1. Introduction.

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

2. ⟨ Declarations 5 ⟩;

3. ⟨ mpmathbinary.h 3 ⟩ ≡

```
#ifndef MPMATHBINARY_H
#define MPMATHBINARY_H 1
#include "mplib.h"
#include "mpmp.h"   /* internal header */
#include <gmp.h>
#include <mpfr.h>
  ⟨ Internal library declarations 8 ⟩;
#endif
```

4. Math initialization.

First, here are some very important constants.

#define `ROUNDING MPFR_RNDN`
#define `E_STRING`
`"2.7182818284590452353602874713526624977572470936999595749669676277240766303535"`
#define `PI_STRING`
`"3.1415926535897932384626433832795028841971693993751058209749445923078164062862"`
#define *fraction_multiplier* 4096
#define *angle_multiplier* 16

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

⟨ Declarations 5 ⟩ ≡

```
#define DEBUG 0
  static void mp_binary_scan_fractional_token(MP mp, int n);
  static void mp_binary_scan_numeric_token(MP mp, int n);
  static void mp_binary_ab_vs_cd(MP mp, mp_number *ret, mp_number a, mp_number b, mp_number c,
        mp_number d);
  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_binary_crossing_point(MP mp, mp_number *ret, mp_number a, mp_number b, mp_number c);
  static void mp_binary_number_modulo(mp_number *a, mp_number b);
  static void mp_binary_print_number(MP mp, mp_number n);
  static char *mp_binary_number_tostring(MP mp, mp_number n);
  static void mp_binary_slow_add(MP mp, mp_number *ret, mp_number x_orig, mp_number y_orig);
  static void mp_binary_square_rt(MP mp, mp_number *ret, mp_number x_orig);
  static void mp_binary_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_binary_m_unif_rand(MP mp, mp_number *ret, mp_number x_orig);
  static void mp_binary_m_norm_rand(MP mp, mp_number *ret);
  static void mp_binary_m_exp(MP mp, mp_number *ret, mp_number x_orig);
  static void mp_binary_m_log(MP mp, mp_number *ret, mp_number x_orig);
  static void mp_binary_pyth_sub(MP mp, mp_number *r, mp_number a, mp_number b);
  static void mp_binary_pyth_add(MP mp, mp_number *r, mp_number a, mp_number b);
  static void mp_binary_n_arg(MP mp, mp_number *ret, mp_number x, mp_number y);
  static void mp_binary_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_binary_from_int(mp_number *A, int B);
  static void mp_set_binary_from_boolean(mp_number *A, int B);
  static void mp_set_binary_from_scaled(mp_number *A, int B);
  static void mp_set_binary_from_addition(mp_number *A, mp_number B, mp_number C);
  static void mp_set_binary_from_substraction(mp_number *A, mp_number B, mp_number C);
  static void mp_set_binary_from_div(mp_number *A, mp_number B, mp_number C);
  static void mp_set_binary_from_mul(mp_number *A, mp_number B, mp_number C);
  static void mp_set_binary_from_int_div(mp_number *A, mp_number B, int C);
  static void mp_set_binary_from_int_mul(mp_number *A, mp_number B, int C);
  static void mp_set_binary_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_binary_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_binary_fraction_to_round_scaled(mp_number *x);
static void mp_binary_number_make_scaled(MP mp, mp_number *r, mp_number p, mp_number q);
static void mp_binary_number_make_fraction(MP mp, mp_number *r, mp_number p, mp_number q);
static void mp_binary_number_take_fraction(MP mp, mp_number *r, mp_number p, mp_number q);
static void mp_binary_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_binary_from_double(mp_number *A, double B);
static void mp_free_binary_math(MP mp);
static void mp_binary_set_precision(MP mp);
static void mp_check_mpfr_t(MP mp, mpfr_t dec);
static int binary_number_check(mpfr_t dec);
static char *mp_binnumber_tostring(mpfr_t n);
static void init_binary_constants(void);
static void free_binary_constants(void);
static mpfr_prec_t precision_digits_to_bits(double i);
static double precision_bits_to_digits(mpfr_prec_t i);
```

See also sections 9, 25, 27, and 31.

This code is used in section 2.

6. We do not want special numbers as return values for functions, so:

```
#define mpfr_negative_p(a) (mpfr_sgn((a)) < 0)
#define mpfr_positive_p(a) (mpfr_sgn((a)) > 0)
#define checkZero(dec)
        if (mpfr_zero_p(dec) ∧ mpfr_negative_p(dec)) {
          mpfr_set_zero(dec, 1);
        }

  int binary_number_check(mpfr_t dec)
  {
    int test = false;
    if (¬mpfr_number_p(dec)) {
      test = true;
      if (mpfr_inf_p(dec)) {
        mpfr_set(dec, EL_GORDO_mpfr_t, ROUNDING);
        if (mpfr_negative_p(dec)) {
          mpfr_neg(dec, dec, ROUNDING);
        }
      }
      else {    /* Nan */
        mpfr_set_zero(dec, 1);    /* 1 == positive */
      }
    }
    checkZero(dec);
    return test;
  }
  void mp_check_mpfr_t(MP mp, mpfr_t dec)
  {
    mp→arith_error = binary_number_check(dec);
  }
```

7. Precision IO uses **double** because `MPFR_PREC_MAX` overflows int.

```
  static double precision_bits;
  mpfr_prec_t precision_digits_to_bits(double i)
  {
    return i/log10(2);
  }
  double precision_bits_to_digits(mpfr_prec_t d)
  {
    return d * log10(2);
  }
```

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

```
⟨Internal library declarations 8⟩ ≡
  void *mp_initialize_binary_math(MP mp);
```

This code is used in section 3.

9.

```
#define unity 1
#define two 2
#define three 3
#define four 4
#define half_unit 0.5
#define three_quarter_unit 0.75
#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 "1E-52"
#define epsilonf pow(2.0, -52.0)
#define EL_GORDO "1E1000000"   /* the largest value that METAPOST likes. */
#define one_third_EL_GORDO (EL_GORDO/3.0)
```

⟨ Declarations 5 ⟩ +≡

```
  static mpfr_t zero;
  static mpfr_t one;
  static mpfr_t minusone;
  static mpfr_t two_mpfr_t;
  static mpfr_t three_mpfr_t;
  static mpfr_t four_mpfr_t;
  static mpfr_t fraction_multiplier_mpfr_t;
  static mpfr_t angle_multiplier_mpfr_t;
  static mpfr_t fraction_one_mpfr_t;
  static mpfr_t fraction_one_plus_mpfr_t;
  static mpfr_t PI_mpfr_t;
  static mpfr_t epsilon_mpfr_t;
  static mpfr_t EL_GORDO_mpfr_t;
```

10. **void** *init_binary_constants*(**void**)

```
{
  mpfr_inits2(precision_bits, one, minusone, zero, two_mpfr_t, three_mpfr_t, four_mpfr_t,
      fraction_multiplier_mpfr_t, fraction_one_mpfr_t, fraction_one_plus_mpfr_t, angle_multiplier_mpfr_t,
      PI_mpfr_t, epsilon_mpfr_t, EL_GORDO_mpfr_t, (mpfr_ptr) 0);
  mpfr_set_si(one, 1, ROUNDING);
  mpfr_set_si(minusone, −1, ROUNDING);
  mpfr_set_si(zero, 0, ROUNDING);
  mpfr_set_si(two_mpfr_t, two, ROUNDING);
  mpfr_set_si(three_mpfr_t, three, ROUNDING);
  mpfr_set_si(four_mpfr_t, four, ROUNDING);
  mpfr_set_si(fraction_multiplier_mpfr_t, fraction_multiplier, ROUNDING);
  mpfr_set_si(fraction_one_mpfr_t, fraction_one, ROUNDING);
  mpfr_set_si(fraction_one_plus_mpfr_t, (fraction_one + 1), ROUNDING);
  mpfr_set_si(angle_multiplier_mpfr_t, angle_multiplier, ROUNDING);
  mpfr_set_str(PI_mpfr_t, PI_STRING, 10, ROUNDING);
  mpfr_set_str(epsilon_mpfr_t, epsilon, 10, ROUNDING);
  mpfr_set_str(EL_GORDO_mpfr_t, EL_GORDO, 10, ROUNDING);
}
void free_binary_constants(void)
{
  mpfr_clears(one, minusone, zero, two_mpfr_t, three_mpfr_t, four_mpfr_t, fraction_multiplier_mpfr_t,
      fraction_one_mpfr_t, fraction_one_plus_mpfr_t, angle_multiplier_mpfr_t, PI_mpfr_t, epsilon_mpfr_t,
      EL_GORDO_mpfr_t, (mpfr_ptr) 0);
  mpfr_free_cache();
}
```

11. *precision_max* is limited to 1000, because the precision of already initialized *mpfr_t* numbers cannot be raised, only lowered. The value of 1000.0 is a tradeoff between precision and allocation size / processing speed.

```
#define MAX_PRECISION 1000.0
#define DEF_PRECISION 34.0

  void *mp_initialize_binary_math(MP mp){ math_data * math = ( math_data * ) mp_xmalloc(mp, 1, sizeof
        (math_data));
    precision_bits = precision_digits_to_bits(MAX_PRECISION);
    init_binary_constants( );    /* alloc */
    math→allocate = mp_new_number;
    math→free = mp_free_number;
    mp_new_number(mp, &math→precision_default, mp_scaled_type);
    mpfr_set_d(math→precision_default.data.num, DEF_PRECISION, ROUNDING);
    mp_new_number(mp, &math→precision_max, mp_scaled_type);
    mpfr_set_d(math→precision_max.data.num, MAX_PRECISION, ROUNDING);
    mp_new_number(mp, &math→precision_min, mp_scaled_type);
      /* really should be precision_bits_to_digits(MPFR_PREC_MIN) but that produces a horrible number
        */
    mpfr_set_d(math→precision_min.data.num, 1.0, ROUNDING);
      /* here are the constants for scaled objects */
    mp_new_number(mp, &math→epsilon_t, mp_scaled_type);
    mpfr_set(math→epsilon_t.data.num, epsilon_mpfr_t, ROUNDING);
    mp_new_number(mp, &math→inf_t, mp_scaled_type);
    mpfr_set(math→inf_t.data.num, EL_GORDO_mpfr_t, ROUNDING);
    mp_new_number(mp, &math→warning_limit_t, mp_scaled_type);
    mpfr_set_d(math→warning_limit_t.data.num, warning_limit, ROUNDING);
    mp_new_number(mp, &math→one_third_inf_t, mp_scaled_type);
    mpfr_div(math→one_third_inf_t.data.num, math→inf_t.data.num, three_mpfr_t, ROUNDING);
    mp_new_number(mp, &math→unity_t, mp_scaled_type);
    mpfr_set(math→unity_t.data.num, one, ROUNDING);
    mp_new_number(mp, &math→two_t, mp_scaled_type);
    mpfr_set_si(math→two_t.data.num, two, ROUNDING);
    mp_new_number(mp, &math→three_t, mp_scaled_type);
    mpfr_set_si(math→three_t.data.num, three, ROUNDING);
    mp_new_number(mp, &math→half_unit_t, mp_scaled_type);
    mpfr_set_d(math→half_unit_t.data.num, half_unit, ROUNDING);
    mp_new_number(mp, &math→three_quarter_unit_t, mp_scaled_type);
    mpfr_set_d(math→three_quarter_unit_t.data.num, three_quarter_unit, ROUNDING);
    mp_new_number(mp, &math→zero_t, mp_scaled_type);
    mpfr_set_zero(math→zero_t.data.num, 1);    /* fractions */
    mp_new_number(mp, &math→arc_tol_k, mp_fraction_type);
    {
      mpfr_div_si(math→arc_tol_k.data.num, one, 4096, ROUNDING);
        /* quit when change in arc length estimate reaches this */
    }
    mp_new_number(mp, &math→fraction_one_t, mp_fraction_type);
    mpfr_set_si(math→fraction_one_t.data.num, fraction_one, ROUNDING);
    mp_new_number(mp, &math→fraction_half_t, mp_fraction_type);
    mpfr_set_si(math→fraction_half_t.data.num, fraction_half, ROUNDING);
    mp_new_number(mp, &math→fraction_three_t, mp_fraction_type);
    mpfr_set_si(math→fraction_three_t.data.num, fraction_three, ROUNDING);
    mp_new_number(mp, &math→fraction_four_t, mp_fraction_type);
```

```
mpfr_set_si(math→fraction_four_t.data.num, fraction_four, ROUNDING);   /* angles */
mp_new_number(mp, &math→three_sixty_deg_t, mp_angle_type);
mpfr_set_si(math→three_sixty_deg_t.data.num, 360 * angle_multiplier, ROUNDING);
mp_new_number(mp, &math→one_eighty_deg_t, mp_angle_type);
mpfr_set_si(math→one_eighty_deg_t.data.num, 180 * angle_multiplier, ROUNDING);
  /* various approximations */
mp_new_number(mp, &math→one_k, mp_scaled_type);
mpfr_set_d(math→one_k.data.num, 1.0/64, ROUNDING);
mp_new_number(mp, &math→sqrt_8_e_k, mp_scaled_type);
{
  mpfr_set_d(math→sqrt_8_e_k.data.num, 112428.82793/65536.0, ROUNDING);
    /* 2^16 √(8/e) ≈ 112428.82793 */
}
mp_new_number(mp, &math→twelve_ln_2_k, mp_fraction_type);
{
  mpfr_set_d(math→twelve_ln_2_k.data.num, 139548959.6165/65536.0, ROUNDING);
    /* 2^24 · 12 ln 2 ≈ 139548959.6165 */
}
mp_new_number(mp, &math→coef_bound_k, mp_fraction_type);
mpfr_set_d(math→coef_bound_k.data.num, coef_bound, ROUNDING);
mp_new_number(mp, &math→coef_bound_minus_1, mp_fraction_type);
mpfr_set_d(math→coef_bound_minus_1.data.num, coef_bound − 1/65536.0, ROUNDING);
mp_new_number(mp, &math→twelvebits_3, mp_scaled_type);
{
  mpfr_set_d(math→twelvebits_3.data.num, 1365/65536.0, ROUNDING);   /* 1365 ≈ 2^12/3 */
}
mp_new_number(mp, &math→twentysixbits_sqrt2_t, mp_fraction_type);
{
  mpfr_set_d(math→twentysixbits_sqrt2_t.data.num, 94906265.62/65536.0, ROUNDING);
    /* 2^26 √2 ≈ 94906265.62 */
}
mp_new_number(mp, &math→twentyeightbits_d_t, mp_fraction_type);
{
  mpfr_set_d(math→twentyeightbits_d_t.data.num, 35596754.69/65536.0, ROUNDING);
    /* 2^28 d ≈ 35596754.69 */
}
mp_new_number(mp, &math→twentysevenbits_sqrt2_d_t, mp_fraction_type);
{
  mpfr_set_d(math→twentysevenbits_sqrt2_d_t.data.num, 25170706.63/65536.0, ROUNDING);
    /* 2^27 √2 d ≈ 25170706.63 */
}   /* thresholds */
mp_new_number(mp, &math→fraction_threshold_t, mp_fraction_type);
mpfr_set_d(math→fraction_threshold_t.data.num, fraction_threshold, ROUNDING);
mp_new_number(mp, &math→half_fraction_threshold_t, mp_fraction_type);
mpfr_set_d(math→half_fraction_threshold_t.data.num, half_fraction_threshold, ROUNDING);
mp_new_number(mp, &math→scaled_threshold_t, mp_scaled_type);
mpfr_set_d(math→scaled_threshold_t.data.num, scaled_threshold, ROUNDING);
mp_new_number(mp, &math→half_scaled_threshold_t, mp_scaled_type);
mpfr_set_d(math→half_scaled_threshold_t.data.num, half_scaled_threshold, ROUNDING);
mp_new_number(mp, &math→near_zero_angle_t, mp_angle_type);
mpfr_set_d(math→near_zero_angle_t.data.num, near_zero_angle, ROUNDING);
mp_new_number(mp, &math→p_over_v_threshold_t, mp_fraction_type);
```

$mpfr\_set\_d(math\to p\_over\_v\_threshold\_t.data.num, p\_over\_v\_threshold, \texttt{ROUNDING});$
$mp\_new\_number(mp, \&math\to equation\_threshold\_t, mp\_scaled\_type);$
$mpfr\_set\_d(math\to equation\_threshold\_t.data.num, equation\_threshold, \texttt{ROUNDING});$
$mp\_new\_number(mp, \&math\to tfm\_warn\_threshold\_t, mp\_scaled\_type);$
$mpfr\_set\_d(math\to tfm\_warn\_threshold\_t.data.num, tfm\_warn\_threshold, \texttt{ROUNDING});$
/* functions */
$math\to from\_int = mp\_set\_binary\_from\_int;$
$math\to from\_boolean = mp\_set\_binary\_from\_boolean;$
$math\to from\_scaled = mp\_set\_binary\_from\_scaled;$
$math\to from\_double = mp\_set\_binary\_from\_double;$
$math\to from\_addition = mp\_set\_binary\_from\_addition;$
$math\to from\_substraction = mp\_set\_binary\_from\_substraction;$
$math\to from\_oftheway = mp\_set\_binary\_from\_of\_the\_way;$
$math\to from\_div = mp\_set\_binary\_from\_div;$
$math\to from\_mul = mp\_set\_binary\_from\_mul;$
$math\to from\_int\_div = mp\_set\_binary\_from\_int\_div;$
$math\to from\_int\_mul = mp\_set\_binary\_from\_int\_mul;$
$math\to negate = mp\_number\_negate;$
$math\to add = mp\_number\_add;$
$math\to substract = mp\_number\_substract;$
$math\to half = mp\_number\_half;$
$math\to halfp = mp\_number\_halfp;$
$math\to do\_double = mp\_number\_double;$
$math\to abs = mp\_binary\_abs;$
$math\to clone = mp\_number\_clone;$
$math\to swap = mp\_number\_swap;$
$math\to add\_scaled = mp\_number\_add\_scaled;$
$math\to multiply\_int = mp\_number\_multiply\_int;$
$math\to divide\_int = mp\_number\_divide\_int;$
$math\to to\_boolean = mp\_number\_to\_boolean;$
$math\to to\_scaled = mp\_number\_to\_scaled;$
$math\to to\_double = mp\_number\_to\_double;$
$math\to to\_int = mp\_number\_to\_int;$
$math\to odd = mp\_number\_odd;$
$math\to equal = mp\_number\_equal;$
$math\to less = mp\_number\_less;$
$math\to greater = mp\_number\_greater;$
$math\to nonequalabs = mp\_number\_nonequalabs;$
$math\to round\_unscaled = mp\_round\_unscaled;$
$math\to floor\_scaled = mp\_number\_floor;$
$math\to fraction\_to\_round\_scaled = mp\_binary\_fraction\_to\_round\_scaled;$
$math\to make\_scaled = mp\_binary\_number\_make\_scaled;$
$math\to make\_fraction = mp\_binary\_number\_make\_fraction;$
$math\to take\_fraction = mp\_binary\_number\_take\_fraction;$
$math\to take\_scaled = mp\_binary\_number\_take\_scaled;$
$math\to velocity = mp\_binary\_velocity;$
$math\to n\_arg = mp\_binary\_n\_arg;$
$math\to m\_log = mp\_binary\_m\_log;$
$math\to m\_exp = mp\_binary\_m\_exp;$
$math\to m\_unif\_rand = mp\_binary\_m\_unif\_rand;$
$math\to m\_norm\_rand = mp\_binary\_m\_norm\_rand;$
$math\to pyth\_add = mp\_binary\_pyth\_add;$

math→*pyth_sub* = *mp_binary_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_binary_sin_cos*;
math→*slow_add* = *mp_binary_slow_add*;
math→*sqrt* = *mp_binary_square_rt*;
math→*print* = *mp_binary_print_number*;
math→*tostring* = *mp_binary_number_tostring*;
math→*modulo* = *mp_binary_number_modulo*;
math→*ab_vs_cd* = *mp_ab_vs_cd*;
math→*crossing_point* = *mp_binary_crossing_point*;
math→*scan_numeric* = *mp_binary_scan_numeric_token*;
math→*scan_fractional* = *mp_binary_scan_fractional_token*;
math→*free_math* = *mp_free_binary_math*;
math→*set_precision* = *mp_binary_set_precision*;
return (**void** ∗) *math*; } **void** *mp_binary_set_precision*(MP *mp*)
{
double *d* = *mpfr_get_d*(*internal_value*(*mp_number_precision*).*data*.*num*, ROUNDING);
precision_bits = *precision_digits_to_bits*(*d*);
}
void *mp_free_binary_math*(MP *mp*){ *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*) → *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*) → *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*) → *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_binary_constants( );
free(*mp*→*math*); }

12. Creating an destroying *mp_number* objects

13. **void** *mp_new_number*(MP *mp*, *mp_number* ∗*n*, *mp_number_type t*)

```
{
  (void) mp;
  n→data.num = mp_xmalloc(mp, 1, sizeof (mpfr_t));
  mpfr_init2((mpfr_ptr)(n→data.num), precision_bits);
  mpfr_set_zero((mpfr_ptr)(n→data.num), 1);    /* 1 == positive */
  n→type = t;
}
```

14.

```
void mp_free_number(MP mp, mp_number *n)
{
  (void) mp;
  if (n→data.num) {
    mpfr_clear(n→data.num);
    n→data.num = Λ;
  }
  n→type = mp_nan_type;
}
```

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

```
void mp_set_binary_from_int(mp_number *A, int B)
{
  mpfr_set_si(A→data.num, B, ROUNDING);
}
void mp_set_binary_from_boolean(mp_number *A, int B)
{
  mpfr_set_si(A→data.num, B, ROUNDING);
}
void mp_set_binary_from_scaled(mp_number *A, int B)
{
  mpfr_set_si(A→data.num, B, ROUNDING);
  mpfr_div_si(A→data.num, A→data.num, 65536, ROUNDING);
}
void mp_set_binary_from_double(mp_number *A, double B)
{
  mpfr_set_d(A→data.num, B, ROUNDING);
}
void mp_set_binary_from_addition(mp_number *A, mp_number B, mp_number C)
{
  mpfr_add(A→data.num, B.data.num, C.data.num, ROUNDING);
}
void mp_set_binary_from_substraction(mp_number *A, mp_number B, mp_number C)
{
  mpfr_sub(A→data.num, B.data.num, C.data.num, ROUNDING);
}
void mp_set_binary_from_div(mp_number *A, mp_number B, mp_number C)
{
  mpfr_div(A→data.num, B.data.num, C.data.num, ROUNDING);
}
void mp_set_binary_from_mul(mp_number *A, mp_number B, mp_number C)
{
  mpfr_mul(A→data.num, B.data.num, C.data.num, ROUNDING);
}
void mp_set_binary_from_int_div(mp_number *A, mp_number B, int C)
{
  mpfr_div_si(A→data.num, B.data.num, C, ROUNDING);
}
void mp_set_binary_from_int_mul(mp_number *A, mp_number B, int C)
{
  mpfr_mul_si(A→data.num, B.data.num, C, ROUNDING);
}
void mp_set_binary_from_of_the_way(MP mp, mp_number *A, mp_number t, mp_number B, mp_number C)
{
  mpfr_t c, r1;
  mpfr_init2(c, precision_bits);
  mpfr_init2(r1, precision_bits);
  mpfr_sub(c, B.data.num, C.data.num, ROUNDING);
  mp_binary_take_fraction(mp, r1, c, t.data.num);
```

```
  mpfr_sub(A→data.num, B.data.num, r1, ROUNDING);
  mpfr_clear(c);
  mpfr_clear(r1);
  mp_check_mpfr_t(mp, A→data.num);
}
void mp_number_negate(mp_number *A)
{
  mpfr_neg(A→data.num, A→data.num, ROUNDING);
  checkZero((mpfr_ptr)A→data.num);
}
void mp_number_add(mp_number *A, mp_number B)
{
  mpfr_add(A→data.num, A→data.num, B.data.num, ROUNDING);
}
void mp_number_substract(mp_number *A, mp_number B)
{
  mpfr_sub(A→data.num, A→data.num, B.data.num, ROUNDING);
}
void mp_number_half(mp_number *A)
{
  mpfr_div_si(A→data.num, A→data.num, 2, ROUNDING);
}
void mp_number_halfp(mp_number *A)
{
  mpfr_div_si(A→data.num, A→data.num, 2, ROUNDING);
}
void mp_number_double(mp_number *A)
{
  mpfr_mul_si(A→data.num, A→data.num, 2, ROUNDING);
}
void mp_number_add_scaled(mp_number *A, int B)
{    /* also for negative B */
  mpfr_add_d(A→data.num, A→data.num, B/65536.0, ROUNDING);
}
void mp_number_multiply_int(mp_number *A, int B)
{
  mpfr_mul_si(A→data.num, A→data.num, B, ROUNDING);
}
void mp_number_divide_int(mp_number *A, int B)
{
  mpfr_div_si(A→data.num, A→data.num, B, ROUNDING);
}
void mp_binary_abs(mp_number *A)
{
  mpfr_abs(A→data.num, A→data.num, ROUNDING);
}
void mp_number_clone(mp_number *A, mp_number B)
{
  mpfr_prec_round(A→data.num, precision_bits, ROUNDING);
```

```
  mpfr_set(A->data.num, (mpfr_ptr) B.data.num, ROUNDING);
}
void mp_number_swap(mp_number *A, mp_number *B)
{
  mpfr_swap(A->data.num, B->data.num);
}
void mp_number_fraction_to_scaled(mp_number *A)
{
  A->type = mp_scaled_type;
  mpfr_div(A->data.num, A->data.num, fraction_multiplier_mpfr_t, ROUNDING);
}
void mp_number_angle_to_scaled(mp_number *A)
{
  A->type = mp_scaled_type;
  mpfr_div(A->data.num, A->data.num, angle_multiplier_mpfr_t, ROUNDING);
}
void mp_number_scaled_to_fraction(mp_number *A)
{
  A->type = mp_fraction_type;
  mpfr_mul(A->data.num, A->data.num, fraction_multiplier_mpfr_t, ROUNDING);
}
void mp_number_scaled_to_angle(mp_number *A)
{
  A->type = mp_angle_type;
  mpfr_mul(A->data.num, A->data.num, angle_multiplier_mpfr_t, ROUNDING);
}
```

16. Query functions.

17. Convert a number to a scaled value. *decNumberToInt32* is not able to make this conversion properly, so instead we are using *decNumberToDouble* and a typecast. Bad!

```
int mp_number_to_scaled(mp_number A)
{
  double v = mpfr_get_d(A.data.num, ROUNDING);
  return (int)(v * 65536.0);
}
```

18.

```
#define odd(A) (abs(A) % 2 ≡ 1)
  int mp_number_to_int(mp_number A)
  {
    int32_t result = 0;
    if (mpfr_fits_sint_p(A.data.num, ROUNDING)) {
      result = mpfr_get_si(A.data.num, ROUNDING);
    }
    return result;
  }
  int mp_number_to_boolean(mp_number A)
  {
    int32_t result = 0;
    if (mpfr_fits_sint_p(A.data.num, ROUNDING)) {
      result = mpfr_get_si(A.data.num, ROUNDING);
    }
    return result;
  }
  double mp_number_to_double(mp_number A)
  {
    double res = 0.0;
    if (mpfr_number_p(A.data.num)) {
      res = mpfr_get_d(A.data.num, ROUNDING);
    }
    return res;
  }
  int mp_number_odd(mp_number A)
  {
    return odd(mp_number_to_int(A));
  }
  int mp_number_equal(mp_number A, mp_number B)
  {
    return mpfr_equal_p(A.data.num, B.data.num);
  }
  int mp_number_greater(mp_number A, mp_number B)
  {
    return mpfr_greater_p(A.data.num, B.data.num);
  }
  int mp_number_less(mp_number A, mp_number B)
  {
    return mpfr_less_p(A.data.num, B.data.num);
  }
  int mp_number_nonequalabs(mp_number A, mp_number B)
  {
    return ¬(mpfr_cmpabs(A.data.num, B.data.num) ≡ 0);
  }
```

19. 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.

20. 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 '$\mathit{half}\,(a+b)$' 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.

21. 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_binnumber_tostring(mpfr_t n)
{
  char *str = Λ, *buffer = Λ;
  mpfr_exp_t exp = 0;
  int neg = 0;
  if ((str = mpfr_get_str(Λ, &exp, 10, 0, n, ROUNDING)) > 0) {
    int numprecdigits = precision_bits_to_digits(precision_bits);
    if (*str ≡ '-') {
      neg = 1;
    }
    while (strlen(str) > 0 ∧ *(str + strlen(str) − 1) ≡ '0') {
      *(str + strlen(str) − 1) = '\0';    /* get rid of trailing zeroes */
    }
    buffer = malloc(strlen(str) + 13 + numprecdigits + 1);    /* the buffer should also fit at least
        strlen("E+%d", exp) or (numprecdigits-2) worth of zeroes, * because with numprecdigits ==
        33, the str for "1E32" will be "1", and needing 32 extra zeroes, * and the decimal dot. To avoid
        miscalculations by myself, it is safer to add these * three together. */
    if (buffer) {
      int i = 0, j = 0;
      if (neg) {
        buffer[i++] = '-';
        j = 1;
      }
      if (strlen(str + j) ≡ 0) {
        buffer[i++] = '0';
      }
      else {    /* non-zero */
        if (exp ≤ numprecdigits ∧ exp > −6) {
          if (exp > 0) {
            buffer[i++] = str[j++];
            while (−− exp > 0) {
              buffer[i++] = (str[j] ? str[j++] : '0');
            }
            if (str[j]) {
              buffer[i++] = '.';
              while (str[j]) {
                buffer[i++] = str[j++];
              }
            }
          }
          else {
            int absexp;
            buffer[i++] = '0';
            buffer[i++] = '.';
            absexp = −exp;
            while (absexp −− > 0) {
              buffer[i++] = '0';
            }
```

```
          while (str[j]) {
            buffer[i++] = str[j++];
          }
        }
      }
      else {
        buffer[i++] = str[j++];
        if (str[j]) {
          buffer[i++] = '.';
          while (str[j]) {
            buffer[i++] = str[j++];
          }
        }
        {
          char msg[256];
          int k = 0;
          mp_snprintf(msg, 256, "%s%d", (exp > 0 ? "+" : ""), (int)(exp > 0 ? (exp - 1) : (exp - 1)));
          buffer[i++] = 'E';
          while (msg[k]) {
            buffer[i++] = msg[k++];
          }
        }
      }
    }
    buffer[i++] = '\0';
  }
  mpfr_free_str(str);
 }
 return buffer;
}
char *mp_binary_number_tostring(MP mp, mp_number n)
{
  return mp_binnumber_tostring(n.data.num);
}
```

22.

```
void mp_binary_print_number(MP mp, mp_number n)
{
  char *str = mp_binary_number_tostring(mp, n);
  mp_print(mp, str);
  free(str);
}
```

23. 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_binary_slow_add(MP mp, mp_number *ret, mp_number A, mp_number B)
{
  mpfr_add(ret->data.num, A.data.num, B.data.num, ROUNDING);
}
```

24. 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.

```
void mp_binary_make_fraction(MP mp, mpfr_t ret, mpfr_t p, mpfr_t q)
{
  mpfr_div(ret, p, q, ROUNDING);
  mp_check_mpfr_t(mp, ret);
  mpfr_mul(ret, ret, fraction_multiplier_mpfr_t, ROUNDING);
}
void mp_binary_number_make_fraction(MP mp, mp_number *ret, mp_number p, mp_number q)
{
  mp_binary_make_fraction(mp, ret->data.num, p.data.num, q.data.num);
}
```

25. ⟨ Declarations 5 ⟩ +≡

```
void mp_binary_make_fraction(MP mp, mpfr_t ret, mpfr_t p, mpfr_t q);
```

26. 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.

```
void mp_binary_take_fraction(MP mp, mpfr_t ret, mpfr_t p, mpfr_t q)
{
  mpfr_mul(ret, p, q, ROUNDING);
  mpfr_div(ret, ret, fraction_multiplier_mpfr_t, ROUNDING);
}
void mp_binary_number_take_fraction(MP mp, mp_number *ret, mp_number p, mp_number q)
{
  mp_binary_take_fraction(mp, ret->data.num, p.data.num, q.data.num);
}
```

27. ⟨ Declarations 5 ⟩ +≡

```
void mp_binary_take_fraction(MP mp, mpfr_t ret, mpfr_t p, mpfr_t q);
```

28. 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_binary_number_take_scaled(MP mp, mp_number *ret, mp_number p_orig, mp_number q_orig)
{
  mpfr_mul(ret→data.num, p_orig.data.num, q_orig.data.num, ROUNDING);
}
```

29. 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.)

```
void mp_binary_number_make_scaled(MP mp, mp_number *ret, mp_number p_orig, mp_number q_orig)
{
  mpfr_div(ret→data.num, p_orig.data.num, q_orig.data.num, ROUNDING);
  mp_check_mpfr_t(mp, ret→data.num);
}
```

30.

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

31. Scanning numbers in the input.

The definitions below are temporarily here.

#define $set\_cur\_cmd(A)$ $mp\to cur\_mod\_\to type = (A)$
#define $set\_cur\_mod(A)$ $mpfr\_set((mpfr\_ptr)(mp\to cur\_mod\_\to data.n.data.num), A,$ `ROUNDING`$)$

⟨ Declarations 5 ⟩ +≡

static void $mp\_wrapup\_numeric\_token($`MP` mp, **unsigned char** $*start$, **unsigned char** $*stop)$;

32. The check of the precision is based on the article "27 Bits are not enough for 8-Digit accuracy"

33. by Bennet Goldberg which roughly says that

34. given p digits in base 10 and q digits in base 2,

35. conversion from base 10 round-trip through base 2 if and only if $10^p < 2^{q-1}$.

36. In our case $p/\log_{10} 2 + 1 < q$, or $q \geq a$

37. where q is the current precision in bits and $a = \lceil p/\log_{10} 2 + 1 \rceil$.

38. Therefore if $a > q$ the required precision could be too high and we emit a warning.

```
#define too_precise(a) (a > precision_bits)
  void mp_wrapup_numeric_token(MP mp, unsigned char *start, unsigned char *stop){ int invalid = 0;
    mpfr_t result;
    size_t l = stop - start + 1;
    unsigned long lp, lpbit;
    char *buf = mp_xmalloc(mp, l + 1, 1);
    char *bufp = buf;
    buf[l] = '\0';
    mpfr_init2(result, precision_bits);
    (void) strncpy(buf, (const char *) start, l);
    invalid = mpfr_set_str(result, buf, 10, ROUNDING);
      /* fprintf(stdout, "scan of [%s] produced %s, ", buf, mp_binnumber_tostring(result)); */
    lp = (unsigned long) l;    /* strip leading - or + or 0 or . */
    if ((*bufp ≡ '-') ∨ (*bufp ≡ '+') ∨ (*bufp ≡ '0') ∨ (*bufp ≡ '.')) {
      lp--;
      bufp++;
    }    /* strip also . */
    lp = strchr(bufp, '.') ? lp - 1 : lp;    /* strip also trailing 0s */
    bufp = buf + l - 1;
    while (*bufp ≡ '0') {
      bufp--;
      lp = (((lp ≡ 0) ∨ (lp ≡ 1)) ? 1 : lp - 1);
    }    /* at least one digit, even if the number is 0 */
    lp = lp > 0 ? lp : 1;    /* bits needed for buf */
    lpbit = (unsigned long) ceil(lp/log10(2) + 1);
    free(buf);
    bufp = Λ;
    if (invalid ≡ 0) {
      set_cur_mod(result);
        /* fprintf(stdout, "mod=%s\n", mp_binary_number_tostring(mp, mp→cur_mod_→data.n)); */
      if (too_precise(lpbit)) {
        if (mpfr_positive_p((mpfr_ptr)(internal_value(mp_warning_check).data.num)) ∧
              (mp→scanner_status ≠ tex_flushing)) {
          char msg[256];
          const char *hlp[] = {"Continue and I'll try to cope",
              "with that value; but it might be dangerous.",
              "(Set warningcheck:=0 to suppress this message.)", Λ};
          mp_snprintf(msg,
              256, "Required precision is too high (%d vs. numberprecision = %f\
              , required precision=%d bits vs internal precision=%f bits)", (unsigned
              int) lp, mpfr_get_d(internal_value(mp_number_precision).data.num, ROUNDING), (int)
              lpbit, precision_bits);
          ;
          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 ( (mpfr_ptr) ( ( ( math_data * ) (mp→math) ) → inf_t.data.num ) ) ; }
      set_cur_cmd((mp_variable_type) mp_numeric_token);
    mpfr_clear(result); }
```

39.
```
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_binary_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);
}
```

40. We just have to collect bytes.

```
void mp_binary_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);
}
```

41. 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 (fraction_multiplier/2)
#define fraction_one (1 * fraction_multiplier)
#define fraction_two (2 * fraction_multiplier)
#define fraction_three (3 * fraction_multiplier)
#define fraction_four (4 * fraction_multiplier)
```

42. 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 $\sin\theta\cos\theta\mathrm{L}\frac{1}{2}$.) 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_binary_velocity(MP mp, mp_number *ret, mp_number st, mp_number ct, mp_number sf,
      mp_number cf, mp_number t)
{
  mpfr_t acc, num, denom;   /* registers for intermediate calculations */
  mpfr_t r1, r2;
  mpfr_t arg1, arg2;
  mpfr_t i16, fone, fhalf, ftwo, sqrtfive;
  mpfr_inits2(precision_bits, acc, num, denom, r1, r2, arg1, arg2, i16, fone, fhalf, ftwo, sqrtfive,
      (mpfr_ptr) 0);
  mpfr_set_si(i16, 16, ROUNDING);
  mpfr_set_si(fone, fraction_one, ROUNDING);
  mpfr_set_si(fhalf, fraction_half, ROUNDING);
  mpfr_set_si(ftwo, fraction_two, ROUNDING);
  mpfr_set_si(sqrtfive, 5, ROUNDING);
  mpfr_sqrt(sqrtfive, sqrtfive, ROUNDING);
  mpfr_div(arg1, sf.data.num, i16, ROUNDING);   /* arg1 = sf / 16 */
  mpfr_sub(arg1, st.data.num, arg1, ROUNDING);   /* arg1 = st - arg1 */
  mpfr_div(arg2, st.data.num, i16, ROUNDING);   /* arg2 = st / 16 */
  mpfr_sub(arg2, sf.data.num, arg2, ROUNDING);   /* arg2 = sf - arg2 */
  mp_binary_take_fraction(mp, acc, arg1, arg2);   /* acc = (arg1 * arg2) / fmul */
  mpfr_set(arg1, acc, ROUNDING);
  mpfr_sub(arg2, ct.data.num, cf.data.num, ROUNDING);   /* arg2 = ct - cf */
  mp_binary_take_fraction(mp, acc, arg1, arg2);   /* acc = (arg1 * arg2 ) / fmul */
  mpfr_sqrt(arg1, two_mpfr_t, ROUNDING);   /* arg1 = sqrt(2) */
  mpfr_mul(arg1, arg1, fone, ROUNDING);   /* arg1 = arg1 * fmul */
  mp_binary_take_fraction(mp, r1, acc, arg1);   /* r1 = (acc * arg1) / fmul */
  mpfr_add(num, ftwo, r1, ROUNDING);   /* num = ftwo + r1 */
  mpfr_sub(arg1, sqrtfive, one, ROUNDING);   /* arg1 = sqrt(5) - 1 */
  mpfr_mul(arg1, arg1, fhalf, ROUNDING);   /* arg1 = arg1 * fmul/2 */
  mpfr_mul(arg1, arg1, three_mpfr_t, ROUNDING);   /* arg1 = arg1 * 3 */
  mpfr_sub(arg2, three_mpfr_t, sqrtfive, ROUNDING);   /* arg2 = 3 - sqrt(5) */
  mpfr_mul(arg2, arg2, fhalf, ROUNDING);   /* arg2 = arg2 * fmul/2 */
  mpfr_mul(arg2, arg2, three_mpfr_t, ROUNDING);   /* arg2 = arg2 * 3 */
  mp_binary_take_fraction(mp, r1, ct.data.num, arg1);   /* r1 = (ct * arg1) / fmul */
  mp_binary_take_fraction(mp, r2, cf.data.num, arg2);   /* r2 = (cf * arg2) / fmul */
  mpfr_set_si(denom, fraction_three, ROUNDING);   /* denom = 3fmul */
  mpfr_add(denom, denom, r1, ROUNDING);   /* denom = denom + r1 */
  mpfr_add(denom, denom, r2, ROUNDING);   /* denom = denom + r2 */
```

```
  if (¬mpfr_equal_p(t.data.num, one)) {   /* t != 1 */
    mpfr_div(num, num, t.data.num, ROUNDING);   /* num = num / t */
  }
  mpfr_set(r2, num, ROUNDING);   /* r2 = num / 4 */
  mpfr_div(r2, r2, four_mpfr_t, ROUNDING);
  if (mpfr_less_p(denom, r2)) {   /* num/4 ¿= denom =¿ denom ¡ num/4 */
    mpfr_set_si(ret→data.num, fraction_four, ROUNDING);
  }
  else {
    mp_binary_make_fraction(mp, ret→data.num, num, denom);
  }
  mpfr_clears(acc, num, denom, r1, r2, arg1, arg2, i16, fone, fhalf, ftwo, sqrtfive, (mpfr_ptr)0);
  mp_check_mpfr_t(mp, ret→data.num);
}
```

43. 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)
{
  mpfr_t q, r, test;   /* temporary registers */
  mpfr_t a, b, c, d;
  int cmp = 0;
  (void) mp;
  mpfr_inits2(precision_bits, q, r, test, a, b, c, d, (mpfr_ptr) 0);
  mpfr_set(a, (mpfr_ptr) a_orig.data.num, ROUNDING);
  mpfr_set(b, (mpfr_ptr) b_orig.data.num, ROUNDING);
  mpfr_set(c, (mpfr_ptr) c_orig.data.num, ROUNDING);
  mpfr_set(d, (mpfr_ptr) d_orig.data.num, ROUNDING);
  mpfr_mul(q, a, b, ROUNDING);
  mpfr_mul(r, c, d, ROUNDING);
  cmp = mpfr_cmp(q, r);
  if (cmp ≡ 0) {
    mpfr_set(ret→data.num, zero, ROUNDING);
    goto RETURN;
  }
  if (cmp > 0) {
    mpfr_set(ret→data.num, one, ROUNDING);
    goto RETURN;
  }
  if (cmp < 0) {
    mpfr_set(ret→data.num, minusone, ROUNDING);
    goto RETURN;
  }   /* TODO: remove this part of the code until RETURN */
  ⟨Reduce to the case that a, c ≥ 0, b, d > 0 44⟩;
  while (1) {
    mpfr_div(q, a, d, ROUNDING);
    mpfr_div(r, c, b, ROUNDING);
    cmp = mpfr_cmp(q, r);
    if (cmp) {
      if (cmp > 1) {
        mpfr_set(ret→data.num, one, ROUNDING);
      }
      else {
        mpfr_set(ret→data.num, minusone, ROUNDING);
      }
      goto RETURN;
    }
    mpfr_remainder(q, a, d, ROUNDING);
    mpfr_remainder(r, c, b, ROUNDING);
    if (mpfr_zero_p(r)) {
      if (mpfr_zero_p(q)) {
        mpfr_set(ret→data.num, zero, ROUNDING);
      }
      else {
```

```
          mpfr_set(ret→data.num, one, ROUNDING);
        }
        goto RETURN;
      }
      if (mpfr_zero_p(q)) {
        mpfr_set(ret→data.num, minusone, ROUNDING);
        goto RETURN;
      }
      mpfr_set(a, b, ROUNDING);
      mpfr_set(b, q, ROUNDING);
      mpfr_set(c, d, ROUNDING);
      mpfr_set(d, r, ROUNDING);
    }    /* 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
    mp_check_mpfr_t(mp, ret→data.num);
    mpfr_clears(q, r, test, a, b, c, d, (mpfr_ptr)0);
    return;
  }
```

44. ⟨Reduce to the case that $a, c \ge 0$, $b, d > 0$ 44⟩ ≡

```
if (mpfr_negative_p(a)) {
  mpfr_neg(a, a, ROUNDING);
  mpfr_neg(b, b, ROUNDING);
}
if (mpfr_negative_p(c)) {
  mpfr_neg(c, c, ROUNDING);
  mpfr_neg(d, d, ROUNDING);
}
if (¬mpfr_positive_p(d)) {
  if (¬mpfr_negative_p(b)) {
    if ((mpfr_zero_p(a) ∨ mpfr_zero_p(b)) ∧ (mpfr_zero_p(c) ∨ mpfr_zero_p(d)))
      mpfr_set(ret→data.num, zero, ROUNDING);
    else mpfr_set(ret→data.num, one, ROUNDING);
    goto RETURN;
  }
  if (mpfr_zero_p(d)) {
    if (mpfr_zero_p(a)) mpfr_set(ret→data.num, zero, ROUNDING);
    else mpfr_set(ret→data.num, minusone, ROUNDING);
    goto RETURN;
  }
  mpfr_set(q, a, ROUNDING);
  mpfr_set(a, c, ROUNDING);
  mpfr_set(c, q, ROUNDING);
  mpfr_neg(q, b, ROUNDING);
  mpfr_neg(b, d, ROUNDING);
  mpfr_set(d, q, ROUNDING);
}
else if (¬mpfr_positive_p(b)) {
  if (mpfr_negative_p(b) ∧ mpfr_positive_p(a)) {
    mpfr_set(ret→data.num, minusone, ROUNDING);
    goto RETURN;
  }
  if (mpfr_zero_p(c)) mpfr_set(ret→data.num, zero, ROUNDING);
  else mpfr_set(ret→data.num, minusone, ROUNDING);
  goto RETURN;
}
```

This code is used in section 43.

45. 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
        {
          mpfr_set(ret→data.num, fraction_one_plus_mpfr_t, ROUNDING);
          goto RETURN;
        }
#define one_crossing
        {
          mpfr_set(ret→data.num, fraction_one_mpfr_t, ROUNDING);
          goto RETURN;
        }
#define zero_crossing
        {
          mpfr_set(ret→data.num, zero, ROUNDING);
          goto RETURN;
        }

  static void mp_binary_crossing_point(MP mp, mp_number *ret, mp_number aa, mp_number bb, mp_number cc)
  {
    mpfr_t a, b, c;
    double d;     /* recursive counter */
    mpfr_t x, xx, x0, x1, x2;     /* temporary registers for bisection */
    mpfr_t scratch;
    mpfr_inits2(precision_bits, a, b, c, x, xx, x0, x1, x2, scratch, (mpfr_ptr) 0);
    mpfr_set(a, (mpfr_ptr) aa.data.num, ROUNDING);
    mpfr_set(b, (mpfr_ptr) bb.data.num, ROUNDING);
    mpfr_set(c, (mpfr_ptr) cc.data.num, ROUNDING);
    if (mpfr_negative_p(a)) zero_crossing;
    if (¬mpfr_negative_p(c)) {
      if (¬mpfr_negative_p(b)) {
        if (mpfr_positive_p(c)) {
          no_crossing;
        }
        else if (mpfr_zero_p(a) ∧ mpfr_zero_p(b)) {
          no_crossing;
        }
        else {
          one_crossing;
```

```
      }
    }
    if (mpfr_zero_p(a)) zero_crossing;
  }
  else if (mpfr_zero_p(a)) {
    if (¬mpfr_positive_p(b)) zero_crossing;
  }    /* Use bisection to find the crossing point... */
  d = epsilonf;
  mpfr_set(x0, a, ROUNDING);
  mpfr_sub(x1, a, b, ROUNDING);
  mpfr_sub(x2, b, c, ROUNDING);
  do {     /* not sure why the error correction has to be ¿= 1E-12 */
    mpfr_add(x, x1, x2, ROUNDING);
    mpfr_div(x, x, two_mpfr_t, ROUNDING);
    mpfr_add_d(x, x, 1 · 10^-12, ROUNDING);
    mpfr_sub(scratch, x1, x0, ROUNDING);
    if (mpfr_greater_p(scratch, x0)) {
      mpfr_set(x2, x, ROUNDING);
      mpfr_add(x0, x0, x0, ROUNDING);
      d += d;
    }
    else {
      mpfr_add(xx, scratch, x, ROUNDING);
      if (mpfr_greater_p(xx, x0)) {
        mpfr_set(x2, x, ROUNDING);
        mpfr_add(x0, x0, x0, ROUNDING);
        d += d;
      }
      else {
        mpfr_sub(x0, x0, xx, ROUNDING);
        if (¬mpfr_greater_p(x, x0)) {
          mpfr_add(scratch, x, x2, ROUNDING);
          if (¬mpfr_greater_p(scratch, x0)) no_crossing;
        }
        mpfr_set(x1, x, ROUNDING);
        d = d + d + epsilonf;
      }
    }
  } while (d < fraction_one);
  mpfr_set_d(scratch, d, ROUNDING);
  mpfr_sub(ret→data.num, scratch, fraction_one_mpfr_t, ROUNDING);
 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
  mpfr_clears(a, b, c, x, xx, x0, x1, x2, scratch, (mpfr_ptr)0);
  mp_check_mpfr_t(mp, ret→data.num);
  return;
}
```

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

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

```
int mp_round_unscaled(mp_number x_orig)
{
  double xx = mp_number_to_double(x_orig);
  int x = (int) ROUND(xx);
  return x;
}
```

48. *number_floor* floors a number

```
void mp_number_floor(mp_number *i)
{
  mpfr_rint_floor(i→data.num, i→data.num, MPFR_RNDD);
}
```

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

```
void mp_binary_fraction_to_round_scaled(mp_number *x_orig)
{
  x_orig→type = mp_scaled_type;
  mpfr_div(x_orig→data.num, x_orig→data.num, fraction_multiplier_mpfr_t, ROUNDING);
}
```

50. 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.

51.

```
void mp_binary_square_rt(MP mp, mp_number *ret, mp_number x_orig)
{    /* return, x: scaled */
  if (¬mpfr_positive_p((mpfr_ptr)x_orig.data.num)) {
    ⟨Handle square root of zero or negative argument 52⟩;
  }
  else {
    mpfr_sqrt(ret→data.num, x_orig.data.num, ROUNDING);
  }
  mp_check_mpfr_t(mp, ret→data.num);
}
```

52. ⟨Handle square root of zero or negative argument 52⟩ ≡

```
{
  if (mpfr_negative_p((mpfr_ptr)x_orig.data.num)) {
    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_binary_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);
  }
  mpfr_set_zero(ret→data.num, 1);    /* 1 == positive */
  return;
}
```

This code is used in section 51.

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

```
void mp_binary_pyth_add(MP mp, mp_number *ret, mp_number a_orig, mp_number b_orig)
{
  mpfr_t a, b, asq, bsq;
  mpfr_inits2(precision_bits, a, b, asq, bsq, (mpfr_ptr)0);
  mpfr_set(a, (mpfr_ptr)a_orig.data.num, ROUNDING);
  mpfr_set(b, (mpfr_ptr)b_orig.data.num, ROUNDING);
  mpfr_mul(asq, a, a, ROUNDING);
  mpfr_mul(bsq, b, b, ROUNDING);
  mpfr_add(a, asq, bsq, ROUNDING);
  mpfr_sqrt(ret→data.num, a, ROUNDING);
  mp_check_mpfr_t(mp, ret→data.num);
  mpfr_clears(a, b, asq, bsq, (mpfr_ptr)0);
}
```

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

```
void mp_binary_pyth_sub(MP mp, mp_number *ret, mp_number a_orig, mp_number b_orig)
{
  mpfr_t a, b, asq, bsq;
  mpfr_inits2(precision_bits, a, b, asq, bsq, (mpfr_ptr) 0);
  mpfr_set(a, (mpfr_ptr) a_orig.data.num, ROUNDING);
  mpfr_set(b, (mpfr_ptr) b_orig.data.num, ROUNDING);
  if (¬mpfr_greater_p(a, b)) {
    ⟨Handle erroneous pyth_sub and set a: = 0 55⟩;
  }
  else {
    mpfr_mul(asq, a, a, ROUNDING);
    mpfr_mul(bsq, b, b, ROUNDING);
    mpfr_sub(a, asq, bsq, ROUNDING);
    mpfr_sqrt(a, a, ROUNDING);
  }
  mpfr_set(ret→data.num, a, ROUNDING);
  mp_check_mpfr_t(mp, ret→data.num);
}
```

55. ⟨Handle erroneous *pyth_sub* and set $a: = 0$ 55⟩ ≡

```
{
  if (mpfr_less_p(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_binary_number_tostring(mp, a_orig);
    char *bstr = mp_binary_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);
  }
  mpfr_set_zero(a, 1);    /* 1 == positive */
}
```

This code is used in section 54.

56. Here is the routine that calculates 2^8 times the natural logarithm of a *scaled* quantity;

```
void mp_binary_m_log(MP mp, mp_number *ret, mp_number x_orig)
{
  if (¬mpfr_positive_p((mpfr_ptr) x_orig.data.num)) {
    ⟨Handle non-positive logarithm 57⟩;
  }
  else {
    mpfr_log(ret→data.num, x_orig.data.num, ROUNDING);
    mp_check_mpfr_t(mp, ret→data.num);
    mpfr_mul_si(ret→data.num, ret→data.num, 256, ROUNDING);
  }
  mp_check_mpfr_t(mp, ret→data.num);
}
```

57. ⟨ Handle non-positive logarithm 57 ⟩ ≡

```
{
  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};
  char *xstr = mp_binary_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);
  mpfr_set_zero(ret->data.num, 1);    /* 1 == positive */
}
```

This code is used in section 56.

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

```
void mp_binary_m_exp(MP mp, mp_number *ret, mp_number x_orig)
{
  mpfr_t temp;
  mpfr_init2(temp, precision_bits);
  mpfr_div_si(temp, x_orig.data.num, 256, ROUNDING);
  mpfr_exp(ret->data.num, temp, ROUNDING);
  mp_check_mpfr_t(mp, ret->data.num);
  mpfr_clear(temp);
}
```

59. 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_binary_n_arg(MP mp, mp_number *ret, mp_number x_orig, mp_number y_orig)
{
  if (mpfr_zero_p((mpfr_ptr) x_orig.data.num) && mpfr_zero_p((mpfr_ptr) y_orig.data.num)) {
    ⟨ Handle undefined arg 60 ⟩;
  }
  else {
    mpfr_t atan2val, oneeighty_angle;
    mpfr_init2(atan2val, precision_bits);
    mpfr_init2(oneeighty_angle, precision_bits);
    ret->type = mp_angle_type;
    mpfr_set_si(oneeighty_angle, 180 * angle_multiplier, ROUNDING);
    mpfr_div(oneeighty_angle, oneeighty_angle, PI_mpfr_t, ROUNDING);
    checkZero((mpfr_ptr) y_orig.data.num);
    checkZero((mpfr_ptr) x_orig.data.num);
    mpfr_atan2(atan2val, y_orig.data.num, x_orig.data.num, ROUNDING);
    mpfr_mul(ret->data.num, atan2val, oneeighty_angle, ROUNDING);
    checkZero((mpfr_ptr) ret->data.num);
    mpfr_clear(atan2val);
    mpfr_clear(oneeighty_angle);
  }
  mp_check_mpfr_t(mp, ret->data.num);
}
```

60. ⟨ Handle undefined arg 60 ⟩ ≡

```
{
  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);
  ;
  mpfr_set_zero(ret→data.num, 1);    /* 1 == positive */
}
```

This code is used in section 59.

61. 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*.

62. Calculate sines and cosines.

```
void mp_binary_sin_cos(MP mp, mp_number z_orig, mp_number *n_cos, mp_number *n_sin)
{
  mpfr_t rad;
  mpfr_t one_eighty;
  mpfr_init2(rad, precision_bits);
  mpfr_init2(one_eighty, precision_bits);
  mpfr_set_si(one_eighty, 180 * 16, ROUNDING);
  mpfr_mul(rad, z_orig.data.num, PI_mpfr_t, ROUNDING);
  mpfr_div(rad, rad, one_eighty, ROUNDING);
  mpfr_sin(n_sin→data.num, rad, ROUNDING);
  mpfr_cos(n_cos→data.num, rad, ROUNDING);
  mpfr_mul(n_cos→data.num, n_cos→data.num, fraction_multiplier_mpfr_t, ROUNDING);
  mpfr_mul(n_sin→data.num, n_sin→data.num, fraction_multiplier_mpfr_t, ROUNDING);
  mp_check_mpfr_t(mp, n_cos→data.num);
  mp_check_mpfr_t(mp, n_sin→data.num);
  mpfr_clear(rad);
  mpfr_clear(one_eighty);
}
```

63. 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];
  }
```

64. 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;
      mpfr_set_si(mp→randoms[(i * 21) % 55].data.num, j, ROUNDING);
    }
    mp_new_randoms(mp);
    mp_new_randoms(mp);
    mp_new_randoms(mp);    /* “warm up” the array */
    ran_start((unsigned long) seed);
  }
```

65.

```
void mp_binary_number_modulo(mp_number * a, mp_number b)
  {
    mpfr_remainder(a→data.num, a→data.num, b.data.num, ROUNDING);
  }
```

66. 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)
{
  mp_number rop;
  unsigned long int op;
  float flt_op;
  (void) mp;
  mp_new_number(mp, &rop, mp_scaled_type);
  op = (unsigned) ran_arr_next();
  flt_op = op/(MM * 1.0);
  mpfr_set_d((mpfr_ptr)(rop.data.num), flt_op, ROUNDING);
  mp_number_clone(ret, rop);
  free_number(rop);
}
```

67. 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]);
}
```

68. 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_binary_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;
  char *r;
  mpfr_exp_t e;
  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_binary_abs(&abs_x);
  mp_next_unif_random(mp, &u);
  mpfr_mul(y.data.num, abs_x.data.num, u.data.num, ROUNDING);
  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);
  }
  r = mpfr_get_str(Λ,     /* char *str, */
  &e,     /* mpfr_exp_t *expptr, */
  10,     /* int b, */
  0,     /* size_t n, */
  ret->data.num,     /* mpfr_t op, */
  ROUNDING     /* mpfr_rnd_t rnd */
  );
  mpfr_free_str(r);
  free_number(abs_x);
  free_number(x);
  free_number(y); }
```

69. 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_binary_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_binary_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_binary_abs(&abs_x); }
    while (¬mp_number_less(abs_x, u)) ;
    mp_binary_number_make_fraction(mp, &r, xa, u);
    mp_number_clone(&xa, r);
    mp_binary_m_log(mp, &la, u); mp_set_binary_from_substraction (&la, ( ( math_data * ) mp→math
        ) → twelve_ln_2_k, la ) ; mp_binary_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); }
```

70. 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.

```
static void mp_binary_ab_vs_cd(MP mp, mp_number *ret, mp_number a_orig, mp_number b_orig,
        mp_number c_orig, mp_number d_orig)
{
  mpfr_t a, b, c, d;
  mpfr_t ab, cd;
  int cmp = 0;
  (void) mp;
  mpfr_inits2(precision_bits, a, b, c, d, ab, cd, (mpfr_ptr)0);
  mpfr_set(a, (mpfr_ptr) a_orig.data.num, ROUNDING);
  mpfr_set(b, (mpfr_ptr) b_orig.data.num, ROUNDING);
  mpfr_set(c, (mpfr_ptr) c_orig.data.num, ROUNDING);
  mpfr_set(d, (mpfr_ptr) d_orig.data.num, ROUNDING);
  mpfr_mul(ab, a, b, ROUNDING);
  mpfr_mul(cd, c, d, ROUNDING);
  mpfr_set(ret→data.num, zero, ROUNDING);
  cmp = mpfr_cmp(ab, cd);
  if (cmp) {
    if (cmp > 0) mpfr_set(ret→data.num, one, ROUNDING);
    else mpfr_set(ret→data.num, minusone, ROUNDING);
  }
  mp_check_mpfr_t(mp, ret→data.num);
  mpfr_clears(a, b, c, d, ab, cd, (mpfr_ptr)0);
  return;
}
```

Math support functions for MPFR based math