Metrum Research Group LLC
Phone: 860.735.7043
billg@metrumrg.com, yiz@metrumrg.com

2 Tunxis Road, Suite 112
Tariffville, CT 06081
www.metrumrg.com

Torsten: A Pharmacokinetic/Pharmacodynamic Model Library for Stan

Developers Guide
(Torsten Version 0.89rc, Stan version 2.27.0)

May 11, 2022

Contents

Development team

- William R. Gillespie , Metrum Research Group
- Yi Zhang , Metrum Research Group
- Charles Margossian , Columbia University, Department of Statistics

Acknowledgements

Institutions

We thank Metrum Research Group, Columbia University, and AstraZeneca.

Funding

This work was funded in part by the following organizations:

Office of Naval Research (ONR) contract N00014-16-P-2039. provided as part of the Small Business Technology Transfer (STTR) program. The content of the information presented in this document does not necessarily reflect the position or policy of the Government and no official endorsement should be inferred.

Bill & Melinda Gates Foundation.

Individuals

We thank the Stan Development Team for giving us guidance on how to create new Stan functions and adding features to Stan's core language that facilitate building ODE-based models.

1

Introduction

Following Stan's code structure, Torsten forks four Stan repositories:

- The automatic differentiation and mathematical infrastructure `Stan/Math`.
- The inference engine and statistical model infrastructure `Stan`.
- Platform-independent command line interface `CmdStan`.
- Stan language to C++ transpiler `stanc3`.

Same as Stan, the forked repos are in a submodule hierarchy

```
Torsten/cmdstan/stan/lib/stan_math
```

Additionally Torsten's own mathematical functions are in a separate repository. It is a submodule of `Stan/Math` that contains all of Torsten functions

```
Torsten/cmdstan/stan/lib/stan_math/stan/math/torsten
```

as well as tests.

```
Torsten/cmdstan/stan/lib/stan_math/stan/math/torsten/test
```

The rest of this document describes the basic layout of the components in Torsten implementation. The details can be found in code comments.

2

torsten_math

Most Torsten development occurs in the `torsten_math` repo. So far most functions there are for solving PKPD ODE systems. To understand the design, let us take a look at the user-facing `pmx_solve_rk45` function as an example.

```
  template <typename F, typename... Ts>
  auto pmx_solve_rk45(const F& f, const int nCmt, Ts... args) {
    using scheme_t = torsten::dsolve::odeint_scheme_rk45;
    return
    PMXSolveODE<dsolve::PMXOdeIntegrator<dsolve::PMXVariadicOdeSystem,
     ↪  dsolve::PMXOdeintIntegrator<scheme_t>>>::solve(f, nCmt, args...);
}
```

The argument `f` is the ODE right-hand-side specification, `nCmt` is the number of the states which equals to the size of `f`'s output. Parameter pack `args` are for NMTRAN-compatible event specification as well as ODE solver controls. The function is just a wrapper of the line

```
PMXSolveODE<dsolve::PMXOdeIntegrator<dsolve::PMXVariadicOdeSystem,
dsolve::PMXOdeintIntegrator<scheme_t>>>::solve(f, nCmt, args...);
```

`PMXOdeIntegrator` class describes the ODE integrator. It has two template arguments. The first is `PMXVariadicOdeSystem` that will be constructed using `f` and provides the ODE information (RHS, Jacobian, etc). The second is `PMXOdeintIntegrator` that indicates we will be using an integrator from `Boost::odeint` library. The specific integrator is `torsten::dsolve::odeint_scheme_rk45`. An alternative is

```
torsten::dsolve::odeint_scheme_ckrk
```

used in `pmx_solve_ckrk`.

Here `PMXVariadicOdeSystem` is used to differentiate from the old ways of using fixed parameters in ODE solvers, which is still used in unit tests. The `PMXOdeintIntegrator` parameter is used to set apartment from `CVODES` ODE solvers that `pmx_solve_bdf` and `pmx_solve_adams` are based on.

2.1. PMX solver

The `PMXSolveODE` class uses the type of ODE integrator and ODE system as arguments. It is the basis of all user-facing numerical ODE functions. It provides a `static` function `solve` that uses event schedule arguments and ODE controls to

- construct events from NMTRAN arguments,
- construct requested ODE integrator,
- construct event solvers according to the events and the type of integrator,
- solve events chronologically.

A similar class `PMXSolveCPT` does the same but for compartment models that employ close-form solutions.

2.2. PMX ODE integrators

The purpose of `PMXOdeIntegrator` class is to separate the ODE solver, the control parameters of numerical solutions, the ODE system, and the user-facing functions. A `PMXOdeIntegrator` object is constructed based on the type of ODE system (variadic or not), the type of solver to be used (`Boost::odeint` vs `CVODES`), and the controls (tolerance and maximum number of steps).

An `PMXOdeIntegrator` object is constructed inside `PMXSolveODE::solve` for numerically solving ODEs.

2.3. ODE integrators

`PMXOdeIntegrator` class delegates to specific types of ODE integrators from `Boost::odeint` and `CVODES` libraries for actual numerical solution. Thus it has a type argument that specifies the integrator to be used. All the supported numerical solvers can be found in `torsten_math/dsolve`. Torsten uses its own implementation based on library APIs instead of directly build upon Stan's ODE integration functions.

2.4. PMX models

In `PMXSolveCPT` and `PMXSolveODE` class, PMX models are constructed using NMTRAN inputs. For example, the one-compartment model class has the following construct.

```
template<typename T_par>
class PMXOneCptModel {
  const T_par &CL_;
  const T_par &V2_;
  const T_par &ka_;
  const T_par k10_;
  const std::vector<T_par> alpha_;
  const std::vector<T_par> par_;

public:
  static constexpr int Ncmt = 2;
  static constexpr int Npar = 3;
  static constexpr PMXOneCptODE f_ = PMXOneCptODE();

//...
};
```

One can see that it stores model parameters clearance, volume of distribution, absorption coefficient, which are used to construct parameters of close-form solutions. The class also

contains static components such as number of compartments, number of parameters, and the RHS of the one-compartment ODE system.

Here is a list of Torsten models ("/" indicate the model consisting two coupled components)

TABLE 2.1. Models supported in Torsten

Model	Class	File
One-cpt PK	`PMXOneCptModel`	`pmx_onecpt_model.hpp`
Two-cpt PK	`PMXTwoCptModel`	`pmx_twocpt_model.hpp`
Linear ODE	`PMXLinODEModel`	`pmx_linode_model.hpp`
One-cpt/effective-cpt	`PMXOneCptEffCptModel`	`pmx_onecpt_effcpt_model.hpp`
Two-cpt/effective-cpt	`PMXTwoCptEffCptModel`	`pmx_twocpt_effcpt_model.hpp`
General ODE	`PKODEModel`	`pmx_ode_model.hpp`
Close-form PK/general ODE	`PKCoupledModel`	`pmx_coupled_model.hpp`

Each model has an overloaded member function `solve` that solves a given event. The overloading is for different signatures in transient and steady-state solutions.

2.5. Event management

The `PMXSolveODE::solve` (`PMXSolveCPT::solve` is similar) function looks like this.

```
template <typename T0, typename T1, typename T2, typename T3, typename T4,
          typename T5, typename T6, typename F>
static stan::matrix_return_t<T0, T1, T2, T3, T4, T5, T6>
solve(
// args...
     ) {
// ...

  using ER = NONMENEventsRecord<T0, T1, T2, T3>;
  using EM = EventsManager<ER, NonEventParameters<T0, T4, std::vector,
   ↪  std::tuple<T5, T6> >>;
  const ER events_rec(nCmt, time, amt, rate, ii, evid, cmt, addl, ss);

  Matrix<typename EM::T_scalar, -1, -1> pred(EM::nCmt(events_rec),
   ↪  events_rec.num_event_times());

  using model_type = torsten::PKODEModel<typename EM::T_par, F>;

  integrator_type integrator(rel_tol, abs_tol, max_num_steps, as_rel_tol,
   ↪  as_abs_tol, as_max_num_steps, msgs);
  EventSolver<model_type, EM> pr;

  pr.pred(0, events_rec, pred, integrator, pMatrix, biovar, tlag, nCmt,
   ↪  f);
  return pred;
}
```

One can see that first a `NONMENEventsRecord` object is created using the NMTRAN arguments (template parameter type T1-T6 are for these arguments), then `EventsManager` and `EventSolver` objects are constructed. The purpose of `EventsManager` is to create and sort events chronologically, based on NMTRAN input. The purpose of `EventSolver` is to solve the events using specified ODE solver.

3

stan/math

The stan/math repo serves as a pass-through fork for torsten_math. It is almost identical to Stan's upstream repo, with the only noticeable difference in the math/stan/math.hpp, in which the torsten namespace is added:

```
#ifndef STAN_MATH_HPP
#define STAN_MATH_HPP

/**
 * \defgroup prob_dists Probability Distributions
 */

/**
 * \ingroup prob_dists
 * \defgroup multivar_dists Multivariate Distributions
 * Distributions with Matrix inputs
 */
/**
 * \ingroup prob_dists
 * \defgroup univar_dists Univariate Distributions
 * Distributions with scalar, vector, or array input.
 */

#include <stan/math/rev.hpp>

#include <stan/math/torsten/torsten.hpp>
using namespace torsten;
#endif
```

One can generate C++ code documention for Torsten using the same doxygen process as in stan/math.

```
make doxygen
```

To access the generated Torsten documentation, point the browser to

```
/stan_math/doc/api/html/index.html
```

and find torsten namespace.

4

Stan

The forked `stan` serves passing `stan/math` through as well as testing ground for experimental inference algorithms such as cross-chain warmup (see Section 6.2).

5

CmdStan

Similar to `Stan`, the forked command line interface has boilerplate code for the cross-chain warmup algorithm. It also contains Torsten-specific `makefile` flags. Similar to its upstream repo, the making process will download a `stanc3` binary in order to transpile `Stan` code to `C++` code. Changes have been made in the repo to download Torsten-compatible `stanc3` binary so that the transpiler recoganizes Torsten function signatures.

6

MPI parallelization

As an alternative to Stan's `reduce_sum` function designed for multicore infrastrue, Torsten provides MPI parallelization for population models as well as experimental cross-chain warmup model.

6.1. Population solver

The population solver functions `pmx_solve_group_rk45|bdf|adams` have similar construct to their single-subject counterparts. For example

```
template <typename F, typename... Ts>
auto pmx_solve_group_bdf(const F& f, const int nCmt,
                         const std::vector<int>& len, Ts... args) {
  return PMXSolveGroupODE<dsolve::PMXOdeIntegrator<dsolve::PMXVariadicOde
    ↪  System, dsolve::PMXCvodesIntegrator<CV_BDF,
    ↪  CV_STAGGERED>>>::solve(f, nCmt, len, args...);
}
```

is the group solver version of `pmx_solve_bdf`. The only difference is instead of using `PMXSolveODE` here we use `PMXSolveGroupODE` class to numerically solve the *population's* ODEs.

When looping through events, group solver distributes the population to parallel processes for ODE solution. At the end of each event, the solver collects results of the entire population from the processes. This mechanism is implemented in the `EventSolver` class, along with its sequential version.

6.2. Cross-chain warmup

The experimental algorithm sits on top of Stan's sampler engine.

```
Torsten/cmdstan/src/stan/mcmc/cross_chain
```

The sampler is modified to include the cross-chain adaptation. For example, function `adapt_diag_e_nuts::transition` becomes

```
sample transition(sample& init_sample, callbacks::logger& logger) {
  sample s = diag_e_nuts<Model, BaseRNG>::transition(init_sample, logger);

  if (this->adapt_flag_) {
    this->stepsize_adaptation_.learn_stepsize(this->nom_epsilon_,
                                              s.accept_stat());
```

```
    if (this -> use_cross_chain_adapt()) {
      /// cross chain adapter has its own var adaptor so needs to add
       ↪  sample
      this -> add_cross_chain_sample(s.log_prob(), this -> z().q);
      bool update = this -> cross_chain_adaptation(this ->
       ↪  z().inv_e_metric_, logger);
      if (update) {
        // this->init_stepsize(logger);
        double new_stepsize = this ->
         ↪  cross_chain_stepsize(this->nom_epsilon_);
        this -> set_nominal_stepsize(new_stepsize);
        this->stepsize_adaptation_.set_mu(log(10 * this->nom_epsilon_));
        this->stepsize_adaptation_.restart();
      }
    } else {
      bool update =
       ↪  this->var_adaptation_.learn_variance(this->z_.inv_e_metric_,
                                                this->z_.q);
      if (update) {
        this->init_stepsize(logger);
        this->stepsize_adaptation_.set_mu(log(10 * this->nom_epsilon_));
        this->stepsize_adaptation_.restart();
      }
    }
  }
  return s;
}
```

The `this -> use_cross_chain_adapt` condition controls if cross-chain warmup is used and choose the adaptation accordingly.

7

stanc3

The forked `stanc3` contains a

```
stanc3/src/middle/torsten.ml
```

file for Torsten function signatures. Unlike Stan functions, functions like `pmx_solve_rk45` supports a long list of signatures in order to allow a combination of

- parameters shared by the entire population vs subject-specifc,
- time-independent parameters vs time-dependent parameters,
- default (thus omittable) $F = 1.0$ vs user-specified bioavailability,
- default (thus omittable) $t_{\text{Lag}} = 0.0$ vs user-specified lag time,
- default (thus omittable) vs user-specified control parameters.

Any higher-order Torsten function must have its signature defined in `torsten.ml` in order to be recognized by the transpiler. Thus the workflow of adding a Torsten function usually is to first implement the function in `torsten_stan` followed by adding its signature in `torsten.ml`.

7.1. Adding a new function

Now let us walkthrough how a new function is added in Torsten using `pmx_solve_linode` function as example. The function solves dosing events using a linear ODE model, specified as a coefficient matrix for the RHS of a linear ODE.

First we implement the C++ function in `torsten_math`. As one can find at

```
stan_math/stan/math/torsten/pmx_solve_linode.hpp
```

the implementation should be in `torsten` namespace.

```
namespace torsten {
  // ...
template <typename T0, typename T1, typename T2, typename T3,
          typename T4, typename T5, typename T6>
stan::matrix_return_t<T0, T1, T2, T3, T4, T5, T6>
pmx_solve_linode(const std::vector<T0>& time,
                 const std::vector<T1>& amt,
                 const std::vector<T2>& rate,
                 const std::vector<T3>& ii,
                 const std::vector<int>& evid,
                 const std::vector<int>& cmt,
                 const std::vector<int>& addl,
```

```
                  const std::vector<int>& ss,
                  const std::vector< Eigen::Matrix<T4, -1, -1> >& system,
                  const std::vector<std::vector<T5> >& biovar,
                  const std::vector<std::vector<T6> >& tlag) {
  // ...
}
}
```

As part of test-based development process, there multiple unit tests for this function

```
stan_math/stan/math/torsten/test/unit/linode_typed_finite_diff_test.cpp
stan_math/stan/math/torsten/test/unit/linode_typed_overload_test.cpp
stan_math/stan/math/torsten/test/unit/linode_typed_test.cpp
```

for testing with finite-difference results, overloaded function signature, and solution correctness, respectively.

To have the Stan language recoganize a regular function like `pmx_solve_linode` we only need to add its signature to the aforementioned `torsten.ml` file. For the above signature, one can find the corresponding signature

```
add_func
  ( "pmx_solve_linode"
  , ReturnType UMatrix
  , [ (AutoDiffable, UArray UReal)        (* time *)
    ; (AutoDiffable, UArray UReal)        (* amt *)
    ; (AutoDiffable, UArray UReal)        (* rate *)
    ; (AutoDiffable, UArray UReal)        (* ii *)
    ; (DataOnly, UArray UInt)             (* evid *)
    ; (DataOnly, UArray UInt)             (* cmt *)
    ; (DataOnly, UArray UInt)             (* addl *)
    ; (DataOnly, UArray UInt)             (* ss *)
    ; (AutoDiffable, UArray UMatrix)      (* pMatrix *)
    ; (AutoDiffable, (UArray (UArray UReal)))      (* biovar *)
    ; (AutoDiffable, (UArray (UArray UReal))) ],   (* tlag *)
  Common.Helpers.AoS) ;
```

One can easily map the above argumentss and return value to the C++ function. Note that we need to make the Stan language aware of whether an argument could be parameter (`AutoDiffable`) or not (`DataOnly`).

7.2. Adding a new high-order function

Torsten's numerical ODE solvers are high-order functions, i.e. functions with function arguments. Adding a new high-order function is slightly more complicated. Let us use `pmx_solve_rk45` as an example. As shown in Section 1, the function uses variadic arguments in order to support different signatures. Here let us assume the one of them as follows.

```
namespace torsten {
    template <typename T0, typename T1, typename T2, typename T3,
     ↪  typename T4,
              typename T5, typename T6, typename F>
    static stan::matrix_return_t<T0, T1, T2, T3, T4, T5, T6>
    pmx_solve_rk45(const F& f,
                   const int nCmt,
                   const std::vector<T1>& amt,
                   const std::vector<T2>& rate,
                   const std::vector<T3>& ii,
                   const std::vector<int>& evid,
                   const std::vector<int>& cmt,
                   const std::vector<int>& addl,
                   const std::vector<int>& ss,
                   const std::vector<std::vector<T4> >& pMatrix,
                   const std::vector<std::vector<T5> >& biovar,
                   const std::vector<std::vector<T6> >& tlag,
                   std::ostream* msgs) {//...}
}
```

Note that now we have f for the ODE RHS, `nCmt` as number of compartments, and `msgs` for I/O of the ODE solver messages.

In order to support the above signature in Stan, in the `torsten.ml` we need first define the ODE function signature.

```
let pmx_solve_ode_func =
  [ ( UnsizedType.AutoDiffable
    , UnsizedType.UFun
        ( [ (UnsizedType.AutoDiffable, UnsizedType.UReal) (* time *)
          ; (UnsizedType.AutoDiffable, UnsizedType.UVector) (* states *)
          ; (UnsizedType.AutoDiffable, UArray UReal) (* real param *)
          ; (DataOnly, UArray UReal); (DataOnly, UArray UInt) ] (* int
           ↪  param *)
        , ReturnType UnsizedType.UVector (* return type *)
        , FnPlain, AoS) ) ]
```

Now we can add the signature as [1]

```
add_func
  ( "pmx_solve_linode"
  , ReturnType UMatrix
  , [ pmx_solve_ode_func                          (* f *)
    ; (UnsizedType.DataOnly, UnsizedType.UInt)    (* nCmt *)
    ; (AutoDiffable, UArray UReal)                (* time *)
    ; (AutoDiffable, UArray UReal)                (* amt *)
    ; (AutoDiffable, UArray UReal)                (* rate *)
    ; (AutoDiffable, UArray UReal)                (* ii *)
```

[1] The actual code in `torsten.ml` is more complicated in order to accomondate many variants of the above signature.

```
    ; (DataOnly, UArray UInt)              (* evid *)
    ; (DataOnly, UArray UInt)              (* cmt *)
    ; (DataOnly, UArray UInt)              (* addl *)
    ; (DataOnly, UArray UInt)              (* ss *)
    ; (AutoDiffable, UArray UMatrix)       (* pMatrix *)
    ; (AutoDiffable, (UArray (UArray UReal)))      (* biovar *)
    ; (AutoDiffable, (UArray (UArray UReal))) ],   (* tlag *)
  Common.Helpers.AoS) ;
```

8

`Torsten` container

All the above repos are collected in the container repo `Torsten` using `git subtree` command, so that user only needs to clone this repo in order to use Torsten. Documentation and example models can also be found in this repo.