MaCh3's User Interface

From a zoo of binaries to one coherent CLI

Picking Up Where the Last Talk Left Off

- **MaCh3 serves at least three audiences:**
  - Users
  - Developers
  - In-betweeners
- Today's reality: the boundary between these is blurry at best
  - Users almost need to know as much as developers just to use the software.
- Goal: More of a clear distinction between these types.

What “The User Interface” Means Today

- **Three layers a MaCh3 user currently has to deal with, in increasing order of required expertise:**

1

Executables

A large, flat collection of standalone binaries: MCMCTutorial, ProcessMCMC, RHat, CombineMaCh3Chains, PlotLLH, MatrixPlotter, PredictivePlotting, GetPostfitParamPlots, …

2

YAML Configuration

e.g. MaCh3Tutorial/TutorialConfigs — covering fitter settings, sample definitions, systematic models, binning

3

C++ Source Code

e.g. registering a “functional parameter“ etc…

What “The User Interface” Means Today

- **What will this talk address?**

1

Executables
A large, flat collection of standalone binaries: MCMCTutorial, ProcessMCMC, RHat, CombineMaCh3Chains, PlotLLH, MatrixPlotter, PredictivePlotting, GetPostfitParamPlots, …



2

YAML Configuration
e.g. MaCh3Tutorial/TutorialConfigs — covering fitter settings, sample definitions, systematic models, binning



3

C++ Source Code
e.g. registering a “functional parameter“ etc…



Motivation Problem 1: A Sprawl of Binaries

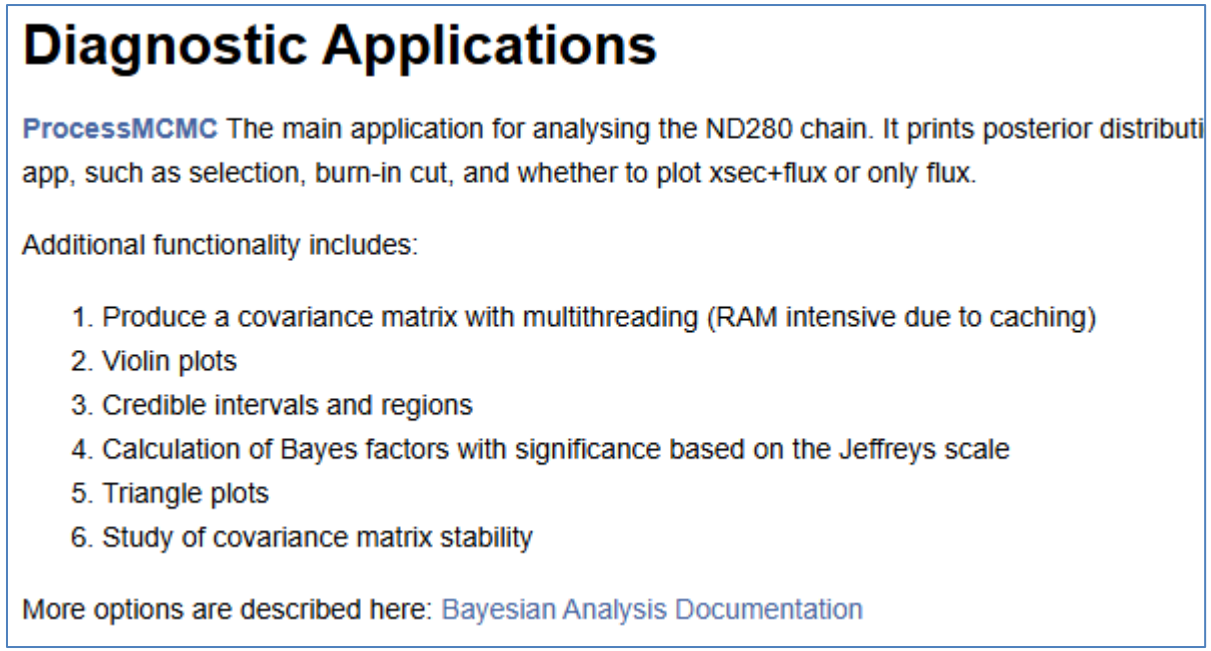
Before this work, MaCh3 shipped one executable per task, each with its own argument conventions:

```
$> ./bin/MCMCTutorial        TutorialConfigs/FitterConfig.yaml
$> ./bin/ProcessMCMC         bin/TutorialDiagConfig.yaml Test.root
$> ./bin/RHat                1000 MCMC_Test_1.root ... MCMC_Test_4.root
$> ./bin/CombineMaCh3Chains  -o MergedChain.root MCMC_Test_1.root ...
$> ./bin/PlotLLH             LLH_Test.root
$> ./bin/MatrixPlotter       bin/TutorialDiagConfig.yaml Test_drawCorr.root
```

- No shared argument parsing conventions — some take configs first, some take ROOT files first, some take raw numbers
- No shared --help, no tab-completion, no discoverability
- Every new tool = a new ad-hoc CLI to learn

Motivation Problem 2: Inconsistent User Experience

- For a user of the CLI tools, if help is needed then the main source would be the doxygen.
- While this is very detailed in terms of implementational details it's not always clear about usage instructions.



Diagnostic Applications

ProcessMCMC The main application for analysing the ND280 chain. It prints posterior distributi
app, such as selection, burn-in cut, and whether to plot xsec+flux or only flux.

Additional functionality includes:

1. Produce a covariance matrix with multithreading (RAM intensive due to caching)
2. Violin plots
3. Credible intervals and regions
4. Calculation of Bayes factors with significance based on the Jeffreys scale
5. Triangle plots
6. Study of covariance matrix stability

More options are described here: Bayesian Analysis Documentation

Motivation Problem 2: Inconsistent User Experience

- For a user of the CLI tools, if help is needed then the main source would be the doxygen.
- While this is very detailed in terms of implementational details it's not always clear about usage instructions.
- The CLI tools have some inline help, but it is limited and also aborts in an unfriendly way.

Diagnostic Applications

ProcessMCMC The main application for analysing the ND280 chain. It prints posterior distributi
app, such as selection, burn-in cut, and whether to plot xsec+flux or only flux.

Additional functionality includes:

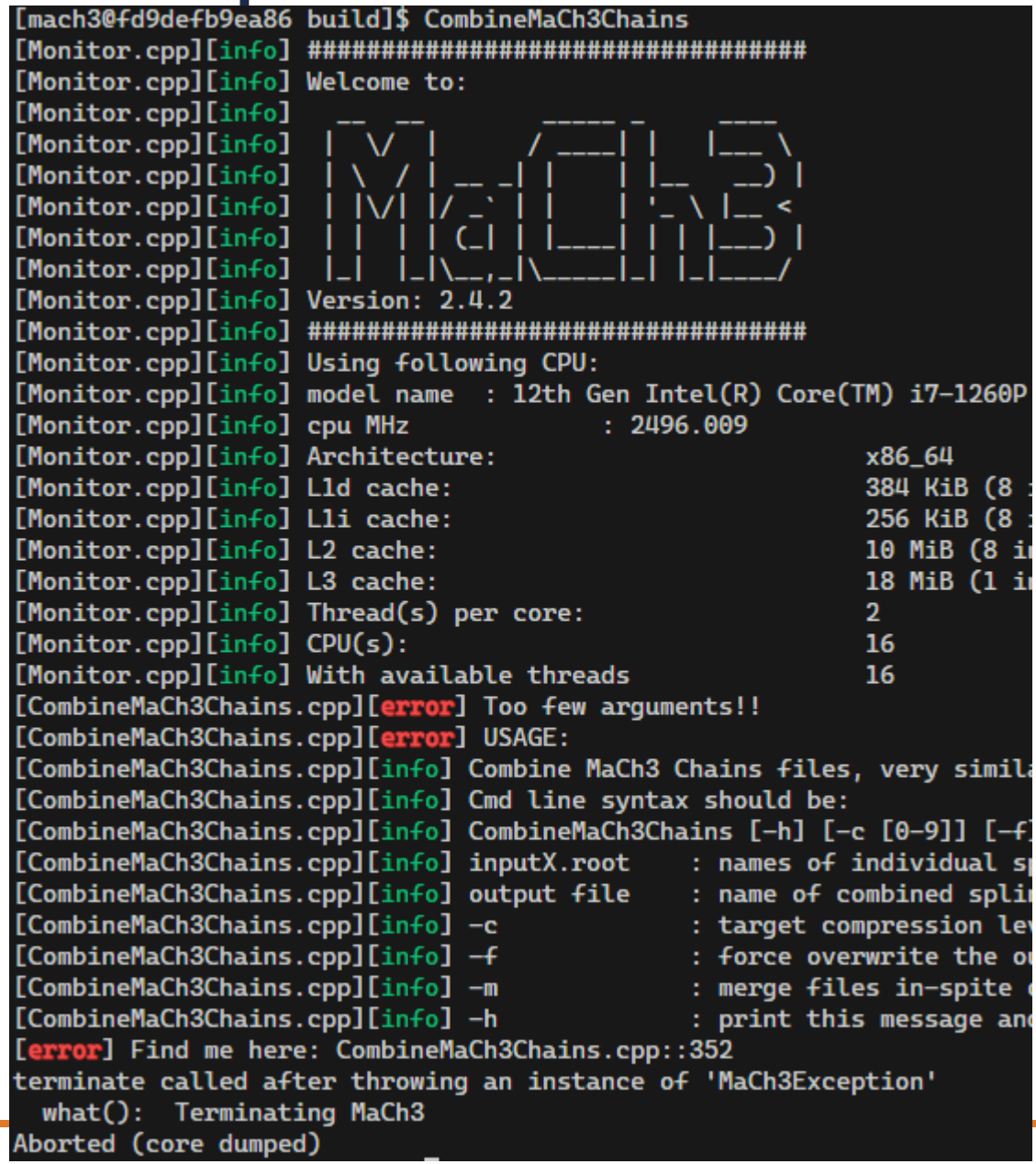
1. Produce a covariance matrix with multithreading (RAM intensive due to caching)
2. Violin plots
3. Credible intervals and regions
4. Calculation of Bayes factors with significance based on the Jeffreys scale
5. Triangle plots
6. Study of covariance matrix stability

More options are described here: Bayesian Analysis Documentation

```
[root@42b203a7f56e build]# ProcessMCMC
[ProcessMCMC.cpp][error] How to use:
[ProcessMCMC.cpp][error]   single chain: ProcessMCMC <Config> <MCMM_ND_Output.root>
[ProcessMCMC.cpp][error]   two chain:    ProcessMCMC <Config> <MCMM_ND_Output_1.root> <Title 1> <MCMC_ND_Output_2.root> <Title 2>
[ProcessMCMC.cpp][error]   three chain:  ProcessMCMC <Config> <MCMM_ND_Output_1.root> <Title 1> <MCMC_ND_Output_2.root> <Title 2> <MCMC_ND_Output_3.root> <Title 3>
[error] Find me here: ProcessMCMC.cpp::42
terminate called after throwing an instance of 'MaCh3Exception'
  what():  Terminating MaCh3
Aborted (core dumped)
```

Motivation Problem 2: Inconsistent User Experience

- For a user of the CLI tools, if help is needed then the main source would be the doxygen.
- While this is very detailed in terms of implementational details it's not always clear about usage instructions.
- The CLI tools have some inline help, but it is limited and also aborts in an unfriendly way.
- It can also be obfuscated with other output.



Motivation Problem 2: Inconsistent User Experience

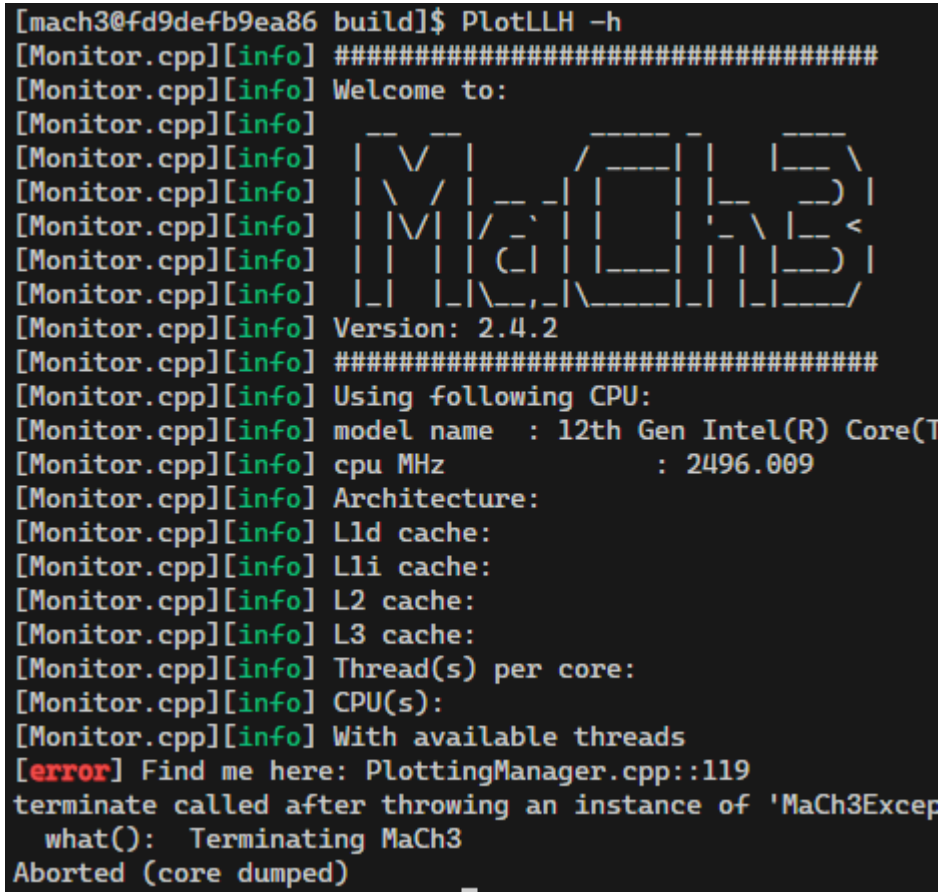
- For a user of the CLI tools, if help is needed then the main source would be the doxygen.
- While this is very detailed in terms of implementational details it's not always clear about usage instructions.
- The CLI tools have some inline help, but it is limited and also aborts in an unfriendly way.
- It can also be obfuscated with other output.
- In some cases, the tool will crash

```
[mach3@fd9defb9ea86 build]$ PlotLLH
[Monitor.cpp][info] ###################################
[Monitor.cpp][info] Welcome to:
[Monitor.cpp][info]   __  __          _____ _     ____
[Monitor.cpp][info]  |  \/  |      / ____| |   |___ \
[Monitor.cpp][info]  | \  / | __ _| |    | |__   __) |
[Monitor.cpp][info]  | |\/| |/ _` | |    | '_ \ |__ <
[Monitor.cpp][info]  | |  | | (_| | |____| | | |___) |
[Monitor.cpp][info]  |_|  |_|\__,_|\_____|_| |_|____/
[Monitor.cpp][info] Version: 2.4.2
[Monitor.cpp][info] ###################################
[Monitor.cpp][info] Using following CPU:
[Monitor.cpp][info] model name  : 12th Gen Intel(R) Core(TM) i7-1
[Monitor.cpp][info] cpu MHz             : 2496.009
[Monitor.cpp][info] Architecture:                          x86_64
[Monitor.cpp][info] L1d cache:                             384 KiB
[Monitor.cpp][info] L1i cache:                             256 KiB
[Monitor.cpp][info] L2 cache:                              10 MiB
[Monitor.cpp][info] L3 cache:                              18 MiB
[Monitor.cpp][info] Thread(s) per core:                    2
[Monitor.cpp][info] CPU(s):                                16
[Monitor.cpp][info] With available threads                 16
[PlottingManager.cpp][info] Input files provided
[PlottingManager.cpp][info] No file labels specified, will just u
Info in <TCanvas::Print>: pdf file LLHScan_Sample.pdf has been cr
Warning in <TPDF::TPDF::NewObject>: An Object is already open.
Info in <TCanvas::Print>: pdf file LLHScan_Penalty.pdf has been c
Warning in <TPDF::TPDF::NewObject>: An Object is already open.
Info in <TCanvas::Print>: pdf file LLHScan_Total.pdf has been cre

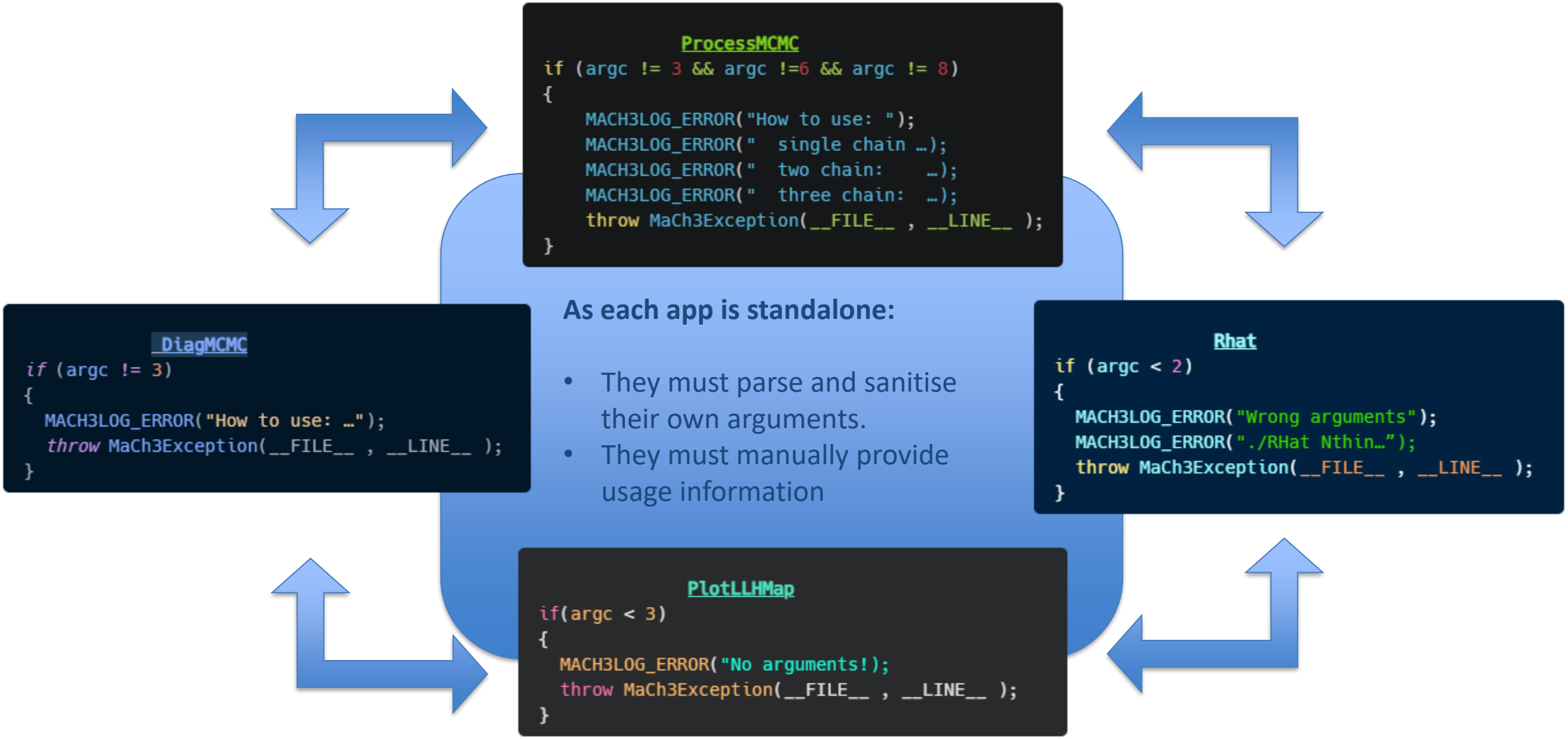
 *** Break *** segmentation violation
 Generating stack trace...
 0x000000000040d46d in <unknown> from PlotLLH
 0x000000000040e303 in <unknown> from PlotLLH
 0x00007c4c34d4e5d0 in <unknown> from /lib64/libc.so.6
 0x00007c4c34d4e680 in __libc_start_main at :? from /lib64/libc.s
 0x00000000004087d5 in <unknown> from PlotLLH
```

Motivation Problem 2: Inconsistent User Experience

- For a user of the CLI tools, if help is needed then the main source would be the doxygen.
- While this is very detailed in terms of implementational details it's not always clear about usage instructions.
- The CLI tools have some inline help, but it is limited and also aborts in an unfriendly way.
- It can also be obfuscated with other output.
- In some cases, the tool will crash
- Or not have help at all



Motivation Problem 3: Repeated Boilerplate

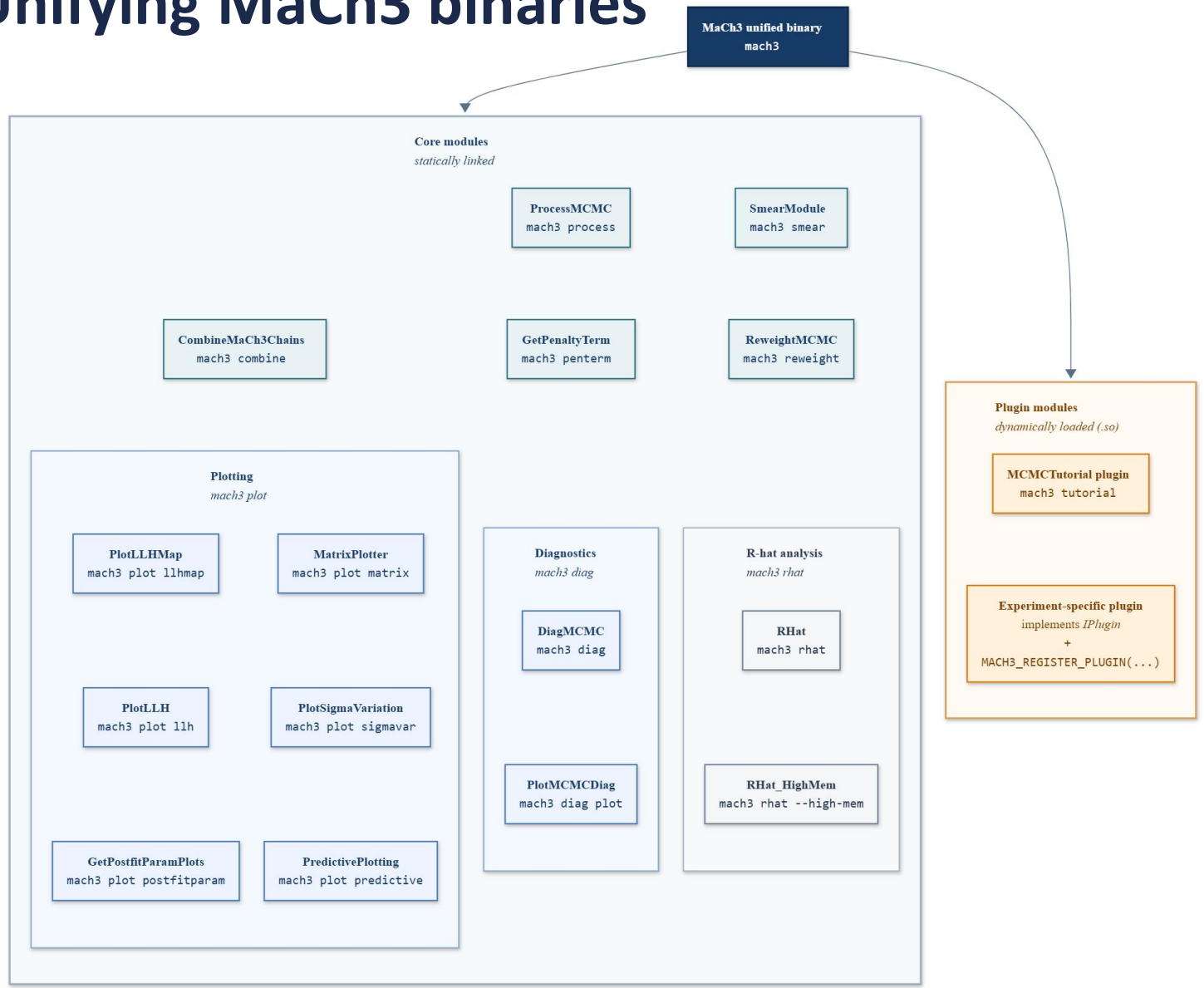


The Model: Look at git

- git is one binary with sub-commands: git commit, git log, git rebase — each with its own flags but a shared feel
- New functionality (git lfs, git flow) plugs in without bloating the core binary
- This is exactly the UX model adopted for MaCh3:
- **One mach3 binary, with sub-commands for every task**

```
$> mach3 tutorial  TutorialConfigs/FitterConfig.yaml       # was: MCMCTutorial
$> mach3 process   bin/TutorialDiagConfig.yaml Test.root   # was: ProcessMCMC
$> mach3 diag      bin/TutorialDiagConfig.yaml Test.root   # was: DiagMCMC
```

Unifying MaCh3 binaries




```
$> mach3 process
$> mach3 diag
$> mach3 diag plot
$> mach3 penterm
$> mach3 rhat
$> mach3 rhat --high-mem
$> mach3 smear
$> mach3 reweight
$> mach3 combine
$> mach3 plot llh
$> mach3 plot llhmap
$> mach3 plot matrix
$> mach3 plot sigmavar
$> mach3 plot postfitparam
$> mach3 plot predictive
$> mach3 tutorial
```

Coherent User interface

- New unified structure gives a single coherent entry point with detailed inline help

```
$> mach3 --help
Usage: mach3 [--help] [--version] [--install-completions] {combine,diag,penterm,plot,process,reweight,rhat,smear}

Optional arguments:
  -h, --help            shows help message and exits
  -v, --version         prints version information and exits
  --install-completions

Subcommands:
  combine               Tool for combining multiple chains or ROOT files into a single output.
  diag                  Tool for MCMC diagnostic like autocorrelations.
  penterm               Calculate penalty term for selected parameters, for every step
  plot                  Plotting subcommand.
  process               Main executable responsible for different types of MCMC processing like drawing posteriors,
triangle plots etc.
  reweight              Tool for MCMC reweighting.
  rhat                  Tool for computing the R-hat diagnostic for MCMC chains.
  smear                 Tool for smearing MCMC chains.
```

Coherent User interface

- This extends to subcommands as well.

```
$> mach3 process --help
Usage: mach3 process [--help] [--corr] [--MakeCredibleIntervals] [--CalcBayesFactor] [--CalcSavageDickey] [--CalcBipolarPlot] [--
CalcParameterEvolution] CONFIG MCMC_CHAIN1 [TITLE1 MCMC_CHAIN2 TITLE2 [MCMC_CHAIN3 TITLE3]]

Main executable responsible for different types of MCMC processing like drawing posteriors, triangle plots etc.

Positional arguments:
  CONFIG                                                  Config file. [required]
  MCMC_CHAIN1 [TITLE1 MCMC_CHAIN2 TITLE2 [MCMC_CHAIN3 TITLE3]]  MCMC chain root files and titles.
                                                          single chain mode: MCMC_CHAIN1
                                                          two chain mode   : MCMC_CHAIN1 TITLE1 MCMC_CHAIN2 TITLE2
                                                          three chain mode : MCMC_CHAIN1 TITLE1 MCMC_CHAIN2 TITLE2 MCMC_CHAIN3
TITLE3 [nargs=1..6] [required]

Optional arguments:
  -h, --help                                              shows help message and exits
  --corr                                                  plot correlation - Same as PlotCorr option in config.
  --MakeCredibleIntervals                                 Same as MakeCredibleIntervals option in config.
  --CalcBayesFactor                                       Same as CalcBayesFactor option in config.
  --CalcSavageDickey                                      Same as CalcSavageDickey option in config.
  --CalcBipolarPlot                                       Same as CalcBipolarPlot option in config.
  --CalcParameterEvolution                                Same as CalcParameterEvolution option in config.

  ProcessMCMC The main application for analysing the ND280 chain.
  It prints posterior distributions after the burn-in cut and allows comparison of two or three different chains.
  Several options can be configured directly in the app, such as selection, burn-in cut, and whether to plot xsec+flux or only flux.

  Additional functionality includes:
    Produce a covariance matrix with multithreading (RAM intensive due to caching)
    Violin plots
    Credible intervals and regions
    Calculation of Bayes factors with significance based on the Jeffreys scale
    Triangle plots
    Study of covariance matrix stability
```

Coherent User interface

- Incorrect usage will also print help along with a description of the usage error.

```
$> mach3 diag
config: 1 argument(s) expected. 0 provided.
Usage: mach3 diag [--help] CONFIG MCMC_CHAIN

Positional arguments:
  CONFIG      Config file. [required]
  MCMC_CHAIN  MCMC chain root file. [required]

Optional arguments:
  -h, --help  shows help message and exits

Optional subcommands:
  plot        Tool for plotting MCMC diagnostic like autocorrelations.
[error] [mach3.cpp:69] config: 1 argument(s) expected. 0 provided.
```

What about the boilerplate

Developer boilerplate problem required:

- Module developers must parse and sanitise their own arguments.

```
int DiagMCMCModule::Run() {
  std::string config = m_parser->get<std::string>("config");
  std::string inputFile = m_parser->get<std::string>("mcmc-output");

  YAML::Node Settings = M3OpenConfig(config);
  auto Processor = std::make_unique<MCMCProcessor>(inputFile);
  ...
}
```

What about the boilerplate

Developer boilerplate problem required:

- Module developers must parse and sanitise their own arguments.

Parsers responsible for gathering and sanitising correct arguments and developers can just use them.



```
int DiagMCMCModule::Run() {
  std::string config = m_parser->get<std::string>("config");
  std::string inputFile = m_parser->get<std::string>("mcmc-output");

  YAML::Node Settings = M3OpenConfig(config);
  auto Processor = std::make_unique<MCMCProcessor>(inputFile);
  ...
}
```

What about the boilerplate

Developer boilerplate problem required:

- Module developers must parse and sanitise their own arguments.

Parsers responsible for gathering and sanitising correct arguments and developers can just use them.

```
int DiagMCMCModule::Run() {
  std::string config = m_parser->get<std::string>("config");
  std::string inputFile = m_parser->get<std::string>("mcmc-output");

  YAML::Node Settings = M3OpenConfig(config);
  auto Processor = std::make_unique<MCMCProcessor>(inputFile);
  ...
}
```

- Module developers must manually provide usage information

```
$> mach3 diag --help
Usage: mach3 diag [--help] CONFIG MCMC_CHAIN

Tool for MCMC diagnostic like autocorrelations.

Positional arguments:
  CONFIG      Config file. [required]
  MCMC_CHAIN  MCMC chain root file. [required]

Optional arguments:
  -h, --help  shows help message and exits
```

What about the boilerplate

Developer boilerplate problem required:

- Module developers must parse and sanitise their own arguments.

Parsers responsible for gathering and sanitising correct arguments and developers can just use them.

```
int DiagMCMCModule::Run() {
  std::string config = m_parser->get<std::string>("config");
  std::string inputFile = m_parser->get<std::string>("mcmc-output");

  YAML::Node Settings = M3OpenConfig(config);
  auto Processor = std::make_unique<MCMCProcessor>(inputFile);
  ...
}
```

- Module developers must manually provide usage information

Sort of… Developers need to provide descriptions, but formatting and concatenation of options is provided.

```
$> mach3 diag --help
Usage: mach3 diag [--help] CONFIG MCMC_CHAIN

Tool for MCMC diagnostic like autocorrelations.

Positional arguments:
  CONFIG      Config file. [required]
  MCMC_CHAIN  MCMC chain root file. [required]

Optional arguments:
  -h, --help  shows help message and exits
```

Nothing comes completely for free

Developers must still define the parsers options and descriptions:

```
MaCh3ArgumentParser* DiagMCMCModule::get_parser(){
  m_parser = std::make_unique<MaCh3ArgumentParser>("diag", "1.0", argparse::default_arguments::help);
  m_parser->add_description("Tool for MCMC diagnostic like autocorrelations.");
  m_parser->add_argument("config")
    .help("Config file.")
    .metavar("CONFIG")
    .required();
  m_parser->add_argument("mcmc-output")
    .help("MCMC chain root file.")
    .metavar("MCMC_CHAIN")
    .required();
  return m_parser.get();
}
```

However, …

- This is more declarative (and easier to follow) than mysterious number of argc and argv
- This should be familiar to Python programmers.
- Hopefully you agree, It's worth it for the achieved unified binary + subcommands style
- Gives the possibility to consider creating one shot tools to create common options further negating excessive developer burden.
- Can possibly use this to autogenerate Doxygen documentation for the cli commands meaning documentation is always up to date and in sync with interactive help, without the need for specific Doxygen comments

Under the hood: The Plugin Interface

- Every sub-command, built in or external implements the same tiny interface
- Aliased to IModule for core application sub-commands as this terminology was preferred by developers

```
class IPlugin {
    public:
        virtual ~IPlugin() = default;
        virtual int Run() = 0;                          // do the work
        virtual MaCh3ArgumentParser* get_parser() = 0;  // expose it's own CLI args
};
```

- Convenience abstract base class takes care of ensuring that all plugins (modules) have a parser member
- Provides a layer for future abstraction

```
class PluginBase : public IPlugin {
    public:
        virtual ~PluginBase() = default;

    protected:
        std::unique_ptr<MaCh3ArgumentParser> m_parser;
};
```

Example: Registering an External Plugin in One Macro

For code that lives outside the core MaCh3 repository (experiment-specific tools, tutorials, …):

- Inherit from PluginBase
- Compiled to shared library (.so)
- A single macro MACH3_REGISTER_PLUGIN exposes a plain C ABI so it can be dlopen'd

```
class MCMCTutorialPlugin: public PluginBase{
  public:
    MaCh3ArgumentParser* get_parser();
    int Run();
};

MACH3_REGISTER_PLUGIN(MCMCTutorialPlugin)
```

MACH3_REGISTER_PLUGIN(MCMCTutorialPlugin) expands to:

```
extern "C" M3::IPlugin* create_plugin () { return new MCMCTutorialPlugin(); }
extern "C" void destroy_plugin(M3::IPlugin* p) { delete p; }
```

MaCh3Program: The top-level dispatcher

MaCh3Program is the top-level dispartcher as well as argparser. It is responsible for:

- Collecting together arguments for the top-level mach3 binary
- Accumulating core modules for dispatch
- Loading and unloading dynamic plugins for dispatch
- Setting up parser tree from core modules and plugins
- Parsing command line arguments
- Dispatching to the correct sub-command
- Installing completion scrips and running completions

```
class MaCh3Program: public MaCh3ArgumentParser{
    public:
        virtual ~MaCh3Program(){
            this->unload_dynamic_plugins();
        }
        /// @brief Add a core module to the program
        void add_core_module(IModule& module);
        /// @brief Load dynamic plugins from the MACH3_PLUGINS environment v
        void load_dynamic_plugins();
        /// @brief Install shell completion scripts for bash and zsh
        void install_completions() const;
        /// @brief Unload all dynamically loaded plugins
        void unload_dynamic_plugins();
        /// @brief Run the selected subcommand
        int Run();
        ...
}
```

- add_core_module

registers a compiled-in module as a sub-parser

- load_dynamic_plugins

reads MACH3_PLUGINS (colon-separated paths/dirs), dlopen's each .so, calls its create_plugin, and registers it exactly like a core module

- **Run**

finds whichever sub-command the user actually typed and forwards to its Run(), whether that's a built-in module or a dynamically loaded plugin — the dispatcher does not care which

- install_completions

shell tab-completion support (bash & zsh)

mach3.cpp: Wiring It All Together

The main mach3 executable code is now streamlined and concise.

```
int main(int argc, char *argv[]) {
    M3::MaCh3Program program("mach3");

    program.add_argument("--complete").hidden().nargs(1);
    program.add_argument("--install-completions").flag();

    M3::ProcessMCMCModule    proc;
    M3::DiagMCMCModule       diag;
    program.add_core_module(proc);
    program.add_core_module(diag);
    // ...
    program.load_dynamic_plugins();     // pick up .so plugins too

    program.parse_args(argc, argv);
    return program.Run();               // dispatch to whichever subcommand matched
}
```

Nice Side-Effects of Unifying the CLI

Because there's now exactly one binary and one argument-parsing framework:

- We provide shell tab-completion for free:
  - mach3 --install-completions, then
  - mach3 <TAB> lists sub-commands
  - mach3 process --<TAB> lists that sub-command's flags
- Discoverability: mach3 --help, mach3 tutorial --help, mach3 process --help all behave consistently
- One install / one PATH entry, rather than a bin/ directory full of loosely related executables
- Uniform way of implementing config overrides across sub-commands (see example below)

```
$> mach3 tutorial TutorialConfigs/FitterConfig.yaml --override General:MCMC:NSteps:100000
# or short form:
$> mach3 tutorial TutorialConfigs/FitterConfig.yaml -o General:MCMC:NSteps:100000
```

- Previously overrides were only viable for binaries using the main config file as other binaries would interpret the extra command line args as input files

Smooth Migration …

The rollout is intentionally incremental:

- Old standalone binaries still exist for now but print a deprecation warning pointing at the new sub-command (see below)

```
$> bin/MCMCTutorial TutorialConfigs/FitterConfig.yaml
[warn] Deprecation Warning: Use of the standalone executable will be deprecated in future releases.
[warn] : you can use 'mach3 tutorial' as a direct replacement instead.
...
```

- The old binaries are simple wrappers around the new sub-commands with the added benefit of getting new-style help for free

```
$> bin/DiagMCMC --help
[warn] [DiagMCMC.cpp:5] Deprecation Warning: Use of the standalone executable will be deprecated in future releases.
[warn] [DiagMCMC.cpp:6]                     : you can use 'mach3 diag' as a direct replacement instead.
Usage: diag [--help] CONFIG MCMC_CHAIN

Tool for MCMC diagnostic like autocorrelations.

Positional arguments:
  CONFIG      Config file. [required]
  MCMC_CHAIN  MCMC chain root file. [required]

Optional arguments:
  -h, --help  shows help message and exits
```

Smooth Migration …

The rollout is intentionally incremental:

- Not every old executable has an equivalent sub-command *yet* — the tutorial docs explicitly flag this
  - May not actually be true by the time of this talk – PR basically ready

- **Experiment-specific executables can be migrated independently, at their own pace, by wrapping existing logic in an IPlugin and registering it — no need to wait for a core MaCh3 release**

Summary

- MaCh3's user interface has three layers: binaries, YAML, and C++ — and we have been addressing the first
- The new mach3 CLI improves the user's day to day experience
  - Single binary gives a cohesive feel to software
  - git-style sub-commands
  - Shell completion
  - Ability to have uniform config overrides (including explicit options for common flags)
- It does not address the other UI layers — but it gives us the stable foundation to consider improvements to them.
  - A shared IPlugin interface
  - A stable plugin ABI (IPlugin + MACH3_REGISTER_PLUGIN) with dynamic plugin loading to ship extensions as plugins without bloating the main executable.
  - A single, consistent argument-parsing surface is the natural place to have CLI schema definition and validation

> *Try it today: mach3 tutorial TutorialConfigs/FitterConfig.yaml*

Questions?

Thank you.