MathMatrixPack package documentation

Wolfram Mathematica® 10.0 package for matrices with labeled entries

Renato Maia Matarazzo Orsino

November 13, 2015

1 Formatting rules

This section presents the following formatting rules in the kernel of Mathematica when Package MathMatrixPack is used.

1.1 Trigonometric functions

```
$PrePrint = # /. {
   Csc[xArgument_] :> 1/Defer @ Sin[xArgument],
   Sec[xArgument_] :> 1/Defer @ Cos[xArgument],
   Tan[xArgument_] :> Defer @ Sin[xArgument]/Defer @ Cos[xArgument],
   Cot[xArgument_] :> Defer @ Cos[xArgument]/Defer @ Sin[xArgument]
   } &;

Unprotect[Cos, Sin];
Format[Cos[xArgument_]] := Subscript[c, xArgument]
Format[Sin[xArgument_]] := Subscript[s, xArgument]
Protect[Cos, Sin];
```

This piece of code modifies the default display notation for trigonometric functions in Mathematica: $\sin(*)$, $\cos(*)$, $\tan(*)$, $\cot(*)$, $\sec(*)$, $\csc(*)$ are denoted respectively as s_*, c_*, s_*/c_*, c_*/s_*, $1/c_*$ and $1/s_*$ for any (assigned or unassigned) variable used in the code.

1.2 Derivatives

```
Format[Subscript[ Subscript[xArgument_, xIndexes1__], xIndexes2__]'[t_]]
    :=
   Subscript[ Subscript[ Overscript[xArgument, "."], xIndexes1],
      xIndexes2][t]
Format[Subscript[ Subscript[xArgument_, xIndexes1__], xIndexes2__]''[t_
   ]] :=
  Subscript[ Subscript[ Overscript[xArgument, ".."], xIndexes1],
     xIndexes2][t]
Format[Subscript[xArgument_, xIndexes1__]'[t_]] :=
  Subscript[ Overscript[xArgument, "."], xIndexes1][t]
```

```
Format[Subscript[xArgument_, xIndexes1__]''[t_]] :=
  Subscript[ Overscript[xArgument, ".."], xIndexes1][t]
Format[xArgument_'[t_]] :=
  Overscript[xArgument, "."][t]
Format[xArgument_''[t_]] :=
  Overscript[xArgument, ".."][t]

SymbolReplacements = {
  Subscript[ Subscript[xBase_, xIndexes__], xIndexes2__]'[t] ->
    Subscript[ Subscript[ Overscript[xBase, "."], xIndexes], xIndexes2],
  Subscript[ Subscript[xBase_, xIndexes__], xIndexes2__]''[t] ->
    Subscript[Subscript[ Overscript[xBase, ".."], xIndexes], xIndexes2],
  Subscript[xBase_, xIndexes__]'[t] ->
    Subscript[ Overscript[xBase, "."], xIndexes],
  Subscript[xBase_, xIndexes__]''[t] ->
      Subscript[ Overscript[xBase, ".."], xIndexes],
  xVariable_'[t] -> Overscript[xVariable, "."],
  xVariable_''[t] -> Overscript[xVariable, ".."],
  xVariable_[t] -> xVariable
  };
```

This piece of code modifies the display notation for first and second order time derivatives: $\zeta'[t]$ and $\zeta''[t]$ are denoted respectively as $\dot{\zeta}[t]$ and $\ddot{\zeta}[t]$.

`SymbolReplacements` is a list of rules for formatting first and second order time derivatives. Whenever this list of rules is used, $\zeta'[t]$ and $\zeta''[t]$ will be replaced respectively by $\dot{\zeta}$ and $\ddot{\zeta}$.

1.3 Round-off rules

```
RoundOffRules = {
  xNumber_?NumericQ /; Abs[xNumber] < 10^-12 -> 0,
  xNumber_?NumericQ /; Abs[xNumber - 1] < 10^-12 -> 1,
  xNumber_?NumericQ /; Abs[xNumber + 1] < 10^-12 -> -1
  };
```

`RoundOffRules` is a list of rules for formatting numbers. Whenever this list of rules is used, numbers in the ranges $]-10^{-12}, +10^{-12}[$ and $]1-10^{-12}, 1+10^{-12}[$ will be displayed as 0 and 1, respectively.

1.4 Displaying matrices

```
SMatrixForm[xA_Association, xL_: Automatic] :=
 If[ KeyExistsQ[xA, "Column␣Labels"],
  MatrixForm[
   xA["Matrix"],
   TableHeadings -> ({xA["Row␣Labels"], xA["Column␣Labels"]} /. SymbolReplacements),
   TableAlignments -> xL
   ],
  MatrixForm[
   xA["Matrix"],
   TableHeadings -> ({xA["Row␣Labels"], None} /. SymbolReplacements),
   TableAlignments -> xL
   ]
  ]
```

```
STableForm[xA_Association, xL_: Left] :=
 If[ KeyExistsQ[xA, "Column␣Labels"],
  TableForm[
   xA["Matrix"],
   TableHeadings -> ({xA["Row␣Labels"], xA["Column␣Labels"]} /. SymbolReplacements),
   TableAlignments -> xL
   ],
  TableForm[
   xA["Matrix"],
   TableHeadings -> ({xA["Row␣Labels"], None} /. SymbolReplacements),
   TableAlignments -> xL
   ]
  ]
```

`SMatrixForm` and `STableForm` extend the application of the built-in functions `MatrixForm` and `TableForm` to matrices given by `Association` elements.

2 General purpose functions

This section presents some general purpose functions that can be used for other applications than the modelling of multibody systems.

2.1 Set complement

```
SetComplement[xMainSet_, xDiffSet_] :=
  Select[xMainSet, Not[MemberQ[xDiffSet, #]] &]
```

`SetComplement` returns the elements of the list `xMainSet` that are not in `xDiffSet` in the same order of occurrence in `xMainSet` (unlike the built-in function `Complement` that does the same opperation but sorts the output list).

2.2 Delete redundant expressions

```
RedundantElim[xX_] := DeleteDuplicates @ (DeleteCases[Simplify @ xX, 0])
    ;
```

`RedundantElim` deletes all repeated elements and all exact zeros (with head `Integer`) of a list.

2.3 Simplify Associations

```
SSimplify[xA_Association] := Association @
  MapThread[#1 -> #2 &, {First /@ (Normal @ xA), Simplify@(Last /@ (
      Normal @ xA))}, 1]

SSimplify[xX_] := Simplify[xX]
```

`SSimplify` is an extension of the built-in function `Simplify` applicable to `Association` elements.

2.4 Replacements in Associations

```
SReplaceRepeated[xA_Association, xL_List] :=
  Association @ MapThread[ #1 -> #2 &, {
    First /@ (Normal@xA),
    ReplaceRepeated[(Last /@ (Normal @ xA)), xL]
    }, 1]

SReplaceRepeated[xX_, xL_List] :=
  ReplaceRepeated[xX, xL]

SReplaceFullSimplify[xA_Association, xRules_List] :=
  Association @ MapThread[ #1 -> #2 &, {
      First /@ (Normal @ xA),
      FullSimplify[ FullSimplify[ Expand[(Last /@ (Normal @ xA)) //.
          xRules] //. xRules] //. xRules]
      }, 1]

SReplaceFullSimplify[xX_, xRules_List] :=
  FullSimplify[ FullSimplify[ Expand[(Flatten @ {xX}) //. xRules] //.
      xRules] //. xRules]

SReplaceSimplify[xA_Association, xRules_List] :=
  Association @ MapThread[ #1 -> #2 &, {
      First /@ (Normal @ xA),
      Simplify[ Simplify[ Expand[(Last /@ (Normal @ xA)) //. xRules] //.
          xRules] //. xRules]
      }, 1]

SReplaceSimplify[xX_, xRules_List] :=
  Simplify[ Simplify[ Expand[(Flatten @ {xX}) //. xRules] //. xRules]
      //. xRules]
```

SReplaceRepeated is an extension of the built-in function ReplaceRepeated applicable to Association elements.

SReplaceFullSimplify and SReplaceSimplify are functions that simultaneously perform replacements and simplify the resulting expressions. They apply the built-in functions ReplaceRepeated, Expand and FullSimplify or Simplify to the corresponding expressions (normally List or Association elements).

2.5 Rename keys and values in Associations

```
SRename[xIn_Association, xNamingRules_, xExtraRules_: {}] :=
 Association @ MapThread[ #1 -> #2 &, {
   If[Head[#] === String,
     StringReplace[#, xNamingRules], #] & /@ (First /@ Normal @ (xIn)),
   Map[SReplaceRepeated[#, xExtraRules] &, Map[SReplaceRepeated[#,
      xNamingRules] &,
     Map[SReplaceRepeated[#, xExtraRules] &, (Last /@ Normal @ xIn), All
       ], All], All]
   }, 1]
```

SRename[xIn, xNamingRules] replaces, according to xNamingRules, string ocurrences both in the keys and values of the Association element xIn.

SRename[xIn, xNamingRules, xExtraRules] also applies replacements according to xExtraRules to the values of the Association element xIn.

2.6 List variables in expressions

```
GetVariables[xX_List, xExcept_List: {}] :=
 Complement[ DeleteDuplicates @ Cases[xX, xVariable_[t], Infinity],
    xExcept]

GetVariables[xX_Association, xExcept_List: {}] :=
 Complement[ DeleteDuplicates @ Cases[xX["Matrix"], xVariable_[t],
    Infinity], xExcept]
```

GetVariables returns a list of all time dependent variables in a given symbolic expression xX (which can be either a List or an Association). With the optional argument xExcept_List the user can list the variables that must not be listed in the output.

```
HeadList = {
```

```
  Or, And,
  Equal, Unequal, Inequality
  Less, LessEqual,
  Greater, GreaterEqual
  };

GetAllVariables[xNumber_?NumericQ] :=
  Sequence[]

GetAllVariables[{}] :=
  Sequence[]

GetAllVariables[xRelationalOperator_] /; MemberQ[HeadList, xRelationalOperator] :=
  Sequence[]

GetAllVariables[x_List] :=
  DeleteDuplicates @ (Flatten @ (Union @ (GetAllVariables[#] & /@ x)))

GetAllVariables[Derivative[xNumber_Integer][xFunction_][xArgument__]] :=
  Module[{xVariable},
      If[MemberQ[Attributes[xFunction], NumericFunction] || MemberQ[HeadList, xFunction],
      (*-TRUE-*)
      xVariable = GetAllVariables[{xArgument}],
      (*-FALSE-*)
        xVariable = Derivative[xNumber][xFunction][xArgument]
        ];
      xVariable
    ]

GetAllVariables[xFunction_Symbol[xArgument__]] :=
  Module[{xVariable},
    If[MemberQ[Attributes[xFunction], NumericFunction] || MemberQ[HeadList, xFunction],
```

```
     (*-TRUE-*)
     xVariable = GetAllVariables[{xArgument}],
     (*-FALSE-*)
     xVariable = xFunction[xArgument]
     ];
  xVariable
  ];

GetAllVariables[xOther_] :=
  xOther
```

GetAllVariables returns a list of all symbolic variables (both time dependent variables and non-numeric parameters) in a given symbolic expression.

3 Matrix calculus

In package MathMatrixPack, matrices must have row and column labels in order to perform correctly the operations of matrix sum/assemble and matrix multiplication. Thus, in this package a matrix is represented by an Association element with 3 keys:

- "Matrix": a two dimensional array (List element) representing the matrix itself.
- "Row␣Labels": an ordered List providing the indexes of the respective rows of the declared matrix.
- "Column␣Labels": an ordered List providing the indexes of the respective columns of the declared matrix.

3.1 Sum, assemble and partitioning of matrices - AngleBracket operator

Wolfram Mathematica has some operators without built-in meanings. In MathMatrixPack, the operator AngleBracket, displayed as ⟨X,Y,...⟩, is used to perform the operations of sum, assemble and partitioning of matrices. The definitions for this operator are shown in the following piece of code:

```
Matrix2Rule[xA_Association] /; (ArrayDepth[xA["Matrix"]] > 1) :=
  Association @ Flatten @ MapThread[ (#1 -> #2) &, {
```

```
    Outer[{#1, #2} &, xA["Row␣Labels"], xA["Column␣Labels"]],
    xA["Matrix"]
    }, 2]

Matrix2Rule[xA_Association] :=
  Association @ Flatten @ MapThread[ (#1 -> #2) &, {
    xA["Row␣Labels"],
    xA["Matrix"]
    }, 1]
```

```
AngleBracket[xA__Association] :=
  Module[{xAList, xRowLabels, xColumnLabels, xRList},
    xAList = List[xA];
    xRowLabels = Union @ (Join @@ ((#["Row␣Labels"]) & /@ xAList));
    If[ And @@ (KeyExistsQ[#, "Column␣Labels"]& /@ xAList),
      xColumnLabels = Union @ (Join @@ ((#["Column␣Labels"]) & /@ xAList)
          );
      xRList = Association @ ((# -> Plus @@ DeleteCases[# /. (Matrix2Rule
           /@ xAList), #]) & /@
        Flatten[Outer[{#1, #2} &, xRowLabels, xColumnLabels], 1]);
      Association[
        "Matrix" -> Outer[{#1, #2} &, xRowLabels, xColumnLabels] /.
           xRList,
        "Row␣Labels" -> xRowLabels,
        "Column␣Labels" -> xColumnLabels
        ],
      xRList = Association @ ((# -> Plus @@ DeleteCases[# /. (Matrix2Rule
           /@ xAList), #]) & /@
        xRowLabels);
      Association[
        "Matrix" -> xRowLabels /. xRList,
        "Row␣Labels" -> xRowLabels
        ]
      ]
    ]
```

```

AngleBracket[xMatrix_List, xLabels_List] /; (ArrayDepth[xMatrix] === 1) :=
  AngleBracket @ Association[
    "Matrix" -> xMatrix,
    "Row Labels" -> xLabels
    ]

AngleBracket[xMatrix_List, xColumnLabels_List, xRowLabels_List: {}] /; (ArrayDepth[xMatrix] > 1) :=
  AngleBracket @ Association[
    "Matrix" -> xMatrix,
    "Row Labels" -> If[xRowLabels === {},
      Range @ (First @ (Dimensions @ xMatrix)),
      xRowLabels
      ],
    "Column Labels" -> xColumnLabels
    ]

AngleBracket[0, xRowLabels_List] :=
  AngleBracket @ Association[
    "Matrix" -> Array[0 &, {Length @ xRowLabels}],
    "Row Labels" -> xRowLabels
    ]

AngleBracket[0, xColumnLabels_List, xRowLabels_List] :=
  AngleBracket @ Association[
    "Matrix" -> Array[0 &, {Length @ xRowLabels, Length @ xColumnLabels}],
    "Row Labels" -> xRowLabels,
    "Column Labels" -> xColumnLabels
    ]

AngleBracket[1, xLabels_List] :=
  Association[
```

```
    "Matrix" -> IdentityMatrix[Length @ xLabels],
    "Row␣Labels" -> xLabels,
    "Column␣Labels" -> xLabels
    ]

AngleBracket[1, xColumnLabels_List, xRowLabels_List] :=
  Module[{xId},
    xId = Intersection[xColumnLabels, xRowLabels];
    AngleBracket[
      Association[
        "Matrix" -> IdentityMatrix[Length @ xId],
        "Row␣Labels" -> xId,
        "Column␣Labels" -> xId
        ],
      AngleBracket[0, xColumnLabels, xRowLabels]
      ]
    ]

AngleBracket[xA_Association, xRowLabels_List] /; KeyExistsQ[xA, "Column␣Labels"] :=
  AngleBracket[
    AngleBracket[0, xA["Column␣Labels"], xRowLabels],
    Association[
      "Matrix" -> Part[xA["Matrix"],
        Flatten @ (Position[xA["Row␣Labels"], #] & /@ Intersection[
            xRowLabels, xA["Row␣Labels"]]), All],
      "Row␣Labels" -> Intersection[xRowLabels, xA["Row␣Labels"]],
      "Column␣Labels" -> xA["Column␣Labels"]
      ]
    ]

AngleBracket[xA_Association, xRowLabels_List] :=
  AngleBracket[
    AngleBracket[0, xRowLabels],
    Association[
```

```
    "Matrix" -> Part[xA["Matrix"],
      Flatten@(Position[xA["Row Labels"], #] & /@ Intersection[xRowLabels, xA["Row Labels"]])],
    "Row Labels" -> Intersection[xRowLabels, xA["Row Labels"]]
    ]
  ]

AngleBracket[xA_Association, xRowLabel_] /; KeyExistsQ[xA, "Column Labels"] :=
 If[First @ Dimensions @ (xA["Column Labels"]) == 1,
  Part[xA["Matrix"], First @ (Flatten @ (Position[xA["Row Labels"], xRowLabel])), 1],
  AngleBracket @ Association[
    "Matrix" -> Part[xA["Matrix"], Flatten @ (Position[xA["Row Labels"], xRowLabel]), All],
    "Row Labels" -> {xRowLabel},
    "Column Labels" -> xA["Column Labels"]
    ]
  ]

AngleBracket[xA_Association, xRowLabel_] :=
 Part[xA["Matrix"], First @ (Flatten @ (Position[xA["Row Labels"], xRowLabel]))]

AngleBracket[xA_Association, xRowLabels_List, xColumnLabels_List] :=
 AngleBracket[
  AngleBracket[0, xColumnLabels, xRowLabels],
  Association[
    "Matrix" -> Part[xA["Matrix"],
      Flatten @ (Position[xA["Row Labels"], #] & /@ Intersection[xRowLabels, xA["Row Labels"]]),
      Flatten @ (Position[xA["Column Labels"], #] & /@ Intersection[xColumnLabels, xA["Column Labels"]])],
    "Row Labels" -> Intersection[xRowLabels, xA["Row Labels"]],
    "Column Labels" -> Intersection[xColumnLabels, xA["Column Labels"]]
```

```
      ]
    ]

AngleBracket[xA_Association, All, xColumnLabels_List] :=
  AngleBracket[
    AngleBracket[0, xColumnLabels, xA["Row␣Labels"]],
    Association[
      "Matrix" -> Part[xA["Matrix"], All,
        Flatten @ (Position[xA["Column␣Labels"], #] & /@ Intersection[
            xColumnLabels, xA["Column␣Labels"]])],
      "Row␣Labels" -> xA["Row␣Labels"],
      "Column␣Labels" -> Intersection[xColumnLabels, xA["Column␣Labels"]]
      ]
    ]

AngleBracket[xA_Association, xRowLabels_List, xColumnLabel_] :=
  AngleBracket[
    AngleBracket[0, {xColumnLabel}, xRowLabels],
    Association[
      "Matrix" -> Transpose @ ({Part[xA["Matrix"],
        Flatten @ (Position[xA["Row␣Labels"], #] & /@ Intersection[
            xRowLabels, xA["Row␣Labels"]]),
        First @ (Flatten @ (Position[xA["Column␣Labels"], xColumnLabel]))
            ]}),
      "Row␣Labels" -> Intersection[xRowLabels, xA["Row␣Labels"]],
      "Column␣Labels" -> {xColumnLabel}
      ]
    ]

AngleBracket[xA_Association, All, xColumnLabel_] :=
  AngleBracket @ Association[
    "Matrix" -> Transpose @ ({Part[xA["Matrix"], All,
      First @ (Flatten @ (Position[xA["Column␣Labels"], xColumnLabel]))
          ]}),
    "Row␣Labels" -> xA["Row␣Labels"],
```

```
  "Column Labels" -> {xColumnLabel}
  ]

AngleBracket[xA_Association, xRowLabel_, xColumnLabels_List] :=
 AngleBracket [
  AngleBracket[0, xColumnLabels, {xRowLabel}],
  Association[
   "Matrix" -> Part[xA["Matrix"], First @ Flatten @ (Position[xA["Row Labels"], xRowLabel]),
    Flatten @ (Position[xA["Column Labels"], #] & /@ Intersection[xColumnLabels, xA["Column Labels"]])],
   "Row Labels" -> {xRowLabel},
   "Column Labels" -> Intersection[xColumnLabels, xA["Column Labels"]]
   ]
  ]

AngleBracket[xA_Association, xRowLabel_, xColumnLabel_] :=
 Part[xA["Matrix"], First @ Flatten @ (Position[xA["Row Labels"], xRowLabel]),
  First @ Flatten @ (Position[xA["Column Labels"], xColumnLabel])]

AngleBracket[xFunction_, xA_Association] :=
 If[KeyExistsQ[xA, "Column Labels"],
  AngleBracket @ Association[
   "Matrix"-> xFunction @ (xA["Matrix"]),
   "Column Labels"-> xA["Column Labels"],
   "Row Labels"-> xA["Row Labels"]
   ],
  AngleBracket @ Association[
   "Matrix"-> xFunction @ (xA["Matrix"]),
   "Row Labels"-> xA["Row Labels"]
   ]
  ]

SApply[xFunction_, xA_Association] := AngleBracket[xFunction, xA]
```

```

SAssemble = SPart = AngleBracket;
```

When `AngleBracket` is called with a sequence of matrices (sequence of `Association` elements, ⟨`X,Y,...`⟩), the output is a new `Association` element (representing a matrix) consisting of an assemble of the inputs in which elements having simultaneously the same row and column labels are added up. The `"Row␣Labels"` and `"Column␣Labels"` lists of the output consist of an sorted version of the union of all the respective lists of the inputs. Thus, in this usage, `AngleBracket` operator performs the opetations of matrix sum and assemble.

Let `xV` be a `List` element representing a column-matrix. `AngleBracket[xV, xRowLabels]` or ⟨`xV, xRowLabels`⟩ gives the `Association` element representing the column-matrix `xV` whose rows are labeled by `xRowLabels`.

Let `xM` be a `List` element representing a matrix (two-dimensional array). `AngleBracket[xM, xColumnLabels]` or ⟨`xM, xColumnLabels`⟩ gives the `Association` element representing the matrix `xM` whose rows are labeled by a sequence of integers and the columns are labeled by `xColumnLabels`. `AngleBracket[xM, xColumnLabels, xRowLabels]` or ⟨`xM, xColumnLabels`⟩ gives the `Association` element representing the matrix `xM` whose rows are labeled by `xRowLabels` and the columns are labeled by `xColumnLabels`.

`AngleBracket[0, xRowLabels]` or ⟨`0, xRowLabels`⟩ gives the `Association` element representing the null column-matrix whose rows are labeled by `xRowLabels`. `AngleBracket[0, xColumnLabels, xRowLabels]` or ⟨`0, xColumnLabels, xRowLabels`⟩ gives the `Association` element representing the null matrix whose rows are labeled by `xRowLabels` and the columns are labeled by `xColumnLabels`. `AngleBracket[1, xLabels]` or ⟨`1, xLabels`⟩ gives the `Association` element representing the identity whose rows and columns are labeled by `xLabels`. `AngleBracket[1, xColumnLabels, xRowLabels]` or ⟨`1, xColumnLabels, xRowLabels`⟩ gives the `Association` element representing the matrix defined by the Kronecker Delta function whose rows are labeled by `xRowLabels` and the columns are labeled by `xColumnLabels`.

`SApply[xFunction, xX]` or ⟨`xFunction, xX`⟩ applies the unary function `xFunction` to the entry whose key is `"Matrix"` in the `Association xX`.

All the other uses of `AngleBracket` correspond to partitioning of matrices. In these cases `AngleBracket` is called with a sequence of two or three arguments (the third argument is optional), in which the first one must correspond to a matrix (`Association` element), the second one can be a list of row labels, a single row label or the keyword `All` and the third (optional) can be a list of column labels or a single column label. When a the first argument represents a column-matrix and the second is a single row label, or when the first represent a matrix, the second is a single row label and the third, a single column label, then the output of the operator is a the expression of the corresponding element (i.e., not a `List` nor an `Association`). In all the other cases, the output is an `Association` representing a matrix constituted only by the corresponding rows and columns of the input matrix. When the keyword `All` is used in the second argument, all rows of the original matrix are selected. When the third argument is not used, all the columns of the original matrix are selected.

3.2 Matrix multiplication and multiplication of a matrix by a scalar

```
CircleDot[xX_Association, xY_Association] /;
  And[xX["Matrix"] === {}, ArrayDepth[xY["Matrix"]] === 1] :=
  Association[
    "Matrix" -> {},
    "Row␣Labels" -> {}
    ]

CircleDot[xX_Association, xY_Association] /;
  And[xX["Matrix"] === {}] :=
  Association[
    "Matrix" -> {},
    "Row␣Labels" -> {},
    "Column␣Labels" -> xY["Column␣Labels"]
    ]

CircleDot[xX_Association, xY_Association] /;
  And[ArrayDepth[xX["Matrix"]] === 1, ArrayDepth[xY["Matrix"]] === 1] :=
  Module[{xA, xB, xU},
    xU = Association[
      "Matrix" -> Array[0&, Length @ #],
      "Row␣Labels" -> #
      ] & @ Union[xX["Row␣Labels"], xY["Row␣Labels"]];
    xA = SAssemble[xX, xU];
    xB = SAssemble[xY, xU];
```

```
    xA["Matrix"].xB["Matrix"]
    ]

CircleDot[xX_Association, xY_Association] /;
  And[ArrayDepth[xY["Matrix"]] === 1] :=
  Module[{xA, xB},
    xA = Association[
      "Matrix" -> Array[0&, {Length @ #1, Length @ #2}],
      "Row␣Labels" -> #1,
      "Column␣Labels" -> #2
      ] & @@ {xX["Row␣Labels"], Union[xX["Column␣Labels"], xY["Row␣Labels"]]};
    xB = Association[
      "Matrix" -> Array[0&, Length @ #],
      "Row␣Labels" -> #
      ] & @ Union[xX["Column␣Labels"], xY["Row␣Labels"]];
    xA = SAssemble[xX, xA];
    xB = SAssemble[xY, xB];
    Association[
      "Matrix" -> xA["Matrix"].xB["Matrix"],
      "Row␣Labels" -> xA["Row␣Labels"]
      ]
    ]

CircleDot[xX_Association, xY_Association] :=
  Module[{xA, xB},
    xA = Association[
      "Matrix" -> Array[0&, {Length @ #1, Length @ #2}],
      "Row␣Labels" -> #1,
      "Column␣Labels" -> #2
      ] & @@ {xX["Row␣Labels"], Union[xX["Column␣Labels"], xY["Row␣Labels"]]};
    xB = Association[
      "Matrix" -> Array[0&, {Length @ #1, Length @ #2}],
      "Row␣Labels" -> #1,
```

```
      "Column␣Labels" -> #2
      ] & @@ {Union[xX["Column␣Labels"], xY["Row␣Labels"]], xY["Column␣Labels"]};
    xA = SAssemble[xX, xA];
    xB = SAssemble[xY, xB];
    Association[
      "Matrix" -> xA["Matrix"].xB["Matrix"],
      "Row␣Labels" -> xA["Row␣Labels"],
      "Column␣Labels" -> xB["Column␣Labels"]
      ]
    ]

CircleDot[xX_Association, xY_List] :=
  xX["Matrix"].xY

CircleDot[xX_Association, xY_] :=
  SApply[(xY #)&, xX]

CircleDot[xY_, xX_Association] :=
  SApply[(xY #)&, xX]

SDot = CircleDot;
```

In the package MathMatrixPack, the operator `CircleDot`, denote by `X⊙Y` or the function `SDot[X, Y]` is used to denote the operations of matrix multiplication and multiplication of a matrix by a scalar. In the case of matrix multiplication, either both `CircleDot` arguments are `Association` elements or the first one is an `Association` element and the second one a `List` element. If the second argument is an `Association` , the output will be an `Association` representing the matrix multiplication between both input arguments. If the second argument is a `List`, the output will be a `List` representing the matrix multiplication between both input arguments. In the case of multiplication of a matrix by a scalar, one argument must be an `Association` and the other an scalar. The order of the arguments is not relevant in this case, and the output is an `Association` representing the corresponding multplication of the matrix by the scalar.

3.3 Matrix transposition

```
SuperDagger[xX_Association] :=
  Association[
    "Matrix"->Transpose @ xX["Matrix"],
    "Column␣Labels"-> xX["Row␣Labels"],
    "Row␣Labels"-> xX["Column␣Labels"]
    ]

STranspose[xX_Association] := SuperDagger[xX]
```

In the package MathMatrixPack, the operator `SuperDagger`, denote by $\mathtt{X}^{\dagger}$ is used to denote the operation of transposition of matrices. It extends the use of the built-in function `Transpose` (that is applicable to `List` elements representing matrices) to `Association` elements representing matrices. The unary function `STranspose` does the same as the operator `SuperDagger`.

3.4 Affine Transformations

```
BracketingBar[xX_Association] :=
  AffineTransform[xX["Matrix"]]

BracketingBar[xX_List /; Dimensions[xX]=={3,3}] :=
  AffineTransform[xX]

BracketingBar[xX_List /; Dimensions[xX]=={4,4}] :=
  LinearFractionalTransform[xX]
```

In the package MathMatrixPack, the operator `BracketingBar`, denote by $\dashv \mathtt{X} \vdash$ is used to convert matrices into affine operators. Whenever the (single) argument of the operator is an `Association`, the output is a `TransformationFunction` given by the application of the built-in `AffineTransform` to the `Association` entry whose key is `"Matrix"`. The same kind of output will be obtained if the argument of the operator is a 3×3 `List` element. However, when the argument is a 4×4 `List` element, the corresponding `TransformationFunction` is obtained by the application of the built-in `LinearFractionalTransform` function (whose output represents a homogeneous transformation).

3.5 Coefficient arrays

```
SCoefficientArrays[xA_Association, xVariables_List, xRules_List:{}] :=
 Module[{x},
   x["Row␣Labels"] = xA["Row␣Labels"];
   x["Expressions"] = Flatten @ (xA["Matrix"]);
   x["Coefficient␣Arrays"] = CoefficientArrays[x["Expressions"] //.
       xRules, xVariables];
   {
   Association[
     "Matrix" -> Part[#, 1]& @ x["Coefficient␣Arrays"],
     "Row␣Labels" -> x["Row␣Labels"]
     ],
   Association[
     "Matrix" -> Part[#, 2]& @ x["Coefficient␣Arrays"],
     "Row␣Labels" -> x["Row␣Labels"],
     "Column␣Labels" -> xVariables
     ]
   }
   ]


SMatrixCoefficientArrays[xA_Association, xRules_List: {}] :=
 Module[{xxMatrix, xxVariables, xxRowLabels, xxColumnLabels,
     xxCoefficientMatrices},
   xxMatrix = xA["Matrix"] //. xRules;
   xxRowLabels = xA["Row␣Labels"];
   xxColumnLabels = xA["Column␣Labels"];
   xxVariables = Union @ GetVariables[xxMatrix];
   xxCoefficientMatrices = CoefficientArrays[xxMatrix,xxVariables];
   {
   Association[ Union@@{
     {
     1->
     Association[
     "Matrix"->Normal@Part[xxCoefficientMatrices,1],
```

```
    "Column Labels"->xxColumnLabels,
    "Row Labels"->xxRowLabels
    ]
    },
    MapThread[ (#1-> Association[
      "Matrix"->Normal@Part[xxCoefficientMatrices,2,All,All,#2],
      "Column Labels"->xxColumnLabels,
      "Row Labels"->xxRowLabels
      ])&, {#,Range@Length@#}, 1]& @ xxVariables
    }],
  xxVariables
  }
  ]
```

`SCoefficientArrays` is an extension of the built-in function `CoefficientArrays` that is applicable to matrices represented by `Association` elements. This function can be called with two or three arguments (being the third optional), i.e., both syntaxes `SCoefficientArrays[M, V, R]` and `SCoefficientArrays[M, V]` are valid. In both cases, the function transforms the `Association` element `M` in a `List` of expressions `E`, applies to this list the transformation rules `R` whenever they are defined, and returns a `List` element `{K, H}`, containing two `Association` elements, `K` and `H`, such that the affine part of `E` (i.e., terms of the expressions in `E` that are either independent or linear dependent of the variables in `V`) is given by ⟨`K, H⊙V`⟩.

In order to understand how the function `SMatrixCoefficientArrays` works, consider a matrix $\boldsymbol{M}$ that may be dependent of some scalar variables $(v_1, \ldots, v_r)$, i.e., $\boldsymbol{M} = \underline{\boldsymbol{M}}(v_1, \ldots, v_r)$. If $\boldsymbol{M}$ is affine with respect to these variables, then there is a list of constant matrices $\boldsymbol{M}_1, \boldsymbol{M}_{v_1}, \ldots, \boldsymbol{M}_{v_r}$ such that:

$$\boldsymbol{M} = 1\,\boldsymbol{M}_1 + \sum_{k=1}^{r} v_k\,\boldsymbol{M}_{v_k}$$

`SMatrixCoefficientArrays[M]` or `SMatrixCoefficientArrays[M,R]` are valid syntaxes for this function, with `M` being an `Association` element representing a matrix $\boldsymbol{M}$ and with `M` being an optional `List` of replacement rules, to be applied to this matrix. The output is the `List` `{X, V}`, with `X` being an `Association` element of the form

```
Association[1 -> M_1, v_1 -> M_v1, ..., v_r -> M_vr ]
```

($\mathtt{M}_1$, $\mathtt{M}_{v_1}$, ..., $\mathtt{M}_{v_r}$ are the `Association` elements representing the corresponding coefficient matrices $\boldsymbol{M}_1, \boldsymbol{M}_{v_1}, \ldots, \boldsymbol{M}_{v_r}$) and with `V` being the `List` $\{v_1, \ldots, v_r\}$.

3.6 Linear Solve

```
SLinearSolve[xX_Association,xY_Association] :=
  Module[{xA,xB},
    xA = SAssemble[xX];
    xB = SAssemble[xY];
    If[xA["Row␣Labels"] === xB["Row␣Labels"],
      Association[
        "Matrix" -> LinearSolve[xA["Matrix"], xB["Matrix"]],
        "Column␣Labels" -> xB["Column␣Labels"],
        "Row␣Labels" -> xA["Column␣Labels"]
        ],
        "Error"
      ]
    ]
```

`SLinearSolve` extends the application of the built-in function `LinearSolve` (originally applicable to a pair of `List` elements representing matrices) to pairs of `Association` elements representing matrices. The output of `SLinearSolve[A, B]` is an `Association` element `Z` such that `A⊙Z == B`.

```
SLeastSquares[xX_Association,xY_Association] :=
  Module[{xA,xB},
    xA = SAssemble[xX];
    xB = SAssemble[xY];
    If[xA["Row␣Labels"] === xB["Row␣Labels"],
      Association[
        "Matrix" -> LeastSquares[xA["Matrix"], xB["Matrix"]],
        "Column␣Labels" -> xB["Column␣Labels"],
        "Row␣Labels" -> xA["Column␣Labels"]
        ],
        "Error"
      ]
    ]
```

`SLeastSquares` extends the application of the built-in function `LeastSquares` (originally applicable to a pair of `List` elements representing matrices) to pairs of `Association` elements representing matrices. The output of `SLeastSquares[A, B]` is an `Association` element `Z` which is a least squares solution for `X` in the matrix equation ⟨`A`⊙`X, B`⟩ `== 0`.

```
LSSolver[xEquations_List, xGenVariables_List, xIndVariables_List,
  xRules_List: {}, xExtraRules_List: {}, xSize_Integer: 0,
      xTestParameters_List: {}, xSymmetry_: Automatic] :=
  Module[{xIn, xSol, xXe, xXi, xC},
    xXi = Union[Complement[GetVariables @ xEquations, xGenVariables],
        xIndVariables];
    xIn = Select[xEquations, (Length[#] <= xSize) &];
    xXe = Complement[GetVariables @ xIn, xXi];
    xSol = Flatten @ (Quiet @ Solve[(# == 0)& /@ xIn, xXe]);
    (*  xS = Jacobi[
      xGenVariables //. xSol,
      Union[Complement[xGenVariables, First /@ xSol], xXi],
      xGenVariables
      ]; *)
    xIn = Collect[RedundantElim @ (Expand @ (xEquations //. xSol) //.
        xExtraRules), xX_[t], Simplify] //. xRules;
    xC = LSReferenceOrthogonalComplement[
      Jacobi[#, Union[GetVariables[#], xXi]]& @ xIn ,
      xXi, xSymmetry, xTestParameters
      ];
    (* xS = xS ~SDot~ xC; *)
    {
      Union[
        xSol,
        Select[ MapThread[(#1 -> #2) &,
          {#["Row Labels"], (#["Matrix"].#["Column Labels"]) /.
            (xSymmetry["Extra Rules"] //. Missing[xX__] -> {})},
          1] & @ xC, Not @ (Expand[First[#] - Last[#]] === 0) &]],
      xC["Test Parameters"]
    }
```

```
  ]
```

3.7 Jacobians

```
Jacobi[xExpressionsList_, xVariablesList_] :=
 Association[
  "Matrix" -> D[xExpressionsList, {xVariablesList}],
  "Column␣Labels" -> xVariablesList,
  "Row␣Labels" -> Range @@ Dimensions @ xExpressionsList
  ]

Jacobi[xExpressionsList_,xVariablesList_, xExpressionsLabels_] :=
 Association[
  "Matrix" -> D[xExpressionsList, {xVariablesList}],
  "Column␣Labels" -> xVariablesList,
  "Row␣Labels" -> xExpressionsLabels
  ]
```

`Jacobi` obtains the Jacobian matrix of a given list of expressions with respect to a list of variables. The syntax of this function is `Jacobi[E, V, L]` or `Jacobi[E, V]` (i.e., the third argument is optional). `E` is an expression or a list of symbolic expressions, `V` is a list of variables and `L` is a list of labels for the corresponding expressions. The output is an `Association` element, representing the Jacobian matrix of `E` with respect to the variables in `V`. The `"Column␣Labels"` entry of the output is the list `V` and the `"Row␣Labels"` entry is `L`, if it is an input argument, or a list of positive integer indexes, otherwise.

3.8 Orthogonal complement

```
OrthogonalComplement[xJacobian_] :=
 Module[{x},
  x["Null␣Space␣Matrix"] = Transpose @ NullSpace[xJacobian["Matrix"]];
  x["Independent␣Variations"] = (Range @ Part[Dimensions[x["Null␣Space␣Matrix"]], 2]);
  Association[
```

```
      "Matrix" -> x["Null Space Matrix"],
      "Column Labels" -> x["Independent Variations"],
      "Row Labels" -> xJacobian["Column Labels"]
      ]
    ]

OrthogonalComplement[xJacobian_, xLabel_String] :=
  Module[{x},
    x["Null Space Matrix"] = Transpose @ NullSpace[xJacobian["Matrix"]];
    x["Independent Variations"] = Subscript[OverTilde[q],xLabel,#][t]& /@
      (Range @ Part[Dimensions[x["Null Space Matrix"]], 2]);
    Association[
      "Matrix" -> x["Null Space Matrix"],
      "Column Labels" -> x["Independent Variations"],
      "Row Labels" -> xJacobian["Column Labels"]
      ]
    ]

OrthogonalComplement[xJacobian_, xIndependentVariablesList_List] :=
  Module[{x, xOrthogonalComplement},
    {x["Number of Constraints"], x["Number of Variables"]} = Dimensions[xJacobian["Matrix"]];
    x["Number of Degrees of Freedom"] = x["Number of Variables"] - x["Number of Constraints"];
    If[ x["Number of Degrees of Freedom"] === Length @ xIndependentVariablesList,
      (*-TRUE-*)
      x["Independent Variables Column Indexes"] =
        Flatten[ Position[xJacobian["Column Labels"], #]& /@ xIndependentVariablesList, Infinity];
      x["Redundant Variables Column Indexes"] =
        Complement[Range @@ Dimensions @ xJacobian["Column Labels"], x["Independent Variables Column Indexes"]];
      xOrthogonalComplement = Association[
```

```
      "Matrix" -> Array[0&, {x["Number␣of␣Variables"], x["Number␣of␣Degrees␣of␣Freedom"]}],
      "Column␣Labels" -> xIndependentVariablesList,
      "Row␣Labels" -> xJacobian["Column␣Labels"]
      ];
    xOrthogonalComplement[["Matrix", x["Independent␣Variables␣Column␣Indexes"]]] =
      IdentityMatrix @ x["Number␣of␣Degrees␣of␣Freedom"];
    xOrthogonalComplement[["Matrix", x["Redundant␣Variables␣Column␣Indexes"]]] =
      LinearSolve @@ {
        xJacobian[["Matrix", All, x["Redundant␣Variables␣Column␣Indexes"]]],
        -xJacobian[["Matrix", All, x["Independent␣Variables␣Column␣Indexes"]]]
        };
    xOrthogonalComplement,
    (*-FALSE-*)
    "Error"
    ]
  ]
```

`OrthogonalComplement` calculates an orthogonal complement of a (Jacobian) matrix. Two syntaxes are possible for this function:

- `OrthogonalComplement[A]` calculates *an* orthogonal complement for the matrix represented by the `Association` element `A` using the built-in `NullSpace` function. The output is an `Association` element `C` whose `"Row␣Labels"` entry is equal to the `"Column␣Labels"` entry of the input argument and whose `"Column␣Labels"` entry is a list of positive integer indexes; also, `A⊙C == 0`.

- `OrthogonalComplement[A, V]` calculates *the* orthogonal complement for the matrix represented by the `Association` element `A` with respect to the independent set of variables represented by the `List` element `V` using the built-in `LinearSolve` function. The output is an `Association` element `C` whose `"Row␣Labels"` entry is equal to the `"Column␣Labels"` entry of the input argument and whose `"Column␣Labels"` entry is equal to `V`; also, `A⊙C == 0`.

```
LSNumericalOrthogonalComplement[xJacobian_, xIndependentVariablesList_List] :=
 Module[{x, xOrthogonalComplement},
   {x["Number␣of␣Constraints"], x["Number␣of␣Variables"]} = Dimensions[xJacobian["Matrix"]];
   x["Number␣of␣Degrees␣of␣Freedom"] = Part[Dimensions @ xIndependentVariablesList, 1];
   x["Independent␣Variables␣Column␣Indexes"] =
     Flatten[Position[xJacobian["Column␣Labels"], #]& /@ xIndependentVariablesList, Infinity];
   x["Redundant␣Variables␣Column␣Indexes"] =
     Complement[Range @@ Dimensions @ xJacobian["Column␣Labels"], x["Independent␣Variables␣Column␣Indexes"]];
   xOrthogonalComplement = Association[
     "Matrix"->Array[0&, {x["Number␣of␣Variables"], x["Number␣of␣Degrees␣of␣Freedom"]}],
     "Column␣Labels" -> xIndependentVariablesList,
     "Row␣Labels" -> xJacobian["Column␣Labels"]
     ];
   xOrthogonalComplement[["Matrix", x["Independent␣Variables␣Column␣Indexes"]]] =
     IdentityMatrix @ x["Number␣of␣Degrees␣of␣Freedom"];
   xOrthogonalComplement[["Matrix", x["Redundant␣Variables␣Column␣Indexes"]]] =
     LeastSquares @@ {
       xJacobian[["Matrix", All, x["Redundant␣Variables␣Column␣Indexes"]]],
       -xJacobian[["Matrix", All, x["Independent␣Variables␣Column␣Indexes"]]]
       };
   xOrthogonalComplement
   ]

```

```
LSReferenceOrthogonalComplement[xJacobian_Association, xIndependentVariables_List,
 xSymmetry_: Automatic, xTestParameters_List: {}, xNZero_Rational:1 10^-5] :=
 Module[{x, xNC, xSC, xNTestParameters},
  xNTestParameters = Union[
   xTestParameters,
   If[xSymmetry === Automatic,
     (#-> RandomReal[1])& /@ (GetAllVariables[(Flatten @ (Union @@ ( Normal @
      (#["Matrix"])& @ xJacobian))) //. xTestParameters]),
    xSymmetry["Function"] /@ (GetAllVariables[(Flatten @ (Union @@ ( Normal @
      (#["Matrix"])& @ xJacobian))) //. xTestParameters])
    ]
   ];
  xNC = LSNumericalOrthogonalComplement[
   SReplaceRepeated[xJacobian, xNTestParameters],
   xIndependentVariables
   ];
  xNC = AppendTo[xNC, "Matrix" -> Round[xNC["Matrix"], xNZero]];
  x["Column␣Labels"] = xNC["Column␣Labels"] //. SymbolReplacements;
  x["Row␣Labels"]= xNC["Row␣Labels"] //.SymbolReplacements;

  x["New␣Parameters"]={};
  If[xSymmetry === Automatic,
   (*-TRUE-*)
   Function[xRowLabel,
    x["Row␣Number"] = First @ (Flatten @ Position[x["Row␣Labels"], xRowLabel]);
    x["Parameters␣Values"] = Flatten @ (Part[xNC["Matrix"], x["Row␣Number"]]);
    x["Parameters␣Names"] = Flatten @
     ((Function[{xColumnLabel}, Subscript[OverBar[\[CapitalGamma]], xRowLabel,xColumnLabel]]) /@
```

```
          x["Column␣Labels"]);
        x["New␣Parameters:1"] = MapThread[(#2-> #1)&, {x["Parameters␣Values"], x["Parameters␣Names"]}, 1];
        x["New␣Parameters:2"] = (Flatten @ (Normal @ DeleteCases[Association[x["New␣Parameters:1"]], _Integer]));
        xNTestParameters = Union[xNTestParameters, N[x["New␣Parameters:2"]]]];
        x["New␣Parameters"] = Union[
          x["New␣Parameters"],
          (Reverse /@ x["New␣Parameters:2"]),
          (Reverse /@ x["New␣Parameters:2"]) /. ((xA_->xB_ )->(-xA->-xB ))
          ];
        ]/@ x["Row␣Labels"],
      (*-FALSE-*)
      Function[{xRowLabel},
        If[Intersection[xSymmetry["Secondary"],
          Flatten @ (Characters /@ Select[xRowLabel /. {Subscript[xV_, xS__] -> {xV, xS}, xV_ -> {xV}}, StringQ])
            ] === {}
          (* Not @ And[
          StringQ[Quiet @ Last[xRowLabel]],
          StringTake[Last[xRowLabel], -1] === xSymmetry["Secondary"]
          ] *),
          x["Row␣Number"] = First @ (Flatten @ Position[x["Row␣Labels"], xRowLabel]);
          x["New␣Subscript"] = If[Intersection[xSymmetry["Primary"],
            Flatten @ (Characters /@ Select[xRowLabel /. {Subscript[xV_, xS__] -> {xV, xS}, xV_ -> {xV}}, StringQ])
              ] === {}
             (* Quiet @ (StringTake[Last[xRowLabel], -1] === xSymmetry["Primary"]) *),
            xRowLabel,
            Subscript @@ (If[StringQ[#], StringReplace[#, (# -> "")& /@ xSymmetry["Primary"]], #] & /@
```

```
            (xRowLabel /. (Subscript[xV_, xS__] -> {xV, xS}))
          ];
        x["Parameters Values"] = Flatten @ (Part[xNC["Matrix"], x["Row Number"]]);
        x["Parameters Names"] = Flatten @ ((Function[{xLabel},
            Subscript[OverBar[\[CapitalGamma]], x["New Subscript"], xLabel]
            ]) /@ x["Column Labels"]);
        x["New Parameters:1"] = MapThread[(#2 -> #1) &,
          {x["Parameters Values"], x["Parameters Names"]}, 1];
        x["New Parameters:2"] = (Flatten @ (Normal @
          DeleteCases[Association[x["New Parameters:1"]], _Integer]));
        xNTestParameters = Union[xNTestParameters, N[x["New Parameters:2"]]];
        x["New Parameters"] = Union[
          x["New Parameters"],
          (Reverse /@ x["New Parameters:2"]),
          (Reverse /@ x["New Parameters:2"]) /. ((xA_ -> xB_) -> (-xA -> -xB))
          ];
        ];
      ] /@ x["Row Labels"];
    ];
  xSC = Association[xNC, "Matrix"-> (xNC["Matrix"] //. x["New Parameters"]),
    "Test Parameters" -> xNTestParameters];
  xSC
  ]


LSLinearizedOrthogonalComplement[xLinearizedJacobian_Association, xIndependentVariables_List,
  xCoordinatesReplacements_: {}, xSymmetry_: Automatic, xTestParameters_List: {}, xNZero_Rational:1 10^-5] :=
  Module[{x, xE, xLSOC, xCoordinates, xLinearizedJacobianCoefficients,
```

```
xNTestParameters, xNA1, xNC1, xNCq, xSC1, xSCq},

{xLinearizedJacobianCoefficients,xCoordinates} =
  SMatrixCoefficientArrays[xLinearizedJacobian];
xNTestParameters = Union[
  xTestParameters,
  If[xSymmetry === Automatic,
    (#-> RandomReal[1])& /@ (GetAllVariables[(Flatten @ (Union @@ ( Normal @
      (#["Matrix"])& /@ xLinearizedJacobianCoefficients))) //. xTestParameters]),
    xSymmetry["Function"] /@ (GetAllVariables[(Flatten @ (Union @@ ( Normal @
      (#["Matrix"])& /@ xLinearizedJacobianCoefficients))) //. xTestParameters])
    ]
  ];

xNA1 = SReplaceRepeated[xLinearizedJacobianCoefficients[1], xNTestParameters];
xNC1 = LSNumericalOrthogonalComplement[xNA1, xIndependentVariables];
xNC1 = AppendTo[xNC1, "Matrix" -> Round[xNC1["Matrix"], xNZero]];
x["Column␣Labels"] = xNC1["Column␣Labels"] //. SymbolReplacements;
x["Row␣Labels"]= xNC1["Row␣Labels"] //. SymbolReplacements;

xE = 1 10^-3;
xNCq = Association[
  (#->SAssemble[
    (+1/(2 xE)) ~SDot~ SAssemble[ LSNumericalOrthogonalComplement[
      SAssemble[ xNA1, (+xE) ~SDot~ SReplaceRepeated[ xLinearizedJacobianCoefficients[#],
        xNTestParameters]], xIndependentVariables],
      (-1) ~SDot~ xNC1],
    (-1/(2 xE)) ~SDot~ SAssemble[ LSNumericalOrthogonalComplement[
```

```
        SAssemble[ xNA1, (-xE) ~SDot~ SReplaceRepeated[xLinearizedJacobianCoefficients[#],
          xNTestParameters]], xIndependentVariables],
        (-1) ~SDot~ xNC1]
      ]
    )& /@ xCoordinates
    ];

  (xNCq[#] = AppendTo[xNCq[#], "Matrix" -> Round[xNCq[#]["Matrix"], xNZero]])& /@ xCoordinates;

  x["New␣Parameters"]={};
  If[xSymmetry === Automatic,
    (*-TRUE-*)
    Function[xRowLabel,
      x["Row␣Number"] = First @ (Flatten @ Position[x["Row␣Labels"], xRowLabel]);
      x["Parameters␣Values"] = Flatten @ (Join[Part[xNC1["Matrix"], x["Row␣Number"]],
        Join @@ ((Part[xNCq[#]["Matrix"],x["Row␣Number"]])& /@ xCoordinates)]);
      x["Parameters␣Names"] = Flatten @ (Join[
        ((Function[{xColumnLabel}, Subscript[OverBar[\[CapitalDelta]],1,xRowLabel,xColumnLabel]]) /@
          x["Column␣Labels"]),
        Join @@ (
          ((Function[{xColumnLabel}, Subscript[OverBar[\[CapitalDelta]],#,xRowLabel,xColumnLabel]]) /@
            x["Column␣Labels"])& /@ (xCoordinates //. SymbolReplacements)
          )
        ]);
      x["New␣Parameters:1"] = MapThread[(#2-> #1)&, {x["Parameters␣Values"], x["Parameters␣Names"]}, 1];
```

```
    x["New␣Parameters:2"] = (Flatten @ (Normal @ DeleteCases[Association[x["New␣Parameters:1"]], _Integer]));
    xNTestParameters = Union[xNTestParameters, N[x["New␣Parameters:2"]]]];
    x["New␣Parameters"] = Union[
      x["New␣Parameters"],
      (Reverse /@ x["New␣Parameters:2"]),
      (Reverse /@ x["New␣Parameters:2"]) /. ((xA_->xB_ )->(-xA->-xB ))
      ];
    ]/@ x["Row␣Labels"],
  (*-FALSE-*)
  Function[{xRowLabel},
    If[Intersection[xSymmetry["Secondary"],
      Flatten @ (Characters /@ Select[xRowLabel /. {Subscript[xV_, xS__] -> {xV, xS}, xV_ -> {xV}}, StringQ])
        ] === {}
      (* Not @ And[
      StringQ[Quiet @ Last[xRowLabel]],
      MemberQ[xSymmetry["Secondary"], StringTake[Last[xRowLabel], -1]]
      ] *),
      x["Row␣Number"] = First @ (Flatten @ Position[x["Row␣Labels"], xRowLabel]);
      x["New␣Subscript"] = If[Intersection[xSymmetry["Primary"],
        Flatten @ (Characters /@ Select[xRowLabel /. {Subscript[xV_, xS__] -> {xV, xS}, xV_ -> {xV}}, StringQ])
          ] === {}
        (* Quiet @ (StringTake[Last[xRowLabel], -1] === xSymmetry["Primary"]) *),
        xRowLabel,
        Subscript @@ (If[StringQ[#], StringReplace[#, (# -> "")& /@ xSymmetry["Primary"]], #] & /@
          (xRowLabel /. (Subscript[xV_, xS__] -> {xV, xS})))
        ];
```

```
        x["Parameters␣Values"] = Flatten @ (Join[
          Part[xNC1["Matrix"], x["Row␣Number"]],
          Join @@ ((Part[xNCq[#]["Matrix"], x["Row␣Number"]]) & /@ xCoordinates)
          ]);
        x["Parameters␣Names"] = Flatten @ (Join[
          (Function[{xLabel},
            Subscript[OverBar[\[CapitalDelta]], 1, x["New␣Subscript"], xLabel]
            ]) /@ x["Column␣Labels"],
          Join @@ (((
            Function[{xLabel},
              Subscript[OverBar[\[CapitalDelta]], #, x["New␣Subscript"], xLabel]
              ]) /@ x["Column␣Labels"]
            ) & /@ (xCoordinates //. SymbolReplacements))
          ]);
        x["New␣Parameters:1"] = MapThread[(#2 -> #1) &,
          {x["Parameters␣Values"], x["Parameters␣Names"]}, 1];
        x["New␣Parameters:2"] = (Flatten @ (Normal @
          DeleteCases[Association[x["New␣Parameters:1"]], _Integer]));
        xNTestParameters = Union[xNTestParameters, N[x["New␣Parameters:2"]]];
        x["New␣Parameters"] = Union[
          x["New␣Parameters"],
          (Reverse /@ x["New␣Parameters:2"]),
          (Reverse /@ x["New␣Parameters:2"]) /. ((xA_ -> xB_) -> (-xA -> -xB))
          ];
        ];
      ] /@ x["Row␣Labels"];
    ];

  xSC1 = Association[xNC1, "Matrix"-> (xNC1["Matrix"] //. x["New␣Parameters"])];
```

```
  (xSCq[#] = Association[xNCq[#], "Matrix"-> (xNCq[#]["Matrix"] //. x[
     "New␣Parameters"])])& /@ xCoordinates;
  xLSOC = SAssemble[xSC1, Inner[#1 ~SDot~ #2&, xCoordinates, xSCq /@
     xCoordinates, SAssemble]];
  xLSOC["Test␣Parameters"] = xNTestParameters;
  xLSOC
  ]
```

`LSLinearizedOrthogonalComplement` provides an symbolic expression for the linearized form of the orthogonal complement of a non-linear Jacobian matrix. The syntax for this function is `LSLinearizedOrthogonalComplement[J,V,R,C,Z,P]`:

- `J` is an `Association` element representing a non-linear Jacobian matrix.
- `V` is a `List` of the variables among the `"Column␣Labels"` of `J` that are considered as independent.
- `R` is a `List` element consisting of replacement rules for the reference values of the generalized variables in the expression of `J` (*optional argument whose default value is an empty* `List`).
- `C` is a `List` element consisting of replacement rules for the linearized expressions of some of the generalized variables in the expression of `J` (*optional argument whose default value is an empty* `List`).
- `Z` is a rational number expressing the precision of the numerical algorithms present in the function; numbers whose difference is less than `Z` are considered as equal during the execution of the algorithm (*optional argument whose default value is* $1 \cdot 10^{-5}$).
- `P` is a `List` element consisting of replacement rules for the values of some of the parameters in the expression of `J` (*optional argument whose default value is an empty* `List`).

The output of this function is a `List` element `{A,C,T}` in which:

- `A` is an `Association` element representing the symbolic linearized expression of `J`.
- `C` is an `Association` element representing the symbolic linearized expression of an orthogonal complement of `J`.

- T is a List element consisting of replacement rules for the test values (i.e., random or prescribed values used in the algorithm for obtaining the expression of C) of the parameters of the symbolic expression of J.

3.9 Linearization procedures

```
LinearExpansion[xE_] = {
 Derivative[2][Subscript[Subscript[xX_, xId__], xId2__]][t] ->
  Superscript[Subscript[Subscript[Overscript[xX, ".."], xId], xId2],
     \[EmptySmallCircle]]
  + xE Derivative[2][Subscript[Subscript[xX, xId], xId2]][t],
 Derivative[1][Subscript[Subscript[xX_, xId__], xId2__]][t] ->
  Superscript[Subscript[Subscript[Overscript[xX, "."], xId], xId2], \[
     EmptySmallCircle]]
  + xE Derivative[1][Subscript[Subscript[xX, xId], xId2]][t],
 Derivative[xD_][Subscript[Subscript[xX_, xId__], xId2__]][t] /; (xD >
    2) ->
  Superscript[Subscript[Subscript[Superscript[xX, "(" <> (ToString @
     xD) <> ")"], xId], xId2], \[EmptySmallCircle]]
  + xE Derivative[xD][Subscript[Subscript[xX, xId], xId2]][t],
 Subscript[Subscript[xX_, xId__], xId2__][t] ->
  Superscript[Subscript[Subscript[xX, xId], xId2], \[EmptySmallCircle
     ]]
  + xE Subscript[Subscript[xX, xId], xId2][t],
 Derivative[2][Subscript[xX_, xId__]][t] ->
  Superscript[Subscript[Overscript[xX, ".."], xId], \[EmptySmallCircle
     ]]
  + xE Derivative[2][Subscript[xX, xId]][t],
 Derivative[1][Subscript[xX_, xId__]][t]->
  Superscript[Subscript[Overscript[xX, "."], xId], \[EmptySmallCircle
     ]]
  + xE Derivative[1][Subscript[xX, xId]][t],
 Derivative[xD_][Subscript[xX_, xId__]][t] /; (xD > 2) ->
  Superscript[Subscript[Overscript[xX, "(" <> (ToString @ xD) <> ")"],
      xId], \[EmptySmallCircle]]
```

```
    + xE Derivative[xD][Subscript[xX, xId]][t],
  Subscript[xX_, xId__][t] ->
    Superscript[Subscript[xX, xId], \[EmptySmallCircle]]
    + xE Subscript[xX, xId][t],
  Derivative[2][Subscript[xX_, xId__]][t] ->
    Superscript[Subscript[Overscript[xX, ".."], xId], \[EmptySmallCircle]]
    + xE Derivative[2][Subscript[xX, xId]][t],
  Derivative[1][Subscript[xX_, xId__]][t] ->
    Superscript[Subscript[Overscript[xX, "."], xId], \[EmptySmallCircle]]
    + xE Derivative[1][Subscript[xX, xId]][t],
  Derivative[xD_][Subscript[xX_, xId__]][t] /; (xD > 2) ->
    Superscript[Subscript[Overscript[xX, "(" <> (ToString @ xD) <> ")"], xId], \[EmptySmallCircle]]
    + xE Derivative[xD][Subscript[xX, xId]][t],
  Subscript[xX_, xId__][t] ->
    Superscript[Subscript[xX, xId], \[EmptySmallCircle]]
    + xE Subscript[xX, xId][t],
  Derivative[2][xX_][t] ->
    Superscript[Overscript[xX, ".."], \[EmptySmallCircle]]
    + xE Derivative[2][xX][t],
  Derivative[1][xX_][t] ->
    Superscript[Overscript[xX, "."], \[EmptySmallCircle]]
    + xE Derivative[1][xX][t],
  Derivative[xD_][xX_][t] /; (xD > 2) ->
    Superscript[Overscript[xX, "(" <> (ToString @ xD) <> ")"], \[EmptySmallCircle]]
    + xE Derivative[xD][xX][t],
  xX_[t] ->
    Superscript[xX, \[EmptySmallCircle]]
    + xE xX[t]
  };

Linearize[xA_Association, xReferenceMotion_: {}] :=
```

```
 SApply[((Series[(((# /. LinearExpansion[xE]) /. xReferenceMotion) /. {
     Superscript[xX_,\[EmptySmallCircle]]-> 0}),
   {xE,0,1}] // Normal) /. {xE-> 1})&, xA]
(*  Association[
   "Matrix"-> (Series[(((xA["Matrix"] /. LinearExpansion[xE]) /.
       xReferenceMotion) /.
     {Superscript[xX_,\[EmptySmallCircle]]-> 0}),
     {xE,0,1}] // Normal) /. {xE-> 1},
   "Row Labels"-> xA["Row Labels"],
   "Column Labels"-> xA["Column Labels"]
   ]
 *)
Linearize[xL_, xReferenceMotion_: {}] :=
 (Series[(((xL/.LinearExpansion[xE]) /. xReferenceMotion) /.
   {Superscript[xX_,\[EmptySmallCircle]]-> 0}),
   {xE,0,1}] // Normal) /. {xE-> 1};
```

`Linearize` obtains the linearized version of an expression (given either by a `List` or by an `Association` element) with respect to some reference values set for its generalized variables. Two syntaxes are admissible for this function:

- `Linearize[E]`: linearizes the expression `E` assuming that the reference values for all its variables are null.
- `Linearize[E,R]`: linearizes the expression `E` with respect to the reference values `R` (which is a list of rules similar to the outputs of function `ReferenceMotion`).

4 Rotation and homogeneous transformations

4.1 Rotation transformation

```
Rotation = Function @ Module[{x},
 x["TransformList"] = List[##] /. {
   "x" -> (RotationTransform[#, {1,0,0}]&),
   "y" -> (RotationTransform[#, {0,1,0}]&),
   "z" -> (RotationTransform[#, {0,0,1}]&),
   "X" -> (RotationTransform[#, {1,0,0}]&),
```

```
    "Y" -> (RotationTransform[#, {0,1,0}]&),
    "Z" -> (RotationTransform[#, {0,0,1}]&)
    };
  Function[(TransformationMatrix @ (Simplify @ Inner[(#1 @ #2)&, x["
     TransformList"], List[##], Dot]))[[1;;3,1;;3]]
  ];

HomogToRot = #[[1;;3,1;;3]]&;

QuatToRot = {
  {#1^2-#2^2-#3^2+#4^2,2 #1 #2-2 #3 #4,2 #1 #3+2 #2 #4},
  {2 #1 #2+2 #3 #4,-#1^2+#2^2-#3^2+#4^2,2 #2 #3-2 #1 #4},
  {2 #1 #3-2 #2 #4,2 #2 #3+2 #1 #4,-#1^2-#2^2+#3^2+#4^2}
  }&;
```

$\texttt{Rotation[}\mathbf{e}_1, \ldots, \mathbf{e}_r\texttt{][}\theta_1, \ldots, \theta_r\texttt{]}$ gives the transformation matrix associated to sucessive rotations around the axes $\mathbf{e}_1, \ldots, \mathbf{e}_r$ (being $\theta_1, \ldots, \theta_r$ the corresponding rotation angles). In this syntax, an axis $\mathbf{e}_k$ can be defined either by a `List` element representing the three Cartesian coordinates of a vector aligned to the axis of rotation in the local basis coordinates or by a `String` element `"x"`, `"y"` or `"z"` whenever any of the canonical local axis is the corresponding axis of rotation.

4.2 Homogeneous transformation

```
Homogeneous = Function @ Module[{x},
  x["TransformList"] = List[##] /. {
    "Rx" -> (RotationTransform[#, {1,0,0}]&),
    "Ry" -> (RotationTransform[#, {0,1,0}]&),
    "Rz" -> (RotationTransform[#, {0,0,1}]&),
    "R"[xVector_] -> (RotationTransform[#, xVector]&),
    "Tx" -> (TranslationTransform[# {1,0,0}]&),
    "Ty" -> (TranslationTransform[# {0,1,0}]&),
    "Tz" -> (TranslationTransform[# {0,0,1}]&),
    "T"[xVector_] -> (TranslationTransform[# xVector]&)
    };
```

```
Function[TransformationMatrix @ (Simplify @ Inner[(#1 @ #2)&, x["TransformList"], List[##], Dot])]
];
```

`Homogeneous[H`$_1$`, ..., H`$_r$`][`$\xi_1, \ldots, \xi_r$`]` gives the homogeneous transformation matrix associated to sucessive rotations or translations `H`$_1$`, ..., H`$_r$ (being $\xi_1, \ldots, \xi_r$ the corresponding rotation angles or displacements). In this syntax, `H`$_k$ can be defined either a rotation `"R"[`$\mathbf{e}_k$`]` around an axis defined by $\mathbf{e}_k$ or a translation `"T"[`$\mathbf{e}_k$`]` in the direction of $\mathbf{e}_k$ (being $\mathbf{e}_k$ a `List` element representing the three Cartesian coordinates of a vector in the local basis coordinates). When the rotation is around a canonical local axis, the following syntaxes are allowed for the `H`$_k$: `"Rx"`, `"Ry"` or `"Rz"`. Analogously, when a translation is in the directions of a canonical local axis, the following syntaxes are allowed for the `H`$_k$: `"Tx"`, `"Ty"` or `"Tz"`.

4.3 Angular velocity

```
SkewToVec = If[ And @@ (Flatten @ PossibleZeroQ[# + Transpose[#]]), {#[[3,2]], #[[1,3]], #[[2,1]]}]&;
VecToSkew = {{0, -#[[3]], #[[2]]}, {#[[3]], 0, -#[[1]]}, {-#[[2]], #[[1]], 0}}&;

AngularVelocity[xRotationMatrix_List] /; (Dimensions[xRotationMatrix]==={3,3}) :=
  Simplify @ (SkewToVec @ ((Transpose @ xRotationMatrix).D[xRotationMatrix, t]))
```

`SkewToVec` converts any 3×3 skew-symmetric `List` element representing a matrix into a 3 entries `List`. `VecToSkew` is its corresponding inverse function.

`AngularVelocity` obtains the angular velocity, in terms of local basis components (3 entries `List`), given the corresponding 3×3 `List` element representing a rotation transformation.

5 Plotting and visualization

5.1 General options

```
SetOptions[Plot, BaseStyle -> {FontFamily -> "Arial", FontSize -> 16}];
SetOptions[Plot3D, BaseStyle -> {FontFamily -> "Arial", FontSize ->
    14}];
SetOptions[ParametricPlot, BaseStyle -> {FontFamily -> "Arial", FontSize
     -> 16}];
SetOptions[ParametricPlot3D, BaseStyle -> {FontFamily -> "Arial",
    FontSize -> 14}];
SetOptions[ListPlot, BaseStyle -> {FontFamily -> "Arial", FontSize ->
    16}];
```

Package MathMatrixPack sets the `FontFamily` and `FontSize` for the following built-in plot functions:

- `Plot`: Arial, 16
- `Plot3D`: Arial, 14
- `ParametricPlot`: Arial, 16
- `ParametricPlot3D`: Arial, 14
- `ListPlot`: Arial, 16

5.2 Custom plot

```
Style8 = {
  {Hue[0.6, 1, 1], Thickness[0.005]},
  {Hue[0.3, 1, 1], Thickness[0.006], Dashed},
  {Hue[1, 1, 1], Thickness[0.007], Dotted},
  {Hue[0.1, 1, 1], Thickness[0.005]},
  {Hue[0.9, 1, 1], Thickness[0.006], Dashed},
  {Hue[0.5, 1, 1], Thickness[0.007], Dotted},
  {Hue[0.2, 1, 1], Thickness[0.005]},
  {Hue[0.8, 1, 1], Thickness[0.006], Dashed}
  };

SPlot[xExpression_, xInterval_, xFrameLabel_ : {}, xLegend_ : {},
    xPlotLabel_String : "", xScale_ : 1.15] :=
```

```
Module[{xStyle = Style8},
  TableForm[ {
    Plot[
      xExpression,
      xInterval,
      PlotStyle -> Style8,
      PlotRange -> Full,
      Frame -> True,
      GridLines -> Automatic,
      ImageSize -> xScale {500, 300},
      FrameLabel -> xFrameLabel,
      PlotLabel -> xPlotLabel
      ],
    Graphics[ {
      Black,
      Directive[FontFamily -> "Arial", FontSize -> 16],
      MapIndexed[ Text[#1, {10 (First[#2] - 1) + 6, 0}] &, xLegend],
      MapIndexed[ Join[
        Last[xStyle = RotateLeft @ xStyle],
        {Line[{{10 (First[#2] - 1), 0}, {10 (First[#2] - 1) + 3, 0}}]}
        ] &,
        xLegend
        ]
      },
    ImageSize -> 1.15 {500, 30}]}
    ]
  ];
```

`SPlot` is a customized version of the built-in function `Plot` for showing in the same frame up to 8 plots with their respective legends. The corresponding list of styles used in this plot are set in the `List` element `Style8`. `SPlot` syntax requires 5 arguments:

- The first argument must be a `List` of functions to be plot.
- The second argument must be a `List` of three elements: the first is the symbol denoting the independent variable, and the second and the third defining the range of this variable.

- The third argument is a `List` of 2 `String` elements representing representing the labels of the axes.
- The fourth argument is the title of the plot.
- The fifth argument is a `List` of legend labels.

For example, consider the following usage of the function:

```
SPlot[Sin[# t] & /@ #, {t, 0, Pi/2}, {"t", "Sin(nt)"},
  "Sin(nt)␣for␣several␣values␣of␣n", #] & @ Range[8]
```

The corresponding output is presented in Figure 1.

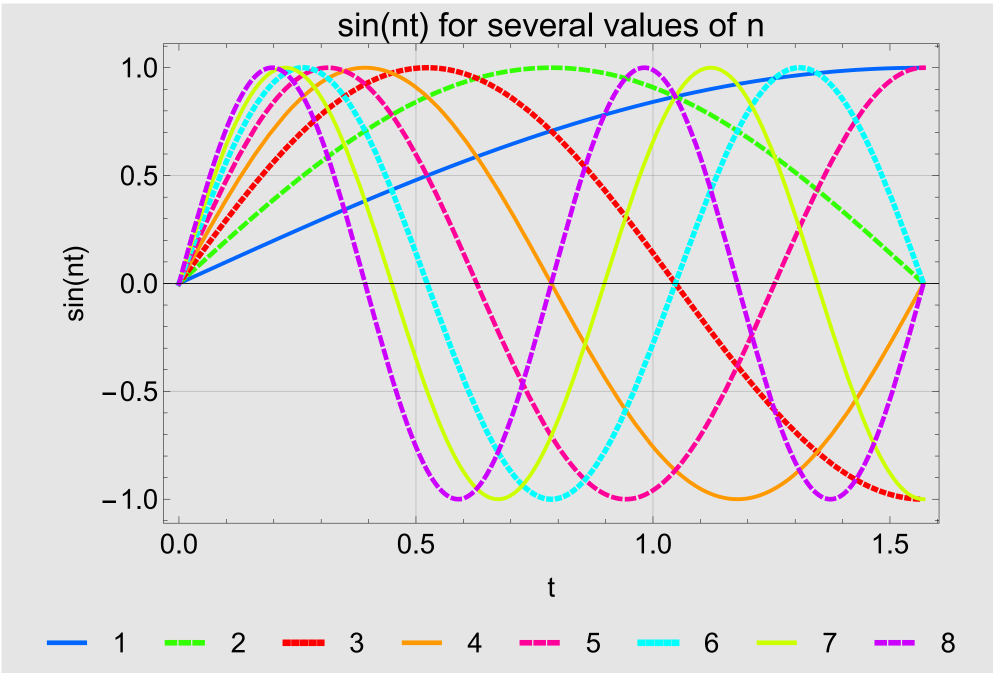

Figure 1: Example of output of the function `SPlot`

5.3 SMatrixPlot

```
SMatrixPlot[xA_Association] /; KeyExistsQ[xA, "Column␣Labels"] :=
    MatrixPlot[
  xA["Matrix"],
  FrameTicks -> {
```

```
    Transpose[{Range @@ Dimensions @ xA["Row Labels"], xA["Row Labels"]}],
    Transpose[{Range @@ Dimensions @ xA["Column Labels"], xA["Column Labels"]}]
    } /. SymbolReplacements,
  ColorFunction -> (RGBColor[
    (0.00130 (1 - #) + 0.99985 #) (2 # - 1)^2,
    (0.35656 (1 - #) + 0.085864 #) (2 # - 1)^2,
    (0.56796 (1 - #)) (2 # - 1)^2,
    0.13 + (2 # - 1)^2
    ] &),
  ColorRules -> {xN_ /; Not @ NumberQ[xN] -> RGBColor[0.63521, 0.99995, 0.19208]}
  ]

SMatrixPlot[xA_Association] := MatrixPlot[
  Transpose @ {xA["Matrix"]},
  FrameTicks -> {
    Transpose[{Range @@ Dimensions @ xA["Row Labels"], xA["Row Labels"]}],
    Transpose[{Range @@ Dimensions @ {""}, {""}}]
    } /. SymbolReplacements,
  ColorFunction -> (RGBColor[
    (0.00130 (1 - #) + 0.99985 #) (2 # - 1)^2,
    (0.35656 (1 - #) + 0.085864 #) (2 # - 1)^2,
    (0.56796 (1 - #)) (2 # - 1)^2,
    0.13 + (2 # - 1)^2
    ] &),
  ColorRules -> {xN_ /; Not @ NumberQ[xN] -> RGBColor[0.63521, 0.99995, 0.19208]}
  ]
```

`SMatrixPlot` extends the application of the built-in functions `MatrixPlot` to matrices given by `Association` elements.

Index