Redesign, Debugging, and Optimization

Christopher R. Genovese

Department of Statistics & Data Science

Thu 25 Sep 2025
Session #10

Plan

Revision and Redesign: A Case Study

Plan

Revision and Redesign: A Case Study

Building State Machines

Plan

Revision and Redesign: A Case Study

Building State Machines

Debugging

Plan

Revision and Redesign: A Case Study

Building State Machines

Debugging

Optimization

Plan

Revision and Redesign: A Case Study

Building State Machines

Debugging

Optimization

Profiling

Plan

Announcements

- Lightning Round
- **Homework**:
  - **laser-tag** assignment due Tue 30 Sep. Available on github problem bank.
  - Push mini-assignment **union-find** when done

Goals for Today

- Revision and Redesign: State Machine Upgrades as a Case Study
- Debugging
- Optimization (if time allows)

Plan

Recall the Types in Our Initial Design

```
data StateMachine =
  record StateMachine where
    currentState : State
    states       : List State
    transitions  : Dict TransitionId Transition
    events       : Dict EventType (Set TransitionId)
    stateActions : Dict (StateActionTiming, State) StateAction
    transActions : Dict (TransActionTiming, TransitionId) TransitionAction
```

Recall the Types in Our Initial Design

```
State = String
TransitionId = String
EventType = String

data Transition = record Transition where
                    source : State
                    target : State
                    name   : TransitionId    -- discretionary

data Event = record Event where
               type : EventType
               payload : payloadType type    -- payload type varies with event type

data StateActionTiming = Exit | Enter
data TransActionTiming = Before | During | After

StateAction = Event -> State -> StateMachine -> ()
TransitionAction = Event -> Transition -> StateMachine -> ()
```

Applications and Extensions

1. Pattern Matching
   A simple config file format

```
[section1]
key1 = value1
key2 = value2    # end of line comment
key3 = value3

[section2]
key4 = value4    # and so forth. Note blank line between sections
```

Applications and Extensions

1. Pattern Matching
   What are the states? Transitions? Actions? What are the payloads in the events?

Applications and Extensions

1. Pattern Matching

State	Transitions to...
Top Level	Section Start, Missing Section
Section Start	Awaiting Key, Missing Key
Awaiting Key	Awaiting Sep, Missing Sep
Awaiting Sep	Awaiting Val, Missing Val
Awaiting Val	Comment, Awaiting Key, Missing Val, Section End
...	...

Think of the input as a sequence of tokens. Events are dispatched with type based on the token with the token as payload.

We can handle this with the design as is, but notice what is less convenient here. An extension would help: .

Applications and Extensions

1. Pattern Matching

State	Transitions to...
Top Level	Section Start, Missing Section
Section Start	Awaiting Key, Missing Key
Awaiting Key	Awaiting Sep, Missing Sep
Awaiting Sep	Awaiting Val, Missing Val
Awaiting Val	Comment, Awaiting Key, Missing Val, Section End
...	...

Think of the input as a sequence of tokens. Events are dispatched with type based on the token with the token as payload.
We can handle this with the design as is, but notice what is less convenient here. An extension would help: **transition data**.

Applications and Extensions

1. Pattern Matching

```
State = String
TransitionId = String
EventType = String

data Transition a = record Transition where
                      source : State
                      target : State
                      name   : TransitionId
                      info   : a

SimpleTransition = Transition ()

TransitionAction = Event -> Transition a -> StateMachine -> ()
```

(Adjusting the code)

Applications and Extensions

2. Finite State Markov Chains

   As we saw last time, Markov chain machines benefit from both transition data and *selectors*. How should the selectors be configured? Stored? For instance,

```
data StateMachine =
  record StateMachine where
    currentState : State
    states       : List State
    transitions  : Dict TransitionId Transition
    events       : Dict EventType (Set TransitionId)
    selectors    : Dict EventType Selector
    stateActions : Dict (StateActionTiming, State) StateAction
    transActions : Dict (TransActionTiming, TransitionId) TransitionAction

Selector = EventType -> Set (Transition a) -> Maybe (Transition a)  -- not pure
```

   Note how we've implicitly handled guards here. Is that a good idea?
   How does our code change?

Applications and Extensions

2. Finite State Markov Chains
   As we saw last time, Markov chain machines benefit from both transition data and *selectors*. How should the selectors be configured? Stored?

Applications and Extensions

3. User Interface Control

   Consider a very simple registration form with an email field, a password field, and a submit button. Code to manage this in an *ad hoc* manner quickly becomes complex. We have a submit button (enabled/disabled), one or more error labels (invisible/visible), and a success label (invisible/visible).

   A state machine allows us to represent much of this **code as data**!

   What are the states, transitions, and actions? Events are derived from the user, what do they look like? How are *guards* used here?

Code Smell

We use **code smell** to mean features of code, concrete or impressionistic, that hints at or suggests weakness of design, susceptibility to complexity, and vulnerability to bugs. These warnings are worth taking seriously as they can indicate deeper problems.

An example include code that is not DRY. What might be code smell in our definition of the `transitions` table?

Code Smell

Notice the repetition of the id. It rankles but is useful. Yet this is code smell. Do we need to look up transitions by id? When? Why? Having written dispatch, how should we organize these?

And notice that changing the organization also allows us to support a more flexible design: implicit or functionally defined transitions?

Code Smell

Notice the repetition of the id. It rankles but is useful. Yet this is code smell. Do we need to look up transitions by id? When? Why? Having written dispatch, how should we organize these?

And notice that changing the organization also allows us to support a more flexible design: implicit or functionally defined transitions?

For example: checking for balanced (and possibly interleaved) delimiters, e.g., LaTeX source.

What are the states and transitions that we want in this case? Can they be defined *a priori*? Should state be `Either String Int`? Pros and cons?

Plan

The Builder Pattern

What should our interface be for the `StateMachineBuilder`?

The Builder Pattern

What should our interface be for the `StateMachineBuilder`?

One desirable feature would be a **fluent** interface, where each builder method (except `build`) returns the builder so that we can chain actions. We specify a state machine by defining states, transitions, events, actions, along with guards and selectors if appropriate.

```
machine = (
    StateMachineBuilder()
    .add_states('a', 'b', 'c', 'd')
    .add_transition('a', 'b')       # Name defaults to 'a -> b'
    .add_transition('c', 'd', name='foo')
    .add_action('enter', 'c', on_c_func)
        #...
    .build()
)
```

or in R

```
machine = StateMachineBuilder() |>
            add_states('a', 'b', 'c', 'd') |>
            add_transition('a', 'b', name='ok') |>       # Name defaults to 'a -> 
            add_transition('c', 'd', name='foo') |>
            add_action('enter', 'c', on_c_func)  |>
                #... |>
            build()
```

The build method ***validates the specification*** and ***creates the*** `StateMachine` object.

Plan

Debugging and the Scientific Method

A common method to identify bugs is to put `print` statements throughout your code to track what it's doing.

This is a blunt tool, though a very easy one to use, and we can often benefit from more sophisticated tools and techniques.

Debugging is hard. As Brian Kernighan said: "Debugging is twice as hard as writing the code in the first place."

A useful perspective is to treat Debugging like the Scientific Method:

1. Formulate hypotheses
2. Make predictions about what you will see
3. Test your hypotheses
4. Record your observations!
5. Update your hypotheses and repeat.

Having a model of your code helps make this effective.

Debugging Techniques

- Readable, well-structured code
  Readable, modular code with clear structure is easier to model and thus easier to debug.
- Print statements and logging
  Low tech and clunky but still useful.
  Logging is diagnostic output (typically to a "log" file) that can be configured and turned off. It is common to specify several levels of logging, which will only run if that level is configured to.

  ```
  log_event( Logging.DEBUG, mesg, data )
  ```
- Using your tests
  Good tests help you catch errors when you introduce bugs. When you encounter a problem, adding a test to check it makes your tests more effective.
  Tests also ensure that you know what a function is supposed to do.
- Interactive Debuggers
  An interactive debugger halts program execution and allows you to inspect the current state: display local variables, view the call stack, set breakpoints, and even run new code. You can step through the code line-by-line to examine how it works.
  Debuggers can often be configured to open automatically when your program crashes or throws an exception. IDEs also let you set breakpoints and run debuggers whenever you'd like, or you can add code to invoke the debugger when desired.
- Memory profiling
  Memory "debuggers" (e.g., `valgrind`) help you track memory usage, memory management, and multi-threaded operations. More useful for languages where memory is managed manually.

Debugging in R

RStudio provides an integrated debugger that's useful for running code step-by-step or inspecting a specific function.

If you're not using RStudio, the `debug()` function can be used to tell R to enter a debugger whenever a certain function is called (see also `debugonce()`); or you can insert a call to `browser()` wherever you want your breakpoint, and when R reaches this, it will stop the code and let you explore.

R also can enter the debugger automatically when an error occurs, and you can define custom handlers to inspect the entire call stack, print out variables, and so on: `recover()`. To use it, set

```
options(error = recover)
```

at the top of your script. This tells R to use `recover()` as the default handler for all errors.

If you use `testthat` for unit tests, it supports opening a debugger automatically when a test fails. You do this by setting a special test "reporter" that reports failures by debugging them:

```
library(testthat)
test_file("test_foo.R", reporter = "debug")
```

Debugging in Python

In Python, the `pdb` debugger is built in. Some IDEs/editors can integrate with it, and you can also use `pdb` from the command line:

```
# Instead of
python ingest_crimes.py -s 2707.1 data/example_data.txt
# Run
python -m pdb ingest_crimes.py -s 2707.1 data/example_data.txt
```

Some unit testing tools pytest can automatically open a debugger when a test fails, so you can figure out exactly what happened. This can help you diagnose finicky tests:

```
# Instead of
pytest test_stuff.py
# Run
pytest --pdb test_stuff.py
```

Plan

*Programmers waste enormous amounts of time thinking about, or worrying about, the speed of noncritical parts of their programs, and these attempts at efficiency actually have a strong negative impact when debugging and maintenance are considered. We should forget about small efficiencies, say about 97% of the time: **premature optimization is the root of all evil**. Yet we should not pass up our opportunities in that critical 3%.*

– Donald Knuth

Make it *run*, Make it *clear*, Make it *right*, Make it *fast* – in that order

Debasish (দেবাশিস্) Ghosh 🇮🇳
@debasishg

Optimize **first** for readability and **then** for performance.

Reasons:

1. The whole codebase needs to be readable. Not the whole of the codebase needs to be equally performant.
2. Once you have a modular and readable codebase, identify the hotspots that need to be optimised for performance. Can sacrifice some readability there as well.
3. It's much easier to optimise a readable code for performance than a performant spaghetti for readability

Optimization Techniques

- Better Algorithms and Data Structures!

  You cannot make up for a bad choice here with optimizations. Profiling won't help an algorithm that's accidentally quadratic.
- Re-writing (parts of the code)

  Vectorizing algorithms; eliminating extra operations; reducing abstractions, checks, and "boxing," operating "closer to the hardware"
- Moving parts to a different language or runtime

  Ex: "Boxed" numbers, garba

  Rcpp in R. C and Cython for Python. Most languages support a *foreign function interface* for linking to code in other languages.
- Parallelization

  Ex: GPUs. Can be powerful, but significant gains are situational.
- Performance Critical Parts: low-level locality optimization

  Cache-aware algorithms, locality of operations and memory. Reducing indirections.

But except for the first, these steps **should be data driven**. For that, we use **profiling**.

Plan

Performance Profiling

There are two common kinds of profiling:

- **Deterministic profiling**

  Every function call and function return is monitored, and the time spent inside each function recorded

- **Statistical profiling**

  Execution is interrupted periodically and the function currently being executed recorded.

Deterministic profiling gives the most accurate data, but adds substantial overhead: every function call requires data to be stored. This overhead may slow down code – and can *distort* the profile results.

Statistical profiling may miss frequently-called fast-running functions and in general, gives less comprehensive results.

Profiling in Python

Python's built-in `timeit` package provides *quick-and-dirty* timing that handles a few of the common pitfalls with measuring execution time. You can use it from the command line or in your code (e.g., at the repl).

```
python -m timeit "'-'.join(str(n) for n in range(100))"
#prints: 10000 loops, best of 5: 30.2 usec per loop

# Or in your code
import timeit

timeit.timeit('"-".join(str(n) for n in range(100))', number=10000)
#=> 0.3018611848820001
```

This won't give you extensive data but it can give you an impression. Don't use this as your main profiling tool.

Profiling in Python

The `time` module is even quicker and dirtier:

```
t1 = time.perf_counter(), time.process_time()
some_function_to_time()
t2 = time.perf_counter(), time.process_time()
print(f"{some_function_to_time.__name__}()")
print(f" Real time: {t2[0] - t1[0]:.2f} seconds")
print(f" CPU time: {t2[1] - t1[1]:.2f} seconds")
```

if you want a very rough impression.

Better Profiling in Python

Python's built-in `cProfile` module uses deterministic profiling. It can be invoked on the the command line:

```
python -m cProfile -s time analyze_my_data.py
```

This means "hey Python, load cProfile and tell it to sort the output by execution time, then run `analyze_my_data.py`."

(There is also a built-in `profile` package, if `cProfile` were not available or if you want to build python profiling tools on top of it. Otherwise, stick to `cProfile`.)

`cProfile` can be run inside code by importing cProfile and calling `cProfile.run`. (If you are using Jupyter, you can use `%prun`.) You can also use it for code snippets.

Better Profiling in Python

```python
from cProfile import Profile
from pstats import SortKey, Stats

with Profile() as profile:
    do_something()
    (
        Stats(profile)
        .strip_dirs()
        .sort_stats(SortKey.CALLS)
        .print_stats()
    )
```

Better Profiling in Python

Example output:

```
      1330241 function calls in 121.351 seconds

 Ordered by: internal time

 ncalls  tottime  percall  cumtime  percall filename:lineno(function)
  30597   56.359    0.002   56.359    0.002 {crime.emtools.intensity}
     91   37.850    0.416   37.859    0.416 {crime.emtools.e_step}
     93   26.350    0.283   83.456    0.897 em.py:207(log_likelihood)
1265544    0.510    0.000    0.510    0.000 {built-in method exp}
  30597    0.229    0.000   56.588    0.002 em.py:181(intensity)
      1    0.025    0.025  120.487  120.487 em.py:118(fit)
    278    0.007    0.000    0.007    0.000 {method 'reduce' of 'numpy.ufu
    366    0.005    0.000    0.005    0.000 {built-in method empty_like}
    273    0.002    0.000    0.009    0.000 numeric.py:81(zeros_like)
    275    0.002    0.000    0.002    0.000 {built-in method copyto}
    183    0.001    0.000    0.008    0.000 fromnumeric.py:1852(all)
    273    0.001    0.000    0.001    0.000 {built-in method zeros}
     94    0.001    0.000    0.006    0.000 fromnumeric.py:1631(sum)
    ...
```

Better Profiling in Python (cont'd)

For statistical profiling in Python, you can use a (third-party) package like pyinstrument.

```
from pyinstrument import Profiler

profiler = Profiler()
profiler.start()

do_something() # or whatever code you want to profile

profiler.stop()
profiler.print()
```

Profiling in R

R's built-in `Rprof` uses statistical profiling. It writes profiling data to a file, which can then be analyzed with `summaryRprof`. Here's a template

```
prof_output <- tempfile()
Rprof(prof_output)  # Start profiling
#... code to be profiled
Rprof(NULL)         # End profiling

summaryRprof(prof_output)
```

Profiling in R

In a bigger program, Rprof can produce output like this:

```
> summaryRprof()
$by.self
                      self.time self.pct total.time total.pct
"specgram"               320.70    41.05     563.00     72.07
"seq.default"             48.40     6.20     114.02     14.60
"getPeaks"                41.02     5.25     780.98     99.97
":"                       34.38     4.40      34.38      4.40
"is.data.frame"           28.60     3.66      47.54      6.09
"pmin"                    25.70     3.29      58.62      7.50
"colSums"                 25.52     3.27      78.02      9.99
"seq"                     20.52     2.63     135.28     17.32
"matrix"                  20.04     2.57      21.58      2.76
"as.matrix"               19.40     2.48      50.44      6.46
...
```

Notice the presence of ":". This profile was collected several years ago, before R introduced an optimization that prevents `:` in `for` loops from literally building the entire vector in memory in advance. (Storing `1:10000` in a variable still stores the entire vector in advance, though.)

Profiling in R (cont'd)

R's **lazy evaluation** can make profiling trickier to interpret. (R does not evaluate the arguments to a function until they are used. Loosely speaking, it produces a small piece of code, called a *thunk*, for each argument that evaluated on demand.)

An alternative way to look at these results is the profvis package.

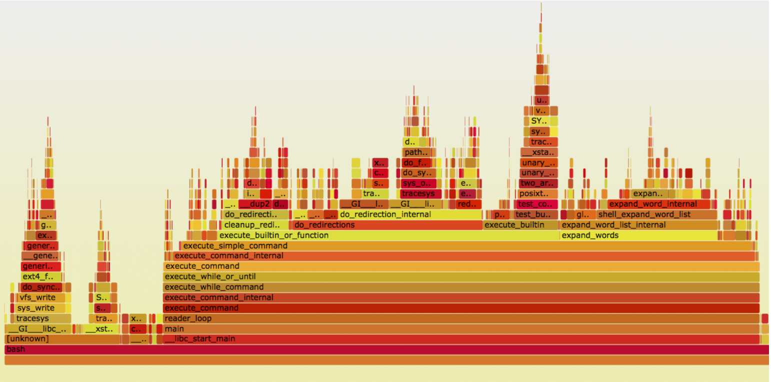
When you call `profvis`, with either a piece of code or a file name with profiling information, it opens an interactive visual display of the profiling data.

(The profvis package is integrated with RStudio, letting you profile with a single button.)

Call Graphs

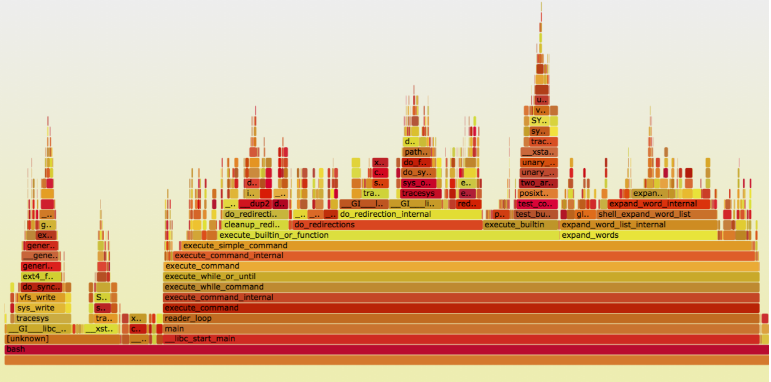
Sophisticated profilers can produce *call graphs*: a graph representing which functions call which other functions, and hence breaking down which callers contribute most to the use of a function. This can be useful if you spot a slow function but aren't sure which function is responsible for calling it the most.

One common visualization of these graphs is called a ***flame graph***. It shows which parts of the code are "hot". Tools like DTrace and SystemTap allow you to instrument code to extract this information:



Call Graphs (cont'd)

See http://www.brendangregg.com/flamegraphs.html for details on how to produce such graphs for compiled code. Julia's ProfileView.jl can build these automatically for Julia code, and the profvis package can produce an interactive web page for R profiles. (See the example here.)



Other profiling tools are Callgrind and gperftools.

Line Profiling

Some profiling tools can measure individual lines instead of whole function calls.

This can be useful if your code isn't cleanly split into many small functions. (But it should be!)

Line profiling isn't as commonly used as function profiling; look at the profvis package for an R implementation.

Always measure your changes!

It is too easy to do optimization voodoo: tweak one line, then another, then another, without knowing what is working and what is not.

Use your profiler to test if the optimizations are worthwhile. (Most optimizations add a complexity cost to your code.)

Remember that program execution time is a random variable with noise: you need more than one run to tell if a change mattered. Profile your code like a statistician!

Many languages provide microbenchmarking packages to do this for you. For example, in Python:

```
>>> import timeit
>>> timeit.timeit('"-".join(str(n) for n in range(100))', number=10000)
0.8187260627746582
>>> timeit.timeit('"-".join([str(n) for n in range(100)])', number=10000)
0.7288308143615723
>>> timeit.timeit('"-".join(map(str, range(100)))', number=10000)
0.5858950614929199
```

R has the microbenchmark package to do the same thing.

Plan

Garbage Collection

R, Python, Julia, Ruby, Java, JavaScript, and all other dynamic languages are *garbage collected* (GCed): at runtime, the interpreter must determine which variables are accessible (live) and which are not (garbage) and free memory accordingly.

In C and some other compiled languages, memory management is manual. There is a distinction between the *stack* and the *heap*:

- **Stack**
  When a function is called, its arguments are pushed onto the stack, as well as any local variables it declares, in a stack *frame*. When the function returns, its frame is popped off the stack, destroying the local variables. But the stack frame has a fixed size, and can only contain variables whose sizes are known in advance.
- **Heap**
  A global pool of explicitly-allocated memory. Contains arbitrary objects shared between functions, but requires explicit management to allocate and deallocate.

```c
double fit_big_model(double *data, int p, double tuning_param, ...) {
  double *betas = malloc(p * sizeof(double));

  // do stuff

  free(betas);
}
```

Garbage Collection (cont'd)

More advanced languages like C++, Rust, D and so on add additional ways to manage memory, like RAII, which make it less cumbersome and less error-prone. Rust uses **ownership types** to control memory use without GC.

In dynamic languages, any object can be any size, and may live arbitrarily long.

A typical strategy is *tracing*: the language keeps track of reachable objects, those referenced by local variables or global variables. It then traces out a graph: any object contained inside a local variable (e.g. inside a list in R) or accessible from one.

This produces the set of "live" objects. Any other objects are garbage and can be deallocated, since they are no longer accessible.

(This is essentially a graph traversal problem, and so there are many variations with different performance characteristics in different use cases.)

This traversal takes time. Most garbage collectors "stop the world": execution stops while they collect. If there's lots of garbage or lots of allocation, GC can be slow.

Tracking Memory Use

Some languages provide simple blunt instruments to see how much memory is allocated by code:

```
julia> @time f(10^6)
elapsed time: 0.04123202 seconds (32002136 bytes allocated)
2.5000025e11
```

Memory profiling (sometimes called "heap profiling") is not as common as ordinary profiling, but can still be very useful. There are a number of tools for different languages. Python's memory_profiler module offers line-by-line memory profiling features. A simple example from its documentation:

```python
@profile
def my_func():
    a = [1] * (10 ** 6)
    b = [2] * (2 * 10 ** 7)
    del b
    return a

if __name__ == '__main__':
    my_func()
```

Tracking Memory Use

Some languages provide simple blunt instruments to see how much memory is allocated by code:

```
julia> @time f(10^6)
elapsed time: 0.04123202 seconds (32002136 bytes allocated)
2.5000025e11
```

Memory profiling (sometimes called "heap profiling") is not as common as ordinary profiling, but can still be very useful. There are a number of tools for different languages. Python's memory_profiler module offers line-by-line memory profiling features. A simple example from its documentation:

```
Line #    Mem usage  Increment   Line Contents
==============================================
     3                           @profile
     4      5.97 MB    0.00 MB   def my_func():
     5     13.61 MB    7.64 MB       a = [1] * (10 ** 6)
     6    166.20 MB  152.59 MB       b = [2] * (2 * 10 ** 7)
     7     13.61 MB -152.59 MB       del b
     8     13.61 MB    0.00 MB       return a
```

Tracking Memory Use (cont'd)

These results are not wholly reliable, since they rely on the OS to report memory usage instead of tracking specific allocations, and garbage collection can occur unpredictably.

R's `Rprof` has memory profiling, and its output is shown in `provis` when you use the profiler in RStudio.

The valgrind tools suite (including in particular `memcheck`) provides powerful tools for memory profiling (and other things) in programs compiled with the C-toolchain.

Reducing Allocations

Common memory pitfalls include unnecessary copying:

```
foo <- function(x) {
    x$weights <- calculate_weights(x)
    s <- sample_by_weight(x)
    ...
}
```

Because we've written to `x`, `x` is copied. (This is peculiar to R's copy-on-write scheme.) Another example:

```
for (x in data) {
    results <- c(results, calculate_stuff(x))
}
```

The same happens with repeated `rbind` or `cbind` calls. Allocate `results` in advance, or use `Map` or `vapply` instead:

```
results <- numeric(nrow(data))

for (i in seq_along(data)) {
    results[i] <- calculate_stuff(data[i])
}
## or:
results <- Map(calculate_stuff, data)
# (depending on if data is a list, vector, data frame...)
```

Reducing Allocations

Intermediate results also require allocations:

```r
pois.grad <- function(y, X) {
    function(beta) {
        t(X) %*% (exp(X %*% beta) - y)
    }
}
```

In Numpy, we can specify the output array to avoid these kinds of problems, with extra tedium:

```python
def pois_grad(y, X):
    def grad(beta):
        tmp = np.empty((X.shape[0], 1))
        np.dot(X, beta, out=tmp)
        np.exp(tmp, out=tmp)
        np.subtract(tmp, y, out=tmp)

        return X.T * tmp

    return grad
```

I do not recommend writing code this ugly unless absolutely necessary. Numba is a smart optimizing compiler for Python code which can perform these kinds of optimizations automatically.

The End