Philosophy of pycrtools

Heino Falcke

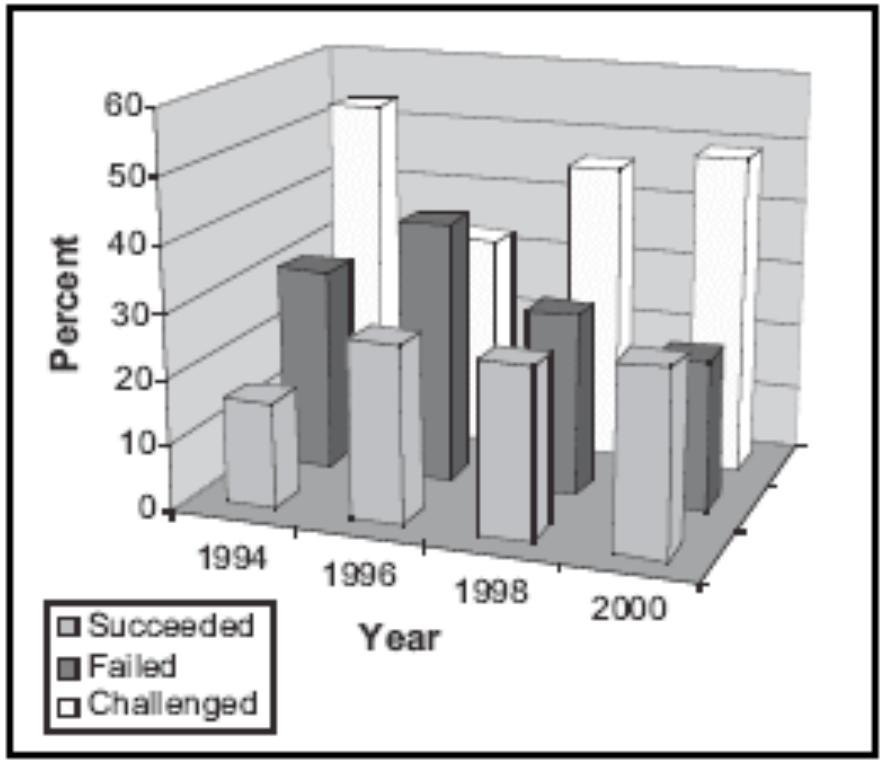
Personal Wisdom

- Programming since 30 years
- Basic, Assembler, Pascal, Fortran 4/77/90, c, c++, Lisp, Prolog, python and other scripting and batch languages (IRAF, Glish, csh, etc…)
- Produced several scientific software pipelines, that produced papers and scientific results:
  - Visualization of biological cell kinetics
  - Line-driven disk wind simulation
  - GR ray tracing program
  - Optical end-to-end speckle imaging pipeline
  - HST image pipeline
  - NIR spectra reduction pipeline
  - VLA & VLBI data reduction pipeline
  - LOPES pipeline
  - etc., etc.

- It is worth investing time in a (fully) automated pipeline.
  - More upfront work, but more powerful
  - repeatable data reduction -> better results
  - Higher inclination to do things over again or try different ways
- Important:
  - Good planning in the beginning
  - Walk through the data reduction steps “manually” once.
  - If some functionality is missing to do things automatically, rejoice that you now can do something useful and write a new function to do it.
  - Make your environment as pleasant as possible (e.g., start things automatically, define enivronment variables, links, shortcuts to get you going).
  - Note things down for future use
  - Combine steps into single pipeline

Most software projects fail!

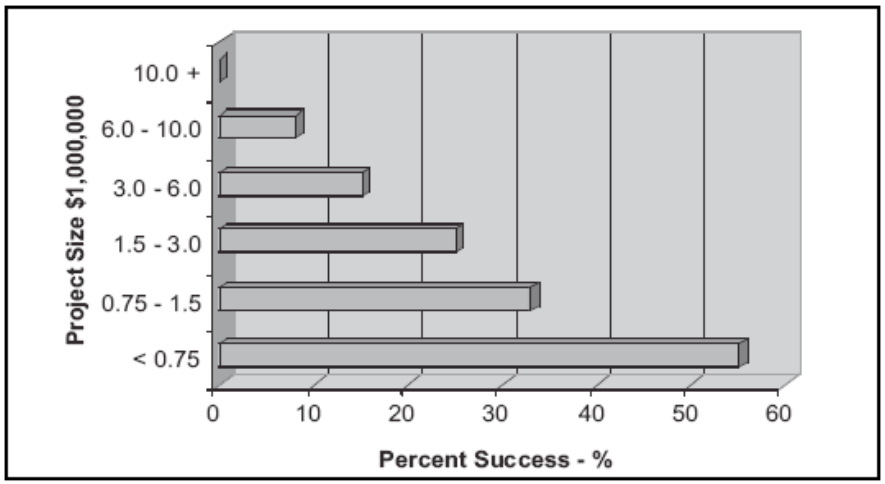
- In 2000 only 20% of software projects succeeded!
- “The principal problem was the lack of plans [1, 2]. In the early years, I never saw a failed project that had a plan, and very few unplanned projects were successful.”
- The problem: physicists are not used to accept any plan other that their own. (HF)



W.S. Humphrey – Why Big Software Projects Fail
http://www.stsc.hill.af.mil/crosstalk/2005/03/0503humphrey.html

Success Rate

- We are in the 1.5 - 3 M$ project range for the (Py)CRTools
- We have a 25% chance of success already (under normal project management circumstances)
- Main problem: SW developers are unmanageable egos
- Practical management problem
  - “With a team of software developers, you cannot tell what they are doing by merely watching. You must ask them or carefully examine what they have produced. It takes a pretty alert and knowledgeable manager to tell what software developers are doing. If you tell them to do something else or to adopt a new practice, you have no easy way to tell if they are actually working the way you told them to work.” (WSH)
- Software development works well with 1-2 people, it becomes ineffective when 4 or more people are involved



Programming for yourself is easy

- Andreas Nigl: Viewers, Browser, daily dynamic spectra
- Stijn Buitink: NuMoon WSRT data processing pipeline
- Sven lafebre: Library of cosmic ray events
- Kalphana Singh: RFI excision routines

- None of these packages were ever used again after the programmer left ….

History of CR-Tools

- First version developed by HF and A. Horneffer:
  - Based on GLISH (scripting language of aips++, GLISH was heritage of the Superconducting Super Collider SSC project which was killed by the US senate/congress)
  - "get-mechanism" which would perform all operations on the fly
  - GUI-based – all operations including pipeline were actually controlled through the GUI but also all intermediated data product could be viewed immediately
- Pros: Excellent for developing a new technique – full view of what the data looked like: crucial for success of LOPES and build-up of LOFAR CR/AUGER radio! Was used by group in Nijmegen and Karlsruhe
- Cons: Scripting makes it slow, poor documentation, restricted to small numbers of antennas

2nd Phase

- Idea: Port the pipeline completely to c++, later wrap this with python
- Result: Complete disaster
  - Software essentially never worked and was unmanageable
  - Forrest of classes that could
    - hardly be wrapped and exposed to python (design flaw)
    - constant changes which had to be propagated through the many layers of inheritance,
    - poor control of data processing, memory, speed, etc.
  - The software became the main focus not the result

3rd try - PyCRTools

- Provide a simple and fast c++ library with compute intensive functions
  - Simple functions no classes!
  - Easy to expand
- Use python as scripting language and provide wrappers for all functions
  - Loop and compute in c++ and glue in python
- Make an integrated approach where python (and other) wrappers are produced automatically

Guiding Principles of pycrtools

- Tools for physicists and astronomers who are expert users, working at the forefront of current knowledge
- Operations will be compute intensive and memory and CPU limited – need full control of these resources
- Work in both worlds: c++ and python
- Build a common development and user frame work
- In the development phase easy and interactive access to all steps and intermediate data products is crucial
- Full responsibility of the expert user: “The software is supposed to do what I tell it to do and not tell me what I can and cannot do!”.
- If you have programmed something wrong the data will tell you! Errors are tolerable - don’t worry: nobody will die, but your final graph will look weird and that is what you need to catch.

pycrtools

- hftools library:
  - Provide a library of very fast and simple c++ routines working with large chunks of *sequential* data in memory (there is no 2D memory yet …).
  - Very short algorithms, trimmed for maximum speed and minimum bureaucratic overhead (it is a Formula I car and not a Family Van!).
  - Few lines only - typically 5-10, most <30 lines (easy to understand and check)
  - Avoid classes and inheritance as much possible - just pure function
  - Memory allocation (including working memory) is done outside the library, i.e. full control and responsibility is with the user
  - Minimum number of error checks – again full freedom and responsibility to the user. Avoid crashing but also throwing errors as much as possible - data will tell.
  - Use templated input, use a minimum number of data types (no uints ….), and use iterator/pointer logic (faster and very direct …)
  - For more complicated algorithms (e.g. Spline interpolation) wrap GSL functions (similar philosophy)

pycrtools

- hftools library:
  - Provide a wrapper generation mechanism that allows one to access the functions with different types of arrays and vectors (e.g., c vectors, c++ STL vectors, CASA arrays, numpy arrays ….) for other future applications
  - Declare things only once (.h files are automatically generated)!
    (Inconsistent .h files are an annoying source of frequent errors, also psychologically important since it reduced the emotional barrier to add a new function).

Example: hFill

```
//$DOCSTRING: Fills a vector with the content of another vector.
//$COPY_TO HFILE START ----------------------------------------------
#define HFPP_FUNC_NAME hFill
//----------------------------------------------------------------------
#define HFPP_WRAPPER_TYPES HFPP_ALL_PYTHONTYPES
#define HFPP_FUNCDEF  (HFPP_VOID)(HFPP_FUNC_NAME)("$DOCSTRING")
(HFPP_PAR_IS_SCALAR)()(HFPP_PASS_AS_VALUE)
#define HFPP_PARDEF_0 (HFPP_TEMPLATED_TYPE_1)(vec)()("Vector to fill")
(HFPP_PAR_IS_VECTOR)(STDIT)(HFPP_PASS_AS_REFERENCE)
#define HFPP_PARDEF_1 (HFPP_TEMPLATED_TYPE_2)(fill_vec)()("Vector of values to fill it with")
(HFPP_PAR_IS_VECTOR)(STDIT)(HFPP_PASS_AS_REFERENCE)
//$COPY_TO END ------------------------------------------------
/*!
  hFill(vec,[0,1,2]) -> [0,1,2,0,1,2,...]
  vec.fill([0,1,2]) -> [0,1,2,0,1,2,...]

  \brief $DOCSTRING
  $PARDOCSTRING
If fill_vec is shorther than vec, the procedure will wrap around and
start from the beginning of fill_vec again. Hence, in this case
fill_vec will appear repeated multiple times in vec.
*/
```

Example: hFill

```
template <class Iter, class IterT>
void HFPP_FUNC_NAME(const Iter vec,const Iter vec_end, const IterT fill_vec, const IterT
fill_vec_end)
{
  if ((fill_vec<=fill_vec_end) || (vec<=vec_end)) return;
  Iter it1=vec;
  IterT it2=fill_vec;
  while (it1!=vec_end) {
    *it1=hfcast<IterValueType>(*it2);
    ++it1;++it2;
    if (it2==fill_vec_end) it2=fill_vec;
  };
}
//$COPY_TO HFILE: #include "hfppnew-generatewrappers.def"
```

Random Notes

- Provide minimum number of parameters
- Vector length (vec_end-vec) is an important parameter that determines the behaviour
  - e.g. hFillRange(vec, start=0.1, increment=1.5) -> vec=[0.1,1.6,3.2,…] needs no end parameter, that is set by the length of the vector
- Python wrappers:
  - Functions are available as pure functions in python
  - Also available as methods (with all lower caps and no preceding h, e.g. vec.hfillrange) to hArrays and vectors. This needs an extra step to include them manually in modules/core/types.py”

hArrays

- They are a convenience container to a sequential section of memory – they are not ordinary arrays. You need to think about how data is arranged in memory!
- Multiple dimensions are only used to access certain slices of memory and to loop over them (see looping instructions in tutorial)
- hArrays can store default parameters for plotting its contents (e.g. xvalues, logplot, etc.), can have a descriptive name, and log operations on it through a logging system.
  - If set-up well “spectrum.plot()” will give you a publication ready plot of the spectum with log axes, Axes descriptions, and frequency in MHz on the x-axis.

hArrays

- Even a multiple dimension array can be accessed simply as 1D
- Example:

```
a=hArray([20,256])
a[…].fftw(a[…]) # take the FFT of the 20 rows
a.abs() # treat as 1D and take abs of all values
```

- Use the looping mechanism if possible – don't loop in python!
- Note: Several hArrays with different dimensions can point to the same memory (vector) – just the container to access it is different.

```
a=hArray([2,256])
b=hArray(a.vec(),[512]) # is the same as above
```

Development steps

- Think and design your algorithm (sometimes done using numpy)
- Check what others have done
- Write the most basic functional unit in c++ in the hftools – if it needs more than 30 lines think more modular
- Test it with a python test script on real data
- Turn test script into a “Task”
- Integrate Tasks into one pipeline

Personal Experience

- Personally I never made so few errors during development of a scientific software:
  - Few iterations during compilation
  - Quick functional testing
  - Quick end-to-end prototypes and results with real data
- Software is very fast and memory efficient, if used properly
  - Simple abs of vector was 30-40% faster than numpy
  - FFTW was 300-1000% faster than old FFT
  - I was able to maximize memory usage and, e.g., do an FFT with 3 Hz resolution instead of 60 Hz in memory

Using other people's software????

- "Oh, it would have taken me much longer to understand it then to re-write it myself"….
- Maybe, but consider the following:
  - It usually takes longer than you think
  - You are paid not only to do science yourself, but also to build a new telescope: testing, improving, expanding, better documenting existing software is part of that process that needs to be done.
  - If noone ever reviews other people's software we we will never get a tested and documented system.
  - How long will it take to get your software up to that level? Will you ever do it?

Using other people's software????

- Every software has limitations one has to work around or wait for bugs to be fixed. The fact that you can rewrite parts of our software does not mean you should do it all the time!
  - For other radio telescopes you get an entire package which you can hardly change at all – there you simply accept things as they are.
  - Structural, functional, and philosophical changes should be limited as much as possible (since it always leads to problems and interoperabilities). Please discuss them with me first.
  - Think ten times before changing I/O interfaces or introduce new ones. It will always mean that other people's software won't work or needs to be upgraded to work with your new "much better" interface – in fact it might just have wasted many month of other people's work. If possible update or integrate into existing interfaces.

⇒ You must have very good arguments to NOT use what your colleagues (or your boss) has already provided.

⇒ When you write new things make sure, they are in line with the overall philosophy and they are modular enough for future use. That investment is worth the time.

Conclusion

- Understand that you are responsible not only for the success of your own part, but also for the sum of all parts
  - Make some effort to understand what others have already provided. Ask, test, complain before you rewrite it yourself. This will help everyone. Rewriting always takes longer than you think!
- Work result oriented:
  - Have a clear scientific goal/task as first priority
  - Have an end-to-end pilot ready and exercise it regularly to work with the data and see if things go wrong
- Document your work and communicate it to the others
  - Every function/task comes with an example on how to use
  - Update the tutorial
- Understand to and adhere to basic software plan and philosophy and accept some basic chain of command
  - Find the right balance between self-empowered programmers and a benevolent dictatorship.
  - Discuss functional/structural changes with an experienced group leader (i.e., me) and ask for help

Building a Cathedral

- Many highly qualified artisans try to build a magnificent construction
- Every builder works fairly independently on his own place and produces, e.g., original statutes
- Still they follow the overall architecture (and the architect) and they adhere to common interfaces.

Random Notes

- Add useful web pages to http://www.astro.ru.nl/wiki/research/cr-tools/cr-tools_startpage
- Let's educate each other at such meetings (e.g., give how-to-presentations).