(continued from previous page)

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

(continues on next page)

(continued from previous page)

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

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

```
>>> (-tfm_8 + tfm_10 + tfm_8*tfm_9).doit()
TransferFunctionMatrix(((TransferFunction((a + s)*(s + 2)**3 - 3*(s +␣
↪2)**2*(s**2 + s + 1) - 9*(s + 2)*(s**2 + s + 1), (s + 2)**3*(s**2 + s␣
↪+ 1), s),), (TransferFunction((p + s)*(-3*p**4 + 9*p - 6), (p +␣
↪s)**2*(s + 2), s),), (TransferFunction((-a + p)*(s + 2)*(s**2 + s +␣
↪1)**2 + (a + s)*(s + 2)*(9*s - 9)*(s**2 + s + 1) + (3*a + 3*s)*(9*s -␣
↪9)*(s**2 + s + 1), (s + 2)*(9*s - 9)*(s**2 + s + 1)**2, s),)))
>>> (-tfm_12 * -tfm_8 * -tfm_9).rewrite(TransferFunctionMatrix)
TransferFunctionMatrix(((TransferFunction(3*(-3*a + 3*p)*(p + s)*(s +␣
↪2)*(s**2 + s + 1)**2 + 3*(-3*a - 3*s)*(p + s)*(s + 2)*(9*s - 9)*(s**2␣
↪+ s + 1) + 3*(a + s)*(s + 2)**2*(9*s - 9)*(-p**4 + 3*p - 2)*(s**2 + s␣
↪+ 1), (p + s)*(s + 2)**3*(9*s - 9)*(s**2 + s + 1)**2, s),),␣
↪(TransferFunction(3*(-a + p)*(p + s)*(s + 2)**2*(-p**4 + 3*p -␣
↪2)*(s**2 + s + 1) + 3*(3*a + 3*s)*(p + s)**2*(s + 2)*(9*s - 9) + 3*(p␣
↪+ s)*(s + 2)*(9*s - 9)*(-3*p**4 + 9*p - 6)*(s**2 + s + 1), (p +␣
↪s)**2*(s + 2)**3*(9*s - 9)*(s**2 + s + 1), s),)))
```

See also:

TransferFunction (page 1891), *MIMOSeries* (page 1922), *MIMOParallel* (page 1925),

Feedback (page 1906)

elem_poles()

Returns the poles of each element of the TransferFunctionMatrix.

Note: Actual poles of a MIMO system are NOT the poles of individual elements.

Examples

```
>>> from sympy.abc import s
>>> from sympy.physics.control.lti import TransferFunction,␣
↪TransferFunctionMatrix
>>> tf_1 = TransferFunction(3, (s + 1), s)
>>> tf_2 = TransferFunction(s + 6, (s + 1)*(s + 2), s)
>>> tf_3 = TransferFunction(s + 3, s**2 + 3*s + 2, s)
>>> tf_4 = TransferFunction(s + 2, s**2 + 5*s - 10, s)
>>> tfm_1 = TransferFunctionMatrix([[tf_1, tf_2], [tf_3, tf_4]])
>>> tfm_1
TransferFunctionMatrix(((TransferFunction(3, s + 1, s),␣
↪TransferFunction(s + 6, (s + 1)*(s + 2), s)), (TransferFunction(s +␣
↪3, s**2 + 3*s + 2, s), TransferFunction(s + 2, s**2 + 5*s - 10,␣
↪s))))
>>> tfm_1.elem_poles()
[[[-1], [-2, -1]], [[-2, -1], [-5/2 + sqrt(65)/2, -sqrt(65)/2 - 5/2]]]
```

See also:

elem_zeros (page 1919)

elem_zeros()

Returns the zeros of each element of the TransferFunctionMatrix.

Note: Actual zeros of a MIMO system are NOT the zeros of individual elements.

Examples

```
>>> from sympy.abc import s
>>> from sympy.physics.control.lti import TransferFunction,␣
↪TransferFunctionMatrix
>>> tf_1 = TransferFunction(3, (s + 1), s)
>>> tf_2 = TransferFunction(s + 6, (s + 1)*(s + 2), s)
>>> tf_3 = TransferFunction(s + 3, s**2 + 3*s + 2, s)
>>> tf_4 = TransferFunction(s**2 - 9*s + 20, s**2 + 5*s - 10, s)
>>> tfm_1 = TransferFunctionMatrix([[tf_1, tf_2], [tf_3, tf_4]])
>>> tfm_1
TransferFunctionMatrix(((TransferFunction(3, s + 1, s),␣
↪TransferFunction(s + 6, (s + 1)*(s + 2), s)), (TransferFunction(s +␣
↪3, s**2 + 3*s + 2, s), TransferFunction(s**2 - 9*s + 20, s**2 + 5*s␣
```

(continues on next page)

(continued from previous page)

```
↪- 10, s))))
>>> tfm_1.elem_zeros()
[[[], [-6]], [[-3], [4, 5]]]
```

See also:

elem_poles (page 1919)

expand(***hints*)

Expands the transfer function matrix

classmethod from_Matrix(*matrix*, *var*)

Creates a new `TransferFunctionMatrix` efficiently from a SymPy Matrix of `Expr` objects.

Parameters

matrix : `ImmutableMatrix` having `Expr/Number` elements.

var : Symbol

Complex variable of the Laplace transform which will be used by the all the `TransferFunction` objects in the `TransferFunctionMatrix`.

Examples

```
>>> from sympy.abc import s
>>> from sympy.physics.control.lti import TransferFunctionMatrix
>>> from sympy import Matrix, pprint
>>> M = Matrix([[s, 1/s], [1/(s+1), s]])
>>> M_tf = TransferFunctionMatrix.from_Matrix(M, s)
>>> pprint(M_tf, use_unicode=False)
[  s    1]
[  -    -]
[  1    s]
[        ]
[  1    s]
[-----  -]
[s + 1  1]{t}
>>> M_tf.elem_poles()
[[[], [0]], [[-1], []]]
>>> M_tf.elem_zeros()
[[[0], []], [[], [0]]]
```

property num_inputs

Returns the number of inputs of the system.

Examples

```
>>> from sympy.abc import s, p
>>> from sympy.physics.control.lti import TransferFunction,␣
↪TransferFunctionMatrix
>>> G1 = TransferFunction(s + 3, s**2 - 3, s)
>>> G2 = TransferFunction(4, s**2, s)
>>> G3 = TransferFunction(p**2 + s**2, p - 3, s)
>>> tfm_1 = TransferFunctionMatrix([[G2, -G1, G3], [-G2, -G1, -G3]])
>>> tfm_1.num_inputs
3
```

See also:

num_outputs (page 1921)

property num_outputs

Returns the number of outputs of the system.

Examples

```
>>> from sympy.abc import s
>>> from sympy.physics.control.lti import TransferFunctionMatrix
>>> from sympy import Matrix
>>> M_1 = Matrix([[s], [1/s]])
>>> TFM = TransferFunctionMatrix.from_Matrix(M_1, s)
>>> print(TFM)
TransferFunctionMatrix(((TransferFunction(s, 1, s),),␣
↪(TransferFunction(1, s, s),)))
>>> TFM.num_outputs
2
```

See also:

num_inputs (page 1920)

property shape

Returns the shape of the transfer function matrix, that is, `(# of outputs, # of inputs)`.

Examples

```
>>> from sympy.abc import s, p
>>> from sympy.physics.control.lti import TransferFunction,␣
↪TransferFunctionMatrix
>>> tf1 = TransferFunction(p**2 - 1, s**4 + s**3 - p, p)
>>> tf2 = TransferFunction(1 - p, p**2 - 3*p + 7, p)
>>> tf3 = TransferFunction(3, 4, p)
>>> tfm1 = TransferFunctionMatrix([[tf1, -tf2]])
>>> tfm1.shape
(1, 2)
```

(continues on next page)

(continued from previous page)

```
>>> tfm2 = TransferFunctionMatrix([[-tf2, tf3], [tf1, -tf1]])
>>> tfm2.shape
(2, 2)
```

transpose()

Returns the transpose of the TransferFunctionMatrix (switched input and output layers).

property var

Returns the complex variable used by all the transfer functions or Series/Parallel objects in a transfer function matrix.

Examples

```
>>> from sympy.abc import p, s
>>> from sympy.physics.control.lti import TransferFunction,␣
↪TransferFunctionMatrix, 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)
>>> G4 = TransferFunction(s + 1, s**2 + s + 1, s)
>>> S1 = Series(G1, G2)
>>> S2 = Series(-G3, Parallel(G2, -G1))
>>> tfm1 = TransferFunctionMatrix([[G1], [G2], [G3]])
>>> tfm1.var
p
>>> tfm2 = TransferFunctionMatrix([[-S1, -S2], [S1, S2]])
>>> tfm2.var
p
>>> tfm3 = TransferFunctionMatrix([[G4]])
>>> tfm3.var
s
```

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

A class for representing a series configuration of MIMO systems.

Parameters

args : MIMOLinearTimeInvariant

MIMO systems in a series configuration.

evaluate : Boolean, Keyword

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

Raises

ValueError

When no argument is passed.

var attribute is not same for every system.

num_outputs of the MIMO system is not equal to the num_inputs of its adjacent MIMO system. (Matrix multiplication constraint, basically)

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, MIMO in this case.

Examples

```
>>> from sympy.abc import s
>>> from sympy.physics.control.lti import MIMOSeries,␣
↪TransferFunctionMatrix
>>> from sympy import Matrix, pprint
>>> mat_a = Matrix([[5*s], [5]])  # 2 Outputs 1 Input
>>> mat_b = Matrix([[5, 1/(6*s**2)]])  # 1 Output 2 Inputs
>>> mat_c = Matrix([[1, s], [5/s, 1]])  # 2 Outputs 2 Inputs
>>> tfm_a = TransferFunctionMatrix.from_Matrix(mat_a, s)
>>> tfm_b = TransferFunctionMatrix.from_Matrix(mat_b, s)
>>> tfm_c = TransferFunctionMatrix.from_Matrix(mat_c, s)
>>> MIMOSeries(tfm_c, tfm_b, tfm_a)
MIMOSeries(TransferFunctionMatrix(((TransferFunction(1, 1, s),␣
↪TransferFunction(s, 1, s)), (TransferFunction(5, s, s),␣
↪TransferFunction(1, 1, s)))),␣
↪TransferFunctionMatrix(((TransferFunction(5, 1, s), TransferFunction(1,
↪ 6*s**2, s)),)), TransferFunctionMatrix(((TransferFunction(5*s, 1, s),
↪), (TransferFunction(5, 1, s),))))
>>> pprint(_, use_unicode=False)  # For Better Visualization
[5*s]                  [1  s]
[---]    [5   1  ]     [-  -]
[ 1 ]    [-  ----]     [1  1]
[   ]   *[1     2]   *[    ]
[ 5 ]    [   6*s ]{t} [5  1]
[ - ]                  [-  -]
[ 1 ]{t}               [s  1]{t}
>>> MIMOSeries(tfm_c, tfm_b, tfm_a).doit()
TransferFunctionMatrix(((TransferFunction(150*s**4 + 25*s, 6*s**3, s),␣
↪TransferFunction(150*s**4 + 5*s, 6*s**2, s)),␣
↪(TransferFunction(150*s**3 + 25, 6*s**3, s), TransferFunction(150*s**3␣
↪+ 5, 6*s**2, s))))
>>> pprint(_, use_unicode=False)  # (2 Inputs -A-> 2 Outputs) -> (2␣
↪Inputs -B-> 1 Output) -> (1 Input -C-> 2 Outputs) is equivalent to (2␣
↪Inputs -Series Equivalent-> 2 Outputs).
[     4              4      ]
[150*s  + 25*s  150*s  + 5*s]
[-------------  ------------]
[        3             2    ]
[     6*s           6*s     ]
[                           ]
[      3             3      ]
[ 150*s  + 25    150*s  + 5  ]
[ -----------    ----------- ]
[        3             2    ]
[     6*s           6*s     ]{t}
```

Notes

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

`MIMOSeries(A, B)` is not equivalent to `A*B`. It is always in the reverse order, that is `B*A`.

See also:

Series (page 1899), *MIMOParallel* (page 1925)

doit(*cancel=False*, ***kwargs*)

Returns the resultant transfer function matrix obtained after evaluating the MIMO systems arranged in a series configuration.

Examples

```
>>> from sympy.abc import s, p, a, b
>>> from sympy.physics.control.lti import TransferFunction,␣
↪MIMOSeries, TransferFunctionMatrix
>>> tf1 = TransferFunction(a*p**2 + b*s, s - p, s)
>>> tf2 = TransferFunction(s**3 - 2, s**4 + 5*s + 6, s)
>>> tfm1 = TransferFunctionMatrix([[tf1, tf2], [tf2, tf2]])
>>> tfm2 = TransferFunctionMatrix([[tf2, tf1], [tf1, tf1]])
>>> MIMOSeries(tfm2, tfm1).doit()
TransferFunctionMatrix(((TransferFunction(2*(-p + s)*(s**3 -␣
↪2)*(a*p**2 + b*s)*(s**4 + 5*s + 6), (-p + s)**2*(s**4 + 5*s + 6)**2,
↪ s), TransferFunction((-p + s)**2*(s**3 - 2)*(a*p**2 + b*s) + (-p +␣
↪s)*(a*p**2 + b*s)**2*(s**4 + 5*s + 6), (-p + s)**3*(s**4 + 5*s + 6),
↪ s)), (TransferFunction((-p + s)*(s**3 - 2)**2*(s**4 + 5*s + 6) +␣
↪(s**3 - 2)*(a*p**2 + b*s)*(s**4 + 5*s + 6)**2, (-p + s)*(s**4 + 5*s␣
↪+ 6)**3, s), TransferFunction(2*(s**3 - 2)*(a*p**2 + b*s), (-p +␣
↪s)*(s**4 + 5*s + 6), s))))
```

property num_inputs

Returns the number of input signals of the series system.

property num_outputs

Returns the number of output signals of the series system.

property shape

Returns the shape of the equivalent MIMO system.

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,␣
↪MIMOSeries, TransferFunctionMatrix
>>> G1 = TransferFunction(p**2 + 2*p + 4, p - 6, p)
>>> G2 = TransferFunction(p, 4 - p, p)
>>> G3 = TransferFunction(0, p**4 - 1, p)
>>> tfm_1 = TransferFunctionMatrix([[G1, G2, G3]])
>>> tfm_2 = TransferFunctionMatrix([[G1], [G2], [G3]])
>>> MIMOSeries(tfm_2, tfm_1).var
p
```

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

A class for representing a parallel configuration of MIMO systems.

Parameters

args : MIMOLinearTimeInvariant

MIMO Systems in a parallel arrangement.

evaluate : Boolean, Keyword

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

Raises

ValueError

When no argument is passed.

`var` attribute is not same for every system.

All MIMO systems passed do not have same shape.

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, MIMO in this case.

Examples

```
>>> from sympy.abc import s
>>> from sympy.physics.control.lti import TransferFunctionMatrix,␣
↪MIMOParallel
>>> from sympy import Matrix, pprint
>>> expr_1 = 1/s
>>> expr_2 = s/(s**2-1)
>>> expr_3 = (2 + s)/(s**2 - 1)
>>> expr_4 = 5
>>> tfm_a = TransferFunctionMatrix.from_Matrix(Matrix([[expr_1, expr_2],␣
↪[expr_3, expr_4]]), s)
>>> tfm_b = TransferFunctionMatrix.from_Matrix(Matrix([[expr_2, expr_1],␣
↪[expr_4, expr_3]]), s)
>>> tfm_c = TransferFunctionMatrix.from_Matrix(Matrix([[expr_3, expr_4],␣
```

(continues on next page)

(continued from previous page)

```
 ↪[expr_1, expr_2]]), s)
>>> MIMOParallel(tfm_a, tfm_b, tfm_c)
MIMOParallel(TransferFunctionMatrix(((TransferFunction(1, s, s),␣
 ↪TransferFunction(s, s**2 - 1, s)), (TransferFunction(s + 2, s**2 - 1,␣
 ↪s), TransferFunction(5, 1, s)))),␣
 ↪TransferFunctionMatrix(((TransferFunction(s, s**2 - 1, s),␣
 ↪TransferFunction(1, s, s)), (TransferFunction(5, 1, s),␣
 ↪TransferFunction(s + 2, s**2 - 1, s)))),␣
 ↪TransferFunctionMatrix(((TransferFunction(s + 2, s**2 - 1, s),␣
 ↪TransferFunction(5, 1, s)), (TransferFunction(1, s, s),␣
 ↪TransferFunction(s, s**2 - 1, s)))))
>>> pprint(_, use_unicode=False)  #  For Better Visualization
[  1       s   ]      [  s       1   ]      [s + 2     5   ]
[  -     ------]      [------    -   ]      [------    -   ]
[  s      2    ]      [ 2        s   ]      [ 2        1   ]
[        s  - 1]      [s  - 1        ]      [s  - 1        ]
[              ]    + [              ]    + [              ]
[s + 2     5   ]      [  5     s + 2 ]      [  1       s   ]
[------    -   ]      [  -     ------]      [  -     ------]
[ 2        1   ]      [  1      2    ]      [  s      2    ]
[s  - 1        ]{t}   [         s  - 1]{t}   [        s  - 1]{t}
>>> MIMOParallel(tfm_a, tfm_b, tfm_c).doit()
TransferFunctionMatrix(((TransferFunction(s**2 + s*(2*s + 2) - 1,␣
 ↪s*(s**2 - 1), s), TransferFunction(2*s**2 + 5*s*(s**2 - 1) - 1,␣
 ↪s*(s**2 - 1), s)), (TransferFunction(s**2 + s*(s + 2) + 5*s*(s**2 - 1)␣
 ↪- 1, s*(s**2 - 1), s), TransferFunction(5*s**2 + 2*s - 3, s**2 - 1,␣
 ↪s))))
>>> pprint(_, use_unicode=False)
[       2                                  2       / 2    \    ]
[      s  + s*(2*s + 2) - 1             2*s  + 5*s*\s  - 1/ - 1]
[      --------------------             -----------------------]
[             / 2    \                          / 2    \      ]
[           s*\s  - 1/                        s*\s  - 1/      ]
[                                                              ]
[ 2                   / 2    \              2                  ]
[s  + s*(s + 2) + 5*s*\s  - 1/ - 1       5*s  + 2*s - 3       ]
[---------------------------------       --------------       ]
[             / 2    \                         2               ]
[           s*\s  - 1/                        s  - 1           ]{t}
```

Notes

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

See also:

Parallel (page 1903), *MIMOSeries* (page 1922)

doit(***hints*)

Returns the resultant transfer function matrix obtained after evaluating the MIMO systems arranged in a parallel configuration.

Examples

```
>>> from sympy.abc import s, p, a, b
>>> from sympy.physics.control.lti import TransferFunction,␣
↪MIMOParallel, TransferFunctionMatrix
>>> tf1 = TransferFunction(a*p**2 + b*s, s - p, s)
>>> tf2 = TransferFunction(s**3 - 2, s**4 + 5*s + 6, s)
>>> tfm_1 = TransferFunctionMatrix([[tf1, tf2], [tf2, tf1]])
>>> tfm_2 = TransferFunctionMatrix([[tf2, tf1], [tf1, tf2]])
>>> MIMOParallel(tfm_1, tfm_2).doit()
TransferFunctionMatrix(((TransferFunction((-p + s)*(s**3 - 2) +␣
↪(a*p**2 + b*s)*(s**4 + 5*s + 6), (-p + s)*(s**4 + 5*s + 6), s),␣
↪TransferFunction((-p + s)*(s**3 - 2) + (a*p**2 + b*s)*(s**4 + 5*s +␣
↪6), (-p + s)*(s**4 + 5*s + 6), s)), (TransferFunction((-p +␣
↪s)*(s**3 - 2) + (a*p**2 + b*s)*(s**4 + 5*s + 6), (-p + s)*(s**4 +␣
↪5*s + 6), s), TransferFunction((-p + s)*(s**3 - 2) + (a*p**2 +␣
↪b*s)*(s**4 + 5*s + 6), (-p + s)*(s**4 + 5*s + 6), s))))
```

property num_inputs

Returns the number of input signals of the parallel system.

property num_outputs

Returns the number of output signals of the parallel system.

property shape

Returns the shape of the equivalent MIMO system.

property var

Returns the complex variable used by all the systems.

Examples

```
>>> from sympy.abc import p
>>> from sympy.physics.control.lti import TransferFunction,␣
↪TransferFunctionMatrix, MIMOParallel
>>> G1 = TransferFunction(p**2 + 2*p + 4, p - 6, p)
>>> G2 = TransferFunction(p, 4 - p, p)
>>> G3 = TransferFunction(0, p**4 - 1, p)
>>> G4 = TransferFunction(p**2, p**2 - 1, p)
>>> tfm_a = TransferFunctionMatrix([[G1, G2], [G3, G4]])
>>> tfm_b = TransferFunctionMatrix([[G2, G1], [G4, G3]])
>>> MIMOParallel(tfm_a, tfm_b).var
p
```

class sympy.physics.control.lti.**MIMOFeedback**(*sys1*, *sys2*, *sign=-1*)

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

Parameters

sys1 : MIMOSeries, TransferFunctionMatrix

The MIMO system placed on the feedforward path.

sys2 : MIMOSeries, TransferFunctionMatrix

The system placed on the feedback path (often a feedback controller).

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.

Forward path model should have an equal number of inputs/outputs to the feedback path outputs/inputs.

When product of `sys1` and `sys2` is not a square matrix.

When the equivalent MIMO system is not invertible.

TypeError

When either `sys1` or `sys2` is not a `MIMOSeries` or a `TransferFunctionMatrix` object.

Examples

```
>>> from sympy import Matrix, pprint
>>> from sympy.abc import s
>>> from sympy.physics.control.lti import TransferFunctionMatrix,␣
↪MIMOFeedback
>>> plant_mat = Matrix([[1, 1/s], [0, 1]])
>>> controller_mat = Matrix([[10, 0], [0, 10]])  # Constant Gain
>>> plant = TransferFunctionMatrix.from_Matrix(plant_mat, s)
>>> controller = TransferFunctionMatrix.from_Matrix(controller_mat, s)
>>> feedback = MIMOFeedback(plant, controller)  # Negative Feedback␣
↪(default)
>>> pprint(feedback, use_unicode=False)
/    [1  1]    [10  0 ]   \-1   [1  1]
|    [-  -]    [--  - ]   |     [-  -]
|    [1  s]    [1   1 ]   |     [1  s]
|I + [    ]   *[      ]   |   * [    ]
|    [0  1]    [0   10]   |     [0  1]
|    [-  -]    [-   --]   |     [-  -]
\    [1  1]{t} [1   1 ]{t}/     [1  1]{t}
```

To get the equivalent system matrix, use either `doit` or `rewrite` method.

```
>>> pprint(feedback.doit(), use_unicode=False)
[1     1  ]
[--  -----]
[11  121*s]
[          ]
[0    1    ]
[-    --   ]
[1    11   ]{t}
```

To negate the MIMOFeedback object, use - operator.

```
>>> neg_feedback = -feedback
>>> pprint(neg_feedback.doit(), use_unicode=False)
[-1    -1  ]
[---  -----]
[ 11  121*s]
[           ]
[ 0    -1   ]
[ -    ---  ]
[ 1     11  ]{t}
```

See also:

Feedback (page 1906), *MIMOSeries* (page 1922), *MIMOParallel* (page 1925)

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

Returns the resultant transfer function matrix obtained by the feedback interconnection.

Examples

```
>>> from sympy import pprint
>>> from sympy.abc import s
>>> from sympy.physics.control.lti import TransferFunction,␣
↪TransferFunctionMatrix, MIMOFeedback
>>> tf1 = TransferFunction(s, 1 - s, s)
>>> tf2 = TransferFunction(1, s, s)
>>> tf3 = TransferFunction(5, 1, s)
>>> tf4 = TransferFunction(s - 1, s, s)
>>> tf5 = TransferFunction(0, 1, s)
>>> sys1 = TransferFunctionMatrix([[tf1, tf2], [tf3, tf4]])
>>> sys2 = TransferFunctionMatrix([[tf3, tf5], [tf5, tf5]])
>>> F_1 = MIMOFeedback(sys1, sys2, 1)
>>> pprint(F_1, use_unicode=False)
/    [  s      1  ]    [5  0]   \-1   [  s      1  ]
|    [-----    -  ]    [-  -]   |     [-----    -  ]
|    [1 - s    s  ]    [1  1]   |     [1 - s    s  ]
|I - [            ]   *[    ]   |   * [            ]
|    [  5    s - 1]    [0  0]   |     [  5    s - 1]
|    [  -    -----]    [-  -]   |     [  -    -----]
\    [  1      s  ]{t} [1  1]{t}/     [  1      s  ]{t}
>>> pprint(F_1.doit(), use_unicode=False)
[  -s           s - 1       ]
[-------     -----------    ]
[6*s - 1     s*(6*s - 1)    ]
[                           ]
[5*s - 5  (s - 1)*(6*s + 24)]
[-------  ------------------]
[6*s - 1     s*(6*s - 1)    ]{t}
```

If the user wants the resultant TransferFunctionMatrix object without canceling the common factors then the cancel kwarg should be passed False.

```
>>> pprint(F_1.doit(cancel=False), use_unicode=False)
[           25*s*(1 - s)                              25 - 25*s          ␣
↪    ]
[       --------------------                      --------------        ␣
↪    ]
[       25*(1 - 6*s)*(1 - s)                      25*s*(1 - 6*s)         ␣
↪    ]
[                                                                       ␣
↪    ]
[s*(25*s - 25) + 5*(1 - s)*(6*s - 1)  s*(s - 1)*(6*s - 1) + s*(25*s - ␣
↪25)]
[-----------------------------------  -------------------------------
↪---]
[          (1 - s)*(6*s - 1)                          2                ␣
↪    ]
[                                                   s *(6*s - 1)       ␣
↪    ]{t}
```

If the user wants the expanded form of the resultant transfer function matrix, the expand kwarg should be passed as `True`.

```
>>> pprint(F_1.doit(expand=True), use_unicode=False)
[  -s          s - 1      ]
[-------      --------    ]
[6*s - 1         2        ]
[              6*s  - s    ]
[                         ]
[            2            ]
[5*s - 5  6*s  + 18*s - 24]
[-------  ----------------]
[6*s - 1         2        ]
[              6*s  - s    ]{t}
```

property sensitivity

Returns the sensitivity function matrix of the feedback loop.

Sensitivity of a closed-loop 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 import pprint
>>> from sympy.abc import p
>>> from sympy.physics.control.lti import TransferFunction,␣
↪TransferFunctionMatrix, MIMOFeedback
>>> tf1 = TransferFunction(p, 1 - p, p)
>>> tf2 = TransferFunction(1, p, p)
>>> tf3 = TransferFunction(1, 1, p)
```

(continues on next page)

(continued from previous page)

```
>>> sys1 = TransferFunctionMatrix([[tf1, tf2], [tf2, tf1]])
>>> sys2 = TransferFunctionMatrix([[tf1, tf3], [tf3, tf2]])
>>> F_1 = MIMOFeedback(sys1, sys2, 1)  # Positive feedback
>>> F_2 = MIMOFeedback(sys1, sys2)  # Negative feedback
>>> pprint(F_1.sensitivity, use_unicode=False)
[   4      3      2                5      4      2           ]
[- p  + 3*p  - 4*p  + 3*p - 1    p  - 2*p  + 3*p  - 3*p + 1 ]
[----------------------------  -----------------------------]
[  4      3      2              5      4      3      2      ]
[ p  + 3*p  - 8*p  + 8*p - 3   p  + 3*p  - 8*p  + 8*p  - 3*p]
[                                                           ]
[       4    3    2                   3      2              ]
[      p  - p  - p  + p             3*p  - 6*p  + 4*p - 1   ]
[ --------------------------    --------------------------  ]
[  4      3      2               4      3      2           ]
[ p  + 3*p  - 8*p  + 8*p - 3    p  + 3*p  - 8*p  + 8*p - 3  ]
>>> pprint(F_2.sensitivity, use_unicode=False)
[ 4      3      2           5      4      2          ]
[p  - 3*p  + 2*p  + p - 1  p  - 2*p  + 3*p  - 3*p + 1]
[------------------------  --------------------------]
[   4      3                    5      4      2       ]
[  p  - 3*p  + 2*p - 1         p  - 3*p  + 2*p  - p   ]
[                                                     ]
[     4    3    2               4      3              ]
[    p  - p  - p  + p         2*p  - 3*p  + 2*p - 1   ]
[  -------------------       --------------------     ]
[   4      3                    4      3              ]
[  p  - 3*p  + 2*p - 1         p  - 3*p  + 2*p - 1    ]
```

property sign

Returns the type of feedback interconnection of two models. 1 for Positive and -1 for Negative.

property sys1

Returns the system placed on the feedforward path of the MIMO feedback interconnection.

Examples

```
>>> from sympy import pprint
>>> from sympy.abc import s
>>> from sympy.physics.control.lti import TransferFunction,␣
↪TransferFunctionMatrix, MIMOFeedback
>>> tf1 = TransferFunction(s**2 + s + 1, s**2 - s + 1, s)
>>> tf2 = TransferFunction(1, s, s)
>>> tf3 = TransferFunction(1, 1, s)
>>> sys1 = TransferFunctionMatrix([[tf1, tf2], [tf2, tf1]])
>>> sys2 = TransferFunctionMatrix([[tf3, tf3], [tf3, tf2]])
>>> F_1 = MIMOFeedback(sys1, sys2, 1)
>>> F_1.sys1
TransferFunctionMatrix(((TransferFunction(s**2 + s + 1, s**2 - s + 1,␣
```

(continues on next page)

(continued from previous page)

```
 ↪s), TransferFunction(1, s, s)), (TransferFunction(1, s, s),␣
 ↪TransferFunction(s**2 + s + 1, s**2 - s + 1, s))))
>>> pprint(_, use_unicode=False)
[ 2                    ]
[s  + s + 1      1     ]
[----------      -     ]
[ 2              s     ]
[s  - s + 1            ]
[                      ]
[             2        ]
[    1       s  + s + 1]
[    -       ----------]
[    s        2        ]
[            s  - s + 1]{t}
```

property sys2

Returns the feedback controller of the MIMO feedback interconnection.

Examples

```
>>> from sympy import pprint
>>> from sympy.abc import s
>>> from sympy.physics.control.lti import TransferFunction,␣
 ↪TransferFunctionMatrix, MIMOFeedback
>>> tf1 = TransferFunction(s**2, s**3 - s + 1, s)
>>> tf2 = TransferFunction(1, s, s)
>>> tf3 = TransferFunction(1, 1, s)
>>> sys1 = TransferFunctionMatrix([[tf1, tf2], [tf2, tf1]])
>>> sys2 = TransferFunctionMatrix([[tf1, tf3], [tf3, tf2]])
>>> F_1 = MIMOFeedback(sys1, sys2)
>>> F_1.sys2
TransferFunctionMatrix(((TransferFunction(s**2, s**3 - s + 1, s),␣
 ↪TransferFunction(1, 1, s)), (TransferFunction(1, 1, s),␣
 ↪TransferFunction(1, s, s))))
>>> pprint(_, use_unicode=False)
[     2        ]
[    s       1]
[----------  -]
[ 3          1]
[s  - s + 1   ]
[             ]
[    1       1]
[    -       -]
[    1       s]{t}
```

property var

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

Examples

```
>>> from sympy.abc import p
>>> from sympy.physics.control.lti import TransferFunction,␣
↪TransferFunctionMatrix, MIMOFeedback
>>> tf1 = TransferFunction(p, 1 - p, p)
>>> tf2 = TransferFunction(1, p, p)
>>> tf3 = TransferFunction(1, 1, p)
>>> sys1 = TransferFunctionMatrix([[tf1, tf2], [tf2, tf1]])
>>> sys2 = TransferFunctionMatrix([[tf1, tf3], [tf3, tf2]])
>>> F_1 = MIMOFeedback(sys1, sys2, 1)  # Positive feedback
>>> F_1.var
p
```

Control System Plots

This module contains plotting functions for some of the common plots used in control system. Matplotlib will be required as an external dependency if the user wants the plots. To get only the numerical data of the plots, NumPy will be required as external dependency.

Pole-Zero Plot

control_plots.**pole_zero_plot**(*pole_color='blue', pole_markersize=10, zero_color='orange', zero_markersize=7, grid=True, show_axes=True, show=True, **kwargs*)

Returns the Pole-Zero plot (also known as PZ Plot or PZ Map) of a system.

A Pole-Zero plot is a graphical representation of a system's poles and zeros. It is plotted on a complex plane, with circular markers representing the system's zeros and 'x' shaped markers representing the system's poles.

Parameters

system : SISOLinearTimeInvariant type systems

The system for which the pole-zero plot is to be computed.

pole_color : str, tuple, optional

The color of the pole points on the plot. Default color is blue. The color can be provided as a matplotlib color string, or a 3-tuple of floats each in the 0-1 range.

pole_markersize : Number, optional

The size of the markers used to mark the poles in the plot. Default pole markersize is 10.

zero_color : str, tuple, optional

The color of the zero points on the plot. Default color is orange. The color can be provided as a matplotlib color string, or a 3-tuple of floats each in the 0-1 range.

zero_markersize : Number, optional

The size of the markers used to mark the zeros in the plot. Default zero markersize is 7.

grid : boolean, optional

If `True`, the plot will have a grid. Defaults to True.

show_axes : boolean, optional

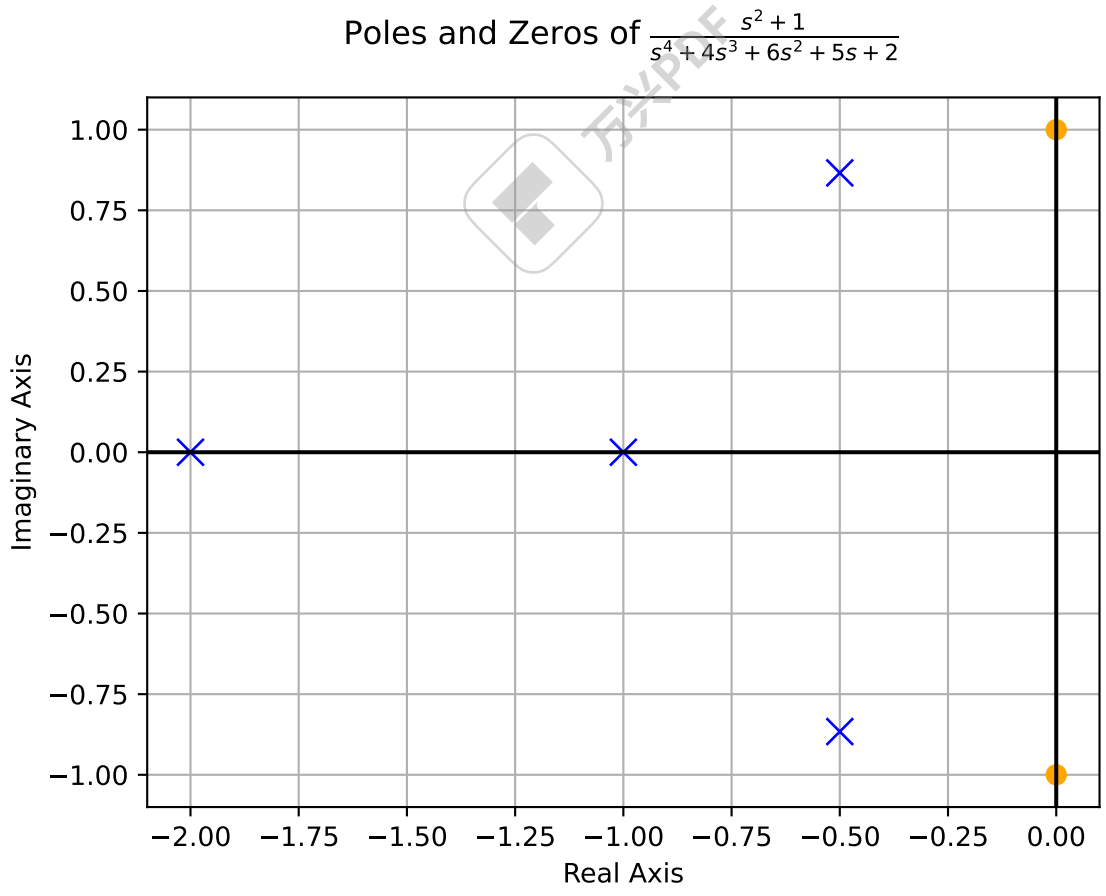
If `True`, the coordinate axes will be shown. Defaults to False.

show : boolean, optional

If `True`, the plot will be displayed otherwise the equivalent matplotlib `plot` object will be returned. Defaults to True.

Examples

```
>>> from sympy.abc import s
>>> from sympy.physics.control.lti import TransferFunction
>>> from sympy.physics.control.control_plots import pole_zero_plot
>>> tf1 = TransferFunction(s**2 + 1, s**4 + 4*s**3 + 6*s**2 + 5*s + 2, s)
>>> pole_zero_plot(tf1)
```



See also:

pole_zero_numerical_data (page 1935)

References

[R652]

control_plots.**pole_zero_numerical_data**()

Returns the numerical data of poles and zeros of the system. It is internally used by pole_zero_plot to get the data for plotting poles and zeros. Users can use this data to further analyse the dynamics of the system or plot using a different backend/plotting-module.

Parameters

system : SISOLinearTimeInvariant

The system for which the pole-zero data is to be computed.

Returns

tuple : (zeros, poles)

zeros = Zeros of the system. NumPy array of complex numbers. poles = Poles of the system. NumPy array of complex numbers.

Raises

NotImplementedError

When a SISO LTI system is not passed.

When time delay terms are present in the system.

ValueError

When more than one free symbol is present in the system. The only variable in the transfer function should be the variable of the Laplace transform.

Examples

```
>>> from sympy.abc import s
>>> from sympy.physics.control.lti import TransferFunction
>>> from sympy.physics.control.control_plots import pole_zero_numerical_
↪data
>>> tf1 = TransferFunction(s**2 + 1, s**4 + 4*s**3 + 6*s**2 + 5*s + 2, s)
>>> pole_zero_numerical_data(tf1)
([-0.+1.j  0.-1.j], [-2. +0.j        -0.5+0.8660254j -0.5-0.8660254j -1.␣
↪+0.j        ])
```

See also:

pole_zero_plot (page 1933)

Bode Plot

control_plots.**bode_plot**(*initial_exp=-5*, *final_exp=5*, *grid=True*, *show_axes=False*, *show=True*, *freq_unit='rad/sec'*, *phase_unit='rad'*, ***kwargs*)

Returns the Bode phase and magnitude plots of a continuous-time system.

Parameters

system : SISOLinearTimeInvariant type

The LTI SISO system for which the Bode Plot is to be computed.

initial_exp : Number, optional

The initial exponent of 10 of the semilog plot. Defaults to -5.

final_exp : Number, optional

The final exponent of 10 of the semilog plot. Defaults to 5.

show : boolean, optional

If `True`, the plot will be displayed otherwise the equivalent matplotlib `plot` object will be returned. Defaults to True.

prec : int, optional

The decimal point precision for the point coordinate values. Defaults to 8.

grid : boolean, optional

If `True`, the plot will have a grid. Defaults to True.

show_axes : boolean, optional

If `True`, the coordinate axes will be shown. Defaults to False.

freq_unit : string, optional

User can choose between `'rad/sec'` (radians/second) and `'Hz'` (Hertz) as frequency units.

phase_unit : string, optional

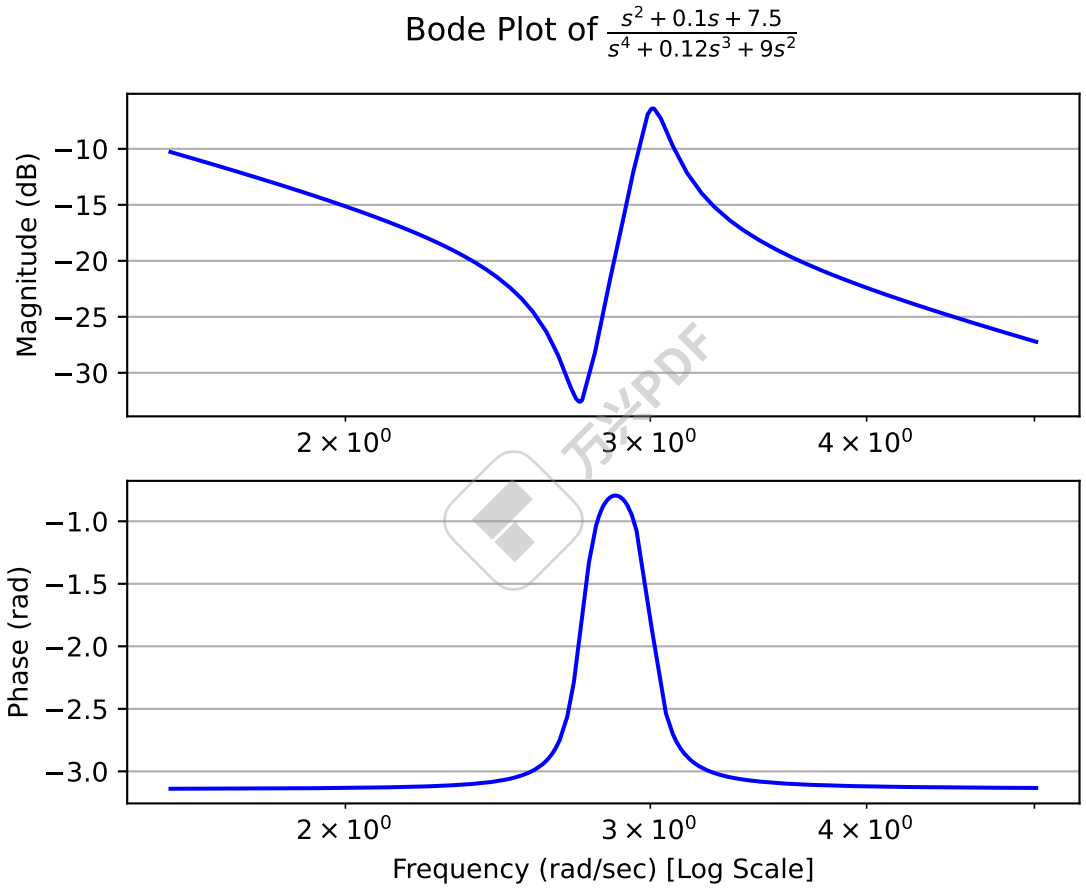
User can choose between `'rad'` (radians) and `'deg'` (degree) as phase units.

Examples

```
>>> from sympy.abc import s
>>> from sympy.physics.control.lti import TransferFunction
>>> from sympy.physics.control.control_plots import bode_plot
>>> tf1 = TransferFunction(1*s**2 + 0.1*s + 7.5, 1*s**4 + 0.12*s**3 +␣
↪9*s**2, s)
>>> bode_plot(tf1, initial_exp=0.2, final_exp=0.7)
```

See also:

bode_magnitude_plot (page 1936), *bode_phase_plot* (page 1938)


Bode Plot of $\frac{s^2 + 0.1s + 7.5}{s^4 + 0.12s^3 + 9s^2}$
Magnitude (dB)
−10
−15
−20
−25
−30
2×10^0
3×10^0
4×10^0
Phase (rad)
−1.0
−1.5
−2.0
−2.5
−3.0
2×10^0
3×10^0
4×10^0
Frequency (rad/sec) [Log Scale]

control_plots.**bode_magnitude_plot**(*initial_exp=-5*, *final_exp=5*, *color='b'*, *show_axes=False*, *grid=True*, *show=True*, *freq_unit='rad/sec'*, ***kwargs*)

Returns the Bode magnitude plot of a continuous-time system.

See `bode_plot` for all the parameters.

control_plots.**bode_phase_plot**(*initial_exp=-5*, *final_exp=5*, *color='b'*, *show_axes=False*, *grid=True*, *show=True*, *freq_unit='rad/sec'*, *phase_unit='rad'*, ***kwargs*)

Returns the Bode phase plot of a continuous-time system.

See `bode_plot` for all the parameters.

control_plots.**bode_magnitude_numerical_data**(*initial_exp=-5*, *final_exp=5*, *freq_unit='rad/sec'*, ***kwargs*)

Returns the numerical data of the Bode magnitude plot of the system. It is internally used by `bode_magnitude_plot` to get the data for plotting Bode magnitude plot. Users can use this data to further analyse the dynamics of the system or plot using a different backend/plotting-module.

Parameters

system : SISOLinearTimeInvariant

The system for which the data is to be computed.

initial_exp : Number, optional

The initial exponent of 10 of the semilog plot. Defaults to -5.

final_exp : Number, optional

The final exponent of 10 of the semilog plot. Defaults to 5.

freq_unit : string, optional

User can choose between `'rad/sec'` (radians/second) and `'Hz'` (Hertz) as frequency units.

Returns

tuple : (x, y)

x = x-axis values of the Bode magnitude plot. y = y-axis values of the Bode magnitude plot.

Raises

NotImplementedError

When a SISO LTI system is not passed.

When time delay terms are present in the system.

ValueError

When more than one free symbol is present in the system. The only variable in the transfer function should be the variable of the Laplace transform.

When incorrect frequency units are given as input.

Examples

```
>>> from sympy.abc import s
>>> from sympy.physics.control.lti import TransferFunction
>>> from sympy.physics.control.control_plots import bode_magnitude_
↪numerical_data
>>> tf1 = TransferFunction(s**2 + 1, s**4 + 4*s**3 + 6*s**2 + 5*s + 2, s)
>>> bode_magnitude_numerical_data(tf1)
([1e-05, 1.5148378120533502e-05,..., 68437.36188804005, 100000.0],
[-6.020599914256786, -6.0205999155219505,..., -193.4117304087953, -200.
↪00000000260573])
```

See also:

bode_magnitude_plot (page 1936), *bode_phase_numerical_data* (page 1939)

control_plots.**bode_phase_numerical_data**(*initial_exp=-5*, *final_exp=5*, *freq_unit='rad/sec'*, *phase_unit='rad'*, ***kwargs*)

Returns the numerical data of the Bode phase plot of the system. It is internally used by `bode_phase_plot` to get the data for plotting Bode phase plot. Users can use this data to further analyse the dynamics of the system or plot using a different backend/plotting-module.

Parameters

system : SISOLinearTimeInvariant

The system for which the Bode phase plot data is to be computed.

initial_exp : Number, optional

The initial exponent of 10 of the semilog plot. Defaults to -5.

final_exp : Number, optional

The final exponent of 10 of the semilog plot. Defaults to 5.

freq_unit : string, optional

User can choose between `'rad/sec'` (radians/second) and ‘``’Hz’`` (Hertz) as frequency units.

phase_unit : string, optional

User can choose between `'rad'` (radians) and `'deg'` (degree) as phase units.

Returns

tuple : (x, y)

x = x-axis values of the Bode phase plot. y = y-axis values of the Bode phase plot.

Raises

NotImplementedError

When a SISO LTI system is not passed.

When time delay terms are present in the system.

ValueError

When more than one free symbol is present in the system. The only variable in the transfer function should be the variable of the Laplace transform.

When incorrect frequency or phase units are given as input.

Examples

```
>>> from sympy.abc import s
>>> from sympy.physics.control.lti import TransferFunction
>>> from sympy.physics.control.control_plots import bode_phase_numerical_
↪data
>>> tf1 = TransferFunction(s**2 + 1, s**4 + 4*s**3 + 6*s**2 + 5*s + 2, s)
>>> bode_phase_numerical_data(tf1)
([1e-05, 1.4472354033813751e-05, 2.035581932165858e-05,..., 47577.
↪3248186011, 67884.09326036123, 100000.0],
[-2.5000000000291665e-05, -3.6180885085e-05, -5.08895483066e-05,...,-3.
↪1415085799262523, -3.14155265358979])
```

See also:

bode_magnitude_plot (page 1936), *bode_phase_numerical_data* (page 1939)

Impulse-Response Plot

control_plots.**impulse_response_plot**(*color='b', prec=8, lower_limit=0, upper_limit=10, show_axes=False, grid=True, show=True, **kwargs*)

Returns the unit impulse response (Input is the Dirac-Delta Function) of a continuous-time system.

Parameters

system : SISOLinearTimeInvariant type

The LTI SISO system for which the Impulse Response is to be computed.

color : str, tuple, optional

The color of the line. Default is Blue.

show : boolean, optional

If `True`, the plot will be displayed otherwise the equivalent matplotlib `plot` object will be returned. Defaults to True.

lower_limit : Number, optional

The lower limit of the plot range. Defaults to 0.

upper_limit : Number, optional

The upper limit of the plot range. Defaults to 10.

prec : int, optional

The decimal point precision for the point coordinate values. Defaults to 8.

show_axes : boolean, optional

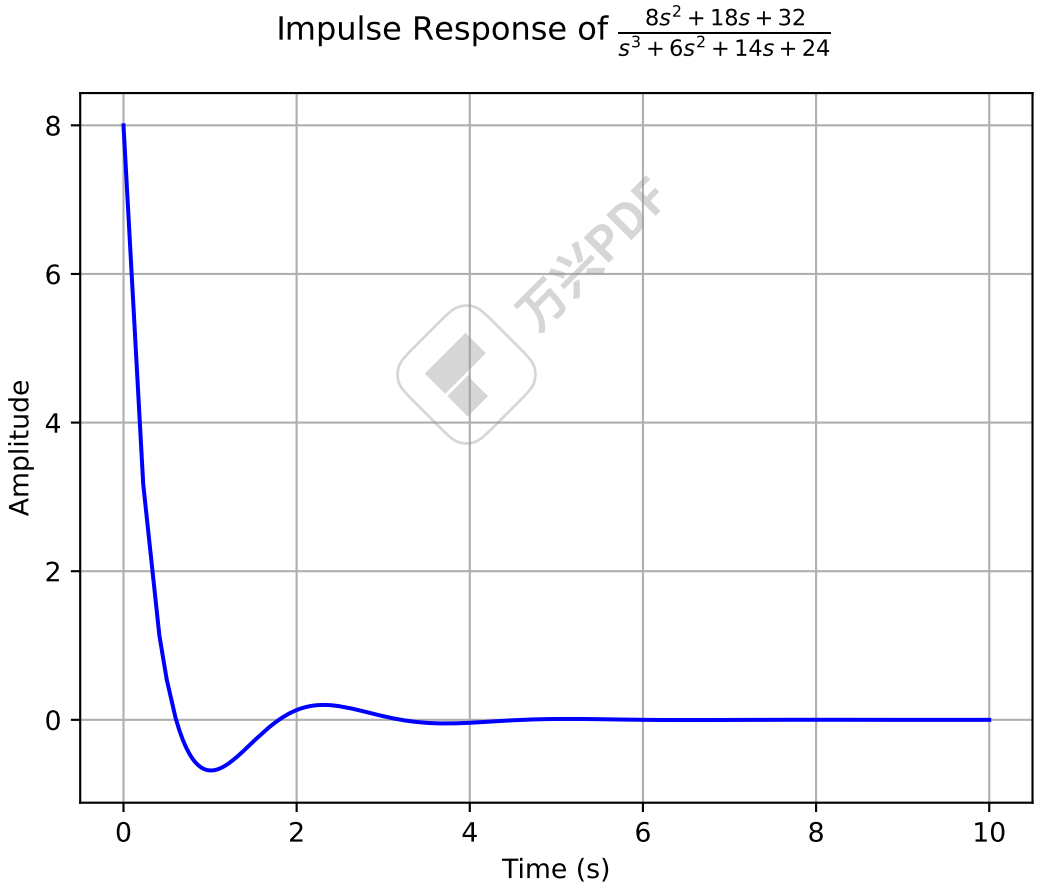
If True, the coordinate axes will be shown. Defaults to False.

grid : boolean, optional

If True, the plot will have a grid. Defaults to True.

Examples

```
>>> from sympy.abc import s
>>> from sympy.physics.control.lti import TransferFunction
>>> from sympy.physics.control.control_plots import impulse_response_plot
>>> tf1 = TransferFunction(8*s**2 + 18*s + 32, s**3 + 6*s**2 + 14*s + 24,
 ↪ s)
>>> impulse_response_plot(tf1)
```



See also:

step_response_plot (page 1943), *ramp_response_plot* (page 1946)

References

[R653]

control_plots.**impulse_response_numerical_data**(*prec=8*, *lower_limit=0*, *upper_limit=10*, ***kwargs*)

Returns the numerical values of the points in the impulse response plot of a SISO continuous-time system. By default, adaptive sampling is used. If the user wants to instead get an uniformly sampled response, then `adaptive` kwarg should be passed `False` and `nb_of_points` must be passed as additional kwargs. Refer to the parameters of class *sympy.plotting.plot.LineOver1DRangeSeries* (page 2866) for more details.

Parameters

system : SISOLinearTimeInvariant

The system for which the impulse response data is to be computed.

prec : int, optional

The decimal point precision for the point coordinate values. Defaults to 8.

lower_limit : Number, optional

The lower limit of the plot range. Defaults to 0.

upper_limit : Number, optional

The upper limit of the plot range. Defaults to 10.

kwargs :

Additional keyword arguments are passed to the underlying *sympy.plotting.plot.LineOver1DRangeSeries* (page 2866) class.

Returns

tuple : (x, y)

x = Time-axis values of the points in the impulse response. NumPy array. y = Amplitude-axis values of the points in the impulse response. NumPy array.

Raises

NotImplementedError

When a SISO LTI system is not passed.

When time delay terms are present in the system.

ValueError

When more than one free symbol is present in the system. The only variable in the transfer function should be the variable of the Laplace transform.

When `lower_limit` parameter is less than 0.

Examples

```
>>> from sympy.abc import s
>>> from sympy.physics.control.lti import TransferFunction
>>> from sympy.physics.control.control_plots import impulse_response_
↪numerical_data
>>> tf1 = TransferFunction(s, s**2 + 5*s + 8, s)
>>> impulse_response_numerical_data(tf1)
([0.0, 0.06616480200395854,... , 9.854500743565858, 10.0],
[0.9999999799999999, 0.7042848373025861,...,7.170748906965121e-13, -5.
↪1901263495547205e-12])
```

See also:

impulse_response_plot (page 1940)

Step-Response Plot

control_plots.**step_response_plot**(*color='b', prec=8, lower_limit=0, upper_limit=10, show_axes=False, grid=True, show=True, **kwargs*)

Returns the unit step response of a continuous-time system. It is the response of the system when the input signal is a step function.

Parameters

system : SISOLinearTimeInvariant type

The LTI SISO system for which the Step Response is to be computed.

color : str, tuple, optional

The color of the line. Default is Blue.

show : boolean, optional

If `True`, the plot will be displayed otherwise the equivalent matplotlib `plot` object will be returned. Defaults to True.

lower_limit : Number, optional

The lower limit of the plot range. Defaults to 0.

upper_limit : Number, optional

The upper limit of the plot range. Defaults to 10.

prec : int, optional

The decimal point precision for the point coordinate values. Defaults to 8.

show_axes : boolean, optional

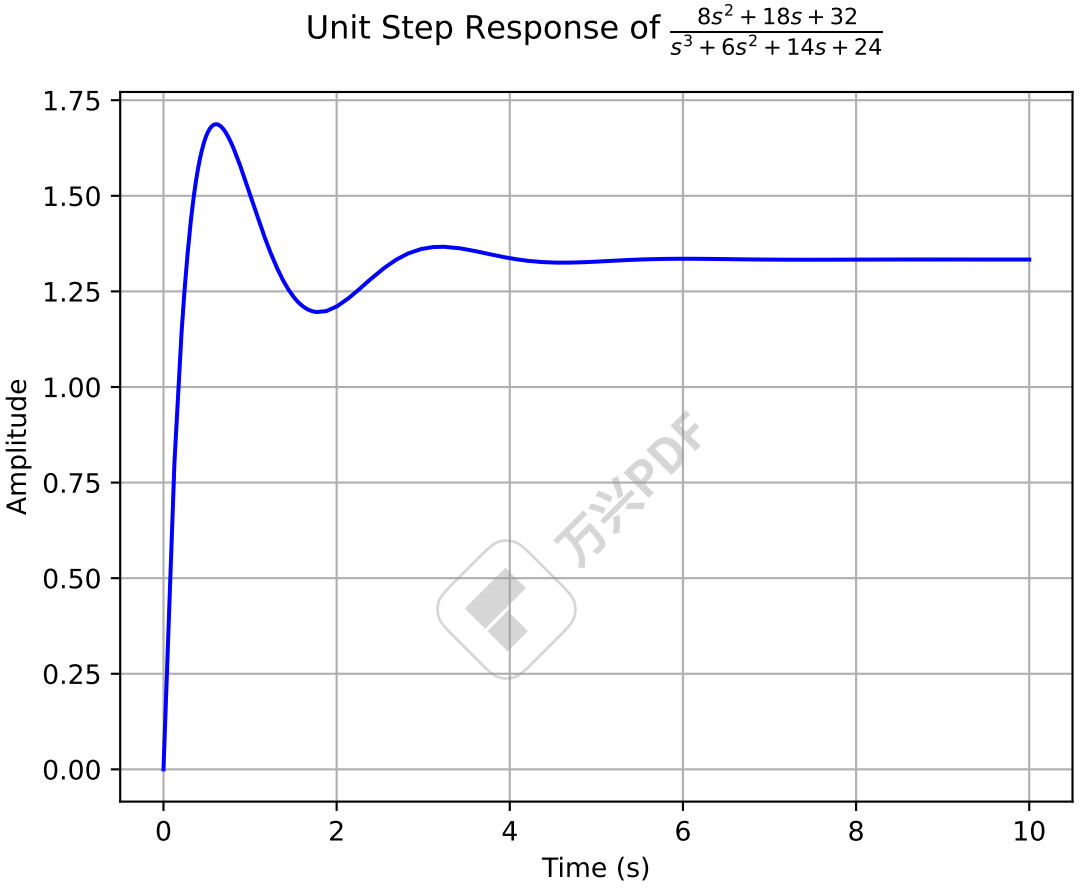
If `True`, the coordinate axes will be shown. Defaults to False.

grid : boolean, optional

If `True`, the plot will have a grid. Defaults to True.

Examples

```
>>> from sympy.abc import s
>>> from sympy.physics.control.lti import TransferFunction
>>> from sympy.physics.control.control_plots import step_response_plot
>>> tf1 = TransferFunction(8*s**2 + 18*s + 32, s**3 + 6*s**2 + 14*s + 24,
 ↪ s)
>>> step_response_plot(tf1)
```



See also:

impulse_response_plot (page 1940), *ramp_response_plot* (page 1946)

References

[R654]

control_plots.**step_response_numerical_data**(*prec=8*, *lower_limit=0*, *upper_limit=10*, ***kwargs*)

Returns the numerical values of the points in the step response plot of a SISO continuous-time system. By default, adaptive sampling is used. If the user wants to instead get an uniformly sampled response, then `adaptive` kwarg should be passed `False` and

`nb_of_points` must be passed as additional kwargs. Refer to the parameters of class *sympy.plotting.plot.LineOver1DRangeSeries* (page 2866) for more details.

Parameters

system : SISOLinearTimeInvariant

The system for which the unit step response data is to be computed.

prec : int, optional

The decimal point precision for the point coordinate values. Defaults to 8.

lower_limit : Number, optional

The lower limit of the plot range. Defaults to 0.

upper_limit : Number, optional

The upper limit of the plot range. Defaults to 10.

kwargs :

Additional keyword arguments are passed to the underlying *sympy.plotting.plot.LineOver1DRangeSeries* (page 2866) class.

Returns

tuple : (x, y)

x = Time-axis values of the points in the step response. NumPy array.
y = Amplitude-axis values of the points in the step response. NumPy array.

Raises

NotImplementedError

When a SISO LTI system is not passed.

When time delay terms are present in the system.

ValueError

When more than one free symbol is present in the system. The only variable in the transfer function should be the variable of the Laplace transform.

When `lower_limit` parameter is less than 0.

Examples

```
>>> from sympy.abc import s
>>> from sympy.physics.control.lti import TransferFunction
>>> from sympy.physics.control.control_plots import step_response_
 ↪numerical_data
>>> tf1 = TransferFunction(s, s**2 + 5*s + 8, s)
>>> step_response_numerical_data(tf1)
([0.0, 0.025413462339411542, 0.0484508722725343, ... , 9.670250533855183,
 ↪ 9.844291913708725, 10.0],
[0.0, 0.023844582399907256, 0.042894276802320226, ..., 6.
 ↪828770759094287e-12, 6.456457160755703e-12])
```

See also:

step_response_plot (page 1943)

Ramp-Response Plot

control_plots.**ramp_response_plot**(*slope=1*, *color='b'*, *prec=8*, *lower_limit=0*, *upper_limit=10*, *show_axes=False*, *grid=True*, *show=True*, ***kwargs*)

Returns the ramp response of a continuous-time system.

Ramp function is defined as the straight line passing through origin ($f(x) = mx$). The slope of the ramp function can be varied by the user and the default value is 1.

Parameters

system : SISOLinearTimeInvariant type

The LTI SISO system for which the Ramp Response is to be computed.

slope : Number, optional

The slope of the input ramp function. Defaults to 1.

color : str, tuple, optional

The color of the line. Default is Blue.

show : boolean, optional

If `True`, the plot will be displayed otherwise the equivalent matplotlib `plot` object will be returned. Defaults to True.

lower_limit : Number, optional

The lower limit of the plot range. Defaults to 0.

upper_limit : Number, optional

The upper limit of the plot range. Defaults to 10.

prec : int, optional

The decimal point precision for the point coordinate values. Defaults to 8.

show_axes : boolean, optional

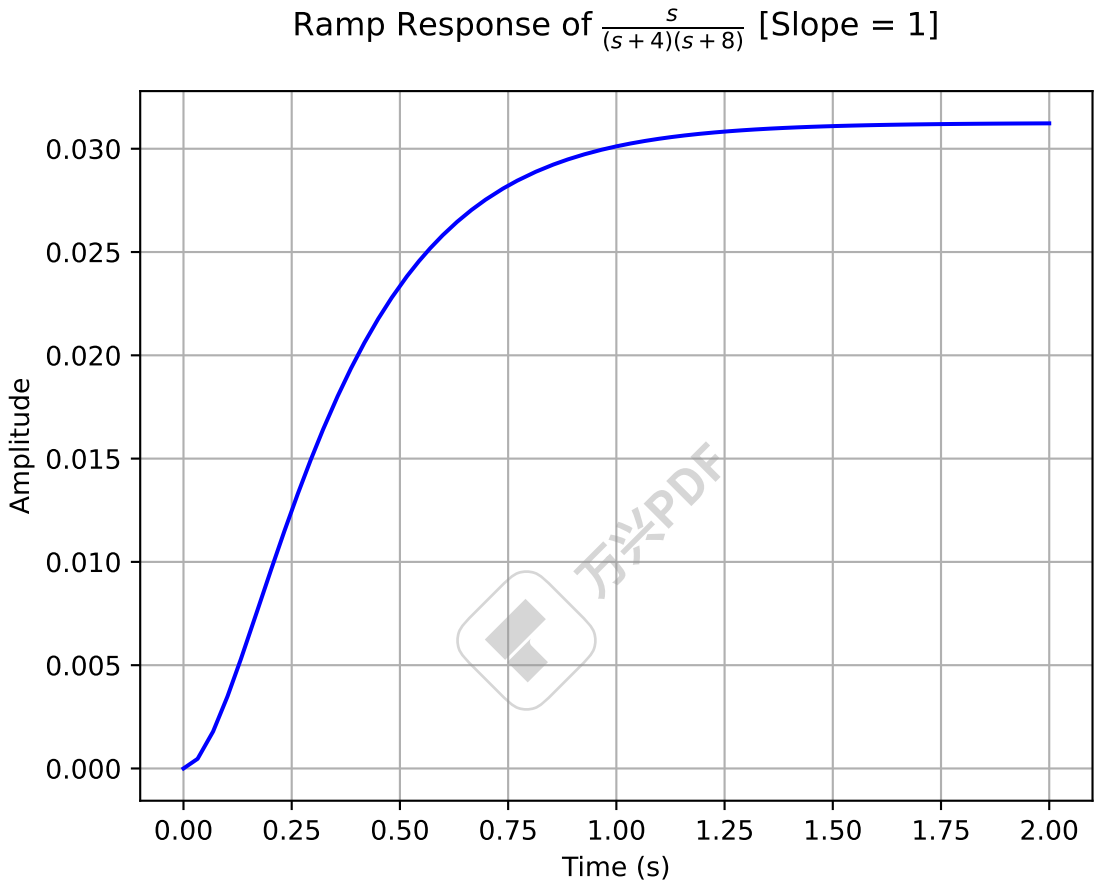
If `True`, the coordinate axes will be shown. Defaults to False.

grid : boolean, optional

If `True`, the plot will have a grid. Defaults to True.

Examples

```
>>> from sympy.abc import s
>>> from sympy.physics.control.lti import TransferFunction
>>> from sympy.physics.control.control_plots import ramp_response_plot
>>> tf1 = TransferFunction(s, (s+4)*(s+8), s)
>>> ramp_response_plot(tf1, upper_limit=2)
```



See also:

step_response_plot (page 1943), *ramp_response_plot* (page 1946)

References

[R655]

control_plots.**ramp_response_numerical_data**(*slope=1*, *prec=8*, *lower_limit=0*, *upper_limit=10*, ***kwargs*)

Returns the numerical values of the points in the ramp response plot of a SISO continuous-time system. By default, adaptive sampling is used. If the user wants to instead get an uniformly sampled response, then `adaptive` kwarg should be passed `False` and `nb_of_points` must be passed as additional kwargs. Refer to the parameters of class *sympy.plotting.plot.LineOver1DRangeSeries* (page 2866) for more details.

Parameters

system : SISOLinearTimeInvariant

The system for which the ramp response data is to be computed.

slope : Number, optional

The slope of the input ramp function. Defaults to 1.

prec : int, optional

The decimal point precision for the point coordinate values. Defaults to 8.

lower_limit : Number, optional

The lower limit of the plot range. Defaults to 0.

upper_limit : Number, optional

The upper limit of the plot range. Defaults to 10.

kwargs :

Additional keyword arguments are passed to the underlying *sympy.plotting.plot.LineOver1DRangeSeries* (page 2866) class.

Returns

tuple : (x, y)

x = Time-axis values of the points in the ramp response plot. NumPy array. y = Amplitude-axis values of the points in the ramp response plot. NumPy array.

Raises

NotImplementedError

When a SISO LTI system is not passed.

When time delay terms are present in the system.

ValueError

When more than one free symbol is present in the system. The only variable in the transfer function should be the variable of the Laplace transform.

When `lower_limit` parameter is less than 0.

When `slope` is negative.

Examples

```
>>> from sympy.abc import s
>>> from sympy.physics.control.lti import TransferFunction
>>> from sympy.physics.control.control_plots import ramp_response_
↪numerical_data
>>> tf1 = TransferFunction(s, s**2 + 5*s + 8, s)
>>> ramp_response_numerical_data(tf1)
(([0.0, 0.12166980856813935,..., 9.861246379582118, 10.0],
[1.4504508011325967e-09, 0.006046440489058766,..., 0.12499999999568202,␣
↪0.12499999999661349]))
```

See also:

ramp_response_plot (page 1946)

Continuum Mechanics

Abstract

Contains docstrings for methods in continuum mechanics module.

Beam

Beam (Docstrings)

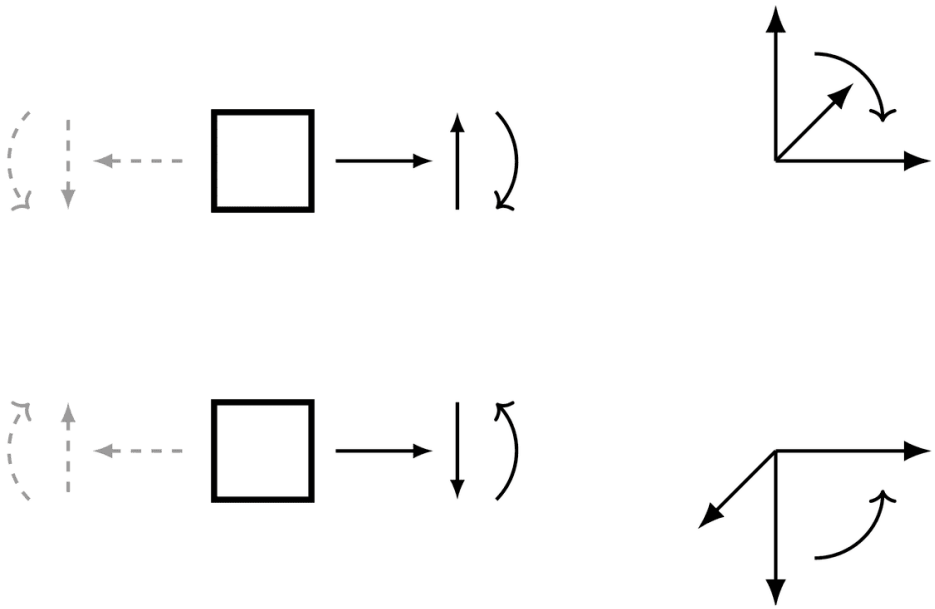
Beam

This module can be used to solve 2D beam bending problems with singularity functions in mechanics.

class sympy.physics.continuum_mechanics.beam.**Beam**(*length*, *elastic_modulus*, *second_moment*, *area=A*, *variable=x*, *base_char='C'*)

A Beam is a structural element that is capable of withstanding load primarily by resisting against bending. Beams are characterized by their cross sectional profile(Second moment of area), their length and their material.

Note: A consistent sign convention must be used while solving a beam bending problem; the results will automatically follow the chosen sign convention. However, the chosen sign convention must respect the rule that, on the positive side of beam's axis (in respect to current section), a loading force giving positive shear yields a negative moment, as below (the curved arrow shows the positive moment and rotation):



Examples

There is a beam of length 4 meters. A constant distributed load of 6 N/m is applied from half of the beam till the end. There are two simple supports below the beam, one at the starting point and another at the ending point of the beam. The deflection of the beam at the end is restricted.

Using the sign convention of downwards forces being positive.

```
>>> from sympy.physics.continuum_mechanics.beam import Beam
>>> from sympy import symbols, Piecewise
>>> E, I = symbols('E, I')
>>> R1, R2 = symbols('R1, R2')
>>> b = Beam(4, E, I)
>>> b.apply_load(R1, 0, -1)
>>> b.apply_load(6, 2, 0)
>>> b.apply_load(R2, 4, -1)
>>> b.bc_deflection = [(0, 0), (4, 0)]
>>> b.boundary_conditions
{'deflection': [(0, 0), (4, 0)], 'slope': []}
>>> b.load
R1*SingularityFunction(x, 0, -1) + R2*SingularityFunction(x, 4, -1) +␣
↪6*SingularityFunction(x, 2, 0)
>>> b.solve_for_reaction_loads(R1, R2)
```

(continues on next page)

(continued from previous page)

```
>>> b.load
-3*SingularityFunction(x, 0, -1) + 6*SingularityFunction(x, 2, 0) -␣
↪9*SingularityFunction(x, 4, -1)
>>> b.shear_force()
3*SingularityFunction(x, 0, 0) - 6*SingularityFunction(x, 2, 1) +␣
↪9*SingularityFunction(x, 4, 0)
>>> b.bending_moment()
3*SingularityFunction(x, 0, 1) - 3*SingularityFunction(x, 2, 2) +␣
↪9*SingularityFunction(x, 4, 1)
>>> b.slope()
(-3*SingularityFunction(x, 0, 2)/2 + SingularityFunction(x, 2, 3) -␣
↪9*SingularityFunction(x, 4, 2)/2 + 7)/(E*I)
>>> b.deflection()
(7*x - SingularityFunction(x, 0, 3)/2 + SingularityFunction(x, 2, 4)/4 -␣
↪3*SingularityFunction(x, 4, 3)/2)/(E*I)
>>> b.deflection().rewrite(Piecewise)
(7*x - Piecewise((x**3, x > 0), (0, True))/2
     - 3*Piecewise(((x - 4)**3, x > 4), (0, True))/2
     + Piecewise(((x - 2)**4, x > 2), (0, True))/4)/(E*I)
```

property applied_loads

Returns a list of all loads applied on the beam object. Each load in the list is a tuple of form (value, start, order, end).

Examples

There is a beam of length 4 meters. A moment of magnitude 3 Nm is applied in the clockwise direction at the starting point of the beam. A pointload of magnitude 4 N is applied from the top of the beam at 2 meters from the starting point. Another pointload of magnitude 5 N is applied at same position.

```
>>> from sympy.physics.continuum_mechanics.beam import Beam
>>> from sympy import symbols
>>> E, I = symbols('E, I')
>>> b = Beam(4, E, I)
>>> b.apply_load(-3, 0, -2)
>>> b.apply_load(4, 2, -1)
>>> b.apply_load(5, 2, -1)
>>> b.load
-3*SingularityFunction(x, 0, -2) + 9*SingularityFunction(x, 2, -1)
>>> b.applied_loads
[(-3, 0, -2, None), (4, 2, -1, None), (5, 2, -1, None)]
```

apply_load(*value, start, order, end=None*)

This method adds up the loads given to a particular beam object.

Parameters

value : Sympifyable

The value inserted should have the units [Force/(Distance**(n+1)] where n is the order of applied load. Units for applied loads:

- For moments, unit = kN*m

- For point loads, unit = kN
- For constant distributed load, unit = kN/m
- For ramp loads, unit = kN/m/m
- For parabolic ramp loads, unit = kN/m/m/m
- ... so on.

start : Sympifyable

The starting point of the applied load. For point moments and point forces this is the location of application.

order : Integer

The order of the applied load.

- For moments, order = -2
- For point loads, order =-1
- For constant distributed load, order = 0
- For ramp loads, order = 1
- For parabolic ramp loads, order = 2
- ... so on.

end : Sympifyable, optional

An optional argument that can be used if the load has an end point within the length of the beam.

Examples

There is a beam of length 4 meters. A moment of magnitude 3 Nm is applied in the clockwise direction at the starting point of the beam. A point load of magnitude 4 N is applied from the top of the beam at 2 meters from the starting point and a parabolic ramp load of magnitude 2 N/m is applied below the beam starting from 2 meters to 3 meters away from the starting point of the beam.

```
>>> from sympy.physics.continuum_mechanics.beam import Beam
>>> from sympy import symbols
>>> E, I = symbols('E, I')
>>> b = Beam(4, E, I)
>>> b.apply_load(-3, 0, -2)
>>> b.apply_load(4, 2, -1)
>>> b.apply_load(-2, 2, 2, end=3)
>>> b.load
-3*SingularityFunction(x, 0, -2) + 4*SingularityFunction(x, 2, -1) -␣
↪2*SingularityFunction(x, 2, 2) + 2*SingularityFunction(x, 3, 0) +␣
↪4*SingularityFunction(x, 3, 1) + 2*SingularityFunction(x, 3, 2)
```

apply_support(*loc*, *type='fixed'*)

This method applies support to a particular beam object.

Parameters

loc : Sympifyable

Location of point at which support is applied.

type : String

Determines type of Beam support applied. To apply support structure with - zero degree of freedom, type = "fixed" - one degree of freedom, type = "pin" - two degrees of freedom, type = "roller"

Examples

There is a beam of length 30 meters. A moment of magnitude 120 Nm is applied in the clockwise direction at the end of the beam. A pointload of magnitude 8 N is applied from the top of the beam at the starting point. There are two simple supports below the beam. One at the end and another one at a distance of 10 meters from the start. The deflection is restricted at both the supports.

Using the sign convention of upward forces and clockwise moment being positive.

```
>>> from sympy.physics.continuum_mechanics.beam import Beam
>>> from sympy import symbols
>>> E, I = symbols('E, I')
>>> b = Beam(30, E, I)
>>> b.apply_support(10, 'roller')
>>> b.apply_support(30, 'roller')
>>> b.apply_load(-8, 0, -1)
>>> b.apply_load(120, 30, -2)
>>> R_10, R_30 = symbols('R_10, R_30')
>>> b.solve_for_reaction_loads(R_10, R_30)
>>> b.load
-8*SingularityFunction(x, 0, -1) + 6*SingularityFunction(x, 10, -1)
+ 120*SingularityFunction(x, 30, -2) + 2*SingularityFunction(x, 30, -
 ↪1)
>>> b.slope()
(-4*SingularityFunction(x, 0, 2) + 3*SingularityFunction(x, 10, 2)
    + 120*SingularityFunction(x, 30, 1) + SingularityFunction(x, 30,␣
 ↪2) + 4000/3)/(E*I)
```

property area

Cross-sectional area of the Beam.

bending_moment()

Returns a Singularity Function expression which represents the bending moment curve of the Beam object.

Examples

There is a beam of length 30 meters. A moment of magnitude 120 Nm is applied in the clockwise direction at the end of the beam. A pointload of magnitude 8 N is applied from the top of the beam at the starting point. There are two simple supports below the beam. One at the end and another one at a distance of 10 meters from the start. The deflection is restricted at both the supports.

Using the sign convention of upward forces and clockwise moment being positive.

```
>>> from sympy.physics.continuum_mechanics.beam import Beam
>>> from sympy import symbols
>>> E, I = symbols('E, I')
>>> R1, R2 = symbols('R1, R2')
>>> b = Beam(30, E, I)
>>> b.apply_load(-8, 0, -1)
>>> b.apply_load(R1, 10, -1)
>>> b.apply_load(R2, 30, -1)
>>> b.apply_load(120, 30, -2)
>>> b.bc_deflection = [(10, 0), (30, 0)]
>>> b.solve_for_reaction_loads(R1, R2)
>>> b.bending_moment()
8*SingularityFunction(x, 0, 1) - 6*SingularityFunction(x, 10, 1) -␣
↪120*SingularityFunction(x, 30, 0) - 2*SingularityFunction(x, 30, 1)
```

property boundary_conditions

Returns a dictionary of boundary conditions applied on the beam. The dictionary has three keywords namely moment, slope and deflection. The value of each keyword is a list of tuple, where each tuple contains location and value of a boundary condition in the format (location, value).

Examples

There is a beam of length 4 meters. The bending moment at 0 should be 4 and at 4 it should be 0. The slope of the beam should be 1 at 0. The deflection should be 2 at 0.

```
>>> from sympy.physics.continuum_mechanics.beam import Beam
>>> from sympy import symbols
>>> E, I = symbols('E, I')
>>> b = Beam(4, E, I)
>>> b.bc_deflection = [(0, 2)]
>>> b.bc_slope = [(0, 1)]
>>> b.boundary_conditions
{'deflection': [(0, 2)], 'slope': [(0, 1)]}
```

Here the deflection of the beam should be `2` at `0`. Similarly, the slope of the beam should be `1` at `0`.

property cross_section

Cross-section of the beam

deflection()

Returns a Singularity Function expression which represents the elastic curve or deflection of the Beam object.

Examples

There is a beam of length 30 meters. A moment of magnitude 120 Nm is applied in the clockwise direction at the end of the beam. A pointload of magnitude 8 N is applied from the top of the beam at the starting point. There are two simple supports below the beam. One at the end and another one at a distance of 10 meters from the start. The deflection is restricted at both the supports.

Using the sign convention of upward forces and clockwise moment being positive.

```
>>> from sympy.physics.continuum_mechanics.beam import Beam
>>> from sympy import symbols
>>> E, I = symbols('E, I')
>>> R1, R2 = symbols('R1, R2')
>>> b = Beam(30, E, I)
>>> b.apply_load(-8, 0, -1)
>>> b.apply_load(R1, 10, -1)
>>> b.apply_load(R2, 30, -1)
>>> b.apply_load(120, 30, -2)
>>> b.bc_deflection = [(10, 0), (30, 0)]
>>> b.solve_for_reaction_loads(R1, R2)
>>> b.deflection()
(4000*x/3 - 4*SingularityFunction(x, 0, 3)/3 + SingularityFunction(x,␣
↪10, 3)
    + 60*SingularityFunction(x, 30, 2) + SingularityFunction(x, 30,␣
↪3)/3 - 12000)/(E*I)
```

draw(*pictorial=True*)

Returns a plot object representing the beam diagram of the beam.

Note: The user must be careful while entering load values. The draw function assumes a sign convention which is used for plotting loads. Given a right handed coordinate system with XYZ coordinates, the beam's length is assumed to be along the positive X axis. The draw function recognizes positve loads(with n>-2) as loads acting along negative Y direction and positve moments acting along positive Z direction.

Parameters

pictorial: Boolean (default=True)

Setting `pictorial=True` would simply create a pictorial (scaled) view of the beam diagram not with the exact dimensions. Although setting `pictorial=False` would create a beam diagram with the exact dimensions on the plot

Examples

```
>>> from sympy.physics.continuum_mechanics.beam import Beam
>>> from sympy import symbols
>>> R1, R2 = symbols('R1, R2')
>>> E, I = symbols('E, I')
>>> b = Beam(50, 20, 30)
>>> b.apply_load(10, 2, -1)
>>> b.apply_load(R1, 10, -1)
>>> b.apply_load(R2, 30, -1)
>>> b.apply_load(90, 5, 0, 23)
>>> b.apply_load(10, 30, 1, 50)
>>> b.apply_support(50, "pin")
>>> b.apply_support(0, "fixed")
>>> b.apply_support(20, "roller")
>>> p = b.draw()
>>> p
Plot object containing:
[0]: cartesian line: 25*SingularityFunction(x, 5, 0) -␣
↪25*SingularityFunction(x, 23, 0)
+ SingularityFunction(x, 30, 1) - 20*SingularityFunction(x, 50, 0)
- SingularityFunction(x, 50, 1) + 5 for x over (0.0, 50.0)
[1]: cartesian line: 5 for x over (0.0, 50.0)
>>> p.show()
```

property elastic_modulus

Young's Modulus of the Beam.

property ild_moment

Returns the I.L.D. moment equation.

property ild_reactions

Returns the I.L.D. reaction forces in a dictionary.

property ild_shear

Returns the I.L.D. shear equation.

join(*beam*, *via='fixed'*)

This method joins two beams to make a new composite beam system. Passed Beam class instance is attached to the right end of calling object. This method can be used to form beams having Discontinuous values of Elastic modulus or Second moment.

Parameters

beam : Beam class object

The Beam object which would be connected to the right of calling object.

via : String

States the way two Beam object would get connected - For axially fixed Beams, via="fixed" - For Beams connected via hinge, via="hinge"