Computational Efficiency

```julia
v = rand(3) # Random vector for demonstration


q = Quaternion(normalize(rand(4))...)
RotM = conversions.quat2rotmatrix(q)

using BenchmarkTools: @benchmark
@benchmark conversions.rotate_vector(v, q)
@benchmark Point{3, Float64}(RotM * v)
```

Julia output Julia

```
BenchmarkTools.Trial: 10000 samples with 957 evaluations per sample.
 Range (min … max):  42.006 ns … 683.281 ns  ┊ GC (min … max): 0.00% … 0.00%
 Time  (median):     53.396 ns               ┊ GC (median):    0.00%
 Time  (mean ± σ):   76.479 ns ±  41.252 ns  ┊ GC (mean ± σ):  0.00% ± 0.00%



  42 ns         Histogram: log(frequency) by time       242 ns <

 Memory estimate: 32 bytes, allocs estimate: 1.

BenchmarkTools.Trial: 10000 samples with 919 evaluations per sample.
 Range (min … max):  115.125 ns … 989.445 ns  ┊ GC (min … max): 0.00% … 0.00%
 Time  (median):     142.220 ns               ┊ GC (median):    0.00%
 Time  (mean ± σ):   172.929 ns ±  79.956 ns  ┊ GC (mean ± σ):  0.00% ± 0.00%



  115 ns        Histogram: log(frequency) by time       564 ns <

 Memory estimate: 112 bytes, allocs estimate: 3.
```

Quaternions provides computational efficiency by 2-3x that of Rotational Matrices.