

## Plan du document

Je vais suivre ce plan :

### **I — Contexte général**

1. Origine du projet (fichiers du .zip)
2. Architecture globale du système
3. Objectif final des modules

### **II — Feature Engineering System (inclus dans le .zip)**

1. Le FeatureEngine
2. Modules :
  - Shitcoin Module
  - BTC Heston Module
  - Sentiment Module
  - Generic Market Module
3. Fusion des features
4. Pourquoi c'est structuré comme ça

### **III — StateBuilder (normalisation + stacking)**

1. Pourquoi normaliser online
2. Comment le buffer temporel fonctionne
3. Sortie finale pour le RL

### **IV — Environnement RL (TradingEnv)**

1. Rôle
2. Structure interne
3. `Step()` et reward simple
4. Limitations de la version initiale

### **V — Agent PPO (inclus dans le .zip)**

1. Acteurs + Critic
2. Log-probabilities
3. GAE + update
4. Ce que PPO apprend dans ce contexte

## **VI — Simulateur de marché (`simulate_market`)**

1. Structure
2. Limites (car fake)
3. Pourquoi destiné à être remplacé

## **VII — Le training complet (`train_ppo.py`)**

1. Architecture end-to-end
2. Ce que l'apprentissage apprend réellement
3. Limites de l'entraînement sur données simulées

## **VIII — Remplacement par données réelles (toutes les briques avancées que tu as demandées)**

1. Loaders Binance
2. Loader Deribit
3. Stockage en `.npz`
4. Alignement temporel Binance/Deribit
5. `RealMarketData`
6. Sentiment réel (squelette)
7. `RewardEngine` financier

## **IX — L'inverseur Heston (partie la plus fondamentale)**

1. Pourquoi `DummyHestonInverse` est une béquille
2. Entraînement synthétique (`train_inverse_heston.py`)
3. Dataset synthétique
4. Fine-tuning réel (`train_heston_real.py`)
5. Entraînement mixte (real + synthetic)

## **X — Pricing et Calibration (Heston pricer)**

1. Char function différentiable
2. Pricing Heston via intégration
3. Calibration via gradient descent

## **XI — Comment tout s'assemble**

1. Flux complet réel
2. Flux complet RL
3. Flux training inverseur
4. Ce que tu dois modifier pour faire tourner en réel

## XII — Conclusion

1. Ce que tu possèdes maintenant
2. Ce que tu dois faire ensuite
3. Comment t’orienter pour la version production

Je vais commencer maintenant. Ce sera le **Chapitre I**, puis j’enchaînerai progressivement. Tu me dis simplement “continue” à la fin de chaque message.

## I — CONTEXTE GÉNÉRAL DU PROJET

Ce chapitre résume exactement **d’où vient le projet, pourquoi** je t’ai donné un `.zip`, et **ce qu’il contient**, pour que tu comprennes clairement la suite.

### 1. Origine : tu voulais un framework RL complet avec Heston

Tu m’as demandé :

- **Un projet complet,**
- **en C++ ou Python,**
- **prêt à être lancé,**
- avec **Feature Engineering,**
- **StateBuilder,**
- **Environnement RL,**
- **Agent PPO,**
- **Simulateur,**
- **et le zip prêt à télécharger.**

Je t’ai donc construit **un framework entier**, à la manière d’un “internal quant research starter-kit” :

```
heston_rl_trader/  
models/  
features/  
data/  
env/  
rl/  
backtester.py  
live_trading.py  
train_ppo.py
```

Ce dépôt reprenait les éléments nécessaires pour faire tourner un agent RL **sur des données simulées**, avec :

- Inverse Heston modèle (mais dummy),
- Feature engineering avancé,

- RL PPO avec GAE,
  - Environnement de trading minimisé.
- C'était **la version prototype**.

## 2. L'envie d'aller vers du "réel"

Tu as ensuite demandé :

"Est-ce que je peux trouver ça sur GitHub?"

Réponse : non, c'est un framework custom. Et tu voulais **le transformer en pipeline réel** :

- loader Binance (spot, perp, funding, OI),
- loader Deribit (surface IV réelle),
- dataset IV réel,
- sentiment réel,
- reward réaliste,
- inverseur Heston entraîné (pas dummy).

Donc je t'ai donné **tout ce qu'il faut** pour transformer l'architecture du .zip en système réel type hedge-fund.

## 3. L'objectif final (ce que ton système deviendra)

Au final, ton projet est :

**Un framework qui te permet :**

- D'extraire des features complexes (Heston, sentiment, market micro-structure, IV surface),
- De reconstruire un état RL normalisé,
- De simuler ou trader réellement BTC ou shitcoins,
- De prendre des décisions basées sur un agent PPO,
- De calibrer Heston en différentiable,
- De fine-tuner un inverseur Heston sur données réelles.

C'est **un environnement total**, combinant :

- Volatility engineering,
- Derivatives microstructure,
- Market data ingestion,
- Time-series synthesis,
- Machine learning (deep),
- Reinforcement Learning,
- Stochastic volatility modeling.

Ce n'est pas un repo pour débutant. C'est un "quant lab kit".

#### 4. Pourquoi l'inverseur Heston est le pivot

Ce projet ne sert à rien sans un inverseur Heston fonctionnel.

Tu as :

- un pipeline RL complet,
- un loader Binance,
- un scraper Deribit,
- un StateBuilder,
- un RewardEngine.

Mais le **signal clé** démarre ici :

IV surface (réelle ou synthétique)

↓

Inverse Heston

↓

Paramètres ( $\kappa$ ,  $\theta$ ,  $\sigma$ ,  $\rho$ ,  $v_0$ )

↓

FeatureEngine

↓

StateBuilder

↓

RL agent

C'est le **tuyau central** qui donne la vision du régime de volatilité.

Tu dois donc :

1. Pré-entraîner l'inverseur sur données synthétiques.
2. Le fine-tuner sur surfaces Deribit réelles.
3. Optionnel : faire un "mixed training".

D'où les trois fichiers :

- `train_inverse_heston.py`
- `train_heston_real.py`
- `mixed_heston_trainer.py`

#### 5. Ce que je vais faire maintenant

Je vais te faire la **synthèse totale** du framework, **point par point**, dans l'ordre logique, avec des explications complètes.

## 1 II — FEATURE ENGINEERING SYSTEM (définition, structure, rôle, code, usage)

*(Chapitre massif – tu dois le comprendre parfaitement, car tout ton RL dépend de lui.)*

---

## 1.1 II.1 — Pourquoi un FeatureEngine ?

Ton agent RL **ne peut pas traiter directement** :

- les prix bruts
- les volumes bruts
- les surfaces IV
- le sentiment
- la profondeur de carnet
- les paramètres Heston
- les deltas de paramètres
- les signaux dérivés
- les signaux shitcoin
- les signaux BTC

Un RL **ne comprend rien** à des inputs hétérogènes. Il te faut une couche organisée, modulaire, qui :

1. **Récupère les données brutes** (prix, IV, funding...)
2. **Récupère les sorties Heston** (inverseur)
3. **Construit des features numériques**
4. **Fusionne tout dans un seul vecteur**
5. **Normalise, stacke, envoie au RL**

C'est exactement ce que fait `FeatureEngine`.

Le plus important :

**LL'ensemble du RL ne voit JAMAIS de données "brutes".**

Tout passe par des modules spécialisés.

---

## 1.2 II.2 — Structure du FeatureEngine

Dans le dépôt, tu as :

`features/`

```
feature_engine.py
state_builder.py
```

`feature_engine.py` contient :

- L'interface `FeatureModule`
  - La classe `FeatureEngine`
  - 4 modules :
    1. `ShitcoinFeatureModule`
    2. `BtcHestonFeatureModule`
    3. `SentimentFeatureModule`
    4. `GenericMarketFeatureModule`
-

### 1.3 II.3 — FeatureModule (interface)

```
class FeatureModule(abc.ABC):
    @abc.abstractmethod
    def compute_features(self, context: Dict[str, Any]) -> Dict[str, float]:
        raise NotImplementedError
```

→ Chaque module doit prendre un petit dictionnaire **context** et **retourner un dictionnaire de features numériques**.

**Important** : chaque feature doit être un **float** propre, pas un **tensor**.

---

### 1.4 II.4 — FeatureEngine (le chef d'orchestre)

```
class FeatureEngine:
    def __init__(self, modules: Dict[str, FeatureModule]):
        self.modules = modules
        self.feature_order = None

    def compute_features(self, context):
        merged = {}
        for name, module in self.modules.items():
            feats = module.compute_features(context[name])
            merged[f"{name}.{k}"] = v for k,v in feats.items()

        if self.feature_order is None:
            self.feature_order = sorted(merged.keys())

        vec = np.array([merged[k] for k in self.feature_order], dtype=np.float32)
        return vec, merged
```

**Points cruciaux :**

- Fusion des features = simple concaténation triée.
  - Ordre des features figé à la première exécution → stable.
  - Le RL reçoit un **vecteur 1D**.
- 

### 1.5 II.5 — MODULE 1 : ShitcoinFeatureModule

C'est celui qui te donne :

- features statistiques (vol, skew, kurt)
- features de volume / funding
- features Heston pseudo-surface
- deltas des paramètres Heston

## Pourquoi un “pseudo-surface” pour shitcoin ?

Parce qu’un shitcoin n’a **pas d’options** → donc pas de surface IV.

Alors on fabrique une surface artificielle :

1. On prend des retours glissants.
2. On découpe en fenêtres (ex : 3, 10, 30 minutes).
3. On calcule les moments :
  - moyenne
  - variance
  - skew
  - kurt
4. On met tout ça dans une pseudo-matrice  $[M, 4]$ .
5. On l’envoie dans ton inverseur Heston (qui voit juste une “surface”).
6. Il te sort **des paramètres Heston cohérents**.

Tu crées donc un **embedding de régime** pour un actif sans options.

---

## 1.6 II.6 — MODULE 2 : BtcHestonFeatureModule

Celui-là :

- Charge la surface IV réelle (ou simulée)  $[NK, NT]$
- Normalise
- Donne :
  - paramètres Heston calibrés (via inverseur)
  - deltas Heston
  - ATM IV
  - slope du smile
  - basis
  - funding
  - OI
  - realized vols spot

C’est le bloc le plus important du feature engineering :

**il capture la structure complète du marché BTC.**

---

## 1.7 II.7 — MODULE 3 : SentimentFeatureModule

Module simple :

```
class SentimentFeatureModule(FeatureModule):  
    def compute_features(self, context):  
        return {key: float(v) for key,v in context.items()}
```

→ Tu remplaces le **context** par ton provider réel Twitter/Telegram.

---



## 1.8 II.8 — MODULE 4 : GenericMarketFeatureModule

Ajoute :  
— OHLC  
— volume  
— éventuellement volatilité courte  
— spreads, etc.

---

## 1.9 II.9 — Fusion finale

Après tous les modules  $\rightarrow$  `FeatureEngine.compute_features()` renvoie :

```
obs_vec = [ shitcoin.ret_mean,
             shitcoin.realized_vol,
             shitcoin.rho_s,
             ...,
             btc.theta_s,
             btc.atm_iv_short,
             btc.funding_rate,
             ...,
             sentiment.sentiment_score,
             ...,
             generic.close,
             generic.volume,
             ...
           ]
```

Tu obtiens un vecteur **dimension D** (~50-300 selon additions).  
C'est cet unique vecteur qui part dans :

`StateBuilder`  $\rightarrow$  RL agent

---

## 1.10 II.10 — Pourquoi cette architecture est “pro”

Car elle sépare :  
— l'ingestion  
— la transformation  
— la normalisation  
— la mémorisation temporelle  
— la décision RL  
C'est exactement ce que tu trouves dans :

- Jane Street (signaux  $\rightarrow$  features  $\rightarrow$  models)
- Jump Trading
- Optiver
- Citadel
- G-Research
- Tower Research

Cette intégration modulaire est **scalable**, répliquable, traçable.

## 2 III — STATE BUILDER (normalisation + stacking temporel)

*(C'est le deuxième pilier absolu de ton pipeline après le FeatureEngine.)*

Le FeatureEngine produit un **vecteur de features** de dimension D :

```
feat_vec = np.array([f1, f2, ..., fD])
```

Mais un RL ne travaille **jamais** sur un seul instant t. Il travaille sur une **séquence** récente (temporal context), tout comme un trader.

Le StateBuilder convertit donc :

- le vecteur brut  $\rightarrow$  vecteur normalisé
- puis  $\rightarrow$  fenêtré (stacking temporel)
- puis  $\rightarrow$  état final [window  $\times$  dim]

Le RL consomme alors un état structuré.

---

### 2.1 III.1 — Pourquoi normaliser online (RunningStats)

Dans le StateBuilder :

```
class RunningStats:
    def update(self, x):
        ...
    @property
    def mean(self): ...
    @property
    def std(self): ...
```

Le but : **normaliser sur la distribution réelle du flux**, pas sur un scaler figé.

Conséquences :

- Quand les marchés bougent, ta normalisation se met à jour.
- Si une feature a un régime shift (ex : funding qui explose), le RL ne s'effondre pas.
- Tu n'as pas besoin d'estimer une stat globale au préalable.

Sans normalisation online, ton RL diverge en 2 minutes.

---

## 2.2 III.2 — Normalizer (z-score + clipping)

```
normed = (x - mean) / std
```

**clipping :**

```
np.clip(normed, -clip_value, clip_value)
```

Pourquoi ?

— éviter les spikes

— éviter les outliers

— éviter que le RL “pète un plomb”

Tu utilises un `clip_value=5.0`, ce qui est très standard.

---

## 2.3 III.3 — StateBuffer (mémoire temporelle)

C’est un buffer circulaire de shape :

```
[window, dim]
```

Exemple :

`window = 16, dim = 80` → state final =  $16 \times 80 = 1280$  valeurs.

**Pourquoi un buffer circulaire ?**

— plus efficace

— pas besoin de re-shift les arrays

— stable dans le temps

— toujours rempli dans le même ordre

Il reconstruit une séquence ordonnée du plus ancien au plus récent :

```
t-15
```

```
t-14
```

```
...
```

```
t-1
```

```
t
```

Le RL voit ainsi la **structure temporelle des features**.

---

## 2.4 III.4 — StateBuilder = Normalizer + StateBuffer

```
class StateBuilder:
    def build_state(self, feature_vec):
        normed = self.normalizer.normalize(feature_vec, update_stats=True)
        self.buffer.push(normed)
        return self.buffer.get_state()
```

Résumé :

1. Prend un vecteur brut
2. Le normalise
3. Le pousse dans un buffer temporel
4. Retourne la séquence [window, dim]

**C'est cette séquence que PPO consomme.**

Sans StateBuilder → ton RL ne fait rien.

## 3 IV — ENVIRONNEMENT RL (TradingEnv)

*(C'est la troisième brique critique : le RL interagit avec un environnement comme un trader avec un marché.)*

Tu as maintenant :

- FeatureEngine → transforme les données brutes en features intelligents
- StateBuilder → crée un état temporel normalisé
- PPO Agent → prend des actions

L'environnement RL fait le lien entre :

Policy(action) ↔ TradingEnv

Il :

- construit l'observation,
- applique l'action,
- calcule le reward,
- déplace le marché d'un pas,
- retourne 'obs, reward, done, info'.

---

### 3.1 IV.1 — Structure générale de l'environnement

Ton fichier :

env/trading\_env.py

Définit :

```
class TradingEnv(gym.Env):  
    ...
```

Ce qui implique :

- compatibilité stable avec gymnasium/gym RL
  - tu peux plugger PPO/SAC/TD3/etc.
  - support immédiat pour des implémentations RL existantes
- 

### 3.2 IV.2 — Le rôle exact de l’environnement

Il ne s’agit **pas** d’une simulation de marché réaliste. L’environnement te donne 3 fonctions essentielles :

#### 1. reset()

- réinitialise l’environnement
- met le temps  $t$  à un index défini
- remet le capital et la position à zéro
- remet le StateBuilder à zéro
- renvoie la première observation (state)

#### 2. step(action)

- applique l’action du RL
- modifie la position
- calcule les coûts
- avance le temps  $t \rightarrow t+1$
- calcule le reward
- renvoie :
  - nouvelle observation
  - reward
  - done/truncated
  - informations debug

#### 3. render() (optionnel, souvent inutilisé)

---

### 3.3 IV.3 — Détaillons step()

Le coeur :

```
action [-1,1] # proportion du capital exposée long/short
```

Exemple :

— +1 → 100% long

— -1 → 100% short

— 0 → neutre

Ensuite l'environnement calcule :

### 1. Position cible

```
target_position_value = a * equity  
target_position = target_position_value / price
```

### 2. Coût de transaction

```
cost = transaction_cost * abs(target_position - old_position) * price
```

### 3. Mise à jour cash et position

### 4. Calcul du nouveau capital (equity)

```
5. reward = (equity_now - equity_prev) / capital_initial
```

---

## 3.4 IV.4 — Le reward initial est “simple”

Ce reward dans la version zip = **PnL instantané** / `capital_initial`.

C'est **volatil** et **pas risk-adjusted**.

C'est pour ça que je t'ai construit ensuite un **RewardEngine avancé**, plus proche du réel.

---

## 3.5 IV.5 — `_get_state()` : lien avec FeatureEngine + State-Builder

Méthode clé :

```
feat_vec, feat_dict = self.fe.compute_features(context)  
state = self.sb.build_state(feat_vec)  
flat = state.reshape(-1)  
return flat
```

Donc l'ordre d'exécution :

Données brutes (prix, IV, funding, surface, etc.)

↓

FeatureEngine

↓ vecteur 1D

```
StateBuilder
  ↓ matrice [window × dim]
flatten
  ↓ vecteur RL-ready
env.step() ← renvoie obs au PPO
```

L'environnement ne contient **aucune** logique financière complexe. Il est un **wrapper** autour de ton moteur de features.

---

### 3.6 IV.6 — Le rôle crucial du context

Pour chaque `t`, l'env construit :

```
context = {
  "shitcoin": {...},
  "btc": {...},
  "sentiment": {...},
  "generic": {...},
}
```

Chaque module lit **uniquement son sous-context**.

C'est **ultra propre** et garantit :

- isolation des modules
  - extensibilité
  - debug facile
  - remplacements possibles sans casser le reste
- 

### 3.7 IV.7 — État complet résumé

Dans `TradingEnv`, tu as :

- accès aux séries (simulées ou réelles)
- un `FeatureEngine` modulaire
- un `StateBuilder`

Les observations du RL reflètent :

- structure vol BTC via inverseur Heston
- structure shitcoin via pseudo-Heston moments
- sentiment
- ohlcv
- funding / OI
- realized vol
- basis futures/spot
- deltas Heston

**C'est un état riche, multi-modèles, multi-niveaux.**

## 4 V — PPO AGENT (POLICY + VALUE FUNCTIONS + GAE + OPTIMISATION)

*(Le “cerveau” de ton système. C’est lui qui apprend la politique de trading.)*

Ton code PPO est un PPO minimaliste, propre, inspiré de SpinningUp d’OpenAI, et totalement compatible avec ton environnement.

Ce chapitre va te permettre de comprendre **exactement ce que fait ton PPO**, comment il apprend, et pourquoi il est adapté à ce pipeline.

---

### 4.1 V.1 — Architecture globale du PPO

Ton agent PPO :

- reçoit `obs` (état RL)
- produit une action `a`
- estime la valeur `v(s)`
- met à jour sa politique et sa value function
- corrige les dérives via clipping

Le code :

`rl/ppo_agent.py`

Contient :

- `PPOConfig`
  - `MLP`
  - `PPOAgent`
  - `compute_gae`
- 

### 4.2 V.2 — Le réseau de POLICY

```
self.pi_net = MLP(obs_dim, 1)
```

→ MLP simple qui prend tout l’état [`window × dim`] et renvoie **une moyenne d’action** ( $\mu$ ), donc une action dans <sup>1</sup>.

#### Bruit d’exploration

Tu as :

```
self.log_std = nn.Parameter(torch.zeros(1,1))
```

→ l’action réelle est :



```
a = mu + exp(log_std) * noise
```

Le bruit est essentiel en PPO pour explorer différentes expositions.

---

### 4.3 V.3 — Le réseau de VALUE

```
self.v_net = MLP(obs_dim, 1)
```

Renvoie la valeur estimée :

$V(s)$  = estimation du discounted reward futur

Cette estimation sert à calculer :  
— l'avantage (Advantage Function)  
— la perte du critic

---

### 4.4 V.4 — Distribution des actions

```
dist = Normal(mu, std)
a = dist.sample()
logp = dist.log_prob(a).sum()
```

Le RL enregistre ces log-probabilités pour l'étape d'optimisation.

---

### 4.5 V.5 — Calcul de l'Avantage : GAE (Generalized Advantage Estimation)

Partie hyper importante.  
Ton code :

```
adv, ret = compute_gae(rews, vals, done, gamma, lam)
```

GAE sert à estimer :

Advantage = Quality of action  $a_t$  wrt baseline  $V(s_t)$

La version complète :

```
delta_t = r_t + V(s_{t+1}) - V(s_t)
adv_t = delta_t + adv_{t+1}
```

GAE donne un avantage plus stable et moins bruité que la méthode naïve.

---

## 4.6 V.6 — Update Policy (the PPO objective)

Objectif PPO :

```
maximize E[ min( r_t * Adv_t , clip(r_t, 1-,1+) * Adv_t ) ]
```

où :

```
r_t = exp(logp_new - logp_old)
```

Le code :

```
ratio = torch.exp(logp - logp_old)
clip_adv = torch.clamp(ratio, 1-clip_ratio, 1+clip_ratio) * adv
loss_pi = -min(ratio*adv , clip_adv)
```

→ Cela empêche la politique de changer trop vite (“policy collapse”).

---

## 4.7 V.7 — Update Value Function

Objectif :

```
loss_v = (V(s_t) - ret_t)^2
```

Rien de compliqué ici, c’est du MSE.

---

## 4.8 V.8 — Training loop (dans train\_ppo.py)

Dans ton fichier :

- On collecte des rollouts (rollout\_len=2048)
- On stocke :
  - obs
  - acts
  - logprobs
  - rewards
  - values
  - dones

Ensuite :

- on appelle `agent.update(buf)`
  - PPO se met à jour
  - on boucle jusqu’à `total_steps`
-

## 4.9 V.9 — Ce que PPO apprend réellement dans ton setup

Il apprend une politique d'exposition en fonction :

- de la dynamique vol shitcoin
- de la structure IV BTC (via inverseur Heston)
- des signaux sentiment
- des signaux microstructure
- des réalisations passées (fenêtre temporelle)

Et donc il apprend :

- **quand être long**,
  - **quand être short**,
  - **quand réduire**,
  - **quand augmenter**,
  - **comment équilibrer le risque**,
  - **comment réagir aux régimes volatiles**.
- 

## 4.10 V.10 — Limites du PPO actuel

Tu dois connaître les limites :

- reward basique = pas risk-adjusted
- simulateur = trop simple
- densité d'information du state = fixe
- horizon = court
- bruit des actions = isotrope (pas conditionnel)

C'est pour ça que je t'ai donné ensuite :

- un RewardEngine complet
- un simulateur réel
- des signaux réels
- un inverse Heston solide

Ce PPO est **suffisant pour débiter** mais tu peux le remplacer par :

- SAC
- TD3 (si actions continues)
- PPO avec LSTM
- PPO avec attention
- PPO multi-actions (hedge + vol + delta)

Mais pour l'instant, c'est parfaitement suffisant.

## 5 VI — SIMULATEUR DE MARCHÉ (simulate\_market) : À QUOI IL SERT ET POURQUOI TU DOIS LE TUER UN JOUR

Ce simulateur est dans :

`data/simulated_data.py`

C'est lui qui fabrique les séries :

- BTC prices
- shitcoin prices
- funding
- volumes
- OI
- IV surface “fake” BTC

Son rôle : te donner **un environnement complet fonctionnel**, sans dépendre d'aucune API, pour que tu puisses :

- lancer `train_ppo.py`
- vérifier l'architecture complète (FeatureEngine → StateBuilder → PPO → Env)
- débbugger rapidement

Tu ne dois pas le prendre pour plus que ce qu'il est : **une maquette**.

---

## 5.1 VI.1 — Structure du simulateur

Dans `SimMarketConfig`, tu spécifies :

```
@dataclass
class SimMarketConfig:
    n_steps: int = 10_000
    dt: float = 1.0
    mu_btc: float = 0.0
    mu_shit: float = 0.0
    vol_btc: float = 0.02
    vol_shit: float = 0.05
    seed: int = 123
```

Puis `simulate_market(config)` :

1. Crée `btc_prices` par un **Geometric Brownian Motion** (GBM) :

$$S_{t+1} = S_t \exp \left[ \left( \mu - \frac{1}{2} \sigma^2 \right) \Delta t + \sigma \sqrt{\Delta t} Z_t \right]$$

2. Crée `shit_prices` de la même manière mais avec vol plus élevée.
3. Génère :
  - `shit_volumes` via log-normal
  - `shit_funding` via normal
  - `btc_fut_prices` ~ `btc_spot` \* (1 + 0.001 \* noise)
  - `btc_funding` via normal

- `btc_oi` via log-normal
  - 4. Crée une surface IV “fake” :
    - `base_iv = vol_btc`
    - ajoute un **smile** en fonction de `k` (log-moneyness)
    - ajoute une term-structure en fonction de `T`
- Résultat : `btc_iv_surface[t, i, j]`  
où `t` = temps, `i` = strike, `j` = maturité.
- 

## 5.2 VI.2 — Ce que la surface IV “fake” contient

La surface IV est :

```
btc_iv_surface[t, i, j] = base_iv + 0.2*|k_i| + 0.1*log(1+T_j)
```

→ Elle a :

- un skew symétrique (via `|k|`)
- une term-structure monotone (via `log(1+T)`)
- mais **aucun lien avec la dynamique simulée** de `S`

C’est un décor. Suffisant pour que l’inverseur Heston voie une “géométrie” cohérente, mais pas un modèle de marché réaliste.

---

## 5.3 VI.3 — Pourquoi ce simulateur est quand même utile

Malgré ses limitations, il te permet de :

- vérifier que le `FeatureEngine` reçoit bien des surfaces IV,
- tester que le `BtcHestonFeatureModule` fonctionne (Heston-inverse + ATM + slope),
- tester que le `ShitcoinFeatureModule` fonctionne (pseudo-surface + embedding),
- tester que le `StateBuilder` gargouille correctement,
- tester que `TradingEnv` + PPO tournent sans bug.

C’est ton **sandbox “dry run”**.

---

## 5.4 VI.4 — Pourquoi tu dois le remplacer sur du vrai travail

Tu ne peux pas :

- valider une stratégie,
- analyser la robustesse,
- mesurer la performance,
- calibrer Heston sérieusement,

- étudier des régimes volatiles réels, sur ce simulateur.
- Tu dois le remplacer par :
  - `RealMarketData` (Binance + Deribit)
  - ou ton propre loader de fichiers historiques (CSV/HDF/Parquet).
- Et c'est pour ça que je t'ai donné :
  - `binance_loader.py`
  - `deribit_loader.py`
  - `real_market_loader.py`
  - `alignment.py`
- pour que tu puisses passer du fake → réel. C'est le script qui orchestre tout :
  - crée les données
  - instancie `FeatureEngine` + `StateBuilder`
  - instancie l'environnement
  - instancie PPO
  - lance la boucle d'entraînement RL
  - calcule un backtest simple (`equity_curve` + stats)
- C'est le "main" de ton framework.

---

## 5.5 VII.1 — Pipeline global de `train_ppo.py`

Dans la version fournie (avec simulateur), la structure est :

```
def train_ppo(...):

    # 1) Market (simulé ou réel)
    market = simulate_market(...)

    # 2) Device
    device = torch.device("cuda" ...)

    # 3) Inverseurs Heston (dummy ou réels)
    btc_model = load_heston_inverse_model(...)
    shit_model = load_heston_inverse_model(...)

    # 4) FeatureEngine
    fe = create_default_feature_engine(shit_model, btc_model, device)

    # 5) Dimension des features
    dummy_vec, _ = fe.compute_features(dummy_context)
    dim = dummy_vec.shape[0]
```

```

# 6) StateBuilder
sb = StateBuilder(dim=dim, window=16, ...)

# 7) Environnement TradingEnv
env = TradingEnv(market, fe, sb, config_env)

# 8) PPOAgent (obs_dim = flatten(window*dim))
obs, info = env.reset()
obs_dim = obs.shape[0]
agent = PPOAgent(obs_dim, ppo_cfg)

# 9) Boucle globale
while global_step < total_steps:
    # collecter un rollout
    # mettre à jour PPO
    # stocker equity pour stats

# 10) Backtester (compute_pnl_stats)

```

Les étapes 1→8 préparent le système ; 9→10 réalisent l'apprentissage.

---

## 5.6 VII.2 — Le dummy context pour déterminer la dimension

```

dummy_ctx = {
    "shitcoin": {...},
    "btc": {...},
    "sentiment": {...},
    "generic": {...},
}
dummy_vec, _ = fe.compute_features(dummy_ctx)
dim = dummy_vec.shape[0]

```

Ce “dummy\_ctx” n’a qu’un but :

- savoir combien de features totales ton FeatureEngine produit
- pour pouvoir dimensionner le StateBuilder et l’observation RL.

Ensuite, le FeatureEngine calcule la vraie dimension dynamiquement.

---

## 5.7 VII.3 — L’environnement, le RL et les rollouts

La boucle RL :

1. obs = env.reset()

2. Pour `rollout_len` étapes :
    - on stocke l'observation
    - on obtient `v_t` via `v_net(obs)`
    - on choisit `action = agent.act(obs)`
    - on appelle `env.step(action)`
    - on stocke `reward, logp, done, new_obs`
    - on met à jour `equity_history` pour le backtester
    - on gère le `reset()` en cas de `done`
  3. On a alors des buffers :
    - `obs_buf`
    - `act_buf`
    - `logp_buf`
    - `rew_buf`
    - `val_buf`
    - `done_buf`
  4. On calcule `adv` et `ret` par `compute_gae`.
  5. On appelle `agent.update(buf)` pour mettre à jour PPO.
  6. On boucle jusqu'à avoir fait `total_steps`.
- 

## 5.8 VII.4 — Le backtest final

À la fin :

```
equity_curve = np.array(equity_history)
stats = compute_pnl_stats(equity_curve)
print("Backtest stats:", stats)
```

Le backtester renvoie :

- `pnl`
- `sharpe`
- `max_drawdown`
- `avg_ret`
- `vol_ret`

C'est une première mesure : **ton agent crée-t-il réellement de la valeur sur ce simulateur ?**

Sur le simulateur, ce n'est pas intéressant. Mais sur des vraies données, c'est ton premier critère.

---



## 5.9 VII.5 — Ce que tu dois modifier quand tu passes en réel

Dans `train_ppo.py`, à terme tu dois remplacer :

1. `simulate_market(...)`  
→ `load_real_market_data(...)`  
ou un loader custom sur ton historique.
2. `load_heston_inverse_model(..., ckpt_path=None)` (dummy)  
→ `load_heston_inverse_model(..., ckpt_path="models/heston_inverse_real.pt"`  
ou `"models/heston_inverse_mixed.pt"`.
3. Reward simple  
→ `RewardEngine` financier (PnL, vol, drawdown, turn-over).
4. Sentiment dummy (0.0)  
→ vrai `SentimentProvider`.

## 6 VIII — DONNÉES RÉELLES : BINANCE + DERIBIT + SENTIMENT + REWARD PRO

C'est ici qu'on quitte la "maquette" pour construire un système utilisable pour de la vraie recherche.

Tu avais au départ :

- `simulate_market()` → données totalement artificielles
- `DummyHestonInverse` → inverseur Heston vide
- pas de sentiment réel
- reward simpliste

Je t'ai donné ensuite **toute la chaîne pour passer au réel** :

1. Loader Binance (spot, futures, funding)
2. Loader Deribit (surface IV)
3. Stockage des surfaces IV en `.npz`
4. Alignement temporel Binance/Deribit
5. Loader structuré `RealMarketData`
6. Module de sentiment prêt à brancher
7. `RewardEngine` financier

On va passer en revue tout ça.

---

### 6.1 VIII.1 — Loader Binance (spot + futures + funding)

Fichier : `data/binance_loader.py`

## Objectif

- Récupérer sur Binance :
  - Spot OHLCV (BTC/USDT, shitcoins/USDT)
  - Futures Perp OHLCV
  - Funding rates
  - Alignés dans un unique DataFrame

## Fonctions clés

```
def fetch_binance_ohlc(symbol, timeframe, limit, futures=False)

    — Utilise ccxt.binance() ou ccxt.binanceusdm()
    — Renvoie un DataFrame indexé par timestamp, avec :
        — open, high, low, close, volume

def fetch_binance_funding(symbol, limit)

    — Récupère l'historique de funding rate futures
    — Retourne un DataFrame avec funding_rate indexé par timestamp

def align_series(*dfs, how="inner")

    — Joint plusieurs DataFrames sur leur index (datetime)
    — Résultat : un seul DataFrame aligné.

def load_binance_data(spot_symbol, futures_symbol, timeframe, limit)

    — Spot OHLCV
    — Futures OHLCV (juste close_fut)
    — Funding rate
    — Renvoie un DataFrame avec colonnes :
        — close_spot, volume_spot, close_fut, funding_rate
    Ce loader est ton pipe standard pour récupérer : BTC spot, BTC perp,
    DOGE spot, DOGE perp, etc.
```

---

## 6.2 VIII.2 — Loader Deribit : surfaces d'IV

Fichier : `data/deribit_loader.py` + `data/deribit_scraper.py`

### But

- Récupérer les instruments d'options BTC chez Deribit
- Construire une surface IV réduite (quelques strikes, quelques maturités)
- La stocker comme matrice  $[NK, NT]$ , avec `k_grid`, `t_grid`, `spot`

`snapshot_iv_surface(...)`

1. Récupère la liste des options non expirées (BTC)
2. Filtre sur les calls
3. Convertit les times to maturity en années (T)
4. Choisit quelques maturités (ex : 4)
5. Pour chaque maturité, définit une grille de  $k = \ln(K/ATM)$
6. Pour chaque (k,T), trouve l'instrument le plus proche
7. Récupère son `mark_iv` via `get_order_book`
8. Remplit `iv_surface[i,j]` en fraction (`iv%` / 100)

Il renvoie :

```
{
  "iv_surface": iv_surface, # [NK,NT]
  "k_grid": k_grid,        # [NK]
  "t_grid": t_grid,        # [NT]
  "spot": spot,
  "timestamp": ts_unix,
}
```

`save_snapshot_npz(...)`

- Appelle `snapshot_iv_surface`
  - Sauvegarde dans `data/deribit_surfaces/<currency>_iv_<timestamp>.npz`
- Tu peux le lancer "toutes les X minutes" pour accumuler des surfaces.

---

### 6.3 VIII.3 — RealMarketData : remplacer `simulate__market`

Fichier : `data/real_market_loader.py`

#### Objectif

Produire une structure analogue à `SimMarketData`, mais avec **données réelles**.

```
@dataclass
class RealMarketData:
    btc_prices: np.ndarray
    btc_fut_prices: np.ndarray
    btc_funding: np.ndarray
    btc_oi: np.ndarray
    btc_iv_surface: np.ndarray # [n_steps, NK, NT]
```

```

k_grid: np.ndarray
t_grid: np.ndarray

shit_prices: np.ndarray
shit_volumes: np.ndarray
shit_funding: np.ndarray

load_real_market_data(...)
    — Utilise load_binance_data pour :
      — BTC (spot & futures & funding)
      — shitcoin (spot & volume & funding)
    — Utilise build_iv_surface ou le scraper .npz pour une surface IV
      BTC
    — Copie/collage de la même surface IV sur toute la série (prototype)
      → à toi d'implémenter un glissement temporel plus fin.
    Ensuite tu remplaces dans train_ppo.py :

market = simulate_market(...)

par :

from data.real_market_loader import load_real_market_data
market = load_real_market_data(...)

```

---

## 6.4 VIII.4 — Alignement Binance/Deribit

Fichier : data/alignment.py

### Problème

- Binance fournit OHLCV à une certaine fréquence (1m, 5m...).
- Deribit fournit des snapshots d'IV à d'autres timestamps.
- Ton RL/FeatureEngine attend des **données alignées** :  
prix BTC et surfaces IV “dans la même timeline”.

### Solution

1. load\_deribit\_npz\_as\_df(directory)
  - liste tous les .npz de surface
  - construit un DF avec index datetime, colonne iv\_file (chemin du fichier)
2. align\_binance\_deribit(binance\_df, deribit\_index\_df, tolerance)

- `merge_asof` sur index datetime
- trouve pour chaque timestamp Binance le fichier IV le plus proche (sous `tolerance`)

Ensuite, dans ton environnement, tu peux pour chaque `t` :

- prendre le chemin `iv_file`
- charger `iv_surface`, `k_grid`, `t_grid` à la volée.

## 6.5 VIII.5 — Sentiment réel (squelette)

Fichier : `sentiment/sentiment_module.py`

Tu as :

```
@dataclass
class SentimentSnapshot:
    sentiment_score: float
    msg_rate: float
    fear_greed: float

class SentimentProvider:
    def fetch_snapshot(self) -> SentimentSnapshot:
        # TODO: brancher Telegram/Twitter/Discord
        return SentimentSnapshot(...)

    def to_feature_dict(self, snap: SentimentSnapshot) -> Dict[str, float]:
        return {...}
```

Tu peux :

- connecter Twitter via API → classifier les tweets (BERT/vader)
- connecter Telegram via le nombre de messages + analyse de texte
- connecter un Crypto Fear & Greed Index

Dans `TradingEnv._build_context`, tu remplaces :

```
sentiment_score = 0.0
msg_rate = 0.0
fear_greed = 0.0
```

par :

```
snap = self.sentiment_provider.fetch_snapshot()
sent_ctx = self.sentiment_provider.to_feature_dict(snap)
"sentiment": sent_ctx,
```

## 6.6 VIII.6 — RewardEngine : reward financier sérieux

Fichier : rl/reward.py

### Problème du reward initial

$\text{Reward} = \text{PnL} / \text{capital\_initial}$

→ incite à prendre des risques violents, sans pénaliser :

- drawdown
- leverage excessif
- turnover
- volatilité du portefeuille

### RewardEngine

@dataclass

class RewardConfig:

```
pnl_scale: float = 1.0
turnover_penalty: float = 0.001
drawdown_penalty: float = 0.1
leverage_penalty: float = 0.05
target_vol: float = 0.02
vol_penalty: float = 0.05
```

class RewardEngine:

```
def compute_reward(
    self,
    equity_prev,
    equity_now,
    position_prev,
    position_now,
    price,
) -> float:
    # rel_pnl = (equity_now - equity_prev)/equity_prev
    # turnover = |position| * price
    # inst_dd = drawdown instantané
    # lev = |position_now * price / equity_now|
    reward = pnl_scale*rel_pnl
            - turnover_penalty*turnover
            - drawdown_penalty*max(0,-inst_dd)
            - leverage_penalty*lev
```

### Intégration dans TradingEnv

Dans `__init__` :

```
self.reward_engine = RewardEngine(RewardConfig(...))
```

Dans `reset()` :

```
self.reward_engine.reset()
```

Dans `step()` :

```
reward = self.reward_engine.compute_reward(
    equity_prev=equity,
    equity_now=new_equity,
    position_prev=self.position,
    position_now=target_position,
    price=price_t,
)
```

Tu obtiens alors un reward beaucoup plus proche de ce qu'un risk manager accepterait : **PnL ajusté du risque**.

## 7 IX — INVERSEUR HESTON (SYNTHÉTIQUE, RÉEL, MIXTE) : LE CŒUR MATHÉMATIQUE DU FRAMEWORK

On arrive à la brique qui fait que ce projet n'est pas juste un énième RL Crypto bidon : **l'inverseur Heston**.

C'est lui qui transforme une surface IV (ou pseudo-surface) en **paramètres de volatilité stochastique** :

$$(\kappa, \theta, \sigma, \rho, v_0)$$

Ces paramètres deviennent des **features puissantes** pour ton RL (et plus largement pour tout modèle de trading / risque).

Tu as trois étages :

1. Entraînement sur données synthétiques (`train_inverse_heston.py`)
2. Fine-tuning sur données Deribit réelles (`train_heston_real.py` + `RealIvSurfaceDataset`)
3. Entraînement mixte (synthétique + réel) (`mixed_heston_trainer.py`)

Cette partie est **fondamentale** : si tu la comprends, tu comprends toute la logique du framework.

---

## 7.1 IX.1 — Le modèle Inverse Heston (models/heston\_inverse\_model.py)

### Structure générale

```
class InverseHestonModel(nn.Module):
    def __init__(self, nk, nt, hidden_dim=128):
        self.encoder = EncoderCNN(nk, nt, hidden_dim)
        self.head_kappa = ParamHead(hidden_dim, 1)
        self.head_theta = ParamHead(hidden_dim, 1)
        self.head_sigma = ParamHead(hidden_dim, 1)
        self.head_rho = ParamHead(hidden_dim, 1)
        self.head_v0 = ParamHead(hidden_dim, 1)
        self.surface_head = SurfaceHead(hidden_dim, nk, nt)
```

Entrée :

— **x** : tensor [B, 1, NK, NT] = surface normalisée (variance totale ou IV)

Sortie :

— **params** : [B,5] = paramètres scalés (z-score)

— **surf\_recon** : [B,1,NK,NT] = reconstruction de la surface

### EncoderCNN

— CNN 2D : 2 blocs de conv+ReLU+pool

— flatten → MLP → embedding latent **z** (dimension **hidden\_dim**)

### ParamHead

— petit MLP qui prend **z** et produit un scalaire (paramètre Heston).

### SurfaceHead

— MLP qui prend **z**

— produit NK\*NT valeurs

— reshape en [1,NK,NT]

— reconstruction de la surface.

**Idée clé :**

Le modèle est un **autoencodeur avec tête de paramètres** :

$$\text{Surface} \rightarrow z \rightarrow \begin{cases} \text{Paramètres} \\ \text{Surface reconstruite} \end{cases}$$

La reconstruction de surface agit comme un **régularisateur** très fort.

---



## 7.2 IX.2 — Entraînement synthétique (train\_inverse\_heston.py)

Ce script :

- génère des paramètres Heston synthétiques
- génère des surfaces Heston-like synthétiques
- entraîne l'inverseur à :
  - reconstruire les paramètres
  - reconstruire la surface
- sauvegarde le modèle

### IX.2.1 — Grille (k,T)

```
K_POINTS = np.array([-0.4, -0.2, 0.0, 0.2, 0.4])
T_POINTS = np.array([0.05, 0.1, 0.25, 0.5, 1.0, 2.0])
NK, NT = 5, 6
```

→ tu définis une grille **fixe** pour l'entraînement.

### IX.2.2 — Sampling des paramètres Heston

```
kappa ~ LogUniform[0.5, 8]
theta ~ LogUniform[0.01, 0.08]
sigma ~ LogUniform[0.1, 1.0]
rho ~ [-0.95, 0.2] via Beta(2,2)
v0 ~ LogUniform[0.01, 0.08]
```

→ couverture de nombreux régimes volatils.

### IX.2.3 — Générateur de surfaces Heston-like

Sketche :

```
KK, TT = np.meshgrid(K_POINTS, T_POINTS)
var_t = theta + (v0 - theta) * exp(-kappa * T)
iv_level = sqrt(var_t)

skew = rho * K / sqrt(1 + T)
curvature = sigma * K^2

iv = max(iv_level + skew + curvature, 1e-4)
w = iv^2 * T # variance totale
```

Ce n'est pas un vrai Heston semi-analytique, mais un **proxy géométrique** suffisant pour :

- générer des surfaces cohérentes
- apprendre le mapping inverse.

### IX.2.4 — Dataset synthétique (HestonInverseDataset)

Pour chaque sample  $i$  :

- `params[i] : [5]`
- `surfaces[i] : [NK,NT] (w)`
- `surfaces_norm : (w - mean_w) / std_w`
- `params_scaled : (params - mean) / std`

Le `__getitem__` retourne :

```
x = w_norm[None, :, :] # [1,NK,NT]
y = params_scaled       # [5]
w_true_norm = w_norm    # [NK,NT]
```

### IX.2.5 — Loss et training loop

Loss totale :

$$\mathcal{L} = \mathcal{L}_{params} + \lambda \mathcal{L}_{recon}$$

où :

- `L_params = SmoothL1(params_pred, y)`
- `L_recon = MSE(surface_pred, w_true_norm)`

`w_recon = = 0.3` typiquement.

Training :

- Optimiseur : AdamW, lr=3e-4
- 30 epochs (tu peux augmenter)
- best model stocké avec :
  - `model_state_dict`
  - `param_mean, param_std`
  - `w_mean, w_std`
  - `K_POINTS, T_POINTS`

Checkpoint : `models/heston_inverse_synth.pt`.

**Après ça**, tu as un inverseur Heston cohérent, capable de :

- prendre une surface synthétique
- sortir des paramètres Heston
- reconstruire une surface approximative.

---

## 7.3 IX.3 — Fine-tuning sur données réelles Deribit (RealIvSurfaceDataset + train\_heston\_real.py)

### IX.3.1 — RealIvSurfaceDataset

Tu sauves d'abord des snapshots `.npz` comme :

```

np.savez("data/deribit_surfaces/BTC_iv_1234567890.npz",
        iv_surface=iv_surface,
        k_grid=k_grid,
        t_grid=t_grid,
        spot=spot,
        timestamp=timestamp)

```

**RealIvSurfaceDataset :**  
— liste tous les `.npz` du répertoire  
— lit `iv_surface`  
— calcule `iv_mean`, `iv_std` global  
— renvoie pour chaque sample :

```

iv_norm = (iv_surface - iv_mean) / iv_std
x = iv_norm[None, :, :] # [1, NK, NT]
y_dummy = zeros(5)

```

→ On n'a pas de labels sur les paramètres, on est en **self-supervised**.

### IX.3.2 — Fine-tuning (train\_heston\_real.py)

**Objectif :** prendre le modèle pré-entraîné sur synthétique, et l'adapter aux vraies surfaces Deribit.

Procédure :

1. Charger `InverseHestonModel`
2. Charger les poids `heston_inverse_synth.pt`
3. Désactiver éventuellement le training des têtes de paramètres :

```

if not train_param_heads:
    for name, p in model.named_parameters():
        if "head_" in name:
            p.requires_grad = False

```

4. Optimiser uniquement la **reconstruction** :

```

params_pred, surf_pred = model(xb) # surfaces réelles normalisées
loss = MSE(surf_pred.squeeze(1), xb.squeeze(1))

```

5. Plusieurs epochs sur les surfaces réelles.
6. Sauvegarde d'un nouveau checkpoint : `models/heston_inverse_real.pt`.

Cela donne un modèle où :

- le latent et le décodeur de surface sont adaptés aux surfaces IV réelles,
- les têtes de paramètres sont soit figées (pré-training synthé), soit affinées.

**Avantage :**

- ton modèle “comprend” la géométrie des surfaces Deribit
  - tout en restant structuré par l’entraînement synthétique (qui donne une structure Heston-like).
- 

## 7.4 IX.4 — Entraînement mixte (synthétique + réel) (mixed\_heston\_trainer.py)

Pour aller plus loin :

Tu veux un modèle qui :

1. garde la supervision complète sur des surfaces synthétiques (où tu connais les paramètres exacts)
2. mais apprend la géométrie des surfaces réelles en parallèle

### IX.4.1 — Setup

- `HestonInverseDataset` synthétique
- `RealIvSurfaceDataset` réel
- `DataLoader` pour les deux

### IX.4.2 — Boucle d’entraînement

Pour chaque batch synthétique :

1. On prend un batch synthétique (`xb_s`, `yb_s`, `wb_s`)
2. On prend un batch réel (`xb_r`, `_`)
3. On passe les deux dans le modèle :

```
params_s, surf_s = model(xb_s)
_, surf_r = model(xb_r)
```

4. On calcule :

$$\mathcal{L} = \mathcal{L}_{params}^{synth} + \lambda_s \mathcal{L}_{recon}^{synth} + \lambda_r \mathcal{L}_{recon}^{real}$$

Avec :

- `Lp = SmoothL1(params_s, yb_s)`
- `Lr_s = MSE(surf_s, wb_s)`
- `Lr_r = MSE(surf_r, xb_r.squeeze(1))`

5. Backward + step.

Tu obtiens un modèle :

- **Heston-consistent** (grâce au synthétique)
- **Deribit-consistent** (grâce au réel)

Checkpoint : `models/heston_inverse_mixed.pt`.

C’est probablement ta meilleure option à terme.

---

## 7.5 IX.5 — Intégration dans le FeatureEngine

Une fois ton inverseur entraîné (synth, real ou mixte), tu l'utilises dans :

- `BtcHestonFeatureModule` (pour surfaces IV réelles, `K_POINTS`, `T_POINTS` alignés)
- `ShitcoinFeatureModule` (pour pseudo-surfaces moments)

C'est transparent : tout ce qui change, c'est le `ckpt_path` lors du `load_heston_inverse_model`.

---

## 7.6 IX.6 — Pourquoi ceci est très puissant

Tu es en train de faire ce que font les desks quant sérieux :

- un modèle génératif & inverse pour la vol (Heston),
- entraîné hors-ligne sur données synthétiques,
- adapté à des surfaces réelles,
- qui produit un **embedding structurel** du marché,
- que tu utilises dans des modèles RL / ML / risk.

Pour un projet perso, c'est **massivement au-dessus** de 99% de ce qui traîne sur le web.

## 8 X — HESTON PRICER DIFFÉRENTIABLE & CALIBRATION (LE LAB DE VOLATILITÉ)

Ce chapitre couvre la partie “pricing / calibration” que je t’ai ajoutée à ton lab :

- un **pricer Heston différentiable** en PyTorch
- un **squelette de calibration** via descente de gradient

Ce n’est pas directement nécessaire pour ton RL, mais c’est crucial pour :

- tester ton inverseur Heston,
  - générer des surfaces synthétiques plus réalistes,
  - faire de la recherche sur la structure de vol,
  - calibrer des paramètres sur des surfaces réelles.
- 

### 8.1 X.1 — Char-fonction Heston différentiable (`models/heston_pricer.py`)

#### 1) Rappel rapide : Heston en termes de char-fonction

Le modèle Heston donne une formule semi-analytique pour le prix d’option via la **char-fonction** du log-prix sous la mesure risque-neutre :

$$\phi(u; T) = \mathbb{E}[e^{iu \log S_T}]$$

Dans Heston, la char-fonction s’écrit avec :

- $\kappa$  (mean-reversion de la variance)
- $\theta$  (level de variance de long terme)
- $\sigma$  (vol-of-vol)
- $\rho$  (corrélation)
- $v_0$  (variance initiale)

Et tu as des formules du type :

$$d = \sqrt{(\kappa - \rho\sigma iu)^2 + \sigma^2(iu + u^2)}$$

etc.

Le fichier `heston_pricer.py` code tout ça en `torch.complex128` pour être différentiable.

## 2) Fonction `heston_charfunc_torch`

Signature :

```
heston_charfunc_torch(
    u, T, kappa, theta, sigma, rho, v0, r, q, logS0
) -> (u)
```

- `u` : tenseur de fréquences
- `T` : maturité
- `r` : taux sans risque
- `q` : dividende / taux de repo / convenience yield
- `logS0` : log spot

Retourne :

- $\phi(u)$  complexe, torch, compatible autograd.

## 3) Calcul technique

Dans le code, tu as :

- `alpha`, `beta`, `gamma`
- $d = \sqrt{\beta^2 - 4\alpha\gamma}$
- $g = (\beta - d)/(\beta + d)$
- $C(T, u)$  et  $D(T, u)$

Puis :

$$\phi(u) = \exp \left( C + Dv_0 + iu(\log S_0 + (r - q)T) \right)$$

Cette fonction est 100% différentiable par rapport à tous ses inputs.

---

## 8.2 X.2 — Prix du call Heston (heston\_call\_price\_torch)

Pour le call européen, on utilise la formule classique (type Heston) :

$$C = S_0 e^{-qT} P_1 - K e^{-rT} P_2$$

où :

$$P_j = \frac{1}{2} + \frac{1}{\pi} \int_0^\infty \Re \left( e^{-iu \ln K} \frac{f_j(u)}{iu} \right) du$$

Avec :

$$f_1(u) = \frac{\phi(u-i)}{S_0 e^{(r-q)T}}, \quad f_2(u) = \frac{\phi(u)}{S_0 e^{(r-q)T}}$$

La fonction `heston_call_price_torch` :

- construit une grille `u` entre 0 et `u_max`,
- calcule `phi1` et `phi2` pour cette grille,
- calcule les intégrales par la méthode des trapèzes (`torch.trapz`),
- renvoie une matrice de prix `[NK,NT]` pour un vecteur de strikes `K` et `T`.

### Signature :

```
heston_call_price_torch(  
    S0: float,  
    K: torch.Tensor, # [NK]  
    T: torch.Tensor, # [NT]  
    r: float,  
    q: float,  
    params: (kappa, theta, sigma, rho, v0),  
    n_integration: int = 256,  
    u_max: float = 100.0,  
) -> torch.Tensor # [NK,NT]
```

Ce pricer :

- est entièrement en torch
- supporte autograd
- est suffisant pour faire des calibrations locales ou de la recherche.

---

## 8.3 X.3 — Calibration Heston via gradient descent (calibration/heston\_calibration)

Le but : pour une surface de prix “market” donnée, trouver des paramètres Heston  $(\kappa, \theta, \sigma, \rho, v_0)$  qui minimisent :

$$\mathcal{L} = \frac{1}{N} \sum_{i,j} (C^{model}(K_i, T_j) - C^{market}(K_i, T_j))^2$$

## Fonctions principales

`calibrate_heston_torch` :

- convertit les données de marché en tenseurs torch
- initialise des paramètres Heston
- boucle gradient descent/Adam
- projette les paramètres dans des bornes raisonnables
- affiche la trajectoire de la loss et des paramètres

Pendant chaque itération :

1. `model_price = heston_call_price_torch(...)`
2. `loss = MSE(model_price, market_price)`
3. `loss.backward()`
4. `opt.step()`
5. clamp des paramètres (ex :  $\theta > 0$ ,  $|\rho| < 1$ , etc.)

## Attention volontaire

Je t'ai mis un placeholder pour `market_price` à partir des IV :

```
market_price = market_iv * 0.0 # à remplacer
```

Dans une calibration réelle :

- soit tu calcules le prix market via **Black-Scholes** à partir de l'IV,
- soit tu calibres directement sur IV (plus complexe, car il faut faire l'inversion price→IV pour Heston).

Ce code te donne la **structure** de la calibration différentiable, mais tu dois :

- ajouter un BS propre (si tu veux calibrer sur prix),
- ou modifier la loss pour travailler sur IV (après inversion Heston).

## 9 XI — COMMENT TOUT S'ASSEMBLE (FLUX COMPLETS)

Maintenant tu as vu chaque brique. On va les remonter ensemble en 3 flux :

1. Flux RL complet
2. Flux training inverseur complet
3. Flux pricing/calibration complet

Tu dois visualiser ça comme trois “pipelines parallèles” qui interagissent.



## 9.1 XI.1 — Flux RL complet (en réel)

Objectif : agent RL qui apprend à trader BTC/shitcoin en utilisant les signaux Heston, sentiment, etc.

### Étapes :

#### 1. Données marché

- Charger `RealMarketData` via `load_real_market_data` (Binance + Deribit alignés)

#### 2. Initialisation `FeatureEngine`

- Charger inverseur Heston pré-entraîné `heston_inverse_mixed.pt`
- Construire `FeatureEngine` avec :
  - `ShitcoinFeatureModule` (pseudo-surface → inverseur Heston)
  - `BtcHestonFeatureModule` (IV Deribit → inverseur Heston)
  - `SentimentFeatureModule` (via `SentimentProvider`)
  - `GenericMarketFeatureModule` (OHLCV, etc.)

#### 3. `StateBuilder`

- `StateBuilder(dim=D, window=16)`

#### 4. `TradingEnv`

- `TradingEnv(market=RealMarketData, feature_engine=fe, state_builder=sb, config=...)`
- reward via `RewardEngine` (PnL + drawdown + leverage + turnover)

#### 5. `PPO Agent`

- `PPOAgent(obs_dim=16*D)`

#### 6. Boucle d'apprentissage

- For `t = 0..T` :
  - `obs → agent.act(obs) → action`
  - `env.step(action)`
  - stocker trajectoire
- Après `rollout_len` steps, `agent.update(buf)`

#### 7. `Backtest`

- `equity_history`
- `compute_pnl_stats(equity_curve)`

Résultat :

Tu as un agent RL qui apprend des politiques conditionnées sur :

- régimes de vol (via inverseur Heston)
- structure IV BTC
- dynamiques shitcoin en “style Heston”
- sentiment
- vol réalisée

## 9.2 XI.2 — Flux training inverseur complet

Objectif : obtenir un inverseur Heston robuste.

### Étape 1 : Synthétique uniquement (pré-training)

- `train_inverse_heston.py`
- Dataset `HestonInverseDataset`
- Loss = param + recon
- Sortie : `models/heston_inverse_synth.pt`

### Étape 2 : Réel seul (fine-tuning reconstruction)

- Récupérer surfaces Deribit via `deribit_scraper.py`
- Stocker dans `data/deribit_surfaces/*.npz`
- `RealIvSurfaceDataset`
- `train_heston_real.py`
- Loss = recon only
- Sortie : `models/heston_inverse_real.pt`

### Étape 3 : Mix synthétique + réel

- `mixed_heston_trainer.py`
  - Mélange synth/real dans la même training loop
  - Loss =  $L_{\text{param}}(\text{synth}) + L_{\text{recon}}(\text{synth}) + L_{\text{recon}}(\text{real})$
  - Sortie : `models/heston_inverse_mixed.pt`
- C'est ce modèle mixte qui est en général **le plus robuste**.
- 

## 9.3 XI.3 — Flux pricing/calibration

Objectif : calibrer/étudier Heston pour des surfaces particulières.

### 1. À partir d'une surface IV de marché

- `market_iv[K,T]`
- `K_grid, T_grid, S0, r, q`

## 2. Calibration :

- Convertir IV  $\rightarrow$  prix (via BS ou approximation)
- Appeler `calibrate_heston_torch`
- Obtenir :

$$(\hat{\kappa}, \hat{\theta}, \hat{\sigma}, \hat{\rho}, \hat{v}_0)$$

## 3. Étude :

- Forward pricing via `heston_call_price_torch`
- Comparer surfaces calibrées vs marché
- Tester la stabilité de l'inverseur Heston NN vs calibration numérique torch

# 10 XII — CONCLUSION : CE QUE TU AS, CE QUE TU DOIS FAIRE

## 10.1 XII.1 — Ce que tu possèdes maintenant

Tu as, dans ce projet :

1. **Un framework RL complet**
  - Environnement de trading
  - PPO agent
  - Backtester
  - FeatureEngine modulaire
  - StateBuilder temporel
2. **Un lab de volatilité Heston complet**
  - Génération de surfaces synthétiques
  - Inverseur Heston CNN + reconstruction
  - Training synthétique
  - Fine-tuning réel (Deribit)
  - Training mixte
  - Pricer Heston différentiable
  - Calibration torch (squelette)
3. **Un début de pipeline data réel**
  - Binance (spot, futures, funding)
  - Deribit (IV surface snapshots)
  - Alignement temporel
  - Construction de `RealMarketData`
4. **Des modules avancés**
  - RewardEngine risk-adjusted
  - SentimentProvider (squelette)

— LiveTrading skeleton (à connecter à binance/deribit API)

C'est un **véritable framework de recherche quant sur vol & RL**, pas un jouet.

---

## 10.2 XII.2 — Ce que tu dois faire concrètement (roadmap)

Si tu veux le rendre utile, ton plan logique :

### Étape 1 : Inverseur Heston

1. Lancer `train_inverse_heston.py` (synth)
2. Scraper quelques centaines de surfaces Deribit
3. Lancer `train_heston_real.py` ou `mixed_heston_trainer.py`
4. Inspecter la qualité de reconstructions

Tant que ça n'est pas solide, **n'avance pas**.

### Étape 2 : Données réelles

1. Charger des séries Binance (BTC + 1-2 shitcoins)
2. Alignement temporel propre
3. Construire un `RealMarketData`
4. Adapter `TradingEnv` pour consommer `RealMarketData`

### Étape 3 : RL sur réelles

1. Brancher `RealMarketData` dans `train_ppo.py`
2. Remplacer reward par `RewardEngine`
3. Lancer un training RL
4. Analyser la distribution des actions, equity, Sharpe, DD

### Étape 4 : Améliorations

- Ajouter un vrai modèle de sentiment (Twitter / TG)
  - Ajouter un vrai pricer BS pour la calibration torch
  - Ajouter un logging sérieux (TensorBoard / wandb)
  - Séparer train/test dans le temps (éviter overfitting sur une période)
  - Tester d'autres algos RL (SAC, TD3, PPO recurrent, etc.)
-

### 10.3 XII.3 — Comment l'utiliser comme "montrer que je suis sérieux" (cv / master / thèse)

Ce framework est idéal pour :

- montrer que tu comprends :
  - Heston
  - surfaces IV
  - calibration
  - RL appliqué au trading
  - ingestion de données marché
  - engineering "propre" de features
- produire :
  - un rapport technique
  - un article de recherche perso
  - un dépôt GitHub structuré

Tu peux :

- présenter l'architecture globale,
- montrer tes résultats sur un backtest Deribit/Binance,
- montrer des reconstructions de surfaces par l'inverseur,
- montrer comment les régimes Heston influencent la politique RL.

C'est exactement le type de projet qui fait la différence quand tu parles à un desk vol ou à une école type M2 203 / X-HEC.