MANCHESTER 1824
The University of Manchester

Numerical accuracy of the Izhikevich neuron model in fixed-point arithmetic

Mantas Mikaitis, PhD student @ University of Manchester, UK

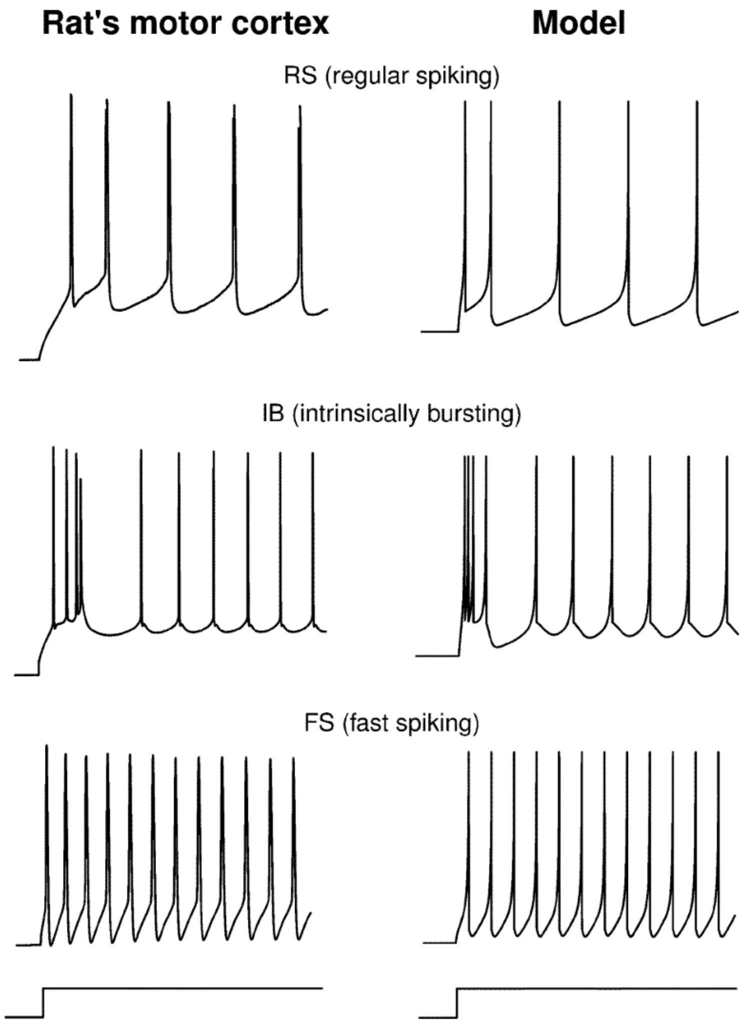
mantas.mikaitis@manchester.ac.uk

With: Michael Hopkins, Dave Lester, Steve Furber.

SpiNNaker team meeting
Manchester, 31 July 2019

SpiNNaker2
Universal Spiking Neural Network Architecture

Izhikevich neuron model: Cortical spiking patterns (Izhikevich 2003)



$$\frac{dV}{dt} = 0.04V^2 + 5V + 140 - u + I(t),$$

$$\frac{dU}{dt} = a(bV - U),$$

(On spike: $V = c$, $U = U + d$)

(RK2 Midpoint ODE solver, Hopkins & Furber, 2015)

$$\theta = 140 + I_{t+h} - U_t$$

$$\alpha = \theta + (5 + 0.04V_t)V_t$$

$$\eta = \frac{h}{2} + V_t$$

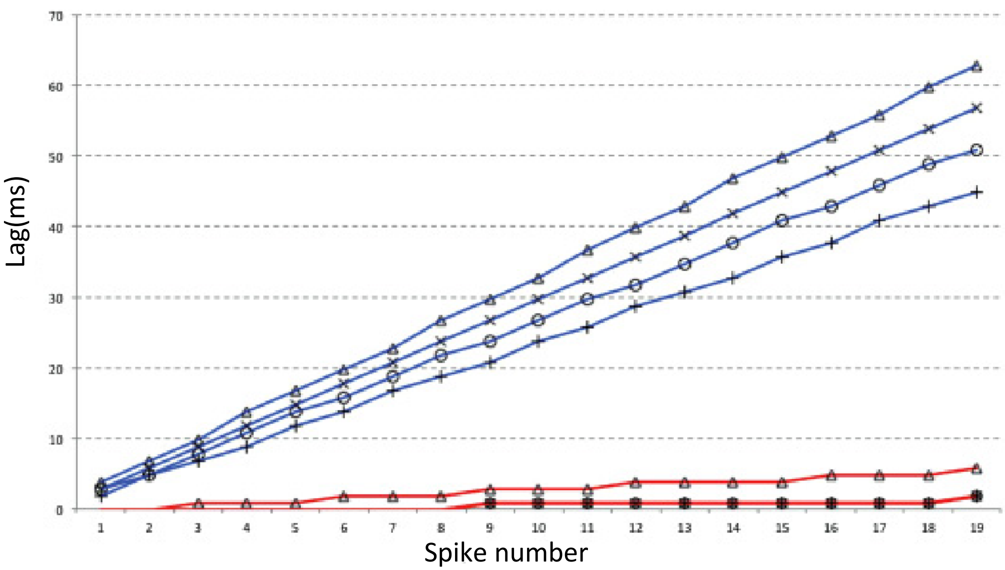
$$\beta = \frac{h(bV_t - U_t)}{2}$$

$$V(t+h) = V_t + h(\theta - \beta + (5 + 0.04\eta)\eta),$$

$$U(t+h) = U_t + ah(-U_t - \beta + b\eta).$$

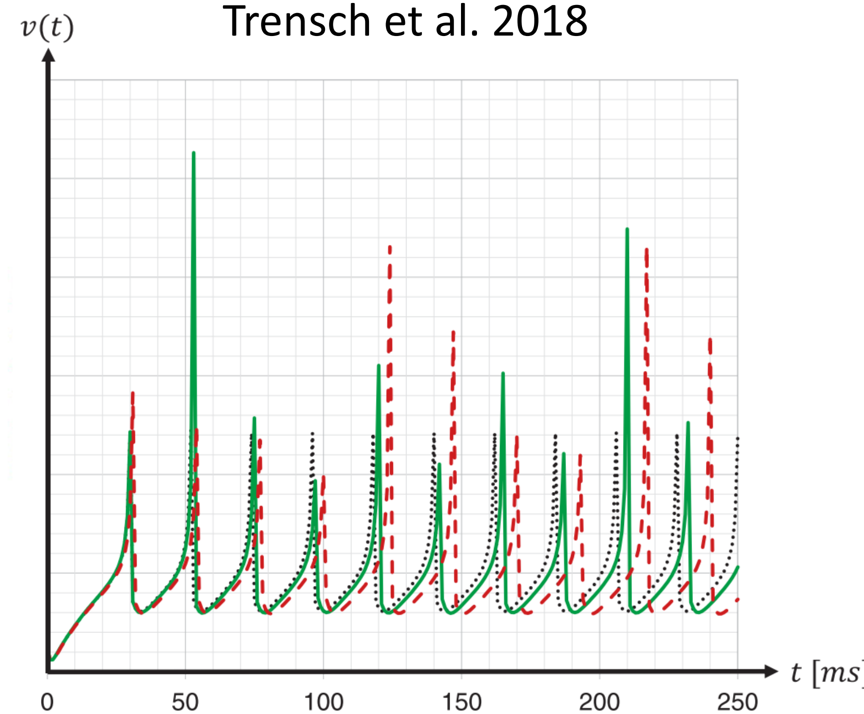
Challenges with fixed-point: spike timing lags on constant DC current

Hopkins & Furber, 2015



Blue: fixed-point. Red: float.
Timestep 0.1ms.

Trensch et al. 2018

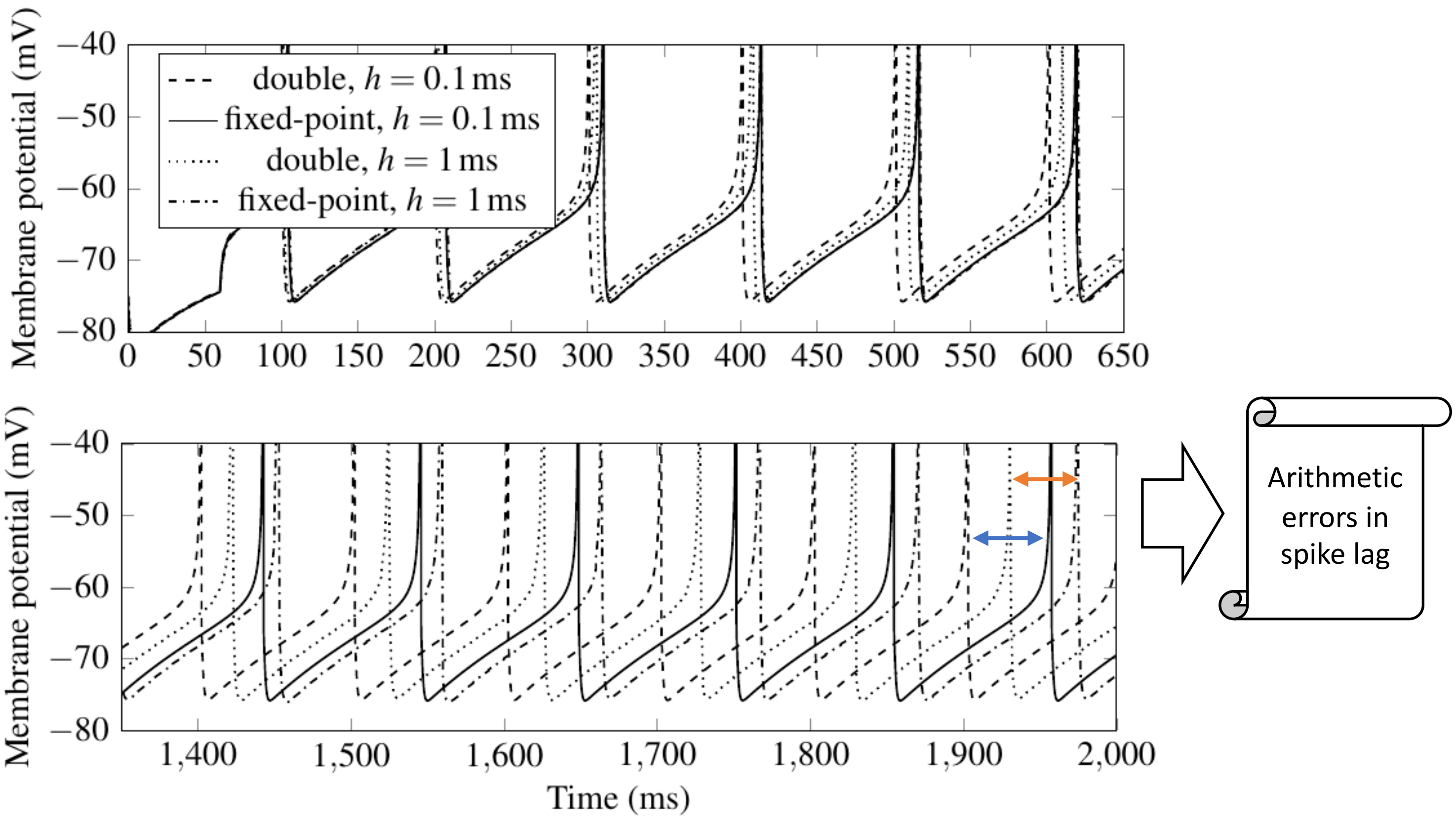


Dotted: double RKF45. Others: fixed-point Euler.
Timestep 1ms.

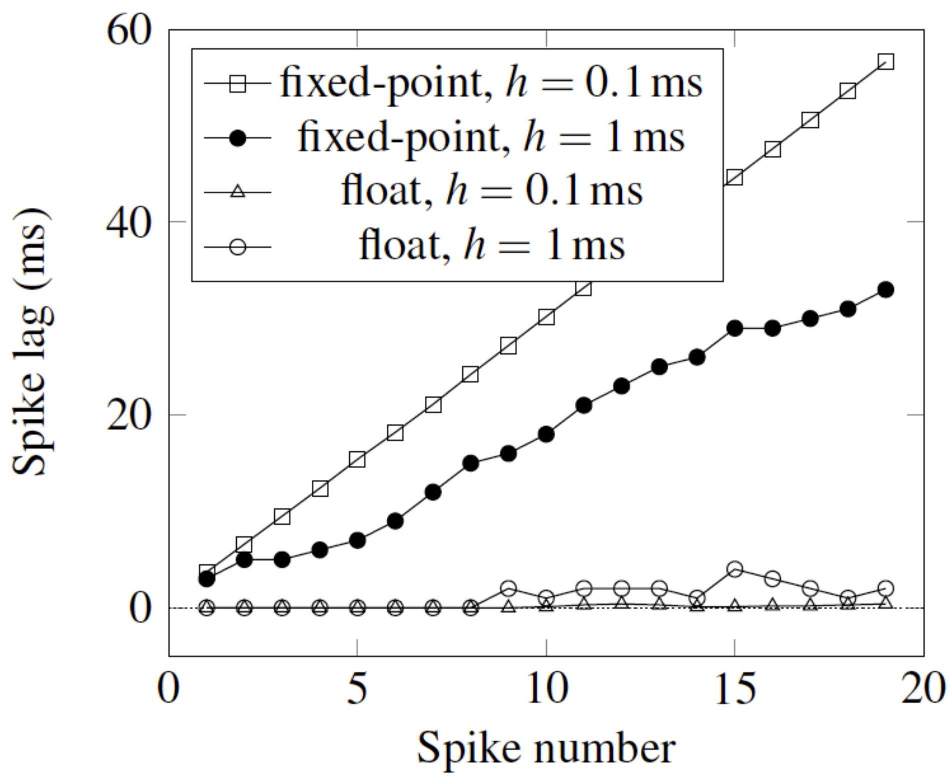
Measurement of error

- There are two types of error: *algorithmic* (ODE solver) and *arithmetic* (quantization, rounding, overflows).
- Choose a reference in such a way, that *algorithmic error* is removed from the comparison.
- Evaluate *spike lags* – how far each spike time is from the corresponding spike time in the *reference system.*
- This gives us a measurement to compare different arithmetics on this problem.

Traces from Michael's original test replicated on SpiNNaker (19 spikes in total, first and last 6 shown)

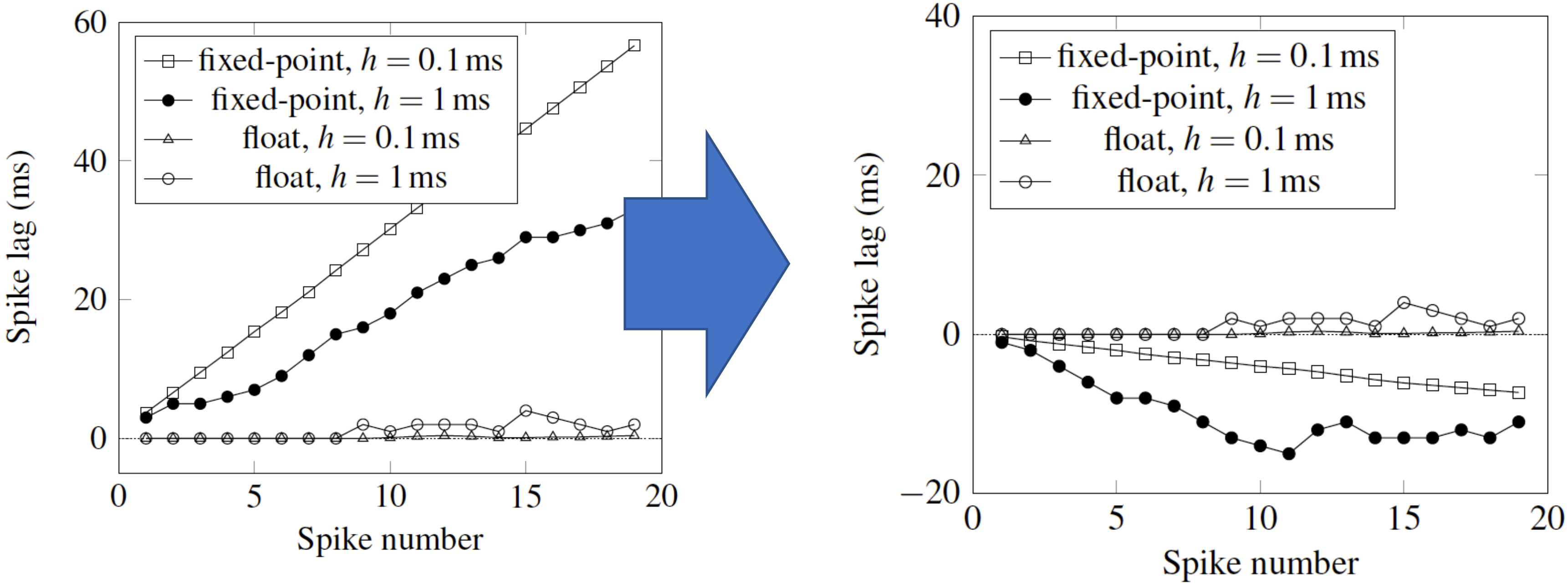


Spike lags from Michael's original test replicated on SpiNNaker (except reference is double precision)



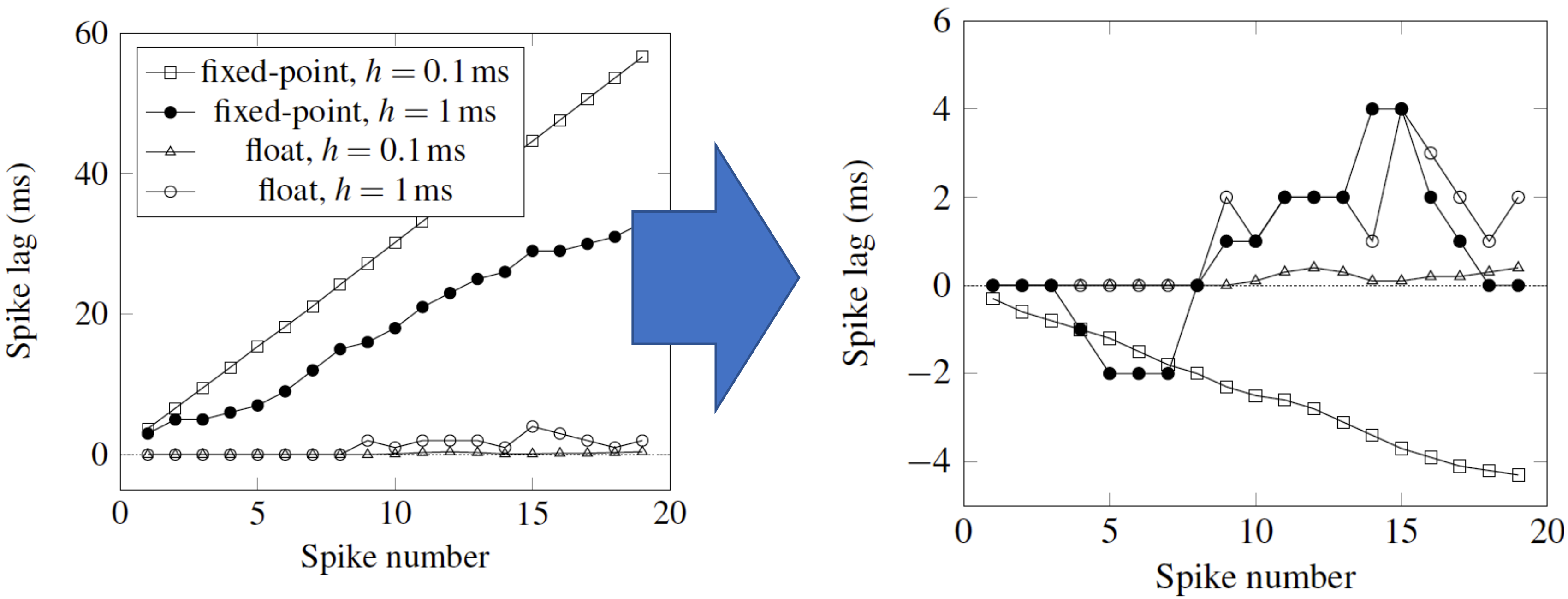
Correct rounding of constants

- Instead of writing the constant as 0.04k, round it to the nearest *accum* explicitly: 0.0400085449218 75k (**error of ~0.28ϵ).**
- **GCC rounds down by default, returning an error of 0.72ϵ (vague specification in the fixed-point ISO standard).**

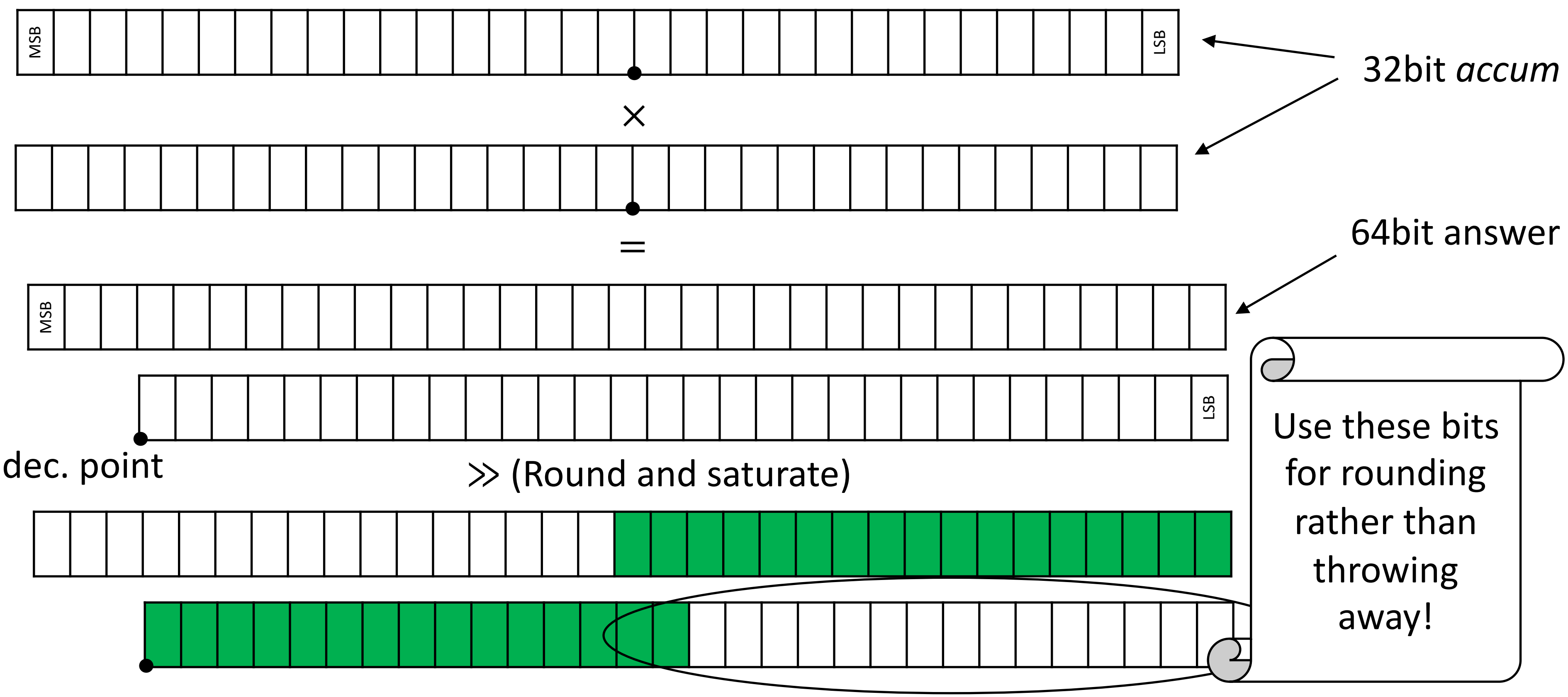


Mixed-precision multiplications

- This time, use *unsigned long fract* type, e.g. 0.040000000037252902984619140625ulr.
- Requires multiplications to be done as *mixed-format*.
- **Limited support in GCC and slow.**

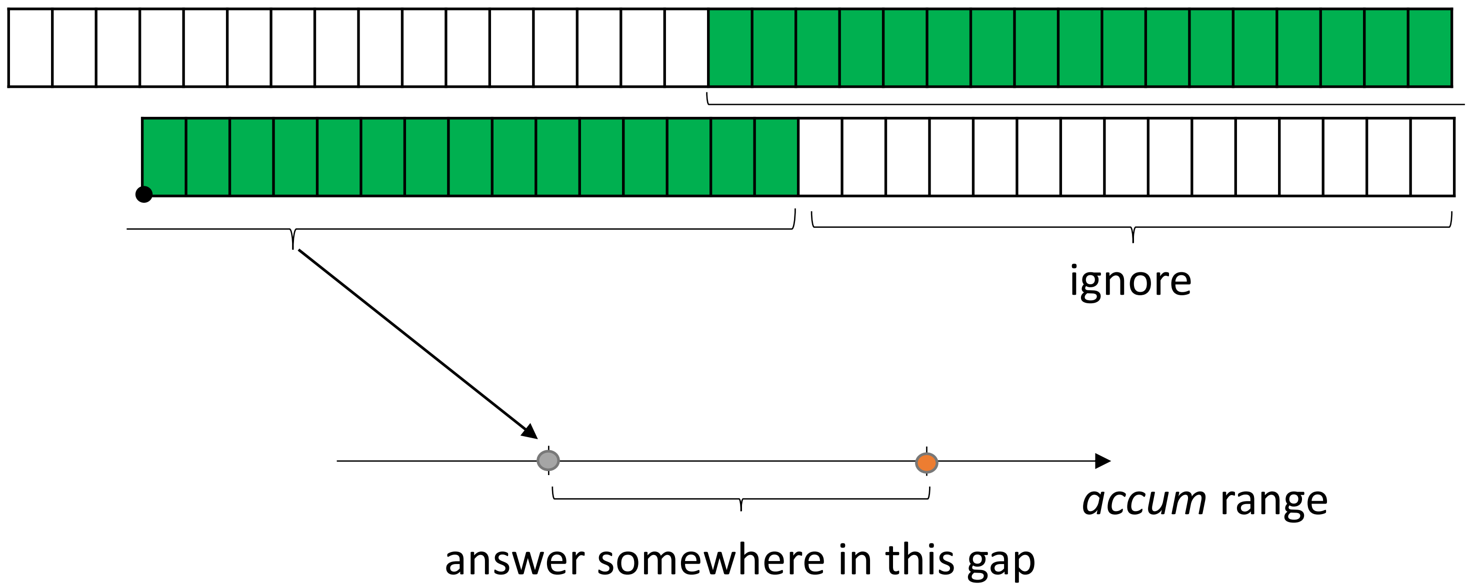


Fixed-point multiplier



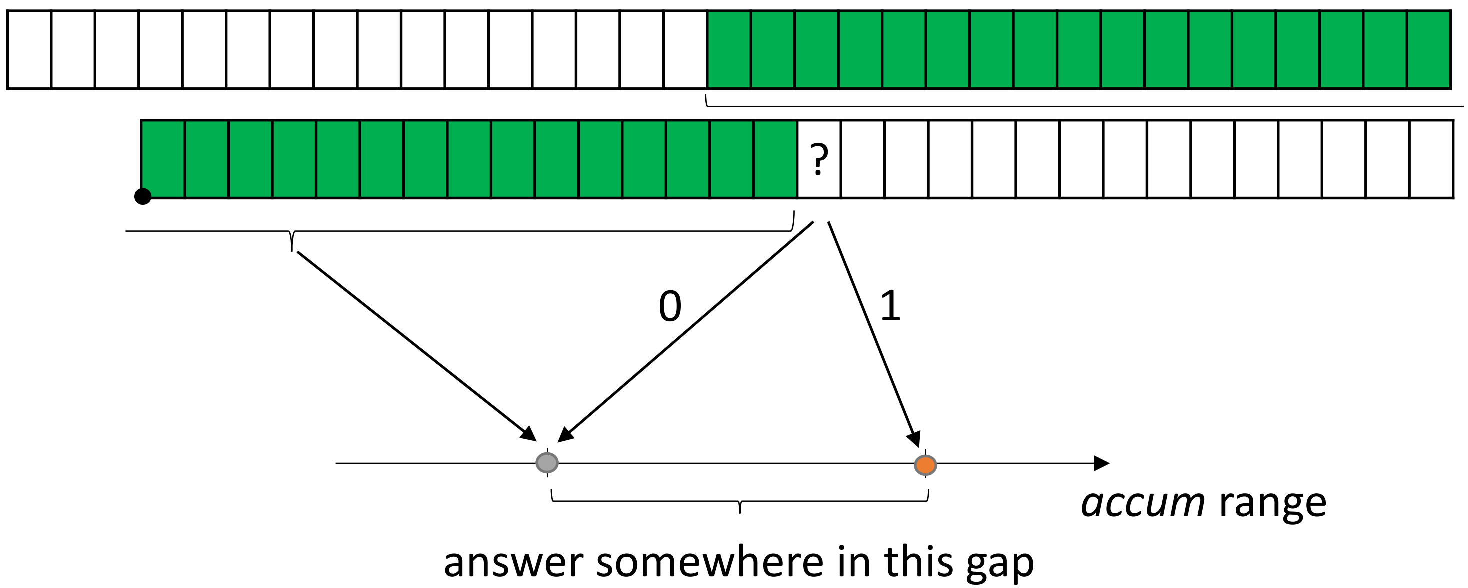
Round-down (RD)

Given the output from multiplier:



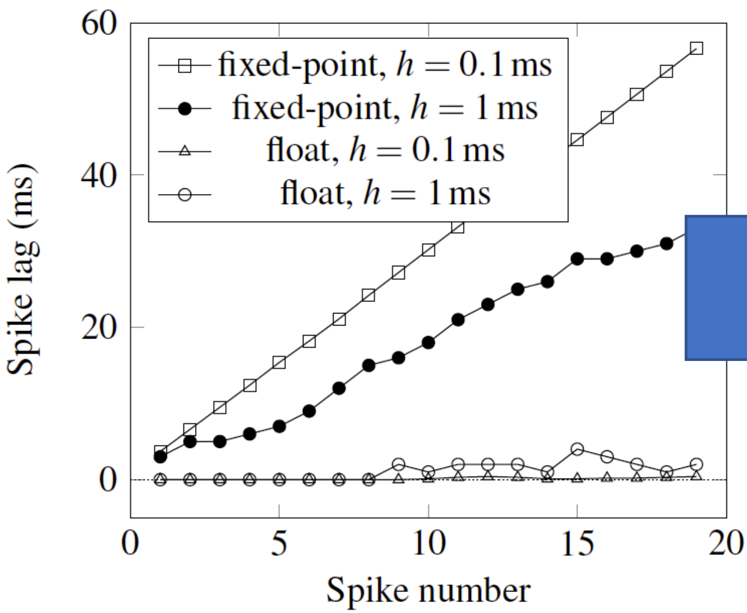
Round-to-nearest (RN)

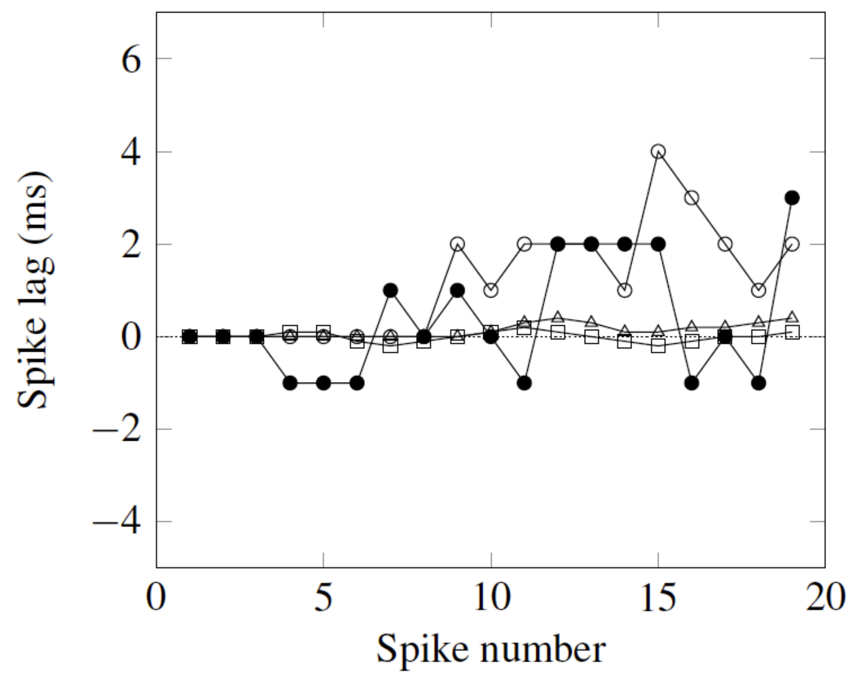
Given the output from multiplier:



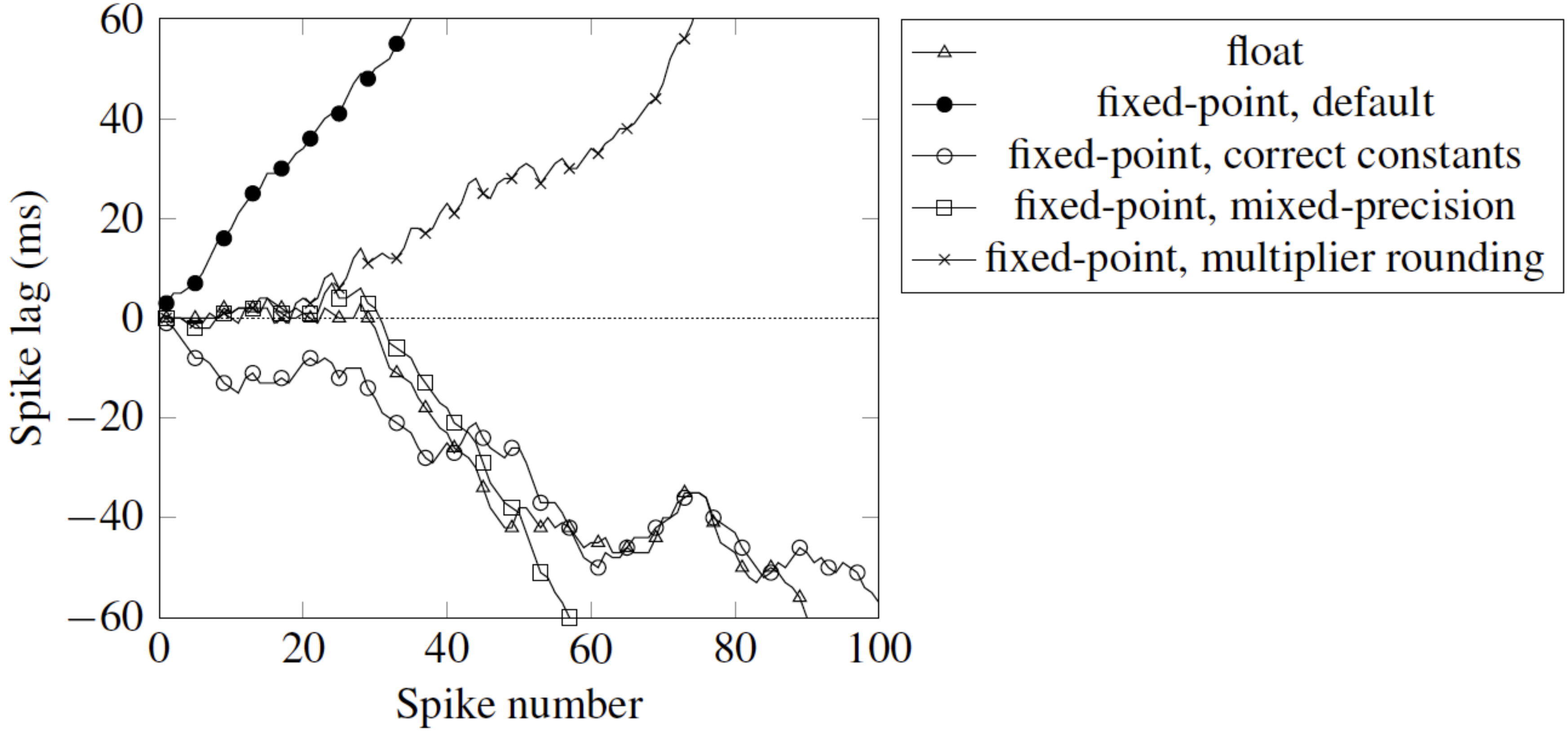
Rounding of multiplication results

- Use correct rounding of constants, mixed-format multiplications and do rounding on multiplications.

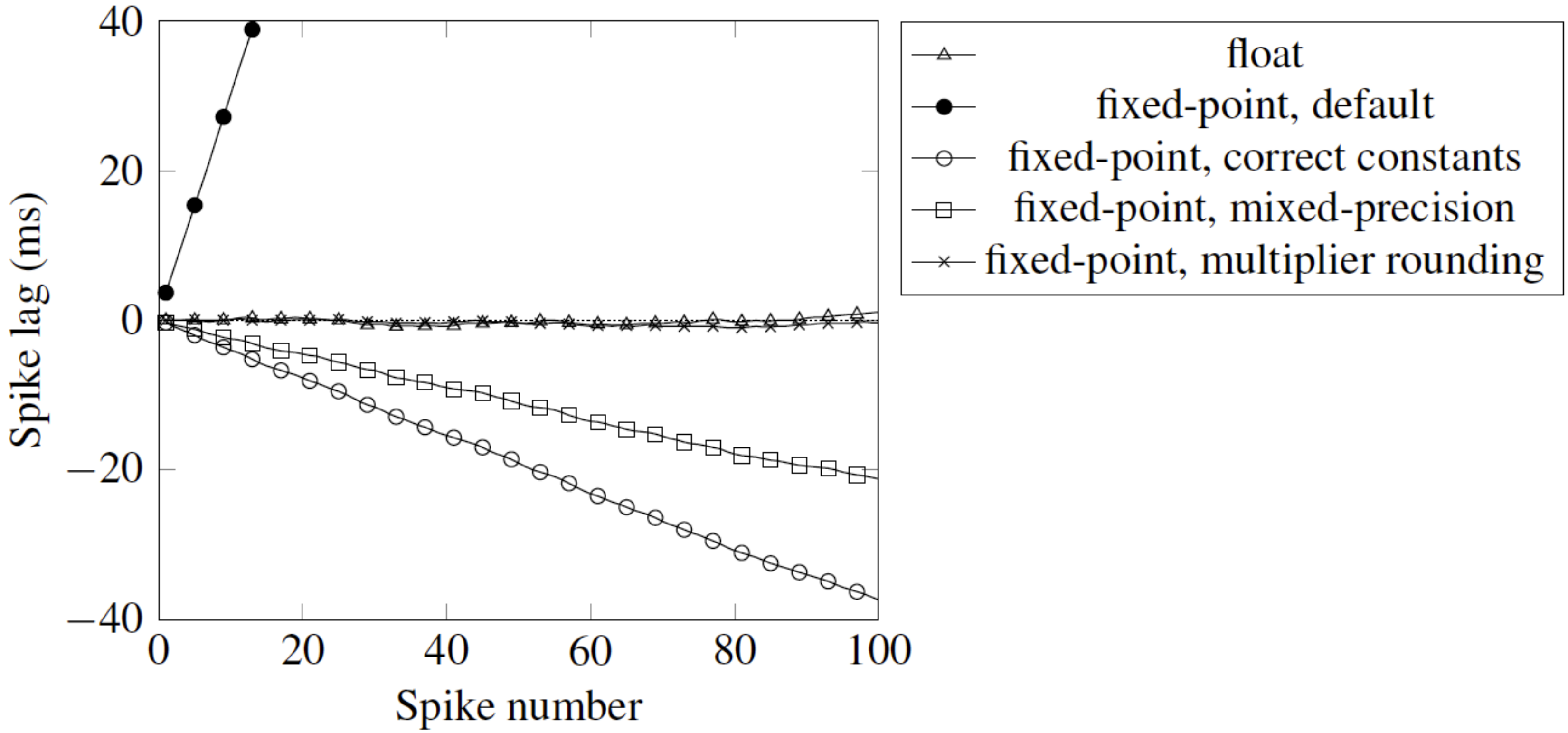




Summary so far; running longer (1ms timestep).



Summary so far; running longer (0.1ms timestep).



Performance

Arithmetic	Speed of ODE (μs)
software double	9.99203
software float	6.68132
fixed-point: default RK2 Midpoint, GCC	0.90881
fixed-point: mixed-precision multipliers, GCC	10.62621
fixed-point: default RK2 Midpoint, custom multipliers	1.86757
fixed-point: mixed-precision, custom multipliers	1.19345
fixed-point: mixed-precision, custom multipliers with RTN	1.59792

Table 1: Speed of RK2 Midpoint ODE solver integration step for different arithmetics, compiled with the *-Ofast* GCC compiler optimization flag.

NOTE: Custom multipliers are slightly more expensive because there is saturation check on them (DRL stdfix-full-iso.h extension).
NOTE: Mixed-format versions can contain more load instructions for constants (>12 bit immediates).

Stochastic rounding: a simple example

- Round to nearest(RTN):

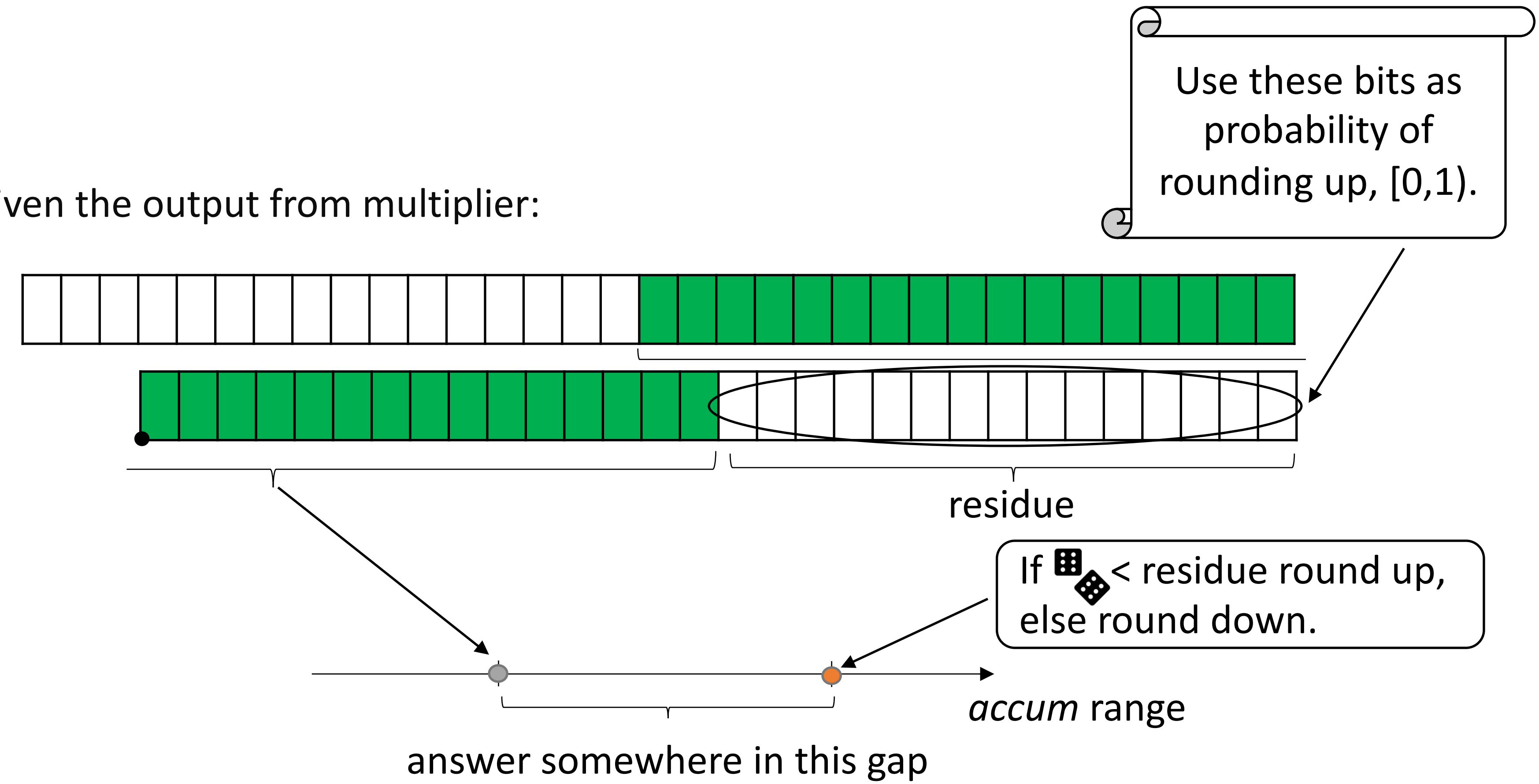
$$RN(0.25) + RN(0.25) + RN(0.25) + RN(0.25) = 0$$

- Stochastic round(SR):

$$SR(0.25) + SR(0.25) + SR(0.25) + SR(0.25) = likely\ 0\ or\ 1$$

Stochastic rounding (SR)

Given the output from multiplier:



SR algorithm

Algorithm 2 Stochastic rounding by addition

```
function SATSR_INT64_INT32(X, n)
    P ← PRNG32()
    P ← P&((1 << n) − 1)
    X ← (X + P) >> n
    if X > MAX_INT32 then
        return MAX_INT32
    if X < MIN_INT32 then
        return MIN_INT32
    return X
```

Harmonic series test: $\sum_{i=1}^{\infty} \frac{1}{i} = 1 + \frac{1}{2} + \frac{1}{3} \ldots$

NOTE: Calculate the addends as *unsigned long fract* and round before adding to the *accum* sum.

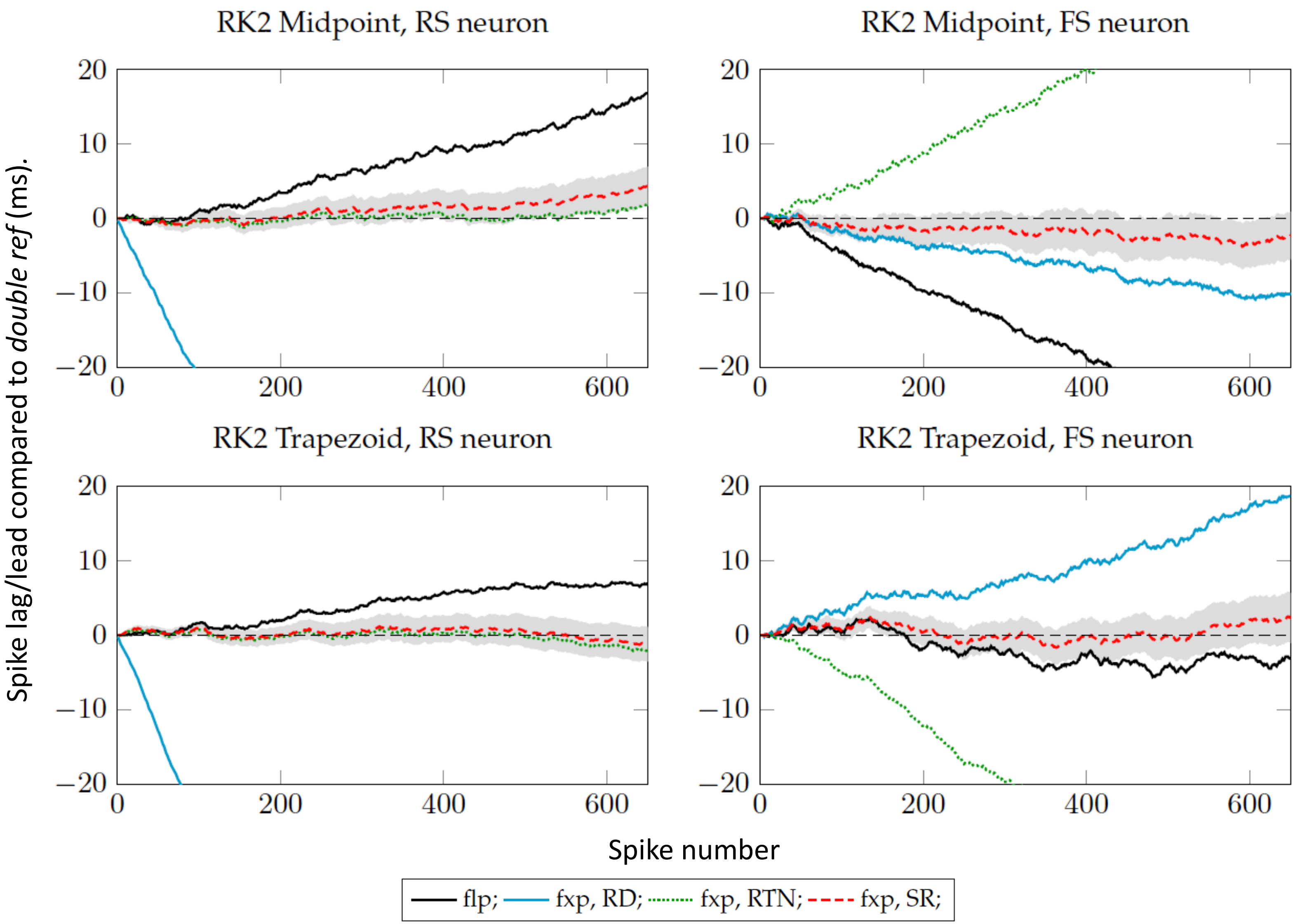
Arithmetic	Sum at $i = 5 \times 10^6$	Error at $i = 5 \times 10^6$	Iterations to converge
FP64	16.002	0	$2.81\ldots \times 10^{14}$
FP32	15.404	0.598	2097152
FP16	7.086	8.916	513
s16.15 RN	11.938	4.064	65537
s16.15 RD	10.553	5.449	32769
s8.7 RN	6.414	9.588	257
s8.7 RD	5.039063	10.963	129
s16.15 SR (50 runs)	$Mean = 16.002$ $std.dev = 0.012$	-0.000135765	-
s8.7 SR (50 runs)	$Mean = 11.205$ $std.dev = 0.242$	4.797	-

Table 2: Iterations until convergence of the harmonic series for different arithmetics. Sums and errors relative to FP64 (double precision floating-point) reported at 5 millionth iteration. Floating-point data from Higham and Pranesh [62]. Averaged sums from running the experiment 50 times in s16.15 and s8.7 SR arithmetics are also provided.

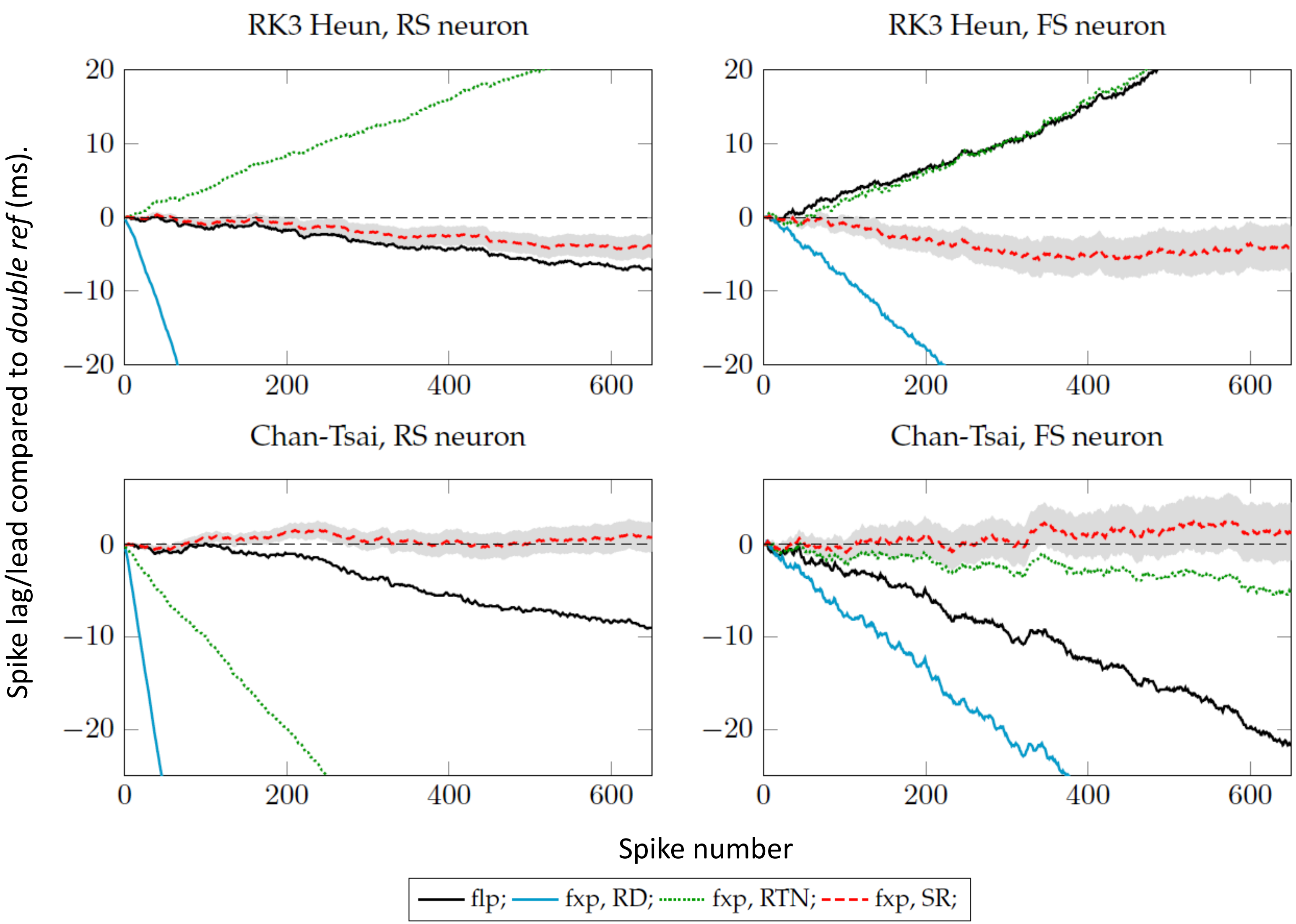
Testing method of Izhikevich ODE solutions

- Four ODE solvers: RK2 Midpoint, RK2 Trapezoid, RK3 Heun, Chan-Tsai
- Two different neuron types (regular and fast spiking - RS/FS)
- Five arithmetics: double (reference), float, fixed-point {round-down, round-to-nearest, stochastic}

Results: 2nd order solvers



Results: 3rd order solvers



Comment on readability of code: it is not necessarily only for bit level programmers!

- Anyone can experiment with different fixed-point types easily
- The only modification needed is instead of using “ * “ use a macro MULT(x, y) – this will call correct multiplications depending on numerical types of x and y and perform rounding specified in a different macro.

Example multiplication macro

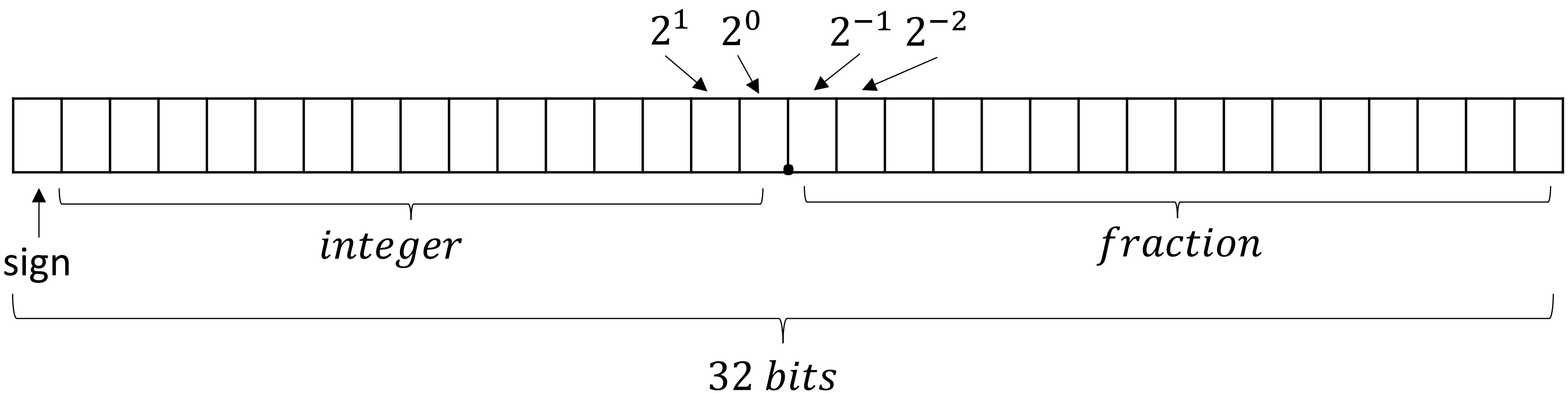
```
#define MULT_ROUND_NEAREST_ACCUM(x,y)                                          \
    ({                                                                         \
    __typeof__(x) temp0 = (x);                                                 \
    __typeof__(y) temp1 = (y);                                                 \
    REAL result;                                                               \
    if (__builtin_types_compatible_p(__typeof__(x), s1615) &&                  \
        __builtin_types_compatible_p(__typeof__(y), s1615)) {                  \
        result =  (kbits(__stdfix_smul_k_round_nearest(bitsk(temp0),          \
                  bitsk(temp1))));                                             \
    } else if ((__builtin_types_compatible_p(__typeof__(x), s1615) &&          \
               __builtin_types_compatible_p(__typeof__(y), s031)))  {          \
        result =  (accum_times_long_fract_nearest(temp0, temp1));              \
    } else if (__builtin_types_compatible_p(__typeof__(x), s031) &&            \
               __builtin_types_compatible_p(__typeof__(y), s1615)) {           \
        result = (accum_times_long_fract_nearest(temp1, temp0));               \
    } else if ((__builtin_types_compatible_p(__typeof__(x), s1615) &&          \
               __builtin_types_compatible_p(__typeof__(y), u032)))  {          \
        result =  (accum_times_u_long_fract_nearest(temp0, temp1));            \
    } else if (__builtin_types_compatible_p(__typeof__(x), u032) &&            \
               __builtin_types_compatible_p(__typeof__(y), s1615)) {           \
        result = (accum_times_u_long_fract_nearest(temp1, temp0));             \
    } else {                                                                   \
    }                                                                          \
    result;                                                                    \
    })
```

Summary

- I have shown how to remove arithmetic error from the fixed-point Izhikevich neuron model.
- Fixed-point arithmetic can perform as well as float in this case.
- Stochastic rounding with 32-bit fixed-point arithmetic is almost equivalent to double.
- Both performance and accuracy of GCC fixed-point libraries is poor.

Thank you for listening! Questions?

(Optional) Fixed-point representation: *accum* type



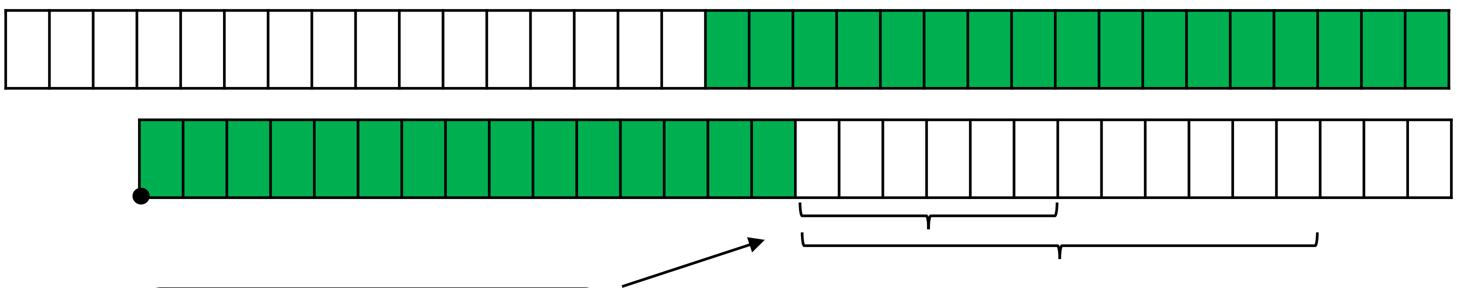
Fixed-point type in GCC: *accum* <s, 16, 15>:

$$\epsilon = 2^{-15} \approx 0.0000305176 \ldots$$

$$\text{Range: } [-2^{15} = -65536, 2^{16} - 2^{-15} \approx 65535.99996948 \ldots]$$

(Optional) Comparison of SR resolutions

Given the output from multiplier:



Use SOME of these bits as probability of rounding up, [0,1).