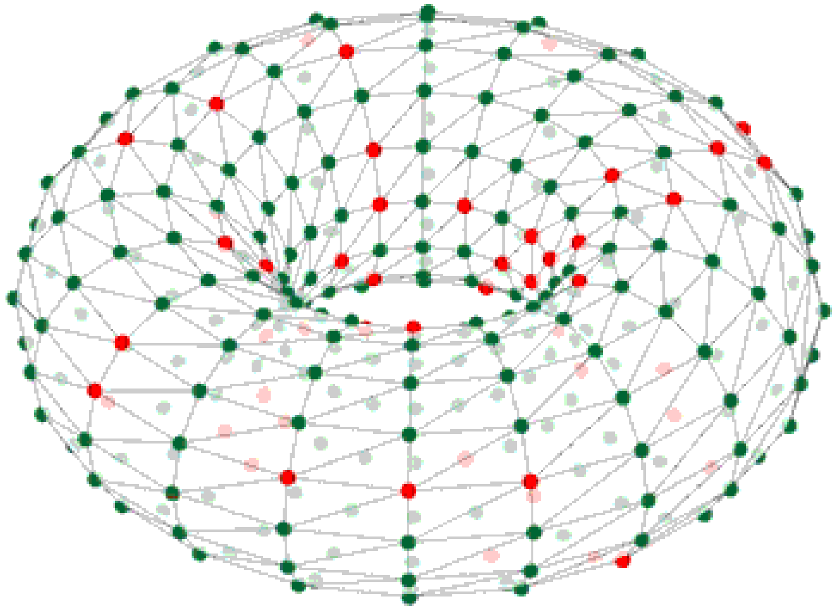
Fixed-Point Maths and Libraries



Michael Hopkins

SpiNNaker Workshop, September 2015

Overview

1. Numerical calculation on SpiNNaker
2. ISO/IEC 18037 types and operations
3. A simple example
4. Some practical considerations
5. Libraries currently available
6. An example using the libraries
7. Using fixed-point to solve ODEs
8. Future directions

Numerical calculation on SpiNNaker

- No floating point hardware on SpiNNaker
- Software floating point available but too slow for most use cases (and larger binaries)
- Until recently, has needed hand-coded fixed point types and manipulations
- This approach not transparent so can be prone to maintenance issues & mysterious bugs
- More difficult than necessary for developers to translate algorithms into source code
- ISO draft 18037 for fixed point types and operations seen as a good solution

ISO 18037 types and operations

- Draft standard for native fixed point types & operations used like integer or floating point
- Currently only available on GNU toolchain >= 4.7 and ARM target architecture
- 8-, 16-, 32 and 64-bit precisions all available in (un-)saturated and (un-)signed versions
- *accum* type is 32-bit 'general purpose real'; we support io_printf() with s16.15 & u16.16
- *fract* type is 16-bit in [0,1]; we support io_printf() with s0.15 & u0.16

Operations supported are:

- prefix and postfix increment and decrement operators (++, --)
- unary arithmetic operators (+, -, !)
- binary arithmetic operators (+, -, *, /)
- binary shift operators (<<, >>)
- relational operators (<, <=, >=, >)
- equality operators (==, !=)
- assignment operators (+=, -=, *=, /=, <<=, >>=)
- conversions to and from integer, floating-point, or fixed-point types

A simple example

```c
#include <stdfix.h>

#define REAL accum
#define REAL_CONST( x ) x##k

REAL  a, b, c = REAL_CONST( 100.001 );
accum d = REAL_CONST( 85.08765 );

int c_main( void )
{
   for( unsigned int i = 0; i < 50; i++ ) {

      a = i * REAL_CONST( 5.7 );

      b = a – i;

      if( a > d ) c  = a + b;
      else        c -= b;

      io_printf( IO_STD,
                 "\n i %u  a = %9.3k  b = %9.3k  c = %9.3k", i, a, b, c );
      }

   return 0;
}
```

Some practical considerations

- Range & precision e.g. for *accum* (s16.15) must have 0.000031 <= | x | <= 65536
- Still need to avoid divides in loops as these are slow on ARM architecture
- *saturated* types safe from overflow but significantly slower
- Need to remember that numerical precision is absolute rather than relative
- Literal constants require type suffix – simplest way is via macro REAL_CONST()
- Don't forget to `#include <stdfix.h>`
- Disciplined use of REAL and REAL_CONST() macros can parameterise entire code base
- Be careful to use the correct type suffix otherwise floating-point will be assumed

Libraries currently available - 1

1) *random.h* – suite of pseudo random number generators by MWH

Provides three high quality uniform generators of *uint32_t* values; Marsaglia's KISS 32 and KISS 64 and L'Ecuyer's WELL1024a.

- All three 'pass' the very stringent DIEHARD, dieharder and TestU01 test suites
- Trade-offs between speed, cycle length and equi-distributional properties
- Available in both simple-to-use form and with full user control over seeds

Have used these Uniform PRNGs as the basis for a set of Non-Uniform PRNGs including currently the following distributions:

- Gaussian
- Poisson (optimised for small rates at the moment)
- Exponential

...with more on the way. Let us know your requirements and we will try to help.

Libraries currently available - 2

2) *stdfix-full-iso.h* & *stdfix-math.h* – ISO & transcendental functions by DRL

Fill in the gaps in the GCC implementation of the ISO draft fixed point maths standard and some extensions:

- Standardised type conversions between fixed point representations
- Utility functions for all types i.e. abs(x), min(x), max(x), round(x), countls(x)
- Mechanism for automatically inferring the right argument type (uses GNU extension)

Fixed point replacements for essential floating point *libm* functions i.e. expk(x), sqrtk(x), logk(x), sink(x), cosk(x) and others such as atank(x), powk(x,y), 1/x on the way

- Hand-optimised for speed and accuracy on ARM architecture
- 10-30x faster than *libm* calls, hence feasible for use inside loops if necessary

An example using the libraries

```c
accum               a, b, c, d;
uint32_t            r1;
unsigned fract      uf1;

init_WELL1024a_simp();  // need to initialise WELL1024a RNG before use

for( unsigned int i = 0; i < 22; i++ ) {

    r1 = WELL1024a_simp();                      // draw from Uniform RNG

    uf1 = (unsigned fract) ulrbits( r1 );       // convert to unsigned fract

// draw from Std Gaussian distribution using MARS64
    a = gaussian_dist_variate( mars_kiss64_simp, NULL );

// do some calculations on a and then log()
    b = logk( absk( a * REAL_CONST( 100.0 ) ) );

// sqrt() of value drawn from Exponential distribution using WELL1024a
    c = sqrtk( exponential_dist_variate( WELL1024a_simp, NULL ) );

    d = expk( (accum) ( i – 10 ) );         // exp() from -10 to 11

    io_printf( IO_STD, "\n i %4u
      uf1=[Uniform{*}]= %8.6R  a=[Gauss{*}]= %7.3k b=[ln(abs(100 a))]= %7.3k
      c=[sqrt(Exponential{*})]= %7.3k  d=[exp(i-10)]= %10.3k ", i, uf1, a, b, c, d );

    }
```

Using fixed-point to solve ODEs - 1

- Simulating neuron models usually means solving Ordinary Differential Equations (ODEs)

- This ranges from very easy (current input LIF has simple closed-form) solution to very challenging i.e. Hodgkin-Huxley with 4 state variables, nonlinear and very 'stiff' ODE

- Numerical calculations are required with a balance between accuracy & efficiency

- With care and attention to detail, fixed-point can be used to get very close to floating-point results. However, models with more complex behaviour are a significant challenge

- A new approach called *Explicit Solver Reduction* (ESR) makes this easier in many cases and is described in an upcoming paper: Hopkins & Furber (2015), “Accuracy and Efficiency in Fixed-Point Neural ODE Solvers” , *Neural Computation* **27**, 1–35

- Good results found for Izhikevich neuron at real-time simulation speed & 1 ms time step

Using fixed-point to solve ODEs - 2

```
/*
   ESR algebraic reduction of the combination of Izhikevich neuron model and
   Runge-Kutta 2nd order midpoint method.  Hand-optimised interim variables and
   arithmetic ordering for balance between speed and accuracy.  See Neural Computation
   paper for more details.
*/
static inline void _rk2_kernel_midpoint( REAL h, neuron_pointer_t neuron,
                                          REAL input_this_timestep ) {
// to match Mathematica names
    REAL lastV1 = neuron->V;
    REAL lastU1 = neuron->U;
    REAL a = neuron->A;
    REAL b = neuron->B;

// generate common interim variables
    REAL pre_alph = REAL_CONST(140.0) + input_this_timestep – lastU1;
    REAL alpha = pre_alph
                 + ( REAL_CONST(5.0) + REAL_CONST(0.0400) * lastV1 ) * lastV1;
    REAL eta = lastV1 + REAL_HALF( h * alpha );

// could be represented as a long fract but need efficient mixed-arithmetic functions
    REAL beta = REAL_HALF( h * ( b * lastV1 – lastU1 ) * a );

// update neuron state
    neuron->V += h * ( pre_alph – beta
                      + ( REAL_CONST(5.0) + REAL_CONST(0.0400) * eta ) * eta );

    neuron->U += a * h * ( -lastU1 – beta + b * eta );
}
```

Future directions

- Optimise operations on differing fixed point types i.e. *accum * long fract*
- Add to *stdfix-math* (e.g. new argument types and special functions)
- Add to *random* (e.g. longer cycle uniform PRNG and more non-uniform distributions)
- New libraries such as probability distributions to allow Bayesian inference tools
- io_printf() to be extended to more types such as *long fract*, *unsigned long fract*
- Linear Algebra operations such as matrix multiply, SVD and other decompositions
- SpiNNaker architecture potentially good choice for massively parallel algorithms e.g. MCMC