Regulating the Pace of von Neumann Correctors

Houda Ferradi[1], Rémi Géraud[1,2], Diana Maimuţ[1], David Naccache[1], and Amaury de Wargny[2]

[1] École normale supérieure
45 rue d'Ulm, F-75230 Paris CEDEX 05, France
given_name.family_name@ens.fr
[2] Ingenico Group
28-32 Boulevard de Grenelle, 75015 Paris, France
given_name.family_name@ingenico.com

Abstract. In a celebrated paper published in 1951, von Neumann presented a simple procedure allowing to correct the bias of random sources. This device outputs bits at irregular intervals. However, cryptographic hardware is usually synchronous.

This paper proposes a new building block called Pace Regulator, inserted between the randomness consumer and the von Neumann regulator to streamline the pace of random bits.

In a celebrated paper published in 1951 [1], von Neumann presented a simple procedure allowing to correct the bias of random sources. Consider a biased binary source $\mathcal{S}$ emitting 1s with probability p and 0s with probability $1-p$. A von Neumann corrector $\mathcal{C}$ queries $\mathcal{S}$ twice to obtain two bits a, b until $a \neq b$. When $a \neq b$ the corrector outputs a.

Because $\mathcal{S}$ is biased, $\Pr[ab = 11] = p^2$ and $\Pr[ab = 00] = (1-p)^2$, but $\Pr[ab = 01] = \Pr[ab = 10] = p(1-p)$. Hence $\mathcal{C}$ emits 0s and 1s with equal probability.

Cryptographic hardware is usually synchronous. Algorithms such as stream ciphers, block ciphers or even modular multipliers usually run in a number of clock cycles which is independent of the operands' values. Feeding such HDL blocks with the inherently irregular output of $\mathcal{C}$ frequently proves tricky[3].

This paper proposes a new building block called Pace Regulator (denoted $\mathcal{R}$). $\mathcal{R}$ is inserted between the randomness consumer $\mathcal{F}$ and $\mathcal{C}$ to regulate the pace at which random bits reach $\mathcal{F}$ (Figure 1).

[3] A similar problem is met when RSA primes must be injected into mobile devices on an assembly line. Because the time taken to generate a prime is variable, optimizing a key injection chain is not straightforward.

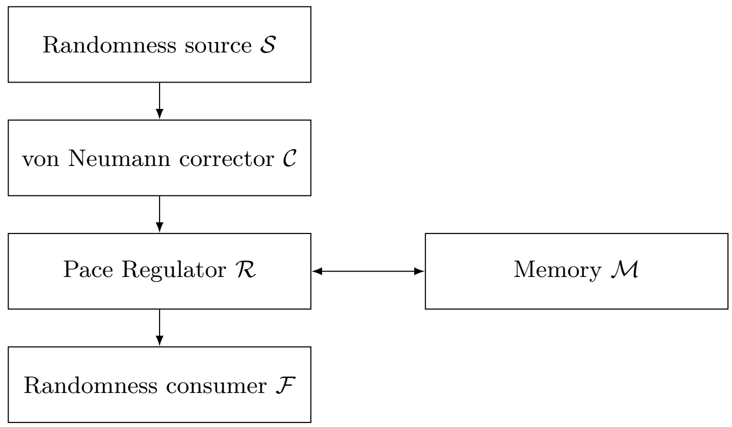

Fig. 1. Source correction and regulation.

1 Model and Assumptions

In all generality we have at one end of a chain a generator $\mathcal{G}$ (here, $\mathcal{G} = \mathcal{S} \circ \mathcal{C}$) that outputs a stream of objects, continuously but at a varying rate. Objects are denoted by $a_1, a_2, \ldots$. At the other end, there is a client $\mathcal{F}$ that we wish to feed objects in a timely fashion, *i.e.* at a near-constant rate.

We wish to design a state machine $\mathcal{R}$ that sits between $\mathcal{G}$ and $\mathcal{F}$, and turns the erratic output of $\mathcal{G}$ into a tame inflow for $\mathcal{F}$. To this end, $\mathcal{R}$ may employ a temporary limited storage $\mathcal{M}$. The setting is illustrated in Figure 2.

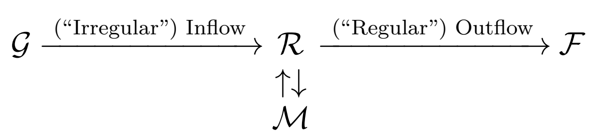

Fig. 2. Problem: Design $\mathcal{R}$ so that the outflow from $\mathcal{R}$ to $\mathcal{F}$ is as smooth as possible, despite the outflow from $\mathcal{G}$ being variable.

The output rate of $\mathcal{G}$ is governed by a probability distribution: an a_i is emitted every t time units, where t is a random variable with probability distribution T.

We make the following important assumptions:

(H1) T is compactly supported, *i.e.* there exists a maximum possible waiting time $t_{\max}$ and a minimum waiting time $t_{\min}$ which we know.
(H2) The a_is produced by $\mathcal{G}$ do not expire, their order does not matter, and they can be stored in $\mathcal{M}$ indefinitely if needed. Hence we can think of $\mathcal{M}$ as a stack of size m.
(H3) Interaction between $\mathcal{R}$ and $\mathcal{M}$ is much faster than waiting times and can for all practical purposes be considered instantaneous.

2 Generic Regulator Description

Informally, the idea behind the regulator concept is that we can use $\mathcal{M}$ to store some a_js, which we may later insert between $\mathcal{G}$'s outputs if $\mathcal{G}$ takes "too long". We cannot store infinitely many objects, and conversely we cannot fill $\mathcal{G}$'s gaps if $\mathcal{M}$ is depleted. Therefore we must determine when to store objects we receive, and when to emit stored objects.

Mathematically, let $\mu > 0$ be some pivot value to be determined later. We assume that $\mathcal{R}$ maintains a timer, so that we know the time t_i elapsed between the emission of a_{i-1} and a_i. We then treat a_is as follows:

- $t_i < \mu$: a_i is "early". Store a_i in $\mathcal{M}$ for later use.
- $t_i = \mu$: a_i is "timely". Output a_i immediately to $\mathcal{F}$.
- If μ time units have elapsed, and still no a_i has been received from $\mathcal{G}$ ("late"), we fetch an a_j from $\mathcal{M}$, send a_j to $\mathcal{F}$, and act as if a_j were just received (*i.e.* a_i is given μ additional time units to arrive: $t_i \leftarrow t_i - \mu$).

Therefore if μ is properly chosen, so that $\mathcal{M}$ never overflows and is never empty, $\mathcal{R}$ outputs one a_i every μ.

Furthermore, we wish $\mathcal{R}$ to be as simple as possible, and in this work consider that $\mathcal{R}$ is an event-driven state machine having access to the following primitives:

- $\mathsf{Push}(a)$ pushes a on the stack $\mathcal{M}$.
- $\mathsf{Pop}()$ pops an object a from the stack and emits it to $\mathcal{F}$.
- $\mathsf{Stack}()$ returns the number of objects currently stored in $\mathcal{M}$.
- $\mathsf{Signal}(t)$ registers an event $\mathsf{EventSig}$ (see below) to be called after time t has elapsed.

The events are:

- $\mathsf{EventSig}$ is called when time t has elapsed since the call of $\mathsf{Signal}(t)$.
- $\mathsf{ObjIn}(a)$ is called when an object is received from $\mathcal{G}$.

– $\mathsf{Setup}(x)$ is called once at initialization.
– $\mathsf{Error}()$ is called upon errors.

$\mathcal{R}$ is inactive between events: it is entirely characterized by describing what it does when events occur.

2.1 Generic Regulator

The regulator's functionality is achieved by using the event handlers described in Algorithms 1 to 3. For the sake of simplicity, we allow $\mathcal{R}$ to use a single global variable s for its operation which we do not count as part of $\mathcal{M}$ in the following discussion. We purposely leave the error handler unspecified.

Algorithm 1 Setup()

```
s ← t_max
Signal(s)
```

Algorithm 2 ObjIn(a)

```
X ← Stack()
if X < |M| then
   Push(a)
else
   Error()
end if
```

Algorithm 3 EventSig

```
X ← Stack()
if 0 < X then
   s ← μ(X)
   Pop()
else
   Error()
end if
Signal(s)
```

The main question thus is how to choose the function μ appropriately. For $\mathcal{M}$ to be neither empty nor overflow in the long term, it is necessary that the number of a_js being stored ("early a_js") and the number of a_js being fetched ("late a_js") balance each other.

3 The Median Regulator

One way to achieve this balance is to choose $\mu(X) = \mu_M$ such that $T(t < \mu_M) = T(t > \mu_M)$, which is exactly the definition of the median. Hence, we can set

$$\mu_M := t_{1/2} = \text{Median}(t) \tag{1}$$

Implementing the generic regulator with this choice of μ yields the *median regulator*. Note that the sample median could be estimated from the data and used here, instead of the theoretical median (if unknown).

Equation (1) is not a *sufficient* condition: it may be that while being zero *on average*, the amount of a_j stored in $\mathcal{M}$ wanders around. Indeed, there is a 1/2 probability to get an early (resp. late) a_i[4], so that the population X_k of $\mathcal{M}$ undergoes a random walk. We have

$$\lim_{k\to\infty} \frac{\mathbb{E}\left(|X_k - \frac{m}{2}|\right)}{\sqrt{k}} = \sqrt{\frac{2}{\pi}} \quad \Rightarrow \quad \left|X_k - \frac{m}{2}\right| \approx \sqrt{k}$$

Therefore, on average, this regulator reaches an error state after receiving $\sqrt{m}$ a_is. $\mathcal{M}$ could be chosen so that $m \approx k^2$ where k is the maximal number of packets that we wish to process. However this limitation is unsatisfactory and we will get rid of it.

4 Memory-Variance Trade-Off: Adaptive Regulators

The key observation is that Equation (1) is not a *necessary condition* either: all that is required is really that $\mathbb{E}(\mu) = t_{1/2}$. Now we may be smarter and adjust the value of μ to the moment's needs. Indeed, if we are about to use too much memory, then decreasing μ would result in more a_js being labelled "late", and we would start emptying $\mathcal{M}$. If on the contrary $\mathcal{M}$ is getting dangerously empty, we may increase μ so that more a_js become "early", and start repopulating $\mathcal{M}$. Note that we may vary μ slowly or quickly over time, this variation being itself irrelevant to the statistical analysis.

Of course, such a strategy incurs a non-zero variance in the outflow, but at this price we may lower the size of $\mathcal{M}$. More precisely, for any given

[4] In other term, we consider that the probability of getting a timely a_i is negligible.

memory capacity $m = |\mathcal{M}|$ and input-time distribution T, we want to construct an $\mathcal{R}$ whose output-time distribution T'_m is such that

$$\begin{aligned} \lim_{m\to\infty} &\operatorname{Var}(T'_m) = 0 \\ \lim_{m\to 0} &\operatorname{Var}(T'_m) = \operatorname{Var}(T) \\ &\operatorname{Var}(T'_m) \leq \operatorname{Var}(T) \end{aligned}$$

This is of course the ideal case and the further question now becomes: How do we modulate μ at any given moment in time, to achieve this?

Let X denote the occupation of $\mathcal{M}$ at a given point in time. If $X = 0$ then we *must* take in new a_is, and we cannot output any more a_js, therefore we have no choice but to set $\mu \leftarrow t_{\max}$. Conversely, if $X = m$ then we must empty the queue and set[5] $\mu \leftarrow t_{\min}$. We already saw that if $X = m/2$ the best choice is the neutral $\mu \leftarrow t_{1/2}$.

We wish to interpolate and describe the function $\mu(X)$ that is such that

$$\mu(0) = t_{\max}, \qquad \mu(m/2) = t_{1/2}, \qquad \mu(m) = t_{\min}$$

There are several ways to do so.

4.1 Lagrange Regulator

Take for instance Lagrange interpolation polynomials: let

$$\begin{aligned} a &= \frac{2}{m^2}\left(t_{\max} + t_{\min} - 2t_{1/2}\right) \\ b &= \frac{1}{m}\left(t_{\max} + 3t_{\min} - 4t_{1/2}\right) \\ c &= t_{\max} \end{aligned}$$

Then we can take

$$\mu_L(X) := aX^2 + bX + c.$$

In the special case where $T = \text{Uniform}(A, 3A)$, we have $\mu_L(X) = (3 - 2X/m)A$.

[5] We do not set $\mu \leftarrow 0$ or any lower value for two reasons: first $\mathcal{R}$ would empty its whole stack immediately, which is not the intended behaviour; and second this makes interpretation and analysis harder.

4.2 Distributional Regulator

The main interest of the Lagrange Regulator is its simplicity. However, there is no reason to consider that the choice of a μ polynomial in X is optimal. Let F_t be the cumulative distribution function $F_t(y) := T(t \leq y)$ and consider its inverse F_t^{-1}. We define the distributional regulator as

$$\mu_D(X) := F_t^{-1}\left(1 - \frac{X}{m}\right).$$

Observe that we have

$$\mu_D(0) = F_t^{-1}(1) = t_{\max}$$
$$\mu_D\left(\frac{m}{2}\right) = F_t^{-1}\left(\frac{1}{2}\right) = t_{1/2}$$
$$\mu_D(m) = F_t^{-1}(0) = t_{\min}$$

This regulator assumes a complete knowledge of t's distribution, but provides the best results in the sense that it minimizes the variance of $\mathcal{R}$'s output. In the special case where $T = \text{Uniform}(A, 3A)$, we have

$$\mu_D(X) := F_t^{-1}\left(1 - \frac{X}{m}\right) = A + 2A\left(1 - \frac{X}{m}\right) = \left(3 - 2\frac{X}{m}\right)A = \mu_L(X)$$

that is, we get the exact same result as the Lagrange Regulator.

5 Parameters for the von Neumann Corrector

We can compute exactly the distribution T for the von Neumann corrector if $\mathcal{S}$ outputs one random value every δ units of time. In that case, one couple is generated every 2δ, and this couple has a probability $2p(1-p)$ to be accepted. Each couple is generated independently from others, so that the probability of k successive rejections is $(1 - 2p(1-p))^k$. Let $\epsilon = 2p^2 - 2p + 1$, we have $0 < \epsilon < 1$ and

$$T(2k\delta) = \epsilon^k(1 - \epsilon).$$

Observe that T is *not* compactly supported, as for any $t > 0$ we have $T(t) > 0$. However we can define a cut-off value above which event probability becomes negligible, *i.e.* $T(t) < 2^{-N}$ for some $N \in \mathbb{N}$. This gives

$$k_{\max} = -\frac{N - \log_2(1-\epsilon)}{\log_2(\epsilon)} \Rightarrow t_{\max} = -2\delta\frac{N - \log_2(1-\epsilon)}{\log_2(\epsilon)}$$

the minimum is $t_{\min} = 0$, and the median is computed from the cumulative probability

$$\sum_{k=0}^{n} T(2k\delta) = \sum_{k=0}^{n} \epsilon^k (1-\epsilon) = 1 - \epsilon^{n+1}$$

so that $k_{1/2} = -\frac{1}{\log_2 \epsilon} - 1$, hence

$$t_{1/2} = -2\delta \left(\frac{1}{\log_2 \epsilon} - 1 \right)$$

Example 1. Assume $\delta = 1$ and $N = 80$, we have the following parameters for different biases p:

p	ϵ	$t_{\min}$	$t_{1/2}$	$t_{\max}$
1/2	1/2	0	4	162
1/4	5/8	0	4.95	241
1/32	481/512	0	24.19	1866

6 Experimental Results

To test our regulator we implemented a simulation in Python. The simulation is event-driven: only a_i reception and emission are considered, which allows for an exact solution (in particular, there is no timer involved). a_i generation by $\mathcal{G}$ is simulated by inverse sampling of a given distribution. In the simulation we assume that this distribution is known, and we implement the corresponding Lagrange regulator. The source code is provided in Appendix A.

We choose a certain amount of memory m and run the simulation for $n \gg m$ objects. The output distribution is then measured.

After some warming-up time (which is of the order of $m/2$), the output distribution reaches a steady state peaked around a central value $\mu' \approx \mu$. The variance of this distribution is *much smaller* than the input variance and a larger memory m results in a narrower distribution.

6.1 Uniform Input Distribution

Figure 3 shows the steady-state distribution of a Lagrange regulator applied to a uniform generator. Memory usage X fluctuates around $m/2$. Figure 4 shows the evolution of variance and interquartile range (IQR) as a function of m.

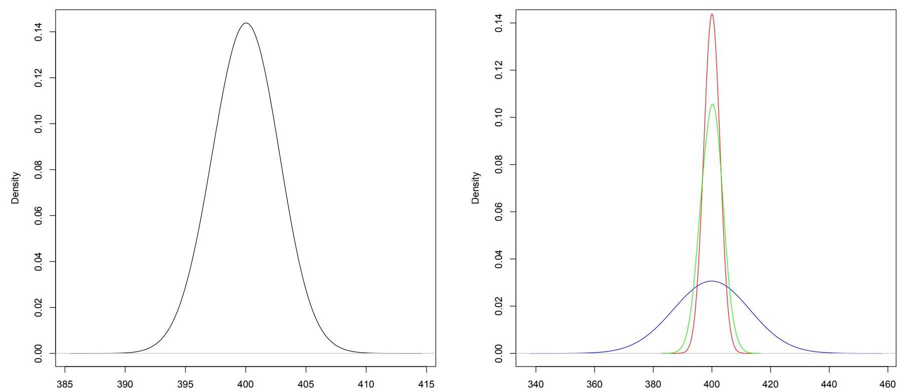

Fig. 3. Left: Steady-state output distribution of a Lagrange regulator, with input distribution $T = \text{Uniform}(200, 600)$ and $m = 1000$. The distribution peaks at $\mu' = 400.0$, and is contained in $[390, 410]$. Compare to the input distribution ($\mu = 400, \sigma = 115.4$). Average memory usage is $500 = m/2$. Right: same thing with $m = 100$ (blue), 500 (green) and 1000 (red).

Statistical dispersion around μ' decreases quickly as m increases: $\log \log \text{IQR}$ decreases almost linearly with m. Both standard deviation and IQR reach a minimum value. IQR decreases faster than standard deviation, which yields a distribution with higher kurtosis as m increases. These observations are consistent across various parameter choices.

6.2 Cut-off Geometric Input Distribution

The output times of the von Neumann corrector follow a geometric distribution (*cf.* Section 5). Since this distribution is *not* compactly supported, we define a cut-off value $t_{\max}$.

We use the `random.geometric` function from `numpy` to automatically generate sequence of appropriately distributed t_is, with a cut-off at 2^{80} for the distributional regulator.

Results are similar to the uniform case, but memory usage is higher on average because of the input distribution's large tail. The cut-off incurs a non-zero (albeit negligible) failure probability, that must be dealt with: When an exceptionally large delay occurs, the degraded operation simply consists in outputting the late object as soon as it arrives.

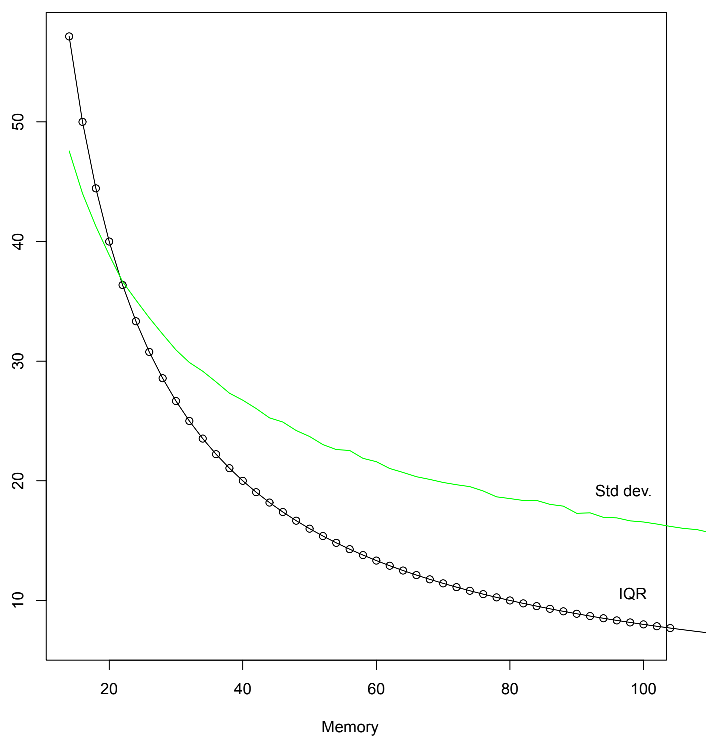

Fig. 4. Steady-state IQR (black, circled) and standard deviation (green) as a function of m, for the same parameter set as Figure 3. Both IQR and standard deviation get lower for larger values of m, and reach a minimal nonzero value; $\log \log \mathrm{IQR}$ is almost linear, with a slope of -0.008.

References

1. von Neumann, J.: Various techniques used in connection with random digits. National Bureau of Standards Applied Math Series 12, 36–38 (1951)

A Source Code

```
import random
import numpy
import math

# Available memory
m = 1000
```

```python
# Distributional regulator
mu_D = lambda x:icdf(1 - x/m)

def unif_icdf(x):
    """
    Inverse cumulative distribution function for the uniform distribution
    U(a, b)
    """
    a = 200
    b = 600
    return a + x * (b-a)

def generator(icdf):
    """
    Generates a random number distributed according to the provided
    inverse cumulative distribution function
    """
    return icdf(random.random())

def simulate(input_events, mu):
    """
    Simulation
        input_events: relative time between input events
        mu: regulator
    """

    # Stack population
    X = 0

    # Current input
    k = 0

    # Lookahead
    j = 0

    # Absolute time for output events
    M = []

    # Compute absolute time for input events
    T = [0] * len(input_events)
    for k in range(1, len(input_events)):
        T[k] = T[k-1] + input_events[k]

    # Push the first input
    X += 1

    while k+j+1 < len(input_events) - 1:
        j = 0
        # Push all early inputs on stack
        while T[k+j+1] < M[-1]:
            X+=1
            j+=1

        # Memory overflow or underflow
        if X < 0 or X >= m:
            print("Error! Memory under- or overflow: X = %s"%X)
```

```python
            return []

        # Pop and emit an object
        M.append(M[-1] + mu(X))
        X -= 1
        k += j

    return M

def save_data(ret, filename):
    """
    Saves data ret to the file 'filename'
    """
    f = open(filename,'w')
    f.write("%s\n"%("mu"))
    for u in ret:
        a = u
        f.write("%s\n"%(a))

    f.close()

def generate_events(N,icdf):
    """
    Generates N events distributed according to the provided
    inverse cumulative distribution function
    """
    return [generator(icdf) for i in range(N)]


events = generate_events(100000,unif_icdf)
ret = simulate(events, mu_D(unif_icdf))
save_data(ret, 'output.txt')
```