A proof of GMP fast division and square root implementations

Paul Zimmermann
INRIA Lorraine and LORIA
zimmerma@loria.fr

1 Division

The low-level division interface routine of GNU MP 3.1 [2] is mpn_tdiv_qr:

```
mp_limb_t mpn_tdiv_qr (mp_limb_t *qp, mp_limb_t *rp, mp_size_t qxn,
                       const mp_limb_t *np, mp_size_t nn, const mp_limb_t *dp, mp_size_t dn)
```

which divides {np,nn} by {dp,dn}, writes the quotient at {qp, nn-dn+1} and the remainder at {rp,dn}. It is required that nn is greater than or equal to dn. The qxn operand must be zero. The quotient is rounded towards zero, and no overlap between arguments is permitted.

When both $nn - dn$ and dn are large, mpn_tdiv_qr calls the internal function mpn_bz_divrem_n which performs fast division. As mpn_bz_divrem_n only works for $nn = 2dn$, one has to split the dividend in smaller pieces.

The fast division algorithm used in GMP is due to Moenck and Borodin, Jebelean, and Burnikel and Ziegler. Moenck and Borodin discovered the idea of the 2/1 and 3/2 divisions in 1972 [4], but expressed it only in the case of $O(n^{1+\varepsilon})$ multiplication (for example using FFT), in which case it has complexity $O(M(n)\log n)$ whereas Newton-based methods require $O(M(n))$ operations only. (We thank Dan Bernstein for pointing us the paper from Moenck and Borodin.) Then in 1997 Jebelean expressed the same algorithm in the Karatsuba case, for which it gives only a constant multiplicative overhead with respect to multiplication [3]. Finally in 1998 Burnikel and Ziegler worked out the details for a practical implementation [1]. The author was first aware of the last paper, that is why it first called the algorithm "Burnikel-Ziegler" algorithm. It would be more correct to call it the "Moenck-Borodin-Jebelean-Burnikel-Ziegler" algorithm.

The fast division implementation consists of two functions:

```
mp_limb_t mpn_bz_divrem_n (mp_limb_t *qp, mp_limb_t *np, mp_limb_t *dp, mp_size_t n)
```

This function divides {np,2n} by {dp,n}, puts the n least significant limbs of the quotient in {qp,n}, and the remainder in {np,n}. It returns the most significant limb of the quotient, which is 0 or 1. It is required that the most significant bit of the divisor is set.

```
mp_limb_t mpn_bz_div_3_halves_by_2 (mp_limb_t *qp, mp_limb_t *np, mp_limb_t *dp, mp_size_t n)
```

This function divides {np,3n} by {dp,2n}, puts the n least significant limbs of the quotient in {qp,n}, and the remainder in {np,2n}. It returns the most significant limb of the quotient, which is 0 or 1. It is required that the most significant bit of the divisor is set.

1.1 Proof of mpn_bz_divrem_n

Let b be the number of bits in a limb (usually 32 or 64, represented in GMP by the macro BITS_PER_MP_LIMB), and $\beta = 2^b$. We denote by N the dividend, i.e. the value of {np,2n}, and by D the divisor, i.e. the value of {dp,n}.

The inputs of mpn_bz_divrem_n satisfy $0 \le N < \beta^{2n}$ and $\beta^n/2 \le D < \beta^n$, so $0 \le \lfloor N/D \rfloor < 2\beta^n$.

Even case. In the easiest case where n is even, `mpn_bz_divrem_n` proceeds as follows:

```
mp_size_t n2 = n/2;
qhl = mpn_bz_div_3_halves_by_2 (qp + n2, np + n2, dp, n2);
qhl += mpn_add_1 (qp + n2, qp + n2, n2,
                  mpn_bz_div_3_halves_by_2 (qp, np, dp, n2));
```

After the first call to `mpn_bz_div_3_halves_by_2`, we have $\lfloor N/\beta^{n2}\rfloor = (qhl\,\beta^{n2} + Q_h)D + N'$ where Q_h represents the value of `{qp+n2,n2}` and N' represents the value of `{np+n2,n}` after the call. Thus if N'' represents the value of `{np,n+n2}` we have $N = (qhl\,\beta^{n2} + Q_h)D\beta^{n2} + N''$. The second call divides N'' by D, and gives $N'' = (c\beta^{n2} + Q_l)D + R$ where c is the output of the second call to `mpn_bz_div_3_halves_by_2`, Q_l is the value of `{qp,n2}` after that call, and R is the new value of `{np,n2}`. Thus we have

$$N = (qhl\,\beta^{2n2} + (Q_h + c)\beta^{n2} + Q_l)D + R$$

where `qhl` is still the value returned by the first call; the `mpn_add_1` call propagates the carry `c` from position `qp+n2` over a length of `n2`, adding the resulting carry out to `qhl`, which cannot exceed 1, since $N/D < 2\beta^n$.

Odd case. In the case where the size n is odd, we first call recursively `mpn_bz_divrem_n` on the upper $2(n-1)$ limbs of the dividend with the upper $n-1$ limbs of the divisor:

```
qhl = mpn_bz_divrem_n (qp + 1, np + 2, dp + 1, n - 1);
```

After that call, we have

$$\lfloor \frac{N}{\beta^2} \rfloor = (qhl\,\beta^{n-1} + Q_h)\lfloor \frac{D}{\beta} \rfloor + N'$$

where Q_h and N' represent `{qp+1,n-1}` and `{np+2,n-1}` respectively. Multiplying by β^2 we get:

$$N = (qhl\,\beta^n + \beta Q_h)(D - dp[0]) + N'\beta^2 + np[1]\beta + np[0].$$

Then we have to correct the remainder by subtracting the product of $dp[0]$ (which was not considered in the `mpn_bz_divrem_n` call) by $qhl\,\beta^n + \beta Q_h$:

```
cc = mpn_submul_1 (np + 1, qp + 1, n - 1, dp[0]);
cc = mpn_sub_1 (np + n, np + n, 1, cc);
if (qhl) cc += mpn_sub_1 (np + n, np + n, 1, dp[0]);
```

After the `mpn_submul_1` call, we have

$$N'\beta + np[1] - Q_h dp[0] = (np[n] - cc)\beta^{n-1} + N''$$

where N'' represents the new value of `{np+1,n-1}` (note that $np[n]$ was unchanged). Combining the two above equations gives:

$$N = (qhl\,\beta^n + \beta Q_h)D + (np[n] - cc - qhl\,dp[0])\beta^n + N''\beta + np[0]. \quad (1)$$

After the `mpn_sub_1` call, we have

$$np[n] - cc = -cc\,\beta + np[n] \quad (2)$$

where the left (resp. right) hand side represents the value before (resp. after) the instruction. Finally, the third line subtracts $dp[0]$ from $np[n]$ in case qhl is set:

$$-cc\,\beta + np[n] - qhl\,dp[0] = -cc\,\beta + np[n] \quad (3)$$

with the same convention than above. Adding equations (1), (2) and (3) gives:

$$N = (qhl\,\beta^n + \beta Q_h)D - cc\beta^{n+1} + np[n]\beta^n + N''\beta + np[0]$$

and we have $0 \le cc \le 2$, since each `mpn_sub_1` call returns 0 or 1.

Now in case cc is not zero (i.e. the estimated quotient was too large), we update the quotient until $cc = 0$:

```
while (cc) /* quotient too large */
  {
    qhl -= mpn_sub_1 (qp + 1, qp + 1, n - 1, (mp_limb_t) 1);
    cc -= mpn_add_n (np + 1, np + 1, dp, n);
  }
```

Each loop subtracts one to the upper part of the quotient `{qp+1,n-1}` (and possibly from the quotient carry `qhl`) and correspondingly adds the divisor to `{np+1,n}`, so at the end we have:

$$N = (qhl\, \beta^n + \beta Q'_h)D + N'''$$

where Q'_h represents `{qp+1,n-1}` and N''' represents `{np,n+1}`. The number of loops is at most 4 since $c \le 2$ and $2D \ge \beta^n$, thus two loops add at least β^n to `{np+1,n}`, giving at least one carry out.

The final instructions divide N''' by D, store the corresponding quotient in $qp[0]$, add the potential quotient carry to Q'_h and `qhl`, and store the remainder in `{np,n}`:

```
qhl += mpn_add_1 (qp + 1, qp + 1, n - 1,
                  mpn_sb_divrem_mn (qp, np, n + 1, dp, n));
```

Thus at the end we have as desired:

$$N = (qhl\, \beta^n + \{qp, n\})D + \{np, n\}.$$

1.2 Proof of mpn_bz_div_3_halves_by_2

Let $N = \{np, 3n\}$ the dividend and $D = \{dp, 2n\}$ the divisor. We decompose N in $N_{2,1}\beta^n + N_0$ where $N_{2,1} = \{np + n, 2n\}$ and $N_0 = \{np, n\}$. Similarly for the divisor $D = D_1\beta^n + D_0$ where $D_1 = \{dp + n, n\}$ and $D_0 = \{dp, n\}$.

The first instruction divides $N_{2,1}$ by D_1, either using `mpn_bz_divrem_n` or `mpn_sb_divrem_mn` of cost $O(n^2)$ if n is smaller than some threshold:

```
if (n < BZ_THRESHOLD)
  qhl = mpn_sb_divrem_mn (qp, np + n, twon, dp + n, n);
else
  qhl = mpn_bz_divrem_n (qp, np + n, dp + n, n);
```

so that we have $N_{2,1} = (qhl\beta^n + \{qp, n\})D_1 + \{np + n, n\}$ therefore

$$N = (qhl\beta^n + \{qp, n\})(D - D_0) + \{np, 2n\}.$$

We now have to subtract $(qhl\beta^n + \{qp, n\})D_0$ from $\{np, 2n\}$:

```
mp_size_t twon = n + n;
tmp = (mp_ptr) TMP_ALLOC (twon * BYTES_PER_MP_LIMB);
mpn_mul_n (tmp, qp, dp, n);
cc = mpn_sub_n (np, np, tmp, twon);
TMP_FREE (marker);
if (qhl) cc += mpn_sub_n(np+n, np+n, dp, n);
```

After the `mpn_mul_n`, $\{tmp, 2n\}$ contains $\{qp, n\}D_0$. The first `mpn_sub_n` call subtracts it from $\{np, 2n\}$, and the last one subtracts $qhl\, D_0$ from $\{np + n, n\}$, giving:

$$N = (qhl\, \beta^n + \{qp, n\})D - cc\, \beta^{2n} + \{np, 2n\}. \tag{4}$$

Finally if cc is not zero, the estimated quotient was too large so we update it with the following loop, which is performed at most 4 times since $cc \le 2$ and $2D \ge \beta^{2n}$:

```
  while (cc)
    {
      qhl -= mpn_sub_1 (qp, qp, n, (mp_limb_t) 1);
      cc -= mpn_add_n (np, np, dp, twon);
    }
```

This loops keeps invariant the right-hand side of (4), so when $cc = 0$ we have as desired:

$$N = (qhl\beta^n + \{qp, n\})D + \{np, 2n\}.$$

1.3 The complete code

```
mp_limb_t mpn_bz_divrem_n (mp_ptr qp, mp_ptr np, mp_srcptr dp, mp_size_t n)
{
  mp_limb_t qhl, cc;

  if (n % 2 != 0)
    {
      qhl = mpn_bz_divrem_n (qp + 1, np + 2, dp + 1, n - 1);
      cc = mpn_submul_1 (np + 1, qp + 1, n - 1, dp[0]);
      cc = mpn_sub_1 (np + n, np + n, 1, cc);
      if (qhl) cc += mpn_sub_1 (np + n, np + n, 1, dp[0]);
      while (cc)
        {
          qhl -= mpn_sub_1 (qp + 1, qp + 1, n - 1, (mp_limb_t) 1);
          cc -= mpn_add_n (np + 1, np + 1, dp, n);
        }
      qhl += mpn_add_1 (qp + 1, qp + 1, n - 1, mpn_sb_divrem_mn (qp, np, n + 1, dp, n));
    }
  else
    {
      mp_size_t n2 = n/2;
      qhl = mpn_bz_div_3_halves_by_2 (qp + n2, np + n2, dp, n2);
      qhl += mpn_add_1 (qp + n2, qp + n2, n2, mpn_bz_div_3_halves_by_2 (qp, np, dp, n2));
    }
  return qhl;
}

mp_limb_t mpn_bz_div_3_halves_by_2 (mp_ptr qp, mp_ptr np, mp_srcptr dp, mp_size_t n)
{
  mp_size_t twon = n + n;
  mp_limb_t qhl, cc;
  mp_ptr tmp;
  TMP_DECL (marker);

  TMP_MARK (marker);
  if (n < BZ_THRESHOLD)
    qhl = mpn_sb_divrem_mn (qp, np + n, twon, dp + n, n);
  else
    qhl = mpn_bz_divrem_n (qp, np + n, dp + n, n);
  tmp = (mp_ptr) TMP_ALLOC (twon * BYTES_PER_MP_LIMB);
  mpn_mul_n (tmp, qp, dp, n);
  cc = mpn_sub_n (np, np, tmp, twon);
  TMP_FREE (marker);
  if (qhl) cc += mpn_sub_n (np + n, np + n, dp, n);
  while (cc)
    {
      qhl -= mpn_sub_1 (qp, qp, n, (mp_limb_t) 1);
```

```
      cc -= mpn_add_n (np, np, dp, twon);
    }
  return qhl;
}
```

1.4 Final remarks

Dealing with the carry out qhl is necessary since even if the initial arguments are normalized, i.e. $N < D\beta^n$, this is not guaranteed for the recursive calls. For example in base 10 when dividing 4523 by 46, one will divide 452 by 46, which will in turn divide 45 by 4, giving as quotient $Q = 11$. This will happen rarely yet. However if we first normalize the initial arguments, the carries cc in the algorithm will be smaller, and as shown by Burnikel and Ziegler the `while`-loops will be performed at most twice each in the worst case (instead of four times). Our implementation doesn't normalize the initial arguments since this better fits with the GMP division semantics, and this avoids a third function for the initial call.

For a PIII-500 under Linux, the threshold with the naïve division is about at 100 limbs. This algorithm is not asymptotically optimal since with FFT it has a complexity of $O(M(n) \log n)$, so for really huge numbers we should use a Newton-like method of cost $O(M(n))$.

When the dividend N has a size nn smaller than twice that dn of the divisor D, but still $nn - dn$ is large, then one should use a variant of `mpn_bz_div_3_halves_by_2` which first divides the upper $2(nn-dn)$ limbs of N by the upper $nn-dn$ limbs of D, and then adjust the estimated quotient as in `mpn_bz_div_3_halves_by_2`.

2 Square root

The square root implementation consists of the following functions:

`mp_size_t mpn_sqrtrem1 (mp_ptr sp, mp_ptr rp, mp_srcptr np)`

> This function writes in `sp[0]` the square root of `np[0]` and puts the remainder at address `rp` if this pointer is not NULL. It returns the size of the remainder, which is either 0 or 1. The input should be normalized, i.e. $np[0] \geq \beta/4$ where $\beta = 2^b$ and b is the number of bits in a limb (`BITS_PER_MP_LIMB`). `rp` might be identical to `sp`.

`mp_limb_t mpn_sqrtrem2 (mp_ptr sp, mp_ptr rp, mp_srcptr np)`

> This function writes in `sp[0]` the square root of `{np, 2}`, which should be normalized, and puts the low significant limb of the remainder in `rp[0]`. It returns the most significant limb of the remainder, which is 0 or 1. `rp` should not be NULL, but might be identical to `sp`.

`mp_limb_t mpn_dq_sqrtrem (mp_ptr sp, mp_ptr np, mp_size_t n)`

> This function writes in `{sp, n}` the square root of `{np, 2n}`, which should be normalized, and puts the n low significant limbs of the remainder in `{rp, n}`. It returns the possible carry out of the remainder, which is 0 or 1.

`mp_size_t mpn_sqrtrem (mp_ptr sp, mp_ptr rp, mp_srcptr np, mp_size_t n)`

> This is the main function. It writes in `{sp, ceil(n/2)}` the square root of `{np, n}`, which does not need to be normalized, and puts the remainder at address `rp`, if this pointer is not NULL. It returns the numbers of limbs of the remainder. `rp` might be identical to `np`.

2.1 Proof of `mpn_sqrtrem1`

We first compute a 8-bit approximation from the high 8-bits of `np[0]`:

```
np0 = np[0];
q = np0 >> (BITS_PER_MP_LIMB - 8);
s = approx_tab[q - 64]; /* 128 <= s < 255 */
```

As the input is normalized, we have $\beta/4 \leq np[0] < \beta$ and thus $2^6 \leq q < 2^8$. The table approx_tab, which contains 192 entries, is such that $T[q-64] = \lfloor\sqrt{2^8 q}\rfloor$. We thus have $2^7 \leq s < 2^8$ and $s^2 \leq 2^8 q < (s+1)^2$.

```
r = (np0 >> (BITS_PER_MP_LIMB - 16)) - s * s;
if (r > 2 * s) { r -= 2 * s + 1; s++; }
```

Let x be `np0 >> (BITS_PER_MP_LIMB - 16)`: x represents the 16 most significant bits of np0. The above two lines result in $x = s^2 + r$ with $0 \leq r \leq 2s$. Since $x < 2^8 q + 2^8$ and $s \geq 2^7$ we have $(s+2)^2 > (s+1)^2 + 2s > 2^8 q + 2^8 > x$, whence if the if-statement is true, necessary after the instructions `r -= 2 * s + 1` and `s++` it becomes false, and we have $0 \leq r \leq 2s$, with s being a 8-bit approximation of the input square root.

The following while-loop increases the accuracy of the approximation $s^2 + r$ up to the number of bits per word BITS_PER_MP_LIMB. The loop invariants are as follows: the value of the $2prec$ most significant bits of $np[0]$ equals $s^2 + r$, where s has exactly $prec$ bits, $0 \leq r \leq 2s$, and the most significant bits of the variable np0 are the next bits not yet considered.

```
prec = 8;
np0 <<= 2 * prec;
while (2*prec < BITS_PER_MP_LIMB) {
  r = (r << prec) + (np0 >> (BITS_PER_MP_LIMB - prec));
  np0 <<= prec;
  u = 2 * s;
  q = r / u;
  u = r - q * u;
  s = (s << prec) + q;
  u = (u << prec) + (np0 >> (BITS_PER_MP_LIMB - prec));
  q = q * q;
  r = u - q;
  if (u < q) {
    r += 2 * s - 1;
    s --;
  }
  np0 <<= prec;
  prec = 2 * prec;
}
```

This while-loop implements algorithm SqrtRem of [5]. We just need to check that no overflow occurs: if p is the current value of prec, then initially $2^{p-1} \leq s < 2^p$ and $r \leq 2s$ After the first instruction of the loop, $r < 2^p(2s) + 2^p$, and after the fourth, $q < \lfloor 2^p + 2^p/(2s)\rfloor$ thus $q \leq 2^p$, and the remainder u is less than 2^{p+1}. The next value of s is at most 2^{2p} and $u < 2^{2p+1}$. After `q = q * q`, we have $q \leq 2^{2p}$, and the following values are smaller. Thus the largest value when prec$= p$ is less than 2^{2p+1}. The value of prec equals BITS_PER_MP_LIMB/2 when we exit the loop (we assume here b=BITS_PER_MP_LIMB is a power of two, which holds on all modern architectures), thus the largest value is $2^{b/2+1}$ and no overflow occurs.

After the while-loop, we have $np[0] = s^2 + r$ with s having exactly $b/2$ bits, and $0 \leq r \leq 2s$. We simply put s into $sp[0]$, r into $rp[0]$ if this pointer is valid, and we return the remainder size:

```
sp[0] = s;
if (rp != NULL) rp[0] = r;
return (r) ? 1 : 0;
```

2.2 Proof of mpn_sqrtrem2

This function uses the same algorithm as mpn_sqrtrem1, but we have to take care of overflows: as we noticed in the proof of mpn_sqrtrem1, algorithm SqrtRem requires intermediate values up to BITS_PER_MP_LIMB+1 bits for the square root of a number of b=BITS_PER_MP_LIMB bits.

After having saved $np[0]$ in a temporary variable (remember rp and np may be equal), we use mpn_sqrtrem1 to get $np[1] = sp[0]^2 + rp[0]$ with $sp[0]$ having exactly $b/2$ bits, and $0 \leq rp[0] \leq 2s[0]$:

```
np0 = np[0];
mpn_sqrtrem1(sp, rp, np+1);
```

Then we write $rp[0] = qhl\, sp[0] + rp'[0]$ with $0 \le rp'[0] < sp[0] \le 2^{b/2}$:

```
qhl = 0;
while (rp[0] >= sp[0]) {
  qhl ++;
  rp[0] -= sp[0];
}
```

The while-loop is executed at most twice: $0 \le qhl \le 2$. We now apply algorithm `SqrtRem`, keeping in mind that $qhl\, sp[0]$ should be added to the value of $rp[0]$:

```
rp[0] = (rp[0] << Prec) + (np0 >> Prec);
u = 2 * sp[0];
q = rp[0] / u;
```

At this point, $rp[0] < 2^b$ and $2^{b/2} \le u$ thus $q < 2^{b/2}$. It is crucial here that $rp[0]$ is considered as unsigned when divided by u.

```
u = rp[0] - q * u;
```

Here we have $rp[0] = (2sp[0])q + u$, and the product `q * u` cannot overflow since by construction it is at most $rp[0]$. We now take into account the value of `qhl`:

```
q += (qhl & 1) << (Prec-1);
qhl >>= 1;
```

and we have that the initial value of $rp[0]$ multiplied by $2^{b/2}$ plus the $b/2$ most significant bits of $np[0]$ equal

$$(qhl\, 2^{b/2} + q)(2sp[0]) + u.$$

As $qhl\, 2^{b/2} + q \le 2^{b/2}$, when qhl is not zero, it is necessarily 1 and $q = 0$; this important fact will be used below.

```
sp[0] = ((sp[0]+qhl) << Prec) + q;
```

The above instruction can overflow only when $sp[0] = 2^{b/2} - 1$ and $qhl = 1$, which implies $rp[0] = 2sp[0]$ and thus $np[1] = 2^b - 1$. In that case (only) we have $sp[0] = 0$; we will use this fact below too.

```
cc = u >> Prec;
rp[0] = (u << Prec) + (np0 & (((mp_limb_t)1 << Prec) - 1));
```

The remainder of the division of $rp[0]$ by $2sp[0]$, multiplied by $2^{b/2}$ and added to the least $b/2$ significant bits of $np[0]$ equals now $cc2^b + rp[0]$. We have to subtract from this $(qhl2^{b/2} + q)^2$; if $qhl = 0$, we subtract q^2, otherwise we subtract $qhl\, 2^b$:

```
cc -= mpn_sub_1(rp, rp, 1, q * q) + qhl;
```

If `cc` is negative, this means we have a negative remainder; we have to adjust it by adding $2sp[0] - 1$ to it and decreasing $sp[0]$:

```
if (cc < 0) {
  cc += (sp[0]) ? mpn_add_1(rp, rp, 1, sp[0]) : 1;
  cc += mpn_add_1(rp, rp, 1, --sp[0]);
}
```

In case $sp[0] = 0$ (see above) this means the approximate square root is 2^b, thus adding 2^b to $rp[1]$ or 1 to cc is equivalent in the first `mpn_add_1` line. The second call does what is wanted, $sp[0]$ being zero or not.

Finally we return the remainder carry:

```
return cc;
```

2.3 Proof of mpn_dq_sqrtrem

This is the hearth of the implementation. This recursive function computes in-place the square root of a number of $2n$ limbs, by reducing it to the square root of a number of $2\lceil n/2 \rceil$ limbs, and so on until $n = 1$, where it calls mpn_sqrtrem2:

```
if (n == 1) return mpn_sqrtrem2 (sp, np, np);
```

In the case $n > 1$, let N corresponding to {np,2n} the input number, $l = \lfloor n/2 \rfloor$, $h = \lceil n/2 \rceil$, and N' corresponding to {np+2nl,2h} the upper part of the input. We call mpn_dq_sqrtrem recursively on N', giving

$$N' = {S'}^2 + q\beta^h + R'$$

with $S' = \{sp + l, h\}$ and $R' = \{np + 2l, h\}$:

```
l = n / 2;
h = n - l;
q = mpn_dq_sqrtrem (sp + l, np + 2 * l, h);
```

Now if q is not zero (then it equals 1) we subtract S' to R' (necessarily this gives a borrow out since $q\beta^h + R' - S' \leq 2S' - S' < \beta^h$):

```
if (q) mpn_sub_n (np + 2 * l, np + 2 * l, sp + l, h);
```

This gives $N' = {S'}^2 + q\,S' + \{np + 2l, h\}$, or multiplying by β^{2l}:

$$N = (S'\beta^l)^2 + q\,\beta^{2l} S' + \{np, n + l\}.$$

We now divide the n most significant limbs of {np,n+l} by S' and put the quotient in {sp,l} — the carry out being added to q — and the remainder in {np+l,h}:

```
q += mpn_divrem (sp, 0, np + l, n, sp + l, h);
```

We have now $0 \leq q \leq 2$ and:

$$N = (S'\beta^l)^2 + (q\beta^l + \{sp, l\})S'\beta^l + \{np, n\}.$$

Dividing that equation by β^{2l}, taking the floor of both sides and using $N' \geq {S'}^2 + 2S'$ gives:

$$qS' + \lfloor \frac{\{sp, l\}S'}{\beta^l} \rfloor \leq 2S'.$$

If $q = 2$, we get $\{sp, l\}S' < \beta^l$ which implies $\{sp, l\} < 2$ since $S' \geq 2^h/2$. The quotient $q\beta^l + \{sp, l\}$ is now divided by two, the parity bit is saved into c, the parity bit of q is stored in the most significant bit of {sp,l} and q is divided by two:

```
c = sp[0] & 1;
mpn_rshift (sp, sp, n, 1);
sp[l-1] |= q << (BITS_PER_MP_LIMB - 1);
q >>= 1;
```

Let S_0 represent {sp,l}: if q is not zero, then S_0 is zero, since in that case q was 2 before, and we have shown above that $\{sp, l\}$ was at most 1, whence became 0 when shifted to the right. At this point, the input number decomposes into:

$$N = (S'\beta^l)^2 + (q\beta^l + S_0)(2S'\beta^l) + cS'\beta^l + \{np, n\}.$$

We now add $cS'\beta^l$ to {np,n}:

```
if (c) c = mpn_add_n (np + l, np + l, sp + l, h);
```

giving:

$$N = (S'\beta^l + q\beta^l + S_0)^2 - q\beta^{2l} - S_0^2 + c\beta^n + \{np, n\}.$$

where we used the fact that $q^2 = q$ since $0 \le q \le 1$, and $qS_0 = 0$ since at least one of them is zero. We now compute S_0^2, store it in {np+n,2l} (this space is free since only the low n limbs of the input are now used), and subtract it from {np,2l}:

```
mpn_mul_n (np + n, sp, sp, l);
b = q + mpn_sub_n (np, np, np + n, 2 * l);
```

This now gives:

$$N = (S'\beta^l + q\beta^l + S_0)^2 - b\beta^{2l} + c\beta^n + \{np, n\}.$$

If n is odd, we have now to propagate the borrow b up to the carry c; and we propagate q into the square root:

```
c -= (l == h) ? b : mpn_sub_1 (np + 2 * l, np + 2 * l, 1, b);
q = mpn_add_1 (sp + l, sp + l, h, q);
```

We have now the following formula for N, where $0 \le q \le 1$ and $-1 \le c \le 1$:

$$N = (q\beta^n + \{sp, n\})^2 + c\beta^n + \{np, n\}.$$

If c is negative (i.e. $c = -1$), we overestimated the square root, and thus have to decrease it by one, and to increase the remainder accordingly:

```
if (c < 0) {
  c += mpn_addmul_1 (np, sp, n, 2) + 2 * q;
  c -= mpn_sub_1 (np, np, n, 1);
  q -= mpn_sub_1 (sp, sp, n, 1);
}
```

Let $S = q\beta^n + \{sp, n\}$. The mpn_addmul_1 call adds $2S$ to the remainder, while the first mpn_sub_1 call subtracts one to it, and the second mpn_sub_1 call subtracts one from S. Notice that q is necessarily zero now, since $\lfloor\sqrt{N}\rfloor < \beta^n$, and we have:

$$N = \{sp, n\}^2 + c\beta^n + \{np, n\}$$

with $0 \le c \le 1$, which we finally return:

```
return c;
```

2.4 Proof of mpn_sqrtrem

This is the user-visible interface. It mainly normalizes the input if it is not or when it has an odd size, calls mpn_dq_sqrtrem, and then "denormalizes" the results.

We first compute the number c of leading zeroes in the most significant limb of the input (that limb is supposed to be non zero):

```
count_leading_zeros(c, np[nn-1]);
```

If the size is one and $c = 0$ we can call directly mpn_sqrtrem1:

```
if (nn == 1 && c == 0) return mpn_sqrtrem1 (sp, rp, np);
```

Otherwise we divide c by two and compute $tn = \lceil nn/2 \rceil$:

```
c = c / 2;
tn = (nn+1) / 2;
```

Thus we have to shift the input by $2c$ bits to the left and add $2tn - nn$ zero limbs to the right to normalize it.

We now distinguish the easy case where nn is even and $c = 0$ from the other more difficult cases:

```
TMP_MARK (marker);
if ((nn % 2) || (c > 0)) {
  tp = (mp_limb_t*) TMP_ALLOC (2 * tn * BYTES_PER_MP_LIMB);
  tp[0] = 0; /* needed only when 2*tn > nn, but saves a test */
  if (c) mpn_lshift(tp + 2*tn - nn, np, nn, 2 * c);
  else MPN_COPY (tp + 2*tn - nn, np, nn);
```

At this point `{tp,2tn}` equals the input multiplied by $2^{2c}\beta^{2tn-nn}$, and is normalized, thus we can call `mpn_dq_sqrtrem` on it:

```
  rn = mpn_dq_sqrtrem (sp, tp, tn);
```

Let $k = c + (2tn - nn)b/2$ where b is the number of bits in a limb. We have now $2^{2k}N = S^2 + R$ where $S = \{sp, tn\}$ and $R = rn\beta^{tn} + \{tp, tn\}$, thus $2^{2k}N = (S - s_0)^2 + 2Ss_0 - s_0^2 + R$ where $s_0 = S \bmod 2^k$.

```
  c += (nn % 2) * BITS_PER_MP_LIMB / 2;
  s0[0] = sp[0] & (((mp_limb_t) 1 << c) - 1);
```

We now have in the variable c the value of k, and $s0[0]$ contains $S \bmod 2^k$. We first add $2Ss_0$ to R:

```
  rn += mpn_addmul_1 (tp, sp, tn, 2 * s0[0]);
```

giving (note that $k < b$ therefore $s_0 < \beta/2$ and $2s_0$ fits in a limb):

$$2^{2k}N = (S - s_0)^2 - s_0^2 + rn\beta^{tn} + \{tp, tn\}.$$

It remains to subtract s_0^2 and to propagate the corresponding borrow:

```
  cc = mpn_submul_1 (tp, s0, 1, s0[0]);
  rn -= (tn > 1) ? mpn_sub_1(tp + 1, tp + 1, tn - 1, cc) : cc;
```

At this point, we have (rn cannot become negative since we augmented the remainder part):

$$2^{2k}N = (S - s_0)^2 + rn\beta^{tn} + \{tp, tn\}.$$

We now shift the square root by tn bits to the right and the remainder by $2tn$ bits (necessarily the low $2tn$ bits of the remainder are zero), storing it directly at address rp:

```
  mpn_rshift (sp, sp, tn, c);
  tp[tn] = rn;
  if (rp == NULL) rp = tp;
  c = c << 1;
  if (c < BITS_PER_MP_LIMB) tn++; else { tp++; c -= BITS_PER_MP_LIMB; }
  if (c) mpn_rshift (rp, tp, tn, c); else MPN_COPY(rp, tp, tn);
  rn = tn;
```

The easy case where the input is normalized and of even size nn is much simpler:

```
else {
  if (rp == NULL)
    rp = (mp_limb_t*) TMP_ALLOC (nn * BYTES_PER_MP_LIMB);
  if (rp != np) MPN_COPY (rp, np, nn);
  rn = tn + (rp[tn] = mpn_dq_sqrtrem (sp, rp, tn));
}
```

Finally for both cases we determine and return the remainder size:

```
while (rp[rn-1]==0 && rn) rn--;
TMP_FREE (marker);
return rn;
```

2.5 The complete code

```
static const unsigned char approx_tab[192] = {
128, 128, 129, 130, 131, 132, 133, 134, 135, 136, 137, 138, 139, 140, 141,
   142, 143, 144, 144, 145, 146, 147, 148, 149, 150, 150, 151, 152, 153,
   154, 155, 155, 156, 157, 158, 159, 160, 160, 161, 162, 163, 163, 164,
   165, 166, 167, 167, 168, 169, 170, 170, 171, 172, 173, 173, 174, 175,
   176, 176, 177, 178, 178, 179, 180, 181, 181, 182, 183, 183, 184, 185,
   185, 186, 187, 187, 188, 189, 189, 190, 191, 192, 192, 193, 193, 194,
   195, 195, 196, 197, 197, 198, 199, 199, 200, 201, 201, 202, 203, 203,
   204, 204, 205, 206, 206, 207, 208, 208, 209, 209, 210, 211, 211, 212,
   212, 213, 214, 214, 215, 215, 216, 217, 217, 218, 218, 219, 219, 220,
   221, 221, 222, 222, 223, 224, 224, 225, 225, 226, 226, 227, 227, 228,
   229, 229, 230, 230, 231, 231, 232, 232, 233, 234, 234, 235, 235, 236,
   236, 237, 237, 238, 238, 239, 240, 240, 241, 241, 242, 242, 243, 243,
   244, 244, 245, 245, 246, 246, 247, 247, 248, 248, 249, 249, 250, 250,
   251, 251, 252, 252, 253, 253, 254, 254, 255};

mp_size_t mpn_sqrtrem1 (mp_ptr sp, mp_ptr rp, mp_srcptr np)
{
  mp_limb_t np0, s, r, q, u;
  int prec;

  np0 = np[0];
  q = np0 >> (BITS_PER_MP_LIMB - 8);
  s = approx_tab[q - 64];
  r = (np0 >> (BITS_PER_MP_LIMB - 16)) - s * s;
  if (r > 2 * s) { r -= 2 * s + 1; s++; }

  prec = 8;
  np0 <<= 2 * prec;
  while (2*prec < BITS_PER_MP_LIMB) {
    r = (r << prec) + (np0 >> (BITS_PER_MP_LIMB - prec));
    np0 <<= prec;
    u = 2 * s;
    q = r / u;
    u = r - q * u;
    s = (s << prec) + q;
    u = (u << prec) + (np0 >> (BITS_PER_MP_LIMB - prec));
    q = q * q;
    r = u - q;
    if (u < q) {
      r += 2 * s - 1;
      s --;
    }
    np0 <<= prec;
    prec = 2 * prec;
  }
  sp[0] = s;
  if (rp != NULL) rp[0] = r;
  return (r) ? 1 : 0;
}

#define Prec (BITS_PER_MP_LIMB >> 1)

mp_limb_t mpn_sqrtrem2 (mp_ptr sp, mp_ptr rp, mp_srcptr np)
{
  mp_limb_t qhl, q, u, np0;
```

```
  int cc;

  np0 = np[0];
  mpn_sqrtrem1(sp, rp, np+1);
  qhl = 0;
  while (rp[0] >= sp[0]) {
    qhl ++;
    rp[0] -= sp[0];
  }
  rp[0] = (rp[0] << Prec) + (np0 >> Prec);
  u = 2 * sp[0];
  q = rp[0] / u;
  u = rp[0] - q * u;
  q += (qhl & 1) << (Prec-1);
  qhl >>= 1;
  sp[0] = ((sp[0]+qhl) << Prec) + q;
  cc = u >> Prec;
  rp[0] = (u << Prec) + (np0 & (((mp_limb_t)1 << Prec) - 1));
  cc -= mpn_sub_1(rp, rp, 1, q * q) + qhl;
  if (cc < 0) {
    cc += (sp[0]) ? mpn_add_1(rp, rp, 1, sp[0]) : 1;
    cc += mpn_add_1(rp, rp, 1, --sp[0]);
  }
  return cc;
}

mp_limb_t mpn_dq_sqrtrem (mp_ptr sp, mp_ptr np, mp_size_t n)
{
  mp_limb_t q;
  int c, b;
  mp_size_t l, h;

  if (n == 1) return mpn_sqrtrem2 (sp, np, np);

  l = n / 2;
  h = n - l;
  q = mpn_dq_sqrtrem (sp + l, np + 2 * l, h);
  if (q) mpn_sub_n (np + 2 * l, np + 2 * l, sp + l, h);
  q += mpn_divrem (sp, 0, np + l, n, sp + l, h);
  c = sp[0] & 1;
  mpn_rshift (sp, sp, l, 1);
  sp[l-1] |= q << (BITS_PER_MP_LIMB - 1);
  q >>= 1;
  if (c) c = mpn_add_n (np + l, np + l, sp + l, h);
  mpn_mul_n (np + n, sp, sp, l);
  b = q + mpn_sub_n (np, np, np + n, 2 * l);
  c -= (l == h) ? b : mpn_sub_1 (np + 2 * l, np + 2 * l, 1, b);
  q = mpn_add_1 (sp + l, sp + l, h, q);

  if (c < 0) {
    c += mpn_addmul_1 (np, sp, n, 2) + 2 * q;
    c -= mpn_sub_1 (np, np, n, 1);
    q -= mpn_sub_1 (sp, sp, n, 1);
  }

  return c;
}
```

```
mp_size_t mpn_sqrtrem_main (mp_ptr sp, mp_ptr rp, mp_srcptr np, mp_size_t nn)
{
  mp_limb_t *tp, s0[1], cc;
  int c, tn;
  mp_size_t rn;
  TMP_DECL (marker);

  count_leading_zeros(c, np[nn-1]);

  if (nn == 1 && c == 0) return mpn_sqrtrem1 (sp, rp, np);

  c = c / 2;
  tn = (nn+1) / 2;

  TMP_MARK (marker);
  if ((nn % 2) || (c > 0)) {
    tp = (mp_limb_t*) TMP_ALLOC (2 * tn * BYTES_PER_MP_LIMB);
    tp[0] = 0;
    if (c) mpn_lshift(tp + 2*tn - nn, np, nn, 2 * c);
    else MPN_COPY (tp + 2*tn - nn, np, nn);
    rn = mpn_dq_sqrtrem (sp, tp, tn);
    c += (nn % 2) * BITS_PER_MP_LIMB / 2;
    s0[0] = sp[0] & (((mp_limb_t) 1 << c) - 1);
    rn += mpn_addmul_1 (tp, sp, tn, 2 * s0[0]);
    cc = mpn_submul_1 (tp, s0, 1, s0[0]);
    rn -= (tn > 1) ? mpn_sub_1(tp + 1, tp + 1, tn - 1, cc) : cc;
    mpn_rshift (sp, sp, tn, c);
    tp[tn] = rn;
    if (rp == NULL) rp = tp;
    c = c << 1;
    if (c < BITS_PER_MP_LIMB) tn++; else tp++;
    mpn_rshift (rp, tp, rn = tn, c % BITS_PER_MP_LIMB);
  }
  else {
    if (rp == NULL)
      rp = (mp_limb_t*) TMP_ALLOC (nn * BYTES_PER_MP_LIMB);
    if (rp != np) MPN_COPY (rp, np, nn);
    rn = tn + (rp[tn] = mpn_dq_sqrtrem (sp, rp, tn));
  }
  while (rp[rn-1]==0 && rn) rn--;
  TMP_FREE (marker);

  return rn;
}
```

2.6 Complexity analysis

The `SqrtRem` algorithm has complexity $S(n)$ where:

$$S(n) = D(n/2) + M(n/2) + D(n/4) + M(n/4) + \cdots$$

If $M(n) = mn^\alpha$ for $\alpha > 1$ and the division uses the algorithm described above, then

$$D(n) = 2M(n/2) + 2D(n/2) = \frac{1}{2^{\alpha-1} - 1} M(n)$$

If $D(n) = dM(n)$ and $M(n) = mn^\alpha$, then we have

$$S(n) = \frac{d+1}{2^\alpha - 1} M(n) = \frac{2^\alpha}{(2^\alpha - 1)(2^\alpha - 2)} M(n).$$

The usual Newton iteration $x_{k+1} = x_k + \frac{1}{2}(x_k + \frac{A}{x_k})$ for the square root requires $D(n) + D(n/2) + \cdots$ plus $M(n)$ to get the remainder, which gives $\frac{2^{2\alpha}-2^{\alpha}+2}{(2^{\alpha}-1)(2^{\alpha}-2)}M(n)$ for $\alpha > 1$, and $2D(n) + M(n)$ using FFT. The `SqrtRem` algorithm is always faster by at least $D(n)$ since $M(n)$ is greater than $M(n/2) + M(n/4) + \cdots$.

Using Newton iteration for the inverse square root $x_{k+1} = x_k + \frac{x_k}{2}(1 - Ax_k^2)$ together with Karp-Markstein improvement, we get a cost of $3M(n/2) + 4M(n/4) + 4M(n/8) + \cdots$, which equals $(32^{\alpha}+1)/2^{\alpha}/(2^{\alpha}-1)M(n)$ for $\alpha > 1$, and $\frac{7}{2}M(n)$ in the FFT case, to which we have to add $M(n)$ to compute the remainder. This method is faster than `SqrtRem` for $\alpha < 1.328$, which means Toom-Cook method of order at least 7.

	Divrem	SqrtRem	Newton $x^{1/2}$	Newton $x^{-1/2}$ + Karp/Markstein
Karatsuba	2	3/2	4	8/3
Toom-Cook	2.63	2.06	5.12	2.91
FFT	5	6	11	9/2

References

[1] Burnikel, C., and Ziegler, J. Fast recursive division. Research Report MPI-I-98-1-022, MPI Saarbrücken, Oct. 1998. Available from `http://www.mpi-sb.mpg.de/~ziegler/TechRep.ps.gz`.

[2] Granlund, T. *GNU MP: The GNU Multiple Precision Arithmetic Library*, 3.1 ed., Aug. 2000. Available from `http://www.swox.se/gmp/`.

[3] Jebelean, T. Practical integer division with Karatsuba complexity. In *Proc. ISSAC'97* (Maui, Hawaii, 1997), W. W. Küchlin, Ed., pp. 339–341.

[4] Moenck, R., and Borodin, A. Fast modular transforms via division. In *Proceedings of the 13th Annual IEEE Symposium on Switching and Automata Theory* (Oct. 1972), pp. 90–96.

[5] Zimmermann, P. Karatsuba square root. Research Report 3805, INRIA, Nov. 1999.