Documentation of `Matlab` object `FinElObj`

Daniel Barczyk and Matthias Kredler

8 January 2014

1 Description

This object automates many programming steps for finite-difference methods in continuous-time dynamic programming and differential games. The object is set up as a *structure* in `Matlab`. The structure's fields contain the information on the model (such as grids for the state variables, value and policy functions), and various functions can be carried out on the object (*methods*, in the language of object-oriented programming).

The reader should first go at least over the first two example program listed in Section 1.1 to get acquainted with how the object works in general. Alongside, one can read the short descriptions of the functions in Section 1. Later on, the other sections of this documentation document provide more detailed information: Section 2 provides an overview of the definitions, notion and assumptions needed. for example on how numerical derivatives and extrapolations are taken (Section 3), how the backward and forward operators for are constructed which are central to the workings of the object (Sections 4 and 5), how value-function iteration is carried out (Section 5.1) and how densities are mapped forward (Section 5.2).

1.1 Example programs

The following three folders contain examples that illustrate how to use `FinElObj`. The file `.._main.m` always contains the main program which calls the other functions. Starting with the simplest and going to the more elaborate ones, the examples are:

1. `Diffusion`: Simple example on how to set up a state-space object for a stochastic process and how to use it to visualize the evolution of a density over time (done by `GetStatDensity`) and to draw a sample path (done by `GetHistory`). It also illustrates how to calculate the distribution of hitting times for a subset of the state space using `GetDistHitTimes`, which may be skipped upon first reading.

2. `Bewley`: Solves a simple Bewley (i.e. consumption-savings) model. It uses the file `BewleyPolicyFct`, which calculates the policies for the value-

function-iteration algorithm (`ValueFctIter`). Also, `GetBewleyVGuess` is used to obtain guess for the value function.

3. `AltruismUncert`: A differential game of two altruistic players under income uncertainty who decide on transfers and savings. Uses `AltruismPolicyFct` to obtain equilibria by backward induction, which is performed by `ValueFctIter2`. The other `m`-files starting on `Altruism` provide more specific (and advanced) functionalities for the object.

1.2 Fields of the object

See the beginning of `FinElObjSetUp` (where the object is set up) for a description of the different fields, their dimensions etc.

1.3 Main functions

The following are the main functions used in the above examples. They are located in the main folder.

- `FinElObjSetUp`: Sets up the object with the state space.
- `ValueFctIter`: Several algorithms for value-function iteration. `ValueFctIter` needs a function handle (`S(1).PolFctHdl`) to obtain optimal policies given a value function.
- `ValueFctIter2`: Like `ValueFctIter`, but allows for a series of pre-runs with coarser grids.
- `GetStatDensity`: Calculates the stationary density for a given law of motion (given in fields `S.a`, `S.s` and `S.h`).
- `GetHistory`: Gets a sample path/history of the system (should be used for illustrative purposes, not for simulation).
- `GetDistHitTimes`: For an arbitrary subset of the state space, this function obtains the distribution of hitting times (i.e. when the process first hits this set) for all points in the state space. Section 5.3 explains the maths used here.

1.4 Other useful functions

The following are other functions provided, some of them used heavily by the main functions above. They can be found in the main folder.

- `GetJac`: Calculates numerical first (upward) and second (centered) derivatives and extrapolates the value function outside the grid (mathematical description in Section 3).

- `GetJacCnt`: Using the upward derivatives calculated by `GetJac`, this function calculates centered first derivatives (mathematical description in Section 3).
- `GetMargDens`: Calculates marginal densities for state variables (using the stationary density calculated by `GetStatDensity`).
- `GetMoments`: Calculates first and second moments for the distribution of state variables (using the stationary density calculated by `GetStatDensity`).
- `GetUpWindDer`: Obtains the derivative of an arbitrary function according to the upwind principle.
- `GetUpWindLOM`: Obtains the law of motion according to the upwind principle.
- `GetHamMatrix`: For a given law of motion, sets up matrices for value-function iteration and density iterations. Described in Section 4, other linear operators provided by the object are described in Section 5.
- `GetDt`: Given the matrices set up by `GetHamMatrix`, obtains a suitable time interval for value-function or density iteration.

1.5 Helper functions

The following are helper functions which are called by the above functions and are of limited use by themselves. They are located in folder `SmallFct`.

- `SetUpField`: Sets up field in object, called by `FinElObjSetUp`.
- `GetLogGrid`: Constructs logarithmic grid.
- `GetDrawProbDist`: Obtains draw from finite-valued probability distribution.
- `GetMomentsProbDist`: Obtains moments for finite-valued probability distribution.
- `GetQuantProbDist`: Obtains percentiles for finite-valued probability distribution.
- `interpnGen`: "Transfers" one function (typically value function) from coarse grid to a finer one; uses `Matlab` function `interpn`.

2 Definitions, assumptions etc.

Definitions:

- x: n-dimensional vector of continuous states with grid sizes $I_1, \ldots, I_n$ (the corresponding in the object is `S(i).xGrid`, which gives the values that variable i takes for each grid point).

- x_{-i}: The $(n-1)$-vector of continuous states *excluding* state i.
- y: m-dimensional vector of discrete states with the respective number of states $J_1, \ldots, J_m$ (values that variable j takes is in `S(j).xGrid`).
- $V(t, x, y)$: value function at time t at point (x, y); the corresponding field in the object is `S(1).V.val`.
- $a_i(t, x, y)$: drift of continuous variable x_i, $i = 1, \ldots, n$, at point (x, y) at t. We write $a_i^+ \equiv \max\{0, a_i\}$ and $a_i^- \equiv \min\{0, a_i\}$ (object field: `S(i).a`).
- $s_i(t, x, y)$: volatility of variable x_i at point (x, y), $i = 1, \ldots, n$ (object field: `S(i).s`).
- $h_j(k, l)$: transition rate of variable j, $j = 1, \ldots, m$, from state k to state l, where $k, l \in \{1, \ldots, J_j\}$ (object field: `S(i).h`). We define $h_j(k, k) = -\sum_{l \neq k} h_j(k, l)$. This definition helps us in writing the Hamilton-Jacobi-Bellman equation (HJB) and keeping the code simple.

Assumptions/Requirements:

1. Shocks to the different states are orthogonal to each other.
2. For all $j = 1, \ldots, m$, the transition rates $h_j(k, l)$ depend only on state y_j, but not on the other states $\{x, y_{-j}\}$. Note that we can always pool any set of states with dependent transition probabilities into one big state.

Continuous variable $i \in \{1, \ldots, n\}$, evolves according to the stochastic differential equation

$$\mathrm{d}x_{i,t} = a_i(t, x_t, y_t)\mathrm{d}t + s_i(t, x_t, y_t)\mathrm{d}B_{i,t},$$

where $\{B_{i,t}\}_{i=1}^n$ are uncorrelated Brownian motions. The following is the continuous-time HJB:

$$\begin{aligned} -V_t(t, x, y) = &- \rho V(t, x, y) + FU(t) + \sum_{i=1}^{n} \left[a_i V_i(t, x, y) + \tfrac{1}{2} s_i^2(x) V_{ii}(t, x, y)\right] + \\ &+ \sum_{j=1}^{m} \sum_{l=1}^{M_j} h_j(y_j, l) V(t, x, y_{-1}, l), \end{aligned} \tag{1}$$

where $FU(t)$ is flow utility at t and subscripts to V denote partial derivatives.

3 Numerical derivatives and extrapolation

We will first describe how numerical derivatives are taken and how extrapolation beyond the grid boundaries is carried out.

3.1 Numerical derivatives

Denote by $\tilde{x} \equiv x_i$ the state variable in one of the continuous dimensions i in order to save on subscripts. Let $(\tilde{x}_1, \ldots, \tilde{x}_k, \ldots, \tilde{x}_{I_i})$ be the grid vector for this dimension.[1] Define the upward difference between grid points in the $\tilde{x}$-dimension by

$$\Delta\tilde{x}_k \equiv \tilde{x}_{k+1} - \tilde{x}_k.$$

This upward difference put into field `S(i).dx` (for dimension k) when the state space is set up by `FinElObjSetUp`.

The centered difference is defined as the average between the upward and downward difference (put into field `S(i).dxc` by `FinElObjSetUp`):

$$\Delta_c\tilde{x}_k \equiv \tfrac{1}{2}(\Delta\tilde{x}_k + \Delta\tilde{x}_{k-1}).$$

The numerical (upward) first derivatives of function f in the $\tilde{x}$-dimension at $(\tilde{x}_k, \cdot)$ are

$$f_i(\tilde{x}_k, \cdot) = \frac{f(\tilde{x}_{k+1}, \cdot) - f(\tilde{x}_k, \cdot)}{\Delta\tilde{x}_k},$$

where the dot stands for other dimensions of the state space which are held constant. For the value function, these numerical derivatives are calculated by `GetJac` and recorded in field `S(1).V.jac(:,..,j,i)` for player j in dimension i. These are the derivatives used by the value-function-iteration algorithms that are described below.

The second derivative is calculated using centered differences of the first derivative (note that the first derivative is a precise estimate of the slope *between* grid points, so we should take the distance between the middle points of the adjacent intervals in the denominator now):

$$f_{ii}(\tilde{x}, \cdot) = \frac{f_i(\tilde{x}_k, \cdot) - f_i(\tilde{x}_{k-1}, \cdot)}{\Delta_c\tilde{x}_k}.$$

For the value function, f_{ii} is calculated by `GetJac` and recorded in `S(1).V.SecDer(..,j,i)` for player j in dimension i. Second cross derivatives are not calculated.

Finally, we define the centered first derivative as the average between the two adjacent upward derivatives (these are used by centered-difference algorithms):

$$f_{i(c)}(\tilde{x}_k, \cdot) = \frac{f_i(\tilde{x}_{k-1}, \cdot) + f_i(\tilde{x}_k, \cdot)}{2}.$$

For the value function, $f_{i(c)}$ is calculated by `GetJacCnt` (given upward derivatives in `S(1).V.jac`) and recorded in `S(1).V.JacCnt(..,j,i)` for player j in dimension i. For the lowest grid point, we set the centered derivative equal to the upward derivative: $f_{i(c)}(\tilde{x}_1, \cdot) \equiv f_i(\tilde{x}_1, \cdot)$. For the highest grid point, we use the above formula with an extrapolated value $f_i(\tilde{x}_N, \cdot)$, which will be discussed right now.

[1] The object allows for both linearly- and non-linearly-spaced grids. However, we have found that linear grids yield the most reliable results. This is in line with the literature on finite-element methods for PDEs, which uses mainly linear grids.

3.2 Extrapolation

Note that the above algorithm leaves us without values for f_i at the top grid point $\tilde{x}_N$ due to differencing (to save on notation, we denote by $N \equiv I_i$ the number of grid points of variable $\tilde{x}$). Similarly, we still have no values for f_{ii} at the top and bottom grid points $\{\tilde{x}_1, \tilde{x}_N\}$. We proceed as follows to fill these gaps.

For the top grid point N we extrapolate the second derivative linearly to obtain an estimate at the grid boundary (for the bottom $\tilde{x}_1$, we can proceed similarly):

$$f_{ii}(\tilde{x}_N, \cdot) = f_{ii}(\tilde{x}_{N-1}, \cdot) + \Delta\tilde{x}_{N-1} \underbrace{\frac{f_{ii}(\tilde{x}_{N-1}, \cdot) - f_{ii}(\tilde{x}_{N-2}, \cdot)}{\Delta\tilde{x}_{N-2}}}_{=f_{iii}(\tilde{x}_{N-2})}.$$

Now, having an estimate for the second derivative on the top of the grid, we obtain an estimate for the first (upward) derivative leaving the grid:

$$f_i(\tilde{x}_N, \cdot) = f_i(\tilde{x}_{N-1}, \cdot) + f_{ii}(\tilde{x}_N, \cdot)\Delta_c\tilde{x}_N.$$

Again, using this estimate for the first derivative, we can now extrapolate the function f itself:

$$f(\tilde{x}_{N+1}, \cdot) = f(\tilde{x}_N, \cdot) + f_i(\tilde{x}_N, \cdot)\Delta\tilde{x}_N,$$

where $\tilde{x}_{N+1}$ lies outside the grid.

By successively substituting the above expressions into each other, we can find the following expression for the extrapolated value $V(\tilde{x}_{N+1}, \cdot)$ as a linear combination of values at the lower grid points:

$$V_{N+1} = c_0 V_N - c_1 V_{N-1} + c_2 V_{N-2} - c_3 V_{N-3},$$

where

$$\begin{aligned}
c_0 &= 1 + \frac{\Delta\tilde{x}_N}{\Delta\tilde{x}_{N-1}} + \frac{\Delta\tilde{x}_N\Delta_c\tilde{x}_N}{\Delta\tilde{x}_{N-1}\Delta_c\tilde{x}_{N-1}} + \frac{\Delta\tilde{x}_N\Delta_c\tilde{x}_N}{\Delta\tilde{x}_{N-2}\Delta_c\tilde{x}_{N-1}} && \simeq 4,\\
c_1 &= \frac{\Delta\tilde{x}_N}{\Delta\tilde{x}_{N-1}} + \frac{\Delta\tilde{x}_N\Delta_c\tilde{x}_N}{\Delta_c\tilde{x}_{N-1}}\left(\frac{1}{\Delta\tilde{x}_{N-1}} + \frac{1}{\Delta\tilde{x}_{N-2}}\right)\\
&\quad + \frac{\Delta\tilde{x}_N\Delta_c\tilde{x}_N\Delta\tilde{x}_{N-1}}{\Delta\tilde{x}_{N-2}}\left(\frac{1}{\Delta_c\tilde{x}_{N-1}\Delta\tilde{x}_{N-1}} + \frac{1}{\Delta_c\tilde{x}_{N-1}\Delta\tilde{x}_{N-2}} + \frac{1}{\Delta_c\tilde{x}_{N-2}\Delta\tilde{x}_{N-2}}\right) && \simeq 6,\\
c_2 &= \frac{\Delta\tilde{x}_N\Delta_c\tilde{x}_N}{\Delta\tilde{x}_{N-2}\Delta_c\tilde{x}_{N-1}}\\
&\quad + \frac{\Delta\tilde{x}_N\Delta_c\tilde{x}_N\Delta\tilde{x}_{N-1}}{\Delta\tilde{x}_{N-2}}\left(\frac{1}{\Delta_c\tilde{x}_{N-1}\Delta\tilde{x}_{N-2}} + \frac{1}{\Delta_c\tilde{x}_{N-2}\Delta\tilde{x}_{N-2}} + \frac{1}{\Delta_c\tilde{x}_{N-2}\Delta\tilde{x}_{N-3}}\right) && \simeq 4,\\
c_3 &= \frac{\Delta\tilde{x}_N\Delta_c\tilde{x}_N\Delta\tilde{x}_{N-1}}{\Delta\tilde{x}_{N-3}\Delta_c\tilde{x}_{N-1}\Delta\tilde{x}_{N-2}} && \simeq 1.
\end{aligned}$$

The closer the grid is to linear, the closer the coefficients are to the estimates given on the right. The estimates hold with equality in the case of a linear grid.

The coefficients $\{c_k\}_{i=1}^4$ are calculated by `FinElObjSetUp` and recorded in the object as `S(1).c = [c0 c1 c2 c3]`. They are also inserted into matrix K_{extrap} by `FinElObjSetUp`.

4 The Hamiltonian as a linear operator

We will now write the Hamiltonian from the HJB as a linear operator on the vectorized value function. We exclude the flow-utility term FU because it is usually not a linear function of V and is thus calculated separately. Using the up-wind principle for the first derivatives and centered derivatives for second derivatives, we obtain

$$\begin{aligned} H =& \sum_{i=1}^{n} a_i^+ \frac{V(x_i+\Delta x_i,\cdot)-V(x_i,\cdot)}{\Delta x_i} + a_i^- \frac{V(x_i,\cdot)-V(x_i-\Delta x_i,\cdot)}{\Delta x_i} \\ &+ \tfrac{1}{2}\sum_{i=1}^{n} s_i^2 \frac{\frac{V(x_i+\Delta x_i,\cdot)-V(x_i,\cdot)}{\Delta x_i} - \frac{V(x_i,\cdot)-V(x_i-\Delta x_i,\cdot)}{\Delta x_{i-1}}}{\Delta_c x_i} \\ &+ \sum_{j=1}^{m}\sum_{l=1}^{M_j} h_j(y_j,l) V(t,x,y_{-1},l). \end{aligned}$$

We see that H is a linear combination of values from $V(t,x,y)$ that lie in the vicinity of the current (x,y)-value. We can thus write in matrix form:

$$\mathrm{vec}(H) = K\mathrm{vec}(V),$$

where $\mathrm{vec}(V)$ is the vectorized version of the array V that contains the value function on the grid points. K is a square matrix of size $n_{Sts} \equiv \prod_{i=1}^{n} N_i \prod_{j=1}^{m} M_j$. The matrix K is a function of the law of motion $\left(\{a_i, s_i\}_{i=1}^n, \{h_j\}_{j=1}^m\right)$. Note that K has negative entries on the diagonal and positive entries off the diagonal and his highly sparse[2].

Our goal in the following section is to write a function that returns the matrix K given the law of motion $\left(\{a_i, s_i\}_{i=1}^n, \{h_j\}_{j=1}^m\right)$.

5 Linear operators implemented in `FinElObj`

`FinElObjSetUp` and `GetHamMatrix` construct various linear operators, which can be used for value-function iteration, density iteration and extrapolation:

- K: $n_{Sts} \times n_{Sts}$. This is the matrix K defined just above in Section 4. Used for value-function iteration on the normal, i.e. non-extended, grid (a backward operation) and for mapping the density forward in time/finding

[2] I.e. it has many zero entries.

the stationary density (a forward operation). Also used by the Howard algorithm (i.e. `Howard=1`) in `ValueFctIter` and by `GetStatDensity`.

- K_{ext}: $n_{Sts} \times n_{ExtSts}$. Like K, but takes in an extended state vector of size $n_{ExtSts} \equiv \prod_{i=1}^{n}(N_i+1)\prod_{j=1}^{m} M_j$. Used for backward operations like value-function iteration where V is first extrapolated one point outside the grid and then these points are used to determine the value of "jumping outside the grid" (important for stochastic laws of motion in continuous dimensions). Is used by algorithms `Howard=0` and `Howard=2` in `ValueFctIter`.

- K_{reflx}: $n_{ExtSts} \times n_{Sts}$. Thought of in a forward sense, this matrix reflects down the mass from the extra grid points into the top grid points of the normal grid. The following holds (and is used in `GetHamMatrix` to obtain K once K_{ext} has been built):

  $$K = K_{ext}K_{reflx}.$$

  It can be thought of as first sending the mass to the extended grid ($K'_{ext}m$, where m is $n_{ExtSts} \times 1$) and then reflecting back the mass from the extra grid points into the regular grid: $K'_{reflx}K'_{ext}m$.

- K_{extend}: $n_{ExtSts} \times n_{Sts}$. Extends a vector of normal grid size (n_{Sts}) to the extended grid size (n_{ExtSts}), putting zeros into the extra grid points.

- K_{extrap}: $n_{ExtSts} \times n_{ExtSts}$. Fills the extra grid points by extrapolating from the four top grid points (see the procedure described in Section 3.2). Eliminates old values for the extra grid points.

The following variations/combinations are useful:

- $K_{extrap}K_{extend}$: $n_{ExtSts} \times n_{Sts}$. First extends a function (say V) to the extended grid, then extrapolates. Is used in `ValueFctIter` by the algorithms that work on the extended grid.

- K'_{extend}: $n_{Sts} \times n_{ExtSts}$. Shrinks a vector of extended-grid size to normal size, eliminating the extra fields.

5.1 Value-function iteration

The function `ValueFctIter` uses the following approximation to iterate backward the HJB given in (1):

$$\begin{aligned} V(t-\Delta t,\cdot) &= FU(t)\Delta t + (1-\rho\Delta t)V(t,\cdot) + K \times V(t,\cdot)\Delta t \\ &= FU(t)\Delta t + \Big[-\rho I\Delta t + \underbrace{I + K\Delta t}_{\equiv H_{\Delta t}} \Big] V(t,\cdot). \end{aligned} \tag{2}$$

Note that we can interpret $H_{\Delta t}$ as defined here in a probabilistic sense: when Δt is set small enough, the coefficient on the diagonal multiplying $V(t,x,y)$ (which

has the interpretation as the probability of staying in the current state) can be made non-negative for all grid points (x, y); the coefficients on the neighboring cells are positive anyway (since $H_{\Delta t}$ has positive entries off the diagonal). It can be easily checked that the coefficients along rows of $H_{\Delta t}$ add up to one, so they may be interpreted as probabilities of jumping to adjacent grid points.[3]

All value-function-iteration algorithms need a function handle specified in `S(1).PolFctHdl` which calculates optimal policies for the agents and the resulting laws of motion (`S.a` and `S.s`) given a value function `S(1).V.val`.

5.1.1 Non-Howard algorithms

Setting options 0 and 2 in `S(1).options.Howard` makes `ValueFctIter` operate backward on V given a guess for the value function in `S(1).V.val`. Option 0 updates policies each single step to the optimal ones given the current value function; option 2 keeps one policy for at most `S(1).options.tUpdatePol` steps time units. Both algorithms extrapolate the value function in each iteration outside the grid for the continuous dimensions and then take the extrapolated values as the value of jumping outside the grid (i.e. they use K_{ext} for updating and `GetJac` for extrapolation).

5.1.2 Howard improvement algorithm

For the Howard improvement algorithm, we note that $V_t = 0$ for a stationary solution V of the HJB. For given flow utility FU and a matrix K, the stationary value function V solves the linear system

$$(\rho I - K)\text{vec}(V) = \text{vec}(FU),$$

where I is the $n_{Sts} \times n_{Sts}$ identity matrix. This system is solved for V with an efficient solver for sparse matrices. Given this solution, optimal policies and flow utility FU are then calculated for the next iteration. This option is chosen by setting `S(1).options.Howard=1`.

5.2 Finding the invariant distribution

Again, we use the insight that the Hamiltonian matrix K can be used to construct a transition-probability H on our state-space grid. We use $H_{\Delta t}$ from (2) to iterate forward the density n (again, the density is given in its vectorized form):

$$n_{t+\Delta t} = H'_{\Delta t} n_t = (I + K'\Delta t) n_t.$$

We now have to use the transpose (or *adjoint*) of $H_{\Delta t}$ instead of $H_{\Delta t}$ itself. Note that also in the PDEs arising from stochastic calculus (the HJB and the Kolmogorov forward equation), the forward operator is the adjoint of the backward operator in a functional-analysis sense, so it is not surprising that a transpose

[3]Note that we could alternatively define $\tilde{H}_{\Delta t} = (1 - \rho\Delta t)I - K$ and interpret $\rho\Delta t$ as the probability of dying and obtaining zero value.

shows up here. The function `GetStatDensity` iterates forward according to the above equation until convergence and records the result in `S(1).nStat`.

The algorithm uses the matrix K (i.e. `S(1).K`), which means that mass jumping outside the state space on the top grid points of continuous dimensions is immediately reflected back into the the grid. So the grid must be constructed such that the probability mass reaching these boundaries is negligible.

5.3 Finding the distribution of hitting times for a set D

Starting a diffusion at a point x, we are interested in when the process will first hit a set $D \subset X$ (where X is the entire state space). We will study the following function:

$$Q(t,x) \equiv \text{Prob}\big(x_{s+\tau} \in D \text{ for some } \tau \in [0,t] | x_s = x\big).$$

On $\bar{D} = X \setminus D$, we can write Q recursively as:

$$Q(t,x) = \underbrace{Q(\Delta t, x)}_{\text{paths that hit } D \text{ within } \Delta t} + \int_{\text{all other paths}} Q(t-\Delta t, \tilde{x}) f(x_{t+\Delta t} = \tilde{x} | x_t = x) \mathrm{d}\tilde{x},$$

where $f(\cdot|\cdot)$ gives the conditional density of the stochastic process. For sufficiently low Δt the probability that the process hits D within Δt goes to 0, so for all interior $x \in \bar{D}$ we can write:

$$Q(t,x) = \lim_{\Delta t \to 0} \mathbb{E}\big[Q(t+\Delta t, x_{t+\Delta t}) | x_t = x\big].$$

Assuming that Q is twice differentiable, stochastic calculus then implies that $Q(\cdot)$ must satisfy the following PDE for $t < 0$ and $x \notin D$:

$$-Q_t = \mathcal{A}Q,$$

where $\mathcal{A}$ is the infinitesimal generator of the diffusion, which contains first and second derivatives in x. The following are the boundary conditions for this PDE:

$$\begin{aligned} Q(0,x) &= 0 && \text{for all } x \notin D, \\ Q(t,x) &= 1 && \text{for all } t \text{ and } x \in D. \end{aligned}$$

The pdf of hitting times for a fixed x, denote it by $q(\cdot)$, may then be obtained by differentiating Q in the time direction:

$$q(t,x) = Q_t(t,x) = -\mathcal{A}Q \geq 0.$$

Since the pdf must be non-negative, this implies $\mathcal{A}Q \leq 0$. For standard Brownian motion, this equation reads as $\frac{1}{2}Q_{xx} \leq 0$, which means that Q is concave in x. Also for different processes and higher dimensions, we will usually expect concave shapes for Q. The equation also highlights the connection of the

Q-function to the Dirichlet boundary-value problem and the theory of (sub-)harmonic functions related to stochastic processes[4]

The function `GetDistHitTimes` obtains the function Q for arbitrary sets D numerically and calculates statistics of the hitting-time distribution for all $x \in X$.

6 Possible extensions

The following are not yet implemented in the object, but may be useful for future applications:

1. Make it an option which boundaries of continuous dimensions are reflecting and which are open. Let also lower boundaries be open.

2. Allow for endogenous hazard (jump) rates `S(i).h` in discrete dimensions (so unlike now, those would be allowed to vary across the state space).

[4]For an excellent treatment of (sub-)harmonic functions in the context of stochastic processes, see Øksendal, *Stochastic Differential Equations* (2003).