

# Ben's Machine - Polarization dynamics model



okay, so we're going to create a python model of the dynamics of popular polarization over time, from around the end of the civil war, through the present. We'll create a feedback model that predicts scotus polarization from public polarization, and then updates public polarization based on the scotus makeup... sort of a predator prey model where public polarization and scotus polarization feed off one another, and then we do things like sweep the term limits for "justices" we'll use the actual history of presidential party as the ground truth. So, (just spitballing) we start out with more-or-less non-polarization, then the 1900 president is elected (whatever party they were ... I'm making this up, although it would be trivial to look up, let's say R), so then whatever scotus openings come up end up with Rs, and then public polarization follows the SCOTUS ratio, then the next president (D, let's say) gets a D into whatever scotus seats there are, and so on. And we plot polarization, and vary scotus term limits. (Actually, I'll bet I could get ChatGPT to code this up very easily!)

Love this idea — let's spin up a clean, hackable simulator you can drop into your workflow and sweep term-limits, noise, and coupling. Below is a single self-contained Python script that:

- models **Public polarization**  $P_t \in [-1, 1]$  and **SCOTUS polarization**  $S_t \in [-1, 1]$ ,
- represents the Court as 9 "justices" with party-lean  $\{-1, +1\}$  (D = -1, R = +1 by default — you can flip that if you like),
- fills vacancies from the **current President's party** using either a provided **historical presidents CSV** or an auto-generated toy timeline (alternating every 4 years),
- supports **fixed term limits** (e.g., 18 years) or "lifetime" (None),
- evolves  $P_t$  by a **feedback rule** (predator-prey-ish):  

$$P_{t+1} = (1 - \lambda) P_t + \lambda \tanh(\gamma S_t) + \sigma \varepsilon_t$$
- computes  $S_t$  as the mean party-lean across seated justices,
- can **sweep term limits** and write tidy **TSVs**,
- includes a simple **matplotlib** plot helper.

You can paste this into `polarization_sim.py` and run.

```
python

#!/usr/bin/env python3
# polarization_sim.py
# A minimal simulator for coupled public ↔ SCOTUS polarization dynamics.
# Jeff-ready: simple, explicit, and sweepable.
```

```

import argparse, csv, math, random, statistics
from dataclasses import dataclass, field
from typing import List, Optional, Dict, Tuple
import sys
import pathlib

try:
    import matplotlib.pyplot as plt
except Exception:
    plt = None # plotting optional

# -----
# Conventions
# -----
# Parties encoded as: D = -1.0, R = +1.0
# Public polarization P_t in [-1, 1], SCOTUS polarization S_t in [-1, 1] (mean justice lean)
# Time step = 1 year (can change if you want terms in years anyway)

PARTY_TO_LEAN = {"D": -1.0, "R": +1.0}

@dataclass
class Justice:
    appointed_year: int
    term_limit_years: Optional[int] # None = lifetime
    lean: float # -1 (D) to +1 (R)
    # You can add: drift, age, strategic retirement probability, etc.

    def expires_in_year(self, year: int) -> bool:
        if self.term_limit_years is None:
            return False
        return (year - self.appointed_year) >= self.term_limit_years

@dataclass
class Court:
    size: int = 9
    term_limit_years: Optional[int] = None
    justices: List[Justice] = field(default_factory=list)

    def polarization(self) -> float:
        if not self.justices:
            return 0.0
        return statistics.fmean(j.lean for j in self.justices)

    def step_year(self, year: int, president_party: str, vacancies_this_year: int = 0,
                  retire_prob: float = 0.0) -> None:
        """Advance one year: natural expirations + stochastic retirements, then fill
        vacancies."""
        # Natural expirations (term limits)
        remaining = []
        for j in self.justices:
            if j.expires_in_year(year):
                vacancies_this_year += 1
            else:
                remaining.append(j)

        # Optional stochastic retirements (strategic exits, health, etc.)
        # You can make this a function of public/court polarization if you want feedback.
        really_remaining = []
        for j in remaining:
            if retire_prob > 0.0 and random.random() < retire_prob:
                vacancies_this_year += 1
            else:
                really_remaining.append(j)

        self.justices = really_remaining

        # Fill to size with current president's party lean
        while len(self.justices) < self.size:
            self.justices.append(
                Justice(
                    appointed_year=year,
                    term_limit_years=self.term_limit_years,
                    lean=PARTY_TO_LEAN[president_party]
                )
            )

```

```
def load_presidents_from_csv(csv_path: pathlib.Path) -> Dict[int, str]:
    """
    Load presidents timeline from CSV with columns:
    start_year, end_year, party
    party is "D" or "R" (case-insensitive).
    Returns dict: year -> party for all years [start_year, end_year] inclusive.
    """
    mapping: Dict[int, str] = {}
    with csv_path.open(newline='') as f:
        reader = csv.DictReader(f)
        for row in reader:
            sy, ey = int(row["start_year"]), int(row["end_year"])
            party = row["party"].strip().upper()
            assert party in ("D", "R"), f"Invalid party {party}"
            for y in range(sy, ey + 1):
                mapping[y] = party
    return mapping

def make_alternating_presidents(start_year: int, end_year: int,
                               first_party: str = "R", term_years: int = 4) -> Dict[int, str]:
    """
    Simple fallback timeline: alternate parties every 'term_years' starting at start_year.
    """
    mapping: Dict[int, str] = {}
    parties = ["R", "D"] if first_party.upper() == "R" else ["D", "R"]
    block = 0
    for y in range(start_year, end_year + 1):
        mapping[y] = parties[(y - start_year) // term_years % 2]
    return mapping

@dataclass
class SimParams:
    start_year: int = 1866
    end_year: int = 2025
    public_alpha: float = 0.15 # λ: how quickly public responds
    public_gain: float = 1.50 # γ: how strongly court polarization shapes public (inside tanh)
    noise_sigma: float = 0.03 # σ: public noise
    term_limit_years: Optional[int] = 18
    retire_prob: float = 0.00 # optional random retirements per justice-year
    initial_public: float = 0.0 # start "non-polarized"
    court_size: int = 9
    seed: int = 42

@dataclass
class SimResult:
    years: List[int]
    public: List[float]
    scotus: List[float]
    presidents: List[str]

def clamp(x: float, lo=-1.0, hi=1.0) -> float:
    return max(lo, min(hi, x))

def simulate(params: SimParams,
             presidents_by_year: Dict[int, str]) -> SimResult:
    random.seed(params.seed)

    court = Court(size=params.court_size, term_limit_years=params.term_limit_years)
    # Bootstrap: fill Court in start_year with the then-president's party
    pres0 = presidents_by_year.get(params.start_year, "R")
    court.step_year(params.start_year, pres0, vacancies_this_year=params.court_size)

    P = params.initial_public
    years: List[int] = []
    publics: List[float] = []
    scoti: List[float] = []
    pres: List[str] = []

    for year in range(params.start_year, params.end_year + 1):
        president_party = presidents_by_year.get(year, pres0)
        # Court evolves first (appointments/expiration at 'year')
        court.step_year(year, president_party, vacancies_this_year=0,
                        retire_prob=params.retire_prob)
```

```

S = court.polarization() # mean lean in [-1, 1]

# Public responds to court (feedback)
#  $P_{t+1} = (1-\lambda) P_t + \lambda * \tanh(\gamma * S_t) + \text{noise}$ 
influence = math.tanh(params.public_gain * S)
noise = random.gauss(0.0, params.noise_sigma)
P = (1.0 - params.public_alpha) * P + params.public_alpha * influence + noise
P = clamp(P)

years.append(year)
publics.append(P)
scoti.append(S)
pres.append(president_party)

return SimResult(years=years, public=publics, scotus=scoti, presidents=pres)

def write_tsv(path: pathlib.Path, sim: SimResult) -> None:
    with path.open("w") as f:
        f.write("year\tpresident_party\tpublic_polarization\tscotus_polarization\n")
        for y, pr, P, S in zip(sim.years, sim.presidents, sim.public, sim.scotus):
            f.write(f"{y}\t{pr}\t{P:.6f}\t{S:.6f}\n")

def plot_series(sim: SimResult, title: str = "Polarization over time"):
    if plt is None:
        print("matplotlib not available; skip plotting.", file=sys.stderr)
        return
    # Public and SCOTUS on same scale [-1,1]
    plt.figure()
    plt.plot(sim.years, sim.public, label="Public polarization (P_t)")
    plt.plot(sim.years, sim.scotus, label="SCOTUS polarization (S_t)")
    plt.axhline(0.0, linestyle="--")
    plt.xlabel("Year")
    plt.ylabel("Polarization (-1=D ... +1=R)")
    plt.title(title)
    plt.legend()
    plt.tight_layout()
    plt.show()

def sweep_term_limits(term_limits: List[Optional[int]],
                      base_params: SimParams,
                      presidents_by_year: Dict[int, str]) -> List[Tuple[Optional[int],
SimResult]]:
    out = []
    for tl in term_limits:
        p = SimParams(**{**base_params.__dict__, "term_limit_years": tl})
        sim = simulate(p, presidents_by_year)
        out.append((tl, sim))
    return out

def parse_term_limit_list(s: str) -> List[Optional[int]]:
    vals = []
    for tok in s.split(","):
        tok = tok.strip().lower()
        if tok in ("none", "lifetime", "inf"):
            vals.append(None)
        else:
            vals.append(int(tok))
    return vals

def main():
    ap = argparse.ArgumentParser(description="SCOTUS-Public polarization simulator")
    ap.add_argument("--start", type=int, default=1866)
    ap.add_argument("--end", type=int, default=2025)
    ap.add_argument("--term-limit", type=str, default="18",
                    help='Years or "none" for lifetime (e.g., 12,18,22,none)')
    ap.add_argument("--alpha", type=float, default=0.15, help="Public response rate λ")
    ap.add_argument("--gain", type=float, default=1.50, help="Public gain γ inside tanh(γ S)")
    ap.add_argument("--noise", type=float, default=0.03, help="Public noise σ")
    ap.add_argument("--retire-prob", type=float, default=0.00, help="Annual random retirement
prob per justice")
    ap.add_argument("--seed", type=int, default=42)
    ap.add_argument("--initial-public", type=float, default=0.0)
    ap.add_argument("--court-size", type=int, default=9)
    ap.add_argument("--presidents-csv", type=str, default="",
                    help='Optional CSV: start_year,end_year,party with party in {D,R}')

```

```

ap.add_argument("--alt-first", type=str, default="R",
                help="If no CSV provided, alternating timeline starts with this party (D or
R)")
ap.add_argument("--alt-term", type=int, default=4, help="Alt timeline term length (years)")
ap.add_argument("--out", type=str, default="", help="Write TSV to this path")
ap.add_argument("--plot", action="store_true", help="Show a matplotlib plot")
ap.add_argument("--sweep", type=str, default="",
                help="Comma list of term-limits to sweep, e.g. \"12,18,22,none\" (writes
multiple TSVs if --out given)")
args = ap.parse_args()

# Presidents mapping
if args.presidents_csv:
    mapping = load_presidents_from_csv(pathlib.Path(args.presidents_csv))
else:
    mapping = make_alternating_presidents(args.start, args.end, first_party=args.alt_first,
term_years=args.alt_term)

# Single run or sweep
if args.sweep:
    tls = parse_term_limit_list(args.sweep)
    base = SimParams(start_year=args.start, end_year=args.end,
                    public_alpha=args.alpha, public_gain=args.gain,
                    noise_sigma=args.noise, term_limit_years=None, # overridden per sweep
                    retire_prob=args.retire_prob,
                    initial_public=args.initial_public, court_size=args.court_size,
                    seed=args.seed)
    results = sweep_term_limits(tls, base, mapping)
    # Write/plot
    for tl, sim in results:
        label = "lifetime" if tl is None else f"{tl}y"
        if args.out:
            p = pathlib.Path(args.out)
            out_path = p if p.suffix else p.with_suffix("")
            out_file = pathlib.Path(f"{out_path}_{label}.tsv")
            write_tsv(out_file, sim)
            print(f"Wrote {out_file}")
        if args.plot:
            plot_series(sim, title=f"Term limit: {label}")
    else:
        tl = None if args.term_limit.lower() in ("none", "lifetime", "inf") else
int(args.term_limit)
        params = SimParams(start_year=args.start, end_year=args.end,
                        public_alpha=args.alpha, public_gain=args.gain,
                        noise_sigma=args.noise, term_limit_years=tl,
                        retire_prob=args.retire_prob,
                        initial_public=args.initial_public, court_size=args.court_size,
                        seed=args.seed)
        sim = simulate(params, mapping)
        if args.out:
            write_tsv(pathlib.Path(args.out), sim)
            print(f"Wrote {args.out}")
        if args.plot:
            plot_series(sim, title=f"Term limit: {'lifetime' if tl is None else f'{tl}y'}")

if __name__ == "__main__":
    main()

```

## How to use

- **Quick demo** (no historical data, alternating 4-year terms, start with R):

```
bash
```

```
python polarization_sim.py --start 1866 --end 2025 --term-limit 18 --plot
```

- **Use actual presidents** (recommended): Make a CSV `presidents.csv` with rows like:

```
text

start_year,end_year,party
1869,1872,R
1873,1876,R
...
```

Then:

```
bash

python polarization_sim.py --presidents-csv presidents.csv --term-limit 18 --plot
```

- **Sweep term limits** and write TSVs:

```
bash

python polarization_sim.py --presidents-csv presidents.csv \
  --sweep 12,18,22,none --out results/polarization --plot
# Produces:
# results/polarization_12y.tsv
# results/polarization_18y.tsv
# results/polarization_22y.tsv
# results/polarization_lifetime.tsv
```

## Model notes & easy extensions

- The **feedback** is intentionally simple: public tracks the Court via a smoothed, saturating response. Tweak **--alpha** (speed) and **--gain** (nonlinearity). Add cross-couplings (e.g., make retirements or vacancies depend on  $P_t$  or  $P_t \cdot S_t$ ) if you want stronger predator-prey dynamics.
- To incorporate **election outcomes driven by public polarization**, swap the exogenous presidents CSV for a Markov model where the probability of D vs R at election years depends on  $P_t$ . You can still run a “ground-truth mode” by feeding the historical CSV.
- Add **court size changes** (court-packing) by changing **court.size** at specific years.
- Plug in **shocks** (wars, crises) by adding a piecewise term to  $P_{t+1}$ .

If you’d like, I can also drop in a ready-to-use presidents CSV (or a scraper to build it once), or wire this into your sweep/plot pipeline so the TSVs slot right into your existing analysis tools.



So, the problem with making election outcomes driven by public polarization is that my intuition is that it's just always almost immediately fall into the hands of whomever is the first winner, and just peg to that side. Actually, showing that (which is sort of obvious) and then looking for ways to avoid it -- perhaps random removal of justices, or something.

Yep — if you let “the electorate” pick presidents deterministically from current public polarity  $P_t$  (winner-take-all) you’ll almost always get a symmetry break and an absorbing state: first winner → stacks seats → Court  $S_t$  reinforces  $P_t$  → same winner forever. Two good steps:

1. **Show the lock-in** with a bare-bones endogenous-election rule.
2. Add **de-pegging mechanisms** (frictions, randomness, backlash, attrition) and see which ones actually keep the system ergodic.

Below are **minimal patches** to the script I gave you to (A) enable endogenous elections and (B) try four de-pegging levers you can sweep.

## A) Endogenous elections (expect lock-in)

**Idea.** Every 4 years, sample the next president from a logistic tied to  $P_t$ . This is still “almost absorbing” when the temperature is low.

Add to your file near the other helpers:

```
python

def logistic(x: float) -> float:
    return 1.0 / (1.0 + math.exp(-x))
```

Extend **SimParams**:

```
python

@dataclass
class SimParams:
    ...
    election_years: int = 4          # election cycle
    endogenous_elections: bool = False
    elect_beta: float = 4.0          # sensitivity of elections to P_t (higher=more lock-in)
    elect_noise: float = 0.25        # shock added inside the logistic
    incumbency_bias: float = 0.0     # + favors incumbent party, - penalizes it
```

Inside **simulate(...)**, replace the presidents mapping logic with an **optional endogenous path**. Right after you set up the court (and before the year loop), add:

```
python

# Presidency process: either exogenous mapping (CSV/alternating) or endogenous.
if presidents_by_year: # existing behavior
    get_party = lambda year, cur_party: presidents_by_year.get(year, cur_party)
    cur_pres = presidents_by_year.get(params.start_year, "R")
else:
    # Endogenous elections from P_t (winner sampled every election_years)
    params.endogenous_elections = True
    cur_pres = "R" # seed; will change at first election
    years_since_election = 0

    def get_party(year, cur_party):
        nonlocal years_since_election, P, cur_pres
        if years_since_election >= params.election_years:
            years_since_election = 0
            # Probability R wins given current P (and small shock)
            shock = random.gauss(0.0, params.elect_noise)
            # Optionally add incumbency term (+ favors incumbent)
            inc = params.incumbency_bias if cur_pres == "R" else -params.incumbency_bias
            pr_R = logistic(params.elect_beta * (P + shock) + inc)
            cur_pres = "R" if random.random() < pr_R else "D"
            years_since_election += 1
        return cur_pres
```

Then, inside the main loop, replace:

```
python

    president_party = presidents_by_year.get(year, pres0)
```

with:

```
python

    president_party = get_party(year, cur_pres)
    cur_pres = president_party
```

Run with:

```
bash

python polarization_sim.py --start 1866 --end 2025 --term-limit 18 --plot
# (with no --presidents-csv, it will use endogenous elections)
```

If you keep `elect_beta` high and `elect_noise` low, you'll see the peg.

## B) Four de-pegging mechanisms (toggle/sweep these)

### 1) Election temperature & backlash (soften winner-take-all)

- Keep `elect_noise` > 0 (temperature).
- Add **backlash**: when Court tilt  $S_t$  and public  $P_t$  align strongly, some voters defect (legitimacy / overreach). Replace the `pr_R` line with:

```
python

    # Backlash term: reduce winning odds when S and P are aligned & extreme
    backlash_k = 1.0 # sweep 0..2
    pr_R = logistic(params.elect_beta * (P + shock - backlash_k * (S * P)))
```

Set `backlash_k=0` to disable. Positive values create alternation pressure.

### 2) Public mean-reversion vs. Court influence (reduce positive feedback)

You already have this, but sweep it explicitly:

- Lower `--gain` ( $\gamma$ ) or `--alpha` ( $\lambda$ ), or add **exogenous centering shocks**:

```
python

    # occasional centering (e.g., crises reset salience)
    if random.random() < 0.02: # 2%/year
        P *= 0.7                # pull toward center
```

### 3) Random justice attrition (unpredictable vacancies)



You already have `--retire-prob`; treat it as **attrition** (health, scandals, strategic exits). Use non-tiny values (e.g., 0.03–0.07) and optionally **polarity-dependent hazard** (overreach breeds exits):

In `Court.step_year` before stochastic retirement, replace:

```
python

    if retire_prob > 0.0 and random.random() < retire_prob:
```

with:

```
python

    hazard = retire_prob
    # Optional: stress exits when Court is extreme (|S| large)
    # You need S here; pass it in as an arg if you want this dependency.
    if random.random() < hazard:
```

(If you want S-dependent hazard, add `current_S` param to `step_year` and set `hazard = retire_prob * (1.0 + 0.75 * abs(current_S))`.)

## 4) Staggered guaranteed appointments (institutional friction)

Force exactly one seat to expire each year (or every k years) regardless of terms; this prevents large single-party waves.

Add to `SimParams`:

```
python

forced_vacancy_every: Optional[int] = None # e.g., 1 = one seat/year, None = off
```

In `Court.step_year(...)` compute forced vacancies:

```
python

def step_year(..., forced_vacancy_every: Optional[int] = None):
    ...
    # After expirations:
    if forced_vacancy_every is not None and forced_vacancy_every > 0:
        if (year % forced_vacancy_every) == 0 and len(self.justices) > 0:
            # Vacate the *most senior* justice to simulate clockwork staggering
            oldest_idx = min(range(len(self.justices)), key=lambda i:
self.justices[i].appointed_year)
            self.justices.pop(oldest_idx)
            vacancies_this_year += 1
```

Then pass it from `simulate(...)`:

```
python

    court.step_year(year, president_party,
                    vacancies_this_year=0,
                    retire_prob=params.retire_prob,
                    forced_vacancy_every=params.forced_vacancy_every)
```

## Suggested experiments (fast to run)

### 1. Baseline lock-in

`--elect_beta 6 --elect_noise 0.05 --retire-prob 0.00 --gain 1.5 --alpha 0.2 --term-limit none`  
 Expect quick peg after the first election.

### 2. Temperature only

Sweep `elect_noise`  $\in \{0.1, 0.2, 0.3, 0.4\}$  — record fraction of years where  $\text{sign}(P)$  flips. Watch for phase transition.

### 3. Backlash

Fix noise modest (0.2). Sweep `backlash_k`  $\in \{0, 0.5, 1.0, 1.5, 2.0\}$  — plot alternation rate vs. `backlash_k`.

### 4. Attrition

`--retire-prob`  $\in \{0.00, 0.02, 0.05, 0.08\}$  — measure time-to-peg and mean  $|S|$ . Attrition destabilizes pegs.

### 5. Staggered seats

`--forced-vacancy-every 1` plus low noise. This alone often prevents lock-in by limiting step changes in  $S_t$ .

### 6. Combo (realistic-ish)

Moderate noise (0.2), mild backlash (0.8), attrition (0.03), staggered seats (1), 18-year terms. This should give long-run mixing without being purely random.

## Why these help (quick intuition)

- **Temperature** makes elections non-deterministic near the boundary, so early luck doesn't freeze the system.
- **Backlash** is negative feedback: extreme alignment  $S \approx P$  reduces same-party win odds.
- **Attrition** decouples seat flow from single presidents; trickle vacancies blur waves.
- **Staggered seats** cap how fast  $S_t$  can move, weakening  $S \rightarrow P \rightarrow S$  runaway.

If you want, I can fold these patches into a single clean diff (or just output the whole updated file) and add TSV metrics (e.g., alternation rate, average  $|P|$ , average  $|S|$ , time-to-peg) so it slots into your sweep/plot pipeline.



Can all of you "Suggested experiments (fast to run)" be run in the current code?

Short answer: **not yet**. In your current script, only part of **#4 Attrition** (sweeping `--retire-prob`) and general feedback tweaks (`--alpha`, `--gain`, `--noise`) run as-is. The others need tiny patches.

Here's the minimal, surgical patch set to enable all six experiments. I've grouped them so you can paste them in quickly.

## 0) Tiny helper (top of file, near other helpers)

```
python

def logistic(x: float) -> float:
    return 1.0 / (1.0 + math.exp(-x))
```

## 1) Params additions (extend SimParams)

```
python

@dataclass
class SimParams:
    # ... keep existing fields ...
    election_years: int = 4
    endogenous_elections: bool = False
    elect_beta: float = 4.0 # sensitivity of elections to P_t
    elect_noise: float = 0.25 # "temperature" of elections
    incumbency_bias: float = 0.0 # + favors incumbent R, - favors D

    backlash_k: float = 0.0 # 0 = off; >0 adds backlash
    forced_vacancy_every: Optional[int] = None # e.g., 1 = one seat/year
```

Also expose CLI flags (add to argparse):

```
python

ap.add_argument("--endogenous-elections", action="store_true")
ap.add_argument("--elect-beta", type=float, default=4.0)
ap.add_argument("--elect-noise", type=float, default=0.25)
ap.add_argument("--incumbency-bias", type=float, default=0.0)
ap.add_argument("--backlash-k", type=float, default=0.0)
ap.add_argument("--forced-vacancy-every", type=int, default=None)
```

When building `SimParams` in `main()`, pass these through.

## 2) Court API change (staggered vacancy hook)

Change the signature of `Court.step_year` to accept `forced_vacancy_every` and the **current year's court polarization** (for optional S-dependent hazard later if you want it):

```
python

def step_year(self, year: int, president_party: str,
              vacancies_this_year: int = 0,
              retire_prob: float = 0.0,
              forced_vacancy_every: Optional[int] = None,
              current_S: Optional[float] = None) -> None:
```

At the **start** of the method, keep expirations as-is. After computing `really_remaining`, insert **staggered vacancy**:

python

```
# After natural expirations + stochastic retirements:
self.justices = really_remaining

# Staggered guaranteed vacancy (optional)
if forced_vacancy_every is not None and forced_vacancy_every > 0:
    if (year % forced_vacancy_every) == 0 and len(self.justices) > 0:
        oldest_idx = min(range(len(self.justices)), key=lambda i:
self.justices[i].appointed_year)
        self.justices.pop(oldest_idx)
        vacancies_this_year += 1
```

Leave your existing random retirement line intact:

python

```
if retire_prob > 0.0 and random.random() < retire_prob:
    vacancies_this_year += 1
```

(If you later want S-dependent hazard, change that line to use **current\_S**.)

Filling vacancies block stays the same.

### 3) Endogenous elections + backlash (inside simulate)

Right after you build the initial court, replace the president mapping with this unified getter:

python

```
# Presidency process: exogenous mapping (CSV/alternating) OR endogenous elections
if presidents_by_year and not params.endogenous_elections:
    cur_pres = presidents_by_year.get(params.start_year, "R")
    def get_party(year, cur_party, P, S):
        return presidents_by_year.get(year, cur_party)
else:
    params.endogenous_elections = True
    cur_pres = "R" # seed; will be updated at first election boundary
    years_since_election = 0
    def get_party(year, cur_party, P, S):
        nonlocal years_since_election, cur_pres
        if years_since_election >= params.election_years:
            years_since_election = 0
            shock = random.gauss(0.0, params.elect_noise)
            inc = params.incumbency_bias if cur_pres == "R" else -params.incumbency_bias
            # Backlash reduces odds when P and S are aligned and large in magnitude
            backlash = params.backlash_k * (S * P)
            pr_R = logistic(params.elect_beta * (P + shock - backlash) + inc)
            cur_pres = "R" if random.random() < pr_R else "D"
            years_since_election += 1
        return cur_pres
```

Then in the **year loop**:

1. Move presidency selection to the **top** of the loop so you can pass **current\_S** into **step\_year**:

python

```
for year in range(params.start_year, params.end_year + 1):
    # use last computed S (or 0 at start) for hazard/backlash plumbing
    S_prev = scoti[-1] if scoti else court.polarization()
    president_party = get_party(year, cur_pres, P, S_prev)
```

```
cur_pres = president_party

# Court evolves (pass forced vacancy value and optional current S)
court.step_year(year, president_party,
                vacancies_this_year=0,
                retire_prob=params.retire_prob,
                forced_vacancy_every=params.forced_vacancy_every,
                current_S=S_prev)
S = court.polarization()
```

2. Then update public:

```
python

influence = math.tanh(params.public_gain * S)
noise = random.gauss(0.0, params.noise_sigma)
P = (1.0 - params.public_alpha) * P + params.public_alpha * influence + noise
P = clamp(P)
```

Push to arrays as before.

## 4) Bootstrap calls must pass new args

Where you initially seed the court at `start_year`, update:

```
python

court.step_year(params.start_year, pres0, vacancies_this_year=params.court_size,
                retire_prob=params.retire_prob,
                forced_vacancy_every=params.forced_vacancy_every,
                current_S=0.0)
```

And in the non-sweep and sweep branches, make sure `SimParams(...)` includes the new fields from `argparse`.

## What this unlocks (mapping to your experiments)

1. **Baseline lock-in** → run with `--endogenous-elections --elect-beta 6 --elect-noise 0.05 --retire-prob 0 --term-limit none`.
2. **Temperature only** → sweep `--elect-noise` with endogenous elections on.
3. **Backlash** → sweep `--backlash-k` (0 disables; try 0-2).
4. **Attrition** → already works (you had `--retire-prob`); now you can also make it S-dependent later via `current_S`.
5. **Staggered seats** → set `--forced-vacancy-every 1` (or another k).
6. **Combo** → mix the above flags in one run or sweep.

If you want, I can output the fully merged file with these exact edits applied so you can drop it in and go.



Sorry, I'm a little lost. Can you give me the complete best current code for the minimal set of experiments with just #4 Attrition (sweeping --retire-prob) and general feedback tweaks (--alpha, --gain, --noise)

Got it—here's a single, clean Python script that supports exactly what you want right now:

- exogenous presidents (CSV or simple alternating fallback)
- feedback knobs: --alpha, --gain, --noise
- attrition only: sweep --retire-prob (or run a single value)
- optional term limits (--term-limit or none)
- write TSVs and/or plot

Save as **polarization\_min\_attrition.py**.

```
python

#!/usr/bin/env python3
# polarization_min_attrition.py
# Minimal simulator for coupled Public ↔ SCOTUS polarization
# Focused on: feedback knobs (--alpha, --gain, --noise) and attrition (--retire-prob)

import argparse, csv, math, random, statistics, sys, pathlib
from dataclasses import dataclass, field
from typing import List, Optional, Dict, Tuple

try:
    import matplotlib.pyplot as plt
except Exception:
    plt = None # plotting is optional

# -----
# Conventions
# -----
# Parties: D = -1.0, R = +1.0
# Public polarization P_t in [-1, 1], SCOTUS polarization S_t in [-1, 1]
# Time step = 1 year

PARTY_TO_LEAN = {"D": -1.0, "R": +1.0}

@dataclass
class Justice:
    appointed_year: int
    term_limit_years: Optional[int] # None = lifetime
    lean: float # -1 (D) to +1 (R)

    def expires_in_year(self, year: int) -> bool:
        if self.term_limit_years is None:
            return False
        return (year - self.appointed_year) >= self.term_limit_years

@dataclass
class Court:
    size: int = 9
    term_limit_years: Optional[int] = None
    justices: List[Justice] = field(default_factory=list)

    def polarization(self) -> float:
        if not self.justices:
            return 0.0
        return statistics.fmean(j.lean for j in self.justices)

    def step_year(
        self,
```

```

        year: int,
        president_party: str,
        vacancies_this_year: int = 0,
        retire_prob: float = 0.0,
    ) -> None:
        """Advance one year: natural expirations + stochastic retirements, then fill
        vacancies."""
        # Natural expirations from term limits
        remaining = []
        for j in self.justices:
            if j.expires_in_year(year):
                vacancies_this_year += 1
            else:
                remaining.append(j)

        # Stochastic attrition (health, scandal, strategic retirement, etc.)
        really_remaining = []
        for j in remaining:
            if retire_prob > 0.0 and random.random() < retire_prob:
                vacancies_this_year += 1
            else:
                really_remaining.append(j)

        self.justices = really_remaining

        # Fill to full size with current president's party
        while len(self.justices) < self.size:
            self.justices.append(
                Justice(
                    appointed_year=year,
                    term_limit_years=self.term_limit_years,
                    lean=PARTY_TO_LEAN[president_party],
                )
            )

def load_presidents_from_csv(csv_path: pathlib.Path) -> Dict[int, str]:
    """
    CSV columns: start_year,end_year,party (party ∈ {D,R})
    Returns dict year->party for years [start_year, end_year] inclusive (per row).
    """
    mapping: Dict[int, str] = {}
    with csv_path.open(newline='') as f:
        reader = csv.DictReader(f)
        for row in reader:
            sy, ey = int(row["start_year"]), int(row["end_year"])
            party = row["party"].strip().upper()
            assert party in ("D", "R"), f"Invalid party {party}"
            for y in range(sy, ey + 1):
                mapping[y] = party
    return mapping

def make_alternating_presidents(
    start_year: int,
    end_year: int,
    first_party: str = "R",
    term_years: int = 4,
) -> Dict[int, str]:
    """Fallback: alternate party every 'term_years' starting at start_year."""
    mapping: Dict[int, str] = {}
    p0 = first_party.upper()
    assert p0 in ("D", "R")
    for y in range(start_year, end_year + 1):
        block = (y - start_year) // term_years
        party = "R" if (p0 == "R" and block % 2 == 0) or (p0 == "D" and block % 2 == 1) else "D"
        mapping[y] = party
    return mapping

@dataclass
class SimParams:
    start_year: int = 1866
    end_year: int = 2025
    public_alpha: float = 0.15 # λ: public response rate to court
    public_gain: float = 1.50 # γ: strength of court → public inside tanh
    noise_sigma: float = 0.03 # σ: public noise
    term_limit_years: Optional[int] = 18

```

```

retire_prob: float = 0.00 # annual random retirement prob per justice-year
initial_public: float = 0.0
court_size: int = 9
seed: int = 42

@dataclass
class SimResult:
    years: List[int]
    public: List[float]
    scotus: List[float]
    presidents: List[str]

def clamp(x: float, lo=-1.0, hi=1.0) -> float:
    return max(lo, min(hi, x))

def simulate(params: SimParams, presidents_by_year: Dict[int, str]) -> SimResult:
    random.seed(params.seed)

    court = Court(size=params.court_size, term_limit_years=params.term_limit_years)

    # Bootstrap: fill the Court at start with that year's president
    pres0 = presidents_by_year.get(params.start_year, "R")
    court.step_year(
        params.start_year,
        pres0,
        vacancies_this_year=params.court_size,
        retire_prob=params.retire_prob,
    )

    P = params.initial_public
    years: List[int] = []
    publics: List[float] = []
    scoti: List[float] = []
    pres: List[str] = []

    for year in range(params.start_year, params.end_year + 1):
        president_party = presidents_by_year.get(year, pres0)

        # Court evolves (expirations + attrition + fills)
        court.step_year(
            year,
            president_party,
            vacancies_this_year=0,
            retire_prob=params.retire_prob,
        )
        S = court.polarization()

        # Public responds:  $P_{t+1} = (1-\lambda)P_t + \lambda \tanh(\gamma S_t) + \text{noise}$ 
        influence = math.tanh(params.public_gain * S)
        noise = random.gauss(0.0, params.noise_sigma)
        P = (1.0 - params.public_alpha) * P + params.public_alpha * influence + noise
        P = clamp(P)

        years.append(year)
        publics.append(P)
        scoti.append(S)
        pres.append(president_party)

    return SimResult(years=years, public=publics, scotus=scoti, presidents=pres)

def write_tsv(path: pathlib.Path, sim: SimResult) -> None:
    path.parent.mkdir(parents=True, exist_ok=True)
    with path.open("w") as f:
        f.write("year\tpresident_party\tpublic_polarization\tscotus_polarization\n")
        for y, pr, P, S in zip(sim.years, sim.presidents, sim.public, sim.scotus):
            f.write(f"{y}\t{pr}\t{P:.6f}\t{S:.6f}\n")

def plot_series(sim: SimResult, title: str = "Polarization over time"):
    if plt is None:
        print("matplotlib not available; skipping plot.", file=sys.stderr)
        return
    plt.figure()
    plt.plot(sim.years, sim.public, label="Public polarization (P_t)")
    plt.plot(sim.years, sim.scotus, label="SCOTUS polarization (S_t)")
    plt.axhline(0.0, linestyle="--")

```



```

plt.xlabel("Year")
plt.ylabel("Polarization (-1=D ... +1=R)")
plt.title(title)
plt.legend()
plt.tight_layout()
plt.show()

def parse_float_list(s: str) -> List[float]:
    vals = []
    for tok in s.split(","):
        tok = tok.strip()
        if tok:
            vals.append(float(tok))
    return vals

def sweep_retire_prob(
    retire_probs: List[float],
    base_params: SimParams,
    presidents_by_year: Dict[int, str],
) -> List[Tuple[float, SimResult]]:
    results = []
    for rp in retire_probs:
        p = SimParams(**{**base_params.__dict__, "retire_prob": rp})
        sim = simulate(p, presidents_by_year)
        results.append((rp, sim))
    return results

def main():
    ap = argparse.ArgumentParser(description="Minimal SCOTUS-Public polarization simulator
(attrition & feedback)")
    ap.add_argument("--start", type=int, default=1866)
    ap.add_argument("--end", type=int, default=2025)
    ap.add_argument("--term-limit", type=str, default="18", help='Years or "none" for lifetime')
    ap.add_argument("--alpha", type=float, default=0.15, help="Public response rate λ")
    ap.add_argument("--gain", type=float, default=1.50, help="Public gain γ inside tanh(γ S)")
    ap.add_argument("--noise", type=float, default=0.03, help="Public noise σ")
    ap.add_argument("--retire-prob", type=float, default=0.00, help="Annual retirement
probability per justice-year")
    ap.add_argument("--seed", type=int, default=42)
    ap.add_argument("--initial-public", type=float, default=0.0)
    ap.add_argument("--court-size", type=int, default=9)

    ap.add_argument("--presidents-csv", type=str, default="",
                    help='Optional CSV: start_year,end_year,party with party in {D,R}')
    ap.add_argument("--alt-first", type=str, default="R",
                    help="If no CSV, alternating timeline starts with this party (D or R)")
    ap.add_argument("--alt-term", type=int, default=4, help="Alt timeline term length (years)")

    ap.add_argument("--out", type=str, default="", help="Write single-run TSV to this path")
    ap.add_argument("--plot", action="store_true", help="Show a matplotlib plot")

    ap.add_argument("--sweep-retire", type=str, default="",
                    help='Comma list of retirement probs to sweep, e.g. "0.00,0.02,0.05,0.08". '
                    'If set, writes multiple TSVs when --out is provided; plots each if --
plot.')

    args = ap.parse_args()

    # Presidents mapping (exogenous only)
    if args.presidents_csv:
        mapping = load_presidents_from_csv(pathlib.Path(args.presidents_csv))
    else:
        mapping = make_alternating_presidents(args.start, args.end,
                                              first_party=args.alt_first,
                                              term_years=args.alt_term)

    tl = None if args.term_limit.lower() in ("none", "lifetime", "inf") else
int(args.term_limit)
    base = SimParams(
        start_year=args.start, end_year=args.end,
        public_alpha=args.alpha, public_gain=args.gain,
        noise_sigma=args.noise, term_limit_years=tl,
        retire_prob=args.retire_prob, initial_public=args.initial_public,
        court_size=args.court_size, seed=args.seed
    )

```

```

if args.sweep_retire:
    rps = parse_float_list(args.sweep_retire)
    results = sweep_retire_prob(rps, base, mapping)
    for rp, sim in results:
        label = f"retire{rp:.3f}".replace(".", "p")
        if args.out:
            p = pathlib.Path(args.out)
            out_path = p if p.suffix else p.with_suffix("")
            out_file = pathlib.Path(f"{out_path}_{label}.tsv")
            write_tsv(out_file, sim)
            print(f"Wrote {out_file}")
        if args.plot:
            plot_series(sim, title=f"Attrition p={rp:.3f}")
else:
    sim = simulate(base, mapping)
    if args.out:
        write_tsv(pathlib.Path(args.out), sim)
        print(f"Wrote {args.out}")
    if args.plot:
        ttl = "lifetime" if tl is None else f"{tl}y"
        plot_series(sim, title=f"Term limit: {ttl} | retire p={base.retire_prob:.3f}")

if __name__ == "__main__":
    main()

```

## Quick runs

- **Single run, alternating presidents, 18-year terms, modest noise:**

```

bash

python polarization_min_attrition.py --start 1866 --end 2025 \
  --term-limit 18 --alpha 0.15 --gain 1.5 --noise 0.03 \
  --retire-prob 0.02 --plot

```

- **Use a presidents CSV (recommended for realism):**

```

bash

python polarization_min_attrition.py --presidents-csv presidents.csv \
  --term-limit 18 --alpha 0.15 --gain 1.5 --noise 0.03 \
  --retire-prob 0.02 --out runs/attrition_single.tsv --plot

```

- **Sweep attrition** (writes multiple TSVs + plots each):

```

bash

python polarization_min_attrition.py --presidents-csv presidents.csv \
  --term-limit 18 --alpha 0.15 --gain 1.5 --noise 0.03 \
  --sweep-retire "0.00,0.02,0.05,0.08" --out runs/attrition

```

That's it—lean and ready to drop into your pipeline. If you want, I can add a tiny metrics TSV (e.g., mean  $|P|$ , mean  $|S|$ , flips/year) without changing the core dynamics.



somehow I lost the cvs of the presidents can you recreate that

Here's a CSV for U.S. Presidents' parties from **1866–present**, with one line per “block” of years where the same president holds office. You can use this to rebuild your `presidents.csv`.

CSV

```
start_year,end_year,party
1866,1868,R
1869,1876,R
1877,1880,R
1881,1881,R
1881,1884,R
1885,1888,D
1889,1892,R
1893,1896,D
1897,1900,R
1901,1908,R
1909,1912,R
1913,1920,D
1921,1922,R
1923,1928,R
1929,1932,R
1933,1944,D
1945,1952,D
1953,1960,R
1961,1962,D
1963,1968,D
1969,1973,R
1974,1976,R
1977,1980,D
1981,1988,R
1989,1992,R
1993,2000,D
2001,2008,R
2009,2016,D
2017,2020,R
2021,2024,D
```

## Notes / checks

- For simplicity, I treated 1866–1868 as “R” because from 1869 Grant (R) took over; you might prefer to start exactly at 1869.
- Years labelled are inclusive; e.g. **1869,1876,R** means that from 1869 through 1876 the President was Republican.
- “D” = Democrat, “R” = Republican.
- The last line ends at 2024 (current through “present”); adjust if you want to project beyond.

If you want, I can generate the full CSV with precise month-of-inauguration splits (so partial years aren’t lumped), or extend backwards before 1866.



Can you give me a separate plotting program that takes a `--rundir` param and aggregates all the tsvs into a single plot with just the public polarization lines (each labeled with the appropriate pretire? (It might be easier to do this in the main program, and then we would create a new main program for each experiment?)

Awesome—here's a small standalone script that does exactly that.

Save as `plot_rundir_public.py`. It:

- accepts `--rundir` (and optional `--recursive`)
- loads every `.tsv` inside, reading only the **public\_polarization** column
- plots all runs on one figure with labels parsed from filenames like `..._retire0p020.tsv` → “retire p=0.020”
- optionally `--save path.png` instead of showing the window
- lets you tweak title and y-limits

python

```
#!/usr/bin/env python3
# plot_rundir_public.py
# Aggregate a directory of TSVs and plot ONLY the public_polarization lines.

import argparse
import pathlib
import re
import sys
from typing import List, Tuple

import csv

try:
    import matplotlib.pyplot as plt
except Exception as e:
    print("matplotlib is required to plot. Install it and re-run.", file=sys.stderr)
    raise

LABEL_RE = re.compile(r"retire(\d+(?:p\d+)?)", re.IGNORECASE)

def parse_label_from_filename(p: pathlib.Path) -> str:
    """
    Pull a retire-prob label from filenames like:
    ..._retire0p020.tsv -> 'retire p=0.020'
    Fallback: use file stem.
    """
    m = LABEL_RE.search(p.stem)
    if m:
        raw = m.group(1)
        # Convert "0p020" -> "0.020"
        nice = raw.replace("p", ".")
        try:
            val = float(nice)
            return f"retire p={val:.3f}"
        except ValueError:
            return f"retire {nice}"
    return p.stem

def read_public_series(tsv_path: pathlib.Path) -> Tuple[List[int], List[float]]:
    years: List[int] = []
    publics: List[float] = []
    with tsv_path.open(newline="") as f:
        reader = csv.DictReader(f, delimiter="\t")
        # Expect columns: year, president_party, public_polarization, scotus_polarization
        if "year" not in reader.fieldnames or "public_polarization" not in reader.fieldnames:
            raise ValueError(f"{tsv_path} missing required columns")
        for row in reader:
            try:
                years.append(int(row["year"]))
                publics.append(float(row["public_polarization"]))
            except Exception:
                # Skip malformed rows
                continue
    return years, publics
```

```
def collect_tsvs(rundir: pathlib.Path, recursive: bool) -> List[pathlib.Path]:
    if recursive:
        return sorted(p for p in rundir.rglob("*.tsv"))
    return sorted(p for p in rundir.glob("*.tsv"))

def main():
    ap = argparse.ArgumentParser(description="Plot public_polarization from all TSVs in a run directory.")
    ap.add_argument("--rundir", required=True, help="Directory containing TSV outputs")
    ap.add_argument("--recursive", action="store_true", help="Recurse into subdirectories")
    ap.add_argument("--title", default="Public Polarization (all runs)", help="Plot title")
    ap.add_argument("--ymin", type=float, default=-1.0, help="Y-axis min (default -1)")
    ap.add_argument("--ymax", type=float, default=1.0, help="Y-axis max (default +1)")
    ap.add_argument("--save", default="", help="If set, save figure to this path instead of showing")
    args = ap.parse_args()

    rundir = pathlib.Path(args.rundir)
    if not rundir.exists() or not rundir.is_dir():
        print(f"--rundir path not found or not a directory: {rundir}", file=sys.stderr)
        sys.exit(1)

    tsvs = collect_tsvs(rundir, args.recursive)
    if not tsvs:
        print(f"No .tsv files found in {rundir} (recursive={args.recursive})", file=sys.stderr)
        sys.exit(1)

    plt.figure()
    any_plotted = False

    for tsv in tsvs:
        try:
            years, publics = read_public_series(tsv)
            if not years or not publics:
                continue
            label = parse_label_from_filename(tsv)
            plt.plot(years, publics, label=label)
            any_plotted = True
        except Exception as e:
            print(f"Skipping {tsv}: {e}", file=sys.stderr)

    if not any_plotted:
        print("Found TSVs but none had valid public_polarization data.", file=sys.stderr)
        sys.exit(1)

    plt.axhline(0.0, linestyle="--")
    plt.xlabel("Year")
    plt.ylabel("Public polarization (-1 ... +1)")
    plt.title(args.title)
    plt.ylim(args.ymin, args.ymax)
    plt.legend(loc="best")
    plt.tight_layout()

    if args.save:
        out_path = pathlib.Path(args.save)
        out_path.parent.mkdir(parents=True, exist_ok=True)
        plt.savefig(out_path, dpi=200)
        print(f"Saved plot to {out_path}")
    else:
        plt.show()

if __name__ == "__main__":
    main()
```

## Examples

- Plot all TSVs in a run dir:

```
bash

python plot_rundir_public.py --rundir runs/attrition
```

- Recurse & save to a PNG:

```
bash

python plot_rundir_public.py --rundir runs/attrition --recursive --save
runs/attrition/summary_public.png
```

- Custom title / bounds:

```
bash

python plot_rundir_public.py --rundir runs/attrition \
  --title "Attrition sweep: public only" --ymin -1 --ymax 1
```

This keeps your main simulator lean. If you later want it to label from inside the TSV (e.g., include **retire\_prob** in the header row), I can add a tiny optional parser branch.



Okay. Create a main that sweep retire now.

Perfect. Here's a tiny **experiment main** that *only* does an attrition sweep, using the simulator you saved as **polarization\_min\_attrition.py**. Save this as **main\_attrition\_sweep.py** in the same folder.

```
python

#!/usr/bin/env python3
# main_attrition_sweep.py
# Runs a sweep over --retire-prob using the minimal simulator.

import argparse
import pathlib
from polarization_min_attrition import (
    SimParams,
    load_presidents_from_csv,
    make_alternating_presidents,
    sweep_retire_prob,
    write_tsv,
    plot_series,
)

def parse_float_list(s: str):
    return [float(t.strip()) for t in s.split(",") if t.strip()]

def main():
    ap = argparse.ArgumentParser(description="Attrition sweep driver (retire_prob)")
    ap.add_argument("--start", type=int, default=1866)
    ap.add_argument("--end", type=int, default=2025)

    ap.add_argument("--term-limit", type=str, default="18", help='Years or "none"')
    ap.add_argument("--alpha", type=float, default=0.15, help="Public response rate λ")
    ap.add_argument("--gain", type=float, default=1.50, help="Public gain γ inside tanh(γ S)")
    ap.add_argument("--noise", type=float, default=0.03, help="Public noise σ")
    ap.add_argument("--seed", type=int, default=42)
    ap.add_argument("--initial-public", type=float, default=0.0)
```

```

ap.add_argument("--court-size", type=int, default=9)

ap.add_argument("--presidents-csv", type=str, default="",
                help='Optional CSV: start_year,end_year,party with party in {D,R}')
ap.add_argument("--alt-first", type=str, default="R",
                help="If no CSV, alternating timeline starts with this party (D or R)")
ap.add_argument("--alt-term", type=int, default=4, help="Alt timeline term length (years)")

ap.add_argument("--retire-list", type=str, required=True,
                help='Comma list, e.g. "0.00,0.02,0.05,0.08"')

ap.add_argument("--outdir", type=str, default="runs/attrition",
                help="Directory to write TSVs")
ap.add_argument("--plot", action="store_true", help="Show a plot for each run")

args = ap.parse_args()

# Presidents mapping (exogenous only)
if args.presidents_csv:
    mapping = load_presidents_from_csv(pathlib.Path(args.presidents_csv))
else:
    mapping = make_alternating_presidents(args.start, args.end,
                                          first_party=args.alt_first,
                                          term_years=args.alt_term)

tl = None if args.term_limit.lower() in ("none", "lifetime", "inf") else
int(args.term_limit)
base = SimParams(
    start_year=args.start, end_year=args.end,
    public_alpha=args.alpha, public_gain=args.gain,
    noise_sigma=args.noise, term_limit_years=tl,
    retire_prob=0.0, # overridden per sweep
    initial_public=args.initial_public,
    court_size=args.court_size,
    seed=args.seed,
)

rps = parse_float_list(args.retire_list)
outdir = pathlib.Path(args.outdir)
outdir.mkdir(parents=True, exist_ok=True)

results = sweep_retire_prob(rps, base, mapping)
for rp, sim in results:
    label = f"retire{rp:.3f}".replace(".", "p")
    out_file = outdir / f"polarization_{label}.tsv"
    write_tsv(out_file, sim)
    print(f"Wrote {out_file}")
    if args.plot:
        plot_series(sim, title=f"Attrition sweep - p={rp:.3f}")

if __name__ == "__main__":
    main()

```

## Example usage

- Alternating presidents, 18-year terms, sweep four attrition rates, write TSVs under `runs/attrition`, and pop plots:

```

bash

python main_attrition_sweep.py \
  --retire-list "0.00,0.02,0.05,0.08" \
  --term-limit 18 --alpha 0.15 --gain 1.5 --noise 0.03 \
  --outdir runs/attrition --plot

```

- Using your presidents CSV:

```
bash
```

```
python main_attrition_sweep.py \
  --presidents-csv presidents.csv \
  --retire-list "0.00,0.02,0.05,0.08" \
  --term-limit 18 --alpha 0.15 --gain 1.5 --noise 0.03 \
  --outdir runs/attrition
```

This keeps the simulator lean and lets you spin up a fresh `main_*` for each experiment style.



I don't understand (or maybe you didn't understand me). The one you gave me previously had `--sweep-retire`, now I want one that does `--sweep-term-limit` instead.

Ah, got it — thanks for clarifying. You want the same minimal simulator, but instead of sweeping attrition (`--sweep-retire`), you want to sweep **term limits** (`--sweep-term-limit`).

Here's the complete main program updated for that. Save it as `polarization_min_term_sweep.py`. It's basically the same structure as before, but the sweep parameter is `term_limit_years` instead of `retire_prob`.

```
python
```

```
#!/usr/bin/env python3
# polarization_min_term_sweep.py
# Minimal simulator with sweep over term limits instead of retire probability.

import argparse, pathlib
from typing import List, Optional, Tuple
from polarization_min_attrition import (
    SimParams,
    load_presidents_from_csv,
    make_alternating_presidents,
    simulate,
    write_tsv,
    plot_series,
)

def parse_term_limit_list(s: str) -> List[Optional[int]]:
    """
    Parse a comma-separated list like "12,18,22,none" into [12,18,22,None].
    """
    vals: List[Optional[int]] = []
    for tok in s.split(","):
        tok = tok.strip().lower()
        if tok in ("none", "lifetime", "inf"):
            vals.append(None)
        elif tok:
            vals.append(int(tok))
    return vals

def sweep_term_limits(
    term_limits: List[Optional[int]],
    base_params: SimParams,
    presidents_by_year,
) -> List[Tuple[Optional[int], any]]:
    results = []
    for tl in term_limits:
        p = SimParams(**base_params.__dict__, "term_limit_years": tl)
        sim = simulate(p, presidents_by_year)
        results.append((tl, sim))
    return results

def main():
```



```

ap = argparse.ArgumentParser(description="Sweep over term limits for SCOTUS–Public
polarization model")
ap.add_argument("--start", type=int, default=1866)
ap.add_argument("--end", type=int, default=2025)

ap.add_argument("--alpha", type=float, default=0.15)
ap.add_argument("--gain", type=float, default=1.50)
ap.add_argument("--noise", type=float, default=0.03)
ap.add_argument("--retire-prob", type=float, default=0.00)
ap.add_argument("--seed", type=int, default=42)
ap.add_argument("--initial-public", type=float, default=0.0)
ap.add_argument("--court-size", type=int, default=9)

ap.add_argument("--presidents-csv", type=str, default="",
                help="CSV with start_year,end_year,party (party ∈ {D,R})")
ap.add_argument("--alt-first", type=str, default="R")
ap.add_argument("--alt-term", type=int, default=4)

ap.add_argument("--sweep-term-limit", type=str, required=True,
                help="Comma list of term limits, e.g. \"12,18,22,none\"")
ap.add_argument("--outdir", type=str, default="runs/term_sweep")
ap.add_argument("--plot", action="store_true")

args = ap.parse_args()

# Presidents mapping
if args.presidents_csv:
    mapping = load_presidents_from_csv(pathlib.Path(args.presidents_csv))
else:
    mapping = make_alternating_presidents(
        args.start, args.end,
        first_party=args.alt_first,
        term_years=args.alt_term,
    )

base = SimParams(
    start_year=args.start, end_year=args.end,
    public_alpha=args.alpha, public_gain=args.gain,
    noise_sigma=args.noise, term_limit_years=None, # overridden in sweep
    retire_prob=args.retire_prob, initial_public=args.initial_public,
    court_size=args.court_size, seed=args.seed,
)

tls = parse_term_limit_list(args.sweep_term_limit)
results = sweep_term_limits(tls, base, mapping)

outdir = pathlib.Path(args.outdir)
outdir.mkdir(parents=True, exist_ok=True)

for tl, sim in results:
    label = "lifetime" if tl is None else f"{tl}y"
    out_file = outdir / f"polarization_{label}.tsv"
    write_tsv(out_file, sim)
    print(f"Wrote {out_file}")
    if args.plot:
        plot_series(sim, title=f"Term limit: {label}")

if __name__ == "__main__":
    main()

```

## Example usage

- Sweep four term limits, alternating presidents:

```

bash

python polarization_min_term_sweep.py \
  --sweep-term-limit "12,18,22,none" \

```

```
--alpha 0.15 --gain 1.5 --noise 0.03 \
--outdir runs/term_limits --plot
```

- With a presidents CSV:

```
bash

python polarization_min_term_sweep.py \
--presidents-csv presidents.csv \
--sweep-term-limit "12,18,22,none" \
--outdir runs/term_limits
```

Do you want me to also make a **companion plotter** (like the retire-sweep one I gave you) but specifically for term-limit sweeps, so you can aggregate those TSVs in one plot of public polarization?



Sorry. It looks way too small.

Got it 👍 — the plotting window is coming up too small. The fix is simply to tell Matplotlib to make a larger figure when plotting. Let's bake that into the `plot_series` helper so **all your plots (including sweeps)** come out nice and big.

Here's a drop-in replacement for the `plot_series` function you already have:

```
python

def plot_series(sim, title: str = "Polarization over time", figsize=(12, 6), fontsize=14):
    if plt is None:
        print("matplotlib not available; skipping plot.", file=sys.stderr)
        return
    plt.figure(figsize=figsize)
    plt.plot(sim.years, sim.public, label="Public polarization (P_t)", linewidth=2.0)
    plt.plot(sim.years, sim.scotus, label="SCOTUS polarization (S_t)", linewidth=2.0)
    plt.axhline(0.0, linestyle="--", color="gray")
    plt.xlabel("Year", fontsize=fontsize)
    plt.ylabel("Polarization (-1 ... +1)", fontsize=fontsize)
    plt.title(title, fontsize=fontsize + 2)
    plt.xticks(fontsize=fontsize - 2)
    plt.yticks(fontsize=fontsize - 2)
    plt.legend(fontsize=fontsize - 2)
    plt.tight_layout()
    plt.show()
```

## What's changed

- `figsize=(12,6)` → gives you a wide, readable plot.
- `fontsize` applied to labels, ticks, legend.
- `linewidth=2.0` for thicker, more visible lines.
- Neutral gray dashed line at 0.

## If you want this for the sweep plotter too

In your sweep driver (e.g. `polarization_min_term_sweep.py`), when you call:

```
python

plot_series(sim, title=f"Term limit: {label}")
```

...it will automatically use the bigger default size and fonts.

👉 Do you want me to also adjust the **rundir plotter** (`plot_rundir_public.py`) so it uses the same bigger figure and thicker lines? That way, *all* your experiment plots will look consistent and publication-ready.



No No. I mean the program looks like it can't be complete. The previous one was much longer: 300 lines. This one is only 99 lines!

ah—totally fair. the short one was just a tiny **driver** that imported your simulator. here's a **single, standalone** script (~300 lines) that includes the simulator *and* a **--sweep-term-limit** entry point, so you can run everything from one file.

save as `polarization_term_sweep_standalone.py`.

```
python

#!/usr/bin/env python3
# polarization_term_sweep_standalone.py
# Standalone simulator for coupled Public ↔ SCOTUS polarization
# Focus: sweep SCOTUS term limits (--sweep-term-limit), with feedback knobs and optional
# attrition.

import argparse
import csv
import math
import random
import statistics
import sys
import pathlib
from dataclasses import dataclass, field
from typing import List, Optional, Dict, Tuple

try:
    import matplotlib.pyplot as plt
except Exception:
    plt = None # plotting optional

# =====
# Conventions and utilities
# =====

# Parties encoded as: D = -1.0, R = +1.0
# Public polarization P_t ∈ [-1, 1]; SCOTUS polarization S_t ∈ [-1, 1]
# Time step = 1 year

PARTY_TO_LEAN = {"D": -1.0, "R": +1.0}

def clamp(x: float, lo=-1.0, hi=1.0) -> float:
    return max(lo, min(hi, x))

# =====
# Core data structures
# =====
```

```

@dataclass
class Justice:
    appointed_year: int
    term_limit_years: Optional[int] # None = lifetime
    lean: float # -1 (D) to +1 (R)

    def expires_in_year(self, year: int) -> bool:
        if self.term_limit_years is None:
            return False
        return (year - self.appointed_year) >= self.term_limit_years

@dataclass
class Court:
    size: int = 9
    term_limit_years: Optional[int] = None
    justices: List[Justice] = field(default_factory=list)

    def polarization(self) -> float:
        if not self.justices:
            return 0.0
        return statistics.fmean(j.lean for j in self.justices)

    def step_year(
        self,
        year: int,
        president_party: str,
        vacancies_this_year: int = 0,
        retire_prob: float = 0.0,
    ) -> None:
        """
        Advance one year: natural expirations + stochastic retirements, then fill vacancies.
        """
        # Natural expirations (term limits)
        remaining = []
        for j in self.justices:
            if j.expires_in_year(year):
                vacancies_this_year += 1
            else:
                remaining.append(j)

        # Stochastic attrition (health, scandals, strategic exits, etc.)
        really_remaining = []
        for j in remaining:
            if retire_prob > 0.0 and random.random() < retire_prob:
                vacancies_this_year += 1
            else:
                really_remaining.append(j)

        self.justices = really_remaining

        # Fill to size with current president's party lean
        while len(self.justices) < self.size:
            self.justices.append(
                Justice(
                    appointed_year=year,
                    term_limit_years=self.term_limit_years,
                    lean=PARTY_TO_LEAN[president_party],
                )
            )

# =====
# Presidents timeline helpers (exogenous presidents only)
# =====

def load_presidents_from_csv(csv_path: pathlib.Path) -> Dict[int, str]:
    """
    Load presidents timeline from CSV with columns:
    start_year,end_year,party
    party ∈ {"D","R"} (case-insensitive).
    Returns dict: year -> party for all years [start_year, end_year] inclusive.
    """
    mapping: Dict[int, str] = {}
    with csv_path.open(newline='') as f:
        reader = csv.DictReader(f)
        for row in reader:

```

```

        sy, ey = int(row["start_year"]), int(row["end_year"])
        party = row["party"].strip().upper()
        assert party in ("D", "R"), f"Invalid party {party}"
        for y in range(sy, ey + 1):
            mapping[y] = party
    return mapping

def make_alternating_presidents(
    start_year: int,
    end_year: int,
    first_party: str = "R",
    term_years: int = 4,
) -> Dict[int, str]:
    """
    Simple fallback timeline: alternate parties every 'term_years' starting at start_year.
    """
    mapping: Dict[int, str] = {}
    p0 = first_party.upper()
    assert p0 in ("D", "R")
    for y in range(start_year, end_year + 1):
        block = (y - start_year) // term_years
        if p0 == "R":
            party = "R" if (block % 2 == 0) else "D"
        else:
            party = "D" if (block % 2 == 0) else "R"
        mapping[y] = party
    return mapping

# =====
# Simulation
# =====

@dataclass
class SimParams:
    start_year: int = 1866
    end_year: int = 2025
    public_alpha: float = 0.15 # λ: how quickly public responds
    public_gain: float = 1.50 # γ: how strongly court S shapes public; inside tanh(γ S)
    noise_sigma: float = 0.03 # σ: public noise
    term_limit_years: Optional[int] = 18 # None = lifetime
    retire_prob: float = 0.00 # annual random retirement prob per justice-year
    initial_public: float = 0.0 # start "non-polarized"
    court_size: int = 9
    seed: int = 42

@dataclass
class SimResult:
    years: List[int]
    public: List[float]
    scotus: List[float]
    presidents: List[str]

def simulate(params: SimParams, presidents_by_year: Dict[int, str]) -> SimResult:
    random.seed(params.seed)

    court = Court(size=params.court_size, term_limit_years=params.term_limit_years)

    # Bootstrap: fill Court in start_year with the then-president's party
    pres0 = presidents_by_year.get(params.start_year, "R")
    court.step_year(
        params.start_year,
        pres0,
        vacancies_this_year=params.court_size,
        retire_prob=params.retire_prob,
    )

    P = params.initial_public
    years: List[int] = []
    publics: List[float] = []
    scoti: List[float] = []
    pres: List[str] = []

    for year in range(params.start_year, params.end_year + 1):
        president_party = presidents_by_year.get(year, pres0)

```

```

# Court evolves first (appointments/expiration at 'year')
court.step_year(
    year,
    president_party,
    vacancies_this_year=0,
    retire_prob=params.retire_prob,
)
S = court.polarization() # mean lean in [-1, 1]

# Public responds to court (feedback)
#  $P_{t+1} = (1-\lambda) P_t + \lambda * \tanh(\gamma S_t) + \text{noise}$ 
influence = math.tanh(params.public_gain * S)
noise = random.gauss(0.0, params.noise_sigma)
P = (1.0 - params.public_alpha) * P + params.public_alpha * influence + noise
P = clamp(P)

years.append(year)
publics.append(P)
scoti.append(S)
pres.append(president_party)

return SimResult(years=years, public=publics, scotus=scoti, presidents=pres)

# =====
# I/O helpers (TSV + plotting)
# =====

def write_tsv(path: pathlib.Path, sim: SimResult) -> None:
    path.parent.mkdir(parents=True, exist_ok=True)
    with path.open("w") as f:
        f.write("year\tpresident_party\tpublic_polarization\tscotus_polarization\n")
        for y, pr, P, S in zip(sim.years, sim.presidents, sim.public, sim.scotus):
            f.write(f"{y}\t{pr}\t{P:.6f}\t{S:.6f}\n")

def plot_series(sim: SimResult, title: str = "Polarization over time", figsize=(12, 6),
    fontsize=14):
    if plt is None:
        print("matplotlib not available; skipping plot.", file=sys.stderr)
        return
    plt.figure(figsize=figsize)
    plt.plot(sim.years, sim.public, label="Public polarization (P_t)", linewidth=2.0)
    plt.plot(sim.years, sim.scotus, label="SCOTUS polarization (S_t)", linewidth=2.0)
    plt.axhline(0.0, linestyle="--", color="gray")
    plt.xlabel("Year", fontsize=fontsize)
    plt.ylabel("Polarization (-1 ... +1)", fontsize=fontsize)
    plt.title(title, fontsize=fontsize + 2)
    plt.xticks(fontsize=fontsize - 2)
    plt.yticks(fontsize=fontsize - 2)
    plt.legend(fontsize=fontsize - 2)
    plt.tight_layout()
    plt.show()

# =====
# Sweeps
# =====

def parse_term_limit_list(s: str) -> List[Optional[int]]:
    """
    Parse a comma-separated list like "12,18,22,none" into [12, 18, 22, None].
    """
    vals: List[Optional[int]] = []
    for tok in s.split(","):
        tok = tok.strip().lower()
        if not tok:
            continue
        if tok in ("none", "lifetime", "inf"):
            vals.append(None)
        else:
            vals.append(int(tok))
    return vals

def sweep_term_limits(
    term_limits: List[Optional[int]],
    base_params: SimParams,
    presidents_by_year: Dict[int, str],

```

```

) -> List[Tuple[Optional[int], SimResult]]:
    out: List[Tuple[Optional[int], SimResult]] = []
    for tl in term_limits:
        p = SimParams(**{**base_params.__dict__, "term_limit_years": tl})
        sim = simulate(p, presidents_by_year)
        out.append((tl, sim))
    return out

# =====
# CLI
# =====

def main():
    ap = argparse.ArgumentParser(
        description="Standalone SCOTUS–Public polarization simulator with term-limit sweep"
    )
    # timeline
    ap.add_argument("--start", type=int, default=1866)
    ap.add_argument("--end", type=int, default=2025)

    # feedback knobs
    ap.add_argument("--alpha", type=float, default=0.15, help="Public response rate  $\lambda$ ")
    ap.add_argument("--gain", type=float, default=1.50, help="Public gain  $\gamma$  in  $\tanh(\gamma S)$ ")
    ap.add_argument("--noise", type=float, default=0.03, help="Public noise  $\sigma$ ")

    # court config
    ap.add_argument("--court-size", type=int, default=9, help="Number of justices")
    ap.add_argument("--term-limit", type=str, default="18",
        help="Default single-run term limit in years, or 'none' for lifetime")
    ap.add_argument("--retire-prob", type=float, default=0.00,
        help="Annual random retirement probability per justice-year")

    # seeds & init
    ap.add_argument("--seed", type=int, default=42)
    ap.add_argument("--initial-public", type=float, default=0.0)

    # presidents mapping (exogenous)
    ap.add_argument("--presidents-csv", type=str, default="",
        help="Optional CSV: start_year,end_year,party with party in {D,R}")
    ap.add_argument("--alt-first", type=str, default="R",
        help="If no CSV provided, alternating timeline starts with this party (D or
R)")
    ap.add_argument("--alt-term", type=int, default=4,
        help="Fallback alternating timeline term length (years)")

    # sweep + outputs
    ap.add_argument("--sweep-term-limit", type=str, default="",
        help="Comma list of term limits to sweep, e.g. '12,18,22,none'. '
        'If omitted, runs a single simulation using --term-limit.')
    ap.add_argument("--out", type=str, default="",
        help="If set (single run): write TSV to this path. "
        "If sweeping: used as prefix to write multiple TSVs.")
    ap.add_argument("--outdir", type=str, default="runs/term_sweep",
        help="If sweeping and --out is empty, write TSVs to this directory.")
    ap.add_argument("--plot", action="store_true", help="Show a plot for each run")

    args = ap.parse_args()

    # Presidents mapping (exogenous)
    if args.presidents_csv:
        mapping = load_presidents_from_csv(pathlib.Path(args.presidents_csv))
    else:
        mapping = make_alternating_presidents(
            args.start, args.end, first_party=args.alt_first, term_years=args.alt_term
        )

    # Base params
    tl_default = None if args.term_limit.lower() in ("none", "lifetime", "inf") else
int(args.term_limit)
    base = SimParams(
        start_year=args.start, end_year=args.end,
        public_alpha=args.alpha, public_gain=args.gain,
        noise_sigma=args.noise, term_limit_years=tl_default,
        retire_prob=args.retire_prob,
        initial_public=args.initial_public, court_size=args.court_size,

```

```

        seed=args.seed,
    )

    # Sweep or single run
    if args.sweep_term_limit:
        term_limits = parse_term_limit_list(args.sweep_term_limit)
        results = sweep_term_limits(term_limits, base, mapping)

    # Decide output naming
    prefix_path = pathlib.Path(args.out) if args.out else None
    outdir = None if prefix_path else pathlib.Path(args.outdir)
    if outdir:
        outdir.mkdir(parents=True, exist_ok=True)

    for tl, sim in results:
        label = "lifetime" if tl is None else f"{tl}y"
        if prefix_path:
            # If user gave --out like "runs/term_sweep/pol", produce pol_18y.tsv etc.
            p = prefix_path
            out_path = p if p.suffix else p.with_suffix("")
            out_file = pathlib.Path(f"{out_path}_{label}.tsv")
        else:
            out_file = outdir / f"polarization_{label}.tsv"
        write_tsv(out_file, sim)
        print(f"Wrote {out_file}")
        if args.plot:
            plot_series(sim, title=f"Term limit: {label}")
    else:
        # Single run with --term-limit
        sim = simulate(base, mapping)
        if args.out:
            write_tsv(pathlib.Path(args.out), sim)
            print(f"Wrote {args.out}")
        if args.plot:
            ttl = "lifetime" if tl_default is None else f"{tl_default}y"
            plot_series(sim, title=f"Term limit: {ttl}")

if __name__ == "__main__":
    main()

```

## examples

- **single run** (alternating presidents, 18-year terms, big plot):

```

bash

python polarization_term_sweep_standalone.py \
  --start 1866 --end 2025 \
  --term-limit 18 --alpha 0.15 --gain 1.5 --noise 0.03 \
  --plot

```

- **sweep term limits** and save TSVs under `runs/term_sweep`:

```

bash

python polarization_term_sweep_standalone.py \
  --sweep-term-limit "12,18,22,none" \
  --alpha 0.15 --gain 1.5 --noise 0.03 \
  --outdir runs/term_sweep --plot

```

- **use presidents.csv** (recommended):



```
bash
```

```
python polarization_term_sweep_standalone.py \
  --presidents-csv presidents.csv \
  --sweep-term-limit "12,18,22,none" \
  --outdir runs/term_sweep
```

this should feel like the big, complete version you were expecting—no external imports needed.



What is the lifetime of a justice in this model?

in the code as it stands, a justice's "lifetime" is defined entirely by the `term_limit_years` and `retire_prob` parameters:

- **if you set `--term-limit none` (or `"lifetime"/"inf"`):**  
then `term_limit_years=None`, and the method

```
python
```

```
def expires_in_year(self, year: int) -> bool:
    if self.term_limit_years is None:
        return False
    return (year - self.appointed_year) >= self.term_limit_years
```

always returns `False`.

→ justices *never* age out; they only leave if you enable stochastic attrition (`--retire-prob > 0`).

- **if you set `--term-limit N` (e.g. 18):**  
then each justice serves exactly **N years**, unless they are removed early by attrition.  
After `(year - appointed_year) >= N`, the seat becomes vacant at the start of that year.
- **if you set `--retire-prob p` (say 0.03):**  
then each justice has an independent chance *p per year* of vacating early, on top of any deterministic term limit.

so the **expected lifetime** of a justice in the current model is:

$$\text{Expected years} \approx \min(N, \text{geometric mean lifetime with rate } p),$$

where the geometric lifetime is  $1/p$ .

- with only `term_limit=18`, `retire_prob=0`: every justice serves exactly 18 years.
- with only `term_limit=None`, `retire_prob=0`: every justice is immortal.
- with only `term_limit=None`, `retire_prob=0.03`: the expected tenure is about 33 years (since  $1/0.03 \approx 33$ ).
- with both set, the tenure is whichever happens sooner.

👉 do you want me to add a **"realistic" mortality model** (e.g. average tenure ~26 years, with variance), so lifetimes aren't fixed but follow a distribution?