

# 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).

---

```
1 void update( const Time& origin, const long_t from, const long_t to )
2 {
3     usual_asserts();
4     test_superthreshold_i();
5
6     for ( long_t lag = from; lag < to; ++lag ) {
7         decrease_refractoriness();
8
9         while ( t < timestep ) {
10             store_previous_values_i();
11
12             get_next_event_i();
13             integrate();
14             test_refractoriness();
15
16             test_spike() or test_spike_i();
17             add_spikes_ps();
18         }
19         add_spikes();
20         add_currents();
21     }
22 }
```

---

Listing 1: Update function.

```

1 void
2 nest::aeif_cond_alpha_gridprecise::update( ... )
3 {
4     // usual assertions and variable declarations
5
6     /* If interpolation
7      Neurons may have been initialized to superthreshold potentials.
8      We need to check for this here and issue spikes at the beginning of
9      the interval.
10
11     if ( S_.y_[ State_::V_M ] >= P_.V_peak_ )
12     {
13         S_.y_[ State_::V_M ] = P_.V_reset_;
14         S_.y_[ State_::W ] += P_.b;
15         SpikeEvent se;
16         kernel().event_delivery_manager.send( *this, se, from );
17     }
18     */
19
20     for ( long_t lag = from; lag < to; ++lag )
21     {
22         t = 0.;
23
24         if ( S_.r_ > 0 )
25             --S_.r_;
26
27         while ( t < B_.step_ )
28         {
29             /* If interpolation: store the previous values of the state variables and t
30             std::copy(S_.y_, S_.y_ + sizeof(S_.y_)/sizeof(S_.y_[0]), S_.y_old_); t_old = t;
31             // check for end of refractory period
32             if ( P_.t_ref_ > 0. && S_.r_ == 0 && t < S_.r_offset_ )
33                 t_next_event = S_.r_offset_;
34             else
35                 t_next_event = B_.step_; */
36
37             // ODE propagation (here using GSL)
38             while ( t < t_next_event )
39             {
40                 const int status = gsl_odeiv_evolve_apply( B_.e_,
41                     B_.c_,
42                     B_.s_,
43                     &B_.sys_, // system of ODE
44                     &t, // from t
45                     t_next_event, // to t <= t_next_event
46                     &B_.IntegrationStep_, // integration step size
47                     S_.y_ ); // neuronal state
48
49                 // checks
50                 if ( status != GSL_SUCCESS )
51                     throw GSLSolverFailure( get_name(), status );
52                 if ( S_.y_[ State_::V_M ] < -1e3 || S_.y_[ State_::W ] < -1e6 || S_.y_[
53                     State_::W ] > 1e6 )
54                     throw NumericalInstability( get_name() );
55             }
56
57             // check refractoriness
58             if ( S_.r_ > 0 || S_.r_offset_ > 0. )
59                 S_.y_[ State_::V_M ] = P_.V_reset_; // only V_m is frozen
60             else if ( S_.y_[ State_::V_M ] >= P_.V_peak_ )
61             {
62                 /* If interpolation:
63                 interpolate_( t, t_old); */
64                 spiking_( lag, t );
65             }
66
67             /* reset refractory offset once refractory period is elapsed;
68             * this cannot be done beforehand because of the previous check */
69             if ( S_.r_ == 0 && std::abs(t - S_.r_offset_) < std::numeric_limits< double
70                 >::epsilon() )
71                 S_.r_offset_ = 0.;
72         }
73
74         // Here spike reception outside the loop
75         S_.y_[ State_::DG_EXC ] += B_.spike_exc_.get_value( lag ) * V_.g0_ex_;
76         S_.y_[ State_::DG_INH ] += B_.spike_inh_.get_value( lag ) * V_.g0_in_;
77
78         // set new input current
79         B_.I_stim_ = B_.currents_.get_value( lag );
80         // log state data
81         B_.logger_.record_data( origin.get_steps() + lag );
82     }
83 }

```

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

```

1 void
2 nest::aeif_cond_alpha_ps::update( ... )
3 {
4     // usual assertions and variable declarations
5
6     /* at start of slice, tell input queue to prepare for delivery
7     if ( from == 0 )
8         B_.events_.prepare_delivery();
9
10     /* Neurons may have been initialized to superthreshold potentials.
11     We need to check for this here and issue spikes at the beginning of
12     the interval.
13
14     if ( S_.y_[ State_::V_M ] >= P_.V_peak_ )
15     {
16         S_.y_[ State_::V_M ] = P_.V_reset_;
17         S_.y_[ State_::W ] += P_.b;
18         SpikeEvent se;
19         se.set_offset( B_.step_ * ( 1 - std::numeric_limits< double_t >::epsilon() ) );
20         kernel().event_delivery_manager.send( *this, se, from );
21     }
22
23     for ( long_t lag = from; lag < to; ++lag )
24     {
25         const long_t T = origin.get_steps() + lag; t = 0.; t_next_event = 0.;
26
27         if ( S_.r_ > 0 )
28             --S_.r_;
29
30         while ( t < B_.step_ )
31         {
32             // store the previous values of the state variables, and t
33             std::copy(S_.y_, S_.y_ + sizeof(S_.y_)/sizeof(S_.y_[0]), S_.y_old_);
34             t_old = t;
35
36             // check for the next event
37             B_.events_.get_next_event(T, t_next_event, spike_in, spike_ex, B_.step_ );
38
39             // ODE propagation (here using GSL)
40             while ( t < t_next_event )
41             {
42                 const int status = gsl_odeiv_evolve_apply( B_.e_,
43                     B_.c_,
44                     B_.s_,
45                     &B_.sys_, // system of ODE
46                     &t, // from t
47                     t_next_event, // to t <= t_next_event
48                     &B_.IntegrationStep_, // integration step size
49                     S_.y_ ); // neuronal state
50
51                 // checks
52                 if ( status != GSL_SUCCESS )
53                     throw GSLSolverFailure( get_name(), status );
54                 if ( S_.y_[ State_::V_M ] < -1e3 || S_.y_[ State_::W ] < -1e6 || S_.y_[
55                     State_::W ] > 1e6 )
56                     throw NumericalInstability( get_name() );
57             }
58
59             // check refractoriness
60             if ( S_.r_ > 0 || S_.r_offset_ > 0. )
61                 S_.y_[ State_::V_M ] = P_.V_reset_; // only V_m is frozen
62             else if ( S_.y_[ State_::V_M ] >= P_.V_peak_ )
63             {
64                 // spiking: find the exact threshpassing, then emit the spike
65                 interpolate_( t, t_old );
66                 spiking_( T, lag, t );
67             }
68
69             // reset refractory offset once refractory period is elapsed
70             if ( S_.r_ == 0 && std::abs(t - S_.r_offset_) < std::numeric_limits< double
71                 >::epsilon() )
72                 S_.r_offset_ = 0.;
73
74             // here spike reception inside the loop
75             if ( t == t_next_event )
76             {
77                 S_.y_[ State_::I_EXC ] += spike_ex;
78                 S_.y_[ State_::I_INH ] += spike_in;
79                 spike_ex = 0.;
80                 spike_in = 0.;
81             }
82         }
83
84         // set new input current
85         B_.I_stim_ = B_.currents_.get_value( lag );
86         // log state data
87         B_.logger_.record_data( origin.get_steps() + lag );
88     }
89 }

```

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