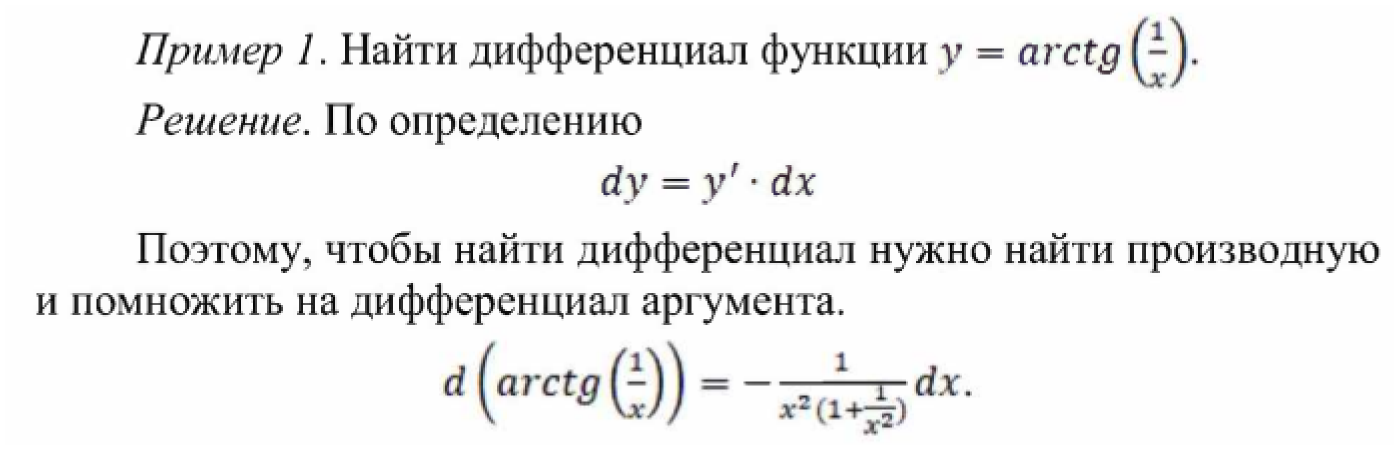
Интегралы

In [10]:
```python
from sympy import *
from scipy.integrate import quad
```



Пример 1. Найти дифференциал функции $y = arctg\left(\frac{1}{x}\right)$.

Решение. По определению

$$dy = y' \cdot dx$$

Поэтому, чтобы найти дифференциал нужно найти производную и помножить на дифференциал аргумента.

$$d\left(arctg\left(\frac{1}{x}\right)\right) = -\frac{1}{x^2\left(1+\frac{1}{x^2}\right)}dx.$$

In [11]:
```python
x = Symbol('x')
dx = Symbol('dx')
a = diff( atan(1/x), x)
print( dx*a )
```

```
-dx/(x**2*(1 + x**(-2)))
```

In [12]:
```python
x = Symbol('x')
dx = Symbol('dx')
y = Symbol('y')
xx = diff(sqrt(1+(sin(x))**2), x)
y = print( xx*dx )
```

```
dx*sin(x)*cos(x)/sqrt(sin(x)**2 + 1)
```

Пример 2. Найти неопределенный интеграл.

$$\int 6x^5 dx$$

```
In [13]:
x = symbols('x')
y = integrate(6*x**5, x)
print(y)
```

```
x**6
```

Пример 3.

$$\int \frac{x}{x+2} dx$$

```
In [14]:
x = symbols('x')
y = integrate(x/(x+2), x)
print(y)
```

```
x - 2*log(x + 2)
```

Пример 4.

$$\int \frac{1}{(x^2+1)^2} dx$$

```
In [15]:
integrate(1/(x**2+1)**2)
```

Out[15]:

$$\frac{x}{2x^2+2} + \frac{\operatorname{atan}(x)}{2}$$

Пример 5.

$$\int x e^{2x} dx$$

In [16]:
```
integrate(x*exp(2 *x),x)
```

Out[16]:
$$\frac{(2x-1)\,e^{2x}}{4}$$

Пример 6.

$$\int \frac{\sqrt{x+4}}{x}\,dx$$

In [17]:
```
integrate(sqrt(x+4)/x)
```

Out[17]:
$$\begin{cases} 2\sqrt{x+4} - 4\operatorname{acoth}\left(\frac{\sqrt{x+4}}{2}\right) & \text{for } \frac{|x+4|}{4} > 1 \\ 2\sqrt{x+4} - 4\operatorname{atanh}\left(\frac{\sqrt{x+4}}{2}\right) & \text{otherwise} \end{cases}$$

Пример 7.

$$\int_0^4 6x^5\,dx$$

In [18]:
```
integrate(6*x**5, (x,0,4))
```

Out[18]: 4096

Пример 8.

$$\int_1^3 \frac{x}{x+2}\,dx$$

In [19]:
```
integrate(x/(x+2),(x,1,3))
```

Out[19]: $-2\log(5) + 2 + 2\log(3)$

Пример 9.

$$\int_{-1}^{1} \frac{1}{(x^2+1)^2} dx$$

In [20]:
```
integrate(1/(x**2 + 1)**2,(x,-1,1) )
```

Out[20]: $\frac{1}{2} + \frac{\pi}{4}$

Пример 10.

$$\int_{0}^{100} xe^{2x} dx$$

In [21]:
```
integrate(x*exp(2*x),(x,0,100))
```

Out[21]: $\frac{1}{4} + \frac{199e^{200}}{4}$

Пример 11.

$$\int_{-1}^{0} \sqrt{x+4} dx$$

In [22]:
```
integrate(sqrt(x+4),(x,-1,0))
```

Out[22]: $\frac{16}{3} - 2\sqrt{3}$

Пример 12.

$$\int_{1}^{\infty} x^{-4} dx$$

In [23]:
```
integrate(x**(-4), (x, 1, oo))
```
Out[23]: $\frac{1}{3}$

Пример 13.

$$\int_{-1}^{\infty} e^{-2x}\,dx$$

In [24]:
```
integrate(exp(-2*x), (x, -1, oo))
```
Out[24]: $\frac{e^2}{2}$

Пример 14.

$$\int_{0}^{1} \ln x\,dx$$

In [25]:
```
integrate(log(x), (x, 0, 1))
```
Out[25]: -1

Пример 15.

$$\int_{0}^{7} \frac{1}{x^{\frac{6}{7}}}\,dx$$

In [26]:
```
integrate(1/x**(6/7), (x, 0,7 ))
```
Out[26]: 9.24328473429286

Пример 16. Найти

$$\iint (y^2 \cdot x - 2 \cdot x \cdot y)dxdy, \text{где } x \le y \le 2, -1 \le x \le 2.$$

Решение. Сначала найдем интеграл по y от x до 2:

integrate (f(x, y), (y, x, 2)),

потом по x от -1 до 2.

In [27]:
```
y = symbols('y')
integrate(y**2*x-2*x*y,(y,x,2))
```

Out[27]:

$$-\frac{x^4}{3} + x^3 - \frac{4x}{3}$$

In [28]:
```
integrate(-x**4/3 + x**3 - 4*x/3,(x,-1,2))
```

Out[28]:

$$-\frac{9}{20}$$

Пример 17. Найти площадь фигуры, ограниченной линиями $y = 2x,\ y = -x^2 + 7x - 6$.

In [29]:
```
integrate(-x**2+7*x-6-2*x,(x,2,3))
```

Out[29]:

$$\frac{1}{6}$$

Пример 18. Найти площадь фигуры, ограниченной линиями $y = -2x,\ y = -x^2 + 5x - 10$.

In [30]:
```
integrate(-x**2+5*x-10+2*x,(x,2,5))
```

```
Out[30]:
```
$$\frac{9}{2}$$

Пример 19. Найти площадь фигуры, ограниченной линиями

$$y = -2x,\ y = -x^2 + 3x - 6.$$

```
In [31]: integrate(-x**2+3*x-6+2*x,(x,2,3))
```

```
Out[31]:
```
$$\frac{1}{6}$$

Пример 20. Вычислите объём тела, образованного вращением вокруг оси Ox области, ограниченной линиями

$$y = x^2 - x \quad \text{и} \quad y = 0 \quad \text{при } x \in [2,4].$$

```
In [32]: pi*integrate((x**2-x)**2,(x,2,4))
```

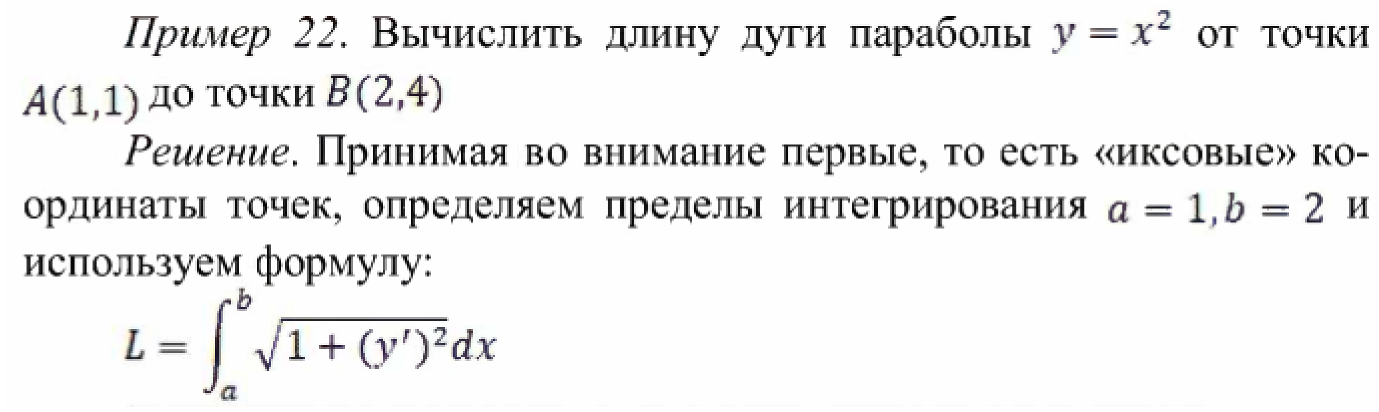
```
Out[32]:
```
$$\frac{1456\pi}{15}$$

Пример 21. Вычислите объём тела, образованного вращением вокруг оси Ox области, ограниченной линиями

$$y = \sqrt{3-x} \quad \text{и} \quad y = -x - 53 \quad \text{при } x \in [-61,-53].$$

```
In [33]: pi*integrate(((sqrt(3-x)) **2-(-x-53)**2),(x,-61,-53))
```

```
Out[33]:
```
$$\frac{928\pi}{3}$$



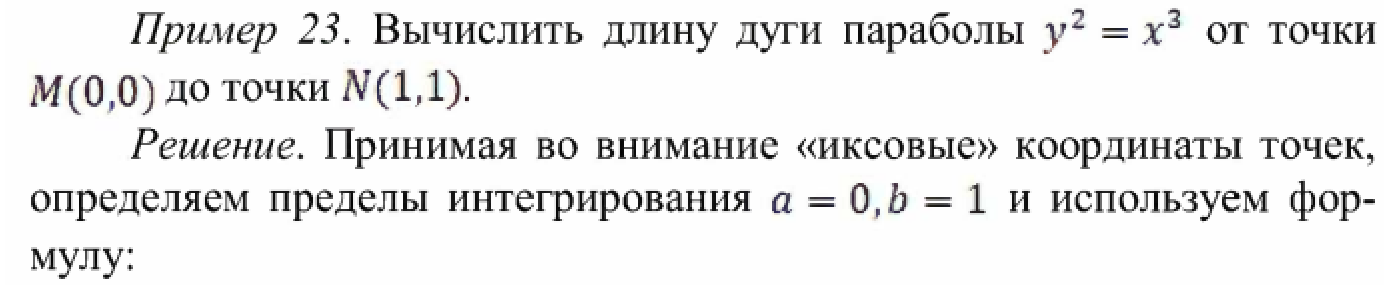
Пример 22. Вычислить длину дуги параболы $y = x^2$ от точки $A(1,1)$ до точки $B(2,4)$

Решение. Принимая во внимание первые, то есть «иксовые» координаты точек, определяем пределы интегрирования $a = 1, b = 2$ и используем формулу:

$$L = \int_a^b \sqrt{1 + (y')^2}dx$$

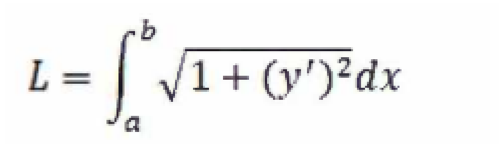
```
In [34]: integrate(sqrt(1+diff(x**2)**2),(x,1,2))
```

Out[34]: $- \frac{\sqrt{5}}{2} - \frac{\operatorname{asinh}{\left(2 \right)}}{4} + \frac{\operatorname{asinh}{\left(4 \right)}}{4} + \sqrt{17}$



Пример 23. Вычислить длину дуги параболы $y^2 = x^3$ от точки $M(0,0)$ до точки $N(1,1)$.

Решение. Принимая во внимание «иксовые» координаты точек, определяем пределы интегрирования $a = 0, b = 1$ и используем формулу:



$$L = \int_a^b \sqrt{1 + (y')^2}dx$$

```
In [35]: integrate(sqrt(1+diff(pow(x,3/2))**2),(x,0,1))
```

```
Out[35]: 1.43970987337155
```

Пример 24. Найдите функцию дохода $R(x)$, если предельный доход при реализации единиц продукции определяется по формуле $MR = 6x^6 - 230$.

$$R(x) = \int (6x^6 - 230)dx = \frac{6x^7}{7} - 230x$$

In [36]:
```
integrate(6*x**6-230,x)
```

Out[36]:
$$\frac{6x^7}{7} - 230x$$

Пример 24. Найти функцию издержек $TC(q)$, если предельные издержки заданы функцией $MC = 18q^5 + 20q^4 + 16q^3$, а начальные фиксированные затраты равны 790.

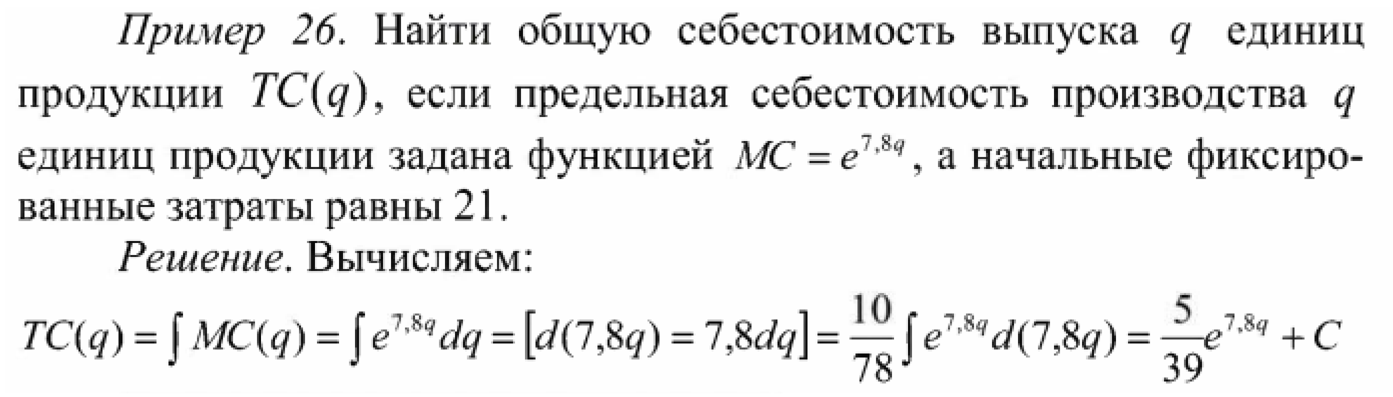
$TC(q) = \int MC(q)dq = \int (18q^5 + 20q^4 + 16q^3)dq = 18q^6/6 + 20q^5/5 + 16q^4/4$

По условию $TC(0) = 790$. Следовательно,

$$\frac{18(0)^6}{6} + \frac{20(0)^5}{5} + \frac{16(0)^4}{4} + C = 790,$$

In [37]:
```
integrate(18*x**5+20*x**4+17*x**3,x)
```

Out[37]:
$$3x^6 + 4x^5 + \frac{17x^4}{4}$$



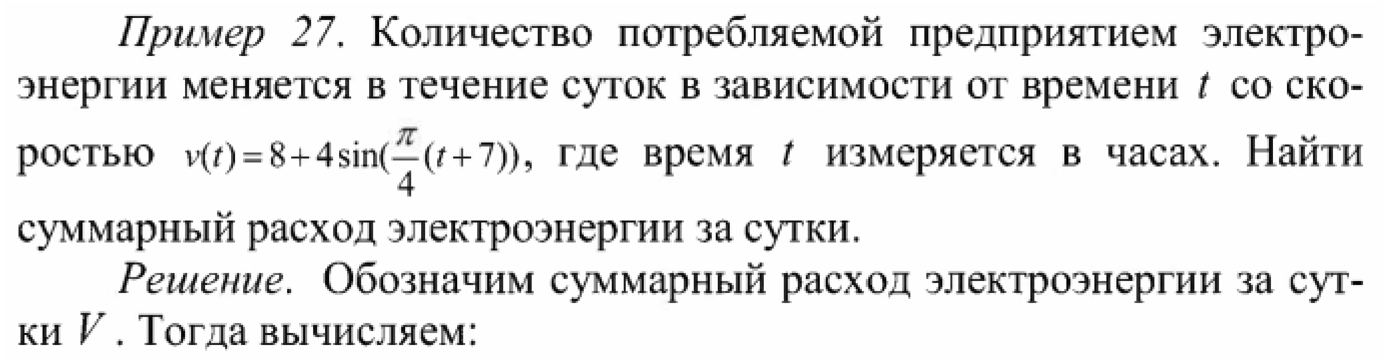
Пример 26. Найти общую себестоимость выпуска q единиц продукции $TC(q)$, если предельная себестоимость производства q единиц продукции задана функцией $MC = e^{7,8q}$, а начальные фиксированные затраты равны 21.

Решение. Вычисляем:

$$TC(q) = \int MC(q) = \int e^{7,8q}dq = [d(7,8q) = 7,8dq] = \frac{10}{78}\int e^{7,8q}d(7,8q) = \frac{5}{39}e^{7,8q} + C$$

In [38]:
```
integrate(exp(7.8*x),x)
```

Out[38]: $0.128205128205128e^{7.8x}$



Пример 27. Количество потребляемой предприятием электроэнергии меняется в течение суток в зависимости от времени t со скоростью $v(t) = 8 + 4\sin(\frac{\pi}{4}(t+7))$, где время t измеряется в часах. Найти суммарный расход электроэнергии за сутки.

Решение. Обозначим суммарный расход электроэнергии за сутки V. Тогда вычисляем:

In [39]:
```
integrate(8+4*sin(pi/4*(x+7)),(x,0,24))
```

Out[39]: 192

Пример 28. Найти объем продукции, произведений за 6 лет, если функция Кобба – Дугласа имеет вид: $F(t) = (1+t)e^{2t}$.

Решение. Объем $V(t)$ произведенной продукции вычисляется по формуле:

$$V(t) = \int_0^6 (1+t)e^{2t}\,dt.$$

In [40]:
```
integrate((1+x)*exp(2*x),(x,0,6))
```

Out[40]:
$$-\frac{1}{4} + \frac{13e^{12}}{4}$$

Примеры решения задач

Найдите неопределенный интеграл $\int 6\sin^2\left(\frac{x}{2}\right)dx$.

In [41]:
```
integrate(6*sin(x/2)**2,x)
```

Out[41]:
$$3x - 6\sin\left(\frac{x}{2}\right)\cos\left(\frac{x}{2}\right)$$

Вычислите интеграл $\int (4x+3)^2 dx$.

In [42]:
```
integrate((4*x+3)**2,x)
```

Out[42]:
$$\frac{16x^3}{3} + 12x^2 + 9x$$

Найдите неопределенный интеграл $\int \frac{dx}{-x^2-8x-12}$.

```
In [43]: integrate(1/((-x**2-8*x-12)),x)
```

Out[43]:

$$-\frac{\log(x+2)}{4}+\frac{\log(x+6)}{4}$$

Найдите неопределенный интеграл $\int \mathrm{tg}2x dx$.

```
In [44]: integrate(tan(2*x),x)
```

Out[44]:

$$-\frac{\log(\cos(2x))}{2}$$

Найдите определенный интеграл $\int_2^3 x(28-3x^2)^{\frac{1}{5}}dx$.

```
In [45]: integrate(x*(28-3*x**2)**(1/5),(x,2,3))
```

Out[45]:

$$\int\limits_2^3 \begin{cases} -0.999999999999999x\left(3x^2-28\right)^{0.2}e^{1.2i\pi} & \text{for } \frac{3x^2}{28}>1 \\ 0.999999999999999x\left(28-3x^2\right)^{0.2} & \text{otherwise} \end{cases}\,dx$$

Найдите определенный интеграл $\int_{10^{-7}}^{1} \frac{\lg x}{x} dx$.

```
In [46]: integrate(log(x,10)/x,(x,10**(-7),1))
```

Out[46]:

$$-\frac{129.896503706721}{\log(10)}$$

Найдите несобственный интеграл или установите его расходимость $\int_{-30}^{+\infty} \frac{dx}{x^2+10x+50}$.

In [47]:
```
integrate(1/(x**2+10*x+50),(x,-30,oo))
```

Out[47]: $\frac{\operatorname{atan}(5)}{5} + \frac{\pi}{10}$

Найти площадь фигуры, ограниченной линиями $y = 5x, y = 3x^2 - 9x + 15$.

In [48]:
```
solve(5*x-(3*x**2-9*x+15),x)
```

Out[48]:
```
[5/3, 3]
```

In [49]:
```
abs(integrate(5*x- (3*x**2-9*x+15), (x, 5/3,3)))
```

Out[49]: 1.18518518518518

Вычислить кратный интеграл $\iint (3y^2x + 7xy)dxdy$ по области

$$D = \{(x, y)\epsilon\mathbb{R}|-3 \le x \le -2, -x \le y \le 2\}.$$

In [50]:
```
x, y = symbols("x y")
f = (3*y ** 2*x+7*x*y)
integrate(f, (y, -x, 2), (x, -3, -2))
```

Out[50]: $\frac{1763}{40}$

Индивидуальное задание

Интеграция

Подпакет scipy.integrate предоставляет несколько методов интегрирования, включая интегратор обыкновенных дифференциальных уравнений. Обзор модуля предоставляется командой help:

In [60]:
```
help(integrate)
```

```
Help on package scipy.integrate in scipy:

NAME
    scipy.integrate

DESCRIPTION
    =============================================
    Integration and ODEs (:mod:`scipy.integrate`)
    =============================================

    .. currentmodule:: scipy.integrate

    Integrating functions, given function object
    ============================================

    .. autosummary::
       :toctree: generated/

       quad          -- General purpose integration
       quad_vec      -- General purpose integration of vector-valued functions
       dblquad       -- General purpose double integration
       tplquad       -- General purpose triple integration
       nquad         -- General purpose N-D integration
       fixed_quad    -- Integrate func(x) using Gaussian quadrature of order n
       quadrature    -- Integrate with given tolerance using Gaussian quadrature
       romberg       -- Integrate func using Romberg integration
       quad_explain  -- Print information for use of quad
       newton_cotes  -- Weights and error coefficient for Newton-Cotes integration
       IntegrationWarning -- Warning on issues during integration
       AccuracyWarning  -- Warning on issues during quadrature integration

    Integrating functions, given fixed samples
    ==========================================

    .. autosummary::
       :toctree: generated/
```

```
   trapezoid             -- Use trapezoidal rule to compute integral.
   cumulative_trapezoid -- Use trapezoidal rule to cumulatively compute integral.
   simpson               -- Use Simpson's rule to compute integral from samples.
   romb                  -- Use Romberg Integration to compute integral from
                         -- (2**k + 1) evenly-spaced samples.

.. seealso::

   :mod:`scipy.special` for orthogonal polynomials (special) for Gaussian
   quadrature roots and weights for other weighting factors and regions.

Solving initial value problems for ODE systems
==============================================

The solvers are implemented as individual classes, which can be used directly
(low-level usage) or through a convenience function.

.. autosummary::
   :toctree: generated/

   solve_ivp     -- Convenient function for ODE integration.
   RK23          -- Explicit Runge-Kutta solver of order 3(2).
   RK45          -- Explicit Runge-Kutta solver of order 5(4).
   DOP853        -- Explicit Runge-Kutta solver of order 8.
   Radau         -- Implicit Runge-Kutta solver of order 5.
   BDF           -- Implicit multi-step variable order (1 to 5) solver.
   LSODA         -- LSODA solver from ODEPACK Fortran package.
   OdeSolver     -- Base class for ODE solvers.
   DenseOutput   -- Local interpolant for computing a dense output.
   OdeSolution   -- Class which represents a continuous ODE solution.


Old API
-------

These are the routines developed earlier for SciPy. They wrap older solvers
implemented in Fortran (mostly ODEPACK). While the interface to them is not
particularly convenient and certain features are missing compared to the new
API, the solvers themselves are of good quality and work fast as compiled
Fortran code. In some cases, it might be worth using this old API.

.. autosummary::
   :toctree: generated/

   odeint        -- General integration of ordinary differential equations.
   ode           -- Integrate ODE using VODE and ZVODE routines.
```

```
       complex_ode   -- Convert a complex-valued ODE to real-valued and integrate.


    Solving boundary value problems for ODE systems
    ===============================================

    .. autosummary::
       :toctree: generated/

       solve_bvp     -- Solve a boundary value problem for a system of ODEs.

PACKAGE CONTENTS
    _bvp
    _dop
    _ivp (package)
    _ode
    _odepack
    _quad_vec
    _quadpack
    _quadrature
    _test_multivariate
    _test_odeint_banded
    lsoda
    odepack
    quadpack
    setup
    tests (package)
    vode

CLASSES
    builtins.UserWarning(builtins.Warning)
        scipy.integrate.quadpack.IntegrationWarning
    builtins.Warning(builtins.Exception)
        scipy.integrate._quadrature.AccuracyWarning
    builtins.object
        scipy.integrate._ivp.base.DenseOutput
        scipy.integrate._ivp.base.OdeSolver
            scipy.integrate._ivp.bdf.BDF
            scipy.integrate._ivp.lsoda.LSODA
            scipy.integrate._ivp.radau.Radau
        scipy.integrate._ivp.common.OdeSolution
        scipy.integrate._ode.ode
            scipy.integrate._ode.complex_ode
    scipy.integrate._ivp.rk.RungeKutta(scipy.integrate._ivp.base.OdeSolver)
        scipy.integrate._ivp.rk.DOP853
        scipy.integrate._ivp.rk.RK23
        scipy.integrate._ivp.rk.RK45
```

```
class AccuracyWarning(builtins.Warning)
 |  Base class for warning categories.
 |
 |  Method resolution order:
 |      AccuracyWarning
 |      builtins.Warning
 |      builtins.Exception
 |      builtins.BaseException
 |      builtins.object
 |
 |  Data descriptors defined here:
 |
 |  __weakref__
 |      list of weak references to the object (if defined)
 |
 |  ----------------------------------------------------------------------
 |  Methods inherited from builtins.Warning:
 |
 |  __init__(self, /, *args, **kwargs)
 |      Initialize self.  See help(type(self)) for accurate signature.
 |
 |  ----------------------------------------------------------------------
 |  Static methods inherited from builtins.Warning:
 |
 |  __new__(*args, **kwargs) from builtins.type
 |      Create and return a new object.  See help(type) for accurate signature.
 |
 |  ----------------------------------------------------------------------
 |  Methods inherited from builtins.BaseException:
 |
 |  __delattr__(self, name, /)
 |      Implement delattr(self, name).
 |
 |  __getattribute__(self, name, /)
 |      Return getattr(self, name).
 |
 |  __reduce__(...)
 |      Helper for pickle.
 |
 |  __repr__(self, /)
 |      Return repr(self).
 |
 |  __setattr__(self, name, value, /)
 |      Implement setattr(self, name, value).
 |
 |  __setstate__(...)
```

```
     |
     |  __str__(self, /)
     |      Return str(self).
     |
     |  with_traceback(...)
     |      Exception.with_traceback(tb) --
     |      set self.__traceback__ to tb and return self.
     |
     |  ----------------------------------------------------------------------
     |  Data descriptors inherited from builtins.BaseException:
     |
     |  __cause__
     |      exception cause
     |
     |  __context__
     |      exception context
     |
     |  __dict__
     |
     |  __suppress_context__
     |
     |  __traceback__
     |
     |  args

    class BDF(scipy.integrate._ivp.base.OdeSolver)
     |  BDF(fun, t0, y0, t_bound, max_step=inf, rtol=0.001, atol=1e-06, jac=None, jac_sparsity=None, vectorized=False, first_step=
None, **extraneous)
     |
     |  Implicit method based on backward-differentiation formulas.
     |
     |  This is a variable order method with the order varying automatically from
     |  1 to 5. The general framework of the BDF algorithm is described in [1]_.
     |  This class implements a quasi-constant step size as explained in [2]_.
     |  The error estimation strategy for the constant-step BDF is derived in [3]_.
     |  An accuracy enhancement using modified formulas (NDF) [2]_ is also implemented.
     |
     |  Can be applied in the complex domain.
     |
     |  Parameters
     |  ----------
     |  fun : callable
     |      Right-hand side of the system. The calling signature is ``fun(t, y)``.
     |      Here ``t`` is a scalar, and there are two options for the ndarray ``y``:
     |      It can either have shape (n,); then ``fun`` must return array_like with
     |      shape (n,). Alternatively it can have shape (n, k); then ``fun``
     |      must return an array_like with shape (n, k), i.e. each column
```

```
 |      corresponds to a single column in ``y``. The choice between the two
 |      options is determined by `vectorized` argument (see below). The
 |      vectorized implementation allows a faster approximation of the Jacobian
 |      by finite differences (required for this solver).
 |  t0 : float
 |      Initial time.
 |  y0 : array_like, shape (n,)
 |      Initial state.
 |  t_bound : float
 |      Boundary time - the integration won't continue beyond it. It also
 |      determines the direction of the integration.
 |  first_step : float or None, optional
 |      Initial step size. Default is ``None`` which means that the algorithm
 |      should choose.
 |  max_step : float, optional
 |      Maximum allowed step size. Default is np.inf, i.e., the step size is not
 |      bounded and determined solely by the solver.
 |  rtol, atol : float and array_like, optional
 |      Relative and absolute tolerances. The solver keeps the local error
 |      estimates less than ``atol + rtol * abs(y)``. Here `rtol` controls a
 |      relative accuracy (number of correct digits). But if a component of `y`
 |      is approximately below `atol`, the error only needs to fall within
 |      the same `atol` threshold, and the number of correct digits is not
 |      guaranteed. If components of y have different scales, it might be
 |      beneficial to set different `atol` values for different components by
 |      passing array_like with shape (n,) for `atol`. Default values are
 |      1e-3 for `rtol` and 1e-6 for `atol`.
 |  jac : {None, array_like, sparse_matrix, callable}, optional
 |      Jacobian matrix of the right-hand side of the system with respect to y,
 |      required by this method. The Jacobian matrix has shape (n, n) and its
 |      element (i, j) is equal to ``d f_i / d y_j``.
 |      There are three ways to define the Jacobian:
 |
 |          * If array_like or sparse_matrix, the Jacobian is assumed to
 |            be constant.
 |          * If callable, the Jacobian is assumed to depend on both
 |            t and y; it will be called as ``jac(t, y)`` as necessary.
 |            For the 'Radau' and 'BDF' methods, the return value might be a
 |            sparse matrix.
 |          * If None (default), the Jacobian will be approximated by
 |            finite differences.
 |
 |      It is generally recommended to provide the Jacobian rather than
 |      relying on a finite-difference approximation.
 |  jac_sparsity : {None, array_like, sparse matrix}, optional
 |      Defines a sparsity structure of the Jacobian matrix for a
 |      finite-difference approximation. Its shape must be (n, n). This argument
 |          is ignored if `jac` is not `None`. If the Jacobian has only few non-zero
 |          elements in *each* row, providing the sparsity structure will greatly
 |          speed up the computations [4]_. A zero entry means that a corresponding
 |          element in the Jacobian is always zero. If None (default), the Jacobian
 |          is assumed to be dense.
 |      vectorized : bool, optional
 |          Whether `fun` is implemented in a vectorized fashion. Default is False.
 |
 |      Attributes
 |      ----------
 |      n : int
 |          Number of equations.
 |      status : string
 |          Current status of the solver: 'running', 'finished' or 'failed'.
 |      t_bound : float
 |          Boundary time.
 |      direction : float
 |          Integration direction: +1 or -1.
 |      t : float
 |          Current time.
 |      y : ndarray
 |          Current state.
 |      t_old : float
 |          Previous time. None if no steps were made yet.
 |      step_size : float
 |          Size of the last successful step. None if no steps were made yet.
 |      nfev : int
 |          Number of evaluations of the right-hand side.
 |      njev : int
 |          Number of evaluations of the Jacobian.
 |      nlu : int
 |          Number of LU decompositions.
 |
 |      References
 |      ----------
 |      .. [1] G. D. Byrne, A. C. Hindmarsh, "A Polyalgorithm for the Numerical
 |             Solution of Ordinary Differential Equations", ACM Transactions on
 |             Mathematical Software, Vol. 1, No. 1, pp. 71-96, March 1975.
 |      .. [2] L. F. Shampine, M. W. Reichelt, "THE MATLAB ODE SUITE", SIAM J. SCI.
 |             COMPUTE., Vol. 18, No. 1, pp. 1-22, January 1997.
 |      .. [3] E. Hairer, G. Wanner, "Solving Ordinary Differential Equations I:
 |             Nonstiff Problems", Sec. III.2.
 |      .. [4] A. Curtis, M. J. D. Powell, and J. Reid, "On the estimation of
 |             sparse Jacobian matrices", Journal of the Institute of Mathematics
 |             and its Applications, 13, pp. 117-120, 1974.
 |
 |      Method resolution order:
     |      BDF
     |      scipy.integrate._ivp.base.OdeSolver
     |      builtins.object
     |
     |  Methods defined here:
     |
     |  __init__(self, fun, t0, y0, t_bound, max_step=inf, rtol=0.001, atol=1e-06, jac=None, jac_sparsity=None, vectorized=False,
first_step=None, **extraneous)
     |      Initialize self.  See help(type(self)) for accurate signature.
     |
     |  ----------------------------------------------------------------------
     |  Methods inherited from scipy.integrate._ivp.base.OdeSolver:
     |
     |  dense_output(self)
     |      Compute a local interpolant over the last successful step.
     |
     |      Returns
     |      -------
     |      sol : `DenseOutput`
     |          Local interpolant over the last successful step.
     |
     |  step(self)
     |      Perform one integration step.
     |
     |      Returns
     |      -------
     |      message : string or None
     |          Report from the solver. Typically a reason for a failure if
     |          `self.status` is 'failed' after the step was taken or None
     |          otherwise.
     |
     |  ----------------------------------------------------------------------
     |  Readonly properties inherited from scipy.integrate._ivp.base.OdeSolver:
     |
     |  step_size
     |
     |  ----------------------------------------------------------------------
     |  Data descriptors inherited from scipy.integrate._ivp.base.OdeSolver:
     |
     |  __dict__
     |      dictionary for instance variables (if defined)
     |
     |  __weakref__
     |      list of weak references to the object (if defined)
     |
     |  ----------------------------------------------------------------------
     |  Data and other attributes inherited from scipy.integrate._ivp.base.OdeSolver:
```

```
 |
 |  TOO_SMALL_STEP = 'Required step size is less than spacing between numb...

class DOP853(RungeKutta)
 |  DOP853(fun, t0, y0, t_bound, max_step=inf, rtol=0.001, atol=1e-06, vectorized=False, first_step=None, **extraneous)
 |
 |  Explicit Runge-Kutta method of order 8.
 |
 |  This is a Python implementation of "DOP853" algorithm originally written
 |  in Fortran [1]_, [2]_. Note that this is not a literate translation, but
 |  the algorithmic core and coefficients are the same.
 |
 |  Can be applied in the complex domain.
 |
 |  Parameters
 |  ----------
 |  fun : callable
 |      Right-hand side of the system. The calling signature is ``fun(t, y)``.
 |      Here, ``t`` is a scalar, and there are two options for the ndarray ``y``:
 |      It can either have shape (n,); then ``fun`` must return array_like with
 |      shape (n,). Alternatively it can have shape (n, k); then ``fun``
 |      must return an array_like with shape (n, k), i.e. each column
 |      corresponds to a single column in ``y``. The choice between the two
 |      options is determined by `vectorized` argument (see below).
 |  t0 : float
 |      Initial time.
 |  y0 : array_like, shape (n,)
 |      Initial state.
 |  t_bound : float
 |      Boundary time - the integration won't continue beyond it. It also
 |      determines the direction of the integration.
 |  first_step : float or None, optional
 |      Initial step size. Default is ``None`` which means that the algorithm
 |      should choose.
 |  max_step : float, optional
 |      Maximum allowed step size. Default is np.inf, i.e. the step size is not
 |      bounded and determined solely by the solver.
 |  rtol, atol : float and array_like, optional
 |      Relative and absolute tolerances. The solver keeps the local error
 |      estimates less than ``atol + rtol * abs(y)``. Here `rtol` controls a
 |      relative accuracy (number of correct digits). But if a component of `y`
 |      is approximately below `atol`, the error only needs to fall within
 |      the same `atol` threshold, and the number of correct digits is not
 |      guaranteed. If components of y have different scales, it might be
 |      beneficial to set different `atol` values for different components by
 |      passing array_like with shape (n,) for `atol`. Default values are
 |      1e-3 for `rtol` and 1e-6 for `atol`.
 |  vectorized : bool, optional
 |      Whether `fun` is implemented in a vectorized fashion. Default is False.
 |
 |  Attributes
 |  ----------
 |  n : int
 |      Number of equations.
 |  status : string
 |      Current status of the solver: 'running', 'finished' or 'failed'.
 |  t_bound : float
 |      Boundary time.
 |  direction : float
 |      Integration direction: +1 or -1.
 |  t : float
 |      Current time.
 |  y : ndarray
 |      Current state.
 |  t_old : float
 |      Previous time. None if no steps were made yet.
 |  step_size : float
 |      Size of the last successful step. None if no steps were made yet.
 |  nfev : int
 |      Number evaluations of the system's right-hand side.
 |  njev : int
 |      Number of evaluations of the Jacobian. Is always 0 for this solver
 |      as it does not use the Jacobian.
 |  nlu : int
 |      Number of LU decompositions. Is always 0 for this solver.
 |
 |  References
 |  ----------
 |  .. [1] E. Hairer, S. P. Norsett G. Wanner, "Solving Ordinary Differential
 |         Equations I: Nonstiff Problems", Sec. II.
 |  .. [2] `Page with original Fortran code of DOP853
 |          <http://www.unige.ch/~hairer/software.html>`_.
 |
 |  Method resolution order:
 |      DOP853
 |      RungeKutta
 |      scipy.integrate._ivp.base.OdeSolver
 |      builtins.object
 |
 |  Methods defined here:
 |
 |  __init__(self, fun, t0, y0, t_bound, max_step=inf, rtol=0.001, atol=1e-06, vectorized=False, first_step=None, **extraneou
s)
 |      Initialize self.  See help(type(self)) for accurate signature.
 |  
 |  ----------------------------------------------------------------------
 |  Data and other attributes defined here:
 |  
 |  A = array([[ 0.00000000e+00,  0.00000000e+00,  0.000...23605672e+01,  ...
 |  
 |  A_EXTRA = array([[ 5.61675023e-02,  0.00000000e+00,  0.000...e+00, -9....
 |  
 |  B = array([ 0.05429373,  0.        ,  0.        ,  0...7, -0.15216095,...
 |  
 |  C = array([0.        , 0.05260015, 0.07890023, 0.118...8205, 0.6      ...
 |  
 |  C_EXTRA = array([0.1       , 0.2       , 0.77777778])
 |  
 |  D = array([[-8.42893828e+00,  0.00000000e+00,  0.000...e+01, -3.917726...
 |  
 |  E3 = array([-0.18980075,  0.        ,  0.        ,  0...5,
 |          0.2...
 |  
 |  E5 = array([ 0.01312004,  0.        ,  0.        ,  0...2,
 |          0.0...
 |  
 |  error_estimator_order = 7
 |  
 |  n_stages = 12
 |  
 |  order = 8
 |  
 |  ----------------------------------------------------------------------
 |  Data and other attributes inherited from RungeKutta:
 |  
 |  E = NotImplemented
 |  
 |  P = NotImplemented
 |  
 |  ----------------------------------------------------------------------
 |  Methods inherited from scipy.integrate._ivp.base.OdeSolver:
 |  
 |  dense_output(self)
 |      Compute a local interpolant over the last successful step.
 |      
 |      Returns
 |      -------
 |      sol : `DenseOutput`
 |          Local interpolant over the last successful step.
 |  
 |  step(self)
 |      Perform one integration step.
 |
 |      Returns
 |      -------
 |      message : string or None
 |          Report from the solver. Typically a reason for a failure if
 |          `self.status` is 'failed' after the step was taken or None
 |          otherwise.
 |
 |  ----------------------------------------------------------------------
 |  Readonly properties inherited from scipy.integrate._ivp.base.OdeSolver:
 |
 |  step_size
 |
 |  ----------------------------------------------------------------------
 |  Data descriptors inherited from scipy.integrate._ivp.base.OdeSolver:
 |
 |  __dict__
 |      dictionary for instance variables (if defined)
 |
 |  __weakref__
 |      list of weak references to the object (if defined)
 |
 |  ----------------------------------------------------------------------
 |  Data and other attributes inherited from scipy.integrate._ivp.base.OdeSolver:
 |
 |  TOO_SMALL_STEP = 'Required step size is less than spacing between numb...

class DenseOutput(builtins.object)
 |  DenseOutput(t_old, t)
 |
 |  Base class for local interpolant over step made by an ODE solver.
 |
 |  It interpolates between `t_min` and `t_max` (see Attributes below).
 |  Evaluation outside this interval is not forbidden, but the accuracy is not
 |  guaranteed.
 |
 |  Attributes
 |  ----------
 |  t_min, t_max : float
 |      Time range of the interpolation.
 |
 |  Methods defined here:
 |
 |  __call__(self, t)
 |      Evaluate the interpolant.
 |
 |      Parameters
 |      ----------
 |      t : float or array_like with shape (n_points,)
 |          Points to evaluate the solution at.
 |
 |      Returns
 |      -------
 |      y : ndarray, shape (n,) or (n, n_points)
 |          Computed values. Shape depends on whether `t` was a scalar or a
 |          1-D array.
 |
 |  __init__(self, t_old, t)
 |      Initialize self.  See help(type(self)) for accurate signature.
 |
 |  ----------------------------------------------------------------------
 |  Data descriptors defined here:
 |
 |  __dict__
 |      dictionary for instance variables (if defined)
 |
 |  __weakref__
 |      list of weak references to the object (if defined)

class IntegrationWarning(builtins.UserWarning)
 |  Warning on issues during integration.
 |
 |  Method resolution order:
 |      IntegrationWarning
 |      builtins.UserWarning
 |      builtins.Warning
 |      builtins.Exception
 |      builtins.BaseException
 |      builtins.object
 |
 |  Data descriptors defined here:
 |
 |  __weakref__
 |      list of weak references to the object (if defined)
 |
 |  ----------------------------------------------------------------------
 |  Methods inherited from builtins.UserWarning:
 |
 |  __init__(self, /, *args, **kwargs)
 |      Initialize self.  See help(type(self)) for accurate signature.
 |
 |  ----------------------------------------------------------------------
 |  Static methods inherited from builtins.UserWarning:
```

```
 |
 |  __new__(*args, **kwargs) from builtins.type
 |      Create and return a new object.  See help(type) for accurate signature.
 |
 |  ----------------------------------------------------------------------
 |  Methods inherited from builtins.BaseException:
 |
 |  __delattr__(self, name, /)
 |      Implement delattr(self, name).
 |
 |  __getattribute__(self, name, /)
 |      Return getattr(self, name).
 |
 |  __reduce__(...)
 |      Helper for pickle.
 |
 |  __repr__(self, /)
 |      Return repr(self).
 |
 |  __setattr__(self, name, value, /)
 |      Implement setattr(self, name, value).
 |
 |  __setstate__(...)
 |
 |  __str__(self, /)
 |      Return str(self).
 |
 |  with_traceback(...)
 |      Exception.with_traceback(tb) --
 |      set self.__traceback__ to tb and return self.
 |
 |  ----------------------------------------------------------------------
 |  Data descriptors inherited from builtins.BaseException:
 |
 |  __cause__
 |      exception cause
 |
 |  __context__
 |      exception context
 |
 |  __dict__
 |
 |  __suppress_context__
 |
 |  __traceback__
 |
 |  args
```

```
    class LSODA(scipy.integrate._ivp.base.OdeSolver)
     |  LSODA(fun, t0, y0, t_bound, first_step=None, min_step=0.0, max_step=inf, rtol=0.001, atol=1e-06, jac=None, lband=None, uba
nd=None, vectorized=False, **extraneous)
     |
     |  Adams/BDF method with automatic stiffness detection and switching.
     |
     |  This is a wrapper to the Fortran solver from ODEPACK [1]_. It switches
     |  automatically between the nonstiff Adams method and the stiff BDF method.
     |  The method was originally detailed in [2]_.
     |
     |  Parameters
     |  ----------
     |  fun : callable
     |      Right-hand side of the system. The calling signature is ``fun(t, y)``.
     |      Here ``t`` is a scalar, and there are two options for the ndarray ``y``:
     |      It can either have shape (n,); then ``fun`` must return array_like with
     |      shape (n,). Alternatively it can have shape (n, k); then ``fun``
     |      must return an array_like with shape (n, k), i.e. each column
     |      corresponds to a single column in ``y``. The choice between the two
     |      options is determined by `vectorized` argument (see below). The
     |      vectorized implementation allows a faster approximation of the Jacobian
     |      by finite differences (required for this solver).
     |  t0 : float
     |      Initial time.
     |  y0 : array_like, shape (n,)
     |      Initial state.
     |  t_bound : float
     |      Boundary time - the integration won't continue beyond it. It also
     |      determines the direction of the integration.
     |  first_step : float or None, optional
     |      Initial step size. Default is ``None`` which means that the algorithm
     |      should choose.
     |  min_step : float, optional
     |      Minimum allowed step size. Default is 0.0, i.e., the step size is not
     |      bounded and determined solely by the solver.
     |  max_step : float, optional
     |      Maximum allowed step size. Default is np.inf, i.e., the step size is not
     |      bounded and determined solely by the solver.
     |  rtol, atol : float and array_like, optional
     |      Relative and absolute tolerances. The solver keeps the local error
     |      estimates less than ``atol + rtol * abs(y)``. Here `rtol` controls a
     |      relative accuracy (number of correct digits). But if a component of `y`
     |      is approximately below `atol`, the error only needs to fall within
     |      the same `atol` threshold, and the number of correct digits is not
     |      guaranteed. If components of y have different scales, it might be
     |      beneficial to set different `atol` values for different components by
```

```
 |      passing array_like with shape (n,) for `atol`. Default values are
 |      1e-3 for `rtol` and 1e-6 for `atol`.
 |  jac : None or callable, optional
 |      Jacobian matrix of the right-hand side of the system with respect to
 |      ``y``. The Jacobian matrix has shape (n, n) and its element (i, j) is
 |      equal to ``d f_i / d y_j``. The function will be called as
 |      ``jac(t, y)``. If None (default), the Jacobian will be
 |      approximated by finite differences. It is generally recommended to
 |      provide the Jacobian rather than relying on a finite-difference
 |      approximation.
 |  lband, uband : int or None
 |      Parameters defining the bandwidth of the Jacobian,
 |      i.e., ``jac[i, j] != 0 only for i - lband <= j <= i + uband``. Setting
 |      these requires your jac routine to return the Jacobian in the packed format:
 |      the returned array must have ``n`` columns and ``uband + lband + 1``
 |      rows in which Jacobian diagonals are written. Specifically
 |      ``jac_packed[uband + i - j , j] = jac[i, j]``. The same format is used
 |      in `scipy.linalg.solve_banded` (check for an illustration).
 |      These parameters can be also used with ``jac=None`` to reduce the
 |      number of Jacobian elements estimated by finite differences.
 |  vectorized : bool, optional
 |      Whether `fun` is implemented in a vectorized fashion. A vectorized
 |      implementation offers no advantages for this solver. Default is False.
 |
 |  Attributes
 |  ----------
 |  n : int
 |      Number of equations.
 |  status : string
 |      Current status of the solver: 'running', 'finished' or 'failed'.
 |  t_bound : float
 |      Boundary time.
 |  direction : float
 |      Integration direction: +1 or -1.
 |  t : float
 |      Current time.
 |  y : ndarray
 |      Current state.
 |  t_old : float
 |      Previous time. None if no steps were made yet.
 |  nfev : int
 |      Number of evaluations of the right-hand side.
 |  njev : int
 |      Number of evaluations of the Jacobian.
 |
 |  References
 |  ----------
 |  .. [1] A. C. Hindmarsh, "ODEPACK, A Systematized Collection of ODE
 |          Solvers," IMACS Transactions on Scientific Computation, Vol 1.,
 |          pp. 55-64, 1983.
 |  .. [2] L. Petzold, "Automatic selection of methods for solving stiff and
 |          nonstiff systems of ordinary differential equations", SIAM Journal
 |          on Scientific and Statistical Computing, Vol. 4, No. 1, pp. 136-148,
 |          1983.
 |
 |  Method resolution order:
 |      LSODA
 |      scipy.integrate._ivp.base.OdeSolver
 |      builtins.object
 |
 |  Methods defined here:
 |
 |  __init__(self, fun, t0, y0, t_bound, first_step=None, min_step=0.0, max_step=inf, rtol=0.001, atol=1e-06, jac=None, lband=
None, uband=None, vectorized=False, **extraneous)
 |      Initialize self.  See help(type(self)) for accurate signature.
 |
 |  ----------------------------------------------------------------------
 |  Methods inherited from scipy.integrate._ivp.base.OdeSolver:
 |
 |  dense_output(self)
 |      Compute a local interpolant over the last successful step.
 |
 |      Returns
 |      -------
 |      sol : `DenseOutput`
 |          Local interpolant over the last successful step.
 |
 |  step(self)
 |      Perform one integration step.
 |
 |      Returns
 |      -------
 |      message : string or None
 |          Report from the solver. Typically a reason for a failure if
 |          `self.status` is 'failed' after the step was taken or None
 |          otherwise.
 |
 |  ----------------------------------------------------------------------
 |  Readonly properties inherited from scipy.integrate._ivp.base.OdeSolver:
 |
 |  step_size
 |
 |  ----------------------------------------------------------------------
 |  Data descriptors inherited from scipy.integrate._ivp.base.OdeSolver:
```

```
 |  
 |  __dict__
 |      dictionary for instance variables (if defined)
 |  
 |  __weakref__
 |      list of weak references to the object (if defined)
 |  
 |  ----------------------------------------------------------------------
 |  Data and other attributes inherited from scipy.integrate._ivp.base.OdeSolver:
 |  
 |  TOO_SMALL_STEP = 'Required step size is less than spacing between numb...

class OdeSolution(builtins.object)
 |  OdeSolution(ts, interpolants)
 |  
 |  Continuous ODE solution.
 |  
 |  It is organized as a collection of `DenseOutput` objects which represent
 |  local interpolants. It provides an algorithm to select a right interpolant
 |  for each given point.
 |  
 |  The interpolants cover the range between `t_min` and `t_max` (see
 |  Attributes below). Evaluation outside this interval is not forbidden, but
 |  the accuracy is not guaranteed.
 |  
 |  When evaluating at a breakpoint (one of the values in `ts`) a segment with
 |  the lower index is selected.
 |  
 |  Parameters
 |  ----------
 |  ts : array_like, shape (n_segments + 1,)
 |      Time instants between which local interpolants are defined. Must
 |      be strictly increasing or decreasing (zero segment with two points is
 |      also allowed).
 |  interpolants : list of DenseOutput with n_segments elements
 |      Local interpolants. An i-th interpolant is assumed to be defined
 |      between ``ts[i]`` and ``ts[i + 1]``.
 |  
 |  Attributes
 |  ----------
 |  t_min, t_max : float
 |      Time range of the interpolation.
 |  
 |  Methods defined here:
 |  
 |  __call__(self, t)
 |      Evaluate the solution.
 |
 |      Parameters
 |      ----------
 |      t : float or array_like with shape (n_points,)
 |          Points to evaluate at.
 |
 |      Returns
 |      -------
 |      y : ndarray, shape (n_states,) or (n_states, n_points)
 |          Computed values. Shape depends on whether `t` is a scalar or a
 |          1-D array.
 |
 |  __init__(self, ts, interpolants)
 |      Initialize self.  See help(type(self)) for accurate signature.
 |
 |  ----------------------------------------------------------------------
 |  Data descriptors defined here:
 |
 |  __dict__
 |      dictionary for instance variables (if defined)
 |
 |  __weakref__
 |      list of weak references to the object (if defined)

class OdeSolver(builtins.object)
 |  OdeSolver(fun, t0, y0, t_bound, vectorized, support_complex=False)
 |
 |  Base class for ODE solvers.
 |
 |  In order to implement a new solver you need to follow the guidelines:
 |
 |      1. A constructor must accept parameters presented in the base class
 |         (listed below) along with any other parameters specific to a solver.
 |      2. A constructor must accept arbitrary extraneous arguments
 |         ``**extraneous``, but warn that these arguments are irrelevant
 |         using `common.warn_extraneous` function. Do not pass these
 |         arguments to the base class.
 |      3. A solver must implement a private method `_step_impl(self)` which
 |         propagates a solver one step further. It must return tuple
 |         ``(success, message)``, where ``success`` is a boolean indicating
 |         whether a step was successful, and ``message`` is a string
 |         containing description of a failure if a step failed or None
 |         otherwise.
 |      4. A solver must implement a private method `_dense_output_impl(self)`,
 |         which returns a `DenseOutput` object covering the last successful
 |         step.
 |      5. A solver must have attributes listed below in Attributes section.
 |          Note that ``t_old`` and ``step_size`` are updated automatically.
 |       6. Use `fun(self, t, y)` method for the system rhs evaluation, this
 |          way the number of function evaluations (`nfev`) will be tracked
 |          automatically.
 |       7. For convenience, a base class provides `fun_single(self, t, y)` and
 |          `fun_vectorized(self, t, y)` for evaluating the rhs in
 |          non-vectorized and vectorized fashions respectively (regardless of
 |          how `fun` from the constructor is implemented). These calls don't
 |          increment `nfev`.
 |       8. If a solver uses a Jacobian matrix and LU decompositions, it should
 |          track the number of Jacobian evaluations (`njev`) and the number of
 |          LU decompositions (`nlu`).
 |       9. By convention, the function evaluations used to compute a finite
 |          difference approximation of the Jacobian should not be counted in
 |          `nfev`, thus use `fun_single(self, t, y)` or
 |          `fun_vectorized(self, t, y)` when computing a finite difference
 |          approximation of the Jacobian.
 |
 |  Parameters
 |  ----------
 |  fun : callable
 |      Right-hand side of the system. The calling signature is ``fun(t, y)``.
 |      Here ``t`` is a scalar and there are two options for ndarray ``y``.
 |      It can either have shape (n,), then ``fun`` must return array_like with
 |      shape (n,). Or, alternatively, it can have shape (n, n_points), then
 |      ``fun`` must return array_like with shape (n, n_points) (each column
 |      corresponds to a single column in ``y``). The choice between the two
 |      options is determined by `vectorized` argument (see below).
 |  t0 : float
 |      Initial time.
 |  y0 : array_like, shape (n,)
 |      Initial state.
 |  t_bound : float
 |      Boundary time --- the integration won't continue beyond it. It also
 |      determines the direction of the integration.
 |  vectorized : bool
 |      Whether `fun` is implemented in a vectorized fashion.
 |  support_complex : bool, optional
 |      Whether integration in a complex domain should be supported.
 |      Generally determined by a derived solver class capabilities.
 |      Default is False.
 |
 |  Attributes
 |  ----------
 |  n : int
 |      Number of equations.
 |  status : string
 |      Current status of the solver: 'running', 'finished' or 'failed'.
 |  t_bound : float
 |      Boundary time.
 |  direction : float
 |      Integration direction: +1 or -1.
 |  t : float
 |      Current time.
 |  y : ndarray
 |      Current state.
 |  t_old : float
 |      Previous time. None if no steps were made yet.
 |  step_size : float
 |      Size of the last successful step. None if no steps were made yet.
 |  nfev : int
 |      Number of the system's rhs evaluations.
 |  njev : int
 |      Number of the Jacobian evaluations.
 |  nlu : int
 |      Number of LU decompositions.
 |
 |  Methods defined here:
 |
 |  __init__(self, fun, t0, y0, t_bound, vectorized, support_complex=False)
 |      Initialize self.  See help(type(self)) for accurate signature.
 |
 |  dense_output(self)
 |      Compute a local interpolant over the last successful step.
 |
 |      Returns
 |      -------
 |      sol : `DenseOutput`
 |          Local interpolant over the last successful step.
 |
 |  step(self)
 |      Perform one integration step.
 |
 |      Returns
 |      -------
 |      message : string or None
 |          Report from the solver. Typically a reason for a failure if
 |          `self.status` is 'failed' after the step was taken or None
 |          otherwise.
 |
 |  ----------------------------------------------------------------------
 |  Readonly properties defined here:
 |
 |  step_size
 |  
 |  ----------------------------------------------------------------------
 |  Data descriptors defined here:
 |  
 |  __dict__
 |      dictionary for instance variables (if defined)
 |  
 |  __weakref__
 |      list of weak references to the object (if defined)
 |  
 |  ----------------------------------------------------------------------
 |  Data and other attributes defined here:
 |  
 |  TOO_SMALL_STEP = 'Required step size is less than spacing between numb...

class RK23(RungeKutta)
 |  RK23(fun, t0, y0, t_bound, max_step=inf, rtol=0.001, atol=1e-06, vectorized=False, first_step=None, **extraneous)
 |  
 |  Explicit Runge-Kutta method of order 3(2).
 |  
 |  This uses the Bogacki-Shampine pair of formulas [1]_. The error is controlled
 |  assuming accuracy of the second-order method, but steps are taken using the
 |  third-order accurate formula (local extrapolation is done). A cubic Hermite
 |  polynomial is used for the dense output.
 |  
 |  Can be applied in the complex domain.
 |  
 |  Parameters
 |  ----------
 |  fun : callable
 |      Right-hand side of the system. The calling signature is ``fun(t, y)``.
 |      Here ``t`` is a scalar and there are two options for ndarray ``y``.
 |      It can either have shape (n,), then ``fun`` must return array_like with
 |      shape (n,). Or alternatively it can have shape (n, k), then ``fun``
 |      must return array_like with shape (n, k), i.e. each column
 |      corresponds to a single column in ``y``. The choice between the two
 |      options is determined by `vectorized` argument (see below).
 |  t0 : float
 |      Initial time.
 |  y0 : array_like, shape (n,)
 |      Initial state.
 |  t_bound : float
 |      Boundary time - the integration won't continue beyond it. It also
 |      determines the direction of the integration.
 |  first_step : float or None, optional
 |      Initial step size. Default is ``None`` which means that the algorithm
 |      should choose.
 |  max_step : float, optional
 |      Maximum allowed step size. Default is np.inf, i.e., the step size is not
 |      bounded and determined solely by the solver.
 |  rtol, atol : float and array_like, optional
 |      Relative and absolute tolerances. The solver keeps the local error
 |      estimates less than ``atol + rtol * abs(y)``. Here, `rtol` controls a
 |      relative accuracy (number of correct digits). But if a component of `y`
 |      is approximately below `atol`, the error only needs to fall within
 |      the same `atol` threshold, and the number of correct digits is not
 |      guaranteed. If components of y have different scales, it might be
 |      beneficial to set different `atol` values for different components by
 |      passing array_like with shape (n,) for `atol`. Default values are
 |      1e-3 for `rtol` and 1e-6 for `atol`.
 |  vectorized : bool, optional
 |      Whether `fun` is implemented in a vectorized fashion. Default is False.
 |
 |  Attributes
 |  ----------
 |  n : int
 |      Number of equations.
 |  status : string
 |      Current status of the solver: 'running', 'finished' or 'failed'.
 |  t_bound : float
 |      Boundary time.
 |  direction : float
 |      Integration direction: +1 or -1.
 |  t : float
 |      Current time.
 |  y : ndarray
 |      Current state.
 |  t_old : float
 |      Previous time. None if no steps were made yet.
 |  step_size : float
 |      Size of the last successful step. None if no steps were made yet.
 |  nfev : int
 |      Number evaluations of the system's right-hand side.
 |  njev : int
 |      Number of evaluations of the Jacobian. Is always 0 for this solver as it does not use the Jacobian.
 |  nlu : int
 |      Number of LU decompositions. Is always 0 for this solver.
 |
 |  References
 |  ----------
 |  .. [1] P. Bogacki, L.F. Shampine, "A 3(2) Pair of Runge-Kutta Formulas",
 |         Appl. Math. Lett. Vol. 2, No. 4. pp. 321-325, 1989.
 |
 |  Method resolution order:
        |      RK23
        |      RungeKutta
        |      scipy.integrate._ivp.base.OdeSolver
        |      builtins.object
        |
        |  Data and other attributes defined here:
        |
        |  A = array([[0.  , 0.  , 0.  ],
        |         [0.5 , 0.  , 0.  ],
        |         [0.  ...
        |
        |  B = array([0.22222222, 0.33333333, 0.44444444])
        |
        |  C = array([0.  , 0.5 , 0.75])
        |
        |  E = array([ 0.06944444, -0.08333333, -0.11111111,  0.125     ])
        |
        |  P = array([[ 1.        , -1.33333333,  0.55555556],
        |  ...
        |         [ 0.   ...
        |
        |  error_estimator_order = 2
        |
        |  n_stages = 3
        |
        |  order = 3
        |
        |  ----------------------------------------------------------------------
        |  Methods inherited from RungeKutta:
        |
        |  __init__(self, fun, t0, y0, t_bound, max_step=inf, rtol=0.001, atol=1e-06, vectorized=False, first_step=None, **extraneou
s)
        |      Initialize self.  See help(type(self)) for accurate signature.
        |
        |  ----------------------------------------------------------------------
        |  Methods inherited from scipy.integrate._ivp.base.OdeSolver:
        |
        |  dense_output(self)
        |      Compute a local interpolant over the last successful step.
        |
        |      Returns
        |      -------
        |      sol : `DenseOutput`
        |          Local interpolant over the last successful step.
        |
        |  step(self)
        |      Perform one integration step.
```

```
 |
 |      Returns
 |      -------
 |      message : string or None
 |          Report from the solver. Typically a reason for a failure if
 |          `self.status` is 'failed' after the step was taken or None
 |          otherwise.
 |
 |  ----------------------------------------------------------------------
 |  Readonly properties inherited from scipy.integrate._ivp.base.OdeSolver:
 |
 |  step_size
 |
 |  ----------------------------------------------------------------------
 |  Data descriptors inherited from scipy.integrate._ivp.base.OdeSolver:
 |
 |  __dict__
 |      dictionary for instance variables (if defined)
 |
 |  __weakref__
 |      list of weak references to the object (if defined)
 |
 |  ----------------------------------------------------------------------
 |  Data and other attributes inherited from scipy.integrate._ivp.base.OdeSolver:
 |
 |  TOO_SMALL_STEP = 'Required step size is less than spacing between numb...

class RK45(RungeKutta)
 |  RK45(fun, t0, y0, t_bound, max_step=inf, rtol=0.001, atol=1e-06, vectorized=False, first_step=None, **extraneous)
 |
 |  Explicit Runge-Kutta method of order 5(4).
 |
 |  This uses the Dormand-Prince pair of formulas [1]_. The error is controlled
 |  assuming accuracy of the fourth-order method accuracy, but steps are taken
 |  using the fifth-order accurate formula (local extrapolation is done).
 |  A quartic interpolation polynomial is used for the dense output [2]_.
 |
 |  Can be applied in the complex domain.
 |
 |  Parameters
 |  ----------
 |  fun : callable
 |      Right-hand side of the system. The calling signature is ``fun(t, y)``.
 |      Here ``t`` is a scalar, and there are two options for the ndarray ``y``:
 |      It can either have shape (n,); then ``fun`` must return array_like with
 |      shape (n,). Alternatively it can have shape (n, k); then ``fun``
 |      must return an array_like with shape (n, k), i.e., each column
 |          corresponds to a single column in ``y``. The choice between the two
 |          options is determined by `vectorized` argument (see below).
 |      t0 : float
 |          Initial time.
 |      y0 : array_like, shape (n,)
 |          Initial state.
 |      t_bound : float
 |          Boundary time - the integration won't continue beyond it. It also
 |          determines the direction of the integration.
 |      first_step : float or None, optional
 |          Initial step size. Default is ``None`` which means that the algorithm
 |          should choose.
 |      max_step : float, optional
 |          Maximum allowed step size. Default is np.inf, i.e., the step size is not
 |          bounded and determined solely by the solver.
 |      rtol, atol : float and array_like, optional
 |          Relative and absolute tolerances. The solver keeps the local error
 |          estimates less than ``atol + rtol * abs(y)``. Here `rtol` controls a
 |          relative accuracy (number of correct digits). But if a component of `y`
 |          is approximately below `atol`, the error only needs to fall within
 |          the same `atol` threshold, and the number of correct digits is not
 |          guaranteed. If components of y have different scales, it might be
 |          beneficial to set different `atol` values for different components by
 |          passing array_like with shape (n,) for `atol`. Default values are
 |          1e-3 for `rtol` and 1e-6 for `atol`.
 |      vectorized : bool, optional
 |          Whether `fun` is implemented in a vectorized fashion. Default is False.
 |
 |      Attributes
 |      ----------
 |      n : int
 |          Number of equations.
 |      status : string
 |          Current status of the solver: 'running', 'finished' or 'failed'.
 |      t_bound : float
 |          Boundary time.
 |      direction : float
 |          Integration direction: +1 or -1.
 |      t : float
 |          Current time.
 |      y : ndarray
 |          Current state.
 |      t_old : float
 |          Previous time. None if no steps were made yet.
 |      step_size : float
 |          Size of the last successful step. None if no steps were made yet.
 |      nfev : int
 |      Number evaluations of the system's right-hand side.
 |  njev : int
 |      Number of evaluations of the Jacobian. Is always 0 for this solver as it does not use the Jacobian.
 |  nlu : int
 |      Number of LU decompositions. Is always 0 for this solver.
 |  
 |  References
 |  ----------
 |  .. [1] J. R. Dormand, P. J. Prince, "A family of embedded Runge-Kutta
 |         formulae", Journal of Computational and Applied Mathematics, Vol. 6,
 |         No. 1, pp. 19-26, 1980.
 |  .. [2] L. W. Shampine, "Some Practical Runge-Kutta Formulas", Mathematics
 |         of Computation,, Vol. 46, No. 173, pp. 135-150, 1986.
 |  
 |  Method resolution order:
 |      RK45
 |      RungeKutta
 |      scipy.integrate._ivp.base.OdeSolver
 |      builtins.object
 |  
 |  Data and other attributes defined here:
 |  
 |  A = array([[  0.        ,   0.        ,   0.        ...8.90642272,   0...
 |  
 |  B = array([ 0.09114583,  0.        ,  0.4492363 ,  0.65104167, -0.3223...
 |  
 |  C = array([0.        , 0.2       , 0.3       , 0.8       , 0.88888889,...
 |  
 |  E = array([-0.00123264,  0.        ,  0.00425277, -0...7,  0.0508638 ,...
 |  
 |  P = array([[ 1.        , -2.85358007,  3.07174346, -...       ,  1.382...
 |  
 |  error_estimator_order = 4
 |  
 |  n_stages = 6
 |  
 |  order = 5
 |  
 |  ----------------------------------------------------------------------
 |  Methods inherited from RungeKutta:
 |  
 |  __init__(self, fun, t0, y0, t_bound, max_step=inf, rtol=0.001, atol=1e-06, vectorized=False, first_step=None, **extraneou
s)
 |      Initialize self.  See help(type(self)) for accurate signature.
 |  
 |  ----------------------------------------------------------------------
 |  Methods inherited from scipy.integrate._ivp.base.OdeSolver:
     |
     |  dense_output(self)
     |      Compute a local interpolant over the last successful step.
     |
     |      Returns
     |      -------
     |      sol : `DenseOutput`
     |          Local interpolant over the last successful step.
     |
     |  step(self)
     |      Perform one integration step.
     |
     |      Returns
     |      -------
     |      message : string or None
     |          Report from the solver. Typically a reason for a failure if
     |          `self.status` is 'failed' after the step was taken or None
     |          otherwise.
     |
     |  ----------------------------------------------------------------------
     |  Readonly properties inherited from scipy.integrate._ivp.base.OdeSolver:
     |
     |  step_size
     |
     |  ----------------------------------------------------------------------
     |  Data descriptors inherited from scipy.integrate._ivp.base.OdeSolver:
     |
     |  __dict__
     |      dictionary for instance variables (if defined)
     |
     |  __weakref__
     |      list of weak references to the object (if defined)
     |
     |  ----------------------------------------------------------------------
     |  Data and other attributes inherited from scipy.integrate._ivp.base.OdeSolver:
     |
     |  TOO_SMALL_STEP = 'Required step size is less than spacing between numb...

    class Radau(scipy.integrate._ivp.base.OdeSolver)
     |  Radau(fun, t0, y0, t_bound, max_step=inf, rtol=0.001, atol=1e-06, jac=None, jac_sparsity=None, vectorized=False, first_ste
p=None, **extraneous)
     |
     |  Implicit Runge-Kutta method of Radau IIA family of order 5.
     |
     |  The implementation follows [1]_. The error is controlled with a
     |  third-order accurate embedded formula. A cubic polynomial which satisfies
     |  the collocation conditions is used for the dense output.
```

```
 |  
 |  Parameters
 |  ----------
 |  fun : callable
 |      Right-hand side of the system. The calling signature is ``fun(t, y)``.
 |      Here ``t`` is a scalar, and there are two options for the ndarray ``y``:
 |      It can either have shape (n,); then ``fun`` must return array_like with
 |      shape (n,). Alternatively it can have shape (n, k); then ``fun``
 |      must return an array_like with shape (n, k), i.e., each column
 |      corresponds to a single column in ``y``. The choice between the two
 |      options is determined by `vectorized` argument (see below). The
 |      vectorized implementation allows a faster approximation of the Jacobian
 |      by finite differences (required for this solver).
 |  t0 : float
 |      Initial time.
 |  y0 : array_like, shape (n,)
 |      Initial state.
 |  t_bound : float
 |      Boundary time - the integration won't continue beyond it. It also
 |      determines the direction of the integration.
 |  first_step : float or None, optional
 |      Initial step size. Default is ``None`` which means that the algorithm
 |      should choose.
 |  max_step : float, optional
 |      Maximum allowed step size. Default is np.inf, i.e., the step size is not
 |      bounded and determined solely by the solver.
 |  rtol, atol : float and array_like, optional
 |      Relative and absolute tolerances. The solver keeps the local error
 |      estimates less than ``atol + rtol * abs(y)``. Here `rtol` controls a
 |      relative accuracy (number of correct digits). But if a component of `y`
 |      is approximately below `atol`, the error only needs to fall within
 |      the same `atol` threshold, and the number of correct digits is not
 |      guaranteed. If components of y have different scales, it might be
 |      beneficial to set different `atol` values for different components by
 |      passing array_like with shape (n,) for `atol`. Default values are
 |      1e-3 for `rtol` and 1e-6 for `atol`.
 |  jac : {None, array_like, sparse_matrix, callable}, optional
 |      Jacobian matrix of the right-hand side of the system with respect to
 |      y, required by this method. The Jacobian matrix has shape (n, n) and
 |      its element (i, j) is equal to ``d f_i / d y_j``.
 |      There are three ways to define the Jacobian:
 |  
 |          * If array_like or sparse_matrix, the Jacobian is assumed to
 |            be constant.
 |          * If callable, the Jacobian is assumed to depend on both
 |            t and y; it will be called as ``jac(t, y)`` as necessary.
 |            For the 'Radau' and 'BDF' methods, the return value might be a
 |              sparse matrix.
 |            * If None (default), the Jacobian will be approximated by
 |              finite differences.
 |
 |        It is generally recommended to provide the Jacobian rather than
 |        relying on a finite-difference approximation.
 |    jac_sparsity : {None, array_like, sparse matrix}, optional
 |        Defines a sparsity structure of the Jacobian matrix for a
 |        finite-difference approximation. Its shape must be (n, n). This argument
 |        is ignored if `jac` is not `None`. If the Jacobian has only few non-zero
 |        elements in *each* row, providing the sparsity structure will greatly
 |        speed up the computations [2]_. A zero entry means that a corresponding
 |        element in the Jacobian is always zero. If None (default), the Jacobian
 |        is assumed to be dense.
 |    vectorized : bool, optional
 |        Whether `fun` is implemented in a vectorized fashion. Default is False.
 |
 |    Attributes
 |    ----------
 |    n : int
 |        Number of equations.
 |    status : string
 |        Current status of the solver: 'running', 'finished' or 'failed'.
 |    t_bound : float
 |        Boundary time.
 |    direction : float
 |        Integration direction: +1 or -1.
 |    t : float
 |        Current time.
 |    y : ndarray
 |        Current state.
 |    t_old : float
 |        Previous time. None if no steps were made yet.
 |    step_size : float
 |        Size of the last successful step. None if no steps were made yet.
 |    nfev : int
 |        Number of evaluations of the right-hand side.
 |    njev : int
 |        Number of evaluations of the Jacobian.
 |    nlu : int
 |        Number of LU decompositions.
 |
 |    References
 |    ----------
 |    .. [1] E. Hairer, G. Wanner, "Solving Ordinary Differential Equations II:
 |           Stiff and Differential-Algebraic Problems", Sec. IV.8.
 |    .. [2] A. Curtis, M. J. D. Powell, and J. Reid, "On the estimation of
```

```
     |         sparse Jacobian matrices", Journal of the Institute of Mathematics
     |         and its Applications, 13, pp. 117-120, 1974.
     |
     |  Method resolution order:
     |      Radau
     |      scipy.integrate._ivp.base.OdeSolver
     |      builtins.object
     |
     |  Methods defined here:
     |
     |  __init__(self, fun, t0, y0, t_bound, max_step=inf, rtol=0.001, atol=1e-06, jac=None, jac_sparsity=None, vectorized=False, 
first_step=None, **extraneous)
     |      Initialize self.  See help(type(self)) for accurate signature.
     |
     |  ----------------------------------------------------------------------
     |  Methods inherited from scipy.integrate._ivp.base.OdeSolver:
     |
     |  dense_output(self)
     |      Compute a local interpolant over the last successful step.
     |
     |      Returns
     |      -------
     |      sol : `DenseOutput`
     |          Local interpolant over the last successful step.
     |
     |  step(self)
     |      Perform one integration step.
     |
     |      Returns
     |      -------
     |      message : string or None
     |          Report from the solver. Typically a reason for a failure if
     |          `self.status` is 'failed' after the step was taken or None
     |          otherwise.
     |
     |  ----------------------------------------------------------------------
     |  Readonly properties inherited from scipy.integrate._ivp.base.OdeSolver:
     |
     |  step_size
     |
     |  ----------------------------------------------------------------------
     |  Data descriptors inherited from scipy.integrate._ivp.base.OdeSolver:
     |
     |  __dict__
     |      dictionary for instance variables (if defined)
     |
     |  __weakref__
```

```
 |      list of weak references to the object (if defined)
 |  
 |  ----------------------------------------------------------------------
 |  Data and other attributes inherited from scipy.integrate._ivp.base.OdeSolver:
 |  
 |  TOO_SMALL_STEP = 'Required step size is less than spacing between numb...

class complex_ode(ode)
 |  complex_ode(f, jac=None)
 |  
 |  A wrapper of ode for complex systems.
 |  
 |  This functions similarly as `ode`, but re-maps a complex-valued
 |  equation system to a real-valued one before using the integrators.
 |  
 |  Parameters
 |  ----------
 |  f : callable ``f(t, y, *f_args)``
 |      Rhs of the equation. t is a scalar, ``y.shape == (n,)``.
 |      ``f_args`` is set by calling ``set_f_params(*args)``.
 |  jac : callable ``jac(t, y, *jac_args)``
 |      Jacobian of the rhs, ``jac[i,j] = d f[i] / d y[j]``.
 |      ``jac_args`` is set by calling ``set_f_params(*args)``.
 |  
 |  Attributes
 |  ----------
 |  t : float
 |      Current time.
 |  y : ndarray
 |      Current variable values.
 |  
 |  Examples
 |  --------
 |  For usage examples, see `ode`.
 |  
 |  Method resolution order:
 |      complex_ode
 |      ode
 |      builtins.object
 |  
 |  Methods defined here:
 |  
 |  __init__(self, f, jac=None)
 |      Initialize self.  See help(type(self)) for accurate signature.
 |  
 |  integrate(self, t, step=False, relax=False)
 |      Find y=y(t), set y as an initial condition, and return y.
 |      
 |      Parameters
 |      ----------
 |      t : float
 |          The endpoint of the integration step.
 |      step : bool
 |          If True, and if the integrator supports the step method,
 |          then perform a single integration step and return.
 |          This parameter is provided in order to expose internals of
 |          the implementation, and should not be changed from its default
 |          value in most cases.
 |      relax : bool
 |          If True and if the integrator supports the run_relax method,
 |          then integrate until t_1 >= t and return. ``relax`` is not
 |          referenced if ``step=True``.
 |          This parameter is provided in order to expose internals of
 |          the implementation, and should not be changed from its default
 |          value in most cases.
 |      
 |      Returns
 |      -------
 |      y : float
 |          The integrated value at t
 |  
 |  set_initial_value(self, y, t=0.0)
 |      Set initial conditions y(t) = y.
 |  
 |  set_integrator(self, name, **integrator_params)
 |      Set integrator by name.
 |      
 |      Parameters
 |      ----------
 |      name : str
 |          Name of the integrator
 |      integrator_params
 |          Additional parameters for the integrator.
 |  
 |  set_solout(self, solout)
 |      Set callable to be called at every successful integration step.
 |      
 |      Parameters
 |      ----------
 |      solout : callable
 |          ``solout(t, y)`` is called at each internal integrator step,
 |          t is a scalar providing the current independent position
 |          y is the current soloution ``y.shape == (n,)``
 |          solout should return -1 to stop integration
 |          otherwise it should return None or 0
 |
 |  ----------------------------------------------------------------------
 |  Readonly properties defined here:
 |
 |  y
 |
 |  ----------------------------------------------------------------------
 |  Methods inherited from ode:
 |
 |  get_return_code(self)
 |      Extracts the return code for the integration to enable better control
 |      if the integration fails.
 |
 |      In general, a return code > 0 implies success, while a return code < 0
 |      implies failure.
 |
 |      Notes
 |      -----
 |      This section describes possible return codes and their meaning, for available
 |      integrators that can be selected by `set_integrator` method.
 |
 |      "vode"
 |
 |      ===========  =======
 |      Return Code  Message
 |      ===========  =======
 |      2            Integration successful.
 |      -1           Excess work done on this call. (Perhaps wrong MF.)
 |      -2           Excess accuracy requested. (Tolerances too small.)
 |      -3           Illegal input detected. (See printed message.)
 |      -4           Repeated error test failures. (Check all input.)
 |      -5           Repeated convergence failures. (Perhaps bad Jacobian
 |                   supplied or wrong choice of MF or tolerances.)
 |      -6           Error weight became zero during problem. (Solution
 |                   component i vanished, and ATOL or ATOL(i) = 0.)
 |      ===========  =======
 |
 |      "zvode"
 |
 |      ===========  =======
 |      Return Code  Message
 |      ===========  =======
 |      2            Integration successful.
 |      -1           Excess work done on this call. (Perhaps wrong MF.)
 |      -2           Excess accuracy requested. (Tolerances too small.)
 |      -3           Illegal input detected. (See printed message.)
```

```
    -4           Repeated error test failures. (Check all input.)
    -5           Repeated convergence failures. (Perhaps bad Jacobian
                 supplied or wrong choice of MF or tolerances.)
    -6           Error weight became zero during problem. (Solution
                 component i vanished, and ATOL or ATOL(i) = 0.)
    ===========  =======

    "dopri5"

    ===========  =======
    Return Code  Message
    ===========  =======
    1            Integration successful.
    2            Integration successful (interrupted by solout).
    -1           Input is not consistent.
    -2           Larger nsteps is needed.
    -3           Step size becomes too small.
    -4           Problem is probably stiff (interrupted).
    ===========  =======

    "dop853"

    ===========  =======
    Return Code  Message
    ===========  =======
    1            Integration successful.
    2            Integration successful (interrupted by solout).
    -1           Input is not consistent.
    -2           Larger nsteps is needed.
    -3           Step size becomes too small.
    -4           Problem is probably stiff (interrupted).
    ===========  =======

    "lsoda"

    ===========  =======
    Return Code  Message
    ===========  =======
    2            Integration successful.
    -1           Excess work done on this call (perhaps wrong Dfun type).
    -2           Excess accuracy requested (tolerances too small).
    -3           Illegal input detected (internal error).
    -4           Repeated error test failures (internal error).
    -5           Repeated convergence failures (perhaps bad Jacobian or tolerances).
    -6           Error weight became zero during problem.
    -7           Internal workspace insufficient to finish (internal error).
    ===========  =======
    |  
    |  set_f_params(self, *args)
    |      Set extra parameters for user-supplied function f.
    |  
    |  set_jac_params(self, *args)
    |      Set extra parameters for user-supplied function jac.
    |  
    |  successful(self)
    |      Check if integration was successful.
    |  
    |  ----------------------------------------------------------------------
    |  Data descriptors inherited from ode:
    |  
    |  __dict__
    |      dictionary for instance variables (if defined)
    |  
    |  __weakref__
    |      list of weak references to the object (if defined)

   class ode(builtins.object)
    |  ode(f, jac=None)
    |  
    |  A generic interface class to numeric integrators.
    |  
    |  Solve an equation system :math:`y'(t) = f(t,y)` with (optional) ``jac = df/dy``.
    |  
    |  *Note*: The first two arguments of ``f(t, y, ...)`` are in the
    |  opposite order of the arguments in the system definition function used
    |  by `scipy.integrate.odeint`.
    |  
    |  Parameters
    |  ----------
    |  f : callable ``f(t, y, *f_args)``
    |      Right-hand side of the differential equation. t is a scalar,
    |      ``y.shape == (n,)``.
    |      ``f_args`` is set by calling ``set_f_params(*args)``.
    |      `f` should return a scalar, array or list (not a tuple).
    |  jac : callable ``jac(t, y, *jac_args)``, optional
    |      Jacobian of the right-hand side, ``jac[i,j] = d f[i] / d y[j]``.
    |      ``jac_args`` is set by calling ``set_jac_params(*args)``.
    |  
    |  Attributes
    |  ----------
    |  t : float
    |      Current time.
    |  y : ndarray
    |      Current variable values.
```

```
|
|  See also
|  --------
|  odeint : an integrator with a simpler interface based on lsoda from ODEPACK
|  quad : for finding the area under a curve
|
|  Notes
|  -----
|  Available integrators are listed below. They can be selected using
|  the `set_integrator` method.
|
|  "vode"
|
|      Real-valued Variable-coefficient Ordinary Differential Equation
|      solver, with fixed-leading-coefficient implementation. It provides
|      implicit Adams method (for non-stiff problems) and a method based on
|      backward differentiation formulas (BDF) (for stiff problems).
|
|      Source: http://www.netlib.org/ode/vode.f
|
|      .. warning::
|
|         This integrator is not re-entrant. You cannot have two `ode`
|         instances using the "vode" integrator at the same time.
|
|      This integrator accepts the following parameters in `set_integrator`
|      method of the `ode` class:
|
|      - atol : float or sequence
|        absolute tolerance for solution
|      - rtol : float or sequence
|        relative tolerance for solution
|      - lband : None or int
|      - uband : None or int
|        Jacobian band width, jac[i,j] != 0 for i-lband <= j <= i+uband.
|        Setting these requires your jac routine to return the jacobian
|        in packed format, jac_packed[i-j+uband, j] = jac[i,j]. The
|        dimension of the matrix must be (lband+uband+1, len(y)).
|      - method: 'adams' or 'bdf'
|        Which solver to use, Adams (non-stiff) or BDF (stiff)
|      - with_jacobian : bool
|        This option is only considered when the user has not supplied a
|        Jacobian function and has not indicated (by setting either band)
|        that the Jacobian is banded. In this case, `with_jacobian` specifies
|        whether the iteration method of the ODE solver's correction step is
|        chord iteration with an internally generated full Jacobian or
|        functional iteration with no Jacobian.
|
|      - nsteps : int
|        Maximum number of (internally defined) steps allowed during one
|        call to the solver.
|      - first_step : float
|      - min_step : float
|      - max_step : float
|        Limits for the step sizes used by the integrator.
|      - order : int
|        Maximum order used by the integrator,
|        order <= 12 for Adams, <= 5 for BDF.
|
|  "zvode"
|
|      Complex-valued Variable-coefficient Ordinary Differential Equation
|      solver, with fixed-leading-coefficient implementation. It provides
|      implicit Adams method (for non-stiff problems) and a method based on
|      backward differentiation formulas (BDF) (for stiff problems).
|
|      Source: http://www.netlib.org/ode/zvode.f
|
|      .. warning::
|
|         This integrator is not re-entrant. You cannot have two `ode`
|         instances using the "zvode" integrator at the same time.
|
|      This integrator accepts the same parameters in `set_integrator`
|      as the "vode" solver.
|
|      .. note::
|
|          When using ZVODE for a stiff system, it should only be used for
|          the case in which the function f is analytic, that is, when each f(i)
|          is an analytic function of each y(j). Analyticity means that the
|          partial derivative df(i)/dy(j) is a unique complex number, and this
|          fact is critical in the way ZVODE solves the dense or banded linear
|          systems that arise in the stiff case. For a complex stiff ODE system
|          in which f is not analytic, ZVODE is likely to have convergence
|          failures, and for this problem one should instead use DVODE on the
|          equivalent real system (in the real and imaginary parts of y).
|
|  "lsoda"
|
|      Real-valued Variable-coefficient Ordinary Differential Equation
|      solver, with fixed-leading-coefficient implementation. It provides
|      automatic method switching between implicit Adams method (for non-stiff
|      problems) and a method based on backward differentiation formulas (BDF)
|      (for stiff problems).
```

```
|
|       Source: http://www.netlib.org/odepack
|
|       .. warning::
|
|          This integrator is not re-entrant. You cannot have two `ode`
|          instances using the "lsoda" integrator at the same time.
|
|       This integrator accepts the following parameters in `set_integrator`
|       method of the `ode` class:
|
|       - atol : float or sequence
|         absolute tolerance for solution
|       - rtol : float or sequence
|         relative tolerance for solution
|       - lband : None or int
|       - uband : None or int
|         Jacobian band width, jac[i,j] != 0 for i-lband <= j <= i+uband.
|         Setting these requires your jac routine to return the jacobian
|         in packed format, jac_packed[i-j+uband, j] = jac[i,j].
|       - with_jacobian : bool
|         *Not used.*
|       - nsteps : int
|         Maximum number of (internally defined) steps allowed during one
|         call to the solver.
|       - first_step : float
|       - min_step : float
|       - max_step : float
|         Limits for the step sizes used by the integrator.
|       - max_order_ns : int
|         Maximum order used in the nonstiff case (default 12).
|       - max_order_s : int
|         Maximum order used in the stiff case (default 5).
|       - max_hnil : int
|         Maximum number of messages reporting too small step size (t + h = t)
|         (default 0)
|       - ixpr : int
|         Whether to generate extra printing at method switches (default False).
|
|   "dopri5"
|
|       This is an explicit runge-kutta method of order (4)5 due to Dormand &
|       Prince (with stepsize control and dense output).
|
|       Authors:
|
|           E. Hairer and G. Wanner
```

```
 |          Universite de Geneve, Dept. de Mathematiques
 |          CH-1211 Geneve 24, Switzerland
 |          e-mail:  ernst.hairer@math.unige.ch, gerhard.wanner@math.unige.ch
 |  
 |      This code is described in [HNW93]_.
 |  
 |      This integrator accepts the following parameters in set_integrator()
 |      method of the ode class:
 |  
 |      - atol : float or sequence
 |        absolute tolerance for solution
 |      - rtol : float or sequence
 |        relative tolerance for solution
 |      - nsteps : int
 |        Maximum number of (internally defined) steps allowed during one
 |        call to the solver.
 |      - first_step : float
 |      - max_step : float
 |      - safety : float
 |        Safety factor on new step selection (default 0.9)
 |      - ifactor : float
 |      - dfactor : float
 |        Maximum factor to increase/decrease step size by in one step
 |      - beta : float
 |        Beta parameter for stabilised step size control.
 |      - verbosity : int
 |        Switch for printing messages (< 0 for no messages).
 |  
 |  "dop853"
 |  
 |      This is an explicit runge-kutta method of order 8(5,3) due to Dormand
 |      & Prince (with stepsize control and dense output).
 |  
 |      Options and references the same as "dopri5".
 |  
 |  Examples
 |  --------
 |  
 |  A problem to integrate and the corresponding jacobian:
 |  
 |  >>> from scipy.integrate import ode
 |  >>>
 |  >>> y0, t0 = [1.0j, 2.0], 0
 |  >>>
 |  >>> def f(t, y, arg1):
 |  ...     return [1j*arg1*y[0] + y[1], -arg1*y[1]**2]
 |  >>> def jac(t, y, arg1):
```

```
 |  ...     return [[1j*arg1, 1], [0, -arg1*2*y[1]]]
 |
 |  The integration:
 |
 |  >>> r = ode(f, jac).set_integrator('zvode', method='bdf')
 |  >>> r.set_initial_value(y0, t0).set_f_params(2.0).set_jac_params(2.0)
 |  >>> t1 = 10
 |  >>> dt = 1
 |  >>> while r.successful() and r.t < t1:
 |  ...     print(r.t+dt, r.integrate(r.t+dt))
 |  1 [-0.71038232+0.23749653j  0.40000271+0.j        ]
 |  2.0 [0.19098503-0.52359246j 0.22222356+0.j        ]
 |  3.0 [0.47153208+0.52701229j 0.15384681+0.j        ]
 |  4.0 [-0.61905937+0.30726255j  0.11764744+0.j        ]
 |  5.0 [0.02340997-0.61418799j 0.09523835+0.j        ]
 |  6.0 [0.58643071+0.339819j 0.08000018+0.j      ]
 |  7.0 [-0.52070105+0.44525141j  0.06896565+0.j        ]
 |  8.0 [-0.15986733-0.61234476j  0.06060616+0.j        ]
 |  9.0 [0.64850462+0.15048982j 0.05405414+0.j        ]
 |  10.0 [-0.38404699+0.56382299j  0.04878055+0.j        ]
 |
 |  References
 |  ----------
 |  .. [HNW93] E. Hairer, S.P. Norsett and G. Wanner, Solving Ordinary
 |      Differential Equations i. Nonstiff Problems. 2nd edition.
 |      Springer Series in Computational Mathematics,
 |      Springer-Verlag (1993)
 |
 |  Methods defined here:
 |
 |  __init__(self, f, jac=None)
 |      Initialize self.  See help(type(self)) for accurate signature.
 |
 |  get_return_code(self)
 |      Extracts the return code for the integration to enable better control
 |      if the integration fails.
 |
 |      In general, a return code > 0 implies success, while a return code < 0
 |      implies failure.
 |
 |      Notes
 |      -----
 |      This section describes possible return codes and their meaning, for available
 |      integrators that can be selected by `set_integrator` method.
 |
 |      "vode"
 |
 ===========  =======
 Return Code  Message
 ===========  =======
 2            Integration successful.
 -1           Excess work done on this call. (Perhaps wrong MF.)
 -2           Excess accuracy requested. (Tolerances too small.)
 -3           Illegal input detected. (See printed message.)
 -4           Repeated error test failures. (Check all input.)
 -5           Repeated convergence failures. (Perhaps bad Jacobian
              supplied or wrong choice of MF or tolerances.)
 -6           Error weight became zero during problem. (Solution
              component i vanished, and ATOL or ATOL(i) = 0.)
 ===========  =======

 "zvode"

 ===========  =======
 Return Code  Message
 ===========  =======
 2            Integration successful.
 -1           Excess work done on this call. (Perhaps wrong MF.)
 -2           Excess accuracy requested. (Tolerances too small.)
 -3           Illegal input detected. (See printed message.)
 -4           Repeated error test failures. (Check all input.)
 -5           Repeated convergence failures. (Perhaps bad Jacobian
              supplied or wrong choice of MF or tolerances.)
 -6           Error weight became zero during problem. (Solution
              component i vanished, and ATOL or ATOL(i) = 0.)
 ===========  =======

 "dopri5"

 ===========  =======
 Return Code  Message
 ===========  =======
 1            Integration successful.
 2            Integration successful (interrupted by solout).
 -1           Input is not consistent.
 -2           Larger nsteps is needed.
 -3           Step size becomes too small.
 -4           Problem is probably stiff (interrupted).
 ===========  =======

 "dop853"

 ===========  =======
 Return Code  Message
```

```
 |      ===========  =======
 |      1            Integration successful.
 |      2            Integration successful (interrupted by solout).
 |      -1           Input is not consistent.
 |      -2           Larger nsteps is needed.
 |      -3           Step size becomes too small.
 |      -4           Problem is probably stiff (interrupted).
 |      ===========  =======
 |
 |      "lsoda"
 |
 |      ===========  =======
 |      Return Code  Message
 |      ===========  =======
 |      2            Integration successful.
 |      -1           Excess work done on this call (perhaps wrong Dfun type).
 |      -2           Excess accuracy requested (tolerances too small).
 |      -3           Illegal input detected (internal error).
 |      -4           Repeated error test failures (internal error).
 |      -5           Repeated convergence failures (perhaps bad Jacobian or tolerances).
 |      -6           Error weight became zero during problem.
 |      -7           Internal workspace insufficient to finish (internal error).
 |      ===========  =======
 |
 |  integrate(self, t, step=False, relax=False)
 |      Find y=y(t), set y as an initial condition, and return y.
 |
 |      Parameters
 |      ----------
 |      t : float
 |          The endpoint of the integration step.
 |      step : bool
 |          If True, and if the integrator supports the step method,
 |          then perform a single integration step and return.
 |          This parameter is provided in order to expose internals of
 |          the implementation, and should not be changed from its default
 |          value in most cases.
 |      relax : bool
 |          If True and if the integrator supports the run_relax method,
 |          then integrate until t_1 >= t and return. ``relax`` is not
 |          referenced if ``step=True``.
 |          This parameter is provided in order to expose internals of
 |          the implementation, and should not be changed from its default
 |          value in most cases.
 |
 |      Returns
 |      -------
 |      y : float
 |          The integrated value at t
 |
 |  set_f_params(self, *args)
 |      Set extra parameters for user-supplied function f.
 |
 |  set_initial_value(self, y, t=0.0)
 |      Set initial conditions y(t) = y.
 |
 |  set_integrator(self, name, **integrator_params)
 |      Set integrator by name.
 |
 |      Parameters
 |      ----------
 |      name : str
 |          Name of the integrator.
 |      integrator_params
 |          Additional parameters for the integrator.
 |
 |  set_jac_params(self, *args)
 |      Set extra parameters for user-supplied function jac.
 |
 |  set_solout(self, solout)
 |      Set callable to be called at every successful integration step.
 |
 |      Parameters
 |      ----------
 |      solout : callable
 |          ``solout(t, y)`` is called at each internal integrator step,
 |          t is a scalar providing the current independent position
 |          y is the current soloution ``y.shape == (n,)``
 |          solout should return -1 to stop integration
 |          otherwise it should return None or 0
 |
 |  successful(self)
 |      Check if integration was successful.
 |
 |  ----------------------------------------------------------------------
 |  Readonly properties defined here:
 |
 |  y
 |
 |  ----------------------------------------------------------------------
 |  Data descriptors defined here:
 |
 |  __dict__
 |      dictionary for instance variables (if defined)
```

```
     |
     |  __weakref__
     |      list of weak references to the object (if defined)

FUNCTIONS
    cumtrapz(y, x=None, dx=1.0, axis=-1, initial=None)
        `An alias of `cumulative_trapezoid`.

        `cumtrapz` is kept for backwards compatibility. For new code, prefer
        `cumulative_trapezoid` instead.

    cumulative_trapezoid(y, x=None, dx=1.0, axis=-1, initial=None)
        Cumulatively integrate y(x) using the composite trapezoidal rule.

        Parameters
        ----------
        y : array_like
            Values to integrate.
        x : array_like, optional
            The coordinate to integrate along. If None (default), use spacing `dx`
            between consecutive elements in `y`.
        dx : float, optional
            Spacing between elements of `y`. Only used if `x` is None.
        axis : int, optional
            Specifies the axis to cumulate. Default is -1 (last axis).
        initial : scalar, optional
            If given, insert this value at the beginning of the returned result.
            Typically this value should be 0. Default is None, which means no
            value at ``x[0]`` is returned and `res` has one element less than `y`
            along the axis of integration.

        Returns
        -------
        res : ndarray
            The result of cumulative integration of `y` along `axis`.
            If `initial` is None, the shape is such that the axis of integration
            has one less value than `y`. If `initial` is given, the shape is equal
            to that of `y`.

        See Also
        --------
        numpy.cumsum, numpy.cumprod
        quad: adaptive quadrature using QUADPACK
        romberg: adaptive Romberg quadrature
        quadrature: adaptive Gaussian quadrature
        fixed_quad: fixed-order Gaussian quadrature
        dblquad: double integrals
        tplquad: triple integrals
        romb: integrators for sampled data
        ode: ODE integrators
        odeint: ODE integrators

        Examples
        --------
        >>> from scipy import integrate
        >>> import matplotlib.pyplot as plt

        >>> x = np.linspace(-2, 2, num=20)
        >>> y = x
        >>> y_int = integrate.cumulative_trapezoid(y, x, initial=0)
        >>> plt.plot(x, y_int, 'ro', x, y[0] + 0.5 * x**2, 'b-')
        >>> plt.show()

    dblquad(func, a, b, gfun, hfun, args=(), epsabs=1.49e-08, epsrel=1.49e-08)
        Compute a double integral.

        Return the double (definite) integral of ``func(y, x)`` from ``x = a..b``
        and ``y = gfun(x)..hfun(x)``.

        Parameters
        ----------
        func : callable
            A Python function or method of at least two variables: y must be the
            first argument and x the second argument.
        a, b : float
            The limits of integration in x: `a` < `b`
        gfun : callable or float
            The lower boundary curve in y which is a function taking a single
            floating point argument (x) and returning a floating point result
            or a float indicating a constant boundary curve.
        hfun : callable or float
            The upper boundary curve in y (same requirements as `gfun`).
        args : sequence, optional
            Extra arguments to pass to `func`.
        epsabs : float, optional
            Absolute tolerance passed directly to the inner 1-D quadrature
            integration. Default is 1.49e-8. `dblquad`` tries to obtain
            an accuracy of ``abs(i-result) <= max(epsabs, epsrel*abs(i))``
            where ``i`` = inner integral of ``func(y, x)`` from ``gfun(x)``
            to ``hfun(x)``, and ``result`` is the numerical approximation.
            See `epsrel` below.
        epsrel : float, optional
            Relative tolerance of the inner 1-D integrals. Default is 1.49e-8.
            If ``epsabs <= 0``, `epsrel` must be greater than both 5e-29
            and ``50 * (machine epsilon)``. See `epsabs` above.

        Returns
        -------
        y : float
            The resultant integral.
        abserr : float
            An estimate of the error.

        See also
        --------
        quad : single integral
        tplquad : triple integral
        nquad : N-dimensional integrals
        fixed_quad : fixed-order Gaussian quadrature
        quadrature : adaptive Gaussian quadrature
        odeint : ODE integrator
        ode : ODE integrator
        simpson : integrator for sampled data
        romb : integrator for sampled data
        scipy.special : for coefficients and roots of orthogonal polynomials

        Examples
        --------

        Compute the double integral of ``x * y**2`` over the box
        ``x`` ranging from 0 to 2 and ``y`` ranging from 0 to 1.

        >>> from scipy import integrate
        >>> f = lambda y, x: x*y**2
        >>> integrate.dblquad(f, 0, 2, lambda x: 0, lambda x: 1)
            (0.6666666666666667, 7.401486830834377e-15)

    fixed_quad(func, a, b, args=(), n=5)
        Compute a definite integral using fixed-order Gaussian quadrature.

        Integrate `func` from `a` to `b` using Gaussian quadrature of
        order `n`.

        Parameters
        ----------
        func : callable
            A Python function or method to integrate (must accept vector inputs).
            If integrating a vector-valued function, the returned array must have
            shape ``(..., len(x))``.
        a : float
            Lower limit of integration.
        b : float
            Upper limit of integration.
        args : tuple, optional
            Extra arguments to pass to function, if any.
        n : int, optional
            Order of quadrature integration. Default is 5.

        Returns
        -------
        val : float
            Gaussian quadrature approximation to the integral
        none : None
            Statically returned value of None


        See Also
        --------
        quad : adaptive quadrature using QUADPACK
        dblquad : double integrals
        tplquad : triple integrals
        romberg : adaptive Romberg quadrature
        quadrature : adaptive Gaussian quadrature
        romb : integrators for sampled data
        simpson : integrators for sampled data
        cumulative_trapezoid : cumulative integration for sampled data
        ode : ODE integrator
        odeint : ODE integrator

        Examples
        --------
        >>> from scipy import integrate
        >>> f = lambda x: x**8
        >>> integrate.fixed_quad(f, 0.0, 1.0, n=4)
        (0.1110884353741496, None)
        >>> integrate.fixed_quad(f, 0.0, 1.0, n=5)
        (0.11111111111111102, None)
        >>> print(1/9.0)  # analytical result
        0.1111111111111111

        >>> integrate.fixed_quad(np.cos, 0.0, np.pi/2, n=4)
        (0.9999999771971152, None)
        >>> integrate.fixed_quad(np.cos, 0.0, np.pi/2, n=5)
        (1.000000000039565, None)
        >>> np.sin(np.pi/2)-np.sin(0)  # analytical result
        1.0

    newton_cotes(rn, equal=0)
```

```
Return weights and error coefficient for Newton-Cotes integration.

Suppose we have (N+1) samples of f at the positions
x_0, x_1, ..., x_N. Then an N-point Newton-Cotes formula for the
integral between x_0 and x_N is:

:math:`\int_{x_0}^{x_N} f(x)dx = \Delta x \sum_{i=0}^{N} a_i f(x_i)
+ B_N (\Delta x)^{N+2} f^{N+1} (\xi)`

where :math:`\xi \in [x_0,x_N]`
and :math:`\Delta x = \frac{x_N-x_0}{N}` is the average samples spacing.

If the samples are equally-spaced and N is even, then the error
term is :math:`B_N (\Delta x)^{N+3} f^{N+2}(\xi)`.

Parameters
----------
rn : int
    The integer order for equally-spaced data or the relative positions of
    the samples with the first sample at 0 and the last at N, where N+1 is
    the length of `rn`. N is the order of the Newton-Cotes integration.
equal : int, optional
    Set to 1 to enforce equally spaced data.

Returns
-------
an : ndarray
    1-D array of weights to apply to the function at the provided sample
    positions.
B : float
    Error coefficient.

Examples
--------
Compute the integral of sin(x) in [0, :math:`\pi`]:

>>> from scipy.integrate import newton_cotes
>>> def f(x):
...     return np.sin(x)
>>> a = 0
>>> b = np.pi
>>> exact = 2
>>> for N in [2, 4, 6, 8, 10]:
...     x = np.linspace(a, b, N + 1)
...     an, B = newton_cotes(N, 1)
...     dx = (b - a) / N
...     quad = dx * np.sum(an * f(x))
```

```
    ...     error = abs(quad - exact)
    ...     print('{:2d}  {:10.9f}  {:.5e}'.format(N, quad, error))
    ...
     2   2.094395102   9.43951e-02
     4   1.998570732   1.42927e-03
     6   2.000017814   1.78136e-05
     8   1.999999835   1.64725e-07
    10   2.000000001   1.14677e-09

    Notes
    -----
    Normally, the Newton-Cotes rules are used on smaller integration
    regions and a composite rule is used to return the total integral.

nquad(func, ranges, args=None, opts=None, full_output=False)
    Integration over multiple variables.

    Wraps `quad` to enable integration over multiple variables.
    Various options allow improved integration of discontinuous functions, as
    well as the use of weighted integration, and generally finer control of the
    integration process.

    Parameters
    ----------
    func : {callable, scipy.LowLevelCallable}
        The function to be integrated. Has arguments of ``x0, ... xn``,
        ``t0, ... tm``, where integration is carried out over ``x0, ... xn``,
        which must be floats.  Where ```t0, ... tm``` are extra arguments
        passed in args.
        Function signature should be ``func(x0, x1, ..., xn, t0, t1, ..., tm)``.
        Integration is carried out in order.  That is, integration over ``x0``
        is the innermost integral, and ``xn`` is the outermost.

        If the user desires improved integration performance, then `f` may
        be a `scipy.LowLevelCallable` with one of the signatures::

            double func(int n, double *xx)
            double func(int n, double *xx, void *user_data)

        where ``n`` is the number of variables and args.  The ``xx`` array
        contains the coordinates and extra arguments. ``user_data`` is the data
        contained in the `scipy.LowLevelCallable`.
    ranges : iterable object
        Each element of ranges may be either a sequence  of 2 numbers, or else
        a callable that returns such a sequence. ``ranges[0]`` corresponds to
        integration over x0, and so on. If an element of ranges is a callable,
        then it will be called with all of the integration arguments available,
```

```
    as well as any parametric arguments. e.g., if
    ``func = f(x0, x1, x2, t0, t1)``, then ``ranges[0]`` may be defined as
    either ``(a, b)`` or else as ``(a, b) = range0(x1, x2, t0, t1)``.
args : iterable object, optional
    Additional arguments ``t0, ..., tn``, required by `func`, `ranges`, and
    ``opts``.
opts : iterable object or dict, optional
    Options to be passed to `quad`. May be empty, a dict, or
    a sequence of dicts or functions that return a dict. If empty, the
    default options from scipy.integrate.quad are used. If a dict, the same
    options are used for all levels of integraion. If a sequence, then each
    element of the sequence corresponds to a particular integration. e.g.,
    opts[0] corresponds to integration over x0, and so on. If a callable,
    the signature must be the same as for ``ranges``. The available
    options together with their default values are:

      - epsabs = 1.49e-08
      - epsrel = 1.49e-08
      - limit  = 50
      - points = None
      - weight = None
      - wvar   = None
      - wopts  = None

    For more information on these options, see `quad` and `quad_explain`.

full_output : bool, optional
    Partial implementation of ``full_output`` from scipy.integrate.quad.
    The number of integrand function evaluations ``neval`` can be obtained
    by setting ``full_output=True`` when calling nquad.

Returns
-------
result : float
    The result of the integration.
abserr : float
    The maximum of the estimates of the absolute error in the various
    integration results.
out_dict : dict, optional
    A dict containing additional information on the integration.

See Also
--------
quad : 1-D numerical integration
dblquad, tplquad : double and triple integrals
fixed_quad : fixed-order Gaussian quadrature
quadrature : adaptive Gaussian quadrature
```

```
        Examples
        --------
        >>> from scipy import integrate
        >>> func = lambda x0,x1,x2,x3 : x0**2 + x1*x2 - x3**3 + np.sin(x0) + (
        ...                             1 if (x0-.2*x3-.5-.25*x1>0) else 0)
        >>> def opts0(*args, **kwargs):
        ...     return {'points':[0.2*args[2] + 0.5 + 0.25*args[0]]}
        >>> integrate.nquad(func, [[0,1], [-1,1], [.13,.8], [-.15,1]],
        ...                 opts=[opts0,{},{},{}], full_output=True)
        (1.5267454070738633, 2.9437360001402324e-14, {'neval': 388962})

        >>> scale = .1
        >>> def func2(x0, x1, x2, x3, t0, t1):
        ...     return x0*x1*x3**2 + np.sin(x2) + 1 + (1 if x0+t1*x1-t0>0 else 0)
        >>> def lim0(x1, x2, x3, t0, t1):
        ...     return [scale * (x1**2 + x2 + np.cos(x3)*t0*t1 + 1) - 1,
        ...             scale * (x1**2 + x2 + np.cos(x3)*t0*t1 + 1) + 1]
        >>> def lim1(x2, x3, t0, t1):
        ...     return [scale * (t0*x2 + t1*x3) - 1,
        ...             scale * (t0*x2 + t1*x3) + 1]
        >>> def lim2(x3, t0, t1):
        ...     return [scale * (x3 + t0**2*t1**3) - 1,
        ...             scale * (x3 + t0**2*t1**3) + 1]
        >>> def lim3(t0, t1):
        ...     return [scale * (t0+t1) - 1, scale * (t0+t1) + 1]
        >>> def opts0(x1, x2, x3, t0, t1):
        ...     return {'points' : [t0 - t1*x1]}
        >>> def opts1(x2, x3, t0, t1):
        ...     return {}
        >>> def opts2(x3, t0, t1):
        ...     return {}
        >>> def opts3(t0, t1):
        ...     return {}
        >>> integrate.nquad(func2, [lim0, lim1, lim2, lim3], args=(0,0),
        ...                 opts=[opts0, opts1, opts2, opts3])
        (25.066666666666666, 2.7829590483937256e-13)

    odeint(func, y0, t, args=(), Dfun=None, col_deriv=0, full_output=0, ml=None, mu=None, rtol=None, atol=None, tcrit=None, h0=0.
0, hmax=0.0, hmin=0.0, ixpr=0, mxstep=0, mxhnil=0, mxordn=12, mxords=5, printmessg=0, tfirst=False)
        Integrate a system of ordinary differential equations.

        .. note:: For new code, use `scipy.integrate.solve_ivp` to solve a
                  differential equation.

        Solve a system of ordinary differential equations using lsoda from the
        FORTRAN library odepack.
```

```
Solves the initial value problem for stiff or non-stiff systems
of first order ode-s::

    dy/dt = func(y, t, ...)  [or func(t, y, ...)]

where y can be a vector.

.. note:: By default, the required order of the first two arguments of
          `func` are in the opposite order of the arguments in the system
          definition function used by the `scipy.integrate.ode` class and
          the function `scipy.integrate.solve_ivp`. To use a function with
          the signature ``func(t, y, ...)``, the argument `tfirst` must be
          set to ``True``.

Parameters
----------
func : callable(y, t, ...) or callable(t, y, ...)
    Computes the derivative of y at t.
    If the signature is ``callable(t, y, ...)``, then the argument
    `tfirst` must be set ``True``.
y0 : array
    Initial condition on y (can be a vector).
t : array
    A sequence of time points for which to solve for y. The initial
    value point should be the first element of this sequence.
    This sequence must be monotonically increasing or monotonically
    decreasing; repeated values are allowed.
args : tuple, optional
    Extra arguments to pass to function.
Dfun : callable(y, t, ...) or callable(t, y, ...)
    Gradient (Jacobian) of `func`.
    If the signature is ``callable(t, y, ...)``, then the argument
    `tfirst` must be set ``True``.
col_deriv : bool, optional
    True if `Dfun` defines derivatives down columns (faster),
    otherwise `Dfun` should define derivatives across rows.
full_output : bool, optional
    True if to return a dictionary of optional outputs as the second output
printmessg : bool, optional
    Whether to print the convergence message
tfirst: bool, optional
    If True, the first two arguments of `func` (and `Dfun`, if given)
    must ``t, y`` instead of the default ``y, t``.

    .. versionadded:: 1.1.0
```

```
Returns
-------
y : array, shape (len(t), len(y0))
    Array containing the value of y for each desired time in t,
    with the initial value `y0` in the first row.
infodict : dict, only returned if full_output == True
    Dictionary containing additional output information

    =======  ============================================================
    key      meaning
    =======  ============================================================
    'hu'     vector of step sizes successfully used for each time step
    'tcur'   vector with the value of t reached for each time step
             (will always be at least as large as the input times)
    'tolsf'  vector of tolerance scale factors, greater than 1.0,
             computed when a request for too much accuracy was detected
    'tsw'    value of t at the time of the last method switch
             (given for each time step)
    'nst'    cumulative number of time steps
    'nfe'    cumulative number of function evaluations for each time step
    'nje'    cumulative number of jacobian evaluations for each time step
    'nqu'    a vector of method orders for each successful step
    'imxer'  index of the component of largest magnitude in the
             weighted local error vector (e / ewt) on an error return, -1
             otherwise
    'lenrw'  the length of the double work array required
    'leniw'  the length of integer work array required
    'mused'  a vector of method indicators for each successful time step:
             1: adams (nonstiff), 2: bdf (stiff)
    =======  ============================================================

Other Parameters
----------------
ml, mu : int, optional
    If either of these are not None or non-negative, then the
    Jacobian is assumed to be banded. These give the number of
    lower and upper non-zero diagonals in this banded matrix.
    For the banded case, `Dfun` should return a matrix whose
    rows contain the non-zero bands (starting with the lowest diagonal).
    Thus, the return matrix `jac` from `Dfun` should have shape
    ``(ml + mu + 1, len(y0))`` when ``ml >=0`` or ``mu >=0``.
    The data in `jac` must be stored such that ``jac[i - j + mu, j]``
    holds the derivative of the `i`th equation with respect to the `j`th
    state variable.  If `col_deriv` is True, the transpose of this
    `jac` must be returned.
rtol, atol : float, optional
    The input parameters `rtol` and `atol` determine the error
    control performed by the solver.  The solver will control the
    vector, e, of estimated local errors in y, according to an
    inequality of the form ``max-norm of (e / ewt) <= 1``,
    where ewt is a vector of positive error weights computed as
    ``ewt = rtol * abs(y) + atol``.
    rtol and atol can be either vectors the same length as y or scalars.
    Defaults to 1.49012e-8.
tcrit : ndarray, optional
    Vector of critical points (e.g., singularities) where integration
    care should be taken.
h0 : float, (0: solver-determined), optional
    The step size to be attempted on the first step.
hmax : float, (0: solver-determined), optional
    The maximum absolute step size allowed.
hmin : float, (0: solver-determined), optional
    The minimum absolute step size allowed.
ixpr : bool, optional
    Whether to generate extra printing at method switches.
mxstep : int, (0: solver-determined), optional
    Maximum number of (internally defined) steps allowed for each
    integration point in t.
mxhnil : int, (0: solver-determined), optional
    Maximum number of messages printed.
mxordn : int, (0: solver-determined), optional
    Maximum order to be allowed for the non-stiff (Adams) method.
mxords : int, (0: solver-determined), optional
    Maximum order to be allowed for the stiff (BDF) method.

See Also
--------
solve_ivp : solve an initial value problem for a system of ODEs
ode : a more object-oriented integrator based on VODE
quad : for finding the area under a curve

Examples
--------
The second order differential equation for the angle `theta` of a
pendulum acted on by gravity with friction can be written::

    theta''(t) + b*theta'(t) + c*sin(theta(t)) = 0

where `b` and `c` are positive constants, and a prime (') denotes a
derivative. To solve this equation with `odeint`, we must first convert
it to a system of first order equations. By defining the angular
velocity ``omega(t) = theta'(t)``, we obtain the system::

    theta'(t) = omega(t)
```

```
        omega'(t) = -b*omega(t) - c*sin(theta(t))

    Let `y` be the vector [`theta`, `omega`]. We implement this system
    in Python as:

    >>> def pend(y, t, b, c):
    ...     theta, omega = y
    ...     dydt = [omega, -b*omega - c*np.sin(theta)]
    ...     return dydt
    ...

    We assume the constants are `b` = 0.25 and `c` = 5.0:

    >>> b = 0.25
    >>> c = 5.0

    For initial conditions, we assume the pendulum is nearly vertical
    with `theta(0)` = `pi` - 0.1, and is initially at rest, so
    `omega(0)` = 0.  Then the vector of initial conditions is

    >>> y0 = [np.pi - 0.1, 0.0]

    We will generate a solution at 101 evenly spaced samples in the interval
    0 <= `t` <= 10.  So our array of times is:

    >>> t = np.linspace(0, 10, 101)

    Call `odeint` to generate the solution. To pass the parameters
    `b` and `c` to `pend`, we give them to `odeint` using the `args`
    argument.

    >>> from scipy.integrate import odeint
    >>> sol = odeint(pend, y0, t, args=(b, c))

    The solution is an array with shape (101, 2). The first column
    is `theta(t)`, and the second is `omega(t)`. The following code
    plots both components.

    >>> import matplotlib.pyplot as plt
    >>> plt.plot(t, sol[:, 0], 'b', label='theta(t)')
    >>> plt.plot(t, sol[:, 1], 'g', label='omega(t)')
    >>> plt.legend(loc='best')
    >>> plt.xlabel('t')
    >>> plt.grid()
    >>> plt.show()

quad(func, a, b, args=(), full_output=0, epsabs=1.49e-08, epsrel=1.49e-08, limit=50, points=None, weight=None, wvar=None, wopt
s=None, maxp1=50, limlst=50)
        Compute a definite integral.

        Integrate func from `a` to `b` (possibly infinite interval) using a
        technique from the Fortran library QUADPACK.

        Parameters
        ----------
        func : {function, scipy.LowLevelCallable}
            A Python function or method to integrate. If `func` takes many
            arguments, it is integrated along the axis corresponding to the
            first argument.

            If the user desires improved integration performance, then `f` may
            be a `scipy.LowLevelCallable` with one of the signatures::

                double func(double x)
                double func(double x, void *user_data)
                double func(int n, double *xx)
                double func(int n, double *xx, void *user_data)

            The ``user_data`` is the data contained in the `scipy.LowLevelCallable`.
            In the call forms with ``xx``,  ``n`` is the length of the ``xx``
            array which contains ``xx[0] == x`` and the rest of the items are
            numbers contained in the ``args`` argument of quad.

            In addition, certain ctypes call signatures are supported for
            backward compatibility, but those should not be used in new code.
        a : float
            Lower limit of integration (use -numpy.inf for -infinity).
        b : float
            Upper limit of integration (use numpy.inf for +infinity).
        args : tuple, optional
            Extra arguments to pass to `func`.
        full_output : int, optional
            Non-zero to return a dictionary of integration information.
            If non-zero, warning messages are also suppressed and the
            message is appended to the output tuple.

        Returns
        -------
        y : float
            The integral of func from `a` to `b`.
        abserr : float
            An estimate of the absolute error in the result.
        infodict : dict
            A dictionary containing additional information.
```

```
    Run scipy.integrate.quad_explain() for more information.
message
    A convergence message.
explain
    Appended only with 'cos' or 'sin' weighting and infinite
    integration limits, it contains an explanation of the codes in
    infodict['ierlst']

Other Parameters
----------------
epsabs : float or int, optional
    Absolute error tolerance. Default is 1.49e-8. `quad` tries to obtain
    an accuracy of ``abs(i-result) <= max(epsabs, epsrel*abs(i))``
    where ``i`` = integral of `func` from `a` to `b`, and ``result`` is the
    numerical approximation. See `epsrel` below.
epsrel : float or int, optional
    Relative error tolerance. Default is 1.49e-8.
    If ``epsabs <= 0``, `epsrel` must be greater than both 5e-29
    and ``50 * (machine epsilon)``. See `epsabs` above.
limit : float or int, optional
    An upper bound on the number of subintervals used in the adaptive
    algorithm.
points : (sequence of floats,ints), optional
    A sequence of break points in the bounded integration interval
    where local difficulties of the integrand may occur (e.g.,
    singularities, discontinuities). The sequence does not have
    to be sorted. Note that this option cannot be used in conjunction
    with ``weight``.
weight : float or int, optional
    String indicating weighting function. Full explanation for this
    and the remaining arguments can be found below.
wvar : optional
    Variables for use with weighting functions.
wopts : optional
    Optional input for reusing Chebyshev moments.
maxp1 : float or int, optional
    An upper bound on the number of Chebyshev moments.
limlst : int, optional
    Upper bound on the number of cycles (>=3) for use with a sinusoidal
    weighting and an infinite end-point.

See Also
--------
dblquad : double integral
tplquad : triple integral
nquad : n-dimensional integrals (uses `quad` recursively)
fixed_quad : fixed-order Gaussian quadrature
```

```
quadrature : adaptive Gaussian quadrature
odeint : ODE integrator
ode : ODE integrator
simpson : integrator for sampled data
romb : integrator for sampled data
scipy.special : for coefficients and roots of orthogonal polynomials

Notes
-----

**Extra information for quad() inputs and outputs**

If full_output is non-zero, then the third output argument
(infodict) is a dictionary with entries as tabulated below. For
infinite limits, the range is transformed to (0,1) and the
optional outputs are given with respect to this transformed range.
Let M be the input argument limit and let K be infodict['last'].
The entries are:

'neval'
    The number of function evaluations.
'last'
    The number, K, of subintervals produced in the subdivision process.
'alist'
    A rank-1 array of length M, the first K elements of which are the
    left end points of the subintervals in the partition of the
    integration range.
'blist'
    A rank-1 array of length M, the first K elements of which are the
    right end points of the subintervals.
'rlist'
    A rank-1 array of length M, the first K elements of which are the
    integral approximations on the subintervals.
'elist'
    A rank-1 array of length M, the first K elements of which are the
    moduli of the absolute error estimates on the subintervals.
'iord'
    A rank-1 integer array of length M, the first L elements of
    which are pointers to the error estimates over the subintervals
    with ``L=K`` if ``K<=M/2+2`` or ``L=M+1-K`` otherwise. Let I be the
    sequence ``infodict['iord']`` and let E be the sequence
    ``infodict['elist']``.  Then ``E[I[1]], ..., E[I[L]]`` forms a
    decreasing sequence.

If the input argument points is provided (i.e., it is not None),
the following additional outputs are placed in the output
dictionary. Assume the points sequence is of length P.
```

'pts'
    A rank-1 array of length P+2 containing the integration limits
    and the break points of the intervals in ascending order.
    This is an array giving the subintervals over which integration
    will occur.
'level'
    A rank-1 integer array of length M (=limit), containing the
    subdivision levels of the subintervals, i.e., if (aa,bb) is a
    subinterval of ``(pts[1], pts[2])`` where ``pts[0]`` and ``pts[2]``
    are adjacent elements of ``infodict['pts']``, then (aa,bb) has level l
    if ``|bb-aa| = |pts[2]-pts[1]| * 2**(-l)``.
'ndin'
    A rank-1 integer array of length P+2. After the first integration
    over the intervals (pts[1], pts[2]), the error estimates over some
    of the intervals may have been increased artificially in order to
    put their subdivision forward. This array has ones in slots
    corresponding to the subintervals for which this happens.

Weighting the integrand

The input variables, *weight* and *wvar*, are used to weight the
integrand by a select list of functions. Different integration
methods are used to compute the integral with these weighting
functions, and these do not support specifying break points. The
possible values of weight and the corresponding weighting functions are.

``weight``	Weight function used	``wvar``
'cos'	cos(w*x)	wvar = w
'sin'	sin(w*x)	wvar = w
'alg'	g(x) = ((x-a)**alpha)*((b-x)**beta)	wvar = (alpha, beta)
'alg-loga'	g(x)*log(x-a)	wvar = (alpha, beta)
'alg-logb'	g(x)*log(b-x)	wvar = (alpha, beta)
'alg-log'	g(x)*log(x-a)*log(b-x)	wvar = (alpha, beta)
'cauchy'	1/(x-c)	wvar = c

wvar holds the parameter w, (alpha, beta), or c depending on the weight
selected. In these expressions, a and b are the integration limits.

For the 'cos' and 'sin' weighting, additional inputs and outputs are
available.

For finite integration limits, the integration is performed using a
Clenshaw-Curtis method which uses Chebyshev moments. For repeated

```
calculations, these moments are saved in the output dictionary:

'momcom'
    The maximum level of Chebyshev moments that have been computed,
    i.e., if ``M_c`` is ``infodict['momcom']`` then the moments have been
    computed for intervals of length ``|b-a| * 2**(-l)``,
    ``l=0,1,...,M_c``.
'nnlog'
    A rank-1 integer array of length M(=limit), containing the
    subdivision levels of the subintervals, i.e., an element of this
    array is equal to l if the corresponding subinterval is
    ``|b-a|* 2**(-l)``.
'chebmo'
    A rank-2 array of shape (25, maxp1) containing the computed
    Chebyshev moments. These can be passed on to an integration
    over the same interval by passing this array as the second
    element of the sequence wopts and passing infodict['momcom'] as
    the first element.

If one of the integration limits is infinite, then a Fourier integral is
computed (assuming w neq 0). If full_output is 1 and a numerical error
is encountered, besides the error message attached to the output tuple,
a dictionary is also appended to the output tuple which translates the
error codes in the array ``info['ierlst']`` to English messages. The
output information dictionary contains the following entries instead of
'last', 'alist', 'blist', 'rlist', and 'elist':

'lst'
    The number of subintervals needed for the integration (call it ``K_f``).
'rslst'
    A rank-1 array of length M_f=limlst, whose first ``K_f`` elements
    contain the integral contribution over the interval
    ``(a+(k-1)c, a+kc)`` where ``c = (2*floor(|w|) + 1) * pi / |w|``
    and ``k=1,2,...,K_f``.
'erlst'
    A rank-1 array of length ``M_f`` containing the error estimate
    corresponding to the interval in the same position in
    ``infodict['rslist']``.
'ierlst'
    A rank-1 integer array of length ``M_f`` containing an error flag
    corresponding to the interval in the same position in
    ``infodict['rslist']``.  See the explanation dictionary (last entry
    in the output tuple) for the meaning of the codes.

Examples
--------
Calculate :math:`\int^4_0 x^2 dx` and compare with an analytic result
```

```
>>> from scipy import integrate
>>> x2 = lambda x: x**2
>>> integrate.quad(x2, 0, 4)
(21.333333333333332, 2.3684757858670003e-13)
>>> print(4**3 / 3.)  # analytical result
21.3333333333
```

Calculate :math:`\int^\infty_0 e^{-x} dx`

```
>>> invexp = lambda x: np.exp(-x)
>>> integrate.quad(invexp, 0, np.inf)
(1.0, 5.842605999138044e-11)

>>> f = lambda x,a : a*x
>>> y, err = integrate.quad(f, 0, 1, args=(1,))
>>> y
0.5
>>> y, err = integrate.quad(f, 0, 1, args=(3,))
>>> y
1.5
```

Calculate :math:`\int^1_0 x^2 + y^2 dx` with ctypes, holding
y parameter as 1::

```
    testlib.c =>
        double func(int n, double args[n]){
            return args[0]*args[0] + args[1]*args[1];}
    compile to library testlib.*
```

::

```
   from scipy import integrate
   import ctypes
   lib = ctypes.CDLL('/home/.../testlib.*') #use absolute path
   lib.func.restype = ctypes.c_double
   lib.func.argtypes = (ctypes.c_int,ctypes.c_double)
   integrate.quad(lib.func,0,1,(1))
   #(1.3333333333333333, 1.4802973661668752e-14)
   print((1.0**3/3.0 + 1.0) - (0.0**3/3.0 + 0.0)) #Analytic result
   # 1.3333333333333333
```

Be aware that pulse shapes and other sharp features as compared to the
size of the integration interval may not be integrated correctly using
this method. A simplified example of this limitation is integrating a
y-axis reflected step function with many zero values within the integrals
bounds.

```
        >>> y = lambda x: 1 if x<=0 else 0
        >>> integrate.quad(y, -1, 1)
        (1.0, 1.1102230246251565e-14)
        >>> integrate.quad(y, -1, 100)
        (1.0000000002199108, 1.0189464580163188e-08)
        >>> integrate.quad(y, -1, 10000)
        (0.0, 0.0)

    quad_explain(output=<ipykernel.iostream.OutStream object at 0x000001DD78B48C10>)
        Print extra information about integrate.quad() parameters and returns.

        Parameters
        ----------
        output : instance with "write" method, optional
            Information about `quad` is passed to ``output.write()``.
            Default is ``sys.stdout``.

        Returns
        -------
        None

        Examples
        --------
        We can show detailed information of the `integrate.quad` function in stdout:

        >>> from scipy.integrate import quad_explain
        >>> quad_explain()

    quad_vec(f, a, b, epsabs=1e-200, epsrel=1e-08, norm='2', cache_size=100000000.0, limit=10000, workers=1, points=None, quadratu
re=None, full_output=False)
        Adaptive integration of a vector-valued function.

        Parameters
        ----------
        f : callable
            Vector-valued function f(x) to integrate.
        a : float
            Initial point.
        b : float
            Final point.
        epsabs : float, optional
            Absolute tolerance.
        epsrel : float, optional
            Relative tolerance.
        norm : {'max', '2'}, optional
            Vector norm to use for error estimation.
```

```
cache_size : int, optional
    Number of bytes to use for memoization.
workers : int or map-like callable, optional
    If `workers` is an integer, part of the computation is done in
    parallel subdivided to this many tasks (using
    :class:`python:multiprocessing.pool.Pool`).
    Supply `-1` to use all cores available to the Process.
    Alternatively, supply a map-like callable, such as
    :meth:`python:multiprocessing.pool.Pool.map` for evaluating the
    population in parallel.
    This evaluation is carried out as ``workers(func, iterable)``.
points : list, optional
    List of additional breakpoints.
quadrature : {'gk21', 'gk15', 'trapezoid'}, optional
    Quadrature rule to use on subintervals.
    Options: 'gk21' (Gauss-Kronrod 21-point rule),
    'gk15' (Gauss-Kronrod 15-point rule),
    'trapezoid' (composite trapezoid rule).
    Default: 'gk21' for finite intervals and 'gk15' for (semi-)infinite
full_output : bool, optional
    Return an additional ``info`` dictionary.

Returns
-------
res : {float, array-like}
    Estimate for the result
err : float
    Error estimate for the result in the given norm
info : dict
    Returned only when ``full_output=True``.
    Info dictionary. Is an object with the attributes:

        success : bool
            Whether integration reached target precision.
        status : int
            Indicator for convergence, success (0),
            failure (1), and failure due to rounding error (2).
        neval : int
            Number of function evaluations.
        intervals : ndarray, shape (num_intervals, 2)
            Start and end points of subdivision intervals.
        integrals : ndarray, shape (num_intervals, ...)
            Integral for each interval.
            Note that at most ``cache_size`` values are recorded,
            and the array may contains *nan* for missing items.
        errors : ndarray, shape (num_intervals,)
            Estimated integration error for each interval.
    Notes
    -----
    The algorithm mainly follows the implementation of QUADPACK's
    DQAG* algorithms, implementing global error control and adaptive
    subdivision.

    The algorithm here has some differences to the QUADPACK approach:

    Instead of subdividing one interval at a time, the algorithm
    subdivides N intervals with largest errors at once. This enables
    (partial) parallelization of the integration.

    The logic of subdividing "next largest" intervals first is then
    not implemented, and we rely on the above extension to avoid
    concentrating on "small" intervals only.

    The Wynn epsilon table extrapolation is not used (QUADPACK uses it
    for infinite intervals). This is because the algorithm here is
    supposed to work on vector-valued functions, in an user-specified
    norm, and the extension of the epsilon algorithm to this case does
    not appear to be widely agreed. For max-norm, using elementwise
    Wynn epsilon could be possible, but we do not do this here with
    the hope that the epsilon extrapolation is mainly useful in
    special cases.

    References
    ----------
    [1] R. Piessens, E. de Doncker, QUADPACK (1983).

    Examples
    --------
    We can compute integrations of a vector-valued function:

    >>> from scipy.integrate import quad_vec
    >>> import matplotlib.pyplot as plt
    >>> alpha = np.linspace(0.0, 2.0, num=30)
    >>> f = lambda x: x**alpha
    >>> x0, x1 = 0, 2
    >>> y, err = quad_vec(f, x0, x1)
    >>> plt.plot(alpha, y)
    >>> plt.xlabel(r"$\alpha$")
    >>> plt.ylabel(r"$\int_{0}^{2} x^\alpha dx$")
    >>> plt.show()

quadrature(func, a, b, args=(), tol=1.49e-08, rtol=1.49e-08, maxiter=50, vec_func=True, miniter=1)
    Compute a definite integral using fixed-tolerance Gaussian quadrature.
```

```
Integrate `func` from `a` to `b` using Gaussian quadrature
with absolute tolerance `tol`.

Parameters
----------
func : function
    A Python function or method to integrate.
a : float
    Lower limit of integration.
b : float
    Upper limit of integration.
args : tuple, optional
    Extra arguments to pass to function.
tol, rtol : float, optional
    Iteration stops when error between last two iterates is less than
    `tol` OR the relative change is less than `rtol`.
maxiter : int, optional
    Maximum order of Gaussian quadrature.
vec_func : bool, optional
    True or False if func handles arrays as arguments (is
    a "vector" function). Default is True.
miniter : int, optional
    Minimum order of Gaussian quadrature.

Returns
-------
val : float
    Gaussian quadrature approximation (within tolerance) to integral.
err : float
    Difference between last two estimates of the integral.

See also
--------
romberg: adaptive Romberg quadrature
fixed_quad: fixed-order Gaussian quadrature
quad: adaptive quadrature using QUADPACK
dblquad: double integrals
tplquad: triple integrals
romb: integrator for sampled data
simpson: integrator for sampled data
cumulative_trapezoid: cumulative integration for sampled data
ode: ODE integrator
odeint: ODE integrator

Examples
--------
```

```
    >>> from scipy import integrate
    >>> f = lambda x: x**8
    >>> integrate.quadrature(f, 0.0, 1.0)
    (0.11111111111111106, 4.163336342344337e-17)
    >>> print(1/9.0)  # analytical result
    0.1111111111111111

    >>> integrate.quadrature(np.cos, 0.0, np.pi/2)
    (0.9999999999999536, 3.9611425250996035e-11)
    >>> np.sin(np.pi/2)-np.sin(0)  # analytical result
    1.0

romb(y, dx=1.0, axis=-1, show=False)
    Romberg integration using samples of a function.

    Parameters
    ----------
    y : array_like
        A vector of ``2**k + 1`` equally-spaced samples of a function.
    dx : float, optional
        The sample spacing. Default is 1.
    axis : int, optional
        The axis along which to integrate. Default is -1 (last axis).
    show : bool, optional
        When `y` is a single 1-D array, then if this argument is True
        print the table showing Richardson extrapolation from the
        samples. Default is False.

    Returns
    -------
    romb : ndarray
        The integrated result for `axis`.

    See also
    --------
    quad : adaptive quadrature using QUADPACK
    romberg : adaptive Romberg quadrature
    quadrature : adaptive Gaussian quadrature
    fixed_quad : fixed-order Gaussian quadrature
    dblquad : double integrals
    tplquad : triple integrals
    simpson : integrators for sampled data
    cumulative_trapezoid : cumulative integration for sampled data
    ode : ODE integrators
    odeint : ODE integrators

    Examples
    --------
    >>> from scipy import integrate
    >>> x = np.arange(10, 14.25, 0.25)
    >>> y = np.arange(3, 12)

    >>> integrate.romb(y)
    56.0

    >>> y = np.sin(np.power(x, 2.5))
    >>> integrate.romb(y)
    -0.742561336672229

    >>> integrate.romb(y, show=True)
    Richardson Extrapolation Table for Romberg Integration
    ====================================================================
    -0.81576
    4.63862  6.45674
    -1.10581 -3.02062 -3.65245
    -2.57379 -3.06311 -3.06595 -3.05664
    -1.34093 -0.92997 -0.78776 -0.75160 -0.74256
    ====================================================================
    -0.742561336672229

romberg(function, a, b, args=(), tol=1.48e-08, rtol=1.48e-08, show=False, divmax=10, vec_func=False)
    Romberg integration of a callable function or method.

    Returns the integral of `function` (a function of one variable)
    over the interval (`a`, `b`).

    If `show` is 1, the triangular array of the intermediate results
    will be printed. If `vec_func` is True (default is False), then
    `function` is assumed to support vector arguments.

    Parameters
    ----------
    function : callable
        Function to be integrated.
    a : float
        Lower limit of integration.
    b : float
        Upper limit of integration.

    Returns
    -------
    results  : float
        Result of the integration.
```

```
Other Parameters
----------------
args : tuple, optional
    Extra arguments to pass to function. Each element of `args` will
    be passed as a single argument to `func`. Default is to pass no
    extra arguments.
tol, rtol : float, optional
    The desired absolute and relative tolerances. Defaults are 1.48e-8.
show : bool, optional
    Whether to print the results. Default is False.
divmax : int, optional
    Maximum order of extrapolation. Default is 10.
vec_func : bool, optional
    Whether `func` handles arrays as arguments (i.e., whether it is a
    "vector" function). Default is False.

See Also
--------
fixed_quad : Fixed-order Gaussian quadrature.
quad : Adaptive quadrature using QUADPACK.
dblquad : Double integrals.
tplquad : Triple integrals.
romb : Integrators for sampled data.
simpson : Integrators for sampled data.
cumulative_trapezoid : Cumulative integration for sampled data.
ode : ODE integrator.
odeint : ODE integrator.

References
----------
.. [1] 'Romberg's method' https://en.wikipedia.org/wiki/Romberg%27s_method

Examples
--------
Integrate a gaussian from 0 to 1 and compare to the error function.

>>> from scipy import integrate
>>> from scipy.special import erf
>>> gaussian = lambda x: 1/np.sqrt(np.pi) * np.exp(-x**2)
>>> result = integrate.romberg(gaussian, 0, 1, show=True)
Romberg integration of <function vfunc at ...> from [0, 1]

::

   Steps  StepSize  Results
       1  1.000000  0.385872
       2  0.500000  0.412631  0.421551
```

```
            4  0.250000  0.419184  0.421368  0.421356
            8  0.125000  0.420810  0.421352  0.421350  0.421350
           16  0.062500  0.421215  0.421350  0.421350  0.421350  0.421350
           32  0.031250  0.421317  0.421350  0.421350  0.421350  0.421350  0.421350

    The final result is 0.421350396475 after 33 function evaluations.

    >>> print("%g %g" % (2*result, erf(1)))
    0.842701 0.842701

simps(y, x=None, dx=1, axis=-1, even='avg')
    `An alias of `simpson`.

    `simps` is kept for backwards compatibility. For new code, prefer
    `simpson` instead.

simpson(y, x=None, dx=1, axis=-1, even='avg')
    Integrate y(x) using samples along the given axis and the composite
    Simpson's rule. If x is None, spacing of dx is assumed.

    If there are an even number of samples, N, then there are an odd
    number of intervals (N-1), but Simpson's rule requires an even number
    of intervals. The parameter 'even' controls how this is handled.

    Parameters
    ----------
    y : array_like
        Array to be integrated.
    x : array_like, optional
        If given, the points at which `y` is sampled.
    dx : int, optional
        Spacing of integration points along axis of `x`. Only used when
        `x` is None. Default is 1.
    axis : int, optional
        Axis along which to integrate. Default is the last axis.
    even : str {'avg', 'first', 'last'}, optional
        'avg' : Average two results:1) use the first N-2 intervals with
                  a trapezoidal rule on the last interval and 2) use the last
                  N-2 intervals with a trapezoidal rule on the first interval.

        'first' : Use Simpson's rule for the first N-2 intervals with
                a trapezoidal rule on the last interval.

        'last' : Use Simpson's rule for the last N-2 intervals with a
               trapezoidal rule on the first interval.

    See Also
    --------
    quad: adaptive quadrature using QUADPACK
    romberg: adaptive Romberg quadrature
    quadrature: adaptive Gaussian quadrature
    fixed_quad: fixed-order Gaussian quadrature
    dblquad: double integrals
    tplquad: triple integrals
    romb: integrators for sampled data
    cumulative_trapezoid: cumulative integration for sampled data
    ode: ODE integrators
    odeint: ODE integrators

    Notes
    -----
    For an odd number of samples that are equally spaced the result is
    exact if the function is a polynomial of order 3 or less. If
    the samples are not equally spaced, then the result is exact only
    if the function is a polynomial of order 2 or less.

    Examples
    --------
    >>> from scipy import integrate
    >>> x = np.arange(0, 10)
    >>> y = np.arange(0, 10)

    >>> integrate.simpson(y, x)
    40.5

    >>> y = np.power(x, 3)
    >>> integrate.simpson(y, x)
    1642.5
    >>> integrate.quad(lambda x: x**3, 0, 9)[0]
    1640.25

    >>> integrate.simpson(y, x, even='first')
    1644.5

solve_bvp(fun, bc, x, y, p=None, S=None, fun_jac=None, bc_jac=None, tol=0.001, max_nodes=1000, verbose=0, bc_tol=None)
    Solve a boundary value problem for a system of ODEs.

    This function numerically solves a first order system of ODEs subject to
    two-point boundary conditions::

        dy / dx = f(x, y, p) + S * y / (x - a), a <= x <= b
        bc(y(a), y(b), p) = 0

    Here x is a 1-D independent variable, y(x) is an N-D
```

```
vector-valued function and p is a k-D vector of unknown
parameters which is to be found along with y(x). For the problem to be
determined, there must be n + k boundary conditions, i.e., bc must be an
(n + k)-D function.

The last singular term on the right-hand side of the system is optional.
It is defined by an n-by-n matrix S, such that the solution must satisfy
S y(a) = 0. This condition will be forced during iterations, so it must not
contradict boundary conditions. See [2]_ for the explanation how this term
is handled when solving BVPs numerically.

Problems in a complex domain can be solved as well. In this case, y and p
are considered to be complex, and f and bc are assumed to be complex-valued
functions, but x stays real. Note that f and bc must be complex
differentiable (satisfy Cauchy-Riemann equations [4]_), otherwise you
should rewrite your problem for real and imaginary parts separately. To
solve a problem in a complex domain, pass an initial guess for y with a
complex data type (see below).

Parameters
----------
fun : callable
    Right-hand side of the system. The calling signature is ``fun(x, y)``,
    or ``fun(x, y, p)`` if parameters are present. All arguments are
    ndarray: ``x`` with shape (m,), ``y`` with shape (n, m), meaning that
    ``y[:, i]`` corresponds to ``x[i]``, and ``p`` with shape (k,). The
    return value must be an array with shape (n, m) and with the same
    layout as ``y``.
bc : callable
    Function evaluating residuals of the boundary conditions. The calling
    signature is ``bc(ya, yb)``, or ``bc(ya, yb, p)`` if parameters are
    present. All arguments are ndarray: ``ya`` and ``yb`` with shape (n,),
    and ``p`` with shape (k,). The return value must be an array with
    shape (n + k,).
x : array_like, shape (m,)
    Initial mesh. Must be a strictly increasing sequence of real numbers
    with ``x[0]=a`` and ``x[-1]=b``.
y : array_like, shape (n, m)
    Initial guess for the function values at the mesh nodes, ith column
    corresponds to ``x[i]``. For problems in a complex domain pass `y`
    with a complex data type (even if the initial guess is purely real).
p : array_like with shape (k,) or None, optional
    Initial guess for the unknown parameters. If None (default), it is
    assumed that the problem doesn't depend on any parameters.
S : array_like with shape (n, n) or None
    Matrix defining the singular term. If None (default), the problem is
    solved without the singular term.
```

```
fun_jac : callable or None, optional
    Function computing derivatives of f with respect to y and p. The
    calling signature is ``fun_jac(x, y)``, or ``fun_jac(x, y, p)`` if
    parameters are present. The return must contain 1 or 2 elements in the
    following order:

        * df_dy : array_like with shape (n, n, m), where an element
          (i, j, q) equals to d f_i(x_q, y_q, p) / d (y_q)_j.
        * df_dp : array_like with shape (n, k, m), where an element
          (i, j, q) equals to d f_i(x_q, y_q, p) / d p_j.

    Here q numbers nodes at which x and y are defined, whereas i and j
    number vector components. If the problem is solved without unknown
    parameters, df_dp should not be returned.

    If `fun_jac` is None (default), the derivatives will be estimated
    by the forward finite differences.
bc_jac : callable or None, optional
    Function computing derivatives of bc with respect to ya, yb, and p.
    The calling signature is ``bc_jac(ya, yb)``, or ``bc_jac(ya, yb, p)``
    if parameters are present. The return must contain 2 or 3 elements in
    the following order:

        * dbc_dya : array_like with shape (n, n), where an element (i, j)
          equals to d bc_i(ya, yb, p) / d ya_j.
        * dbc_dyb : array_like with shape (n, n), where an element (i, j)
          equals to d bc_i(ya, yb, p) / d yb_j.
        * dbc_dp : array_like with shape (n, k), where an element (i, j)
          equals to d bc_i(ya, yb, p) / d p_j.

    If the problem is solved without unknown parameters, dbc_dp should not
    be returned.

    If `bc_jac` is None (default), the derivatives will be estimated by
    the forward finite differences.
tol : float, optional
    Desired tolerance of the solution. If we define ``r = y' - f(x, y)``,
    where y is the found solution, then the solver tries to achieve on each
    mesh interval ``norm(r / (1 + abs(f)) < tol``, where ``norm`` is
    estimated in a root mean squared sense (using a numerical quadrature
    formula). Default is 1e-3.
max_nodes : int, optional
    Maximum allowed number of the mesh nodes. If exceeded, the algorithm
    terminates. Default is 1000.
verbose : {0, 1, 2}, optional
    Level of algorithm's verbosity:
        * 0 (default) : work silently.
        * 1 : display a termination report.
        * 2 : display progress during iterations.
bc_tol : float, optional
    Desired absolute tolerance for the boundary condition residuals: `bc`
    value should satisfy ``abs(bc) < bc_tol`` component-wise.
    Equals to `tol` by default. Up to 10 iterations are allowed to achieve this
    tolerance.

Returns
-------
Bunch object with the following fields defined:
sol : PPoly
    Found solution for y as `scipy.interpolate.PPoly` instance, a C1
    continuous cubic spline.
p : ndarray or None, shape (k,)
    Found parameters. None, if the parameters were not present in the
    problem.
x : ndarray, shape (m,)
    Nodes of the final mesh.
y : ndarray, shape (n, m)
    Solution values at the mesh nodes.
yp : ndarray, shape (n, m)
    Solution derivatives at the mesh nodes.
rms_residuals : ndarray, shape (m - 1,)
    RMS values of the relative residuals over each mesh interval (see the
    description of `tol` parameter).
niter : int
    Number of completed iterations.
status : int
    Reason for algorithm termination:

        * 0: The algorithm converged to the desired accuracy.
        * 1: The maximum number of mesh nodes is exceeded.
        * 2: A singular Jacobian encountered when solving the collocation
          system.

message : string
    Verbal description of the termination reason.
success : bool
    True if the algorithm converged to the desired accuracy (``status=0``).

Notes
-----
This function implements a 4th order collocation algorithm with the
control of residuals similar to [1]_. A collocation system is solved
by a damped Newton method with an affine-invariant criterion function as
```

```
described in [3]_.

Note that in [1]_  integral residuals are defined without normalization
by interval lengths. So, their definition is different by a multiplier of
h**0.5 (h is an interval length) from the definition used here.

.. versionadded:: 0.18.0

References
----------
.. [1] J. Kierzenka, L. F. Shampine, "A BVP Solver Based on Residual
       Control and the Maltab PSE", ACM Trans. Math. Softw., Vol. 27,
       Number 3, pp. 299-316, 2001.
.. [2] L.F. Shampine, P. H. Muir and H. Xu, "A User-Friendly Fortran BVP
       Solver".
.. [3] U. Ascher, R. Mattheij and R. Russell "Numerical Solution of
       Boundary Value Problems for Ordinary Differential Equations".
.. [4] `Cauchy-Riemann equations
        <https://en.wikipedia.org/wiki/Cauchy-Riemann_equations>`_ on
        Wikipedia.

Examples
--------
In the first example, we solve Bratu's problem::

    y'' + k * exp(y) = 0
    y(0) = y(1) = 0

for k = 1.

We rewrite the equation as a first-order system and implement its
right-hand side evaluation::

    y1' = y2
    y2' = -exp(y1)

>>> def fun(x, y):
...     return np.vstack((y[1], -np.exp(y[0])))

Implement evaluation of the boundary condition residuals:

>>> def bc(ya, yb):
...     return np.array([ya[0], yb[0]])

Define the initial mesh with 5 nodes:

>>> x = np.linspace(0, 1, 5)
```

This problem is known to have two solutions. To obtain both of them, we
use two different initial guesses for y. We denote them by subscripts
a and b.

```
>>> y_a = np.zeros((2, x.size))
>>> y_b = np.zeros((2, x.size))
>>> y_b[0] = 3
```

Now we are ready to run the solver.

```
>>> from scipy.integrate import solve_bvp
>>> res_a = solve_bvp(fun, bc, x, y_a)
>>> res_b = solve_bvp(fun, bc, x, y_b)
```

Let's plot the two found solutions. We take an advantage of having the
solution in a spline form to produce a smooth plot.

```
>>> x_plot = np.linspace(0, 1, 100)
>>> y_plot_a = res_a.sol(x_plot)[0]
>>> y_plot_b = res_b.sol(x_plot)[0]
>>> import matplotlib.pyplot as plt
>>> plt.plot(x_plot, y_plot_a, label='y_a')
>>> plt.plot(x_plot, y_plot_b, label='y_b')
>>> plt.legend()
>>> plt.xlabel("x")
>>> plt.ylabel("y")
>>> plt.show()
```

We see that the two solutions have similar shape, but differ in scale
significantly.

In the second example, we solve a simple Sturm-Liouville problem::

```
    y'' + k**2 * y = 0
    y(0) = y(1) = 0
```

It is known that a non-trivial solution y = A * sin(k * x) is possible for
k = pi * n, where n is an integer. To establish the normalization constant
A = 1 we add a boundary condition::

```
    y'(0) = k
```

Again, we rewrite our equation as a first-order system and implement its
right-hand side evaluation::

```
    y1' = y2
```

```
            y2' = -k**2 * y1

        >>> def fun(x, y, p):
        ...     k = p[0]
        ...     return np.vstack((y[1], -k**2 * y[0]))

        Note that parameters p are passed as a vector (with one element in our
        case).

        Implement the boundary conditions:

        >>> def bc(ya, yb, p):
        ...     k = p[0]
        ...     return np.array([ya[0], yb[0], ya[1] - k])

        Set up the initial mesh and guess for y. We aim to find the solution for
        k = 2 * pi, to achieve that we set values of y to approximately follow
        sin(2 * pi * x):

        >>> x = np.linspace(0, 1, 5)
        >>> y = np.zeros((2, x.size))
        >>> y[0, 1] = 1
        >>> y[0, 3] = -1

        Run the solver with 6 as an initial guess for k.

        >>> sol = solve_bvp(fun, bc, x, y, p=[6])

        We see that the found k is approximately correct:

        >>> sol.p[0]
        6.28329460046

        And, finally, plot the solution to see the anticipated sinusoid:

        >>> x_plot = np.linspace(0, 1, 100)
        >>> y_plot = sol.sol(x_plot)[0]
        >>> plt.plot(x_plot, y_plot)
        >>> plt.xlabel("x")
        >>> plt.ylabel("y")
        >>> plt.show()

    solve_ivp(fun, t_span, y0, method='RK45', t_eval=None, dense_output=False, events=None, vectorized=False, args=None, **option
s)
        Solve an initial value problem for a system of ODEs.

        This function numerically integrates a system of ordinary differential
```

```
equations given an initial value::

    dy / dt = f(t, y)
    y(t0) = y0

Here t is a 1-D independent variable (time), y(t) is an
N-D vector-valued function (state), and an N-D
vector-valued function f(t, y) determines the differential equations.
The goal is to find y(t) approximately satisfying the differential
equations, given an initial value y(t0)=y0.

Some of the solvers support integration in the complex domain, but note
that for stiff ODE solvers, the right-hand side must be
complex-differentiable (satisfy Cauchy-Riemann equations [11]_).
To solve a problem in the complex domain, pass y0 with a complex data type.
Another option always available is to rewrite your problem for real and
imaginary parts separately.

Parameters
----------
fun : callable
    Right-hand side of the system. The calling signature is ``fun(t, y)``.
    Here `t` is a scalar, and there are two options for the ndarray `y`:
    It can either have shape (n,); then `fun` must return array_like with
    shape (n,). Alternatively, it can have shape (n, k); then `fun`
    must return an array_like with shape (n, k), i.e., each column
    corresponds to a single column in `y`. The choice between the two
    options is determined by `vectorized` argument (see below). The
    vectorized implementation allows a faster approximation of the Jacobian
    by finite differences (required for stiff solvers).
t_span : 2-tuple of floats
    Interval of integration (t0, tf). The solver starts with t=t0 and
    integrates until it reaches t=tf.
y0 : array_like, shape (n,)
    Initial state. For problems in the complex domain, pass `y0` with a
    complex data type (even if the initial value is purely real).
method : string or `OdeSolver`, optional
    Integration method to use:

        * 'RK45' (default): Explicit Runge-Kutta method of order 5(4) [1]_.
          The error is controlled assuming accuracy of the fourth-order
          method, but steps are taken using the fifth-order accurate
          formula (local extrapolation is done). A quartic interpolation
          polynomial is used for the dense output [2]_. Can be applied in
          the complex domain.
        * 'RK23': Explicit Runge-Kutta method of order 3(2) [3]_. The error
          is controlled assuming accuracy of the second-order method, but
          steps are taken using the third-order accurate formula (local
          extrapolation is done). A cubic Hermite polynomial is used for the
          dense output. Can be applied in the complex domain.
        * 'DOP853': Explicit Runge-Kutta method of order 8 [13]_.
          Python implementation of the "DOP853" algorithm originally
          written in Fortran [14]_. A 7-th order interpolation polynomial
          accurate to 7-th order is used for the dense output.
          Can be applied in the complex domain.
        * 'Radau': Implicit Runge-Kutta method of the Radau IIA family of
          order 5 [4]_. The error is controlled with a third-order accurate
          embedded formula. A cubic polynomial which satisfies the
          collocation conditions is used for the dense output.
        * 'BDF': Implicit multi-step variable-order (1 to 5) method based
          on a backward differentiation formula for the derivative
          approximation [5]_. The implementation follows the one described
          in [6]_. A quasi-constant step scheme is used and accuracy is
          enhanced using the NDF modification. Can be applied in the
          complex domain.
        * 'LSODA': Adams/BDF method with automatic stiffness detection and
          switching [7]_, [8]_. This is a wrapper of the Fortran solver
          from ODEPACK.

    Explicit Runge-Kutta methods ('RK23', 'RK45', 'DOP853') should be used
    for non-stiff problems and implicit methods ('Radau', 'BDF') for
    stiff problems [9]_. Among Runge-Kutta methods, 'DOP853' is recommended
    for solving with high precision (low values of `rtol` and `atol`).

    If not sure, first try to run 'RK45'. If it makes unusually many
    iterations, diverges, or fails, your problem is likely to be stiff and
    you should use 'Radau' or 'BDF'. 'LSODA' can also be a good universal
    choice, but it might be somewhat less convenient to work with as it
    wraps old Fortran code.

    You can also pass an arbitrary class derived from `OdeSolver` which
    implements the solver.
t_eval : array_like or None, optional
    Times at which to store the computed solution, must be sorted and lie
    within `t_span`. If None (default), use points selected by the solver.
dense_output : bool, optional
    Whether to compute a continuous solution. Default is False.
events : callable, or list of callables, optional
    Events to track. If None (default), no events will be tracked.
    Each event occurs at the zeros of a continuous function of time and
    state. Each function must have the signature ``event(t, y)`` and return
    a float. The solver will find an accurate value of `t` at which
    ``event(t, y(t)) = 0`` using a root-finding algorithm. By default, all
    zeros will be found. The solver looks for a sign change over each step,
    so if multiple zero crossings occur within one step, events may be
    missed. Additionally each `event` function might have the following
    attributes:

        terminal: bool, optional
            Whether to terminate integration if this event occurs.
            Implicitly False if not assigned.
        direction: float, optional
            Direction of a zero crossing. If `direction` is positive,
            `event` will only trigger when going from negative to positive,
            and vice versa if `direction` is negative. If 0, then either
            direction will trigger event. Implicitly 0 if not assigned.

    You can assign attributes like ``event.terminal = True`` to any
    function in Python.
vectorized : bool, optional
    Whether `fun` is implemented in a vectorized fashion. Default is False.
args : tuple, optional
    Additional arguments to pass to the user-defined functions.  If given,
    the additional arguments are passed to all user-defined functions.
    So if, for example, `fun` has the signature ``fun(t, y, a, b, c)``,
    then `jac` (if given) and any event functions must have the same
    signature, and `args` must be a tuple of length 3.
options
    Options passed to a chosen solver. All options available for already
    implemented solvers are listed below.
first_step : float or None, optional
    Initial step size. Default is `None` which means that the algorithm
    should choose.
max_step : float, optional
    Maximum allowed step size. Default is np.inf, i.e., the step size is not
    bounded and determined solely by the solver.
rtol, atol : float or array_like, optional
    Relative and absolute tolerances. The solver keeps the local error
    estimates less than ``atol + rtol * abs(y)``. Here `rtol` controls a
    relative accuracy (number of correct digits). But if a component of `y`
    is approximately below `atol`, the error only needs to fall within
    the same `atol` threshold, and the number of correct digits is not
    guaranteed. If components of y have different scales, it might be
    beneficial to set different `atol` values for different components by
    passing array_like with shape (n,) for `atol`. Default values are
    1e-3 for `rtol` and 1e-6 for `atol`.
jac : array_like, sparse_matrix, callable or None, optional
    Jacobian matrix of the right-hand side of the system with respect
    to y, required by the 'Radau', 'BDF' and 'LSODA' method. The
    Jacobian matrix has shape (n, n) and its element (i, j) is equal to
    ``d f_i / d y_j``.  There are three ways to define the Jacobian:
        * If array_like or sparse_matrix, the Jacobian is assumed to
          be constant. Not supported by 'LSODA'.
        * If callable, the Jacobian is assumed to depend on both
          t and y; it will be called as ``jac(t, y)``, as necessary.
          For 'Radau' and 'BDF' methods, the return value might be a
          sparse matrix.
        * If None (default), the Jacobian will be approximated by
          finite differences.

    It is generally recommended to provide the Jacobian rather than
    relying on a finite-difference approximation.
jac_sparsity : array_like, sparse matrix or None, optional
    Defines a sparsity structure of the Jacobian matrix for a finite-
    difference approximation. Its shape must be (n, n). This argument
    is ignored if `jac` is not `None`. If the Jacobian has only few
    non-zero elements in *each* row, providing the sparsity structure
    will greatly speed up the computations [10]_. A zero entry means that
    a corresponding element in the Jacobian is always zero. If None
    (default), the Jacobian is assumed to be dense.
    Not supported by 'LSODA', see `lband` and `uband` instead.
lband, uband : int or None, optional
    Parameters defining the bandwidth of the Jacobian for the 'LSODA'
    method, i.e., ``jac[i, j] != 0 only for i - lband <= j <= i + uband``.
    Default is None. Setting these requires your jac routine to return the
    Jacobian in the packed format: the returned array must have ``n``
    columns and ``uband + lband + 1`` rows in which Jacobian diagonals are
    written. Specifically ``jac_packed[uband + i - j , j] = jac[i, j]``.
    The same format is used in `scipy.linalg.solve_banded` (check for an
    illustration).  These parameters can be also used with ``jac=None`` to
    reduce the number of Jacobian elements estimated by finite differences.
min_step : float, optional
    The minimum allowed step size for 'LSODA' method.
    By default `min_step` is zero.

Returns
-------
Bunch object with the following fields defined:
t : ndarray, shape (n_points,)
    Time points.
y : ndarray, shape (n, n_points)
    Values of the solution at `t`.
sol : `OdeSolution` or None
    Found solution as `OdeSolution` instance; None if `dense_output` was
    set to False.
t_events : list of ndarray or None
    Contains for each event type a list of arrays at which an event of
    that type event was detected. None if `events` was None.
y_events : list of ndarray or None
    For each value of `t_events`, the corresponding value of the solution.
    None if `events` was None.
nfev : int
    Number of evaluations of the right-hand side.
njev : int
    Number of evaluations of the Jacobian.
nlu : int
    Number of LU decompositions.
status : int
    Reason for algorithm termination:

        * -1: Integration step failed.
        *  0: The solver successfully reached the end of `tspan`.
        *  1: A termination event occurred.

message : string
    Human-readable description of the termination reason.
success : bool
    True if the solver reached the interval end or a termination event
    occurred (``status >= 0``).

References
----------
.. [1] J. R. Dormand, P. J. Prince, "A family of embedded Runge-Kutta
       formulae", Journal of Computational and Applied Mathematics, Vol. 6,
       No. 1, pp. 19-26, 1980.
.. [2] L. W. Shampine, "Some Practical Runge-Kutta Formulas", Mathematics
       of Computation,, Vol. 46, No. 173, pp. 135-150, 1986.
.. [3] P. Bogacki, L.F. Shampine, "A 3(2) Pair of Runge-Kutta Formulas",
       Appl. Math. Lett. Vol. 2, No. 4. pp. 321-325, 1989.
.. [4] E. Hairer, G. Wanner, "Solving Ordinary Differential Equations II:
       Stiff and Differential-Algebraic Problems", Sec. IV.8.
.. [5] `Backward Differentiation Formula
        <https://en.wikipedia.org/wiki/Backward_differentiation_formula>`_
        on Wikipedia.
.. [6] L. F. Shampine, M. W. Reichelt, "THE MATLAB ODE SUITE", SIAM J. SCI.
       COMPUTE., Vol. 18, No. 1, pp. 1-22, January 1997.
.. [7] A. C. Hindmarsh, "ODEPACK, A Systematized Collection of ODE
       Solvers," IMACS Transactions on Scientific Computation, Vol 1.,
       pp. 55-64, 1983.
.. [8] L. Petzold, "Automatic selection of methods for solving stiff and
       nonstiff systems of ordinary differential equations", SIAM Journal
       on Scientific and Statistical Computing, Vol. 4, No. 1, pp. 136-148,
       1983.
.. [9] `Stiff equation <https://en.wikipedia.org/wiki/Stiff_equation>`_ on
```

```
       Wikipedia.
.. [10] A. Curtis, M. J. D. Powell, and J. Reid, "On the estimation of
        sparse Jacobian matrices", Journal of the Institute of Mathematics
        and its Applications, 13, pp. 117-120, 1974.
.. [11] `Cauchy-Riemann equations
         <https://en.wikipedia.org/wiki/Cauchy-Riemann_equations>`_ on
         Wikipedia.
.. [12] `Lotka-Volterra equations
        <https://en.wikipedia.org/wiki/Lotka%E2%80%93Volterra_equations>`_
        on Wikipedia.
.. [13] E. Hairer, S. P. Norsett G. Wanner, "Solving Ordinary Differential
        Equations I: Nonstiff Problems", Sec. II.
.. [14] `Page with original Fortran code of DOP853
        <http://www.unige.ch/~hairer/software.html>`_.

Examples
--------
Basic exponential decay showing automatically chosen time points.

>>> from scipy.integrate import solve_ivp
>>> def exponential_decay(t, y): return -0.5 * y
>>> sol = solve_ivp(exponential_decay, [0, 10], [2, 4, 8])
>>> print(sol.t)
[ 0.          0.11487653  1.26364188  3.06061781  4.81611105  6.57445806
  8.33328988 10.        ]
>>> print(sol.y)
[[2.         1.88836035 1.06327177 0.43319312 0.18017253 0.07483045
  0.03107158 0.01350781]
 [4.         3.7767207  2.12654355 0.86638624 0.36034507 0.14966091
  0.06214316 0.02701561]
 [8.         7.5534414  4.25308709 1.73277247 0.72069014 0.29932181
  0.12428631 0.05403123]]

Specifying points where the solution is desired.

>>> sol = solve_ivp(exponential_decay, [0, 10], [2, 4, 8],
...                 t_eval=[0, 1, 2, 4, 10])
>>> print(sol.t)
[ 0  1  2  4 10]
>>> print(sol.y)
[[2.         1.21305369 0.73534021 0.27066736 0.01350938]
 [4.         2.42610739 1.47068043 0.54133472 0.02701876]
 [8.         4.85221478 2.94136085 1.08266944 0.05403753]]

Cannon fired upward with terminal event upon impact. The ``terminal`` and
``direction`` fields of an event are applied by monkey patching a function.
Here ``y[0]`` is position and ``y[1]`` is velocity. The projectile starts
```

at position 0 with velocity +10. Note that the integration never reaches
t=100 because the event is terminal.

```
>>> def upward_cannon(t, y): return [y[1], -0.5]
>>> def hit_ground(t, y): return y[0]
>>> hit_ground.terminal = True
>>> hit_ground.direction = -1
>>> sol = solve_ivp(upward_cannon, [0, 100], [0, 10], events=hit_ground)
>>> print(sol.t_events)
[array([40.])]
>>> print(sol.t)
[0.00000000e+00 9.99900010e-05 1.09989001e-03 1.10988901e-02
 1.11088891e-01 1.11098890e+00 1.11099890e+01 4.00000000e+01]
```

Use `dense_output` and `events` to find position, which is 100, at the apex
of the cannonball's trajectory. Apex is not defined as terminal, so both
apex and hit_ground are found. There is no information at t=20, so the sol
attribute is used to evaluate the solution. The sol attribute is returned
by setting ``dense_output=True``. Alternatively, the `y_events` attribute
can be used to access the solution at the time of the event.

```
>>> def apex(t, y): return y[1]
>>> sol = solve_ivp(upward_cannon, [0, 100], [0, 10],
...                 events=(hit_ground, apex), dense_output=True)
>>> print(sol.t_events)
[array([40.]), array([20.])]
>>> print(sol.t)
[0.00000000e+00 9.99900010e-05 1.09989001e-03 1.10988901e-02
 1.11088891e-01 1.11098890e+00 1.11099890e+01 4.00000000e+01]
>>> print(sol.sol(sol.t_events[1][0]))
[100.   0.]
>>> print(sol.y_events)
[array([[-5.68434189e-14, -1.00000000e+01]]), array([[1.00000000e+02, 1.77635684e-15]])]
```

As an example of a system with additional parameters, we'll implement
the Lotka-Volterra equations [12]_.

```
>>> def lotkavolterra(t, z, a, b, c, d):
...     x, y = z
...     return [a*x - b*x*y, -c*y + d*x*y]
...
```

We pass in the parameter values a=1.5, b=1, c=3 and d=1 with the `args`
argument.

```
>>> sol = solve_ivp(lotkavolterra, [0, 15], [10, 5], args=(1.5, 1, 3, 1),
...                 dense_output=True)
```

```
    Compute a dense solution and plot it.

    >>> t = np.linspace(0, 15, 300)
    >>> z = sol.sol(t)
    >>> import matplotlib.pyplot as plt
    >>> plt.plot(t, z.T)
    >>> plt.xlabel('t')
    >>> plt.legend(['x', 'y'], shadow=True)
    >>> plt.title('Lotka-Volterra System')
    >>> plt.show()

tplquad(func, a, b, gfun, hfun, qfun, rfun, args=(), epsabs=1.49e-08, epsrel=1.49e-08)
    Compute a triple (definite) integral.

    Return the triple integral of ``func(z, y, x)`` from ``x = a..b``,
    ``y = gfun(x)..hfun(x)``, and ``z = qfun(x,y)..rfun(x,y)``.

    Parameters
    ----------
    func : function
        A Python function or method of at least three variables in the
        order (z, y, x).
    a, b : float
        The limits of integration in x: `a` < `b`
    gfun : function or float
        The lower boundary curve in y which is a function taking a single
        floating point argument (x) and returning a floating point result
        or a float indicating a constant boundary curve.
    hfun : function or float
        The upper boundary curve in y (same requirements as `gfun`).
    qfun : function or float
        The lower boundary surface in z.  It must be a function that takes
        two floats in the order (x, y) and returns a float or a float
        indicating a constant boundary surface.
    rfun : function or float
        The upper boundary surface in z. (Same requirements as `qfun`.)
    args : tuple, optional
        Extra arguments to pass to `func`.
    epsabs : float, optional
        Absolute tolerance passed directly to the innermost 1-D quadrature
        integration. Default is 1.49e-8.
    epsrel : float, optional
        Relative tolerance of the innermost 1-D integrals. Default is 1.49e-8.

    Returns
    -------
```

```
        y : float
            The resultant integral.
        abserr : float
            An estimate of the error.

        See Also
        --------
        quad: Adaptive quadrature using QUADPACK
        quadrature: Adaptive Gaussian quadrature
        fixed_quad: Fixed-order Gaussian quadrature
        dblquad: Double integrals
        nquad : N-dimensional integrals
        romb: Integrators for sampled data
        simpson: Integrators for sampled data
        ode: ODE integrators
        odeint: ODE integrators
        scipy.special: For coefficients and roots of orthogonal polynomials

        Examples
        --------

        Compute the triple integral of ``x * y * z``, over ``x`` ranging
        from 1 to 2, ``y`` ranging from 2 to 3, ``z`` ranging from 0 to 1.

        >>> from scipy import integrate
        >>> f = lambda z, y, x: x*y*z
        >>> integrate.tplquad(f, 1, 2, lambda x: 2, lambda x: 3,
        ...                   lambda x, y: 0, lambda x, y: 1)
        (1.8750000000000002, 3.324644794257407e-14)

    trapezoid = trapz(y, x=None, dx=1.0, axis=-1)
        Integrate along the given axis using the composite trapezoidal rule.

        Integrate `y` (`x`) along given axis.

        Parameters
        ----------
        y : array_like
            Input array to integrate.
        x : array_like, optional
            The sample points corresponding to the `y` values. If `x` is None,
            the sample points are assumed to be evenly spaced `dx` apart. The
            default is None.
        dx : scalar, optional
            The spacing between sample points when `x` is None. The default is 1.
        axis : int, optional
            The axis along which to integrate.
        Returns
        -------
        trapz : float
            Definite integral as approximated by trapezoidal rule.

        See Also
        --------
        numpy.cumsum

        Notes
        -----
        Image [2]_ illustrates trapezoidal rule -- y-axis locations of points
        will be taken from `y` array, by default x-axis distances between
        points will be 1.0, alternatively they can be provided with `x` array
        or with `dx` scalar.  Return value will be equal to combined area under
        the red lines.


        References
        ----------
        .. [1] Wikipedia page: https://en.wikipedia.org/wiki/Trapezoidal_rule

        .. [2] Illustration image:
               https://en.wikipedia.org/wiki/File:Composite_trapezoidal_rule_illustration.png

        Examples
        --------
        >>> np.trapz([1,2,3])
        4.0
        >>> np.trapz([1,2,3], x=[4,6,8])
        8.0
        >>> np.trapz([1,2,3], dx=2)
        8.0
        >>> a = np.arange(6).reshape(2, 3)
        >>> a
        array([[0, 1, 2],
               [3, 4, 5]])
        >>> np.trapz(a, axis=0)
        array([1.5, 2.5, 3.5])
        >>> np.trapz(a, axis=1)
        array([2.,  8.])

    trapz(y, x=None, dx=1.0, axis=-1)
        `An alias of `trapezoid`.

        `trapz` is kept for backwards compatibility. For new code, prefer
```

```
        `trapezoid` instead.

DATA
    __all__ = ['AccuracyWarning', 'BDF', 'DOP853', 'DenseOutput', 'Integra...

FILE
    d:\programdata\anaconda3\lib\site-packages\scipy\integrate\__init__.py
```

Общая интеграция (quad)

Функция quad предназначена для интегрирования функции одной переменной между двумя точками. Точки могут быть для указания бесконечных пределов.

$$I = \int_0^{4.5} J_{2.5}\left(x\right)\, dx.$$

Это можно вычислить с помощью quad:

In [64]:
```
import scipy.integrate as integrate
```

In [65]:
```
import scipy.special as special
```

In [66]:
```
result = integrate.quad(lambda x: special.jv(2.5,x), 0, 4.5)
```

In [67]:
```
result
```

Out[67]:
```
(1.1178179380783244, 7.866317216380707e-09)
```

In [68]:
```
from numpy import sqrt, sin, cos, pi
```

In [69]:
```
I = sqrt(2/pi)*(18.0/27*sqrt(2)*cos(4.5) - 4.0/27*sqrt(2)*sin(4.5) + sqrt(2*pi) * special.fresnel(3/sqrt(pi))[0])
```

In [70]:
```
I
```
Out[70]:
```
1.117817938088701
```
In [71]:
```
print(abs(result[0]-I))
```
```
1.0376588477356563e-11
```
Первый аргумент quad — это «вызываемый» объект Python (то есть функция, метод или экземпляр класса). Обратите внимание на использование в данном случае лямбда-функции в качестве аргумента. Следующие два аргумента являются пределами интегрирования. Возвращаемое значение представляет собой кортеж, первый элемент которого содержит оценочное значение интеграла, а второй элемент содержит верхнюю границу ошибки. Заметим, что в этом случае истинное значение этого интеграла равно

$$I = \sqrt{\frac{2}{\pi}}\left(\frac{18}{27}\sqrt{2}\cos(4.5) - \frac{4}{27}\sqrt{2}\sin(4.5) + \sqrt{2\pi}\mathrm{Si}\left(\frac{3}{\sqrt{\pi}}\right)\right),$$

$$\mathrm{Si}\,(x) = \int_0^x \sin\left(\frac{\pi}{2}t^2\right)\,dt.$$

является синусоидальным интегралом Френеля. Обратите внимание, что численно вычисленный интеграл находится в пределах 1.04×10^{-11} точного результата — значительно ниже установленной границы ошибки.

Если интегрируемая функция принимает дополнительные параметры, их можно указать в аргументе args . Предположим, что необходимо вычислить следующий интеграл:

Если интегрируемая функция принимает дополнительные параметры, их можно указать в аргументе args . Предположим, что необходимо

вычислить следующий интеграл: $$I(a,b) = \int_0^1 ax^2 + b\,dx.$$ Этот интеграл можно вычислить с помощью следующего кода:

```
In [72]: from scipy.integrate import quad
```

```
In [73]: def integrand(x, a, b): return a*x**2 + b
```

```
In [74]: a = 2
```

```
In [75]: b = 1
```

```
In [76]: I = quad(integrand, 0, 1, args=(a,b))
```

```
In [77]: I
```

```
Out[77]: (1.6666666666666667, 1.8503717077085944e-14)
```