``**phia**`` : numeric type or SymPy Symbol

The phase difference between transmitted and reflected component for output mode a.

``**phib**`` : numeric type or SymPy Symbol

The phase difference between transmitted and reflected component for output mode b.

Returns

SymPy Matrix

A 4x4 matrix representing the PBS. This matrix acts on a 4x1 vector whose first two entries are the Jones vector on one of the PBS ports, and the last two entries the Jones vector on the other port.

Examples

Generic polarizing beam-splitter.

```
>>> from sympy import pprint, symbols
>>> from sympy.physics.optics.polarization import polarizing_beam_
↪splitter
>>> Ts, Rs, Tp, Rp = symbols(r"Ts, Rs, Tp, Rp", positive=True)
>>> phia, phib = symbols("phi_a, phi_b", real=True)
>>> PBS = polarizing_beam_splitter(Tp, Rs, Ts, Rp, phia, phib)
>>> pprint(PBS, use_unicode=False)
[   ____                          ____                        ]
[ \/ Tp            0          I*\/ Rp           0             ]
[                                                             ]
[                 ____                          ____  I*phi_a]
[   0           \/ Ts             0      -I*\/ Rs *e           ]
[                                                             ]
[  ____                         ____                          ]
[I*\/ Rp           0          \/ Tp            0             ]
[                                                             ]
[                 ____  I*phi_b                         ____  ]
[   0      -I*\/ Rs *e            0           \/ Ts          ]
```

sympy.physics.optics.polarization.**quarter_wave_retarder**(*theta*)

A quarter-wave retarder Jones matrix at angle $theta$.

Parameters

``**theta**`` : numeric type or SymPy Symbol

The angle of the fast axis relative to the horizontal plane.

Returns

SymPy Matrix

A Jones matrix representing the retarder.

Examples

A generic quarter-wave plate.

```
>>> from sympy import pprint, symbols
>>> from sympy.physics.optics.polarization import quarter_wave_retarder
>>> theta= symbols("theta", real=True)
>>> QWP = quarter_wave_retarder(theta)
>>> pprint(QWP, use_unicode=True)
⎡                          -i·π            -i·π                ⎤
⎢                          ────            ────                ⎥
⎢⎛     2         2   ⎞     4                 4                  ⎥
⎢⎝i·sin (θ) + cos (θ)⎠·e        (1 - i)·e     ·sin(θ)·cos(θ)⎥
⎢                                                              ⎥
⎢          -i·π                                        -i·π    ⎥
⎢          ────                                        ────    ⎥
⎢            4                  ⎛   2           2   ⎞    4    ⎥
⎣(1 - i)·e     ·sin(θ)·cos(θ)     ⎝sin (θ) + i·cos (θ)⎠·e      ⎦
```

sympy.physics.optics.polarization.**reflective_filter**(*R*)

A reflective filter Jones matrix with reflectance *R*.

Parameters

``**R**`` : numeric type or SymPy Symbol

The reflectance of the filter.

Returns

SymPy Matrix

A Jones matrix representing the filter.

Examples

A generic filter.

```
>>> from sympy import pprint, symbols
>>> from sympy.physics.optics.polarization import reflective_filter
>>> R = symbols("R", real=True)
>>> pprint(reflective_filter(R), use_unicode=True)
⎡√R   0 ⎤
⎢       ⎥
⎣0   -√R⎦
```

sympy.physics.optics.polarization.**stokes_vector**(*psi*, *chi*, *p=1*, *I=1*)

A Stokes vector corresponding to a polarization ellipse with `psi` tilt, and `chi` circularity.

Parameters

``**psi**`` : numeric type or SymPy Symbol

The tilt of the polarization relative to the `x` axis.

``**chi**`` : numeric type or SymPy Symbol

The angle adjacent to the mayor axis of the polarization ellipse.

``**p**`` : numeric type or SymPy Symbol

The degree of polarization.

``**I**`` : numeric type or SymPy Symbol

The intensity of the field.

Returns

Matrix :

A Stokes vector.

Examples

The axes on the Poincaré sphere.

```
>>> from sympy import pprint, symbols, pi
>>> from sympy.physics.optics.polarization import stokes_vector
>>> psi, chi, p, I = symbols("psi, chi, p, I", real=True)
>>> pprint(stokes_vector(psi, chi, p, I), use_unicode=True)
⎡          I          ⎤
⎢                     ⎥
⎢I·p·cos(2·χ)·cos(2·ψ)⎥
⎢                     ⎥
⎢I·p·sin(2·ψ)·cos(2·χ)⎥
⎢                     ⎥
⎣    I·p·sin(2·χ)     ⎦
```

Horizontal polarization >>> pprint(stokes_vector(0, 0), use_unicode=True) ⎡1⎤ ⎢ ⎥ ⎢1⎥ ⎢ ⎥ ⎢0⎥ ⎢ ⎥ ⎣0⎦

Vertical polarization >>> pprint(stokes_vector(pi/2, 0), use_unicode=True) ⎡ 1⎤ ⎢ ⎥ ⎢-1⎥ ⎢ ⎥ ⎢0 ⎥ ⎢ ⎥ ⎣0 ⎦

Diagonal polarization >>> pprint(stokes_vector(pi/4, 0), use_unicode=True) ⎡1⎤ ⎢ ⎥ ⎢0⎥ ⎢ ⎥ ⎢1⎥ ⎢ ⎥ ⎣0⎦

Anti-diagonal polarization >>> pprint(stokes_vector(-pi/4, 0), use_unicode=True) ⎡ 1⎤ ⎢ ⎥ ⎢0 ⎥ ⎢ ⎥ ⎢-1⎥ ⎢ ⎥ ⎣0 ⎦

Right-hand circular polarization >>> pprint(stokes_vector(0, pi/4), use_unicode=True) ⎡1⎤ ⎢ ⎥ ⎢0⎥ ⎢ ⎥ ⎢0⎥ ⎢ ⎥ ⎣1⎦

Left-hand circular polarization >>> pprint(stokes_vector(0, -pi/4), use_unicode=True) ⎡ 1⎤ ⎢ ⎥ ⎢0 ⎥ ⎢ ⎥ ⎢0 ⎥ ⎢ ⎥ ⎣-1⎦

Unpolarized light >>> pprint(stokes_vector(0, 0, 0), use_unicode=True) ⎡1⎤ ⎢ ⎥ ⎢0⎥ ⎢ ⎥ ⎢0⎥ ⎢ ⎥ ⎣0⎦

sympy.physics.optics.polarization.**transmissive_filter**(*T*)

An attenuator Jones matrix with transmittance T.

Parameters

``**T**`` : numeric type or SymPy Symbol

The transmittance of the attenuator.

Returns

SymPy Matrix

A Jones matrix representing the filter.

Examples

A generic filter.

```
>>> from sympy import pprint, symbols
>>> from sympy.physics.optics.polarization import transmissive_filter
>>> T = symbols("T", real=True)
>>> NDF = transmissive_filter(T)
>>> pprint(NDF, use_unicode=True)
⎡√T  0 ⎤
⎢      ⎥
⎣0   √T⎦
```

Utilities

Contains

- refraction_angle
- fresnel_coefficients
- deviation
- brewster_angle
- critical_angle
- lens_makers_formula
- mirror_formula
- lens_formula
- hyperfocal_distance
- transverse_magnification

sympy.physics.optics.utils.**brewster_angle**(*medium1*, *medium2*)

This function calculates the Brewster's angle of incidence to Medium 2 from Medium 1 in radians.

Parameters

medium 1 : Medium or sympifiable

Refractive index of Medium 1

medium 2 : Medium or sympifiable

Refractive index of Medium 1

Examples

```
>>> from sympy.physics.optics import brewster_angle
>>> brewster_angle(1, 1.33)
0.926093295503462
```

sympy.physics.optics.utils.**critical_angle**(*medium1*, *medium2*)

This function calculates the critical angle of incidence (marking the onset of total internal) to Medium 2 from Medium 1 in radians.

Parameters

medium 1 : Medium or sympifiable

Refractive index of Medium 1.

medium 2 : Medium or sympifiable

Refractive index of Medium 1.

Examples

```
>>> from sympy.physics.optics import critical_angle
>>> critical_angle(1.33, 1)
0.850908514477849
```

sympy.physics.optics.utils.**deviation**(*incident*, *medium1*, *medium2*, *normal=None*, *plane=None*)

This function calculates the angle of deviation of a ray due to refraction at planar surface.

Parameters

incident : Matrix, Ray3D, sequence or float

Incident vector or angle of incidence

medium1 : sympy.physics.optics.medium.Medium or sympifiable

Medium 1 or its refractive index

medium2 : sympy.physics.optics.medium.Medium or sympifiable

Medium 2 or its refractive index

normal : Matrix, Ray3D, or sequence

Normal vector

plane : Plane

Plane of separation of the two media.

Returns angular deviation between incident and refracted rays

Examples

```
>>> from sympy.physics.optics import deviation
>>> from sympy.geometry import Point3D, Ray3D, Plane
>>> from sympy.matrices import Matrix
>>> from sympy import symbols
>>> n1, n2 = symbols('n1, n2')
>>> n = Matrix([0, 0, 1])
>>> P = Plane(Point3D(0, 0, 0), normal_vector=[0, 0, 1])
>>> r1 = Ray3D(Point3D(-1, -1, 1), Point3D(0, 0, 0))
>>> deviation(r1, 1, 1, n)
0
>>> deviation(r1, n1, n2, plane=P)
-acos(-sqrt(-2*n1**2/(3*n2**2) + 1)) + acos(-sqrt(3)/3)
>>> round(deviation(0.1, 1.2, 1.5), 5)
-0.02005
```

sympy.physics.optics.utils.**fresnel_coefficients**(*angle_of_incidence*, *medium1*, *medium2*)

This function uses Fresnel equations to calculate reflection and transmission coefficients. Those are obtained for both polarisations when the electric field vector is in the plane of incidence (labelled 'p') and when the electric field vector is perpendicular to the plane of incidence (labelled 's'). There are four real coefficients unless the incident ray reflects in total internal in which case there are two complex ones. Angle of incidence is the angle between the incident ray and the surface normal. `medium1` and `medium2` can be `Medium` or any sympifiable object.

Parameters

angle_of_incidence : sympifiable

medium1 : Medium or sympifiable

Medium 1 or its refractive index

medium2 : Medium or sympifiable

Medium 2 or its refractive index

Returns

Returns a list with four real Fresnel coefficients:

[reflection p (TM), reflection s (TE),

transmission p (TM), transmission s (TE)]

If the ray is undergoes total internal reflection then returns a

list of two complex Fresnel coefficients:

[reflection p (TM), reflection s (TE)]

Examples

```
>>> from sympy.physics.optics import fresnel_coefficients
>>> fresnel_coefficients(0.3, 1, 2)
[0.317843553417859, -0.348645229818821,
        0.658921776708929, 0.651354770181179]
>>> fresnel_coefficients(0.6, 2, 1)
[-0.235625382192159 - 0.971843958291041*I,
         0.816477005968898 - 0.577377951366403*I]
```

References

[R667]

sympy.physics.optics.utils.**hyperfocal_distance**(*f*, *N*, *c*)

Parameters

f: sympifiable

Focal length of a given lens.

N: sympifiable

F-number of a given lens.

c: sympifiable

Circle of Confusion (CoC) of a given image format.

Example

```
>>> from sympy.physics.optics import hyperfocal_distance
>>> round(hyperfocal_distance(f = 0.5, N = 8, c = 0.0033), 2)
9.47
```

sympy.physics.optics.utils.**lens_formula**(*focal_length=None*, *u=None*, *v=None*)

This function provides one of the three parameters when two of them are supplied. This is valid only for paraxial rays.

Parameters

focal_length : sympifiable

Focal length of the mirror.

u : sympifiable

Distance of object from the optical center on the principal axis.

v : sympifiable

Distance of the image from the optical center on the principal axis.

Examples

```
>>> from sympy.physics.optics import lens_formula
>>> from sympy.abc import f, u, v
>>> lens_formula(focal_length=f, u=u)
f*u/(f + u)
>>> lens_formula(focal_length=f, v=v)
f*v/(f - v)
>>> lens_formula(u=u, v=v)
u*v/(u - v)
```

sympy.physics.optics.utils.**lens_makers_formula**(*n_lens*, *n_surr*, *r1*, *r2*, *d=0*)

This function calculates focal length of a lens. It follows cartesian sign convention.

Parameters

n_lens : Medium or sympifiable

Index of refraction of lens.

n_surr : Medium or sympifiable

Index of reflection of surrounding.

r1 : sympifiable

Radius of curvature of first surface.

r2 : sympifiable

Radius of curvature of second surface.

d : sympifiable, optional

Thickness of lens, default value is 0.

Examples

```
>>> from sympy.physics.optics import lens_makers_formula
>>> from sympy import S
>>> lens_makers_formula(1.33, 1, 10, -10)
15.1515151515151
>>> lens_makers_formula(1.2, 1, 10, S.Infinity)
50.0000000000000
>>> lens_makers_formula(1.33, 1, 10, -10, d=1)
15.3418463277618
```

sympy.physics.optics.utils.**mirror_formula**(*focal_length=None*, *u=None*, *v=None*)

This function provides one of the three parameters when two of them are supplied. This is valid only for paraxial rays.

Parameters

focal_length : sympifiable

Focal length of the mirror.

u : sympifiable

Distance of object from the pole on the principal axis.

v : sympifiable

Distance of the image from the pole on the principal axis.

Examples

```
>>> from sympy.physics.optics import mirror_formula
>>> from sympy.abc import f, u, v
>>> mirror_formula(focal_length=f, u=u)
f*u/(-f + u)
>>> mirror_formula(focal_length=f, v=v)
f*v/(-f + v)
>>> mirror_formula(u=u, v=v)
u*v/(u + v)
```

sympy.physics.optics.utils.**refraction_angle**(*incident*, *medium1*, *medium2*, *normal=None*, *plane=None*)

This function calculates transmitted vector after refraction at planar surface. `medium1` and `medium2` can be `Medium` or any sympifiable object. If `incident` is a number then treated as angle of incidence (in radians) in which case refraction angle is returned.

If `incident` is an object of $Ray3D$, $normal$ also has to be an instance of $Ray3D$ in order to get the output as a $Ray3D$. Please note that if plane of separation is not provided and normal is an instance of $Ray3D$, `normal` will be assumed to be intersecting incident ray at the plane of separation. This will not be the case when $normal$ is a $Matrix$ or any other sequence. If `incident` is an instance of $Ray3D$ and $plane$ has not been provided and `normal` is not $Ray3D$, output will be a $Matrix$.

Parameters

incident : Matrix, Ray3D, sequence or a number

Incident vector or angle of incidence

medium1 : sympy.physics.optics.medium.Medium or sympifiable

Medium 1 or its refractive index

medium2 : sympy.physics.optics.medium.Medium or sympifiable

Medium 2 or its refractive index

normal : Matrix, Ray3D, or sequence

Normal vector

plane : Plane

Plane of separation of the two media.

Returns

Returns an angle of refraction or a refracted ray depending on inputs.

Examples

```
>>> from sympy.physics.optics import refraction_angle
>>> from sympy.geometry import Point3D, Ray3D, Plane
>>> from sympy.matrices import Matrix
>>> from sympy import symbols, pi
>>> n = Matrix([0, 0, 1])
>>> P = Plane(Point3D(0, 0, 0), normal_vector=[0, 0, 1])
>>> r1 = Ray3D(Point3D(-1, -1, 1), Point3D(0, 0, 0))
>>> refraction_angle(r1, 1, 1, n)
Matrix([
[ 1],
[ 1],
[-1]])
>>> refraction_angle(r1, 1, 1, plane=P)
Ray3D(Point3D(0, 0, 0), Point3D(1, 1, -1))
```

With different index of refraction of the two media

```
>>> n1, n2 = symbols('n1, n2')
>>> refraction_angle(r1, n1, n2, n)
Matrix([
[                                n1/n2],
[                                n1/n2],
[-sqrt(3)*sqrt(-2*n1**2/(3*n2**2) + 1)]])
>>> refraction_angle(r1, n1, n2, plane=P)
Ray3D(Point3D(0, 0, 0), Point3D(n1/n2, n1/n2, -sqrt(3)*sqrt(-2*n1**2/
↪(3*n2**2) + 1)))
>>> round(refraction_angle(pi/6, 1.2, 1.5), 5)
0.41152
```

sympy.physics.optics.utils.**transverse_magnification**(*si*, *so*)

Calculates the transverse magnification, which is the ratio of the image size to the object size.

Parameters

so: sympifiable

Lens-object distance.

si: sympifiable

Lens-image distance.

Example

```
>>> from sympy.physics.optics import transverse_magnification
>>> transverse_magnification(30, 15)
-2
```

Waves

This module has all the classes and functions related to waves in optics.

Contains

- TWave

class sympy.physics.optics.waves.**TWave**(*amplitude*, *frequency=None*, *phase=0*, *time_period=None*, *n=n*)

This is a simple transverse sine wave travelling in a one-dimensional space. Basic properties are required at the time of creation of the object, but they can be changed later with respective methods provided.

Raises

ValueError : When neither frequency nor time period is provided

or they are not consistent.

TypeError : When anything other than TWave objects is added.

Explanation

It is represented as $A \times cos(k * x - \omega \times t + \phi)$, where A is the amplitude, ω is the angular frequency, k is the wavenumber (spatial frequency), x is a spatial variable to represent the position on the dimension on which the wave propagates, and ϕ is the phase angle of the wave.

Arguments

amplitude
[Sympifyable] Amplitude of the wave.

frequency
[Sympifyable] Frequency of the wave.

phase
[Sympifyable] Phase angle of the wave.

time_period
[Sympifyable] Time period of the wave.

n
[Sympifyable] Refractive index of the medium.

Examples

```
>>> from sympy import symbols
>>> from sympy.physics.optics import TWave
>>> A1, phi1, A2, phi2, f = symbols('A1, phi1, A2, phi2, f')
>>> w1 = TWave(A1, f, phi1)
>>> w2 = TWave(A2, f, phi2)
>>> w3 = w1 + w2  # Superposition of two waves
>>> w3
```

(continues on next page)

(continued from previous page)

```
TWave(sqrt(A1**2 + 2*A1*A2*cos(phi1 - phi2) + A2**2), f,
    atan2(A1*sin(phi1) + A2*sin(phi2), A1*cos(phi1) + A2*cos(phi2)), 1/f,
↪ n)
>>> w3.amplitude
sqrt(A1**2 + 2*A1*A2*cos(phi1 - phi2) + A2**2)
>>> w3.phase
atan2(A1*sin(phi1) + A2*sin(phi2), A1*cos(phi1) + A2*cos(phi2))
>>> w3.speed
299792458*meter/(second*n)
>>> w3.angular_velocity
2*pi*f
```

property amplitude

Returns the amplitude of the wave.

Examples

```
>>> from sympy import symbols
>>> from sympy.physics.optics import TWave
>>> A, phi, f = symbols('A, phi, f')
>>> w = TWave(A, f, phi)
>>> w.amplitude
A
```

property angular_velocity

Returns the angular velocity of the wave, in radians per second.

Examples

```
>>> from sympy import symbols
>>> from sympy.physics.optics import TWave
>>> A, phi, f = symbols('A, phi, f')
>>> w = TWave(A, f, phi)
>>> w.angular_velocity
2*pi*f
```

property frequency

Returns the frequency of the wave, in cycles per second.

Examples

```
>>> from sympy import symbols
>>> from sympy.physics.optics import TWave
>>> A, phi, f = symbols('A, phi, f')
>>> w = TWave(A, f, phi)
>>> w.frequency
f
```

property n

Returns the refractive index of the medium

property phase

Returns the phase angle of the wave, in radians.

Examples

```
>>> from sympy import symbols
>>> from sympy.physics.optics import TWave
>>> A, phi, f = symbols('A, phi, f')
>>> w = TWave(A, f, phi)
>>> w.phase
phi
```

property speed

Returns the propagation speed of the wave, in meters per second. It is dependent on the propagation medium.

Examples

```
>>> from sympy import symbols
>>> from sympy.physics.optics import TWave
>>> A, phi, f = symbols('A, phi, f')
>>> w = TWave(A, f, phi)
>>> w.speed
299792458*meter/(second*n)
```

property time_period

Returns the temporal period of the wave, in seconds per cycle.

Examples

```
>>> from sympy import symbols
>>> from sympy.physics.optics import TWave
>>> A, phi, f = symbols('A, phi, f')
>>> w = TWave(A, f, phi)
>>> w.time_period
1/f
```

property wavelength

Returns the wavelength (spatial period) of the wave, in meters per cycle. It depends on the medium of the wave.

Examples

```
>>> from sympy import symbols
>>> from sympy.physics.optics import TWave
>>> A, phi, f = symbols('A, phi, f')
>>> w = TWave(A, f, phi)
>>> w.wavelength
299792458*meter/(second*f*n)
```

property wavenumber

Returns the wavenumber of the wave, in radians per meter.

Examples

```
>>> from sympy import symbols
>>> from sympy.physics.optics import TWave
>>> A, phi, f = symbols('A, phi, f')
>>> w = TWave(A, f, phi)
>>> w.wavenumber
pi*second*f*n/(149896229*meter)
```


Control Module

Abstract

Contains docstrings of Physics-Control module

Control

Currently, *sympy.physics.control* (page 1890) is able to deal with LTI (Linear, time-invariant) systems. The `TransferFunction` class is used to represent Continuous-time Transfer functions in the Laplace domain; where Transfer functions are input to output representations of dynamic systems. The additive property is used for transfer functions in the `Parallel` class, and the multiplicative property is used for transfer functions in the `Series` class. Also, there is a `Feedback` class which is used to represent negative feedback interconnection between two input/output systems. MIMO systems are also supported with `TransferFunctionMatrix` as the base class for representing one. `MIMOSeries`, `MIMOParallel` and `MIMOFeedback` are MIMO equivalent of `Series`, `Parallel` and `Feedback` classes.

The advantage of this symbolic Control system package is that the solutions obtained from it are highly accurate and do not rely on numerical methods to approximate the solutions. Symbolic solutions obtained are also in a compact form that can be used for further analysis.

Control API

lti

class sympy.physics.control.lti.**TransferFunction**(*num*, *den*, *var*)

A class for representing LTI (Linear, time-invariant) systems that can be strictly described by ratio of polynomials in the Laplace transform complex variable. The arguments are `num`, `den`, and `var`, where `num` and `den` are numerator and denominator polynomials of the `TransferFunction` respectively, and the third argument is a complex variable of the Laplace transform used by these polynomials of the transfer function. `num` and `den` can be either polynomials or numbers, whereas `var` has to be a *Symbol* (page 976).

Parameters

num : Expr, Number

The numerator polynomial of the transfer function.

den : Expr, Number

The denominator polynomial of the transfer function.

var : Symbol

Complex variable of the Laplace transform used by the polynomials of the transfer function.

Raises

TypeError

When `var` is not a Symbol or when `num` or `den` is not a number or a polynomial.

ValueError

When `den` is zero.

Explanation

Generally, a dynamical system representing a physical model can be described in terms of Linear Ordinary Differential Equations like -

$$b_m y^{(m)} + b_{m-1} y^{(m-1)} + \cdots + b_1 y^{(1)} + b_0 y = a_n x^{(n)} + a_{n-1} x^{(n-1)} + \cdots + a_1 x^{(1)} + a_0 x$$

Here, x is the input signal and y is the output signal and superscript on both is the order of derivative (not exponent). Derivative is taken with respect to the independent variable, t. Also, generally m is greater than n.

It is not feasible to analyse the properties of such systems in their native form therefore, we use mathematical tools like Laplace transform to get a better perspective. Taking the Laplace transform of both the sides in the equation (at zero initial conditions), we get -

$$\mathcal{L}[b_m y^{(m)} + b_{m-1} y^{(m-1)} + \cdots + b_1 y^{(1)} + b_0 y] = \mathcal{L}[a_n x^{(n)} + a_{n-1} x^{(n-1)} + \cdots + a_1 x^{(1)} + a_0 x]$$

Using the linearity property of Laplace transform and also considering zero initial conditions (i.e. $y(0^-) = 0$, $y'(0^-) = 0$ and so on), the equation above gets translated to -

$$b_m \mathcal{L}[y^{(m)}] + \cdots + b_1 \mathcal{L}[y^{(1)}] + b_0 \mathcal{L}[y] = a_n \mathcal{L}[x^{(n)}] + \cdots + a_1 \mathcal{L}[x^{(1)}] + a_0 \mathcal{L}[x]$$

Now, applying Derivative property of Laplace transform,

$$b_m s^m \mathcal{L}[y] + \cdots + b_1 s\mathcal{L}[y] + b_0 \mathcal{L}[y] = a_n s^n \mathcal{L}[x] + \cdots + a_1 s\mathcal{L}[x] + a_0 \mathcal{L}[x]$$

Here, the superscript on s is **exponent**. Note that the zero initial conditions assumption, mentioned above, is very important and cannot be ignored otherwise the dynamical system cannot be considered time-independent and the simplified equation above cannot be reached.

Collecting $\mathcal{L}[y]$ and $\mathcal{L}[x]$ terms from both the sides and taking the ratio $\frac{\mathcal{L}\{y\}}{\mathcal{L}\{x\}}$, we get the typical rational form of transfer function.

The numerator of the transfer function is, therefore, the Laplace transform of the output signal (The signals are represented as functions of time) and similarly, the denominator of the transfer function is the Laplace transform of the input signal. It is also a convention to denote the input and output signal's Laplace transform with capital alphabets like shown below.

$$H(s) = \frac{Y(s)}{X(s)} = \frac{\mathcal{L}\{y(t)\}}{\mathcal{L}\{x(t)\}}$$

s, also known as complex frequency, is a complex variable in the Laplace domain. It corresponds to the equivalent variable t, in the time domain. Transfer functions are sometimes also referred to as the Laplace transform of the system's impulse response. Transfer function, H, is represented as a rational function in s like,

$$H(s) = \frac{a_n s^n + a_{n-1} s^{n-1} + \cdots + a_1 s + a_0}{b_m s^m + b_{m-1} s^{m-1} + \cdots + b_1 s + b_0}$$

Examples

```
>>> from sympy.abc import s, p, a
>>> from sympy.physics.control.lti import TransferFunction
>>> tf1 = TransferFunction(s + a, s**2 + s + 1, s)
>>> tf1
TransferFunction(a + s, s**2 + s + 1, s)
>>> tf1.num
a + s
>>> tf1.den
s**2 + s + 1
>>> tf1.var
s
>>> tf1.args
(a + s, s**2 + s + 1, s)
```

Any complex variable can be used for `var`.

```
>>> tf2 = TransferFunction(a*p**3 - a*p**2 + s*p, p + a**2, p)
>>> tf2
TransferFunction(a*p**3 - a*p**2 + p*s, a**2 + p, p)
>>> tf3 = TransferFunction((p + 3)*(p - 1), (p - 1)*(p + 5), p)
>>> tf3
TransferFunction((p - 1)*(p + 3), (p - 1)*(p + 5), p)
```

To negate a transfer function the `-` operator can be prepended:

```
>>> tf4 = TransferFunction(-a + s, p**2 + s, p)
>>> -tf4
```

(continues on next page)

(continued from previous page)

```
TransferFunction(a - s, p**2 + s, p)
>>> tf5 = TransferFunction(s**4 - 2*s**3 + 5*s + 4, s + 4, s)
>>> -tf5
TransferFunction(-s**4 + 2*s**3 - 5*s - 4, s + 4, s)
```

You can use a float or an integer (or other constants) as numerator and denominator:

```
>>> tf6 = TransferFunction(1/2, 4, s)
>>> tf6.num
0.500000000000000
>>> tf6.den
4
>>> tf6.var
s
>>> tf6.args
(0.5, 4, s)
```

You can take the integer power of a transfer function using the ** operator:

```
>>> tf7 = TransferFunction(s + a, s - a, s)
>>> tf7**3
TransferFunction((a + s)**3, (-a + s)**3, s)
>>> tf7**0
TransferFunction(1, 1, s)
>>> tf8 = TransferFunction(p + 4, p - 3, p)
>>> tf8**-1
TransferFunction(p - 3, p + 4, p)
```

Addition, subtraction, and multiplication of transfer functions can form unevaluated `Series` or `Parallel` objects.

```
>>> tf9 = TransferFunction(s + 1, s**2 + s + 1, s)
>>> tf10 = TransferFunction(s - p, s + 3, s)
>>> tf11 = TransferFunction(4*s**2 + 2*s - 4, s - 1, s)
>>> tf12 = TransferFunction(1 - s, s**2 + 4, s)
>>> tf9 + tf10
Parallel(TransferFunction(s + 1, s**2 + s + 1, s), TransferFunction(-p +␣
↪s, s + 3, s))
>>> tf10 - tf11
Parallel(TransferFunction(-p + s, s + 3, s), TransferFunction(-4*s**2 -␣
↪2*s + 4, s - 1, s))
>>> tf9 * tf10
Series(TransferFunction(s + 1, s**2 + s + 1, s), TransferFunction(-p + s,
↪ s + 3, s))
>>> tf10 - (tf9 + tf12)
Parallel(TransferFunction(-p + s, s + 3, s), TransferFunction(-s - 1,␣
↪s**2 + s + 1, s), TransferFunction(s - 1, s**2 + 4, s))
>>> tf10 - (tf9 * tf12)
Parallel(TransferFunction(-p + s, s + 3, s), Series(TransferFunction(-1,␣
↪1, s), TransferFunction(s + 1, s**2 + s + 1, s), TransferFunction(1 -␣
↪s, s**2 + 4, s)))
>>> tf11 * tf10 * tf9
```

(continues on next page)

(continued from previous page)

```
Series(TransferFunction(4*s**2 + 2*s - 4, s - 1, s), TransferFunction(-p␣
↪+ s, s + 3, s), TransferFunction(s + 1, s**2 + s + 1, s))
>>> tf9 * tf11 + tf10 * tf12
Parallel(Series(TransferFunction(s + 1, s**2 + s + 1, s),␣
↪TransferFunction(4*s**2 + 2*s - 4, s - 1, s)),␣
↪Series(TransferFunction(-p + s, s + 3, s), TransferFunction(1 - s,␣
↪s**2 + 4, s)))
>>> (tf9 + tf12) * (tf10 + tf11)
Series(Parallel(TransferFunction(s + 1, s**2 + s + 1, s),␣
↪TransferFunction(1 - s, s**2 + 4, s)), Parallel(TransferFunction(-p +␣
↪s, s + 3, s), TransferFunction(4*s**2 + 2*s - 4, s - 1, s)))
```

These unevaluated `Series` or `Parallel` objects can convert into the resultant transfer function using `.doit()` method or by `.rewrite(TransferFunction)`.

```
>>> ((tf9 + tf10) * tf12).doit()
TransferFunction((1 - s)*((-p + s)*(s**2 + s + 1) + (s + 1)*(s + 3)), (s␣
↪+ 3)*(s**2 + 4)*(s**2 + s + 1), s)
>>> (tf9 * tf10 - tf11 * tf12).rewrite(TransferFunction)
TransferFunction(-(1 - s)*(s + 3)*(s**2 + s + 1)*(4*s**2 + 2*s - 4) + (-
↪p + s)*(s - 1)*(s + 1)*(s**2 + 4), (s - 1)*(s + 3)*(s**2 + 4)*(s**2 +␣
↪s + 1), s)
```

See also:

Feedback (page 1906), *Series* (page 1899), *Parallel* (page 1903)

References

[R656], [R657]

dc_gain()

Computes the gain of the response as the frequency approaches zero.

The DC gain is infinite for systems with pure integrators.

Examples

```
>>> from sympy.abc import s, p, a, b
>>> from sympy.physics.control.lti import TransferFunction
>>> tf1 = TransferFunction(s + 3, s**2 - 9, s)
>>> tf1.dc_gain()
-1/3
>>> tf2 = TransferFunction(p**2, p - 3 + p**3, p)
>>> tf2.dc_gain()
0
>>> tf3 = TransferFunction(a*p**2 - b, s + b, s)
>>> tf3.dc_gain()
(a*p**2 - b)/b
>>> tf4 = TransferFunction(1, s, s)
```

(continues on next page)

(continued from previous page)

```
>>> tf4.dc_gain()
oo
```

property den

Returns the denominator polynomial of the transfer function.

Examples

```
>>> from sympy.abc import s, p
>>> from sympy.physics.control.lti import TransferFunction
>>> G1 = TransferFunction(s + 4, p**3 - 2*p + 4, s)
>>> G1.den
p**3 - 2*p + 4
>>> G2 = TransferFunction(3, 4, s)
>>> G2.den
4
```

expand()

Returns the transfer function with numerator and denominator in expanded form.

Examples

```
>>> from sympy.abc import s, p, a, b
>>> from sympy.physics.control.lti import TransferFunction
>>> G1 = TransferFunction((a - s)**2, (s**2 + a)**2, s)
>>> G1.expand()
TransferFunction(a**2 - 2*a*s + s**2, a**2 + 2*a*s**2 + s**4, s)
>>> G2 = TransferFunction((p + 3*b)*(p - b), (p - b)*(p + 2*b), p)
>>> G2.expand()
TransferFunction(-3*b**2 + 2*b*p + p**2, -2*b**2 + b*p + p**2, p)
```

classmethod from_rational_expression(*expr*, *var=None*)

Creates a new `TransferFunction` efficiently from a rational expression.

Parameters

expr : Expr, Number

The rational expression representing the `TransferFunction`.

var : Symbol, optional

Complex variable of the Laplace transform used by the polynomials of the transfer function.

Raises

ValueError

When `expr` is of type `Number` and optional parameter `var` is not passed.

When `expr` has more than one variables and an optional parameter `var` is not passed.

ZeroDivisionError

When denominator of `expr` is zero or it has `ComplexInfinity` in its numerator.

Examples

```
>>> from sympy.abc import s, p, a
>>> from sympy.physics.control.lti import TransferFunction
>>> expr1 = (s + 5)/(3*s**2 + 2*s + 1)
>>> tf1 = TransferFunction.from_rational_expression(expr1)
>>> tf1
TransferFunction(s + 5, 3*s**2 + 2*s + 1, s)
>>> expr2 = (a*p**3 - a*p**2 + s*p)/(p + a**2)  # Expr with more than␣
↪one variables
>>> tf2 = TransferFunction.from_rational_expression(expr2, p)
>>> tf2
TransferFunction(a*p**3 - a*p**2 + p*s, a**2 + p, p)
```

In case of conflict between two or more variables in a expression, SymPy will raise a `ValueError`, if `var` is not passed by the user.

```
>>> tf = TransferFunction.from_rational_expression((a + a*s)/(s**2 +␣
↪s + 1))
Traceback (most recent call last):
...
ValueError: Conflicting values found for positional argument `var` (
↪{a, s}). Specify it manually.
```

This can be corrected by specifying the `var` parameter manually.

```
>>> tf = TransferFunction.from_rational_expression((a + a*s)/(s**2 +␣
↪s + 1), s)
>>> tf
TransferFunction(a*s + a, s**2 + s + 1, s)
```

`var` also need to be specified when `expr` is a `Number`

```
>>> tf3 = TransferFunction.from_rational_expression(10, s)
>>> tf3
TransferFunction(10, 1, s)
```

property is_biproper

Returns True if degree of the numerator polynomial is equal to degree of the denominator polynomial, else False.

Examples

```
>>> from sympy.abc import s, p, a, b
>>> from sympy.physics.control.lti import TransferFunction
>>> tf1 = TransferFunction(a*p**2 + b*s, s - p, s)
>>> tf1.is_biproper
True
>>> tf2 = TransferFunction(p**2, p + a, p)
>>> tf2.is_biproper
False
```

property is_proper

Returns True if degree of the numerator polynomial is less than or equal to degree of the denominator polynomial, else False.

Examples

```
>>> from sympy.abc import s, p, a, b
>>> from sympy.physics.control.lti import TransferFunction
>>> tf1 = TransferFunction(b*s**2 + p**2 - a*p + s, b - p**2, s)
>>> tf1.is_proper
False
>>> tf2 = TransferFunction(p**2 - 4*p, p**3 + 3*p + 2, p)
>>> tf2.is_proper
True
```

is_stable()

Returns True if the transfer function is asymptotically stable; else False.

This would not check the marginal or conditional stability of the system.

Examples

```
>>> from sympy.abc import s, p, a
>>> from sympy import symbols
>>> from sympy.physics.control.lti import TransferFunction
>>> q, r = symbols('q, r', negative=True)
>>> tf1 = TransferFunction((1 - s)**2, (s + 1)**2, s)
>>> tf1.is_stable()
True
>>> tf2 = TransferFunction((1 - p)**2, (s**2 + 1)**2, s)
>>> tf2.is_stable()
False
>>> tf3 = TransferFunction(4, q*s - r, s)
>>> tf3.is_stable()
False
>>> tf4 = TransferFunction(p + 1, a*p - s**2, p)
>>> tf4.is_stable() is None   # Not enough info about the symbols to␣
↪determine stability
True
```

property is_strictly_proper

Returns True if degree of the numerator polynomial is strictly less than degree of the denominator polynomial, else False.

Examples

```
>>> from sympy.abc import s, p, a, b
>>> from sympy.physics.control.lti import TransferFunction
>>> tf1 = TransferFunction(a*p**2 + b*s, s - p, s)
>>> tf1.is_strictly_proper
False
>>> tf2 = TransferFunction(s**3 - 2, s**4 + 5*s + 6, s)
>>> tf2.is_strictly_proper
True
```

property num

Returns the numerator polynomial of the transfer function.

Examples

```
>>> from sympy.abc import s, p
>>> from sympy.physics.control.lti import TransferFunction
>>> G1 = TransferFunction(s**2 + p*s + 3, s - 4, s)
>>> G1.num
p*s + s**2 + 3
>>> G2 = TransferFunction((p + 5)*(p - 3), (p - 3)*(p + 1), p)
>>> G2.num
(p - 3)*(p + 5)
```

poles()

Returns the poles of a transfer function.

Examples

```
>>> from sympy.abc import s, p, a
>>> from sympy.physics.control.lti import TransferFunction
>>> tf1 = TransferFunction((p + 3)*(p - 1), (p - 1)*(p + 5), p)
>>> tf1.poles()
[-5, 1]
>>> tf2 = TransferFunction((1 - s)**2, (s**2 + 1)**2, s)
>>> tf2.poles()
[I, I, -I, -I]
>>> tf3 = TransferFunction(s**2, a*s + p, s)
>>> tf3.poles()
[-p/a]
```

to_expr()

Converts a TransferFunction object to SymPy Expr.

Examples

```
>>> from sympy.abc import s, p, a, b
>>> from sympy.physics.control.lti import TransferFunction
>>> from sympy import Expr
>>> tf1 = TransferFunction(s, a*s**2 + 1, s)
>>> tf1.to_expr()
s/(a*s**2 + 1)
>>> isinstance(_, Expr)
True
>>> tf2 = TransferFunction(1, (p + 3*b)*(b - p), p)
>>> tf2.to_expr()
1/((b - p)*(3*b + p))
>>> tf3 = TransferFunction((s - 2)*(s - 3), (s - 1)*(s - 2)*(s - 3),␣
↪s)
>>> tf3.to_expr()
((s - 3)*(s - 2))/(((s - 3)*(s - 2)*(s - 1)))
```

property var

Returns the complex variable of the Laplace transform used by the polynomials of the transfer function.

Examples

```
>>> from sympy.abc import s, p
>>> from sympy.physics.control.lti import TransferFunction
>>> G1 = TransferFunction(p**2 + 2*p + 4, p - 6, p)
>>> G1.var
p
>>> G2 = TransferFunction(0, s - 5, s)
>>> G2.var
s
```

zeros()

Returns the zeros of a transfer function.

Examples

```
>>> from sympy.abc import s, p, a
>>> from sympy.physics.control.lti import TransferFunction
>>> tf1 = TransferFunction((p + 3)*(p - 1), (p - 1)*(p + 5), p)
>>> tf1.zeros()
[-3, 1]
>>> tf2 = TransferFunction((1 - s)**2, (s**2 + 1)**2, s)
>>> tf2.zeros()
[1, 1]
>>> tf3 = TransferFunction(s**2, a*s + p, s)
>>> tf3.zeros()
[0, 0]
```

class sympy.physics.control.lti.**Series**(**args*, *evaluate=False*)

A class for representing a series configuration of SISO systems.

Parameters

args : SISOLinearTimeInvariant

SISO systems in a series configuration.

evaluate : Boolean, Keyword

When passed `True`, returns the equivalent `Series(*args).doit()`. Set to `False` by default.

Raises

ValueError

When no argument is passed.

`var` attribute is not same for every system.

TypeError

Any of the passed `*args` has unsupported type

A combination of SISO and MIMO systems is passed. There should be homogeneity in the type of systems passed, SISO in this case.

Examples

```
>>> from sympy.abc import s, p, a, b
>>> from sympy.physics.control.lti import TransferFunction, Series,␣
↪Parallel
>>> tf1 = TransferFunction(a*p**2 + b*s, s - p, s)
>>> tf2 = TransferFunction(s**3 - 2, s**4 + 5*s + 6, s)
>>> tf3 = TransferFunction(p**2, p + s, s)
>>> S1 = Series(tf1, tf2)
>>> S1
Series(TransferFunction(a*p**2 + b*s, -p + s, s), TransferFunction(s**3 -
↪ 2, s**4 + 5*s + 6, s))
>>> S1.var
s
>>> S2 = Series(tf2, Parallel(tf3, -tf1))
>>> S2
Series(TransferFunction(s**3 - 2, s**4 + 5*s + 6, s),␣
↪Parallel(TransferFunction(p**2, p + s, s), TransferFunction(-a*p**2 -␣
↪b*s, -p + s, s)))
>>> S2.var
s
>>> S3 = Series(Parallel(tf1, tf2), Parallel(tf2, tf3))
>>> S3
Series(Parallel(TransferFunction(a*p**2 + b*s, -p + s, s),␣
↪TransferFunction(s**3 - 2, s**4 + 5*s + 6, s)),␣
↪Parallel(TransferFunction(s**3 - 2, s**4 + 5*s + 6, s),␣
↪TransferFunction(p**2, p + s, s)))
>>> S3.var
s
```

You can get the resultant transfer function by using `.doit()` method:

```
>>> S3 = Series(tf1, tf2, -tf3)
>>> S3.doit()
TransferFunction(-p**2*(s**3 - 2)*(a*p**2 + b*s), (-p + s)*(p + s)*(s**4␣
↪+ 5*s + 6), s)
>>> S4 = Series(tf2, Parallel(tf1, -tf3))
>>> S4.doit()
TransferFunction((s**3 - 2)*(-p**2*(-p + s) + (p + s)*(a*p**2 + b*s)), (-
↪p + s)*(p + s)*(s**4 + 5*s + 6), s)
```

Notes

All the transfer functions should use the same complex variable `var` of the Laplace transform.

See also:

MIMOSeries (page 1922), *Parallel* (page 1903), *TransferFunction* (page 1891), *Feedback* (page 1906)

doit(***hints*)

Returns the resultant transfer function obtained after evaluating the transfer functions in series configuration.

Examples

```
>>> from sympy.abc import s, p, a, b
>>> from sympy.physics.control.lti import TransferFunction, Series
>>> tf1 = TransferFunction(a*p**2 + b*s, s - p, s)
>>> tf2 = TransferFunction(s**3 - 2, s**4 + 5*s + 6, s)
>>> Series(tf2, tf1).doit()
TransferFunction((s**3 - 2)*(a*p**2 + b*s), (-p + s)*(s**4 + 5*s + 6),
↪ s)
>>> Series(-tf1, -tf2).doit()
TransferFunction((2 - s**3)*(-a*p**2 - b*s), (-p + s)*(s**4 + 5*s +␣
↪6), s)
```

property is_biproper

Returns True if degree of the numerator polynomial of the resultant transfer function is equal to degree of the denominator polynomial of the same, else False.

Examples

```
>>> from sympy.abc import s, p, a, b
>>> from sympy.physics.control.lti import TransferFunction, Series
>>> tf1 = TransferFunction(a*p**2 + b*s, s - p, s)
>>> tf2 = TransferFunction(p, s**2, s)
>>> tf3 = TransferFunction(s**2, 1, s)
>>> S1 = Series(tf1, -tf2)
>>> S1.is_biproper
False
>>> S2 = Series(tf2, tf3)
>>> S2.is_biproper
True
```

property is_proper

Returns True if degree of the numerator polynomial of the resultant transfer function is less than or equal to degree of the denominator polynomial of the same, else False.

Examples

```
>>> from sympy.abc import s, p, a, b
>>> from sympy.physics.control.lti import TransferFunction, Series
>>> tf1 = TransferFunction(b*s**2 + p**2 - a*p + s, b - p**2, s)
>>> tf2 = TransferFunction(p**2 - 4*p, p**3 + 3*s + 2, s)
>>> tf3 = TransferFunction(s, s**2 + s + 1, s)
>>> S1 = Series(-tf2, tf1)
>>> S1.is_proper
False
>>> S2 = Series(tf1, tf2, tf3)
>>> S2.is_proper
True
```

property is_strictly_proper

Returns True if degree of the numerator polynomial of the resultant transfer function is strictly less than degree of the denominator polynomial of the same, else False.

Examples

```
>>> from sympy.abc import s, p, a, b
>>> from sympy.physics.control.lti import TransferFunction, Series
>>> tf1 = TransferFunction(a*p**2 + b*s, s - p, s)
>>> tf2 = TransferFunction(s**3 - 2, s**2 + 5*s + 6, s)
>>> tf3 = TransferFunction(1, s**2 + s + 1, s)
>>> S1 = Series(tf1, tf2)
>>> S1.is_strictly_proper
False
>>> S2 = Series(tf1, tf2, tf3)
>>> S2.is_strictly_proper
True
```

to_expr()

Returns the equivalent `Expr` object.

property var

Returns the complex variable used by all the transfer functions.

Examples

```
>>> from sympy.abc import p
>>> from sympy.physics.control.lti import TransferFunction, Series,␣
↪Parallel
>>> G1 = TransferFunction(p**2 + 2*p + 4, p - 6, p)
>>> G2 = TransferFunction(p, 4 - p, p)
>>> G3 = TransferFunction(0, p**4 - 1, p)
>>> Series(G1, G2).var
p
>>> Series(-G3, Parallel(G1, G2)).var
p
```

class sympy.physics.control.lti.**Parallel**(**args*, *evaluate=False*)

A class for representing a parallel configuration of SISO systems.

Parameters

args : SISOLinearTimeInvariant

SISO systems in a parallel arrangement.

evaluate : Boolean, Keyword

When passed `True`, returns the equivalent `Parallel(*args).doit()`. Set to `False` by default.

Raises

ValueError

When no argument is passed.

`var` attribute is not same for every system.

TypeError

Any of the passed `*args` has unsupported type

A combination of SISO and MIMO systems is passed. There should be homogeneity in the type of systems passed.

Examples

```
>>> from sympy.abc import s, p, a, b
>>> from sympy.physics.control.lti import TransferFunction, Parallel,␣
↪Series
>>> tf1 = TransferFunction(a*p**2 + b*s, s - p, s)
>>> tf2 = TransferFunction(s**3 - 2, s**4 + 5*s + 6, s)
>>> tf3 = TransferFunction(p**2, p + s, s)
>>> P1 = Parallel(tf1, tf2)
```

(continues on next page)

(continued from previous page)

```
>>> P1
Parallel(TransferFunction(a*p**2 + b*s, -p + s, s),␣
↪TransferFunction(s**3 - 2, s**4 + 5*s + 6, s))
>>> P1.var
s
>>> P2 = Parallel(tf2, Series(tf3, -tf1))
>>> P2
Parallel(TransferFunction(s**3 - 2, s**4 + 5*s + 6, s),␣
↪Series(TransferFunction(p**2, p + s, s), TransferFunction(-a*p**2 -␣
↪b*s, -p + s, s)))
>>> P2.var
s
>>> P3 = Parallel(Series(tf1, tf2), Series(tf2, tf3))
>>> P3
Parallel(Series(TransferFunction(a*p**2 + b*s, -p + s, s),␣
↪TransferFunction(s**3 - 2, s**4 + 5*s + 6, s)),␣
↪Series(TransferFunction(s**3 - 2, s**4 + 5*s + 6, s),␣
↪TransferFunction(p**2, p + s, s)))
>>> P3.var
s
```

You can get the resultant transfer function by using `.doit()` method:

```
>>> Parallel(tf1, tf2, -tf3).doit()
TransferFunction(-p**2*(-p + s)*(s**4 + 5*s + 6) + (-p + s)*(p +␣
↪s)*(s**3 - 2) + (p + s)*(a*p**2 + b*s)*(s**4 + 5*s + 6), (-p + s)*(p +␣
↪s)*(s**4 + 5*s + 6), s)
>>> Parallel(tf2, Series(tf1, -tf3)).doit()
TransferFunction(-p**2*(a*p**2 + b*s)*(s**4 + 5*s + 6) + (-p + s)*(p +␣
↪s)*(s**3 - 2), (-p + s)*(p + s)*(s**4 + 5*s + 6), s)
```

Notes

All the transfer functions should use the same complex variable `var` of the Laplace transform.

See also:

Series (page 1899), *TransferFunction* (page 1891), *Feedback* (page 1906)

doit(***hints*)

Returns the resultant transfer function obtained after evaluating the transfer functions in parallel configuration.

Examples

```
>>> from sympy.abc import s, p, a, b
>>> from sympy.physics.control.lti import TransferFunction, Parallel
>>> tf1 = TransferFunction(a*p**2 + b*s, s - p, s)
>>> tf2 = TransferFunction(s**3 - 2, s**4 + 5*s + 6, s)
>>> Parallel(tf2, tf1).doit()
TransferFunction((-p + s)*(s**3 - 2) + (a*p**2 + b*s)*(s**4 + 5*s +␣
↪6), (-p + s)*(s**4 + 5*s + 6), s)
>>> Parallel(-tf1, -tf2).doit()
TransferFunction((2 - s**3)*(-p + s) + (-a*p**2 - b*s)*(s**4 + 5*s +␣
↪6), (-p + s)*(s**4 + 5*s + 6), s)
```

property is_biproper

Returns True if degree of the numerator polynomial of the resultant transfer function is equal to degree of the denominator polynomial of the same, else False.

Examples

```
>>> from sympy.abc import s, p, a, b
>>> from sympy.physics.control.lti import TransferFunction, Parallel
>>> tf1 = TransferFunction(a*p**2 + b*s, s - p, s)
>>> tf2 = TransferFunction(p**2, p + s, s)
>>> tf3 = TransferFunction(s, s**2 + s + 1, s)
>>> P1 = Parallel(tf1, -tf2)
>>> P1.is_biproper
True
>>> P2 = Parallel(tf2, tf3)
>>> P2.is_biproper
False
```

property is_proper

Returns True if degree of the numerator polynomial of the resultant transfer function is less than or equal to degree of the denominator polynomial of the same, else False.

Examples

```
>>> from sympy.abc import s, p, a, b
>>> from sympy.physics.control.lti import TransferFunction, Parallel
>>> tf1 = TransferFunction(b*s**2 + p**2 - a*p + s, b - p**2, s)
>>> tf2 = TransferFunction(p**2 - 4*p, p**3 + 3*s + 2, s)
>>> tf3 = TransferFunction(s, s**2 + s + 1, s)
>>> P1 = Parallel(-tf2, tf1)
>>> P1.is_proper
False
>>> P2 = Parallel(tf2, tf3)
>>> P2.is_proper
True
```

property is_strictly_proper

Returns True if degree of the numerator polynomial of the resultant transfer function is strictly less than degree of the denominator polynomial of the same, else False.

Examples

```
>>> from sympy.abc import s, p, a, b
>>> from sympy.physics.control.lti import TransferFunction, Parallel
>>> tf1 = TransferFunction(a*p**2 + b*s, s - p, s)
>>> tf2 = TransferFunction(s**3 - 2, s**4 + 5*s + 6, s)
>>> tf3 = TransferFunction(s, s**2 + s + 1, s)
>>> P1 = Parallel(tf1, tf2)
>>> P1.is_strictly_proper
False
>>> P2 = Parallel(tf2, tf3)
>>> P2.is_strictly_proper
True
```

to_expr()

Returns the equivalent Expr object.

property var

Returns the complex variable used by all the transfer functions.

Examples

```
>>> from sympy.abc import p
>>> from sympy.physics.control.lti import TransferFunction, Parallel,␣
↪Series
>>> G1 = TransferFunction(p**2 + 2*p + 4, p - 6, p)
>>> G2 = TransferFunction(p, 4 - p, p)
>>> G3 = TransferFunction(0, p**4 - 1, p)
>>> Parallel(G1, G2).var
p
>>> Parallel(-G3, Series(G1, G2)).var
p
```

class sympy.physics.control.lti.**Feedback**(*sys1*, *sys2=None*, *sign=-1*)

A class for representing closed-loop feedback interconnection between two SISO input/output systems.

The first argument, sys1, is the feedforward part of the closed-loop system or in simple words, the dynamical model representing the process to be controlled. The second argument, sys2, is the feedback system and controls the fed back signal to sys1. Both sys1 and sys2 can either be Series or TransferFunction objects.

Parameters

sys1 : Series, TransferFunction

The feedforward path system.

sys2 : Series, TransferFunction, optional

The feedback path system (often a feedback controller). It is the model sitting on the feedback path.

If not specified explicitly, the sys2 is assumed to be unit (1.0) transfer function.

sign : int, optional

The sign of feedback. Can either be `1` (for positive feedback) or `-1` (for negative feedback). Default value is -1.

Raises

ValueError

When `sys1` and `sys2` are not using the same complex variable of the Laplace transform.

When a combination of `sys1` and `sys2` yields zero denominator.

TypeError

When either `sys1` or `sys2` is not a `Series` or a `TransferFunction` object.

Examples

```
>>> from sympy.abc import s
>>> from sympy.physics.control.lti import TransferFunction, Feedback
>>> plant = TransferFunction(3*s**2 + 7*s - 3, s**2 - 4*s + 2, s)
>>> controller = TransferFunction(5*s - 10, s + 7, s)
>>> F1 = Feedback(plant, controller)
>>> F1
Feedback(TransferFunction(3*s**2 + 7*s - 3, s**2 - 4*s + 2, s),␣
↪TransferFunction(5*s - 10, s + 7, s), -1)
>>> F1.var
s
>>> F1.args
(TransferFunction(3*s**2 + 7*s - 3, s**2 - 4*s + 2, s),␣
↪TransferFunction(5*s - 10, s + 7, s), -1)
```

You can get the feedforward and feedback path systems by using `.sys1` and `.sys2` respectively.

```
>>> F1.sys1
TransferFunction(3*s**2 + 7*s - 3, s**2 - 4*s + 2, s)
>>> F1.sys2
TransferFunction(5*s - 10, s + 7, s)
```

You can get the resultant closed loop transfer function obtained by negative feedback interconnection using `.doit()` method.

```
>>> F1.doit()
TransferFunction((s + 7)*(s**2 - 4*s + 2)*(3*s**2 + 7*s - 3), ((s +␣
↪7)*(s**2 - 4*s + 2) + (5*s - 10)*(3*s**2 + 7*s - 3))*(s**2 - 4*s + 2),␣
↪s)
>>> G = TransferFunction(2*s**2 + 5*s + 1, s**2 + 2*s + 3, s)
```

(continues on next page)

(continued from previous page)

```
>>> C = TransferFunction(5*s + 10, s + 10, s)
>>> F2 = Feedback(G*C, TransferFunction(1, 1, s))
>>> F2.doit()
TransferFunction((s + 10)*(5*s + 10)*(s**2 + 2*s + 3)*(2*s**2 + 5*s + 1),
 ↪ (s + 10)*((s + 10)*(s**2 + 2*s + 3) + (5*s + 10)*(2*s**2 + 5*s +␣
 ↪1))*(s**2 + 2*s + 3), s)
```

To negate a `Feedback` object, the `-` operator can be prepended:

```
>>> -F1
Feedback(TransferFunction(-3*s**2 - 7*s + 3, s**2 - 4*s + 2, s),␣
 ↪TransferFunction(10 - 5*s, s + 7, s), -1)
>>> -F2
Feedback(Series(TransferFunction(-1, 1, s), TransferFunction(2*s**2 +␣
 ↪5*s + 1, s**2 + 2*s + 3, s), TransferFunction(5*s + 10, s + 10, s)),␣
 ↪TransferFunction(-1, 1, s), -1)
```

See also:

MIMOFeedback (page 1927), *Series* (page 1899), *Parallel* (page 1903)

doit(*cancel=False*, *expand=False*, ***hints*)

Returns the resultant transfer function obtained by the feedback interconnection.

Examples

```
>>> from sympy.abc import s
>>> from sympy.physics.control.lti import TransferFunction, Feedback
>>> plant = TransferFunction(3*s**2 + 7*s - 3, s**2 - 4*s + 2, s)
>>> controller = TransferFunction(5*s - 10, s + 7, s)
>>> F1 = Feedback(plant, controller)
>>> F1.doit()
TransferFunction((s + 7)*(s**2 - 4*s + 2)*(3*s**2 + 7*s - 3), ((s +␣
 ↪7)*(s**2 - 4*s + 2) + (5*s - 10)*(3*s**2 + 7*s - 3))*(s**2 - 4*s +␣
 ↪2), s)
>>> G = TransferFunction(2*s**2 + 5*s + 1, s**2 + 2*s + 3, s)
>>> F2 = Feedback(G, TransferFunction(1, 1, s))
>>> F2.doit()
TransferFunction((s**2 + 2*s + 3)*(2*s**2 + 5*s + 1), (s**2 + 2*s +␣
 ↪3)*(3*s**2 + 7*s + 4), s)
```

Use kwarg `expand=True` to expand the resultant transfer function. Use `cancel=True` to cancel out the common terms in numerator and denominator.

```
>>> F2.doit(cancel=True, expand=True)
TransferFunction(2*s**2 + 5*s + 1, 3*s**2 + 7*s + 4, s)
>>> F2.doit(expand=True)
TransferFunction(2*s**4 + 9*s**3 + 17*s**2 + 17*s + 3, 3*s**4 +␣
 ↪13*s**3 + 27*s**2 + 29*s + 12, s)
```

property sensitivity

Returns the sensitivity function of the feedback loop.

Sensitivity of a Feedback system is the ratio of change in the open loop gain to the change in the closed loop gain.

Note: This method would not return the complementary sensitivity function.

Examples

```
>>> from sympy.abc import p
>>> from sympy.physics.control.lti import TransferFunction, Feedback
>>> C = TransferFunction(5*p + 10, p + 10, p)
>>> P = TransferFunction(1 - p, p + 2, p)
>>> F_1 = Feedback(P, C)
>>> F_1.sensitivity
1/((1 - p)*(5*p + 10)/((p + 2)*(p + 10)) + 1)
```

property sign

Returns the type of MIMO Feedback model. 1 for Positive and -1 for Negative.

property sys1

Returns the feedforward system of the feedback interconnection.

Examples

```
>>> from sympy.abc import s, p
>>> from sympy.physics.control.lti import TransferFunction, Feedback
>>> plant = TransferFunction(3*s**2 + 7*s - 3, s**2 - 4*s + 2, s)
>>> controller = TransferFunction(5*s - 10, s + 7, s)
>>> F1 = Feedback(plant, controller)
>>> F1.sys1
TransferFunction(3*s**2 + 7*s - 3, s**2 - 4*s + 2, s)
>>> G = TransferFunction(2*s**2 + 5*s + 1, p**2 + 2*p + 3, p)
>>> C = TransferFunction(5*p + 10, p + 10, p)
>>> P = TransferFunction(1 - s, p + 2, p)
>>> F2 = Feedback(TransferFunction(1, 1, p), G*C*P)
>>> F2.sys1
TransferFunction(1, 1, p)
```

property sys2

Returns the feedback controller of the feedback interconnection.

Examples

```
>>> from sympy.abc import s, p
>>> from sympy.physics.control.lti import TransferFunction, Feedback
>>> plant = TransferFunction(3*s**2 + 7*s - 3, s**2 - 4*s + 2, s)
>>> controller = TransferFunction(5*s - 10, s + 7, s)
>>> F1 = Feedback(plant, controller)
>>> F1.sys2
TransferFunction(5*s - 10, s + 7, s)
>>> G = TransferFunction(2*s**2 + 5*s + 1, p**2 + 2*p + 3, p)
>>> C = TransferFunction(5*p + 10, p + 10, p)
>>> P = TransferFunction(1 - s, p + 2, p)
>>> F2 = Feedback(TransferFunction(1, 1, p), G*C*P)
>>> F2.sys2
Series(TransferFunction(2*s**2 + 5*s + 1, p**2 + 2*p + 3, p),␣
↪TransferFunction(5*p + 10, p + 10, p), TransferFunction(1 - s, p +␣
↪2, p))
```

property var

Returns the complex variable of the Laplace transform used by all the transfer functions involved in the feedback interconnection.

Examples

```
>>> from sympy.abc import s, p
>>> from sympy.physics.control.lti import TransferFunction, Feedback
>>> plant = TransferFunction(3*s**2 + 7*s - 3, s**2 - 4*s + 2, s)
>>> controller = TransferFunction(5*s - 10, s + 7, s)
>>> F1 = Feedback(plant, controller)
>>> F1.var
s
>>> G = TransferFunction(2*s**2 + 5*s + 1, p**2 + 2*p + 3, p)
>>> C = TransferFunction(5*p + 10, p + 10, p)
>>> P = TransferFunction(1 - s, p + 2, p)
>>> F2 = Feedback(TransferFunction(1, 1, p), G*C*P)
>>> F2.var
p
```

class sympy.physics.control.lti.**TransferFunctionMatrix**(*arg*)

A class for representing the MIMO (multiple-input and multiple-output) generalization of the SISO (single-input and single-output) transfer function.

It is a matrix of transfer functions (`TransferFunction`, SISO-`Series` or SISO-`Parallel`). There is only one argument, `arg` which is also the compulsory argument. `arg` is expected to be strictly of the type list of lists which holds the transfer functions or reducible to transfer functions.

Parameters

arg : Nested `List` (strictly).

Users are expected to input a nested list of `TransferFunction`, `Series` and/or `Parallel` objects.

Examples

Note: pprint() can be used for better visualization of TransferFunctionMatrix objects.

```
>>> from sympy.abc import s, p, a
>>> from sympy import pprint
>>> from sympy.physics.control.lti import TransferFunction,␣
↪TransferFunctionMatrix, Series, Parallel
>>> tf_1 = TransferFunction(s + a, s**2 + s + 1, s)
>>> tf_2 = TransferFunction(p**4 - 3*p + 2, s + p, s)
>>> tf_3 = TransferFunction(3, s + 2, s)
>>> tf_4 = TransferFunction(-a + p, 9*s - 9, s)
>>> tfm_1 = TransferFunctionMatrix([[tf_1], [tf_2], [tf_3]])
>>> tfm_1
TransferFunctionMatrix(((TransferFunction(a + s, s**2 + s + 1, s),),␣
↪(TransferFunction(p**4 - 3*p + 2, p + s, s),), (TransferFunction(3, s␣
↪+ 2, s),)))
>>> tfm_1.var
s
>>> tfm_1.num_inputs
1
>>> tfm_1.num_outputs
3
>>> tfm_1.shape
(3, 1)
>>> tfm_1.args
(((TransferFunction(a + s, s**2 + s + 1, s),), (TransferFunction(p**4 -␣
↪3*p + 2, p + s, s),), (TransferFunction(3, s + 2, s),)),)
>>> tfm_2 = TransferFunctionMatrix([[tf_1, -tf_3], [tf_2, -tf_1], [tf_3,␣
↪-tf_2]])
>>> tfm_2
TransferFunctionMatrix(((TransferFunction(a + s, s**2 + s + 1, s),␣
↪TransferFunction(-3, s + 2, s)), (TransferFunction(p**4 - 3*p + 2, p +␣
↪s, s), TransferFunction(-a - s, s**2 + s + 1, s)), (TransferFunction(3,
↪ s + 2, s), TransferFunction(-p**4 + 3*p - 2, p + s, s))))
>>> pprint(tfm_2, use_unicode=False)  # pretty-printing for better␣
↪visualization
[   a + s           -3       ]
[ ----------       -----     ]
[  2               s + 2     ]
[ s  + s + 1                 ]
[                            ]
[ 4                          ]
[p  - 3*p + 2      -a - s    ]
[------------    ----------  ]
[   p + s         2          ]
[                s  + s + 1  ]
[                            ]
[                 4          ]
[     3        - p  + 3*p - 2]
```

(continues on next page)

(continued from previous page)

```
[   -----      --------------]
[   s + 2          p + s     ]{t}
```

TransferFunctionMatrix can be transposed, if user wants to switch the input and output transfer functions

```
>>> tfm_2.transpose()
TransferFunctionMatrix((TransferFunction(a + s, s**2 + s + 1, s),␣
↪TransferFunction(p**4 - 3*p + 2, p + s, s), TransferFunction(3, s + 2,␣
↪s)), (TransferFunction(-3, s + 2, s), TransferFunction(-a - s, s**2 +␣
↪s + 1, s), TransferFunction(-p**4 + 3*p - 2, p + s, s)))
>>> pprint(_, use_unicode=False)
[              4                                ]
[  a + s     p  - 3*p + 2         3             ]
[----------  ------------      -----            ]
[ 2             p + s          s + 2            ]
[s  + s + 1                                     ]
[                                               ]
[                                4              ]
[   -3           -a - s      - p  + 3*p - 2     ]
[  -----      ----------   --------------]
[  s + 2       2                 p + s          ]
[             s  + s + 1                        ]{t}
```

```
>>> tf_5 = TransferFunction(5, s, s)
>>> tf_6 = TransferFunction(5*s, (2 + s**2), s)
>>> tf_7 = TransferFunction(5, (s*(2 + s**2)), s)
>>> tf_8 = TransferFunction(5, 1, s)
>>> tfm_3 = TransferFunctionMatrix([[tf_5, tf_6], [tf_7, tf_8]])
>>> tfm_3
TransferFunctionMatrix((TransferFunction(5, s, s), TransferFunction(5*s,
↪ s**2 + 2, s)), (TransferFunction(5, s*(s**2 + 2), s),␣
↪TransferFunction(5, 1, s))))
>>> pprint(tfm_3, use_unicode=False)
[    5        5*s  ]
[    -       ------]
[    s        2    ]
[             s  + 2]
[                   ]
[    5          5   ]
[----------     -   ]
[  / 2    \     1   ]
[s*\s  + 2/         ]{t}
>>> tfm_3.var
s
>>> tfm_3.shape
(2, 2)
>>> tfm_3.num_outputs
2
>>> tfm_3.num_inputs
2
>>> tfm_3.args
```

(continues on next page)

(continued from previous page)

```
(((TransferFunction(5, s, s), TransferFunction(5*s, s**2 + 2, s)),␣
↪(TransferFunction(5, s*(s**2 + 2), s), TransferFunction(5, 1, s))),)
```

To access the `TransferFunction` at any index in the `TransferFunctionMatrix`, use the index notation.

```
>>> tfm_3[1, 0]  # gives the TransferFunction present at 2nd Row and 1st␣
↪Col. Similar to that in Matrix classes
TransferFunction(5, s*(s**2 + 2), s)
>>> tfm_3[0, 0]  # gives the TransferFunction present at 1st Row and 1st␣
↪Col.
TransferFunction(5, s, s)
>>> tfm_3[:, 0]  # gives the first column
TransferFunctionMatrix(((TransferFunction(5, s, s),),␣
↪(TransferFunction(5, s*(s**2 + 2), s),)))
>>> pprint(_, use_unicode=False)
[    5     ]
[    -     ]
[    s     ]
[          ]
[    5     ]
[----------]
[  / 2    \]
[s*\s  + 2/]{t}
>>> tfm_3[0, :]  # gives the first row
TransferFunctionMatrix(((TransferFunction(5, s, s), TransferFunction(5*s,
↪ s**2 + 2, s)),))
>>> pprint(_, use_unicode=False)
[5   5*s  ]
[-  ------]
[s   2    ]
[   s  + 2]{t}
```

To negate a transfer function matrix, `-` operator can be prepended:

```
>>> tfm_4 = TransferFunctionMatrix([[tf_2], [-tf_1], [tf_3]])
>>> -tfm_4
TransferFunctionMatrix(((TransferFunction(-p**4 + 3*p - 2, p + s, s),),␣
↪(TransferFunction(a + s, s**2 + s + 1, s),), (TransferFunction(-3, s +␣
↪2, s),)))
>>> tfm_5 = TransferFunctionMatrix([[tf_1, tf_2], [tf_3, -tf_1]])
>>> -tfm_5
TransferFunctionMatrix(((TransferFunction(-a - s, s**2 + s + 1, s),␣
↪TransferFunction(-p**4 + 3*p - 2, p + s, s)), (TransferFunction(-3, s␣
↪+ 2, s), TransferFunction(a + s, s**2 + s + 1, s))))
```

`subs()` returns the `TransferFunctionMatrix` object with the value substituted in the expression. This will not mutate your original `TransferFunctionMatrix`.

```
>>> tfm_2.subs(p, 2)  #  substituting p everywhere in tfm_2 with 2.
TransferFunctionMatrix(((TransferFunction(a + s, s**2 + s + 1, s),␣
↪TransferFunction(-3, s + 2, s)), (TransferFunction(12, s + 2, s),␣
```

(continues on next page)

(continued from previous page)

```
↪TransferFunction(-a - s, s**2 + s + 1, s)), (TransferFunction(3, s + 2,
↪ s), TransferFunction(-12, s + 2, s))))
>>> pprint(_, use_unicode=False)
[  a + s         -3     ]
[----------    -----    ]
[ 2            s + 2    ]
[s  + s + 1             ]
[                       ]
[    12         -a - s  ]
[  -----      ----------]
[  s + 2       2        ]
[             s  + s + 1]
[                       ]
[    3           -12    ]
[  -----         -----  ]
[  s + 2        s + 2   ]{t}
>>> pprint(tfm_2, use_unicode=False) # State of tfm_2 is unchanged after␣
↪substitution
[   a + s            -3       ]
[ ----------         -----    ]
[  2                 s + 2    ]
[ s  + s + 1                  ]
[                             ]
[ 4                           ]
[p  - 3*p + 2        -a - s   ]
[------------     ----------  ]
[   p + s          2          ]
[                 s  + s + 1  ]
[                             ]
[                   4         ]
[     3         - p  + 3*p - 2]
[   -----       --------------]
[   s + 2            p + s    ]{t}
```

subs() also supports multiple substitutions.

```
>>> tfm_2.subs({p: 2, a: 1})  # substituting p with 2 and a with 1
TransferFunctionMatrix(((TransferFunction(s + 1, s**2 + s + 1, s),␣
↪TransferFunction(-3, s + 2, s)), (TransferFunction(12, s + 2, s),␣
↪TransferFunction(-s - 1, s**2 + s + 1, s)), (TransferFunction(3, s + 2,
↪ s), TransferFunction(-12, s + 2, s))))
>>> pprint(_, use_unicode=False)
[  s + 1         -3     ]
[----------    -----    ]
[ 2            s + 2    ]
[s  + s + 1             ]
[                       ]
[    12         -s - 1  ]
[  -----      ----------]
[  s + 2       2        ]
[             s  + s + 1]
[                       ]
```

(continues on next page)

(continued from previous page)

```
[    3          -12    ]
[  -----       -----   ]
[  s + 2       s + 2   ]{t}
```

Users can reduce the `Series` and `Parallel` elements of the matrix to `TransferFunction` by using `doit()`.

```
>>> tfm_6 = TransferFunctionMatrix([[Series(tf_3, tf_4), Parallel(tf_3,␣
↪tf_4)]])
>>> tfm_6
TransferFunctionMatrix(((Series(TransferFunction(3, s + 2, s),␣
↪TransferFunction(-a + p, 9*s - 9, s)), Parallel(TransferFunction(3, s␣
↪+ 2, s), TransferFunction(-a + p, 9*s - 9, s))),))
>>> pprint(tfm_6, use_unicode=False)
[ -a + p   3     -a + p     3  ]
[-------*-----  ------- + -----]
[9*s - 9 s + 2  9*s - 9   s + 2]{t}
>>> tfm_6.doit()
TransferFunctionMatrix(((TransferFunction(-3*a + 3*p, (s + 2)*(9*s - 9),␣
↪s), TransferFunction(27*s + (-a + p)*(s + 2) - 27, (s + 2)*(9*s - 9),␣
↪s)),))
>>> pprint(_, use_unicode=False)
[    -3*a + 3*p     27*s + (-a + p)*(s + 2) - 27]
[-----------------  ----------------------------]
[(s + 2)*(9*s - 9)       (s + 2)*(9*s - 9)      ]{t}
>>> tf_9 = TransferFunction(1, s, s)
>>> tf_10 = TransferFunction(1, s**2, s)
>>> tfm_7 = TransferFunctionMatrix([[Series(tf_9, tf_10), tf_9], [tf_10,␣
↪Parallel(tf_9, tf_10)]])
>>> tfm_7
TransferFunctionMatrix(((Series(TransferFunction(1, s, s),␣
↪TransferFunction(1, s**2, s)), TransferFunction(1, s, s)),␣
↪(TransferFunction(1, s**2, s), Parallel(TransferFunction(1, s, s),␣
↪TransferFunction(1, s**2, s)))))
>>> pprint(tfm_7, use_unicode=False)
[ 1      1   ]
[----    -   ]
[   2    s   ]
[s*s         ]
[            ]
[ 1    1    1]
[ --   -- + -]
[  2    2   s]
[ s    s     ]{t}
>>> tfm_7.doit()
TransferFunctionMatrix(((TransferFunction(1, s**3, s),␣
↪TransferFunction(1, s, s)), (TransferFunction(1, s**2, s),␣
↪TransferFunction(s**2 + s, s**3, s))))
>>> pprint(_, use_unicode=False)
[1     1   ]
[--    -   ]
[ 3    s   ]
```

(continues on next page)

(continued from previous page)

```
[s         ]
[          ]
[     2    ]
[1   s  + s]
[--  ------]
[ 2     3  ]
[s     s   ]{t}
```

Addition, subtraction, and multiplication of transfer function matrices can form unevaluated `Series` or `Parallel` objects.

- For addition and subtraction: All the transfer function matrices must have the same shape.
- For multiplication (C = A * B): The number of inputs of the first transfer function matrix (A) must be equal to the number of outputs of the second transfer function matrix (B).

Also, use pretty-printing (`pprint`) to analyse better.

```
>>> tfm_8 = TransferFunctionMatrix([[tf_3], [tf_2], [-tf_1]])
>>> tfm_9 = TransferFunctionMatrix([[-tf_3]])
>>> tfm_10 = TransferFunctionMatrix([[tf_1], [tf_2], [tf_4]])
>>> tfm_11 = TransferFunctionMatrix([[tf_4], [-tf_1]])
>>> tfm_12 = TransferFunctionMatrix([[tf_4, -tf_1, tf_3], [-tf_2, -tf_4,␣
↪-tf_3]])
>>> tfm_8 + tfm_10
MIMOParallel(TransferFunctionMatrix(((TransferFunction(3, s + 2, s),),␣
↪(TransferFunction(p**4 - 3*p + 2, p + s, s),), (TransferFunction(-a -␣
↪s, s**2 + s + 1, s),))), TransferFunctionMatrix(((TransferFunction(a +␣
↪s, s**2 + s + 1, s),), (TransferFunction(p**4 - 3*p + 2, p + s, s),),␣
↪(TransferFunction(-a + p, 9*s - 9, s),))))
>>> pprint(_, use_unicode=False)
[     3      ]      [   a + s    ]
[   -----    ]      [ ---------- ]
[   s + 2    ]      [  2         ]
[            ]      [ s  + s + 1 ]
[ 4          ]      [            ]
[p  - 3*p + 2]      [ 4          ]
[------------]    + [p  - 3*p + 2]
[   p + s    ]      [------------]
[            ]      [   p + s    ]
[   -a - s   ]      [            ]
[ ---------- ]      [   -a + p   ]
[  2         ]      [  -------   ]
[ s  + s + 1 ]{t}   [  9*s - 9   ]{t}
>>> -tfm_10 - tfm_8
MIMOParallel(TransferFunctionMatrix(((TransferFunction(-a - s, s**2 + s␣
↪+ 1, s),), (TransferFunction(-p**4 + 3*p - 2, p + s, s),),␣
↪(TransferFunction(a - p, 9*s - 9, s),))),␣
↪TransferFunctionMatrix(((TransferFunction(-3, s + 2, s),),␣
↪(TransferFunction(-p**4 + 3*p - 2, p + s, s),), (TransferFunction(a +␣
↪s, s**2 + s + 1, s),))))
```

(continues on next page)