A Fast Sine/Cosine Routine

W. D. Cotton, July 17, 2009

***Abstract*—The calculation of sine/cosine pairs is a common and relatively expensive operation in radio interferometry. The very flexible "DFT" calculation of the interferometer response to a sky model makes heavy use of this operation and the cost of sine and cosine calculations can dominate the run time of a process using this technique. This memo describes a technique for fast sine/cosine calculations to moderate accuracy. The technique is a table lookup followed by a single term Taylor's series expansion. An Streaming SIMD Extensions (SSE) implementation further increased the efficiency. A factor of 8 increase in the sine/cosine calculation speed and a factor of 3 increase in the speed of an actual application are reported. The precision obtained is more than adequate for the intended applications.**

***Index Terms*—interferometry, performance**

I. Introduction

CAlculating the response of an interferometer to a sky model is a common operation in radio interferometry and for complex sky models can be one of the more computationally expensive operations. One generic type of sky model calculation is the so called Direct Fourier Transform (AKA "DFT") technique wherein the response to each component of a sky model (e.g. CLEAN component) is evaluated for each complex correlation in the data set. The real and imaginary parts of such model calculations are evaluated by sine and cosine functions of the component phase. This can result in VERY LARGE numbers of calls to sin and cos routines which dominate the cost of this operation.

The sine and cosine routines in standard mathematical libraries have much higher precision that is needed for this radio interferometry application in that they exceed by many orders of magnitude the accuracy of practical phase calibration. Standard library versions of these functions are subject to additional constraints such as smoothness and range checking which are unnecessary in the applications discussed here. Furthermore, much of the cost of the operation for the separate sine and cosine calculations is in common. A moderate relaxation of the precision and other constraints in addition to and coupling the calculations of sines and cosines has the potential for significant performance enhancements.

This memo explores such a technique in the Obit package [1] [1].

II. Sines and Cosines

Sines and cosines are periodic functions that repeat every 2π radians of their argument. Thus, while the range of arguments to these functions is $\pm\infty$, they can be fully described by the interval $[0, 2\pi]$ and replacing the argument with its value modulo 2π. This facilitates a table lookup scheme.

National Radio Astronomy Observatory, 520 Edgemont Rd., Charlottesville, VA, 22903 USA email: bcotton@nrao.edu

[1]http://www.cv.nrao.edu/∼bcotton/Obit.html

While an arbitrary precision can be obtained by a simple table lookup, adequate resolution can require quite large tables. Further improvements in the precision of a table lookup can be obtained using either an interpolation in the table or a series expansion about the tabulated value. A Taylor's series expansion is given by:

$$f(x) \;=\; f(a) \;+\; f'(a)\frac{x-a}{1!} \;+\; f''(a)\frac{(x-a)^2}{2!} + ...$$

Since the derivatives of sines and cosines are the cosines and sines of the same angles, evaluation of such a series is straightforward. The one term expansions for sine and cosines of angle x about tabulated value a are:

$$sin(x) \;=\; sin(a) \;+\; cos(a)[x-a]$$

$$cos(x) \;=\; cos(a) \;-\; sin(a)[x-a]$$

III. Implementation

A fast sine/cosine routine, ObitSinCosCalc, to calculate a sine/scosine pair was implemented in Obit utility module ObitSinCos using a 1381 element (1 1/4 turn) table lookup followed by the single term expansion given above. This routine will initialize the tables on the first call. Once initialized, this function should be threadsafe; a single call prior to initializing threading using this routine will make usage threadsafe.

A. Scalar

A single lookup table can be used for sine and cosine by noting that cos(phase) = sin(phase + 1/4 turn). Further reductions in the size of the table needed can be had by using the symmetries in the sine/cosine functions but at a cost of increased logic and computation.

B. Vector

It is possible to go one step further. The overhead of function calls can be reduced by collecting the phases for which the sines and cosines are desired into an array and doing the calculations in a single function call. A "vector" version of ObitSinCosCalc is implemented in routine ObitSinCosVec. Organizing data into vectors also improves the cache hit ratio of the code.

C. SSE

Streaming SIMD Extensions (SSE) is an implementation of SIMD (Single Instruction Multiple Data) architecture for length 4 vectors on Pentium and similar design CPU chipsets. Furthermore, it is supported by the gcc compiler (and likely others) meaning available on all (or nearly all) architectures of interest for radio interferometry. This feature allows coding

TABLE I
SIN/COS TIMINGS

method	total CPU time sec.	vector length	CPU used sec.
c sinf/cosf	9.310		7.549
c sincosf	6.169		4.408
ObitSinCosCalc	4.727		2.966
ObitSinCosVec	4.064	10	2.303
ObitSinCosVec	3.780	100	2.019
ObitSinCosVec	3.756	1000	1.995
ObitSinCos SSE	2.696	1000	0.935
SSE + aligned	2.526	1000	0.764
none	1.761		

TABLE II
UVSUB TIMINGS

method	CPU time sec.	Real time sec.	CPU/Real
c sinf/cosf	40.6	24.5	1.65
c sincosf	27.1	16.9	1.60
ObitSinCosCalc	19.1	12.5	1.52
ObitSinCosVec(100)	14.9	9.9	1.51
ObitSinCosVec(SSE)	11.0	7.7	1.43

very fast routines although at the level of assembler and with a Spartan instruction set. This system was designed for fast video game software but has the basics needed for interferometric calculation. A version of ObitSinCosCalc for length 4 vectors was coded using SSE (the most basic version) and implemented in ObitSinCosVec. This code is wrapped in #ifdefs such that if SSE is not available, the non-SSE version will be used. The text of the ObitSinCos utility package is given in the appendix.

IV. TESTING

The following give the results of various precision and timing tests.

A. Precision

In order to evaluate the precision of this technique compared to the standard library sinf/cosf routines, a test program was employed that used 10^8 angles randomly spaced from -100 to 100 radians and compared the results of ObitSinCosCalc with the c library sinf and cosf routines. The average difference in this test was 8.9×10^{-9}, the rms difference was 1.8×10^{-6} and the maximum difference was 6.0×10^{-6}. This rms difference corresponds to an equivalent phase error rms of $\sim 1.0\times10^{-4\circ}$.

B. Sin/cos Timing

There are several standard c library routines that can be used to compute sines and cosines. First, there are the sinf and cosf individual routines; then there is the combined routine sincosf for computing both sines and cosines. The test calculations described in the previous section were repeated doing only the c library (sinf/cosf or sincosf) or ObitSinCosCalc calculation and the run times measured with the Unix time utility. Further tests were performed using the ObitSinCosVec vector function with a variety of vector lengths. The results are given in Table I. The final entry, “none”, measures the overhead of calculating the angles; this value was subtracted from the total run times and are given in the “CPU used” column. The SSE implementation is given in the entry “ObitSinCos SSE”. In this implementation, a nontrivial portion of the time was used in copying data between regular c arrays and the 16-byte aligned arrays needed for SSE. When the regular arrays were replaced with 16-byte aligned arrays and the copy replaced by changing pointers, the result is given by entry “SSE + aligned”. This feature has serious implications for the calling routine and was not implemented in ObitSinCos. However, if another 20% performance is important this is a viable technique.

The ObitSinCosCalc version was 3.0 times faster than the c library sinf/cosf version and 1.7 times faster than the combined sincosf version. The ObitSinCosVec version ran as much as 3.8 times faster than the sinf/cosf version. The SSE version implemented was 8.1 times faster than the sinf/cosf version. The SSE version with aligned buffers (not implemented) was 9.9 times faster. These tests were run on a 3 GHz Xenon intel machine; the ObitSinCosVec SSE routine took about 23 CPU cycles for each sine/cosine pair.

C. UVSub Test

To test the accuracy and speed of the sine/cosine routine on a real dataset, Obit task UVSub was used on a VLA data set containing 78,000 visibilities each with 15 channels and a model composed of 22 facets with a total of 386 CLEAN components after summing all components in the same pixel. The field contains a 12 Jy point source; previous calibration and editing were applied to the input data. UVSub was run using both the c library sinf/cosf and sincosf routines as well as ObitSinCosCalc and ObitSinCosVec (length 100) versions. These all used the same data sets, 2 threads and the “DFT” model algorithm. The timing results are given in Table II. The use of the vector (non-SSE) routine reduced the CPU time by a factor of 3.2 and the run time by a factor of 2.5. The SSE version reduced the CPU time by a factor of 3.7 and the run time by a factor of 3.2.

The output data files were compared using Obit utility UV.PUtilVisCompare which gave a relative rms difference between all real and imaginary parts of the visibilities of 1.448×10^{-4}. Note: this is the fraction of the residuals **after** subtracting the model.

V. DISCUSSION

The cost of “DFT” interferometer model calculations is strongly dominated by the cost of calculating sines and cosines. The new sine/cosine SSE based routine described above appears to be 8 times faster than the c library sinf and cosf versions and, when implemented in a real application, reduced the run time by a factor of 3.2. The SSE version of the sin/cosine function appear to to be sufficiently fast as to no longer dominate the run time in the UVSub test. The precision of these routines appears to be more than adequate

and far better than the precision of the calibration. Use of this fast sine/cosine technique can significantly reduce the cost of a common, and expensive, operation in radio interferometry.

ACKNOWLEDGMENT

The author thanks Scott Ransom for discussions and pointing out the existance of SSE.

APPENDIX

Text of the Obit utility ObitSinCos.c follows.

REFERENCES

[1] W. D. Cotton, "Obit: A Development Environment for Astronomical Algorithms," *PASP*, vol. 120, pp. 439–448, 2008.

```
/* Utility routine for fast sine/cosine calculation   */
#include "ObitSinCos.h"
#define OBITSINCOSNTAB 1024  /* size of tables -1 */
#define OBITSINCOSNTAB4 256   /* 1/4 size of tables */
/** Is initialized? */
isInit = FALSE;
/** Sine lookup table covering 1 1/4 turn of phase */
 ofloat sincostab[OBITSINCOSNTAB+OBITSINCOSNTAB4+1];
/** Angle spacing (radian) in table */
ofloat delta;
/** 1/2pi */
ofloat itwopi = 1.0/ (2.0 * G_PI);
/** 2pi */
ofloat twopi = (2.0 * G_PI);
/**
 * Initialization
 */
 void ObitSinCosInit(void)
{
  olong i;
  ofloat angle;
  isInit = TRUE;  /* Now initialized */
  delta = ((2.0 *G_PI)/OBITSINCOSNTAB);
  for (i=0; i<(OBITSINCOSNTAB+OBITSINCOSNTAB4+1); i++) {
    angle = delta * i;
    sincostab[i] = sinf(angle);
  }
} /* end ObitSinCosInit */
```

```
/**
 * Calculate sine/cosine of angle
 * Lookup table initialized on first call
 * \param angle  angle in radians
 * \param sin    [out] sine(angle)
 * \param cos    [out] cosine(angle)
*/
void ObitSinCosCalc(ofloat angle, ofloat *sin, ofloat *cos)
{
  olong it, itt;
  ofloat anglet, ss, cc, d;

  /* Initialize? */
  if (!isInit) ObitSinCosInit();

  /* angle in turns */
  anglet = angle*itwopi;

  /* truncate to [0,1] turns */
  it = (olong)anglet;
  if (anglet<0.0) it--;   /* fold to positive */
  anglet -= it;

  /* Lookup, cos(phase) = sin(phase + 1/4 turn) */
  itt = (olong)(0.5 + anglet*OBITSINCOSNTAB);
  ss  = sincostab[itt];
  cc  = sincostab[itt+OBITSINCOSNTAB4];

  /* One term Taylors series  */
  d = anglet*twopi - delta*itt;
  *sin = ss + cc * d;
  *cos = cc - ss * d;
} /* end ObitSinCosCalc */
```

```
/**
 * Calculate sine/cosine of vector of angles uses SSE implementation is available
 * Lookup table initialized on first call
 * \param n      Number of elements to process
 * \param angle  array of angles in radians
 * \param sin    [out] sine(angle)
 * \param cos    [out] cosine(angle)
*/
void ObitSinCosVec(olong n, ofloat *angle, ofloat *sin, ofloat *cos)
{
  olong i, nleft, it, itt;
  ofloat anglet, ss, cc, d;
  /** SSE implementation */
#ifdef HAVE_SSE
  olong ndo;
  V4SF vanglet, vss, vcc;
#endif /* HAVE_SSE */

  /* Initialize? */
  if (!isInit) ObitSinCosInit();

  nleft = n;   /* Number left to do */
  i     = 0;   /* None done yet */

 /** SSE implementation */
#ifdef HAVE_SSE
  /* Loop in groups of 4 */
  ndo = nleft - nleft%4;  /* Only full groups of 4 */
  for (i=0; i<ndo; i+=4) {
    vanglet.f[0] = *angle++;
    vanglet.f[1] = *angle++;
    vanglet.f[2] = *angle++;
    vanglet.f[3] = *angle++;
    fast_sincos_ps(vanglet.v, sincostab, &vss.v, &vcc.v);
    *sin++ = vss.f[0];
    *sin++ = vss.f[1];
    *sin++ = vss.f[2];
    *sin++ = vss.f[3];
    *cos++ = vcc.f[0];
    *cos++ = vcc.f[1];
    *cos++ = vcc.f[2];
    *cos++ = vcc.f[3];
  } /* end SSE loop */
#endif /* HAVE_SSE */

  nleft = n-i;  /* How many left? */

 /* Loop doing any elements not done in SSE loop */
  for (i=0; i<nleft; i++) {
    /* angle in turns */
    anglet = (*angle++)*itwopi;

    /* truncate to [0,1] turns */
    it = (olong)anglet;
    if (anglet<0.0) it--;   /* fold to positive */
    anglet -= it;
```

```
   /* Lookup, cos(phase) = sin(phase + 1/4 turn) */
    itt = (olong)(0.5 + anglet*OBITSINCOSNTAB);
    ss  = sincostab[itt];
    cc  = sincostab[itt+OBITSINCOSNTAB4];

    /* One term Taylor series  */
    d = anglet*twopi - delta*itt;
    *sin++ = ss + cc * d;
    *cos++ = cc - ss * d;
  } /* end loop over vector */
} /* end ObitSinCosVec */
```

```
/** SSE implementation */
#ifdef HAVE_SSE
#include <xmmintrin.h>

typedef __m128 v4sf;
typedef __m64 v2si;

/* gcc or icc */
# define ALIGN16_BEG
# define ALIGN16_END __attribute__((aligned(16)))

/* Union allowing c interface */
typedef ALIGN16_BEG union {
  float f[4];
  int   i[4];
  v4sf  v;
} ALIGN16_END V4SF;

/* Union allowing c interface */
typedef ALIGN16_BEG union {
  int   i[2];
  v2si  v;
} ALIGN16_END V2SI;

#define _OBIT_TWOPI  6.2831853071795862     /* 2pi */
#define _OBIT_ITWOPI 0.15915494309189535    /* 1/2pi */
#define _OBIT_DELTA  0.0061359231515425647  /* table spacing = 2pi/Obit_NTAB */
#define _OBIT_NTAB   1024.0                 /* size of table -1 */
#define _OBIT_NTAB4  256
```

```
/**
 * Fast sine/cosine of angle
 * Approximate sine/cosine, no range or value checking
 * \param angle  angle in radians
 * \param table  lookup table
 * \param s      [out] sine(angle)
 * \param c      [out] cosine(angle)
*/
void fast_sincos_ps(v4sf angle, float *table, v4sf *s, v4sf *c) {
  v4sf anglet, temp, it, zero, mask, one, sine, cosine, d;
  v2si itLo, itHi;
  V2SI iaddrLo, iaddrHi;

  /* angle in turns */
  temp   = _mm_set_ps1 (_OBIT_ITWOPI);
  anglet = _mm_mul_ps(angle, temp);

  /* truncate to [0,1] turns */
  /* Get full turns */
  itLo   = _mm_cvttps_pi32 (anglet);       /* first two truncated */
  temp   = _mm_movehl_ps (anglet,anglet);  /* upper two values into lower */
  itHi   = _mm_cvttps_pi32 (temp);         /* second two truncated */
  it     = _mm_cvtpi32_ps (it, itHi);      /* float upper values */
  temp   = _mm_movelh_ps (it, it);         /* swap */
  it     = _mm_cvtpi32_ps (temp, itLo);    /* float lower values */

  /* If anglet negative, decrement it */
  zero   = _mm_setzero_ps ();              /* Zeros */
  mask   = _mm_cmplt_ps (anglet,zero);     /* Comparison to mask */
  one    = _mm_set_ps1 (1.0);              /* ones */
  one    = _mm_and_ps(one, mask);          /* mask out positive values */
  it     = _mm_sub_ps (it, one);
  anglet = _mm_sub_ps (anglet, it);        /* fold to [0,2pi] */

  /* Table lookup, cos(phase) = sin(phase + 1/4 turn)*/
  temp      = _mm_set_ps1 (_OBIT_NTAB);
  it        = _mm_mul_ps(anglet, temp);          /* To cells in table */
  temp      = _mm_set_ps1 (0.5);
  it        = _mm_add_ps(it, temp);                  /* To cells in table */
  iaddrLo.v = _mm_cvttps_pi32 (it);
  temp      = _mm_movehl_ps (it,it);                 /* Round */
  iaddrHi.v = _mm_cvttps_pi32 (temp);
  sine      = _mm_setr_ps(table[iaddrLo.i[0]],table[iaddrLo.i[1]],
  table[iaddrHi.i[0]],table[iaddrHi.i[1]]);
  cosine    = _mm_setr_ps(table[iaddrLo.i[0]+_OBIT_NTAB4],
  table[iaddrLo.i[1]+_OBIT_NTAB4],
  table[iaddrHi.i[0]+_OBIT_NTAB4],
  table[iaddrHi.i[1]+_OBIT_NTAB4]);
```

```
  /* One term Taylor series  */
  temp   = _mm_set_ps1 (_OBIT_TWOPI);
  anglet = _mm_mul_ps(anglet, temp);            /* Now angle on radians */
  temp   = _mm_cvtpi32_ps (it, iaddrHi.v);      /* float upper values */
  it     = _mm_movelh_ps (temp,temp);           /* swap */
  it     = _mm_cvtpi32_ps (it, iaddrLo.v);      /* float lower values */
  temp   = _mm_set_ps1 (_OBIT_DELTA);
  d      = _mm_mul_ps (it, temp);               /* tabulated phase */
  d      = _mm_sub_ps (anglet, d);              /* actual-tabulated phase */
  /* Sine */
  temp   = _mm_mul_ps (cosine,d);
  *s     = _mm_add_ps (sine, temp);
  /* Cosine */
  temp   = _mm_mul_ps (sine,d);
  *c     = _mm_sub_ps (cosine, temp);

  _mm_empty();  /* wait for operations to finish */
  return ;
} /* end fast_sincos_ps */

#endif  /* HAVE_SSE */
```