approximations

Claude Heiland-Allen

2015–2018

Contents

1 README.md

Approximations
==========

The problem

Mathematical functions like `cos()` can be expensive to evaluate accurately. Sometimes high precision isn't necessary, and a cheap to compute approximation will do just as well.

Symmetry and range reduction

The cosine function oscillates smoothly between `1` and `-1`. It is even:

```
cos(-x0) = cos(x)
```

It also has another symmetry:

```
cos(pi - x) = -cos(x)
```

And it is periodic:

```
cos(x + 2 pi) = cos(x)
```

Irrational numbers like `pi` are a pain for computers, so for ease of implementation we'll define a `cos` variant that has a period of `1` instead of `2 pi`.

Due to the symmetries it's only necessary to approximate `1/4` of the waveform. The cosine is closely related to the sine, they are out of phase by `pi/2`:

```
sin(x + pi/2) = cos(x)
```

Given an `x`, first make it positive (using the evenness of `cos`):

```
x := |x|
```

Then reduce it into the range `[0..1]` by keeping the fractional part (using the periodicity of our modified `cos`):

```
x := x - floor(x)
```

Now we can fold over the symmetry line at `1/2` (originally `pi`):

```
x := |x - 1/2|
```

Our `x` is now in `[0..1/2]`, and its cosine should be `-1` at `x = 0` and `1` at `x = 1/2`. The halves are a bit awkward, it'd be nicer if the cosine was `-1` at `x = -1` and `1` at `x = 1`. So:

```
x := 4 x - 1
```

The prevoius steps could be combined into one, as in the C99 source code file `cosf9.h`, at the start of the function `cosf9_unsafe()`:

```
x := |4 x - 2| - 1
```

Polynomial approximation

We now have `x` in `[-1..1]` and the desired cosine of the original value in the same range. The symmetries of cosine mean we have an odd function now, with:

```
f(-x) = -f(x)
```

A polynomial is a sum of powers of a variable, and it's fairly
obvious that an odd polynomial can have only odd powers. Let's write

```
f(x) = a x + b x^3 + c x^5 + d x^7 + ...
```

Which are the best values of `a, b, c, d, ...` to pick to best
approximate the desired function? Given that it's odd, it's pointless
to consider both positive and negative inputs (or zero), so positive
it is. We know that `f(1) = 1`, which gives the contraint:

```
1 = a + b + c + d + ...
```

If the function `f` matches the desired curve, then its slope will
match too. The derivative of `sin` is `cos`, and of `cos` is `-sin`,
and of the power `x^n` is `n x^(n-1)`. Our function `f` happens to
be `sin(x pi / 2)` (exercise: demonstrate it), and by the chain rule
and linearity of differentiation its derivative at `0` is `pi/2`. So
we get another constraint:

```
pi/2 = a
```

That's an easy one to solve! The derivative at `1` is `0` (the top
of the waveform cycle is instantaneously horizontal), whcih gives:

```
0 = a + 3 b + 5 c + 7 d + ...
```

So far so good, but we have 3 linear equations and 4 unknowns, so
there's not enough information to find a unique solution. The second
derivative at `1` should be `pi^2/4`, so our final equation can be:

```
pi^2/4 = 6 b + 20 c + 42 d + ...
```

Putting into matrix form:

```
[  1 ,  1 ,  1 ,  1 ]  [ a ]      [   1    ]
[  1 ,  0 ,  0 ,  0 ]  [ b ]  ==  [  pi/2  ]
[  1 ,  3 ,  5 ,  7 ]  [ c ]  ==  [   0    ]
[  0 ,  6 , 20 , 42 ]  [ d ]      [ pi^2/4 ]
```

Luckily there are plenty of software packages for solving matrix
equations, I used GNU Octave (see the source code file `cosf7.m`,
which solves a `3x3` system with `a = pi/2` already substituted).

The answer that Octave gives is this:

```
[ a ]      [ + 1.5707963267948965580e+00 ]
[ b ]  ==  [ - 6.4581411791873211126e-01 ]
[ c ]  ==  [ + 7.9239255452774770561e-02 ]
[ d ]      [ - 4.2214643289391062808e-03 ]
```

Quality

Now we have an approximation. But we don't know how good it is.
A quick visual check is to plot the difference between our function

and the the high quality ‘cos()‘ in the standard C maths library. But without a frame of reference, the numbers aren’t necessarily all that meaningful.

The music software Pure-data contains its own approximation of the cosine function, using an entirely different technique: linearly interpolated table lookup. Plotting the error in its approximation on the same scale as our function reveals a pleasant surprise: our order 7 polynomial’s error deviates from the ‘0‘ axis (perfection) much less wildly than Pd’s. But the deviation has a different character, Pd’s is wildly swinging with a low frequency component, while ours has a hump. In the context of audio, these different variations might change the perception of the error (exercise: try double-blind listening tests to see which form is most obnoxious).

We can quantify the error in (at least) two ways, one is the maximum absolute deviation from perfection, and another is the root mean square (RMS) error, calculated by averaging the squares of the errors before taking the square root. Taking ‘2^30‘ randomly distributed phases and accumulating their error shows that the 7th order polynomial has about half the error as Pd’s table look up, in both measures.

To gain higher quality (lower error statistics), we can increase the order of the polynomial, by adding an extra ‘e x^9‘. We need another constraint to be able to solve the equations, and a reasonable choice seems to be to pick the worst phase (around ‘2/pi‘) on the 7th order polynomial and force it to be exact. This gives:

```
sin(1) = (2/pi) a + (2/pi)^3 b + (2/pi)^5 c + (2/pi)^7 d + (2/pi)^9 e
```

and we can solve the new ‘5x5‘ matrix equation to get different values for all the coefficients. See the GNU Octave source code file ‘cosf9.m‘ for details, and the C source code file ‘cosf9.h‘ for the coefficients.

The 9th order polynomial is about five times more accurate than the 7th order polynomial, getting close to the limits of the 24bit mantissa of single precision floating point.

Speed

It’s no point having a superb approximation if it’s not faster than the full precision version, so benchmarking and optimisation is the final part of the evaluation (but any changes to the code in the hope of making it faster might have an impact on the quality, in sometimes unexpected ways - so check the metrics and graphs on each change).

Unsurpisingly, the math library ‘cos()‘ is very slow, and the Pd cosine table is several times faster. But our polynomial approximations are faster still, with the 9th order not lagging far behind the 7th order. This took some effort, however. Putting too much into one inner loop puts pressure on the hardware, and performance can suffer. Something as counterintuitive as making two passes over the data, performing range reduction in the first pass and evaluating the polynomial in the second pass, can make a significant difference. Also important is splitting into smaller functions - with a smaller scope, the compiler

```
can do a better job of making it go really fast.

The first version (not shown in the graphs) was somewhat disappointing
- adding range reduction around the polynomial evaluation (which was
already fast enough) made it seven times slower, slower than Pd's table
implementation.  Splitting it into two passes solved that problem, and
then it turned out that with the functions split neatly, a single
pass could now do nearly as well as two.  See the source code
file `cosf9slow.h` to see what it was like before the changes.

Conclusion
----------

A 9th order polynomial approximation to the cosine function is about
ten times faster than the accurate math library implementation, and
about three times faster than linearly interpolated table lookup.
Moreover the polynomial is over ten times more accurate than the
table version.

--
https://mathr.co.uk
```

2 src/arithmetic.c

```
#include <math.h>

extern void addfs(int n, const float * restrict in1, const float * restrict in2,↙
    ↪ float * restrict out) {
  while (n--) {
    *out++ = *in1++ + *in2++;
  }
}

extern void subfs(int n, const float * restrict in1, const float * restrict in2,↙
    ↪ float * restrict out) {
  while (n--) {
    *out++ = *in1++ - *in2++;
  }
}

extern void mulfs(int n, const float * restrict in1, const float * restrict in2,↙
    ↪ float * restrict out) {
  while (n--) {
    *out++ = *in1++ * *in2++;
  }
}

extern void mulffs(int n, float in1, const float * restrict in2, float * ↙
    ↪ restrict out) {
  while (n--) {
    *out++ = in1 * *in2++;
  }
}
```

```
extern void divfs(int n, const float * restrict in1, const float * restrict in2,↙
    ↪  float * restrict out) {
  while (n--) {
    *out++ = *in1++ / *in2++;
  }
}

extern void absfs(int n, const float * restrict in, float * restrict out) {
  while (n--) {
    *out++ = fabsf(*in++);
  }
}

extern void cosfs(int n, const float * restrict in, float * restrict out) {
  while (n--) {
    *out++ = cosf(*in++);
  }
}

extern float minimumfs(int n, float minimum, const float * restrict in) {
  while (n--) {
    minimum = fminf(minimum, *in++);
  }
  return minimum;
}

extern float maximumfs(int n, float maximum, const float * restrict in) {
  while (n--) {
    maximum = fmaxf(maximum, *in++);
  }
  return maximum;
}

extern float maximumabssubfs(int n, float maximum, const float * restrict in1, ↙
    ↪ const float * restrict in2) {
  while (n--) {
    maximum = fmaxf(maximum, fabsf(*in1++ - *in2++));
  }
  return maximum;
}

extern float sumsqrsubfs(int n, const float * restrict in1, const float * ↙
    ↪ restrict in2) {
  float sum = 0;
  while (n--) {
    float d = *in1++ - *in2++;
    sum += d * d;
  }
  return sum;
}
```

3 src/arithmetic.h

```
#ifndef ARITHMETIC_H
#define ARITHMETIC_H 1

extern void addfs(int n, const float * restrict in1, const float * restrict in2,↙
```

```
    ↪ float * restrict out);
extern void subfs(int n, const float * restrict in1, const float * restrict in2,↙
    ↪ float * restrict out);
extern void mulfs(int n, const float * restrict in1, const float * restrict in2,↙
    ↪ float * restrict out);
extern void mulffs(int n, float in1, const float * restrict in2, float * ↙
    ↪ restrict out);
extern void divfs(int n, const float * restrict in1, const float * restrict in2,↙
    ↪ float * restrict out);
extern void absfs(int n, const float * restrict in, float * restrict out);
extern void cosfs(int n, const float * restrict in, float * restrict out);
extern float minimumfs(int n, float minimum, const float * restrict in);
extern float maximumfs(int n, float maximum, const float * restrict in);
extern float maximumabssubfs(int n, float maximum, const float * restrict in1, ↙
    ↪ const float * restrict in2);
extern float sumsqrsubfs(int n, const float * restrict in1, const float * ↙
    ↪ restrict in2);

#endif
```

4 src/cosf1.c

```
#include "cosf1.h"

#include <math.h>
#include <stdint.h>

#define COSTABCOUNT 14

static float cosf1_table[COSTABCOUNT][(1 << (COSTABCOUNT - 1)) + 1];

extern void cosf1_initialize(void) {
  for (int i = 0; i < COSTABCOUNT; ++i) {
    int size = 1 << i;
    double s = 6.283185307179586 / size;
    for (int k = 0; k <= size; ++k) {
      double x0 = s *  k;
      double p0 = cos(x0);
      cosf1_table[i][k] = p0;
    }
  }
}

static inline float cosf1(float phase, int index, float size, int mask) {
  float p = fabsf(phase) * size;
  int q = p;
  p -= q;
  int i = q & mask;
  float x1 = p;
  float c0 = cosf1_table[index][i];
  float c1 = cosf1_table[index][i + 1];
  return c0 + x1 * (c1 - c0);
}

static inline void cosf1_reduce(float phase, float size, int mask, float * ↙
    ↪ restrict out_phase, int * restrict out_index) {
  float p = fabsf(phase) * size;
```

```
  int q = p;
  p -= q;
  int i = q & mask;
  *out_phase = p;
  *out_index = i;
}

static inline float cosf1_unsafe(float p, int i, int index) {
  const float * cs = &cosf1_table[index][i];
  float x1 = p;
  float c0 = cs[0];
  float c1 = cs[1];
  return c0 + x1 * (c1 - c0);
}

extern void cosf1s1(int n, int index, const float * restrict in, float * ↙
    ↳ restrict out) {
  float size = 1 << index;
  int mask = (1 << index) - 1;
  while (n--) {
    *out++ = cosf1(*in++, index, size, mask);
  }
}

extern void cosf1s2(int n, int index, const float * restrict in, float * ↙
    ↳ restrict out, float * restrict tmp_phase, int * restrict tmp_index) {
  float size = 1 << index;
  int mask = (1 << index) - 1;
  float * restrict tmp2_phase = tmp_phase;
  int * restrict tmp2_index = tmp_index;
  int m = n;
  while (n--) {
    cosf1_reduce(*in++, size, mask, tmp_phase++, tmp_index++);
  }
  while (m--) {
    *out++ = cosf1_unsafe(*tmp2_phase++, *tmp2_index++, index);
  }
}
```

5 src/cosf1.h

```
#ifndef COSF1_H
#define COSF1_H 1

extern void cosf1_initialize(void);
extern void cosf1s1(int n, int index, const float * restrict in, float * ↙
    ↳ restrict out);
extern void cosf1s2(int n, int index, const float * restrict in, float * ↙
    ↳ restrict out, float * restrict tmp_phase, int * restrict tmp_index);

#endif
```

6 src/cosf2.c

```
#include "cosf2.h"
```

```
#include <math.h>
#include <stdint.h>

#define COSTABCOUNT 14

static float cosf2_table[COSTABCOUNT][1 << (COSTABCOUNT - 1)][2];

extern void cosf2_initialize(void) {
  for (int i = 0; i < COSTABCOUNT; ++i) {
    int size = 1 << i;
    double s = 6.283185307179586 / size;
    for (int k = 0; k < size; ++k) {
      double x0 = s *  k;
      double x1 = s * (k + 1);
      double p0 = cos(x0);
      double p1 = cos(x1);
      double m0 = p1 - p0;
      cosf2_table[i][k][0] = p0;
      cosf2_table[i][k][1] = m0;
    }
  }
}

static inline float cosf2(float phase, int index, float size, int mask) {
  float p = fabsf(phase) * size;
  int q = p;
  p -= q;
  int i = q & mask;
  float x1 = p;
  float c0 = cosf2_table[index][i][0];
  float c1 = cosf2_table[index][i][1];
  return c0 + x1 * c1;
}

static inline void cosf2_reduce(float phase, float size, int mask, float * ↙
    ↳ restrict out_phase, int * restrict out_index) {
  float p = fabsf(phase) * size;
  int q = p;
  p -= q;
  int i = q & mask;
  *out_phase = p;
  *out_index = i;
}

static inline float cosf2_unsafe(float p, int i, int index) {
  const float * cs = &cosf2_table[index][i][0];
  float x1 = p;
  float c0 = cs[0];
  float c1 = cs[1];
  return c0 + x1 * c1;
}

extern void cosf2s1(int n, int index, const float * restrict in, float * ↙
    ↳ restrict out) {
  float size = 1 << index;
  int mask = (1 << index) - 1;
  while (n--) {
```

```
    *out++ = cosf2(*in++, index, size, mask);
  }
}

extern void cosf2s2(int n, int index, const float * restrict in, float * ↙
    ↳ restrict out, float * restrict tmp_phase, int * restrict tmp_index) {
  float size = 1 << index;
  int mask = (1 << index) - 1;
  float * restrict tmp2_phase = tmp_phase;
  int * restrict tmp2_index = tmp_index;
  int m = n;
  while (n--) {
    cosf2_reduce(*in++, size, mask, tmp_phase++, tmp_index++);
  }
  while (m--) {
    *out++ = cosf2_unsafe(*tmp2_phase++, *tmp2_index++, index);
  }
}
```

7 src/cosf2.h

```
#ifndef COSF2_H
#define COSF2_H 1

extern void cosf2_initialize(void);
extern void cosf2s1(int n, int index, const float * restrict in, float * ↙
    ↳ restrict out);
extern void cosf2s2(int n, int index, const float * restrict in, float * ↙
    ↳ restrict out, float * restrict tmp_phase, int * restrict tmp_index);

#endif
```

8 src/cosf4.c

```
#include "cosf4.h"

#include <math.h>

#define COSTABCOUNT 9

static float cosf4_table[COSTABCOUNT][1 << (COSTABCOUNT - 1)][4];

extern void cosf4_initialize(void) {
  for (int i = 0; i < COSTABCOUNT; ++i) {
    int size = 1 << i;
    double s = 6.283185307179586 / size;
    for (int k = 0; k < size; ++k) {
      double x0 = s *  k;
      double x1 = s * (k + 1);
      double p0 =      cos(x0);
      double m0 = -s * sin(x0);
      double p1 =      cos(x1);
      double m1 = -s * sin(x1);
      double c0 = p0;
      double c1 = m0;
      double c2 = - 3 * p0 - 2 * m0 + 3 * p1 - m1;
```

```
      double c3 =   2 * p0 +     m0 - 2 * p1 + m1;
      cosf4_table[i][k][0] = c0;
      cosf4_table[i][k][1] = c1;
      cosf4_table[i][k][2] = c2;
      cosf4_table[i][k][3] = c3;
    }
  }
}

static inline float cosf4(float phase, int index, float size, int mask) {
  float p = fabsf(phase) * size;
  int q = p;
  p -= q;
  int i = q & mask;
  float x1 = p;
  float x2 = p * p;
  float x3 = p * p * p;
  float c0 = cosf4_table[index][i][0];
  float c1 = cosf4_table[index][i][1];
  float c2 = cosf4_table[index][i][2];
  float c3 = cosf4_table[index][i][3];
  return c0 + x1 * c1 + x2 * c2 + x3 * c3;
}

static inline void cosf4_reduce(float phase, float size, int mask, float * ↙
    ↳ restrict out_phase, int * restrict out_index) {
  float p = fabsf(phase) * size;
  int q = p;
  p -= q;
  int i = q & mask;
  *out_phase = p;
  *out_index = i;
}

static inline float cosf4_unsafe(float p, int i, int index) {
  const float * cs = &cosf4_table[index][i][0];
  float x1 = p;
  float x2 = p * p;
  float x3 = p * p * p;
  float c0 = cs[0];
  float c1 = cs[1];
  float c2 = cs[2];
  float c3 = cs[3];
  return c0 + x1 * c1 + x2 * c2 + x3 * c3;
}

extern void cosf4s1(int n, int index, const float * restrict in, float * ↙
    ↳ restrict out) {
  float size = 1 << index;
  int mask = (1 << index) - 1;
  while (n--) {
    *out++ = cosf4(*in++, index, size, mask);
  }
}

extern void cosf4s2(int n, int index, const float * restrict in, float * ↙
    ↳ restrict out, float * restrict tmp_phase, int * restrict tmp_index) {
```

```
  float size = 1 << index;
  int mask = (1 << index) - 1;
  float * restrict tmp2_phase = tmp_phase;
  int * restrict tmp2_index = tmp_index;
  int m = n;
  while (n--) {
    cosf4_reduce(*in++, size, mask, tmp_phase++, tmp_index++);
  }
  while (m--) {
    *out++ = cosf4_unsafe(*tmp2_phase++, *tmp2_index++, index);
  }
}
```

9 src/cosf4.h

```
#ifndef COSF4_H
#define COSF4_H 1

extern void cosf4_initialize(void);
extern void cosf4s1(int n, int index, const float * restrict in, float * restrict out);
extern void cosf4s2(int n, int index, const float * restrict in, float * restrict out, float * restrict tmp_phase, int * restrict tmp_index);

#endif
```

10 src/cosf7.c

```
#include "cosf7.h"

#include <math.h>

#define   likely(x) __builtin_expect((x),1)
#define unlikely(x) __builtin_expect((x),0)

// postcondition: 0 <= phase <= 1
static inline float cosf7_reduce(float phase) {
  float p = fabsf(phase);
  // p >= 0
  if (likely(p < (1 << 24))) {
    int q = p;
    return p - q;
  } else {
    if (unlikely(isnanf(p) || isinff(p))) {
      // return NaN
      return p - p;
    } else {
      // int could overflow, and it will be integral anyway
      return 0.0f;
    }
  }
}

// precondition: 0 <= phase <= 1
static inline float cosf7_unsafe(float phase) {
  float p = fabsf(4.0f * phase - 2.0f) - 1.0f;
```

```
  // p in -1 .. 1
  float s
    = 1.5707963267948965580e+00f * p
    - 6.4581411791873211126e-01f * p * p * p
    + 7.9239255452774770561e-02f * p * p * p * p * p
    - 4.2214643289391062808e-03f * p * p * p * p * p * p * p;
  // compiler figures out optimal simd multiplications
  return s;
}

static inline float cosf7(float phase) {
  return cosf7_unsafe(cosf7_reduce(phase));
}


extern void cosf7s1(int n, const float * restrict in, float * restrict out) {
  while (n--) {
    *out++ = cosf7(*in++);
  }
}

extern void cosf7s2(int n, const float * restrict in, float * restrict out, ↙
    ↳ float * restrict tmp) {
  float * restrict tmp2 = tmp;
  int m = n;
  while (n--) {
    *tmp++ = cosf7_reduce(*in++);
  }
  while (m--) {
    *out++ = cosf7_unsafe(*tmp2++);
  }
}
```

11 src/cosf7.h

```
#ifndef COSF7_H
#define COSF7_H 1

extern void cosf7s1(int n, const float * restrict in, float * restrict out);
extern void cosf7s2(int n, const float * restrict in, float * restrict out, ↙
    ↳ float * restrict tmp);

#endif
```

12 src/cosf7.m

```
output_precision(20);
inverse([        1,          1,          1
        ;        3,          5,          7
        ;        6,         20,         42 ]
 ) *    [ 1 - pi/2,     -pi/2,    -pi^2/4 ]'
```

13 src/cosf9.c

```
#include "cosf9.h"
```

```c
#include <math.h>

#define   likely(x) __builtin_expect((x),1)
#define unlikely(x) __builtin_expect((x),0)

// postcondition: 0 <= phase <= 1
static inline float cosf9_reduce(float phase) {
  float p = fabsf(phase);
  // p >= 0
  if (likely(p < (1 << 24))) {
    int q = p;
    return p - q;
  } else {
    if (unlikely(isnanf(p) || isinff(p))) {
      // return NaN
      return p - p;
    } else {
      // int could overflow, and it will be integral anyway
      return 0.0f;
    }
  }
}

// precondition: 0 <= phase <= 1
static inline float cosf9_unsafe(float phase) {
  float p = fabsf(4.0f * phase - 2.0f) - 1.0f;
  // p in -1 .. 1
  float s
    = 1.5707963267948965580e+00f * p
    - 6.4596271553942852250e-01f * p * p * p
    + 7.9685048314861006702e-02f * p * p * p * p * p
    - 4.6672571910271187789e-03f * p * p * p * p * p * p * p
    + 1.4859762069630022552e-04f * p * p * p * p * p * p * p * p * p;
  // compiler figures out optimal simd multiplications
  return s;
}

static inline float cosf9(float phase) {
  return cosf9_unsafe(cosf9_reduce(phase));
}

extern void cosf9s1(int n, const float * restrict in, float * restrict out) {
  while (n--) {
    *out++ = cosf9(*in++);
  }
}

extern void cosf9s2(int n, const float * restrict in, float * restrict out, ↙
    ↳ float * restrict tmp) {
  float * restrict tmp2 = tmp;
  int m = n;
  while (n--) {
    *tmp++ = cosf9_reduce(*in++);
  }
  while (m--) {
    *out++ = cosf9_unsafe(*tmp2++);
  }
```

```
}
```

14 src/cosf9.h

```
#ifndef COSF9_H
#define COSF9_H 1

extern void cosf9s1(int n, const float * restrict in, float * restrict out);
extern void cosf9s2(int n, const float * restrict in, float * restrict out, ↙
    ↳ float * restrict tmp);

#endif
```

15 src/cosf9.m

```
output_precision(20);
inverse([        1,          1,          1,          1
        ;        3,          5,          7,          9
        ;        6,         20,         42,         72
        ; (2/pi)^3,   (2/pi)^5,   (2/pi)^7,   (2/pi)^9 ]
 ) *    [ 1 - pi/2,      -pi/2,    -pi^2/4, sin(1) - 1 ]'
```

16 src/cosf9slow.h

```
#ifndef COSF9SLOW_H
#define COSF9SLOW_H 1

#include <math.h>

#define   likely(x) __builtin_expect((x),1)
#define unlikely(x) __builtin_expect((x),0)

static inline float cosf9slow(float phase) {
  float p = fabsf(phase);
  // p >= 0
  if (likely(p < (1 << 24))) {
    int q = p;
    p -= q;
    // p in 0 .. 1
    p = fabsf(4.0f * p - 2.0f) - 1.0f;
    // p in -1 .. 1
    float s
      = 1.5707963267948965580e+00f * p
      - 6.4596271553942852250e-01f * p * p * p
      + 7.9685048314861006702e-02f * p * p * p * p * p
      - 4.6672571910271187789e-03f * p * p * p * p * p * p * p
      + 1.4859762069630022552e-04f * p * p * p * p * p * p * p * p * p;
    // compiler figures out optimal simd multiplications
    return s;
  } else {
    if (unlikely(isnanf(p) || isinff(p))) {
      // return NaN
      return p - p;
    } else {
      // int could overflow, and it will be integral anyway
      return 1.0f;
```

```
    }
  }
}

#endif
```

17 src/cosf_quality.gnuplot

```
set term pngcairo mono enhanced font "LMSans10" size 1872,702
set xlabel "Algorithm"
set ylabel "Error (-log_2 E)"
set boxwidth 0.333
set style fill solid
set xtics in nomirror offset first 0.2,0 rotate by -30
set ytics 1 in
set grid
set grid noxtics
unset key
set title "Comparison of cosine approximation quality."
set output "cosf_quality.png"
plot [-0.666:16.666][12:24] \
  "./cosf_quality.txt" u (column(0)-0.2):(-log(column(2))/log(2)):xtic(1) w ↙
    ↳ boxes t "Maximum absolute error" lc rgb "#555555", \
  "./cosf_quality.txt" u (column(0)+0.2):(-log(column(3))/log(2))          w ↙
    ↳ boxes t "Root mean square error" lc rgb "#aaaaaa"
# replot as first time isn't aligned well
set title "Comparison of cosine approximation quality."
set output "cosf_quality.png"
plot [-0.666:16.666][12:24] \
  "./cosf_quality.txt" u (column(0)-0.2):(-log(column(2))/log(2)):xtic(1) w ↙
    ↳ boxes t "Maximum absolute error" lc rgb "#555555", \
  "./cosf_quality.txt" u (column(0)+0.2):(-log(column(3))/log(2))          w ↙
    ↳ boxes t "Root mean square error" lc rgb "#aaaaaa"
```

18 src/cosf_speed.gnuplot

```
set term pngcairo enhanced font "LMSans10" size 1872,702
set xlabel "Algorithm"
set ylabel "Time (seconds)"
set style line 1 lt 1 lc rgb "#555555"
set style line 2 lt 2 lc rgb "#aaaaaa"
set boxwidth 0.666
set style fill solid
set style histogram rowstacked
set style data histogram
set xtics in nomirror rotate by -30
set ytics in
set grid
set grid noxtics
unset key
set title "Comparison of cosine approximation speed (randomized small input)."
set output "cosf_speed_rnd_small.png"
plot [-0.666:36.666][0:] "./cosf_speed_rnd_small.txt" u 2:xtic(1) lc rgb ↙
  ↳ "#555555", '' u 3 lc rgb "#aaaaaa"
# replot as first time isn't aligned well
set title "Comparison of cosine approximation speed (randomized small input)."
```

```
set output "cosf_speed_rnd_small.png"
plot [-0.666:36.666][0:] "./cosf_speed_rnd_small.txt" u 2:xtic(1) lc rgb ↙
    ↳ "#555555", '' u 3 lc rgb "#aaaaaa"
set title "Comparison of cosine approximation speed (randomized large input)."
set output "cosf_speed_rnd_large.png"
plot [-0.666:36.666][0:] "./cosf_speed_rnd_large.txt" u 2:xtic(1) lc rgb ↙
    ↳ "#555555", '' u 3 lc rgb "#aaaaaa"
set title "Comparison of cosine approximation speed (sequential small input)."
set output "cosf_speed_seq_small.png"
plot [-0.666:36.666][0:] "./cosf_speed_seq_small.txt" u 2:xtic(1) lc rgb ↙
    ↳ "#555555", '' u 3 lc rgb "#aaaaaa"
set title "Comparison of cosine approximation speed (sequential large input)."
set output "cosf_speed_seq_large.png"
plot [-0.666:36.666][0:] "./cosf_speed_seq_large.txt" u 2:xtic(1) lc rgb ↙
    ↳ "#555555", '' u 3 lc rgb "#aaaaaa"
```

19 src/cosft.c

```
#include "cosft.h"

/* Copyright (c) 1997-1999 Miller Puckette.
* For information on usage and redistribution, and for a DISCLAIMER OF ALL
* WARRANTIES, see the file, "LICENSE.txt," in this distribution.  */

#include <math.h>
#include <stdint.h>

#define COSTABSIZE 512
#define UNITBIT32 1572864.  /* 3*2^19; bit 32 has place value 1 */
#define HIOFFSET 1
#define LOWOFFSET 0

union tabfudge
{
    double tf_d;
    int32_t tf_i[2];
};

/* ------------------------ cos~ ----------------------------- */

static float cos_table[COSTABSIZE + 1];

static inline float cospd1(float in)
{
    float *tab = cos_table, *addr, f1, f2, frac;
    double dphase;
    int normhipart;
    union tabfudge tf;

    tf.tf_d = UNITBIT32;
    normhipart = tf.tf_i[HIOFFSET];

        dphase = (double)(in * (float)(COSTABSIZE)) + UNITBIT32;
        tf.tf_d = dphase;
        addr = tab + (tf.tf_i[HIOFFSET] & (COSTABSIZE-1));
        tf.tf_i[HIOFFSET] = normhipart;
        frac = tf.tf_d - UNITBIT32;
```

```
        f1 = addr[0];
        f2 = addr[1];
        return f1 + frac * (f2 - f1);
}

static inline float cospd2(float in, int normhipart)
{
  float *tab = cos_table, *addr, f1, f2, frac;
  double dphase;
  union tabfudge tf;
  dphase = (double)(in * (float)(COSTABSIZE)) + UNITBIT32;
  tf.tf_d = dphase;
  addr = tab + (tf.tf_i[HIOFFSET] & (COSTABSIZE-1));
  tf.tf_i[HIOFFSET] = normhipart;
  frac = tf.tf_d - UNITBIT32;
  f1 = addr[0];
  f2 = addr[1];
  return f1 + frac * (f2 - f1);
}

extern void cosft_initialize(void)
{
    int i;
    float *fp, phase, phsinc = (2. * 3.14159) / COSTABSIZE;

    for (i = COSTABSIZE + 1, fp = cos_table, phase = 0; i--;
        fp++, phase += phsinc)
            *fp = cos(phase);
}

extern void cosfts1(int n, const float * restrict in, float * restrict out) {
  while (n--) {
    *out++ = cospd1(*in++);
  }
}

extern void cosfts2(int n, const float * restrict in, float * restrict out) {
  int normhipart;
  union tabfudge tf;
  tf.tf_d = UNITBIT32;
  normhipart = tf.tf_i[HIOFFSET];
  while (n--) {
    *out++ = cospd2(*in++, normhipart);
  }
}
```

20 src/cosft.h

```
#ifndef COSFT_H
#define COSFT_H 1

extern void cosft_initialize(void);
extern void cosfts1(int n, const float * restrict in, float * restrict out);
extern void cosfts2(int n, const float * restrict in, float * restrict out);

#endif
```

21 src/.gitignore

```
approximations
*.d
*.o
```

22 src/main.c

```
#include <math.h>
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

#include "noiseg.h"
#include "arithmetic.h"
#include "cosft.h"
#include "cosf7.h"
#include "cosf9.h"
#include "cosf4.h"
#include "cosf2.h"
#include "cosf1.h"

void compare_cosf(void) {
  const int blocksize = 64;
  float *b_phase = 0;
  float *b_phase2pi = 0;
  float *b_cosf = 0;
  float *b_cosfa = 0;
  float *b_tmpf = 0;
  int   *b_tmpi = 0;
#define ALLOC(p) posix_memalign((void **)(&(p)), 256, blocksize * sizeof(*(p)))
  ALLOC(b_phase);
  ALLOC(b_phase2pi);
  ALLOC(b_cosf);
  ALLOC(b_cosfa);
  ALLOC(b_tmpf);
  ALLOC(b_tmpi);
#undef ALLOC

  { // quality
#define ALGORITHMS 17
    const char *name[ALGORITHMS] =
      { "513 linear"
      , "1025 linear"
      , "2049 linear"
      , "4097 linear"
      , "8193 linear"
      , "512x2 linear"
      , "1024x2 linear"
      , "2048x2 linear"
      , "4096x2 linear"
      , "8192x2 linear"
      , "16x4 cubic"
      , "32x4 cubic"
      , "64x4 cubic"
```

```
    , "128x4 cubic"
    , "256x4 cubic"
    , "7 order poly"
    , "9 order poly"
    };
  float  abserr[ALGORITHMS];
  double rmserr[ALGORITHMS];
  for (int algorithm = 0; algorithm < ALGORITHMS; ++algorithm) {
    abserr[algorithm] = 0;
    rmserr[algorithm] = 0;
  }
  for (int i = 0; i < 1 << 24; ++i) {
    for (int j = 0; j < blocksize; ++j) {
      int k = i * blocksize + j;
      float p = k / ((double) (1 << 24) * blocksize);
      b_phase[j] = p;
    }
    mulffs(blocksize, 6.283185307179586f, b_phase, b_phase2pi);
    cosfs(blocksize, b_phase2pi, b_cosf);
    for (int algorithm = 0; algorithm < ALGORITHMS; ++algorithm) {
      switch (algorithm) {
        case 0:
        case 1:
        case 2:
        case 3:
        case 4:
          cosf1s2(blocksize, algorithm + 9, b_phase, b_cosfa, b_tmpf, b_tmpi);
          break;
        case 5:
        case 6:
        case 7:
        case 8:
        case 9:
          cosf2s2(blocksize, algorithm + 4, b_phase, b_cosfa, b_tmpf, b_tmpi);
          break;
        case 10:
        case 11:
        case 12:
        case 13:
        case 14:
          cosf4s2(blocksize, algorithm - 6, b_phase, b_cosfa, b_tmpf, b_tmpi);
          break;
        case 15:
          cosf7s2(blocksize, b_phase, b_cosfa, b_tmpf);
          break;
        case 16:
          cosf9s2(blocksize, b_phase, b_cosfa, b_tmpf);
          break;
      }
      abserr[algorithm] = maximumabssubfs(blocksize, abserr[algorithm], ↙
          ↳ b_cosfa, b_cosf);
      rmserr[algorithm] += sumsqrsubfs(blocksize, b_cosfa, b_cosf);
    }
  }
  for (int algorithm = 0; algorithm < ALGORITHMS; ++algorithm) {
    rmserr[algorithm] /= 1 << 24;
    rmserr[algorithm] /= blocksize;
```

```
105         rmserr[algorithm] = sqrt(rmserr[algorithm]);
        }
        FILE *f = fopen("cosf_quality.txt", "wb");
        fprintf(f, "# algorithm\tabserr\trmserr\n");
        for (int algorithm = 0; algorithm < ALGORITHMS; ++algorithm) {
110         fprintf(f, "\"%s\"\t%.16g\t%.16g\n", name[algorithm], abserr[algorithm], ↙
                ↳ rmserr[algorithm]);
        }
        fclose(f);
    #undef ALGORITHMS
      } // quality
115
      { // speed
    #define MODES 4
    #define ALGORITHMS 37
        const char *modes[MODES] =
120       { "cosf_speed_seq_small.txt"
          , "cosf_speed_rnd_small.txt"
          , "cosf_speed_seq_large.txt"
          , "cosf_speed_rnd_large.txt"
          };
125     const char *name[ALGORITHMS] =
          { "513 linear 1"
          , "1025 linear 1"
          , "2049 linear 1"
          , "4097 linear 1"
130       , "8193 linear 1"
          , "513 linear 2"
          , "1025 linear 2"
          , "2049 linear 2"
          , "4097 linear 2"
135       , "8193 linear 2"
          , "512x2 linear 1"
          , "1024x2 linear 1"
          , "2048x2 linear 1"
          , "4096x2 linear 1"
140       , "8192x2 linear 1"
          , "512x2 linear 2"
          , "1024x2 linear 2"
          , "2048x2 linear 2"
          , "4096x2 linear 2"
145       , "8192x2 linear 2"
          , "16x4 cubic 1"
          , "32x4 cubic 1"
          , "64x4 cubic 1"
          , "128x4 cubic 1"
150       , "256x4 cubic 1"
          , "16x4 cubic 2"
          , "32x4 cubic 2"
          , "64x4 cubic 2"
          , "128x4 cubic 2"
155       , "256x4 cubic 2"
          , "7 order poly 1"
          , "7 order poly 2"
          , "9 order poly 1"
          , "9 order poly 2"
160       , "pd costab 1"
```

```
      , "pd costab 2"
      , "libm cosf()"
      };
    for (int mode = 0; mode < 4; ++mode) {
      FILE *f = fopen(modes[mode], "wb");
      fprintf(f, "# algorithm\tseconds\toverheads\n\n");
      double tnull = 0;
      for (int algorithm = -1; algorithm < ALGORITHMS; ++algorithm) {
        struct timespec then, now;
        clock_gettime(CLOCK_THREAD_CPUTIME_ID, &then);
        uint32_t seed = 1;
        for (int i = 0; i < 1 << 24; ++i) {
          if (mode & 1) {
            seed = noisegifs(blocksize, seed, b_tmpf);
          } else {
            for (int j = 0; j < blocksize; ++j) {
              int k = ((i * blocksize + j) & ((1 << 16) - 1));
              b_tmpf[j] = k / (float) (1 << 16);
            }
          }
          if (mode & 2) {
            mulffs(blocksize, 5.0f, b_tmpf, b_phase);
          } else {
            mulffs(blocksize, 0.5f, b_tmpf, b_phase);
          }
          mulffs(blocksize, 6.283185307179586f, b_phase, b_phase2pi);
          switch (algorithm) {
            case -1:
              // overheads
              break;
            case 0:
            case 1:
            case 2:
            case 3:
            case 4:
              cosf1s1(blocksize, algorithm + 9, b_phase, b_cosfa);
              break;
            case 5:
            case 6:
            case 7:
            case 8:
            case 9:
              cosf1s2(blocksize, algorithm + 4, b_phase, b_cosfa, b_tmpf, b_tmpi↙
                  ↳ );
              break;
            case 10:
            case 11:
            case 12:
            case 13:
            case 14:
              cosf2s1(blocksize, algorithm - 1, b_phase, b_cosfa);
              break;
            case 15:
            case 16:
            case 17:
            case 18:
            case 19:
```

```
                cosf2s2(blocksize, algorithm - 6, b_phase, b_cosfa, b_tmpf, b_tmpi↙
                    ↳ );
                break;
              case 20:
220           case 21:
              case 22:
              case 23:
              case 24:
                cosf4s1(blocksize, algorithm - 16, b_phase, b_cosfa);
225             break;
              case 25:
              case 26:
              case 27:
              case 28:
230           case 29:
                cosf4s2(blocksize, algorithm - 21, b_phase, b_cosfa, b_tmpf, ↙
                    ↳ b_tmpi);
                break;
              case 30:
                cosf7s1(blocksize, b_phase, b_cosfa);
235             break;
              case 31:
                cosf7s2(blocksize, b_phase, b_cosfa, b_tmpf);
                break;
              case 32:
240             cosf9s1(blocksize, b_phase, b_cosfa);
                break;
              case 33:
                cosf9s2(blocksize, b_phase, b_cosfa, b_tmpf);
                break;
245           case 34:
                cosfts1(blocksize, b_phase2pi, b_cosfa);
                break;
              case 35:
                cosfts2(blocksize, b_phase2pi, b_cosfa);
250             break;
              case 36:
                cosfs(blocksize, b_phase2pi, b_cosfa);
                break;
            }
255       }
          clock_gettime(CLOCK_THREAD_CPUTIME_ID, &now);
          double tdelta = (now.tv_sec  - then.tv_sec )
                        + (now.tv_nsec - then.tv_nsec) / 1.0e9;
          if (algorithm < 0) {
260         tnull = tdelta;
          } else {
            fprintf(f, "\"%s\"\t%.16g\t%.16g\n", name[algorithm], tdelta - tnull, ↙
                ↳ tnull);
            fflush(f);
          }
265     }
        fclose(f);
      }
#undef MODES
#undef ALGORITHMS
270   } // speed
```

```
  { // waveforms
#define ALGORITHMS 18
    const char *name[ALGORITHMS] =
      { "cosf1-9.raw"
      , "cosf1-10.raw"
      , "cosf1-11.raw"
      , "cosf1-12.raw"
      , "cosf1-13.raw"
      , "cosf2-9.raw"
      , "cosf2-10.raw"
      , "cosf2-11.raw"
      , "cosf2-12.raw"
      , "cosf2-13.raw"
      , "cosf4-4.raw"
      , "cosf4-5.raw"
      , "cosf4-6.raw"
      , "cosf4-7.raw"
      , "cosf4-8.raw"
      , "cosf7.raw"
      , "cosf9.raw"
      , "cosft.raw"
      };
    FILE *fs[18];
    for (int algorithm = 0; algorithm < ALGORITHMS; ++algorithm) {
      fs[algorithm] = fopen(name[algorithm], "wb");
    }
    double phase = 0.75;
    for (int i = 0; i < (5 * 48000) / blocksize; ++i) {
      for (int j = 0; j < blocksize; ++j) {
        b_phase[j] = phase;
        int k = i * blocksize + j;
        double pitch = 20000 * pow(0.5, k / 24000.0);
        double increment = pitch / 48000.0;
        phase += increment;
        int q = phase;
        phase -= q;
      }
      mulffs(blocksize, 6.283185307179586f, b_phase, b_phase2pi);
      cosfs(blocksize, b_phase2pi, b_cosf);
      for (int algorithm = 0; algorithm < ALGORITHMS; ++algorithm) {
        switch (algorithm) {
          case 0:
          case 1:
          case 2:
          case 3:
          case 4:
            cosf1s2(blocksize, algorithm + 9, b_phase, b_cosfa, b_tmpf, b_tmpi);
            break;
          case 5:
          case 6:
          case 7:
          case 8:
          case 9:
            cosf2s2(blocksize, algorithm + 4, b_phase, b_cosfa, b_tmpf, b_tmpi);
            break;
          case 10:
```

```
            case 11:
            case 12:
330         case 13:
            case 14:
              cosf4s2(blocksize, algorithm - 6, b_phase, b_cosfa, b_tmpf, b_tmpi);
              break;
            case 15:
335           cosf7s2(blocksize, b_phase, b_cosfa, b_tmpf);
              break;
            case 16:
              cosf9s2(blocksize, b_phase, b_cosfa, b_tmpf);
              break;
340         case 17:
              cosfts2(blocksize, b_phase, b_cosfa);
              break;
          }
          subfs(blocksize, b_cosfa, b_cosf, b_tmpf);
345       fwrite(b_tmpf, sizeof(*b_tmpf) * blocksize, 1, fs[algorithm]);
        }
      }
      for (int algorithm = 0; algorithm < ALGORITHMS; ++algorithm) {
        fclose(fs[algorithm]);
350   }
#undef ALGORITHMS
  }

  free(b_phase);
355 free(b_phase2pi);
  free(b_cosf);
  free(b_cosfa);
  free(b_tmpf);
  free(b_tmpi);
360 }

int main() {
  cosf1_initialize();
  cosf2_initialize();
365 cosf4_initialize();
  cosft_initialize();
  compare_cosf();
  return 0;
}
```

23 src/Makefile

```
# for raspberry pi model 3 b try:
# make ARCH_FLAGS="-mfloat-abi=hard -mfpu=neon"

ARCH_FLAGS ?= -march=native
 5
COMPILE = gcc
COMPILE_FLAGS = \
        -std=c99 -Wall -Wextra -pedantic \
        -O3 -ffast-math -funroll-loops \
10      $(ARCH_FLAGS) \
        -D_DEFAULT_SOURCE -D_POSIX_C_SOURCE=200112L \
        -MMD -c
```

```
LINK = gcc
LINK_FLAGS = -lm

OBJECTS := $(patsubst %.c,%.o,$(wildcard *.c))
DEPENDS := $(patsubst %.o,%.d,$(OBJECTS))

all: approximations

clean:
	@echo "CLEAN" ; rm -f approximations $(OBJECTS) $(DEPENDS)

SUFFIXES:
.PHONY: all clean

approximations: $(OBJECTS)
	@echo "LINK    $@" ; $(LINK) -o $@ $(OBJECTS) $(LINK_FLAGS) || ( echo "↙
	    ↳ ERROR   $(LINK) -o $@ $(OBJECTS) $(LINK_FLAGS)" && false )

%.o: %.c
	@echo "C       $<" ; $(COMPILE) $(COMPILE_FLAGS) -o $@ $< || ( echo "↙
	    ↳ ERROR   $(COMPILE) $(COMPILE_FLAGS) -o $@ $<" && false )

-include $(DEPENDS)
```

24 src/noiseg.c

```
#include "noiseg.h"

// seed 1 has period 2^30
static inline uint32_t noiseg_u32(uint32_t seed) {
  return 1640531525u * seed;
}

static inline float i32_to_f32(int32_t i) {
  return i / 2147483648.f;
}

static inline float u32_to_f32(uint32_t u) {
  return u / 4294967296.f;
}

extern uint32_t noisegufs(int n, uint32_t seed, float * restrict out) {
  while (n--) {
    *out++ = u32_to_f32(seed = noiseg_u32(seed));
  }
  return seed;
}

extern uint32_t noisegifs(int n, uint32_t seed, float * restrict out) {
  while (n--) {
    *out++ = i32_to_f32(seed = noiseg_u32(seed));
  }
  return seed;
}
```

25 src/noiseg.h

```
#ifndef NOISEG_H
#define NOISEG_H 1

#include <stdint.h>

extern uint32_t noisegufs(int n, uint32_t seed, float * restrict out);
extern uint32_t noisegifs(int n, uint32_t seed, float * restrict out);

#endif
```