M A C T O R

Matrix / Vector Handling

May 1989

This is a C function set, providing fast 16-bit scalar / vector / matrix arithmetic in two and three dimensions (typically, for graphics applications).

Entities

An **element** (or **co-ordinate**, or **component**) is represented as a 16-bit two's-complement fixed-point number with the point before the 12^{th}, ..., 9^{th} binary digit; its place is given in the compile-time parameter **fixpnt**. The minimum and maximum values representable are 0x8001 and 0x7FFF, resp., interpreted according to fixpnt. 0x8000 is accepted as operand yet never produced as result. Overflow results in the corresponding extremal value (signed "machine infinity"). Division by zero is handled as overflow.

Two- and three-dimensional **vectors** and **matrices** are arrays consisting of 2, 4, 3, 9 elements, resp.; matrices are ordered rowwise. Positive rotation is ccw; three-dimensional space is right-oriented.

Types

The following types are predefined.

elem is a 16-bit signed integer quantity (normally a short), representing an element.

scalar, **vector**, **matrix**, **vect3** and **matr3** are arrays of elems (or pointers to elems) corresponding to the above entities.

Arithmetic

Functions take pointers (or (names of) arrays) as arguments. The operand is x, or, when two of them, x and y in this order; the result (if any) goes to z.

Functions return a condition code, which is an elem carrying this information. Iff any of the result's components is negative, the leftmost (or sign) bit is set; iff any of them is (strictly) positive, the rightmost (or least significant) bit is set; iff any of them has overflown, then the bit next to the rightmost is set. The restly bits are zero. (Thus the condition code is zero iff all the components are.) (Overflow means overflow in the result: whenever a resulting component fits in, it is correctly computed, even if the mathematical formula defining the operation seems to suggest that there might have been intermediate overflow.)

Function names are hoped to be self-explanatory; merely such behaviour as felt non-obvious is set forth in more detail.

```
elem     stest   (scalar x)

elem     sadd    (scalar z,scalar x,scalar y)

elem     ssub    (scalar z,scalar x,scalar y)

elem     scomp   (scalar x,scalar y)
```

Like ssub without the result (overflow may be set).

```
elem     smul    (scalar z,scalar x,scalar y)

elem     sdiv    (scalar z,scalar x,scalar y)

elem     ssqrt   (scalar z,scalar x)
```

Square-root of negative yields negative of square-root of absolute, and sets overflow.

```
elem     scos    (scalar z,scalar x)
```

Operand in radians. π is represented 0x3244>>(12-fixpnt).

```
elem     ssin    (scalar z,scalar x)
```

Like scos.

```
elem     vtest   (vector x)

elem     vadd    (vector z,vector x,vector y)

elem     vsub    (vector z,vector x,vector y)

elem     vcomp   (vector x,vector y)
```

Cf. scomp.

```
elem     vscale  (vector z,vector x,scalar y)
```

```
elem     dot     (scalar z,vector x,vector y)

elem     cross   (scalar z,vector x,vector y)

elem     ppend   (vector z,vector x)

elem     length  (scalar z,vector x)

elem     unit    (vector z,vector x)
```

A zero vector remains unchanged, with overflow set.

```
elem     cossin  (vector z,scalar x)
```

Yields both cos and sin (in this order). It is faster than computing them separately.

```
elem     angle   (scalar z,vector x)
```

Computes arctan(x[1]/x[0]), result in radians; a zero x yields zero and sets overflow.

```
elem     mtest   (matrix x)

elem     madd    (matrix z,matrix x,matrix y)

elem     msub    (matrix z,matrix x,matrix y)

elem     mcomp   (matrix x,matrix y)
```

Cf. scomp.

```
elem     mscale  (matrix z,matrix x,scalar y)

elem     mmul    (matrix z,matrix x,matrix y)

elem     adjung  (matrix z,matrix x)

elem     trpose  (matrix z,matrix x)

elem     trace   (scalar z,matrix x)

elem     det     (scalar z,matrix x)

elem     invert  (matrix z,matrix x)

elem     trform  (vector z,matrix x,vector y)

elem     strf    (vector z,vector y)
```

When transforming a sequence of vectors with the same matrix and no other mactor calls in between, a first call to trform and subsequent calls to strf speed up the computation.

```
elem     rotor   (matrix z,scalar x)
```

The result matrix rotates the vectors transformed by it; angle in radians.

```
elem     vtest3  (vect3 x)

elem     vadd3   (vect3 z,vect3 x,vect3 y)

elem     vsub3   (vect3 z,vect3 x,vect3 y)

elem     vcomp3  (vect3 x,vect3 y)
```

Cf. scomp.

```
elem     vscal3  (vect3 z,vect3 x,scalar y)

elem     dot3    (scalar z,vect3 x,vect3 y)

elem     cross3  (vect3 z,vect3 x,vect3 y)

elem     lngth3  (scalar z,vect3 x)

elem     unit3   (vect3 z,vect3 x)
```

Like unit above.

```
elem     mtest3  (matr3 x)

elem     madd3   (matr3 z,matr3 x,matr3 y)

elem     msub3   (matr3 z,matr3 x,matr3 y)

elem     mcomp3  (matr3 x,matr3 y)
```

Cf. scomp.

```
elem     mscal3  (matr3 z,matr3 x,scalar y)

elem     mmul3   (matr3 z,matr3 x,matr3 y)

elem     adjng3  (matr3 z,matr3 x)

elem     trpos3  (matr3 z,matr3 x)

elem     trace3  (scalar z,matr3 x)

elem     miv3    (scalar z,matr3 x)
```

Middle invariant: the one in the co-efficient of the linear term in the characteristic polynomial.

```
elem     det3    (scalar z,matr3 x)

elem     invrt3  (matr3 z,matr3 x)
```

```
elem     trfrm3  (vect3 z,matr3 x,vect3 y)

elem     strf3   (vect3 z,vect3 y)
```

Related to trfrm3 as strf to trform.

```
elem     rot03   (matr3 z,scalar x)
```

Rotator around the 0^{th} axis; cf. rotor above.

```
elem     rot13   (matr3 z,scalar x)
```

Around the 1^{st} axis.

```
elem     rot23   (matr3 z,scalar x)
```

Around the 2^{nd} axis.

N o t e

Where there is no danger of overflow, ordinary C addition and subtraction may be applied between components; similarly, shifts and multiplication or division by ordinary C integer-type values would yield the expected results. This can save time and space.

Conversions

Conversion is between an elem and a double (n o t pointers).

```
elem     sfix    (double d)
```

If not fitting in, signed "machine infinity" is given.

```
double   sflt    (elem e)
```

Structure

The code consists of two files: "mactor.c", which is (absolutely) clean C, and "mactas.c", which comprises everything that is implementation-dependent. ("mactas" consists of "#define" bodies, comprising about 90 lines of assembler code, chiefly concerned with overflow handling, a thing inordinately expensive if done in C.) - Presently, the only "mactas" existing is the one for the IBM PC 8086 Macro Assembler, accepted by Turbo C (tcc, using -B).

Compilation parameters, placed in the beginning of "mactor", are **fixpnt**, spoken of above, and **dim3**, which in- or excludes three-dimensional features according to its being non-zero or zero.

To use "mactor", include it in the program to be compiled ("mactor" includes "mactas"). - A feasible usage on PC/XT/AT is to precompile "mactor" with -B and -c (and maybe some memory model prescription), forming an obj file that can be linked; "mactdc.c" contains all the public declarations needed by, and to be included in, the program using "mactor".

Mactor functions must not be used in interrupt routines possibly interrupting (the same copy of) mactor functions. If such a working is ever needed, the possibly endangered static variables x_n, y_n, z_n and c are to be saved.

Size

The Turbo C version including dim3 is about 33 KBytes; without dim3, 14 KBytes. The size can be very considerably reduced at a little cost in speed (simply by turning part of the macros into functions).

Speed

Execution times depend to a certain extent on the operands. As an orientation, some averages on typical functions with typical data are listed below, using an AT with Norton performance index 11.7; times are in microsecs.

sadd	25	smul	41	cossin	171	ssqrt	72
length	106	unit	134	dot	50	cross	52
mmul	138	det	49	invert	134	trform	81
lngth3	189	unit3	231	dot3	60	cross3	114
mmul3	360	det3	144	invrt3	655	trfrm3	143

=====

Error accumulation in limited-precision scalar arithmetic is now a fairly well understood topic, having found its way into computer nursery wisdom, like "independent errors add up to square-root order", or "try to avoid small differences of large numbers", or "multiplication/division is comparatively harmless, at least if done in floating-point rather than fixed-point". With multidimensional arithmetic, additional subtleties turn up. Per-

haps the simplest example is this. Take a rotator matrix containing sine and cosine values with rounding errors. While applying it repeatedly, angle errors add up, whereas an error in the determinant has an exponential effect on sizes. (So it might be quite sound to make components worse in order to make the determinant better.) Sometimes, computational stability can be achieved by simple but rather unconventional methods (such as "antitruncating", that is, always rounding a w a y from zero); sometimes, it is inherently lacking. In fact, no comprehensive theory is available; the best we have is case studies. A rule of thumb might be this: Anything that can be computed directly ought to be computed that way rather than through a series of un-analyzed iterations; "absolute is preferable to relative". Except when otherwise.