Static, Logger-Style Output:
Functional and Technical Specifications

Top-level table of contents here

Motivation

This is a proposal for static, logger-style output to handle data and message output from the Stan services. The goal is to make it easier to do three things

1. Add a new type of data output to an existing service.
2. Add a new algorithm, service, model method, etc.
3. Write handlers in the interfaces.

The basis of the refactor will be to have a single, statically accessible output handler used for all output communication to interfaces.

Functional Specification

The proposal is to use a singleton to route all output in the style of traditional loggers.

Types of output

Stan's output can be usefully divided into two types of messages:

- *Message output*: text messages for information, warnings, and errors,
    - includes everything printed to the console in CmdStan

- *Data output*: string data for headers and numerical output for draws, adaptation, etc.
    - includes everything printed to file in CmdStan

Clients

There are two types of clients for the logger object:

1. *Writers/Producers*: The core algorithms and services use the logger to write output.

2. *Readers/Consumers*: The interfaces use the logger to configure the logger and read output.

Message Output

Message output will consist of two components:

1. *Log level*: an enum with values and application
    a. TRACE: we won't use this yet
    b. DEBUG: low level trace information
    c. INFO: informational output about status of jobs
    d. WARN: warnings that a user may want to understand but which do not cause processing to be interrupted, such as Jacobian warnings from the parser
    e. ERROR: errors that are recoverable at a higher level, such as rejections in the language
    f. CRITICAL: fatal errors that are non-recoverable

2. *Message*: the textual message to be delivered

The debug level will allow interfaces to filter and route output appropriately. Note that the component from which the message arose is not a part of the structured message, but can be included in the text.

Examples of log messages:

- 2018-08-10 10:35|WARN|Found possible non-linear transformation which may need Jacobian adjustment on left-hand side of line 127: exp(x) ~ normal(0, 1);
- 2018-08-10 10:36|ERROR|Expected positive scale expression on line 127, found sigma = -0.03
- 2018-08-10 10:36|INFO|Iteration 200/2000 (warmup) 2 seconds elapsed; projected completion 18 seconds

Data Output

Data output will consist of two components

1. *Tags*: one or more tags for output (e.g., parameters, sampling)
2. *Data shape description*: shape and size of data (e.g., vector<4>)
3. *Data payload*: a sequence of comma-separated numerical or string values

Examples of data messages:

- 2018-08-10 10:35|param head|theta.0,theta.1,theta.2,theta.3
- 2018-08-10 10:35|param constrain|4.239,1.87493,2.17e-19,3.14
- 2018-08-10 10:35|adapt stepsize|0.38459

Data will be row major for arrays and column major for matrices to match our internal representations. In cases where we're outputting an entire matrix on one line, it will be serialized as a single CSV sequence.

Message Producer Interface

Stan's core C++ code will be responsible for two things:

1. *Create and configure logger*: create logger singleton and configure it for output format (timestamp style, division style), available tags, and numerical precision

2. *Writing messages*: writing log messages using the logger singleton, tags, and Stan data types like `std::vector` and `Eigen::Matrix`

Message Consumer Interface

The message consumer interface is responsible for one thing:

1. *Configuring logger routing*: setting up targets for the log messages, which may either
    a. use built-in types like file loggers and standard output loggers, or
    b. Define custom loggers

For example, a built-in logger may be configured to send colored messages to the console (standard output) and send information about draws to a file and information about warmup to a second file. A custom logger can be configured to send messages to built-in sockets or to save directly as binary data structures.

There will be a reference CmdStan implementation for logging to standard output and to files with ASCII-based strings. Any specialized handlers for binary or in-memory output are the responsibility of the interfaces.

The flexibility will be available to the interfaces to use a binary representation. This will require custom handler implementation in C++. This will allow the current style of writing directly to memory in RStan and PyStan to be recreated.

Thread Safety

The logger will be synchronized at the message level to allow multiple threads to send log messages asynchronously.

Survey of Current Output Sources

The proposal needs to handle the current output sources, which originate in three locations (1) the model class, (2) the algorithms, and (3) the services. Many of these messages originate as exceptions thrown by the math library or algorithms, which are caught in the algorithms or services and converted to output. This proposal is not going to change the way exceptions are handled in the math library or algorithms, just how output is routed.

Model class output

- $K \times N$ constrained/unconstrained/algorithm-defined augmented parameters
- $K \times K \times N$ for same (HMC mass matrix, however many times you want to output it, or ADVI fullrank covariance matrix).

Algorithm output

- small tuples (Q metrics × N iterations or sub-iterations where Q is 1-10 in size so not a performance issue).

Services output

- timing information in a std::vector

Math library output

- logger messages in text (Carpenter: not sure what source there is other than model, algorithm, and services)

Services-defined:

- `double` values representing the time taken for P algorithm stages (warmup, sampling in MCMC).

- `struct` holding the config.

Model-file defined:

- K parameter names, provided once

Model-file defined for sampling:

- K *model* ('constrained') parameter values, provided per-iteration, so ultimately K×N, with Niterations.
- K *algorithm* ('unconstrained') parameter values, provided per-iteration so ultimately K×N

Model-file defined for optimization:

- K *model* ('constrained') parameter values, provided once as an estimate
- K *model* ('constrained') parameter values, provided per-iteration if requested
- K×K hessian matrix, provided once at a given point.
- K *gradient* parameter values, provided per iteration if requested so ultimately K×N … not sure that we actually have that plumbed r.n.

Model-file defined for ADVI:

- K *model* ('constrained') parameter values, provided once as an estimate
- K *model* ('constrained') parameter values, provided per iteration if requested (so K×N.
- K×K covariance estimate, provided once at a given point.
- K×K covariance estimate, provided per-iteration if requested (no plumbed r.n.)

Model-file defined, for HMC:

- K *algorithm* ('unconstrained') parameter values, provided per iteration if requested so ultimately K×N
- K *momentum* parameter values, provided per iteration if requested so ultimately K×N
- K×K or 1×K mass matrix, provided on request (at most K×K×N).

Model-file defined, for whatever algorithm:

- K×M *algorithm* parameter values holding state per N algorithm steps.
- K *gradient* parameter values, provided per iteration if requested so ultimately K×N (for any gradient-based algorithm).

Algorithm-defined for HMC:

- iteration, per-iteration
- adapted stepsize, `double`, per-iteration during warmup (fixed in sampling), so N_w
- treedepth (per-iteration), N
- acceptance rate, N, per-iteration
- energy, N, per-iteration
- log-density, N, per-itertaion
- number of integrator (leapfrog) steps, N, per-iteration
- divergence, N, per-iteration

Algorithm-defined for BFGS/LBFGS:

- iteration
- log-density, N, per-iteration

- ||dx||, per-iteration
- ||grad||, per-iteration
- α, per-iteration
- α_0, per-iteration
- number of evals, per-iteration
- “notes” <- Huh, should be broken out
- termination/convergence messages

Algorithm-defined, for Newton optimization:

- … uh, same informational message on initialization failure as MCMC… (?)
- iteration,
- log-density, N
- improvement in the log-density, N

Algorithm-defined, for ADVI:

- iteration, N
- ELBO, N
- delta_ELBO_mean, N
- delta_ELBO_med, N
- notes, N
- “MEAN ELBO CONVERGED” (once)
- “MEDIAN ELBO CONVERGED” (once)
- “MAY BE DIVERGING… INSEPCT ELBO” (once?)
- informational messages (once?)
- “time in seconds” for iteration, N
- ELBO, N
- “eta” (adapted) , once
- “drawing a sample of size X from approximate posterior”, once
- “completed” , once

Universal Output Proposal

There is one overarching theme here that has come about in my own research and deserves extra scrutiny - I’m proposing we use a popular C++ logging library as the basis for our implementation. I will detail my thought process here before breaking out the 4 parts of the implementation because it’s a very cross-cutting concern. Spdlog is an MIT-licensed, fast, header-only C++ logging library that solves a few problems we would otherwise need to solve ourselves:

1. Various types of file-based sinks as well as a simple plug-in system for custom ones. This allows us to be flexible in our choice of encoding as it will be easy to switch in the future.
2. Global static logger instance registry.
3. A great API for all of the standard log levels, timestamps, etc.

4. Thread-safe asynchronous logging backed by a lock-free MPMC queue. This means formatting, encoding, and writing to the sync happen on a backend thread pool rather than obstructing the main process.
5. A formatting library that allows encoding of user-defined data types (you must provide an implementation of `operator<<`).

For normal log messages, we should be able to use this directly. We're proposing using another static logger instance to send data to the Stan interfaces, and for that we'll want a wrapper layer that facilitates that use-case. This wrapper will be responsible for creating an API for Stan repo developers as well as registering and calling middleware plug-ins.

Let's go through the design bottom-up, starting with the interface presented to Stan interfaces.

Stan interface interface

Interfaces like RStan and PyStan will use spdlog's existing sink creation and registration mechanisms to configure where Stan will output data and log messages, respectively. They support a number of configurations beyond your garden-variety log files, including sockets and C++ ostreams. A custom sink receives a `const spdlog::details::log_msg&`, the ostream sink can be passed a `std::ostringstream` to output to a string, and the file/socket loggers work the way you would expect. Sinks are also responsible for the encoding step, so it would be possible here for a Stan interface to switch encodings, though the core of Stan will only support and provide a wrapper for spdlog's built-in ASCII file-based sinks. The flexibility for an interface to create a custom sink is really nice though - RStan could decide that formatting to ASCII is too expensive and that they would like to use callbacks, so they could implement a fairly simple custom sink to call specific callbacks like the current ones (or switch to protobuf).

We can work together a bit to come up with something convenient, but right now the way to configure Stan loggers from the interface side would be something like:

```
#include <spdlog/sinks/basic_file_sink.h>
#include <spdlog/sinks/stdout_color_sinks.h>

void init_loggers() {
 static_cast<void>(spdlog::stdout_color_mt("messages"));
 static_cast<void>(spdlog::basic_logger_mt("data", "data.log"));
}
```

All we're doing here is registering two loggers, one called "messages" and another called "data" with spdlog (the "_mt" refers to the multithreaded implementation). In this process we're telling them each where to go - the first one will print colored messages to stdout and the 2nd will write to a file called data.log in whatever the current directory is. The interfaces can choose between files, sockets, stdout/stderr, or custom sinks wrapping callbacks by using the predefined sinks or defining a custom one.

Serialization

We'll start by serializing to lightly structured ASCII using the builtin fmt library. We will just need to serialize tags and a few kinds of data objects, as described further in the Developer API section. So this very thin "serialization layer" consists mostly of custom `operator<<` implementations for pre-existing classes. Here might be how we'd serialize tags and Eigen matrices:

```
using std::ostream;

enum class Tag { header, parameters, constrained };

inline ostream &operator<<(ostream &os, const Tag &tag) {
 switch (tag) {
   case Tag::header:
     return os << "header";
   case Tag::parameters:
     return os << "parameters";
   case Tag::constrained:
     return os << "constrained";
 }
}

inline ostream &operator<<(ostream &os, const std::vector<Tag> &tags) {
 for (auto &&t : tags) {
   os << t << " ";
 }
 return os;
}

template <typename T, int R, int C>
inline std::ostream &operator<<(std::ostream &os, const Eigen::Matrix<T, R, C> &m) {
 os << "Matrix<" << m.rows() << ", " << m.cols() << ">(";
 for (int i = 0; i < m.size() - 1; i++) {
   os << m(i) << ", ";
 }
 os << m(m.size() - 1) << ")";
 return os;
}
```

Developer API

Similar to spdlog's provided global static logger, we will have a global singleton functor for outputting data (with no registry necessary in this case, though its sole state will consist of a reference to the correct spdlog data logger). This will have one function available [0] that takes in a `std::vector` of tags (members of an enum) as the first argument with the data object to be emitted as the second. The key idea here being that a constellation of tags uniquely specifies the type of data being emitted and as such there is just one type of data associated with those tags. Under the hood, the call will

present the spdlog data logger with a standardized format string encoding the list of tags and the data object, relying on its `operator<<` for eventual encoding.

```
using namespace stan::log;
emit_data({Tag::header, Tag::parameters},
        std::vector<std::string>{"theta0", "theta1", "theta2", "theta3"});
Eigen::VectorXd theta(4);
theta << 4, 5, 6, 7;
emit_data({Tag::parameters, Tag::constrained}, theta);
```

Which emits this:
[2018-08-10 10:35:41.274] [data] [info] header parameters | theta0, theta1, theta2, theta3
[2018-08-10 10:35:41.274] [data] [info] parameters constrained | Matrix<4, 1>(4, 5, 6, 7)

Obviously, exact formatting and naming TBD by whoever implements. Here is what the emit_data function should basically look like:

```
/*
* @tparam T type of data to be output; must implement operator<<
* @param tags constellation of tags uniquely identifying this type of data
* @param data data object to write out
*/
template <typename T>
inline void emit_data(std::vector<Tag> tags, const T &data) {
  auto data_logger = spdlog::get("data");
  data_logger->info("{}|{}", tags, data);
}
```

[0] We could have more than one function if we decide we want to add “data levels” similar to log levels to aid in turning some output on or off, but I am not sure we need to do that initially. Please correct me if I’m wrong on this as I likely don’t know enough yet! We can just add another overload to the function that additionally takes in a log level if need be.

Plug-ins

The problem we’re solving here is that we would like to allow one component to emit output while another component post-processes it. If one of these components knows about the other, it’s simple enough to allow that one to directly post-process rather than creating a global system for plug-in registry.

TODO: I am actually not sure if the model needing to re-constrain the parameters before logging represents a use-case where it’s best to create a global plug-in system for this, versus just having the algorithm call the appropriate method on the model before outputting parameter draws. But just in case it does require that, here’s how it could work.

We described building a global singleton functor for data logging. This same functor could hold a map from tag constellation to functions that operate any data objects written out with that constellation and returns any log messages to be written (potentially replacing both the tag constellation and the data object, suppressing messages entirely, or generating multiple messages for output). I will leave code for this to be fleshed out if we decide we must have plugins. Speaking with Bob it seemed reasonable enough to let the algorithm ask the model to re-constrain the parameters before logging, but I realize that, like any other choice, this might be contentious for some reason. :P

References

Original discussion was in this discourse thread.