Modular Bellman Dynamic Programming

Dyn-X: Theory to High-Dimensional Implementation

Chris Carroll (*JHU*) Alan Lujan (*JHU*)
Akshay Shanker (*UNSW/Econ-ARK*) Matt White (*Econ-ARK*)

CEF 2025

July 2025

Roadmap

1. **Problem:** Monolithic DP code
2. **Core Idea:** Factored Bellman → Modular CDA decomposition
3. **Implementation:** `perches`, `movers`, and `stages`
4. **Case Study:** Housing model with branching
5. **Future:** Implementation, open questions & roadmap

Welcome to the 21st Century

We have clean and **modular theory** with all primitives.

$$\mathrm{v}_{t+1} = \mathbb{T}\mathrm{v}_t, \qquad \mathrm{v}_t, \mathrm{v}_t \in \mathcal{B}(X)$$

What happens when we are tasked to compute and estimate models with policy or business applications specific features?

Why Is Dynamic-Programming Code Still Monolithic?

Introduction

- **Ad hoc, one-off, copy-paste implementations** → near-zero reuse.
- **Curse of dimensionality** × **curse of idiosyncratic engineering**: each new model is bespoke with handcrafted hacks.
- **Implications for supercomputing**: Solution structure matters.
- *Powell (2012)*: introducing a post-decision state can "*dramatically simplify*" dynamic programs — yet frameworks rarely expose it.

Traditional Monolithic DP

```python
# All-in-one nested loops
for t in range(T-1, -1, -1):
  for i, a in enumerate(asset_grid):
    best_val = -np.inf
    for c in np.linspace(0, a, 101):
      # Utility + transition mixed
      u = np.sqrt(c)
      a_next = a - c
      j = int(round(a_next))
      # Bellman buried in loops
      val = u + beta * V[t+1, j]
      if val > best_val:
        best_val = val
    V[t, i] = best_val
```

Extreme Monolith:

- Triple nested loops
- State dynamics entangled with optimization
- Hard to modify or extend
- No real operator representation

Our premise

Operator theoretic approach reduces mathematical complexity, it should also reduce computational complexity.

Modular Approach


```python
# Separate operators for each decision
def cntn_to_dcsn(EV_next, model):
  # Backward: optimize consumption
  ...
  return V_dcsn

def dcsn_to_arvl(V_dcsn, model):
  # Backward: integrate over shocks
  ...
  return V_arvl

# Clean operator composition
T_stage = dcsn_to_arvl(cntn_to_dcsn(...))
```

Bellman factored: $\mathscr{T}^{F} = \mathscr{T}^{va} \circ \mathscr{T}^{ev}$

Main ideas:

- Recursive problem decomposed into three operations.
- Explicitly define operations as flexible computational objects and remove reference to time within stage.
- Compose operators into canonical stages.
- Dynamic program turns out to be a graph where operator arguments are nodes (`perches`) and operations are edges (`movers`)

From Primal to Factored Bellman

Primal: Mixes action with future shock

$$V_{t+1}(x) = \max_{a \in A(x)} \Big\{ r(x, a) + \beta\, \mathbb{E}\big[V_t(f(x, a, W_{t+1}))\big] \Big\}$$

where V_{t+1} is the value function at time $t+1$ and V_t is the value function at time t, $A(x)$ is the action set at state x, $r(x, a)$ is the reward function, $f(x, a, W_{t+1})$ is the transition function, and W_{t+1} is the shock at time $t+1$.

Factored: Separate transitions and continuation value

$$x_e = g_{ve}(x_v, a), \quad x_v' = g_{av}(x_e, W'), \quad \mathscr{E}(x_e) = \mathbb{E}_{W'}[\, V(x_v') \,]$$

$$V(x_v) = \max_a \{\, r(x_v, a) + \beta\, \mathscr{E}\big(g_{ve}(x_v, a)\big) \,\}$$

Key: Post-Decision State

x_e = Powell's post-decision state — no nested expectation inside max

CDA State Types

Label	Timing	Carries value
Arrival x_a	start (pre-shock)	$\mathscr{A}(x_a)$
Decision x_v	post-shock	$\mathscr{V}(x_v)$
Continuation x_e	post-action	$\mathscr{E}(x_e)$

Operators: $\mathscr{T}^{ev}$ (optimise), $\mathscr{T}^{va}$ (expectation).

Theory Meets Implementation

Mathematical Decomposition:

$$\boxed{\mathscr{T}^F = \mathscr{T}^{va} \circ \mathscr{T}^{ev}}$$

$$\begin{cases} (\mathscr{T}^{ev}\mathscr{E})(x_v) = \max_a\{r + \beta\mathscr{E}\} \\ (\mathscr{T}^{va}\mathscr{V})(x_a) = \mathbb{E}_W[\mathscr{V}(g_{av})] \end{cases}$$

Computational Needs:

- Store functions & policies
- Swap solvers (VFI/EGM)
- Connect stages
- Enable simulation

Solution

Map mathematical objects directly to computational structures

`Perches` & `Movers`: Computational Objects

`Perches` **(nodes):**

- `arvl`: $\mathscr{A}$, μ_a
- `dcsn`: $\mathscr{V}$, policy, μ_v
- `cntn`: $\mathscr{E}$, μ_e

Each `perch` stores `.sol`, `.dist`, `.grid`

`Movers` **(edges):**

- Backward: $\mathscr{T}^{ev}$, $\mathscr{T}^{va}$
- Forward: push distributions
- Swap solvers via config

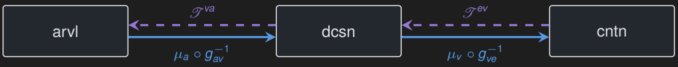
CDA Graph Template

A stage is a union of two direct acyclic graphs (DAGs) with `perches` as nodes and `movers` as edges.

```
arvl --shock--> dcsn --action--> cntn
 ^                             |
 |------- Bellman backward -----|
```

Graph *is* the composition $\mathscr{T}^{va} \circ \mathscr{T}^{ev}$.

CDA Cycle — One `Stage`



Dashed = Bellman (value) recursion; solid = push-forward of measures. This template underlies every Dyn-X `stage`.

`Stage` Lifecycle

1. Load YAML $\Rightarrow$ symbolic model
2. Compile $\Rightarrow$ NumPy/JIT functions & grids
3. Solve backward via **Horse** (solver engine) + **Whisperer** (orchestrator)
4. Simulate forward (horse)

Clean Separation

- **Model specification** (YAML/config) is separate from **solution algorithm**
- Horse = pluggable solver (VFI, EGM, etc.)
- Whisperer = coordinates solving across `stages`

GPU in Dyn-X: Why It's Almost Free

Implementation

- **Thin operators** each mover $\approx$ one NumPy/Numba kernel $\Rightarrow$ launch exactly the same kernel on a GPU array
- **No global state** per-stage data live in `perch.sol` $\Rightarrow$ host $\leftrightarrow$ device copy is local, automated
- **Zero code change** Switch backends via config: CPU $\leftrightarrow$ GPU $\leftrightarrow$ MPI Unlike ad-hoc GPU implementations
- **Tailored kernels** Custom CUDA for Bellman max (not generic autodiff) Handles non-differentiable policies, discrete choices

Outcome in the housing benchmark (10k wealth × 30 housing × 5 income points):

$$\text{run-time: 74 s (CPU)} \quad \rightarrow \quad \text{2.6 s (1 V100)}$$

$\sim$**30× speedup** — comparable to specialized GPU codes, but modular

YAML Excerpt

```yaml
state_space:
  decision: [assets, y, house]
  arrival:  [assets, house]
functions:
  utility: "alpha*np.log(c)+(1-alpha)*np.log(kappa*(house+1))"
```

StageCraft infers the CDA graph automatically.

Five Housing Stages

1. TENU — tenure choice (branching)
2. OWNH — owner housing stock
3. RNTH — renter housing services
4. OWNC — owner consumption (EGM)
5. RNTC — renter consumption (EGM)

Branching Bellman — Tenure Choice

Decision state (a, H, y):

$$V_v(a, H, y) = \max\Big\{ V_e^{\text{rent}}\big((1+r)a + H, y\big),\ V_e^{\text{own}}(a, y, H) \Big\}.$$

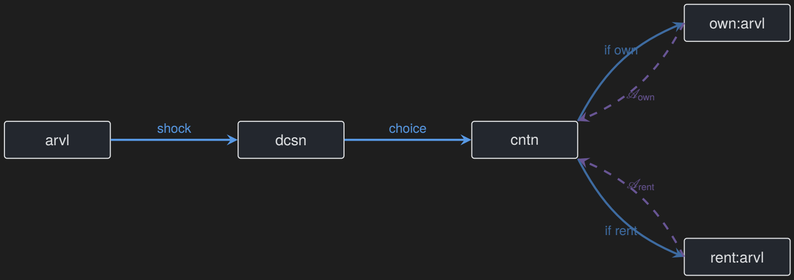
Arrival update:

$$V_a(a, H, \bar{y}) = \mathbb{E}_\xi\big[V_v\big(a, H, f(\bar{y}, \xi)\big)\big].$$

- Rent branch liquidates housing $\Rightarrow w = (1+r)a + H$.
- Both continuations stored in `cntn.sol` under keys `"rent"`, `"own"`.

Branching `Stage` Graph (single `cntn` perch)

Case Study: Housing Model



Forward: Single path within `stage`, branches after `cntn`.
Backward: Multiple value functions aggregate at `cntn`.

Discrete Choice — Renter Housing

Decision state (w, y) with discrete grid $\mathbb{H}$:

$$\mathscr{T}^{ev}\mathscr{E}(w, y) = \max_{\substack{S \in \mathbb{H} \\ P^r S \leq w}} \mathscr{E}\big(S, y, w - P^r S\big).$$

Budget feasibility $P^r S \leq w$ is enforced in the choice set.

YAML Snippet — Discrete Choice


```yaml
dcsn_to_cntn:
  type: forward
  method: discrete_choice
  choice_grid:
    S: {type: linspace, min: S_min, max: S_max, points: S_pts}
constraints:
  - "P_r*S <= w"
```

The backward `cntn_to_dcsn` mover enumerates the S-grid (stored in `cntn.grid["S"]`) and takes the element-wise max.

Solver Toolkit & Simulation

Solver Methods:

- **VFI** + GPU (30× speedup)
- **EGM** (Carroll 2006)
- **DCEGM** (Iskhakov et al. 2017)
- **FUES** (4× fewer grid points)

Forward Simulation:

`arvl.dist` → `dcsn.dist` → `cntn.dist`

Mass preserved at each step

Key: Swap solvers via config, not code

Why Not Use Existing Tools?

Generic ML Frameworks:

- TensorFlow/JAX: Arrays & ops
- No economic primitives
- Generic autodiff $\neq$ EGM

Economics Libraries:

- HARK: Modular but not graph
- Dolo: DSL but no factoring
- QuantEcon: Solvers not stages

Dyn-X: Economic objects as nodes, operators as edges

Open Questions & Future Work

Core conjectures:

- **Universality:** Any MDP $\rightarrow$ CDA form?
- **Minimality:** Are irreducible stages truly minimal?
- **Edge cases:** Simultaneous shocks, non-period models

Scaling up:

- **Heterogeneous agents:** Multiple circuits in parallel
- **General equilibrium:** Link circuits via market clearing
- **Infinite horizon:** Period structure with stationary shocks

Empirical validation: 30× speedup on housing model

Roadmap

- **GPU via Numba CUDA** (implemented, 30× speedup)
- **MPI** for distributed computing
- **JAX/cuPy backends** (planned)
- **Lean-4 export** for formal verification

Dyn-X 1.0 beta: Q4 2025 — github.com/econ-ark/dynx

Key Takeaways

- **Problem:** Monolithic DP code — no reusability
- **Solution:** Factored Bellman → CDA stages
- **Implementation:** Graph of economic objects
- **Performance:** 30× speedup with GPU

Impact

Focus on economics, not implementation
Dyn-X 1.0 beta: Q4 2025

Thank You

Questions & discussion.

Slides & code → `github.com/akshay-shanker/dynx-cef-2025`

Modular MDP — Formal Definition

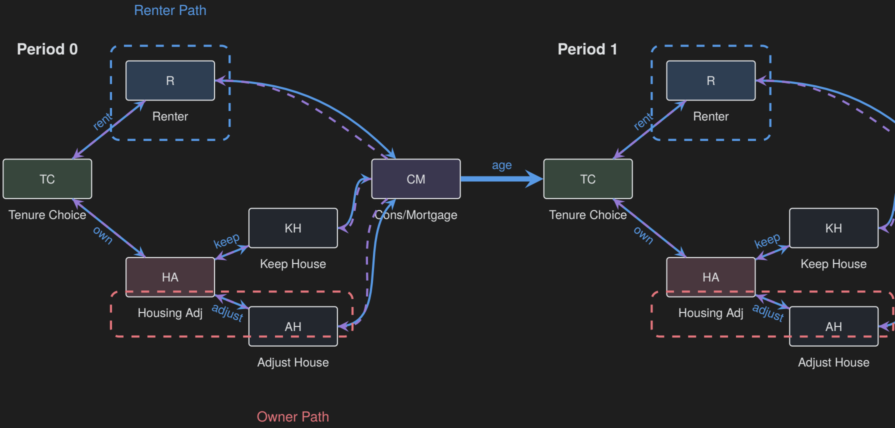
Tuple

$$((\Omega, \Sigma, \mathbb{P}), J, \{\mathcal{X}_j\}, \{\mathcal{F}_{\mathcal{X}_j}\}, \{\mathfrak{S}_j, \tilde{\mathfrak{S}}_j\}, \{\mathrm{CDC}_j\})$$

1. J: stage indices (finite or countable)
2. $\mathcal{X}_j$: topological vector space of states
3. $\mathcal{F}_{\mathcal{X}_j} \subseteq \mathscr{M}(\mathcal{X}_j, \mathbb{R})$
4. Shocks $\mathfrak{S}_j$ (observable) and $\tilde{\mathfrak{S}}_j$ (latent)
5. $\mathrm{CDC}_j : \mathcal{F}_{\mathcal{X}_{j+1}} \to \mathcal{F}_{\mathcal{X}_j}$

Stage j is *irreducible* if $\sigma(\mathfrak{S}_j)$ and $\sigma(\tilde{\mathfrak{S}}_j)$ admit no non-trivial independent sub-algebras.

Complex Example: Mortgage Model with Sub-Circuits



TC = Tenure Choice, R = Renter, HA = Housing Adjustment, KH = Keep House, AH = Adjust House, CM = Consumption/Mortgage
Two periods showing within-period sub-circuits: renter path (blue) and owner path (red) can be solved in parallel.