Quaternions

Package for symbolic calculations with quaternions

```
(*
   Quaternions package implements Hamilton's quaternion algebra.
   This package mimics original Mathematica Quaternions.m, however,
   it fixes various bugs and also allows symbolic transformations
   of quaternions to more extent than Mathematica's original.
   Author:  Mikica B Kocic
   Version: 0.4, 2012-04-22
*)
```

Begin Package

```
$Pre =.
```

```
BeginPackage[ "Quat`" ];
```

```
Unprotect[ Quat`Q ];
```

```
ClearAll[ "Quat`Private`*" ];
ClearAll[ "Quat`*" ];
```

Exported Symbols

```
Q::usage = "Q[w,x,y,z] represents the quaternion with real part w \
(also called scalar part) and imaginary part {x,y,z} (also called vector part)";
```

```
QQ::usage = "QQ[q] gives True if q is a quaternion, \
and False otherwise.";
```

```
ToQ::usage = "ToQ[expr] transforms expr into a quaternion object if at all possible.";
```

```
ScalarQ::usage = "ScalarQ[q] gives True if q is a scalar, and False otherwise.";
```

```
ToList::usage = "ToList[q] gives components of q in a list";
```

```
ToVector::usage = "ToVector[q] gives q as column vector matrix";
```

```
ToMatrix::usage = "ToMatrix[q] gives matrix representation of quaternion q";
```

```
Abs$Im::usage = "Abs$Im[q] gives the absolute value of the vector quaternion part of q.";
```

```
AdjustedSign$Im::usage = "AdjustedSign$Im[q] gives the Sign of the vector part of q, \
adjusted so its first non-zero part is positive.";
```

```
NonCommutativeMultiply::usage = "Implements non-commuatative quaternion multiplication.";
```

```
ToQ$AngleAxis::usage = "ToQ$AngleAxis[θ,{ux,uy,uz}] transforms axis u and angle θ into qua
```

```
ToQ::invargs = "ToQ: failed to construct Q from argument:\n args == `1`"
```

```
RotationMatrix4::usage = "RotationMatrix4[q] gives the 4×4 rotation matrix for a \
counterclockwise 3D rotation around the quaterion q"
```

```
QForm::usage = "QForm[q] prints with the elements of matrix or quaternion \
arranged in a regular array."
```

```
Assert::usage = "QForm[q] prints with the elements of matrix or quaternion \
arranged in a regular array."
```

Private Section

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

Operators

```
Subscript[ q_ ?QQ, n_Integer /; 1 ≤ n ≤ 4 ] := q[[n]]
```

```
AngleBracket[ q_ ?QQ ] := Re[q]
```

```
OverVector[ q_ ?QQ ] := q[[2;;4]] /. Q → List
```

```
OverHat[ q_ ?QQ ] := q / Abs[ q ]
```

```
SuperStar[ q_ ?QQ ] := Conjugate[ q ]
```

```
BracketingBar[ q_ ?QQ ] := Abs[ q ]
```

Utility Functions

Strip Output Label of Form Info

```
SetAttributes[ TimeIt, HoldAll ];
```

```
TimeIt[ expr_ ] :=
Module[
  { result = expr, out, form },
  If[ TrueQ[ MemberQ[ $OutputForms, Head[result] ] ],
    (* Then *) out = First[result]; form = "//" <> ToString[ Head[result] ],
    (* Else *) out = result; form = ""
  ];
  If[ out =!= Null,
    CellPrint[
      ExpressionCell[ result, "Output",
        CellLabelAutoDelete → True,
        CellLabel → StringJoin[ "Out[", ToString[$Line], "]:="]
      ]
    ];
    Unprotect[ Out ];
    Out[ $Line ] = out;
    Protect[ Out ];
    out (* needed for % *)
  ];
];
```

Assert truth of a logical statement (i.e. prove a theorem)

```
Assert[ statement_, params__: Null ] :=
Module[
  { result },
  (* Try to proove the statement... *)
  result = statement;
  If[ result,
    (* is true *)  Return@ Style[ "⊢ True ∎", Blue ],
    (* is false *) Return@ Style[ "False ☹", Red, Large, Bold ],
    (* is neither true or false *) Null (* continue... *)
  ];
  (* ... now, try even harder to proove the statement *)
  result = Simplify[ statement,params ];
  If[ result,
    (* is true *)  Return@ Style[ "⊢ True (after Simplify) ∎", Blue ],
    (* is false *) Return@ Style[ "False (after Simplify) ☹", Red, Large, Bold ],
    (* is neither true or false *) Null (* continue... *)
  ];
  (* ... now, try even harder to proove the statement *)
  result = FullSimplify[ statement, params ];
  If[ result,
    (* is true *)  Style[ "⊢ True (after FullSimplify) ∎", Blue ],
    (* is false *) Style[ "False (after FullSimplify) ☹", Red, Large, Bold ],
    (* is neither true or false *)
        Style[ "Indeterminate ☹", Darker[Red], Large, Bold ]
  ]
]
```

Transformation Rules

Scalar test

```
ScalarQ[ x_ ] := AtomQ[ x ] ∨ Length[ x ] === 0 ∨
  ( Length[ x ] =!= 0 ∧ ¬ListQ[ x ] ∧ Head[ x ] =!= Q ∧ Head[ x ] =!= Complex )
```

Quaternion test

```
QQ[ q_ ] :=
  Head[ q ] === Q ∧ Length[ q ] === 4 ∧
  And @@ ( ScalarQ /@ q )
```

Transform expressions into *Q* objects

```
ToQ[ w_ ?ScalarQ, { x_ ?ScalarQ, y_ ?ScalarQ, z_ ?ScalarQ } ] :=
    Q[ w, x, y, z ]
```

```
ToQ[ q_ ] := q /. {
    Q[ w_, x_, y_, z_ ] ⧴ Q[ w, x, y, z ],
    { x_ ?ScalarQ, y_ ?ScalarQ, z_ ?ScalarQ }  ⧴ Q[ 0, x, y, z ],
    { w_ ?ScalarQ, x_ ?ScalarQ, y_ ?ScalarQ, z_ ?ScalarQ } ⧴ Q[ w, x, y, z ],
    { {w_ ?ScalarQ}, {x_ ?ScalarQ}, {y_ ?ScalarQ}, {z_ ?ScalarQ} } ⧴ Q[ w, x, y, z ],
    Complex[ x_, y_ ] ⧴ Q[ x, y, 0, 0 ],
    Plus[ x_, Times[ Complex[ 0, 1 ], y_ ] ] ⧴ Q[ x, y, 0, 0 ],
    Times[ Complex[ 0, 1 ], x_ ] ⧴ Q[ 0, x, 0,  0 ],
    Times[ Complex[ 0, x_ ], y_ ] ⧴ Q[ 0, x y, 0, 0 ],
    x_ ?ScalarQ ⧴ Q[ x, 0, 0, 0],
    (* unknown list *)
    list_ ⧴ Module[ {}, Message[ ToQ::invargs, list ]; Abort[] ]
} // QSimplify
```

▼ Axis and angle into Q object

```
ToQ$AngleAxis[ θ_, { ux_, uy_, uz_ } ] := Q[
    Cos[θ/2], ux Sin[θ/2], uy Sin[θ/2], uz Sin[θ/2]
]
```

▼ To list

```
ToList[ q_ ?QQ ] := ( q /. Q→ List )
```

▼ Vector

```
ToVector[ q_ ?QQ ] := Transpose@{ q /. Q→ List }
```

▼ To matrix representation

```
ToMatrix[ Q[ w_, x_, y_, z_ ] ] := {
   {  w,  x,  y,  z },
   { -x,  w, -z,  y },
   { -y,  z,  w, -x },
   { -z, -y,  x,  w }
}
```

▼ To matrix form

```
Q /: MatrixForm[ q: Q[ __ ?ScalarQ ] ] := MatrixForm@ ToList@ q
```

▼ To paritioned matrix form (with divider lines)

```
QForm[ Q_ ?MatrixQ ] := DisplayForm@
  RowBox[{
    "(", "",
    GridBox[ Release[ Q ],
      RowSpacings → 1, ColumnSpacings → 1,
      RowAlignments → Baseline,
      ColumnAlignments → Center,
      GridBoxDividers → {
        "Columns" → { False, GrayLevel[0.84] },
        "Rows" → { False, GrayLevel[0.84] }
      }
    ],
    "", ")"
 }]
```

```
QForm[ q_ ?QQ ] := QForm[ { Release[ ToList@ q ] } ]
```

▼ To rotation matrix, Shoemake's form

```
Q /: RotationMatrix[ Q[ w_, x_, y_, z_ ] ] :=
{
   { 1 - 2( y^2 + z^2 ),  2 x y - 2 w z,      2 x z + 2 w y   },
   { 2 x y + 2 w z,       1 - 2( x^2 + z^2 ), 2 y z - 2 w x   },
   { 2 x z - 2 w y ,      2 w x+2 y z,        1 - 2( x^2 + y^2 ) }
}
```

```
Q /: RotationMatrix4[ Q[ w_, x_, y_, z_ ] ] :=
{
   { 1,  0,                  0,                  0                  },
   { 0,  1 - 2( y^2 + z^2 ), 2 x y - 2 w z,      2 x z + 2 w y      },
   { 0,  2 x y + 2 w z,      1 - 2( x^2 + z^2 ), 2 y z - 2 w x      },
   { 0,  2 x z - 2 w y ,     2 w x + 2 y z,      1 - 2( x^2 + y^2 ) }
}
```

Conjugate

```
Q /: Conjugate[ Q[ w_, x_, y_, z_ ] ]:=
    Q[ w, -x, -y, -z ]
```

Squared Norm

```
Q /: SqNorm[ q:  Q[ __ ?ScalarQ ] ] :=
    Plus @@ ( ( List @@ q )^2 )
```

Norm

```
Q /: Norm[ q: Q[ __ ?ScalarQ ] ]:=
    Sqrt[ SqNorm[ q ] ]
```

Abs

```
Q /: Abs[ q:  Q[ __ ?ScalarQ ] ] :=
    Sqrt[ SqNorm[ q ] ]
```

Round

```
Q /: Round[ q: Q[ __ ?ScalarQ ] ]:=
Module[
  {
    cent = Round /@ q;
    mid = ( Floor /@ q ) + Q[ 1/2, 1/2, 1/2, 1/2 ];
  },
  If[ SqNorm[ q - cent ] <= SqNorm[ q - mid ], cent, mid ]
]
```

Real Part

```
Q /: Re[ q: Q[ __ ?ScalarQ ] ] :=
    q[[1]]
```

Sign (returns Versor)

```
Q /: Sign[ q: Q[ __ ?ScalarQ ] ] :=
    Q[ 1, 0, 0, 0 ] /; Abs[ q ] == 0
```

```
Q /: Sign[ q: Q[ __ ?ScalarQ ] ] :=
    q / Abs[ q ]
```

Imaginary Part

```
Q /: Im[ q: Q[ __ ?ScalarQ ] ] :=
    { q[[2]], q[[3]], q[[4]] }
```

```
AdjustedSign$Im[ q_Complex | q_ ?ScalarQ ] := i
```

```
AdjustedSign$Im[ Q[ w_ ?ScalarQ, x_ ?ScalarQ, y_ ?ScalarQ, z_ ?ScalarQ ] ] :=
Which[
    x != 0,
        Sign[ x ] * Sign[ Q[ 0, x, y, z ] ],
    y != 0,
        Sign[ y ] * Sign[ Q[ 0, x, y, z ] ],
    z != 0,
        Sign[ z ] * Sign[ Q[ 0, x, y, z ] ],
    True,
        i (* ?abort? *)
]
```

```
Abs$Im[ Q[ w_ ?ScalarQ, x_ ?ScalarQ, y_ ?ScalarQ, z_ ?ScalarQ ] ]:=
    Sqrt[ x^2 + y^2 + z^2 ]
```

```
Abs$Im[ x_ ? NumericQ ] :=
    Im[ x ]
```

```
Sign$Im[ Q[ w_ ?ScalarQ, x_ ?ScalarQ, y_ ?ScalarQ, z_ ?ScalarQ ] ] :=
    Sign[ Q[ 0, x, y, z ] ]
```

Addition

```
Q /: Q[ w1_, x1_, y1_, z1_ ] + Q[ w2_, x2_, y2_, z2_ ] :=
    Q[ w1 + w2, x1 + x2, y1 + y2, z1 + z2 ] // QSimplify
```

```
Q /: Complex[ re_, im_ ] + Q[ w_, x_, y_, z_ ] :=
    Q[ w + re, x + im, y, z ] // QSimplify
```

```
Q /: λ_ ?ScalarQ + Q[ w_, x_, y_, z_ ] :=
    Q[ w + λ, x, y, z ]
```

Multiplication

```
Q /: λ_ ?ScalarQ * Q[ w_, x_, y_, z_ ]:=
    Q[ λ w, λ x, λ y, λ z ]
```

Non-commutative multiplication

```
Q /: Q[ w1_, x1_, y1_, z1_ ] ** Q[ w2_, x2_, y2_, z2_ ]:=
    Q[
        w1 w2 - x1 x2 - y1 y2 - z1 z2,
        w1 x2 + x1 w2 + y1 z2 - z1 y2,
        w1 y2 - x1 z2 + y1 w2 + z1 x2,
        w1 z2 + x1 y2 - y1 x2 + z1 w2
    ] // QSimplify
```

Non-commutative multiplication with complex numbers

```
Unprotect[ NonCommutativeMultiply ]
(*SetAttributes[ NonCommutativeMultiply, Listable ]*)
```

```
a_ ?ScalarQ ** b_ ?QQ := a * b
```

```
a_ ?QQ ** b_ ?ScalarQ := a * b
```

```
{ x_ ?ScalarQ, y_ ?ScalarQ, z_ ?ScalarQ } ** q_ ?QQ :=
    Q[ 0, x, y, z ] ** q
```

```
q_ ?QQ ** { x_ ?ScalarQ, y_ ?ScalarQ, z_ ?ScalarQ } :=
    q ** Q[ 0, x, y, z ]
```

```
{ w_ ?ScalarQ, x_ ?ScalarQ, y_ ?ScalarQ, z_ ?ScalarQ } ** q_ ?QQ :=
    Q[ w, x, y, z ] ** q

q_ ?QQ ** { w_ ?ScalarQ, x_ ?ScalarQ, y_ ?ScalarQ, z_ ?ScalarQ } :=
    q ** Q[ w, x, y, z ]

{ {w_ ?ScalarQ}, {x_ ?ScalarQ}, {y_ ?ScalarQ}, {z_ ?ScalarQ} } ** q_ ?QQ :=
    Q[ w, x, y, z ] ** q

q_ ?QQ ** { {w_ ?ScalarQ}, {x_ ?ScalarQ}, {y_ ?ScalarQ}, {z_ ?ScalarQ} } :=
    q ** Q[ w, x, y, z ]

( ( a_: 1 ) * Complex[ b_, c_ ] ) ** ( ( x_: 1 ) * Complex[ y_, z_ ] ) :=
    a b x y - a c x z + ( a c x y + a b x z ) ⅈ

( x_ + y_ ) ** a_ := ( x ** a ) + ( y ** a )
a_ ** (x_ + y_) := ( a ** x ) + ( a ** y )

Protect[ NonCommutativeMultiply ]
```

Math Functions

```
extfunc[ func_, a_, b_ ] :=
    Re[b] + Abs$Im[b] * AdjustedSign$Im[a]

Block[
    { extend, $Output = {} },

    extend[ foo_ ]:= (
        Unprotect[ foo ];
        Q /: foo[ a: Q[ __ ?ScalarQ ] ] :=
           extfunc[ foo, a, foo[ Re[a] + Abs$Im[a] * ⅈ ] ];
        Protect[ foo ];
    );

    extend /@ {
        Log,
        Cos, Sin, Tan, Sec, Csc, Cot,
        ArcCos, ArcSin, ArcTan, ArcSec, ArcCsc, ArcCot,
        Cosh, Sinh, Tanh, Sech, Csch, Coth,
        ArcCosh, ArcSinh, ArcTanh, ArcSech, ArcCsch, ArcCoth
    };
]
```

Exp e^q

```
Q /: Exp[ q: Q[ __ ?ScalarQ ] ] :=
    Exp[ Re[ q] ] *
    ( Cos[ Abs$Im[ q ] ] + Sin[ Abs$Im[ q ] ] * Sign$Im[ q] ) // QSimplify
```

Power

```
Q /: Power[ q: Q[ __ ?ScalarQ ], 0 ] :=
    1

Q /: Power[ q: Q[ __ ?ScalarQ ], 1 ] :=
    q

Q /: Power[ q: Q[ __ ?ScalarQ ], -1 ] :=
    Conjugate[q] ** ( 1 / SqNorm[q] )
```

```
Q /: Power[ q: Q[ __ ?ScalarQ ], n_ ] :=
    q ** Power[ q, n - 1 ] /; n > 1 ⋀ n ∈ Integers
```

```
Q /: Power[ q: Q[ __ ?ScalarQ ], n_ ] :=
    Power[ 1/q, -n ] /; n < 0 ⋀ n != -1
```

```
Q /: Power[ q: Q[ __ ?ScalarQ ], n_ ] :=
Module[
  {
    μ = Abs[ q ], (* modulus of {q1,q2,q3,q4} *)
    re = Re[ q ], (* scalar part re = {q1} *)
    im = Abs$Im[ q ], (* modulus of vector part im = {q2,q3,q4} *)
    θ, (* Angle between vector im and scalar re *)
    ϕ = 0, (* Angle between q2 and vector part {q2,q3,q4} *)
    γ = 0 (* Angle between q3 and subvector {q3,q4} *)
  },

  θ = If[ re =!= 0,
        (* Then *) ArcTan[ im / re ],
        (* Else *) π/2
      ];

  If[ im =!= 0,
  (* Then *)
     ϕ = ArcCos[ q[[2]] / im ];
     γ = If[ Sin[ϕ] =!= 0,
          (* Then *)
             ArcCos[ q[[3]] / ( im Sin[ϕ] ) ] Sign[ q[[4]] ],
             ( π/2 ) Sign[ q[[4]] ]
          ]
  ];

  Q[
     μ^n Cos[ n θ ] ,
     μ^n Sin[ n θ ] Cos[ϕ] ,
     μ^n Sin[ n θ ] Sin[ϕ] Cos[γ],
     μ^n Sin[ n θ ] Sin[ϕ] Sin[γ]
  ]  // QSimplify

] /; ScalarQ[n] ⋀ n > 0
```

Power e^q

```
Q /: Power[ E, q: Q[ __ ?ScalarQ ] ] :=
    Exp[ q ]
```

Sqrt

```
Q /: Sqrt[ q: Q[ __ ?ScalarQ ] ] :=
    Power[ q, 1/2 ]
```

Right Divide

```
Q /: Divide[ left: Q[ __ ?ScalarQ ], right : Q[ __ ?ScalarQ ] ] :=
    left ** ( Conjugate[right] ** 1/SqNorm[right] )
```

Simplification

```
Q /: QSimplify[ q: Q[ __ ?ScalarQ ] ] :=
  Simplify[ TrigExpand /@ q ]
```

```
QSimplify[ q_ ] := q
```

End Package

```
$Pre = TimeIt;
```

```
End[];
```

```
EndPackage[];
```