into **c**.

And the following Python code implements this algorithm.

```
def _add_neg_power_of_base(psrn, pwr, digits=2):
    """ Adds digits^(-pwr) to a PSRN. """
    if pwr <= 0:
        raise ValueError
    i = pwr - 1
    incr = -1 if psrn[0] < 0 else 1
    while i >= 0:
        psrn[2][i] += incr
        if psrn[2][i] < 0:
            psrn[2][i] += digits
        elif psrn[2][i] > digits:
            psrn[2][i] -= digits
        else:
            return psrn
        i -= 1
    psrn[1] += incr
    return psrn

def add_psrns(psrn1, psrn2, digits=2):
    """ Adds two partially-sampled random numbers.
        psrn1: List containing the sign, integer part, and fractional part
            of the first PSRN.  Fractional part is a list of digits
            after the point, starting with the first.
        psrn1: List containing the sign, integer part, and fractional part
            of the second PSRN.
        digits: Digit base of PSRNs' digits.  Default is 2, or binary. """
    if psrn1[0] == None or psrn1[1] == None or psrn2[0] == None or psrn2[1] == None:
        raise ValueError
    for i in range(len(psrn1[2])):
        psrn1[2][i] = (
            random.randint(0, digits - 1) if psrn1[2][i] == None else psrn1[2][i]
        )
    for i in range(len(psrn2)):
        psrn2[2][i] = (
            random.randint(0, digits - 1) if psrn2[2][i] == None else psrn2[2][i]
        )
    while len(psrn1[2]) < len(psrn2[2]):
        psrn1[2].append(random.randint(0, digits - 1))
    while len(psrn1[2]) > len(psrn2[2]):
        psrn2[2].append(random.randint(0, digits - 1))
    digitcount = len(psrn1[2])
    cpsrn = _add_psrns_internal(
        psrn1[0], psrn1[1], psrn1[2], psrn2[0], psrn2[1], psrn2[2], digits
    )
    # If negative, adjust to the correct region
    targetd = 1 if cpsrn[0] < 0 else 0
    if targetd == 1:
        _add_neg_power_of_base(cpsrn, digitcount)
    if digits == 2 and cpsrn[0] >= 0:
        g = 0
        d = random.randint(0, 1)
        while random.randint(0, 1) == 1:
            g += 1
        while len(cpsrn[2]) <= g + digitcount:
            cpsrn[2].append(None)
        cpsrn[2][g + digitcount] = d
        if d == 0:
            _add_neg_power_of_base(cpsrn, digitcount)
    else:
        sampleresult = 0
        d = random.randint(0, 1)
        while True:
            bag = []
            sampleresult = 1
            i = 0
            while True:
                while len(bag) <= i:
                    bag.append(None)
                if bag[i] == None:
                    bag[i] = random.randint(0, digits - 1)
                a1 = bag[i]
                b1 = random.randint(0, digits - 1)
                if a1 < b1:
                    sampleresult = 0
                if a1 > b1:
                    sampleresult = 1
                if a1 != b1:
                    break
                i += 1
            if d == sampleresult:
                for v in bag:
                    cpsrn[2].append(v)
                if d == targetd:
                    _add_neg_power_of_base(cpsrn, digitcount)
                break
    return cpsrn

def _add_psrns_internal(asign, aint, afrac, bsign, bint, bfrac, digits=2):
    # For this to work, fractional parts must have the same
    # length and all their digits must be sampled
    if len(afrac) != len(bfrac):
        raise ValueError
    if aint < 0 or bint < 0 or digits < 2:
        raise ValueError
    if asign == bsign:
        # Addition
        cfrac = [0 for i in range(len(afrac))]
        carry = 0
        i = len(afrac) - 1
        while i >= 0:
            b = afrac[i] + bfrac[i] + carry
            cfrac[i] = b % digits if b >= digits else b
            carry = 1 if b >= digits else 0
            i -= 1
        ci = aint + bint + carry
        return [asign, ci, cfrac]
    else:
        # Subtraction
        aIsLess = False
        if aint != bint:
            aIsLess = aint < bint
        else:
            for i in range(len(afrac)):
                if afrac[i] != bfrac[i]:
                    aIsLess = afrac[i] < bfrac[i]
                    break
        if aIsLess:
            tmp = aint
            aint = bint
            bint = tmp
            tmp = afrac
            afrac = bfrac
            bfrac = tmp
            tmp = asign
            asign = bsign
            bsign = tmp
        cfrac = [0 for i in range(len(afrac))]
        carry = 0
        i = len(afrac) - 1
        while i >= 0:
            b = afrac[i] - bfrac[i] - carry
            if b < 0:
                cfrac[i] = b + digits
                carry = 1
            else:
                cfrac[i] = b
                carry = 0
            i -= 1
        ci = abs(aint - bint - carry)
        csign = asign
        return [csign, ci, cfrac]
```

8 Rayleigh Distribution

The following is an arbitrary-precision sampler for the Rayleigh distribution with parameter s, which is a rational number greater than 0.

1. Set k to 0, and set y to 2 * s * s.
2. With probability exp(−(k * 2 + 1)/y), go to step 3. Otherwise, add 1 to k and repeat this step. (The probability check should be done with the **exp(−x/y) algorithm** in "Bernoulli Factory Algorithms", with x/y = (k * 2 + 1)/y.)
3. (Now we sample the piece located at [k, k + 1).) Create an empty uniform PSRN, and create an input coin that returns the result of **SampleGeometricBag** on that uniform PSRN.
4. Set ky to k * k / y.
5. (At this point, we simulate exp(−U^2/y), exp(−k^2/y) , exp(−U*k*2/y), as well as a scaled-down version of U + k, where U is the number built up by the uniform PSRN.) Call the **exp(−x/y) algorithm** with x/y = ky, then call the **exp(−(λ^k * x)) algorithm** using the input coin from step 2, x = 1/y, and k = 2, then call the same algorithm using the same input coin, x = k * 2 / y, and k = 1, then call the **sub-algorithm** given later with the uniform PSRN and k = k. If all of these calls return 1, the uniform PSRN was accepted. Otherwise, remove all digits from the uniform PSRN and go to step 4.
6. If the uniform PSRN, call it ret, was accepted by step 5, fill it with uniform random digits as necessary to give its fractional part the desired number of digits (similarly to **FillGeometricBag**), and return k + ret.

The sub-algorithm below simulates (U+k)/$base^z$, where U is the number built by the uniform PSRN, *base* is the base (radix) of digits stored by that PSRN, k is an integer 0 or greater, and z is the number of significant digits in k (for this purpose, z is 0 if k is 0).

For base 2:

1. Set N to 0.
2. With probability 1/2, go to the next step. Otherwise, add 1 to N and repeat this step.
3. If N is less than z, return rem(k / 2^{z-1-N}, 2). (Alternatively, shift k to the right, by $z - 1 - N$ bits, then return k *AND* 1, where "*AND*" is a bitwise AND-operation.)
4. Subtract z from N. Then, if the item at position N in the uniform PSRN's fractional part (positions start at 0) is not set to a digit (e.g., 0 or 1 for base 2), set the item at that position to a digit chosen uniformly at random (e.g., either 0 or 1 for base 2), increasing the uniform PSRN's capacity as necessary.
5. Return the item at position N.

For bases other than 2, such as 10 for decimal, this can be implemented as follows (based on **URandLess**):

1. Set i to 0, and set **b** to an empty uniform PSRN.
2. If i is less than z:
    1. Set *da* to rem(k / 2^{z-1-i}, *base*), and set *db* to a digit chosen uniformly at random (that is, an integer in the interval [0, *base*)).
    2. Return 1 if *da* is less than *db*, or 0 if *da* is greater than *db*.
3. If i is z or greater:
    1. If the digit at position ($i - z$) in the input PSRN's fractional part is not set, set the item at that position to a digit chosen uniformly at random (positions start at 0 where 0 is the most significant digit after the point, 1 is the next, etc.), and append the result to that fractional part's digit expansion.
    2. Set *da* to the item at that position, and set *db* to a digit chosen uniformly at random (that is, an integer in the interval [0, *base*)).
    3. Return 1 if *da* is less than *db*, or 0 if *da* is greater than *db*.
4. Add 1 to i and go to step 3.

9 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.

10 License