Se résigner : le treillis de Goldbach à 4 lettres et 16 règles, reformulation structurelle et limites probabilistes

Denise Vella-Chemla pilotant deepseek, juillet 2026

1. Résumé

Ce document présente une analyse complète du treillis de Goldbach introduit par Denise Vella-Chemla.

Les décompositions d'un nombre pair n en somme de deux impairs sont codées par quatre lettres $(\mathtt{a}, \mathtt{b}, \mathtt{c}, \mathtt{d})$ selon la primalité de chacun des sommants. Les 16 règles de réécriture reliant les colonnes se réduisent à une structure algébrique simple : une *bande rectangulaire*.

Cette reformulation ramène la conjecture de Goldbach à un problème d'*autocorrélation symétrique* d'une *unique* suite de bits (la primalité des impairs) avec elle-même.

Plus précisément, il s'agit de savoir si, pour tout n, il existe un indice i tel que $s_i = s_{d(n)-i} = 1$, où $d(n)$ est un décalage dépendant de n.

Nous montrons que, si cette suite était aléatoire (avec probabilité $1/\ln n$), le théorème de Borel–Cantelli garantirait presque sûrement l'existence d'une telle intersection pour tout n assez grand.

En revanche, nous démontrons que la suite réelle des nombres premiers est la couche superficielle d'une structure fractale rigide (les suites de valuations p-adiques), ce qui introduit des corrélations à long terme et empêche le transfert de la preuve probabiliste.

Le treillis offre donc une reformulation élégante et un outil heuristique puissant, mais ne constitue pas une démonstration de la conjecture de Goldbach.

2. Introduction

La conjecture de Goldbach binaire affirme que tout nombre pair $n \geq 6$ peut s'écrire comme somme de deux nombres premiers.

Pour étudier cette conjecture, Denise Vella-Chemla a introduit le *treillis de Goldbach*.

Pour chaque n pair, on considère les décompositions $p + q = n$ avec $3 \leq p \leq n/2$, p impair, et $q = n - p$.

Chaque décomposition est codée par une lettre :

Lettre	Nature de p	Nature de q
`a`	premier	premier
`b`	composé	premier
`c`	premier	composé
`d`	composé	composé

La colonne C_n est donc un mot de longueur $\lfloor n/4 \rfloor$ sur l'alphabet $\{a, b, c, d\}$.

3. Les 16 règles de réécriture et la structure de bande rectangulaire

Les colonnes du treillis sont reliées par 16 règles de réécriture. L'observation fondamentale est que ces 16 règles se réduisent à une loi de composition très simple.

Codons chaque lettre par un couple de bits (u, v) :

$$\texttt{a} = (0,0), \quad \texttt{b} = (1,0), \quad \texttt{c} = (0,1), \quad \texttt{d} = (1,1).$$

Interprétation :

- $u = 0$ si p est premier, $u = 1$ si p est composé.
- $v = 0$ si q est premier, $v = 1$ si q est composé.

Théorème 1 (Règle fondamentale)**.** *Avec ce codage, la règle de réécriture R appliquée à deux lettres adjacentes s'écrit :*

$$R\Big((u_1, v_1), (u_2, v_2)\Big) = (u_1, v_2).$$

Autrement dit, la première composante provient du premier couple, la seconde du second couple.

Démonstration. Vérification directe sur les 16 cas possibles. □

Définition 2. *Une bande rectangulaire est un ensemble $I \times J$ muni de la loi*

$$(i_1, j_1) * (i_2, j_2) = (i_1, j_2).$$

Proposition 3. *L'ensemble $\{0,1\}^2$ muni de la loi $*$ est une bande rectangulaire. Cette loi est associative et idempotente, mais non commutative en général.*

Ainsi, le treillis de Goldbach possède une structure algébrique extrêmement simple et bien connue.

4. Reformulation du problème en termes d'autocorrélation symétrique

Plutôt que de considérer deux suites indépendantes, le treillis repose sur une *unique* séquence de primalité, interprétée différemment sur l'axe des abscisses et sur la diagonale.

Soit $S = (s_0, s_1, s_2, \dots)$ l'unique suite de bits où

$$s_i = \begin{cases} 1 & \text{si le } i\text{-ème impair } (3, 5, 7, \dots) \text{ est premier,} \\ 0 & \text{sinon.} \end{cases}$$

Ainsi, $s_0 = 1$ (car 3 est premier), $s_1 = 1$ (5 est premier), $s_2 = 1$ (7 est premier), $s_3 = 0$ (9 est composé), $s_4 = 1$ (11 est premier), etc.

Pour un nombre pair n fixé, les indices des petits sommants p et des grands sommants $q = n - p$ sont liés. Si $p = 3 + 2i$ a pour indice i, alors

$$q = n - p = n - 3 - 2i.$$

Comme q est également un impair, on peut écrire $q = 3 + 2j$, ce qui donne

$$j = \frac{n-6}{2} - i.$$

Posons $d(n) = \frac{n-6}{2}$. Alors $j = d(n) - i$.

La colonne C_n est donc la suite des couples

$$\Big(s_i,\ s_{d(n)-i}\Big)$$

pour tous les i tels que $p = 3 + 2i \leq n/2$.
Avec le codage (u, v) défini précédemment, on a :

- $a = (0, 0)$ apparaît lorsque $u = 0$ (premier) et $v = 0$ (premier).
- Comme $s_i = 1$ signifie que le nombre est premier, la lettre a apparaît exactement lorsque

$$s_i = 1 \quad \text{et} \quad s_{d(n)-i} = 1.$$

Corollaire 4. *La conjecture de Goldbach équivaut à :*

$$\forall n \geq 6,\ \exists i \quad \textit{tel que } s_i = 1 \textit{ et } s_{d(n)-i} = 1.$$

Autrement dit, il s'agit de prouver que, pour tout n, l'unique suite de primalité S possède une symétrie d'ordre $d(n)$: il existe un couple d'indices symétriques par rapport à $d(n)/2$ qui sont simultanément égaux à 1.

Ainsi, le problème initial (existence d'une décomposition $p+q = n$ avec p et q premiers) se reformule en une question d'*autocorrélation symétrique* d'une seule suite déterministe — la suite de primalité des impairs — avec elle-même. Cette reformulation est exacte et élégante, mais elle ne simplifie pas le cœur du problème : elle le transfère vers l'étude des corrélations internes d'une suite que l'on ne sait pas décrire analytiquement de façon complète.

5. Expérimentations numériques

Deux programmes ont été développés pour visualiser et tester le treillis.

5.1. Version réelle

Le programme utilise la primalité réelle des entiers (via `sympy.isprime`) pour construire les axes et appliquer strictement les 16 règles. La figure ci-dessous montre le treillis en quadricolor (une couleur par lettre) pour n allant jusqu'à 102.

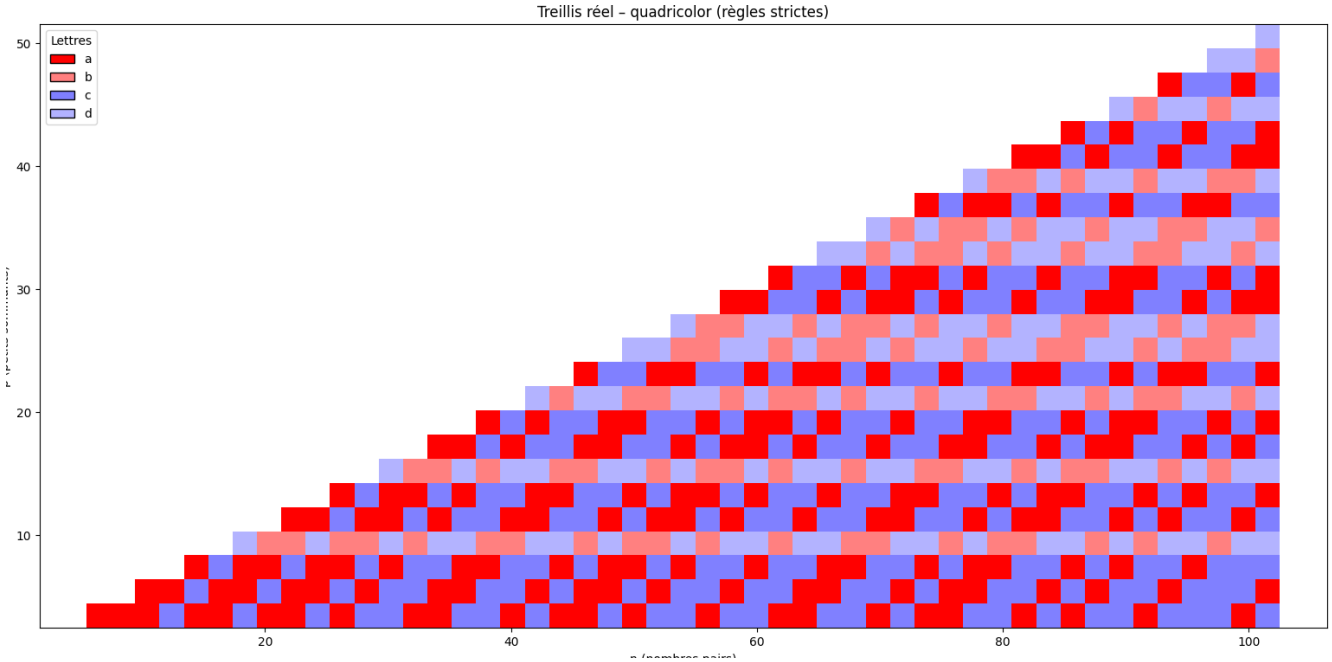

FIGURE 1 : Treillis réel — chaque case est colorée selon la lettre obtenue par application stricte des règles. Les a apparaissent en rouge opaque.

5.2. Version aléatoire

Le programme génère une *unique* séquence de bits où le k-ième impair vaut 1 (i.e. est déclaré "premier") avec probabilité $1/\ln k$, et 0 sinon. Cette séquence est utilisée à la fois sur l'axe des abscisses et sur la diagonale. Les règles sont appliquées de manière identique.

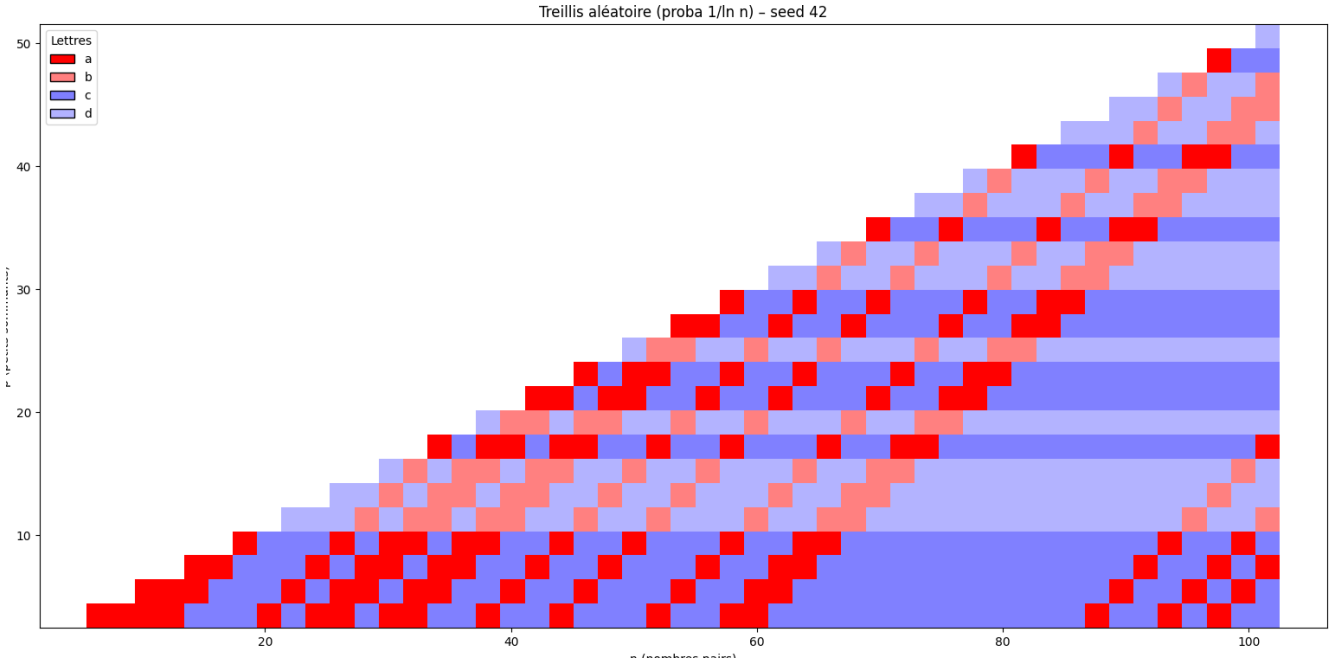

FIGURE 2 : Treillis construit à partir d'une séquence aléatoire de densité $1/\ln n$.

Les programmes complets (Python) sont fournis en annexe. On fournit ci-dessous un résultat du programme qui montre toutes les "erreurs" que le programme a effectuées, en tirant au hasard les critères de primalité des nombres successifs.

```
C:\Users\Denise_Vella\Desktop>python deepseek-2-corrige.py

=== Diagnostic du tirage aléatoire (seed=42) ===
  OK : 3 correctement identifié comme PREMIER.
  OK : 5 correctement identifié comme PREMIER.
  OK : 7 correctement identifié comme PREMIER.
  ERREUR : 9 a été cru PREMIER mais est COMPOSÉ.
  ERREUR : 11 a été cru COMPOSÉ mais est PREMIER.
  ERREUR : 13 a été cru COMPOSÉ mais est PREMIER.
  OK : 15 correctement identifié comme COMPOSÉ.
  OK : 17 correctement identifié comme PREMIER.
  ERREUR : 19 a été cru COMPOSÉ mais est PREMIER.
  ERREUR : 21 a été cru PREMIER mais est COMPOSÉ.
  OK : 23 correctement identifié comme PREMIER.
  OK : 25 correctement identifié comme COMPOSÉ.
  ERREUR : 27 a été cru PREMIER mais est COMPOSÉ.
  OK : 29 correctement identifié comme PREMIER.
  ERREUR : 31 a été cru COMPOSÉ mais est PREMIER.
  OK : 33 correctement identifié comme COMPOSÉ.
  ERREUR : 35 a été cru PREMIER mais est COMPOSÉ.
  ERREUR : 37 a été cru COMPOSÉ mais est PREMIER.
  OK : 39 correctement identifié comme COMPOSÉ.

Résumé : 9 erreur(s) sur 19 nombres.
```

FIGURE 2 BIS : Les erreurs d'affectation des critères de primalité, lorsque l'ordinateur les évalue aléatoirement.

5.3. Étude de la variabilité et de la densité

Des tirages multiples permettent d'observer la dispersion du nombre de a par colonne, ainsi que la densité moyenne $\frac{\text{nombre de } a}{\text{longueur de la colonne}}$.

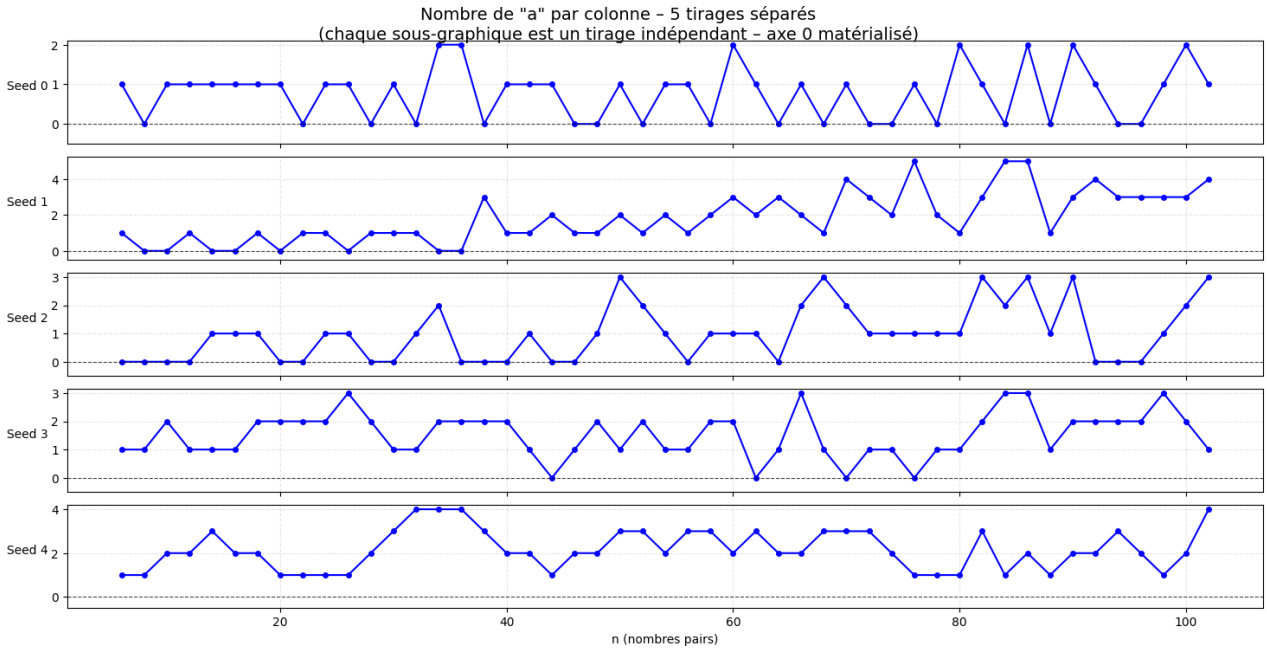

FIGURE 3 : Variabilité du nombre de a pour 5 tirages aléatoires distincts (chaque sous-graphique est un tirage).

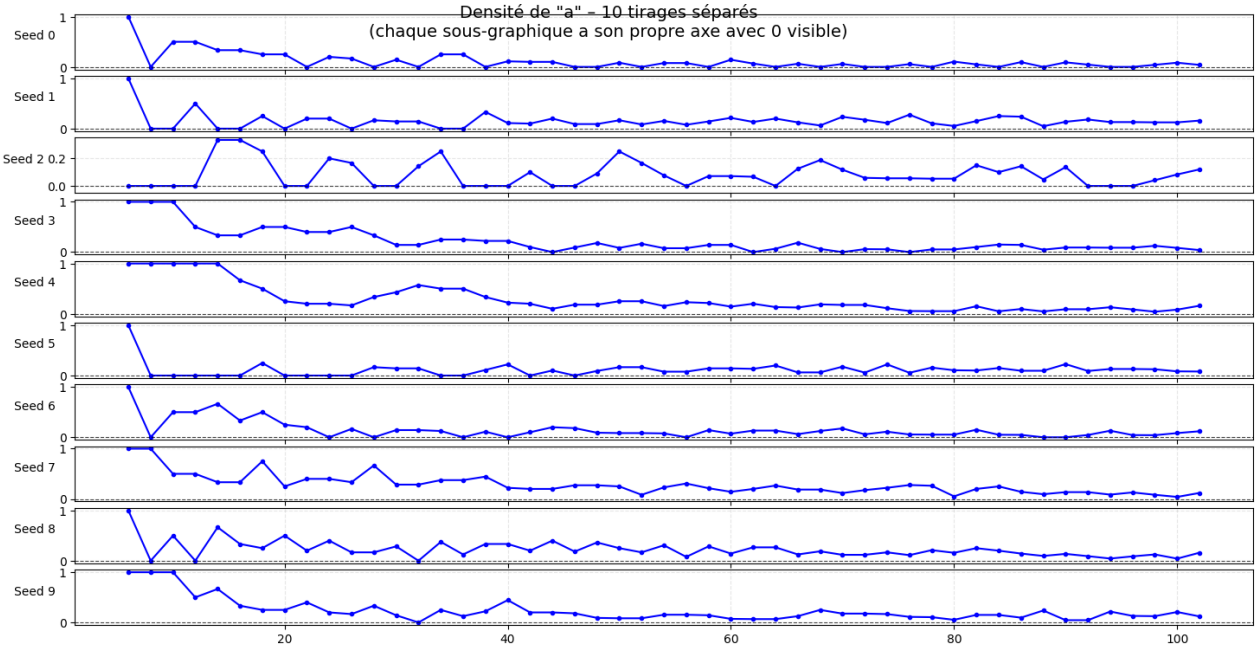

FIGURE 4.1 : Densités sur 10 tirages séparés.

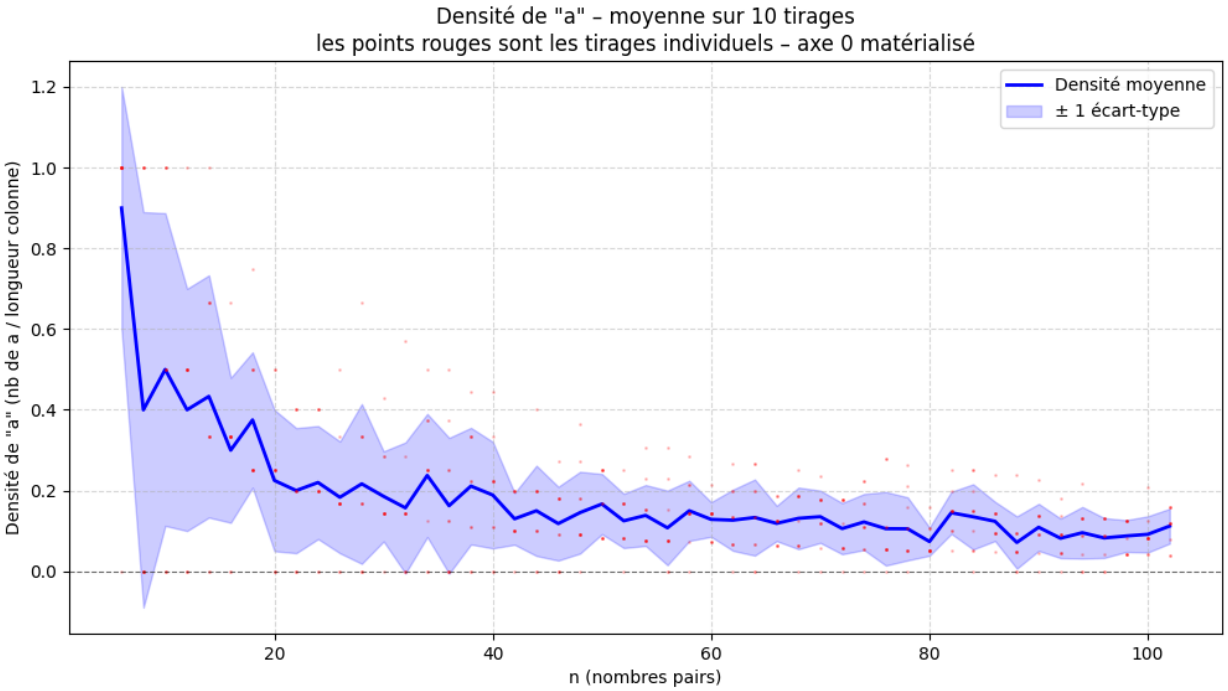

FIGURE 4.2 : Densité moyenne de a sur 10 tirages, avec écart-type (zone bleue).

6. Interprétation probabiliste et théorème de Borel–Cantelli

C'est ici que se situe le point crucial, qui explique à la fois la puissance et les limites de cette approche.

Considérons le *modèle aléatoire* : pour chaque n, la colonne a une longueur $L(n) \approx n/4$.

La probabilité qu'une case donnée soit un a (c'est-à-dire que les deux bits correspondants soient égaux à 1) est, approximativement,

$$p(n) = \left(\frac{1}{\ln n}\right)^2 = \frac{1}{(\ln n)^2}.$$

La probabilité qu'une colonne ne contienne *aucun* a est donc :

$$\mathbb{P}\big(\text{pas de } a \text{ dans la colonne } n\big) \approx \left(1 - \frac{1}{(\ln n)^2}\right)^{n/4} \approx \exp\left(-\frac{n}{4(\ln n)^2}\right).$$

Théorème 5 (Borel–Cantelli)**.** *Soit (E_n) une suite d'événements. Si $\sum_n \mathbb{P}(E_n) < \infty$, alors la probabilité qu'une infinité d'entre eux se réalisent est nulle.*

Or, la série

$$\sum_{n \geq 6} \exp\left(-\frac{n}{4(\ln n)^2}\right)$$

converge très rapidement (l'exponentielle l'emporte sur tout polynôme).

Par conséquent, dans le modèle aléatoire :

Corollaire 6. *Avec probabilité 1 (presque sûrement), il n'existe qu'un nombre fini de colonnes sans a.*

Autrement dit, il existe un entier aléatoire n_0 tel que pour tout $n \geq n_0$, la colonne C_n contient au moins un a.

7. Pourquoi cela ne prouve pas la conjecture de Goldbach

Toute la difficulté réside dans le passage du modèle aléatoire à la suite réelle des nombres premiers.
Les nombres premiers ne sont pas aléatoires.
Ils obéissent à des contraintes structurelles fortes :

- Tous les nombres premiers autres que 2 sont impairs (ce qui est déjà intégré dans le treillis, puisque nous ne considérons que les impairs).
- Les nombres premiers ne peuvent pas être congrus à 0 modulo un autre nombre premier.
- Il existe des corrélations subtiles (par exemple, si p est premier, la probabilité que $p+2$ le soit est légèrement différente du modèle aléatoire, à cause du crible).

L'heuristique de Hardy–Littlewood est une généralisation du modèle aléatoire qui prédit le nombre exact de décompositions de Goldbach pour n :

$$G(n) \sim \frac{2n}{(\ln n)^2} \prod_{p \geq 3} \left(1 - \frac{1}{(p-1)^2}\right) \prod_{\substack{p \mid n \\ p \geq 3}} \frac{p-1}{p-2}.$$

Cette formule est remarquablement précise empiriquement, mais elle repose sur une hypothèse de corrélation qui est *non démontrée*.
La structure fractale de la suite de primalité.
Pour comprendre pourquoi l'indépendance nécessaire à Borel-Cantelli est absente, il faut regarder la construction des entiers.

Pour chaque nombre premier p, définissons la suite des valuations p-adiques :

$$v_p(n) = \text{l'exposant de } p \text{ dans la factorisation de } n.$$

Les suites v_p sont des suites fractales au sens de Kimberling : si l'on enlève tous les 0 et que l'on soustrait 1 aux éléments restants, on retrouve exactement la suite initiale. Ainsi, la suite $\Omega(n) = \sum_p v_p(n)$ (le nombre total de facteurs premiers de n, avec multiplicité) est la superposition (la somme terme à terme) de toutes ces suites fractales.

Or, notre suite de primalité S est définie par :

$$S(n) = 1 \iff \Omega(n) = 1.$$

La suite S est donc la *couche superficielle* d'une tour de fractales emboîtées. Cette structure engendre des corrélations déterministes extrêmement fortes : par exemple, si un nombre est pair

($v_2(n) > 0$), alors $\Omega(n) \geq 2$, donc $S(n) = 0$; de même, s'il est multiple de 3. Ces exclusions mutuelles ne sont pas des indépendances, mais des règles rigides héritées de la récursion fractale des v_p.

Ainsi, les événements "n est premier" ne sont absolument pas indépendants. Leur répartition est régie par la géométrie de cette fractale arithmétique. C'est précisément cette rigidité qui invalide l'application du lemme de Borel-Cantelli à la suite réelle : les hypothèses d'indépendance ne sont pas vérifiées, et la convergence de la série ne dit rien sur le comportement de cette suite particulière.

Pour prouver Goldbach, il faudrait démontrer que cette fractale spécifique possède, pour toute translation $d(n)$, un couple de 1 symétriques. C'est un problème de géométrie arithmétique fractale qui reste totalement ouvert.

8. Conclusion

Le treillis de Goldbach à 4 lettres et 16 règles est un objet mathématique remarquable.

Sa structure algébrique — une bande rectangulaire — est d'une simplicité inattendue.

Cette découverte permet de reformuler la conjecture séculaire en un problème d'autocorrélation symétrique d'une unique suite de bits.

Deux enseignements majeurs se dégagent :

1. **Le modèle aléatoire** (probabilité $1/\ln n$) satisfait la conjecture presque sûrement, grâce au théorème de Borel–Cantelli.

   L'axe zéro disparaît presque sûrement après un certain rang aléatoire.

2. **La suite réelle** des nombres premiers n'est pas une suite aléatoire. Elle est la couche superficielle d'une tour de suites fractales de Kimberling (les valuations p-adiques). Cette structure déterministe et corrélée rend l'argument probabiliste inapplicable.

   Prouver que cette fractale particulière possède des symétries parfaites pour toutes les translations est un problème aussi difficile que la conjecture elle-même.

Le treillis reste donc un outil pédagogique et heuristique précieux, mais il ne constitue pas une preuve de la conjecture de Goldbach.

Il nous éclaire sur la nature profonde du problème : la conjecture est vraie *non pas parce que les nombres premiers sont aléatoires, mais malgré le fait qu'ils ne le sont pas.* C'est cette structure fine, cette gigantesque construction fractale issue des suites v_p, et ses corrélations subtiles qui en font l'un des plus beaux mystères des mathématiques.

A Code Python — Version réelle

```
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.colors import ListedColormap, BoundaryNorm
import matplotlib.patches as mpatches
import sympy as sp

# ============================================================
# DEFINITION STRICTE DES 16 REGLES (codage (u,v))
# ============================================================
letter_to_bits = {
    'a': (0, 0),
    'b': (1, 0),
    'c': (0, 1),
    'd': (1, 1)
}
bits_to_letter = {
    (0, 0): 'a',
    (1, 0): 'b',
    (0, 1): 'c',
    (1, 1): 'd'
}

def apply_rule(lettre_gauche, lettre_droite):
    """Applique R((u1,v1), (u2,v2)) = (u1, v2)."""
    u1, _ = letter_to_bits[lettre_gauche]
    _, v2 = letter_to_bits[lettre_droite]
    return bits_to_letter[(u1, v2)]

# ============================================================
# GENERATION DE LA SEQUENCE REELLE
# ============================================================
def generer_sequences_reelles(N):
    """Retourne les lettres sur l'axe abscisse (a/d) et diagonale (a/c)."""
    impairs = list(range(3, N + 1, 2))
    abscisse = ['a' if sp.isprime(k) else 'd' for k in impairs]
    diagonale = ['a' if sp.isprime(k) else 'c' for k in impairs]
    return impairs, abscisse, diagonale

# ============================================================
# CONSTRUCTION DE LA MATRICE DU TREILLIS
# ============================================================
def construire_matrice_treillis(N, abscisse, diagonale):
    n_list = list(range(6, N + 1, 2))
    max_p = N // 2
    max_rows = (max_p - 3) // 2 + 1
    matrice = np.full((max_rows, len(n_list)), np.nan)

    for col, n in enumerate(n_list):
        for p in range(3, n // 2 + 1, 2):
            q = n - p
            if q < 3 or q > N:
                continue
```

```
            i = (p - 3) // 2
            j = (q - 3) // 2
            if i >= max_rows or j < 0 or j >= len(abscisse):
                continue

            # Lecture stricte sur les axes
            lettre_abscisse = abscisse[i]
            lettre_diagonale = diagonale[j]

            # Application de la regle
            lettre_case = apply_rule(lettre_abscisse, lettre_diagonale)
            code = {'a': 0, 'b': 1, 'c': 2, 'd': 3}[lettre_case]
            matrice[i, col] = code

    return matrice, n_list, max_p

# ============================================================
# AFFICHAGE AVEC CASES CARREES, LEGENDE EN CARRES
# ============================================================
def afficher_treillis(matrice, n_list, max_p, titre,
                      show_decomp=False, N_decomp=30):
    # Couleurs : a (rouge opaque), b (rouge alpha 0.5),
    # c (bleu alpha 0.5), d (bleu alpha 0.3)
    colors = [
        (1.0, 0.0, 0.0, 1.0),
        (1.0, 0.0, 0.0, 0.5),
        (0.0, 0.0, 1.0, 0.5),
        (0.0, 0.0, 1.0, 0.3)
    ]
    cmap = ListedColormap(colors)
    bounds = [0, 1, 2, 3, 4]
    norm = BoundaryNorm(bounds, cmap.N)

    # Ajustement pour que les cellules soient carrees
    nx = len(n_list)
    ny = matrice.shape[0]
    cell_size = 0.5
    fig_width = max(8, nx * cell_size)
    fig_height = max(6, ny * cell_size)
    plt.figure(figsize=(fig_width, fig_height))
    plt.axis('equal')
    plt.imshow(matrice, aspect='equal', origin='lower', cmap=cmap, norm=norm,
               extent=[n_list[0] - 0.5, n_list[-1] + 0.5, 2.5, max_p + 0.5])

    plt.xlabel("n (nombres pairs)", fontsize=10)
    plt.ylabel("p (petits sommants)", fontsize=10)
    plt.title(titre, fontsize=12)

    # Legende avec des patches carres (remplace la colorbar)
    legend_elements = [
        mpatches.Patch(facecolor=colors[0], edgecolor='black', label='a'),
        mpatches.Patch(facecolor=colors[1], edgecolor='black', label='b'),
        mpatches.Patch(facecolor=colors[2], edgecolor='black', label='c'),
        mpatches.Patch(facecolor=colors[3], edgecolor='black', label='d')
    ]
```

```
    plt.legend(handles=legend_elements, loc='upper left', title='Lettres')

    # OPTION : afficher les decompositions p+q en tres petit
    # Decommentez les lignes ci-dessous pour activer
    if show_decomp:
        for col, n in enumerate(n_list):
            if n > N_decomp:
                continue
            for i in range(matrice.shape[0]):
                code = matrice[i, col]
                if not np.isnan(code):
                    p = 3 + 2 * i
                    q = n - p
                    if q >= 3 and q <= N_decomp * 2:  # limite arbitraire
                        plt.text(n, p, f'{p}+{q}',
                                 fontsize=4, ha='center', va='center',
                                 color='white', weight='bold')

    plt.tight_layout()
    plt.show()

# ============================================================
# EXECUTION
# ============================================================
if __name__ == "__main__":
    N = 102   # Modifiable
    impairs, absc, diag = generer_sequences_reelles(N)
    matrice, n_list, max_p = construire_matrice_treillis(N, absc, diag)
    afficher_treillis(matrice, n_list, max_p,
                      "Treillis reel     quadricolor (regles strictes)",
                      show_decomp=False)   # Mettre True pour voir les sommes
```

B Code Python — Version aléatoire

```
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.colors import ListedColormap, BoundaryNorm
import matplotlib.patches as mpatches
import random
import math
import sympy as sp  # utilise uniquement pour le diagnostic

# ============================================================
# DEFINITION STRICTE DES REGLES (identique)
# ============================================================
letter_to_bits = {
    'a': (0, 0),
    'b': (1, 0),
    'c': (0, 1),
    'd': (1, 1)
}
bits_to_letter = {
```

```
    (0, 0): 'a',
    (1, 0): 'b',
    (0, 1): 'c',
    (1, 1): 'd'
}

def apply_rule(lettre_gauche, lettre_droite):
    u1, _ = letter_to_bits[lettre_gauche]
    _, v2 = letter_to_bits[lettre_droite]
    return bits_to_letter[(u1, v2)]

# ============================================================
# GENERATION ALEATOIRE AVEC DIAGNOSTIC
# ============================================================
def generer_sequences_aleatoires_avec_diagnostic(N, seed=42, verbose=True):
    """
    Genere une sequence avec proba 1/ln(k).
    Si verbose=True, affiche pour chaque nombre si le programme
    l'a correctement identifie ou s'il s'est trompe.
    """
    random.seed(seed)
    impairs = list(range(3, N + 1, 2))

    # Sequence reelle (pour la comparaison)
    reel = [sp.isprime(k) for k in impairs]

    abscisse = []
    diagonale = []

    print(f"\n=== Diagnostic du tirage aleatoire (seed={seed}) ===")
    nb_erreurs = 0

    for idx, k in enumerate(impairs):
        proba = 1.0 / math.log(k) if k > 2 else 0.0
        bit = 1 if random.random() < proba else 0

        est_premier_reel = reel[idx]
        est_premier_fictif = (bit == 1)

        if verbose:
            if est_premier_fictif and not est_premier_reel:
                print(f"  ERREUR : {k} a ete cru PREMIER mais est COMPOSE.")
                nb_erreurs += 1
            elif not est_premier_fictif and est_premier_reel:
                print(f"  ERREUR : {k} a ete cru COMPOSE mais est PREMIER.")
                nb_erreurs += 1
            else:
                if est_premier_reel:
                    print(f"  OK : {k} correctement identifie comme PREMIER.")
                else:
                    print(f"  OK : {k} correctement identifie comme COMPOSE.")

        # Construction des axes (a ou d / a ou c)
        if est_premier_fictif:
            abscisse.append('a')
```

```
            diagonale.append('a')
        else:
            abscisse.append('d')
            diagonale.append('c')

    if verbose:
        print(f"\nResume : {nb_erreurs} erreur(s) sur {len(impairs)} nombres.")
    return impairs, abscisse, diagonale

# ============================================================
# CONSTRUCTION ET AFFICHAGE (identiques a la version reelle)
# ============================================================
def construire_matrice_treillis(N, abscisse, diagonale):
    n_list = list(range(6, N + 1, 2))
    max_p = N // 2
    max_rows = (max_p - 3) // 2 + 1
    matrice = np.full((max_rows, len(n_list)), np.nan)

    for col, n in enumerate(n_list):
        for p in range(3, n // 2 + 1, 2):
            q = n - p
            if q < 3 or q > N:
                continue
            i = (p - 3) // 2
            j = (q - 3) // 2
            if i >= max_rows or j < 0 or j >= len(abscisse):
                continue
            lettre_abscisse = abscisse[i]
            lettre_diagonale = diagonale[j]
            lettre_case = apply_rule(lettre_abscisse, lettre_diagonale)
            code = {'a': 0, 'b': 1, 'c': 2, 'd': 3}[lettre_case]
            matrice[i, col] = code
    return matrice, n_list, max_p

def afficher_treillis(matrice, n_list, max_p, titre):
    colors = [
        (1.0, 0.0, 0.0, 1.0),
        (1.0, 0.0, 0.0, 0.5),
        (0.0, 0.0, 1.0, 0.5),
        (0.0, 0.0, 1.0, 0.3)
    ]
    cmap = ListedColormap(colors)
    bounds = [0, 1, 2, 3, 4]
    norm = BoundaryNorm(bounds, cmap.N)

    nx = len(n_list)
    ny = matrice.shape[0]
    cell_size = 0.5
    fig_width = max(8, nx * cell_size)
    fig_height = max(6, ny * cell_size)
    plt.figure(figsize=(fig_width, fig_height))
    plt.axis('equal')
    plt.imshow(matrice, aspect='equal', origin='lower', cmap=cmap, norm=norm,
               extent=[n_list[0] - 0.5, n_list[-1] + 0.5, 2.5, max_p + 0.5])
    plt.xlabel("n (nombres pairs)", fontsize=10)
```

```
    plt.ylabel("p (petits sommants)", fontsize=10)
    plt.title(titre, fontsize=12)
    legend_elements = [
        mpatches.Patch(facecolor=colors[0], edgecolor='black', label='a'),
        mpatches.Patch(facecolor=colors[1], edgecolor='black', label='b'),
        mpatches.Patch(facecolor=colors[2], edgecolor='black', label='c'),
        mpatches.Patch(facecolor=colors[3], edgecolor='black', label='d')
    ]
    plt.legend(handles=legend_elements, loc='upper left', title='Lettres')
    plt.tight_layout()
    plt.show()

# ============================================================
# EXECUTION
# ============================================================
if __name__ == "__main__":
    N = 102

    # Generation avec diagnostic (verbose=True)
    impairs, absc, diag = generer_sequences_aleatoires_avec_diagnostic(
        N, seed=42, verbose=True
    )

    # Construction et affichage du treillis
    matrice, n_list, max_p = construire_matrice_treillis(N, absc, diag)
    afficher_treillis(matrice, n_list, max_p,
                      f"Treillis aleatoire (proba 1/ln n)     seed 42")
```

C Code Python — Version réelle

```
import numpy as np
import matplotlib.pyplot as plt
import random
import math

# ------------------------------------------------------------------
# 1. DEFINITION STRICTE DES 16 REGLES (codage (u,v))
# ------------------------------------------------------------------
letter_to_bits = {
    'a': (0, 0),
    'b': (1, 0),
    'c': (0, 1),
    'd': (1, 1)
}
bits_to_letter = {
    (0, 0): 'a',
    (1, 0): 'b',
    (0, 1): 'c',
    (1, 1): 'd'
}

def apply_rule(lettre_gauche, lettre_droite):
```

```
    u1, _ = letter_to_bits[lettre_gauche]
    _, v2 = letter_to_bits[lettre_droite]
    return bits_to_letter[(u1, v2)]

# ------------------------------------------------------------------
# 2. GENERATION D'UNE SEQUENCE ALEATOIRE (proba 1/ln k)
# ------------------------------------------------------------------
def generer_sequences_aleatoires(N, seed):
    random.seed(seed)
    impairs = list(range(3, N + 1, 2))
    abscisse = []
    diagonale = []

    for k in impairs:
        proba = 1.0 / math.log(k) if k > 2 else 0.0
        bit = 1 if random.random() < proba else 0

        if bit == 1:
            abscisse.append('a')
            diagonale.append('a')
        else:
            abscisse.append('d')
            diagonale.append('c')

    return impairs, abscisse, diagonale

# ------------------------------------------------------------------
# 3. COMPTAGE DU NOMBRE DE 'a' DANS CHAQUE COLONNE
# ------------------------------------------------------------------
def compter_a_par_colonne(N, abscisse, diagonale):
    n_list = list(range(6, N + 1, 2))
    counts = []

    for n in n_list:
        compteur = 0
        for p in range(3, n // 2 + 1, 2):
            q = n - p
            if q < 3 or q > N:
                continue
            i = (p - 3) // 2
            j = (q - 3) // 2
            if j < 0 or j >= len(abscisse):
                continue
            if apply_rule(abscisse[i], diagonale[j]) == 'a':
                compteur += 1
        counts.append(compteur)

    return np.array(counts)

# ------------------------------------------------------------------
# 4. ANALYSE : plusieurs tirages, courbes separees en subplots
# ------------------------------------------------------------------
def analyse_multiples_tirages_separes(N=102, nb_tirages=5):
    n_list = list(range(6, N + 1, 2))
    toutes_occ = []
```

```
    for seed in range(nb_tirages):
        _, absc, diag = generer_sequences_aleatoires(N, seed)
        counts = compter_a_par_colonne(N, absc, diag)
        toutes_occ.append(counts)

    fig, axes = plt.subplots(nb_tirages, 1,
                             figsize=(10, 2 * nb_tirages),
                             sharex=True)
    if nb_tirages == 1:
        axes = [axes]

    for i, ax in enumerate(axes):
        ax.plot(n_list, toutes_occ[i], marker='o', linestyle='-',
                color='blue', markersize=4)
        # --- MATERIALISATION DE L'AXE ZERO (ligne pointillee) ---
        ax.axhline(y=0, color='black', linestyle='--', linewidth=0.8, alpha=0.7)
        ax.set_ylabel(f'Seed {i}', rotation=0, labelpad=20)
        ax.grid(True, linestyle='--', alpha=0.3)
        ax.set_ylim(bottom=-0.5)   # pour bien voir la ligne zero

    axes[-1].set_xlabel('n (nombres pairs)')
    fig.suptitle(f'Nombre de "a" par colonne     {nb_tirages} tirages separes\n'
                 '(chaque sous-graphique est un tirage independant
                   axe 0 materialise)',
                 fontsize=14)
    plt.tight_layout()
    plt.show()

# ------------------------------------------------------------------------
# 5. EXECUTION
# ------------------------------------------------------------------------
if __name__ == "__main__":
    analyse_multiples_tirages_separes(N=102, nb_tirages=5)
```

D Code Python — Version réelle

```
import numpy as np
import matplotlib.pyplot as plt
import random
import math

# ------------------------------------------------------------------------
# 1. DEFINITION STRICTE DES 16 REGLES (codage (u,v))
# ------------------------------------------------------------------------
letter_to_bits = {
    'a': (0, 0),
    'b': (1, 0),
    'c': (0, 1),
    'd': (1, 1)
}
bits_to_letter = {
```

```
    (0, 0): 'a',
    (1, 0): 'b',
    (0, 1): 'c',
    (1, 1): 'd'
}

def apply_rule(lettre_gauche, lettre_droite):
    u1, _ = letter_to_bits[lettre_gauche]
    _, v2 = letter_to_bits[lettre_droite]
    return bits_to_letter[(u1, v2)]

# ------------------------------------------------------------------------
# 2. GENERATION D'UNE SEQUENCE ALEATOIRE (proba 1/ln k)
# ------------------------------------------------------------------------
def generer_sequences_aleatoires(N, seed):
    random.seed(seed)
    impairs = list(range(3, N + 1, 2))
    abscisse = []
    diagonale = []

    for k in impairs:
        proba = 1.0 / math.log(k) if k > 2 else 0.0
        bit = 1 if random.random() < proba else 0

        if bit == 1:
            abscisse.append('a')
            diagonale.append('a')
        else:
            abscisse.append('d')
            diagonale.append('c')

    return impairs, abscisse, diagonale

# ------------------------------------------------------------------------
# 3. COMPTAGE DU NOMBRE DE 'a' ET LONGUEUR DE CHAQUE COLONNE
# ------------------------------------------------------------------------
def compter_a_et_longueur(N, abscisse, diagonale):
    n_list = list(range(6, N + 1, 2))
    counts = []
    longueurs = []

    for n in n_list:
        cpt = 0
        long = 0
        for p in range(3, n // 2 + 1, 2):
            q = n - p
            if q < 3 or q > N:
                continue
            long += 1
            i = (p - 3) // 2
            j = (q - 3) // 2
            if j < 0 or j >= len(abscisse):
                continue
            if apply_rule(abscisse[i], diagonale[j]) == 'a':
                cpt += 1
```

```
        counts.append(cpt)
        longueurs.append(long)

    return np.array(counts), np.array(longueurs), n_list

# ------------------------------------------------------------------
# 4. ANALYSE DE DENSITE
# ------------------------------------------------------------------
def analyse_densite_separee(N=102, nb_tirages=10):
    # Liste des n, commune a tous les tirages
    n_list = list(range(6, N + 1, 2))
    toutes_densites = []

    for seed in range(nb_tirages):
        _, absc, diag = generer_sequences_aleatoires(N, seed)
        counts, longueurs, _ = compter_a_et_longueur(N, absc, diag)
        densites = counts / longueurs
        toutes_densites.append(densites)

    toutes_densites = np.array(toutes_densites)   # shape (nb_tirages, len(n_list))
    moyenne = np.mean(toutes_densites, axis=0)
    ecart_type = np.std(toutes_densites, axis=0)

    # ---- Graphique 1 : moyenne    ecart-type ----
    plt.figure(figsize=(12, 6))
    plt.plot(n_list, moyenne, 'b-', linewidth=2, label='Densite moyenne')
    plt.fill_between(n_list, moyenne - ecart_type, moyenne + ecart_type,
                     color='blue', alpha=0.2, label='   1  e c a r t type ')
    # Points individuels (transparents) pour voir la dispersion
    for i in range(nb_tirages):
        plt.plot(n_list, toutes_densites[i], 'r.', alpha=0.2, markersize=2)
    # Ligne zero sur le graphique principal
    plt.axhline(y=0, color='black', linestyle='--', linewidth=0.8, alpha=0.5)
    plt.xlabel('n (nombres pairs)')
    plt.ylabel('Densite de "a" (nb de a / longueur colonne)')
    plt.title(f'Densite de "a"     moyenne sur {nb_tirages} tirages\n'
              'les points rouges sont les tirages individuels
                   axe 0 materialise')
    plt.grid(True, linestyle='--', alpha=0.5)
    plt.legend()
    plt.show()

    # ---- Graphique 2 : courbes separees en sous-graphiques
    # (chacun avec son axe 0) ----
    fig, axes = plt.subplots(nb_tirages, 1,
                             figsize=(10, 2 * nb_tirages),
                             sharex=True)
    if nb_tirages == 1:
        axes = [axes]

    for i, ax in enumerate(axes):
        ax.plot(n_list, toutes_densites[i], marker='o',
                linestyle='-', color='blue', markersize=3)
        # -- MATERIALISATION DE L'AXE ZERO pour chaque sous-graphique --
        ax.axhline(y=0, color='black', linestyle='--', linewidth=0.8, alpha=0.7)
```

```
        ax.set_ylabel(f'Seed {i}', rotation=0, labelpad=20)
        ax.grid(True, linestyle='—', alpha=0.3)
        ax.set_ylim(bottom=-0.05)    # on met un petit negatif
                                     # pour bien voir la ligne 0

    axes[-1].set_xlabel('n (nombres pairs)')
    fig.suptitle(f'Densite de "a"     {nb_tirages} tirages separes\n'
                 '(chaque sous-graphique a son propre axe avec 0 visible)',
                 fontsize=14)
    plt.tight_layout()
    plt.show()

# ------------------------------------------------------------------------
# 5. EXECUTION
# ------------------------------------------------------------------------
if __name__ == "__main__":
    analyse_densite_separee(N=102, nb_tirages=10)
```