A constant-time sampler for close-to-uniform bitsliced ternary vectors

Pierre Karpman

A constant-time sampler for close-to-uniform bitsliced ternary vectors

Pierre Karpman

pierre.karpman@univ-grenoble-alpes.fr

Université Grenoble Alpes, Grenoble, France

September 15, 2022

Abstract

In this note we describe an algorithm for sampling close-to-uniform random vectors of $\mathbb{F}_3^{64}$ stored in "bitsliced" representation. This algorithm can be implemented in a "constant-time" fashion at some cost and benefits heavily from *bit manipulation instructions*. We propose three main instantiations that respectively use 256, 384 and 512 uniform random bits to sample vectors whose statistical distance to uniform is upper-bounded by $2^{-31.4}$, $2^{-71.9}$ and $2^{-141.6}$.

It appears that the algorithm described in this note has a performance inferior to a well-implemented constant-time change-of-basis algorithm followed by conversion to a bitsliced format. Since the latter also use the minimum amount of randomness for a given distance to uniform, there seems to be no setting where the results from this note have any particular interest.

1 Preliminaries

Our algorithm was designed from the beginning to be efficiently bitslicable and to benefit from *bit manipulation instructions* "BMI" extensions. We define here the bitsliced representation and the BMI instructions we use.

1.1 Bitsliced representation of $\mathbb{F}_3^{64}$

We use a specific bitsliced representation introduced by Boothby and Bradshaw [BB09], although it would be straightforward to adapt the algorithm to an alternative. In this representation, a vector $\boldsymbol{a} \in \mathbb{F}_3^{64}$ is stored as two 64-bit words `a0`, `a1` s.t. the i^{th} bits of `a0` and `a1` encode the i^{th} coordinate $\boldsymbol{a}_i \in \{0, 1, 2\}$ of $\boldsymbol{a}$ as $0 \hookrightarrow (0, 0)$, $1 \hookrightarrow (1, 0)$, $2 \hookrightarrow (1, 1)$.

1.2 BMI instructions

We use the following x86 BMI instructions,* for which we give both the mnemonic and the typical intrinsic name through which it is accessible in C:

*Technically, those instructions come from the POPCNT, BMI and BMI2 extension sets.

— `popcnt`/`_popcnt64`: computes the population count (or weight) of its unique operand, *i.e.* the number of bits set to one.

— `andn`/`_andn_u64()`: computes the bitwise and of the second operand with the negated first operand, *i.e.* `_andn_u64(a, b)` gives the same result as `(~a) & b`. This instruction is particularly useful in our context since the '1' bits of the expression `_andn_u64(a0, a1)` are by definition at the indices i s.t. the i^{th} bits $(\texttt{a0}_i, \texttt{a1}_i)$ of `a0` and `a1` are equal to $(0, 1)$, *i.e.* do not form a valid encoding of an element of $\mathbb{F}_3$.

— `pext`/`_pext_u64`: extracts the bits of the first operand for which the corresponding bit of the second "mask" operand is one. The extracted bits are returned contiguously in the low bits of the result, preserving their relative order; the unused upper bits of the result (if any) are set to zero.

— `pdep`/`_pdep_u64`: "deposits" the contiguous low bits of the first operand into the positions where the second "mask" operand is one, preserving their relative order; all the other bits of the result (if any) are set to zero.

One may remark that `pext` and `pdep` are "pseudo-inverses" of each other, in that for all `a`, `m`, and —letting w be the weight of `m`— `c` the mask with its w low bits set to one, one has:

— `_pext_u64(_pdep_u64(a, m), m)` = `a & c`

— `_pdep_u64(_pext_u64(a, m), m)` = `a & m`

2 The algorithm

2.1 Parallel rejection sampling

The starting point for our algorithm is a simple bit-parallel rejection sampler: initialise two 64-bit words `a0`, `a1` to zero and a mask to the all-one word; draw two uniform 64-bit words; XOR one each to `a0` and `a1` at the bit positions were the mask is one; recompute the mask so that it indicates the bit positions where $(\texttt{a0}, \texttt{a1})$ is not a valid encoding of an element of $\mathbb{F}_3$; repeat from the second step until the mask is all-zero. A possible C implementation of this algorithm is given in Figure 1.

```
1    a0 =  0ULL;
2    a1 =  0ULL;
3    rt = ~0ULL;
4    do
5    {
6      a0 ^= (rand64() & rt);
7      a1 ^= (rand64() & rt);
8      rt  = _andn_u64(a0, a1);
9    } while (rt != 0);
```

Figure 1: A bit-parallel rejection sampler.

This simple algorithm samples uniform bitsliced representations of vectors of $\mathbb{F}_3^{64}$ (the proof is standard and omitted) and has an unbounded running time. One drawback of this algorithm is that it is not parcimonious in its use of randomness, since only an

exponentially-decreasing minority of the sampled bits will ever be used after the first iteration. This might not be a huge penalty in a non-cryptographic context where statistically-good "uniform" random bits are rather cheap to generate and constant-time implementations irrelevant, but a cryptographic constant-time implementation may mandate a (fixed number of a) few dozen iterations[†] and thus the sampling of several thousand uniform random bits of cryptographic quality.

2.2 Parallel rejection sampling with recycling

An obvious improvement to the algorithm of Figure 1 in terms of randomness usage is to "recycle" the unused random bits. We propose to do this in the following way: the first two iterations are identical to the ones of Figure 1, except that at the end of the second iteration one stores all the random bits that were not used (if any) for future use; from the third iteration on, if there are more pairs of stored bits than the number of unset pairs in $(\texttt{a0}, \texttt{a1})$ no new random bits are sampled and the stored bits are used instead, otherwise new bits are sampled and the unused ones are stored for future use; one proceeds thusly until all pairs in $(\texttt{a0}, \texttt{a1})$ have been set.

This algorithm again samples uniform bitsliced representations of vectors of $\mathbb{F}_3^{64}$ and again has an unbounded running time. While it is much more economical than the one of Figure 1 in terms of randomness usage, its recycling is not "total" since some unused random bits may still be discarded. However, this will not be the case any more for the constant-time variants that we will describe next. From an implementation perspective, the main feature of this algorithm is that the storage (resp. the usage) of unused bits can be efficiently implemented using `pext` (resp. `pdep`) instructions. A possible C implementation is given in Figure 2.

The algorithm of Figure 2 can be made constant-time in a straightforward way: one simply needs to fix *a priori* how many (pairs of) random words to use in total and then perform a fixed number of iterations between every fresh sampling. However, it seems easier (cf. Section 3) to upper-bound the statistical distance of the resulting distribution to uniform when the number of iterations between two fresh samplings is the maximum possible —*i.e.* 64— and there is thus some incentive to keep the (fixed) number of samplings as small as possible. We give in Figure 3 a C implementation of a constant-time variant of the algorithm of Figure 2 that only uses two fresh samples.

This algorithm has a constant running time and uses exactly 256 random bits, but its output is not uniformly distributed since there is a non-zero probability that some coordinates of the returned vector will be forced to 0 in the last two lines. Using the techniques developped in Section 3, one may upper-bound the distance to uniform of its output distribution by $2^{-31.4}$. Using more samplings by repeating lines 4–22 allows to (significantly) decrease the distance to uniform, but as pointed above and as is obvious from the code, this implies a large number of iterations of lines 14–21. An alternative which converges slower to the uniform distribution but that improves instruction-level parallelism is to "double" the samplings and use four bits instead of two to try sampling a uniform element of $\mathbb{F}_3$, so that this now only fails with probability 2^{-4} instead of 2^{-2}. Doubling both samplings in the above gives an algorithm requiring 512 bits whose distance to uniform is upper-bounded by $2^{-141.6}$, while doubling only one (either the first or the second) gives an algorithm requiring 384 bits whose distance to uniform is upper-bounded by $2^{-71.9}$.

[†]Informally, we need the number of iterations to be s.t. the probability p that at least one pair of bits of `a0` and `a1` is $(0, 1)$ is small. For one pair the probability after N iterations is 2^{-2N}, and so from the union bound it is at most 2^{-2N+6} for the full vector. A sufficient (and in fact mostly necessary) number of iterations for, say, $p \leq 2^{-64}$ is then $N = 35$.

```
a0   =  0ULL; a1   =  0ULL;
rs0  =  0ULL; rs1  =  0ULL; rt   = ~0ULL;
wrtt =  64; wrt  =  64; wrs  =  64; nw   =  1;
do
{
  if (nw) // need fresh 64-bit random words
  {
    rs0 = rand64();
    rs1 = rand64();
    wrs = 64; // #fresh pairs
  }
  a0  ^= (rs0 & rt);
  a1  ^= (rs1 & rt);
  wrs -= wrt; // #pairs available for recycling @ next round
  rtt  = _andn_u64(a[0], a[1]);
  wrtt = _popcnt64(rtt);
  if (rtt && (wrtt <= wrs)) // recycling possible
  {
    if (nw) // random bits to recycle are scattered thru rs's
    {
      rrs0 = _pext_u64(rs0, ~rt);
      rrs1 = _pext_u64(rs1, ~rt);
      nw   = 0;
    }
    else // random bits ready to be recycled have been gathered in
     ↪  rrs's already, just drop used ones
    {
      rrs0 >>= wrt;
      rrs1 >>= wrt;
    }
    rs0 = _pdep_u64(rrs0, rtt); // deposits recycled bits
    rs1 = _pdep_u64(rrs1, rtt); // where they're needed
  }
  else
  {
    nw = 1;
  }
  rt  = rtt;
  wrt = wrtt;
} while (rt != 0);
```

Figure 2: A bit-parallel rejection sampler with recycling.

3 Statistical analysis

3.1 Distance to uniform for one sample

We now wish to upper-bound the statistical distance to uniform of the output distributions of the algorithm of Figure 3 and its variants. That is, we wish to compute an upper-bound

```
a0   = rand64();
a1   = rand64();
rt   = _andn_u64(a0, a1);
rs0  = rand64();
rs1  = rand64();
a0  ^= (rs0 & rt);
a1  ^= (rs1 & rt);
rrs0 = _pext_u64(rs0, ~rt);
rrs1 = _pext_u64(rs1, ~rt);
rt   = _andn_u64(a0, a1);
wrt  = _popcnt64(rt);
for (int i = 0; i < 63; i++)
{
  rs0    = _pdep_u64(rrs0, rt);
  rs1    = _pdep_u64(rrs1, rt);
  a0    ^= rs0;
  a1    ^= rs1;
  rrs0 >>= wrt;
  rrs1 >>= wrt;
  rt     = _andn_u64(a0, a1);
  wrt    = _popcnt64(rt);
}
a0 &= ~rt;
a1 &= ~rt;
```

Figure 3: A bit-parallel rejection sampler with recycling, constant-time.

for:

$$\Delta(\mathcal{U},\mathcal{R}) := \frac{1}{2}\sum_{\boldsymbol{a}\in\mathbb{F}_3^{64}} |\mathcal{U}(\boldsymbol{a}) - \mathcal{R}(\boldsymbol{a})| \tag{1}$$

Where $\mathcal{R}(\boldsymbol{a})$ (resp. $\mathcal{U}(\boldsymbol{a}) = 3^{-64}$) stands for the probability that $\boldsymbol{a}$ is sampled from the algorithm under study (resp. the uniform distribution). We will do this in a standard way by computing "good" and "bad" terms $p^G \leq 3^{-64}$ and $p^B_* \geq 0$ s.t. $\forall\, \boldsymbol{a} \in \mathbb{F}_3^{64}$, $\mathcal{R}(\boldsymbol{a}) = p^G + p^B_{\boldsymbol{a}}$. Letting $\beta = \sum_{\boldsymbol{a}\in\mathbb{F}_3^{64}} p^B_{\boldsymbol{a}}$ and applying the triangular inequality then allows to rewrite (1) and bound it as:

$$\frac{1}{2}\sum_{\boldsymbol{a}\in\mathbb{F}_3^{64}} \left|3^{-64} - p^G - p^B_{\boldsymbol{a}}\right| \leq \frac{1}{2}\sum_{\boldsymbol{a}\in\mathbb{F}_3^{64}} \left|3^{-64} - p^G\right| + \frac{1}{2}\sum_{\boldsymbol{a}\in\mathbb{F}_3^{64}} \left|p^B_{\boldsymbol{a}}\right| = \frac{1 - 3^{64}p^G + \beta}{2} \tag{2}$$

In the following, let S denote the algorithm under study, $\mathfrak{C}$ the set of possible values for the random bits (or "coins") used in one of its run, and $S(C)$, $C \in \mathfrak{C}$ the output of S when its coins take the value C. It follows that $\mathcal{R}(\boldsymbol{a}) = \Pr[S(C) = \boldsymbol{a} : C \twoheadleftarrow \mathfrak{C}]$, where the probability is computed over the uniform sample $C \twoheadleftarrow \mathfrak{C}$ of C from $\mathfrak{C}$.

In order to compute p^G and p^B_*, we partition $\mathfrak{C}$ into two non-overlapping good and bad sets $\mathfrak{C}^G$ and $\mathfrak{C}^B$ s.t. for all $\boldsymbol{a} \in \mathbb{F}_3^{64}$, $\Pr[S(C) = \boldsymbol{a} : C \twoheadleftarrow \mathfrak{C}^G] = 3^{-64}$, *i.e.* the distribution of S conditioned on the coins coming from the good set $\mathfrak{C}^G$ is uniform. It will then be possible to take $p^G = \frac{\#\mathfrak{C}^G}{\#\mathfrak{C}} 3^{-64}$ and $\beta = \frac{\#\mathfrak{C}^{\mathfrak{B}}}{\#\mathfrak{C}}$.

Applying this strategy to the algorithm of Figure 3, we have $\mathfrak{C} = \{0,1\}^{256}$ and a natural choice for $\mathfrak{C}^G$ is the set of coins s.t. no coordinate of the output is arbitrarily set to zero in the last two lines. It is easy to see that such a set satisfies the defining property

of a good set by noticing that the output of the algorithm for coins belonging to this set is the *same* as the uniform rejection sampler of Figure 2. Alternatively, the statement may be shown directly by noting that for each $C \in \mathfrak{C}^G$ one can identify 64 pairs of bits that determine each the value of one coordinate of the output vector; each of these pairs may independently take three different values while still resulting in coins belonging to the good set, and this defines an equivalence relation for the elements of $\mathfrak{C}^G$; finally, each equivalence class has size 3^{64} and is in bijection with $\mathbb{F}_3^{64}$ through S.

We will now address the computation of p^G (or rather $p^G 3^{64}$) from which β can be readily recovered. From the above this can mostly be done by counting the number of equivalence classes of $\mathfrak{C}^G$, with the nuance that each class also needs to be weighted by its probability of occurrence. A class is fully characterised by which coins (abstracting the "pair of bits" from above so that all variants of the algorithm of Figure 3 can be treated similarly) determine which coordinates of the output. We can think of the coins as being used in sequence, with a minimum of 64 being necessary[‡] and a maximum allowed of 128. We thus need to compute for each additionnal "budget" $b \in [\![1, 64]\!]$: 1) the number of classes; 2) the probability of occurrence of such a class.

We start with 2), which is easier. To accommodate for the variant of the algorithm of Figure 3 that uses a different number of bits for the two samplings (and hence coins of different "currencies"), we additionally count the number h of "bad" coins (that do not determine a coordinate from the output) specifically coming from the first 64 ones. Then letting q (resp. r) denote the probability that a coin from the first (resp. last) 64 ones is good, the probability of occurrence of a given class is (by symmetry) fully characterised by b and h and equal to $q^{64-h}(1-q)^h r^h (1-r)^{b-h}$.

Now to compute 1), given again b, h, the number of equivalence classes is equal to the product of the number of ways of choosing h bad coins from the first 64 ones times the number of ways of choosing $h-1$ good coins from $b-1$ ones. The reason for the two "-1" in the second term comes from the fact that for a fixed b, an equivalence class is indeed fully determined by the position of the first 63 good coins, which are to be found amond the first $b-1$ ones. Alternatively this term can be thought of and computed as the number of paths $\mathrm{P}(b,h)$ on a two-dimensional discrete grid that start from $(h,0)$, end on $(0,*)$ and have area b; one then has $\mathrm{P}(b,h) = \sum_{i=0}^{h} \mathrm{P}(b-h,i)\binom{h}{i}$ with the recursion initialised from $\mathrm{P}(u,u) = \mathrm{P}(u \geq 1, 1) = 1$ and $\mathrm{P}(u < v, v) = 0$. One may indeed check that $\mathrm{P}(b,h) = \binom{b-1}{h-1}$.

Summing up, $p^G 3^{64} = \frac{\#\mathfrak{C}^G}{\#\mathfrak{C}}$ is equal to the probability that $C \in \mathfrak{C}^G$ when picked uniformly at random, which is equal to the number of good equivalence classes weighted by the probability that one draws from them:

$$\frac{\#\mathfrak{C}^G}{\#\mathfrak{C}} = q^{64} + \sum_{h=1}^{64} \sum_{b=h}^{64} \binom{64}{h} \binom{b-1}{h-1} q^{64-h}(1-q)^h r^h (1-r)^{b-h} \tag{3}$$

To analyse the algorithm from Figure 3 and its variants, one then only needs to fix $\#\mathfrak{C}$, q and r in (3) and then use (2), which is straightforward and leads to the claimed results.

Tightness of the bound. The bound from (2) is not tight, since it is equivalent (up to a negligible quantity) to assuming that all bad samples reinforce a single outcome $\boldsymbol{a}$, which is not what the algorithm from Figure 3 does.[§] It is also certainly possible to improve it by using the symmetry of the outcomes reinforced by bad samples to derive bounds on p_*^B.

[‡] Defining one trivial equivalence class.

[§] One could easily modifiy the algorithm to make it implement this worst case and thus get a tight bound with no effort, but there is no real point in doing that.

In order to check how tight the current bound is, we computed the exact distance from an exhaustive enumeration of $\mathfrak{C}$ for small-dimension instantiations of the algorithm from Figure 3 (without any sample doubled). The results are shown in Table 1 along with the upper-bounds computed from (the immediate generalisations of) (2, 3). As far as those small dimensions are concerned, the gap between the bound and the actual distance seems to be reasonable and does not increase too much with the dimension.

Table 1: Distance to uniform of the algorithm from Figure 3 in reduced dimensions, and the corresponding upper-bounds from (2, 3).

Dimension	$\Delta(\mathcal{U}, \mathcal{R})$	Upper-bound
8	$2^{-8.78}$	$2^{-7.06}$
9	$2^{-9.25}$	$2^{-7.52}$
10	$2^{-9.79}$	$2^{-7.98}$
11	$2^{-10.31}$	$2^{-8.44}$

3.2 Distance to uniform for many samples

In the case where more than one vector of $\mathbb{F}_3^{64}$ is sampled from $\mathcal{R}$, one will be interested in the distance to uniform of the product distribution $\mathcal{R} \otimes \cdots \otimes \mathcal{R}$. Let $\mathcal{R}^D$ denote this distribution for D samples and $\mathcal{U}^D$ the corresponding uniform distribution. Then from the distance properties of Δ one has $\Delta(\mathcal{U}^D, \mathcal{R}^D) \leq D\delta$, where $\delta := \Delta(\mathcal{U}, \mathcal{R})$. This bound is sometimes quite loose, and one may hope instead for better bounds of the form $\Delta(\mathcal{U}^D, \mathcal{R}^D) \leq \sqrt{D}\delta'$ for some δ'. We derived such bounds in two ways: 1) by using a one-sample upper-bound on the relative entropy divergence to uniform $\nabla(\mathcal{U}, \mathcal{R}) := \sum_{\boldsymbol{a} \in \mathbf{F}_3} \mathcal{U}(\boldsymbol{a}) \log(\mathcal{U}(\boldsymbol{a}) / \mathcal{R}(\boldsymbol{a}))$, the fact that $\nabla(\mathcal{U}^D, \mathcal{R}^D) = D\,\nabla(\mathcal{U}, \mathcal{R})$, and its relation to Δ through the inequality $\Delta(\mathcal{U}, \mathcal{R}) \leq \sqrt{1/2\,\nabla(\mathcal{U}, \mathcal{R})}$; 2) Renner's upper-bound which gives $\Delta(\mathcal{U}^D, \mathcal{R}^D) \leq \sqrt{D/(2p^G)}\delta$ [Ren05], where p^G has the same meaning as in the above section. This fails to improve the upper-bounds for the algorithm with distance to uniform upper-bounded by $2^{-31.4}$ and $2^{-71.9}$, but it does moderately for the one upper-bounded by $2^{-141.6}$. In this case, both techniques show that the number of samples needed to make the bound vacuously equal to 1 is about $2^{182.8}$ instead of $2^{141.6}$. Yet even this remains of mostly theoretical interest, since these latter bounds start being better than the former only for the unrealistically large number of samples $D \gtrapprox 2^{100}$.

References

[BB09] Tomas J. Boothby and Robert W. Bradshaw. Bitslicing and the method of four russians over larger finite fields. *CoRR*, abs/0901.1413, 2009.

[Ren05] Renato Renner. On the variational distance of independently repeated experiments. *CoRR*, abs/cs/0509013, 2005.