Go-HEP: what can Go do for science?

Sébastien Binet
CNRS/IN2P3
@0xbins
binet@cern.ch

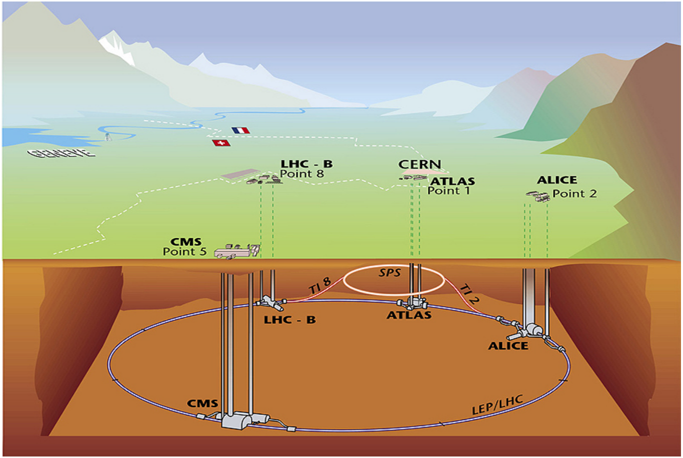
2019-10-22

Field of physics which studies the fundamental laws of Nature and the properties of the constituents of matter.

Many labs working on HEP around the world. But, perhaps one of the most famous ones is CERN.

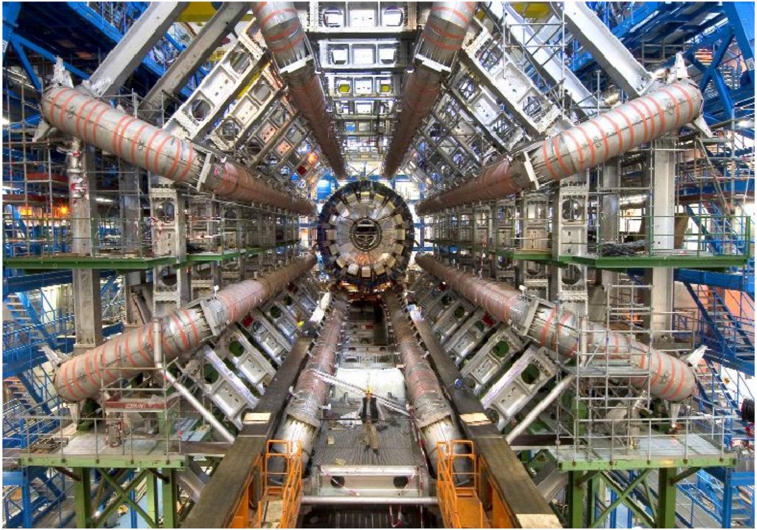
Figure: Aerial view of CERN & LHC

CERN-LHC



The Large Hadron Collider (LHC).
A proton-proton collider of 27km of circumference.

ATLAS installation



(a brief) History of software in HEP

50's-90's: FORTRAN77

```fortran
c      == hello.f ==
       program main
       implicit none
       write ( *, '(a)' ) 'Hello from FORTRAN'
       stop
       end
```

```
$ gfortran -c hello.f && gfortran -o hello hello.o
$ ./hello
Hello from FORTRAN
```

- `FORTRAN77` is the **king**
- 1964: **CERNLIB**
- REAP (paper tape measurements), THRESH (geometry reconstruction)
- SUMX, **HBOOK** (statistical analysis chain)
- ZEBRA (memory management, I/O, ...)
- GEANT3, **PAW**

90's-...: C++

```cpp
#include <iostream>
int main(int, char **) {
  std::cout << "Hello from C++" << std::endl;
  return EXIT_SUCCESS;
}
```

```
$ c++ -o hello hello.cxx && ./hello
Hello from C++
```

- object-oriented programming (OOP) is the cool kid on the block
- **ROOT**, POOL, LHC++, AIDA, **Geant4**
- `C++` takes roots in HEP

00's-...: python

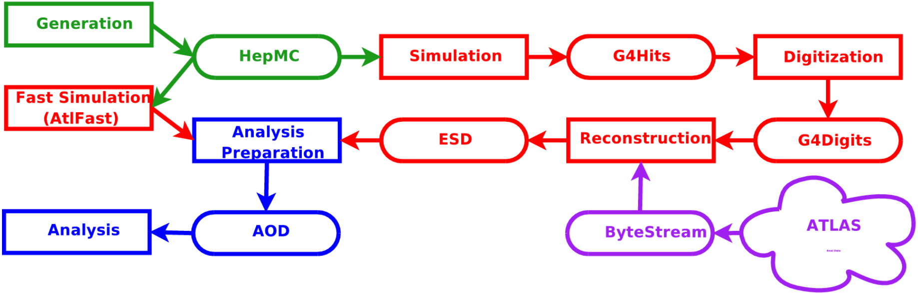
```python
print "Hello from python"
```

```
$ python ./hello.py
Hello from python
```

- `python` becomes the *de facto* scripting language in HEP
- framework data-cards
- analysis glue, (whole) analyses in `python`
- **PyROOT**, rootpy
- numpy, scipy, matplotlib, **IPython/Jupyter**

Current software in a nutshell

- **Generators**: generation of true particles from fondamental physics first principles
- **Full Simulation**: tracking of all stable particles in magnetic field through the detector simulating interaction, recording energy deposition (**CPU intensive**)
- **Reconstruction**: from real data, or from `Monte-Carlo` simulation data as above
- **Fast Simulation**: parametric simulation, faster, coarser
- **Analysis**: daily work of physicists, running on output of reconstruction to derive analysis specific information (**I/O intensive**)
- everything in the same `C++` offline control framework (except analysis)



Software in HEP - some numbers

An LHC experiment (*e.g.* `ATLAS`, `CMS`) is about ∼3000 physicists large but only a fraction of those is developing code.
Reconstruction frameworks grew from ∼3M SLOC to ∼5M

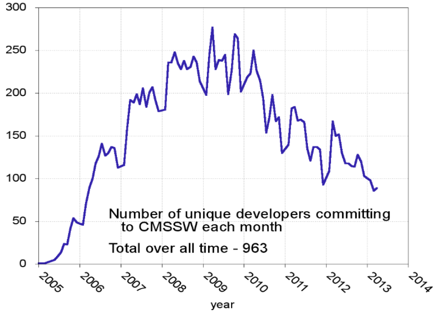
Summing over all HEP software stack for *e.g.* ATLAS:

- event generators: ∼1.4M SLOC (C++, FORTRAN-77)
- I/O libraries ∼1.7M SLOC (C++)
- simulation libraries ∼1.2M SLOC (C++)
- reconstruction framework ∼5M SLOC (C++) + steering/configuration (∼1M SLOC python) (want to have a look at the `ATLAS code`? `CMS code`?)

GCC: ∼7M SLOC
Linux kernel 3.6: 15.9M SLOC

People committing code to VCS per month

- Variety of skills
- Huge turn-around
- Once the physics data is pouring, people go doing physics instead of software



See also "The Life Cycle of HEP Offline Software", P.Elmer, L. Sexton-Kennedy, C.Jones, CHEP 2007

Software developers

∼300 active developers (per experiment)
∼1000 different developers integrated over the lifetime of a single LHC experiment.

- few "real" s/w experts
- some physicists with strong skill set in s/w
- many with some experience in s/w development
- some with **no** experience in s/w development

A multi-timezone environment

- Europe, America, Asia, Australia

Many communities (core s/w people, generators, simulation, ...)

Development and infrastructures usually CERN-centric
Heavily Linux based (`Scientific Linux CERN`, `CERN CentOS`)

Software development cycle

VCS (CVS, then SVN. GIT: getting there.)

Nightlies (Jenkins, Travis or homegrown solution)

- need a sizeable cluster of build machines (distcc, ccache, ...)
- builds the framework stack in $\sim$8h
- produces $\sim$2000 shared libraries
- installs them on AFS (also creates RPMs and tarballs)

Devs can then test and develop off the nightly *via* AFS

Every 6 months or so a new production release is cut, validated (then patched) and deployed on the World Wide LHC Computing Grid (`WLCG`).

Release size: $\sim$**5Gb**

- binaries, libraries (externals+framework stack)
- extra data (sqlite files, physics processes' modelisation data, ...)

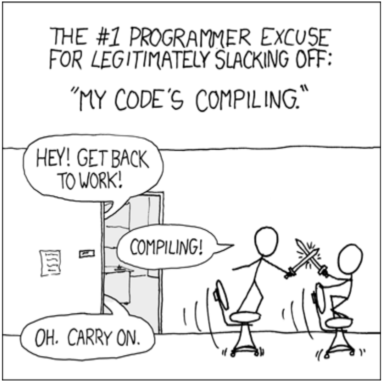
Software runtime ?

Big science, big data, big software, big numbers.

- $\sim$1min to initialize the application
- loading $>$500 shared libraries
- connecting to databases (detector description, geometry, ...)
- instantiating $\sim$2000 `C++` components (steered from a `Python` script)
- 2Gb/4Gb memory footprint per process

(obligatory xkcd reference)

- `C++`: **slow** (very slow?) to compile/develop, **fast** to execute
- `python`: **fast** development cycle (no compilation), **slow** to execute



We learn to love hating our framework. *(every step of the way)*

And even more so in the future:

- work to make our software stack thread-safe
- or at least parts of it multithread friendly to harness multicore machines
- quite a lot of fun ahead

Other software engineering problems

- scalability (development, teaching, maintenance, build)
- installation of dependencies
- code deployment
- code robustness
- code readability
- multicores/manycores, multithreading
- distributed programming
- etc...

talks.golang.org/2012/splash.slide
talks.golang.org/2012/splash.article

Are those our only options ?

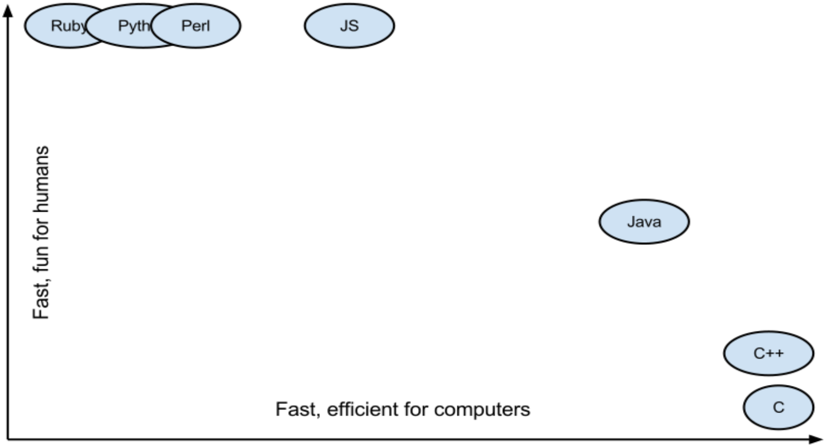


Figure: Credits: B. Fitzpatrick

Remember Go ?

- compiles quickly (no warnings, imports)
- enforces coherent coding rules (across projects)
- builtin test/benchmark/documentation facilities
- deploys easily, cross-compiles easily
- installs easily (also 3rd-party packages: `go get`)
- fast to pick up, not as complicated as **C++**
- builtin reflection system
- builtin (de)serialization capabilities
- concurrency support
- garbage collected

Perfect match for many HEP use cases.

Migrating to Go ? (evil plan for (HEP) world domination)

Migrating ∼5M SLOC of C++ code to Go, during data taking, **unfortunately**, won't fly.

Creating new applications for data massaging or post-processing **might**.
Creating a new concurrent and parallel framework for the next accelerator **might**.

Need to build a critical mass of Go HEP enthusiasts

Go-HEP

What can Go bring to science and HEP?

- a **simple** language anybody can learn in days, proficient in a couple of months
- **standard** tool to handle **dependencies** (across OSes, architectures)
- **fast** edit-compile-run development cycle
- **simple** deployment (on clusters, laptops, ...), **easiest cross-compilation** system to date
- fast at runtime, builtin tools for profiling (CPU, mem, flamegraph, tracing)
- support for **concurrency** programming and multi-core machines
- no magic, no voodoo, **reduce** disconnect b/w HEP sw experts and HEP users
- **easy** code sharing: between packages, between experiments, between scientists
- easier **reproducibility** and cross-check tests
- **refactoring** tools & linters, builtin/standard testing tools

`sbinet.github.io/posts/2018-07-31-go-hep-manifesto`

Go for HEP: challenges

`$EXPERIMENT` is taking data:

- one can't really migrate (monolithic) MLoC software stacks while taking data, neither during a long shutdown

Convincing physicists:

- need to prove `Go` is useful, pleasant to use and viable
- need to prove one can carry an analysis in `Go`, faster than by other means
- provide 1 or 2 "poster child" applications

Need to implement:

- histograms, PDFs, plotting, fits, BLAS/LAPACK, I/O
- (some level of) inter-operability with `C++/ROOT`

$\Rightarrow$ `go-hep.org` is the beginning of such an endeavour

https://go-hep.org

go-hep

go-hep

News

Getting started

Tutorials

Documentation

Contributing

License

Users of Go-HEP

Go HEP

- Motivation
- Installation
- License
- Authors and Contributors
- Support or Contact
- About

Welcome to Go-HEP

`Go-HEP` is a set of libraries and applications allowing physicists from High Energy Physicis (HEP) to write efficient analysis code in the Go programming language.

release v0.19.0 | build passing | build passing | codecov 63% | godoc reference | License BSD-3 | DOI 10.5281/zenodo.597940 | JOSS 10.21105/joss.00372 | launch binder

Go brings the fast edit-compile-run cycle that interpreted language users know and the runtime efficiency that compiled languages users expect. Go-HEP provides the needed `HEP` oriented packages on top of this concurrency-enabled language.

`Go-HEP` currently sports the following packages:

- go-hep.org/x/hep/brio: a toolkit to generate serialization code
- go-hep.org/x/hep/fads: a fast detector simulation toolkit
- go-hep.org/x/hep/fastjet: a jet clustering algorithms package (WIP)
- go-hep.org/x/hep/fit: a fitting function toolkit (WIP)
- go-hep.org/x/hep/fmom: a 4-vectors library
- go-hep.org/x/hep/fwk: a concurrency-enabled framework
- go-hep.org/x/hep/groot: a pure Go package for ROOT I/O (WIP)
- go-hep.org/x/hep/hbook: histograms and n-tuples (WIP)
- go-hep.org/x/hep/hplot: interactive plotting (WIP)
- go-hep.org/x/hep/hepmc: `HepMC` in pure Go (EDM + I/O)
- go-hep.org/x/hep/hepevt: `HEPEVT` bindings
- go-hep.org/x/hep/heppdt: `HEP` particle data table
- go-hep.org/x/hep/lcio: read/write support for `LCIO` event data model
- go-hep.org/x/hep/lhef: Les Houches Event File format
- go-hep.org/x/hep/rio: `go-hep` record oriented I/O
- go-hep.org/x/hep/sio: basic, low-level, serial I/O used by `LCIO`
- go-hep.org/x/hep/slha: `SUSY` Les Houches Accord I/O
- go-hep.org/x/hep/xrootd: XRootD client in pure Go
- go-hep.org/x/cgo/croot: bindings to a subset of ROOT I/O

Motivation

Writing analyses in HEP involves many steps and one needs a few tools to successfully carry out such an endeavour. But – at minima – one needs to be able to read (and possibly write) ROOT files to be able to interoperate with the rest of the HEP community or to insert one's work into an already existing analysis pipeline.

Go-HEP provides this necessary interoperability layer, in the Go programming language. This allows physicists to leverage the great concurrency primitives of Go, together with the surrounding tooling and software engineering ecosystem of Go, to implement physics

Available and tested (TravisCI, AppVeyor) on Windows, Linux, Darwin, ...
AMD64, 386, ARM, ARM64, ...

- `go-hep.org/x/hep/brio`: a toolkit to generate serialization code
- `go-hep.org/x/hep/fads`: a fast detector simulation toolkit
- `go-hep.org/x/hep/fastjet`: a jet clustering algorithms package (WIP)
- `go-hep.org/x/hep/fit`: a fitting function toolkit (WIP)
- `go-hep.org/x/hep/fmom`: a 4-vectors library
- `go-hep.org/x/hep/fwk`: a concurrency-enabled control framework
- `go-hep.org/x/hep/hbook`: histograms and n-tuples (WIP)
- `go-hep.org/x/hep/hplot`: interactive plotting (WIP)
- *[...]*
- `go-hep.org/x/hep/groot`: a pure `Go` package to for `ROOT` I/O
- `go-hep.org/x/hep/xrootd`: `XRootD` client in pure `Go`

DOI:10.5281 (Zenodo:597940)
JOSS Paper

2 work areas to demonstrate 'Go''s applicability for HEP use cases have been identified:

- data acquisition (DAQ), monitoring, control command
- detector fast simulation toolkit (à la Delphes (C++))

Go-HEP - fast-simulation & analysis

fads

`fads` is a "FAst Detector Simulation" toolkit.

- morally a translation of `C++-Delphes` into Go
- uses `hep/fwk` to expose, manage and harness concurrency into the usual `HEP` event loop (`initialize` | `process-events` | `finalize`)
- uses `hep/hbook` for histogramming, `hep/hepmc` for `HepMC` input/output

Code is on github (BSD-3):
`go-hep.org/x/hep/fwk`
`go-hep.org/x/hep/fads`

Documentation is served by `godoc.org`:
`godoc.org/go-hep.org/x/hep/fwk`
`godoc.org/go-hep.org/x/hep/fads`

go-hep/fwk

`fwk` is a Go-based concurrent control framework inspired from:

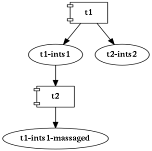
- GaudiHive
- ILC Marlin
- CMSSW
- previous incarnations of *fwk* (*go-ng-gaudi*, *go-gaudi*)

```
$ fwk-ex-tuto-1 -help
Usage: fwk-ex-tuto1 [options]

ex:
 $ fwk-ex-tuto-1 -l=INFO -evtmax=-1

options:
  -evtmax=10: number of events to process
  -l="INFO": message level (DEBUG|INFO|WARN|ERROR)
  -nprocs=0: number of events to process concurrently
```

Runs 2 tasks.



```
$ fwk-ex-tuto-1
::: fwk-ex-tuto-1...
t2                   INFO configure...
t2                   INFO configure... [done]
t1                   INFO configure ...
t1                   INFO configure ... [done]
t2                   INFO start...
t1                   INFO start...
app                  INFO >>> running evt=0...
t1                   INFO proc... (id=0|0) => [10, 20]
t2                   INFO proc... (id=0|0) => [10 -> 100]
[...]
app                  INFO >>> running evt=9...
t1                   INFO proc... (id=9|0) => [10, 20]
t2                   INFO proc... (id=9|0) => [10 -> 100]
t2                   INFO stop...
t1                   INFO stop...
app                  INFO cpu: 654.064us
app                  INFO mem: alloc:             62 kB
app                  INFO mem: tot-alloc:         74 kB
::: fwk-ex-tuto-1... [done] (cpu=788.578us)
```

go-hep/fwk - Components

A `fwk` application consists of a set of components (`fwk.Task`) which are:

- (optionally) configured
- started
- given the chance to process each event
- stopped

Helper components (`fwk.Svc`) can provide additional features (such as a whiteboard/event-store service, a data-flow service, ...) but do not typically take (directly) part of the event processing.

```go
// Component is the interface satisfied by all values in fwk.
//
// A component can be asked for:
// its Type() (ex: "go-hep.org/x/hep/fads.MomentumSmearing")
// its Name() (ex: "MuonsMomSmearing")
type Component interface {
        Type() string
        Name() string
}
```

`Tasks` (and `Services`) are called with a `Context` argument to enable concurrency/parallelism.

```go
// Task is a component processing event-level data.
// Task.Process is called for every component and for every input event.
type Task interface {
        Component

        StartTask(ctx Context) error
        Process(ctx Context) error
        StopTask(ctx Context) error
}
```

go-hep/fwk - Interfaces

```go
// Context is the interface to access context-local data.
type Context interface {
        ID() int64         // id of this context (e.g. entry/event number)
        Slot() int         // slot number in the pool of event sequences
        Store() Store      // data store corresponding to the id+slot
        Msg() MsgStream // messaging for this context (id+slot)

        // Svc retrieves an already existing Svc by name.
        Svc(n string) (Svc, error)
}
```

`Context` is a bit of a grab bag of what needs to be available/queried during event processing.

- `Msg()` allows to relieve pressure on the I/O system. Eventually, should allow to have human-readable log files even with many events in-flight.
- similarly, `Store()` and `Svc()` allow to have event-level local state.

go-hep/fwk - Interfaces

```go
// DeclPorter is the interface to declare in/out ports for the data flow.
type DeclPorter interface {
	DeclInPort(name string, t reflect.Type) error
	DeclOutPort(name string, t reflect.Type) error
}
```

- Note there is no *update* nor *R/W* ports: simplifies the data flow, make it more **functional-like**,
- Updates handled by copying input data under a new event store key,
- The `dflowsvc` detects (long-range) cycles and missing data-nodes.

```go
func (tsk *task1) Configure(ctx fwk.Context) error {
	err := tsk.DeclOutPort(tsk.i1prop, reflect.TypeOf(int64(0)))
	if err != nil {
		return xerrors.Errorf(
			"could not declare output port: %w", err,
		)
	}
	// ...
	return err
}
```

go-hep/fwk - Interfaces

```go
// Store provides access to a goroutine-safe map[string]interface{} store.
type Store interface {
	Get(key string) (interface{}, error)
	Put(key string, value interface{}) error
	Has(key string) bool
}
```

Examples:

```go
func (tsk *task2) Process(ctx fwk.Context) error {
	store := ctx.Store()
	// blocks until data for this event/slot is available
	v, err := store.Get(tsk.input)
	if err != nil {
		return err
	}
	i := v.(int64)
	o := tsk.fct(i)
	err = store.Put(tsk.output, o)
	return err
}
```

go-hep/fwk - appmgr

```go
func (app *appmgr) run(ctx Context) error {
	var err error
	defer app.msg.flush()
	app.state = fsm.Running

	switch app.nprocs {
	case 0:
		err = app.runSequential(ctx)
	default:
		err = app.runConcurrent(ctx)
	}

	return err
}
```

- run sequentially
- run *N* workers, each worker processing events as they become available
- all tasks are started at the beginning of the event processing, letting the dataflow works its magic

go-hep/fwk - workers

```go
type worker struct {
	slot  int
	keys  []string // nodes in data-flow (Input/Output)
	store datastore
	ctxs  []context // a Context for each component
	msg   msgstream

	evts <-chan int64    // channel of event indices
	quit <-chan struct{} // channel to notify early-abort
	done chan<- struct{} // channel to notify we are done
	errc chan<- error    // channel of errors during event processing
}
```

- each worker manages its own event store
- each worker manages contexts for each component it runs

`fwk` enables:

- event-level concurrency
- tasks-level concurrency

`fwk` relies on `Go`'s runtime to properly schedule *goroutines*.

For sub-task concurrency, users are by construction required to use `Go`'s constructs (*goroutines* and *channels*) so everything is consistent **and** the *runtime* has the **complete picture**.

go-hep/fwk - configuration & steering

- use regular `Go` to configure and steer.
- still on the fence on a DSL-based configuration language (`HCL`, `Toml`, ...)
- probably **not** `Python` though

```go
// job is the scripting interface to 'fwk'
import "go-hep.org/x/hep/fwk/job"

func main() {
	// create a default fwk application, with some properties
	app := job.New(job.P{
		"EvtMax": 10,
		"NProcs": 2,
	})

	// ... cont'd on next page...
```

```go
// create a task that reads integers from some location
// and publish the square of these integers under some other location
app.Create(job.C{
	Type: "go-hep.org/x/hep/fwk/testdata.task2",
	Name: "t2",
	Props: job.P{
		"Input":  "t1-ints1",
		"Output": "t1-ints1-massaged",
	},
})
// create a task that publish integers to some location(s)
app.Create(job.C{
	Type: "go-hep.org/x/hep/fwk/testdata.task1",
	Name: "t1",
	Props: job.P{
		"Ints1": "t1-ints1",
		"Ints2": "t2-ints2",
		"Int1":  int64(10), // initial value for the Ints1
		"Int2":  int64(20), // initial value for the Ints2
	},
})
app.Run()
```

go-hep/fads - real world use case

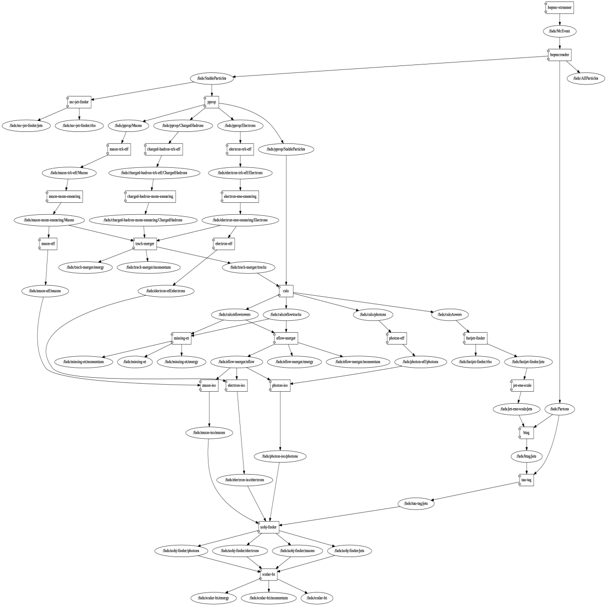
- translated `C++-Delphes`' ATLAS data-card into Go
- `go-hep/fads/cmd/fads-app`
- installation:

```
$ go get go-hep.org/x/hep/fads/cmd/fads-app
$ fads-app -help
Usage: fads-app [options] <hepmc-input-file>

ex:
 $ fads-app -l=INFO -evtmax=-1 ./testdata/hepmc.data

options:
  -cpu-prof=false: enable CPU profiling
  -evtmax=-1: number of events to process
  -l="INFO": log level (DEBUG|INFO|WARN|ERROR)
  -nprocs=0: number of concurrent events to process
```

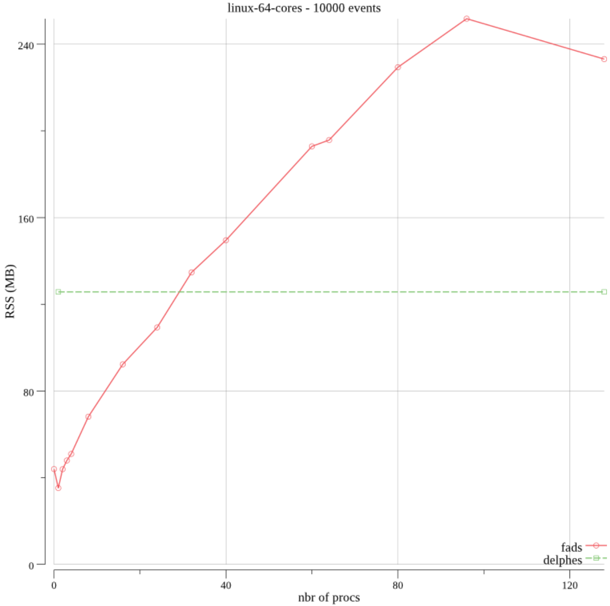
- a `HepMC` converter
- particle propagator
- calorimeter simulator
- energy rescaler, momentum smearer
- isolation
- b-tagging, tau-tagging
- jet-finder (reimplementation of FastJet in Go: `go-hep/fastjet`)
- histogram service (from `go-hep/fwk`)

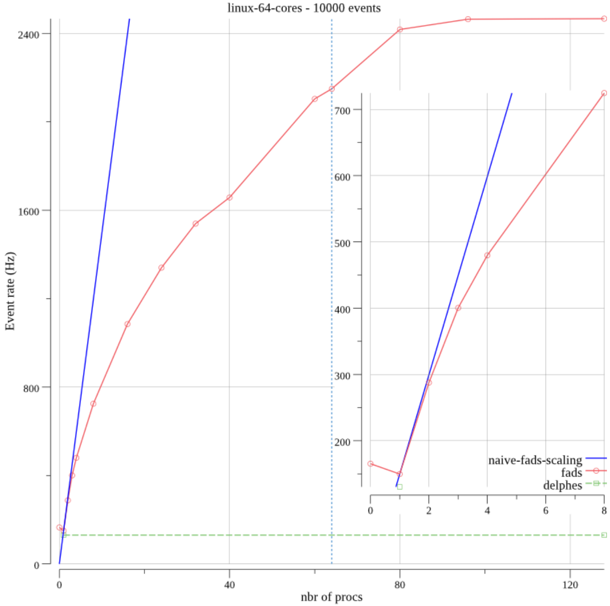

hepmc-streamer
/fads/McEvent
hepmcreader
/fads/StableParticles
/fads/AllParticles
mc-jet-finder
pprop
/fads/mc-jet-finder/jets
/fads/mc-jet-finder/rho
/fads/pprop/Muons
/fads/pprop/ChargedHadrons
/fads/pprop/Electrons
/fads/pprop/StableParticles
muon-trk-eff
charged-hadron-trk-eff
electron-trk-eff
/fads/muon-trk-eff/Muons
/fads/charged-hadron-trk-eff/ChargedHadrons
/fads/electron-trk-eff/Electrons
muon-mom-smearing
charged-hadron-mom-smearing
electron-ene-smearing
/fads/muon-mom-smearing/Muons
/fads/charged-hadron-mom-smearing/ChargedHadrons
/fads/electron-ene-smearing/Electrons
muon-eff
track-merger
electron-eff
/fads/track-merger/energy
/fads/track-merger/momentum
/fads/track-merger/tracks
/fads/muon-eff/muons
/fads/electron-eff/electrons
calo
/fads/calo/eflowtowers
/fads/calo/eflowtracks
/fads/calo/photons
/fads/calo/towers
missing-et
eflow-merger
photon-eff
fastjet-finder
/fads/missing-et/momentum
/fads/missing-et
/fads/missing-et/energy
/fads/eflow-merger/eflow
/fads/eflow-merger/energy
/fads/eflow-merger/momentum
/fads/photon-eff/photons
/fads/fastjet-finder/rho
/fads/fastjet-finder/jets
muon-iso
electron-iso
photon-iso
jet-ene-scale
/fads/jet-ene-scale/jets
/fads/Partons
/fads/muon-iso/muons
btag
/fads/photon-iso/photons
/fads/btag/jets
/fads/electron-iso/electrons
tau-tag
/fads/tau-tag/jets
uobj-finder
/fads/uobj-finder/photons
/fads/uobj-finder/electrons
/fads/uobj-finder/muons
/fads/uobj-finder/jets
scalar-ht
/fads/scalar-ht/energy
/fads/scalar-ht/momentum
/fads/scalar-ht

Results - testbenches

- Linux: Intel(R) Xeon(R) CPU E5-4620 0 @ 2.20GHz, 64 cores, 128Gb RAM
- Delphes, 3.0.12, gcc4.8
- fads, Go-1.9

```
$> time delphes ./input.hepmc
$> time fads-app ./input.hepmc
```


linux-64-cores - 10000 events
RSS (MB)
240
160
80
0
0
40
80
120
nbr of procs
fads
delphes



linux-64-cores - 10000 events
Event rate (Hz)
nbr of procs
2400
1600
800
0
40
80
120
700
600
500
400
300
200
2
4
6
8
naive-fads-scaling
fads
delphes

fads: Results & Conclusions

- good RSS scaling
- good CPU scaling
- bit-by-bit matching physics results *wrt* `Delphes`
- no need to merge output files, less chaotic I/O, less I/O wait

Rivet & fads

Rivet

The `Rivet` toolkit (Robust Independent Validation of Experiment and Theory) is a system for validation of Monte Carlo event generators. It provides a large (and ever growing) set of experimental analyses useful for MC generator development, validation, and tuning, as well as a convenient infrastructure for adding your own analyses.

```
$> repeat 10 'time rivet --analysis=MC_GENERIC -q  ./Z-hadronic-LEP.hepmc'
real=13.32 user=12.97 sys=0.33 CPU=99% MaxRSS=26292
real=13.31 user=12.93 sys=0.37 CPU=99% MaxRSS=26356
real=13.29 user=12.93 sys=0.35 CPU=99% MaxRSS=26440
real=13.31 user=12.95 sys=0.35 CPU=99% MaxRSS=26356
real=13.29 user=13.01 sys=0.27 CPU=99% MaxRSS=26280
real=13.31 user=12.97 sys=0.32 CPU=99% MaxRSS=26328
real=13.35 user=12.93 sys=0.41 CPU=99% MaxRSS=26276
real=13.30 user=12.96 sys=0.33 CPU=99% MaxRSS=26624
real=13.30 user=12.93 sys=0.36 CPU=99% MaxRSS=26440
real=13.35 user=12.98 sys=0.36 CPU=99% MaxRSS=26484
```

fads-rivet-mc-generic

Reimplementation on top of go-hep/fwk+fads of the MC_GENERIC analysis.
Bit-to-bit identical results.

```
$> go get go-hep.org/x/hep/fads/cmd/fads-rivet-mc-generic

$> repeat 10 'time fads-rivet-mc-generic -nprocs=1 ./Z-hadronic-LEP.hepmc'
real=6.04 user=5.66 sys=0.12 CPU= 95% MaxRSS=23384
real=5.70 user=5.62 sys=0.09 CPU=100% MaxRSS=21128
real=5.71 user=5.58 sys=0.11 CPU= 99% MaxRSS=22208
real=5.68 user=5.60 sys=0.08 CPU=100% MaxRSS=23156
real=5.71 user=5.63 sys=0.08 CPU=100% MaxRSS=20672
real=5.78 user=5.62 sys=0.09 CPU= 98% MaxRSS=22328
real=5.67 user=5.62 sys=0.05 CPU=100% MaxRSS=20968
real=5.68 user=5.57 sys=0.07 CPU= 99% MaxRSS=23748
real=5.70 user=5.60 sys=0.10 CPU=100% MaxRSS=21360
real=5.72 user=5.65 sys=0.07 CPU=100% MaxRSS=22764
```

About 2x as fast, using a bit less memory.

Go in science

Science-y packages

Even if `Go` is relatively new, support for general purpose scientific libraries is there and growing, thanks to the `Gonum.org` community:

- `gonum/blas`, a `Go` based implementation of Basic Linear Algebra Subprograms
- `gonum/lapack`, a lapack implementation for `Go`
- `gonum/mat`, to work with matrices
- `gonum/graph`, to work with graphs
- `gonum/optimize`, for finding the optimum value of functions
- `gonum/integrate`, provides routines for numerical integration
- `gonum/diff`, for computing derivatives of a function
- `gonum/stat`, for statistics and distributions
- `gonum/plot`, to create publication quality plots (most of the plots seen earlier are made w/ `gonum/plot`)
- ...

I/O

I/O support for some formats:

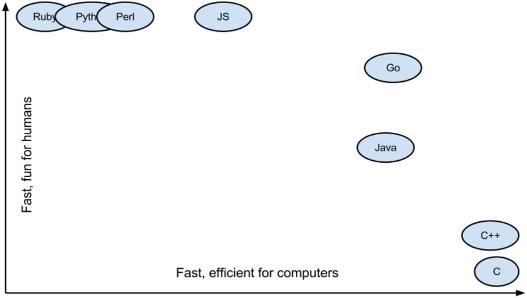
- `sbinet/npyio`: read/write support for `NumPy data files`
- `ready-steady/mat`, `sbinet/matfio`: r/w support for `MATLAB files`
- `gonum/hdf5`: access to `HDF5` (using cgo)
- `go-arrow`: access to `Apache Arrow` data and IPC protocol

Go for Data Science

A data science community is gathering around `gopherdata.io`.

- `gopherdata/gophernotes`, a `Jupyter` kernel for `Go`
- `gopherdata/mybinder-go`, a web-based Jupyter kernel for `Go`
- `gopherdata/resources`: many resources for machine learning, classifiers, neural networks, ...

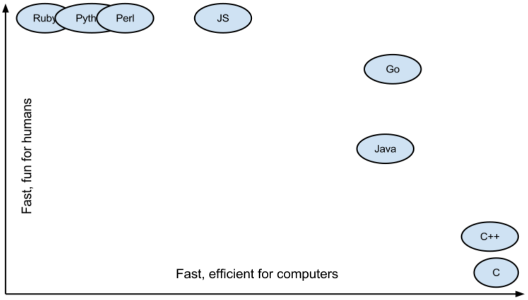
Go is great at writing small and large (concurrent) programs. Also true for **science-y** programs, even if the amount of libraries can still be improved.



Write your next tool/analysis/simulation/software in `Go`? and `Go-HEP` or `astrogo`, or `Biogo`, or `Gonum`, or ...

Conclusions

Go is great at writing small and large (concurrent) programs. Also true for **science-y** programs, even if the amount of libraries can still be improved.




```
Software engineering is what happens to programming
when you add time and other programmers.
(Russ Cox)
```

(Also true for science-y programs and scientists)

Thank you

Sébastien Binet
CNRS/IN2P3/LPC-Clermont
github.com/sbinet
@0xbins
binet@cern.ch