Smooth Methods for the Standard Incomplete Markets Model

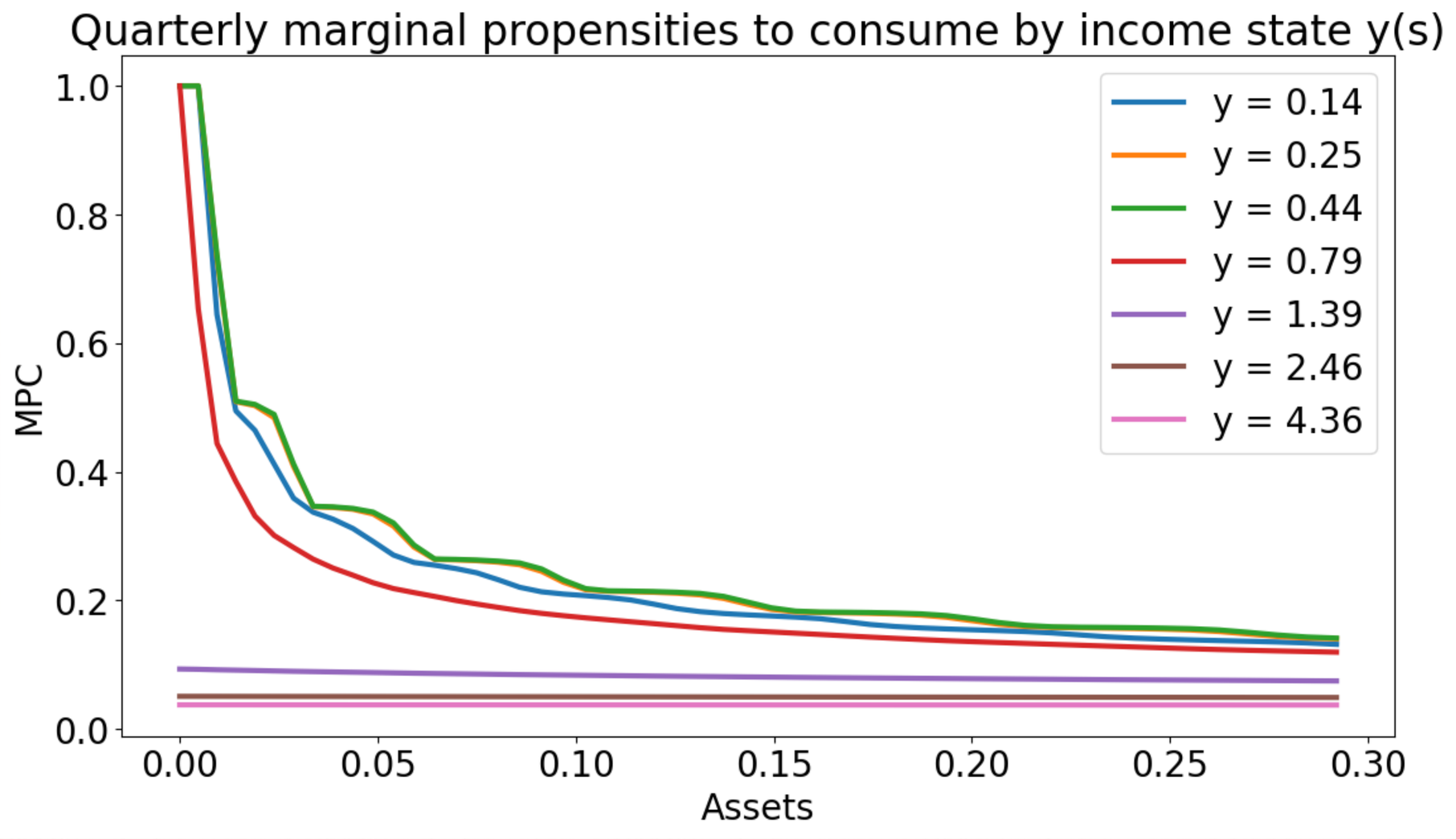
Matthew Rognlie

NBER Heterogeneous-Agent Macro Workshop, 2025

Methods so far: not very smooth!

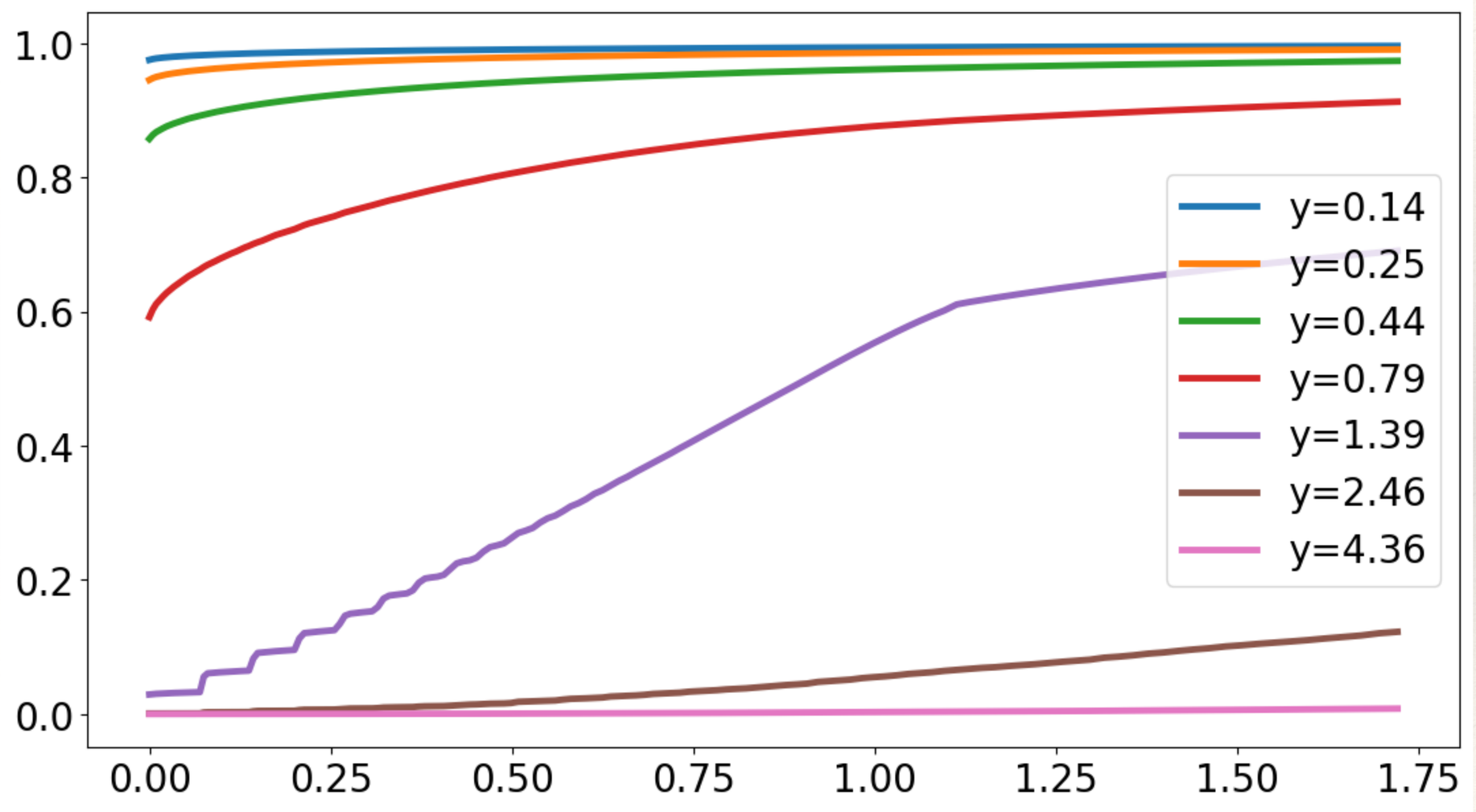
- Discretize income process into **finitely many exogenous states**
- Theoretically, this leads to:
  - Distribution consisting entirely of **mass points**
  - Infinitely many kinks in consumption function **(jumps in MPC)**
- Our solution methods mirror this lack of smoothness:
  - **"Lottery method"** for distribution as set of mass points
  - **Linear interpolation** for marginal value function
- (Silly to use, e.g., smoother interpolation when true solution is non-smooth!)

Example of MPC jumps (from computation supplement)



Jump in MPC from 1 to ~1/2 when going form constrained today (MPC=1) to level of assets where you're likely constrained tomorrow (MPC = ~1/2)

Example of mass points (from computation supplement)



The “stairs" in the CDF for income y=1.39 reflect mass points in true distribution: how many periods have you had y=1.39 and built up assets after starting from the constraint?

(True distribution has only mass points, but lottery method smooths these out.)

Why might this be a problem?

- Jumps in MPCs can be annoying when, e.g., calibrating MPCs
  - Erratic behavior is why solver takes so long to hit our calibration
  - Steady-state behavior in general not always smooth as we vary parameters
- Analytical formulas that should hold don't hold as exactly
  - e.g. perturbing tomorrow's value function should affect today by $dV(e,a) = \beta\mathbb{E}[dV(e',a'(e,a))]$, and roughly does, but imperfect
- Current methods fine for first-order aggregate perturbation, but we **can't do higher-order perturbation**

How do we solve this?

- Need smoother income process!
- Could avoid discretization entirely, and use true lognormal income process
  - Downside: adds continuous state to model
- Alternative: keep discretized persistent income process, but **add smooth iid component**
  - Since it's iid, doesn't make the state space larger
  - If component large enough, solution well-behaved!

Smooth model: theory and implementation of backward iteration

Bellman equation almost unchanged

- Bellman equation looks **almost** exactly the same

$$V(e,a) = \max_{c,a'} u(c) + \beta \mathbb{E}[V(e',a') \mid e]$$

$$\text{s.t. } a' + c = (1+r)a + Z\xi e \qquad\qquad a' \geq \underline{a}$$

- Only difference is that we have an **iid shock ξ multiplying income**
- Need to **take expectations** over this ξ
  - Can use numerical integration, but carefully, because V_a is kinked

Breaking Bellman into conceptual steps

- Step 1: discounting and expectations (unchanged!)

$$W_t(e, a') = \beta \mathbb{E}_{e'|e}[V_{t+1}(e', a')]$$

- Step 2: solve optimal consumption-savings given **"cash on hand"**

$$\mathscr{V}_t(e, coh) = \max_{a' \geq \underline{a}} u(coh - a') + W_t(e, a')$$

- Step 3: take expectations over cash on hand to recover new value function

$$V_t(e, a) = \mathbb{E}_{\xi} \mathscr{V}_t \left(e, (1 + r_t^p)a + Z_t \xi e \right)$$

Understanding consumption-savings step

- For simplicity, drop e in step 2 since it's parallel in e:

$$\mathcal{V}_t(coh) = \max_{a' \geq \underline{a}} u(coh - a') + W_t(a')$$

- We iterate over marginal values, and envelope condition $\mathcal{V}'_t(coh) = u'(c)$ means we just need **consumption policy** $c = coh - a'$
- Characterized by first-order condition $u'(c) \geq W'(a')$
- Can invert FOC to get c for each a', giving consumption at "endogenous gridpoints" $coh = c + a'$
- Interpolate these observations (coh, c) to get spline for consumption $c(coh)$

Code implementing this step (smooth_sim.py)

```
# Part 2: mapping from endogenous coh grid to consumption
(constrained below coh_endog[0])
c_endog = Wa**(-eis)
coh_endog = c_endog + a_grid

q = np.empty_like(Va)
for s in range(len(y)):
    q[s] = spline.interp(coh_endog[s], c_endog[s])
```

Endogenous gridpoint *coh* corresponding to $a' = \underline{a}$ is **exactly** the cash-on-hand where constraint stops binding.

Cubic spline in "q" represents consumption above that kink; below that kink, consumption is mechanical, $c = coh$

Expectations step

- Step 3: take expectations over cash on hand to recover new value function

$$V_t(e,a) = \mathbb{E}_\xi \mathcal{V}_t(e, \underbrace{(1+r_t^p)a + Z_t \xi e}_{=coh})$$

- What distribution for mean-1 iid shock ξ?
  - Common choice in income process would be lognormal
  - We choose lognormal plus constant, so that some income is guaranteed
  - Why? Subtle: don't want "natural borrowing constraint" to be 0, otherwise no one would ever reach constraint

Expectations step, continued

- We iterate on **derivatives** with respect to *coh* or *a* (drop *t* for simplicity)

$$V_a(e,a) = (1+r^p)\cdot \mathbb{E}_{coh}\mathscr{V}_{coh}(e,coh)$$

- How do we take the expectation?
- Integrate **separately on two sides of kink** coh^* **where constraint hits**:

$$\mathbb{E}_{coh}\mathscr{V}_{coh}(e,coh) = \int_0^{coh^*} f(coh)u'(coh) + \int_{coh^*}^{\infty} f(coh)u'(c(e,coh))$$

- Here, $c(e,coh)$ is from policy function spline we just obtained

Code implementing this step (1/2)

```
# Part 3: integrate over lognormal part of income to get Va(s, a)
coh_certain, coh_lognormal_mu = coh_components(a_grid, y, r, sigma, share)

Va = np.empty_like(Va)
for s in range(len(y)):
    for a in range(len(a_grid)):
        Va[s, a] = expectation_lognormal_coh(coh_certain[s, a],
                                             coh_endog[s], q[s], coh_lognormal_mu[s], sigma, eis)

# Scale by 1+r to reflect returns
Va *= (1+r)
```

"share" is minimum value of ξ, which is share plus 1-share of lognormal with sd sigma. Together with assets, this makes "certain" component of cash on hand

At each point in space, then need to take expectations with respect to this lognormal.

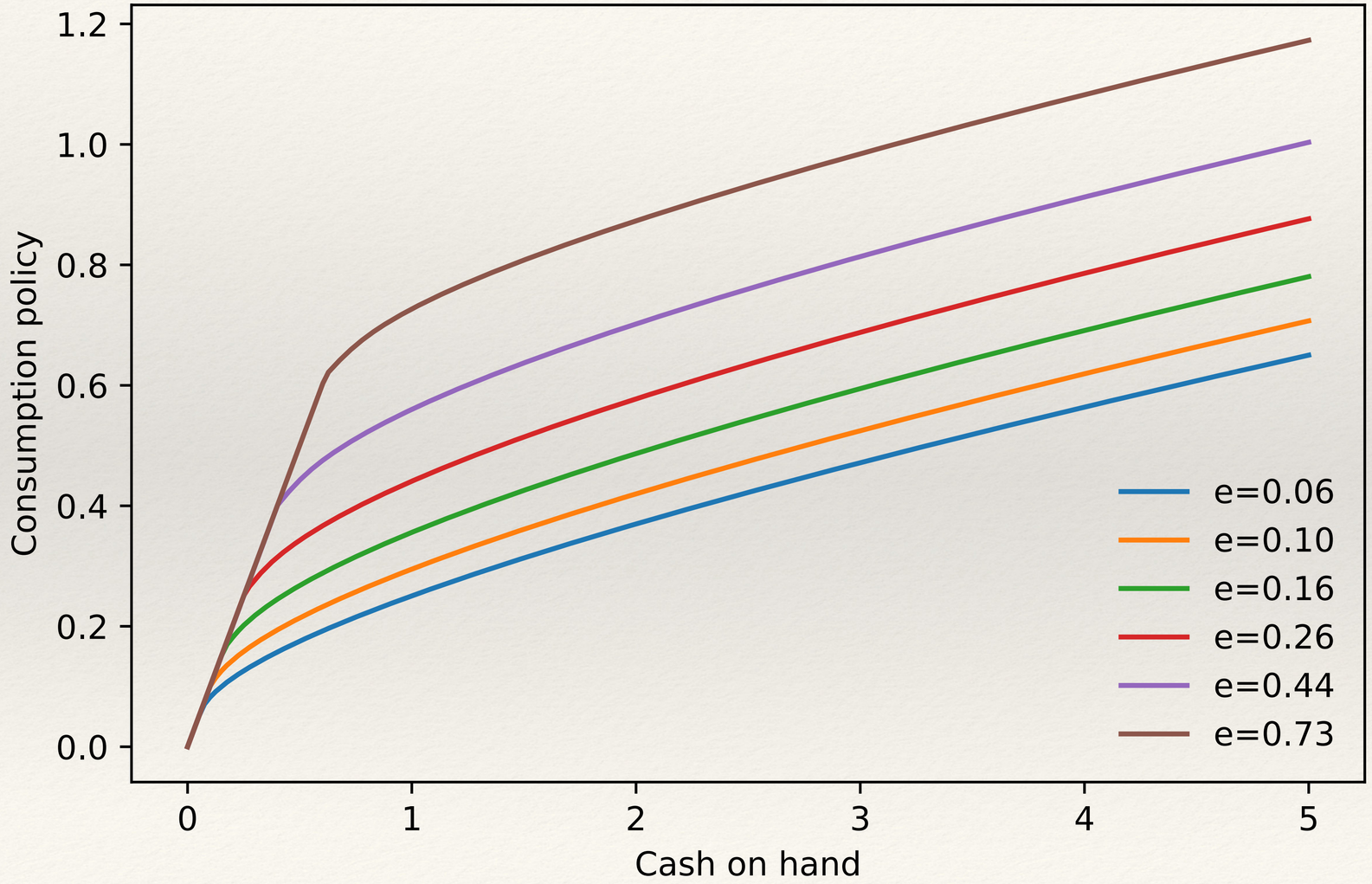
Code implementing this step: part (2/2)

```
@njit
def expectation_lognormal_coh(coh_certain, coh_grid, q, mu, sigma, eis):
    """Take expectations of marg utility at coh_certain over lognormal part
    of cash-on-hand, given spline q on coh_grid for unconstrained consumption"""
    # constrained part: marginal utility is coh**(-1/eis)
    w, x = utils.integrate_lognormal_interval(coh_certain, mu, sigma, 0, coh_grid[0])
    constrained_part = w @ x**(-1/eis)

    # unconstrained part: marginal utility is q(coh)**(-1/eis)
    w, x = utils.integrate_lognormal_interval(coh_certain, mu, sigma, coh_grid[0], np.inf)
    unconstrained_part = w @ spline.val_monotonic(q, coh_grid, x)**(-1/eis)

    return constrained_part + unconstrained_part
```

Below minimum endogenous gridpoint coh^*, $c = coh$, so just integrate that. Above, we integrate spline. **Need to integrate separately on both sides of kink**. Otherwise, with standard quadrature, won't be smooth.

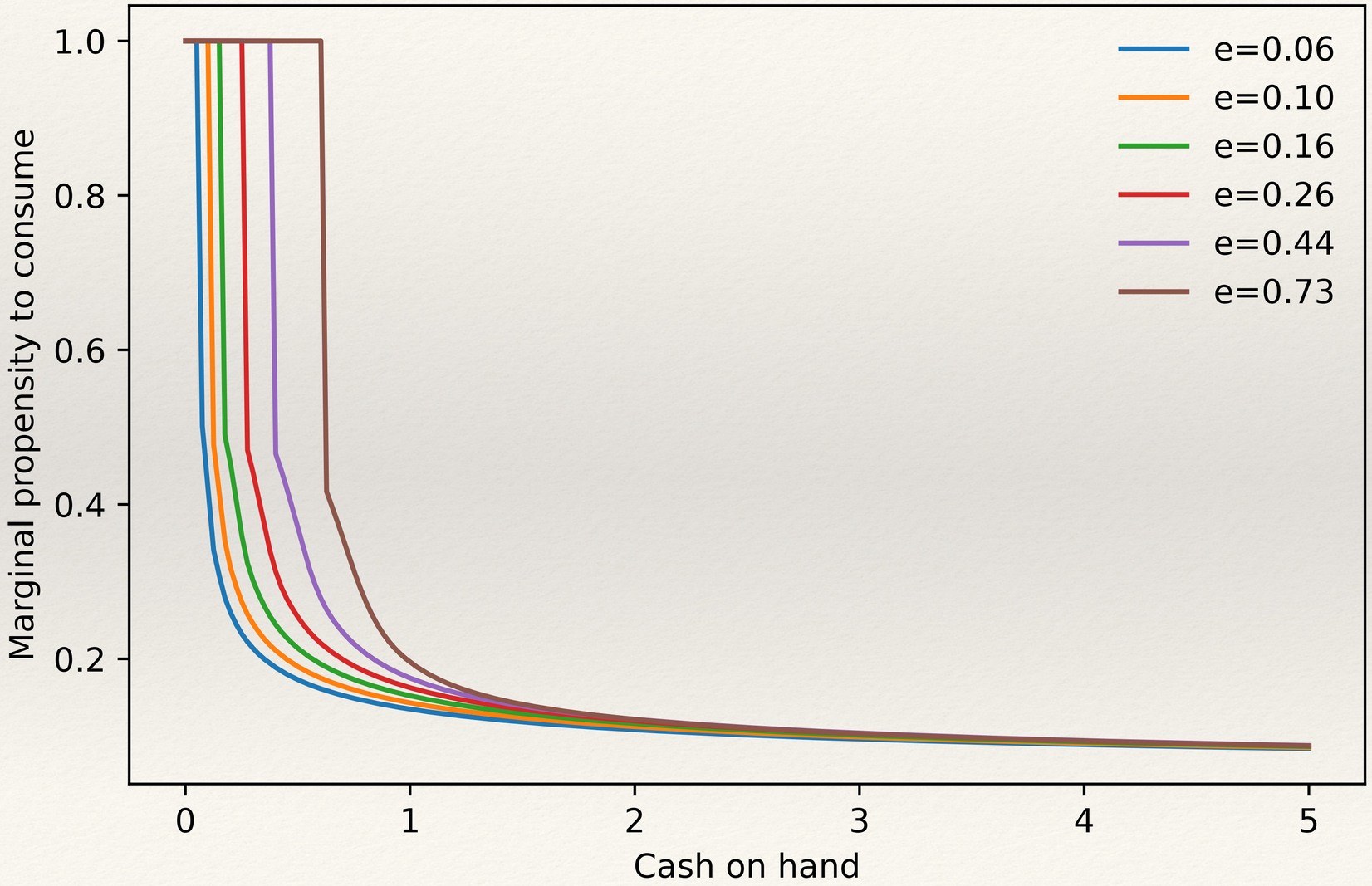
Brute-force here: using weights and nodes implied by **Gauss-Legendre quadrature**. (Could try more efficient , specialized quadrature. But adaptive quadrature provided by SciPy would be too slow.

Results: consumption policy in cash-on-hand



Still a kink in consumption, but after that it's smooth—no "secondary kinks" implied by anticipation of hitting constraint tomorrow

Results: smooth MPC away from constraint



MPCs jump down from 1 to less than 1 when constraint stops being binding, but after that they're smooth—no further jumps!

Smooth model: distribution and forward iteration

Tracking the distribution

- Also need to keep track of the distribution
  - Lottery method with mass points—not very smooth
- What to use now? Smooth **cumulative distribution functions (CDFs)**!
- These will start above zero, reflecting mass point at constraint
- But above that, they should be smooth!

Iterating on the distribution

- Go in opposite order to before, **forward** in time
- Three steps:
  - Iterate forward from assets a to cash-on-hand coh, given iid ξ
  - Iterate forward from cash-on-hand coh to asset policy a'
  - Iterate forward on exogenous state e to e' [almost unchanged]
- Awesome fact about second step: we can **use the endogenous gridpoints we already have!** [Closely related to Bayer, Luetticke, Weiss, Winkelmann]
  - CDF over endogenous gridpoints coh is the CDF over a'

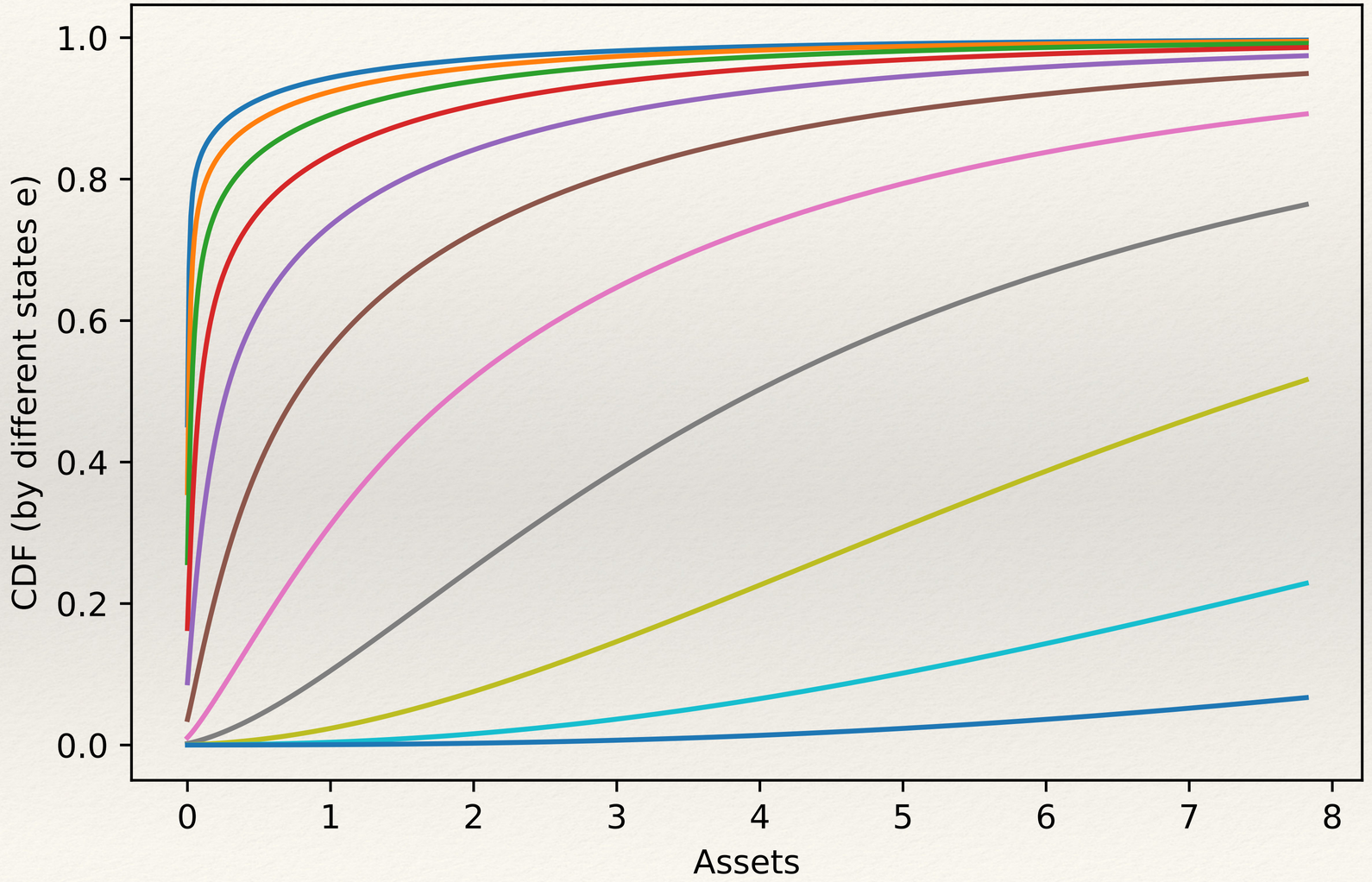
Overall distribution iteration code

```
def forward_iteration(F, coh_endog, Pi_F, a_grid, y, r, sigma, share):
    return Pi_F @ forward_policy(F, coh_endog, a_grid, y, r, sigma, share)


@njit
def forward_policy(F, coh_endog, a_grid, y, r, sigma, share):
    coh_certain, coh_lognormal_mu = coh_components(a_grid, y, r, sigma, share)

    Fnew = np.empty_like(F)
    for s in range(len(y)):
        qF = spline.interp(a_grid, F[s])
        # get CDF on coh_endog, which maps directly to CDF on assets
        Fnew[s] = iteration_lognormal_coh(qF, coh_certain[s], coh_endog[s],
        coh_lognormal_mu[s], sigma)

    # ensure that near the top of the distribution, we have exactly 1
    w = utils.smooth_weight(a_grid)
    Fnew = (1-w)*Fnew + w

    return Fnew
```

Step 1 (on iid shock) is mechanical integration

```
@njit
def iteration_lognormal_coh(qF, coh_certain, coh_endog, mu, sigma):
    """Iterate forward distribution from CDF spline qF on coh_certain,
    integrating over lognormal part of cash-on-hand to get CDF on coh_endog"""
    Fnew = np.zeros_like(coh_endog)
    for i, coh in enumerate(coh_endog):
        if coh >= coh_certain[0]:
            w, x = utils.integrate_lognormal_interval(coh_certain[0], mu, sigma,
            coh_certain[0], coh)
            Fvals = spline.val_monotonic(qF, coh_certain, (coh + coh_certain[0] - x)
            [::-1])[::-1].copy()
            Fnew[i] = w @ Fvals

    return Fnew
```

Resulting CDF of assets



Conclusion

Smooth model

- Figured out the basics of a “smooth” model, with smooth CDFs over assets and a smooth policy function (except at kink)
- Transitions, fake news algorithm, etc. work much the same as before
- Foundation of higher-order perturbation analysis (next lecture!)
- Analytical formulas hold much more exactly

- Much room for improvement!