Lecture 5: Function Approximation + The Curse of Dimensionality

Dynamic Programming

Thomas Jørgensen

Recap: Linear interpolation

- **Recap ($\#+1$ known nodes!):**
  - Let $\hat{x} = [\hat{x}_1, \hat{x}_2, \dots, \hat{x}_{\#+1}]$ be *grid points*
  - Let $\hat{f} = [\hat{f}_1, \hat{f}_2, \dots, \hat{f}_{\#+1}] = [f(\hat{x}_1), f(\hat{x}_2), \dots, f(\hat{x}_{\#+1})]$ be the *function values* at these points
  - **Linear interpolation** then is

$$f(x) \approx \breve{f}(x; \hat{f}) = \hat{f}_n + \frac{\hat{f}_{n+1} - \hat{f}_n}{\hat{x}_{n+1} - \hat{x}_n}(x - \hat{x}_n)$$
$$\text{where } \hat{x}_n \leq x < \hat{x}_{n+1}$$

- **Requirements for good interpolants:**
  1. Fast to set up and evaluate
  2. Shape preserving (monotonicity and concavity)
  3. High flexibility per grid point
  4. Continuously differentiable

Beyond linear interpolation

1. **Finite element methods** (local)
   1. Piecewise splines
   2. Local basis functions (e.g. B-splines)
2. **Spectral methods** (global basis functions)

1a. Piecewise splines

- **Linear interpolation** can also be written as

$$\begin{aligned}\breve{f}(x;\hat{f}) &= \sum_{n=1}^{\#} \mathbf{1}_{x\in[\hat{x}_n;\hat{x}_{n+1})}\phi_n(x;\hat{f}) \\ &\phi_n(x;\hat{f}) = [a_n + b_n(x-\hat{x}_n)]\end{aligned}$$

where the (a_n, b_n)'s are chosen such that

$$\begin{aligned}\phi_n(\hat{x}_n) &= \hat{f}_n, \forall n \in \{1,2,\ldots,\#\} \\ \phi_n(\hat{x}_{n+1}) &= \hat{f}_{n+1}, \forall n \in \{1,2,\ldots,\#\}\end{aligned}$$

- **Higher order piecewise splines:**

$$\phi_n(x;\hat{f}) = a_n + b_n(x-x_n) + c_n(x-x_n)^2 + \ldots$$

1a. Piecewise cubic splines I

- **Piecewise cubic splines:**

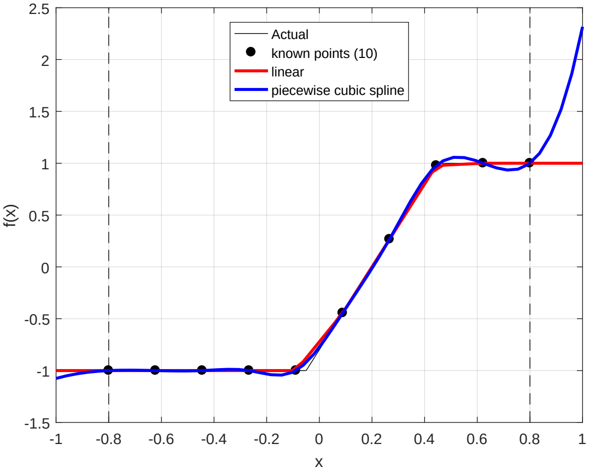
$$\phi_n(x;\hat{f}) = a_n + b_n(x-x_n) + c_n(x-x_n)^2 + d_n(x-x_n)^3$$

where the (a_n, b_n, c_n, d_n)'s are chosen such that

$$\begin{aligned}
\phi_n(\hat{x}_n) &= \hat{f}_n, \forall n \in \{1,2,\dots,\#\} \\
\phi_n(\hat{x}_{n+1}) &= \hat{f}_{n+1}, \forall n \in \{1,2,\dots,\#\} \\
\phi_n'(\hat{x}_{n+1}) &= \phi_{n+1}'(\hat{x}_{n+1}), \forall n \in \{1,2,\dots,\#-1\} \\
\phi_n''(\hat{x}_{n+1}) &= \phi_{n+1}''(\hat{x}_{n+1}), \forall n \in \{1,2,\dots,\#-1\} \\
\phi_1''(\hat{x}_1) &= 0 \,(\text{could be something else}) \\
\phi_n''(\hat{x}_{\#+1}) &= 0 \,(\text{could be something else})
\end{aligned}$$

(number of equations: $2\# + 2(\#-1) + 2 = 4\#$)

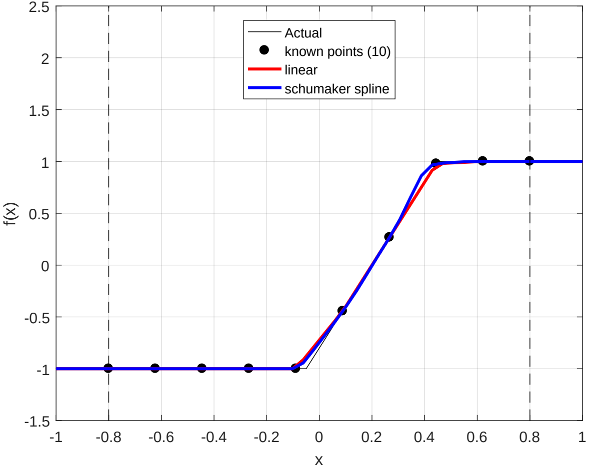
Illustration: Linear and cubic spline



1a. Piecewise cubic splines II

- **Pro piecewise cubic splines:**
  1. No error at known points
  2. More flexible than linear interpolation
  3. Twice continuously differentiable
- **Con piecewise cubic splines:**
  1. Slower than linear interpolation
  2. Not shape-preserving
  3. Poor extrapolation
- **Shape-preserving:** Schumaker (1983)

Illustration: Schumaker (1983) spline



- Discrete choice example on white-board

1b. B-splines

- Until now **piecewise interpolation**:

$$\breve{f}(x) = \sum_{n=1}^{\#} \mathbf{1}_{x \in [x_n; x_{n+1})} \phi_n(x; \hat{f})$$

- **Alternative finite element method:** *Local basis functions*

$$\breve{f}(x) = \sum_{n=1}^{\#} \omega_n(\hat{f}) \phi_n(x)$$

where we note that the basis functions, $\phi_n(x)$, are independent of $\hat{f}$

1b. Linear B-spline

1. **Choose**

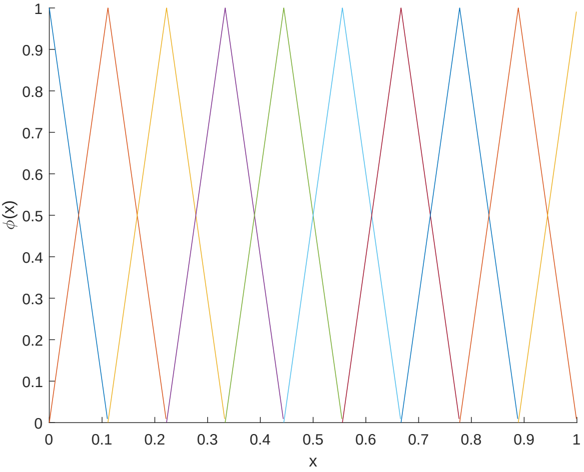
$$
\begin{aligned}
\omega_n(\hat{f}) &= f_n \\
\phi_n(x) &= \begin{cases} 0 & \text{if } x \notin [x_{n-1}, x_{n+1}] \\ \frac{x - x_{n-1}}{x_n - x_{n-1}} & \text{if } n > 1 \text{ and } x \in [x_{n-1}, x_n) \\ \frac{x_{n+1} - x}{x_{n+1} - x_n} & \text{if } n < \# + 1 \text{ and } x \in [x_n, x_{n+1}) \end{cases}
\end{aligned}
$$

2. then this is yet another way to do **linear interpolation**

$$
\begin{aligned}
\breve{f}(x) &= 0 + \cdots + 0 + \hat{f}_n \phi_n(x) + \hat{f}_{n+1} \phi_{n+1}(x) + 0 + \cdots + 0 \\
&= \hat{f}_n \frac{x_{n+1} - x}{x_{n+1} - x_n} + \hat{f}_{n+1} \frac{x - x_n}{x_{n+1} - x_n} \\
&= \hat{f}_n + \frac{\hat{f}_{n+1} - \hat{f}_n}{x_{n+1} - x_n} (x - x_n)
\end{aligned}
$$

3. More general $\phi_n(x) \rightarrow$ higher-order **B-splines**

1b. The $\phi_n(x)$'s for a linear B-spline (10 nodes)



2. Regression with polynomials

- **Interpolant** with P ***global*** **basis functions**

$$\breve{f}(x) = [\phi_1(x)\,\phi_2(x)\,\cdots\,\phi_P(x)]\omega(\hat{f}) = \sum_{i=1}^{P} \omega_i(\hat{f})\phi_i(x)$$

- $\omega(\hat{f})$ can be found by **OLS regression**:

$$\begin{aligned} X &= \begin{pmatrix} \phi_1(\hat{x}_1) & \phi_2(\hat{x}_1) & \cdots & \phi_P(\hat{x}_1) \\ \phi_1(\hat{x}_2) & \phi_2(\hat{x}_2) & \cdots & \phi_P(\hat{x}_2) \\ \vdots & \vdots & \ddots & \vdots \\ \phi_1(\hat{x}_{\#+1}) & \phi_2(\hat{x}_{\#+1}) & \cdots & \phi_P(\hat{x}_{\#+1}) \end{pmatrix} \\ Y &= [\hat{f}_1\,\hat{f}_2\,\ldots\,\hat{f}_{\#+1}]' \\ \omega(\hat{f}) &= [\omega_1\,\omega_2\,\cdots\,\omega_P]' = (X'X)^{-1}X'Y \end{aligned}$$

- **Convention:** $\phi_i(x) = T_i(g(x))$
- **Ordinary polynomials**: $T_i(z) = z^{i-1}$ and $g(x) = x$

2. Regression with Chebyshev polynomials

- **Interval** $x \in [a;b]$**:** Use $g(x) = -1 + 2\frac{x-a}{b-a} \in [0,1]$
- **Chebyshev polynomials**

$$T_i(z) \quad = \quad \cos(i \cos^{-1}(z))$$

- **Orthogonal**

$$\sum_{n=1}^{\#} T_i(z_n) T_j(z_n) \quad = \quad 0 \text{ for } i \neq j$$

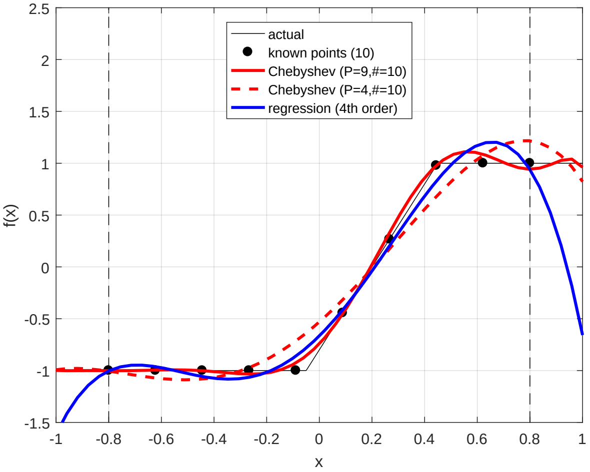
if nodes are chosen as

$$z_n = -\cos\left(\frac{2n-1}{2(\#+1)}\pi\right) \in [-1,1] \text{ for } n = 1,\ldots,\#+1$$

$$x_n = g^{-1}(z_n) = a + \frac{z_n + 1}{2}(b-a)$$

- **Drawback:** Not shape-preserving; can be added at a cost

Illustration: Chebyshev and regression



Multi-dimensional interpolation

- **Linear interpolation:** Simple (next slide)
- **Higher order piecewise splines:** Complicated to find the parameters and no simple shape preserving splines available
- **Basis functions:** Include cross products of one dimensional basis functions
  1. Global polynomial regression
  2. Local B-splines
- **Frontier topics:**
  1. *Global sparse* grids (not all cross products) (Judd et. al. 2014)
  2. *Locally adaptive sparse* grids (+ hierarchy of basis functions) (Brumm and Scheidegger, 2014)
  3. *Scattered* data (triangulation and barycentric interpolation)

Bi-linear interpolation

- For $f : \mathbb{R}^2 \rightarrow \mathbb{R}$ known at the grid points

$$(x_1, y_1), (x_2, y_2), \ldots, (x_{\#+1}, y_{\#+1})$$

1. Locate **neighboring points**

$$\begin{aligned} x_n &\leq x < x_{n+1} \\ y_m &\leq y < x_{m+1} \end{aligned}$$

2. Interpolate in x-dimension for **constant** y

$$\begin{aligned} f_{n,m} &\equiv f(x_n, y_m) + \frac{f(x_{n+1}, y_m) - f(x_n, y_m)}{x_{n+1} - x_n}(x - x_n) \\ f_{n,m+1} &\equiv f(x_n, y_{m+1}) + \frac{f(x_{n+1}, y_{m+1}) - f(x_n, y_{m+1})}{x_{n+1} - x_n}(x - x_n) \end{aligned}$$

3. Interpolate **across** y

$$\breve{f}(x, y) = f_{n,m} + \frac{(f_{n,m+1} - f_{n,m})}{y_{n+1} - y_n}(y - y_n)$$

- **TASK:** illustrate this in 2 dimensions
- Similar in higher dimensions

The Three Curses of Dimensionality

1. **Multiple states:** Exponential growth in total number of grid points for tensor product grids
2. **Multiple choices:** Harder to solve the optimization problem given the states
3. **Multiple shocks:** Exponential growth in the number of quadrature points needed to approximate the continuation value

$\rightarrow$ lots of tips and tricks to alleviate the curse in practice

1. Think!

1. Only put in stuff you need
2. Can you use a discrete state or choice instead of a continuous one?

2. Use analytical structure

1. The problem might contain **intra-temporal sub-problems**, which can be solved fast (e.g. in closed form)

$$\begin{aligned} V_t(M_t) &= \max_{C_t,D_t} u(C_t^\alpha D_t^{1-\alpha}) + \beta \mathbb{E}_t \left[V_{t+1}(M_{t+1}) \right] \\ &\text{s.t.} \\ A_t &= M_t - (p_C C_t - p_D D_t) \\ M_{t+1} &= RA_t + Y_{t+1} \end{aligned}$$

2. There might be freedom of choice wrt. to **state variables**

$$\begin{aligned} V_t(A_{t-1}, Y_t) &= \max_{C_t,D_t} u(C_t^\alpha D_t^{1-\alpha}) + \beta \mathbb{E}_t \left[V_{t+1}(A_t, Y_{t+1}) \right] \\ &\text{s.t.} \\ A_t &= \underbrace{RA_{t-1} + Y_t}_{=M_t} - p_C C_t - p_D D_t \end{aligned}$$

3. The problem might be **scaleable** in a state → the problem can be solved in ratio form with one fewer state variable

3. Taste shocks I

1. **i.i.d. taste shocks** for working or not

$$
\begin{aligned}
V_t(M_t, \varepsilon_t^0, \varepsilon_t^1) &= \max_{L_t \in \{0,1\}} \left\{ v_t(M_t|L_t) + \sigma_\varepsilon \varepsilon_t^{L_t} \right\} \\
v_t(M_t|L_t) &= \max_{C_t} u(C_t, L_t) + \beta \mathbb{E}_t \left[V_{t+1}(\bullet_{t+1}) \right] \\
&\text{s.t.} \\
M_{t+1} &= R(M_t - C_t) + W \cdot L_t
\end{aligned}
$$

2. Assume that ε_t^0 and ε_t^1 are **Extreme Value Type I** then

$$
\begin{aligned}
\mathbb{E}[V_t(M_t, \varepsilon_t^0, \varepsilon_t^1)|M_t] &= \sigma_\varepsilon \log \left(\sum_{j \in \{0,1\}} \exp \left(\frac{v_t(M_t|L_t)}{\sigma_\varepsilon} \right) \right) \\
&\equiv \mathcal{W}_t(M_t)
\end{aligned}
$$

3. Taste shocks II

1. We only need to **find choice-specific value functions**

$$v_t(M_t|L_t) = \max_{C_t} u(C_t, L_t) + \beta \mathbb{E}_t \left[\mathcal{W}_t(M_{t+1}) \right]$$

2. The **choice probabilities** for the discrete choices are

$$\begin{aligned} \Pr(L_t = 1|M_t) &= \Pr(v_t(M_t|1) - v_t(M_t|0) \geq \sigma_\varepsilon(\varepsilon_t^0 - \varepsilon_t^1)) \\ &= \frac{\exp(v_t(M_t|1)/\sigma_\varepsilon)}{\sum_{j\in\{0,1\}} \exp(v_t(M_t|j)/\sigma_\varepsilon)} \end{aligned}$$

- **Question:** What happens as $\sigma_\varepsilon \to 0$ or $\sigma_\varepsilon \to \infty$?

4. Pre-computations

- For the problem

$$
\begin{aligned}
V_t(M_t) &= \max_{C_t} u(C_t) + \beta \mathbb{E}_t \left[V_{t+1}(M_{t+1}) \right] \\
&\text{s.t.} \\
A_t &= M_t - C_t \\
M_{t+1} &= RA_t + Y_{t+1}
\end{aligned}
$$

1. Construct a grid of the **post-decision state** A_t and pre-compute the **post-decision value function**

$$
W_t(A_t) \equiv \mathbb{E}_t \left[V_{t+1}(RA_t + Y_{t+1}) \right]
$$

2. Solve the **simpler problem**

$$
V_t(M_t) = \max_{C_t} u(C_t) + \beta W_t(A_t)
$$

- In **infinite horizon**: Can be a good idea to iterate on W_t instead of V_t (see Hull 2015)

5. Grids and inverse interpolation

- **Grid points should be spend wisely**
  $\rightarrow$ most where there is most curvature
  1. e.g. *Rust-spaced*

$$\begin{aligned} i \geq 2: x_i &= x_{i-1} + \frac{\overline{x} - x_{i-1}}{(n-i+1)^{\phi}} \\ x_1 &= \underline{x} \end{aligned}$$

  2. or *polynomially spaced*

$$x_i = \underline{x} + (\overline{x} - \underline{x}) \left(\frac{i-1}{N} \right)^{\phi}$$

- **Value function inherits curvature of utility function:** Interpolate $u^{-1}(V_t)$, instead of V_t, and convert back. (require that the transformation $u^{-1}(\bullet)$ is monotone)

6. Time-iterations

- We know that optimal interior choices satisfy the **Euler-equation**

$$u'(C_t) = \beta R \mathbb{E}_t \left[u'(C^{\star}_{t+1}(\Gamma(M_t, C_t)) \right]$$

- **Idea:** Find $C^{\star}_t(M_t)$ by **solving the Euler-equation in** $C_t < M_t$, else $C_t = M_t$
  - This can be easier to solve than the optimization problem itself
  - We do not need the value function at all
  - More accurate - why?
- Basis for the ***endogenous grid point method*** we will discuss in a number of lectures

Programming principles

1. **Correct code** beats fast/smart incorrect code
2. **Understandable code** is alfa and omega - others and your future-self need to be able to understand it
   - **Names.** Variables and functions should have precise names
   - **Structure:** Each part of the code should have its own special purpose; preferably numbered. Repeated complex calculations should be in functions.
   - **Comments:** Short and add new information
   - **Testing:** Use `assert(x==1,'error msg.')`
   - **Replicability:** Everything should be called from a single file
3. Code should only be **optimized** when the bottleneck has been located
   - **Time:** `tic; fun(x) toc;`
   - **Profile:** `profile on; fun(x); profile off; profile viewer;`

More: `appendix_good_programming.mlx`
in the MATLAB online course

Optimization principles

1. **Optimize the algorithm you use**. Only calculate what you need and especially avoid repeating expensive operations. Below are some "computational costs"
   - addition, subtraction, comparison = 1
   - multiplication = 4
   - division = 10
   - exp/log= 50
   - power= 100
2. **Tell the computer what you know in advance**
   - pre-allocate memory
3. **Work on consecutive chucks of memory**
   - correct loop order (MATLAB is "column-major": first index as the inner-most loop)
   - vectorize all you can
4. **(Parallize)** (*parfor* in MATLAB)

Do not optimize your code before you are sure it is working and you know where the bottleneck is

Beyond MATLAB

- Numerical precision (floating point arithmetic)
- Especially **loops** are slow in MATLAB
- **Alternatives:**
  1. **`Python`:** Similar speed and complexity (but free)
  2. **`Fortran`:** Very fast, but more complex (easy parallization)
  3. **`C++`:** Very fast, but more complex (easy parallization)
  4. **`Julia`:** Almost as fast as `C++`/`Fortran`, but still simple (still not fully developed)
- `MATLAB` can call e.g. `C++` code using the **mex** interface $\rightarrow$ what I am doing all the time:
  1. `MATLAB` for setup and figures
  2. `C++` with `OpenMP` for parallization of the central stuff
  3. `NLopt` as an optimizer in `C++`

Until next

- **Ensure that you understand:**
  1. Linear interpolation
  2. Interpolation by regression
  3. The curse of dimensionality
  4. The use of taste shocks
  5. The method of time iteration
- Go to **PadLet** and ask or answer a question (https://padlet.com/tjo2/dp)
- **Next time:** Recap! Send me an email (tjo@econ.ku.dk) with stuff you want me to recap.
  Remaining time will be devoted to exercises.