Computational Bootcamp

Advanced Topics in Matlab

Kathrin Schlafmann

Copenhagen Business School, DFI, and CEPR

Outline

Interpolation & Extrapolation

Optimization

Interpolation & Extrapolation

When do we need this?

- analytical solutions:
  - solution to an optimization problem: optimal behavior or the equilibrium outcome as a function of the initial state variables and parameter values
  - ie general solution for any value of the state variables and any value for the parameters of the problem

- however: often such analytic closed form solutions do not exist!

$\Rightarrow$ that is the reason why we use the computer in the first place!!

Interpolation & Extrapolation

When do we need this?

- numerical solutions: we typically work with specific values of parameters / variables
  - set values for economic parameters at the beginning of the solution (e.g. values for preferences parameters such as risk aversion or discount factor)
  - solve optimization problem on a grid for state variables (need to set the value for these grid points at the beginning of the program!)
- thus: we obtain a numerical solution for a particular value of the parameters and the state variables (more precisely: for a grid of values of the state variables)

Interpolation & Extrapolation

When do we need this?

Example:

- optimization problem:

$$\max_{c_1, c_2} \ \log(c_1) + \beta \log(c_2)$$
$$s.t. \ c_2 = (1 + r)(x - c_1)$$

- analytical solution: $c_1(x) = \frac{1}{1+\beta} x$
- numerical solution:
  - set values for parameters: `beta = 0.95, r = 0.02`
  - set grid for state variables: `xx = [0.1, 0.3, 1]'`
  - solution: `c1 = [0.0513, 0.1538, 0.5128]'`

Interpolation & Extrapolation

When do we need this?

- challenge: what if we need to know the optimal consumption c_1 for value of x that is not on the grid (i.e. not on the vector `xx`)?

  $\rightarrow$ this where we use interpolation / extrapolation:

  - interpolation: find the value of `c1` that corresponds to a value of x that lies *between two grid points* of `xx`
  - extrapolation: find the value of `c1` that corresponds to a value of x that lies *outside the range of* `xx`

- note: both interpolation and extrapolations are only approximations to the value a complete numerical solution of the problem at this point would deliver

Interpolation & Extrapolation

Caveats

- how good the approximation is depends on
  - nature of problem (ie the shape of the true function c_1 to be solved for)
  - the number of grid points in `xx`: the smaller the gaps between grid points the more precise the approximation when interpolating
  - the choice of interpolation procedure

- WARNING: extrapolation is typically very dangerous to do (can lead to rather unpredictable outcomes for the approximation!)
  - that is particularly true if you need to extrapolate for a value that is very far from the range of the grid you have solved for!
  - always make sure your grid values are chosen in an appropriate range for your optimization problem and make sure to check the impact of any extrapolation you do!

Interpolation & Extrapolation

Matlab

Matlab has built-in functions which can do the interpolation (and extrapolation):

- `vq = interp1(x,v,xq,method,extrapolation)`
  this function can interpolate and extrapolate a 1-D function $v(x)$
  - `x` : the grid of the state variable you have solved the problem for
  - `v` : the values of the function that correspond to the grid points in `x`
  - `xq` : the points you want to approximate the value for
  - `vq` : the appoximated value of the function at point `xq`
  - `method` (optional): here you need to choose which interpolation procedure you want to use
  - `extrapolation` (optional): here you need specify whether you want to allow extrapolation or not
- `interp2(...)`, `interp3(...)`, and `interpn(...)` are very similar functions which interpolate 2-D, 3-D and N-D functions

Interpolation & Extrapolation

Matlab

- 1-D: `F = griddedInterpolant(x,v)`
  N-D: `F = griddedInterpolant(X1,X2,...,Xn,V)`

  this function is very similar to the `interpn()`-family of functions with one difference:

  - `interpn()`: immediately returns the interpolated value
  - `griddedInterpolant()`: returns an the interpolant `F` instead of the value
    $\rightarrow$ this interpolant is then used to evaluate at points later on:
    `vq = F(xq)`

note: if you need to evaluate the same function repeatedly in your code it is faster to use `griddedInterpolant()` since the construction of the interpolant is the time consuming step, not the evaluation!

Interpolation & Extrapolation

Interpolation Methods

- `nearest`: sets the value at the query point `xq` equal to the value `y_i` at the point `x_i` that is closest to `xq`
  $\rightarrow$ this leads to a discontinuous function!
- `linear`: constructs a piece-wise linear function (connect the points linearly) which is used to determine the value `vq`
  $\rightarrow$ the resulting function is continuous but the derivatives are not; however: this method weakly preserves monotonicity!
- `spline`: uses a cubic polynomial (different for each interval between grid points) such that the resulting function is continuous and has continuous first and second derivatives
  $\rightarrow$ while the resulting function is smooth, it does *not* preserve monotonicity! (in fact: it can shoot off in quite unpredictable undulations sometimes!)
- `pchip` (piece-wise cubic Hermite spline): an interpolation method similar to `spline` that avoids the overshooting problem and preserves monotonicity (only available for 1D interpolation)

Interpolation & Extrapolation

Splines:

$$S(x) = \begin{cases} C_1, & x_0 \leq x \leq x_1 \\ C_i, & x_{i-1} \leq x \leq x_i \\ C_n, & x_{n-1} \leq x \leq x_n \end{cases}$$

where each C_i is a cubic function:

$$C_i = a_i + b_i x + c_i x^2 + d_i x^3$$

so the interpolation method needs to determine 4n coefficients.

- end points of each segment have to be equal to the data points (2n conditions):

$$C_i(x_{i-1}) = y_{i-1}, \qquad C_i(x_i) = y_i$$

- first and second derivatives have to be smooth (2(n-1) conditions):

$$C_i'(x_i) = C_{i+1}'(x_i), \qquad C_i''(x_i) = C_{i+1}''(x_i)$$

- we need 2 additional conditions: end point conditions
  `spline` uses the not-a-knot end point condition (force continuity of the third derivative at the second and penultimate points)

Outline

Interpolation & Extrapolation

Optimization

Optimization

- In economic problems we typically deal with optimization problems
  - firms maximize profits
  - consumers maximize utility
  - etc
- if there is a closed-form solution: we know how to solve for these analytically
- numerical solution comes in if there is no closed form solution

Optimization

back to the earlier example:

$$\max_{c_1, c_2} \quad \log(c_1) + \beta \log(c_2)$$
$$s.t. \;\; c_2 = (1 + r)(x - c_1)$$

How do we solve this numerically?

Optimization

1. set parameter values for the economic parameters:
   `beta = 0.95`

2. fix a grid for the state variables for which we will solve the problem: `xx = [0.1, 0.3, 1]'`

   note: this is a small example, in practice of numerical solution you will typically use a grid with more grid points

3. choose an optimization routine & solve

Optimization routines:

1. grid search:
    - procedure:
        - fix a grid for the control variables: e.g. `linspace(0.01,0.99,100)`
        - compute value of objective function for each possible choice on this grid
        - pick the one that maximizes the objective function
    - advantage:
        - if grid for control variable adequate: global solution
        - derivative-free, hence robust for non-smooth problems
    - downside:
        - time and memory intensive
        - precision strongly depends on how fine the grid for control variable is

2. Matlab's build-in optimizers:
   - `fmincon`: minimize an objective function with constraints
   - `fminsearch`: minimize an objective function without constraints, derivative-free method
   - `fsolve`: solve a system of equations

advantage: higher precision

downside: might get stuck in local optimimum; can be very time-consuming

note: while economists always *maximize* objective functions, Matlab always *minimizes*!

$\rightarrow$ make sure to always reverse the sign of your objective functions (most common error when using optimization routines!!!)

Optimization

Optimization routine: `fmincon`

$$\min_x f(x) \quad s.t. \begin{cases} A \cdot x \leq b \\ Aeq \cdot x = beq \\ lb \leq x \leq ub \\ c(x) \leq 0 \\ ceq(0) = 0 \end{cases}$$

```
x =
fmincon(fun,x0,A,b,Aeq,beq,lb,ub,nonlcon,options)
```

- `fun`: the objective function to minimized
  $\rightarrow$ can be defined in-line or in a function-file
- `x0`: initial value that the optimizer uses to start its search
  $\rightarrow$ the choice of a good starting value is an art
  $\rightarrow$ you should use different starting values to make sure you don't get stuck in a local optimum
- `A, Aeq`: matrices for (in)equality constraints
- `b, beq`: vectors for (in)equality constraints
- `nonlcon`: function-file that contains nonlinear constraints (returns $c(x)$ and $ceq(x)$)

Optimization

Optimization routine: `fmincon`

- `options`: this is an object that contains *a lot of* options that governs the optimization routine (look up `fmincon` in the Matlab help functionality to learn more!)

  some important examples:

  - `Algorithm`
  - `Display ('off', 'iter', 'notify', 'final')`
  - `MaxIter`: maximum number of iterations before the solver exits
  - `TolX`: maximum change in the control variable (solver stops if smaller)
  - `TolFun`: maximum change in the objective function (solver stops if smaller)
  - `GradObj ('off', 'on')`: tell the optimizer whether the file with the objective function supplies the gradient, too

    $\rightarrow$ if you can you should always supply the gradient (saves time and more precise than numerical differentiation; can be checked with option `DerivativeCheck`)