Apprentissage en ligne pour l'exploitation d'une machine sous information partielle

1 Modélisation

Présentation du modèle du POMDP

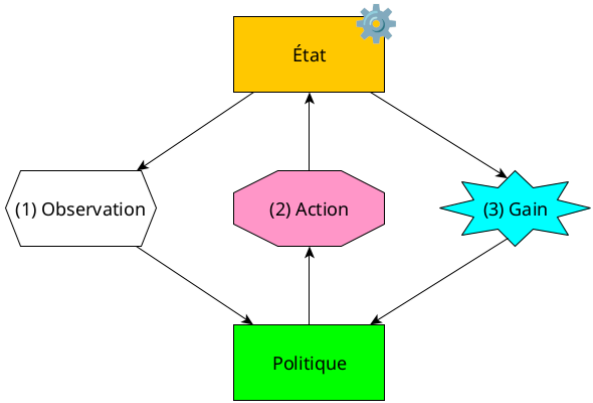

Figure 1: Schéma du processus vu par l'agent. L'état est caché.

Regret

L'écart **relatif** entre l'espérance de gain d'une politique et celle d'une politique optimale connaissant l'état caché (moyenne sur les états initiaux).

Présentation du modèle d'une machine particulière

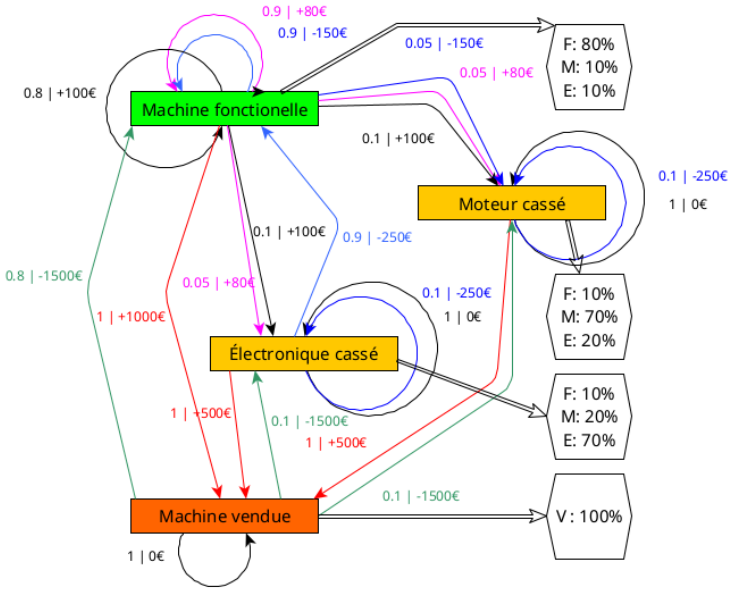

Figure 2: POMDP de la machine à laver.
Rose : Entretien; Bleu : Réparation; Vert: Achat; Rouge: Vente; Noir: Autres dont inaction.

Mon analyse et mon approche

1.c.a Difficultés spécifiques identifiées

- Apprentissage en ligne → pas une résolution classique de POMDP
- Information partielle quant à l'état → contrairement à la politique optimale
- Conséquences retardées → contrairement à un bandit classique

1.c.b Objectif

Concevoir une politique minimisant le regret, apprenant uniquement à partir des observations (pas de croyance sur l'état caché, la structure étant inconnue):

$$\text{observations, actions et gains passés, observation} \mapsto \text{action}$$

1.c.c Approche

1. Travail sur un modèle particulier de machine
2. Hypothèses quant aux améliorations possibles
3. Vérification de l'amélioration par simulation
4. Généralisation sur des modèles aléatoires (pas nécessairement réalistes pour une machine)

2 Adaptation des politiques classiques issues du bandit

Epsilon-Greedy

2.a.a Test 1

Action aléatoire sur une proportion ε des choix. Autrement, action maximisant la moyenne historique du prochain gain après l'observation reçue.

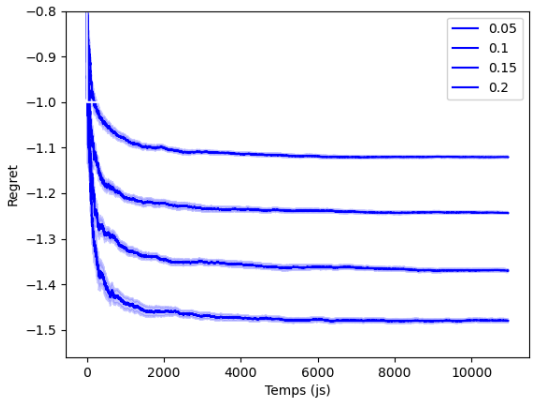

Figure 3: **Test 1** : évolution du regret relatif et des incertitudes des 4 meilleures politiques testées (meilleures en moyenne sur les 10 dernières années).

Impact d'action différé

Au lieu de choisir l'action qui maximise le gain immédiat, on choisit celle qui maximise **la somme des n gains suivants la décision** : $g_t + g_{t+1} + g_{t+2} + ... + g_{t+n-1}$

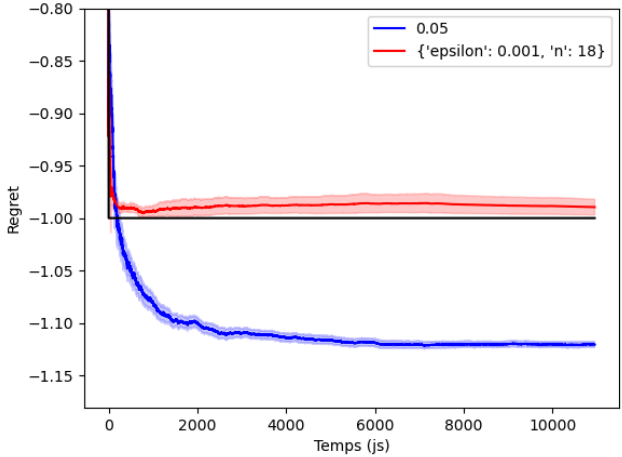

Figure 4: **Test 2** : évolution du regret relatif et de l'incertitude de la meilleure politique testée (en rouge). En bleu celle du Test 1. En noir la barre de rentabilité.

Impact d'action différé (ii)

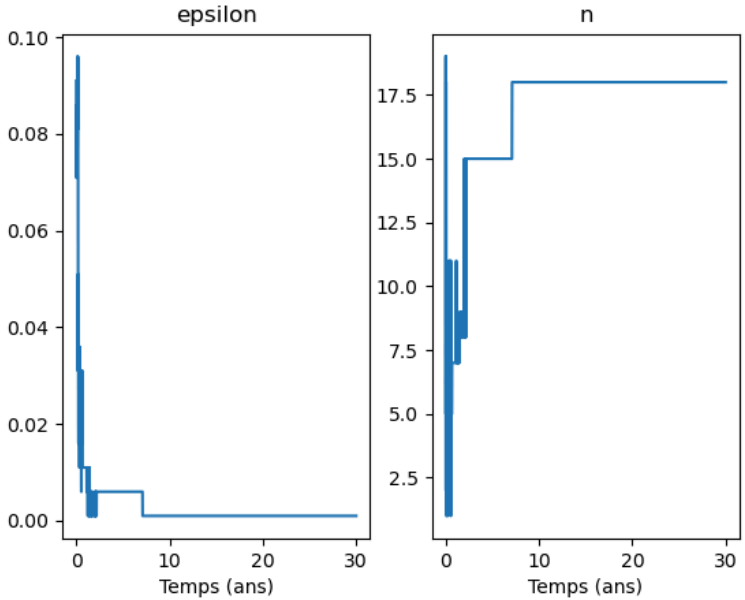

Figure 5: **Test 2** : évolution des maxima

Pondération de la mémoire

On affecte « la somme des n gains suivants le choix », d'une fonction γ:

$$\gamma^n(g_t) + \gamma^{n-1}(g_{t+1}) + ... + \gamma(g_{t+n-2}) + g_{t+n-1}$$

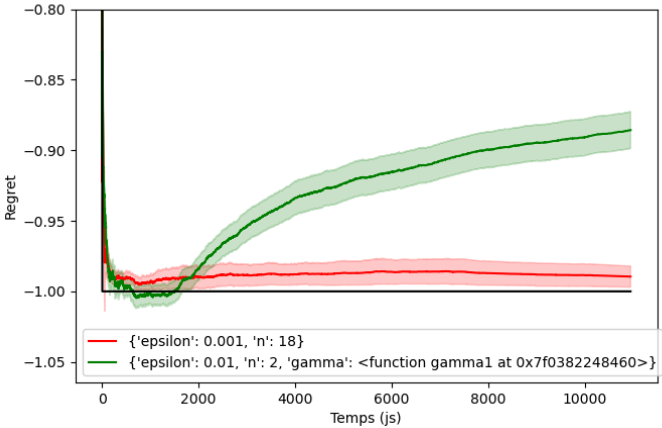

Figure 6: **Test 3** : évolution du regret relatif et de l'incertitude de la meilleure politique testée (en vert). En rouge celle du Test 2. En noir la barre de rentabilité.

Pondération de la mémoire (ii)

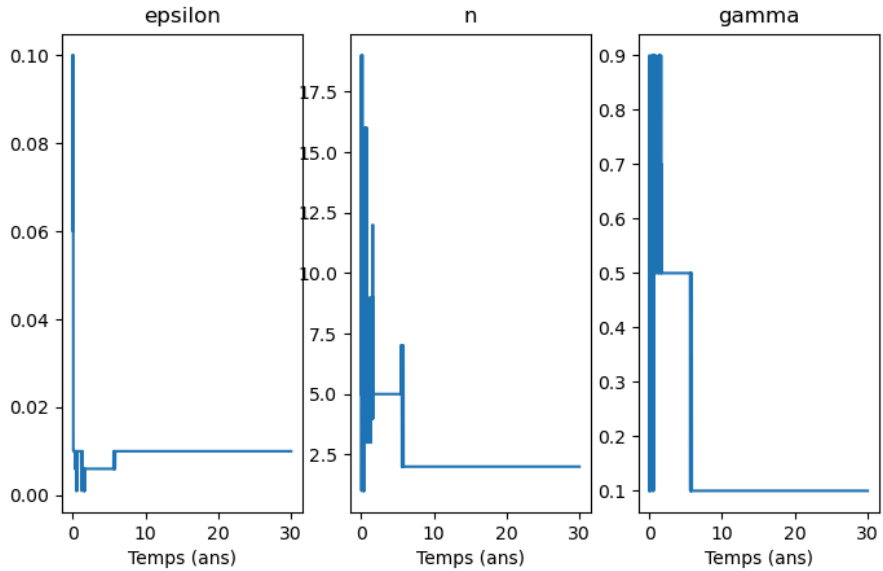

Figure 7: **Test 3** : évolution des maxima
Pour γ, a tel que $\gamma(x) = ax$.

Prise de confiance progressive

À chaque tour, on diminue ε en lui appliquant une fonction η.

Raffinement possible : On ne diminue qu'après un choix aléatoire.

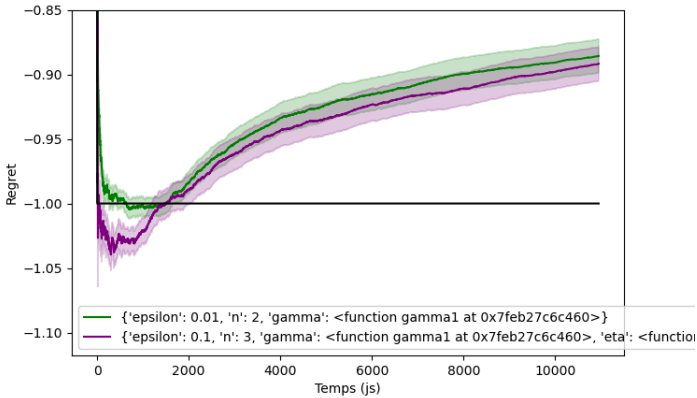

Figure 8: **Test 4** : évolution du regret relatif et de l'incertitude de la meilleure politique testée (en violet). Ici, $\eta(t, \varepsilon) = \varepsilon \frac{1+(t-1)^{\frac{1}{4}}}{1+t^{\frac{1}{4}}}$ sans raffinement,.
En vert celle du Test 3. En noir la barre de rentabilité.

Prise de confiance progressive (ii)

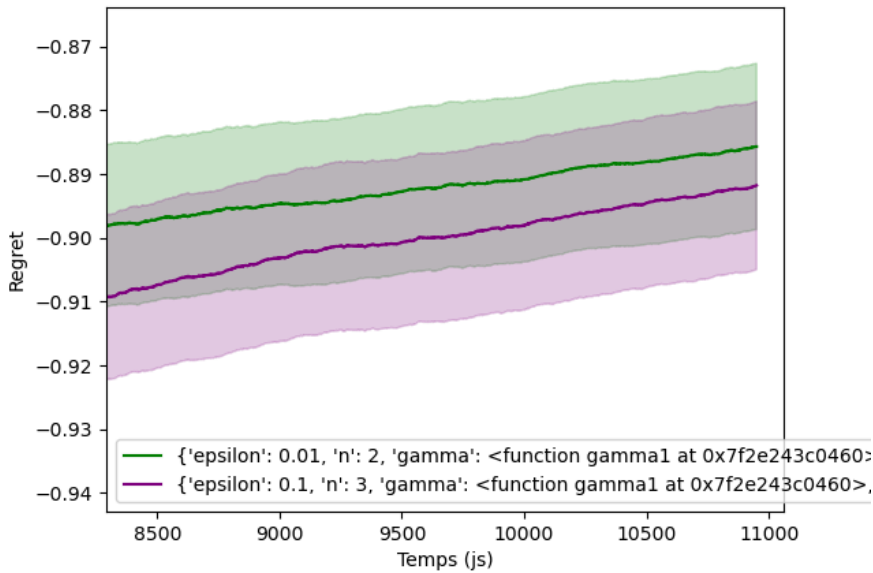

Figure 9: **Test 4** : Figure précédente, tronquée.

Biais optimiste

On change la manière d'analyser la moyenne des résultats passés : $m + f(m, n, t)$

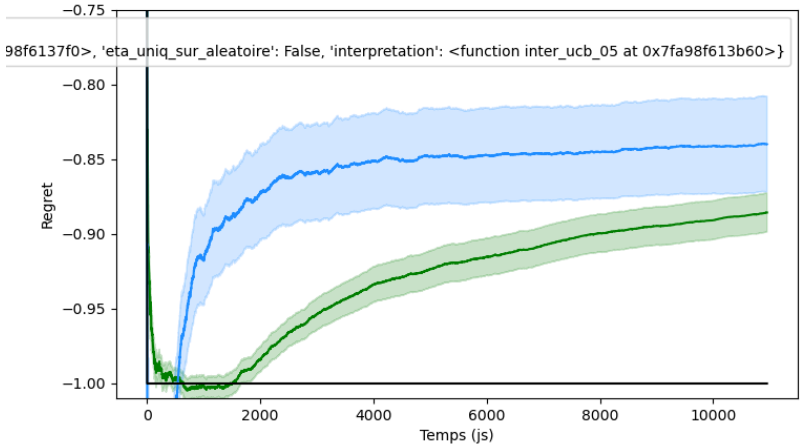

Figure 10: **Test 5** : évolution du regret relatif et de l'incertitude de la meilleure politique testée (en bleu brique). Ici, UCB(0.5), $\eta(t, \varepsilon) = \varepsilon \frac{1+(t-1)^3}{1+t^3}$ sans raffinement, $\gamma(x) = 0.1x$. En vert celle du Test 3. En noir la barre de rentabilité.

Biais optimiste (ii)

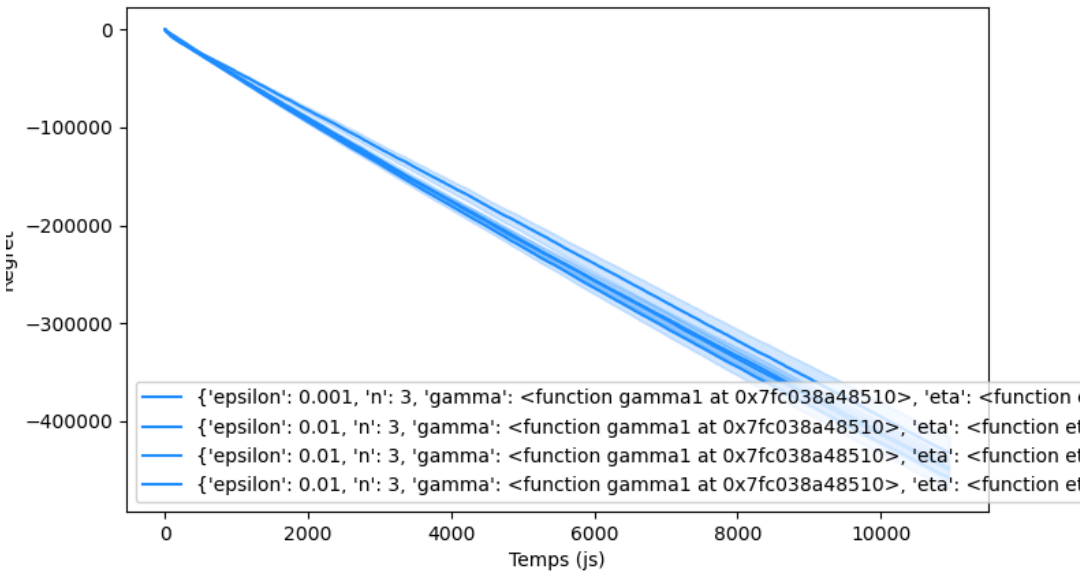

Figure 11: **Test 5** : évolution du regret non-relatif des politiques testées.

3 Évaluation sur des modèles aléatoires

Évaluation sur des modèles aléatoires

epsilon	COUNT de eps	*g_name*	COUNTA de g	Facteur	Surrep.
0,001	66	g01	3	epsilon	212,80%
0,01	39	g05	15	n	175,20%
0,05	69	g07	41	g_name	305,60%
0,1	76	g1	191	eta	148,80%
Total général	**250**	**Total général**	**250**	i_name	176,40%

n		*eta*		*i_name*	
3	137	0,9	57	i0	63
5	66	0,99	119	i1	57
10	47	0,999	74	i3	130
Total général	**250**	**Total général**	**250**	**Total général**	**250**

Figure 12: **Test 6** : population par paramètre sur les 25 meilleures politiques de chacun des 10 modèles.

En vert : facteur moyen de surreprésentation de la modalité dominante d'un paramètre parmi les 25 meilleures politiques de chaque modèle, relativement à une répartition uniforme. Pour ε, la modalité dominante apparaît en moyenne 2,13 fois plus souvent que prévu sous l'hypothèse uniforme (53,2% contre 25%).

4 Conclusion

Objectifs atteints ?

4.a.a Rappel des objectifs

1. Modéliser une ou plusieurs machines sous la forme de POMDPs.
2. S'inspirer des algorithmes classiques des bandits à plusieurs bras pour développer des politiques d'exploitation.
3. Identifier les contraintes supplémentaires apportées par les POMDPs.
4. Adapter ces politiques à la structure et ses contraintes.

4.a.b Bilan

1. Objectifs principaux atteints
2. Des améliorations graduelles
3. Un regret qui reste linéaire, bien qu'`UCB` puisse théoriquement atteindre un regret logarithmique dans le cadre des bandits à plusieurs bras.

5 Annexes - Mathématiques & Algorithmique

Définition formelle du POMDP

Un **POMDP** est défini par le sextuple : (S, A, O, T, Z, R)

- S : ensemble des états cachés de l'univers
- A : ensemble des actions possibles
- O : ensemble des observations possibles
- $T : S \times A \times S \mapsto [0; 1]$: fonction de transition, $T(s, a, s')$ est la probabilité d'atteindre s' après avoir choisi a depuis s.
- $Z : S \times A \times O \mapsto [0; 1]$: fonction d'observation, $Z(s, a, o)$ est la probabilité d'observer o après avoir choisi a en s. Mon modèle simplifie en $Z : S \times O \mapsto [0; 1]$.
- $R(s, a)$: fonction de gain, $R(s, a)$ est le gain reçu en prenant a dans l'état s

Définition formelle du regret

5.b.a Gain

On calcule $G_T(\pi, s, \sigma, o)$ le **gain espéré** à l'horizon $T \in \mathbb{N}$ en suivant la politique π depuis $s \in S$ après avoir observé $o \in O$, avec l'historique σ comme suit:

$$G_T(\pi, s, \sigma, o) = r + \sum_{(s', o') \in S \times O} T(s, a, s') \times Z(s', o') \times G_{T-1}(\pi, s', \sigma \cup (o, a, r), o')$$

où $r = R(s, \pi(\sigma, o))$ et $a = \pi(\sigma, o)$. On pose par ailleurs $G_0 = 0$.

5.b.b Gain optimal

On définit le **gain optimal** à la profondeur $T \in \mathbb{N}$ depuis $s \in S$ comme

$$G^*_T(s) = \max_{a \in A} \left(R(s, a) + \sum_{s' \in S} T(s, a, s') \times G^*_{T-1}(s') \right)$$

et $G^*_0(s) = 0$.

Définition formelle du regret (ii)

5.b.c Regret

On définit ensuite le **regret** à la profondeur $T \in \mathbb{N}$ d'une politique π comme

$$\Delta_T(\pi) = \frac{1}{|S|} \sum_{(s,o) \in S \times O} Z(s,o)(G_T^*(s) - G_T(\pi, s, \varnothing, o))$$

Et $\Delta_T^r(\Pi)$, le **regret relatif** à la profondeur $T \in \mathbb{N}$ d'une politique Π comme suit :

$$\frac{\Delta_T(\Pi)}{\frac{1}{|S|}\sum_{s \in S} g_T(s)}$$

Limites du POMDP

- États et temps discrets
- Hypothèse de Markov

UCB

On considère

$$Q_{t(a)} + c\sqrt{\frac{\log(t)}{N_t(a)}}$$

pour évaluer la valeur d'une action, où t est l'instant actuel, $Q_t(a)$ est la moyenne des récompenses obtenues en conséquence du choix de a, c une constante et $N_t(a)$ le nombre de fois où à a été choisie à l'instant t.

6 Annexes - Code Python

Contexte

On suppose les imports adéquats effectués, ainsi que ces lignes :

```
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
import random as rd

N = 10
T = 30*365  # 30 ans en jours
```

Fichier `tools.py`

```
class Moyenne:
    """
    Moyenne sans mémoire.

    Appeler pour obtenir la moyenne, ajouter(valeur) pour ajouter une
valeur à l'ensemble.

    Initialisation, ajout d'un élément et obtention de la moyenne des
éléments ajoutés en O(1).
    """

    def __init__(self, compte=0, moyenne=0.0):
        self.compte = compte
        self.moyenne = moyenne

    def __repr__(self):
        return f"Moyenne(compte={self.compte}, moyenne={self.moyenne})"

    def __call__(self):
```

Fichier `tools.py` (ii)

```
        return self.moyenne

    def ajouter(self, nv):
        if self.compte == 0:
            self.compte = 1
            self.moyenne = nv
        else:
            self.compte += 1
            self.moyenne = (
                self.moyenne * (self.compte - 1) + nv) / self.compte


class Maillon:
    """
    Maillon pour la liste doublement chaînée qu'est la file ci-dessous.

    Initialisation et modification en O(1).
    """
```

Fichier `tools.py` (iii)

```
    def __init__(self, prec, val, suiv):
        self.prec = prec
        self.val = val
        self.suiv = suiv

    def set_suiv(self, val):
        self.suiv = val

    def set_prec(self, prec):
        self.prec = prec


class FileOuPile:
    def __init__(self, *args, mode: str = None):
        '''
        File ou pile, dépendamment du mode spécifié.
        Implémenté par une liste doublement chaînée.

        Utiliser ajouter(valeur) et prendre().
```

Fichier `tools.py` (iv)

```
        Initialisation, ajout comme retrait en O(1).
        '''
        self.lg = 0
        self.debut = Maillon(None, None, None)
        self.fin = Maillon(self.debut, None, None)
        self.debut.set_suiv(self.fin)
        self.mode = mode
        if self.mode == 'File':
            self.prendre = self.prendre_premier_entre
        elif self.mode == 'Pile':
            self.prendre = self.prendre_dernier_entre

        for el in args:
            self.ajouter(el)

    def ajouter(self, val):
        self.debut.set_suiv(
            Maillon(self.debut, val, self.debut.suiv))
```

Fichier `tools.py` (v)

```
        self.debut.suiv.suiv.set_prec(self.debut.suiv)
        self.lg += 1

    def prendre_premier_entre(self):
        if self.fin.prec == self.debut:
            return None
        val = self.fin.prec.val
        self.fin.prec.prec.set_suiv(self.fin)
        self.fin.set_prec(self.fin.prec.prec)
        self.lg -= 1
        return val

    def prendre_dernier_entre(self):
        if self.debut.suiv == self.fin:
            return None
        val = self.debut.suiv.val
        self.debut.suiv.suiv.set_prec(self.debut)
        self.debut.set_suiv(self.debut.suiv.suiv)
        self.lg -= 1
```

Fichier `tools.py` (vi)

```
        return val


def ecart_norm(a, b):
    return (a - b)/b if b != 0 else None


def save(obj, fn):
    with open(fn, 'wb') as f:
        pickle.dump(obj, f)


def load(fn):
    with open(fn, 'rb') as f:
        obj = pickle.load(f)
    return obj
```

Fichier `classes.py`

```
class Markov:
    def __init__(self, nb_state, nb_action, nb_observation,
observation_pmatrix, transition_pmatrix, reward_matrix):
        """
        Initialise un processus de Markov avec les paramètres donnés.
        """
        # Initialisation des paramètres
        self.nb_etats = nb_state
        self.nb_action = nb_action
        self.nb_observation = nb_observation

        # Initialisation des matrices
        self.observation_pmatrix = observation_pmatrix
        self.transition_pmatrix = transition_pmatrix
        self.gain_matrix = reward_matrix

    def initialiser_politique(self, policy):
        """
        Initialise la politique avec les paramètres du processus de
```

Fichier `classes.py` (ii)

```
Markov.
        """
        policy.mem_fab(self.nb_observation, self.nb_action)

    def suivre_politique(self, politique, SIM, etat_actuel=None):
        """
        Simule une politique sur le processus de Markov et retourne
l'historique des gains'.
        """
        # Initialisation de la politique
        self.initialiser_politique(politique)

        # Choix aléatoire d'un état initial
        if etat_actuel is None:
            etat_actuel = rd.randint(0, self.nb_etats - 1)

        gains = []

        for _ in range(SIM):
```

Fichier `classes.py` (iii)

```
            # Tirage d'une observation, application de la politique et
récupération de la récompense
            observation = np.random.choice(
                self.nb_observation,
p=self.observation_pmatrix[etat_actuel])
            action = politique(observation)
            gain = self.gain_matrix[etat_actuel, action]

            # Mise à jour de l'historique
            gains.append(gain)

            # Mise à jour de la mémoire
            politique.apprend(observation, action, gain)

            # Transition vers le prochain état et observation
            etat_actuel = np.random.choice(
                self.nb_etats, p=self.transition_pmatrix[etat_actuel,
action])
```

Fichier `classes.py` (iv)

```
        return gains

    def gain_optimal(self, profondeur=900):
        V = np.zeros(self.nb_etats)
        V_hist = [np.mean(V)]

        for _ in range(profondeur):
            V_nv = np.zeros(self.nb_etats)
            # Pour chaque état
            for s in range(self.nb_etats):
                a_val = np.zeros(self.nb_action)
                # et pour chaque action :
                for a in range(self.nb_action):
                    # On calcule le gain associé à cette action depuis
cet état
                    gain = self.gain_matrix[s, a]
                    # et l'espérance de gain depuis cette état, après
cette action, à une profondeur décrémentée. self.transition_pmatrix[s,
a] est une ligne telle que ligne[q] est la probabilité d'être en q après
```

Fichier `classes.py` (v)

```
a en s. multiplié par V, on obtient l'espérance de gain à profondeur
décrémentée depuis q.
                    gain_lg_terme = np.sum(self.transition_pmatrix[s, a]
* V)
                    # On enregistre la somme comme l'espérance de gain
associée à l'action a depuis s.
                    a_val[a] = gain + gain_lg_terme
                # Et donc on met à jour l'espérance de gain maximale
pour s.
                V_nv[s] = np.max(a_val)
            V = V_nv
            V_hist.append(np.mean(V))

        return V_hist


class RandomMarkov(Markov):
    def __init__(self):
        # Initialisation de paramètres
```

Fichier `classes.py` (vi)

```
        nb_etats = rd.randint(2, 5)
        nb_action = rd.randint(2, 5)
        nb_observation = rd.randint(2, 5)

        # Matrice de transition
        transition_pmatrix = np.random.rand(nb_etats, nb_action,
nb_etats)
        transition_pmatrix = transition_pmatrix / \
            transition_pmatrix.sum(axis=2, keepdims=True)

        # Matrice d'observation
        observation_pmatrix = np.random.rand(nb_etats, nb_observation)
        observation_pmatrix = observation_pmatrix / \
            observation_pmatrix.sum(axis=1, keepdims=True)

        # Matrice de récompense entre -1 et 1
        reward_matrix = np.random.rand(nb_etats, nb_action)*2 - 1

        # Appel du constructeur de la classe mère
```

Fichier `classes.py` (vii)

```
        super().__init__(
            nb_etats,
            nb_action,
            nb_observation,
            observation_pmatrix,
            transition_pmatrix,
            reward_matrix
        )


class Politique:
    def __init__(self, name, decide_func, apprend_func=None,
mem_fab_func=None):
        """
        Initialise une politique avec un nom, une fonction de politique,
une fonction de mémorisation et une fonction de création de mémoire.
        """
        self.name = name
        self.decide_func = decide_func
```

Fichier `classes.py` (viii)

```
        self.apprend_func = apprend_func
        self.mem_fab_func = mem_fab_func
        self.mem = None

    def __call__(self, observation):
        """
        Décide de la prochaine action.
        """
        return self.decide_func(observation=observation, mem=self.mem)

    def __repr__(self):
        return f"Politique(name={self.name})"

    def mem_fab(self, nb_observation, nb_action):
        """
        Crée la mémoire pour la politique si une fonction de création de
mémoire est définie.
        """
        if self.mem_fab_func is not None:
```

Fichier `classes.py` (ix)

```
            self.mem = self.mem_fab_func(nb_observation, nb_action)

    def apprend(self, observation, action, gain):
        """
        Appelle la fonction d'apprentissage pour mettre à jour
l'expérience de la politique.
        """
        if self.apprend_func is not None:
            self.apprend_func(observation=observation,
                              action=action,
                              gain=gain,
                              mem=self.mem)
```

Fichier `policies.py`

```
# Politique epsilon-greedy qui choisit une action aléatoire avec une
probabilité epsilon


class Epsilon_Mem:
    def __init__(self, epsilon, nb_observation, nb_action):

        # Initialisation du tableau des gains moyens
        self.Obs_Act_donne_gain_moy = np.array(
            [[Moyenne() for _ in range(nb_action)]
             for _ in range(nb_observation)]
        )

        # Initialisation de espilon
        self.epsilon = epsilon

        # Paramêtres du POMDP
        self.nb_action = nb_action
        self.nb_observation = nb_observation
```

Fichier `policies.py` (ii)

```
    def meilleure_action_pour_observation(self, observation):
        return np.argmax([mem() for mem in
self.Obs_Act_donne_gain_moy[observation]])

    def apprend(self, observation, action, gain):
        self.Obs_Act_donne_gain_moy[observation, action].ajouter(gain)


def epsilon_greedy_pol(epsilon=0.1):
    """
    Crée une politique epsilon-greedy, qui choisit une action aléatoire
avec une probabilité epsilon,
    ou choisit l'action avec la meilleure moyenne dans la mémoire.
    """
    def decide(observation, mem):
        """
        Fonction de politique epsilon-greedy qui choisit une action en
fonction de la mémoire.
```

Fichier `policies.py` (iii)

```
        """
        if rd.random() < mem.epsilon:
            return rd.randint(0, mem.nb_action - 1)
        return mem.meilleure_action_pour_observation(observation)

    def apprend(observation, action, gain, mem):
        """
        Fonction de mémorisation qui met à jour la mémoire avec
l'observation, l'action et la récompense.
        """
        mem.apprend(observation, action, gain)

    def mem_fab(nb_observation, nb_action):
        """
        Fonction qui crée la mémoire pour la politique epsilon-greedy.
        """
        return Epsilon_Mem(
            epsilon,
            nb_observation,
```

Fichier `policies.py` (iv)

```
            nb_action
        )

    return Politique(f"Epsilon-Greedy (epsilon={epsilon})",
                     decide_func=decide,
                     apprend_func=apprend,
                     mem_fab_func=mem_fab)

# Politique epsilon-greedy modifiée


class FileDiff:
    '''
    File différée.

    Permet de garder des au plus n éléments en mémoire.
    Si un élément supplémentaire est ajouté, renvoie le plus ancien.

    S'initialise avec sa profondeur, s'appelle avec l'objet à ajouter.
```

Fichier `policies.py` (v)

```
    Initialisation comme appel en O(1).
    '''

    def __init__(self, n: int):
        self.n = n
        self.compte: int = 0
        self.file = FileOuPile(mode="File")

    def __call__(self, nv):
        self.file.ajouter(nv)

        if self.compte >= self.n:
            return self.file.prendre()
        else:
            self.compte += 1


class FileDiffOubli:
```

Fichier `policies.py` (vi)

```
    '''
    File différée à facteur d'oubli.

    Identique à une file sauf qu'a chaque ajoute les éléments précédents
se voient appliqués gamma puis ajoutés la nouvelle valeur.

    Initialisation en O(n), ajout en O(n).
    '''

    def __init__(self, n, gamma):
        self.n = n
        self.objets = [None]*n
        self.gamma = gamma
        self.last = 0

    def __call__(self, nv):
        for i in range(self.n):
            if self.objets[i] is not None:
                self.objets[i] = self.gamma(self.objets[i]) + nv
```

Fichier `policies.py` (vii)

```
        res = self.objets[self.last]

        self.objets[self.last] = nv
        self.last = (self.last + 1) % self.n

        return res


class Epsilon_Params_Mem:
    """
    Mémoire pour les différentes variantes de epsilon-greedy.


    Initialisation

        Utilise des files de profondeur n sur les points de données
(observation, action, gain) de l'historique afin ne les traiter qu'avec
n étapes de recul.
```

Fichier `policies.py` (viii)

```
        Pour les gains, utilise une file à facteur d'oubli gamma afin de
prendre en compte les gain long-terme.

        Utilise une tableau numpy pour coder la moyenne des gain obtenus
après avoir reçu une observation et effectué une action.

        Garde en mémoire le epsilon et la fonction eta.


    Apprentissage de nouveaux points de donnée

        Ajoute le nouveau point aux files, et traite le point mature
renvoyé par les files, si elles en renvoient un.


    Meilleure action

        Renvoie, pour une observation donnée, l'action qui s'est révélée
la plus rentable en moyenne.
```

Fichier `policies.py` (ix)

```
    Appliquer eta

        Remplace epsilon par eta(t, epsilon).
    """

    def __init__(self, nb_observation, nb_action, epsilon, n, gamma,
eta, eta_uniq_sur_aleatoire, interpretation):

        # Initialisation des files
        self.a_traiter_obs = FileDiff(n)
        self.a_traiter_action = FileDiff(n)
        self.a_traiter_gain = FileDiffOubli(n, gamma)

        # Initialisation du tableau des gains moyens
        self.Obs_Act_donne_gain_moy = np.array(
            [[Moyenne() for _ in range(nb_action)]
             for _ in range(nb_observation)]
```

Fichier `policies.py` (x)

```
        )

        # Initialisation et préparation de l'évolution de epsilon
        self.epsilon = epsilon
        self.eta = eta
        self.eta_uniq_sur_aleatoire = eta_uniq_sur_aleatoire

        # Paramêtres du POMDP
        self.nb_action = nb_action
        self.nb_observation = nb_observation
        self.interpretation = interpretation

        self.t = 0

    def apprend(self, observation, action, gain):
        act_obs = self.a_traiter_obs(observation)
        act_action = self.a_traiter_action(action)
        act_gain = self.a_traiter_gain(gain)
        if act_obs is not None and act_action is not None and act_gain
```

Fichier `policies.py` (xi)

```
is not None:
            self.Obs_Act_donne_gain_moy[act_obs,
act_action].ajouter(act_gain)

    def meilleure_action_pour_observation(self, observation):
        return np.argmax([self.interpretation(self.t, mem) for mem in
self.Obs_Act_donne_gain_moy[observation]])

    def appliquer_eta(self):
        self.t += 1
        self.epsilon = self.eta(self.t, self.epsilon)


def epsilon_params(
    epsilon=0.1,
    n=2,
    gamma=lambda x: x,
    eta=lambda _, x: x,
    eta_text="off",
```

Fichier `policies.py` (xii)

```
    eta_uniq_sur_aleatoire=False,
    interpretation=lambda _, m: m()
):
    """
    Crée une politique epsilon-greedy, qui choisit une action aléatoire
avec une probabilité epsilon,
    ou choisit l'action avec la meilleure moyenne (avec interpretation)
dans la mémoire.

    La mémoire aura une profondeur n, un facteur d'oubli sur le gain
gamma,
    et appliquera eta à epsilon soit à chaque tour si
eta_uniq_sur_aleatoire=False,
    sinon à chaque fois qu'une action aléatoire et choisie.
    """

    def decide(observation, mem: Epsilon_Params_Mem):
        """
        Fonction de politique epsilon-greedy qui choisit une action en
```

Fichier `policies.py` (xiii)

```
fonction de la mémoire.
        """
        if rd.random() < mem.epsilon:
            mem.appliquer_eta()
            return rd.randint(0, mem.nb_action - 1)
        else:
            if not eta_uniq_sur_aleatoire:
                mem.appliquer_eta()
            return mem.meilleure_action_pour_observation(observation)

    def apprend(observation, action, gain, mem):
        """
        Fonction de mémorisation qui met à jour la mémoire avec
l'observation, l'action et la récompense.
        """
        mem.apprend(observation, action, gain)

    def mem_fab(nb_observation, nb_action):
        """
```

Fichier `policies.py` (xiv)

```
        Fonction qui crée la mémoire pour la politique epsilon-greedy.
        """
        return Epsilon_Params_Mem(
            nb_observation,
            nb_action,
            epsilon,
            n,
            gamma,
            eta,
            eta_uniq_sur_aleatoire,
            interpretation
        )

    return Politique(f"Epsilon-Greedy ({epsilon} n={n}, gamma={gamma},
eta={eta_text}, eta_switch={eta_uniq_sur_aleatoire})",
                     decide_func=decide,
                     apprend_func=apprend,
                     mem_fab_func=mem_fab,)
```

Fichier `machine.py`

```
machine = Markov(4, 5, 4,
                 np.array([
                     [0.8, 0.1, 0.1, 0.0],
                     [0.1, 0.7, 0.2, 0.0],
                     [0.1, 0.2, 0.7, 0.0],
                     [0.0, 0.0, 0.0, 1.0]
                 ]),
                 np.array([
                     [
                         [0.8, 0.1, 0.1, 0.0],
                         [0.9, 0.05, 0.05, 0.0],
                         [0.8, 0.1, 0.1, 0.0],
                         [0.9, 0.05, 0.05, 0.0],
                         [0.0, 0.0, 0.0, 1.0]

                     ],
                     [
                         [0.0, 1.0, 0.0, 0.0],
                         [0.9, 0.1, 0.0, 0.0],
```

Fichier `machine.py` (ii)

```
            [0.0, 1.0, 0.0, 0.0],
            [0.0, 1.0, 0.0, 0.0],
            [0.0, 0.0, 0.0, 1.0]

        ],
        [
            [0.0, 0.0, 1.0, 0.0],
            [0.9, 0.0, 0.1, 0.0],
            [0.0, 0.0, 1.0, 0.0],
            [0.0, 0.0, 1.0, 0.0],
            [0.0, 0.0, 0.0, 1.0]

        ],
        [
            [0.0, 0.0, 0.0, 1.0],
            [0.0, 0.0, 0.0, 1.0],
            [0.8, 0.1, 0.1, 0.0],
            [0.0, 0.0, 0.0, 1.0],
            [0.0, 0.0, 0.0, 1.0]
```

Fichier `machine.py` (iii)

```
                    ]
                ]),
                np.array([
                    [100, -150, 100, 80, 1000],
                    [0, -250, 0, 0, 500],
                    [0, -250, 0, 0, 500],
                    [0, 0, -1500, 0, 0]
                ]),
                )

gaffe = 4

opti = machine.gain_optimal(profondeur=T)
save(opti, f'machine_opti_{T}.pkl')
```

Fichier `simulations.py`

```
def afficher(nm_fichier, duree_sympt, tps, topnb, opti, color, alpha):

    data = load(nm_fichier)
    top = sorted(
        data,
        key=lambda e: np.mean(e[1][-duree_sympt:]),
        reverse=True
    )[:topnb]

    for param, avg, err in top:

        regret = np.array([
            ecart_norm(avg[i], opti[i])
            for i in range(len(avg))
        ])

        err_rel = np.array([
            err[i] / abs(opti[i]) if opti[i] != 0 else 0
            for i in range(len(err))
```

Fichier `simulations.py` (ii)

```
        ])

        plt.plot(
            tps,
            regret,
            color=color,
            label=str(param)
        )

        plt.fill_between(
            tps,
            regret - err_rel,
            regret + err_rel,
            alpha=alpha,
            color=color
        )


def simuler_epsilon(epsilons, machine, T, N, nm_fichier, noisy=False):
```

Fichier `simulations.py` (iii)

```
    res = []
    for epsilon in (tqdm(epsilons) if noisy else epsilons):
        hists = []

        for _ in range(N):
            hist = np.cumsum(
                machine.suivre_politique(
                    epsilon_greedy_pol(epsilon),
                    T
                )
            )
            hists.append(hist)

        hists = np.array(hists)

        avg = np.mean(hists, axis=0)
        err = np.std(hists, axis=0) / np.sqrt(N)

        res.append((epsilon, avg, err))
```

Fichier `simulations.py` (iv)

```
    save(res, nm_fichier)


def simuler_epsilon_w_params(params, machine, T, N, nm_fichier,
noisy=False):
    res = []
    for param in (tqdm(params) if noisy else params):
        hists = []

        for _ in range(N):
            hist = np.cumsum(
                machine.suivre_politique(
                    epsilon_params(**param),
                    T
                )
            )
            hists.append(hist)
```

Fichier `simulations.py` (v)

```
        hists = np.array(hists)

        avg = np.mean(hists, axis=0)
        err = np.std(hists, axis=0) / np.sqrt(N)

        res.append((param, avg, err))

    save(res, nm_fichier)

def _run_simulation(args):
    machine, param, T = args

    policy = epsilon_params(**param)

    hist = np.cumsum(
        machine.suivre_politique(
            policy,
            T
        )
```

Fichier `simulations.py` (vi)

```
    )

    return param, hist


def simuler_epsilon_w_params_parallel(
    params,
    machine,
    T,
    N,
    nm_fichier,
    noisy=False,
):
    workers = max(1, cpu_count() - 1)

    tasks = [
        (machine, param, T)
        for param in params
        for _ in range(N)
```

Fichier `simulations.py` (vii)

```
    ]

    grouped = defaultdict(list)

    with Pool(processes=workers) as pool:
        iterator = pool.imap_unordered(_run_simulation, tasks)

        if noisy:
            iterator = tqdm(iterator, total=len(tasks))

        for param, hist in iterator:
            grouped[repr(param)].append(hist)

    res = []

    for param in params:

        hists = np.array([
            hist
```

Fichier `simulations.py` (viii)

```
            for hist in grouped[repr(param)]
        ])

        avg = np.mean(hists, axis=0)
        err = np.std(hists, axis=0) / np.sqrt(N)

        res.append((param, avg, err))

    save(res, nm_fichier)
```

Fichier `tests.py`

```
T = 30*365  # 30 ans en jours


def gamma1(x): return 0.1*x
def gamma3(x): return 0.3*x
def gamma5(x): return 0.5*x
def gamma7(x): return 0.7*x
def gamma9(x): return 0.9*x
def gamma_id(x): return x


def eta_geom_1(_, x): return 0.9*x
def eta_geom_2(_, x): return 0.99*x
def eta_geom_3(_, x): return 0.999*x
def eta_geom_4(_, x): return 0.9999*x
def eta_geom_5(_, x): return 0.99999*x
def eta_geom_6(_, x): return 0.999999*x
```

Fichier `tests.py` (ii)

```
def eta_log(t, x): return (1 + np.log(1 + (t - 1))) * \
    x / (1 + np.log(1 + t))


def eta_poly_1s4(t, x): return (1 + (t - 1)**(1/4)) * x / (1 +
(t)**(1/4))
def eta_poly_1s3(t, x): return (1 + (t - 1)**(1/3)) * x / (1 +
(t)**(1/3))
def eta_poly_05(t, x): return (1 + np.sqrt(t - 1)) * x / (1 +
np.sqrt(t))
def eta_poly_1(t, x): return (1 + (t - 1)**1) * x / (1 + t**1)
def eta_poly_2(t, x): return (1 + (t - 1)**2) * x / (1 + t**2)
def eta_poly_3(t, x): return (1 + (t - 1)**3) * x / (1 + t**3)


def inter0(_, moy): return moy()
def inter_geom_1(_, moy): return moy() + 1 * np.abs(moy()) / (1 +
moy.compte)
def inter_geom_2(_, moy): return moy() + 2 * np.abs(moy()) / (1 +
```

Fichier `tests.py` (iii)

```
moy.compte)
def inter_geom_3(_, moy): return moy() + 3 * np.abs(moy()) / (1 +
moy.compte)


def inter_ucb_05(t, moy):
    if moy.compte != 0:
        return moy() + .5 * np.sqrt(np.log(1 + t) / moy.compte)
    return float('inf')


def inter_ucb_1(t, moy):
    if moy.compte != 0:
        return moy() + 1 * np.sqrt(np.log(1 + t) / moy.compte)
    return float('inf')


def inter_ucb_2(t, moy):
    if moy.compte != 0:
```

Fichier `tests.py` (iv)

```
        return moy() + 2 * np.sqrt(np.log(1 + t) / moy.compte)
    return float('inf')


def inter_ucb_3(t, moy):
    if moy.compte != 0:
        return moy() + 3 * np.sqrt(np.log(1 + t) / moy.compte)
    return float('inf')


if __name__ == "__main__":

    ##### TEST 1 #####
    N = 40
    params = []

    for epsilon_sk in range(5, 101, 5):
        # de 0.5 (inclus) à 1 (exclus) avec un pas de 0.5
        epsilon = epsilon_sk/100
```

Fichier `tests.py` (v)

```
        params.append(epsilon)

    simuler_epsilon(params, machine, T, N,
                    f"Test1.withStd.{N}.pkl", noisy=True)

    #### TEST 2 ####
    N = 40
    params = []

    for epsilon_sk in range(1, 101, 5):
        # de 0.001 (inclus) à 0.1 (inclus) avec un pas de 5
        epsilon = epsilon_sk/1000
        for n in range(1, 21):
            params.append({
                "epsilon": epsilon,
                "n": n
            })

    simuler_epsilon_w_params_parallel(params, machine, T, N,
```

Fichier `tests.py` (vi)

```
                                        f"Test2.withStd.{N}.pkl",
noisy=True)

    #### TEST 3 ####
    N = 100
    params = []

    for gamma in [gamma1, gamma3, gamma5, gamma7, gamma9]:
        for epsilon in [0.001, 0.006, 0.01, 0.06, 0.1]:
            for n in range(1, 21):
                params.append({
                    "epsilon": epsilon,
                    "n": n,
                    "gamma": gamma
                })

    simuler_epsilon_w_params_parallel(params, machine, T, N,
                                        f"Test3.withStd.{N}.pkl",
noisy=True)
```

Fichier `tests.py` (vii)

```
    #### TEST 4 ####
    N = 100
    params = []

    for n in [3, 5, 10]:
        for gamma in [gamma1, gamma5, gamma7]:
            for epsilon in [0.001, 0.01, 0.1]:
                for eta in [eta_poly_1s4, eta_poly_1s3, eta_log,
eta_poly_05, eta_poly_1, eta_poly_2, eta_poly_3, eta_geom_1, eta_geom_3,
eta_geom_5]:
                    for eta_switch in [True, False]:
                        params.append({
                            "epsilon": epsilon,
                            "n": n,
                            "gamma": gamma,
                            "eta": eta,
                            "eta_uniq_sur_aleatoire": eta_switch
                        })
```

Fichier `tests.py` (viii)

```
    simuler_epsilon_w_params_parallel(params, machine, T, N,
                                      f"Test4.withStd.{N}.pkl",
noisy=True)

    #### TEST 5 ####
    N = 40
    params = []

    for n in [3, 5, 10]:
        for gamma in [gamma1, gamma5, gamma7]:
            for epsilon in [0.001, 0.01, 0.1]:
                for eta in [eta_poly_05, eta_poly_1, eta_poly_3,
eta_geom_1, eta_geom_3, eta_geom_5]:
                    for interpretation in [inter0, inter_geom_1,
inter_geom_3, inter_ucb_05, inter_ucb_1, inter_ucb_3]:
                        params.append({
                            "epsilon": epsilon,
                            "n": n,
```

Fichier `tests.py` (ix)

```
                            "gamma": gamma,
                            "eta": eta,
                            "eta_uniq_sur_aleatoire": eta in
[eta_geom_1, eta_geom_3, eta_geom_5],
                            "interpretation": interpretation
                        })

    simuler_epsilon_w_params_parallel(params, machine, T, N,
                                      f"Test5.withStd.{N}.pkl",
noisy=True)

    #### TEST 6 ####
    N = 40
    ENVS = 10
    HORIZONS = np.arange(1, T//365 + 1) * 365 - 1

    machines = [
        RandomMarkov() for _ in range(ENVS)
    ]
```

Fichier `tests.py` (x)

```
    params = []
    for n in [3, 5, 10]:
        for gamma in [gamma_id, gamma1, gamma5, gamma7]:
            for epsilon in [0.001, 0.01, 0.05, 0.1]:
                for eta in [eta_geom_1, eta_geom_2, eta_geom_3]:
                    for interpretation in [inter0, inter_geom_1,
inter_geom_3]:
                        for eta_switch in [True, False]:
                            params.append({
                                "epsilon": epsilon,
                                "n": n,
                                "gamma": gamma,
                                "eta": eta,
                                "eta_uniq_sur_aleatoire": eta_switch,
                                "interpretation": interpretation
                            })

    for i in range(len(machines)):
```

Fichier `tests.py` (xi)

```
        simuler_epsilon_w_params_parallel(params, machines[i], T, N,
                                          f"Test6.withStd.{i}.{N}.pkl",
noisy=True)

    rows = []
    for i in range(len(machines)):
        opti = machines[i].gain_optimal(profondeur=T)
        res = load(f"Test6.withStd.{i}.{N}.pkl")
        for parami, avgi, erri in res:
            for j, t in enumerate(HORIZONS):
                row = parami.copy()
                row[f"avg(t = {t})"] = ecart_norm(avgi[t], opti[t])
                row[f"err(t = {t})"] = (
                    erri[t] / opti[t]
                    if opti[t] != 0
                    else 0
                )
            rows.append(row)
```

Fichier `tests.py` (xii)

```
    df = pd.DataFrame(rows)
    df.to_csv(f"Test6.N{N}.ENV{ENVS}.csv", index=False)
```

Fichier `affichage.py`

```
opti = load(f'machine_opti_{T}.pkl')
tps = np.linspace(0, T, T)

# Décommenter au besoin

# afficher("Test1.withStd.40.pkl", 365*10, tps, 1, opti, 'blue', .2)
# afficher("Test2.withStd.40.pkl", 365*10, tps, 1, opti, 'red', .2)
# afficher("Test3.withStd.100.pkl", 365*10, tps, 1, opti, 'green', .2)
# afficher("Test4.withStd.100.pkl", 365*10, tps, 1, opti, 'purple', .2)
# afficher("Test5.withStd.40.pkl", 365*10, tps, 1, opti,
'dodgerblue', .2)


# plt.plot([ecart_norm(0, opti[i]) for i in range(T)], color='black')


# afficher("Test5.withStd.40.pkl", 365, tps, 4,
#          opti, 'dodgerblue', .2, linear=True)
```

Fichier `affichage.py` (ii)

```
# plt.xlabel("Temps (js)")
# plt.ylabel("Regret")
# plt.legend()
# plt.show()

# afficher_evo_maximas("Test3.withStd.100.pkl", T)
```