```
BeginPackage["QuantumGroups`Utilities`MatrixWrapper`",
  {"QuantumGroups`Utilities`Debugging`"}];
```

```
OnesMatrix;
ZeroesMatrix;
ZeroMatrixQ;
NonZeroMatrixQ;
Matrix;
MatrixData;
identityMatrix;
MatrixKroneckerProduct;
BlockDiagonalMatrix;
AppendRows;
AppendColumns;
MatrixInverse;
PrepareInverse;
InterpolationInverseThreshold;
```

```
Begin["`Private`"];
```

```
If[$VersionNumber < 6,
 UnitVector[n_Integer, k_Integer] /; 1 ≤ k ≤ n := Table[If[i == k, 1, 0], {i, 1, n}]
]
UnitVectorQ[v_?VectorQ] := Complement[v, {0, 1}] == {} ∧ Count[v, 1] == 1
```

```
OnesMatrix[n_, m_] := Matrix[n, m, Table[1, {n}, {m}]]
```

```
ZeroesMatrix[n_, m_] := Matrix[n, m, Table[0, {n}, {m}]]
ZeroesMatrix[n_] := ZeroesMatrix[n, n]
```

```
ZeroMatrixQ[Matrix[0, _, _]] := True
ZeroMatrixQ[Matrix[_, 0, _]] := True
ZeroMatrixQ[Matrix[_, _, data_]] := And @@ (Together[#] === 0 &) /@ Flatten[data]
```

```
NonZeroMatrixQ[m_] := ! ZeroMatrixQ[m]
```

We can construct Matrix objects from 2-dimensional arrays, in the usual way

```
Matrix[data_?MatrixQ] := With[{d = Dimensions[data]}, Matrix[d[[1]], d[[2]], data]]
```

There's also a special constructor for matrices with 0 rows or 0 columns, because there's no need to specify the data explicitly

```
Matrix[0, c_] := Matrix[0, c, {}]
Matrix[r_, 0] := Matrix[r, 0, Table[{}, {r}]]
```

```
MatrixData[Matrix[_, _, data_]] := data
```

```
identityMatrix[n_] := Matrix[n, n, IdentityMatrix[n]]
```

```
Matrix /: Dimensions[Matrix[r_, c_, _]] := {r, c}
```

```
Matrix /: Part[Matrix[_, _, data_], p__] := data[[p]]
```

```
Matrix /: MatrixForm[Matrix[_, _, data_]] := MatrixForm[data]
```

```
Matrix /: m_Matrix.{} := {}
Matrix /: m_Matrix.v_?VectorQ /; (Dimensions[m][[2]] == Length[v]) :=
 If[Dimensions[m][[1]] == 0, {}, MatrixData[m].v]
Matrix /: m1_Matrix.m2_?MatrixQ /; (Dimensions[m1][[2]] == Length[m2]) := MatrixData[m1].m2
```

```
Matrix /: Together[Matrix[r_, c_, data_]] := Matrix[r, c, Together[data]]
```

```
Matrix /: Transpose[Matrix[0, c_, _]] := Matrix[c, 0]
Matrix /: Transpose[Matrix[r_, 0, _]] := Matrix[0, r]
Matrix /: Transpose[Matrix[r_, c_, data_]] := Matrix[c, r, Transpose[data]]
```

```
Matrix /: Inverse[Matrix[0, 0, _]] := Matrix[0, 0]
Matrix /: Inverse[Matrix[r_, r_, data_]] := Matrix[r, r, MatrixInverse[data]]
```

```
Matrix /: Det[Matrix[0, 0, _]] := 1
Matrix /: Det[Matrix[r_, r_, data_]] := Det[data]
Matrix /: Tr[Matrix[0, 0, _]] := 0
Matrix /: Tr[Matrix[r_, r_, data_]] := Tr[data]
```

```
Matrix /: NullSpace[m_Matrix, opts___] := NullSpace[MatrixData[m], opts]
```

```
Matrix /: AppendRows[Matrix[r_, 0, _], Matrix[r_, c_, data_]] := Matrix[r, c, data]
Matrix /: AppendRows[Matrix[r_, c_, data_], Matrix[r_, 0, _]] := Matrix[r, c, data]
Matrix /: AppendRows[Matrix[0, c1_, _], Matrix[0, c2_, _]] := Matrix[0, c1 + c2]
Matrix /: AppendRows[Matrix[r_, c1_, data1_], Matrix[r_, c2_, data2_]] :=
 Matrix[r, c1 + c2, Join[data1, data2, 2]]
```

```
Matrix /: AppendRows[m1 : Matrix[0, _, _], m2 : (Matrix[0, _, _] ..)] :=
 ZeroesMatrix[0, Plus @@ ({m1, m2}[[All, 2]])]
Matrix /: AppendRows[m1 : Matrix[r_, 0, _], m2 : (Matrix[r_, 0, _] ..)] :=
 ZeroesMatrix[r, 0]
Matrix /: AppendRows[m1 : Matrix[r_, _, _], m2 : (Matrix[r_, _, _] ..)] :=
 Matrix[r, Plus @@ ({m1, m2}[[All, 2]]), Join[##, 2] & @@ (MatrixData /@ {m1, m2})]
```

```
Matrix /: AppendRows[m1_Matrix] := m1
(*Matrix/:AppendRows[m1_Matrix,m2__Matrix]:=AppendRows[m1,AppendRows[m2]]*)
```

```
Matrix /: AppendColumns[Matrix[0, c_, _], Matrix[r_, c_, data_]] := Matrix[r, c, data]
Matrix /: AppendColumns[Matrix[r_, c_, data_], Matrix[0, c_, _]] := Matrix[r, c, data]
Matrix /: AppendColumns[Matrix[r1_, 0, _], Matrix[r2_, 0, _]] := Matrix[r1 + r2, 0]
Matrix /: AppendColumns[Matrix[r1_, c_, data1_], Matrix[r2_, c_, data2_]] :=
 Matrix[r1 + r2, c, Join[data1, data2]]
```

```
Matrix /: AppendColumns[m1 : Matrix[_, 0, _], m2 : (Matrix[_, 0, _] ..)] :=
 ZeroesMatrix[Plus @@ ({m1, m2}[[All, 1]]), 0]
Matrix /: AppendColumns[m1 : Matrix[0, c_, _], m2 : (Matrix[0, c_, _] ..)] :=
 ZeroesMatrix[0, c]
Matrix /: AppendColumns[m1 : Matrix[_, c_, _], m2 : (Matrix[_, c_, _] ..)] :=
 Matrix[Plus @@ ({m1, m2}[[All, 1]]), c, Join @@ (DeleteCases[MatrixData /@ {m1, m2}, {}])]
```

```
Matrix /: AppendColumns[m1_Matrix] := m1
(*Matrix/:AppendColumns[m1_Matrix,m2__Matrix]:=AppendColumns[m1,AppendColumns[m2]]*)
```

```
Matrix /: Dot[m1_Matrix, m2__Matrix] /; (! MemberQ[Flatten[Dimensions /@ {m1, m2}], 0] ∧
    Most[Last /@ Dimensions /@ {m1, m2}] == Rest[First /@ Dimensions /@ {m1, m2}]) :=
 Matrix[Dimensions[{m1, m2}[[1]]][[1]], Dimensions[{m1, m2}[[-1]]][[2]],
  Dot @@ (MatrixData /@ {m1, m2})]
```

```
Matrix /: Dot[m1_Matrix, m2__Matrix] /; (MemberQ[Flatten[Dimensions /@ {m1, m2}], 0]) :=
 ZeroesMatrix[Dimensions[{m1, m2}[[1]]][[1]], Dimensions[{m1, m2}[[-1]]][[2]]]
```

```
Matrix /: Plus[m1_Matrix, m2__Matrix] /; (SameQ[Dimensions /@ {m1, m2}]) :=
 Matrix[Sequence @@ Dimensions[First[{m1, m2}]], Plus @@ (MatrixData /@ {m1, m2})]
```

```
Matrix /: α_ Matrix[j_, k_, data_] := Matrix[j, k, α data]
```

```
If[$VersionNumber ≥ 6, BlockMatrix = ArrayFlatten];
```

```
MatrixKroneckerProduct[Matrix[r1_, c1_, data1_], Matrix[r2_, c2_, data2_]] /;
  r1 > 0 ∧ r2 > 0 ∧ c1 > 0 ∧ c2 > 0 :=
 Matrix[r1 r2, c1 c2, BlockMatrix[Outer[Times, data1, data2]]]
```

```
MatrixKroneckerProduct[Matrix[0, c1_, _], Matrix[_, c2_, _]] := Matrix[0, c1 c2]
MatrixKroneckerProduct[Matrix[_, c1_, _], Matrix[0, c2_, _]] := Matrix[0, c1 c2]
```

```
MatrixKroneckerProduct[Matrix[r1_, 0, _], Matrix[r2_, _, _]] := Matrix[r1 r2, 0]
MatrixKroneckerProduct[Matrix[r1_, _, _], Matrix[r2_, 0, _]] := Matrix[r1 r2, 0]
```

Careful here; tensor products with more than 256 factors will cause $RecursionLimit problems.

```
(*MatrixKroneckerProduct[a_,b_,c__]:=
 MatrixKroneckerProduct[MatrixKroneckerProduct[a,b],c]*)
```

```
MatrixKroneckerProduct[a_, b_, c__] := MatrixKroneckerProduct[
  MatrixKroneckerProduct @@ (Take[{a, b, c}, Floor[Length[{a, b, c}] / 2]]),
  MatrixKroneckerProduct @@ (Drop[{a, b, c}, Floor[Length[{a, b, c}] / 2]])]
```

Again, more than 256 blocks will cause problems.

```
BlockDiagonalMatrix[m1 : Matrix[r1_, c1_, _], m2 : Matrix[r2_, c2_, _]] := AppendColumns[
  AppendRows[m1, ZeroesMatrix[r1, c2]], AppendRows[ZeroesMatrix[r2, c1], m2]]
BlockDiagonalMatrix[] := Matrix[0, 0]
BlockDiagonalMatrix[m_Matrix] := m
(*BlockDiagonalMatrix[m1_,m2_,m3__]:=
 BlockDiagonalMatrix[BlockDiagonalMatrix[m1,m2],m3]*)
```

```
BlockDiagonalMatrix[m1_, m2_, m3__] := BlockDiagonalMatrix[
  BlockDiagonalMatrix @@ (Take[{m1, m2, m3}, Floor[Length[{m1, m2, m3}] / 2]]),
  BlockDiagonalMatrix @@ (Drop[{m1, m2, m3}, Floor[Length[{m1, m2, m3}] / 2]])]
```

```
InterpolationInverseThreshold = 21;
```

```
PrepareInverse[x_] := Null
```

```
If[10 > $VersionNumber ≥ 6,
 SquareMatrixQ[m_] := (MatrixQ[m] ∧ Dimensions[m][[1]] == Dimensions[m][[2]])]
```

```
MatrixInverse[m_] /; (SquareMatrixQ[m] ∨
    (Print["Warning: tried to take the inverse of a non-square matrix! ", m];
     False)) := If[Length[m] ≥ InterpolationInverseThreshold,
  RowOrderedInterpolationInverse[m], RowReductionInverse[m]]
```

```
RowReductionInverse[m_?SquareMatrixQ] := Module[{result},
  If[Length[m] ≥ 8,
   DebugPrint["Performing (built-in) row reduction on a matrix of size ", Length[m]]];
  result = Together[Inverse[m, Method → "OneStepRowReduction"]];
  If[Length[m] ≥ 8, DebugPrint["Finished row reduction"]];
  result
 ]
```

```
MatrixRowFactors[mat_?SquareMatrixQ] :=
 Module[{rowFactors, rowOrdering, n = Length[mat]},
  rowFactors = (PolynomialLCM @@ # &) /@ (Denominator[Together[#]] &) /@ mat;
  rowOrdering = UnitVector[n, #] & /@ Ordering[rowFactors mat];
  rowOrdering.DiagonalMatrix[rowFactors]
 ]
```

```
RowOrderedInterpolationInverse[mat_?SquareMatrixQ] :=
 Module[{rf = MatrixRowFactors[mat]},
  Simplify[InterpolationInverse[Expand[Together[rf.mat]]].rf]
 ]
```

```
InterpolationInverseLargestRequestSize = 0;
InterpolationInverseRequests = {};
```

```
recordInterpolationInverseRequest[mat_] :=
 If[Length[mat] ≥ InterpolationInverseLargestRequestSize,
  DebugPrint["New largest matrix! Size ", Length[mat]];
  AppendTo[InterpolationInverseRequests, mat];
  InterpolationInverseLargestRequestSize = Length[mat]]
```

```
InterpolationInverse[mat_?SquareMatrixQ] :=
 Module[{size, newMatrix, det, degree, n, abcissa, data, inverse},
  DebugPrint["Starting InterpolationInverse on a matrix of size ", Length[mat]];
  size = Length[mat];
  det = Together[Det[mat]];
  newMatrix = mat/det;
  degree = Min[Apply[Plus, Map[Max[Exponent[#, Global`q]] &, mat]],
    Apply[Plus, Map[Max[Exponent[#, Global`q]] &, Transpose[mat]]]];
  If[degree == 0,
   Global`interpolationInverseNoQExample = mat;
   If[And @@ (UnitVectorQ /@ mat),
    DebugPrint["... inverting a permutation matrix"];
    Return[IdentityMatrix[Length[mat]][[
      Ordering[Ordering[Plus @@ (Range[Length[mat]] mat)]]]]]
   ];
   DebugPrint["... it doesn't seem to involve q, so I'm just using Inverse"];
   Return[Inverse[mat, Method → OneStepRowReduction]]
  ];
  recordInterpolationInverseRequest[mat];
  DebugPrint["inverting matrix of size ", Length[mat], " by interpolation"];
  abcissa = {};
  n = Floor[-degree/2] + 1;
  While[Length[abcissa] < degree + 2,
   If[(det /. Global`q → n) =!= 0, abcissa = Append[abcissa, n]];
   n++
  ];
  If[size > 20, DebugPrint["Inverting numerical matrices:"]];
  data = Transpose[Table[If[size > 20, DebugPrint[i]];
     Inverse[newMatrix /. Global`q → abcissa[[i]]], {i, 1, Length[abcissa]}], {3, 1, 2}];
  If[size > 20, DebugPrint["Interpolating numerical matrices:"]];
  inverse = Table[If[j == 1 ∧ size > 20, DebugPrint[i]];
    Simplify[InterpolatingPolynomial[Transpose[{abcissa, data[[i, j]]}], Global`q]],
    {i, 1, size}, {j, 1, size}];
  DebugPrint["done"];
  Together[1/det inverse]
 ]
```

```
End[];
```

```
EndPackage[];
```