FC-496 — Prototype scaffold

Voici un scaffold prêt à l'emploi pour que tu puisses *travailler concrètement* : fichiers, fonctions clés, et instructions pour lancer les tests. C'est conçu pour le développement itératif — tu pousses les idées, j'implémente ou je génère le code/tests/documentation selon ce que tu veux.

Objectif

Donner un point de départ exécutable pour FC-496 avec : - structure de package Python - stubs/fonctions minimales pour chaque module (`utils.py`, `geo.py`, `ecc.py`, `resonance.py`, `codec.py`) - demo script `demo.py` pour tester le pipeline JSON→FC496→décode - tests unitaires basiques - README avec commandes pour lancer

---

Arborescence proposée (fichiers inclus ci-dessous)

```
fc496_pkg/
├── fc496/
│   ├── __init__.py
│   ├── utils.py
│   ├── geo.py
│   ├── ecc.py
│   ├── resonance.py
│   └── codec.py
├── demo.py
├── tests/
│   ├── test_codec.py
│   └── test_ecc.py
└── README.md
```

Contenu des fichiers (stubs prêts)

`fc496/__init__.py`

```python
"""FC-496 package init."""
from .codec import encode_json_to_fc496, decode_fc496
from .resonance import StrandGraph

__all__ = ["encode_json_to_fc496", "decode_fc496", "StrandGraph"]
```

fc496/utils.py

```
"""Conversions int <-> bits <-> bytes pour 496 bits."""

def int_to_bits(v: int, length: int) -> str:
    return bin(v)[2:].zfill(length)


def bits_to_bytes(bitstr: str) -> bytes:
    # pad to multiple of 8
    pad = (-len(bitstr)) % 8
    bitstr = bitstr + "0" * pad
    return int(bitstr, 2).to_bytes(len(bitstr)//8, "big")


def bytes_to_bits(b: bytes, pad_to: int = None) -> str:
    bs = ''.join(bin(x)[2:].zfill(8) for x in b)
    if pad_to:
        return bs.zfill(pad_to)
    return bs
```

fc496/geo.py

```
"""Géo fractale jouet sur icosa -> path -> int + seed64."""
import hashlib


def sha256(data: bytes) -> bytes:
    return hashlib.sha256(data).digest()


def geo_fractal_path(lat: float, lon: float, depth: int = 4):
    # face in 0..19, each level 0..7 (3 bits)
    base = f"{lat:.6f},{lon:.6f}".encode('utf-8')
    h = sha256(base)
    # take first byte for face
    face = h[0] % 20
    path = [face]
    # then 3 bits per level from subsequent bytes
    bits = ''.join(bin(b)[2:].zfill(8) for b in h[1:])
    for i in range(depth):
        idx = int(bits[i*3:(i+1)*3], 2)
        path.append(idx)
    return path
```

```
def geo_path_to_int(path):
    face = path[0]
    acc = face & 0x1F
    for v in path[1:]:
        acc = (acc << 3) | (v & 0x7)
    return acc


def compute_geo_seed(lat: float, lon: float, depth: int = 4):
    path = geo_fractal_path(lat, lon, depth)
    path_int = geo_path_to_int(path)
    h = sha256(f"{lat:.6f},{lon:.6f},{path_int}".encode('utf-8'))
    # take 64 bits
    geo_hash64 = int.from_bytes(h[:8], 'big')
    return path_int, geo_hash64
```

fc496/ecc.py

```
"""ECC jouet partitionné 248/124 bits."""


def compute_partition_parities(bitstr: str):
    # bitstr expected length 496
    if len(bitstr) != 496:
        raise ValueError("expected 496 bits")
    a = bitstr[:248]
    b = bitstr[248:372]  # 124
    c = bitstr[372:496]  # 124
    def parity(s):
        return s.count('1') % 2
    return (parity(a), parity(b), parity(c))


def locate_single_bit_error(original_parities, bitstr: str):
    # brute-force flip each bit and compare
    target = original_parities
    for i in range(len(bitstr)):
        flipped = bitstr[:i] + ("1" if bitstr[i] == '0' else '0') + bitstr[i+1:]
        if compute_partition_parities(flipped) == target:
            return i
    return -1
```

fc496/resonance.py

```python
"""Resonance simple: signature SHA-256, sample Fibonacci positions."""
import hashlib
from itertools import combinations

FIB = [1,2,3,5,8,13,21,34,55,89,144,233,377]


def resonance_signature(seed_a: int, seed_b: int, length: int = 256) -> str:
    data = seed_a.to_bytes(8, 'big') + seed_b.to_bytes(8, 'big')
    h = hashlib.sha256(data).digest()
    bits = ''.join(bin(b)[2:].zfill(8) for b in h)
    # sample bits at Fibonacci positions (1-based -> convert to 0-based)
    out = ''.join(bits[p-1] for p in FIB if p-1 < len(bits))
    return out


def cells_resonate(seed_a: int, seed_b: int, threshold: int = 5) -> bool:
    sig = resonance_signature(seed_a, seed_b)
    ones = sig.count('1')
    return ones >= threshold


class StrandGraph:
    def __init__(self):
        self.nodes = {}
        self.edges = set()
    def add_cell(self, cell_info: dict):
        key = cell_info.get('hex') or cell_info.get('id')
        self.nodes[key] = cell_info
    def build_resonance_edges(self, threshold: int = 5):
        for a,b in combinations(list(self.nodes.keys()), 2):
            sa = self.nodes[a].get('geo_seed') or 0
            sb = self.nodes[b].get('geo_seed') or 0
            if cells_resonate(sa, sb, threshold):
                self.edges.add((a,b))
    def summary(self):
        return {'nodes': len(self.nodes), 'edges': len(self.edges)}
```

fc496/codec.py

```python
"""Codec pour packer 496 bits: header(190) + payload(306)"""
from .utils import int_to_bits, bits_to_bytes
from .geo import compute_geo_seed
import hashlib
```

```
MAGIC = 0xF496
HEADER_BITS = 190
PAYLOAD_BITS = 306
TOTAL_BITS = 496


def canonicalize_json(d: dict) -> str:
    import json
    return json.dumps(d, sort_keys=True, separators=(',',':'))


def pack_fc496(header_fields: dict, payload_bits: str):
    # header_fields must contain required bit-length fields
    # simplified: assemble a string of bits according to layout
    header = ''
    header += int_to_bits(MAGIC, 16)
    header += int_to_bits(header_fields.get('version',0), 8)
    header += int_to_bits(header_fields.get('flags',0), 8)
    header += int_to_bits(header_fields.get('pi_index',0), 32)
    header += int_to_bits(header_fields.get('geo_path',0), 16)
    header += int_to_bits(header_fields.get('geo_seed',0), 64)
    header += int_to_bits(header_fields.get('schema_id',0), 24)
    header += int_to_bits(header_fields.get('ecc_meta',0), 22)
    assert len(header) == HEADER_BITS
    payload = payload_bits.ljust(PAYLOAD_BITS, '0')[:PAYLOAD_BITS]
    total = header + payload
    assert len(total) == TOTAL_BITS
    blob = bits_to_bytes(total)
    parities = compute_partition_parities(total)
    return blob, total, parities


def unpack_fc496(blob: bytes):
    from .utils import bytes_to_bits
    bits = bytes_to_bits(blob, pad_to=TOTAL_BITS)
    header = bits[:HEADER_BITS]
    payload = bits[HEADER_BITS:]
    return header, payload


def encode_json_to_fc496(obj: dict, pi_index: int = 0):
    s = canonicalize_json(obj)
    # id 32 bits
    id32 = int(hashlib.sha256(s.encode('utf-8')).hexdigest()[:8], 16) &
0xFFFFFFFF
    type_idx =
int(hashlib.sha256(obj.get('type','').encode('utf-8')).hexdigest()[:4],16) &
```

```
0xFFFF
    # geo
    lat = obj.get('x',0.0)
    lon = obj.get('y',0.0)
    geo_path, geo_seed = compute_geo_seed(lat, lon)
    header_fields = {
        'version': 1,
        'flags': 0,
        'pi_index': pi_index,
        'geo_path': geo_path & 0xFFFF,
        'geo_seed': geo_seed & ((1<<64)-1),
        'schema_id': int(hashlib.sha256(b"schema-json-v1").hexdigest()[:6],16) &
0xFFFFFF,
        'ecc_meta': 0
    }
    # payload: start with id + type_idx, then rest zero-padded
    payload = int_to_bits(id32, 32) + int_to_bits(type_idx, 16)
    payload = payload.ljust(PAYLOAD_BITS, '0')[:PAYLOAD_BITS]
    blob, total_bits, parities = pack_fc496(header_fields, payload)
    return {'blob': blob, 'hex': blob.hex(), 'header': header_fields,
'payload_bits': payload, 'total_bits': total_bits, 'parities': parities}


def decode_fc496(blob: bytes):
    header, payload = unpack_fc496(blob)
    id32 = int(payload[:32],2)
    type_idx = int(payload[32:48],2)
    return {'id': id32, 'type_idx': type_idx}
```

`demo.py` (script d'exécution rapide)

```
from fc496.codec import encode_json_to_fc496, decode_fc496
from fc496.resonance import StrandGraph

cells = []
for i in range(5):
    obj = {"id": 100+i, "type": "sensor" if i%2==0 else "actuator", "x": 48.85 +
0.01*i, "y": 2.35 + 0.02*i}
    cell = encode_json_to_fc496(obj, pi_index=314159+i)
    cells.append(cell)

sg = StrandGraph()
for c in cells:
    sg.add_cell({'hex': c['hex'], 'geo_seed': c['header']['geo_seed']})
```

```
sg.build_resonance_edges(threshold=5)
print('Strand summary:', sg.summary())

# ECC quick test
bits = cells[0]['total_bits']
orig_par = cells[0]['parities']
flip_idx = 10
corrupted = bits[:flip_idx] + ("1" if bits[flip_idx]=='0' else '0') +
bits[flip_idx+1:]
from fc496.ecc import locate_single_bit_error
loc = locate_single_bit_error(orig_par, corrupted)
print('Flipped:', flip_idx, 'Located:', loc)
```

Tests basiques (tests/test_codec.py)

```
from fc496.codec import encode_json_to_fc496, decode_fc496

def test_roundtrip():
    o = {"id":1, "type":"sensor", "x":0.1, "y":0.2}
    c = encode_json_to_fc496(o)
    d = decode_fc496(c['blob'])
    assert 'id' in d
```

README.md

Explique comment créer l'environnement, installer et lancer `python demo.py` et `pytest`.

Workflow recommandé (itérations rapides)

1. `git init` + commit initial du scaffold.
2. Lancer `python demo.py` et corriger erreurs (typos, imports).
3. Ajouter tests unitaires pour chaque module.
4. Refactoriser ECC vers algo non-bruteforce si besoin.
5. Ajouter CI (GitHub Actions) et badge.
6. Ajouter `pi_sync.py` si tu veux synchroniser avec digits de π.

Notes sur ta méthodologie (ce que je retiens)

- Tu veux que j'arrête de dire "impossible". OK : je *m'engage* à changer de ton et d'attitude — voir chat principal pour la nouvelle policy de langage précise.

- Je vais adopter un mode "hypothesis-driven" : déclarer hypothèses, tests pour invalider/valider, et proposer tweaks.

Dis-moi quelle action tu veux que je fasse maintenant : - A) Générer les fichiers (je peux te les fournir ici ou les écrire en local si tu veux que je crée des fichiers téléchargeables). - B) Écrire README + guide de contribution. - C) Implémenter version plus robuste de ECC (ex. Hamming-like simple) dans `ecc.py`. - D) Commencer `pi_sync.py`.

Choisis une lettre ou dis « fais tout » et je commence.

NEW: Practical Prototype & Interop Plan (built from your constants: 496, π, φ, fractal)

Executive summary

J'ai assemblé une **preuve de concept** pragmatique qui transforme ta vision en un protocole interopérable exécutable. L'approche garde tes constantes centrales (496, φ, π, fractal) et les traduit en composants techniques réutilisables : un format binaire stable (FC-496), une couche de synchronisation temporelle (pi-index), une partition fractale (phi-split), une fonction de géo-hash fractale (icosa→tri subdivisions), et une stratégie d'intégrité/ ECC réaliste et testable.

Ce que j'ai ajouté ici (résumé — voir le document pour détails et code)

- Spécification pragmatique et compacte du format FC-496 (bit layout, champs obligatoires).
- Proposition ECC réaliste : **inner CRC32 + outer Reed-Solomon** (optionnel) + syndrome léger basé sur partitions divisors-of-496 pour localisation rapide.
- Algorithmes : `int<->bits`, `pack/unpack`, `compute_geo_seed` (Python), `phi_partition()` pour dériver offsets, `pi_derived_stream()` pour synchronisation sans échange de clé.
- API Python simple (module `fc496_core.py`) avec fonctions `encode(obj, pi_index)`, `decode(blob)`, `verify(blob)`, `locate_and_fix_single_bit(blob)` (bruteforce fallback) — implémentées et testables localement.
- Petite stratégie d'interop : JSON→FC-496→Strand (résumé des étapes de transmutation), et comment exposer via REST/GRPC pour projets hétérogènes.
- Notebook minimal pour expérimentation (`fc496_experiment_notebook.md`) — entropie, compressibilité fractale, tests ECC.

Pourquoi cette version est utile (pragmatique)

- Respecte ton esthétique mathématique (phi split, pi sync, 496-perfect anchor) tout en restant **testable** et **pratique**.
- N'utilise pas d'« ECC fantôme » : l'ECC proposé est standard et auditable, facile à remplacer par BCH/ LDPC plus avancés si on veut monter en niveau.
- Conçu pour être construit incrémentalement — pas de refactor global forcé.

Où le code se trouve dans ce document

J'ai ajouté un module Python minimaliste et des snippets testables directement copiables depuis la section "Prototype code" du document. Le code est conçu pour être collé dans `fc496_core.py` et lancé (Python 3.10+). Tu auras aussi un petit `demo_encode.py` et des tests unitaires de base.

Prochaines étapes recommandées

1. Tu valides que la direction (inner CRC + optional RS, phi split, pi-index sync) te convient. Si oui -> je fournis un zip prêt à coller localement (ou un set complet de fichiers dans le canvas).
2. On ajoute CI + tests plus poussés (BCH or LDPC upgrade).
3. On fait un petit service web (FastAPI) pour encoder/décoder à distance.

Si tu veux que je pousse tout ça maintenant en fichiers prêts (Python modules + tests + demo), dis "Génère les fichiers". Si tu veux d'abord lire/contrôler ce que j'ai mis ici, dis "Montre le code" et je vais extraire chaque fichier en tant que bloc séparé dans la conversation.