NEST models and NESTML

1 Proposed algorithm

Here is the principle of the systematic algorithm I propose for neuronal model implementation:

- functions used for normal models are in black,
- functions necessary for precise spikes are in red, ending with `_ps` (replace the usual one),
- functions necessary for model with interpolation are in yellow, ending with `_i` (they are also used by precise models).

```
void update( const Time& origin, const long_t from, const long_t to )
{
  usual_asserts();
  test_superthreshold_i();

  for ( long_t lag = from; lag < to; ++lag ) {
    decrease_refractoriness();

    while ( t < timestep ) {
      store_previous_values_i();

      get_next_event_i();
      integrate();
      test_refractoriness();

      test_spike() or test_spike_i();
      add_spikes_ps();
    }
    add_spikes();
    add_currents();
  }
}
```

Listing 1: Update function.

```cpp
void
nest::aeif_cond_alpha_gridprecise::update( ... )
{
  // usual assertions and variable declarations

  /* If interpolation
     Neurons may have been initialized to superthreshold potentials.
     We need to check for this here and issue spikes at the beginning of
     the interval.

  if ( S_.y_[ State_::V_M ] >= P_.V_peak_ )
  {
    S_.y_[ State_::V_M ] = P_.V_reset_;
    S_.y_[ State_::W ] += P_.b;
    SpikeEvent se;
    kernel().event_delivery_manager.send( *this, se, from );
  }
  */




  for ( long_t lag = from; lag < to; ++lag )
  {
    t = 0.;

    if ( S_.r_ > 0 )
      --S_.r_;

    while ( t < B_.step_ )
    {
      /* If interpolation: store the previous values of the state variables and t
      std::copy(S_.y_, S_.y_ + sizeof(S_.y_)/sizeof(S_.y_[0]), S_.y_old_); t_old = t;
      // check for end of refractory period
      if ( P_.t_ref_ > 0. && S_.r_ == 0 && t < S_.r_offset_ )
        t_next_event = S_.r_offset_;
      else
        t_next_event = B_.step_; */

      // ODE propagation (here using GSL)
      while (t < t_next_event)
      {
        const int status = gsl_odeiv_evolve_apply( B_.e_,
          B_.c_,
          B_.s_,
          &B_.sys_,             // system of ODE
          &t,                   // from t
          t_next_event,         // to t <= t_next_event
          &B_.IntegrationStep_, // integration step size
          S_.y_ );              // neuronal state

        // checks
        if ( status != GSL_SUCCESS )
          throw GSLSolverFailure( get_name(), status );
        if ( S_.y_[ State_::V_M ] < -1e3 || S_.y_[ State_::W ] < -1e6 || S_.y_[ State_::W ] > 1e6 )
          throw NumericalInstability( get_name() );
      }

      // check refractoriness
      if ( S_.r_ > 0 || S_.r_offset_ > 0. )
        S_.y_[ State_::V_M ] = P_.V_reset_; // only V_m is frozen
      else if ( S_.y_[ State_::V_M ] >= P_.V_peak_ )
      {
        /* If interpolation:
        interpolate_( t, t_old); */
        spiking_( lag, t );
      }

      /* reset refractory offset once refractory period is elapsed;
       * this cannot be done beforehand because of the previous check */
      if ( S_.r_ == 0 && std::abs(t - S_.r_offset_ ) < std::numeric_limits< double >::epsilon() )
        S_.r_offset_ = 0.;
    }

    // Here spike reception outside the loop
    S_.y_[ State_::DG_EXC ] += B_.spike_exc_.get_value( lag ) * V_.g0_ex_;
    S_.y_[ State_::DG_INH ] += B_.spike_inh_.get_value( lag ) * V_.g0_in_;







    // set new input current
    B_.I_stim_ = B_.currents_.get_value( lag );
    // log state data
    B_.logger_.record_data( origin.get_steps() + lag );
  }
}
```

Listing 2: Algorithm for “on-grid” spikes for and AEIF model with alpha-shaped conductances.

```cpp
void
nest::aeif_cond_alpha_ps::update( ... )
{
  // usual assertions and variable declarations

  // at start of slice, tell input queue to prepare for delivery
  if ( from == 0 )
    B_.events_.prepare_delivery();

  /* Neurons may have been initialized to superthreshold potentials.
     We need to check for this here and issue spikes at the beginning of
     the interval.
  */
  if ( S_.y_[ State_::V_M ] >= P_.V_peak_ )
  {
    S_.y_[ State_::V_M ] = P_.V_reset_;
    S_.y_[ State_::W ] += P_.b;
    SpikeEvent se;
    se.set_offset( B_.step_ * ( 1 - std::numeric_limits< double_t >::epsilon() ) );
    kernel().event_delivery_manager.send( *this, se, from );
  }

  for ( long_t lag = from; lag < to; ++lag )
  {
    const long_t T = origin.get_steps() + lag; t = 0.; t_next_event = 0.;

    if ( S_.r_ > 0 )
      --S_.r_;

    while ( t < B_.step_ )
    {
      // store the previous values of the state variables, and t
      std::copy(S_.y_, S_.y_ + sizeof(S_.y_)/sizeof(S_.y_[0]), S_.y_old_);
      t_old = t;

      // check for the next event
      B_.events_.get_next_event(T, t_next_event, spike_in, spike_ex, B_.step_ );


      // ODE propagation (here using GSL)
      while (t < t_next_event)
      {
        const int status = gsl_odeiv_evolve_apply( B_.e_,
          B_.c_,
          B_.s_,
          &B_.sys_,             // system of ODE
          &t,                   // from t
          t_next_event,         // to t <= t_next_event
          &B_.IntegrationStep_, // integration step size
          S_.y_ );              // neuronal state

        // checks
        if ( status != GSL_SUCCESS )
          throw GSLSolverFailure( get_name(), status );
        if ( S_.y_[ State_::V_M ] < -1e3 || S_.y_[ State_::W ] < -1e6 || S_.y_[ State_::W ] > 1e6 )
          throw NumericalInstability( get_name() );
      }

      // check refractoriness
      if ( S_.r_ > 0 || S_.r_offset_ > 0. )
        S_.y_[ State_::V_M ] = P_.V_reset_; // only V_m is frozen
      else if ( S_.y_[ State_::V_M ] >= P_.V_peak_ )
      {
        // spiking: find the exact threshpassing, then emit the spike
        interpolate_( t, t_old );
        spiking_( T, lag, t );
      }

      // reset refractory offset once refractory period is elapsed
      if ( S_.r_ == 0 && std::abs(t - S_.r_offset_ ) < std::numeric_limits< double >::epsilon() )
        S_.r_offset_ = 0.;



      // here spike reception inside the loop
      if (t == t_next_event)
      {
        S_.y_[ State_::I_EXC ] += spike_ex;
        S_.y_[ State_::I_INH ] += spike_in;
        spike_ex = 0.;
        spike_in = 0.;
      }
    }

    // set new input current
    B_.I_stim_ = B_.currents_.get_value( lag );
    // log state data
    B_.logger_.record_data( origin.get_steps() + lag );
  }
}
```

Listing 3: Algorithm for “off-grid” spikes (precise spike timing) in the case of an AEIF neuron with exponential post-synaptic currents.