THE TWO-STATE RANDOM WALK ON A 2-D SQUARE LATTICE

DAVE SCHNEIDER

Abstract. Particles that switch between two different behaviors appear in many models of complex physical, chemical and biological processes. In this example motivated by the classic work of Weiss [1], particles are viewed as hopping among adjacent sites on a two-dimensional lattice. The particles switch between two classes of behavior, fixed and mobile. The fixed particles do not move until they switch to the mobile class. Particles in the mobile class wait on the current lattice site for a residence time defined by a random variable, then jump to an adjacent lattice site. The random walk process is Markovian if the residence times are drawn from a negative exponential distribution, otherwise it is semi-Markovian.

1. High-level structure

⟨*⟩≡
  ⟨*weiss.hpp*⟩
  ⟨*weiss.cpp*⟩

1.1. Intellectual property rights.

This software is in the public domain.

⟨*Disclaimer*⟩≡

1.2. **Header file.**

⟨*weiss.hpp*⟩≡
  ⟨*Disclaimer*⟩
  ⟨*Standard C++ headers*⟩
  `#include "stochnet.h"`
  ⟨*Boost headers*⟩
  ⟨`AFIDD` *headers*⟩
  ⟨*Local type definitions*⟩

```
// --------------------------------------------------------------------------
// This is magic.  The requisite free functions must be declared in the
// afidd::svm namespace prior to the inclusion of partial_core_matrix.h
// --------------------------------------------------------------------------

#include "partial_core_matrix.h"
```

This code makes heavy use of templates. It is convenient to define some derived types that will be used as template parameters and in declarations of local variables in lieu of standard types such as `int` and `float`. Changing these `typedef` statements here will maintain consistency of types throughout the code.

⟨*Local type definitions*⟩≡

```
typedef std::size_t counter_type ;
typedef std::string string_type ;
typedef long coordinate_type ;
typedef double real_type ;
```

1.3. **The main program.**

⟨*weiss.cpp*⟩≡
  ⟨*Disclaimer*⟩

  `#include "weiss.hpp"`

  ⟨*Version information*⟩

```
int main(int argc, char* argv[])
{
```

    ⟨*Local variable definitions*⟩
    ⟨*Process command line arguments*⟩
    ⟨*Generate help message as appropriate*⟩
    ⟨*Set logging level*⟩
    ⟨*Display version as appropriate*⟩
    ⟨*Open output stream*⟩
    ⟨*Set up model*⟩
    ⟨*Loop over steps*⟩
    ⟨*Close output stream*⟩
    ⟨*Return with status code*⟩

```
}
```

2. Versioning

Versioning information is hard-coded as two `const counter_type` variables. This information will be used to construct help messages.

⟨*Version information*⟩≡

```
static const counter_type major_version = 0 ;
static const counter_type minor_version = 1 ;
static const string_type version =
       boost::lexical_cast<string_type>(major_version) +
       "." +
       boost::lexical_cast<string_type>(minor_version) ;
```

3. Command line processing

If help is requested, write message to `std::cerr` and return without error.

⟨*Standard C++ headers*⟩≡

```
#include <iostream>
```

⟨*Generate help message as appropriate*⟩≡

```
if (vm.count("help"))
{
  std::cerr << combined_options << std::endl ;
  return boost::exit_success ;
}
```

Set logging level.

⟨`AFIDD` *headers*⟩≡

```
#include "stochnet.h"
#include "logging.h"
```

⟨*Set logging level*⟩≡

```
afidd::log_init(vm["loglevel"].as<string_type>()) ;
```

Display version if request, and return without error.

⟨*Display version as appropriate*⟩≡

```
if (vm.count("version"))
{
  std::cout << argv[0] << " version " << version << std::endl ;
  return boost::exit_success ;
} ;
```

Manage output stream.

⟨*Standard C++ headers*⟩+≡

```
#include <fstream>
```

⟨*Open output stream*⟩≡
```
std::ofstream os ;
string_type ofname = vm["output"].as<string_type>() ;
BOOST_LOG_TRIVIAL(info) << "Opening output file " << ofname ;

os.open(ofname.c_str(), std::ofstream::out) ;
```

⟨*Close output stream*⟩≡
```
BOOST_LOG_TRIVIAL(info) << "Closing output file " << ofname ;
os.close() ;
```

Boost redefines return codes to reduce platform dependencies.

⟨*Boost headers*⟩≡
```
#include <boost/cstdlib.hpp>
```

⟨*Return with status code*⟩≡
```
return boost::exit_success ;
```

4. Program options

Arguments supplied on the command line are interpreted using the `boost::program_options` library.

⟨*Boost headers*⟩+≡
```
#include <boost/program_options.hpp>
namespace po = boost::program_options ;
```

⟨*Process command line arguments*⟩≡
  ⟨*Declare general options*⟩
  ⟨*Declare method-specific options*⟩
  ⟨*Combine options*⟩
  ⟨*Parse command line*⟩

General options.

⟨*Local variable definitions*⟩≡
```
po::options_description general_options("General options") ;
```

Help.

⟨*Declare general options*⟩≡
```
general_options.add_options()
  (
   "help,h",
   "Produce help message"
   ) ;
```

Version.

⟨*Declare general options*⟩+≡
```
general_options.add_options()
  (
   "version,v",
   "Print version string to standard output"
   ) ;
```

Logging level.

⟨*Declare general options*⟩+≡
```
general_options.add_options()
  (
  "loglevel,l",
  po::value<string_type>()->default_value("error"),
  "Logging level: trace, debug, info, warning, error, or fatal"
  ) ;
```

Output file name.

⟨*Declare general options*⟩+≡
```
general_options.add_options()
  (
   "output,o",
   po::value<string_type>()->default_value("weiss.out"),
   "Name of output file"
   ) ;
```

Number of transitions.

The number of jumps (movements between lattice sites) is less than or equal to the number of transitions because some transitions involve change of mobility status, not location.

⟨*Declare general options*⟩+≡
```
general_options.add_options()
  (
   "transitions,t",
   po::value<counter_type>()->default_value(100),
   "Number of transitions"
   ) ;

general_options.add_options()
  (
   "replicates,r",
   po::value<counter_type>()->default_value(100),
   "Number of replicates"
   ) ;
```

Method-specific options.

⟨*Local variable definitions*⟩+≡
```
po::options_description method_options("Method-specific options") ;
```

Seed for random number generator.

⟨*Declare method-specific options*⟩≡

```
method_options.add_options()
(
  "seed,s",
  po::value<boost::uint32_t>()->default_value(2342387),
  "Seed for Boost random number generator (boost::uint32_t)"
) ;
```

Parameters for Weibull distribution of residence times.

⟨*Declare method-specific options*⟩+≡

```
method_options.add_options()
(
 "alpha,a",
 po::value<real_type>()->default_value(2.0),
 "Shape parameter for Weibull distribution of residence times."
) ;

method_options.add_options()
(
 "beta,b",
 po::value<real_type>()->default_value(3.0),
 "Scale parameter for Weibull distribution of residence times."
) ;
```

Combine options.

⟨*Local variable definitions*⟩+≡

```
po::options_description combined_options("Usage: weiss") ;
```

⟨*Combine options*⟩≡

```
combined_options.add(general_options) ;
combined_options.add(method_options) ;
```

Parse options supplied on command line.

⟨*Local variable definitions*⟩+≡

```
po::variables_map vm ;
```

⟨*Parse command line*⟩≡

```
po::command_line_parser clp(argc, argv) ;
clp.options(combined_options) ;
po::store(clp.run(), vm) ;
po::notify(vm) ;
```

5. Model specification

The random walk process will be represented using an uncolored stochastic Petri net where the residence times in each state are determined by user-select probability distributions. Each token will represent a single walker. The model could be generalized to represent the behavior of a population of walkers by creating multiple tokens.

5.1. **Tokens.** Since unncolored tokens to not maintain any state, the `Walker` data type is trivial.

⟨*Local type definitions*⟩+≡
```
struct Walker
{
  // Intentionally empty
} ;
```

5.2. **Places.** Places are identified by unique keys. In this model, each lattice site is associated with two places, one to hold the subset of tokens representing walkers that are temporarily immobile and the other to the remaining tokens representing mobile walkers. Unique identifiers for places will be constructed from the coordinates of the site and the kind of tokens, mobile or immobile, that will be held.

Lattice sites will be identified by x and y coordinates and the mobility class.

⟨*Local type definitions*⟩+≡
```
enum class Mobility : int { mobile = 1, immobile = -1 } ;

std::ostream& operator<<(std::ostream& os, const Mobility& m)
{
  return os << (m == Mobility::mobile ? "M" : "I") ;
} ;
```

⟨`AFIDD` *headers*⟩+≡
```
#include "smv_algorithm.h"
```

⟨*Local type definitions*⟩+≡

```
struct PlaceKey
{
  PlaceKey() = default ;
  PlaceKey(const PlaceKey&) = default ;
  PlaceKey(coordinate_type _x, coordinate_type _y, Mobility _m) :
    x(_x), y(_y), mobility(_m)
  {
    // Empty
  }

  coordinate_type x, y ;
  Mobility mobility ;

  friend inline
  std::ostream& operator<<(std::ostream& os, const PlaceKey& pk)
  {
    return os << "{" << pk.x << ", " << pk.y << ", " << pk.mobility << "}" ;
  }

  friend inline
  bool operator==(const PlaceKey& a, const PlaceKey& b)
  {
    return (a.x == b.x) && (a.y == b.y) && (a.mobility == b.mobility);
  }

  friend inline
  bool operator<(const PlaceKey& a, const PlaceKey& b)
  {
    return afidd::smv::lazy_less(a.x, b.x, a.y, b.y, a.mobility, b.mobility) ;
  }
} ;
```

5.3. **Transitions.** Like places, transitions are also idenfied by unique keys. In this model, all transitions will have a single input place and a single output place. This makes it easy to define unique keys.

⟨*Local type definitions*⟩+≡

```
struct TransitionKey
{
  TransitionKey() = default ;
  TransitionKey(const TransitionKey&) = default ;
  TransitionKey(PlaceKey _from, PlaceKey _to) :
    from(_from), to(_to)
  {
    // Empty
  }

  PlaceKey from, to ;

  friend inline
  std::ostream& operator<<(std::ostream& os, const TransitionKey& tk)
  {
    return os << "{" << tk.from << " -> " << tk.to << "}" ;
  }

  friend inline
  bool operator==(const TransitionKey& a, const TransitionKey& b)
  {
    return (a.from == b.from) && (a.to == b.to) ;
  }

  friend inline
  bool operator<(const TransitionKey& a, const TransitionKey& b)
  {
    return afidd::smv::lazy_less(a.from, b.from, a.to, b.to) ;
  }
} ;
```

5.4. **Generation of (pseudo) random variates.**

⟨*Standard C++ headers*⟩+≡

```
#include <random>
```

⟨*Local type definitions*⟩+≡

```
using RandGen = std::mt19937 ;
```

6. Model specification

⟨`AFIDD` *headers*⟩+≡

```
#include "logging.h"
#include "distributions.h"
```

⟨*Local type definitions*⟩+≡
```
using Dist    = afidd::smv::TransitionDistribution<RandGen> ;
using ExpDist = afidd::smv::ExponentialDistribution<RandGen> ;
using Weibull = afidd::smv::WeibullDistribution<RandGen> ;
```

⟨AFIDD *headers*⟩+≡
```
#include "gspn.h" // Needed only for trans_t?
```

⟨*Local type definitions*⟩+≡
```
class BrownionGSPN
{
  // Could store the state parameters and distributions here
  // if we wanted.
};

struct UserState
{
  real_type weibull_shape, weibull_scale ;

  void shape(const real_type& s) { weibull_shape=s ; }
  void scale(const real_type& s) { weibull_scale=s ; }

  const real_type& shape() const { return(weibull_shape) ;}
  const real_type& scale() const { return(weibull_scale) ;}
};

namespace afidd
{
  namespace smv
  {
    template<>
      struct petri_place<BrownionGSPN>
      {
        typedef PlaceKey type;
      };

    template<>
      struct petri_transition<BrownionGSPN>
      {
        typedef TransitionKey type;
      };
  }
}
```

⟨AFIDD *headers*⟩+≡
```
#include "marking.h"
```

⟨*Local type definitions*⟩+≡

```
using TokenContainer = afidd::smv::Uncolored<Walker> ;
using Mark  = afidd::smv::Marking<afidd::smv::place_t<BrownionGSPN>,TokenContainer> ;
using Local = afidd::smv::LocalMarking<TokenContainer> ;

namespace afidd
{
  namespace smv
  {
    std::pair<bool,std::unique_ptr<TransitionDistribution<RandGen>>>
      enabled(const BrownionGSPN& et, TransitionKey trans_id,
              const UserState& s, const Local& lm, double te, double t0)
      {
        if (lm.template length<0>(0)>0)
          {
            // This is where we choose the distributions for the two
            // Brownion states.
            if (trans_id.from.mobility==Mobility::mobile)
              return {true, std::unique_ptr<Weibull>(new Weibull(6.0, 3.0, te))};
            else
                return {true, std::unique_ptr<Dist>(new ExpDist(1.0, te))};
          }
        else
            return {false, std::unique_ptr<Dist>(nullptr)};
      }

    void
      fire(BrownionGSPN& et, TransitionKey trans_id, UserState& s, Local& lm, RandGen& rng)
    {
      lm.template move<0,0>(0, 1, 1);
    }

    std::vector<std::tuple<place_t<BrownionGSPN>,size_t,int>>
      neighbors_of_transition(BrownionGSPN& g, trans_t<BrownionGSPN> trans_id)
    {
      return {std::make_tuple(trans_id.from, 0, -1),
              std::make_tuple(trans_id.to, 0, 1)} ;
    }

    template<typename F>
      void neighbors_of_places(BrownionGSPN& g,
                               const std::set<place_t<BrownionGSPN>>& place_id,
                               const F& func)
      {
        for (auto p : place_id)
          {
            // Transitions that start at this place.
            if (p.mobility == Mobility::mobile)
            {
              func(TransitionKey{p, {p.x, p.y-1, p.mobility}}) ;
              func(TransitionKey{p, {p.x, p.y+1, p.mobility}}) ;
              func(TransitionKey{p, {p.x-1, p.y, p.mobility}}) ;
              func(TransitionKey{p, {p.x+1, p.y, p.mobility}}) ;
```

```
            // Transitions that end at this place.
            func(TransitionKey{{p.x+1, p.y, p.mobility}, p}) ;
            func(TransitionKey{{p.x-1, p.y, p.mobility}, p}) ;
            func(TransitionKey{{p.x, p.y+1, p.mobility}, p}) ;
            func(TransitionKey{{p.x, p.y-1, p.mobility}, p}) ;

            // Transition to immobile
            func(TransitionKey{p, {p.x, p.y, Mobility::immobile}}) ;
          }
          else
          {
            // Transition to mobile
            func(TransitionKey{p, {p.x, p.y, Mobility::mobile}}) ;
          }
        }
      }
    } // smv
  } // afidd
```

7. Model instantiation

⟨AFIDD *headers*⟩+≡

```
#include "continuous_state.h" // Needed for GSPNState?
#include "continuous_dynamics.h" // Needed for propagate_competing_processes()
```

⟨*Set up model*⟩≡

```
BOOST_LOG_TRIVIAL(info) << "Using " << vm["seed"].as<boost::uint32_t>() << " as seed" ;
RandGen rng(vm["seed"].as<boost::uint32_t>()) ;


using BrownionState = afidd::smv::GSPNState<Mark,UserState>;
```

Create partial core matrix using the GSPN.

⟨*Set up model*⟩+≡

```
using SemiMarkovKernel = afidd::smv::PartialCoreMatrix<BrownionGSPN, BrownionState, RandGen> ;
```

Define initial marking.

The initial marking corresponds to one immobile walker at the origin. Thus, the first transition will be to switch the mobility status.

⟨*Set up model*⟩+≡

```
auto initialize_walkers=[](BrownionState& s)->void
{
  afidd::smv::add<0>(s.marking, PlaceKey{0,0,Mobility::immobile}, Walker()) ;
} ;
```

8. Time evolution

The loop over timesteps is simple. The (next) state of the system computed by calling `propagate_competing_processes` and the time increment is used to update the total elapsed time. The `reporter` function is trivial because the output is generated in the main loop.

⟨*Loop over steps*⟩≡

```
auto reporter=[](BrownionState& s) -> void
{
  // Intentionally empty
} ;

for (counter_type repl=0 ; repl < vm["replicates"].as<counter_type>() ;
     repl++)
{
  BOOST_LOG_TRIVIAL(info) << "Working on replicate " << repl ;

  BrownionGSPN gspn ;
  BrownionState state ;
  SemiMarkovKernel Q(gspn, state) ;
  auto next = afidd::smv::propagate_competing_processes(Q, initialize_walkers, rng) ;

  real_type elapsed_time = 0.0 ;

  for (counter_type tcount=0 ; tcount<vm["transitions"].as<counter_type>(); tcount++)
  {
    auto tk = std::get<0>(next) ;
    auto residence_time = std::get<1>(next) ;
    elapsed_time += residence_time ;

    BOOST_LOG_TRIVIAL(info) << "replicate: " << repl
                            << ", transition: " << tcount
                            << ", " << tk
                            << ", residence time: " << residence_time
                            << ", elapsed time: " << elapsed_time ;

    os << repl << ", " << tcount << ", "
       << residence_time << ", " << elapsed_time << ", "
       << tk.from.x << ", " << tk.from.y << ", \"" << tk.from.mobility  << "\""
       <<  std::endl ;

    next = afidd::smv::propagate_competing_processes(Q, reporter, rng);
  }
}
```

References

[1] G. H. Weiss, "The two-state random walk," *Journal of Statistical Physics*, vol. 15, no. 2, pp. 157–165, 1976.

E-mail address: `Dave.Schneider@ars.usda.gov`

USDA Agricultural Research Service, Plant-Microbe Interaction Research Unit, R.W. Holley Center, Tower Road, Ithaca NY 14853

Department of Plant Pathology and Plant-Microbe Biology, Cornell University, Tower Road, Ithaca NY 14853