Computer algebra independent integration tests

4-Trig-functions/4.6-Cosecant/4.6.4.2-a+b-csc-^m-d-csc-^n-A+B-csc+C-csc^2-

Nasser M. Abbasi

May 23, 2020 Compiled on May 23, 2020 at 7:09am

Contents

1 Introduction **3**
1.1 Listing of CAS systems tested . . . 3
1.2 Results . . . 3
1.3 Performance . . . 5
1.4 list of integrals that has no closed form antiderivative . . . 6
1.5 list of integrals solved by CAS but has no known antiderivative . . . 6
1.6 list of integrals solved by CAS but failed verification . . . 6
1.7 Timing . . . 7
1.8 Verification . . . 7
1.9 Important notes about some of the results . . . 7
1.10 Design of the test system . . . 8

2 detailed summary tables of results **11**
2.1 List of integrals sorted by grade for each CAS . . . 11
2.2 Detailed conclusion table per each integral for all CAS systems . . . 12
2.3 Detailed conclusion table specific for Rubi results . . . 12

3 Listing of integrals **15**
3.1 $\int \frac{(a+b\csc(x))\left(A+B\csc(x)+C\csc^2(x)\right)}{\sqrt{\csc(x)}}\,dx$ . . . 15

4 Listing of Grading functions **19**

Chapter 1

Introduction

This report gives the result of running the computer algebra independent integration problems. The listing of the problems are maintained by and can be downloaded from `https://rulebasedintegration.org`

The number of integrals in this report is [1]. This is test number [133].

1.1 Listing of CAS systems tested

The following systems were tested at this time.

1. Mathematica 12.1 (64 bit) on windows 10.
2. Rubi 4.16.1 in Mathematica 12 on windows 10.
3. Maple 2020 (64 bit) on windows 10.
4. Maxima 5.43 on Linux. (via sagemath 8.9)
5. Fricas 1.3.6 on Linux (via sagemath 9.0)
6. Sympy 1.5 under Python 3.7.3 using Anaconda distribution.
7. Giac/Xcas 1.5 on Linux. (via sagemath 8.9)

Maxima, Fricas and Giac/Xcas were called from inside SageMath. This was done using SageMath integrate command by changing the name of the algorithm to use the different CAS systems.

Sympy was called directly using Python.

1.2 Results

Important note: A number of problems in this test suite have no antiderivative in closed form. This means the antiderivative of these integrals can not be expressed in terms of elementary, special functions or `Hypergeometric2F1` functions. `RootSum` and `RootOf` are not allowed.

If a CAS returns the above integral unevaluated within the time limit, then the result is counted as passed and assigned an A grade.

However, if CAS times out, then it is assigned an F grade even if the integral is not integrable, as this implies CAS could not determine that the integral is not integrable in the time limit.

If a CAS returns an antiderivative to such an integral, it is assigned an A grade automatically and this special result is listed in the introduction section of each individual test report to make it easy to identify as this can be important result to investigate.

The results given in in the table below reflects the above.

System	solved	Failed
Rubi	% 100. (1)	% 0. (0)
Mathematica	% 100. (1)	% 0. (0)
Maple	% 100. (1)	% 0. (0)
Maxima	% 0. (0)	% 100. (1)
Fricas	% 0. (0)	% 100. (1)
Sympy	% 0. (0)	% 100. (1)
Giac	% 0. (0)	% 100. (1)

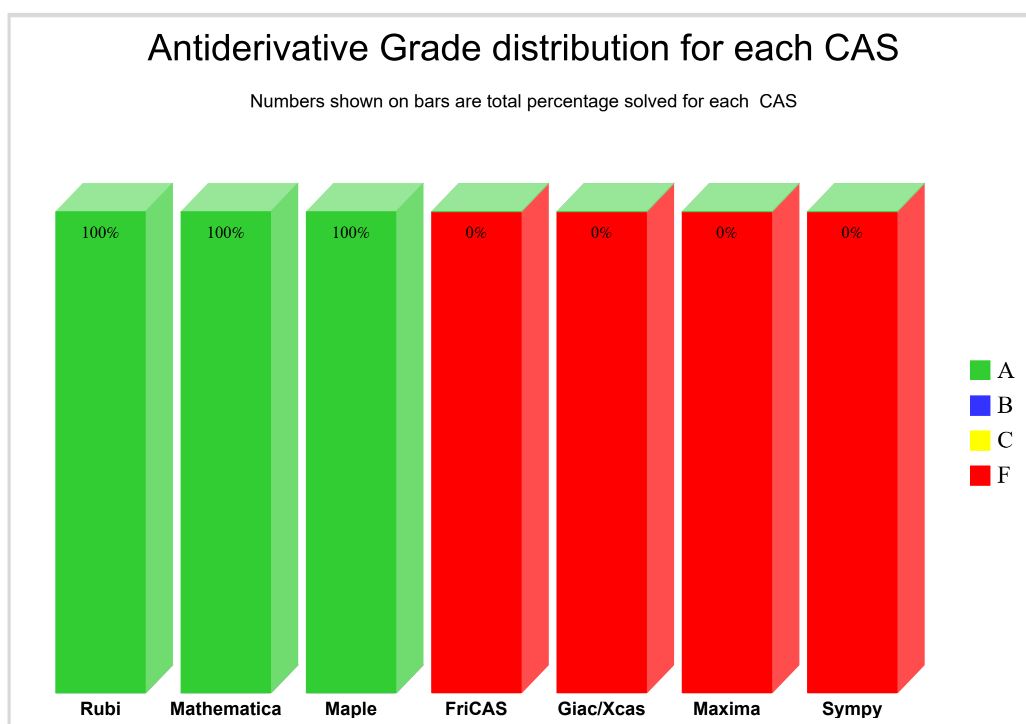
The table below gives additional break down of the grading of quality of the antiderivatives generated by each CAS. The grading is given using the letters A,B,C and F with A being the best quality. The grading is accomplished by comparing the antiderivative generated with the optimal antiderivatives included in the test suite. The following table describes the meaning of these grades.

grade	description
A	Integral was solved and antiderivative is optimal in quality and leaf size.
B	Integral was solved and antiderivative is optimal in quality but leaf size is larger than twice the optimal antiderivatives leaf size.
C	Integral was solved and antiderivative is non-optimal in quality. This can be due to one or more of the following reasons 1. antiderivative contains a hypergeometric function and the optimal antiderivative does not. 2. antiderivative contains a special function and the optimal antiderivative does not. 3. antiderivative contains the imaginary unit and the optimal antiderivative does not.
F	Integral was not solved. Either the integral was returned unevaluated within the time limit, or it timed out, or CAS hanged or crashed or an exception was raised.

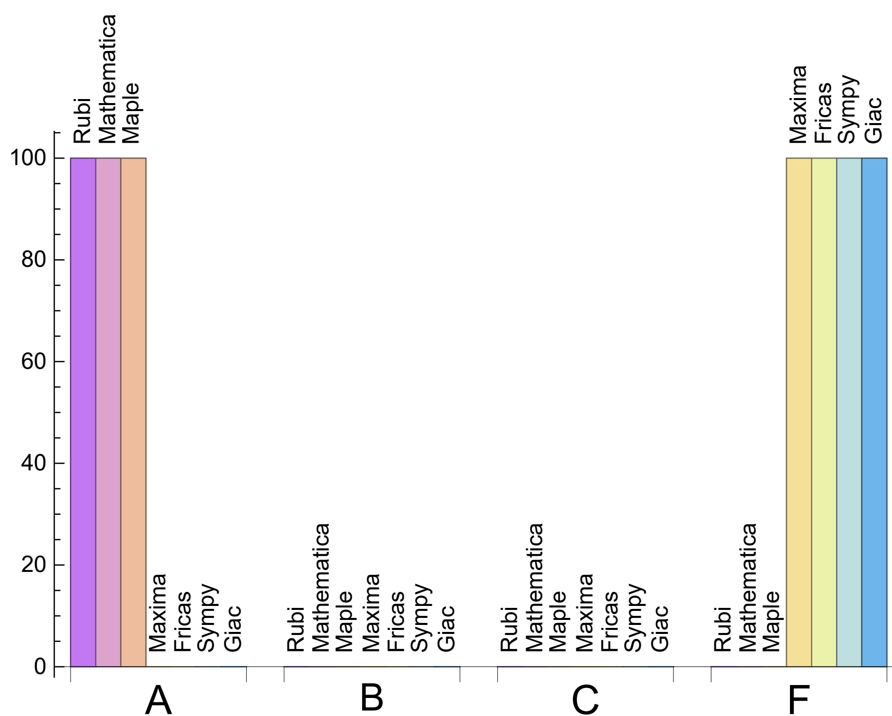
Grading is implemented for all CAS systems. Based on the above, the following table summarizes the grading for this test suite.

System	% A grade	% B grade	% C grade	% F grade
Rubi	100.	0.	0.	0.
Mathematica	100.	0.	0.	0.
Maple	100.	0.	0.	0.
Maxima	0.	0.	0.	100.
Fricas	0.	0.	0.	100.
Sympy	0.	0.	0.	100.
Giac	0.	0.	0.	100.

The following is a Bar chart illustration of the data in the above table.



The figure below compares the CAS systems for each grade level.



1.3 Performance

The table below summarizes the performance of each CAS system in terms of CPU time and leaf size of results.

System	Mean time (sec)	Mean size	Normalized mean	Median size
Rubi	0.18	112.	1.	112.
Mathematica	0.74	133.	1.19	133.
Maple	1.64	204.	1.82	204.
Maxima	Round[Mean[], 0.01]	Round[Mean[], 0.01]	Round[Mean[], 0.01]	Round[Median[], 0.0
Fricas	Round[Mean[], 0.01]	Round[Mean[], 0.01]	Round[Mean[], 0.01]	Round[Median[], 0.0
Sympy	Round[Mean[], 0.01]	Round[Mean[], 0.01]	Round[Mean[], 0.01]	Round[Median[], 0.0
Giac	Round[Mean[], 0.01]	Round[Mean[], 0.01]	Round[Mean[], 0.01]	Round[Median[], 0.0

1.4 list of integrals that has no closed form antiderivative

{}

1.5 list of integrals solved by CAS but has no known antiderivative

Rubi {}

Mathematica {}

Maple {}

Maxima {}

Fricas {}

Sympy {}

Giac {}

1.6 list of integrals solved by CAS but failed verification

The following are integrals solved by CAS but the verification phase failed to verify the anti-derivative produced is correct. This does not mean necessarily that the anti-derivative is wrong, as additional methods of verification might be needed, or more time is needed (3 minutes time limit was used). These integrals are listed here to make it easier to do further investigation to determine why it was not possible to verify the result produced.

Rubi {}

Mathematica {}

Maple Verification phase not implemented yet.

Maxima Verification phase not implemented yet.

Fricas Verification phase not implemented yet.

Sympy Verification phase not implemented yet.

Giac Verification phase not implemented yet.

1.7 Timing

The command `AboluteTiming[]` was used in Mathematica to obtain the elapsed time for each integrate call. In Maple, the command `Usage` was used as in the following example

```
cpu_time := Usage(assign ('result_of _int',int(expr,x)),output='realtime'
```

For all other CAS systems, the elapsed time to complete each integral was found by taking the difference between the time after the call has completed from the time before the call was made. This was done using Python's `time.time()` call.

All elapsed times shown are in seconds. A time limit of 3 minutes was used for each integral. If the integrate command did not complete within this time limit, the integral was aborted and considered to have failed and assigned an F grade. The time used by failed integrals due to time out is not counted in the final statistics.

1.8 Verification

A verification phase was applied on the result of integration for Rubi and Mathematica. Future version of this report will implement verification for the other CAS systems. For the integrals whose result was not run through a verification phase, it is assumed that the antiderivative produced was correct.

Verification phase has 3 minutes time out. An integral whose result was not verified could still be correct. Further investigation is needed on those integrals which failed verifications. Such integrals are marked in the summary table below and also in each integral separate section so they are easy to identify and locate.

1.9 Important notes about some of the results

1.9.1 Important note about Maxima results

Since these integrals are run in a batch mode, using an automated script, and by using sagemath (SageMath uses Maxima), then any integral where Maxima needs an interactive response from the user to answer a question during evaluation of the integral in order to complete the integration, will fail and is counted as failed.

The exception raised is `ValueError`. Therefore Maxima result below is lower than what could result if Maxima was run directly and each question Maxima asks was answered correctly.

The percentage of such failures were not counted for each test file, but for an example, for the Timofeev test file, there were about 30 such integrals out of total 705, or about 4 percent. This pecrentage can be higher or lower depending on the specific input test file.

Such integrals can be indentified by looking at the output of the integration in each section for Maxima. If the output was an exception `ValueError` then this is most likely due to this reason.

Maxima integrate was run using SageMath with the following settings set by default

```
'besselexpand : true'
'display2d : false'
'domain : complex'
'keepfloat : true'
'load(to_poly_solve)'
'load(simplify_sum)'
'load(abs_integrate)' 'load(diag)'
```

SageMath loading of Maxima `abs_integrate` was found to cause some problem. So the following code was added to disable this effect.

```
from sage.interfaces.maxima_lib import maxima_lib
maxima_lib.set('extra_definite_integration_methods', '[]')
maxima_lib.set('extra_integration_methods', '[]')
```

See https://ask.sagemath.org/question/43088/integrate-results-that-are-different-from-using-maxima/ for reference.

1.9.2 Important note about FriCAS and Giac/XCAS results

There are Few integrals which failed due to SageMath not able to translate the result back to SageMath syntax and not because these CAS system were not able to do the integrations.

These will fail With error `Exception raised: NotImplementedError`

The number of such cases seems to be very small. About 1 or 2 percent of all integrals.

Hopefully the next version of SageMath will have complete translation of FriCAS and XCAS syntax and I will re-run all the tests again when this happens.

1.9.3 Important note about finding leaf size of antiderivative

For Mathematica, Rubi and Maple, the buildin system function `LeafSize` is used to find the leaf size of each antiderivative.

The other CAS systems (SageMath and Sympy) do not have special buildin function for this purpose at this time. Therefore the leaf size is determined as follows.

For Fricas, Giac and Maxima (all called via sagemath) the following code is used

```
#see https://stackoverflow.com/questions/25202346/how-to-obtain-leaf-count-expression-size-in-s

def tree(expr):
    if expr.operator() is None:
       return expr
    else:
       return [expr.operator()]+map(tree, expr.operands())

try:
    # 1.35 is a fudge factor since this estimate of leaf count is bit lower than
    #what it should be compared to Mathematica's
    leafCount = round(1.35*len(flatten(tree(anti))))
except Exception as ee:
    leafCount =1
```

For Sympy, called directly from Python, the following code is used

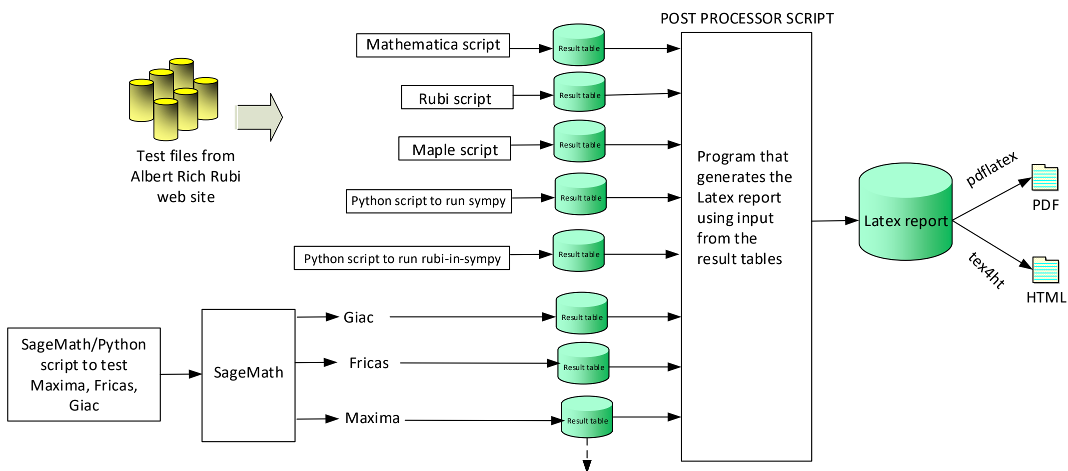
```
try:
  # 1.7 is a fudge factor since it is low side from actual leaf count
  leafCount = round(1.7*count_ops(anti))

  except Exception as ee:
         leafCount =1
```

When these cas systems have a buildin function to find the leaf size of expressions, it will be used instead, and these tests run again.

1.10 Design of the test system

The following diagram gives a high level view of the current test build system.



One record (line) per one integral result. The line is CSV comma separated. It contains 13 fields. This is description of each record (line)

1. integer, the problem number.
2. integer. 0 or 1 for failed or passed. (this is not the grade field)
3. integer. Leaf size of result.
4. integer. Leaf size of the optimal antiderivative.
5. number. CPU time used to solve this integral. 0 if failed.
6. string. The integral in Latex format
7. string. The input used in CAS own syntx.
8. string. The result (antiderivative) produced by CAS in Latex format
9. string. The optimal antiderivative in Latex format.
10. integer. 0 or 1. Indicates if problem has known antiderivative or not
11. String. The result (antiderivative) in CAS own syntax.
12. String. The grade of the antiderivative. Can be “A”, “B”, “C”, or “F”
13. String. The optimal antiderivative in CAS own syntax.

High level overview of the CAS independent integration test build system

Nasser M. Abbasi
June 22, 2018

Chapter 2

detailed summary tables of results

2.1 List of integrals sorted by grade for each CAS

2.1.1 Rubi

A grade: { 1 }

B grade: { }

C grade: { }

F grade: { }

2.1.2 Mathematica

A grade: { 1 }

B grade: { }

C grade: { }

F grade: { }

2.1.3 Maple

A grade: { 1 }

B grade: { }

C grade: { }

F grade: { }

2.1.4 Maxima

A grade: {

B grade: { }

C grade: { }

F grade: { 1 }

2.1.5 FriCAS

A grade: { }

B grade: { }

C grade: { }

F grade: { 1 }

2.1.6 Sympy

A grade: { }

B grade: { }

C grade: { }

F grade: { 1 }

2.1.7 Giac

A grade: { }

B grade: { }

C grade: { }

F grade: { 1 }

2.2 Detailed conclusion table per each integral for all CAS systems

Detailed conclusion table per each integral is given by table below. The elapsed time is in seconds. For failed result it is given as F(-1) if the failure was due to timeout. It is given as F(-2) if the failure was due to an exception being raised, which could indicate a bug in the system. If the failure was due to integral not being evaluated within the time limit, then it is given just an F.

In this table,the column **normalized size** is defined as $\frac{\text{antiderivative leaf size}}{\text{optimal antiderivative leaf size}}$

Problem 1	Optimal	Rubi	Mathematica	Maple	Maxima	Fricas	Sympy	Giac
grade	A	A	A	A	F	F	F(-1)	F
verified	N/A	Yes	Yes	TBD	TBD	TBD	TBD	TBD
size	112	112	133	204	0	0	0	0
normalized size	1	1.	1.19	1.82	0.	0.	0.	0.
time (sec)	N/A	0.18	0.743	1.643	0.	0.	0.	0.

2.3 Detailed conclusion table specific for Rubi results

The following table is specific to Rubi. It gives additional statistics for each integral. the column **steps** is the number of steps used by Rubi to obtain the antiderivative. The **rules** column is the number of unique rules used. The **integrand size** column is the leaf size of the integrand. Finally the ratio $\frac{\text{number of rules}}{\text{integrand size}}$ is given. The larger this ratio is, the harder the integral was to solve. In this test, problem number [1] had the largest ratio of [0.24]

Table 2.1: Rubi specific breakdown of results for each integral

#	grade	number of steps used	number of unique rules	normalized antiderivative leaf size	integrand leaf size	$\frac{\text{number of rules}}{\text{integrand leaf size}}$
1	A	7	6	1.	25	0.24

Chapter 3

Listing of integrals

3.1 $\int \frac{(a+b\csc(x))\left(A+B\csc(x)+C\csc^2(x)\right)}{\sqrt{\csc(x)}}\,dx$

Optimal. Leaf size=112

$$-\frac{2}{3}\sqrt{\sin(x)}\sqrt{\csc(x)}\text{EllipticF}\left(\frac{\pi}{4}-\frac{x}{2},2\right)(3aB+3Ab+bC)+2\sqrt{\sin(x)}\sqrt{\csc(x)}E\left(\frac{\pi}{4}-\frac{x}{2}\middle|2\right)(bB-a(A-C))-2\cos($$

```
[Out] -2*(b*B + a*C)*Cos[x]*Sqrt[Csc[x]] - (2*b*C*Cos[x]*Csc[x]^(3/2))/3 + 2*(b*B
 - a*(A - C))*Sqrt[Csc[x]]*EllipticE[Pi/4 - x/2, 2]*Sqrt[Sin[x]] - (2*(3*A*
b + 3*a*B + b*C)*Sqrt[Csc[x]]*EllipticF[Pi/4 - x/2, 2]*Sqrt[Sin[x]])/3
```

Rubi [A] time = 0.180087, antiderivative size = 112, normalized size of antiderivative = 1., number of steps used = 7, number of rules used = 6, integrand size = 25, $\frac{\text{number of rules}}{\text{integrand size}}$ = 0.24, Rules used = {4076, 4047, 3771, 2641, 4046, 2639}

$$-\frac{2}{3}\sqrt{\sin(x)}\sqrt{\csc(x)}F\left(\frac{\pi}{4}-\frac{x}{2}\middle|2\right)(3aB+3Ab+bC)+2\sqrt{\sin(x)}\sqrt{\csc(x)}E\left(\frac{\pi}{4}-\frac{x}{2}\middle|2\right)(bB-a(A-C))-2\cos(x)\sqrt{\csc($$

Antiderivative was successfully verified.

```
[In] Int[((a + b*Csc[x])*(A + B*Csc[x] + C*Csc[x]^2))/Sqrt[Csc[x]],x]
```

```
[Out] -2*(b*B + a*C)*Cos[x]*Sqrt[Csc[x]] - (2*b*C*Cos[x]*Csc[x]^(3/2))/3 + 2*(b*B
 - a*(A - C))*Sqrt[Csc[x]]*EllipticE[Pi/4 - x/2, 2]*Sqrt[Sin[x]] - (2*(3*A*
b + 3*a*B + b*C)*Sqrt[Csc[x]]*EllipticF[Pi/4 - x/2, 2]*Sqrt[Sin[x]])/3
```

Rule 4076

```
Int[((A_.) + csc[(e_.) + (f_.)*(x_)]*(B_.) + csc[(e_.) + (f_.)*(x_)]^2*(C_.
))*(csc[(e_.) + (f_.)*(x_)]*(d_.))^(n_.)*(csc[(e_.) + (f_.)*(x_)]*(b_.) + (
a_)), x_Symbol] :> -Simp[(b*C*Csc[e + f*x]*Cot[e + f*x]*(d*Csc[e + f*x])^n)
/(f*(n + 2)), x] + Dist[1/(n + 2), Int[(d*Csc[e + f*x])^n*Simp[A*a*(n + 2)
+ (B*a*(n + 2) + b*(C*(n + 1) + A*(n + 2)))*Csc[e + f*x] + (a*C + B*b)*(n +
 2)*Csc[e + f*x]^2, x], x], x] /; FreeQ[{a, b, d, e, f, A, B, C, n}, x] &&
 !LtQ[n, -1]
```

Rule 4047

```
Int[(csc[(e_.) + (f_.)*(x_)]*(b_.))^(m_.)*((A_.) + csc[(e_.) + (f_.)*(x_)]*
(B_.) + csc[(e_.) + (f_.)*(x_)]^2*(C_.)), x_Symbol] :> Dist[B/b, Int[(b*Csc
[e + f*x])^(m + 1), x], x] + Int[(b*Csc[e + f*x])^m*(A + C*Csc[e + f*x]^2),
```

```
x] /; FreeQ[{b, e, f, A, B, C, m}, x]
```

Rule 3771

```
Int[(csc[(c_.) + (d_.)*(x_)]*(b_.))^(n_), x_Symbol] :> Dist[(b*Csc[c + d*x]
)^n*Sin[c + d*x]^n, Int[1/Sin[c + d*x]^n, x], x] /; FreeQ[{b, c, d}, x] &&
EqQ[n^2, 1/4]
```

Rule 2641

```
Int[1/Sqrt[sin[(c_.) + (d_.)*(x_)]], x_Symbol] :> Simp[(2*EllipticF[(1*(c -
 Pi/2 + d*x))/2, 2])/d, x] /; FreeQ[{c, d}, x]
```

Rule 4046

```
Int[(csc[(e_.) + (f_.)*(x_)]*(b_.))^(m_.)*(csc[(e_.) + (f_.)*(x_)]^2*(C_.)
+ (A_)), x_Symbol] :> -Simp[(C*Cot[e + f*x]*(b*Csc[e + f*x])^m)/(f*(m + 1))
, x] + Dist[(C*m + A*(m + 1))/(m + 1), Int[(b*Csc[e + f*x])^m, x], x] /; Fr
eeQ[{b, e, f, A, C, m}, x] && NeQ[C*m + A*(m + 1), 0] &&  !LeQ[m, -1]
```

Rule 2639

```
Int[Sqrt[sin[(c_.) + (d_.)*(x_)]], x_Symbol] :> Simp[(2*EllipticE[(1*(c - P
i/2 + d*x))/2, 2])/d, x] /; FreeQ[{c, d}, x]
```

Rubi steps

$$\int \frac{(a+b\csc(x))\left(A+B\csc(x)+C\csc^2(x)\right)}{\sqrt{\csc(x)}}\,dx = -\frac{2}{3}bC\cos(x)\csc^{\frac{3}{2}}(x)+\frac{2}{3}\int \frac{\frac{3aA}{2}+\frac{1}{2}(3Ab+3aB+bC)\csc(x)+\frac{3}{2}(bB}{\sqrt{\csc(x)}}$$

$$= -\frac{2}{3}bC\cos(x)\csc^{\frac{3}{2}}(x)+\frac{2}{3}\int \frac{\frac{3aA}{2}+\frac{3}{2}(bB+aC)\csc^2(x)}{\sqrt{\csc(x)}}\,dx+\frac{1}{3}(3Ab+$$

$$= -2(bB+aC)\cos(x)\sqrt{\csc(x)}-\frac{2}{3}bC\cos(x)\csc^{\frac{3}{2}}(x)+(-bB+a(A-C))$$

$$= -2(bB+aC)\cos(x)\sqrt{\csc(x)}-\frac{2}{3}bC\cos(x)\csc^{\frac{3}{2}}(x)-\frac{2}{3}(3Ab+3aB+b$$

$$= -2(bB+aC)\cos(x)\sqrt{\csc(x)}-\frac{2}{3}bC\cos(x)\csc^{\frac{3}{2}}(x)+2(bB-a(A-C))$$

Mathematica [A] time = 0.74282, size = 133, normalized size = 1.19

$$-\frac{4(a+b\csc(x))\left(A+B\csc(x)+C\csc^2(x)\right)\left(\sqrt{\sin(x)}\text{EllipticF}\left(\frac{1}{4}(\pi-2x),2\right)(3aB+3Ab+bC)-3\sqrt{\sin(x)}E\left(\frac{1}{4}(\pi-2x\right.\right.}{3\csc^{\frac{5}{2}}(x)(a\sin(x)+b)(A(-\cos(2x))+A+2B\sin(x)+2C}$$

Antiderivative was successfully verified.

```
[In] Integrate[((a + b*Csc[x])*(A + B*Csc[x] + C*Csc[x]^2))/Sqrt[Csc[x]],x]
```

```
[Out] (-4*(a + b*Csc[x])*(A + B*Csc[x] + C*Csc[x]^2)*(3*b*B*Cos[x] + 3*a*C*Cos[x]
 + b*C*Cot[x] - 3*(b*B + a*(-A + C))*EllipticE[(Pi - 2*x)/4, 2]*Sqrt[Sin[x]
] + (3*A*b + 3*a*B + b*C)*EllipticF[(Pi - 2*x)/4, 2]*Sqrt[Sin[x]]))/(3*Csc[
x]^(5/2)*(b + a*Sin[x])*(A + 2*C - A*Cos[2*x] + 2*B*Sin[x]))
```

Maple [A] time = 1.643, size = 204, normalized size = 1.8

$$-\frac{1}{3\cos(x)}\left((6bB+6Ca)\sin(x)(\cos(x))^2+\sqrt{\sin(x)+1}\sqrt{-2\sin(x)+2}\sqrt{-\sin(x)}\left(6A\mathrm{EllipticE}\left(\sqrt{\sin(x)+1},1/2\right.\right.\right.$$

Verification of antiderivative is not currently implemented for this CAS.

```
[In] int((a+b*csc(x))*(A+B*csc(x)+C*csc(x)^2)/csc(x)^(1/2),x)
```

```
[Out] -1/3/sin(x)^(3/2)*((6*B*b+6*C*a)*sin(x)*cos(x)^2+(sin(x)+1)^(1/2)*(-2*sin(x
)+2)^(1/2)*(-sin(x))^(1/2)*(6*A*EllipticE((sin(x)+1)^(1/2),1/2*2^(1/2))*a-3
*A*EllipticF((sin(x)+1)^(1/2),1/2*2^(1/2))*a-3*A*EllipticF((sin(x)+1)^(1/2)
,1/2*2^(1/2))*b-6*B*EllipticE((sin(x)+1)^(1/2),1/2*2^(1/2))*b-3*B*EllipticF
((sin(x)+1)^(1/2),1/2*2^(1/2))*a+3*B*EllipticF((sin(x)+1)^(1/2),1/2*2^(1/2)
)*b-6*C*EllipticE((sin(x)+1)^(1/2),1/2*2^(1/2))*a+3*C*EllipticF((sin(x)+1)^
(1/2),1/2*2^(1/2))*a-C*EllipticF((sin(x)+1)^(1/2),1/2*2^(1/2))*b)*sin(x)+2*
C*b*cos(x)^2)/cos(x)
```

Maxima [F] time = 0., size = 0, normalized size = 0.

$$\int \frac{\left(C\csc(x)^2+B\csc(x)+A\right)(b\csc(x)+a)}{\sqrt{\csc(x)}}\,dx$$

Verification of antiderivative is not currently implemented for this CAS.

```
[In] integrate((a+b*csc(x))*(A+B*csc(x)+C*csc(x)^2)/csc(x)^(1/2),x, algorithm="m
axima")
```

```
[Out] integrate((C*csc(x)^2 + B*csc(x) + A)*(b*csc(x) + a)/sqrt(csc(x)), x)
```

Fricas [F] time = 0., size = 0, normalized size = 0.

$$\mathrm{integral}\left(\frac{Cb\csc(x)^3+(Ca+Bb)\csc(x)^2+Aa+(Ba+Ab)\csc(x)}{\sqrt{\csc(x)}},x\right)$$

Verification of antiderivative is not currently implemented for this CAS.

```
[In] integrate((a+b*csc(x))*(A+B*csc(x)+C*csc(x)^2)/csc(x)^(1/2),x, algorithm="f
ricas")
```

```
[Out] integral((C*b*csc(x)^3 + (C*a + B*b)*csc(x)^2 + A*a + (B*a + A*b)*csc(x))/s
qrt(csc(x)), x)
```

Sympy [F(-1)] time = 0., size = 0, normalized size = 0.

Timed out

Verification of antiderivative is not currently implemented for this CAS.

```
[In] integrate((a+b*csc(x))*(A+B*csc(x)+C*csc(x)**2)/csc(x)**(1/2),x)
```

```
[Out] Timed out
```

Giac [F] time = 0., size = 0, normalized size = 0.

$$\int \frac{\left(C\csc(x)^2 + B\csc(x) + A\right)(b\csc(x) + a)}{\sqrt{\csc(x)}}\,dx$$

Verification of antiderivative is not currently implemented for this CAS.

```
[In] integrate((a+b*csc(x))*(A+B*csc(x)+C*csc(x)^2)/csc(x)^(1/2),x, algorithm="g
iac")
```

```
[Out] integrate((C*csc(x)^2 + B*csc(x) + A)*(b*csc(x) + a)/sqrt(csc(x)), x)
```

Chapter 4

Listing of Grading functions

The following are the current version of the grading functions used for grading the quality of the antiderivative with reference to the optimal antiderivative included in the test suite.

There is a version for Maple and for Mathematica/Rubi. There is a version for grading Sympy and version for use with Sagemath.

The following are links to the current source code.

The following are the listings of source code of the grading functions.

4.0.1 Mathematica and Rubi grading function

```
(* Original version thanks to Albert Rich emailed on 03/21/2017 *)
(* ::Package:: *)

(* ::Subsection:: *)
(*GradeAntiderivative[result,optimal]*)


(* ::Text:: *)
(*If result and optimal are mathematical expressions, *)
(*          GradeAntiderivative[result,optimal] returns*)
(*  "F" if the result fails to integrate an expression that*)
(*      is integrable*)
(*  "C" if result involves higher level functions than necessary*)
(*  "B" if result is more than twice the size of the optimal*)
(*      antiderivative*)
(*  "A" if result can be considered optimal*)


GradeAntiderivative[result_,optimal_] :=
  If[ExpnType[result]<=ExpnType[optimal],
    If[FreeQ[result,Complex] || Not[FreeQ[optimal,Complex]],
      If[LeafCount[result]<=2*LeafCount[optimal],
        "A",
      "B"],
    "C"],
  If[FreeQ[result,Integrate] && FreeQ[result,Int],
    "C",
  "F"]]


(* ::Text:: *)
(*The following summarizes the type number assigned an *)
(*expression based on the functions it involves*)
(*1 = rational function*)
(*2 = algebraic function*)
(*3 = elementary function*)
(*4 = special function*)
```

```
38 (*5 = hyperpergeometric function*)
39 (*6 = appell function*)
40 (*7 = rootsum function*)
41 (*8 = integrate function*)
42 (*9 = unknown function*)
43
44
45 ExpnType[expn_] :=
46   If[AtomQ[expn],
47     1,
48   If[ListQ[expn],
49     Max[Map[ExpnType,expn]],
50   If[Head[expn]===Power,
51     If[IntegerQ[expn[[2]]],
52       ExpnType[expn[[1]]],
53     If[Head[expn[[2]]]===Rational,
54       If[IntegerQ[expn[[1]]] || Head[expn[[1]]]===Rational,
55         1,
56       Max[ExpnType[expn[[1]]],2]],
57     Max[ExpnType[expn[[1]]],ExpnType[expn[[2]]],3]]],
58   If[Head[expn]===Plus || Head[expn]===Times,
59     Max[ExpnType[First[expn]],ExpnType[Rest[expn]]],
60   If[ElementaryFunctionQ[Head[expn]],
61     Max[3,ExpnType[expn[[1]]]],
62   If[SpecialFunctionQ[Head[expn]],
63     Apply[Max,Append[Map[ExpnType,Apply[List,expn]],4]],
64   If[HypergeometricFunctionQ[Head[expn]],
65     Apply[Max,Append[Map[ExpnType,Apply[List,expn]],5]],
66   If[AppellFunctionQ[Head[expn]],
67     Apply[Max,Append[Map[ExpnType,Apply[List,expn]],6]],
68   If[Head[expn]===RootSum,
69     Apply[Max,Append[Map[ExpnType,Apply[List,expn]],7]],
70   If[Head[expn]===Integrate || Head[expn]===Int,
71     Apply[Max,Append[Map[ExpnType,Apply[List,expn]],8]],
72   9]]]]]]]]]]
73
74
75 ElementaryFunctionQ[func_] :=
76   MemberQ[{
77         Exp,Log,
78         Sin,Cos,Tan,Cot,Sec,Csc,
79         ArcSin,ArcCos,ArcTan,ArcCot,ArcSec,ArcCsc,
80         Sinh,Cosh,Tanh,Coth,Sech,Csch,
81         ArcSinh,ArcCosh,ArcTanh,ArcCoth,ArcSech,ArcCsch
82 },func]
83
84
85 SpecialFunctionQ[func_] :=
86   MemberQ[{
87         Erf, Erfc, Erfi,
88         FresnelS, FresnelC,
89         ExpIntegralE, ExpIntegralEi, LogIntegral,
90         SinIntegral, CosIntegral, SinhIntegral, CoshIntegral,
91         Gamma, LogGamma, PolyGamma,
92         Zeta, PolyLog, ProductLog,
93         EllipticF, EllipticE, EllipticPi
94 },func]
95
96
97 HypergeometricFunctionQ[func_] :=
98   MemberQ[{Hypergeometric1F1,Hypergeometric2F1,HypergeometricPFQ},func]
99
100
```

```
101 AppellFunctionQ[func_] :=
102   MemberQ[{AppellF1},func]
```

4.0.2 Maple grading function

```
1  # File: GradeAntiderivative.mpl
2  # Original version thanks to Albert Rich emailed on 03/21/2017
3
4  #Nasser 03/22/2017  Use Maple leaf count instead since buildin
5  #Nasser 03/23/2017  missing 'ln' for ElementaryFunctionQ added
6  #Nasser 03/24/2017  corrected the check for complex result
7  #Nasser 10/27/2017  check for leafsize and do not call ExpnType()
8  #                   if leaf size is "too large". Set at 500,000
9  #Nasser 12/22/2019  Added debug flag, added 'dilog' to special functions
10 #                   see problem 156, file Apostol_Problems
11
12 GradeAntiderivative := proc(result,optimal)
13 local leaf_count_result, leaf_count_optimal,ExpnType_result,ExpnType_optimal,
       debug:=false;
14
15       leaf_count_result:=leafcount(result);
16       #do NOT call ExpnType() if leaf size is too large. Recursion problem
17       if leaf_count_result > 500000 then
18           return "B";
19       fi;
20
21       leaf_count_optimal:=leafcount(optimal);
22
23       ExpnType_result:=ExpnType(result);
24       ExpnType_optimal:=ExpnType(optimal);
25
26       if debug then
27          print("ExpnType_result",ExpnType_result," ExpnType_optimal=",
       ExpnType_optimal);
28       fi;
29
30 # If result and optimal are mathematical expressions,
31 #  GradeAntiderivative[result,optimal] returns
32 #   "F" if the result fails to integrate an expression that
33 #       is integrable
34 #   "C" if result involves higher level functions than necessary
35 #   "B" if result is more than twice the size of the optimal
36 #       antiderivative
37 #   "A" if result can be considered optimal
38
39   #This check below actually is not needed, since I only
40   #call this grading only for passed integrals. i.e. I check
41   #for "F" before calling this. But no harm of keeping it here.
42   #just in case.
43
44
45   if not type(result,freeof('int')) then
46      return "F";
47   end if;
48
49
50   if ExpnType_result<=ExpnType_optimal then
51       if debug then
52            print("ExpnType_result<=ExpnType_optimal");
53       fi;
54       if is_contains_complex(result) then
55          if is_contains_complex(optimal) then
56             if debug then
```

```
57               print("both result and optimal complex");
58            fi;
59            #both result and optimal complex
60            if leaf_count_result<=2*leaf_count_optimal then
61               return "A";
62            else
63               return "B";
64            end if
65         else #result contains complex but optimal is not
66            if debug then
67               print("result contains complex but optimal is not");
68            fi;
69            return "C";
70         end if
71      else # result do not contain complex
72           # this assumes optimal do not as well
73           if debug then
74               print("result do not contain complex, this assumes optimal do
    not as well");
75           fi;
76           if leaf_count_result<=2*leaf_count_optimal then
77              if debug then
78                 print("leaf_count_result<=2*leaf_count_optimal");
79              fi;
80              return "A";
81           else
82              if debug then
83                 print("leaf_count_result>2*leaf_count_optimal");
84              fi;
85              return "B";
86           end if
87      end if
88  else #ExpnType(result) > ExpnType(optimal)
89      if debug then
90         print("ExpnType(result) > ExpnType(optimal)");
91      fi;
92      return "C";
93  end if
94
95 end proc:
96
97 #
98 # is_contains_complex(result)
99 # takes expressions and returns true if it contains "I" else false
100 #
101 #Nasser 032417
102 is_contains_complex:= proc(expression)
103   return (has(expression,I));
104 end proc:
105
106 # The following summarizes the type number assigned an expression
107 # based on the functions it involves
108 # 1 = rational function
109 # 2 = algebraic function
110 # 3 = elementary function
111 # 4 = special function
112 # 5 = hyperpergeometric function
113 # 6 = appell function
114 # 7 = rootsum function
115 # 8 = integrate function
116 # 9 = unknown function
117
118 ExpnType := proc(expn)
```

```
119   if type(expn,'atomic') then
120     1
121   elif type(expn,'list') then
122     apply(max,map(ExpnType,expn))
123   elif type(expn,'sqrt') then
124     if type(op(1,expn),'rational') then
125        1
126     else
127        max(2,ExpnType(op(1,expn)))
128     end if
129   elif type(expn,'`^`') then
130     if type(op(2,expn),'integer') then
131       ExpnType(op(1,expn))
132     elif type(op(2,expn),'rational') then
133       if type(op(1,expn),'rational') then
134          1
135       else
136          max(2,ExpnType(op(1,expn)))
137       end if
138     else
139          max(3,ExpnType(op(1,expn)),ExpnType(op(2,expn)))
140     end if
141   elif type(expn,'`+`') or type(expn,'`*`') then
142     max(ExpnType(op(1,expn)),max(ExpnType(rest(expn))))
143   elif ElementaryFunctionQ(op(0,expn)) then
144     max(3,ExpnType(op(1,expn)))
145   elif SpecialFunctionQ(op(0,expn)) then
146     max(4,apply(max,map(ExpnType,[op(expn)])))
147   elif HypergeometricFunctionQ(op(0,expn)) then
148     max(5,apply(max,map(ExpnType,[op(expn)])))
149   elif AppellFunctionQ(op(0,expn)) then
150     max(6,apply(max,map(ExpnType,[op(expn)])))
151   elif op(0,expn)='int' then
152     max(8,apply(max,map(ExpnType,[op(expn)]))) else
153   9
154   end if
155 end proc:
156
157
158 ElementaryFunctionQ := proc(func)
159   member(func,[
160         exp,log,ln,
161         sin,cos,tan,cot,sec,csc,
162         arcsin,arccos,arctan,arccot,arcsec,arccsc,
163         sinh,cosh,tanh,coth,sech,csch,
164         arcsinh,arccosh,arctanh,arccoth,arcsech,arccsch])
165 end proc:
166
167 SpecialFunctionQ := proc(func)
168   member(func,[
169         erf,erfc,erfi,
170         FresnelS,FresnelC,
171         Ei,Ei,Li,Si,Ci,Shi,Chi,
172         GAMMA,lnGAMMA,Psi,Zeta,polylog,dilog,LambertW,
173         EllipticF,EllipticE,EllipticPi])
174 end proc:
175
176 HypergeometricFunctionQ := proc(func)
177   member(func,[Hypergeometric1F1,hypergeom,HypergeometricPFQ])
178 end proc:
179
180 AppellFunctionQ := proc(func)
181   member(func,[AppellF1])
```

```
182 end proc:
183
184 # u is a sum or product.  rest(u) returns all but the
185 # first term or factor of u.
186 rest := proc(u) local v;
187   if nops(u)=2 then
188      op(2,u)
189   else
190      apply(op(0,u),op(2..nops(u),u))
191   end if
192 end proc:
193
194 #leafcount(u) returns the number of nodes in u.
195 #Nasser 3/23/17  Replaced by build-in leafCount from package in Maple
196 leafcount := proc(u)
197    MmaTranslator[Mma][LeafCount](u);
198 end proc:
```

4.0.3 Sympy grading function

```
1 #Dec 24, 2019. Nasser M. Abbasi:
2 #              Port of original Maple grading function by
3 #              Albert Rich to use with Sympy/Python
4 #Dec 27, 2019  Nasser. Added `RootSum`. See problem 177, Timofeev file
5 #              added 'exp_polar'
6 from sympy import *
7
8 def leaf_count(expr):
9     #sympy do not have leaf count function. This is approximation
10     return round(1.7*count_ops(expr))
11
12 def is_sqrt(expr):
13     if isinstance(expr,Pow):
14         if expr.args[1] == Rational(1,2):
15             return True
16         else:
17             return False
18     else:
19         return False
20
21 def is_elementary_function(func):
22     return func in [exp,log,ln,sin,cos,tan,cot,sec,csc,
23             asin,acos,atan,acot,asec,acsc,sinh,cosh,tanh,coth,sech,csch,
24             asinh,acosh,atanh,acoth,asech,acsch
25         ]
26
27 def is_special_function(func):
28     return func in [ erf,erfc,erfi,
29              fresnels,fresnelc,Ei,Ei,Li,Si,Ci,Shi,Chi,
30              gamma,loggamma,digamma,zeta,polylog,LambertW,
31              elliptic_f,elliptic_e,elliptic_pi,exp_polar
32          ]
33
34 def is_hypergeometric_function(func):
35     return func in [hyper]
36
37 def is_appell_function(func):
38     return func in [appellf1]
39
40 def is_atom(expn):
41     try:
42         if expn.isAtom or isinstance(expn,int) or isinstance(expn,float):
43            return True
```

```
44          else:
45             return False
46
47      except AttributeError as error:
48          return False
49
50  def expnType(expn):
51      debug=False
52      if debug:
53          print("expn=",expn,"type(expn)=",type(expn))
54
55      if is_atom(expn):
56         return 1
57      elif isinstance(expn,list):
58          return max(map(expnType, expn))   #apply(max,map(ExpnType,expn))
59      elif  is_sqrt(expn):
60          if isinstance(expn.args[0],Rational): #type(op(1,expn),'rational')
61              return 1
62          else:
63              return max(2,expnType(expn.args[0]))  #max(2,ExpnType(op(1,expn)))
64      elif isinstance(expn,Pow):   #type(expn,'`^`')
65          if isinstance(expn.args[1],Integer):  #type(op(2,expn),'integer')
66              return expnType(expn.args[0])   #ExpnType(op(1,expn))
67          elif isinstance(expn.args[1],Rational):  #type(op(2,expn),'rational')
68              if isinstance(expn.args[0],Rational): #type(op(1,expn),'rational')
69                  return 1
70              else:
71                  return max(2,expnType(expn.args[0]))  #max(2,ExpnType(op(1,expn
    )))
72          else:
73              return max(3,expnType(expn.args[0]),expnType(expn.args[1])) #max(3,
    ExpnType(op(1,expn)),ExpnType(op(2,expn)))
74      elif isinstance(expn,Add) or isinstance(expn,Mul): #type(expn,'`+`') or
    type(expn,'`*`')
75          m1 = expnType(expn.args[0])
76          m2 = expnType(list(expn.args[1:]))
77          return max(m1,m2)  #max(ExpnType(op(1,expn)),max(ExpnType(rest(expn))))
78      elif is_elementary_function(expn.func):  #ElementaryFunctionQ(op(0,expn))
79          return max(3,expnType(expn.args[0]))  #max(3,ExpnType(op(1,expn)))
80      elif is_special_function(expn.func): #SpecialFunctionQ(op(0,expn))
81          m1 = max(map(expnType, list(expn.args)))
82          return max(4,m1)   #max(4,apply(max,map(ExpnType,[op(expn)])))
83      elif is_hypergeometric_function(expn.func): #HypergeometricFunctionQ(op(0,
    expn))
84          m1 = max(map(expnType, list(expn.args)))
85          return max(5,m1)   #max(5,apply(max,map(ExpnType,[op(expn)])))
86      elif is_appell_function(expn.func):
87          m1 = max(map(expnType, list(expn.args)))
88          return max(6,m1)   #max(5,apply(max,map(ExpnType,[op(expn)])))
89      elif isinstance(expn,RootSum):
90          m1 = max(map(expnType, list(expn.args))) #Apply[Max,Append[Map[ExpnType
    ,Apply[List,expn]],7]],
91          return max(7,m1)
92      elif str(expn).find("Integral") != -1:
93          m1 = max(map(expnType, list(expn.args)))
94          return max(8,m1)   #max(5,apply(max,map(ExpnType,[op(expn)])))
95      else:
96          return 9
97
98  #main function
99  def grade_antiderivative(result,optimal):
100
101     leaf_count_result  = leaf_count(result)
```

```
102     leaf_count_optimal = leaf_count(optimal)
103 
104     expnType_result  = expnType(result)
105     expnType_optimal = expnType(optimal)
106 
107     if str(result).find("Integral") != -1:
108         return "F"
109 
110     if expnType_result <= expnType_optimal:
111         if result.has(I):
112             if optimal.has(I): #both result and optimal complex
113                 if leaf_count_result <= 2*leaf_count_optimal:
114                     return "A"
115                 else:
116                     return "B"
117             else: #result contains complex but optimal is not
118                 return "C"
119         else: # result do not contain complex, this assumes optimal do not as
    well
120             if leaf_count_result <= 2*leaf_count_optimal:
121                 return "A"
122             else:
123                 return "B"
124     else:
125         return "C"
```

4.0.4 SageMath grading function

```
1 #Dec 24, 2019. Nasser: Ported original Maple grading function by
2 #              Albert Rich to use with Sagemath. This is used to
3 #              grade Fricas, Giac and Maxima results.
4 #Dec 24, 2019. Nasser: Added 'exp_integral_e' and 'sng', 'sin_integral'
5 #              'arctan2','floor','abs','log_integral'
6 
7 from sage.all import *
8 from sage.symbolic.operators import add_vararg, mul_vararg
9 
10 def tree(expr):
11     debug=False;
12     if debug:
13          print ("Enter tree(expr), expr=",expr)
14          print ("expr.operator()=",expr.operator())
15          print ("expr.operands()=",expr.operands())
16          print ("map(tree, expr.operands()=",map(tree, expr.operands()))
17 
18     if expr.operator() is None:
19         return expr
20     else:
21         return [expr.operator()]+list(map(tree, expr.operands()))
22 
23 def leaf_count(anti):
24     debug=False;
25 
26     if debug: print ("Enter leaf_count, anti=", anti, " len(anti)=", len(anti))
27 
28     if len(anti) == 0: #special check for optimal being 0 for some test cases.
29        if debug: print ("len(anti) == 0")
30        return 1
31     else:
32         if debug: print ("round(1.35*len(flatten(tree(anti))))=",round(1.35*len
    (flatten(tree(anti)))))
33         return round(1.35*len(flatten(tree(anti)))) #fudge factor
34                          #since this estimate of leaf count is bit lower than
```

```
35                      #what it should be compared to Mathematica's
36
37 def is_sqrt(expr):
38     debug=False;
39     if expr.operator() == operator.pow:   #isinstance(expr,Pow):
40         if expr.operands()[1]==1/2: #expr.args[1] == Rational(1,2):
41             if debug: print ("expr is sqrt")
42             return True
43         else:
44             return False
45     else:
46         return False
47
48 def is_elementary_function(func):
49     debug = False
50
51     m = func.name() in ['exp','log','ln',
52             'sin','cos','tan','cot','sec','csc',
53             'arcsin','arccos','arctan','arccot','arcsec','arccsc',
54             'sinh','cosh','tanh','coth','sech','csch',
55             'arcsinh','arccosh','arctanh','arccoth','arcsech','arccsch','sgn',
56         'arctan2','floor','abs'
57         ]
58     if debug:
59         if m:
60             print ("func ", func , " is elementary_function")
61         else:
62             print ("func ", func , " is NOT elementary_function")
63
64
65     return m
66
67 def is_special_function(func):
68     debug = False
69
70     if debug: print ("type(func)=", type(func))
71
72     m= func.name() in ['erf','erfc','erfi','fresnel_sin','fresnel_cos','Ei',
73            'Ei','Li','Si','sin_integral','Ci','cos_integral','Shi','
    sinh_integral'
74            'Chi','cosh_integral','gamma','log_gamma','psi,zeta',
75            'polylog','lambert_w','elliptic_f','elliptic_e',
76            'elliptic_pi','exp_integral_e','log_integral']
77
78     if debug:
79         print ("m=",m)
80         if m:
81             print ("func ", func ," is special_function")
82         else:
83             print ("func ", func ," is NOT special_function")
84
85
86     return m
87
88
89 def is_hypergeometric_function(func):
90     return func.name() in ['hypergeometric','hypergeometric_M','
    hypergeometric_U']
91
92 def is_appell_function(func):
93     return func.name() in ['hypergeometric']   #[appellf1] can't find this in
    sagemath
94
```

```
95  def is_atom(expn):
96
97      #thanks to answer at https://ask.sagemath.org/question/49179/what-is-
    sagemath-equivalent-to-atomic-type-in-maple/
98      try:
99          if expn.parent() is SR:
100             return expn.operator() is None
101         if expn.parent() in (ZZ, QQ, AA, QQbar):
102             return expn in expn.parent() # Should always return True
103         if hasattr(expn.parent(),"base_ring") and hasattr(expn.parent(),"gens")
    :
104             return expn in expn.parent().base_ring() or expn in expn.parent().
    gens()
105         return False
106
107     except AttributeError as error:
108         return False
109
110
111 def expnType(expn):
112     debug=False
113
114     if debug:
115         print (">>>>>Enter expnType, expn=", expn)
116         print (">>>>>is_atom(expn)=", is_atom(expn))
117
118     if is_atom(expn):
119         return 1
120     elif type(expn)==list:   #isinstance(expn,list):
121         return max(map(expnType, expn))   #apply(max,map(ExpnType,expn))
122     elif  is_sqrt(expn):
123         if  type(expn.operands()[0])==Rational: #type(isinstance(expn.args[0],
    Rational):
124             return 1
125         else:
126             return max(2,expnType(expn.operands()[0]))  #max(2,expnType(expn.
    args[0]))
127     elif expn.operator() == operator.pow:   #isinstance(expn,Pow)
128         if type(expn.operands()[1])==Integer:  #isinstance(expn.args[1],Integer
    )
129             return expnType(expn.operands()[0])   #expnType(expn.args[0])
130         elif type(expn.operands()[1])==Rational:  #isinstance(expn.args[1],
    Rational)
131             if type(expn.operands()[0])==Rational: #isinstance(expn.args[0],
    Rational)
132                 return 1
133             else:
134                 return max(2,expnType(expn.operands()[0]))  #max(2,expnType(
    expn.args[0]))
135         else:
136             return max(3,expnType(expn.operands()[0]),expnType(expn.operands()
    [1])) #max(3,expnType(expn.operands()[0]),expnType(expn.operands()[1]))
137     elif expn.operator() == add_vararg or expn.operator() == mul_vararg: #
    isinstance(expn,Add) or isinstance(expn,Mul)
138         m1 = expnType(expn.operands()[0]) #expnType(expn.args[0])
139         m2 = expnType(expn.operands()[1:]) #expnType(list(expn.args[1:]))
140         return max(m1,m2)  #max(ExpnType(op(1,expn)),max(ExpnType(rest(expn))))
141     elif is_elementary_function(expn.operator()):  #is_elementary_function(expn
    .func)
142         return max(3,expnType(expn.operands()[0]))
143     elif is_special_function(expn.operator()): #is_special_function(expn.func)
144         m1 = max(map(expnType, expn.operands()))      #max(map(expnType, list(
    expn.args)))
```

```
145        return max(4,m1)   #max(4,m1)
146    elif is_hypergeometric_function(expn.operator()): #
    is_hypergeometric_function(expn.func)
147        m1 = max(map(expnType, expn.operands()))       #max(map(expnType, list(
    expn.args)))
148        return max(5,m1)   #max(5,m1)
149    elif is_appell_function(expn.operator()):
150        m1 = max(map(expnType, expn.operands()))       #max(map(expnType, list(
    expn.args)))
151        return max(6,m1)   #max(6,m1)
152    elif str(expn).find("Integral") != -1: #this will never happen, since it
153                #is checked before calling the grading function that is passed.
154                #but kept it here.
155        m1 = max(map(expnType, expn.operands()))       #max(map(expnType, list(
    expn.args)))
156        return max(8,m1)   #max(5,apply(max,map(ExpnType,[op(expn)])))
157    else:
158        return 9
159
160 #main function
161 def grade_antiderivative(result,optimal):
162    debug = False;
163
164    if debug: print ("Enter grade_antiderivative for sagemath")
165
166    leaf_count_result  = leaf_count(result)
167    leaf_count_optimal = leaf_count(optimal)
168
169    if debug: print ("leaf_count_result=", leaf_count_result, "
    leaf_count_optimal=",leaf_count_optimal)
170
171
172    expnType_result  = expnType(result)
173    expnType_optimal = expnType(optimal)
174
175    if debug: print ("expnType_result=", expnType_result, "expnType_optimal=",
    expnType_optimal)
176
177    if expnType_result <= expnType_optimal:
178        if result.has(I):
179            if optimal.has(I): #both result and optimal complex
180                if leaf_count_result <= 2*leaf_count_optimal:
181                    return "A"
182                else:
183                    return "B"
184            else: #result contains complex but optimal is not
185                return "C"
186        else: # result do not contain complex, this assumes optimal do not as
    well
187            if leaf_count_result <= 2*leaf_count_optimal:
188                return "A"
189            else:
190                return "B"
191    else:
192        return "C"
```