Dynamic economics in Practice

Numerical methods with Matlab

Monica Costa Dias and Cormac O'Dea

Introduction

Matlab

- Matlab is a software package and programming language
- Widely used in Dynamic Programming and in economics in general
- Proprietary and expensive
  - Though most universities have it and a substantially discounted student version can be obtained
- Has a number of additional 'toolboxes' that supplement standard features
  - Optimisation toolbox is essential for work in dynamic programming

Alternatives to Matlab

- Fortran, C++, C
  - Much much faster, which can be very important
- R
  - Free, increasingly popular in economics
- Mata (Stata's programming language)
  - Newer, less widely used

Code developed for this course

- The code we will go through is designed to be easy to understand, not to be as fast as possible
- It does not come with a warranty!
  - There may be parameter sets that deliver an error or an odd result
- You may not understand it all completely as we zoom through it
  - It is annotated
  - To properly understand it, it will be necessary to go through it more meticulously than we will have time to do here
- The existing code does not work as a black box – you need to understand it to edit/change
- But then it can be generalised/specialised to different problems
- And it is simple to transpose to other more efficient languages

Code for this course

We have six versions of the code, that follow the increasingly more sophisticated models that will be discussed

1. Finite consumption saving endowment problem (solved by maximising the Value function)
2. Adds to (1) option to solve using the Euler equation
3. Adds to (2) deterministic stream of income and capacity to borrow
4. Adds to (3) simple uncertainty in income
5. Adds to (4) a conventional stochastic income process
6. Solves infinite horizon version of the model in (5)

Matlab basics

Functions

We can create a simple function by creating a file with the following text:

```
function [ output ] = myFunction( input )
% This function outputs the twice the square of the input
  output = 2 * (input^2);
end
```

and saving it by the name `myFunction.m`.

When we place this file in our working directory, we can then call the function from Matlab:

```
[y] = myFunction(2);
```

This returns `y = 8`.

Scripts

- Scripts are also organise in files
- They contain a lists of commands
- Unlike functions they don't start with a header . . .
  - `function [ output ] = myFunction( input )`
- . . . and they don't finish with:
  - `end`
- A key difference between scripts and functions concerns which parts of memory they have access to
  - Functions accept inputs and return outputs; other variables are internal to the function
  - Scripts do not accept arguments and operate on data in the workspace

Locals and Globals I

Local and global variables

- Local variables are those that can only be accessed by the file in which they are introduced
- Global variables are those that can be accessed (and changed) by any file

Key difference between scripts and functions:

- If *script or function* `a.m` calls *script* `b.m`, then `b.m` can access any variable that `a.m` has access to (in the workspace of `a.m`)
- If *script or function* `a.m` calls *function* `b.m`, then `b.m` has a separate worspace and can only access *global* variables declared in `b.m`

Locals and Globals II

To declare a variable (`myvar`) as a global, in the file where it is first initialised include the line:

```
global myvar
```

We can access this variable in any other script or function called by this file by also including the line:

```
global myvar
```

Setup of a project

- In a particular folder you will need a 'master' script (called perhaps `main.m`) and all the scripts and functions that will be called
- Set the working directory in Matlab as this folder
- Run the script `main.m` by typing `main` at the Matlab prompt or by clicking the green 'run' button in the Matlab text editor

Some basic commands

- `+ - * / ^` basic operators
- `.* ./ .^` elementwise operators
- `;` suppress output
  - `A = 1;` and `A = 1` both store scalar 1 in A
  - The latter also prints `A=1` to screen
  - You'll want to terminate most commands with `;`
- `tic, toc` starts and finishes a stopwatch and prints time taken

Creating a matrix

- `A=[1 2 3; 3 5 6]` creates the matrix

$$A = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}$$

- `B=zeros(1,3)` creates a 1×3 vector of zeros
- `C=NaN(2,1)` creates a 2×1 vector of missing values
- `X=[A;B]` concatenates vertically matrices `A` and `B`
  - `X` has dimension 3×3
- `X=[A,C]` concatenates horizontally matrices `A` and `C`
  - `X` has dimension 2×4
- `X=repmat(A, (n, m))` replicates matrix A n by m times
  - If matrix A has dimension $j \times k$, `X` will have dimension $nj \times mk$

Selecting out elements of a matrix

Say we want a vector that contains the row `t` of matrix `V`

```
Vt  = V(t , :) ;
```

Say we want elements in columns 3 to 5 of row `t` of `V`

```
Vt  = V(t , 3:5) ;
```

Loops

A ‘for’ loop:

```
squares = NaN(10, 1);
for ix = 1:1:10
   squares(ix) = ix ^ 2;
end
```

A ‘while’ loop producing the same result:

```
squares = NaN(10, 1);
ix = 1;
while ix ≤10
   squares(ix) = ix ^ 2;
   ix = ix + 1;
end
```

If blocks

A simple 'if' block:

```
if x1 < 0
    fprintf('x1 is negative')
elseif x1 == 0
    fprintf('x1 is zero')
else
    fprintf('x1 is positive')
end
```

Generating a grid of equally spaced values in Matlab

```
grid = linspace(0, 1, 4)
```

Generates and stores in a vector a grid with minimum 0, maximum 1 and four points such that the space between the points is equal

```
grid = 0    0.3333    0.6667    1.0000
```

Generating a grid of values more concentrated towards lower values

Do this by equalising the log-distance between 0 and some upper bound (call it `top`) for the grid:

```
top     = 1;
loggrid = linspace(log(1),log(1 + top), 4);
grid    = exp(loggrid)-1;
```

Returns a grid with minimum 0, maximum `top` and four points at increasing distance:

```
grid =  0    0.2599    0.5874    1.0000
```

First Matlab program – folder 'code\v1'

Set up Matlab folder and program to solve and simulate simple cake eating problem without uncertainty

Optimisation

Optimisation in Matlab I

- Problem: Find x such that $f(x)$ is minimised
- Matlab has many commands as part of its optimisation toolbox.
- We will discuss `fminbnd`

```
[Y, fval] = fminbnd(@(x) f(x), lowerbound, upperbound);
```

This returns the value of the argument that minimises the function in `Y` and the minimised value of the function in `fval`

```
[Y, fval] = fminbnd(@(x) (2*(x^2) - x), -100, 100);
```

This returns `Y = 0.25` and `fval = -0.125`

Optimisation in Matlab II

- What would we do if we wanted to *maximise* a function rather than minimise it?
- We could minimise the negative of that function

```
[Y, fval] = fminbnd(@(x) -(2*(x^2) - x), -100, 100);
```

- The algorithm used by `fminbnd` alternates between the Golden Section Search and Successive Parabolic Interpolation
  - Optimiser for one-dimensional continuous functions
  - As other numerical optimisers: returns local solutions
  - Does not return a minimum on the boundary of the optimising interval – will return a close approximation
  - Slow convergence if solution is on the boundary

Optimisation in Matlab III

- Instead of writing the function directly in the `fminbnd` command line, we can write it in a separate file
- The function may include arguments other than the one we are maximising with respect to

First define the function and save the file as `myFunction.m`

```
function [ z ] = myFunction(x, a)
    z = a*(x^2) - x;
end
```

Then find its minimum for a given value of `a`

```
a = 5;
[Y, fval] = fminbnd(@(x)  myFunction(x, a), -100, 100);
```

This returns `Y = 0.1` and `fval = -0.05`

Approximation

Representing a function numerically

How can we store a function that is defined numerically?

- Take a simple function $f(x) = \ln(x)$
- Closed-form representation exists but say we wanted to store it numerically?
- We would define two vectors (say x and f)
- x will contain a number of points in the domain of the function, at which we decide to evaluate $f(x)$ ...
- and f will contain the value of the function at those points

$$\begin{pmatrix} 0.1 \\ 0.2 \\ 0.5 \\ 1 \\ 2 \end{pmatrix} \begin{pmatrix} -2.3 \\ -1.6 \\ -0.7 \\ 0.0 \\ 0.7 \end{pmatrix}$$

Approximation of a function using Matlab

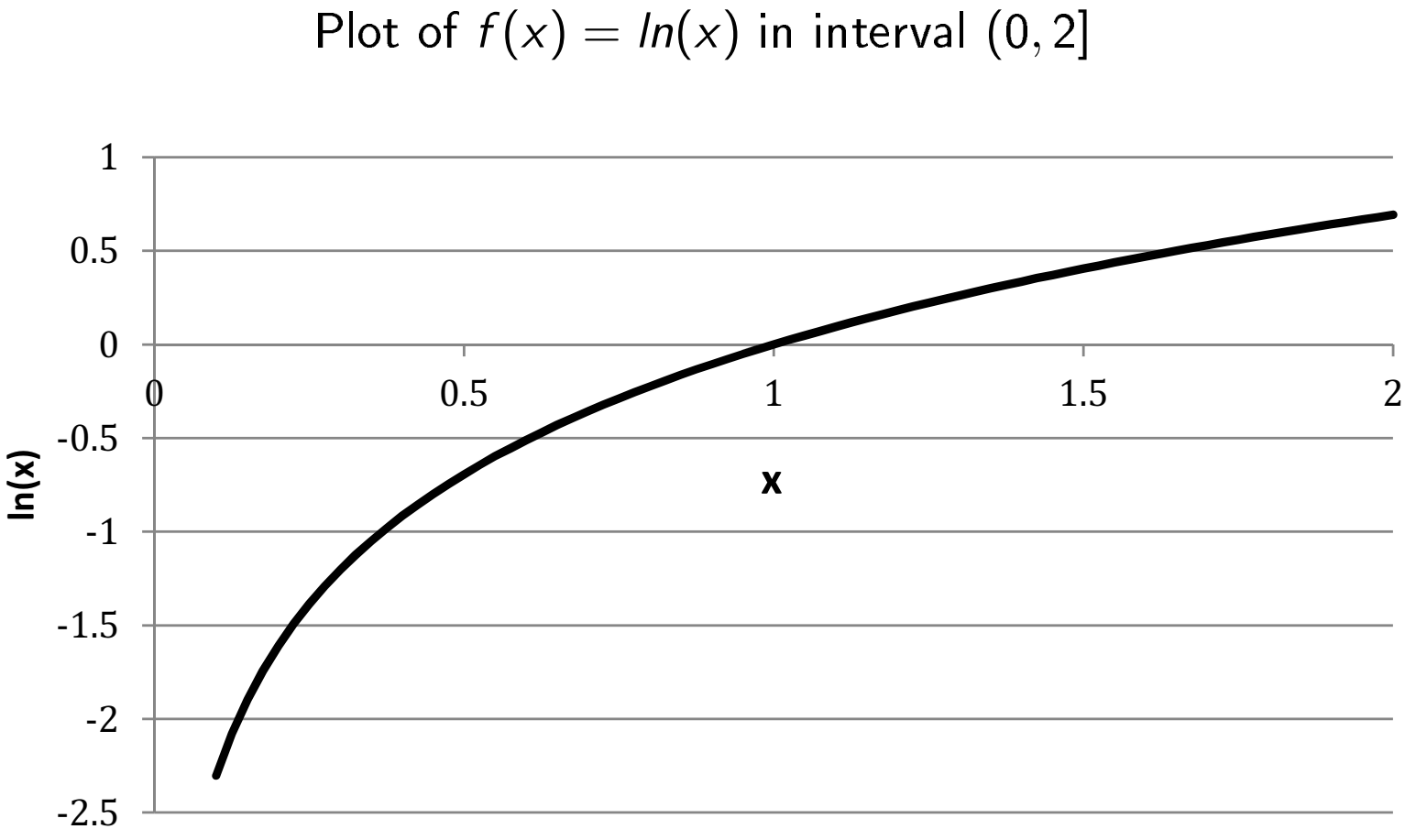
- Suppose we only know the value of a function on a discrete subset of its domain
- And we want to know the value of the function at a point `x0` on the domain but not in that subset
- Matlab has a tool (`interp1`) that allows us to approximate the value of the function at that point

```
approx = interp1(x, f, x0, method);
```

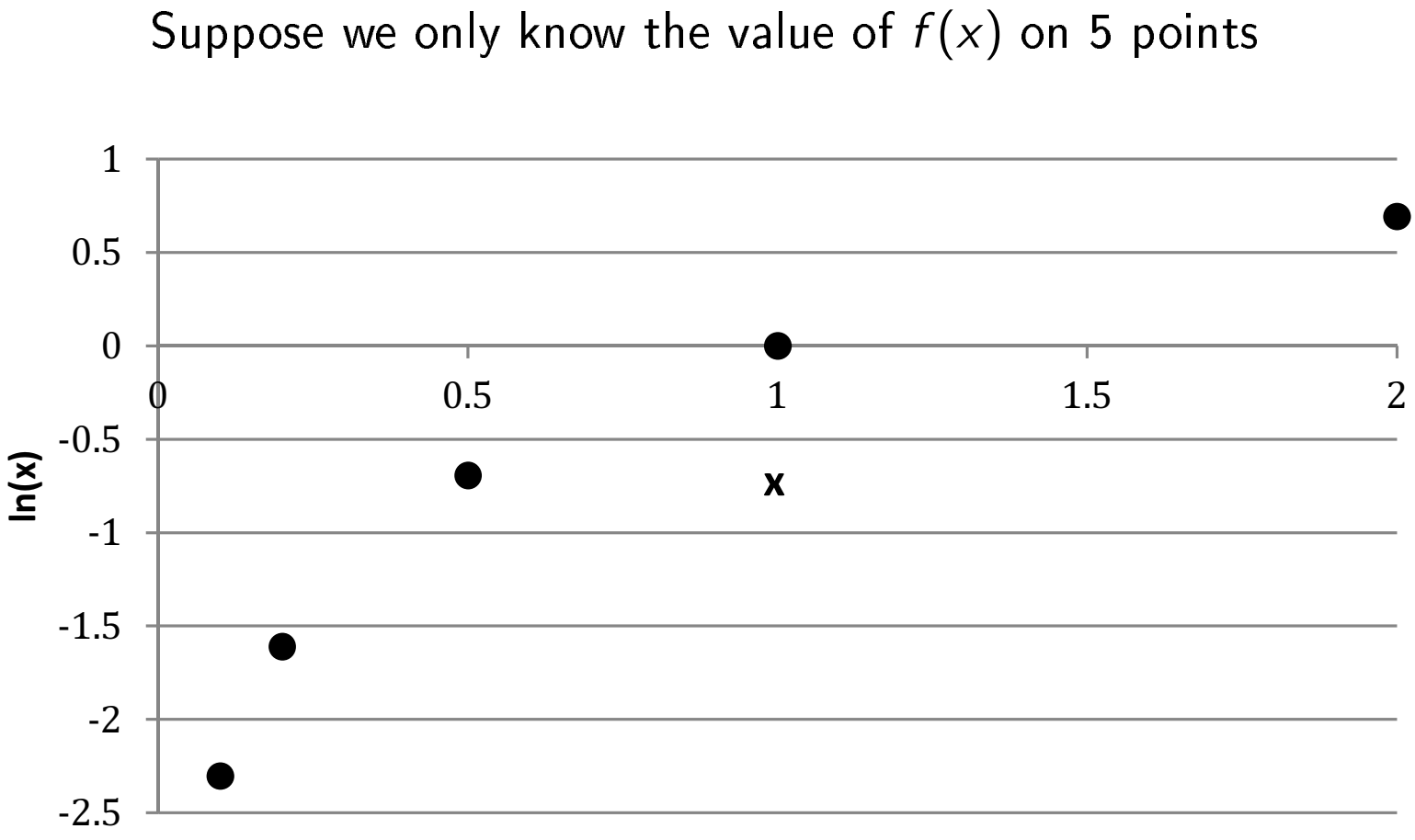
where:

- `x` is the vector containing the points at which we know $f(x)$
- `f` is the vector containing the value of f at each point in `x`
- `x0` is the point at which we want to approximate f
- `method` specifies which algorithm to use

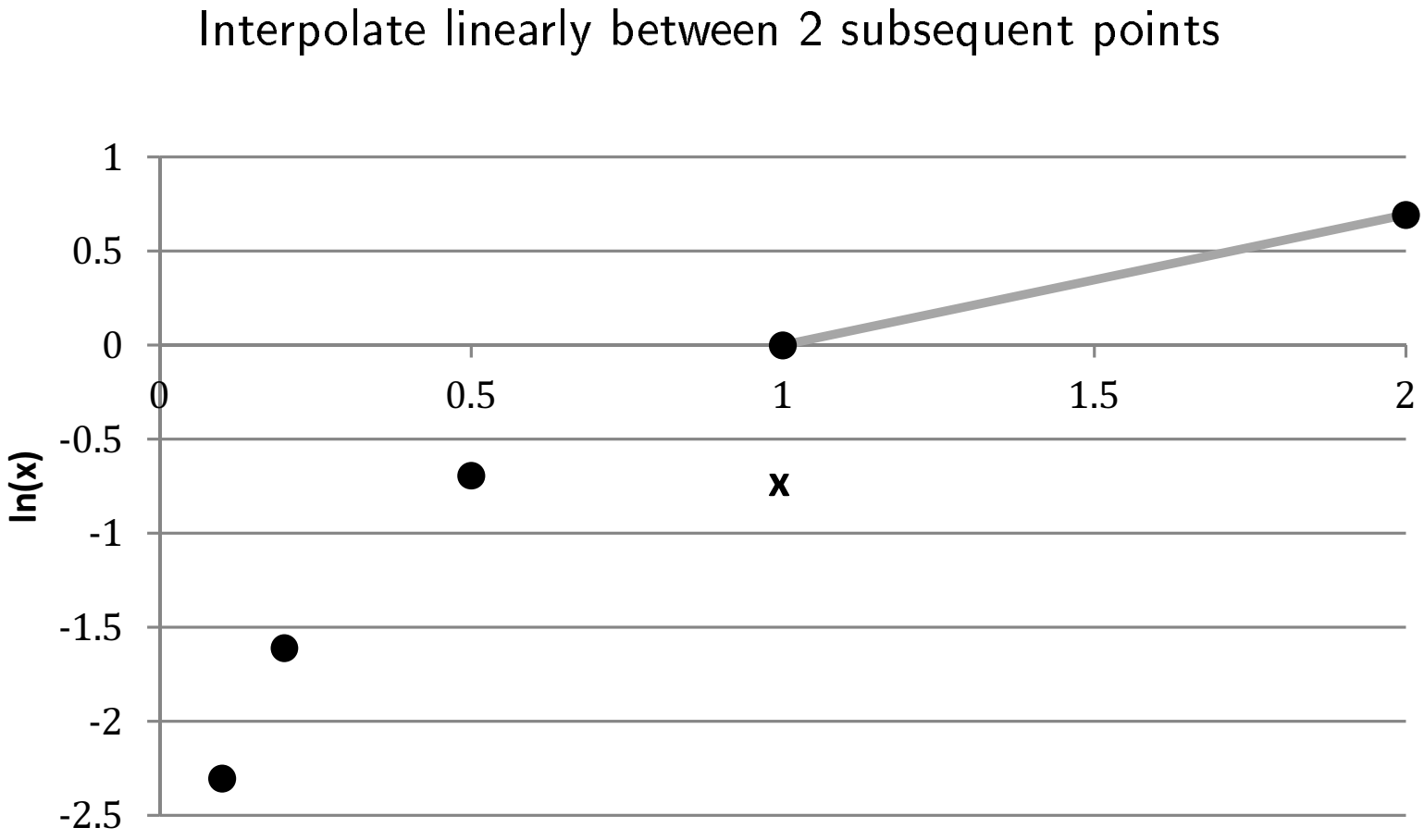
Linear interpolation



Linear interpolation

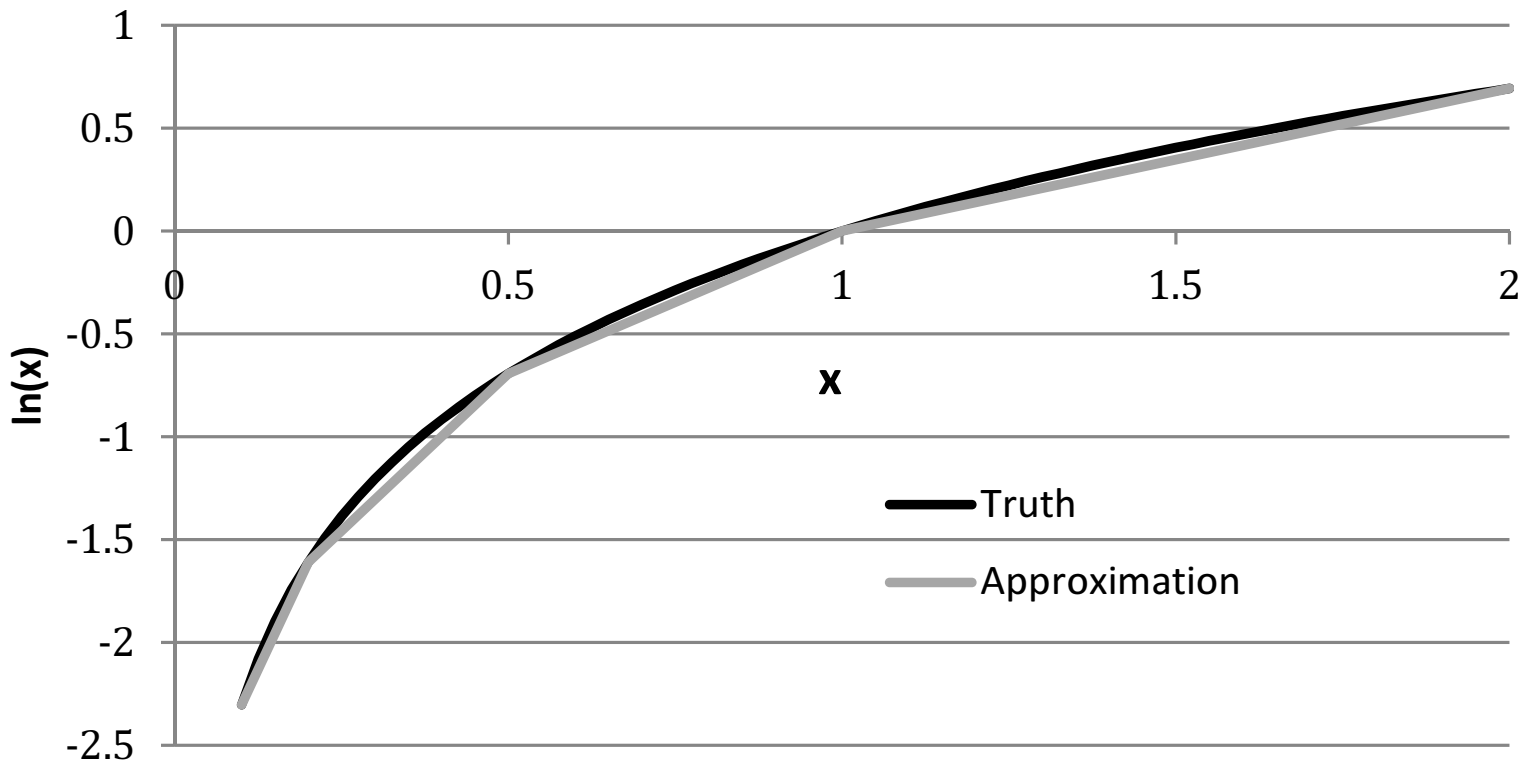


Linear interpolation



Linear interpolation

Approximate $ln(x)$ by linear interpolation in the space defined by the grid



Linear interpolation

Let's approximate the value of $f(x) = \ln(x)$ at $x = 1.5$ given the values of $f(x)$ at the 5 points we previously defined:

$$\begin{pmatrix} 0.1 \\ 0.2 \\ 0.5 \\ 1 \\ 2 \end{pmatrix} \begin{pmatrix} -2.3 \\ -1.6 \\ -0.7 \\ 0.0 \\ 0.7 \end{pmatrix}$$

```
x = [0.1, 0.2, 0.5, 1, 2];
f = [-2.3, -1.6, -0.7, 0.0, 0.7];
approx = interp1(x, f, 1.5, 'linear');
```

This returns `approx = 0.35` (the truth is $ln(1.5) = 0.405$)

How good will the approximation be?

- Approximating a function using linear interpolation will be better:
  - The closer together are the 'grid points', at which we know the true function value
  - The closer the underlying true function to linear
- It helps to concentrate grid points where the function is more non-linear
- Alternative approximation techniques are generally more appropriate to deal with non-linearities:
  - Local polynomials (splines), can be shape preserving
  - But they increase computation time

First Matlab program – folder 'code\v1'

Back to the code to solve and simulate simple cake eating problem without uncertainty

- Use `fminbnd` to maximise the value function
- Use `interp1` to interpolate the solution on grid

Root-finding

Root-finding in Matlab I

- Problem: Find x such that $f(x) = 0$
- For many functions $f(x)$, there is no closed form solution for x such that $f(x) = 0$
- Matlab has a command (`fzero`) to do this:

```
[Y, fval] = fzero(@(x) f(x), StartingValue)
```

- This returns the value of the argument that minimises the function in `Y` and the value of the function f at Y in `fval`.
- `fval` should be very close to 0 if the function has worked correctly

```
[Y, fval] = fzero(@(x) (2*x - 1), 0.2);
```

returns `Y = 0.5` and `fval = 0`

Root-finding using Matlab II

Instead of giving Matlab a starting value, it is better, if you can, to give it a bound within which you know the zero lies:

```
boundforzero = [0, 1];
[Y] = fzero(@(x) 2*x - 1, boundforzero);
```

This returns `Y = 0.5`

The algorithm uses bisection, secant, and inverse quadratic interpolation methods and was developed by Dekker (1969) and Brent (1973).

Root-finding using Matlab III

Instead of writing the function directly when `fzero` is called, we can write the function separately and include other arguments

First write the function and save it in file `myFunction.m`

```
function [ z ] = myFunction(x, a)
    z = a*x - 1;
end
```

Then call it after defining the parameter `a` and the starting interval

```
a = 3
boundforzero = [0, 1];
[Y] = fzero(@(x) myFunction(x, a), boundforzero);
```

This returns `Y = 0.333`

Second Matlab program – folder 'code\v2'

Solve cake eating problem without uncertainty using `fzero` to find the root of the Euler equation

More on function approximation

Improving linear approximation with other known information about the function: quasi-linearisation

- Revisit the problem of approximating a non-linear function f
- Linear interpolation is fast but not very accurate
- Suppose however that we know another function g such that $g(f(x)) = (g \circ f)(x)$ is close to linear in x and that g is invertible
- We can use g to do the following:
  1. At each point for which we know $f(x)$, calculate $(g \circ f)(x)$
  2. Approximate $(g \circ f)(x)$ using linear interpolation and store this $\widehat{(g \circ f)}(x)$
  3. Calculate $g^{-1}\left(\widehat{(g \circ f)}(x)\right)$ which is our approximation $\widehat{f}(x)$ to $f(x)$
- As long as $(g \circ f)$ is closer to linear than is f, then this should deliver a better approximation than directly interpolating on f

Quasi-linearisation example

Suppose we want to approximate marginal utility (with a CRRA utility function) using linear interpolation

$$f(c) = c^{-\gamma}$$

There exists an invertible function:

$$g(x) = x^{-\frac{1}{\gamma}}$$

such that $(g \circ f)(c) = c$ is linear and so can accurately be interpolated linearly

We can then recover $f(c)$ by applying the inverse of g to $(g \circ f)(c)$ where

$$g^{-1}(y) = f(y) = y^{-\gamma}$$

Quasi-linearisation example

Consider three vectors – $c, f(c), g(f(c))$ where (f, g) are functions as above with $\gamma = 2$

$$\begin{pmatrix} 0.1 \\ 0.2 \\ 0.5 \\ 1 \\ 2 \end{pmatrix} \begin{pmatrix} 100 \\ 25 \\ 4 \\ 1 \\ 0.25 \end{pmatrix} \begin{pmatrix} 0.1 \\ 0.2 \\ 0.5 \\ 1 \\ 2 \end{pmatrix}$$

- The true marginal utility at $c = 1.5$ is $f(1.5) = 1.5^{-2} = 0.444$
- Approximating f at $c = 1.5$ by linear interpolation yields $\widehat{f}(1.5) = 0.625$
- Approximating $(g \circ f)(c)$ at $c = 1.5$ by linear interpolation yields $\widehat{(g \circ f)}(1.5) = 1.5$
- Inverting $\widehat{(g \circ f)}$ to recover f at $c = 1.5$ yields $\widehat{f}(1.5) = (1.5)^{-2} = 0.444$

Alternatives to linear interpolation

Alternative to linear approximation that Matlab provides include:

- 'nearest': nearest neighbour matching
- 'spline': piecewise cubic spline
- 'pchip': shape-preserving piecewise cubic interpolation

Recall our attempt to approximate the value of $ln(1.5)$:

```
approx1 = interp1(x, f, 1.5, 'linear'); (0.34)
approx2 = interp1(x, f, 1.5, 'nearest'); (0.7)
approx3 = interp1(x, f, 1.5, 'spline'); (0.467)
approx4 = interp1(x, f, 1.5, 'pchip'); (0.442)
```

The truth is $ln(1.5) = 0.405$.

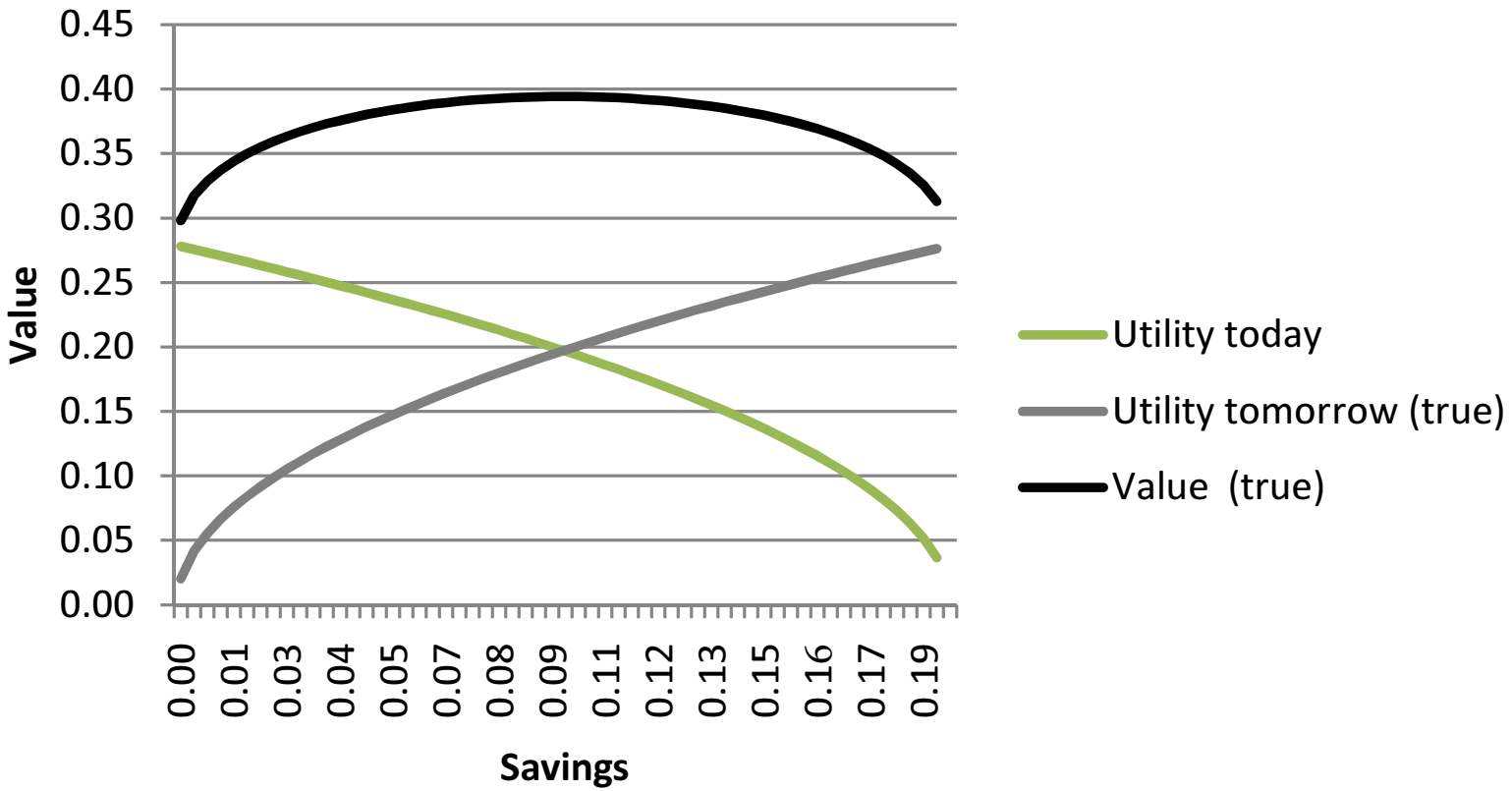
Approximation error I

To see how approximation error can lead to serious misleading results consider the following example:

- A consumer lives for 2 periods and has endowment of 0.2 units of consumption to split between those two periods
- Utility is CRRA with $\gamma = 0.5$
- The interest rate and discount rate are equal to zero
- The true optimal is $C_1 = C_2 = 0.1$

Approximation error II

Plot utilities at $t = 1, 2$ and value at $t = 1$ for consumption at $t = 1$ varying in the interval $(0, 0.2)$

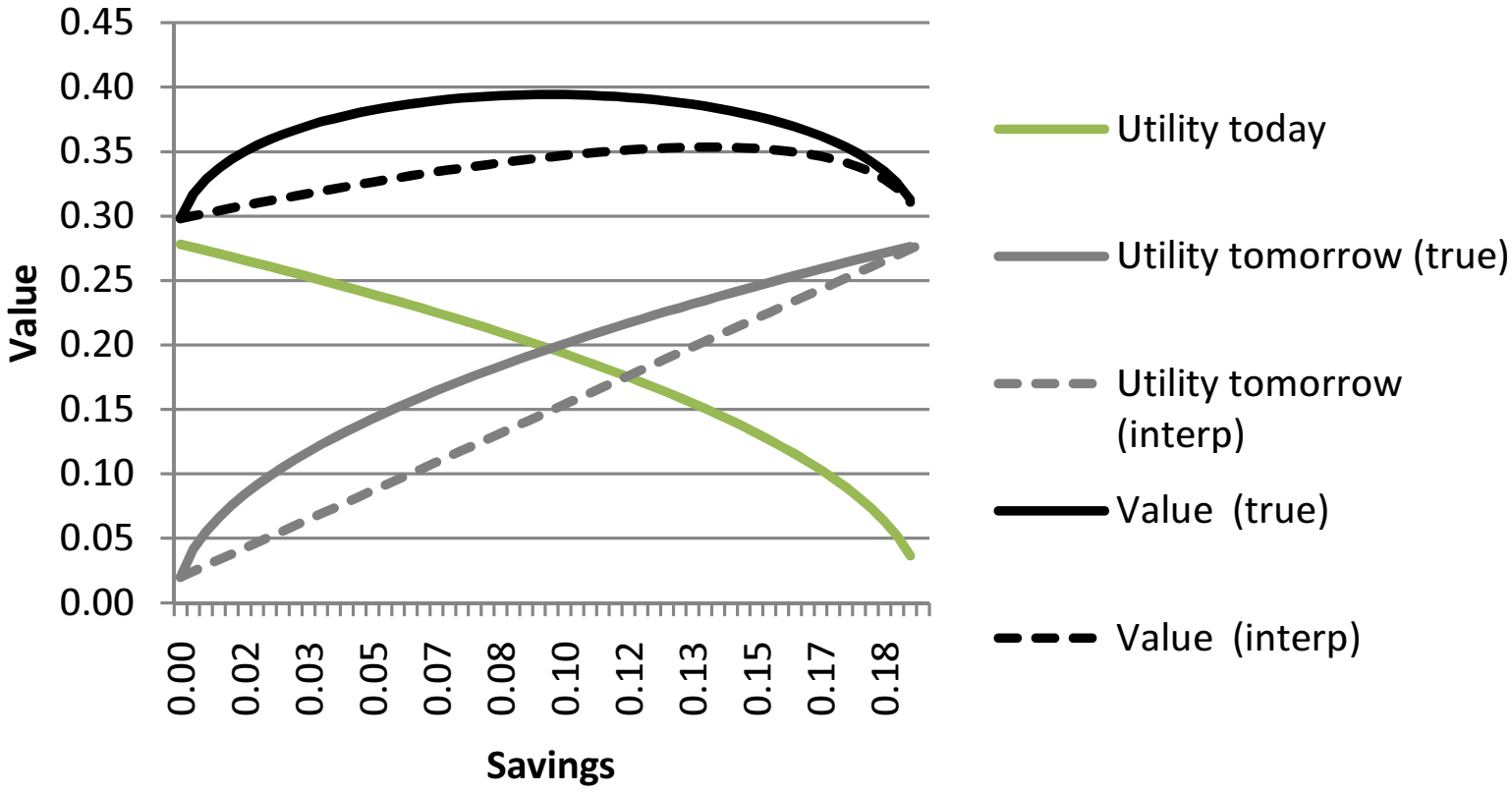


Approximation error III

To see how approximation error can lead to serious misleading results consider the following example:

- A consumer lives for 2 periods and has endowment of 0.2 units of consumption to split between those two periods
- Utility is CRRA with $\gamma = 0.5$
- The interest rate and discount rate are equal to zero
- The true optimal is $C_1 = C_2 = 0.1$
- Now suppose we only knew the value of savings (utility of savings tomorrow) at $C_1 = 0.001$ and $C_1 = 0.019$ and had to interpolate the utilities between these points

Approximation error



Approximated optimal consumption would be $C_1 = 0.065; C_2 = 0.135$

Back to second Matlab program – folder 'code\v2'

Solve cake eating problem without uncertainty using `fzero` to find the root of the Euler equation and inverse marginal utility to reduce approximation error

Conclusion

Conclusion

- When solving things numerically, one does not know what the true answer is
- It is difficult to absolutely verify that there is no mistake
- Calls for disciplined coding practices:
  - Write many many many checks and warnings into your code
  - Once you think your program works - try to break it by stress-testing - select extreme parameters - may help you find bugs