Approximate Math Library

for Intel® Streaming SIMD Extensions

Release 2.0, October 2000
Documentation File

Content

Approximate Math Library for Intel Streaming SIMD Extensions

Approximate Math Library (AM Library) is a set of fast routines to calculate math functions using Intel® Streaming SIMD Extensions (SSE) and Streaming SIMD Extensions 2 (SSE2). The Library offers trigonometric, reverse trigonometric, logarithmic, and exponential functions for packed and scalar arguments. The processing speed is many times faster than that of x87 instructions and table lookups. The accuracy of AM Library routines can be adequate for many applications. It is comparable with that of reciprocal SSE instructions, and is hundreds times better than what is achievable with lookup tables.

AM Library can be used with C, C++, or Assembler programs targeting Intel processors with SSE and SSE2, such as Intel® Pentium® III, Pentium III Xeon™, and Pentium 4 processors, as well as some Intel® Celeron™ processors.

Problem: Calculating Math Functions in SIMD Code

Except for the square root instruction, SSE and SSE2 do not include instructions to calculate math functions like sine or arctangent. To calculate such functions in SIMD code, one could:

- use x87 instructions or
- use table lookups

x87 Instructions

x87 instructions for math functions are generally slow (many dozens or even hundreds processor cycles) and non-pipelined. To slow things down yet more, the data must be passed over from SIMD xmm registers to x87 registers and back, a process involving two write-read memory accesses. The generated code will look similar to:

```
movaps         [eax], xmm0  // store out four packed arguments
emms           // needed if MMX(TM) code has been used
fld            dword ptr [eax]  // load first argument
fsin           // calculate sine of the first argument
fstp           dword ptr [eax]  // store out the result
fld            dword ptr [eax + 4]  // load second argument
fsin           // calculate sine of the second argument
fstp           dword ptr [eax + 4]  // store out the result
//... repeat for [eax + 8], [eax + 12]
movaps         xmm0, [eax]  // load four results -- memory stall!
```

The results are written out of x87 stack in four dwords, but are read into a SIMD register in one 16-byte xmmword. This causes a memory stall.

If MMX™ code is used in the routine or calling code, emms instruction must be inserted before any x87 operations. The content of MMX registers might need to be saved and later restored. This further adds to the processing time.

Table Lookups

Table lookups are generally significantly faster than x87 instructions, but their accuracy is low even with large tables and degrades quickly when table size is reduced. Using large tables increases the cache miss rate and cache trashing, slowing down not only the function itself, but also the calling code.

Like x87 instructions, table lookups are scalar operations. They deliver their results element by element. As it is the case with x87 instructions, this requires two write-read memory accesses and causes a memory stall once the results are loaded into an xmm register.

Finally, lookup tables are not practical for many unlimited and non-periodical functions (for example, exponential functions) and functions taking several arguments (power function).

Introducing AM Library

AM Library uses polynomial and rational approximations to calculate math functions efficiently using SSE and, if present, SSE2. (Some AM Library functions also use MMX™ instructions.) AM Library shares the philosophy of reciprocal SSE instructions "Trade little accuracy for much speed". AM Library functions are several times faster than x87 instructions and table lookups. AM Library functions are also significantly more precise than table lookups.

AM Library currently implements the following functions:

Description	SSE version, packed	SSE version, scalar	SSE2 version, packed	SSE2 version, scalar
Sine	`am_sin_ps`	`am_sin_ss`	`am_sin_eps`	`am_sin_ess`
Cosine	`am_cos_ps`	`am_cos_ss`	`am_cos_eps`	`am_cos_ess`
Sine and cosine	`am_sincos_ps`	`am_sincos_ss`	`am_sincos_eps`	`am_sincos_ess`
Tangent	`am_tan_ps`	`am_tan_ss`	`am_tan_eps`	`am_tan_ess`
Arch sine	`am_asin_ps`	`am_asin_ss`	`am_asin_eps`	`am_asin_ess`
Arch cosine	`am_acos_ps`	`am_acos_ss`	`am_acos_eps`	`am_acos_ess`
Arch tangent	`am_atan_ps`	`am_atan_ss`	`am_atan_eps`	`am_atan_ess`
Arch tangent 2, reciprocal*	`am_atanr2_ps`	`am_atanr2_ss`	`am_atanr2_eps`	`am_atanr2_ess`
Exponent	`am_exp_ps`	`am_exp_ss`	`am_exp_eps`	`am_exp_ess`
Binary exponent	`am_exp2_ps`	`am_exp2_ss`	`am_exp2_eps`	`am_exp2_ess`
Logarithm	`am_log_ps`	`am_log_ss`	`am_log_eps`	`am_log_ess`
Binary logarithm	`am_log2_ps`	`am_log2_ss`	`am_log2_eps`	`am_log2_ess`
Power	`am_pow_ps`	`am_pow_ss`	`am_pow_eps`	`am_pow_ess`

* atanr2 is equivalent to atan2 with the reciprocal of the second argument, i.e. atanr2(x, y) == atan2(x, 1/y).

Versions of AM Library Functions

Every AM Library routine comes in packed and scalar version. For every version, there are two variants. One is optimized for Pentium® III processor and SSE, the other is optimized for Pentium® 4 processor with SSE2. The versions and variants follow the naming convention:

Optimized for Pentium® III processor and SSE	Packed	`am_xxx_ps`
	Scalar	`am_xxx_ss`
Optimized for Pentium® 4 processor and SSE2	Packed	`am_xxx_eps`
	Scalar	`am_xxx_ess`

Packed vs. Scalar AM Library Functions

Compared to the packed AM Library functions, the scalar ones are slower per operand but faster per call. Therefore, when just one result is needed, it is faster to call the scalar version. Otherwise (for two or more results), the packed version should be used.

AM Library Functions Optimized for Pentium® 4 Processor

In the code optimized for Pentium® 4 processor, the AM Library functions optimized for this processor should be used, since in most cases they provide a significant additional speedup. These functions should only be used when executing on Intel processors supporting SSE2. On other processors, their behavior is undefined. Please refer to Intel documentation on how an application can detect support for SSE2 at runtime.

For detailed information on performance of AM Library functions, refer to Appendix A in the end of this document.

Accuracy of AM Library Functions

The average relative error of all AM Library functions is below 0.03%, and for many AM Library functions it is significantly lower. For comparison, the average relative error for a 256-entry cosine lookup table is over 1.5%, and for rcpps instruction it is 0.01%.

For detailed information on accuracy of AM Library functions, refer to Appendices B and C in the end of this document.

Valid Argument Ranges

AM Library functions have been optimized for speed and so do not perform extensive checking for illegal arguments or arguments outside the valid argument range. For AM Library functions, the valid argument ranges as well as the return values for illegal arguments may differ from that of the equivalent x87 functions. If the packed argument for a packed AM Library function contains some illegal and some legal elements, the values for legal ones will be calculated correctly.

AM Library guarantees that, if SSE/SSE2 exceptions are disabled (default setting), there will be no exceptions raised regardless of the argument values, i.e. all calls to AM Library functions will return normally. If SSE/SSE2 exceptions are enabled, AM Library does not guarantee to raise an exception in case of illegal arguments.

For detailed information on valid argument ranges for AM Library functions, refer to Appendix D in the end of this document. Appendix D also specifies the behavior of AM Library functions for illegal arguments and arguments outside of the valid range.

Using AM Library

AM Library can be used with C, C++, or Assembler programs targeting Intel processors with SSE and SSE2. To use AM Library, a C or C++ program should #include the header file `AMaths.h` and link with the library file `AMaths.lib`. All AM Library functions expect arguments of type `__m128` and return one `__m128` value. Therefore, AM Library functions can easily be called from C/C++ code that uses Intel SIMD intrinsics or Intel SIMD Vector Classes, and from assembly code.

AM Library Functions Requiring `emms` *Instruction*

The following AM Library functions

```
am_sin_ps
am_cos_ps
am_tan_ps
am_log_ps
am_log2_ps
am_exp_ps
am_exp2_ps
am_pow_ps
```

use MMX™ instructions and do *not* reset the x87 state. If x87 code should be executed after a call to any of these functions, an `emms` instruction (or `_m_empty()` intrinsic) should be inserted *after* the call to any of the listed AM Library functions and *before* the first x87 instruction or C/C++ code using x87 instructions. C/C++ code that operates on data of type float or double is likely to use x87 instructions. Code using Intel SIMD intrinsics does not generate x87 instructions.

Avoid using `emms` instruction (or `_m_empty()` intrinsic) in performance-critical code, such as loops that are executed many times. A typical usage scenario might look like

```
for (i = 0; i < BIG_NUMBER; i++)
{
   // ... code using am_xxx_ps functions
}
_m_empty();  // call once outside critical code
// ... code using x87 instructions
```

Calling AM Library Functions from Assembly Code

AM Library functions follow the `__stdcall` calling convention. They expect the first `__m128` argument in `xmm0` register and the second `__m128` argument (if present) in `xmm1` register. The result is returned in `xmm0` register.

When calling AM Library functions from out-of-line assembly code, the `__stdcall` decoration of function names should be taken into account. This decoration inserts an underscore before the name and adds a ‘@<size of all arguments in bytes>’ sequence after the name. For example, `am_sin_ps` becomes `_am_sin_ps@16` and `am_atanr2_ps` becomes `_am_atanr2_ps@32`.

The in-line assembly code should use the normal (non-decorated) names since the compiler will handle the decoration.

Scalar AM Library Functions

The scalar AM Library functions do not retain the most significant three elements of their `__m128` argument.

Appendix A. Processing Speed Measurements

The following table lists the processing speed measurement results in clocks per one element of result for AM Library packed and scalar functions as well as x87 instructions and (for selected functions) table lookups. Since packed AM Library functions (`am_xxx_ps` and `am_xxx_eps`) calculate four result elements in one call, their total call time is equal table value times four. The results include the overhead for argument loading and result storing and are rounded to the nearest 1/2 clock. All measurements were performed on evenly distributed random data from the legal argument range.

Results with a test system with Pentium® III 866 MHz processor and Intel 820 chipset:

Function	x87	Table Lookup	am_xxx_ps	am_xxx_ss	Speedup over x87, times
sin	92	76	16.5	48	5.6
cos	92	75	16.5	49	5.6
sincos*	134	149	25	88	5.4
tan	131.5		25.5	86.5	5.2
atan	151.5		18	44	8.4
atan2	145		22.5	69.5	6.4
asin	207.5		21	48.5	9.9
acos	215.5		21	58.5	10.3
exp	150		21	65.5	7.1
exp2*	151		17.5	55.5	8.6
log	86.5		15.5	49	5.6
log2*	104		16	49	6.5
pow	246.5		35	121	7.0

* exp2 and log2 functions are not included with the standard C math library and were implemented for the purpose of this test using the appropriate x87 instructions. The test implementations are included with the AM Library package.

Results with a test system with Pentium 4 1.5 GHz processor and Intel 850 chipset:

Function	x87	Table Lookup	am_xxx_ps	am_xxx_ss	am_xxx_eps	am_xxx_ess	Speedup over x87
sin	189	89	38	67	12.5	49.5	15.1
cos	178.5	82.5	38	67	13.5	55	13.2
sincos*	214.5	155	47	101	20.5	80.5	10.5
tan	223		38	116.5	31.5	114	7.1
atan	313.5		18.5	58	18.5	57.5	16.9
atan2	249		25.5	118	25.5	93	9.8
asin	230.5		26.5	75	26.5	75	8.7
acos	321.5		25	107	25	107	12.9
exp	199.5		34.5	104.5	22.5	89.5	8.9
exp2*	205.5		28	75.5	19	71.5	10.8
log	98		32	79	17.5	68.5	5.6
log2*	103		32.5	75	17.5	70.5	5.7
pow	460		79.5	179	40.5	160.5	11.4

* exp2 and log2 functions are not included with the standard C math library and were implemented for the purpose of this test using the appropriate x87 instructions. The test implementations are included with the AM Library package.

Appendix B. Average Relative Errors

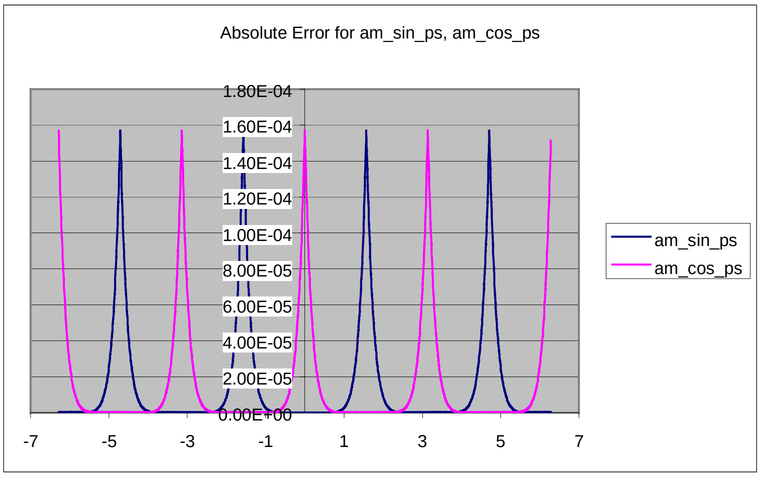
The following table lists the average relative errors in percents for packed and scalar AM Library functions as well as for some table lookups. The results have been averaged over 40,000 evenly distributed measurements from the test argument range and rounded up.

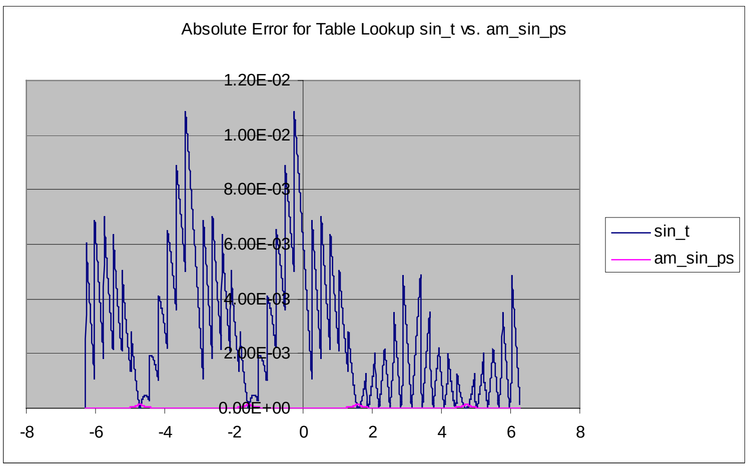
Functions	Test Argument Range(s)	Average Relative Error, %	Average Relative Error of Table Lookup, %
am_sin_xs*	-1e3…+1e3	0.0085	1.9
am_cos_xs**	-1e3…+1e3	0.018	1.7
am_tan_xs	-1e3…+1e3	0.0098	
am_atan_xs	-1e3…+1e3	0.00017	
am_atan2_xs	-1e3…+1e3, -1e3…+1e3	0.0062	
am_asin_xs	-1…+1	0.016	
am_acos_xs	-1…+1	0.013	
am_exp_xs	-88…+88	0.0018	
am_exp2_xs***	-127…+127	0.0018	
am_log_xs	-1e3…+1e3	0.00069	
am_log2_xs***	-1e3…+1e3	0.00069	
am_pow_xs	-26…+26, -26…+26	0.024	

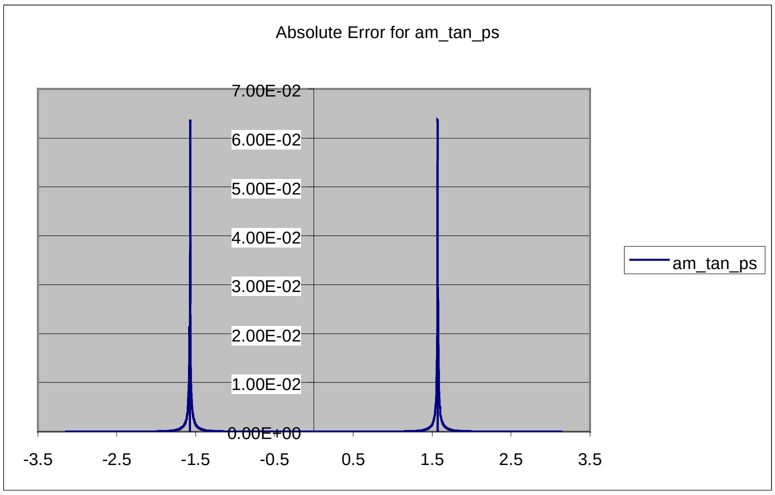
* Also applies to the sin result of `am_sincos_xs` function
** Also applies to the cos result of `am_sincos_xs` function
*** exp2 and log2 functions are not provided with the standard C math library and were implemented for the purpose of this test using appropriate x87 instructions.

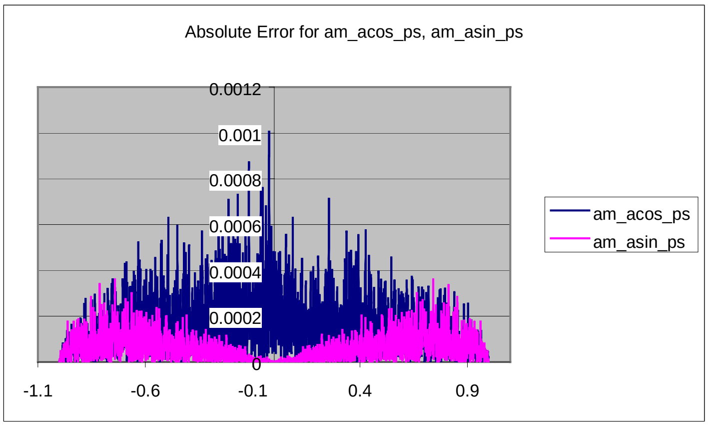
Appendix C. Absolute Errors

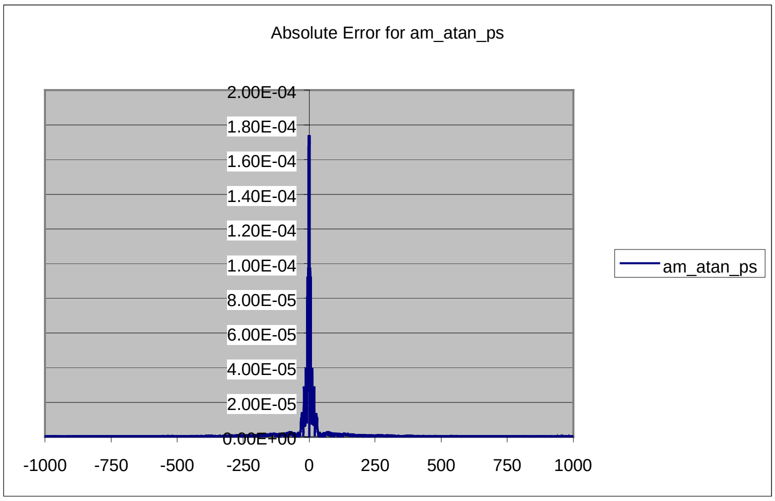
The following graphs illustrate absolute error values of packed AM Library functions. The values for scalar AM Library functions are similar. For comparison, a graph for 256-entry table lookup-based sine (function `sin_t`) is given as well.

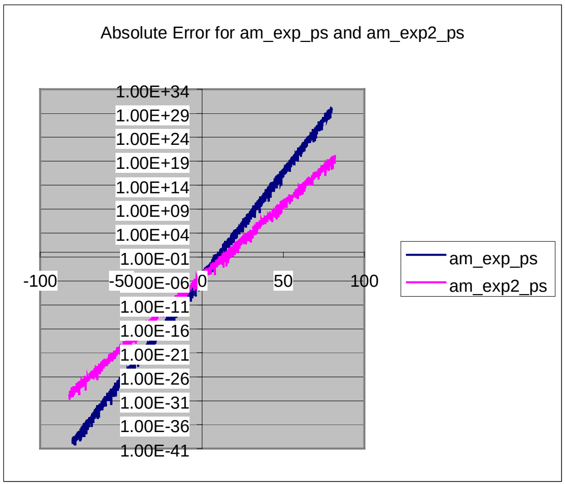


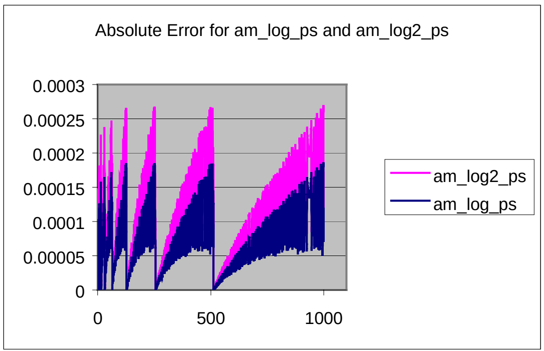



Absolute Error for am_tan_ps
7.00E-02
6.00E-02
5.00E-02
4.00E-02
3.00E-02
2.00E-02
1.00E-02
0.00E+00
-3.5
-2.5
-1.5
-0.5
0.5
1.5
2.5
3.5
am_tan_ps


Absolute Error for am_acos_ps, am_asin_ps
0.0012
0.001
0.0008
0.0006
0.0004
0.0002
0
-1.1
-0.6
-0.1
0.4
0.9
am_acos_ps
am_asin_ps


Absolute Error for am_atan_ps
2.00E-04
1.80E-04
1.60E-04
1.40E-04
1.20E-04
1.00E-04
8.00E-05
6.00E-05
4.00E-05
2.00E-05
0.00E+00
-1000
-750
-500
-250
0
250
500
750
1000
am_atan_ps





Appendix D. Valid Argument Ranges

Functions	Valid Argument Range	Behavior outside Valid Argument Range
`am_sin_xs`[1]	-3.3732593e9…+3.3732593e9 [2]	Loss of accuracy
`am_cos_xs`[3]	-3.3732593e9…+3.3732593e9 [2]	Loss of accuracy
`am_tan_xs`	-FLT_MAX…+FLT_MAX[4]	n/a[5]
`am_atan_xs`	-FLT_MAX…+FLT_MAX[4]	n/a
`am_atan2_xs`	For both arguments: -FLT_MAX…+FLT_MAX[4]	n/a[6]
`am_asin_xs`	-1…+1	Returns indefinite QNaN[7]
`am_acos_xs`	-1…+1	Returns indefinite QNaN[8]
`am_exp_xs`	-FLT_MAX[4]...+88.3762626647949	Loss of accuracy. For x > 88.3762626647949 returns 2.40619e+038 ≅ exp(88.3762626647949)
`am_exp2_xs`	-FLT_MAX[4]...+127.4999961853	Loss of accuracy. For x > 127.4999961853 returns 2.40614e+038 ≅ exp2(127.4999961853)
`am_log_xs`	1.17549e-038 [9]…+FLT_MAX[4]	Loss of accuracy. For x < 1.17549e-038 [9] returns -87.3365 ≅ log(1.17549e-038)
`am_log2_xs`	1.17549e-038 [9]…+FLT_MAX[4]	Loss of accuracy. For x < 1.17549e-038 [9] returns -126 ≅ log2(1.17549e-038)
`am_pow_xs`	For 1st argument (x): 1.17549e-038 [9]…+FLT_MAX[4] For 2nd argument (y): y * log2(x) ≤ 127.4999961853	Loss of accuracy. For x ≤ 0 returns 0. [10] For y * log2(x) > 127.4999961853 returns 2.40614e+038 ≅ exp2(127.4999961853)

[1] Also applies to the sin result of `am_sincos_xs` functions
[2] 3.3732593e9 ≅ 2.147483568e9 * π/2
[3] Also applies to the cos result of `am_sincos_xs` functions
[4] Maximum single precision floating point value, 3.402823466e+38.
[5] In poles π/2 + π*k, `am_tan_xs` functions return large positive or negative values, but not infinities. This behavior is consistent with that of x87 tan function.
[6] If the second argument is zero, `am_atanr2_xs` functions return zero. This behavior is consistent with that of x87 atan2 function.
[7] This behavior is consistent with that of x87 asin function.
[8] This behavior is consistent with that of x87 acos function.
[9] 1.17549e-038 ≅ exp2(-126) is the smallest positive single precision normalized number.

[10] Unlike x87-based pow function which accepts negative power basis with integer exponents, AM Library am_pow_xs functions do not accept negative power basis and return 0 in this case.