

Juliaでクォータニオンと双対クォータニオン

```
In [1]: ] activate .
```

```
In [2]: ] instantiate
```

クオータニオン

English version: <https://arxiv.org/pdf/1711.02508>

クオータニオンの定義

$$q = w + xi + yj + zk$$

- w はスカラー部（実部）
- x, y, z はベクトル部（虚部）
- i, j, k は虚数単位で、以下の関係を満たす.

$$i^2 = j^2 = k^2 = ijk = -1$$

$$ij = k, \quad jk = i, \quad ki = j$$

$$ji = -k, \quad kj = -i, \quad ik = -j$$

```
In [5]: # コンストラクタ
Quaternion(v::SVector{4}) = Quaternion(v[1], v[2], v[3], v[4])
```

```
In [6]: import Base: show
function show(io::IO, q::Quaternion)
    print(io, "($(q.w) + $(q.x)i + $(q.y)j + $(q.z)k)")
end
```

```
In [7]: # スカラー部
function get_scalar(q::Quaternion)
    return q.w
end

# ベクトル部
function get_vector(q::Quaternion)
    return SVector(q.x, q.y, q.z)
end
```

```

# 実数か
function is_real(q::Quaternion)
    if get_vector(q) == SVector{0.0, 0.0, 0.0}
        return true
    else
        return false
    end
end

# 純クォータニオンか
function is_pure(q::Quaternion)
    if get_scalar(q) == 0.0
        return true
    else
        return false
    end
end
end

```

Out[7]: is_pure (generic function with 1 method)

例

```

In [8]: q1 = Quaternion(1.0, 2.0, 3.0, 4.0)
        q2 = Quaternion(0.0, 1.0, 1.0, 1.0)
        q3 = Quaternion(1.0, 0.0, 0.0, 0.0)
        q4 = Quaternion(2.0, 3.0, 4.0, 5.0)

```

Out[8]: 4-element Quaternion with indices SOneTo(4):
 2.0
 3.0
 4.0
 5.0

```

In [9]: get_scalar(q1), get_scalar(q2), get_scalar(q3), get_scalar(q4)

```

Out[9]: (1.0, 0.0, 1.0, 2.0)

```

In [10]: get_vector(q1), get_vector(q2), get_vector(q3), get_vector(q4)

```

Out[10]: ([2.0, 3.0, 4.0], [1.0, 1.0, 1.0], [0.0, 0.0, 0.0], [3.0, 4.0, 5.0])

```

In [11]: is_pure(q1), is_pure(q2), is_pure(q3), is_pure(q4)

```

Out[11]: (false, true, false, false)

```

In [12]: is_real(q1), is_real(q2), is_real(q3), is_real(q4)

```

Out[12]: (false, false, true, false)

補助関数

共役

$$q^* = w - xi - yj - zk$$

```

In [13]: function conjugate(q::Quaternion)
        return Quaternion(q.w, -q.x, -q.y, -q.z)
    end

```

Out[13]: conjugate (generic function with 1 method)

ノルム

$$||q|| = \sqrt{w^2 + x^2 + y^2 + z^2}$$

```

In [14]: function norm(q::Quaternion)
        return (q.w^2 + q.x^2 + q.y^2 + q.z^2)^0.5
    end

```

Out[14]: norm (generic function with 1 method)

逆元

$$q^{-1} = \frac{q^*}{||q||^2}$$

```
In [15]: function inverse(q::Quaternion)
          return conjugate(q)/(norm(q)^2)
        end
```

Out[15]: inverse (generic function with 1 method)

正規化

```
In [16]: function normalize(q::Quaternion)
          return q/norm(q)
        end
```

Out[16]: normalize (generic function with 1 method)

例

```
In [17]: norm(q1), norm(q2), norm(q3), norm(q4)
```

Out[17]: (5.477225575051661, 1.7320508075688772, 1.0, 7.3484692283495345)

```
In [18]: conjugate(q1), conjugate(q2), conjugate(q3), conjugate(q4)
```

Out[18]: ((1.0 + -2.0i + -3.0j + -4.0k), (0.0 + -1.0i + -1.0j + -1.0k), (1.0 + -0.0i + -0.0j + -0.0k), (2.0 + -3.0i + -4.0j + -5.0k))

```
In [19]: inverse(q1), inverse(q2), inverse(q3), inverse(q4)
```

Out[19]: ((0.03333333333333333 + -0.06666666666666667i + -0.1j + -0.1333333333333333k), (0.0 + -0.33333333333333337i + -0.33333333333333337j + -0.33333333333333337k), (1.0 + -0.0i + -0.0j + -0.0k), (0.037037037037037035 + -0.05555555555555555i + -0.07407407407407407j + -0.09259259259259259k))

```
In [20]: normalize(q1), normalize(q2), normalize(q3), normalize(q4)
```

Out[20]: ((0.18257418583505536 + 0.3651483716701107i + 0.5477225575051661j + 0.7302967433402214k), (0.0 + 0.5773502691896258i + 0.5773502691896258j + 0.5773502691896258k), (1.0 + 0.0i + 0.0j + 0.0k), (0.2721655269759087 + 0.408248290463863i + 0.5443310539518174j + 0.6804138174397717k))

演算

加法

$$q_1 + q_2 = (w_1 + w_2) + (x_1 + x_2)i + (y_1 + y_2)j + (z_1 + z_2)k$$

例

```
In [21]: q1 + q2
```

Out[21]: 4-element Quaternion with indices S0neTo(4):
1.0
3.0
4.0
5.0

乘法 (eq. 17-18)

$$q_1 q_2 = (w_1 + x_1 i + y_1 j + z_1 k)(w_2 + x_2 i + y_2 j + z_2 k) \quad (1)$$

$$= w_1 w_2 - x_1 x_2 - y_1 y_2 - z_1 z_2 \quad (2)$$

$$+ (w_1 x_2 + x_1 w_2 + y_1 z_2 - z_1 y_2) i \quad (3)$$

$$+ (w_1 y_2 - x_1 z_2 + y_1 w_2 + z_1 x_2) j \quad (4)$$

$$+ (w_1 z_2 + x_1 y_2 - y_1 x_2 + z_1 w_2) k \quad (5)$$

```
In [22]: function quat2mat(q::Quaternion)
          mat = [q.w -q.x -q.y -q.z;
                 q.x q.w -q.z q.y;
                 q.y q.z q.w -q.x;
                 q.z -q.y q.x q.w]
          return SMatrix{4,4}(mat)
        end
```

Out[22]: quat2mat (generic function with 1 method)

```
In [23]: import Base:*
```

```
In [24]: function *(q1::Quaternion, q2::Quaternion)
          return Quaternion(quat2mat(q1)*q2)
        end
```

Out[24]: * (generic function with 320 methods)

例

```
In [25]: quat2mat(q1)
```

Out[25]: 4x4 SMatrix{4, 4, Float64, 16} with indices S0neTo(4)×S0neTo(4):

1.0	-2.0	-3.0	-4.0
2.0	1.0	-4.0	3.0
3.0	4.0	1.0	-2.0
4.0	-3.0	2.0	1.0

```
In [26]: q1*q2
```

Out[26]: 4-element Quaternion with indices S0neTo(4):

-9.0
0.0
3.0
0.0

回転行列

回転行列の定義

```
In [27]: struct Rotation
          matrix::SMatrix{3, 3, Float64}

          Rotation() = new(@SMatrix [1.0 0.0 0.0; 0.0 1.0 0.0; 0.0 0.0 1.0])

          function Rotation(mat::AbstractMatrix)
              @assert size(mat) == (3, 3) "Rotation matrix must be 3x3"
              new(SMatrix{3,3}(mat))
          end

          function Rotation(m11, m12, m13, m21, m22, m23, m31, m32, m33)
              new(@SMatrix [m11 m12 m13; m21 m22 m23; m31 m32 m33])
          end
        end
```

積の定義

```
In [28]: import Base: *
```

```
In [29]: function *(r1::Rotation, r2::Rotation)
          Rotation(r1.matrix * r2.matrix)
        end

# ベクトルに回転を適用する場合 (オプション)
function *(r::Rotation, v::AbstractVector)
    @assert length(v) == 3 "Vector must have length 3"
    r.matrix * v
end
```

Out[29]: * (generic function with 322 methods)

補助関数

描画関数

```
visualise_rotation(rotation::Rotation; scale=1.0, origin=[0,0,0], colors=:red, :green, :blue)
```

回転行列を3次元空間内の3本の矢印として可視化する。各矢印は回転後の座標系の基底ベクトルを表す。

引数:

- `rotation::Rotation`: 可視化する回転行列
- `scale=1.0`: 矢印のスケール
- `origin=[0,0,0]`: 矢印の始点
- `colors=:red, :green, :blue`: x, y, z軸の色

- `show_original=true` : 元の座標系も表示するかどうか

戻り値:

- Plotsオブジェクト

使い方: `x = visualise_rotation` のようにして, `display(x)` とすることで描画できる.

```
In [30]: function visualise(rotation::Rotation;
                    scale=1.0,
                    origin=[0,0,0],
                    colors=[:red, :green, :blue],
                    show_original=true)

    # 元の基底
    ex = [1.0, 0.0, 0.0] * scale
    ey = [0.0, 1.0, 0.0] * scale
    ez = [0.0, 0.0, 1.0] * scale

    # 回転後の基底
    rex = rotation.matrix * ex
    rey = rotation.matrix * ey
    rez = rotation.matrix * ez

    # 原点
    o = collect(origin)

    # 可視化
    p = plot(
        legend=true,
        xlabel="X",
        ylabel="Y",
        zlabel="Z",
        aspect_ratio=:equal
    )

    # 元の座標系
    if show_original
        # 点線で元の座標系を表示
        quiver!(p,
            [o[1]], [o[2]], [o[3]],
            quiver=([ex[1]], [ex[2]], [ex[3]]),
            color=colors[1], alpha=0.3, label="Original X", line=:dash
        )
        quiver!(p,
            [o[1]], [o[2]], [o[3]],
            quiver=([ey[1]], [ey[2]], [ey[3]]),
            color=colors[2], alpha=0.3, label="Original Y", line=:dash
        )
        quiver!(p,
            [o[1]], [o[2]], [o[3]],
            quiver=([ez[1]], [ez[2]], [ez[3]]),
            color=colors[3], alpha=0.3, label="Original Z", line=:dash
        )
    end

    # 回転後の座標系
    quiver!(p,
        [o[1]], [o[2]], [o[3]],
        quiver=([rex[1]], [rex[2]], [rex[3]]),
        color=colors[1], linewidth=2, label="Rotated X"
    )
    quiver!(p,
        [o[1]], [o[2]], [o[3]],
        quiver=([rey[1]], [rey[2]], [rey[3]]),
        color=colors[2], linewidth=2, label="Rotated Y"
    )
    quiver!(p,
        [o[1]], [o[2]], [o[3]],
        quiver=([rez[1]], [rez[2]], [rez[3]]),
        color=colors[3], linewidth=2, label="Rotated Z"
    )

    # グラフの表示範囲を調整
    max_range = scale * 1.2 # 少し余裕を持たせる
    plot!(p, xlim=[-max_range, max_range], ylim=[-max_range, max_range], zlim=[-max_range, max_range])

    return p
end
```

Out[30]: visualise (generic function with 1 method)

度→ラジアン

```
In [31]: function deg2rad(angle_deg::Real)
          return angle_deg*pi/180
        end
```

Out[31]: deg2rad (generic function with 1 method)

ラジアン→度

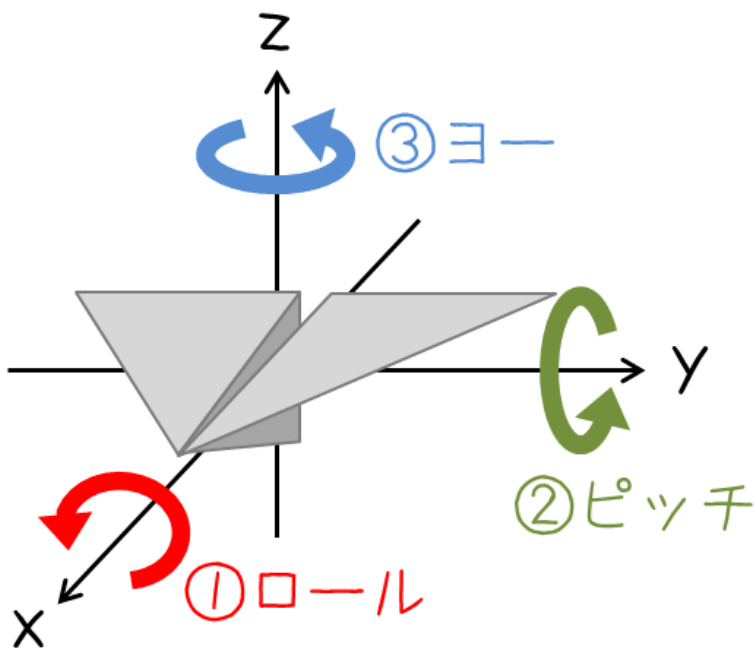
```
In [32]: function rad2deg(angle_rad::Real)
          return angle_rad*180/pi
        end
```

Out[32]: rad2deg (generic function with 1 method)

```
In [33]: # 回転行列の有効性チェック (直交行列であることを確認)
          function is_valid_rotation(r::Rotation)
              mat = r.matrix
              identity_approx = mat * transpose(mat)
              return isapprox(identity_approx, I, atol=1e-10)
          end
```

Out[33]: is_valid_rotation (generic function with 1 method)

回転行列の生成 (RPYから)



出典: https://watako-lab.com/wp-content/uploads/2019/01/roll_pitch_yaw.png

```
In [34]: # roll
          function rotation_x(angle_rad::Real)
              c = cos(angle_rad)
              s = sin(angle_rad)
              return Rotation{@SMatrix {
                  1.0 0.0 0.0;
                  0.0 c -s;
                  0.0 s c
              }}
          end
```

Out[34]: rotation_x (generic function with 1 method)

```
In [35]: # pitch
          function rotation_y(angle_rad::Real)
              c = cos(angle_rad)
              s = sin(angle_rad)
```

```

    return Rotation(@SMatrix [
        c 0.0 s;
        0.0 1.0 0.0;
        -s 0.0 c
    ])
end

```

Out[35]: rotation_y (generic function with 1 method)

```

In [36]: # yaw
function rotation_z(angle_rad::Real)
    c = cos(angle_rad)
    s = sin(angle_rad)
    return Rotation(@SMatrix [
        c -s 0.0;
        s c 0.0;
        0.0 0.0 1.0
    ])
end

```

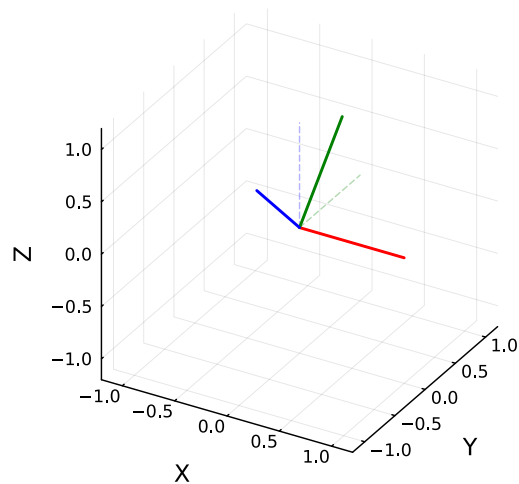
Out[36]: rotation_z (generic function with 1 method)

例

```

In [37]: px = visualise(rotation_x(deg2rad(45))) # roll
display(px)

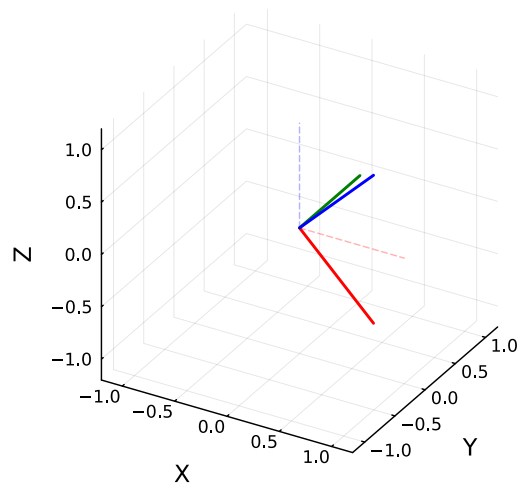
```



```

In [38]: py = visualise(rotation_y(deg2rad(45))) # pitch
display(py)

```

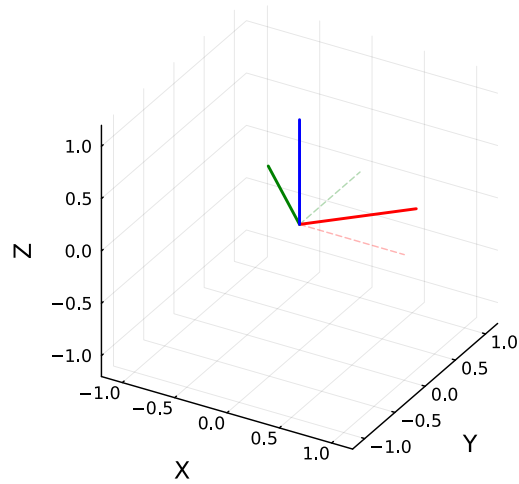


```

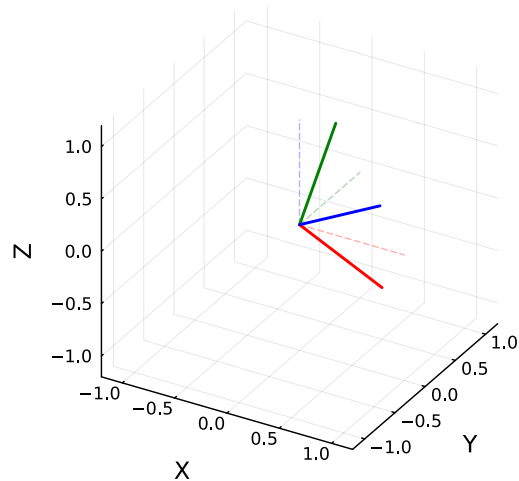
In [39]: pz = visualise(rotation_z(deg2rad(45))) # yaw

```

```
display(pz)
```



```
In [40]: p = visualise(rotation_z(deg2rad(45))*rotation_y(deg2rad(45))*rotation_x(deg2rad(45)))
display(p)
```



回転行列とRPYの変換

```
In [41]: function rotation2rpy(rotation::Rotation)
    r = rotation.matrix # 行列の要素を抽出(配列のindex処理に必要)
    # ジンバルロックの確認
    # cos(pitch) ≈ 0 のとき, すなわち pitch ≈ ±π/2 のとき
    if abs(r[3, 1]) > 1.0 - 1e-6
        # ジンバルロックの処理
        # r[3, 1] = -sin(pitch) が -1 に近い場合, pitch ≈ π/2
        if r[3, 1] < 0
            roll = 0.0
            pitch = π/2
            yaw = atan(r[1, 2], r[2, 2])
        else
            # r[3, 1] = -sin(pitch) が 1 に近い場合, pitch ≈ -π/2
            roll = 0.0
            pitch = -π/2
            yaw = -atan(r[1, 2], r[2, 2])
        end
    else
        # 通常のケース
        roll = atan(r[3, 2] / r[3, 3]) # z
        pitch = -asin(r[3, 1]) # y
        yaw = atan(r[2, 1] / r[1, 1]) # x
    end
end
```



```
    return (roll, pitch, yaw)
end
```

Out[41]: rotation2rpy (generic function with 1 method)

```
In [42]: function rpy2rotation(roll::Real, pitch::Real, yaw::Real)
        R = rotation_z(yaw) * rotation_y(pitch) * rotation_x(roll)
        return R
    end
```

Out[42]: rpy2rotation (generic function with 1 method)

例

```
In [43]: roll = 0.3
        pitch = 0.2
        yaw = 0.1
```

Out[43]: 0.1

```
In [44]: rotation2rpy(rpy2rotation(0.3,0.2,0.1))
```

Out[44]: (0.29999999999999993, 0.2, 0.09999999999999999)

ロドリゲスの公式

```
In [45]: # eq. 65
        function skew_sym(v::AbstractVector{<:Real})

            x, y, z = v

            mat = [0 -z y;
                  z 0 -x;
                  -y x 0]

            return Rotation(mat)
        end
```

Out[45]: skew_sym (generic function with 1 method)

```
In [46]: # eq. 77
        function Rodrigues(v::AbstractVector{<:Real}, angle_rad::Real)
            # 単位ベクトル化
            u = v/LinearAlgebra.norm(v)
            θ = angle_rad

            # skew_sym関数は反対称行列を返すが、この実装ではRotation型を返している
            # そのため、内部の行列要素にアクセスする必要がある
            S = skew_sym(u).matrix # .matrixでRotation型から行列を取り出す

            # Rodriguesの公式:  $R = I + \sin(\theta)S + (1-\cos(\theta))S^2$ 
            mat = I(3) + sin(θ)*S + (1-cos(θ))*(S*S)

            return Rotation(mat)
        end
```

Out[46]: Rodrigues (generic function with 1 method)

例

```
In [47]: Rodrigues([1.0, 0.0, 0.0], pi)
```

Out[47]: Rotation([1.0 0.0 0.0; 0.0 -1.0 0.0; 0.0 0.0 -1.0])

```
In [48]: rotation_x(pi)
```

Out[48]: Rotation([1.0 0.0 0.0; 0.0 -1.0 -0.0; 0.0 0.0 -1.0])

```
In [49]: # 上記の2つが同一のものになっているか確認
        isapprox(Rodrigues([1.0, 0.0, 0.0], pi).matrix, rotation_x(pi).matrix, atol=1e-10)
```

Out[49]: true

クォータニオンと回転行列の変換 (式115)

単位クォータニオン $q = w + xi + yj + zk$ から回転行列 R への変換は以下の式で表される。

$$R = \begin{pmatrix} w^2 + x^2 - y^2 - z^2 & 2(xy - wz) & 2(xz + wy) \\ 2(xy + wz) & w^2 - x^2 + y^2 - z^2 & 2(yz - wx) \\ 2(xz - wy) & 2(yz + wx) & w^2 - x^2 - y^2 + z^2 \end{pmatrix}$$

```
In [50]: function quat2rotation(q::Quaternion)
          q = normalize(q)
          mat = [q.w^2+q.x^2-q.y^2-q.z^2 2(q.x*q.y-q.w*q.z) 2(q.x*q.z+q.w*q.y);
                  2(q.x*q.y+q.w*q.z) q.w^2-q.x^2+q.y^2-q.z^2 2(q.y*q.z-q.w*q.x);
                  2(q.x*q.z-q.w*q.y) 2(q.y*q.z+q.w*q.x) q.w^2-q.x^2-q.y^2+q.z^2]
          return Rotation(mat)
        end
```

Out[50]: quat2rotation (generic function with 1 method)

例

```
In [51]: r1 = quat2rotation(q1)
```

Out[51]: Rotation([-0.6666666666666665 0.1333333333333336 0.7333333333333332; 0.6666666666666665 -0.3333333333333326 0.6666666666666666; 0.3333333333333326 0.9333333333333332 0.1333333333333333])

```
In [52]: is_valid_rotation(r1)
```

Out[52]: true

```
In [53]: r3 = quat2rotation(q3)
```

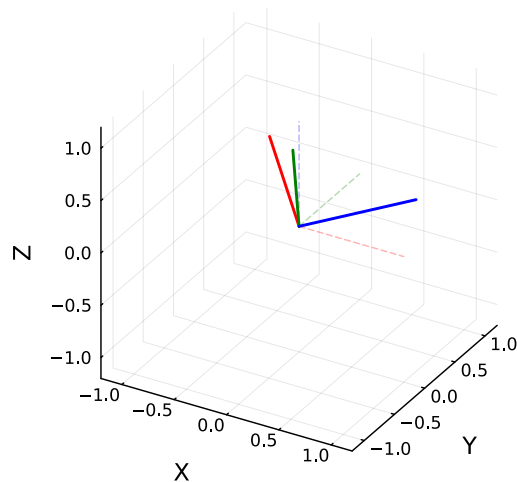
Out[53]: Rotation([1.0 0.0 0.0; 0.0 1.0 0.0; 0.0 0.0 1.0])

```
In [54]: is_valid_rotation(r3)
```

Out[54]: true

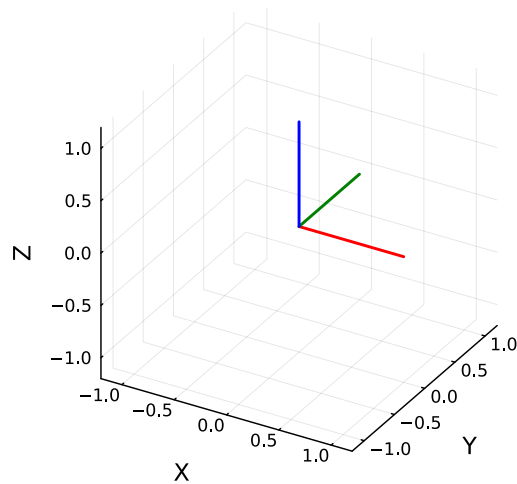
```
In [55]: visualise(r1)
```

Out[55]:



```
In [56]: visualise(r3)
```

Out[56]:

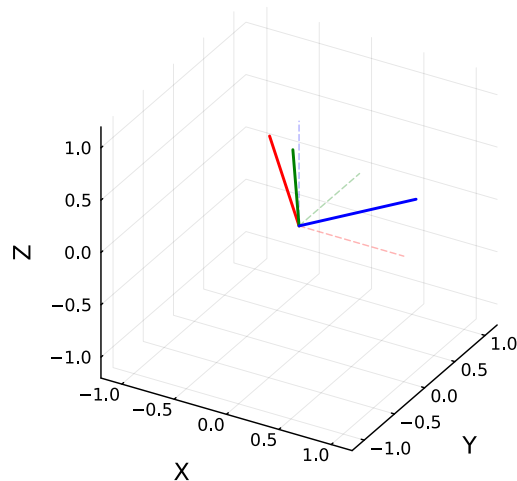


```
In [57]: function visualise(q::Quaternion; scale=1.0, origin=[0,0,0], colors=[red, :green, :blue], show_original=true)
          q_norm = normalize(q)
          rot = quat2rotation(q_norm)
          return visualise(rot; scale=scale, origin=origin, colors=colors, show_original=show_original)
        end
```

Out[57]: visualise (generic function with 2 methods)

```
In [58]: visualise(q1)
```

Out[58]:



```
In [59]: function rotation2quat(r::Rotation)
          R = r.matrix
          # 対角和を計算
          trace = R[1,1] + R[2,2] + R[3,3]

          if trace > 0
              # 対角和が正の場合
              s = 0.5 / sqrt(trace + 1.0)
              w = 0.25 / s
              x = (R[3,2] - R[2,3]) * s
              y = (R[1,3] - R[3,1]) * s
              z = (R[2,1] - R[1,2]) * s
          elseif (R[1,1] > R[2,2]) && (R[1,1] > R[3,3])
              # R[1,1]が最大の場合
              s = 2.0 * sqrt(1.0 + R[1,1] - R[2,2] - R[3,3])
              w = (R[3,2] - R[2,3]) / s
              x = 0.25 * s
              y = (R[1,2] + R[2,1]) / s
              z = (R[1,3] + R[3,1]) / s
          elseif R[2,2] > R[3,3]
              # R[2,2]が最大の場合
              s = 2.0 * sqrt(1.0 + R[2,2] - R[1,1] - R[3,3])
```

```

w = (R[1,3] - R[3,1]) / s
x = (R[1,2] + R[2,1]) / s
y = 0.25 * s
z = (R[2,3] + R[3,2]) / s
else
# R[3,3]が最大の場合
s = 2.0 * sqrt(1.0 + R[3,3] - R[1,1] - R[2,2])
w = (R[2,1] - R[1,2]) / s
x = (R[1,3] + R[3,1]) / s
y = (R[2,3] + R[3,2]) / s
z = 0.25 * s
end

return Quaternion(w, x, y, z)
end

```

Out[59]: rotation2quat (generic function with 1 method)

双対クォータニオン (Dual Quaternion)

Reference: Dual Quaternions, Yan-Bin Jia,

<https://faculty.sites.iastate.edu/jia/files/inline-files/dual-quaternion.pdf>

双対クォータニオンの定義

双対クォータニオンは、3次元空間における回転と並進を同時に表現できる数学的構造である。これはクォータニオン（四元数）とデュアル数の概念を組み合わせたものである。デュアルクォータニオンは特に以下の分野で活用される。

- ロボティクス（運動学と動力学）
- コンピュータグラフィックス（3D変換とアニメーション）
- コンピュータビジョン（剛体変換の表現）
- 宇宙工学（姿勢制御）

対応関係

- クォータニオン \leftrightarrow 回転行列
- 双対クォータニオン \leftrightarrow 同次変換行列

の対応になっている。

デュアル数とは

デュアル数は以下の形式で表される数である。

$$a + \varepsilon b$$

ここで、 a, b は実数、 ε はデュアル単位で、 $\varepsilon^2 = 0$ かつ $\varepsilon \neq 0$ の性質を持つ。

双対クォータニオンの定義

デュアルクォータニオンは以下の形式で表される。

$$\hat{q} = q_r + \varepsilon q_d$$

ここで、

- q_r は実部クォータニオン（rotation part）
- q_d は双対部クォータニオン（dual part）
- ε はデュアル単位（ $\varepsilon^2 = 0, \varepsilon \neq 0$ ）

```

In [60]: struct DualQuaternion
          real::Quaternion # 実部
          dual::Quaternion # 双対部
        end

```

例

```

In [61]: q1 = Quaternion(1.0, 0.0, 0.0, 0.0) # 1
          q2 = Quaternion(0.0, 1.0, 0.0, 0.0) # i

```

```
Out[61]: 4-element Quaternion with indices SOneTo(4):
 0.0
 1.0
 0.0
 0.0
```

```
In [62]: dq1 = DualQuaternion(q1, q2)
dq2 = DualQuaternion(q2, q1)
```

```
Out[62]: DualQuaternion((0.0 + 1.0i + 0.0j + 0.0k), (1.0 + 0.0i + 0.0j + 0.0k))
```

演算

加算と減算

デュアルクォータニオンの加減算は、実部と双対部それぞれで独立に行う。

$$\hat{q}_1 \pm \hat{q}_2 = (q_{r1} \pm q_{r2}) + \varepsilon(q_{d1} \pm q_{d2})$$

```
In [63]: import Base: +
function +(dq1::DualQuaternion, dq2::DualQuaternion)
    real_part = dq1.real + dq2.real
    dual_part = dq1.dual + dq2.dual
    return DualQuaternion(real_part, dual_part)
end
```

```
Out[63]: + (generic function with 245 methods)
```

```
In [64]: import Base: -
function -(dq1::DualQuaternion, dq2::DualQuaternion)
    real_part = dq1.real - dq2.real
    dual_part = dq1.dual - dq2.dual
    return DualQuaternion(real_part, dual_part)
end
```

```
Out[64]: - (generic function with 257 methods)
```

例

```
In [65]: dq3 = dq1 + dq2
```

```
Out[65]: DualQuaternion((1.0 + 1.0i + 0.0j + 0.0k), (1.0 + 1.0i + 0.0j + 0.0k))
```

スカラー倍

スカラー s との乗算は、実部と双対部の両方に適用される。

$$s\hat{q} = sq_r + \varepsilon sq_d$$

```
In [66]: import Base: *
function *(s::Real, dq::DualQuaternion)
    return DualQuaternion(s * dq.real, s * dq.dual)
end
```

```
Out[66]: * (generic function with 323 methods)
```

```
In [67]: function *(dq::DualQuaternion, s::Real)
    return s * dq
end
```

```
Out[67]: * (generic function with 324 methods)
```

例

```
In [68]: dq4 = 2.0 * dq1
```

```
Out[68]: DualQuaternion((2.0 + 0.0i + 0.0j + 0.0k), (0.0 + 2.0i + 0.0j + 0.0k))
```

乗算

デュアルクォータニオンの乗算は、デュアル数の性質 ($\varepsilon^2 = 0$) を考慮して行う。

$$\hat{q}_1 \hat{q}_2 = q_{r1} q_{r2} + \varepsilon(q_{r1} q_{d2} + q_{d1} q_{r2})$$

```
In [69]: function *(dq1::DualQuaternion, dq2::DualQuaternion)
    real_part = dq1.real * dq2.real
    dual_part = dq1.real * dq2.dual + dq1.dual * dq2.real
    return DualQuaternion(real_part, dual_part)
end
```

Out[69]: * (generic function with 325 methods)

例

```
In [70]: dq5 = dq1 * dq2
```

Out[70]: DualQuaternion((0.0 + 1.0i + 0.0j + 0.0k), (0.0 + 0.0i + 0.0j + 0.0k))

共役

デュアルクォータニオンには3種類の共役が定義される。

1. **クォータニオン共役**：実部と双対部の両方にクォータニオン共役を適用

$$\hat{q}^* = q_r^* + \varepsilon q_d^*$$

2. **デュアル共役**： ε の符号を反転

$$\hat{q}^\varepsilon = q_r - \varepsilon q_d$$

3. **完全共役**：両方の共役を適用

$$\hat{q}^{*\varepsilon} = q_r^* - \varepsilon q_d^*$$

```
In [71]: function conjugate(dq::DualQuaternion)
    return DualQuaternion(conjugate(dq.real), conjugate(dq.dual))
end
```

Out[71]: conjugate (generic function with 2 methods)

```
In [72]: function dual_conjugate(dq::DualQuaternion)
    return DualQuaternion(dq.real, -dq.dual)
end
```

Out[72]: dual_conjugate (generic function with 1 method)

```
In [73]: function full_conjugate(dq::DualQuaternion)
    conj_quat(q::Quaternion) = Quaternion(q.w, -q.x, -q.y, -q.z)
    return DualQuaternion(conj_quat(dq.real), -conj_quat(dq.dual))
end
```

Out[73]: full_conjugate (generic function with 1 method)

ノルムと正規化

デュアルクォータニオンのノルムは以下のように定義される。

$$||\hat{q}|| = ||q_r|| + \varepsilon \frac{q_r \cdot q_d}{||q_r||}$$

単位デュアルクォータニオン ($||\hat{q}|| = 1$) は、以下の条件を満たす。

- $||q_r|| = 1$ (実部が単位クォータニオン)
- $q_r \cdot q_d = 0$ (実部と双対部が直交)

```
In [74]: function norm(dq::DualQuaternion)
    return sqrt(dq.real.w^2 + dq.real.x^2 + dq.real.y^2 + dq.real.z^2)
end
```

Out[74]: norm (generic function with 2 methods)

```
In [75]: function normalize(dq::DualQuaternion)
    n = norm(dq)
    if n ≈ 0
        throw(DomainError(n, "Cannot normalize a dual quaternion with zero norm"))
    end

    n_inv = 1.0 / n
    real_part = n_inv * dq.real
    dual_part = n_inv * dq.dual
end
```

```

# 実部と双対部の内積を計算
dot_prod = real_part.w * dual_part.w + real_part.x * dual_part.x +
            real_part.y * dual_part.y + real_part.z * dual_part.z

# dual_partを調整して実部と双対部が直交するようにする
dual_part = dual_part - dot_prod * real_part

return DualQuaternion(real_part, dual_part)
end

```

Out[75]: normalize (generic function with 2 methods)

```

In [76]: function one(::Type{DualQuaternion})
            return DualQuaternion(Quaternion(1.0, 0.0, 0.0, 0.0), Quaternion(0.0, 0.0, 0.0, 0.0))
        end

```

Out[76]: one (generic function with 1 method)

剛体の運動（回転と並進）

回転の表現

回転を表すデュアルクォータニオンは以下ようになる．

$$\hat{q}_{rot} = q_r + \varepsilon 0$$

ただし， q_r : 回転を表す単位クォータニオン

```

In [77]: function rotation_dq(q::Quaternion)
            q_norm=normalize(q)
            return DualQuaternion(q_norm, Quaternion(0.0, 0.0, 0.0, 0.0))
        end

```

Out[77]: rotation_dq (generic function with 1 method)

並進の表現

並進ベクトル $\mathbf{t} = [t_x, t_y, t_z]$ を表すデュアルクォータニオンは以下ようになる．

$$\hat{q}_{trans} = 1 + \varepsilon \frac{1}{2} (t_x i + t_y j + t_z k)$$

```

In [78]: function translation_dq(v::Vector{Float64})
            if length(v) != 3
                throw(ArgumentError("Translation vector must have 3 elements"))
            end

            real_part = Quaternion(1.0, 0.0, 0.0, 0.0)

            dual_part = Quaternion(0.0, 0.5*v[1], 0.5*v[2], 0.5*v[3])

            return DualQuaternion(real_part, dual_part)
        end

```

Out[78]: translation_dq (generic function with 1 method)

回転と並進の組み合わせ

回転 q_r と並進 \mathbf{t} を組み合わせたデュアルクォータニオンは，以下ようになる．

$$\hat{q} = q_r + \varepsilon \frac{1}{2} (t_x i + t_y j + t_z k) q_r$$

ただし，まず回転してから，並進運動する点に注意．（同次変換行列と同じ順序なのでわかりやすい．）

```

In [79]: function transform_dq(q::Quaternion, v::Vector{Float64})
            if length(v) != 3
                throw(ArgumentError("Translation vector must have 3 elements"))
            end

            # 回転
            rot_dq = rotation_dq(q)

            # 並進
            t = Quaternion(0.0, 0.5*v[1], 0.5*v[2], 0.5*v[3])
            trans_dual = t * rot_dq.real

```

```
    return DualQuaternion(rot_dq.real, trans_dual)
end
```

Out[79]: transform_dq (generic function with 1 method)

デュアルクォータニオンから回転と並進を抽出

```
In [80]: function extract_transform(dq::DualQuaternion)
    # 回転
    rotation = dq.real

    # 並進
    t = 2.0 * (dq.dual * Quaternion(dq.real.w, -dq.real.x, -dq.real.y, -dq.real.z))
    translation = [t.x, t.y, t.z]

    return rotation, translation
end
```

Out[80]: extract_transform (generic function with 1 method)

例

```
In [81]: # Y軸周りに45度回転
θ = π/4
axis = [0.0, 0.0, 1.0]
rot_q = Quaternion(cos(θ/2), sin(θ/2)*axis[1], sin(θ/2)*axis[2], sin(θ/2)*axis[3])
```

Out[81]: 4-element Quaternion with indices SOneTo(4):
0.9238795325112867
0.0
0.0
0.3826834323650898

```
In [82]: rot_dq = rotation_dq(rot_q) # 回転
```

Out[82]: DualQuaternion((0.9238795325112867 + 0.0i + 0.0j + 0.3826834323650898k), (0.0 + 0.0i + 0.0j + 0.0k))

```
In [83]: # X方向に2単位, Z方向に1単位の並進
trans = [2.0, 0.0, 1.0]
```

Out[83]: 3-element Vector{Float64}:
2.0
0.0
1.0

```
In [84]: trans_dq = translation_dq(trans) # 並進
```

Out[84]: DualQuaternion((1.0 + 0.0i + 0.0j + 0.0k), (0.0 + 1.0i + 0.0j + 0.5k))

```
In [85]: dq = transform_dq(rot_q, trans)
```

Out[85]: DualQuaternion((0.9238795325112867 + 0.0i + 0.0j + 0.3826834323650898k), (-0.1913417161825449 + 0.9238795325112867i + -0.3826834323650898j + 0.46193976625564337k))

描画関数

```
visualise(dq::DualQuaternion; scale=1.0, origin=[0,0,0], colors=:red, :green, :blue, show_original=true, show_translation=true)
```

デュアルクォータニオンを3D空間で可視化する。回転と並進の両方を表示する。

引数:

- `dq::DualQuaternion`: 可視化するデュアルクォータニオン
- `scale=1.0`: 座標軸のスケール
- `origin=[0,0,0]`: 初期座標系の原点
- `colors=:red, :green, :blue`: X, Y, Z軸の色
- `show_original=true`: 元の座標系を表示するかどうか
- `show_translation=true`: 並進ベクトルを表示するかどうか

戻り値:

- Plotsオブジェクト

```
In [86]: function visualise(dq::DualQuaternion; scale=1.0, origin=[0,0,0], colors=:red, :green, :blue, show_ori
    # デュアルクォータニオンから回転と並進を抽出
    rotation, translation = extract_transform(dq)
```



```

# 回転行列に変換
rot_matrix = quat2rotation(rotation)

# 元の基底
ex = [1.0, 0.0, 0.0] * scale
ey = [0.0, 1.0, 0.0] * scale
ez = [0.0, 0.0, 1.0] * scale

# 回転後の基底
rex = rot_matrix.matrix * ex
rey = rot_matrix.matrix * ey
rez = rot_matrix.matrix * ez

# 原点と変換後の原点
o = collect(origin)
new_o = o + translation

# 可視化
p = plot(
    legend=true,
    xlabel="X",
    ylabel="Y",
    zlabel="Z",
    aspect_ratio=:equal
)

# 元の座標系 (オプション)
if show_original
    # 点線で元の座標系を表示
    quiver!(p,
        [o[1]], [o[2]], [o[3]],
        quiver=([ex[1]], [ex[2]], [ex[3]]),
        color=colors[1], alpha=0.3, label="Original X", line=:dash
    )
    quiver!(p,
        [o[1]], [o[2]], [o[3]],
        quiver=([ey[1]], [ey[2]], [ey[3]]),
        color=colors[2], alpha=0.3, label="Original Y", line=:dash
    )
    quiver!(p,
        [o[1]], [o[2]], [o[3]],
        quiver=([ez[1]], [ez[2]], [ez[3]]),
        color=colors[3], alpha=0.3, label="Original Z", line=:dash
    )
end

# 変換後の座標系
quiver!(p,
    [new_o[1]], [new_o[2]], [new_o[3]],
    quiver=([rex[1]], [rex[2]], [rex[3]]),
    color=colors[1], linewidth=2, label="Transformed X"
)
quiver!(p,
    [new_o[1]], [new_o[2]], [new_o[3]],
    quiver=([rey[1]], [rey[2]], [rey[3]]),
    color=colors[2], linewidth=2, label="Transformed Y"
)
quiver!(p,
    [new_o[1]], [new_o[2]], [new_o[3]],
    quiver=([rez[1]], [rez[2]], [rez[3]]),
    color=colors[3], linewidth=2, label="Transformed Z"
)

# 並進ベクトルの表示
if show_translation && !isapprox(LinAlg.norm(translation), 0.0)
    quiver!(p,
        [o[1]], [o[2]], [o[3]],
        quiver=([translation[1]], [translation[2]], [translation[3]]),
        color=:purple, linewidth=2, label="Translation"
    )
end

# 原点と変換後の原点をマーカで表示
scatter!(p,
    [o[1], new_o[1]],
    [o[2], new_o[2]],
    [o[3], new_o[3]],
    color=[:black, :purple], markersize=[4, 6],
    label=["Original Origin", "Transformed Origin"]
)
end

```

```

# グラフの表示範囲を調整
max_range = max(scale, LinearAlgebra.norm(translation)) * 1.5 # 少し余裕を持たせる
plot!(p, xlim=[-max_range, max_range], ylim=[-max_range, max_range], zlim=[-max_range, max_range])

return p
end

```

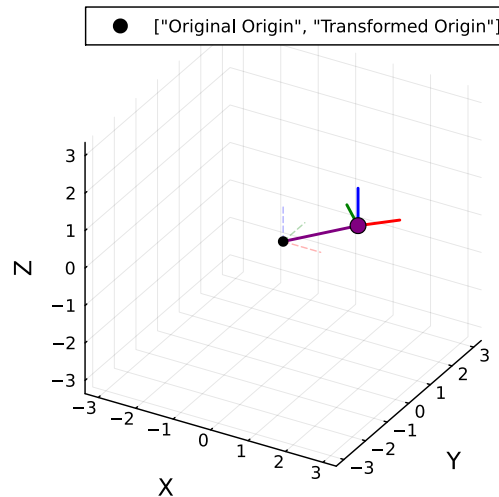
Out[86]: visualise (generic function with 3 methods)

例

```

In [87]: p = visualise(dq, scale=1.0)
display(p)

```



同次変換行列と双対クォータニオンの関係

デュアルクォータニオンから同次変換行列への変換

デュアルクォータニオン $\hat{q} = q_r + \varepsilon q_d$ から同次変換行列 $H \in \mathbb{R}^{4 \times 4}$ への変換は、回転成分と並進成分を抽出して組み合わせることで実現できる。

同次変換行列 H は、回転行列 $R \in \mathbb{R}^{3 \times 3}$ と並進ベクトル $\mathbf{t} \in \mathbb{R}^3$ を用いて以下の形式で表される。

$$H = \begin{bmatrix} R & \mathbf{t} \\ \mathbf{0}^T & 1 \end{bmatrix} = \begin{bmatrix} r_{11} & r_{12} & r_{13} & t_x \\ r_{21} & r_{22} & r_{23} & t_y \\ r_{31} & r_{32} & r_{33} & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

デュアルクォータニオン $\hat{q} = q_r + \varepsilon q_d$ から回転と並進を抽出する過程は以下のとおりである。

1. **回転の抽出:** 実部 q_r は回転を表すクォータニオンであり、これを回転行列 R に変換する。
2. **並進の抽出:** 双対部 q_d と実部 q_r の共役から並進ベクトル \mathbf{t} を計算する。具体的には、

$$\mathbf{t} = 2q_d q_r^*$$

ここで q_r^* は q_r のクォータニオン共役である。

```

In [88]: function dq2H(dq::DualQuaternion)

    rotation, translation = extract_transform(dq)

    R = quat2rotation(rotation).matrix

    result = @SMatrix [
        R[1,1] R[1,2] R[1,3] translation[1];
        R[2,1] R[2,2] R[2,3] translation[2];
        R[3,1] R[3,2] R[3,3] translation[3];
        0.0    0.0    0.0    1.0
    ]

    return result
end

```

Out[88]: dq2H (generic function with 1 method)

同次変換行列からデュアルクォータニオンへの変換

同次変換行列 H からデュアルクォータニオン \hat{q} への変換は、回転行列 R から回転クォータニオン q_r を計算し、並進ベクトル \mathbf{t} と合わせてデュアルクォータニオンを構築する。

同次変換行列 H から回転行列 R と並進ベクトル \mathbf{t} を抽出する。

$$H = \begin{bmatrix} R & \mathbf{t} \\ \mathbf{0}^T & 1 \end{bmatrix}$$

1. 回転クォータニオンの計算: 回転行列 R から回転クォータニオン q_r へ変換する。これは一般的に以下の方法で計算される。

回転行列 R の対角和（トレース）を $S = r_{11} + r_{22} + r_{33}$ とする。

$$q_r = \begin{cases} \left(\frac{1}{2}\sqrt{1+S}, \frac{r_{32}-r_{23}}{4\sqrt{1+S}}, \frac{r_{13}-r_{31}}{4\sqrt{1+S}}, \frac{r_{21}-r_{12}}{4\sqrt{1+S}} \right) & \text{if } S > 0 \\ \left(\frac{r_{32}-r_{23}}{4\sqrt{1+r_{11}-r_{22}-r_{33}}}, \frac{1}{2}\sqrt{1+r_{11}-r_{22}-r_{33}}, \frac{r_{12}+r_{21}}{4\sqrt{1+r_{11}-r_{22}-r_{33}}}, \frac{r_{13}+r_{31}}{4\sqrt{1+r_{11}-r_{22}-r_{33}}} \right) & \text{if } r_{11} > r_{22} \text{ and } r_{11} > r_{33} \\ \left(\frac{r_{13}-r_{31}}{4\sqrt{1+r_{22}-r_{11}-r_{33}}}, \frac{r_{12}+r_{21}}{4\sqrt{1+r_{22}-r_{11}-r_{33}}}, \frac{1}{2}\sqrt{1+r_{22}-r_{11}-r_{33}}, \frac{r_{23}+r_{32}}{4\sqrt{1+r_{22}-r_{11}-r_{33}}} \right) & \text{if } r_{22} > r_{33} \\ \left(\frac{r_{21}-r_{12}}{4\sqrt{1+r_{33}-r_{11}-r_{22}}}, \frac{r_{13}+r_{31}}{4\sqrt{1+r_{33}-r_{11}-r_{22}}}, \frac{r_{23}+r_{32}}{4\sqrt{1+r_{33}-r_{11}-r_{22}}}, \frac{1}{2}\sqrt{1+r_{33}-r_{11}-r_{22}} \right) & \text{otherwise} \end{cases}$$

2. デュアルクォータニオンの構築: 回転クォータニオン q_r と並進ベクトル \mathbf{t} からデュアルクォータニオン \hat{q} を構築する。

$$\hat{q} = q_r + \varepsilon \frac{1}{2} \mathbf{t} q_r$$

ここで、 \mathbf{t} は純クォータニオン $(0, t_x, t_y, t_z)$ として扱われる。

```
In [89]: function H2dq(matrix::AbstractMatrix)
    mat = @SMatrix [
        matrix[1,1] matrix[1,2] matrix[1,3];
        matrix[2,1] matrix[2,2] matrix[2,3];
        matrix[3,1] matrix[3,2] matrix[3,3];
    ]
    R = Rotation(mat)
    t = [matrix[1,4], matrix[2,4], matrix[3,4]]
    rot_q = rotation2quat(R)

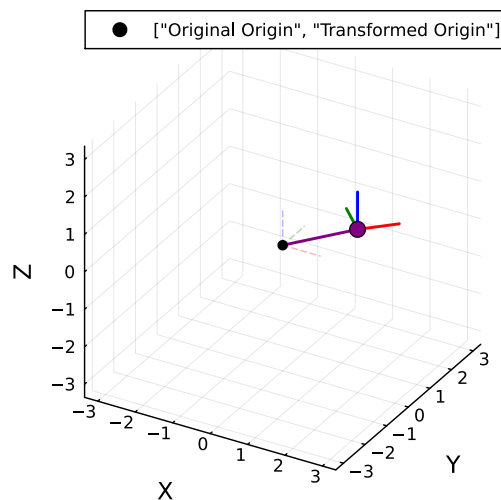
    return transform_dq(rot_q, t)
end
```

Out[89]: H2dq (generic function with 1 method)

例

```
In [90]: visualise(dq)
```

Out[90]:



```
In [91]: # 同次変換行列に変換する
H = dq2H(dq)
```

```
Out[91]: 4x4 SMatrix{4, 4, Float64, 16} with indices S0neTo(4)×S0neTo(4):
 0.707107 -0.707107 0.0 2.0
 0.707107 0.707107 0.0 -4.327e-17
 0.0 0.0 1.0 1.0
 0.0 0.0 0.0 1.0
```

```
In [92]: println("同次変換行列:")
display(H)
```

同次変換行列:

```
4x4 SMatrix{4, 4, Float64, 16} with indices S0neTo(4)×S0neTo(4):
 0.707107 -0.707107 0.0 2.0
 0.707107 0.707107 0.0 -4.327e-17
 0.0 0.0 1.0 1.0
 0.0 0.0 0.0 1.0
```

```
In [93]: # 双対クォータニオンに戻す
dq2 = H2dq(H)
```

```
Out[93]: DualQuaternion((0.9238795325112867 + 0.0i + 0.0j + 0.38268343236508984k), (-0.19134171618254492 + 0.92387
95325112867i + -0.38268343236508984j + 0.46193976625564337k))
```

```
In [94]: visualise(dq2)
```

Out[94]:

