A shared key.

Examples

```
>>> from sympy.crypto.crypto import (
...     dh_private_key, dh_public_key, dh_shared_key)
>>> prk = dh_private_key();
>>> p, g, x = dh_public_key(prk);
>>> sk = dh_shared_key((p, g, x), 1000)
>>> sk == pow(x, 1000, p)
True
```

sympy.crypto.crypto.**gm_public_key**(*p*, *q*, *a=None*, *seed=None*)

Compute public keys for `p` and `q`. Note that in Goldwasser-Micali Encryption, public keys are randomly selected.

Parameters

p, q, a : int, int, int

Initialization variables.

Returns

tuple : (a, N)

`a` is the input `a` if it is not `None` otherwise some random integer co-prime to `p` and `q`.

`N` is the product of `p` and `q`.

sympy.crypto.crypto.**gm_private_key**(*p*, *q*, *a=None*)

Check if `p` and `q` can be used as private keys for the Goldwasser-Micali encryption. The method works roughly as follows.

Parameters

p, q, a

Initialization variables.

Returns

tuple : (p, q)

The input value `p` and `q`.

Raises

ValueError

If `p` and `q` are not distinct odd primes.

Explanation

1. Pick two large primes p and q.
2. Call their product N.
3. Given a message as an integer i, write i in its bit representation $b_0, \ldots, b_n$.
4. For each k,

   if $b_k = 0$**:**
   let a_k be a random square (quadratic residue) modulo pq such that `jacobi_symbol(a, p*q) = 1`

   if $b_k = 1$**:**
   let a_k be a random non-square (non-quadratic residue) modulo pq such that `jacobi_symbol(a, p*q) = 1`

returns $[a_1, a_2, \ldots]$

b_k can be recovered by checking whether or not a_k is a residue. And from the b_k's, the message can be reconstructed.

The idea is that, while `jacobi_symbol(a, p*q)` can be easily computed (and when it is equal to -1 will tell you that a is not a square mod pq), quadratic residuosity modulo a composite number is hard to compute without knowing its factorization.

Moreover, approximately half the numbers coprime to pq have *jacobi_symbol()* (page 1522) equal to 1 . And among those, approximately half are residues and approximately half are not. This maximizes the entropy of the code.

sympy.crypto.crypto.**encipher_gm**(*i*, *key*, *seed=None*)

Encrypt integer 'i' using public_key 'key' Note that gm uses random encryption.

Parameters

i : int

The message to encrypt.

key : (a, N)

The public key.

Returns

list : list of int

The randomized encrypted message.

sympy.crypto.crypto.**decipher_gm**(*message*, *key*)

Decrypt message 'message' using public_key 'key'.

Parameters

message : list of int

The randomized encrypted message.

key : (p, q)

The private key.

Returns

int

The encrypted message.

sympy.crypto.crypto.**encipher_railfence**(*message*, *rails*)

Performs Railfence Encryption on plaintext and returns ciphertext

Parameters

message : string, the message to encrypt.

rails : int, the number of rails.

Returns

The Encrypted string message.

Examples

```
>>> from sympy.crypto.crypto import encipher_railfence
>>> message = "hello world"
>>> encipher_railfence(message,3)
'horel ollwd'
```

References

[R160]

sympy.crypto.crypto.**decipher_railfence**(*ciphertext*, *rails*)

Decrypt the message using the given rails

Parameters

message : string, the message to encrypt.

rails : int, the number of rails.

Returns

The Decrypted string message.

Examples

```
>>> from sympy.crypto.crypto import decipher_railfence
>>> decipher_railfence("horel ollwd",3)
'hello world'
```

Differential Geometry

Introduction

Base Class Reference

class sympy.diffgeom.**Manifold**(*name*, *dim*, ***kwargs*)

A mathematical manifold.

Parameters

name : str

The name of the manifold.

dim : int

The dimension of the manifold.

Explanation

A manifold is a topological space that locally resembles Euclidean space near each point [1]. This class does not provide any means to study the topological characteristics of the manifold that it represents, though.

Examples

```
>>> from sympy.diffgeom import Manifold
>>> m = Manifold('M', 2)
>>> m
M
>>> m.dim
2
```

References

[R161]

class sympy.diffgeom.**Patch**(*name*, *manifold*, ***kwargs*)

A patch on a manifold.

Parameters

name : str

The name of the patch.

manifold : Manifold

The manifold on which the patch is defined.

Explanation

Coordinate patch, or patch in short, is a simply-connected open set around a point in the manifold [1]. On a manifold one can have many patches that do not always include the whole manifold. On these patches coordinate charts can be defined that permit the parameterization of any point on the patch in terms of a tuple of real numbers (the coordinates).

This class does not provide any means to study the topological characteristics of the patch that it represents.

Examples

```
>>> from sympy.diffgeom import Manifold, Patch
>>> m = Manifold('M', 2)
>>> p = Patch('P', m)
>>> p
P
>>> p.dim
2
```

References

[R162]

class sympy.diffgeom.**CoordSystem**(*name*, *patch*, *symbols=None*, *relations={}*, ***kwargs*)

A coordinate system defined on the patch.

Parameters

name : str

The name of the coordinate system.

patch : Patch

The patch where the coordinate system is defined.

symbols : list of Symbols, optional

Defines the names and assumptions of coordinate symbols.

relations : dict, optional

Key is a tuple of two strings, who are the names of the systems where the coordinates transform from and transform to. Value is a tuple of the symbols before transformation and a tuple of the expressions after transformation.

Explanation

Coordinate system is a system that uses one or more coordinates to uniquely determine the position of the points or other geometric elements on a manifold [1].

By passing `Symbols` to *symbols* parameter, user can define the name and assumptions of coordinate symbols of the coordinate system. If not passed, these symbols are generated automatically and are assumed to be real valued.

By passing *relations* parameter, user can define the tranform relations of coordinate systems. Inverse transformation and indirect transformation can be found automatically. If this parameter is not passed, coordinate transformation cannot be done.

Examples

We define two-dimensional Cartesian coordinate system and polar coordinate system.

```
>>> from sympy import symbols, pi, sqrt, atan2, cos, sin
>>> from sympy.diffgeom import Manifold, Patch, CoordSystem
>>> m = Manifold('M', 2)
>>> p = Patch('P', m)
>>> x, y = symbols('x y', real=True)
>>> r, theta = symbols('r theta', nonnegative=True)
>>> relation_dict = {
... ('Car2D', 'Pol'): [(x, y), (sqrt(x**2 + y**2), atan2(y, x))],
... ('Pol', 'Car2D'): [(r, theta), (r*cos(theta), r*sin(theta))]
... }
>>> Car2D = CoordSystem('Car2D', p, (x, y), relation_dict)
>>> Pol = CoordSystem('Pol', p, (r, theta), relation_dict)
```

`symbols` property returns `CoordinateSymbol` instances. These symbols are not same with the symbols used to construct the coordinate system.

```
>>> Car2D
Car2D
>>> Car2D.dim
2
>>> Car2D.symbols
(x, y)
>>> _[0].func
<class 'sympy.diffgeom.diffgeom.CoordinateSymbol'>
```

`transformation()` method returns the transformation function from one coordinate system to another. `transform()` method returns the transformed coordinates.

```
>>> Car2D.transformation(Pol)
Lambda((x, y), Matrix([
[sqrt(x**2 + y**2)],
[      atan2(y, x)]]))
>>> Car2D.transform(Pol)
Matrix([
[sqrt(x**2 + y**2)],
[      atan2(y, x)]])
>>> Car2D.transform(Pol, [1, 2])
Matrix([
[sqrt(5)],
[atan(2)]])
```

`jacobian()` method returns the Jacobian matrix of coordinate transformation between two systems. `jacobian_determinant()` method returns the Jacobian determinant of coordinate transformation between two systems.

```
>>> Pol.jacobian(Car2D)
Matrix([
[cos(theta), -r*sin(theta)],
[sin(theta),  r*cos(theta)]])
>>> Pol.jacobian(Car2D, [1, pi/2])
```

(continues on next page)

(continued from previous page)

```
Matrix([
[0, -1],
[1,  0]])
>>> Car2D.jacobian_determinant(Pol)
1/sqrt(x**2 + y**2)
>>> Car2D.jacobian_determinant(Pol, [1,0])
1
```

References

[R163]

base_oneform(*coord_index*)
Return a basis 1-form field. The basis one-form field for this coordinate system. It is also an operator on vector fields.

base_oneforms()
Returns a list of all base oneforms. For more details see the `base_oneform` method of this class.

base_scalar(*coord_index*)
Return `BaseScalarField` that takes a point and returns one of the coordinates.

base_scalars()
Returns a list of all coordinate functions. For more details see the `base_scalar` method of this class.

base_vector(*coord_index*)
Return a basis vector field. The basis vector field for this coordinate system. It is also an operator on scalar fields.

base_vectors()
Returns a list of all base vectors. For more details see the `base_vector` method of this class.

coord_function(*coord_index*)
Return `BaseScalarField` that takes a point and returns one of the coordinates.

coord_functions()
Returns a list of all coordinate functions. For more details see the `base_scalar` method of this class.

coord_tuple_transform_to(*to_sys*, *coords*)
Transform `coords` to coord system `to_sys`.

jacobian(*sys*, *coordinates=None*)
Return the jacobian matrix of a transformation on given coordinates. If coordinates are not given, coordinate symbols of *self* are used.

Parameters
sys : CoordSystem

coordinates : Any iterable, optional.

Returns
sympy.ImmutableDenseMatrix

Examples

```
>>> from sympy.diffgeom.rn import R2_r, R2_p
>>> R2_p.jacobian(R2_r)
Matrix([
[cos(theta), -rho*sin(theta)],
[sin(theta),  rho*cos(theta)]])
>>> R2_p.jacobian(R2_r, [1, 0])
Matrix([
[1, 0],
[0, 1]])
```

jacobian_determinant(*sys*, *coordinates=None*)

Return the jacobian determinant of a transformation on given coordinates. If coordinates are not given, coordinate symbols of *self* are used.

Parameters

sys : CoordSystem

coordinates : Any iterable, optional.

Returns

sympy.Expr

Examples

```
>>> from sympy.diffgeom.rn import R2_r, R2_p
>>> R2_r.jacobian_determinant(R2_p)
1/sqrt(x**2 + y**2)
>>> R2_r.jacobian_determinant(R2_p, [1, 0])
1
```

jacobian_matrix(*sys*, *coordinates=None*)

Return the jacobian matrix of a transformation on given coordinates. If coordinates are not given, coordinate symbols of *self* are used.

Parameters

sys : CoordSystem

coordinates : Any iterable, optional.

Returns

sympy.ImmutableDenseMatrix

Examples

```
>>> from sympy.diffgeom.rn import R2_r, R2_p
>>> R2_p.jacobian(R2_r)
Matrix([
[cos(theta), -rho*sin(theta)],
[sin(theta),  rho*cos(theta)]])
>>> R2_p.jacobian(R2_r, [1, 0])
```

(continues on next page)

(continued from previous page)

```
Matrix([
[1, 0],
[0, 1]])
```

point(*coords*)

Create a `Point` with coordinates given in this coord system.

point_to_coords(*point*)

Calculate the coordinates of a point in this coord system.

transform(*sys*, *coordinates=None*)

Return the result of coordinate transformation from *self* to *sys*. If coordinates are not given, coordinate symbols of *self* are used.

Parameters

sys : CoordSystem

coordinates : Any iterable, optional.

Returns

sympy.ImmutableDenseMatrix containing CoordinateSymbol

Examples

```
>>> from sympy.diffgeom.rn import R2_r, R2_p
>>> R2_r.transform(R2_p)
Matrix([
[sqrt(x**2 + y**2)],
[      atan2(y, x)]])
>>> R2_r.transform(R2_p, [0, 1])
Matrix([
[   1],
[pi/2]])
```

transformation(*sys*)

Return coordinate transformation function from *self* to *sys*.

Parameters

sys : CoordSystem

Returns

sympy.Lambda

Examples

```
>>> from sympy.diffgeom.rn import R2_r, R2_p
>>> R2_r.transformation(R2_p)
Lambda((x, y), Matrix([
[sqrt(x**2 + y**2)],
[      atan2(y, x)]]))
```

class sympy.diffgeom.**CoordinateSymbol**(*coord_sys*, *index*, ***assumptions*)

A symbol which denotes an abstract value of i-th coordinate of the coordinate system with given context.

Parameters

coord_sys : CoordSystem

index : integer

Explanation

Each coordinates in coordinate system are represented by unique symbol, such as x, y, z in Cartesian coordinate system.

You may not construct this class directly. Instead, use *symbols* method of CoordSystem.

Examples

```
>>> from sympy import symbols, Lambda, Matrix, sqrt, atan2, cos, sin
>>> from sympy.diffgeom import Manifold, Patch, CoordSystem
>>> m = Manifold('M', 2)
>>> p = Patch('P', m)
>>> x, y = symbols('x y', real=True)
>>> r, theta = symbols('r theta', nonnegative=True)
>>> relation_dict = {
... ('Car2D', 'Pol'): Lambda((x, y), Matrix([sqrt(x**2 + y**2), atan2(y,␣
↪x)])),
... ('Pol', 'Car2D'): Lambda((r, theta), Matrix([r*cos(theta),␣
↪r*sin(theta)]))
... }
>>> Car2D = CoordSystem('Car2D', p, [x, y], relation_dict)
>>> Pol = CoordSystem('Pol', p, [r, theta], relation_dict)
>>> x, y = Car2D.symbols
```

`CoordinateSymbol` contains its coordinate symbol and index.

```
>>> x.name
'x'
>>> x.coord_sys == Car2D
True
>>> x.index
0
>>> x.is_real
True
```

You can transform `CoordinateSymbol` into other coordinate system using `rewrite()` method.

```
>>> x.rewrite(Pol)
r*cos(theta)
>>> sqrt(x**2 + y**2).rewrite(Pol).simplify()
r
```

class sympy.diffgeom.**Point**(*coord_sys*, *coords*, ***kwargs*)

Point defined in a coordinate system.

Parameters

coord_sys : CoordSystem

coords : list

The coordinates of the point.

Explanation

Mathematically, point is defined in the manifold and does not have any coordinates by itself. Coordinate system is what imbues the coordinates to the point by coordinate chart. However, due to the difficulty of realizing such logic, you must supply a coordinate system and coordinates to define a Point here.

The usage of this object after its definition is independent of the coordinate system that was used in order to define it, however due to limitations in the simplification routines you can arrive at complicated expressions if you use inappropriate coordinate systems.

Examples

```
>>> from sympy import pi
>>> from sympy.diffgeom import Point
>>> from sympy.diffgeom.rn import R2, R2_r, R2_p
>>> rho, theta = R2_p.symbols
```

```
>>> p = Point(R2_p, [rho, 3*pi/4])
```

```
>>> p.manifold == R2
True
```

```
>>> p.coords()
Matrix([
[   rho],
[3*pi/4]])
>>> p.coords(R2_r)
Matrix([
[-sqrt(2)*rho/2],
[ sqrt(2)*rho/2]])
```

coords(*sys=None*)

Coordinates of the point in given coordinate system. If coordinate system is not passed, it returns the coordinates in the coordinate system in which the poin was defined.

class sympy.diffgeom.**BaseScalarField**(*coord_sys*, *index*, ***kwargs*)

Base scalar field over a manifold for a given coordinate system.

Parameters

coord_sys : CoordSystem

index : integer

Explanation

A scalar field takes a point as an argument and returns a scalar. A base scalar field of a coordinate system takes a point and returns one of the coordinates of that point in the coordinate system in question.

To define a scalar field you need to choose the coordinate system and the index of the coordinate.

The use of the scalar field after its definition is independent of the coordinate system in which it was defined, however due to limitations in the simplification routines you may arrive at more complicated expression if you use unappropriate coordinate systems. You can build complicated scalar fields by just building up SymPy expressions containing `BaseScalarField` instances.

Examples

```
>>> from sympy import Function, pi
>>> from sympy.diffgeom import BaseScalarField
>>> from sympy.diffgeom.rn import R2_r, R2_p
>>> rho, _ = R2_p.symbols
>>> point = R2_p.point([rho, 0])
>>> fx, fy = R2_r.base_scalars()
>>> ftheta = BaseScalarField(R2_r, 1)
```

```
>>> fx(point)
rho
>>> fy(point)
0
```

```
>>> (fx**2+fy**2).rcall(point)
rho**2
```

```
>>> g = Function('g')
>>> fg = g(ftheta-pi)
>>> fg.rcall(point)
g(-pi)
```

class sympy.diffgeom.**BaseVectorField**(*coord_sys*, *index*, ***kwargs*)

Base vector field over a manifold for a given coordinate system.

Parameters

coord_sys : CoordSystem

index : integer

Explanation

A vector field is an operator taking a scalar field and returning a directional derivative (which is also a scalar field). A base vector field is the same type of operator, however the derivation is specifically done with respect to a chosen coordinate.

To define a base vector field you need to choose the coordinate system and the index of the coordinate.

The use of the vector field after its definition is independent of the coordinate system in which it was defined, however due to limitations in the simplification routines you may arrive at more complicated expression if you use unappropriate coordinate systems.

Examples

```
>>> from sympy import Function
>>> from sympy.diffgeom.rn import R2_p, R2_r
>>> from sympy.diffgeom import BaseVectorField
>>> from sympy import pprint
```

```
>>> x, y = R2_r.symbols
>>> rho, theta = R2_p.symbols
>>> fx, fy = R2_r.base_scalars()
>>> point_p = R2_p.point([rho, theta])
>>> point_r = R2_r.point([x, y])
```

```
>>> g = Function('g')
>>> s_field = g(fx, fy)
```

```
>>> v = BaseVectorField(R2_r, 1)
>>> pprint(v(s_field))
/ d           \|
|---(g(x, xi))||
\dxi          /|xi=y
>>> pprint(v(s_field).rcall(point_r).doit())
d
--(g(x, y))
dy
>>> pprint(v(s_field).rcall(point_p))
/ d                         \|
|---(g(rho*cos(theta), xi))||
\dxi                        /|xi=rho*sin(theta)
```

class sympy.diffgeom.**Commutator**(*v1*, *v2*)

Commutator of two vector fields.

Explanation

The commutator of two vector fields v_1 and v_2 is defined as the vector field $[v_1, v_2]$ that evaluated on each scalar field f is equal to $v_1(v_2(f)) - v_2(v_1(f))$.

Examples

```
>>> from sympy.diffgeom.rn import R2_p, R2_r
>>> from sympy.diffgeom import Commutator
>>> from sympy import simplify
```

```
>>> fx, fy = R2_r.base_scalars()
>>> e_x, e_y = R2_r.base_vectors()
>>> e_r = R2_p.base_vector(0)
```

```
>>> c_xy = Commutator(e_x, e_y)
>>> c_xr = Commutator(e_x, e_r)
>>> c_xy
0
```

Unfortunately, the current code is not able to compute everything:

```
>>> c_xr
Commutator(e_x, e_rho)
>>> simplify(c_xr(fy**2))
-2*cos(theta)*y**2/(x**2 + y**2)
```

class sympy.diffgeom.**Differential**(*form_field*)

Return the differential (exterior derivative) of a form field.

Explanation

The differential of a form (i.e. the exterior derivative) has a complicated definition in the general case. The differential df of the 0-form f is defined for any vector field v as $df(v) = v(f)$.

Examples

```
>>> from sympy import Function
>>> from sympy.diffgeom.rn import R2_r
>>> from sympy.diffgeom import Differential
>>> from sympy import pprint
```

```
>>> fx, fy = R2_r.base_scalars()
>>> e_x, e_y = R2_r.base_vectors()
>>> g = Function('g')
>>> s_field = g(fx, fy)
>>> dg = Differential(s_field)
```

```
>>> dg
d(g(x, y))
>>> pprint(dg(e_x))
/ d           \|
|---(g(xi, y))||
\dxi          /|xi=x
>>> pprint(dg(e_y))
/ d           \|
|---(g(x, xi))||
\dxi          /|xi=y
```

Applying the exterior derivative operator twice always results in:

```
>>> Differential(dg)
0
```

class sympy.diffgeom.**TensorProduct**(**args*)

Tensor product of forms.

Explanation

The tensor product permits the creation of multilinear functionals (i.e. higher order tensors) out of lower order fields (e.g. 1-forms and vector fields). However, the higher tensors thus created lack the interesting features provided by the other type of product, the wedge product, namely they are not antisymmetric and hence are not form fields.

Examples

```
>>> from sympy.diffgeom.rn import R2_r
>>> from sympy.diffgeom import TensorProduct
```

```
>>> fx, fy = R2_r.base_scalars()
>>> e_x, e_y = R2_r.base_vectors()
>>> dx, dy = R2_r.base_oneforms()
```

```
>>> TensorProduct(dx, dy)(e_x, e_y)
1
>>> TensorProduct(dx, dy)(e_y, e_x)
0
>>> TensorProduct(dx, fx*dy)(fx*e_x, e_y)
x**2
>>> TensorProduct(e_x, e_y)(fx**2, fy**2)
4*x*y
>>> TensorProduct(e_y, dx)(fy)
dx
```

You can nest tensor products.

```
>>> tp1 = TensorProduct(dx, dy)
>>> TensorProduct(tp1, dx)(e_x, e_y, e_x)
1
```

You can make partial contraction for instance when 'raising an index'. Putting None in the second argument of rcall means that the respective position in the tensor product is left as it is.

```
>>> TP = TensorProduct
>>> metric = TP(dx, dx) + 3*TP(dy, dy)
>>> metric.rcall(e_y, None)
3*dy
```

Or automatically pad the args with None without specifying them.

```
>>> metric.rcall(e_y)
3*dy
```

class sympy.diffgeom.**WedgeProduct**(**args*)

Wedge product of forms.

Explanation

In the context of integration only completely antisymmetric forms make sense. The wedge product permits the creation of such forms.

Examples

```
>>> from sympy.diffgeom.rn import R2_r
>>> from sympy.diffgeom import WedgeProduct
```

```
>>> fx, fy = R2_r.base_scalars()
>>> e_x, e_y = R2_r.base_vectors()
>>> dx, dy = R2_r.base_oneforms()
```

```
>>> WedgeProduct(dx, dy)(e_x, e_y)
1
>>> WedgeProduct(dx, dy)(e_y, e_x)
-1
>>> WedgeProduct(dx, fx*dy)(fx*e_x, e_y)
x**2
>>> WedgeProduct(e_x, e_y)(fy, None)
-e_x
```

You can nest wedge products.

```
>>> wp1 = WedgeProduct(dx, dy)
>>> WedgeProduct(wp1, dx)(e_x, e_y, e_x)
0
```

class sympy.diffgeom.**LieDerivative**(*v_field*, *expr*)

Lie derivative with respect to a vector field.

Explanation

The transport operator that defines the Lie derivative is the pushforward of the field to be derived along the integral curve of the field with respect to which one derives.

Examples

```
>>> from sympy.diffgeom.rn import R2_r, R2_p
>>> from sympy.diffgeom import (LieDerivative, TensorProduct)
```

```
>>> fx, fy = R2_r.base_scalars()
>>> e_x, e_y = R2_r.base_vectors()
>>> e_rho, e_theta = R2_p.base_vectors()
>>> dx, dy = R2_r.base_oneforms()
```

```
>>> LieDerivative(e_x, fy)
0
>>> LieDerivative(e_x, fx)
1
>>> LieDerivative(e_x, e_x)
0
```

The Lie derivative of a tensor field by another tensor field is equal to their commutator:

```
>>> LieDerivative(e_x, e_rho)
Commutator(e_x, e_rho)
>>> LieDerivative(e_x + e_y, fx)
1
```

```
>>> tp = TensorProduct(dx, dy)
>>> LieDerivative(e_x, tp)
LieDerivative(e_x, TensorProduct(dx, dy))
>>> LieDerivative(e_x, tp)
LieDerivative(e_x, TensorProduct(dx, dy))
```

class sympy.diffgeom.**BaseCovarDerivativeOp**(*coord_sys*, *index*, *christoffel*)

Covariant derivative operator with respect to a base vector.

Examples

```
>>> from sympy.diffgeom.rn import R2_r
>>> from sympy.diffgeom import BaseCovarDerivativeOp
>>> from sympy.diffgeom import metric_to_Christoffel_2nd, TensorProduct
```

```
>>> TP = TensorProduct
>>> fx, fy = R2_r.base_scalars()
>>> e_x, e_y = R2_r.base_vectors()
>>> dx, dy = R2_r.base_oneforms()
```

```
>>> ch = metric_to_Christoffel_2nd(TP(dx, dx) + TP(dy, dy))
>>> ch
[[[0, 0], [0, 0]], [[0, 0], [0, 0]]]
>>> cvd = BaseCovarDerivativeOp(R2_r, 0, ch)
>>> cvd(fx)
1
>>> cvd(fx*e_x)
e_x
```

class sympy.diffgeom.**CovarDerivativeOp**(*wrt*, *christoffel*)

Covariant derivative operator.

Examples

```
>>> from sympy.diffgeom.rn import R2_r
>>> from sympy.diffgeom import CovarDerivativeOp
>>> from sympy.diffgeom import metric_to_Christoffel_2nd, TensorProduct
>>> TP = TensorProduct
>>> fx, fy = R2_r.base_scalars()
>>> e_x, e_y = R2_r.base_vectors()
>>> dx, dy = R2_r.base_oneforms()
>>> ch = metric_to_Christoffel_2nd(TP(dx, dx) + TP(dy, dy))
```

```
>>> ch
[[[0, 0], [0, 0]], [[0, 0], [0, 0]]]
>>> cvd = CovarDerivativeOp(fx*e_x, ch)
>>> cvd(fx)
x
>>> cvd(fx*e_x)
x*e_x
```

sympy.diffgeom.**intcurve_series**(*vector_field*, *param*, *start_point*, *n=6*, *coord_sys=None*, *coeffs=False*)

Return the series expansion for an integral curve of the field.

Parameters

vector_field

the vector field for which an integral curve will be given

param

the argument of the function γ from R to the curve

start_point

the point which corresponds to $\gamma(0)$

n

the order to which to expand

coord_sys

the coordinate system in which to expand coeffs (default False) - if True return a list of elements of the expansion

Explanation

Integral curve is a function γ taking a parameter in R to a point in the manifold. It verifies the equation:

$V(f)\big(\gamma(t)\big) = \frac{d}{dt} f\big(\gamma(t)\big)$

where the given `vector_field` is denoted as V. This holds for any value t for the parameter and any scalar field f.

This equation can also be decomposed of a basis of coordinate functions $V(f_i)\big(\gamma(t)\big) = \frac{d}{dt} f_i\big(\gamma(t)\big) \quad \forall i$

This function returns a series expansion of $\gamma(t)$ in terms of the coordinate system `coord_sys`. The equations and expansions are necessarily done in coordinate-system-dependent way as there is no other way to represent movement between points on the manifold (i.e. there is no such thing as a difference of points for a general manifold).

Examples

Use the predefined R2 manifold:

```
>>> from sympy.abc import t, x, y
>>> from sympy.diffgeom.rn import R2_p, R2_r
>>> from sympy.diffgeom import intcurve_series
```

Specify a starting point and a vector field:

```
>>> start_point = R2_r.point([x, y])
>>> vector_field = R2_r.e_x
```

Calculate the series:

```
>>> intcurve_series(vector_field, t, start_point, n=3)
Matrix([
[t + x],
[    y]])
```

Or get the elements of the expansion in a list:

```
>>> series = intcurve_series(vector_field, t, start_point, n=3,␣
↪coeffs=True)
>>> series[0]
Matrix([
[x],
[y]])
>>> series[1]
Matrix([
[t],
[0]])
>>> series[2]
Matrix([
[0],
[0]])
```

The series in the polar coordinate system:

```
>>> series = intcurve_series(vector_field, t, start_point,
...              n=3, coord_sys=R2_p, coeffs=True)
>>> series[0]
Matrix([
[sqrt(x**2 + y**2)],
[      atan2(y, x)]])
>>> series[1]
Matrix([
[t*x/sqrt(x**2 + y**2)],
[   -t*y/(x**2 + y**2)]])
>>> series[2]
Matrix([
[t**2*(-x**2/(x**2 + y**2)**(3/2) + 1/sqrt(x**2 + y**2))/2],
[                                   t**2*x*y/(x**2 + y**2)**2]])
```

See also:

intcurve_diffequ (page 2816)

sympy.diffgeom.**intcurve_diffequ**(*vector_field*, *param*, *start_point*, *coord_sys=None*)

Return the differential equation for an integral curve of the field.

Parameters

vector_field

the vector field for which an integral curve will be given

param

the argument of the function γ from R to the curve

start_point

the point which corresponds to $\gamma(0)$

coord_sys

the coordinate system in which to give the equations

Returns

a tuple of (equations, initial conditions)

Explanation

Integral curve is a function γ taking a parameter in R to a point in the manifold. It verifies the equation:

$V(f)\big(\gamma(t)\big) = \frac{d}{dt} f\big(\gamma(t)\big)$

where the given `vector_field` is denoted as V. This holds for any value t for the parameter and any scalar field f.

This function returns the differential equation of $\gamma(t)$ in terms of the coordinate system `coord_sys`. The equations and expansions are necessarily done in coordinate-system-dependent way as there is no other way to represent movement between points on the manifold (i.e. there is no such thing as a difference of points for a general manifold).

Examples

Use the predefined R2 manifold:

```
>>> from sympy.abc import t
>>> from sympy.diffgeom.rn import R2, R2_p, R2_r
>>> from sympy.diffgeom import intcurve_diffequ
```

Specify a starting point and a vector field:

```
>>> start_point = R2_r.point([0, 1])
>>> vector_field = -R2.y*R2.e_x + R2.x*R2.e_y
```

Get the equation:

```
>>> equations, init_cond = intcurve_diffequ(vector_field, t, start_point)
>>> equations
[f_1(t) + Derivative(f_0(t), t), -f_0(t) + Derivative(f_1(t), t)]
>>> init_cond
[f_0(0), f_1(0) - 1]
```

The series in the polar coordinate system:

```
>>> equations, init_cond = intcurve_diffequ(vector_field, t, start_point,
 ↪ R2_p)
>>> equations
[Derivative(f_0(t), t), Derivative(f_1(t), t) - 1]
>>> init_cond
[f_0(0) - 1, f_1(0) - pi/2]
```

See also:

intcurve_series (page 2814)

sympy.diffgeom.**vectors_in_basis**(*expr*, *to_sys*)

Transform all base vectors in base vectors of a specified coord basis. While the new base vectors are in the new coordinate system basis, any coefficients are kept in the old system.

Examples

```
>>> from sympy.diffgeom import vectors_in_basis
>>> from sympy.diffgeom.rn import R2_r, R2_p
```

```
>>> vectors_in_basis(R2_r.e_x, R2_p)
-y*e_theta/(x**2 + y**2) + x*e_rho/sqrt(x**2 + y**2)
>>> vectors_in_basis(R2_p.e_r, R2_r)
sin(theta)*e_y + cos(theta)*e_x
```

sympy.diffgeom.**twoform_to_matrix**(*expr*)

Return the matrix representing the twoform.

For the twoform w return the matrix M such that $M[i,j] = w(e_i, e_j)$, where e_i is the i-th base vector field for the coordinate system in which the expression of w is given.

Examples

```
>>> from sympy.diffgeom.rn import R2
>>> from sympy.diffgeom import twoform_to_matrix, TensorProduct
>>> TP = TensorProduct
```

```
>>> twoform_to_matrix(TP(R2.dx, R2.dx) + TP(R2.dy, R2.dy))
Matrix([
[1, 0],
[0, 1]])
>>> twoform_to_matrix(R2.x*TP(R2.dx, R2.dx) + TP(R2.dy, R2.dy))
Matrix([
[x, 0],
[0, 1]])
>>> twoform_to_matrix(TP(R2.dx, R2.dx) + TP(R2.dy, R2.dy) - TP(R2.dx, R2.
↪dy)/2)
Matrix([
[   1, 0],
[-1/2, 1]])
```

sympy.diffgeom.**metric_to_Christoffel_1st**(*expr*)

Return the nested list of Christoffel symbols for the given metric. This returns the Christoffel symbol of first kind that represents the Levi-Civita connection for the given metric.

Examples

```
>>> from sympy.diffgeom.rn import R2
>>> from sympy.diffgeom import metric_to_Christoffel_1st, TensorProduct
>>> TP = TensorProduct
```

```
>>> metric_to_Christoffel_1st(TP(R2.dx, R2.dx) + TP(R2.dy, R2.dy))
[[[0, 0], [0, 0]], [[0, 0], [0, 0]]]
```

(continues on next page)

(continued from previous page)

```
>>> metric_to_Christoffel_1st(R2.x*TP(R2.dx, R2.dx) + TP(R2.dy, R2.dy))
[[[1/2, 0], [0, 0]], [[0, 0], [0, 0]]]
```

sympy.diffgeom.**metric_to_Christoffel_2nd**(*expr*)

Return the nested list of Christoffel symbols for the given metric. This returns the Christoffel symbol of second kind that represents the Levi-Civita connection for the given metric.

Examples

```
>>> from sympy.diffgeom.rn import R2
>>> from sympy.diffgeom import metric_to_Christoffel_2nd, TensorProduct
>>> TP = TensorProduct
```

```
>>> metric_to_Christoffel_2nd(TP(R2.dx, R2.dx) + TP(R2.dy, R2.dy))
[[[0, 0], [0, 0]], [[0, 0], [0, 0]]]
>>> metric_to_Christoffel_2nd(R2.x*TP(R2.dx, R2.dx) + TP(R2.dy, R2.dy))
[[[1/(2*x), 0], [0, 0]], [[0, 0], [0, 0]]]
```

sympy.diffgeom.**metric_to_Riemann_components**(*expr*)

Return the components of the Riemann tensor expressed in a given basis.

Given a metric it calculates the components of the Riemann tensor in the canonical basis of the coordinate system in which the metric expression is given.

Examples

```
>>> from sympy import exp
>>> from sympy.diffgeom.rn import R2
>>> from sympy.diffgeom import metric_to_Riemann_components,␣
↪TensorProduct
>>> TP = TensorProduct
```

```
>>> metric_to_Riemann_components(TP(R2.dx, R2.dx) + TP(R2.dy, R2.dy))
[[[[0, 0], [0, 0]], [[0, 0], [0, 0]]], [[[0, 0], [0, 0]], [[0, 0], [0,␣
↪0]]]]
>>> non_trivial_metric = exp(2*R2.r)*TP(R2.dr, R2.dr) +         R2.
↪r**2*TP(R2.dtheta, R2.dtheta)
>>> non_trivial_metric
exp(2*rho)*TensorProduct(drho, drho) + rho**2*TensorProduct(dtheta,␣
↪dtheta)
>>> riemann = metric_to_Riemann_components(non_trivial_metric)
>>> riemann[0, :, :, :]
[[[0, 0], [0, 0]], [[0, exp(-2*rho)*rho], [-exp(-2*rho)*rho, 0]]]
>>> riemann[1, :, :, :]
[[[0, -1/rho], [1/rho, 0]], [[0, 0], [0, 0]]]
```

sympy.diffgeom.**metric_to_Ricci_components**(*expr*)

Return the components of the Ricci tensor expressed in a given basis.

Given a metric it calculates the components of the Ricci tensor in the canonical basis of the coordinate system in which the metric expression is given.

Examples

```
>>> from sympy import exp
>>> from sympy.diffgeom.rn import R2
>>> from sympy.diffgeom import metric_to_Ricci_components, TensorProduct
>>> TP = TensorProduct
```

```
>>> metric_to_Ricci_components(TP(R2.dx, R2.dx) + TP(R2.dy, R2.dy))
[[0, 0], [0, 0]]
>>> non_trivial_metric = exp(2*R2.r)*TP(R2.dr, R2.dr) +
↪            R2.r**2*TP(R2.dtheta, R2.dtheta)
>>> non_trivial_metric
exp(2*rho)*TensorProduct(drho, drho) + rho**2*TensorProduct(dtheta,
↪dtheta)
>>> metric_to_Ricci_components(non_trivial_metric)
[[1/rho, 0], [0, exp(-2*rho)*rho]]
```

Plotting

Introduction

The plotting module allows you to make 2-dimensional and 3-dimensional plots. Presently the plots are rendered using `matplotlib` as a backend. It is also possible to plot 2-dimensional plots using a *TextBackend* (page 2869) if you do not have `matplotlib`.

The plotting module has the following functions:

- *plot()* (page 2825): Plots 2D line plots.
- *plot_parametric()* (page 2828): Plots 2D parametric plots.
- *plot_implicit()* (page 2847): Plots 2D implicit and region plots.
- *plot3d()* (page 2838): Plots 3D plots of functions in two variables.
- *plot3d_parametric_line()* (page 2839): Plots 3D line plots, defined by a parameter.
- *plot3d_parametric_surface()* (page 2846): Plots 3D parametric surface plots.

The above functions are only for convenience and ease of use. It is possible to plot any plot by passing the corresponding `Series` class to *Plot* (page 2821) as argument.

Plot Class

class sympy.plotting.plot.**Plot**(**args*, *title=None*, *xlabel=None*, *ylabel=None*, *zlabel=None*, *aspect_ratio='auto'*, *xlim=None*, *ylim=None*, *axis_center='auto'*, *axis=True*, *xscale='linear'*, *yscale='linear'*, *legend=False*, *autoscale=True*, *margin=0*, *annotations=None*, *markers=None*, *rectangles=None*, *fill=None*, *backend='default'*, *size=None*, ***kwargs*)

The central class of the plotting module.

Explanation

For interactive work the function *plot()* (page 2825) is better suited.

This class permits the plotting of SymPy expressions using numerous backends (matplotlib, textplot, the old pyglet module for SymPy, Google charts api, etc).

The figure can contain an arbitrary number of plots of SymPy expressions, lists of coordinates of points, etc. Plot has a private attribute _series that contains all data series to be plotted (expressions for lines or surfaces, lists of points, etc (all subclasses of BaseSeries)). Those data series are instances of classes not imported by `from sympy import *`.

The customization of the figure is on two levels. Global options that concern the figure as a whole (e.g. title, xlabel, scale, etc) and per-data series options (e.g. name) and aesthetics (e.g. color, point shape, line type, etc.).

The difference between options and aesthetics is that an aesthetic can be a function of the coordinates (or parameters in a parametric plot). The supported values for an aesthetic are:

- None (the backend uses default values)
- a constant
- a function of one variable (the first coordinate or parameter)
- a function of two variables (the first and second coordinate or parameters)
- a function of three variables (only in nonparametric 3D plots)

Their implementation depends on the backend so they may not work in some backends.

If the plot is parametric and the arity of the aesthetic function permits it the aesthetic is calculated over parameters and not over coordinates. If the arity does not permit calculation over parameters the calculation is done over coordinates.

Only cartesian coordinates are supported for the moment, but you can use the parametric plots to plot in polar, spherical and cylindrical coordinates.

The arguments for the constructor Plot must be subclasses of BaseSeries.

Any global option can be specified as a keyword argument.

The global options for a figure are:

- title : str
- xlabel : str or Symbol

- ylabel : str or Symbol
- zlabel : str or Symbol
- legend : bool
- xscale : {'linear', 'log'}
- yscale : {'linear', 'log'}
- axis : bool
- axis_center : tuple of two floats or {'center', 'auto'}
- xlim : tuple of two floats
- ylim : tuple of two floats
- aspect_ratio : tuple of two floats or {'auto'}
- autoscale : bool
- margin : float in [0, 1]
- backend : {'default', 'matplotlib', 'text'} or a subclass of BaseBackend
- size : optional tuple of two floats, (width, height); default: None

The per data series options and aesthetics are: There are none in the base series. See below for options for subclasses.

Some data series support additional aesthetics or options:

LineOver1DRangeSeries (page 2866), *Parametric2DLineSeries* (page 2866), and *Parametric3DLineSeries* (page 2867) support the following:

Aesthetics:

- **line_color**
  [string, or float, or function, optional] Specifies the color for the plot, which depends on the backend being used.

  For example, if `MatplotlibBackend` is being used, then Matplotlib string colors are acceptable (`"red"`, `"r"`, `"cyan"`, `"c"`, ...). Alternatively, we can use a float number, 0 < color < 1, wrapped in a string (for example, `line_color="0.5"`) to specify grayscale colors. Alternatively, We can specify a function returning a single float value: this will be used to apply a color-loop (for example, `line_color=lambda x: math.cos(x)`).

  Note that by setting line_color, it would be applied simultaneously to all the series.

Options:

- label : str
- steps : bool
- integers_only : bool

SurfaceOver2DRangeSeries (page 2867) and *ParametricSurfaceSeries* (page 2867) support the following:

Aesthetics:

- surface_color : function which returns a float.

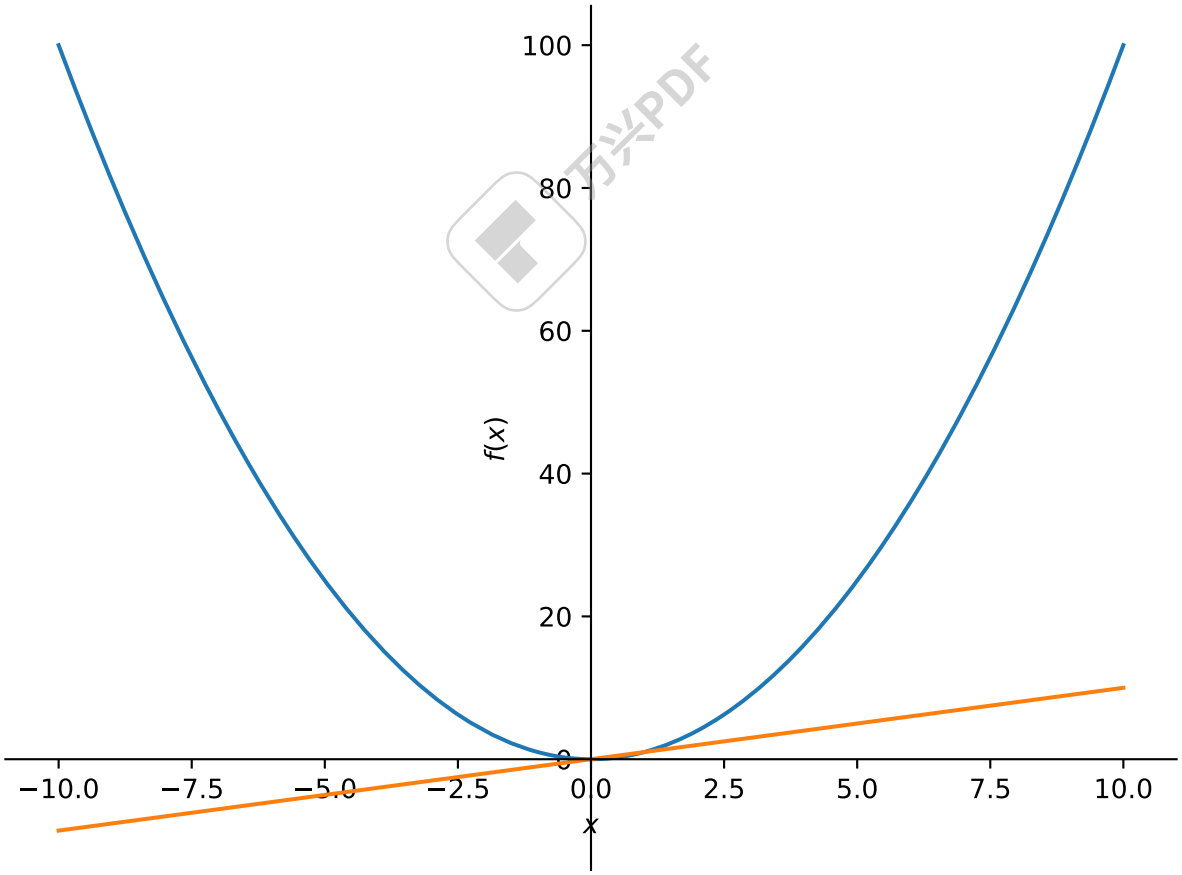
append(*arg*)

Adds an element from a plot's series to an existing plot.

Examples

Consider two `Plot` objects, `p1` and `p2`. To add the second plot's first series object to the first, use the `append` method, like so:

```
>>> from sympy import symbols
>>> from sympy.plotting import plot
>>> x = symbols('x')
>>> p1 = plot(x*x, show=False)
>>> p2 = plot(x, show=False)
>>> p1.append(p2[0])
>>> p1
Plot object containing:
[0]: cartesian line: x**2 for x over (-10.0, 10.0)
[1]: cartesian line: x for x over (-10.0, 10.0)
>>> p1.show()
```



See also:

extend (page 2823)

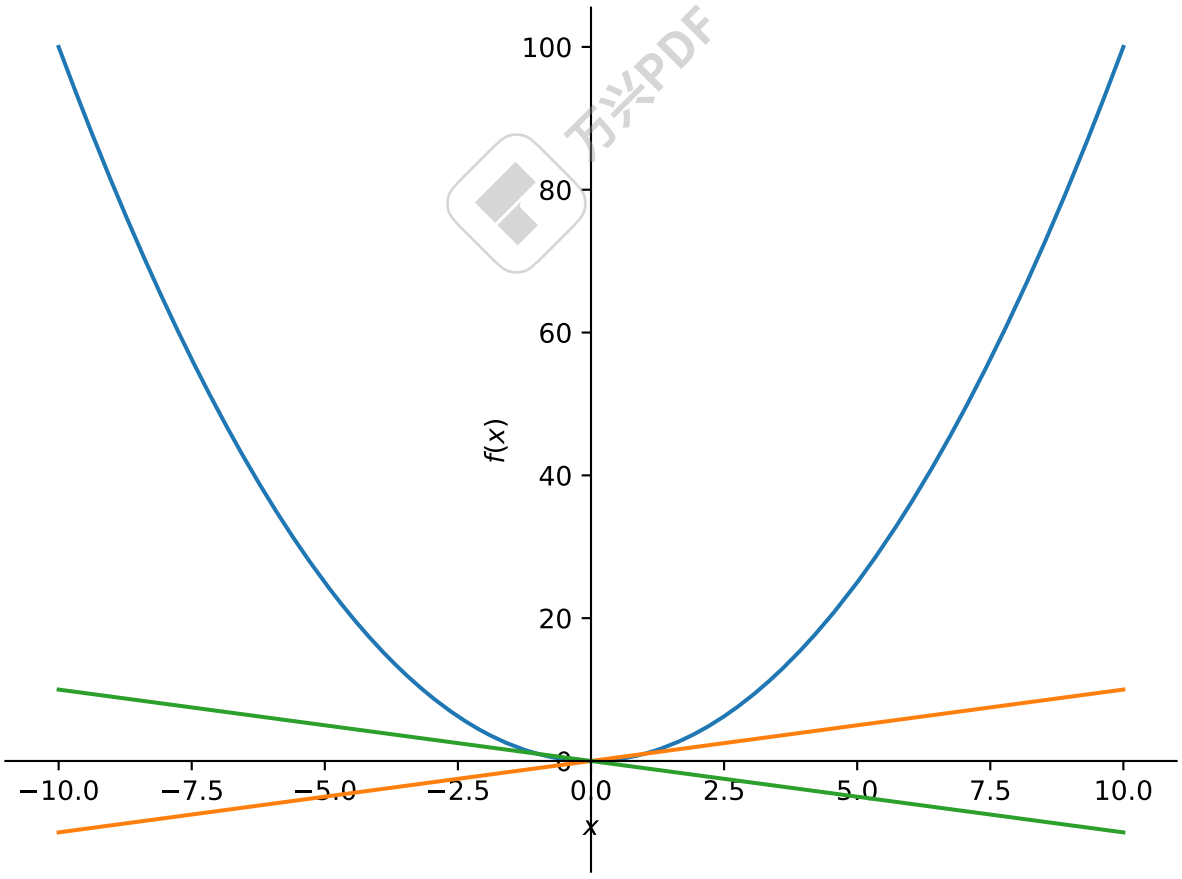
extend(*arg*)

Adds all series from another plot.

Examples

Consider two `Plot` objects, `p1` and `p2`. To add the second plot to the first, use the `extend` method, like so:

```
>>> from sympy import symbols
>>> from sympy.plotting import plot
>>> x = symbols('x')
>>> p1 = plot(x**2, show=False)
>>> p2 = plot(x, -x, show=False)
>>> p1.extend(p2)
>>> p1
Plot object containing:
[0]: cartesian line: x**2 for x over (-10.0, 10.0)
[1]: cartesian line: x for x over (-10.0, 10.0)
[2]: cartesian line: -x for x over (-10.0, 10.0)
>>> p1.show()
```



Plotting Function Reference

sympy.plotting.plot.**plot**(**args*, *show=True*, ***kwargs*)

Plots a function of a single variable as a curve.

Parameters

args :

The first argument is the expression representing the function of single variable to be plotted.

The last argument is a 3-tuple denoting the range of the free variable. e.g. `(x, 0, 5)`

Typical usage examples are in the followings:

- **Plotting a single expression with a single range.**
  `plot(expr, range, **kwargs)`
- **Plotting a single expression with the default range (-10, 10).**
  `plot(expr, **kwargs)`
- **Plotting multiple expressions with a single range.**
  `plot(expr1, expr2, ..., range, **kwargs)`
- **Plotting multiple expressions with multiple ranges.**
  `plot((expr1, range1), (expr2, range2), ..., **kwargs)`

It is best practice to specify range explicitly because default range may change in the future if a more advanced default range detection algorithm is implemented.

show : bool, optional

The default value is set to `True`. Set show to `False` and the function will not display the plot. The returned instance of the `Plot` class can then be used to save or display the plot by calling the `save()` and `show()` methods respectively.

line_color : string, or float, or function, optional

Specifies the color for the plot. See `Plot` to see how to set color for the plots. Note that by setting `line_color`, it would be applied simultaneously to all the series.

title : str, optional

Title of the plot. It is set to the latex representation of the expression, if the plot has only one expression.

label : str, optional

The label of the expression in the plot. It will be used when called with `legend`. Default is the name of the expression. e.g. `sin(x)`

xlabel : str or expression, optional

Label for the x-axis.

ylabel : str or expression, optional

Label for the y-axis.

xscale : 'linear' or 'log', optional

Sets the scaling of the x-axis.

yscale : 'linear' or 'log', optional

Sets the scaling of the y-axis.

axis_center : (float, float), optional

Tuple of two floats denoting the coordinates of the center or {'center', 'auto'}

xlim : (float, float), optional

Denotes the x-axis limits, `(min, max)`.

ylim : (float, float), optional

Denotes the y-axis limits, `(min, max)`.

annotations : list, optional

A list of dictionaries specifying the type of annotation required. The keys in the dictionary should be equivalent to the arguments of the `matplotlib`'s `annotate()` method.

markers : list, optional

A list of dictionaries specifying the type the markers required. The keys in the dictionary should be equivalent to the arguments of the `matplotlib`'s `plot()` function along with the marker related keyworded arguments.

rectangles : list, optional

A list of dictionaries specifying the dimensions of the rectangles to be plotted. The keys in the dictionary should be equivalent to the arguments of the `matplotlib`'s `Rectangle` class.

fill : dict, optional

A dictionary specifying the type of color filling required in the plot. The keys in the dictionary should be equivalent to the arguments of the `matplotlib`'s `fill_between()` method.

adaptive : bool, optional

The default value is set to `True`. Set adaptive to `False` and specify `nb_of_points` if uniform sampling is required.

The plotting uses an adaptive algorithm which samples recursively to accurately plot. The adaptive algorithm uses a random point near the midpoint of two points that has to be further sampled. Hence the same plots can appear slightly different.

depth : int, optional

Recursion depth of the adaptive algorithm. A depth of value n samples a maximum of 2^n points.

If the `adaptive` flag is set to `False`, this will be ignored.

nb_of_points : int, optional

Used when the `adaptive` is set to `False`. The function is uniformly sampled at `nb_of_points` number of points.

If the `adaptive` flag is set to `True`, this will be ignored.

size : (float, float), optional

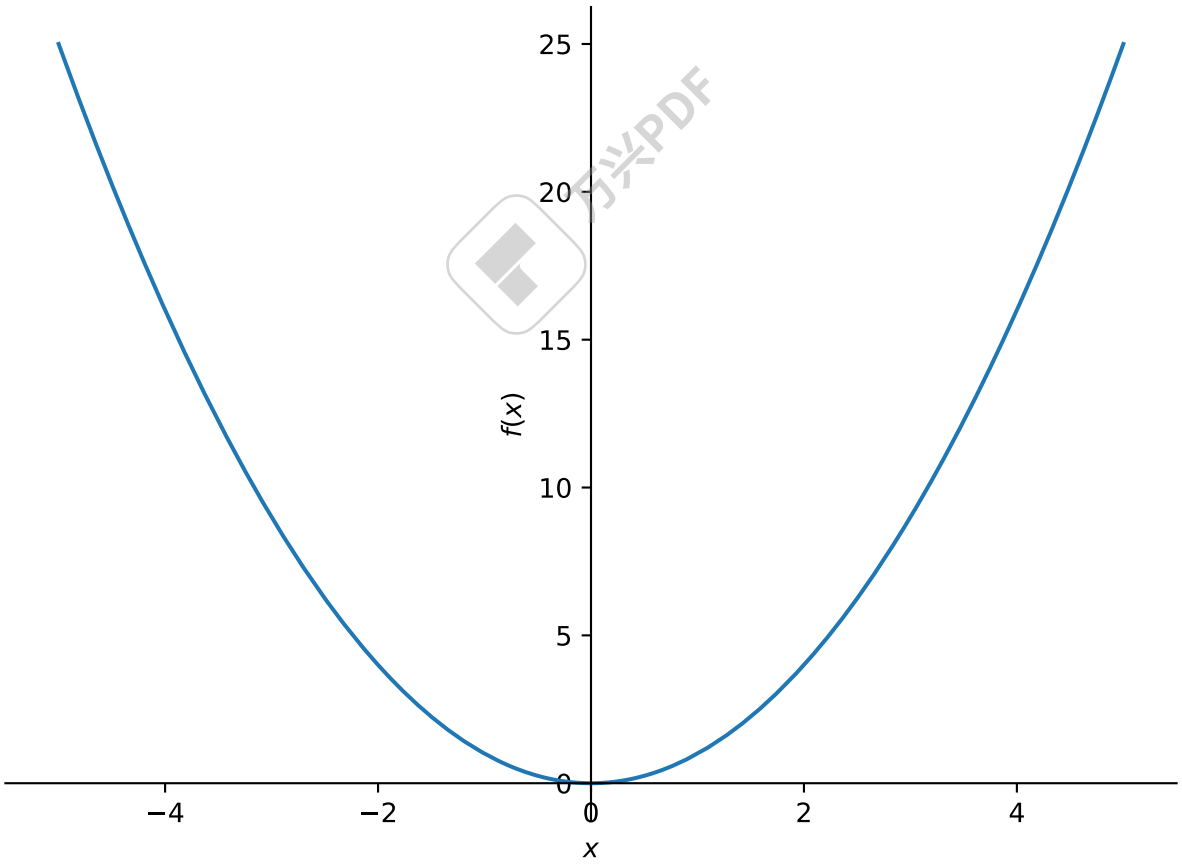
A tuple in the form (width, height) in inches to specify the size of the overall figure. The default value is set to `None`, meaning the size will be set by the default backend.

Examples

```
>>> from sympy import symbols
>>> from sympy.plotting import plot
>>> x = symbols('x')
```

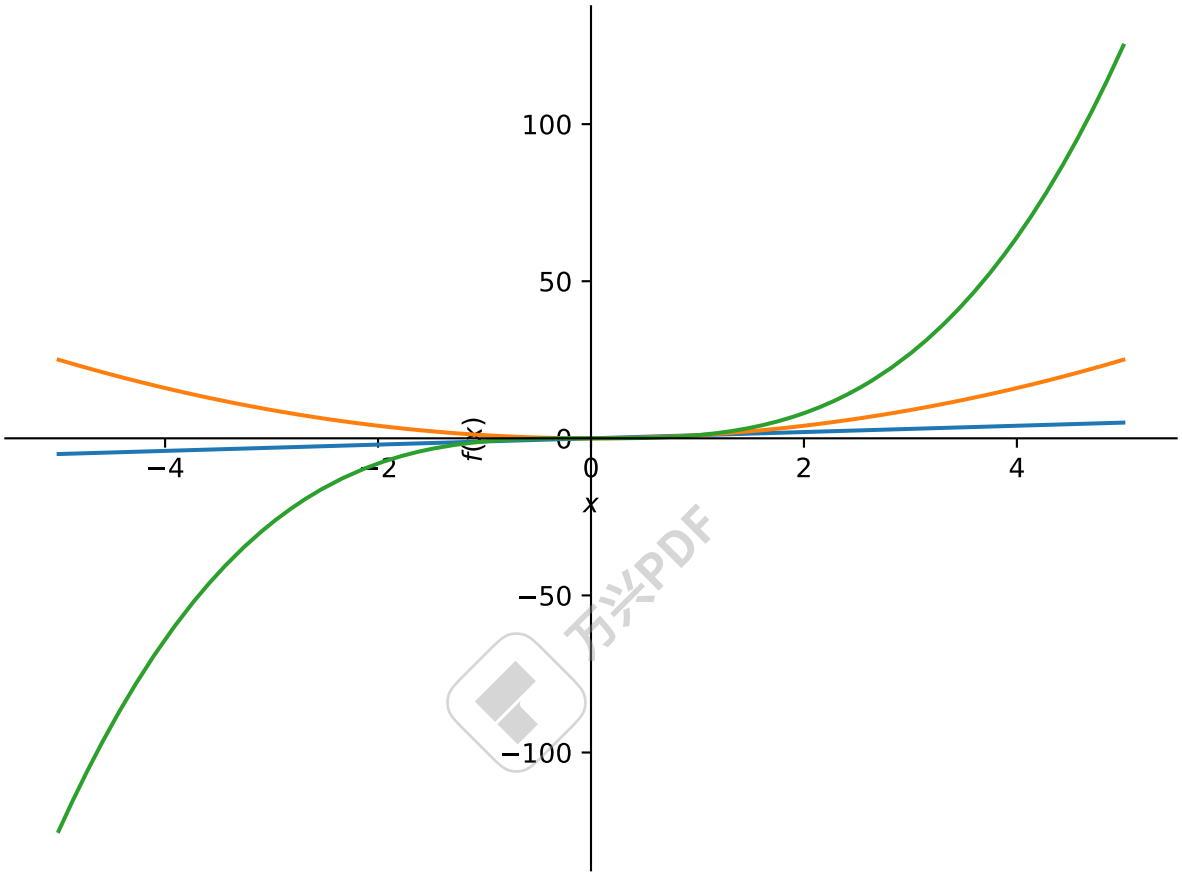
Single Plot

```
>>> plot(x**2, (x, -5, 5))
Plot object containing:
[0]: cartesian line: x**2 for x over (-5.0, 5.0)
```



Multiple plots with single range.

```
>>> plot(x, x**2, x**3, (x, -5, 5))
Plot object containing:
[0]: cartesian line: x for x over (-5.0, 5.0)
[1]: cartesian line: x**2 for x over (-5.0, 5.0)
[2]: cartesian line: x**3 for x over (-5.0, 5.0)
```



Multiple plots with different ranges.

```
>>> plot((x**2, (x, -6, 6)), (x, (x, -5, 5)))
Plot object containing:
[0]: cartesian line: x**2 for x over (-6.0, 6.0)
[1]: cartesian line: x for x over (-5.0, 5.0)
```

No adaptive sampling.

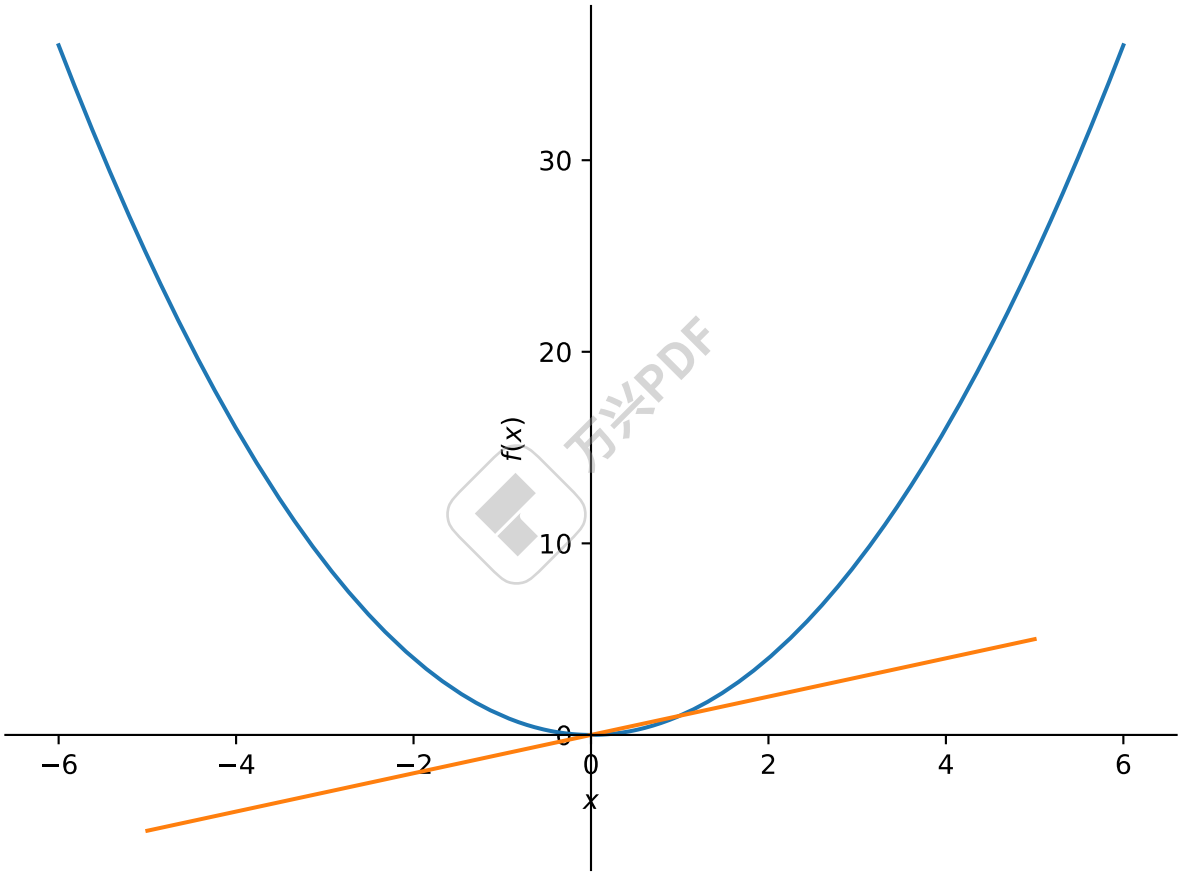
```
>>> plot(x**2, adaptive=False, nb_of_points=400)
Plot object containing:
[0]: cartesian line: x**2 for x over (-10.0, 10.0)
```

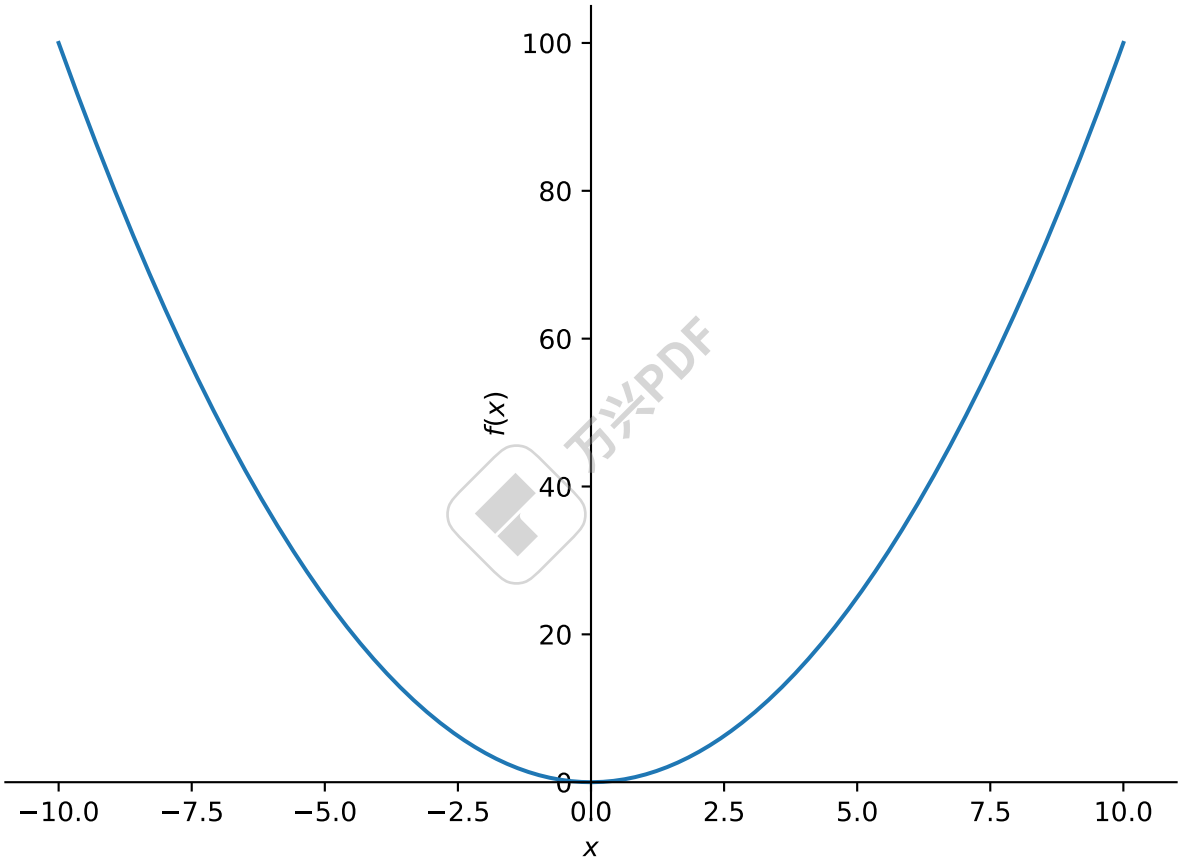
See also:

Plot (page 2821), *LineOver1DRangeSeries* (page 2866)

sympy.plotting.plot.**plot_parametric**(**args*, *show=True*, ***kwargs*)

Plots a 2D parametric curve.


30
20
10
0
f(x)
−6
−4
−2
0
2
4
6
x


100
80
60
40
20
0
f(x)
−10.0
−7.5
−5.0
−2.5
0.0
2.5
5.0
7.5
10.0
x

Parameters

args

Common specifications are:

- **Plotting a single parametric curve with a range**
  `plot_parametric((expr_x, expr_y), range)`
- **Plotting multiple parametric curves with the same range**
  `plot_parametric((expr_x, expr_y), ..., range)`
- **Plotting multiple parametric curves with different ranges**
  `plot_parametric((expr_x, expr_y, range), ...)`

`expr_x` is the expression representing x component of the parametric function.

`expr_y` is the expression representing y component of the parametric function.

`range` is a 3-tuple denoting the parameter symbol, start and stop. For example, `(u, 0, 5)`.

If the range is not specified, then a default range of (-10, 10) is used.

However, if the arguments are specified as `(expr_x, expr_y, range), ...`, you must specify the ranges for each expressions manually.

Default range may change in the future if a more advanced algorithm is implemented.

adaptive : bool, optional

Specifies whether to use the adaptive sampling or not.

The default value is set to `True`. Set adaptive to `False` and specify `nb_of_points` if uniform sampling is required.

depth : int, optional

The recursion depth of the adaptive algorithm. A depth of value n samples a maximum of 2^n points.

nb_of_points : int, optional

Used when the `adaptive` flag is set to `False`.

Specifies the number of the points used for the uniform sampling.

line_color : string, or float, or function, optional

Specifies the color for the plot. See `Plot` to see how to set color for the plots. Note that by setting `line_color`, it would be applied simultaneously to all the series.

label : str, optional

The label of the expression in the plot. It will be used when called with `legend`. Default is the name of the expression. e.g. `sin(x)`

xlabel : str, optional

Label for the x-axis.

ylabel : str, optional

Label for the y-axis.

xscale : 'linear' or 'log', optional

Sets the scaling of the x-axis.

yscale : 'linear' or 'log', optional

Sets the scaling of the y-axis.

axis_center : (float, float), optional

Tuple of two floats denoting the coordinates of the center or {'center', 'auto'}

xlim : (float, float), optional

Denotes the x-axis limits, `(min, max)`.

ylim : (float, float), optional

Denotes the y-axis limits, `(min, max)`.

size : (float, float), optional

A tuple in the form (width, height) in inches to specify the size of the overall figure. The default value is set to `None`, meaning the size will be set by the default backend.

Examples

```
>>> from sympy import plot_parametric, symbols, cos, sin
>>> u = symbols('u')
```

A parametric plot with a single expression:

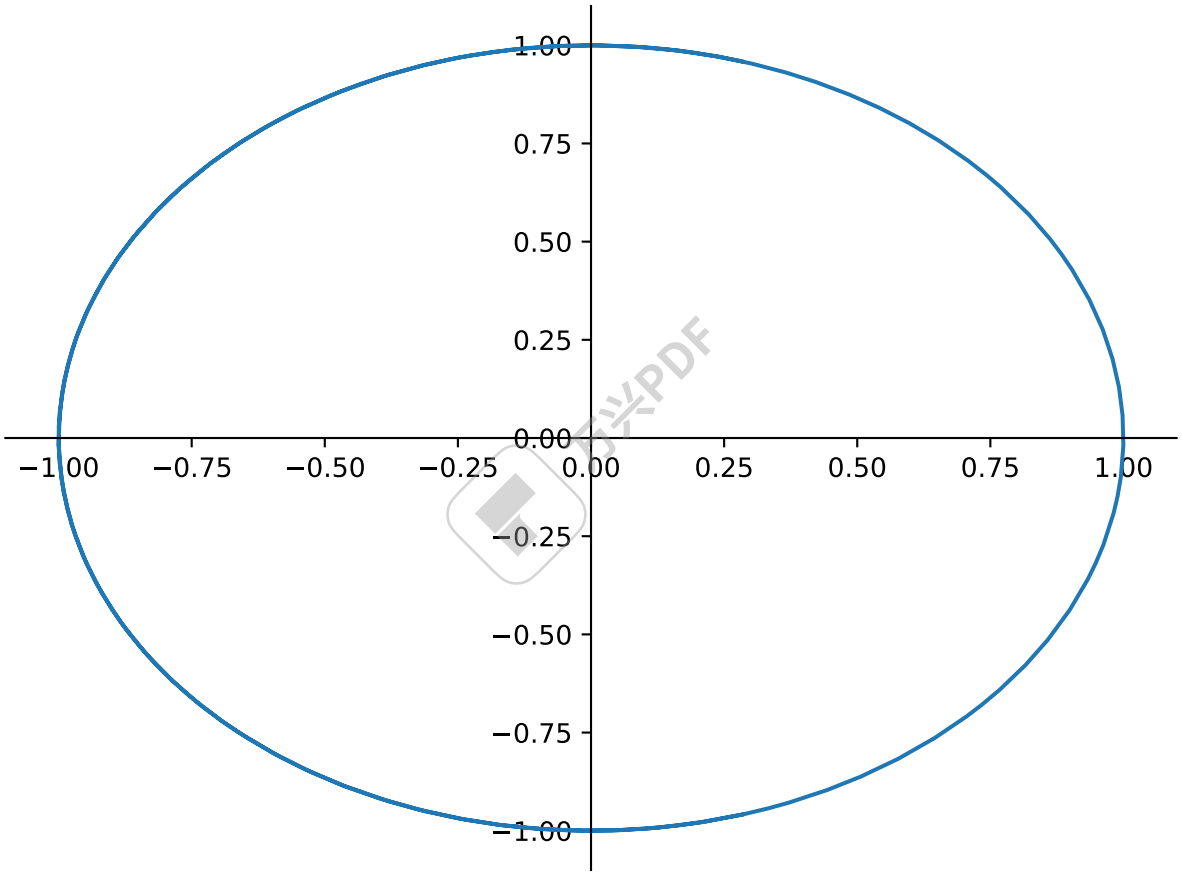
```
>>> plot_parametric((cos(u), sin(u)), (u, -5, 5))
Plot object containing:
[0]: parametric cartesian line: (cos(u), sin(u)) for u over (-5.0, 5.0)
```

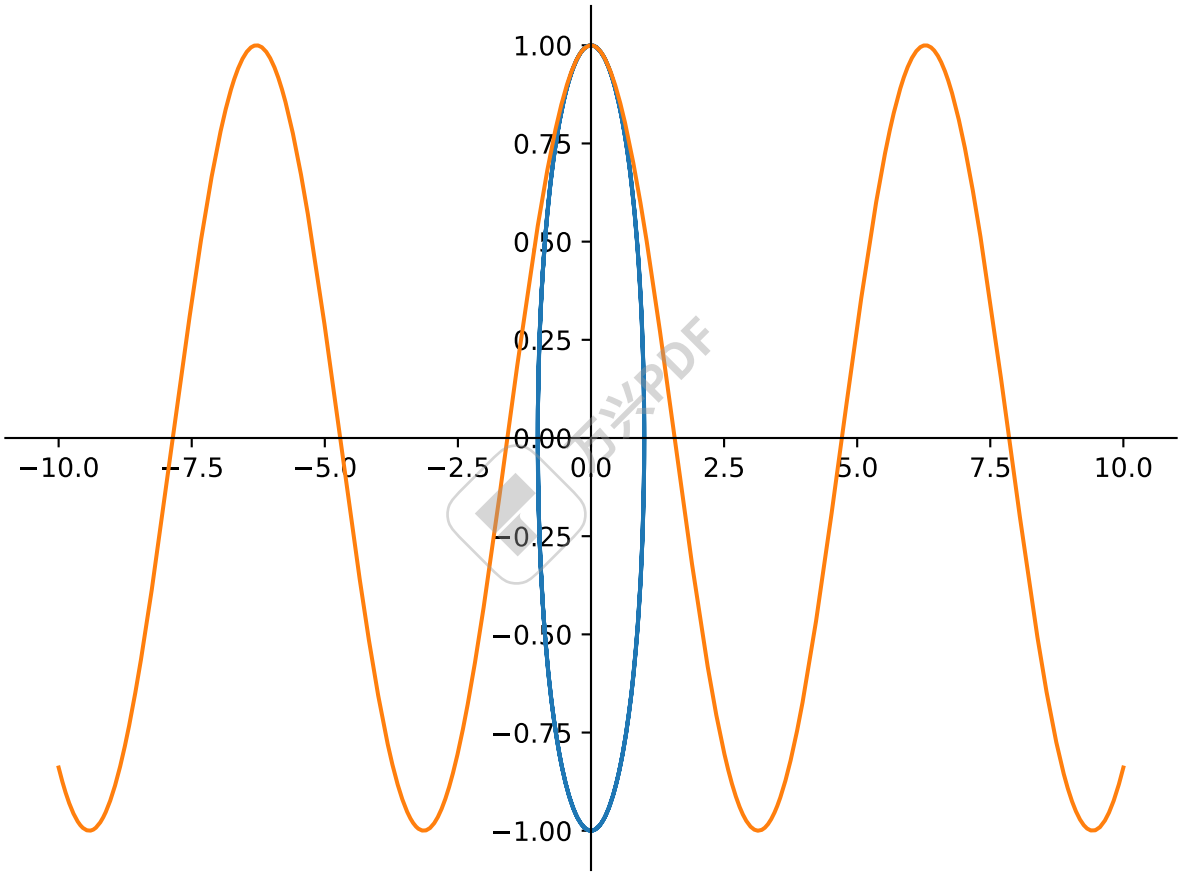
A parametric plot with multiple expressions with the same range:

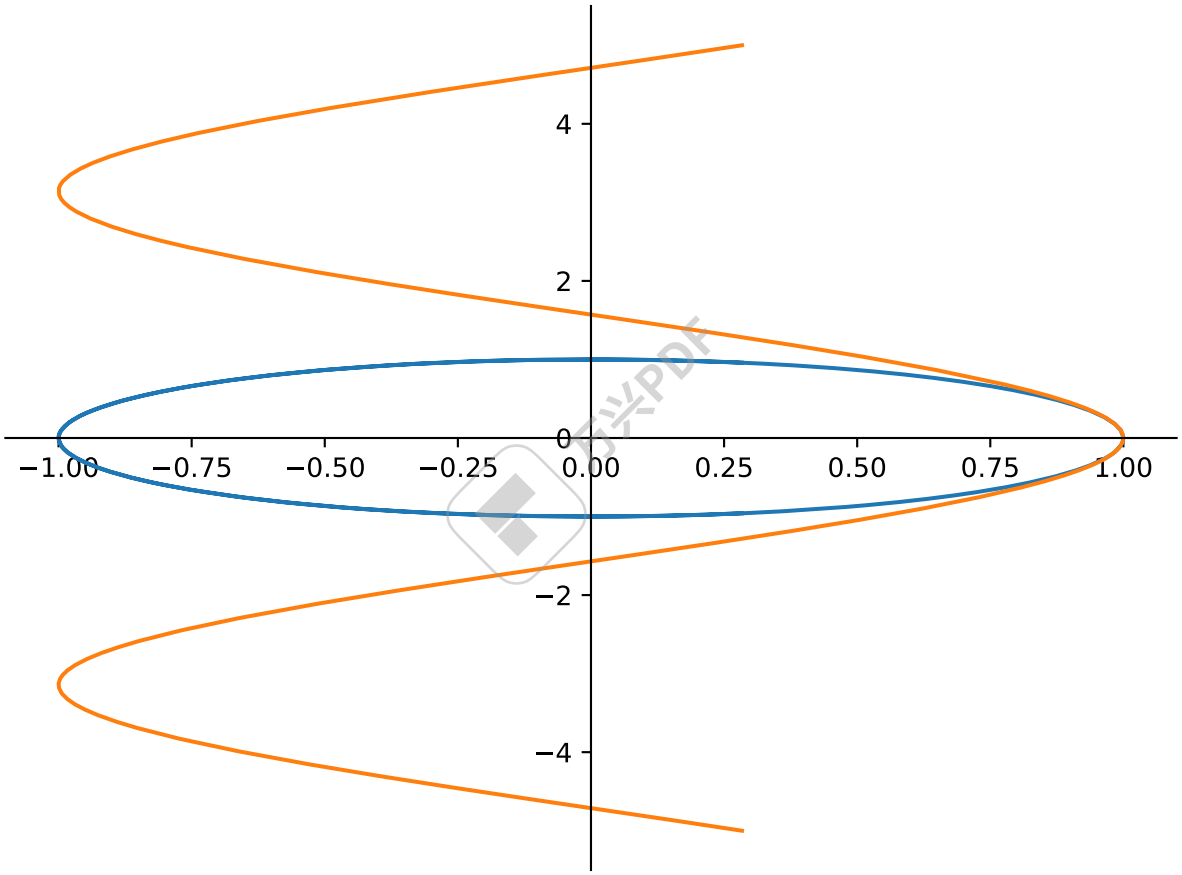
```
>>> plot_parametric((cos(u), sin(u)), (u, cos(u)), (u, -10, 10))
Plot object containing:
[0]: parametric cartesian line: (cos(u), sin(u)) for u over (-10.0, 10.0)
[1]: parametric cartesian line: (u, cos(u)) for u over (-10.0, 10.0)
```

A parametric plot with multiple expressions with different ranges for each curve:

```
>>> plot_parametric((cos(u), sin(u), (u, -5, 5)),
...     (cos(u), u, (u, -5, 5)))
Plot object containing:
[0]: parametric cartesian line: (cos(u), sin(u)) for u over (-5.0, 5.0)
[1]: parametric cartesian line: (cos(u), u) for u over (-5.0, 5.0)
```


1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
−1.00
−0.75
−0.50
−0.25
0.00
0.25
0.50
0.75
1.00


1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
−10.0
−7.5
−5.0
−2.5
0.0
2.5
5.0
7.5
10.0



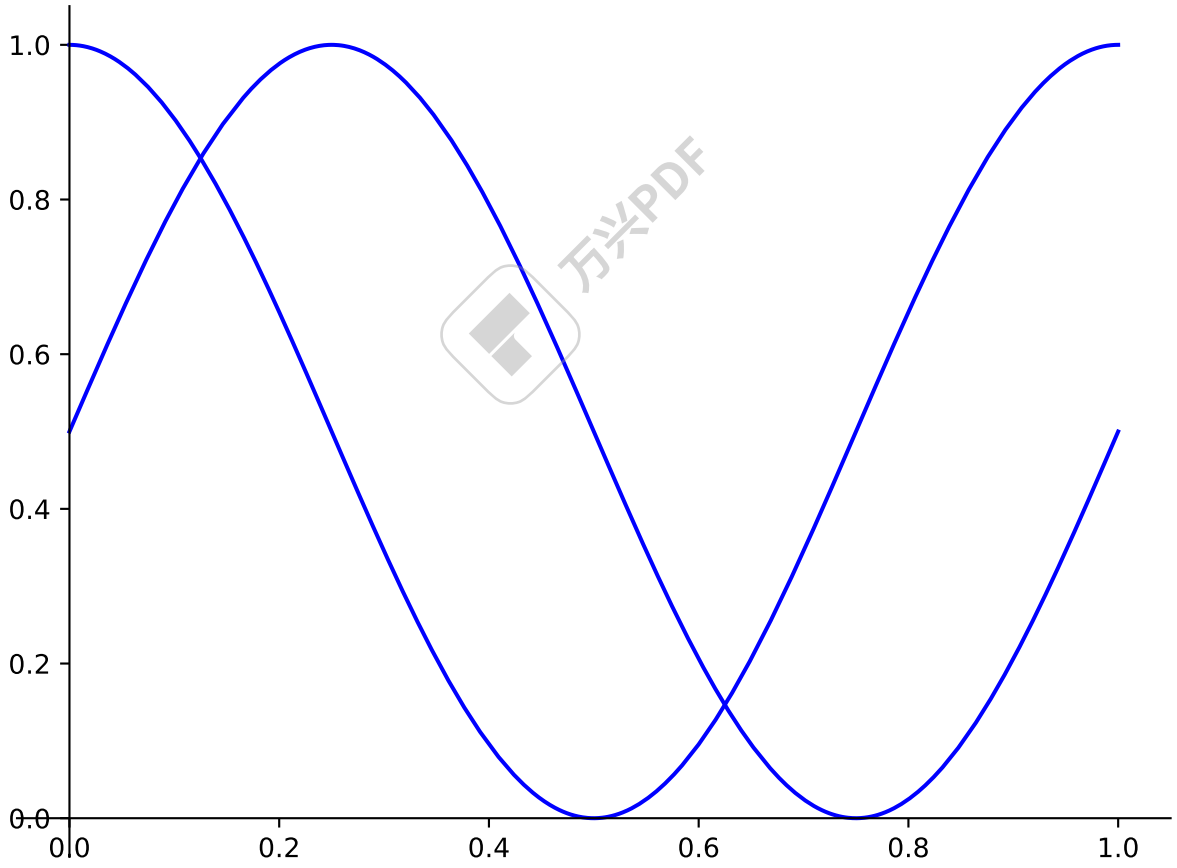
Notes

The plotting uses an adaptive algorithm which samples recursively to accurately plot the curve. The adaptive algorithm uses a random point near the midpoint of two points that has to be further sampled. Hence, repeating the same plot command can give slightly different results because of the random sampling.

If there are multiple plots, then the same optional arguments are applied to all the plots drawn in the same canvas. If you want to set these options separately, you can index the returned `Plot` object and set it.

For example, when you specify `line_color` once, it would be applied simultaneously to both series.

```
>>> from sympy import pi
>>> expr1 = (u, cos(2*pi*u)/2 + 1/2)
>>> expr2 = (u, sin(2*pi*u)/2 + 1/2)
>>> p = plot_parametric(expr1, expr2, (u, 0, 1), line_color='blue')
```



If you want to specify the line color for the specific series, you should index each item and apply the property manually.

```
>>> p[0].line_color = 'red'
>>> p.show()
```