Maxima by Example:
Ch.9: Bigfloats and Arbitrary Precision Quadrature *

Edwin L. Woollett

February 3, 2011

Contents

*This version uses **Maxima 5.18.1** except for the revised section on bfallroots, which uses **Maxima 5.23.2**. The author would like to thank the Maxima developers for their friendly help via the Maxima mailing list, and Michel Talon for feedback about **bfallroots** behavior.This is a live document. Check **http://www.csulb.edu/~woollett/** for the latest version of these notes. Send comments and suggestions to **woollett@charter.net**

```
COPYING AND DISTRIBUTION POLICY
This document is part of a series of notes titled
"Maxima by Example" and is made available
via the author's webpage http://www.csulb.edu/~woollett/
to aid new users of the Maxima computer algebra system.

NON-PROFIT PRINTING AND DISTRIBUTION IS PERMITTED.

You may make copies of this document and distribute them
to others as long as you charge no more than the costs of printing.

These notes (with some modifications) will be published in book form
eventually via Lulu.com in an arrangement which will continue
to allow unlimited free download of the pdf files as well as the option
of ordering a low cost paperbound version of these notes.
```

Feedback from readers is the best way for this series of notes to become more helpful to new users of Maxima. *All* comments and suggestions for improvements will be appreciated and carefully considered.

```
LOADING FILES
The defaults allow you to use the brief version load(brmbrg) to load in the
Maxima file brmbrg.lisp.
To load in your own file, such as qbromberg.mac (used in this chapter),
using the brief version load(qbromberg), you either need to place
qbromberg.mac in one of the folders Maxima searches by default, or
else put a line like:

file_search_maxima : append(["c:/work3/###.{mac,mc}"],file_search_maxima )$

in your personal startup file maxima-init.mac (see Ch. 1, Introduction to Maxima
for more information about this).

Otherwise you need to provide a complete path in double quotes,
as in load("c:/work3/qbromberg.mac"),

We always use the brief load version in our examples, which are generated
using the Xmaxima graphics interface on a Windows XP computer, and copied
into a fancy verbatim environment in a latex file which uses the fancyvrb
and color packages.

We use qdraw.mac for plots (see Ch.5), which uses draw2d defined
in share/draw/draw.lisp.
```

```
  Maxima, a Computer Algebra System.
  Some numerical results depend on the Lisp version used.
  This chapter uses Version 5.18.1 (2009) using Lisp GNU
  Common Lisp (GCL) GCL 2.6.8 (aka GCL).
  http://maxima.sourceforge.net/
```

9.1 Introduction

This chapter is divided into two sections.

In the first section we discuss the use of **bfloat**, including examples which also involve **fpprec**, **bfloatp**, **bfallroots**, **fpprintprec**, **print**, and **printf**. The second section of Chapter 9 presents examples of the use of Maxima for arbitrary precision quadrature (numerical integration). (In Chapter 8, we gave numerous examples of numerical integration using the Quadpack functions such as **quad_qags** as well as **romberg**. Those examples all accepted the default floating point precision of Maxima).

Chapter 10 covers both Fourier transform and Laplace transform type integrals. Chapter 11 presents tools for the use of fast Fourier transforms with examples of use.

Software files developed for Ch. 9 and available on the author's web page include:
1. fdf.mac, 2. qbromberg.mac. ,
3. quad_de.mac, 4. quad_ts.mac, 5. quad_gs.mac.

9.2 The Use of Bigfloat Numbers in Maxima

9.2.1 Bigfloat Numbers Using bfloat, fpprec, and fpprintprec

bfloat(expr) converts all numbers and functions of numbers in **expr** to bigfloat numbers. You can enter explicit bigfloat numbers using the notation **2.38b0**, or **2.38b7**, or **2.38b-4**, for example. **bfloatp(val)** returns **true** if **val** is a bigfloat number, otherwise **false** is returned.

The number of significant digits in the resulting bigfloat is specified by the parameter **fpprec**, whose default value is **16**. The setting of **fpprec** does not affect computations on ordinary floating point numbers.

The underlying Lisp code has two variables: 1. **$fpprec**, which defines the Maxima variable **fpprec** (which determines the number of DECIMAL digits requested for arithmetic), and 2. **fpprec** which is related to the number of bits used for the fractional part of the bigfloat, and which can be accessed from Maxima using **?fpprec**. We can also use the **:lisp foobar** construct to look at a lisp variable **foobar** from "inside" the Lisp interpreter. You do not end with a semi-colon; you just press ENTER to get the response, and the input line number of the Maxima prompt does not advance.

```
(%i1) fpprec;
(%o1)                                16
(%i2) ?fpprec;
(%o2)                                56
(%i3) :lisp $fpprec
16
(%i3) :lisp fpprec
56
```

We discuss some effects of the fact that computer arithmetic is carried out with binary bit representation of decimal numbers in Sec. 9.2.5.

Controlling Printed Digits with fpprintprec

When using bigfloat numbers, the screen can quickly fill up with numbers with many digits, and **fpprintprec** allows you to control how many digits are displayed. For bigfloat numbers, when **fpprintprec** has a value between **2** and **fpprec** (inclusive), the number of digits printed is equal to **fpprintprec**. Otherwise, **fpprintprec** can be **0**, or greater than **fpprec**, in which case the number of digits printed is equal to **fpprec**. **fpprintprec** cannot be **1**. The setting of **fpprintprec** does not affect the precision of the bigfloat arithmetic carried out, only the setting of **fpprec** matters.

The parameter **fpprec** can be used as a local variable in a function defined using **block**, and set to a local value which does not affect the global setting. Such a locally defined value of **fpprec** governs the bigfloat calculations in that function and in any functions called by that function, and in any third layer functions called by the secondary layer functions, etc. **fpprintprec** can be set to a value inside a function defined with **block**,without changing the global value, provided you include the name **fpprintprec** in the local variables bracket **[ ]**:

```
(%i1) [fpprec,fpprintprec];
(%o1)                                [16, 0]
(%i2) piby2 : block([fpprintprec,fpprec:30,val],
                 val:bfloat(%pi/2),
                 fpprintprec:8,
                 disp(val),
                  print("  ",val),
                  val);
                                1.5707963b0

   1.5707963b0
(%o2)                  1.57079632679489661923132169164b0
(%i3) [fpprec,fpprintprec];
(%o3)                                [16, 0]
```

For simple **bfloat** uses in interactive mode, one can use the syntax **bfloat-job, fpprec:fp ;** which implicitly uses the **ev(...)** construct with temporary settings of global flags as in

```
(%i1) bfloat(%pi),fpprec:20;
(%o1)                       3.1415926535897932385b0
(%i2) slength(string(%));
(%o2)                                 23
(%i3) fpprec;
(%o3)                                 16
(%i4) bfloat(%pi),fpprec:40;
(%o4)             3.1415926535897932384626433832795028841971b0
(%i5) slength(string(%));
(%o5)                                 43
(%i6) fpprec;
(%o6)                                 16
(%i7) tval : bfloat(integrate(exp(x),x,-1,1)),fpprec:30;
(%o7)                  2.35040238728760291376476370119b0
(%i8) slength(string(%));
(%o8)                                 33
```

Next we illustrate passing both a bigfloat number as well as local values of **fpprec** and **fpprintprec** to a second function. Function **f1** is designed to call function **f2**:

```
(%i1) f2(w) := block([v2 ],
             disp(" in f2 "),
             display([w,fpprec,fpprintprec]),
              v2 : sin(w),
              display(v2),
              print("  "),
               v2 )$
(%i2) f1(x,fp,fprt) :=
            block([fpprintprec,fpprec:fp,v1],
              fpprintprec:fprt,
              disp(" in f1 "),
               display([x,fpprec,fpprintprec]),
               v1 : f2(bfloat(x))^2,
               print(" in f1, v1 = ",v1),
               v1 )$
```

And here we call **f1** with values **x = 0.5, fp = 30, fprt = 8**:

```
(%i3) f1(0.5,30,8);
                                    in f1
                      [x, fpprec, fpprintprec] = [0.5, 30, 8]
                                    in f2
                    [w, fpprec, fpprintprec] = [5.0b-1, 30, 8]
                                 v2 = 4.7942553b-1
 in f1, v1 =  2.2984884b-1
(%o3)                 2.29848847065930141299531696279b-1
(%i4) [fpprec,fpprintprec];
(%o4)                               [16, 0]
```

We see that the called function (**f2**) maintains the values of **fpprec** and **fpprintprec** which exist in the calling function (**f1**).

Bigfloat numbers are "contagious" in the sense that, for example, multiplying (or adding) an integer or ordinary float number with a bigfloat results in a bigfloat number. In the above example **sin(w)** is a bigfloat since **w** is one.

```
(%i5) 1 + 2.0b0;
(%o5)                                3.0b0
(%i6) 1.0 + 2.0b0;
(%o6)                                3.0b0
(%i7) sin(0.5b0);
(%o7)                          4.79425538604203b-1
```

The Maxima symbol **%pi** is not automatically converted by contagion (in the present version of Maxima), and an extra use of **bfloat** does the conversion.

```
(%i8) a:2.38b-1$
(%i9) exp(%pi*a);
                                    2.38b-1 %pi
(%o9)                              %e
(%i10) bfloat(%);
(%o10)                          2.112134508503361b0
```

A recent comment by Maxima developer Volker van Nek on the Maxima mailing list illustrates how the setting of **fpprec** can affect what is printed to the screen when doing bigfloat arithmetic.

```
(%i11) 1b0 + 1b-25,fpprec:26;
(%o11)                    1.00000000000000000000000001b0
(%i12) slength(string(%));
(%o12)                                  29
```

Note that **1b-25** is the bigfloat version of 10^{-25}. In the case above the value of **fpprec** is large enough to see the effect of the bigfloat addition. The number of digits identified by the conversion of the number to a string and then finding the length of the string is $29 - 3 = 26$ since the three string characters $[., \mathrm{b}, 0]$ do not contribute to the precision of the number. If we leave the **fpprec** at **26** and add to **1** the bigfloat equivalent of 10^{-26}, Maxima prints out a bigfloat version of **1**, but if we subtract from that answer **1b0** we do not get zero:

```
(%i13) 1b0 + 1b-25,fpprec:26;
(%o13)                    1.00000000000000000000000001b0
(%i14) slength(string(%));
(%o14)                                  29
(%i15) 1b0 +10b0^(-26),fpprec:26;
(%o15)                                1.0b0
(%i16) % - 1b0,fpprec:26;
(%o16)                   9.6935228033557930648993206b-27
(%i17) slength(string(%));
(%o17)                                  31
```

Of the thirty one characters present in this string, the five characters $[., \mathrm{b}, -, 2, 7]$ do not contribute to the precision, , leaving **26** digits of precision. Note that the setting of **fpprintprec** was **0** in the above, which provides for printing of the same number of significant digits as the value of **fpprec**.

Here are three ways to enter the bigfloat equivalent of $1/10 = 10^{-1} = 0.1$:

```
(%i18) [1b-1,bfloat(10^(-1)),10b0^(-1)];
(%o18)                     [1.0b-1, 1.0b-1, 1.0b-1]
```

We next show that once bigfloat numbers are in play inside a **block** function in which a local value of **fpprec** is set, the subsequent arithmetic is the same whether or not you wrap each operation with **bfloat**.

```
(%i19) test1(h,j,fp):=
  block([fpprec:fp,u,a,x,w],
   u:bfloat(h*j),
   a:bfloat(exp(-u)),
   x:bfloat(exp(u-a)),
   w:bfloat(exp(-a)+x),
   [x,w])$
(%i20) test2(h,j,fp):=
  block([fpprec:fp,u,a,x,w],
   u:bfloat(h*j),
   a:exp(-u),
   x:exp(u-a),
   w:exp(-a)+x,
   [x,w])$
(%i21) [xt,wt]:test1(1/16,9,60)$
(%i22) [x1,w1]:test1(1/16,9,40)$
(%i23) [x2,w2]:test2(1/16,9,40)$
```

```
(%i24) [x1,w1]-[xt,wt],fpprec:60;
(%o24) [- 2.29699398191727953386796229322249912456243487302509210328785b-42,
          - 9.07907621847706474924315736438559811433348894389391230442376b-42]
(%i25) [x2,w2]-[xt,wt],fpprec:60;
(%o25) [- 2.29699398191727953386796229322249912456243487302509210328785b-42,
          - 9.07907621847706474924315736438559811433348894389391230442376b-42]
(%i26) bfloat([x1,w1]-[xt,wt]),fpprec:60;
(%o26) [- 2.29699398191727953386796229322249912456243487302509210328785b-42,
          - 9.07907621847706474924315736438559811433348894389391230442376b-42]
(%i27) map('bfloatp,%o5);
(%o27)                              [true, true]
(%i28) map('bfloatp,%o4);
(%o28)                             [true, true]
```

In the above, we used the "completely **bfloat** wrapped" version test1(..) to define answers with **60** digit precision, and then used test1(..) and test2(..) to compute comparison answers at **40** digit precision. We see that there is no difference in precision between the answers returned by the two test versions (each using 40 digit precision).

We also see, from the output **%08**, that **`arith_job, fpprec:60;`** using interactive mode produces the same answer (with bigfloats already in play) whether or not the "**`arithmetic_job`**" is wrapped in **bfloat**. The numbers returned by both test versions are bigfloats, as indicated by the last two outputs.

9.2.2 Using print and printf with Bigfloats

In Sec. 9.2.1 we described the relations between the settings of **fpprec** and **fpprintprec**. Once you have generated a bigfloat with some precision, it is convenient to be able to control how many digits are displayed. We start with the use of **`print`**. If you start with the global default value of **16** for **fpprec** and the default value of **0** for **fpprintprec**, you can use a simple one line command for a low number of digits, as shown in the following. We first define a bigfloat **`bf1`** to have **`fpprec = 45`** digits of precision:

```
(%i1) [fpprec,fpprintprec];
(%o1)                                [16, 0]
(%i2) bf1:bfloat(integrate(exp(x),x,-1,1)),fpprec:45;
(%o2)          2.35040238728760291376476370119120163031143596b0
(%i3) slength(string(%));
(%o3)                                 48
```

We then use **`print`** with **`fpprintprec`** to get increasing numbers of digits on the screen:

```
(%i4) print(bf1),fpprintprec:12$
2.35040238728b0
(%i5) [fpprec,fpprintprec];
(%o5)                                [16, 0]
(%i6) print(bf1),fpprintprec:15$
2.3504023872876b0
(%i7) [fpprec,fpprintprec];
(%o7)                                [16, 0]
(%i8) print(bf1),fpprintprec:16$
2.35040238728760291376476370119120163031143596b0
(%i9) [fpprec,fpprintprec];
(%o9)                                [16, 0]
(%i10) slength(string(%o8));
(%o10)                                48
```

As you see above, when **fpprintprec** reaches the global value of **fpprec = 16** all **45** digits are printed. To control the number of printed digits, you need to locally set the value of **fpprec** as shown here:

```
(%i11) print(bf1),fpprec:20,fpprintprec:18$
2.35040238728760291b0
```

To use this construct in a **do** loop, wrap it in **ev(...)**:

```
(%i12) for j:14 thru 18 do ev(print(bf1),fpprec:j+2,fpprintprec:j)$
2.3504023872876b0
2.3504023872876b0
2.350402387287602b0
2.3504023872876029b0
2.35040238728760291b0
```

A more formal approach is to define a small function which we call **bfprint**:

```
(%i13) bfprint(bf,fpp):=
  block([fpprec, fpprintprec ],
        fpprec : fpp+2,
        fpprintprec:fpp,
        print("   number of digits = ",fpp),
        print("  ",bf) )$
```

with the behavior:

```
(%i14) bfprint(bf1,24)$
   number of digits =  24
   2.3504023872876029137647 6b0
```

Using printf with bigfloats

We first show some interactive use of **printf** with bigfloats.

```
(%i1) bf:bfloat(exp(-20)),fpprec:30;
(%o1)                  2.061153622438557827965940380 16b-9
(%i2) slength(string(%));
(%o2)                                  34
(%i3) printf(true,"~d~a",3,string(bf))$
32.061153622438557827965940380 16b-9
```

The format string is enclosed in double quotes, with **~d** used for an integer, **~f** used for a floating point number, **~a** used for a Maxima string, **~e** used for exponential display of a floating point number, and **~h** used for a bigfloat number. You can include the newline instruction with **~%** anywhere and as many times as you wish. In the example above, we used the string formatting to display the bigfloat number bf, which required that bf be converting to a Maxima string using **string**. Because we did not include any spaces between the integer format instruction **~d** and the string format character **~a**, we get **32.0...** instead of **3 2.0...**.

```
(%i4) printf(true,"  ~d~a",3,string(bf))$
  32.061153622438557827965940380 16b-9
(%i5) printf(true,"  ~d    ~a",3,string(bf))$
  3    2.061153622438557827965940380 16b-9
```

```
(%i6) (printf(true,"  ~d    ~a",3,string(bf)),
           printf(true,"  ~d    ~a",3,string(bf)))$
  3    2.06115362243855782796594038016b-9  3    2.06115362243855782796594038016b-9
(%i7) (printf(true,"  ~d    ~a~%",3,string(bf)),
           printf(true,"  ~d    ~a",3,string(bf)))$
  3    2.06115362243855782796594038016b-9
  3    2.06115362243855782796594038016b-9
```

To get the output on successive lines we had to include the newline instruction **~ %**. To control the number of significant figures displayed, we use **fpprintprec**:

```
(%i8) fpprintprec:8$
(%i9) printf(true,"  ~d    ~a",3,string(bf))$
  3    2.0611536b-9
```

Next let's show what we get if we use the other options:

```
(%i10) printf(true,"  ~d    ~f",3,bf)$
  3    0.00000000206115362243855790
(%i11) printf(true,"  ~d    ~e",3,bf)$
  3    2.0611536224385579E-9
(%i12) printf(true,"  ~d    ~h",3,bf)$
  3    0.0000000020611536
```

A Table of Bigfloats using block and printf

Here is an example of using **printf** with bigfloats inside a **block** to make a table.

```
(%i1) print_test(fp) :=
       block([fpprec,fpprintprec,val],
        fpprec : fp,
        fpprintprec : 8,
        display(fpprec),
        print("  k          value "),
        print("  "),
        for k thru 4 do
         ( val : bfloat(exp(k^2)),
           printf(true,"  ~d       ~a  ~%",k,string(val) ) ))$
(%i2) print_test(30)$
                                fpprec = 30

  k          value

  1       2.7182818b0
  2       5.459815b1
  3       8.1030839b3
  4       8.8861105b6
```

Note the crucial use of the newline instruction **~ %** to get the table output. Some general use examples of **printf** can be found in the Maxima manual and in the file

```
C:\Program Files\Maxima-5.17.1\share\maxima\5.17.1\share\
                              contrib\stringproc\rtestprintf.mac
```

We can use **printf** for the titles and empty lines with the alternative version. We first define an alternative function **print_test2**:

```
(%i3) print_test2(fp) :=
       block([fpprec,fpprintprec,val],
        fpprec : fp,
        fpprintprec : 8,
        display(fpprec),
        printf(true,"~%  ~a          ~a ~%~%",k,value),
        for k thru 4 do
         ( val : bfloat(exp(k^2)),
           printf(true,"  ~d       ~a  ~%",k,string(val) ) ))$
```

Here we try out the alternative function with **fp = 30**:

```
(%i4) print_test2(30)$
                                fpprec = 30

  k          value

  1       2.7182818b0
  2       5.459815b1
  3       8.1030839b3
  4       8.8861105b6
```

9.2.3 Adding Bigfloats Having Differing Precision

If **A** and **B** are bigfloats with different precisions, the precision of the sum **(A + B)** is the precision of the least precise number. As an example, we calculate an approximation to π using both 30 and 50 digit precision, and add the numbers using 40 digit precision, and then using 60 digit precision. In both cases, the result has 30 digit precision.

```
(%i1) fpprintprec:8$
(%i2) pi50 : bfloat(%pi),fpprec:50;
(%o2)                              3.1415926b0
(%i3) pi30 : bfloat(%pi),fpprec:30;
(%o3)                              3.1415926b0
(%i4) abs(pi30 - pi50),fpprec:60;
(%o4)                           1.6956855b-31
(%i5) twopi : bfloat(2*%pi),fpprec:60;
(%o5)                              6.2831853b0
(%i6) pisum40 : pi30 + pi50,fpprec:40;
(%o6)                              6.2831853b0
(%i7) abs(pisum40 - twopi),fpprec:60;
(%o7)                           1.6956855b-31
(%i8) pisum60 : pi30 + pi50,fpprec:60;
(%o8)                              6.2831853b0
(%i9) abs(pisum60 - twopi),fpprec:60;
(%o9)                           1.6956855b-31
```

9.2.4 Polynomial Roots Using bfallroots

The Maxima function **bfallroots** has the same syntax as **allroots**, and computes numerical approximations of the real and complex roots of a polynomial or polynomial expression of one variable. In all respects, **bfallroots** is identical to **allroots** except that **bfallroots** computes the roots using bigfloats, and to take advantage of bigfloats you need to set both ffprec and ratepsilon to compatible values (as our example shows). The source code of **bfallroots** with some comments is in the file **cpoly.lisp** in the **src** directory.

Our example is a cubic equation whose three degenerate roots are simply π. We are using Maxima 5.23.2 for this revised section, with **display2d:false** set in our init file. We first compute a 50 digit approximation to the true root.

```
(%i1) fpprec;
(%o1) 16
(%i2) pi50 : ev (bfloat (%pi), fpprec:50);
(%o2) 3.1415926535897932384626433832795028841971693993751b0
(%i3) slength( string (%));
(%o3) 53
```

We next define the symbolic cubic expression whose roots we would like to approximately calculate.

```
(%i4) e : expand ( (x-%pi)^3);
(%o4) x^3-3*%pi*x^2+3*%pi^2*x-%pi^3
```

As a warm-up, we use the default 16 digit floating point precision and find the root(s) using both **allroots** and **bfallroots**. We first need to turn the symbolic expression into a polynomial whose coefficients have the default 16 digit accuracy.

```
(%i6) e_f16 : float (e);
(%o6) x^3-9.424777960769379*x^2+29.60881320326807*x-31.00627668029982
```

Now find the approximate roots of this numerical polynomial in **x** using **allroots**.

```
(%i7) sar16 : map ('rhs, allroots (%i*e_f16));
(%o7) [3.14159265358979-1.8873791418627661E-15*%i,
       9.9920072216264089E-16*%i+3.141592653589795,
       8.8817841970012523E-16*%i+3.141592653589795]
```

We first check to see how well the approximate solutions behave as far as causing the approximate numerical polynomial to be zero (as roots should do).

```
(%i8) for s in sar16 do (subst (s,x,e_f16), disp (expand (%%)))$
6.3108872417680944E-30*%i

-6.3108872417680944E-30*%i

-3.1554436208840472E-30*%i
```

which is very good root behavior.

We next compare the approximate roots (taking realpart) to pi50.

```
(%i9) for s in sar16 do disp (pi50 - realpart(s))$
3.663735981263017b-15

-1.665334536937735b-15

-1.665334536937735b-15
```

The above accuracy in finding π corresponds to the default floating point precision being used.

Retaining the default precision, we try out **bfallroots**.

```
(%i10) sbfar16 : map ('rhs, bfallroots (%i*e_f16));
(%o10) [3.141592653589788b0-1.207367539279858b-15*%i,
        5.967448757360216b-16*%i+3.141592653589797b0,
        6.106226635438361b-16*%i+3.141592653589795b0]
```

We then again check the roots against the expression:

```
(%i11) for s in sbfar16 do (subst (s,x,e_f16), disp (expand (%%)))$
7.888609052210118b-31*%i+2.664535259100376b-15

1.332267629550188b-15

1.332267629550188b-15-3.944304526105059b-31*%i
```

and compare the accuracy against our "true value".

```
(%i12) for s in sbfar16 do disp (pi50 - realpart(s))$
5.662137425588298b-15

-3.774758283725532b-15

-1.554312234475219b-15
```

Thus we see that **bfallroots** provides no increased accuracy unless we set **fpprec** and **ratepsilon** to values which will cause Maxima to use higher precision.

In order to demonstrate the necessity of setting **ratepsilon**, we first try out **bfallroots** using only the **fpprec** setting. Let's try to solve for the roots with 40 digit accuracy, first converting the symbolic cubic to a numerical cubic with coefficients having 40 digit accuracy.

```
(%i13) e_f40 : ev (bfloat (e),fpprec : 40);
(%o13) x^3-9.424777960769379715387930149838508652592b0*x^2
          +2.960881320326807585650347299962845340594b1*x
          -3.100627668029982017547631506710139520223b1
```

The coefficients are now big floats, with the tell-tale **b0** or **b1** power of ten factor attached to the end.

Now set **fpprec : 40** and use **bfallroots**:

```
(%i16) fpprec:40$
(%i17) sbfar40 : map ('rhs, bfallroots (%i*e_f40));
'rat' replaced -3.10062766802998201754763150671013952022 3B1
     by -14821/478 = -3.10062761506276150627615062761506276150 6B1
'rat' replaced 2.96088132032680758565034729996284534059 4B1
      by 32925/1112 = 2.96088129496402877697841726618705035 9712B1
'rat' replaced -9.42477796076937971538793014983850865259 2B0
   by -103993/11034 = -9.42477795903570781221678448432118 9052021B0
(%o17) [5.44458491269014927386086037237509616401 9b-3*%i
         +3.13843637674189964130608967670335456242 9b0,
        3.13843637674189964130608967670335468707 2b0
         -5.44458491269014927386086037237516783394 5b-3*%i,
        7.16699258651373003821307027194226450112 6b-35*%i
         +3.14790520555190852960460513091447980252 b0]
```

Check the 40 digit expression using these roots

```
(%i18) for s in sbfar40 do (subst (s,x,e_f40), disp (expand (%%)))$
1.32164984890934009965622426522183815370 3b-9*%i
 +2.49246230713128020808577868767593307175 2b-7

2.49246230713128020808577868767594041859 2b-7
 -1.32164984890934009965622426520477397057 3b-9*%i

8.56777675967247283239974238974913335735 6b-39*%i
 +2.51544541834781986427561133516173418284 1b-7
```

and check the closeness of the roots to the "true value",

```
(%i19) for s in sbfar40 do disp (pi50 - realpart(s))$
3.15627684789359715655370657614832176814 4b-3

3.15627684789359715655370657614819712496 8b-3

-6.31255196211529114196174763497691832245 6b-3
```

which are really poor results, apparently caused by inaccurate **rat** replacement of decimal coefficients by ratios of whole numbers. Look, for example, at the third rat replacemnt above and its difference from the actual 40 digit accurate number:

```
(%i20) bfloat (9.42477796076937971538793014983850865259 2B0 -
                  103993/11034 );
(%o20) 1.73367190317114566551731960057093141813 8b-9
(%i21) ratepsilon;
(%o21) 2.0E-8
```

So we are driven to the conclusion that, with the present design of Maxima, we must set **ratepsilon** to a small number which somehow "matches" the setting of **fpprec**.

```
(%i22) ratepsilon : 1.0e-41$
(%i23) sbfar40 : map ('rhs, bfallroots (%i*e_f40));
`rat' replaced -3.10062766802998201754763150671013952022 3B1
   by -689775162029634828708/2224630738936452 4529
       = -3.10062766802998201754763150671013952022 3B1
`rat' replaced 2.96088132032680758565034729996284534059 4B1
  by 1094430324967716480409/369629919799324688 48
        = 2.96088132032680758565034729996284534059 4B1
`rat' replaced -9.42477796076937971538793014983850865259 2B0
   by -787357891006146598194/83541266890691994833
        = -9.42477796076937971538793014983850865259 2B0
(%o23) [3.14159265358979323846264338327950288419 7b0,
        3.14159265358979323846264338327950288419 2b0
         -2.06629866355480226010129469433597873054 1b-40*%i,
        2.06629866355480226010129469433597873054 1b-40*%i
         +3.14159265358979323846264338327950288420 3b0]
(%i24) for s in sbfar40 do (subst (s,x,e_f40), disp (expand (%%)))$
2.20405190779178907744138100729171064591b-39

0.0b0

7.34683969263929692480460335763903548636 7b-40

(%i25) for s in sbfar40 do disp (pi50 - realpart(s))$
9.18354961579912115600575419704879435795 8b-41

5.05095228868951663580316480837683689687 7b-39

-5.51012976947947269360345251822927661477 5b-39
```

which provides roughly 40 digit accuracy solutions for the roots.

Of course, you can use **ratprint : false** to avoid those pesky **rat** conversion messages.

9.2.5 Bigfloat Number Gaps and Binary Arithmetic

fpprec as set by the user is the number of DECIMAL digits being requested. In fact, the actual arithmetic is carried out with binary arithmetic. Due to the inevitably finite number of binary bits used to represent a floating point number there will be a range of floating point numbers which are not recognised as different.

For a simple example, let's take the case **fpprec = 4**. Consider the gap around the number **x:bfloat(2/3)** whose magnitude is less than **1**. We will find that **?fpprec** has the value **16** and that Maxima behaves as if (for this case) the fractional part of a bigfloat number is represented by the state of a system consisting of **18** binary bits.

Let $\mathbf{u} = \mathbf{2}^{-\mathbf{18}}$. If we let $\mathbf{x1} = \mathbf{x} + \mathbf{u}$ we get a number which is treated as having a nonzero difference from $\mathbf{x}$. However, if we let $\mathbf{w}$ be a number which is one significant digit less than $\mathbf{u}$, and define $\mathbf{x2} = \mathbf{x} + \mathbf{w}$, $\mathbf{x2}$ is treated as having **zero** difference from $\mathbf{x}$. Thus the gap in bigfloats around our chosen $\mathbf{x}$ is $\mathbf{ulp} = \mathbf{2} \cdot \mathbf{2}^{-\mathbf{18}} = \mathbf{2}^{-\mathbf{17}}$, and this gap should be the same size (as long as **fpprec = 4**) for any bigfloat with a magnitude less than **1**.

If we consider a bigfloat number whose decimal magnitude is less than **1**, its value is represented by a "fractional binary number". For the case that this fractional binary number is the state of **18** (binary) bits, the smallest base 2 number which can occur is the state in which all bits are off (0) except the least significant bit which is on (1), and the decimal equivalent of this fractional binary number is precisely $\mathbf{2}^{-\mathbf{18}}$. Adding two bigfloats (each of which has a decimal magnitude less than **1**) when each is represented by the state of an **18** binary bit system (interpreted as a fractional binary number), it is not possible to increase the value of any one bigfloat by less than this smallest base **2** number.

```
(%i1) fpprec:4$
(%i2) ?fpprec;
(%o2)                                16
(%i3) x :bfloat(2/3);
(%o3)                             6.667b-1
(%i4) u : bfloat(2^(-18));
(%o4)                             3.815b-6
(%i5) x1 : x + u;
(%o5)                             6.667b-1
(%i6) x1 - x;
(%o6)                             1.526b-5
(%i7) x2 : x + 3.814b-6;
(%o7)                             6.667b-1
(%i8) x2 - x;
(%o8)                               0.0b0
(%i9) ulp : bfloat(2^(-17));
(%o9)                             7.629b-6
```

In computer science **Unit in the Last Place**, or **Unit of Least Precision**, **ulp(x)**, associated with a floating point number **x** is the gap between the two floating-point numbers closest to the value **x**. We assume here that the magnitude of **x** is less than 1. These two closest numbers will be $\mathbf{x} + \mathbf{u}$ and $\mathbf{x} - \mathbf{u}$ where $\mathbf{u}$ is the smallest positive floating point number which can be accurately represented by the systems of binary bits whose states are used to represent the fractional parts of the floating point numbers.

The amount of error in the evaluation of a floating-point operation is often expressed in ULP. We see that for **fpprec = 4**, 1 ULP is about $8 \cdot 10^{-6}$. An average error of 1 ULP is often seen as a tolerable error.

We can repeat this example for the case **fpprec = 16**.

```
(%i10) fpprec:16$
(%i11) ?fpprec;
(%o11)                                  56
(%i12) x :bfloat(2/3);
(%o12)                          6.666666666666667b-1
(%i13) u : bfloat(2^(-58));
(%o13)                          3.469446951953614b-18
(%i14) x1 : x + u;
(%o14)                          6.666666666666667b-1
(%i15) x1 - x;
(%o15)                          1.387778780781446b-17
(%i16) x2 : x + 3.469446951953613b-18;
(%o16)                          6.666666666666667b-1
(%i17) x2 - x;
(%o17)                                  0.0b0
(%i18) ulp : bfloat(2^(-57));
(%o18)                          6.938893903907228b-18
```

9.2.6 Effect of Floating Point Precision on Function Evaluation

Increasing the value of **fpprec** allows a more accurate numerical value to be found for the value of a function at some point. A simple function which allows one to find the absolute value of the change produced by increasing the value of **fpprec** has been presented by Richard Fateman.* This function is **uncert(f, arglist)**, in which **f** is a Maxima function, depending on one or more variables, and **arglist** is the n-dimensional point at which one wants the change in value of **f** produced by an increase of **fpprec** by **10**. This function returns a two element list consisting of the numerical value of the function at the requested point and also the absolute value of the difference induced by increasing the value of the current **fpprec** setting by the amount **10**.

We present here a version of Fateman's function which has an additional argument to control the amount of the increase of **fpprec**, and also has been simplified to accept only a function of one variable.

The function **fdf** is available in the Ch. 8 files **fdf.mac**, **qbromberg.mac**, **quad_ts.mac**, and **quad_de.mac**, and is defined by the code

```
fdf (%ff, %xx, %dfp) :=
   block([fv1,fv2,df],
    fv1:bfloat(%ff(bfloat(%xx))),
    block([fpprec:fpprec + %dfp ],
     fv2:  bfloat(%ff(bfloat(%xx))),
     df: abs(fv2 - fv1) ),
   [bfloat(fv2),bfloat(df)] )$
```

Here is an example of how this function can be used.

```
(%i1) (fpprintprec:8,load(fdf))$
(%i2) fpprec;
(%o2)                                   16
(%i3) g(x) := sin(x/2)$
```

*see his draft paper "Numerical Quadrature in a Symbolic/Numerical Setting", Oct. 16, 2008, available as the file **quad.pdf** in the folder: **http://www.cs.berkeley.edu/~fateman/papers/**

```
(%i4) fdf(g,1,10);
(%o4)                    [4.7942553b-1, 1.834924b-18]
(%i5) fdf(g,1,10),fpprec:30;
(%o5)                    [4.7942553b-1, 2.6824592b-33]
(%i6) fpprec;
(%o6)                                 16
```

In the first example, **fpprec** is **16**, and increasing the value to **26** produces a change in the function value of about $\mathbf{2 \times 10^{-18}}$. In the second example, **fpprec** is **30**, and increasing the value to **40** produces a change in the function value of about $\mathbf{3 \times 10^{-33}}$.

In the later section describing the "tanh-sinh" quadrature method, we will use this function for a heuristic estimate of the contribution of floating point errors to the approximate numerical value produced for an integral by that method.

9.3 Arbitrary Precision Quadrature with Maxima

9.3.1 Using bromberg for Arbitrary Precision Quadrature

A bigfloat version of the **romberg** function is defined in the file **`brmbrg.lisp`** located in **`share/numeric`**. You need to use **load(brmbrg)** or **load("brmbrg.lisp")** to use the function **bromberg**.

The use of **bromberg** is identical to the use of the **romberg** which we discussed in Chapter 8 (Numerical Integration), except that **rombergtol** (used for a relative error precision return) is replaced by the bigfloat **brombergtol** with a default value of **1.0b-4**, and **rombergabs** (used for an absolute error return) is replaced by the bigfloat **brombergabs** which has the default value **0.0b0**, and **rombergit** (which causes an return after halving the step size that many times) is replaced by the integer **brombergit** which has the default value **11**, and finally, **rombergmin** (the minimum number of halving iterations) is replaced by the integer **brombergmin** which has the default value **0**.

If the function being integrated has a magnitude of order one over the domain of integration, then an absolute error precision of a given size is approximately equivalent to a relative error precision of the same size. We will test **bromberg** using the function **exp(x)** over the domain **[-1, 1]**, and use only the absolute error precision parameter **brombergabs**, setting **brombergtol** to **0.0b0** so that the relative error test cannot be satisfied. Then the approximate value of the integral is returned when the absolute value of the change in value from one halving iteration to the next is less than the bigfloat number **brombergabs**.

We explore the use and behavior of **bromberg** for the simple integral $\int_{-1}^{1} \mathrm{e}^{\mathrm{x}}\,\mathrm{dx}$, binding a value accurate to 42 digits to **tval**, defining parameter values, calling **bromberg** first with **fpprec** equal to 30 together with **brombergabs** set to **1.0b-15** and find an actual error (compared with **tval**) of about $\mathbf{7 \times 10^{-24}}$.

```
(%i1) (fpprintprec:8,load(brmbrg));
(%o1) C:/PROGRA~1/MAXIMA~3.1/share/maxima/5.18.1/share/numeric/brmbrg.lisp
(%i2) [brombergtol,brombergabs,brombergit,brombergmin,fpprec,fpprintprec];
(%o2)                    [1.0b-4, 0.0b0, 11, 0, 16, 8]
(%i3) tval: bfloat(integrate(exp(x),x,-1,1)),fpprec:42;
(%o3)                              2.3504023b0
(%i4) fpprec;
(%o4)                                 16
```

```
(%i5) (brombergtol:0.0b0,brombergit:100)$
(%i6) b15:(brombergabs:1.0b-15,bromberg(exp(x),x,-1,1) ),fpprec:30;
(%o6)                              2.3504023b0
(%i7) abs(b15 - tval),fpprec:42;
(%o7)                             6.9167325b-24
(%i8) b20:(brombergabs:1.0b-20,bromberg(exp(x),x,-1,1) ),fpprec:30;
(%o8)                              2.3504023b0
(%i9) abs(b20 - tval),fpprec:42;
(%o9)                             1.5154761b-29
```

We see that, for the case of this test integral involving a well behaved integrand, the actual error of the result returned by **bromberg** is much smaller than the requested "difference error" supplied by the parameter **brombergabs**.

For later use, we define **qbromberg** (in a file **qbromberg.mac**) with the code:

```
qbromberg(%f,a,b,rprec,fp, itmax ) :=
        block([brombergtol,brombergabs,brombergit,
                    fpprec:fp ],
          if rprec > fp then
           ( print(" rprec should be less than fp "),
              return(done) ),
          brombergabs : bfloat(10^(-rprec)),
          brombergtol : 0.0b0,
          brombergit : itmax,
          bromberg(%f(x),x,a,b) )$
```

This function, with the syntax

```
    qbromberg ( f, a, b, rprec, fp, itmax )
```

uses the Maxima function **bromberg** to integrate the Maxima function **f** over the interval **[a, b]**, setting the local value of **fpprec** to **fp**, setting **brombergtol** to **0**, setting **brombergabs** to $10^{-\mathrm{rprec}}$, where **rprec** is called the "requested precision".

Here is a test of **qbromberg** for this simple integral.

```
(%i10) load(qbromberg)$
(%i11) qbr20 : qbromberg(exp,-1,1,20,40,100);
(%o11)                              2.3504023b0
(%i12) abs(qbr20 - tval);
(%o12)                                 0.0b0
(%i13) abs(qbr20 - tval),fpprec:40;
(%o13)                            1.0693013b-29
```

We have to be careful in the above step-by-step method to set **fpprec** to a large enough value to see the actual size of the error in the returned answer.

Instead of the work involved in the above step by step method, it is more convenient to define a function **qbrlist** which is passed a desired **fpprec** as well as a list of requested precision goals for **bromberg**. The function **qbrlist** then assumes a sufficiently accurate **tval** is globally defined, and proceeds through the list to calculate the **bromberg** value for each requested precision, computes the error in the result, and prints a line containing (rprec, fpprec, value, value-error). Here is the code for such a function, available in **qbromberg.mac**:

```
qbrlist(%f,a,b,rplist,fp,itmax) :=
  block([fpprec:fp,fpprintprec,brombergtol,
              brombergabs,brombergit,val,verr,pr],
     if not listp(rplist) then (print("rplist # list"),return(done)),
     brombergtol : 0.0b0,
     brombergit : itmax,
     fpprintprec:8,
     print(" rprec   fpprec     val              verr "),
     print(" "),
     for pr in rplist do
      ( brombergabs : bfloat(10^(-pr)),
        val: bromberg(%f(x),x,a,b),
        verr: abs(val - tval),
        print("  ",pr,"    ",fp,"    ",val,"    ",verr) ) )$
```

and here is an example of use of **qbrlist** in which the requested precision **rprec** (called **pr** in the code) is set to three different values supplied by the list **rplist** for each setting of **fpprec** used. We first define an accurate comparison value **tval**:

```
(%i1) (fpprintprec:8, load(brmbrg), load(qbromberg))$
(%i2) tval: bfloat(integrate(exp(x),x,-1,1)),fpprec:42;
(%o2)                              2.3504023b0
```

Here is our test for three different values of **fpprec**:

```
(%i3) qbrlist(exp,-1,1,[10,15,17 ],20,100)$
 rprec   fpprec     val              verr
   10      20     2.3504023b0     4.5259436b-14
   15      20     2.3504023b0     1.3552527b-20
   17      20     2.3504023b0     1.3552527b-20
(%i4) qbrlist(exp,-1,1,[10,20,27 ],30,100)$
 rprec   fpprec     val              verr
   10      30     2.3504023b0     4.5259437b-14
   20      30     2.3504023b0     1.4988357b-29
   27      30     2.3504023b0     5.5220263b-30
(%i5) qbrlist(exp,-1,1,[10,20,30,35],40,100)$
 rprec   fpprec     val              verr
   10      40     2.3504023b0     4.5259437b-14
   20      40     2.3504023b0     1.0693013b-29
   30      40     2.3504023b0     1.1938614b-39
   35      40     2.3504023b0     1.1938614b-39
```

We see that with **fpprec** equal to **40**, increasing **rprec** from **30** to **35** results in no improvement in the actual error of the result.

When bromberg Fails

If the integrand has end point algebraic and/or logarithmic singularities, **bromberg** may fail. Here is an example in which the integrand has a logarithmic singularity at the lower end point: $\int_0^1 \sqrt{t}\,\ln(t)\,dt$. The **integrate** function has no problem with this integral.

```
(%i6) g(x):= sqrt(x)*log(x)$
(%i7) integrate(g(t),t,0,1);
                                        4
(%o7)                                 - -
                                        9
(%i8) (load(brmbrg),load(qbromberg))$
(%i9) qbromberg(g,0,1,30,40,100);
log(0) has been generated.
#0: qbromberg(%f=g,a=0,b=1,rprec=30,fp=40,itmax=100)
 -- an error.  To debug this try debugmode(true);
```

You can instead use the tanh-sinh quadrature method for this integral (see Sec. 9.3.3).

9.3.2 A Double Exponential Quadrature Method for $a \le x < \infty$

This method (H. Takahasi and M. Mori, 1974; see Sec 9.3.3) is effective for integrands which contain a factor with some sort of exponential damping as the integration variable becomes large.

An integral of the form $\int_a^\infty g(y)\,dy$ can be converted into the integral $\int_0^\infty f(x)\,dx$ by making the change of variable of integration $y \to x$ given by $y = x + a$. Then $f(x) = g(x + a)$.

The double exponential method used here then converts the integral $\int_0^\infty f(x)\,dx$ into the integral $\int_{-\infty}^\infty F(u)\,du$ using a variable transformation $x \to u$:

$$x(u) = \exp(u - \exp(-u)) \tag{9.1}$$

and hence

$$F(u) = f(x(u))\,w(u), \qquad \text{where} \qquad w(u) = \frac{dx}{du} = \exp(-\exp(-u)) + x(u). \tag{9.2}$$

You can confirm that $x(0) = \exp(-1)$, $\quad w(0) = 2\,x(0)$ and that $x(-\infty) = 0$, $\quad x(\infty) = \infty$.

Because of the rapid decay of the integrand when the magnitude of u is large, one can approximate the value of the infinite domain u integral by using a trapezoidal numerical approximation with step size h using a modest number $(2N + 1)$ of function evaluations.

$$I(h, N) \simeq h \sum_{j=-N}^{N} F(u_j) \quad \text{where} \quad u_j = j\,h \tag{9.3}$$

This method is implemented in the Ch. 8 file **quad_de.mac**. We first demonstrate the available functions on the simple integral $\int_0^\infty e^{-x}\,dx = 1$.

```
(%i1) fpprintprec:8$
(%i2) g(x):= exp(-x)$
(%i3) tval : bfloat(integrate(g(x),x,0,inf)),fpprec:45;
(%o3)                                1.0b0
```

```
(%i4) load(quad_de);
(%o4)                          c:/work3/quad_de.mac
(%i5) quad_de(g,0,30,40);
(%o5)                        [1.0b0, 4, 4.8194669b-33]
(%i6) abs(first(%) - tval),fpprec:45;
(%o6)                               9.1835496b-41
```

The package function **quad_de(f, a, rp, fp)** integrates the Maxima function **f** over the domain $[x \geq a]$, using **fpprec : fp**, and returns a three element list when **vdiff** (the absolute value of the difference obtained for the integral in successive k levels) becomes less than or equal to 10^{-rp}. The parameter **rp** is called the "requested precision", and the value of **h** is repeatedly halved until the **vdiff** magnitude either satisfies this criterion or starts increasing. The first element is the appoximate value of the integral. The second element (4 above) is the "final k-level" used, where $h = 2^{-k}$. The third and last element is the final value of **vdiff**. We see in the above example that requesting precision $rp = 30$ and using floating point precision **fpprec : 40** results in an answer good to about **40** digits. This sort of accuracy is typical.

The package function **idek(f, a, k, fp)** integrates the Maxima function **f** over the domain $[a, \infty]$ using a "k-level approximation" with $h = 1/2^k$ and **fpprec : fp**.

```
(%i7) idek(g,0,4,40);
(%o7)                                  1.0b0
(%i8) abs(% - tval),fpprec:45;
(%o8)                               9.1835496b-41
```

The package function **idek_e(f, a, k, fp)** does the same calculation as **idek(f, a, k, fp)**, but returns both the approximate value of the integral and also a rough estimate of the amount of the error which is due to the floating point arithmetic precision being used. (The error of the approximation has three contributions: 1. the quadrature algorithm being used, 2. the step size **h** being used, and 3. the precision of the floating point arithmetic being used.)

```
(%i9) idek_e(g,0,4,40);
(%o9)                        [1.0b0, 8.3668155b-42]
(%i10) abs(first(%) - tval),fpprec:45;
(%o10)                              9.1835496b-41
```

The package function **ide(f, a, rp, fp)** follows the same path as **quad_de(f, a, rp, fp)**, but shows the progression toward success as the k level increases (and **h** decreases):

```
(%i11) ide(g,0,30,40);
      rprec =  30  fpprec =  40
 k     value            vdiff
 1    1.0b0
 2    1.0b0           4.9349774b-8
 3    9.9999999b-1    4.8428706b-16
 4    1.0b0           4.8194669b-33
```

The package function **ide_test(f, a, rp, fp)** follows the path of **ide(f, a, rp, fp)**, but adds to the table the value of the error of the approximate result for each **k** level attempted. The use of this function depends on an accurate value of the integral being bound to the global variable **tval**.

```
(%i12) ide_test(g,0,30,40);
      rprec =  30  fpprec =  40
 k     value           vdiff           verr
 1    1.0b0                          4.9349775b-8
 2    1.0b0          4.9349774b-8    4.8428706b-16
 3    9.9999999b-1   4.8428706b-16   4.8194668b-33
 4    1.0b0          4.8194669b-33   9.1835496b-41
```

Test Integral 1

Here we test this double exponential method code with the known integral

$$\int_0^\infty \frac{e^{-t}}{\sqrt{t}}\,dt = \sqrt{\pi} \tag{9.4}$$

```
(%i13) g(x):= exp(-x)/sqrt(x)$
(%i14) integrate(g(t),t,0,inf);
(%o14)                              sqrt(%pi)
(%i15) tval : bfloat(%),fpprec:45;
(%o15)                            1.7724538b0
(%i16) quad_de(g,0,30,40);
(%o16)                 [1.7724538b0, 4, 1.0443243b-34]
(%i17) abs(first(%) - tval),fpprec:45;
(%o17)                           1.8860005b-40
(%i18) idek_e(g,0,4,40);
(%o18)                  [1.7724538b0, 2.7054206b-41]
```

Again we see that the combination **rp = 30, fp = 40** leads to an answer good to about **40** digits of precision.

Test Integral 2

Our second known integral is

$$\int_0^\infty e^{-t^2/2}\,dt = \sqrt{\pi/2} \tag{9.5}$$

```
(%i19) g(x) := exp(-x^2/2)$
(%i20) tval : bfloat(sqrt(%pi/2)),fpprec:45$
(%i21) quad_de(g,0,30,40);
(%o21)                  [1.2533141b0, 5, 1.099771b-31]
(%i22) abs(first(%) - tval),fpprec:45;
(%o22)                           2.1838045b-40
(%i23) idek_e(g,0,5,40);
(%o23)                   [1.2533141b0, 1.3009564b-41]
```

Test Integral 3

Our third test integral is

$$\int_0^\infty \mathrm{e}^{-\mathrm{t}} \cos \mathrm{t}\, \mathrm{dt} = 1/2 \tag{9.6}$$

```
(%i24) g(x) := exp(-x)*cos(x)$
(%i25) integrate(g(x),x,0,inf);
                                       1
(%o25)                                 -
                                       2
(%i26) tval : bfloat(%),fpprec:45$
(%i27) quad_de(g,0,30,40);
(%o27)                    [5.0b-1, 5, 1.7998243b-33]
(%i28) abs(first(%) - tval),fpprec:45;
(%o28)                             9.1835496b-41
(%i29) idek_e(g,0,5,40);
(%o29)                      [5.0b-1, 9.8517724b-42]
```

9.3.3 The tanh-sinh Quadrature Method for $\mathrm{a} \le \mathrm{x} \le \mathrm{b}$

H. Takahasi and M. Mori (1974: see references at the end of this section) presented an efficient method for the numerical integration of the integral of a function over a finite domain. This method is known under the names "tanh-sinh method" and "double exponential method". This method can handle integrands which have algebraic and logarithmic end point singularities, and is well suited for use with arbitrary precision work.

Quoting (loosely) David Bailey's (see references below) slide presentations on this subject:

> The tanh-sinh quadrature method can accurately handle all "reasonable functions", even those with "blow-up singularities" or vertical slopes at the end points of the integration interval. In many cases, reducing the step size h by half doubles the number of correct digits in the result returned ("quadratic convergence").

An integral of the form $\int_{\mathrm{a}}^{\mathrm{b}} \mathrm{g}(\mathrm{y})\, \mathrm{dy}$ can be converted into the integral $\int_{-1}^{1} \mathrm{f}(\mathrm{x})\, \mathrm{dx}$ by making the change of variable of integration $\mathrm{y} \to \mathrm{x}$ given by $\mathrm{y} = \alpha\, \mathrm{x} + \beta$ with $\alpha = (\mathrm{b} - \mathrm{a})/2$ and $\beta = (\mathrm{a} + \mathrm{b})/2$. Then $\mathrm{f}(\mathrm{x}) = \alpha\, \mathrm{g}(\alpha\, \mathrm{x} + \beta)$.

The tanh-sinh method introduces a change of variables $\mathrm{x} \to \mathrm{u}$ which implies

$$\int_{-1}^{1} \mathrm{f}(\mathrm{x})\, \mathrm{dx} = \int_{-\infty}^{\infty} \mathrm{F}(\mathrm{u})\, \mathrm{du}. \tag{9.7}$$

The change of variables is expressed by

$$\mathrm{x}(\mathrm{u}) = \tanh\left(\frac{\pi}{2} \sinh \mathrm{u}\right) \tag{9.8}$$

and you can confirm that

$$\mathrm{u} = 0 \Rightarrow \mathrm{x} = 0, \qquad \mathrm{u} \to -\infty \Rightarrow \mathrm{x} \to -1, \mathrm{u} \to \infty \Rightarrow \mathrm{x} \to 1 \tag{9.9}$$

We also have $\mathrm{x}(-\mathrm{u}) = -\mathrm{x}(\mathrm{u})$.

The "weight" $\mathbf{w(u) = dx(u)/du}$ is

$$\mathbf{w(u) = \frac{\frac{\pi}{2}\cosh u}{\cosh^2\left(\frac{\pi}{2}\sinh u\right)}} \tag{9.10}$$

with the property $\mathbf{w(-u) = w(u)}$, in terms of which $\mathbf{F(u) = f(x(u)\,w(u)}$. Moreover, $\mathbf{F(u)}$ has "double exponential behavior" of the form

$$\mathbf{F(u) \approx \exp\left(-\frac{\pi}{2}\exp(\,|\,u\,|\,)\right)} \qquad \text{for} \quad \mathbf{u \to \pm\infty}. \tag{9.11}$$

Because of the rapid decay of the integrand when the magnitude of $\mathbf{u}$ is large, one can approximate the value of the infinite domain integral by using a trapezoidal numerical approximation with step size $\mathbf{h}$ using a modest number $\mathbf{(2\,N + 1)}$ of function evaluations.

$$\mathbf{I(h, N) \simeq h \sum_{j=-N}^{N} F(u_j)} \quad \text{where} \quad \mathbf{u_j = j\,h} \tag{9.12}$$

This method is implemented in the Ch.8 file **quad_ts.mac** and we will illustrate the available functions using the simple integral $\int_{-1}^{1} e^x\,dx$.

The package function **quad_ts (f, a, b, rp, fp)** is the most useful workhorse for routine use, and uses the tanh-sinh method to integrate the Maxima function **f** over the finite interval **[a, b]**, stopping when the absolute value of the difference $(\mathbf{I_k - I_{k-1}})$ is less than $\mathbf{10^{-rp}}$ (rp is the "requested precision" for the result), using **fp** digit precision arithmetic (**fpprec** set to **fp**, and **bfloat** being used to enforce this arithmetic precision). This function returns the list
[approx-value, k-level-used, abs(vdiff)], where the last element should be smaller than $\mathbf{10^{-rp}}$.

```
(%i1) fpprintprec:8$
(%i2) tval : bfloat( integrate( exp(x),x,-1,1 )),fpprec:45;
(%o2)                              2.3504023b0
(%i3) load(quad_ts);
   _kmax% =  8   _epsfac% =  2
(%o3)                         c:/work3/quad_ts.mac
(%i4) bfprint(tval,45)$
   number of digits =  45
   2.35040238728760291376476370119120163031143596b0
(%i5) quad_ts(exp,-1,1,30,40);
 construct _yw%[kk,fpprec] array for kk =  8  and fpprec =  40   ...working...
(%o5)                        [2.3504023b0, 5, 0.0b0]
(%i6) abs(first(%) - tval),fpprec:45;
(%o6)                             2.719612b-40
```

A value of the integral accurate to about 45 digits is bound to the symbol **tval**. The package function **bfprint(bf, fpp)** allows controlled printing of **fpp** digits of the "true value" **tval** to the screen. We then compare the approximate quadrature result with this "true value". The package **quad_ts.mac** defines two global parameters. **_kmax%** is the maximum "k-level" possible (the defined default is 8, which means the minimum step size for the transformed "u-integral" is $\mathbf{du = h = 1/2^8 = 1/256}$. The actual "k-level" needed to return a result with the requested precision **rp** is the integer in the second element of the returned list. The global parameter **_epsfac%** (default value **2**) is used to decide how many **(y, w)** numbers to pre-compute (see below).

We see that a "k-level" approximation with $\mathbf{k} = 5$ and $\mathbf{h} = 1/2^5 = 1/32$ returned an answer with an actual precision of about 40 digits (when $\mathbf{rp} = 30$ and $\mathbf{fp} = 40$).

The first time 40 digit precision arithmetic is called for, a set of (y, w) numbers are calculated and stored in an array which we call **_yw%[8, 40]**. The **y(u)** values will later be converted to **x(u)** numbers using high precision, and the original integrand function **f(x(u))** is also calculated at high precision. The **w(u)** numbers are what we call "weights", and are needed for the numbers $\mathbf{F}(\mathbf{u}) = \mathbf{f}(\mathbf{x}(\mathbf{u}))\,\mathbf{w}(\mathbf{u})$ used in the trapezoidal rule evaluation. The package precomputes pairs **(y, w)** for larger and larger values of **u** until the magnitude of the weight **w** becomes less than **eps**, where $\mathbf{eps} = 10^{-\mathbf{np}}$, where **n** is the global parameter **_epsfac%** (default **2**) and **p** is the requested floating point precision **fp**.

Once the set of **40-digit** precision **(y, w)** numbers have been "pre-computed", they can be used for the evaluation of any similar precision integrals later, since these numbers are independent of the actual function being integrated, but depend only on the nature of the tanh-sinh transformation being used.

The package function **qtsk(f, a, b, k, fp)** (note: arg **k** replaces **rp**) integrates the Maxima function **f** over the domain $[\mathbf{a}, \mathbf{b}]$ using a "k-level approximation" with $\mathbf{h} = 1/2^{\mathbf{k}}$ and **fpprec : fp**.

```
(%i7) qtsk(exp,-1,1,5,40);
(%o7)                              2.3504023b0
(%i8) abs(% - tval),fpprec:45;
(%o8)                             2.719612b-40
```

A heuristic value of the error contribution due to the arithmetic precision being used (which is separate from the error contribution due to the nature of the algorithm and the step size being used) can be found by using the package function **qtsk_e(f, a, b, k, fp);**. The first element of the returned list is the value of the integral, the second element of the returned list is a rough estimate of the contribution of the floating point arithmetic precision being used to the error of the returned answer.

```
(%i9) qtsk_e(exp,-1,1,5,40);
(%o9)                    [2.3504023b0, 2.0614559b-94]
(%i10) abs(first(%) - tval),fpprec:45;
(%o10)                            2.719612b-40
```

The very small estimate of the arithmetic precision contribution (two parts in 10^{94}) to the error of the answer is due to the high precision being used to convert from the pre-computed **y** to the needed abcissa **x** via $\mathbf{x} : \mathbf{bfloat}(1 - \mathbf{y})$ and the subsequent evaluation $\mathbf{f}(\mathbf{x})$. The precision being used depends on the size of the smallest **y** number, which will always be that appearing in the last element of the hashed array **_yw%[8, 40]**.

```
(%i11) last(_yw%[8,40]);
(%o11)                  [4.7024891b-83, 8.9481574b-81]
```

(In Eq. (9.12) we have separated out the **(u = 0, x = 0)** term, and used the symmetry properties $\mathbf{x}(-\mathbf{u}) = -\mathbf{x}(\mathbf{u})$, and $\mathbf{w}(-\mathbf{u}) = \mathbf{w}(\mathbf{u})$ to write the remainder as a sum over positive values of **u** (and hence positive values of **x**) so only the large **u** values of **y(u)** need to be pre-computed).

We see that the smallest **y** number is about 5×10^{-83} and if we subtract this from **1** we will get **1** unless we use a very high precision. It turns out that as **u** approaches plus infinity, **x** (as used here) approaches **b** (which is **1** in our example) from values less than **b**. Since a principal virtue of the tanh-sinh method is its ability to handle integrands which "blow up" at the limits of integration, we need to make sure we stay away (even if

only a little) from those end limits.

We can see the precision with which the arithmetic is being carried out in this crucial step by using the **fpxy(fp)** function

```
(%i12) fpxy(40)$
 the last y value =  4.7024891b-83
 the fpprec being used for x and f(x) is  93
```

and this explains the small number returned (as the second element) by **qtsk_e(exp, -1, 1, 5, 40);**.

The package function **qts(f, a, b, rp, fp)** follows the same path as **quad_ts(f, a, b, rp, fp)**, but shows the progression toward success as the **k** level increases (and h decreases):

```
(%i13) qts(exp,-1,1,30,40)$
  rprec =  30   fpprec =  40
  k       newval          vdiff
  1    2.350282b0
  2    2.3504023b0    1.2031242b-4
  3    2.3504023b0    8.136103b-11
  4    2.3504023b0    1.9907055b-23
  5    2.3504023b0    0.0b0
```

The package function **qts_test(f, a, b, rp, fp)** follows the path of **qts(f, a, b, rp, fp)**, but adds to the table the value of the error of the approximate result for each **k** level attempted. The use of this function depends on an accurate value of the integral being bound to the global variable **tval**.

```
(%i14) qts_test(exp,-1,1,30,40)$
      rprec =  30  fpprec =  40
 k     value           vdiff           verr
 1    2.350282b0                    1.2031234b-4
 2    2.3504023b0    1.2031242b-4    8.136103b-11
 3    2.3504023b0    8.136103b-11    1.9907055b-23
 4    2.3504023b0    1.9907055b-23   2.7550648b-40
 5    2.3504023b0    0.0b0           2.7550648b-40
```

Test Integral 1

Here we test this tanh-sinh method code with the known integral which confounded **bromberg** in Sec. 9.3.1 :

$$\int_0^1 \sqrt{t}\,\ln(t)\,dt = -4/9 \tag{9.13}$$

```
(%i15) g(x):= sqrt(x)*log(x)$
(%i16) tval : bfloat(integrate(g(t),t,0,1)),fpprec:45;
(%o16)                          - 4.4444444b-1
(%i17) quad_ts(g,0,1,30,40);
(%o17)                [- 4.4444444b-1, 5, 3.4438311b-41]
(%i18) abs(first(%) - tval),fpprec:45;
(%o18)                           4.4642216b-41
(%i19) qtsk_e(g,0,1,5,40);
(%o19)                  [- 4.4444444b-1, 7.6556481b-43]
```

Requesting thirty digit accuracy with forty digit arithmetic returns a value for this integral which has about forty digit precision. Note that “vdiff” is approximately the same as the actual absolute error.

Test Integral 2

Consider the integral

$$\int_0^1 \frac{\arctan(\sqrt{2+t^2})}{(1+t^2)\sqrt{2+t^2}}\,dt = 5\,\pi^2/96. \tag{9.14}$$

```
(%i20) g(x):= atan(sqrt(2+x^2))/(sqrt(2+x^2)*(1+x^2))$
(%i21) integrate(g(t),t,0,1);
                          1
                          /                2
                          [    atan(sqrt(t  + 2))
(%o21)                    I   ---------------------- dt
                          ]     2             2
                          /   (t  + 1) sqrt(t  + 2)
                           0
(%i22) quad_qags(g(t),t,0,1);
(%o22)                 [0.514042, 5.70701148E-15, 21, 0]
(%i23) float(5*%pi^2/96);
(%o23)                              0.514042
(%i24) tval: bfloat(5*%pi^2/96),fpprec:45;
(%o24)                           5.1404189b-1
(%i25) quad_ts(g,0,1,30,40);
(%o25)                 [5.1404189b-1, 5, 1.5634993b-36]
(%i26) abs(first(%) - tval),fpprec:45;
(%o26)                           7.3300521b-41
(%i27) qtsk_e(g,0,1,5,40);
(%o27)                   [5.1404189b-1, 1.3887835b-43]
```

Test Integral 3

We consider the integral

$$\int_0^1 \frac{\sqrt{t}}{\sqrt{1-t^2}}\,dt = 2\,\sqrt{\pi}\,\Gamma(3/4)/\Gamma(1/4) \tag{9.15}$$

```
(%i28) g(x):= sqrt(x)/sqrt(1 - x^2)$
(%i29) quad_qags(g(t),t,0,1);
(%o29)                [1.1981402, 8.67914629E-11, 567, 0]
(%i30) integrate(g(t),t,0,1);
                                     1  3
                                  beta(-, -)
                                     2  4
(%o30)                            ----------
                                      2
(%i31) tval : bfloat(%),fpprec:45;
(%o31)                            1.1981402b0
(%i32) quad_ts(g,0,1,30,40);
(%o32)                  [1.1981402b0, 5, 1.3775324b-40]
(%i33) abs(first(%) - tval),fpprec:45;
(%o33)                           1.8628161b-40
(%i34) qtsk_e(g,0,1,5,40);
(%o34)                   [1.1981402b0, 1.5833892b-45]
```

An alternative route to the "true value" is to convert **beta** to **gamma**'s using **makegamma**:

```
(%i35) makegamma(%o30);
                                                3
                                2 sqrt(%pi) gamma(-)
                                                4
(%o35)                          --------------------
                                          1
                                     gamma(-)
                                          4
(%i36) float(%);
(%o36)                            1.1981402
(%i37) bfloat(%o11),fpprec:45;
(%o37)                          1.1981402b0
```

Test Integral 4

We next consider the integral

$$\int_0^1 \ln^2 t \, dt = 2 \tag{9.16}$$

```
(%i38) g(x) := log(x)^2$
(%i39) integrate(g(t),t,0,1);
(%o39)                                2
(%i40) quad_ts(g,0,1,30,40);
(%o40)                       [2.0b0, 5, 0.0b0]
(%i41) abs( first(%) - bfloat(2) ),fpprec:45;
(%o41)                         1.8367099b-40
(%i42) qtsk_e(g,0,1,5,40);
(%o42)                   [2.0b0, 1.2570464b-42]
```

Test Integral 5

We finally consider the integral

$$\int_0^{\pi/2} \ln(\cos t) \, dt = -\pi \, \ln(2)/2 \tag{9.17}$$

```
(%i43) g(x) := log( cos(x) )$
(%i44) quad_qags(g(t),t,0,%pi/2);
(%o44)              [- 1.088793, 1.08801856E-14, 231, 0]
(%i45) integrate(g(t),t,0,%pi/2);
                                %pi
                                ---
                                 2
                               /
                               [
(%o45)                         I    log(cos(t)) dt
                               ]
                               /
                                0
```

```
(%i46) float(-%pi*log(2)/2);
(%o46)                              - 1.088793
(%i47) tval : bfloat(-%pi*log(2)/2),fpprec:45;
(%o47)                             - 1.088793b0
(%i48) quad_ts(g,0,%pi/2,30,40);
(%o48)          [1.2979374b-80 %i - 1.088793b0, 5, 9.1835496b-41]
(%i49) ans: realpart( first(%) );
(%o49)                             - 1.088793b0
(%i50) abs(ans - tval),fpprec:45;
(%o50)                             1.9661653b-40
(%i51) qtsk_e(g,0,%pi/2,5,40);
(%o51)           [1.2979374b-80 %i - 1.088793b0, 2.4128523b-42]
```

We see that the tanh-sinh result includes a tiny imaginary part due to bigfloat errors, and taking the real part produces an answer good to about 40 digits (using **`rp = 30, fp = 40`**).

References for the tanh-sinh Quadrature Method

This method was initially described in the article **Double Exponential Formulas for Numerical Integration**, by Hidetosi Takahasi and Masatake Mori, in the journal Publications of the Research Institute for Mathematical Sciences (Publ. RIMS), vol.9, Number 3, (1974), 721-741, Kyoto University, Japan. A recent summary by the second author is **Discovery of the Double Exponential Transformation and Its Developments**, by Masatake Mori, Publ. RIMS, vol.41, Number 4, (2005), 897-935. Both of the above articles can be downloaded from the Project Euclid RIMS webpage

```
http://projecteuclid.org/
              DPubS?service=UI&version=1.0&verb=Display&page=past&handle=euclid.prims
```

A good summary of implementation ideas can be found in the report **Tanh-Sinh High-Precision Quadrature**, by David H. Bailey, Jan. 2006, LBNL-60519, which can be downloaded from the webpage

```
  http://crd.lbl.gov/~dhbailey/dhbpapers/
```

Further Improvements for the tanh-sinh Quadrature Method

The code provided in the file **`quad_ts.mac`** has been lightly tested, and should be used with caution.

No proper investigation has been made of the efficiency of choosing to use a floating point precision (for all terms of the sum) based on the small size of the smallest $\mathbf{y}$ value.

No attempt has been made to translate into Lisp and compile the code to make timing trials for comparison purposes.

These (and other) refinements are left to the initiative of the hypothetical alert reader of limitless dedication (HAROLD).

9.3.4 The Gauss-Legendre Quadrature Method for $\mathbf{a \le x \le b}$

Loosely quoting from David Bailey's slide presentations (see references at the end of the previous section)

> The Gauss-Legendre quadrature method is an efficient method for continuous, well-behaved functions. In many cases, doubling the number of points at which the integrand is evaluated doubles the number of correct digits in the result. This method performs poorly for functions with algebraic and/or logarithmic end point singularities. The cost of computing the zeros of the Legendre polynomials and the corresponding "weights" increases as $\mathbf{n^2}$ and thus becomes impractical for use beyond a few hundred digits.

Since one can always make a change of integration variable from the domain **[a, b]** to the integration domain **[-1, 1]**, this method approximates an integral over **[-1, 1]** as the sum

$$\int_{-1}^{1} f(x)\,dx \approx \sum_{j=1}^{N} w_j\, f(x_j) \tag{9.18}$$

where the x_j are the roots of the N-th degree Legendre polynomial $P_N(x)$ on **[-1, 1]**, and the weights w_j are

$$w_j = \frac{-2}{(N+1)\,P'_N(x_j)\,P_{N+1}(x_j)} \tag{9.19}$$

This method is implemented with our Ch.9 package file **quad_gs.mac**, and Richard Fateman's lisp file:

http://www.cs.berkeley.edu/~fateman/generic/quad-maxima.lisp

which should be downloaded to use this package.

Except for the "driver" Maxima function **quad_gs(f,a,b,rp)**, all the Maxima functions in **quad_gs.mac** are functions defined by Fateman in either the comment section of the above lisp file, or in the file **http://www.cs.berkeley.edu/~fateman/papers/quadmax.mac**.

An introduction to arbitrary precision code and some background to the problem of arbitrary precision quadrature has been provided by Richard Fateman in his draft paper **Numerical Quadrature in a Symbolic/Numerical Setting**, **quad.pdf** (see Sec. 9.2.6).

We will illustrate the available functions using the simple integral $\int_{-1}^{1} e^x\,dx$.

The package function **gaussunit(f, N)** integrates the Maxima function **f** over the domain $[-1, 1]$ using N point Gauss-Legendre quadrature.

```
(%i1) load("quad-maxima.lisp");
(%o1)                          quad-maxima.lisp
(%i2) fpprintprec:8$
(%i3) tval : bfloat(integrate(exp(x),x,-1,1)),fpprec:45;
(%o3)                            2.3504023b0
(%i4) load(quad_gs);
(%o4)                       c:/work3/quad_gs.mac
(%i5) arrays;
(%o5)                           [ab_and_wts]
(%i6) arrayinfo(ab_and_wts);
(%o6)                            [hashed, 2]
(%i7) gaussunit(exp,4);
(%o7)                            2.350402b0
(%i8) abs(% - tval),fpprec:45;
(%o8)                           2.9513122b-7
(%i9) fpprec;
(%o9)                                16
(%i10) arrayinfo(ab_and_wts);
(%o10)                     [hashed, 2, [4, 16]]
(%i11) first( ab_and_wts[4, 16] );
(%o11)                 [8.6113631b-1, 3.3998104b-1]
(%i12) second( ab_and_wts[4, 16] );
(%o12)                 [3.4785484b-1, 6.5214515b-1]
```

```
(%i13) lp4 : legenp(4,x);
                                    4        2
                                35 x    15 x    3
(%o13)                          ----- - ----- + -
                                  8        4     8
(%i14) float(solve(lp4));
(%o14)   [x = - 0.861136, x = 0.861136, x = - 0.339981, x = 0.339981]
```

With the default value of **fpprec = 16** and using only four integrand evaluation points, the error is about 2 parts in 10^7. The first element of the two index hashed array **ab_and_wts[4,16]** is a list of the positive zeros of the fourth order Legendre polynomial $P_4(x)$ (calculated with the arithmetic precision **fpprec = 16**).

That fourth order Legendre polynomial $P_4(x)$ can be displayed with this package using **legenp(4, x)**. Using **solve** we see that the roots for negative x are simply the the positive roots with a minus sign, so the algorithm used makes use of this symmetry and keeps track of only the positive roots.

We can verify that the list of roots returned is correct to within the global floating point precision (we do this two different ways):

```
(%i15) lfp4(x) := legenp(4,x)$
(%i16) map('lfp4,%o11);
(%o16)                      [0.0b0, - 3.4694469b-18]
(%i17) map( lambda([z],legenp(4,z)),%o11 );
(%o17)                      [0.0b0, - 3.4694469b-18]
```

The second element of **ab_and_wts[4,16]** is a list of the weights which are associated with the positive roots (the negative roots have the same weights), with order corresponding to the order of the returned positive roots.

The package function **gaussunit_e(f, N)** does the same job as **gaussunit(f, N)**, but returns a rough estimate of the amount contributed to the error by the floating point precision used (as the second element of a list:

```
(%i18) gaussunit_e(exp,4);
(%o18)                    [2.350402b0, 1.2761299b-17]
```

We see that the error attributable to the floating point precision used is insignificant compared to the error due to the low number of integrand evaluation points for this example.

An arbitrary finite integration interval is allowed with the functions **gaussab(f, a, b, N)** and **gaussab_e(f, a, b, N)** which use **N** point Gauss-Legendre quadrature over the interval $[a, b]$, with the latter function being the analog of **gaussunit_e(f, N)**.

```
(%i19) gaussab(exp,-1,1,4);
(%o19)                           2.350402b0
(%i20) abs(% - tval),fpprec:45;
(%o20)                         2.9513122b-7
(%i21) gaussab_e(exp,-1,1,4);
(%o21)                    [2.350402b0, 1.2761299b-17]
```

The package function **quad_gs (f, a, b, rp)** integrates the Maxima function **f** over the finite interval $[a, b]$, successively doubling the number of integrand evaluation points, stopping when the absolute value of the difference $(I_n - I_{n/2})$ is less than 10^{-rp} (**rp** is the "requested precision" for the result), using the global setting of **fpprec** to use the corresponding precision arithmetic. We emphasize that Fateman's code uses a global setting of **fpprec** to achieve higher precision quadrature, rather than the method used in the previous two sections in which **fpprec** was set "locally" inside a **block**. This function returns the list **[approx-value, number-function-evaluations, abs(vdiff)]**, where the last element should be smaller than 10^{-rp}.

Here we test this function for **fpprec = 16, 30, and 40**.

```
(%i22) quad_gs(exp,-1,1,10);
                                    fpprec = 16

(%o22)                  [2.3504023b0, 20, 6.6613381b-16]
(%i23) abs(first(%) -tval),fpprec:45;
(%o23)                             6.2016267b-16
(%i24) fpprec:30$
(%i25) quad_gs(exp,-1,1,20);
                                    fpprec = 30

(%o25)                  [2.3504023b0, 20, 1.2162089b-24]
(%i26) abs(first(%) -tval),fpprec:45;
(%o26)                             2.2001783b-30
(%i27) fpprec:40$
(%i28) quad_gs(exp,-1,1,30);
                                    fpprec = 40

(%o28)                  [2.3504023b0, 40, 1.8367099b-40]
(%i29) abs(first(%) -tval),fpprec:45;
(%o29)                             2.7905177b-40
(%i30) gaussab_e(exp,-1,1,40);
(%o30)                    [2.3504023b0, 1.8492214b-41]
```

We have checked the contribution to the error due to the forty digit arithmetic precision used, with $N = 40$ point Gauss-Legendre quadrature (remember that **N** is the middle element of the list returned by **quad_gs (f, a, b, rp)** and is also the last argument of the function **gaussab_e(f, a, b, N)**.

We see that requesting **30** digit precision for the answer while using the global **fpprec** set to **40** results in an answer good to about **39** digits. Finally, let's check on what abscissae and weight arrays have been calculated so far:

```
(%i31) arrayinfo(ab_and_wts);
(%o31) [hashed, 2, [4, 16], [10, 16], [10, 30], [10, 40], [20, 16], [20, 30],
                                                           [20, 40], [40, 40]]
```

Using **quad_gs (f, a, b, rp)** with **fpprec = 16** led to the calculation of the abscissae and weight array for the index pairs **[10,16]** and **[20,16]** before the requested precision was achieved (the function always starts with **N = 10** point quadrature and then successively doubles that number until success is achieved).

Using **quad_gs (f, a, b, rp)** with **fpprec = 30** led to the calculation of the abscissae and weight array for the index pairs **[10,30]** and **[20,30]** before the requested precision was achieved.

Using **quad_gs (f, a, b, rp)** with **fpprec = 40** led to the calculation of the abscissae and weight array for the index pairs **[10,40]**, **[20,40]**, and **[40,40]** before the requested precision was achieved.

Finally, we have the function **quad_gs_table(f, a, b, rp)** which prints out a table showing the progression toward success:

```
(%i32) quad_gs_table(exp,-1,1,30)$
                                    fpprec = 40
       new val         N            vdiff
   2.3504023b0        10
   2.3504023b0        20        1.2162183b-24
   2.3504023b0        40        1.8367099b-40
```