# Stan Output Specification

Krzysztof Sakrejda

January 12, 2017

## 1 Signatures for Writers

### 1.1 Summary

I think this is less work internally in Stan, maybe a little more work on the interface side but it's work that we could mitigate by providing some generic implementations. We define types:

```
ConfigurationEchoWriter
EstimateWriter
DiagnosticWriter
SamplerParameterWriter
MessageWriter
```

`ConfigurationEchoWriter` must have methods for outputting all the current config input types.

```
ConfigurationEchoWriter&
ConfigurationEchoWriter::operator()(std::string key, double value);
ConfigurationEchoWriter&
ConfigurationEchoWriter::operator()(std::string key, int value);
ConfigurationEchoWriter&
ConfigurationEchoWriter::operator()(std::string key, std::string value);
ConfigurationEchoWriter&
ConfigurationEchoWriter::operator()(std::string key, bool value);
ConfigurationEchoWriter&
ConfigurationEchoWriter::operator()(std::string key, std::vector<double> value);
```

Returning the reference is (I think... should check) to make the operator call chainable so that it's easy to pass all the arguments to this config writer that currently go into a service method call (example below).

`EstimateWriter` and `DiagnosticWriter` would be for outputting the parameters currently sent to the parameter and diagnostic writers, excluding sampler parameters. These writers need methods for establishing column names and writing repeat sets of `double` values:

```
void EstimateWriter::operator()(std::vector<std::string>);
void EstimateWriter::operator()(std::vector<double>);
void DiagnosticWriter::operator()(std::vector<std::string>);
void DiagnosticWriter::operator()(std::vector<double>);
```

In practice the two might end up being the same class (?).

The `SamplerParameterWriter` needs to deal with both single *and* repeat values of scalar and (at least) vector types. The method signatures look much like the `ConfigurationEchoWriter` signatures but they will need to be less trivial to deal with the fact that there will be repeat calls to some of these (and the writer must respect order in those calls).

```
ConfigurationEchoWriter&
ConfigurationEchoWriter::operator()(std::string key, double value);
ConfigurationEchoWriter&
ConfigurationEchoWriter::operator()(std::string key, int value);
ConfigurationEchoWriter&
ConfigurationEchoWriter::operator()(std::string key, std::string value);
ConfigurationEchoWriter&
ConfigurationEchoWriter::operator()(std::string key, bool value);
ConfigurationEchoWriter&
ConfigurationEchoWriter::operator()(std::string key, std::vector<double> value);
```

The `MessageWriter` is meant to consume all the messages currently sent to the `error_writer` and `message_writer` in the new `services`. The fallback signature is:

```
MessageWriter::operator()(int log_level, std::string key, std::string value);
```

The other signatures are all:

```
MessageWriter::operator()(int log_level, std::string key, Message value);
```

Where a `Message` must have a `std::string Message::string()` that would allow interfaces to writer a `MessageWriter` for new types quickly by relying on their fallback method for strings.

1. log_level would need predefined values (maybe the log4j ones).
2. It should be possible to define an operator¡¡ s.t. you could stream text in without multiple method invocations(?) We talked about this on discourse at some point but nobody ever investigated

## 1.2 Configuration Echo

When dealing with non-trivial models we should encourage users to save their configuration so that they have a record of how an algorithm/model was run. Though it's always possible to save configuration information in scripts that run

Stan, those can be either lost or out of sync with the options passed to the Stan C++ code. For example in `rstan` a user might mis-spell an option and it would be ignored by the interface leading the code to substitute the default. To make it easy to tell what option Stan was run with, we specify a standard method for returning this information from Stan. The goal of this specification is to return the information on algorithm configuration passed to Stan's `service` methods.

### 1.2.1 Writer Callback Approach

Stan's `service` methods take many configuration arguments rather than a single type that information can be extracted from. Rather than creating and returning an object from this information we add a configuration echo writer (`ConfigurationEchoWriter`) which is *passed to the service method* like any other writer. The `ConfigurationEchoWriter` defines a number of methods for `operator()(std::string key, T value)` where `T` must include `double`, `int`, `string`, `bool`, and `std::vector<double>` (for mass matrix). Each method returns a reference to the writer to allow for chaining. In the service method, the `hmc_nuts_dense_e` method currently has a signature like:

```
int hmc_nuts_dense_e(Model& model, stan::io::var_context& init,
                           unsigned int random_seed, unsigned int chain,
                           double init_radius, int num_warmup, int num_samples,
                           int num_thin, bool save_warmup, int refresh,
                           double stepsize, double stepsize_jitter,
                           int max_depth,
                           callbacks::interrupt& interrupt,
                           callbacks::writer& message_writer,
                           callbacks::writer& error_writer,
                           callbacks::writer& init_writer,
                           callbacks::writer& sample_writer,
                           callbacks::writer& diagnostic_writer)
```

the additional `ConfigurationEchoWriter` call would look like

```
config_writer("random_seed", random_seed).("chain", chain).
  ("init_radius", init_radius).("num_warmup", num_warmup).
  ("num_samples", num_samples).("num_thin", num_thin).
  ("save_warmup", save_warmup).("refresh", refresh).
  ("stepsize", stepsize).("stepsize_jitter", stepsize_jitter).
  ("max_depth", max_depth).("algorithm", "hmc_nuts_dense_e")
```

The order of calls should not matter although the writer could choose to impose an order. There are some obvious opportunities here for us to save ourselves lines of code if in the future we choose to pass in a configuration object or maybe pass the configuration in as something like `std::map<std::string,boost::any>`.

### 1.2.2 Implications

1. Obvious boilerplate which should not change much or often now that the service methods are in place. The obvious exception is that all the boilerplate changes if we move to a more unified config object as the argument to the service method.

2. Writing a `ConfigurationEchoWriter` callback that writes a key-value format like JSON should be trivial as all the key-value pairs are on hand at the time of the call and the types would need to be cast away.

3. Writing a `ConfigurationEchoWriter` callback that constructs a typed struct should also be trivial except that the struct would once again need to specify the message components and keeping them compatible as the config information changes would involve some overhead.

4. Writing a `ConfigurationEchoWriter` callback that constructs typed ProtocolBuffers messages (or Capnproto or Flatbuffers) would also be straightforward and these formats a) can be written out as text; b) can be written out as binary messages; and c) support schema evolution in various ways so they should be simpler to implement than structs.

Other than then boilerplate there are no clear negatives to the writer approach and the boilerplate aspect could be removed by going to a more comprehensive config type.

## 1.3 Standard Estimate Output

For all models one of the outputs is either a single parameter vector or a series of parameter vectors. The meaning of these outputs is algorithm-specific and we want to ignore that when we consider the output format. This is currently handled with a generic `Writer` and there is no reason to change that approach except to simplify the requirements for the writer. These outputs match the names in the Stan program code but *exclude* sampler parameters (e.g.-treedepth, divergent, lp) and internal parameters currently included in the diagnostic writer.

### 1.3.1 Estimate-Specific Writer Approach

A simpler approach would be to only require the `EstimateWriter` to handle a `std::vector<std::string>` and a `std::vector<double>` input. The call with `std::vector<std::string>` should come first and the writer can throw otherwise. Additionally the `EstimateWriter` must respect the order of inputs as they are meaningful for many algorithms.

The signatures would look like:

```
// @throw ??? when names can not be written
void estimate_writer(std::vector<std::string> parameter_names);
```

```
// @throw ??? thrown of called before the writer is called with names.
// @throw ??? when estimate can not be written.
void estimate_writer(std::vector<double> estimate);
```

We should decide what gets thrown if the names are not provided prior to the first set of estimates being written. The writer should also throw if 1) names can not be written, 2) if estimate can not be written, or 3) if length of the estimate vector does not match the length of the names vector. Would be nice for interfaces to handle this properly so people don't end up seeing a model finish with no output.

### 1.3.2 Implications

1. The writer class for this task can still inherit from a generic writer but will be simpler to implement.

2. We will need similar classes for the sampler and diagnostic parameters, with signatures as described above, but we will be able to remove a lot of boilerplate code that is used to combine sampler, output, and diagnostic parameters before handing them to the generic writers.

3. Writing a `EstimateWriter` for text formats like JSON or YAML should be straightforward as the writer has both names and parameter values on hand.

4. Writing a `EstimateWriter` that constructs and saves a typed struct should also be straightforward as all type information is fixed and lengths are available at construction.

5. Writing a Protobuf-type writer should be straightforward as all type information is available at object constuction. Protobuf-type frameworks seem to all have a size limit but long vectors can be broken up and written out as chunks.

6. Interfaces which currently only pass in one writer for the sampler, output, and diagnostic parameters will have to pass in three writers, but each writers will be simpler to implement and all three could potentially use the same class.

## 1.4 Standard Diagnostic Output

Diagnostic output, *excluding* sampler parameters, is currently doubles-only which allows us to take the same approach as the `EstimateWriter`. In fact diagnostic output could be handled by an `EstimateWriter`.

### 1.5 Sampler Output

Sampler parameters are sometimes output per iteration (e.g.-stepsize in HMC or gradient values in optimization). Sometimes sampler parameters are only output at specific time (e.g.-mass matrix in NUTS). Currently we drop type information when these sampler parameters are passed in with other output parameters.

#### 1.5.1 Writer Callback Approach

Stan's samplers return a small set of typed parameters with not set size. We could take a similar approach to the `ConfigurationEchoWriter`. The `SamplerParameterWriter` would define methods for `operator(std::string key, T value)` where `T` must include `double`, `int`, `string`, `bool`, and `std::vector<double>` (for mass matrix). Each method returns a reference to the writer to allow for chaining.

The writer must deal with the possibility that a given operator may be called with a given key multiple times. An in-memory writer could simply append to `std::vector<T>`.

#### 1.5.2 Implications

1. This approach allows for typed sampler output as well as non-scalar sampler output which is already relevant for HMC.

2. Creating one writer for these diverse types does defer the choice about how sampler parameters should be output to the writer. This might push some more work onto the interfaces but we could provide a lower-common-denominator output via a key-value text writer.

3. Creating a binary writer requires a little more design, see CapnProto section below (Protobuf schema would be very similar).

## 2 Binary Output Message Types

To some extent this section is relevant for either a roll-your own binary output type which I envision as some structs + boost::serialize as well as schema-based formats such as Protobuf. I'm going to write out the spec with the capnproto specification for schema but Protobuf would be similar.