The NCAlgebra Suite

Version 5.0.6

J. William Helton   Mauricio C. de Oliveira

with earlier contributions by Bob Miller & Mark Stankus

Contents

License 1

I User Guide 3

1 Acknowledgements 5

2 Changes in Version 5.0 7
- 2.1 Version 5.0.6 . . . 7
- 2.2 Version 5.0.5 . . . 7
- 2.3 Version 5.0.4 . . . 7
- 2.4 Version 5.0.3 . . . 7
- 2.5 Version 5.0.2 . . . 7
- 2.6 Version 5.0.1 . . . 8
- 2.7 Version 5.0.0 . . . 8

3 Introduction 9
- 3.1 Running NCAlgebra . . . 9
- 3.2 Now what? . . . 9
- 3.3 Testing . . . 10
- 3.4 Pre-2017 NCGB C++ version . . . 10

4 Most Basic Commands 11
- 4.1 To Commute Or Not To Commute? . . . 11
- 4.2 Inverses, Transposes and Adjoints . . . 12
- 4.3 Replace . . . 13
- 4.4 Polynomials . . . 14
- 4.5 Rationals and Simplification . . . 16
- 4.6 Calculus . . . 17
- 4.7 Matrices . . . 18
  - 4.7.1 Inverses, products, adjoints, etc . . . 18
  - 4.7.2 LU Decomposition . . . 19
  - 4.7.3 LU Decomposition with complete pivoting . . . 21
  - 4.7.4 LDL Decomposition . . . 21
  - 4.7.5 Replace with matrices . . . 22
- 4.8 Quadratic polynomials, second direction derivatives and convexity . . . 23

5 More Advanced Commands 27
- 5.1 Advanced Rules and Replacements . . . 27
  - 5.1.1 `ReplaceAll` (`/.`) and `ReplaceRepeated` (`//.`) often fail . . . 27
  - 5.1.2 The fix is `NCReplace` . . . 28
  - 5.1.3 Trouble with `Block` and `Module` . . . 29

5.2 Expanding matrix products . . . 30
5.2.1 Trouble with `Plus` (`+`) and matrices . . . 32
5.3 Polynomials with commutative coefficients . . . 33
5.4 Polynomials with noncommutative coefficients . . . 36
5.5 Linear polynomials . . . 37

6 Noncommutative Gröbner Basis 39
6.1 What is a Gröbner Basis? . . . 39
6.2 Solving equations . . . 40
6.3 A slightly more challenging example . . . 41
6.4 Simplifying polynomial expresions . . . 42
6.5 Minimal versus reduced Gröbner Basis . . . 43
6.6 Simplifying rational expresions . . . 43
6.7 Simplification with NCGBSimplifyRational . . . 45
6.8 Ordering on variables and monomials . . . 45
6.8.1 Lex Order: the simplest elimination order . . . 45
6.8.2 Graded lex ordering: a non-elimination order . . . 46
6.8.3 Multigraded lex ordering: a variety of elimination orders . . . 46
6.9 A complete example: the partially prescribed matrix inverse problem . . . 47

7 Semidefinite Programming 51
7.1 Semidefinite Programs in Matrix Variables . . . 51
7.2 Semidefinite Programs in Vector Variables . . . 53

8 Pretty Output with Notebooks and TeX 55
8.1 Pretty Output . . . 55
8.2 Using NCTeX . . . 56
8.2.1 NCTeX Options . . . 57
8.3 Using NCTeXForm . . . 58

II Reference Manual 61

9 Reference Manual 63
9.1 NC . . . 63
9.1.1 Options . . . 63
9.2 NCAlgebra . . . 63
9.2.1 Messages . . . 64

10 Packages for manipulating NC expressions 65
10.1 NonCommutativeMultiply . . . 65
10.1.1 aj . . . 66
10.1.2 co . . . 66
10.1.3 Id . . . 66
10.1.4 inv . . . 66
10.1.5 rt . . . 66
10.1.6 tp . . . 66
10.1.7 CommutativeQ . . . 66
10.1.8 NonCommutativeQ . . . 66
10.1.9 SetCommutative . . . 66
10.1.10 SetCommutativeHold . . . 67
10.1.11 SetNonCommutative . . . 67
10.1.12 SetNonCommutativeHold . . . 67
10.1.13 SetCommutativeFunction . . . 67
10.1.14 SNC . . . 67

10.1.15 SetCommutingOperators . . . 67
10.1.16 UnsetCommutingOperators . . . 68
10.1.17 CommutingOperatorsQ . . . 68
10.1.18 Commutative . . . 68
10.1.19 CommuteEverything . . . 68
10.1.20 BeginCommuteEverything . . . 68
10.1.21 EndCommuteEverything . . . 68
10.1.22 ExpandNonCommutativeMultiply . . . 68
10.1.23 NCExpand . . . 68
10.1.24 NCE . . . 69
10.2 NCTr . . . 69
10.2.1 tr . . . 69
10.2.2 SortCyclicPermutation . . . 69
10.2.3 SortedCyclicPermutationQ . . . 69
10.3 NCCollect . . . 69
10.3.1 NCCollect . . . 69
10.3.2 NCCollectSelfAdjoint . . . 70
10.3.3 NCCollectSymmetric . . . 70
10.3.4 NCStrongCollect . . . 70
10.3.5 NCStrongCollectSelfAdjoint . . . 71
10.3.6 NCStrongCollectSymmetric . . . 71
10.3.7 NCCompose . . . 71
10.3.8 NCDecompose . . . 71
10.3.9 NCTermsOfDegree . . . 71
10.3.10 NCTermsOfTotalDegree . . . 72
10.4 NCReplace . . . 72
10.4.1 NCReplace . . . 73
10.4.2 NCReplaceAll . . . 73
10.4.3 NCReplaceList . . . 73
10.4.4 NCReplaceRepeated . . . 73
10.4.5 NCR . . . 73
10.4.6 NCRA . . . 74
10.4.7 NCRR . . . 74
10.4.8 NCRL . . . 74
10.4.9 NCMakeRuleSymmetric . . . 74
10.4.10 NCMakeRuleSelfAdjoint . . . 74
10.4.11 NCReplaceSymmetric . . . 74
10.4.12 NCReplaceAllSymmetric . . . 74
10.4.13 NCReplaceRepeatedSymmetric . . . 74
10.4.14 NCReplaceListSymmetric . . . 74
10.4.15 NCRSym . . . 75
10.4.16 NCRASym . . . 75
10.4.17 NCRRSym . . . 75
10.4.18 NCRLSym . . . 75
10.4.19 NCReplaceSelfAdjoint . . . 75
10.4.20 NCReplaceAllSelfAdjoint . . . 75
10.4.21 NCReplaceRepeatedSelfAdjoint . . . 75
10.4.22 NCReplaceListSelfAdjoint . . . 75
10.4.23 NCRSA . . . 75
10.4.24 NCRASA . . . 76
10.4.25 NCRRSA . . . 76
10.4.26 NCRLSA . . . 76
10.4.27 NCMatrixExpand . . . 76
10.4.28 NCMatrixReplaceAll . . . 76

10.4.29 NCMatrixReplaceRepeated . . . 76
10.5 NCSelfAdjoint . . . 76
10.5.1 NCSymmetricQ . . . 77
10.5.2 NCSymmetricTest . . . 77
10.5.3 NCSymmetricPart . . . 77
10.5.4 NCSelfAdjointQ . . . 78
10.5.5 NCSelfAdjointTest . . . 78
10.6 NCSimplifyRational . . . 78
10.6.1 NCNormalizeInverse . . . 79
10.6.2 NCSimplifyRational . . . 80
10.6.3 NCSR . . . 80
10.6.4 NCSimplifyRationalSinglePass . . . 80
10.6.5 NCPreSimplifyRational . . . 80
10.6.6 NCPreSimplifyRationalSinglePass . . . 80
10.7 NCDiff . . . 80
10.7.1 NCDirectionalD . . . 80
10.7.2 NCGrad . . . 81
10.7.3 NCHessian . . . 82
10.7.4 DirectionalD . . . 82
10.7.5 NCIntegrate . . . 82
10.8 NCConvexity . . . 82
10.8.1 NCIndependent . . . 83
10.8.2 NCConvexityRegion . . . 83

11 Packages for manipulating NC block matrices 85
11.1 NCDot . . . 85
11.1.1 tpMat . . . 85
11.1.2 ajMat . . . 85
11.1.3 coMat . . . 85
11.1.4 NCDot . . . 85
11.1.5 NCInverse . . . 86
11.2 NCMatrixDecompositions . . . 86
11.2.1 NCLUDecompositionWithPartialPivoting . . . 86
11.2.2 NCLUDecompositionWithCompletePivoting . . . 86
11.2.3 NCLDLDecomposition . . . 87
11.2.4 NCUpperTriangularSolve . . . 87
11.2.5 NCLowerTriangularSolve . . . 87
11.2.6 NCLUInverse . . . 87
11.2.7 NCLUPartialPivoting . . . 87
11.2.8 NCLUCompletePivoting . . . 87
11.2.9 NCLeftDivide . . . 88
11.2.10 NCRightDivide . . . 88
11.3 MatrixDecompositions: linear algebra templates . . . 88
11.3.1 LUDecompositionWithPartialPivoting . . . 89
11.3.2 LUDecompositionWithCompletePivoting . . . 89
11.3.3 LDLDecomposition . . . 89
11.3.4 UpperTriangularSolve . . . 90
11.3.5 LowerTriangularSolve . . . 90
11.3.6 LUInverse . . . 90
11.3.7 GetLUMatrices . . . 91
11.3.8 GetFullLUMatrices . . . 91
11.3.9 GetLDUMatrices . . . 91
11.3.10 GetFullLDUMatrices . . . 91
11.3.11 GetDiagonal . . . 92

11.3.12 LUPartialPivoting . . . 92
11.3.13 LUCompletePivoting . . . 92
11.3.14 LURowReduce . . . 92
11.3.15 LURowReduceIncremental . . . 92

12 Packages for pretty output, testing, and utilities 93

12.1 NCOutput . . . 93
12.1.1 NCSetOutput . . . 93
12.2 NCTeX . . . 93
12.2.1 NCTeX . . . 94
12.2.2 NCRunDVIPS . . . 94
12.2.3 NCRunLaTeX . . . 94
12.2.4 NCRunPDFLaTeX . . . 94
12.2.5 NCRunPDFViewer . . . 94
12.2.6 NCRunPS2PDF . . . 94
12.3 NCTeXForm . . . 94
12.3.1 NCTeXForm . . . 94
12.3.2 NCTeXFormSetStarStar . . . 94
12.3.3 NCTeXFormSetStar . . . 94
12.4 NCRun . . . 95
12.4.1 NCRun . . . 95
12.5 NCTest . . . 95
12.5.1 NCTest . . . 95
12.5.2 NCTestCheck . . . 95
12.5.3 NCTestRun . . . 95
12.5.4 NCTestSummarize . . . 96
12.6 NCDebug . . . 96
12.6.1 NCDebug . . . 96
12.7 NCUtil . . . 96
12.7.1 NCConsistentQ . . . 96
12.7.2 NCGrabFunctions . . . 96
12.7.3 NCGrabSymbols . . . 97
12.7.4 NCGrabIndeterminants . . . 97
12.7.5 NCVariables . . . 97
12.7.6 NCConsolidateList . . . 98
12.7.7 NCLeafCount . . . 98
12.7.8 NCReplaceData . . . 98
12.7.9 NCToExpression . . . 98

13 Data structures for fast calculations 99

13.1 NCPoly . . . 99
13.1.1 Efficient storage of NC polynomials with rational coefficients . . . 99
13.1.2 Ways to represent NC polynomials . . . 100
13.1.2.1 NCPoly . . . 100
13.1.2.2 NCPolyMonomial . . . 100
13.1.2.3 NCPolyConstant . . . 101
13.1.2.4 NCPolyConvert . . . 101
13.1.2.5 NCPolyFromCoefficientArray . . . 102
13.1.2.6 NCPolyFromGramMatrix . . . 102
13.1.3 Access and utlity functions . . . 102
13.1.3.1 NCPolyMonomialQ . . . 102
13.1.3.2 NCPolyDegree . . . 102
13.1.3.3 NCPolyPartialDegree . . . 103
13.1.3.4 NCPolyMonomialDegree . . . 103

13.1.3.5 NCPolyNumberOfVariables . . . 103
13.1.3.6 NCPolyNumberOfTerms . . . 103
13.1.3.7 NCPolyCoefficient . . . 103
13.1.3.8 NCPolyCoefficientArray . . . 103
13.1.3.9 NCPolyGramMatrix . . . 104
13.1.3.10 NCPolyGetCoefficients . . . 104
13.1.3.11 NCPolyGetDigits . . . 104
13.1.3.12 NCPolyGetIntegers . . . 104
13.1.3.13 NCPolyLeadingMonomial . . . 105
13.1.3.14 NCPolyLeadingTerm . . . 105
13.1.3.15 NCPolyOrderType . . . 105
13.1.3.16 NCPolyToRule . . . 105
13.1.3.17 NCPolyTermsOfDegree . . . 106
13.1.3.18 NCPolyTermsOfTotalDegree . . . 106
13.1.3.19 NCPolyQuadraticTerms . . . 107
13.1.3.20 NCPolyQuadraticChipset . . . 107
13.1.3.21 NCPolyReverseMonomials . . . 107
13.1.3.22 NCPolyGetOptions . . . 108
13.1.4 Formating functions . . . 108
13.1.4.1 NCPolyDisplay . . . 108
13.1.4.2 NCPolyDisplayOrder . . . 108
13.1.5 Arithmetic functions . . . 108
13.1.5.1 NCPolyDivideDigits . . . 108
13.1.5.2 NCPolyDivideLeading . . . 108
13.1.5.3 NCPolyFullReduce . . . 108
13.1.5.4 NCPolyNormalize . . . 108
13.1.5.5 NCPolyProduct . . . 108
13.1.5.6 NCPolyQuotientExpand . . . 109
13.1.5.7 NCPolyReduce . . . 109
13.1.5.8 NCPolySum . . . 109
13.1.6 State space realization functions . . . 109
13.1.6.1 NCPolyHankelMatrix . . . 109
13.1.6.2 NCPolyRealization . . . 110
13.1.7 Auxiliary functions . . . 110
13.1.7.1 NCPolyVarsToIntegers . . . 110
13.1.7.2 NCFromDigits . . . 110
13.1.7.3 NCIntegerDigits . . . 111
13.1.7.4 NCIntegerReverse . . . 111
13.1.7.5 NCDigitsToIndex . . . 112
13.1.7.6 NCPadAndMatch . . . 112
13.2 NCPolyInterface . . . 112
13.2.1 NCToNCPoly . . . 113
13.2.2 NCPolyToNC . . . 113
13.2.3 NCRuleToPoly . . . 113
13.2.4 NCMonomialList . . . 113
13.2.5 NCCoefficientRules . . . 114
13.2.6 NCCoefficientList . . . 114
13.2.7 NCCoefficientQ . . . 114
13.2.8 NCMonomialQ . . . 114
13.2.9 NCPolynomialQ . . . 115
13.3 NCPolynomial . . . 115
13.3.1 Efficient storage of NC polynomials with nc coefficients . . . 115
13.3.2 Ways to represent NC polynomials . . . 116
13.3.2.1 NCPolynomial . . . 116

13.3.2.2 NCToNCPolynomial . . . 116
13.3.2.3 NCPolynomialToNC . . . 117
13.3.2.4 NCRationalToNCPolynomial . . . 117
13.3.3 Grouping terms by degree . . . 117
13.3.3.1 NCPTermsOfDegree . . . 117
13.3.3.2 NCPTermsOfTotalDegree . . . 118
13.3.3.3 NCPTermsToNC . . . 118
13.3.4 Utilities . . . 118
13.3.4.1 NCPDegree . . . 118
13.3.4.2 NCPMonomialDegree . . . 118
13.3.4.3 NCPCoefficients . . . 118
13.3.4.4 NCPLinearQ . . . 119
13.3.4.5 NCPQuadraticQ . . . 119
13.3.4.6 NCPCompatibleQ . . . 119
13.3.4.7 NCPSameVariablesQ . . . 119
13.3.4.8 NCPMatrixQ . . . 119
13.3.4.9 NCPNormalize . . . 119
13.3.5 Operations on NC polynomials . . . 119
13.3.5.1 NCPPlus . . . 119
13.3.5.2 NCPTimes . . . 119
13.3.5.3 NCPDot . . . 120
13.3.5.4 NCPSort . . . 120
13.3.5.5 NCPDecompose . . . 120
13.4 NCQuadratic . . . 120
13.4.1 NCToNCQuadratic . . . 120
13.4.2 NCPToNCQuadratic . . . 120
13.4.3 NCQuadraticToNC . . . 121
13.4.4 NCQuadraticToNCPolynomial . . . 121
13.4.5 NCMatrixOfQuadratic . . . 121
13.4.6 NCQuadraticMakeSymmetric . . . 122
13.5 NCSylvester . . . 122
13.5.1 NCToNCSylvester . . . 122
13.5.2 NCPToNCSylvester . . . 122
13.5.3 NCSylvesterToNC . . . 123
13.5.4 NCSylvesterToNCPolynomial . . . 123

14 Noncommutative Gröbner Bases Algorithms 125
14.1 NCGBX . . . 125
14.1.1 SetMonomialOrder . . . 125
14.1.2 SetKnowns . . . 126
14.1.3 SetUnknowns . . . 127
14.1.4 ClearMonomialOrder . . . 128
14.1.5 GetMonomialOrder . . . 128
14.1.6 PrintMonomialOrder . . . 128
14.1.7 NCMakeGB . . . 128
14.1.8 NCProcess . . . 129
14.1.9 NCGBSimplifyRational . . . 129
14.1.10 NCReduce . . . 129
14.2 NCPolyGroebner . . . 130
14.2.1 NCPolyGroebner . . . 130
14.3 NCGB . . . 130

15 Semidefinite Programming Algorithms 131
15.1 NCSDP . . . 131

15.1.1 NCSDP . . . 131
15.1.2 NCSDPForm . . . 132
15.1.3 NCSDPDual . . . 132
15.1.4 NCSDPDualForm . . . 132
15.2 SDP . . . 132
15.2.1 SDPMatrices . . . 133
15.2.2 SDPSolve . . . 133
15.2.3 SDPEval . . . 134
15.2.4 SDPPrimalEval . . . 134
15.2.5 SDPDualEval . . . 134
15.2.6 SDPSylvesterEval . . . 134
15.3 SDPFlat . . . 134
15.3.1 SDPFlatData . . . 135
15.3.2 SDPFlatPrimalEval . . . 135
15.3.3 SDPFlatDualEval . . . 135
15.3.4 SDPFlatSylvesterEval . . . 135
15.4 SDPSylvester . . . 135
15.4.1 SDPEval . . . 136
15.4.2 SDPSylvesterPrimalEval . . . 136
15.4.3 SDPSylvesterDualEval . . . 136
15.4.4 SDPSylvesterSylvesterEval . . . 136
15.5 PrimalDual . . . 136
15.5.1 PrimalDual . . . 137

16 Work in Progress **139**
16.1 NCPolySOS . . . 139
16.1.1 NCPolySOS . . . 139
16.1.2 NCPolySOSToSDP . . . 139
16.2 NCRational . . . 139
16.2.1 State-space realizations for NC rationals . . . 140
16.2.1.1 NCRational . . . 140
16.2.1.2 NCToNCRational . . . 140
16.2.1.3 NCRationalToNC . . . 140
16.2.1.4 NCRationalToCanonical . . . 140
16.2.1.5 CanonicalToNCRational . . . 140
16.2.2 Utilities . . . 140
16.2.2.1 NCROrder . . . 140
16.2.2.2 NCRLinearQ . . . 140
16.2.2.3 NCRStrictlyProperQ . . . 141
16.2.3 Operations on NC rationals . . . 141
16.2.3.1 NCRPlus . . . 141
16.2.3.2 NCRTimes . . . 141
16.2.3.3 NCRTranspose . . . 141
16.2.3.4 NCRInverse . . . 141
16.2.4 Minimal realizations . . . 141
16.2.4.1 NCRControllableRealization . . . 141
16.2.4.2 NCRControllableSubspace . . . 141
16.2.4.3 NCRObservableRealization . . . 141
16.2.4.4 NCRMinimalRealization . . . 141
16.3 NCRealization . . . 141
16.3.1 NCDescriptorRealization . . . 142
16.3.2 NCDeterminantalRepresentationReciprocal . . . 142
16.3.3 NCMatrixDescriptorRealization . . . 142
16.3.4 NCMinimalDescriptorRealization . . . 142

16.3.5 NCSymmetricDescriptorRealization . . . 142
16.3.6 NCSymmetricDeterminantalRepresentationDirect . . . 142
16.3.7 NCSymmetricDeterminantalRepresentationReciprocal . . . 142
16.3.8 NCSymmetrizeMinimalDescriptorRealization . . . 143
16.3.9 NonCommutativeLift . . . 143
16.3.10 SignatureOfAffineTerm . . . 143
16.3.11 TestDescriptorRealization . . . 143
16.3.12 PinnedQ . . . 143
16.3.13 PinningSpace . . . 143

License

NCAlgebra is distributed under the terms of the BSD License:

```
Copyright (c) 2017, J. William Helton and Mauricio C. de Oliveira
All rights reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions are met:
    * Redistributions of source code must retain the above copyright
      notice, this list of conditions and the following disclaimer.
    * Redistributions in binary form must reproduce the above copyright
      notice, this list of conditions and the following disclaimer in the
      documentation and/or other materials provided with the distribution.
    * Neither the name NCAlgebra nor the names of its contributors may be
      used to endorse or promote products derived from this software without
      specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND
ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED
WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE
DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDERS BE LIABLE FOR ANY
DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES
(INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES;
LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND
ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT
(INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS
SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
```

Part I

User Guide

Chapter 1

Acknowledgements

This work was partially supported by the Division of Mathematical Sciences of the National Science Foundation.

The program was written by the authors with mjor earlier contributions from:

- Mark Stankus, Math, Cal Poly San Luis Obispo
- Robert L. Miller, General Atomics Corp

Considerable recent help came from

- Igor Klep, University of Ljubljana

Other contributors include:

- David Hurst, Daniel Lamm, Orlando Merino, Robert Obar, Henry Pfister, Mike Walker, John Wavrik, Lois Yu, J. Camino, J. Griffin, J. Ovall, T. Shaheen, John Shopple.

The beginnings of the program come from eran@slac.

Chapter 2

Changes in Version 5.0

2.1 Version 5.0.6

1. `NC` and `NCAlgebra` are now Contexts
2. NCMatrixExpand moved from NCDot to NCReplace
3. Added SetCommutativeFunction
4. Added tr operator
5. Tests fixed to work even if a-z are not defined as NC globally
6. More tests for CommutativeQ
7. Compatibility with Mathematica 13.0.0

2.2 Version 5.0.5

1. Improved documentation
2. Bug fixes. Thanks Igor Klep!

2.3 Version 5.0.4

1. First implementation of NCPolyGramMatrix and NCPolyFromGramMatrix.
2. Improvements in NCReplace.
3. Several bug fixes and improvements. Thanks Eric Evert!

2.4 Version 5.0.3

1. Restored functionality of SetCommutingOperators.
2. Improved implementation of CommutativeQ for arrays.

2.5 Version 5.0.2

1. `NCCollect` and `NCStrongCollect` can handle commutative variables.
2. Cleaned up initialization files.
3. New function `SetNonCommutativeHold` with `HoldAll` attribute can be used to set Symbols that have been previously assigned values.

2.6 Version 5.0.1

1. Introducing `NCWebInstall` and `NCWebUpdate`.
2. Bug fixes.

2.7 Version 5.0.0

1. Completely rewritten core handling of noncommutative expressions with significant speed gains.
2. Completely rewritten noncommutative Gröbner basis algorithm without any dependence on compiled code. See chapter Noncommutative Gröbner Basis in the user guide and the NCGBX package. Some `NCGB` features are not fully supported yet, most notably NCProcess.
3. New algorithms for representing and operating with noncommutative polynomials with commutative coefficients. These support the new package NCGBX. See this section in the chapter More Advanced Commands and the packages NCPolyInterface and NCPoly.
4. New algorithms for representing and operating with noncommutative polynomials with noncommutative coefficients (NCPolynomial) with specialized facilities for noncommutative quadratic polynomials (NCQuadratic) and noncommutative linear polynomials (NCSylvester).
5. Modified behavior of `CommuteEverything` (see important notes in CommuteEverything).
6. Improvements and consolidation of noncommutative calculus in the package NCDiff.
7. Added a complete set of linear algebra algorithms in the new package MatrixDecompositions and their noncommutative versions in the new package NCMatrixDecompositions.
8. General improvements on the Semidefinite Programming package NCSDP.
9. New algorithms for simplification of noncommutative rationals (NCSimplifyRational).
10. Commands `Transform`, `Substitute`, `SubstituteSymmetric`, etc, have been replaced by the much more reliable commands in the new package NCReplace.
11. Command `MatMult` has been replaced by NCDot. Alias `MM` has been deprecated.
12. Noncommutative power is now supported, with `x^3` expanding to `x**x**x`, `x^-1` expanding to `inv[x]`.
13. `x^T` expands to `tp[x]` and `x^*` expands to `aj[x]`. Symbol `T` is now protected.
14. Support for subscripted variables in notebooks.

Chapter 3

Introduction

This *User Guide* attempts to document the many improvements introduced in `NCAlgebra` Version 5.0. Please be patient, as we move to incorporate the many recent changes into this document.

See Reference Manual for a detailed description of the available commands.

There are also notebooks in the `NC/DEMOS` directory that accompany each of the chapters of this user guide.

3.1 Running NCAlgebra

In *Mathematica* (notebook or text interface), type

```
<< NC`
```

If this step fails, your installation has problems (check out installation instructions on the main page). If your installation is succesful you will see a message like:

```
You are using the version of NCAlgebra which is found in:
   /your_home_directory/NC.
You can now use "<< NCAlgebra`" to load NCAlgebra.
```

Then just type

```
<< NCAlgebra`
```

to load `NCAlgebra`.

Advanced options for controlling the loading of `NC` and `NCAlgebra` can be found in here and here.

3.2 Now what?

Extensive documentation is found in the directory `DOCUMENTATION`, including this document.

Basic documentation is found in the project wiki:

https://github.com/NCAlgebra/NC/wiki

You may want to try some of the several demo files in the directory `DEMOS` after installing `NCAlgebra`.

You can also run some tests to see if things are working fine.

3.3 Testing

You do not need to load `NCAlgebra` before running any of the tests below, but you need to load `NC` as in

```
<< NC`
```

There are 3 test sets which you can use to troubleshoot parts of NCAlgebra. The most comprehensive test set is run by typing:

```
<< NCTEST
```

This will test the core functionality of NCAlgebra.

You can test functionality related to the package `NCPoly`, including the new `NCGBX` package `NCGBX`, by typing:

```
<< NCPOLYTEST
```

Finally our Semidefinite Programming Solver `NCSDP` can be tested with

```
<< NCSDPTEST
```

We recommend that you restart the kernel before and after running tests. Each test takes a few minutes to run.

You can also call

```
<< NCPOLYTESGB
```

to perform extensive and long testing of `NCGBX`.

3.4 Pre-2017 NCGB C++ version

The old `C++` version of our Groebner Basis Algorithm still ships with this version and can be loaded using:

```
<< NCGB`
```

This will at once load `NCAlgebra` *and* `NCGB`. It can be tested using

```
<< NCGBTEST
```

Chapter 4

Most Basic Commands

This chapter provides a gentle introduction to some of the commands available in `NCAlgebra`.

If you want a living analog of this chapter just run the notebook `NC/DEMOS/1_MostBasicCommands.nb`.

Before you can use `NCAlgebra` you first load it with the following commands:

```
<< NC`
<< NCAlgebra`
```

4.1 To Commute Or Not To Commute?

In `NCAlgebra`, the operator `**` denotes *noncommutative multiplication.* At present, single-letter lower case variables are noncommutative by default and all others are commutative by default. For example:

```
a**b-b**a
```

results in

```
a**b-b**a
```

while

```
A**B-B**A
A**b-b**A
```

both result in `0`.

One of Bill's favorite commands is `CommuteEverything`, which temporarily makes all noncommutative symbols appearing in a given expression to behave as if they were commutative and returns the resulting commutative expression. For example:

```
CommuteEverything[a**b-b**a]
```

results in `0`. The command

```
EndCommuteEverything[]
```

restores the original noncommutative behavior.

One can make any symbol behave as noncommutative using `SetNonCommutative`. For example:

```
SetNonCommutative[A,B]
A**B-B**A
```

results in:

```
A**B-B**A
```

Likewise, symbols can be made commutative using `SetCommutative`. For example:

```
SetNonCommutative[A]
SetCommutative[B]
A**B-B**A
```

results in `0`. `SNC` is an alias for `SetNonCommutative`. So, `SNC` can be typed rather than the longer `SetNonCommutative`:

```
SNC[A];
A**a-a**A
```

results in:

```
-a**A+A**a
```

One can check whether a given symbol is commutative or not using `CommutativeQ` or `NonCommutativeQ`. For example:

```
CommutativeQ[B]
NonCommutativeQ[a]
```

both return `True`.

4.2 Inverses, Transposes and Adjoints

The multiplicative identity is denoted `Id` in the program. At the present time, `Id` is set to 1.

A symbol `a` may have an inverse, which will be denoted by `inv[a]`. `inv` operates as expected in most cases.

For example:

```
inv[a]**a
inv[a**b]**a**b
```

both lead to `Id = 1` and

```
a**b**inv[b]
```

results in `a`.

Version 5: `inv` no longer automatically distributes over noncommutative products. If this more aggressive behavior is desired use `SetOptions[inv, Distribute -> True]`. For example

```
SetOptions[inv, Distribute -> True]
inv[a**b]
```

returns `inv[b]**inv[a]`. Conversely

```
SetOptions[inv, Distribute -> False]
inv[a**b]
```

returns `inv[a**b]`.

`tp[x]` denotes the transpose of symbol `x` and `aj[x]` denotes the adjoint of symbol `x`. Like `inv`, the properties of transposes and adjoints that everyone uses constantly are built-in. For example:

```
tp[a**b]
```

leads to

```
tp[b]**tp[a]
```

and

```
tp[a+b]
```

returns

```
tp[a]+tp[b]
```

Likewise `tp[tp[a]] == a` and `tp` for anything for which `CommutativeQ` returns `True` is simply the identity. For example `tp[5] == 5`, `tp[2 + 3I] == 2 + 3 I`, and `tp[B] == B`.

Similar properties hold to `aj`. Moreover

```
aj[tp[a]]
tp[aj[a]]
```

return `co[a]` where `co` stands for complex-conjugate.

Version 5: transposes (`tp`), adjoints (`aj`), complex conjugates (`co`), and inverses (`inv`) in a notebook environment render as x^T, x^*, $\bar{x}$, and x^{-1}. `tp` and `aj` can also be input directly as `x^T` and `x^*`. For this reason the symbol `T` is now protected in `NCAlgebra`.

A trace like operator, `tr`, was introduced in v5.0.6. It is a commutative operator keeps its list of arguments cyclicly sorted so that `tr[b ** a]` evaluates to `tr[a ** b]` and that automatically distribute over sums so that an expression like

```
tr[a ** b - b ** a]
```

always simplifies to zero. Also `b ** a ** tr[b ** a]` simpplifies to

```
tr[a ** b] a ** b
```

because `tr` is a commutative function. See SetCommutativeFunction.

Use NCMatrixExpand to expand `tr` over matrices with noncommutative entries. For example,

```
NCMatrixExpand[tr[{{a,b},{c,d}}]]
```

evaluates to

```
tr[a] + tr[b]
```

4.3 Replace

A key feature of symbolic computation is the ability to perform substitutions. The Mathematica substitute commands, e.g. `ReplaceAll` (`/.`) and `ReplaceRepeated` (`//.`), are not reliable in `NCAlgebra`, so you must use our `NC` versions of these commands. For example:

```
NCReplaceAll[x**a**b,a**b->c]
```

results in

```
x**c
```

and

```
NCReplaceAll[tp[b**a]+b**a,b**a->c]
```

results in

```
c+tp[a]**tp[b]
```

Use NCMakeRuleSymmetric and NCMakeRuleSelfAdjoint to automatically create symmetric and self adjoint versions of your rules:

```
NCReplaceAll[tp[b**a]+b**a, NCMakeRuleSymmetric[b**a -> c]]
```

returns

```
c + tp[c]
```

The difference between `NCReplaceAll` and `NCReplaceRepeated` can be understood in the example:

```
NCReplaceAll[a**b**b, a**b -> a]
```

that results in

```
a**b
```

and

```
NCReplaceRepeated[a**b**b, a**b -> a]
```

that results in

```
a
```

Beside `NCReplaceAll` and `NCReplaceRepeated` we offer `NCReplace` and `NCReplaceList`, which are analogous to the standard `ReplaceAll` (`/.`), `ReplaceRepeated` (`//.`), `Replace` and `ReplaceList`. Note that one rarely uses `NCReplace` and `NCReplaceList`.

See the Section Advanced Rules and Replacement for a deeper discussion on some issues involved with rules and replacements in `NCAlgebra`.

Version 5: the commands `Substitute` and `Transform` have been deprecated in favor of the above nc versions of `Replace`.

4.4 Polynomials

The command `NCExpand` expands noncommutative products. For example:

```
NCExpand[(a+b)**x]
```

returns

```
a**x+b**x
```

Conversely, one can collect noncommutative terms involving same powers of a symbol using `NCCollect`. For example:

```
NCCollect[a**x+b**x,x]
```

recovers

```
(a+b)**x
```

`NCCollect` groups terms by degree before collecting and accepts more than one variable. For example:

```
expr = a**x+b**x+y**c+y**d+a**x**y+b**x**y
NCCollect[expr, {x}]
```

returns

```
y**c+y**d+(a+b)**x**(1+y)
```

and

```
NCCollect[expr, {x, y}]
```

returns

```
(a+b)**x+y**(c+d)+(a+b)**x**y
```

Note that the last term has degree 2 in `x` and `y` and therefore does not get collected with the first order terms.

The list of variables accepts `tp`, `aj` and `inv`, and

```
NCCollect[tp[x]**a**x+tp[x]**b**x+z,{x,tp[x]}]
```

returns

```
z+tp[x]**(a+b)**x
```

Alternatively one could use

```
NCCollectSymmetric[tp[x]**a**x+tp[x]**b**x+z,{x}]
```

to obtain the same result. A similar command, NCCollectSelfAdjoint, works with self-adjoint variables.

There is also a stronger version of collect called `NCStrongCollect`. `NCStrongCollect` does not group terms by degree. For instance:

```
NCStrongCollect[expr, {x, y}]
```

produces

```
y**(c+d)+(a+b)**x**(1+y)
```

Keep in mind that `NCStrongCollect` often collects *more* than one would normally expect.

`NCAlgebra` provides some commands for noncommutative polynomial manipulation that are similar to the native Mathematica (commutative) polynomial commands. For example:

```
expr = B + A y**x**y - 2 x
NCVariables[expr]
```

returns

```
{x,y}
```

and

```
NCCoefficientList[expr, vars]
NCMonomialList[expr, vars]
NCCoefficientRules[expr, vars]
```

returns

```
{B, -2, A}
{1, x, y**x**y}
{1 -> B, x -> -2, y**x**y -> A}
```

Also for testing

```
NCMonomialQ[expr]
```

will return `False` and

```
NCPolynomialQ[expr]
```

will return `True`.

Another useful command is `NCTermsOfDegree`, which will returns an expression with terms of a certain degree. For instance:

```
NCTermsOfDegree[x**y**x - x**x**y + x**w + z**w, {x,y}, {2,1}]
```

returns `x**y**x - x**x**y`,

```
NCTermsOfDegree[x**y**x - x**x**y + x**w + z**w, {x,y}, {0,0}]
```

returns `z**w`, and

```
NCTermsOfDegree[x**y**x - x**x**y + x**w + z**w, {x,y}, {0,1}]
```

returns `0`.

A similar command is `NCTermsOfTotalDegree`, which works just like `NCTermsOfDegree` but considers the total degree in all variables. For example:

For example,

```
NCTermsOfTotalDegree[x**y**x - x**x**y + x**w + z**w, {x,y}, 3]
```

returns `x**y**x - x**x**y`, and

```
NCTermsOfTotalDegree[x**y**x - x**x**y + x**w + z**w, {x,y}, 2]
```

returns `0`.

The above commands are based on special packages for efficiently storing and calcuating with nc polynomials. Those packages are

- `NCPoly`: which handles polynomials with noncommutative coefficients, and
- `NCPolynomial`: which handles polynomials with real coefficients.

For example:

```
1 + y**x**y - A x
```

is a polynomial with commutative coefficients in x and y, whereas

```
a**y**b**x**c**y - A x**d
```

is a polynomial with nc coefficients in x and y, where the letters a, b, c, and d, are the *nc coefficients*. Of course

```
1 + y**x**y - A x
```

is a polynomial with nc coefficients if one considers only x as the variable of interest.

In order to take full advantage of `NCPoly` and `NCPolynomial` one would need to *convert* an expression into those special formats. See NCPolyInterface, NCPoly, and NCPolynomial for details.

4.5 Rationals and Simplification

One of the great challenges of noncommutative symbolic algebra is the simplification of rational nc expressions. `NCAlgebra` provides various algorithms that can be used for simplification and general manipulation of nc rationals.

One such function is `NCSimplifyRational`, which attempts to simplify noncommutative rationals using a predefined set of rules. For example:

```
expr = 1+inv[d]**c**inv[S-a]**b-inv[d]**c**inv[S-a+b**inv[d]**c]**b \
       -inv[d]**c**inv[S-a+b**inv[d]**c]**b**inv[d]**c**inv[S-a]**b
NCSimplifyRational[expr]
```

leads to `1`. Of course the great challenge here is to reveal well known identities that can lead to simplification. For example, the two expressions:

```
expr1 = a**inv[1+b**a]
expr2 = inv[1+a**b]**a
```

and one can use `NCSimplifyRational` to test such equivalence by evaluating

```
NCSimplifyRational[expr1 - expr2]
```

which results in `0` or

```
NCSimplifyRational[expr1**inv[expr2]]
```

which results in `1`. `NCSimplifyRational` works by transforming nc rationals. For example, one can verify that

```
NCSimplifyRational[expr2] == expr1
```

`NCAlgebra` has a number of packages that can be used to manipulate rational nc expressions. The packages:

- `NCGBX` perform calculations with nc rationals using Gröbner basis, and
- `NCRational` creates state-space representations of nc rationals. This package is still experimental.

4.6 Calculus

The package `NCDiff` provide functions for calculating derivatives and integrals of nc polynomials and nc rationals.

The main command is `NCDirectionalD` which calculates directional derivatives in one or many variables. For example, if:

```
expr = a**inv[1+x]**b + x**c**x
```

then

```
NCDirectionalD[expr, {x,h}]
```

returns

```
h**c**x + x**c**h - a**inv[1+x]**h**inv[1+x]**b
```

In the case of more than one variables `NCDirectionalD[expr, {x,h}, {y,k}]` takes the directional derivative of `expr` with respect to `x` in the direction `h` and with respect to `y` in the direction `k`. For example, if:

```
expr = x**q**x - y**x
```

then

```
NCDirectionalD[expr, {x,h}, {y,k}]
```

returns

```
h**q**x + x**q*h - y**h - k**x
```

A further example, if:

```
expr = x**a**x**b + x**c**x**d
```

then its directional derivative in the direction `h` is

```
NCDirectionalD[expr, {x,h}]
```

which returns

```
h**a**x**b + x**a**h**b + h**c**x**d + x**c**h**d
```

The command `NCGrad` calculates nc *gradients*[1].

For example:

```
NCGrad[expr, x]
```

returns the nc gradient

```
a**x**b + b**x**a + c**x**d + d**x**c
```

A further example, if:

[1] The transpose of the gradient of the nc expression `expr` is the derivative with respect to the direction `h` of the trace of the directional derivative of `expr` in the direction `h`.

```
expr = x**a**x**b + x**c**y**d
```

is a function on variables `x` and `y` then

```
NCGrad[expr, x, y]
```

returns the nc gradient list

```
{a**x**b + b**x**a + c**y**d, d**x**c}
```

Version 5: introduces experimental support for integration of nc polynomials. See `NCIntegrate`.

4.7 Matrices

`NCAlgebra` has many commands for manipulating matrices with noncommutative entries. Think block-matrices. Matrices are represented in Mathematica using *lists of lists*. For example

```
m = {{a, b}, {c, d}}
```

is a representation for the matrix

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix}$$

The Mathematica command `MatrixForm` output pretty matrices. `MatrixForm[m]` prints `m` in a form similar to the above matrix. Beware when copying and pasting parts of an expression rendered by`MatrixForm` because it may not execute correctly. If in doubt, use `FullForm` to reveal the contents of the expression.

The experienced matrix analyst should always remember that the Mathematica convention for handling vectors is tricky.

- `{{1, 2, 4}}` is a 1x3 *matrix* or a *row vector*;
- `{{1}, {2}, {4}}` is a 3x1 *matrix* or a *column vector*;
- `{1, 2, 4}` is a *vector* but **not** a *matrix*. Indeed whether it is a row or column vector depends on the context. We advise not to use *vectors*.

4.7.1 Inverses, products, adjoints, etc

A useful command is `NCInverse`, which is akin to Mathematica's `Inverse` command and produces a block-matrix inverse formula[2] for an nc matrix. For example

```
NCInverse[m]
```

returns

```
{{inv[a]**(1 + b**inv[d - c**inv[a]**b]**c**inv[a]), -inv[a]**b**inv[d - c**inv[a]**b]},
 {-inv[d - c**inv[a]**b]**c**inv[a], inv[d - c**inv[a]**b]}}
```

or, using `MatrixForm`

```
NCInverse[m] // MatrixForm NCInverse[m] // MatrixForm
```

returns

$$\begin{bmatrix} a^{-1}(1+b(d-ca^{-1}b)^{-1}ca^{-1}) & -a^{-1}b(d-ca^{-1}b)^{-1} \\ -(d-ca^{-1}b)^{-1}ca^{-1} & (d-ca^{-1}b)^{-1} \end{bmatrix}$$

Note that `a` and `d - c**inv[a]**b` were assumed invertible during the calculation.

Similarly, one can multiply matrices using `NCDot`, which is similar to Mathematica's `Dot`. For example

[2] Contrary to what happens with symbolic inversion of matrices with commutative entries, there exist multiple formulas for the symbolic inverse of a matrix with noncommutative entries. Furthermore, it may be possible that none of such formulas is "correct". Indeed, it is easy to construct a matrix `m` with block structure as shown that is invertible but for which none of the blocks `a`, `b`, `c`, and `d` are invertible. In this case no *correct* formula exists for the calculation of the inverse of `m`.

```
m1 = {{a, b}, {c, d}}
m2 = {{d, 2}, {e, 3}}
NCDot[m1, m2]
```

result in

```
{{a ** d + b ** e, 2 a + 3 b}, {c ** d + d ** e, 2 c + 3 d}}
```

Note that products of nc symbols appearing in the matrices are multiplied using `**`. Compare that with the standard `Dot` (`.`) operator.

WARNING: `NCDot` replaces `MatMult`, which is still available for backward compatibility but will be deprecated in future releases.

There are many new improvements with **Version 5**. For instance, operators `tp`, `aj`, and `co` now operate directly over matrices. That is

```
aj[{{a,tp[b]},{co[c],aj[d]}}]
```

returns

```
{{aj[a],tp[c]},{co[b],d}}
```

In previous versions one had to use the special commands `tpMat`, `ajMat`, and `coMat`. Those are still supported for backward compatibility.

See advanced matrix commands for other useful matrix manipulation routines, such as `NCMatrixExpand`, `NCMatrixReplaceAll`, `NCMatrixReplaceRepeated`, etc, that allow one to work with matrices with symbolic noncommutative entries.

4.7.2 LU Decomposition

Behind `NCInverse` there are a host of linear algebra algorithms which are available in the package:

- `NCMatrixDecompositions`: implements versions of the LU Decomposition with partial and complete pivoting, as well as LDL Decomposition which are suitable for calculations with nc matrices. Those functions are based on the templated algorithms from the package `MatrixDecompositions`.

For instance the function `NCLUDecompositionWithPartialPivoting` can be used as

```
m = {{a, b}, {c, d}}
{lu, p} = NCLUDecompositionWithPartialPivoting[m]
```

which returns

```
lu = {{a, b}, {c**inv[a], d - c**inv[a]**b}}
p = {1, 2}
```

Using `MatrixForm`:

$$\begin{bmatrix} a & b \\ ca^{-1} & d - ca^{-1}b \end{bmatrix}$$

The list `p` encodes the sequence of permutations calculated during the execution of the algorithm. The matrix `lu` contains the factors L and U in the way most common to numerical analysts. These factors can be recovered using

```
{l, u} = GetFullLUMatrices[lu]
```

resulting in this case in

```
l = {{1, 0}, {c**inv[a], 1}}
u = {{a, b}, {0, d - c**inv[a]**b}}
```

Using `MatrixForm`:

$$L = \begin{bmatrix} 1 & 0 \\ ca^{-1} & 1 \end{bmatrix}, \qquad U = \begin{bmatrix} a & b \\ 0 & d - ca^{-1}b \end{bmatrix}$$

To verify that $M = LU$ input

```
m - NCDot[l, u]
```

which should return a zero matrix.

Note: if you are looking for efficiency, the function `GetLUMatrices` (also `GetLDUMatrices`) returns the factors `l` and `u` as `SparseArrays`.

The default pivoting strategy prioritizes pivoting on simpler expressions. For instance,

```
m = {{a, b}, {1, d}}
{lu, p} = NCLUDecompositionWithPartialPivoting[m]
{l, u} = GetFullLUMatrices[lu]
```

results in the factors

```
l = {{1, 0}, {a, 1}}
u = {{1, d}, {0, b - a**d}}
```

and a permutation list

```
p = {2, 1}
```

which indicates that the number `1`, appearing in the second row, was used as the pivot rather than the symbol `a` appearing on the first row. Because of the permutation, to verify that $PM = LU$ input

```
m[[p]] - NCDot[l, u]
```

which should return a zero matrix. Note that in the above example the permutation matrix P is never constructed. Instead, the rows of M are directly permuted using Mathematica's `Part` (`[[]]`) command. Of course, if one prefers to work with permutation matrices, they can be easily obtained by permuting the rows of the identity matrix as in the following example

```
p = {2, 1, 3}
IdentityMatrix[3][[p]] // MatrixForm
```

to produce

$$\begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Likewise

```
m = {{a + b, b}, {c, d}}
{lu, p} = NCLUDecompositionWithPartialPivoting[m]
{l, u} = GetFullLUMatrices[lu]
```

returns

```
p = {2, 1}
l = {{1, 0}, {(a + b)**inv[c], 1}}
u = {{c, d}, {0, b - (a + b)**inv[c]**d}}
```

showing that the *simpler* expression `c` was taken as a pivot instead of `a + b`.

The function `NCLUDecompositionWithPartialPivoting` is the one that is used by `NCInverse`.

4.7.3 LU Decomposition with complete pivoting

Another factorization algorithm is `NCLUDecompositionWithCompletePivoting`, which can be used to calculate the symbolic rank of nc matrices. For example

```
m = {{2 a, 2 b}, {a, b}}
{lu, p, q, rank} = NCLUDecompositionWithCompletePivoting[m]
```

returns the *left* and *right* permutation lists

```
p = {2, 1}
q = {1, 2}
```

and `rank` equal to `1`. Note that `p = {2, 1}` and `q = {1,2}` tell us that the element that was pivoted on was the symbol `a`, which is the first entry of the second row, rather then `2 a`, which is the first entry of the first row, because `a` is *simpler* than `2 a` . The L and U factors can be obtained as before using

```
{l, u} = GetFullLUMatrices[lu]
```

to get

```
l = {{1, 0}, {2, 1}}
u = {{a, b}, {0, 0}}
```

Using `MatrixForm`:

$$L = \begin{bmatrix} 1 & 0 \\ 2 & 1 \end{bmatrix}, \qquad U = \begin{bmatrix} a & b \\ 0 & 0 \end{bmatrix}$$

In this case, to verify that $PMQ = LU$ input

```
NCDot[l, u] - m[[p, q]]
```

which should return a zero matrix. As with partial pivoting, the permutation matrices P and Q are never constructed. Instead we used `Part` (`[[]]`) to permute both rows and columns.

4.7.4 LDL Decomposition

Finally `NCLDLDecomposition` computes the LDL^T decomposition of symmetric symbolic nc matrices. For example

```
m = {{a, b}, {b, c}}
{ldl, p, s, rank} = NCLDLDecomposition[m]
```

returns `ldl`, which contain the factors, and

```
p = {1, 2}
s = {1, 1}
rank = 2
```

The list `p` encodes left and right permutations, `s` is a list specifying the size of the diagonal blocks (entries can be either 1 or 2). The factors can be obtained using `GetLDUMatrices` as in

```
{l, d, u} = GetFullLDUMatrices[ldl, s]
```

which in this case returns

```
l = {{1, 0}, {b**inv[a], 1}}
d = {{a, 0}, {0, c - b**inv[a]**b}}
u = {{1, inv[a]**b}, {0, 1}}}
```

Because $PMP^T = LDL^T$,

```
NCDot[l, d, u] - m[[p, p]]
```

is the zero matrix and $U = L^T$.

`NCLDLDecomposition` works only on symmetric matrices and, whenever possible, will make invertibility and symmetry assumptions on variables so that it can run successfully. If not possible it will warn the users.

WARNING: Versions prior to 5 contained the command `NCLDUDecomposition` which is being deprecated in **Version 5** as its functionality is now provided by `NCLDLDecomposition`, with a slightly different syntax.

4.7.5 Replace with matrices

`NCMatrixReplaceAll` and `NCMatrixReplaceRepeated` are special versions of `NCReplaceAll` and `NCReplaceRepeated` that take extra steps to preserve matrix consistency when replacing expressions with nc matrices. For example, with

```
m1 = {{a, b}, {c, d}}
m2 = {{d, 2}, {e, 3}}
```

and

```
M = {{a11,a12}}
```

the call

```
NCMatrixReplaceRepeated[M, {a11 -> m1, a12 -> m2}]
```

produces as a result the matrix

```
{{a, b, d, 2}, {c, d, e, 3}}
```

or, using `MatrixForm`:

$$\begin{bmatrix} a & b & d & 2 \\ c & d & e & 3 \end{bmatrix}$$

Note how the symbols were treated as block-matrices during the substitution. As a second example, with

```
M = {{a11, 0}, {0, a22}}
```

the command

```
NCMatrixReplaceRepeated[M, {a11 -> m1, a22 -> m2}]
```

produces the matrix

```
{{a, b, 0, 0}, {c, d, 0, 0}, {0, 0, d, 2}, {0, 0, e, 3}}
```

or, using `MatrixForm`:

$$\begin{bmatrix} a & b & 0 & 0 \\ c & d & 0 & 0 \\ 0 & 0 & d & 2 \\ 0 & 0 & e & 3 \end{bmatrix}$$

in which the `0` blocks were automatically expanded to fit the adjacent block matrices.

Another feature of `NCMatrixReplace` and its variants is its ability to withold evaluation until all matrix substitutions have taken place. For example,

```
NCMatrixReplaceAll[x**y + y, {x -> m1, y -> m2}]
```

produces

```
{{d + a**d + b**e, 2 + 2 a + 3 b},
 {e + c**d + d**e, 3 + 2 c + 3 d}}
```

Finally, `NCMatrixReplace` substitutes `NCInverse` for `inv` so that, for instance, the result of

```
rule = {x -> m1, y -> m2, id -> IdentityMatrix[2], z -> {{id,x},{x,id}}}
NCMatrixReplaceRepeated[inv[z], rule]
```

coincides with

```
NCInverse[ArrayFlatten[{{IdentityMatrix[2], m1}, {m1, IdentityMatrix[2]}}]]
```

4.8 Quadratic polynomials, second direction derivatives and convexity

The closest related demo to the material in this section is `NC/DEMOS/NCConvexity.nb`.

When working with nc quadratics it is useful to be able to "factor" the quadratic into the following form

$$q(x) = c + s(x) + l(x)Mr(x)$$

where s is linear x and l and r are vectors and M is a matrix. Load the package

```
<< NCQuadratic`
```

and use the command `NCToNCQuadratic` to factor an nc polynomial into the the above form:

```
vars = {x, y};
expr = tp[x]**a**x**d + tp[x]**b**y + tp[y]**c**y + tp[y]**tp[b]**x**d;
{const, lin, left, middle, right} = NCToNCQuadratic[expr, vars];
```

which returns

```
left = {tp[x],tp[y]}
right = {y, x**d}
middle = {{a,b}, {tp[b],c}}
```

and zero `const` and `lin`. The format for the linear part `lin` will be discussed lated in Section Linear. Note that coefficients of an nc quadratic may also appear on the left and right vectors, as `d` did in the above example. Conversely, `NCQuadraticToNC` converts a list with factors back to an nc expression as in:

```
NCQuadraticToNC[{const, lin, left, middle, right}]
```

which results in

```
(tp[x]**b + tp[y]**c)**y + (tp[x]**a + tp[y]**tp[b])**x**d
```

An interesting application is the verification of the domain in which an nc rational function is *convex*. This uses the second direction derivative, called the Hessian. Take for example the quartic

```
expr = x**x**x**x;
```

and calculate its noncommutative directional *Hessian*

```
hes = NCHessian[expr, {x, h}]
```

This command returns

```
2 h**h**x**x + 2 h**x**h**x + 2 h**x**x**h + 2 x**h**h**x + 2 x**h**x**h + 2 x**x**h**h
```

which is quadratic in the direction `h`. The decomposition of the nc Hessian using `NCToNCQuadratic`

```
{const, lin, left, middle, right} = NCToNCQuadratic[hes, {h}];
```

produces

```
left = {h, x**h, x**x**h}
right = {h**x**x, h**x, h}
middle = {{2, 2 x, 2 x**x},{0, 2, 2 x},{0, 0, 2}}
```

Note that the middle matrix

$$\begin{bmatrix} 2 & 2x & 2x^2 \\ 0 & 2 & 2x \\ 0 & 0 & 2 \end{bmatrix}$$

is not *symmetric*, as one might have expected. The command NCQuadraticMakeSymmetric can fix that and produce a symmetric decomposition. For the above example

```
{const, lin, sleft, smiddle, sright} =
  NCQuadraticMakeSymmetric[{const, lin, left, middle, right},
                           SymmetricVariables -> {x, h}]
```

results in

```
sleft = {x**x**h, x**h, h}
sright = {h**x**x, h**x, h}
middle = {{0, 0, 2}, {0, 2, 2 x}, {2, 2 x, 2 x**x}}
```

in which `middle` is the symmetric matrix

$$\begin{bmatrix} 0 & 0 & 2 \\ 0 & 2 & 2x \\ 2 & 2x & 2x^2 \end{bmatrix}$$

Note the argument `SymmetricVariables -> {x,h}` which tells `NCQuadraticMakeSymmetric` to consider `x` and `y` as symmetric variables. Because the `middle` matrix is never positive semidefinite for any possible value of x the conclusion[3] is that the nc quartic x^4 is *not convex*.

The production of such symmetric quadratic decompositions is automated by the convenience command NCMatrixOfQuadratic. Verify that

```
{sleft, smiddle, sright} = NCMatrixOfQuadratic[hes, {h}]
```

automatically assumes that both `x` and `h` are symmetric variables and produces suitable left and right vectors as well as a symmetric middle matrix. Now we illustrate the application of such command to checking the convexity region of a noncommutative rational function.

If one is interested in checking convexity of nc rationals the package NCConvexity has functions that automate the whole process, including the calculation of the Hessian and the middle matrix, followed by the diagonalization of the middle matrix as produced by NCLDLDecomposition.

For example, the commands evaluate the nc Hessian and calculates its quadratic decomposition

```
expr = (x + b**y)**inv[1 - a**x**a + b**y + y**b]**(x + y**b);
{left, middle, right} = NCMatrixOfQuadratic[NCHessian[expr, {x, h}], {h}];
```

The resulting middle matrix can be factored using

```
{ldl, p, s, rank} = NCLDLDecomposition[middle];
{ll, dd, uu} = GetLDUMatrices[ldl, s];
```

which produces the diagonal factors

$$\begin{bmatrix} 2(1 + by + yb - axa)^{-1} & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

which indicates the the original nc rational is convex whenever

$$(1 + by + yb - axa)^{-1} \succeq 0$$

or, equivalently, whenever

$$1 + by + yb - axa \succeq 0$$

The above sequence of calculations is automated by the command NCConvexityRegion as in

[3]This is in contrast with the commutative x^4 which is convex everywhere. See [Cam+03] for details.

```
<< NCConvexity`
NCConvexityRegion[expr, {x}]
```

which results in

```
{2 inv[1 + b**y + y**b - a**x**a], 0}
```

which correspond to the diagonal entries of the LDL decomposition of the middle matrix of the nc Hessian.

Chapter 5

More Advanced Commands

In this chapter we describe some more advance features and commands. Most of these were introduced in **Version 5**.

If you want a living version of this chapter just run the notebook `NC/DEMOS/2_MoreAdvancedCommands.nb`.

5.1 Advanced Rules and Replacements

Substitution is a key feature of Symbolic computation. We will now discuss some issues related to Mathematica's implementation of rules and replacements that can affect the behavior of `NCAlgebra` expressions.

5.1.1 `ReplaceAll` (`/.`) and `ReplaceRepeated` (`//.`) often fail

The first issue is related to how Mathematica performs substitutions, which is through pattern matching. For a rule to be effective if has to match the *structural* representation of an expression. That representation might be different than one would normally think based on the usual properties of mathematical operators. For example, one would expect the rule:

```
rule = 1 + x_ -> x
```

to match all the expressions bellow:

```
1 + a
1 + 2 a
1 + a + b
1 + 2 a * b
```

so that

```
expr /. rule
```

with `expr` taking the above expressions would result in:

```
a
2 a
a + b
2 a * b
```

Indeed, Mathematica's attribute `Flat` does precisely that. Note that this is still *structural matching*, not *mathematical matching*, since the pattern `1 + x_` would not match an integer `2`, even though one could write `2 = 1 + 1`!

Unfortunately, `**`, which is the `NonCommutativeMultiply` operator, *is not* `Flat`[1]. This is the reason why substitution based on a simple rule such as:

```
rule = a**b -> c
```

so that

```
expr /. rule
```

will work for some `expr` like

```
1 + 2 a**b
```

resulting in

```
1 + 2 c
```

but will fail to produce the *expected* result in cases like:

```
a**b**c
c**a**b
c**a**b**d
1 + 2 a**b**c
```

That's what the `NCAlgebra` family of replacement functions discussed in the next section are made for.

5.1.2 The fix is `NCReplace`

Continuing with the example in the previous section, the calls

```
NCReplaceAll[a**b**c, rule]
NCReplaceAll[ c**a**b, rule ]
NCReplaceAll[c**a**b**d, rule]
NCReplaceAll[1 + 2 a**b**c, rule ]
```

produces the results one would expect:

```
c**c
c**c
c**c**d
1 + 2 c**c
```

For this reason, when substituting in `NCAlgebra` it is always safer to use functions from the `NCReplace` package rather than the corresponding Mathematica `Replace` family of functions. Unfortunately, this comes at a the expense of sacrificing the standard operators `/.` (`ReplaceAll`) and `//.` (`ReplaceRepeated`), which cannot be safely overloaded, forcing one to use the full names `NCReplaceAll` and `NCReplaceRepeated`.

On the same vein, the following substitution rule

```
NCReplace[2 a**b + c, 2 a -> b]
```

will return `2 a**b + c` intact since `FullForm[2 a**b]` is indeed

```
Times[2, NonCommutativeMuliply[a, b]]
```

which is not structurally related to `FullForm[2 a]`, which is `Times[2, a]`. Of course, in this case a simple solution is to use the alternative rule:

```
NCReplace[2 a**b + c, a -> b / 2]
```

which results in `b**b + c`, as one might expect.

[1] The reason is that making an operator `Flat` is a convenience that comes with a price: lack of control over execution and evaluation. Since `NCAlgebra` has to operate at a very low level this lack of control over evaluation is fatal. Indeed, making `NonCommutativeMultiply` have an attribute `Flat` will throw Mathematica into infinite loops in seemingly trivial noncommutative expression. Hey, email us if you find a way around that :)

5.1.3 Trouble with `Block` and `Module`

A second more esoteric issue related to substitution in `NCAlgebra` does not have a clean solution. It is also one that usually lurks in hidden pieces of code and can be very difficult to spot. We have been victim of such "bugs" many times. Luckily it only affect advanced users that are using `NCAlgebra` inside their own functions using Mathematica's `Block` and `Module` constructions. It is also not a real bug, since it follows from some often not well understood issues with the usage of `Block` versus `Module`. Our goal here is therefore not to *fix* the issue but simply to alert advanced users of its existence. Start by first revisiting the following example from the Mathematica documentation. Let

```
m = i^2
```

and run

```
Block[{i = a}, i + m]
```

which returns the "expected''

```
a + a**a
```

versus

```
Module[{i = a}, i + m]
```

which returns the "surprising''

```
a + i**i
```

The reason for this behavior is that `Block` effectively evaluates `i` as a *local variable* and evaluates `m` using whatever values are available at the time of evaluation, whereas `Module` only evaluates the symbol `i` which appears *explicitly* in `i + m` and not `m` using the local value of `i = a`. This can lead to many surprises when using rules and substitution inside a `Module`. For example:

```
Block[{i = a}, i_ -> i]
```

will return

```
i_ -> a
```

whereas

```
Module[{i = a}, i_ -> i]
```

will return

```
i_ -> i
```

More devastating for `NCAlgebra` is the fact that `Module` will hide local definitions from rules, which will often lead to disaster if local variables need to be declared noncommutative. Consider for example the trivial definitions for `F` and `G` below:

```
F[exp_] := Module[
  {rule, aa, bb},
  SetNonCommutative[aa, bb];
  rule = aa_**bb_ -> bb**aa;
  NCReplaceAll[exp, rule]
]

G[exp_] := Block[
  {rule, aa, bb},
  SetNonCommutative[aa, bb];
  rule = aa_**bb_ -> bb**aa;
  NCReplaceAll[exp, rule]
]
```

Their only difference is that one is defined using a `Block` and the other is defined using a `Module`. The task is to apply a rule that *flips* the noncommutative product of their arguments, say, `x**y`, into `y**x`. The problem is that only one of those definitions work "as expected''. Indeed, verify that

```
G[x**y]
```

returns the "expected''

```
y**x
```

whereas

```
F[x**y]
```

returns

```
x y
```

which completely destroys the noncommutative product. The reason for the catastrophic failure of the definition of `F`, which is inside a `Module`, is that the letters `aa` and `bb` appearing in `rule` are *not treated as the local symbols* `aa` *and* `bb`[2]. For this reason, the right-hand side of the rule `rule` involves the global symbols `aa` and `bb`, which are, in the absence of a declaration to the contrary, commutative. On the other hand, the definition of `G` inside a `Block` makes sure that `aa` and `bb` are evaluated with whatever value they might have locally at the time of execution.

The above subtlety often manifests itself partially, sometimes causing what might be perceived as some kind of *erratic behavior*. Indeed, if one had used symbols that were already declared globaly noncommutative by `NCAlgebra`, such as single small cap roman letters as in the definition:

```
H[exp_] := Module[
  {rule, a, b},
  SetNonCommutative[a, b];
  rule = a_**b_ -> b**a];
  NCReplaceAll[exp, rule]
]
```

then calling `H[x**y]` would have worked "as expected'', even if for the wrong reasons!

Another possible "fix" is to use a delayed rule, as in:

```
H[exp_] := Module[
  {rule, aa, bb},
  SetNonCommutative[aa, bb];
  rule = aa_**bb_ :> bb**aa];
  NCReplaceAll[exp, rule]
]
```

which would also work as the evaluation of the right-hand side of the rule is delayed until the time of its application.

5.2 Expanding matrix products

Starting at **Version 5** the operators `**` and `inv` apply also to matrices. However, in order for `**` and `inv` to continue to work as full fledged operators, the result of multiplications or inverses of matrices is

[2] By the way, I find that behavior of Mathematica's `Module` questionable, since something like

```
F[exp_] := Module[{aa, bb},
  SetNonCommutative[aa, bb];
  aa**bb
]
```

would not fail to treat `aa` and `bb` locally. It is their appearance in a rule that triggers the mostly odd behavior.

held unevaluated until the user calls NCMatrixExpand. This is in the the same spirit as good old fashion commutative operations in Mathematica.

For example, with

```
m1 = {{a, b}, {c, d}}
m2 = {{d, 2}, {e, 3}}
```

the call

```
m1**m2
```

results in

```
{{a, b}, {c, d}}**{{d, 2}, {e, 3}}
```

Upon calling

```
m1**m2 // NCMatrixExpand
```

evaluation takes place returning

```
{{a**d + b**e, 2a + 3b}, {c**d + d**e, 2c + 3d}}
```

which is what would have arisen from calling NCDot[m1,m2][3]. Likewise

```
inv[m1]
```

results in

```
inv[{{a, b}, {c, d}}]
```

and

```
inv[m1] // NCMatrixExpand
```

returns the evaluated result

```
{{inv[a]**(1 + b**inv[d - c**inv[a]**b]**c**inv[a]), -inv[a]**b**inv[d - c**inv[a]**b]},
 {-inv[d - c**inv[a]**b]**c**inv[a], inv[d - c**inv[a]**b]}}
```

or, using MatrixForm:

$$\begin{bmatrix} a^{-1}(1+b(d-ca^{-1}b)^{-1}ca^{-1}) & -a^{-1}b(d-ca^{-1}b)^{-1} \\ -(d-ca^{-1}b)^{-1}ca^{-1} & (d-ca^{-1}b)^{-1} \end{bmatrix}$$

A less trivial example is

```
m3 = m1**inv[IdentityMatrix[2] + m1] - inv[IdentityMatrix[2] + m1]**m1
```

that returns

```
-inv[{{1 + a, b}, {c, 1 + d}}]**{{a, b}, {c, d}} +
    {{a, b}, {c, d}}**inv[{{1 + a, b}, {c, 1 + d}}]
```

Expanding

```
NCMatrixExpand[m3]
```

results in

```
{{b**inv[b - (1 + a)**inv[c]**(1 + d)] - inv[c]**(1 + (1 + d)**inv[b -
    (1 + a)**inv[c]**(1 + d)]**(1 + a)**inv[c])**c - a**inv[c]**(1 + d)**inv[b -
    (1 + a)**inv[c]**(1 + d)] + inv[c]**(1 + d)**inv[b - (1 + a)**inv[c]**(1 + d)]**a,
  a**inv[c]**(1 + (1 + d)**inv[b - (1 + a)**inv[c]**(1 + d)]**(1 + a)**inv[c]) -
    inv[c]**(1 + (1 + d)**inv[b - (1 + a)**inv[c]**(1 + d)]**(1 + a)**inv[c])**d -
```

[3]Formerly MatMult[m1,m2].

```
    b**inv[b - (1 + a)**inv[c]**(1 + d)]**(1 + a)**inv[c] + inv[c]**(1 + d)**inv[b -
    (1 + a)**inv[c]**(1 + d)]** b},
 {d**inv[b - (1 + a)**inv[c]**(1 + d)] - (1 + d)**inv[b - (1 + a)**inv[c]**(1 + d)] -
    inv[b - (1 + a)**inv[c]**(1 + d)]**a + inv[b - (1 + a)**inv[c]**(1 + d)]**(1 + a),
  1 - inv[b - (1 + a)**inv[c]**(1 + d)]**b - d**inv[b - (1 + a)**inv[c]**(1 + d)]**(1 +
    a)**inv[c] + (1 + d)**inv[b - (1 + a)**inv[c]**(1 + d)]**(1 + a)**inv[c] +
    inv[b - (1 + a)**inv[c]**(1 + d)]**(1 + a)**inv[c]**d}}
```

and finally

```
NCMatrixExpand[m3] // NCSimplifyRational
```

returns

```
{{0, 0}, {0, 0}}
```

as expected.

5.2.1 Trouble with `Plus` (`+`) and matrices

Mathematica's choice of treating lists and matrix indistinctively can cause much trouble when mixing `**` with `Plus` (`+`) operator. The reason for that is that `Plus` is `Listable`, and an expression like

```
z + m1
```

where `m1` is an array and `z` is not, is automatically expanded into

```
{{a + z, b + z}, {c + z, d + z}}
```

or, using `MatrixForm`:

$$\begin{bmatrix} a+z & b+z \\ c+z & d+z \end{bmatrix}$$

Because of this "feature", the expression

```
m1**m2 + m2 // NCMatrixExpand
```

evaluates to the "wrong" result

```
{{{{d + a**d + b**e, 2 a + 3 b + d}, {d + c**d + d**e, 2 c + 4 d}},
 {{2 + a**d + b**e, 2 + 2 a + 3 b}, {2 + c**d + d**e, 2 + 2 c + 3 d}}},
 {{{e + a**d + b**e, 2 a + 3 b + e}, {e + c**d + d**e, 2 c + 3 d + e}},
 {{3 + a**d + b**e, 3 + 2 a + 3 b}, {3 + c**d + d**e, 3 + 2 c + 3 d}}}}
```

which is different than the "correct" result

```
{{d + a**d + b**e, 2 + 2 a + 3 b},
 {e + c**d + d**e, 3 + 2 c + 3 d}}
```

which is returned by either

```
NCMatrixExpand[m1**m2] + m2
```

or

```
NCDot[m1, m2] + m2
```

The reason for the behavior is that `m1**m2` is essentially treated as a *scalar* (it does not have *head* `List`) and therefore gets added entrywise to `m2` *before* `NCMatrixExpand` has a chance to evaluate the `**` product.

There are no easy fixes for this problem, which affects not only `NCAlgebra` but any similar type of matrix product evaluation in Mathematica. With `NCAlgebra`, a better option is to use `NCMatrixReplaceAll` or `NCMatrixReplaceRepeated`. As seen in the section Replace with matrices, the command

```
NCMatrixReplaceAll[x**y + y, {x -> m1, y -> m2}]
```

produces the expected result:

```
{{d + a**d + b**e, 2 + 2 a + 3 b},
 {e + c**d + d**e, 3 + 2 c + 3 d}}
```

Note that the above behavior might be erratic, since in an expression like

```
m1**m2 + m2**m1
```

in which `**` is kept unevaluated as in

```
{{a, b}, {c, d}}**{{d, 2}, {e, 3}} + {{d, 2}, {e, 3}}**{{a, b}, {c, d}}
```

the command

```
m1**m2 + m2**m1 // NCMatrixExpand
```

expands to the expected result

```
{{2 c + a**d + b**e + d**a, 2 a + 3 b + 2 d + d**b},
 {3 c + c**d + d**e + e**a, 2 c + 6 d + e**b}}
```

5.3 Polynomials with commutative coefficients

The package `NCPoly` provides an efficient structure for storing and operating with noncommutative polynomials with commutative coefficients. There are two main goals:

1. *Ordering*: to be able to sort polynomials based on an *ordering* specified by the user. See the chapter Noncommutative Gröbner Basis for more details.
2. *Efficiency*: to efficiently perform polynomial algebra with as little overhead as possible.

Those two properties allow for an efficient implementation of `NCAlgebra`'s noncommutative Gröbner basis algorithm, new in **Version 5**, without the use of auxiliary accelerating `C` code, as in `NCGB`. See Noncommutative Gröbner Basis.

Before getting into details, to see how much more efficient `NCPoly` is when compared with standard `NCAlgebra` objects try

```
Table[Timing[NCExpand[(1 + x)^i]][[1]], {i, 0, 20, 5}]
```

which would typically return something like

```
{0.000088, 0.001074, 0.017322, 0.240704, 3.61492, 52.0254}
```

whereas the equivalent

```
<< NCPoly`
Table[Timing[(1 + NCPolyMonomial[{x}, {x}])^i][[1]], {i, 0, 20, 5}]
```

would return

```
{0.00097, 0.001653, 0.002208, 0.003908, 0.004306, 0.005049}
```

Beware that `NCPoly` objects have limited functionality and should still be considered experimental at this point.

The best way to work with `NCPoly` in `NCAlgebra` is by loading the package `NCPolyInterface`:

```
<< NCPolyInterface`
```

which provides the commands `NCToNCPoly` and `NCPolyToNC` to convert nc expressions back and forth between `NCAlgebra` and `NCPoly`.

For example

```
vars = {x, y, z};
p = NCToNCPoly[1 + x**x - 2 x**y**z, vars]
```

converts the polynomial `1 + x**x - 2 x**y**z` from the standard `NCAlgebra` format into an `NCPoly` object. The reason for the braces in the definition of `vars` will be explained below, when we introduce *ordering*. See also Section Noncommutative Gröbner Basis. The result in this case is the `NCPoly` object

```
NCPoly[{1, 1, 1}, <|{0, 0, 0, 0} -> 1, {0, 0, 2, 0} -> 1, {1, 1, 1, 5} -> -2|>]
```

Conversely the command `NCPolyToNC` converts an `NCPoly` back into `NCAlgebra` format. For example

```
NCPolyToNC[p, vars]
```

returns

```
1 + x**x - 2 x**y**z
```

as expected. Note that an `NCPoly` object does not store symbols, but rather a representation of the polynomial based on specially encoded monomials. This is the reason why one should provide `vars` as an argument to `NCPolyToNC`.

Alternatively, one could construct the same `NCPoly` object by calling `NCPoly` directly as in

```
NCPoly[{1, 1, -2}, {{}, {x, x}, {x, y, z}}, vars]
```

In this syntax the first argument is a list of *coefficients*, the second argument is a list of *monomials*, and the third is the list of *variables*. *Monomials* are given as lists, with `{}` being equivalent to a constant `1`.

The particular coefficients in the `NCPoly` object depend not only on the polynomial being represented but also on the *ordering* implied by the sequence of symbols in the list of variables `vars`. For example:

```
vars = {{x}, {y, z}};
p = NCToNCPoly[1 + x**x - 2 x**y**z, vars]
```

produces:

```
NCPoly[{1, 2}, <|{0, 0, 0} -> 1, {0, 2, 0} -> 1, {2, 1, 5} -> -2|>
```

The sequence of braces in the list of *variables* encodes the *ordering* to be used for sorting `NCPoly`s. Orderings specify how monomials should be ordered, and is discussed in detail in Noncommutative Gröbner Basis. We provide the convenience command `NCPolyDisplayOrder` that prints the polynomial ordering implied by a list of symbols. For example

```
NCPolyDisplayOrder[{x,y,z}]
```

prints out

$x \ll y \ll z$

and

```
NCPolyDisplayOrder[{{x},{y,z}}]
```

prints out

$x \ll y < z$

from where you can see that grouping variables inside braces induces a graded type ordering, as discussed in Noncommutative Gröbner Basis. `NCPoly`s constructed from different orderings cannot be combined.

There is also a special constructor for monomials. For example

```
NCPolyMonomial[{y,x}, vars]
NCPolyMonomial[{x,y}, vars]
```

return the monomials corresponding to yx and xy.

Operations on `NCPoly` objects result in another `NCPoly` object that is always expanded. For example:

```
vars = {{x}, {y, z}};
1 + NCPolyMonomial[{x, y}, vars] - 2 NCPolyMonomial[{y, x}, vars]
```

returns

```
NCPoly[{1, 2}, <|{0, 0, 0} -> 1, {1, 1, 1} -> 1, {1, 1, 3} -> -2|>]
```

and

```
p = (1 + NCPolyMonomial[{x}, vars]**NCPolyMonomial[{y}, vars])^2
```

returns

```
NCPoly[{1, 2}, <|{0, 0, 0} -> 1, {1, 1, 1} -> 2, {2, 2, 10} -> 1|>]
```

Another convenience function is `NCPolyDisplay` which returns a list with the monomials appearing in an `NCPoly` object. For example:

```
NCPolyDisplay[p, vars]
```

returns

```
{x.y.x.y, 2 x.y, 1}
```

The reason for displaying an `NCPoly` object as a list is so that the monomials can appear in the same order as they are stored. Using `Plus` would revert to Mathematica's default ordering. For example

```
p = NCToNCPoly[1 + x**x**x - 2 x**x + z, vars]
NCPolyDisplay[p, vars]
```

returns

```
{z, x.x.x, -2 x.x, 1}
```

whereas

```
NCPolyToNC[p, vars]
```

would return

```
1 + z - 2 x**x + x**x**x
```

in which the sorting of the monomials has been destroyed by `Plus`.

The monomials appear sorted in decreasing order from left to right, with `z` being the *leading term* in the above example.

With `NCPoly` the Mathematica command `Sort` is modified to sort lists of polynomials. For example

```
polys = NCToNCPoly[{x**x**x, 2 y**x - z, z, y**x - x**x}, vars]
ColumnForm[NCPolyDisplay[Sort[polys], vars]]
```

returns

```
{x.x.x}
{z}
{y.x, -x.x}
{2 y.x, -z}
```

`Sort` produces a list of polynomials sorted in *ascending* order based on their *leading terms*.

5.4 Polynomials with noncommutative coefficients

A larger class of polynomials in noncommutative variables is that of polynomials with noncommutative coefficients. Think of a polynomial with commutative coefficients in which certain variables are considered to be unknown, i.e. *variables*, where others are considered to be known, i.e. *coefficients*. For example, in many problems in systems and control the following expression

$p(x) = ax + xa^T - xbx + c$

is often seen as a polynomial in the noncommutative unknown `x` with known noncommutative coefficients `a`, `b`, and `c`. A typical problem is the determination of a solution to the equation $p(x) = 0$ or the inequality $p(x) \succeq 0$.

The package `NCPolynomial` handles such polynomials with noncommutative coefficients. As with `NCPoly`, the package provides the commands `NCToNCPolynomial` and `NCPolynomialToNC` to convert nc expressions back and forth between `NCAlgebra` and `NCPolynomial`. For example

```
vars = {x}
p = NCToNCPolynomial[a**x + x**tp[a] - x**b**x + c, vars]
```

converts the polynomial `a**x + x**tp[a] - x**b**x + c` from the standard `NCAlgebra` format into an `NCPolynomial` object. The result in this case is the `NCPolynomial` object

```
NCPolynomial[c, <|{x} -> {{1, a, 1}, {1, 1, tp[a]}}, {x, x} -> {{-1, 1, b, 1}}|>, {x}]
```

Conversely the command `NCPolynomialToNC` converts an `NCPolynomial` back into `NCAlgebra` format. For example

```
NCPolynomialToNC[p]
```

returns

```
c + a**x + x**tp[a] - x**b**x
```

An `NCPolynomial` does store information about the polynomial symbols and a list of variables is required only at the time of creation of the `NCPolynomial` object.

As with `NCPoly`, operations on `NCPolynomial` objects result on another `NCPolynomial` object that is always expanded. For example:

```
vars = {x,y}
1 + NCToNCPolynomial[x**y, vars] - 2 NCToNCPolynomial[y**x, vars]
```

returns

```
NCPolynomial[1, <|{y**x} -> {{-2, 1, 1}}, {x**y} -> {{1, 1, 1}}|>, {x, y}]
```

and

```
(1 + NCToNCPolynomial[x, vars]**NCToNCPolynomial[y, vars])^2
```

returns

```
NCPolynomial[1, <|{x**y**x**y} -> {{1, 1, 1}}, {x**y} -> {{2, 1, 1}}|>, {x, y}]
```

To see how much more efficient `NCPolynomial` is when compared with standard `NCAlgebra` objects try

```
Table[Timing[(NCToNCPolynomial[x, vars])^i][[1]], {i, 0, 20, 5}]
```

would return

```
{0.000493, 0.003345, 0.005974, 0.013479, 0.018575, 0.02896}
```

As you can see, `NCPolynomials` are not as efficient as `NCPolys` but still much more efficient than `NCAlgebra` polynomials.

`NCPolynomials` do not support *orderings* but we do provide the `NCPSort` command that produces a list of terms sorted by degree. For example

```
NCPSort[p]
```

returns

```
{c, a**x, x**tp[a], -x**b**x}
```

A useful feature of `NCPolynomial` is the capability of handling polynomial matrices. For example

```
mat1 = {{a**x + x**tp[a] + c**y + tp[y]**tp[c] - x**q**x, b**x},
        {x**tp[b], 1}};
p1 = NCToNCPolynomial[mat1, {x, y}];

mat2 = {{1, x**tp[c]}, {c**x, 1}};
p2 = NCToNCPolynomial[mat2, {x, y}];
```

constructs `NCPolynomial` objects representing the polynomial matrices `mat1` and `mat2`. Verify that

```
NCPolynomialToNC[p1**p2] - NCDot[mat1, mat2] // NCExpand
```

is zero as expected. Internally `NCPolynomial` represents a polynomial matrix by constructing matrix factors. For example the representation of the matrix `mat1` correspond to the factors

$$\begin{bmatrix} ax + xa^T + cy + y^T c^T - xqx & bx \\ xb^T & 1 \end{bmatrix} = \begin{bmatrix} 0 & 0 \\ 0 & 1 \end{bmatrix} + \begin{bmatrix} a \\ 0 \end{bmatrix} x \begin{bmatrix} 1 & 0 \end{bmatrix} + \begin{bmatrix} 1 \\ 0 \end{bmatrix} x \begin{bmatrix} a^T & 0 \end{bmatrix} + \begin{bmatrix} -1 \\ 0 \end{bmatrix} xqx \begin{bmatrix} 1 & 0 \end{bmatrix} + \begin{bmatrix} b \\ 0 \end{bmatrix} x \begin{bmatrix} 0 & 1 \end{bmatrix} + \begin{bmatrix} 0 \\ 1 \end{bmatrix} x \begin{bmatrix} b^T & 0 \end{bmatrix} + \begin{bmatrix} c \\ 0 \end{bmatrix} y \begin{bmatrix} 1 & 0 \end{bmatrix} + \begin{bmatrix} 1 \\ 0 \end{bmatrix} y^T \begin{bmatrix} c^T & 0 \end{bmatrix}$$

See section linear polynomials for more features on linear polynomial matrices.

5.5 Linear polynomials

Another interesting class of nc polynomials is that of linear polynomials, which can be factored in the form:

$$s(x) = l(F \otimes x)r$$

where l and r are vectors with symbolic expressions and F is a numeric matrix. This functionality is in the package

```
<< NCSylvester`
```

Use the command `NCToNCSylvester` to factor a linear nc polynomial into the the above form. For example:

```
vars = {x, y};
expr = 1 + a**x + x**tp[a] - x + b**y**d + tp[d]**tp[y]**tp[b];
{const, lin} = NCToNCSylvester[expr, vars];
```

which returns

```
const = 1
```

and an `Association` `lin` containing the factorization. For example

```
lin[x]
```

returns a list with the left and right vectors `l` and `r` and the coefficient array `F`.

```
{{1, a}, {1, a^T}, SparseArray[< 2 >, {2, 2}]}
```

which in this case is the matrix:

$$\begin{bmatrix} -1 & 1 \\ 1 & 0 \end{bmatrix}$$

and

```
lin[tp[y]]
```

returns

```
{{d^T}, {b^T}, SparseArray[< 1 >, {1, 1}]}
```

Note that transposes and adjoints are treated as independent variables.

Perhaps the most useful consequence of the above factorization is the possibility of producing a linear polynomial which has the smallest possible number of terms, as explaining in detail in [de 12]. This is done automatically by `NCSylvesterToNC`. For example

```
vars = {x, y};
expr = a**x**c - a**x**d - a**y**c + a**y**d + b**x**c - b**x**d - b**y**c + b**y**d;
{const, lin} = NCToNCSylvester[expr, vars];
NCSylvesterToNC[{const, lin}]
```

produces:

```
(a + b) ** x ** (c - d) + (a + b) ** y ** (-c + d)
```

This factorization even works with linear matrix polynomials, and is used by the our semidefinite programming algorithm (see Chapter Semidefinite Programming) to factor linear matrix inequalities in the least possible number of terms. For example:

```
vars = {x};
expr = {{a ** x + x ** tp[a], b ** x, tp[c]},
        {x ** tp[b], -1, tp[d]},
        {c, d, -1}};
{const, lin} = NCToNCSylvester[expr, vars]
```

result in:

```
const = SparseArray[< 6 >, {3, 3}]
lin = <|x -> {{1, a, b}, {1, tp[a], tp[b]}, SparseArray[< 4 >, {9, 9}]}|>
```

See [de 12] for details on the structure of the constant array F in this case.

Chapter 6

Noncommutative Gröbner Basis

The package `NCGBX` provides an implementation of a noncommutative Gröbner Basis algorithm. It is a Mathematica only replacement to the C++ `NCGB` which is still provided with this distribution.

If you want a living version of this chapter just run the notebook `NC/DEMOS/3_NCGroebnerBasis.nb`.

Gröbner Basis are useful in the study of algebraic relations.

In order to load `NCGBX` one types:

```
<< NC`
<< NCAlgebra`
<< NCGBX`
```

or simply

```
<< NCGBX`
```

if `NC` and `NCAlgebra` have already been loaded.

6.1 What is a Gröbner Basis?

Most commutative algebra packages contain commands based on Gröbner Basis and uses of Gröbner Basis. For example, in Mathematica, the `Solve` command puts collections of equations in a *canonical form* which, for simple collections, readily yields a solution. Likewise, the Mathematica `Eliminate` command tries to convert a collection of m polynomial equations (often called relations)

$$\begin{aligned} p_1(x_1,\ldots,x_n) &= 0 \\ p_2(x_1,\ldots,x_n) &= 0 \\ \vdots \quad & \quad \vdots \\ p_m(x_1,\ldots,x_n) &= 0 \end{aligned}$$

in variables $x_1, x_2, \ldots x_n$ to a *triangular* form, that is a new collection of equations like

$$\begin{aligned} q_1(x_1) &= 0 \\ q_2(x_1, x_2) &= 0 \\ q_3(x_1, x_2) &= 0 \\ q_4(x_1, x_2, x_3) &= 0 \\ \vdots \quad & \quad \vdots \\ q_r(x_1, \ldots, x_n) &= 0. \end{aligned}$$

Here the polynomials $\{q_j : 1 \le j \le k_2\}$ generate the same *ideal* that the polynomials $\{p_j : 1 \le j \le k_1\}$ generate. Therefore, the set of solutions to the collection of polynomial equations $\{p_j = 0 : 1 \le j \le k_1\}$ equals the set of solutions to the collection of polynomial equations $\{q_j = 0 : 1 \le j \le k_2\}$. This canonical form greatly simplifies the task of solving collections of polynomial equations by facilitating backsolving for x_j in terms of $x_1, \ldots, x_{j-1}$.

Readers who would like to know more about Gröbner Basis may want to read [CLS]. The noncommutatative version of the algorithm implemented by `NCGB` is loosely based on [Mora].

6.2 Solving equations

Before calculating a Gröbner Basis, one must declare which variables will be used during the computation and must declare a *monomial order* which can be done using `SetMonomialOrder` as in:

```
SetMonomialOrder[{a, b, c}, x];
```

The monomial ordering imposes a relationship between the variables which are used to *sort* the monomials in a polynomial. The ordering implied by the above command can be visualized using:

```
PrintMonomialOrder[];
```

which in this case prints:

$$a < b < c \ll x.$$

A user does not need to know theoretical background related to monomials orders. Indeed, as we shall see soon, in many engineering problems, it suffices to know which variables correspond to quantities which are *known* and which variables correspond to quantities which are *unknown*. If one is solving for a variable or desires to prove that a certain quantity is zero, then one would want to view that variable as *unknown*. In the above example, the symbol '$\ll$' separate the *knowns*, a, b, c, from the *unknown*, x. For more details on orderings see Section Orderings.

Our goal is to calculate the Gröbner basis associated with the following relations (i.e. a list of polynomials):

$$a\,x\,a = c, \quad a\,b = 1, \quad b\,a = 1.$$

We shall use the word *relation* to mean a polynomial in noncommuting indeterminates. For example, if an analyst saw the equation $AB = 1$ for matrices A and B, then he might say that A and B satisfy the polynomial equation $a\,b - 1 = 0$. An algebraist would say that $a\,b - 1$ is a relation.

To calculate a Gröbner basis one defines a list of relations:

```
rels = {a ** x ** a - c, a ** b - 1, b ** a - 1}
```

and issues the command:

```
gb = NCMakeGB[rels, 10]
```

which should produces an output similar to:

```
* * * * * * * * * * * * * * * *
* * *   NCPolyGroebner    * * *
```

```
* * * * * * * * * * * * * * * *
* Monomial order: a < b < c << x
* Reduce and normalize initial set
> Initial set could not be reduced
* Computing initial set of obstructions
> MAJOR Iteration 1, 4 polys in the basis, 2 obstructions
> MAJOR Iteration 2, 5 polys in the basis, 2 obstructions
* Cleaning up...
* Found Groebner basis with 3 polynomials
* * * * * * * * * * * * * * * *
```

The number `10` in the call to `NCMakeGB` is very important because a finite GB may not exist. It instructs `NCMakeGB` to abort after `10` iterations if a GB has not been found at that point.

The result of the above calculation is the list of relations in the form of a list of rules:

```
{x -> b ** c ** b, a ** b -> 1, b ** a -> 1}
```

Version 5: For efficiency, `NCMakeGB` returns a list of rules instead of a list of polynomials. The left-hand side of the rule is the leading monomial in the current order. This is incompatible with early versions, which returned a list of polynomials. You can recover the old behavior setting the option `ReturnRules -> False`. This can be done in the `NCMakeGB` command or globally through `SetOptions[ReturnRules -> False]`.

Our favorite format for displaying lists of relations is `ColumnForm`.

```
ColumnForm[gb]
```

which results in

```
x -> b ** c ** b
a ** b -> 1
b ** a -> 1
```

The *rules* in the output represent the relations in the GB with the left-hand side of the rule being the leading monomial. Replacing `Rule` by `Subtract` recovers the relations but one would then loose the leading monomial as Mathematica alphabetizes the resulting sum.

Someone not familiar with GB's might find it instructive to note this output GB effectively *solves* the input equation

$$a\,x\,a - c = 0$$

under the assumptions that

$$b\,a - 1 = 0, \quad a\,b - 1 = 0,$$

that is $a = b^{-1}$ and produces the expected result in the form of the relation:

$$x = b\,c\,b.$$

6.3 A slightly more challenging example

For a slightly more challenging example consider the same monomial order as before:

```
SetMonomialOrder[{a, b, c}, x];
```

that is

$a < b < c \ll x$

and the relations:

$$\begin{aligned} a\,x - c &= 0, \\ a\,b\,a - a &= 0, \\ b\,a\,b - b &= 0, \end{aligned}$$

from which one can recognize the problem of solving the linear equation $a\,x = c$ in terms of the *pseudo-inverse* $b = a^\dagger$. The calculation:

```
gb = NCMakeGB[{a ** x - c, a ** b ** a - a, b ** a ** b - b}, 10];
```

finds the Gröbner basis:

```
a ** x -> c
a ** b ** c -> c
a ** b ** a -> a
b ** a ** b -> b
```

In this case the Gröbner basis cannot quite *solve* the equations but it remarkably produces the necessary condition for existence of solutions:

$$0 = a\,b\,c - c = a\,a^\dagger c - c$$

that can be interpreted as c being in the range-space of a.

6.4 Simplifying polynomial expresions

Our goal now is to verify if it is possible to *simplify* the following expression:

$$b\,b\,a\,a - a\,a\,b\,b + a\,b\,a$$

if we know that

$$a\,b\,a = b$$

using Gröbner basis. With that in mind we set the order:

```
SetMonomialOrder[a,b];
```

and calculate the GB associated with the constraint:

```
rels = {a ** b ** a - b};
rules = NCMakeGB[rels, 10];
```

which produces the output

```
* * * * * * * * * * * * * * * *
* * *   NCPolyGroebner    * * *
* * * * * * * * * * * * * * * *
* Monomial order: a << b
* Reduce and normalize initial set
> Initial set could not be reduced
* Computing initial set of obstructions
> MAJOR Iteration 1, 2 polys in the basis, 1 obstructions
* Cleaning up...
* Found Groebner basis with 2 polynomials
* * * * * * * * * * * * * * * *
```

and the associated GB

```
a ** b ** a -> b
b ** b ** a -> a ** b ** b
```

The GB revealed another relationship that must hold true if $a\,b\,a = b$. One can use these relationships to simplify the original expression using `NCReplaceRepeated` as in

```
expr = b ** b ** a ** a - a ** a ** b ** b + a ** b ** a
simp = NCReplaceRepeated[expr, rules]
```

which results in

```
simp = b
```

6.5 Minimal versus reduced Gröbner Basis

The algorithm implemented by `NCGB` always produces a Gröbner Basis with the *minimal* possible number of polynomials. However, such polynomials are not necessarily the "simplest" possible polynomials; called the *reduced* Gröbner Basis. The *reduced* Gröbner Basis is unique given the relations and the monomial order. Consider for example the following monimial order

```
SetMonomialOrder[x, y]
```

and the relations

```
rels = {x^3 - 2 x ** y, x^2 ** y - 2 y^2 + x}
```

for which

```
NCMakeGB[rels]
```

produces the following *minimal* Gröbner Basis

```
x**x->0
x**y->x**x**x/2
y**x->x**y
y**y->x/2+x**x**y/2
```

but

```
NCMakeGB[rels, ReduceBasis -> True]
```

returns the *reduced* Gröbner Basis

```
x**x->0
x**y->0
y**x->0
y**y->x/2
```

in which not only the leading mononials but also all lower-order monomials have been reduced by the basis' leading monomials.

6.6 Simplifying rational expresions

It is often desirable to simplify expressions involving inverses of noncommutative expressions. One challenge is to recognize identities implied by the existence of certain inverses. For example, that the expression

$$x(1-x)^{-1} - (1-x)^{-1}x$$

is equivalent to 0. One can use a nc Gröbner basis for that task. Consider for instance the order

$$x \ll (1-x)^{-1}$$

implied by the command:

```
SetMonomialOrder[x, inv[1-x]]
```

This ordering encodes the following precise idea of what we mean by *simple* versus *complicated*: it formally corresponds to specifying that x is simpler than $(1-x)^{-1}$, which might sits well with one's intuition.

Now consider the following command:

```
rules = NCMakeGB[{}, 3]
```

which produces the output

```
* * * * * * * * * * * * * * * *
* * *   NCPolyGroebner    * * *
* * * * * * * * * * * * * * * *
* Monomial order: x <<  inv[x] << inv[1 - x]
* Reduce and normalize initial set
> Initial set could not be reduced
* Computing initial set of obstructions
> MAJOR Iteration 1, 6 polys in the basis, 6 obstructions
* Cleaning up...
* Found Groebner basis with 6 polynomials
* * * * * * * * * * * * * * * *
```

and results in the rules:

```
x ** inv[1 - x] -> -1 + inv[1 - x],
inv[1-x] ** x -> -1 + inv[1-x],
```

As in the previous example, the GB revealed new relationships that must hold true if $1-x$ is invertible, and one can use this relationship to *simplify* the original expression using `NCReplaceRepeated` as in:

```
NCReplaceRepeated[x ** inv[1 - x] - inv[1 - x] ** x, rules]
```

The above command results in `0`, as one would hope.

For a more challenging example consider the identity:

$$\left(1-x-y(1-x)^{-1}y\right)^{-1}=\frac{1}{2}(1-x-y)^{-1}+\frac{1}{2}(1-x+y)^{-1}$$

One can verify that the rule based command NCSimplifyRational fails to simplify the expression:

```
expr = inv[1 - x - y ** inv[1 - x] ** y] - 1/2 (inv[1 - x + y] + inv[1 - x - y])
NCSimplifyRational[expr]
```

We set the monomial order and calculate the Gröbner basis

```
SetMonomialOrder[x, y, inv[1-x], inv[1-x+y], inv[1-x-y], inv[1-x-y**inv[1-x]**y]];
rules = NCMakeGB[{}, 3];
```

based on the rational involved in the original expression. The result is the nc GB:

```
inv[1-x-y**inv[1-x]**y] -> (1/2)inv[1-x-y]+(1/2)inv[1-x+y]
x**inv[1-x] -> -1+inv[1-x]
y**inv[1-x+y] -> 1-inv[1-x+y]+x**inv[1-x+y]
y**inv[1-x-y] -> -1+inv[1-x-y]-x**inv[1-x-y]
inv[1-x]**x -> -1+inv[1-x]
inv[1-x+y]**y -> 1-inv[1-x+y]+inv[1-x+y]**x
inv[1-x-y]**y -> -1+inv[1-x-y]-inv[1-x-y]**x
inv[1-x+y]**x**inv[1-x-y] -> -(1/2)inv[1-x-y]-(1/2)inv[1-x+y]+inv[1-x+y]**inv[1-x-y]
inv[1-x-y]**x**inv[1-x+y] -> -(1/2)inv[1-x-y]-(1/2)inv[1-x+y]+inv[1-x-y]**inv[1-x+y]
```

which succesfully simplifyes the original expression using:

```
expr = inv[1 - x - y ** inv[1 - x] ** y] - 1/2 (inv[1 - x + y] + inv[1 - x - y])
NCReplaceRepeated[expr, rules] // NCExpand
```

resulting in `0`.

6.7 Simplification with NCGBSimplifyRational

The simplification process described above is automated in the function NCGBSimplifyRational.

For example, calls to

```
expr = x ** inv[1 - x] - inv[1 - x] ** x
NCGBSimplifyRational[expr]
```

or

```
expr = inv[1 - x - y ** inv[1 - x] ** y] - 1/2 (inv[1 - x + y] + inv[1 - x - y])
NCGBSimplifyRational[expr]
```

both result in `0`.

6.8 Ordering on variables and monomials

As seen above, one needs to declare a *monomial order* before making a Gröbner Basis. There are various monomial orders which can be used when computing Gröbner Basis. The most common are *lexicographic* and *graded lexicographic* orders. We consider also *multi-graded lexicographic* orders.

Lexicographic and multi-graded lexicographic orders are examples of elimination orderings. An elimination ordering is an ordering which is used for solving for some of the variables in terms of others.

We now discuss each of these types of orders.

6.8.1 Lex Order: the simplest elimination order

To impose lexicographic order, say $a \ll b \ll x \ll y$ on a, b, x and y, one types

```
SetMonomialOrder[a,b,x,y];
```

or, equivalently

```
SetMonomialOrder[{a},{b},{x},{y}];
```

This order is useful for attempting to solve for y in terms of a, b and x, since the highest priority of the GB algorithm is to produce polynomials which do not contain y. If producing high order polynomials is a consequence of this fanaticism so be it. Unlike graded orders, lex orders pay little attention to the degree of terms. Likewise its second highest priority is to eliminate x.

Once this order is set, one can use all of the commands in the preceeding section in exactly the same form.

We now give a simple example how one can solve for y given that a,b,x and y satisfy the equations:

$$\begin{aligned} -b\,x + x\,y\,a + x\,b\,a\,a &= 0 \\ x\,a - 1 &= 0 \\ a\,x - 1 &= 0 \end{aligned}$$

The command

```
NCMakeGB[{-b**x+x**y**a+x**b**a**a, x**a-1, a**x-1},4]
```

produces the Gröbner basis:

```
y -> -b**a + a**b**x**x
a**x -> 1
x**a -> 1
```

after one iteration.

Now, we change the order to

```
SetMonomialOrder[y,x,b,a];
```

and run the same `NCMakeGB` as above:

```
NCMakeGB[{-b**x+x**y**a+x**b**a**a, x**a-1, a**x-1},4]
```

which, this time, results in

```
x**a -> 1
a**x -> 1
x**b**a -> -x**y+b**x**x
b**a**a -> -y**a+a**b**x
x**b**b**a -> -x**b**y-x**y**b**x**x+b**x**x**b**x**x
b**x**x**x -> x**b+x**y**x
b**a**b**a -> -y**y-b**a**y-y**b**a+a**b**x**b**x**x
a**b**x**x -> y+b**a
b**a**b**b**a -> -y**b**y-b**a**b**y-y**b**b**a-y**y**b**x**x-
                 b**a**y**b**x**x+a**b**x**b**x**x**b**x**x
```

which is not a Gröbner basis since the algorithm was interrupted at 4 iterations. Note the presence of the rule

```
a**b**x**x -> y+b**a
```

which shows that the order is not set up to solve for y in terms of the other variables in the sense that y is not on the left hand side of this rule (but a human could easily solve for y using this rule). Also the algorithm created a number of other relations which involved y.

6.8.2 Graded lex ordering: a non-elimination order

To impose graded lexicographic order, say $a < b < x < y$ on a, b, x and y, one types

```
SetMonomialOrder[{a,b,x,y}];
```

This ordering puts high degree monomials high in the order. Thus it tries to decrease the total degree of expressions. A call to

```
NCMakeGB[{-b**x+x**y**a+x**b**a**a, x**a-1, a**x-1},4]
```

now produces

```
a**x -> 1
x**a -> 1
b**a**a -> -y**a+a**b**x
x**b**a -> -x**y+b**x**x
a**b**x**x -> y+b**a
b**x**x**x -> x**b+x**y**x
a**b**x**b**x**x -> y**y+b**a**y+y**b**a+b**a**b**a
b**x**x**b**x**x -> x**b**y+x**b**b**a+x**y**b**x**x
a**b**x**b**x**b**x**x -> y**y**y+b**a**y**y+y**b**a**y+y**y**b**a+
                          b**a**b**a**y+b**a**y**b**a+y**b**a**b**a+
                          b**a**b**a**b**a
b**x**x**b**x**b**x**x -> x**b**y**y+x**b**b**a**y+x**b**y**b**a+
                          x**b**b**a**b**a+x**y**b**x**b**x**x
```

which again fails to be a Gröbner basis and does not eliminate y. Instead, it tries to decrease the total degree of expressions involving a, b, x, and y.

6.8.3 Multigraded lex ordering: a variety of elimination orders

There are other useful monomial orders which one can use other than graded lex and lex. Another type of order is what we call multigraded lex and is a mixture of graded lex and lex order. To impose multi-graded lexicographic order, say $a < b < x \ll y$ on a, b, x and y, one types

```
SetMonomialOrder[{a,b,x},y];
```

which separates y from the remaining variables. This time, a call to

```
NCMakeGB[{-b**x+x**y**a+x**b**a**a, x**a-1, a**x-1},4]
```

yields once again

```
y -> -b**a+a**b**x**x
a**x -> 1
x**a -> 1
```

which not only eliminates y but is also Gröbner basis, calculated after one iteration.

For an intuitive idea of why multigraded lex is helpful, we think of a, b, and x as corresponding to variables in some engineering problem which represent quantities which are *known* and y to be *unknown*. The fact that a, b and x are in the top level indicates that we are very interested in solving for y in terms of a, b, and x, but are not willing to solve for, say x, in terms of expressions involving y.

This situation is so common that we provide the commands `SetKnowns` and `SetUnknowns`. The above ordering would be obtained after setting

```
SetKnowns[a,b,x];
SetUnknowns[y];
```

6.9 A complete example: the partially prescribed matrix inverse problem

This is a type of problem known as a *matrix completion problem*. This particular one was suggested by Hugo Woerdeman. We are grateful to him for discussions.

Problem: *Given matrices a, b, c, and d, we wish to determine under what conditions there exists matrices x, y, z, and w such that the block matrices*

$$\begin{bmatrix} a & x \\ y & b \end{bmatrix} \qquad \begin{bmatrix} w & c \\ d & z \end{bmatrix}$$

are inverses of each other. Also, we wish to find formulas for x, y, z, and w.

This problem was solved in a paper by W.W. Barrett, C.R. Johnson, M. E. Lundquist and H. Woerderman [BJLW] where they showed it splits into several cases depending upon which of a, b, c and d are invertible. In our example, we assume that a, b, c and d are invertible and discover the result which they obtain in this case.

First we set the matrices a, b, c, and d and their inverses as *knowns* and x, y, w, and z as unknowns:

```
SetKnowns[a, inv[a], b, inv[b], c, inv[c], d, inv[d]];
SetUnknowns[{z}, {x, y, w}];
```

Note that the graded ordedring of the unknowns means that we care more about solving for x, y and w than for z.

Then we define the relations we are interested in, which are obtained after multiplying the two block matrices on both sides and equating to identity

```
A = {{a, x}, {y, b}}
B = {{w, c}, {d, z}}

rels = {
```

```
  MatMult[A, B] - IdentityMatrix[2],
  MatMult[B, A] - IdentityMatrix[2]
} // Flatten
```

We use `Flatten` to reduce the matrix relations to a simple list of relations. The resulting relations in this case are:

```
rels = {-1+a**w+x**d, a**c+x**z, b**d+y**w, -1+b**z+y**c,
        -1+c**y+w**a, c**b+w**x, d**a+z**y, -1+d**x+z**b}
```

After running

```
NCMakeGB[rels, 8]
```

we obtain the Gröbner basis:

```
x -> inv[d]-inv[d]**z**b
y -> inv[c]-b**z**inv[c]
w -> inv[a]**inv[d]**z**b**d
z**b**z -> z+d**a**c
c**b**z**inv[c]**inv[a] -> inv[a]**inv[d]**z**b**d
inv[c]**inv[a]**inv[d]**z**b -> b**z**inv[c]**inv[a]**inv[d]
inv[d]**z**b**d**a -> a**c**b**z**inv[c]
z**b**d**a**c -> d**a**c**b**z
z**inv[c]**inv[a]**inv[d]**inv[b] -> inv[b]**inv[c]**inv[a]**inv[d]**z
z**inv[c]**inv[a]**inv[d]**z -> inv[b]+inv[b]**inv[c]**inv[a]**inv[d]**z
d**a**c**b**z**inv[c] -> z**b**d**a
```

after seven iterations. The first four relations

$$
\begin{aligned}
x &= d^{-1} - d^{-1}\, z\, b \\
y &= c^{-1} - b\, z\, c^{-1} \\
w &= a^{-1}\, d^{-1}\, z\, b\, d \\
z\, b\, z &= z + d\, a\, c
\end{aligned}
$$

are the solutions we are looking for, which states that one can find x, y, z, and w such that the matrices above are inverses of each other if and only if $z\, b\, z = z + d\, a\, c$. The first three relations gives formulas for x, y and w in terms of z.

A variety of scenarios can be quickly investigated under different assumptions. For example, say that c is not invertible. Is it still possible to solve the problem? One solution is obtained with the ordering implied by

```
SetKnowns[a, inv[a], b, inv[b], c, d, inv[d]];
SetUnknowns[{y}, {z, w, x}];
```

In this case

```
NCMakeGB[rels, 8]
```

produces the Gröbner basis:

```
z -> inv[b]-inv[b]**y**c
w -> inv[a]-c**y**inv[a]
x -> a**c**y**inv[a]**inv[d]
y**c**y -> y+b**d**a
c**y**inv[a]**inv[d]**inv[b] -> inv[a]**inv[d]**inv[b]**y**c
d**a**c**y**inv[a] -> inv[b]**y**c**b**d
inv[d]**inv[b]**y**c**b -> a**c**y**inv[a]**inv[d]
y**c**b**d**a -> b**d**a**c**y
y**inv[a]**inv[d]**inv[b]**y**c -> 1+y**inv[a]**inv[d]**inv[b]
```

after five iterations. Once again, the first four relations

$$\begin{aligned} z &= b^{-1} - b^{-1}\, y\, c \\ w &= a^{-1} - c\, y\, a^{-1} \\ x &= a\, c\, y\, a^{-1}\, d^{-1} \\ y\, c\, y &= y + b\, d\, a \end{aligned}$$

provide formulas, this time for z, w, and z in terms of y satisfying $y\, c\, y = y + b\, d\, a$. Note that these formulas do not involve c^{-1} since c is no longer assumed invertible.

Chapter 7

Semidefinite Programming

If you want a living version of this chapter just run the notebook `NC/DEMOS/4_SemidefiniteProgramming.nb`.

There are two different packages for solving semidefinite programs:

- `SDP` provides a template algorithm that can be customized to solve semidefinite programs with special structure. Users can provide their own functions to evaluate the primal and dual constraints and the associated Newton system. A built in solver along conventional lines, working on vector variables, is provided by default. It does not require NCAlgebra to run.
- `NCSDP` coordinates with NCAlgebra to handle matrix variables, allowing constraints, etc, to be entered directly as noncommutative expressions.

7.1 Semidefinite Programs in Matrix Variables

The package NCSDP allows the symbolic manipulation and numeric solution of semidefinite programs.

After loading NCAlgebra, the package NCSDP must be loaded using:

```
<< NCSDP`
```

Semidefinite programs consist of symbolic noncommutative expressions representing inequalities and a list of rules for data replacement. For example the semidefinite program:

$$\begin{aligned} \min_{Y} \quad & < I, Y > \\ \text{s.t.} \quad & AY + YA^T + I \preceq 0 \\ & Y \succeq 0 \end{aligned}$$

can be solved by defining the noncommutative expressions

```
SNC[a, y];
obj = {-1};
ineqs = {a ** y + y ** tp[a] + 1, -y};
```

The inequalities are stored in the list `ineqs` in the form of noncommutative linear polyonomials in the variable `y` and the objective function constains the symbolic coefficients of the inner product, in this case `-1`. The reason for the negative signs in the objective as well as in the second inequality is that semidefinite programs are expected to be cast in the following *canonical form*:

$$\begin{aligned} \max_{y} \quad & < b, y > \\ \text{s.t.} \quad & f(y) \preceq 0 \end{aligned}$$

or, equivalently:

$$\begin{aligned} \max_{y} \quad & < b, y > \\ \text{s.t.} \quad & f(y) + s = 0, \quad s \succeq 0 \end{aligned}$$

Semidefinite programs can be visualized using NCSDPForm as in:

```
vars = {y};
NCSDPForm[ineqs, vars, obj]
```

The above commands produce a formatted output similar to the ones shown above.

In order to obtaining a numerical solution for an instance of the above semidefinite program one must provide a list of rules for data substitution. For example:

```
A = {{0, 1}, {-1, -2}};
data = {a -> A};
```

Equipped with the above list of rules representing a problem instance one can load SDPSylvester and use NCSDP to create a problem instance as follows:

```
{abc, rules} = NCSDP[ineqs, vars, obj, data];
```

The resulting `abc` and `rules` objects are used for calculating the numerical solution using SDPSolve. The command:

```
<< SDPSylvester`
{Y, X, S, flags} = SDPSolve[abc, rules];
```

produces an output like the folowing:

```
Problem data:
* Dimensions (total):
  - Variables             = 4
  - Inequalities          = 2
* Dimensions (detail):
  - Variables             = {{2,2}}
  - Inequalities          = {2,2}
Method:
* Method                  = PredictorCorrector
* Search direction        = NT
Precision:
* Gap tolerance           = 1.*10^(-9)
* Feasibility tolerance   = 1.*10^(-6)
* Rationalize iterates    = False
Other options:
* Debug level             = 0

 K     <B, Y>         mu  theta/tau      alpha     |X S|2    |X S|oo  |A* X-B|   |A Y+S-C|
-------------------------------------------------------------------------------------------
 1  1.638e+00  1.846e-01  2.371e-01  8.299e-01  1.135e+00  9.968e-01  9.868e-16  2.662e-16
 2  1.950e+00  1.971e-02  2.014e-02  8.990e-01  1.512e+00  9.138e-01  2.218e-15  2.937e-16
 3  1.995e+00  1.976e-03  1.980e-03  8.998e-01  1.487e+00  9.091e-01  1.926e-15  3.119e-16
 4  2.000e+00  9.826e-07  9.826e-07  9.995e-01  1.485e+00  9.047e-01  8.581e-15  2.312e-16
 5  2.000e+00  4.913e-10  4.913e-10  9.995e-01  1.485e+00  9.047e-01  1.174e-14  4.786e-16
-------------------------------------------------------------------------------------------
* Primal solution is not strictly feasible but is within tolerance
(0 <= max eig(A* Y - C) = 8.06666*10^-10 < 1.*10^-6 )
* Dual solution is within tolerance
```

```
(|| A X - B || = 1.96528*10^-9 < 1.*10^-6)
* Feasibility radius = 0.999998
(should be less than 1 when feasible)
```

The output variables `Y` and `S` are the *primal* solutions and `X` is the *dual* solution.

A symbolic dual problem can be calculated easily using `NCSDPDual`:

```
{dIneqs, dVars, dObj} = NCSDPDual[ineqs, vars, obj];
```

The dual program for the example problem above is:

$$\begin{aligned} \max_{x} \quad & < c, x > \\ \text{s.t.} \quad & f^*(x) + b = 0, \quad x \succeq 0 \end{aligned}$$

In the case of the above problem the dual program is

$$\begin{aligned} \max_{X_1, X_2} \quad & < I, X_1 > \\ \text{s.t.} \quad & A^T X_1 + X_1 A - X_2 - I = 0 \\ & X_1 \succeq 0, \\ & X_2 \succeq 0 \end{aligned}$$

which can be visualized using `NCSDPDualForm` using:

```
NCSDPDualForm[dIneqs, dVars, dObj]
```

7.2 Semidefinite Programs in Vector Variables

The package SDP provides a crude and not very efficient way to define and solve semidefinite programs in standard form, that is vectorized. You do not need to load `NCAlgebra` if you just want to use the semidefinite program solver. But you still need to load `NC` as in:

```
<< NC`
<< SDP`
```

Semidefinite programs are optimization problems of the form:

$$\begin{aligned} \max_{y, S} \quad & b^T y \\ \text{s.t.} \quad & Ay + S = c \\ & S \succeq 0 \end{aligned}$$

where S is a symmetric positive semidefinite matrix and y is a vector of decision variables.

A user can input the problem data, the triplet (A, b, c), or use the following convenient methods for producing data in the proper format.

For example, problems can be stated as:

$$\begin{aligned} \min_{y} \quad & f(y), \\ \text{s.t.} \quad & G(y) \succeq 0 \end{aligned}$$

where $f(y)$ and $G(y)$ are affine functions of the vector of variables y.

Here is a simple example:

```
y = {y0, y1, y2};
f = y2;
G = {y0 - 2, {{y1, y0}, {y0, 1}}, {{y2, y1}, {y1, 1}}};
```

The list of constraints in `G` is to be interpreted as:

$$
\begin{aligned}
y_0 - 2 &\geq 0, \\
\begin{bmatrix} y_1 & y_0 \\ y_0 & 1 \end{bmatrix} &\succeq 0, \\
\begin{bmatrix} y_2 & y_1 \\ y_1 & 1 \end{bmatrix} &\succeq 0.
\end{aligned}
$$

The function `SDPMatrices` convert the above symbolic problem into numerical data that can be used to solve an SDP.

```
abc = SDPMatrices[f, G, y]
```

All required data, that is A, b, and c, is stored in the variable `abc` as Mathematica's sparse matrices. Their contents can be revealed using the Mathematica command `Normal`.

```
Normal[abc]
```

The resulting SDP is solved using `SDPSolve`:

```
{Y, X, S, flags} = SDPSolve[abc];
```

The variables `Y` and `S` are the *primal* solutions and `X` is the *dual* solution. Detailed information on the computed solution is found in the variable `flags`.

The package `SDP` is built so as to be easily overloaded with more efficient or more structure functions. See for example SDPFlat and SDPSylvester.

Chapter 8

Pretty Output with Notebooks and TEX

If you want a living version of this chapter just run the notebook `NC/DEMOS/5_PrettyOutput.nb`.

`NCAlgebra` comes with several utilities for beautifying expressions which are output. `NCTeXForm` converts NC expressions into LATEX. `NCTeX` goes a step further and compiles the results expression in LATEX and produces a PDF that can be embedded in notebooks of used on its own.

8.1 Pretty Output

In a Mathematica notebook session the package NCOutput can be used to control how nc expressions are displayed. `NCOutput` does not alter the internal representation of nc expressions, just the way they are displayed on the screen.

The function NCSetOutput can be used to set the display options. For example:

```
NCSetOutput[tp -> False, inv -> True];
```

makes the expression

```
expr = inv[tp[a] + b]
```

be displayed as

$$(\mathrm{tp}[\mathrm{a}] + \mathrm{b})^{-1}$$

Conversely

```
NCSetOutput[tp -> True, inv -> False];
```

makes `expr` be displayed as

$$\mathrm{inv}[\mathrm{a}^{T} + \mathrm{b}]$$

The default settings are

```
NCSetOutput[tp -> True, inv -> True];
```

which makes `expr` be displayed as

$$(\mathrm{a}^{\mathrm{T}} + \mathrm{b})^{-1}$$

The complete set of options and their default values are:

- `NonCommutativeMultiply` (`False`): If `True` `x**y` is displayed as ‘$\mathrm{x} \bullet \mathrm{y}$’;
- `tp` (`True`): If `True` `tp[x]` is displayed as ‘x^{T}’;

- `inv` (`True`): If `True` `inv[x]` is displayed as ‘x^{-1}’;
- `aj` (`True`): If `True` `aj[x]` is displayed as ‘x^*’;
- `co` (`True`): If `True` `co[x]` is displayed as ‘$\bar{\mathrm{x}}$’;
- `rt` (`True`): If `True` `rt[x]` is displayed as ‘$\mathrm{x}^{1/2}$’.

The special symbol `All` can be used to set all options to `True` or `False`, as in

```
NCSetOutput[All -> True];
```

8.2 Using NCTeX

You can load NCTeX using the following command

```
<< NC`
<< NCTeX`
```

`NCTeX` does not need `NCAlgebra` to work. You may want to use it even when not using NCAlgebra. It uses `NCRun`, which is a replacement for Mathematica's Run command to run `pdflatex`, `latex`, `divps`, etc.

WARNING: Mathematica does not come with LaTeX, dvips, etc. The package `NCTeX` does not install these programs but rather assumes that they have been previously installed and are available at the user's standard shell. Use the `Verbose` option to troubleshoot installation problems.

With `NCTeX` loaded you simply type `NCTeX[expr]` and your expression will be converted to a PDF image which, by default, appears in your notebook after being processed by `LaTeX`. See options for information on how to change this behavior to display the PDF on a separate window.

For example:

```
expr = 1 + Sin[x + (y - z)/Sqrt[2]];
NCTeX[expr]
```

produces

$$1 + \sin\left(x + \frac{y-z}{\sqrt{2}}\right)$$

If `NCAlgebra` is not loaded then `NCTeX` uses the built in `TeXForm` to produce the LaTeX expressions. If `NCAlgebra` is loaded, `NCTeXForm` is used. See NCTeXForm for details.

Here is another example:

```
expr = {{1 + Sin[x + (y - z)/2 Sqrt[2]], x/y}, {z, n Sqrt[5]}};
NCTeX[expr]
```

that produces

$$\begin{pmatrix} \sin\left(x + \frac{y-z}{\sqrt{2}}\right) + 1 & \frac{x}{y} \\ z & \sqrt{5}n \end{pmatrix}$$

In some cases Mathematica will have difficulty displaying certain PDF files. When this happens `NCTeX` will span a PDF viewer so that you can look at the formula. If your PDF viewer does not pop up automatically you can force it by passing the following option to `NCTeX`:

```
expr = {{1 + Sin[x + (y - z)/2 Sqrt[2]], x/y}, {z, n Sqrt[5]}};
NCTeX[exp, DisplayPDF -> True]
```

Here is another example were the current version of Mathematica fails to import the PDF:

```
expr = Table[x^i y^(-j) , {i, 0, 10}, {j, 0, 30}];
NCTeX[expr, DisplayPDF -> True]
```

You can also suppress Mathematica from importing the PDF altogether as well. This and other options are covered in detail in the next section.

8.2.1 NCTeX Options

The following command:

```
expr = {{1 + Sin[x + (y - z)/2 Sqrt[2]], x/y}, {z, n Sqrt[5]}};
NCTeX[exp, DisplayPDF -> True, ImportPDF -> False]
```

uses `DisplayPDF -> True` to ensure that the PDF viewer is called and `ImportPDF -> False` to prevent Mathematica from displaying the formula inline. In other words, it displays the formula in the PDF viewer without trying to import the PDF into Mathematica. The default values for these options when using the Mathematica notebook interface are:

1. `DisplayPDF` (`False`)
2. `ImportPDF` (`True`)

When `NCTeX` is invoked using the command line interpreter version of Mathematica the defaults are:

1. `DisplayPDF` (`False`)
2. `ImportPDF` (`True`)

Other useful options and their default options are:

1. `Verbose` (`False`),
2. `BreakEquations` (`True`)
3. `TeXProcessor` (`NCTeXForm`)

Set `BreakEquations -> True` to use the LaTeX package `beqn` to produce nice displays of long equations. Try the following example:

```
expr = Series[Exp[x], {x, 0, 20}]
NCTeX[expr]
```

Use `TexProcessor` to select your own `TeX` converter. If `NCAlgebra` is loaded then `NCTeXForm` is the default. Otherwise Mathematica's `TeXForm` is used.

If `Verbose -> True` you can see a detailed display of what is going on behing the scenes. This is very useful for debugging. For example, try:

```
expr = BesselJ[2, x]
NCTeX[exp, Verbose -> True]
```

to produce an output similar to the following one:

```
* NCTeX - LaTeX processor for NCAlgebra - Version 0.1
> Creating temporary file '/tmp/mNCTeX.tex'...
> Processing '/tmp/mNCTeX.tex'...
> Running 'latex -output-directory=/tmp/  /tmp/mNCTeX 1> "/tmp/mNCRun.out" 2> "/tmp/mNCRun.err"'...
> Running 'dvips -o /tmp/mNCTeX.ps -E /tmp/mNCTeX 1> "/tmp/mNCRun.out" 2> "/tmp/mNCRun.err"'...
> Running 'epstopdf  /tmp/mNCTeX.ps 1> "/tmp/mNCRun.out" 2> "/tmp/mNCRun.err"'...
> Importing pdf file '/tmp/mNCTeX.pdf'...
```

Locate the files with extension .err as indicated by the verbose run of NCTeX to diagnose errors.

The remaining options:

1. `PDFViewer` (`"open"`),
2. `LaTeXCommand` (`"latex"`)
3. `PDFLaTeXCommand` (`Null`)
4. `DVIPSCommand` (`"dvips"`)
5. `PS2PDFCommand` (`"epstopdf"`)

let you specify the names and, when appropriate, the path, of the corresponding programs to be used by `NCTeX`. Alternatively, you can also directly implement custom versions of

```
NCRunDVIPS
NCRunLaTeX
NCRunPDFLaTeX
NCRunPDFViewer
NCRunPS2PDF
```

Those commands are invoked using `NCRun`. Look at the documentation for the package NCRun for more details.

8.3 Using NCTeXForm

`NCTeXForm` is a replacement for Mathematica's `TeXForm` which adds definitions allowing it to handle noncommutative expressions. It works just as `TeXForm`. `NCTeXForm` is automatically loaded with `NCAlgebra` and is the default TEX processor for `NCTeX`.

Here is an example:

```
SetNonCommutative[a, b, c, x, y];
exp = a ** x ** tp[b] - inv[c ** inv[a + b ** c] ** tp[y] + d]
NCTeXForm[exp]
```

produces

```
a.x.{b}^T-{\left(d+c.{\left(a+b.c\right)}^{-1}.{y}^T\right)}^{-1}
```

Note that the LaTeX output contains special code so that the expression looks neat on the screen. You can see the result using `NCTeX` to convert the expression to PDF. Try

```
SetOptions[NCTeX, TeXProcessor -> NCTeXForm];
NCTeX[exp]
```

to produce

$$a.x.b^T-\left(d+c.(a+b.c)^{-1}.y^T\right)^{-1}$$

`NCTeX` represents noncommutative products with a dot (`.`) in order to distinguish it from its commutative cousin. We can see the difference in an expression that has both commutative and noncommutative products:

```
exp = 2 a ** b - 3 c ** d
NCTeX[exp]
```

produces

$$2\,(a.b)-3(c.d)$$

NCTeXForm handles lists and matrices as well. Here is a list:

```
exp = {x, tp[x], x + y, x + tp[y], x + inv[y], x ** x}
NCTeX[exp]
```

and its output:

$$\{x, x^T, x+y, x+y^T, x+y^{-1}, x.x\}$$

and here is a matrix example:

```
exp = {{x, y}, {y, z}}
NCTeX[exp]
```

and its output:

$$\begin{bmatrix} x & y \\ y & z \end{bmatrix}$$

Here are some more examples:

```
exp = {{1 + Sin[x + (y - z)/2 Sqrt[2]], x/y}, {z, n Sqrt[5]}}
NCTeX[exp]
```

produces

$$\begin{bmatrix} 1+\sin\left(x+\frac{1}{\sqrt{2}}\left(y-z\right)\right) & xy^{-1} \\ z & \sqrt{5}n \end{bmatrix}$$

```
exp = {inv[x + y], inv[x + inv[y]]}
NCTeX[exp]
```

produces:

$$\{(x+y)^{-1}, \left(x+y^{-1}\right)^{-1}\}$$

```
exp = {Sin[x], x y, Sin[x] y, Sin[x + y], Cos[gamma],
       Sin[alpha] tp[x] ** (y - tp[y]), (x + tp[x]) (y ** z), -tp[y], 1/2,
       Sqrt[2] x ** y}
NCTeX[exp]
```

produces:

$$\{\sin x, xy, y\sin x, \sin\left(x+y\right), \cos\gamma, \left(x^T . \left(y-y^T\right)\right)\sin\alpha, yz\left(x+x^T\right), -y^T, \frac{1}{2}, \sqrt{2}\left(x.y\right)\}$$

```
exp = inv[x + tp[inv[y]]]
NCTeX[exp]
```

produces:

$$\left(x+y^{T^{-1}}\right)^{-1}$$

`NCTeXForm` does not know as many functions as `TeXForm`. In some cases `TeXForm` will produce better results. Compare:

```
exp = BesselJ[2, x]
NCTeX[exp, TeXProcessor -> NCTeXForm]
```

output:

$$\mathrm{BesselJ}\left(2, x\right)$$

with

```
NCTeX[exp, TeXProcessor -> TeXForm]
```

output:

$$J_2(x)$$

It should be easy to customize `NCTeXForm` though. Just overload `NCTeXForm`. In this example:

```
NCTeXForm[BesselJ[x_, y_]] := Format[BesselJ[x, y], TeXForm]
```

makes

```
NCTeX[exp, TeXProcessor -> NCTeXForm]
```

produce

$$J_2(x)$$

Part II

Reference Manual

Chapter 9

Reference Manual

The following chapters and sections describes packages inside `NCAlgebra`.

Packages are automatically loaded unless otherwise noted.

9.1 NC

NC is a meta package that enables the functionality of the *NCAlgebra suite* of non-commutative algebra packages for Mathematica.

The package can be loaded using `Get`, as in

```
<< NC`
```

or `Needs`, as in

```
Needs["NC`"]
```

Once `NC` is loaded, you can then proceed to load any other package from the *NCAlgebra suite*.

For example you can load the package `NCAlgebra` using

```
<< NCAlgebra`
```

9.1.1 Options

The following `options` can be set using `SetOptions` before loading other packages:

- `SmallCapSymbolsNonCommutative` (`True`): If `True`, loading `NCAlgebra` will set all global single letter small cap symbols as noncommutative;
- `ShowBanner` (`True`): If `True`, a banner, when available, will be shown during the first loading of a package.

9.2 NCAlgebra

NCAlgebra is the main package of the *NCAlgebra suite* of non-commutative algebra packages for Mathematica.

The package can be loaded using `Get`, as in

```
<< NCAlgebra`
```

or `Needs`, as in

```
Needs["NCAlgebra`"]
```

If the option SmallCapSymbolsNonCommutative is `True` then `NCAlgebra` will set all global single letter small cap symbols as noncommutative. If that is not desired simply set `SmallCapSymbolsNonCommutative` to `False` before loading `NCAlgebra`, as in

```
SetOptions[NC, SmallCapSymbolsNonCommutative -> False]
<< NCAlgebra`
```

A message will be issued warning users whether any letters have been set as noncommutative upon loading. Those messages are documented here. Users can use Mathematica's `Quiet` and `Off` if they do not want these messages to display. For example,

```
Off[NCAlgebra::SmallCapSymbolsNonCommutative]
<< NCAlgebra`
```

or

```
SetOptions[NC, SmallCapSymbolsNonCommutative -> False]
Off[NCAlgebra::NoSymbolsNonCommutative]
<< NCAlgebra`
```

will load `NCAlgebra` without issuing a symbol assignment message.

Upon loading `NCAlgebra` for the first time, a large banner will be shown. If you do not want this banner to be displayed at all set the option ShowBanner to `False` before loading, as in

```
SetOptions[NC, ShowBanner -> False]
<< NCAlgebra`
```

9.2.1 Messages

One of the following messages will be displayed after loading.

- `NCAlgebra::SmallCapSymbolsNonCommutative`, if small cap single letter symbols have been set as noncomutative;
- `NCAlgebra::NoSymbolsNonCommutative`, if no symbols have been set as noncomutative by `NCAlgebra`.

Chapter 10

Packages for manipulating NC expressions

10.1 NonCommutativeMultiply

`NonCommutativeMultiply` is the main package that provides noncommutative functionality to Mathematica's native `NonCommutativeMultiply` bound to the operator `**`.

Members are:

- aj
- co
- Id
- inv
- tp
- rt
- CommutativeQ
- NonCommutativeQ
- SetCommutative
- SetCommutativeHold
- SetNonCommutative
- SetNonCommutativeHold
- SetCommutativeFunction
- SetCommutingOperators
- UnsetCommutingOperators
- CommutingOperatorsQ
- Commutative
- CommuteEverything
- BeginCommuteEverything
- EndCommuteEverything
- ExpandNonCommutativeMultiply

Aliases are:

- SNC for SetNonCommutative
- NCExpand for ExpandNonCommutativeMultiply
- NCE for ExpandNonCommutativeMultiply

10.1.1 aj

`aj[expr]` is the adjoint of expression `expr`. It is a conjugate linear involution.

See also: tp, co.

10.1.2 co

`co[expr]` is the conjugate of expression `expr`. It is a linear involution.

See also: aj.

10.1.3 Id

`Id` is noncommutative multiplicative identity. Actually Id is now set equal `1`.

10.1.4 inv

`inv[expr]` is the 2-sided inverse of expression `expr`.

If `Options[inv, Distrubute]` is `False` (the default) then

`inv[a**b]`

returns `inv[a**a]`. Conversely, if `Options[inv, Distrubute]` is `True` then it returns `inv[b]**inv[a]`.

10.1.5 rt

`rt[expr]` is the root of expression `expr`.

10.1.6 tp

`tp[expr]` is the tranpose of expression `expr`. It is a linear involution.

See also: aj, co.

10.1.7 CommutativeQ

`CommutativeQ[expr]` is `True` if expression `expr` is commutative (the default), and `False` if `expr` is noncommutative.

See also: SetCommutative, SetNonCommutative.

10.1.8 NonCommutativeQ

`NonCommutativeQ[expr]` is equal to `Not[CommutativeQ[expr]]`.

See also: CommutativeQ.

10.1.9 SetCommutative

`SetCommutative[a,b,c,...]` sets all the `Symbols` `a`, `b`, `c`, ... to be commutative.

See also: SetNonCommutative, CommutativeQ, NonCommutativeQ.

10.1.10 SetCommutativeHold

`SetCommutativeHold[a,b,c,...]` sets all the `Symbols` `a`, `b`, `c`, ... to be commutative.

`SetCommutativeHold` has attribute `HoldAll` and can be used to set Symbols which have already been assigned a value.

See also: SetNonCommutativeHold, SetCommutative, SetNonCommutative, CommutativeQ, NonCommutativeQ.

10.1.11 SetNonCommutative

`SetNonCommutative[a,b,c,...]` sets all the `Symbols` `a`, `b`, `c`, ... to be noncommutative.

See also: SetCommutative, CommutativeQ, NonCommutativeQ.

10.1.12 SetNonCommutativeHold

`SetNonCommutativeHold[a,b,c,...]` sets all the `Symbols` `a`, `b`, `c`, ... to be noncommutative.

`SetNonCommutativeHold` has attribute `HoldAll` and can be used to set Symbols which have already been assigned a value.

See also: SetCommutativeHold, SetCommutative, CommutativeQ, NonCommutativeQ.

10.1.13 SetCommutativeFunction

`SetCommutativeFunction[f]` sets expressions with `Head` `f`, i.e. functions, to be commutative.

By default, expressions in which the `Head` or any of its arguments is noncommutative will be considered noncommutative. For example,

```
SetCommutative[trace];
a ** b ** trace[a ** b]
```

evaluates to `a ** b ** trace[a ** b]` while

```
SetCommutativeFunction[trace];
a ** b ** trace[a ** b]
```

evaluates to `trace[a ** b] * a ** b`.

See also: SetCommutative, SetNonCommutative, CommutativeQ, NonCommutativeQ, tr.

10.1.14 SNC

`SNC` is an alias for `SetNonCommutative`.

See also: SetNonCommutative.

10.1.15 SetCommutingOperators

`SetCommutingOperators[a,b]` will define a rule that substitute any noncommutative product `b ** a` by `a ** b`, effectively making the pair `a` and `b` commutative. If you want to create a rule to replace `a ** b` by `b ** a` use `SetCommutingOperators[b,a]` instead.

See also: UnsetCommutingOperators, CommutingOperatorsQ

10.1.16 UnsetCommutingOperators

`UnsetCommutingOperators[a,b]` remove any rules previously created by `SetCommutingOperators[a,b]` or `SetCommutingOperators[b,a]`.

See also: SetCommutingOperators, CommutingOperatorsQ

10.1.17 CommutingOperatorsQ

`CommutingOperatorsQ[a,b]` returns `True` if `a` and `b` are commuting operators.

See also: SetCommutingOperators, UnsetCommutingOperators

10.1.18 Commutative

`Commutative[symbol]` is commutative even if `symbol` is noncommutative.

See also: CommuteEverything, CommutativeQ, SetCommutative, SetNonCommutative.

10.1.19 CommuteEverything

`CommuteEverything[expr]` is an alias for BeginCommuteEverything.

See also: BeginCommuteEverything, Commutative.

10.1.20 BeginCommuteEverything

`BeginCommuteEverything[expr]` sets all symbols appearing in `expr` as commutative so that the resulting expression contains only commutative products or inverses. It issues messages warning about which symbols have been affected.

`EndCommuteEverything[]` restores the symbols noncommutative behaviour.

`BeginCommuteEverything` answers the question *what does it sound like?*

See also: EndCommuteEverything, Commutative.

10.1.21 EndCommuteEverything

`EndCommuteEverything[expr]` restores noncommutative behaviour to symbols affected by `BeginCommuteEverything`.

See also: BeginCommuteEverything, Commutative.

10.1.22 ExpandNonCommutativeMultiply

`ExpandNonCommutativeMultiply[expr]` expands out `**`s in `expr`.

For example

`ExpandNonCommutativeMultiply[a**(b+c)]`

returns

`a**b + a**c`.

See also: NCExpand, NCE.

10.1.23 NCExpand

`NCExpand` is an alias for `ExpandNonCommutativeMultiply`.

See also: ExpandNonCommutativeMultiply, NCE.

10.1.24 NCE

`NCE` is an alias for `ExpandNonCommutativeMultiply`.

See also: ExpandNonCommutativeMultiply, NCExpand.

10.2 NCTr

Members are:

- tr
- SortCyclicPermutation
- SortedCyclicPermutationQ

10.2.1 tr

`tr[expr]` is an linear operator with the following properties:

- `tr` automatically distributes over sums;
- when `expr` is a noncommutative product, then product is sorted; for example `tr[b ** a]` evaluates into `tr[a ** b]`

See also: SortCyclicPermutation, SortedCyclicPermutationQ.

10.2.2 SortCyclicPermutation

`SortedCyclicPermutation[list]` returns a cyclic permutation of list sorted in ascending order.

See also: SortedCyclicPermutationQ.

10.2.3 SortedCyclicPermutationQ

`SortCyclicPermutationQ[list]` returns `True` if `list` is a sorted cyclic permutation.

See also: SortCyclicPermutation.

10.3 NCCollect

Members are:

- NCCollect
- NCCollectSelfAdjoint
- NCCollectSymmetric
- NCStrongCollect
- NCStrongCollectSelfAdjoint
- NCStrongCollectSymmetric
- NCCompose
- NCDecompose
- NCTermsOfDegree

10.3.1 NCCollect

`NCCollect[expr,vars]` collects terms of nc expression `expr` according to the elements of `vars` and attempts to combine them. It is weaker than NCStrongCollect in that only same order terms are collected togther. It basically is `NCCompose[NCStrongCollect[NCDecompose]]`.

If `expr` is a rational nc expression then degree correspond to the degree of the polynomial obtained using NCRationalToNCPolynomial.

`NCCollect` also works with nc expressions instead of *Symbols* in vars. In this case nc expressions are replaced by new variables and `NCCollect` is called using the resulting expression and the newly created *Symbols*.

This command internally converts nc expressions into the special `NCPolynomial` format.

`NCCollect[expr,vars,options]` uses options.

The following option is available:

- `ByTotalDegree` (`False`): whether to collect by total or partial degree.

Notes:

While `NCCollect[expr, vars]` always returns mathematically correct expressions, it may not collect `vars` from as many terms as one might think it should.

See also: NCStrongCollect, NCCollectSymmetric, NCCollectSelfAdjoint, NCStrongCollectSymmetric, NCStrongCollectSelfAdjoint, NCRationalToNCPolynomial.

10.3.2 NCCollectSelfAdjoint

`NCCollectSelfAdjoint[expr,vars]` allows one to collect terms of nc expression `expr` on the variables `vars` and their adjoints without writing out the adjoints.

This command internally converts nc expressions into the special `NCPolynomial` format.

`NCCollectSelfAdjoint[expr,vars,options]` uses options.

The following option is available:

- `ByTotalDegree` (`False`): whether to collect by total or partial degree.

See also: NCCollect, NCStrongCollect, NCCollectSymmetric, NCStrongCollectSymmetric, NCStrongCollectSelfAdjoint.

10.3.3 NCCollectSymmetric

`NCCollectSymmetric[expr,vars]` allows one to collect terms of nc expression `expr` on the variables `vars` and their transposes without writing out the transposes.

This command internally converts nc expressions into the special `NCPolynomial` format.

`NCCollectSymmetric[expr,vars,options]` uses options.

The following option is available:

- `ByTotalDegree` (`False`): whether to collect by total or partial degree.

See also: NCCollect, NCStrongCollect, NCCollectSelfAdjoint, NCStrongCollectSymmetric, NCStrongCollectSelfAdjoint.

10.3.4 NCStrongCollect

`NCStrongCollect[expr,vars]` collects terms of expression `expr` according to the elements of `vars` and attempts to combine by association.

In the noncommutative case the Taylor expansion and so the collect function is not uniquely specified. The function `NCStrongCollect` often collects too much and while correct it may be stronger than you want.

For example, a symbol `x` will factor out of terms where it appears both linearly and quadratically thus mixing orders.

This command internally converts nc expressions into the special `NCPolynomial` format.

See also: NCCollect, NCCollectSymmetric, NCCollectSelfAdjoint, NCStrongCollectSymmetric, NCStrongCollectSelfAdjoint.

10.3.5 NCStrongCollectSelfAdjoint

`NCStrongCollectSymmetric[expr,vars]` allows one to collect terms of nc expression `expr` on the variables `vars` and their transposes without writing out the transposes.

This command internally converts nc expressions into the special `NCPolynomial` format.

See also: NCCollect, NCStrongCollect, NCCollectSymmetric, NCCollectSelfAdjoint, NCStrongCollectSymmetric.

10.3.6 NCStrongCollectSymmetric

`NCStrongCollectSymmetric[expr,vars]` allows one to collect terms of nc expression `expr` on the variables `vars` and their transposes without writing out the transposes.

This command internally converts nc expressions into the special `NCPolynomial` format.

See also: NCCollect, NCStrongCollect, NCCollectSymmetric, NCCollectSelfAdjoint, NCStrongCollectSelfAdjoint.

10.3.7 NCCompose

`NCCompose[dec]` will reassemble the terms in `dec` which were decomposed by `NCDecompose`.

`NCCompose[dec, degree]` will reassemble only the terms of degree `degree`.

The expression `NCCompose[NCDecompose[p,vars]]` will reproduce the polynomial `p`.

The expression `NCCompose[NCDecompose[p,vars], degree]` will reproduce only the terms of degree `degree`.

This command internally converts nc expressions into the special `NCPolynomial` format.

See also: NCDecompose, NCPDecompose.

10.3.8 NCDecompose

`NCDecompose[p,vars]` gives an association of elements of the nc polynomial `p` in variables `vars` in which elements of the same order are collected together.

`NCDecompose[p]` treats all nc letters in `p` as variables.

This command internally converts nc expressions into the special `NCPolynomial` format.

Internally `NCDecompose` uses `NCPDecompose`.

See also: NCCompose, NCPDecompose.

10.3.9 NCTermsOfDegree

`NCTermsOfDegree[expr,vars,degrees]` returns an expression such that each term has degree `degrees` in variables `vars`.

For example,

`NCTermsOfDegree[x**y**x - x**x**y + x**w + z**w, {x,y}, {2,1}]`

returns `x**y**x - x**x**y`,

`NCTermsOfDegree[x**y**x - x**x**y + x**w + z**w, {x,y}, {1,0}]`

returns `x**w`,

```
NCTermsOfDegree[x**y**x - x**x**y + x**w + z**w, {x,y}, {0,0}]
```

returns `z**w`, and

```
NCTermsOfDegree[x**y**x - x**x**y + x**w + z**w, {x,y}, {0,1}]
```

returns `0`.

This command internally converts nc expressions into the special `NCPolynomial` format.

See also: NCTermsOfTotalDegree, NCDecompose, NCPDecompose.

10.3.10 NCTermsOfTotalDegree

`NCTermsOfTotalDegree[expr,vars,degree]` returns an expression such that each term has total degree `degree` in variables `vars`.

For example,

```
NCTermsOfTotalDegree[x**y**x - x**x**y + x**w + z**w, {x,y}, 3]
```

returns `x**y**x - x**x**y`,

```
NCTermsOfTotalDegree[x**y**x - x**x**y + x**w + z**w, {x,y}, 1]
```

returns `x**w`,

```
NCTermsOfTotalDegree[x**y**x - x**x**y + x**w + z**w, {x,y}, 0]
```

returns `z**w`, and

```
NCTermsOfTotalDegree[x**y**x - x**x**y + x**w + z**w, {x,y}, 2]
```

returns `0`.

This command internally converts nc expressions into the special `NCPolynomial` format.

See also: NCTermsOfDegree, NCDecompose, NCPDecompose.

10.4 NCReplace

NCReplace is a package containing several functions that are useful in making replacements in noncommutative expressions. It offers replacements to Mathematica's `Replace`, `ReplaceAll`, `ReplaceRepeated`, and `ReplaceList` functions.

Commands in this package replace the old `Substitute` and `Transform` family of command which are been deprecated. The new commands are much more reliable and work faster than the old commands. From the beginning, substitution was always problematic and certain patterns would be missed. We reassure that the call expression that are returned are mathematically correct but some opportunities for substitution may have been missed.

Members are:

- NCReplace
- NCReplaceAll
- NCReplaceList
- NCReplaceRepeated
- NCMakeRuleSymmetric
- NCMakeRuleSelfAdjoint
- NCReplaceSymmetric
- NCReplaceAllSymmetric
- NCReplaceListSymmetric
- NCReplaceRepeatedSymmetric

- NCReplaceSelfAdjoint
- NCReplaceAllSelfAdjoint
- NCReplaceListSelfAdjoint
- NCReplaceRepeatedSelfAdjoint
- NCMatrixExpand
- NCMatrixReplaceAll
- NCMatrixReplaceRepeated

Aliases:

- NCR for NCReplace
- NCRA for NCReplaceAll
- NCRL for NCReplaceList
- NCRR for NCReplaceRepeated
- NCRSym for NCReplaceSymmetric
- NCRASym for NCReplaceAllSymmetric
- NCRLSym for NCReplaceListSymmetric
- NCRRSym for NCReplaceRepeatedSymmetric
- NCRSA for NCReplaceSelfAdjoint
- NCRASA for NCReplaceAllSelfAdjoint
- NCRLSA for NCReplaceListSelfAdjoint
- NCRRSA for NCReplaceRepeatedSelfAdjoint

10.4.1 NCReplace

`NCReplace[expr,rules]` applies a rule or list of rules `rules` in an attempt to transform the entire nc expression `expr`.

`NCReplace[expr,rules,levelspec]` applies `rules` to parts of `expr` specified by `levelspec`.

See also: NCReplaceAll, NCReplaceList, NCReplaceRepeated.

10.4.2 NCReplaceAll

`NCReplaceAll[expr,rules]` applies a rule or list of rules `rules` in an attempt to transform each part of the nc expression `expr`.

See also: NCReplace, NCReplaceList, NCReplaceRepeated.

10.4.3 NCReplaceList

`NCReplace[expr,rules]` attempts to transform the entire nc expression `expr` by applying a rule or list of rules `rules` in all possible ways, and returns a list of the results obtained.

`ReplaceList[expr,rules,n]` gives a list of at most `n` results.

See also: NCReplace, NCReplaceAll, NCReplaceRepeated.

10.4.4 NCReplaceRepeated

`NCReplaceRepeated[expr,rules]` repeatedly performs replacements using rule or list of rules `rules` until `expr` no longer changes.

See also: NCReplace, NCReplaceAll, NCReplaceList.

10.4.5 NCR

`NCR` is an alias for `NCReplace`.

See also: NCReplace.

10.4.6 NCRA

`NCRA` is an alias for `NCReplaceAll`.

See also: NCReplaceAll.

10.4.7 NCRR

`NCRR` is an alias for `NCReplaceRepeated`.

See also: NCReplaceRepeated.

10.4.8 NCRL

`NCRL` is an alias for `NCReplaceList`.

See also: NCReplaceList.

10.4.9 NCMakeRuleSymmetric

`NCMakeRuleSymmetric[rules]` add rules to transform the transpose of the left-hand side of `rules` into the transpose of the right-hand side of `rules`.

See also: NCMakeRuleSelfAdjoint, NCReplace, NCReplaceAll, NCReplaceList, NCReplaceRepeated.

10.4.10 NCMakeRuleSelfAdjoint

`NCMakeRuleSelfAdjoint[rules]` add rules to transform the adjoint of the left-hand side of `rules` into the adjoint of the right-hand side of `rules`.

See also: NCMakeRuleSymmetric, NCReplace, NCReplaceAll, NCReplaceList, NCReplaceRepeated.

10.4.11 NCReplaceSymmetric

`NCReplaceSymmetric[expr, rules]` applies `NCMakeRuleSymmetric` to `rules` before calling `NCReplace`.

See also: NCReplace, NCMakeRuleSymmetric.

10.4.12 NCReplaceAllSymmetric

`NCReplaceAllSymmetric[expr, rules]` applies `NCMakeRuleSymmetric` to `rules` before calling `NCReplaceAll`.

See also: NCReplaceAll, NCMakeRuleSymmetric.

10.4.13 NCReplaceRepeatedSymmetric

`NCReplaceRepeatedSymmetric[expr, rules]` applies `NCMakeRuleSymmetric` to `rules` before calling `NCReplaceRepeated`.

See also: NCReplaceRepeated, NCMakeRuleSymmetric.

10.4.14 NCReplaceListSymmetric

`NCReplaceListSymmetric[expr, rules]` applies `NCMakeRuleSymmetric` to `rules` before calling `NCReplaceList`.

See also: NCReplaceList, NCMakeRuleSymmetric.

10.4.15 NCRSym

`NCRSym` is an alias for `NCReplaceSymmetric`.

See also: NCReplaceSymmetric.

10.4.16 NCRASym

`NCRASym` is an alias for `NCReplaceAllSymmetric`.

See also: NCReplaceAllSymmetric.

10.4.17 NCRRSym

`NCRRSym` is an alias for `NCReplaceRepeatedSymmetric`.

See also: NCReplaceRepeatedSymmetric.

10.4.18 NCRLSym

`NCRLSym` is an alias for `NCReplaceListSymmetric`.

See also: NCReplaceListSymmetric.

10.4.19 NCReplaceSelfAdjoint

`NCReplaceSelfAdjoint[expr, rules]` applies `NCMakeRuleSelfAdjoint` to `rules` before calling `NCReplace`.

See also: NCReplace, NCMakeRuleSelfAdjoint.

10.4.20 NCReplaceAllSelfAdjoint

`NCReplaceAllSelfAdjoint[expr, rules]` applies `NCMakeRuleSelfAdjoint` to `rules` before calling `NCReplaceAll`.

See also: NCReplaceAll, NCMakeRuleSelfAdjoint.

10.4.21 NCReplaceRepeatedSelfAdjoint

`NCReplaceRepeatedSelfAdjoint[expr, rules]` applies `NCMakeRuleSelfAdjoint` to `rules` before calling `NCReplaceRepeated`.

See also: NCReplaceRepeated, NCMakeRuleSelfAdjoint.

10.4.22 NCReplaceListSelfAdjoint

`NCReplaceListSelfAdjoint[expr, rules]` applies `NCMakeRuleSelfAdjoint` to `rules` before calling `NCReplaceList`.

See also: NCReplaceList, NCMakeRuleSelfAdjoint.

10.4.23 NCRSA

`NCRSA` is an alias for `NCReplaceSymmetric`.

See also: NCReplaceSymmetric.

10.4.24 NCRASA

`NCRASA` is an alias for `NCReplaceAllSymmetric`.

See also: NCReplaceAllSymmetric.

10.4.25 NCRRSA

`NCRRSA` is an alias for `NCReplaceRepeatedSymmetric`.

See also: NCReplaceRepeatedSymmetric.

10.4.26 NCRLSA

`NCRLSA` is an alias for `NCReplaceListSymmetric`.

See also: NCReplaceListSymmetric.

10.4.27 NCMatrixExpand

`NCMatrixExpand[expr]` expands `inv` and `**` of matrices appearing in nc expression `expr`. It effectively substitutes `inv` for `NCInverse` and `**` by `NCDot`.

See also: NCInverse, NCDot.

10.4.28 NCMatrixReplaceAll

`NCMatrixReplaceAll[expr,rules]` applies a rule or list of rules `rules` in an attempt to transform each part of the nc expression `expr`.

`NCMatrixReplaceAll` works as `NCReplaceAll` but takes extra steps to make sure substitutions work with matrices.

See also: NCReplaceAll, NCMatrixReplaceRepeated.

10.4.29 NCMatrixReplaceRepeated

`NCMatrixReplaceRepeated[expr,rules]` repeatedly performs replacements using rule or list of rules `rules` until `expr` no longer changes.

`NCMatrixReplaceRepeated` works as `NCReplaceRepeated` but takes extra steps to make sure substitutions work with matrices.

See also: NCReplaceRepeated, NCMatrixReplaceAll.

10.5 NCSelfAdjoint

Members are:

- NCSymmetricQ
- NCSymmetricTest
- NCSymmetricPart
- NCSelfAdjointQ
- NCSelfAdjointTest

10.5.1 NCSymmetricQ

`NCSymmetricQ[expr]` returns *True* if `expr` is symmetric, i.e. if `tp[exp] == exp`.

`NCSymmetricQ` attempts to detect symmetric variables using `NCSymmetricTest`.

See also: NCSelfAdjointQ, NCSymmetricTest.

10.5.2 NCSymmetricTest

`NCSymmetricTest[expr]` attempts to establish symmetry of `expr` by assuming symmetry of its variables.

`NCSymmetricTest[exp,options]` uses `options`.

`NCSymmetricTest` returns a list of two elements:

- the first element is *True* or *False* if it succeeded to prove `expr` symmetric.
- the second element is a list of the variables that were made symmetric.

The following options can be given:

- `SymmetricVariables`: list of variables that should be considered symmetric; use `All` to make all variables symmetric;
- `ExcludeVariables`: list of variables that should not be considered symmetric; use `All` to exclude all variables;
- `Strict`: treats as non-symmetric any variable that appears inside `tp`.

See also: NCSymmetricQ, NCNCSelfAdjointTest.

10.5.3 NCSymmetricPart

`NCSymmetricPart[expr]` returns the *symmetric part* of `expr`.

`NCSymmetricPart[exp,options]` uses `options`.

`NCSymmetricPart[expr]` returns a list of two elements:

- the first element is the *symmetric part* of `expr`;
- the second element is a list of the variables that were made symmetric.

`NCSymmetricPart[expr]` returns `{$Failed, {}}` if `expr` is not symmetric.

For example:

```
{answer, symVars} = NCSymmetricPart[a ** x + x ** tp[a] + 1];
```

returns

```
answer = 2 a ** x + 1
symVars = {x}
```

The following options can be given:

- `SymmetricVariables`: list of variables that should be considered symmetric; use `All` to make all variables symmetric;
- `ExcludeVariables`: list of variables that should not be considered symmetric; use `All` to exclude all variables.
- `Strict`: treats as non-symmetric any variable that appears inside `tp`.

See also: NCSymmetricTest.

10.5.4 NCSelfAdjointQ

`NCSelfAdjointQ[expr]` returns true if `expr` is self-adjoint, i.e. if `aj[exp] == exp`.

See also: NCSymmetricQ, NCSelfAdjointTest.

10.5.5 NCSelfAdjointTest

`NCSelfAdjointTest[expr]` attempts to establish whether `expr` is self-adjoint by assuming that some of its variables are self-adjoint or symmetric. `NCSelfAdjointTest[expr,options]` uses `options`.

`NCSelfAdjointTest` returns a list of three elements:

- the first element is *True* or *False* if it succeded to prove `expr` self-adjoint.
- the second element is a list of variables that were made self-adjoint.
- the third element is a list of variables that were made symmetric.

The following options can be given:

- `SelfAdjointVariables`: list of variables that should be considered self-adjoint; use `All` to make all variables self-adjoint;
- `SymmetricVariables`: list of variables that should be considered symmetric; use `All` to make all variables symmetric;
- `ExcludeVariables`: list of variables that should not be considered symmetric; use `All` to exclude all variables.
- `Strict`: treats as non-self-adjoint any variable that appears inside `aj`.

See also: NCSelfAdjointQ.

10.6 NCSimplifyRational

NCSimplifyRational is a package with function that simplifies noncommutative expressions and certain functions of their inverses.

`NCSimplifyRational` simplifies rational noncommutative expressions by repeatedly applying a set of reduction rules to the expression. `NCSimplifyRationalSinglePass` does only a single pass.

Rational expressions of the form

`inv[A + terms]`

are first normalized to

inv[1 + terms/A]/A

using `NCNormalizeInverse`. Here `A` is commutative.

For each `inv` found in expression, a custom set of rules is constructed based on its associated NC Groebner basis.

For example, if

`inv[mon1 + ... + K lead]`

where `lead` is the leading monomial with the highest degree then the following rules are generated:

Original	Transformed
inv[mon1 + … + K lead] lead	(1 - inv[mon1 + … + K lead] (mon1 + …))/K
lead inv[mon1 + … + K lead]	(1 - (mon1 + …) inv[mon1 + … + K lead])/K

Finally the following pattern based rules are applied:

Original	Transformed
inv[a] inv[1 + K a b]	inv[a] - K b inv[1 + K a b]
inv[a] inv[1 + K a]	inv[a] - K inv[1 + K a]
inv[1 + K a b] inv[b]	inv[b] - K inv[1 + K a b] a
inv[1 + K a] inv[a]	inv[a] - K inv[1 + K a]
inv[1 + K a b] a	a inv[1 + K b a]
inv[A inv[a] + B b] inv[a]	(1/A) inv[1 + (B/A) a b]
inv[a] inv[A inv[a] + K b]	(1/A) inv[1 + (B/A) b a]

`NCPreSimplifyRational` only applies pattern based rules from the second table above. In addition, the following two rules are applied:

Original	Transformed
inv[1 + K a b] a b	(1 - inv[1 + K a b])/K
inv[1 + K a] a	(1 - inv[1 + K a])/K
a b inv[1 + K a b]	(1 - inv[1 + K a b])/K
a inv[1 + K a]	(1 - inv[1 + K a])/K

Rules in `NCSimplifyRational` and `NCPreSimplifyRational` are applied repeatedly.

Rules in `NCSimplifyRationalSinglePass` and `NCPreSimplifyRationalSinglePass` are applied only once.

The particular ordering of monomials used by `NCSimplifyRational` is the one implied by the `NCPolynomial` format. This ordering is a variant of the deg-lex ordering where the lexical ordering is Mathematica's natural ordering.

`NCSimplifyRational` is limited by its rule list and what rules are best is unknown and might depend on additional assumptions. For example:

```
NCSimplifyRational[y ** inv[y + x ** y]]
```

returns `y ** inv[y + x ** y]` not `inv[1 + x]`, which is what one would expect if `y` were to be invertible. Indeed,

```
NCSimplifyRational[inv[y] ** inv[inv[y] + x ** inv[y]]]
```

does return `inv[1 + x]`, since in this case the appearing of `inv[y]` trigger rules that implicitely assume `y` is invertible.

Members are:

- NCNormalizeInverse
- NCSimplifyRational
- NCSimplifyRationalSinglePass
- NCPreSimplifyRational
- NCPreSimplifyRationalSinglePass

Aliases:

- NCSR for NCSimplifyRational

10.6.1 NCNormalizeInverse

`NCNormalizeInverse[expr]` transforms all rational NC expressions of the form `inv[K + b]` into `inv[1 + (1/K) b]/K` if `A` is commutative.

See also: NCSimplifyRational, NCSimplifyRationalSinglePass.

10.6.2 NCSimplifyRational

`NCSimplifyRational[expr]` repeatedly applies `NCSimplifyRationalSinglePass` in an attempt to simplify the rational NC expression `expr`.

See also: NCNormalizeInverse, NCSimplifyRationalSinglePass.

10.6.3 NCSR

`NCSR` is an alias for `NCSimplifyRational`.

See also: NCSimplifyRational.

10.6.4 NCSimplifyRationalSinglePass

`NCSimplifyRationalSinglePass[expr]` applies a series of custom rules only once in an attempt to simplify the rational NC expression `expr`.

See also: NCNormalizeInverse, NCSimplifyRational.

10.6.5 NCPreSimplifyRational

`NCPreSimplifyRational[expr]` repeatedly applies `NCPreSimplifyRationalSinglePass` in an attempt to simplify the rational NC expression `expr`.

See also: NCNormalizeInverse, NCPreSimplifyRationalSinglePass.

10.6.6 NCPreSimplifyRationalSinglePass

`NCPreSimplifyRationalSinglePass[expr]` applies a series of custom rules only once in an attempt to simplify the rational NC expression `expr`.

See also: NCNormalizeInverse, NCPreSimplifyRational.

10.7 NCDiff

NCDiff is a package containing several functions that are used in noncommutative differention of functions and polynomials.

Members are:

- NCDirectionalD
- NCGrad
- NCHessian
- NCIntegrate

Members being deprecated:

- DirectionalD

10.7.1 NCDirectionalD

`NCDirectionalD[expr, {var1, h1}, ...]` takes the directional derivative of expression `expr` with respect to variables `var1`, `var2`, … successively in the directions `h1`, `h2`, ….

For example, if:

```
expr = a**inv[1+x]**b + x**c**x
```

then

NCDirectionalD[expr, {x,h}]

returns

h**c**x + x**c**h - a**inv[1+x]**h**inv[1+x]**b

In the case of more than one variables NCDirectionalD[expr, {x,h}, {y,k}] takes the directional derivative of expr with respect to x in the direction h and with respect to y in the direction k. For example, if:

expr = x**q**x - y**x

then

NCDirectionalD[expr, {x,h}, {y,k}]

returns

h**q**x + x**q*h - y**h - k**x

See also: NCGrad, NCHessian.

10.7.2 NCGrad

NCGrad[expr, var1, ...] gives the nc gradient of the expression expr with respect to variables var1, var2, ... If there is more than one variable then NCGrad returns the gradient in a list.

The transpose of the gradient of the nc expression expr is the derivative with respect to the direction h of the trace of the directional derivative of expr in the direction h.

For example, if:

expr = x**a**x**b + x**c**x**d

then its directional derivative in the direction h is

NCDirectionalD[expr, {x,h}]

which returns

h**a**x**b + x**a**h**b + h**c**x**d + x**c**h**d

and

NCGrad[expr, x]

returns the nc gradient

a**x**b + b**x**a + c**x**d + d**x**c

For example, if:

expr = x**a**x**b + x**c**y**d

is a function on variables x and y then

NCGrad[expr, x, y]

returns the nc gradient list

{a**x**b + b**x**a + c**y**d, d**x**c}

IMPORTANT: The expression returned by NCGrad is the transpose or the adjoint of the standard gradient. This is done so that no assumption on the symbols are needed. The calculated expression is correct even if symbols are self-adjoint or symmetric.

See also: NCDirectionalD.

10.7.3 NCHessian

`NCHessian[expr, {var1, h1}, ...]` takes the second directional derivative of nc expression `expr` with respect to variables `var1`, `var2`, … successively in the directions `h1`, `h2`, ….

For example, if:

```
expr = y**inv[x]**y + x***a**x
```

then

```
NCHessian[expr, {x,h}, {y,s}]
```

returns

```
2 h**a**h + 2 s**inv[x]**s - 2 s**inv[x]**h**inv[x]**y -
2 y**inv[x]**h**inv[x]**s + 2 y**inv[x]**h**inv[x]**h**inv[x]**y
```

In the case of more than one variables `NCHessian[expr, {x,h}, {y,k}]` takes the second directional derivative of `expr` with respect to `x` in the direction `h` and with respect to `y` in the direction `k`.

See also: NCDiretionalD, NCGrad.

10.7.4 DirectionalD

`DirectionalD[expr,var,h]` takes the directional derivative of nc expression `expr` with respect to the single variable `var` in direction `h`.

DEPRECATION NOTICE: This syntax is limited to one variable and is being deprecated in favor of the more general syntax in NCDirectionalD.

See also: NCDirectionalD.

10.7.5 NCIntegrate

`NCIntegrate[expr,{var1,h1},...]` attempts to calculate the nc antiderivative of nc expression `expr` with respect to the single variable `var` in direction `h`.

For example:

```
NCIntegrate[x**h+h**x, {x,h}]
```

returns

```
x**x
```

See also: NCDirectionalD.

10.8 NCConvexity

NCConvexity is a package that provides functionality to determine whether a rational or polynomial noncommutative function is convex.

Members are:

- NCIndependent
- NCConvexityRegion

10.8.1 NCIndependent

`NCIndependent[list]` attempts to determine whether the nc entries of `list` are independent.

Entries of `NCIndependent` can be nc polynomials or nc rationals.

For example:

```
NCIndependent[{x,y,z}]
```

return *True* while

```
NCIndependent[{x,0,z}]
NCIndependent[{x,y,x}]
NCIndependent[{x,y,x+y}]
NCIndependent[{x,y,A x + B y}]
NCIndependent[{inv[1+x]**inv[x], inv[x], inv[1+x]}]
```

all return *False*.

See also: NCConvexityRegion.

10.8.2 NCConvexityRegion

`NCConvexityRegion[expr,vars]` is a function which can be used to determine whether the nc rational `expr` is convex in `vars` or not.

For example:

```
d = NCConvexityRegion[x**x**x, {x}];
```

returns

```
d = {2 x, -2 inv[x]}
```

from which we conclude that `x**x**x` is not convex in `x` because $x \succ 0$ and $-x^{-1} \succ 0$ cannot simultaneously hold.

`NCConvexityRegion` works by factoring the `NCHessian`, essentially calling:

```
hes = NCHessian[expr, {x, h}];
```

then

```
{lt, mq, rt} = NCMatrixOfQuadratic[hes, {h}]
```

to decompose the Hessian into a product of a left row vector, `lt`, times a middle matrix, `mq`, times a right column vector, `rt`. The middle matrix, `mq`, is factored using the `NCLDLDecomposition`:

```
{ldl, p, s, rank} = NCLDLDecomposition[mq];
{lf, d, rt} = GetLDUMatrices[ldl, s];
```

from which the output of NCConvexityRegion is the a list with the block-diagonal entries of the matrix `d`.

See also: NCHessian, NCMatrixOfQuadratic, NCLDLDecomposition.

Chapter 11

Packages for manipulating NC block matrices

11.1 NCDot

Members are:

- tpMat
- ajMat
- coMat
- NCDot
- NCInverse

11.1.1 tpMat

`tpMat[mat]` gives the transpose of matrix `mat` using `tp`.

See also: ajMat, coMat, NCDot.

11.1.2 ajMat

`ajMat[mat]` gives the adjoint transpose of matrix `mat` using `aj` instead of `ConjugateTranspose`.

See also: tpMat, coMat, NCDot.

11.1.3 coMat

`coMat[mat]` gives the conjugate of matrix `mat` using `co` instead of `Conjugate`.

See also: tpMat, ajMat, NCDot.

11.1.4 NCDot

`NCDot[mat1, mat2, ...]` gives the matrix multiplication of `mat1`, `mat2`, ... using `NonCommutativeMultiply` rather than `Times`.

Notes:

The experienced matrix analyst should always remember that the Mathematica convention for handling vectors is tricky.

- `{{1,2,4}}` is a 1x3 *matrix* or a *row vector*;

- `{{1},{2},{4}}` is a 3x1 *matrix* or a *column vector*;
- `{1,2,4}` is a *vector* but **not** a *matrix*. Indeed whether it is a row or column vector depends on the context. We advise not to use *vectors*.

See also: tpMat, ajMat, coMat.

11.1.5 NCInverse

`NCInverse[mat]` gives the nc inverse of the square matrix `mat`. `NCInverse` uses partial pivoting to find a nonzero pivot.

`NCInverse` is primarily used symbolically. Usually the elements of the inverse matrix are huge expressions. We recommend using `NCSimplifyRational` to improve the results.

See also: tpMat, ajMat, coMat.

11.2 NCMatrixDecompositions

`NCMatrixDecompositions` provide noncommutative versions of the linear algebra algorithms in the package MatrixDecompositions.

See the documentation for the package MatrixDecompositions for details on the algorithms and options.

Members are:

- Decompositions
  - NCLUDecompositionWithPartialPivoting
  - NCLUDecompositionWithCompletePivoting
  - NCLDLDecomposition
- Solvers
  - NCLowerTriangularSolve
  - NCUpperTriangularSolve
  - NCLUInverse
- Utilities
  - NCLUCompletePivoting
  - NCLUPartialPivoting
  - NCLeftDivide
  - NCRightDivide

11.2.1 NCLUDecompositionWithPartialPivoting

`NCLUDecompositionWithPartialPivoting` is a noncommutative version of NCLUDecompositionWithPartialPivoting.

The following `options` can be given:

- `ZeroTest` (`PossibleZeroQ`): function used to decide if a pivot is zero;
- `RightDivide` (NCRightDivide): function used to divide a vector by an entry;
- `Dot` (NCDot): function used to multiply vectors and matrices;
- `Pivoting` (NCLUPartialPivoting): function used to sort rows for pivoting;
- `SuppressPivoting` (`False`): whether to perform pivoting or not.

See also: LUDecompositionWithPartialPivoting.

11.2.2 NCLUDecompositionWithCompletePivoting

`NCLUDecompositionWithCompletePivoting` is a noncommutative version of NCLUDecompositionWithCompletePivoting.

The following `options` can be given:

- `ZeroTest` (`PossibleZeroQ`): function used to decide if a pivot is zero;
- `RightDivide` (NCRightDivide): function used to divide a vector by an entry;
- `Dot` (NCDot): function used to multiply vectors and matrices;
- `Pivoting` (NCLUCompletePivoting): function used to sort rows for pivoting;
- `SuppressPivoting` (`False`): whether to perform pivoting or not.

See also: LUDecompositionWithCompletePivoting.

11.2.3 NCLDLDecomposition

`NCLDLDecomposition` is a noncommutative version of LDLDecomposition.

The following `options` can be given:

- `ZeroTest` (`PossibleZeroQ`): function used to decide if a pivot is zero;
- `RightDivide` (NCRightDivide): function used to divide a vector by an entry on the right;
- `LeftDivide` (NCLeftDivide): function used to divide a vector by an entry on the left;
- `Dot` (`NCDot`): function used to multiply vectors and matrices;
- `CompletePivoting` (NCLUCompletePivoting): function used to sort rows for complete pivoting;
- `PartialPivoting` (NCLUPartialPivoting): function used to sort matrices for complete pivoting;
- `Inverse` (NCLUInverse): function used to invert 2x2 diagonal blocks;
- `SelfAdjointMatrixQ` (NCSelfAdjointQ): function to test if matrix is self-adjoint;
- `SuppressPivoting` (`False`): whether to perform pivoting or not.

See also: LUDecompositionWithCompletePivoting.

11.2.4 NCUpperTriangularSolve

`NCUpperTriangularSolve` is a noncommutative version of UpperTriangularSolve.

See also: UpperTriangularSolve.

11.2.5 NCLowerTriangularSolve

`NCLowerTriangularSolve` is a noncommutative version of LowerTriangularSolve.

See also: LowerTriangularSolve.

11.2.6 NCLUInverse

`NCLUInverse` is a noncommutative version of LUInverse.

See also: LUInverse.

11.2.7 NCLUPartialPivoting

`NCLUPartialPivoting` is a noncommutative version of LUPartialPivoting.

See also: LUPartialPivoting.

11.2.8 NCLUCompletePivoting

`NCLUCompletePivoting` is a noncommutative version of LUCompletePivoting.

See also: LUCompletePivoting.

11.2.9 NCLeftDivide

`NCLeftDivide[x,y]` divides each entry of the list `y` by `x` on the left.

For example:

```
NCLeftDivide[x, {a,b,c}]
```

returns

```
{inv[x]**a, inv[x]**b, inv[x]**c}
```

See also: NCRightDivide.

11.2.10 NCRightDivide

`NCRightDivide[x,y]` divides each entry of the list `x` by `y` on the right.

For example:

```
NCRightDivide[{a,b,c}, y]
```

returns

```
{a**inv[y], b**inv[y], c**inv[y]}
```

See also: NCLeftDivide.

11.3 MatrixDecompositions: linear algebra templates

`MatrixDecompositions` is a package that implements various linear algebra algorithms, such as *LU Decomposition* with *partial* and *complete pivoting*, and *LDL Decomposition*. The algorithms have been written with correctness and ease of customization rather than efficiency as the main goals. They were originally developed to serve as the core of the noncommutative linear algebra algorithms for NCAlgebra.

See the package NCMatrixDecompositions for noncommutative versions of these algorithms.

Members are:

- Decompositions
  - LUDecompositionWithPartialPivoting
  - LUDecompositionWithCompletePivoting
  - LDLDecomposition
  - LURowReduce
  - LURowReduceIncremental
- Solvers
  - LowerTriangularSolve
  - UpperTriangularSolve
  - LUInverse
- Utilities
  - GetLUMatrices
  - GetFullLUMatrices
  - GetLDUMatrices
  - GetFullLDUMatrices
  - GetDiagonal
  - LUPartialPivoting
  - LUCompletePivoting

11.3.1 LUDecompositionWithPartialPivoting

`LUDecompositionWithPartialPivoting[m]` generates a representation of the LU decomposition of the rectangular matrix `m`.

`LUDecompositionWithPartialPivoting[m, options]` uses `options`.

`LUDecompositionWithPartialPivoting` returns a list of two elements:

- the first element is a combination of upper- and lower-triangular matrices;
- the second element is a vector specifying rows used for pivoting.

`LUDecompositionWithPartialPivoting` is similar in functionality with the built-in `LUDecomposition`. It implements a *partial pivoting* strategy in which the sorting can be configured using the options listed below. It also applies to general rectangular matrices as well as square matrices.

The triangular factors are recovered using GetLUMatrices or GetFullLUMatrices.

The following `options` can be given:

- `ZeroTest` (`PossibleZeroQ`): function used to decide if a pivot is zero;
- `RightDivide` (`Divide`): function used to divide a vector by an entry;
- `Dot` (`Dot`): function used to multiply vectors and matrices;
- `Pivoting` (LUPartialPivoting): function used to sort rows for pivoting;
- `SuppressPivoting` (`False`): whether to perform pivoting or not.

See also: LUDecompositionWithPartialPivoting, LUDecompositionWithCompletePivoting, GetLUMatrices, GetFullLUMatrices, LUPartialPivoting.

11.3.2 LUDecompositionWithCompletePivoting

`LUDecompositionWithCompletePivoting[m]` generates a representation of the LU decomposition of the rectangular matrix `m`.

`LUDecompositionWithCompletePivoting[m, options]` uses `options`.

`LUDecompositionWithCompletePivoting` returns a list of four elements:

- the first element is a combination of upper- and lower-triangular matrices;
- the second element is a vector specifying rows used for pivoting;
- the third element is a vector specifying columns used for pivoting;
- the fourth element is the rank of the matrix.

`LUDecompositionWithCompletePivoting` implements a *complete pivoting* strategy in which the sorting can be configured using the options listed below. It also applies to general rectangular matrices as well as square matrices.

The triangular factors are recovered using GetLUMatrices or GetFullLUMatrices.

The following `options` can be given:

- `ZeroTest` (`PossibleZeroQ`): function used to decide if a pivot is zero;
- `Divide` (`Divide`): function used to divide a vector by an entry;
- `Dot` (`Dot`): function used to multiply vectors and matrices;
- `Pivoting` (LUCompletePivoting): function used to sort rows for pivoting;

See also: LUDecomposition, GetLUMatrices, GetFullLUMatrices, LUCompletePivoting, LUDecompositionWithPartialPivoting.

11.3.3 LDLDecomposition

`LDLDecomposition[m]` generates a representation of the LDL decomposition of the symmetric or self-adjoint matrix `m`.

`LDLDecomposition[m, options]` uses `options`.

`LDLDecomposition` returns a list of four elements:

- the first element is a combination of upper- and lower-triangular matrices;
- the second element is a vector specifying rows and columns used for pivoting;
- the third element is a vector specifying the size of the diagonal blocks (entries can be either 1 or 2);
- the fourth element is the rank of the matrix.

`LUDecompositionWithCompletePivoting` implements a *Bunch-Parlett pivoting* strategy in which the sorting can be configured using the options listed below. It applies only to square symmetric or self-adjoint matrices.

The triangular factors are recovered using GetLDUMatrices or GetFullLDUMatrices.

The following `options` can be given:

- `ZeroTest` (`PossibleZeroQ`): function used to decide if a pivot is zero;
- `RightDivide` (`Divide`): function used to divide a vector by an entry on the right;
- `LeftDivide` (`Divide`): function used to divide a vector by an entry on the left;
- `Dot` (`Dot`): function used to multiply vectors and matrices;
- `CompletePivoting` (LUCompletePivoting): function used to sort rows for complete pivoting;
- `PartialPivoting` (LUPartialPivoting): function used to sort matrices for complete pivoting;
- `Inverse` (`Inverse`): function used to invert 2x2 diagonal blocks;
- `SelfAdjointMatrixQ` (HermitianQ): function to test if matrix is self-adjoint;
- `SuppressPivoting` (`False`): whether to perform pivoting or not.

See also: LUDecompositionWithPartialPivoting, LUDecompositionWithCompletePivoting, GetLUMatrices, GetFullLDUMatrices, LUCompletePivoting, LUPartialPivoting.

11.3.4 UpperTriangularSolve

`UpperTriangularSolve[u, b]` solves the upper-triangular system of equations $ux = b$ using back-substitution.

For example:

```
x = UpperTriangularSolve[u, b];
```

returns the solution `x`.

See also: LUDecompositionWithPartialPivoting, LUDecompositionWithCompletePivoting, LDLDecomposition.

11.3.5 LowerTriangularSolve

`LowerTriangularSolve[l, b]` solves the lower-triangular system of equations $lx = b$ using forward-substitution.

For example:

```
x = LowerTriangularSolve[l, b];
```

returns the solution `x`.

See also: LUDecompositionWithPartialPivoting, LUDecompositionWithCompletePivoting, LDLDecomposition.

11.3.6 LUInverse

`LUInverse[a]` calculates the inverse of matrix `a`.

`LUInverse` uses the LUDecompositionWithPartialPivoting and the triangular solvers LowerTriangularSolve and UpperTriangularSolve.

See also: LUDecompositionWithPartialPivoting.

11.3.7 GetLUMatrices

`GetLUMatrices[m]` extracts lower- and upper-triangular blocks produced by `LDUDecompositionWithPartialPivoting` and `LDUDecompositionWithCompletePivoting`.

For example:

```
{lu, p} = LUDecompositionWithPartialPivoting[A];
{l, u} = GetLUMatrices[lu];
```

returns the lower-triangular factor `l` and upper-triangular factor `u` as `SparseArray`s.

See also: LUDecompositionWithPartialPivoting, LUDecompositionWithCompletePivoting, GetFullLUMatrices.

11.3.8 GetFullLUMatrices

`GetFullLUMatrices[m]` extracts lower- and upper-triangular blocks produced by `LDUDecompositionWithPartialPivoting` and `LDUDecompositionWithCompletePivoting`.

For example:

```
{lu, p} = LUDecompositionWithPartialPivoting[A];
{l, u} = GetFullLUMatrices[lu];
```

returns the lower-triangular factor `l` and upper-triangular factor `u`.

See also: LUDecompositionWithPartialPivoting, LUDecompositionWithCompletePivoting, GetLUMatrices.

11.3.9 GetLDUMatrices

`GetLDUMatrices[m,s]` extracts lower-, upper-triangular and diagonal blocks produced by `LDLDecomposition`.

For example:

```
{ldl, p, s, rank} = LDLDecomposition[A];
{l,d,u} = GetLDUMatrices[ldl,s];
```

returns the lower-triangular factor `l`, the upper-triangular factor `u`, and the block-diagonal factor `d` as `SparseArray`s.

See also: LDLDecomposition, GetFullLDUMatrices.

11.3.10 GetFullLDUMatrices

`GetLDUMatrices[m,s]` extracts lower-, upper-triangular and diagonal blocks produced by `LDLDecomposition`.

For example:

```
{ldl, p, s, rank} = LDLDecomposition[A];
{l,d,u} = GetLDUMatrices[ldl,s];
```

returns the lower-triangular factor `l`, the upper-triangular factor `u`, and the block-diagonal factor `d`.

See also: LDLDecomposition, GetLDUMatrices.

11.3.11 GetDiagonal

`GetDiagonal[m]` extracts the diagonal entries of matrix `m`.

`GetDiagonal[m, s]` extracts the block-diagonal entries of matrix `m` with block size `s`.

For example:

```
d = GetDiagonal[{{1,-1,0},{-1,2,0},{0,0,3}}];
```

returns

```
d = {1,2,3}
```

and

```
d = GetDiagonal[{{1,-1,0},{-1,2,0},{0,0,3}}, {2,1}];
```

returns

```
d = {{{1,-1},{-1,2}},3}
```

See also: LDLDecomposition.

11.3.12 LUPartialPivoting

`LUPartialPivoting[v]` returns the index of the element with largest absolute value in the vector `v`. If `v` is a matrix, it returns the index of the element with largest absolute value in the first column.

`LUPartialPivoting[v, f]` sorts with respect to the function `f` instead of the absolute value.

See also: LUDecompositionWithPartialPivoting, LUCompletePivoting.

11.3.13 LUCompletePivoting

`LUCompletePivoting[m]` returns the row and column index of the element with largest absolute value in the matrix `m`.

`LUCompletePivoting[v, f]` sorts with respect to the function `f` instead of the absolute value.

See also: LUDecompositionWithCompletePivoting, LUPartialPivoting.

11.3.14 LURowReduce

11.3.15 LURowReduceIncremental

Chapter 12

Packages for pretty output, testing, and utilities

12.1 NCOutput

NCOutput is a package that can be used to beautify the display of noncommutative expressions. NCOutput does not alter the internal representation of nc expressions, just the way they are displayed on the screen.

Members are:

- NCSetOutput

12.1.1 NCSetOutput

`NCSetOutput[options]` controls the display of expressions in a special format without affecting the internal representation of the expression.

The following `options` can be given:

- `NonCommutativeMultiply` (`False`): If `True` `x**y` is displayed as '$x \bullet y$';
- `tp` (`True`): If `True` `tp[x]` is displayed as 'x^T';
- `inv` (`True`): If `True` `inv[x]` is displayed as 'x^{-1}';
- `aj` (`True`): If `True` `aj[x]` is displayed as 'x^*';
- `co` (`True`): If `True` `co[x]` is displayed as '$\bar{x}$';
- `rt` (`True`): If `True` `rt[x]` is displayed as '$x^{1/2}$'.
- `All`: Set all available options to `True` or `False`.

See also: NCTex, NCTexForm.

12.2 NCTeX

Members are:

- NCTeX
- NCRunDVIPS
- NCRunLaTeX
- NCRunPDFLaTeX
- NCRunPDFViewer
- NCRunPS2PDF

12.2.1 NCTeX

`NCTeX[expr]` typesets the LaTeX version of `expr` produced with TeXForm or NCTeXForm using LaTeX.

12.2.2 NCRunDVIPS

`NCRunDVIPS[file]` run dvips on `file`. Produces a ps output.

12.2.3 NCRunLaTeX

`NCRunLaTeX[file]` typesets the LaTeX `file` with latex. Produces a dvi output.

12.2.4 NCRunPDFLaTeX

`NCRunLaTeX[file]` typesets the LaTeX `file` with pdflatex. Produces a pdf output.

12.2.5 NCRunPDFViewer

`NCRunPDFViewer[file]` display pdf `file`.

12.2.6 NCRunPS2PDF

`NCRunPS2PDF[file]` run pd2pdf on `file`. Produces a pdf output.

12.3 NCTeXForm

Members are:

- NCTeXForm
- NCTeXFormSetStarStar

12.3.1 NCTeXForm

`NCTeXForm[expr]` prints a LaTeX version of `expr`.

The format is compatible with AMS-LaTeX.

Should work better than the Mathematica `TeXForm` :)

12.3.2 NCTeXFormSetStarStar

`NCTeXFormSetStarStar[string]` replaces the standard '**' for `string` in noncommutative multiplications.

For example:

`NCTeXFormSetStarStar["."]`

uses a dot (`.`) to replace `NonCommutativeMultiply(**)`.

See also: NCTeXFormSetStar.

12.3.3 NCTeXFormSetStar

`NCTeXFormSetStar[string]` replaces the standard '*' for `string` in noncommutative multiplications.

For example:

`NCTeXFormSetStar[" "]`

uses a space () to replace `Times(*)`.

NCTeXFormSetStarStar.

12.4 NCRun

Members are:

- NCRun

12.4.1 NCRun

`NCRun[command]` is a replacement for the built-in `Run` command that gives a bit more control over the execution process.

`NCRun[command, options]` uses `options`.

The following `options` can be given:

- `Verbose` (`True`): print information on command being run;
- `CommandPrefix` (`""`): prefix to `command`;

See also: `Run`.

12.5 NCTest

These are commands for automatically testing if our algorithms produce the correct answer. Problems and answers are stored under the directory `NC/TESTING`.

Members are:

- NCTest
- NCTestCheck
- NCTestRun
- NCTestSummarize

12.5.1 NCTest

`NCTest[expr,answer]` asserts whether `expr` is equal to `answer`. The result of the test is collected when `NCTest` is run from `NCTestRun`.

See also: NCTestCheck, NCTestRun, NCTestSummarize.

12.5.2 NCTestCheck

`NCTestCheck[expr,messages]` evaluates `expr` and asserts that the messages in `messages` have been issued. The result of the test is collected when `NCTest` is run from `NCTestRun`.

`NCTestCheck[expr,answer,messages]` also asserts whether `expr` is equal to `answer`.

`NCTestCheck[expr,answer,messages,quiet]` quiets messages in `quiet`.

See also: NCTest, NCTestRun, NCTestSummarize.

12.5.3 NCTestRun

`NCTest[list]` runs the test files listed in `list` after appending the '.NCTest' suffix and return the results.

For example:

```
results = NCTestRun[{"NCCollect", "NCSylvester"}]
```

will run the test files "NCCollect.NCTest" and "NCSylvester.NCTest" and return the results in `results`.

See also: NCTest, NCTestCheck, NCTestSummarize.

12.5.4 NCTestSummarize

`NCTestSummarize[results]` will print a summary of the results in `results` as produced by `NCTestRun`.

See also: NCTestRun.

12.6 NCDebug

Members are:

- NCDebug

12.6.1 NCDebug

`NCDebug[level, message]` prints the objects `message` if level is higher than the current `DebugLevel` option.

Use `SetOptions[NCDebug, DebugLevel -> level]` to set up the current debug level.

Available options are:

- `DebugLevel` (0): current debug level;
- `DebugLogFile` (`$Ouput`): current file to which messages are printed.

12.7 NCUtil

NCUtil is a package with a collection of utilities used throughout NCAlgebra.

Members are:

- NCConsistentQ
- NCGrabFunctions
- NCGrabSymbols
- NCGrabIndeterminants
- NCVariables
- NCConsolidateList
- NCLeafCount
- NCReplaceData
- NCToExpression

12.7.1 NCConsistentQ

`NCConsistentQ[expr]` returns *True* is `expr` contains no commutative products or inverses involving noncommutative variables.

12.7.2 NCGrabFunctions

`NCGrabFunctions[expr]` returns a list with all fragments of `expr` containing functions.

`NCGrabFunctions[expr,f]` returns a list with all fragments of `expr` containing the function `f`.

For example:

```
NCGrabFunctions[inv[x] + tp[y]**inv[1+inv[1+tp[x]**y]], inv]
```

returns

```
{inv[1+inv[1+tp[x]**y]], inv[1+tp[x]**y], inv[x]}
```

and

```
NCGrabFunctions[inv[x] + tp[y]**inv[1+inv[1+tp[x]**y]]]
```

returns

```
{inv[1+inv[1+tp[x]**y]], inv[1+tp[x]**y], inv[x], tp[x], tp[y]}
```

See also: NCGrabSymbols.

12.7.3 NCGrabSymbols

`NCGrabSymbols[expr]` returns a list with all *Symbols* appearing in `expr`.

`NCGrabSymbols[expr,f]` returns a list with all *Symbols* appearing in `expr` as the single argument of function `f`.

For example:

```
NCGrabSymbols[inv[x] + y**inv[1+inv[1+x**y]]]
```

returns `{x,y}` and

```
NCGrabSymbols[inv[x] + y**inv[1+inv[1+x**y]], inv]
```

returns `{inv[x]}`.

See also: NCGrabFunctions.

12.7.4 NCGrabIndeterminants

`NCGrabIndeterminants[expr]` returns a list with first level symbols and nc expressions involved in sums and nc products in `expr`.

For example:

```
NCGrabIndeterminants[y - inv[x] + tp[y]**inv[1+inv[1+tp[x]**y]]]
```

returns

```
{y, inv[x], inv[1 + inv[1 + tp[x] ** y]], tp[y]}
```

See also: NCGrabFunctions, NCGrabSymbols.

12.7.5 NCVariables

`NCVariables[expr]` gives a list of all independent nc variables in the expression `expr`.

For example:

```
NCVariables[B + A y ** x ** y - 2 x]
```

returns

```
{x,y}
```

See also: NCGrabSymbols.

12.7.6 NCConsolidateList

`NCConsolidateList[list]` produces two lists:

- The first list contains a version of `list` where repeated entries have been suppressed;
- The second list contains the indices of the elements in the first list that recover the original `list`.

For example:

```
{list,index} = NCConsolidateList[{z,t,s,f,d,f,z}];
```

results in:

```
list = {z,t,s,f,d};
index = {1,2,3,4,5,4,1};
```

See also: `Union`

12.7.7 NCLeafCount

`NCLeafCount[expr]` returns an number associated with the complexity of an expression:

- If `PossibleZeroQ[expr] == True` then `NCLeafCount[expr]` is `-Infinity`;
- If `NumberQ[expr]] == True` then `NCLeafCount[expr]` is `Abs[expr]`;
- Otherwise `NCLeafCount[expr]` is `-LeafCount[expr]`;

`NCLeafCount` is `Listable`.

See also: `LeafCount`.

12.7.8 NCReplaceData

`NCReplaceData[expr, rules]` applies `rules` to `expr` and convert resulting expression to standard Mathematica, for example replacing `**` by `.`.

`NCReplaceData` does not attempt to resize entries in expressions involving matrices. Use `NCToExpression` for that.

See also: NCToExpression.

12.7.9 NCToExpression

`NCToExpression[expr, rules]` applies `rules` to `expr` and convert resulting expression to standard Mathematica.

`NCToExpression` attempts to resize entries in expressions involving matrices.

See also: NCReplaceData.

Chapter 13

Data structures for fast calculations

This chapter describes packages that handle special data structures that enable fast calculations in Mathematica.

13.1 NCPoly

13.1.1 Efficient storage of NC polynomials with rational coefficients

Members are:

- Constructors
  - NCPoly
  - NCPolyMonomial
  - NCPolyConstant
  - NCPolyConvert
  - NCPolyFromCoefficientArray
  - NCPolyFromGramMatrix
- Access and utilities
  - NCPolyMonomialQ
  - NCPolyDegree
  - NCPolyPartialDegree
  - NCPolyMonomialDegree
  - NCPolyNumberOfVariables
  - NCPolyNumberOfTerms
  - NCPolyCoefficient
  - NCPolyCoefficientArray
  - NCPolyGramMatrix
  - NCPolyGetCoefficients
  - NCPolyGetDigits
  - NCPolyGetIntegers
  - NCPolyLeadingMonomial
  - NCPolyLeadingTerm
  - NCPolyOrderType
  - NCPolyToRule
  - NCPolyTermsOfDegree
  - NCPolyTermsOfTotalDegree
  - NCPolyQuadraticTerms
  - NCPolyQuadraticChipset
  - NCPolyReverseMonomials

  - NCPolyGetOptions
- Formatting
  - NCPolyDisplay
  - NCPolyDisplayOrder
- Arithmetic
  - NCPolyDivideDigits
  - NCPolyDivideLeading
  - NCPolyFullReduce
  - NCPolyNormalize
  - NCPolyProduct
  - NCPolyQuotientExpand
  - NCPolyReduce
  - NCPolySum
- State space realization
  - NCPolyHankelMatrix
  - NCPolyRealization (#NCPolyRealization)
- Auxiliary functions
  - NCPolyVarsToIntegers
  - NCFromDigits
  - NCIntegerDigits
  - NCIntegerReverse
  - NCDigitsToIndex
  - NCPadAndMatch

13.1.2 Ways to represent NC polynomials

13.1.2.1 NCPoly

`NCPoly[coeff, monomials, vars]` constructs a noncommutative polynomial object in variables `vars` where the monomials have coefficient `coeff`.

Monomials are specified in terms of the symbols in the list `vars` as in NCPolyMonomial.

For example:

```
vars = {x,y,z};
poly = NCPoly[{-1, 2}, {{x,y,x}, {z}}, vars];
```

constructs an object associated with the noncommutative polynomial $2z - xyx$ in variables `x`, `y` and `z`.

The internal representation varies with the implementation but it is so that the terms are sorted according to a degree-lexicographic order in `vars`. In the above example, `x < y < z`.

The construction:

```
vars = {{x},{y,z}};
poly = NCPoly[{-1, 2}, {{x,y,x}, {z}}, vars];
```

represents the same polyomial in a graded degree-lexicographic order in `vars`, in this example, `x << y < z`.

See also: NCPolyMonomial, NCIntegerDigits, NCFromDigits.

13.1.2.2 NCPolyMonomial

`NCPolyMonomial[monomial, vars]` constructs a noncommutative monomial object in variables `vars`.

Monic monomials are specified in terms of the symbols in the list `vars`, for example:

```
vars = {x,y,z};
mon = NCPolyMonomial[{x,y,x},vars];
```

returns an `NCPoly` object encoding the monomial xyx in noncommutative variables `x`,`y`, and `z`. The actual representation of `mon` varies with the implementation.

Monomials can also be specified implicitly using indices, for example:

```
mon = NCPolyMonomial[{0,1,0}, 3];
```

also returns an `NCPoly` object encoding the monomial xyx in noncommutative variables `x`,`y`, and `z`.

If graded ordering is supported then

```
vars = {{x},{y,z}};
mon = NCPolyMonomial[{x,y,x},vars];
```

or

```
mon = NCPolyMonomial[{0,1,0}, {1,2}];
```

construct the same monomial xyx in noncommutative variables `x`,`y`, and `z` this time using a graded order in which `x << y < z`.

There is also an alternative syntax for `NCPolyMonomial` that allows users to input the monomial along with a coefficient using rules and the output of NCFromDigits. For example:

```
mon = NCPolyMonomial[{3, 3} -> -2, 3];
```

or

```
mon = NCPolyMonomial[NCFromDigits[{0,1,0}, 3] -> -2, 3];
```

represent the monomial $-2xyx$ that has coefficient `-2`.

See also: NCPoly, NCIntegerDigits, NCFromDigits.

13.1.2.3 NCPolyConstant

`NCPolyConstant[value, vars]` constructs a noncommutative monomial object in variables `vars` representing the constant `value`.

For example:

```
NCPolyConstant[3, {x, y, z}]
```

constructs an object associated with the constant `3` in variables `x`, `y` and `z`.

See also: NCPoly, NCPolyMonomial.

13.1.2.4 NCPolyConvert

`NCPolyConvert[poly, vars]` convert NCPoly `poly` to the ordering implied by `vars`.

For example, if

```
vars1 = {{x, y, z}};
coeff = {1, 2, 3, -1, -2, -3, 1/2};
mon = {{}, {x}, {z}, {x, y}, {x, y, x, x}, {z, x}, {z, z, z, z}};
poly1 = NCPoly[coeff, mon, vars1];
```

with respect to the ordering

$x \ll y \ll z$

then

```
vars2 = {{x},{y,z}};
poly2 = NCPolyConvert[poly, vars];
```

is the same polynomial as `poly1` but in the ordering

$x \ll y < z$

See also: NCPoly, NCPolyCoefficient.

13.1.2.5 NCPolyFromCoefficientArray

`NCPolyFromCoefficientArray[mat, vars]` returns an `NCPoly` constructed from the coefficient array `mat` in variables `vars`.

For example, for `mat` equal to the `SparseArray` corresponding to the rules:

```
{{1} -> 1, {2} -> 2, {6} -> -1, {50} -> -2, {4} -> 3, {11} -> -3, {121} -> 1/2}
```

the commands

```
vars = {{x},{y,z}};
NCPolyFromCoefficientArray[mat, vars]
```

return

```
NCPoly[{1, 2}, <|{0, 0, 0} -> 1, {0, 1, 0} -> 2, {1, 0, 2} -> 3, {1, 1, 1} -> -1,
       {1, 1, 6} -> -3, {1, 3, 9} -> -2, {4, 0, 80} -> 1/2|>]
```

See also: NCPolyCoefficientArray, NCPolyCoefficient.

13.1.2.6 NCPolyFromGramMatrix

`NCPolyFromGramMatrix[mat, vars]` returns an `NCPoly` constructed from the Gram matrix `mat` in variables `vars`.

For example, for `mat` equal to the `SparseArray` corresponding to the rules:

```
{{1, 1} -> 1, {2, 1} -> 2, {2, 3} -> -1, {4, 1} -> 3, {4, 2} -> -3, {6, 5} -> -2, {13, 13} -> 1/2}
```

the commands

```
vars = {{x},{y,z}};
NCPolyFromCoefficientArray[mat, vars]
```

return

```
NCPoly[{1, 2}, <|{0, 0, 0} -> 1, {0, 1, 0} -> 2, {1, 0, 2} -> 3, {1, 1, 1} -> -1,
       {1, 1, 6} -> -3, {1, 3, 9} -> -2, {4, 0, 80} -> 1/2|>]
```

See also: NCPolyGramMatrix.

13.1.3 Access and utlity functions

13.1.3.1 NCPolyMonomialQ

`NCPolyMonomialQ[poly]` returns `True` if `poly` is a `NCPoly` monomial.

See also: NCPoly, NCPolyMonomial.

13.1.3.2 NCPolyDegree

`NCPolyDegree[poly]` returns the degree of the nc polynomial `poly`.

See also: NCPolyPartialDegree

13.1.3.3 NCPolyPartialDegree

`NCPolyPartialDegree[poly]` returns the maximum degree appearing in the monomials of the nc polynomial `poly`.

See also: NCPolyDegree

13.1.3.4 NCPolyMonomialDegree

`NCPolyMonomialDegree[poly]` returns the partial degree of each symbol appearing in the monomials of the nc polynomial `poly`.

See also: NCPolyDegree

13.1.3.5 NCPolyNumberOfVariables

`NCPolyNumberOfVariables[poly]` returns the number of variables of the nc polynomial `poly`.

13.1.3.6 NCPolyNumberOfTerms

`NCPolyNumberOfTerms[poly]` returns the number of terms of the nc polynomial `poly`.

13.1.3.7 NCPolyCoefficient

`NCPolyCoefficient[poly, mon]` returns the coefficient of the monomial `mon` in the nc polynomial `poly`.

For example, in:

```
coeff = {1, 2, 3, -1, -2, -3, 1/2};
mon = {{}, {x}, {z}, {x, y}, {x, y, x, x}, {z, x}, {z, z, z, z}};
vars = {x,y,z};
poly = NCPoly[coeff, mon, vars];

c = NCPolyCoefficient[poly, NCPolyMonomial[{x,y},vars]];
```

returns

```
c = -1
```

See also: NCPoly, NCPolyMonomial.

13.1.3.8 NCPolyCoefficientArray

`NCPolyCoefficientArray[poly]` returns a coefficient array corresponding to the monomials in the nc polynomial `poly`.

For example:

```
coeff = {1, 2, 3, -1, -2, -3, 1/2};
mon = {{}, {x}, {z}, {x, y}, {x, y, x, x}, {z, x}, {z, z, z, z}};
vars = {x,y,z};
poly = NCPoly[coeff, mon, vars];

mat = NCPolyCoefficient[poly];
```

returns `mat` as a `SparseArray` corresponding to the rules:

```
{{1} -> 1, {2} -> 2, {6} -> -1, {50} -> -2, {4} -> 3, {11} -> -3, {121} -> 1/2}
```

See also: NCPolyFromCoefficientArray, NCPolyCoefficient.

13.1.3.9 NCPolyGramMatrix

`NCPolyGramMatrix[poly]` returns a Gram matrix corresponding to the monomials in the nc polynomial `poly`.

For example:

```
coeff = {1, 2, 3, -1, -2, -3, 1/2};
mon = {{}, {x}, {z}, {x, y}, {x, y, x, x}, {z, x}, {z, z, z, z}};
vars = {x,y,z};
poly = NCPoly[coeff, mon, vars];

mat = NCPolyGramMatrix[poly];
```

returns `mat` as a `SparseArray` corresponding to the rules:

```
{{1, 1} -> 1, {2, 1} -> 2, {2, 3} -> -1, {4, 1} -> 3, {4, 2} -> -3, {6, 5} -> -2, {13, 13} -> 1/2}
```

See also: NCPolyFromGramMatrix.

13.1.3.10 NCPolyGetCoefficients

`NCPolyGetCoefficients[poly]` returns a list with the coefficients of the monomials in the nc polynomial `poly`.

For example:

```
vars = {x,y,z};
poly = NCPoly[{-1, 2}, {{x,y,x}, {z}}, vars];
coeffs = NCPolyGetCoefficients[poly];
```

returns

```
coeffs = {2,-1}
```

The coefficients are returned according to the current graded degree-lexicographic ordering, in this example `x < y < z`.

See also: NCPolyGetDigits, NCPolyCoefficient, NCPoly.

13.1.3.11 NCPolyGetDigits

`NCPolyGetDigits[poly]` returns a list with the digits that encode the monomials in the nc polynomial `poly` as produced by NCIntegerDigits.

For example:

```
vars = {x,y,z};
poly = NCPoly[{-1, 2}, {{x,y,x}, {z}}, vars];
digits = NCPolyGetDigits[poly];
```

returns

```
digits = {{2}, {0,1,0}}
```

The digits are returned according to the current ordering, in this example `x < y < z`.

See also: NCPolyGetCoefficients, NCPoly.

13.1.3.12 NCPolyGetIntegers

`NCPolyGetIntegers[poly]` returns a list with the digits that encode the monomials in the nc polynomial `poly` as produced by NCFromDigits.

For example:

```
vars = {x,y,z};
poly = NCPoly[{-1, 2}, {{x,y,x}, {z}}, vars];
digits = NCPolyGetIntegers[poly];
```

returns

```
digits = {{1,2}, {3,3}}
```

The digits are returned according to the current ordering, in this example `x < y < z`.

See also: NCPolyGetCoefficients, NCPoly.

13.1.3.13 NCPolyLeadingMonomial

`NCPolyLeadingMonomial[poly]` returns an `NCPoly` representing the leading term of the nc polynomial `poly`.

For example:

```
vars = {x,y,z};
poly = NCPoly[{-1, 2}, {{x,y,x}, {z}}, vars];
lead = NCPolyLeadingMonomial[poly];
```

returns an `NCPoly` representing the monomial xyx. The leading monomial is computed according to the current ordering, in this example `x < y < z`. The actual representation of `lead` varies with the implementation.

See also: NCPolyLeadingTerm, NCPolyMonomial, NCPoly.

13.1.3.14 NCPolyLeadingTerm

`NCPolyLeadingTerm[poly]` returns a rule associated with the leading term of the nc polynomial `poly` as understood by NCPolyMonomial.

For example:

```
vars = {x,y,z};
poly = NCPoly[{-1, 2}, {{x,y,x}, {z}}, vars];
lead = NCPolyLeadingTerm[poly];
```

returns

```
lead = {3,3} -> -1
```

representing the monomial $-xyx$. The leading monomial is computed according to the current ordering, in this example `x < y < z`.

See also: NCPolyLeadingMonomial, NCPolyMonomial, NCPoly.

13.1.3.15 NCPolyOrderType

`NCPolyOrderType[poly]` returns the type of monomial order in which the nc polynomial `poly` is stored. Order can be `NCPolyGradedDegLex` or `NCPolyDegLex`.

See also: NCPoly,

13.1.3.16 NCPolyToRule

`NCPolyToRule[poly]` returns a `Rule` associated with polynomial `poly`. If `poly = lead + rest`, where `lead` is the leading term in the current order, then `NCPolyToRule[poly]` returns the rule `lead -> -rest` where the coefficient of the leading term has been normalized to `1`.

For example:

```
vars = {x, y, z};
poly = NCPoly[{-1, 2, 3}, {{x, y, x}, {z}, {x, y}}, vars];
rule = NCPolyToRule[poly]
```

returns the rule `lead -> rest` where `lead` represents is the nc monomial xyx and `rest` is the nc polynomial $2z + 3xy$

See also: NCPolyLeadingTerm, NCPolyLeadingMonomial, NCPoly.

13.1.3.17 NCPolyTermsOfDegree

`NCPolyTermsOfDegree[p, d]` returns a polynomial `p` in which only the monomials with degree `d` are present. The degree `d` is a list with the partial degrees on each variable.

For example:

```
vars = {x, y};
poly = NCPoly[{1, 2, 3, 4}, {{x}, {x, y}, {y, x}, {x, x}}, vars];
```

corresponds to the polynomial $x + 2xy + 3yx + 4x^2$ and

```
NCPolyTermsOfTotalDegree[p, {1,1}]
```

returns

```
NCPoly[{1, 1}, {1, 1, 1} -> 2, {1, 1, 2} -> 3|>]
```

which corresponds to the polyomial $2xy + 3yx$. Likewise

```
NCPolyTermsOfTotalDegree[p, {2,0}]
```

returns

```
NCPoly[{1, 1}, {0, 2, 0} -> 4|>]
```

which corresponds to the polyomial $4x^2$.

See also: NCPolyTermsOfTotalDegree.

13.1.3.18 NCPolyTermsOfTotalDegree

`NCPolyTermsOfTotalDegree[p, d]` returns a polynomial `p` in which only the monomials with total degree `d` are present. The degree `d` is an integer.

For example:

```
vars = {x, y};
poly = NCPoly[{1, 2, 3, 4}, {{x}, {x, y}, {y, x}, {x, x}}, vars];
```

corresponds to the polynomial $x + 2xy + 3yx + 4x^2$ and

```
NCPolyTermsOfTotalDegree[p, 2]
```

returns

```
NCPoly[{1, 1}, <|{0, 2, 0} -> 4, {1, 1, 1} -> 2, {1, 1, 2} -> 3|>]
```

which corresponds to the polyomial $2xy + 3yx + 4x^2$.

See also: NCPolyTermsOfDegree.

13.1.3.19 NCPolyQuadraticTerms

`NCPolyQuadraticTerms[p]` returns a polynomial with only the "square" quadratic terms of `p`.

For example:

```
vars = {{x, y, z}}
coeff = {1, 1, 4, 3, 2, -1}
digits = {{}, {x}, {y, x}, {z, x}, {z, y}, {x, z, z, y}}
p = NCPoly[coeff, digits, vars, TransposePairs -> {{x, y}}]
```

corresponds to the polynomial $p(x, y, z) = -x.z.z.y + 2z.y + 3z.x + 4y.x + x + 1$ in which x and y are transposes of each other, that is $y = x^T$. Its `NCPoly` object is

```
NCPoly[{3}, <|{0, 0} -> 1, {1, 0} -> 1, {2, 3} -> 4, {2, 6} -> 3, {2, 7} -> 2, {4, 25} -> -1|>, Transpos
```

A call to

```
NCPolyQuadraticTerms[p]
```

results in

```
NCPoly[{3}, <|{0, 0} -> 1, {2, 3} -> 4, {4, 25} -> -1|>, TransposePairs -> {{0, 1}}]
```

corresponding to the polynomial $-x.z.z.y + 4y.x + 1$ which contains only "square" quadratic terms of $p(x, y, z)$.

See also: `NCPolyQuadraticChipset`(#NCPolyQuadraticChipset).

13.1.3.20 NCPolyQuadraticChipset

`NCPolyQuadraticChipset[p]` returns a polynomial with only the "half" terms of `p` that can be in an NC SOS decomposition of `p`.

For example:

```
vars = {{x, y, z}}
coeff = {1, 1, 4, 3, 2, -1}
digits = {{}, {x}, {y, x}, {z, x}, {z, y}, {x, z, z, y}}
p = NCPoly[coeff, digits, vars, TransposePairs -> {{x, y}}]
```

corresponds to the polynomial $p(x, y, z) = -x.z.z.y + 2z.y + 3z.x + 4y.x + x + 1$ in which x and y are transposes of each other, that is $y = x^T$. Its `NCPoly` object is

```
NCPoly[{3}, <|{0, 0} -> 1, {1, 0} -> 1, {2, 3} -> 4, {2, 6} -> 3, {2, 7} -> 2, {4, 25} -> -1|>, Transpos
```

A call to

```
NCPolyQuadraticChipset[p]
```

results in

```
NCPoly[{3}, <|{0, 0} -> 1, {1, 0} -> 1, {1, 1} -> 1, {2, 2} -> 1|>, TransposePairs -> {{0, 1}}]
```

corresponding to the polynomial $x.z + y + x + 1$ that contains only terms which contain monomials with the "left half" of the monomials of $p(x, y, z)$ which can appear in an NC SOS decomposition of `p`.

See also: `NCPolyQuadraticTerms`(#NCPolyQuadraticTerms).

13.1.3.21 NCPolyReverseMonomials

`NCPolyReverseMonomials[p]` reverses the order of the symbols appearing in each monomial of the polynomial `p`.

For example:

```
vars = {x, y};
poly = NCPoly[{1, 2, 3, 4}, {{x}, {x, y}, {y, x}, {x, x}}, vars];
```

corresponds to the polynomial $x + 2xy + 3yx + 4x^2$ and

```
NCPolyReverseMonomials[p]
```

returns

```
NCPoly[{1, 1}, <|{0, 1, 0} -> 1, {0, 2, 0} -> 4, {1, 1, 2} -> 2, {1, 1, 1} -> 3|>]
```

which correspond to the polynomial $x + 2yx + 3xy + 4x^2$.

See also: NCIntegerReverse.

13.1.3.22 NCPolyGetOptions

`NCPolyGetOptions[p]` returns the options embedded in the polynomial `p`.

Available options are:

- `TransposePairs`: list with pairs of variables to be treated as transposes of each other;
- `SelfAdjointPairs`: list with pairs of variables to be treated as adjoints of each other.

13.1.4 Formating functions

13.1.4.1 NCPolyDisplay

`NCPolyDisplay[poly]` prints the noncommutative polynomial `poly`.

`NCPolyDisplay[poly, vars]` uses the symbols in the list `vars`.

13.1.4.2 NCPolyDisplayOrder

`NCPolyDisplayOrder[vars]` prints the order implied by the list of variables `vars`.

13.1.5 Arithmetic functions

13.1.5.1 NCPolyDivideDigits

`NCPolyDivideDigits[F,G]` returns the result of the division of the leading digits lf and lg.

13.1.5.2 NCPolyDivideLeading

`NCPolyDivideLeading[lF,lG,base]` returns the result of the division of the leading Rules lf and lg as returned by NCGetLeadingTerm.

13.1.5.3 NCPolyFullReduce

`NCPolyFullReduce[f,g]` applies NCPolyReduce successively until the remainder does not change.

See also: NCPolyReduce, NCPolyQuotientExpand.

13.1.5.4 NCPolyNormalize

`NCPolyNormalize[poly]` makes the coefficient of the leading term of `p` to unit. It also works when `poly` is a list.

13.1.5.5 NCPolyProduct

`NCPolyProduct[f,g]` returns a NCPoly that is the product of the NCPoly's f and g.

13.1.5.6 NCPolyQuotientExpand

`NCPolyQuotientExpand[q,g]` returns a NCPoly that is the left-right product of the quotient as returned by NCPolyReduce by the NCPoly g. It also works when g is a list.

13.1.5.7 NCPolyReduce

`NCPolyReduce[polys, rules]` reduces the list of `NCPolys` `polys` with respect to the list of `NCPolys` `rules`. The substitutions implied by `rules` are applied repeatedly to the polynomials in the `polys` until no further reduction occurs.

`NCPolyReduce[polys]` reduces each polynomial in the list of `NCPolys` `polys` with respect to the remaining elements of the list of polyomials `polys` until no further reduction occurs.

By default, `NCPolyReduce` only reduces the leading monomial in the current order. Use the optional boolean flag `complete` to completely reduce all monomials. For example, `NCPolyReduce[polys, rules, True]` and `NCPolyReduce[polys, True]`.

See also: NCPolyGroebner.

13.1.5.8 NCPolySum

`NCPolySum[f,g]` returns a NCPoly that is the sum of the NCPoly's f and g.

13.1.6 State space realization functions

13.1.6.1 NCPolyHankelMatrix

`NCPolyHankelMatrix[poly]` produces the nc *Hankel matrix* associated with the polynomial `poly` and also their shifts per variable.

For example:

```
vars = {{x, y}};
poly = NCPoly[{1, -1}, {{x, y}, {y, x}}, vars];
{H, Hx, Hy} = NCPolyHankelMatrix[poly]
```

results in the matrices

```
H =  {{  0,  0,  0,  1, -1 },
      {  0,  0,  1,  0,  0 },
      {  0, -1,  0,  0,  0 },
      {  1,  0,  0,  0,  0 },
      { -1,  0,  0,  0,  0 }}
Hx = {{  0,  0,  1,  0,  0 },
      {  0,  0,  0,  0,  0 },
      { -1,  0,  0,  0,  0 },
      {  0,  0,  0,  0,  0 },
      {  0,  0,  0,  0,  0 }}
Hy = {{  0, -1,  0,  0,  0 },
      {  1,  0,  0,  0,  0 },
      {  0,  0,  0,  0,  0 },
      {  0,  0,  0,  0,  0 },
      {  0,  0,  0,  0,  0 }}
```

which are the Hankel matrices associated with the commutator $xy - yx$.

See also: NCPolyRealization, NCDigitsToIndex.

13.1.6.2 NCPolyRealization

`NCPolyRealization[poly]` calculate a minimal descriptor realization for the polynomial `poly`.

`NCPolyRealization` uses `NCPolyHankelMatrix` and the resulting realization is compatible with the format used by `NCRational`.

For example:

```
vars = {{x, y}};
poly = NCPoly[{1, -1}, {{x, y}, {y, x}}, vars];
{{a0,ax,ay},b,c,d} = NCPolyRealization[poly]
```

produces a list of matrices `{a0,ax,ay}`, a column vector `b` and a row vector `c`, and a scalar `d` such that $c.inv[a0 + ax\,x + ay\,y].b + d = xy - yx$.

See also: NCPolyHankelMatrix, NCRational.

13.1.7 Auxiliary functions

13.1.7.1 NCPolyVarsToIntegers

`NCPolyVarsToIntegers[vars]` converts the list of symbols `vars` into a list of integers corresponding to the graded ordering implied by `vars`.

For example:

```
NCPolyVarsToIntegers[{{x},{y,z}}]
```

returns `{1,2}`, indicating that there are a total of three variables with the last two ranking higher than the first.

`NCPolyVarsToIntegers` raises `NCPoly::InvalidList` in case it cannot correctly parse the list of variables.

If `vars` is a list of integers, `NCPolyVarsToIntegers` returns this list intact.

See also: NCPoly.

13.1.7.2 NCFromDigits

`NCFromDigits[list, b]` constructs a representation of a monomial in `b` encoded by the elements of `list` where the digits are in base `b`.

`NCFromDigits[{list1,list2}, b]` applies `NCFromDigits` to each `list1`, `list2`, ...

List of integers are used to codify monomials. For example the list `{0,1}` represents a monomial xy and the list `{1,0}` represents the monomial yx. The call

```
NCFromDigits[{0,0,0,1}, 2]
```

returns

```
{4,1}
```

in which `4` is the degree of the monomial $xxxy$ and `1` is `0001` in base `2`. Likewise

```
NCFromDigits[{0,2,1,1}, 3]
```

returns

```
{4,22}
```

in which `4` is the degree of the monomial $xzyy$ and `22` is `0211` in base `3`.

If `b` is a list, then degree is also a list with the partial degrees of each letters appearing in the monomial. For example:

`NCFromDigits[{0,2,1,1}, {1,2}]`

returns

`{3, 1, 22}`

in which `3` is the partial degree of the monomial *xzyy* with respect to letters `y` and `z`, `1` is the partial degree with respect to letter `x` and `22` is `0211` in base `3 = 1 + 2`.

This construction is used to represent graded degree-lexicographic orderings.

See also: NCIntegerDigits.

13.1.7.3 NCIntegerDigits

`NCIntegerDigits[n,b]` is the inverse of the `NCFromDigits`.

`NCIntegerDigits[{list1,list2}, b]` applies `NCIntegerDigits` to each `list1`, `list2`, ….

For example:

`NCIntegerDigits[{4,1}, 2]`

returns

`{0,0,0,1}`

in which `4` is the degree of the monomial `x**x**x**y` and `1` is `0001` in base `2`. Likewise

`NCIntegerDigits[{4,22}, 3]`

returns

`{0,2,1,1}`

in which `4` is the degree of the monomial `x**z**y**y` and `22` is `0211` in base `3`.

If `b` is a list, then degree is also a list with the partial degrees of each letters appearing in the monomial. For example:

`NCIntegerDigits[{3, 1, 22}, {1,2}]`

returns

`{0,2,1,1}`

in which `3` is the partial degree of the monomial `x**z**y**y` with respect to letters `y` and `z`, `1` is the partial degree with respect to letter `x` and `22` is `0211` in base `3 = 1 + 2`.

See also: NCFromDigits.

13.1.7.4 NCIntegerReverse

`NCIntegerReverse[n,b]` reverses the integer `n` on the base `b` as returned by `NCFromDigits`.

`NCIntegerReverse[{list1,list2}, b]` applies `NCIntegerReverse` to each `list1`, `list2`, ….

For example:

`NCIntegerReverse[{4,1}, 2]`

in which `{4,1}` correspond to the digits `{1,0,0,0}` returns

`{4, 8}`

which correspond to the digits `{1,0,0,0}`.

See also: NCIntegerDigits.

13.1.7.5 NCDigitsToIndex

`NCDigitsToIndex[digits, b]` returns the index that the monomial represented by `digits` in the base `b` would occupy in the standard monomial basis.

`NCDigitsToIndex[{digit1,digits2}, b]` applies `NCDigitsToIndex` to each `digit1`, `digit2`, ...

`NCDigitsToIndex[digits, b, Reverse -> True]` returns the index as occupied by the permuted digits.

`NCDigitsToIndex` returns the same index for graded or simple basis.

For example:

```
digits = {0, 1};
NCDigitsToIndex[digits, 2]
NCDigitsToIndex[digits, {2}]
NCDigitsToIndex[digits, {1, 1}]
```

all return

```
5
```

which is the index of the monomial xy in the standard monomial basis of polynomials in x and y. Likewise

```
digits = {{}, {1}, {0, 1}, {0, 2, 1, 1}};
NCDigitsToIndex[digits, 2]
```

returns

```
{1,3,5,27}
```

Finally

```
NCDigitsToIndex[{0, 1}, 2, Reverse -> True]
```

returns `6` instead of `5`.

See also: NCFromDigits, NCIntegerDigits.

13.1.7.6 NCPadAndMatch

When list `a` is longer than list `b`, `NCPadAndMatch[a,b]` returns the minimum number of elements from list a that should be added to the left and right of list `b` so that `a = l b r`. When list `b` is longer than list `a`, return the opposite match.

`NCPadAndMatch` returns all possible matches with the minimum number of elements.

13.2 NCPolyInterface

The package `NCPolyInterface` provides a basic interface between `NCPoly` and `NCAlgebra`. Note that to take full advantage of the speed-up possible with `NCPoly` one should always convert and manipulate `NCPoly` expressions before converting back to `NCAlgebra`.

Members are:

- NCToNCPoly
- NCPolyToNC
- NCRuleToPoly
- NCMonomialList
- NCCoefficientRules
- NCCoefficientList
- NCCoefficientQ
- NCMonomialQ

- NCPolynomialQ

13.2.1 NCToNCPoly

`NCToNCPoly[expr, var]` constructs a noncommutative polynomial object in variables `var` from the nc expression `expr`.

For example

```
NCToNCPoly[x**y - 2 y**z, {x, y, z}]
```

constructs an object associated with the noncommutative polynomial $xy - 2yz$ in variables `x`, `y` and `z`. The internal representation is so that the terms are sorted according to a degree-lexicographic order in `vars`. In the above example, $x < y < z$.

13.2.2 NCPolyToNC

`NCPolyToNC[poly, vars]` constructs an nc expression from the noncommutative polynomial object `poly` in variables `vars`. Monomials are specified in terms of the symbols in the list `var`.

For example

```
poly = NCToNCPoly[x**y - 2 y**z, {x, y, z}];
expr = NCPolyToNC[poly, {x, y, z}];
```

returns

```
expr = x**y - 2 y**z
```

See also: NCPolyToNC, NCPoly.

13.2.3 NCRuleToPoly

`NCRuleToPoly[a -> b]` converts the rule `a -> b` into the relation `a - b`.

For instance:

```
NCRuleToPoly[x**y**y -> x**y - 1]
```

returns

```
x**y**y - x**y + 1
```

13.2.4 NCMonomialList

`NCMonomialList[poly]` gives the list of all monomials in the polynomial `poly`.

For example:

```
vars = {x, y}
expr = B + A y ** x ** y - 2 x
NCMonomialList[expr, vars]
```

returns

```
{1, x, y ** x ** y}
```

See also: NCCoefficientRules, NCCoefficientList, NCVariables.

13.2.5 NCCoefficientRules

`NCCoefficientRules[poly]` gives a list of rules between all the monomials polynomial `poly`.

For example:

```
vars = {x, y}
expr = B + A y ** x ** y - 2 x
NCCoefficientRules[expr, vars]
```

returns

```
{1 -> B, x -> -2, y ** x ** y -> A}
```

See also: NCMonomialList, NCCoefficientRules, NCVariables.

13.2.6 NCCoefficientList

`NCCoefficientList[poly]` gives the list of all coefficients in the polynomial `poly`.

For example:

```
vars = {x, y}
expr = B + A y ** x ** y - 2 x
NCCoefficientList[expr, vars]
```

returns

```
{B, -2, A}
```

See also: NCMonomialList, NCCoefficientRules, NCVariables.

13.2.7 NCCoefficientQ

`NCCoefficientQ[expr]` returns True if `expr` is a valid polynomial coefficient.

For example:

```
SetCommutative[A]
NCCoefficientQ[1]
NCCoefficientQ[A]
NCCoefficientQ[2 A]
```

all return `True` and

```
SetNonCommutative[x]
NCCoefficientQ[x]
NCCoefficientQ[x**x]
NCCoefficientQ[Exp[x]]
```

all return `False`.

IMPORTANT: `NCCoefficientQ[expr]` does not expand `expr`. This means that `NCCoefficientQ[2 (A + 1)]` will return `False`.

See also: NCMonomialQ, NCPolynomialQ

13.2.8 NCMonomialQ

`NCCoefficientQ[expr]` returns True if `expr` is an nc monomial.

For example:

```
SetCommutative[A]
NCMonomialQ[1]
NCMonomialQ[x]
NCMonomialQ[A x ** y]
NCMonomialQ[2 A x ** y ** x]
```

all return `True` and

```
NCMonomialQ[x + x ** y]
```

returns `False`.

IMPORTANT: `NCMonomialQ[expr]` does not expand `expr`. This means that `NCMonomialQ[2 (A + 1) x**x]` will return `False`.

See also: NCCoefficientQ, NCPolynomialQ

13.2.9 NCPolynomialQ

`NCPolynomialQ[expr]` returns True if `expr` is an nc polynomial with commutative coefficients.

For example:

```
NCPolynomialQ[A x ** y]
```

all return `True` and

```
NCMonomialQ[x + x ** y]
```

returns `False`.

IMPORTANT: `NCPolynomialQ[expr]` does expand `expr`. This means that `NCPolynomialQ[(x + y)^3]` will return `True`.

See also: NCCoefficientQ, NCMonomialQ

13.3 NCPolynomial

13.3.1 Efficient storage of NC polynomials with nc coefficients

This package contains functionality to convert an nc polynomial expression into an expanded efficient representation that can have commutative or noncommutative coefficients.

For example the polynomial

```
exp = a**x**b - 2 x**y**c**x + a**c
```

in variables `x` and `y` can be converted into an NCPolynomial using

```
p = NCToNCPolynomial[exp, {x,y}]
```

which returns

```
p = NCPolynomial[a**c, <|{x}->{{1,a,b}},{x**y,x}->{{2,1,c,1}}|>, {x,y}]
```

Members are:

- Constructors
  - NCPolynomial
  - NCToNCPolynomial
  - NCPolynomialToNC
  - NCRationalToNCPolynomial
- Access and utilities
  - NCPCoefficients

  - NCPTermsOfDegree
  - NCPTermsOfTotalDegree
  - NCPTermsToNC
  - NCPDecompose
  - NCPDegree
  - NCPMonomialDegree
  - NCPCompatibleQ
  - NCPSameVariablesQ
  - NCPMatrixQ
  - NCPLinearQ
  - NCPQuadraticQ
  - NCPNormalize
- Arithmetic
  - NCPTimes
  - NCPDot
  - NCPPlus
  - NCPSort

13.3.2 Ways to represent NC polynomials

13.3.2.1 NCPolynomial

`NCPolynomial[indep,rules,vars]` is an expanded efficient representation for an nc polynomial in `vars` which can have commutative or noncommutative coefficients.

The nc expression `indep` collects all terms that are independent of the letters in `vars`.

The *Association* `rules` stores terms in the following format:

```
{mon1, ..., monN} -> {scalar, term1, ..., termN+1}
```

where:

- `mon1, ..., monN`: are nc monomials in vars;
- `scalar`: contains all commutative coefficients; and
- `term1, ..., termN+1`: are nc expressions on letters other than the ones in vars which are typically the noncommutative coefficients of the polynomial.

`vars` is a list of *Symbols*.

For example the polynomial

```
a**x**b - 2 x**y**c**x + a**c
```

in variables `x` and `y` is stored as:

```
NCPolynomial[a**c, <|{x}->{{1,a,b}},{x**y,x}->{{2,1,c,1}}|>, {x,y}]
```

NCPolynomial specific functions are prefixed with NCP, e.g. NCPDegree.

See also: `NCToNCPolynomial`, `NCPolynomialToNC`, `NCPTermsToNC`.

13.3.2.2 NCToNCPolynomial

`NCToNCPolynomial[p, vars]` generates a representation of the noncommutative polynomial `p` in `vars` which can have commutative or noncommutative coefficients.

`NCToNCPolynomial[p]` generates an `NCPolynomial` in all nc variables appearing in `p`.

Example:

```
exp = a**x**b - 2 x**y**c**x + a**c
p = NCToNCPolynomial[exp, {x,y}]
```

returns

```
NCPolynomial[a**c, <|{x}->{{1,a,b}},{x**y,x}->{{2,1,c,1}}|>, {x,y}]
```

See also: NCPolynomial, NCPolynomialToNC.

13.3.2.3 NCPolynomialToNC

`NCPolynomialToNC[p]` converts the NCPolynomial `p` back into a regular nc polynomial.

See also: NCPolynomial, NCToNCPolynomial.

13.3.2.4 NCRationalToNCPolynomial

`NCRationalToNCPolynomial[r, vars]` generates a representation of the noncommutative rational expression `r` in `vars` which can have commutative or noncommutative coefficients.

`NCRationalToNCPolynomial[r]` generates an `NCPolynomial` in all nc variables appearing in `r`.

`NCRationalToNCPolynomial` creates one variable for each `inv` expression in `vars` appearing in the rational expression `r`. It returns a list of three elements:

- the first element is the `NCPolynomial`;
- the second element is the list of new variables created to replace `invs`;
- the third element is a list of rules that can be used to recover the original rational expression.

For example:

```
exp = a**inv[x]**y**b - 2 x**y**c**x + a**c
{p,rvars,rules} = NCRationalToNCPolynomial[exp, {x,y}]
```

returns

```
p = NCPolynomial[a**c, <|{rat1**y}->{{1,a,b}},{x**y,x}->{{2,1,c,1}}|>, {x,y,rat1}]
rvars = {rat1}
rules = {rat1->inv[x]}
```

See also: NCToNCPolynomial, NCPolynomialToNC.

13.3.3 Grouping terms by degree

13.3.3.1 NCPTermsOfDegree

`NCPTermsOfDegree[p,deg]` gives all terms of the NCPolynomial `p` of degree `deg`.

The degree `deg` is a list with the degree of each symbol.

For example:

```
p = NCPolynomial[0, <|{x,y}->{{2,a,b,c}},
                      {x,x}->{{1,a,b,c}},
                      {x**x}->{{-1,a,b}}|>, {x,y}]
NCPTermsOfDegree[p, {1,1}]
```

returns

```
<|{x,y}->{{2,a,b,c}}|>
```

and

```
NCPTermsOfDegree[p, {2,0}]
```

returns

```
<|{x,x}->{{1,a,b,c}}, {x**x}->{{-1,a,b}}|>
```

See also: `NCPTermsOfTotalDegree`,`NCPTermsToNC`.

13.3.3.2 NCPTermsOfTotalDegree

`NCPTermsOfDegree[p,deg]` gives all terms of the NCPolynomial `p` of total degree `deg`.

The degree `deg` is the total degree.

For example:

```
p = NCPolynomial[0, <|{x,y}->{{2,a,b,c}},
                       {x,x}->{{1,a,b,c}},
                       {x**x}->{{-1,a,b}}|>, {x,y}]
NCPTermsOfDegree[p, 2]
```

returns

```
<|{x,y}->{{2,a,b,c}},{x,x}->{{1,a,b,c}},{x**x}->{{-1,a,b}}|>
```

See also: `NCPTermsOfDegree`,`NCPTermsToNC`.

13.3.3.3 NCPTermsToNC

`NCPTermsToNC` gives a nc expression corresponding to terms produced by `NCPTermsOfDegree` or `NCPTermsOfTotalDegree`.

For example:

```
terms = <|{x,x}->{{1,a,b,c}}, {x**x}->{{-1,a,b}}|>
NCPTermsToNC[terms]
```

returns

```
a**x**b**c-a**x**b
```

See also: `NCPTermsOfDegree`,`NCPTermsOfTotalDegree`.

13.3.4 Utilities

13.3.4.1 NCPDegree

`NCPDegree[p]` gives the degree of the NCPolynomial `p`.

See also: `NCPMonomialDegree`.

13.3.4.2 NCPMonomialDegree

`NCPMonomialDegree[p]` gives the degree of each monomial in the NCPolynomial `p`.

See also: `NCDegree`.

13.3.4.3 NCPCoefficients

`NCPCoefficients[p, m]` gives all coefficients of the NCPolynomial `p` in the monomial `m`.

For example:

```
exp = a**x**b - 2 x**y**c**x + a**c + d**x
p = NCToNCPolynomial[exp, {x, y}]
NCPCoefficients[p, {x}]
```

returns

```
{{1, d, 1}, {1, a, b}}
```

and

```
NCPCoefficients[p, {x**y, x}]
```

returns

```
{{-2, 1, c, 1}}
```

See also: `NCPTermsToNC`.

13.3.4.4 NCPLinearQ

`NCPLinearQ[p]` gives True if the NCPolynomial `p` is linear.

See also: `NCPQuadraticQ`.

13.3.4.5 NCPQuadraticQ

`NCPQuadraticQ[p]` gives True if the NCPolynomial `p` is quadratic.

See also: `NCPLinearQ`.

13.3.4.6 NCPCompatibleQ

`NCPCompatibleQ[p1,p2,...]` returns *True* if the polynomials `p1`,`p2`,… have the same variables and dimensions.

See also: NCPSameVariablesQ, NCPMatrixQ.

13.3.4.7 NCPSameVariablesQ

`NCPSameVariablesQ[p1,p2,...]` returns *True* if the polynomials `p1`,`p2`,… have the same variables.

See also: NCPCompatibleQ, NCPMatrixQ.

13.3.4.8 NCPMatrixQ

`NCMatrixQ[p]` returns *True* if the polynomial `p` is a matrix polynomial.

See also: NCPCompatibleQ.

13.3.4.9 NCPNormalize

`NCPNormalizes[p]` gives a normalized version of NCPolynomial p where all factors that have free commutative products are collectd in the scalar.

This function is intended to be used mostly by developers.

See also: `NCPolynomial`

13.3.5 Operations on NC polynomials

13.3.5.1 NCPPlus

`NCPPlus[p1,p2,...]` gives the sum of the nc polynomials `p1`,`p2`,… .

13.3.5.2 NCPTimes

`NCPTimes[s,p]` gives the product of a commutative `s` times the nc polynomial `p`.

13.3.5.3 NCPDot

`NCPDot[p1,p2,...]` gives the product of the nc polynomials `p1`,`p2`,... .

13.3.5.4 NCPSort

`NCPSort[p]` gives a list of elements of the NCPolynomial `p` in which monomials are sorted first according to their degree then by Mathematica's implicit ordering.

For example

```
NCPSort[NCToNCPolynomial[c + x**x - 2 y, {x,y}]]
```

will produce the list

```
{c, -2 y, x**x}
```

See also: NCPDecompose, NCDecompose, NCCompose.

13.3.5.5 NCPDecompose

`NCPDecompose[p]` gives an association of elements of the NCPolynomial `p` in which elements of the same order are collected together.

For example

```
NCPDecompose[NCToNCPolynomial[a**x**b+c+d**x**e+a**x**e**x**b+a**x**y, {x,y}]]
```

will produce the Association

```
<|{1,0}->a**x**b + d**x**e, {1,1}->a**x**y, {2,0}->a**x**e**x**b, {0,0}->c|>
```

See also: NCPSort, NCDecompose, NCCompose.

13.4 NCQuadratic

NCQuadratic is a package that provides functionality to handle quadratic polynomials in NC variables.

Members are:

- NCToNCQuadratic
- NCPToNCQuadratic
- NCQuadraticToNC
- NCQuadraticToNCPolynomial
- NCMatrixOfQuadratic
- NCQuadraticMakeSymmetric

13.4.1 NCToNCQuadratic

`NCToNCQuadratic[p, vars]` is shorthand for

```
NCPToNCQuadratic[NCToNCPolynomial[p, vars]]
```

See also: NCToNCQuadratic,NCToNCPolynomial.

13.4.2 NCPToNCQuadratic

`NCPToNCQuadratic[p]` gives an expanded representation for the quadratic `NCPolynomial p`.

`NCPToNCQuadratic` returns a list with four elements:

- the first element is the independent term;
- the second represents the linear part as in `NCSylvester`;

- the third element is a list of left NC symbols;
- the fourth element is a numeric `SparseArray`;
- the fifth element is a list of right NC symbols.

Example:

```
exp = d + x + x**x + x**a**x + x**e**x + x**b**y**d + d**y**c**y**d;
vars = {x,y};
p = NCToNCPolynomial[exp, vars];
{p0,sylv,left,middle,right} = NCPToNCQuadratic[p];
```

produces

```
p0 = d
sylv = <|x->{{1},{1},SparseArray[{{1}}]}, y->{{},{},{}}|>
left =  {x,d**y}
middle = SparseArray[{{1+a+e,b},{0,c}}]
right = {x,y**d}
```

See also: NCSylvester,NCQuadraticToNCPolynomial,NCPolynomial.

13.4.3 NCQuadraticToNC

`NCQuadraticToNC[{const, lin, left, middle, right}]` is shorthand for

`NCPolynomialToNC[NCQuadraticToNCPolynomial[{const, lin, left, middle, right}]]`

See also: NCQuadraticToNCPolynomial,NCPolynomialToNC.

13.4.4 NCQuadraticToNCPolynomial

`NCQuadraticToNCPolynomial[rep]` takes the list `rep` produced by `NCPToNCQuadratic` and converts it back to an `NCPolynomial`.

`NCQuadraticToNCPolynomial[rep,options]` uses options.

The following options can be given:

- `Collect` (*True*): controls whether the coefficients of the resulting `NCPolynomial` are collected to produce the minimal possible number of terms.

See also: NCPToNCQuadratic, NCPolynomial.

13.4.5 NCMatrixOfQuadratic

`NCMatrixOfQuadratic[p, vars]` gives a factorization of the symmetric quadratic function `p` in noncommutative variables `vars` and their transposes.

`NCMatrixOfQuadratic` checks for symmetry and automatically sets variables to be symmetric if possible.

Internally it uses NCPToNCQuadratic and NCQuadraticMakeSymmetric.

It returns a list of three elements:

- the first is the left border row vector;
- the second is the middle matrix;
- the third is the right border column vector.

For example:

```
expr = x**y**x + z**x**x**z;
{left,middle,right}=NCMatrixOfQuadratics[expr, {x}];
```

returns:

```
left={x, z**x}
middle=SparseArray[{{y,0},{0,1}}]
right={x,x**z}
```

The answer from `NCMatrixOfQuadratics` always satisfies `p = NCDot[left,middle,right]`.

See also: NCPToNCQuadratic, NCQuadraticMakeSymmetric.

13.4.6 NCQuadraticMakeSymmetric

`NCQuadraticMakeSymmetric[{p0, sylv, left, middle, right}]` takes the output of `NCPToNCQuadratic` and produces, if possible, an equivalent symmetric representation in which `Map[tp, left] = right` and `middle` is a symmetric matrix.

See also: NCPToNCQuadratic.

13.5 NCSylvester

NCSylvester is a package that provides functionality to handle linear polynomials in NC variables.

Members are:

- NCToNCSylvester
- NCPToNCSylvester
- NCSylvesterToNC
- NCSylvesterToNCPolynomial

13.5.1 NCToNCSylvester

`NCToNCSylvester[p, vars]` is shorthand for

`NCPToNCSylvester[NCToNCPolynomial[p, vars]]`

See also: NCToNCSylvester, NCToNCPolynomial.

13.5.2 NCPToNCSylvester

`NCPToNCSylvester[p]` gives an expanded representation for the linear `NCPolynomial` `p`.

`NCPToNCSylvester` returns a list with two elements:

- the first is a the independent term;
- the second is an association where each key is one of the variables and each value is a list with three elements:
  - the first element is a list of left NC symbols;
  - the second element is a list of right NC symbols;
  - the third element is a numeric `SparseArray`.

Example:

```
p = NCToNCPolynomial[2 + a**x**b + c**x**d + y, {x,y}];
{p0,sylv} = NCPolynomialToNCSylvester[p]
```

produces

```
p0 = 2
sylv = <|x->{{a,c},{b,d},SparseArray[{{1,0},{0,1}}]},
         y->{{1},{1},SparseArray[{{1}}]}|>
```

See also: NCSylvesterToNCPolynomial, NCSylvesterToNC, NCToNCSylvester, NCPolynomial.

13.5.3 NCSylvesterToNC

`NCSylvesterToNC[{const, lin}]` is shorthand for

`NCPolynomialToNC[NCSylvesterToNCPolynomial[{const, lin}]]`

See also: NCSylvesterToNCPolynomial, NCPolynomialToNC.

13.5.4 NCSylvesterToNCPolynomial

`NCSylvesterToNCPolynomial[rep]` takes the list `rep` produced by `NCPToNCSylvester` and converts it back to an `NCPolynomial`.

`NCSylvesterToNCPolynomial[rep,options]` uses `options`.

The following `options` can be given: * `Collect` (*True*): controls whether the coefficients of the resulting NCPolynomial are collected to produce the minimal possible number of terms.

See also: NCPToNCSylvester, NCToNCSylvester, NCPolynomial.

Chapter 14

Noncommutative Gröbner Bases Algorithms

14.1 NCGBX

This is an interface to a Gröebner Bases code that runs purely under Mathematica. The actual algorithm is implemented in the package NCPolyGroebner. Its function names, inputs and outputs are very similar (but not always exactly the same) to the ones provided in the legacy package NCGB, which requires both Mathematica and auxiliary executables compiled from C++ to run. NCGBX may run slower on some medium size problems but will succeed on large size problems which might fail under NCGB.

Members are:

- SetMonomialOrder
- SetKnowns
- SetUnknowns
- ClearMonomialOrder
- GetMonomialOrder
- PrintMonomialOrder
- NCMakeGB
- NCProcess
- NCGBSimplifyRational
- NCReduce

14.1.1 SetMonomialOrder

`SetMonomialOrder[var1, var2, ...]` sets the current monomial order.

For example

`SetMonomialOrder[a,b,c]`

sets the lex order $a \ll b \ll c$.

If one uses a list of variables rather than a single variable as one of the arguments, then multigraded lex order is used. For example

`SetMonomialOrder[{a,b,c}]`

sets the graded lex order $a < b < c$.

Another example:

```
SetMonomialOrder[{{a, b}, {c}}]
```

or

```
SetMonomialOrder[{a, b}, c]
```

set the multigraded lex order $a < b \ll c$.

Finally

```
SetMonomialOrder[{a,b}, {c}, {d}]
```

or

```
SetMonomialOrder[{a,b}, c, d]
```

is equivalent to the following two commands

```
SetKnowns[a,b]
SetUnknowns[c,d]
```

There is also an older syntax which is still supported:

```
SetMonomialOrder[{a, b, c}, n]
```

sets the order of monomials to be $a < b < c$ and assigns them grading level `n`.

```
SetMonomialOrder[{a, b, c}, 1]
```

is equivalent to `SetMonomialOrder[{a, b, c}]`. When using this older syntax the user is responsible for calling ClearMonomialOrder to make sure that the current order is empty before starting.

See also: ClearMonomialOrder, GetMonomialOrder, PrintMonomialOrder, SetKnowns, SetUnknowns.

14.1.2 SetKnowns

`SetKnowns[var1, var2, ...]` records the variables `var1`, `var2`, ... to be corresponding to known quantities.

`SetUnknowns` and `Setknowns` prescribe a monomial order with the knowns at the the bottom and the unknowns at the top.

For example

```
SetKnowns[a,b]
SetUnknowns[c,d]
```

is equivalent to

```
SetMonomialOrder[{a,b}, {c}, {d}]
```

which corresponds to the order $a < b \ll c \ll d$ and

```
SetKnowns[a,b]
SetUnknowns[{c,d}]
```

is equivalent to

```
SetMonomialOrder[{a,b}, {c, d}]
```

which corresponds to the order $a < b \ll c < d$.

Note that `SetKnowns` flattens grading so that

```
SetKnowns[a,b]
```

and

```
SetKnowns[{a},{b}]
```

result both in the order $a < b$.

Successive calls to `SetUnknowns` and `SetKnowns` overwrite the previous knowns and unknowns. For example

```
SetKnowns[a,b]
SetUnknowns[c,d]
SetKnowns[c,d]
SetUnknowns[a,b]
```

results in an ordering $c < d \ll a \ll b$.

See also: SetUnknowns, SetMonomialOrder.

14.1.3 SetUnknowns

`SetUnknowns[var1, var2, ...]` records the variables `var1`, `var2`, ... to be corresponding to unknown quantities.

`SetUnknowns` and `SetKnowns` prescribe a monomial order with the knowns at the the bottom and the unknowns at the top.

For example

```
SetKnowns[a,b]
SetUnknowns[c,d]
```

is equivalent to

```
SetMonomialOrder[{a,b}, {c}, {d}]
```

which corresponds to the order $a < b \ll c \ll d$ and

```
SetKnowns[a,b]
SetUnknowns[{c,d}]
```

is equivalent to

```
SetMonomialOrder[{a,b}, {c, d}]
```

which corresponds to the order $a < b \ll c < d$.

Note that `SetKnowns` flattens grading so that

```
SetKnowns[a,b]
```

and

```
SetKnowns[{a},{b}]
```

result both in the order $a < b$.

Successive calls to `SetUnknowns` and `SetKnowns` overwrite the previous knowns and unknowns. For example

```
SetKnowns[a,b]
SetUnknowns[c,d]
SetKnowns[c,d]
SetUnknowns[a,b]
```

results in an ordering $c < d \ll a \ll b$.

See also: SetKnowns, SetMonomialOrder.

14.1.4 ClearMonomialOrder

`ClearMonomialOrder[]` clear the current monomial ordering.

It is only necessary to use `ClearMonomialOrder` if using the indexed version of `SetMonomialOrder`.

See also: SetKnowns, SetUnknowns, SetMonomialOrder, ClearMonomialOrder, PrintMonomialOrder.

14.1.5 GetMonomialOrder

`GetMonomialOrder[]` returns the current monomial ordering in the form of a list.

For example

```
SetMonomialOrder[{a,b}, {c}, {d}]
order = GetMonomialOrder[]
```

returns

```
order = {{a,b},{c},{d}}
```

See also: SetKnowns, SetUnknowns, SetMonomialOrder, ClearMonomialOrder, PrintMonomialOrder.

14.1.6 PrintMonomialOrder

`PrintMonomialOrder[]` prints the current monomial ordering.

For example

```
SetMonomialOrder[{a,b}, {c}, {d}]
PrintMonomialOrder[]
```

print $a < b \ll c \ll d$.

See also: SetKnowns, SetUnknowns, SetMonomialOrder, ClearMonomialOrder, PrintMonomialOrder.

14.1.7 NCMakeGB

`NCMakeGB[{poly1, poly2, ...}, k]` attempts to produces a nc Gröbner Basis (GB) associated with the list of nc polynomials `{poly1, poly2, ...}`. The GB algorithm proceeds through *at most* `k` iterations until a Gröbner basis is found for the given list of polynomials with respect to the order imposed by SetMonomialOrder.

If `NCMakeGB` terminates before finding a GB the message `NCMakeGB::Interrupted` is issued.

The output of `NCMakeGB` is a list of rules with left side of the rule being the *leading* monomial of the polynomials in the GB.

For example:

```
SetMonomialOrder[x];
gb = NCMakeGB[{x^2 - 1, x^3 - 1}, 20]
```

returns

```
gb = {x -> 1}
```

that corresponds to the polynomial $x - 1$, which is the nc Gröbner basis for the ideal generated by $x^2 - 1$ and $x^3 - 1$.

`NCMakeGB[{poly1, poly2, ...}, k, options]` uses `options`.

For example

```
gb = NCMakeGB[{x^2 - 1, x^3 - 1}, 20, RedudeBasis -> True]
```

runs the Gröbner basis algortihm and completely reduces the output set of polynomials.

The following `options` can be given:

- `ReduceBasis` (`False`): control whether the resulting basis output by the command is a reduced Gröbner basis at the completion of the algorithm. This corresponds to running `NCReduce` with the Boolean flag `True` to completely reduce the output basis. Can be set globally as `SetOptions[NCMakeGB, ReturnBasis -> True]`.
- `SimplifyObstructions` (`True`): control whether whether to remove obstructions before constructing more S-polynomials;
- `SortObstructions` (`False`): control whether obstructions are sorted before being processed;
- `SortBasis` (`False`): control whether initial basis is sorted before initiating algorithm;
- `VerboseLevel` (`1`): control level of verbosity from `0` (no messages) to `5` (very verbose);
- `PrintBasis` (`False`): if `True` prints current basis at each major iteration;
- `PrintObstructions` (`False`): if `True` prints current list of obstructions at each major iteration;
- `PrintSPolynomials` (`False`): if `True` prints every S-polynomial formed at each minor iteration.
- `ReturnRules` (`True`): if `True` rules representing relations in which the left-hand side is the leading monomial are returned instead of polynomials. Use `False` for backward compatibility. Can be set globally as `SetOptions[NCMakeGB, ReturnRules -> False]`.

`NCMakeGB` makes use of the algorithm `NCPolyGroebner` implemented in NCPolyGroebner.

See also: NCReduce, ClearMonomialOrder, GetMonomialOrder, PrintMonomialOrder, SetKnowns, SetUnknowns, NCPolyGroebner.

14.1.8 NCProcess

`NCProcess[{poly1, poly2, ...}, k]` finds a new generating set for the ideal generated by `{poly1, poly2, ...}` using NCMakeGB then produces an summary report on the findings.

Not all features of `NCProcess` in the old `NCGB` C++ version are supported yet.

See also: NCMakeGB.

14.1.9 NCGBSimplifyRational

`NCGBSimplifyRational[expr]` creates a set of relations for each rational expression and sub-expression found in `expr` which are used to produce simplification rules using NCMakeGB then replaced using NCReduce.

For example:

```
expr = x ** inv[1 - x] - inv[1 - x] ** x
NCGBSimplifyRational[expr]
```

or

```
expr = inv[1 - x - y ** inv[1 - x] ** y] - 1/2 (inv[1 - x + y] + inv[1 - x - y])
NCGBSimplifyRational[expr]
```

both result in `0`.

See also: NCMakeGB, NCReduce.

14.1.10 NCReduce

`NCReduce[polys, rules]` reduces the list of polynomials `polys` with respect to the list of polyomials `rules`. The substitutions implied by `rules` are applied repeatedly to the polynomials in the `polys` until no further reduction occurs.

`NCReduce[polys]` reduces each polynomial in the list of polynomials `polys` with respect to the remaining elements of the list of polyomials `polys` until no further reduction occurs.

By default, `NCReduce` only reduces the leading monomial in the current order. Use the optional boolean flag `complete` to completely reduce all monomials. For example, `NCReduce[polys, rules, True]` and `NCReduce[polys, True]`.

See also: NCMakeGB, NCGBSimplifyRational.

14.2 NCPolyGroebner

This packages implements a Gröebner Bases algorithm that runs purely under Mathematica. This algorithm is the one called by the user-friendly functions in the package NCGBX.

Members are:

- NCPolyGroebner

14.2.1 NCPolyGroebner

`NCPolyGroebner[G]` computes the noncommutative Groebner basis of the list of `NCPoly` polynomials `G`.

`NCPolyGroebner[G, options]` uses `options`.

The following `options` can be given:

- `SimplifyObstructions` (`True`) whether to remove obstructions before constructing more S-polynomials;
- `SortObstructions` (`False`) whether to sort obstructions using Mora's SUGAR ranking;
- `SortBasis` (`False`) whether to sort basis before starting algorithm;
- `Labels` (`{}`) list of labels to use in verbose printing;
- `VerboseLevel` (`1`): function used to decide if a pivot is zero;
- `PrintBasis` (`False`): function used to divide a vector by an entry;
- `PrintObstructions` (`False`);
- `PrintSPolynomials` (`False`);

The algorithm is based on [Mor94] and uses the terminology there.

See also: NCPoly.

14.3 NCGB

This packages supports our legacy Gröebner Bases algorithm that requires both Mathematica and auxiliary executables compiled from C++ to run. NCGBX may run slower on some medium size problems but will succeed on large size problems which might fail under NCGB.

This code has become hard to maintain and support and may be deprecated in the future. See older versions of the NC documentation for a complete description of its functionality.

Chapter 15

Semidefinite Programming Algorithms

15.1 NCSDP

NCSDP is a package that allows the symbolic manipulation and numeric solution of semidefinite programs.

Members are:

- NCSDP
- NCSDPForm
- NCSDPDual
- NCSDPDualForm

15.1.1 NCSDP

`NCSDP[inequalities,vars,obj,data]` converts the list of NC polynomials and NC matrices of polynomials `inequalities` that are linear in the unknowns listed in `vars` into the semidefinite program with linear objective `obj`. The semidefinite program (SDP) should be given in the following canonical form:

```
max  <obj, vars>  s.t.  inequalities <= 0.
```

It returns a list with two entries:

- The first is a list with the an instance of SDPSylvester;
- The second is a list of rules with properties of certain variables.

Both entries should be supplied to SDPSolve in order to numerically solve the semidefinite program. For example:

```
{abc, rules} = NCSDP[inequalities, vars, obj, data];
```

generates an instance of SDPSylvester that can be solved using:

```
<< SDPSylvester`
{Y, X, S, flags} = SDPSolve[abc, rules];
```

`NCSDP` uses the user supplied rules in `data` to set up the problem data.

`NCSDP[inequalities,vars,data]` converts problem into a feasibility semidefinite program.

`NCSDP[inequalities,vars,obj,data,options]` uses `options`.

The following `options` can be given:

- `DebugLevel` (`0`): control printing of debugging information.

See also: NCSDPForm, NCSDPDual, SDPSolve.

15.1.2 NCSDPForm

`NCSDPForm[[inequalities,vars,obj]` prints out a pretty formatted version of the SDP expressed by the list of NC polynomials and NC matrices of polynomials `inequalities` that are linear in the unknowns listed in `vars`.

See also: NCSDP, NCSDPDualForm.

15.1.3 NCSDPDual

`{dInequalities, dVars, dObj} = NCSDPDual[inequalities,vars,obj]` calculates the symbolic dual of the SDP expressed by the list of NC polynomials and NC matrices of polynomials `inequalities` that are linear in the unknowns listed in `vars` with linear objective `obj` into a dual semidefinite in the following canonical form:

```
max <dObj, dVars>  s.t.  dInequalities == 0,   dVars >= 0.
```

`{dInequalities, dVars, dObj} = NCSDPDual[inequalities,vars,obj,dualVars]` uses the symbols in `dualVars` as `dVars`.

`NCSDPDual[inequalities,vars,...,options]` uses `options`.

The following `options` can be given:

- `DualSymbol` (`"w"`): letter to be used as symbol for dual variable;
- `DebugLevel` (`0`): control printing of debugging information.

See also: NCSDPDualForm, NCSDP.

15.1.4 NCSDPDualForm

`NCSDPForm[[dInequalities,dVars,dObj]` prints out a pretty formatted version of the dual SDP expressed by the list of NC polynomials and NC matrices of polynomials `dInequalities` that are linear in the unknowns listed in `dVars` with linear objective `dObj`.

See also: NCSDPDual, NCSDPForm.

15.2 SDP

`SDP` is a package that provides data structures for the numeric solution of semidefinite programs of the form:

$$\begin{aligned} \max_{y,S} \quad & b^T y \\ \text{s.t.} \quad & Ay + S = c \\ & S \succeq 0 \end{aligned}$$

where S is a symmetric positive semidefinite matrix and y is a vector of decision variables.

See the package SDP for a potentially more efficient alternative to the basic implementation provided by this package.

Members are:

- SDPMatrices
- SDPSolve
- SDPEval

- SDPPrimalEval
- SDPDualEval
- SDPSylvesterEval

15.2.1 SDPMatrices

`SDPMatrices[f, G, y]` converts the symbolic linear functions `f`, `G` in the variables `y` associated to the semidefinite program:

$$\begin{aligned} \min_{y} \quad & f(y), \\ \text{s.t.} \quad & G(y) \succeq 0 \end{aligned}$$

into numerical data that can be used to solve an SDP in the form:

$$\begin{aligned} \max_{y,S} \quad & b^T y \\ \text{s.t.} \quad & Ay + S = c \\ & S \succeq 0 \end{aligned}$$

`SDPMatrices` returns a list with three entries:

- The first is the coefficient array `A`;
- The second is the coefficient array `b`;
- The third is the coefficient array `c`.

For example:

```
f = -x
G = {{1, x}, {x, 1}}
vars = {x}
{A,b,c} = SDPMatrices[f, G, vars]
```

results in

```
A = {{{{0, -1}, {-1, 0}}}}
b = {{{1}}}
c = {{{1, 0}, {0, 1}}}
```

All data is stored as `SparseArray`s.

See also: SDPSolve.

15.2.2 SDPSolve

`SDPSolve[{A,b,c}]` solves an SDP in the form:

$$\begin{aligned} \max_{y,S} \quad & b^T y \\ \text{s.t.} \quad & Ay + S = c \\ & S \succeq 0 \end{aligned}$$

`SDPSolve` returns a list with four entries:

- The first is the primal solution y;
- The second is the dual solution X;
- The third is the primal slack variable S;
- The fourth is a list of flags:

- `PrimalFeasible`: `True` if primal problem is feasible;
- `FeasibilityRadius`: less than one if primal problem is feasible;
- `PrimalFeasibilityMargin`: close to zero if primal problem is feasible;
- `DualFeasible`: `True` if dual problem is feasible;
- `DualFeasibilityRadius`: close to zero if dual problem is feasible.

For example:

```
{Y, X, S, flags} = SDPSolve[abc]
```

solves the SDP `abc`.

`SDPSolve[{A,b,c}, options]` uses `options`.

`options` are those of PrimalDual.

See also: SDPMatrices.

15.2.3 SDPEval

`SDPEval[A, y]` evaluates the linear function Ay in an `SDP`.

This is a convenient replacement for SDPPrimalEval in which the list `y` can be used directly.

See also: SDPPrimalEval, SDPDualEval, SDPSolve, SDPMatrices.

15.2.4 SDPPrimalEval

`SDPPrimalEval[A, {{y}}]` evaluates the linear function Ay in an `SDP`.

See SDPEval for a convenient replacement for `SDPPrimalEval` in which the list `y` can be used directly.

See also: SDPEval, SDPDualEval, SDPSolve, SDPMatrices.

15.2.5 SDPDualEval

`SDPDualEval[A, X]` evaluates the linear function A^*X in an `SDP`.

See also: SDPPrimalEval, SDPSolve, SDPMatrices.

15.2.6 SDPSylvesterEval

`SDPSylvesterEval[a, W]` returns a matrix representation of the Sylvester mapping $A^*(WA(\Delta_y)W)$ when applied to the scaling `W`.

`SDPSylvesterEval[a, Wl, Wr]` returns a matrix representation of the Sylvester mapping $A^*(W_l A(\Delta_y)W_r)$ when applied to the left- and right-scalings `Wl` and `Wr`.

See also: SDPPrimalEval, SDPDualEval.

15.3 SDPFlat

`SDPFlat` is a package that provides data structures for the numeric solution of semidefinite programs of the form:

$$\begin{aligned} \max_{y,S} \quad & b^T y \\ \text{s.t.} \quad & Ay + S = c \\ & S \succeq 0 \end{aligned}$$

where S is a symmetric positive semidefinite matrix and y is a vector of decision variables.

It is a potentially more efficient alternative to the basic implementation provided by the package SDP.

Members are:

- SDPFlatData
- SDPFlatPrimalEval
- SDPFlatDualEval
- SDPFlatSylvesterEval

15.3.1 SDPFlatData

`SDPFlatData[{a,b,c}]` converts the triplet `{a,b,c}` from the format of the package SDP to the `SDPFlat` format.

It returns a list with four entries:

- The first is the input array `a`;
- The second is its flattened version `AFlat`;
- The third is the flattened version of `c`, `cFlat`;
- The fourth is an array with the flattened dimensions.

See also: SDP.

15.3.2 SDPFlatPrimalEval

`SDPFlatPrimalEval[aFlat, y]` evaluates the linear function Ay in an `SDPFlat`.

See also: SDPFlatDualEval, SDPFlatSylvesterEval.

15.3.3 SDPFlatDualEval

`SDPFlatDualEval[aFlat, X]` evaluates the linear function A^*X in an `SDPFlat`.

See also: SDPFlatPrimalEval, SDPFlatSylvesterEval.

15.3.4 SDPFlatSylvesterEval

`SDPFlatSylvesterEval[a, aFlat, W]` returns a matrix representation of the Sylvester mapping $A^*(WA(\Delta_y)W)$ when applied to the scaling `W`.

`SDPFlatSylvesterEval[a, aFlat, Wl, Wr]` returns a matrix representation of the Sylvester mapping $A^*(W_l A(\Delta_y)W_r)$ when applied to the left- and right-scalings `Wl` and `Wr`.

See also: SDPFlatPrimalEval, SDPFlatDualEval.

15.4 SDPSylvester

`SDPSylvester` is a package that provides data structures for the numeric solution of semidefinite programs of the form:

$$\begin{aligned} \max_{y,S} \quad & \sum_i \operatorname{trace}(b_i^T y_i) \\ \text{s.t.} \quad & Ay + S = \frac{1}{2}\sum_i a_i y_i b_i + (a_i y_i b_i)^T + S = C \\ & S \succeq 0 \end{aligned}$$

where S is a symmetric positive semidefinite matrix and $y = \{y_1, \ldots, y_n\}$ is a list of matrix decision variables.

Members are:

- SDPEval
- SDPSylvesterPrimalEval

- SDPSylvesterDualEval
- SDPSylvesterSylvesterEval

15.4.1 SDPEval

`SDPEval[A, y]` evaluates the linear function $Ay = \frac{1}{2}\sum_i a_i y_i b_i + (a_i y_i b_i)^T$ in an `SDPSylvester`.

This is a convenient replacement for SDPSylvesterPrimalEval in which the list `y` can be used directly.

See also: SDPSylvesterPrimalEval, SDPSylvesterDualEval.

15.4.2 SDPSylvesterPrimalEval

`SDPSylvesterPrimalEval[a, y]` evaluates the linear function $Ay = \frac{1}{2}\sum_i a_i y_i b_i + (a_i y_i b_i)^T$ in an `SDPSylvester`.

See SDPSylvesterEval for a convenient replacement for `SDPPrimalEval` in which the list `y` can be used directly.

See also: SDPSylvesterDualEval, SDPSylvesterSylvesterEval.

15.4.3 SDPSylvesterDualEval

`SDPSylvesterDualEval[A, X]` evaluates the linear function $A^*X = \{b_1 X a_1, \cdots, b_n X a_n\}$ in an `SDPSylvester`.

For example

See also: SDPSylvesterPrimalEval, SDPSylvesterSylvesterEval.

15.4.4 SDPSylvesterSylvesterEval

`SDPSylvesterEval[a, W]` returns a matrix representation of the Sylvester mapping $A^*(WA(\Delta_y)W)$ when applied to the scaling `W`.

`SDPSylvesterEval[a, Wl, Wr]` returns a matrix representation of the Sylvester mapping $A^*(W_l A(\Delta_y) W_r)$ when applied to the left- and right-scalings `Wl` and `Wr`.

See also: SDPSylvesterPrimalEval, SDPSylvesterDualEval.

15.5 PrimalDual

`PrimalDual` provides an algorithm for solving a pair of primal-dual semidefinite programs in the form

$$\begin{aligned} \min_X \quad & \operatorname{trace}(cX) \\ \text{s.t.} \quad & A^*(X) = b \\ & X \succeq 0 \end{aligned} \tag{Primal}$$

$$\begin{aligned} \max_{y,S} \quad & b^T y \\ \text{s.t.} \quad & A(y) + S = c \\ & S \succeq 0 \end{aligned} \tag{Dual}$$

where X is the primal variable and (y, S) are the dual variables.

The algorithm is parametrized and users should provide their own means of evaluating the mappings A, A^* and also the Sylvester mapping

$$A^*(W_l A(\Delta_y) W_r)$$

used to solve the least-square subproblem.

Users can develop custom algorithms that can take advantage of special structure, as done for instance in NCSDP.

The algorithm constructs a feasible solution using the Self-Dual Embedding of [].

Members are:

- PrimalDual

15.5.1 PrimalDual

`PrimalDual[PrimalEval,DualEval,SylvesterEval,b,c]` solves the semidefinite program using a primal dual method.

`PrimalEval` should return the primal mapping $A^*(X)$ when applied to the current primal variable `X` as in `PrimalEval @@ X`.

`DualEval` should return the dual mapping $A(y)$ when applied to the current dual variable `y` as in `DualEval @@ y`.

`SylvesterVecEval` should return a matrix representation of the Sylvester mapping $A^*(W_l A(\Delta_y) W_r)$ when applied to the left- and right-scalings `Wl` and `Wr` as in `SylvesterVecEval @@ {Wl, Wr}`.

`PrimalDual[PrimalEval,DualEval,SylvesterEval,b,c,options]` uses `options`.

The following `options` can be given:

- `Method` (`PredictorCorrector`): choice of method for updating duality gap; possible options are `ShortStep`, `LongStep` and `PredictorCorrector`;
- `SearchDirection` (`NT`): choice of search direction to use; possible options are `NT` for Nesterov-Todd, `KSH` for HRVM/KSH/M, `KSHDual` for dual HRVM/KSH/M;
- `FeasibilityTol` (10^-3): tolerance used to assess feasibility;
- `GapTol` (10^-9): tolerance used to assess optimality;
- `MaxIter` (250): maximum number of iterations allowed;
- `SparseWeights` (`True`): whether weights should be converted to a `SparseArray`;
- `RationalizeIterates` (`False`): whether to rationalize iterates in an attempt to construct a rational solution;
- `SymmetricVariables` (`{}`): list of index of dual variables to be considered symmetric.
- `ScaleHessian` (`True`): whether to scale the least-squares subproblem coefficient matrix;
- `PrintSummary` (`True`): whether to print summary information;
- `PrintIterations` (`True`): whether to print progrees at each iteration;
- `DebugLevel` (0): whether to print debug information;
- `Profiling` (`False`): whether to print messages with detailed timing of steps.

Chapter 16

Work in Progress

Sections in this chapter describe experimental packages which are still under development.

16.1 NCPolySOS

Members are:

- NCPolySOS
- NCPolySOSToSDP (#NCPolySOSToSDP)

16.1.1 NCPolySOS

`NCPolySOS[p, var]` returns an `NCPoly` with symbolic coefficients on the variable `var` corresponding to a possible Gram representation of the polynomial `p`.

`NCPolySOS` uses NCPolyQuadraticChipset to generate a sparse Gram representation.

`NCPolySOS[p]` uses `q` as default symbol.

See also: NCPolySOSToSDP, NCPolyQuadraticChipset

16.1.2 NCPolySOSToSDP

`NCPolySOSToSDP[G, options]`

See also: NCPolySOS.

16.2 NCRational

This package contains functionality to convert an nc rational expression into a descriptor representation.

For example the rational

```
exp = 1 + inv[1 + x]
```

in variables `x` and `y` can be converted into an NCPolynomial using

```
p = NCToNCPolynomial[exp, {x,y}]
```

which returns

```
p = NCPolynomial[a**c, <|{x}->{{1,a,b}},{x**y,x}->{{2,1,c,1}}|>, {x,y}]
```

Members are:

- NCRational
- NCToNCRational
- NCRationalToNC
- NCRationalToCanonical
- CanonicalToNCRational
- NCROrder
- NCRLinearQ
- NCRStrictlyProperQ
- NCRPlus
- NCRTimes
- NCRTranspose
- NCRInverse
- NCRControllableSubspace
- NCRControllableRealization
- NCRObservableRealization
- NCRMinimalRealization

16.2.1 State-space realizations for NC rationals

16.2.1.1 NCRational

NCRational::usage

16.2.1.2 NCToNCRational

NCToNCRational::usage

16.2.1.3 NCRationalToNC

NCRationalToNC::usage

16.2.1.4 NCRationalToCanonical

NCRationalToCanonical::usage

16.2.1.5 CanonicalToNCRational

CanonicalToNCRational::usage

16.2.2 Utilities

16.2.2.1 NCROrder

NCROrder::usage

16.2.2.2 NCRLinearQ

NCRLinearQ::usage

16.2.2.3 NCRStrictlyProperQ

NCRStrictlyProperQ::usage

16.2.3 Operations on NC rationals

16.2.3.1 NCRPlus

NCRPlus::usage

16.2.3.2 NCRTimes

NCRTimes::usage

16.2.3.3 NCRTranspose

NCRTranspose::usage

16.2.3.4 NCRInverse

NCRInverse::usage

16.2.4 Minimal realizations

16.2.4.1 NCRControllableRealization

NCRControllableRealization::usage

16.2.4.2 NCRControllableSubspace

NCRControllableSubspace::usage

16.2.4.3 NCRObservableRealization

NCRObservableRealization::usage

16.2.4.4 NCRMinimalRealization

NCRMinimalRealization::usage

16.3 NCRealization

WARNING: OBSOLETE PACKAGE WILL BE REPLACED BY NCRational

The package **NCRealization** implements an algorithm due to N. Slinglend for producing minimal realizations of nc rational functions in many nc variables. See "Toward Making LMIs Automatically".

It actually computes formulas similar to those used in the paper "Noncommutative Convexity Arises From Linear Matrix Inequalities" by J William Helton, Scott A. McCullough, and Victor Vinnikov. In particular, there are functions for calculating (symmetric) minimal descriptor realizations of nc (symmetric) rational functions, and determinantal representations of polynomials.

Members are:

- Drivers:
  - NCDescriptorRealization
  - NCMatrixDescriptorRealization
  - NCMinimalDescriptorRealization
  - NCDeterminantalRepresentationReciprocal

  - NCSymmetrizeMinimalDescriptorRealization
  - NCSymmetricDescriptorRealization
  - NCSymmetricDeterminantalRepresentationDirect
  - NCSymmetricDeterminantalRepresentationReciprocal
  - NonCommutativeLift
- Auxiliary:
  - PinnedQ
  - PinningSpace
  - TestDescriptorRealization
  - SignatureOfAffineTerm

16.3.1 NCDescriptorRealization

`NCDescriptorRealization[RationalExpression,UnknownVariables]` returns a list of 3 matrices `{C,G,B}` such that $CG^{-1}B$ is the given `RationalExpression`. i.e. `NCDot[C,NCInverse[G],B] === RationalExpression`.

`C` and `B` do not contain any `UnknownsVariables` and `G` has linear entries in the `UnknownVariables`.

16.3.2 NCDeterminantalRepresentationReciprocal

`NCDeterminantalRepresentationReciprocal[Polynomial,Unknowns]` returns a linear pencil matrix whose determinant equals `Constant * CommuteEverything[Polynomial]`. This uses the reciprocal algorithm: find a minimal descriptor realization of `inv[Polynomial]`, so `Polynomial` must be nonzero at the origin.

16.3.3 NCMatrixDescriptorRealization

`NCMatrixDescriptorRealization[RationalMatrix,UnknownVariables]` is similar to `NCDescriptorRealization` except it takes a *Matrix* with rational function entries and returns a matrix of lists of the vectors/matrix `{C,G,B}`. A different `{C,G,B}` for each entry.

16.3.4 NCMinimalDescriptorRealization

`NCMinimalDescriptorRealization[RationalFunction,UnknownVariables]` returns `{C,G,B}` where `NCDot[C,NCInverse[G],B] == RationalFunction`, `G` is linear in the `UnknownVariables`, and the realization is minimal (may be pinned).

16.3.5 NCSymmetricDescriptorRealization

`NCSymmetricDescriptorRealization[RationalSymmetricFunction, Unknowns]` combines two steps: `NCSymmetrizeMinimalDescriptorRealization[NCMinimalDescriptorRealization[RationalSymmetricFunction, Unknowns]]`.

16.3.6 NCSymmetricDeterminantalRepresentationDirect

`NCSymmetricDeterminantalRepresentationDirect[SymmetricPolynomial,Unknowns]` returns a linear pencil matrix whose determinant equals `Constant * CommuteEverything[SymmetricPolynomial]`. This uses the direct algorithm: Find a realization of 1 - NCSymmetricPolynomial,...

16.3.7 NCSymmetricDeterminantalRepresentationReciprocal

`NCSymmetricDeterminantalRepresentationReciprocal[SymmetricPolynomial,Unknowns]` returns a linear pencil matrix whose determinant equals `Constant * CommuteEverything[NCSymmetricPolynomial]`. This uses the reciprocal algorithm: find a symmetric minimal descriptor realization of `inv[NCSymmetricPolynomial]`, so NCSymmetricPolynomial must be nonzero at the origin.

16.3.8 NCSymmetrizeMinimalDescriptorRealization

`NCSymmetrizeMinimalDescriptorRealization[{C,G,B},Unknowns]` symmetrizes the minimal realization `{C,G,B}` (such as output from `NCMinimalRealization`) and outputs `{Ctilda,Gtilda}` corresponding to the realization `{Ctilda, Gtilda,Transpose[Ctilda]}`.

WARNING: May produces errors if the realization doesn't correspond to a symmetric rational function.

16.3.9 NonCommutativeLift

`NonCommutativeLift[Rational]` returns a noncommutative symmetric lift of `Rational`.

16.3.10 SignatureOfAffineTerm

`SignatureOfAffineTerm[Pencil,Unknowns]` returns a list of the number of positive, negative and zero eigenvalues in the affine part of `Pencil`.

16.3.11 TestDescriptorRealization

`TestDescriptorRealization[Rat,{C,G,B},Unknowns]` checks if `Rat` equals $CG^{-1}B$ by substituting random 2-by-2 matrices in for the unknowns. `TestDescriptorRealization[Rat,{C,G,B},Unknowns,NumberOfTests]` can be used to specify the `NumberOfTests`, the default being 5.

16.3.12 PinnedQ

`PinnedQ[Pencil_,Unknowns_]` is True or False.

16.3.13 PinningSpace

`PinningSpace[Pencil_,Unknowns_]` returns a matrix whose columns span the pinning space of `Pencil`. Generally, either an empty matrix or a d-by-1 matrix (vector).

References

[Cam+03] Juan F. Camino et al. “Matrix Inequalities: a Symbolic Procedure to Determine Convexity Automatically”. In: *Integral Equation and Operator Theory* 46.4 (2003), pp. 399–454.

[de 12] Mauricio C. de Oliveira. “Simplification of symbolic polynomials on non-commutative variables”. In: *Linear Algebra and its Applications* 437.7 (2012), pp. 1734–1748. ISSN: 0024-3795. DOI: 10.1016/j.laa.2012.05.015.

[Mor94] Teo Mora. “An introduction to commutative and noncommutative Groebner Bases”. In: *Theoretical Computer Science* 134 (1994), pp. 131–173.