1. Introduction.

```
#include <w2c/config.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>
#include "mpmath.h"   /* internal header */
  ⟨ Preprocessor definitions ⟩
```

2. ⟨ Declarations 5 ⟩;

3. ⟨ mpmath.h 3 ⟩ ≡

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

4. Math initialization.

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

⟨ Declarations 5 ⟩ ≡

```
static void mp_scan_fractional_token(MP mp, int n);
static void mp_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_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_print_number(MP mp, mp_number n);
static char *mp_number_tostring(MP mp, mp_number n);
static void mp_slow_add(MP mp, mp_number *ret, mp_number x_orig, mp_number y_orig);
static void mp_square_rt(MP mp, mp_number *ret, mp_number x_orig);
static void mp_n_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_m_unif_rand(MP mp, mp_number *ret, mp_number x_orig);
static void mp_m_norm_rand(MP mp, mp_number *ret);
static void mp_m_exp(MP mp, mp_number *ret, mp_number x_orig);
static void mp_m_log(MP mp, mp_number *ret, mp_number x_orig);
static void mp_pyth_sub(MP mp, mp_number *r, mp_number a, mp_number b);
static void mp_n_arg(MP mp, mp_number *ret, mp_number x, mp_number y);
static void mp_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_number_from_int(mp_number *A, int B);
static void mp_set_number_from_boolean(mp_number *A, int B);
static void mp_set_number_from_scaled(mp_number *A, int B);
static void mp_set_number_from_boolean(mp_number *A, int B);
static void mp_set_number_from_addition(mp_number *A, mp_number B, mp_number C);
static void mp_set_number_from_substraction(mp_number *A, mp_number B, mp_number C);
static void mp_set_number_from_div(mp_number *A, mp_number B, mp_number C);
static void mp_set_number_from_mul(mp_number *A, mp_number B, mp_number C);
static void mp_set_number_from_int_div(mp_number *A, mp_number B, int C);
static void mp_set_number_from_int_mul(mp_number *A, mp_number B, int C);
static void mp_set_number_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_number_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_scaled(mp_number A);
static int mp_number_to_boolean(mp_number A);
```

static int *mp_number_to_int*(*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_fraction_to_round_scaled*(*mp_number* $*x$);
static void *mp_number_make_scaled*(`MP` *mp*, *mp_number* $*r$, *mp_number p*, *mp_number q*);
static void *mp_number_make_fraction*(`MP` *mp*, *mp_number* $*r$, *mp_number p*, *mp_number q*);
static void *mp_number_take_fraction*(`MP` *mp*, *mp_number* $*r$, *mp_number p*, *mp_number q*);
static void *mp_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_free_scaled_math*(`MP` *mp*);
static void *mp_scaled_set_precision*(`MP` *mp*);

See also sections 15, 22, 26, 28, 50, and 56.

This code is used in section 2.

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

⟨ Internal library declarations 6 ⟩ ≡
void $*$*mp_initialize_scaled_math*(`MP` *mp*);
void *mp_set_number_from_double*(*mp_number* $*A$, **double** *B*);
void *mp_pyth_add*(`MP` *mp*, *mp_number* $*r$, *mp_number a*, *mp_number b*);
double *mp_number_to_double*(*mp_number A*);

See also sections 20 and 24.

This code is used in section 3.

7.

```
#define coef_bound °4525252525   /* fraction approximation to 7/3 */
#define fraction_threshold 2685   /* a fraction coefficient less than this is zeroed */
#define half_fraction_threshold 1342   /* half of fraction_threshold */
#define scaled_threshold 8   /* a scaled coefficient less than this is zeroed */
#define half_scaled_threshold 4   /* half of scaled_threshold */
#define near_zero_angle 26844
#define p_over_v_threshold #80000
#define equation_threshold 64
#define tfm_warn_threshold 4096

  void *mp_initialize_scaled_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.val = unity * 10;
    mp_new_number(mp, &math→precision_max, mp_scaled_type);
    math→precision_max.data.val = unity * 10;
    mp_new_number(mp, &math→precision_min, mp_scaled_type);
    math→precision_min.data.val = unity * 10;   /* here are the constants for scaled objects */
    mp_new_number(mp, &math→epsilon_t, mp_scaled_type);
    math→epsilon_t.data.val = 1;
    mp_new_number(mp, &math→inf_t, mp_scaled_type);
    math→inf_t.data.val = EL_GORDO;
    mp_new_number(mp, &math→warning_limit_t, mp_scaled_type);
    math→warning_limit_t.data.val = fraction_one;
    mp_new_number(mp, &math→one_third_inf_t, mp_scaled_type);
    math→one_third_inf_t.data.val = one_third_EL_GORDO;
    mp_new_number(mp, &math→unity_t, mp_scaled_type);
    math→unity_t.data.val = unity;
    mp_new_number(mp, &math→two_t, mp_scaled_type);
    math→two_t.data.val = two;
    mp_new_number(mp, &math→three_t, mp_scaled_type);
    math→three_t.data.val = three;
    mp_new_number(mp, &math→half_unit_t, mp_scaled_type);
    math→half_unit_t.data.val = half_unit;
    mp_new_number(mp, &math→three_quarter_unit_t, mp_scaled_type);
    math→three_quarter_unit_t.data.val = 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.val = (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.val = fraction_one;
    mp_new_number(mp, &math→fraction_half_t, mp_fraction_type);
    math→fraction_half_t.data.val = fraction_half;
    mp_new_number(mp, &math→fraction_three_t, mp_fraction_type);
    math→fraction_three_t.data.val = fraction_three;
    mp_new_number(mp, &math→fraction_four_t, mp_fraction_type);
    math→fraction_four_t.data.val = fraction_four;   /* angles */
    mp_new_number(mp, &math→three_sixty_deg_t, mp_angle_type);
    math→three_sixty_deg_t.data.val = three_sixty_deg;
```

```
mp_new_number(mp, &math→one_eighty_deg_t, mp_angle_type);
math→one_eighty_deg_t.data.val = one_eighty_deg;   /* various approximations */
mp_new_number(mp, &math→one_k, mp_scaled_type);
math→one_k.data.val = 1024;
mp_new_number(mp, &math→sqrt_8_e_k, mp_scaled_type);
math→sqrt_8_e_k.data.val = 112429;   /* 2^16 √(8/e) ≈ 112428.82793 */
mp_new_number(mp, &math→twelve_ln_2_k, mp_fraction_type);
math→twelve_ln_2_k.data.val = 139548960;   /* 2^24 · 12 ln 2 ≈ 139548959.6165 */
mp_new_number(mp, &math→coef_bound_k, mp_fraction_type);
math→coef_bound_k.data.val = coef_bound;
mp_new_number(mp, &math→coef_bound_minus_1, mp_fraction_type);
math→coef_bound_minus_1.data.val = coef_bound − 1;
mp_new_number(mp, &math→twelvebits_3, mp_scaled_type);
math→twelvebits_3.data.val = 1365;   /* 1365 ≈ 2^12/3 */
mp_new_number(mp, &math→twentysixbits_sqrt2_t, mp_fraction_type);
math→twentysixbits_sqrt2_t.data.val = 94906266;   /* 2^26 √2 ≈ 94906265.62 */
mp_new_number(mp, &math→twentyeightbits_d_t, mp_fraction_type);
math→twentyeightbits_d_t.data.val = 35596755;   /* 2^28 d ≈ 35596754.69 */
mp_new_number(mp, &math→twentysevenbits_sqrt2_d_t, mp_fraction_type);
math→twentysevenbits_sqrt2_d_t.data.val = 25170707;   /* 2^27 √2 d ≈ 25170706.63 */
  /* thresholds */
mp_new_number(mp, &math→fraction_threshold_t, mp_fraction_type);
math→fraction_threshold_t.data.val = fraction_threshold;
mp_new_number(mp, &math→half_fraction_threshold_t, mp_fraction_type);
math→half_fraction_threshold_t.data.val = half_fraction_threshold;
mp_new_number(mp, &math→scaled_threshold_t, mp_scaled_type);
math→scaled_threshold_t.data.val = scaled_threshold;
mp_new_number(mp, &math→half_scaled_threshold_t, mp_scaled_type);
math→half_scaled_threshold_t.data.val = half_scaled_threshold;
mp_new_number(mp, &math→near_zero_angle_t, mp_angle_type);
math→near_zero_angle_t.data.val = near_zero_angle;
mp_new_number(mp, &math→p_over_v_threshold_t, mp_fraction_type);
math→p_over_v_threshold_t.data.val = p_over_v_threshold;
mp_new_number(mp, &math→equation_threshold_t, mp_scaled_type);
math→equation_threshold_t.data.val = equation_threshold;
mp_new_number(mp, &math→tfm_warn_threshold_t, mp_scaled_type);
math→tfm_warn_threshold_t.data.val = tfm_warn_threshold;   /* functions */
math→from_int = mp_set_number_from_int;
math→from_boolean = mp_set_number_from_boolean;
math→from_scaled = mp_set_number_from_scaled;
math→from_double = mp_set_number_from_double;
math→from_addition = mp_set_number_from_addition;
math→from_substraction = mp_set_number_from_substraction;
math→from_oftheway = mp_set_number_from_of_the_way;
math→from_div = mp_set_number_from_div;
math→from_mul = mp_set_number_from_mul;
math→from_int_div = mp_set_number_from_int_div;
math→from_int_mul = mp_set_number_from_int_mul;
math→negate = mp_number_negate;
math→add = mp_number_add;
math→substract = mp_number_substract;
math→half = mp_number_half;
```

```
math->halfp = mp_number_halfp;
math->do_double = mp_number_double;
math->abs = mp_number_abs;
math->clone = mp_number_clone;
math->swap = mp_number_swap;
math->add_scaled = mp_number_add_scaled;
math->multiply_int = mp_number_multiply_int;
math->divide_int = mp_number_divide_int;
math->to_int = mp_number_to_int;
math->to_boolean = mp_number_to_boolean;
math->to_scaled = mp_number_to_scaled;
math->to_double = mp_number_to_double;
math->odd = mp_number_odd;
math->equal = mp_number_equal;
math->less = mp_number_less;
math->greater = mp_number_greater;
math->nonequalabs = mp_number_nonequalabs;
math->round_unscaled = mp_round_unscaled;
math->floor_scaled = mp_number_floor;
math->fraction_to_round_scaled = mp_fraction_to_round_scaled;
math->make_scaled = mp_number_make_scaled;
math->make_fraction = mp_number_make_fraction;
math->take_fraction = mp_number_take_fraction;
math->take_scaled = mp_number_take_scaled;
math->velocity = mp_velocity;
math->n_arg = mp_n_arg;
math->m_log = mp_m_log;
math->m_exp = mp_m_exp;
math->m_unif_rand = mp_m_unif_rand;
math->m_norm_rand = mp_m_norm_rand;
math->pyth_add = mp_pyth_add;
math->pyth_sub = mp_pyth_sub;
math->fraction_to_scaled = mp_number_fraction_to_scaled;
math->scaled_to_fraction = mp_number_scaled_to_fraction;
math->scaled_to_angle = mp_number_scaled_to_angle;
math->angle_to_scaled = mp_number_angle_to_scaled;
math->init_randoms = mp_init_randoms;
math->sin_cos = mp_n_sin_cos;
math->slow_add = mp_slow_add;
math->sqrt = mp_square_rt;
math->print = mp_print_number;
math->tostring = mp_number_tostring;
math->modulo = mp_number_modulo;
math->ab_vs_cd = mp_ab_vs_cd;
math->crossing_point = mp_crossing_point;
math->scan_numeric = mp_scan_numeric_token;
math->scan_fractional = mp_scan_fractional_token;
math->free_math = mp_free_scaled_math;
math->set_precision = mp_scaled_set_precision;
return (void *) math; } void mp_scaled_set_precision(MP mp)
{ } void mp_free_scaled_math(MP mp){ free_number ( ( ( math_data * ) mp->math ) -> epsilon_t ) ;
    free_number ( ( ( math_data * ) mp->math ) -> inf_t ) ; free_number ( ( ( math_data * )
```

```
        mp->math ) -> arc_tol_k ) ; free_number ( ( ( math_data * ) mp->math ) -> three_sixty_deg_t
        ) ; free_number ( ( ( math_data * ) mp->math ) -> one_eighty_deg_t ) ; free_number ( ( (
        math_data * ) mp->math ) -> fraction_one_t ) ; free_number ( ( ( math_data * ) mp->math )
        -> fraction_half_t ) ; free_number ( ( ( math_data * ) mp->math ) -> fraction_three_t ) ;
        free_number ( ( ( math_data * ) mp->math ) -> fraction_four_t ) ; free_number ( ( ( math_data
        * ) mp->math ) -> zero_t ) ; free_number ( ( ( math_data * ) mp->math ) -> half_unit_t ) ;
        free_number ( ( ( math_data * ) mp->math ) -> three_quarter_unit_t ) ; free_number ( ( (
        math_data * ) mp->math ) -> unity_t ) ; free_number ( ( ( math_data * ) mp->math ) -> two_t )
        ; free_number ( ( ( math_data * ) mp->math ) -> three_t ) ; free_number ( ( ( math_data * )
        mp->math ) -> one_third_inf_t ) ; free_number ( ( ( math_data * ) mp->math ) -> warning_limit_t
        ) ; free_number ( ( ( math_data * ) mp->math ) -> one_k ) ; free_number ( ( ( math_data * )
        mp->math ) -> sqrt_8_e_k ) ; free_number ( ( ( math_data * ) mp->math ) -> twelve_ln_2_k ) ;
        free_number ( ( ( math_data * ) mp->math ) -> coef_bound_k ) ; free_number ( ( ( math_data
        * ) mp->math ) -> coef_bound_minus_1 ) ; free_number ( ( ( math_data * ) mp->math )
        -> twelvebits_3 ) ; free_number ( ( ( math_data * ) mp->math ) -> twentysixbits_sqrt2_t ) ;
        free_number ( ( ( math_data * ) mp->math ) -> twentyeightbits_d_t ) ; free_number ( ( (
        math_data * ) mp->math ) -> twentysevenbits_sqrt2_d_t ) ; free_number ( ( ( math_data
        * ) mp->math ) -> fraction_threshold_t ) ; free_number ( ( ( math_data * ) mp->math ) ->
        half_fraction_threshold_t ) ; free_number ( ( ( math_data * ) mp->math ) -> scaled_threshold_t
        ) ; free_number ( ( ( math_data * ) mp->math ) -> half_scaled_threshold_t ) ; free_number ( ( (
        math_data * ) mp->math ) -> near_zero_angle_t ) ; free_number ( ( ( math_data * ) mp->math )
        -> p_over_v_threshold_t ) ; free_number ( ( ( math_data * ) mp->math ) -> equation_threshold_t
        ) ; free_number ( ( ( math_data * ) mp->math ) -> tfm_warn_threshold_t ) ;
    free(mp->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->data.val = 0;
  n->type = t;
}
```

10.

```
void mp_free_number(MP mp, mp_number *n)
{
  (void) mp;
  n->type = mp_nan_type;
}
```

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

```
void mp_set_number_from_int(mp_number *A, int B)
{
  A→data.val = B;
}
void mp_set_number_from_boolean(mp_number *A, int B)
{
  A→data.val = B;
}
void mp_set_number_from_scaled(mp_number *A, int B)
{
  A→data.val = B;
}
void mp_set_number_from_double(mp_number *A, double B)
{
  A→data.val = (int)(B * 65536.0);
}
void mp_set_number_from_addition(mp_number *A, mp_number B, mp_number C)
{
  A→data.val = B.data.val + C.data.val;
}
void mp_set_number_from_substraction(mp_number *A, mp_number B, mp_number C)
{
  A→data.val = B.data.val − C.data.val;
}
void mp_set_number_from_div(mp_number *A, mp_number B, mp_number C)
{
  A→data.val = B.data.val/C.data.val;
}
void mp_set_number_from_mul(mp_number *A, mp_number B, mp_number C)
{
  A→data.val = B.data.val * C.data.val;
}
void mp_set_number_from_int_div(mp_number *A, mp_number B, int C)
{
  A→data.val = B.data.val/C;
}
void mp_set_number_from_int_mul(mp_number *A, mp_number B, int C)
{
  A→data.val = B.data.val * C;
}
void mp_set_number_from_of_the_way(MP mp, mp_number *A, mp_number t, mp_number B, mp_number C)
{
  A→data.val = B.data.val − mp_take_fraction(mp, (B.data.val − C.data.val), t.data.val);
}
void mp_number_negate(mp_number *A)
{
  A→data.val = −A→data.val;
}
```

```
void mp_number_add(mp_number *A, mp_number B)
{
  A→data.val = A→data.val + B.data.val;
}
void mp_number_substract(mp_number *A, mp_number B)
{
  A→data.val = A→data.val − B.data.val;
}
void mp_number_half(mp_number *A)
{
  A→data.val = A→data.val/2;
}
void mp_number_halfp(mp_number *A)
{
  A→data.val = (A→data.val ≫ 1);
}
void mp_number_double(mp_number *A)
{
  A→data.val = A→data.val + A→data.val;
}
void mp_number_add_scaled(mp_number *A, int B)
{     /* also for negative B */
  A→data.val = A→data.val + B;
}
void mp_number_multiply_int(mp_number *A, int B)
{
  A→data.val = B * A→data.val;
}
void mp_number_divide_int(mp_number *A, int B)
{
  A→data.val = A→data.val/B;
}
void mp_number_abs(mp_number *A)
{
  A→data.val = abs(A→data.val);
}
void mp_number_clone(mp_number *A, mp_number B)
{
  A→data.val = B.data.val;
}
void mp_number_swap(mp_number *A, mp_number *B)
{
  int swap_tmp = A→data.val;
  A→data.val = B→data.val;
  B→data.val = swap_tmp;
}
void mp_number_fraction_to_scaled(mp_number *A)
{
  A→type = mp_scaled_type;
```

```
    A→data.val = A→data.val/4096;
  }
  void mp_number_angle_to_scaled(mp_number * A)
  {
    A→type = mp_scaled_type;
    if (A→data.val ≥ 0) {
      A→data.val = (A→data.val + 8)/16;
    }
    else {
      A→data.val = −((−A→data.val + 8)/16);
    }
  }
  void mp_number_scaled_to_fraction(mp_number * A)
  {
    A→type = mp_fraction_type;
    A→data.val = A→data.val * 4096;
  }
  void mp_number_scaled_to_angle(mp_number * A)
  {
    A→type = mp_angle_type;
    A→data.val = A→data.val * 16;
  }
```

12. Query functions

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

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 #10000   /* 2^16, represents 1.00000 */
#define two (2 * unity)   /* 2^17, represents 2.00000 */
#define three (3 * unity)   /* 2^17 + 2^16, represents 3.00000 */
#define half_unit (unity/2)   /* 2^15, represents 0.50000 */
#define three_quarter_unit (3 * (unity/4))   /* 3 · 2^14, represents 0.75000 */
#define EL_GORDO #7fffffff   /* 2^31 − 1, the largest value that METAPOST likes */
#define one_third_EL_GORDO °5252525252
```

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.

#define *halfp*(A) (*integer*)((**unsigned**)$(A) \gg 1$)

15. Here is a procedure analogous to *print_int*. If the output of this procedure is subsequently read by METAPOST and converted by the *round_decimals* routine above, it turns out that the original value will be reproduced exactly. A decimal point is printed only if the value is not an integer. If there is more than one way to print the result with the optimum number of digits following the decimal point, the closest possible value is given.

The invariant relation in the **repeat** loop is that a sequence of decimal digits yet to be printed will yield the original number if and only if they form a fraction f in the range $s - \delta L10 \cdot 2^{16} f < s$. We can stop if and only if $f = 0$ satisfies this condition; the loop will terminate before s can possibly become zero.

⟨ Declarations 5 ⟩ +≡
static void *mp_print_scaled*(`MP` *mp*, **int** *s*); /* scaled */
static char **mp_string_scaled*(`MP` *mp*, **int** *s*);

16.

```
static void mp_print_scaled(MP mp, int s)
{   /* s=scaled prints scaled real, rounded to five digits */
  int delta;   /* amount of allowable inaccuracy, scaled */
  if (s < 0) {
    mp_print_char(mp, xord('-'));
    s = -s;   /* print the sign, if negative */
  }
  mp_print_int(mp, s/unity);   /* print the integer part */
  s = 10 * (s % unity) + 5;
  if (s ≠ 5) {
    delta = 10;
    mp_print_char(mp, xord('.'));
    do {
      if (delta > unity) s = s + °100000 - (delta/2);   /* round the final digit */
      mp_print_char(mp, xord('0' + (s/unity)));
      s = 10 * (s % unity);
      delta = delta * 10;
    } while (s > delta);
  }
}
static char *mp_string_scaled(MP mp, int s)
{   /* s=scaled prints scaled real, rounded to five digits */
  static char scaled_string[32];
  int delta;   /* amount of allowable inaccuracy, scaled */
  int i = 0;
  if (s < 0) {
    scaled_string[i++] = xord('-');
    s = -s;   /* print the sign, if negative */
  }   /* print the integer part */
  mp_snprintf((scaled_string + i), 12, "%d", (int)(s/unity));
  while (*(scaled_string + i)) i++;
  s = 10 * (s % unity) + 5;
  if (s ≠ 5) {
    delta = 10;
    scaled_string[i++] = xord('.');
    do {
      if (delta > unity) s = s + °100000 - (delta/2);   /* round the final digit */
      scaled_string[i++] = xord('0' + (s/unity));
      s = 10 * (s % unity);
      delta = delta * 10;
    } while (s > delta);
  }
  scaled_string[i] = '\0';
  return scaled_string;
}
```

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_slow_add(MP mp, mp_number *ret, mp_number x_orig, mp_number y_orig)
{
  integer x, y;
  x = x_orig.data.val;
  y = y_orig.data.val;
  if (x ≥ 0) {
    if (y ≤ EL_GORDO − x) {
      ret→data.val = x + y;
    }
    else {
      mp→arith_error = true;
      ret→data.val = EL_GORDO;
    }
  }
  else if (−y ≤ EL_GORDO + x) {
    ret→data.val = x + y;
  }
  else {
    mp→arith_error = true;
    ret→data.val = −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.

19. We need these preprocessor values

```
#define TWEXP31 2147483648.0
#define TWEXP28 268435456.0
#define TWEXP16 65536.0
#define TWEXP_16 (1.0/65536.0)
#define TWEXP_28 (1.0/268435456.0)
  static integer mp_make_fraction(MP mp, integer p, integer q)
  {
    integer i;
    if (q ≡ 0) mp_confusion(mp, "/");
    {
      register double d;
      d = TWEXP28 ∗ (double) p/(double) q;
      if ((p ⊕ q) ≥ 0) {
        d += 0.5;
        if (d ≥ TWEXP31) {
          mp→arith_error = true;
          i = EL_GORDO;
          goto RETURN;
        }
        i = (integer) d;
        if (d ≡ (double) i ∧ (((q > 0 ? −q : q) & °77777) ∗ (((i & °37777) ≪ 1) − 1) & °4000) ≠ 0) −−i;
      }
      else {
        d −= 0.5;
        if (d ≤ −TWEXP31) {
          mp→arith_error = true;
          i = −EL_GORDO;
          goto RETURN;
        }
        i = (integer) d;
        if (d ≡ (double) i ∧ (((q > 0 ? q : −q) & °77777) ∗ (((i & °37777) ≪ 1) + 1) & °4000) ≠ 0) ++i;
      }
    }
  RETURN: return i;
  }
  void mp_number_make_fraction(MP mp, mp_number ∗ ret, mp_number p, mp_number q)
  {
    ret→data.val = mp_make_fraction(mp, p.data.val, q.data.val);
  }
```

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.

```
⟨ Internal library declarations 6 ⟩ +≡   /∗ still in use by tfmin.w ∗/
  integer mp_take_fraction(MP mp, integer q, int f);
```

21.

```
integer mp_take_fraction(MP mp, integer p, int q)
{    /* q = fraction */
  register double d;
  register integer i;
  d = (double) p * (double) q * TWEXP_28;
  if ((p ⊕ q) ≥ 0) {
    d += 0.5;
    if (d ≥ TWEXP31) {
      if (d ≠ TWEXP31 ∨ (((p & °77777) * (q & °77777)) & °40000) ≡ 0) mp→arith_error = true;
      return EL_GORDO;
    }
    i = (integer) d;
    if (d ≡ (double) i ∧ (((p & °77777) * (q & °77777)) & °40000) ≠ 0) −−i;
  }
  else {
    d −= 0.5;
    if (d ≤ −TWEXP31) {
      if (d ≠ −TWEXP31 ∨ ((−(p & °77777) * (q & °77777)) & °40000) ≡ 0) mp→arith_error = true;
      return −EL_GORDO;
    }
    i = (integer) d;
    if (d ≡ (double) i ∧ ((−(p & °77777) * (q & °77777)) & °40000) ≠ 0) ++i;
  }
  return i;
}
void mp_number_take_fraction(MP mp, mp_number * ret, mp_number p_orig, mp_number q_orig)
{
  ret→data.val = mp_take_fraction(mp, p_orig.data.val, q_orig.data.val);
}
```

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.

```
⟨ Declarations 5 ⟩ +≡
  static integer mp_take_scaled(MP mp, integer q, int f);
```

23. **static** *integer mp_take_scaled*(MP *mp*, *integer p*, **int** *q*)

```
{    /* q = scaled */
  register double d;
  register integer i;
  d = (double) p * (double) q * TWEXP_16;
  if ((p ⊕ q) ≥ 0) {
    d += 0.5;
    if (d ≥ TWEXP31) {
      if (d ≠ TWEXP31 ∨ (((p & °77777) * (q & °77777)) & °40000) ≡ 0) mp→arith_error = true;
      return EL_GORDO;
    }
    i = (integer)d;
    if (d ≡ (double) i ∧ (((p & °77777) * (q & °77777)) & °40000) ≠ 0) −−i;
  }
  else {
    d −= 0.5;
    if (d ≤ −TWEXP31) {
      if (d ≠ −TWEXP31 ∨ ((−(p & °77777) * (q & °77777)) & °40000) ≡ 0) mp→arith_error = true;
      return −EL_GORDO;
    }
    i = (integer)d;
    if (d ≡ (double) i ∧ ((−(p & °77777) * (q & °77777)) & °40000) ≠ 0) ++i;
  }
  return i;
}
void mp_number_take_scaled(MP mp, mp_number * ret, mp_number p_orig, mp_number q_orig)
{
  ret→data.val = mp_take_scaled(mp, p_orig.data.val, q_orig.data.val);
}
```

24. 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.)

⟨ Internal library declarations 6 ⟩ +≡ /* still in use by svgout.w */

```
int mp_make_scaled(MP mp, integer p, integer q);
```

25. **int** *mp_make_scaled*(MP *mp*, *integer p*, *integer q*)

```
{   /* return scaled */
  register integer i;
  if (q ≡ 0) mp_confusion(mp, "/");
  {
    register double d;
    d = TWEXP16 ∗ (double) p/(double) q;
    if ((p ⊕ q) ≥ 0) {
      d += 0.5;
      if (d ≥ TWEXP31) {
        mp→arith_error = true;
        return EL_GORDO;
      }
      i = (integer)d;
      if (d ≡ (double) i ∧ (((q > 0 ? −q : q) & °77777) ∗ (((i & °37777) ≪ 1) − 1) & °4000) ≠ 0) −−i;
    }
    else {
      d −= 0.5;
      if (d ≤ −TWEXP31) {
        mp→arith_error = true;
        return −EL_GORDO;
      }
      i = (integer)d;
      if (d ≡ (double) i ∧ (((q > 0 ? q : −q) & °77777) ∗ (((i & °37777) ≪ 1) + 1) & °4000) ≠ 0) ++i;
    }
  }
  return i;
}
void mp_number_make_scaled(MP mp, mp_number ∗ ret, mp_number p_orig, mp_number q_orig)
{
  ret→data.val = mp_make_scaled(mp, p_orig.data.val, q_orig.data.val);
}
```

26. The following function is used to create a scaled integer from a given decimal fraction $(.d_0 d_1 \ldots d_{k-1})$, where $0 \le k \le 17$.

⟨ Declarations 5 ⟩ +≡
  static int *mp_round_decimals*(MP *mp*, **unsigned char** ∗*b*, *quarterword k*);

27. **static int** *mp_round_decimals*(MP *mp*, **unsigned char** ∗*b*, *quarterword k*)

```
{   /* return: scaled */   /* converts a decimal fraction */
  unsigned a = 0;   /* the accumulator */
  int l = 0;
  (void) mp;   /* Will be needed later */
  for (l = k − 1; l ≥ 0; l−−) {
    if (l < 16)   /* digits for k ≥ 17 cannot affect the result */
      a = (a + (unsigned)(∗(b + l) − '0') ∗ two)/10;
  }
  return (int) halfp(a + 1);
}
```

28. 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*.*val* = (*A*)

⟨ Declarations 5 ⟩ +≡
static void *mp_wrapup_numeric_token*(MP *mp*, **int** *n*, **int** *f*);

29. **static void** *mp_wrapup_numeric_token*(MP *mp*, **int** *n*, **int** *f*)

```
{  /* n,f: scaled */
  int mod;  /* scaled */
  if (n < 32768) {
    mod = (n * unity + f);
    set_cur_mod(mod);
    if (mod ≥ fraction_one) {
      if (internal_value(mp_warning_check).data.val > 0 ∧ (mp→scanner_status ≠ tex_flushing)) {
        char msg[256];
        const char *hlp[] = {"It␣is␣at␣least␣4096.␣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␣(%s)", mp_string_scaled(mp, mod));
        ;
        mp_error(mp, msg, hlp, true);
      }
    }
  }
  else if (mp→scanner_status ≠ tex_flushing) {
    const char *hlp[] = {"I␣can\'t␣handle␣numbers␣bigger␣than␣32767.99998;",
        "so␣I've␣changed␣your␣constant␣to␣that␣maximum␣amount.", Λ};
    mp_error(mp, "Enormous␣number␣has␣been␣reduced", hlp, false);
    ;
    set_cur_mod(EL_GORDO);
  }
  set_cur_cmd((mp_variable_type) mp_numeric_token);
}
```

30. **void** *mp_scan_fractional_token*(MP *mp*, **int** *n*)

```
{  /* n: scaled */
  int f;  /* scaled */
  int k = 0;
  do {
    k++;
    mp→cur_input.loc_field++;
  } while (mp→char_class[mp→buffer[mp→cur_input.loc_field]] ≡ digit_class);
  f = mp_round_decimals(mp, (unsigned char *)(mp→buffer + mp→cur_input.loc_field − k),
      (quarterword) k);
  if (f ≡ unity) {
    n++;
    f = 0;
  }
  mp_wrapup_numeric_token(mp, n, f);
}
```

31. **void** *mp_scan_numeric_token*(MP *mp*, **int** *n*)
```
{   /* n: scaled */
  while (mp→char_class[mp→buffer[mp→cur_input.loc_field]] ≡ digit_class) {
    if (n < 32768) n = 10 * n + mp→buffer[mp→cur_input.loc_field] − '0';
    mp→cur_input.loc_field ++;
  }
  if (¬(mp→buffer[mp→cur_input.loc_field] ≡ '.' ∧ mp→char_class[mp→buffer[mp→cur_input.loc_field + 1]] ≡
        digit_class)) {
    mp_wrapup_numeric_token(mp, n, 0);
  }
  else {
    mp→cur_input.loc_field ++;
    mp_scan_fractional_token(mp, n);
  }
}
```

32. 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 °1000000000   /* 2^27, represents 0.50000000 */
#define fraction_one °2000000000   /* 2^28, represents 1.00000000 */
#define fraction_two °4000000000   /* 2^29, represents 2.00000000 */
#define fraction_three °6000000000   /* 3 · 2^28, represents 3.00000000 */
#define fraction_four °10000000000   /* 2^30, represents 4.00000000 */
```

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

$$\frac{1}{3\tau}f(\theta,\phi)=\frac{\tau^{-1}\bigl(2+\sqrt{2}\,(\sin\theta-\frac{1}{16}\sin\phi)(\sin\phi-\frac{1}{16}\sin\theta)(\cos\theta-\cos\phi)\bigr)}{3\,\bigl(1+\frac12(\sqrt{5}-1)\cos\theta+\frac12(3-\sqrt{5}\,)\cos\phi\bigr)},$$

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 $\sin\theta\cos\theta L\frac12$.) Thus the numerator is positive; and since the tension τ is constrained to be at least $\frac34$, the numerator is less than $\frac{16}{3}$. The denominator is nonnegative and at most 6. Hence the fixed-point calculations below are guaranteed to stay within the bounds of a 32-bit computer word.

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_velocity(MP mp, mp_number *ret, mp_number st, mp_number ct, mp_number sf, mp_number cf,
        mp_number t)
{
  integer acc, num, denom;   /* registers for intermediate calculations */
  acc = mp_take_fraction(mp, st.data.val − (sf.data.val/16), sf.data.val − (st.data.val/16));
  acc = mp_take_fraction(mp, acc, ct.data.val − cf.data.val);
  num = fraction_two + mp_take_fraction(mp, acc, 379625062);   /* 2^28 √2 ≈ 379625062.497 */
  denom = fraction_three + mp_take_fraction(mp, ct.data.val, 497706707) + mp_take_fraction(mp,
      cf.data.val, 307599661);
    /* 3 · 2^27 · (√5 − 1) ≈ 497706706.78 and 3 · 2^27 · (3 − √5) ≈ 307599661.22 */
  if (t.data.val ≠ unity) num = mp_make_scaled(mp, num, t.data.val);
      /* make_scaled(fraction, scaled) = fraction */
  if (num/4 ≥ denom) {
    ret→data.val = fraction_four;
  }
  else {
    ret→data.val = mp_make_fraction(mp, num, denom);
  }
      /* printf("num,denom=%f,%f␣−=>␣%f\n", num/65536.0, denom/65536.0, ret.data.val/65536.0); */
}
```

34. 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.

```
static 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;
  a = a_orig.data.val;
  b = b_orig.data.val;
  c = c_orig.data.val;
  d = d_orig.data.val;
  ⟨Reduce to the case that a, c ≥ 0, b, d > 0 35⟩;
  while (1) {
    q = a/d;
    r = c/b;
    if (q ≠ r) {
      ret→data.val = (q > r ? 1 : −1);
      return;
    }
    q = a % d;
    r = c % b;
    if (r ≡ 0) {
      ret→data.val = (q ? 1 : 0);
      return;
    }
    if (q ≡ 0) {
      ret→data.val = −1;
      return;
    }
    a = b;
    b = q;
    c = d;
    d = r;
  }    /* now a > d > 0 and c > b > 0 */
}
```

35. ⟨ Reduce to the case that $a, c \ge 0$, $b, d > 0$ 35 ⟩ ≡

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

This code is used in section 34.

36. 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 $x_0 \geq 0$ 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 $x_0 \geq 0$ and $\min(x_1,x_2) < 0$ are equivalent to $\max(X_1, X_1 + X_2) > X_0 \geq 0$.

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.val = fraction_one + 1;
      return;
    }
#define one_crossing
    {
      ret→data.val = fraction_one;
      return;
    }
#define zero_crossing
    {
      ret→data.val = 0;
      return;
    }

  static void mp_crossing_point(MP mp, mp_number *ret, mp_number aa, mp_number bb, mp_number cc)
  {
    integer a, b, c;
    integer d;    /* recursive counter */
    integer x, xx, x0, x1, x2;    /* temporary registers for bisection */
    a = aa.data.val;
    b = bb.data.val;
    c = cc.data.val;
    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 = 1;
  x0 = a;
  x1 = a − b;
  x2 = b − c;
  do {
    x = (x1 + x2)/2;
    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 + 1;
      }
    }
  } while (d < fraction_one);
  ret→data.val = (d − fraction_one);
}
```

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

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

```
int mp_round_unscaled(mp_number x_orig)
{
  int x = x_orig.data.val;
  if (x ≥ 32768) {
    return 1 + ((x − 32768)/65536);
  }
  else if (x ≥ −32768) {
    return 0;
  }
  else {
    return −(1 + ((−(x + 1) − 32768)/65536));
  }
}
```

39. *number_floor* floors a *scaled*

void $mp\_number\_floor(mp\_number * i)$
{
$i \rightarrow data.val = i \rightarrow data.val \mathbin{\&} -65536;$
}

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

void $mp\_fraction\_to\_round\_scaled(mp\_number * x\_orig)$
{
int $x = x\_orig \rightarrow data.val;$
$x\_orig \rightarrow type = mp\_scaled\_type;$
$x\_orig \rightarrow data.val = (x \geq 2048 \mathbin{?} 1 + ((x - 2048)/4096) : (x \geq -2048 \mathbin{?} 0 : -(1 + ((-(x + 1) - 2048)/4096))));$
}

41. 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.

42. To get the square root of a *scaled* number x, we want to calculate $s = \lfloor 2^8\sqrt{x} + \frac{1}{2}\rfloor$. If $x > 0$, this is the unique integer such that $2^{16}x - s \text{L} s^2 < 2^{16}x + s$. The following subroutine determines s by an iterative method that maintains the invariant relations $x = 2^{46-2k}x_0 \bmod 2^{30}$, $0 < y = \lfloor 2^{16-2k}x_0\rfloor - s^2 + s \text{L} q = 2s$, where x_0 is the initial value of x. The value of y might, however, be zero at the start of the first iteration.

```
void mp_square_rt(MP mp, mp_number *ret, mp_number x_orig)
{   /* return, x: scaled */
  integer x;
  quarterword k;   /* iteration control counter */
  integer y;   /* register for intermediate calculations */
  integer q;   /* register for intermediate calculations */
  x = x_orig.data.val;
  if (x ≤ 0) {
    ⟨Handle square root of zero or negative argument 43⟩;
  }
  else {
    k = 23;
    q = 2;
    while (x < fraction_two) {   /* i.e., while x < */
      k−−;
      x = x + x + x + x;
    }
    if (x < fraction_four) y = 0;
    else {
      x = x − fraction_four;
      y = 1;
    }
    do {
      ⟨Decrease k by 1, maintaining the invariant relations between x, y, and q 44⟩;
    } while (k ≠ 0);
    ret→data.val = (int)(halfp(q));
  }
}
```

43. ⟨Handle square root of zero or negative argument 43⟩ ≡

```
{
  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};
    mp_snprintf(msg, 256, "Square root of %s has been replaced by 0", mp_string_scaled(mp, x));
    ;
    mp_error(mp, msg, hlp, true);
  }
  ret→data.val = 0;
  return;
}
```

This code is used in section 42.

44. ⟨Decrease k by 1, maintaining the invariant relations between x, y, and q 44⟩ ≡

```
x += x;
y += y;
if (x ≥ fraction_four) {      /* note that fraction_four = 2^30 */
   x = x − fraction_four;
   y++;
}
;
x += x;
y = y + y − q;
q += q;
if (x ≥ fraction_four) {
   x = x − fraction_four;
   y++;
}
;
if (y > (int) q) {
   y −= q;
   q += 2;
}
else if (y ≤ 0) {
   q −= 2;
   y += q;
}
; k−−
```

This code is used in section 42.

45. Pythagorean addition $\sqrt{a^2+b^2}$ is implemented by an elegant iterative scheme due to Cleve Moler and Donald Morrison [*IBM Journal of Research and Development* **27** (1983), 577–581]. It modifies a and b in such a way that their Pythagorean sum remains invariant, while the smaller argument decreases.

```
void mp_pyth_add(MP mp, mp_number *ret, mp_number a_orig, mp_number b_orig)
{
  int a, b;    /* a,b : scaled */
  int r;    /* register used to transform a and b, fraction */
  boolean big;    /* is the result dangerously near 2^31? */
  a = abs(a_orig.data.val);
  b = abs(b_orig.data.val);
  if (a < b) {
    r = b;
    b = a;
    a = r;
  }
  ;    /* now 0 ≤ b ≤ a */
  if (b > 0) {
    if (a < fraction_two) {
      big = false;
    }
    else {
      a = a/4;
      b = b/4;
      big = true;
    }
    ;    /* we reduced the precision to avoid arithmetic overflow */
    ⟨Replace a by an approximation to √(a^2 + b^2) 46⟩;
    if (big) {
      if (a < fraction_two) {
        a = a + a + a + a;
      }
      else {
        mp→arith_error = true;
        a = EL_GORDO;
      }
      ;
    }
  }
  ret→data.val = a;
}
```

46. The key idea here is to reflect the vector (a, b) about the line through $(a, b/2)$.

⟨ Replace a by an approximation to $\sqrt{a^2 + b^2}$ 46 ⟩ ≡

```
while (1) {
   r = mp_make_fraction(mp, b, a);
   r = mp_take_fraction(mp, r, r);   /* now r ≈ b^2/a^2 */
   if (r ≡ 0) break;
   r = mp_make_fraction(mp, r, fraction_four + r);
   a = a + mp_take_fraction(mp, a + a, r);
   b = mp_take_fraction(mp, b, r);
}
```

This code is used in section 45.

47. Here is a similar algorithm for $\sqrt{a^2 - b^2}$. It converges slowly when b is near a, but otherwise it works fine.

```
void mp_pyth_sub(MP mp, mp_number *ret, mp_number a_orig, mp_number b_orig)
{
   int a, b;   /* a,b: scaled */
   int r;   /* register used to transform a and b, fraction */
   boolean big;   /* is the result dangerously near 2^31? */
   a = abs(a_orig.data.val);
   b = abs(b_orig.data.val);
   if (a ≤ b) {
      ⟨ Handle erroneous pyth_sub and set a: = 0 49 ⟩;
   }
   else {
      if (a < fraction_four) {
         big = false;
      }
      else {
         a = (integer) halfp(a);
         b = (integer) halfp(b);
         big = true;
      }
      ⟨ Replace a by an approximation to √(a^2 − b^2) 48 ⟩;
      if (big) a *= 2;
   }
   ret→data.val = a;
}
```

48. ⟨ Replace a by an approximation to $\sqrt{a^2 - b^2}$ 48 ⟩ ≡

```
while (1) {
   r = mp_make_fraction(mp, b, a);
   r = mp_take_fraction(mp, r, r);   /* now r ≈ b^2/a^2 */
   if (r ≡ 0) break;
   r = mp_make_fraction(mp, r, fraction_four − r);
   a = a − mp_take_fraction(mp, a + a, r);
   b = mp_take_fraction(mp, b, r);
}
```

This code is used in section 47.

49. ⟨ Handle erroneous *pyth_sub* and set a: $= 0$ 49 ⟩ ≡

```
{
  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.", NULL};
    char *astr = strdup(mp_string_scaled(mp, a));
    assert(astr);
    mp_snprintf(msg, 256, "Pythagorean subtraction %s+-+%s has been replaced by 0", astr,
        mp_string_scaled(mp, b));
    free(astr);
    ;
    mp_error(mp, msg, hlp, true);
  }
  a = 0;
}
```

This code is used in section 47.

50. The subroutines for logarithm and exponential involve two tables. The first is simple: *two_to_the*[k] equals 2^k. The second involves a bit more calculation, which the author claims to have done correctly: *spec_log*[k] is 2^{27} times $\ln\bigl(1/(1-2^{-k})\bigr) = 2^{-k} + \frac{1}{2}2^{-2k} + \frac{1}{3}2^{-3k} + \cdots$, rounded to the nearest integer.

```
#define two_to_the(A) (1 << (unsigned)(A))
```

⟨ Declarations 5 ⟩ +≡

```
static const integer spec_log[29] = {0,    /* special logarithms */
93032640, 38612034, 17922280, 8662214, 4261238, 2113709, 1052693, 525315, 262400, 131136, 65552, 32772,
    16385, 8192, 4096, 2048, 1024, 512, 256, 128, 64, 32, 16, 8, 4, 2, 1, 1};
```

51. 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.

The method is based on exercise 1.2.2–25 in *The Art of Computer Programming*: During the main iteration we have 1Ł$2^{-30}x < 1/(1-2^{1-k})$, and the logarithm of $2^{30}x$ remains to be added to an accumulator register called y. Three auxiliary bits of accuracy are retained in y during the calculation, and sixteen auxiliary bits to extend y are kept in z during the initial argument reduction. (We add $100 \cdot 2^{16} = 6553600$ to z and subtract 100 from y so that z will not become negative; also, the actual amount subtracted from y is 96, not 100, because we want to add 4 for rounding before the final division by 8.)

```
void mp_m_log(MP mp, mp_number *ret, mp_number x_orig)
{      /* return, x: scaled */
  int x;
  integer y, z;      /* auxiliary registers */
  integer k;      /* iteration counter */
  x = x_orig.data.val;
  if (x ≤ 0) {
    ⟨ Handle non-positive logarithm 53 ⟩;
  }
  else {
    y = 1302456956 + 4 − 100;      /* 14 × 2^27 ln 2 ≈ 1302456956.421063 */
    z = 27595 + 6553600;      /* and 2^16 × .421063 ≈ 27595 */
    while (x < fraction_four) {
      x = 2 * x;
      y −= 93032639;
      z −= 48782;
    }      /* 2^27 ln 2 ≈ 93032639.74436163 and 2^16 × .74436163 ≈ 48782 */
    y = y + (z/unity);
    k = 2;
    while (x > fraction_four + 4) {
      ⟨ Increase k until x can be multiplied by a factor of 2^−k, and adjust y accordingly 52 ⟩;
    }
    ret→data.val = (y/8);
  }
}
```

52. ⟨ Increase k until x can be multiplied by a factor of 2^{-k}, and adjust y accordingly 52 ⟩ ≡

```
{
  z = ((x − 1)/two_to_the(k)) + 1;      /* z = ⌈x/2^k⌉ */
  while (x < fraction_four + z) {
    z = halfp(z + 1);
    k++;
  }
  ;
  y += spec_log[k];
  x −= z;
}
```

This code is used in section 51.

53. ⟨ Handle non-positive logarithm 53 ⟩ ≡

```
{
  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.", NULL};
  mp_snprintf(msg, 256, "Logarithm of %s has been replaced by 0", mp_string_scaled(mp, x));
  ;
  mp_error(mp, msg, hlp, true);
  ret->data.val = 0;
}
```

This code is used in section 51.

54. Conversely, the exponential routine calculates $\exp(x/2^8)$, when x is *scaled*. The result is an integer approximation to $2^{16}\exp(x/2^{24})$, when x is regarded as an integer.

```
void mp_m_exp(MP mp, mp_number *ret, mp_number x_orig)
{
  quarterword k;   /* loop control index */
  integer y, z;   /* auxiliary registers */
  int x;
  x = x_orig.data.val;
  if (x > 174436200) {   /* 2^24 ln((2^31 − 1)/2^16) ≈ 174436199.51 */
    mp->arith_error = true;
    ret->data.val = EL_GORDO;
  }
  else if (x < −197694359) {   /* 2^24 ln(2^−1/2^16) ≈ −197694359.45 */
    ret->data.val = 0;
  }
  else {
    if (x ≤ 0) {
      z = −8 * x;
      y = °4000000;   /* y = 2^20 */
    }
    else {
      if (x ≤ 127919879) {
        z = 1023359037 − 8 * x;   /* 2^27 ln((2^31 − 1)/2^20) ≈ 1023359037.125 */
      }
      else {
        z = 8 * (174436200 − x);   /* z is always nonnegative */
      }
      y = EL_GORDO;
    }
    ⟨ Multiply y by exp(−z/2^27) 55 ⟩;
    if (x ≤ 127919879) ret->data.val = ((y + 8)/16);
    else ret->data.val = y;
  }
}
```

55. The idea here is that subtracting *spec_log* $[k]$ from z corresponds to multiplying y by $1 - 2^{-k}$.

A subtle point (which had to be checked) was that if $x = 127919879$, the value of y will decrease so that $y + 8$ doesn't overflow. In fact, z will be 5 in this case, and y will decrease by 64 when $k = 25$ and by 16 when $k = 27$.

$\langle$ Multiply y by $\exp(-z/2^{27})$ 55 $\rangle \equiv$

```
k = 1;
while (z > 0) {
  while (z ≥ spec_log[k]) {
    z −= spec_log[k];
    y = y − 1 − ((y − two_to_the(k − 1))/two_to_the(k));
  }
  k++;
}
```

This code is used in section 54.

56. The trigonometric subroutines use an auxiliary table such that *spec_atan* $[k]$ contains an approximation to the *angle* whose tangent is $1/2^k$. $\arctan 2^{-k}$ times $2^{20} \cdot 180/\pi$

$\langle$ Declarations 5 $\rangle +\equiv$

```
static const int spec_atan[27] = {0, 27855475, 14718068, 7471121, 3750058, 1876857, 938658, 469357,
    234682, 117342, 58671, 29335, 14668, 7334, 3667, 1833, 917, 458, 229, 115, 57, 29, 14, 7, 4, 2, 1};
```

57. Given integers x and y, not both zero, the *n_arg* function returns the *angle* whose tangent points in the direction (x, y). This subroutine first determines the correct octant, then solves the problem for $0 \le y \le x$, then converts the result appropriately to return an answer in the range $-one\_eighty\_deg \le \theta \le one\_eighty\_deg$. (The answer is $+one\_eighty\_deg$ if $y = 0$ and $x < 0$, but an answer of $-one\_eighty\_deg$ is possible if, for example, $y = -1$ and $x = -2^{30}$.)

The octants are represented in a "Gray code," since that turns out to be computationally simplest.

```
#define negate_x 1
#define negate_y 2
#define switch_x_and_y 4
#define first_octant 1
#define second_octant (first_octant + switch_x_and_y)
#define third_octant (first_octant + switch_x_and_y + negate_x)
#define fourth_octant (first_octant + negate_x)
#define fifth_octant (first_octant + negate_x + negate_y)
#define sixth_octant (first_octant + switch_x_and_y + negate_x + negate_y)
#define seventh_octant (first_octant + switch_x_and_y + negate_y)
#define eighth_octant (first_octant + negate_y)

  void mp_n_arg(MP mp, mp_number *ret, mp_number x_orig, mp_number y_orig)
  {
    integer z;   /* auxiliary register */
    integer t;   /* temporary storage */
    quarterword k;   /* loop counter */
    int octant;   /* octant code */
    integer x, y;
    x = x_orig.data.val;
    y = y_orig.data.val;
    if (x ≥ 0) {
      octant = first_octant;
    }
    else {
      x = −x;
      octant = first_octant + negate_x;
    }
    if (y < 0) {
      y = −y;
      octant = octant + negate_y;
    }
    if (x < y) {
      t = y;
      y = x;
      x = t;
      octant = octant + switch_x_and_y;
    }
    if (x ≡ 0) {
      ⟨Handle undefined arg 58⟩;
    }
    else {
      ret→type = mp_angle_type;
      ⟨Set variable z to the arg of (x, y) 60⟩;
      ⟨Return an appropriate answer based on z and octant 59⟩;
    }
```

```
}
```

58. ⟨ Handle undefined arg 58 ⟩ ≡

```
{
  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.val = 0;
}
```

This code is used in section 57.

59. ⟨ Return an appropriate answer based on z and *octant* 59 ⟩ ≡

```
switch (octant) {
case first_octant: ret→data.val = z;
  break;
case second_octant: ret→data.val = (ninety_deg − z);
  break;
case third_octant: ret→data.val = (ninety_deg + z);
  break;
case fourth_octant: ret→data.val = (one_eighty_deg − z);
  break;
case fifth_octant: ret→data.val = (z − one_eighty_deg);
  break;
case sixth_octant: ret→data.val = (−z − ninety_deg);
  break;
case seventh_octant: ret→data.val = (z − ninety_deg);
  break;
case eighth_octant: ret→data.val = (−z);
  break;
}   /* there are no other cases */
```

This code is used in section 57.

60. At this point we have $x \ge y \ge 0$, and $x > 0$. The numbers are scaled up or down until $2^{28}Łx < 2^{29}$, so that accurate fixed-point calculations will be made.

⟨ Set variable z to the arg of (x, y) 60 ⟩ ≡

```
while (x ≥ fraction_two) {
  x = halfp(x);
  y = halfp(y);
}
z = 0;
if (y > 0) {
  while (x < fraction_one) {
    x += x;
    y += y;
  }
  ;
  ⟨ Increase z to the arg of (x, y) 61 ⟩;
}
```

This code is used in section 57.

61. During the calculations of this section, variables x and y represent actual coordinates $(x, 2^{-k}y)$. We will maintain the condition $x \geq y$, so that the tangent will be at most 2^{-k}. If $x < 2y$, the tangent is greater than 2^{-k-1}. The transformation $(a,b) \mapsto (a + b\tan\phi, b - a\tan\phi)$ replaces (a,b) by coordinates whose angle has decreased by ϕ; in the special case $a = x$, $b = 2^{-k}y$, and $\tan\phi = 2^{-k-1}$, this operation reduces to the particularly simple iteration shown here. [Cf. John E. Meggitt, *IBM Journal of Research and Development* **6** (1962), 210–226.]

The initial value of x will be multiplied by at most $(1 + \frac{1}{2})(1 + \frac{1}{8})(1 + \frac{1}{32}) \cdots \approx 1.7584$; hence there is no chance of integer overflow.

⟨ Increase z to the arg of (x, y) 61 ⟩ ≡

```
k = 0;
do {
  y += y;
  k++;
  if (y > x) {
    z = z + spec_atan[k];
    t = x;
    x = x + (y/two_to_the(k + k));
    y = y − t;
  }
  ;
} while (k ≠ 15); do
{
  y += y;
  k++;
  if (y > x) {
    z = z + spec_atan[k];
    y = y − x;
  }
  ;
}
while (k ≠ 26)
```

This code is used in section 60.

62. 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*.

63. 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 forty_five_deg °264000000      /* 45 · 2^20, represents 45° */
#define ninety_deg °550000000      /* 90 · 2^20, represents 90° */
#define one_eighty_deg °1320000000      /* 180 · 2^20, represents 180° */
#define three_sixty_deg °2640000000      /* 360 · 2^20, represents 360° */
#define odd(A) (abs(A) % 2 ≡ 1)
```

64. Compute a multiple of the sine and cosine

```
void mp_n_sin_cos(MP mp, mp_number z_orig, mp_number *n_cos, mp_number *n_sin)
{
  quarterword k;    /* loop control variable */
  int q;    /* specifies the quadrant */
  integer x, y, t;    /* temporary registers */
  int z;    /* scaled */
  mp_number x_n, y_n, ret;
  new_number(ret);
  new_number(x_n);
  new_number(y_n);
  z = z_orig.data.val;
  while (z < 0) z = z + three_sixty_deg;
  z = z % three_sixty_deg;    /* now 0 ≤ z < three_sixty_deg */
  q = z / forty_five_deg;
  z = z % forty_five_deg;
  x = fraction_one;
  y = x;
  if (¬odd(q)) z = forty_five_deg − z;
  ⟨Subtract angle z from (x, y) 66⟩;
  ⟨Convert (x, y) to the octant determined by q 65⟩;
  x_n.data.val = x;
  y_n.data.val = y;
  mp_pyth_add(mp, &ret, x_n, y_n);
  n_cos→data.val = mp_make_fraction(mp, x, ret.data.val);
  n_sin→data.val = mp_make_fraction(mp, y, ret.data.val);
  free_number(ret);
  free_number(x_n);
  free_number(y_n);
}
```

65. In this case the octants are numbered sequentially.

⟨ Convert (x, y) to the octant determined by q 65 ⟩ ≡

```
switch (q) {
case 0: break;
case 1: t = x;
  x = y;
  y = t;
  break;
case 2: t = x;
  x = −y;
  y = t;
  break;
case 3: x = −x;
  break;
case 4: x = −x;
  y = −y;
  break;
case 5: t = x;
  x = −y;
  y = −t;
  break;
case 6: t = x;
  x = y;
  y = −t;
  break;
case 7: y = −y;
  break;
}    /* there are no other cases */
```

This code is used in section 64.

66. The main iteration of *n_sin_cos* is similar to that of *n_arg* but applied in reverse. The values of *spec_atan*[*k*] decrease slowly enough that this loop is guaranteed to terminate before the (nonexistent) value *spec_atan*[27] would be required.

⟨ Subtract angle z from (x, y) 66 ⟩ ≡

```
k = 1;
while (z > 0) {
  if (z ≥ spec_atan[k]) {
    z = z − spec_atan[k];
    t = x;
    x = t + y/two_to_the(k);
    y = y − t/two_to_the(k);
  }
  k++;
}
if (y < 0) y = 0    /* this precaution may never be needed */
```

This code is used in section 64.

67. 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.val = j;
  }
  mp_new_randoms(mp);
  mp_new_randoms(mp);
  mp_new_randoms(mp);    /* "warm up" the array */
}
```

68.
```
void mp_print_number(MP mp, mp_number n)
{
  mp_print_scaled(mp, n.data.val);
}
```

69.
```
char *mp_number_tostring(MP mp, mp_number n)
{
  return mp_string_scaled(mp, n.data.val);
}
```

70.
```
void mp_number_modulo(mp_number *a, mp_number b)
{
  a→data.val = a→data.val % b.data.val;
}
```

71. 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]);
}
```

72. 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_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_number_abs(&abs_x);
  mp_next_random(mp, &u);    /* take_fraction(y, abs_x, u); */
  mp_number_take_fraction(mp, &y, abs_x, u);
  free_number(u); if (mp_number_equal(y, abs_x)) {    /* set_number_to_zero(*ret); */
  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); }
```

73. 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_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_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_number_abs(&abs_x); }
    while (¬mp_number_less(abs_x, u)) ;
    mp_number_make_fraction(mp, &r, xa, u);
    mp_number_clone(&xa, r);
    mp_m_log(mp, &la, u); mp_set_number_from_substraction (&la, ( ( math_data * ) mp→math ) →
        twelve_ln_2_k, la ) ; mp_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); }
```

⟨ Convert (x, y) to the octant determined by q 65 ⟩ Used in section 64.
⟨ Declarations 5, 15, 22, 26, 28, 50, 56 ⟩ Used in section 2.
⟨ Decrease k by 1, maintaining the invariant relations between x, y, and q 44 ⟩ Used in section 42.
⟨ Handle erroneous *pyth_sub* and set $a: = 0$ 49 ⟩ Used in section 47.
⟨ Handle non-positive logarithm 53 ⟩ Used in section 51.
⟨ Handle square root of zero or negative argument 43 ⟩ Used in section 42.
⟨ Handle undefined arg 58 ⟩ Used in section 57.
⟨ Increase k until x can be multiplied by a factor of 2^{-k}, and adjust y accordingly 52 ⟩ Used in section 51.
⟨ Increase z to the arg of (x, y) 61 ⟩ Used in section 60.
⟨ Internal library declarations 6, 20, 24 ⟩ Used in section 3.
⟨ Multiply y by $\exp(-z/2^{27})$ 55 ⟩ Used in section 54.
⟨ Reduce to the case that $a, c \geq 0$, $b, d > 0$ 35 ⟩ Used in section 34.
⟨ Replace a by an approximation to $\sqrt{a^2 + b^2}$ 46 ⟩ Used in section 45.
⟨ Replace a by an approximation to $\sqrt{a^2 - b^2}$ 48 ⟩ Used in section 47.
⟨ Return an appropriate answer based on z and *octant* 59 ⟩ Used in section 57.
⟨ Set variable z to the arg of (x, y) 60 ⟩ Used in section 57.
⟨ Subtract angle z from (x, y) 66 ⟩ Used in section 64.
⟨ `mpmath.h` 3 ⟩

Math support functions for 32-bit integer math