Implementing dynamic programming

Prof. Paolo Brandimarte
Dip. di Scienze Matematiche – Politecnico di Torino
e-mail: `paolo.brandimarte@polito.it`
URL: `https://staff.polito.it/paolo.brandimarte`

This version: May 7, 2025

These slides are taken from my book:

P. Brandimarte. *From Shortest Paths to Reinforcement Learning: A MATLAB-Based Introduction to Dynamic Programming*. Springer, 2021.

MATLAB code may be downloaded from my web page:

`https://staff.polito.it/paolo.brandimarte/`

Other references:

- D.P. Bertsekas. *Dynamic Programming and Optimal Control Vol. 1* (4th ed). Athena Scientific, 2017.
- D.P. Bertsekas. *Dynamic Programming and Optimal Control Vol. 2* (2nd ed). Athena Scientific, 2012.
- L.A. Wolsey. *Integer Programming*. Wiley, 1998.

The knapsack problem requires to select a subset of items of maximal total value, subject to a budget (capacity) constraint.

We introduce binary decision variables to model the selection of each item:

$$x_k = \begin{cases} 1 & \text{if item } k \text{ is selected,} \\ 0 & \text{otherwise.} \end{cases}$$

The problem can be formulated as a pure binary linear program:

$$\begin{aligned} \max \quad & \sum_{k=1}^{n} v_k x_k \\ \text{s.t.} \quad & \sum_{k=1}^{n} w_k x_k \le B \\ & x_k \in \{0,1\} \qquad \forall k. \end{aligned}$$

Note that we are assuming a discrete budget allocation, as the selection of an activity is an all-or-nothing decision.

The problem can be solved quite efficiently by state-of-the-art branch-and-cut algorithms for integer linear programming, but let us pursue a DP approach.

The problem is not really dynamic, but we may recast it as a sequential resource allocation problem by introducing a fictitious discrete time index k, corresponding to items.

For each time index (or decision stage) $k = 1, \ldots, n$, we have to decide whether to include item k in the subset or not. At stage $k = 1$, we have the full budget B at our disposal and we have to decide whether to include item 1 or not in the subset, facing a difficult tradeoff between immediate and future rewards. At stage $k = n$, however, the problem is trivial, since we have only to consider the set $\{n\}$ consisting of the last item, given the residual budget.

Note that the selection of the next items is influenced by past decisions only through the residual budget. The natural state variable at stage k is the available budget s_k *before* selecting item k, and the decision variable at each stage is the binary variable x_k.

Since there is no item $k = 0$, we write the state transition equation as

$$s_{k+1} = s_k - w_k x_k, \qquad k = 1, \ldots, n,$$

with initial condition $s_1 = B$. Assuming that the resource requirements w_k are integers, the state variable will be an integer number too.

Then, we define the following value function:

$$V_k(s) \doteq \text{profit from the optimal subset selection within the set of items } \{k, k+1, \ldots, n\}, \text{ when the residual budget is } s.$$

In this discrete case, we may tabulate the whole set of value functions $V_k(s)$, $k = 1, \ldots, n$, for integer states s in the range from 0 to B.

The DP recursion is

$$V_k(s) = \begin{cases} V_{k+1}(s) & \text{for } 0 \le s < w_k, \\ \max\{V_{k+1}(s),\ V_{k+1}(s - w_k) + v_k\} & \text{for } w_k \le s \le B. \end{cases} \tag{1}$$

This equation simply states that, at stage k, we consider item k:

- If its weight w_k does not fit the residual budget s, i.e., if $s < w_k$, we can forget about the item. Hence, the state variable is left unchanged, and the value function $V_k(s)$ in that state is the same as $V_{k+1}(s)$.

- Otherwise, we must compare two alternatives:
    1. If we include item k in the subset, the reward from collecting its value v_k and then allocating the updated budget $s - w_k$ optimally to items $k + 1, \ldots, n$.
    2. If we do not include item k in the subset, the reward from allocating all of the residual budget s optimally to items $k + 1, \ldots, n$.

Since the decision is binary, the single-step optimization problem is trivial. The terminal condition, for the last item $k = n$, is just:

$$V_n(s) = \begin{cases} 0 & \text{for } 0 \le s < w_n, \\ v_n & \text{for } w_n \le s \le B. \end{cases} \tag{2}$$

- The function `DPKnapsack` receives vectors `value` and `weight` and a scalar `capacity`.

- It returns the optimal subset of items, represented by a binary vector `X` and the total value `reward` of the optimal subset.

- The value function is stored into the table `valueTable`, a matrix with n columns corresponding to items and $B+1$ rows corresponding to states $s = 0, 1, \ldots, B$. As usual with MATLAB, actual indexing runs from 1 to $B+1$.

- In this case, we may afford not only to store value functions in a tabular form, but also the decisions for each state, which are collected into the matrix `decisionTable`.

- The outermost `for` loop is a straightforward implementation of the DP backward recursion. We do some more work than necessary, since we could do without the function $V_1(\cdot)$. Only $V_1(B)$ is needed, but in this way we are able to find the optimal solution for any initial budget $s_1 = 1, \ldots, B$.

- After the main loop, we build the optimal solution by stepping forward with respect to items, making decisions and adjusting the state variable accordingly.

```
function [X, reward] = DPKnapsack(value, weight, capacity)
% preallocate tables
numItems = length(value);
valueTable = zeros(1+capacity,numItems);
decisionTable = zeros(1+capacity,numItems);
capValues = (0:capacity)'; % this is for convenience
% Initialize last column of tables
decisionTable(:,numItems) = (capValues >= weight(numItems));
valueTable(:,numItems) = decisionTable(:,numItems) * value(numItems);
% Backward loop on items (columns in the value table)
for k=(numItems-1):-1:1
    % Loop on rows (state: residual budget)
    for residual = 0:capacity
        idx = residual+1; % MATLAB starts indexing from 1....
        if residual < weight(k)
            % cannot insert
            decisionTable(idx,k) = 0;
            valueTable(idx,k) = valueTable(idx,k+1);
        elseif valueTable(idx,k+1) > ...
                valueTable(idx-weight(k), k+1) + value(k)
            % it is better to not insert item
            decisionTable(idx,k) = 0;
            valueTable(idx,k) = valueTable(idx,k+1);
        else
            % it is better to insert
            decisionTable(idx,k) = 1;
            valueTable(idx,k) = valueTable(idx-weight(k), k+1) + value(k);
        end
    end % for i
end % for k (items)
```

```
% now find the solution, by a forward decision
% process based on state values
X = zeros(numItems,1);
resCapacity = capacity;
for k = 1:numItems
    if decisionTable(resCapacity+1,k) == 1
        X(k) = 1;
        resCapacity = resCapacity - weight(k);
    else
        X(k) = 0;
    end
end
reward = dot(X,value);
end
```

Let us test the function with a simple example, borrowed from pp. 73–74 of Wolsey:

```
>> value = [10; 7; 25; 24];
weight = [2; 1; 6; 5];
capacity = 7;
[xDP, rewardDP] = DPKnapsack(value, weight, capacity)
xDP =
     1
     0
     0
     1
rewardDP =
    34
```

Apart from the (rather obvious) optimal solution, it is instructive to have a look at the value and decision tables that are produced inside the DP computation, where we show an asterisk alongside each optimal decision.

The decision table is an explicit representation of the policy, where the k-th column corresponds to the optimal policy at stage k.

```
valueTable =                              decisionTable =
     0     0     0     0                        0     0     0     0
     7     7     0     0                        0     1     0     0
    10     7     0     0                        1     1     0     0
    17     7     0     0                        1     1     0     0
    17     7     0     0                        1     1     0     0
    24    24    24    24                        0    *0    *0    *1
    31    31    25    24                        0     1     1     1
    34    32    25    24                       *1     1     1     1
```

The computational complexity of this procedure, assuming integer data, is $O(nB)$, which is not really polynomial but pseudo-polynomial (on a binary computer).

The practical implication is that, when B is a large number, we must build a huge table, making this algorithm not quite efficient.

Let us consider a continuous version of the resource allocation problem, where a resource budget B must be allocated to a set of n activities. The allocation to activity k is expressed by a continuous decision variable $x_k \geq 0$, $k = 1, \ldots, n$.

The contribution to profit from activity k depends on the resource allocation through an increasing and concave function $f_k(\cdot)$:

$$\begin{aligned} \max \quad & \sum_{k=1}^{n} f_k(x_k) \\ \text{s.t.} \quad & \sum_{k=1}^{n} x_k \leq B, \\ & x_k \geq 0 \qquad \forall k. \end{aligned}$$

If the profit functions are concave, like

$$f_k(x) = \sqrt{x}, \qquad k = 1, \ldots, n,$$

the problem is rather easy to solve.

Since the profit functions are strictly increasing, we may assume that the full budget will be used, so that the budget constraint is satisfied as an equality at the optimal solution.

Furthermore, let us assume an interior solution $x_k^* > 0$, so that the optimization problem boils down to a nonlinear program with a single equality constraint.

After introducing a Lagrange multiplier λ associated with the budget constraint, we build the Lagrangian function

$$\mathcal{L}(\mathbf{x}, \lambda) = \sum_{k=1}^{n} \sqrt{x_k} + \lambda \left(\sum_{k=1}^{n} x_k - B \right).$$

The first-order optimality conditions are

$$\begin{aligned} \frac{1}{2\sqrt{x_k}} + \lambda = 0, & \qquad k = 1, \ldots, n, \\ \sum_{k=1}^{n} x_k - B = 0, & \end{aligned}$$

which imply a uniform allocation of the budget among activities:

$$x_k^* = \frac{B}{n}, \qquad k = 1, \ldots, n.$$

For instance, if $n = 3$ and the budget is $B = 20$, the trivial solution is to split it equally among the three activities:

$$x_1^* = x_2^* = x_3^* = \frac{20}{3} \approx 6.666667, \quad \sum_{k=1}^{3} \sqrt{x_k^*} = 3 \times \sqrt{\frac{20}{3}} \approx 7.745967.$$

In this case, the objective function is concave, and the above optimality conditions are necessary and sufficient.

This problem may be recast, just like the knapsack problem, within a DP framework, by associating a fictitious discrete time index $k = 1, \ldots, n$ with each activity.

Let $V_k(s)$ be the optimal profit from allocating a residual budget s (the state variable) to activities in the set $\{k, k+1, \ldots, n\}$. The state transition equation is

$$s_{k+1} = s_k - x_k,$$

with initial condition $s_1 = B$.

The value functions satisfy the optimality equations

$$V_k(s_k) = \max_{0 \le x_k \le s_k} \big\{ f_k(x_k) + V_{k+1}(s_k - x_k) \big\}, \tag{3}$$

with terminal condition

$$V_n(s_n) = \max_{0 \le x_n \le s_n} f_n(x_n) = f_n(s_n).$$

Note that we should enforce a constraint on the state variable, as the residual budget s_k should never be negative. However, since the state transition is deterministic, we transform the constraint on the state into a constraint on the decision variable.

In the continuous case, the value function $V_k(s)$ is an object within an infinite-dimensional space of functions.

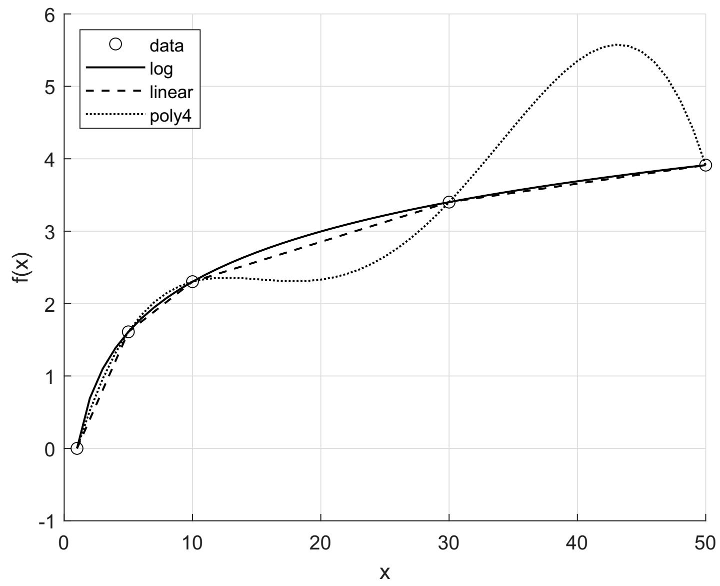
Since we may evaluate $V_k(s)$ only at a finite set of states, we have to resort to some approximation or interpolation method to find state values outside the grid. Let us consider polynomial interpolation and cubic splines.

Polynomial interpolation is performed quite easily in MATLAB:

```
% write data and choose grid
xGrid = [1 5 10 30 50];   % grid points (in sample)
yGrid = log(xGrid);
xOut = 1:50;              % out of sample points
% plot the data points and the function
hold on
plot(xGrid,yGrid,'ok','DisplayName','data')
plot(xOut,log(xOut),'k','DisplayName','log','Linewidth',1);
% plot the linear interpolating function
plot(xOut,interp1(xGrid,yGrid,xOut),'k--','DisplayName','linear', ...
    'Linewidth',1);
% plot the polynomial interpolating function
poly4 = polyfit(xGrid, yGrid, 4);
plot(xOut,polyval(poly4,xOut),'k:','DisplayName','poly4','Linewidth',1)
legend('location','northwest');
grid on
hold off
xlabel('x'); ylabel('f(x)');
```

It is well known that polynomial interpolation suffers from unacceptable oscillations.

Unfortunately, a nice, concave, and monotonically increasing function is approximated by a non-monotonic function, which will clearly play havoc with optimization procedures.



A standard alternative is to resort to a piecewise polynomial function of lower order, where each piece is a polynomial associated with a single subinterval on the grid. A common choice is to use cubic splines, which is easily accomplished in MATLAB by a pair of functions:

- `spline`, which returns a spline object, on the basis of grid values;
- `ppval`, which evaluates the spline on arbitrary points, out of sample (i.e., outside the grid of data points).

Let us prepare a function to experiment with different grids:

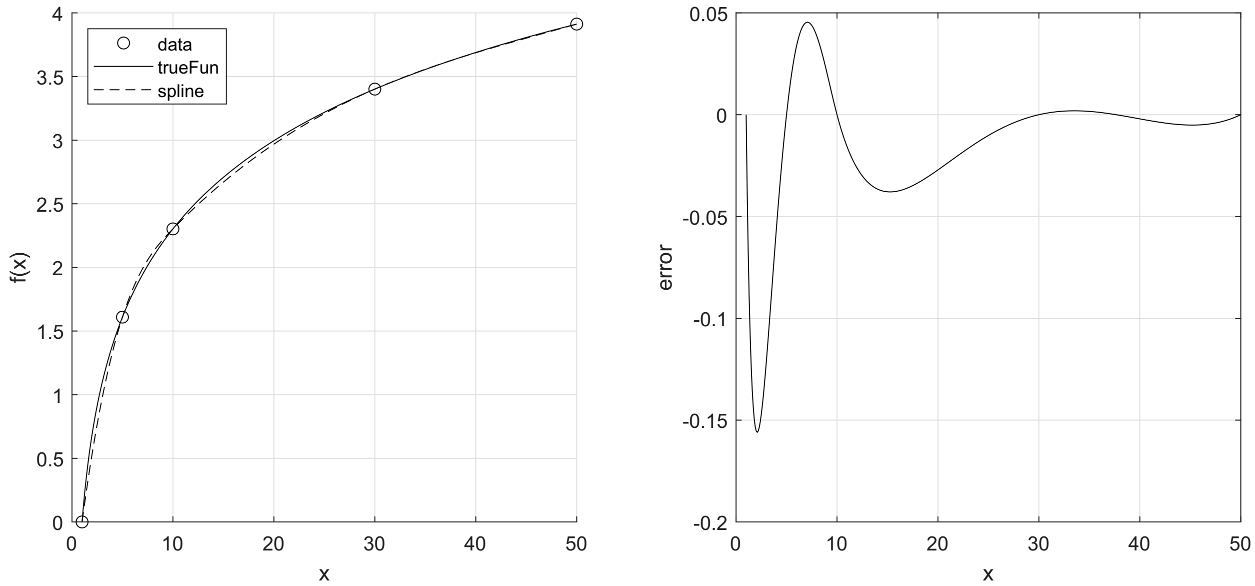
```
function PlotSpline(xGrid,xOut,fun)
yGrid = feval(fun,xGrid);
yOut = feval(fun,xOut);
subplot(1,2,1)
hold on
plot(xGrid,yGrid,'ok','DisplayName','data')
plot(xOut,yOut,'k','DisplayName','trueFun');
% plot the spline
spl = spline(xGrid,yGrid);
splineVals = ppval(spl,xOut);
plot(xOut,splineVals,'k--','DisplayName','spline')
grid on
legend('location','northwest');
hold off
xlabel('x'); ylabel('f(x)');
% plot error
subplot(1,2,2)
plot(xOut,splineVals-yOut,'k')
grid on
xlabel('x'); ylabel('error');
end
```

Here, we use `feval` to evaluate a generic function `fun`, which is an input parameter.

For the previous (coarse) grid, we may call the function as follows:

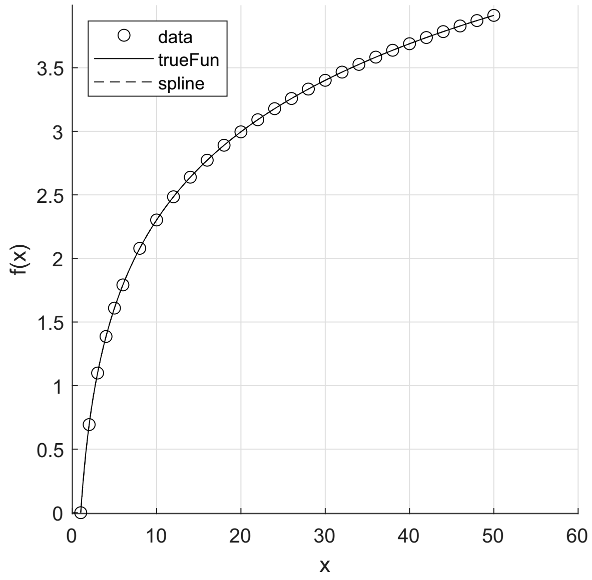
```
xgrid1 = [1 5 10 30 50];
xout = 1:0.1:50;
PlotSpline(xgrid1,xout,@log)
```

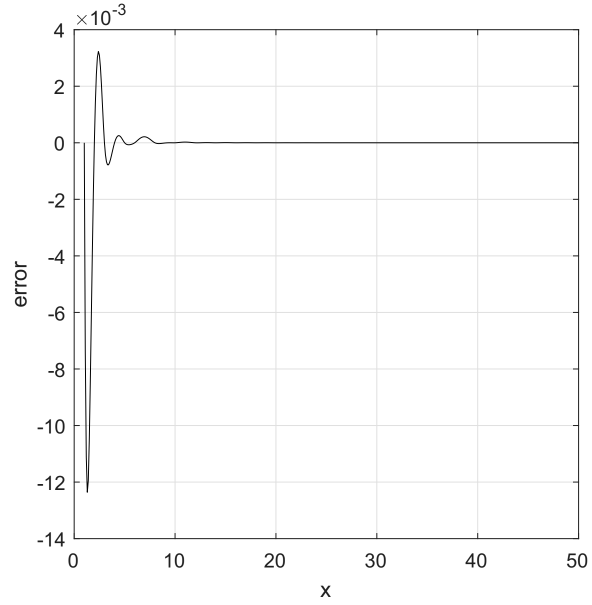
The error is not quite negligible, especially close to point $x = 1$, where the log function is steeper.



Let us try a finer grid:

```
xgrid2 = [1:5, 6:2:50];
xout = 1:0.1:50;
PlotSpline(xgrid2,xout,@log)
```





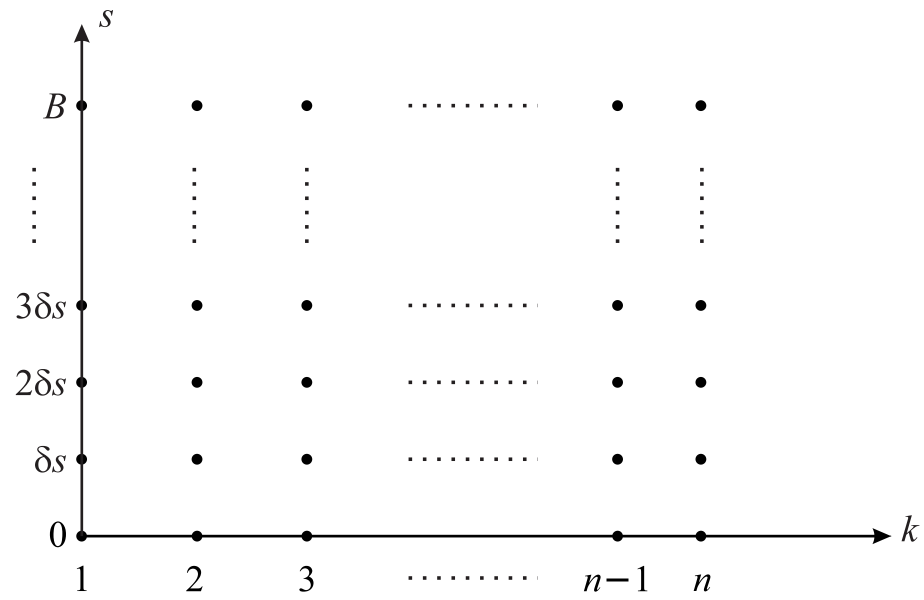
Cubic splines can be a valuable approximation tool, but there are some open issues:

- How can we be sure that a monotonic function will be approximated by a monotonic spline?
- How should we place interpolation nodes to obtain a satisfactory tradeoff between computational effort and approximation error?
- How does the idea generalize to higher dimensions?

We use a cubic spline to approximate the value function of the continuous budget allocation problem.

We set up a uniform grid for the interval $[0, B]$, which is replicated for each stage.

The grid includes $m + 1$ state values for each time instant. The discretization step is $\delta s = B/m$, and we consider states of the form $j \cdot \delta s$, $j = 0, 1, \ldots, m$.



We have to solve a subproblem of the form (3) for each point on the grid, which can be accomplished by the MATLAB function `fminbnd`.

We interpolate outside the grid by cubic splines to approximate values of $V_k(s)$ for an arbitrary residual budget s.

The boundary condition on $V_n(\cdot)$ is trivial, like in the knapsack problem, and we stop with the value function $V_2(\cdot)$, as we assume that we want to solve the problem for a specific budget B.

We cannot store the policy in explicit form as a table, and the optimal policy $\mathbf{x}_t^* = \mu_t^*(\mathbf{s}_t)$ is implicit in the sequence of value functions.

```
function splinesList = findPolicy(budget,funcList,numPoints)
% NOTE: The first element in the spline list is actually EMPTY
% Initialize by setting up a simple grid
budgetGrid = linspace(0,budget,numPoints);
% prepare matrix of function values for each time step
numSteps = length(funcList);
valFunMatrix = zeros(numPoints,numSteps);
% splines will be contained in a cell array
splinesList = cell(numSteps,1);
% start from the last step (increasing contribution functions)
valFunMatrix(:,numSteps) = feval(funcList{numSteps}, budgetGrid);
splinesList{numSteps} = spline(budgetGrid, valFunMatrix(:,numSteps));
% now step backward in time
for t = numSteps-1:-1:2
    % build an objective function by adding the immediate contribution
    % and the value function at the next stage
    for k = 1:numPoints
        b = budgetGrid(k); % budget for this problem
        objFun = @(x) -(feval(funcList{t},x) + ppval(splinesList{t+1},b-x));
        [~,outVal] = fminbnd(objFun, 0, b);
        valFunMatrix(k,t) = -outVal;
    end % for k
    splinesList{t} = spline(budgetGrid, valFunMatrix(:,t));
end % for t
```

The function `findPolicy` receives:

- The amount `budget` to be allocated, which is a single number.
- A cell array `funcList` of functions giving the reward from each activity. This allows us to model profits with arbitrary functions, which may differ across stages.
- The number `numPoints` of points on the grid.

The output is `splinesList`, a cell array of cubic splines, i.e., a list of data structures produced by the function `spline`. Note that the first spline of the list, corresponding to $V_1(\cdot)$, is actually empty, as the first value function we actually need is $V_2(s_2)$, depending on the state s_2 resulting from the first decision x_1. This is why the outermost `for` loop goes from `t = numSteps-1` down to `t = 2`. As we have pointed out, we could extend the for loop down to `t = 1`, in order to find the optimal value for a *range* of possible initial budgets.

The objective function `objFun`, to be used in each optimization subproblem, is built by creating an anonymous function: the `@` operator abstracts a function based on an expression in which `feval` evaluates a function in the list `funcList` and `ppval` evaluates a spline.

```
function [x, objVal] = applyPolicy(budget,funcList,splinesList)
% optimize forward in time and return values and objVal
residualB = budget;
objVal = 0;
numSteps = length(funcList);
x = zeros(numSteps,1);
for t = 1:(numSteps-1)
    objFun = @(z) -(feval(funcList{t},z) + ...
        ppval(splinesList{t+1},residualB-z));
    x(t) = fminbnd(objFun, 0, residualB);
    objVal = objVal + feval(funcList{t}, x(t));
    residualB = residualB - x(t);
end
x(numSteps) = residualB; % the last decision is trivial
objVal = objVal + feval(funcList{numSteps}, x(numSteps));
```

Given the set of value functions produced by `findPolicy`, the function `applyPolicy` of applies the policy forward in time, producing the vector `x` of optimal allocations and the resulting reward `objVal`.

```
f = @(x) sqrt(x);
funHandles = {f; f; f};
budget = 20;
numPoints = 50; % points on the grid, for each time period
splinesList = findPolicy(budget,funHandles,numPoints);
[x, objVal] = applyPolicy(budget,funHandles,splinesList);
```

This simple script creates the cell array of reward functions (square roots in our simple example), and returns

```
>> x
x =
    6.6667
    6.6666
    6.6667
>> objVal
objVal =
    7.7460
```

which is more or less what we expected, within numerical rounding. As we can see, we do find a sensible approximation of the optimal solution.

In this third example, we consider a stochastic variation of the deceptively simple lot-sizing problem, borrowing the example from pp. 23–31 of Bertsekas, Vol. 1.

We assume a discrete random demand. Hence, we may adopt a tabular representation of the value function.

We must specify the state dynamics in the case of a stockout. Here we assume lost sales:

$$I_{t+1} = \max\left\{0, I_t + x_t - d_{t+1}\right\}, \tag{4}$$

where $\left(d_t\right)_{\{t=1,\ldots,T\}}$ is a sequence of i.i.d. variables, and x_t is the amount ordered at time t and immediately delivered.

It is important to understand the exact event sequence:

1. At time instant t, we observe on-hand inventory I_t.
2. Then, we make ordering decision x_t; this amount is immediately received, raising available inventory to $I_t + x_t$.
3. Then, we observe random demand d_{t+1} during time interval $t+1$ and update on-hand inventory accordingly.

When tabulating the value function, we must define an upper bound on the state variable. We assume that there is a limit on inventory, which also implies a constraint on available actions:

$$I_t \le I_{\text{max}}; \qquad \mathcal{X}(I_t) = \{0, 1, \ldots, I_{\text{max}} - I_t\}.$$

The immediate cost comprises:

- A linear ordering cost cx_t, proportional to the ordered amount.

- A quadratic cost related to the "accounting" inventory after meeting demand in the next time interval:

$$\beta\big(I_t + x_t - d_{t+1}\big)^2. \tag{5}$$

  Note that the physical on-hand inventory cannot be negative and is given by Eq. (4). A negative "accounting" inventory represents unmet demand.

- Since we consider a finite horizon problem, we should take care of the terminal state. Nevertheless, for the sake of simplicity, the terminal inventory cost is assumed to be zero,

$$F_T(I_T) = 0.$$

Arguably, the overall penalty should not be symmetric, and we could define a piecewise linear function with different slopes, but this is not really essential for our illustration purposes.

What is more relevant is that we have an immediate cost term that depends on the realization of the risk factor during the time period $t+1$ after making the decision x_t.

This implies that in the DP recursion we do not have a deterministic immediate cost term of the form $f_t(\mathbf{s}_t, \mathbf{x}_t)$, but a stochastic one of the form $h_t(\mathbf{s}_t, \mathbf{x}_t, \boldsymbol{\xi}_{t+1})$.

The resulting DP recursion is

$$V_t\big(I_t\big) = \min_{x_t \in \mathcal{X}(I_t)} \mathrm{E}_{d_{t+1}}\Big[cx_t + \beta\big(I_t + x_t - d_{t+1}\big)^2 + \\ V_{t+1}\Big(\max\big\{0, I_t + x_t - d_{t+1}\big\}\Big)\Big],$$

for $t = 0, 1, \ldots, T-1$, and $I_t \in \{0, 1, 2, \ldots, I_{\max}\}$.

Since the risk factors are just a sequence of i.i.d. discrete random variables, all we need in order to model uncertainty is a probability mass function, i.e., a vector of probabilities π_k for each possible value of demand $k = 0, 1, 2, \ldots, d_{\max}$.

We must also specify the initial state I_0 and the time horizon T that we consider for planning.

```
function [valueTable, actionTable] = MakePolicy(maxOnHand, demandProbs, ...
    orderCost, invPenalty, horizon)
valueTable = zeros(maxOnHand+1,horizon+1);
actionTable = zeros(maxOnHand+1,horizon);
maxDemand = length(demandProbs)-1;
demandValues = (0:maxDemand)';
% Value at t = horizon is identically zero
for t = (horizon-1):-1:0
    for onHand = 0:maxOnHand
        minCost = Inf;
        bestOrder = NaN;
        for order = 0:(maxOnHand-onHand)
            nextInv = onHand+order-demandValues;
            expCost = orderCost*order + dot(demandProbs,...
                invPenalty*(nextInv.^2) + valueTable(max(nextInv,0)+1,t+2));
            if expCost < minCost
                minCost = expCost;
                bestOrder = order;
            end
        end
        valueTable(onHand+1,t+1) = minCost;
        actionTable(onHand+1,t+1) = bestOrder;
    end
end
```

The task of learning the sequence of value functions is performed by the function `MakePolicy`, which receives:

- The maximum inventory level `maxOnHand`.
- The vector `demandProbs` of demand probabilities, where the first element is the probability of zero demand.
- The planning `horizon`, corresponding to T.
- The economic parameters `orderCost` and `invPenalty`, corresponding to c and β, respectively.

We do not make any assumption about the initial inventory, as we also learn the value function $V_0(\cdot)$ for every possible value of the initial state.

The output consists of two matrices, `valueTable` and `actionTable`, representing the value functions ad the optimal policy, respectively, in tabular form.

For both matrices, rows correspond to states, $I_t \in \{0, 1, \ldots, I_{\max}\}$, and columns to time instants.

However, `valueTable` gives the value function for time instants $t = 0, 1, \ldots, T$, whereas `actionTable` has one less column and gives the optimal ordering decisions for time instants $t = 0, 1, \ldots, T - 1$.

The following MATLAB snapshot replicates Example 1.3.2 of Bertsekas, Vol. 1:

```
>> probs = [0.1; 0.7; 0.2];
>> maxInv = 2;
>> [valueTable, actionTable] = MakePolicy(maxInv, probs, 1, 1, 3);
>> valueTable
valueTable =
    3.7000    2.5000    1.3000         0
    2.7000    1.5000    0.3000         0
    2.8180    1.6800    1.1000         0
>> actionTable
actionTable =
     1     1     1
     0     0     0
     0     0     0
```

Demand can take values $0, 1, 2$ with probabilities $0.1, 0.7, 0.2$, respectively, and there is an upper bound 2 on inventory. The two cost coefficients are $c = 1$ and $\beta = 1$. We have to make an ordering decision at times $t = 0, 1, 2$.

The policy is, in a sense, just-in-time and looks intuitive. Since the most likely demand value is 1 and the inventory penalty is symmetric, it is optimal to order one item when inventory is empty, so that the most likely on-hand inventory after meeting demand is zero. When we hold some inventory, we should not order anything.

We may simulate the application of the decision policy for each level of the initial state:

```
function costScenarios = SimulatePolicy(actionTable, demandProbs, ...
    orderCost, invPenalty, horizon, numScenarios, startState)

pd = makedist('Multinomial','probabilities',demandProbs);
demandScenarios = random(pd,numScenarios,horizon)-1;
costScenarios = zeros(numScenarios,1);
for k = 1:numScenarios
    state = startState;
    cost = 0;
    for t = 1:horizon
       % below, we add 1, since MATLAB indexing starts from 1, not 0
       order = actionTable(state+1, t);
       cost = cost + orderCost*order + ...
           invPenalty*(state + order - demandScenarios(k,t))^2;
       state = max(0, state + order - demandScenarios(k,t));
    end
    costScenarios(k) = cost;
end  % for
```

We generate a random sample of demand scenarios, collected in `demandScenarios`, as a `Multinomial` distribution; we have to subtract 1, since in MATLAB a multinomial distribution has support starting from 1, rather than 0.

The `for` loop simulates, for each scenario, the application of the optimal actions, the collection of immediate costs, and the updates of the state variable.

The following snapshot shows the good agreement between the exact value function and its statistical estimate by simulation:

```
>> rng('default')
>> numScenarios = 1000;
>> costScenarios = SimulatePolicy(actionTable,probs,1,1,3,numScenarios,0);
>> cost0 = mean(costScenarios);
>> costScenarios = SimulatePolicy(actionTable,probs,1,1,3,numScenarios,1);
>> cost1 = mean(costScenarios);
>> costScenarios = SimulatePolicy(actionTable,probs,1,1,3,numScenarios,2);
>> cost2 = mean(costScenarios);
>> [valueTable(:,1), [cost0;cost1;cost2]]
ans =
    3.7000    3.7080
    2.7000    2.6920
    2.8180    2.8430
```

In the toy lot-sizing example, we easily store value and policy functions into a table, but this is inefficient when the number of possible demand values is huge. It clearly become impossible when the state space is continuous.

However, sometimes can drastically simplify the problem by taking advantage of its structure.

Let us consider a simple version of deterministic lot-sizing, where we only deal with fixed ordering charges ϕ and inventory holding cost h and, without loss of generality, we assume that both initial and terminal inventories are zero.

Let us also assume, for the sake of convenience, that demand is nonzero in the first time period (otherwise, we just shift time forward).

In principle, we may solve the problem by the following DP recursion:

$$V_t(I_t) = \min_{x_t \geq d_{t+1} - I_t} \left\{ \phi \cdot \delta(x_t) + h\big(I_t + x_t - d_{t+1}\big) + V_{t+1}\big(I_{t+1}\big) \right\}, \quad t = 0, \ldots, T-1,$$

with boundary condition $V_T(I_T) \equiv 0$.

The constraint on the ordered amount x_t makes sure that demand is always satisfied and that the state variable never gets negative.

Theorem: Wagner–Whitin property. For the uncapacitated and deterministic lot-sizing problem, with fixed charges and linear inventory costs, there exists an optimal solution where the following complementarity condition holds:

$$I_t x_t = 0, \qquad t = 0, 1, \ldots, T-1.$$

The message is that it is never optimal to order, unless inventory is empty.

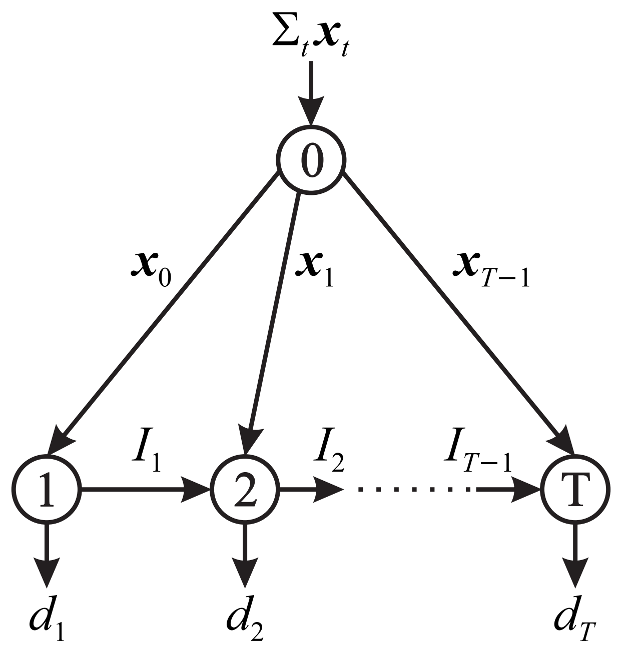
To see this, let us consider a network flow representation shown.

We observe that, for the overall network, the global flow balance

$$\sum_{t=0}^{T-1} x_t = \sum_{t=1}^{T} d_t$$

must hold.

This equilibrium condition is expressed by introducing the dummy node 0, whose inflow is the total ordered amount over the planning horizon.



We also have to make sure that demand is met during each time interval. To this aim, we introduce a set of nodes corresponding to time instants $t = 1, \ldots, T$. Note that we associate these nodes with the last time *instant* of each time interval, which essentially amounts to saying that we may satisfy demand at the end of the time interval.

Flow balance at node t corresponds to the state transition equation

$$I_t = I_{t-1} + x_{t-1} - d_t.$$

For this node, let us assume that, contrary to the above theorem, both $I_{t-1} > 0$ and $x_{t-1} > 0$.

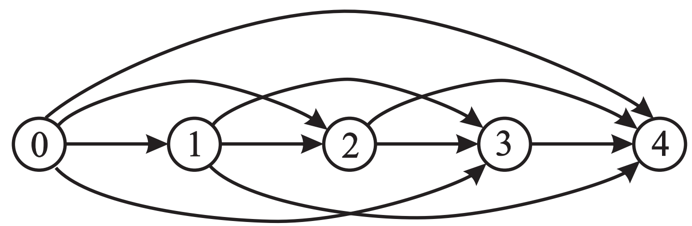
Then, it is easy to see that by redirecting the horizontal inflow I_{t-1} along the ordering arc corresponding to x_{t-1}, we may improve the overall cost.

The practical consequence of the Wagner–Whitin condition is that, at time instant t, we should only consider the following ordering possibilities:

$$x_t \in \left\{ 0,\ d_{t+1},\ (d_{t+1} + d_{t+2}),\ (d_{t+1} + d_{t+2} + d_{t+3}),\ \dots\ ,\ \sum_{\tau=t+1}^{T} d_\tau \right\}.$$

Given this property, we can reformulate the single-item problem as a shortest path on a fairly small network.

The initial node 0 represents the initial state. For each time instant t, we have a set of arcs linking it to time instants $t+1, \dots, T$, and we must move from the initial node to the terminal one, along the minimum cost path.



The selected arcs correspond to the number of time intervals that we cover with the next order. For instance, a path

$$0 \to 2 \to 3 \to 4$$

corresponds to the ordering decisions

$$\begin{aligned} x_0 &= d_1 + d_2 \\ x_1 &= 0 \\ x_2 &= d_3 \\ x_3 &= d_4. \end{aligned}$$

Each arc cost is computed by accounting for the fixed ordering charge and the resulting inventory holding cost. For instance, the cost of the arcs emanating from node 0 are:

$$\begin{aligned} c_{0,1} &= \phi, \\ c_{0,2} &= \phi + hd_2, \\ c_{0,3} &= \phi + hd_2 + 2hd_3, \\ c_{0,4} &= \phi + hd_2 + 2hd_3 + 3hd_4. \end{aligned}$$

Due to limited number of nodes in this network, by solving the lot-sizing problem as this kind shortest path problem we obtain a very efficient (polynomial complexity) algorithm.

The Wagner–Whitin condition does not apply to stochastic lot-sizing but, in some cases, we may find useful structural results.

Let us assume that customers are patient and willing to wait, so that there is no lost sales penalty, and that the total cost function includes a term like

$$q(s) = h \max\{0, s\} + b \max\{0, -s\},$$

where the inventory level s may be positive (on-hand inventory) or negative (backlog); h is the usual inventory holding cost, and $b > h$ is a backlog cost.

Note that $q(\cdot)$ is a convex penalty and goes to $+\infty$ when $s \to \pm\infty$.

For now, we disregard fixed charges, but we also include a linear variable cost, with unit ordering cost c.

Hence, the overall problem requires to find a policy minimizing the expected total cost over T time periods:

$$\mathsf{E}_0 \left[\sum_{t=0}^{T-1} \Big\{ c x_t + q\big(I_t + x_t - d_{t+1}\big) \Big\} \right],$$

where x_t is the amount ordered and immediately received at time t, as before.

We may write the DP recursion as

$$V_t(I_t) = \min_{x_t \geq 0} \left\{ cx_t + H(I_t + x_t) + \mathrm{E}\left[V_{t+1}(I_t + x_t - d_{t+1})\right] \right\}$$

where we define

$$H(y_t) \doteq \mathrm{E}\Big[q\big(y_t - d_{t+1}\big)\Big] = h\,\mathrm{E}\big[\max\{0, y_t - d_{t+1}\}\big] + b\,\mathrm{E}\big[\max\{0, d_{t+1} - y_t\}\big].$$

Here, y_t is the available inventory *after* ordering and immediate delivery, as we assume zero lead time: $y_t \doteq I_t + x_t$. The terminal condition is $V_T(I_T) = 0$.

We also assume that the probability distribution of demand is constant.

The introduction of inventory after ordering allows for a convenient rewriting of the DP recursion as

$$V_t(I_t) = \min_{y_t \geq I_t} G_t(y_t) - cI_t,$$

where

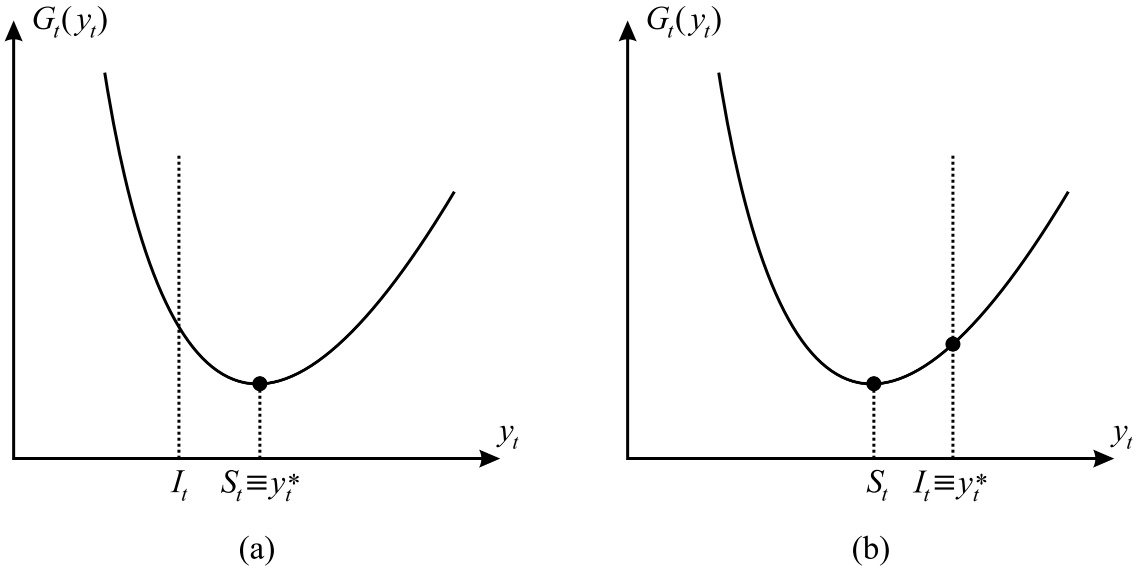
$$G_t(y_t) = cy_t + H(y_t) + \mathrm{E}\big[V_{t+1}(y_t - d_{t+1})\big].$$

Now, we claim, without proof, that $V_t(\cdot)$ and $G_t(\cdot)$ are convex for every t, and that the latter goes to $+\infty$ when $y \to \pm\infty$, which is essentially a consequence of the shape of the penalty function $q(\cdot)$.

The implication of the properties of $G_t(\cdot)$ is that it has a finite unconstrained minimizer,

$$S_t = \arg\min_{y_t \in \mathbb{R}} G_t\big(y_t\big).$$

The following figure illustrates the relative positioning of unconstrained and constrained minima in a stochastic lot-sizing problem.



- Case (a): S_t satisfies the constraint ($S_t \geq I_t$) and $S_t \equiv y_t^*$, i.e., the unconstrained and constrained minimizers coincide.

- Case (b): S_t does not satisfy the constraint ($S_t < I_t$) and $I_t \equiv y_t^*$, i.e., the constrained minimizer is located on the boundary of the feasible set.

The optimal policy is given by a *base-stock* (or order-up-to) policy:

$$x_t^* = \mu_t^*(I_t) = \begin{cases} S_t - I_t, & \text{if } I_t < S_t, \\ 0, & \text{if } I_t \geq S_t. \end{cases}$$

The amounts S_t can be regarded as target inventory levels: we should order what we need to reach the optimal target at each time instant.

All we have to do, in a finite horizon problem is finding the optimal sequence of target inventory levels S_t.

If we include fixed ordering charges, we lose convexity. However, a related property (K-convexity) can be proved, leading to the following optimal policy

$$\mu_t^*(I_t) = \begin{cases} S_t - I_t, & \text{if } I_t < s_t, \\ 0, & \text{if } I_t \geq s_t, \end{cases}$$

depending on two sequences of parameters s_t and S_t, where $s_t \leq S_t$. In a stationary environment, we find that a stationary (s, S) policy is optimal.

We should order only when inventory is less than a critical amount called the small s, in which case we bring the level back to the big S. Note that $S - s$ is a minimum order quantity, which keeps fixed ordering charges under control.

DP is a powerful and flexible principle, but it does have some important limitations.

- The **curse of state dimensionality**. We need the value function for each element in the state space. If this is finite and not too large, value functions can be stored in tabular form, but this will not be feasible for huge state spaces.

- The **curse of optimization**. We use DP to decompose an intractable multistage problem into a sequence of single-stage subproblems. However, even the single-stage problems may be quite hard to solve.

- The **curse of expectation**. If the risk factors $\boldsymbol{\xi}_t$ are represented by continuous random variables, the expectation requires the computation of a difficult multidimensional integral; hence, some discretization strategy must be applied.

- The **curse of modeling**. The system itself may be so complex that it is impossible to find an explicit model of state transitions. The matter is more complicated in the DP case, since transitions are at least partially influenced by control decisions.