**1.** Introduction.

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

**2.** ⟨ Declarations 5 ⟩;

**3.** ⟨ mpmathdecimal.h 3 ⟩ ≡

```
#ifndef MPMATHDECIMAL_H
#define MPMATHDECIMAL_H 1
#include "mplib.h"
#include "mpmp.h"    /* internal header */
#define DECNUMDIGITS 1000
#include "decNumber.h"
  ⟨ Internal library declarations 9 ⟩;
#endif
```

**4. Math initialization.**

First, here are some very important constants.

**#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_decimal_scan_fractional_token(MP mp, int n);
  static void mp_decimal_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_decimal_ab_vs_cd(MP mp, mp_number *
      ret, mp_number a, mp_number b, mp_number c, mp_number d); */
  static void mp_decimal_crossing_point(MP mp, mp_number *ret, mp_number a, mp_number b, mp_number c);
  static void mp_decimal_number_modulo(mp_number *a, mp_number b);
  static void mp_decimal_print_number(MP mp, mp_number n);
  static char *mp_decimal_number_tostring(MP mp, mp_number n);
  static void mp_decimal_slow_add(MP mp, mp_number *ret, mp_number x_orig, mp_number y_orig);
  static void mp_decimal_square_rt(MP mp, mp_number *ret, mp_number x_orig);
  static void mp_decimal_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_decimal_m_unif_rand(MP mp, mp_number *ret, mp_number x_orig);
  static void mp_decimal_m_norm_rand(MP mp, mp_number *ret);
  static void mp_decimal_m_exp(MP mp, mp_number *ret, mp_number x_orig);
  static void mp_decimal_m_log(MP mp, mp_number *ret, mp_number x_orig);
  static void mp_decimal_pyth_sub(MP mp, mp_number *r, mp_number a, mp_number b);
  static void mp_decimal_pyth_add(MP mp, mp_number *r, mp_number a, mp_number b);
  static void mp_decimal_n_arg(MP mp, mp_number *ret, mp_number x, mp_number y);
  static void mp_decimal_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_decimal_from_int(mp_number *A, int B);
  static void mp_set_decimal_from_boolean(mp_number *A, int B);
  static void mp_set_decimal_from_scaled(mp_number *A, int B);
  static void mp_set_decimal_from_addition(mp_number *A, mp_number B, mp_number C);
  static void mp_set_decimal_from_substraction(mp_number *A, mp_number B, mp_number C);
  static void mp_set_decimal_from_div(mp_number *A, mp_number B, mp_number C);
  static void mp_set_decimal_from_mul(mp_number *A, mp_number B, mp_number C);
  static void mp_set_decimal_from_int_div(mp_number *A, mp_number B, int C);
  static void mp_set_decimal_from_int_mul(mp_number *A, mp_number B, int C);
  static void mp_set_decimal_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_decimal_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_decimal_fraction_to_round_scaled(mp_number *x);
static void mp_decimal_number_make_scaled(MP mp, mp_number *r, mp_number p, mp_number q);
static void mp_decimal_number_make_fraction(MP mp, mp_number *r, mp_number p, mp_number q);
static void mp_decimal_number_take_fraction(MP mp, mp_number *r, mp_number p, mp_number q);
static void mp_decimal_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_decimal_from_double(mp_number *A, double B);
static void mp_free_decimal_math(MP mp);
static void mp_decimal_set_precision(MP mp);
static void mp_check_decNumber(MP mp, decNumber *dec, decContext *context);
static int decNumber_check(decNumber *dec, decContext *context);
static char *mp_decnumber_tostring(decNumber *n);
```

See also sections 10, 25, 27, and 31.

This code is used in section 2.

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

```
int decNumber_check(decNumber *dec, decContext *context)
{
  int test = false;
  if (context->status & DEC_Overflow) {
    test = true;
    context->status &= ~DEC_Overflow;
  }
  if (context->status & DEC_Underflow) {
    test = true;
    context->status &= ~DEC_Underflow;
  }
  if (context->status & DEC_Errors) {
      /* fprintf(stdout, "DEC_ERROR %x (%s)\n", context->status, decContextStatusToString(context)); */
    test = true;
    decNumberZero(dec);
  }
  context->status = 0;
  if (decNumberIsSpecial(dec)) {
    test = true;
    if (decNumberIsInfinite(dec)) {
      if (decNumberIsNegative(dec)) {
        decNumberCopyNegate(dec, &EL_GORDO_decNumber);
      }
      else {
        decNumberCopy(dec, &EL_GORDO_decNumber);
      }
    }
    else {      /* Nan */
      decNumberZero(dec);
    }
  }
  if (decNumberIsZero(dec) && decNumberIsNegative(dec)) {
    decNumberZero(dec);
  }
  return test;
}

void mp_check_decNumber(MP mp, decNumber *dec, decContext *context)
{
  mp->arith_error = decNumber_check(dec, context);
}
```

**7.** There are a few short decNumber functions that do not exist, but make life easier for us:

```
#define decNumberIsPositive(A) ¬(decNumberIsZero(A) ∨ decNumberIsNegative(A))
  static decContext set;
  static decContext limitedset;
  static void checkZero(decNumber *ret)
  {
    if (decNumberIsZero(ret) ∧ decNumberIsNegative(ret)) decNumberZero(ret);
  }
  static int decNumberLess(decNumber *a, decNumber *b)
  {
    decNumber comp;
    decNumberCompare(&comp, a, b, &set);
    return decNumberIsNegative(&comp);
  }
  static int decNumberGreater(decNumber *a, decNumber *b)
  {
    decNumber comp;
    decNumberCompare(&comp, a, b, &set);
    return decNumberIsPositive(&comp);
  }
  static void decNumberFromDouble(decNumber *A, double B)
  {
    char buf[1000];
    char *c;
    snprintf(buf, 1000, "%-650.325lf", B);
    c = buf;
    while (*c++) {
      if (*c ≡ '␣') {
        *c = '\0';
        break;
      }
    }
    decNumberFromString(A, buf, &set);
  }
  static double decNumberToDouble(decNumber *A)
  {
    char *buffer = malloc(A→digits + 14);
    double res = 0.0;
    assert(buffer);
    decNumberToString(A, buffer);
    if (sscanf(buffer, "%lf", &res)) {
      free(buffer);
      return res;
    }
    else {
      free(buffer);    /* mp→arith_error = 1; */
      return 0.0;    /* whatever */
    }
  }
```

**8.** Borrowed code from libdfp:

$$\arctan(x) = x - \frac{x^3}{3} + \frac{x^5}{5} - \frac{x^7}{7} + \ldots$$

This power series works well, if $x$ is close to zero ($x < 0.5$). If x is larger, the series converges too slowly, so in order to get a smaller x, we apply the identity

$$\arctan(x) = 2 \ \arctan \frac{\sqrt{1+x^2}-1}{x}$$

twice. The first application gives us a new $x$ with $x < 1$. The second application gives us a new x with $x < 0.4142136$. For that $x$, we use the power series and multiply the result by four.

```
static void decNumberAtan(decNumber *result, decNumber *x_orig, decContext *set)
{
  decNumber x, f, g, mx2, term;
  int i;
  decNumberCopy(&x, x_orig);
  if (decNumberIsZero(&x)) {
    decNumberCopy(result, &x);
    return;
  }
  for (i = 0; i < 2; i++) {
    decNumber y;
    decNumberMultiply(&y, &x, &x, set);    /* y = x^2 */
    decNumberAdd(&y, &y, &one, set);    /* y = y + 1 */
    decNumberSquareRoot(&y, &y, set);    /* y = sqrt(y) */
    decNumberSubtract(&y, &y, &one, set);    /* y = y - 1 */
    decNumberDivide(&x, &y, &x, set);    /* x = y/x */
    if (decNumberIsZero(&x)) {
      decNumberCopy(result, &x);
      return;
    }
  }
  decNumberCopy(&f, &x);    /* f(0) = x */
  decNumberCopy(&g, &one);    /* g(0) = 1 */
  decNumberCopy(&term, &x);    /* term = x */
  decNumberCopy(result, &x);    /* sum = x */
  decNumberMultiply(&mx2, &x, &x, set);    /* mx2 = x^2 */
  decNumberMinus(&mx2, &mx2, set);    /* mx2 = -x^2 */
  for (i = 0; i < 2 * set->digits; i++) {
    decNumberMultiply(&f, &f, &mx2, set);
    decNumberAdd(&g, &g, &two_decNumber, set);
    decNumberDivide(&term, &f, &g, set);
    decNumberAdd(result, result, &term, set);
  }
  decNumberAdd(result, result, result, set);
  decNumberAdd(result, result, result, set);
  return;
}
static void decNumberAtan2(decNumber *result, decNumber *y, decNumber *x, decContext *set)
{
  decNumber temp;
```

```
if (¬decNumberIsInfinite(x) ∧ ¬decNumberIsZero(y) ∧ ¬decNumberIsInfinite(y) ∧ ¬decNumberIsZero(x))
  {
  decNumberDivide(&temp, y, x, set);
  decNumberAtan(result, &temp, set);    /* decNumberAtan doesn't quite return the values in the
      ranges we * want for x ¡ 0. So we need to do some correction */
  if (decNumberIsNegative(x)) {
    if (decNumberIsNegative(y)) {
      decNumberSubtract(result, result, &PI_decNumber, set);
    }
    else {
      decNumberAdd(result, result, &PI_decNumber, set);
    }
  }
  return;
}
if (decNumberIsInfinite(y) ∧ decNumberIsInfinite(x)) {
    /* If x and y are both inf, the result depends on the sign of x */
  decNumberDivide(result, &PI_decNumber, &four_decNumber, set);
  if (decNumberIsNegative(x)) {
    decNumber a;
    decNumberFromDouble(&a, 3.0);
    decNumberMultiply(result, result, &a, set);
  }
}
else if (¬decNumberIsZero(y) ∧ ¬decNumberIsInfinite(x)) {
    /* If y is non-zero and x is non-inf, the result is +-pi/2 */
  decNumberDivide(result, &PI_decNumber, &two_decNumber, set);
}
else {    /* Otherwise it is +0 if x is positive, +pi if x is neg */
  if (decNumberIsNegative(x)) {
    decNumberCopy(result, &PI_decNumber);
  }
  else {
    decNumberZero(result);
  }
}    /* Atan2 will be negative if y¡0 */
if (decNumberIsNegative(y)) {
  decNumberMinus(result, result, set);
}
}
```

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

⟨ Internal library declarations 9 ⟩ ≡

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

This code is used in section 3.

**10.**

```
#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 epsilon "1E-52"
#define epsilonf pow(2.0, -52.0)
#define EL_GORDO "1E1000000"   /* the largest value that METAPOST likes. */
#define warning_limit "1E1000000"
        /* this is a large value that can just be expressed without loss of precision */
#define DECPRECISION_DEFAULT 34
```

⟨ Declarations 5 ⟩ +≡

```
static decNumber zero;
static decNumber one;
static decNumber minusone;
static decNumber two_decNumber;
static decNumber three_decNumber;
static decNumber four_decNumber;
static decNumber fraction_multiplier_decNumber;
static decNumber angle_multiplier_decNumber;
static decNumber fraction_one_decNumber;
static decNumber fraction_one_plus_decNumber;
static decNumber PI_decNumber;
static decNumber epsilon_decNumber;
static decNumber EL_GORDO_decNumber;
static decNumber **factorials = Λ;
static int last_cached_factorial = 0;
static boolean initialized = false;
```

**11.**
```
void *mp_initialize_decimal_math(MP mp){ math_data * math = ( math_data * )
      mp_xmalloc(mp, 1, sizeof (math_data));     /* various decNumber initializations */
  decContextDefault(&set, DEC_INIT_BASE);     /* initialize */
  set.traps = 0;     /* no traps, thank you */
  decContextDefault(&limitedset, DEC_INIT_BASE);     /* initialize */
  limitedset.traps = 0;     /* no traps, thank you */
  limitedset.emax = 999999;
  limitedset.emin = -999999;
  set.digits = DECPRECISION_DEFAULT;
  limitedset.digits = DECPRECISION_DEFAULT; if (!initialized) { initialized = true;
  decNumberFromInt32(&one, 1);
  decNumberFromInt32(&minusone, -1);
  decNumberFromInt32(&zero, 0);
  decNumberFromInt32(&two_decNumber, two);
  decNumberFromInt32(&three_decNumber, three);
  decNumberFromInt32(&four_decNumber, four);
  decNumberFromInt32(&fraction_multiplier_decNumber, fraction_multiplier);
  decNumberFromInt32(&fraction_one_decNumber, fraction_one);
  decNumberFromInt32(&fraction_one_plus_decNumber, (fraction_one + 1));
  decNumberFromInt32(&angle_multiplier_decNumber, angle_multiplier);
  decNumberFromString(&PI_decNumber, PI_STRING, &set);
  decNumberFromString(&epsilon_decNumber, epsilon, &set);
  decNumberFromString(&EL_GORDO_decNumber, EL_GORDO, &set); factorials = ( decNumber ** )
      mp_xmalloc (mp, PRECALC_FACTORIALS_CACHESIZE, sizeof ( decNumber * ) ) ; factorials[0] = (
      decNumber * ) mp_xmalloc(mp, 1, sizeof (decNumber));
  decNumberCopy(factorials[0], &one); }     /* alloc */
  math->allocate = mp_new_number;
  math->free = mp_free_number;
  mp_new_number(mp, &math->precision_default, mp_scaled_type);
  decNumberFromInt32(math->precision_default.data.num, DECPRECISION_DEFAULT);
  mp_new_number(mp, &math->precision_max, mp_scaled_type);
  decNumberFromInt32(math->precision_max.data.num, DECNUMDIGITS);
  mp_new_number(mp, &math->precision_min, mp_scaled_type);
  decNumberFromInt32(math->precision_min.data.num, 1);
    /* here are the constants for scaled objects */
  mp_new_number(mp, &math->epsilon_t, mp_scaled_type);
  decNumberCopy(math->epsilon_t.data.num, &epsilon_decNumber);
  mp_new_number(mp, &math->inf_t, mp_scaled_type);
  decNumberCopy(math->inf_t.data.num, &EL_GORDO_decNumber);
  mp_new_number(mp, &math->warning_limit_t, mp_scaled_type);
  decNumberFromString(math->warning_limit_t.data.num, warning_limit, &set);
  mp_new_number(mp, &math->one_third_inf_t, mp_scaled_type);
  decNumberDivide(math->one_third_inf_t.data.num, math->inf_t.data.num, &three_decNumber, &set);
  mp_new_number(mp, &math->unity_t, mp_scaled_type);
  decNumberCopy(math->unity_t.data.num, &one);
  mp_new_number(mp, &math->two_t, mp_scaled_type);
  decNumberFromInt32(math->two_t.data.num, two);
  mp_new_number(mp, &math->three_t, mp_scaled_type);
  decNumberFromInt32(math->three_t.data.num, three);
  mp_new_number(mp, &math->half_unit_t, mp_scaled_type);
  decNumberFromString(math->half_unit_t.data.num, "0.5", &set);
  mp_new_number(mp, &math->three_quarter_unit_t, mp_scaled_type);
```

```
decNumberFromString(math→three_quarter_unit_t.data.num, "0.75", &set);
mp_new_number(mp, &math→zero_t, mp_scaled_type);
decNumberZero(math→zero_t.data.num);   /* fractions */
mp_new_number(mp, &math→arc_tol_k, mp_fraction_type);
{
  decNumber fourzeroninesix;
  decNumberFromInt32(&fourzeroninesix, 4096);
  decNumberDivide(math→arc_tol_k.data.num, &one, &fourzeroninesix, &set);
    /* quit when change in arc length estimate reaches this */
}
mp_new_number(mp, &math→fraction_one_t, mp_fraction_type);
decNumberFromInt32(math→fraction_one_t.data.num, fraction_one);
mp_new_number(mp, &math→fraction_half_t, mp_fraction_type);
decNumberFromInt32(math→fraction_half_t.data.num, fraction_half);
mp_new_number(mp, &math→fraction_three_t, mp_fraction_type);
decNumberFromInt32(math→fraction_three_t.data.num, fraction_three);
mp_new_number(mp, &math→fraction_four_t, mp_fraction_type);
decNumberFromInt32(math→fraction_four_t.data.num, fraction_four);   /* angles */
mp_new_number(mp, &math→three_sixty_deg_t, mp_angle_type);
decNumberFromInt32(math→three_sixty_deg_t.data.num, 360 * angle_multiplier);
mp_new_number(mp, &math→one_eighty_deg_t, mp_angle_type);
decNumberFromInt32(math→one_eighty_deg_t.data.num, 180 * angle_multiplier);
  /* various approximations */
mp_new_number(mp, &math→one_k, mp_scaled_type);
decNumberFromDouble(math→one_k.data.num, 1.0/64);
mp_new_number(mp, &math→sqrt_8_e_k, mp_scaled_type);
{
  decNumberFromDouble(math→sqrt_8_e_k.data.num, 112428.82793/65536.0);
    /* 2^16 √(8/e) ≈ 112428.82793 */
}
mp_new_number(mp, &math→twelve_ln_2_k, mp_fraction_type);
{
  decNumberFromDouble(math→twelve_ln_2_k.data.num, 139548959.6165/65536.0);
    /* 2^24 · 12 ln 2 ≈ 139548959.6165 */
}
mp_new_number(mp, &math→coef_bound_k, mp_fraction_type);
decNumberFromDouble(math→coef_bound_k.data.num, coef_bound);
mp_new_number(mp, &math→coef_bound_minus_1, mp_fraction_type);
decNumberFromDouble(math→coef_bound_minus_1.data.num, coef_bound − 1/65536.0);
mp_new_number(mp, &math→twelvebits_3, mp_scaled_type);
{
  decNumberFromDouble(math→twelvebits_3.data.num, 1365/65536.0);   /* 1365 ≈ 2^12/3 */
}
mp_new_number(mp, &math→twentysixbits_sqrt2_t, mp_fraction_type);
{
  decNumberFromDouble(math→twentysixbits_sqrt2_t.data.num, 94906265.62/65536.0);
    /* 2^26 √2 ≈ 94906265.62 */
}
mp_new_number(mp, &math→twentyeightbits_d_t, mp_fraction_type);
{
  decNumberFromDouble(math→twentyeightbits_d_t.data.num, 35596754.69/65536.0);
    /* 2^28 d ≈ 35596754.69 */
```

```
}
mp_new_number(mp, &math→twentysevenbits_sqrt2_d_t, mp_fraction_type);
{
  decNumberFromDouble(math→twentysevenbits_sqrt2_d_t.data.num, 25170706.63/65536.0);
    /* 2^27 √2 d ≈ 25170706.63 */
}   /* thresholds */
mp_new_number(mp, &math→fraction_threshold_t, mp_fraction_type);
decNumberFromDouble(math→fraction_threshold_t.data.num, fraction_threshold);
mp_new_number(mp, &math→half_fraction_threshold_t, mp_fraction_type);
decNumberFromDouble(math→half_fraction_threshold_t.data.num, half_fraction_threshold);
mp_new_number(mp, &math→scaled_threshold_t, mp_scaled_type);
decNumberFromDouble(math→scaled_threshold_t.data.num, scaled_threshold);
mp_new_number(mp, &math→half_scaled_threshold_t, mp_scaled_type);
decNumberFromDouble(math→half_scaled_threshold_t.data.num, half_scaled_threshold);
mp_new_number(mp, &math→near_zero_angle_t, mp_angle_type);
decNumberFromDouble(math→near_zero_angle_t.data.num, near_zero_angle);
mp_new_number(mp, &math→p_over_v_threshold_t, mp_fraction_type);
decNumberFromDouble(math→p_over_v_threshold_t.data.num, p_over_v_threshold);
mp_new_number(mp, &math→equation_threshold_t, mp_scaled_type);
decNumberFromDouble(math→equation_threshold_t.data.num, equation_threshold);
mp_new_number(mp, &math→tfm_warn_threshold_t, mp_scaled_type);
decNumberFromDouble(math→tfm_warn_threshold_t.data.num, tfm_warn_threshold);
  /* functions */
math→from_int = mp_set_decimal_from_int;
math→from_boolean = mp_set_decimal_from_boolean;
math→from_scaled = mp_set_decimal_from_scaled;
math→from_double = mp_set_decimal_from_double;
math→from_addition = mp_set_decimal_from_addition;
math→from_substraction = mp_set_decimal_from_substraction;
math→from_oftheway = mp_set_decimal_from_of_the_way;
math→from_div = mp_set_decimal_from_div;
math→from_mul = mp_set_decimal_from_mul;
math→from_int_div = mp_set_decimal_from_int_div;
math→from_int_mul = mp_set_decimal_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_decimal_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_boolean = mp_number_to_boolean;
math→to_scaled = mp_number_to_scaled;
math→to_double = mp_number_to_double;
math→to_int = mp_number_to_int;
math→odd = mp_number_odd;
math→equal = mp_number_equal;
```

$math\rightarrow less = mp\_number\_less$;
$math\rightarrow greater = mp\_number\_greater$;
$math\rightarrow nonequalabs = mp\_number\_nonequalabs$;
$math\rightarrow round\_unscaled = mp\_round\_unscaled$;
$math\rightarrow floor\_scaled = mp\_number\_floor$;
$math\rightarrow fraction\_to\_round\_scaled = mp\_decimal\_fraction\_to\_round\_scaled$;
$math\rightarrow make\_scaled = mp\_decimal\_number\_make\_scaled$;
$math\rightarrow make\_fraction = mp\_decimal\_number\_make\_fraction$;
$math\rightarrow take\_fraction = mp\_decimal\_number\_take\_fraction$;
$math\rightarrow take\_scaled = mp\_decimal\_number\_take\_scaled$;
$math\rightarrow velocity = mp\_decimal\_velocity$;
$math\rightarrow n\_arg = mp\_decimal\_n\_arg$;
$math\rightarrow m\_log = mp\_decimal\_m\_log$;
$math\rightarrow m\_exp = mp\_decimal\_m\_exp$;
$math\rightarrow m\_unif\_rand = mp\_decimal\_m\_unif\_rand$;
$math\rightarrow m\_norm\_rand = mp\_decimal\_m\_norm\_rand$;
$math\rightarrow pyth\_add = mp\_decimal\_pyth\_add$;
$math\rightarrow pyth\_sub = mp\_decimal\_pyth\_sub$;
$math\rightarrow fraction\_to\_scaled = mp\_number\_fraction\_to\_scaled$;
$math\rightarrow scaled\_to\_fraction = mp\_number\_scaled\_to\_fraction$;
$math\rightarrow scaled\_to\_angle = mp\_number\_scaled\_to\_angle$;
$math\rightarrow angle\_to\_scaled = mp\_number\_angle\_to\_scaled$;
$math\rightarrow init\_randoms = mp\_init\_randoms$;
$math\rightarrow sin\_cos = mp\_decimal\_sin\_cos$;
$math\rightarrow slow\_add = mp\_decimal\_slow\_add$;
$math\rightarrow sqrt = mp\_decimal\_square\_rt$;
$math\rightarrow print = mp\_decimal\_print\_number$;
$math\rightarrow tostring = mp\_decimal\_number\_tostring$;
$math\rightarrow modulo = mp\_decimal\_number\_modulo$;
$math\rightarrow ab\_vs\_cd = mp\_ab\_vs\_cd$;
$math\rightarrow crossing\_point = mp\_decimal\_crossing\_point$;
$math\rightarrow scan\_numeric = mp\_decimal\_scan\_numeric\_token$;
$math\rightarrow scan\_fractional = mp\_decimal\_scan\_fractional\_token$;
$math\rightarrow free\_math = mp\_free\_decimal\_math$;
$math\rightarrow set\_precision = mp\_decimal\_set\_precision$;
**return** (**void** $*$) $math$; } **void** $mp\_decimal\_set\_precision$(MP $mp$){ **int** $i$; $i = decNumberToInt32$ ( ( $decNumber$ $*$ ) $internal\_value(mp\_number\_precision).data.num, \&set$ ) ;
$set.digits = i$;
$limitedset.digits = i$; } **void** $mp\_free\_decimal\_math$(MP $mp$){ **int** $i$; $free\_number$ ( ( ( $math\_data$ $*$ ) $mp\rightarrow math$ ) $\rightarrow$ $three\_sixty\_deg\_t$ ) ; $free\_number$ ( ( ( $math\_data$ $*$ ) $mp\rightarrow math$ ) $\rightarrow$ $one\_eighty\_deg\_t$ ) ; $free\_number$ ( ( ( $math\_data$ $*$ ) $mp\rightarrow math$ ) $\rightarrow$ $fraction\_one\_t$ ) ; $free\_number$ ( ( ( $math\_data$ $*$ ) $mp\rightarrow math$ ) $\rightarrow$ $zero\_t$ ) ; $free\_number$ ( ( ( $math\_data$ $*$ ) $mp\rightarrow math$ ) $\rightarrow$ $half\_unit\_t$ ) ; $free\_number$ ( ( ( $math\_data$ $*$ ) $mp\rightarrow math$ ) $\rightarrow$ $three\_quarter\_unit\_t$ ) ; $free\_number$ ( ( ( $math\_data$ $*$ ) $mp\rightarrow math$ ) $\rightarrow$ $unity\_t$ ) ; $free\_number$ ( ( ( $math\_data$ $*$ ) $mp\rightarrow math$ ) $\rightarrow$ $two\_t$ ) ; $free\_number$ ( ( ( $math\_data$ $*$ ) $mp\rightarrow math$ ) $\rightarrow$ $three\_t$ ) ; $free\_number$ ( ( ( $math\_data$ $*$ ) $mp\rightarrow math$ ) $\rightarrow$ $one\_third\_inf\_t$ ) ; $free\_number$ ( ( ( $math\_data$ $*$ ) $mp\rightarrow math$ ) $\rightarrow$ $inf\_t$ ) ; $free\_number$ ( ( ( $math\_data$ $*$ ) $mp\rightarrow math$ ) $\rightarrow$ $warning\_limit\_t$ ) ; $free\_number$ ( ( ( $math\_data$ $*$ ) $mp\rightarrow math$ ) $\rightarrow$ $one\_k$ ) ; $free\_number$ ( ( ( $math\_data$ $*$ ) $mp\rightarrow math$ ) $\rightarrow$ $sqrt\_8\_e\_k$ ) ; $free\_number$ ( ( ( $math\_data$ $*$ ) $mp\rightarrow math$ ) $\rightarrow$ $twelve\_ln\_2\_k$ ) ; $free\_number$ ( ( ( $math\_data$ $*$ ) $mp\rightarrow math$ ) $\rightarrow$ $coef\_bound\_k$ ) ; $free\_number$ ( ( ( $math\_data$ $*$ ) $mp\rightarrow math$ ) $\rightarrow$ $coef\_bound\_minus\_1$ ) ; $free\_number$ ( ( ( $math\_data$ $*$ ) $mp\rightarrow math$ ) $\rightarrow$ $fraction\_threshold\_t$ ) ; $free\_number$

```
        ( ( ( math_data * ) mp→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 ) ;
    for (i = 0; i ≤ last_cached_factorial; i++) {
      free(factorials[i]);
    }
    free(factorials);
    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 (decNumber));
  decNumberZero(n→data.num);
  n→type = t;
}
```

**14.**

```
void mp_free_number(MP mp, mp_number *n)
{
  (void) mp;
  free(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_decimal_from_int(mp_number *A, int B)
{
  decNumberFromInt32(A→data.num, B);
}
void mp_set_decimal_from_boolean(mp_number *A, int B)
{
  decNumberFromInt32(A→data.num, B);
}
void mp_set_decimal_from_scaled(mp_number *A, int B)
{
  decNumber c;
  decNumberFromInt32(&c, 65536);
  decNumberFromInt32(A→data.num, B);
  decNumberDivide(A→data.num, A→data.num, &c, &set);
}
void mp_set_decimal_from_double(mp_number *A, double B)
{
  decNumberFromDouble(A→data.num, B);
}
void mp_set_decimal_from_addition(mp_number *A, mp_number B, mp_number C)
{
  decNumberAdd(A→data.num, B.data.num, C.data.num, &set);
}
void mp_set_decimal_from_substraction(mp_number *A, mp_number B, mp_number C)
{
  decNumberSubtract(A→data.num, B.data.num, C.data.num, &set);
}
void mp_set_decimal_from_div(mp_number *A, mp_number B, mp_number C)
{
  decNumberDivide(A→data.num, B.data.num, C.data.num, &set);
}
void mp_set_decimal_from_mul(mp_number *A, mp_number B, mp_number C)
{
  decNumberMultiply(A→data.num, B.data.num, C.data.num, &set);
}
void mp_set_decimal_from_int_div(mp_number *A, mp_number B, int C)
{
  decNumber c;
  decNumberFromInt32(&c, C);
  decNumberDivide(A→data.num, B.data.num, &c, &set);
}
void mp_set_decimal_from_int_mul(mp_number *A, mp_number B, int C)
{
  decNumber c;
  decNumberFromInt32(&c, C);
  decNumberMultiply(A→data.num, B.data.num, &c, &set);
}
void mp_set_decimal_from_of_the_way(MP mp, mp_number *A, mp_number t, mp_number B, mp_number C)
```

```
{
  decNumber c;
  decNumber r1;
  decNumberSubtract(&c, B.data.num, C.data.num, &set);
  mp_decimal_take_fraction(mp, &r1, &c, t.data.num);
  decNumberSubtract(A→data.num, B.data.num, &r1, &set);
  mp_check_decNumber(mp, A→data.num, &set);
}
void mp_number_negate(mp_number * A)
{
  decNumberCopyNegate(A→data.num, A→data.num);
  checkZero(A→data.num);
}
void mp_number_add(mp_number * A, mp_number B)
{
  decNumberAdd(A→data.num, A→data.num, B.data.num, &set);
}
void mp_number_substract(mp_number * A, mp_number B)
{
  decNumberSubtract(A→data.num, A→data.num, B.data.num, &set);
}
void mp_number_half(mp_number * A)
{
  decNumber c;
  decNumberFromInt32(&c, 2);
  decNumberDivide(A→data.num, A→data.num, &c, &set);
}
void mp_number_halfp(mp_number * A)
{
  decNumber c;
  decNumberFromInt32(&c, 2);
  decNumberDivide(A→data.num, A→data.num, &c, &set);
}
void mp_number_double(mp_number * A)
{
  decNumber c;
  decNumberFromInt32(&c, 2);
  decNumberMultiply(A→data.num, A→data.num, &c, &set);
}
void mp_number_add_scaled(mp_number * A, int B)
{     /* also for negative B */
  decNumber b, c;
  decNumberFromInt32(&c, 65536);
  decNumberFromInt32(&b, B);
  decNumberDivide(&b, &b, &c, &set);
  decNumberAdd(A→data.num, A→data.num, &b, &set);
}
void mp_number_multiply_int(mp_number * A, int B)
{
  decNumber b;
```

```
    decNumberFromInt32(&b, B);
    decNumberMultiply(A->data.num, A->data.num, &b, &set);
  }
  void mp_number_divide_int(mp_number *A, int B)
  {
    decNumber b;
    decNumberFromInt32(&b, B);
    decNumberDivide(A->data.num, A->data.num, &b, &set);
  }
  void mp_decimal_abs(mp_number *A)
  {
    decNumberAbs(A->data.num, A->data.num, &set);
  }
  void mp_number_clone(mp_number *A, mp_number B)
  {
    decNumberCopy(A->data.num, B.data.num);
  }
  void mp_number_swap(mp_number *A, mp_number *B)
  {
    decNumber swap_tmp;
    decNumberCopy(&swap_tmp, A->data.num);
    decNumberCopy(A->data.num, B->data.num);
    decNumberCopy(B->data.num, &swap_tmp);
  }
  void mp_number_fraction_to_scaled(mp_number *A)
  {
    A->type = mp_scaled_type;
    decNumberDivide(A->data.num, A->data.num, &fraction_multiplier_decNumber, &set);
  }
  void mp_number_angle_to_scaled(mp_number *A)
  {
    A->type = mp_scaled_type;
    decNumberDivide(A->data.num, A->data.num, &angle_multiplier_decNumber, &set);
  }
  void mp_number_scaled_to_fraction(mp_number *A)
  {
    A->type = mp_fraction_type;
    decNumberMultiply(A->data.num, A->data.num, &fraction_multiplier_decNumber, &set);
  }
  void mp_number_scaled_to_angle(mp_number *A)
  {
    A->type = mp_angle_type;
    decNumberMultiply(A->data.num, A->data.num, &angle_multiplier_decNumber, &set);
  }
```

**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)
{
  int32_t result;
  decNumber corrected;
  decNumberFromInt32(&corrected, 65536);
  decNumberMultiply(&corrected, &corrected, A.data.num, &set);
  decNumberReduce(&corrected, &corrected, &set);
  result = (int) floor(decNumberToDouble(&corrected) + 0.5);
  return result;
}
```

**18.**

```
#define odd(A) (abs(A) % 2 ≡ 1)
  int mp_number_to_int(mp_number A)
  {
    int32_t result;
    set.status = 0;
    result = decNumberToInt32(A.data.num, &set);
    if (set.status ≡ DEC_Invalid_operation) {
      set.status = 0;    /* mp→arith_error = 1; */
      return 0;    /* whatever */
    }
    else {
      return result;
    }
  }
  int mp_number_to_boolean(mp_number A)
  {
    uint32_t result;
    set.status = 0;
    result = decNumberToUInt32(A.data.num, &set);
    if (set.status ≡ DEC_Invalid_operation) {
      set.status = 0;    /* mp→arith_error = 1; */
      return mp_false_code;    /* whatever */
    }
    else {
      return result;
    }
  }
  double mp_number_to_double(mp_number A){ char *buffer = malloc ( ( ( decNumber * ) A.data.num )
        → digits + 14 ) ;
      double res = 0.0;
      assert(buffer);
      decNumberToString(A.data.num, buffer);
      if (sscanf(buffer, "%lf", &res)) {
        free(buffer);
        return res;
      }
      else {
        free(buffer);    /* mp→arith_error = 1; */
        return 0.0;    /* whatever */
      }
      } int mp_number_odd(mp_number A)
      {
        return odd(mp_number_to_int(A));
      }
      int mp_number_equal(mp_number A, mp_number B)
      {
        decNumber res;
        decNumberCompare(&res, A.data.num, B.data.num, &set);
        return decNumberIsZero(&res);
      }
```

```
int mp_number_greater(mp_number A, mp_number B)
{
  decNumber res;
  decNumberCompare(&res, A.data.num, B.data.num, &set);
  return decNumberIsPositive(&res);
}
int mp_number_less(mp_number A, mp_number B)
{
  decNumber res;
  decNumberCompare(&res, A.data.num, B.data.num, &set);
  return decNumberIsNegative(&res);
}
int mp_number_nonequalabs(mp_number A, mp_number B)
{
  decNumber res, a, b;
  decNumberCopyAbs(&a, A.data.num);
  decNumberCopyAbs(&b, B.data.num);
  decNumberCompare(&res, &a, &b, &set);
  return ¬decNumberIsZero(&res);
}
```

**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 '*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_decnumber_tostring(decNumber *n){ decNumber corrected; char *buffer = malloc ( ( (
      decNumber * ) n ) → digits + 14 ) ;
  assert(buffer);
  decNumberCopy(&corrected, n);
  decNumberTrim(&corrected);
  decNumberToString(&corrected, buffer);
  return buffer; } char *mp_decimal_number_tostring(MP mp, mp_number n)
  {
    return mp_decnumber_tostring(n.data.num);
  }
```

**22.** **void** *mp_decimal_print_number*(MP *mp*, *mp_number n*)
```
{
  char *str = mp_decimal_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_decimal_slow_add(MP mp, mp_number *ret, mp_number A, mp_number B)
{
  decNumberAdd(ret→data.num, A.data.num, B.data.num, &set);
}
```

**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 $(2^{28} * p) \operatorname{div} q$, 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_decimal_make_fraction(MP mp, decNumber *ret, decNumber *p, decNumber *q)
{
  decNumberDivide(ret, p, q, &set);
  mp_check_decNumber(mp, ret, &set);
  decNumberMultiply(ret, ret, &fraction_multiplier_decNumber, &set);
}
void mp_decimal_number_make_fraction(MP mp, mp_number *ret, mp_number p, mp_number q)
{
  mp_decimal_make_fraction(mp, ret→data.num, p.data.num, q.data.num);
}
```

**25.** ⟨ Declarations 5 ⟩ +≡
```
void mp_decimal_make_fraction(MP mp, decNumber *ret, decNumber *p, decNumber *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_decimal_take_fraction(MP mp, decNumber *ret, decNumber *p, decNumber *q)
{
  decNumberMultiply(ret, p, q, &set);
  decNumberDivide(ret, ret, &fraction_multiplier_decNumber, &set);
}
void mp_decimal_number_take_fraction(MP mp, mp_number *ret, mp_number p, mp_number q)
{
  mp_decimal_take_fraction(mp, ret→data.num, p.data.num, q.data.num);
}
```

**27.** ⟨ Declarations 5 ⟩ +≡

```
void mp_decimal_take_fraction(MP mp, decNumber *ret, decNumber *p, decNumber *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_decimal_number_take_scaled(MP mp, mp_number *ret, mp_number p_orig, mp_number q_orig)
{
  decNumberMultiply(ret→data.num, p_orig.data.num, q_orig.data.num, &set);
}
```

**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_decimal_number_make_scaled(MP mp, mp_number *ret, mp_number p_orig, mp_number q_orig)
{
  decNumberDivide(ret→data.num, p_orig.data.num, q_orig.data.num, &set);
  mp_check_decNumber(mp, ret→data.num, &set);
}
```

**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→cur_mod_→type = (A)
#define set_cur_mod(A) decNumberCopy ( ( decNumber ∗ ) (mp→cur_mod_→data.n.data.num), &A )
⟨ Declarations 5 ⟩ +≡
  static void mp_wrapup_numeric_token(MP mp, unsigned char ∗start, unsigned char ∗stop);
```

**32.**

```
#define too_precise(a) (a ≡ (DEC_Inexact + DEC_Rounded))
#define too_large(a) (a & DEC_Overflow)
  void mp_wrapup_numeric_token(MP mp, unsigned char ∗start, unsigned char ∗stop){ decNumber result;
      size_t l = stop − start + 1;
      char ∗buf = mp_xmalloc(mp, l + 1, 1);
      buf[l] = '\0';
      (void) strncpy(buf, (const char ∗) start, l);
      set.status = 0;
      decNumberFromString(&result, buf, &set);
      free(buf);
      if (set.status ≡ 0) {
        set_cur_mod(result);
      }
      else if (mp→scanner_status ≠ tex_flushing) {
      if (too_large(set.status)) {
        const char ∗hlp[] = {"I␣could␣not␣handle␣this␣number␣specification",
            "because␣it␣is␣out␣of␣range.", Λ};
        decNumber_check(&result, &set);
        set_cur_mod(result);
        mp_error(mp, "Enormous␣number␣has␣been␣reduced", hlp, false);
      }
      else if (too_precise(set.status)) { set_cur_mod(result); if ( decNumberIsPositive ( ( decNumber ∗ )
          internal_value(mp_warning_check).data.num ) ∧ (mp→scanner_status ≠ tex_flushing) )
      {
        char msg[256];
        const char ∗hlp[] = {"Continue␣and␣I'll␣round␣the␣value␣until␣it␣fits␣the␣current\
            ␣numberprecision", "(Set␣warningcheck:=0␣to␣suppress␣this␣message.)", Λ};
        mp_snprintf(msg, 256, "Number␣is␣too␣precise␣(numberprecision␣=␣%d)", set.digits);
        mp_error(mp, msg, hlp, true);
      }
      }
      else {    /∗ this also captures underflow ∗/
        const char ∗hlp[] = {"I␣could␣not␣handle␣this␣number␣specification", "Error:", "", Λ};
        hlp[2] = decContextStatusToString(&set);
        mp_error(mp, "Erroneous␣number␣specification␣changed␣to␣zero", hlp, false);
        decNumberZero(&result);
        set_cur_mod(result);
      }
      } set_cur_cmd((mp_variable_type)mp_numeric_token); }
```

**33.** **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_decimal_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);
}
```

**34.** We just have to collect bytes.

```
void mp_decimal_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);
}
```

**35.** 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*)

**36.** 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 $\tau$ 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 $\theta$ and ends at an angle $\phi$ 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 $\tau$ 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 $\theta$ and $\phi$ 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_decimal_velocity(MP mp, mp_number *ret, mp_number st, mp_number ct, mp_number sf,
        mp_number cf, mp_number t)
{
  decNumber acc, num, denom;   /* registers for intermediate calculations */
  decNumber r1, r2;
  decNumber arg1, arg2;
  decNumber i16, fone, fhalf, ftwo, sqrtfive;
  decNumberFromInt32(&i16, 16);
  decNumberFromInt32(&fone, fraction_one);
  decNumberFromInt32(&fhalf, fraction_half);
  decNumberFromInt32(&ftwo, fraction_two);
  decNumberFromInt32(&sqrtfive, 5);   /* √5 */
  decNumberSquareRoot(&sqrtfive, &sqrtfive, &set);
  decNumberDivide(&arg1, sf.data.num, &i16, &set);   /* arg1 = sf / 16 */
  decNumberSubtract(&arg1, st.data.num, &arg1, &set);   /* arg1 = st - arg1 */
  decNumberDivide(&arg2, st.data.num, &i16, &set);   /* arg2 = st / 16 */
  decNumberSubtract(&arg2, sf.data.num, &arg2, &set);   /* arg2 = sf - arg2 */
  mp_decimal_take_fraction(mp, &acc, &arg1, &arg2);   /* acc = (arg1 * arg2) / fmul */
  decNumberCopy(&arg1, &acc);
  decNumberSubtract(&arg2, ct.data.num, cf.data.num, &set);   /* arg2 = ct - cf */
  mp_decimal_take_fraction(mp, &acc, &arg1, &arg2);   /* acc = (arg1 * arg2 ) / fmul */
  decNumberSquareRoot(&arg1, &two_decNumber, &set);   /* arg1 = √2 */
  decNumberMultiply(&arg1, &arg1, &fone, &set);   /* arg1 = arg1 * fmul */
  mp_decimal_take_fraction(mp, &r1, &acc, &arg1);   /* r1 = (acc * arg1) / fmul */
  decNumberAdd(&num, &ftwo, &r1, &set);   /* num = ftwo + r1 */
  decNumberSubtract(&arg1, &sqrtfive, &one, &set);   /* arg1 = √5 - 1 */
  decNumberMultiply(&arg1, &arg1, &fhalf, &set);   /* arg1 = arg1 * fmul/2 */
  decNumberMultiply(&arg1, &arg1, &three_decNumber, &set);   /* arg1 = arg1 * 3 */
  decNumberSubtract(&arg2, &three_decNumber, &sqrtfive, &set);   /* arg2 = 3 - √5 */
  decNumberMultiply(&arg2, &arg2, &fhalf, &set);   /* arg2 = arg2 * fmul/2 */
  decNumberMultiply(&arg2, &arg2, &three_decNumber, &set);   /* arg2 = arg2 * 3 */
  mp_decimal_take_fraction(mp, &r1, ct.data.num, &arg1);   /* r1 = (ct * arg1) / fmul */
  mp_decimal_take_fraction(mp, &r2, cf.data.num, &arg2);   /* r2 = (cf * arg2) / fmul */
  decNumberFromInt32(&denom, fraction_three);   /* denom = 3fmul */
  decNumberAdd(&denom, &denom, &r1, &set);   /* denom = denom + r1 */
  decNumberAdd(&denom, &denom, &r2, &set);   /* denom = denom + r1 */
  decNumberCompare(&arg1, t.data.num, &one, &set);
  if (¬decNumberIsZero(&arg1)) {   /* t != r1 */
```

```
      decNumberDivide(&num, &num, t.data.num, &set);     /* num = num / t */
    }
    decNumberCopy(&r2, &num);     /* r2 = num / 4 */
    decNumberDivide(&r2, &r2, &four_decNumber, &set);
    if (decNumberLess(&denom, &r2)) {     /* num/4 ¿= denom =¿ denom ¡ num/4 */
      decNumberFromInt32(ret→data.num, fraction_four);
    }
    else {
      mp_decimal_make_fraction(mp, ret→data.num, &num, &denom);
    }
#if DEBUG
    fprintf(stdout, "\n%f␣=␣velocity(%f,%f,%f,%f,%f)", mp_number_to_double(*ret),
        mp_number_to_double(st), mp_number_to_double(ct), mp_number_to_double(sf),
        mp_number_to_double(cf), mp_number_to_double(t));
#endif
    mp_check_decNumber(mp, ret→data.num, &set);
  }
```

**37.** 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){ decNumber q, r, test;    /* temporary registers */
    decNumber a, b, c, d;
    decNumber ab, cd;
    (void) mp; decNumberCopy (&a, ( decNumber * ) a_orig.data.num ) ; decNumberCopy (&b, (
        decNumber * ) b_orig.data.num ) ; decNumberCopy (&c, ( decNumber * ) c_orig.data.num
        ) ; decNumberCopy (&d, ( decNumber * ) d_orig.data.num ) ; decNumberMultiply (&ab, (
        decNumber * ) a_orig.data.num , ( decNumber * ) b_orig.data.num, &set ) ; decNumberMultiply
        (&cd, ( decNumber * ) c_orig.data.num , ( decNumber * ) d_orig.data.num, &set ) ;
    decNumberCompare(ret→data.num, &ab, &cd, &set);
    mp_check_decNumber(mp, ret→data.num, &set);
    if (1 > 0) return;
    ⟨Reduce to the case that a, c ≥ 0, b, d > 0 38⟩;
    while (1) {
      decNumberDivide(&q, &a, &d, &set);
      decNumberDivide(&r, &c, &b, &set);
      decNumberCompare(&test, &q, &r, &set);
      if (¬decNumberIsZero(&test)) {
        if (decNumberIsPositive(&test)) {
          decNumberCopy(ret→data.num, &one);
        }
        else {
          decNumberCopy(ret→data.num, &minusone);
        }
        goto RETURN;
      }
      decNumberRemainder(&q, &a, &d, &set);
      decNumberRemainder(&r, &c, &b, &set);
      if (decNumberIsZero(&r)) {
        if (decNumberIsZero(&q)) {
          decNumberCopy(ret→data.num, &zero);
        }
        else {
          decNumberCopy(ret→data.num, &one);
        }
        goto RETURN;
      }
      if (decNumberIsZero(&q)) {
        decNumberCopy(ret→data.num, &minusone);
        goto RETURN;
      }
      decNumberCopy(&a, &b);
      decNumberCopy(&b, &q);
      decNumberCopy(&c, &d);
      decNumberCopy(&d, &r);
    }    /* now a > d > 0 and c > b > 0 */
  RETURN:
#if DEBUG
```

```
    fprintf(stdout, "\n%f␣=␣ab_vs_cd(%f,%f,%f,%f)", mp_number_to_double(*ret),
        mp_number_to_double(a_orig), mp_number_to_double(b_orig), mp_number_to_double(c_orig),
        mp_number_to_double(d_orig));
#endif
    mp_check_decNumber(mp, ret→data.num, &set);
    return; }
```

**38.** ⟨ Reduce to the case that $a, c \ge 0$, $b, d > 0$ 38 ⟩ ≡

```
  if (decNumberIsNegative(&a)) {
    decNumberCopyNegate(&a, &a);
    decNumberCopyNegate(&b, &b);
  }
  if (decNumberIsNegative(&c)) {
    decNumberCopyNegate(&c, &c);
    decNumberCopyNegate(&d, &d);
  }
  if (¬decNumberIsPositive(&d)) {
    if (¬decNumberIsNegative(&b)) {
      if ((decNumberIsZero(&a) ∨ decNumberIsZero(&b)) ∧ (decNumberIsZero(&c) ∨ decNumberIsZero(&d)))
        decNumberCopy(ret→data.num, &zero);
      else decNumberCopy(ret→data.num, &one);
      goto RETURN;
    }
    if (decNumberIsZero(&d)) {
      if (decNumberIsZero(&a)) decNumberCopy(ret→data.num, &zero);
      else decNumberCopy(ret→data.num, &minusone);
      goto RETURN;
    }
    decNumberCopy(&q, &a);
    decNumberCopy(&a, &c);
    decNumberCopy(&c, &q);
    decNumberCopyNegate(&q, &b);
    decNumberCopyNegate(&b, &d);
    decNumberCopy(&d, &q);
  }
  else if (¬decNumberIsPositive(&b)) {
    if (decNumberIsNegative(&b) ∧ decNumberIsPositive(&a)) {
      decNumberCopy(ret→data.num, &minusone);
      goto RETURN;
    }
    if (decNumberIsZero(&c)) decNumberCopy(ret→data.num, &zero);
    else decNumberCopy(ret→data.num, &minusone);
    goto RETURN;
  }
```

This code is used in section 37.

**39.** 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
        {
          decNumberCopy(ret→data.num, &fraction_one_plus_decNumber);
          goto RETURN;
        }
#define one_crossing
        {
          decNumberCopy(ret→data.num, &fraction_one_decNumber);
          goto RETURN;
        }
#define zero_crossing
        {
          decNumberCopy(ret→data.num, &zero);
          goto RETURN;
        }

  static void mp_decimal_crossing_point(MP mp, mp_number *ret, mp_number aa, mp_number bb,
          mp_number cc){ decNumber a, b, c;
    double d;     /* recursive counter */
    decNumber x, xx, x0, x1, x2;     /* temporary registers for bisection */
    decNumber scratch, scratch2; decNumberCopy (&a, ( decNumber * ) aa.data.num ) ; decNumberCopy
        (&b, ( decNumber * ) bb.data.num ) ; decNumberCopy (&c, ( decNumber * ) cc.data.num ) ;
    if (decNumberIsNegative(&a)) zero_crossing;
    if (¬decNumberIsNegative(&c)) {
      if (¬decNumberIsNegative(&b)) {
        if (decNumberIsPositive(&c)) {
          no_crossing;
        }
        else if (decNumberIsZero(&a) ∧ decNumberIsZero(&b)) {
          no_crossing;
        }
        else {
          one_crossing;
        }
      }
      if (decNumberIsZero(&a)) zero_crossing;
    }
```

```
    else if (decNumberIsZero(&a)) {
      if (¬decNumberIsPositive(&b)) zero_crossing;
    }    /* Use bisection to find the crossing point... */
    d = epsilonf;
    decNumberCopy(&x0, &a);
    decNumberSubtract(&x1, &a, &b, &set);
    decNumberSubtract(&x2, &b, &c, &set);
      /* not sure why the error correction has to be ¿= 1E-12 */
    decNumberFromDouble(&scratch2, 1 · 10^−12);
    do {
      decNumberAdd(&x, &x1, &x2, &set);
      decNumberDivide(&x, &x, &two_decNumber, &set);
      decNumberAdd(&x, &x, &scratch2, &set);
      decNumberSubtract(&scratch, &x1, &x0, &set);
      if (decNumberGreater(&scratch, &x0)) {
        decNumberCopy(&x2, &x);
        decNumberAdd(&x0, &x0, &x0, &set);
        d += d;
      }
      else {
        decNumberAdd(&xx, &scratch, &x, &set);
        if (decNumberGreater(&xx, &x0)) {
          decNumberCopy(&x2, &x);
          decNumberAdd(&x0, &x0, &x0, &set);
          d += d;
        }
        else {
          decNumberSubtract(&x0, &x0, &xx, &set);
          if (¬decNumberGreater(&x, &x0)) {
            decNumberAdd(&scratch, &x, &x2, &set);
            if (¬decNumberGreater(&scratch, &x0)) no_crossing;
          }
          decNumberCopy(&x1, &x);
          d = d + d + epsilonf;
        }
      }
    } while (d < fraction_one);
    decNumberFromDouble(&scratch, d);
    decNumberSubtract(ret→data.num, &scratch, &fraction_one_decNumber, &set);
  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
    mp_check_decNumber(mp, ret→data.num, &set);
    return; }
```

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

**41.** *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;
}
```

**42.** *number_floor* floors a number

```
void mp_number_floor(mp_number *i)
{
  int round = set.round;
  set.round = DEC_ROUND_FLOOR;
  decNumberToIntegralValue(i→data.num, i→data.num, &set);
  set.round = round;
}
```

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

```
void mp_decimal_fraction_to_round_scaled(mp_number *x_orig)
{
  x_orig→type = mp_scaled_type;
  decNumberDivide(x_orig→data.num, x_orig→data.num, &fraction_multiplier_decNumber, &set);
}
```

**44. 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.

**45.**

```
void mp_decimal_square_rt(MP mp, mp_number *ret, mp_number x_orig)
{    /* return, x: scaled */
  decNumber x;
  decNumberCopy(&x, x_orig.data.num);
  if (¬decNumberIsPositive(&x)) {
    ⟨Handle square root of zero or negative argument 46⟩;
  }
  else {
    decNumberSquareRoot(ret→data.num, &x, &set);
  }
  mp_check_decNumber(mp, ret→data.num, &set);
}
```

**46.** ⟨Handle square root of zero or negative argument 46⟩ ≡

```
{
  if (decNumberIsNegative(&x)) {
    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_decimal_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);
  }
  decNumberZero(ret→data.num);
  return;
}
```

This code is used in section 45.

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

```
void mp_decimal_pyth_add(MP mp, mp_number *ret, mp_number a_orig, mp_number b_orig)
{
  decNumber a, b;
  decNumber asq, bsq;
  decNumberCopyAbs(&a, a_orig.data.num);
  decNumberCopyAbs(&b, b_orig.data.num);
  decNumberMultiply(&asq, &a, &a, &set);
  decNumberMultiply(&bsq, &b, &b, &set);
  decNumberAdd(&a, &asq, &bsq, &set);
  decNumberSquareRoot(ret→data.num, &a, &set);    /* if (set.status ≠ 0) { */
    /* mp→arith_error = true; */    /* decNumberCopy(ret→data.num, &EL_GORDO_decNumber); */
    /* } */
  mp_check_decNumber(mp, ret→data.num, &set);
}
```

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

```
void mp_decimal_pyth_sub(MP mp, mp_number *ret, mp_number a_orig, mp_number b_orig)
{
  decNumber a, b;
  decNumberCopyAbs(&a, a_orig.data.num);
  decNumberCopyAbs(&b, b_orig.data.num);
  if (¬decNumberGreater(&a, &b)) {
    ⟨Handle erroneous pyth_sub and set a: = 0 49⟩;
  }
  else {
    decNumber asq, bsq;
    decNumberMultiply(&asq, &a, &a, &set);
    decNumberMultiply(&bsq, &b, &b, &set);
    decNumberSubtract(&a, &asq, &bsq, &set);
    decNumberSquareRoot(&a, &a, &set);
  }
  decNumberCopy(ret→data.num, &a);
  mp_check_decNumber(mp, ret→data.num, &set);
}
```

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

```
{
  if (decNumberLess(&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_decimal_number_tostring(mp, a_orig);
    char *bstr = mp_decimal_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);
  }
  decNumberZero(&a);
}
```

This code is used in section 48.

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

```
void mp_decimal_m_log(MP mp, mp_number *ret, mp_number x_orig){ if ( ¬decNumberIsPositive ( (
        decNumber * ) x_orig.data.num ) )
    {
      ⟨ Handle non-positive logarithm 51 ⟩;
    }
    else {
      decNumber twofivesix;
      decNumberFromInt32(&twofivesix, 256);
      decNumberLn(ret→data.num, x_orig.data.num, &limitedset);
      mp_check_decNumber(mp, ret→data.num, &limitedset);
      decNumberMultiply(ret→data.num, ret→data.num, &twofivesix, &set);
    }
    mp_check_decNumber(mp, ret→data.num, &set); }
```

**51.** ⟨ Handle non-positive logarithm 51 ⟩ ≡

```
{
  char msg[256];
  const char *hlp[] = {"Since I don't take logs of non-positive numbers,",
      "I'm zeroing this one. Proceed, with fingers crossed.", Λ};
  char *xstr = mp_decimal_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);
  decNumberZero(ret→data.num);
}
```

This code is used in section 50.

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

```
void mp_decimal_m_exp(MP mp, mp_number *ret, mp_number x_orig){ decNumber temp, twofivesix;
    decNumberFromInt32(&twofivesix, 256);
    decNumberDivide(&temp, x_orig.data.num, &twofivesix, &set);
    limitedset.status = 0;
    decNumberExp(ret→data.num, &temp, &limitedset); if (limitedset.status & DEC_Clamped) { if (
        decNumberIsPositive ( ( decNumber * ) x_orig.data.num ) )
    {
      mp→arith_error = true;
      decNumberCopy(ret→data.num, &EL_GORDO_decNumber);
    }
    else {
      decNumberZero(ret→data.num);
    }
    } mp_check_decNumber(mp, ret→data.num, &limitedset);
    limitedset.status = 0; }
```

**53.** 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_decimal_n_arg(MP mp, mp_number *ret, mp_number x_orig, mp_number y_orig){ if (
        decNumberIsZero ( ( decNumber * ) x_orig.data.num ) ∧ decNumberIsZero ( ( decNumber * )
        y_orig.data.num ) )
    {
      ⟨Handle undefined arg 54⟩;
    }
    else {
      decNumber atan2val, oneeighty_angle;
      ret→type = mp_angle_type;
      decNumberFromInt32(&oneeighty_angle, 180 * angle_multiplier);
      decNumberDivide(&oneeighty_angle, &oneeighty_angle, &PI_decNumber, &set);
      checkZero(y_orig.data.num);
      checkZero(x_orig.data.num);
      decNumberAtan2(&atan2val, y_orig.data.num, x_orig.data.num, &set);
#if DEBUG
      fprintf(stdout, "\n%g␣=␣atan2(%g,%g)", decNumberToDouble(&atan2val),
          mp_number_to_double(x_orig), mp_number_to_double(y_orig));
#endif
      decNumberMultiply(ret→data.num, &atan2val, &oneeighty_angle, &set);
      checkZero(ret→data.num);
#if DEBUG
      fprintf(stdout, "\nn_arg(%g,%g,%g)", mp_number_to_double(*ret), mp_number_to_double(x_orig),
          mp_number_to_double(y_orig));
#endif
    }
    mp_check_decNumber(mp, ret→data.num, &set); }
```

**54.** ⟨Handle undefined arg 54⟩ ≡

```
  {
    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);
    ;
    decNumberZero(ret→data.num);
  }
```

This code is used in section 53.

**55.** 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*.

First, we need a decNumber function that calculates sines and cosines using the Taylor series. This function is fairly optimized.

```
#define PRECALC_FACTORIALS_CACHESIZE 50
  static void sinecosine(decNumber *theangle, decNumber *c, decNumber *s)
  {
    int n, i, prec;
    decNumber p, pxa, fac, cc;
    decNumber n1, n2, p1;
    decNumberZero(c);
    decNumberZero(s);
    prec = (set.digits/2);
    if (prec < DECPRECISION_DEFAULT) prec = DECPRECISION_DEFAULT;
    for (n = 0; n < prec; n++) {
      decNumberFromInt32(&p1, n);
      decNumberFromInt32(&n1, 2 * n);
      decNumberPower(&p, &minusone, &p1, &limitedset);
      if (n ≡ 0) {
        decNumberCopy(&pxa, &one);
      }
      else {
        decNumberPower(&pxa, theangle, &n1, &limitedset);
      }
      if (2 * n < last_cached_factorial) {
        decNumberCopy(&fac, factorials[2 * n]);
      }
      else {
        decNumberCopy(&fac, factorials[last_cached_factorial]);
        for (i = last_cached_factorial + 1; i ≤ 2 * n; i++) {
          decNumberFromInt32(&cc, i);
          decNumberMultiply(&fac, &fac, &cc, &set);
          if (i < PRECALC_FACTORIALS_CACHESIZE) {
            factorials[i] = malloc(sizeof (decNumber));
            decNumberCopy(factorials[i], &fac);
            last_cached_factorial = i;
          }
        }
      }
      decNumberDivide(&pxa, &pxa, &fac, &set);
      decNumberMultiply(&pxa, &pxa, &p, &set);
      decNumberAdd(s, s, &pxa, &set);
      decNumberFromInt32(&n2, 2 * n + 1);
      decNumberMultiply(&fac, &fac, &n2, &set);   /* fac = fac * (2*n+1) */
      decNumberPower(&pxa, theangle, &n2, &limitedset);
      decNumberDivide(&pxa, &pxa, &fac, &set);
      decNumberMultiply(&pxa, &pxa, &p, &set);
      decNumberAdd(c, c, &pxa, &set);
        /* printf("\niteration␣%2d:␣%-42s␣%-42s", n, tostring(c), tostring(s)); */
    }
  }
```

**56.** Calculate sines and cosines.

```
void mp_decimal_sin_cos(MP mp, mp_number z_orig, mp_number *n_cos, mp_number *n_sin)
{
  decNumber rad;
  double tmp;
  decNumber one_eighty;
  tmp = mp_number_to_double(z_orig)/16.0;
#if DEBUG
  fprintf(stdout, "\nsin_cos(%f)", mp_number_to_double(z_orig));
#endif
#if 0
  if (decNumberIsNegative(&rad)) {
    while (decNumberLess(&rad, &PI_decNumber))
      decNumberAdd(&rad, &rad, &PI_decNumber, &set);
  }
  else {
    while (decNumberGreater(&rad, &PI_decNumber))
      decNumberSubtract(&rad, &rad, &PI_decNumber, &set);
  }
#endif
  if ((tmp ≡ 90.0) ∨ (tmp ≡ −270)) {
    decNumberZero(n_cos→data.num);
    decNumberCopy(n_sin→data.num, &fraction_multiplier_decNumber);
  }
  else if ((tmp ≡ −90.0) ∨ (tmp ≡ 270.0)) {
    decNumberZero(n_cos→data.num);
    decNumberCopyNegate(n_sin→data.num, &fraction_multiplier_decNumber);
  }
  else if ((tmp ≡ 180.0) ∨ (tmp ≡ −180.0)) {
    decNumberCopyNegate(n_cos→data.num, &fraction_multiplier_decNumber);
    decNumberZero(n_sin→data.num);
  }
  else {
    decNumberFromInt32(&one_eighty, 180 * 16);
    decNumberMultiply(&rad, z_orig.data.num, &PI_decNumber, &set);
    decNumberDivide(&rad, &rad, &one_eighty, &set);
    sinecosine(&rad, n_sin→data.num, n_cos→data.num);
    decNumberMultiply(n_cos→data.num, n_cos→data.num, &fraction_multiplier_decNumber, &set);
    decNumberMultiply(n_sin→data.num, n_sin→data.num, &fraction_multiplier_decNumber, &set);
  }
#if DEBUG
  fprintf(stdout, "\nsin_cos(%f,%f,%f)", decNumberToDouble(&rad), mp_number_to_double(*n_cos),
      mp_number_to_double(*n_sin));
#endif
  mp_check_decNumber(mp, n_cos→data.num, &set);
  mp_check_decNumber(mp, n_sin→data.num, &set);
}
```

**57.** 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];
  }
```

**58.** 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;
      decNumberFromInt32(mp→randoms[(i * 21) % 55].data.num, j);
    }
    mp_new_randoms(mp);
    mp_new_randoms(mp);
    mp_new_randoms(mp);    /* "warm up" the array */
    ran_start((unsigned long) seed);
  }
```

**59.** 
```
  void mp_decimal_number_modulo(mp_number *a, mp_number b)
  {
    decNumberRemainder(a→data.num, a→data.num, b.data.num, &set);
  }
```

**60.** 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)
{
   decNumber a;
   decNumber b;
   unsigned long int op;
   (void) mp;
   op = (unsigned) ran_arr_next();
   decNumberFromInt32(&a, op);
   decNumberFromInt32(&b, MM);
   decNumberDivide(&a, &a, &b, &set);     /* a = a/b */
   decNumberCopy(ret→data.num, &a);
   mp_check_decNumber(mp, ret→data.num, &set);
}
```

**61.** 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]);
}
```

**62.** 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_decimal_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_decimal_abs(&abs_x);
   mp_next_unif_random(mp, &u);
   decNumberMultiply(y.data.num, abs_x.data.num, u.data.num, &set);
   free_number(u); if (mp_number_equal(y, abs_x)) { mp_number_clone (ret, ( ( math_data * ) mp->math
      ) -> zero_t ) ; } else if ( mp_number_greater (x, ( ( math_data * ) mp->math ) -> zero_t ) )
   {
      mp_number_clone(ret, y);
   }
   else {
      mp_number_clone(ret, y);
      mp_number_negate(ret);
   }
   free_number(abs_x);
   free_number(x);
   free_number(y); }
```

**63.** 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_decimal_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_decimal_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_decimal_abs(&abs_x); }
    while (¬mp_number_less(abs_x, u)) ;
    mp_decimal_number_make_fraction(mp, &r, xa, u);
    mp_number_clone(&xa, r);
    mp_decimal_m_log(mp, &la, u); mp_set_decimal_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); }
```

**64.** The following subroutine could be used 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. This is not necessary, even if it's shorter than the current *ab_vs_cd* and looks as a native implementation.

```
    /* void mp_decimal_ab_vs_cd(MP mp, mp_number *
    ret, mp_number a_orig, mp_number b_orig, mp_number c_orig, mp_number d_orig) {
    decNumber a, b, c, d; decNumber ab, cd; (void) mp;   decNumberCopy (&a, ( decNumber * )
    a_orig.data.num ) ; decNumberCopy (&b, ( decNumber * ) b_orig.data.num ) ; decNumberCopy
    (&c, ( decNumber * ) c_orig.data.num ) ; decNumberCopy (&d, ( decNumber * ) d_orig.data.num
    ) ;    decNumberMultiply (&ab, ( decNumber * ) a_orig.data.num , ( decNumber * )
    b_orig.data.num, &set ) ; decNumberMultiply (&cd, ( decNumber * ) c_orig.data.num , (
    decNumber * ) d_orig.data.num, &set ) ; decNumberCompare(ret→data.num, &ab, &cd, &set);
    mp_check_decNumber(mp, ret→data.num, &set); return;  } */
```

# Math support functions for decNumber based math