Generació de Nombres Aleatòris Mitjançant el Mètode de la Funció de Probabilitat Inversa

Pau Soler Valadés
09-01-2025
Simulació, MESIO UPC-UB

1. Introducció

En el següent document s'explica la implementació de codi Zig per generar nombres aleatòris seguint les distribucions Uniforme, Exponencial, Gamma, Weibull, i Cauchy. Addicionalment es contesten les preguntes proposades de relacions de les distribucions entre elles, així com el mètode de la CDF inversa per generar nombres aleatòris.

2. Per què Zig?

Zig és un llenguatge fortament tipat amb gestor de memòria manual que intenta ser el que hauria de ser el llenguatge de programació `C` si s'hagués dissenyat al segle XXI. Destaca per evitar el control de flux implicit i reconeixent que els ordinadors moders són sistemes altament complexos que s'han de tractar amb simplicitat. A nivell pràctic ofereix més garanties de seguretat que C (bound checking on array iteration, millor gestió de Strings, many-item pointers) amb un rendiment igual.

Així i tot, la decisió del llenguatge ha estat molt influenciada a nivell personal, ja que he fet diversos projectes amb ell i em sento molt còmode amb ell.

3. Generació de Nombres pel Mètode de la CFD inversa.

El mètode empra la composició de dues funcions: la funció de densitat d'una distribució uniforme $U(0,1)$ juntament amb la inversa de la funció de probabilitat F de la distribució que es vulgui generar. Una variable aleatòria $U \sim U(0,1)$ té la densitat f_U següent:

$$f_U(x) = 1I_{\{a \leq x \leq b\}}(x)$$

i està definida en $f_U : \mathbb{R} \mapsto [0,1]$, i una CDF seguint una variable aleatòria X està definida a $F_X : \mathbb{R} \mapsto [0,1]$, i la seva inversa per tant a $F_X^{-1} : [0,1] \mapsto \mathbb{R}$. Aleshores, la seva composició

$$F_X^{-1} \circ f_U \sim X$$

$$F_X^{-1} \circ f_U : \mathbb{R} \overset{f_U}{\longmapsto} [0,1] \overset{F_X^{-1}}{\longmapsto} \mathbb{R}$$

No només és possible, sinó que és una CDF de la variable X. Això és possible ja que U genera els nombres uniformement entre l'interval $[0,1]$, fent que la imatge de la composició es comporti com la distrbució de la variable aleatòria caldria esperar.

Per tant, la única limitació d'aquest mètode és evident: necessitem coneixer la inversa de la CDF analíticament o bé aproximar-la numèricament.

3.1. Distribució Uniforme

El següent codi mostra com es genera un valor aleatòri seguint una variable uniforme en Zig.

```
fn runif(comptime T: type, a: T, b: T, rng: *Random) !T {
    if ((T != f32) and (T != f64)){
        return RNGError.NotAFloat;
    }
```

```
    if (b < a) {
        return RNGError.InvalidRange;
    }
    // scale if needed
    return a + (b - a) * rng.float(T);
}
```

Zig ofereix una interfície `Random`, la qual implementa diversos mètodes per generar nombres aleatòris; `rng.float(T)` retorna un nombre aleatori entre 0 i 1 del tipus `T`, que per igualar-ho a `R` és un `f64`, un nombre en coma flotant de precisió doble.

L'algorisme per a generar els nombres aleatòris però és temendament important, i Zig dona diverses opcions incloent-hi algoritmes garantint aleatorietat criptograficament segura. L'algorsime escollit en aquest cas és el `Xorshiro256` (TODO POSAR LINK), que empra l'operació XOR juntament amb shifts i rights. Recuperaré aquest terme a la secció de rendiment TODO citar.

Degut als requeriments del programa i voler imitar en la mesura del possible el comportament d'`R`, s'ha assumit que no es coneixia la longitud del vector demanat, que s'ha de proporcionar a l'executable. D'aquesta manera, és necessari demanar memòria dinàmica per a assegurar-se sempre tenir espai suficient per a qualsevol tamany de mostra demanat.

```
fn runifSampleAlloc(allocator: *Allocator, n: u32, comptime T: type, a: T, b: T, rng:
*Random) !ArrayList(T) {
    var sample: ArrayList(T) = .empty;
    // reservem l'espai d'avantmà per millorar rendiment
    try sample.ensureTotalCapacity(allocator.*, n);
    for (0..n) |_| {
        const u = try runif(T, a, b, rng);
        _ = try sample.append(allocator.*, u);
    }

    return sample;
}
```

Zig separa l'adquisició de memòria dinàmica en objectes implementats a la seva llibreria estàndard anomentars `Allocators`, i `ArrayList` seria la implementació de la classe `std::Vector` de `C++`, és a dir, una llista de tamany ampliable.

El codi és molt directe: cridar la funció que genera un nombre aleatòri d'una distribució uniforme entre [0,1] i guardar-lo a la llista corresponent. La figura Figure 1 mostra l'histograma dels valors resultants d'aquest codi.

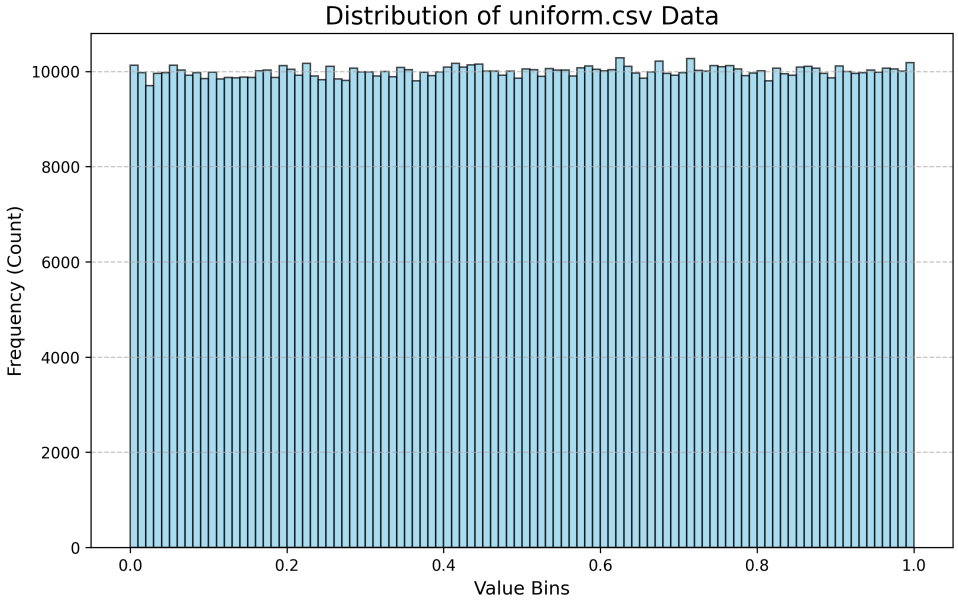

Figure 1: Histograma amb 100 bins d'una mostra de 1E6 elements d'una $U(0,1)$.

3.2. Distribució Exponencial

La distribució exponencial modelitza la probabilitat entre punts en un procés de punts de poisson, és a dir, un procés on els esdeveniments ocorren continuament i independent de manera constant de mitjana. [1] La principal propietat que té és la falta de memòria: la probabilitat d'un esdeveniment a un temps t depen només de l'estat anterior del sistema, i no de tots els que ha passat fins ara.

3.2.1. Relacions amb Distribucions

Sigui una variable aleatoria $U \sim U(0,1)$ i $X \sim \text{Exp}(\lambda) : \lambda > 0$. Aleshores, la relació entre U i X ve donada per la següent expressió:

$X = -\frac{\ln(U)}{\lambda}$

A més a més, X també es pot veure seguint una distribució $\text{Gamma}\left(1, \frac{1}{\lambda}\right)$. A més a més, la suma de k variables aleatòries independents X és una $\text{Gamma}\left(k, \frac{1}{\lambda}\right)$, és a dir:

$$\sum_{i=1}^{k} X \sim \text{Gamma}\left(k, \frac{1}{\lambda}\right)$$

[2]

3.2.2. Codi i Mètode de la CDF inversa

Sigui $X \sim \text{Exp}(\lambda)$. La CDF de $F_X(x) = 1 - e^{-\lambda}$ v, i la seva inversa és

$$F_X^{-1}(u) = -\lambda \log(u)$$

La funció en Zig és la següent:

```
fn rexpSampleAlloc(allocator: *Allocator, n: u32, comptime T: type, lambda: T, rng:
*Random) !ArrayList(T) {
    var sample: ArrayList(T) = .empty;
    try sample.ensureTotalCapacity(allocator.*, n);

    for (0..n) |_| {
        const u = try runif(T, 0, 1, rng);
        const e = lambda*(-@log(u));
```

```
        _ = try sample.append(allocator.*, e);
    }

    return sample;
}
```

És essencialment la mateixa funció que la uniforme però amb la inversa de la CDF just després. A la figura Figure 2 es pot veure el resultat d'aquest codi.

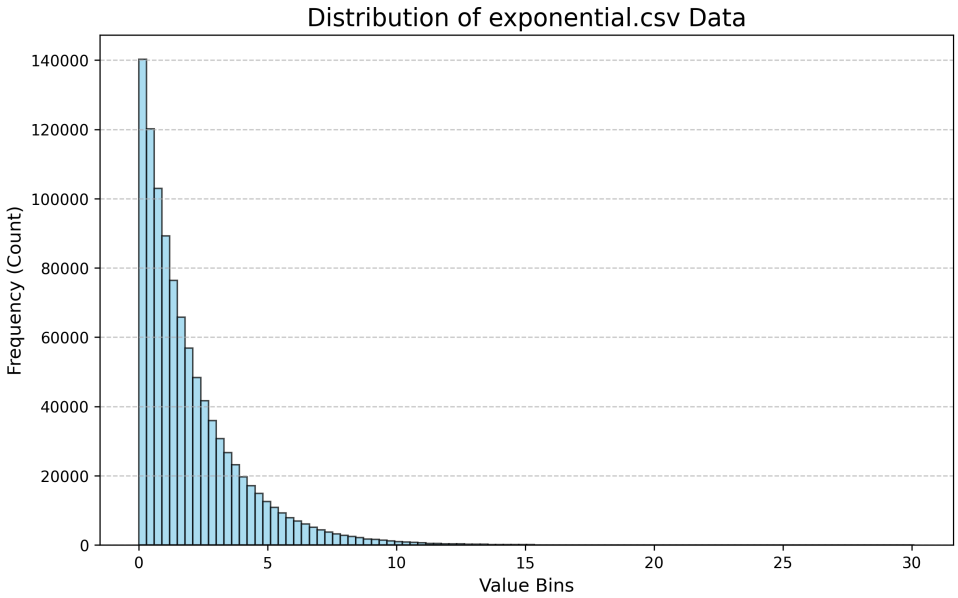

Figure 2: Histograma amb 100 bins d'una mostra de 1E6 elements d'una $\text{Exp}(2)$.

3.3. Distribució Weibull

La distribució Weibull modelitza el temps que succeeix entre dos esdeveniments o el temps de fallada d'un esdeveniment. És molt flexible, i s'utilitza en anàlisi de supervivència, automoció i molts altres dominis.

Està definida per dos paràmetres, d'escala i de forma de la distribució.

3.3.1. Relacions amb Altres Distribucions

Sigui $X \sim \text{Weibull}(k, \lambda)$.

1. $X \sim \text{Weibull}\left(1, \frac{1}{\lambda}\right) = \text{Exp}(\lambda)$
2. $\lambda(-\ln(U))^{\frac{1}{k}} \sim \text{Weibull}(k, \lambda)$
3. $X \sim \text{Weibull}\left(2, \sqrt{2}\beta\right) = \text{Rayleigh}(\beta)$

3.3.2. Codi i Mètode de la CDF Inversa

Sigui $X \sim \text{Weibull}(k, \lambda)$, la seva CDF és $F_{X(x)} = 1 - e^{\left(-\frac{x}{\lambda}\right)^k}$ i aleshores té com a inversa:

$$F_X^{-1}(x) = \lambda - \ln(x)^{\frac{1}{k}}$$

La funció en Zig és la següent:

```
fn rwbSampleAlloc(allocator: *Allocator, n: u32, comptime T: type, lambda: T, k: T,
rng: *Random) !ArrayList(T) {
    var sample: ArrayList(T) = .empty;
    try sample.ensureTotalCapacity(allocator.*, n);

    for (0..n) |_| {
```

```
        const u = try runif(T, 0, 1, rng);
        const w = lambda*(pow(T, -@log(u), 1.0 / k));
        _ = try sample.append(allocator.*, w);
    }

    return sample;
}
```

La figura Figure 3 mostra l'histograma resultant d'aquest codi.

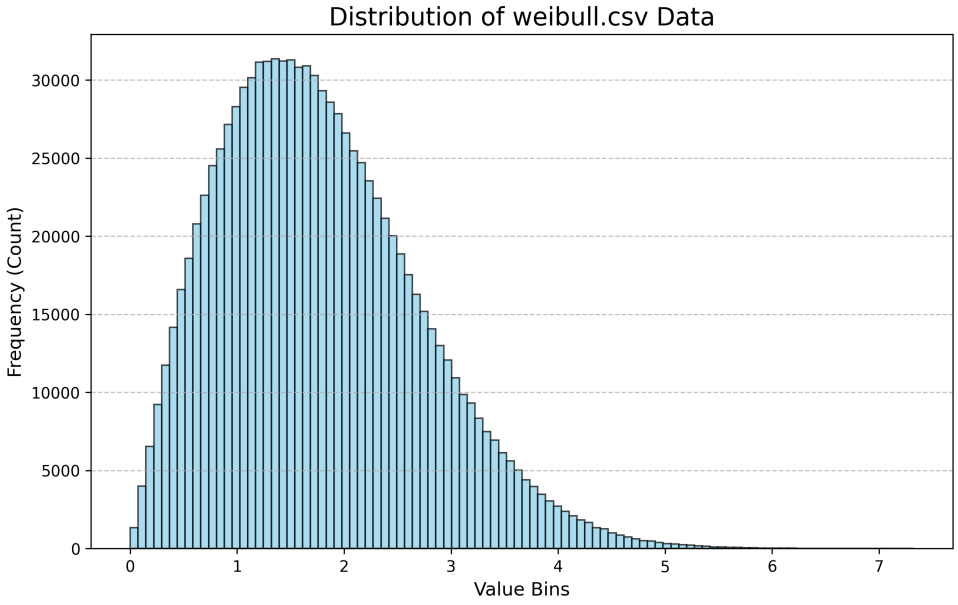

Figure 3: Histograma amb 100 bins d'una mostra de 1E6 elements d'una $\text{Weibull}(2, 2)$.

3.4. Distirbució Gamma

La distribució Gamma és una familia de distribucions que conté molts casos particulars d'altres distribucions importants. A més a més, és molt versàtil i s'empra en diversos camps, com l'econometria, estadística Bayesiana i anàlisi de supervivència [3].

3.4.1. Relacions amb Altres Distribucions

Sigui $X \sim \text{Gamma}(\alpha, \theta)$, té les següents relacions amb altres distribucions:

1. Siguin $X_1, X_2, ..., X_n$ n variables aleatòries independents i idènticament distribuïdes (i.i.d) que segueixen una distribució exponencial amb paràmetre de taxa λ, llavors

$$\sum_i X_i \sim \text{Gamma}(n, \lambda)$$

   on n és el paràmetre de forma i λ és la taxa, i

$$|(X) = \frac{1}{n} \sum_i X_i \sim \text{Gamma}(n, n\lambda)$$

   .

2. Si $X \sim \text{Gamma}(1, \lambda)$ (en la parametrització forma-taxa), llavors X té una distribució exponencial amb paràmetre de taxa λ. En la parametrització forma-escala, $X \sim \text{Gamma}(1, \theta)$ té una distribució exponencial amb paràmetre de taxa $\frac{1}{\theta}$.

3. Combinant els dos fets anteriors, es veu clarament que es pot escriure una Gamma com la suma d'exponencials si $k \in \mathbb{Z}$.

4. Si $X \sim \text{Gamma}(\frac{\nu}{2}, 2)$ (en la parametrització forma-escala), llavors X és idèntica a $\chi^2(\nu)$, la distribució khi quadrat amb ν graus de llibertat. Recíprocament, si $Q \sim \chi^2(\nu)$ i c és una constant positiva, llavors $cQ \sim \text{Gamma}(\frac{\nu}{2}, 2c)$ [3].

3.4.2. Codi i Mètode de la CDF Inversa

En el cas de la Gamma, si es vol emprar el mètode de la CDF inversa, no és possible fer-ho sobre la CDF, ja que aquesta és:

$$F_X(x) = \frac{1}{\Gamma(\alpha)} \theta^\alpha x^{\alpha-1} e^{-\frac{x}{\theta}}$$

On $\Gamma(\alpha) = \int_0^\infty t^{\alpha-1} e^{-x} dx$. La funció Γ no té una inversa explícita, així que per programar-ho s'hauria d'utilitzar algun mètode per aproximar la integració o trobar el zeros adients. En canvi, amb només la limitació de que el paràmetre $\alpha \in \mathbb{Z}$, podem utilitzar el fet que, si $X \sim \text{Exp}(\lambda)$, $\sum_{i=0}^n X_i \sim \text{Gamma}(n, \lambda)$, tal com s'ha mencionat a l'apartat de la distribució exponencial. Per tant, per implementar la inversa, usem la suma de funcions inverses de la exponencial.

$$F_G^{-1}(x) = \sum_{i=0}^n F_X^{-1}(x) = -\sum_{i=0}^n \ln(x)$$

A continuació es mostra el codi de la implementació.

```
fn rgammaSampleAlloc(allocator: *Allocator, n: u32, comptime T: type, shape: u32,
scale: T, rng: *Random) !ArrayList(T) {

    var sample: ArrayList(T) = .empty;
    try sample.ensureTotalCapacity(allocator.*, n);

    for (0..n) |_| {
        var g: T = 0.0;
        for (0..shape) |_| {  // sum of exponentials
            const u = try runif(T, 0, 1, rng);
            g -= @log(u);
        }
        g *= scale;
        _ = try sample.append(allocator.*, g);
    }

    return sample;
}
```

Tot i compartir una estructura similar amb els altres codis mostrats fins ara, el doble sumartori és la suma de la inversa de la CDF de l'exponencial. La figura Figure 4 mostra l'histograma del resultat del codi anterior.

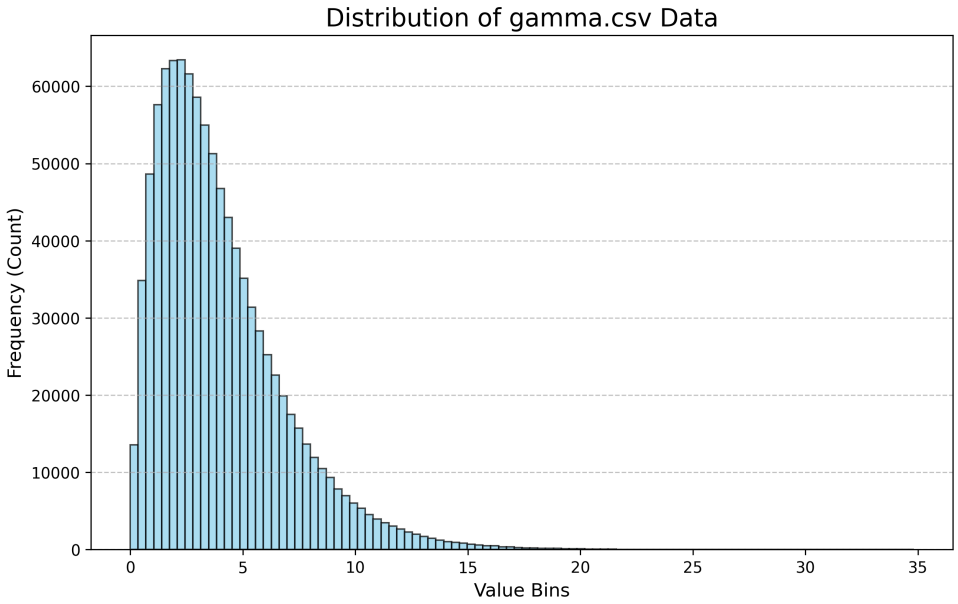

Figure 4: Histograma amb 100 bins d'una mostra de 1E6 elements d'una Weibull$(2, 2)$.

3.5. Distribució Cauchy

La distribució de Cauchy sorgeix del quocient de dividir dues variables aleatòries normals amb mitjana zero, i té dos paràmetres, x_0 i γ. En física és molt coneguda per modelitzar l'intercepte d'un raig sortint de (x_0, γ) amb un angle uniformement distribuït, i en altres camps s'acostuma a emprar com una normal amb observacions atípiques, ja que la seva característica principal són les seves cues més gruixudes [4], [5].

3.5.1. Relacions amb Altres Distribucions

Sigui $Z \sim \text{Cauchy}(x_0, \gamma)$.

1. Si X i Y són dues variables aleatòries independents que segueixen una **distribució normal estàndard** $(N(0, 1))$, aleshores el seu quocient $Z = \frac{X}{Y}$ segueix una **distribució de Cauchy estàndard** amb paràmetres $x_0 = 0$ i $\gamma = 1$.
2. La distribució de Cauchy estàndard és un cas especial de la **distribució t de Student** amb un grau de llibertat. És a dir, $\text{Cauchy}(0, 1) \equiv t(\nu = 1)$.

3.5.2. Codi i Mètode de la CDF Inversa

Si U és una variable aleatòria que segueix una distribució uniforme en l'interval $(0, 1)$, aleshores la variable $X = x_0 + \gamma \tan\left(\pi\left(U - \frac{1}{2}\right)\right)$ segueix una distribució de Cauchy amb paràmetres x_0 i γ. Aquesta és la base del mètode de la transformada inversa per a la generació de mostres, i com s'ha implementat en el codi següent.

```
fn rcauchySampleAlloc(allocator: *Allocator, n: u32, comptime T: type, gamma: T, x0:
T, rng: *Random) !ArrayList(T) {
    var sample: ArrayList(T) = .empty;
    try sample.ensureTotalCapacity(allocator.*, n);

    for (0..n) |_| {
        const u = try runif(T, 0, 1, rng);
        const e = x0 + gamma * @tan(std.math.pi*(u - 0.5));
        _ = try sample.append(allocator.*, e);
    }
```

```
    return sample;
}
```

El codi segueix la mateixa estructura que els altres tres. La figura Figure 5 mostra el resultat del codi, amb la peculiaritat de que perque es veiés tota la distribució s'ha hagut de reduir el nombre d'elements a només 1000, ja que a més elements hi havia, desapareixien les cues ràpidament.

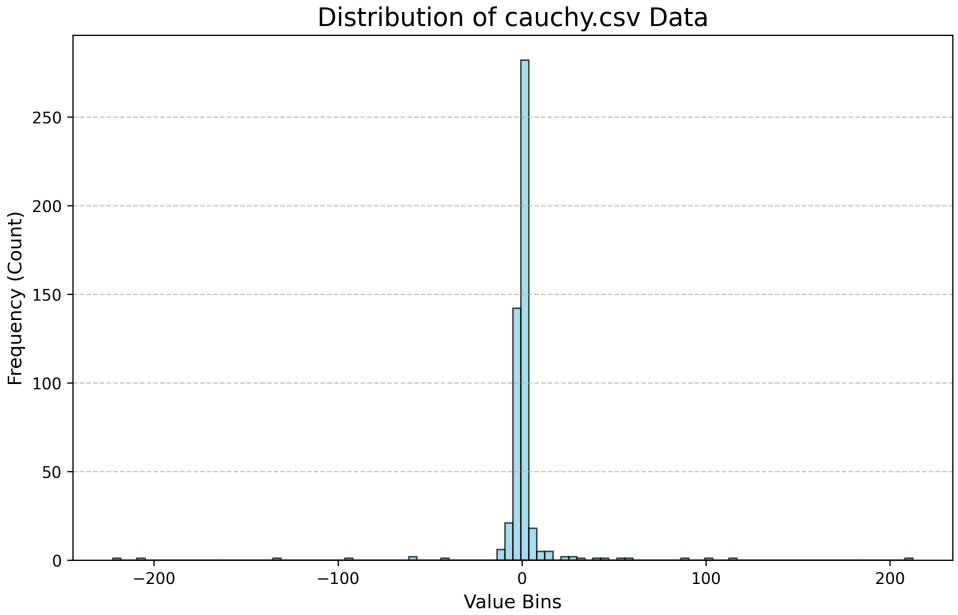

Figure 5: Histograma amb 100 bins d'una mostra de 1000 elements d'una $\text{Cauchy}(0, 1)$.

4. Comparacions

Ja que s'ha decidit no utilitzar R per la realització d'aquest petit exercici, s'ha considerat apropiat llistar els avantatges que suposa utilitzar un llenguatge compilat amb gestió de memòria manual en comptes d'un llenguatge interpretat.

La implementació de Zig té diverses limitacions per a fer una comparació justa amb R:

- Tipus de precisió doble: R per defecte utilitza el tipus numèric, que en el cas que l'arquitectura el tingui disponible, es tradueix com a tipus `f64` (precisió doble) i encara que la implementació rebi un tipus genèric `T`, totes les execucions s'han fet amb `f64`.
- Utilització de memòria dinàmica: al ser un llenguatge compilat, es podria canviar el valor `n` al propi codi, recompilar el programa i tornar a executar, prescindint de la utilització de la memòria heap. Això acceleraria el programa sobretot per mostres petites (per exemple, en una arquitectura x86 amb linux s'acostuma reservar 8MB de memòria estàtica per programa. Assumint l'ús de `f64` i reservant 1KB per les instruccions i codi, podriem arribar a generar arrays de fins a aproximadament $10^6 \approx \frac{7.9*1024^2}{8}$), però degut a que R sempre ha d'emprar memòria dinàmica ja que és interpetat, la implementació en Zig usa allocadors d'arena i `ArrayList` per emmagatzemar els nombres.

Els principals avantatges de Zig respecte R són exclusivament de rendiment. Tot i tenir certa proficiencia en R, la implementació naïf en Zig ha necessitat unes tres hores, i només la part del codi en R podria haver trigar approxiamadament uns vint minuts. La següent secció compara a nivell de rendiment si val la pena fer l'intercanvi.

4.1. Benchmarking

Un *benchmark* en si mateix ja és una prova molt complicada, i encara ho és més comparant un llenguatge interpretat i un compilat. Els codis candidats han d'implementar la mateixa funcionalitat,

i en la mesura del possible, les mateixes operacions, fent-ho idiomàticament segons el llenguatge. Per reduir la complexitat (i el temps) només s'han realitzat proves sobre la generació de nombres uniformes (`runif`). Els tres codis candidats són:

1. Zig naïf: dues funcions, versió presentada en aquest document fins ara.

```
/// Generate a random number of a uniform distribution in [a,b).
fn runif(comptime T: type, a: T, b: T, rng: *Random) !T {
    if ((T != f32) and (T != f64)){
        return RNGError.NotAFloat;
    }

    if (b < a) {
        return RNGError.InvalidRange;
    }
    // scale if needed
    return a + (b - a) * rng.float(T);
}

/// Generate an sample of a uniform distribution [a,b)
fn runifSampleAlloc(allocator: *Allocator, n: u32, comptime T: type, a: T, b: T, rng:
*Random) !ArrayList(T) {
    var sample: ArrayList(T) = .empty;
    try sample.ensureTotalCapacity(allocator.*, n);

    for (0..n) |_| {
        const u = try runif(T, a, b, rng);
        _ = try sample.append(allocator.*, u);
    }

    return sample;
}
```

2. Zig vectoritzat: versió utilitzant SIMD (Single Instruction Multiple Data) per a accelerar els còmputs. Utilitza la classe `@Vector` de Zig per a implementar-ho manualment. La constant `VEC_LEN` depen de l'arquitectura de la CPU i de la capacitat de la màquina en executar instruccions SIMD.

```
/// Generate an sample of a uniform distribution [a,b)
fn runifSampleAllocSIMD(allocator: *Allocator, n: u32, comptime T: type, a: T, b: T,
rng: *Random) !ArrayList(T) {
    var sample: ArrayList(T) = .empty;
    try sample.ensureTotalCapacity(allocator.*, n);

    var rng_buff: [VEC_LEN]T = undefined;
    const min_vec: @Vector(VEC_LEN, T) = @splat(a);
    const range_vec: @Vector(VEC_LEN, T) = @splat(b - a);

    var pos: usize = 0;
    var left: usize = n;

    while (left > 0) {
        for (0..VEC_LEN) |i| {
            const u = rng.float(T);
            rng_buff[i] = u;
        }
        const u_vec: @Vector(VEC_LEN, T) = rng_buff[0..VEC_LEN].*;
        const res_vec = min_vec + range_vec*u_vec;
```

```
            const res_arr: [VEC_LEN]T = res_vec;
            try sample.appendSlice(allocator.*, &res_arr);

            pos += VEC_LEN;
            left -= VEC_LEN;
        }

        if (left < VEC_LEN) {
            // si l'input és menor que el tamany, no facis SIMD
            for (0..left) |_| {
                const u = try runif(T, a,b, rng);
                _ = try sample.append(allocator.*, u);
            }
        }

        return sample;
}
```

3. R: crida a `runif`.

```
    results <- numeric(n)
    results <- runif(n, min_val, max_val)
```

Cada codi s'ha executat amb una $n = 10^6$ 100 vegades, i a la taula Table 1 es reporta el temps mitjà en milisegons que han trigat les execucions. Cal destacar que un *benchmark* més complet inclouria diverses n i reportaria una gràfica amb tots els valors, ja que l'escalabilitat és un factor molt important al codi. Degut a les limitacions de temps i que l'objectiu d'aquest document no és fer un *benchmark*, sinó donar alguna intuïció sobre rendiment, s'ha considerat suficient.

Implementation	Average Time (ms)	PRNG Algorithm
R (runif)	42.11 ms	Mersenne-Twister
Zig (Naïf)	20.30 ms	Xoshiro256++
Zig (Vectorized/SIMD)	18.94 ms	Xoshiro256++

Table 1: Mitjanes en milisegons dels tres programes generant un vector de 10^6 nombres aleatòris sota una $U(0,1) m = 100$

Sorprenentment, R és approximadament el doble de lent que les versions de Zig, cosa que em sorprèn. El resultat esperat hagués estat una velocitat similar entre tots o que R guanyés als programes de Zig, ja que `runif` és una funció en C precompilada, s'interactua amb cap part "lenta" de l'intèrpret. Algunes de les raons que podrien explicar aquesta discrepància són les següents:

1. Algorisme d'RNG: l'alorisme Xoshiro és molt ràpid, i probablement no tingui les mateixes garanties aleatòries que Mersenne-Twister. Malauradament Zig no disposava de generadors de nombres aleatòris tan sofisticats com els d'R i s'ha hagut de fer la comparació amb aquest, ja que les limitacions de temps i *scope* no m'han permès implementar el generador d'R en Zig.
2. Tamany: R és un llenguatge complex, i aquest benchmark està comparant un codi d'unes 300 línies de Zig approximadament amb un llenguatge sencer amb moltes més funcionalitats. Tot i ser una comparació per defecte desequilibrada, és la més justa que es pot fer en aquest entorn.

Amb la implementació de l'algorisme d'R en Zig es podria fer una implementació més fidedigne a el que fa R i millorar la generació vectoritzada dels valors de la versió SIMD, que evitaria copiar els valors aleatòris a un buffer per a casterjar-los a un `@Vector`. Cal notar també que la diferència entre les dues versions de Zig és minsa, cosa que ens indica que el compilador ha estat capaç d'aplicar vectorització a

l'execució normal del codi fent-lo competitiu amb la versió manual, que és més llarga i requereix una reestructuració de la lògica del programa per a poder-ho aplicar.

Queda clar doncs que l'esorç extra d'implementar el codi amb un llenguatge de baix nivell no queda compensat per la poca diferència de rendiment que s'obté, a menys que sigui una tasca que s'hagi d'executar moltes vegades i valgui la pena sortir de l'ecosistema d'R per trobar millor rendiment en processos concrets. Així i tot, cal apreciar la bellesa de la implementació en Zig, ja que ofereix molt més contol sobre les dades i el rendiment.

5. Descripció de l'Entrega

S'entreguen juntament amb aquesta memòria els següents fitxers:

- `rng.zig`: el codi font que conté la lectura del fitxer `parametres.txt`, la generació de nombres aleatòris i l'escriptura d'ells a un fitxer en format SSV (space separated value).
- `parametres.txt`: fitxer separat per espais que ha de contenir tantes linies com distribucions (max 5) i els parametres d'aquestes.
- `plot_distributions.py`: script de python que llegeix els CSV, crea els historgrames i els guarda com a PNG.
- `rng_vectorized.zig`: conté la implementació en SIMD de la generació de valors aleatòris de la distribució uniforme juntament amb els *benchmarks* de la implementació vectoritzada i la implementació naïf.
- `Rbenchmark.R`: script d'R que executa el paquet `microbenchmark` sobre generar un vector aleatori d'una uniforme entre zero i u.

Bibliography

[1] "Exponential distribution." [Online]. Available: https://en.wikipedia.org/wiki/Exponential_distribution

[2] "Distribució Exponencial." [Online]. Available: https://ca.wikipedia.org/wiki/Distribuci%C3%B3_exponencial

[3] "Gamma Distribution." [Online]. Available: https://en.wikipedia.org/wiki/Gamma_distribution

[4] "Caucy Distribution." [Online]. Available: https://ca.wikipedia.org/wiki/Distribuci%C3%B3_de_Cauchy

[5] "Distribució de Cauchu." [Online]. Available: https://en.wikipedia.org/wiki/Cauchy_distribution