# Mean Field Games Project

Dylan Riboulet

March 14, 2024

## Contents

## 1 Introduction to Mean Field Games

The theory of mean field games (MFGs in short), has been introduced in the pioneering works of J-M. Lasry and P-L. Lions [3, 4, 5], and aims at studying deterministic or stochastic differential games (Nash equilibria) as the number of players tends to infinity

In game theory, a Nash equilibrium is a situation where:

- Each player correctly anticipates the choices of the others;
- Each player maximizes their payoff, given this anticipation.

In other words, a strategy profile $s^* = ((s_i^*)_{i \in 1,n})$ is a Nash equilibrium if each player $i$ plays an optimal strategy $s_i^*$ (which maximizes their payoff $\pi$) given the strategies of the other players $s_j^*$, where $j \in 1, n$:

$$\forall (i,j) \in 1, n^2, \forall s_i \in ((s_i^*)_{i \in 1,n}), \pi(s_i^*, s_j^*) \geq \pi(s_i, s_j^*)$$

Thus, the Nash equilibrium is such that no player regrets their choice (they could not have done better) given the choices of others, with choices being, as always in game theory, simultaneous.

We consider the following MFG: find a flow of probability densities $\hat{m} : [0,T] \times \mathbb{R}^d \to \mathbb{R}$ and a feedback control $\hat{v} : [0,T] \times \mathbb{R}^d \to \mathbb{R}^d$ satisfying the following two conditions:

1. $\hat{v}$ minimizes $J_{\hat{m}} : v \mapsto J_{\hat{m}}(v) = \mathbb{E}\left[\int_0^T f(X_v^t, \hat{m}(t, X_v^t), v(t, X_v^t))dt + \varphi(X_v^T, \hat{m}(T, X_v^T))\right]$ subject to the constraint that the process $X_v = (X_v^t)_{t \geq 0}$ solves the stochastic differential equation (SDE)
$$dX_v^t = b(X_v^t, \hat{m}(t, X_v^t), v(t, X_v^t))dt + \sigma dW_t, \quad t \geq 0, \tag{1}$$
where $\sigma$ is the volatility, $b$ is a given function from $\mathbb{R}^d \times \mathbb{R} \times \mathbb{R}^d$ with values in $\mathbb{R}^d$, and $X_v^0$ is an independent random variable in $\mathbb{R}^d$, distributed according to the law $m_0$;
2. For all $t \in [0,T]$, $\hat{m}(t, \cdot)$ is the law of $X_{\hat{v}}^t$.

It is useful to note that for a given feedback control $v$, the density $m_v^t$ of the law of $X_v^t$ following (1) solves the Kolmogorov-Fokker-Planck (KFP) equation:

$$\frac{\partial m_v}{\partial t}(t,x) - \nu \Delta m_v(t,x) + \operatorname{div}\left(m_v(t,\cdot) b(\cdot, \hat{m}(t,\cdot), v(t,\cdot))\right)(x) = 0, \quad \text{in } (0,T] \times \mathbb{R}^d, \tag{1}$$

$$m_v(0,x) = m_0(x), \quad \text{in } \mathbb{R}^d, \tag{2}$$

where $\nu = \frac{\sigma^2}{2}$.

Let $H : \mathbb{R}^d \times \mathbb{R} \times \mathbb{R}^d \ni (x, m, p) \mapsto H(x, m, p) \in \mathbb{R}$ be the Hamiltonian of the control problem faced by an infinitesimal player. It is defined by

$$H(x, m, p) = \max_{\gamma \in \mathbb{R}^d} \left(-f(x, m, \gamma) - \langle b(x, m, \gamma), p\rangle\right) \in \mathbb{R}. \tag{3}$$

From standard optimal control theory, one can characterize the best strategy through the value function $u$ of the above optimal control problem for a typical player, which satisfies a Hamilton-Jacobi-Bellman (HJB) equation. Together with the equilibrium condition on the distribution, we obtain that the equilibrium best response $\hat{v}$ is characterized by

$$\hat{v}(t, x) = \arg\max_{a \in \mathbb{R}^d} \left(-f(x, m(t, x), a) - \langle b(x, m(t, x), a), \nabla u(t, x)\rangle\right), \tag{4}$$

and, denoting $H_p$ the gradient of $H$ with respect to $p$, that the drift at equilibrium is

$$b(x, m(t, x), \hat{v}(t, x)) = -H_p(x, m(t, x), \nabla u(t, x)). \tag{5}$$

$(u, m)$ solves the following forward-backward PDE system:

$$\begin{cases} -\frac{\partial u}{\partial t}(t, x) - \nu \Delta u(t, x) + H(x, m(t, x), \nabla u(t, x)) = 0, & \text{in } [0, T) \times \mathbb{R}^d, \\ \frac{\partial m}{\partial t}(t, x) - \nu \Delta m(t, x) - \operatorname{div}\left(m(t, \cdot) H_p(\cdot, m(t, \cdot), \nabla u(t, \cdot))\right)(x) = 0, & \text{in } (0, T] \times \mathbb{R}^d, \\ u(T, x) = \varphi(x, m(T, x)), \quad m(0, x) = m_0(x), & \text{in } \mathbb{R}^d. \end{cases} \tag{3a, 3b, 3c}$$

**Remark** The Hamilton-Jacobi equation (3a) is a backward parabolic nonlinear equation and is supplemented with a terminal condition, the left part of (3c). The Komogorov-Fokker-Planck equation (3b) is a forward parabolic linear equation and is supplemented with an initial condition, the right part of (3c).

## 2 Implementation of the Mean Field Game

### 2.1 Simulation Data

The parameters and functions are defined as follows:

- $\Omega = ]0, 1[$, $T = 1$
- $\sigma = 0.2$
- The Hamiltonian $H_0(x, p) = \frac{1}{\beta}|p|^\beta - g(x)$ with $g(x) = -\exp\left(-40(x - \frac{1}{2})^2\right)$. Try $\beta = 2$, $\beta = 1.1$, $\beta = 4$.
- The Discrete Hamiltonian $\widetilde{H}(x, p_1, p_2) = \frac{1}{\beta}\left((p_1^-)^2 + (p_2^+)^2\right)^{\frac{\beta}{2}} - g(x)$
- $\widetilde{f}(m(x)) = \frac{m(x)}{10}$
- $\varphi(x, m) = -\exp\left(-40(x - 0.7)^2\right)$

- $m_0(x) = \exp\left(-3000(x-0.2)^2\right)$ (this is not a probability law because the integral is not 1, but this does not matter).
- $N_h = 201$, $N_T = 100$, $\theta = 0.01$
- Stopping criteria in the Newton method: $10^{-12}$
- Stopping criteria in the Picard fixed-point method: $10^{-6}$ (with norms normalized so that $\|(1,\dots,1)\| = 1$)

**Python Code**

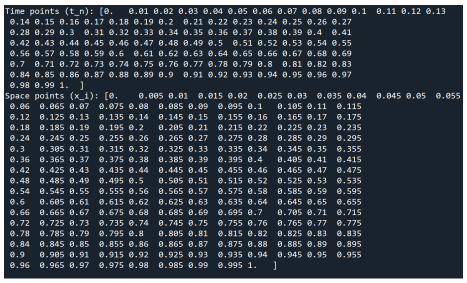
```python
import numpy as np
import matplotlib.pyplot as plt

# Parameters
T = 1.0
N_h = 201
N_T = 100
sigma = 0.2
beta_values = [2, 1.1, 4]
theta = 0.01
tol_newton = 1e-12
tol_picard = 1e-6

# Functions
g = lambda x: -np.exp(-40 * (x - 0.5)**2)
f_tilde = lambda m: m / 10
phi = lambda x, m: -np.exp(-40 * (x - 0.7)**2)
m0 = lambda x: np.exp(-3000 * (x - 0.2)**2)
```

## 2.2 Discretization

Let $N_T$ and $N_h$ be two positive integers. We consider $N_T + 1$ and $N_h$ points in time and space respectively. Set $\Delta t = \frac{T}{N_T}$, $h = \frac{1}{N_h - 1}$, and $t_n = n \times \Delta t$, $x_i = i \times h$ for $(n, i) \in \{0, \dots, N_T\} \times \{0, \dots, N_h - 1\}$.

**Python Code**

```python
def discretize_space_time(T, N_T, N_h):
    # Calculate increments
    delta_t = T / N_T
    h = 1 / (N_h - 1)

    # Generate time points
    t_n = np.linspace(0, T, N_T + 1)

    # Generate space points
    x_i = np.linspace(0, 1, N_h)

    return t_n, x_i

t_n, x_i = discretize_space_time(T, N_T, N_h)
```



```
Time points (t_n): [0.   0.01 0.02 0.03 0.04 0.05 0.06 0.07 0.08 0.09 0.1  0.11 0.12 0.13
 0.14 0.15 0.16 0.17 0.18 0.19 0.2  0.21 0.22 0.23 0.24 0.25 0.26 0.27
 0.28 0.29 0.3  0.31 0.32 0.33 0.34 0.35 0.36 0.37 0.38 0.39 0.4  0.41
 0.42 0.43 0.44 0.45 0.46 0.47 0.48 0.49 0.5  0.51 0.52 0.53 0.54 0.55
 0.56 0.57 0.58 0.59 0.6  0.61 0.62 0.63 0.64 0.65 0.66 0.67 0.68 0.69
 0.7  0.71 0.72 0.73 0.74 0.75 0.76 0.77 0.78 0.79 0.8  0.81 0.82 0.83
 0.84 0.85 0.86 0.87 0.88 0.89 0.9  0.91 0.92 0.93 0.94 0.95 0.96 0.97
 0.98 0.99 1.  ]
Space points (x_i): [0.    0.005 0.01  0.015 0.02  0.025 0.03  0.035 0.04  0.045 0.05  0.055
 0.06  0.065 0.07  0.075 0.08  0.085 0.09  0.095 0.1   0.105 0.11  0.115
 0.12  0.125 0.13  0.135 0.14  0.145 0.15  0.155 0.16  0.165 0.17  0.175
 0.18  0.185 0.19  0.195 0.2   0.205 0.21  0.215 0.22  0.225 0.23  0.235
 0.24  0.245 0.25  0.255 0.26  0.265 0.27  0.275 0.28  0.285 0.29  0.295
 0.3   0.305 0.31  0.315 0.32  0.325 0.33  0.335 0.34  0.345 0.35  0.355
 0.36  0.365 0.37  0.375 0.38  0.385 0.39  0.395 0.4   0.405 0.41  0.415
 0.42  0.425 0.43  0.435 0.44  0.445 0.45  0.455 0.46  0.465 0.47  0.475
 0.48  0.485 0.49  0.495 0.5   0.505 0.51  0.515 0.52  0.525 0.53  0.535
 0.54  0.545 0.55  0.555 0.56  0.565 0.57  0.575 0.58  0.585 0.59  0.595
 0.6   0.605 0.61  0.615 0.62  0.625 0.63  0.635 0.64  0.645 0.65  0.655
 0.66  0.665 0.67  0.675 0.68  0.685 0.69  0.695 0.7   0.705 0.71  0.715
 0.72  0.725 0.73  0.735 0.74  0.745 0.75  0.755 0.76  0.765 0.77  0.775
 0.78  0.785 0.79  0.795 0.8   0.805 0.81  0.815 0.82  0.825 0.83  0.835
 0.84  0.845 0.85  0.855 0.86  0.865 0.87  0.875 0.88  0.885 0.89  0.895
 0.9   0.905 0.91  0.915 0.92  0.925 0.93  0.935 0.94  0.945 0.95  0.955
 0.96  0.965 0.97  0.975 0.98  0.985 0.99  0.995 1.   ]
```

Figure 1: Discretization in time and space

## 2.3 Ghost Nodes

We approximate $u$ and $m$ respectively by vectors $\mathbf{U}$ and $\mathbf{M} \in \mathbb{R}^{(N_T+1)\times N_h}$, that is, $u(t_n, x_i) \approx U_n^i$ and $m(t_n, x_i) \approx M_n^i$ for each $(n, i) \in \{0, \dots, N_T\} \times \{0, \dots, N_h - 1\}$. We use a superscript and a subscript respectively for the time and space indices.

## Python Code

```python
# Adjust x_i to include ghost nodes
# We prepend and append the ghost nodes to the spatial points array
x_ghost = np.hstack(([x_i[0] - h], x_i, [x_i[-1] + h]))

# Initialize U and M matrices with zeros for simplicity
# Now we have N_h + 2 columns due to the two ghost nodes
U = np.zeros((N_T + 1, N_h + 2))
M = np.zeros((N_T + 1, N_h + 2))

print(f"Spatial points with ghost nodes (x_ghost): {x_ghost}")
```

## 2.4 Discrete Neumann boundary conditions

To take into account Neumann boundary conditions, we introduce ghost nodes $x_{-1} = -h$, $x_{N_h} = 1 + h$, and set

$$U_n^{-1} = U_n^0, \quad U_n^{N_h} = U_n^{N_h-1}, \quad M_n^{-1} = M_n^0, \quad M_n^{N_h} = M_n^{N_h-1}. \tag{6}$$

### Python Code

```python
for n in range(N_T + 1):
    # Set ghost node values equal to the nearest interior node
    values
    U[n, 0] = U[n, 1]  # U_n,-1 = U_n,0 in our indexing, ghost node
     at start
    U[n, -1] = U[n, -2]  # U_n,N_h = U_n,N_h-1, ghost node at end

    M[n, 0] = M[n, 1]   # M_n,-1 = M_n,0, same for M
    M[n, -1] = M[n, -2]  # M_n,N_h = M_n,N_h-1
```

## 2.5 Finite Difference Operators

To implement the finite difference operators as described, let's define each operator and then show how they can be implemented in Python. These operators are essential for discretizing derivatives in both time and space, which is a fundamental step in solving partial differential equations numerically.

### 1. Time Derivative Operator $(D_t W)$

The time derivative operator is defined as:

$$D_t W_n = \frac{1}{\Delta t}(W_{n+1} - W_n), \quad n \in \{0, \dots, N_T - 1\}, \quad W \in \mathbb{R}^{N_T+1}$$

### 2. Space Derivative Operator $(DW)$

The space derivative operator is defined as:

$$DW_i = \frac{1}{h}(W_{i+1} - W_i), \quad i \in \{0, \dots, N_h - 1\}, \quad W \in \mathbb{R}^{N_h}$$

### 3. Discrete Laplacian Operator $(\Delta_h W)$

The discrete Laplacian operator is defined as:

$$\Delta_h W_i = -\frac{1}{h^2}(2W_i - W_{i+1} - W_{i-1}), \quad i \in \{0, \dots, N_h - 1\}, \quad W \in \mathbb{R}^{N_h}$$

### 4. Discrete Gradient Operator $(\nabla_h W)$

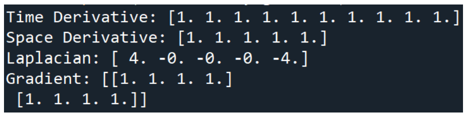
The discrete gradient operator is defined as:

$$[\nabla_h W]_i = ((DW)_i, (DW)_{i-1}) \in \mathbb{R}^2, \quad i \in \{0, \dots, N_h - 1\}, \quad W \in \mathbb{R}^{N_h}$$

In which the special cases $i = 0$ and $i = N_h - 1$ can be written thanks to the above mentionned discrete version of the Neumann boundary conditions.

#### Python Code

```python
def Dt(W, Delta_t):
    """
    Time derivative operator.

    Args:
    - W: np.array, vector in R^(N_T+1) representing a quantity over
      time.
    - Delta_t: float, time step size.

    Returns:
    - np.array of time derivatives of W.
    """
    return (W[1:] - W[:-1]) / Delta_t

def D(W, h):
    """
    Spatial derivative operator.

    Args:
    - W: np.array, vector in R^N_h representing a spatial quantity.
    - h: float, spatial step size.

    Returns:
    - np.array of spatial derivatives of W.
    """
    # Use np.pad to handle Neumann boundary conditions implicitly
    W_padded = np.pad(W, (1, 1), 'edge')
    return (W_padded[2:] - W_padded[:-2]) / (2*h)

def Delta_h(W, h):
    """
    Laplacian operator using central differences.

    Args:
    - W: np.array, vector in R^N_h.
    - h: float, spatial step size.

    Returns:
    - np.array, Laplacian of W.
    """
    W_padded = np.pad(W, (1, 1), 'edge')
    return (2*W - W_padded[2:] - W_padded[:-2]) / h**2

def nabla_h(W, h):
    """
    Gradient operator returning forward and backward spatial
    derivatives.

    Args:
    - W: np.array, vector in R^N_h.
    - h: float, spatial step size.

    Returns:
    - np.array of shape (N_h, 2), gradients of W.
    """
    forward = np.pad(W, (0, 1), 'edge')[1:] - W
    backward = W - np.pad(W, (1, 0), 'edge')[:-1]
    return np.vstack((forward[:-1] / h, backward[1:] / h)).T

# Apply operators
time_derivative = D_tW(W_time,  t )
space_derivative = DW(W_space, h)
laplacian =  hW (W_space, h)
gradient = nabla_hW(W_space, h)
```



```
Time Derivative: [1. 1. 1. 1. 1. 1. 1. 1. 1. 1.]
Space Derivative: [1. 1. 1. 1. 1.]
Laplacian: [ 4. -0. -0. -0. -4.]
Gradient: [[1. 1. 1. 1.]
 [1. 1. 1. 1.]]
```

Figure 2: Finite difference operators with example $N_h = 5$ and $N_T = 10$

## 2.6 Discrete Hamiltonian

Let $\widetilde{H} : \Omega \times \mathbb{R} \times \mathbb{R} \to \mathbb{R}$, $(x, p_1, p_2) \mapsto \widetilde{H}(x, p_1, p_2)$ be a discrete Hamiltonian, assumed to satisfy the following properties:

$(\widetilde{H}_1)$ Monotonicity: for every $x \in \Omega$, $\widetilde{H}$ is nonincreasing in $p_1$ and nondecreasing in $p_2$.

$(\widetilde{H}_2)$ Consistency: for every $x \in \Omega$, $p \in \mathbb{R}$, $\widetilde{H}(x, p, p) = H_0(x, p)$.

$(\widetilde{H}_3)$ Differentiability: for every $x \in \Omega$, $\widetilde{H}$ is of class $C^1$ in $p_1$, $p_2$.

$(\widetilde{H}_4)$ Convexity: for every $x \in \Omega$, $(p_1, p_2) \mapsto \widetilde{H}(x, p_1, p_2)$ is convex.

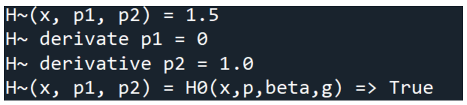
We will consider if $H_0(x, p) = \frac{1}{\beta}|p|^\beta - g(x)$, then we can take $\widetilde{H}(x, p_1, p_2) = \frac{1}{\beta}\left((p_1^-)^2 + (p_2^+)^2\right)^{\frac{\beta}{2}} - g(x)$, where $X^+$, resp. $X^-$ stand for the positive (resp. negative) part of $X$: $X = X^+ - X^-$ and $|X| = X^+ + X^-$ for the simulation of the mean field game.

$$\widetilde{H}_{p1}(x, p_1, p_2, \beta, g) = p_1^- \left( ((p_1^-)^2 + (p_2^+)^2)^{\left(\frac{\beta}{2}-1\right)} \right)$$

$$\widetilde{H}_{p2}(x, p_1, p_2, \beta, g) = p_2^+ \left( ((p_1^-)^2 + (p_2^+)^2)^{\left(\frac{\beta}{2}-1\right)} \right)$$

### Python Code

```python
def positive_part(x):
    return np.maximum(x, 0)

def negative_part(x):
    return np.maximum(-x, 0)

def discrete_hamiltonian(x, p1, p2, beta, g):
    """
    Calculate the discrete Hamiltonian as per Example 2.1.

    Args:
    - x: np.array, spatial domain.
    - p1: float or np.array, momentum component 1.
    - p2: float or np.array, momentum component 2.
    - beta: float, a parameter of the Hamiltonian.
    - g: function, a function g(x) representing the potential.

    Returns:
    - np.array, the discrete Hamiltonian evaluated at (x, p1, p2).
    """

    return ((negative_part(p1) ** 2 + positive_part(p2) ** 2) ** (
    beta / 2)) / beta - g(x)

def H_p1(x, p1, p2, beta, g):
    if p1 <= 0 :
        H_tilde_p1 = p1 * ((negative_part(p1) ** 2 + positive_part(
    p2) ** 2)**((beta / 2)-1))
    else :
        H_tilde_p1 = 0
    return H_tilde_p1

def H_p2(x, p1, p2, beta, g):
    if p2 >= 0 :
        H_tilde_p2 = p2 * ((negative_part(p1) ** 2 + positive_part(
    p2) ** 2)**((beta / 2)-1))
    else :
        H_tilde_p2 = 0
    return H_tilde_p2

def H0(x,p,beta,g):
    """Compute the original Hamiltonian."""
    return (abs(p) ** beta) / beta - g(x)



g = lambda x: -np.exp(-40 * (x - 0.5)**2)

# Example
x = 0.5
p1 = 1
p2 = 1
beta = 2
p = 1

H_tilde_value = discrete_hamiltonian(x, p1, p2, beta, g)
Htilde_p1 = H_p1(x, p1, p2, beta, g) # derivative / p1
Htilde_p2 = H_p2(x, p1, p2, beta, g) # derivative / p2
print("H~(x, p1, p2) =", H_tilde_value)
print("H~ derivate p1 =", Htilde_p1)
print("H~ derivative p2 =", Htilde_p2)
print(H0(x,p,beta,g), H_tilde_value, H0(x,p,beta,g) ==
    H_tilde_value)
print(H0(x,p,beta,g))
```



```
H~(x, p1, p2) = 1.5
H~ derivate p1 = 0
H~ derivative p2 = 1.0
H~(x, p1, p2) = H0(x,p,beta,g) => True
```

Figure 3: Case p1=p2=p=1 and beta = 2

Hypothesis $(\widetilde{H}_1)$] Monotonicity: for every $x \in \Omega$, $\widetilde{H}$ is nonincreasing in $p_1$ and nondecreasing in $p_2$. is verified since the derivate according to p1 is always negative or nul and the derivate according to p2 is always positive or nul

Furthermore, hypothesis $[(\widetilde{H}_2)]$ Consistency: for every $x \in \Omega$, $p \in \mathbb{R}$, $\widetilde{H}(x,p,p) = H_0(x,p)$. is also verified for the example Discrete Hamiltonian

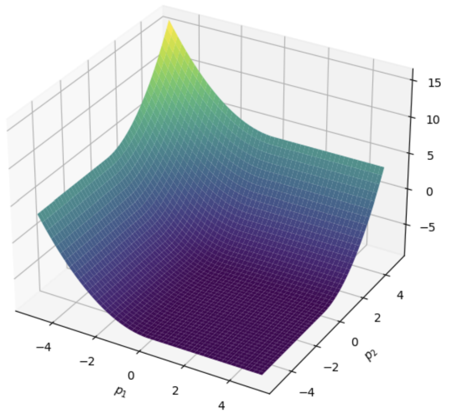


Figure 4: $(\mathrm{p}_1, p_2) \mapsto \widetilde{H}(x, p_1, p_2)$

## Python Code

```python

from mpl_toolkits.mplot3d import Axes3D

# Define the meshgrid for p1 and p2
p1 = np.linspace(-5, 5, 400)
p2 = np.linspace(-5, 5, 400)
p1, p2 = np.meshgrid(p1, p2)

# Calculate H_tilde for each pair of (p1, p2)
H_tilde_values = H_tilde(x_example, p1, p2, beta_example, g_example
    )

# Plot
fig = plt.figure(figsize=(10, 7))
ax = fig.add_subplot(111, projection='3d')
ax.plot_surface(p1, p2, H_tilde_values, cmap='viridis')

ax.set_xlabel('$p_1$')
ax.set_ylabel('$p_2$')
ax.set_zlabel('$\~{H}(x, p_1, p_2)$')
ax.set_title('Plot of $\~{H}(x, p_1, p_2)$')

plt.show()
```

## 3 Solving the discrete HJB equation

The implementation of the discrete Hamilton-Jacobi-Bellman (HJB) equation involves dealing with a backward differential equation in time with Neumann boundary conditions. Given the complexity of the problem, we will break down the implementation into stages and focus on a Python implementation that addresses each part of the HJB equation, as indicated.

The discrete HJB equation is given by:

$$-(D_tU_i)^n - \nu(\Delta hU^n)_i + \tilde{H}(x_i, [\nabla hU^n]_i) = \tilde{f}_0(M_i^{n+1}), \quad 0 \leq i < N_h, \quad 0 \leq n < N_T, \tag{7}$$

$$U_{-1}^n = U_0^n, \quad 0 \leq n < N_T, \tag{8}$$

$$U_{N_h}^n = U_{N_h-1}^n, \quad 0 \leq n < N_T, \tag{9}$$

$$U_i^{N_T} = \phi(M_i^{N_T}), \quad 0 \leq i < N_h. \tag{10}$$

## Discrete HJB Equation Components

Implementing the discrete Hamilton-Jacobi-Bellman (HJB) equation involves addressing a backward differential equation in time with Neumann boundary conditions. We break down the implementation into discrete components as follows.

### Temporal Derivative

The backward difference in time for $U$ at time $n$ and space $i$ is represented as:

$(-D_tU_i)^n$ : The backward difference in time for $U$ at time $n$ and space $i$.

### Laplacian Term

Represents diffusion in space for $U$ at time $n$ and space $i$, scaled by a factor $\nu$:

$\nu(\Delta_h U^n)_i$ : The Laplacian term, representing diffusion in space for $U$ at time $n$ and space $i$, scaled by $\nu$.

### Hamiltonian Term

The discrete Hamiltonian evaluated at space $i$ and time $n$, taking into account the gradient of $U$ at that point:

$H(\tilde{x}_i, [\nabla_h U^n]_i)$ : The Hamiltonian term, evaluated at space $i$ and time $n$, incorporating the gradient of $U$ at t

### Source Term

Represents a source term or a function of the state variable $M$ at the next time step $n + 1$ and space $i$:

$\tilde{f}_0(M_i^{n+1})$ : The source term, representing a function of the state variable $M$ at the next time step $n + 1$ and

## Terminal and Neumann Boundary Conditions

The value of $U$ at the final time step $N_T$ and the conditions at the spatial boundaries are critical for solving the HJB equation. Terminal condition is given by $U$ at the final time step, and Neumann boundary conditions are applied at the spatial boundaries to ensure no flux across them.

## Definition of $F$

The function $F$ is defined as:

$$F(U^n, U^{n+1}, M^{n+1}) = -(D_t U)^n - \nu(\Delta_h U^n) + H(x, [\nabla_h U^n]) - \tilde{f}_0(M_i^{n+1}).$$

This function $F$ represents the discrepancy between the two sides of the equation for each point in space at a given time step $n$.

## Implementation of $F$

To implement $F$, we consider the components of the discrete HJB equation including the temporal derivative, the Laplacian term, the Hamiltonian term, and the source term. The implementation involves calculating these components at each spatial point and time step, ensuring the equation $F(U^n, U^{n+1}, M^{n+1}) = 0$ is satisfied.

#### Python Code

## 3.1 Computation of F

```python
def compute_F(U_n, U_np1, M_np1, nu, h,  t , x, beta, g):
    """
    Compute the function F for the HJB equation.

    Parameters:
    - U_n: The current estimate of U at time step n.
    - U_np1: U at the next time step (n+1).
    - M_np1: M at the next time step (n+1).
    - nu: The diffusion coefficient.
    - h: The spatial discretization step.
    -  t : The temporal discretization step.
    - x: The spatial domain.
    - beta: A parameter for the Hamiltonian.
    - g: The function g(x) used in the Hamiltonian.

    Returns:
    - The value of F for each point in space at time step n.
    """
    D_tU_n = D_tW(U_n,  t )  # Assuming D_tW implements the backward difference in time
     hU_n  =  hW (U_n, h)  # Laplacian of U at time n
    grad_U_n = nabla_hW(U_n, h)  # Gradient of U at time n

    # Hamiltonian term. Note: This requires appropriate handling of
     grad_U_n to extract p1, p2
    H_tilde_terms = np.array([discrete_hamiltonian(xi, p1, p2, beta
    , g(xi)) for xi, (p1, p2) in zip(x, grad_U_n.T)])

    # Source term
    f_tilde_terms = f_tilde_0(M_np1)

    # F(U^n, U^{n+1}, M^{n+1})
    F_values = -D_tU_n - nu *  hU_n  + H_tilde_terms -
    f_tilde_terms

    return F_values
```

To solve the nonlinear system $F(U^n, U^{n+1}, M^{n+1}) = 0$ for the discrete Hamilton-Jacobi-Bellman (HJB) equation with Neumann boundary conditions, we apply the Newton-Raphson method. This section outlines the computational approach to solving $F(U^n, U^{n+1}, M^{n+1}) = 0$ at a specific timestep $n$.

The Newton-Raphson method is an iterative method used for finding successively better approximations to the roots (or zeroes) of a real-valued function. To solve the equation $F(U^n, U^{n+1}, M^{n+1}) = 0$, the method is applied as follows:

First, we compute the Jacobian $J$ of $F$ with respect to $U^n$, which involves partial derivatives of $F$ with respect to each element of $U^n$.

The $U^n$ value is updated iteratively according to the formula:

$$U^{n,k+1} = U^{n,k} - J^{-1}(U^{n,k}, U^{n+1}, M^{n+1})F(U^{n,k}, U^{n+1}, M^{n+1})$$

where $U^{n,k}$ is the approximation of $U^n$ at the $k$-th iteration. This process involves:

- Computing the Jacobian matrix $J$.
- Inverting $J$ (or solving the corresponding linear system).
- Iteratively updating $U^n$ until convergence is achieved.

The Newton-Raphson method offers a powerful tool for solving the nonlinear system $F(U^n, U^{n+1}, M^{n+1}) = 0$, integral to the backward time-marching solution of the discrete HJB equation. By iteratively updating $U^n$ with respect to $J$ and $F$, we approach the solution with increasing accuracy until convergence.

## 3.2 Computation of the Jacobian Matrix

Computing the Jacobian of the map $\mathbb{R}^{N_h} \ni U \mapsto (H(\tilde{x}_i, [\nabla_h U]_i))_{0 \le i < N_h}$ involves understanding how small changes in the vector $U$ affect the values of the discrete Hamiltonian $\tilde{H}$ at each point in the spatial discretization. The Jacobian matrix $J$ will have dimensions $N_h \times N_h$, with each element defined as:

$$J_{ij} = \frac{\partial H(\tilde{x}_i, [\nabla_h U]_i)}{\partial U_j}$$

Given the functions $H_{p1}$ and $H_{p2}$ for computing the partial derivatives of the discrete Hamiltonian $\tilde{H}$ with respect to $p_1$ and $p_2$, we can now fully implement the computation of the Jacobian matrix $J$. This implementation will incorporate the derivatives obtained from $H_{p1}$ and $H_{p2}$ and will handle the dependencies between the spatial points in $U$ through the gradient $[\nabla_h U]$.

The Jacobian matrix $J$ at a given time step is crucial for solving the discrete HJB equation using the Newton-Raphson method. It represents how the discretized nonlinear system $F(U^n, U^{n+1}, M^{n+1}) = 0$ changes with small variations in $U^n$ and is defined as:

$$J_{ij} = \frac{\partial F(U^n)}{\partial U_j^n}$$

Considering the spatial derivatives $[\nabla_h U]_i = ((DW)_i, (DW)_{i-1})$ involves computing $DW$, which represents the discrete spatial derivative of $U$ with respect to the space. We integrate the provided partial derivatives $H_{p1}$ and $H_{p2}$ into the computation.

Implementing the Jacobian Matrix $J$

**Python Code**

```python
def compute_gradient(U, h):
    # Compute forward and backward differences
    forward_diff = np.diff(U) / h
    backward_diff = np.roll(forward_diff, 1)
    # Handle boundary conditions (Neumann)
    backward_diff[0] = 0  # Assuming symmetric boundary at the start
    forward_diff = np.append(forward_diff, forward_diff[-1])  # Extend to maintain size
    return forward_diff, backward_diff

def compute_jacobian(U, h, x, beta, g, N_h):
    """
    Compute the Jacobian matrix of the map U -> H~(xi, [nabla_h U]i).

    Parameters:
    - U: The current estimate of U.
    - h: The spatial discretization step.
    - x: The spatial domain.
    - beta: A parameter for the Hamiltonian.
    - g: The function g(x).
    - N_h: Number of spatial discretization points.

    Returns:
    - J: The Jacobian matrix of dimensions N_h x N_h.
    """
    J = np.zeros((N_h, N_h))  # Initialize the Jacobian matrix

    p1_forward, p1_backward = compute_gradient(U, h)
    p2_forward, p2_backward = p1_forward, p1_backward  # For this Hamiltonian, p1 and p2 gradients are computed the same way

    for i in range(N_h):
        for j in range(max(0, i-1), min(i+2, N_h)):
            if j == i:  # Diagonal elements
                J[i, j] += H_p1(x[i], p1_forward[i], p2_forward[i], beta, g(x[i]))
                J[i, j] += H_p2(x[i], p1_backward[i], p2_backward[i], beta, g(x[i]))
            elif j == i-1:  # Elements to the left of diagonal
                J[i, j] += H_p2(x[i], p1_backward[i], p2_backward[i], beta, g(x[i]))
            elif j == i+1:  # Elements to the right of diagonal
                J[i, j] += H_p1(x[i], p1_forward[i], p2_forward[i], beta, g(x[i]))

    return J
```

**Python Code**

```python
def compute_jacobian(U_n, U_np1, M_np1, x, h, Delta_t, beta, nu, g):
    N_h = len(x)
    J = np.zeros((N_h, N_h))  # Initialize Jacobian matrix

    epsilon = 1e-5  # Small perturbation for numerical differentiation

    for i in range(N_h):
        # Perturb U_n at the i-th position
        U_n_perturbed = U_n.copy()
        U_n_perturbed[i] += epsilon

        # Compute F with and without perturbation
        F_original = F(U_n, U_np1, M_np1, x, h, Delta_t, beta, nu, g)
        F_perturbed = F(U_n_perturbed, U_np1, M_np1, x, h, Delta_t, beta, nu, g)
        print(F_original)
        print(F_perturbed)
        # Numerical derivative (partial derivative of F with respect to U_n[i])
        J[:, i] = (F_perturbed - F_original) / epsilon
        print(J)

    return J

This implementation computes the Jacobian matrix $J$ based on the spatial discretization of $U$ and incorporates the partial derivatives of the discrete Hamiltonian as provided by $H_{p1}$ and $H_{p2}$. Note that this implementation assumes that $g(x)$ is defined elsewhere in your code and is accessible within this function.

The \texttt{compute\_gradient} function calculates the forward and backward differences to approximate the spatial derivatives, accounting for Neumann boundary conditions at the domain
```

```
    boundaries. The Jacobian matrix is then assembled by filling in
     the appropriate partial derivatives for each spatial point,
    considering the relationships between adjacent points in the
    discretized spatial domain.
```

## 3.3 Newton-Raphson iterations to solve HJB Equation

To solve the discrete Hamilton-Jacobi-Bellman (HJB) equation using the functions we've implemented, we'll follow the backward time-marching procedure outlined previously. This involves starting from the terminal time step $N_T$ and iteratively solving the equation backward in time using the Newton-Raphson method at each step.

Given that we already have:

- The functions for computing the Hamiltonian derivatives $H_{p1}$ and $H_{p2}$.
- A function `compute_jacobian` for the Jacobian matrix $J$.
- The functions `positive_part` and `negative_part`.

We'll now implement the procedure to solve the discrete HJB equation. This solution requires an iterative process at each time step, solving the nonlinear system $F(U^n, U^{n+1}, M^{n+1}) = 0$ using Newton-Raphson iterations.

**Python Code**

```python
def newton_raphson_for_hjb(U_n_plus_1, M_n_plus_1, x, beta, sigma,
    Nh, h, delta_t, tol=1e-12, max_iter=100):
    """
    Solve the HJB equation for U_n using the Newton-Raphson method.

    Parameters:
    - U_n_plus_1: Approximation of U at time step n+1.
    - M_n_plus_1: M at time step n+1.
    - x, beta, sigma, Nh, h, delta_t: Parameters and discretization
     variables.
    - tol: Tolerance for the stopping criterion.
    - max_iter: Maximum number of iterations for the Newton-Raphson
     method.

    Returns:
    - U_n: Solution for U at time step n.
    """
    U_n = U_n_plus_1.copy()

    for iteration in range(max_iter):
        print(iteration)
        # Compute F and the Jacobian J at the current approximation
        F_val = compute_F(U_n, U_n_plus_1, M_n_plus_1, x, beta,
    sigma, Nh, h, delta_t)
        J_val = compute_Jacobian(U_n, U_n_plus_1, M_n_plus_1, x,
    beta, sigma, Nh, h, delta_t)
```

```python

        # Solve the linear system J * delta = -F to find the update delta
        delta = np.linalg.solve(J_val, -F_val)

        # Update U_n
        U_n += delta
        print(np.linalg.norm(delta, np.inf))
        print(tol)
        # Check for convergence
        if np.linalg.norm(delta, np.inf) < tol:
            break

    return U_n
```

## 3.4 Solving HJB backward

### Python Code

```python
def solve_hjb_backward_with_newton(U, M, x, beta, sigma, Nh, NT, h, delta_t):
    """
    Solve the HJB equation backward in time using the Newton-Raphson method.

    Parameters:
    - U: Initial matrix for U, with terminal condition set.
    - M: Matrix for M.
    - x, beta, sigma, Nh, NT, h, delta_t: Problem parameters and discretization variables.

    Returns:
    - The updated matrix U after solving the HJB equation.
    """
    for n in reversed(range(NT)):
        U_n_plus_1 = U[:, n+1]
        M_n_plus_1 = M[:, n+1]
        U[:, n] = newton_raphson_for_hjb(U_n_plus_1, M_n_plus_1, x, beta, sigma, Nh, h, delta_t)
        print(U_n_plus_1)
        print(M_n_plus_1)
    #  Neumann boundary conditions as per the problem statement
    U[0, :] = U[1, :]
    U[-1, :] = U[-2, :]

    return U
```

## 4 Solving the discrete Fokker-Planck equation

### 4.1 Discrete transport operator

$T_i$ is the discrete transport operator defined as :

$$T_i(U,M) = \frac{1}{h}\big(M_i \widetilde{H}_{p_1}(x_i,[\nabla_h U]_i) - M_{i-1}\widetilde{H}_{p_1}(x_{i-1},[\nabla_h U]_{i-1}) \\ + M_{i+1}\widetilde{H}_{p_2}(x_{i+1},[\nabla_h U]_{i+1}) - M_i \widetilde{H}_{p_2}(x_i,[\nabla_h U]_i)\big)$$

**Python Code**

```python
def T_i(U, M, i, h, beta):
    """
    Compute the discrete transport operator Ti for a given spatial index i.

    Parameters:
    - U: The U matrix for all time steps and spatial points.
    - M: The M matrix for all time steps and spatial points.
    - i: The current spatial index.
    - h: The spatial discretization step.
    - beta: The beta parameter for the Hamiltonian.

    Returns:
    - The value of the discrete transport operator Ti at spatial index i.
    """
    # Ensure i is within valid range
    if i == 0 or i == len(M) - 1:
        return 0

    # Gradient of U at i, using central difference
    grad_h_U_i = [(U[i+1] - U[i-1]) / (2 * h), (U[i+1] - U[i-1]) / (2 * h)]  # Assuming symmetric for simplicity

    # Compute H _p1 and H _p2
    H_p1_i = H_tilde_p1(x[i], grad_h_U_i, beta)
    H_p2_i_plus_1 = H_tilde_p2(x[i+1], grad_h_U_i, beta)  # Assuming H_p2 depends on the gradient at i for simplicity

    # Compute Ti using the provided formula
    Ti_value = (1/h) * (
        M[i] * H_p1_i - M[i-1] * H_p1_i +
        M[i+1] * H_p2_i_plus_1 - M[i] * H_p2_i_plus_1
    )

    return Ti_value
```

## 4.2 Resolution

The discrete KFP equation is given by:

$$(D_t M_i)^n - \nu(\Delta h M_{n+1})_i - T_i(U^n, M_{n+1}) = 0, \quad 0 \le i < N_h, \quad 0 \le n < N_T, \tag{11}$$

$$M^n_{-1} = M^n_0, \quad 0 < n \le N_T, \tag{12}$$

$$M^n_{N_h} = M^n_{N_h-1}, \quad 0 < n \le N_T, \tag{13}$$

$$M^0_i = \bar{m}_0(x_i), \quad 0 \le i < N_h. \tag{14}$$

The discrete Kolmogorov-Forward (KFP) equation involves a time derivative of the distribution $M$, a diffusion term with Laplacian $\Delta_h M$, and a transport term $T_i(U, M)$, with Neumann boundary conditions. Solving this equation requires iteratively updating $M$ forward in time, starting from an initial condition $M_0$.

Given:

- $(D_t M_i)^n$ represents the forward difference in time for $M$ at time $n$ and space $i$.
- $\nu(\Delta_h M^{n+1})_i$ is the diffusion term for $M$ at the next time step $n+1$ and space $i$, scaled by a diffusion coefficient $\nu$.
- $T_i(U^n, M^{n+1})$ is the discrete transport operator at space $i$, given $U$ at time $n$ and $M$ at the next time step $n+1$.
- The initial condition $M^0_i = \bar{m}_0(x_i)$, where $\bar{m}_0(x_i) = m_0(x_i)$ and $m_0(x) = \exp(-3000(x-0.2)^2)$.

**Python Code**

```
from scipy.sparse import diags
from scipy.sparse.linalg import spsolve

def solve_fp_forward(U, M, sigma, Nh, NT, h, delta_t, beta):
    """
    Solve the FP equation forward in time using an implicit scheme.
    """
    # Precompute constants for the diagonal matrix construction
    diffusion_term = sigma / h**2
    main_diag = 1 + 2 * diffusion_term * delta_t  # Main diagonal
    off_diag = -diffusion_term * delta_t  # Off diagonal

    # Construct the sparse matrix A (implicit scheme)
    diagonals = [main_diag * np.ones(Nh), off_diag * np.ones(Nh-1),
     off_diag * np.ones(Nh-1)]
    A = diags(diagonals, [0, -1, 1], format="csr")

    for n in range(NT):
        U_n_plus_1 = U[:, n+1]

        # Compute the right-hand side b
        b = M[:, n]

        # Modify b based on the transport effect (Ti)
        # This is a simplified representation; adjust based on actual Ti implementation
        for i in range(1, Nh-1):
            transport_effect = T_i(U_n_plus_1, M[:, n], i, h, beta)
            b[i] -= transport_effect * delta_t

        # Solve the linear system A * M_{n+1} = b
        M[:, n+1] = spsolve(A, b)

        # Apply Neumann boundary conditions
        M[0, n+1] = M[1, n+1]
        M[-1, n+1] = M[-2, n+1]

    return M
```

## 5 Fixed point iterations for the whole forward-backward system

### 5.1 Picard fixed point iterations

Let $M$ stand for the collection $(M_n)_{n=0,\dots,N_T}$ and $U$ stand for the collection $(U_n)_{n=0,\dots,N_T}$. The program consists of approximating $(M,U)$ by Picard fixed point iterations. Let $\theta$ be a parameter, $0<\theta<1$, $\theta=0.01$ is often a sensible choice. Let $(M^{(k)},U^{(k)})$ be the running approximation of $(M,U)$. The next approximation $(M^{(k+1)},U^{(k+1)})$ is computed as follows:

1. Solve the discrete HJB equation given $M^{(k)}$. The solution is named $U^{(k+1)}$.
2. Solve the discrete FP equation given $U^{(k+1)}$. The solution is named $M^{(k+1)}$.
3. Set $(M^{(k+1)},U^{(k+1)}) = (1-\theta)(M^{(k)},U^{(k)}) + \theta(M^{(k+1)},U^{(k+1)})$.

These iterations are stopped when the norm of the increment $(M^{(k+1)},U^{(k+1)}) - (M^{(k)},U^{(k)})$ becomes smaller than a given threshold, say $10^{-7}$.

#### Python Code

```
import numpy as np
import numpy.linalg as la

def picard_iteration(M_initial, U_initial, x, Nh, NT, h, delta_t, sigma, beta, theta=0.01, tol=1e-7, max_iter=100):
    """
    Perform Picard fixed point iterations to approximate (M, U).

    Parameters:
    - M_initial, U_initial: Initial guesses for M and U.
    - x, Nh, NT, h, delta_t, sigma, beta: Parameters and
discretization variables.
    - theta: Mixing parameter for the Picard iteration.
    - tol: Tolerance for convergence.
    - max_iter: Maximum number of iterations.

    Returns:
    - M, U: Approximations of M and U after convergence.
    """
    M = M_initial.copy()
    U = U_initial.copy()

    for k in range(max_iter):
        # Step 1: Solve the HJB equation for Ub given current M
        Ub = solve_hjb_backward_with_newton(U, M, x, beta, sigma,
Nh, NT, h, delta_t)
        print(Ub)

        # Step 2: Solve the FP equation for Mc given updated Ub
        Mc = solve_fp_forward(Ub, M, sigma, Nh, NT, h, delta_t,
beta)
        print(Mc)

        # Step 3: Update (M, U) using the mixing parameter theta
        M_next = (1 - theta) * M + theta * Mc
        U_next = (1 - theta) * U + theta * Ub

        # Check for convergence based on the norm of the increment
        delta_M = la.norm(M_next - M, np.inf)
        delta_U = la.norm(U_next - U, np.inf)

        if delta_M < tol and delta_U < tol:
            print(f"Convergence achieved after {k+1} iterations.")
            break

        M, U = M_next, U_next

    return M, U
```

## 5.2 Outputs

### 5.2.1 The contour lines of u and m in the plane (x,t)

#### Python Code

```
for beta in beta_values:
    # Solve the MFG using Picard iterations
    M, U = picard_iteration(M_initial, U_initial, x, Nh, NT, h,
    delta_t, sigma, beta, theta, tol_picard)
    # Plot contour lines of u and m
    plt.figure(figsize=(12, 6))
    plt.subplot(1, 2, 1)
    plt.contourf(x, np.linspace(0, T, NT+1), U.T, levels=50)
    plt.colorbar()
    plt.title(f"Contour of u for beta={beta}")
    plt.xlabel('x')
    plt.ylabel('t')

    plt.subplot(1, 2, 2)
    plt.contourf(x, np.linspace(0, T, NT+1), M.T, levels=50)
    plt.colorbar()
    plt.title(f"Contour of m for beta={beta}")
    plt.xlabel('x')
    plt.ylabel('t')

    plt.tight_layout()
    plt.show()
```

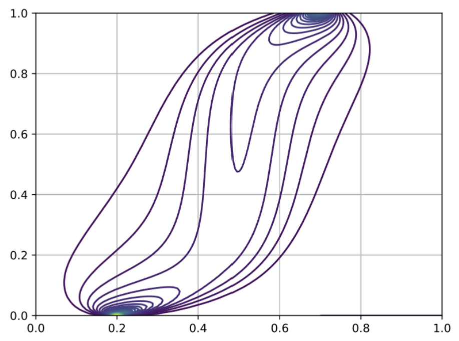


Figure 5: Figure 1: Contour lines of $m$ in the example described above with $g(x) = -\exp(-40(x-\frac{1}{2})^2)$ and $\beta = 2$ ($x$ in abscissa, $t$ in ordinate).

#### 5.2.2 Animations (in the mp4 format) showing the evolution of u and m with respect to t

**Python Code**

```
from matplotlib.animation import FuncAnimation
from matplotlib.animation import FFMpegWriter

Nh, NT = U.shape
x = np.linspace(0, 1, Nh)  # Update according to your actual
    spatial domain

fig, ax = plt.subplots(2, 1, figsize=(10, 8))

ax[0].set_xlim((0, 1))
ax[0].set_ylim((np.min(U), np.max(U)))
ax[1].set_xlim((0, 1))
ax[1].set_ylim((np.min(M), np.max(M)))

line1, = ax[0].plot([], [], lw=2)
```

```python
line2, = ax[1].plot([], [], lw=2)

ax[0].set_title('Evolution of u over Time')
ax[1].set_title('Evolution of m over Time')

ax[0].set_xlabel('x')
ax[0].set_ylabel('u')

ax[1].set_xlabel('x')
ax[1].set_ylabel('m')

def init():
    line1.set_data([], [])
    line2.set_data([], [])
    return line1, line2

def animate(i):
    line1.set_data(x, U[:, i])
    line2.set_data(x, M[:, i])
    return line1, line2

writer = FFMpegWriter(fps=30, extra_args=['-vcodec', 'libx264'])

anim = FuncAnimation(fig, animate, init_func=init, frames=NT,
    interval=100, blit=True)

anim.save('mfg_evolution.mp4', writer=writer)

plt.show()
```

### 5.3 Second Case of function g

$$g(x) = -\exp\left(-40\left(x - \frac{1}{3}\right)^2\right) - \exp\left(-40\left(x - \frac{2}{3}\right)^2\right)$$

```python
def g(x):
    return -np.exp(-40 * (x - 1/3)**2) - np.exp(-40 * (x - 2/3)**2)
```

## 6 Conservation of the total mass of m

## Introduction

The mass of $M_n$, defined as

$$\sum_{i=0}^{N_h-1} M_n^i,$$

remaining constant and not depending on $n$ in the context of Mean Field Games (MFGs) can be understood through the properties of the Fokker-Planck (FP) equation which governs the evolution of $M$. The FP equation describes the

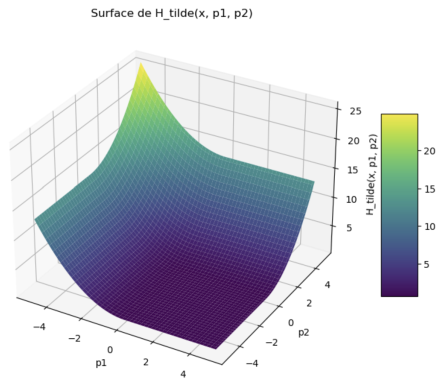


Figure 6: $(\mathrm{p}_1, p_2) \mapsto \widetilde{H}(x, p_1, p_2)$

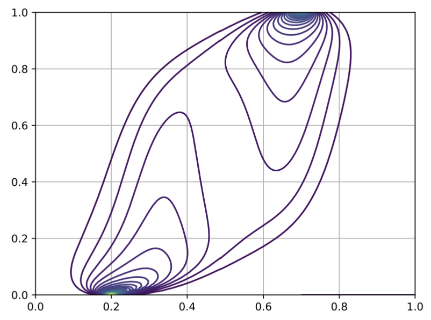


Figure 7: Figure 2: Contour lines of $m$ in the example described above with $g(x) = -\exp(-40(x - \frac{1}{3})^2) - \exp(-40(x - \frac{2}{3})^2)$ and $\beta = 2$ ($x$ in abcissa, $t$ in ordinate).

density evolution of agents in the game over time and space, accounting for both diffusion (spread) and drift (directed motion).

## Key Points for Constant Mass

### Conservation Law

The FP equation, in its continuous form, represents a conservation law. It implies that the total mass (or the total number of agents) in the system is conserved over time if there are no sources or sinks within the domain. This conservation is inherent in the equation's structure, reflecting the physical principle that agents can move or spread out but cannot magically appear or disappear.

### Neumann Boundary Conditions

The imposition of Neumann boundary conditions further ensures mass conservation. These boundary conditions imply that there is no flux of agents across the domain's boundaries; agents are neither entering nor leaving the domain, thus preserving the total number within the domain over time.

### Implicit Scheme and Linearity

The solution approach for the FP equation, especially when using implicit time-stepping schemes, maintains this conservation property at each time step. The linearity of the FP equation (assuming the drift and diffusion coefficients do not depend on $M$) under the specified boundary conditions ensures that any initial mass of $M_0$ is propagated through each subsequent time step without change.

### Numerical Discretization

The numerical discretization and solution of the FP equation (e.g., finite difference methods), when correctly implemented with the conservation law and boundary conditions in mind, inherently preserves the total mass. The discretized equations, solved at each time step, account for the redistribution of $M$ across the spatial domain without adding to or subtracting from the total.

## Mathematical Insight

The conservation of mass in the FP equation can be seen from its integral form over the spatial domain. When integrated, the diffusion term (second spatial derivative) becomes a boundary term due to the divergence theorem, which is nullified by Neumann boundary conditions. The drift term (first spatial derivative), when integrated, also becomes a boundary term, which again is nullified by the boundary conditions. Hence, the integral of $M$ over space, representing the total mass, does not change over time.

## Conclusion

The constancy of the mass of $M_n$ across time steps $n$ is a fundamental property stemming from the physical principles of conservation, the mathematical structure of the FP equation, and the specific boundary conditions applied. This ensures that the numerical simulation of the system accurately reflects the expected behavior of the underlying physical or theoretical model being represented.

# 7 Uniqueness in Discrete HJB Equation

*Proof.* Pour prouver l'unicité de la solution à l'équation discrète de Hamilton-Jacobi-Bellman (HJB) compte tenu des conditions énoncées, nous utiliserons un argument de contradiction en exploitant la propriété de monotonie du Hamiltonien discret ($\tilde{H}$). Nous supposerons qu'il existe deux solutions différentes $U$ et $V$ à l'équation discrète HJB et démontrerons que cela conduit à une contradiction, prouvant ainsi que la solution doit être unique.

Soit $\tilde{U}$ et $\tilde{V}$ deux solutions de l'équation discrète HJB. Cela signifie que les deux satisfont :

$$-(D_tU)_n - \nu(\Delta_h U)_n + \tilde{H}(x_i, [\nabla_h U]_n) = \tilde{f}_0(M_{n+1})$$
$$-(D_tV)_n - \nu(\Delta_h V)_n + \tilde{H}(x_i, [\nabla_h V]_n) = \tilde{f}_0(M_{n+1})$$

pour tout $0 \leq i < N_h$, $0 \leq n < N_T$, avec les conditions aux limites données.

Considérons la différence $W_n = U_n - V_n$. Nous sommes intéressés par le maximum de cette différence sur tous les $n$ et $i$. Par l'hypothèse d'avoir deux solutions, il existe au moins une paire $(n_0, i_0)$ telle que $W_{n_0 i_0} = \max_{n,i}(U_{ni} - V_{ni})$ est positif (autrement, si la différence maximale est zéro, les solutions ne sont pas différentes).

À $(n_0, i_0)$, les deux solutions $U$ et $V$ satisfont à l'équation discrète HJB, donc nous pouvons soustraire les équations pour $V$ de $U$ pour obtenir une équation pour $W$. Cependant, pour exploiter la monotonie de $\tilde{H}$, nous observons que puisque $\tilde{H}$ est monotone dans ses arguments :

$$\tilde{H}(x_{i_0}, [\nabla_h U]_{n_0}) - \tilde{H}(x_{i_0}, [\nabla_h V]_{n_0})$$

Le signe de cette expression est déterminé par la monotonie de $\tilde{H}$ par rapport à ses arguments. Si $U_{n_0,i_0} > V_{n_0,i_0}$, alors les termes de gradient $[\nabla_h U]_{n_0}$ et $[\nabla_h V]_{n_0}$ conduisent à une contradiction avec la maximalité supposée de $W_{n_0 i_0}$ si $\tilde{H}$ devait croitre, en raison de ses propriétés de monotonie.

La contradiction apparait parce que, si $W_{n_0 i_0}$ est en effet la différence strictement positive maximale, alors l'équation discrète HJB obligerait $W$ à être non positive en ce point en raison des propriétés de $\tilde{H}$. Cela signifie que $U_{n_0 i_0}$ ne peut pas être strictement supérieur à $V_{n_0 i_0}$, contredisant l'hypothèse que $W_{n_0 i_0}$ est positif.

Le cas $U_{n_0,i_0} < V_{n_0,i_0}$ est obtenue par symétrie.

$\square$

## 8 Uniqueness in Discrete KFP equation

On suppose que $U_i^{N_T} = \phi(M_i^{N_T})$

d'après l'équation de KFP discrète :

$$(D_t M_i)^n - \nu(\Delta_h M^{n+1})_i - \mathcal{T}_i(U^n, M^{n+1}) = 0$$

$$M_i^0 = \overline{m_0}(x_i) = m_0(x_i)$$

$$M_i^n \simeq m(x_i) = \overline{m}(x_i)$$

Soit $i \in [\![0, N_h]\!]$

Soit $n \in [\![0, N_T]\!]$

On cherche $M$ tel que

$$\frac{1}{\Delta t}(M_i^{n+1} - M_i^n) - \nu(\Delta_h M^{n+1})_i - \mathcal{T}_i(U^n, M^{n+1}) = 0$$

avec $\mathcal{T}_i(U, M) = \frac{1}{h}\Big(M_i \tilde{H}_{p_1}(x_i, [\nabla_h U]_i) - M_{i-1} \tilde{H}_{p_1}(x_{i-1}, [\nabla_h U]_{i-1})$

$+ M_{i+1} \tilde{H}_{p_2}(x_{i+1}, [\nabla_h U]_{i+1}) - M_i \tilde{H}_{p_2}(x_i, [\nabla_h U]_i)\Big)$

et $(\Delta_h M)_i = -\frac{1}{h^2}(2M_i - M_{i+1} - M_{i-1})$

donc cela revient à écrire

$$\frac{M_i^{n+1}}{\Delta t} - \nu(\Delta_h M^{n+1})_i - \mathcal{T}_i(U^n, M^{n+1}) = \frac{M_i^n}{\Delta t}$$

ce problème linéaire peut se réécrire : $\mu \cdot M + AM = \mu M_i^n$

où $A$ est un opérateur linéaire dépendant de $U$

Les hypothèses de monotonicité de $\tilde{H}$ implique $\frac{\partial \tilde{H}}{\partial p_1}(x, p_1, p_2) \leq 0$ et $\frac{\partial \tilde{H}(x, p_1, p_2)}{\partial p_2} \geq 0$

ce qui implique que la matrice correspondant à $A$ a des coefficients diagonaux positifs et des coefficients non diagonaux négatifs.

De plus, comme $\tilde{H} \in \mathcal{C}^1$ de $\|D_h U^{(e)}\|_\infty \leq c(h)$ implique qu'il existe une constante $C$ indépendante de $M$ (mais possible de $h$) tel que $\forall p \in \{1, 2\}$, $\left|\frac{\partial \tilde{H}}{\partial p_p}(x, [D_h U])\right| \leq C$

$\mu = \frac{1}{\Delta t} = \frac{N_T}{T} = 100$ (dans le cas de la simulation)

donc pour $\mu$ qui dépend de $h$ mais pas de $M$, la matrice correspondant à $\mu \cdot I_d + A$ est une M-matrice (càd une matrice de la forme $s \cdot I_d - B$ où $B \geq 0_d$ et $s > \rho(B)$ avec $\rho$ le rayon spectrale de $B$)

et donc (par un th. pour les M-matrices) $\mu \cdot I_d + A$ est inversible

donc le système d'équations linéaire dans l'équation de KFP discrète admet une une unique solution $M$ étant donné $U$.

$\square$

## 9 Positivity of $M_n$ for all $n$

Montrons par récurrence la propriété $\mathcal{P}_n$ : "$\forall n$, $M^n$ est positive"

$\mathcal{P}_0$ : $M^0$ est positive (par hypothèse) ok

Supposons $\mathcal{P}_n$, Montrons $\mathcal{P}_{n+1}$

comme $\mu > 0$ et $L = \mu \cdot I_d + A$ est inversible d'après la question précédente

et $-A \geqslant O_d$

$L^{-1}$ est positive car c'est l'inverse d'une matrice avec des coefficients non diagonaux négatifs (par def d'une M-matrice)

et $M^n$ est positive d'après l'hypothèse de récurrence

donc en isolant $M^{n+1}$, on conclus que $M^{n+1}$ est positive

donc $\mathcal{P}_{n+1}$ est vraie

donc si $M^0$ est positive, $\forall n$, $M^n$ est positive.

7

## References

[1] Y. Achdou and I. Capuzzo-Dolcetta, Mean field games: numerical methods, *SIAM J. Numer. Anal.*, 48 (2010), pp. 1136–1162.

[2] Y. Achdou and J.-M. Lasry, Mean field games for modeling crowd motion, in *Contributions to Partial Differential Equations and Applications*, Springer International Publishing, 2019, ch. 4, pp. 17–42.

[3] J.-M. Lasry and P.-L. Lions, Jeux à champ moyen. I. Le cas stationnaire, *C. R. Math. Acad. Sci. Paris*, 343 (2006), pp. 619–625.

[4] J.-M. Lasry and P.-L. Lions, Jeux à champ moyen. II. Horizon fini et contrôle optimal, *C. R. Math. Acad. Sci. Paris*, 343 (2006), pp. 679–684.

[5] J.-M. Lasry and P.-L. Lions, Mean field games, *Jpn. J. Math.*, 2 (2007), pp. 229–260.