Formalisation du planeprem, Denise Vella-Chemla, juin 2026.

Il s'agit ici de proposer différentes modélisations de systèmes physiques inspirés par le crible d'Ératosthène. Sont proposés trois niveaux de modélisation, allant de l'observation cinématique à la formalisation matricielle.

1) première approche (dynamique) : "utiliser la cinématique d'un tore de phases".

À chaque nombre premier p_k est associé une tige rotative tournant d'une fraction de tour $1/p_k$ à chaque clic d'horloge [1]. Au départ, toutes les tiges sont sur la position angulaire 0. L'utilisation de k tiges correspondant aux k premiers nombres premiers de p_1 à p_k permet de savoir si un nombre inférieur ou égal à $(p_k)^2$ est premier. Si après n clics, aucune roue de nombre premier n'est sur la position 0, n est un nombre premier.

On peut imaginer également le déplacement de particules sur le tore. L'idée est alors de voir si les particules, au bout de n clics, sont revenues à leur position initiale ou pas. Cela revient à projeter géométriquement les différents mouvements circulaires des tiges ci-dessus sur un tore de dimension k et simuler les positions angulaires en les ramenant dans l'intervalle $[0, 1]$ par l'opération $position = (n/p_k) \pmod 1$ pour chaque particule modélisant un nombre premier particulier p_k. Ici, la primalité est vue non comme une simple propriété arithmétique statique, mais comme une signature dynamique émergeant d'un système à plusieurs particules.

2) seconde approche (dynamique) : "utiliser un système d'oscillateurs harmoniques".

À chaque nombre premier p est associé un oscillateur de fréquence $f = 1/n$. On représente les oscillations par des sinusoïdes $\sin\left(\frac{\pi n}{p}\right)$ et le caractère de primalité correspond à un phénomène de résonance : aux nombres composés correspondent des interférences destructrices (une de leurs ondes est sur un nœud), tandis que les nombres premiers correspondent à des interférences constructrices (aucune de leurs ondes n'est sur un nœud). La primalité est une rupture de symétrie : les nombres premiers correspondent aux états dans lesquels le système évite toute interférence constructive avec les modes fondamentaux.

3) troisième approche (structurelle) : "utiliser des opérateurs de permutations".

a) l'espace d'états : soit l'espace des matrices booléennes infinies, structuré en somme directe de blocs circulaires B_p de dimension $p \times p$ pour chaque entier $p \geq 2$.

Soit G l'opérateur de permutation globale défini par la matrice diagonale par blocs :

$$G = \bigoplus_{p=2}^{\infty} C_p$$

où C_p est la matrice de permutation circulaire associée au cycle de taille p.

1. L'idée initiale avait été présentée ici.

b) l'opérateur d'évolution : l'évolution du système est régie par l'élévation à la puissance k de l'opérateur G, notée G^k. Cette transformation permet d'atteindre un état du système représenté par une matrice booléenne dont la structure de blocs est déterminée par les propriétés arithmétiques de k.

c) caractérisation : la primalité d'un entier k est un simple calcul de trace :

- Théorème (propriété invariante) : k est premier si et seulement si $\text{Trace}(G^k) = k$.
- Mécanisme : Pour k premier, seuls les blocs C_p dont la taille divise k contribuent à la trace. Puisque k est premier, seul le bloc de taille k ($p = k$) est activé (le bloc correspondant a tous ses 1 qui sont revenus sur la diagonale).

Le programme d'élévation de la matrice "infinie" à une certaine puissance fournit en outre la somme des diviseurs moins 1 pour les nombres composés. Le voici :

```
import numpy as np

def creer_matrice_G(n):
    """Cree une matrice diagonale par blocs circulaires jusqu'a taille n."""
    G = np.zeros((n, n), dtype=int)
    index = 0
    p = 2
    while index + p <= n:
        bloc = np.eye(p, k=1)
        bloc[p-1, 0] = 1
        G[index:index+p, index:index+p] = bloc
        index += p
        p += 1
    return G

def tester_primalite_matrice(G, k):
    """Calcule la trace de G^k et verifie si elle egale k."""
    G_k = np.linalg.matrix_power(G, k)
    trace_k = np.trace(G_k)
    return trace_k

taille = 45
G = creer_matrice_G(taille)
print(f"{'Nombre (k)':<12} | {'Trace(G^k)':<12} | {'Statut'}")
print("-" * 35)
for k in range(2, 10):
    tr = tester_primalite_matrice(G, k)
    statut = "Premier" if tr == k else "Compos\'{e}"
    print(f"{k:<12} | {tr:<12} | {statut}")
```

L'exécution est très lente : les multiplications matricielles sont très coûteuses. On se lasse, on demande à gemini d'écrire le même programme mais en utilisant des bitsets : chaque ligne matricielle est un entier de grande taille (BigInt), permettant de remplacer les multiplications coûteuses par des opérateurs logiques de bas niveau (parallélisme intrinsèque : le processeur traite 64 bits en une seule opération. Pour une ligne de 64 éléments, le calcul est 64 fois plus rapide ; du point de vue de la mémoire : une matrice 1000×1000 booléenne prend 1 Mo en bits, contre plusieurs Go en float64 ;

complexité : le coût de la multiplication matricielle est divisé par un facteur énorme). En utilisant des entiers arbitrairement longs pour représenter les lignes de l'opérateur, le produit matriciel est transformé en une série d'opérations logiques bit-à-bit, permettant d'atteindre des puissances de k significatives tout en conservant la structure booléenne exacte. Cela reste quand même très lent (262 secondes pour tester la primalité des 50 premiers nombres).

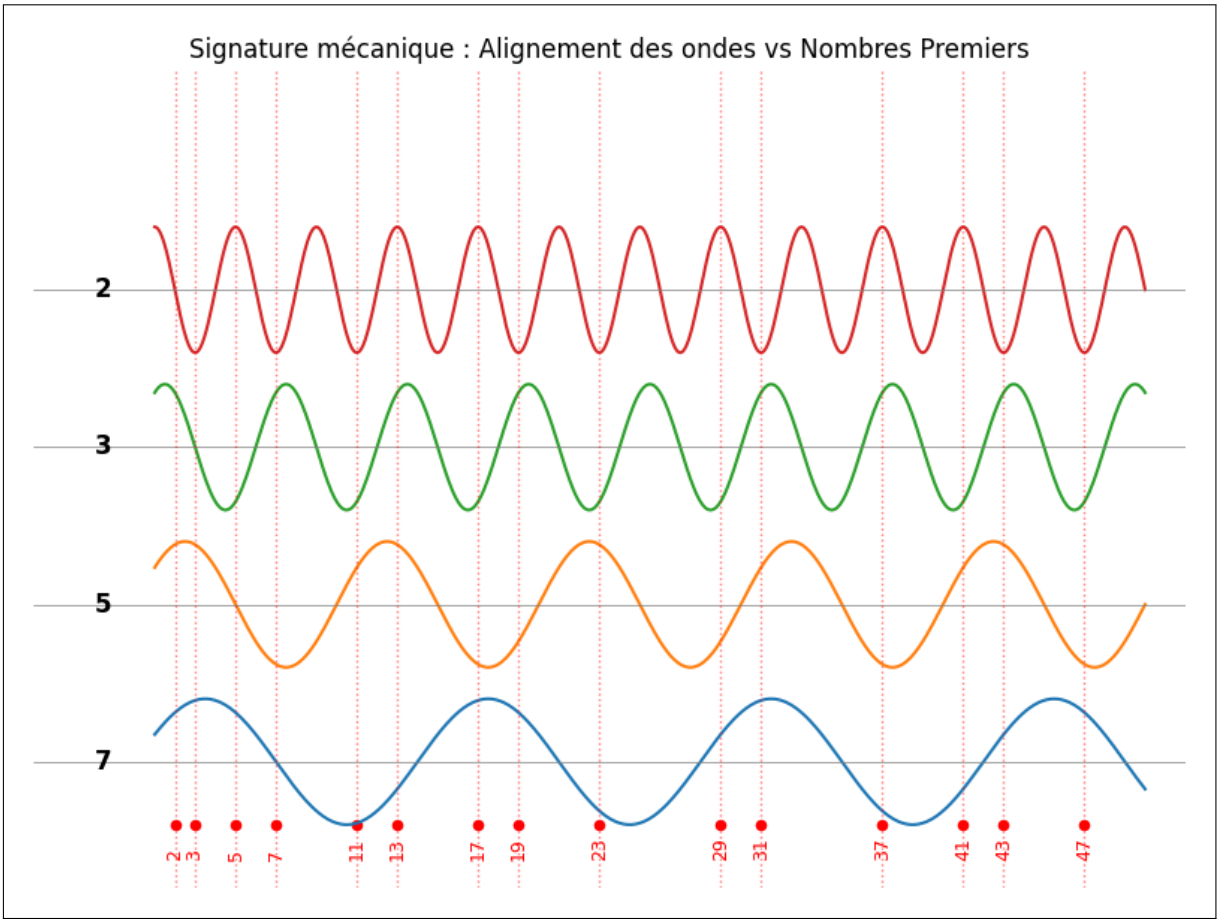
```
import numpy as np
import time

def creer_matrice_G_bits(n_max):
    """
    Cree l'operateur G sous forme de liste d'entiers (bitsets).
    Chaque entier represente une ligne de la matrice.
    """
    taille_totale = sum(range(2, n_max + 1))
    G = np.zeros(taille_totale, dtype=object)
    index = 0
    for p in range(2, n_max + 1):
        # Pour chaque bloc circulaire de taille p
        for i in range(p):
            # Le 1 se deplace a la colonne (i+1)%p
            col = index + ((i + 1) % p)
            G[index + i] = 1 << col
        index += p
    return G, taille_totale

def mult_bool_bits(A, B_t, taille):
    """
    Multiplication de matrices booleennes optimisee.
    B_t est la transposee de B pour un acces rapide aux colonnes.
    """
    C = np.zeros(taille, dtype=object)
    for i in range(taille):
        row_A = A[i]
        # On utilise le bitwise pour multiplier la ligne i
        # par toutes les colonnes
        for j in range(taille):
            if row_A & B_t[j]:
                C[i] |= (1 << j)
    return C

def puissance_k_bits(G, k, taille):
    """Calcule G^k par exponentiation rapide."""
    # Matrice Identite en format bitset
    res = np.array([(1 << i) for i in range(taille)], dtype=object)
    base = G
    while k > 0:
        if k % 2 == 1:
            # Transposee de la base pour l'acces aux colonnes
            base_t = [(sum(1 << i for i in range(taille) if (base[i]
                           & (1 << j)))) for j in range(taille)]
            res = mult_bool_bits(res, base_t, taille)
        # Carre de la base
        base_t = [(sum(1 << i for i in range(taille) if (base[i]
```

```
                            & (1 << j)))) for j in range(taille)]
        base = mult_bool_bits(base, base_t, taille)
        k //= 2
    return res

def tester_primalite_opt(k, n_max):
    G, taille = creer_matrice_G_bits(n_max)
    G_k = puissance_k_bits(G, k, taille)
    # La trace est la somme des bits situes sur la diagonale (1 << i)
    trace = 0
    for i in range(taille):
        if G_k[i] & (1 << i):
            trace += 1
    return trace

tic = time.time()
# Test
nmax = 50
print(f"{'Nombre (k)':<12} | {'Trace calculee'} | {'Statut'}")
print("-" * 40)
for k in range(2, nmax):
    tr = tester_primalite_opt(k, nmax)
    statut = "Premier" if tr == k else "Compose"
    print(f"{k:<12} | {tr:14} | {statut}")
tac = time.time()
print(tac-tic,' s.')
```

Nombre (k)	Trace calculee	Statut
2	2	Premier
3	3	Premier
4	6	Compose
5	5	Premier
6	11	Compose
7	7	Premier
8	14	Compose
9	12	Compose
10	17	Compose
11	11	Premier
12	27	Compose
13	13	Premier
14	23	Compose
15	23	Compose
16	30	Compose
17	17	Premier
18	38	Compose
19	19	Premier
20	41	Compose
21	31	Compose
22	35	Compose
23	23	Premier
24	59	Compose
25	30	Compose
26	41	Compose

27		39	Compose
28		55	Compose
29		29	Premier
30		71	Compose
31		31	Premier
32		62	Compose
33		47	Compose
34		53	Compose
35		47	Compose
36		90	Compose
37		37	Premier
38		59	Compose
39		55	Compose
40		89	Compose
41		41	Premier
42		95	Compose
43		43	Premier
44		83	Compose
45		77	Compose
46		71	Compose
47		47	Premier
48		123	Compose
49		56	Compose

262.77316904067993 s.



Sinusoïdes des oscillateurs harmoniques de l'approche 2