The Risch Algorithm for Symbolic Integration in SymPy

Aaron Meurer

July 14, 2011
Over the summer of 2010, I worked for the Python Software Organization with the SymPy project under the Google Summer of Code program to implement the transcendental Risch Algorithm in SymPy.
- End result: `risch_integrate()` function
- Orders of magnitude faster than old `integrate()` function
- Able to solve much more
- And it can prove that integrals are nonelementary!
In [1]: f1 = (1 + exp(x**2))*(1 - x - 4*x**2*exp(x**2)*log(x) - x*exp(x**2) + 4*x**3*exp(x**2) + exp(x**2))/(x*(x - log(x))**2)

In [2]: f1
Out[2]:
\[
\frac{\left(1 + e^x\right) \left(1 - x - 4 \cdot x \cdot e^{2x} \cdot \log(x) - x \cdot e^x + 4 \cdot x^3 \cdot e^{2x} + e^{2x}\right)}{x \cdot (x - \log(x))}
\]

In [3]: factor(cancel(risch_integrate(f1, x)))
Out[3]:
\[
\frac{\left(1 + e^x\right)^2}{x - \log(x)}
\]

In [4]: %timeit risch_integrate(f1, x)
1 loops, best of 3: 390 ms per loop
In [5]: risch_integrate(exp(-x**2), x)
Out[5]:
\[ \int -x^2 e^x \, dx \]

In [7]: risch_integrate(1/log(x), x)
Out[7]:
\[ \int \frac{1}{\log(x)} \, dx \]

In [10]: risch_integrate(exp(x)*log(x), x)
Out[10]:
\[ \int x e^{\log(x)} \, dx \]

In [11]: risch_integrate(exp(exp(x)), x)
Out[11]:
\[ \int (e^x)^x e^{e^x} \, dx \]
Let’s look at how long it takes to compute \( \int x^{10} e^x \, dx \) and \( \int x^{20} e^x \, dx \).

In [14]: %timeit integrate(x**10*exp(x), x)
1 loops, best of 3: 2.75 s per loop

In [15]: %timeit integrate(x**20*exp(x), x)
1 loops, best of 3: 31.5 s per loop

In [16]: %timeit risch_integrate(x**10*exp(x), x)
10 loops, best of 3: 75.9 ms per loop

In [17]: %timeit risch_integrate(x**20*exp(x), x)
10 loops, best of 3: 98 ms per loop

(integrate() is the old function and risch_integrate() is the new function)
Time of integrate() and risch_integrate() to evaluate $\int x^n e^x \, dx$

The new implementation is both orders of magnitude faster, and asymptotically faster!
def no_cancel_b_small(b, c, n, DE):
    
    Poly Risch Differential Equation - No cancelation: deg(b) small enough.

    Given a derivation D on k[t], n either an integer or +∞, and b, c in k[t] with deg(b) < deg(D) - 1 and either D = d/dt or deg(D) >= 2, either raise NonElementaryIntegralException, in which case, equation Dq + bq == c has no solution of degree at most n in k[t], or a solution q in k[t] of this equation with deg(q) <= n, or the tuple (h, b0, c0) such that h ∈ k[t], b0, c0 ∈ k, and for any solution q in k[t] of degree at most n of Dq + bq = c, y = q - h is a solution in k of Dy + b0y = c0.

    q = Poly(0, DE.t)

    while not c.is_zero:
        if n == 0:
            m = 0
        else:
            m = c.degree(DE.t) - DE.d.degree(DE.t) + 1

        if not 0 <= m <= n:
            # n < 0 or m < 0 or m > n
            raise NonElementaryIntegralException

        if m > 0:
            p = Poly(c.as_poly(DE.t).LC()/(m*DE.d.as_poly(DE.t).LC())*DE.t**m, DE.t, expand=False)
            q = q + p
            n = m - 1
            c = c - derivation(p, DE) - b*p
        else:
            if deg(b) != deg(c):
                raise NonElementaryIntegralException

            if deg(b) == 0:
                if not 0 <= m <= n:
                    # n < 0 or m < 0 or m > n
                    raise NonElementaryIntegralException

                p = Poly(c.as_poly(DE.t).LC()/(m*DE.d.as_poly(DE.t).LC())*DE.t**m, DE.t, expand=False)

            return q, b.as_poly(DE.T[DE.level - 1]), c.as_poly(DE.T[DE.level - 1])

        return q

    return q + p

n = m - 1
c = c - derivation(p, DE) - b*p

return q
- You can download SymPy at www.sympy.org
- Or pip install sympy (We just released 0.7.0)
- My development branch is at www.github.com/asmeurer/sympy/tree/integration3 (it isn't merged into the main repo yet)