# ACCELERATION

# DETERMINISTIC OPTIMAL GROWTH MODEL

- Consider the following social planner's problem:

$$\max_{c_t, k_{t+1}} \sum_{t=0}^{\infty} \beta^t \frac{c_t^{1-\gamma}}{1-\gamma},$$

s.t. $c_t + k_{t+1} = k_t^{\alpha} + (1-\delta)k_t$, and $k_0 > 0$ is given.

- The recursive formulation is given by

$$v(k_t) = \max_{c_t, k_{t+1}} [\frac{c_t^{1-\gamma}}{1-\gamma} + \beta v(k_{t+1})],$$

s.t. $c_t + k_{t+1} = k_t^{\alpha} + (1-\delta)k_t$, and $k_0 > 0$ is given. Or

$$v(k_t) = \max_{c_t, k_{t+1}} \{\frac{[k_t^{\alpha} + (1-\delta)k_t - k_{t+1}]^{1-\gamma}}{1-\gamma} + \beta v(k_{t+1})\}$$

# TRADEOFF: PRECISION VS DURATION

| grid number | maximum error | Duration |
|---|---|---|
| 100 | 0.0768 | 1.56 |
| 200 | 0.0364 | 6.47 |
| 400 | 0.0192 | 25.41 |
| 800 | 0.0118 | 101.88 |
| 1600 | 0.0059 | 415.94 |

- The curse of dimensionality occurs when dealing with multiple state variables (eg., stochastic models).

# WHY IS THE BENCHMARK ALGORITHM SO SLOW?

- using lots of loops (scalar operation)
- interpreted language is slow
    - interpreted language vs compiled languages
- code is executing on a single CPU core
- algorithm is not optimized

# POSSIBLE SOLUTIONS

| reasons | solutions |
|---|---|
| scalar operation | vectorization |
| slow interpreted language | compilation |
| executing on a single core | parallelization |
| algorithm is not optimized | optimization |

# ACCELERATION

- vectorization
- compilation
- parallelization
    - GPU acceleration
- algorithm optimization
    - standard tricks: concavity and monotonicity
    - multigrid method
    - interpolation
    - beyond value function iteration

# VECTORIZATION

- an operation that operates on entire arrays (vector or matrix), also called array programming
    - calculate $v$ as a vector function, treat $k$ as a vector
        - for each $k_t$, calculate the objective function for all $k_{t+1}$
        - calculate all $k_t$ together
- In Python, most functions can operate on vectors directly.
- tradeoff
    - RAM demanding for huge vectors
    - programming language dependent
    - not possible to implement lower-level optimization techniques

# EXAMPLE

```
def CRRA(c):
    return c**(1-gamma)/(1-gamma)

def u(c):
    res = np.empty_like(c, dtype=float)
    positive_mask = c > 0
    res[positive_mask] = CRRA(c[positive_mask])
    res[~positive_mask] = -np.inf
    return res

def budget(k, x):
    return np.subtract.outer(k**alpha + (1-delta)*k, x)

c = budget(kgrid, kgrid)  # consumption matrix, calculate once for all
utility = u(c)  # utility matrix, calculate once for all

def V_update(V_next):
    rhs = utility + beta * V_next
    V_max = np.max(rhs,axis=1)
    g_index = np.argmax(rhs, axis=1)
    g_k = kgrid[g_index]
    return V_max, g_k
```

# COMPARISON OF TIME

| grid | benchmark | vectorization |
|---|---|---|
| 100 | 1.56 | 0.005 |
| 200 | 6.47 | 0.021 |
| 400 | 25.41 | 0.040 |
| 800 | 101.88 | 0.129 |
| 1600 | 415.94 | 0.886 |

# COMPILATION

- Pros: substantial speed gain, comparable to that of C/Fortran
- Cons:
    - requires additional compilation time
    - imposes limitations on coding
    - OS/platform/hardware dependent

# A SIMPLE WAY: USE `Numba` JIT

1. installation: `conda install numba`
2. import: `from numba import njit`
3. usage

```
@njit
def test():
    pass
```

or

```
def test():
    pass

test = njit(test)
```

The second method is recommended, since it can be easily switched off:

```
bytecode = True

if bytecode:
    test = njit(test)
```

# COMPARISON OF TIME

| grid | benchmark | bytecode |
|---|---|---|
| 100 | 1.56 | 0.025 |
| 200 | 6.47 | 0.098 |
| 400 | 25.41 | 0.388 |
| 800 | 101.88 | 1.549 |
| 1600 | 415.94 | 6.262 |

# PARALLELIZATION

- Pros: allows for parallel execution of independent computations
    - GPU computing
- Cons:
    - not suitable for performing serial dependent computations
    - hardware dependent
    - requires specific modifications to the algorithm
    - communication cost might be very high

# A SIMPLE WAY: USE `Numba` JIT

1. installation: `conda install numba`
2. import: `from numba import njit, prange`
3. usage

```
@njit(parallel=True)
def test():
    for i in prange(n):
        pass
```

or

```
def test():
    pass

test = njit(test, parallel=True)
```

The second method is recommended, since it can be easily switched off:

```
parallel = True
test = njit(test, parallel=parallel)
```

# COMPARISON OF TIME

| grid | benchmark | bytecode | parallelization |
|---|---|---|---|
| 100 | 1.56 | 0.025 | 0.018 |
| 200 | 6.47 | 0.098 | 0.039 |
| 400 | 25.41 | 0.388 | 0.087 |
| 800 | 101.88 | 1.549 | 0.269 |
| 1600 | 415.94 | 6.262 | 1.382 |

# ALGORITHM OPTIMIZATION

- standard tricks: concavity and monotonicity
- multigrid method
- interpolation
- beyond value function iteration

# BENCHMARK ALGORITHM

```
def V_max(k_index, V):
    k = kgrid[k_index]
    k_bound = budget(k, 0)
    V_max = -np.inf
    for j in range(n):
        k_next = kgrid[j]
        V_next = V[j]
        if k_next < k_bound:
            V_new = V_current(k, k_next, V_next)
            if V_new > V_max:
                V_max = V_new
                g_k = k_next
    return V_max, g_k
```

# OPTIMIZED ALGORITHM

```
def V_max(k_index, V):
    k = kgrid[k_index]
    k_bound = budget(k, 0)
    V_max = -np.inf
    for j in range(n):
        k_next = kgrid[j]
        V_next = V[j]
        if k_next < k_bound:
            V_new = V_current(k, k_next, V_next)
            if V_new > V_max:
                V_max = V_new
                g_k = k_next
            else:
                break
    return V_max, g_k
```

# COMPARISON OF TIME

| grid | benchmark | concavity |
|---|---|---|
| 100 | 1.56 | 0.911 |
| 200 | 6.47 | 3.621 |
| 400 | 25.41 | 14.596 |
| 800 | 101.88 | 57.786 |
| 1600 | 415.94 | 242.854 |

- can work with compilation and parallelization

# MONOTONICITY OF POLICY FUNCTION

- For each $k_t$, we must have $g(k_t) \geq g(k_{t+1})$. Hence we only need to search between $g(k_{t+1})$ and $k_{max}$.
    - Pros: reduce half of the computational workload
    - Cons: serial dependent, cannot be parallelized

# BENCHMARK ALGORITHM

```
def V_max(k_index, V):
    k = kgrid[k_index]
    k_bound = budget(k, 0)
    V_max = -np.inf
    for j in range(n):
        k_next = kgrid[j]
        V_next = V[j]
        if k_next < k_bound:
            V_new = V_current(k, k_next, V_next)
            if V_new > V_max:
                V_max = V_new
                g_k = k_next
    return V_max, g_k
```

# OPTIMIZED ALGORITHM

```
def V_max(k_index, k_start, V):
    k = kgrid[k_index]
    k_bound = budget(k, 0)
    V_max = -np.inf
    for j in range(k_start, n):
        k_next = kgrid[j]
        V_next = V[j]
        if k_next < k_bound:
            V_new = V_current(k, k_next, V_next)
            if V_new > V_max:
                V_max = V_new
                g_k = k_next
                k_start = j
    return V_max, g_k, k_start
```

# COMPARISON OF TIME

| grid | benchmark | concavity | monotonicity |
|---|---|---|---|
| 100 | 1.56 | 0.911 | 0.601 |
| 200 | 6.47 | 3.621 | 2.397 |
| 400 | 25.41 | 14.596 | 9.497 |
| 800 | 101.88 | 57.786 | 37.096 |
| 1600 | 415.94 | 242.854 | 151.907 |

- can work with concavity
    - How much speed improvement can we attain if we make use of both concavity and monotonicity?

# COMPARISON OF TIME

| grid | benchmark | concavity | monotonicity | both concavity and monotonicity |
|---|---|---|---|---|
| 100 | 1.56 | 0.911 | 0.601 | 0.067 |
| 200 | 6.47 | 3.621 | 2.397 | 0.150 |
| 400 | 25.41 | 14.596 | 9.497 | 0.295 |
| 800 | 101.88 | 57.786 | 37.096 | 0.588 |
| 1600 | 415.94 | 242.854 | 151.907 | 1.123 |

- substantial speed gains can be achieved with both concavity and monotonicity

# MULTIGRID METHOD

- Solve the model under a relatively sparse grid with mesh size $h_0$, which results in a fixed point $V^{h_0}$ (**converge very fast!**).
- $V^{h_0}$ can then be used as initial condition for the value function iteration with a finer mesh size $h_1 < h_0$.
    - Since the distance between $V^{h_0}$ and the true fixed point is very close, it requires significantly fewer steps to achieve convergence.

# COMPARISON OF TIME

| algorithm | grid number | duration | Iterations |
|---|---|---|---|
| benchmark | 1600 | 416 | 284 |
| multigrid | 100+1600 | 1.5+47 | 284+36 |

- The time has been reduced by 88%.
- can work with any optimization method

# INTERPOLATION

- use iterpolation to make discrete function continuous
    - pros: It is possible to achieve a desired level of precision using sparser grids, which results in reduced computation time compared to the standard algorithm.
    - cons: Utilizing lower-level optimization techniques can be challenging, unless you are willing to write the interpolation and maximization code from scratch.

# CODE EXAMPLE

```
from scipy.optimize import minimize_scalar

def V_current(k_next, k, kgrid, V):
    c = budget(k, k_next)
    V_interp = np.interp(k_next, kgrid, V)
    res = -(u(c) + beta * V_interp)
    return res

def V_max(k_index, kgrid, V):
    k = kgrid[k_index]
    k_bound = budget(k, 0)
    res = minimize_scalar(fun=V_current, bounds=(
        kmin, k_bound), args=(k, kgrid, V))
    V_max = -res.fun
    g_k = res.x
    return V_max, g_k
```

# COMPARISON OF TIME

| grid | benchmark | interpolation |
|---|---|---|
| 100 | 1.56 | 3.58 |
| 200 | 6.47 | 6.74 |
| 400 | 25.41 | 12.79 |
| 800 | 101.88 | 25.52 |
| 1600 | 415.94 | 50.56 |

- can work with both compilation and parallelization

# NUMBER OF GRIDS NEEDED TO ACHIEVE THE SAME PRICISION

| benchmark: grid | benchmark: error | interpolation: grid | interpolation: error |
|---:|---:|---:|---:|
| 800 | 0.008 | 140 | 0.008 |
| 1600 | 0.003 | 350 | 0.003 |

- Aside from the interpolation method, the benchmark algorithm maintains the same level of precision as the other algorithms mentioned above.
- It is possible to reduced computation time by using sparser grids.

# SUMMARY OF DURATION

- the same precision as the benchmark model with 1600 grids
- without compilation or parallelization

| algorithm | compilation | parallelization | duration |
|---|---|---|---|
| benchmark | | | 415.940 |
| concavity | | | 242.854 |
| monotonicity | | | 151.907 |
| multigrids | | | 49.950 |
| interpolation (350) | | | 11.247 |
| interpolation with multigrids (350) | | | 3.926 |
| concavity and monotonicity | | | 1.123 |
| vectorization | | | 0.886 |
| multigrids with concavity and monotonicity | | | 0.209 |

# SUMMARY OF DURATION

- the same precision as the benchmark model with 1600 grids
- with compilation or/and parallelization

| algorithm | compilation | parallelization | duration |
|---|---|---|---|
| monotonicity | Y | | 2.508 |
| benchmark | Y | Y | 1.382 |
| concavity | Y | Y | 1.036 |
| multigrids | Y | Y | 0.149 |
| interpolation (350) | Y | Y | 0.036 |
| interpolation with multigrids (350) | Y | Y | 0.023 |
| concavity and monotonicity | Y | | 0.023 |
| multigrids with concavity and monotonicity | Y | | 0.005 |

# BEYOND VALUE FUNCTION ITERATION

- policy function iteration
- time iteration
- endogenous grid
- machine learning
- etc