14

Numerical Integration and Differential Equations

This chapter covers the numerical computation of integrals (§14.1) and the numerical resolution of ordinary differential equations (§14.2) with Sage. We recall the theoretical bases of integration methods, then we detail the available functions and their usage (§14.1.1).

We have already seen in §2.3.8 how to compute an integral *symbolically* with Sage; this will only be briefly mentioned in this chapter. This "symbolic-numeric" approach, when it is possible, is one of the strengths of Sage, and should be preferred because the number of numerical computations performed — and thus the number of roundoff errors — is usually less than with purely numerical integration methods.

For differential equations, we give a quick introduction to the classical resolution methods, and after an introductory example (§14.2.1), we describe the functionalities of Sage (§14.2.2).

14.1 Numerical Integration

We consider the numerical integration of real functions; for a function $f : I \longrightarrow \mathbb{R}$, where I is an interval of $\mathbb{R}$, we want to approximate:

$$\int_I f(x)\,\mathrm{d}x.$$

For example, let us compute

$$\int_1^3 \exp(-x^2)\log(x)\,\mathrm{d}x.$$

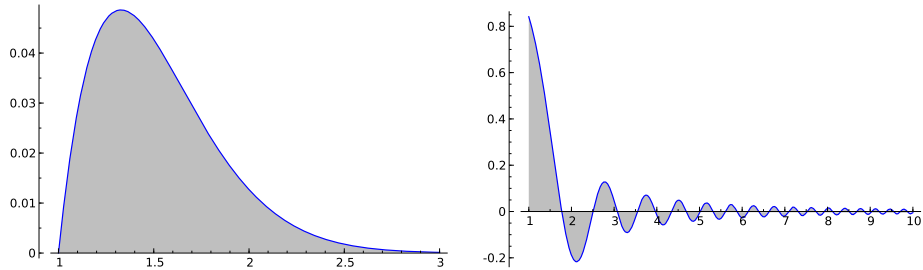


FIGURE 14.1 – The functions $x \mapsto \exp(-x^2)\log(x)$ and $x \mapsto \frac{\sin(x^2)}{x^2}$.

```
sage: x = var('x'); f(x) = exp(-x^2) * log(x)
sage: N(integrate(f, x, 1, 3))
0.035860294991267694
```

```
sage: plot(f, 1, 3, fill='axis')
```

Since the `integrate` function computes a symbolic integral of the given expression, we have to explicitly ask if we want a numerical value.

It is also possible, in principle, to compute integrals on an unbounded interval:

```
sage: N(integrate(sin(x^2)/(x^2), x, 1, infinity))
0.285736646322853 + 6.93889390390723e-18*I
```

```
sage: plot(sin(x^2)/(x^2), x, 1, 10, fill='axis')
```

Several methods exist in Sage to perform numerical integration, and if their implementations differ technically, they all follow one of the two following principles:

- a polynomial interpolation (in particular the Gauss-Kronrod method);
- a function transformation (double exponential method).

Interpolation Methods. In these methods, we evaluate the function f to integrate at a given number n of well-chosen points $x_1, x_2, \ldots, x_n$, and we deduce an approximation of the integral of f on $[a, b]$ by

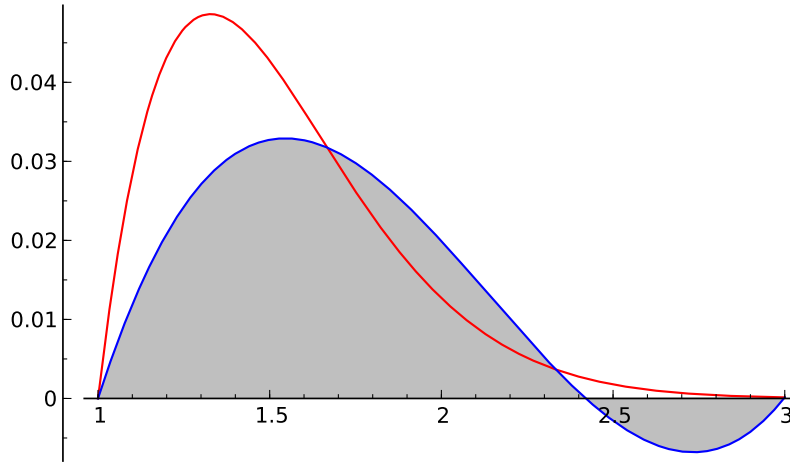
$$\int_a^b f(x)\,\mathrm{d}x \approx \sum_{i=1}^{n} w_i f(x_i).$$

The w_i coefficients are the "weights" of the method, which are determined by the fact that the method should be exact for any polynomial f of degree less or equal to $n-1$. For fixed points (x_i), the weights (w_i) are uniquely determined by this condition. We define the *order* of the method as the maximal degree of

polynomials whose integral is exact; this order is thus at least $n-1$ by construction, but it might be larger.

For example, the family of Newton-Cotes integration methods (which contains the rectangle rule, the trapezoidal rule, Simpson's rule) use equally spaced points on the interval $[a, b]$:

```
sage: fp = plot(f, 1, 3, color='red')
sage: n = 4
sage: interp_points = [(1+2*u/(n-1), N(f(1+2*u/(n-1))))
....:                  for u in xrange(n)]
sage: A = PolynomialRing(RR, 'x')
sage: pp = plot(A.lagrange_polynomial(interp_points), 1, 3, fill='axis')
sage: fp+pp
```



For the interpolation methods, we can consider that we first compute the Lagrange interpolating polynomial of the given function, and that the integral of this polynomial is the chosen approximate value for the integral. These two steps are in fact merged into a formula called "quadrature rule", the Lagrange interpolation polynomial being never explicitly computed. The choice of the interpolation points is crucial for the quality of the polynomial approximation, and equally spaced points do not ensure convergence when the number of points increases (this is called Runge's phenomenon). The corresponding quadrature rule might thus suffer from this problem, illustrated in Figure 14.2.

When we ask Sage to compute a numerical approximation of an integral on an interval $[a, b]$, the quadrature rule is not directly applied to the whole domain: $[a, b]$ is divided into sub-intervals small enough such that the quadrature rule gives enough accuracy (this is called "method composition"). The subdivision strategy might be for example dynamically tuned for the function to integrate: if we denote $I_a^b(f)$ the value of $\int_a^b f(x)\,\mathrm{d}x$ computed by the quadrature rule, we compare

$$I_0 = I_a^b(f)$$

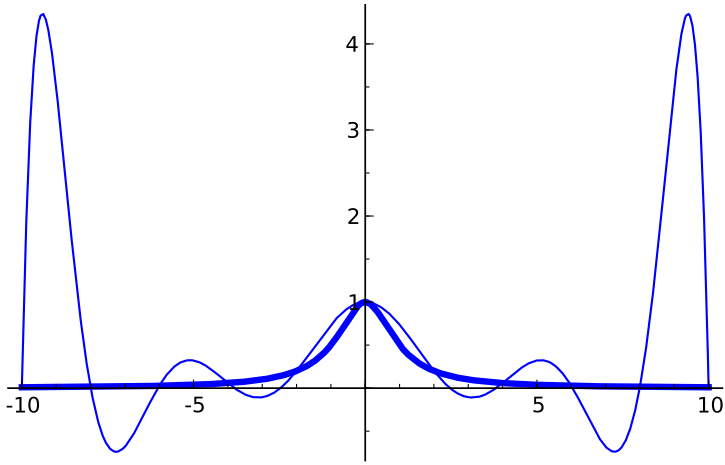


FIGURE 14.2 – Interpolation using a degree-10 polynomial (thin line) of the function $x \mapsto 1/(1+x^2)$ (thick line) at 11 points equally spaced on $[-10, 10]$. Runge's phenomenon appears at endpoints.

with

$$I_1 = I_a^{(a+b)/2}(f) + I_{(a+b)/2}^b(f)$$

and we stop the subdivision process when $|I_0 - I_1|$ is small enough compared to I_0 with the required precision. Here comes into play the method order: for a quadrature rule of order n, dividing the interval in 2 will divide the theoretical error by 2^n, i.e., without taking into account roundoff errors.

One of the interpolation methods available within Sage is the Gauss-Legendre method. In this method, the n integration points are the roots of the Legendre polynomial of degree n (with a translated interval of definition to match the considered range $[a, b]$). The properties of Legendre polynomials, which are orthogonal for the scalar product

$$\langle f, g \rangle = \int_a^b f(x)g(x)\,\mathrm{d}x,$$

imply that the corresponding quadrature rule computes exactly the integrals of polynomials of degree up to $2n-1$, instead of only $n-1$ as in the general case. Moreover, the corresponding integration weights are always positive, which minimises the effect of numerical issues like *cancellation*[1].

To conclude on interpolation methods, the Gauss-Kronrod method with $2n+1$ points is an "extension" of the Gauss-Legendre method with n points:

- n of the $2n+1$ points are the Gauss-Legendre integration points;

[1]This phenomenon happens when a sum of real numbers is significantly smaller (in absolute value) than the summands: each rounding error might then be larger than the final result, which yields a total loss of accuracy. See also §11.3.3.

- the method is exact for any polynomial of degree up to $3n+1$.

We can naively remark that the $3n+2$ unknowns (the $2n+1$ weights and the $n+1$ added points) are *a priori* determined by requiring that the method is of order at least $3n+1$ (which indeed yields $3n+2$ equalities). Beware that the weights associated in the Gauss-Kronrod method to the n Gauss-Legendre points have no reason to coincide with those associated in the original Gauss-Legendre method.

Such an extension method becomes particularly interesting when the main cost of a quadrature rule is the number of evaluations of the function f to integrate (moreover if the integration points and weights are tabulated). The Gauss-Kronrod method being in principle more precise than the Gauss-Legendre method, we can use its result I_1 to validate the result I_0 of the latter method, and obtain an estimate of the error as $|I_1 - I_0|$, while minimising the number of evaluations of f. The reader might compare this strategy, which is particular to the Gauss-Legendre method, with the more general subdivision strategy described on page 300.

Double Exponential Methods. The double exponential (DE) methods rely on a change of variable which transforms a bounded integration range into $\mathbb{R}$, and on the very good accuracy of the trapezoidal rule on $\mathbb{R}$ for analytic functions. For a function f integrable on $\mathbb{R}$, and an integration step h, the trapezoidal rule computes

$$I_h = h \sum_{i=-\infty}^{+\infty} f(hi)$$

as approximate value for $\int_{-\infty}^{+\infty} f(x)\,\mathrm{d}x$. Discovered in 1973 by Takahasi and Mori, the double exponential transform is commonly used by numerical integration software tools. We describe here its main features; an introduction to this transform and its discovery is given in [Mor05]. This article gives in particular an explanation of the surprising good accuracy of the trapezoidal rule, which is optimal in a certain sense, for analytic functions on $\mathbb{R}$.

To compute

$$I = \int_{-1}^{1} f(x)\,\mathrm{d}x,$$

it is possible to use a transform $x = \varphi(t)$ where φ is analytic on $\mathbb{R}$ and satisfies

$$\lim_{t\to-\infty} \varphi(t) = -1, \quad \lim_{t\to\infty} \varphi(t) = 1,$$

and then

$$I = \int_{-\infty}^{\infty} f(\varphi(t))\varphi'(t)\,\mathrm{d}t.$$

Applying the trapezoidal rule to this last expression, we compute

$$I_h^N = h \sum_{k=-N}^{N} f(\varphi(kh))\varphi'(kh)$$

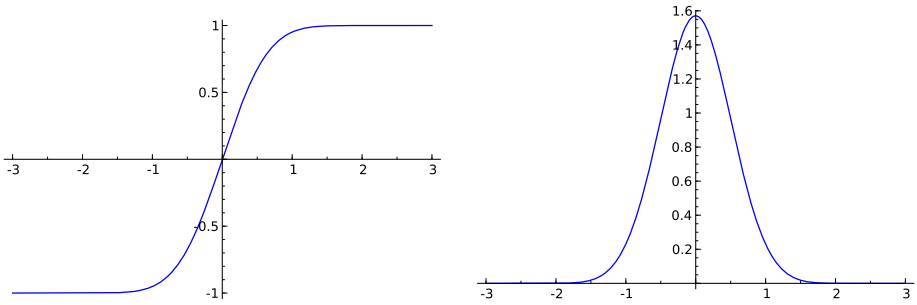


FIGURE 14.3 – The transform $\varphi(t) = \tanh(\frac{\pi}{2}\sinh t)$ used in the double exponential method (left) and the decreasing of $\varphi'(t)$ (right).

for a given step h, and truncating the sum to terms from $-N$ to N. The proposed transform is chosen to be

$$\varphi(t) = \tanh\left(\frac{\pi}{2}\sinh t\right)$$

which yields the formula

$$I_h^N = h\sum_{k=-N}^{N} f\left(\tanh\left(\frac{\pi}{2}\sinh kh\right)\right)\frac{\frac{\pi}{2}\cosh kh}{\cosh^2(\frac{\pi}{2}\sinh kh)}.$$

The name of the method comes from the doubly exponential decreasing of

$$\varphi'(t) = \frac{\frac{\pi}{2}\cosh t}{\cosh^2(\frac{\pi}{2}\sinh t)}$$

when $|t| \to \infty$ (see Figure 14.3). The main idea of the transform is to concentrate the contribution of the function to integrate around 0, which explains the huge decreasing of $\varphi'(t)$ when $|t|$ grows. A compromise should be found between the choice of parameters and of the transform φ: a decreasing more than doubly exponential decreases the truncation error but increases the discretisation error.

Since the discovery of the DE transform, this method is used alone or with other transforms, according to the nature of the integrand, of its singularities and of the integration domain. An example of other transform is the cardinal sine function "sinc":

$$f(x) \approx \sum_{k=-N}^{N} f(kh)S_{k,h}(x)$$

where

$$S_{k,h}(x) = \frac{\sin(\pi(x-kh)/h)}{\pi(x-kh)/h},$$

used together with the double exponential method in [TSM05] to improve the previous methods which relied on a single exponential transform $\varphi(t) = \tanh(t/2)$.

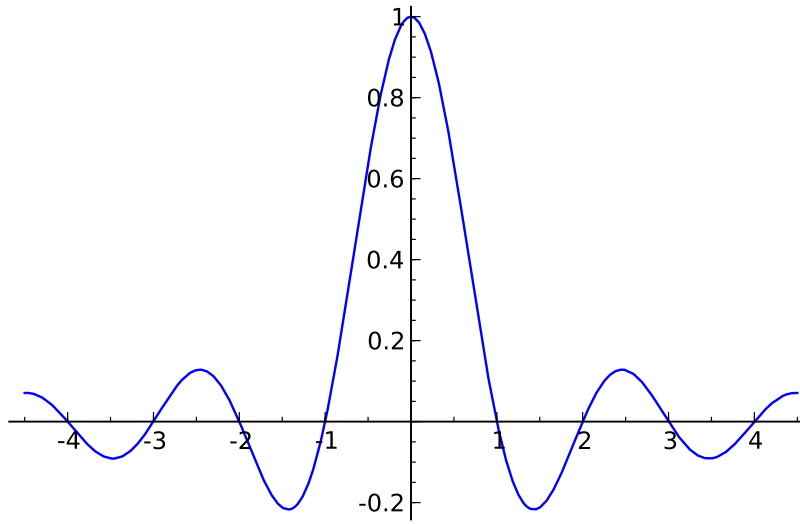


FIGURE 14.4 – The cardinal sine function.

The sinc function is defined by

$$\text{sinc} = \begin{cases} 1 & \text{if } x = 0, \\ \dfrac{\sin(\pi x)}{\pi x} & \text{otherwise,} \end{cases}$$

and its graph is shown in Figure 14.4.

The choice of the transform greatly influences the quality of the result in the case of singularities at the interval bounds (there is no good solution yet in the case of singularities inside the interval). We will see later that in the version of Sage we consider, PARI is the only system providing double exponential transforms allowing to specify the behaviour at interval bounds.

14.1.1 Available Integration Functions

We will now see in more detail the various ways to compute a numerical integral with Sage, through the following examples:

$$\begin{aligned} I_1 &= \int_{17}^{42} \exp(-x^2)\log(x)\,\mathrm{d}x, \quad I_2 = \int_0^1 x\log(1+x)\,\mathrm{d}x = \frac{1}{4}, \\ I_3 &= \int_0^1 \sqrt{1-x^2}\,\mathrm{d}x = \frac{\pi}{4}, \\ I_4 &= \int_0^1 \max(\sin(x),\cos(x))\,\mathrm{d}x = \int_0^{\frac{\pi}{4}} \cos(x)\,\mathrm{d}x + \int_{\frac{\pi}{4}}^1 \sin(x)\,\mathrm{d}x \\ &= \sin\frac{\pi}{4} + \cos\frac{\pi}{4} - \cos 1 = \sqrt{2} - \cos 1, \end{aligned}$$

$$
\begin{aligned}
I_5 &= \int_0^1 \sin(\sin(x))\,\mathrm{d}x, \quad I_6 = \int_0^\pi \sin(x)\exp(\cos(x))\,\mathrm{d}x = e - \frac{1}{e},\\
I_7 &= \int_0^1 \frac{1}{1+10^{10}x^2}\,\mathrm{d}x = 10^{-5}\arctan(10^5), \quad I_8 = \int_0^{1,1} \exp(-x^{100})\,\mathrm{d}x,\\
I_9 &= \int_0^{10} x^2\sin(x^3)\,\mathrm{d}x = \frac{1}{3}(1-\cos(1000)), \quad I_{10} = \int_0^1 \sqrt{x}\,\mathrm{d}x = \frac{2}{3}.
\end{aligned}
$$

We do not give an exhausive description of the integration functions — which can be found in the on-line help — but only their more common usage.

`N(integrate(...))`. The first numerical method which can be used with Sage is `N(integrate(...))`:

```
sage: N(integrate(exp(-x^2)*log(x), x, 17, 42))
2.5657285006962035e-127
```

It is not guaranteed that the integral will be computed really *numerically* this way. Indeed, the `integrate` command requires a symbolic integration; if this succeeds, then Sage will just evaluate numerically the obtained symbolic expression:

```
sage: integrate(log(1+x)*x, x, 0, 1)
1/4
sage: N(integrate(log(1+x)*x, x, 0, 1))
0.250000000000000
```

`numerical_integral`. On the contrary, the `numerical_integral` function *explicitely* requires a numerical integration of the given function. It calls the GSL library (GNU Scientific Library), which implements the Gauss-Kronrod method for a fixed number n of integration points. The points and weights are precomputed, and the precision is limited to that of machine floating-point numbers (53-bit significand). The result is a pair with the computed approximation and an estimate of the error:

```
sage: numerical_integral(exp(-x^2)*log(x), 17, 42)
(2.5657285006962035e-127, 3.3540254049238093e-128)
```

The error estimate is not a guaranteed bound on the error, but a simple indication of the difficulty to approximate the given integral. In the above example, the error estimate is so large that all the digits of the result might be wrong, except the most significant one.

The arguments of `numerical_integral` allow in particular:

- to choose the number of evaluation points (six choices from `rule=1` for 15 points to `rule=6` for 61 points, which is the default value);
- to ask for an adaptive subdivision (default choice), or require a direct application of the composition method on the integration interval (with the option `algorithm='qng'`);
- to bound the number of evaluations of the integrand.

Forbidding GSL to perform an adaptive integration might lead to a loss of accuracy:

```
sage: numerical_integral(exp(-x^100), 0, 1.1)
(0.99432585119150..., 4.0775730...e-09)
sage: numerical_integral(exp(-x^100), 0, 1.1, algorithm='qng')
(0.994327538576531..., 0.016840666914688864)
```

When the `integrate` command does not find an analytic expression for the requested integral, it returns the input integral unchanged:

```
sage: integrate(exp(-x^2)*log(x), x, 17, 42)
integrate(e^(-x^2)*log(x), x, 17, 42)
```

and the numerical evaluation with `N` calls `numerical_integral`. This explains in particular why the precision parameter is ignored in that case:

```
sage: N(integrate(exp(-x^2)*log(x), x, 17, 42), digits=60)
2.5657285006962035e-127
```

but we get:

```
sage: N(integrate(sin(x)*exp(cos(x)), x, 0, pi), digits=60)
2.350402387287602913764763701191201630311435962668191740459 13
```

because the symbolic integration succeeds in that case.

`sage.calculus.calculus.nintegral`. For the symbolic functions, it is possible to ask Maxima for a numerical approximation of the integral:

```
sage: sage.calculus.calculus.nintegral(sin(sin(x)), x, 0, 1)
(0.430606103120690..., 4.78068810228705...e-15, 21, 0)
```

and it is also possible to directly call the `nintegral` method on an object of type `Expression`:

```
sage: g = sin(sin(x))
sage: g.nintegral(x, 0, 1)
(0.430606103120690..., 4.78068810228705...e-15, 21, 0)
```

Maxima calls the QUADPACK numerical quadrature library, which like GSL is limited to machine floating-point numbers. The `nintegral` method uses an adaptive subdivision strategy of the integration interval, and we might indicate:

- the wanted relative accuracy of the result;
- the maximal number of sub-intervals for the computation.

The output is a tuple:

1. the approximation of the integral;
2. an estimate of the absolute error;
3. the number of evaluations of the integrand;
4. an error code (0 if no problem was encountered, for more details on the other possible values the reader should look at the reference manual with `sage.calculus.calculus.nintegral?`).

gp('intnum(...)'). The PARI calculator, which is available within Sage, also implements a numerical integration command called `intnum`:

```
sage: gp('intnum(x=17, 20, exp(-x^2)*log(x))')
2.5657285005610514829173563961304785900 E-127
```

The `intnum` command uses the double exponential method, but beware, it does not guarantee any correct significant digit of the result!

We might ask for a given precision of the result by modifying the global precision of the PARI interpreter:

```
sage: gp('intnum(x=0, 1, sin(sin(x)))')
0.43060610312069060491237735524846578643
sage: old_prec = gp.set_precision(50)
sage: gp('intnum(x=0, 1, sin(sin(x)))')
0.43060610312069060491237735524846578643360804182200
```

A major bottleneck is that the integrand must be given as a character string, following the PARI syntax, it is thus not possible to integrate arbitrary functions this way.

The `intnum` command allows us to indicate the behaviour of the integrand at the interval bounds. The following example demonstrates the corresponding effect on the result accuracy. Let us integrate $x \mapsto x^{-99/100}$ without any indication:

```
sage: p = gp.set_precision(old_prec) # we reset the default precision
sage: gp('intnum(x=0, 1, x^(-99/100))')
73.62914262423378369
```

If we give the nature of the singularity, i.e., that the function behaves like $x \mapsto x^{-99/100}$ at 0:

```
sage: gp('intnum(x=[0, -99/100], 1, x^(-99/100))')
100.00000000000000000000000000000000000
```

The user is responsible of the exactness of the given behaviour; if we erroneously say that the function behaves like $x \mapsto x^{-1/42}$ at 0, the error will remain large:

```
sage: gp('intnum(x=[0, -1/42], 1, x^(-99/100))')
74.47274932028288503
```

mpmath.quad*. The `mpmath` library is an arbitrary precision numerical library written in Python. It is able to compute with floating-point real and complex numbers, with matrices, and floating-point real intervals.

It provides numerical quadrature functions (the principal one being `quad`) and is available within Sage, after importing it:

```
sage: import mpmath
sage: mpmath.mp.prec = 53
sage: mpmath.quad(lambda x: mpmath.sin(mpmath.sin(x)), [0, 1])
mpf('0.43060610312069059')
```

The mpmath floating-point numbers mpf(...) might be converted in Sage floating-point numbers and vice-versa:

```
sage: a = RDF(pi); b = mpmath.mpf(a); b
mpf('3.1415926535897931')
sage: c = RDF(b); c
3.141592653589793
```

The user might specify the wanted precision either in decimal digits (mpmath.mp.dps) or in bits (mpmath.mp.prec), as with the command N from Sage: N(...,53) or N(...,digits=17).

```
sage: mpmath.mp.prec = 113
sage: mpmath.quad(lambda x: mpmath.sin(mpmath.sin(x)), [0, 1])
mpf('0.43060610312069060491237735524846580 9')
```

The mpmath.quad function might use either the Gauss-Legendre method, or the double exponential method (this latter being used by default). The method to use might be fixed with the functions mpmath.quadgl and mpmath.quadts.[2]

An important limitation of the mpmath integration functions within Sage is that they cannot manipulate arbitrary Sage expressions:

```
sage: mpmath.quad(sin(sin(x)), [0, 1])
Traceback (most recent call last):
...
TypeError: no canonical coercion from <type 'sage.libs.mpmath.ext_main.
    mpf'> to Symbolic Ring
```

The situation is however less dramatic than for PARI which is limited to its own syntax. It is indeed possible to define evaluation and conversion procedures to integrate via mpmath.quad arbitrary functions[3]:

```
sage: g(x) = max_symbolic(sin(x), cos(x))
sage: mpmath.mp.prec = 100
sage: mpmath.quadts(lambda x: g(N(x, 100)), [0, 1])
mpf('0.87391241626303543595797908625 2')
```

The integration of irregular functions (like the above I_4 example) might lead to a significant accuracy loss, even when asking for a large precision:

```
sage: mpmath.mp.prec = 170
sage: mpmath.quadts(lambda x: g(N(x, 190)), [0, 1])
mpf('0.873910907574009752053930059819624763440541483541887 94')
sage: N(sqrt(2) - cos(1), 100)
0.87391125650495533140075211677
```

We get only 5 correct digits here. We can nevertheless help mpmath by suggesting a subdivision of the interval domain (here at the irregular point, cf. Figure 14.5):

[2] Which recalls the name of the transform $\varphi : t \mapsto \tanh(\frac{\pi}{2}\sinh(t))$ which was seen above.

[3] The reader wondering why we used max_symbolic will try with max instead, and will look at the max_symbolic on-line help.

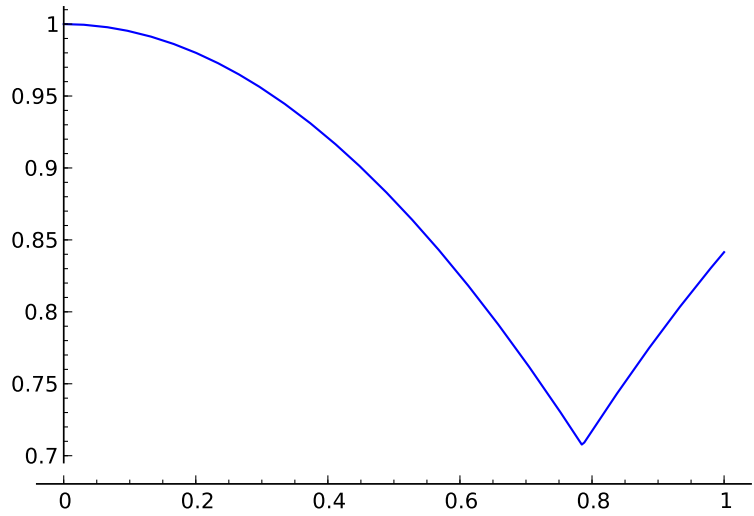


FIGURE 14.5 – The function $x \mapsto \max(\sin(x), \cos(x))$. The irregularity in $\pi/4$ renders numerical integration quite troublesome.

```
sage: mpmath.quadts(lambda x: g(N(x, 170)), [0, mpmath.pi / 4, 1])
mpf('0.87391125650495533140075211676672147483736145475902551')
```

The discontinuous functions (or those having a discontinuous derivative) are a classical "trap" for integration methods; however an automatic subdivision strategy, as described above, can limit the damage.

Exercise 48 (Computation of Newton-Cotes coefficients)**.** We want to compute the coefficients of the Newton-Cotes method with n points, which is not available within Sage. We consider for the sake of simplicity that the interval domain is $I = [0, n-1]$, the integration points being thus $x_1 = 0, x_2 = 1, \ldots, x_n = n-1$. The coefficients (w_i) of the quadrature rule are such that the equation

$$\int_0^{n-1} f(x)\,\mathrm{d}x = \sum_{i=0}^{n-1} w_i f(i) \tag{14.1}$$

is exact for any polynomial f of degree up to $n-1$.

1. We consider for $i \in \{0, \ldots, n-1\}$ the polynomial $P_i(X) = \prod_{\substack{j=0 \\ j \neq i}}^{n-1} (X - x_j)$. By applying Equation (14.1) to P_i, determine w_i in terms of P_i.
2. Deduce a function `NCRule` which associates to n the coefficients of Newton-Cotes' rule with n points on the interval $[0, n-1]$.
3. Show how to apply these weights to an interval $[a, b]$, with a, b any real numbers.
4. Write a function `QuadNC` which computes the integral of a function given on a segment of $\mathbb{R}$ given as parameter. Compare its results with the integration functions available in Sage on the integrals I_1 to I_{10}.

14.1.2 Multiple Integrals

Let us consider the double integral:

$$I = \int_0^1 \int_0^{\sqrt{y}} \exp(y \sin x)\, \mathrm{d}x\, \mathrm{d}y.$$

We first try to reduce it to a simple integral, by looking for a closed form for the inner integral, which fails in this case:

```
sage: y = var('y'); integrate(exp(y*sin(x)), (x, 0, sqrt(y)))
integrate(e^(y*sin(x)), x, 0, sqrt(y))
```

Since Sage does not provide any functionality for multiple integrals, we rewrite the problem as $I = \int_0^1 f(y)\, \mathrm{d}y$, where f is a Sage function, itself calling a numerical integration method:

```
sage: f = lambda y: numerical_integral(lambda x: exp(y*sin(x)),  \
                                       0, sqrt(y))[0]
sage: f(0.0), f(0.5), f(1.0)
(0.0, 0.8414895067661431, 1.6318696084180513)
```

We can now evaluate the integral of f on $[0, 1]$ numerically:

```
sage: numerical_integral(f, 0, 1)
(0.8606791942204567, 6.301207560882073e-07)
```

We might also use `sage.calculus.calculus.nintegral` to compute f:

```
sage: f = lambda y: sage.calculus.calculus.nintegral(exp(y*sin(x)), \
                                                     x, 0, sqrt(y))[0]
sage: numerical_integral(f, 0, 1)
(0.860679194220456..., 6.301207560882096e-07)
```

or even `mpmath.quad`:

```
sage: f = lambda y: RDF(mpmath.quad(lambda x: mpmath.exp(y*mpmath.sin(x)), \
                                    [0, sqrt(y)]))
sage: numerical_integral(f, 0, 1)
(0.8606791942204567, 6.301207561187562e-07)
```

Note that `mpmath` is able to compute multiple integrals directly, even in arbitrary precision, however only on a rectangular domain:

```
sage: mpmath.mp.dps = 60
sage: f = lambda x, y: mpmath.exp(y*mpmath.sin(x))
sage: mpmath.quad(f, [0,1], [0,1])
mpf('1.28392205755238471754385917646324675741664250325189751108716305')
```

Sometimes, in particular when the integrand is expensive to compute, we want an answer with at most (say) n^2 evaluations, even if we get a worse accuracy of the result. Here is a solution using the options `algorithm` and `max_points` of `numerical_integral`:

```
sage: def evalI(n):
....:   f = lambda y: numerical_integral(lambda x: exp(y*sin(x)),
....:                0, sqrt(y), algorithm='qng', max_points=n)[0]
....:   return numerical_integral(f, 0, 1, algorithm='qng', max_points=n)
sage: evalI(100)
(0.8606792028826138, 5.553962923506737e-07)
```

14.2 Solving Ordinary Differential Equations Numerically

In this section, we are interested in solving ordinary differential equations numerically. The available functions in Sage are able to deal with systems of the form:

$$\left\{\begin{array}{rcl} \frac{\mathrm{d}y_1}{\mathrm{d}t}(t) &=& f_1(t, y_1(t), y_2(t), \ldots, y_n(t)) \\ \frac{\mathrm{d}y_2}{\mathrm{d}t}(t) &=& f_2(t, y_1(t), y_2(t), \ldots, y_n(t)) \\ \vdots & & \\ \frac{\mathrm{d}y_n}{\mathrm{d}t}(t) &=& f_n(t, y_1(t), y_2(t), \ldots, y_n(t)) \end{array}\right.$$

with known initial conditions $(y_1(0), y_2(0), \ldots, y_n(0))$.

This setting enables us to solve also problems of order larger than 1, by introducing auxiliary variables (see the detailed example in §14.2.1). It does not however allow to represent the system of equations satisfied by Dickman's ρ function:

$$\left\{\begin{array}{rcll} u\rho'(u) + \rho(u-1) &=& 0 & \text{for } u \geqslant 1, \\ \rho(u) &=& 1 & \text{for } 0 \leqslant u \leqslant 1. \end{array}\right.$$

Indeed, the tools to solve ordinary differential equations are not suited to such an equation (called *with delay*).

The "one-step" numerical methods all use the same general principle: for a given step h and known values of $y(t_0)$ and $y'(t_0)$, we compute an approximation of $y(t_0+h)$ from a estimate of $y'(t)$ taken on the interval $[t_0, t_0+h]$. The simplest method consists in the approximation:

$$\begin{array}{rcl} \forall t \in [t_0, t_0+h], \quad y'(t) &\approx& y'(t_0), \\ \displaystyle\int_{t_0}^{t_0+h} y'(t)\,\mathrm{d}t &\approx& hy'(t_0), \\ y(t_0+h) &\approx& y(t_0) + hy'(t_0). \end{array}$$

The approximation of y' by a constant on $[t_0, t_0+h]$ reminds us the rectangle quadrature rule. The obtained method is of order 1, i.e., the error made after one computation step is $O(h^2)$, assuming f is regular enough. In general a method is of order p if the error made on a step of width h is $O(h^{p+1})$. The value obtained in $t_1 = t_0 + h$ is used as starting point to make one further step, until the desired target.

This order 1 method, called Euler's method, is not renowned for its accuracy (as the rectangle rule for numerical integration), and some higher order methods

exist, for example the Runge-Kutta method or order 2 to solve the equation $y' = f(t, y)$:

$$\begin{aligned} k_1 &= hf(t_n, y(t_n)) \\ k_2 &= hf(t_n + \frac{1}{2}h, y(t_n) + \frac{1}{2}k_1) \\ y(t_{n+1}) &\approx y(t_n) + k_2 + O(h^3). \end{aligned}$$

In this method, we try to evaluate $y'(t_n + h/2)$ to get a better estimate of $y(t_n + h)$.

Some multi-step methods also exist (for example Gear's method): they consist in computing $y(t_n)$ from the already computed values $y(t_{n-1}), y(t_{n-2}), \ldots, y(t_{n-\ell})$, for a given number ℓ of steps. These methods necessarily require an initial phase, before enough values are available.

Similarly to the Gauss-Kronrod quadrature method, some hybrid methods exist for solving differential equations. For example, the Dormand-Prince method computes with the same approximation points a value at orders 4 and 5, the latter being used to estimate the error made for the former. We say it is an adaptive method.

We also distinguish between explicit and implicit methods: in an explicit method, the value of $y(t_{n+1})$ is given by a formula using known values only; for an implicit method we have to solve an equation. Let us consider for example Euler's implicit method:

$$y(t_{n+1}) = y(t_n) + hf(t_{n+1}, y(t_{n+1})).$$

The wanted value $y(t_{n+1})$ appears on both sides of the equation; if the function f is complex enough, we have to solve a non-linear algebraic system, typically using Newton's method (see §12.2.2).

A priori, we expect more accurate results if we decrease the integration step h; in addition to the extra computations it implies, the expected accuracy gain is however counter-balanced by the increased roundoff errors which, at the end, might be significant with respect to the final result.

14.2.1 An Example

Let us consider the Van der Pol oscillator of parameter μ, satisfying the following differential equation:

$$\frac{\mathrm{d}^2x}{\mathrm{d}t^2}(t) - \mu(1 - x^2)\frac{\mathrm{d}x}{\mathrm{d}t}(t) + x(t) = 0.$$

Writing $y_0(t) = x(t)$ and $y_1(t) = \frac{\mathrm{d}x}{\mathrm{d}t}$, we get the order 1 system:

$$\left\{ \begin{array}{lcl} \frac{\mathrm{d}y_0}{\mathrm{d}t} &=& y_1, \\ \frac{\mathrm{d}y_1}{\mathrm{d}t} &=& \mu(1 - y_0^2)y_1 - y_0. \end{array} \right.$$

To solve it, we will use a "solver" object provided by the `ode_solver` command:

```
sage: T = ode_solver()
```

A solver object enables us to register the definition and the parameters of the system we want to solve; it gives access to the numerical tools for solving differential equations from the GSL library, already mentioned for numerical quadrature.

The system equations are given in the form of a function:

```
sage: def f_1(t,y,params): return [y[1],params[0]*(1-y[0]^2)*y[1]-y[0]]
sage: T.function = f_1
```

The parameter `y` represents the vector of unknown functions, and we should return the right-hand side vector of the system, in terms of t and an optional parameter (here `params[0]` which represents μ).

Some of the GSL algorithms require the system jacobian as well (the matrix whose (i, j) term is $\frac{\partial f_i}{\partial y_j}$, and whose last line contains $\frac{\partial f_j}{\partial t}$):

```
sage: def j_1(t,y,params):
....:     return [[0, 1],
....:             [-2*params[0]*y[0]*y[1]-1, params[0]*(1-y[0]^2)],
....:             [0,0]]
sage: T.jacobian = j_1
```

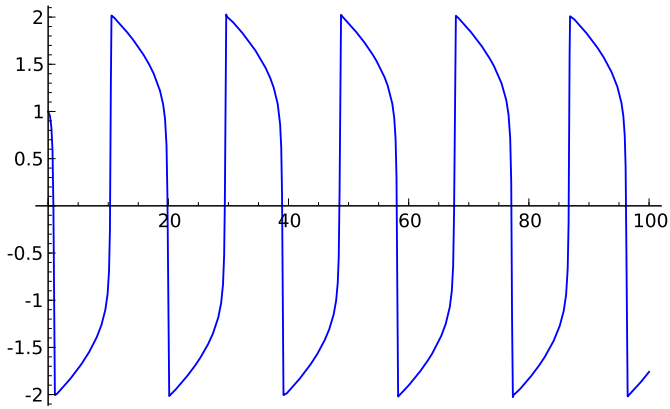
it is now possible to ask for a numerical solution. We choose the algorithm, the interval on which we want the solution, and the number of steps, which determines h:

```
sage: T.algorithm = "rk8pd"
sage: T.ode_solve(y_0=[1,0], t_span=[0,100], params=[10],
....:             num_points=1000)
sage: f = T.interpolate_solution()
```

Here, we have chosen the Runge-Kutta Dormand-Prince algorithm to compute the solution on $[0, 100]$; the initial conditions and the value of the parameters (here one only) are given too: `y_0=[1,0]` means $y_0(0) = 1, y_1(0) = 0$, i.e., $x(0) = 1, x'(0) = 0$.

To show the graph of the solution (we might try `plot(f, 0, 2)` to see the zero derivative in $t = 0$ more clearly):

```
sage: plot(f, 0, 100)
```



14.2.2 Available Functions

We have already mentioned for the solver objects from GSL the `rk8pd` method. Other methods are available:

`rkf45`: Runga-Kutta-Fehlberg, an adaptive method of orders 5 and 4;

`rk2`: adaptative Runge-Kutta of orders 3 and 2;

`rk4`: the classical Runge-Kutta method of order 4;

`rk2imp`: an implicit order 2 Runge-Kutta method with evaluation in the middle of the interval;

`rk4imp`: an implicit order 4 Runge-Kutta method with evaluation at "Gaussian points"[4];

`bsimp`: the implicit Burlisch-Stoer method;

`gear1`: the implicit one-step Gear method;

`gear2`: the implicit two-step Gear method.

For more details on all these methods, we refer the reader to [AP98] or [CM84].

One should note that the GSL limitation to machine floating-point numbers — thus of fixed precision — that we mentioned for the numerical integration also holds for solving differential equations.

Maxima also provides routines to solve differential equations numerically, with its own syntax:

```
sage: t, y = var('t, y')
sage: desolve_rk4(t*y*(2-y), y, ics=[0,1], end_points=[0, 1], step=0.5)
[[0, 1], [0.5, 1.12419127424558], [1.0, 1.461590162288825]]
```

The `desolve_rk4` function uses the order-4 Runge-Kutta method (the same as `rk4` for GSL) and takes as parameters:

- the right-hand side of the equation $y'(t) = f(t, y(t))$, here $y' = ty(2 - y)$;
- the name of the unknown function, here y;
- the initial conditions `ics`, here $t = 0$ and $y = 1$;
- the resolution interval `end_points`, here $[0, 1]$;
- the resolution step, here 0.5.

We omit the similar command `desolve_system_rk4`, already mentioned in Chapter 4, and which applies to a differential system. Maxima is limited to machine precision too.

If we want arbitrary precision solutions, we might use `odefun` from the `mpmath` package:

[4]The roots of the degree-2 Legendre polynomial, which are shifted on the interval $[t, t + h]$, and whose name makes reference to the Gauss-Legendre quadrature rule.

```
sage: import mpmath
sage: mpmath.mp.prec = 53
sage: sol = mpmath.odefun(lambda t, y: y, 0, 1)
sage: sol(1)
mpf('2.7182818284590451')
sage: mpmath.mp.prec = 100
sage: sol(1)
mpf('2.7182818284590452353602874802307')
sage: N(exp(1), 100)
2.7182818284590452353602874714
```

The arguments of the `mpmath.odefun` function are:

- the right-hand sides of the system of equations, in the form of a function $(t, y) \mapsto f(t, y(t))$, here $y' = y$, like for the `ode_solver` function. The dimension of the system is automatically deduced from the dimension of the function return value;
- the initial conditions t_0 and $y(t_0)$, here $y(0) = 1$.

For example for this two-dimensional system

$$\begin{cases} y_1' &= -y_2 \\ y_2' &= y_1 \end{cases}$$

whose solutions are $(\cos(t), \sin(t))$, with initial conditions $y_1(0) = 1$ and $y_2(0) = 0$:

```
sage: mpmath.mp.prec = 53
sage: f = mpmath.odefun(lambda t, y: [-y[1], y[0]], 0, [1, 0])
sage: f(3)
[mpf('-0.98999249660044542'), mpf('0.14112000805986721')]
sage: (cos(3.), sin(3.))
(-0.989992496600445, 0.141120008059867)
```

The `mpmath.odefun` function relies on Taylor's method. For degree p it uses:

$$y(t_{n+1}) = y(t_n) + h\frac{\mathrm{d}y}{\mathrm{d}t}(t_n) + \frac{h^2}{2!}\frac{\mathrm{d}^2y}{\mathrm{d}t^2}(t_n) + \ldots + \frac{h^p}{p!}\frac{\mathrm{d}^py}{\mathrm{d}t^p}(t_n) + O(h^{p+1}).$$

The main question is the computation of the derivatives of y. For this purpose, `odefun` computes approximate values

$$[\widetilde{y}(t_n + h), \ldots, \widetilde{y}(t_n + ph)] \approx [y(t_n + h), \ldots, y(t_n + ph)]$$

using p steps of the less precise method of Euler. We then compute

$$\widetilde{\frac{\mathrm{d}y}{\mathrm{d}t}}(t_n) \approx \frac{\widetilde{y}(t_n + h) - \widetilde{y}(t_n)}{h}, \quad \widetilde{\frac{\mathrm{d}y}{\mathrm{d}t}}(t_n + h) \approx \frac{\widetilde{y}(t_n + 2h) - \widetilde{y}(t_n + h)}{h}$$

then

$$\widetilde{\frac{\mathrm{d}^2y}{\mathrm{d}t^2}}(t_n) \approx \frac{\widetilde{\frac{\mathrm{d}y}{\mathrm{d}t}}(t_n + h) - \widetilde{\frac{\mathrm{d}y}{\mathrm{d}t}}(t_n)}{h},$$

and so on until we obtain estimates of the derivative of y in t_n up to order p.

Care must be taken when we change the floating-point precision of `mpmath`. To illustrate this problem, let us consider again the differential equation $y' = y$ seen above, satisfied by the exp function:

```
sage: mpmath.mp.prec = 10
sage: sol = mpmath.odefun(lambda t, y: y, 0, 1)
sage: sol(1)
mpf('2.7148')
sage: mpmath.mp.prec = 100
sage: sol(1)
mpf('2.7135204235459511323824699502438')
```

The last approximation of exp(1) is quite bad, albeit being computed with 100 bits of precision! The solution function `sol` (an "interpolator" in the `mpmath` jargon) has been computed with 10 bits of precision only, and its coefficients are not recomputed when the precision is changed, which explains the result.