Introduction to OSCAR and Julia
Lecture 2

Julia workflows (feat. VS Code)

Ibraheem Sajid
University of Leeds (UK)

Overview

- Ways to run Julia code
- A recommended text editor
- Saving objects
- Debugging and getting feedback from your code
- Improving performance

Not mentioned but worth considering:

- Jupyter/Pluto notebooks
- Specify types and define your own structs
- Write your own packages for explicit dependency management

Note: Text between backticks (`) represents code

How to run Julia code

- Just use the REPL (this is what you see when you type `julia` into a terminal)
- Write scripts in a text editor and run them using `julia <filename>` from terminal or `include("<filename")` from the REPL
- Define functions in a file, import it into the REPL using `include("<filename>")` then call functions as normal
  - This has the best of the first two options (performance*, interactivity, ease of editing, IDE integration)
- You can also include files within other files to further modularise your code

* Julia code is fastest in a function due to compilation, but this takes some time on the first call of the function

IDE: VS Code

- WSL integration
- Git integration
- Text editor and terminal in one
- Julia extension with plenty of useful features
  - Syntax highlighting
  - Snippets
  - Code navigation
  - Hover help
  - Plot gallery
  - Some of the above is incomplete due to lacking a decent language server (hopefully fixed soon)
  - https://www.julia-vscode.org/docs/stable/
- Extremely customisable

Revise.jl

- Lets you edit your code and have changes immediately reflected in the REPL without having the call `include` again.
- Install Revise then load it in the REPL (`using Revise`)
  - REPLs launched from the Julia extension in VS Code automatically load Revise
- Include our files with `includet("<filename>")`

Saving objects

- JSON.jl is great for creating human readable output
- `JSON.json("<filename>", x)` serialises the contents of x into a file
  - adding `; pretty=true` to the arguments can be useful
- `x = JSON.parsefile("<filename>")` deserialises a file into x (optionally provide a type as a second argument)

- Be aware of precision issues
- Some types might not be supported
  - Look at Serialization.jl or JLD2.jl

Getting feedback

- Macros in Julia do special things when they prepend code, e.g. change `test()` into `@macro test()`

- @show, @debug/info/warn/error are versatile ways of printing
  - Appending your code with `;` stops it automatically printing its output to the REPL
- @assert checks a condition
- @progress (with ProgressLogging.jl) tracks loop iterations
  - Integrates with VS Code

Debugging (in VS Code)

- Set breakpoints (places to pause code) in the editor
- Hit “Run and Debug” (F5) on a script, or use the `@run` macro in the REPL
  - `@enter` immediately breaks
- When paused, you can inspect and edit variables – you can even execute code using the current workspace

Multithreading

- If you have a loop where each iteration doesn't need to interact with any other iteration, then this is a good candidate for parallelisation (we let the computer run multiple iterations of the loop at once).
- We can give Julia access to multiple threads by starting it with `julia --threads n` (or `julia --threads auto`)
- Then a top-level loop can be parallelised with the macro `Threads.@threads`

- Note that LoopVectorization.jl or Tullio.jl might be a better fit

Benchmarking / Profiling

- `@time` will print the time taken
- Remember that the first time will include compilation (so run a small size first if you have variable input size)
- BenchmarkTools.jl provides more advanced options

- `@profview` measures the time spent in each function
- `@profview_allocs` lets you look at memory allocations
- Can also change the sample rate with e.g. `@profview example_function() sample_rate=0.1` (default is 0.0001)
- If not using VS Code, you'll need to load ProfileView.jl

Thanks!

Any QUESTIONS?