High-Performance Computing

Stepan Gordeev

September 17, 2020

Outline

1. Parallelization

2. BlueHive

3. General Coding

4. Julia Recommendations

5. Exposition of Advanced Techniques

Parallelization

Overview

- All consumer CPUs have multiple cores, but by default your code will run sequentially on one core!
- Can write code that will explicitly split tasks across cores.
- Work is split between cores, they all work simultaneously, then they pool results together.
- Matlab and Julia will run certain things in parallel on their own, e.g. linear algebra.
- Split sequential code across two processors ↛ 2x speedup
    - Parallelization adds overhead: CPU does extra work to split the task across cores and then to collect the results.

Overview

- All consumer CPUs have multiple cores, but by default your code will run sequentially on one core!
- Can write code that will explicitly split tasks across cores.
- Work is split between cores, they all work simultaneously, then they pool results together.
- Matlab and Julia will run certain things in parallel on their own, e.g. linear algebra.
- Split sequential code across two processors $\nrightarrow$ 2x speedup
    - Parallelization adds overhead: CPU does extra work to split the task across cores and then to collect the results.

Overview

- All consumer CPUs have multiple cores, but by default your code will run sequentially on one core!
- Can write code that will explicitly split tasks across cores.
- Work is split between cores, they all work simultaneously, then they pool results together.
- Matlab and Julia will run certain things in parallel on their own, e.g. linear algebra.
- Split sequential code across two processors $\nrightarrow$ 2x speedup
    - Parallelization adds overhead: CPU does extra work to split the task across cores and then to collect the results.

Overview

- All consumer CPUs have multiple cores, but by default your code will run sequentially on one core!
- Can write code that will explicitly split tasks across cores.
- Work is split between cores, they all work simultaneously, then they pool results together.
- Matlab and Julia will run certain things in parallel on their own, e.g. linear algebra.
- Split sequential code across two processors $\nrightarrow$ 2x speedup
  - Parallelization adds overhead: CPU does extra work to split the task across cores and then to collect the results.

Overview

- All consumer CPUs have multiple cores, but by default your code will run sequentially on one core!
- Can write code that will explicitly split tasks across cores.
- Work is split between cores, they all work simultaneously, then they pool results together.
- Matlab and Julia will run certain things in parallel on their own, e.g. linear algebra.
- Split sequential code across two processors ↛ 2x speedup
    - Parallelization adds overhead: CPU does extra work to split the task across cores and then to collect the results.

When to Parallelize

- Cannot parallelize tasks that have to be done sequentially
  - Outer loop of VFI (go through V_l): No (V_{l+1} depends on V_l).
  - Inner loop of VFI (go through k_i): Yes (k_{i+1} does not depend on k_i). Send the $V_{l+1}(k_i) = \max \dots$ problem to each core, then collect results before moving on to V_{l+2}.
- Run $V_{l+1}(k_i) = \max \dots$ tasks in parallel:
  - e.g. CPU has 4 cores, grid is $[k_1, \dots, k_{100}]$
  - core 1 solves $V_{l+1}(k_i) = \max \dots \quad$ for $i = 1, \dots, 25$
  - core 2 solves $V_{l+1}(k_i) = \max \dots \quad$ for $i = 26, \dots, 50$
  - ...
  - CPU collects four pairs of $V'(k)$, $g(k)$, merges into one pair
  - Move on to V_{l+2}

When to Parallelize

- Cannot parallelize tasks that have to be done sequentially
  - Outer loop of VFI (go through V_l): No (V_{l+1} depends on V_l).
  - Inner loop of VFI (go through k_i): Yes (k_{i+1} does not depend on k_i). Send the $V_{l+1}(k_i) = \max \dots$ problem to each core, then collect results before moving on to V_{l+2}.
- Run $V_{l+1}(k_i) = \max \dots$ tasks in parallel:
  - e.g. CPU has 4 cores, grid is $[k_1, \dots, k_{100}]$
  - core 1 solves $V_{l+1}(k_i) = \max \dots \quad$ for $i = 1, \dots, 25$
  - core 2 solves $V_{l+1}(k_i) = \max \dots \quad$ for $i = 26, \dots, 50$
  - ...
  - CPU collects four pairs of $V'(k)$, $g(k)$, merges into one pair
  - Move on to V_{l+2}

When to Parallelize

- Cannot parallelize tasks that have to be done sequentially
  - Outer loop of VFI (go through V_l): No (V_{l+1} depends on V_l).
  - Inner loop of VFI (go through k_i): Yes (k_{i+1} does not depend on k_i). Send the $V_{l+1}(k_i) = \max \dots$ problem to each core, then collect results before moving on to V_{l+2}.
- Run $V_{l+1}(k_i) = \max \dots$ tasks in parallel:
  - e.g. CPU has 4 cores, grid is $[k_1, ..., k_{100}]$
  - core 1 solves $V_{l+1}(k_i) = \max \dots \quad$ for $i = 1, ..., 25$
  - core 2 solves $V_{l+1}(k_i) = \max \dots \quad$ for $i = 26, ..., 50$
  - ...
  - CPU collects four pairs of $V'(k)$, $g(k)$, merges into one pair
  - Move on to V_{l+2}

When to Parallelize

- Cannot parallelize tasks that have to be done sequentially
  - Outer loop of VFI (go through V_l): No (V_{l+1} depends on V_l).
  - Inner loop of VFI (go through k_i): Yes (k_{i+1} does not depend on k_i). Send the $V_{l+1}(k_i) = \max \dots$ problem to each core, then collect results before moving on to V_{l+2}.
- Run $V_{l+1}(k_i) = \max \dots$ tasks in parallel:
  - e.g. CPU has 4 cores, grid is $[k_1, \dots, k_{100}]$
  - core 1 solves $V_{l+1}(k_i) = \max \dots \quad$ for $i = 1, \dots, 25$
  - core 2 solves $V_{l+1}(k_i) = \max \dots \quad$ for $i = 26, \dots, 50$
  - ...
  - CPU collects four pairs of $V'(k)$, $g(k)$, merges into one pair
  - Move on to V_{l+2}

When to Parallelize

- Cannot parallelize tasks that have to be done sequentially
  - Outer loop of VFI (go through V_l): No (V_{l+1} depends on V_l).
  - Inner loop of VFI (go through k_i): Yes (k_{i+1} does not depend on k_i). Send the $V_{l+1}(k_i) = \max \ldots$ problem to each core, then collect results before moving on to V_{l+2}.
- Run $V_{l+1}(k_i) = \max \ldots$ tasks in parallel:
  - e.g. CPU has 4 cores, grid is $[k_1, ..., k_{100}]$
  - core 1 solves $V_{l+1}(k_i) = \max \ldots \quad$ for $i = 1, ..., 25$
  - core 2 solves $V_{l+1}(k_i) = \max \ldots \quad$ for $i = 26, ..., 50$
  - ...
  - CPU collects four pairs of $V'(k)$, $g(k)$, merges into one pair
  - Move on to V_{l+2}

When to Parallelize

- Cannot parallelize tasks that have to be done sequentially
  - Outer loop of VFI (go through V_l): No (V_{l+1} depends on V_l).
  - Inner loop of VFI (go through k_i): Yes (k_{i+1} does not depend on k_i). Send the $V_{l+1}(k_i) = \max \ldots$ problem to each core, then collect results before moving on to V_{l+2}.
- Run $V_{l+1}(k_i) = \max \ldots$ tasks in parallel:
  - e.g. CPU has 4 cores, grid is $[k_1, \ldots, k_{100}]$
  - core 1 solves $V_{l+1}(k_i) = \max \ldots \quad$ for $i = 1, \ldots, 25$
  - core 2 solves $V_{l+1}(k_i) = \max \ldots \quad$ for $i = 26, \ldots, 50$
  - ...
  - CPU collects four pairs of $V'(k)$, $g(k)$, merges into one pair
  - Move on to V_{l+2}

When to Parallelize

- Cannot parallelize tasks that have to be done sequentially
  - Outer loop of VFI (go through V_l): No (V_{l+1} depends on V_l).
  - Inner loop of VFI (go through k_i): Yes (k_{i+1} does not depend on k_i). Send the $V_{l+1}(k_i) = \max \dots$ problem to each core, then collect results before moving on to V_{l+2}.
- Run $V_{l+1}(k_i) = \max \dots$ tasks in parallel:
  - e.g. CPU has 4 cores, grid is $[k_1, \dots, k_{100}]$
  - core 1 solves $V_{l+1}(k_i) = \max \dots \quad$ for $i = 1, \dots, 25$
  - core 2 solves $V_{l+1}(k_i) = \max \dots \quad$ for $i = 26, \dots, 50$
  - ...
  - CPU collects four pairs of $V'(k)$, $g(k)$, merges into one pair
  - Move on to V_{l+2}

When to Parallelize

- Cannot parallelize tasks that have to be done sequentially
  - Outer loop of VFI (go through V_l): No (V_{l+1} depends on V_l).
  - Inner loop of VFI (go through k_i): Yes (k_{i+1} does not depend on k_i). Send the $V_{l+1}(k_i) = \max \ldots$ problem to each core, then collect results before moving on to V_{l+2}.
- Run $V_{l+1}(k_i) = \max \ldots$ tasks in parallel:
  - e.g. CPU has 4 cores, grid is $[k_1, ..., k_{100}]$
  - core 1 solves $V_{l+1}(k_i) = \max \ldots \quad$ for $i = 1, ..., 25$
  - core 2 solves $V_{l+1}(k_i) = \max \ldots \quad$ for $i = 26, ..., 50$
  - ...
  - CPU collects four pairs of $V'(k)$, $g(k)$, merges into one pair
  - Move on to V_{l+2}

When to Parallelize

- Cannot parallelize tasks that have to be done sequentially
  - Outer loop of VFI (go through V_l): No (V_{l+1} depends on V_l).
  - Inner loop of VFI (go through k_i): Yes (k_{i+1} does not depend on k_i). Send the $V_{l+1}(k_i) = \max \ldots$ problem to each core, then collect results before moving on to V_{l+2}.
- Run $V_{l+1}(k_i) = \max \ldots$ tasks in parallel:
  - e.g. CPU has 4 cores, grid is $[k_1, ..., k_{100}]$
  - core 1 solves $V_{l+1}(k_i) = \max \ldots \quad \text{for } i = 1, ..., 25$
  - core 2 solves $V_{l+1}(k_i) = \max \ldots \quad \text{for } i = 26, ..., 50$
  - ...
  - CPU collects four pairs of $V'(k)$, $g(k)$, merges into one pair
  - Move on to V_{l+2}

When to Parallelize

- Cannot parallelize tasks that have to be done sequentially
  - Outer loop of VFI (go through V_l): No (V_{l+1} depends on V_l).
  - Inner loop of VFI (go through k_i): Yes (k_{i+1} does not depend on k_i). Send the $V_{l+1}(k_i) = \max \dots$ problem to each core, then collect results before moving on to V_{l+2}.
- Run $V_{l+1}(k_i) = \max \dots$ tasks in parallel:
  - e.g. CPU has 4 cores, grid is $[k_1, \dots, k_{100}]$
  - core 1 solves $V_{l+1}(k_i) = \max \dots \quad \text{for } i = 1, \dots, 25$
  - core 2 solves $V_{l+1}(k_i) = \max \dots \quad \text{for } i = 26, \dots, 50$
  - ...
  - CPU collects four pairs of $V'(k)$, $g(k)$, merges into one pair
  - Move on to V_{l+2}

What to Parallelize

- Split tasks to maximize computation by cores and minimize communication between cores

-
  ```
  for i=1:100
      for j=1:100
          ...
      end
  end
  ```
- if parallelize outer `i` loop: 100 tasks, each has 1/100th work
    - CPU sends out work to cores and collects results just once
- if parallelize inner `j` loop: 10,000 tasks, each has 1/10,000th work
    - CPU sends out work to cores and collects results 100 times
- outer is faster: less overhead

What to Parallelize

- Split tasks to maximize computation by cores and minimize communication between cores
-
  ```
  for i=1:100
      for j=1:100
          ...
      end
  end
  ```
- if parallelize outer `i` loop: 100 tasks, each has 1/100th work
  - CPU sends out work to cores and collects results just once
- if parallelize inner `j` loop: 10,000 tasks, each has 1/10,000th work
  - CPU sends out work to cores and collects results 100 times
- outer is faster: less overhead

What to Parallelize

- Split tasks to maximize computation by cores and minimize communication between cores
-
  ```
  for i=1:100
      for j=1:100
          ...
      end
  end
  ```
- if parallelize outer `i` loop: 100 tasks, each has 1/100th work
  - CPU sends out work to cores and collects results just once
- if parallelize inner `j` loop: 10,000 tasks, each has 1/10,000th work
  - CPU sends out work to cores and collects results 100 times
- outer is faster: less overhead

What to Parallelize

- Split tasks to maximize computation by cores and minimize communication between cores

-
  ```
  for i=1:100
      for j=1:100
          ...
      end
  end
  ```
- if parallelize outer `i` loop: 100 tasks, each has 1/100th work
    - CPU sends out work to cores and collects results just once
- if parallelize inner `j` loop: 10,000 tasks, each has 1/10,000th work
    - CPU sends out work to cores and collects results 100 times
- outer is faster: less overhead

What to Parallelize

- Split tasks to maximize computation by cores and minimize communication between cores
-
  ```
  for i=1:100
      for j=1:100
          ...
      end
  end
  ```
- if parallelize outer `i` loop: 100 tasks, each has 1/100th work
  - CPU sends out work to cores and collects results just once
- if parallelize inner `j` loop: 10,000 tasks, each has 1/10,000th work
  - CPU sends out work to cores and collects results 100 times
- outer is faster: less overhead

What to Parallelize

- Split tasks to maximize computation by cores and minimize communication between cores
-
  ```
  for i=1:100
      for j=1:100
          ...
      end
  end
  ```
- if parallelize outer `i` loop: 100 tasks, each has 1/100th work
  - CPU sends out work to cores and collects results just once
- if parallelize inner `j` loop: 10,000 tasks, each has 1/10,000th work
  - CPU sends out work to cores and collects results 100 times
- outer is faster: less overhead

What to Parallelize

- Split tasks to maximize computation by cores and minimize communication between cores

-
  ```
  for i=1:100
      for j=1:100
          ...
      end
  end
  ```
- if parallelize outer `i` loop: 100 tasks, each has 1/100th work
    - CPU sends out work to cores and collects results just once
- if parallelize inner `j` loop: 10,000 tasks, each has 1/10,000th work
    - CPU sends out work to cores and collects results 100 times
- outer is faster: less overhead

How to Parallelize

- Julia: several ways, threads are simplest.
- Julia needs to be started with a pre-set number of threads.
    - Juno will pass this setting automatically.
    - Explicitly: Preferences → Packages → Julia Client → Number of Threads
- Verify threads setting: `Threads.nthreads()`
- Put `Threads.@threads` before a loop you want to make parallel:

```
Threads.@threads for i=1:n
   ...
end
```

- In Matlab, use `parpool` to control the number of worker processes and replace `for` with `parfor`.

How to Parallelize

- Julia: several ways, threads are simplest.
- Julia needs to be started with a pre-set number of threads.
  - Juno will pass this setting automatically.
  - Explicitly: Preferences → Packages → Julia Client → Number of Threads
- Verify threads setting: `Threads.nthreads()`
- Put `Threads.@threads` before a loop you want to make parallel:

```
Threads.@threads for i=1:n
    ...
end
```

- In Matlab, use `parpool` to control the number of worker processes and replace `for` with `parfor`.

How to Parallelize

- Julia: several ways, threads are simplest.
- Julia needs to be started with a pre-set number of threads.
  - Juno will pass this setting automatically.
  - Explicitly: Preferences → Packages → Julia Client → Number of Threads
- Verify threads setting: `Threads.nthreads()`
- Put `Threads.@threads` before a loop you want to make parallel:

```
Threads.@threads for i=1:n
    ...
end
```

- In Matlab, use `parpool` to control the number of worker processes and replace `for` with `parfor`.

How to Parallelize

- Julia: several ways, threads are simplest.
- Julia needs to be started with a pre-set number of threads.
    - Juno will pass this setting automatically.
    - Explicitly: Preferences → Packages → Julia Client → Number of Threads
- Verify threads setting: `Threads.nthreads()`
- Put `Threads.@threads` before a loop you want to make parallel:

```
Threads.@threads for i=1:n
   ...
end
```

- In Matlab, use `parpool` to control the number of worker processes and replace `for` with `parfor`.

How to Parallelize

- Julia: several ways, threads are simplest.
- Julia needs to be started with a pre-set number of threads.
  - Juno will pass this setting automatically.
  - Explicitly: Preferences → Packages → Julia Client → Number of Threads
- Verify threads setting: `Threads.nthreads()`
- Put `Threads.@threads` before a loop you want to make parallel:

```
Threads.@threads for i=1:n
    ...
end
```

- In Matlab, use `parpool` to control the number of worker processes and replace `for` with `parfor`.

BlueHive

Overview

- BlueHive is a university cluster running RHEL (a Linux distribution)
- Hundreds of compute nodes, each with 12-64 processors.
- Info: `info.circ.rochester.edu`
- Uses of BlueHive
  - Large parallel problems: exploit huge number of cores
  - Work with large datasets: exploit huge RAM
  - Expensive specialized software unavailable on lab computers

Overview

- BlueHive is a university cluster running RHEL (a Linux distribution)
- Hundreds of compute nodes, each with 12-64 processors.
- Info: `info.circ.rochester.edu`
- Uses of BlueHive
  - Large parallel problems: exploit huge number of cores
  - Work with large datasets: exploit huge RAM
  - Expensive specialized software unavailable on lab computers

Overview

- BlueHive is a university cluster running RHEL (a Linux distribution)
- Hundreds of compute nodes, each with 12-64 processors.
- Info: `info.circ.rochester.edu`
- Uses of BlueHive
    - Large parallel problems: exploit huge number of cores
    - Work with large datasets: exploit huge RAM
    - Expensive specialized software unavailable on lab computers

Overview

- BlueHive is a university cluster running RHEL (a Linux distribution)
- Hundreds of compute nodes, each with 12-64 processors.
- Info: `info.circ.rochester.edu`
- Uses of BlueHive
  - Large parallel problems: exploit huge number of cores
  - Work with large datasets: exploit huge RAM
  - Expensive specialized software unavailable on lab computers

Overview

- BlueHive is a university cluster running RHEL (a Linux distribution)
- Hundreds of compute nodes, each with 12-64 processors.
- Info: `info.circ.rochester.edu`
- Uses of BlueHive
  - Large parallel problems: exploit huge number of cores
  - Work with large datasets: exploit huge RAM
  - Expensive specialized software unavailable on lab computers

Overview

- BlueHive is a university cluster running RHEL (a Linux distribution)
- Hundreds of compute nodes, each with 12-64 processors.
- Info: `info.circ.rochester.edu`
- Uses of BlueHive
    - Large parallel problems: exploit huge number of cores
    - Work with large datasets: exploit huge RAM
    - Expensive specialized software unavailable on lab computers

Overview

- BlueHive is a university cluster running RHEL (a Linux distribution)
- Hundreds of compute nodes, each with 12-64 processors.
- Info: `info.circ.rochester.edu`
- Uses of BlueHive
    - Large parallel problems: exploit huge number of cores
    - Work with large datasets: exploit huge RAM
    - Expensive specialized software unavailable on lab computers

Accessing BlueHive

- Access through `bluehive.circ.rochester.edu`
    - if off campus, have to connect through the university VPN
- Launch session with desired parameters:

  ```
  mate-session --time=8:00:00 --cpus-per-task=16 --mem=32g
  ```
- Wait for the session to appear. Wait time depends on queue, faster if ask for fewer resources.
- Session opens in a virtual desktop. The desktop environment is MATE.
- Pop-up bar in the top: toggle fullscreen, exchange text between your computer's clipboard and the virtual desktop's clipboard.

Accessing BlueHive

- Access through `bluehive.circ.rochester.edu`
  - if off campus, have to connect through the university VPN
- Launch session with desired parameters:
  `mate-session --time=8:00:00 --cpus-per-task=16 --mem=32g`
- Wait for the session to appear. Wait time depends on queue, faster if ask for fewer resources.
- Session opens in a virtual desktop. The desktop environment is MATE.
- Pop-up bar in the top: toggle fullscreen, exchange text between your computer's clipboard and the virtual desktop's clipboard.

Accessing BlueHive

- Access through `bluehive.circ.rochester.edu`
    - if off campus, have to connect through the university VPN
- Launch session with desired parameters:

  ```
  mate-session --time=8:00:00 --cpus-per-task=16 --mem=32g
  ```
- Wait for the session to appear. Wait time depends on queue, faster if ask for fewer resources.
- Session opens in a virtual desktop. The desktop environment is MATE.
- Pop-up bar in the top: toggle fullscreen, exchange text between your computer's clipboard and the virtual desktop's clipboard.

Accessing BlueHive

- Access through `bluehive.circ.rochester.edu`
  - if off campus, have to connect through the university VPN
- Launch session with desired parameters:

  ```
  mate-session --time=8:00:00 --cpus-per-task=16 --mem=32g
  ```
- Wait for the session to appear. Wait time depends on queue, faster if ask for fewer resources.
- Session opens in a virtual desktop. The desktop environment is MATE.
- Pop-up bar in the top: toggle fullscreen, exchange text between your computer's clipboard and the virtual desktop's clipboard.

Accessing BlueHive

- Access through `bluehive.circ.rochester.edu`
  - if off campus, have to connect through the university VPN
- Launch session with desired parameters:
  `mate-session --time=8:00:00 --cpus-per-task=16 --mem=32g`
- Wait for the session to appear. Wait time depends on queue, faster if ask for fewer resources.
- Session opens in a virtual desktop. The desktop environment is MATE.
- Pop-up bar in the top: toggle fullscreen, exchange text between your computer's clipboard and the virtual desktop's clipboard.

SOFTWARE

- BH has an extensive list of free and paid software pre-installed (everything that lab computers have, and more): see `info.circ.rochester.edu/#BlueHive/Software/` for the list.
- Some programs available in Applications menu (e.g. MATLAB, Stata)
- If a program isn't there, need to activate it through the terminal
  - see the software list to find the appropriate package name for the program you want
  - open terminal through the applications menu or a top panel shortcut
  - type `module load package_name` and press Enter

Software

- BH has an extensive list of free and paid software pre-installed (everything that lab computers have, and more): see `info.circ.rochester.edu/#BlueHive/Software/` for the list.
- Some programs available in Applications menu (e.g. MATLAB, Stata)
- If a program isn't there, need to activate it through the terminal
  - see the software list to find the appropriate package name for the program you want
  - open terminal through the applications menu or a top panel shortcut
  - type `module load package_name` and press Enter

Software

- BH has an extensive list of free and paid software pre-installed (everything that lab computers have, and more): see `info.circ.rochester.edu/#BlueHive/Software/` for the list.
- Some programs available in Applications menu (e.g. MATLAB, Stata)
- If a program isn't there, need to activate it through the terminal
  - see the software list to find the appropriate package name for the program you want
  - open terminal through the applications menu or a top panel shortcut
  - type `module load package_name` and press Enter

Software

- BH has an extensive list of free and paid software pre-installed (everything that lab computers have, and more): see `info.circ.rochester.edu/#BlueHive/Software/` for the list.
- Some programs available in Applications menu (e.g. MATLAB, Stata)
- If a program isn't there, need to activate it through the terminal
  - see the software list to find the appropriate package name for the program you want
  - open terminal through the applications menu or a top panel shortcut
  - type `module load package_name` and press Enter

Software

- BH has an extensive list of free and paid software pre-installed (everything that lab computers have, and more): see `info.circ.rochester.edu/#BlueHive/Software/` for the list.
- Some programs available in Applications menu (e.g. MATLAB, Stata)
- If a program isn't there, need to activate it through the terminal
  - see the software list to find the appropriate package name for the program you want
  - open terminal through the applications menu or a top panel shortcut
  - type `module load package_name` and press Enter

SOFTWARE

- BH has an extensive list of free and paid software pre-installed (everything that lab computers have, and more): see `info.circ.rochester.edu/#BlueHive/Software/` for the list.
- Some programs available in Applications menu (e.g. MATLAB, Stata)
- If a program isn't there, need to activate it through the terminal
    - see the software list to find the appropriate package name for the program you want
    - open terminal through the applications menu or a top panel shortcut
    - type `module load package_name` and press Enter

Software: Julia

- Running Julia shell
  - `module load julia`
  - `export JULIA_NUM_THREADS=16` (or however many CPUs you requested)
  - `julia`
  - you are now in the Julia REPL
- Running a .jl file
  - load module, set threads
  - `julia ''path/to/your/file.jl''`
- Using Atom as an IDE for Julia
  - `module load julia`
  - `module load atom`
  - `atom`
  - once Atom has launched, install the `uber-juno` package in it
  - find the path to the Julia executable by running `which julia`. Paste into the “Julia Path” setting of the `julia-client` Atom package settings. Set the number of threads manually there as well.

Software: Julia

- Running Julia shell
  - `module load julia`
  - `export JULIA_NUM_THREADS=16` (or however many CPUs you requested)
  - `julia`
  - you are now in the Julia REPL
- Running a .jl file
  - load module, set threads
  - `julia ''path/to/your/file.jl''`
- Using Atom as an IDE for Julia
  - `module load julia`
  - `module load atom`
  - `atom`
  - once Atom has launched, install the `uber-juno` package in it
  - find the path to the Julia executable by running `which julia`. Paste into the “Julia Path” setting of the `julia-client` Atom package settings. Set the number of threads manually there as well.

Software: Julia

- Running Julia shell
  - `module load julia`
  - `export JULIA_NUM_THREADS=16` (or however many CPUs you requested)
  - `julia`
  - you are now in the Julia REPL
- Running a .jl file
  - load module, set threads
  - `julia ''path/to/your/file.jl''`
- Using Atom as an IDE for Julia
  - `module load julia`
  - `module load atom`
  - `atom`
  - once Atom has launched, install the `uber-juno` package in it
  - find the path to the Julia executable by running `which julia`. Paste into the "Julia Path" setting of the `julia-client` Atom package settings. Set the number of threads manually there as well.

Software: Julia

- Running Julia shell
  - `module load julia`
  - `export JULIA_NUM_THREADS=16` (or however many CPUs you requested)
  - `julia`
  - you are now in the Julia REPL
- Running a .jl file
  - load module, set threads
  - `julia ''path/to/your/file.jl''`
- Using Atom as an IDE for Julia
  - `module load julia`
  - `module load atom`
  - `atom`
  - once Atom has launched, install the `uber-juno` package in it
  - find the path to the Julia executable by running `which julia`. Paste into the "Julia Path" setting of the `julia-client` Atom package settings. Set the number of threads manually there as well.

Software: Julia

- Running Julia shell
  - `module load julia`
  - `export JULIA_NUM_THREADS=16` (or however many CPUs you requested)
  - `julia`
  - you are now in the Julia REPL
- Running a .jl file
  - load module, set threads
  - `julia ''path/to/your/file.jl''`
- Using Atom as an IDE for Julia
  - `module load julia`
  - `module load atom`
  - `atom`
  - once Atom has launched, install the `uber-juno` package in it
  - find the path to the Julia executable by running `which julia`. Paste into the "Julia Path" setting of the `julia-client` Atom package settings. Set the number of threads manually there as well.

Software: Julia

- Running Julia shell
  - `module load julia`
  - `export JULIA_NUM_THREADS=16` (or however many CPUs you requested)
  - `julia`
  - you are now in the Julia REPL
- Running a .jl file
  - load module, set threads
  - `julia ''path/to/your/file.jl''`
- Using Atom as an IDE for Julia
  - `module load julia`
  - `module load atom`
  - `atom`
  - once Atom has launched, install the `uber-juno` package in it
  - find the path to the Julia executable by running `which julia`. Paste into the "Julia Path" setting of the `julia-client` Atom package settings. Set the number of threads manually there as well.

Software: Julia

- Running Julia shell
    - `module load julia`
    - `export JULIA_NUM_THREADS=16` (or however many CPUs you requested)
    - `julia`
    - you are now in the Julia REPL
- Running a .jl file
    - load module, set threads
    - `julia ''path/to/your/file.jl''`
- Using Atom as an IDE for Julia
    - `module load julia`
    - `module load atom`
    - `atom`
    - once Atom has launched, install the `uber-juno` package in it
    - find the path to the Julia executable by running `which julia`. Paste into the "Julia Path" setting of the `julia-client` Atom package settings. Set the number of threads manually there as well.

Software: Julia

- Running Julia shell
  - `module load julia`
  - `export JULIA_NUM_THREADS=16` (or however many CPUs you requested)
  - `julia`
  - you are now in the Julia REPL
- Running a .jl file
  - load module, set threads
  - `julia ''path/to/your/file.jl''`
- Using Atom as an IDE for Julia
  - `module load julia`
  - `module load atom`
  - `atom`
  - once Atom has launched, install the `uber-juno` package in it
  - find the path to the Julia executable by running `which julia`. Paste into the “Julia Path” setting of the `julia-client` Atom package settings. Set the number of threads manually there as well.

Software: Julia

- Running Julia shell
  - `module load julia`
  - `export JULIA_NUM_THREADS=16` (or however many CPUs you requested)
  - `julia`
  - you are now in the Julia REPL
- Running a .jl file
  - load module, set threads
  - `julia ''path/to/your/file.jl''`
- Using Atom as an IDE for Julia
  - `module load julia`
  - `module load atom`
  - `atom`
  - once Atom has launched, install the `uber-juno` package in it
  - find the path to the Julia executable by running `which julia`. Paste into the "Julia Path" setting of the `julia-client` Atom package settings. Set the number of threads manually there as well.

Software: Julia

- Running Julia shell
  - `module load julia`
  - `export JULIA_NUM_THREADS=16` (or however many CPUs you requested)
  - `julia`
  - you are now in the Julia REPL
- Running a .jl file
  - load module, set threads
  - `julia ''path/to/your/file.jl''`
- Using Atom as an IDE for Julia
  - `module load julia`
  - `module load atom`
  - `atom`
  - once Atom has launched, install the `uber-juno` package in it
  - find the path to the Julia executable by running `which julia`. Paste into the "Julia Path" setting of the `julia-client` Atom package settings. Set the number of threads manually there as well.

Software: Julia

- Running Julia shell
  - `module load julia`
  - `export JULIA_NUM_THREADS=16` (or however many CPUs you requested)
  - `julia`
  - you are now in the Julia REPL
- Running a .jl file
  - load module, set threads
  - `julia ''path/to/your/file.jl''`
- Using Atom as an IDE for Julia
  - `module load julia`
  - `module load atom`
  - `atom`
  - once Atom has launched, install the `uber-juno` package in it
  - find the path to the Julia executable by running `which julia`. Paste into the "Julia Path" setting of the `julia-client` Atom package settings. Set the number of threads manually there as well.

Software: Julia

- Running Julia shell
  - `module load julia`
  - `export JULIA_NUM_THREADS=16` (or however many CPUs you requested)
  - `julia`
  - you are now in the Julia REPL
- Running a .jl file
  - load module, set threads
  - `julia ''path/to/your/file.jl''`
- Using Atom as an IDE for Julia
  - `module load julia`
  - `module load atom`
  - `atom`
  - once Atom has launched, install the `uber-juno` package in it
  - find the path to the Julia executable by running `which julia`. Paste into the "Julia Path" setting of the `julia-client` Atom package settings. Set the number of threads manually there as well.

Software: Julia

- Running Julia shell
  - `module load julia`
  - `export JULIA_NUM_THREADS=16` (or however many CPUs you requested)
  - `julia`
  - you are now in the Julia REPL
- Running a .jl file
  - load module, set threads
  - `julia ''path/to/your/file.jl''`
- Using Atom as an IDE for Julia
  - `module load julia`
  - `module load atom`
  - `atom`
  - once Atom has launched, install the `uber-juno` package in it
  - find the path to the Julia executable by running `which julia`. Paste into the “Julia Path” setting of the `julia-client` Atom package settings. Set the number of threads manually there as well.

Software: Julia

- Running Julia shell
  - `module load julia`
  - `export JULIA_NUM_THREADS=16` (or however many CPUs you requested)
  - `julia`
  - you are now in the Julia REPL
- Running a .jl file
  - load module, set threads
  - `julia ''path/to/your/file.jl''`
- Using Atom as an IDE for Julia
  - `module load julia`
  - `module load atom`
  - `atom`
  - once Atom has launched, install the `uber-juno` package in it
  - find the path to the Julia executable by running `which julia`. Paste into the "Julia Path" setting of the `julia-client` Atom package settings. Set the number of threads manually there as well.

Files and Jobs

- Quick and dirty way to transfer files: upload to some cloud service, open its web app in Firefox on the virtual desktop, download.
- For more efficiency and speed, use FTP. See the info page.
- Jobs: don't need to run code through GUI
  - can set up a script asking for certain resources and executing a certain command (e.g. `julia file.jl`)
  - most software packages we use (MATLAB, Julia, Stata, R) can run scripts (.m, .jl, .do, .R) from the command line
  - the job enters a queue and is executed once resources are available
  - preferable for more demanding programs (don't need to wait around for session to open)
  - more efficient when want to run many scripts simultaneously
  - see the info page

Files and Jobs

- Quick and dirty way to transfer files: upload to some cloud service, open its web app in Firefox on the virtual desktop, download.
- For more efficiency and speed, use FTP. See the info page.
- Jobs: don't need to run code through GUI
  - can set up a script asking for certain resources and executing a certain command (e.g. `julia file.jl`)
  - most software packages we use (MATLAB, Julia, Stata, R) can run scripts (.m, .jl, .do, .R) from the command line
  - the job enters a queue and is executed once resources are available
  - preferable for more demanding programs (don't need to wait around for session to open)
  - more efficient when want to run many scripts simultaneously
  - see the info page

Files and Jobs

- Quick and dirty way to transfer files: upload to some cloud service, open its web app in Firefox on the virtual desktop, download.
- For more efficiency and speed, use FTP. See the info page.
- Jobs: don't need to run code through GUI
    - can set up a script asking for certain resources and executing a certain command (e.g. `julia file.jl`)
    - most software packages we use (MATLAB, Julia, Stata, R) can run scripts (.m, .jl, .do, .R) from the command line
    - the job enters a queue and is executed once resources are available
    - preferable for more demanding programs (don't need to wait around for session to open)
    - more efficient when want to run many scripts simultaneously
    - see the info page

Files and Jobs

- Quick and dirty way to transfer files: upload to some cloud service, open its web app in Firefox on the virtual desktop, download.
- For more efficiency and speed, use FTP. See the info page.
- Jobs: don't need to run code through GUI
  - can set up a script asking for certain resources and executing a certain command (e.g. `julia file.jl`)
  - most software packages we use (MATLAB, Julia, Stata, R) can run scripts (.m, .jl, .do, .R) from the command line
  - the job enters a queue and is executed once resources are available
  - preferable for more demanding programs (don't need to wait around for session to open)
  - more efficient when want to run many scripts simultaneously
  - see the info page

Files and Jobs

- Quick and dirty way to transfer files: upload to some cloud service, open its web app in Firefox on the virtual desktop, download.
- For more efficiency and speed, use FTP. See the info page.
- Jobs: don't need to run code through GUI
  - can set up a script asking for certain resources and executing a certain command (e.g. `julia file.jl`)
  - most software packages we use (MATLAB, Julia, Stata, R) can run scripts (.m, .jl, .do, .R) from the command line
  - the job enters a queue and is executed once resources are available
  - preferable for more demanding programs (don't need to wait around for session to open)
  - more efficient when want to run many scripts simultaneously
  - see the info page

Files and Jobs

- Quick and dirty way to transfer files: upload to some cloud service, open its web app in Firefox on the virtual desktop, download.
- For more efficiency and speed, use FTP. See the info page.
- Jobs: don't need to run code through GUI
  - can set up a script asking for certain resources and executing a certain command (e.g. `julia file.jl`)
  - most software packages we use (MATLAB, Julia, Stata, R) can run scripts (.m, .jl, .do, .R) from the command line
  - the job enters a queue and is executed once resources are available
  - preferable for more demanding programs (don't need to wait around for session to open)
  - more efficient when want to run many scripts simultaneously
  - see the info page

Files and Jobs

- Quick and dirty way to transfer files: upload to some cloud service, open its web app in Firefox on the virtual desktop, download.
- For more efficiency and speed, use FTP. See the info page.
- Jobs: don't need to run code through GUI
  - can set up a script asking for certain resources and executing a certain command (e.g. `julia file.jl`)
  - most software packages we use (MATLAB, Julia, Stata, R) can run scripts (.m, .jl, .do, .R) from the command line
  - the job enters a queue and is executed once resources are available
  - preferable for more demanding programs (don't need to wait around for session to open)
  - more efficient when want to run many scripts simultaneously
  - see the info page

Files and Jobs

- Quick and dirty way to transfer files: upload to some cloud service, open its web app in Firefox on the virtual desktop, download.
- For more efficiency and speed, use FTP. See the info page.
- Jobs: don't need to run code through GUI
  - can set up a script asking for certain resources and executing a certain command (e.g. `julia file.jl`)
  - most software packages we use (MATLAB, Julia, Stata, R) can run scripts (.m, .jl, .do, .R) from the command line
  - the job enters a queue and is executed once resources are available
  - preferable for more demanding programs (don't need to wait around for session to open)
  - more efficient when want to run many scripts simultaneously
  - see the info page

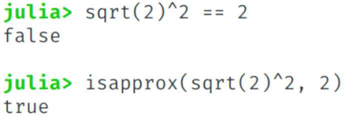
Files and Jobs

- Quick and dirty way to transfer files: upload to some cloud service, open its web app in Firefox on the virtual desktop, download.
- For more efficiency and speed, use FTP. See the info page.
- Jobs: don't need to run code through GUI
  - can set up a script asking for certain resources and executing a certain command (e.g. `julia file.jl`)
  - most software packages we use (MATLAB, Julia, Stata, R) can run scripts (.m, .jl, .do, .R) from the command line
  - the job enters a queue and is executed once resources are available
  - preferable for more demanding programs (don't need to wait around for session to open)
  - more efficient when want to run many scripts simultaneously
  - see the info page

General Coding

Precision

- Computers store approximations of real numbers in floating point type.
- Default: double precision (64 bits).
- In Julia: can go down to 16/32, can go up to 128 with packages, can do arbitrary precision.
- Can declare all variables as 32-bit types instead of 64-bit
  - RAM consumption ↓ almost 50%, speed ↑
  - (usually) negligible precision loss
- Machine epsilon for double: $2^{-52} = 2.2 \times 10^{-16}$.
- Results of intermediate computations are approximated:

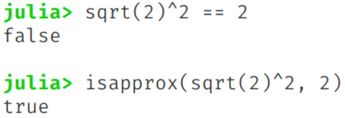


```
julia> sqrt(2)^2 == 2
false

julia> isapprox(sqrt(2)^2, 2)
true
```

Precision

- Computers store approximations of real numbers in floating point type.
- Default: double precision (64 bits).
- In Julia: can go down to 16/32, can go up to 128 with packages, can do arbitrary precision.
- Can declare all variables as 32-bit types instead of 64-bit
  - RAM consumption ↓ almost 50%, speed ↑
  - (usually) negligible precision loss
- Machine epsilon for double: $2^{-52} = 2.2 \times 10^{-16}$.
- Results of intermediate computations are approximated:

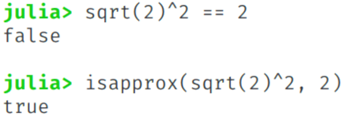


```
julia> sqrt(2)^2 == 2
false

julia> isapprox(sqrt(2)^2, 2)
true
```

Precision

- Computers store approximations of real numbers in floating point type.
- Default: double precision (64 bits).
- In Julia: can go down to 16/32, can go up to 128 with packages, can do arbitrary precision.
- Can declare all variables as 32-bit types instead of 64-bit
  - RAM consumption ↓ almost 50%, speed ↑
  - (usually) negligible precision loss
- Machine epsilon for double: $2^{-52} = 2.2 \times 10^{-16}$.
- Results of intermediate computations are approximated:

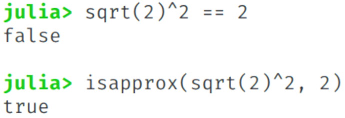


```
julia> sqrt(2)^2 == 2
false

julia> isapprox(sqrt(2)^2, 2)
true
```

Precision

- Computers store approximations of real numbers in floating point type.
- Default: double precision (64 bits).
- In Julia: can go down to 16/32, can go up to 128 with packages, can do arbitrary precision.
- Can declare all variables as 32-bit types instead of 64-bit
  - RAM consumption ↓ almost 50%, speed ↑
  - (usually) negligible precision loss
- Machine epsilon for double: $2^{-52} = 2.2 \times 10^{-16}$.
- Results of intermediate computations are approximated:

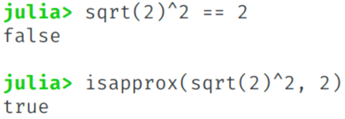


```
julia> sqrt(2)^2 == 2
false

julia> isapprox(sqrt(2)^2, 2)
true
```

Precision

- Computers store approximations of real numbers in floating point type.
- Default: double precision (64 bits).
- In Julia: can go down to 16/32, can go up to 128 with packages, can do arbitrary precision.
- Can declare all variables as 32-bit types instead of 64-bit
    - RAM consumption ↓ almost 50%, speed ↑
    - (usually) negligible precision loss
- Machine epsilon for double: $2^{-52} = 2.2 \times 10^{-16}$.
- Results of intermediate computations are approximated:

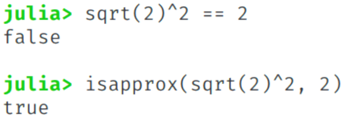


```
julia> sqrt(2)^2 == 2
false

julia> isapprox(sqrt(2)^2, 2)
true
```

Precision

- Computers store approximations of real numbers in floating point type.
- Default: double precision (64 bits).
- In Julia: can go down to 16/32, can go up to 128 with packages, can do arbitrary precision.
- Can declare all variables as 32-bit types instead of 64-bit
    - RAM consumption ↓ almost 50%, speed ↑
    - (usually) negligible precision loss
- Machine epsilon for double: $2^{-52} = 2.2 \times 10^{-16}$.
- Results of intermediate computations are approximated:

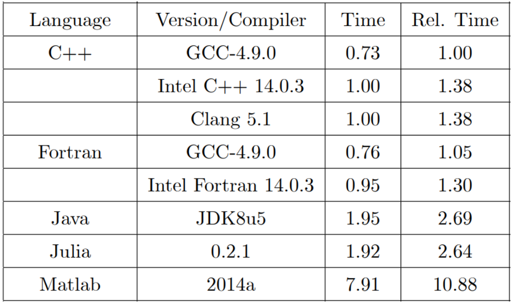


```
julia> sqrt(2)^2 == 2
false

julia> isapprox(sqrt(2)^2, 2)
true
```

Comparison of Languages

- Aruoba and Fernández-Villaverde (2014), “A Comparison of Programming Languages in Economics”:

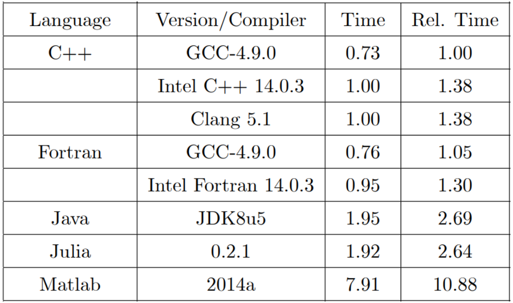


Language	Version/Compiler	Time	Rel. Time
C++	GCC-4.9.0	0.73	1.00
	Intel C++ 14.0.3	1.00	1.38
	Clang 5.1	1.00	1.38
Fortran	GCC-4.9.0	0.76	1.05
	Intel Fortran 14.0.3	0.95	1.30
Java	JDK8u5	1.95	2.69
Julia	0.2.1	1.92	2.64
Matlab	2014a	7.91	10.88

- Running time usually less important than development time.
- MATLAB and Julia run slower, but much faster to code and debug.
- Julia considerably faster than MATLAB

Comparison of Languages

- Aruoba and Fernández-Villaverde (2014), "A Comparison of Programming Languages in Economics":

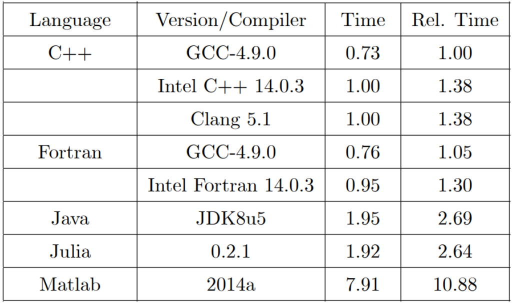


Language	Version/Compiler	Time	Rel. Time
C++	GCC-4.9.0	0.73	1.00
	Intel C++ 14.0.3	1.00	1.38
	Clang 5.1	1.00	1.38
Fortran	GCC-4.9.0	0.76	1.05
	Intel Fortran 14.0.3	0.95	1.30
Java	JDK8u5	1.95	2.69
Julia	0.2.1	1.92	2.64
Matlab	2014a	7.91	10.88

- Running time usually less important than development time.
- MATLAB and Julia run slower, but much faster to code and debug.
- Julia considerably faster than MATLAB

Comparison of Languages

- Aruoba and Fernández-Villaverde (2014), "A Comparison of Programming Languages in Economics":

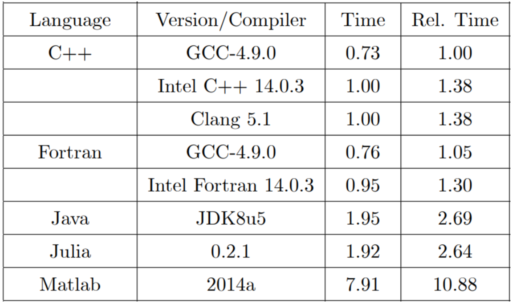


Language	Version/Compiler	Time	Rel. Time
C++	GCC-4.9.0	0.73	1.00
	Intel C++ 14.0.3	1.00	1.38
	Clang 5.1	1.00	1.38
Fortran	GCC-4.9.0	0.76	1.05
	Intel Fortran 14.0.3	0.95	1.30
Java	JDK8u5	1.95	2.69
Julia	0.2.1	1.92	2.64
Matlab	2014a	7.91	10.88

- Running time usually less important than development time.
- MATLAB and Julia run slower, but much faster to code and debug.
- Julia considerably faster than MATLAB

Comparison of Languages

- Aruoba and Fernández-Villaverde (2014), "A Comparison of Programming Languages in Economics":



Language	Version/Compiler	Time	Rel. Time
C++	GCC-4.9.0	0.73	1.00
	Intel C++ 14.0.3	1.00	1.38
	Clang 5.1	1.00	1.38
Fortran	GCC-4.9.0	0.76	1.05
	Intel Fortran 14.0.3	0.95	1.30
Java	JDK8u5	1.95	2.69
Julia	0.2.1	1.92	2.64
Matlab	2014a	7.91	10.88

- Running time usually less important than development time.
- MATLAB and Julia run slower, but much faster to code and debug.
- Julia considerably faster than MATLAB

Profiling

- Time and profile your code
  - Julia: `@time some_fn()` to measure time and memory allocations
  - Julia: Profile module to show time spent on each line (`@profile` and `@profiler`)
- Find bottlenecks, figure out how to optimize them

Profiling

- Time and profile your code
    - Julia: `@time some_fn()` to measure time and memory allocations
    - Julia: Profile module to show time spent on each line (`@profile` and `@profiler`)
- Find bottlenecks, figure out how to optimize them

Git

- Version control: revert to *any* version of the code
  - backups and file copies are unreliable, inconvenient, and inflexible
  - a file name from my 2nd year project: `model2_4parallelVer2OptimizedMore.jl`
  - instead, use a Version Control System
- Git: most popular VCS
  - change a file ⟹ *commit* the change with a descriptive name
  - Git will store this commit forever
  - can easily see differences between commits, revert to any commit
  - each commit takes up little space: only the difference from previous commit needs to be stored
- Collaboration
  - can work in parallel (*branches*), easily merge commits
  - host code on GitHub or GitLab

Git

- Version control: revert to *any* version of the code
  - backups and file copies are unreliable, inconvenient, and inflexible
  - a file name from my 2nd year project: `model2_4parallelVer2OptimizedMore.jl`
  - instead, use a Version Control System
- Git: most popular VCS
  - change a file ⟹ *commit* the change with a descriptive name
  - Git will store this commit forever
  - can easily see differences between commits, revert to any commit
  - each commit takes up little space: only the difference from previous commit needs to be stored
- Collaboration
  - can work in parallel (*branches*), easily merge commits
  - host code on GitHub or GitLab

Git

- Version control: revert to *any* version of the code
  - backups and file copies are unreliable, inconvenient, and inflexible
  - a file name from my 2nd year project: `model2_4parallelVer2OptimizedMore.jl`
  - instead, use a Version Control System
- Git: most popular VCS
  - change a file ⟹ *commit* the change with a descriptive name
  - Git will store this commit forever
  - can easily see differences between commits, revert to any commit
  - each commit takes up little space: only the difference from previous commit needs to be stored
- Collaboration
  - can work in parallel (*branches*), easily merge commits
  - host code on GitHub or GitLab

Git

- Version control: revert to *any* version of the code
  - backups and file copies are unreliable, inconvenient, and inflexible
  - a file name from my 2nd year project: `model2_4parallelVer2OptimizedMore.jl`
  - instead, use a Version Control System
- Git: most popular VCS
  - change a file ⟹ *commit* the change with a descriptive name
  - Git will store this commit forever
  - can easily see differences between commits, revert to any commit
  - each commit takes up little space: only the difference from previous commit needs to be stored
- Collaboration
  - can work in parallel (*branches*), easily merge commits
  - host code on GitHub or GitLab

Git

- Version control: revert to *any* version of the code
  - backups and file copies are unreliable, inconvenient, and inflexible
  - a file name from my 2nd year project: `model2_4parallelVer2OptimizedMore.jl`
  - instead, use a Version Control System
- Git: most popular VCS
  - change a file ⟹ *commit* the change with a descriptive name
  - Git will store this commit forever
  - can easily see differences between commits, revert to any commit
  - each commit takes up little space: only the difference from previous commit needs to be stored
- Collaboration
  - can work in parallel (*branches*), easily merge commits
  - host code on GitHub or GitLab

Git

- Version control: revert to *any* version of the code
  - backups and file copies are unreliable, inconvenient, and inflexible
  - a file name from my 2nd year project: `model2_4parallelVer2OptimizedMore.jl`
  - instead, use a Version Control System
- Git: most popular VCS
  - change a file ⟹ *commit* the change with a descriptive name
  - Git will store this commit forever
  - can easily see differences between commits, revert to any commit
  - each commit takes up little space: only the difference from previous commit needs to be stored
- Collaboration
  - can work in parallel (*branches*), easily merge commits
  - host code on GitHub or GitLab

Git

- Version control: revert to *any* version of the code
    - backups and file copies are unreliable, inconvenient, and inflexible
    - a file name from my 2nd year project: `model2_4parallelVer2OptimizedMore.jl`
    - instead, use a Version Control System
- Git: most popular VCS
    - change a file $\implies$ *commit* the change with a descriptive name
    - Git will store this commit forever
    - can easily see differences between commits, revert to any commit
    - each commit takes up little space: only the difference from previous commit needs to be stored
- Collaboration
    - can work in parallel (*branches*), easily merge commits
    - host code on GitHub or GitLab

Git

- Version control: revert to *any* version of the code
  - backups and file copies are unreliable, inconvenient, and inflexible
  - a file name from my 2nd year project: `model2_4parallelVer2OptimizedMore.jl`
  - instead, use a Version Control System
- Git: most popular VCS
  - change a file ⟹ *commit* the change with a descriptive name
  - Git will store this commit forever
  - can easily see differences between commits, revert to any commit
  - each commit takes up little space: only the difference from previous commit needs to be stored
- Collaboration
  - can work in parallel (*branches*), easily merge commits
  - host code on GitHub or GitLab

Git

- Version control: revert to *any* version of the code
    - backups and file copies are unreliable, inconvenient, and inflexible
    - a file name from my 2nd year project: `model2_4parallelVer2OptimizedMore.jl`
    - instead, use a Version Control System
- Git: most popular VCS
    - change a file $\implies$ *commit* the change with a descriptive name
    - Git will store this commit forever
    - can easily see differences between commits, revert to any commit
    - each commit takes up little space: only the difference from previous commit needs to be stored
- Collaboration
    - can work in parallel (*branches*), easily merge commits
    - host code on GitHub or GitLab

Git

- Version control: revert to *any* version of the code
  - backups and file copies are unreliable, inconvenient, and inflexible
  - a file name from my 2nd year project: `model2_4parallelVer2OptimizedMore.jl`
  - instead, use a Version Control System
- Git: most popular VCS
  - change a file $\implies$ *commit* the change with a descriptive name
  - Git will store this commit forever
  - can easily see differences between commits, revert to any commit
  - each commit takes up little space: only the difference from previous commit needs to be stored
- Collaboration
  - can work in parallel (*branches*), easily merge commits
  - host code on GitHub or GitLab

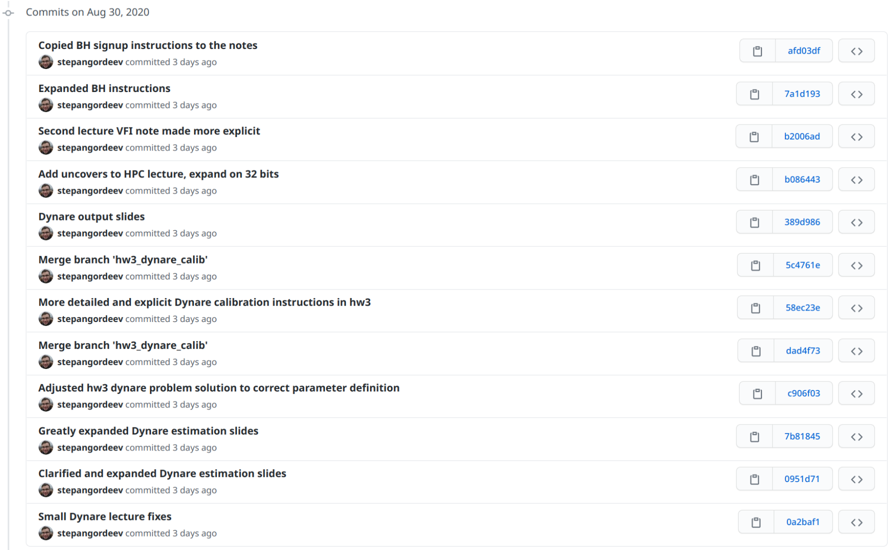
Git

- Version control: revert to *any* version of the code
    - backups and file copies are unreliable, inconvenient, and inflexible
    - a file name from my 2nd year project: `model2_4parallelVer2OptimizedMore.jl`
    - instead, use a Version Control System
- Git: most popular VCS
    - change a file $\Longrightarrow$ *commit* the change with a descriptive name
    - Git will store this commit forever
    - can easily see differences between commits, revert to any commit
    - each commit takes up little space: only the difference from previous commit needs to be stored
- Collaboration
    - can work in parallel (*branches*), easily merge commits
    - host code on GitHub or GitLab

Git

- Version control: revert to *any* version of the code
    - backups and file copies are unreliable, inconvenient, and inflexible
    - a file name from my 2nd year project: `model2_4parallelVer2OptimizedMore.jl`
    - instead, use a Version Control System
- Git: most popular VCS
    - change a file $\Longrightarrow$ *commit* the change with a descriptive name
    - Git will store this commit forever
    - can easily see differences between commits, revert to any commit
    - each commit takes up little space: only the difference from previous commit needs to be stored
- Collaboration
    - can work in parallel (*branches*), easily merge commits
    - host code on GitHub or GitLab

Git Example: This Course



Commits on Aug 30, 2020

Copied BH signup instructions to the notes
stepangordeev committed 3 days ago — afd03df

Expanded BH instructions
stepangordeev committed 3 days ago — 7a1d193

Second lecture VFI note made more explicit
stepangordeev committed 3 days ago — b2006ad

Add uncovers to HPC lecture, expand on 32 bits
stepangordeev committed 3 days ago — b086443

Dynare output slides
stepangordeev committed 3 days ago — 389d986

Merge branch 'hw3_dynare_calib'
stepangordeev committed 3 days ago — 5c4761e

More detailed and explicit Dynare calibration instructions in hw3
stepangordeev committed 3 days ago — 58ec23e

Merge branch 'hw3_dynare_calib'
stepangordeev committed 3 days ago — dad4f73

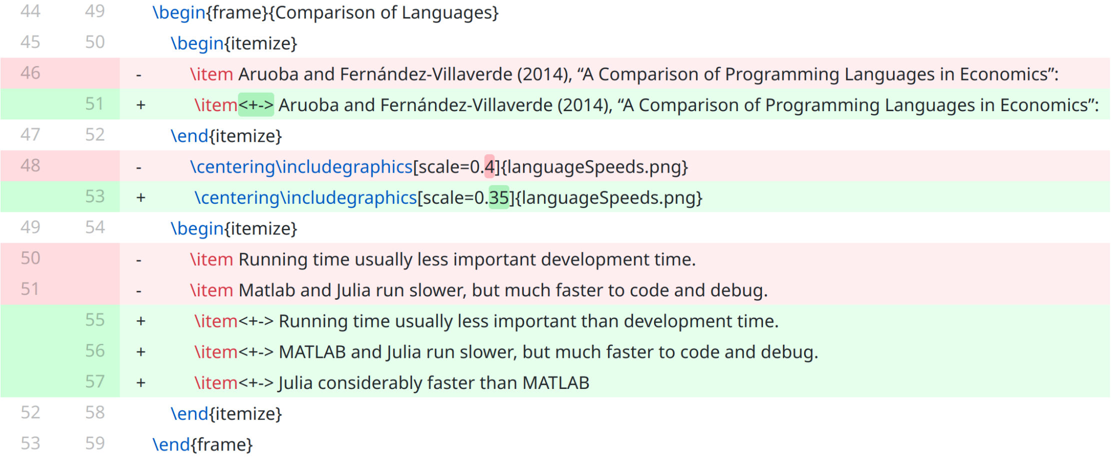
Adjusted hw3 dynare problem solution to correct parameter definition
stepangordeev committed 3 days ago — c906f03

Greatly expanded Dynare estimation slides
stepangordeev committed 3 days ago — 7b81845

Clarified and expanded Dynare estimation slides
stepangordeev committed 3 days ago — 0951d71

Small Dynare lecture fixes
stepangordeev committed 3 days ago — 0a2baf1

Git Example: This Course



```diff
   \begin{frame}{Comparison of Languages}
     \begin{itemize}
-        \item Aruoba and Fernández-Villaverde (2014), “A Comparison of Programming Languages in Economics”:
+        \item<+-> Aruoba and Fernández-Villaverde (2014), “A Comparison of Programming Languages in Economics”:
     \end{itemize}
-        \centering\includegraphics[scale=0.4]{languageSpeeds.png}
+        \centering\includegraphics[scale=0.35]{languageSpeeds.png}
     \begin{itemize}
-        \item Running time usually less important development time.
-        \item Matlab and Julia run slower, but much faster to code and debug.
+        \item<+-> Running time usually less important than development time.
+        \item<+-> MATLAB and Julia run slower, but much faster to code and debug.
+        \item<+-> Julia considerably faster than MATLAB
     \end{itemize}
   \end{frame}
```

Git Resources

- Pro Git: free comprehensive guide
- Git is a command line tool, but many GUIs available
    - e.g. GitKraken: normally paid, but free with GitHub Student account (can get with UR email)

Git Resources

- Pro Git: free comprehensive guide
- Git is a command line tool, but many GUIs available
    - e.g. GitKraken: normally paid, but free with GitHub Student account (can get with UR email)

Git Resources

- Pro Git: free comprehensive guide
- Git is a command line tool, but many GUIs available
    - e.g. GitKraken: normally paid, but free with GitHub Student account (can get with UR email)

General Advice

- Maintainable code
  - write your code so that it could be read by anyone: then your future self will definitely be able to read it!
  - self-explanatory names
  - ample comments
- Reproducible code
  - make sure your results can be obtained by running a single file on any computer.
  - make sure you can reproduce results obtained in the past (VCS!)
- Acquire technical skills and learn tools now—won't have time after the PhD!

General Advice

- Maintainable code
  - write your code so that it could be read by anyone: then your future self will definitely be able to read it!
  - self-explanatory names
  - ample comments
- Reproducible code
  - make sure your results can be obtained by running a single file on any computer.
  - make sure you can reproduce results obtained in the past (VCS!)
- Acquire technical skills and learn tools now—won't have time after the PhD!

General Advice

- Maintainable code
  - write your code so that it could be read by anyone: then your future self will definitely be able to read it!
  - self-explanatory names
  - ample comments
- Reproducible code
  - make sure your results can be obtained by running a single file on any computer.
  - make sure you can reproduce results obtained in the past (VCS!)
- Acquire technical skills and learn tools now—won't have time after the PhD!

General Advice

- Maintainable code
    - write your code so that it could be read by anyone: then your future self will definitely be able to read it!
    - self-explanatory names
    - ample comments
- Reproducible code
    - make sure your results can be obtained by running a single file on any computer.
    - make sure you can reproduce results obtained in the past (VCS!)
- Acquire technical skills and learn tools now—won't have time after the PhD!

General Advice

- Maintainable code
  - write your code so that it could be read by anyone: then your future self will definitely be able to read it!
  - self-explanatory names
  - ample comments
- Reproducible code
  - make sure your results can be obtained by running a single file on any computer.
  - make sure you can reproduce results obtained in the past (VCS!)
- Acquire technical skills and learn tools now—won't have time after the PhD!

General Advice

- Maintainable code
    - write your code so that it could be read by anyone: then your future self will definitely be able to read it!
    - self-explanatory names
    - ample comments
- Reproducible code
    - make sure your results can be obtained by running a single file on any computer.
    - make sure you can reproduce results obtained in the past (VCS!)
- Acquire technical skills and learn tools now—won't have time after the PhD!

General Advice

- Maintainable code
    - write your code so that it could be read by anyone: then your future self will definitely be able to read it!
    - self-explanatory names
    - ample comments
- Reproducible code
    - make sure your results can be obtained by running a single file on any computer.
    - make sure you can reproduce results obtained in the past (VCS!)
- Acquire technical skills and learn tools now—won't have time after the PhD!

General Advice

- Maintainable code
  - write your code so that it could be read by anyone: then your future self will definitely be able to read it!
  - self-explanatory names
  - ample comments
- Reproducible code
  - make sure your results can be obtained by running a single file on any computer.
  - make sure you can reproduce results obtained in the past (VCS!)
- Acquire technical skills and learn tools now—won't have time after the PhD!

Julia Recommendations

Explicit Types

- Explicitly state the type when declaring fields or arguments
    - field in a struct: `V::Array{Float64, 2}`, not `V`
    - `function y(x::Float64, z::Int64)`, not `function y(x, z)`
    - faster in some cases: compiler optimizes code for specified type
    - easier to catch passing a wrong variable, e.g. grid index (`Int64`) vs grid value (`Float64`)
- Preallocate arrays with explicit types
    - e.g. need to create an array element-by-element (grid, value function, etc)
    - specify array length and element type: preallocates the exact amount of memory needed
    - `policy_fn = Array{Float64}(undef, n_z, n_k)`

Explicit Types

- Explicitly state the type when declaring fields or arguments
  - field in a struct: `V::Array{Float64, 2}`, not `V`
  - `function y(x::Float64, z::Int64)`, not `function y(x, z)`
  - faster in some cases: compiler optimizes code for specified type
  - easier to catch passing a wrong variable, e.g. grid index (`Int64`) vs grid value (`Float64`)
- Preallocate arrays with explicit types
  - e.g. need to create an array element-by-element (grid, value function, etc)
  - specify array length and element type: preallocates the exact amount of memory needed
  - `policy_fn = Array{Float64}(undef, n_z, n_k)`

Explicit Types

- Explicitly state the type when declaring fields or arguments
    - field in a struct: `V::Array{Float64, 2}`, not `V`
    - `function y(x::Float64, z::Int64)`, not `function y(x, z)`
    - faster in some cases: compiler optimizes code for specified type
    - easier to catch passing a wrong variable, e.g. grid index (`Int64`) vs grid value (`Float64`)
- Preallocate arrays with explicit types
    - e.g. need to create an array element-by-element (grid, value function, etc)
    - specify array length and element type: preallocates the exact amount of memory needed
    - `policy_fn = Array{Float64}(undef, n_z, n_k)`

Explicit Types

- Explicitly state the type when declaring fields or arguments
  - field in a struct: `V::Array{Float64, 2}`, not `V`
  - `function y(x::Float64, z::Int64)`, not `function y(x, z)`
  - faster in some cases: compiler optimizes code for specified type
  - easier to catch passing a wrong variable, e.g. grid index (`Int64`) vs grid value (`Float64`)
- Preallocate arrays with explicit types
  - e.g. need to create an array element-by-element (grid, value function, etc)
  - specify array length and element type: preallocates the exact amount of memory needed
  - `policy_fn = Array{Float64}(undef, n_z, n_k)`

Explicit Types

- Explicitly state the type when declaring fields or arguments
    - field in a struct: `V::Array{Float64, 2}`, not `V`
    - `function y(x::Float64, z::Int64)`, not `function y(x, z)`
    - faster in some cases: compiler optimizes code for specified type
    - easier to catch passing a wrong variable, e.g. grid index (`Int64`) vs grid value (`Float64`)
- Preallocate arrays with explicit types
    - e.g. need to create an array element-by-element (grid, value function, etc)
    - specify array length and element type: preallocates the exact amount of memory needed
    - `policy_fn = Array{Float64}(undef, n_z, n_k)`

Explicit Types

- Explicitly state the type when declaring fields or arguments
  - field in a struct: `V::Array{Float64, 2}`, not `V`
  - `function y(x::Float64, z::Int64)`, not `function y(x, z)`
  - faster in some cases: compiler optimizes code for specified type
  - easier to catch passing a wrong variable, e.g. grid index (`Int64`) vs grid value (`Float64`)
- Preallocate arrays with explicit types
  - e.g. need to create an array element-by-element (grid, value function, etc)
  - specify array length and element type: preallocates the exact amount of memory needed
  - `policy_fn = Array{Float64}(undef, n_z, n_k)`

Explicit Types

- Explicitly state the type when declaring fields or arguments
    - field in a struct: `V::Array{Float64, 2}`, not `V`
    - `function y(x::Float64, z::Int64)`, not `function y(x, z)`
    - faster in some cases: compiler optimizes code for specified type
    - easier to catch passing a wrong variable, e.g. grid index (`Int64`) vs grid value (`Float64`)
- Preallocate arrays with explicit types
    - e.g. need to create an array element-by-element (grid, value function, etc)
    - specify array length and element type: preallocates the exact amount of memory needed
    - `policy_fn = Array{Float64}(undef, n_z, n_k)`

Explicit Types

- Explicitly state the type when declaring fields or arguments
    - field in a struct: `V::Array{Float64, 2}`, not `V`
    - `function y(x::Float64, z::Int64)`, not `function y(x, z)`
    - faster in some cases: compiler optimizes code for specified type
    - easier to catch passing a wrong variable, e.g. grid index (`Int64`) vs grid value (`Float64`)
- Preallocate arrays with explicit types
    - e.g. need to create an array element-by-element (grid, value function, etc)
    - specify array length and element type: preallocates the exact amount of memory needed
    - `policy_fn = Array{Float64}(undef, n_z, n_k)`

Explicit Types

- Explicitly state the type when declaring fields or arguments
    - field in a struct: `V::Array{Float64, 2}`, not `V`
    - `function y(x::Float64, z::Int64)`, not `function y(x, z)`
    - faster in some cases: compiler optimizes code for specified type
    - easier to catch passing a wrong variable, e.g. grid index (`Int64`) vs grid value (`Float64`)
- Preallocate arrays with explicit types
    - e.g. need to create an array element-by-element (grid, value function, etc)
    - specify array length and element type: preallocates the exact amount of memory needed
    - `policy_fn = Array{Float64}(undef, n_z, n_k)`

Type Stability

- Keep types stable: variables shouldn't change types
    - `x = 1; x = 1.5` is slow
    - `x = 1.0; x = 1.5` is fast
- Return of a function shouldn't change type depending on input value
    - e.g. `some_fn(x::Float)` returns x if x>0, else 0
    - faster if it returns 0.0: compiler will optimize the function for floats

Type Stability

- Keep types stable: variables shouldn't change types
  - `x = 1; x = 1.5` is slow
  - `x = 1.0; x = 1.5` is fast
- Return of a function shouldn't change type depending on input value
  - e.g. `some_fn(x::Float)` returns x if x>0, else 0
  - faster if it returns 0.0: compiler will optimize the function for floats

Type Stability

- Keep types stable: variables shouldn't change types
  - `x = 1; x = 1.5` is slow
  - `x = 1.0; x = 1.5` is fast
- Return of a function shouldn't change type depending on input value
  - e.g. `some_fn(x::Float)` returns x if x>0, else 0
  - faster if it returns 0.0: compiler will optimize the function for floats

Type Stability

- Keep types stable: variables shouldn't change types
  - `x = 1; x = 1.5` is slow
  - `x = 1.0; x = 1.5` is fast
- Return of a function shouldn't change type depending on input value
  - e.g. `some_fn(x::Float)` returns x if x>0, else 0
  - faster if it returns 0.0: compiler will optimize the function for floats

Type Stability

- Keep types stable: variables shouldn't change types
  - `x = 1; x = 1.5` is slow
  - `x = 1.0; x = 1.5` is fast
- Return of a function shouldn't change type depending on input value
  - e.g. `some_fn(x::Float)` returns x if x>0, else 0
  - faster if it returns 0.0: compiler will optimize the function for floats

Type Stability

- Keep types stable: variables shouldn't change types
  - `x = 1; x = 1.5` is slow
  - `x = 1.0; x = 1.5` is fast
- Return of a function shouldn't change type depending on input value
  - e.g. `some_fn(x::Float)` returns x if x>0, else 0
  - faster if it returns 0.0: compiler will optimize the function for floats

Vectorization

- apply a function to each element of a vector: can write a loop or use vectorized operations
    - automatically parallelized
    - not much performance difference otherwise (although loops MUCH slower in MATLAB)
    - but vectorization often more concise and readable
- e.g. want `a * b + scalar_fn(c)` done for 10 different a, b, c
    - dot notation: append a dot after any scalar operand or function

      ```
      vec_sum = Array{Float64}(undef, 10)
      vec_sum = vec_a .* vec_b .+ scalar_fn.(vec_c)
      ```

    - `@.` macro: appends . everywhere in the line

      ```
      vec_sum = Array{Float64}(undef, 10)
      @. vec_sum = vec_a * vec_b + scalar_fn(vec_c)
      ```

Vectorization

- apply a function to each element of a vector: can write a loop or use vectorized operations
  - automatically parallelized
  - not much performance difference otherwise (although loops MUCH slower in MATLAB)
  - but vectorization often more concise and readable
- e.g. want `a * b + scalar_fn(c)` done for 10 different a, b, c
  - dot notation: append a dot after any scalar operand or function

    ```
    vec_sum = Array{Float64}(undef, 10)
    vec_sum = vec_a .* vec_b .+ scalar_fn.(vec_c)
    ```

  - `@.` macro: appends . everywhere in the line

    ```
    vec_sum = Array{Float64}(undef, 10)
    @. vec_sum = vec_a * vec_b + scalar_fn(vec_c)
    ```

Vectorization

- apply a function to each element of a vector: can write a loop or use vectorized operations
  - automatically parallelized
  - not much performance difference otherwise (although loops MUCH slower in MATLAB)
  - but vectorization often more concise and readable
- e.g. want `a * b + scalar_fn(c)` done for 10 different a, b, c
  - dot notation: append a dot after any scalar operand or function

    ```
    vec_sum = Array{Float64}(undef, 10)
    vec_sum = vec_a .* vec_b .+ scalar_fn.(vec_c)
    ```

  - `@.` macro: appends . everywhere in the line

    ```
    vec_sum = Array{Float64}(undef, 10)
    @. vec_sum = vec_a * vec_b + scalar_fn(vec_c)
    ```

Vectorization

- apply a function to each element of a vector: can write a loop or use vectorized operations
  - automatically parallelized
  - not much performance difference otherwise (although loops MUCH slower in MATLAB)
  - but vectorization often more concise and readable
- e.g. want `a * b + scalar_fn(c)` done for 10 different a, b, c
  - dot notation: append a dot after any scalar operand or function

    ```
    vec_sum = Array{Float64}(undef, 10)
    vec_sum = vec_a .* vec_b .+ scalar_fn.(vec_c)
    ```

  - `@.` macro: appends . everywhere in the line

    ```
    vec_sum = Array{Float64}(undef, 10)
    @. vec_sum = vec_a * vec_b + scalar_fn(vec_c)
    ```

Vectorization

- apply a function to each element of a vector: can write a loop or use vectorized operations
  - automatically parallelized
  - not much performance difference otherwise (although loops MUCH slower in MATLAB)
  - but vectorization often more concise and readable
- e.g. want `a * b + scalar_fn(c)` done for 10 different a, b, c
  - dot notation: append a dot after any scalar operand or function

    ```
    vec_sum = Array{Float64}(undef, 10)
    vec_sum = vec_a .* vec_b .+ scalar_fn.(vec_c)
    ```

  - `@.` macro: appends . everywhere in the line

    ```
    vec_sum = Array{Float64}(undef, 10)
    @. vec_sum = vec_a * vec_b + scalar_fn(vec_c)
    ```

Vectorization

- apply a function to each element of a vector: can write a loop or use vectorized operations
  - automatically parallelized
  - not much performance difference otherwise (although loops MUCH slower in MATLAB)
  - but vectorization often more concise and readable
- e.g. want `a * b + scalar_fn(c)` done for 10 different a, b, c
  - dot notation: append a dot after any scalar operand or function

    ```
    vec_sum = Array{Float64}(undef, 10)
    vec_sum = vec_a .* vec_b .+ scalar_fn.(vec_c)
    ```
  - `@.` macro: appends . everywhere in the line

    ```
    vec_sum = Array{Float64}(undef, 10)
    @. vec_sum = vec_a * vec_b + scalar_fn(vec_c)
    ```

Vectorization

- apply a function to each element of a vector: can write a loop or use vectorized operations
  - automatically parallelized
  - not much performance difference otherwise (although loops MUCH slower in MATLAB)
  - but vectorization often more concise and readable
- e.g. want `a * b + scalar_fn(c)` done for 10 different a, b, c
  - dot notation: append a dot after any scalar operand or function

    ```
    vec_sum = Array{Float64}(undef, 10)
    vec_sum = vec_a .* vec_b .+ scalar_fn.(vec_c)
    ```

  - `@.` macro: appends . everywhere in the line

    ```
    vec_sum = Array{Float64}(undef, 10)
    @. vec_sum = vec_a * vec_b + scalar_fn(vec_c)
    ```

Exposition of Advanced Techniques

Krussel and Smith ‘98

- Aiyagari + transition dynamics
  - response of the economy to shocks
  - agents need to forecast the LOM of asset distribution
- Krussel and Smith: approximate the distribution with a sequence of moments
- Forecast error from looking at mean (K) alone is tiny
  - $\implies$ agents need to just forecast K, not the whole distribution
  - makes solving a heterogenous agent model with aggregate shocks feasible

Krussel and Smith ‘98

- Aiyagari + transition dynamics
  - response of the economy to shocks
  - agents need to forecast the LOM of asset distribution
- Krussel and Smith: approximate the distribution with a sequence of moments
- Forecast error from looking at mean (K) alone is tiny
  - $\implies$ agents need to just forecast K, not the whole distribution
  - makes solving a heterogenous agent model with aggregate shocks feasible

Krussel and Smith ‘98

- Aiyagari + transition dynamics
  - response of the economy to shocks
  - agents need to forecast the LOM of asset distribution
- Krussel and Smith: approximate the distribution with a sequence of moments
- Forecast error from looking at mean (K) alone is tiny
  - $\implies$ agents need to just forecast K, not the whole distribution
  - makes solving a heterogenous agent model with aggregate shocks feasible

Krussel and Smith ‘98

- Aiyagari + transition dynamics
  - response of the economy to shocks
  - agents need to forecast the LOM of asset distribution
- Krussel and Smith: approximate the distribution with a sequence of moments
- Forecast error from looking at mean (K) alone is tiny
  - $\implies$ agents need to just forecast K, not the whole distribution
  - makes solving a heterogenous agent model with aggregate shocks feasible

Krussel and Smith ‘98

- Aiyagari + transition dynamics
  - response of the economy to shocks
  - agents need to forecast the LOM of asset distribution
- Krussel and Smith: approximate the distribution with a sequence of moments
- Forecast error from looking at mean (K) alone is tiny
  - $\Longrightarrow$ agents need to just forecast K, not the whole distribution
  - makes solving a heterogenous agent model with aggregate shocks feasible

Projection Methods

- To solve a model, need to find policy functions that satisfy FOCs
- Replace policy functions with parameterized approximations
  - e.g. Chebyshev polynomials
- Solve for parameters that approximately solve FOCs
  - minimize sum of squared errors over state space
- Less subject to curse of dimensionality (like log-linearization)
- Global solution (like VFI)
- Resources
  - Heer and Maußner, ch. 6
  - Fernández-Villaverde et al (2016), Solution and Estimation Methods for DSGE Models (handbook of macro chapter), section 5.

Projection Methods

- To solve a model, need to find policy functions that satisfy FOCs
- Replace policy functions with parameterized approximations
  - e.g. Chebyshev polynomials
- Solve for parameters that approximately solve FOCs
  - minimize sum of squared errors over state space
- Less subject to curse of dimensionality (like log-linearization)
- Global solution (like VFI)
- Resources
  - Heer and Maußner, ch. 6
  - Fernández-Villaverde et al (2016), Solution and Estimation Methods for DSGE Models (handbook of macro chapter), section 5.

Projection Methods

- To solve a model, need to find policy functions that satisfy FOCs
- Replace policy functions with parameterized approximations
  - e.g. Chebyshev polynomials
- Solve for parameters that approximately solve FOCs
  - minimize sum of squared errors over state space
- Less subject to curse of dimensionality (like log-linearization)
- Global solution (like VFI)
- Resources
  - Heer and Maußner, ch. 6
  - Fernández-Villaverde et al (2016), Solution and Estimation Methods for DSGE Models (handbook of macro chapter), section 5.

Projection Methods

- To solve a model, need to find policy functions that satisfy FOCs
- Replace policy functions with parameterized approximations
  - e.g. Chebyshev polynomials
- Solve for parameters that approximately solve FOCs
  - minimize sum of squared errors over state space
- Less subject to curse of dimensionality (like log-linearization)
- Global solution (like VFI)
- Resources
  - Heer and Maußner, ch. 6
  - Fernández-Villaverde et al (2016), Solution and Estimation Methods for DSGE Models (handbook of macro chapter), section 5.

Projection Methods

- To solve a model, need to find policy functions that satisfy FOCs
- Replace policy functions with parameterized approximations
    - e.g. Chebyshev polynomials
- Solve for parameters that approximately solve FOCs
    - minimize sum of squared errors over state space
- Less subject to curse of dimensionality (like log-linearization)
- Global solution (like VFI)
- Resources
    - Heer and Maußner, ch. 6
    - Fernández-Villaverde et al (2016), Solution and Estimation Methods for DSGE Models (handbook of macro chapter), section 5.

Projection Methods

- To solve a model, need to find policy functions that satisfy FOCs
- Replace policy functions with parameterized approximations
  - e.g. Chebyshev polynomials
- Solve for parameters that approximately solve FOCs
  - minimize sum of squared errors over state space
- Less subject to curse of dimensionality (like log-linearization)
- Global solution (like VFI)
- Resources
  - Heer and Maußner, ch. 6
  - Fernández-Villaverde et al (2016), Solution and Estimation Methods for DSGE Models (handbook of macro chapter), section 5.

Projection Methods

- To solve a model, need to find policy functions that satisfy FOCs
- Replace policy functions with parameterized approximations
    - e.g. Chebyshev polynomials
- Solve for parameters that approximately solve FOCs
    - minimize sum of squared errors over state space
- Less subject to curse of dimensionality (like log-linearization)
- Global solution (like VFI)
- Resources
    - Heer and Maußner, ch. 6
    - Fernández-Villaverde et al (2016), Solution and Estimation Methods for DSGE Models (handbook of macro chapter), section 5.

Projection Methods

- To solve a model, need to find policy functions that satisfy FOCs
- Replace policy functions with parameterized approximations
    - e.g. Chebyshev polynomials
- Solve for parameters that approximately solve FOCs
    - minimize sum of squared errors over state space
- Less subject to curse of dimensionality (like log-linearization)
- Global solution (like VFI)
- Resources
    - Heer and Maußner, ch. 6
    - Fernández-Villaverde et al (2016), Solution and Estimation Methods for DSGE Models (handbook of macro chapter), section 5.

Projection Methods

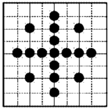
- To solve a model, need to find policy functions that satisfy FOCs
- Replace policy functions with parameterized approximations
    - e.g. Chebyshev polynomials
- Solve for parameters that approximately solve FOCs
    - minimize sum of squared errors over state space
- Less subject to curse of dimensionality (like log-linearization)
- Global solution (like VFI)
- Resources
    - Heer and Maußner, ch. 6
    - Fernández-Villaverde et al (2016), Solution and Estimation Methods for DSGE Models (handbook of macro chapter), section 5.

Projection Methods

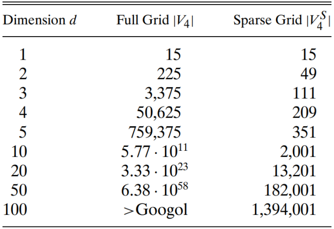
- To solve a model, need to find policy functions that satisfy FOCs
- Replace policy functions with parameterized approximations
    - e.g. Chebyshev polynomials
- Solve for parameters that approximately solve FOCs
    - minimize sum of squared errors over state space
- Less subject to curse of dimensionality (like log-linearization)
- Global solution (like VFI)
- Resources
    - Heer and Maußner, ch. 6
    - Fernández-Villaverde et al (2016), Solution and Estimation Methods for DSGE Models (handbook of macro chapter), section 5.

Sparse Grids

- Sparse Grids: a more efficient way of spacing grids in multi-dimensional problems
    - when function approximated is reasonably smooth, accuracy loss is small
    - objective function in VFI usually is!
- A sparse version of a 9x9 grid: 17 grid points instead of 81



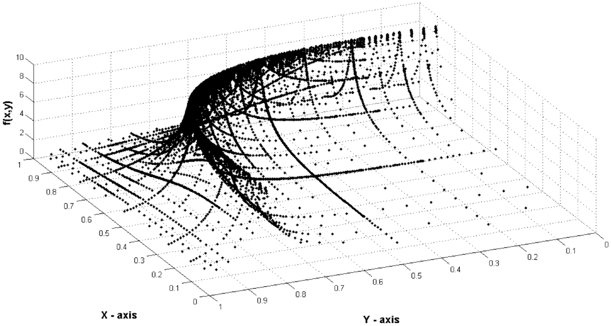
- Gets dramatic for more dimensions:



Dimension d	Full Grid $\|V_4\|$	Sparse Grid $\|V_4^S\|$
1	15	15
2	225	49
3	3,375	111
4	50,625	209
5	759,375	351
10	$5.77 \cdot 10^{11}$	2,001
20	$3.33 \cdot 10^{23}$	13,201
50	$6.38 \cdot 10^{58}$	182,001
100	>Googol	1,394,001

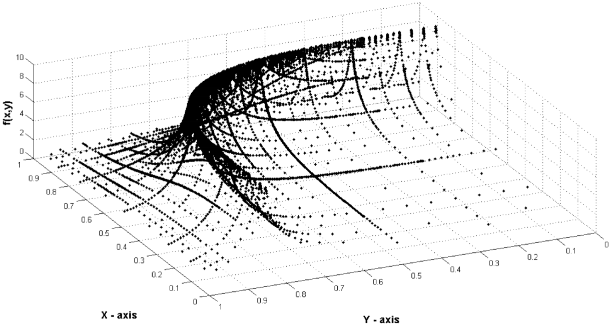
Adaptive Sparse Grids

- Spacing of grids can be varied while solving the model
  - put more grid points where curvature seems higher
- Can put very few grid points in regions with relatively lower curvature
- As fast as classical sparse grids, but far more accuracy



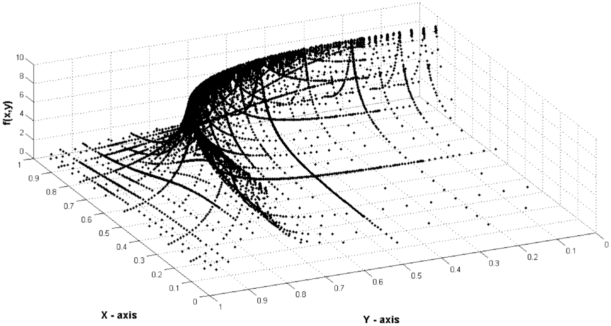
Adaptive Sparse Grids

- Spacing of grids can be varied while solving the model
  - put more grid points where curvature seems higher
- Can put very few grid points in regions with relatively lower curvature
- As fast as classical sparse grids, but far more accuracy



Adaptive Sparse Grids

- Spacing of grids can be varied while solving the model
    - put more grid points where curvature seems higher
- Can put very few grid points in regions with relatively lower curvature
- As fast as classical sparse grids, but far more accuracy



Adaptive Sparse Grids

- Can make VFI feasible for very high-dimensional problems
- Resources
    - Brumm and Scheidegger (2017), Using Adaptive Sparse Grids To Solve High-Dimensional Dynamic Models
    - Tasmanian.jl for Julia

Adaptive Sparse Grids

- Can make VFI feasible for very high-dimensional problems
- Resources
  - Brumm and Scheidegger (2017), Using Adaptive Sparse Grids To Solve High-Dimensional Dynamic Models
  - Tasmanian.jl for Julia

Adaptive Sparse Grids

- Can make VFI feasible for very high-dimensional problems
- Resources
  - Brumm and Scheidegger (2017), Using Adaptive Sparse Grids To Solve High-Dimensional Dynamic Models
  - Tasmanian.jl for Julia

Adaptive Sparse Grids

- Can make VFI feasible for very high-dimensional problems
- Resources
  - Brumm and Scheidegger (2017), Using Adaptive Sparse Grids To Solve High-Dimensional Dynamic Models
  - Tasmanian.jl for Julia

Computation on the GPU

- GPUs have thousands of small cores designed for rendering graphics
- CUDA is Nvidia's library that allows using GPU for general processing
- Thousands of cores $\implies$ need a problem with thousands of parallel tasks
- Data transfer between RAM and GPU is slow $\implies$ each task needs to be big
- VFI with a huge state space is well-suited for GPUs
    - many parallel tasks
    - more computation than communication
    - VFI can be significantly faster on your GPU than your CPU
- Resources
    - Aldrich et al. (2010), Tapping the supercomputer under your desk: Solving dynamic equilibrium models with graphics processors
    - CUDA.jl package for Julia (and its documentation)

Computation on the GPU

- GPUs have thousands of small cores designed for rendering graphics
- CUDA is Nvidia's library that allows using GPU for general processing
- Thousands of cores $\implies$ need a problem with thousands of parallel tasks
- Data transfer between RAM and GPU is slow $\implies$ each task needs to be big
- VFI with a huge state space is well-suited for GPUs
  - many parallel tasks
  - more computation than communication
  - VFI can be significantly faster on your GPU than your CPU
- Resources
  - Aldrich et al. (2010), Tapping the supercomputer under your desk: Solving dynamic equilibrium models with graphics processors
  - CUDA.jl package for Julia (and its documentation)

Computation on the GPU

- GPUs have thousands of small cores designed for rendering graphics
- CUDA is Nvidia's library that allows using GPU for general processing
- Thousands of cores $\implies$ need a problem with thousands of parallel tasks
- Data transfer between RAM and GPU is slow $\implies$ each task needs to be big
- VFI with a huge state space is well-suited for GPUs
  - many parallel tasks
  - more computation than communication
  - VFI can be significantly faster on your GPU than your CPU
- Resources
  - Aldrich et al. (2010), Tapping the supercomputer under your desk: Solving dynamic equilibrium models with graphics processors
  - CUDA.jl package for Julia (and its documentation)

Computation on the GPU

- GPUs have thousands of small cores designed for rendering graphics
- CUDA is Nvidia's library that allows using GPU for general processing
- Thousands of cores $\implies$ need a problem with thousands of parallel tasks
- Data transfer between RAM and GPU is slow $\implies$ each task needs to be big
- VFI with a huge state space is well-suited for GPUs
  - many parallel tasks
  - more computation than communication
  - VFI can be significantly faster on your GPU than your CPU
- Resources
  - Aldrich et al. (2010), Tapping the supercomputer under your desk: Solving dynamic equilibrium models with graphics processors
  - CUDA.jl package for Julia (and its documentation)

Computation on the GPU

- GPUs have thousands of small cores designed for rendering graphics
- CUDA is Nvidia's library that allows using GPU for general processing
- Thousands of cores $\implies$ need a problem with thousands of parallel tasks
- Data transfer between RAM and GPU is slow $\implies$ each task needs to be big
- VFI with a huge state space is well-suited for GPUs
    - many parallel tasks
    - more computation than communication
    - VFI can be significantly faster on your GPU than your CPU
- Resources
    - Aldrich et al. (2010), Tapping the supercomputer under your desk: Solving dynamic equilibrium models with graphics processors
    - CUDA.jl package for Julia (and its documentation)

Computation on the GPU

- GPUs have thousands of small cores designed for rendering graphics
- CUDA is Nvidia's library that allows using GPU for general processing
- Thousands of cores $\Longrightarrow$ need a problem with thousands of parallel tasks
- Data transfer between RAM and GPU is slow $\Longrightarrow$ each task needs to be big
- VFI with a huge state space is well-suited for GPUs
    - many parallel tasks
    - more computation than communication
    - VFI can be significantly faster on your GPU than your CPU
- Resources
    - Aldrich et al. (2010), Tapping the supercomputer under your desk: Solving dynamic equilibrium models with graphics processors
    - CUDA.jl package for Julia (and its documentation)

Computation on the GPU

- GPUs have thousands of small cores designed for rendering graphics
- CUDA is Nvidia's library that allows using GPU for general processing
- Thousands of cores $\implies$ need a problem with thousands of parallel tasks
- Data transfer between RAM and GPU is slow $\implies$ each task needs to be big
- VFI with a huge state space is well-suited for GPUs
    - many parallel tasks
    - more computation than communication
    - VFI can be significantly faster on your GPU than your CPU
- Resources
    - Aldrich et al. (2010), Tapping the supercomputer under your desk: Solving dynamic equilibrium models with graphics processors
    - CUDA.jl package for Julia (and its documentation)

Computation on the GPU

- GPUs have thousands of small cores designed for rendering graphics
- CUDA is Nvidia's library that allows using GPU for general processing
- Thousands of cores $\implies$ need a problem with thousands of parallel tasks
- Data transfer between RAM and GPU is slow $\implies$ each task needs to be big
- VFI with a huge state space is well-suited for GPUs
    - many parallel tasks
    - more computation than communication
    - VFI can be significantly faster on your GPU than your CPU
- Resources
    - Aldrich et al. (2010), Tapping the supercomputer under your desk: Solving dynamic equilibrium models with graphics processors
    - CUDA.jl package for Julia (and its documentation)

Computation on the GPU

- GPUs have thousands of small cores designed for rendering graphics
- CUDA is Nvidia's library that allows using GPU for general processing
- Thousands of cores $\implies$ need a problem with thousands of parallel tasks
- Data transfer between RAM and GPU is slow $\implies$ each task needs to be big
- VFI with a huge state space is well-suited for GPUs
    - many parallel tasks
    - more computation than communication
    - VFI can be significantly faster on your GPU than your CPU
- Resources
    - Aldrich et al. (2010), Tapping the supercomputer under your desk: Solving dynamic equilibrium models with graphics processors
    - CUDA.jl package for Julia (and its documentation)