Conditions of Convergence for Integral (1)

We can without loss of generality assume $p \leq q$, since the G-functions of indices m, n, p, q and of indices n, m, q, p can be related easily (see e.g. [Luke1969], section 5.3). We introduce the following notation:

$$\xi = m + n - p$$
$$\delta = m + n - \frac{p+q}{2}$$
$$C_3 : -\Re(b_j) < 1 \text{ for } j = 1, \ldots, m$$
$$0 < -\Re(a_j) \text{ for } j = 1, \ldots, n$$
$$C_3^* : -\Re(b_j) < 1 \text{ for } j = 1, \ldots, q$$
$$0 < -\Re(a_j) \text{ for } j = 1, \ldots, p$$
$$C_4 : -\Re(\delta) + \frac{q+1-p}{2} > q - p$$

The convergence conditions will be detailed in several "cases", numbered one to five. For later use it will be helpful to separate conditions "at infinity" from conditions "at zero". By conditions "at infinity" we mean conditions that only depend on the behaviour of the integrand for large, positive values of x, whereas by conditions "at zero" we mean conditions that only depend on the behaviour of the integrand on $(0, \epsilon)$ for any $\epsilon > 0$. Since all our conditions are specified in terms of parameters of the G-functions, this distinction is not immediately visible. They are, however, of very distinct character mathematically; the conditions at infinity being in particular much harder to control.

In order for the integral theorem to be valid, conditions n "at zero" and "at infinity" both have to be fulfilled, for some n.

These are the conditions "at infinity":

1.
$$\delta > 0 \wedge |\arg(\eta)| < \delta\pi \wedge (A \vee B \vee C),$$

where

$$A = 1 \leq n \wedge p < q \wedge 1 \leq m$$
$$B = 1 \leq p \wedge 1 \leq m \wedge q = p + 1 \wedge \neg(n = 0 \wedge m = p + 1)$$
$$C = 1 \leq n \wedge q = p \wedge |\arg(\eta)| \neq (\delta - 2k)\pi \text{ for } k = 0, 1, \ldots \left\lceil \frac{\delta}{2} \right\rceil.$$

2.
$$n = 0 \wedge p + 1 \leq m \wedge |\arg(\eta)| < \delta\pi$$

3.
$$(p < q \wedge 1 \leq m \wedge \delta > 0 \wedge |\arg(\eta)| = \delta\pi) \vee (p \leq q - 2 \wedge \delta = 0 \wedge \arg(\eta) = 0)$$

4.
$$p = q \wedge \delta = 0 \wedge \arg(\eta) = 0 \wedge \eta \neq 0 \wedge \Re\left(\sum_{j=1}^{p} b_j - a_j\right) < 0$$

5.

$$\delta > 0 \wedge |\arg(\eta)| < \delta\pi$$

And these are the conditions “at zero”:

1.

$$\eta \neq 0 \wedge C_3$$

2.

$$C_3$$

3.

$$C_3 \wedge C_4$$

4.

$$C_3$$

5.

$$C_3$$

Conditions of Convergence for Integral (2)

We introduce the following notation:

$$b^* = s + t - \frac{u+v}{2}$$

$$c^* = m + n - \frac{p+q}{2}$$

$$\rho = \sum_{j=1}^{v} d_j - \sum_{j=1}^{u} c_j + \frac{u-v}{2} + 1$$

$$\mu = \sum_{j=1}^{q} b_j - \sum_{j=1}^{p} a_j + \frac{p-q}{2} + 1$$

$$\phi = q - p - \frac{u-v}{2} + 1$$

$$\eta = 1 - (v-u) - \mu - \rho$$

$$\psi = \frac{\pi(q-m-n) + |\arg(\omega)|}{q-p}$$

$$\theta = \frac{\pi(v-s-t) + |\arg(\sigma)|)}{v-u}$$

$$\lambda_c = (q-p)|\omega|^{1/(q-p)}\cos\psi + (v-u)|\sigma|^{1/(v-u)}\cos\theta$$

$$\lambda_{s0}(c_1, c_2) = c_1(q-p)|\omega|^{1/(q-p)}\sin\psi + c_2(v-u)|\sigma|^{1/(v-u)}\sin\theta$$

$$\lambda_s = \begin{cases} \lambda_{s0}\left(-1,-1\right)\lambda_{s0}\left(1,1\right) & \text{for } \arg(\omega)=0 \wedge \arg(\sigma)=0 \\ \lambda_{s0}\left(\operatorname{sign}\left(\arg\left(\omega\right)\right),-1\right)\lambda_{s0}\left(\operatorname{sign}\left(\arg\left(\omega\right)\right),1\right) & \text{for } \arg(\omega)\neq 0 \wedge \arg(\sigma)=0 \\ \lambda_{s0}\left(-1,\operatorname{sign}\left(\arg\left(\sigma\right)\right)\right)\lambda_{s0}\left(1,\operatorname{sign}\left(\arg\left(\sigma\right)\right)\right) & \text{for } \arg(\omega)=0 \wedge \arg(\sigma)\neq 0) \\ \lambda_{s0}\left(\operatorname{sign}\left(\arg\left(\omega\right)\right),\operatorname{sign}\left(\arg\left(\sigma\right)\right)\right) & \text{otherwise} \end{cases}$$

$$z_0 = \frac{\omega}{\sigma}e^{-i\pi(b^*+c^*)}$$

$$z_1 = \frac{\sigma}{\omega}e^{-i\pi(b^*+c^*)}$$

The following conditions will be helpful:

$$C_1 : (a_i - b_j \notin \mathbb{Z}_{>0} \text{ for } i=1,\ldots,n, j=1,\ldots,m) \wedge (c_i - d_j \notin \mathbb{Z}_{>0} \text{ for } i=1,\ldots,t, j=1,\ldots,s)$$

$$C_2 : \Re(1+b_i+d_j) > 0 \text{ for } i=1,\ldots,m, j=1,\ldots,s$$

$$C_3 : \Re(a_i+c_j) < 1 \text{ for } i=1,\ldots,n, j=1,\ldots,t$$

$$C_4 : (p-q)\Re(c_i) - \Re(\mu) > -\frac{3}{2} \text{ for } i=1,\ldots,t$$

$$C_5 : (p-q)\Re(1+d_i) - \Re(\mu) > -\frac{3}{2} \text{ for } i=1,\ldots,s$$

$$C_6 : (u-v)\Re(a_i) - \Re(\rho) > -\frac{3}{2} \text{ for } i=1,\ldots,n$$

$$C_7 : (u-v)\Re(1+b_i) - \Re(\rho) > -\frac{3}{2} \text{ for } i=1,\ldots,m$$

$$C_8 : 0 < |\phi| + 2\Re\left((\mu-1)(-u+v) + (-p+q)(\rho-1) + (-p+q)(-u+v)\right)$$

$$C_9 : 0 < |\phi| - 2\Re\left((\mu-1)(-u+v) + (-p+q)(\rho-1) + (-p+q)(-u+v)\right)$$

$$C_{10} : |\arg\left(\sigma\right)| < \pi b^*$$

$$C_{11} : |\arg\left(\sigma\right)| = \pi b^*$$

$$C_{12} : |\arg(\omega)| < c^*\pi$$

$$C_{13} : |\arg(\omega)| = c^*\pi$$

$$C_{14}^1 : (z_0 \neq 1 \wedge |\arg(1-z_0)| < \pi) \vee (z_0 = 1 \wedge \Re(\mu+\rho-u+v) < 1)$$

$$C_{14}^2 : (z_1 \neq 1 \wedge |\arg(1-z_1)| < \pi) \vee (z_1 = 1 \wedge \Re(\mu+\rho-p+q) < 1)$$

$$C_{14} : \phi = 0 \wedge b^* + c^* \leq 1 \wedge (C_{14}^1 \vee C_{14}^2)$$

$$C_{15} : \lambda_c > 0 \vee (\lambda_c = 0 \wedge \lambda_s \neq 0 \wedge \Re(\eta) > -1) \vee (\lambda_c = 0 \wedge \lambda_s = 0 \wedge \Re(\eta) > 0)$$

$$C_{16} : \int_0^\infty G_{u,v}^{s,t}(\sigma x)\mathrm{d}x \text{ converges at infinity}$$

$$C_{17} : \int_0^\infty G_{p,q}^{m,n}(\omega x)\mathrm{d}x \text{ converges at infinity}$$

Note that C_{16} and C_{17} are the reason we split the convergence conditions for integral (1).

With this notation established, the implemented convergence conditions can be enumerated as follows:

1.

$$mnst \neq 0 \wedge 0 < b^* \wedge 0 < c^* \wedge C_1 \wedge C_2 \wedge C_3 \wedge C_{10} \wedge C_{12}$$

2.
$$u = v \wedge b^* = 0 \wedge 0 < c^* \wedge 0 < \sigma \wedge \Re\rho < 1 \wedge C_1 \wedge C_2 \wedge C_3 \wedge C_{12}$$

3.
$$p = q \wedge u = v \wedge b^* = 0 \wedge c^* = 0 \wedge 0 < \sigma \wedge 0 < \omega \wedge \Re\mu < 1 \wedge \Re\rho < 1 \wedge \sigma \neq \omega \wedge C_1 \wedge C_2 \wedge C_3$$

4.
$$p = q \wedge u = v \wedge b^* = 0 \wedge c^* = 0 \wedge 0 < \sigma \wedge 0 < \omega \wedge \Re\left(\mu + \rho\right) < 1 \wedge \omega \neq \sigma \wedge C_1 \wedge C_2 \wedge C_3$$

5.
$$p = q \wedge u = v \wedge b^* = 0 \wedge c^* = 0 \wedge 0 < \sigma \wedge 0 < \omega \wedge \Re\left(\mu + \rho\right) < 1 \wedge \omega \neq \sigma \wedge C_1 \wedge C_2 \wedge C_3$$

6.
$$q < p \wedge 0 < s \wedge 0 < b^* \wedge 0 \leq c^* \wedge C_1 \wedge C_2 \wedge C_3 \wedge C_5 \wedge C_{10} \wedge C_{13}$$

7.
$$p < q \wedge 0 < t \wedge 0 < b^* \wedge 0 \leq c^* \wedge C_1 \wedge C_2 \wedge C_3 \wedge C_4 \wedge C_{10} \wedge C_{13}$$

8.
$$v < u \wedge 0 < m \wedge 0 < c^* \wedge 0 \leq b^* \wedge C_1 \wedge C_2 \wedge C_3 \wedge C_7 \wedge C_{11} \wedge C_{12}$$

9.
$$u < v \wedge 0 < n \wedge 0 < c^* \wedge 0 \leq b^* \wedge C_1 \wedge C_2 \wedge C_3 \wedge C_6 \wedge C_{11} \wedge C_{12}$$

10.
$$q < p \wedge u = v \wedge b^* = 0 \wedge 0 \leq c^* \wedge 0 < \sigma \wedge \Re\rho < 1 \wedge C_1 \wedge C_2 \wedge C_3 \wedge C_5 \wedge C_{13}$$

11.
$$p < q \wedge u = v \wedge b^* = 0 \wedge 0 \leq c^* \wedge 0 < \sigma \wedge \Re\rho < 1 \wedge C_1 \wedge C_2 \wedge C_3 \wedge C_4 \wedge C_{13}$$

12.
$$p = q \wedge v < u \wedge 0 \leq b^* \wedge c^* = 0 \wedge 0 < \omega \wedge \Re\mu < 1 \wedge C_1 \wedge C_2 \wedge C_3 \wedge C_7 \wedge C_{11}$$

13.
$$p = q \wedge u < v \wedge 0 \leq b^* \wedge c^* = 0 \wedge 0 < \omega \wedge \Re\mu < 1 \wedge C_1 \wedge C_2 \wedge C_3 \wedge C_6 \wedge C_{11}$$

14.
$$p < q \wedge v < u \wedge 0 \leq b^* \wedge 0 \leq c^* \wedge C_1 \wedge C_2 \wedge C_3 \wedge C_4 \wedge C_7 \wedge C_{11} \wedge C_{13}$$

15.

$$q < p \wedge u < v \wedge 0 \leq b^* \wedge 0 \leq c^* \wedge C_1 \wedge C_2 \wedge C_3 \wedge C_5 \wedge C_6 \wedge C_{11} \wedge C_{13}$$

16.

$$q < p \wedge v < u \wedge 0 \leq b^* \wedge 0 \leq c^* \wedge C_1 \wedge C_2 \wedge C_3 \wedge C_5 \wedge C_7 \wedge C_8 \wedge C_{11} \wedge C_{13} \wedge C_{14}$$

17.

$$p < q \wedge u < v \wedge 0 \leq b^* \wedge 0 \leq c^* \wedge C_1 \wedge C_2 \wedge C_3 \wedge C_4 \wedge C_6 \wedge C_9 \wedge C_{11} \wedge C_{13} \wedge C_{14}$$

18.

$$t = 0 \wedge 0 < s \wedge 0 < b^* \wedge 0 < \phi \wedge C_1 \wedge C_2 \wedge C_{10}$$

19.

$$s = 0 \wedge 0 < t \wedge 0 < b^* \wedge \phi < 0 \wedge C_1 \wedge C_3 \wedge C_{10}$$

20.

$$n = 0 \wedge 0 < m \wedge 0 < c^* \wedge \phi < 0 \wedge C_1 \wedge C_2 \wedge C_{12}$$

21.

$$m = 0 \wedge 0 < n \wedge 0 < c^* \wedge 0 < \phi \wedge C_1 \wedge C_3 \wedge C_{12}$$

22.

$$st = 0 \wedge 0 < b^* \wedge 0 < c^* \wedge C_1 \wedge C_2 \wedge C_3 \wedge C_{10} \wedge C_{12}$$

23.

$$mn = 0 \wedge 0 < b^* \wedge 0 < c^* \wedge C_1 \wedge C_2 \wedge C_3 \wedge C_{10} \wedge C_{12}$$

24.

$$p < m + n \wedge t = 0 \wedge \phi = 0 \wedge 0 < s \wedge 0 < b^* \wedge c^* < 0 \wedge |\arg(\omega)| < \pi(m + n - p + 1) \wedge C_1 \wedge C_2 \wedge C_{10} \wedge C_{14} \wedge C_{15}$$

25.

$$q < m + n \wedge s = 0 \wedge \phi = 0 \wedge 0 < t \wedge 0 < b^* \wedge c^* < 0 \wedge |\arg(\omega)| < \pi(m + n - q + 1) \wedge C_1 \wedge C_3 \wedge C_{10} \wedge C_{14} \wedge C_{15}$$

26.

$$p = q - 1 \wedge t = 0 \wedge \phi = 0 \wedge 0 < s \wedge 0 < b^* \wedge 0 \leq c^* \wedge \pi c^* < |\arg(\omega)| \wedge C_1 \wedge C_2 \wedge C_{10} \wedge C_{14} \wedge C_{15}$$

27.

$$p = q + 1 \wedge s = 0 \wedge \phi = 0 \wedge 0 < t \wedge 0 < b^* \wedge 0 \leq c^* \wedge \pi c^* < |\arg(\omega)| \wedge C_1 \wedge C_3 \wedge C_{10} \wedge C_{14} \wedge C_{15}$$

28.

$$p < q - 1 \wedge t = 0 \wedge \phi = 0 \wedge 0 < s \wedge 0 < b^* \wedge 0 \leq c^* \wedge \pi c^* < |\arg(\omega)| \wedge |\arg(\omega)| < \pi(m + n - p + 1) \wedge C_1 \wedge C_2 \wedge C_{10} \wedge$$

29.

$$q + 1 < p \wedge s = 0 \wedge \phi = 0 \wedge 0 < t \wedge 0 < b^* \wedge 0 \leq c^* \wedge \pi c^* < |\arg(\omega)| \wedge |\arg(\omega)| < \pi(m + n - q + 1) \wedge C_1 \wedge C_3 \wedge C_{10} \wedge$$

30.

$$n = 0 \wedge \phi = 0 \wedge 0 < s + t \wedge 0 < m \wedge 0 < c^* \wedge b^* < 0 \wedge |\arg(\sigma)| < \pi(s + t - u + 1) \wedge C_1 \wedge C_2 \wedge C_{12} \wedge C_{14} \wedge C_{15}$$

31.

$$m = 0 \wedge \phi = 0 \wedge v < s + t \wedge 0 < n \wedge 0 < c^* \wedge b^* < 0 \wedge |\arg(\sigma)| < \pi(s + t - v + 1) \wedge C_1 \wedge C_3 \wedge C_{12} \wedge C_{14} \wedge C_{15}$$

32.

$$n = 0 \wedge \phi = 0 \wedge u = v - 1 \wedge 0 < m \wedge 0 < c^* \wedge 0 \leq b^* \wedge \pi b^* < |\arg(\sigma)| \wedge |\arg(\sigma)| < \pi(b^* + 1) \wedge C_1 \wedge C_2 \wedge C_{12} \wedge C_{14} \wedge$$

33.

$$m = 0 \wedge \phi = 0 \wedge u = v + 1 \wedge 0 < n \wedge 0 < c^* \wedge 0 \leq b^* \wedge \pi b^* < |\arg(\sigma)| \wedge |\arg(\sigma)| < \pi(b^* + 1) \wedge C_1 \wedge C_3 \wedge C_{12} \wedge C_{14} \wedge$$

34.

$$n = 0 \wedge \phi = 0 \wedge u < v - 1 \wedge 0 < m \wedge 0 < c^* \wedge 0 \leq b^* \wedge \pi b^* < |\arg(\sigma)| \wedge |\arg(\sigma)| < \pi(s + t - u + 1) \wedge C_1 \wedge C_2 \wedge C_{12} \wedge$$

35.

$$m = 0 \wedge \phi = 0 \wedge v + 1 < u \wedge 0 < n \wedge 0 < c^* \wedge 0 \leq b^* \wedge \pi b^* < |\arg(\sigma)| \wedge |\arg(\sigma)| < \pi(s + t - v + 1) \wedge C_1 \wedge C_3 \wedge C_{12} \wedge$$

36.

$$C_{17} \wedge t = 0 \wedge u < s \wedge 0 < b^* \wedge C_{10} \wedge C_1 \wedge C_2 \wedge C_3$$

37.

$$C_{17} \wedge s = 0 \wedge v < t \wedge 0 < b^* \wedge C_{10} \wedge C_1 \wedge C_2 \wedge C_3$$

38.

$$C_{16} \wedge n = 0 \wedge p < m \wedge 0 < c^* \wedge C_{12} \wedge C_1 \wedge C_2 \wedge C_3$$

39.

$$C_{16} \wedge m = 0 \wedge q < n \wedge 0 < c^* \wedge C_{12} \wedge C_1 \wedge C_2 \wedge C_3$$

The Inverse Laplace Transform of a G-function

The inverse laplace transform of a Meijer G-function can be expressed as another G-function. This is a fairly versatile method for computing this transform. However, I could not find the details in the literature, so I work them out here. In [Luke1969], section 5.6.3, there is a formula for the inverse Laplace transform of a G-function of argument bz, and convergence conditions are also given. However, we need a formula for argument bz^a for rational a.

We are asked to compute

$$f(t) = \frac{1}{2\pi i}\int_{c-i\infty}^{c+i\infty} e^{zt}G(bz^a)\mathrm{d}z,$$

for positive real t. Three questions arise:

1. When does this integral converge?
2. How can we compute the integral?
3. When is our computation valid?

How to compute the integral

We shall work formally for now. Denote by $\Delta(s)$ the product of gamma functions appearing in the definition of G, so that

$$G(z) = \frac{1}{2\pi i}\int_L \Delta(s)z^s\mathrm{d}s.$$

Thus

$$f(t) = \frac{1}{(2\pi i)^2}\int_{c-i\infty}^{c+i\infty}\int_L e^{zt}\Delta(s)b^s z^{as}\mathrm{d}s\mathrm{d}z.$$

We interchange the order of integration to get

$$f(t) = \frac{1}{2\pi i}\int_L b^s\Delta(s)\int_{c-i\infty}^{c+i\infty} e^{zt}z^{as}\frac{\mathrm{d}z}{2\pi i}\mathrm{d}s.$$

The inner integral is easily seen to be $\frac{1}{\Gamma(-as)}\frac{1}{t^{1+as}}$. (Using Cauchy's theorem and Jordan's lemma deform the contour to run from $-\infty$ to $-\infty$, encircling 0 once in the negative sense. For as real and greater than one, this contour can be pushed onto the negative real axis and the integral is recognised as a product of a sine and a gamma function. The formula is then proved using the functional equation of the gamma function, and extended to the entire domain of convergence of the original integral by appealing to analytic continuation.) Hence we find

$$f(t) = \frac{1}{t}\frac{1}{2\pi i}\int_L \Delta(s)\frac{1}{\Gamma(-as)}\left(\frac{b}{t^a}\right)^s\mathrm{d}s,$$

which is a so-called Fox H function (of argument $\frac{b}{t^a}$). For rational a, this can be expressed as a Meijer G-function using the gamma function multiplication theorem.

When this computation is valid

There are a number of obstacles in this computation. Interchange of integrals is only valid if all integrals involved are absolutely convergent. In particular the inner integral has to converge. Also, for our identification of the final integral as a Fox H / Meijer G-function to be correct, the poles of the newly obtained gamma function must be separated properly.

It is easy to check that the inner integral converges absolutely for $\Re(as) < -1$. Thus the contour L has to run left of the line $\Re(as) = -1$. Under this condition, the poles of the newly-introduced gamma function are separated properly.

It remains to observe that the Meijer G-function is an analytic, unbranched function of its parameters, and of the coefficient b. Hence so is $f(t)$. Thus the final computation remains valid as long as the initial integral converges, and if there exists a changed set of parameters where the computation is valid. If we assume w.l.o.g. that $a > 0$, then the latter condition is fulfilled if G converges along contours (2) or (3) of [Luke1969], section 5.2, i.e. either $\delta \geq \frac{a}{2}$ or $p \geq 1, p \geq q$.

When the integral exists

Using [Luke1969], section 5.10, for any given meijer G-function we can find a dominant term of the form $z^a e^{bz^c}$ (although this expression might not be the best possible, because of cancellation).

We must thus investigate

$$\lim_{T\to\infty} \int_{c-iT}^{c+iT} e^{zt} z^a e^{bz^c} \mathrm{d}z.$$

(This principal value integral is the exact statement used in the Laplace inversion theorem.) We write $z = c + i\tau$. Then $arg(z) \to \pm\frac{\pi}{2}$, and so $e^{zt} \sim e^{it\tau}$ (where $\sim$ shall always mean "asymptotically equivalent up to a positive real multiplicative constant"). Also $z^{x+iy} \sim |\tau|^x e^{iy\log|\tau|} e^{\pm xi\frac{\pi}{2}}$.

Set $\omega_\pm = be^{\pm i\Re(c)\frac{\pi}{2}}$. We have three cases:

1. $b = 0$ or $\Re(c) \leq 0$. In this case the integral converges if $\Re(a) \leq -1$.
2. $b \neq 0$, $\Im(c) = 0$, $\Re(c) > 0$. In this case the integral converges if $\Re(\omega_\pm) < 0$.
3. $b \neq 0$, $\Im(c) = 0$, $\Re(c) > 0$, $\Re(\omega_\pm) \leq 0$, and at least one of $\Re(\omega_\pm) = 0$. Here the same condition as in (1) applies.

Implemented G-Function Formulae

An important part of the algorithm is a table expressing various functions as Meijer G-functions. This is essentially a table of Mellin Transforms in disguise. The following automatically generated table shows the formulae currently implemented in SymPy. An entry "generated" means that the corresponding G-function has a variable number of parameters. This table is intended to shrink in future, when the algorithm's capabilities of deriving new formulae improve. Of course it has to grow whenever a new class of special functions is to be dealt with.

Elementary functions:

$$a = aG^{1,0}_{1,1}\left(\begin{matrix} & 1 \\ 0 & \end{matrix}\middle| z\right) + aG^{0,1}_{1,1}\left(\begin{matrix} 1 & \\ & 0 \end{matrix}\middle| z\right)$$

$$(z^q p + b)^{-a} = \frac{b^{-a} G_{1,1}^{1,1}\left(\begin{matrix}1-a \\ 0\end{matrix}\middle| \frac{z^q p}{b}\right)}{\Gamma(a)}$$

$$\frac{-b^a + (z^q p)^a}{z^q p - b} = \frac{b^{a-1} G_{2,2}^{2,2}\left(\begin{matrix}0, a \\ 0, a\end{matrix}\middle| \frac{z^q p}{b}\right) \sin(\pi a)}{\pi}$$

$$\left(a + \sqrt{z^q p + a^2}\right)^b = -\frac{a^b b G_{2,2}^{1,2}\left(\begin{matrix}\frac{b}{2}+\frac{1}{2}, \frac{b}{2}+1 \\ 0 \qquad b\end{matrix}\middle| \frac{z^q p}{a^2}\right)}{2\sqrt{\pi}}$$

$$\left(-a + \sqrt{z^q p + a^2}\right)^b = \frac{a^b b G_{2,2}^{1,2}\left(\begin{matrix}\frac{b}{2}+\frac{1}{2}, \frac{b}{2}+1 \\ b \qquad 0\end{matrix}\middle| \frac{z^q p}{a^2}\right)}{2\sqrt{\pi}}$$

$$\frac{\left(a + \sqrt{z^q p + a^2}\right)^b}{\sqrt{z^q p + a^2}} = \frac{a^{b-1} G_{2,2}^{1,2}\left(\begin{matrix}\frac{b}{2}+\frac{1}{2}, \frac{b}{2} \\ 0 \qquad b\end{matrix}\middle| \frac{z^q p}{a^2}\right)}{\sqrt{\pi}}$$

$$\frac{\left(-a + \sqrt{z^q p + a^2}\right)^b}{\sqrt{z^q p + a^2}} = \frac{a^{b-1} G_{2,2}^{1,2}\left(\begin{matrix}\frac{b}{2}+\frac{1}{2}, \frac{b}{2} \\ b \qquad 0\end{matrix}\middle| \frac{z^q p}{a^2}\right)}{\sqrt{\pi}}$$

$$\left(z^{\frac{q}{2}}\sqrt{p} + \sqrt{z^q p + a}\right)^b = -\frac{a^{\frac{b}{2}} b G_{2,2}^{2,1}\left(\begin{matrix}\frac{b}{2}+1 \qquad 1-\frac{b}{2} \\ 0, \frac{1}{2}\end{matrix}\middle| \frac{z^q p}{a}\right)}{2\sqrt{\pi}}$$

$$\left(-z^{\frac{q}{2}}\sqrt{p} + \sqrt{z^q p + a}\right)^b = \frac{a^{\frac{b}{2}} b G_{2,2}^{2,1}\left(\begin{matrix}1-\frac{b}{2} \qquad \frac{b}{2}+1 \\ 0, \frac{1}{2}\end{matrix}\middle| \frac{z^q p}{a}\right)}{2\sqrt{\pi}}$$

$$\frac{\left(z^{\frac{q}{2}}\sqrt{p} + \sqrt{z^q p + a}\right)^b}{\sqrt{z^q p + a}} = \frac{a^{\frac{b}{2}-\frac{1}{2}} G_{2,2}^{2,1}\left(\begin{matrix}\frac{b}{2}+\frac{1}{2} \qquad \frac{1}{2}-\frac{b}{2} \\ 0, \frac{1}{2}\end{matrix}\middle| \frac{z^q p}{a}\right)}{\sqrt{\pi}}$$

$$\frac{\left(-z^{\frac{q}{2}}\sqrt{p} + \sqrt{z^q p + a}\right)^b}{\sqrt{z^q p + a}} = \frac{a^{\frac{b}{2}-\frac{1}{2}} G_{2,2}^{2,1}\left(\begin{matrix}\frac{1}{2}-\frac{b}{2} \qquad \frac{b}{2}+\frac{1}{2} \\ 0, \frac{1}{2}\end{matrix}\middle| \frac{z^q p}{a}\right)}{\sqrt{\pi}}$$

Functions involving $|z^q p - b|$:

$$|z^q p - b|^{-a} = 2G_{2,2}^{1,1}\left(\begin{matrix}1-a \qquad \frac{1}{2}-\frac{a}{2} \\ 0 \qquad \frac{1}{2}-\frac{a}{2}\end{matrix}\middle| \frac{z^q p}{b}\right) \sin\left(\frac{\pi a}{2}\right) |b|^{-a} \Gamma(1-a), \text{ if } \operatorname{re}(a) < 1$$

Functions involving $\operatorname{Chi}(z^q p)$:

$$\operatorname{Chi}(z^q p) = -\frac{\pi^{\frac{3}{2}} G_{2,4}^{2,0}\left(\begin{matrix} \qquad \frac{1}{2}, 1 \\ 0, 0 \qquad \frac{1}{2}, \frac{1}{2}\end{matrix}\middle| \frac{z^{2q} p^2}{4}\right)}{2}$$

Functions involving $\operatorname{Ci}(z^q p)$:

$$\operatorname{Ci}(z^q p) = -\frac{\sqrt{\pi} G_{1,3}^{2,0}\left(\begin{matrix} \qquad 1 \\ 0, 0 \qquad \frac{1}{2}\end{matrix}\middle| \frac{z^{2q} p^2}{4}\right)}{2}$$

Functions involving $\operatorname{Ei}(z^q p)$:

$$\operatorname{Ei}(z^q p) = -i\pi G_{1,1}^{1,0}\left(\begin{matrix}0 \\ \end{matrix}\middle| \begin{matrix}1 \\ z\end{matrix}\right) - G_{1,2}^{2,0}\left(\begin{matrix}0, 0 \\ \end{matrix}\middle| \begin{matrix}1 \\ z^q p e^{i\pi}\end{matrix}\right) - i\pi G_{1,1}^{0,1}\left(\begin{matrix}1 \\ \end{matrix}\middle| \begin{matrix}0 \\ z\end{matrix}\right)$$

Functions involving $\theta\left(z^q p - b\right)$:

$$\left(z^q p - b\right)^{a-1} \theta\left(z^q p - b\right) = b^{a-1} G_{1,1}^{0,1}\left(\begin{matrix} a \\ 0 \end{matrix}\middle| \frac{z^q p}{b}\right) \Gamma\left(a\right), \text{ if } b > 0$$

$$\left(-z^q p + b\right)^{a-1} \theta\left(-z^q p + b\right) = b^{a-1} G_{1,1}^{1,0}\left(\begin{matrix} \\ 0 \end{matrix}\middle| \begin{matrix} a \\ \end{matrix} \frac{z^q p}{b}\right) \Gamma\left(a\right), \text{ if } b > 0$$

$$\left(z^q p - b\right)^{a-1} \theta\left(z - \left(\frac{b}{p}\right)^{\frac{1}{q}}\right) = b^{a-1} G_{1,1}^{0,1}\left(\begin{matrix} a \\ 0 \end{matrix}\middle| \frac{z^q p}{b}\right) \Gamma\left(a\right), \text{ if } b > 0$$

$$\left(-z^q p + b\right)^{a-1} \theta\left(-z + \left(\frac{b}{p}\right)^{\frac{1}{q}}\right) = b^{a-1} G_{1,1}^{1,0}\left(\begin{matrix} \\ 0 \end{matrix}\middle| \begin{matrix} a \\ \end{matrix} \frac{z^q p}{b}\right) \Gamma\left(a\right), \text{ if } b > 0$$

Functions involving $\theta\left(-z^q p + 1\right)$, $\log\left(z^q p\right)$:

$$\log\left(z^q p\right)^n \theta\left(-z^q p + 1\right) = \text{generated}$$

$$\log\left(z^q p\right)^n \theta\left(z^q p - 1\right) = \text{generated}$$

Functions involving $\operatorname{Shi}\left(z^q p\right)$:

$$\operatorname{Shi}\left(z^q p\right) = \frac{z^q \sqrt{\pi} p G_{1,3}^{1,1}\left(\begin{matrix} \frac{1}{2} \\ 0 \end{matrix}\middle| \begin{matrix} \\ -\frac{1}{2}, -\frac{1}{2} \end{matrix} \frac{z^{2q} p^2 e^{i\pi}}{4}\right)}{4}$$

Functions involving $\operatorname{Si}\left(z^q p\right)$:

$$\operatorname{Si}\left(z^q p\right) = \frac{\sqrt{\pi} G_{1,3}^{1,1}\left(\begin{matrix} 1 \\ \frac{1}{2} \end{matrix}\middle| \begin{matrix} \\ 0, 0 \end{matrix} \frac{z^{2q} p^2}{4}\right)}{2}$$

Functions involving $I_a\left(z^q p\right)$:

$$I_a\left(z^q p\right) = \pi G_{1,3}^{1,0}\left(\begin{matrix} \frac{a}{2} \\ -\frac{a}{2}, \frac{a}{2} + \frac{1}{2} \end{matrix}\middle| \begin{matrix} \frac{a}{2} + \frac{1}{2} \\ \end{matrix} \frac{z^{2q} p^2}{4}\right)$$

Functions involving $J_a\left(z^q p\right)$:

$$J_a\left(z^q p\right) = G_{0,2}^{1,0}\left(\begin{matrix} \frac{a}{2} \\ -\frac{a}{2} \end{matrix}\middle| \frac{z^{2q} p^2}{4}\right)$$

Functions involving $K_a\left(z^q p\right)$:

$$K_a\left(z^q p\right) = \frac{G_{0,2}^{2,0}\left(\begin{matrix} \\ \frac{a}{2}, -\frac{a}{2} \end{matrix}\middle| \frac{z^{2q} p^2}{4}\right)}{2}$$

Functions involving $Y_a\left(z^q p\right)$:

$$Y_a\left(z^q p\right) = G_{1,3}^{2,0}\left(\begin{matrix} -\frac{a}{2} - \frac{1}{2} \\ \frac{a}{2}, -\frac{a}{2} \end{matrix}\middle| \begin{matrix} -\frac{a}{2} - \frac{1}{2} \\ \end{matrix} \frac{z^{2q} p^2}{4}\right)$$

Functions involving $\cos\left(z^q p\right)$:

$$\cos\left(z^q p\right) = \sqrt{\pi} G_{0,2}^{1,0}\left(\begin{matrix} 0 \\ \frac{1}{2} \end{matrix}\middle| \frac{z^{2q} p^2}{4}\right)$$

Functions involving $\cosh(z^q p)$:

$$\cosh(z^q p) = \pi^{\frac{3}{2}} G_{1,3}^{1,0}\left(\begin{matrix} & \frac{1}{2} \\ 0 & \frac{1}{2}, \frac{1}{2} \end{matrix}\middle| \frac{z^{2q}p^2}{4}\right)$$

Functions involving $E(z^q p)$:

$$E(z^q p) = -\frac{G_{2,2}^{1,2}\left(\begin{matrix} \frac{1}{2}, \frac{3}{2} & \\ 0 & 0 \end{matrix}\middle| -z^q p\right)}{4}$$

Functions involving $K(z^q p)$:

$$K(z^q p) = \frac{G_{2,2}^{1,2}\left(\begin{matrix} \frac{1}{2}, \frac{1}{2} & \\ 0 & 0 \end{matrix}\middle| -z^q p\right)}{2}$$

Functions involving $\operatorname{erf}(z^q p)$:

$$\operatorname{erf}(z^q p) = \frac{G_{1,2}^{1,1}\left(\begin{matrix} 1 & \\ \frac{1}{2} & 0 \end{matrix}\middle| z^{2q}p^2\right)}{\sqrt{\pi}}$$

Functions involving $\operatorname{erfc}(z^q p)$:

$$\operatorname{erfc}(z^q p) = \frac{G_{1,2}^{2,0}\left(\begin{matrix} & 1 \\ 0, \frac{1}{2} & \end{matrix}\middle| z^{2q}p^2\right)}{\sqrt{\pi}}$$

Functions involving $\operatorname{erfi}(z^q p)$:

$$\operatorname{erfi}(z^q p) = \frac{z^q p G_{1,2}^{1,1}\left(\begin{matrix} \frac{1}{2} & \\ 0 & -\frac{1}{2} \end{matrix}\middle| -z^{2q}p^2\right)}{\sqrt{\pi}}$$

Functions involving $e^{z^q p e^{i\pi}}$:

$$e^{z^q p e^{i\pi}} = G_{0,1}^{1,0}\left(\begin{matrix} & \\ 0 & \end{matrix}\middle| z^q p\right)$$

Functions involving $\mathrm{E}_a(z^q p)$:

$$\mathrm{E}_a(z^q p) = G_{1,2}^{2,0}\left(\begin{matrix} & a \\ a-1, 0 & \end{matrix}\middle| z^q p\right)$$

Functions involving $C(z^q p)$:

$$C(z^q p) = \frac{G_{1,3}^{1,1}\left(\begin{matrix} 1 & \\ \frac{1}{4} & 0, \frac{3}{4} \end{matrix}\middle| \frac{z^{4q}\pi^2 p^4}{16}\right)}{2}$$

Functions involving $S(z^q p)$:

$$S(z^q p) = \frac{G_{1,3}^{1,1}\left(\begin{matrix} 1 & \\ \frac{3}{4} & 0, \frac{1}{4} \end{matrix}\middle| \frac{z^{4q}\pi^2 p^4}{16}\right)}{2}$$

Functions involving $\log(z^q p)$:

$$\log(z^q p)^n = \text{generated}$$

$$\log\left(z^q p + a\right) = G_{1,1}^{1,0}\left(\begin{matrix} & 1 \\ 0 & \end{matrix}\middle| z\right)\log\left(a\right) + G_{1,1}^{0,1}\left(\begin{matrix} 1 & \\ & 0 \end{matrix}\middle| z\right)\log\left(a\right) + G_{2,2}^{1,2}\left(\begin{matrix} 1, 1 & \\ 1 & 0 \end{matrix}\middle| \frac{z^q p}{a}\right)$$

$$\log\left(\left|z^q p - a\right|\right) = G_{1,1}^{1,0}\left(\begin{matrix} & 1 \\ 0 & \end{matrix}\middle| z\right)\log\left(\left|a\right|\right) + G_{1,1}^{0,1}\left(\begin{matrix} 1 & \\ & 0 \end{matrix}\middle| z\right)\log\left(\left|a\right|\right) + \pi G_{3,3}^{1,2}\left(\begin{matrix} 1, 1 & \frac{1}{2} \\ 1 & 0, \frac{1}{2} \end{matrix}\middle| \frac{z^q p}{a}\right)$$

Functions involving $\sin\left(z^q p\right)$:

$$\sin\left(z^q p\right) = \sqrt{\pi} G_{0,2}^{1,0}\left(\begin{matrix} \frac{1}{2} & \\ & 0 \end{matrix}\middle| \frac{z^{2q} p^2}{4}\right)$$

Functions involving $\operatorname{sinc}\left(z^q p\right)$:

$$\operatorname{sinc}\left(z^q p\right) = \frac{\sqrt{\pi} G_{0,2}^{1,0}\left(\begin{matrix} 0 & -\frac{1}{2} \end{matrix}\middle| \frac{z^{2q} p^2}{4}\right)}{2}$$

Functions involving $\sinh\left(z^q p\right)$:

$$\sinh\left(z^q p\right) = \pi^{\frac{3}{2}} G_{1,3}^{1,0}\left(\begin{matrix} & 1 \\ \frac{1}{2} & 1, 0 \end{matrix}\middle| \frac{z^{2q} p^2}{4}\right)$$

Internal API Reference

Integrate functions by rewriting them as Meijer G-functions.

There are three user-visible functions that can be used by other parts of the sympy library to solve various integration problems:

- meijerint_indefinite
- meijerint_definite
- meijerint_inversion

They can be used to compute, respectively, indefinite integrals, definite integrals over intervals of the real line, and inverse laplace-type integrals (from c-I*oo to c+I*oo). See the respective docstrings for details.

The main references for this are:

[L] Luke, Y. L. (1969), The Special Functions and Their Approximations,
Volume 1

[R] Kelly B. Roach. Meijer G Function Representations.
In: Proceedings of the 1997 International Symposium on Symbolic and Algebraic Computation, pages 205-211, New York, 1997. ACM.

[P] A. P. Prudnikov, Yu. A. Brychkov and O. I. Marichev (1990).
Integrals and Series: More Special Functions, Vol. 3,. Gordon and Breach Science Publisher

exception sympy.integrals.meijerint.**_CoeffExpValueError**

Exception raised by _get_coeff_exp, for internal use only.

sympy.integrals.meijerint.**_check_antecedents**(*g1*, *g2*, *x*)

Return a condition under which the integral theorem applies.

sympy.integrals.meijerint.**_check_antecedents_1**(*g*, *x*, *helper=False*)

Return a condition under which the mellin transform of g exists. Any power of x has already been absorbed into the G function, so this is just $\int_0^\infty g\,dx$.

See [L, section 5.6.1]. (Note that s=1.)

If `helper` is True, only check if the MT exists at infinity, i.e. if $\int_1^\infty g\,dx$ exists.

sympy.integrals.meijerint.**_check_antecedents_inversion**(*g*, *x*)

Check antecedents for the laplace inversion integral.

sympy.integrals.meijerint.**_condsimp**(*cond*, *first=True*)

Do naive simplifications on `cond`.

Explanation

Note that this routine is completely ad-hoc, simplification rules being added as need arises rather than following any logical pattern.

Examples

```
>>> from sympy.integrals.meijerint import _condsimp as simp
>>> from sympy import Or, Eq
>>> from sympy.abc import x, y
>>> simp(Or(x < y, Eq(x, y)))
x <= y
```

sympy.integrals.meijerint.**_create_lookup_table**(*table*)

Add formulae for the function -> meijerg lookup table.

sympy.integrals.meijerint.**_dummy**(*name*, *token*, *expr*, ***kwargs*)

Return a dummy. This will return the same dummy if the same token+name is requested more than once, and it is not already in expr. This is for being cache-friendly.

sympy.integrals.meijerint.**_dummy_**(*name*, *token*, ***kwargs*)

Return a dummy associated to name and token. Same effect as declaring it globally.

sympy.integrals.meijerint.**_eval_cond**(*cond*)

Re-evaluate the conditions.

sympy.integrals.meijerint.**_exponents**(*expr*, *x*)

Find the exponents of `x` (not including zero) in `expr`.

Examples

```
>>> from sympy.integrals.meijerint import _exponents
>>> from sympy.abc import x, y
>>> from sympy import sin
>>> _exponents(x, x)
{1}
>>> _exponents(x**2, x)
```

(continues on next page)

(continued from previous page)

```
{2}
>>> _exponents(x**2 + x, x)
{1, 2}
>>> _exponents(x**3*sin(x + x**y) + 1/x, x)
{-1, 1, 3, y}
```

sympy.integrals.meijerint.**_find_splitting_points**(*expr*, *x*)

Find numbers a such that a linear substitution x -> x + a would (hopefully) simplify expr.

Examples

```
>>> from sympy.integrals.meijerint import _find_splitting_points as fsp
>>> from sympy import sin
>>> from sympy.abc import x
>>> fsp(x, x)
{0}
>>> fsp((x-1)**3, x)
{1}
>>> fsp(sin(x+3)*x, x)
{-3, 0}
```

sympy.integrals.meijerint.**_flip_g**(*g*)

Turn the G function into one of inverse argument (i.e. G(1/x) -> G'(x))

sympy.integrals.meijerint.**_functions**(*expr*, *x*)

Find the types of functions in expr, to estimate the complexity.

sympy.integrals.meijerint.**_get_coeff_exp**(*expr*, *x*)

When expr is known to be of the form c*x**b, with c and/or b possibly 1, return c, b.

Examples

```
>>> from sympy.abc import x, a, b
>>> from sympy.integrals.meijerint import _get_coeff_exp
>>> _get_coeff_exp(a*x**b, x)
(a, b)
>>> _get_coeff_exp(x, x)
(1, 1)
>>> _get_coeff_exp(2*x, x)
(2, 1)
>>> _get_coeff_exp(x**3, x)
(1, 3)
```

sympy.integrals.meijerint.**_guess_expansion**(*f*, *x*)

Try to guess sensible rewritings for integrand f(x).

sympy.integrals.meijerint.**_inflate_fox_h**(*g*, *a*)

Let d denote the integrand in the definition of the G function g. Consider the function H which is defined in the same way, but with integrand d/Gamma(a*s) (contour conventions as usual).

If a is rational, the function H can be written as C*G, for a constant C and a G-function G.

This function returns C, G.

sympy.integrals.meijerint.**_inflate_g**(*g*, *n*)

Return C, h such that h is a G function of argument z**n and g = C*h.

sympy.integrals.meijerint.**_int0oo**(*g1*, *g2*, *x*)

Express integral from zero to infinity g1*g2 using a G function, assuming the necessary conditions are fulfilled.

Examples

```
>>> from sympy.integrals.meijerint import _int0oo
>>> from sympy.abc import s, t, m
>>> from sympy import meijerg, S
>>> g1 = meijerg([], [], [-S(1)/2, 0], [], s**2*t/4)
>>> g2 = meijerg([], [], [m/2], [-m/2], t/4)
>>> _int0oo(g1, g2, t)
4*meijerg(((1/2, 0), ()), ((m/2,), (-m/2,)), s**(-2))/s**2
```

sympy.integrals.meijerint.**_int0oo_1**(*g*, *x*)

Evaluate $\int_0^\infty g\,dx$ using G functions, assuming the necessary conditions are fulfilled.

Examples

```
>>> from sympy.abc import a, b, c, d, x, y
>>> from sympy import meijerg
>>> from sympy.integrals.meijerint import _int0oo_1
>>> _int0oo_1(meijerg([a], [b], [c], [d], x*y), x)
gamma(-a)*gamma(c + 1)/(y*gamma(-d)*gamma(b + 1))
```

sympy.integrals.meijerint.**_int_inversion**(*g*, *x*, *t*)

Compute the laplace inversion integral, assuming the formula applies.

sympy.integrals.meijerint.**_is_analytic**(*f*, *x*)

Check if f(x), when expressed using G functions on the positive reals, will in fact agree with the G functions almost everywhere

sympy.integrals.meijerint.**_meijerint_definite_2**(*f*, *x*)

Try to integrate f dx from zero to infinity.

The body of this function computes various 'simplifications' f1, f2, ... of f (e.g. by calling expand_mul(), trigexpand() - see _guess_expansion) and calls _meijerint_definite_3 with each of these in succession. If _meijerint_definite_3 succeeds with any of the simplified functions, returns this result.

sympy.integrals.meijerint.**_meijerint_definite_3**(*f*, *x*)

Try to integrate f dx from zero to infinity.

This function calls _meijerint_definite_4 to try to compute the integral. If this fails, it tries using linearity.

sympy.integrals.meijerint.**_meijerint_definite_4**(*f*, *x*, *only_double=False*)

Try to integrate f dx from zero to infinity.

Explanation

This function tries to apply the integration theorems found in literature, i.e. it tries to rewrite f as either one or a product of two G-functions.

The parameter `only_double` is used internally in the recursive algorithm to disable trying to rewrite f as a single G-function.

sympy.integrals.meijerint.**_meijerint_indefinite_1**(*f*, *x*)

Helper that does not attempt any substitution.

sympy.integrals.meijerint.**_mul_args**(*f*)

Return a list `L` such that `Mul(*L) == f`.

If `f` is not a `Mul` or `Pow`, `L=[f]`. If `f=g**n` for an integer `n`, `L=[g]*n`. If `f` is a `Mul`, `L` comes from applying `_mul_args` to all factors of `f`.

sympy.integrals.meijerint.**_mul_as_two_parts**(*f*)

Find all the ways to split `f` into a product of two terms. Return None on failure.

Explanation

Although the order is canonical from multiset_partitions, this is not necessarily the best order to process the terms. For example, if the case of len(gs) == 2 is removed and multiset is allowed to sort the terms, some tests fail.

Examples

```
>>> from sympy.integrals.meijerint import _mul_as_two_parts
>>> from sympy import sin, exp, ordered
>>> from sympy.abc import x
>>> list(ordered(_mul_as_two_parts(x*sin(x)*exp(x))))
[(x, exp(x)*sin(x)), (x*exp(x), sin(x)), (x*sin(x), exp(x))]
```

sympy.integrals.meijerint.**_my_principal_branch**(*expr*, *period*, *full_pb=False*)

Bring expr nearer to its principal branch by removing superfluous factors. This function does *not* guarantee to yield the principal branch, to avoid introducing opaque principal_branch() objects, unless full_pb=True.

sympy.integrals.meijerint.**_mytype**(*f*, *x*)

Create a hashable entity describing the type of f.

sympy.integrals.meijerint.**_rewrite1**(*f*, *x*, *recursive=True*)

Try to rewrite `f` using a (sum of) single G functions with argument a*x**b. Return fac, po, g such that f = fac*po*g, fac is independent of `x`. and po = x**s. Here g is a result from _rewrite_single. Return None on failure.

sympy.integrals.meijerint.**_rewrite2**(*f*, *x*)

Try to rewrite `f` as a product of two G functions of arguments a*x**b. Return fac, po, g1, g2 such that f = fac*po*g1*g2, where fac is independent of x and po is x**s. Here g1 and g2 are results of _rewrite_single. Returns None on failure.

sympy.integrals.meijerint.**_rewrite_inversion**(*fac*, *po*, *g*, *x*)

Absorb `po` == x**s into g.

sympy.integrals.meijerint.**_rewrite_saxena**(*fac*, *po*, *g1*, *g2*, *x*, *full_pb=False*)

Rewrite the integral `fac*po*g1*g2` from 0 to oo in terms of G functions with argument `c*x`.

Explanation

Return C, f1, f2 such that integral C f1 f2 from 0 to infinity equals integral fac `po`, `g1`, `g2` from 0 to infinity.

Examples

```
>>> from sympy.integrals.meijerint import _rewrite_saxena
>>> from sympy.abc import s, t, m
>>> from sympy import meijerg
>>> g1 = meijerg([], [], [0], [], s*t)
>>> g2 = meijerg([], [], [m/2], [-m/2], t**2/4)
>>> r = _rewrite_saxena(1, t**0, g1, g2, t)
>>> r[0]
s/(4*sqrt(pi))
>>> r[1]
meijerg(((), ()), ((-1/2, 0), ()), s**2*t/4)
>>> r[2]
meijerg(((), ()), ((m/2,), (-m/2,)), t/4)
```

sympy.integrals.meijerint.**_rewrite_saxena_1**(*fac*, *po*, *g*, *x*)

Rewrite the integral fac*po*g dx, from zero to infinity, as integral fac*G, where G has argument a*x. Note po=x**s. Return fac, G.

sympy.integrals.meijerint.**_rewrite_single**(*f*, *x*, *recursive=True*)

Try to rewrite f as a sum of single G functions of the form C*x**s*G(a*x**b), where b is a rational number and C is independent of x. We guarantee that result.argument.as_coeff_mul(x) returns (a, (x**b,)) or (a, ()). Returns a list of tuples (C, s, G) and a condition cond. Returns None on failure.

sympy.integrals.meijerint.**_split_mul**(*f*, *x*)

Split expression `f` into fac, po, g, where fac is a constant factor, po = x**s for some s independent of s, and g is “the rest”.

Examples

```
>>> from sympy.integrals.meijerint import _split_mul
>>> from sympy import sin
>>> from sympy.abc import s, x
>>> _split_mul((3*x)**s*sin(x**2)*x, x)
(3**s, x*x**s, sin(x**2))
```

sympy.integrals.meijerint.**meijerint_definite**(*f*, *x*, *a*, *b*)

Integrate `f` over the interval [`a`, `b`], by rewriting it as a product of two G functions, or as a single G function.

Return res, cond, where cond are convergence conditions.

Examples

```
>>> from sympy.integrals.meijerint import meijerint_definite
>>> from sympy import exp, oo
>>> from sympy.abc import x
>>> meijerint_definite(exp(-x**2), x, -oo, oo)
(sqrt(pi), True)
```

This function is implemented as a succession of functions meijerint_definite, _meijerint_definite_2, _meijerint_definite_3, _meijerint_definite_4. Each function in the list calls the next one (presumably) several times. This means that calling meijerint_definite can be very costly.

sympy.integrals.meijerint.**meijerint_indefinite**(*f*, *x*)

Compute an indefinite integral of `f` by rewriting it as a G function.

Examples

```
>>> from sympy.integrals.meijerint import meijerint_indefinite
>>> from sympy import sin
>>> from sympy.abc import x
>>> meijerint_indefinite(sin(x), x)
-cos(x)
```

sympy.integrals.meijerint.**meijerint_inversion**(*f*, *x*, *t*)

Compute the inverse laplace transform $\int_{c+i\infty}^{c-i\infty} f(x)e^{tx}\,dx$, for real c larger than the real part of all singularities of `f`.

Note that `t` is always assumed real and positive.

Return None if the integral does not exist or could not be evaluated.

Examples

```
>>> from sympy.abc import x, t
>>> from sympy.integrals.meijerint import meijerint_inversion
>>> meijerint_inversion(1/x, x, t)
Heaviside(t)
```

Integrals

The `integrals` module in SymPy implements methods to calculate definite and indefinite integrals of expressions.

Principal method in this module is *integrate()* (page 598)

- `integrate(f, x)` returns the indefinite integral $\int f\,dx$
- `integrate(f, (x, a, b))` returns the definite integral $\int_a^b f\,dx$

Examples

SymPy can integrate a vast array of functions. It can integrate polynomial functions:

```
>>> from sympy import *
>>> init_printing(use_unicode=False, wrap_line=False)
>>> x = Symbol('x')
>>> integrate(x**2 + x + 1, x)
 3    2
x    x
-- + -- + x
3    2
```

Rational functions:

```
>>> integrate(x/(x**2+2*x+1), x)
             1
log(x + 1) + -----
             x + 1
```

Exponential-polynomial functions. These multiplicative combinations of polynomials and the functions `exp`, `cos` and `sin` can be integrated by hand using repeated integration by parts, which is an extremely tedious process. Happily, SymPy will deal with these integrals.

```
>>> integrate(x**2 * exp(x) * cos(x), x)
 2  x           2  x                          x           x
x *e *sin(x)   x *e *cos(x)      x          e *sin(x)   e *cos(x)
------------ + ------------ - x*e *sin(x) + --------- - ---------
     2              2                         2           2
```

even a few nonelementary integrals (in particular, some integrals involving the error function) can be evaluated:

```
>>> integrate(exp(-x**2)*erf(x), x)
  ____      2
\/ pi *erf (x)
--------------
      4
```

Integral Transforms

SymPy has special support for definite integrals, and integral transforms.

sympy.integrals.transforms.**mellin_transform**(*f*, *x*, *s*, ***hints*)

Compute the Mellin transform $F(s)$ of $f(x)$,

$$F(s) = \int_0^\infty x^{s-1} f(x) \mathrm{d}x.$$

For all "sensible" functions, this converges absolutely in a strip

$a < \mathrm{Re}(s) < b$.

Explanation

The Mellin transform is related via change of variables to the Fourier transform, and also to the (bilateral) Laplace transform.

This function returns `(F, (a, b), cond)` where `F` is the Mellin transform of `f`, `(a, b)` is the fundamental strip (as above), and `cond` are auxiliary convergence conditions.

If the integral cannot be computed in closed form, this function returns an unevaluated *MellinTransform* (page 576) object.

For a description of possible hints, refer to the docstring of *sympy.integrals.transforms.IntegralTransform.doit()* (page 586). If `noconds=False`, then only F will be returned (i.e. not `cond`, and also not the strip `(a, b)`).

Examples

```
>>> from sympy import mellin_transform, exp
>>> from sympy.abc import x, s
>>> mellin_transform(exp(-x), x, s)
(gamma(s), (0, oo), True)
```

See also:

inverse_mellin_transform (page 577), *laplace_transform* (page 577), *fourier_transform* (page 579), *hankel_transform* (page 584), *inverse_hankel_transform* (page 585)

class sympy.integrals.transforms.**MellinTransform**(**args*)

Class representing unevaluated Mellin transforms.

For usage of this class, see the *IntegralTransform* (page 586) docstring.

For how to compute Mellin transforms, see the *mellin_transform()* (page 576) docstring.

sympy.integrals.transforms.**inverse_mellin_transform**(*F*, *s*, *x*, *strip*, ***hints*)

Compute the inverse Mellin transform of $F(s)$ over the fundamental strip given by `strip=(a, b)`.

Explanation

This can be defined as

$$f(x) = \frac{1}{2\pi i}\int_{c-i\infty}^{c+i\infty} x^{-s}F(s)\mathrm{d}s,$$

for any c in the fundamental strip. Under certain regularity conditions on F and/or f, this recovers f from its Mellin transform F (and vice versa), for positive real x.

One of a or b may be passed as `None`; a suitable c will be inferred.

If the integral cannot be computed in closed form, this function returns an unevaluated *InverseMellinTransform* (page 577) object.

Note that this function will assume x to be positive and real, regardless of the SymPy assumptions!

For a description of possible hints, refer to the docstring of *sympy.integrals.transforms.IntegralTransform.doit()* (page 586).

Examples

```
>>> from sympy import inverse_mellin_transform, oo, gamma
>>> from sympy.abc import x, s
>>> inverse_mellin_transform(gamma(s), s, x, (0, oo))
exp(-x)
```

The fundamental strip matters:

```
>>> f = 1/(s**2 - 1)
>>> inverse_mellin_transform(f, s, x, (-oo, -1))
x*(1 - 1/x**2)*Heaviside(x - 1)/2
>>> inverse_mellin_transform(f, s, x, (-1, 1))
-x*Heaviside(1 - x)/2 - Heaviside(x - 1)/(2*x)
>>> inverse_mellin_transform(f, s, x, (1, oo))
(1/2 - x**2/2)*Heaviside(1 - x)/x
```

See also:

mellin_transform (page 576), *hankel_transform* (page 584), *inverse_hankel_transform* (page 585)

class sympy.integrals.transforms.**InverseMellinTransform**(**args*)

Class representing unevaluated inverse Mellin transforms.

For usage of this class, see the *IntegralTransform* (page 586) docstring.

For how to compute inverse Mellin transforms, see the *inverse_mellin_transform()* (page 577) docstring.

sympy.integrals.transforms.**laplace_transform**(*f*, *t*, *s*, *legacy_matrix=True*, ***hints*)

Compute the Laplace Transform $F(s)$ of $f(t)$,

$$F(s) = \int_{0^-}^{\infty} e^{-st} f(t) \mathrm{d}t.$$

Explanation

For all sensible functions, this converges absolutely in a half-plane

$$a < \mathrm{Re}(s)$$

This function returns `(F, a, cond)` where `F` is the Laplace transform of `f`, a is the half-plane of convergence, and $cond$ are auxiliary convergence conditions.

The implementation is rule-based, and if you are interested in which rules are applied, and whether integration is attemped, you can switch debug information on by setting `sympy.SYMPY_DEBUG=True`.

The lower bound is $0-$, meaning that this bound should be approached from the lower side. This is only necessary if distributions are involved. At present, it is only done if $f(t)$ contains `DiracDelta`, in which case the Laplace transform is computed implicitly as

$$F(s) = \lim_{\tau \to 0^-} \int_{\tau}^{\infty} e^{-st} f(t) \mathrm{d}t$$

by applying rules.

If the integral cannot be fully computed in closed form, this function returns an unevaluated *LaplaceTransform* (page 579) object.

For a description of possible hints, refer to the docstring of *sympy.integrals.transforms.IntegralTransform.doit()* (page 586). If `noconds=True`, only F will be returned (i.e. not `cond`, and also not the plane `a`).

Deprecated since version 1.9: Legacy behavior for matrices where `laplace_transform` with `noconds=False` (the default) returns a Matrix whose elements are tuples. The behavior of `laplace_transform` for matrices will change in a future release of SymPy to return a tuple of the transformed Matrix and the convergence conditions for the matrix as a whole. Use `legacy_matrix=False` to enable the new behavior.

Examples

```
>>> from sympy import DiracDelta, exp, laplace_transform
>>> from sympy.abc import t, s, a
>>> laplace_transform(t**4, t, s)
(24/s**5, 0, True)
>>> laplace_transform(t**a, t, s)
(gamma(a + 1)/(s*s**a), 0, re(a) > -1)
>>> laplace_transform(DiracDelta(t)-a*exp(-a*t),t,s)
(s/(a + s), Max(0, -a), True)
```

See also:

inverse_laplace_transform (page 579), *mellin_transform* (page 576), *fourier_transform* (page 579), *hankel_transform* (page 584), *inverse_hankel_transform* (page 585)

class sympy.integrals.transforms.**LaplaceTransform**(**args*)

Class representing unevaluated Laplace transforms.

For usage of this class, see the *IntegralTransform* (page 586) docstring.

For how to compute Laplace transforms, see the *laplace_transform()* (page 577) docstring.

sympy.integrals.transforms.**inverse_laplace_transform**(*F*, *s*, *t*, *plane=None*, ***hints*)

Compute the inverse Laplace transform of $F(s)$, defined as

$$f(t) = \frac{1}{2\pi i}\int_{c-i\infty}^{c+i\infty} e^{st}F(s)\mathrm{d}s,$$

for c so large that $F(s)$ has no singularites in the half-plane $\mathrm{Re}(s) > c - \epsilon$.

Explanation

The plane can be specified by argument `plane`, but will be inferred if passed as None.

Under certain regularity conditions, this recovers $f(t)$ from its Laplace Transform $F(s)$, for non-negative t, and vice versa.

If the integral cannot be computed in closed form, this function returns an unevaluated *InverseLaplaceTransform* (page 579) object.

Note that this function will always assume t to be real, regardless of the SymPy assumption on t.

For a description of possible hints, refer to the docstring of *sympy.integrals.transforms.IntegralTransform.doit()* (page 586).

Examples

```
>>> from sympy import inverse_laplace_transform, exp, Symbol
>>> from sympy.abc import s, t
>>> a = Symbol('a', positive=True)
>>> inverse_laplace_transform(exp(-a*s)/s, s, t)
Heaviside(-a + t)
```

See also:

laplace_transform (page 577), *_fast_inverse_laplace* (page 579), *hankel_transform* (page 584), *inverse_hankel_transform* (page 585)

class sympy.integrals.transforms.**InverseLaplaceTransform**(**args*)

Class representing unevaluated inverse Laplace transforms.

For usage of this class, see the *IntegralTransform* (page 586) docstring.

For how to compute inverse Laplace transforms, see the *inverse_laplace_transform()* (page 579) docstring.

sympy.integrals.transforms.**_fast_inverse_laplace**(*e*, *s*, *t*)

Fast inverse Laplace transform of rational function including RootSum

sympy.integrals.transforms.**fourier_transform**(*f*, *x*, *k*, ***hints*)

Compute the unitary, ordinary-frequency Fourier transform of `f`, defined as

$$F(k) = \int_{-\infty}^{\infty} f(x) e^{-2\pi i x k} \mathrm{d}x.$$

Explanation

If the transform cannot be computed in closed form, this function returns an unevaluated *FourierTransform* (page 580) object.

For other Fourier transform conventions, see the function *sympy.integrals.transforms._fourier_transform()* (page 580).

For a description of possible hints, refer to the docstring of *sympy.integrals.transforms.IntegralTransform.doit()* (page 586). Note that for this transform, by default `noconds=True`.

Examples

```
>>> from sympy import fourier_transform, exp
>>> from sympy.abc import x, k
>>> fourier_transform(exp(-x**2), x, k)
sqrt(pi)*exp(-pi**2*k**2)
>>> fourier_transform(exp(-x**2), x, k, noconds=False)
(sqrt(pi)*exp(-pi**2*k**2), True)
```

See also:

inverse_fourier_transform (page 580), *sine_transform* (page 581), *inverse_sine_transform* (page 582), *cosine_transform* (page 583), *inverse_cosine_transform* (page 583), *hankel_transform* (page 584), *inverse_hankel_transform* (page 585), *mellin_transform* (page 576), *laplace_transform* (page 577)

sympy.integrals.transforms.**_fourier_transform**(*f*, *x*, *k*, *a*, *b*, *name*, *simplify=True*)

Compute a general Fourier-type transform

$$F(k) = a \int_{-\infty}^{\infty} e^{bixk} f(x)\, dx.$$

For suitable choice of *a* and *b*, this reduces to the standard Fourier and inverse Fourier transforms.

class sympy.integrals.transforms.**FourierTransform**(**args*)

Class representing unevaluated Fourier transforms.

For usage of this class, see the *IntegralTransform* (page 586) docstring.

For how to compute Fourier transforms, see the *fourier_transform()* (page 579) docstring.

sympy.integrals.transforms.**inverse_fourier_transform**(*F*, *k*, *x*, ***hints*)

Compute the unitary, ordinary-frequency inverse Fourier transform of F, defined as

$$f(x) = \int_{-\infty}^{\infty} F(k) e^{2\pi i x k} \mathrm{d}k.$$

Explanation

If the transform cannot be computed in closed form, this function returns an unevaluated *InverseFourierTransform* (page 581) object.

For other Fourier transform conventions, see the function *sympy.integrals.transforms._fourier_transform()* (page 580).

For a description of possible hints, refer to the docstring of *sympy.integrals.transforms.IntegralTransform.doit()* (page 586). Note that for this transform, by default `noconds=True`.

Examples

```
>>> from sympy import inverse_fourier_transform, exp, sqrt, pi
>>> from sympy.abc import x, k
>>> inverse_fourier_transform(sqrt(pi)*exp(-(pi*k)**2), k, x)
exp(-x**2)
>>> inverse_fourier_transform(sqrt(pi)*exp(-(pi*k)**2), k, x,␣
↪noconds=False)
(exp(-x**2), True)
```

See also:

fourier_transform (page 579), *sine_transform* (page 581), *inverse_sine_transform* (page 582), *cosine_transform* (page 583), *inverse_cosine_transform* (page 583), *hankel_transform* (page 584), *inverse_hankel_transform* (page 585), *mellin_transform* (page 576), *laplace_transform* (page 577)

class sympy.integrals.transforms.**InverseFourierTransform**(**args*)

Class representing unevaluated inverse Fourier transforms.

For usage of this class, see the *IntegralTransform* (page 586) docstring.

For how to compute inverse Fourier transforms, see the *inverse_fourier_transform()* (page 580) docstring.

sympy.integrals.transforms.**sine_transform**(*f*, *x*, *k*, ***hints*)

Compute the unitary, ordinary-frequency sine transform of f, defined as

$$F(k) = \sqrt{\frac{2}{\pi}} \int_0^\infty f(x) \sin(2\pi x k) \mathrm{d}x.$$

Explanation

If the transform cannot be computed in closed form, this function returns an unevaluated *SineTransform* (page 582) object.

For a description of possible hints, refer to the docstring of *sympy.integrals.transforms.IntegralTransform.doit()* (page 586). Note that for this transform, by default `noconds=True`.

Examples

```
>>> from sympy import sine_transform, exp
>>> from sympy.abc import x, k, a
>>> sine_transform(x*exp(-a*x**2), x, k)
sqrt(2)*k*exp(-k**2/(4*a))/(4*a**(3/2))
>>> sine_transform(x**(-a), x, k)
2**(1/2 - a)*k**(a - 1)*gamma(1 - a/2)/gamma(a/2 + 1/2)
```

See also:

fourier_transform (page 579), *inverse_fourier_transform* (page 580), *inverse_sine_transform* (page 582), *cosine_transform* (page 583), *inverse_cosine_transform* (page 583), *hankel_transform* (page 584), *inverse_hankel_transform* (page 585), *mellin_transform* (page 576), *laplace_transform* (page 577)

class sympy.integrals.transforms.**SineTransform**(**args*)

Class representing unevaluated sine transforms.

For usage of this class, see the *IntegralTransform* (page 586) docstring.

For how to compute sine transforms, see the *sine_transform()* (page 581) docstring.

sympy.integrals.transforms.**inverse_sine_transform**(*F*, *k*, *x*, ***hints*)

Compute the unitary, ordinary-frequency inverse sine transform of F, defined as

$$f(x) = \sqrt{\frac{2}{\pi}} \int_0^\infty F(k) \sin(2\pi x k) \mathrm{d}k.$$

Explanation

If the transform cannot be computed in closed form, this function returns an unevaluated *InverseSineTransform* (page 583) object.

For a description of possible hints, refer to the docstring of *sympy.integrals.transforms.IntegralTransform.doit()* (page 586). Note that for this transform, by default `noconds=True`.

Examples

```
>>> from sympy import inverse_sine_transform, exp, sqrt, gamma
>>> from sympy.abc import x, k, a
>>> inverse_sine_transform(2**((1-2*a)/2)*k**(a - 1)*
...     gamma(-a/2 + 1)/gamma((a+1)/2), k, x)
x**(-a)
>>> inverse_sine_transform(sqrt(2)*k*exp(-k**2/(4*a))/(4*sqrt(a)**3), k,␣
↪x)
x*exp(-a*x**2)
```

See also:

fourier_transform (page 579), *inverse_fourier_transform* (page 580), *sine_transform* (page 581), *cosine_transform* (page 583),

inverse_cosine_transform (page 583), *hankel_transform* (page 584), *inverse_hankel_transform* (page 585), *mellin_transform* (page 576), *laplace_transform* (page 577)

class sympy.integrals.transforms.**InverseSineTransform**(**args*)

Class representing unevaluated inverse sine transforms.

For usage of this class, see the *IntegralTransform* (page 586) docstring.

For how to compute inverse sine transforms, see the *inverse_sine_transform()* (page 582) docstring.

sympy.integrals.transforms.**cosine_transform**(*f*, *x*, *k*, ***hints*)

Compute the unitary, ordinary-frequency cosine transform of f, defined as

$$F(k) = \sqrt{\frac{2}{\pi}} \int_0^\infty f(x) \cos(2\pi x k) \mathrm{d}x.$$

Explanation

If the transform cannot be computed in closed form, this function returns an unevaluated *CosineTransform* (page 583) object.

For a description of possible hints, refer to the docstring of *sympy.integrals.transforms.IntegralTransform.doit()* (page 586). Note that for this transform, by default `noconds=True`.

Examples

```
>>> from sympy import cosine_transform, exp, sqrt, cos
>>> from sympy.abc import x, k, a
>>> cosine_transform(exp(-a*x), x, k)
sqrt(2)*a/(sqrt(pi)*(a**2 + k**2))
>>> cosine_transform(exp(-a*sqrt(x))*cos(a*sqrt(x)), x, k)
a*exp(-a**2/(2*k))/(2*k**(3/2))
```

See also:

fourier_transform (page 579), *inverse_fourier_transform* (page 580), *sine_transform* (page 581), *inverse_sine_transform* (page 582), *inverse_cosine_transform* (page 583), *hankel_transform* (page 584), *inverse_hankel_transform* (page 585), *mellin_transform* (page 576), *laplace_transform* (page 577)

class sympy.integrals.transforms.**CosineTransform**(**args*)

Class representing unevaluated cosine transforms.

For usage of this class, see the *IntegralTransform* (page 586) docstring.

For how to compute cosine transforms, see the *cosine_transform()* (page 583) docstring.

sympy.integrals.transforms.**inverse_cosine_transform**(*F*, *k*, *x*, ***hints*)

Compute the unitary, ordinary-frequency inverse cosine transform of F, defined as

$$f(x) = \sqrt{\frac{2}{\pi}} \int_0^\infty F(k) \cos(2\pi x k) \mathrm{d}k.$$

Explanation

If the transform cannot be computed in closed form, this function returns an unevaluated *InverseCosineTransform* (page 584) object.

For a description of possible hints, refer to the docstring of *sympy.integrals.transforms.IntegralTransform.doit()* (page 586). Note that for this transform, by default `noconds=True`.

Examples

```
>>> from sympy import inverse_cosine_transform, sqrt, pi
>>> from sympy.abc import x, k, a
>>> inverse_cosine_transform(sqrt(2)*a/(sqrt(pi)*(a**2 + k**2)), k, x)
exp(-a*x)
>>> inverse_cosine_transform(1/sqrt(k), k, x)
1/sqrt(x)
```

See also:

fourier_transform (page 579), *inverse_fourier_transform* (page 580), *sine_transform* (page 581), *inverse_sine_transform* (page 582), *cosine_transform* (page 583), *hankel_transform* (page 584), *inverse_hankel_transform* (page 585), *mellin_transform* (page 576), *laplace_transform* (page 577)

class sympy.integrals.transforms.**InverseCosineTransform**(**args*)

Class representing unevaluated inverse cosine transforms.

For usage of this class, see the *IntegralTransform* (page 586) docstring.

For how to compute inverse cosine transforms, see the *inverse_cosine_transform()* (page 583) docstring.

sympy.integrals.transforms.**hankel_transform**(*f*, *r*, *k*, *nu*, ***hints*)

Compute the Hankel transform of f, defined as

$$F_\nu(k) = \int_0^\infty f(r) J_\nu(kr) r \mathrm{d}r.$$

Explanation

If the transform cannot be computed in closed form, this function returns an unevaluated *HankelTransform* (page 585) object.

For a description of possible hints, refer to the docstring of *sympy.integrals.transforms.IntegralTransform.doit()* (page 586). Note that for this transform, by default `noconds=True`.

Examples

```
>>> from sympy import hankel_transform, inverse_hankel_transform
>>> from sympy import exp
>>> from sympy.abc import r, k, m, nu, a
```

```
>>> ht = hankel_transform(1/r**m, r, k, nu)
>>> ht
2*k**(m - 2)*gamma(-m/2 + nu/2 + 1)/(2**m*gamma(m/2 + nu/2))
```

```
>>> inverse_hankel_transform(ht, k, r, nu)
r**(-m)
```

```
>>> ht = hankel_transform(exp(-a*r), r, k, 0)
>>> ht
a/(k**3*(a**2/k**2 + 1)**(3/2))
```

```
>>> inverse_hankel_transform(ht, k, r, 0)
exp(-a*r)
```

See also:

fourier_transform (page 579), *inverse_fourier_transform* (page 580), *sine_transform* (page 581), *inverse_sine_transform* (page 582), *cosine_transform* (page 583), *inverse_cosine_transform* (page 583), *inverse_hankel_transform* (page 585), *mellin_transform* (page 576), *laplace_transform* (page 577)

class sympy.integrals.transforms.**HankelTransform**(**args*)

Class representing unevaluated Hankel transforms.

For usage of this class, see the *IntegralTransform* (page 586) docstring.

For how to compute Hankel transforms, see the *hankel_transform()* (page 584) docstring.

sympy.integrals.transforms.**inverse_hankel_transform**(*F*, *k*, *r*, *nu*, ***hints*)

Compute the inverse Hankel transform of F defined as

$$f(r) = \int_0^\infty F_\nu(k) J_\nu(kr) k \mathrm{d}k.$$

Explanation

If the transform cannot be computed in closed form, this function returns an unevaluated *InverseHankelTransform* (page 586) object.

For a description of possible hints, refer to the docstring of *sympy.integrals.transforms.IntegralTransform.doit()* (page 586). Note that for this transform, by default `noconds=True`.

Examples

```
>>> from sympy import hankel_transform, inverse_hankel_transform
>>> from sympy import exp
>>> from sympy.abc import r, k, m, nu, a
```

```
>>> ht = hankel_transform(1/r**m, r, k, nu)
>>> ht
2*k**(m - 2)*gamma(-m/2 + nu/2 + 1)/(2**m*gamma(m/2 + nu/2))
```

```
>>> inverse_hankel_transform(ht, k, r, nu)
r**(-m)
```

```
>>> ht = hankel_transform(exp(-a*r), r, k, 0)
>>> ht
a/(k**3*(a**2/k**2 + 1)**(3/2))
```

```
>>> inverse_hankel_transform(ht, k, r, 0)
exp(-a*r)
```

See also:

fourier_transform (page 579), *inverse_fourier_transform* (page 580), *sine_transform* (page 581), *inverse_sine_transform* (page 582), *cosine_transform* (page 583), *inverse_cosine_transform* (page 583), *hankel_transform* (page 584), *mellin_transform* (page 576), *laplace_transform* (page 577)

class sympy.integrals.transforms.**InverseHankelTransform**(**args*)

Class representing unevaluated inverse Hankel transforms.

For usage of this class, see the *IntegralTransform* (page 586) docstring.

For how to compute inverse Hankel transforms, see the *inverse_hankel_transform()* (page 585) docstring.

class sympy.integrals.transforms.**IntegralTransform**(**args*)

Base class for integral transforms.

Explanation

This class represents unevaluated transforms.

To implement a concrete transform, derive from this class and implement the `_compute_transform(f, x, s, **hints)` and `_as_integral(f, x, s)` functions. If the transform cannot be computed, raise *IntegralTransformError* (page 587).

Also set `cls._name`. For instance,

```
>>> from sympy import LaplaceTransform
>>> LaplaceTransform._name
'Laplace'
```

Implement `self._collapse_extra` if your function returns more than just a number and possibly a convergence condition.

doit(***hints*)

Try to evaluate the transform in closed form.

Explanation

This general function handles linearity, but apart from that leaves pretty much everything to _compute_transform.

Standard hints are the following:

- `simplify`: whether or not to simplify the result
- `noconds`: if True, do not return convergence conditions
- **`needeval`: if True, raise IntegralTransformError instead of** returning IntegralTransform objects

The default values of these hints depend on the concrete transform, usually the default is `(simplify, noconds, needeval) = (True, False, False)`.

property function

The function to be transformed.

property function_variable

The dependent variable of the function to be transformed.

property transform_variable

The independent transform variable.

exception sympy.integrals.transforms.**IntegralTransformError**(*transform*, *function*, *msg*)

Exception raised in relation to problems computing transforms.

Explanation

This class is mostly used internally; if integrals cannot be computed objects representing unevaluated transforms are usually returned.

The hint `needeval=True` can be used to disable returning transform objects, and instead raise this exception if an integral cannot be computed.

Internals

SymPy uses a number of algorithms to compute integrals. Algorithms are tried in order until one produces an answer. Most of these algorithms can be enabled or disabled manually using various flags to *integrate()* (page 598) or *doit()* (page 603).

SymPy first applies several heuristic algorithms, as these are the fastest:

1. If the function is a rational function, there is a complete algorithm for integrating rational functions called the Lazard-Rioboo-Trager and the Horowitz-Ostrogradsky algorithms. They are implemented in *ratint()* (page 587).

   sympy.integrals.rationaltools.**ratint**(*f*, *x*, ***flags*)

   Performs indefinite integration of rational functions.

Explanation

Given a field K and a rational function $f = p/q$, where p and q are polynomials in $K[x]$, returns a function g such that $f = g'$.

Examples

```
>>> from sympy.integrals.rationaltools import ratint
>>> from sympy.abc import x
```

```
>>> ratint(36/(x**5 - 2*x**4 - 2*x**3 + 4*x**2 + x - 2), x)
(12*x + 6)/(x**2 - 1) + 4*log(x - 2) - 4*log(x + 1)
```

See also:

sympy.integrals.integrals.Integral.doit (page 603), *sympy.integrals.rationaltools.ratint_logpart* (page 588), *sympy.integrals.rationaltools.ratint_ratpart* (page 588)

References

[R525]

sympy.integrals.rationaltools.**ratint_ratpart**(*f*, *g*, *x*)

Horowitz-Ostrogradsky algorithm.

Explanation

Given a field K and polynomials f and g in K[x], such that f and g are coprime and deg(f) < deg(g), returns fractions A and B in K(x), such that f/g = A' + B and B has square-free denominator.

Examples

```
>>> from sympy.integrals.rationaltools import ratint_ratpart
>>> from sympy.abc import x, y
>>> from sympy import Poly
>>> ratint_ratpart(Poly(1, x, domain='ZZ'),
... Poly(x + 1, x, domain='ZZ'), x)
(0, 1/(x + 1))
>>> ratint_ratpart(Poly(1, x, domain='EX'),
... Poly(x**2 + y**2, x, domain='EX'), x)
(0, 1/(x**2 + y**2))
>>> ratint_ratpart(Poly(36, x, domain='ZZ'),
... Poly(x**5 - 2*x**4 - 2*x**3 + 4*x**2 + x - 2, x, domain='ZZ'), x)
((12*x + 6)/(x**2 - 1), 12/(x**2 - x - 2))
```

See also:

ratint (page 587), *ratint_logpart* (page 588)

sympy.integrals.rationaltools.**ratint_logpart**(*f*, *g*, *x*, *t=None*)

Lazard-Rioboo-Trager algorithm.

Explanation

Given a field K and polynomials f and g in K[x], such that f and g are coprime, deg(f) < deg(g) and g is square-free, returns a list of tuples (s_i, q_i) of polynomials, for i = 1..n, such that s_i in K[t, x] and q_i in K[t], and:

```
                ___`   ___`
d  f   d  \     \      \
-- - = --  )      )    a log(s_i(a, x))
dx g   dx /__,   /__,
          i=1..n a | q_i(a) = 0
```

Examples

```
>>> from sympy.integrals.rationaltools import ratint_logpart
>>> from sympy.abc import x
>>> from sympy import Poly
>>> ratint_logpart(Poly(1, x, domain='ZZ'),
... Poly(x**2 + x + 1, x, domain='ZZ'), x)
[(Poly(x + 3*_t/2 + 1/2, x, domain='QQ[_t]'),
...Poly(3*_t**2 + 1, _t, domain='ZZ'))]
>>> ratint_logpart(Poly(12, x, domain='ZZ'),
... Poly(x**2 - x - 2, x, domain='ZZ'), x)
[(Poly(x - 3*_t/8 - 1/2, x, domain='QQ[_t]'),
...Poly(-_t**2 + 16, _t, domain='ZZ'))]
```

See also:

ratint (page 587), *ratint_ratpart* (page 588)

2. *trigintegrate()* (page 589) solves integrals of trigonometric functions using pattern matching

sympy.integrals.trigonometry.**trigintegrate**(*f*, *x*, *conds='piecewise'*)

Integrate f = Mul(trig) over x.

Examples

```
>>> from sympy import sin, cos, tan, sec
>>> from sympy.integrals.trigonometry import trigintegrate
>>> from sympy.abc import x
```

```
>>> trigintegrate(sin(x)*cos(x), x)
sin(x)**2/2
```

```
>>> trigintegrate(sin(x)**2, x)
x/2 - sin(x)*cos(x)/2
```

```
>>> trigintegrate(tan(x)*sec(x), x)
1/cos(x)
```

```
>>> trigintegrate(sin(x)*tan(x), x)
-log(sin(x) - 1)/2 + log(sin(x) + 1)/2 - sin(x)
```

See also:

sympy.integrals.integrals.Integral.doit (page 603), *sympy.integrals.integrals.Integral* (page 601)

References

[R526]

3. *deltaintegrate()* (page 590) solves integrals with *DiracDelta* (page 450) objects.

sympy.integrals.deltafunctions.**deltaintegrate**(*f*, *x*)

Explanation

The idea for integration is the following:

- If we are dealing with a DiracDelta expression, i.e. DiracDelta(g(x)), we try to simplify it.

  If we could simplify it, then we integrate the resulting expression. We already know we can integrate a simplified expression, because only simple DiracDelta expressions are involved.

  If we couldn't simplify it, there are two cases:

  1) The expression is a simple expression: we return the integral, taking care if we are dealing with a Derivative or with a proper DiracDelta.
  2) The expression is not simple (i.e. DiracDelta(cos(x))): we can do nothing at all.

- If the node is a multiplication node having a DiracDelta term:

  First we expand it.

  If the expansion did work, then we try to integrate the expansion.

  If not, we try to extract a simple DiracDelta term, then we have two cases:

  1) We have a simple DiracDelta term, so we return the integral.
  2) We didn't have a simple term, but we do have an expression with simplified DiracDelta terms, so we integrate this expression.

Examples

```
>>> from sympy.abc import x, y, z
>>> from sympy.integrals.deltafunctions import deltaintegrate
>>> from sympy import sin, cos, DiracDelta
>>> deltaintegrate(x*sin(x)*cos(x)*DiracDelta(x - 1), x)
sin(1)*cos(1)*Heaviside(x - 1)
>>> deltaintegrate(y**2*DiracDelta(x - z)*DiracDelta(y - z), y)
z**2*DiracDelta(x - z)*Heaviside(y - z)
```

See also:

sympy.functions.special.delta_functions.DiracDelta (page 450), *sympy.integrals.integrals.Integral* (page 601)

4. *singularityintegrate()* (page 591) is applied if the function contains a *SingularityFunction* (page 456)

sympy.integrals.singularityfunctions.**singularityintegrate**(*f*, *x*)

This function handles the indefinite integrations of Singularity functions. The `integrate` function calls this function internally whenever an instance of SingularityFunction is passed as argument.

Explanation

The idea for integration is the following:

- If we are dealing with a SingularityFunction expression, i.e. `SingularityFunction(x, a, n)`, we just return `SingularityFunction(x, a, n + 1)/(n + 1)` if `n >= 0` and `SingularityFunction(x, a, n + 1)` if `n < 0`.
- If the node is a multiplication or power node having a SingularityFunction term we rewrite the whole expression in terms of Heaviside and DiracDelta and then integrate the output. Lastly, we rewrite the output of integration back in terms of SingularityFunction.
- If none of the above case arises, we return None.

Examples

```
>>> from sympy.integrals.singularityfunctions import␣
↪singularityintegrate
>>> from sympy import SingularityFunction, symbols, Function
>>> x, a, n, y = symbols('x a n y')
>>> f = Function('f')
>>> singularityintegrate(SingularityFunction(x, a, 3), x)
SingularityFunction(x, a, 4)/4
>>> singularityintegrate(5*SingularityFunction(x, 5, -2), x)
5*SingularityFunction(x, 5, -1)
>>> singularityintegrate(6*SingularityFunction(x, 5, -1), x)
6*SingularityFunction(x, 5, 0)
>>> singularityintegrate(x*SingularityFunction(x, 0, -1), x)
```

(continues on next page)

(continued from previous page)

```
0
>>> singularityintegrate(SingularityFunction(x, 1, -1) * f(x), x)
f(1)*SingularityFunction(x, 1, 0)
```

5. If the heuristic algorithms cannot be applied, *risch_integrate()* (page 592) is tried next. The *Risch algorithm* is a general method for calculating antiderivatives of elementary functions. The Risch algorithm is a decision procedure that can determine whether an elementary solution exists, and in that case calculate it. It can be extended to handle many nonelementary functions in addition to the elementary ones. However, the version implemented in SymPy only supports a small subset of the full algorithm, particularly, on part of the transcendental algorithm for exponentials and logarithms is implemented. An advantage of *risch_integrate()* (page 592) over other methods is that if it returns an instance of *NonElementaryIntegral* (page 594), the integral is proven to be nonelementary by the algorithm, meaning the integral cannot be represented using a combination of exponentials, logarithms, trig functions, powers, rational functions, algebraic functions, and function composition.

   sympy.integrals.risch.**risch_integrate**(*f*, *x*, *extension=None*, *handle_first='log'*, *separate_integral=False*, *rewrite_complex=None*, *conds='piecewise'*)

   The Risch Integration Algorithm.

   ### Explanation

   Only transcendental functions are supported. Currently, only exponentials and logarithms are supported, but support for trigonometric functions is forthcoming.

   If this function returns an unevaluated Integral in the result, it means that it has proven that integral to be nonelementary. Any errors will result in raising NotImplementedError. The unevaluated Integral will be an instance of NonElementaryIntegral, a subclass of Integral.

   handle_first may be either ‘exp’ or ‘log’. This changes the order in which the extension is built, and may result in a different (but equivalent) solution (for an example of this, see issue 5109). It is also possible that the integral may be computed with one but not the other, because not all cases have been implemented yet. It defaults to ‘log’ so that the outer extension is exponential when possible, because more of the exponential case has been implemented.

   If `separate_integral` is `True`, the result is returned as a tuple (ans, i), where the integral is ans + i, ans is elementary, and i is either a NonElementaryIntegral or 0. This useful if you want to try further integrating the NonElementaryIntegral part using other algorithms to possibly get a solution in terms of special functions. It is False by default.

Examples

```
>>> from sympy.integrals.risch import risch_integrate
>>> from sympy import exp, log, pprint
>>> from sympy.abc import x
```

First, we try integrating exp(-x**2). Except for a constant factor of 2/sqrt(pi), this is the famous error function.

```
>>> pprint(risch_integrate(exp(-x**2), x))
  /
 |
 |    2
 |  -x
 | e    dx
 |
/
```

The unevaluated Integral in the result means that risch_integrate() has proven that exp(-x**2) does not have an elementary anti-derivative.

In many cases, risch_integrate() can split out the elementary anti-derivative part from the nonelementary anti-derivative part. For example,

```
>>> pprint(risch_integrate((2*log(x)**2 - log(x) - x**2)/(log(x)**3 -
... x**2*log(x)), x))
                                      /
                                     |
  log(-x + log(x))   log(x + log(x))  |   1
- ---------------- + --------------- + | ------ dx
         2                  2         | log(x)
                                      |
                                     /
```

This means that it has proven that the integral of 1/log(x) is nonelementary. This function is also known as the logarithmic integral, and is often denoted as Li(x).

risch_integrate() currently only accepts purely transcendental functions with exponentials and logarithms, though note that this can include nested exponentials and logarithms, as well as exponentials with bases other than E.

```
>>> pprint(risch_integrate(exp(x)*exp(exp(x)), x))
 / x\
 \e /
e
>>> pprint(risch_integrate(exp(exp(x)), x))
  /
 |
 |  / x\
 |  \e /
 | e     dx
 |
/
```

```
>>> pprint(risch_integrate(x*x**x*log(x) + x**x + x*x**x, x))
   x
x*x
>>> pprint(risch_integrate(x**x, x))
  /
 |
 |  x
 | x  dx
 |
/
```

```
>>> pprint(risch_integrate(-1/(x*log(x)*log(log(x))**2), x))
     1
-----------
log(log(x))
```

class sympy.integrals.risch.**NonElementaryIntegral**(*function*, **symbols*, ***assumptions*)

Represents a nonelementary Integral.

Explanation

If the result of integrate() is an instance of this class, it is guaranteed to be nonelementary. Note that integrate() by default will try to find any closed-form solution, even in terms of special functions which may themselves not be elementary. To make integrate() only give elementary solutions, or, in the cases where it can prove the integral to be nonelementary, instances of this class, use integrate(risch=True). In this case, integrate() may raise NotImplementedError if it cannot make such a determination.

integrate() uses the deterministic Risch algorithm to integrate elementary functions or prove that they have no elementary integral. In some cases, this algorithm can split an integral into an elementary and nonelementary part, so that the result of integrate will be the sum of an elementary expression and a NonElementaryIntegral.

Examples

```
>>> from sympy import integrate, exp, log, Integral
>>> from sympy.abc import x
```

```
>>> a = integrate(exp(-x**2), x, risch=True)
>>> print(a)
Integral(exp(-x**2), x)
>>> type(a)
<class 'sympy.integrals.risch.NonElementaryIntegral'>
```

```
>>> expr = (2*log(x)**2 - log(x) - x**2)/(log(x)**3 - x**2*log(x))
>>> b = integrate(expr, x, risch=True)
>>> print(b)
```

(continues on next page)

(continued from previous page)

```
-log(-x + log(x))/2 + log(x + log(x))/2 + Integral(1/log(x), x)
>>> type(b.atoms(Integral).pop())
<class 'sympy.integrals.risch.NonElementaryIntegral'>
```

6. For non-elementary definite integrals, SymPy uses so-called Meijer G-functions. Details are described in *Computing Integrals using Meijer G-Functions* (page 554).

7. All the algorithms mentioned thus far are either pattern-matching based heuristic, or solve integrals using algorithms that are much different from the way most people are taught in their calculus courses. SymPy also implements a method that can solve integrals in much the same way you would in calculus. The advantage of this method is that it is possible to extract the integration steps from, so that one can see how to compute the integral "by hand". This is used by SymPy Gamma. This is implemented in the *manualintegrate()* (page 595) function. The steps for an integral can be seen with the *integral_steps()* (page 596) function.

   sympy.integrals.manualintegrate.**manualintegrate**(*f*, *var*)

   #### Explanation

   Compute indefinite integral of a single variable using an algorithm that resembles what a student would do by hand.

   Unlike *integrate()* (page 598), var can only be a single symbol.

   #### Examples

```
>>> from sympy import sin, cos, tan, exp, log, integrate
>>> from sympy.integrals.manualintegrate import manualintegrate
>>> from sympy.abc import x
>>> manualintegrate(1 / x, x)
log(x)
>>> integrate(1/x)
log(x)
>>> manualintegrate(log(x), x)
x*log(x) - x
>>> integrate(log(x))
x*log(x) - x
>>> manualintegrate(exp(x) / (1 + exp(2 * x)), x)
atan(exp(x))
>>> integrate(exp(x) / (1 + exp(2 * x)))
RootSum(4*_z**2 + 1, Lambda(_i, _i*log(2*_i + exp(x))))
>>> manualintegrate(cos(x)**4 * sin(x), x)
-cos(x)**5/5
>>> integrate(cos(x)**4 * sin(x), x)
-cos(x)**5/5
>>> manualintegrate(cos(x)**4 * sin(x)**3, x)
cos(x)**7/7 - cos(x)**5/5
>>> integrate(cos(x)**4 * sin(x)**3, x)
cos(x)**7/7 - cos(x)**5/5
>>> manualintegrate(tan(x), x)
```

(continues on next page)

(continued from previous page)

```
-log(cos(x))
>>> integrate(tan(x), x)
-log(cos(x))
```

See also:

sympy.integrals.integrals.integrate (page 598), *sympy.integrals.integrals.Integral.doit* (page 603), *sympy.integrals.integrals.Integral* (page 601)

sympy.integrals.manualintegrate.**integral_steps**(*integrand*, *symbol*, ***options*)

Returns the steps needed to compute an integral.

Returns

rule : namedtuple

The first step; most rules have substeps that must also be considered. These substeps can be evaluated using `manualintegrate` to obtain a result.

Explanation

This function attempts to mirror what a student would do by hand as closely as possible.

SymPy Gamma uses this to provide a step-by-step explanation of an integral. The code it uses to format the results of this function can be found at https://github.com/sympy/sympy_gamma/blob/master/app/logic/intsteps.py.

Examples

```
>>> from sympy import exp, sin
>>> from sympy.integrals.manualintegrate import integral_steps
>>> from sympy.abc import x
>>> print(repr(integral_steps(exp(x) / (1 + exp(2 * x)), x)))
URule(u_var=_u, u_func=exp(x), constant=1,
substep=ArctanRule(a=1, b=1, c=1, context=1/(_u**2 + 1), symbol=_u),
context=exp(x)/(exp(2*x) + 1), symbol=x)
>>> print(repr(integral_steps(sin(x), x)))
TrigRule(func='sin', arg=x, context=sin(x), symbol=x)
>>> print(repr(integral_steps((x**2 + 3)**2, x)))
RewriteRule(rewritten=x**4 + 6*x**2 + 9,
substep=AddRule(substeps=[PowerRule(base=x, exp=4, context=x**4,␣
↪symbol=x),
    ConstantTimesRule(constant=6, other=x**2,
        substep=PowerRule(base=x, exp=2, context=x**2, symbol=x),
            context=6*x**2, symbol=x),
    ConstantRule(constant=9, context=9, symbol=x)],
context=x**4 + 6*x**2 + 9, symbol=x), context=(x**2 + 3)**2, symbol=x)
```

8. Finally, if all the above fail, SymPy also uses a simplified version of the Risch algorithm, called the *Risch-Norman algorithm*. This algorithm is tried last because it is often the slowest to compute. This is implemented in *heurisch()* (page 596):