Maxima by Example:
Ch.9: Bigfloats and High Accuracy Quadrature *

Edwin L. (Ted) Woollett

May 30, 2016

Contents

*This version uses **Maxima 5.36.1 SBCL**. This is a live document. Check `http://web.csulb.edu/~woollett/` for the latest version of these notes. Send comments and suggestions to `woollett@charter.net`

Preface

```
COPYING AND DISTRIBUTION POLICY
This document is part of the series 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.
```

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 (for example).
To load in your own file, such as bfloat.mac (used in this chapter),
using the brief version load(bfloat), you either need to place
bfloat.mac in one of the folders Maxima searches by default, or
else put a line like (this assumes your work folder is c:\work3):

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/bfloat.mac"),

We always use the brief load version in our examples, which are generated
using the Xmaxima graphics interface on a  computer using the Windows
operating system. Our maxima-init.mac file also contains display2d:false$,
which allows more information per XMaxima screen, and also allows one to
copy output and directly use it in a further step.
```

```
  Maxima, a Computer Algebra System.
  Some numerical results depend on the Lisp version used.
  This chapter uses Version 5.36.1 ( SBCL).
  http://maxima.sourceforge.net/
```

1 Introduction

This chapter is divided into two sections.

In the first section we experiment with the use of **bfloat**, including examples which also involve **fpprec**, **bfloatp**, **bfallroots**, **bf_find_root**, **fpprintprec**, **print**, and **printf**. The second section of Chapter 9 presents examples of the use of Maxima for high accuracy quadrature (numerical integration). (In Chapter 8, we gave numerous examples of numerical integration using the Maxima Quadpack functions such as **quad_qags** which use the default 16 digit floating point arithmetic. In addition, Ch. 8 includes the functions **apnint**, **apquad**, and **ndefint** which use bigfloat methods. The initial letters "ap" stand for "arbitrary precision"; perhaps a better prefix would be "hp" for "high precision" or "ha" for "high accuracy".)

Software files developed for Ch. 9 and available on the author's web page include:

1. bfloat.mac, 2. quad-maxima.lisp ,
3. quad_de.mac, 4. quad_ts.mac, 5. quad_gs.mac.

2 Bigfloat Numbers in Maxima

You can find the Lisp definition of the Maxima function **bfloat** (in Lisp, called **$bfloat**) in the source code file **...share/src/float.lisp**. **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, and the number of stored decimal digits will be equal to the current value of **fpprec** (default value 16). By default, trailing zeros of bigfloats are not printed to the screen.

```
(%i1) load(bfloat);
(%o1) "c:/work3/bfloat.mac"
(%i2) [fpprec,fpprintprec,display2d];
(%o2) [16,0,false]
(%i3) bfloat(1);
(%o3) 1.0b0
(%i4) bfdigits(%);
(%o4) 16
(%i5) bfloat(10^5);
(%o5) 1.0b5
(%i6) bfloat(10^-5);
(%o6) 1.0b-5
(%i7) bfdigits(%);
(%o7) 16
(%i8) bftrunc:false$
(%i9) bfloat(10^-5);
(%o9) 1.000000000000000b-5
(%i10) bfdigits(%);
(%o10) 16
(%i11) bftrunc:true$
(%i12) fpprec;
(%o12) 16
(%i13) q : 1.0b-5;
(%o13) 1.0b-5
(%i14) bfdigits(q);
(%o14) 16
(%i15) bftrunc:false$
(%i16) q;
(%o16) 1.000000000000000b-5
(%i17) bftrunc:true$
```

Here are three ways to enter the bigfloat equivalent of 0.1:

```
(%i1) load(bfloat);
(%o1) "c:/work3/bfloat.mac"
(%i2) [fpprec,fpprintprec];
(%o2) [16,0]
(%i3) q1 : bfloat(1/10);
(%o3) 1.0b-1
(%i4) q2 : 1b-1;
(%o4) 1.0b-1
(%i5) is(equal(q2,q1));
(%o5) true
(%i6) q3 : 10b0^(-1);
(%o6) 1.0b-1
(%i7) is(equal(q3,q1));
(%o7) true
(%i8) 10b0;
(%o8) 1.0b1
```

bfloatp(val) returns **true** if **val** is a bigfloat number, otherwise **false** is returned.

The number of digits in the resulting bigfloat is specified by the parameter **fpprec**, whose default value is **16**. The function **bfdigits**, defined in **bfloat.mac**, can be used to count the digits of a bigfloat. The setting of **fpprec** does not affect computations on ordinary 16 digit 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 **:lisp foobar** 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 will discuss some effects of the fact that computer arithmetic is carried out with binary bit representation of decimal numbers in Sec. 2.9.

2.1 Experimenting with Bigfloats

Mathematical functions in Maxima generally return floating-point results if given floating-point arguments, and bigfloat results if given bigfloat arguments. If a mathematical function does not return a bigfloat result when given a bigfloat argument, the function probably cannot return a bigfloat result, and one has available only the 16 digit floating point result given by **float** (or **expr,numer;** in interactive mode). Consider **sin(1)** and ordinary 16 digit floating point. **float(expr)** converts integers, rational numbers and bigfloats in **expr** to floating point numbers.

```
(%i1) sin(1);
(%o1) sin(1)
(%i2) sin(1.0);
(%o2) 0.8414709848078965
(%i3) sin(1),numer;
(%o3) 0.8414709848078965
(%i4) float(sin(1));
(%o4) 0.8414709848078965
```

Now consider **sin(1)** and the use of **bfloat**. The function **bfloat** uses the current value of the global parameter **fpprec** (which has the value 16 at normal startup) to return a bigfloat with the corresponding number of digits. You can temporarily override the current global value of **fpprec** (as shown below) without changing that global value.

```
(%i5) fpprec;
(%o5) 16
(%i6) q : 1.0b0;
(%o6) 1.0b0
(%i7) bfdigits(q);
(%o7) 16
(%i8) q1 : sin(q);
(%o8) 8.414709848078965b-1
(%i9) q2 : bfloat(sin(1));
(%o9) 8.414709848078965b-1
(%i10) map ('bfdigits,[q1,q2]);
(%o10) [16,16]
(%i11) is (equal (q1,q2));
(%o11) true
(%i12) q3 : bfloat(sin(1)), fpprec:20;
(%o12) 8.4147098480789650665b-1
(%i13) q4 : sin(1.0b0), fpprec : 20;
(%o13) 8.4147098480789650665b-1
(%i14) map ('bfdigits,[q3,q4]);
(%o14) [20,20]
(%i15) is (equal (q3,q4));
(%o15) true
(%i16) fpprec;
(%o16) 16
```

If we compute the absolute error of the 16 digit floating point approximation (which we call **sfp** below) to **sin(1)**, using as the "true value" the 20 digit approximation (**s20**), overriding the global value of **fpprec** temporarily as needed,

```
(%i17) s20 : bfloat (sin(1)), fpprec:20;
(%o17) 8.4147098480789650665b-1
(%i18) sfp : sin(1.0);
(%o18) 0.8414709848078965
(%i19) sfp_20 : bfloat(sfp),fpprec:20;
(%o19) 8.4147098480789650488b-1
(%i20) abs(sfp_20 - s20);
(%o20) 0.0b0
(%i21) abs(sfp_20 - s20), fpprec:20;
(%o21) 1.7770751233395221114b-18
(%i22) abs(sfp - s20), fpprec:20;
(%o22) 1.7770751233395221114b-18
```

we see that we need to do the absolute error calculation using 20 digit arithmetic, (as in step **%i21** above) to get a sensible answer, and also that we get the same absolute error without first converting the 16 digit floating point result **sfp** to 20 digits, as we did in step **%i19** above. The 20 digit bigfloat **sfp_20** is no more accurate than the 16 digit floating point number **sfp**. Input **%i22** involves one bigfloat number, **s20**, and "bigfloat contagion" occurs, which causes the floating point number **sfp** to be promoted to 20 bigfloat digits, by adding binary zeros, resulting in the decimal digits displayed in **%o19** above.

If we do a similar calculation with a **global value** of **fpprec** set at the start, we should get the same results. However, the calculation is easier to do with the global value of **fpprec** set to 20, since if at least one of the numbers involved is a bigfloat, all of the numbers are promoted to bigfloats ("bigfloat contagion"), and 20 digit arithmetic is then automatically used to get the result.

```
(%i23) fpprec;
(%o23) 16
(%i24) fpprec:20$
```

```
(%i25) sfp : sin(1.0);
(%o25) 0.8414709848078965
(%i26) sfp_20 : bfloat(sfp);
(%o26) 8.41470984807896504 88b-1
(%i27) s20 : bfloat (sin(1));
(%o27) 8.41470984807896506 65b-1
(%i28) abs(sfp_20 - s20);
(%o28) 1.77707512333952211 14b-18
(%i29) abs(sfp - s20);
(%o29) 1.77707512333952211 14b-18
```

Again we see that Maxima promotes **sfp** to 20 bigfloat digits automatically in this calculation, (a number having less digits is extended by **binary zeros** to the number of digits possessed by that bigfloat in the expression with the most digits), since we get the same answer whether or not we separately convert **sfp** to a bigfloat.

Retaining the global setting of **fpprec:20**, we now try to get both floating point and bigfloats from two Maxima functions, **erf(z)** (the error function) and one of the Bessel functions **bessel_j(nu,z)**. We find that the former will return a 20 digit bigfloat, but the latter cannot.

```
(%i30) fpprec;
(%o30) 20
(%i31) erf(1.0);
(%o31) 0.8427007929497148
(%i32) erf(1.0b0);
(%o32) 8.42700792949714869 34b-1
(%i33) bessel_j(0,1.0);
(%o33) 0.7651976865579666
(%i34) bessel_j(0,1.0b0);
(%o34) bessel_j(0,1.0b0)
```

If you use the Index of the Maxima Manual, and type **bf**, you get links to the definitions of the following functions:

```
   bf_find_root   bf_fmin_cobyla   bfallroots   bffac
   bfhzeta   bfloat   bfloatp   bfpsi   bfpsi0
   bftorat   bftrunc
```

The function **bfloatp** returns either **true** or **false**, and the function **bftrunc** (default value **true**) surpresses the printing out of trailing decimal digits with the value **0** by default. We are starting this section with the default value **fpprec = 16**, and assigning a bigfloat value to the symbol **b1**, using 3 significant digits. Maxima then automatically creates a 16 (decimal) digit value which you can see by setting **bftrunc** to the value **false** (ie., don't truncate bigfloat numbers having "trailing zeros"). You can also count the number of bigfloat digits easily using our function **bfdigits** (available in our code file **bfloat.mac**), as shown below. In normal work, it is convenient to surpress the sight of trailing zeros in bigfloats.

```
(%i1) b1 : 1.57b0;
(%o1) 1.57b0
(%i2) load(bfloat);
(%o2) "c:/work3/bfloat.mac"
(%i3) bfdigits(b1);
(%o3) 16
(%i4) bfloatp(b1);
(%o4) true
(%i5) bftrunc;
(%o5) true
(%i6) bftrunc : false$
(%i7) b1;
(%o7) 1.57000000000000 00b0
(%i8) fpprec;
(%o8) 16
(%i9) fpprintprec;
(%o9) 0
```

```
(%i10) floatnump(b1);
(%o10) false
(%i11) bfdigits(1.000000000000000b0);
(%o11) 16
```

The description of **bftrunc** from the Maxima Manual:

```
Option variable: bftrunc
Default value: true
bftrunc causes trailing zeroes in non-zero bigfloat numbers
 not to be displayed. Thus, if bftrunc is false, bfloat (1)
 displays as 1.000000000000000b0.
 Otherwise, this is displayed as 1.0b0.
```

Next consider obtaining numerical values of **complex expressions**. First, 16 digit numbers using **float**. Our first attempt does not succeed:

```
(%i12) float(exp(%i*%pi/sin(1)));
(%o12) 2.718281828459045^((%i*%pi)/sin(1))
```

Here are two paths to what we want:

```
(%i13) float(exp(%i*%pi/sin(1))),numer;
(%o13) (-0.5579061738629681*%i)-0.8299040312985494
(%i14) float( rectform (exp(%i*%pi/sin(1))));
(%o14) (-0.5579061738629681*%i)-0.8299040312985494
```

The bigfloat value is easier to get:

```
(%i15) fpprec;
(%o15) 16
(%i16) q : bfloat(exp(%i*%pi/sin(1)));
(%o16) (-5.57906173862968b-1*%i)-8.299040312985495b-1
(%i17) bfloatp(q);
(%o17) false
(%i18) realpart(q);
(%o18) -8.299040312985495b-1
(%i19) imagpart(q);
(%o19) -5.57906173862968b-1
(%i20) imagpart(1.2b0);
(%o20) 0
```

Note that **bfloat** of a complex expression does not result in another bigfloat (as indicated by the return value of **bfloatp**), because of the presence of the symbol **%i**. In **bfloat.mac** the function **bfdigits** is defined which gets around this difficulty by always working with the real part (or, if necessary, the imaginary part) of an expression:

```
(%i1) load(bfloat);
(%o1) "c:/work3/bfloat.mac"
(%i2) q : bfloat(exp(%i*%pi/sin(1)));
(%o2) (-5.57906173862968b-1*%i)-8.299040312985495b-1
(%i3) bfdigits(q);
(%o3) 16
(%i4) q : bfloat(exp(%i*%pi/sin(1))), fpprec:20;
(%o4) (-5.5790617386296779922b-1*%i)-8.2990403129854944112b-1
(%i5) bfdigits(q);
(%o5) 20
```

The default value of **bftrunc** (which is **true**) surpresses the printing of trailing (decimal) zeros, but **bfdigits** still gives the correct number of decimal digits.

```
(%i6) [fpprec,fpprintprec,bftrunc];
(%o6) [16,0,true]
(%i7) q : 1.0b0;
(%o7) 1.0b0
(%i8) bfdigits(q);
(%o8) 16
(%i9) bftrunc:false$
(%i10) q;
(%o10) 1.000000000000000b0
(%i11) bftrunc:true$
(%i12) q : bfloat(1),fpprec:20;
(%o12) 1.0b0
(%i13) bfdigits(q);
(%o13) 20
```

CAUTION ... With **q** defined above as a 20 digit number, and with the global value of **fpprec** set to the default value of 16, if we define **q1 : -q**, the result is a 16 digit number, and if we define **q2 : abs(q)**, the result is also a 16 digit number.

```
(%i14) fpprec;
(%o14) 16
(%i15) q1 : -q;
(%o15) -1.0b0
(%i16) bfdigits(q1);
(%o16) 16
(%i17) q2 : abs(q);
(%o17) 1.0b0
(%i18) bfdigits(q2);
(%o18) 16
```

This occurred because the global setting of the parameter **fpprec** was 16. You can either change that global setting, or use a local temporary higher setting as in:

```
(%i19) q5 : -q, fpprec:20;
(%o19) -1.0b0
(%i20) bfdigits(q5);
(%o20) 20
(%i21) q6 : abs(q), fpprec:20;
(%o21) 1.0b0
(%i22) bfdigits(q6);
(%o22) 20
```

Again with **q** a 20 digit number defined above, defining **q3 : sin(q)** in a global environment in which **fpprec = 16**, results in a 16 digit number.

```
(%i23) q3 : sin(q);
(%o23) 8.414709848078965b-1
(%i24) bfdigits(q3);
(%o24) 16
(%i25) q4 : sin(q), fpprec:20;
(%o25) 8.4147098480789650665b-1
(%i26) bfdigits(q4);
(%o26) 20
```

2.2 Bigfloat Numbers and fpprintprec

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 when calling **disp**, **display**, **print**, and **printf**.

For bigfloat numbers, when **fpprintprec** has a value greater than **2** and less than **fpprec**, the number of digits printed is equal to **fpprintprec**. If **fpprintprec = 0** then all the bigfloat digits are printed. **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.

We can use local temporary values of **`fpprintprec`** and **`fpprec`** in interactive work.

```
(%i1) load(bfloat);
(%o1) "c:/work3/bfloat.mac"
(%i2) [fpprec,fpprintprec];
(%o2) [16,0]
(%i3) q : integrate(exp(x),x,-1,1);
(%o3) %e-%e^-1
(%i4) q : bfloat(q),fpprec : 45;
(%o4) 2.350402387287602913764763701191201630311435 96b0
(%i5) bfdigits(q);
(%o5) 45
(%i6) print (q),fpprintprec : 12$
2.35040238728b0
(%i7) fpprec;
(%o7) 16
(%i8) print (q),fpprintprec : 15$
2.3504023872876b0
(%i9) print (q),fpprintprec : 16$
2.350402387287602913764763701191201630311435 96b0
(%i10) print (q),fpprintprec : 17$
2.350402387287602913764763701191201630311435 96b0
(%i11) print (q),fpprec:18, fpprintprec : 17$
2.3504023872876029b0
(%i12) print (q),fpprec:19, fpprintprec : 18$
2.35040238728760291b0
(%i13) fpprintprec;
(%o13) 0
(%i14) print(q)$
2.350402387287602913764763701191201630311435 96b0
```

Either or both of the parameters **fpprec** and **fpprintprec** can be used as local variables inside a function or in an assignment statement defined using **block**, and set to local value(s) which do not affect the global setting. If used in a function, such locally defined values of **fpprec** and **fpprintprec** govern 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.

Here is a simple pedagogical example of using **`block`** to produce an assignment statement (not a function). (Recall that our **`maxima-init.mac`** file has the line **`display2d:false`**).

```
(%i15) [fpprec,fpprintprec];
(%o15) [16,0]
(%i16) piby2 : block([fpprintprec,fpprec:30,val],
                val:bfloat(%pi/2),
                fpprintprec:8,
                disp (val),
                display (val),
                print("  val =   ",val),
                print("  bfdigits(val) = ", bfdigits(val) ),
                val);
1.5707963b0

val = 1.5707963b0

  val =    1.5707963b0
  bfdigits(val) =  30
(%o16) 1.57079632679489661923132169164b0
(%i17) bfdigits(piby2);
(%o17) 30
(%i18) [fpprec,fpprintprec];
(%o18) [16,0]
```

An alternative assignment statement which does not use **block** but does use local temporary values of **fpprintprec** and **fpprec** (and has the defect of introducing a global value for the symbol **val**) is:

```
(%i19) piby2 : (val:bfloat(%pi/2),
                disp (val),
                display (val),
                print("  val =   ",val),
                print("  bfdigits(val) = ", bfdigits(val) ),val ), fpprintprec:8, fpprec:30;
1.5707963b0

val = 1.5707963b0

  val =    1.5707963b0
  bfdigits(val) =  30
(%o19) 1.57079632679489661923132169164b0
(%i20) piby2;
(%o20) 1.57079632679489661923132169164b0
(%i21) bfdigits(piby2);
(%o21) 30
(%i22) val;
(%o22) 1.57079632679489661923132169164b0
```

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

```
(%i23) f2(w) := block([v2 ],
                  print ("   in function f2 "),
                  display([w,fpprec,fpprintprec]),
                  print ("  bfdigits(w) = ", bfdigits(w)),
                  v2 : sin(w),
                  print (" v2 = sin(w) =  ", v2),
                  print ("     bfdigits(v2) = ", bfdigits(v2) ),
                  print("  "),
                  v2 )$
(%i24) f1(x, fpp, fpprt) := block([fpprintprec,fpprec:fpp,v1],
                  fpprintprec:fpprt,
                  display([x,fpprec,fpprintprec]),
                  print("  in function f1, call f2(x) "),
                  v1 : f2(bfloat(x))^2,
                  print(" back  in f1, v1 = sin(x)^2 =  ",v1),
                  print(" bfdigits(v1) = ",bfdigits(v1)),
                  v1 )$
```

And here we call **f1** with values **x = 0.5, fpp = 30, fpprt = 8**:

```
(%i25) q : f1(0.5,30,8);
[x,fpprec,fpprintprec] = [0.5,30,8]

  in function f1, call f2(x)
   in function f2
[w,fpprec,fpprintprec] = [5.0b-1,30,8]

  bfdigits(w) =  30
 v2 = sin(w) =   4.7942553b-1
     bfdigits(v2) =  30

 back  in f1, v1 = sin(x)^2 =   2.2984884b-1
 bfdigits(v1) =  30
(%o25) 2.29848847065930141299531696279b-1
(%i26) q;
(%o26) 2.29848847065930141299531696279b-1
(%i27) bfdigits(q);
(%o27) 30
(%i28) [fpprec,fpprintprec];
(%o28) [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.

```
(%i29) fpprec:20$
(%i30) q : bfloat(2);
(%o30) 2.0b0
(%i31) bfdigits(q);
(%o31) 20
(%i32) q1 : 1 + q;
(%o32) 3.0b0
(%i33) bfdigits(q1);
(%o33) 20
(%i34) q2 : 1.0 + q;
(%o34) 3.0b0
(%i35) bfdigits(q2);
(%o35) 20
(%i36) q3 : sin(q2);
(%o36) 1.411200080598672221b-1
(%i37) bfdigits(q3);
(%o37) 20
```

The Maxima symbol **%pi** is not automatically converted to a numerical value by contagion and an extra use of **bfloat** does the conversion.

```
(%i38) a : 2.38b-1$
(%i39) a;
(%o39) 2.38b-1
(%i40) bfdigits(a);
(%o40) 20
(%i41) q4 : exp(%pi*a);
(%o41) %e^(2.38b-1*%pi)
(%i42) q4 : bfloat(q4);
(%o42) 2.1121345085033606761b0
(%i43) bfdigits(q4);
(%o43) 20
```

The following example illustrates how the setting of **fpprec** can affect what is printed to the screen when doing bigfloat arithmetic.

```
(%i44) fpprec;
(%o44) 20
(%i45) q1 : bfloat(1);
(%o45) 1.0b0
(%i46) bfdigits(q1);
(%o46) 20
(%i47) q2 : bfloat(10^(-25));
(%o47) 1.0b-25
(%i48) bfdigits(q2);
(%o48) 20
(%i49) q3 : q1 + q2;
(%o49) 1.0b0
(%i50) q3, fpprec:25;
(%o50) 1.0b0
(%i51) q3, fpprec:26;
(%o51) 1.0b0
(%i52) q3 : q1 + q2,fpprec:26;
(%o52) 1.00000000000000000000000001b0
(%i53) bfdigits(q3);
(%o53) 26
```

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.

2.3 Using print with Bigfloats

In Sec. 2.2 we described the relations between the settings of **fpprec** and **fpprintprec** and we repeat some of that discussion here. 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 **`q`** to have **`fpprec = 45`** digits of precision:

```
(%i1) load(bfloat);
(%o1) "c:/work3/bfloat.mac"
(%i2) [fpprec,fpprintprec];
(%o2) [16,0]
(%i3) integrate(exp(x),x,-1,1);
(%o3) %e-%e^-1
(%i4) q : bfloat(%),fpprec : 45;
(%o4) 2.35040238728760291376476370119120163031143596b0
(%i5) bfdigits(q);
(%o5) 45
```

We then use **`print`** with **`fpprintprec`** to get increasing numbers of digits on the screen. When the temporary value of **`fpprintprec`** is equal to the global value of **`fpprec`** (here 16), all 45 digits are printed out. To get only 16 digits printed to the screen, you need to set a local temporary value of **`fpprec`** greater than or equal to 17.

```
(%i6) print(q),fpprintprec:5$
2.3504b0
(%i7) print(q),fpprintprec:15$
2.3504023872876b0
(%i8) print(q),fpprintprec:16$
2.35040238728760291376476370119120163031143596b0
(%i9) print(q),fpprintprec:0$
2.35040238728760291376476370119120163031143596b0
(%i10) print(q),fpprec:21,fpprintprec:20$
2.3504023872876029137b0
(%i11) print(q),fpprec:17,fpprintprec:16$
2.350402387287602b0
(%i12) [fpprec,fpprintprec];
(%o12) [16,0]
```

A function **`bfprint`**, defined in **`bfloat.mac`** with the code

```
bfprint(smsg,xbf,fpp) :=
  block([fpprec, fpprintprec ],
        fpprec : fpp + 1,
        fpprintprec : fpp,
        print (smsg, xbf))$
```

can be used to implement this idea:

```
(%i13) bfprint(" q16 = ",q,16)$
 q16 =  2.350402387287602b0
(%i14) bfprint(" q20 = ",q,20)$
 q20 =  2.350402387287602913 7b0
```

2.4 Using printf with Bigfloats

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

```
(%i15) q : bfloat(exp(-20)),fpprec : 30;
(%o15) 2.06115362243855782796594038016b-9
(%i16) bfdigits(q);
(%o16) 30
(%i17) printf(true,"~d~a",3,string(q))$
32.06115362243855782796594038016b-9
(%i18) printf(true,"  ~d    ~a",3,string(q))$
  3    2.06115362243855782796594038016b-9
(%i19) (printf(true,"  ~d    ~a~%",3,string(q)),
                  printf(true,"  ~d    ~a",3,string(q)))$
  3    2.06115362243855782796594038016b-9
  3    2.06115362243855782796594038016b-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 **q**, which required that **q** 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** in our first attempt, we get **32.0...** instead of **3 2.0...**.

To control the number of significant figures displayed, we use **fpprintprec**:

```
(%i20) printf(true,"  ~d    ~a",3,string(q)), fpprintprec:8$
  3    2.0611536b-9
```

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

```
(%i21) printf(true,"  ~d    ~f",3,q)$
  3    0.000000002061153622438558
(%i22) printf(true,"  ~d    ~e",3,q)$
  3    2.061153622438558e-9
(%i23) printf(true,"  ~d    ~h",3,q)$
  3    0.00000000206115362243855782796594038016
(%i24) printf(true,"  ~d    ~h",3,q),fpprintprec : 12$
  3    0.0000000206115362243
```

2.5 A Table of Bigfloats using printf

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

```
(%i25) 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) ) ))$
(%i26) 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. Many examples of **printf** can be found in the Maxima manual under the index listing **printf** and many more in **..../share/stringproc/rtestprintf.mac**.

We can use **printf** (for headings and space) instead of **print** with an alternative version.

```
(%i27) 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**:

```
(%i28) print_test2(30)$
fpprec = 30


  k          value

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

2.6 Adding Bigfloats Having Differing Accuracy

If **A** and **B** are bigfloats with different accuracy, the accuracy (correct digits) of the sum **(A + B)** is the accuracy (correct digits) of the least accurate number. As an example, let **pi50** be an approximation to π accurate to 50 digits, and let **pi100** be an approximation to π accurate to 100 digits. The sum, **pisum**, is only accurate to 50 digits, as seen by a comparison with the value of $2\,\pi$ calculated using 100 digit arithmetic (**twopi**) which is taken as the "true value" when calculating absolute error.

```
(%i1) fpprintprec:8$
(%i2) fpprec;
(%o2) 16
(%i3) pi100 : bfloat(%pi),fpprec:100;
(%o3) 3.1415926b0
(%i4) pi50 : bfloat(%pi),fpprec:50;
(%o4) 3.1415926b0
(%i5) abs(pi50 - pi100),fpprec:101;
(%o5) 1.0106957b-51
(%i6) twopi : bfloat(2*%pi),fpprec:100;
(%o6) 6.2831853b0
(%i7) pisum : pi50 + pi100,fpprec:100;
(%o7) 6.2831853b0
(%i8) pisum - twopi,fpprec:101;
(%o8) 1.0106957b-51
```

2.7 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** is in the file **.../share/src/cpoly.lisp**.

Our example is a cubic equation whose three degenerate roots are simply π. We first compute a 50 digit approximation to the true root.

```
(%i1) load(bfloat);
(%o1) "c:/work3/bfloat.mac"
(%i2) fpprec;
(%o2) 16
(%i3) pi50 : bfloat(%pi),fpprec:50;
(%o3) 3.1415926535897932384626433832795028841971693993751b0
(%i4) bfdigits(pi50);
(%o4) 50
```

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

```
(%i5) e : expand ( (x-%pi)^3);
(%o5) 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) e16 : float(e);
(%o6) x^3-9.42477796076938*x^2+29.60881320326807*x-31.00627668029982
```

Now find the approximate roots of this numerical polynomial in **x** using **allroots**. We remind the reader of the Maxima Manual description of **allroots**:

```
allroots (eqn)
Computes numerical approximations of the real and complex roots of the
 polynomial expr or polynomial equation eqn of one variable.

The flag polyfactor when true causes allroots to factor the polynomial over
  the real numbers if the polynomial is real, or over the complex numbers,
  if the polynomial is complex.

allroots may give inaccurate results in case of multiple roots. If the polynomial
  is real, allroots (%i*p) may yield more accurate approximations than allroots (p),
  as allroots invokes a different algorithm in that case.
```

Following the manual's suggestion,

```
(%i7) sar16 : map ('rhs, allroots (%i*e16));
(%o7) [3.14159265358979-1.887379141862766e-15*%i,
       9.992007221626409e-16*%i+3.141592653589795,
       8.881784197001252e-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). We are using the do loop version **for s in alist do...**.

```
(%i8) for s in sar16 do disp( expand (subst (s, x, e16)))$
6.310887241768094e-30*%i

-6.310887241768094e-30*%i

-3.155443620884047e-30*%i
```

which is very good root behavior. The appearance of **%i** is due to our asking **allroots** for the roots of **%i*e15**:

```
(%i9) for s in sar16 do disp( expand (subst (s, x, %i*e16)))$
-6.310887241768094e-30

6.310887241768094e-30

3.155443620884047e-30
```

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

```
(%i10) 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 16 digit precision, we try out **bfallroots**. The Manual has the description

```
bfallroots (eqn)
Computes numerical approximations of the real and complex roots
    of the polynomial expr or polynomial equation eqn of one variable.

In all respects, bfallroots is identical to allroots except that bfallroots
 computes the roots using bigfloats. See allroots for more information
```

```
(%i11) sbfar16 : map ('rhs, bfallroots (%i*e16));
(%o11) [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:

```
(%i12) for s in sbfar16 do  disp( expand (subst (s,x,e16)))$
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".

```
(%i13) 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.

```
(%i14) e;
(%o14) x^3-3*%pi*x^2+3*%pi^2*x-%pi^3
(%i15) fpprec:40$
(%i16) e40 : bfloat(e);
(%o16) x^3-9.42477796076937971538793014983850865259 2b0*x^2
          +2.96088132032680758565034729996284534059 4b1*x
          -3.10062766802998201754763150671013952022 3b1
```

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

Now we seek the roots using **bfallroots**, using 40 digit arithmetic. The global parameter **bftorat** by default has the value **false**, which causes the parameter **ratepsilon** (default value = **2.0e-15**) to govern the tolerance for the conversion of floats and bigfloats to rational numbers. The code file **bfloat.mac** has the line **ratprint:false**, so we are restoring the Maxima default (**true**) in order to monitor the conversion of floats and bigfloats to rational numbers here.

```
(%i17) bftorat;
(%o17) false
(%i18) ratprint:true$
(%i19) ratepsilon;
(%o19) 2.0e-15
(%i20) sbfar40 : map ('rhs, bfallroots (%i*e40));
'rat' replaced -3.10062766802998201754763150671013952022 3B1
        by -314736901/10150748 = -3.10062766802998163288065076583518771 2275B1
'rat' replaced 2.96088132032680758565034729996284534059 4B1
     by 81466873/2751440 = 2.96088132032681068822144040938563079696 5B1
'rat' replaced -9.42477796076937971538793014983850865259 2B0
      by -245850922/26085593 = -9.42477796076937948084983155261220245213 5B0
(%o20) [4.06783676915217150269117024049013599924 2b-5*%i
         +3.14161613902569635620081376662257120350 2b0,
        3.14154568271798676844820401936706004245 b0
         -1.54800931201080803327142969725461434101 7b-36*%i,
        3.14161613902569635620081376662257120618 3b0
         -4.06783676915217150269117024048998119831 1b-5*%i]
```

Check the 40 digit expression using these roots

```
(%i21) for s in sbfar40 do disp (expand  (subst (s,x,e40)))$
(-1.32202125274312868406372503721079185291 8b-18*%i)
 -1.03632376852386467494669744515246912335 1b-13

(-1.02459460693477637348099139928252243115 2b-44*%i)
 -1.03630087068446744746541254601202292767 5b-13

1.32202125274312868406372503721079185291 8b-18*%i
 -1.03632376852386467494669740841827066015 4b-13
```

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

```
(%i22) for s in sbfar40 do disp (pi50 - realpart(s))$
-2.34854359031177381703833430683193047550 6b-5

4.69708718064700144393639124428417470138 3b-5

-2.34854359031177381703833430683219858005 3b-5
```

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 replacement above and its difference from the actual 40 digit accurate number (because **fpprec** now has the global value 40, we don't need to wrap the bigfloat arithmetic with **bfloat**):

```
(%i23) 9.42477796076937971538793014983850865259 2b0 -
                   9.42477796076937948084983155261220245213 5b0;
(%o23) 2.34538098597226306200457284757742668342 5b-16
(%i24) bfloat (9.42477796076937971538793014983850865259 2b0 -
                  9.42477796076937948084983155261220245213 5b0);
(%o24) 2.34538098597226306200457284757742668342 5b-16
```

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**.

```
(%i25) ratepsilon : 1.0e-41$
(%i26) sbfar40 : map ('rhs, bfallroots (%i*e40));
'rat' replaced -3.1006276680299820175476315067101395202 23B1
         by -689775162029634828708/22246307389364524529 =
                -3.1006276680299820175476315067101395202 23B1
'rat' replaced 2.960881320326807585650347299962845340594B1
   by 109443032496771648040 9/36962991979932468848 =
   2.960881320326807585650347299962845340594B1
'rat' replaced -9.42477796076937971538793014983850865259 2B0
   by -787357891006146598194/83541266890691994833 =
   -9.42477796076937971538793014983850865259 2B0
(%o26) [3.14159265358979323846264338327950288419 7b0,
        3.14159265358979323846264338327950288419 2b0
         -2.06629866355480226010129469433597873054 1b-40*%i,
        2.06629866355480226010129469433597873054 1b-40*%i
         +3.14159265358979323846264338327950288420 3b0]
(%i27) for s in sbfar40 do disp (expand  (subst (s,x,e40)))$
2.20405190779178907744138100729171064591b-39

0.0b0

7.34683969263929692480460335763903548636 7b-40
(%i28) 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.

An alternative to increasing the size of **ratepsilon** to match **fpprec** is to set **bftorat** to **true** with almost the same good results as increasing **ratepsilon** from its default value. Recall that we have set the global value **fpprec:40** above, and we restore **ratepsilon** to its low default value.

```
(%i29) fpprec;
(%o29) 40
(%i30) ratepsilon:2.0e-15;
(%o30) 2.0e-15
(%i31) bftorat:true$
(%i32) sbfar40 : map ('rhs, bfallroots (%i*e40));
'rat' replaced -3.1006276680299820175476315067101395202 23B1
      by -689775162029634828708/22246307389364524529 =
      -3.1006276680299820175476315067101395202 23B1
'rat' replaced 2.960881320326807585650347299962845340594B1
       by 109443032496771648040 9/36962991979932468848
           = 2.960881320326807585650347299962845340594B1
'rat' replaced -9.42477796076937971538793014983850865259 2B0
        by -265099323460521503743/28127911826039502680
                = -9.42477796076937971538793014983850865259 1B0
(%o32) [3.14159265358979323846264338327950288417 3b0
         -5.48717089543997489071343813273665462888 b-39*%i,
        3.05353024725320778437191327051872412402 1b-39*%i
         +3.14159265358979323846264338327950288421 5b0,
        2.43364064818676710634152486221793050485 9b-39*%i
         +3.14159265358979323846264338327950288420 3b0]
```

```
(%i33) for s in sbfar40 do disp (expand  (subst (s,x,e40)))$
0.0b0

4.31808427754722231269317593140019978558b-78*%i
 -2.204051907791789077441381007291710645 91b-39

-2.15904213877361115634658796570009989 2779b-78*%i

(%i34) for s in sbfar40 do disp (pi50 - realpart(s))$
2.37853935049197237940549033703563773 8711b-38

-1.77242507584923038310911056003041731 1086b-38

-5.69380076179545511672356760217025250 1934b-39
```

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

2.8 Finding Highly Accurate Roots using bf_find_root

The syntax of **`bf_find_root`** is the same as **`find_root`**. If **`expr`** is an expression depending on the variable **`x`**, then

```
    bf_find_root (expr, x, a, b)
```

or, if **`f`** is a defined function, for example **`sin`**,

```
    bf_find_root (f, a, b)
```

There are two optional arguments, for example **`abserr = 1.0b-20`** or **`relerr = 1.0b-30`**, for example, but normally one can omit these optional arguments with good results, as we will see.

The given expression (or function, in the second syntax version) should have opposite signs at the two ends of the interval; the root is sought in the given interval **`[a,b]`**. Thus, this function will not find a root which involves an expression which only touches the **`y=0`** axis but does not cross it.

A simple example using an expression is a fourth order Legendre polynomial. We first use **`solve`** to get approximate roots using the default 16 digit floating point arithmetic.

```
(%i1) p4 : legendre_p(4,x);
STYLE-WARNING: redefining MAXIMA::SIMP-UNIT-STEP in DEFUN
STYLE-WARNING: redefining MAXIMA::SIMP-POCHHAMMER in DEFUN
(%o1) (-10*(1-x))+(35*(1-x)^4)/8-(35*(1-x)^3)/2+(45*(1-x)^2)/2+1
(%i2) solve(p4,x);
(%o2) [x = -sqrt(2*sqrt(30)+15)/sqrt(35),x = sqrt(2*sqrt(30)+15)/sqrt(35),
       x = -sqrt(15-2*sqrt(30))/sqrt(35),x = sqrt(15-2*sqrt(30))/sqrt(35)]
(%i3) float(%);
(%o3) [x = -0.8611363115940526,x = 0.8611363115940526,x = -0.3399810435848562,
       x = 0.3399810435848562]
(%i4) fpprintprec:8$
(%i5) [x1,x2,x3,x4] : map ('rhs,%o3);
(%o5) [-0.86113631,0.86113631,-0.33998104,0.33998104]
(%i6) for z in % do print (z, ev(p4,x = z))$
-0.86113631 3.5527137e-15
0.86113631 -2.220446e-16
-0.33998104 3.5527137e-15
0.33998104 8.8817842e-16
```

Now we focus on getting the value of the root near **0.34** using 40 digit arithmetic with **bf_find_root**.

```
(%i7) fpprec : 40$
(%i8) rt : bf_find_root(p4,x,0.3,0.4);
(%o8) 3.3998104b-1
(%i9) p4,x = rt;
(%o9) 3.6734198b-40
(%i10) fpprintprec : 0$
(%i11) rt;
(%o11) 3.399810435848562648026657591032446872006b-1
```

A simple example using a defined function is $f(x) = 1 - 2\,x/3 - \sin(x)$, which has a root which can be graphically located using

```
(%i10) plot2d([1-2*x/3 - sin(x),[discrete,[[0,0],[1,0]]],[x,0,1],
                  [style,[lines,2]],[legend,false])$
```

which produces the plot (after toggling on the grid):

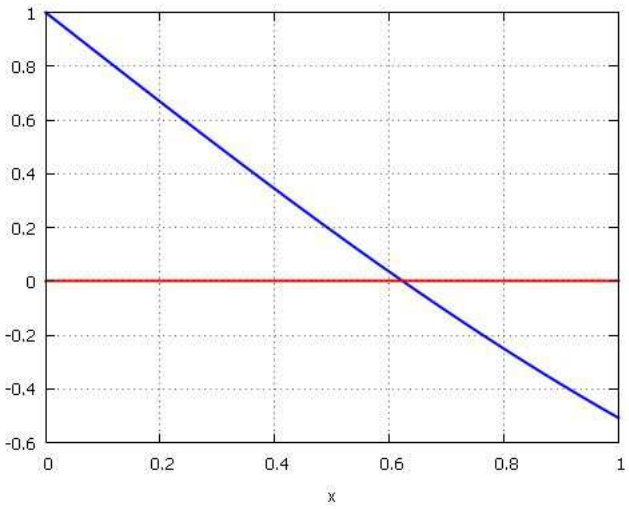

Figure 1: $f(x) = 1 - 2\,x/3 - \sin(x)$

Placing the cursor at the position of the curve crossing the x-axis, we find **x-root = 0.62** roughly. We can then find a more accurate value:

```
(%i1) g(x) := 1 - 2*x/3 - sin(x)$
(%i2) g(0.62);
(%o2) 0.005631506129361585
(%i3) fpprec : 40$
(%i4) rt : bf_find_root(g,0.6,0.64);
(%o4) 6.23806518961612319987615226164879204 95b-1
(%i5) g(rt);
(%o5) -2.29588740394978028900143854926219858949b-41
```

2.9 Bigfloat Number Gaps and Binary Arithmetic

The value of the integer **fpprec** determines the number of DECIMAL digits to be retained in the subsequent arithmetic. 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 or a bigfloat, there will be a range of floating point numbers (or bigfloats) which are not recognised as different.

The internal Lisp representation of a bigfloat is

```
( (BIGFLOAT SIMP  precision)  mantissa  exponent )
```

in which **precision** is an integer which is related to the number of bits of precision in the mantissa (the number of bits in the "fractional part" **[see below]** of a bigfloat is **precision - 1**), and this number can be retrieved from the console prompt using **?fpprec**.

Here is an example of setting **fpprec** to the integer 2:

```
(%i1) fpprec:2$
(%i2) ?fpprec;
(%o2) 9
(%i3) :lisp $fpprec
2
(%i3) :lisp fpprec
9
```

so if we ask for 2 **decimal** digits of precision (two digit arithmetic, with the answer rounded to 2 digits), Lisp uses 8 bits of precision in the fractional part of the bigfloat.

The "exponent" is a signed integer representing the scale of the number.

The "mantissa" is a signed integer used to represent a fractional portion defined by

```
fraction = mantissa / 2^(precision)
```

such that the actual decimal number is the product

```
fraction * 2^(exponent)
```

We start with a bigfloat which is represented by both a zero exponent and a zero mantissa (and hence a zero fraction), retaining **fpprec=2** for now.

```
(%i3) x : 0.0b0$
(%i4) :lisp $x;
((BIGFLOAT SIMP 9) 0 0)
```

Here is another example

```
(%i4) x : 1.0b0;
(%o4) 1.0b0
(%i5) ?print(x);
((BIGFLOAT SIMP 9) 256 1)
(%o5) 1.0b0
(%i6) :lisp $x
((BIGFLOAT SIMP 9) 256 1)
```

We can also define a value of **x** inside the Lisp interpreter. (We are using **msetq** instead of **setq** or **setf** to avoid useless noise from the SBCL Lisp interpreter.)

```
(%i6) :lisp (msetq $x '((BIGFLOAT SIMP 9) 0 0))
((BIGFLOAT SIMP 9) 0 0)
(%i6) x;
(%o6) 0.0b0
```

If we increase the mantissa integer by 1, we get **2.0b-3**.

```
(%i7) :lisp (msetq $u '((BIGFLOAT SIMP 9) 1 0))
((BIGFLOAT SIMP 9) 1 0)
(%i7) u;
(%o7) 2.0b-3
```

Consider the gap around the number **x:bfloat(2/3)** for the case **fpprec = 4**. 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 whose least significant decimal digit is one 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 roughly $\mathbf{ulp} = \mathbf{2} \cdot \mathbf{2}^{-\mathbf{18}} = \mathbf{2}^{-\mathbf{17}} \approx \mathbf{7.63b-6}$, 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.

Continuing with our **fpprec = 4** example:

```
(%i1) fpprec;
(%o1) 16
(%i2) fpprec:4$
(%i3) ?fpprec;
(%o3) 16
(%i4) x :bfloat(2/3);
(%o4) 6.667b-1
(%i5) u : bfloat(2^(-18));
(%o5) 3.815b-6
(%i6) x1 : x + u;
(%o6) 6.667b-1
(%i7) x1 - x;
(%o7) 1.526b-5
(%i8) x2 : x + 3.814b-6;
(%o8) 6.667b-1
(%i9) x2 - x;
(%o9) 0.0b0
(%i10) ulp : bfloat(2^(-17));
(%o10) 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**.

```
(%i11) fpprec:16$
(%i12) ?fpprec;
(%o12) 56
(%i13) x :bfloat(2/3);
(%o13) 6.666666666666667b-1
(%i14) u : bfloat(2^(-58));
(%o14) 3.469446951953614b-18
(%i15) x1 : x + u;
(%o15) 6.666666666666667b-1
(%i16) x1 - x;
(%o16) 1.387778780781446b-17
(%i17) x2 : x + 3.469446951953613b-18;
(%o17) 6.666666666666667b-1
(%i18) x2 - x;
(%o18) 0.0b0
(%i19) ulp : bfloat(2^(-57));
(%o19) 6.938893903907228b-18
```

2.10 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 defined in the Ch. 9 file **bfloat.mac**,

```
/*   fdf(f,x,dfp) finds the absolute value
       of the difference of f(x) at the current
     value of fpprec and at the value (fpprec+dfp),
     and returns [f(x), df(x)]
*/

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) load(bfloat);
(%o1) "c:/work3/bfloat.mac"
(%i2) fpprintprec:8$
(%i3) g(x) := sin(x/2)$
(%i4) fpprec;
(%o4) 16
(%i5) fdf(g,1,10);
(%o5) [4.7942553b-1,1.834924b-18]
```

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

```
(%i6) fdf(g,1,10),fpprec:30;
(%o6) [4.7942553b-1,2.6824592b-33]
(%i7) fpprec;
(%o7) 16
```

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

3 High Accuracy Quadrature with Maxima

3.1 Using bromberg for High Accuracy Quadrature

A bigfloat version of the Romberg quadrature method is defined in Lisp code in the file **`.../share/numeric/brmbrg.lisp`**. You need to use **`load(brmbrg)`** or **`load("brmbrg.lisp")`** to be able to use the function **bromberg**.

The use of **bromberg** is identical to the use of **romberg** (see the Maxima Manual entry for **`romberg`**) except that **rombergtol** (used for a possible return based on relative error) is replaced by the bigfloat **brombergtol** with a default value of **1.0b-4**, and **rombergabs** (used for a possible return based on the absolute error) 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 given absolute error is approximately equal to a the same relative error. We will test **bromberg** using the function **exp(x)** over the domain **[-1, 1]**, and use only the absolute error 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 $\int_{-1}^{1} e^x \, dx$, 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 absolute error (compared with **tval**) of about 7×10^{-24}.

```
(%i1) load(bfloat);
(%o1) "c:/work3/bfloat.mac"
(%i2) fpprintprec:8$
(%i3) load(brmbrg);
(%o3) "C:/Program Files (x86)/Maxima-sbcl-5.36.1/share/maxima/5.36.1/share/numeric/brmbrg.lisp"
(%i4) [brombergtol,brombergabs,brombergit,brombergmin,fpprec,fpprintprec];
(%o4) [1.0b-4,0.0b0,11,0,16,8]
(%i5) integrate(exp(x),x,-1,1);
(%o5) %e-%e^-1
(%i6) tval : bfloat(%),fpprec:42;
(%o6) 2.3504023b0
(%i7) fpprec;
(%o7) 16
```

```
(%i8) (brombergtol:0.0b0,brombergit:100)$
(%i9) b15:(brombergabs:1.0b-15, bromberg (exp(x),x,-1,1) ), fpprec:30;
(%o9) 2.3504023b0
(%i10) abs (b15 - tval),fpprec:42;
(%o10) 6.9167325b-24
(%i11) b20:(brombergabs:1.0b-20, bromberg (exp(x),x,-1,1) ), fpprec:30;
(%o11) 2.3504023b0
(%i12) abs (b20 - tval),fpprec:42;
(%o12) 1.5154761b-29
```

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

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

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

This function, with the syntax

```
    qbromberg ( f, a, b, raccur, 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 $10^{-\mathrm{raccur}}$, where **raccur** is the "requested absolute error".

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

```
(%i1) load(bfloat);
(%o1) "c:/work3/bfloat.mac"
(%i2) fpprintprec:8$
(%i3) load(brmbrg);
(%o3) "C:/Program Files (x86)/Maxima-sbcl-5.36.1/share/maxima/5.36.1/share/numeric/brmbrg.lisp"
(%i4) [brombergtol,brombergabs,brombergit,brombergmin,fpprec,fpprintprec];
(%o4) [1.0b-4,0.0b0,11,0,16,8]
(%i5) qbr20 : qbromberg(exp,-1,1,20,40,100);
(%o5) 2.3504023b0
(%i6) abs(qbr20 - tval);
(%o6) 0.0b0
(%i7) abs(qbr20 - tval),fpprec:40;
(%o7) 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 absolute 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 absolute accuracy 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 absolute accuracy, computes the actual absolute error in the result, and prints a line containing (raccur, fpprec, value, value-error). Here is the code for such a function, defined in **bfloat.mac**:

```
/*   qbrlist(f,a,b,rplist,fp,itmax) assumes tval is globally defined, sets fpprec to fp,
       brombergit to itmax, computes bromberg integral of function f over [a,b] with
       each raccur in rplist and computes absolute error of result.
*/
qbrlist(%f,a,b,rplist,fp,itmax) :=
  block([fpprec:fp,fpprintprec,brombergtol,brombergabs,brombergit,val,verr,raccur],
     if not listp(rplist) then (print("rplist # list"),return(done)),

     brombergtol : 0.0b0,
     brombergit : itmax,
     fpprintprec:8,
     print(" raccur   fpprec     val             verr "),
     print(" "),
     for raccur in rplist do
      ( brombergabs : bfloat(10^(-raccur)),
        val: bromberg(%f(x),x,a,b),
        verr: abs(val - tval),
        print("  ",raccur,"   ",fp,"   ",val,"   ",verr) ) )$
```

and here is an example of use of **qbrlist** in which the requested absolute error **raccur** is set to three different values supplied by the list **rplist** for each setting of **fpprec** used. We use the "true" value **tval** computed above. Here is our test for three different values of **fpprec** (the next to last arg):

```
(%i8) qbrlist(exp,-1,1,[10,15,17 ],20,100)$
 raccur   fpprec       val               verr

   10       20       2.3504023b0       4.5259436b-14
   15       20       2.3504023b0       1.3552527b-20
   17       20       2.3504023b0       1.3552527b-20
(%i9) qbrlist(exp,-1,1,[10,20,27 ],30,100)$
 raccur   fpprec       val               verr

   10       30       2.3504023b0       4.5259437b-14
   20       30       2.3504023b0       1.4988357b-29
   27       30       2.3504023b0       5.5220263b-30
(%i10) qbrlist(exp,-1,1,[10,20,30,35],40,100)$
 raccur   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 **raccur** from **30** to **35** results in no improvement in the actual absolute error of the result.

When bromberg Fails

One should not try to use **bromberg** for an integrand which has end point algebraic and/or logarithmic singularities. 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. To illustrate the problem, we use our homemade function **bromberg_abs(f,a,b,abserr)**, defined in **bfloat.mac**.

```
(%i1) load(bfloat);
(%o1) "c:/work3/bfloat.mac"
(%i2) g(x):= sqrt(x)*log(x)$
(%i3) i1 : integrate(g(t),t,0,1);
(%o3) -4/9
(%i4) fpprec : 20$
(%i5) tval : bfloat(i1);
(%o5) -4.4444444444444444445b-1
(%i6) i18 : bromberg_abs(g,0,1,1.0b-18);
log: encountered log(0).
#0: g(x=0.0b0)
#1: bromberg1(%f=g,a=0,b=1,n=2)
#2: bromberg_abs(%g=g,%x1=0,%x2=1,%abserr=1.0b-18)(bfloat.mac line 286)
 -- an error. To debug this try: debugmode(true);
```

The Maxima code for **bromberg_abs**, easier to follow than the Lisp code in **brmbrg.lisp**, is based on the Romberg Method pseudo-code in Ch. 4 of the Seventh Ed. (2001) of **Numerical Analysis** by Burden and Faires.

```
bromberg_abs(%g, %x1, %x2, %abserr) :=
block([count,maxit:20,rval1,rval2 ],
     %abserr : bfloat(%abserr),
     rval1 : bromberg1(%g,%x1,%x2,2),
     count :1,
     do (
          rval2 : bromberg1(%g,%x1,%x2,count+3),
          if abs (rval2 - rval1) < %abserr then return(),
          count : count + 1,
          if count > maxit then (
              print(" reached max number of iterations, maxit = ",maxit),
              return()),
          rval1 : rval2 ),
     rval2)$
```

and the first call is to **bromberg1(f,a,b,n)** with **n=2**. The code for **bromberg1** is

```
bromberg1 (%f, a, b, n) :=
block ([h, fxv,  p,xL,fxL  ],
     local(R),
     a : bfloat(a),
     b : bfloat(b),
     h : (b - a),
     R[1,1] : h*(%f(a) + %f(b))/2.0b0,
     for i:2 thru n do (
          p : 2^(i-2),
          xL : makelist(a + (k - 0.5b0)*h, k ,1,p),
          fxL : map (%f, xL),
          /* approximation from trapezoidal method */
          R[2,1] : ( R[1,1] + h*apply ("+", fxL)   )/ 2.0b0,
          /* extrapolation */
          for j:2 thru i do
              R[2,j] : R[2,j-1] + (R[2,j-1] - R[1,j-1])/ bfloat(4^(j-1) - 1),
          h : h/2.0b0,
          /* update  R[1,k]  */
          for j:1 thru i do R[1,j] : R[2,j]),
     R[2,n])$
```

We then see that the first step taken by **bromberg1(f,a,b,n)** is to evaluate the given integrand at both of the endpoints of the given interval. Thus, this is more accurately called the Romberg End Point Method.

You can instead use the tanh-sinh quadrature method for this integral (see Sec. 3.3).

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

This method (H. Takahasi and M. Mori, 1974; see Sec 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 $x \to u$:

$$x(u) = exp(u - exp(-u)) \tag{3.1}$$

and hence

$$F(u) = f(x(u))\,w(u), \qquad \text{where} \qquad w(u) = \frac{dx}{du} = exp(-exp(-u)) + x(u). \tag{3.2}$$

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

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 $(2\,N + 1)$ of function evaluations.

$$I(h, N) \simeq h \sum_{j=-N}^{N} F(u_j) \quad \text{where} \quad u_j = j\,h \tag{3.3}$$

This method is implemented with, for example, the function **quad_de(f, a, ra, fp)** defined in the Ch. 9 file **quad_de.mac**. We demonstrate the available functions on the simple integral $\int_0^\infty e^{-x}\,dx = 1$

```
(%i1) fpprec;
(%o1) 16
(%i2) fpprintprec:8$
(%i3) g(x):= exp(-x)$
(%i4) ival : integrate(g(x),x,0,inf);
(%o4) 1
(%i5) tval : bfloat(ival),fpprec:45;
(%o5) 1.0b0
(%i6) load(quad_de);
(%o6) "c:/work3/quad_de.mac"
(%i7) quad_de(g,0,30,40);
(%o7) [1.0b0,4,4.8194669b-33]
(%i8) abs(first(%) - tval),fpprec:45;
(%o8) 9.1835496b-41
```

The function **quad_de(f, a, ra, fp)** integrates the Maxima function **f** over the domain $[x \ge a]$, 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^{-ra}. The parameter **ra** is the "requested accuracy", and the value of **h** is repeatedly halved until the **vdiff** magnitude either satisfies this criterion or starts increasing. The first element of the returned list is the approximate value of the integral. The second element (4 above) is the "final k-level" used, where **k** is related to the step size **h** by $h = 2^{-k}$, so a larger value of **k** corresponds to a smaller value of **h**. The third and last element of the returned list is the final value of **vdiff**. We see in the above example that requesting accuracy $ra = 30$ 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**, and returns an approximate value of the integral.

```
(%i9) idek(g,0,4,40);
(%o9) 1.0b0
(%i10) abs(% - tval),fpprec:45;
(%o10) 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 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 arithmetic being used.)

```
(%i11) idek_e(g,0,4,40);
(%o11) [1.0b0,8.3668155b-42]
(%i12) abs(first(%) - tval),fpprec:45;
(%o12) 9.1835496b-41
```

The package function **ide(f, a, ra, fp)** follows the same path as **quad_de(f, a, ra, fp)**, but shows the progression toward success as the k level increases (and correspondingly **h** decreases), The value **vdiff** is the absolute difference between the returned approximation and the previous approximation defined by the algorithm.

```
(%i13) ide(g,0,30,40)$

      requested accuracy =  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, ra, fp)** follows the path of **ide(f, a, ra, fp)**, but adds to the table the value of the absolute value of the error of the returned result (when compared to the global value of **tval**) for each **k** level attempted. The usefulness of this function depends on having an accurate value of the integral bound to the global variable **tval**.

```
(%i14) ide_test(g,0,30,40)$

      requested accuracy =  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

$$\int_0^\infty \frac{e^{-t}}{\sqrt{t}}\,dt = \sqrt{\pi} \tag{3.4}$$

```
(%i15) g(x):= exp(-x)/sqrt(x)$
(%i16) integrate(g(t),t,0,inf);
(%o16) sqrt(%pi)
(%i17) tval : bfloat(%),fpprec:45;
(%o17) 1.7724538b0
(%i18) quad_de(g,0,30,40);
(%o18) [1.7724538b0,4,1.0443243b-34]
(%i19) abs(first(%) - tval),fpprec:45;
(%o19) 1.8860005b-40
(%i20) idek_e(g,0,4,40);
(%o20) [1.7724538b0,2.7054206b-41]
```

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

Test Integral 2

Our second known integral is

$$\int_0^\infty e^{-t^2/2}\,dt = \sqrt{\pi/2} \tag{3.5}$$

```
(%i21) g(x) := exp(-x^2/2)$
(%i22) tval : bfloat(sqrt(%pi/2)),fpprec:45$
(%i23) quad_de(g,0,30,40);
(%o23) [1.2533141b0,5,1.099771b-31]
(%i24) abs(first(%) - tval),fpprec:45;
(%o24) 2.1838045b-40
(%i25) idek_e(g,0,5,40);
(%o25) [1.2533141b0,1.3009564b-41]
```

Test Integral 3

Our third test integral is

$$\int_0^\infty e^{-t}\cos t\,dt = 1/2 \tag{3.6}$$

```
(%i26) g(x) := exp(-x)*cos(x)$
(%i27) integrate(g(x),x,0,inf);
(%o27) 1/2
(%i28) tval : bfloat(%),fpprec:45$
(%i29) quad_de(g,0,30,40);
(%o29) [5.0b-1,5,1.7998243b-33]
(%i30) abs(first(%) - tval),fpprec:45;
(%o30) 9.1835496b-41
(%i31) idek_e(g,0,5,40);
(%o31) [5.0b-1,9.8517724b-42]
```

3.3 The tanh-sinh Quadrature Method for $\mathbf{a \le x \le b}$

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

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

$$\int_{-1}^{1} \mathbf{f(x)}\,\mathbf{dx} = \int_{-\infty}^{\infty} \mathbf{F(u)}\,\mathbf{du}. \tag{3.7}$$

The change of variables is expressed by

$$\mathbf{x(u) = \tanh\left(\frac{\pi}{2}\sinh u\right)} \tag{3.8}$$

and you can confirm that

$$\mathbf{u = 0 \Rightarrow x = 0}, \qquad \mathbf{u \to -\infty \Rightarrow x \to -1, u \to \infty \Rightarrow x \to 1} \tag{3.9}$$

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

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

$$\mathbf{w(u) = \frac{\frac{\pi}{2}\cosh u}{\cosh^2\left(\frac{\pi}{2}\sinh u\right)}} \tag{3.10}$$

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

$$\mathbf{F(u) \approx \exp\left(-\frac{\pi}{2}\exp(\,|\,u\,|\,)\right)} \qquad \text{for} \quad \mathbf{u \to \pm\infty}. \tag{3.11}$$

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 $\mathbf{(2N + 1)}$ of function evaluations.

$$\mathbf{I(h, N) \simeq h \sum_{j=-N}^{N} F(u_j)} \quad \text{where} \quad \mathbf{u_j = j\,h} \tag{3.12}$$

This method is implemented with, for example, the function **quad_ts (f, a, b, ra, fp)** defined in the Ch. 9 file **quad_ts.mac**, and we will illustrate the available package functions using the simple integral $\int_{-1}^{1} e^x \, dx$.

The package function **quad_ts (f, a, b, ra, 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 $(I_k - I_{k-1})$ is less than 10^{-ra} (ra is the "requested accuracy" 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 10^{-ra} .

```
(%i1) fpprec;
(%o1) 16
(%i2) fpprintprec:8$
(%i3) tval : bfloat( integrate( exp(x),x,-1,1 )),fpprec:45;
(%o3) 2.3504023b0
(%i4) load(quad_ts);
   _kmax% =  8   _epsfac% =  2
(%o4) "c:/work3/quad_ts.mac"
(%i5) bfprint(tval,45)$
   number of digits =  45
   2.35040238728760291376476370119120163031143596b0
(%i6) quad_ts(exp,-1,1,30,40);
 construct _yw%[kk,fpprec] array for kk =
  8  and fpprec =  40   ...working...
(%o6) [2.3504023b0,5,0.0b0]
(%i7) abs(first(%) - tval),fpprec:45;
(%o7) 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 $du = h = 1/2^8 = 1/256$. The actual "k-level" needed to return a result with the requested accuracy **ra** 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 $k = 5$ and $h = 1/2^5 = 1/32$ returned an answer with an actual accuracy of about 40 correct digits (when $ra = 30$ and $fp = 40$).

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 $F(u) = f(x(u))\, w(u)$ 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 $eps = 10^{-n\,p}$, 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 **ra**) integrates the Maxima function **f** over the domain $[a, b]$ using a "k-level approximation" with $h = 1/2^k$ and **fpprec : fp**.

```
(%i8) qtsk(exp,-1,1,5,40);
(%o8) 2.3504023b0
(%i9) abs(% - tval),fpprec:45;
(%o9) 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.

```
(%i10) qtsk_e(exp,-1,1,5,40);
(%o10) [2.3504023b0,2.0614559b-94]
(%i11) abs(first(%) - tval),fpprec:45;
(%o11) 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 $x : \mathrm{bfloat}(1 - y)$ and the subsequent evaluation $f(x)$. 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]**.

```
(%i12) last(_yw%[8,40]);
(%o12) [4.7024891b-83,8.9481574b-81]
```

(In Eq. (3.12) we have separated out the **(u = 0, x = 0)** term, and used the symmetry properties $x(-u) = -x(u)$, and $w(-u) = w(u)$ 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

```
(%i13) 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, ra, fp)** follows the same path as **quad_ts(f, a, b, ra, fp)**, but shows the progression toward success as the **k** level increases (and **h** decreases):

```
(%i14) qts(exp,-1,1,30,40)$
  ra =  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, ra, fp)** follows the path of **qts(f, a, b, ra, fp)**, but adds to the table the value of the absolute 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**.

```
(%i15) qts_test(exp,-1,1,30,40)$
      ra =  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_abs** in Sec. 3.1 :

$$\int_0^1 \sqrt{t}\,\ln(t)\,dt = -4/9 \tag{3.13}$$

```
(%i16) g(x):= sqrt(x)*log(x)$
(%i17) tval : bfloat(integrate(g(t),t,0,1)),fpprec:45;
(%o17) -4.4444444b-1
(%i18) quad_ts(g,0,1,30,40);
(%o18) [-4.4444444b-1,5,3.4438311b-41]
(%i19) abs(first(%) - tval),fpprec:45;
(%o19) 4.4642216b-41
(%i20) qtsk_e(g,0,1,5,40);
(%o20) [-4.4444444b-1,7.9678502b-44]
```

Requesting thirty digit accuracy with forty digit arithmetic returns a value for this integral which has about forty correct digits. Note that “vdiff” (the third and last element of the list returned by **quad_ts(g,0,1,30,40)**) is approximately the same as the actual absolute error.

Test Integral 2

Consider the integral

$$\int_0^1 \frac{\arctan(\sqrt{2+t^2})}{(1+t^2)\sqrt{2+t^2}}\,dt = 5\,\pi^2/96. \tag{3.14}$$

```
(%i21) g(x):= atan(sqrt(2+x^2))/(sqrt(2+x^2)*(1+x^2))$
(%i22) integrate(g(t),t,0,1);
(%o22) 'integrate(atan(sqrt(t^2+2))/((t^2+1)*sqrt(t^2+2)),t,0,1)
(%i23) quad_qags(g(t),t,0,1);
(%o23) [0.5140419,5.7070115e-15,21,0]
(%i24) float(5*%pi^2/96);
(%o24) 0.5140419
(%i25) tval: bfloat(5*%pi^2/96),fpprec:45;
(%o25) 5.1404189b-1
(%i26) quad_ts(g,0,1,30,40);
(%o26) [5.1404189b-1,5,1.5634993b-36]
(%i27) abs(first(%) - tval),fpprec:45;
(%o27) 7.3300521b-41
(%i28) qtsk_e(g,0,1,5,40);
(%o28) [5.1404189b-1,1.3887835b-43]
```

Test Integral 3

We consider the integral

$$\int_0^1 \frac{\sqrt{t}}{\sqrt{1-t^2}}\,dt = 2\,\sqrt{\pi}\,\Gamma(3/4)/\Gamma(1/4) \tag{3.15}$$

```
(%i29) g(x):= sqrt(x)/sqrt(1 - x^2)$
(%i30) quad_qags(g(t),t,0,1);
(%o30) [1.1981402,8.6787244e-11,567,0]
(%i31) integrate(g(t),t,0,1);
(%o31) beta(1/2,3/4)/2
(%i32) tval : bfloat(%),fpprec:45;
(%o32) 1.1981402b0
(%i33) quad_ts(g,0,1,30,40);
(%o33) [1.1981402b0,5,1.3775324b-40]
(%i34) abs(first(%) - tval),fpprec:45;
(%o34) 1.8625393b-40
(%i35) qtsk_e(g,0,1,5,40);
(%o35) [1.1981402b0,1.5833892b-45]
```

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

```
(%i36) integrate(g(t),t,0,1);
(%o36) beta(1/2,3/4)/2
(%i37) makegamma(%);
(%o37) (2*sqrt(%pi)*gamma(3/4))/gamma(1/4)
(%i38) bfloat(%),fpprec:45;
(%o38) 1.1981402b0
```

Test Integral 4

We next consider the integral

$$\int_0^1 \ln^2 t\,dt = 2 \tag{3.16}$$

```
(%i39) g(x) := log(x)^2$
(%i40) integrate(g(t),t,0,1);
(%o40) 2
(%i41) tval : bfloat(%), fpprec:45;
(%o41) 2.0b0
(%i42) quad_ts(g,0,1,30,40);
(%o42) [2.0b0,5,0.0b0]
(%i43) abs( first(%) - tval ),fpprec:45;
(%o43) 1.8367099b-40
(%i44) qtsk_e(g,0,1,5,40);
(%o44) [2.0b0,4.3344016b-43]
```

Test Integral 5

We finally consider the known integral

$$\int_0^{\pi/2} \ln(\cos t)\,dt = -\pi\ \ln(2)/2 \tag{3.17}$$

```
(%i45) g(x) := log( cos(x) )$
(%i46) quad_qags(g(t),t,0,%pi/2);
(%o46) [-1.088793,8.8817842e-15,231,0]
(%i47) integrate(g(t),t,0,%pi/2);
(%o47) (%i*%pi^2)/24-(6*%pi*log(4)+%i*%pi^2)/24
(%i48) tval : bfloat(%),fpprec:45;
(%o48) 4.1123351b-1*%i-4.1666666b-2*(9.8696044b0*%i+2.6131033b1)
```

```
(%i49) rectform(%);
(%o49) (-6.9388939b-18*%i)-1.088793b0
(%i50) float(-%pi*log(2)/2);
(%o50) -1.088793
(%i51) tval : bfloat(-%pi*log(2)/2),fpprec:45;
(%o51) -1.088793b0
(%i52) quad_ts(g,0,%pi/2,30,40);
(%o52) [1.2979374b-80*%i-1.088793b0,5,9.1835496b-41]
(%i53) ans: realpart( first(%) ),fpprec:45;
(%o53) -1.088793b0
(%i54) abs(ans - tval),fpprec:45;
(%o54) 1.9661688b-40
(%i55) qtsk_e(g,0,%pi/2,5,40);
(%o55) [1.2979374b-80*%i-1.088793b0,1.2858613b-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 **`ra = 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 **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).

3.4 The Gauss-Legendre Quadrature Method for $a \le x \le b$

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 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

$$\int_{-1}^{1} f(x)\,dx \approx \sum_{j=1}^{N} w_j\, f(x_j) \tag{3.18}$$

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

$$w_j = \frac{-2}{(N+1)\,P'_N(x_j)\,P_{N+1}(x_j)} \tag{3.19}$$

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 can be downloaded to use this package. This needed lisp file is now also available with the other Ch. 9 software files on the Maxima by Example webpage.

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 high accuracy code and some background to the problem of high accuracy quadrature has been provided by Richard Fateman in his draft paper **Numerical Quadrature in a Symbolic/Numerical Setting**, **quad.pdf** (see Sec. 2.10).

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. We use **gaussunit(exp, 4)** with the default value **fpprec = 16**, which integrates the exponential function over **[-1,1]** using 4 point Gauss-Legendre quadrature. This generates the hashed array **ab_and_wts[4,16]** with elements appropriate to the function being integrated.

```
(%i1) load("quad-maxima.lisp");
STYLE-WARNING: using deprecated EVAL-WHEN situation names EVAL COMPILE LOAD
(%o1) "c:/work3/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);
(%o13) (35*x^4)/8-(15*x^2)/4+3/8
(%i14) float(solve(lp4));
(%o14) [x = -0.86113631,x = 0.86113631,x = -0.33998104,x = 0.33998104]
(%i15) ab_and_wts[4,16];
(%o15) [[8.6113631b-1,3.3998104b-1],[3.4785484b-1,6.5214515b-1]]
```

With the default value of **`fpprec = 16`** and using only four integrand evaluation points (with approximate values given by **`%o14`**), the error is about 2 parts in 10^7. The **first element `%o11`** 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 16 digit arithmetic, **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, as seen in the output **`%o11`**.

We can verify that the list of (positive) roots **`%o11`** returned is correct to within the global floating point precision (we do this two different ways):

```
(%i16) lfp4(x) := legenp(4,x)$
(%i17) map('lfp4,%o11);
(%o17) [0.0b0,-3.4694469b-18]
(%i18) map( lambda([z], legenp (4, z)),%o11 );
(%o18) [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 the returned list:

```
(%i19) gaussunit_e(exp,4);
(%o19) [2.350402b0,1.2761299b-17]
(%i20) arrayinfo(ab_and_wts);
(%o20) [hashed,2,[4,16]]
```

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)`**.

```
(%i21) gaussab(exp,-1,1,4);
(%o21) 2.350402b0
(%i22) abs(% - tval),fpprec:45;
(%o22) 2.9513122b-7
(%i23) gaussab_e(exp,-1,1,4);
(%o23) [2.350402b0,1.2761299b-17]
(%i24) arrayinfo(ab_and_wts);
(%o24) [hashed,2,[4,16]]
```

The package function **quad_gs (f, a, b, ra)** 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^{-ra} (**ra** is the "requested accuracy" 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 accuracy quadrature, rather than the method used in the previous two sections in which **fpprec** was set "locally" inside a **block**.

This function **quad_gs (f, a, b, ra)** returns the list
[approx-value, number-function-evaluations, abs(vdiff)], where the last element should be smaller than 10^{-ra}.

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

```
(%i25) quad_gs(exp,-1,1,10);
fpprec = 16

(%o25) [2.3504023b0,20,6.6613381b-16]
(%i26) abs(first(%) -tval),fpprec:45;
(%o26) 6.2016267b-16
(%i27) arrayinfo(ab_and_wts);
(%o27) [hashed,2,[4,16],[10,16],[20,16]]
(%i28) fpprec:30$
(%i29) quad_gs(exp,-1,1,20);
fpprec = 30

(%o29) [2.3504023b0,20,1.2162089b-24]
(%i30) abs(first(%) -tval),fpprec:45;
(%o30) 2.2001783b-30
(%i31) arrayinfo(ab_and_wts);
(%o31) [hashed,2,[4,16],[10,16],[10,30],[20,16],[20,30]]
(%i32) fpprec:40$
(%i33) quad_gs(exp,-1,1,30);
fpprec = 40

(%o33) [2.3504023b0,40,1.8367099b-40]
(%i34) abs(first(%) -tval),fpprec:45;
(%o34) 2.7905177b-40
(%i35) arrayinfo(ab_and_wts);
(%o35) [hashed,2,[4,16],[10,16],[10,30],[10,40],[20,16],
                    [20,30],[20,40],[40,40]]
(%i36) gaussab_e(exp,-1,1,40);
(%o36) [2.3504023b0,1.8492214b-41]
(%i37) arrayinfo(ab_and_wts);
(%o37) [hashed,2,[4,16],[10,16],[10,30],[10,40],[20,16],
                   [20,30],[20,40],[40,40]]
```

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, ra)** and is also the last argument of the function **gaussab_e(f, a, b, N)**.

We see that requesting **30** digit accuracy for the answer while using the global **fpprec** set to **40** results in an answer good to about **39** digits.

Using **quad_gs (f, a, b, ra)** 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, ra)** 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, ra)** 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, ra)** which prints out a table showing the progression toward success:

```
(%i38) 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
(%i39) arrayinfo(ab_and_wts);
(%o39) [hashed,2,[4,16],[10,16],[10,30],[10,40],[20,16],
                                        [20,30],[20,40],[40,40]]
```