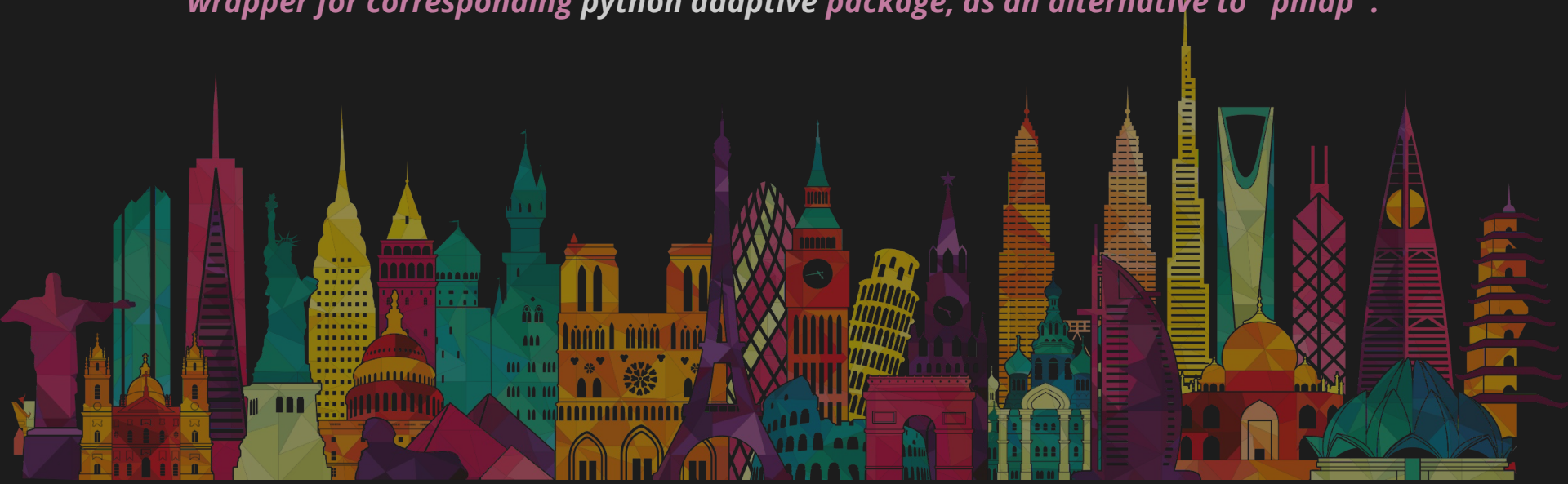
How to not lose a mind by paralelizing a feedback loop?

Feedback loops are notoriously hard to reason and debug when parallelism is introduced. Thus in this poster, I will describe abstractions introduced in **TaskMaster** *package for concurrent feedback loops which one can replay to debug, and I will demonstrate how they can be used together with* **Adaptive**, *a wrapper for corresponding* **python adaptive** *package, as an alternative to `pmap`.*



Janis Erdmanis (GitHub: akels, Twitter: graphitewriter, janiserdmanis.org)

An idea worth exploring

Motivation: Often when we face an embarrassingly parallelizable problem which runs a long time, we reach out for a single `pmap`, and if we are lucky, we can run that on the cluster with plenty of cores. However, often we are faced with resources in front of us. In such a case, it is worth to think whether an evenly spaced grid is the most optimal one.

Choosing an optimal grid for the function in advance is often a problem itself more significant than waiting a little extra time. Instead, it would be great if the grid itself would adjust as more knowledge of function comes in during evaluation. Which thus forms a feedback loop.

*Introducing parallelism to a feedback loop in a way that it would allow deterministic debugging while not staling resources is tricky. Additionally, the feedback loop should be completely independent of the hardware it is executed (CPU, GPU, Cluster jobs, etc.) and preferably sit in a separate package. The question I found fascinating was what would be the best abstractions to such problem in Julia, which gave rise to the **TaskMaster** package.*

*Background: I started this project as a wrapper to **python adaptive** package which my colleagues were making. It quickly became apparent that it would be dull to let python deal with parallelism. Also, I found precisely those abstractions in **python adaptive** package unsettling, which could be great to approach from Julia perspective.*

Exploring how to solve this problem in Julia was attractive because:

- *Very natural abstractions for parallelism.*
- *Channel makes for easy piping.*
- *Type system and multiple dispatch.*
- *Performance. If one has contained resources, better make sure that f(x) runs as fast as possible!*

First let's abstract a learning strategy

Learner API

Abstraction for feedback loop is done similarly as in python adaptive package. First one needs a mutable type `Learner` which contains parameters and the state of evaluation. For it, one needs two methods `ask!` for getting the point and `tell!` for updating the state and thus the grid of the `Learner`, which are expected to be deterministic.

```
using Adaptive
learner = AdaptiveLearner1D((0,1))
```

which defines interval `(0,1)` for the function sampling. Now the `learner` can be used to `ask!` points:

```
xi =  ask!(learner, si)
```

Where `si` contains an external data passed to the Learner, for example, some random number. For Adaptive package, one passes value `si=true`.

The last step is to evaluate the function and feed it back to the `learner`:

```
yi = f(xi)
tell!(learner, yi)
```

Sequential Driving: *When execution is sequential one can write full execution as a single while loop:*

```
while !(learner.loss() < step)
        xi = ask!(learner,true)
        yi = f(xi)
        tell!(learner,(xi,yi))
end
```

where `step` is just a convergence parameter specific to a particular learner.

Concurrent Driving: *Parallelism, on the other hand, is harder due to race conditions. For simplicity, let's assume that we have a process which takes values from tasks Channel and puts results in a form `(xi,f(xi))` in the results Channel:*

```
unresolved = 0
while true
    if !(learner.loss() < step)

        xi = ask!(learner,true)
        put!(tasks,xi)
        unresolved += 1

        if unresolved < N
            continue
        end
    end

    if unresolved == 0
        break
    end

    yi = take!(results)
    tell!(learner,yi)
    unresolved -= 1
end
```

TaskMaster

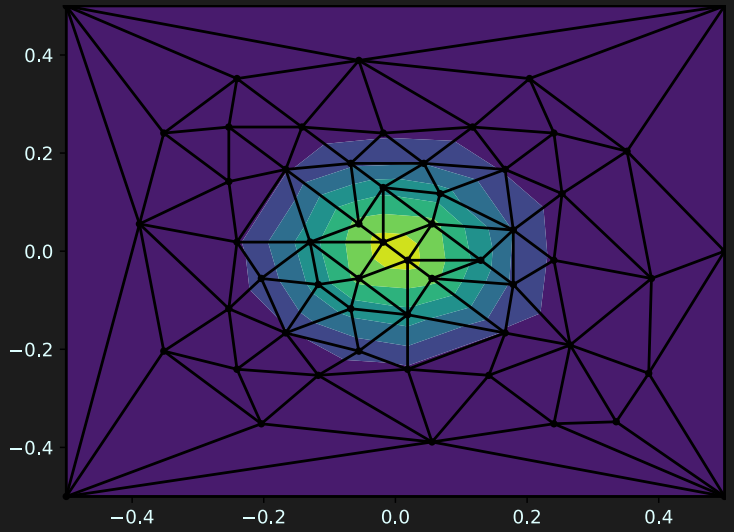
Abstracting Loop

Writing that while loop accurately counting `unresolved` tasks can be a little challenging for every time one would like to make a figure. Additionally, to use the loop one needs to set up evaluation `Task` between tasks and results `Channel`.

```
using Adaptive
using TaskMaster

@everywhere f(p) = exp(-p[1]^2 - p[2]^2)

master = WorkMaster(f)
learner2d = AdaptiveLearner2D([(-3,+3),(-3,+3)])
loop = Loop(master,learner2d)
output = evaluate!(loop,learner->learner.loss()<0.05)
```



Here `WorkMaster` sets up processes for evaluating `f(x)` from input to output `Channel`. `Loop` defines what is to be evaluated with which `Learner` and `evaluate!` is running the loop until specified convergence condition.

Learner Replay

Let's imagine a situation where you had spent hours evaluating the function with a `Learner`. For some particular reason looking at the output, the `Learner` seems had misbehaved. The question then is how one could debug that?

Through replaying the master as long as `Learner` is deterministic:*

```
hmaster = HistoryMaster(output,length(master.slaves))
hlearner = AdaptiveLearner2D([(-3,+3),(-3,+3)])
hloop = Loop(hmaster,hlearner,loop->println("Learner state
$(hloop.hlearner.state)"))
evaluate!(hloop,learner->learner.loss()<0.05)
```

Thus one can understand the `Learner` by knowing what state caused the problems.

** Adaptive learners uses random numbers and thus the replay would not work. It would be possible if random numbers or it's seeds would be passed with `ask!` method.*

Conclusions and References

Conclusions

- *Feedback loop only needs `ask!` and `tell!` methods. Those methods need to be deterministic if one wants easy debugging of a feedback loop in a concurrent context.*
- *Abstractions Julia offers by default almost makes concurrent executor part redundant (**TaskMaster**) in contrast to Python (Runner module).*
- *The costs of using **Adaptive** package are quite high when all python dependencies get installed. Also, it is in active development; thus, the wrapper might break after a while.*
- *Plotting of the output is cumbersome for two, and larger dimensions as topology need to be constructed from output or extracted from the `Learner`.*

Future Work

- *Implement function executor, which uses Julia threads.*
- *Port some learners from the python adaptive package.*
- *A more concise API, for example:*

```
evaluate!(f::Function, learner::AbstractLearner,
cond::Function)
```

- *What is the best way to interact with plotting packages?*

References

- *janiserdmanis.org/TaskMaster.jl/dev*
- *github.com/python-adaptive/adaptive*
- *https://github.com/akels/Adaptive.jl*
- *Bas Nijholt and Joseph Weston and Jorn Hoofwijk and Anton Akhmerov (2019). Adaptive: parallel active learning of mathematical functions. Zenodo*