desr Documentation

Release 0

Richard Tanburn

Oct 19, 2019

Contents

1 Introduction **3**
- 1.1 Prerequisites . . . 3
- 1.2 Installing . . . 3
- 1.3 Running the tests . . . 3
- 1.4 Built With . . . 3
- 1.5 Contributing . . . 4
- 1.6 Authors . . . 4
- 1.7 License . . . 4
- 1.8 Acknowledgments . . . 4

2 Hermite and Smith Normal Forms **5**
- 2.1 Hermite Normal Forms . . . 5
- 2.2 Smith Normal Form . . . 7

3 Systems of ODEs **9**
- 3.1 Creating ODESystems . . . 9
- 3.2 Finding Scaling Actions . . . 11
- 3.3 Output Functions . . . 12

4 ODETranslation and Reduction of ODESystems **15**
- 4.1 Creating ODE Translations . . . 15
- 4.2 Reduction Methods . . . 16
- 4.3 Reverse Translation . . . 17
- 4.4 Extending from a set of Invariants . . . 20
- 4.5 Useful Attributes . . . 21
- 4.6 Output Functions . . . 22
- 4.7 Advanced Methods . . . 22

5 Chemical Reaction Networks **25**
- 5.1 Example Use . . . 25
- 5.2 Diagrams . . . 26

6 Examples **27**
- 6.1 Example Michaelis-Menten . . . 27
- 6.2 Walkthroughs from Supplementary Information . . . 34

7 desr Reference **37**

7.1 Submodules . . . 37
7.2 desr.ode_system module . . . 37
7.3 desr.ode_translation module . . . 46
7.4 desr.matrix_normal_forms module . . . 62
7.5 desr.diophantine module . . . 68
7.6 desr.chemical_reaction_network module . . . 69
7.7 desr.sympy_helper module . . . 71
7.8 desr.tex_tools module . . . 73
7.9 desr.unittests module . . . 75

8 Bibliography **77**

9 Indices and tables **79**

Bibliography **81**

Python Module Index **83**

Index **85**

Contents:

CHAPTER 1

Introduction

desr is a package used for Differental Equation Symmetry Reduction and is particularly useful for reducing the number of parameters in dynamical systems. It implements algorithms outlined by Evelyne Hubert and George Labahn [HL13].

The Masters dissertation Differential Algebra and Applications that inspired this project places the algorithms into the theoretical framework of *differential algebraic geometry* and shows how to extend them to parameter reduction of arbitrary systems of partial differential equations, though this is not yet implemented.

1.1 Prerequisites

This package requires the Sympy package.

1.2 Installing

To install, download the package and run:

```
python setup.py install
```

1.3 Running the tests

Doctests are included in most files. To run them, simply run the module. E.g. "python -m doctest -v module.py"

1.4 Built With

- Sphinx - Used to generate the docs.

1.5 Contributing

Submissions for contribution are always welcome.

1.6 Authors

- **Richard Tanburn** - *Initial work*

1.7 License

This project is licensed under the Apache 2.0 License.

1.8 Acknowledgments

- Dr Heather Harrington and Dr Emilie Dufresne for their supervision of the dissertation.
- Thomas Close for writing his diophantine module, which is included in this package.

CHAPTER 2

Hermite and Smith Normal Forms

desr uses the diophantine package, which in turn uses the methods found in [HMM98], to calculate the Hermite normal form of matrices.

matrix_normal_forms should be used for all normal form calculations - we never call the diophantine package directly from other parts of desr.

This is also where the Smith normal form functions live, which use Hermite normal forms at their core.

2.1 Hermite Normal Forms

desr.matrix_normal_forms.**is_hnf_row**(*matrix_*)

Decide whether a matrix is in Hermite normal form, when acting on rows. This is according to the Havas Majewski Matthews definition.

Parameters **matrix** (*sympy.Matrix*) – The matrix in question.

Returns bool

```
>>> is_hnf_row(sympy.eye(4))
True
>>> is_hnf_row(sympy.ones(2))
False
>>> is_hnf_row(sympy.Matrix([[1, 2, 0], [0, 1, 0]]))
False
>>> is_hnf_row(sympy.Matrix([[1, 0, 0], [0, 2, 1]]))
True
>>> is_hnf_row(sympy.Matrix([[1, 0, 0], [0, -2, 1]]))
False
```

desr.matrix_normal_forms.**is_hnf_col**(*matrix_*)

Decide whether a matrix is in row Hermite normal form, when acting on rows.

Parameters **matrix** (*sympy.Matrix*) – The matrix in question.

Returns bool

desr.matrix_normal_forms.**hnf_row**(*matrix_*)
Default function for calculating row Hermite normal forms.

Parameters **matrix** (*sympy.Matrix*) – Input matrix.

Returns

hermite_normal_form (sympy.Matrix): The column Hermite normal form of matrix_.

normal_multiplier (sympy.Matrix): The normal Hermite multiplier.

Return type tuple

Return type (sympy.Matrix, sympy.Matrix)

desr.matrix_normal_forms.**hnf_col**(*matrix_*)
Default function for calculating column Hermite normal forms.

Parameters **matrix** (*sympy.Matrix*) – Input matrix.

Returns

Tuple containing: hermite_normal_form (sympy.Matrix): The column Hermite normal form of matrix_.

normal_multiplier (sympy.Matrix): The normal Hermite multiplier.

Return type tuple

Return type (sympy.Matrix, sympy.Matrix)

desr.matrix_normal_forms.**normal_hnf_row**(*matrix_*)
Return the row HNF and the unique normal Hermite multiplier.

Parameters **matrix** (*sympy.Matrix*) – Input matrix.

Returns

Tuple containing:

hermite_normal_form (sympy.Matrix): The row Hermite normal form of matrix_. normal_multiplier (sympy.Matrix): The normal Hermite multiplier.

Return type tuple

desr.matrix_normal_forms.**normal_hnf_col**(*matrix_*)
Return the column HNF and the unique normal Hermite multiplier.

Parameters **matrix** (*sympy.Matrix*) – Input matrix.

Returns

Tuple containing: hermite_normal_form (sympy.Matrix): The column Hermite normal form of matrix_. normal_multiplier (sympy.Matrix): The normal Hermite multiplier.

Return type tuple

```
>>> A = sympy.Matrix([[8, 2, 15, 9, 11],
...                   [6, 0, 6, 2, 3]])
>>> h, v = normal_hnf_col(A)
>>> h
Matrix([
[1, 0, 0, 0, 0],
[0, 1, 0, 0, 0]])
>>> v
Matrix([
```

(continues on next page)

(continued from previous page)

```
[  0,  0,   1,  0,   0],
[  0,  0,   0,  1,   0],
[  2,  1,   3,  1,   5],
[  9,  2,  21,  3,  21],
[-10, -3, -22, -4, -24]])
>>> (A * v) == h
True
```

2.2 Smith Normal Form

desr.matrix_normal_forms.**is_smf**(*matrix_*)

Given a rectangular $n \times m$ integer matrix, determine whether it is in Smith normal form or not.

Parameters **matrix** (*sympy.Matrix*) – The rectangular matrix to be decomposed

Returns True if in Smith normal form, False otherwise.

Return type bool

```
>>> matrix_ = sympy.diag(1, 1, 2)
>>> is_smf(matrix_)
True
>>> matrix_ = sympy.diag(-1, 1, 2)
>>> is_smf(matrix_)
False
>>> matrix_ = sympy.diag(2, 1, 1)
>>> is_smf(matrix_)
False
>>> matrix_ = sympy.diag(1, 2, 0)
>>> is_smf(matrix_)
True
>>> matrix_ = sympy.diag(2, 6, 0)
>>> is_smf(matrix_)
True
>>> matrix_ = sympy.diag(2, 5, 0)
>>> is_smf(matrix_)
False
>>> matrix_ = sympy.diag(0, 1, 1)
>>> is_smf(matrix_)
False
>>> matrix_ = sympy.diag(0)
>>> is_smf(sympy.diag(0)), is_smf(sympy.diag(1)), is_smf(sympy.Matrix()),
(True, True, True)
```

Check a real example:

```
>>> matrix_ = sympy.Matrix([[2, 4, 4],
...                         [-6, 6, 12],
...                         [10, -4, -16]])
>>> is_smf(matrix_)
False
```

```
>>> matrix_ = sympy.diag(2, 6, 12)
>>> is_smf(matrix_)
True
```

Check it works for non-square matrices:

```
>>> matrix_ = sympy.Matrix(4, 5, range(20))
>>> is_smf(matrix_)
False
```

```
>>> matrix_ = sympy.Matrix([[1, 0], [1, 2]])
>>> is_smf(matrix_)
False
```

desr.matrix_normal_forms.**smf**(*matrix_*)

Given a rectangular $n \times m$ integer matrix, calculate the Smith Normal Form S and multipliers U, V such that U

Parameters

matrix (*sympy.Matrix*) – The rectangular matrix to be decomposed.

Returns

- **S** (*sympy.Matrix*) – The Smith normal form of matrix_.
- **U** (*sympy.Matrix*) – U (the matrix representing the row operations of the decomposition).
- **V** (*sympy.Matrix*) – V (the matrix representing the column operations of the decomposition).

Return type

(sympy.Matrix, sympy.Matrix, sympy.Matrix)

```
>>> matrix_ = sympy.Matrix([[2, 4, 4],
...                         [-6, 6, 12],
...                         [10, -4, -16]])
>>> smf(matrix_)[0]
Matrix([
[2, 0,  0],
[0, 6,  0],
[0, 0, 12]])
```

```
>>> matrix_ = sympy.diag(2, 1, 0)
>>> smf(matrix_)[0]
Matrix([
[1, 0, 0],
[0, 2, 0],
[0, 0, 0]])
```

```
>>> matrix_ = sympy.diag(5, 2, 0)
>>> smf(matrix_)[0]
Matrix([
[1,  0, 0],
[0, 10, 0],
[0,  0, 0]])
```

CHAPTER 3

Systems of ODEs

The *ODESystem* class is what we use to represent our system of ordinary differential equations.

3.1 Creating ODESystems

There are a number of different ways of creating an *ODESystem*.

class desr.ode_system.**ODESystem**(*variables*, *derivatives*, *indep_var=None*, *initial_conditions=None*)

A system of differential equations.

The main attributes are *variables* and *derivatives*. *variables* is an ordered tuple of variables, which includes the independent variable. *derivatives* is an ordered tuple of the same length that contains the derivatives with respect to *indep_var*.

Parameters

- **variables** (*tuple of sympy.Symbol*) – Ordered tuple of variables.
- **derivatives** (*tuple of sympy.Expression*) – Ordered tuple of derivatives.
- **indep_var** (*sympy.Symbol, optional*) – Independent variable we are differentiating with respect to.
- **initial_conditions** (*tuple of sympy.Symbol*) – The initial values of non-constant variables

classmethod ODESystem.**from_equations**(*equations*, *indep_var=t*, *initial_conditions=None*)

Instantiate from multiple equations.

Parameters

- **equations** (*str, iter of str*) – Equations of the form "dx/dt = expr", optionally seperated by `\n`.
- **indep_var** (*sympy.Symbol*) – The independent variable, usually `t`.

- **initial_conditions** (*tuple of sympy.Symbol*) – The initial values of non-constant variables

Returns System of equations.

Return type *ODESystem*

```
>>> eqns = ['dx/dt = c_0*x*y', 'dy/dt = c_1*(1-x)*(1-y)']
>>> ODESystem.from_equations(eqns)
dt/dt = 1
dx/dt = c_0*x*y
dy/dt = c_1*(-x + 1)*(-y + 1)
dc_0/dt = 0
dc_1/dt = 0
>>> eqns = '\n'.join(['dy/dx = c_0*x*y', 'dz/dx = c_1*(1-y)*z**2'])
>>> ODESystem.from_equations(eqns, indep_var=sympy.Symbol('x'))
dx/dx = 1
dy/dx = c_0*x*y
dz/dx = c_1*z**2*(-y + 1)
dc_0/dx = 0
dc_1/dx = 0
```

classmethod ODESystem.**from_dict**(*deriv_dict*, *indep_var=t*, *initial_conditions=None*)

Instantiate from a text of equations.

Parameters

- **deriv_dict** (*dict*) – {variable: derivative} mapping.
- **indep_var** (*sympy.Symbol*) – Independent variable, that the derivatives are with respect to.
- **initial_conditions** (*tuple of sympy.Symbol*) – The initial values of non-constant variables

Returns System of ODEs.

Return type *ODESystem*

```
>>> _input = {'x': 'c_0*x*y', 'y': 'c_1*(1-x)*(1-y)'}
>>> _input = {sympy.Symbol(k): sympy.sympify(v) for k, v in _input.iteritems()}
>>> ODESystem.from_dict(_input)
dt/dt = 1
dx/dt = c_0*x*y
dy/dt = c_1*(-x + 1)*(-y + 1)
dc_0/dt = 0
dc_1/dt = 0
```

```
>>> _input = {'y':  'c_0*x*y', 'z': 'c_1*(1-y)*z**2'}
>>> _input = {sympy.Symbol(k): sympy.sympify(v) for k, v in _input.iteritems()}
>>> ODESystem.from_dict(_input, indep_var=sympy.Symbol('x'))
dx/dx = 1
dy/dx = c_0*x*y
dz/dx = c_1*z**2*(-y + 1)
dc_0/dx = 0
dc_1/dx = 0
```

classmethod ODESystem.**from_tex**(*tex*)

Given the LaTeX of a system of differential equations, return a ODESystem of it.

Parameters **tex** (*str*) – LaTeX

Returns System of ODEs.

Return type *ODESystem*

```
>>> eqns = ['\frac{dE}{dt} &= - k_1 E S + k_{-1} C + k_2 C \\',
... '\frac{dS}{dt} &= - k_1 E S + k_{-1} C \\',
... '\frac{dC}{dt} &= k_1 E S - k_{-1} C - k_2 C \\',
... '\frac{dP}{dt} &= k_2 C']
>>> ODESystem.from_tex('\n'.join(eqns))
dt/dt = 1
dC/dt = -C*k_2 - C*k_m1 + E*S*k_1
dE/dt = C*k_2 + C*k_m1 - E*S*k_1
dP/dt = C*k_2
dS/dt = C*k_m1 - E*S*k_1
dk_1/dt = 0
dk_2/dt = 0
dk_m1/dt = 0
```

Todo:

- Allow initial conditions to be set from tex.

3.2 Finding Scaling Actions

ODESystem.**power_matrix**()

Determine the 'exponent' or 'power' matrix of the system, denoted by K in the literature, by gluing together the power matrices of each derivative.

In particular, it concatenates $K_{\left(\frac{t}{x}\cdot\frac{dx}{dt}\right)}$ for x in *variables*, where t is the independent variable.

```
>>> eqns = '\n'.join(['ds/dt = -k_1*e_0*s + (k_1*s + k_m1)*c',
... 'dc/dt = k_1*e_0*s - (k_1*s + k_m1 + k_2)*c'])
>>> system = ODESystem.from_equations(eqns)
>>> system.variables
(t, c, s, e_0, k_1, k_2, k_m1)
>>> system.power_matrix()
Matrix([
[1, 1, 1,  1, 1, 1,  1],
[0, 0, 0, -1, 0, 1,  1],
[1, 0, 0,  1, 0, 0, -1],
[0, 0, 0,  1, 1, 0,  0],
[1, 0, 0,  1, 1, 1,  0],
[0, 1, 0,  0, 0, 0,  0],
[0, 0, 1,  0, 0, 0,  1]])
```

While we get a different answer to the example in the paper, this is just due to choosing our reference exponent in a different way.

Todo:

- Change the code to agree with the paper.

```
>>> system.update_initial_conditions({'s': 's_0'})
>>> system.power_matrix()
Matrix([
[1, 1, 1,  1, 1, 1,  1,  0],
[0, 0, 0, -1, 0, 1,  1,  0],
[1, 0, 0,  1, 0, 0, -1,  1],
[0, 0, 0,  1, 1, 0,  0,  0],
[1, 0, 0,  1, 1, 1,  0,  0],
[0, 1, 0,  0, 0, 0,  0,  0],
[0, 0, 1,  0, 0, 0,  1,  0],
[0, 0, 0,  0, 0, 0,  0, -1]])
```

ODESystem.**maximal_scaling_matrix**()

Determine the maximal scaling matrix leaving this system invariant.

Returns Maximal scaling matrix.

Return type sympy.Matrix

```
>>> eqns = '\n'.join(['ds/dt = -k_1*e_0*s + (k_1*s + k_m1)*c',
... 'dc/dt = k_1*e_0*s - (k_1*s + k_m1 + k_2)*c'])
>>> system = ODESystem.from_equations(eqns)
>>> system.maximal_scaling_matrix()
Matrix([
[1, 0, 0, 0, -1, -1, -1],
[0, 1, 1, 1, -1,  0,  0]])
```

3.3 Output Functions

There are a number of useful ways to output the system.

ODESystem.**derivative_dict**

expr mapping, filtering out the None's in expr.

Returns Keys are non-constant variables, value is the derivative with respect to the independent variable.

Return type dict

Type Return a variable

ODESystem.**to_tex**()

Returns TeX representation.

Return type str

```
>>> eqns = ['dC/dt = -C*k_2 - C*k_m1 + E*S*k_1',
... 'dE/dt = C*k_2 + C*k_m1 - E*S*k_1',
... 'dP/dt = C*k_2',
... 'dS/dt = C*k_m1 - E*S*k_1']
>>> system = ODESystem.from_equations('\n'.join(eqns))
>>> print system.to_tex()
\frac{dt}{dt} &= 1 \\
\frac{dC}{dt} &= - C k_{2} - C k_{-1} + E S k_{1} \\
\frac{dE}{dt} &= C k_{2} + C k_{-1} - E S k_{1} \\
\frac{dP}{dt} &= C k_{2} \\
\frac{dS}{dt} &= C k_{-1} - E S k_{1} \\
```

(continues on next page)

(continued from previous page)

```
\frac{dk_{1}}{dt} &= 0 \\
\frac{dk_{2}}{dt} &= 0 \\
\frac{dk_{-1}}{dt} &= 0
```

```
>>> system.update_initial_conditions({'C': 'C_0'})
>>> print system.to_tex()
\frac{dt}{dt} &= 1 \\
\frac{dC}{dt} &= - C k_{2} - C k_{-1} + E S k_{1} \\
\frac{dE}{dt} &= C k_{2} + C k_{-1} - E S k_{1} \\
\frac{dP}{dt} &= C k_{2} \\
\frac{dS}{dt} &= C k_{-1} - E S k_{1} \\
\frac{dk_{1}}{dt} &= 0 \\
\frac{dk_{2}}{dt} &= 0 \\
\frac{dk_{-1}}{dt} &= 0 \\
\frac{dC_{0}}{dt} &= 0 \\
C\left(0\right) &= C_{0}
```

```
>>> system.add_constraints('K_m', '(k_m1 + k_2) / k_1')
>>> print system.to_tex()
\frac{dt}{dt} &= 1 \\
\frac{dC}{dt} &= - C k_{2} - C k_{-1} + E S k_{1} \\
\frac{dE}{dt} &= C k_{2} + C k_{-1} - E S k_{1} \\
\frac{dP}{dt} &= C k_{2} \\
\frac{dS}{dt} &= C k_{-1} - E S k_{1} \\
\frac{dk_{1}}{dt} &= 0 \\
\frac{dk_{2}}{dt} &= 0 \\
\frac{dk_{-1}}{dt} &= 0 \\
\frac{dC_{0}}{dt} &= 0 \\
\frac{dK_{m}}{dt} &= 0 \\
C\left(0\right) &= C_{0} \\
K_{m} &= \frac{1}{k_{1}} \left(k_{2} + k_{-1}\right)
```

CHAPTER 4

ODETranslation and Reduction of ODESystems

The *ODETranslation* is the class used to represent the scaling symmetries of a system.

4.1 Creating ODE Translations

class desr.ode_translation.**ODETranslation**(*scaling_matrix*, *variables_domain=None*, *hermite_multiplier=None*)

An object used for translating between systems of ODEs according to a scaling matrix. The key data are *scaling_matrix* and *herm_mult*, which together contain all the information needed (along with a variable order) to reduce an *ODESystem*.

Parameters

- **scaling_matrix** (*sympy.Matrix*) – Matrix that defines the torus action on the system.
- **variables_domain** (*iter of sympy.Symbol, optional*) – An ordering of the variables we expect to act upon. If this is not given, we will act on a system according to the position of the variables in *variables*, as long as there is the correct number of variables.
- **hermite_multiplier** (*sympy.Matrix, optional*) – User-defined Hermite multiplier, that puts *scaling_matrix* into column Hermite normal form. If not given, the normal Hermite multiplier will be calculated.

classmethod ODETranslation.**from_ode_system**(*ode_system*)

Create a *ODETranslation* given an ode_system.ODESystem instance, by taking the maximal scaling matrix.

Parameters **ode_system** (ODESystem) –

Return type *ODETranslation*

4.2 Reduction Methods

ODETranslation.**translate**(*system*)
Translate an ode_system.ODESystem into a reduced system.

First, try the simplest parameter reduction method, then the dependent variable translation (where the scaling action ignores the independent variable) and finally the general reduction scheme.

Parameters **system** (ODESystem) – System to reduce.

Return type *ODESystem*

ODETranslation.**translate_parameter**(*system*)
Translate according to parameter scheme

ODETranslation.**translate_dep_var**(*system*)
Given a system of ODEs, translate the system into a simplified version. Assume we are only working on dependent variables, not the independent variable.

Parameters **system** (ODESystem) – System to reduce.

Return type *ODESystem*

```
>>> equations = 'dz1/dt = z1*(1+z1*z2);dz2/dt = z2*(1/t - z1*z2)'.split(';')
>>> system = ODESystem.from_equations(equations)
>>> translation = ODETranslation.from_ode_system(system)
>>> translation.translate_dep_var(system=system)
dt/dt = 1
dx0/dt = x0*(y0 - 1/t)
dy0/dt = y0*(1 + 1/t)
```

ODETranslation.**translate_general**(*system*)
The most general reduction scheme. If there are n variables (including the independent variable) then there will be a system of $n - r + 1$ invariants and r auxiliary variables.

Parameters **system** (ODESystem) – System to reduce.

Return type *ODESystem*

```
>>> equations = 'dn/dt = n*( r*(1 - n/K) - k*p/(n+d) );dp/dt = s*p*(1 - h*p / n)'.
↪split(';')
>>> system = ODESystem.from_equations(equations)
>>> translation = ODETranslation.from_ode_system(system)
>>> translation.translate_general(system=system)
dt/dt = 1
dx0/dt = 0
dx1/dt = 0
dx2/dt = 0
dy0/dt = y0/t
dy1/dt = y1*(-y0*y1*y5/y3 - y0*y2/(y1 + 1) + y0*y5)/t
dy2/dt = y2*(y0 - y0*y2*y4/y1)/t
dy3/dt = 0
dy4/dt = 0
dy5/dt = 0
```

4.3 Reverse Translation

Reverse translation is the process of taking solutions of the reduced system and recovering solutions of the original system.

ODETranslation.**reverse_translate**(*variables*)

Given the solutions of a reduced system, reverse translate them into solutions of the original system.

Note: This *doesn't* reverse translate the parameter reduction scheme.

It *only* guesses between *reverse_translate_general()* and *reverse_translate_dep_var()*

Parameters

- **variables** (*iter of sympy.Expression*) – The solution auxiliary variables $x(t)$ and solution invariants $y(t)$ of the reduced system. Variables should be ordered auxiliary variables followed by invariants.
- **indep_var_index** (*int*) – The location of the independent variable.

Return type tuple

ODETranslation.**reverse_translate_parameter**(*variables*)

Given the solutions of a reduced system, reverse translate them into solutions of the original system.

Parameters **variables** (*iter of sympy.Expression*) – r constants (auxiliary variables) followed by the independent, dependent and invariant constants.

Example 7.5 (Prey-predator model) from [HL13].

Use the matrices from the paper, which differ to ours as different conventions are used.

$$\frac{dn}{dt} = n\left(r\left(1-\frac{n}{K}\right)-\frac{kp}{n+d}\right)$$
$$\frac{dp}{dt} = sp\left(1-\frac{hp}{n}\right)$$

```
>>> equations = ['dn/dt = n*( r*(1 - n/K) - k*p/(n+d) )','dp/dt = s*p*(1 - h*p /␣
↪n)']
>>> system = ODESystem.from_equations(equations)
>>> system.variables
(t, n, p, K, d, h, k, r, s)
>>> maximal_scaling_matrix = system.maximal_scaling_matrix()
>>> # Hand craft a Hermite multiplier given the invariants from the paper.
>>> # This is done by placing the auxiliaries first, then rearranging the␣
↪invariants and reading them off
>>> # individually from the Hermite multiplier given in the paper
>>> herm_mult = sympy.Matrix([[ 0,  0,  0,  1,  0,  0,  0,  0,  0],  # t
...                           [ 0,  0,  0,  0,  1,  0,  0,  0,  0],  # n
...                           [ 0,  0,  0,  0,  0,  1,  0,  0,  0],  # p
...                           [ 0,  1,  1,  0, -1, -1, -1,  0,  0],  # K
...                           [ 0,  0,  0,  0,  0,  0,  1,  0,  0],  # d
...                           [ 0,  0, -1,  0,  0,  1,  0, -1,  0],  # h
...                           [ 0,  0,  0,  0,  0,  0,  0,  1,  0],  # k
...                           [-1,  0,  0,  1,  0,  0,  0, -1, -1],  # r
...                           [ 0,  0,  0,  0,  0,  0,  0,  0,  1]])  # s
>>> translation = ODETranslation(maximal_scaling_matrix,
```

(continues on next page)

(continued from previous page)

```
...                                 variables_domain=system.variables,
...                                 hermite_multiplier=herm_mult)
>>> translation.translate_parameter(system)
dt/dt = 1
dn/dt = -c1*n*p/(c0 + n) - n**2 + n
dp/dt = c2*p - c2*p**2/n
dc0/dt = 0
dc1/dt = 0
dc2/dt = 0
>>> translation.translate_parameter_substitutions(system)
{k: c1, n: n, r: 1, d: c0, K: 1, h: 1, s: c2, p: p, t: t}
>>> soln_reduced = sympy.var('x1, x2, x3, t, n, p, c0, c1, c2')
>>> translation.reverse_translate_parameter(soln_reduced)
Matrix([[t*x1, n*x2, p*x3, x2, c0*x2, x2/x3, c1*x2/(x1*x3), 1/x1, c2/x1]])
```

ODETranslation.**reverse_translate_dep_var**(*variables*, *indep_var_index*)

Given the solutions of a (dep_var) reduced system, reverse translate them into solutions of the original system.

Parameters

- **variables** (*iter of sympy.Expression*) – The solution auxiliary variables $x(t)$ and solution invariants $y(t)$ of the reduced system. Variables should be ordered auxiliary variables followed by invariants.
- **indep_var_index** (*int*) – The location of the independent variable.

Return type tuple

Example 6.4 from [HL13]. We will just take the matrices from the paper, rather than use our own (which are constructed using different conventions).

$$\frac{dz_1}{dt} = z_1\left(1 + z_1 z_2\right)$$
$$\frac{dz_2}{dt} = z_2\left(\frac{1}{t} - z_1 z_2\right)$$

```
>>> equations = 'dz1/dt = z1*(1+z1*z2);dz2/dt = z2*(1/t - z1*z2)'.split(';')
>>> system = ODESystem.from_equations(equations)
>>> var_order = ['t', 'z1', 'z2']
>>> system.reorder_variables(var_order)
>>> scaling_matrix = sympy.Matrix([[0, 1, -1]])
>>> hermite_multiplier = sympy.Matrix([[0, 1, 0],
...                                    [1, 0, 1],
...                                    [0, 0, 1]])
>>> translation = ODETranslation(scaling_matrix=scaling_matrix,
...                              hermite_multiplier=hermite_multiplier)
>>> reduced = translation.translate_dep_var(system)
>>> # Check reduction
>>> answer = ODESystem.from_equations('dx0/dt = x0*(y0 + 1);dy0/dt = y0*(1 + 1/t)
↪'.split(';'))
>>> reduced == answer
True
>>> reduced
dt/dt = 1
dx0/dt = x0*(y0 + 1)
dy0/dt = y0*(1 + 1/t)
>>> t_var, c1_var, c2_var = sympy.var('t c1 c2')
```

(continues on next page)

(continued from previous page)

```
>>> reduced_soln = (c2_var*sympy.exp(t_var+c1_var*(1-t_var)*sympy.exp(t_var)),
...                 c1_var * t_var * sympy.exp(t_var))
>>> original_soln = translation.reverse_translate_dep_var(reduced_soln, system.
↪indep_var_index)
>>> original_soln == (reduced_soln[0], reduced_soln[1] / reduced_soln[0])
True
>>> original_soln
(c2*exp(c1*(-t + 1)*exp(t) + t), c1*t*exp(t)*exp(-c1*(-t + 1)*exp(t) - t)/c2)
```

ODETranslation.**reverse_translate_general**(*variables*, *system_indep_var_index=0*)

Given an iterable of variables, or exprs, reverse translate into the original variables. Here we expect t as the first variable, since we need to divide by it and substitute

Given the solutions of a (general) reduced system, reverse translate them into solutions of the original system.

Parameters

- **variables** (*iter of sympy.Expression*) – The independent variable t, the solution auxiliary variables $x(t)$ and the solution invariants $y(t)$ of the reduced system. Variables should be ordered: independent variable, auxiliary variables then invariants.
- **indep_var_index** (*int*) – The location of the independent variable.

Return type tuple

Example 6.6 from [HL13]. We will just take the matrices from the paper, rather than use our own (which are constructed using different conventions).

$$\frac{dz_1}{dt} = \frac{z_1\left(z_1^5 z_2 - 2\right)}{3t}$$
$$\frac{dz_2}{dt} = \frac{z_2\left(10 - 2z_1^5 z_2 + \frac{3z_1^2 z_2}{t}\right)}{3t}$$

```
>>> equations = ['dz1/dt = z1*(z1**5*z2 - 2)/(3*t)',
...              'dz2/dt = z2*(10 - 2*z1**5*z2 + 3*z1**2*z2/t )/(3*t)']
>>> system = ODESystem.from_equations(equations)
>>> var_order = ['t', 'z1', 'z2']
>>> system.reorder_variables(var_order)
>>> scaling_matrix = sympy.Matrix([[3, -1, 5]])
>>> hermite_multiplier = sympy.Matrix([[1, 1, -1],
...                                    [2, 3, 2],
...                                    [0, 0, 1]])
>>> translation = ODETranslation(scaling_matrix=scaling_matrix,
...                              hermite_multiplier=hermite_multiplier)
>>> reduced = translation.translate_general(system)
>>> # Quickly check the reduction
>>> answer = ODESystem.from_equations(['dx0/dt = x0*(2*y0*y1/3 - 1/3)/t',
...                                    'dy0/dt = y0*(y0*y1 - 1)/t',
...                                    'dy1/dt = y1*(y1 + 1)/t'])
>>> reduced == answer
True
>>> reduced
dt/dt = 1
dx0/dt = x0*(2*y0*y1/3 - 1/3)/t
dy0/dt = y0*(y0*y1 - 1)/t
dy1/dt = y1*(y1 + 1)/t
```

Check reverse translation:

```
>>> reduced_soln = (sympy.var('t'),
...                 sympy.sympify('c3/(t**(1/3)*(ln(t-c1)-ln(t)+c2)**(2/3))'),  #␣
↪x
...                 sympy.sympify('c1/(t*(ln(t-c1)-ln(t)+c2))'),  # y1
...                 sympy.sympify('t/(c1 - t)'))  # y2
>>> original_soln = translation.reverse_translate_general(reduced_soln, system.
↪indep_var_index)
>>> original_soln_answer = [#reduced_soln[1] ** 3 / (reduced_soln[2] ** 2)  # x^3␣
↪/ y1^2
...                          reduced_soln[2] / reduced_soln[1],  # y1 / x
...                          reduced_soln[1] ** 5 * reduced_soln[3] / reduced_
↪soln[2] ** 4]  # x^5 y2 / y1^4
>>> original_soln_answer = tuple([soln.subs({sympy.var('t'): sympy.sympify('t /␣
↪(c3**3 / c1**2)')})
...                                for soln in original_soln_answer])
>>> original_soln == original_soln_answer
True
>>> original_soln[0]
c1/(c3*(c1**2*t/c3**3)**(2/3)*(c2 - log(c1**2*t/c3**3) + log(c1**2*t/c3**3 -␣
↪c1))**(1/3))
>>> original_soln[1]
c3**5*(c1**2*t/c3**3)**(10/3)*(c2 - log(c1**2*t/c3**3) + log(c1**2*t/c3**3 -␣
↪c1))**(2/3)/(c1**4*(-c1**2*t/c3**3 + c1))
```

4.4 Extending from a set of Invariants

ODETranslation.**extend_from_invariants**(*invariant_choice*)

Extend a given set of invariants, expressed as a matrix of exponents, to find a Hermite multiplier that will rewrite the system in terms of invariant_choice.

Parameters **invariant_choice** (*sympy.Matrix*) – The $n \times k$ matrix representing the invariants, where n is the number of variables and k is the number of given invariants.

Returns An ODETranslation representing the rewrite rules in terms of the given invariants.

Return type *ODETranslation*

```
>>> variables = sympy.symbols(' '.join(['y{}'.format(i) for i in xrange(6)]))
>>> ode_translation = ODETranslation(sympy.Matrix([[1, 0, 3, 0, 2, 2],
...                                                [0, 2, 0, 1, 0, 1],
...                                                [2, 0, 0, 3, 0, 0]]))
>>> ode_translation.invariants(variables=variables)
Matrix([[y0**3*y2*y5**2/(y3**2*y4**5), y1*y4**2/y5**2, y2**2/y4**3]])
```

Now we can express two new invariants, which we think are more interesting, as a matrix. We pick the product of the first two invariants, and the product of the last two invariants: `y0**3 * y1 * y2/(y3**2 * y4**3)` and `y1 * y2**2/(y4 * y5**2)`

```
>>> new_inv = sympy.Matrix([[3, 1, 1, -2, -3, 0],
...                         [0, 1, 2, 0, -1, -2]]).T
```

```
>>> new_translation = ode_translation.extend_from_invariants(new_inv)
>>> new_translation.invariants(variables=variables)
Matrix([[y0**3*y1*y2/(y3**2*y4**3), y1*y2**2/(y4*y5**2), y1*y4**2/y5**2]])
```

4.5 Useful Attributes

ODETranslation.**invariants**(*variables=None*)

The invariants of the system, as determined by *herm_mult_n*.

Returns

The invariants of the system, in terms of *variables_domain* if not None. Otherwise, use `variables` and failing that use y_i.

Return type tuple

```
>>> equations = ['dn/dt = n*( r*(1 - n/K) - k*p/(n+d) )','dp/dt = s*p*(1 - h*p /␣
↪n)']
>>> system = ODESystem.from_equations(equations)
>>> translation = ODETranslation.from_ode_system(system)
>>> translation.invariants()
Matrix([[s*t, n/d, k*p/(d*s), K/d, h*s/k, r/s]])
```

ODETranslation.**auxiliaries**(*variables=None*)

The auxiliary variables of the system, as determined by *herm_mult_i*.

Returns

The auxiliary variables of the system, in terms of *variables_domain* if not `None`. Otherwise, use `variables` and failing that use x_i.

Return type tuple

```
>>> equations = ['dn/dt = n*( r*(1 - n/K) - k*p/(n+d) )','dp/dt = s*p*(1 - h*p /␣
↪n)']
>>> system = ODESystem.from_equations(equations)
>>> translation = ODETranslation.from_ode_system(system)
>>> translation.auxiliaries()
Matrix([[1/s, d, d*s/k]])
```

ODETranslation.**rewrite_rules**(*variables=None*)

Given a set of variables, print the rewrite rules. These are the rules that allow you to write any rational invariant in terms of the *invariants*.

Parameters **variables** (*iter of sympy.Symbol*) – The names of the variables appearing in the rational invariant.

Returns A `dict` whose keys are the given variables and whose values are the values to be substituted.

Return type dict

```
>>> equations = ['dn/dt = n*( r*(1 - n/K) - k*p/(n+d) )','dp/dt = s*p*(1 - h*p /␣
↪n)']
>>> system = ODESystem.from_equations(equations)
>>> translation = ODETranslation.from_ode_system(system)
>>> translation.rewrite_rules()
{k: 1, n: n/d, r: r/s, d: 1, K: K/d, h: h*s/k, s: 1, p: k*p/(d*s), t: s*t}
```

For example, rt is an invariant. Substituting in the above mapping gives us a way to write it in terms of our generating set of invariants:

$$rt \mapsto \left(\frac{r}{s}\right)(st) = rt$$

4.6 Output Functions

ODETranslation.**to_tex**()

Returns The scaling matrix A, the Hermite multiplier V and $W = V^{-1}$, in beautiful LaTeX.

Return type str

```
>>> print ODETranslation(sympy.Matrix(range(12)).reshape(3, 4)).to_tex()
A=
0 & 1 & 2 & 3 \\
4 & 5 & 6 & 7 \\
8 & 9 & 10 & 11 \\
V=
0 & 0 & 1 & 0 \\
0 & 0 & 0 & 1 \\
-1 & 3 & -3 & -2 \\
1 & -2 & 2 & 1 \\
W=
0 & 1 & 2 & 3 \\
1 & 1 & 1 & 1 \\
1 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 \\
```

4.7 Advanced Methods

These methods will be familiar to those who use Lie groups to analyse more general symmetries of differential equations. For more information, see [FO99] or [Hub07].

ODETranslation.**moving_frame**(*variables=None*)

Given a finite-dimensional Lie group G acting on a manifold M, a moving frame is defined as a G-equivariant map $\rho : M \to G$.

We can construct a moving frame given a rational section, by sending a point $x \in M$ to the Lie group element $\rho(x)$ such that $\rho(x) \cdot x$ lies on K.

```
>>> equations = 'dz1/dt = z1*(1+z1*z2);dz2/dt = z2*(1/t - z1*z2)'.split(';')
>>> system = ODESystem.from_equations(equations)
>>> translation = ODETranslation.from_ode_system(system)
>>> moving_frame = translation.moving_frame()
>>> moving_frame
(z2,)
```

Now we can check that the moving frame moves a point to the cross-section.

```
>>> translation.rational_section()
Matrix([[1 - 1/z2]])
>>> scale_action(moving_frame, translation.scaling_matrix)  # \rho(x)
Matrix([[1, z2, 1/z2]])
>>> # :math:`\rho(x).x` should satisfy the equations of the rational section.
>>> scale_action(moving_frame, translation.scaling_matrix).multiply_
↪elementwise(sympy.Matrix(system.variables).T)
Matrix([[t, z1*z2, 1]])
```

Another example:

```
>>> equations = ['dn/dt = n*( r*(1 - n/K) - k*p/(n+d) )','dp/dt = s*p*(1 - h*p /␣
↪n)']
>>> system = ODESystem.from_equations(equations)
>>> translation = ODETranslation.from_ode_system(system)
>>> moving_frame = translation.moving_frame()
>>> moving_frame
(s, 1/d, k/(d*s))
```

Now we can check that the moving frame moves a point to the cross-section.

```
>>> translation.rational_section()
Matrix([[1 - 1/s, d - 1, d*s - 1/k]])
>>> scale_action(moving_frame, translation.scaling_matrix)  # \rho(x)
Matrix([[s, 1/d, k/(d*s), 1/d, 1/d, s/k, 1/k, 1/s, 1/s]])
>>> # :math:`\rho(x).x` should satisfy the equations of the rational section.
>>> scale_action(moving_frame, translation.scaling_matrix).multiply_
↪elementwise(sympy.Matrix(system.variables).T)
Matrix([[s*t, n/d, k*p/(d*s), K/d, 1, h*s/k, 1, r/s, 1]])
```

See [FO99] for more details.

ODETranslation.**rational_section**(*variables=None*)

Given a finite-dimensional Lie group G acting on a manifold M, a cross-section $K \subset M$ is a subset that intersects each orbit of M in exactly one place.

Parameters **variables** (*iter of sympy.Symbol, optional*) – Variables of the system.

Returns A set of equations defining a variety that is a cross-section.

Return type sympy.Matrix

```
>>> equations = 'dz1/dt = z1*(1+z1*z2);dz2/dt = z2*(1/t - z1*z2)'.split(';')
>>> system = ODESystem.from_equations(equations)
>>> translation = ODETranslation.from_ode_system(system)
>>> translation.rational_section()
Matrix([[1 - 1/z2]])
```

```
>>> equations = ['dn/dt = n*( r*(1 - n/K) - k*p/(n+d) )','dp/dt = s*p*(1 - h*p /␣
↪n)']
>>> system = ODESystem.from_equations(equations)
>>> translation = ODETranslation.from_ode_system(system)
>>> translation.rational_section()
Matrix([[1 - 1/s, d - 1, d*s - 1/k]])
```

See [FO99] for more details.

CHAPTER 5

Chemical Reaction Networks

One flavour of dynamical system that can be analysed using desr is the Chemical Reaction Network (CRN). The *ChemicalReactionNetwork* class is useful for drawing out dynamical systems and automatically generating *ODESystem* instances.

5.1 Example Use

It is possible to build up a *ChemicalReactionNetwork* instance using the constituent parts as follows. While this is the most precise, the easiest way to create a network is using *Diagrams*.

First we must define some chemical species:

```
>>> species = sympy.var('x1 x2')
>>> species = map(ChemicalSpecies, species)
>>> x1, x2 = species
```

Then we must make complexes, which are representations of one side of a chemical reaction diagram. It is represented using a dictionary, where the keys represent the chemical elements and the value represents its coefficient in the equation.

```
>>> complex0 = Complex({x1: 1, x2: 1})
>>> complex1 = Complex({x2: 2})
>>> complex2 = Complex({x1: 1})
>>> complex3 = Complex({x2: 1})
>>> complexes = (complex0, complex1, complex2, complex3)
```

The final step is to make reactions, which simply take two complexes (two sides of an equation) and form an arrow from the first side to the second. For example, `r1` represents the chemical reaction `x1 + x2 -> 2*x2`.

```
>>> r1 = Reaction(complex0, complex1)
>>> r2 = Reaction(complex3, complex2)
>>> reactions = [r1, r2]
```

Finally, we form a network from these constituent parts and form an *ODESystem* instance. The rate constants are autogenerated and follow the convention that `k_i_j` is the rate constant for complex `i` -> complex `j`.

```
>>> reaction_network = ChemicalReactionNetwork(species, complexes, reactions)
>>> system = reaction_network.to_ode_system()
>>> system
dt/dt = 1
dx1/dt = -k_0_1*x1*x2 + k_3_2*x2
dx2/dt = k_0_1*x1*x2 - k_3_2*x2
dk_0_1/dt = 0
dk_3_2/dt = 0
>>> answer = ODESystem.from_equations('dx1/dt = -k_0_1*x1*x2 + k_3_2*x2\ndx2/dt = k_0_
↪1*x1*x2 - k_3_2*x2')
>>> system == answer
True
```

5.2 Diagrams

A much more convenient way is to generate *ChemicalReactionNetwork*'s, and hence *ODESystem*'s, with diagrams.

This can be done like so:

```
>>> reactions = ['x1 + x2 -> 2*x2',
...              'x2 -> x1']
>>> reaction_network = ChemicalReactionNetwork.from_diagram('\n'.join(reactions))
>>> reaction_network
1.x2 + 1.x1 -> 2.x2
1.x2 -> 1.x1
>>> system = reaction_network.to_ode_system()
>>> system
dt/dt = 1
dx1/dt = -k_0_1*x1*x2 + k_2_3*x2
dx2/dt = k_0_1*x1*x2 - k_2_3*x2
dk_0_1/dt = 0
dk_2_3/dt = 0
```

Note that `k_3_2` has become `k_2_3` since complex order is automatically determined when feeding in a diagram. Otherwise, we have an entirely equivalent way of constructing CRNs and their associated dynamical systems.

CHAPTER 6

Examples

6.1 Example Michaelis-Menten

We derive the analysis of the Michaelis-Menten equations found in [SS89]: in a systematic manner.

After using the law of mass action, we are able to reduce the initial set of equations to

$$\frac{ds}{dt} = -k_1 e_0 s + k_1 c s + k_{-1} c \tag{6.1}$$

$$\frac{dc}{dt} = k_1 e_0 s - k_1 c s - k_{-1} c - k_2 c \tag{6.2}$$

$$s(0) = s_0 \tag{6.3}$$

First we must create the system. Before we perform any analysis, we set $K = k_2 + k_{-1}$ and eliminate k_{-1}. Why can/do we do this??

```
>>> system_tex = '''\frac{ds}{dt} &= - k_1 e_0 s + k_1 c s + k_{-1} c \\\\
...           \frac{dc}{dt} &= k_1 e_0 s - k_1 c s - k_{-1} c - k_2 c'''
>>> system_tex_reduced_km1 = system_tex.replace('k_{-1}', '(K - k_2)')
>>> reduced_system_km1 = ODESystem.from_tex(system_tex_reduced_km1)
```

We reorder the variables, so that we try to normalise by later oners.

```
>>> reduced_system_km1.reorder_variables(['t', 's', 'c', 'K', 'k_2', 'k_1', 'e_0'])
>>> reduced_system_km1.variables
(t, s, c, K, k_2, k_1, e_0)
```

We can then see the exponent (or power) matrix, maximal scaling matrix and corresponding invariants:

```
>>> reduced_system_km1.power_matrix()
Matrix([
[ 1, 1,  1, 1, 1, 1,  1],
[-1, 0, -1, 0, 0, 1,  1],
[ 1, 0,  1, 1, 0, 0, -1],
```

(continues on next page)

(continued from previous page)

```
[ 0, 0,  1, 0, 1, 0,  0],
[ 1, 0,  0, 0, 0, 0,  0],
[ 0, 1,  0, 1, 0, 1,  1],
[ 0, 1,  0, 0, 0, 0,  1]])
>>> max_scal1 = ODETranslation.from_ode_system(reduced_system_km1)
>>> max_scal1.scaling_matrix
Matrix([
[1, 0, 0, -1, -1, -1, 0],
[0, 1, 1,  0,  0, -1, 1]])
>>> max_scal1.invariants()
Matrix([[e_0*k_1*t, s/e_0, c/e_0, K/(e_0*k_1), k_2/(e_0*k_1)]])
```

The reduced system is also computed:

```
>>> max_scal1.translate(reduced_system_km1)
dt/dt = 1
dc/dt = -c*c0 - c*s + s
ds/dt = c*c0 - c*c1 + c*s - s
dc0/dt = 0
dc1/dt = 0
```

However, we need to add in our initial condition for s.

```
>>> reduced_system_km1.update_initial_conditions({'s': 's_0'})
>>> max_scal2 = ODETranslation.from_ode_system(reduced_system_km1)
>>> max_scal2.scaling_matrix
Matrix([
[1, 0, 0, -1, -1, -1, 0, 0],
[0, 1, 1,  0,  0, -1, 1, 1]])
>>> max_scal2.invariants()
Matrix([[k_1*s_0*t, s/s_0, c/s_0, K/(k_1*s_0), k_2/(k_1*s_0), e_0/s_0]])
>>> max_scal2.translate(reduced_system_km1)
dt/dt = 1
dc/dt = -c*c0 - c*s + c2*s
ds/dt = c*c0 - c*c1 + c*s - c2*s
dc0/dt = 0
dc1/dt = 0
dc2/dt = 0
d1/dt = 0
s(0) = 1
```

Some elementary column operations give us equations 8

```
>>> max_scal2.multiplier_add_columns(2, -1, 1)  # Scale time by e_0 not s_0
>>> max_scal2.multiplier_add_columns(4, -1, -1)  # Scale c by e_0
>>> max_scal2.invariants()
Matrix([[e_0*k_1*t, s/s_0, c/e_0, K/(k_1*s_0), k_2/(k_1*s_0), e_0/s_0]])
>>> max_scal2.translate(reduced_system_km1)
dt/dt = 1
dc/dt = -c*c0/c2 - c*s/c2 + s/c2
ds/dt = c*c0 - c*c1 + c*s - s
dc0/dt = 0
dc1/dt = 0
dc2/dt = 0
d1/dt = 0
s(0) = 1
```

We can also scale time by ϵ to get the "inner" equation 11:

```
>>> max_scal2.multiplier_add_columns(2, -1, -1)  # Divide time through by epsilon
>>> max_scal2.invariants()
Matrix([[k_1*s_0*t, s/s_0, c/e_0, K/(k_1*s_0), k_2/(k_1*s_0), e_0/s_0]])
>>> max_scal2.translate(reduced_system_km1)
dt/dt = 1
dc/dt = -c*c0 - c*s + s
ds/dt = c*c0*c2 - c*c1*c2 + c*c2*s - c2*s
dc0/dt = 0
dc1/dt = 0
dc2/dt = 0
d1/dt = 0
s(0) = 1
```

What is epsilon is not small? We can find that $s_0 + K_m$ is an invariant systematically. So we add a variable $L = s_0 + K_m$.

```
>>> # Substitute K_m into the equations
>>> system_tex_reduced_l = system_tex.replace('k_{-1}', '(K - k_2)').replace('K', 'K_
↪m k_1')
>>> reduced_system_l = ODESystem.from_tex(system_tex_reduced_l)
>>> reduced_system_l
dt/dt = 1
dc/dt = -c*k_1*s - c*k_2 - c*(K_m*k_1 - k_2) + e_0*k_1*s
ds/dt = c*k_1*s + c*(K_m*k_1 - k_2) - e_0*k_1*s
dK_m/dt = 0
de_0/dt = 0
dk_1/dt = 0
dk_2/dt = 0
>>> reduced_system_l.update_initial_conditions({'s': 's_0'})
>>> reduced_system_l.add_constraints('L', 's_0 + K_m')
```

Check that if we keep L at the end, we have the same reduced system as before

```
>>> reduced_system_l.reorder_variables(['t', 's', 'c', 'K_m', 'k_2', 'k_1', 'e_0', 'L
↪', 's_0'])
>>> max_scal = ODETranslation.from_ode_system(reduced_system_l)
>>> max_scal.scaling_matrix
Matrix([
[1, 0, 0, 0, -1, -1, 0, 0, 0],
[0, 1, 1, 1,  0, -1, 1, 1, 1]])
>>> max_scal.invariants()
Matrix([[k_1*s_0*t, s/s_0, c/s_0, K_m/s_0, k_2/(k_1*s_0), e_0/s_0, L/s_0]])
>>> max_scal.translate(reduced_system_l)
dt/dt = 1
dc/dt = -c*c0 - c*s + c2*s
ds/dt = c*c0 - c*c1 + c*s - c2*s
dc0/dt = 0
dc1/dt = 0
dc2/dt = 0
d1/dt = 0
dc3/dt = 0
s(0) = 1
c3 == c0 + 1
```

Now we put L into the mix:

```
>>> reduced_system_l.reorder_variables(['t', 's', 'c', 'k_2', 'k_1', 'e_0', 's_0', 'L
↪', 'K_m'])
>>> max_scal3 = ODETranslation.from_ode_system(reduced_system_l)
>>> # Scale t correctly to t/t_C = k_1 L t
>>> max_scal3.multiplier_add_columns(2, -1, 1)
>>> # Scale s correctly to s / s_0
>>> max_scal3.multiplier_add_columns(3, -2, -1)
>>> # Scale c correctly to c / (e_0 s_0 / L)
>>> max_scal3.multiplier_add_columns(4, 6, -1)
>>> max_scal3.multiplier_add_columns(4, 7, -1)
>>> max_scal3.multiplier_add_columns(4, -1, 1)
>>> # Find kappa = k_{-1} / k_2 = (K_m k_1 / k_2) - 1
>>> max_scal3.multiplier_negate_column(5)
>>> # Find epsilon = e_0 / L
>>> max_scal3.multiplier_add_columns(6, -1, -1)
>>> # Find sigma = s_0 / K_m
>>> max_scal3.invariants()
Matrix([[L*k_1*t, s/s_0, L*c/(e_0*s_0), K_m*k_1/k_2, e_0/L, s_0/K_m, L/K_m]])
```

We now have:

$$c_0 = \kappa + 1 \tag{6.4}$$

$$c_1 = \epsilon \tag{6.5}$$

$$c_2 = \sigma \tag{6.6}$$

$$c_3 = \frac{L}{K_m} \tag{6.7}$$

Which gives us exactly equations 24 from Segel, after some trivial rearrangement.

```
>>> max_scal3.translate(reduced_system_l)
dt/dt = 1
dc/dt = -c*c2*s/c3 - c/c3 + s
ds/dt = c*c1*c2*s/c3 + c*c1/c3 - c*c1/(c0*c3) - c1*s
dc0/dt = 0
dc1/dt = 0
dc2/dt = 0
dc3/dt = 0
d1/dt = 0
s(0) = 1
c3 == c2 + 1
```

To get the equations on the other timescale, we need to multiply Lk_1t by $\frac{e_0}{L}\frac{k_2}{K_m*k_1}\frac{K_m}{L} = \frac{c_1}{c_0c_3}$

```
>>> max_scal3.multiplier_add_columns(2, 6, 1)
>>> max_scal3.multiplier_add_columns(2, 5, -1)
>>> max_scal3.multiplier_add_columns(2, -1, -1)
>>> max_scal3.invariants()
Matrix([[e_0*k_2*t/L, s/s_0, L*c/(e_0*s_0), K_m*k_1/k_2, e_0/L, s_0/K_m, L/K_m]])
>>> max_scal3.translate(reduced_system_l)
dt/dt = 1
dc/dt = -c*c0*c2*s/c1 - c*c0/c1 + c0*c3*s/c1
ds/dt = c*c0*c2*s + c*c0 - c - c0*c3*s
dc0/dt = 0
dc1/dt = 0
dc2/dt = 0
dc3/dt = 0
d1/dt = 0
```

(continues on next page)

(continued from previous page)

```
s(0) = 1
c3 == c2 + 1
```

6.1.1 Raw Michaelis-Menten Equation Analysis

We perform an example analysis of the Michael-Mentis equations [SS89]:

$$\frac{dE}{dt} = -k_1 ES + k_{-1} C + k_2 C \tag{6.8}$$

$$\frac{dS}{dt} = -k_1 ES + k_{-1} C \tag{6.9}$$

$$\frac{dC}{dt} = k_1 ES - k_{-1} C - k_2 C \tag{6.10}$$

$$\frac{dP}{dt} = k_2 C \tag{6.11}$$

First we initiate the system from LaTeX and find the maximal scaling matrix such that the system is invariant. Note that negative subscripts are turned into 'm' so that they comply with `sympy`. The 'm's are turned back into negatives when printint to LaTeX using *`desr.tex_tools()`*.

```
>>> import sympy
>>> from desr.matrix_normal_forms import smf
>>> from desr.ode_system import ODESystem
>>> from desr.ode_translation import ODETranslation, scale_action
>>> from desr.tex_tools import expr_to_tex
>>> system_tex = '''\frac{dE}{dt} &= - k_1 E S + k_{-1} C + k_2 C \\\\
...                 \frac{dS}{dt} &= - k_1 E S + k_{-1} C \\\\
...                 \frac{dC}{dt} &= k_1 E S - k_{-1} C - k_2 C \\\\
...                 \frac{dP}{dt} &= k_2 C'''
>>> original_system = ODESystem.from_tex(system_tex)
>>> max_scal1 = ODETranslation.from_ode_system(original_system)
>>> print 'Variable order: ', max_scal1.variables_domain
Variable order:  (t, C, E, P, S, k_1, k_2, k_m1)
>>> print 'Scaling Matrix:\n', max_scal1.scaling_matrix.__repr__()
Scaling Matrix:
Matrix([
[1, 0, 0, 0, 0, -1, -1, -1],
[0, 1, 1, 1, 1, -1,  0,  0]])
```

Now we can inspect the invariants easily:

```
>>> print 'Invariants: ', max_scal1.invariants()
Invariants:  Matrix([[k_m1*t, C*k_1/k_m1, E*k_1/k_m1, P*k_1/k_m1, S*k_1/k_m1, k_2/k_
↪m1]])
```

Finding the reduced system is also easy. Since the Hermite multiplier and inverse are compatible with the simplest parameter reduction scheme, *`translate()`* will automatically perform this reduction.

```
>>> print 'Reduced system:\n', max_scal1.translate(original_system)
Reduced system:
dt/dt = 1
dC/dt = -C*c0 - C + E*S
dE/dt = C*c0 + C - E*S
dP/dt = C*c0
```

(continues on next page)

(continued from previous page)

```
dS/dt = C - E*S
dc0/dt = 0
```

6.1.2 Changing the variable order

In our previous example, we had k_{-1} at the end of the variable order, so that the algorithm tries to normalise using k_{-1}. Instead, we can choose to normalise by k_2, by swapping around the last two variables. Note that we need to recalculate the *ODETranslation* instance.

```
>>> original_system_reorder = original_system.copy()
>>> variable_order = list(original_system.variables)
>>> variable_order[-1], variable_order[-2] = variable_order[-2], variable_order[-1]
↪# Swap the last two variables
>>> original_system_reorder.reorder_variables(variable_order)
>>> original_system_reorder.variables
(t, C, E, P, S, k_1, k_m1, k_2)
>>> max_scal1_reorder = ODETranslation.from_ode_system(original_system_reorder)
>>> print 'Invariants:', ', '.join(map(str, max_scal1_reorder.invariants()))
Invariants: k_2*t, C*k_1/k_2, E*k_1/k_2, P*k_1/k_2, S*k_1/k_2, k_m1/k_2
```

Now we can reduce to find another, equivalent system.

```
>>> reduced_system = max_scal1_reorder.translate(original_system_reorder)
>>> reduced_system
dt/dt = 1
dC/dt = -C*c0 - C + E*S
dE/dt = C*c0 + C - E*S
dP/dt = C
dS/dt = C*c0 - E*S
dc0/dt = 0
```

6.1.3 Extending a choice of invariants

We return to our original variable order: $t, C, E, P, S, k_1, k_2, k_{-1}$.

Suppose we wish to study the invariants $\frac{k_1}{k_2}C$ and $\frac{k_1}{k_{-1}}P$. Then we must create a matrix representing these invariants:

$$P = \begin{bmatrix} 0 & 0 \\ 1 & 0 \\ 0 & 0 \\ 0 & 1 \\ 0 & 0 \\ 1 & 1 \\ -1 & 0 \\ 0 & -1 \end{bmatrix}.$$

We can easily check we have correct matrix:

```
>>> invariant_choice = sympy.Matrix([[0, 1, 0, 0, 0, 1, -1, 0],
...                                  [0, 0, 0, 1, 0, 1, 0, -1]]).T
>>> scale_action(max_scal1.variables_domain, invariant_choice)
Matrix([[C*k_1/k_2, P*k_1/k_m1]])
```

Finding a maximal scaling matrix that can be used to rewrite the system in terms of these invariants is also simple.

```
>>> max_scal2 = max_scal1.extend_from_invariants(invariant_choice=invariant_choice)
>>> max_scal2
A=
Matrix([
[1, 0, 0, 0, 0, -1, -1, -1],
[0, 1, 1, 1, 1, -1,  0,  0]])
V=
Matrix([
[ 0,  0,  0,  0, 1,  0,  0,  0],
[ 0,  0,  1,  0, 0,  0,  0,  0],
[ 0,  0,  0,  0, 0,  1,  0,  0],
[ 0,  0,  0,  1, 0,  0,  0,  0],
[ 0,  0,  0,  0, 0,  0,  1,  0],
[ 0, -1,  1,  1, 0,  1,  1,  0],
[ 0,  0, -1,  0, 0,  0,  0,  1],
[-1,  1,  0, -1, 1, -1, -1, -1]])
W=
Matrix([
[1, 0, 0, 0, 0, -1, -1, -1],
[0, 1, 1, 1, 1, -1,  0,  0],
[0, 1, 0, 0, 0,  0,  0,  0],
[0, 0, 0, 1, 0,  0,  0,  0],
[1, 0, 0, 0, 0,  0,  0,  0],
[0, 0, 1, 0, 0,  0,  0,  0],
[0, 0, 0, 0, 1,  0,  0,  0],
[0, 1, 0, 0, 0,  0,  1,  0]])
```

For Python code that steps through this procedure, see `desr.examples.example_michael_mentis`.

Now, this transformation doesn't satisfy the conditions of the parameter reduction scheme, so if we try to reduce it *`translate()`* will use the dependent reduction scheme implemented in *`translate_dep_var()`*.

```
>>> max_scal2.invariants()
Matrix([[C*k_1/k_2, P*k_1/k_m1, k_m1*t, E*k_1/k_m1, S*k_1/k_m1, k_2/k_m1]])
>>> max_scal2.translate(original_system)
dt/dt = 1
dx0/dt = 0
dx1/dt = 0
dy0/dt = y0*(-y2*y5 - y2 + y2*y3*y4/(y0*y5))/t
dy1/dt = y0*y2*y5**2/t
dy2/dt = y2/t
dy3/dt = y3*(y0*y2*y5**2/y3 + y0*y2*y5/y3 - y2*y4)/t
dy4/dt = y4*(y0*y2*y5/y4 - y2*y3)/t
dy5/dt = 0
```

Here, `x0` and `x1` are auxiliary variables, which can be fixed at any value at all. `(y0, y1, y2, y3, y4) = (C*k_1/k_2, P*k_1/k_m1, k_m1*t, E*k_1/k_m1, S*k_1/k_m1)` are our new dependent invariants. Finally, `y5 = k_2/k_m1` is the single parameter of the reduced system.

However, we can see that after performing a permutation of the columns, we can satisfy the parameter reduction scheme. While this isn't implemented yet, we can do it by hand for the moment. We must apply the cycle $\begin{pmatrix}0 & 1 & 3 & 2\end{pmatrix}$ to the last $n - r$ columns.

```
>>> max_scal3 = max_scal2.herm_mult_n
>>> max_scal3.col_swap(0, 1)
>>> max_scal3.col_swap(0, 3)
>>> max_scal3.col_swap(0, 2)
>>> print 'Permuted Vn:\n', max_scal3.__repr__()
```

(continues on next page)

(continued from previous page)

```
Permuted Vn:
Matrix([
[1,  0,  0,  0,  0,  0],
[0,  1,  0,  0,  0,  0],
[0,  0,  1,  0,  0,  0],
[0,  0,  0,  1,  0,  0],
[0,  0,  0,  0,  1,  0],
[0,  1,  1,  1,  1,  0],
[0, -1,  0,  0,  0,  1],
[1,  0, -1, -1, -1, -1]])
>>> max_scal3 = sympy.Matrix.hstack(max_scal1.herm_mult_i, max_scal3)
>>> max_scal3 = ODETranslation(max_scal1.scaling_matrix, hermite_multiplier=max_scal3)
>>> print max_scal3.translate(original_system)
dt/dt = 1
dC/dt = -C*c0 - C + E*S/c0
dE/dt = C*c0**2 + C*c0 - E*S
dP/dt = C*c0**2
dS/dt = C*c0 - E*S
dc0/dt = 0
```

So we have found a third different reparametrization of the Michaelis-Menten equations.

Todo: Add a method to *ODETranslation* that will try and re-order the last $n - r$ columns so that the parameter reduction scheme can be applied.

6.2 Walkthroughs from Supplementary Information

Matching Segel and Slemrod's analysis.

```
>>> system_tex = '''\frac{ds}{dt} &= - k_1 e_0 s + k_1 c s + k_{-1} c \\\\
... \frac{dc}{dt} &= k_1 e_0 s - k_1 c s - k_{-1} c - k_2 c \\\\'''
>>> system_mm = ODESystem.from_tex(system_tex)
>>> system_mm.update_initial_conditions({'s': 's_0'})
>>> system_mm.add_constraints('K_m', '(k_2 + k_m1) / k_1')
>>> system_mm.add_constraints('epsilon', 'e_0 / (s_0 + K_m)')
>>> system_mm.reorder_variables(['t', 's', 'c', 'epsilon', 'k_m1', 'k_2', 'k_1', 'K_m
↪', 'e_0', 's_0'])
>>> system_mm.variables
(t, s, c, epsilon, k_m1, k_2, k_1, K_m, e_0, s_0)
>>> max_scal1 = ODETranslation.from_ode_system(system_mm)
>>> max_scal1.scaling_matrix
Matrix([
[1, 0, 0, 0, -1, -1, -1, 0, 0, 0],
[0, 1, 1, 0,  0,  0, -1, 1, 1, 1]])
>>> max_scal1.translate(system_mm)
dt/dt = 1
dc/dt = -c*c1 - c*c2 - c*s + c4*s
ds/dt = c*c1 + c*s - c4*s
dc1/dt = 0
dc2/dt = 0
dc4/dt = 0
d1/dt = 0
```

(continues on next page)

(continued from previous page)

```
dc3/dt = 0
dc0/dt = 0
s(0) = 1
c3 == c1 + c2
c0 == c4/(c3 + 1)
```

First recreate the original system.

```
>>> # Scale t correctly to t/t_C = k_1 L t = e_0 k_1 t / epsilon
>>> max_scal1.multiplier_add_columns(2, 5, -1)
>>> max_scal1.multiplier_add_columns(2, -1, 1)
>>> # Scale s correctly to s / s_0
>>> # Scale c correctly to c / (e_0 s_0 / L) = c / (s_0 epsilon)
>>> max_scal1.multiplier_add_columns(4, 5, -1)
>>> # Find epsilon = e_0 / L
>>> # Find kappa = k_{-1} / k_2 = (K_m k_1 / k_2) - 1
>>> max_scal1.multiplier_add_columns(6, 7, -1)
>>> # Find sigma = s_0 / K_m
>>> max_scal1.multiplier_negate_column(-2)
```

Inner equations (21)

```
>>> max_scal1.invariants()
Matrix([[e_0*k_1*t/epsilon, s/s_0, c/(epsilon*s_0), epsilon, k_m1/k_2, k_2/(k_1*s_0),␣
↪s_0/K_m, e_0/s_0]])
>>> system_mm_red = max_scal1.translate(system_mm)
>>> system_mm_red = system_mm_red.diff_subs({sympy.sympify('c2'): sympy.sympify('1 /␣
↪(sigma * (kappa + 1))'),
...                                           sympy.sympify('c4'): sympy.sympify(
↪'epsilon * (1 + 1 / sigma)'),
...                                           })
>>> system_mm_red.diff_subs({'c0': 'epsilon',
...                          'c3': 'sigma',
...                          'c1': 'kappa',
...                          's': 'u', 'c': 'v'},
...                         subs_constraints=True,
...                         expand_after=True,
...                         factor_after=True)
dt/dt = 1
dc/dt = -(sigma*u*v - sigma*u - u + v)/(sigma + 1)
ds/dt = epsilon*(kappa*sigma*u*v - kappa*sigma*u - kappa*u + kappa*v + sigma*u*v -␣
↪sigma*u - u)/((kappa + 1)*(sigma + 1))
dc0/dt = 0
dc1/dt = 0
dc2/dt = 0
dc4/dt = 0
d1/dt = 0
dc3/dt = 0
dkappa/dt = 0
dsigma/dt = 0
depsilon/dt = 0
s(0) = 1
1/sigma == c2*kappa + c2
epsilon == c4/(1 + 1/sigma)
```

Outer equations (24)

```
>>> # Scale t correctly to t/t_S = k_2 epsilon t
>>> max_scal1.multiplier_add_columns(2, 7, 1)
>>> max_scal1.multiplier_add_columns(2, -1, -1)
>>> max_scal1.multiplier_add_columns(2, 5, 2)
>>> max_scal1.invariants()
Matrix([[epsilon*k_2*t, s/s_0, c/(epsilon*s_0), epsilon, k_m1/k_2, k_2/(k_1*s_0), s_0/
↪K_m, e_0/s_0]])
>>> system_mm_red = max_scal1.translate(system_mm)
>>> system_mm_red = system_mm_red.diff_subs({sympy.sympify('c2'): sympy.sympify('1 /␣
↪(sigma * (kappa + 1))'),
...                                         sympy.sympify('c4'): sympy.sympify(
↪'epsilon * (1 + 1 / sigma)'),
...                                         })
>>> system_mm_red.diff_subs({'c0': 'epsilon',
...                          'c3': 'sigma',
...                          'c1': 'kappa',
...                          's': 'u', 'c': 'v'},
...                         subs_constraints=True,
...                         expand_after=True,
...                         factor_after=True)
dt/dt = 1
dc/dt = -(kappa + 1)*(sigma*u*v - sigma*u - u + v)/epsilon
ds/dt = kappa*sigma*u*v - kappa*sigma*u - kappa*u + kappa*v + sigma*u*v - sigma*u - u
dc0/dt = 0
dc1/dt = 0
dc2/dt = 0
dc4/dt = 0
d1/dt = 0
dc3/dt = 0
dkappa/dt = 0
dsigma/dt = 0
depsilon/dt = 0
s(0) = 1
1/sigma == c2*kappa + c2
epsilon == c4/(1 + 1/sigma)
```

CHAPTER 7

desr Reference

7.1 Submodules

7.2 desr.ode_system module

class desr.ode_system.**ODESystem**(*variables*, *derivatives*, *indep_var=None*, *initial_conditions=None*)

Bases: object

A system of differential equations.

The main attributes are *variables* and *derivatives*. *variables* is an ordered tuple of variables, which includes the independent variable. *derivatives* is an ordered tuple of the same length that contains the derivatives with respect to *indep_var*.

Parameters

- **variables** (*tuple of sympy.Symbol*) – Ordered tuple of variables.
- **derivatives** (*tuple of sympy.Expression*) – Ordered tuple of derivatives.
- **indep_var** (*sympy.Symbol, optional*) – Independent variable we are differentiating with respect to.
- **initial_conditions** (*tuple of sympy.Symbol*) – The initial values of non-constant variables

add_constraints(*lhs*, *rhs*)

Add constraints that must be obeyed by the system. :param lhs: The left hand side of the constraint. :type lhs: sympy.Expr :param rhs: The right hand side of the constraint. :type rhs: sympy.Expr

---

Todo:

- Finish docstring and tests, here and for: finding scaling symmetries and also translation
- Check for 0 case

```
>>> eqns = ['dx/dt = c_0*x*y', 'dy/dt = c_1*(1-x)*(1-y)']
```

```
>>> system = ODESystem.from_equations(eqns)
>>> system
dt/dt = 1
dx/dt = c_0*x*y
dy/dt = c_1*(-x + 1)*(-y + 1)
dc_0/dt = 0
dc_1/dt = 0
```

```
>>> system.add_constraints('c_2', 'c_0 + c_1')
>>> system
dt/dt = 1
dx/dt = c_0*x*y
dy/dt = c_1*(-x + 1)*(-y + 1)
dc_0/dt = 0
dc_1/dt = 0
dc_2/dt = 0
c_2 == c_0 + c_1
```

```
>>> system.add_constraints('c_2', 'c_0 + x')
Traceback (most recent call last):
    ...
ValueError: Cannot add constraints on non-constant parameters set([x]). This␣
↪would make an interesting project though...
```

```
>>> system.add_constraints('c_0', 0)
Traceback (most recent call last):
    ...
ValueError: Cannot express equality with 0.
```

constant_variables

Return the constant variables - specifically those which have a None derivative.

Returns The constant variables.

Return type tuple

constraints

- Finish docstring

Returns:

Type Todo

copy()

Returns A copy of the system.

Return type *ODESystem*

default_order_variables()

Reorder the variables into (independent variable, dependent variables, constant variables), which generally gives the simplest reductions. Variables of the same type are sorted by their string representations.

```
>>> eqns = ['dz_1/dt = z_1*z_3', 'dz_2/dt = z_1*z_2 / (z_3 ** 2)']
>>> system = ODESystem.from_equations('\n'.join(eqns))
>>> system.variables
(t, z_1, z_2, z_3)
```

```
>>> system.reorder_variables(['z_2', 'z_3', 't', 'z_1'])
>>> system.variables
(z_2, z_3, t, z_1)
```

```
>>> system.default_order_variables()
>>> system.variables
(t, z_1, z_2, z_3)
```

derivative_dict
expr mapping, filtering out the `None`'s in expr.

Returns Keys are non-constant variables, value is the derivative with respect to the independent variable.

Return type dict

Type Return a variable

derivatives
Getter for an ordered tuple of expressions representing the derivatives of self.variables.

Returns Ordered tuple of sympy.Expressions.

Return type tuple

diff_subs(*to_sub*, *expand_before=False*, *expand_after=True*, *factor_after=False*, *subs_constraints=False*)
Make substitutions into the derivatives, returning a new system. :param to_sub: Dictionary of substitutions to make. :type to_sub: dict :param expand_before: Expand the sympy expression for each derivative before substitution. :type expand_before: bool :param expand_after: Expand the sympy expression for each derivative after substitution. :type expand_after: bool :param factor_after: Factorise the sympy expression for each derivative after substitution. :type factor_after: bool :param subs_constraints: Perform the substitutions into the initial constraints. :type subs_constraints: bool

Returns System with substitutions carried out.

Return type *ODESystem*

```
>>> eqns = ['dx/dt = c_0*x*y', 'dy/dt = c_1*(1-x)*(1-y)']
>>> system = ODESystem.from_equations(eqns)
>>> system.diff_subs({'1-x': 'z'}, expand_before=False, expand_after=False,␣
↪factor_after=False)
dt/dt = 1
dx/dt = c_0*x*y
dy/dt = c_1*z*(-y + 1)
dc_0/dt = 0
dc_1/dt = 0
>>> system.diff_subs({'1-x': 'z'}, expand_before=True, expand_after=False,␣
↪factor_after=False)
dt/dt = 1
dx/dt = c_0*x*y
dy/dt = c_1*x*y - c_1*x - c_1*y + c_1
dc_0/dt = 0
dc_1/dt = 0
```

```
>>> system.diff_subs({'x': '1-z'}, expand_before=True, expand_after=True,␣
↪factor_after=False)
dt/dt = 1
dx/dt = -c_0*y*z + c_0*y
dy/dt = -c_1*y*z + c_1*z
dc_0/dt = 0
dc_1/dt = 0
```

```
>>> system.add_constraints('c_0', 'c_1**2')
>>> system.diff_subs({'c_0': '1'}, subs_constraints=False)
dt/dt = 1
dx/dt = x*y
dy/dt = c_1*x*y - c_1*x - c_1*y + c_1
dc_0/dt = 0
dc_1/dt = 0
c_0 == c_1**2
>>> system.diff_subs({'c_0': '1'}, subs_constraints=True)
dt/dt = 1
dx/dt = x*y
dy/dt = c_1*x*y - c_1*x - c_1*y + c_1
dc_0/dt = 0
dc_1/dt = 0
1 == c_1**2
```

classmethod from_dict(*deriv_dict*, *indep_var=t*, *initial_conditions=None*)

Instantiate from a text of equations.

Parameters

- **deriv_dict** (*dict*) – {variable: derivative} mapping.
- **indep_var** (*sympy.Symbol*) – Independent variable, that the derivatives are with respect to.
- **initial_conditions** (*tuple of sympy.Symbol*) – The initial values of non-constant variables

Returns System of ODEs.

Return type *ODESystem*

```
>>> _input = {'x': 'c_0*x*y', 'y': 'c_1*(1-x)*(1-y)'}
>>> _input = {sympy.Symbol(k): sympy.sympify(v) for k, v in _input.
↪iteritems()}
>>> ODESystem.from_dict(_input)
dt/dt = 1
dx/dt = c_0*x*y
dy/dt = c_1*(-x + 1)*(-y + 1)
dc_0/dt = 0
dc_1/dt = 0
```

```
>>> _input = {'y':  'c_0*x*y', 'z': 'c_1*(1-y)*z**2'}
>>> _input = {sympy.Symbol(k): sympy.sympify(v) for k, v in _input.
↪iteritems()}
>>> ODESystem.from_dict(_input, indep_var=sympy.Symbol('x'))
dx/dx = 1
dy/dx = c_0*x*y
dz/dx = c_1*z**2*(-y + 1)
```

(continues on next page)

(continued from previous page)

```
dc_0/dx = 0
dc_1/dx = 0
```

classmethod from_equations(*equations*, *indep_var=t*, *initial_conditions=None*)

Instantiate from multiple equations.

Parameters

- **equations** (*str, iter of str*) – Equations of the form "dx/dt = expr", optionally seperated by `\n`.
- **indep_var** (*sympy.Symbol*) – The independent variable, usually `t`.
- **initial_conditions** (*tuple of sympy.Symbol*) – The initial values of non-constant variables

Returns System of equations.

Return type *ODESystem*

```
>>> eqns = ['dx/dt = c_0*x*y', 'dy/dt = c_1*(1-x)*(1-y)']
>>> ODESystem.from_equations(eqns)
dt/dt = 1
dx/dt = c_0*x*y
dy/dt = c_1*(-x + 1)*(-y + 1)
dc_0/dt = 0
dc_1/dt = 0
>>> eqns = '\n'.join(['dy/dx = c_0*x*y', 'dz/dx = c_1*(1-y)*z**2'])
>>> ODESystem.from_equations(eqns, indep_var=sympy.Symbol('x'))
dx/dx = 1
dy/dx = c_0*x*y
dz/dx = c_1*z**2*(-y + 1)
dc_0/dx = 0
dc_1/dx = 0
```

classmethod from_tex(*tex*)

Given the LaTeX of a system of differential equations, return a ODESystem of it.

Parameters **tex** (*str*) – LaTeX

Returns System of ODEs.

Return type *ODESystem*

```
>>> eqns = ['\frac{dE}{dt} &= - k_1 E S + k_{-1} C + k_2 C \\',
... '\frac{dS}{dt} &= - k_1 E S + k_{-1} C \\',
... '\frac{dC}{dt} &= k_1 E S - k_{-1} C - k_2 C \\',
... '\frac{dP}{dt} &= k_2 C']
>>> ODESystem.from_tex('\n'.join(eqns))
dt/dt = 1
dC/dt = -C*k_2 - C*k_m1 + E*S*k_1
dE/dt = C*k_2 + C*k_m1 - E*S*k_1
dP/dt = C*k_2
dS/dt = C*k_m1 - E*S*k_1
dk_1/dt = 0
dk_2/dt = 0
dk_m1/dt = 0
```

Todo:

- Allow initial conditions to be set from tex.

---

indep_var
Return the independent variable.

Returns The independent variable, which we are differentiating with respect to.

Return type sympy.Symbol

indep_var_index
Return the independent variable index.

Returns The index of *indep_var* in `variables`.

Return type int

initial_conditions
initial-value mapping.

Returns Keys are non-constant variables, value is the constant representing their initial condition.

Return type dict

Type Return a variable

maximal_scaling_matrix()
Determine the maximal scaling matrix leaving this system invariant.

Returns Maximal scaling matrix.

Return type sympy.Matrix

```
>>> eqns = '\n'.join(['ds/dt = -k_1*e_0*s + (k_1*s + k_m1)*c',
... 'dc/dt = k_1*e_0*s - (k_1*s + k_m1 + k_2)*c'])
>>> system = ODESystem.from_equations(eqns)
>>> system.maximal_scaling_matrix()
Matrix([
[1, 0, 0, 0, -1, -1, -1],
[0, 1, 1, 1, -1,  0,  0]])
```

non_constant_variables
Return the non-constant variables - specifically those which have a derivative that isn't None or 1.

Returns The constant variables.

Return type tuple

```
>>> _input = {'x': 'c_0*x*y', 'y': 'c_1*(1-x)*(1-y)*t'}
>>> _input = {sympy.Symbol(k): sympy.sympify(v) for k, v in _input.
↪iteritems()}
>>> system = ODESystem.from_dict(_input)
>>> system.non_constant_variables
(x, y)
```

num_constants
Return the number of constant variables - specifically those which have a `None` derivative

Returns Number of non-constant variables.

Return type int

power_matrix()

Determine the 'exponent' or 'power' matrix of the system, denoted by K in the literature, by gluing together the power matrices of each derivative.

In particular, it concatenates $K_{\left(\frac{t}{x}\cdot\frac{dx}{dt}\right)}$ for x in *variables*, where t is the independent variable.

```
>>> eqns = '\n'.join(['ds/dt = -k_1*e_0*s + (k_1*s + k_m1)*c',
... 'dc/dt = k_1*e_0*s - (k_1*s + k_m1 + k_2)*c'])
>>> system = ODESystem.from_equations(eqns)
>>> system.variables
(t, c, s, e_0, k_1, k_2, k_m1)
>>> system.power_matrix()
Matrix([
[1, 1, 1,  1, 1, 1,  1],
[0, 0, 0, -1, 0, 1,  1],
[1, 0, 0,  1, 0, 0, -1],
[0, 0, 0,  1, 1, 0,  0],
[1, 0, 0,  1, 1, 1,  0],
[0, 1, 0,  0, 0, 0,  0],
[0, 0, 1,  0, 0, 0,  1]])
```

While we get a different answer to the example in the paper, this is just due to choosing our reference exponent in a different way.

Todo:

- Change the code to agree with the paper.

```
>>> system.update_initial_conditions({'s': 's_0'})
>>> system.power_matrix()
Matrix([
[1, 1, 1,  1, 1, 1,  1,  0],
[0, 0, 0, -1, 0, 1,  1,  0],
[1, 0, 0,  1, 0, 0, -1,  1],
[0, 0, 0,  1, 1, 0,  0,  0],
[1, 0, 0,  1, 1, 1,  0,  0],
[0, 1, 0,  0, 0, 0,  0,  0],
[0, 0, 1,  0, 0, 0,  1,  0],
[0, 0, 0,  0, 0, 0,  0, -1]])
```

reorder_variables(*variables*)

Reorder the equation according to the new order of variables.

Parameters **variables** (*str, iter*) – Another ordering of the variables.

```
>>> eqns = ['dz_1/dt = z_1*z_3', 'dz_2/dt = z_1*z_2 / (z_3 ** 2)']
>>> system = ODESystem.from_equations('\n'.join(eqns))
>>> system.variables
(t, z_1, z_2, z_3)
>>> system.derivatives
[1, z_1*z_3, z_1*z_2/z_3**2, 0]
```

```
>>> system.reorder_variables(['z_2', 'z_3', 't', 'z_1'])
>>> system.variables
(z_2, z_3, t, z_1)
>>> system.derivatives
[z_1*z_2/z_3**2, 0, 1, z_1*z_3]
```

to_tex()

Returns TeX representation.

Return type str

```
>>> eqns = ['dC/dt = -C*k_2 - C*k_m1 + E*S*k_1',
... 'dE/dt = C*k_2 + C*k_m1 - E*S*k_1',
... 'dP/dt = C*k_2',
... 'dS/dt = C*k_m1 - E*S*k_1']
>>> system = ODESystem.from_equations('\n'.join(eqns))
>>> print system.to_tex()
\frac{dt}{dt} &= 1 \\
\frac{dC}{dt} &= - C k_{2} - C k_{-1} + E S k_{1} \\
\frac{dE}{dt} &= C k_{2} + C k_{-1} - E S k_{1} \\
\frac{dP}{dt} &= C k_{2} \\
\frac{dS}{dt} &= C k_{-1} - E S k_{1} \\
\frac{dk_{1}}{dt} &= 0 \\
\frac{dk_{2}}{dt} &= 0 \\
\frac{dk_{-1}}{dt} &= 0
```

```
>>> system.update_initial_conditions({'C': 'C_0'})
>>> print system.to_tex()
\frac{dt}{dt} &= 1 \\
\frac{dC}{dt} &= - C k_{2} - C k_{-1} + E S k_{1} \\
\frac{dE}{dt} &= C k_{2} + C k_{-1} - E S k_{1} \\
\frac{dP}{dt} &= C k_{2} \\
\frac{dS}{dt} &= C k_{-1} - E S k_{1} \\
\frac{dk_{1}}{dt} &= 0 \\
\frac{dk_{2}}{dt} &= 0 \\
\frac{dk_{-1}}{dt} &= 0 \\
\frac{dC_{0}}{dt} &= 0 \\
C\left(0\right) &= C_{0}
```

```
>>> system.add_constraints('K_m', '(k_m1 + k_2) / k_1')
>>> print system.to_tex()
\frac{dt}{dt} &= 1 \\
\frac{dC}{dt} &= - C k_{2} - C k_{-1} + E S k_{1} \\
\frac{dE}{dt} &= C k_{2} + C k_{-1} - E S k_{1} \\
\frac{dP}{dt} &= C k_{2} \\
\frac{dS}{dt} &= C k_{-1} - E S k_{1} \\
\frac{dk_{1}}{dt} &= 0 \\
\frac{dk_{2}}{dt} &= 0 \\
\frac{dk_{-1}}{dt} &= 0 \\
\frac{dC_{0}}{dt} &= 0 \\
\frac{dK_{m}}{dt} &= 0 \\
C\left(0\right) &= C_{0} \\
K_{m} &= \frac{1}{k_{1}} \left(k_{2} + k_{-1}\right)
```

update_initial_conditions(*initial_conditions*)

Update the internal record of initial conditions.

Parameters **initial_conditions** (*dict*) – non-constant variable: initial value constant.

```
>>> _input = {'x': 'c_0*x*y', 'y': 'c_1*(1-x)*(1-y)*t'}
>>> _input = {sympy.Symbol(k): sympy.sympify(v) for k, v in _input.
↪iteritems()}
>>> system = ODESystem.from_dict(_input)
```

(continues on next page)

(continued from previous page)

```
>>> system.update_initial_conditions({'x': 'x_0'})
>>> system.initial_conditions
{x: x_0}
```

```
>>> system.update_initial_conditions({'c_0': 'k'})
Traceback (most recent call last):
    ...
ValueError: Cannot set initial condition k for variable c_0 with derivative␣
↪None.
```

```
>>> system
dt/dt = 1
dx/dt = c_0*x*y
dy/dt = c_1*t*(-x + 1)*(-y + 1)
dc_0/dt = 0
dc_1/dt = 0
dx_0/dt = 0
x(0) = x_0
```

variables
Return the variables appearing in the system.

Returns Ordered tuple of variables appearing in the system.

Return type tuple

desr.ode_system.**maximal_scaling_matrix**(*exprs*, *variables=None*)
Determine the maximal scaling matrix leaving this system invariant, in row Hermite normal form.

Parameters

- **exprs** (*iter*) – Iterable of sympy.Expressions.
- **variables** – An ordering on the variables. If None, sort according to the string representation.

Returns sympy.Matrix

```
>>> exprs = ['z_1*z_3', 'z_1*z_2 / (z_3 ** 2)']
>>> exprs = map(sympy.sympify, exprs)
>>> maximal_scaling_matrix(exprs)
Matrix([[1, -3, -1]])
```

```
>>> exprs = ['(z_1 + z_2**2) / z_3']
>>> exprs = map(sympy.sympify, exprs)
>>> maximal_scaling_matrix(exprs)
Matrix([[2, 1, 2]])
```

desr.ode_system.**parse_de**(*diff_eq*, *indep_var='t'*)
Parse a first order ordinary differential equation and return (variable of derivative, rational function

```
>>> parse_de('dn/dt = n( r(1 - n/K) - kp/(n+d) )')
(n, n(-kp/(d + n) + r(1 - n/K)))
```

```
>>> parse_de('dp/dt==sp(1 - hp / n)')
(p, sp(-hp/n + 1))
```

desr.ode_system.**rational_expr_to_power_matrix**(*expr*, *variables*)

Take a rational expression and determine the power matrix wrt an ordering on the variables, as on page 497 of Hubert-Labahn.

```
>>> exprs = map(sympy.sympify, "n*( r*(1 - n/K) - k*p/(n+d) );s*p*(1 - h*p / n)".
↪split(';'))
>>> variables = sorted(expressions_to_variables(exprs), key=str)
>>> variables
[K, d, h, k, n, p, r, s]
>>> rational_expr_to_power_matrix(exprs[0], variables)
Matrix([
[0, -1, -1, 0, 0,  0],
[0,  1,  0, 1, 0,  1],
[0,  0,  0, 0, 0,  0],
[1,  0,  0, 0, 0,  0],
[0,  1,  2, 0, 1, -1],
[1,  0,  0, 0, 0,  0],
[0,  1,  1, 1, 1,  0],
[0,  0,  0, 0, 0,  0]])
```

```
>>> rational_expr_to_power_matrix(exprs[1], variables)
Matrix([
[ 0, 0],
[ 0, 0],
[ 1, 0],
[ 0, 0],
[-1, 0],
[ 2, 1],
[ 0, 0],
[ 1, 1]])
```

7.3 desr.ode_translation module

class desr.ode_translation.**ODETranslation**(*scaling_matrix*, *variables_domain=None*, *hermite_multiplier=None*)

Bases: object

An object used for translating between systems of ODEs according to a scaling matrix. The key data are *scaling_matrix* and *herm_mult*, which together contain all the information needed (along with a variable order) to reduce an *ODESystem*.

Parameters

- **scaling_matrix** (*sympy.Matrix*) – Matrix that defines the torus action on the system.
- **variables_domain** (*iter of sympy.Symbol, optional*) – An ordering of the variables we expect to act upon. If this is not given, we will act on a system according to the position of the variables in *variables*, as long as there is the correct number of variables.
- **hermite_multiplier** (*sympy.Matrix, optional*) – User-defined Hermite multiplier, that puts *scaling_matrix* into column Hermite normal form. If not given, the normal Hermite multiplier will be calculated.

auxiliaries(*variables=None*)

The auxiliary variables of the system, as determined by *herm_mult_i*.

Returns

The auxiliary variables of the system, in terms of *variables_domain* if not None. Otherwise, use variables and failing that use x_i.

Return type tuple

```
>>> equations = ['dn/dt = n*( r*(1 - n/K) - k*p/(n+d) )','dp/dt = s*p*(1 -␣
↪h*p / n)']
>>> system = ODESystem.from_equations(equations)
>>> translation = ODETranslation.from_ode_system(system)
>>> translation.auxiliaries()
Matrix([[1/s, d, d*s/k]])
```

dep_var_herm_mult (*indep_var_index=0*)

Parameters **indep_var_index** (*int*) – The index of the independent variable.

Returns The Hermite multiplier V, ignoring the independent variable.

Return type sympy.Matrix

```
>>> translation = ODETranslation(sympy.Matrix(range(12)).reshape(3, 4))
>>> translation.herm_mult
Matrix([
[ 0,  0,  1,  0],
[ 0,  0,  0,  1],
[-1,  3, -3, -2],
[ 1, -2,  2,  1]])
>>> translation.dep_var_herm_mult(1)
Matrix([
[ 0,  0,  1],
[-1,  3, -3],
[ 1, -2,  2]])
```

dep_var_inv_herm_mult (*indep_var_index=0*)

Parameters **indep_var_index** (*int*) – The index of the independent variable.

Returns The inverse Hermite multiplier W, ignoring the independent variable.

Return type sympy.Matrix

```
>>> translation = ODETranslation(sympy.Matrix(range(12)).reshape(3, 4))
>>> translation.inv_herm_mult
Matrix([
[0, 1, 2, 3],
[1, 1, 1, 1],
[1, 0, 0, 0],
[0, 1, 0, 0]])
>>> translation.dep_var_inv_herm_mult(1)
Matrix([
[0, 2, 3],
[1, 1, 1],
[1, 0, 0]])
```

extend_from_invariants (*invariant_choice*)

Extend a given set of invariants, expressed as a matrix of exponents, to find a Hermite multiplier that will rewrite the system in terms of invariant_choice.

Parameters **invariant_choice** (*sympy.Matrix*) – The $n \times k$ matrix representing the invariants, where n is the number of variables and k is the number of given invariants.

Returns An ODETranslation representing the rewrite rules in terms of the given invariants.

Return type *ODETranslation*

```
>>> variables = sympy.symbols(' '.join(['y{}'.format(i) for i in xrange(6)]))
>>> ode_translation = ODETranslation(sympy.Matrix([[1, 0, 3, 0, 2, 2],
...                                                [0, 2, 0, 1, 0, 1],
...                                                [2, 0, 0, 3, 0, 0]]))
>>> ode_translation.invariants(variables=variables)
Matrix([[y0**3*y2*y5**2/(y3**2*y4**5), y1*y4**2/y5**2, y2**2/y4**3]])
```

Now we can express two new invariants, which we think are more interesting, as a matrix. We pick the product of the first two invariants, and the product of the last two invariants: `y0**3 * y1 * y2/(y3**2 * y4**3)` and `y1 * y2**2/(y4 * y5**2)`

```
>>> new_inv = sympy.Matrix([[3, 1, 1, -2, -3, 0],
...                         [0, 1, 2, 0, -1, -2]]).T
```

```
>>> new_translation = ode_translation.extend_from_invariants(new_inv)
>>> new_translation.invariants(variables=variables)
Matrix([[y0**3*y1*y2/(y3**2*y4**3), y1*y2**2/(y4*y5**2), y1*y4**2/y5**2]])
```

classmethod from_ode_system(*ode_system*)

Create a *ODETranslation* given an `ode_system.ODESystem` instance, by taking the maximal scaling matrix.

Parameters **ode_system** (ODESystem) –

Return type *ODETranslation*

herm_form

The Hermite normal form of the scaling matrix: $H = AV$.

Type Returns

Type sympy.Matrix

herm_mult

A column Hermite multiplier V that puts the *scaling_matrix* in column Hermite normal form. That is: $AV = H$ is in column Hermite normal form.

Type Returns

Type sympy.Matrix

herm_mult_i

V_{i}: the first r columns of V.

The columns represent the auxiliary variables of the reduction.

Type Returns

Type sympy.Matrix

herm_mult_n

V_{n}: the last $n - r$ columns of the Hermite multiplier V. The columns represent the invariants of the scaling action.

Type Returns

Type sympy.Matrix

inv_herm_mult

The inverse of the Hermite multiplier $W = V^{-1}$.

Type Returns

Type sympy.Matrix

inv_herm_mult_d

$W_{\mathfrak{d}}$: the last $n - r$ rows of W.

Type Returns

Type sympy.Matrix

inv_herm_mult_u

$W_{\mathfrak{u}}$: the first r rows of W.

Type Returns

Type sympy.Matrix

invariants(*variables=None*)

The invariants of the system, as determined by *herm_mult_n*.

Returns

The invariants of the system, in terms of *variables_domain* if not None.
Otherwise, use `variables` and failing that use y_i.

Return type tuple

```
>>> equations = ['dn/dt = n*( r*(1 - n/K) - k*p/(n+d) )','dp/dt = s*p*(1 -␣
↪h*p / n)']
>>> system = ODESystem.from_equations(equations)
>>> translation = ODETranslation.from_ode_system(system)
>>> translation.invariants()
Matrix([[s*t, n/d, k*p/(d*s), K/d, h*s/k, r/s]])
```

is_invariant_expr(*expr*, *variable_order=None*)

Determine whether an expression is an invariant or not

Parameters

- **expr** (*sympy.Expr*) – A symbolic expression which may or may not be an invariant under the scaling action.
- **variable_order** (*list, tuple*) – An ordered list that determines how the matrix acts on the variables.

Returns True if the expression is an invariant.

Return type bool

moving_frame(*variables=None*)

Given a finite-dimensional Lie group G acting on a manifold M, a moving frame is defined as a G-equivariant map $\rho : M \to G$.

We can construct a moving frame given a rational section, by sending a point $x \in M$ to the Lie group element $\rho(x)$ such that $\rho(x) \cdot x$ lies on K.

```
>>> equations = 'dz1/dt = z1*(1+z1*z2);dz2/dt = z2*(1/t - z1*z2)'.split(';')
>>> system = ODESystem.from_equations(equations)
>>> translation = ODETranslation.from_ode_system(system)
>>> moving_frame = translation.moving_frame()
>>> moving_frame
(z2,)
```

Now we can check that the moving frame moves a point to the cross-section.

```
>>> translation.rational_section()
Matrix([[1 - 1/z2]])
>>> scale_action(moving_frame, translation.scaling_matrix)  # \rho(x)
Matrix([[1, z2, 1/z2]])
>>> # :math:`\rho(x).x` should satisfy the equations of the rational section.
>>> scale_action(moving_frame, translation.scaling_matrix).multiply_
↪elementwise(sympy.Matrix(system.variables).T)
Matrix([[t, z1*z2, 1]])
```

Another example:

```
>>> equations = ['dn/dt = n*( r*(1 - n/K) - k*p/(n+d) )','dp/dt = s*p*(1 -␣
↪h*p / n)']
>>> system = ODESystem.from_equations(equations)
>>> translation = ODETranslation.from_ode_system(system)
>>> moving_frame = translation.moving_frame()
>>> moving_frame
(s, 1/d, k/(d*s))
```

Now we can check that the moving frame moves a point to the cross-section.

```
>>> translation.rational_section()
Matrix([[1 - 1/s, d - 1, d*s - 1/k]])
>>> scale_action(moving_frame, translation.scaling_matrix)  # \rho(x)
Matrix([[s, 1/d, k/(d*s), 1/d, 1/d, s/k, 1/k, 1/s, 1/s]])
>>> # :math:`\rho(x).x` should satisfy the equations of the rational section.
>>> scale_action(moving_frame, translation.scaling_matrix).multiply_
↪elementwise(sympy.Matrix(system.variables).T)
Matrix([[s*t, n/d, k*p/(d*s), K/d, 1, h*s/k, 1, r/s, 1]])
```

See [FO99] for more details.

multiplier_add_columns(*i*, *j*, *alpha*)

Add column j alpha times to the ith column of the Hermite multiplier: C_i <- C_i + alpha * C_j Check that we are only operating with 0 columns of the scaling matrix.

Parameters

- **i** (*int*) – Column index
- **j** (*int*) – Column index
- **alpha** (*int*) – Coefficient for addition

```
>>> translation = ODETranslation(sympy.Matrix([[1, 0, 1, 1, -1],
...                                            [0, 1, 0, -1, 1]]))
>>> translation.herm_form
```

(continues on next page)

(continued from previous page)

```
Matrix([
[1, 0, 0, 0, 0],
[0, 1, 0, 0, 0]])
>>> translation.herm_mult
Matrix([
[0, 0,  1,  0, 0],
[0, 0,  0,  1, 0],
[1, 1, -1, -1, 0],
[0, 0,  0,  0, 1],
[0, 1,  0, -1, 1]])
>>> translation.inv_herm_mult
Matrix([
[1, 0, 1,  1, -1],
[0, 1, 0, -1,  1],
[1, 0, 0,  0,  0],
[0, 1, 0,  0,  0],
[0, 0, 0,  1,  0]])
>>> translation.multiplier_add_columns(2, 3, 1)
```

```
>>> translation.herm_mult
Matrix([
[0, 0,  1,  0, 0],
[0, 0,  1,  1, 0],
[1, 1, -2, -1, 0],
[0, 0,  0,  0, 1],
[0, 1, -1, -1, 1]])
```

Check we have recalculated the inverse after a column operation.

```
>>> translation.inv_herm_mult
Matrix([
[ 1, 0, 1,  1, -1],
[ 0, 1, 0, -1,  1],
[ 1, 0, 0,  0,  0],
[-1, 1, 0,  0,  0],
[ 0, 0, 0,  1,  0]])
```

```
>>> translation.multiplier_add_columns(1, 3, 1)
Traceback (most recent call last):
    ...
ValueError: Cannot swap non-zero column 1
```

```
>>> translation.multiplier_add_columns(3, 3, 1)
Traceback (most recent call last):
    ...
ValueError: Cannot add column 3 to itself
```

Sympy's natural column accessing is unhappy with negative indices, so make sure we don't pick up any bad habits

```
>>> translation = ODETranslation(sympy.Matrix([[1, 0, 1, 0], [0, 1, 0, 1]]))
>>> translation.herm_mult
Matrix([
[0, 0,  1,  0],
[0, 0,  0,  1],
```

(continues on next page)

(continued from previous page)

```
[1, 0, -1,  0],
[0, 1,  0, -1]])
>>> translation.multiplier_add_columns(2, -1, 1)
>>> translation.herm_mult
Matrix([
[0, 0,  1,  0],
[0, 0,  1,  1],
[1, 0, -1,  0],
[0, 1, -1, -1]])
```

multiplier_negate_column(*i*)

Negate column i: C_i <- -C_i Check that we are only operating with 0 columns of the scaling matrix.

Parameters **i** (*int*) – Column index

```
>>> translation = ODETranslation(sympy.Matrix([[1, 0, 1, 1, -1],
...                                            [0, 1, 0, -1, 1]]))
>>> translation.herm_form
Matrix([
[1, 0, 0, 0, 0],
[0, 1, 0, 0, 0]])
>>> translation.herm_mult
Matrix([
[0, 0,  1,  0, 0],
[0, 0,  0,  1, 0],
[1, 1, -1, -1, 0],
[0, 0,  0,  0, 1],
[0, 1,  0, -1, 1]])
>>> translation.inv_herm_mult
Matrix([
[1, 0, 1,  1, -1],
[0, 1, 0, -1,  1],
[1, 0, 0,  0,  0],
[0, 1, 0,  0,  0],
[0, 0, 0,  1,  0]])
>>> translation.multiplier_negate_column(3)
```

```
>>> translation.herm_mult
Matrix([
[0, 0,  1,  0, 0],
[0, 0,  0, -1, 0],
[1, 1, -1,  1, 0],
[0, 0,  0,  0, 1],
[0, 1,  0,  1, 1]])
```

Check we have recalculated the inverse after a column operation.

```
>>> translation.inv_herm_mult
Matrix([
[1,  0, 1,  1, -1],
[0,  1, 0, -1,  1],
[1,  0, 0,  0,  0],
[0, -1, 0,  0,  0],
[0,  0, 0,  1,  0]])
```

```
>>> translation.multiplier_negate_column(1)
Traceback (most recent call last):
    ...
ValueError: Cannot negate non-zero column 1
```

Sympy's natural column accessing is unhappy with negative indices, so make sure we don't pick up any bad habits

```
>>> translation = ODETranslation(sympy.Matrix([[1, 0, 1], [0, 1, 0]]))
>>> translation.herm_mult
Matrix([
[0, 0,  1],
[0, 1,  0],
[1, 0, -1]])
>>> translation.multiplier_negate_column(-1)
>>> translation.herm_mult
Matrix([
[0, 0, -1],
[0, 1,  0],
[1, 0,  1]])
```

multiplier_swap_columns(*i*, *j*)

Swap columns i and j of the Hermite multiplier, checking that we are only swapping 0 columns of the scaling matrix.

Parameters

- **i** (*int*) – Column index
- **j** (*int*) – Column index

```
>>> translation = ODETranslation(sympy.Matrix([[1, 0, 1, 1, -1],
...                                            [0, 1, 0, -1, 1]]))
>>> translation.herm_form
Matrix([
[1, 0, 0, 0, 0],
[0, 1, 0, 0, 0]])
>>> translation.herm_mult
Matrix([
[0, 0,  1,  0, 0],
[0, 0,  0,  1, 0],
[1, 1, -1, -1, 0],
[0, 0,  0,  0, 1],
[0, 1,  0, -1, 1]])
>>> translation.inv_herm_mult
Matrix([
[1, 0, 1,  1, -1],
[0, 1, 0, -1,  1],
[1, 0, 0,  0,  0],
[0, 1, 0,  0,  0],
[0, 0, 0,  1,  0]])
```

```
>>> translation.multiplier_swap_columns(2, 3)
```

```
>>> translation.herm_mult
Matrix([
[0, 0,  0,  1, 0],
```

(continues on next page)

(continued from previous page)

```
[0, 0,  1,  0, 0],
[1, 1, -1, -1, 0],
[0, 0,  0,  0, 1],
[0, 1, -1,  0, 1]])
```

Check we have recalculated the inverse after a column operation.

```
>>> translation.inv_herm_mult
Matrix([
[1, 0, 1,  1, -1],
[0, 1, 0, -1,  1],
[0, 1, 0,  0,  0],
[1, 0, 0,  0,  0],
[0, 0, 0,  1,  0]])
```

```
>>> translation.multiplier_swap_columns(1, 3)
Traceback (most recent call last):
    ...
ValueError: Cannot swap non-zero column 1
```

```
>>> translation.multiplier_swap_columns(2, 6)
Traceback (most recent call last):
    ...
IndexError: Index out of range: a[6]
```

Sympy's natural column accessing is unhappy with negative indices, so make sure we don't pick up any bad habits

```
>>> translation = ODETranslation(sympy.Matrix([[1, 0, 1, 0], [0, 1, 0, 1]]))
>>> translation.herm_mult
Matrix([
[0, 0,  1,  0],
[0, 0,  0,  1],
[1, 0, -1,  0],
[0, 1,  0, -1]])
>>> translation.multiplier_swap_columns(2, -1)
>>> translation.herm_mult
Matrix([
[0, 0,  0,  1],
[0, 0,  1,  0],
[1, 0,  0, -1],
[0, 1, -1,  0]])
```

n

The number of original variables that the scaling action is acting on: n**.** In particular, it is the number of columns of *scaling_matrix*.

Type Returns

Type int

r

The dimension of the scaling action: r**.** In particular it is the number of rows of *scaling_matrix*.

Type Returns

Type int

rational_section (*variables=None*)

Given a finite-dimensional Lie group G acting on a manifold M, a cross-section $K \subset M$ is a subset that intersects each orbit of M in exactly one place.

Parameters **variables** (*iter of sympy.Symbol, optional*) – Variables of the system.

Returns A set of equations defining a variety that is a cross-section.

Return type sympy.Matrix

```
>>> equations = 'dz1/dt = z1*(1+z1*z2);dz2/dt = z2*(1/t - z1*z2)'.split(';')
>>> system = ODESystem.from_equations(equations)
>>> translation = ODETranslation.from_ode_system(system)
>>> translation.rational_section()
Matrix([[1 - 1/z2]])
```

```
>>> equations = ['dn/dt = n*( r*(1 - n/K) - k*p/(n+d) )','dp/dt = s*p*(1 -␣
↪h*p / n)']
>>> system = ODESystem.from_equations(equations)
>>> translation = ODETranslation.from_ode_system(system)
>>> translation.rational_section()
Matrix([[1 - 1/s, d - 1, d*s - 1/k]])
```

See [FO99] for more details.

reverse_translate (*variables*)

Given the solutions of a reduced system, reverse translate them into solutions of the original system.

Note: This *doesn't* reverse translate the parameter reduction scheme.

It *only* guesses between *reverse_translate_general()* and *reverse_translate_dep_var()*

Parameters

- **variables** (*iter of sympy.Expression*) – The solution auxiliary variables $x(t)$ and solution invariants $y(t)$ of the reduced system. Variables should be ordered auxiliary variables followed by invariants.
- **indep_var_index** (*int*) – The location of the independent variable.

Return type tuple

reverse_translate_dep_var (*variables*, *indep_var_index*)

Given the solutions of a (dep_var) reduced system, reverse translate them into solutions of the original system.

Parameters

- **variables** (*iter of sympy.Expression*) – The solution auxiliary variables $x(t)$ and solution invariants $y(t)$ of the reduced system. Variables should be ordered auxiliary variables followed by invariants.
- **indep_var_index** (*int*) – The location of the independent variable.

Return type tuple

Example 6.4 from [HL13]. We will just take the matrices from the paper, rather than use our own (which are constructed using different conventions).

$$\frac{dz_1}{dt} = z_1 (1 + z_1 z_2)$$
$$\frac{dz_2}{dt} = z_2 \left(\frac{1}{t} - z_1 z_2 \right)$$

```
>>> equations = 'dz1/dt = z1*(1+z1*z2);dz2/dt = z2*(1/t - z1*z2)'.split(';')
>>> system = ODESystem.from_equations(equations)
>>> var_order = ['t', 'z1', 'z2']
>>> system.reorder_variables(var_order)
>>> scaling_matrix = sympy.Matrix([[0, 1, -1]])
>>> hermite_multiplier = sympy.Matrix([[0, 1, 0],
...                                    [1, 0, 1],
...                                    [0, 0, 1]])
>>> translation = ODETranslation(scaling_matrix=scaling_matrix,
...                              hermite_multiplier=hermite_multiplier)
>>> reduced = translation.translate_dep_var(system)
>>> # Check reduction
>>> answer = ODESystem.from_equations('dx0/dt = x0*(y0 + 1);dy0/dt = y0*(1 +␣
↪1/t)'.split(';'))
>>> reduced == answer
True
>>> reduced
dt/dt = 1
dx0/dt = x0*(y0 + 1)
dy0/dt = y0*(1 + 1/t)
>>> t_var, c1_var, c2_var = sympy.var('t c1 c2')
>>> reduced_soln = (c2_var*sympy.exp(t_var+c1_var*(1-t_var)*sympy.exp(t_var)),
...                 c1_var * t_var * sympy.exp(t_var))
>>> original_soln = translation.reverse_translate_dep_var(reduced_soln,␣
↪system.indep_var_index)
>>> original_soln == (reduced_soln[0], reduced_soln[1] / reduced_soln[0])
True
>>> original_soln
(c2*exp(c1*(-t + 1)*exp(t) + t), c1*t*exp(t)*exp(-c1*(-t + 1)*exp(t) - t)/c2)
```

reverse_translate_general (*variables*, *system_indep_var_index=0*)

Given an iterable of variables, or exprs, reverse translate into the original variables. Here we expect t as the first variable, since we need to divide by it and substitute

Given the solutions of a (general) reduced system, reverse translate them into solutions of the original system.

Parameters

- **variables** (*iter of sympy.Expression*) – The independent variable t, the solution auxiliary variables $x(t)$ and the solution invariants $y(t)$ of the reduced system. Variables should be ordered: independent variable, auxiliary variables then invariants.
- **indep_var_index** (*int*) – The location of the independent variable.

Return type tuple

Example 6.6 from [HL13]. We will just take the matrices from the paper, rather than use our own (which

are constructed using different conventions).

$$\frac{dz_1}{dt} = \frac{z_1\left(z_1^5 z_2 - 2\right)}{3t}$$
$$\frac{dz_2}{dt} = \frac{z_2\left(10 - 2z_1^5 z_2 + \frac{3z_1^2 z_2}{t}\right)}{3t}$$

```
>>> equations = ['dz1/dt = z1*(z1**5*z2 - 2)/(3*t)',
...              'dz2/dt = z2*(10 - 2*z1**5*z2 + 3*z1**2*z2/t )/(3*t)']
>>> system = ODESystem.from_equations(equations)
>>> var_order = ['t', 'z1', 'z2']
>>> system.reorder_variables(var_order)
>>> scaling_matrix = sympy.Matrix([[3, -1, 5]])
>>> hermite_multiplier = sympy.Matrix([[1, 1, -1],
...                                    [2, 3, 2],
...                                    [0, 0, 1]])
>>> translation = ODETranslation(scaling_matrix=scaling_matrix,
...                              hermite_multiplier=hermite_multiplier)
>>> reduced = translation.translate_general(system)
>>> # Quickly check the reduction
>>> answer = ODESystem.from_equations(['dx0/dt = x0*(2*y0*y1/3 - 1/3)/t',
...                                    'dy0/dt = y0*(y0*y1 - 1)/t',
...                                    'dy1/dt = y1*(y1 + 1)/t'])
>>> reduced == answer
True
>>> reduced
dt/dt = 1
dx0/dt = x0*(2*y0*y1/3 - 1/3)/t
dy0/dt = y0*(y0*y1 - 1)/t
dy1/dt = y1*(y1 + 1)/t
```

Check reverse translation:

```
>>> reduced_soln = (sympy.var('t'),
...                 sympy.sympify('c3/(t**(1/3)*(ln(t-c1)-ln(t)+c2)**(2/3))'),
↪  # x
...                 sympy.sympify('c1/(t*(ln(t-c1)-ln(t)+c2))'),  # y1
...                 sympy.sympify('t/(c1 - t)'))  # y2
>>> original_soln = translation.reverse_translate_general(reduced_soln,␣
↪system.indep_var_index)
>>> original_soln_answer = [#reduced_soln[1] ** 3 / (reduced_soln[2] ** 2)  #␣
↪x^3 / y1^2
...                          reduced_soln[2] / reduced_soln[1],  # y1 / x
...                          reduced_soln[1] ** 5 * reduced_soln[3] / reduced_
↪soln[2] ** 4]  # x^5 y2 / y1^4
>>> original_soln_answer = tuple([soln.subs({sympy.var('t'): sympy.sympify('t␣
↪/ (c3**3 / c1**2)')})
...                                for soln in original_soln_answer])
>>> original_soln == original_soln_answer
True
>>> original_soln[0]
c1/(c3*(c1**2*t/c3**3)**(2/3)*(c2 - log(c1**2*t/c3**3) + log(c1**2*t/c3**3 -␣
↪c1))**(1/3))
>>> original_soln[1]
c3**5*(c1**2*t/c3**3)**(10/3)*(c2 - log(c1**2*t/c3**3) + log(c1**2*t/c3**3 -␣
↪c1))**(2/3)/(c1**4*(-c1**2*t/c3**3 + c1))
```

reverse_translate_parameter(*variables*)

Given the solutions of a reduced system, reverse translate them into solutions of the original system.

Parameters **variables** (*iter of sympy.Expression*) – r constants (auxiliary variables) followed by the independent, dependent and invariant constants.

Example 7.5 (Prey-predator model) from [HL13].

Use the matrices from the paper, which differ to ours as different conventions are used.

$$\frac{dn}{dt} = n\left(r\left(1 - \frac{n}{K}\right) - \frac{kp}{n+d}\right)$$
$$\frac{dp}{dt} = sp\left(1 - \frac{hp}{n}\right)$$

```
>>> equations = ['dn/dt = n*( r*(1 - n/K) - k*p/(n+d) )','dp/dt = s*p*(1 -␣
↪h*p / n)']
>>> system = ODESystem.from_equations(equations)
>>> system.variables
(t, n, p, K, d, h, k, r, s)
>>> maximal_scaling_matrix = system.maximal_scaling_matrix()
>>> # Hand craft a Hermite multiplier given the invariants from the paper.
>>> # This is done by placing the auxiliaries first, then rearranging the␣
↪invariants and reading them off
>>> # individually from the Hermite multiplier given in the paper
>>> herm_mult = sympy.Matrix([[ 0,  0,  0,  1,  0,  0,  0,  0,  0],  # t
...                           [ 0,  0,  0,  0,  1,  0,  0,  0,  0],  # n
...                           [ 0,  0,  0,  0,  0,  1,  0,  0,  0],  # p
...                           [ 0,  1,  1,  0, -1, -1, -1,  0,  0],  # K
...                           [ 0,  0,  0,  0,  0,  0,  1,  0,  0],  # d
...                           [ 0,  0, -1,  0,  0,  1,  0, -1,  0],  # h
...                           [ 0,  0,  0,  0,  0,  0,  0,  1,  0],  # k
...                           [-1,  0,  0,  1,  0,  0,  0, -1, -1],  # r
...                           [ 0,  0,  0,  0,  0,  0,  0,  0,  1]])  # s
>>> translation = ODETranslation(maximal_scaling_matrix,
...                              variables_domain=system.variables,
...                              hermite_multiplier=herm_mult)
>>> translation.translate_parameter(system)
dt/dt = 1
dn/dt = -c1*n*p/(c0 + n) - n**2 + n
dp/dt = c2*p - c2*p**2/n
dc0/dt = 0
dc1/dt = 0
dc2/dt = 0
>>> translation.translate_parameter_substitutions(system)
{k: c1, n: n, r: 1, d: c0, K: 1, h: 1, s: c2, p: p, t: t}
>>> soln_reduced = sympy.var('x1, x2, x3, t, n, p, c0, c1, c2')
>>> translation.reverse_translate_parameter(soln_reduced)
Matrix([[t*x1, n*x2, p*x3, x2, c0*x2, x2/x3, c1*x2/(x1*x3), 1/x1, c2/x1]])
```

rewrite_rules(*variables=None*)

Given a set of variables, print the rewrite rules. These are the rules that allow you to write any rational invariant in terms of the *invariants*.

Parameters **variables** (*iter of sympy.Symbol*) – The names of the variables appearing in the rational invariant.

Returns A `dict` whose keys are the given variables and whose values are the values to be substituted.

Return type dict

```
>>> equations = ['dn/dt = n*( r*(1 - n/K) - k*p/(n+d) )','dp/dt = s*p*(1 -␣
↪h*p / n)']
>>> system = ODESystem.from_equations(equations)
>>> translation = ODETranslation.from_ode_system(system)
>>> translation.rewrite_rules()
{k: 1, n: n/d, r: r/s, d: 1, K: K/d, h: h*s/k, s: 1, p: k*p/(d*s), t: s*t}
```

For example, rt is an invariant. Substituting in the above mapping gives us a way to write it in terms of our generating set of invariants:

$$rt \mapsto \left(\frac{r}{s}\right)(st) = rt$$

scaling_matrix
The scaling matrix that this translation corresponds to, often denoted A.

Type Returns

Type sympy.Matrix

to_tex()

Returns The scaling matrix A, the Hermite multiplier V and $W = V^{-1}$, in beautiful LaTeX.

Return type str

```
>>> print ODETranslation(sympy.Matrix(range(12)).reshape(3, 4)).to_tex()
A=
0 & 1 & 2 & 3 \\
4 & 5 & 6 & 7 \\
8 & 9 & 10 & 11 \\
V=
0 & 0 & 1 & 0 \\
0 & 0 & 0 & 1 \\
-1 & 3 & -3 & -2 \\
1 & -2 & 2 & 1 \\
W=
0 & 1 & 2 & 3 \\
1 & 1 & 1 & 1 \\
1 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 \\
```

translate(*system*)
Translate an `ode_system.ODESystem` into a reduced system.

First, try the simplest parameter reduction method, then the dependent variable translation (where the scaling action ignores the independent variable) and finally the general reduction scheme.

Parameters **system** (`ODESystem`) – System to reduce.

Return type *ODESystem*

translate_dep_var(*system*)
Given a system of ODEs, translate the system into a simplified version. Assume we are only working on dependent variables, not the independent variable.

Parameters **system** (`ODESystem`) – System to reduce.

Return type *ODESystem*

```
>>> equations = 'dz1/dt = z1*(1+z1*z2);dz2/dt = z2*(1/t - z1*z2)'.split(';')
>>> system = ODESystem.from_equations(equations)
>>> translation = ODETranslation.from_ode_system(system)
>>> translation.translate_dep_var(system=system)
dt/dt = 1
dx0/dt = x0*(y0 - 1/t)
dy0/dt = y0*(1 + 1/t)
```

translate_general (*system*)

The most general reduction scheme. If there are n variables (including the independent variable) then there will be a system of $n - r + 1$ invariants and r auxiliary variables.

Parameters **system** (ODESystem) – System to reduce.

Return type *ODESystem*

```
>>> equations = 'dn/dt = n*( r*(1 - n/K) - k*p/(n+d) );dp/dt = s*p*(1 - h*p /␣
↪n)'.split(';')
>>> system = ODESystem.from_equations(equations)
>>> translation = ODETranslation.from_ode_system(system)
>>> translation.translate_general(system=system)
dt/dt = 1
dx0/dt = 0
dx1/dt = 0
dx2/dt = 0
dy0/dt = y0/t
dy1/dt = y1*(-y0*y1*y5/y3 - y0*y2/(y1 + 1) + y0*y5)/t
dy2/dt = y2*(y0 - y0*y2*y4/y1)/t
dy3/dt = 0
dy4/dt = 0
dy5/dt = 0
```

translate_parameter (*system*)

Translate according to parameter scheme

translate_parameter_substitutions (*system*)

Given a system, determine the substitutions made in the parameter reduction.

Parameters **system** (ODESystem) – The system in question.

Return type dict

```
>>> equations = 'dn/dt = n*( r*(1 - n/K) - k*p/(n+d) );dp/dt = s*p*(1 - h*p /␣
↪n)'.split(';')
>>> system = ODESystem.from_equations(equations)
>>> translation = ODETranslation.from_ode_system(system)
>>> translation.translate_parameter_substitutions(system=system)
{k: 1, n: n, r: c2, d: 1, K: c0, h: c1, s: 1, p: p, t: t}
```

variables_domain

The variables that the scaling action acts on.

Type Returns

Type tuple, None

desr.ode_translation.**extend_rectangular_matrix** (*matrix_*, *check_unimodular=True*)

Given a rectangular $n \times m$ integer matrix, extend it to a unimodular one by appending columns.

Parameters **matrix** (*sympy.Matrix*) – The rectangular matrix to be extended.

Returns Square matrix of determinant 1.

Return type sympy.Matrix

```
>>> matrix_ = sympy.Matrix([[1, 0],
...                         [0, 1],
...                         [0, 0],])
>>> extend_rectangular_matrix(matrix_)
Matrix([
[1, 0, 0],
[0, 1, 0],
[0, 0, 1]])
```

By default, we will throw if we cannot extend a matrix to a unimodular matrix.

```
>>> matrix_[0, 0] = 2
>>> extend_rectangular_matrix(matrix_=matrix_)
Traceback (most recent call last):
    ...
ValueError: Unable to extend the matrix
Matrix([[2, 0], [0, 1], [0, 0]])
to a unimodular matrix.
```

We can extend some more interesting matrices.

```
>>> matrix_ = sympy.Matrix([[3, 2],
...                         [-2, 1],
...                         [5, 6],])
>>> extend_rectangular_matrix(matrix_)
Matrix([
[ 3, 2, 0],
[-2, 1, 1],
[ 5, 6, 1]])
```

desr.ode_translation.**scale_action**(*vect*, *scaling_matrix*)

Given a vector of sympy expressions, determine the action defined by scaling_matrix I.e. Given vect, calculate vect^scaling_matrix (in notation of Hubert Labahn).

Parameters

- **vect** (*iter of sympy.Expression*) – Element of the algebraic torus that is going to act on the system.
- **scaling_matrix** (*sympy.Matrix*) – Matrix that defines the action on the system.

Returns Matrix that, when multiplied element-wise with an element of the manifold, gives the result of the action.

Return type sympy.Matrix

Example 3.3

```
>>> x = sympy.var('x')
>>> A = sympy.Matrix([[2, 3]])
>>> scale_action([x], A)
Matrix([[x**2, x**3]])
```

Example 3.4

```
>>> v = sympy.var('m v')
>>> A = sympy.Matrix([[6, 0, -4, 1, 3], [0, 3, 1, -4, 3]])
>>> scale_action(v, A)
Matrix([[m**6, v**3, v/m**4, m/v**4, m**3*v**3]])
```

7.4 desr.matrix_normal_forms module

desr.matrix_normal_forms.**element_wise_lt**(*matrix_*, *other*)

Given a rectangular $n \times m$ integer matrix, return a matrix of bools with (i, j)th entry equal to matrix_[i,j] < other or other[i,j], depending on the type of other.

Parameters

- **matrix** (*sympy.Matrix*) – The input matrix
- **other** (*sympy.Matrix, int*) – Value(s) to compare it to.

Returns $n \times m$ Boolean matrix

Return type sympy.Matrix

```
>>> x = sympy.eye(2) * 3
>>> element_wise_lt(x, 2)
Matrix([
[False,  True],
[ True, False]])
>>> element_wise_lt(x, sympy.Matrix([[4, -1], [1, 1]]))
Matrix([
[True, False],
[True, False]])
```

desr.matrix_normal_forms.**expand_matrix**(*matrix_*)

Given a rectangular $n \times m$ integer matrix, return an $(n+1) \times (m+1)$ matrix where the extra row and column are 0 except on the first entry which is 1.

Parameters **matrix** (*sympy.Matrix*) – The rectangular matrix to be expanded

Returns An $(n+1) \times (m+1)$ matrix

Return type sympy.Matrix

```
>>> matrix_ = sympy.diag(*[1, 1, 2])
>>> expand_matrix(matrix_)
Matrix([
[1, 0, 0, 0],
[0, 1, 0, 0],
[0, 0, 1, 0],
[0, 0, 0, 2]])
```

```
>>> matrix_ = sympy.Matrix([1])
>>> expand_matrix(matrix_)
Matrix([
[1, 0],
[0, 1]])
```

```
>>> matrix_ = sympy.Matrix([[0]])
>>> expand_matrix(matrix_)
Matrix([
[1, 0],
[0, 0]])
```

```
>>> matrix_ = sympy.Matrix([[]])
>>> expand_matrix(matrix_)
Matrix([[1]])
```

```
>>> matrix_ = sympy.Matrix([[1, 2, 3]])
>>> expand_matrix(matrix_)
Matrix([
[1, 0, 0, 0],
[0, 1, 2, 3]])
>>> expand_matrix(matrix_.T)
Matrix([
[1, 0],
[0, 1],
[0, 2],
[0, 3]])
```

desr.matrix_normal_forms.**get_pivot_row_indices**(*matrix_*)

Return the pivot indices of the matrix

Parameters **matrix** (*sympy.Matrix*) – The matrix in question.

Returns A list of indices, where the pivot entry of row i is [i, indices[i]]

Return type indices (list)

```
>>> matrix_ = sympy.eye(3)
>>> get_pivot_row_indices(matrix_)
[0, 1, 2]
```

```
>>> matrix_ = sympy.Matrix([[1, 0, 0],[0, 0, 1]])
>>> get_pivot_row_indices(matrix_)
[0, 2]
```

desr.matrix_normal_forms.**hnf_col**(*matrix_*)

Default function for calculating column Hermite normal forms.

Parameters **matrix** (*sympy.Matrix*) – Input matrix.

Returns

Tuple containing: hermite_normal_form (sympy.Matrix): The column Hermite normal form of matrix_.

normal_multiplier (sympy.Matrix): The normal Hermite multiplier.

Return type tuple

Return type (sympy.Matrix, sympy.Matrix)

desr.matrix_normal_forms.**hnf_col_lll**(*matrix_*)

Compute the Hermite normal form, acts on the COLUMNS of a matrix. The Havas, Majewski, Matthews LLL method is used. We usually take $\alpha = \frac{m_1}{n_1}$, with $(m_1, n_1) = (1, 1)$ to get best results.

Parameters **matrix** (*sympy.Matrix*) – Integer mxn matrix, nonzero, at least two rows

Returns hnf (sympy.Matrix): The column Hermite normal form of matrix_. unimod (sympy.Matrix): Unimodular matrix representing the column actions.

Return type tuple

Return type (sympy.Matrix, sympy.Matrix)

```
>>> A = sympy.Matrix([[8, 2, 15, 9, 11],
...                   [6, 0, 6, 2, 3]])
>>> h, v = hnf_col(A)
>>> h
Matrix([
[1, 0, 0, 0, 0],
[0, 1, 0, 0, 0]])
>>> v
Matrix([
[-1,  0,  0, -1, -1],
[-3, -1,  6, -1,  0],
[ 1,  0,  1,  1,  2],
[ 0, -1, -3, -3,  0],
[ 0,  1,  0,  2, -2]])
 >>> A * v == h
 True
```

desr.matrix_normal_forms.**hnf_row**(*matrix_*)

Default function for calculating row Hermite normal forms.

Parameters **matrix** (*sympy.Matrix*) – Input matrix.

Returns

hermite_normal_form (sympy.Matrix): The column Hermite normal form of matrix_.

normal_multiplier (sympy.Matrix): The normal Hermite multiplier.

Return type tuple

Return type (sympy.Matrix, sympy.Matrix)

desr.matrix_normal_forms.**hnf_row_lll**(*matrix_*)

Compute the Hermite normal form, acts on the ROWS of a matrix. The Havas, Majewski, Matthews LLL method is used. We usually take $\alpha = \frac{m_1}{n_1}$, with $(m_1, n_1) = (1, 1)$ to get best results.

Parameters **matrix** (*sympy.Matrix*) – Integer mxn matrix, nonzero, at least two rows

Returns

hnf (sympy.Matrix): The row Hermite normal form of matrix_.

unimod (sympy.Matrix): Unimodular matrix representing the row actions.

Return type tuple

Return type (sympy.Matrix, sympy.Matrix)

```
>>> matrix_ = sympy.Matrix([[2, 0],
...                         [3, 3],
...                         [0, 0]])
>>> result = hnf_row_lll(matrix_)
>>> result[0]
Matrix([
[1, 3],
[0, 6],
```

(continues on next page)

(continued from previous page)

```
[0, 0]])
>>> result[1]
Matrix([
[-1, 1, 0],
[-3, 2, 0],
[ 0, 0, 1]])
>>> result[1] * matrix_ == result[0]
True
```

```
>>> hnf_row_lll(sympy.Matrix([[0, -2, 0]]))
(Matrix([[0, 2, 0]]), Matrix([[-1]]))
```

```
>>> matrix_ = sympy.Matrix([[0, 1, 0, 1], [-1, 0, 1, 0]])
>>> hnf_row_lll(matrix_)[0]
Matrix([
[1, 0, -1, 0],
[0, 1,  0, 1]])
```

desr.matrix_normal_forms.**is_hnf_col**(*matrix_*)

Decide whether a matrix is in row Hermite normal form, when acting on rows.

Parameters **matrix** (*sympy.Matrix*) – The matrix in question.

Returns bool

desr.matrix_normal_forms.**is_hnf_row**(*matrix_*)

Decide whether a matrix is in Hermite normal form, when acting on rows. This is according to the Havas Majewski Matthews definition.

Parameters **matrix** (*sympy.Matrix*) – The matrix in question.

Returns bool

```
>>> is_hnf_row(sympy.eye(4))
True
>>> is_hnf_row(sympy.ones(2))
False
>>> is_hnf_row(sympy.Matrix([[1, 2, 0], [0, 1, 0]]))
False
>>> is_hnf_row(sympy.Matrix([[1, 0, 0], [0, 2, 1]]))
True
>>> is_hnf_row(sympy.Matrix([[1, 0, 0], [0, -2, 1]]))
False
```

desr.matrix_normal_forms.**is_normal_hermite_multiplier**(*hermite_multiplier*, *matrix_*)

Determine whether hermite_multiplier is the normal Hermite multiplier of matrix. This is the COLUMN version.

Parameters

- **hermite_multiplier** (*sympy.Matrix*) – Candidate column normal Hermite multiplier
- **matrix** (*sympy.Matrix*) – Matrix

Returns matrix_ * hermite_multiplier is in Hermite normal form and hermite_multiplier is in normal form.

Return type bool

desr.matrix_normal_forms.**is_smf**(*matrix_*)

Given a rectangular $n \times m$ integer matrix, determine whether it is in Smith normal form or not.

Parameters **matrix** (*sympy.Matrix*) – The rectangular matrix to be decomposed

Returns True if in Smith normal form, False otherwise.

Return type bool

```
>>> matrix_ = sympy.diag(1, 1, 2)
>>> is_smf(matrix_)
True
>>> matrix_ = sympy.diag(-1, 1, 2)
>>> is_smf(matrix_)
False
>>> matrix_ = sympy.diag(2, 1, 1)
>>> is_smf(matrix_)
False
>>> matrix_ = sympy.diag(1, 2, 0)
>>> is_smf(matrix_)
True
>>> matrix_ = sympy.diag(2, 6, 0)
>>> is_smf(matrix_)
True
>>> matrix_ = sympy.diag(2, 5, 0)
>>> is_smf(matrix_)
False
>>> matrix_ = sympy.diag(0, 1, 1)
>>> is_smf(matrix_)
False
>>> matrix_ = sympy.diag(0)
>>> is_smf(sympy.diag(0)), is_smf(sympy.diag(1)), is_smf(sympy.Matrix()),
(True, True, True)
```

Check a real example:

```
>>> matrix_ = sympy.Matrix([[2, 4, 4],
...                         [-6, 6, 12],
...                         [10, -4, -16]])
>>> is_smf(matrix_)
False
```

```
>>> matrix_ = sympy.diag(2, 6, 12)
>>> is_smf(matrix_)
True
```

Check it works for non-square matrices:

```
>>> matrix_ = sympy.Matrix(4, 5, range(20))
>>> is_smf(matrix_)
False
```

```
>>> matrix_ = sympy.Matrix([[1, 0], [1, 2]])
>>> is_smf(matrix_)
False
```

desr.matrix_normal_forms.**normal_hnf_col**(*matrix_*)

Return the column HNF and the unique normal Hermite multiplier.

Parameters **matrix** (*sympy.Matrix*) – Input matrix.

Returns

Tuple containing: hermite_normal_form (sympy.Matrix): The column Hermite normal form of matrix_. normal_multiplier (sympy.Matrix): The normal Hermite multiplier.

Return type tuple

```
>>> A = sympy.Matrix([[8, 2, 15, 9, 11],
...                   [6, 0, 6, 2, 3]])
>>> h, v = normal_hnf_col(A)
>>> h
Matrix([
[1, 0, 0, 0, 0],
[0, 1, 0, 0, 0]])
>>> v
Matrix([
[  0,  0,   1,  0,   0],
[  0,  0,   0,  1,   0],
[  2,  1,   3,  1,   5],
[  9,  2,  21,  3,  21],
[-10, -3, -22, -4, -24]])
>>> (A * v) == h
True
```

desr.matrix_normal_forms.**normal_hnf_row**(*matrix_*)

Return the row HNF and the unique normal Hermite multiplier.

Parameters **matrix** (*sympy.Matrix*) – Input matrix.

Returns

Tuple containing:

hermite_normal_form (sympy.Matrix): The row Hermite normal form of matrix_. normal_multiplier (sympy.Matrix): The normal Hermite multiplier.

Return type tuple

desr.matrix_normal_forms.**smf**(*matrix_*)

Given a rectangular $n \times m$ integer matrix, calculate the Smith Normal Form S and multipliers U, V such that U

Parameters

matrix (*sympy.Matrix*) – The rectangular matrix to be decomposed.

Returns

- **S** (*sympy.Matrix*) – The Smith normal form of matrix_.
- **U** (*sympy.Matrix*) – U (the matrix representing the row operations of the decomposition).
- **V** (*sympy.Matrix*) – V (the matrix representing the column operations of the decomposition).

Return type

(sympy.Matrix, sympy.Matrix, sympy.Matrix)

```
>>> matrix_ = sympy.Matrix([[2, 4, 4],
...                         [-6, 6, 12],
...                         [10, -4, -16]])
>>> smf(matrix_)[0]
Matrix([
[2, 0,  0],
[0, 6,  0],
[0, 0, 12]])
```

```
>>> matrix_ = sympy.diag(2, 1, 0)
>>> smf(matrix_)[0]
Matrix([
[1, 0, 0],
[0, 2, 0],
[0, 0, 0]])
```

```
>>> matrix_ = sympy.diag(5, 2, 0)
>>> smf(matrix_)[0]
Matrix([
[1,  0, 0],
[0, 10, 0],
[0,  0, 0]])
```

7.5 desr.diophantine module

Diophantine is a python package for finding small solutions of systems of diophantine equations (see https://en.wikipedia.org/wiki/Diophantine_equation). It is based on PHP code by Keith Matthews (see www.number-theory.org) that implements the algorithm described in the included 'algorithm.pdf' (see http://www.numbertheory.org/lll.html for a list of associated publications).

There are two branches of this code in the GitHub repository (see https://github.com/tclose/Diophantine.git), 'master', which uses the sympy library and therefore uses arbitrarily long integer representations, and 'numpy', which uses the numpy library, which is faster but can suffer from integer overflow errors despite using int64 representations.

Diophantine is released under the MIT Licence (see Licence for details)

Author: Thomas G. Close (tom.g.close@gmail.com)

exception desr.diophantine.**NoSolutionException**
Bases: exceptions.Exception

desr.diophantine.**addr**(*a*, *b*, *c*, *d*)

desr.diophantine.**cholesky**(*A*)
A is positive definite mxm

desr.diophantine.**comparer**(*a*, *b*, *c*, *d*)
Assumes b>0 and d>0. Returns -1, 0 or 1 according as a/b <,=,> c/d+

desr.diophantine.**first_nonzero_is_negative**(*A*)
returns 0 if the first nonzero column j of A contains more than one nonzero entry, or contains only one nonzero entry and which is positive+ returns 1 if the first nonzero column j of A contains only one nonzero entry, which is negative+ This assumes A is a nonzero matrix with at least two rows+

desr.diophantine.**get_solutions**(*A*)

desr.diophantine.**gram**(*A*)
Need to check for row and column operations

desr.diophantine.**initialise_working_matrices**(*G*)
G is a nonzero matrix with at least two rows.

desr.diophantine.**introot**(*a*, *b*, *c*, *d*)
With Z=a/b, U=c/d, returns [sqrt(a/b)+c/d]. First ANSWER = [sqrt(Z)] + [U]. One then tests if Z < ([sqrt(Z)] + 1 -U)^2. If this does not hold, ANSWER += 1+ For use in fincke_pohst()+

desr.diophantine.**lcasvector**(*A*, *x*)
lcv[j]=X[1]A[1, j]=. . . +X[m]A[m, j], 1 <= j <= n+

desr.diophantine.**lllhermite**(*G*, *m1=1*, *n1=1*)
Input: integer mxn matrix A, nonzero, at least two rows+ Output: small unimodular matrix B and HNF(A), such that BA=HNF(A)+ The Havas, Majewski, Matthews LLL method is used+ We usually take alpha=m1/n1, with (m1,n1)=(1,1) to get best results+

desr.diophantine.**lnearint**(*a*, *b*)
left nearest integer returns y+1/2 if a/b=y+1/2, y integral+

desr.diophantine.**minus**(*j*, *L*)

desr.diophantine.**multr**(*a*, *b*, *c*, *d*)

desr.diophantine.**nonzero**(*m*)

desr.diophantine.**ratior**(*a*, *b*, *c*, *d*)
returns (a/b)/(c/d)

desr.diophantine.**reduce_matrix**(*A*, *B*, *L*, *k*, *i*, *D*)

desr.diophantine.**sign**(*x*)

desr.diophantine.**solve**(*A*, *b*)
Finds small solutions to systems of diophantine equations, A x = b, where A is a M x N matrix of coefficents, b is a M x 1 vector and x is the N x 1 solution vector, e.g.

```
>>> A = sympy.Matrix([[1, 0, 0, 2], [0, 2, 3, 5], [2, 0, 3, 1], [-6, -1, 0, 2],
...                   [0, 1, 1, 1], [-1, 2, 0,1], [-1, -2, 1, 0]]).T
>>> b = sympy.Matrix([1, 1, 1, 1])
>>> solve(A, b)
[Matrix([
[-1],
[ 1],
[ 0],
[ 0],
[-1],
[-1],
[-1]])]
```

The returned solution vector will tend to be one with the smallest norms. If multiple solutions with the same norm are found they will all be returned. If there are no solutions the empty list will be returned.

desr.diophantine.**subr**(*a*, *b*, *c*, *d*)

desr.diophantine.**swap_rows**(*k*, *A*, *B*, *L*, *D*)

7.6 desr.chemical_reaction_network module

class desr.chemical_reaction_network.**ChemicalReactionNetwork**(*chemical_species*, *complexes*, *reactions*)

Bases: object

Chemical reaction network, made up of Species, Complexes and Reactions.

classmethod from_diagram(*diagram*)
Given a text diagram, return an interpreted chemical reaction network.

```
>>> ChemicalReactionNetwork.from_diagram('x + y -> z \n y + z -> 2*z')
1.y + 1.x -> 1.z
1.y + 1.z -> 2.z
```

We can add reversible reactions like so:

```
>>> ChemicalReactionNetwork.from_diagram('x + y -> z \n z -> x + y')
1.y + 1.x -> 1.z
1.z -> 1.y + 1.x
```

n
The number of chemical species in the network.

Returns int

ode_equations()
Return a tuple of differential equations for each species.

Returns A differential equation, represented by a sympy.Expression, for the dynamics of each species.

Return type tuple

p
The number of complexes in the network.

Returns int

r
The number of different chemical reactions in the network.

Returns int

to_ode_system()
Generate a system of ODEs based on the current network.

Returns A system describing the current network.

Return type *ODESystem*

class desr.chemical_reaction_network.**ChemicalSpecies**(*id_*)
Bases: object

Chemical species, A_i from Harrington paper. Typically represents a single chemical element.

class desr.chemical_reaction_network.**Complex**(**args*, ***kwargs*)
Bases: _abcoll.MutableMapping

A complex of ChemicalSpecies. Represented as a dictionary where the keys are chemical species and the value represents its coefficient in the complex.

as_vector(*variable_order*)
Represent the complex as a vector with respect to a given variable order.

Parameters **variable_order** (*iter*) – Iterable of *ChemicalSpecies*'s.

Return type tuple

class desr.chemical_reaction_network.**Reaction**(*complex1*, *complex2*)
Bases: object

Represents a reaction between complexes, from complex1 to complex2

7.7 desr.sympy_helper module

Created on Fri Dec 26 12:35:16 2014

Helper functions to deal with sympy expressions and equations

Author: Richard Tanburn (richard.tanburn@gmail.com)

desr.sympy_helper.**degree**(*expr*)

Return the degree of a sympy expression. I.e. the largest number of variables multiplied together. NOTE DOES take into account idempotency of binary variables

```
>>> str_eqns = ['x + y',
...             'x*y*z - 1',
...             'x ** 2 + a*b*c',
...             'x**2 + y',
...             'x',
...             'x*y',]
>>> eqns = str_exprs_to_sympy_eqns(str_eqns)
>>> for e in eqns: print degree(e.lhs - e.rhs)
1
3
3
1
1
2
```

Check we deal with constants correctly >>> (degree(0), degree(1), degree(4), ... degree(sympy.S.Zero), degree(sympy.S.One), degree(sympy.sympify(4))) (0, 0, 0, 0, 0, 0)

desr.sympy_helper.**dict_as_eqns**(*dict_*)

Given a dictionary of lhs: rhs, return the sympy Equations in a list

```
>>> x, y, z = sympy.symbols('x y z')
>>> dict_as_eqns({x: 1, y: z, x*y: 1 - z})
[Eq(x*y, -z + 1), Eq(x, 1), Eq(y, z)]
```

desr.sympy_helper.**eqns_with_variables**(*eqns*, *variables*, *strict=False*)

Given a set of atoms, return only equations that have something in common

```
>>> x, y, z1, z2 = sympy.symbols('x y z1 z2')
>>> eqns = ['x + y == 1', '2*z1 + 1 == z2', 'x*z1 == 0']
>>> eqns = str_eqns_to_sympy_eqns(eqns)
>>> eqns_with_variables(eqns, [x])
[Eq(x + y - 1, 0), Eq(x*z1, 0)]
>>> eqns_with_variables(eqns, [z1])
[Eq(2*z1 - z2 + 1, 0), Eq(x*z1, 0)]
>>> eqns_with_variables(eqns, [y])
[Eq(x + y - 1, 0)]
```

```
>>> eqns_with_variables(eqns, [x], strict=True)
[]
>>> eqns_with_variables(eqns, [x, z1], strict=True)
[Eq(x*z1, 0)]
>>> eqns_with_variables(eqns, [x, y, z1], strict=True)
[Eq(x + y - 1, 0), Eq(x*z1, 0)]
```

desr.sympy_helper.**expressions_to_variables**(*exprs*)

Take a list of equations or expressions and return a set of variables

```
>>> eqn = sympy.Eq(sympy.sympify('x*a + 1'))
>>> expr = sympy.sympify('x + y*z + 2*a^b')
>>> to_test = [expr, eqn]
>>> expressions_to_variables(to_test)
set([x, z, a, b, y])
```

desr.sympy_helper.**is_constant**(*expr*)

Determine whether an expression is constant >>> expr = 'x + 2*y' >>> is_constant(sympy.sympify(expr)) False >>> expr = 'x + 5' >>> is_constant(sympy.sympify(expr)) False >>> expr = '3' >>> is_constant(sympy.sympify(expr)) True >>> expr = '2*x - 4' >>> is_constant(sympy.sympify(expr)) False

desr.sympy_helper.**is_equation**(*eqn*, *check_true=True*)

Return True if it is an equation rather than a boolean value. If it is False, raise a ContradictionException. We never want anything that might be False.

Optionally, we can turn the check off, but THE DEFAULT VALUE SHOULD ALWAYS BE TRUE. Otherwise bad things will happen.

```
>>> x, y = sympy.symbols('x y')
>>> eq1 = sympy.Eq(x, y)
>>> eq2 = sympy.Eq(x, x)
>>> eq3 = sympy.Eq(x, y).subs(y, x)
>>> eq4 = sympy.Eq(2*x*y, 2)
```

```
>>> is_equation(eq1)
True
>>> is_equation(eq2)
False
>>> is_equation(eq3)
False
>>> is_equation(eq4)
True
```

Now check that it raises exceptions for the right things

```
>>> is_equation(0)
False
```

desr.sympy_helper.**is_monomial**(*expr*)

Determine whether expr is a monomial

```
>>> is_monomial(sympy.sympify('a*b**2/c'))
True
>>> is_monomial(sympy.sympify('a*b**2/c + d/e'))
False
>>> is_monomial(sympy.sympify('a*b**2/c + 1'))
False
>>> is_monomial(sympy.sympify('a*(b**2/c + 1)'))
False
```

desr.sympy_helper.**monomial_to_powers**(*monomial*, *variables*)

Given a monomial, return powers wrt some variables

```
>>> variables = sympy.var('a b c d e')
>>> monomial_to_powers(sympy.sympify('a*b'), variables)
[1, 1, 0, 0, 0]
```

```
>>> monomial_to_powers(sympy.sympify('a*b**2/c'), variables)
[1, 2, -1, 0, 0]
```

```
>>> monomial_to_powers(sympy.sympify('a*b**2/c + d/e'), variables)
Traceback (most recent call last):
    ...
ValueError: a*b**2/c + d/e is not a monomial
```

```
>>> monomial_to_powers(sympy.sympify('a*b**2/c + 1'), variables)
Traceback (most recent call last):
    ...
ValueError: a*b**2/c + 1 is not a monomial
```

desr.sympy_helper.**standardise_equation**(*eqn*)
Remove binary squares etc

desr.sympy_helper.**str_eqns_to_sympy_eqns**(*str_eqns*)
Take string equations and sympify

```
>>> str_eqns = ['x + y == 1', 'x*y*z - 3*a == -3']
>>> eqns = str_eqns_to_sympy_eqns(str_eqns)
>>> for e in eqns: print e
Eq(x + y - 1, 0)
Eq(-3*a + x*y*z + 3, 0)
```

desr.sympy_helper.**str_exprs_to_sympy_eqns**(*str_exprs*)
Take some strings and return the sympy expressions

```
>>> str_eqns = ['x + y - 1', 'x*y*z - 3*a + 3', '2*a - 4*b']
>>> eqns = str_exprs_to_sympy_eqns(str_eqns)
>>> for e in eqns: print e
Eq(x + y - 1, 0)
Eq(-3*a + x*y*z + 3, 0)
Eq(2*a - 4*b, 0)
```

desr.sympy_helper.**unique_array_stable**(*array*)
Given a list of things, return a new list with unique elements with original order preserved (by first occurence)

```
>>> print unique_array_stable([1, 3, 5, 4, 7, 4, 2, 1, 9])
[1, 3, 5, 4, 7, 2, 9]
```

7.8 desr.tex_tools module

Created on Wed Aug 12 01:37:16 2015

Author: Richard Tanburn (richard.tanburn@gmail.com)

desr.tex_tools.**eqn_to_tex**(*eqn*)

desr.tex_tools.**eqns_to_tex**(*eqns*)
To convert to array environment, copy the output into a lyx LaTeX cell, then copy this entire cell into an eqnarray of sufficient size

desr.tex_tools.**expr_to_tex**(*expr*)
Given a sympy expression, write out the TeX.

Parameters **expr** (*sympy.Expression*) – Expression to turn into TeX

Returns str

```
>>> print map(expr_to_tex, map(sympy.sympify, ['(x + y - 1.5)**2', '(x + y_m1)**1
↪', 'k_m1*t']))
['\\left(x + y - 1.5\\right)^{2}', 'x + y_{-1}', 'k_{-1} t']
```

desr.tex_tools.**matrix_to_tex**(*matrix_*)
Given a matrix, write out the TeX.

Parameters **matrix** (*sympy.Matrix*) – Matrix to turn into TeX

Returns str

Printing is the correct way to use this function, but the docstring looks a bit odd. >>> print matrix_to_tex(sympy.eye(2)) 1 & 0 \ 0 & 1 \

desr.tex_tools.**tex_to_sympy**(*tex*)
Given some TeX, turn it into sympy expressions and equations. Each line is parsed seperately.

Parameters **tex** (*str*) – LaTeX

Returns list

```
>>> lines = [r'\frac{dE}{dt} &= - k_1 E S + k_{-1} C + k_2 C \\',
... r'\frac{dS}{dt} &= - k_1 E S + k_{-1} C \\',
... r'\frac{dC}{dt} &= k_1 E S - k_{-1} C - k_2 C \\',
... r'\frac{dP}{dt} &= k_2 C']
>>> sym = tex_to_sympy('\n'.join(lines))
>>> for s in sym: print s
Eq(Derivative(E, t), C*k_2 + C*k_m1 - E*S*k_1)
Eq(Derivative(S, t), C*k_m1 - E*S*k_1)
Eq(Derivative(C, t), -C*k_2 - C*k_m1 + E*S*k_1)
Eq(Derivative(P, t), C*k_2)
```

```
>>> print tex_to_sympy('k_2 &= V_2d ( APCT - APCs ) + V_2dd APCs')
Eq(k_2, APCs*V_2dd + V_2d*(APCT - APCs))
```

desr.tex_tools.**var_to_tex**(*var*)
Given a sympy variable, write out the TeX.

Parameters **var** (*sympy.Symbol*) – Variable to turn into TeX

Returns str

```
>>> print map(var_to_tex, sympy.symbols('x y_1 Kw_3 z_{3} k_m1'))
['x', 'y_{1}', 'Kw_{3}', 'z_{3}', 'k_{-1}']
```

7.9 desr.unittests module

class desr.unittests.**TestChemicalReactionNetwork**(*methodName='runTest'*)
Bases: unittest.case.TestCase

Test cases for the ChemicalReactionNetwork class

test_crn_harrington()
Example 2.8 from Harrington - Joining and decomposing

test_crn_harrington2()
Example 1 from Harrington board notes - Joining and decomposing

class desr.unittests.**TestHermiteMethods**(*methodName='runTest'*)
Bases: unittest.case.TestCase

test_example1()
Example from Extended gcd and Hermite nomal form via lattice basis reduction

test_example_sage()
Example from the Sage website See: http://doc.sagemath.org/html/en/reference/matrices/sage/matrix/matrix_integer_dense.html

test_normal_hermite_multiplier_example()
Example from Hubert Labahn

test_wiki_example()
Test the examples from wikipedia https://en.wikipedia.org/wiki/Hermite_normal_form

class desr.unittests.**TestInitialConditions**(*methodName='runTest'*)
Bases: unittest.case.TestCase

Test methods relating to initial conditions

test_reduced_michaelis_menten()
Michaelis Menten equations from [MM]

class desr.unittests.**TestODESystemScaling**(*methodName='runTest'*)
Bases: unittest.case.TestCase

Test ode_system.py scaling methods

test_example_6_4_hub_lab()
Predator prey model from [HL13], example 6.4 dz1/dt = z1*(1+z1*z2) dz2/dt = z2*(1/t - z1*z2)

test_example_6_6_hub_lab()
Example 6.6 from [HL13], where we act on time dz1/dt = z1*(z1**5*z2 - 2)/(3*t) dz2/dt = z2*(10 - 2*z1**5*z2 + 3*z1**2*z2/t)/(3*t)

test_example_pred_prey_choosing_invariants()
Predator prey model from [HL13] dn/dt = n(r(1 - n/K) - kp/(n+d)) dp/dt = sp(1 - hp / n)

Rather than using the invariants suggested by the algorithm, we pick our own and extend it.

test_example_pred_prey_hub_lab()
Predator prey model from [HL13] dn/dt = n(r(1 - n/K) - kp/(n+d)) dp/dt = sp(1 - hp / n)

test_verhulst_log_growth()
Verhult logistic growth model from [HL13] dn/dt = r*n*(1-n/k)

CHAPTER 8

Bibliography

CHAPTER 9

Indices and tables

- genindex
- modindex
- search

Bibliography

[FO99] Mark Fels and Peter J Olver. Moving Coframes: II. Regularization and Theoretical Foundations. *Acta Applicandae Mathematicae*, 55:127–208, 1999.

[HMM98] George Havas, Bohdan S. Majewski, and Keith R. Matthews. Extended GCD and Hermite Normal Form Algorithms via Lattice Basis Reduction. *Experimental Mathematics*, 7(2):125–136, jan 1998. URL: http://www.tandfonline.com/doi/abs/10.1080/10586458.1998.10504362, doi:10.1080/10586458.1998.10504362.

[Hub07] Evelyne Hubert. Differential invariants of a Lie group action: syzygies on a generating set. *Journal of Symbolic Computation*, oct 2007. URL: http://arxiv.org/abs/0710.4318http://dx.doi.org/10.1016/j.jsc.2008.08.003, arXiv:0710.4318, doi:10.1016/j.jsc.2008.08.003.

[HL13] Evelyne Hubert and George Labahn. Scaling Invariants and Symmetry Reduction of Dynamical Systems. *Foundations of Computational Mathematics*, 13(4):479–516, aug 2013. URL: http://link.springer.com/10.1007/s10208-013-9165-9, doi:10.1007/s10208-013-9165-9.

[SS89] Lee A. Segel and Marshall Slemrod. The Quasi-Steady-State Assumption: A Case Study in Perturbation. 1989. URL: https://www.jstor.org/stable/2031405, doi:10.2307/2031405.

Python Module Index

d

desr.chemical_reaction_network, 69
desr.diophantine, 68
desr.matrix_normal_forms, 62
desr.ode_system, 37
desr.ode_translation, 46
desr.sympy_helper, 71
desr.tex_tools, 73
desr.unittests, 75

Index

A

add_constraints() (*desr.ode_system.ODESystem method*), 37
addr() (*in module desr.diophantine*), 68
as_vector() (*desr.chemical_reaction_network.Complex method*), 70
auxiliaries() (*desr.ode_translation.ODETranslation method*), 46

C

ChemicalReactionNetwork (*class in desr.chemical_reaction_network*), 69
ChemicalSpecies (*class in desr.chemical_reaction_network*), 70
cholesky() (*in module desr.diophantine*), 68
comparer() (*in module desr.diophantine*), 68
Complex (*class in desr.chemical_reaction_network*), 70
constant_variables (*desr.ode_system.ODESystem attribute*), 38
constraints (*desr.ode_system.ODESystem attribute*), 38
copy() (*desr.ode_system.ODESystem method*), 38

D

default_order_variables() (*desr.ode_system.ODESystem method*), 38
degree() (*in module desr.sympy_helper*), 71
dep_var_herm_mult() (*desr.ode_translation.ODETranslation method*), 47
dep_var_inv_herm_mult() (*desr.ode_translation.ODETranslation method*), 47
derivative_dict (*desr.ode_system.ODESystem attribute*), 39
derivatives (*desr.ode_system.ODESystem attribute*), 39
desr.chemical_reaction_network (*module*), 69
desr.diophantine (*module*), 68
desr.matrix_normal_forms (*module*), 62
desr.ode_system (*module*), 37
desr.ode_translation (*module*), 46
desr.sympy_helper (*module*), 71
desr.tex_tools (*module*), 73
desr.unittests (*module*), 75
dict_as_eqns() (*in module desr.sympy_helper*), 71
diff_subs() (*desr.ode_system.ODESystem method*), 39

E

element_wise_lt() (*in module desr.matrix_normal_forms*), 62
eqn_to_tex() (*in module desr.tex_tools*), 73
eqns_to_tex() (*in module desr.tex_tools*), 73
eqns_with_variables() (*in module desr.sympy_helper*), 71
expand_matrix() (*in module desr.matrix_normal_forms*), 62
expr_to_tex() (*in module desr.tex_tools*), 74
expressions_to_variables() (*in module desr.sympy_helper*), 71
extend_from_invariants() (*desr.ode_translation.ODETranslation method*), 47
extend_rectangular_matrix() (*in module desr.ode_translation*), 60

F

first_nonzero_is_negative() (*in module desr.diophantine*), 68
from_diagram() (*desr.chemical_reaction_network.ChemicalReactionN class method*), 69
from_dict() (*desr.ode_system.ODESystem class method*), 40
from_equations() (*desr.ode_system.ODESystem class method*), 41
from_ode_system() (*desr.ode_translation.ODETranslation class*

method), 48
from_tex() (*desr.ode_system.ODESystem class method*), 41

G

get_pivot_row_indices() (*in module desr.matrix_normal_forms*), 63
get_solutions() (*in module desr.diophantine*), 68
gram() (*in module desr.diophantine*), 68

H

herm_form (*desr.ode_translation.ODETranslation attribute*), 48
herm_mult (*desr.ode_translation.ODETranslation attribute*), 48
herm_mult_i (*desr.ode_translation.ODETranslation attribute*), 48
herm_mult_n (*desr.ode_translation.ODETranslation attribute*), 48
hnf_col() (*in module desr.matrix_normal_forms*), 63
hnf_col_lll() (*in module desr.matrix_normal_forms*), 63
hnf_row() (*in module desr.matrix_normal_forms*), 64
hnf_row_lll() (*in module desr.matrix_normal_forms*), 64

I

indep_var (*desr.ode_system.ODESystem attribute*), 42
indep_var_index (*desr.ode_system.ODESystem attribute*), 42
initial_conditions (*desr.ode_system.ODESystem attribute*), 42
initialise_working_matrices() (*in module desr.diophantine*), 68
introot() (*in module desr.diophantine*), 68
inv_herm_mult (*desr.ode_translation.ODETranslation attribute*), 49
inv_herm_mult_d (*desr.ode_translation.ODETranslation attribute*), 49
inv_herm_mult_u (*desr.ode_translation.ODETranslation attribute*), 49
invariants() (*desr.ode_translation.ODETranslation method*), 49
is_constant() (*in module desr.sympy_helper*), 72
is_equation() (*in module desr.sympy_helper*), 72
is_hnf_col() (*in module desr.matrix_normal_forms*), 65
is_hnf_row() (*in module desr.matrix_normal_forms*), 65
is_invariant_expr() (*desr.ode_translation.ODETranslation method*), 49
is_monomial() (*in module desr.sympy_helper*), 72
is_normal_hermite_multiplier() (*in module desr.matrix_normal_forms*), 65
is_smf() (*in module desr.matrix_normal_forms*), 65

L

lcasvector() (*in module desr.diophantine*), 68
lllhermite() (*in module desr.diophantine*), 69
lnearint() (*in module desr.diophantine*), 69

M

matrix_to_tex() (*in module desr.tex_tools*), 74
maximal_scaling_matrix() (*desr.ode_system.ODESystem method*), 42
maximal_scaling_matrix() (*in module desr.ode_system*), 45
minus() (*in module desr.diophantine*), 69
monomial_to_powers() (*in module desr.sympy_helper*), 72
moving_frame() (*desr.ode_translation.ODETranslation method*), 49
multiplier_add_columns() (*desr.ode_translation.ODETranslation method*), 50
multiplier_negate_column() (*desr.ode_translation.ODETranslation method*), 52
multiplier_swap_columns() (*desr.ode_translation.ODETranslation method*), 53
multr() (*in module desr.diophantine*), 69

N

n (*desr.chemical_reaction_network.ChemicalReactionNetwork attribute*), 70
n (*desr.ode_translation.ODETranslation attribute*), 54
non_constant_variables (*desr.ode_system.ODESystem attribute*), 42
nonzero() (*in module desr.diophantine*), 69
normal_hnf_col() (*in module desr.matrix_normal_forms*), 66
normal_hnf_row() (*in module desr.matrix_normal_forms*), 67
NoSolutionException, 68
num_constants (*desr.ode_system.ODESystem attribute*), 42

O

ode_equations() (*desr.chemical_reaction_network.ChemicalReaction method*), 70
ODESystem (*class in desr.ode_system*), 37
ODETranslation (*class in desr.ode_translation*), 46

P

p (*desr.chemical_reaction_network.ChemicalReactionNetwork attribute*), 70
parse_de() (*in module desr.ode_system*), 45
power_matrix() (*desr.ode_system.ODESystem method*), 42

R

r (*desr.chemical_reaction_network.ChemicalReactionNetwork attribute*), 70
r (*desr.ode_translation.ODETranslation attribute*), 54
rational_expr_to_power_matrix() (*in module desr.ode_system*), 45
rational_section() (*desr.ode_translation.ODETranslation method*), 55
ratior() (*in module desr.diophantine*), 69
Reaction (*class in desr.chemical_reaction_network*), 70
reduce_matrix() (*in module desr.diophantine*), 69
reorder_variables() (*desr.ode_system.ODESystem method*), 43
reverse_translate() (*desr.ode_translation.ODETranslation method*), 55
reverse_translate_dep_var() (*desr.ode_translation.ODETranslation method*), 55
reverse_translate_general() (*desr.ode_translation.ODETranslation method*), 56
reverse_translate_parameter() (*desr.ode_translation.ODETranslation method*), 57
rewrite_rules() (*desr.ode_translation.ODETranslation method*), 58

S

scale_action() (*in module desr.ode_translation*), 61
scaling_matrix (*desr.ode_translation.ODETranslation attribute*), 59
sign() (*in module desr.diophantine*), 69
smf() (*in module desr.matrix_normal_forms*), 67
solve() (*in module desr.diophantine*), 69
standardise_equation() (*in module desr.sympy_helper*), 73
str_eqns_to_sympy_eqns() (*in module desr.sympy_helper*), 73
str_exprs_to_sympy_eqns() (*in module desr.sympy_helper*), 73
subr() (*in module desr.diophantine*), 69
swap_rows() (*in module desr.diophantine*), 69

T

test_crn_harrington() (*desr.unittests.TestChemicalReactionNetwork method*), 75
test_crn_harrington2() (*desr.unittests.TestChemicalReactionNetwork method*), 75
test_example1() (*desr.unittests.TestHermiteMethods method*), 75
test_example_6_4_hub_lab() (*desr.unittests.TestODESystemScaling method*), 75
test_example_6_6_hub_lab() (*desr.unittests.TestODESystemScaling method*), 75
test_example_pred_prey_choosing_invariants() (*desr.unittests.TestODESystemScaling method*), 75
test_example_pred_prey_hub_lab() (*desr.unittests.TestODESystemScaling method*), 75
test_example_sage() (*desr.unittests.TestHermiteMethods method*), 75
test_normal_hermite_multiplier_example() (*desr.unittests.TestHermiteMethods method*), 75
test_reduced_michaelis_menten() (*desr.unittests.TestInitialConditions method*), 75
test_verhulst_log_growth() (*desr.unittests.TestODESystemScaling method*), 75
test_wiki_example() (*desr.unittests.TestHermiteMethods method*), 75
TestChemicalReactionNetwork (*class in desr.unittests*), 75
TestHermiteMethods (*class in desr.unittests*), 75
TestInitialConditions (*class in desr.unittests*), 75
TestODESystemScaling (*class in desr.unittests*), 75
tex_to_sympy() (*in module desr.tex_tools*), 74
to_ode_system() (*desr.chemical_reaction_network.ChemicalReaction method*), 70
to_tex() (*desr.ode_system.ODESystem method*), 43
to_tex() (*desr.ode_translation.ODETranslation method*), 59
translate() (*desr.ode_translation.ODETranslation method*), 59
translate_dep_var() (*desr.ode_translation.ODETranslation method*), 59
translate_general()

(*desr.ode_translation.ODETranslation method*), 60
translate_parameter() (*desr.ode_translation.ODETranslation method*), 60
translate_parameter_substitutions() (*desr.ode_translation.ODETranslation method*), 60

U

unique_array_stable() (*in module desr.sympy_helper*), 73
update_initial_conditions() (*desr.ode_system.ODESystem method*), 44

V

var_to_tex() (*in module desr.tex_tools*), 74
variables (*desr.ode_system.ODESystem attribute*), 45
variables_domain (*desr.ode_translation.ODETranslation attribute*), 60