POLYTECHNIC OF TURIN

Politecnico di Torino

Master's degree programme in Mathematical Engineering

Consumption-Saving under Labor Uncertainty

Supervisor

Prof. PAOLO BRANDIMARTE

Candidate

FULVIO RADDI

March 2024

Summary

This thesis, authored by Fulvio Raddi from the Polytechnic of Turin, delves into the numerical dynamic programming approach for addressing consumption and saving decisions under labor income uncertainty. The central objective is to ascertain optimal strategies for consumption and savings in the presence of two stochastic factors: varying labor conditions (modeled as unemployed, fully employed, and an intermediate state) and portfolio investment in both a riskless and a risky asset, with observed returns post-investment. Transitions between labor conditions are captured through a Markov chain. Utilizing Matlab scripts, the thesis employs a numerical dynamic programming framework, emphasizing scenario generation, value function approximation, and state space discretization. The overarching aim is to pinpoint the optimal strategy that maximizes a utility function linked to consumption, considering the dynamic interplay of wealth and employment conditions within the numerical dynamic programming paradigm.

Acknowledgements

I would like to express my deepest gratitude to Prof. Paolo Brandimarte, my thesis advisor, for his invaluable guidance, unwavering support, and insightful feedback throughout the entire research process. His expertise and encouragement played a pivotal role in shaping this work.
I would like to extend my heartfelt gratitude to the European Central Bank for providing me with the opportunity to work within its esteemed institution during the period of writing this thesis. The exposure to the dynamic environment and the invaluable experiences gained have significantly enriched me, contributing to my personal and professional growth in every aspect of my human being. I am truly appreciative of the support and resources extended to me during this period.

Gratitude is extended to my family and friends for their unwavering support, understanding, and encouragement during the challenging phases of this academic journey. Their belief in my abilities has been a constant source of motivation.

Finally, I want to express my appreciation to the entire Polytechnic of Turin community for providing a conducive environment for academic growth and research exploration.

Fulvio Raddi

"Pauca sed matura"
Gauss's motto

Table of Contents

List of Tables

List of Figures

Acronyms

AI

artificial intelligence

Chapter 1

Investing for Retirement

1.1 Introduction to the problem

In this chapter, we will introduce a simplified model for making consumption and saving decisions.[1] We have T time instants and we have to decide the allocation of wealth W_t between consumptions C_t and savings S_t for each $t = 0, \dots, T-1$. The amount S_t must be divided between a risky and a risk-free asset. In this model, the time horizon T is fixed; however, in more realistic models, T is treated as a random variable.
Another source of uncertainty is the labour income L_t. To model it, we assume that L_t can take one of three values based on the employment state. Let λ_t be the state of the employment at time t, which take three values in the set $\mathcal{L} = \{\alpha, \beta, \gamma\}$. If we interpret α as "fully-employed", η as "unemployed" and β as an intermediate condition then we assume that the labour income $L(\cdot)$ is a function of the state with $L(\alpha) > L(\eta) > L(\beta)$. To maintain temporal coherence, the dynamics of the employment state can be model as a finite Markov chain, where the matrix $\mathbf{\Pi}$ is formed by the time-indipendent transition probabilities

$$\pi_{i,j} = \mathbb{P}\{\lambda_{t+1} = j | \lambda_t = i\}, \qquad i, j \in \mathcal{L}.$$

The decision process unfolds as follows:

- At a time instant $t = 0,1,2,\dots,T-1$, i.e., at the beggining of each time interval, the agent owns a current financial wealth denoted by W_t, resulting from the previous saving and investment decisions.

[1] For the interested reader see [1] chapters 3 and 6, for a complete treatment see [2]

- Labor income L_t is collected; this is the income earned during the previous time interval t, from time instant $t-1$ to time instant t.
- The total available wealth is $W_t + L_t$, the sum of financial wealth and the last earned labor income; this is devided between saving S_t and consumption C_t.
- Consumption yields a utility represented by a concave function $u(\cdot)$. The overall objective is to maximize the total expected utility

$$\max \mathbb{E}\left[\sum_{t=0}^{T-1} \gamma^t u(C_t) + \gamma^T H(W_T + L_T)\right],$$

where $\gamma \in (0,1)$ is a subjective discount factor. The function $H(\cdot)$ might represent utility from bequest or serve as a means to ensure the acceptability of the terminal wealth. For simplicity, we assume that terminal wealth at time $t = T$ is entirely consumed, i.e., $C_T = W_T + L_T$. Therefore, our focus is on the objective function

$$\max \mathbb{E}\left[\sum_{t=0}^{T} \gamma^t u(C_t)\right].$$

The saved amount S_t, $t = 0, \ldots, T-1$, is allocated between a risk-free asset, with deterministic rate of return r_f for each time interval, and a risky asset, with random rate of return R_{t+1}. As is customary, the subscript $t+1$ emphasizes the fact that this return will only be known at the subsequent time instant $t+1$, or, in other words, at the end of the next time interval $t+1$, whereas the allocation decision must be made now, at the beginning of the time interval, i.e., at time instant t. Let us denote the fraction of saving that is allocated to the risky asset by $\alpha_t \in [0,1]$. Given a decision α_t , the rate of return of the investment portfolio over the next time interval is

$$\alpha_t R_{t+1} + (1-\alpha_t) r_f = \alpha_t (R_{t+1} - r_f) + r_f,$$

where $R_{t+1} - r_f$ is typically referred to as excess return. Therefore, the financial wealth at time instant $t+1$, before collecting income L_{t+1}, is

$$W_{t+1} = [1 + \alpha_t (R_{t+1} - r_f) + r_f].$$

Let's assume that the initial state λ_0 is known, and the corresponding income $L_0 = L(\lambda_0)$ is immediately available at time $t = 0$ for consumption and saving, along with the initial financial wealth W_0. Therefore, the available wealth at time $t = 0$ is $W_0 + L_0$. More generally, at time instant t, we observe the employment state λ_t and collect labor income $L_t = L(\lambda_t)$, which is added to financial wealth W_t to yield the available wealth $W_t + L_t$. The current employment state λ_t does

not specify the labor income $L_{t+1} = L(\lambda_{t+1})$ over the next time interval, but it provides a clue about it, since we have some information about the conditional probability distribution of the next state λ_{t+1}, via the transition probability $\mathbf{\Pi}$. We need to define our state variables, to address the problem with a dynamic programming framework. In our case the most natural choice is the pair

$$s_t = (W_t, \lambda_t),$$

where W_t is the financial wealth at time instant t and λ_t the current employment state; It's worth noting once more that the total available (cash) wealth at time instant t is $W_t + L(\lambda_t)$. We observe that the state space is continuous with respect to the first component and discrete with respect to the second one. Dynamic programming requires to find the set of value functions

$$V_t(W_t, \lambda_t), \qquad t = 1,2,\ldots,T-1,$$

subject to the terminal condition

$$V_T(W_T, \lambda_T) = u(W_T + L(\lambda_T)).$$

The functional equation is

$$V_t(W_t, \lambda_t) = \max_{C_t, \alpha_t} \left\{ u(C_t) + \gamma \mathbb{E}_t \Big[V_{t+1}(W_{t+1}, \lambda_{t+1}) \Big] \right\}, \tag{1.1}$$

where the notation $\mathbb{E}_t[\cdot]$ emphasizes that expectation is conditional on the currentstate and decisions. The dynamic equations for state transitions are

$$\lambda_{t+1} = M_\Pi(\lambda_t), \tag{1.2}$$

$$W_{t+1} = (W_t + L_t - C_t)\Big[1 + r_f + \alpha(R_{t+1} - r_f)\Big], \tag{1.3}$$

where $L_t = L(\lambda_t)$, and M_Π in Eq. (1.2) represents the stochastic evolution of the employment state according to the matrix $\mathbf{\Pi}$, and the evolution of wealth in Eq. (1.3) depends on the random rate of return of the risky asset and the labor income L_t, which is a function of employment state λ_t. The constraints on the decision variables are

$$\begin{aligned} &\alpha_t \in [0,1], \qquad t = 0,\ldots,T-1 \\ &0 \le C_t \le W_t + L_t, \qquad t = 0,\ldots,T-1. \end{aligned}$$

We are tacitly assuming that the employment state evolution and the return from the risky asset are independent. If we consider that both financial returns and employment depend on underlying macroeconomic factors, we should factor in their correlation.

1.2 Approximating the Optimal Policy by Numerical DP

In this section we extend the approach of [1] (see chapter 6) with refinements and generalization when it's possible.

1.2.1 State space discretization

We have to discretize the wealth component of our state variable. In [1], the author employs a uniform grid to discretize the state space. Typically, this approach may not be the most optimal choice. One alternative idea is to utilize the Tchebycheff nodes. By Tchebycheff nodes, we refer to the numbers

$$x_j = \cos\frac{\pi j}{N}, \qquad 0 \leq j \leq N \tag{1.4}$$

for some $N \geq 0$, where for $N = 0$ we take $x_0 = 1$. The reason why we use such nodes can be expressed by the following theorem [2]

Theorem 1 *Let f be a continuous function on $[-1,1]$, p_N its degree N polynomial interpolant in the Tchebysheff points (1.4), and p_N^* its best approximation of on $[-1,1]$ in the norm $||\cdot|| = ||\cdot||_\infty$. Then*

1. *$||f - p_N||$;*
2. *if f has a kth derivative in $[-1,1]$ of bounded variation for some $k > 1$, $||f - p_N|| = O(N^{-k})$ as $N \to \infty$;*
3. *if f is analytic in a neighborhood of $[-1,1]$, $||f - p_N|| = O(C^N)$ as $N \to \infty$ for some $C < 1$; in particular we may take $C = \frac{1}{M+m}$ if f is analytical in the closed ellipse with foci ± 1 and semimajor and semiminor axis lengths $M > 1$ and $m \geq 0$.*

It follows from condition (1) of Theorem 1 that the Tchebycheff interpolant of a function f on $[-1,1]$ is within a factor 10 of the best approximation if $N < 10^5$, a factor 100 if $N < 10^{66}$. Thus Tchebycheff interpolants are *near-best*.
With this result in mind, we are going to test whether a grid based on Tchebycheff nodes prevents undesirable effects on the value functions. To achieve this, we will employ a Matlab script provided by [1] (see chapter 6) where we consider a Tchebycheff grid instead of a uniform one.

[2] See [3] for further details

```
%% set up discretization for DP: WealthMax 200, numPoints 30
Wmin = 1; Wmax = 200; numPoints = 30;
% wealthValues = linspace(Wmin,Wmax,numPoints);
chebychevgrid=@(Wmin,Wmax,n)Wmin+((Wmax-Wmin)/2)*(cos(linspace(pi,0,numPoints))+1);
wealthValues = chebychevgrid(Wmin,Wmax,numPoints);
wealthGrid = repmat({wealthValues(:)}, timeHorizon, 1);
% Learn DP policy (cubic)
splineList_1 = FindDPPolicy(utilFun, gamma, wealthGrid, incomeValues, ...
    transMatrix, retScenarios, retProbs, riskFree);
figure(1); PlotDPSplines(splineList_1, gridW, timeList);
```

Figure 1.1: Part of the code where the Tchebycheff grid is implemented

1.2.2 Sparse Grid discretization

Another option is to use a sparse grid discretization. The sparse grid method is a general numerical discretization technique for multivariate problems. This approach, first introduced by the Russian mathematician Smolyak in 1963, is based on a sparse tensor product construction.
We are not delving into the technicalities of this topic, but interested readers can follow this guide [4]. In Matlab, we can experiment with approaches to generate sparse grids. The initial option is to commence with a uniform grid and introduce a certain degree of sparsity using the built-in function *datasample*.

```
numSparsePoints = 20;  % Adjust this based on your desired sparsity
%% set up discretization for DP: WealthMax 200, numPoints 30
Wmin = 1; Wmax = 500; numPoints = 30;
wealthValues = linspace(Wmin, Wmax, numPoints);
sparseGridIndices = datasample(1:numPoints, numSparsePoints, 'Replace', false);
wealthValues = wealthValues(sparseGridIndices);
wealthGrid = repmat({wealthValues(:)}, timeHorizon, 1);
% Learn DP policy (cubic)
splineList_1 = FindDPPolicy(utilFun, gamma, wealthGrid, incomeValues, ...
    transMatrix, retScenarios, retProbs, riskFree);
figure(1); PlotDPSplines(splineList_1, gridW, timeList);
```

Figure 1.2: Implementing the Sparce grid using the datasample function

We can see from Figure 1.3 that we set 30 points, of which 20 are coming from the datasample function. With this selection, the resulting graph is as follows:

For each time period, we observe a convex and increasing value function that aligns with the problem setting.

The second way is to generate sparse grids with basis functions. Since the function we want to approximate is concave and increasing, we can choose basis

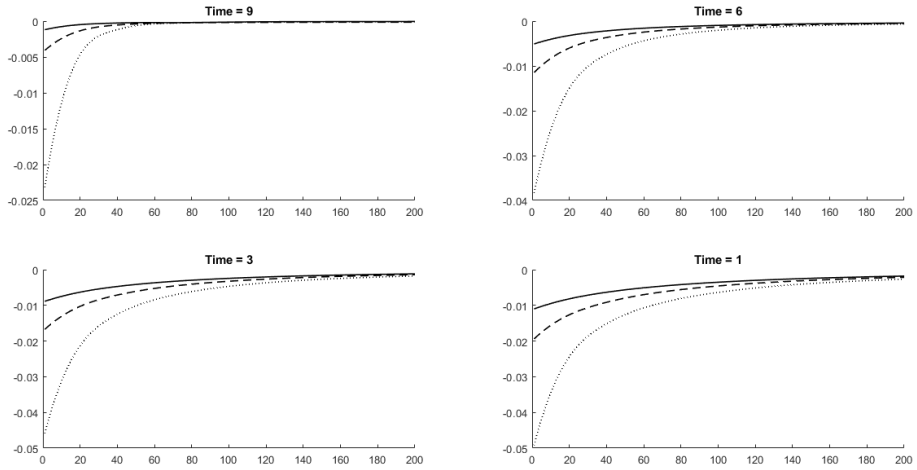

Figure 1.3: Implementing the Sparce grid using the datasample function

functions that capture these characteristics. Here are some considerations for selecting basis functions for a sparse grid in this context:

- **Basis Functions:**

  For concave and increasing functions, we might want basis functions that can represent such behavior. Polynomial basis functions like Tchebysheff polynomials or Legendre polynomials could be suitable. We could consider basis functions that are known to capture concavity, such as Chebyshev polynomials of the first kind or shifted Legendre polynomials.

- **Adaptive Refinement:**

  Since the function is concave and increasing, areas of interest are likely to be regions with high gradients. We can use adaptive refinement strategies to focus on regions where the function exhibits rapid changes. This can be achieved by adding new points in regions with high gradients.

- **Sparse Grid Construction:**

  We exploit the tensor product structure of sparse grids. If our function was defined in a multi-dimensional space, we could have used tensor products of one-dimensional basis functions. This allows us to construct high-dimensional basis functions efficiently.

- **Consider Constraints:**

If there are known constraints on the function (e.g., bounded intervals, concavity constraints), we can incorporate these constraints into the basis functions or the refinement strategy.

We are going to use $f(x) = e^{-\alpha x}$ where $\alpha > 0$ as a basis function. This function is concave and increasing, so let's see what will happen during our experiments. The Matlab code to perform it is described in Fig. 1.4.

```
%% Grid Generation with Concave and Increasing Basis Functions
function gridValues = concaveIncreasingBasisGrid(Wmin, Wmax, numPoints)
    % Generate concave and increasing basis nodes
    alpha = 0.1;
    k = 1:numPoints;
    basisNodes = -exp(-alpha* k);

    % Map basis nodes to the interval [Wmin, Wmax]
    gridValues = (Wmax - Wmin) * (basisNodes - min(basisNodes)) / (max(basisNodes) - min(basisNodes)) + Wmin;
end
```

Figure 1.4: Matlab code for the concave and increasing basis function

The graphs obtained by this choice are summarize in Fig. 1.5

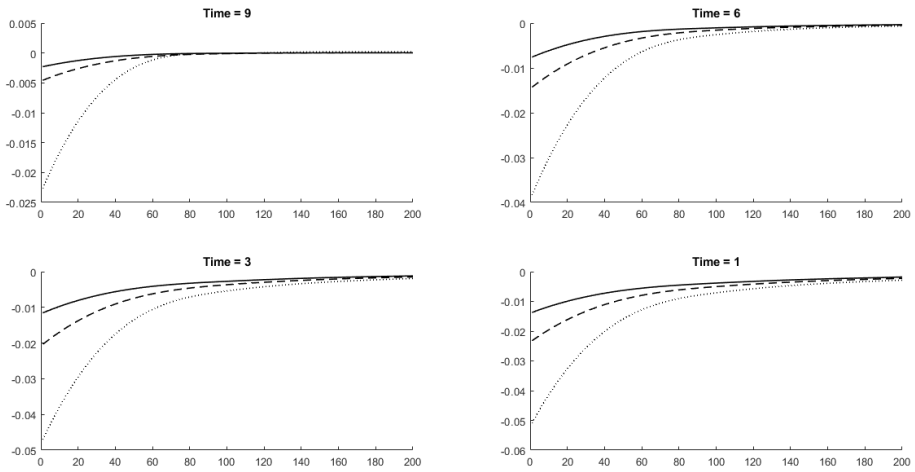

Figure 1.5: Value functions using sparse grid with increasing concave basis function

As expected, we have obtained concave and increasing value functions.

In the upcoming chapters, we will explore all the components involved in the Matlab script. For now, it's worth mentioning that we are approximating the

value function with cubic splines. In the following image, the value functions are displayed for each employment condition. The lowest dotted line corresponds to the unemployed state, and the highest continuous line corresponds to the fully employed state.

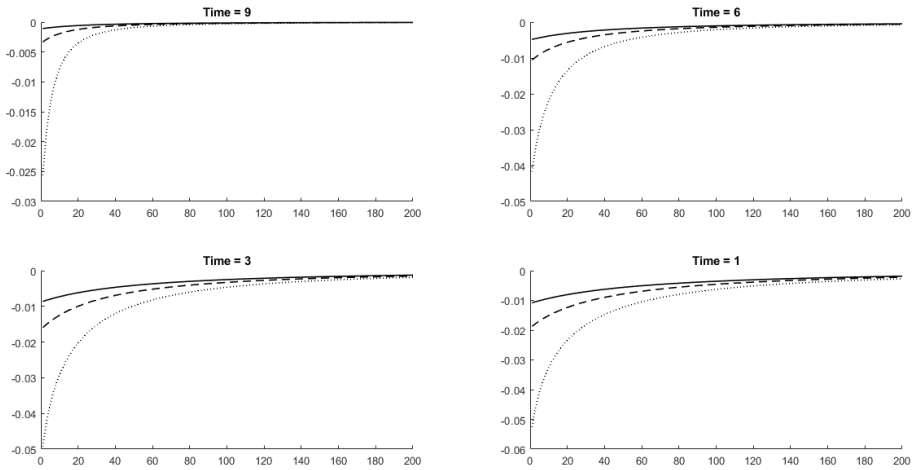

Figure 1.6: Value functions for the three labor conditions

We can observe that they are concave, even for large values of wealth, which is coherent with risk aversion.

1.2.3 Approximation architectures for the value functions

For now, we have used interpolating cubic splines in our experiments. A possible alternative is to use Piecewise Cubic Hermite Interpolation (PCHIP); the reason is that this method preserves monotonicity and the shape of the data. On the other hand, cubic splines aim to reduce an oscillatory behavior; it makes sense to compare the two approach and to see wich one helds to better results. Matlab provides a function called *pchip* that generates the shape-preserving Hermite splines. If we run our script using the Hermite splines we have the Fig 1.7

As we can see, it's worst than the cubic splines. If we focus on the periods 1 and 3, we observe that, for large values of the wealth, the value functions are convex. This is opposite to the risk-aversion like we pointed out in the last section.
Another test that can be performed is to use the shape-preserving splines with an uniform grid. In this case, the results are shown in Fig.1.9.

We conducted this last experiment to emphasize the importance of the grid choice in approximating the value functions. As depicted from Fig. 1.7, all the values functions are concave and so, in this case, the Hermite splines perorm better

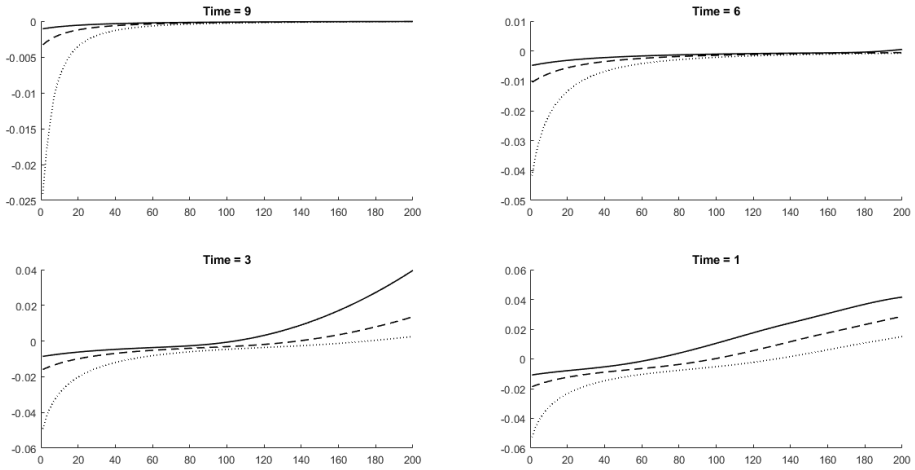

Figure 1.7: Value functions with shape-preserving splines

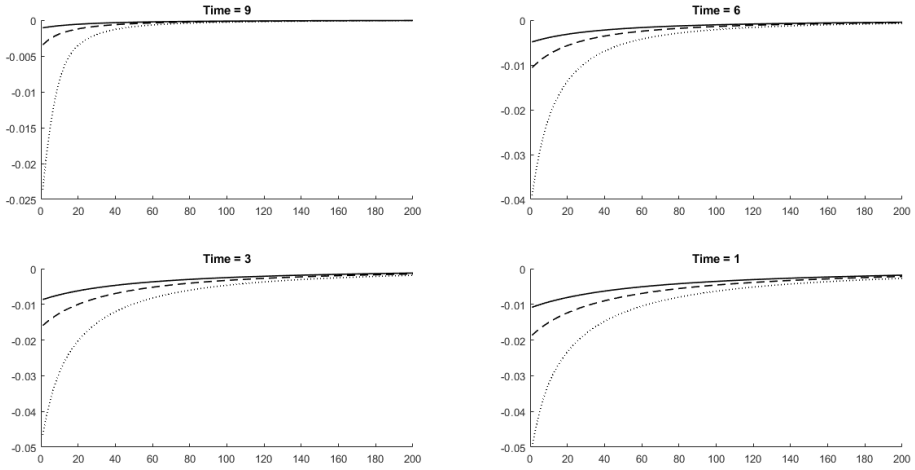

Figure 1.8: Value functions with shape-preserving splines and uniform grid

than the cubic ones. We are not going to consider the uniform grid as a serious candidate for our model, but it can be used as a benchmark to underline the sensitivity of the model for different choices.
Another strategy is to use sentinel points [3]. For istance, we can think to add few more points corresponding to large and unlikely wealth levels and acting

[3] see[1] section 6.2.3

as "sentinels". If we add the points 300 and 500 to the grid, we obtain a good approximation both for cubic and shape-preserving splines as we can see in Figure 1.9 and Figure 1.10.

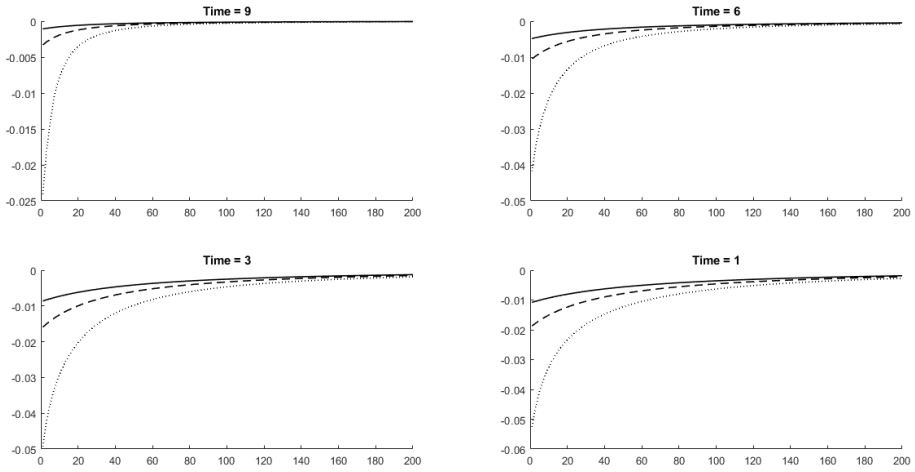

Figure 1.9: Value functions with shape-preserving splines and sentinels point

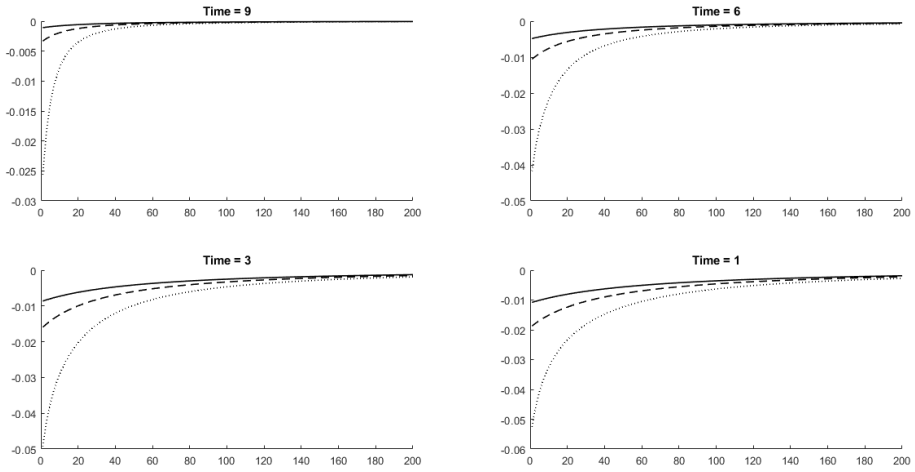

Figure 1.10: Value functions with cubic splines and sentinels point

It's worth stressing again how crucial the choice of the grid is in approximating the value functions. Simply adding two extra points to our grid proved sufficient to eliminate non-concave behaviors. Moving forward, we will consistently employ the Tchebysheff grid and cubic splines, unless otherwise specified.

1.3 Scenario Generation

Let's delve into how we generated the scenario in the previous Matlab script. Given that the employment state follows a discrete-time Markov chain, our task is to discretize the risk factor associated with the price (or return) of the risky asset. The random return from the risky asset is connected to the relative change in its price P_t :

$$R_{t+1} = \frac{P_{t+1} - P_t}{P_t} \tag{1.5}$$

If we assume that the price of the risky asset is modeled by geometric Brownian motion, it can be shown that its evolution is given by

$$P_{t+1} = P_t \exp\left[\left(\mu - \frac{\sigma^2}{2}\right) + \frac{\sigma}{\epsilon}\right],$$

where the length of the discrete time intervals in the model and the time unit used in expressing the drift and volatility coefficients μ and σ are the same (say, one year). The underlying risk factor is a standard normal variable $\epsilon \sim \mathcal{N}(0,1)$. This means that prices P_t are lognormal random variables, i.e., exponentials of a normal random variable with expected value $\mu - \frac{\sigma^2}{2}$ and standard deviation σ. Since we have a single risk factor, there is no need to resort to random sampling. Nevertheless, there are different options to come up with a clever discretization of the underlying normal random variable or of the lognormal price. In the script we have pursued a stratification approach to discretize a lognormal random variable Y with parameters μ and σ. The idea is based on partitioning the support of the lognormal distribution, the halfline $[0, +\infty]$, into m intervals with the same probability, and then choose, as a representative point of each interval, the conditional expected value on that interval. The procedure is the following:

1. Define $m - 1$ probability levels

$$p_h = \frac{h}{m}, \qquad h = 1, \ldots, m$$

2. Find the corresponding quantiles y_h by inverting the cumulative distribution function of the lognormal distribution.

3. Use the quantiles y_h to define m intervals

$$[0, y_1), [y_1, y_2), \ldots, [y_{m-1}, +\infty]$$

By construction, there is a probability $q_h = \frac{1}{m}$ that Y falls into each interval.

4. We want to use the expected value ξ_h of Y , conditional on falling into the interval $(y_{h-1}, y_h$, as the representative point of the interval:

$$\xi_h \doteq \mathbb{E}\Big[Y|Y \in (y_{h-1}, y_h)\Big].$$

This defines a discrete distribution, with a support consisting of m points, approximating the lognormal variable Y . In order to find the conditional expected value, we use standard numerical integration:

$$\xi_h = m \int_{y_{h-1}}^{y_h} x f_Y(x) dx.$$

The conditional expected value is obtained by integrating the probability density $f_Y(\cdot)$ over the interval (y_{h-1}, y_h) and dividing by the corresponding probability. Since the probability of observing a realization of Y in that interval is $1/m$, we end up multiplying by m. In particular, for $h = 1$, we integrate from $y_0 = 0$ to y_1. For $h = m$, we should integrate from y_{m-1} to $y_m = +\infty$; we replace $+\infty$ by a suitably large value,

$$y_m = e^{\mu+20\sigma},$$

which is based on the well-known fact that, for the underlying normal variable, values beyond $\mu + 3\sigma$ are not too likely.
As suggested in [1] this is not the state-of-the-art but, whatever approach we adopt, we will define scenarios characterized by realizations y_h of the lognormal variable Y with parameters μ and σ. We can rewrite (1.5) as

$$R_{t+1} = \frac{P_{t+1} - P_t}{P_t} = \exp\left[\left(\mu - \frac{\sigma^2}{2}\right) + \sigma\epsilon\right] - 1 = Y - 1 \tag{1.6}$$

Hence, we define a set of return scenarios

$$R_h = y_h - 1, \qquad h = 1, \ldots, m$$

When using the above stratification approach, each scenario has probability $q_h = 1/m \quad h = 1, \ldots, m$; these probabilities are uniform, but this need not be the case in general.
Let's try another strategy. Let's consider the problem of computing the expectation of a real-valued function f of a random variable $\tilde{X}$ with probability density function $p(\cdot)$:

$$\mathbb{E}\Big[f(\tilde{X})\Big] = \int f(x)p(x)dx.$$

Gaussian quadrature is a numerical integration technique that calls for $\tilde{X}$ to be replaced with a discrete random variable whose distribution matches that of $\tilde{X}$ as

closely as possible. Specifically, in a Gaussian quadrature scheme, the mass points $x_1, x_2, \ldots, x_n$ and probabilities $p_1, \ldots, p_n$ of the discrete approximant are chosen in such a way that the approximant possesses the same moments of order $2n-1$ and below as $\tilde{X}$:

$$\sum_{i=1}^{n} p_i x_i^k = E(\tilde{X}^k) \qquad k = 0, \ldots, 2n-1.$$

Given the mass point and probabilities of the descrete approximant, the expectation of $f(\tilde{X})$ is approximated as follows:

$$\mathbb{E}\Big[f(\tilde{X})\Big] = \int f(x)p(x)dx \approx \sum_{i=1}^{n} f(x_i)p_i.$$

Computing the n-degree Gaussian mass points x_i and probabilities p_i for a random variable involves solving $2n$ nonlinear equations in as many unknowns. Efficient, specialized numerical routines for computing Gaussian mass points and probabilities are available for virtually every major distribution, including the normal, uniform, gamma, exponential, Chi-square, and beta distributions. In our applications, we will be exclusively concerned with computing expectations of functions of normal random variates and related random variates.

By design, an $n-$point Gaussian quadrature rule will compute the expectation of $f(\tilde{X})$ exactly if f is a polynomial of order $2n-1$ or less. Thus, if f can be closely approximated by a polynomial, the Gaussian quadrature rule should provide an accurate estimate of the expectations. Gaussian quadrature rules are consistent for Riemann integrable functions. That is, if f is Riemann integrable, then the approximation offered by Gaussian quadrature can be made arbitrarily precise simply by increasing the number of mass points in the discretizing distribution. Gaussian quadrature is the numerical integration method of choice when the integrand is bounded and possesses continuous derivatives, but should be applied with great caution otherwise. If the integrand is unbounded, it is often possible to transform the integration problem into an equivalent one with bounded integrand. If the function possesses known kink points, it is often possible to break the integral into the sum of two integrals of smooth functions. If these or similar steps do not produce smooth, bounded integrands, then Newton-Cotes quadrature methods may be more accurate than Gaussian quadrature because they contain the error caused by the kinks and singularities to the interval in they occur.

The Gaussian quadrature scheme for normal variates may be used to develop a good scheme for lognormal random variates. By definition, Y is log-normally distributed with parameters μ and σ if, and only if, it is distributed as $\exp \tilde{X}$ were $\tilde{X}$ is normally distributed with mean μ and standard deviation σ. It follows that if $\{(x_i, p_i\}$ are Gaussian mass points and probabilities for a normal distribution, then $\{(y_i, p_i\}$, where $y_i = \exp x_i$, provides a reasonable a discrete approximant

for a log-normal distribution. Given this discrete approximant for the log-normal distribution, one can estimate the expectation of a function of $\tilde{Y}$ follows:

$$\mathbb{E}\Big[f(\tilde{Y})\Big] = \int f(y)p(y)dy \approx \sum_{i=1}^{n} f(y_i)p_i.$$

This integration rule for log-normal distributions will be exact if f is a polynomial of degree $2n-1$.
In our case, we can create a Matlab script that creates scenarios with a Gaussian quadrature. Since Matlab does not provide a built-in function for Gaussian quadrature, we can develop one. The idea is to use three functions.

hermite gauss: This function computes nodes **x** and weights **w** for Gauss-Hermite quadrature. Gauss-Hermite quadrature is a numerical method for approximating integrals of functions weighted by the standard normal distribution.

```
function [x, w] = hermite_gauss(N)
    % Compute nodes x and weights w for Gauss-Hermite quadrature
    [V, D] = eig(diag(0:N-1) + diag(sqrt(1:N-1), 1) + diag(sqrt(1:N-1), -1));
    x = diag(D);
    w = (pi^(-0.5) * V(1, :)').^2;
end
```

Figure 1.11: hermite gauss function

- The matrix V represents the matrix of eigenvectors obtained from the diagonalization of the tridiagonal matrix A (in the code A is the argument of the function eig());
- The eig function in MATLAB returns two matrices, V and D, where D is a diagonal matrix containing the eigenvalues, and V is a matrix whose columns are the corresponding eigenvectors; The eigenvalues on the diagonal are related to the zeros of Hermite polynomials.

In the context of Gauss-Hermite quadrature, V is used to obtain the weights associated with the zeros of Hermite polynomials. The first column of V is extracted and normalized, and then squared to obtain the weights.

gaussHermite: The **gaussHermite** function is a wrapper function in the provided MATLAB script that calls the more specific **hermite_gauss**
It's used for simplicity and readability, allowing to use gaussHermite in the main script without exposing the details of the specific Hermite quadrature implementation. When we call gaussHermite(N), it internally invokes hermite_gauss(N) and returns the resulting nodes and weights.

```
function [x, w] = hermite_gauss(N)
    % Compute nodes x and weights w for Gauss-Hermite quadrature
    [V, D] = eig(diag(0:N-1) + diag(sqrt(1:N-1), 1) + diag(sqrt(1:N-1), -1));
    x = diag(D);
    w = (pi^(-0.5) * V(1, :)').^2;
end
```

Figure 1.12: hermite gauss function

MakeScenarioQ The function calls gaussHermite(numScenarios) to obtain nodes and weights for Gauss-Hermite quadrature. These nodes and weights are associated with the standard normal distribution.

```
function [values, probs] = MakeScenariosQ(numScenarios, mu, sigma)
    % Initialize output
    values = zeros(numScenarios, 1);

    % Find extreme points of subintervals using Gauss-Hermite nodes
    [nodi, pesi] = gaussHermite(numScenarios);
    y = exp(mu + sqrt(2) * sigma * nodi);

    f = @(x) x .* lognpdf(x, mu, sigma);

    % Find expected values using Gaussian quadrature
    for h = 1:numScenarios
        values(h) = sum(pesi .* f(y)) / numScenarios;
    end
end
```

Figure 1.13: Gaussian scenarios

The nodes obtained from Gauss-Hermite quadrature are transformed using $y = \exp(\mu + \sqrt{2} \times \sigma \times nodes)$. This transformation is necessary because the original quadrature nodes are associated with the standard normal distribution, and this line adjusts them to match a lognormal distribution with parameters μ and σ.
The normalization step in the MakeScenariosQ function is performed to ensure that the computed expected values (values) are properly scaled and represent the average over the specified number of scenarios (numScenarios). It is important because without it, the values vector would represent the sum of expected values, not the average. By dividing each element by numScenarios, we obtain the average expected value for each scenario, making the results more interpretable, especially when comparing across different numbers of scenarios or simulations.
With this quadrature we obtain the following approximated value functions:

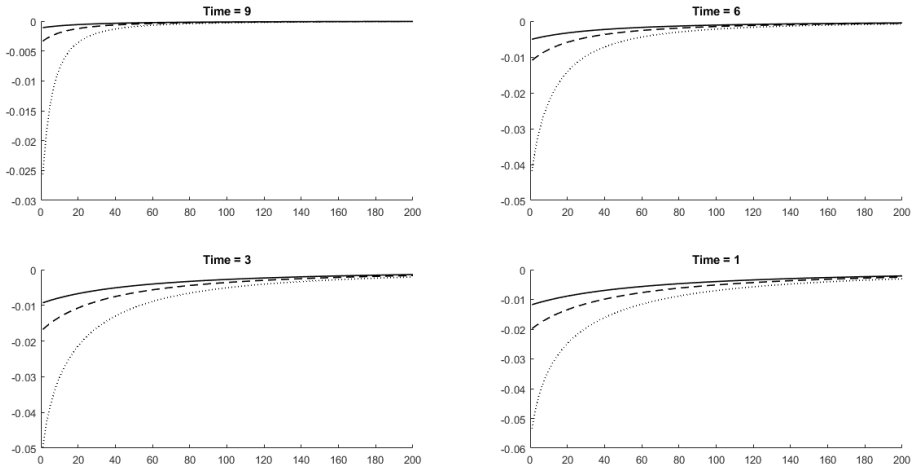

Figure 1.14: Gaussian scenarios

Chapter 2

Selecting the best approximation

In the previous chapter, our emphasis was on exploring various approaches to solving the problem. In this chapter, we will delve into how to compare these approaches. As we are tackling an optimization problem, the optimal choice would be the one that maximizes the utility function within a reasonable timeframe. To validate this, we will employ different utility functions to assess the robustness of our choice.

2.1 Implementation in Matlab

To compare different approaches we will use modified scripts from [1][1]. The two most important Matlab functions we are going to use are `FindDPPolicy` and `RunDPPolicy`. The first one takes the following input arguments:

- the utility function and the discount factor γ;
- the cell array *WealthGrid*, which contains an array of wealth values for each time instant; the cell array allows for different grids over time;
- the vector incomeValues, containing the labor income for each state, and the state transition matrix transMatrix;
- vectors *retScenarios* and *retProbs*, containing the return scenarios for the risky asset; the risk-free return is stored in riskFree

[1] See pp.169-180

The output of the function is the cell array *splineList*, containing three splines (one for each employment state) for each time instant.
The *RunDPPolicy function* it's needed to perform the simulation. It's similar to the first one, but we need in-sample return scenarios *inRetScenarios* and out-of-sample scenarios *outRetScenarios* to check the actual performance. In-sample scenarios may be generated by deterministic stratification, whereas out-of-sample scenarios are randomly generated (hence, they have uniform probabilities). The simulation returns a vector *utilVals* containing a sample of total discounted utilities obtained over time, one entry for each sample path starting from state (W_0, λ_0), and matrices *alphaPaths, consPaths,* and *wealthPath*s that contain the full sample paths of allocation to the risky asset, consumption, and wealth, respectively.

2.2 Comparing utility functions

The first experiments we can do is to evaluate the expected utility with Tchebycheff grid in terms of the log utility function and the power utility function

$$u(x) = log(x); \qquad u(x) = \frac{x^{1-\delta}}{1-\delta}, \quad \delta < 1.$$

We are going to run the simulation with fixed values for our parameters as showed in the figure 2.1

```
% scriptSetData.m -> set example data
gamma = 0.97; timeHorizon = 10; W0 = 100;
mu = 0.07; sigma = 0.2; riskFree = 0.03;
empl0 = 2; incomeValues = [5; 20; 40];
transMatrix = [0.6, 0.3, 0.1
    0.2, 0.5, 0.3
    0.2, 0.1, 0.7];
```

Figure 2.1: Setting the parameters

First, when we compute the average utility values for the log utility and for the power utility function we get, respectively, 33.6202 and -0.0055. We are going to keep in mind these values for the next comparisons but we have to dig in a little further. Due to the ordinal nature of the utility functions, it's also important to monitor the avarage sample paths for consumption, wealth and wealth fraction allocated to the risky asset (called α). In the Fig. 2.5, 2.3 and 2.4 are shown the paths for the wealth, the consumption and the risky assets fraction.

```
%% Learn and evaluate DP policy
% set up discretizations for DP
numScenarios = 20;
% [values, retProbs] = MakeScenariosQ(numScenarios,mu,sigma,targetMean,targetStd);
[values, retProbs] = MakeScenariosQ(numScenarios,mu,sigma);
retScenarios = values - 1;
Wmin = 1; Wmax = 200; numPoints = 30;
chebychevgrid=@(Wmin,Wmax,n)Wmin+((Wmax-Wmin)/2)*(cos(linspace(pi,0,numPoints))+1);
wealthValues = chebychevgrid(Wmin,Wmax,numPoints);
% wealthValues = [linspace(Wmin,Wmax,numPoints), 300, 500];
wealthGrid = repmat({wealthValues(:)}, timeHorizon, 1);
% Learn DP policy
splineList = FindDPPolicy(utilFun, gamma, wealthGrid, incomeValues, ...
    transMatrix, retScenarios, retProbs, riskFree);
% Run DP policy
[utilValsDP, alphaPathsDP, consPathsDP, wealthPathsDP] = ...
    RunDPPolicy(splineList, W0, empl0, utilFun, gamma, ...
    incomeValues, transMatrix, retScenarios, retProbs, riskFree, ...
    retOutPaths, emplOutPaths);
aveUtilDP = mean(utilValsDP);
% Compare results
aveUtilDP
```

Figure 2.2: Learning and evaluating the DP policy

We can observe that the consumption increases in time, the wealth is accumulated and then consumed and alpha decreases in time. This is coherent with intuition and we can argue that DP approach does not contain any extra insights. To check if it is or not true, we can try to use other different techniques.
Our degree of freedom are three and they are:

- State space discretization;
- Scenario generation;
- Approximation of the value function.

Concerning the discretization of the state space, as discussed in the previous chapter, two promising candidates are sparse grids and Tchebysheff grids. We conducted various experiments altering the utility function and the type of grid, but fundamentally observed no significant differences in terms of utility function values. In the Tables 2.1 and 2.2, we can find the values of the utility function resulting from different combinations of scenarios and grids.

The Tables 2.1 and 2.2 have two rows and three columns. Each row represents the grid selection (Sparse or Tchebysheff), and each column corresponds to a scenario generation method. The "Quantile" column signifies the basic scenario generation approach, where we adopt a quantile-based stratification for the lognormal returns.

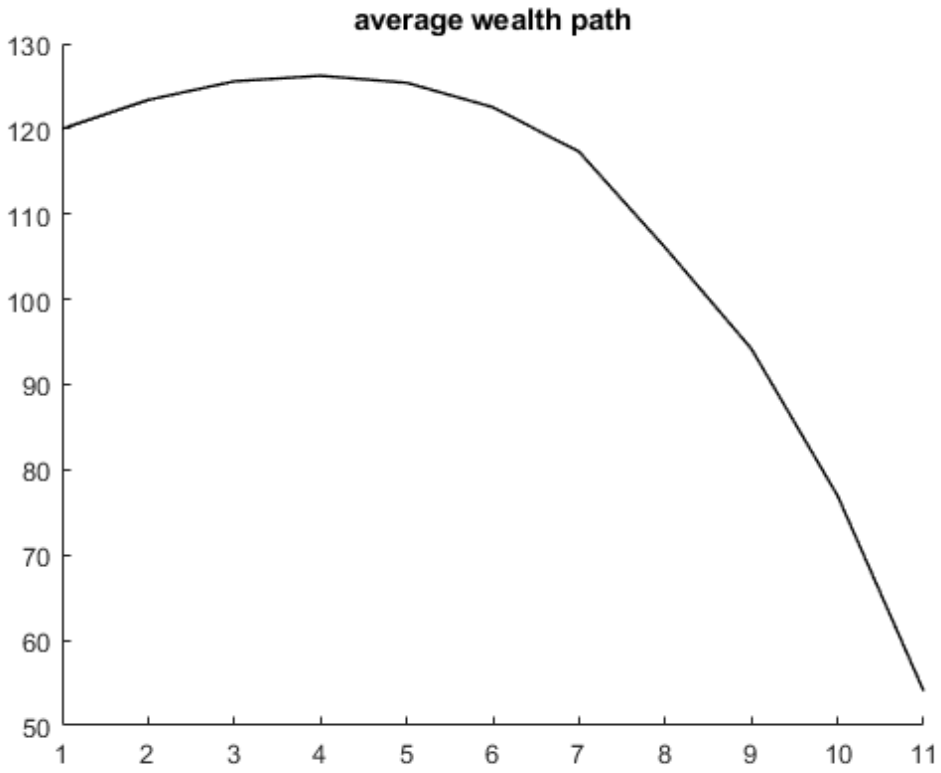

Figure 2.3: Wealth path with Tchebychev grid

Table 2.1: Log utility table broken by Grids and Scenarios

Grid	**Gauss Hermite**	**Quantile**	**Monte Carlo**
Tchebysheff Grid	29.4612	33.5292	33.6190
Sparse Grid	29.4515	33.6159	33.6159

Table 2.2: Power utility table broken by Grids and Scenarios

Grid	**Gauss Hermite**	**Quantile**	**Monte Carlo**
Tchebysheff Grid	-0.0234	-0.0055	-0.0056
Sparse Grid	-0.0055	-0.0055	-0.0056

The "Gauss Hermite" column refers to the scenario generation method used in chapter 1.An additional column named "Monte Carlo" presents results from a Monte Carlo simulation, generating 10,000 random samples from the lognormal

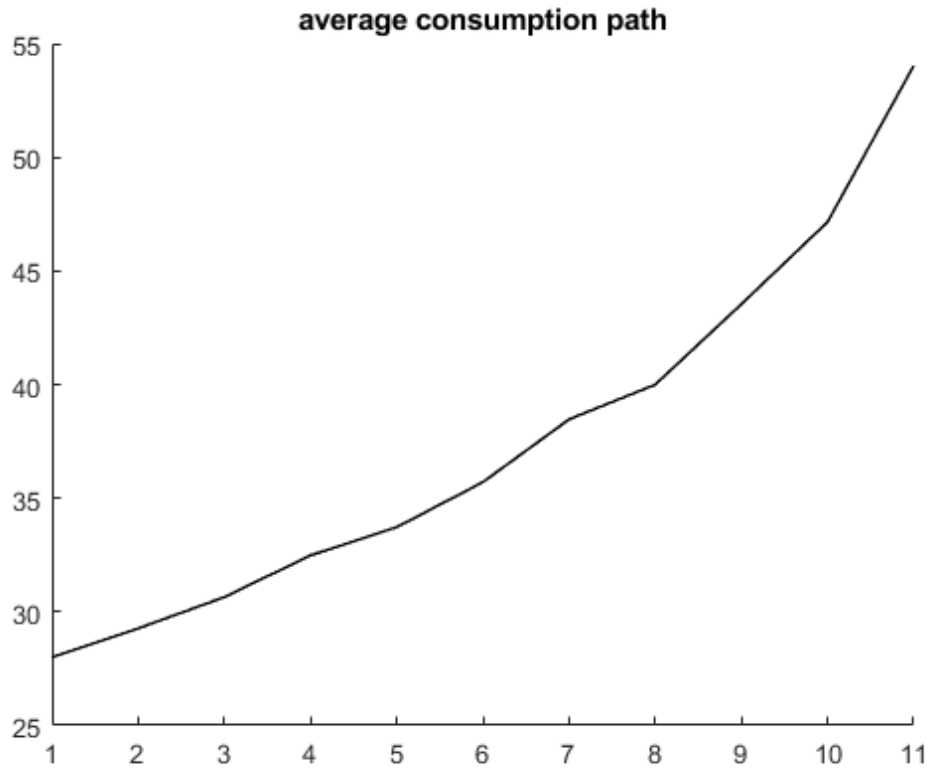

Figure 2.4: Consumption path with Tchebychev grid

distribution with a mean of 0.07 and a standard deviation of 0.2. It's crucial to compare our results with a Monte Carlo simulation, particularly when risk factors are high-dimensional, as it might be the only viable alternative.
For the logarithmic utility function, we identified that the best combination (where "best" denotes the one with the highest expected utility value) is a Monte Carlo stratification with Tchebysheff Grid. Meanwhile, for the power utility function, the optimal combination is a Tchebysheff Grid with a Gauss Hermite approach. The key question now is: which scenario generation method do we choose?
Our decision can be based on the value functions approximation and we are going to do so. Before delving into that, it's crucial to consider another fundamental variable in finance and computer science: the *time*.
When we adopt numerical approximation techniques, an important role is played by the total amount of time taken to run the algorithms. This is particularly significant in dynamic programming problems, where different stages of approximations are involved. A feasible approach to compute the total running time involves the use of Matlab functions *tic* and *toc*. It's enough to include the function *tic* before

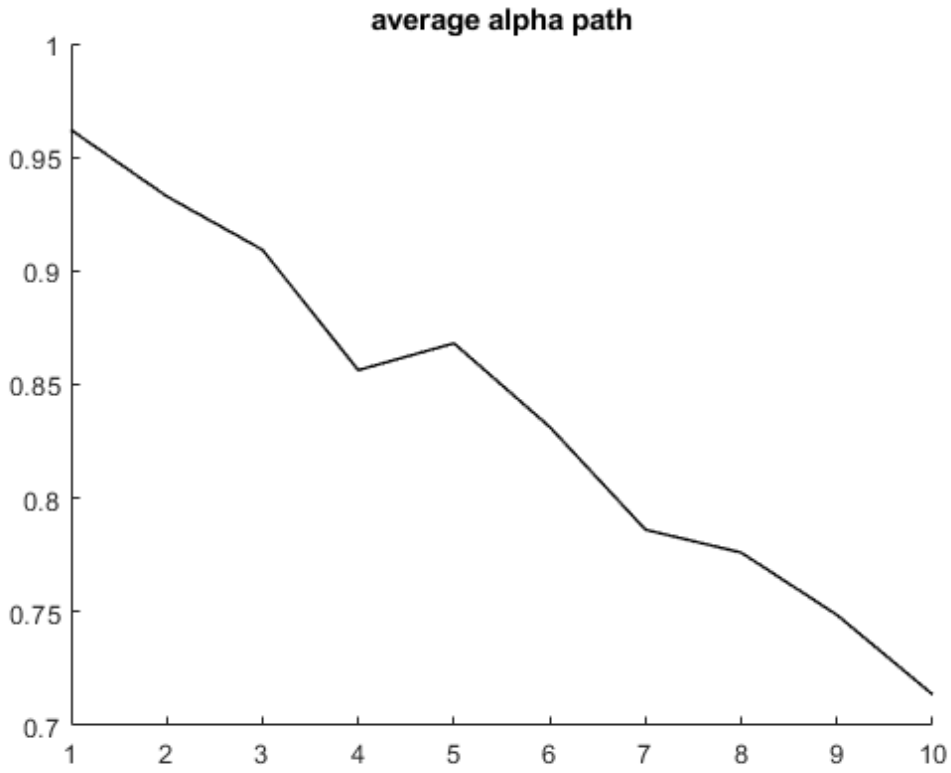

Figure 2.5: Alpha path with Tchebychev grid

the code and *toc* as the final line of the script we want to measure the elapsed time for. For instance, the running time of the combination Tchebysheff grids and Monte Carlo sampling is 305.512764 while with Gauss-Hermite is 268.822719. We could think to prefer the Gauss-Hermite one but we have to keep in mind that Monte Carlo simulation is costly (in terms of time) by default and when we face multivariable rsky factors there might be no other choice. In the next section, we are goig to consider also the value functions approximation keeping in mind the time evaluation.

2.3 Value functions approximation and utility function evaluation

In the previous chapter, we performed some experiments concerning how to approximate value functions. In particular, we focused on cubic and piecewise cubic Hermite interpolating polynomial (PCHIP) splines. Another possibility is to use

Modified Akima piecewise cubic Hermite interpolation (makima). The 'makima' cubic interpolation method was introduced in MATLAB in the R2017b release as a new option in interp1, interp2, interp3, interpn, and griddedInterpolant. In a nutshell, it represents a MATLAB-specific modification of Akima's derivative formula and has the following key properties:

- It produces undulations which find a nice middle ground between 'spline' and 'pchip';
- It is a local cubic interpolant which generalizes to 2-D grids and higher-dimensional n-D grids;
- It increases the robustness of Akima's formula in the edge case of equal side slopes;
- It eliminates a special type of overshoot arising when the data is constant for more than two consecutive nodes.

We are not going into deeper details because we are interested in how we can use it in our model. For our purpose, it is important to highlight the differences between the pchip, spline, and makima functions as we intend to use them in our experiments. Initially, we compare them using sample data that connects flat regions.

```
x = -3:3;
y = [-1 -1 -1 0 1 1 1];
xq1 = -3:.01:3;
p = pchip(x,y,xq1);
s = spline(x,y,xq1);
m = makima(x,y,xq1);
plot(x,y,'o',xq1,p,'-',xq1,s,'-.',xq1,m,'--',LineWidth=
   3)
legend('Sample Points','pchip','spline','makima','
   Location','SouthEast')
%% Figure 2.7
x = 0:15;
y = besselj(1,x);
xq2 = 0:0.01:15;
p = pchip(x,y,xq2);
s = spline(x,y,xq2);
m = makima(x,y,xq2);
```

```
16 plot(x,y,'o',xq2,p,'-',xq2,s,'-.',xq2,m,'--',LineWidth
   =3)
17 legend('Sample Points','pchip','spline','makima','
   Location','NorthEast','FontSize',16)
```

The results are shown in Fig.2.6

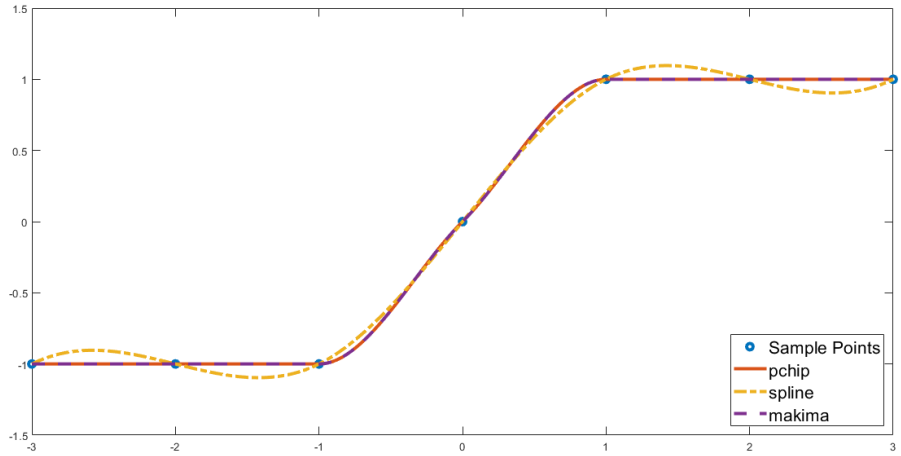

Figure 2.6: Comparison between pchip, spline and makima over flat regions

In this case, pchip and makima have similar behavior in that they avoid overshoots and can accurately connect the flat regions. If we perform another comparison, using an oscillatory sample function, we get the graph in Fig 2.7

When the underlying function is oscillatory, *spline* and *makima* capture the movement between points better than *pchip*, which is aggressively flattened near local extrema.

In the last section, we have used the *spline* function to run our experiments. Let's see what happens if we use *makima* or *pchip*. Since we are using two utility functions, the idea is to represent the data in four tables, where for each table we have a fixed utility function and a fixed value function approximation method:

Table 2.3: Power utility-makima table broken by Grids and Scenarios

Grid	Gauss Hermite	Quantile	Monte Carlo
Tchebysheff Grid	−0.023	-0.0055	-0.0056
Sparse Grid	−0.024	-0.0056	-0.0056

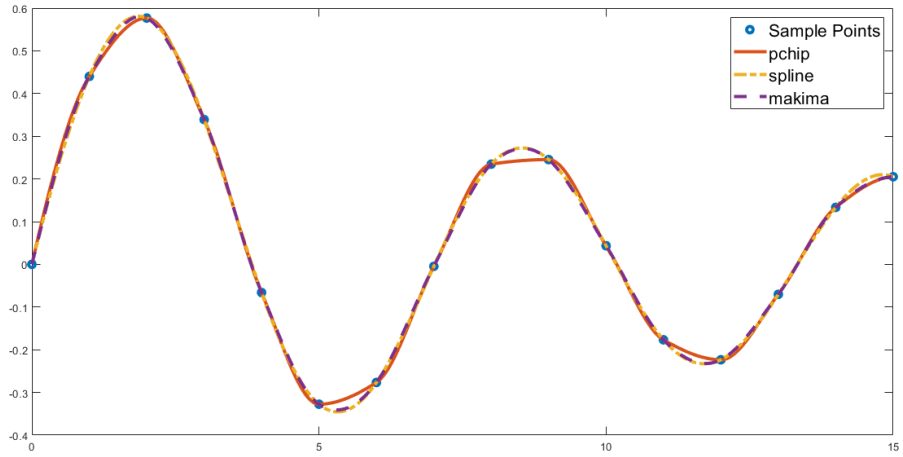

Figure 2.7: Comparison between pchip, spline and makima using an oscillatory function

Table 2.4: Power utility-pchip table broken by Grids and Scenarios

Grid	**Gauss Hermite**	**Quantile**	**Monte Carlo**
Tchebysheff Grid	−0.023	-1.1791	-0.0059
Sparse Grid	−0.024	-0.0055	-0.0056

Table 2.5: Log-makima table broken by Grids and Scenarios

Grid	**Gauss Hermite**	**Quantile**	**Monte Carlo**
Tchebysheff Grid	29.461	33.6187	33.6199
Sparse Grid	29.451	33.6113	33.6203

Table 2.6: Log-pchip table broken by Grids and Scenarios

Grid	**Gauss Hermite**	**Quantile**	**Monte Carlo**
Tchebysheff Grid	29.458	0.3234	27.78
Sparse Grid	29.449	33.624	33.627

From the tables, we observe that the couple (Sparse Grid, Monte Carlo) seems the one that maximize the utility function for each value function approximation method. We can try to play a little bit with the scenrio generation methods and the

grids looking for strategies to improve our results. Since the Monte Carlo approach is computational expensive, first we can try with low discrepancy sequences as we are going to see in the next section.

2.4 Refinements

To adjust the script for deterministic discretization using low discrepancy sequences, we can replace the random sampling part with the generation of quasi-random points using a low discrepancy sequence. One common choice for this purpose is the Sobol sequence. MATLAB provides the *sobolset* and *sobol* functions for generating Sobol sequences. To do that, we can use the following Matlab script

```
1   function [values, probs] = MakeScenariosQ(
    numScenarios, mu, sigma)
2   % Initialize output
3   probs = ones(numScenarios, 1) / numScenarios;
4   values = zeros(numScenarios, 1)';
5
6   % Generate Sobol sequence points
7   sobolPoints = sobolset(1, 'Skip', 1e3, 'Leap', 1e2);
8   sobolSamples = net(sobolPoints, numScenarios);
9
10  % Find extreme points of subintervals
11  y = zeros(numScenarios + 1, 1);
12  for h = 1:(numScenarios - 1)
13      P = h / numScenarios;
14      y(h + 1) = logninv(P, mu, sigma);
15  end
16  y(numScenarios + 1) = exp(mu + 20 * sigma);
17
18  f = @(x) x .* lognpdf(x, mu, sigma);
19
20  % Find expected values by integrating pdf on each
    subinterval
21  for h = 1:numScenarios
22      intervalStart = y(h);
23      intervalEnd = y(h + 1);
24
25      % Map Sobol samples to the specified interval
```

```
26          mappedSamples = intervalStart + (intervalEnd -
    intervalStart) * sobolSamples(h);
27
28          % Calculate expected values using the mapped
    samples
29          values(h) = mean(f(mappedSamples)) * (
    intervalEnd - intervalStart);
30      end
31  end
```

In this script, the Sobol sequence is utilized to generate quasi-random points (sobolSamples), which are then mapped to the specified subintervals. Expected values are computed using the mapped samples and the specified probability density function. We can extract the optimal results from the previous section by running simulations using the low discrepancy approach instead of the Monte Carlo one. In the case of the power utility function, each value function approximation method yields -0.0061. For the log utility function, the results are 32.7568 for the pchip function, 32.7557 for makima, and 32.7582 for spline. In both cases, we do not observe superior results compared to the Monte Carlo approach.
Another approach could involve using Voronoi cell sampling [2]. The Voronoi cell sampling method entails generating random points in a space and assigning each point to the nearest cell boundary. Here, we will employ this approach to create scenarios for lognormal variables.

```
1     function [values, probs] = MakeScenariosQ(
    numScenarios, mu, sigma)
2     % Initialize output
3     probs = ones(numScenarios, 1) / numScenarios;
4     values = zeros(numScenarios, 1)';
5
6     % Generate Sobol sequence points
7     sobolPoints = sobolset(1, 'Skip', 1e3, 'Leap', 1e2);
8     sobolSamples = net(sobolPoints, numScenarios);
9
10    % Find extreme points of subintervals
11    y = zeros(numScenarios + 1, 1);
```

[2] see [5] for further details

```
    for h = 1:(numScenarios - 1)
        P = h / numScenarios;
        y(h + 1) = logninv(P, mu, sigma);
    end
    y(numScenarios + 1) = exp(mu + 20 * sigma);

    f = @(x) x .* lognpdf(x, mu, sigma);

    % Find expected values by integrating pdf on each
  subinterval
    for h = 1:numScenarios
        intervalStart = y(h);
        intervalEnd = y(h + 1);

        % Map Sobol samples to the specified interval
        mappedSamples = intervalStart + (intervalEnd -
  intervalStart) * sobolSamples(h);

        % Calculate expected values using the mapped
  samples
        values(h) = mean(f(mappedSamples)) * (
  intervalEnd - intervalStart);
    end
end
```

Table 2.7: Log-Sparse grid table with Voronoi cell sampling broken by approximation functions

function	Value
pchip	32.7635
makima	32.7624
spline	-75.8984

From 2.7 and 2.8 we can infer that we are not able to do better than the Monte Carlo approach and in general we have poor results comparing the Voronoi cell sampling with the low discrepancy method. It's interesting to underline the nasty effects that we get when we combine the Voronoi cell sampling with the *spline*. This is linked with the oscillatory behaviour of the spline an the possibility of

Table 2.8: Power utility-Sparse grid table with Voronoi cell sampling broken by approximation functions

function	Value
pchip	-0.0061
makima	-0.0061
spline	-3.89×10^{10}

having flat regions when using Voronoi cell sampling.

2.4.1 Scenario generation alternatives

We can explore alternative approaches to the sparse grid method, investigating whether we can achieve better results or not. An idea is to employ a "Convex" grid. Recall that $W_{\max}$ and $W_{\min}$ are respectively the maximum and minimum values of wealth W (in our experiments, we will continue to set $W_{\min} = 1$ and $W_{\max} = 200$). We can use the *nthroot* function from Matlab to construct an adaptive grid based on a geometric progression. A possible implementation in MATLAB could be the following script:

```
    % Calculate the progression factor to achieve exactly
     numPoints values
alpha = nthroot(Wmax / Wmin, numPoints);

% Geometric progression for adaptive grid with exactly
   numPoints values
wealthValues = Wmin * alpha.^(1:numPoints); %convex

% Ensure the last value is not greater than Wmax
wealthValues(end) = min(wealthValues(end), Wmax);

wealthGrid = repmat({wealthValues(:)}, timeHorizon, 1);
```

In our experiments we set $numPoints = 30$. It's interesting to see the results when run simulations using the Monte Carlo approach and the different types of value function approximation methods as shown in Table 2.9 and Table 2.10.

We observe that we are really close to what we have found combining the other methods. Since we are running simulation and the value are relatively close, how

Table 2.9: Log-Adaptive grid table with Monte Carlo sampling broken by approximation functions

function	**Value**
pchip	33.6265
makima	33.6265
spline	33.5961

Table 2.10: Power utility-Adaptive grid table with Monte Carlo sampling broken by approximation functions

function	**Value**
pchip	-0.0056
makima	-0.0056
spline	-0.0056

do we select the *best* method ? A metric that we could take into account is the average elapsed time of a method, as we will see in the next chapter.

Chapter 3

Time analysis

In the previous chapter, we selected the pair (Sparse Grid, Monte Carlo) as the one that maximizes our utility function. However, there were other combinations that achieved either the same or close values of the utility function. To choose which of these combinations is suitable for applications, a crucial variable to consider is the execution time of these methods. We will treat these times as random variables and aim to understand which one is the fastest and provides the right trade-off between utility function maximization and execution time. Every simulation will be run with the same default parameters used in the previous chapter. It's essential to emphasize that these time values are not absolute, as they depend on the device used for the simulation. What truly matters is that we conduct a *comparison* between them with the same input parameters and on the same computer. In this case, the comparison and analysis make sense and can provide a better understanding of the speed of convergence of our algorithms.

3.1 Sparse Grid - Monte Carlo time analysis

Let's focus on the Sparse Grid-Monte Carlo approach, considering that it yielded the best results for the log-utility function with each value function approximation. Assuming we fix *pchip* as the value function approximation method and the log as the utility function, we can run the method multiple times, noting the elapsed times. Let's consider running our script 10 times and recording the elapsed time each time. We can model the elapsed time T_1 as a random variable. Our goal initially is to test if it's reasonable to assume that T_1 follows a normal distribution. If this assumption holds, we can consider the sampling mean as a representative value for T_1. To check if T_1 is normally distributed, we can employ the Anderson-Darling test. A possible implementation of this test in MATLAB is provided in the following script:

```
%% Time Analysis of Sparse Grid, pchip, Monte Carlo

data = [205.632836, 204.033521, 204.472278, 204.6828,
   204.149618, 204.116087, 202.880624, 204.297133,
   204.665476, 204.770684];
mean(data)
std(data)
% Anderson-Darling test
[h_ad, p_ad, stat_ad, crit_ad] = adtest(data);

% Normal probability plot
figure;
qqplot(data);
title('Normal Probability Plot');

% Display results of the Anderson-Darling test
fprintf('Anderson-Darling Test:\n');
fprintf('Test Statistic: %f\n', stat_ad);
fprintf('p-value: %f\n', p_ad);

% Display result based on significance level (e.g.,
   0.05)
if p_ad < 0.05
    fprintf('The data does not appear to be from a
   normal distribution.\n');
else
    fprintf('The data appears to be from a normal
   distribution.\n');
end
```

The outcome of the test is:

Anderson-Darling Test:
Test Statistic: 0.457316
p-value: 0.215949
The data appears to be from a normal distribution.

We can resonably consider the mean of our sample, that in our case is 204.3701

seconds. Matlab allows us to plot the normal probability plot. In this case the normal probability plot is shown in Fig. 3.1

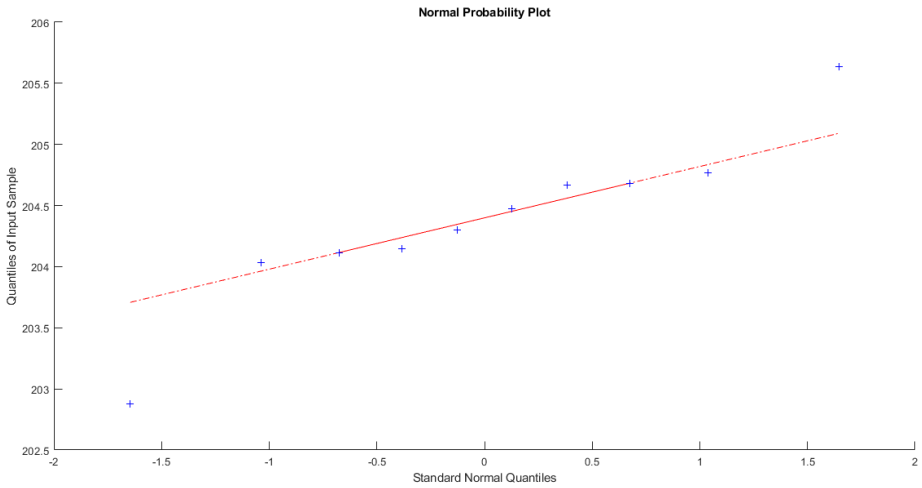

Figure 3.1: Normplot of the sample data taken from T_1

3.2 Comparing more methods

As we have done in the previous section, we can use the same strategy for other methods. Let's always consider the log-utility function and we are going to consider the following methods:

1. Sparse Grid, pchip, Monte Carlo
2. Sparse Grid, pchip, low discrepancy
3. Adaptive Grid, pchip, Monte Carlo
4. Tchebysheff Grid, makima, Monte Carlo

We can map each of these methods in a couple of real numbers (u_i, t_i) where u_i is the utility function value and t_i is the mean elapsed time for $i = 1, \ldots, 4$. For the first one, in the last section we had $t_1 = 204.3701$. For the other three methods, the strategy is always the same. We have performed an Anderson-Darling test on every sample data of size 10. As in the first case, we can consider that the samples are taken from a normal distribution and their sampling mean times are $t_2 = 199.3491$, $t_3 = 207.9577$, $t_4 = 213.7311$. If we recall that $u_1 = 33.627$,

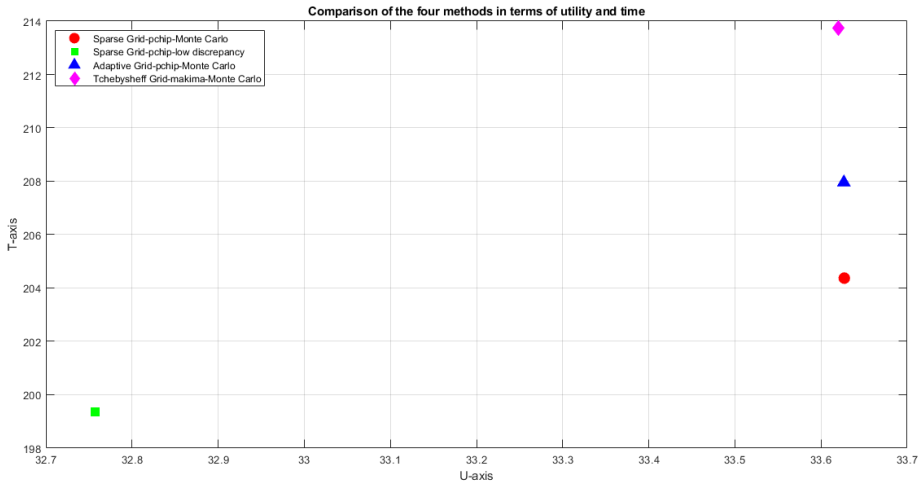

Figure 3.2: Comparison of the four methods

$u_2 = 32.7568$, $u_3 = 33.6265$, $u_4 = 33.6199$ we can plot the points u_i, t_i on a plane as swown in Fig. 3.2.

We can observe that the low discrepancy one (labeled as a green square) is in the left bottom corner, as expected. In general, the low discrepancy approach is less time-consuming than the Monte Carlo one. It's also true that we have worse performances in terms of the utility function value.
On the other hand, the Tchebysheff approach is the most time-consuming (violet rhombus), and compared to the other two, it gives a lower utility function value. The Adaptive Grid (blue triangle) and Sparse Grid (orange circle) are really close in terms of the utility function (33.6265 vs. 33.627), but the Sparse Grid one is faster.
So, a possible choice among all the methods could be the Sparse Grid-pchip-Monte Carlo one.

Bibliography

[1] Paolo Brandimarte. «From Shortest Paths to Reinforcement Learning A MATLAB-Based Tutorial on Dynamic Programming». In: 20 (Nov. 1999), pp. 569–571 (cit. on pp. 1, 4, 9, 12, 17).

[2] John Y. Campbell and Luis M. Viceira. «Strategic Asset Allocation: Portfolio Choice for Long-Term Investors». In: (Apr. 2001), pp. 174–197 (cit. on p. 1).

[3] Zachary Battle† and Lloyd N. Trefethen. «AN EXTENSION OF MATLAB TO CONTINUOUS FUNCTIONS AND OPERATORS». In: *SIAM J. SCI. COMPUT* 25 (2004), pp. 1743–1770 (cit. on p. 4).

[4] Garcke Jochen. «Sparse Grids and Applications». In: (2012), pp. 57–80 (cit. on p. 5).

[5] Nils Löhndorf. «An empirical analysis of scenario generation methods for stochastic optimization.» In: *European Journal of Operational Research* 255 (2016), pp. 121–132 (cit. on p. 27).