An Introduction to Numerical Optimization Methods and Dynamic Programming using C++

Toke Ward Petersen[*]

June 5th 2001

Abstract

This appendix contains a short introduction to numerical methods and describes how to solve …nite horizon dynamic programming in C++.

[*]Statistics Denmark and University of Copenhagen. E-mail: toke.ward.petersen@econ.ku.dk.

Contents

1 Introduction 3

2 Optimization methods 3
- 2.1 Analytic methods 4
- 2.2 Gradient based numeric methods 5
  - 2.2.1 Initial phase 6
  - 2.2.2 Find the "descent direction" 7
  - 2.2.3 Line search: successive step-size increase 7
  - 2.2.4 Second iteration 8
  - 2.2.5 Advantages and disadvantages of the gradient based method 10
- 2.3 Discrete numeric methods 10
- 2.4 Summary 12

3 Deterministic dynamic programming 13
- 3.1 The basic savings problem 13
  - 3.1.1 Step one: identify state and control variables 13
  - 3.1.2 Step 2: Solve the last period 14
  - 3.1.3 Step 3: The second-last period 15
  - 3.1.4 Step 4: The period before the second last period (here the …rst period) 17
  - 3.1.5 Results 19

4 An implementation of the savings problem in C++ 20
- 4.1 Initialize program 21
- 4.2 Initialize grid 22
- 4.3 Calculate the last period 22
- 4.4 Calculate all other periods recursively in a loop 23
- 4.5 Display results 24
- 4.6 Output from the program 25
- 4.7 Acceleration 26
- 4.8 Addendum: source code for the deterministic program 27

1 Introduction

This appendix contains a short introduction to numerical methods and describes how to solve …nite horizon dynamic programming in C++. The purpose is to give an idea of how Dynamic Programming is carried out in practice; hopefully this will equip the reader to understand the way more complicated models are solved.

The …rst section describes various optimization methods used in applied economics, with particular focus on the grid-optimization techniques used when doing dynamic programming. The second section shows the practical work involved in solving a simple dynamic programming problem. The problem is very simple and could in principle be solved in a spread-sheet - however the purpose is to explain the technique, and for this purpose a simple example is better. The last section shows how the described method for solving a simple deterministic model can be implemented in C++.

2 Optimization methods

The purpose of this section is to present the various approaches to optimization that can be used in applied work. We will consider 3 methods of obtaining the solution to the optimization problem: by i) analytic means ii) using gradient based numerical methods and iii) using discrete optimization techniques. The latter method will be employed later when doing numerical dynamic programming.

To keep the problem simple and keep the focus on the techniques, we look at the problem of …nding a solution to the maximization problem:

$$\max f(x) = -\frac{1}{4}x^2 + 8x + 25$$

which for the interval $[0; 30]$ is shown in Figure 1 below:

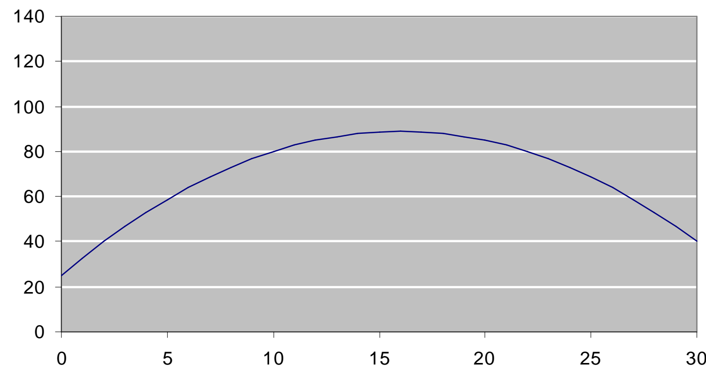

Figure 1. The polynomial $-\frac{1}{4}x^2 + 8x + 25$.

As the ...gure shows the polynomial has one unique maximum. Clearly the methods described below work with higher dimensional problems, but the very simple maximization problem above serves a useful point of departure when demonstrating the method.

2.1 Analytic methods

The analytic approach to solving the problem uses the problem's ...rst order condition. Di¤erentiation yields:

$$\begin{aligned} 0 &= \frac{d}{dx}\left(-\frac{1}{4}x^2 + 8x + 25\right) \\ 0 &= -\frac{1}{2}x + 8 \\ x &= 16 \end{aligned} \tag{1}$$

Clearly this is a very fast way to ...nd the solution in the example above. Unfortunately it requires that the system of equations of ...rst order conditions $\nabla f = 0$ (here it is just one equation with one unknown) can be solved; this is not always an easy task.

Obtaining an analytic solution can often require large amounts of calculations and algebraic manipulation. However, often the analytic solution in itself has no interest - it is only used as an tool to calculate the solution for speci...c numeric values. From an overall perspective it may therefore be more e¢cient to calculate the solution using numerical methods right away.

The analytic method is a direct method in the sense that we get the correct solution right away using the formula above. The numeric methods presented below are all iterative: they do not get the correct solution immediately, but rather start with one solution and improve it a bit at a time. This is a procedure that is repeated a number of times until the found solution converges.

Numeric methods can be divided in two groups depending on how the search procedure determines where to search for the solution. The …rst group relies on information about the gradient to determine the descent[1] direction (i.e. use numeric computed derivatives for the search procedure), whereas the second group uses other methods to …nd the direction in which the objective function improves. In the next subsection we will look at each of these methods in turn.

2.2 Gradient based numeric methods

A gradient based (or gradient related) method proceeds in the following way:

1. Choose some initial value to get the search procedure started. If a good guess is available this should be used.

2. Determine the direction that improves the objective function by using information about the derivative in the initial point. Compute the …rst derivative of f in this point (in more advanced methods the second derivative is also computed or approximated somehow to capture the curvature of the function), and prepare to go in this direction to look for the optima.

3. Determine how far to move in this direction - exactly how long is determined by a so-called step-size rule. Check if the objective function has increased in this new point - otherwise choose a smaller step-size until a reasonable improvement has been

[1] It is customary in the numerical analysis literature to consider the problem of (cost) minimization instead of maximization. In this case the term descent direction refers to the "downhill" direction in a minimization problem. Here the term is used to refer to the direction in which the objective function gets "better" (here: increases, but in the classic case of minimization decreases) no matter if we analyze a maximization or a minimization problem.

made. If on the other hand the objective function increases then proceed further in this direction until further steps cease to increase the objective value.

4. Use this new found point as initial value in the next iteration. Go to step 2.
5. If there is no improvement (or the step-size is below a certain threshold) stop the iterations: then the local optima may have been found.

This method can be improved in many directions - however this a topic for a separate volume, and the interested reader should consult ?. Notice that while the outline above captures the main points it also leaves out many details.

Next is a demonstration of how the method works on the problem outlined above - using a particularly simple type of gradient related method called the steepest descent method. The idea in the steepest descent method is to move in the direction where the objective function changes the fastest (= the gradient is the steepest). In general this is not the most e¢cient method in terms of the number of iterations required to arrive at the solution[2], but the method is very easy to explain. For the line-search and step-size rule mentioned in point 3 above we will use a geometric step-size rule (explained below).

2.2.1 Initial phase

First we must choose an initial value for the search. Here we choose $x^0 = 5$ as a candidate (the superscript refers to the iteration number). With this particular function the choice of initial value does not matter, but in general some degree of "luck" is necessary when choosing an initial value.

[2] Although in this one-dimensional case the method's weakness is not apparent. In two dimensions however, it is quite easy to demonstrate why steepest descent can require an excessive number of iterations. See ?

2.2.2 Find the "descent direction"

Next we must determine the numeric derivative in $f(x^0)$. Here we use the central approximation (?) and compute $f'(x^0)$ as

$$f'\left(x^0\right) = \frac{f(x^0+\delta) - f(x^0-\delta)}{2\delta} \tag{2}$$

which for $\delta = 0.1$ gives $f'(x^0) = 5.5$. The direction of the gradient is illustrated with the dotted line in Figure 2 below:

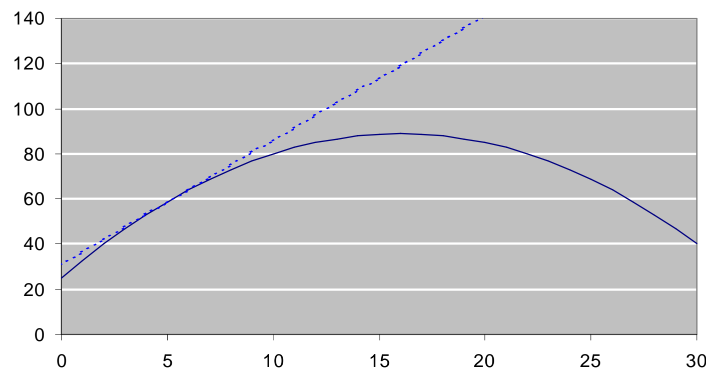

Figure 2. Initial point $x^0 = 5$ at first iteration (gradient in dotted line).

In this simple one-dimensional case the direction is also one-dimensional, and the steepest descent direction is either to the left or to the right. Here the criteria function increases when we move to the right.

2.2.3 Line search: successive step-size increase

Now we know the direction to search (to the right of the initial point, since the derivative is positive), but not how far in this direction it is optimal to move - this problem is called line search. Here we will use a simple geometric series rule (this is a variant of the diminishing step-size rule) where the initial step length is given by φ. Then try a longer (or shorter) step-size depending on whether the function evaluated in the initial candidate point improves the objective value (or not) compared to the initial point. Here $\varphi = 5$ which means that the first candidate iterate $x^1 = x^0 + \varphi = 10$.

Next we must check whether this causes an increase in the criteria function, i.e. whether $f(x^1) > f(x^0)$ which is the case since $f(x^1) = 80{:}00$ and $f(x^0) = 58{:}75$: Since the step caused an increase in the criteria value we then try to proceed in the same direction (to the right) with a larger step length than 5 - a step length that increases for each iterate candidate. For the n'th step the next iterate candidate is found using the formula:

$$x^1_+ = x^0 + Á\left(¼^{n ¡ 1}\right)$$

where ¼ = 1:25: This gives the following series of step lengths and iterates:

n	1	2	3	4	5	6
$¼^n$	1	1.250	1.5625	1.9531	2.4414	3.0518
$Á(¼^n)$ [step length]	5	6.250	7.8125	9.7656	12.2070	15.2588
$x^0 + Á(¼^n)$ [iterate candidate]	10	11.250	12.8125	14.7656	17.2070	20.2588
$f(x^0 + Á(¼^n))$ [objective]	80.00	83.3594	86.4600	88.6191	88.6358	84.4657
marginal improvement	21.2500	3.3594	3.1006	2.1591	0.0167	-4.1701

The table shows that the marginal improvement in the criteria function becomes negative after iteration number 5 (at $x^5 = 17{:}207$), which means that our line search should stop here, and the next iterate should be $x^1 = 17{:}207$. The various trial iterates are shown with a black box in the …gure below:

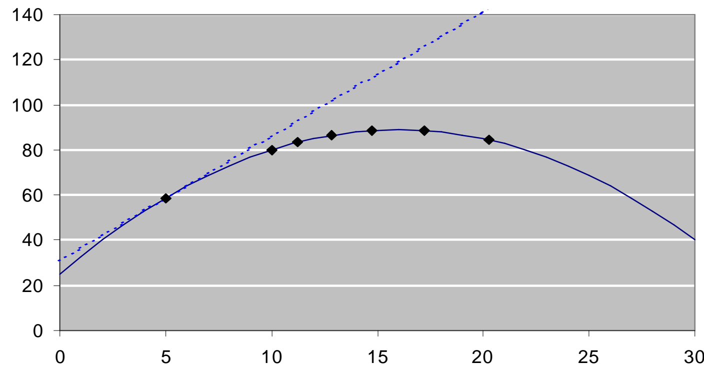

Figure 3. Iterates at the …rst iteration (gradient in dotted line).

2.2.4 Second iteration

The second iteration uses $x^1 = 17{:}207$ as the initial point. Once again we compute the derivative using equation (2), which with ± = 0:1 gives $f^0(x^1) = ¡\,0{:}6035$; which is shown

in Figure 4 below.

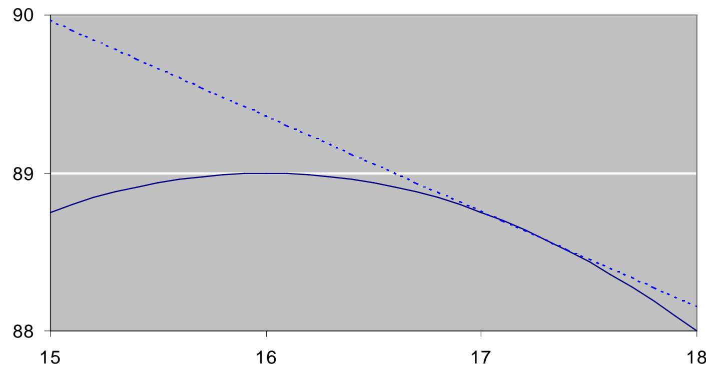

Figure 4. Initial point $x^1 = 17.207$ at second iteration (gradient in dotted line).

Having determined the direction to move (left since the derivative is negative) we must determine the step length. As before we use an initial step-size of 5, but since we move in the opposite direction then $\lambda = -5$. First we try the full step (length 5) which implies $x^2 = 17.207 - 5 = 12.207$ with $f(x^2) = 85.4033$. Compared to $f(x^1) = 88.6358$ this is a decrease in the objective function, and therefore we start by decreasing the step size until an improvement is found. This is done using the iterate candicate rule:

$$x_i^1 = x^0 + \lambda \left(\rho^{(n-1)} \right)$$

which implies the following sequence of step lengths and iterates:

n	1	2	3	4	5	6	7	8
$\rho^{(n-1)}$	1	0.8	0,6400	0,5120	0,4096	0,3277	0,2621	0,2097
$\lambda \rho^{(n-1)}$ [step length]	5	4	3.2	2.5600	2.0480	1.6384	1.3107	1.0486
x^n [iterate candicate]	12.207	13.207	14.007	14.647	15.159	15.569	15.896	16.158
$f(x^n)$ [objective]	85.403	87.050	88.007	88.542	88.823	88.9545	88.997	88.994
marg. improvement	-3.233	1.647	0.957	0.535	0.281	0.130	0.049	-0.004

From the table we see that the improvement in the criteria function stops after the 7th step, and therefore $x^2 = 15.896$ with $f(x^2) = 88.9973$.

We could continue this process a number of time more, and would end up getting iterates that are converging to the true solution. In the table below the series of iterates using the method is listed:

n	x^n	$f(x^n)$	$\lvert x^n - x^{n-1} \rvert$	$\lvert f(x^n) - f(x^{n-1}) \rvert$
0	5.0000	58.75	$-$	$-$
1	17.2070	88.6358	12.270	30.8858
2	15.8963	88.9973	1.3107	0.36150
3	16.00886999	88.99998	0.11257	0.002480

As the table shows convergence is achieved reasonably quickly in the case considered here (the true value is $x^* = 16$ and $f(x^*) = 89$). We can measure the convergence of the iterates by the metric $\lvert x^n - x^{n-1} \rvert$ or the convergence on the function value $\lvert f(x^n) - f(x^{n-1}) \rvert$.

2.2.5 Advantages and disadvantages of the gradient based method

The success of the gradient based methods depend on many factors, but of particular importance is how well-behaved the function we seek to maximize is: the more well-behaved the better the results. A second problem, is the requirement that the function must be once (or twice) continuously differentiable. Depending on the economic problem at hand this condition may be impossible to meet - in particular if one wants to analyze more realistic problems with kinked budget sets or discrete choice sets. A more practical problem is that the method requires a lot of programming finesse: for the solution procedure to work properly, it is often not enough to implement a basic version of Newton's method from a numerical analysis book such as ?. For the routine to work properly there are many details and special cases during the solution, that need to be taken care of.

2.3 Discrete numeric methods

The second numeric approach to finding a solution is a grid search procedure - this method does not rely on gradient information to determine which direction the search should proceed. Rather the method relies on bracketing the solution in a interval that decrease in length for each iteration - the classic one-dimensional example is the Golden Section method (?, page 605). This class of methods may be slow and inefficient, but will arrive at a solution no matter how ill-behaved the problem at hand may be (but if the problem is very ill-behaved however, the solution found may not be the global optima). In return

the method requires more function evaluations to arrive at a solution.

Discrete optimization techniques have less mathematical ...nesse than the gradient based procedures, and rely more on heuristics. The method that will be presented here is particularly simple: grid optimization. The idea is to evaluate the function in a number of points in the initial interval and determine an interval in which the optima is believed to be located (if this interval is shorter than the initial interval we have made progress, and are closer to the solution). Afterwards this procedure is repeated a number of times until the interval in which the solution is located is smaller than some predetermined tolerance.

To illustrate the method we will ...rst need a lower and upper bound for the search. Here we pick the interval -100 to 100 and divide it into 11 equidistant grid points, and evaluate the function in each point:

x	-50	-40	-30	-20	-10	0	10	20	30	40	50
f (x)	-1000	-695	-440	-235	-80	25	80	85	40	-55	-200

The table shows that the largest function value on the grid is x = 20; but we do not know if the optima is in the interval [10;20] or [20;30]. Therefore we must next repeat the procedure on the interval [10;30] and get:

x	10	12	14	16	18	20	22	24	26	28	30
f (x)	80	85	88	89	88	85	80	73	64	53	40

after which it appears that the optimum is located between 14 and 18. Once again we re...ne the grid to the interval [14;18] and get:

x	14.0	14.4	14.8	15.2	15.6	16.0	16.4	16.8	17.2	17.6	18.0
f (x)	88	88.36	88.64	88.84	88.96	89	88.96	88.84	88.64	88.36	88

This time the maximum is bracketed between 15.6 and 16.4, and further iterations would decrease this interval. In this case it is a coincidence that the optimal value happens to be on the grid.

This procedure of successively re…ning the interval containing the solution continues for a couple of iterations more, and information about each iteration is shown in the table below:

iteration	min	max	mean	interval length	f(mean)
0	-50	50	0	10	25
1	10	30	20	2	85
2	14	18	16	0.4	89
3	15.6	16.4	16	0.08	89
4	15.92	16.08	16	0.016	89
5	15.984	16.016	16	0.0032	89

With the number of grid points chosen above, the interval that contains the solution decreases to $\frac{1}{5}$ for each iteration. Thus we have a fairly good approximation after 5 iterations, where the solution is 16 § 0:0032 which is equivalent to 16 § 0:02%.

The method can be re…ned in a number of ways. Clearly the number of grid points in‡uences the solution time in a linear fashion, which suggests that the fewer the number of grid points the better. However too few grid points increases the likelihood that the grid-points may miss the true global optima, and instead …nd a local optima. This kind of problem can of cause occur at any grid density, but is more likely to happen with few grid points (in particular if the function is ill-behaved). This leads to a discussion of "how few is few" and what is a suitable number of grid points. This is where heuristics and experience comes in: these parameters depend on the problem at hand and how well-behaved it is, and these parameters need to be set by experimentation.

2.4 Summary

The three methods presented above have their strengths and weaknesses. For the purpose of doing dynamic programming of realistic economic models, the requirement that functions must remain di¤erentiable, is di¢cult to meet in practise. The only method that allows this, is the discrete optimization method, that also for other reasons turns out to be useful when doing dynamic programming.

3 Deterministic dynamic programming

This chapter describes the practical aspects of the deterministic dynamic programming method - with the emphasis on applied, leaving the rigorous mathematical technicalities to ?. As the section's name suggests, this description is meant for beginners without any knowledge of the practical workings of dynamic programming.

3.1 The basic savings problem

The simplest application of dynamic programming is the classical savings problem: an agent endowed with some initial amount of money lives for several periods and must decide when to use his money. For simplicity we assume that the agent lives for 3 periods. The agent's only decision is how much to consume in each of the three periods, which is equivalent to the problem of how much to save between periods (since the non-negativity constraint on asset holdings means that forgone consumption equals savings).

To be more specific, the utility function of the consumer is to maximize the time-separable utility function:

$$U(c_1; c_2; c_3) = \sum_{i=\{1;2;3\}} \beta^{i-1}\sqrt{c_i} \tag{3}$$

where $\beta = 0.9$ is the one period discount factor. In addition the consumer faces the budget constraint

$$M = c_1 + c_2 + c_3 \tag{4}$$

where $M > 0$ is the agent's initial income, as well as the non-negativity constraints $c_i > 0$ $(\forall i)$. Notice that for simplicity the interest rate on savings is zero.

3.1.1 Step one: identify state and control variables

Clearly a closed form solution can be found to this problem using the Lagrange method, but we will proceed with numerical dynamic programming. The first step is to identify which are the state and control variables of the problem. Here assets is an obvious candidate for a state variable: at any point in time the utility from the remaining periods

of life depends on the assets; it does not depend on past or present consumption. There are two possible choices of control variable: either end-of-period assets (which together with the start-of-periods assets implicitly gives the size of the consumption) or the consumption (which together with the start-of-period assets implicitly gives the end-of-period assets). The solution is the same no matter what method is used. However from a practical perspective choosing end-of-period assets as the control variable makes it easier to solve the model. Thus with the state variable being the beginning-of-period assets and the control variable being the end-of-period assets, the consumption in the period can be calculated as the difference between the two. The transition for the state variable is also quite simple, since the previous period's control variable (end-of-period assets) becomes next period's state variable.

Before we can solve the problem using dynamic programming the problem must be formulated recursively. We can rewrite the problem using a recursive formulation:

$$\begin{aligned} V_i(M_{i-1}) &= \max_{c_i;M_i} [\sqrt{c_i} + \beta V_{i+1}(M_i)] \\ st &: M_i = M_{i-1} - c_i \end{aligned} \tag{5}$$

Notice that the end-of-period notation is used for assets: thus the beginning of period assets in the first period (V_1) are denoted M_0. Since there are only 3 periods we can set $V_4 = 0$.

3.1.2 Step 2: Solve the last period

Next we start by considering the last period: consider the agent who reaches the last period with some beginning-of-period assets M_2. Assets left behind yields no utility, and therefore the optimal decision for the agent in the last period is to consume everything - an optimality condition that is not surprising.

Next we compute $V_3(M_2)$ which is the utility level for an agent in the last period with the beginning-of-period assets M_2. This value is computed for selected values of $M_2 \in [0; 5]$; and this interval is divided into 6 equidistant nodes.

Since we know that the optimal decision for the agent in the last period is to consume

everything, we have that

$$V_3(M_2) = \sqrt{M_2} \qquad (6)$$

since as noted $M_2 = C_3$. Plotting these values in a table is quite straight-forward:

Optimal solutions in the last period:

M_2	0	1	2	3	4	5
M_3	0	0	0	0	0	0
C_3	0	1	2	3	4	5
$\sqrt{C_3}$	0	1	1.414	1.732	2	2.236
$V_3(\cdot)$	0	1	1.414	1.732	2	2.236

(7)

Thus we have the choices that maximize $V_3(M_2)$ for various values of M_2 (here sampled in 6 points). In other words, we have the optimal actions for a consumer in his last period, given his asset holdings when the period begins. With this solution to the last period problem, we can continue to the second last period.

3.1.3 Step 3: The second-last period

Having solved the problem for the last period for given values of beginning-of-period assets M_2, we have knowledge about $V_3(M_2)$ as well as the associated optimal choices for the last period. Armed with this knowledge, we begin solving the problem of the second last period. For this period the problem is:

$$\begin{aligned} V_2(M_1) &= \max_{c_2, M_2} [\sqrt{c_2} + \beta V_3(M_2)] \\ st &: M_2 = M_1 - c_2 \end{aligned} \qquad (8)$$

Again we take an numeric approach to solving the problem: we discritize M_1 into 6 grid points as before and compute the utility maximizing optimal choices for these points. As there is one state variable (M_1) and one control variable (M_2) we need to search through all combinations of M_1 and M_2 that give a feasible (=non-negative) value of consumption, and for each M_1 to determine the optimal policy.

We start by computing the consumption level implicitly associated with the choice of M_2 for a given level of M_1:

consumption	$M_2 = 0$	$M_2 = 1$	$M_2 = 2$	$M_2 = 3$	$M_2 = 4$	$M_2 = 5$
$M_1 = 0$	$C_2 = 0$	infes.	infes.	infes.	infes.	infes.
$M_1 = 1$	$C_2 = 1$	$C_2 = 0$	infes.	infes.	infes.	infes.
$M_1 = 2$	$C_2 = 2$	$C_2 = 1$	$C_2 = 0$	infes.	infes.	infes.
$M_1 = 3$	$C_2 = 3$	$C_2 = 2$	$C_2 = 1$	$C_2 = 0$	infes.	infes.
$M_1 = 4$	$C_2 = 4$	$C_2 = 3$	$C_2 = 2$	$C_2 = 1$	$C_2 = 0$	infes.
$M_1 = 5$	$C_2 = 5$	$C_2 = 4$	$C_2 = 3$	$C_2 = 2$	$C_2 = 1$	$C_2 = 0$

(9)

Notice that the …elds above the diagonal are infeasible: since there is no income it is not possible to choose end-of-period assets that are higher than the beginning-of-period assets.

This matrix of present consumptions has the associated present period utility levels (given by the transformation $U(x) = \sqrt{x}$):

present utility	$M_2 = 0$	$M_2 = 1$	$M_2 = 2$	$M_2 = 3$	$M_2 = 4$	$M_2 = 5$
$M_1 = 0$	0	infes.	infes.	infes.	infes.	infes.
$M_1 = 1$	1	0	infes.	infes.	infes.	infes.
$M_1 = 2$	1:414	1	0	infes.	infes.	infes.
$M_1 = 3$	1:732	1:414	1	0	infes.	infes.
$M_1 = 4$	2	1:732	1:414	1	0	infes.
$M_1 = 5$	2:236	2	1:732	1:414	1	0

(10)

and with tomorrow's utility given by the term $\beta V_3(M_2)$ given by β times the value $V_3(M_2)$ taken from (6). This gives the following values for $V_2(M_1)$:

$V_2(M_1)$	$M_2 = 0$	$M_2 = 1$	$M_2 = 2$	$M_2 = 3$	$M_2 = 4$	$M_2 = 5$
$M_1 = 0$	0	-	-	-	-	-
$M_1 = 1$	1.000	0.900	-	-	-	-
$M_1 = 2$	1.414	1.900	1.273	-	-	-
$M_1 = 3$	1.732	2.314	2.273	1.559	-	-
$M_1 = 4$	2.000	2.632	2.687	2.559	1.800	-
$M_1 = 5$	2.236	2.900	3.005	2.973	2.800	2.012

(11)

The table above should be read the following way: given the beginning-of-period assets M_1 (given by the row) the agent can choose between a number of end-of-periods assets M_2 (given by the column) that implicitly de…nes the consumption in the period as well as

future utility from the choice of end-of-period assets. This value is given by equation (8) as the sum of the utility of present consumption and discounted value of utility from the end-of-period asset holdings. For example the value of 2.632 in row $M_1 = 4$ and $M_2 = 1$ is the sum of the utility of consuming $4 - 1 = 3$ units now ($= \sqrt{3} = 1.732$) plus the value of $V_3(M_2 = 1) = 1$ discounted by $\beta = 0.9$, i.e. $1.732 + 0.900 = 2.632$. Another example is the row $M_1 = 1$: here the choice is between consuming the unit now (and gaining the utility $\sqrt{1}$) or saving the unit for one period and then consuming it (which also would give the $\sqrt{1}$ in utility but discounted one period, i.e. a total of 0.9). Not surprisingly the utility from consuming the unit now is higher.

The problem for the consumer is now for given beginning-of period assets (M_1) to choose the end-of-period assets (M_2) to achieve the maximum utility: in other words to choose the set of control variables that maximize the value in each row of table (11). In the table these optimal choices are shown in a box, and the optimal policy response to the various initial values are:

Optimal solutions in the second-last period:

M_1	0	1	2	3	4	5
M_2	0	0	1	1	2	2
C_2	0	1	1	2	2	3
$\sqrt{C_2}$	0	1	1	1.414	1.414	1.732
$V_2(\cdot)$	0	1	1.900	2.314	2.687	3.005

(12)

This completes the computations for period 2; now we know the utility levels and the optimal policy choices for given values of initial assets (in the table above)

3.1.4 Step 4: The period before the second last period (here the first period)

Having solved the second-last period the next step is to solve the period before the second-last period - which in the present 3-period model happens to be the first period. However the techniques employed would have been the same whether one solves a model with 3 or 60 periods.

With the new knowledge of $V_2(M_1)$ the next step is to solve the problem $V_1(M_0)$ - the

problem faced by the consumer in the initial period. Similar to before the problem for the consumer is:

$$\begin{aligned} V_1(M_0) &= \max_{c_1;M_1} [\sqrt{c_1} + \beta V_2(M_1)] \\ st &: M_1 = M_0 - c_1 \end{aligned} \quad (13)$$

where we again discritize M_1 into 6 grid points and …nd the utility maximizing optimal choices. Just as before we must search through all feasible combinations of M_1 and M_2 for the optimal solution.

Once again we start by computing the consumption level implicitly associated with the choice of M_1 for a given M_0:

consumption	$M_1 = 0$	$M_1 = 1$	$M_1 = 2$	$M_1 = 3$	$M_1 = 4$	$M_1 = 5$
$M_0 = 0$	$C_2 = 0$	infes.	infes.	infes.	infes.	infes.
$M_0 = 1$	$C_2 = 1$	$C_2 = 0$	infes.	infes.	infes.	infes.
$M_0 = 2$	$C_2 = 2$	$C_2 = 1$	$C_2 = 0$	infes.	infes.	infes.
$M_0 = 3$	$C_2 = 3$	$C_2 = 2$	$C_2 = 1$	$C_2 = 0$	infes.	infes.
$M_0 = 4$	$C_2 = 4$	$C_2 = 3$	$C_2 = 2$	$C_2 = 1$	$C_2 = 0$	infes.
$M_0 = 5$	$C_2 = 5$	$C_2 = 4$	$C_2 = 3$	$C_2 = 2$	$C_2 = 1$	$C_2 = 0$

(14)

Once again …elds above the diagonal represent infeasible combinations that does not satisfy the budget constraint.

The associated present period utility levels associated with these consumption levels are given by the $\sqrt{\ }$-transformation:

present utility	$M_1 = 0$	$M_1 = 1$	$M_1 = 2$	$M_1 = 3$	$M_1 = 4$	$M_1 = 5$
$M_0 = 0$	0	infes.	infes.	infes.	infes.	infes.
$M_0 = 1$	1	0	infes.	infes.	infes.	infes.
$M_0 = 2$	1:414	1	0	infes.	infes.	infes.
$M_0 = 3$	1:732	1:414	1	0	infes.	infes.
$M_0 = 4$	2	1:732	1:414	1	0	infes.
$M_0 = 5$	2:236	2	1:732	1:414	1	0

(15)

Notice that the values inside the tables above are identical to the tables for the previous period - only the indices change.

Finally we add to the present utility the term $\beta V_2(M_1)$ given by β times the value $V_2(M_1)$ taken from (13). This gives the following values for $V_1(M_0)$:

$V_1(M_0)$	$M_1 = 0$	$M_1 = 1$	$M_1 = 2$	$M_1 = 3$	$M_1 = 4$	$M_1 = 5$
$M_0 = 0$	**0**	¡	¡	¡	¡	¡
$M_0 = 1$	**1:000**	0:900	¡	¡	¡	¡
$M_0 = 2$	1:414	**1:900**	1:710	¡	¡	¡
$M_0 = 3$	1:732	2:314	**2:710**	2:083	¡	¡
$M_0 = 4$	2:000	2:632	**3:124**	3:083	2:418	¡
$M_0 = 5$	2:236	2:900	3:442	**3:497**	3:418	2:704

(16)

where once again the utility maximizing optima per row is shown in bold and in a thin box. This information can be summarized in a table with the optimal choices for given values of initial assets (M_0):

Optimal solutions in the …rst period: (17)

M_0	0	1	2	3	4	5
M_1	0	0	1	2	2	3
C_1	0	1	1	1	2	2
$\sqrt{C_1}$	0	1	1	1	1.414	1.414
$V_1(\cdot)$	0	1.000	1.900	2.710	3.124	3.497

(18)

This …nishes the computations associated with the consumer's problem; now we know the optimal actions for consumer depending on initial assets. Combining the information in (17), (12) and (7) we have the complete solution to the consumer's problem. For instance a consumer starting with initial assets of 5 will consume 2 units and save 3 [we get this information from (17)], consume 2 units in period 2 and save 1 [we get this information from (12)], and …nally in the last period consume 1 [the optimal behavior in the last period can be seen from table (7)].

3.1.5 Results

At last we arrive at the solution to the consumer's problem at the …rst period: the table above (17) shows the optimal choices of consumption and savings in the …rst period for various values of initial assets (M_0). The utility for a consumer with these initial combinations of assets are given by the value function $V_1(M_0)$ and are plotted in the …gure below with black diamonds:

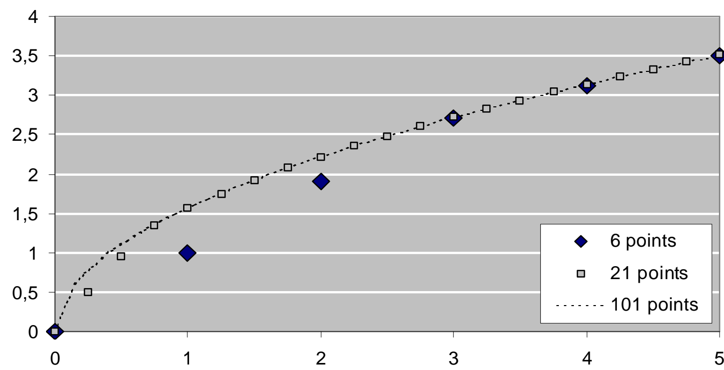

Figure 5: $V_1(M_0)$ for various grid sizes.

The ...gure shows the value function in the ...rst period, $V_1(M_0)$, for various grid resolutions: 6, 21 and 101 mesh points corresponding to distance between mesh-points of 1.00, 0.25 and 0.05. Notice that more points in the grid makes the approximation better. What the ...gure does not tell, is that there is a trade-o¤ between level of detail and speed, since the number of function evaluations requires varies. With 6 mesh points the $u(c_i) = \sqrt{c_i}$ function must be evaluated 83 times, with 21 points a total of 908 times (10 times more), and with 101 mesh points 20508 times (246 times more). For a small size problem such as the present, this represents no problem, but for a large-scale problem this may be a concern.

4 An implementation of the savings problem in C++

The problem outlined above can easily be solved by hand or for instance in a spread-sheet. However solving more complicated models can only be done using proper programming methods. Therefore it makes more sense to start of using C++ right away, since it will prove to be the right choice later[3] when solving more complex models.[4] This section outlines a program that can solve the problem above - and easily be extended to solve more di¢cult problems. This section can be skipped by readers who are not interested in

[3] It could also be argued that Fortran, Matlab or Gauss (etc.) is the right choice. Here C++ is chosen - this choice is a matter of taste.

[4] Recommended readings to get started with C++ include ? and ?. Text books with focus on numerics and mathematics are ? and ?.

looking at source code, since some notion of C++ is required.

The program will contain the following steps:

1. Initialize program
2. Initialize grid
3. Calculate last period
4. Calculate all other periods recursively in a loop
5. Display results

4.1 Initialize program

First, we must initialize the C++ program and include various libraries:

```
//Value function iteration in C++
#include <stdio.h>
#include <math.h>
#include <stdlib.h>
```

Secondly, we must de…ne the size of the grid: here containing 6 equidistant points between 0 (ASSET_MIN) and 5(ASSET_MAX):

```
//Compiler directive definitions:
#define PERIODS 3
#define ASSET_GRID 6
#define ASSET_MAX 5
#define ASSET_MIN 0
#define beta 0.9
```

Thirdly we must define matrices to save the results. Notice that Vmax and the associated optimal consumption level Cmax are of the type double, whereas Amax (the optimal choice of end-of-period assets) is an index on the grid, and therefore an integer. Saving the grid point's index, instead of its associated value, saves resources.

```
//Define matrice to save the results in
double Vmax[PERIODS+1][ASSET_GRID];
int Amax[PERIODS+1][ASSET_GRID];
double Cmax[PERIODS+1][ASSET_GRID];
```

The last part of the initialization is to define the utility function $u(x) = \sqrt{x}$ as a function:

```
//define function prototype for utility function
double u(double x) {return pow(x,0.5);}
```

4.2 Initialize grid

Next we must initialize the grid. This is done filling the vector Asset[i] with the numeric value associated with the i'th grid point on the asset grid. In this case the points are equidistant - with the distance between points given by the formulae below:

```
int main(){
int t,i, from, to;
//define and initialize asset grid:
double Asset[ASSET_GRID];
for(i=0;i<ASSET_GRID;i++) Asset[i] =
    (double)i*(ASSET_MAX-ASSET_MIN)/(ASSET_GRID-1)+ASSET_MIN;
```

4.3 Calculate the last period

Finally we have the ingredients for computing the last period utility for assets on the grid. As argued above we know that the solution is $M_2 = C_3$ and therefore that $V_3(M_2) = \sqrt{C_3}$.

The procedure is quite simple: for each element in the asset grid (A) we compute V (A) (and save it in the Vmax), find the optimal value of the control variable (C) (and save it in the Cmax), and save the optimal end-of-period asset holdings in Amax:

```
//calculate last period
for(i=0;i<ASSET_GRID;i++) {
   Vmax[PERIODS][i] = u(Asset[i]);
   Amax[PERIODS][i] = i;
   Cmax[PERIODS][i] = Asset[i];
   };
```

4.4 Calculate all other periods recursively in a loop

All the other periods can be computed using the same methodology; only the last period requires a special solution technique because of its particularly simple optimal policy function. This means computing each of the remaining periods recursively until we reach the first period.

```
//calculate the remaining periods:
for(t=PERIODS-1;t>0;t--)
    {
```

Compute utility associated with all feasible combinations of beginning-of-period and end-of-period assets. Consumption (cons_incom) is calculated as the difference between these two asset holdings. Since negative consumption is not allowed we introduce sufficiently large negative penalty for these combinations (=-99999999), and hence these combinations can never be chosen as utility maximizing choices.

```
for(from=0;from<ASSET_GRID;from++){
    double best_option_value=-99999, best_option_cons;
    int best_option=0;
```

```
        for(to=0;to<ASSET_GRID;to++){
            double utility=0;
            double cons_income = Asset[from]-Asset[to];
            if (cons_income<0) utility=-99999999;
            utility = utility + u(cons_income);
            utility = utility + beta*Vmax[t+1][to];
```

Next we need to check if a combination gives higher utility than other combinations for the given beginning-of-period assets. In this case the associated value and index of the control variable is saved.

```
            if(utility>best_option_value){
                best_option_value=utility;
                best_option = to;
                best_option_cons = cons_income;
                };
        }//next to
```

Finally we need to save the optimal choices as well as the associated control variables for each point on the beginning-of-period asset grid:

```
        //save optimal values:
        Vmax[t][from] = best_option_value;
        Amax[t][from] = best_option;
        Cmax[t][from] = best_option_cons;
    }//next from
}//next period
```

4.5 Display results

Last we need to display the optimal choices computed above. This is done with the following C-style printf statements:

```
//display
for(t=PERIODS;t>0;t--){
    printf(''nnnnPeriod:  %i'',t);
    for(i=0;i<ASSET_GRID;i++)
        printf(''nnV%#3i(%#5.3f) = nt%#8.5f'',t,Asset[i], Vmax[t][i]);
}
```

And …nish the program:

```
return 0;
};
```

4.6 Output from the program

Below is shown the output from the program: the value function for each period.

```
Period: 3
V 3(0.000) = 0.00000
V 3(1.000) = 1.00000
V 3(2.000) = 1.41421
V 3(3.000) = 1.73205
V 3(4.000) = 2.00000
V 3(5.000) = 2.23607
Period: 2
V 2(0.000) = 0.00000
V 2(1.000) = 1.00000
V 2(2.000) = 1.90000
V 2(3.000) = 2.31421
V 2(4.000) = 2.68701
V 2(5.000) = 3.00484
Period: 1
V 1(0.000) = 0.00000
V 1(1.000) = 1.00000
V 1(2.000) = 1.90000
V 1(3.000) = 2.71000
V 1(4.000) = 3.12421
V 1(5.000) = 3.49701
```

The numbers from the …rst period can be compared with the values in table (17) - they are the same.

4.7 Acceleration

The program outlined above can be accelerated in various ways. Here are a few examples that will be left to the reader to implement:

1) The points on the grid need not be equidistant as in the example above. Looking at Figure 5 we see that the Value function at time 1 is less non-linear the higher the initial assets are. Therefore one idea is to locate the grid-points closer for small assets holdings, and coarser for higher levels of assets, instead of equidistant. One way of achieving this is by using a logarithmic transformation, such that the distance between the i'th and the $i+1$'th grid point is $(1+0{:}05)^i$ instead of 0:05. In the program above this is easily implemented by altering the de…nition of the grid.

2) The computations of $u(c)$ can done outside the loop - this would mean less evaluations of the utility function. Since the grid is the same in all periods the consumption possibilities are either 0,1,2,3,4 or 5 - hence these need only be evaluated once.

3) The grids need not be the same in each period. In this simple model where there is no income during the life, there is no point in computing $V_3(M_2 = 5)$. Why? Simply because economic intuition about the problem tells us, that no agent would ever make this choice if the highest grid point in the period before is 5, since this implies a zero consumption in period 2. The table with optimal policy choices in the previous period shows that the richest agent in period 2 will at most leave 2 units for the …nal period (in the case where his assets are at the max: 5). This acceleration trick is of cause only useful if we need to solve the dynamic programming more than once - or have prior information about which values are reasonable from elsewhere.

4) In general, economic intuition can often be used to perform optimizations. For instance one can often place upper and lower bounds on the discritized variables. Suppose in the previous example that we know that the richest consumer will be born with an income

of 4 (this knowledge can come from previous analysis of the problem or just be a guess). Since there is no interest on savings in the set-up above, this means that it does not make sense to spend time evaluating the value function with asset holdings of 5. In the more realistic case with interest on savings (of say 10 percent), we know that the consumer's asset holdings in period j cannot exceed 5×1.1^j since this is the pure accumulation stream (i.e. with no consumption in any period). This knowledge gives us an upper bound for the discritization of assets, and calculating the valuefunction for larger asset holdings would be a waste of clock-cycles and time.

4.8 Addendum: source code for the deterministic program

```
//Value function iteration in C++
#include <stdio.h>
#include <math.h>
#include <stdlib.h>
//Compiler directive definitions:
#define PERIODS 3
#define ASSET_GRID 6
#define ASSET_MAX 5
#define ASSET_MIN 0
#define beta 0.9
//Define matrice to save the results in
double Vmax[PERIODS+1][ASSET_GRID];
int Amax[PERIODS+1][ASSET_GRID];
double Cmax[PERIODS+1][ASSET_GRID];
//define function prototype for utility function
double u(double x) {return pow(x,0.5);}
int main(){
 int t,i, from, to;
 //define and initialize asset grid:
 double Asset[ASSET_GRID];
 for(i=0;i<ASSET_GRID;i++) Asset[i] =
    (double)i*(ASSET_MAX-ASSET_MIN)/(ASSET_GRID-1) + ASSET_MIN;
 //calculate last period
 for(i=0;i<ASSET_GRID;i++)
 { Vmax[PERIODS][i] = u(Asset[i]);
 Amax[PERIODS][i] = i;
 Cmax[PERIODS][i] = Asset[i]; };
 //calculate the remaining periods:
 for(t=PERIODS-1;t>0;t--)
```

```
{
for(from=0;from<ASSET_GRID;from++){
double best_option_value=-99999, best_option_cons; int best_option=0;
for(to=0;to<ASSET_GRID;to++){
double utility=0;
double cons_income = Asset[from]-Asset[to];
if (cons_income<0) utility=-99999999;
utility = utility + u(cons_income);
utility = utility + beta*Vmax[t+1][to];
if(utility>best_option_value)
{ best_option_value=utility;
best_option = to;
best_option_cons = cons_income;
};
}//next to
//save optimal values:
Vmax[t][from] = best_option_value;
Amax[t][from] = best_option;
Cmax[t][from] = best_option_cons;
}//next from
}//next period
//display
for(t=PERIODS;t>0;t--)
{
printf(''nnnnPeriod:  %i'',t);
for(i=0;i<ASSET_GRID;i++)
printf(''nnV%#3i(%#5.3f) = nt%#8.5f'',t,Asset[i], Vmax[t][i]);
}
return 0;
};
```