RAMICES III

(JF)^{2}G

April 26, 2025

Oh no, here we go again

We're refactoring RAMICES. Again.

Why? Why would we do this to ourselves? What on God's Green Earth would drive us to such extreme lengths, when we could be doing so much else with our lives?

The long story short is that more than 3 years have passed since it was written: we've both learned a lot. There's also been a bit of space between its original design and the idea that it would have a lifetime outside of my thesis[1]

In particular, there was one big black mark that has hung over RAMICES II (`R2`) which was the stellar catalogue synthesis: Jenny has since found a whole bunch of stuff that's basically held together by hope and good intentions - or simply not held together at all.

Don't Be So Dramatic

I'm being deliberately hyperbolic here. It's worth noting in writing that I don't expect this to be anywhere near as painful as the original Great Refactoring. `R2` is leagues ahead of its predecessor in terms of being able to be read by a mortal mind without causing it to fragment into a trillion tiny pieces.

If all goes well, this will be a nice quick project that results in a vastly superior end product. Make a wish, children!

What This Document Is

Ensuring that this new code update is swift, and non-brain melting means doing some work up front: an up-front statement of intent, and a record of our internal thoughts and decisions which lead to certain elements of the design. That way, when someone says 'why is X like Y', they can look at this document to understand those decision.

What it isn't

This is **not** a checklist of everything that needs doing on a day to day basis. There will be a list of things that we think of at the start, but as new challenges arise, we'll track them using a more suitable project tracking system.

[1] My thesis made some grand claims about it being a general purpose code. Lies.

1 Up Front Thoughts

The main argument we should justify in this document is why a complete code refactor is necessary, rather than simply 'tidying up' the old code.

I think we highlight that the following four things are the main target of RAMICES III:

1. **A Revised Population-Tracking Method**

   An attached theory document details a major overhaul of the current mass-binning of stars. This requires a complete rewrite of the internal logic of the stellar tracking system

2. **A Revised Stellar Catalogue System**

   The current system of synthesising stellar catalogues and selection functions has never been to my satisfaction.

3. **Improved Modularity**

   `R2` is a drastic improvement over its predecessor in terms of being able to 'plug and play' new modules[2], but many things are still somewhat hardcoded and assumed, simply because they were necessary for the Thesis development.

4. **More Sophisticated Development**

   `R2` was written in three months of PhD-Crunch. It is not a particularly pretty sight, there is little to no internal documentation, and things such as the git history are best left unspoken-of. We should like to write these things with a mind to future maintenance and use by people who still have their sanity.

1.1 Other Design Thoughts

1.1.1 Test-Driven Development

A risk of scientific software design is that you end up attributing some intriguing scientific interpretation to a software bug. Both `R1` and `R2` are highly state-dependent systems with a lot of external state-modification. This can make debugging the system highly difficult, and allows for the possibility of strange and unusual software bugs.

When refactoring the code we should strive to adhere to the following principles:

- *Functions should return values, not modify input objects*

  It is generally easier to validate if `float y` is the expected value of `float f(T x)`, than checking to see if `x.y` matches expectations after calling `void f(T & x)`

- *Complex Actions should have their own helper Functor*

  Rather than relying on writing a long method for (say) the Galaxy to handle the radial mixing prescription, give the Galaxy as `MixingHandling` object which can be called. Since this then separates out the radial mixing from the other state of the Galaxy, this allows for a more easy testing

- *Objects should have (static) factory methods*

  This allows us to initialise objects into a known state, and thereby test them (and apply tests to them) from controlled starting points

[2] This is evidence by Jenny's NSD work, which remains astounding to me that it worked at all.

- *Test early, test often*

  Each meaningful functional unit should have its own test - and the tests should be written (or at least, conceptualised) before the function itself is written.

1.1.2 Code Re-Use

Much of the code is dedicated to specialised scientific algorithms which is largely independent of the overarching software design.

We're not throwing the baby out with the bathwater: these sections of the code are largely going to be perfectly functional. We do not need to rewrite, for example, the section of code dedicated to the generation of Stellar Yield Tables from SynthNet data: the algorithm is fine – we just need to ensure that the final product is written into a new, compliant object, instead of a `std::vector<std::vector<std::vector<double»>`, as it is currently.

Where possible, therefore, we will try to re-use algorithmic code, wrapping it inside our new design paradigm.

1.1.3 Minimises JSL Dependence

The JSL library is good at what it does (which is act as a central repo for C++ code that I use all the time).

However, it's not particularly suitable for deployment into an external codebase. We should extract the useful parts and have them as part of the core Ramices sourcecode.

We should think about how much of JSL to port over. Do we want to include the JSL::gnuplot library to have plotting integrated into the simulation?

1.1.4 Input & Output

The JSL::Argument package already does a reasonable job of allowing for runtime configuration of the simulation, through standard command-line arguments and configuration files. However, manually messing around with config files is irksome. Jack has already developed a fun python module which allows for easy construction of a config-file GUI. We should implement our own – this requires no rewriting of the config-handling system, but is instead a system built

2 Modular Design

In keeping with our design principle that a GCE simulation is a cohesive collection of disparate parts, we shall endeavour to write the code in a way which minimises the interactions between these distinct segments. This will make validation easier, and improve the modularity – by minimising the interaction between units, we make it much more possible to 'swap out' various units, and thereby extend this code to perform more physics than it was originally designed for! From a software design standpoint, it also isolates individual units which is a good design principle.

Of course, it also means that we get to decide what to call each of the individual modules. That being said, I remember the annoyance of trying to remember what `ariadne` did versus `persephone`. The temptation to give everything fun, thematic names is great, but it's not sustainable in the long term. I suggest that modules have a 'formal name' which describes what they do[3], and a 'code name', which is a short, snappy name used to refer to them in the code, even if it's not fully representative of what the module is doing.

`ramices` This is the main simulation module, and the focal point of the code. The RaMiCES module will incorporate all of the information form the other modules, and channel information from one module to another.

`isochrone` (or, *Computed Lifetimes & Isochrone Observables*: `CLIO`)

This module is responsible for the interpretation of Isochrone information. This is used at two points in the code: first for inferring the mass-lifetime relationship of stars and the (much more complex) task of inferring the properties of the stars at the end of the simulation, such that an observable catalogue can be synthesised.

`yield` (or, *Metal & Remnant Yield Tables*: `MeRYT`)

This modules is responsible for computing the end-of-lifetime processes: the elemental composition of ejecta, as well as determining the nature of any remnants. It should be possible for this module to be lifted more or less wholesale from the original code – given that the `R2` implementation was already lifted out of the version I wrote for `R1`.

`population` (or, *Observation Synthesis, Reflecting Instrument Selection*: `OSyRIS`)

This module is the one which will probably require the most new theory, and comprises the module which generates the

`utility` (or, *Testers, Handlers, Operators, Templates & Helpers*: `THOTH`)

This is a basic module containing a bunch of 'stuff' – mainly the non-scientific portions of the code, such as loggers, parallel code execution, file readers and so on. The utility module is a basic grab-bag of functions which are used in multiple places.

`test` (or, *Automated Testing Network*: `ATeN`)

This is the unit test module. It is the most important module if we are committed to going ahead with a Test-Driven Development philosophy, and the one that should have considerable thought put into before the others. The test module is obviously unusual.

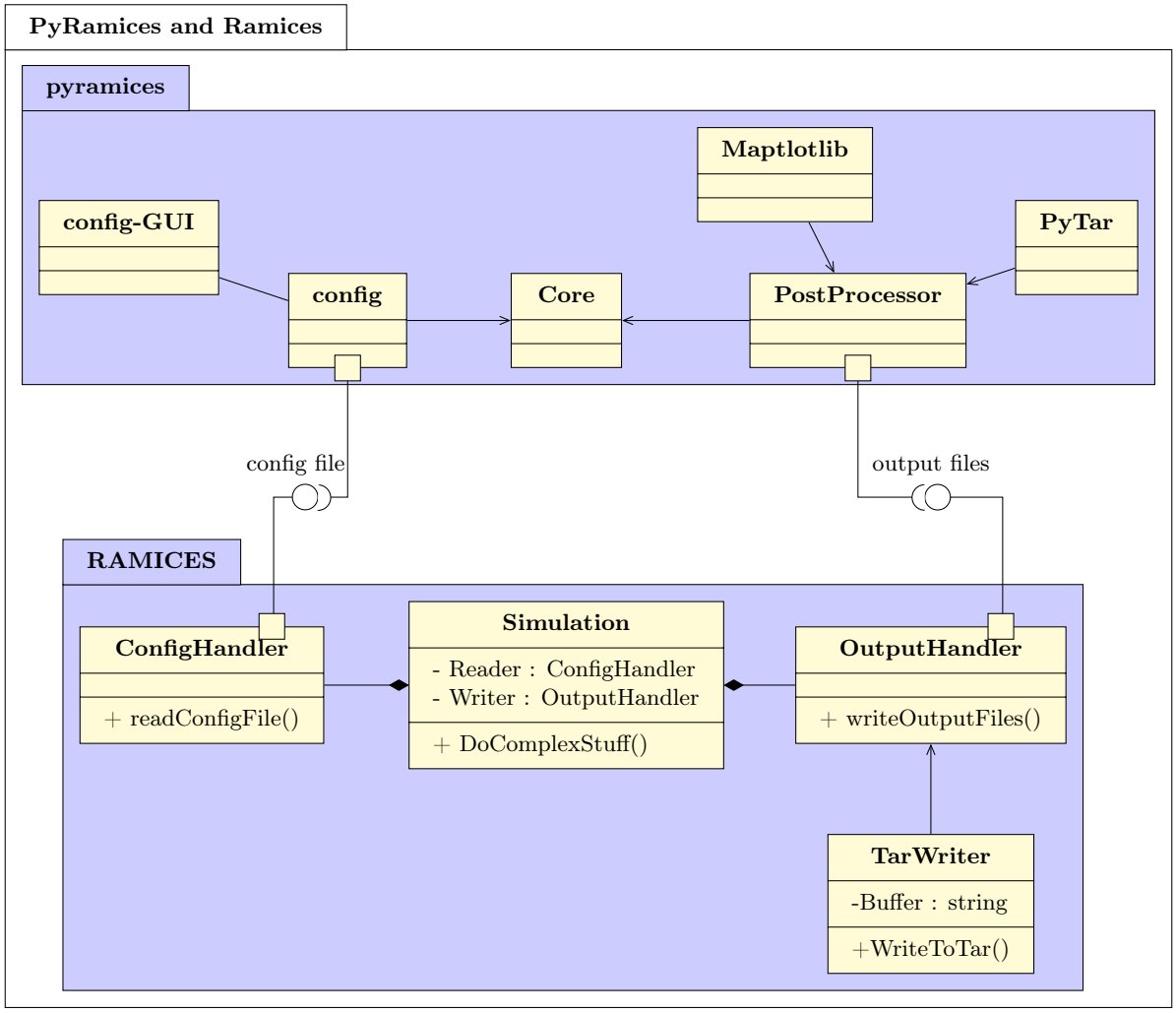
[3] And by sheer happenstance, happens to have a fun, Egyptian-thematic acronym.

3 PyRamices

One potential issue with our 'neater output' ideology[4] is that it makes post-processing of the data troublesome. Of course, since tarballs are common data-types, we could just expect users to un-tar when they want to access files – but then files spill out all over the place, and we're back to square one.

Given that we are already in the process of implementing a (Python-based) GUI framework for the code, the logical extension is to provide an entire wrapper environment. This would (largely) amount to a wrapper class and some aliased calls to the excellent pytar library. That way you can configure, run and then post-process the data, all from within python.

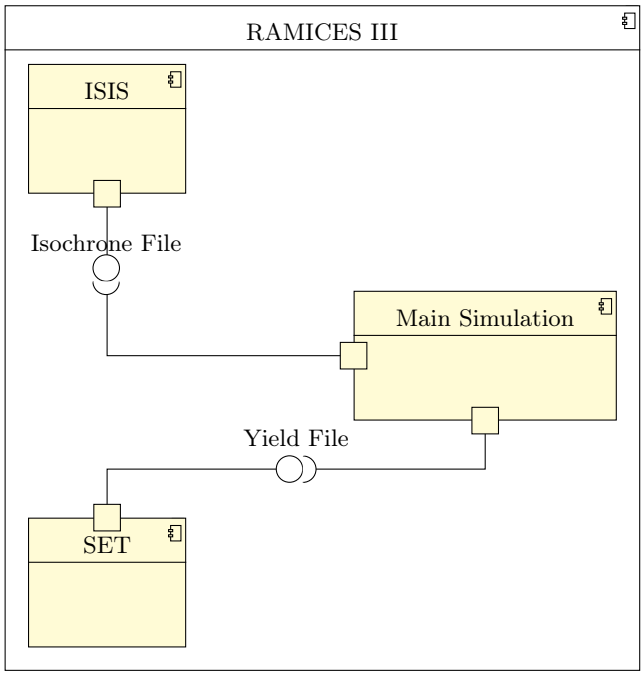
An approximate design looks like this:

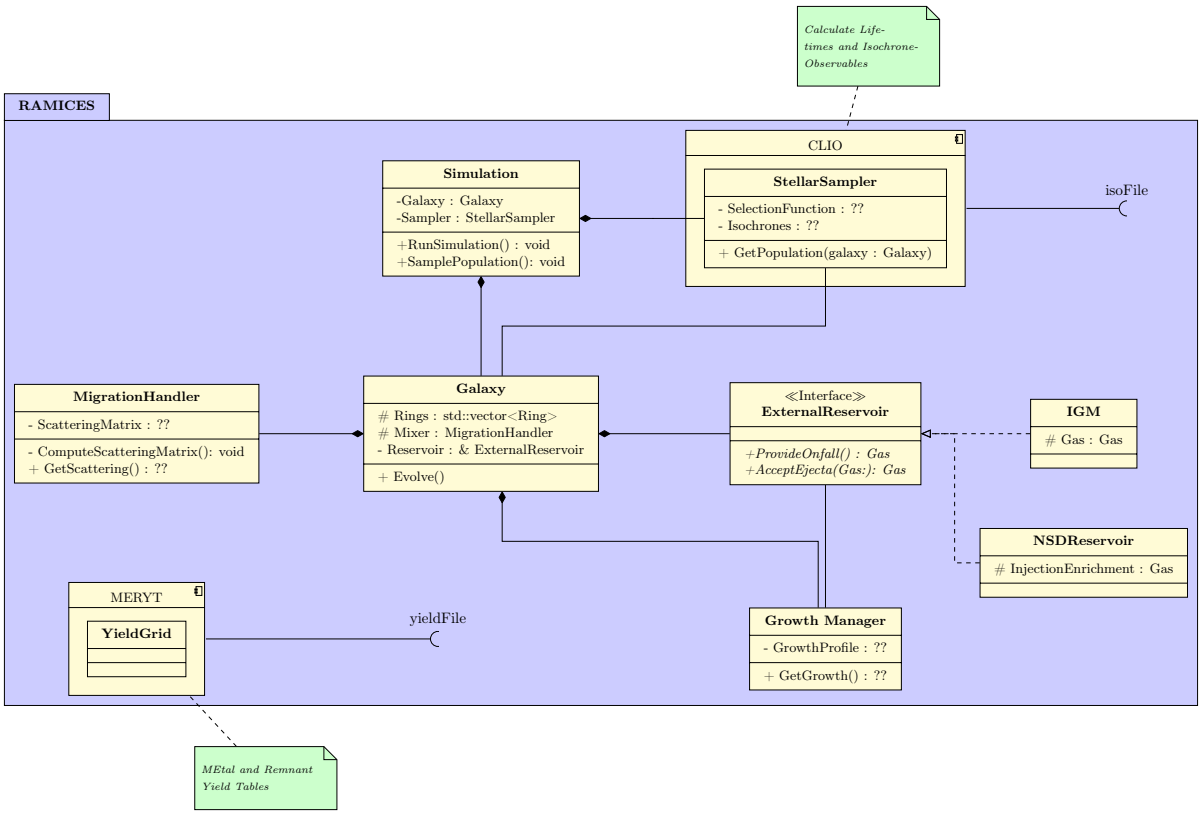


Functionally, all 'pyramices' needs to do is generate config files, launch instances of the main simulation, and then read output files – this is stuff we would be doing already, and so it cannot hurt to formalise it.

[4]This will take the form of packaging the dozens of files into basic tarballs, for which I already have a neat C++ implementation

4 Diagrams





(Note: + is public, # is protected and - is private)