1. Introduction.

```
#include <w2c/config.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>
#include "mpmathdouble.h"    /* internal header */
#define ROUND(a) floor ((a) + 0.5)
  ⟨ Preprocessor definitions ⟩
```

2. ⟨ Declarations 5 ⟩;

3. ⟨ mpmathdouble.h 3 ⟩ ≡

```
#ifndef MPMATHDOUBLE_H
#define MPMATHDOUBLE_H 1
#include "mplib.h"
#include "mpmp.h"    /* internal header */
  ⟨ Internal library declarations 6 ⟩;
#endif
```

4. Math initialization.

First, here are some very important constants.

#define `PI` 3.14159265358979323846264338327950288841971
#define *fraction_multiplier* 4096.0
#define *angle_multiplier* 16.0

5. Here are the functions that are static as they are not used elsewhere

⟨ Declarations 5 ⟩ ≡

static void *mp_double_scan_fractional_token*(MP *mp*, **int** *n*);
static void *mp_double_scan_numeric_token*(MP *mp*, **int** *n*);
static void *mp_ab_vs_cd*(MP *mp*, *mp_number* ∗ *ret*, *mp_number a*, *mp_number b*, *mp_number c*, *mp_number d*);
static void *mp_double_ab_vs_cd*(MP *mp*, *mp_number* ∗ *ret*, *mp_number a*, *mp_number b*, *mp_number c*, *mp_number d*);
static void *mp_double_crossing_point*(MP *mp*, *mp_number* ∗ *ret*, *mp_number a*, *mp_number b*, *mp_number c*);
static void *mp_number_modulo*(*mp_number* ∗ *a*, *mp_number b*);
static void *mp_double_print_number*(MP *mp*, *mp_number n*);
static char ∗*mp_double_number_tostring*(MP *mp*, *mp_number n*);
static void *mp_double_slow_add*(MP *mp*, *mp_number* ∗ *ret*, *mp_number x_orig*, *mp_number y_orig*);
static void *mp_double_square_rt*(MP *mp*, *mp_number* ∗ *ret*, *mp_number x_orig*);
static void *mp_double_sin_cos*(MP *mp*, *mp_number z_orig*, *mp_number* ∗ *n_cos*, *mp_number* ∗ *n_sin*);
static void *mp_init_randoms*(MP *mp*, **int** *seed*);
static void *mp_number_angle_to_scaled*(*mp_number* ∗ *A*);
static void *mp_number_fraction_to_scaled*(*mp_number* ∗ *A*);
static void *mp_number_scaled_to_fraction*(*mp_number* ∗ *A*);
static void *mp_number_scaled_to_angle*(*mp_number* ∗ *A*);
static void *mp_double_m_unif_rand*(MP *mp*, *mp_number* ∗ *ret*, *mp_number x_orig*);
static void *mp_double_m_norm_rand*(MP *mp*, *mp_number* ∗ *ret*);
static void *mp_double_m_exp*(MP *mp*, *mp_number* ∗ *ret*, *mp_number x_orig*);
static void *mp_double_m_log*(MP *mp*, *mp_number* ∗ *ret*, *mp_number x_orig*);
static void *mp_double_pyth_sub*(MP *mp*, *mp_number* ∗ *r*, *mp_number a*, *mp_number b*);
static void *mp_double_pyth_add*(MP *mp*, *mp_number* ∗ *r*, *mp_number a*, *mp_number b*);
static void *mp_double_n_arg*(MP *mp*, *mp_number* ∗ *ret*, *mp_number x*, *mp_number y*);
static void *mp_double_velocity*(MP *mp*, *mp_number* ∗ *ret*, *mp_number st*, *mp_number ct*, *mp_number sf*, *mp_number cf*, *mp_number t*);
static void *mp_set_double_from_int*(*mp_number* ∗ *A*, **int** *B*);
static void *mp_set_double_from_boolean*(*mp_number* ∗ *A*, **int** *B*);
static void *mp_set_double_from_scaled*(*mp_number* ∗ *A*, **int** *B*);
static void *mp_set_double_from_addition*(*mp_number* ∗ *A*, *mp_number B*, *mp_number C*);
static void *mp_set_double_from_substraction*(*mp_number* ∗ *A*, *mp_number B*, *mp_number C*);
static void *mp_set_double_from_div*(*mp_number* ∗ *A*, *mp_number B*, *mp_number C*);
static void *mp_set_double_from_mul*(*mp_number* ∗ *A*, *mp_number B*, *mp_number C*);
static void *mp_set_double_from_int_div*(*mp_number* ∗ *A*, *mp_number B*, **int** *C*);
static void *mp_set_double_from_int_mul*(*mp_number* ∗ *A*, *mp_number B*, **int** *C*);
static void *mp_set_double_from_of_the_way*(MP *mp*, *mp_number* ∗ *A*, *mp_number t*, *mp_number B*, *mp_number C*);
static void *mp_number_negate*(*mp_number* ∗ *A*);
static void *mp_number_add*(*mp_number* ∗ *A*, *mp_number B*);
static void *mp_number_substract*(*mp_number* ∗ *A*, *mp_number B*);
static void *mp_number_half*(*mp_number* ∗ *A*);
static void *mp_number_halfp*(*mp_number* ∗ *A*);
static void *mp_number_double*(*mp_number* ∗ *A*);
static void *mp_number_add_scaled*(*mp_number* ∗ *A*, **int** *B*); /∗ also for negative B ∗/
static void *mp_number_multiply_int*(*mp_number* ∗ *A*, **int** *B*);
static void *mp_number_divide_int*(*mp_number* ∗ *A*, **int** *B*);
static void *mp_double_abs*(*mp_number* ∗ *A*);
static void *mp_number_clone*(*mp_number* ∗ *A*, *mp_number B*);
static void *mp_number_swap*(*mp_number* ∗ *A*, *mp_number* ∗ *B*);
static int *mp_round_unscaled*(*mp_number x_orig*);

```
static int mp_number_to_int(mp_number A);
static int mp_number_to_scaled(mp_number A);
static int mp_number_to_boolean(mp_number A);
static double mp_number_to_double(mp_number A);
static int mp_number_odd(mp_number A);
static int mp_number_equal(mp_number A, mp_number B);
static int mp_number_greater(mp_number A, mp_number B);
static int mp_number_less(mp_number A, mp_number B);
static int mp_number_nonequalabs(mp_number A, mp_number B);
static void mp_number_floor(mp_number *i);
static void mp_double_fraction_to_round_scaled(mp_number *x);
static void mp_double_number_make_scaled(MP mp, mp_number *r, mp_number p, mp_number q);
static void mp_double_number_make_fraction(MP mp, mp_number *r, mp_number p, mp_number q);
static void mp_double_number_take_fraction(MP mp, mp_number *r, mp_number p, mp_number q);
static void mp_double_number_take_scaled(MP mp, mp_number *r, mp_number p, mp_number q);
static void mp_new_number(MP mp, mp_number *n, mp_number_type t);
static void mp_free_number(MP mp, mp_number *n);
static void mp_set_double_from_double(mp_number *A, double B);
static void mp_free_double_math(MP mp);
static void mp_double_set_precision(MP mp);
```

See also sections 19, 21, 24, and 26.

This code is used in section 2.

6. And these are the ones that *are* used elsewhere

⟨ Internal library declarations 6 ⟩ ≡

```
void *mp_initialize_double_math(MP mp);
```

This code is used in section 3.

7.

```
#define coef_bound ((7.0/3.0) * fraction_multiplier)  /* fraction approximation to 7/3 */
#define fraction_threshold 0.04096  /* a fraction coefficient less than this is zeroed */
#define half_fraction_threshold (fraction_threshold/2)  /* half of fraction_threshold */
#define scaled_threshold 0.000122  /* a scaled coefficient less than this is zeroed */
#define half_scaled_threshold (scaled_threshold/2)  /* half of scaled_threshold */
#define near_zero_angle (0.0256 * angle_multiplier)  /* an angle of about 0.0256 */
#define p_over_v_threshold #80000  /* TODO */
#define equation_threshold 0.001
#define tfm_warn_threshold 0.0625
#define warning_limit pow(2.0, 52.0)
    /* this is a large value that can just be expressed without loss of precision */
#define epsilon pow(2.0, -52.0)

  void *mp_initialize_double_math(MP mp){ math_data *math = ( math_data * ) mp_xmalloc(mp, 1, sizeof
      (math_data));  /* alloc */
    math->allocate = mp_new_number;
    math->free = mp_free_number;
    mp_new_number(mp, &math->precision_default, mp_scaled_type);
    math->precision_default.data.dval = 16 * unity;
    mp_new_number(mp, &math->precision_max, mp_scaled_type);
    math->precision_max.data.dval = 16 * unity;
    mp_new_number(mp, &math->precision_min, mp_scaled_type);
    math->precision_min.data.dval = 16 * unity;  /* here are the constants for scaled objects */
    mp_new_number(mp, &math->epsilon_t, mp_scaled_type);
    math->epsilon_t.data.dval = epsilon;
    mp_new_number(mp, &math->inf_t, mp_scaled_type);
    math->inf_t.data.dval = EL_GORDO;
    mp_new_number(mp, &math->warning_limit_t, mp_scaled_type);
    math->warning_limit_t.data.dval = warning_limit;
    mp_new_number(mp, &math->one_third_inf_t, mp_scaled_type);
    math->one_third_inf_t.data.dval = one_third_EL_GORDO;
    mp_new_number(mp, &math->unity_t, mp_scaled_type);
    math->unity_t.data.dval = unity;
    mp_new_number(mp, &math->two_t, mp_scaled_type);
    math->two_t.data.dval = two;
    mp_new_number(mp, &math->three_t, mp_scaled_type);
    math->three_t.data.dval = three;
    mp_new_number(mp, &math->half_unit_t, mp_scaled_type);
    math->half_unit_t.data.dval = half_unit;
    mp_new_number(mp, &math->three_quarter_unit_t, mp_scaled_type);
    math->three_quarter_unit_t.data.dval = three_quarter_unit;
    mp_new_number(mp, &math->zero_t, mp_scaled_type);  /* fractions */
    mp_new_number(mp, &math->arc_tol_k, mp_fraction_type);
    math->arc_tol_k.data.dval = (unity/4096);
      /* quit when change in arc length estimate reaches this */
    mp_new_number(mp, &math->fraction_one_t, mp_fraction_type);
    math->fraction_one_t.data.dval = fraction_one;
    mp_new_number(mp, &math->fraction_half_t, mp_fraction_type);
    math->fraction_half_t.data.dval = fraction_half;
    mp_new_number(mp, &math->fraction_three_t, mp_fraction_type);
    math->fraction_three_t.data.dval = fraction_three;
    mp_new_number(mp, &math->fraction_four_t, mp_fraction_type);
```

$math\rightarrow fraction\_four\_t.data.dval = fraction\_four$; /* *angles* */
$mp\_new\_number(mp, \&math\rightarrow three\_sixty\_deg\_t, mp\_angle\_type)$;
$math\rightarrow three\_sixty\_deg\_t.data.dval = three\_sixty\_deg$;
$mp\_new\_number(mp, \&math\rightarrow one\_eighty\_deg\_t, mp\_angle\_type)$;
$math\rightarrow one\_eighty\_deg\_t.data.dval = one\_eighty\_deg$; /* various approximations */
$mp\_new\_number(mp, \&math\rightarrow one\_k, mp\_scaled\_type)$;
$math\rightarrow one\_k.data.dval = 1.0/64$;
$mp\_new\_number(mp, \&math\rightarrow sqrt\_8\_e\_k, mp\_scaled\_type)$;
$math\rightarrow sqrt\_8\_e\_k.data.dval = 1.71552776992141359295$; /* $2^{16}\sqrt{8/e} \approx 112428.82793$ */
$mp\_new\_number(mp, \&math\rightarrow twelve\_ln\_2\_k, mp\_fraction\_type)$;
$math\rightarrow twelve\_ln\_2\_k.data.dval = 8.31776616671934371292 * 256$;
/* $2^{24} \cdot 12 \ln 2 \approx 139548959.6165$ */
$mp\_new\_number(mp, \&math\rightarrow coef\_bound\_k, mp\_fraction\_type)$;
$math\rightarrow coef\_bound\_k.data.dval = coef\_bound$;
$mp\_new\_number(mp, \&math\rightarrow coef\_bound\_minus\_1, mp\_fraction\_type)$;
$math\rightarrow coef\_bound\_minus\_1.data.dval = coef\_bound - 1/65536.0$;
$mp\_new\_number(mp, \&math\rightarrow twelvebits\_3, mp\_scaled\_type)$;
$math\rightarrow twelvebits\_3.data.dval = 1365/65536.0$; /* $1365 \approx 2^{12}/3$ */
$mp\_new\_number(mp, \&math\rightarrow twentysixbits\_sqrt2\_t, mp\_fraction\_type)$;
$math\rightarrow twentysixbits\_sqrt2\_t.data.dval = 94906266/65536.0$; /* $2^{26}\sqrt{2} \approx 94906265.62$ */
$mp\_new\_number(mp, \&math\rightarrow twentyeightbits\_d\_t, mp\_fraction\_type)$;
$math\rightarrow twentyeightbits\_d\_t.data.dval = 35596755/65536.0$; /* $2^{28}d \approx 35596754.69$ */
$mp\_new\_number(mp, \&math\rightarrow twentysevenbits\_sqrt2\_d\_t, mp\_fraction\_type)$;
$math\rightarrow twentysevenbits\_sqrt2\_d\_t.data.dval = 25170707/65536.0$; /* $2^{27}\sqrt{2}\,d \approx 25170706.63$ */
/* thresholds */
$mp\_new\_number(mp, \&math\rightarrow fraction\_threshold\_t, mp\_fraction\_type)$;
$math\rightarrow fraction\_threshold\_t.data.dval = fraction\_threshold$;
$mp\_new\_number(mp, \&math\rightarrow half\_fraction\_threshold\_t, mp\_fraction\_type)$;
$math\rightarrow half\_fraction\_threshold\_t.data.dval = half\_fraction\_threshold$;
$mp\_new\_number(mp, \&math\rightarrow scaled\_threshold\_t, mp\_scaled\_type)$;
$math\rightarrow scaled\_threshold\_t.data.dval = scaled\_threshold$;
$mp\_new\_number(mp, \&math\rightarrow half\_scaled\_threshold\_t, mp\_scaled\_type)$;
$math\rightarrow half\_scaled\_threshold\_t.data.dval = half\_scaled\_threshold$;
$mp\_new\_number(mp, \&math\rightarrow near\_zero\_angle\_t, mp\_angle\_type)$;
$math\rightarrow near\_zero\_angle\_t.data.dval = near\_zero\_angle$;
$mp\_new\_number(mp, \&math\rightarrow p\_over\_v\_threshold\_t, mp\_fraction\_type)$;
$math\rightarrow p\_over\_v\_threshold\_t.data.dval = p\_over\_v\_threshold$;
$mp\_new\_number(mp, \&math\rightarrow equation\_threshold\_t, mp\_scaled\_type)$;
$math\rightarrow equation\_threshold\_t.data.dval = equation\_threshold$;
$mp\_new\_number(mp, \&math\rightarrow tfm\_warn\_threshold\_t, mp\_scaled\_type)$;
$math\rightarrow tfm\_warn\_threshold\_t.data.dval = tfm\_warn\_threshold$; /* functions */
$math\rightarrow from\_int = mp\_set\_double\_from\_int$;
$math\rightarrow from\_boolean = mp\_set\_double\_from\_boolean$;
$math\rightarrow from\_scaled = mp\_set\_double\_from\_scaled$;
$math\rightarrow from\_double = mp\_set\_double\_from\_double$;
$math\rightarrow from\_addition = mp\_set\_double\_from\_addition$;
$math\rightarrow from\_substraction = mp\_set\_double\_from\_substraction$;
$math\rightarrow from\_oftheway = mp\_set\_double\_from\_of\_the\_way$;
$math\rightarrow from\_div = mp\_set\_double\_from\_div$;
$math\rightarrow from\_mul = mp\_set\_double\_from\_mul$;
$math\rightarrow from\_int\_div = mp\_set\_double\_from\_int\_div$;
$math\rightarrow from\_int\_mul = mp\_set\_double\_from\_int\_mul$;

$math\rightarrow negate = mp\_number\_negate;$
$math\rightarrow add = mp\_number\_add;$
$math\rightarrow substract = mp\_number\_substract;$
$math\rightarrow half = mp\_number\_half;$
$math\rightarrow halfp = mp\_number\_halfp;$
$math\rightarrow do\_double = mp\_number\_double;$
$math\rightarrow abs = mp\_double\_abs;$
$math\rightarrow clone = mp\_number\_clone;$
$math\rightarrow swap = mp\_number\_swap;$
$math\rightarrow add\_scaled = mp\_number\_add\_scaled;$
$math\rightarrow multiply\_int = mp\_number\_multiply\_int;$
$math\rightarrow divide\_int = mp\_number\_divide\_int;$
$math\rightarrow to\_boolean = mp\_number\_to\_boolean;$
$math\rightarrow to\_scaled = mp\_number\_to\_scaled;$
$math\rightarrow to\_double = mp\_number\_to\_double;$
$math\rightarrow to\_int = mp\_number\_to\_int;$
$math\rightarrow odd = mp\_number\_odd;$
$math\rightarrow equal = mp\_number\_equal;$
$math\rightarrow less = mp\_number\_less;$
$math\rightarrow greater = mp\_number\_greater;$
$math\rightarrow nonequalabs = mp\_number\_nonequalabs;$
$math\rightarrow round\_unscaled = mp\_round\_unscaled;$
$math\rightarrow floor\_scaled = mp\_number\_floor;$
$math\rightarrow fraction\_to\_round\_scaled = mp\_double\_fraction\_to\_round\_scaled;$
$math\rightarrow make\_scaled = mp\_double\_number\_make\_scaled;$
$math\rightarrow make\_fraction = mp\_double\_number\_make\_fraction;$
$math\rightarrow take\_fraction = mp\_double\_number\_take\_fraction;$
$math\rightarrow take\_scaled = mp\_double\_number\_take\_scaled;$
$math\rightarrow velocity = mp\_double\_velocity;$
$math\rightarrow n\_arg = mp\_double\_n\_arg;$
$math\rightarrow m\_log = mp\_double\_m\_log;$
$math\rightarrow m\_exp = mp\_double\_m\_exp;$
$math\rightarrow m\_unif\_rand = mp\_double\_m\_unif\_rand;$
$math\rightarrow m\_norm\_rand = mp\_double\_m\_norm\_rand;$
$math\rightarrow pyth\_add = mp\_double\_pyth\_add;$
$math\rightarrow pyth\_sub = mp\_double\_pyth\_sub;$
$math\rightarrow fraction\_to\_scaled = mp\_number\_fraction\_to\_scaled;$
$math\rightarrow scaled\_to\_fraction = mp\_number\_scaled\_to\_fraction;$
$math\rightarrow scaled\_to\_angle = mp\_number\_scaled\_to\_angle;$
$math\rightarrow angle\_to\_scaled = mp\_number\_angle\_to\_scaled;$
$math\rightarrow init\_randoms = mp\_init\_randoms;$
$math\rightarrow sin\_cos = mp\_double\_sin\_cos;$
$math\rightarrow slow\_add = mp\_double\_slow\_add;$
$math\rightarrow sqrt = mp\_double\_square\_rt;$
$math\rightarrow print = mp\_double\_print\_number;$
$math\rightarrow tostring = mp\_double\_number\_tostring;$
$math\rightarrow modulo = mp\_number\_modulo;$
$math\rightarrow ab\_vs\_cd = mp\_ab\_vs\_cd;$
$math\rightarrow crossing\_point = mp\_double\_crossing\_point;$
$math\rightarrow scan\_numeric = mp\_double\_scan\_numeric\_token;$
$math\rightarrow scan\_fractional = mp\_double\_scan\_fractional\_token;$
$math\rightarrow free\_math = mp\_free\_double\_math;$

$math\rightarrow set\_precision = mp\_double\_set\_precision$;
return (**void** $*$) $math$; } **void** $mp\_double\_set\_precision$(MP mp)
{ } **void** $mp\_free\_double\_math$(MP mp){ $free\_number$ ((($math\_data$ $*$) $mp\rightarrow math$) $\rightarrow$ $three\_sixty\_deg\_t$) ; $free\_number$ ((($math\_data$ $*$) $mp\rightarrow math$) $\rightarrow$ $one\_eighty\_deg\_t$) ; $free\_number$ ((($math\_data$ $*$) $mp\rightarrow math$) $\rightarrow$ $fraction\_one\_t$) ; $free\_number$ ((($math\_data$ $*$) $mp\rightarrow math$) $\rightarrow$ $zero\_t$) ; $free\_number$ ((($math\_data$ $*$) $mp\rightarrow math$) $\rightarrow$ $half\_unit\_t$) ; $free\_number$ ((($math\_data$ $*$) $mp\rightarrow math$) $\rightarrow$ $three\_quarter\_unit\_t$) ; $free\_number$ ((($math\_data$ $*$) $mp\rightarrow math$) $\rightarrow$ $unity\_t$) ; $free\_number$ ((($math\_data$ $*$) $mp\rightarrow math$) $\rightarrow$ two_t) ; $free\_number$ ((($math\_data$ $*$) $mp\rightarrow math$) $\rightarrow$ $three\_t$) ; $free\_number$ ((($math\_data$ $*$) $mp\rightarrow math$) $\rightarrow$ $one\_third\_inf\_t$) ; $free\_number$ ((($math\_data$ $*$) $mp\rightarrow math$) $\rightarrow$ inf_t) ; $free\_number$ ((($math\_data$ $*$) $mp\rightarrow math$) $\rightarrow$ $warning\_limit\_t$) ; $free\_number$ ((($math\_data$ $*$) $mp\rightarrow math$) $\rightarrow$ one_k) ; $free\_number$ ((($math\_data$ $*$) $mp\rightarrow math$) $\rightarrow$ $sqrt\_8\_e\_k$) ; $free\_number$ ((($math\_data$ $*$) $mp\rightarrow math$) $\rightarrow$ $twelve\_ln\_2\_k$) ; $free\_number$ ((($math\_data$ $*$) $mp\rightarrow math$) $\rightarrow$ $coef\_bound\_k$) ; $free\_number$ ((($math\_data$ $*$) $mp\rightarrow math$) $\rightarrow$ $coef\_bound\_minus\_1$) ; $free\_number$ ((($math\_data$ $*$) $mp\rightarrow math$) $\rightarrow$ $fraction\_threshold\_t$) ; $free\_number$ ((($math\_data$ $*$) $mp\rightarrow math$) $\rightarrow$ $half\_fraction\_threshold\_t$) ; $free\_number$ ((($math\_data$ $*$) $mp\rightarrow math$) $\rightarrow$ $scaled\_threshold\_t$) ; $free\_number$ ((($math\_data$ $*$) $mp\rightarrow math$) $\rightarrow$ $half\_scaled\_threshold\_t$) ; $free\_number$ ((($math\_data$ $*$) $mp\rightarrow math$) $\rightarrow$ $near\_zero\_angle\_t$) ; $free\_number$ ((($math\_data$ $*$) $mp\rightarrow math$) $\rightarrow$ $p\_over\_v\_threshold\_t$) ; $free\_number$ ((($math\_data$ $*$) $mp\rightarrow math$) $\rightarrow$ $equation\_threshold\_t$) ; $free\_number$ ((($math\_data$ $*$) $mp\rightarrow math$) $\rightarrow$ $tfm\_warn\_threshold\_t$) ;
$free(mp\rightarrow math)$; }

8. Creating an destroying *mp_number* objects

9. **void** mp_new_number(MP mp, $mp\_number * n$, $mp\_number\_type\, t$)
{
(**void**) mp;
$n\rightarrow data.dval = 0.0$;
$n\rightarrow type = t$;
}

10.

void mp_free_number(MP mp, $mp\_number * n$)
{
(**void**) mp;
$n\rightarrow type = mp\_nan\_type$;
}

11. Here are the low-level functions on *mp_number* items, setters first.

```
void mp_set_double_from_int(mp_number *A, int B)
{
  A→data.dval = B;
}
void mp_set_double_from_boolean(mp_number *A, int B)
{
  A→data.dval = B;
}
void mp_set_double_from_scaled(mp_number *A, int B)
{
  A→data.dval = B/65536.0;
}
void mp_set_double_from_double(mp_number *A, double B)
{
  A→data.dval = B;
}
void mp_set_double_from_addition(mp_number *A, mp_number B, mp_number C)
{
  A→data.dval = B.data.dval + C.data.dval;
}
void mp_set_double_from_substraction(mp_number *A, mp_number B, mp_number C)
{
  A→data.dval = B.data.dval − C.data.dval;
}
void mp_set_double_from_div(mp_number *A, mp_number B, mp_number C)
{
  A→data.dval = B.data.dval/C.data.dval;
}
void mp_set_double_from_mul(mp_number *A, mp_number B, mp_number C)
{
  A→data.dval = B.data.dval * C.data.dval;
}
void mp_set_double_from_int_div(mp_number *A, mp_number B, int C)
{
  A→data.dval = B.data.dval/C;
}
void mp_set_double_from_int_mul(mp_number *A, mp_number B, int C)
{
  A→data.dval = B.data.dval * C;
}
void mp_set_double_from_of_the_way(MP mp, mp_number *A, mp_number t, mp_number B, mp_number C)
{
  A→data.dval = B.data.dval − mp_double_take_fraction(mp, (B.data.dval − C.data.dval), t.data.dval);
}
void mp_number_negate(mp_number *A)
{
  A→data.dval = −A→data.dval;
  if (A→data.dval ≡ −0.0) A→data.dval = 0.0;
```

```
}
void mp_number_add(mp_number *A, mp_number B)
{
  A->data.dval = A->data.dval + B.data.dval;
}
void mp_number_substract(mp_number *A, mp_number B)
{
  A->data.dval = A->data.dval - B.data.dval;
}
void mp_number_half(mp_number *A)
{
  A->data.dval = A->data.dval/2.0;
}
void mp_number_halfp(mp_number *A)
{
  A->data.dval = (A->data.dval/2.0);
}
void mp_number_double(mp_number *A)
{
  A->data.dval = A->data.dval * 2.0;
}
void mp_number_add_scaled(mp_number *A, int B)
{     /* also for negative B */
  A->data.dval = A->data.dval + (B/65536.0);
}
void mp_number_multiply_int(mp_number *A, int B)
{
  A->data.dval = (double)(A->data.dval * B);
}
void mp_number_divide_int(mp_number *A, int B)
{
  A->data.dval = A->data.dval/(double) B;
}
void mp_double_abs(mp_number *A)
{
  A->data.dval = fabs(A->data.dval);
}
void mp_number_clone(mp_number *A, mp_number B)
{
  A->data.dval = B.data.dval;
}
void mp_number_swap(mp_number *A, mp_number *B)
{
  double swap_tmp = A->data.dval;
  A->data.dval = B->data.dval;
  B->data.dval = swap_tmp;
}
void mp_number_fraction_to_scaled(mp_number *A)
{
```

```
    A→type = mp_scaled_type;
    A→data.dval = A→data.dval / fraction_multiplier;
  }
  void mp_number_angle_to_scaled(mp_number * A)
  {
    A→type = mp_scaled_type;
    A→data.dval = A→data.dval / angle_multiplier;
  }
  void mp_number_scaled_to_fraction(mp_number * A)
  {
    A→type = mp_fraction_type;
    A→data.dval = A→data.dval * fraction_multiplier;
  }
  void mp_number_scaled_to_angle(mp_number * A)
  {
    A→type = mp_angle_type;
    A→data.dval = A→data.dval * angle_multiplier;
  }
```

12. Query functions

```
int mp_number_to_scaled(mp_number A)
{
  return (int) ROUND(A.data.dval * 65536.0);
}
int mp_number_to_int(mp_number A)
{
  return (int)(A.data.dval);
}
int mp_number_to_boolean(mp_number A)
{
  return (int)(A.data.dval);
}
double mp_number_to_double(mp_number A)
{
  return A.data.dval;
}
int mp_number_odd(mp_number A)
{
  return odd((int) ROUND(A.data.dval * 65536.0));
}
int mp_number_equal(mp_number A, mp_number B)
{
  return (A.data.dval ≡ B.data.dval);
}
int mp_number_greater(mp_number A, mp_number B)
{
  return (A.data.dval > B.data.dval);
}
int mp_number_less(mp_number A, mp_number B)
{
  return (A.data.dval < B.data.dval);
}
int mp_number_nonequalabs(mp_number A, mp_number B)
{
  return (¬(fabs(A.data.dval) ≡ fabs(B.data.dval)));
}
```

13. Fixed-point arithmetic is done on *scaled integers* that are multiples of 2^{-16}. In other words, a binary point is assumed to be sixteen bit positions from the right end of a binary computer word.

```
#define unity 1.0
#define two 2.0
#define three 3.0
#define half_unit 0.5
#define three_quarter_unit 0.75
#define EL_GORDO (DBL_MAX/2.0 − 1.0)   /* the largest value that METAPOST likes. */
#define one_third_EL_GORDO (EL_GORDO/3.0)
```

14. One of METAPOST's most common operations is the calculation of $\lfloor \frac{a+b}{2} \rfloor$, the midpoint of two given integers a and b. The most decent way to do this is to write '$(a+b)/2$'; but on many machines it is more efficient to calculate '$(a+b) \gg 1$'.

Therefore the midpoint operation will always be denoted by '*half* $(a+b)$' in this program. If METAPOST is being implemented with languages that permit binary shifting, the *half* macro should be changed to make this operation as efficient as possible. Since some systems have shift operators that can only be trusted to work on positive numbers, there is also a macro *halfp* that is used only when the quantity being halved is known to be positive or zero.

15. Here is a procedure analogous to *print_int*. The current version is fairly stupid, and it is not round-trip safe, but this is good enough for a beta test.

```
char *mp_double_number_tostring(MP mp, mp_number n)
{
  static char set[64];
  int l = 0;
  char *ret = mp_xmalloc(mp, 64, 1);
  snprintf(set, 64, "%.17g", n.data.dval);
  while (set[l] ≡ '␣') l++;
  strcpy(ret, set + l);
  return ret;
}
```

16.
```
void mp_double_print_number(MP mp, mp_number n)
{
  char *str = mp_double_number_tostring(mp, n);
  mp_print(mp, str);
  free(str);
}
```

17. Addition is not always checked to make sure that it doesn't overflow, but in places where overflow isn't too unlikely the *slow_add* routine is used.

```
void mp_double_slow_add(MP mp, mp_number *ret, mp_number x_orig, mp_number y_orig)
{
  double x, y;
  x = x_orig.data.dval;
  y = y_orig.data.dval;
  if (x ≥ 0) {
    if (y ≤ EL_GORDO − x) {
      ret→data.dval = x + y;
    }
    else {
      mp→arith_error = true;
      ret→data.dval = EL_GORDO;
    }
  }
  else if (−y ≤ EL_GORDO + x) {
    ret→data.dval = x + y;
  }
  else {
    mp→arith_error = true;
    ret→data.dval = −EL_GORDO;
  }
}
```

18. The *make_fraction* routine produces the *fraction* equivalent of p/q, given integers p and q; it computes the integer $f = \lfloor 2^{28}p/q + \frac{1}{2}\rfloor$, when p and q are positive. If p and q are both of the same scaled type t, the "type relation" $make\_fraction(t,t) = fraction$ is valid; and it's also possible to use the subroutine "backwards," using the relation $make\_fraction(t, fraction) = t$ between scaled types.

If the result would have magnitude 2^{31} or more, *make_fraction* sets $arith\_error := true$. Most of METAPOST's internal computations have been designed to avoid this sort of error.

If this subroutine were programmed in assembly language on a typical machine, we could simply compute $(2^{28} * p)\, div\, q$, since a double-precision product can often be input to a fixed-point division instruction. But when we are restricted to int-eger arithmetic it is necessary either to resort to multiple-precision maneuvering or to use a simple but slow iteration. The multiple-precision technique would be about three times faster than the code adopted here, but it would be comparatively long and tricky, involving about sixteen additional multiplications and divisions.

This operation is part of METAPOST's "inner loop"; indeed, it will consume nearly 10% of the running time (exclusive of input and output) if the code below is left unchanged. A machine-dependent recoding will therefore make METAPOST run faster. The present implementation is highly portable, but slow; it avoids multiplication and division except in the initial stage. System wizards should be careful to replace it with a routine that is guaranteed to produce identical results in all cases.

As noted below, a few more routines should also be replaced by machine-dependent code, for efficiency. But when a procedure is not part of the "inner loop," such changes aren't advisable; simplicity and robustness are preferable to trickery, unless the cost is too high.

```
double mp_double_make_fraction(MP mp, double p, double q)
{
  return ((p/q) * fraction_multiplier);
}
void mp_double_number_make_fraction(MP mp, mp_number * ret, mp_number p, mp_number q)
{
  ret→data.dval = mp_double_make_fraction(mp, p.data.dval, q.data.dval);
}
```

19. ⟨ Declarations 5 ⟩ +≡

```
double mp_double_make_fraction(MP mp, double p, double q);
```

20. The dual of *make_fraction* is *take_fraction*, which multiplies a given integer q by a fraction f. When the operands are positive, it computes $p = \lfloor qf/2^{28} + \frac{1}{2}\rfloor$, a symmetric function of q and f.

This routine is even more "inner loopy" than *make_fraction*; the present implementation consumes almost 20% of METAPOST's computation time during typical jobs, so a machine-language substitute is advisable.

```
double mp_double_take_fraction(MP mp, double p, double q)
{
  return ((p * q)/fraction_multiplier);
}
void mp_double_number_take_fraction(MP mp, mp_number * ret, mp_number p, mp_number q)
{
  ret→data.dval = mp_double_take_fraction(mp, p.data.dval, q.data.dval);
}
```

21. ⟨ Declarations 5 ⟩ +≡

```
double mp_double_take_fraction(MP mp, double p, double q);
```

22. When we want to multiply something by a *scaled* quantity, we use a scheme analogous to *take_fraction* but with a different scaling. Given positive operands, *take_scaled* computes the quantity $p = \lfloor qf/2^{16} + \frac{1}{2} \rfloor$.

Once again it is a good idea to use a machine-language replacement if possible; otherwise *take_scaled* will use more than 2% of the running time when the Computer Modern fonts are being generated.

```
void mp_double_number_take_scaled(MP mp, mp_number *ret, mp_number p_orig, mp_number q_orig)
{
  ret→data.dval = p_orig.data.dval * q_orig.data.dval;
}
```

23. For completeness, there's also *make_scaled*, which computes a quotient as a *scaled* number instead of as a *fraction*. In other words, the result is $\lfloor 2^{16}p/q + \frac{1}{2} \rfloor$, if the operands are positive. (This procedure is not used especially often, so it is not part of METAPOST's inner loop.)

```
double mp_double_make_scaled(MP mp, double p, double q)
{
  return p/q;
}
void mp_double_number_make_scaled(MP mp, mp_number *ret, mp_number p_orig, mp_number q_orig)
{
  ret→data.dval = p_orig.data.dval/q_orig.data.dval;
}
```

24. ⟨ Declarations 5 ⟩ +≡

```
double mp_double_make_scaled(MP mp, double p, double q);
```

25.

```
#define halfp(A) (integer)((unsigned)(A) ≫ 1)
```

26. Scanning numbers in the input.

The definitions below are temporarily here

```
#define set_cur_cmd(A) mp→cur_mod_→type = (A)
#define set_cur_mod(A) mp→cur_mod_→data.n.data.dval = (A)
```

⟨ Declarations 5 ⟩ +≡

```
static void mp_wrapup_numeric_token(MP mp, unsigned char *start, unsigned char *stop);
```

27.

```
void mp_wrapup_numeric_token(MP mp, unsigned char *start, unsigned char *stop)
{
  double result;
  char *end = (char *) stop;
  errno = 0;
  result = strtod((char *) start, &end);
  if (errno ≡ 0) {
    set_cur_mod(result);
    if (result ≥ warning_limit) {
      if (internal_value(mp_warning_check).data.dval > 0 ∧ (mp→scanner_status ≠ tex_flushing)) {
        char msg[256];
        const char *hlp[] = {"Continue␣and␣I'll␣try␣to␣cope",
            "with␣that␣big␣value;␣but␣it␣might␣be␣dangerous.",
            "(Set␣warningcheck:=0␣to␣suppress␣this␣message.)", Λ};
        mp_snprintf(msg, 256, "Number␣is␣too␣large␣(%g)", result);
        ;
        mp_error(mp, msg, hlp, true);
      }
    }
  }
  else if (mp→scanner_status ≠ tex_flushing) {
    const char *hlp[] = {"I␣could␣not␣handle␣this␣number␣specification",
        "probably␣because␣it␣is␣out␣of␣range.␣Error:", "", Λ};
    hlp[2] = strerror(errno);
    mp_error(mp, "Enormous␣number␣has␣been␣reduced.", hlp, false);
    ;
    set_cur_mod(EL_GORDO);
  }
  set_cur_cmd((mp_variable_type)mp_numeric_token);
}
```

28.
```
static void find_exponent(MP mp)
{
  if (mp->buffer[mp->cur_input.loc_field] ≡ 'e' ∨ mp->buffer[mp->cur_input.loc_field] ≡ 'E') {
    mp->cur_input.loc_field++;
    if (¬(mp->buffer[mp->cur_input.loc_field] ≡ '+' ∨ mp->buffer[mp->cur_input.loc_field] ≡
          '-' ∨ mp->char_class[mp->buffer[mp->cur_input.loc_field]] ≡ digit_class)) {
      mp->cur_input.loc_field--;
      return;
    }
    if (mp->buffer[mp->cur_input.loc_field] ≡ '+' ∨ mp->buffer[mp->cur_input.loc_field] ≡ '-') {
      mp->cur_input.loc_field++;
    }
    while (mp->char_class[mp->buffer[mp->cur_input.loc_field]] ≡ digit_class) {
      mp->cur_input.loc_field++;
    }
  }
}
void mp_double_scan_fractional_token(MP mp, int n)
{    /* n: scaled */
  unsigned char *start = &mp->buffer[mp->cur_input.loc_field − 1];
  unsigned char *stop;
  while (mp->char_class[mp->buffer[mp->cur_input.loc_field]] ≡ digit_class) {
    mp->cur_input.loc_field++;
  }
  find_exponent(mp);
  stop = &mp->buffer[mp->cur_input.loc_field − 1];
  mp_wrapup_numeric_token(mp, start, stop);
}
```

29. Input format is the same as for the C language, so we just collect valid bytes in the buffer, then call *strtod*( )

```
void mp_double_scan_numeric_token(MP mp, int n)
{    /* n: scaled */
  unsigned char *start = &mp->buffer[mp->cur_input.loc_field − 1];
  unsigned char *stop;
  while (mp->char_class[mp->buffer[mp->cur_input.loc_field]] ≡ digit_class) {
    mp->cur_input.loc_field++;
  }
  if (mp->buffer[mp->cur_input.loc_field] ≡ '.' ∧ mp->buffer[mp->cur_input.loc_field + 1] ≠ '.') {
    mp->cur_input.loc_field++;
    while (mp->char_class[mp->buffer[mp->cur_input.loc_field]] ≡ digit_class) {
      mp->cur_input.loc_field++;
    }
  }
  find_exponent(mp);
  stop = &mp->buffer[mp->cur_input.loc_field − 1];
  mp_wrapup_numeric_token(mp, start, stop);
}
```

30. The *scaled* quantities in METAPOST programs are generally supposed to be less than 2^{12} in absolute value, so METAPOST does much of its internal arithmetic with 28 significant bits of precision. A *fraction* denotes a scaled integer whose binary point is assumed to be 28 bit positions from the right.

#define *fraction_half* $(0.5 * \mathit{fraction\_multiplier})$
#define *fraction_one* $(1.0 * \mathit{fraction\_multiplier})$
#define *fraction_two* $(2.0 * \mathit{fraction\_multiplier})$
#define *fraction_three* $(3.0 * \mathit{fraction\_multiplier})$
#define *fraction_four* $(4.0 * \mathit{fraction\_multiplier})$

31. Here is a typical example of how the routines above can be used. It computes the function

$$\frac{1}{3\tau}f(\theta,\phi)=\frac{\tau^{-1}\bigl(2+\sqrt{2}\,(\sin\theta-\frac{1}{16}\sin\phi)(\sin\phi-\frac{1}{16}\sin\theta)(\cos\theta-\cos\phi)\bigr)}{3\,\bigl(1+\frac{1}{2}(\sqrt{5}-1)\cos\theta+\frac{1}{2}(3-\sqrt{5}\,)\cos\phi\bigr)},$$

where τ is a *scaled* "tension" parameter. This is METAPOST's magic fudge factor for placing the first control point of a curve that starts at an angle θ and ends at an angle ϕ from the straight path. (Actually, if the stated quantity exceeds 4, METAPOST reduces it to 4.)

The trigonometric quantity to be multiplied by $\sqrt{2}$ is less than $\sqrt{2}$. (It's a sum of eight terms whose absolute values can be bounded using relations such as $\sin\theta\cos\theta L\frac{1}{2}$.) Thus the numerator is positive; and since the tension τ is constrained to be at least $\frac{3}{4}$, the numerator is less than $\frac{16}{3}$. The denominator is nonnegative and at most 6.

The angles θ and ϕ are given implicitly in terms of *fraction* arguments *st*, *ct*, *sf*, and *cf*, representing $\sin\theta$, $\cos\theta$, $\sin\phi$, and $\cos\phi$, respectively.

```
void mp_double_velocity(MP mp, mp_number *ret, mp_number st, mp_number ct, mp_number sf,
        mp_number cf, mp_number t)
  {
    double acc, num, denom;     /* registers for intermediate calculations */
    acc = mp_double_take_fraction(mp, st.data.dval − (sf.data.dval/16.0),
        sf.data.dval − (st.data.dval/16.0));
    acc = mp_double_take_fraction(mp, acc, ct.data.dval − cf.data.dval);
    num = fraction_two + mp_double_take_fraction(mp, acc, sqrt(2) * fraction_one);
    denom = fraction_three + mp_double_take_fraction(mp, ct.data.dval,
        3 * fraction_half * (sqrt(5.0) − 1.0)) + mp_double_take_fraction(mp, cf.data.dval,
        3 * fraction_half * (3.0 − sqrt(5.0)));
    if (t.data.dval ≠ unity) num = mp_double_make_scaled(mp, num, t.data.dval);
    if (num/4 ≥ denom) {
      ret→data.dval = fraction_four;
    }
    else {
      ret→data.dval = mp_double_make_fraction(mp, num, denom);
    }
#if DEBUG
    fprintf(stdout, "\n%f␣=␣velocity(%f,%f,%f,%f,%f)", mp_number_to_double(*ret),
        mp_number_to_double(st), mp_number_to_double(ct), mp_number_to_double(sf),
        mp_number_to_double(cf), mp_number_to_double(t));
#endif
  }
```

32. The following somewhat different subroutine tests rigorously if ab is greater than, equal to, or less than cd, given integers (a, b, c, d). In most cases a quick decision is reached. The result is $+1$, 0, or -1 in the three respective cases.

```
void mp_ab_vs_cd(MP mp, mp_number *ret, mp_number a_orig, mp_number b_orig, mp_number c_orig,
        mp_number d_orig)
{
  integer q, r;    /* temporary registers */
  integer a, b, c, d;
  (void) mp;
  mp_double_ab_vs_cd(mp, ret, a_orig, b_orig, c_orig, d_orig);
  if (1 > 0) return;    /* TODO: remove this code until the end */
  a = a_orig.data.dval;
  b = b_orig.data.dval;
  c = c_orig.data.dval;
  d = d_orig.data.dval;
  ⟨Reduce to the case that a, c ≥ 0, b, d > 0 33⟩;
  while (1) {
    q = a/d;
    r = c/b;
    if (q ≠ r) {
      ret→data.dval = (q > r ? 1 : −1);
      goto RETURN;
    }
    q = a % d;
    r = c % b;
    if (r ≡ 0) {
      ret→data.dval = (q ? 1 : 0);
      goto RETURN;
    }
    if (q ≡ 0) {
      ret→data.dval = −1;
      goto RETURN;
    }
    a = b;
    b = q;
    c = d;
    d = r;
  }    /* now a > d > 0 and c > b > 0 */
RETURN:
#if DEBUG
  fprintf(stdout, "\n%f␣=␣ab_vs_cd(%f,%f,%f,%f)", mp_number_to_double(*ret),
      mp_number_to_double(a_orig), mp_number_to_double(b_orig), mp_number_to_double(c_orig),
      mp_number_to_double(d_orig));
#endif
  return;
}
```

33. $\langle$ Reduce to the case that $a, c \geq 0,\ b, d > 0$ 33 $\rangle \equiv$

```
if (a < 0) {
  a = −a;
  b = −b;
}
if (c < 0) {
  c = −c;
  d = −d;
}
if (d ≤ 0) {
  if (b ≥ 0) {
    if ((a ≡ 0 ∨ b ≡ 0) ∧ (c ≡ 0 ∨ d ≡ 0)) ret→data.dval = 0;
    else ret→data.dval = 1;
    goto RETURN;
  }
  if (d ≡ 0) {
    ret→data.dval = (a ≡ 0 ? 0 : −1);
    goto RETURN;
  }
  q = a;
  a = c;
  c = q;
  q = −b;
  b = −d;
  d = q;
}
else if (b ≤ 0) {
  if (b < 0 ∧ a > 0) {
    ret→data.dval = −1;
    return;
  }
  ret→data.dval = (c ≡ 0 ? 0 : −1);
  goto RETURN;
}
```

This code is used in section 32.

34. Now here's a subroutine that's handy for all sorts of path computations: Given a quadratic polynomial $B(a,b,c;t)$, the *crossing_point* function returns the unique *fraction* value t between 0 and 1 at which $B(a,b,c;t)$ changes from positive to negative, or returns $t = \mathit{fraction\_one} + 1$ if no such value exists. If $a < 0$ (so that $B(a,b,c;t)$ is already negative at $t = 0$), *crossing_point* returns the value zero.

The general bisection method is quite simple when $n = 2$, hence *crossing_point* does not take much time. At each stage in the recursion we have a subinterval defined by l and j such that $B(a,b,c;2^{-l}(j+t)) = B(x_0,x_1,x_2;t)$, and we want to "zero in" on the subinterval where $x_0 \ge 0$ and $\min(x_1,x_2) < 0$.

It is convenient for purposes of calculation to combine the values of l and j in a single variable $d = 2^l + j$, because the operation of bisection then corresponds simply to doubling d and possibly adding 1. Furthermore it proves to be convenient to modify our previous conventions for bisection slightly, maintaining the variables $X_0 = 2^l x_0$, $X_1 = 2^l(x_0 - x_1)$, and $X_2 = 2^l(x_1 - x_2)$. With these variables the conditions $x_0 \ge 0$ and $\min(x_1,x_2) < 0$ are equivalent to $\max(X_1, X_1 + X_2) > X_0 \ge 0$.

The following code maintains the invariant relations 0Ł$x0 < \max(x1, x1 + x2)$, $|x1| < 2^{30}$, $|x2| < 2^{30}$; it has been constructed in such a way that no arithmetic overflow will occur if the inputs satisfy $a < 2^{30}$, $|a - b| < 2^{30}$, and $|b - c| < 2^{30}$.

```
#define no_crossing
        {
          ret→data.dval = fraction_one + 1;
          goto RETURN;
        }
#define one_crossing
        {
          ret→data.dval = fraction_one;
          goto RETURN;
        }
#define zero_crossing
        {
          ret→data.dval = 0;
          goto RETURN;
        }

  static void mp_double_crossing_point(MP mp, mp_number *ret, mp_number aa, mp_number bb, mp_number cc)
  {
    double a, b, c;
    double d;    /* recursive counter */
    double x, xx, x0, x1, x2;    /* temporary registers for bisection */
    a = aa.data.dval;
    b = bb.data.dval;
    c = cc.data.dval;
    if (a < 0) zero_crossing;
    if (c ≥ 0) {
      if (b ≥ 0) {
        if (c > 0) {
          no_crossing;
        }
        else if ((a ≡ 0) ∧ (b ≡ 0)) {
          no_crossing;
        }
        else {
          one_crossing;
        }
      }
```

```
      if (a ≡ 0) zero_crossing;
    }
    else if (a ≡ 0) {
      if (b ≤ 0) zero_crossing;
    }    /* Use bisection to find the crossing point... */
    d = epsilon;
    x0 = a;
    x1 = a − b;
    x2 = b − c;
    do {    /* not sure why the error correction has to be ¿= 1E-12 */
      x = (x1 + x2)/2 + 1 · 10^−12;
      if (x1 − x0 > x0) {
        x2 = x;
        x0 += x0;
        d += d;
      }
      else {
        xx = x1 + x − x0;
        if (xx > x0) {
          x2 = x;
          x0 += x0;
          d += d;
        }
        else {
          x0 = x0 − xx;
          if (x ≤ x0) {
            if (x + x2 ≤ x0) no_crossing;
          }
          x1 = x;
          d = d + d + epsilon;
        }
      }
    } while (d < fraction_one);
    ret→data.dval = (d − fraction_one);
  RETURN:
#if DEBUG
    fprintf(stdout, "\n%f␣=␣crossing_point(%f,%f,%f)", mp_number_to_double(*ret),
        mp_number_to_double(aa), mp_number_to_double(bb), mp_number_to_double(cc));
#endif
    return;
  }
```

35. We conclude this set of elementary routines with some simple rounding and truncation operations.

36. *round_unscaled* rounds a *scaled* and converts it to **int**

```
  int mp_round_unscaled(mp_number x_orig)
  {
    int x = (int) ROUND(x_orig.data.dval);
    return x;
  }
```

37. *number_floor* floors a number

```
void mp_number_floor(mp_number *i)
{
  i->data.dval = floor(i->data.dval);
}
```

38. *fraction_to_scaled* rounds a *fraction* and converts it to *scaled*

```
void mp_double_fraction_to_round_scaled(mp_number *x_orig)
{
  double x = x_orig->data.dval;
  x_orig->type = mp_scaled_type;
  x_orig->data.dval = x/fraction_multiplier;
}
```

39. Algebraic and transcendental functions. METAPOST computes all of the necessary special functions from scratch, without relying on *real* arithmetic or system subroutines for sines, cosines, etc.

40.

```
void mp_double_square_rt(MP mp, mp_number *ret, mp_number x_orig)
{     /* return, x: scaled */
  double x;
  x = x_orig.data.dval;
  if (x ≤ 0) {
    ⟨Handle square root of zero or negative argument 41⟩;
  }
  else {
    ret→data.dval = sqrt(x);
  }
}
```

41. ⟨Handle square root of zero or negative argument 41⟩ ≡

```
{
  if (x < 0) {
    char msg[256];
    const char *hlp[] = {"Since I don't take square roots of negative numbers,",
        "I'm zeroing this one. Proceed, with fingers crossed.", Λ};
    char *xstr = mp_double_number_tostring(mp, x_orig);
    mp_snprintf(msg, 256, "Square root of %s has been replaced by 0", xstr);
    free(xstr);
    ;
    mp_error(mp, msg, hlp, true);
  }
  ret→data.dval = 0;
  return;
}
```

This code is used in section 40.

42. Pythagorean addition $\sqrt{a^2 + b^2}$ is implemented by a quick hack

```
void mp_double_pyth_add(MP mp, mp_number *ret, mp_number a_orig, mp_number b_orig)
{
  double a, b;     /* a,b : scaled */
  a = fabs(a_orig.data.dval);
  b = fabs(b_orig.data.dval);
  errno = 0;
  ret→data.dval = sqrt(a * a + b * b);
  if (errno) {
    mp→arith_error = true;
    ret→data.dval = EL_GORDO;
  }
}
```

43. Here is a similar algorithm for $\sqrt{a^2 - b^2}$. Same quick hack, also.

```
void mp_double_pyth_sub(MP mp, mp_number *ret, mp_number a_orig, mp_number b_orig)
{
  double a, b;
  a = fabs(a_orig.data.dval);
  b = fabs(b_orig.data.dval);
  if (a ≤ b) {
    ⟨Handle erroneous pyth_sub and set a: = 0 44⟩;
  }
  else {
    a = sqrt(a * a − b * b);
  }
  ret→data.dval = a;
}
```

44. ⟨Handle erroneous *pyth_sub* and set *a*: = 0 44⟩ ≡

```
{
  if (a < b) {
    char msg[256];
    const char *hlp[] = {"Since␣I␣don't␣take␣square␣roots␣of␣negative␣numbers,",
        "I'm␣zeroing␣this␣one.␣Proceed,␣with␣fingers␣crossed.", Λ};
    char *astr = mp_double_number_tostring(mp, a_orig);
    char *bstr = mp_double_number_tostring(mp, b_orig);
    mp_snprintf(msg, 256, "Pythagorean␣subtraction␣%s+-+%s␣has␣been␣replaced␣by␣0", astr, bstr);
    free(astr);
    free(bstr);
    ;
    mp_error(mp, msg, hlp, true);
  }
  a = 0;
}
```

This code is used in section 43.

45. The subroutines for logarithm and exponential involve two tables. The first is simple: *two_to_the*[k] equals 2^k.

#define *two_to_the*(A) (1 ≪ (**unsigned**)(A))

46. Here is the routine that calculates 2^8 times the natural logarithm of a *scaled* quantity; it is an integer approximation to $2^{24}\ln(x/2^{16})$, when x is a given positive integer.

```
void mp_double_m_log(MP mp, mp_number *ret, mp_number x_orig)
{
  if (x_orig.data.dval ≤ 0) {
    ⟨Handle non-positive logarithm 47⟩;
  }
  else {
    ret→data.dval = log(x_orig.data.dval) * 256.0;
  }
}
```

47. ⟨ Handle non-positive logarithm 47 ⟩ ≡

```
{
  char msg[256];
  const char *hlp[] = {"Since␣I␣don't␣take␣logs␣of␣non-positive␣numbers,",
      "I'm␣zeroing␣this␣one.␣Proceed,␣with␣fingers␣crossed.", Λ};
  char *xstr = mp_double_number_tostring(mp, x_orig);
  mp_snprintf(msg, 256, "Logarithm␣of␣%s␣has␣been␣replaced␣by␣0", xstr);
  free(xstr);
  ;
  mp_error(mp, msg, hlp, true);
  ret→data.dval = 0;
}
```

This code is used in section 46.

48. Conversely, the exponential routine calculates $\exp(x/2^8)$, when x is *scaled*.

```
void mp_double_m_exp(MP mp, mp_number * ret, mp_number x_orig)
{
  errno = 0;
  ret→data.dval = exp(x_orig.data.dval/256.0);
  if (errno) {
    if (x_orig.data.dval > 0) {
      mp→arith_error = true;
      ret→data.dval = EL_GORDO;
    }
    else {
      ret→data.dval = 0;
    }
  }
}
```

49. Given integers x and y, not both zero, the *n_arg* function returns the *angle* whose tangent points in the direction (x, y).

```
void mp_double_n_arg(MP mp, mp_number * ret, mp_number x_orig, mp_number y_orig)
{
  if (x_orig.data.dval ≡ 0.0 ∧ y_orig.data.dval ≡ 0.0) {
    ⟨ Handle undefined arg 50 ⟩;
  }
  else {
    ret→type = mp_angle_type;
    ret→data.dval = atan2(y_orig.data.dval, x_orig.data.dval) * (180.0/PI) * angle_multiplier;
    if (ret→data.dval ≡ −0.0) ret→data.dval = 0.0;
#if DEBUG
    fprintf(stdout, "\nn_arg(%g,%g,%g)", mp_number_to_double(*ret), mp_number_to_double(x_orig),
        mp_number_to_double(y_orig));
#endif
  }
}
```

50. ⟨ Handle undefined arg 50 ⟩ ≡

```
{
  const char *hlp[] = {"The␣‘angle’␣between␣two␣identical␣points␣is␣undefined.",
      "I’m␣zeroing␣this␣one.␣Proceed,␣with␣fingers␣crossed.", Λ};
  mp_error(mp, "angle(0,0)␣is␣taken␣as␣zero", hlp, true);
  ;
  ret→data.dval = 0;
}
```

This code is used in section 49.

51. Conversely, the *n_sin_cos* routine takes an *angle* and produces the sine and cosine of that angle. The results of this routine are stored in global integer variables *n_sin* and *n_cos*.

52. Given an integer z that is 2^{20} times an angle θ in degrees, the purpose of *n_sin_cos*(z) is to set $x = r\cos\theta$ and $y = r\sin\theta$ (approximately), for some rather large number r. The maximum of x and y will be between 2^{28} and 2^{30}, so that there will be hardly any loss of accuracy. Then x and y are divided by r.

```
#define one_eighty_deg (180.0 * angle_multiplier)
#define three_sixty_deg (360.0 * angle_multiplier)
#define odd(A) (abs(A) % 2 ≡ 1)
```

53. Compute a multiple of the sine and cosine

```
void mp_double_sin_cos(MP mp, mp_number z_orig, mp_number *n_cos, mp_number *n_sin)
{
  double rad;
  rad = (z_orig.data.dval/angle_multiplier);    /* still degrees */
  if ((rad ≡ 90.0) ∨ (rad ≡ −270)) {
    n_cos→data.dval = 0.0;
    n_sin→data.dval = fraction_multiplier;
  }
  else if ((rad ≡ −90.0) ∨ (rad ≡ 270.0)) {
    n_cos→data.dval = 0.0;
    n_sin→data.dval = −fraction_multiplier;
  }
  else if ((rad ≡ 180.0) ∨ (rad ≡ −180.0)) {
    n_cos→data.dval = −fraction_multiplier;
    n_sin→data.dval = 0.0;
  }
  else {
    rad = rad * PI/180.0;
    n_cos→data.dval = cos(rad) * fraction_multiplier;
    n_sin→data.dval = sin(rad) * fraction_multiplier;
  }
#if DEBUG
  fprintf(stdout, "\nsin_cos(%f,%f,%f)", mp_number_to_double(z_orig), mp_number_to_double(*n_cos),
      mp_number_to_double(*n_sin));
#endif
}
```

54. This is the http://www-cs-faculty.stanford.edu/ uno/programs/rng.c with small cosmetic modifications.

```
#define KK 100 /* the long lag */
#define LL 37 /* the short lag */
#define MM (1L << 30) /* the modulus */
#define mod_diff(x, y) (((x) - (y)) & (MM - 1)) /* subtraction mod MM */ /* */
  static long ran_x[KK]; /* the generator state */ /* */
  static void ran_array(long aa[], int n) /* put n new random numbers in aa */
    /* long aa[] destination */ /* int n array length (must be at least KK) */
  {
    register int i, j;
    for (j = 0; j < KK; j++) aa[j] = ran_x[j];
    for ( ; j < n; j++) aa[j] = mod_diff(aa[j - KK], aa[j - LL]);
    for (i = 0; i < LL; i++, j++) ran_x[i] = mod_diff(aa[j - KK], aa[j - LL]);
    for ( ; i < KK; i++, j++) ran_x[i] = mod_diff(aa[j - KK], ran_x[i - LL]);
  } /* */ /* the following routines are from exercise 3.6–15 */
      /* after calling ran_start, get new randoms by, e.g., x = ran_arr_next() */ /* */
#define QUALITY 1009 /* recommended quality level for high-res use */
  static long ran_arr_buf[QUALITY];
  static long ran_arr_dummy = -1, ran_arr_started = -1;
  static long *ran_arr_ptr = &ran_arr_dummy; /* the next random number, or -1 */ /* */
#define TT 70 /* guaranteed separation between streams */
#define is_odd(x) ((x) & 1) /* units bit of x */ /* */
  static void ran_start(long seed) /* do this before using ran_array */
    /* long seed selector for different streams */
  {
    register int t, j;
    long x[KK + KK - 1]; /* the preparation buffer */
    register long ss = (seed + 2) & (MM - 2);
    for (j = 0; j < KK; j++) {
      x[j] = ss; /* bootstrap the buffer */
      ss <<= 1;
      if (ss ≥ MM) ss -= MM - 2; /* cyclic shift 29 bits */
    }
    x[1]++; /* make x[1] (and only x[1]) odd */
    for (ss = seed & (MM - 1), t = TT - 1; t; ) {
      for (j = KK - 1; j > 0; j--) x[j + j] = x[j], x[j + j - 1] = 0; /* "square" */
      for (j = KK + KK - 2; j ≥ KK; j--)
        x[j - (KK - LL)] = mod_diff(x[j - (KK - LL)], x[j]), x[j - KK] = mod_diff(x[j - KK], x[j]);
      if (is_odd(ss)) { /* "multiply by z" */
        for (j = KK; j > 0; j--) x[j] = x[j - 1];
        x[0] = x[KK]; /* shift the buffer cyclically */
        x[LL] = mod_diff(x[LL], x[KK]);
      }
      if (ss) ss >>= 1;
      else t--;
    }
    for (j = 0; j < LL; j++) ran_x[j + KK - LL] = x[j];
    for ( ; j < KK; j++) ran_x[j - LL] = x[j];
    for (j = 0; j < 10; j++) ran_array(x, KK + KK - 1); /* warm things up */
    ran_arr_ptr = &ran_arr_started;
```

```
  }    /* */
#define ran_arr_next() (*ran_arr_ptr ≥ 0 ? *ran_arr_ptr++ : ran_arr_cycle())
  static long ran_arr_cycle(void)
  {
    if (ran_arr_ptr ≡ &ran_arr_dummy) ran_start(314159L);    /* the user forgot to initialize */
    ran_array(ran_arr_buf, QUALITY);
    ran_arr_buf[KK] = −1;
    ran_arr_ptr = ran_arr_buf + 1;
    return ran_arr_buf[0];
  }
```

55. To initialize the *randoms* table, we call the following routine.

```
  void mp_init_randoms(MP mp, int seed)
  {
    int j, jj, k;    /* more or less random integers */
    int i;    /* index into randoms */
    j = abs(seed);
    while (j ≥ fraction_one) {
      j = j/2;
    }
    k = 1;
    for (i = 0; i ≤ 54; i++) {
      jj = k;
      k = j − k;
      j = jj;
      if (k < 0) k += fraction_one;
      mp→randoms[(i * 21) % 55].data.dval = j;
    }
    mp_new_randoms(mp);
    mp_new_randoms(mp);
    mp_new_randoms(mp);    /* "warm up" the array */
    ran_start((unsigned long) seed);
  }
```

56. **static double** *modulus*(**double** *left*, **double** *right*);

```
  double modulus(double left, double right)
  {
    double quota = left/right;
    double frac, tmp;
    frac = modf(quota, &tmp);    /* frac contains what's beyond the '.' */
    frac *= right;
    return frac;
  }
  void mp_number_modulo(mp_number * a, mp_number b)
  {
    a→data.dval = modulus(a→data.dval, b.data.dval);
  }
```

57. To consume a random integer for the uniform generator, the program below will say '*next_unif_random*'.

```
static void mp_next_unif_random(MP mp, mp_number *ret)
{
  double a;
  unsigned long int op;
  (void) mp;
  op = (unsigned) ran_arr_next();
  a = op/(MM * 1.0);
  ret→data.dval = a;
}
```

58. To consume a random fraction, the program below will say '*next_random*'.

```
static void mp_next_random(MP mp, mp_number *ret)
{
  if (mp→j_random ≡ 0) mp_new_randoms(mp);
  else mp→j_random = mp→j_random − 1;
  mp_number_clone(ret, mp→randoms[mp→j_random]);
}
```

59. To produce a uniform random number in the range $0 \le u < x$ or $0 \ge u > x$ or $0 = u = x$, given a *scaled* value x, we proceed as shown here.

Note that the call of *take_fraction* will produce the values 0 and x with about half the probability that it will produce any other particular values between 0 and x, because it rounds its answers.

```
static void mp_double_m_unif_rand(MP mp, mp_number *ret, mp_number x_orig){ mp_number y;
      /* trial value */
    mp_number x, abs_x;
    mp_number u;
    new_fraction(y);
    new_number(x);
    new_number(abs_x);
    new_number(u);
    mp_number_clone(&x, x_orig);
    mp_number_clone(&abs_x, x);
    mp_double_abs(&abs_x);
    mp_next_unif_random(mp, &u);
    y.data.dval = abs_x.data.dval * u.data.dval;
    free_number(u); if (mp_number_equal(y, abs_x)) { mp_number_clone (ret, ( ( math_data * ) mp→math
        ) → zero_t ) ; } else if ( mp_number_greater (x, ( ( math_data * ) mp→math ) → zero_t ) )
    {
      mp_number_clone(ret, y);
    }
    else {
      mp_number_clone(ret, y);
      mp_number_negate(ret);
    }
    free_number(abs_x);
    free_number(x);
    free_number(y); }
```

60. Finally, a normal deviate with mean zero and unit standard deviation can readily be obtained with the ratio method (Algorithm 3.4.1R in *The Art of Computer Programming*).

```
static void mp_double_m_norm_rand(MP mp, mp_number *ret){ mp_number ab_vs_cd;
    mp_number abs_x;
    mp_number u;
    mp_number r;
    mp_number la, xa;
    new_number(ab_vs_cd);
    new_number(la);
    new_number(xa);
    new_number(abs_x);
    new_number(u);
    new_number(r); do { do { mp_number v;
    new_number(v);
    mp_next_random(mp, &v); mp_number_substract (&v, ( ( math_data * ) mp→math ) → fraction_half_t
        ) ; mp_double_number_take_fraction (mp, &xa, ( ( math_data * ) mp→math ) → sqrt_8_e_k, v ) ;
    free_number(v);
    mp_next_random(mp, &u);
    mp_number_clone(&abs_x, xa);
    mp_double_abs(&abs_x); }
    while (¬mp_number_less(abs_x, u)) ;
    mp_double_number_make_fraction(mp, &r, xa, u);
    mp_number_clone(&xa, r);
    mp_double_m_log(mp, &la, u); mp_set_double_from_substraction (&la, ( ( math_data * ) mp→math
        ) → twelve_ln_2_k, la ) ; mp_double_ab_vs_cd (mp, &ab_vs_cd, ( ( math_data * ) mp→math ) →
        one_k, la, xa, xa ) ; } while ( mp_number_less (ab_vs_cd, ( ( math_data * ) mp→math ) → zero_t )
        ) ;
    mp_number_clone(ret, xa);
    free_number(ab_vs_cd);
    free_number(r);
    free_number(abs_x);
    free_number(la);
    free_number(xa);
    free_number(u); }
```

61. The following subroutine is used only in *norm_rand* and tests if *ab* is greater than, equal to, or less than *cd*. The result is $+1$, 0, or -1 in the three respective cases.

```
void mp_double_ab_vs_cd(MP mp, mp_number *ret, mp_number a_orig, mp_number b_orig,
        mp_number c_orig, mp_number d_orig)
{
  double ab, cd;
  (void) mp;
  ret→data.dval = 0;
  ab = a_orig.data.dval * b_orig.data.dval;
  cd = c_orig.data.dval * d_orig.data.dval;
  if (ab > cd) ret→data.dval = 1;
  else if (ab < cd) ret→data.dval = −1;
  return;
}
```

Math support functions for IEEE double based math