PSTL : Rapport sur ALIAS.

Comparaisons entre l'algorithme de A. J. Walker, implanté par UNURAN, et un nouvel algorithme basé sur les entiers.

MORANDEAU Timothée et VILA Rodrigo,
sous la supervision de GENITRINI Antoine et NAÏMA Mehdi

Sorbonne Université

1 Introduction

Il est bien connu en informatique que la représentation des nombres n'est pas triviale, particulièrement lorsqu'il s'agit de nombres réels. Contrairement aux mathématiques pures où les nombres ont une précision infinie, l'utilisation de nombres flottants en informatique introduit nécessairement des approximations. Ces limitations se manifestent par des comportements contre-intuitifs, comme des égalités numériques qui échouent ou des accumulations d'erreurs d'arrondi.

La méthode des Alias, algorithme classique de génération aléatoire selon une distribution discrète prédéfinie, repose fondamentalement sur des calculs en arithmétique flottante. Cette caractéristique n'est pas simplement le fait de son implémentation dans la bibliothèque UNURAN[1], mais relève de sa conception même - à ce jour, aucune variante aussi efficace n'a été proposée utilisant une arithmétique de précision arbitraire.

Face à ce constat, une question naturelle émerge : serait-il possible de reformuler cet algorithme pour n'utiliser que des entiers ? Et si oui, quels gains en précision, performance ou simplicité pourrait-on en attendre ? L'implémentation de base de cette approche entière nous a été proposée par nos encadrants de projet, que nous avons ensuite enrichie et optimisée au cours de notre travail.

2 Présentation de l'algorithme initial

2.1 Description de l'algorithme de A. J. Walker

Dans les années 1970, A. J. Walker[2] propose l'idée suivante pour pouvoir générer discrètement des nombres, aléatoirement, en suivant une distribution discrète donné. On ferait un premier pré-traitement en ($O(n)$, qui nous donnerait un tableau dans lequel on tirerait aléatoirement un élément en un temps ($O(1)$).
Les cases de ce tableau, d'une taille n contiennent trois informations : un premier élément m, la probabilité p que cet élément tombe, et potentiellement un deuxième élément o si p est plus petit que n.

On peut voir ci-dessous (Figure 1 à 3), une illustration de l'idée génerale de la méthode des ALIAS, proposée par wikipédia[3]. On a en figure 1, le tableau des poids de notre distribution.
Elles sont ensuite réparties dans le tableau pour obtenir la figure 3. Les cases 3 et 5 ont ici un alias. On peut se pencher par exemple sur la case 3 : avec le premier élément (jaune), la probabilité que ce premier élément tombe (ici quelque chose de l'ordre de 50%), et le deuxième élément (rouge).
On tirera ensuite aléatoirement une case du tableau, puis à l'intérieur de la case un des deux éléments.

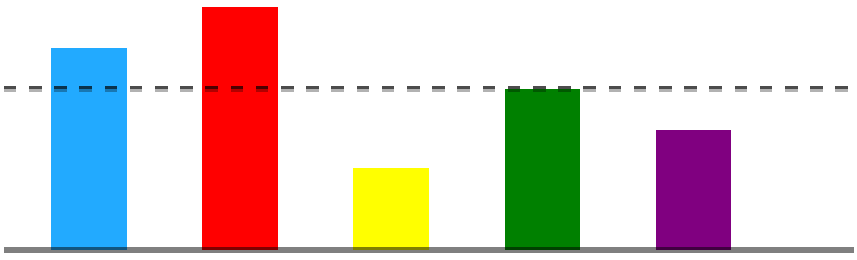

Fig. 1. Tableau initial

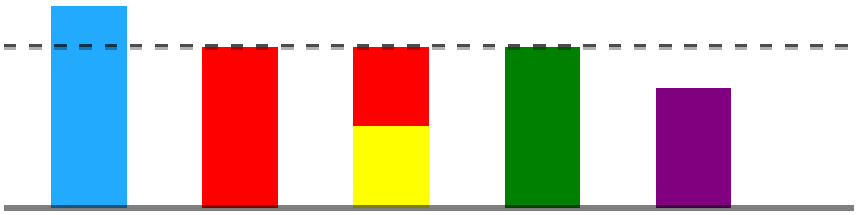

Fig. 2. Calcul...

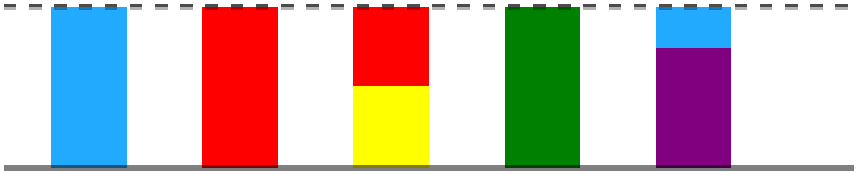

Fig. 3. Tableau final

Passons à un exemple d'exécution pour comprendre plus en détail comment cet algorithme fonctionne.

Exemple d'exécution Prenons un exemple concret avec comme entrée les poids $[3, 4, 5]$ pour illustrer le fonctionnement de l'algorithme :

- **Input**: $PV = [3, 4, 5]$
- **Normalisation**:

$$sum = 3 + 4 + 5 = 12$$
$$ratio = 3/12 = 0.25$$
$$new\_pv = [0.75, 1.0, 1.25]$$

- **Classification**:

  - $poor = [0]$ (index du seul élément avec $new\_pv < 1$)
  - $rich = [1, 2]$ (indexes des éléments avec $new\_pv \geq 1$)
- **Initialisation**: $alias = [\text{None}, 1, 2]$
- **Première itération**:
  - On prend le pauvre ($tmp\_poor = 0$) et le riche ($tmp\_rich = 2$)
  - On complète: $alias[0] = 2$
  - On ajuste: $new\_pv[2] = 1.25 - (1 - 0.75) = 1.0$
- **Résultat final**:
  - $new\_pv = [0.75, 1.0, 1.0]$
  - $alias = [2, 1, 2]$

Cette table permet ensuite une génération aléatoire efficace en temps constant. On tire un indice i entre 0 et la taille de $new\_pv - 1$, puis on prend une probabilité aléatoire p, si $p \leq new\_pv[i]$ alors on retourne i, sinon on retourne $alias[i]$

2.2 L'algorithme

On peut proposer le pseudo-code suivant pour construire cette table.

Algorithm 1 Construction de la table Alias

Require: PV: Liste des probabilités
Ensure: $(prob\_table, alias\_table)$
1: $poor \leftarrow []$
2: $rich \leftarrow []$
3: $n\_pv \leftarrow \text{length}(PV)$
4: $sum \leftarrow \sum PV$
5: $ratio \leftarrow n\_pv/sum$
6: $new\_pv \leftarrow [p \times ratio \text{ for } p \in PV]$
7: $alias \leftarrow [\text{None}] \times n\_pv$
8: **for** $i \in [0, n\_pv)$ **do**
9: **if** $new\_pv[i] \geq 1.0$ **then**
10: $rich.push(i)$ ▷ push: ajoute en fin de liste
11: $alias[i] \leftarrow i$
12: **else**
13: $poor.push(i)$
14: **end if**
15: **end for**
16: **while** $poor \neq []$ **do**
17: $tmp\_poor \leftarrow poor[-1]$
18: $tmp\_rich \leftarrow rich[-1]$
19: $alias[tmp\_poor] \leftarrow tmp\_rich$
20: $new\_pv[tmp\_rich] \leftarrow new\_pv[tmp\_rich] - (1.0 - new\_pv[tmp\_poor])$
21: **if** $new\_pv[tmp\_rich] < 1.0$ **then**
22: $poor.pop()$ ▷ pop: retire le dernier élément de la liste
23: $rich.pop()$
24: $poor.push(tmp\_rich)$
25: **else**
26: $poor.pop()$
27: **end if**
28: **end while**
29: **return** $(new\_pv, alias)$

Cette méthode, utilisé par UNURAN, utilise l'algorithme "Robin des bois" de Marsaglia[4]. Le principe est le suivant : premièrement on normalise les données de notre distribution (l6). Puis on sépare les données entre deux catégories : les riches, ceux dont la valeur normalisée d'apparaître est supérieur ou égale à 1, et les pauvres (ceux qui restent). On va, pour chaque pauvre le compléter avec un riche. Ce riche soit reste riche, soit devient pauvre. Dans ce dernier cas, on le déplace dans la pile des pauvres.

2.3 Problèmes liés à la gestion des flottants

Lors de notre tentative de ré-implémentation de l'algorithme d'échantillonnage par alias utilisé par UNURAN en Python, nous avons été confrontés à un comportement inattendu : certaines distributions parfaitement valides provoquaient des erreurs dans la construction de la table d'alias.
Après analyse, nous avons identifié l'origine du problème dans les différences de gestion des nombres flottants entre les langages C (dans lequel UNURAN est écrit) et Python.

En effet, en Python, les opérations arithmétiques sur des flottants peuvent introduire de légères erreurs d'arrondi, qui suffisent à fausser l'algorithme, en particulier lorsqu'on effectue des soustractions de valeurs très proches. Le test suivant illustre cette différence :

```
print(1.3333333333333333 - 0.6666666666666667)
```

Ce calcul, censé donner $2/3 = 0.666\ldots$, retourne en réalité `0.6666666666666665` en Python, soit une très légère sous-estimation. En comparaison, un langage comme C tronque directement à `0.666667`, ce qui est plus proche du résultat attendu.

Cette différence peut paraître anodine, mais dans le cadre de la construction de la table d'alias, elle a des conséquences concrètes. Dans notre cas, ce léger décalage a provoqué une mauvaise classification d'une probabilité comme étant strictement inférieure à 1, alors qu'elle aurait dû être considérée comme supérieure ou égale. Cela a entraîné une erreur qui a causé l'échec de la construction de la table.

Pour remédier à ce problème, nous avons opté pour l'utilisation de la bibliothèque `fractions.Fraction` de Python, qui permet de travailler avec des nombres rationnels exacts. Grâce à cette approche, les erreurs d'arrondi ont été complètement éliminées et l'algorithme a pu fonctionner comme prévu, en respectant la logique théorique sous-jacente à la méthode d'A.J. Walker.

Cependant, l'implémentation de la méthode d'Alias dans la bibliothèque `UNURAN` traite ces erreurs d'arrondi différemment. Elle repose sur une tolérance explicite fondée sur une constante $FLT\_EPSILON$, utilisée notamment dans des conditions de type:
`if (`$new_pv[i]$ `>= 1. -` $FLT\_EPSILON$`)`
Afin de corriger les dépassements numériques. Cette stratégie permet de compenser les imprécisions des opérations en virgule flottante sans recourir à une arithmétique exacte, au prix d'un léger écart vis-à-vis du modèle théorique.

3 Présentation de l'algorithme Alias avec poids entiers

Algorithm 2 Construction d'une table Alias basée sur des poids entiers

Require: $\bar{S}, \bar{W}$: Listes des objets et de leurs poids entiers
Ensure: $(\bar{T}, cs)$

procedure TABLE_BUILDING($\bar{S}, \bar{W}$)
    # Initialisation
    $n \leftarrow \text{length}(\bar{S})$
    $w \leftarrow \sum \bar{W}$
    $cs \leftarrow w//n$ ▷ Taille des cellules
    $r \leftarrow w \mod cs$
    if $r > 0$ **then**
        $\bar{S}.\text{push}(x_n)$
        $\bar{W}.\text{push}(cs - r)$
        $w \leftarrow w + \bar{W}[-1]$
    end if
    $m \leftarrow w \div cs$
    $\bar{T} \leftarrow [\text{None}] \times m$ ▷ Table d'Alias
    $rich \leftarrow [i \text{ for } i, w \text{ in enumerate}(\bar{W}) \text{ if } w \geq cs]$ ▷ Liste des poids riches
    $poor \leftarrow [i \text{ for } i, w \text{ in enumerate}(\bar{W}) \text{ if } w < cs]$ ▷ Liste des poids pauvres
    $k \leftarrow 0$

    # Corps de l'algorithme
    while *rich* **not empty do**
        $i \leftarrow rich.\text{pop}()$
        $x, w_x \leftarrow \bar{S}[i], \bar{W}[i]$
        if *poor* **not empty then**
            $j \leftarrow poor.\text{pop}()$
            $x', w'_x \leftarrow \bar{S}[j], \bar{W}[j]$
            if $x' \neq x_n$ **then**
                $\bar{T}[k] \leftarrow (w'_x, x', x)$
            else
                $\bar{T}[m-1] \leftarrow (cs - w'_x, x, x')$
                $k \leftarrow k - 1$
            end if
            $\bar{W}[i] \leftarrow \bar{W}[i] - (cs - w'_x)$
        else
            $\bar{T}[k] \leftarrow (cs, x, \text{None})$
            $\bar{W}[i] \leftarrow \bar{W}[i] - cs$
        end if
        if $0 < \bar{W}[i] < cs$ **then**
            $poor.\text{push}(i)$
        else if $\bar{W}[i] \geq cs$ **then**
            $rich.\text{push}(i)$
        end if
        $k \leftarrow k + 1$
    end while
    return $(\bar{T}, cs)$
end procedure

Algorithm 3 Génération aléatoire à partir de la table Alias

Require: $\bar{T}$: Table Alias (liste de triplets (w, x, x'))
Require: cs: Taille des cellules
Ensure: x: Objet généré aléatoirement (jamais x_n)

```
procedure GENERATE(T̄, cs)
    m ← length(T̄)
    total_weight ← m × cs
    last_weight, _, potential_xn ← T̄[−1]
    if potential_xn == x_n then                ▷ Évite la génération de l'objet virtuel
        xn_weight ← cs − last_weight
        total_weight ← total_weight − xn_weight
    end if
    rand_value ← random(0, total_weight − 1)
    index ← rand_value//cs
    w, obj1, obj2 ← T̄[index]
    if (rand_value mod cs) < w then
        return obj1
    else
        return obj2
    end if
end procedure
```

Les algorithmes présentés ci-dessus permettent donc de générer efficacement des échantillons selon une distribution de poids entiers définie. Nous avons fait en sorte que l'algorithme proposé soit au plus proche de celui dans UNURAN. Les manières de dépiler nos piles de riches et de pauvres, par exemple, est identique à ce que fait UNURAN.

3.1 Construction de la Table Alias avec Poids Entiers

Objectif : Construire une table qui, à partir d'une liste d'objets $\bar{S}$ et de leurs poids entiers associés $\bar{W}$, permette de générer rapidement des tirages suivant la distribution initiale. La table Alias contient des informations (typiquement sous forme de triplets) qui indiquent, pour chaque cellule de taille fixe cs, comment combiner le poids d'un objet avec celui d'un autre (alias) pour reproduire la distribution d'origine.

Étapes clés :

1. **Initialisation :**
   - Calcul de la somme totale des poids w et détermination de la taille des cellules cs via la division entière du poids total par le nombre d'objets n.
   - Calcul du reste r (i.e. $w \mod cs$) pour vérifier si la somme totale est un multiple de cs. Si ce n'est pas le cas ($r > 0$), un objet virtuel x_n est ajouté à la liste des objets avec un poids complémentaire ($cs - r$). Cette étape permet de normaliser la distribution.

2. **Création des listes de poids riches et pauvres :**
   - $\mathcal{H}$ est la liste des indices des objets dont le poids est supérieur ou égal à cs (poids « riches »).
   - $\mathcal{L}$ est la liste des indices des objets dont le poids est inférieur à cs (poids « pauvres »).
3. **Remplissage de la table Alias :**
   - En utilisant un compteur k pour indexer les cellules, l'algorithme traite les listes $\mathcal{H}$ et $\mathcal{L}$ en ajoutant les objets à la table d'alias.
   - La cellule k de la table Alias est remplie par un triplet associant un poids et l'objet correspondant ainsi que l'objet eventuel qui complete ce poids.
   - Le poids de l'objet qui complete est réduit, et selon le nouveau poids, il est soit mit dans la liste des pauvres, soit conservé parmi les riches.
4. **Retour de la table Alias :** L'algorithme retourne la table Alias ainsi construite ainsi que la taille cs des cellules, qui seront utilisées dans la phase de tirage.

3.2 Génération Aléatoire à partir de la Table Alias

Objectif : Utiliser la table Alias pour générer aléatoirement un objet, selon la distribution définie par les poids initiaux, tout en évitant le tirage de l'objet virtuel x_n.

Étapes clés :

1. **Préparation :**
   - Détermination du nombre de cellules m dans la table et calcul du poids total comme $m \times cs$.
   - Vérification de la dernière cellule pour éviter la sélection de l'objet virtuel x_n. Si nécessaire, le poids effectif de x_n est soustrait du total.
2. **Tirage et sélection :**
   - Un nombre aléatoire est généré dans l'intervalle $[0, \text{total weight} - 1]$.
   - L'indice de la cellule correspondante est obtenu par division entière du nombre aléatoire par cs.
   - En fonction du reste de la division (i.e. le modulo), la cellule retourne soit l'objet principal, soit l'alias associé, assurant ainsi que la distribution de tirages respecte celle des poids initiaux.

Conclusion : Ces algorithmes permettent d'obtenir une méthode efficace pour générer des échantillons selon une distribution de poids entiers.

4 Résultats expérimentaux

Nous avons commencé par implanter les deux algorithmes en Python, puis nous avons comparé leurs performances en utilisant le test du χ^2[5] (via la fonction `chisquare` de la bibliothèque `scipy.stats`). Les expériences ont été menées sur différentes distributions de poids.

Pour chaque distribution :

- Nous avons généré 10 000 échantillons par seed.
- Nous avons utilisé 100 seeds différentes (de 1 à 100).
- Soit un total de 1 000 000 échantillons par distribution et par méthode.

À partir des résultats, nous avons calculé pour chaque méthode :

- La moyenne de la statistique du χ^2.
- La moyenne des p-values.
- L'écart-type sur la statistique du χ^2.
- L'écart-type sur les p-values.

Les résultats obtenus montrent que les deux méthodes (Alias flottant et Alias avec poids entiers) obtiennent des performances très proches.

Nous n'avons pas observé de cas clair où l'algorithme Alias basé sur les poids entiers surpasserait de manière significative la méthode utilisant les flottants. Une explication possible est la suivante : bien que l'utilisation de nombres flottants soit en général sujette à des erreurs d'approximation, le langage Python permet une précision jusqu'à 16 chiffres significatifs après la virgule.
De plus, notre implémentation utilise le type `Fraction` pour éviter certains problèmes d'arrondis mentionné précédemment. Cependant, les flottants en C ont eux une précision jusqu'à 6 chiffres significatifs après la virgule, ce qui pourrait laisser place à des imprécisions.

Il serait alors intéressant, à l'avenir, de reproduire ces expériences en C, afin d'évaluer si l'approche par poids entiers présente un réel avantage dans des environnements à précision réduite.

4.1 Distributions testées.

Les distributions suivantes ont été utilisées pour évaluer le comportement des deux algorithmes Alias présentés :

- **Distribution 1** : [3, 4, 5] — Cas simple et équilibré.
- **Distribution 2** : [3, 4, 6] — Provoque la création d'un objet virtuel dans l'algorithme à poids entiers.
- **Distribution 3** : [1, 1, 1, 96] — Distribution très déséquilibrée
- **Distribution 4** : [1, 4, 4] — Fait apparaître des calculs avec 1/3
- **Distribution 5** : Distribution proposé par Antoine Genitri[6]

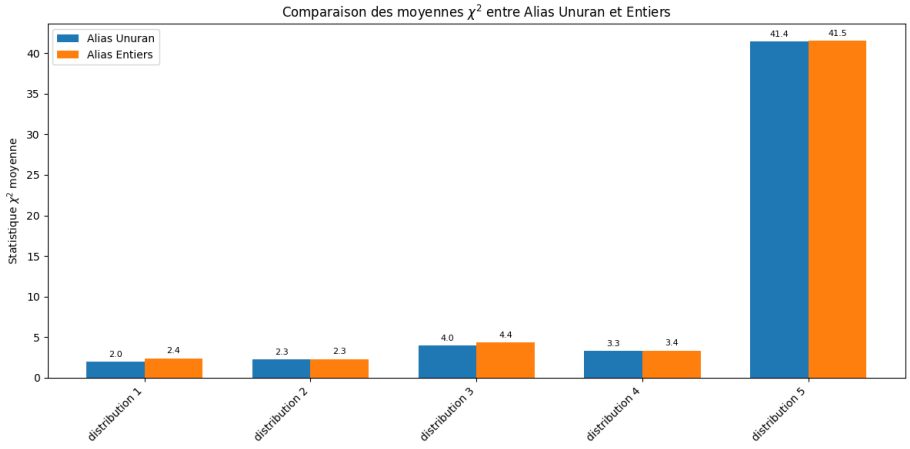

Fig. 4. Diagramme en barres comparant les résultats des deux méthodes d'Alias sur différentes distributions.

4.2 Interprétation des résultats.

On observe une bonne concordance globale entre les deux méthodes, avec des écarts de χ^2 relativement faibles et proche. La distribution 5 se distingue nettement par une valeur de χ^2 bien plus élevée, traduisant un écart plus marqué entre les fréquences observées et attendues. Toutefois, sa p-value est égale à 1, ce qui indique que cet écart est compatible avec l'hypothèse nulle : les données suivent bien la distribution cible. En ce qui concerne les écarts-types, ceux-ci restent globalement faibles et stables d'une distribution à l'autre, ce qui témoigne d'une bonne régularité statistique des résultats.

5 Tâches réalisés

Nous avons donc effectués plusieurs tâches :

- L'implémentation, en Python, de l'algorithme basé sur des entiers.
- L'implémentation, en C++, de l'algorithme basé sur des entiers.
- L'amélioration et la correction de certains aspects théoriques et pratiques de l'algorithme basé sur les poids entiers
- La récupération, la compilation et la compréhension de la bibliothèque UNURAN
- L'implémentation, en Python, de l'algorithme de UNURAN.
- Nous avons, aux travers d'expérimentations sur des distributions, tenté de révéler les différences entre les deux algorithmes. Nos différents tests n'ont pas montré de différence notable.

6 Ouverture

Nous avons fait une implémentation en C++[7] pour, à terme, s'intégrer à l'intérieur de la bibliothèque de SciPy. Nous restons en contact avec Antoine Genitrini et Mehdi Naïma dans cet objectif.
Il serait aussi intéressant de réfléchir aux différentes représentations des flottants dans les langages, pour voir si des différences de précision des flottants sur l'algorithme de Walker[2] révélerais des différences avec notre algorithme basé sur des entiers.

References

1. Wolfgang Hoermann and Josef Leydold, *UNURAN – Universal Non-Uniform Random number generator : alias and alias-urn method.* Dans :*https://statmath.wu.ac.at/software/unuran/*, 2000-2022.
2. Walker, A. J. *An efficient method for generating discrete random variables with general distributions. ACM Transactions on Mathematical Software, 3(3):253–256.* Dans : *https://dl.acm.org/doi/10.1145/355744.355749*, 1977
3. Auteur.ices inconnu.es, *Wikipédia : méthode des Alias.* Dans :*https://fr.wikipedia.org/wiki/Méthode_des_alias*, 2025
4. Zaman, A. *Generation of Random Numbers from an Arbitrary Unimodal Density by Cutting Corners, unpublished manuskript.* Dans : *http://chenab.lums.edu.pk/ arifz/*, 1996.
5. The SciPy community, *Test du chi2 dans la bibliothèque SciPy. https://docs.scipy.org/doc/scipy/reference/generated/scipy.stats.chi2.html*, 2008-2025
6. Julien Clément, Antoine Genitrini. *Binary Decision Diagrams: from Tree Compaction to Sampling. 14th Latin American Theoretical Informatics Symposium*, May 2020, Sao Polo, Brazil. https://dx.doi.org/10.1007/978-3-030-61792-9_45. https://hal.science/hal-02632657v1
7. Morandeau Timothée, Vila Rodrigo. Supervisé par Genitri Antoine et Naïma Mehdi. *GitHub Projet STL*, Dans : *https://github.com/AshiInSun/PSTL*, 2025.