# Simple Random Sampling: Not Simple

Kellie Ottoboni Ron L. Rivest Philip B. Stark

draft September 21, 2016

**Abstract**

A simple random sample (SRS) of size $k$ from a population of size $n$ is a sample drawn at random in such a way that every subset of $k$ of the $n$ items is equally likely to be selected. The theory of inference from SRSs is fundamental in statistics; many statistical techniques and formulae assume that the data are an SRS. True SRSs are rare; in practice, people tend to draw samples by using pseudo-random number generators (PRNGs) and algorithms that map a set of pseudo-random numbers into a subset of the population. Most statisticians take for granted that the software they use "does the right thing," producing samples that can be treated as if they are SRSs. In fact, the PRNG algorithm and the algorithm for drawing samples using the PRNG matter enormously. Using basic counting principles, we show that some widely used methods cannot generate all SRSs of size $k$. In simulations, we demonstrate that the subsets that they do generate do not have equal frequencies, which introduces bias and makes uncertainty calculations meaningless. We compare the "randomness" and computational efficiency of commonly-used PRNGs to a PRNG based on the SHA-256 hash function, which avoids these pitfalls because its state space is countably infinite. We propose several best practices for researchers using PRNGs, including the wide adoption of hash function based PRNGs.

# Contents

# 1 Introduction

Random sampling is one of the most fundamental tools in Statistics. It is used to conduct surveys, including opinion surveys, population surveys like the census, and litigation; to run medical, agricultural, and marketing experiments; quality control in industry and auditing in finance and elections; and countless other purposes. Simple random sampling refers to drawing $k \leq n$ items from a population of $n$ items, in such a way that each of $\binom{n}{k}$ subsets of size $k$ is equally likely. Many standard statistical methods assume that the sample is drawn this way, or allocated between treatment and control groups this way (e.g. $k$ of $n$ subjects are assigned to treatment, and the remaining $n - k$ to control).

- We examine methods for drawing pseudo-random simple random samples. We include a discussion of pseudo-random number generators (PRNGs) and of algorithms used to select samples using PRNGs. Among other things, we find bounds on the number of samples that can be generated using a variate of PRNGs, for a number of sampling algorithms. We also consider how that affects the bias and uncertainty of estimates based on pseudo-random samples rather than on actual simple random samples.

- PRNGs considered include linear congruential generators (LCGs, including RANDU) and the Mersenne Twister. We discuss using cryptographic hash functions to generate PRNs.

- We conclude with recommendations for best practices using PRNGs to generate random samples.

# 2 Background

## 2.1 Definition of "random" numbers

Most computers lack the hardware needed to generate truly random numbers. Instead, they use algorithms called pseudo-random number generators (PRNGs) to generate deterministic sequences from an initial "seed," which generally can be set by the user, for instance to an externally generated random value. Each time a number is generated, the PRNG's "state" changes.

Depending on the quality of the PRNG, the sequences behave more or less like sequences of random numbers. How does one gauge "how random" sequences from a PRNG are? A PRNG yields

sequences of random numbers on the interval $[0, 1]$ or over the binary set $\{0, 1\}$. The sequences output by such PRNGs should be statistically indistinguishable from IID $U(0, 1)$ sequences or IID Bernoulli$(1/2)$ sequences, respectively. The tests for the bit sequence case applies to the $U(0, 1)$ case as well, because there is a one-to-one relationship between sequences of bits and integers. Namely, every sequence of $k$ bits corresponds to an integer on $0, 1, \ldots, 2^k - 1$, and scaling by $2^k$ yields a unique value on $[0, 1)$. (cite L'Ecuyer, Simard)

There are a multitude of ways to test the hypothesis that sequences from a PRNG are indistinguishable from random sequences. TO DO: list types of tests + citations: L'Ecuyer Simard TestU01 (2007), Knuth (1969), Marsaglia DIEHARD tests (1968), NIST tests

- Two notions of uniformity: 1) uniformly distributed within a sequence with a fixed starting state, 2) the set of sequences of length $t$ that can be hit by the PRNG should be uniformly distributed among the set of all sequences of length $t$ – this gets at the SRS problem
- Make sure to define terms like $k$-distributed

## 2.2 Sampling algorithms

A simple random sample of size $k$ from a population of size $n$ is a sample drawn in such a way that each of the $\binom{n}{k}$ possible subsets of size $k$ is equally likely. Given a good source of randomness, there are many ways to operationalize this definition to draw simple random samples.

### 2.2.1 Algorithm PIKK (permute indices and keep $k$)

One basic approach is like shuffling a deck of $n$ cards, then dealing the top $k$: permute the population at random, then take the first $k$ elements of the permutation to be the sample. There are a number of standard ways to generate a random permutation – i.e., to shuffle the deck. If we had a way to generate independent, identically distributed (iid) $U[0, 1]$ random numbers, we could sample $k$ out of $n$ as follows: TO DO: create an algorithm environment Algorithm PIKK

- assign iid $U[0, 1]$ numbers to the $n$ elements of the population
- the sample consists of the $k$ items assigned the smallest random numbers (break ties randomly)

- amounts to generating a random permutation of the population, then taking first $k$.
- if the numbers really are iid, every permutation is equally likely, and it follows that the first $k$ are an SRS requires $n$ random numbers (and sorting)

This algorithm is inefficient: it requires the generation of $n$ random numbers and then a sorting operation.

### 2.2.2 Shuffle

There are more efficient ways to generate a random permutation than assigning a number to each element and sorting. One example is the "Fisher-Yates shuffle" or "Knuth shuffle" (Knuth attributes it to Durstenfeld).

```
Algorithm Fisher-Yates-Knuth-Durstenfeld shuffle (backwards version)
for i=1, ..., n-1:
    J <- random integer uniformly distributed on {i, ..., n}
    (a[J], a[i]) <- (a[i], a[J])
```

This algorithm requires the ability to generate independent random integers on various ranges, but doesn't require sorting. There is also a version suitable for streaming, i.e. generating a random permutation of a list that has an (initially) unknown number of elements.

```
Algorithm Fisher-Yates-Knuth-Durstenfeld shuffle (streaming version)
i <- 0
a = []
while there are records left:
    i <- i+1
    J <- random integer uniformly distributed on {1, ..., i}
    if J < i:
        a[i] <- a[J]
        a[J] <- next record
    else:
        a[i] <- next record
```

*Proof that the streaming shuffle works.* We prove by induction. The base case $i = 1$ is trivial. At stage $i$, $i > 1$, suppose that all $(i-1)!$ permutations of $\{1, 2, \ldots, i-1\}$ are equally likely. $J$ may take on $i-1$ distinct values less than $i$, and the swapping procedure thus yields $(i-1)! \times (i-1)$ possible distinct permutations. If $J = i$, then there is no swap and $(i-1)$ possible distinct permutations. In total, there are $(i-1)! \times (i-1+1) = i!$ possible distinct permutations, and all are equally likely because all values of $J$ and permutations of the first $(i-1)$ elements were equally likely. □

### 2.2.3 Algorithm `Random_Sample` from Cormen et al TO DO: CITE

This is a recursive algorithm that requires only $k$ random integers and does not require sorting. TO DO: TURN INTO PSEUDOCODE; PROVE BY RECURSION THAT THE METHOD WORKS

```
def Random_Sample(n, k, gen=np.random):  # from Cormen et al.
    if k==0:
        return set()
    else:
        S = Random_Sample(n-1, k-1)
        i = gen.randint(1,n)
        if i in S:
            S = S.union([n])
        else:
            S = S.union([i])
    return S
```

## 2.3 Instances of problems

The sampling algorithms in the previous section rely on the ability to generate random integers uniformly distributed on various ranges. In this section, we illustrate several known failures in common software packages.

### 2.3.1 Linear Congruential Generators and RANDU

- Linear congruential generators (LCGs) have the form $X_{n+1} = (aX_n + c) \mod m$, for a modulus $m$, multiplier $a$, and an additive parameter $c$. The behavior of LCGs is well-understood from fundamental number theory. There are three criteria used to choose good values of

the parameters of a LCG: the LCG should generate the full period of length $m$, the full period sequence $X_1, \ldots, X_{m-1}$ should be statistically indistinguishable from random, and the multiplication and modulus operations should be able to be implemented efficiently in bitwise arithmetic (TO DO: cite Hornfeck and Harbrecht). For instance, there is theory on which LCGs yield a full period. TO DO: cite theorem

**Theorem 2.1** (Hull-Dobell Theorem)**.** *The period of an LCG is $m$ for all seeds $X_0$ if and only if*

- *$m$ and $c$ are relatively prime,*
- *$a - 1$ is divisible by all prime factors of $m$, and*
- *$a - 1$ is divisible by 4 if $m$ is divisible by 4.*

- For the sake of computational speed, programmers developed LCGs using moduli of the form $m = 2^b$, where $b$ was the integer word size of the computer. Such an LCG violates the first principle of choosing good parameters, as no non-prime $m$ can yield an LCG with a full period TO DO: cite or prove. One particularly nefarious version of this LCG is RANDU, a PRNG promulgated in the 1960s and widely copied TO DO: cite. RANDU takes the form

$$X_{n+1} = 65539 X_n \mod 2^{31}.$$

  It has a relatively short period ($2^{29}$), all its outputs are odd integers, and it fails the spectral test, which studies the lattice structures of LCGs. Triples of values from RANDU fall on 15 planes in 3-dimensional space.

- TO DO: mention RANDU blowing up science, finding erroneous crystallographic structure

- TO DO: mention Marsaglia random numbers fall on the planes

### 2.3.2 Wichmann-Hill (1982) TO DO: CITE and Excel

The Wichmann-Hill PRNG is a sum of three LCGs, used to produce random values on $[0, 1)$. Given three seed values $s_1, s_2, s_3$, the next value $r$ is given by

$$\begin{aligned} s_1 &= 171 s_1 \mod (30269) \\ s_2 &= 172 s_2 \mod (30307) \\ s_3 &= 170 s_3 \mod (30323) \\ r &= (s_1/30269 + s_2/30307 + s_3/30323) \mod (1) \end{aligned}$$

The Wichmann-Hill generator is generally not considered adequate for statistics, but was (nominally) the PRNG in Excel for several generations. Excel did not allow the seed to be set, so analyses were not reproducible. Moreover, the generator in Excel had an implementation bug that persisted for several generations. Excel didn't allow the seed to be set so issues could not be replicated, but users reportedly generated negative numbers on occasion. TO DO: CITE McCullough, B.D., 2008. Microsoft Excel?s ?Not The Wichmann?Hill? random number generators

### 2.3.3 Stata software

Prior to April 2011, the `rnormal()` function in Stata 10 exhibited predictable behavior: 95.1% of the $2^{31}$ values that could be used to seed the PRNG resulted in first and second draws having the same sign. Resetting the seed too frequently, therefore, would cause this behavior to happen more frequently than if the random values came from a true normal distribution. This is a case of "burn-in": some PRNGs require many advances in the state space before exhibiting "random" behavior. Resetting the seed frequently does not allow the PRNG this burn-in period. Stata uses the KISS algorithm to generate PRNs. KISS is a combination of an LCG with three other PRNGs, each with a different period. TO DO: DESCRIBE KISS AND CITE Marsaglia Resetting the seed in Stata updates the seed of the LCG, but not the other three PRNGs, so KISS will exhibit the same "numbers fall in the planes" phenomenon as RANDU and other LCGs. TO DO: CITE Ozier: "Perils of Simulation..."

### 2.3.4 Random integer generation in R

The sampling and permutation algorithms described above require $k$ or $n$ independent random integers uniformly distributed on $\{1, \ldots, m\}$ for varying values of $m$, depending on the algorithm. A standard way to generate a random integer is to start with $X \sim U[0, 1)$ and define $Y \equiv 1 + \lfloor mX \rfloor$. In theory, that's fine. But in practice, $X$ is not really $U[0, 1)$ but instead is derived by normalizing a PRN that's uniform on $w$-bit integers. Then, unless $m$ is a power of 2, the distribution of $Y$ isn't uniform on $\{1, \ldots, m\}$.

**Lemma 2.2.** *For $m < 2^w$, the ratio of the largest to smallest selection probability is, to first order, $1 + m2^{-w}$. (See, e.g., Knuth v2 3.4.1.A.)*

*Proof.* Define $\tilde{X}$ to be a uniform random integer on $\{0, 1, \ldots, 2^w - 1\}$. The selection probability for a particular integer value is

$$\begin{aligned}
\mathbb{P}(Y = y) &= \mathbb{P}(1 + \lfloor mX \rfloor = y) \\
&= \mathbb{P}(y - 1 \leq mX < y) \\
&= \mathbb{P}\left(\tilde{X} < \frac{y2^w}{m}\right) - \mathbb{P}\left(\tilde{X} \leq \frac{(-1)y2^w}{m}\right) \\
&= \mathbb{P}\left(\tilde{X} < \left\lfloor \frac{y2^w}{m} \right\rfloor\right) - \mathbb{P}\left(\tilde{X} \leq \left\lfloor \frac{(y-1)2^w}{m} \right\rfloor\right) \\
&= 2^{-w}\left(k^+(y-1) - k^-(y-1)\right)
\end{aligned}$$

where, for fixed $m$, we define $k^-(i) \equiv \min\{k : k2^{-w} \geq i/m\}$ for all $i$, $k^+(i) \equiv \max\{k : k2^{-w} < i/m\} = k^-(i+1) - 1$ for $i = 0, \ldots, m-1$ and $k^+(m) \equiv 2^w$. The maximum ratio of selection probabilities is

$$\begin{aligned}
\max_{i,j \in \{0,\ldots,m-1\}} \frac{k^+(i) - k^-(i)}{k^+(j) - k^-(j)} &= \frac{\max_{i=0}^{m-1}(k^+(i) - k^-(i))}{\min_{i=0}^{m-1}(k^+(i) - k^-(i))} \\
&= \frac{\max_{i=0}^{m-1}(k^+(i) - k^+(i+1) + 1)}{\min_{i=0}^{m-1}(k^-(i+1) - k^-(i) - 1)} \\
&= \frac{\lceil 2^w/m \rceil + 1}{\lfloor 2^w/m \rfloor - 1}.
\end{aligned}$$

□

TO DO: IS THIS PROOF RIGHT? SEEMS WRONG

For $m = 10^9$ and $w = 32$, $1 + m2^{-w} \approx 1.233$. That could easily matter. In R, one would generally use the function `sample(1:m, k, replace=FALSE)` to draw pseudo-random integers. It seems that `sample()` uses the faulty $1 + \lfloor mX \rfloor$ approach. TO DO: MAKE ABSOLUTE SURE THAT THIS IS WHAT R DOES!! TO DO: ILLUSTRATE WITH A FIGURE

A better way to generate a (pseudo-)random integer on $\{1, \dots m\}$ from a (pseudo-random) $w$-bit integer in practice is as follows:

1. Set $\mu = \log_2(m - 1)$.
2. Define a $w$-bit mask consisting of $\mu$ bits set to 1 and $(w - \mu)$ bits set to zero.
3. Generate a random $w$-bit integer $Y$.
4. Define $y$ to be the bitwise and of $y$ and the mask.
5. If $y \leq m - 1$, output $x = y + 1$; otherwise, return to step 3.

This is how random integers are generated in `numpy` by `numpy.random.randint()`. However, `numpy.random.choice()` does something else that's biased: it finds the closest integer to $mX$.

## 2.4 The right way to generate PRNs

### 2.4.1 xorShift

### 2.4.2 Mersenne Twister

### 2.4.3 Methods based on cryptographic hash functions

# 3 Results

## 3.1 Possibility Bounds using the Pidgeon-Hole Principle

We now consider whether, in principle, a particular PRNG combined with a particular sampling algorithm could generate an SRS of size $k$ from a population of size $n$. We also consider whether a particular PRNG combined with an "optimal" sampling algorithm that minimized the number of random bits required to generate samples, rather than "wasting" random bits, could generate an SRS of size $k$ from a population of size $n$.

**Lemma 3.1** (One output per state)**.** *If an algorithm uses at least one (entire) output of a PRNG, each state of the PRNG produces at most one distinct output of the algorithm.*

For instance, an algorithm for drawing a sample might "consume" more than one state of the PRNG, but each initial state of the PRNG yields at most one sample.

**Corollary 3.2.** *The number of distinct permutations of a set of $n$ items attainable by assigning a PRN to each element and sorting the result is less than or equal to the number of states of the PRNG.*

See also R. Salfi, 1974, Compstat, 28?35, cited by Knuth, 3.4.2, p. 145: An LCG with modulus $m$ can generate at most $m$ permutations.

**Corollary 3.3.** *The number of distinct samples of size $k$ of a set of $n$ items attainable by a method that uses at least one PRN state to select the sample is less than or equal to the number of states of the PRNG.*

**Proposition 3.4.** *The algorithm that constructs permutations of a set of $n$ objects by assigning a PRN to each, then sorting, cannot construct all permutations of a set of $n$ objects if $n! > S$, where $S$ is the number of states of the PRNG. In particular, such an algorithm cannot construct all permutations of a set of 13 or more objects if the PRNG has $2^{32}$ states or fewer, and cannot construct all permutations of a set of 35 or more objects, if the PRNG has $2^{128}$ states or fewer. Such an algorithm cannot construct all permutations of a set of 2081 or more objects, if the PRNG is the Mersenne Twister.*

*Proof.*

$$13! = 6,227,020,800 > 2^{32} = 4,294,967,296 > 12! = 479,001,600.$$

$$35! = 1.03331479664 \times 10^{40} > 2^{128} \approx 3.402e + 38 > 34! = 2.63130836934 \times 10^{35}.$$

And

$$\ln(2081!) \geq 13823.83 > 13818.582 = \log(32 \times 623),$$

where the first inequality in the last line follows from the Stirling's bound on the factorial:

$$en^{n+1/2}e^{-n} \geq n! \geq \sqrt{2\pi}n^{n+1/2}e^{-n}.$$

□

Therefore, the usual proof that PIKK gives an SRS cannot apply for $n \geq 13$ if the PRNG has a 32-bit state space, nor for $n \geq 35$ if the PRNG has a 128-bit state space, nor for $n \geq 2081$ for the Mersenne Twister. To show that PIKK works for larger $n$ would require close analysis of the frequencies with which the permutations corresponding to distinct samples of size $k$ each occur, showing they are equal. However, if the PRNG can attain all $2^w$ states, then unless $2^w$ is divisible by $\binom{n}{k}$, it's impossible that there are an equal number of permutations corresponding to each sample of size $k$ from $n$. Since $\binom{n}{k}$ is generally not a power of 2, that's the usual situation.

**Proposition 3.5.** *Any algorithm that constructs a subset of $k$ of $n$ objects by using at least one PRNG state per sample cannot construct all $\binom{n}{k}$ subsets if the number of states of the PRNG is less than $\binom{n}{k}$.*

In particular, no PRNG with 32-bit or smaller state space can construct all samples of size 10 from a population of size 47 or more. No PRNG with 128-bit or smaller state space can construct all samples of size 25 from a population of size 366 or more. The Mersenne Twister cannot construct all samples of size 1000 from a population of size $3.8 \times 10^8$ or more.

*Proof.* Use raw calculation for PRNGs with 32-bit and 128-bit state and entropy bounds for the Mersenne Twister. TO DO: FLESH OUT □

$3.8 \times 10^8$ is big, but not in the world of big data.

## 3.2 Theoretical bias

Suppose we take simple random samples of size $k$ from a population of size $n$. Denote the population by $\mathbf{X} = \{X_1, \ldots, X_n\}$ and define the set of all possible samples by $\Omega = \{\omega : \omega \subset \mathbf{X}, |\omega| = k\}$. Let $F$ be the distribution over samples: that is, $F(\omega) = \binom{n}{k}^{-1}$ for all $\omega \in \Omega$. Define $\tilde{F}$ to be the empirical distribution of samples over a period of the PRNG.

Suppose the PRNG never hits a subset $S \subseteq \Omega$. Mathematically,

$$S \equiv \left\{ \omega : \tilde{F}(\omega) = 0, F(\omega) = \binom{n}{k}^{-1} \right\}.$$

Furthermore, the PRNG may not generate the other possible samples on $S^c$ with equal probability. In practice, we know that a PRNG can only hit a certain *number* of samples, but we don't know *which ones*. TO DO: FOR EXAMPLE, A PRNG WITH A 32-BIT STATE SPACE CAN GENERATE

NO MORE THAN $2^{32}$ POSSIBLE SUBSETS OF $k$ OF $n$ OBJECTS, SO FOR $k = 10$ AND $n = 47$, $\nu \geq \binom{47}{10} - 2^{32} = 883,099,455$. If our particular PRNG omits $\nu$ samples, then define

$$\mathcal{S} = \{S : S \subseteq \Omega, |S| \geq \nu\}.$$

$\mathcal{S}$ contains all possible sets of samples that the PRNG could miss.

Define $\mathcal{G} = \{G : G(S) = 0, S \in \mathcal{S}\}$ to be the set of all distributions on $\Omega$ with at least $\nu$ samples with 0 probability. By definition, $\tilde{F} \in \mathcal{G}$. Conceptually, the distribution functions in $\mathcal{G}$ take the mass $F$ assigns to subsets $S$ and redistributes it to other samples. This leads to a simple lower bound on the distance between $F$ and distributions in $\mathcal{G}$.

**Lemma 3.6.** *For any $G \in \mathcal{G}$, $\|F - G\|_1 \geq \frac{2\nu}{\binom{n}{k}}$*

*Proof.* Fix $S \in \mathcal{S}$ and choose some $G \in \mathcal{G}$ such that $G(S) = 0$.

$$\begin{aligned}
\|F - G\|_1 &= \sum_{\omega \in \Omega} |F(\omega) - G(\omega)| \\
&= \sum_{\omega \in S} |F(\omega) - G(\omega)| + \sum_{\omega \in S^c} |F(\omega) - G(\omega)| \\
&= \sum_{\omega \in S} |F(\omega)| + \sum_{\omega \in S^c} |F(\omega) - G(\omega)| \\
&= \frac{|S|}{\binom{n}{k}} + \sum_{\omega \in S^c} |F(\omega) - G(\omega)| \\
&\geq \frac{|S|}{\binom{n}{k}} + \left| \sum_{\omega \in S^c} (F(\omega) - G(\omega)) \right| \\
&= \frac{|S|}{\binom{n}{k}} + \left| \sum_{\omega \in S^c} F(\omega) - 1 \right| \\
&= \frac{2|S|}{\binom{n}{k}}
\end{aligned}$$

Finally, $\|F - G\|_1 \geq \inf_{S \in \mathcal{S}} \frac{2|S|}{\binom{n}{k}} \geq \frac{2\nu}{\binom{n}{k}}$. □

*another way.* Fix $S$ and choose $G \in \mathcal{G}$ such that $G(S) = 0, G(\omega) > 0$ for $\omega \in S^c$.

$$\begin{aligned}
\|F - G\|_1 &= \sum_{\omega \in \Omega} |F(\omega) - G(\omega)| \\
&= \sum_{\omega \in S} |F(\omega) - G(\omega)| + \sum_{\omega \in S^c} |F(\omega) - G(\omega)| \\
&= \sum_{\omega \in S} |F(\omega)| + \sum_{\omega \in S^c} |F(\omega) - G(\omega)| \\
&= \frac{|S|}{\binom{n}{k}} + \sum_{\omega \in S^c} |F(\omega) - (F(\omega) + \varepsilon_\omega)|
\end{aligned}$$

where $\varepsilon_\omega \in [-\binom{n}{k}^{-1}, 1 - \binom{n}{k}^{-1}]$ and $\sum_{\omega \in S^c} \varepsilon_\omega = \sum_{\omega \in S} F(\omega) = \frac{|S|}{\binom{n}{k}}$. NB this must be the case to ensure that $\sum_\omega G(\omega) = 1$, since

$$\sum_\omega G(\omega) = \sum_{\omega \in S^c} G(\omega) = \sum_{\omega \in S^c} F(\omega) + \varepsilon_\omega = \sum_{\omega \in S^c} F(\omega) + \sum_{\omega \in S} F(\omega) = 1.$$

Therefore,

$$\begin{aligned}
\|F - G\|_1 &= \frac{|S|}{\binom{n}{k}} + \sum_{\omega \in S^c} |\varepsilon_\omega| \\
&= \frac{|S|}{\binom{n}{k}} + \sum_{\omega \in S} |F(\omega) \\
&= \frac{2|S|}{\binom{n}{k}}
\end{aligned}$$

□

TO DO: RELATE THIS BACK TO TEST STATISTICS, E.G. THE MEAN THEN FOR ANY BOUNDED FUNCTION $\psi : \Omega \to \mathbb{R}$ AND FOR ANY $G \in \mathcal{G}$,

$$\left| \int \psi dG - \int \psi dF \right| \leq \|F - G\|_1 \|\psi\|_\infty$$

### 3.3 Simulations

# 4 Hash-function based RNGs

# 5 Discussion

### 5.1 Best Practices

# 6 Conclusions