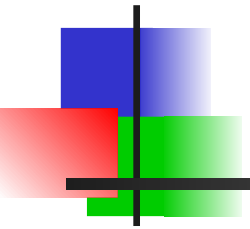
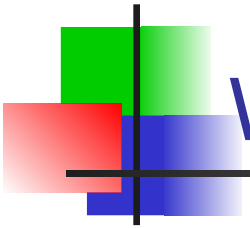


PyOpal - a Python API for OPAL

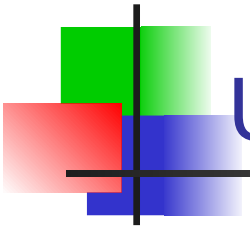
C. Rogers

Science & Technology Facilities Council
ISIS



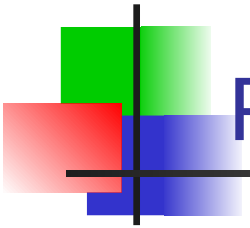
Why?

- All OPAL use cases require pre/post-processing
  - Some sort of programming/scripting tool required to do this
- MAD language is quite limited
  - Basic programming syntax
    - Even some common programming features are not supported
  - No plotting facilities
  - No support libraries
  - Known only within accelerator community
- Python is the *de facto* standard scripting tool for scientists
  - Reasonable plotting library (matplotlib)
  - Reasonable scientific/numeric libraries (numpy/scipy)
  - Good support libraries e.g. subprocess management
  - Well-known inside and outside accelerator community



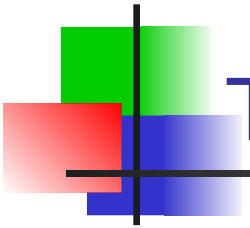
Use Cases

- Two distinct use cases
  - Track a lattice and look at performance
    - Variable manipulation in MAD language is sufficient
      - e.g. Calculating energy vs momentum
      - e.g. Calculating trigonometric functions/geometry factors
    - Requires some post-processing/plotting
  - Optimise a lattice with space charge
    - Requires lattice manipulation (e.g. changing magnet setting/etc)
    - Requires post-processing/plotting
    - Probably subprocess management
- Python interface very useful in the second use case
  - May help with the first
- How should we do it?



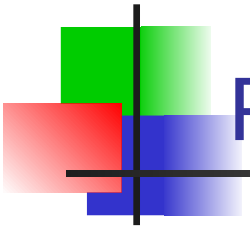
Python interface - options

- Options for implementing lattice file interface to python
  - Text manipulation
    - Direct writing of OPAL input files
  - Python/C API
    - Call OPAL C++ functions directly from python
  - Boost::Python
    - Wraps Python/C API, simplifying some function calls
  - SWIG
    - Scripting tool to automatically generate Python/C API code
  - Cython
- Post-processing is also a thing



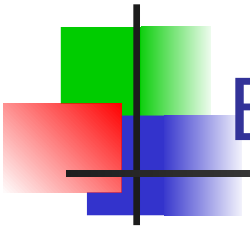
Text Manipulation

- Basic concept – call python functions to:
  - Modify an existing lattice file
  - Generate entirely new lattice elements
  - Write to a text file
  - Run OPAL subprocesses
  - Read output files into memory
- Pros:
  - Relatively easy to implement
  - Maybe possible to “backfill” with direct Python/C API calls
- Cons:
  - Can limit execution time for many small jobs, where “set up” is a major part of the execution time



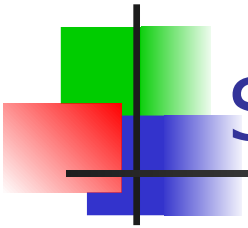
Python/C API

- Basic concept – call existing OPAL C functions directly
- Pros:
  - OPAL calls can be as quick as any python function
    - e.g. move/scale elements “on the fly”
    - Helpful when running lots of short jobs
  - Can implement things like “what is the field at this point?”
    - Useful for debugging
    - Useful for analysis (canonical momenta etc)
- Cons:
  - More fiddly to implement
  - Another dependency to support
    - Python loads modules dynamically => “libOPAL.so”
  - Note – python does not support multithreading



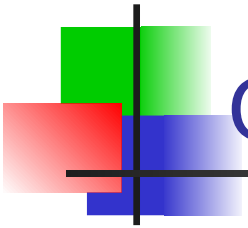
Boost::Python

- Boost::Python library provides template classes to help make Python/C API code
- Pros
  - C++ (object oriented)
  - Better handling of memory/etc
- Cons
  - Another dependency to support – tho’ Boost is already supported



Swig

- Generate python/C API code automatically
  - Based on “definitions” .i file
- Pros
  - Pretty straight forward to implement simple things
  - Support for other languages in addition to python (java, perl, R, ...)
- Cons
  - Poor documentation for anything complicated
  - Rumours of performance issues

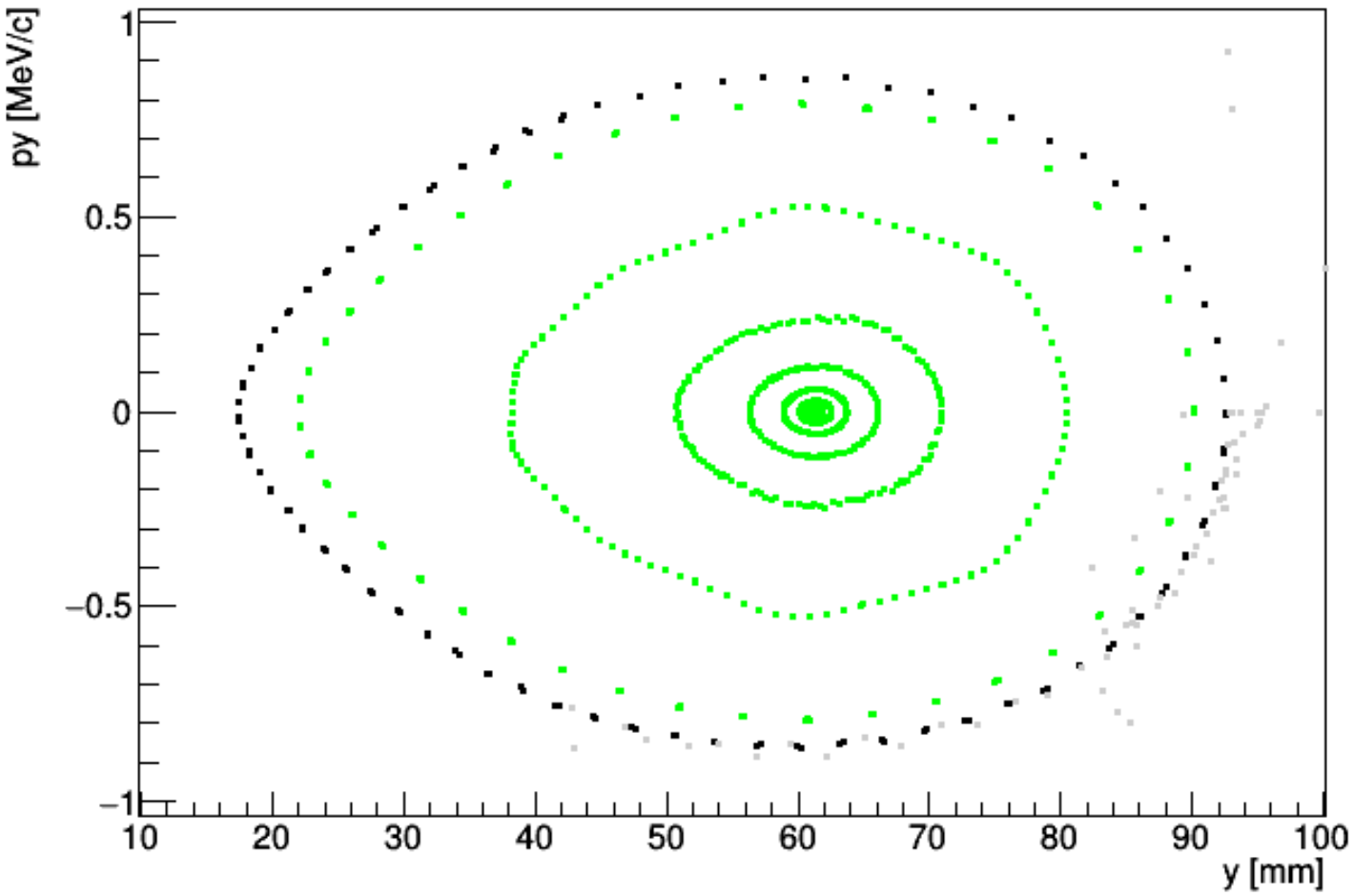


Cython

- Write compilable python-like code
  - Handle statically typed things in python
    - Rather than making C wrappers that are dynamically typed
  - Can interface with/wrap C functions
- Pros
  - Good for python programmers who are scared of C
  - Enables statically typed python – which may run faster
- Cons
  - Need to write wrapper classes for all C objects that we want to use
  - Rapidly becomes quite complicated, even for simple things
  - Cython is a learning curve for C++ programmers
    - More so than native C++ solutions

Dynamic aperture calculation

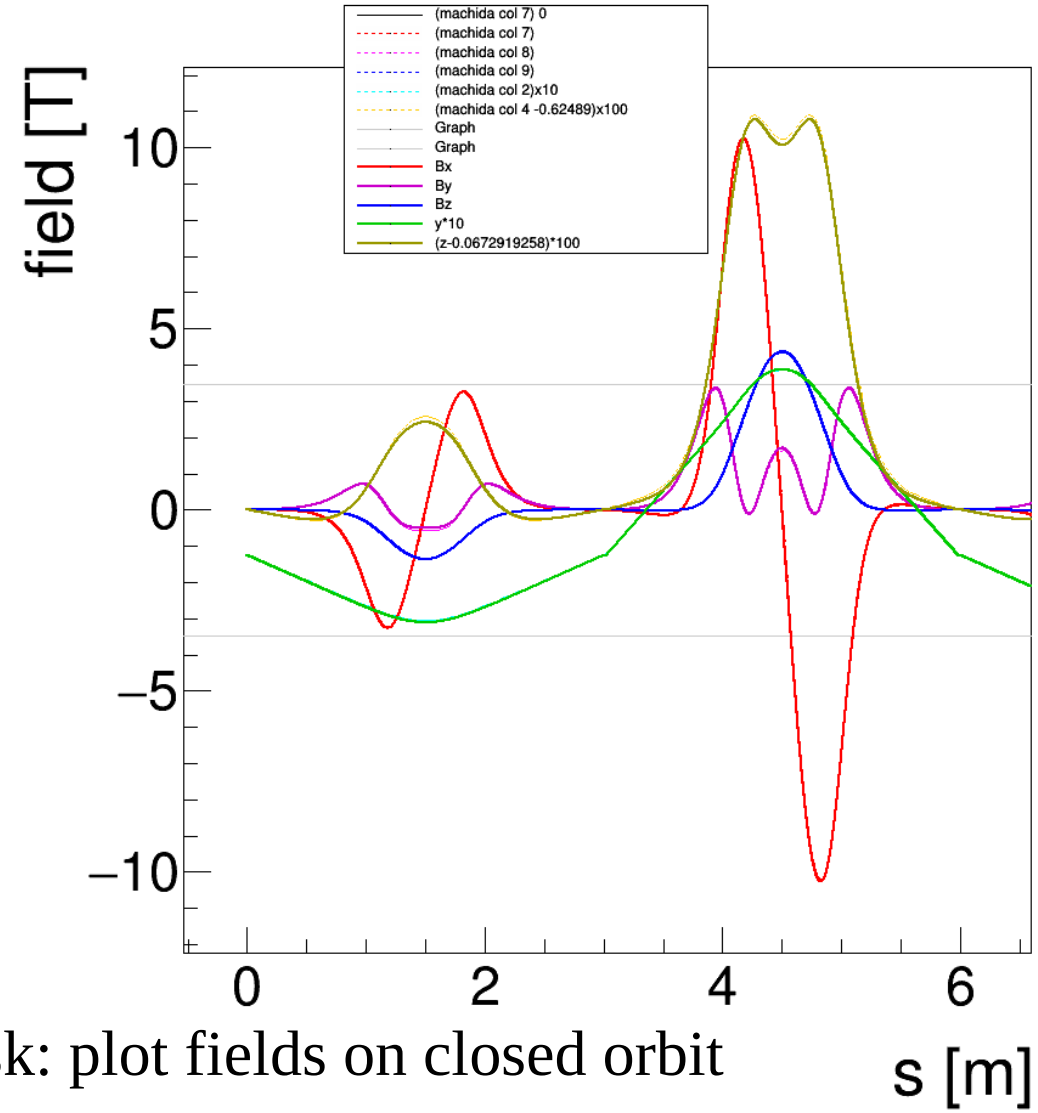
Task: calculate DA



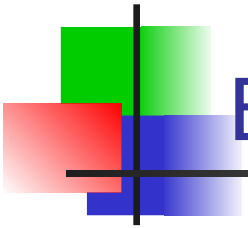
Example: text file manipulation

- Task: calculate DA
- Class OpalTracking
  - Read lattice input file
  - Substitute keywords for values (e.g. step size, field strength, etc)
  - Write lattice file for Opal in temp folder
  - Write distribution input file in temp folder
  - Run Opal using python subprocess module
  - Read output probe files
- Class DAFinder
  - Track a particle few mm from closed orbit using OpalTracking
  - Count number of turns the particle survived
  - If > 100 turns
    - Particle is inside DA; move away from closed orbit
  - If < 100 turns
    - Particle is outside DA; move towards closed orbit
  - Iterate

Field on Closed Orbit

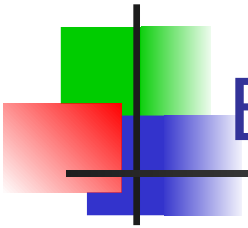


Task: plot fields on closed orbit



Example: Python/C API

- Task: plot fields on closed orbit
- Add compiler options to link to python
  - Add python library
  - Add option to build *libOpal.so*
- All python/C code goes in new directory *src/Py0pal/*
  - Add CMake function to build .so file for each python module
  - Add *src/PyOpal* to *${PYTHONPATH}* environment variable
  - Add *__init__.py* to *src/PyOpal*
  - Add *PyParser.cpp* which initialises OPAL and parses OPAL input file
  - Add *PyField.cpp* which handles API calls to *Ring* to get fields
  - Modify *CmakeLists.txt* to generate *parser.so* and *field.so*
- Now
  - *import parser* called from python will import parser module
  - *Import field* called from python will import field module

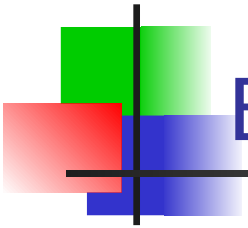


Example: Python/C API (2)

- Within Python C code, each modules must perform few tasks
- Define module attributes
  - Module name
  - Module functions
  - Documentation string for the module
- Module functions are defined as a C array of structs each having
  - Function name
  - Function pointer
  - Options for function setup
  - Documentation string
- Function to initialise the module
- Nb: similar API for python classes also exists
  - Define class attributes
  - Define class methods (including memory allocation/deallocation)
  - Define struct that holds class data
  - Initialise class in the module initialisation function

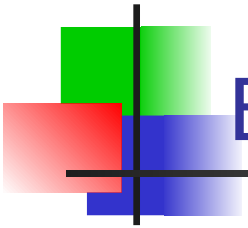
Example: Python/C API (3)

- A few thoughts
  - Clearly python/C API is quite complicated on development level
  - Much easier on user level (couple of commands)
    - Load OPAL lattice
    - Do the field lookups
  - On the other hand, field lookups would be clunky if implemented in OPAL directly
    - Python writes out a set of points to a support file
    - Add a command to the OPAL input lattice file
    - Call OPAL to load in the points
    - OPAL writes out the field values to another support file
    - Python reads them in
  - … and developers still need to implement function to “get field value on arbitrary set of points”



Boost::Python

- Boost::Python somewhat simplifies the job of writing Python/C API
  - Simpler API for C++ objects that manipulate primitives
  - Better type safety
  - Can always fall back on regular Python/C API
- As we already support Boost, probably good idea to use Boost::Python



Examples

- Examples
  - git@gitlab.psi.ch:ext-rogers_c/src.git
- See src/PyOpal/

Discussion

- Various options exist for embedding OPAL within python
  - Text file wrapping in the first instance
  - Binary API offers more elegant solution
    - Practical improvement only for some use cases
- Personal approach is:
  - Use text file manipulation for running OPAL
    - Some “optimisation” of OPAL run, e.g. switch off field map output when running quick jobs
  - Use Python/C API calls for stuff that doesn’t fit well e.g. field look ups