Moebius

By filip

January 3, 2014

Contents

1 More about complex numbers

theory *MoreComplex*
imports *Complex-Main*
begin

lemma *mult-pow2-lt0*:
  assumes $b \neq 0$
  shows $a < 0 \longleftrightarrow b^2 * a < (0::real)$
using *assms*
by (*metis mult.commute mult-eq-0-iff mult-neg-pos mult-pos-pos not-less-iff-gr-or-eq not-real-square-gt-zero power2-eq-square*)

lemma *mult-pow2-gt0*:
  assumes $b \neq 0$
  shows $a > 0 \longleftrightarrow b^2 * a > (0::real)$
using *assms*

by (*metis mult.commute mult-eq-0-iff mult-neg-pos mult-pos-pos not-less-iff-gr-or-eq not-real-square-gt-zero power2-eq-square*)

lemma *square-cancel*:
  assumes $a^2 \geq b^2$ $a \geq 0$ $b \geq (0::real)$
  shows $a \geq b$
using *real-sqrt-le-iff*[*of* b^2 a^2]
using *assms*
by *auto*

lemmas *complex-cnj* = *complex-cnj-diff complex-cnj-mult complex-cnj-add complex-cnj-divide complex-cnj-minus*

abbreviation *cor* ≡ *complex-of-real*

lemma [*simp*]: *cor* -1 = -1
by (*simp add*: *of-real-neg-numeral*)

lemma [*simp*]: $-$ *cor* -1 = *1*
by *simp*

lemma *rcis-cnj*: *cnj a* = *rcis* (*cmod a*) ($-$ *arg a*)
by (*subst rcis-cmod-arg*[*of a*, *symmetric*]) (*simp add*: *complex-cnj cis-def rcis-def*)

lemma *cmod-cis* [*simp*]:
  assumes $a \neq 0$
  shows *cor* (*cmod a*) $*$ *cis* (*arg a*) = *a*
using *assms*
by (*metis rcis-cmod-arg rcis-def*)

lemma *cis-cmod* [*simp*]:
  assumes $a \neq 0$
  shows *cis* (*arg a*) $*$ *cor* (*cmod a*) = *a*
using *assms cmod-cis*[*of a*]
by (*simp add*: *field-simps*)

lemma *cor-squared*: $(cor\ x)^2 = cor\ (x^2)$
by (*simp add*: *power2-eq-square*)

lemma *cor-add*: *cor* (*a* + *b*) = *cor a* + *cor b*
by *auto*

lemma *cor-mult*: *cor* (*a* $*$ *b*) = *cor a* $*$ *cor b*
by *auto*

lemma *cor-sqrt-mult-cor-sqrt* [*simp*]:
  shows *cor* (*sqrt A*) $*$ *cor* (*sqrt A*) = *cor* $|A|$

using *assms*
by (*metis cor-mult real-sqrt-abs2 real-sqrt-mult-distrib2*)

lemma [*simp*]: (*Complex a b*) ∗ *2* = *Complex* (*2*∗*a*) (*2*∗*b*)
by (*metis complex-add mult-2 mult-2-right*)

lemma *re-complex*:
Complex (*Re z*) *0* = (*z* + *cnj z*)/*2*
by (*cases z*) *simp*

lemma *im-complex*:
Complex 0 (*Im z*) = (*z* − *cnj z*)/*2*
by (*cases z*) *simp*

lemma *Complex-scale1*: *Complex* (*a* ∗ *b*) (*a* ∗ *c*) = *cor a* ∗ *Complex b c*
unfolding *complex-of-real-def*
by *auto*

lemma *Complex-scale2*: *Complex* (*a* ∗ *c*) (*b* ∗ *c*) = *Complex a b* ∗ *cor c*
unfolding *complex-of-real-def*
by *auto*

lemma *Complex-scale3*: *Complex* (*a* / *b*) (*a* / *c*) = *cor a* ∗ *Complex* (*1* / *b*) (*1* / *c*)
unfolding *complex-of-real-def*
by *auto*

lemma *Complex-scale4*: *c* ≠ *0* ⟹ *Complex* (*a* / *c*) (*b* / *c*) = *Complex a b* / *cor c*
unfolding *complex-of-real-def*
by (*auto simp add*: *field-simps*)

lemma *complex-mult-cnj-cmod*:
z ∗ *cnj z* = *cor* ((*cmod z*)^2)
by (*cases z*) (*simp add*: *complex-of-real-def*, *simp add*: *power2-eq-square*)

lemma
cmod-square: (*cmod z*)^2 = *Re* (*z* ∗ *cnj z*)
using *complex-mult-cnj-cmod*[*of z*]
by (*simp add*: *power2-eq-square*)

lemma *cnjE*:
assumes *x* ≠ *0*
shows *cnj x* = *cor* ((*cmod x*)^2) / *x*
using *complex-mult-cnj-cmod*[*of x*] *assms*

```
by (auto simp add: field-simps)

lemma cmod-mult [simp]: cmod (a * b) = cmod a * cmod b
unfolding cmod-def
by (metis complex-norm-def norm-mult)

lemma cmod-divide [simp]: cmod (a / b) = cmod a / cmod b
unfolding cmod-def
by (metis complex-norm-def norm-divide)

lemma [simp]: cmod (z / cor k) = cmod z / |k|
by auto

lemma [simp]: cmod (z*z1 − z*z2) = cmod z*cmod(z1 − z2)
by (metis bounded-bilinear.diff-right bounded-bilinear-mult cmod-mult)

lemma cmod-eqI:
  assumes z1 * cnj z1 = z2 * cnj z2
  shows cmod z1 = cmod z2
using assms
by (subst complex-mod-sqrt-Re-mult-cnj)+ auto

lemma cmod-eqE:
  assumes cmod z1 = cmod z2
  shows z1 * cnj z1 = z2 * cnj z2
proof−
  from assms have cor ((cmod z1)^2) = cor ((cmod z2)^2)
    by auto
  thus ?thesis
    using complex-mult-cnj-cmod
    by auto
qed

lemma [simp]: cmod a = 1 ⟷ a*cnj a = 1
by (metis cmod-eqE cmod-eqI complex-cnj-one monoid-mult-class.mult.left-neutral
norm-one)



abbreviation is-real where
 is-real z ≡ Im z = 0

lemma complex-eq-if-Re-eq:
  assumes is-real z1 is-real z2
  shows z1 = z2 ⟷ Re z1 = Re z2
using assms
by (cases z1, cases z2) auto

lemma mult-reals:
```

assumes *is-real a is-real b*
shows *is-real* (*a* * *b*)
using *assms*
by *auto*

lemma *div-reals*:
assumes *is-real a is-real b*
shows *is-real* (*a* / *b*)
using *assms*
by (*simp add*: *divide-inverse complex-inverse-def*)

lemma *complex-of-real-Re*:
assumes *is-real k*
shows *cor* (*Re k*) = *k*
using *assms*
by (*cases k*) (*auto simp add*: *complex-of-real-def*)

lemma *is-real-complex-of-real*:
is-real (*cor x*)
by *auto*

lemma *cor-cmod-real*:
assumes *is-real a*
shows *cor* (*cmod a*) = *a* ∨ *cor* (*cmod a*) = −*a*
using *assms*
unfolding *cmod-def*
by (*cases Re a* > *0*) (*auto*, (*metis complex-of-real-Re*)+)

lemma *eq-cnj-iff-real*:
z = *cnj z* ⟷ *is-real z*
by (*cases z*) *auto*

lemma *Re-divide-real*:
assumes *is-real b b* ≠ *0*
shows *Re* (*a* / *b*) = (*Re a*) / (*Re b*)
using *assms*
unfolding *complex-divide-def*
by (*cases a*, *cases b*) (*auto simp add*: *field-simps power2-eq-square*)

lemma *Re-mult-real*:
assumes *is-real a*
shows *Re* (*a* * *b*) = (*Re a*) * (*Re b*)
using *assms*
by *auto*

lemma *Im-mult-real*:
assumes *is-real a*
shows *Im* (*a* * *b*) = (*Re a*) * (*Im b*)
using *assms*

```
by auto

lemma Im-divide-real:
  assumes is-real b b ≠ 0
  shows Im (a / b) = (Im a) / (Re b)
using assms
by (cases a, cases b) (auto simp add: complex-divide-def field-simps power2-eq-square)

lemma [simp]: Re (x / 2) = Re x / 2
  using Re-divide-real[of 2 x]
  by simp

lemma [simp]: Re (2 ∗ x) = 2 ∗ Re x
  using Re-mult-real[of 2 x]
  by simp

lemma Re-sgn:
  assumes is-real R
  shows Re (sgn R) = sgn (Re R)
using assms
by (metis Re-sgn complex-of-real-Re norm-of-real real-sgn-eq)


abbreviation rot90 where
  rot90 z ≡ Complex (−Im z) (Re z)

lemma rot90-ii: rot90 z = z ∗ ii
by (cases z) simp



abbreviation cnj-mix where
  cnj-mix z1 z2 ≡ cnj z1 ∗ z2 + z1 ∗ cnj z2

lemma cnj-mix-minus:
  shows cnj z1∗z2 − z1∗cnj z2 = ii ∗ cnj-mix (rot90 z1) z2
using assms
by (cases z1, cases z2) simp

lemma cnj-mix-minus':
  shows cnj z1∗z2 − z1∗cnj z2 = rot90 (cnj-mix (rot90 z1) z2)
using assms
by (cases z1, cases z2) simp

lemma cnj-mix-real:
  is-real (cnj-mix z1 z2)
by (cases z1, cases z2) simp

abbreviation scalprod where
```

```
  scalprod z1 z2 ≡ cnj-mix z1 z2 / 2


lemma cos-periodic-pi2:  cos (pi + x) = − cos x
  using cos-periodic-pi[of x]
  by (simp add: field-simps)

lemma cos-periodic-pi3: cos (x − pi) = − cos x
  by (smt cos-periodic-pi)

lemma cos-periodic-4 [simp]: cos (pi − x) = − cos x
by (metis cos-minus cos-periodic-pi2 minus-real-def)

lemma sin-periodic-pi3: sin (x − pi) = − sin x
  by (smt sin-periodic-pi)

lemma cos-lt-zero:
  assumes x > pi/2 x ≤ pi
  shows cos x < 0
  using cos-gt-zero-pi[of pi − x] assms
  by simp

lemma sin-kpi:
  fixes k::int
  shows sin (real k ∗ pi) = 0
using sin-npi[of nat k]
using sin-npi[of nat (−k)]
by (cases k ≥ 0) auto

lemma cos-kpi-odd:
  fixes k::int
  assumes odd k
  shows cos (real k ∗ pi) = −1
proof (cases k ≥ 0)
  case True
  hence odd (nat k)
    using ‹odd k›
    by (metis pos-int-even-equiv-nat-even)
  thus ?thesis
    using ‹k ≥ 0› cos-npi[of nat k]
    by auto
next
  case False
  hence −k ≥ 0 odd (nat (−k))
    using ‹odd k›
    by (auto, smt even-neg pos-int-even-equiv-nat-even)
  thus ?thesis
    using cos-npi[of nat (−k)]
    by auto
qed

lemma cos-kpi-even:
  fixes k::int
  assumes even k
  shows cos (real k ∗ pi) = 1
proof (cases k ≥ 0)
  case True
  hence even (nat k)
    using ‹even k›
    by (metis pos-int-even-equiv-nat-even)
  thus ?thesis
    using ‹k ≥ 0› cos-npi[of nat k]
    by auto
next
  case False
  hence −k ≥ 0 even (nat (−k))
    using ‹even k›
    by (auto, smt even-neg pos-int-even-equiv-nat-even)
  thus ?thesis
    using  cos-npi[of nat (−k)]
    by auto
qed

lemma sin-pi2-kpi-odd:
  fixes k::int
  assumes odd k
  shows sin (pi / 2 + real k ∗ pi) = −1
using assms
by (simp add: sin-add cos-kpi-odd)

lemma sin-pi2-kpi-even:
  fixes k::int
  assumes even k
  shows sin (pi / 2 + real k ∗ pi) = 1
using assms
by (simp add: sin-add cos-kpi-even)

lemma cos-zero-iff-int:
  shows cos x = 0 ⟷ (∃ k::int. odd k ∧ x = real k ∗ (pi / 2))
proof
  assume cos x = 0
  then obtain n::nat where ∗: x = real n ∗ (pi / 2) ∨ x = − (real n ∗ (pi / 2))
and odd n
    using cos-zero-iff[of x]
    by blast
   hence (odd (int n) ∧ x = real (int n) ∗ (pi / 2)) ∨ (odd (−int n) ∧ x = real
(− int n) ∗ pi / 2)
```

```
    by (auto simp add: Parity.transfer-int-nat-relations)
  thus ∃ k::int. odd k ∧ x = real k ∗ (pi / 2)
    by (metis times-divide-eq-right)
next
  assume ∃ k::int. odd k ∧ x = real k ∗ (pi / 2)
  then obtain k::int where ∗: odd k x = real k ∗ (pi / 2)
    by blast
  show cos x = 0
  proof (cases k ≥ 0)
    case True
    hence ∃ n::nat. odd n ∧ x = real n ∗ (pi / 2)
      using ∗
      by (rule-tac x=nat k in exI) (auto simp add: pos-int-even-equiv-nat-even)
    thus ?thesis
      using cos-zero-iff [of x]
      by auto
  next
    case False
    hence ∃ n::nat. odd n ∧ x = − (real n ∗ (pi / 2))
      using ∗
    by (rule-tac x=nat (−k) in exI, auto) (smt even-neg pos-int-even-equiv-nat-even)
    thus ?thesis
      using cos-zero-iff [of x]
      by auto
  qed
qed

lemma sin-zero-iff-int:
  sin x = 0 ⟷ (∃ k::int. even k ∧ x = real k ∗ (pi / 2))
proof−
  have sin x = 0 ⟷ cos (x − pi/2) = 0
    using cos-minus[of x − pi/2]
    by (simp add: sin-cos-eq)
  hence sin x = 0 ⟷ (∃ k::int. odd k ∧ x − pi/2 = real k ∗ (pi / 2))
    using cos-zero-iff-int
    by simp
  thus ?thesis
    by auto (rule-tac x=k+1 in exI, simp add: field-simps, rule-tac x=k−(1::int)
in exI, simp add: field-simps)
qed

lemma cos0-sin1:
  assumes cos φ = 0 sin φ = 1
  shows ∃ k::int. φ = pi/2 + 2∗k∗pi
proof−
  from ‹cos φ = 0›
  obtain k::int where odd k φ = real k ∗ (pi / 2)
    using cos-zero-iff-int[of φ]
    by auto
  then obtain k'::int where k = 2*k' + 1
    by (metis odd-equiv-def)
  hence φ = pi/2 + (real k' * pi)
    using ‹φ = real k * (pi / 2)›
    by (auto simp add: field-simps)
  hence even k'
    using ‹sin φ = 1› sin-pi2-kpi-odd[of k']
    by auto
  thus ?thesis
    using ‹φ = pi /2 + (real k' * pi)›
    unfolding even-def
    by auto
qed

lemma cos-0-iff-normalized:
  assumes cos φ = 0 −pi < φ φ ≤ pi
  shows φ = pi/2 ∨ φ = −pi/2
proof−
  obtain k::int where odd k φ = real k * pi/2
    using cos-zero-iff-int[of φ] assms(1)
    by auto
  thus ?thesis
  proof (cases k > 1 ∨ k < −1)
    case True
    hence k ≥ 3 ∨ k ≤ −3
      using ‹odd k›
    by auto (smt odd-one-int odd-plus-odd, smt odd-one-int odd-plus-even odd-plus-odd)
    hence φ ≥ 3*pi/2 ∨ φ ≤ −3*pi/2
      using ‹φ = real k * pi/2›
      by auto
    thus ?thesis
      using ‹− pi < φ› ‹φ ≤ pi›
      by auto
  next
    case False
    hence k = −1 ∨ k = 0 ∨ k = 1
      by auto
    hence k = −1 ∨ k = 1
      using ‹odd k›
      by auto
    thus ?thesis
      using ‹φ = real k * pi/2›
      by auto
  qed
qed

lemma sin-0-iff-normalized:
  assumes sin φ = 0 −pi < φ φ ≤ pi
  shows φ = 0 ∨ φ = pi
```

```
proof−
  obtain k::int where even k φ = real k ∗ pi/2
    using sin-zero-iff-int[of φ] assms(1)
    by auto
  thus ?thesis
  proof (cases k > 2 ∨ k < 0)
    case True
    hence k ≥ 4 ∨ k ≤ −2
      using ‹even k›
      by auto (smt even-difference odd-one-int)+
    hence φ ≥ 2∗pi ∨ φ ≤ −pi
    proof
      assume 4 ≤ k
      hence 4 ∗ pi/2 ≤ φ
        by (subst ‹φ = real k ∗ pi/2›) auto
      thus ?thesis
        by simp
    next
      assume k ≤ −2
      hence real k ≤ −2
        by simp
      hence −2∗pi/2 ≥ φ
            by (subst ‹φ = real k ∗ pi/2›, metis mult-right-mono pi-half-ge-zero
times-divide-eq-right)
      thus ?thesis
        by simp
    qed
    thus ?thesis
      using ‹− pi < φ› ‹φ ≤ pi›
      by auto
  next
    case False
    hence k = 0 ∨ k = 1 ∨ k = 2
      by auto
    hence k = 0 ∨ k = 2
      using ‹even k›
      by auto
    thus ?thesis
      using ‹φ = real k ∗ pi/2›
      by auto
  qed
qed

lemma cos1-sin0:
  assumes cos φ = 1 sin φ = 0
  shows ∃ k::int. φ = 2∗k∗pi
proof−
  from ‹sin φ = 0›
  obtain k::int where even k φ = real k ∗ (pi / 2)
```

```
    using sin-zero-iff-int[of φ]
    by auto
  then obtain k′::int where k = 2∗k′
    by (metis even-equiv-def)
  hence φ = real k′ ∗ pi
    using ‹φ = real k ∗ (pi / 2)›
    by (auto simp add: field-simps)
  hence even k′
    using ‹cos φ = 1› cos-kpi-odd[of k′]
    by auto
  thus ?thesis
    using ‹φ = real k′ ∗ pi›
    unfolding even-def
    by auto
qed



lemma sin-cos-eq:
  fixes a b :: real
  assumes cos a = cos b sin a = sin b
  shows ∃ k::int. a − b = 2∗k∗pi
proof−
  from assms have sin (a − b) = 0 cos (a − b) = 1
    using sin-diff[of a b] cos-diff[of a b]
    by auto
  thus ?thesis
    using cos1-sin0
    by auto
qed

lemma sin-monotone-2pi: assumes − (pi / 2) ≤ y and y < x and x ≤ pi / 2
shows sin y < sin x
proof −
  have 0 ≤ y + pi / 2 and y + pi / 2 < x + pi / 2 and x + pi /2 ≤ pi
    using pi-ge-two and assms by auto
 from cos-monotone-0-pi[OF this] show ?thesis unfolding minus-sin-cos-eq[symmetric]
by auto
qed

lemma sin-inj:
  assumes α ≠ α′ −pi/2 ≤ α ∧ α ≤ pi/2 −pi/2 ≤ α′ ∧ α′ ≤ pi/2
  shows sin α ≠ sin α′
using assms
using sin-monotone-2pi[of α α′] sin-monotone-2pi[of α′ α]
by (cases α < α′) auto

lemma arccos-le-pi2:
  assumes a ≥ 0 a ≤ 1
  shows arccos a ≤ pi/2
using assms
by (smt antisym arccos-cos arccos-ubound cos-arccos cos-monotone-0-pi' cos-pi-half
pi-half-ge-zero)

definition atan2 where
  atan2 y x =
    (if x > 0 then arctan (y/x)
     else if x < 0 then
          if y > 0 then arctan (y/x) + pi else arctan (y/x) − pi
     else
          if y > 0 then pi/2 else if y < 0 then −pi/2 else 0)

lemma atan2-bounded: −pi ≤ atan2 y x ∧ atan2 y x < pi
  using arctan-bounded[of y/x] zero-le-arctan-iff[of y/x] arctan-le-zero-iff[of y/x]
zero-less-arctan-iff[of y/x] arctan-less-zero-iff[of y/x]
 using divide-neg-neg[of y x] divide-neg-pos[of y x] divide-pos-pos[of y x] divide-pos-neg[of
y x]
  unfolding atan2-def
  by (simp (no-asm-simp)) auto


lemma cos-periodic-nat[simp]: fixes n :: nat shows cos (x + n ∗ (2 ∗ pi)) = cos
x
proof (induct n arbitrary: x)
  case (Suc n)
  have split-pi-off: x + (Suc n) ∗ (2 ∗ pi) = (x + n ∗ (2 ∗ pi)) + 2 ∗ pi
    unfolding Suc-eq-plus1 real-of-nat-add real-of-one distrib-right by auto
  show ?case unfolding split-pi-off using Suc by auto
qed auto

lemma cos-periodic-int[simp]: fixes i :: int shows cos (x + i ∗ (2 ∗ pi)) = cos x
proof (cases 0 ≤ i)
  case True hence i-nat: real i = nat i by auto
  show ?thesis unfolding i-nat by auto
next
  case False hence i-nat: i = − real (nat (−i)) by auto
  have cos x = cos (x + i ∗ (2 ∗ pi) − i ∗ (2 ∗ pi)) by auto
  also have ... = cos (x + i ∗ (2 ∗ pi))
    unfolding i-nat mult-minus-left diff-minus-eq-add by (rule cos-periodic-nat)
  finally show ?thesis by auto
qed




abbreviation canon-ang-P where
 canon-ang-P α α' ≡ (−pi < α' ∧ α' ≤ pi) ∧ (∃ k::int. α − α' = 2∗k∗pi)
```

```
definition canon-ang :: real ⇒ real (⌊-⌋) where
  ⌊α⌋ = (THE α′. canon-ang-P α α′)

lemma canon-ang-ex:
  shows ∃ α′. canon-ang-P α α′
proof−
  have ∗∗∗: ∀ α::real. ∃ α′. 0 < α′ ∧ α′ ≤ 1 ∧ (∃ k::int. α′ = α − k)
  proof
    fix α::real
    show ∃α′>0. α′ ≤ 1 ∧ (∃k::int. α′ = α − real k)
    proof (cases α = floor α)
      case True
      thus ?thesis
        by (rule-tac x=α − floor α + 1 in exI, auto) (rule-tac x=floor α − 1 in
exI, auto)
    next
      case False
      thus ?thesis
        using  real-of-int-floor-ge-diff-one[of α]
        using real-of-int-floor-le[of α]
         by (rule-tac x=α − floor α in exI) (metis antisym diff-self floor-subtract
le-cases le-iff-diff-le-0 less-int-code(1) not-leE zero-less-floor)
    qed
  qed

  have ∗∗: ∀ α::real. ∃ α′. 0 < α′ ∧ α′ ≤ 2 ∧ (∃ k::int. α − α′ = 2∗k − 1)
  proof
    fix α::real
    from ∗∗∗[rule-format, of (α + 1) /2]
    obtain α′ and k::int where 0 < α′ α′ ≤ 1 α′ = (α + 1)/2 − k
      by force
    hence 0 < α′ α′ ≤ 1 α′ = α/2 − k + 1/2
      by auto
    thus ∃α′>0. α′ ≤ 2 ∧ (∃k::int. α − α′ = real (2 ∗ k − 1))
      by (rule-tac x=2∗α′ in exI) auto
  qed
  have ∗: ∀ α::real. ∃ α′. −1 < α′ ∧ α′ ≤ 1 ∧ (∃ k::int. α − α′ = 2∗k)
  proof
    fix α::real
    from ∗∗ obtain α′ and k :: int where
      0 < α′ ∧ α′ ≤ 2 ∧ α − α′ = 2∗k − 1
      by force
    thus ∃α′>−1. α′ ≤ 1 ∧ (∃k. α − α′ = real (2 ∗ (k::int)))
      by (rule-tac x=α′ − 1 in exI) (auto simp add: field-simps)
  qed
  show ?thesis
    using ∗[rule-format, of α / pi]
    apply auto
    apply (rule-tac x=α'*pi in exI)
   by (auto simp add: field-simps) (metis mult.commute mult-minus1-right not-less
pi-gt-zero real-mult-le-cancel-iff2)
qed

lemma canon-ang-unique:
  assumes canon-ang-P α α' canon-ang-P α α''
  shows α' = α''
proof–
  obtain k1::int where α − α' = 2*k1*pi
    using assms(1)
    by auto
  obtain k2::int where α − α'' = 2*k2*pi
    using assms(2)
    by auto
  hence −α' + α'' = 2*(k1 − k2)*pi
    using ‹α − α' = 2*k1*pi›
    by (simp add:field-simps)
  moreover
  have −α' + α'' < 2 * pi −α' + α'' > −2*pi
    using assms
    by auto
  ultimately
  have −α' + α'' = 0
    by auto
  thus ?thesis
    by auto
qed

lemma canon-ang:

  −pi < ⇂α⇃ ⇂α⇃ ≤ pi ∃ k::int. α − ⇂α⇃ = 2*k*pi
proof–
  obtain α' where canon-ang-P α α'
    using canon-ang-ex[of α]
    by auto
  have canon-ang-P α ⇂α⇃
    unfolding canon-ang-def
  proof (rule theI[where a=α'])
    show canon-ang-P α α'
      by fact
  next
    fix α''
    assume canon-ang-P α α''
    thus α'' = α'
      using ‹canon-ang-P α α'›
      using canon-ang-unique[of α' α α'']
      by simp
  qed
  thus −pi < ⇂α⇃ ⇂α⇃ ≤ pi ∃ k::int. α − ⇂α⇃ = 2*k*pi
    by auto
qed

lemma canon-ang-id:
assumes  −pi < α ∧ α ≤ pi
  shows ⇂α⇃ = α
  using assms
  using canon-ang-unique[of canon-ang α α α] canon-ang[of α]
  by auto

lemma canon-ang-eq:
  assumes ∃ k::int. α′ − α″ = 2*k*pi
  shows ⇂α′⇃ = ⇂α″⇃
proof−
  obtain k′::int where *: − pi < ⇂α′⇃ ⇂α′⇃ ≤ pi α′ − ⇂α′⇃ = 2 * real k′ * pi
    using canon-ang[of α′]
    by auto

  obtain k″::int where **: − pi < ⇂α″⇃ ⇂α″⇃ ≤ pi α″ − ⇂α″⇃ = 2 * real k″ * pi
    using canon-ang[of α″]
    by auto

  obtain k::int where ***: α′ − α″ = 2*k*pi
    using assms
    by auto

  have ∃ m::int. α′ − ⇂α″⇃ = 2 * m * pi
    using **(3) ***
    by (rule-tac x=k+k″ in exI) (auto simp add: field-simps)

  thus ?thesis
    using canon-ang-unique[of ⇂α′⇃ α′ ⇂α″⇃] * **
    by auto
qed

lemma canon-ang-eqI:
  assumes ∃ k::int. α′ − α = 2 * k * pi − pi < α′ ∧ α′ ≤ pi
  shows ⇂α⇃ = α′
using assms
using canon-ang-eq[of α′ α]
using canon-ang-id[of α′]
by auto

lemma canon-ang-arg:
  ⇂arg z⇃ = arg z
using canon-ang-id[of arg z] arg-bounded
by simp
```

```
lemma canon-ang-uminus:
  assumes ⇂α⇃ ≠ pi
  shows ⇂−α⇃ = −⇂α⇃
proof (rule canon-ang-eqI)
  show ∃x::int. − ⇂α⇃ − − α = 2 ∗ real x ∗ pi
    using canon-ang(3)[of α]
    by (metis minus-diff-eq minus-diff-minus)
next
  show − pi < − ⇂α⇃ ∧ − ⇂α⇃ ≤ pi
    using canon-ang(1)[of α] canon-ang(2)[of α] assms
    by auto
qed

lemma canon-ang-uminus-pi:
  assumes ⇂α⇃ = pi
  shows ⇂−α⇃ = ⇂α⇃
proof (rule canon-ang-eqI)
  obtain k::int where α − ⇂α⇃ = 2 ∗ real k ∗ pi
    using canon-ang(3)[of α]
    by auto
  thus ∃x::int. ⇂α⇃ − − α = 2 ∗ real x ∗ pi
    using assms
    by (rule-tac x=k+(1::int) in exI) (auto simp add: field-simps)
next
  show − pi < ⇂α⇃ ∧ ⇂α⇃ ≤ pi
    using assms
    by auto
qed

lemma canon-ang-diff:
  ⇂α − β⇃ = ⇂⇂α⇃ − ⇂β⇃⇃
proof (rule canon-ang-eq)
  show ∃x::int. α − β − (⇂α⇃ − ⇂β⇃) = 2 ∗ real x ∗ pi
  proof−
    obtain k1::int where α − ⇂α⇃ = 2∗k1∗pi
      using canon-ang(3)
      by auto
    moreover
    obtain k2::int where β − ⇂β⇃ = 2∗k2∗pi
      using canon-ang(3)
      by auto
    ultimately
    show ?thesis
      by (rule-tac x=k1 − k2 in exI) (auto simp add: field-simps)
  qed
qed

lemma canon-ang-sum:
  ⇂α + β⇃ = ⇂⇂α⇃ + ⇂β⇃⇃
```

```
proof (rule canon-ang-eq)
  show ∃x::int. α + β − (⇂α⇃ + ⇂β⇃) = 2 ∗ real x ∗ pi
  proof−
    obtain k1::int where α − ⇂α⇃ = 2∗k1∗pi
      using canon-ang(3)
      by auto
    moreover
    obtain k2::int where β − ⇂β⇃ = 2∗k2∗pi
      using canon-ang(3)
      by auto
    ultimately
    show ?thesis
      by (rule-tac x=k1 + k2 in exI) (auto simp add: field-simps)
  qed
qed

lemma canon-ang-plus-pi1:
  assumes 0 < α α ≤ 2∗pi
  shows ⇂α + pi⇃ = α − pi
proof (rule canon-ang-eqI)
  show ∃ x::int. α − pi − (α + pi) = 2 ∗ real x ∗ pi
    by (rule-tac x=−1 in exI) auto
next
  show − pi < α − pi ∧ α − pi ≤ pi
    using assms
    by auto
qed

lemma canon-ang-plus-pi2:
  assumes −2∗pi < α α ≤ 0
  shows ⇂α + pi⇃ = α + pi
proof (rule canon-ang-id)
  show − pi < α + pi ∧ α + pi ≤ pi
    using assms
    by auto
qed

lemma canon-ang-minus-pi1:
  assumes 0 < α α ≤ 2∗pi
  shows ⇂α − pi⇃ = α − pi
proof (rule canon-ang-id)
  show − pi < α − pi ∧ α − pi ≤ pi
    using assms
    by auto
qed

lemma canon-ang-minus-pi2:
  assumes −2∗pi < α α ≤ 0
  shows ⇂α − pi⇃ = α + pi
```

```
proof (rule canon-ang-eqI)
  show ∃ x::int. α + pi − (α − pi) = 2 ∗ real x ∗ pi
    by (rule-tac x=1 in exI) auto
next
  show − pi < α + pi ∧ α + pi ≤ pi
    using assms
    by auto
qed

lemma [simp]: ⇂0⇃ = 0
  using canon-ang-eqI[of 0 0]
  by simp

lemma canon-ang-cos [simp]: cos ⇂α⇃ = cos α
proof−
  obtain x::int where α = ⇂α⇃ + pi ∗ (real x ∗ 2)
    using canon-ang(3)[of α]
    by (auto simp add: field-simps)
  thus ?thesis
    using cos-periodic-int[of ⇂α⇃ x]
    by (simp add: field-simps)
qed



lemma [simp]: cis φ ∗ cis (− φ) = 1
by (metis cis-mult cis-zero right-minus)

lemma cis-eq:
  assumes cis a = cis b
  shows ∃ k::int. a − b = 2 ∗ k ∗ pi
using assms sin-cos-eq[of a b]
using Re-cis[of a] Re-cis[of b] Im-cis[of a] Im-cis[of b]
by (cases cis a, cases cis b) auto

lemma cis-inj:
  assumes cis α = cis α' −pi < α α ≤ pi −pi < α' α' ≤ pi
  shows α = α'
using assms
by (metis arg-unique sgn-cis)



lemma re-complex-zero-arg1:
  assumes arg z = pi/2 ∨ arg z = −pi/2
  shows Re z = 0
using assms
using rcis-cmod-arg[of z] Re-rcis[of cmod z arg z]
by (metis cos-minus cos-pi-half minus-divide-left mult-eq-0-iff)
```

```
lemma re-complex-zero-arg2:
  assumes Re z = 0 z ≠ 0
  shows arg z = pi/2 ∨ arg z = −pi/2
proof−
  have cos (arg z) = 0
    using assms
    by (metis Re-rcis no-zero-divisors norm-eq-zero rcis-cmod-arg)
  thus ?thesis
    using arg-bounded[of z]
    using cos-0-iff-normalized[of arg z]
    by simp
qed

lemma im-complex-zero-arg1:
  assumes arg z = 0 ∨ arg z = pi
  shows Im z = 0
using assms
using rcis-cmod-arg[of z] Im-rcis[of cmod z arg z]
by auto

lemma im-complex-zero-arg2:
  assumes Im z = 0
  shows arg z = 0 ∨ arg z = pi
proof (cases z = 0)
  case True
  thus ?thesis
    by (auto simp add: arg-zero)
next
  case False
  hence sin (arg z) = 0
    using assms rcis-cmod-arg[of z] Im-rcis[of cmod z arg z]
    by auto
  thus ?thesis
    using arg-bounded[of z]
    using sin-0-iff-normalized
    by simp
qed

lemma arg-complex-of-real-positive:
  assumes k > 0
  shows arg (cor k) = 0
proof−
  have cos (arg (Complex k 0)) > 0
    using assms
      using rcis-cmod-arg[of Complex k 0] Re-rcis[of cmod (Complex k 0) arg (Complex k 0)]
    by auto
  thus ?thesis
```

```
    using assms im-complex-zero-arg2[of cor k]
    unfolding complex-of-real-def
    by auto
qed

lemma arg-complex-of-real-negative:
  assumes k < 0
  shows arg (cor k) = pi
proof−
  have cos (arg (Complex k 0)) < 0
      using rcis-cmod-arg[of Complex k 0] Re-rcis[of cmod (Complex k 0) arg
(Complex k 0)]
   by auto (metis assms less-asym′ mult-eq-0-iff mult-pos-pos neqE zero-less-abs-iff)
  thus ?thesis
    using assms im-complex-zero-arg2[of cor k]
    unfolding complex-of-real-def
    by auto
qed

lemma
  [simp]: arg ii = pi/2
proof−
  have ii = cis (arg ii)
    using rcis-cmod-arg[of ii]
    by (simp add: rcis-def)
  hence cos (arg ii) = 0 sin (arg ii) = 1
    by (metis Re-cis complex-Re-i, metis Im-cis complex-Im-i)
  thus ?thesis
    using cos-0-iff-normalized[of arg ii] arg-bounded[of ii]
    by (auto simp add: field-simps)
qed

lemma
  [simp]: arg (−ii) = −pi/2
proof−
  have −ii = cis (arg (− ii))
    using rcis-cmod-arg[of −ii]
    by (simp add: rcis-def)
  hence cos (arg (−ii)) = 0 sin (arg (−ii)) = −1
    using Re-cis[of arg (−ii)] Im-cis[of arg (−ii)]
    by auto
  thus ?thesis
    using cos-0-iff-normalized[of arg (−ii)] arg-bounded[of −ii]
    by (metis one-neq-neg-numeral sin-pi-half)
qed

lemma arg-cis:
  shows arg (cis φ) = ⌊φ⌋
proof (rule canon-ang-eqI[symmetric])
```

```
  show − pi < arg (cis φ) ∧ arg (cis φ) ≤ pi
    using arg-bounded
    by simp
next
  show ∃ k::int. arg (cis φ) − φ = 2∗k∗pi
  proof−
    have cis (arg (cis φ)) = cis φ
      using cis-arg[of cis φ]
      by auto
    thus ?thesis
      using cis-eq
      by auto
  qed
qed

lemma cos-arg:
  assumes z ≠ 0
  shows cos (arg z) = Re z / cmod z
by (metis Complex.Re-sgn Re-cis assms cis-arg)

lemma sin-arg:
  assumes z ≠ 0
  shows sin (arg z) = Im z / cmod z
by (metis Complex.Im-sgn Im-cis assms cis-arg)


lemma cis-arg-mult:
  assumes a ∗ z ≠ 0
  shows cis (arg (a ∗ z)) = cis (arg a + arg z)
proof−
  have a ∗ z = cor (cmod a) ∗ cor (cmod z) ∗ cis (arg a) ∗ cis (arg z)
    using rcis-cmod-arg[of z, symmetric] rcis-cmod-arg[of a, symmetric]
    unfolding rcis-def
    by algebra
  hence a ∗ z = cor (cmod (a ∗ z)) ∗ cis (arg a + arg z)
    using cis-mult[of arg a arg z]
    by auto
  hence cor (cmod (a ∗ z)) ∗ cis (arg a + arg z) = cor (cmod (a ∗ z)) ∗ cis (arg
(a ∗ z))
    using assms
    using rcis-cmod-arg[of a∗z]
    unfolding rcis-def
    by auto
  thus ?thesis
    using mult-cancel-left[of cor (cmod (a ∗ z)) cis (arg a + arg z) cis (arg (a ∗
z))]
    using assms
    by auto
qed
```

```
lemma arg-mult-2kpi:
  assumes a ∗ z ≠ 0
  shows ∃ k::int. arg (a ∗ z) = arg a + arg z + 2∗k∗pi
proof−
  have cis (arg (a∗z)) = cis (arg a + arg z)
    by (rule cis-arg-mult[OF assms])
  thus ?thesis
    using cis-eq[of arg (a∗z) arg a + arg z]
    by (auto simp add: field-simps)
qed

lemma arg-mult:
  assumes z1 ∗ z2 ≠ 0
  shows arg(z1 ∗ z2) = ⇂arg z1 + arg z2⇃
proof−
  obtain k::int where arg(z1 ∗ z2) = arg z1 + arg z2 + 2∗k∗pi
    using arg-mult-2kpi[of z1 z2]
    using assms
    by auto
  hence ⇂arg(z1 ∗ z2)⇃ = ⇂arg z1 + arg z2⇃
    using canon-ang-eq
    by(simp add:field-simps)
  thus ?thesis
    using canon-ang-arg[of z1∗z2]
    by auto
qed

lemma arg-mult-real-positive:
  assumes k > 0
  shows arg (cor k ∗ z) = arg z
proof (cases z = 0)
  case True
  thus ?thesis
    by (auto simp add: arg-zero)
next
  case False
  thus ?thesis
    using assms
    using arg-mult[of cor k z]
    by (auto simp add: arg-complex-of-real-positive canon-ang-arg)
qed

lemma arg-mult-real-negative:
  assumes k < 0
  shows arg (cor k ∗ z) = arg (−z)
proof (cases z = 0)
  case True
  thus ?thesis
```

```
    by (auto simp add: arg-zero)
next
  case False
  thus ?thesis
    using assms
    using arg-mult[of cor k z]
    using arg-mult[of −1 z]
    using arg-complex-of-real-negative[of k] arg-complex-of-real-negative[of −1]
    by auto
qed
```

```
lemma arg-cnj1:
  assumes arg z = pi
  shows arg (cnj z) = pi
proof−
  have cos (arg (cnj z)) = cos (arg z)
    using rcis-cmod-arg[of z, symmetric] Re-rcis[of cmod z arg z]
    using rcis-cmod-arg[of cnj z, symmetric] Re-rcis[of cmod (cnj z) arg (cnj z)]
    by auto
  hence arg (cnj z) = arg z ∨ arg(cnj z) = −arg z
    using arg-bounded[of z] arg-bounded[of cnj z]
    by (metis arccos-cos arccos-cos2 less-eq-real-def linorder-le-cases minus-minus)
  thus ?thesis
    using assms
    using arg-bounded[of cnj z]
    by auto
qed

lemma arg-cnj2:
  assumes arg z ≠ pi
  shows arg (cnj z) = −arg z
proof(cases arg z = 0)
  case True
  thus ?thesis
    by (metis cnj-def complex-surj im-complex-zero-arg1 minus-zero)
next
  case False
  have cos (arg (cnj z)) = cos (arg z)
    using rcis-cmod-arg[of z] Re-rcis[of cmod z arg z]
    using rcis-cmod-arg[of cnj z] Re-rcis[of cmod (cnj z) arg (cnj z)]
    by auto
  hence arg (cnj z) = arg z ∨ arg(cnj z) = −arg z
    using arg-bounded[of z] arg-bounded[of cnj z]
    by (metis arccos-cos arccos-cos2 less-eq-real-def linorder-le-cases minus-minus)
  moreover
  have sin (arg (cnj z)) = −sin (arg z)
    using rcis-cmod-arg[of z] Im-rcis[of cmod z arg z]
    using rcis-cmod-arg[of cnj z] Im-rcis[of cmod (cnj z) arg (cnj z)]
   by auto (metis complex-Im-cnj complex-Im-zero complex-mod-cnj im-complex-zero-arg2
minus-mult-right norm-eq-zero real-mult-left-cancel sin-pi sin-zero)
  hence arg (cnj z) ≠ arg z
    using sin-0-iff-normalized[of arg (cnj z)] arg-bounded False assms
    by auto
  ultimately
  show ?thesis
    by auto
qed


lemma arg-div-real-positive:
  assumes k ≠ 0 k > 0
  shows arg (z / cor k) = arg z
proof(cases z = 0)
  case True
  thus ?thesis
    by auto
next
  case False
  thus ?thesis
    using assms
    using arg-mult-real-positive[of 1/k z]
    by auto
qed

lemma arg-inv1:
  assumes z ≠ 0 arg z ≠ pi
  shows arg (1 / z) = − arg z
proof−
  have 1/z = cnj z / cor ((cmod z)^2 )
    using ‹z ≠ 0› complex-mult-cnj-cmod[of z]
    by (auto simp add:field-simps)
  thus ?thesis
    using arg-div-real-positive[of (cmod z)^2 cnj z] ‹z ≠ 0›
    using arg-cnj2[of z] ‹arg z ≠ pi›
    by auto
qed

lemma arg-inv2:
  assumes z ≠ 0 arg z = pi
  shows arg (1 / z) = pi
proof−
  have 1/z = cnj z / cor ((cmod z)^2 )
    using ‹z ≠ 0› complex-mult-cnj-cmod[of z]
    by (auto simp add:field-simps)
  thus ?thesis
    using arg-div-real-positive[of (cmod z)^2 cnj z] ‹z ≠ 0›
    using arg-cnj1[of z] ‹arg z = pi›
    by auto
qed

lemma arg-inv-2kpi:
  assumes z ≠ 0
  shows ∃ k::int. arg (1 / z) = − arg z + 2∗k∗pi
using arg-inv1[OF assms]
using arg-inv2[OF assms]
by (cases arg z = pi) (rule-tac x=1 in exI, simp, rule-tac x=0 in exI, simp)

lemma arg-inv:
  assumes z ≠ 0
  shows arg (1 / z) = ⌊− arg z⌋
proof−
  obtain k::int where arg(1 / z) = − arg z + 2∗k∗pi
    using arg-inv-2kpi[of z]
    using assms
    by auto
  hence ⌊arg(1 / z)⌋ = ⌊− arg z⌋
    using canon-ang-eq
    by(simp add:field-simps)
  thus ?thesis
    using canon-ang-arg[of 1 / z]
    by auto
qed



lemma arg-div-2kpi:
  assumes z1 ≠ 0 z2 ≠ 0
  shows ∃ k::int. arg (z1 / z2) = arg z1 − arg z2 + 2∗k∗pi
using assms
unfolding complex-divide-def[of z1 z2]
using inverse-eq-divide[of z2]
using arg-mult-2kpi[of z1 1/z2]
using arg-inv-2kpi[of z2]
by auto (metis comm-semiring-class.distrib distrib-left-numeral real-of-int-add)

lemma arg-div:
  assumes z1 ≠ 0 z2 ≠ 0
  shows arg(z1 / z2) = ⌊arg z1 − arg z2⌋
proof−
  obtain k::int where arg(z1 / z2) = arg z1 − arg z2 + 2∗k∗pi
    using arg-div-2kpi[of z1 z2]
    using assms
    by auto
  hence canon-ang(arg(z1 / z2)) = canon-ang(arg z1 − arg z2)
    using canon-ang-eq
    by(simp add:field-simps)
  thus ?thesis
    using canon-ang-arg[of z1/z2]
    by auto
qed
```

```
lemma arg-uminus:
  assumes z ≠ 0
  shows arg (−z) = ⌊arg z + pi⌋
using assms
using arg-mult[of −1 z]
using arg-complex-of-real-negative[of −1]
by auto (metis comm-semiring-1-class.normalizing-semiring-rules(24))
```

```
definition
  csqrt z = rcis (sqrt (cmod z)) (arg z / 2)

lemma [simp]: (csqrt x)^2 = x
  unfolding csqrt-def
  by (subst DeMoivre2) (simp add: rcis-cmod-arg)

lemma ex-complex-sqrt: ∃ s::complex. s∗s = z
  unfolding power2-eq-square[symmetric]
  by (rule-tac x=csqrt z in exI) simp

lemma csqrt:
  assumes s ∗ s = z
  shows s = csqrt z ∨ s = −csqrt z
proof (cases s = 0)
  case True
  thus ?thesis
    using assms
    unfolding csqrt-def
    by simp
next
  case False
   then obtain k::int where cmod s ∗ cmod s = cmod z 2 ∗ arg s − arg z =
2∗k∗pi
    using assms
    using rcis-cmod-arg[of z] rcis-cmod-arg[of s]
    using arg-mult[of s s]
    using canon-ang(3)[of 2∗arg s]
    by (auto simp add: norm-mult arg-mult)
  have ∗: sqrt (cmod z) = cmod s
```

```
    using ‹cmod s * cmod s = cmod z›
    by (smt norm-not-less-zero real-sqrt-abs2)

  have **: arg z / 2 = arg s - k*pi
    using ‹2 * arg s - arg z = 2*k*pi›
    by simp

  have cis (arg s - k*pi) = cis (arg s) ∨ cis (arg s - k*pi) = -cis (arg s)
  proof (cases even k)
    case True
    hence cis (arg s - k*pi) = cis (arg s)
      by (simp add: cis-def cos-diff sin-diff cos-kpi-even sin-kpi)
    thus ?thesis
      by simp
  next
    case False
    hence cis (arg s - k*pi) = -cis (arg s)
      by (simp add: cis-def cos-diff sin-diff cos-kpi-odd sin-kpi)
    thus ?thesis
      by simp
  qed
  thus ?thesis
  proof
    assume ***: cis (arg s - real k * pi) = cis (arg s)
    hence s = csqrt z
      using rcis-cmod-arg[of s]
      unfolding csqrt-def rcis-def
      by (subst *, subst **, subst ***, simp)
    thus ?thesis
      by simp
  next
    assume ***: cis (arg s - real k * pi) = -cis (arg s)
    hence s = - csqrt z
      using rcis-cmod-arg[of s]
      unfolding csqrt-def rcis-def
      by (subst *, subst **, subst ***, simp)
    thus ?thesis
      by simp
  qed
qed

lemma [simp]: csqrt x = 0 ⟷ x = 0
unfolding csqrt-def
by auto

lemma csqrt-mult: csqrt (a * b) = csqrt a * csqrt b ∨ csqrt (a * b) = - csqrt a
* csqrt b
proof (cases a = 0 ∨ b = 0)
  case True
  thus ?thesis
    by auto
next
  case False
  obtain k::int where *: ⌊arg a + arg b⌋ = arg a + arg b − 2 ∗ real k ∗ pi
    using canon-ang(3)[of arg a + arg b]
    by smt
  have cis (⌊arg a + arg b⌋ / 2) = cis (arg a / 2 + arg b / 2) ∨ cis (⌊arg a +
arg b⌋ / 2) = − cis (arg a / 2 + arg b / 2)
    using cos-kpi-even[of k] cos-kpi-odd[of k]
    by ((subst ∗)+, (subst diff-divide-distrib)+, (subst add-divide-distrib)+)
       (cases even k, auto simp add: cis-def cos-diff sin-diff sin-kpi)
 thus ?thesis
    using False
    unfolding csqrt-def
    by (simp add: rcis-mult real-sqrt-mult arg-mult)
       (auto simp add: rcis-def)
qed

lemma csqrt-real:
  assumes is-real x
  shows (Re x ≥ 0 ∧ csqrt x = cor (sqrt (Re x))) ∨
        (Re x < 0 ∧ csqrt x = ii ∗ cor (sqrt (− (Re x))))
proof (cases x = 0)
  case True
  thus ?thesis
    by auto
next
  case False
  show ?thesis
  proof (cases Re x > 0)
    case True
    hence arg x = 0
      using ‹is-real x›
      by (metis arg-complex-of-real-positive complex-of-real-Re)
    thus ?thesis
      using ‹Re x > 0›
      unfolding csqrt-def
      by simp (metis Re.simps complex-of-real-def rcis-cmod-arg rcis-zero-arg)
  next
    case False
    hence Re x < 0
      using ‹x ≠ 0› ‹is-real x›
      by (cases x, auto)
    hence arg x = pi
      using ‹is-real x›
      by (metis arg-complex-of-real-negative complex-of-real-Re)
    thus ?thesis
      using ‹Re x < 0›
```

```
   unfolding csqrt-def
   by (simp add: rcis-def cis-def complex-of-real-def) (metis Complex-eq-0 False
Re.simps assms complex-minus-def complex-of-real-def cor-cmod-real le-less-linear
norm-le-zero-iff)
 qed
qed
```

```
lemma is-real-rot-to-xaxis:
 assumes z ≠ 0
 shows is-real (cis (−arg z) ∗ z)
proof (cases arg z = pi)
 case True
 thus ?thesis
  using im-complex-zero-arg1[of z]
  by auto
next
 case False
 hence ⌊− arg z⌋ = − arg z
  using canon-ang-eqI[of − arg z −arg z]
  using arg-bounded[of z]
  by (auto simp add: field-simps)
 hence arg (cis (− (arg z)) ∗ z) = 0
  using arg-mult[of cis (− (arg z)) z] ‹z ≠ 0›
  using arg-cis[of − arg z]
  by simp
 thus ?thesis
  using im-complex-zero-arg1[of cis (− arg z) ∗ z]
  by auto
qed
```

```
lemma cmod-1-plus-mult-le:
 cmod (1 + z∗w) ≤ sqrt((1 + (cmod z)^2) ∗ (1 + (cmod w)^2))
proof−
 have Re ((1+z∗w)∗(1+cnj z∗cnj w)) ≤ Re (1+z∗cnj z)∗ Re (1+w∗cnj w)
 proof−
  have Re ((w − cnj z)∗cnj(w − cnj z)) ≥ 0
   by (subst complex-mult-cnj-cmod) (simp add: power2-eq-square)
  hence Re (z∗w + cnj z ∗ cnj w) ≤ Re (w∗cnj w) + Re(z∗cnj z)
  by (simp only: complex-cnj complex-cnj-cnj field-simps complex-Re-diff complex-Re-add)
  thus ?thesis
   by (simp add: field-simps)
 qed
 hence (cmod (1 + z ∗ w))^2 ≤ (1 + (cmod z)^2) ∗ (1 + (cmod w)^2)
  by (subst cmod-square)+ simp
```

thus _?thesis_
by (_metis abs-norm-cancel real-sqrt-abs real-sqrt-le-iff_)
qed

lemma _cmod-diff-ge_: _cmod_ (_b_ − _c_) ≥ _sqrt_ (_1_ + (_cmod b_)^2) − _sqrt_ (_1_ + (_cmod c_)^2)
proof−
have (_cmod_ (_b_ − _c_))^2 + (_1/2_∗_Im_(_b_∗_cnj c_ − _c_∗_cnj b_))^2 ≥ _0_
by _simp_
hence (_cmod_ (_b_ − _c_))^2 ≥ − (_1/2_∗_Im_(_b_∗_cnj c_ − _c_∗_cnj b_))^2
by _simp_
hence (_cmod_ (_b_ − _c_))^2 ≥ (_1/2_∗_Re_(_b_∗_cnj c_ + _c_∗_cnj b_))^2 − _Re_(_b_∗_cnj b_∗_c_∗_cnj c_)
by (_auto simp add_: _power2-eq-square field-simps_)
hence _Re_ ((_b_ − _c_)∗(_cnj b_ − _cnj c_)) ≥ (_1/2_∗_Re_(_b_∗_cnj c_ + _c_∗_cnj b_))^2 − _Re_(_b_∗_cnj b_∗_c_∗_cnj c_)
by (_subst_ (_asm_) _cmod-square_) (_simp add_: _complex-cnj_)
moreover
have (_1_ + (_cmod b_)^2) ∗ (_1_ + (_cmod c_)^2) = _1_ + _Re_(_b_∗_cnj b_) + _Re_(_c_∗_cnj c_) + _Re_(_b_∗_cnj b_∗_c_∗_cnj c_)
by (_subst cmod-square_)+ (_simp add_: _field-simps power2-eq-square_)
moreover
have (_1_ + _Re_ (_scalprod b c_))^2 = _1_ + _2_∗_Re_(_scalprod b c_) + ((_Re_ (_scalprod b c_))^2)
by (_subst power2-sum_) _simp_
hence (_1_ + _Re_ (_scalprod b c_))^2 = _1_ + _Re_(_b_∗_cnj c_ + _c_∗_cnj b_) + (_1/2_ ∗ _Re_ (_b_∗_cnj c_ + _c_∗_cnj b_))^2
by _simp_
ultimately
have (_1_ + (_cmod b_)^2) ∗ (_1_ + (_cmod c_)^2) ≥ (_1_ + _Re_ (_scalprod b c_))^2
by (_simp add_: _field-simps_)
moreover
have _sqrt_((_1_ + (_cmod b_)^2) ∗ (_1_ + (_cmod c_)^2)) ≥ _0_
by (_metis one-power2 real-sqrt-sum-squares-mult-ge-zero_)
ultimately
have _sqrt_((_1_ + (_cmod b_)^2) ∗ (_1_ + (_cmod c_)^2)) ≥ _1_ + _Re_ (_scalprod b c_)
by (_metis power2-le-imp-le real-sqrt-ge-0-iff real-sqrt-pow2-iff_)
hence _Re_ ((_b_ − _c_) ∗ (_cnj b_ − _cnj c_)) ≥ _1_ + _Re_ (_c_∗_cnj c_) + _1_ + _Re_ (_b_∗_cnj b_) − _2_∗_sqrt_((_1_ + (_cmod b_)^2) ∗ (_1_ + (_cmod c_)^2))
by (_simp add_: _field-simps_)
hence ∗: (_cmod_ (_b_ − _c_))^2 ≥ (_sqrt_ (_1_ + (_cmod b_)^2) − _sqrt_ (_1_ + (_cmod c_)^2))^2
apply (_subst cmod-square_)+
apply (_subst_ (_asm_) _cmod-square_)+
apply (_subst power2-diff_)
apply (_subst real-sqrt-pow2_, _simp_)
apply (_subst real-sqrt-pow2_, _simp_)
apply (_simp add_: _real-sqrt-mult complex-cnj_)
done
thus _?thesis_

```
  proof (cases sqrt (1 + (cmod b)^2) - sqrt (1 + (cmod c)^2) > 0)
    case True
    thus ?thesis
      using square-cancel[OF *]
      by simp
  next
    case False
    hence 0 ≥ sqrt (1 + (cmod b)^2) - sqrt (1 + (cmod c)^2)
      by (metis less-eq-real-def linorder-neqE-linordered-idom)
    moreover
    have cmod (b - c) ≥ 0
      by simp
    ultimately
    show ?thesis
      by (metis add-increasing monoid-add-class.add.right-neutral)
  qed
qed

lemma cmod-diff-le: cmod (b - c) ≤ sqrt (1 + (cmod b)^2) + sqrt (1 + (cmod
c)^2)
proof -
  have (cmod (b + c))^2 + (1/2*Im(b*cnj c - c*cnj b))^2 ≥ 0
    by simp
  hence (cmod (b + c))^2 ≥ - (1/2*Im(b*cnj c - c*cnj b))^2
    by simp
  hence (cmod (b + c))^2 ≥ (1/2*Re(b*cnj c + c*cnj b))^2 - Re(b*cnj b*c*cnj
c)
    by (auto simp add: power2-eq-square field-simps)
  hence Re ((b + c)*(cnj b + cnj c)) ≥ (1/2*Re(b*cnj c + c*cnj b))^2 - Re(b*cnj
b*c*cnj c)
    by (subst (asm) cmod-square) (simp add: complex-cnj)
  moreover
  have (1 + (cmod b)^2) * (1 + (cmod c)^2) = 1 + Re(b*cnj b) + Re(c*cnj c) +
Re(b*cnj b*c*cnj c)
    by (subst cmod-square)+ (simp add: field-simps power2-eq-square)
  moreover
  have ++: 2*Re(scalprod b c) = Re(b*cnj c + c*cnj b)
    by simp
  have (1 - Re (scalprod b c))^2 = 1 - 2*Re(scalprod b c) + ((Re (scalprod b
c))^2)
    by (subst power2-diff) simp
  hence (1 - Re (scalprod b c))^2 = 1 - Re(b*cnj c + c*cnj b) + (1/2 * Re
(b*cnj c + c*cnj b))^2
    by (subst ++[symmetric]) simp
  ultimately
  have (1 + (cmod b)^2) * (1 + (cmod c)^2) ≥ (1 - Re (scalprod b c))^2
    by (simp add: field-simps)
  moreover
  have sqrt((1 + (cmod b)^2) * (1 + (cmod c)^2)) ≥ 0
```

```
    by (metis one-power2 real-sqrt-sum-squares-mult-ge-zero)
  ultimately
  have sqrt((1 + (cmod b)^2) ∗ (1 + (cmod c)^2)) ≥ 1 − Re (scalprod b c)
    by (metis power2-le-imp-le real-sqrt-ge-0-iff real-sqrt-pow2-iff)
  hence Re ((b − c) ∗ (cnj b − cnj c)) ≤ 1 + Re (c∗cnj c) + 1 + Re (b∗cnj b)
+ 2∗sqrt((1 + (cmod b)^2) ∗ (1 + (cmod c)^2))
    by (simp add: field-simps)
  hence ∗: (cmod (b − c))^2 ≤ (sqrt (1 + (cmod b)^2) + sqrt (1 + (cmod c)^2))^2
    apply (subst cmod-square)+
    apply (subst (asm) cmod-square)+
    apply (subst power2-sum)
    apply (subst real-sqrt-pow2, simp)
    apply (subst real-sqrt-pow2, simp)
    apply (simp add: real-sqrt-mult complex-cnj)
    done
  thus ?thesis
    using square-cancel[OF ∗]
    by simp
qed



definition cdist where
  [simp]: cdist z1 z2 ≡ cmod (z2 − z1)



lemma [simp]:
  fixes z1 z2 :: complex
  assumes z1 ≠ 0 z2 ≠ 0
  shows ∃ k. k ≠ 0 ∧ z2 = k ∗ z1
using assms
by (rule-tac x=z2/z1 in exI) simp

lemma [simp]:
  fixes z::complex
  assumes z ≠ 0
  shows ∃ k. k ≠ 0 ∧ k ∗ z = 1
using assms
by (rule-tac x=1/z in exI) simp

lemma [simp]:
  fixes z::complex
  shows ∃ k. k ≠ 0 ∧ k ∗ z = z
by (rule-tac x=1 in exI) simp

end
```

2 Systems of linear equations

```
theory LinearSystems
imports MoreComplex
begin

definition det2 where
  [simp]: det2 a11 a12 a21 a22 ≡ a11*a22 − a12*a21

lemma regular-homogenous-system:
  fixes a11::complex
  assumes a11*a22 − a12*a21 ≠ 0 a11*x1 + a12*x2 = 0 a21*x1 + a22*x2 =
0
  shows x1 = 0 ∧ x2 = 0
proof (cases a11 = 0)
  case True
  with assms(1) have a12 ≠ 0 a21 ≠ 0
    by auto
  thus ?thesis
    using ‹a11 = 0› assms(2) assms(3)
    by auto
next
  case False
  hence x1 = − a12*x2 / a11
    using assms(2)
   by (auto simp add: field-simps) (metis diff-divide-eq-iff diff-minus-eq-add divide-zero-left
eq-iff-diff-eq-0 minus-divide-left)
  hence (a11*a22 − a12*a21)*x2 = 0
    using assms(3) ‹a11 ≠ 0›
    by (auto simp add: field-simps)
  thus ?thesis
    using assms(1) assms(2) ‹a11 ≠ 0›
    by auto
qed

lemma regular-system:
  fixes a11::complex
  assumes a11*a22 − a12*a21 ≠ 0
  shows ∃! x.
       a11*(fst x) + a12*(snd x) = b1 ∧
       a21*(fst x) + a22*(snd x) = b2
proof
  let ?d = a11*a22 − a12*a21 and ?d1 = b1*a22 − b2*a12 and ?d2 = b2*a11
− b1*a21
  let ?x = (?d1 / ?d, ?d2 / ?d)
  have a11 * ?d1 + a12 * ?d2 = b1*?d a21 * ?d1 + a22 * ?d2 = b2*?d
    by (auto simp add: field-simps)
  thus a11 * fst ?x + a12 * snd ?x = b1 ∧ a21 * fst ?x + a22 * snd ?x = b2
    using assms
    by (metis (hide-lams, no-types) add-divide-distrib eq-divide-imp fst-eqD snd-eqD
times-divide-eq-right)

  fix x'
  assume a11 ∗ fst x' + a12 ∗ snd x' = b1 ∧ a21 ∗ fst x' + a22 ∗ snd x' = b2
  with ‹a11 ∗ fst ?x + a12 ∗ snd ?x = b1 ∧ a21 ∗ fst ?x + a22 ∗ snd ?x = b2›
  have a11 ∗ (fst x' − fst ?x) + a12 ∗ (snd x' − snd ?x) = 0 ∧ a21 ∗ (fst x' −
fst ?x) + a22 ∗ (snd x' − snd ?x) = 0
    by (auto simp add: field-simps)
  thus x' = ?x
    using regular-homogenous-system[OF assms, of fst x' − fst ?x snd x' − snd ?x]
    by (cases x') auto
qed

lemma singular-system:
  fixes a11::complex
  assumes a11∗a22 − a12∗a21 = 0 a11 ≠ 0 ∨ a12 ≠ 0
  assumes ∗: a11∗fst x0 + a12∗snd x0 = b1 a21∗fst x0 + a22∗snd x0 = b2
  assumes ∗∗: a11∗fst x + a12∗snd x = b1
  shows a21∗fst x + a22∗snd x = b2
proof (cases a11 = 0)
  case True
  with assms have a21 = 0 a12 ≠ 0
    by auto
  let ?k = a22 / a12
  have b2 = ?k ∗ b1
    using ∗ ‹a11 = 0› ‹a21 = 0› ‹a12 ≠ 0›
    by auto
  thus ?thesis
    using ‹a11 = 0› ‹a21 = 0› ‹a12 ≠ 0› ∗∗
    by auto
next
  case False
  let ?k = a21 / a11
  from ∗∗
  have ?k ∗ a11 ∗ fst x + ?k ∗ a12 ∗ snd x = ?k ∗ b1
    using ‹a11 ≠ 0›
    by (auto simp add: field-simps)
  moreover
  have a21 = ?k ∗ a11 a22 = ?k ∗ a12 b2 = ?k ∗ b1
    using assms(1) ∗ ‹a11 ≠ 0›
    by (auto simp add: field-simps)
  ultimately
  show ?thesis
    by auto
qed

lemma cnj-equation:
  assumes a∗z1 + b∗z2 = c
  shows cnj a ∗ cnj z1 + cnj b ∗ cnj z2 = cnj c
using assms
by (auto simp add: complex-cnj-mult complex-cnj-add)

lemma regular-cnj-system:
  assumes det2 a1 (cnj a1) a2 (cnj a2) ≠ 0 is-real b1 is-real b2
  shows ∃! μ. a1 ∗ cnj μ + cnj a1 ∗ μ = b1 ∧
             a2 ∗ cnj μ + cnj a2 ∗ μ = b2
proof−
  have ∃! x. a1 ∗ fst x + cnj a1 ∗ snd x = b1 ∧
            a2 ∗ fst x + cnj a2 ∗ snd x = b2
    using regular-system assms(1)
    by simp

  then obtain x where
    ∗: a1 ∗ fst x + cnj a1 ∗ snd x = b1
       a2 ∗ fst x + cnj a2 ∗ snd x = b2
    and ∗∗:
    ∀ x′. a1 ∗ fst x′ + cnj a1 ∗ snd x′ = b1 ∧
          a2 ∗ fst x′ + cnj a2 ∗ snd x′ = b2 ⟶
           x′ = x
    unfolding Ex1-def
    by blast
  have cnj b1 = b1 cnj b2 = b2
    using ‹is-real b1› ‹is-real b2›
    by (case-tac[!] b1, case-tac[!] b2) auto
  hence a1 ∗ cnj (snd x) + cnj a1 ∗ cnj (fst x) = b1
        a2 ∗ cnj (snd x) + cnj a2 ∗ cnj (fst x) = b2
    using cnj-equation[OF ∗(1)] cnj-equation[OF ∗(2)] ‹is-real b1› ‹is-real b2›
    by (auto simp add: field-simps)
  hence (cnj (snd x), cnj (fst x)) = x
    using ∗∗
    by auto
  hence fst x = cnj (snd x)
    by (cases x) auto
  thus ?thesis
    using ∗ ∗∗
    unfolding Ex1-def
    by (rule-tac x=snd x in exI, auto) (metis prod.inject)
qed

end
```

3 Quadratic equations

```
theory Quadratic
imports Complex MoreComplex
begin
```

lemma *real-quadratic-equation*:
fixes ξ :: *real*
assumes $\xi^2 + b * \xi + c = 0$ $b^2 - 4{*}c \geq 0$
shows $\xi = (-b + sqrt(b^2 - 4{*}c)) \mathbin{/} 2 \vee \xi = (-b - sqrt(b^2 - 4{*}c)) \mathbin{/} 2$
using *assms*
proof−
from *assms* **have** $(2 * (\xi + b/2))^2 = b^2 - 4{*}c$
by (*simp add*: *power2-eq-square field-simps*)
hence $2 * (\xi + b/2) = sqrt\ (b^2 - 4{*}c) \vee 2 * (\xi + b/2) = -\ sqrt\ (b^2 - 4{*}c)$
by (*metis abs-minus-cancel power2-abs power2-eq-iff real-sqrt-abs*)
thus *?thesis*
by (*auto simp add*: *field-simps*)
qed

lemma *real-quadratic-equation′*:
fixes ξ :: *real*
assumes $b^2 - 4{*}c \geq 0$ $\xi = (-b + sqrt(b^2 - 4{*}c)) \mathbin{/} 2 \vee \xi = (-b - sqrt(b^2 - 4{*}c)) \mathbin{/} 2$
shows $\xi^2 + b * \xi + c = 0$
using *assms*(*2*)
proof
assume $*$: $\xi = (-\ b + sqrt\ (b^2 - 4 * c)) \mathbin{/} 2$
show *?thesis*
using *assms*(*1*)
by ((*subst* $*$)+, *subst power-divide*, *subst power2-sum*, *simp add*: *field-simps*, *simp add*: *power2-eq-square*)
next
assume $*$: $\xi = (-\ b - sqrt\ (b^2 - 4 * c)) \mathbin{/} 2$
show *?thesis*
using *assms*(*1*)
by ((*subst* $*$)+, *subst power-divide*, *subst power2-diff*, *simp add*: *field-simps*, *simp add*: *power2-eq-square*)
qed

lemma *complex-quadratic-equation*:
fixes ξ :: *complex*
assumes $\xi^2 + b * \xi + c = 0$
shows $\xi = (-b + csqrt(b^2 - 4{*}c)) \mathbin{/} 2 \vee \xi = (-b - csqrt(b^2 - 4{*}c)) \mathbin{/} 2$
using *assms*
proof−
from *assms* **have** $(2 * (\xi + b/2))^2 = b^2 - 4{*}c$
by (*simp add*: *power2-eq-square field-simps*)
(*metis ab-semigroup-mult-class.mult-ac*(*1*) *comm-semiring-1-class.normalizing-semiring-rules*(*34*) *comm-semiring-class.distrib mult-zero-left*)
hence $2 * (\xi + b/2) = csqrt\ (b^2 - 4{*}c) \vee 2 * (\xi + b/2) = -\ csqrt\ (b^2 - 4{*}c)$
using *csqrt*[*of* $(2 * (\xi + b \mathbin{/} 2))$ $b^2 - 4 * c$]
by (*simp add*: *power2-eq-square*)
thus *?thesis*

using *mult-cancel-right*[*of* $b + \xi * 2\ 2\ csqrt\ (b^2 - 4*c)$]
using *mult-cancel-right*[*of* $b + \xi * 2\ 2\ -csqrt\ (b^2 - 4*c)$]
by (*auto simp add*: *field-simps*) (*metis add-diff-cancel diff-minus-eq-add minus-diff-eq*)
qed

lemma *complex-quadratic-equation'*:
fixes ξ :: *complex*
assumes $\xi = (-b + csqrt(b^2 - 4*c)) / 2 \vee$
$\xi = (-b - csqrt(b^2 - 4*c)) / 2$
shows $\xi^2 + b * \xi + c = 0$
using *assms*
proof
assume $*$: $\xi = (- b + csqrt\ (b^2 - 4 * c)) / 2$
show *?thesis*
by ((*subst* $*$)+) (*subst power-divide*, *subst power2-sum*, *simp add*: *field-simps*, *simp add*: *power2-eq-square*)
next
assume $*$: $\xi = (- b - csqrt\ (b^2 - 4 * c)) / 2$
show *?thesis*
by ((*subst* $*$)+, *subst power-divide*, *subst power2-diff*, *simp add*: *field-simps*, *simp add*: *power2-eq-square*)
qed

lemma *complex-quadratic-equation-full*:
fixes ξ :: *complex*
assumes $a*\xi^2 + b * \xi + c = 0\ a \neq 0$
shows $\xi = (-b + csqrt(b^2 - 4*a*c)) / (2*a) \vee$
$\xi = (-b - csqrt(b^2 - 4*a*c)) / (2*a)$
proof−
from *assms* **have** $\xi^2 + (b/a) * \xi + (c/a) = 0$
by (*simp add*: *field-simps*)
hence $\xi = (-(b/a) + csqrt((b/a)^2 - 4*(c/a))) / 2 \vee \xi = (-(b/a) - csqrt((b/a)^2 - 4*(c/a))) / 2$
using *complex-quadratic-equation*[*of* ξ *b*/*a* *c*/*a*]
by *simp*
hence $\exists\ k.\ \xi = (-(b/a) + (-1)\hat{}k * csqrt((b/a)^2 - 4*(c/a))) / 2$
by *safe* (*rule-tac* $x=2$ **in** *exI*, *simp*, *rule-tac* $x=1$ **in** *exI*, *simp*)
then obtain *k1* **where** $\xi = (-(b/a) + (-1)\hat{}k1 * csqrt((b/a)^2 - 4*(c/a))) / 2$
by *auto*
moreover
have $(b / a)^2 - 4 * (c / a) = (b^2 - 4 * a * c) * (1 / a^2)$
by (*simp add*: *field-simps power2-eq-square*)
hence $csqrt\ ((b / a)^2 - 4 * (c / a)) = csqrt\ (b^2 - 4 * a * c) * csqrt\ (1/a^2) \vee$
$csqrt\ ((b / a)^2 - 4 * (c / a)) = -\ csqrt\ (b^2 - 4 * a * c) * csqrt\ (1/a^2)$
using *csqrt-mult*[*of* $b^2 - 4 * a * c\ 1/a^2$]
by *auto*
hence $\exists\ k.\ csqrt\ ((b / a)^2 - 4 * (c / a)) = (-1)\hat{}k * csqrt\ (b^2 - 4 * a * c) * csqrt\ (1 / a^2)$

```
    by safe (rule-tac x=2 in exI, simp, rule-tac x=1 in exI, simp)
  then obtain k2 where csqrt ((b / a)^2 − 4 ∗ (c / a)) = (−1)ˆk2 ∗ csqrt (b^2
− 4 ∗ a ∗ c) ∗ csqrt (1 / a^2)
    by auto
  moreover
  have csqrt (1 / a^2) = 1/a ∨ csqrt (1 / a^2) = −1/a
    using csqrt[of 1/a 1 / a^2]
    by (auto simp add: power2-eq-square)
  hence ∃ k. csqrt (1 / a^2) = (−1)ˆk ∗ 1/a
    by safe (rule-tac x=2 in exI, simp, rule-tac x=1 in exI, simp)
  then obtain k3 where csqrt (1 / a^2) = (−1)ˆk3 ∗ 1/a
    by auto
  ultimately
  have ξ = (− (b / a) + ((−1) ˆ k1 ∗ (−1) ˆ k2 ∗ (−1) ˆ k3) ∗ csqrt (b^2 − 4 ∗
a ∗ c) ∗ 1/a) / 2
    by simp
  moreover
  have (−(1::complex)) ˆ k1 ∗ (−1) ˆ k2 ∗ (−1) ˆ k3 = 1 ∨ (−(1::complex)) ˆ
k1 ∗ (−1) ˆ k2 ∗ (−1) ˆ k3 = −1
    using neg-one-even-power[of k1 + k2 + k3]
    using neg-one-odd-power[of k1 + k2 + k3]
    by (simp add: comm-semiring-1-class.normalizing-semiring-rules(26))
       (cases even (k1 + k2 + k3), auto)
  ultimately
  have ξ = (− (b / a) + csqrt (b^2 − 4 ∗ a ∗ c) ∗ 1 / a) / 2 ∨ ξ = (− (b / a) −
csqrt (b^2 − 4 ∗ a ∗ c) ∗ 1 / a) / 2
    by auto
  thus ?thesis
    using ‹a ≠ 0›
    by (simp add: field-simps)
qed

lemma complex-quadratic-two-solutions:
  fixes b c :: complex
  assumes b^2 − 4∗c ≠ 0
  shows ∃ k_1 k_2. k_1 ≠ k_2 ∧ k_1^2 + b∗k_1 + c = 0 ∧ k_2^2 + b∗k_2 + c = 0
proof−
  let ?ξ1 = (−b + csqrt(b^2 − 4∗c)) / 2
  let ?ξ2 = (−b − csqrt(b^2 − 4∗c)) / 2
  show ?thesis
    apply (rule-tac x=?ξ1 in exI)
    apply (rule-tac x=?ξ2 in exI)
   using assms complex-quadratic-equation'[of ?ξ1 b c] complex-quadratic-equation'[of
?ξ2 b c]
    by simp
qed

end
```

4 Vectors, Matrices

theory *Matrices*
imports *MoreComplex LinearSystems Quadratic*
begin

4.1 Vectors

Type of complex vector

type-synonym *complex-vec* = *complex* × *complex*

definition *vec-zero* :: *complex-vec* **where**
[*simp*]: *vec-zero* = (*0*, *0*)

Vector scalar multiplication

fun *mult-sv* :: *complex* ⇒ *complex-vec* ⇒ *complex-vec* (**infixl** $*_{sv}$ *100*) **where**
$k *_{sv} (x, y) = (k{*}x, k{*}y)$

lemma *fst-mult-sv* [*simp*]: $fst\ (k *_{sv} v) = k * fst\ v$
by (*cases v*) *simp*

lemma *snd-mult-sv* [*simp*]: $snd\ (k *_{sv} v) = k * snd\ v$
by (*cases v*) *simp*

lemma *mult-sv-mult-sv* [*simp*]: $k1 *_{sv} (k2 *_{sv} v) = (k1{*}k2) *_{sv} v$
by (*cases v*) *simp*

lemma *one-mult-sv* [*simp*]: $1 *_{sv} v = v$
by (*cases v*) *simp*

Multiplication of two vectors

fun *mult-vv* :: *complex* × *complex* ⇒ *complex* × *complex* ⇒ *complex* (**infixl** $*_{vv}$ *100*) **where**
$(x, y) *_{vv} (a, b) = x{*}a + y{*}b$

lemma *mult-vv-commute*: $v1 *_{vv} v2 = v2 *_{vv} v1$
by (*cases v1*, *cases v2*) *auto*

lemma *mult-vv-scale-sv1*:
$(k *_{sv} v1) *_{vv} v2 = k * (v1 *_{vv} v2)$
by (*cases v1*, *cases v2*) (*auto simp add*: *field-simps*)

lemma *mult-vv-scale-sv2*:
$v1 *_{vv} (k *_{sv} v2) = k * (v1 *_{vv} v2)$
by (*cases v1*, *cases v2*) (*auto simp add*: *field-simps*)

Conjugate vector

fun *vec-map* **where**

vec-map f (*x*, *y*) = (*f x*, *f y*)

definition *vec-cnj* **where** *vec-cnj* = *vec-map cnj*

lemma *vec-cnj-vec-cnj* [*simp*]: *vec-cnj* (*vec-cnj v*) = *v*
by (*cases v*) (*simp add*: *vec-cnj-def*)

lemma *cnj-mult-vv*: *cnj* (*v1* $*_{vv}$ *v2*) = (*vec-cnj v1*) $*_{vv}$ (*vec-cnj v2*)
by (*cases v1*, *cases v2*) (*simp add*: *vec-cnj-def complex-cnj*)

lemma *vec-cnj-sv* [*simp*]: *vec-cnj* (*k* $*_{sv}$ *A*) = *cnj k* $*_{sv}$ *vec-cnj A*
by (*cases A*) (*auto simp add*: *vec-cnj-def complex-cnj*)

lemma *scalsquare-vv-zero*:
(*vec-cnj v*) $*_{vv}$ *v* = *0* ⟷ *v* = *vec-zero*
apply (*cases v*)
apply (*auto simp add*: *vec-cnj-def field-simps complex-mult-cnj-cmod*)
apply (*smt norm-eq-zero of-real-add of-real-eq-0-iff of-real-power sum-power2-eq-zero-iff*)+
done

4.2 Matrices

Type of complex matrices

type-synonym *complex-mat* = *complex* × *complex* × *complex* × *complex*

Matrix scalar multiplication

fun *mult-sm* :: *complex* ⇒ *complex-mat* ⇒ *complex-mat* (**infixl** $*_{sm}$ *100*) **where**
k $*_{sm}$ (*a*, *b*, *c*, *d*) = (*k***a*, *k***b*, *k***c*, *k***d*)

lemma [*simp*]: *k1* $*_{sm}$ (*k2* $*_{sm}$ *A*) = (*k1***k2*) $*_{sm}$ *A*
by (*cases A*) *auto*

lemma [*simp*]: *1* $*_{sm}$ *A* = *A*
by (*cases A*) *auto*

lemma *mult-sm-inv-l*:
assumes *k* ≠ *0 k* $*_{sm}$ *A* = *B*
shows *A* = (*1*/*k*) $*_{sm}$ *B*
using *assms*
by *auto*

Matrix addition and subtraction

definition *mat-zero* :: *complex-mat* **where** [*simp*]: *mat-zero* = (*0*, *0*, *0*, *0*)

fun *mat-plus* :: *complex-mat* ⇒ *complex-mat* ⇒ *complex-mat* (**infixl** $+_{mm}$ *100*)
where
mat-plus (*a1*, *b1*, *c1*, *d1*) (*a2*, *b2*, *c2*, *d2*) = (*a1*+*a2*, *b1*+*b2*, *c1*+*c2*, *d1*+*d2*)

fun *mat-minus* :: *complex-mat* ⇒ *complex-mat* ⇒ *complex-mat* (**infixl** $-_{mm}$ *100*)
where
mat-minus (*a1*, *b1*, *c1*, *d1*) (*a2*, *b2*, *c2*, *d2*) = (*a1*−*a2*, *b1*−*b2*, *c1*−*c2*, *d1*−*d2*)

fun *mat-uminus* :: *complex-mat* ⇒ *complex-mat* **where**
mat-uminus (*a*, *b*, *c*, *d*) = (−*a*, −*b*, −*c*, −*d*)

lemma *nonzero-mult-real*:
assumes *A* ≠ *mat-zero* *k* ≠ *0*
shows *k* $*_{sm}$ *A* ≠ *mat-zero*
using *assms*
by (*cases A*) *simp*

Matrix multiplication

fun *mult-mm* :: *complex-mat* ⇒ *complex-mat* ⇒ *complex-mat* (**infixl** $*_{mm}$ *100*)
where
(*a1*, *b1*, *c1*, *d1*) $*_{mm}$ (*a2*, *b2*, *c2*, *d2*) =
(*a1*∗*a2* + *b1*∗*c2*, *a1*∗*b2* + *b1*∗*d2*, *c1*∗*a2*+*d1*∗*c2*, *c1*∗*b2*+*d1*∗*d2*)

lemma *mult-mm-assoc*: *A* $*_{mm}$ (*B* $*_{mm}$ *C*) = (*A* $*_{mm}$ *B*) $*_{mm}$ *C*
by (*cases A*, *cases B*, *cases C*) (*auto simp add*: *field-simps*)

lemma *mult-assoc-5*: *A* $*_{mm}$ (*B* $*_{mm}$ *C* $*_{mm}$ *D*) $*_{mm}$ *E* = (*A* $*_{mm}$ *B*) $*_{mm}$ *C* $*_{mm}$ (*D* $*_{mm}$ *E*)
by (*simp only*: *mult-mm-assoc*)

lemma *mat-zero-r* [*simp*]: *A* $*_{mm}$ *mat-zero* = *mat-zero*
by (*cases A*) *simp*

lemma *mat-zero-l* [*simp*]: *mat-zero* $*_{mm}$ *A* = *mat-zero*
by (*cases A*) *simp*

definition *eye* :: *complex-mat* **where**
[*simp*]: *eye* = (*1*, *0*, *0*, *1*)

lemma *mat-eye-l*:
eye $*_{mm}$ *A* = *A*
by (*cases A*) *auto*

lemma *mat-eye-r*:
A $*_{mm}$ *eye* = *A*
by (*cases A*) *auto*

lemma *mult-mm-sm* [*simp*]: *A* $*_{mm}$ (*k* $*_{sm}$ *B*) = *k* $*_{sm}$ (*A* $*_{mm}$ *B*)
by (*cases A*, *cases B*) (*simp add*: *field-simps*)

lemma *mult-sm-mm* [*simp*]: $(k *_{sm} A) *_{mm} B = k *_{sm} (A *_{mm} B)$
  by (*cases A*, *cases B*) (*simp add*: *field-simps*)

lemma *mult-sm-eye-mm* [*simp*]: $k *_{sm} eye *_{mm} A = k *_{sm} A$
by (*cases A*) *simp*

Matrix determinant

fun *mat-det* **where** *mat-det* $(a, b, c, d) = a*d - b*c$

lemma *mat-det-mult* [*simp*]: *mat-det* $(A *_{mm} B)$ = *mat-det* $A *$ *mat-det* B
by (*cases A*, *cases B*) (*auto simp add*: *field-simps*)

lemma *mat-det-mult-sm* [*simp*]: *mat-det* $(k *_{sm} A) = (k*k) *$ *mat-det* A
by (*cases A*) (*auto simp add*: *field-simps*)

Matrix inverse

fun *mat-inv* :: *complex-mat* ⇒ *complex-mat* **where**
  mat-inv $(a, b, c, d) = (1/(a*d - b*c)) *_{sm} (d, -b, -c, a)$

lemma *mat-inv-r*:
  assumes *mat-det* $A \neq 0$
  shows $A *_{mm}$ (*mat-inv* A) = *eye*
using *assms*
by (*cases A*, *auto simp add*: *field-simps*) *algebra*

lemma *mat-inv-l*:
  assumes *mat-det* $A \neq 0$
  shows (*mat-inv* A) $*_{mm} A$ = *eye*
using *assms*
by (*cases A*, *auto simp add*: *field-simps*) *algebra*

lemma *mat-det-inv*:
  assumes *mat-det* $A \neq 0$
  shows *mat-det* (*mat-inv* A) = 1 / *mat-det* A
proof−
  have *mat-det eye* = *mat-det* $A *$ *mat-det* (*mat-inv* A)
    using *mat-inv-l*[*OF assms*, *symmetric*]
    by *simp*
  thus *?thesis*
    using *assms*
    by (*simp add*: *field-simps*)
qed

lemma *mult-mm-inv-l*:
  assumes *mat-det* $A \neq 0$ $A *_{mm} B = C$
  shows B = *mat-inv* $A *_{mm} C$
using *assms mat-eye-l*[*of B*]
by (*auto simp add*: *mult-mm-assoc mat-inv-l*)

lemma *mult-mm-inv-r*:
  assumes *mat-det B* ≠ *0 A* $*_{mm}$ *B* = *C*
  shows *A* = *C* $*_{mm}$ *mat-inv B*
using *assms mat-eye-r*[*of A*]
by (*auto simp add*: *mult-mm-assoc*[*symmetric*] *mat-inv-r*)

lemma *mult-mm-non-zero-l*:
  assumes *mat-det A* ≠ *0 B* ≠ *mat-zero*
  shows *A* $*_{mm}$ *B* ≠ *mat-zero*
using *assms mat-zero-r*
using *mult-mm-inv-l*[*OF assms*(*1*), *of B mat-zero*]
by *auto*

lemma *mat-inv-mult-mm*:
  assumes *mat-det A* ≠ *0 mat-det B* ≠ *0*
  shows *mat-inv* (*A* $*_{mm}$ *B*) = *mat-inv B* $*_{mm}$ *mat-inv A*
using *assms*
proof−
  have (*A* $*_{mm}$ *B*) $*_{mm}$ (*mat-inv B* $*_{mm}$ *mat-inv A*) = *eye*
    using *assms*
    by (*metis mat-inv-r mult-mm-assoc mult-mm-inv-r*)
  thus *?thesis*
    using *mult-mm-inv-l*[*of A* $*_{mm}$ *B mat-inv B* $*_{mm}$ *mat-inv A eye*] *assms mat-eye-r*
    by *simp*
qed

lemma *mult-mm-cancel-l*:
  assumes *mat-det M* ≠ *0* *M* $*_{mm}$ *A* = *M* $*_{mm}$ *B*
  shows *A* = *B*
using *assms*
by (*metis mult-mm-inv-l*)

lemma *mult-mm-cancel-r*:
  assumes *mat-det M* ≠ *0* *A* $*_{mm}$ *M* = *B* $*_{mm}$ *M*
  shows *A* = *B*
using *assms*
by (*metis mult-mm-inv-r*)

lemma *mult-mm-non-zero-r*:
  assumes *A* ≠ *mat-zero mat-det B* ≠ *0*
  shows *A* $*_{mm}$ *B* ≠ *mat-zero*
using *assms mat-zero-l*
using *mult-mm-inv-r*[*OF assms*(*2*), *of A mat-zero*]
by *auto*

lemma *mat-inv-mult-sm*:
  assumes *k* ≠ *0*

shows *mat-inv* $(k *_{sm} A) = (1 / k) *_{sm}$ *mat-inv A*
proof−
obtain *a b c d* **where** *A* = (*a*, *b*, *c*, *d*)
by (*cases A*) *auto*
thus *?thesis*
using *assms*
by *auto* (*subst mult-assoc*[*of k a k*∗*d*], *subst mult-assoc*[*of k b k*∗*c*], *subst right-diff-distrib*[*of k a*∗(*k*∗*d*) *b*∗(*k*∗*c*), *symmetric*], *simp*, *simp add*: *field-simps*)+
qed

lemma *mat-inv-inv* [*simp*]:
assumes *mat-det M* ≠ *0*
shows *mat-inv* (*mat-inv M*) = *M*
proof−
have *mat-inv* $M *_{mm} M$ = *eye*
using *mat-inv-l*[*OF assms*]
by *simp*
thus *?thesis*
using *assms mat-det-inv*[*of M*]
using *mult-mm-inv-l*[*of mat-inv M M eye*] *mat-eye-r*
by (*auto simp del*: *eye-def*)
qed

Matrix transpose

fun *mat-transpose* **where** *mat-transpose* (*a*, *b*, *c*, *d*) = (*a*, *c*, *b*, *d*)

lemma [*simp*]: *mat-transpose* (*mat-transpose A*) = *A*
by (*cases A*) *auto*

lemma [*simp*]: *mat-transpose* $(k *_{sm} A) = k *_{sm}$ (*mat-transpose A*)
by (*cases A*) *simp*

lemma [*simp*]: *mat-transpose* $(A *_{mm} B)$ = *mat-transpose* $B *_{mm}$ *mat-transpose A*
by (*cases A*, *cases B*) *auto*

lemma *mat-inv-transpose*: *mat-transpose* (*mat-inv M*) = *mat-inv* (*mat-transpose M*)
by (*cases M*) *auto*

lemma *mat-det-transpose*:
fixes *M* :: *complex-mat*
shows [*simp*]: *mat-det* (*mat-transpose M*) = *mat-det M*
by (*cases M*) *auto*

Diagonal matrices

fun *mat-diagonal* **where**
mat-diagonal (*A*, *B*, *C*, *D*) = (*B* = *0* ∧ *C* = *0*)

Matrix conjugate

fun *mat-map* **where**
mat-map f (*a*, *b*, *c*, *d*) = (*f a*, *f b*, *f c*, *f d*)

definition *mat-cnj* **where** *mat-cnj* = *mat-map cnj*

lemma [*simp*]: *mat-cnj* (*mat-cnj A*) = *A*
unfolding *mat-cnj-def*
by (*cases A*) *auto*

lemma *mat-cnj-sm* [*simp*]: *mat-cnj* (*k* $*_{sm}$ *A*) = *cnj k* $*_{sm}$ (*mat-cnj A*)
by (*cases A*) (*simp add*: *mat-cnj-def complex-cnj*)

lemma *mat-det-cnj* [*simp*]: *mat-det* (*mat-cnj A*) = *cnj* (*mat-det A*)
by (*cases A*) (*simp add*: *mat-cnj-def complex-cnj*)

lemma *nonzero-mat-cnj*: *mat-cnj A* = *mat-zero* ⟷ *A* = *mat-zero*
by (*cases A*) (*auto simp add*: *mat-cnj-def*)

lemma *mat-inv-cnj*: *mat-cnj* (*mat-inv M*) = *mat-inv* (*mat-cnj M*)
unfolding *mat-cnj-def*
by (*cases M*) (*auto simp add*: *complex-cnj*)

Matrix adjoint (conjugate

definition *mat-adj* **where** *mat-adj A* = *mat-cnj* (*mat-transpose A*)

lemma *mat-adj-mult-mm* [*simp*]: *mat-adj* (*A* $*_{mm}$ *B*) = *mat-adj B* $*_{mm}$ *mat-adj A*
by (*cases A*, *cases B*) (*auto simp add*: *mat-adj-def mat-cnj-def complex-cnj*)

lemma *mat-adj-mult-sm* [*simp*]: *mat-adj* (*k* $*_{sm}$ *A*) = *cnj k* $*_{sm}$ *mat-adj A*
  by (*cases A*) (*auto simp add*: *mat-adj-def mat-cnj-def complex-cnj*)

lemma *mat-det-adj*: *mat-det* (*mat-adj A*) = *cnj* (*mat-det A*)
by (*cases A*) (*auto simp add*: *mat-adj-def mat-cnj-def complex-cnj*)

lemma *mat-adj-inv*:
  assumes *mat-det M* ≠ *0*
  shows *mat-adj* (*mat-inv M*) = *mat-inv* (*mat-adj M*)
  by (*cases M*) (*auto simp add*: *mat-adj-def mat-cnj-def complex-cnj*)

lemma *mat-transpose-mat-cnj*: *mat-transpose* (*mat-cnj A*) = *mat-adj A*
by (*cases A*) (*auto simp add*: *mat-adj-def mat-cnj-def*)

lemma [*simp*]: *mat-adj* (*mat-adj A*) = *A*
unfolding *mat-adj-def*
by (*subst mat-transpose-mat-cnj*) (*simp add*: *mat-adj-def*)

Matrix trace

fun *mat-trace* **where**

mat-trace $(a, b, c, d) = a + d$

Multiplication of matrix and a vector

fun *mult-mv* :: *complex-mat* ⇒ *complex-vec* ⇒ *complex-vec* (**infixl** $*_{mv}$ *100*) **where**
$(a, b, c, d) *_{mv} (x, y) = (x*a + y*b, x*c + y*d)$

fun *mult-vm* :: *complex-vec* ⇒ *complex-mat* ⇒ *complex-vec* (**infixl** $*_{vm}$ *100*) **where**
$(x, y) *_{vm} (a, b, c, d) = (x*a + y*c, x*b + y*d)$

lemma *eye-mv-l* [*simp*]: *eye* $*_{mv}$ $v = v$
by (*cases v*) *simp*

lemma *mult-mv-mv* [*simp*]: $B *_{mv} (A *_{mv} v) = (B *_{mm} A) *_{mv} v$
by (*cases v*, *cases A*, *cases B*) (*auto simp add*: *field-simps*)

lemma *mult-vm-vm* [*simp*]: $(v *_{vm} A) *_{vm} B = v *_{vm} (A *_{mm} B)$
by (*cases v*, *cases A*, *cases B*) (*auto simp add*: *field-simps*)

lemma *mult-mv-inv*:
assumes $x = A *_{mv} y$ *mat-det* $A \neq 0$
shows $y =$ (*mat-inv* A) $*_{mv}$ x
using *assms*
by (*cases y*) (*simp add*: *mat-inv-l*)

lemma *mult-vm-inv*:
assumes $x = y *_{vm} A$ *mat-det* $A \neq 0$
shows $y = x *_{vm}$ (*mat-inv* A)
using *assms*
by (*cases y*) (*simp add*: *mat-inv-r*)

lemma *mult-mv-cancel-l*:
assumes *mat-det* $A \neq 0$ $A *_{mv} v = A *_{mv} v'$
shows $v = v'$
using *assms*
using *mult-mv-inv*
by *blast*

lemma *mult-vm-cancel-r*:
assumes *mat-det* $A \neq 0$ $v *_{vm} A = v' *_{vm} A$
shows $v = v'$
using *assms*
using *mult-vm-inv*
by *blast*

lemma *vec-zero-l* [*simp*]:
$A *_{mv}$ *vec-zero* = *vec-zero*
by (*cases A*) *simp*

lemma *vec-zero-r* [*simp*]:

vec-zero $*_{vm}$ *A* = *vec-zero*
by (*cases A*) *simp*

lemma *mult-mv-nonzero*:
assumes *v* ≠ *vec-zero mat-det A* ≠ *0*
shows *A* $*_{mv}$ *v* ≠ *vec-zero*
apply (*rule ccontr*)
using *assms mult-mv-inv*[*of vec-zero A v*] *mat-inv-l vec-zero-l*
by *auto*

lemma *mult-vm-nonzero*:
assumes *v* ≠ *vec-zero mat-det A* ≠ *0*
shows *v* $*_{vm}$ *A* ≠ *vec-zero*
apply (*rule ccontr*)
using *assms mult-vm-inv*[*of vec-zero v A*] *mat-inv-r vec-zero-r*
by *auto*

lemma *mult-sv-mv*: *k* $*_{sv}$ (*A* $*_{mv}$ *v*) = (*A* $*_{mv}$ (*k* $*_{sv}$ *v*))
by (*cases A*, *cases v*) (*simp add*: *field-simps*)

lemma *mult-mv-mult-vm*: *A* $*_{mv}$ *x* = *x* $*_{vm}$ (*mat-transpose A*)
by (*cases A*, *cases x*) *auto*

lemma
mult-mv-vv: *A* $*_{mv}$ *v1* $*_{vv}$ *v2* = *v1* $*_{vv}$ (*mat-transpose A* $*_{mv}$ *v2*)
by (*cases v1*, *cases v2*, *cases A*) (*auto simp add*: *field-simps*)

lemma *mult-vv-mv*: *x* $*_{vv}$ (*A* $*_{mv}$ *y*) = (*x* $*_{vm}$ *A*) $*_{vv}$ *y*
by (*cases x*, *cases y*, *cases A*) (*auto simp add*: *field-simps*)

lemma *vec-cnj-mult-mv*:
shows *vec-cnj* (*A* $*_{mv}$ *x*) = (*mat-cnj A*) $*_{mv}$ (*vec-cnj x*)
using *assms*
by (*cases A*, *cases x*) (*auto simp add*: *vec-cnj-def mat-cnj-def complex-cnj*)

lemma *vec-cnj-mult-vm*: *vec-cnj* (*v* $*_{vm}$ *A*) = *vec-cnj v* $*_{vm}$ *mat-cnj A*
unfolding *vec-cnj-def mat-cnj-def*
by (*cases A*, *cases v*, *auto simp add*: *complex-cnj*)

4.3 Eigenvalues and eigenvectors

definition *eigenpair* **where**
[*simp*]: *eigenpair k v H* ⟷ *v* ≠ *vec-zero* ∧ *H* $*_{mv}$ *v* = *k* $*_{sv}$ *v*

definition *eigenval* **where**
[*simp*]: *eigenval k H* ⟷ (∃ *v*. *v* ≠ *vec-zero* ∧ *H* $*_{mv}$ *v* = *k* $*_{sv}$ *v*)

lemma *eigen-equation*:
shows *eigenval k H* ⟷ k^2 − *mat-trace H* ∗ *k* + *mat-det H* = *0* (**is** *?lhs* ⟷

```
?rhs)
proof−
  obtain A B C D where HH: H = (A, B, C, D)
    by (cases H) auto
  show ?thesis
  proof
    assume ?lhs
    then obtain v where v ≠ vec-zero H *mv v = k *sv v
      unfolding eigenval-def
      by blast
    obtain v1 v2 where vv: v = (v1, v2)
      by (cases v) auto
    from ‹H *mv v = k *sv v› have (H −mm (k *sm eye)) *mv v = vec-zero
      using HH vv
      by (auto simp add: field-simps)
    hence mat-det (H −mm (k *sm eye)) = 0
      using ‹v ≠ vec-zero› vv HH
      using regular-homogenous-system[of A − k D − k B C v1 v2]
      by (auto simp add: field-simps)
    thus ?rhs
      using HH
      by (auto simp add: power2-eq-square field-simps)
  next
    assume ?rhs
    hence *: mat-det (H −mm (k *sm eye)) = 0
      using HH
      by (auto simp add: field-simps power2-eq-square)
    show ?lhs
    proof (cases H −mm (k *sm eye) = mat-zero)
      case True
      thus ?thesis
        using HH
        by (auto) (rule-tac x=1 in exI, simp)
    next
      case False
      hence (A − k ≠ 0 ∨ B ≠ 0) ∨ (D − k ≠ 0 ∨ C ≠ 0)
        using HH
        by auto
      thus ?thesis
      proof
        assume A − k ≠ 0 ∨ B ≠ 0
        hence C * B + (D − k) * (k − A) = 0
          using * singular-system[of A−k D−k B C (0, 0) 0 0 (B, k−A)] HH
          by (auto simp add: field-simps)
          hence (B, k−A) ≠ vec-zero (H −mm (k *sm eye)) *mv (B, k−A) = vec-zero
          using HH ‹A − k ≠ 0 ∨ B ≠ 0›
          by (auto simp add: field-simps)
         then obtain v where v ≠ vec-zero ∧ (H −mm (k *sm eye)) *mv v =
```

vec-zero
by *blast*
thus *?thesis*
using *HH*
unfolding *eigenval-def*
by (*rule-tac* $x{=}v$ **in** *exI*) (*case-tac v*, *simp add*: *field-simps*)
next
assume $D - k \neq 0 \vee C \neq 0$
hence $C * B + (D - k) * (k - A) = 0$
using $*$ *singular-system*[*of* $D{-}k$ $A{-}k$ C B $(0, 0)$ 0 0 $(C, k{-}D)$] *HH*
by (*auto simp add*: *field-simps*)
hence $(k{-}D, C) \neq$ *vec-zero* $(H -_{mm} (k *_{sm}$ *eye*$)) *_{mv} (k{-}D, C) =$ *vec-zero*
using *HH* ‹$D - k \neq 0 \vee C \neq 0$›
by (*auto simp add*: *field-simps*)
then obtain v **where** $v \neq$ *vec-zero* $\wedge$ $(H -_{mm} (k *_{sm}$ *eye*$)) *_{mv} v =$ *vec-zero*
by *blast*
thus *?thesis*
using *HH*
unfolding *eigenval-def*
by (*rule-tac* $x{=}v$ **in** *exI*) (*case-tac v*, *simp add*: *field-simps*)
qed
qed
qed
qed

4.4 Bilinear and Quadratic forms; Congruence

Bilinear forms

definition *bilinear-form* **where**
[*simp*]: *bilinear-form v1 v2 H* = (*vec-cnj v1*) $*_{vm}$ *H* $*_{vv}$ *v2*

lemma *bilinear-form-scale-m*:
shows *bilinear-form v1 v2* ($k *_{sm} H$) = $k *$ *bilinear-form v1 v2 H*
by (*cases v1*, *cases v2*, *cases H*) (*simp add*: *vec-cnj-def complex-cnj field-simps*)

lemma *bilinear-form-scale-v1*:
shows *bilinear-form* ($k *_{sv}$ *v1*) *v2 H* = *cnj k* $*$ *bilinear-form v1 v2 H*
by (*cases v1*, *cases v2*, *cases H*) (*simp add*: *vec-cnj-def complex-cnj field-simps*)

lemma *bilinear-form-scale-v2*:
shows *bilinear-form v1* ($k *_{sv}$ *v2*) *H* = $k *$ *bilinear-form v1 v2 H*
by (*cases v1*, *cases v2*, *cases H*) (*simp add*: *vec-cnj-def complex-cnj field-simps*)

Quadratic forms

definition *quad-form* **where**
[*simp*]: *quad-form v H* = (*vec-cnj v*) $*_{vm}$ *H* $*_{vv}$ *v*

lemma *quad-form v H = bilinear-form v v H*
by *simp*

lemma *quad-form-scale-v*:
  shows *quad-form* $(k *_{sv} v)$ *H = cor* ((*cmod k*)^2) $*$ *quad-form v H*
using *bilinear-form-scale-v1 bilinear-form-scale-v2*
by (*simp add*: *complex-mult-cnj-cmod field-simps*)

lemma *quad-form-scale-m*:
  shows *quad-form v* $(k *_{sm} H)$ $= k *$ *quad-form v H*
using *bilinear-form-scale-m*
by *simp*

lemma *cnj-quad-form* [*simp*]: *cnj* (*quad-form z H*) = *quad-form z* (*mat-adj H*)
by (*cases H*, *cases z*) (*auto simp add*: *mat-adj-def mat-cnj-def vec-cnj-def complex-cnj field-simps*)

Matrix congruence

abbreviation *congruence* **where**
  congruence M H ≡ *mat-adj M* $*_{mm}$ *H* $*_{mm}$ *M*

lemma *bilinear-form-congruence*:
  assumes *mat-det M* ≠ *0*
  shows *bilinear-form v1 v2 H* = *bilinear-form* $(M *_{mv} v1)$ $(M *_{mv} v2)$ (*congruence* (*mat-inv M*) *H*)
proof−
  have *mat-det* (*mat-adj M*) ≠ *0*
    using *assms*
    by (*simp add*: *mat-det-adj*)
  show *?thesis*
    unfolding *bilinear-form-def*
    apply (*subst mult-mv-mult-vm*)
    apply (*subst vec-cnj-mult-vm*)
    apply (*subst mat-adj-def*[*symmetric*])
    apply (*subst mult-vm-vm*)
    apply (*subst mult-vv-mv*)
    apply (*subst mult-vm-vm*)
    apply (*subst mat-adj-inv*[*OF* ‹*mat-det M* ≠ *0*›])
    apply (*subst mult-assoc-5*)
    apply (*subst mat-inv-r*[*OF* ‹*mat-det* (*mat-adj M*) ≠ *0*›])
    apply (*subst mat-inv-l*[*OF* ‹*mat-det M* ≠ *0*›])
    apply (*subst mat-eye-l*, *subst mat-eye-r*)
    by *simp*
qed

lemma *quad-form-congruence*:
  assumes *mat-det M* ≠ *0*
  shows *quad-form* $(M *_{mv} z)$ (*congruence* (*mat-inv M*) *H*) = *quad-form z H*
using *bilinear-form-congruence*[*OF assms*]

by *simp*

lemma *congruence-nonzero*:
  assumes *H ≠ mat-zero mat-det M ≠ 0*
  shows *congruence M H ≠ mat-zero*
using *assms*
by (*subst mult-mm-non-zero-r*, *subst mult-mm-non-zero-l*) (*auto simp add*: *mat-det-adj*)

lemma *congruence-congruence*:
  shows *congruence M1* (*congruence M2 A*) = *congruence* (*M2* $*_{mm}$ *M1*) *A*
apply (*subst mult-mm-assoc*)
apply (*subst mult-mm-assoc*)
apply (*subst mat-adj-mult-mm*)
apply (*subst mult-mm-assoc*)
by *simp*

lemma [*simp*]: *congruence eye A* = *A*
  by (*cases A*) (*simp add*: *mat-adj-def mat-cnj-def*)

lemma *congruence-congruence-inv*:
  assumes *mat-det M ≠ 0*
  shows *congruence M* (*congruence* (*mat-inv M*) *A*) = *A*
using *assms congruence-congruence*[*of M mat-inv M A*]
using *mat-inv-l*[*of M*] *mat-eye-l*
by (*simp del*: *eye-def*)

lemma *congruence-inv*:
  assumes *mat-det M ≠ 0 congruence M A = B*
  shows *congruence* (*mat-inv M*) *B* = *A*
  using *assms*
  using ‹*mat-det M ≠ 0*› *mult-mm-inv-l*[*of mat-adj M A* $*_{mm}$ *M B*]
  using *mult-mm-inv-r*[*of M A mat-inv* (*mat-adj M*) $*_{mm}$ *B*]
  by (*simp add*: *mat-det-adj mult-mm-assoc mat-adj-inv*)

lemma *congruence-scale-m*:
  shows *congruence A* (*k* $*_{sm}$ *B*) = *k* $*_{sm}$ (*congruence A B*)
by (*cases A*, *cases B*) (*auto simp add*: *mat-adj-def mat-cnj-def field-simps*)

lemma *inj-congruence*:
  assumes *mat-det M ≠ 0 congruence M H = congruence M H'*
  shows *H = H'*
proof−
  have *H* $*_{mm}$ *M* = *H'* $*_{mm}$ *M*
    using *assms*
    using *mult-mm-cancel-l*[*of mat-adj M H* $*_{mm}$ *M H'* $*_{mm}$ *M*]
    by (*simp add*: *mat-det-adj mult-mm-assoc*)
  thus *?thesis*
    using *assms*
    using *mult-mm-cancel-r*[*of M H H'*]

```
    by simp
qed

definition similarity where similarity I M = mat-inv I ∗mm M ∗mm I

lemma
  mat-det-similarity:
  assumes mat-det I ≠ 0
  shows mat-det (similarity I M) = mat-det M
using assms
unfolding similarity-def
by (simp add: mat-det-inv)

lemma mat-trace-similarity:
  assumes mat-det I ≠ 0
  shows mat-trace (similarity I M) = mat-trace M
proof−
  obtain a b c d where II: I = (a, b, c, d)
    by (cases I) auto
  obtain A B C D where MM: M = (A, B, C, D)
    by (cases M) auto
  have A ∗ (a ∗ d) / (a ∗ d − b ∗ c) + D ∗ (a ∗ d) / (a ∗ d − b ∗ c) =
        A + D + A ∗ (b ∗ c) / (a ∗ d − b ∗ c) + D ∗ (b ∗ c) / (a ∗ d − b ∗ c)
    using assms II
    by (simp add: field-simps)
  thus ?thesis
    using II MM
    by (simp add: field-simps similarity-def)
qed

end
```

5 Unitary matrices

```
theory UnitaryMatrices
imports Matrices
begin
```

```
definition unitary where
  unitary M ⟷ mat-adj M ∗mm M = eye

definition unitary-gen where
  unitary-gen M ⟷ (∃ k::complex. k ≠ 0 ∧ mat-adj M ∗mm M = k ∗sm eye)

lemma uniary-gen-scale [simp]:
```

assumes *unitary-gen M k* $\neq$ *0*
shows *unitary-gen* (*k* $*_{sm}$ *M*)
using *assms*
unfolding *unitary-gen-def*
by *auto*

lemma *unitary-unitary-gen* [*simp*]: *unitary M* $\Longrightarrow$ *unitary-gen M*
unfolding *unitary-gen-def unitary-def*
by *auto*

lemma *unitary-gen-real*:
assumes *unitary-gen M*
shows ($\exists$ *k*::*real*. *k* > *0* $\wedge$ *mat-adj M* $*_{mm}$ *M* = *cor k* $*_{sm}$ *eye*)
proof−
obtain *k* **where** ∗: *mat-adj M* $*_{mm}$ *M* = *k* $*_{sm}$ *eye k* $\neq$ *0*
using *assms*
by (*auto simp add*: *unitary-gen-def*)
obtain *a b c d* **where** *M* = (*a*, *b*, *c*, *d*)
by (*cases M*) *auto*
hence *k* = *cor* ((*cmod a*)^2) + *cor* ((*cmod c*)^2)
using ∗
by (*subst complex-mult-cnj-cmod*[*symmetric*])+ (*auto simp add*: *mat-adj-def mat-cnj-def*)
hence *is-real k Re k* > *0*
using ⟨*k* $\neq$ *0*⟩
by (*auto simp add*: *power2-eq-square*) (*metis comm-semiring-1-class.normalizing-semiring-rules*(*6*) *mult-eq-0-iff of-real-eq-0-iff sum-squares-gt-zero-iff*)
thus *?thesis*
using ∗
by (*rule-tac x*=*Re k* **in** *exI*) (*simp add*: *complex-of-real-Re*)
qed

lemma *unitary-gen-regular*:
assumes *unitary-gen M*
shows *mat-det M* $\neq$ *0*
proof−
from *assms* **obtain** *k* **where**
k $\neq$ *0 mat-adj M* $*_{mm}$ *M* = *k* $*_{sm}$ *eye*
unfolding *unitary-gen-def*
by *auto*
hence *mat-det* (*mat-adj M* $*_{mm}$ *M*) $\neq$ *0*
by *simp*
thus *?thesis*
by (*simp add*: *mat-det-adj*)
qed

lemmas *unitary-regular* = *unitary-gen-regular*[*OF unitary-unitary-gen*]

lemma
unitary-gen M $\longleftrightarrow$ ($\exists$ *k::complex. k* $\neq$ *0* $\wedge$ *mat-adj M* $*_{mm}$ (*1, 0, 0, 1*) $*_{mm}$ *M* = *k* $*_{sm}$ (*1, 0, 0, 1*))
unfolding *unitary-gen-def*
using *mat-eye-r*
by (*auto simp add*: *mult-assoc*)

lemma *unitary-comp*:
assumes *unitary M1 unitary M2*
shows *unitary* (*M1* $*_{mm}$ *M2*)
using *assms*
unfolding *unitary-def*
by (*simp del*: *eye-def*) (*metis mat-eye-r mult-mm-assoc*)

lemma *unitary-gen-comp*:
assumes *unitary-gen M1 unitary-gen M2*
shows *unitary-gen* (*M1* $*_{mm}$ *M2*)
proof−
obtain *k1 k2* **where** ∗: *k1* ∗ *k2* $\neq$ *0 mat-adj M1* $*_{mm}$ *M1* = *k1* $*_{sm}$ *eye mat-adj M2* $*_{mm}$ *M2* = *k2* $*_{sm}$ *eye*
using *assms*
unfolding *unitary-gen-def*
by *auto*
have *mat-adj M2* $*_{mm}$ *mat-adj M1* $*_{mm}$ (*M1* $*_{mm}$ *M2*) = *mat-adj M2* $*_{mm}$ (*mat-adj M1* $*_{mm}$ *M1*) $*_{mm}$ *M2*
by (*auto simp add*: *mult-mm-assoc*)
also have ... = *mat-adj M2* $*_{mm}$ ((*k1* $*_{sm}$ *eye*) $*_{mm}$ *M2*)
using ∗
by (*auto simp add*: *mult-mm-assoc*)
also have ... = *mat-adj M2* $*_{mm}$ (*k1* $*_{sm}$ *M2*)
using *mult-sm-eye-mm*[*of k1 M2*]
by (*simp del*: *eye-def*)
also have ... = *k1* $*_{sm}$ (*k2* $*_{sm}$ *eye*)
using ∗
by *auto*
finally
show *?thesis*
using ∗
unfolding *unitary-gen-def*
by (*rule-tac x=k1∗k2* **in** *exI*, *simp del*: *eye-def*)
qed

lemma *unitary-adj-eq-inv*:
unitary M $\longleftrightarrow$ *mat-det M* $\neq$ *0* $\wedge$ *mat-adj M* = *mat-inv M*
using *unitary-regular*[*of M*] *mult-mm-inv-r*[*of M mat-adj M eye*] *mat-eye-l*[*of mat-inv M*] *mat-inv-l*[*of M*]
unfolding *unitary-def*
by − (*rule*, *simp-all*)

lemma *unitary-inv*:
assumes *unitary M*
shows *unitary* (*mat-inv M*)
using *assms*
unfolding *unitary-adj-eq-inv*
using *mat-adj-inv*[*of M*] *mat-det-inv*[*of M*]
by *simp*

lemma *unitary-gen-unitary*:
shows *unitary-gen M* $\longleftrightarrow$ ($\exists$ *k M'*. $k > 0$ $\wedge$ *unitary M'* $\wedge$ M = (*cor k* $*_{sm}$ *eye*) $*_{mm}$ *M'*) (**is** *?lhs = ?rhs*)
proof
assume *?lhs*
then obtain *k* **where** $*$: $k>0$ *mat-adj M* $*_{mm}$ *M* = *cor k* $*_{sm}$ *eye*
using *unitary-gen-real*[*of M*]
by *auto*

let *?k'* = *cor* (*sqrt k*)
have *?k'* $*$ *cnj ?k'* = *cor k*
using ‹$k > 0$›
by *simp*
moreover
have *Re ?k'* > 0 *is-real ?k' ?k'* $\neq 0$
using ‹$k > 0$›
by *auto*
ultimately
show *?rhs*
using $*$ *mat-eye-l*
unfolding *unitary-gen-def unitary-def*
by (*rule-tac x=Re ?k'* **in** *exI*) (*rule-tac x*=(1/*?k'*)$*_{sm}$*M* **in** *exI*, *simp add*: *complex-cnj mult-sm-mm*[*symmetric*])
next
assume *?rhs*
then obtain *k M'* **where** $k > 0$ *unitary M' M* = (*cor k* $*_{sm}$ *eye*) $*_{mm}$ *M'*
by *blast*
hence *M* = *cor k* $*_{sm}$ *M'*
using *mult-sm-mm*[*of cor k eye M'*] *mat-eye-l*
by *simp*
thus *?lhs*
using ‹*unitary M'*› ‹$k > 0$›
by (*simp add*: *unitary-gen-def unitary-def*)
qed

lemma *unitary-gen-inv*:
assumes *unitary-gen M*
shows *unitary-gen* (*mat-inv M*)
proof−
obtain *k M'* **where** $0 < k$ *unitary M' M* = *cor k* $*_{sm}$ *eye* $*_{mm}$ *M'*

```
      using unitary-gen-unitary[of M] assms
      by blast
    hence mat-inv M = cor (1/k) *_sm mat-inv M'
         by (metis mat-inv-mult-sm mult-sm-eye-mm norm-not-less-zero of-real-1
of-real-divide of-real-eq-0-iff sgn-1-neg sgn-greater sgn-if sgn-pos sgn-sgn)
    thus ?thesis
      using ‹k > 0› ‹unitary M'›
     by (subst unitary-gen-unitary[of mat-inv M]) (rule-tac x=1/k in exI, rule-tac
x=mat-inv M' in exI, metis divide-pos-pos mult-sm-eye-mm unitary-inv zero-less-one)

qed

lemma unitary-special:
  assumes unitary M mat-det M = 1
  shows ∃ a b. M = (a, b, −cnj b, cnj a)
proof−
  have mat-adj M = mat-inv M
    using assms mult-mm-inv-r[of M mat-adj M eye] mat-eye-r mat-eye-l
    by (simp add: unitary-def)
  thus ?thesis
    using ‹mat-det M = 1›
    by (cases M) (auto simp add: mat-adj-def mat-cnj-def)
qed

lemma unitary-gen-special:
  assumes unitary-gen M mat-det M = 1
  shows ∃ a b. M = (a, b, −cnj b, cnj a)
proof−
  from assms
  obtain k where ∗: k ≠ 0 mat-adj M *_mm M = k *_sm eye
    unfolding unitary-gen-def
    by auto
  hence mat-det (mat-adj M *_mm M) = k∗k
    by simp
  hence k∗k = 1
    using assms(2)
    by (simp add: mat-det-adj)
  hence k = 1 ∨ k = −1
    using square-eq-1-iff[of k]
    by simp
  moreover
  have mat-adj M = k *_sm mat-inv M
    using ∗
    using assms mult-mm-inv-r[of M mat-adj M k *_sm eye] mat-eye-r mat-eye-l
    by simp (metis mult-sm-eye-mm ∗(2))
  moreover
  obtain a b c d where M = (a, b, c, d)
    by (cases M) auto
  ultimately
```

have $M = (a,\ b,\ -cnj\ b,\ cnj\ a) \vee M = (a,\ b,\ cnj\ b,\ -cnj\ a)$
using *assms*(*2*)
by (*auto simp add*: *mat-adj-def mat-cnj-def*)
moreover
have $Re\ (-\ (cor\ (cmod\ a))^2 - (cor\ (cmod\ b))^2) < 1$
by (*auto simp add*: *power2-eq-square*) (*smt add-increasing2 add-nonneg-nonneg is-num-normalize*(*8*) *less-le minus-add-distrib neg-le-0-iff-le norm-ge-zero norm-mult not-one-le-zero real-0-le-add-iff zero-le-one*)
hence $-\ (cor\ (cmod\ a))^2 - (cor\ (cmod\ b))^2 \neq 1$
by *force*
hence $M \neq (a,\ b,\ cnj\ b,\ -cnj\ a)$
using ‹*mat-det M* = *1*› *complex-mult-cnj-cmod*[*of a*] *complex-mult-cnj-cmod*[*of b*]
by *auto*
ultimately
show *?thesis*
by *auto*
qed

lemma *unitary-gen-iff*:
shows *unitary-gen M* $\longleftrightarrow$ ($\exists$ *a b k* . $k \neq 0 \wedge$ *mat-det* $(a,\ b,\ -cnj\ b,\ cnj\ a) \neq 0 \wedge (M = k *_{sm} (a,\ b,\ -cnj\ b,\ cnj\ a))$) (**is** *?lhs* = *?rhs*)
proof
assume *?lhs*
obtain *d* **where** ∗: $d*d$ = *mat-det M*
using *ex-complex-sqrt*
by *auto*
hence $d \neq 0$
using *unitary-gen-regular*[*OF* ‹*unitary-gen M*›]
by *auto*
from ‹*unitary-gen M*›
obtain *k* **where** $k \neq 0$ *mat-adj M* $*_{mm} M = k *_{sm}$ *eye*
unfolding *unitary-gen-def*
by *auto*
hence *mat-adj* $((1/d)*_{sm}M) *_{mm} ((1/d)*_{sm}M) = (k\ /\ (d*cnj\ d)) *_{sm}$ *eye*
by (*simp add*: *complex-cnj*)
obtain *a b* **where** $(a,\ b,\ -\ cnj\ b,\ cnj\ a) = (1\ /\ d) *_{sm} M$
using *unitary-gen-special*[*of* $(1\ /\ d) *_{sm} M$] ‹*unitary-gen M*› ∗ *unitary-gen-regular*[*of M*] ‹$d \neq 0$›
by *force*
moreover
hence *mat-det* $(a,\ b,\ -\ cnj\ b,\ cnj\ a) \neq 0$
using *unitary-gen-regular*[*OF* ‹*unitary-gen M*›] ‹$d \neq 0$›
by *auto*
ultimately
show *?rhs*
apply (*rule-tac x*=*a* **in** *exI*, *rule-tac x*=*b* **in** *exI*, *rule-tac x*=*d* **in** *exI*)
using *mult-sm-inv-l*[*of 1*/*d M*]
by (*auto simp add*: *field-simps*)

```
next
  assume ?rhs
  then obtain a b k where k ≠ 0 ∧ mat-det (a, b, − cnj b, cnj a) ≠ 0 ∧ M =
k *sm (a, b, − cnj b, cnj a)
    by auto
  thus ?lhs
    unfolding unitary-gen-def
    apply (auto simp add: mat-adj-def mat-cnj-def complex-cnj)
    using mult-eq-0-iff [of cnj k * k cnj a * a + cnj b * b]
    by (auto simp add: field-simps)
qed

lemma unitary-iff:
  shows unitary M ⟷
    (∃ a b k . (cmod a)^2 + (cmod b)^2 ≠ 0 ∧ (cmod k)^2 = 1 / ((cmod a)^2 + (cmod
b)^2) ∧ M = k *sm (a, b, −cnj b, cnj a)) (is ?lhs = ?rhs)
proof
  assume ?lhs
  obtain k a b where *: M = k *sm (a, b, −cnj b, cnj a) k ≠ 0 mat-det (a, b,
−cnj b, cnj a) ≠ 0
    using unitary-gen-iff  unitary-unitary-gen[OF ‹unitary M›]
    by auto

  have md: mat-det (a, b, −cnj b, cnj a) = cor ((cmod a)^2 + (cmod b)^2)
    by (auto simp add: complex-mult-cnj-cmod)

  have k * cnj k * mat-det (a, b, −cnj b, cnj a) = 1
    using ‹unitary M› *
    unfolding unitary-def
    by (auto simp add: mat-adj-def mat-cnj-def complex-cnj field-simps)
  hence (cmod k)^2 * ((cmod a)^2 + (cmod b)^2) = 1
   by (subst (asm) complex-mult-cnj-cmod, subst (asm) md, subst (asm) cor-mult[symmetric])
(metis of-real-1 of-real-eq-iff)
  thus ?rhs
    using * mat-eye-l
    apply (rule-tac x=a in exI, rule-tac x=b in exI, rule-tac x=k in exI)
    apply (auto simp add: complex-mult-cnj-cmod)
   by (metis ‹(cmod k)^2 * ((cmod a)^2 + (cmod b)^2) = 1› mult-eq-0-iff nonzero-eq-divide-eq
zero-neq-one)
next
  assume ?rhs
   then obtain a b k where  *: (cmod a)^2 + (cmod b)^2 ≠ 0 (cmod k)^2 = 1 /
((cmod a)^2 + (cmod b)^2) M = k *sm (a, b, −cnj b, cnj a)
    by auto
  have (k * cnj k) * (a * cnj a) + (k * cnj k) * (b * cnj b) = 1
    apply (subst complex-mult-cnj-cmod)+
    using *(1−2)
    apply (auto simp add: field-simps)
    apply (metis cor-add cor-mult of-real-1 of-real-power)+
    done
  thus ?lhs
    using ∗
    unfolding unitary-def
    by (simp add: mat-adj-def mat-cnj-def complex-cnj field-simps)
qed
```

```
definition unitary11 where
  unitary11 M ←→ mat-adj M ∗_mm (1, 0, 0, −1) ∗_mm M = (1, 0, 0, −1)

definition unitary11-gen where
  unitary11-gen M ←→ (∃ k. k ≠ 0 ∧ mat-adj M ∗_mm (1, 0, 0, −1) ∗_mm M =
k ∗_sm (1, 0, 0, −1))

lemma unitary11-gen-real:
  unitary11-gen M ←→ (∃ k. k ≠ 0 ∧ mat-adj M ∗_mm (1, 0, 0, −1) ∗_mm M =
cor k ∗_sm (1, 0, 0, −1))
unfolding unitary11-gen-def
proof auto
  fix k
  assume k ≠ 0 congruence M (1, 0, 0, −1) = (k, 0, 0, − k)
  hence mat-det (congruence M (1, 0, 0, −1)) = −k∗k
    by simp
  moreover
  have is-real (mat-det (congruence M (1, 0, 0, −1))) Re (mat-det (congruence
M (1, 0, 0, −1))) ≤ 0
    by (auto simp add: mat-det-adj) (smt real-minus-mult-self-le)
  ultimately
  have is-real (k∗k) Re (−k∗k) ≤ 0
    by auto
  hence is-real k
    using ‹k ≠ 0›
    by auto (smt not-real-square-gt-zero)
  thus ∃ka. ka ≠ 0 ∧ k = cor ka
    using ‹k ≠ 0›
    by (rule-tac x=Re k in exI) (cases k, auto simp add: complex-of-real-Re)
qed

lemma unitary11-unitary11-gen [simp]: unitary11 M =⇒ unitary11-gen M
unfolding unitary11-gen-def unitary11-def
by (rule-tac x=1 in exI, auto)


lemma unitary11-gen-regular:
  assumes unitary11-gen M
  shows mat-det M ≠ 0
proof−
  from assms obtain k where
    k ≠ 0 mat-adj M *mm (1, 0, 0, −1) *mm M = cor k *sm (1, 0, 0, −1)
    unfolding unitary11-gen-real
    by auto
  hence mat-det (mat-adj M *mm (1, 0, 0, −1) *mm M) ≠ 0
    by simp
  thus ?thesis
    by (simp add: mat-det-adj)
qed

lemmas unitary11-regular = unitary11-gen-regular[OF unitary11-unitary11-gen]

lemma unitary11-gen-mult-sm:
  assumes k ≠ 0 unitary11-gen M
  shows unitary11-gen (k *sm M)
proof−
  have k * cnj k = cor (Re (k * cnj k))
    by (subst complex-of-real-Re) auto
  thus ?thesis
    using assms
    unfolding unitary11-gen-real
    by auto (rule-tac x=Re (k*cnj k) * ka in exI, auto)
qed

lemma unitary11-gen-div-sm:
  assumes k ≠ 0 unitary11-gen (k *sm M)
  shows unitary11-gen M
using assms unitary11-gen-mult-sm[of 1/k k *sm M]
by simp

lemma unitary11-special:
  assumes unitary11 M mat-det M = 1
  shows ∃ a b. M = (a, b, cnj b, cnj a)
proof−
  have mat-adj M *mm (1, 0, 0, −1) = (1, 0, 0, −1) *mm mat-inv M
    using assms mult-mm-inv-r
    by (simp add: unitary11-def)
  thus ?thesis
    using assms(2)
    by (cases M) (simp add: mat-adj-def mat-cnj-def)
qed

lemma unitary11-gen-special:
  assumes unitary11-gen M mat-det M = 1
  shows ∃ a b. M = (a, b, cnj b, cnj a) ∨ M = (a, b, −cnj b, −cnj a)
proof−
  from assms
  obtain k where *: k ≠ 0 mat-adj M *mm (1, 0, 0, −1) *mm M = cor k *sm (1, 0, 0, −1)
    unfolding unitary11-gen-real
    by auto
  hence mat-det (mat-adj M *mm (1, 0, 0, −1) *mm M) = − cor k* cor k
    by simp
  hence mat-det (mat-adj M *mm M) = cor k* cor k
    by simp
  hence cor k* cor k = 1
    using assms(2)
    by (simp add: mat-det-adj)
  hence cor k = 1 ∨ cor k = −1
    using square-eq-1-iff [of cor k]
    by simp
  moreover
  have mat-adj M *mm (1, 0, 0, −1) = (cor k *sm (1, 0, 0, −1)) *mm mat-inv M
    using *
    using assms mult-mm-inv-r mat-eye-r mat-eye-l
    by auto
  moreover
  obtain a b c d where M = (a, b, c, d)
    by (cases M) auto
  ultimately
  have M = (a, b, cnj b, cnj a) ∨ M = (a, b, −cnj b, −cnj a)
    using assms(2)
    by (auto simp add: mat-adj-def mat-cnj-def)
  thus ?thesis
    by auto
qed

lemma unitary11-gen-iff':
  shows unitary11-gen M ⟷
  (∃ a b k . k ≠ 0 ∧ mat-det (a, b, cnj b, cnj a) ≠ 0 ∧
            (M = k *sm (a, b, cnj b, cnj a) ∨ M = k *sm (−1, 0, 0, 1) *mm (a, b, cnj b, cnj a))) (is ?lhs = ?rhs)
proof
  assume ?lhs
  obtain d where *: d*d = mat-det M
    using ex-complex-sqrt
    by auto
  hence d ≠ 0
    using unitary11-gen-regular[OF ‹unitary11-gen M›]
    by auto
  from ‹unitary11-gen M›
  obtain k where k ≠ 0 mat-adj M *mm (1, 0, 0, −1) *mm M = cor k *sm (1, 0, 0, −1)
    unfolding unitary11-gen-real
    by auto
  hence mat-adj ((1/d)*smM)*mm (1, 0, 0, −1) *mm ((1/d)*smM) = (cor k / (d*cnj d)) *sm (1, 0, 0, −1)
    by (simp add: complex-cnj)
  moreover
  have is-real (cor k / (d * cnj d))
    by (metis complex-In-mult-cnj-zero div-reals is-real-complex-of-real)
  hence cor (Re (cor k / (d * cnj d))) = cor k / (d * cnj d)
    by (simp add: complex-of-real-Re)
  ultimately
  have unitary11-gen ((1/d)*smM)
    unfolding unitary11-gen-real
    using ‹d ≠ 0› ‹k ≠ 0›
    by (rule-tac x=Re (cor k / (d * cnj d)) in exI, auto)
  moreover
  have mat-det ((1 / d) *sm M) = 1
    using * unitary11-gen-regular[of M] ‹unitary11-gen M›
    by auto
  ultimately
  obtain a b where (a, b, cnj b, cnj a) = (1 / d) *sm M ∨ (a, b, −cnj b, −cnj a) = (1 / d) *sm M
    using unitary11-gen-special[of (1 / d) *sm M]
    by force
  thus ?rhs
  proof
    assume (a, b, cnj b, cnj a) = (1 / d) *sm M
    moreover
    hence mat-det (a, b, cnj b, cnj a) ≠ 0
      using unitary11-gen-regular[OF ‹unitary11-gen M›] ‹d ≠ 0›
      by auto
    ultimately
    show ?rhs
      using ‹d ≠ 0›
      by (rule-tac x=a in exI, rule-tac x=b in exI, rule-tac x=d in exI, simp)
  next
    assume *: (a, b, −cnj b, −cnj a) = (1 / d) *sm M
    hence (1 / d) *sm M = (a, b, −cnj b, −cnj a)
      by simp
    hence M = (a * d, b * d, − (d * cnj b), − (d * cnj a))
      using ‹d ≠ 0›
      using mult-sm-inv-l[of 1/d M (a, b, −cnj b, −cnj a), symmetric]
      by (simp add: field-simps)
    moreover
    have mat-det (a, b, −cnj b, −cnj a) ≠ 0
      using * unitary11-gen-regular[OF ‹unitary11-gen M›] ‹d ≠ 0›
      by auto
    ultimately
    show ?thesis
      using ‹d ≠ 0›
      by (rule-tac x=a in exI, rule-tac x=b in exI, rule-tac x=−d in exI) (simp
add: field-simps)
  qed
next
  assume ?rhs
  then obtain a b k where k ≠ 0 mat-det (a, b, cnj b, cnj a) ≠ 0
    M = k *_sm (a, b, cnj b, cnj a) ∨ M = k *_sm (−1, 0, 0, 1) *_mm (a, b, cnj b, cnj a)
    by auto
  moreover
  let ?x = cnj k * cnj a * (k * a) + − (cnj k * b * (k * cnj b))
  have ?x = (k*cnj k)*(a*cnj a − b*cnj b)
    by (auto simp add: field-simps)
  hence is-real ?x
    by simp
  hence cor (Re ?x) = ?x
    by (rule complex-of-real-Re)
  moreover
  have ?x ≠ 0
    using mult-eq-0-iff[of cnj k * k (cnj a * a + − cnj b * b)]
    using ⟨mat-det (a, b, cnj b, cnj a) ≠ 0⟩ ⟨k ≠ 0⟩
    by (auto simp add: field-simps)
  hence Re ?x ≠ 0
    using ⟨is-real ?x⟩
    by (cases ?x) simp
  ultimately
  show ?lhs
    unfolding unitary11-gen-real
    by (auto simp add: mat-adj-def mat-cnj-def complex-cnj)
qed


lemma unitary11-gen-cis-blaschke:
  assumes k ≠ 0 M = k *_sm (a, b, cnj b, cnj a) a ≠ 0 mat-det (a, b, cnj b, cnj a) ≠ 0
  shows ∃ k' φ a'. k' ≠ 0 ∧ a' * cnj a' ≠ 1 ∧ M = k' *_sm (cis φ, 0, 0, 1) *_mm (1, −a', −cnj a', 1)
proof−
  have a = cnj a * cis (2 * arg a)
    using rcis-cmod-arg[of a] rcis-cnj[of a]
    using cis-rcis-eq rcis-mult
    by simp
  thus ?thesis
    using assms
    by (rule-tac x=k*cnj a in exI, rule-tac x=2*arg a in exI, rule-tac x=− b / a in exI) (auto simp add: field-simps complex-cnj)
qed

lemma unitary11-gen-cis-blaschke':
```

assumes $k \neq 0$ $M = k *_{sm} (-1, 0, 0, 1) *_{mm} (a, b, cnj\ b, cnj\ a)$ $a \neq 0$ $mat\text{-}det\ (a, b, cnj\ b, cnj\ a) \neq 0$
shows $\exists\ k'\ \varphi\ a'.\ k' \neq 0 \wedge a' * cnj\ a' \neq 1 \wedge M = k' *_{sm} (cis\ \varphi, 0, 0, 1) *_{mm} (1, -a', -cnj\ a', 1)$
proof−
obtain $k'\ \varphi\ a'$ **where** $*$: $k' \neq 0$ $k *_{sm} (a, b, cnj\ b, cnj\ a) = k' *_{sm} (cis\ \varphi, 0, 0, 1) *_{mm} (1, -a', -cnj\ a', 1)$ $a' * cnj\ a' \neq 1$
using *unitary11-gen-cis-blaschke*[*OF* ‹$k \neq 0$› - ‹$a \neq 0$›] ‹$mat\text{-}det\ (a, b, cnj\ b, cnj\ a) \neq 0$›
by *blast*
have $(cis\ \varphi, 0, 0, 1) *_{mm} (-1, 0, 0, 1) = (cis\ (\varphi + pi), 0, 0, 1)$
by (*simp add*: *cis-def*)
thus *?thesis*
using $*$ ‹$M = k *_{sm} (-1, 0, 0, 1) *_{mm} (a, b, cnj\ b, cnj\ a)$›
by (*rule-tac x=k'* **in** *exI*, *rule-tac x=φ + pi* **in** *exI*, *rule-tac x=a'* **in** *exI*, *simp*)
(*metis minus-mult-right equation-minus-iff minus-mult-left minus-mult-right*)
qed

lemma *unitary11-gen-cis-blaschke-rev*:
assumes $k' \neq 0$ $M = k' *_{sm} (cis\ \varphi, 0, 0, 1) *_{mm} (1, -a', -cnj\ a', 1)$ $a' * cnj\ a' \neq 1$
shows $\exists\ k\ a\ b.\ k \neq 0 \wedge mat\text{-}det\ (a, b, cnj\ b, cnj\ a) \neq 0 \wedge M = k *_{sm} (a, b, cnj\ b, cnj\ a)$
using *assms*
by (*rule-tac x=k'*cis(φ/2)* **in** *exI*, *rule-tac x=cis(φ/2)* **in** *exI*, *rule-tac x=−a'*cis(φ/2)* **in** *exI*) (*simp add*: *complex-cnj cis-mult*, *simp add*: *cis-def*)

lemma *unitary11-gen-cis-inversion*:
assumes $k \neq 0$ $M = k *_{sm} (0, b, cnj\ b, 0)$ $b \neq 0$
shows $\exists\ k'\ \varphi.\ k' \neq 0 \wedge M = k' *_{sm} (cis\ \varphi, 0, 0, 1) *_{mm} (0, 1, 1, 0)$
using *assms*
using *rcis-cmod-arg*[*of b*, *symmetric*] *rcis-cnj*[*of b*] *cis-rcis-eq*
by *simp* (*rule-tac x=2*arg b* **in** *exI*, *simp add*: *rcis-mult*)

lemma *unitary11-gen-cis-inversion'*:
assumes $k \neq 0$ $M = k *_{sm} (-1, 0, 0, 1) *_{mm} (0, b, cnj\ b, 0)$ $b \neq 0$
shows $\exists\ k'\ \varphi.\ k' \neq 0 \wedge M = k' *_{sm} (cis\ \varphi, 0, 0, 1) *_{mm} (0, 1, 1, 0)$
proof−
obtain $k'\ \varphi$ **where** $*$: $k' \neq 0$ $k *_{sm} (0, b, cnj\ b, 0) = k' *_{sm} (cis\ \varphi, 0, 0, 1) *_{mm} (0, 1, 1, 0)$
using *unitary11-gen-cis-inversion*[*OF* ‹$k \neq 0$› - ‹$b \neq 0$›]
by *metis*
have $(cis\ \varphi, 0, 0, 1) *_{mm} (-1, 0, 0, 1) = (cis\ (\varphi + pi), 0, 0, 1)$
by (*simp add*: *cis-def*)
thus *?thesis*
using $*$ ‹$M = k *_{sm} (-1, 0, 0, 1) *_{mm} (0, b, cnj\ b, 0)$›
by (*rule-tac x=k'* **in** *exI*, *rule-tac x=φ + pi* **in** *exI*, *simp*)
(*metis minus-mult-right*)
qed

lemma *unitary11-gen-cis-inversion-rev*:
assumes $k' \neq 0$ $M = k' *_{sm}$ (*cis* φ, *0*, *0*, *1*) $*_{mm}$ (*0*, *1*, *1*, *0*)
shows $\exists$ *k a b*. $k \neq 0$ $\wedge$ *mat-det* (*a*, *b*, *cnj b*, *cnj a*) $\neq 0$ $\wedge$ $M = k *_{sm}$ (*a*, *b*, *cnj b*, *cnj a*)
using *assms*
by (*rule-tac x=k'∗cis*(φ/*2*) **in** *exI*, *rule-tac x=0* **in** *exI*, *rule-tac x=cis*(φ/*2*) **in** *exI*) (*simp add*: *cis-mult*, *simp add*: *cis-def*)

lemma *unitary11-gen-iff*:
shows *unitary11-gen M* $\longleftrightarrow$ ($\exists$ *k a b* . $k \neq 0$ $\wedge$ *mat-det* (*a*, *b*, *cnj b*, *cnj a*) $\neq 0$ $\wedge$ $M = k *_{sm}$ (*a*, *b*, *cnj b*, *cnj a*)) (**is** *?lhs* = *?rhs*)
proof
assume *?lhs*
then obtain *a b k* **where** ∗: $k \neq 0$ *mat-det* (*a*, *b*, *cnj b*, *cnj a*) $\neq 0$ $M = k *_{sm}$ (*a*, *b*, *cnj b*, *cnj a*) $\vee$ $M = k *_{sm}$ (*−1*, *0*, *0*, *1*) $*_{mm}$ (*a*, *b*, *cnj b*, *cnj a*)
using *unitary11-gen-iff'*
by *auto*
show *?rhs*
proof (*cases* $M = k *_{sm}$ (*a*, *b*, *cnj b*, *cnj a*))
case *True*
thus *?thesis*
using ∗
by *auto*
next
case *False*
hence ∗∗: $M = k *_{sm}$ (*−1*, *0*, *0*, *1*) $*_{mm}$ (*a*, *b*, *cnj b*, *cnj a*)
using ∗
by *simp*
show *?thesis*
proof (*cases a* = *0*)
case *True*
hence $b \neq 0$
using ∗
by *auto*
show *?thesis*
using *unitary11-gen-cis-inversion-rev*[*of - M*]
using ∗∗ ‹*a* = *0*›
using *unitary11-gen-cis-inversion'*[*OF* ‹$k \neq 0$› - ‹$b \neq 0$›, *of M*]
by *auto*
next
case *False*
show *?thesis*
using *unitary11-gen-cis-blaschke-rev*[*of - M*]
using ∗∗
using *unitary11-gen-cis-blaschke'*[*OF* ‹$k \neq 0$› - ‹$a \neq 0$›, *of M b*] ‹*mat-det* (*a*, *b*, *cnj b*, *cnj a*) $\neq 0$›

```
        by blast
      qed
    qed
next
  assume ?rhs
  thus ?lhs
    using unitary11-gen-iff'
    by auto
qed

lemma unitary11-iff:
  shows unitary11 M ⟷
    (∃ a b k . (cmod a)^2 > (cmod b)^2 ∧ (cmod k)^2 = 1 / ((cmod a)^2 − (cmod b)^2)
∧ M = k *_sm (a, b, cnj b, cnj a)) (is ?lhs = ?rhs)
proof
  assume ?lhs
  obtain k a b where *:
    M = k *_sm (a, b, cnj b, cnj a)mat-det (a, b, cnj b, cnj a) ≠ 0 k ≠ 0
    using unitary11-gen-iff unitary11-unitary11-gen[OF ‹unitary11 M›]
    by auto

  have md: mat-det (a, b, cnj b, cnj a) = cor ((cmod a)^2 − (cmod b)^2)
    by (auto simp add: complex-mult-cnj-cmod)
  hence **: (cmod a)^2 ≠ (cmod b)^2
    using ‹mat-det (a, b, cnj b, cnj a) ≠ 0›
    by auto (metis of-real-power)

  have k * cnj k * mat-det (a, b, cnj b, cnj a) = 1
    using ‹M = k *_sm (a, b, cnj b, cnj a)›
    using ‹unitary11 M›
    unfolding unitary11-def
   by (auto simp add: mat-adj-def mat-cnj-def complex-cnj) (simp add: field-simps)
  hence (cmod k)^2 * ((cmod a)^2 − (cmod b)^2) = 1
   by (subst (asm) complex-mult-cnj-cmod, subst (asm) md, subst (asm) cor-mult[symmetric])
(metis of-real-1 of-real-eq-iff)
  thus ?rhs
    using ‹M = k *_sm (a, b, cnj b, cnj a)› ** mat-eye-l
    apply (rule-tac x=a in exI, rule-tac x=b in exI, rule-tac x=k in exI)
    apply (auto simp add: complex-mult-cnj-cmod)
      apply (metis less-iff-diff-less-0 linorder-neqE-linordered-idom mult-pow2-lt0
mult-zero-left not-one-less-zero zero-eq-power2 zero-neq-one)
   apply (metis ‹(cmod k)^2 * ((cmod a)^2 − (cmod b)^2) = 1› mult-eq-0-iff nonzero-eq-divide-eq
zero-neq-one)
    done
next
  assume ?rhs
  then obtain a b k where (cmod b)^2 < (cmod a)^2 ∧ (cmod k)^2 = 1 / ((cmod
a)^2 − (cmod b)^2) ∧ M = k *_sm (a, b, cnj b, cnj a)
    by auto
```

  moreover
  have cnj k * cnj a * (k * a) + − (cnj k * b * (k * cnj b)) = (cor ((cmod k)^2 * ((cmod a)^2 − (cmod b)^2)))
  proof−
    have cnj k * cnj a * (k * a) = cor ((cmod k)^2 * (cmod a)^2)
      using *complex-mult-cnj-cmod*[*of a*] *complex-mult-cnj-cmod*[*of k*]
      by (*auto simp add*: *field-simps*)
    moreover
    have cnj k * b * (k * cnj b) = cor ((cmod k)^2 * (cmod b)^2)
      using *complex-mult-cnj-cmod*[*of b*, *symmetric*] *complex-mult-cnj-cmod*[*of k*]
      by (*auto simp add*: *field-simps*)
    ultimately
    show *?thesis*
      by (*auto simp add*: *field-simps*)
  qed
  ultimately
  show *?lhs*
    unfolding *unitary11-def*
    by (*auto simp add*: *mat-adj-def mat-cnj-def complex-cnj field-simps*)
qed

lemma *unitary11-inv*:
  assumes k ≠ 0 M = k $*_{sm}$ (a, b, cnj b, cnj a) mat-det (a, b, cnj b, cnj a) ≠ 0
  shows ∃ k' a' b'. k' ≠ 0 ∧ mat-inv M = k' $*_{sm}$ (a', b', cnj b', cnj a') ∧ mat-det (a', b', cnj b', cnj a') ≠ 0
using *assms*
by (*subst assms*, *subst mat-inv-mult-sm*[*OF assms*(*1*)])
  (rule-tac x=1/(k * mat-det (a, b, cnj b, cnj a)) **in** *exI*, *rule-tac x=cnj a* **in** *exI*, *rule-tac x=−b* **in** *exI*, *simp add*: *complex-cnj field-simps*)

lemma *unitary11-comp*:
  assumes k1 ≠ 0 M1 = k1 $*_{sm}$ (a1, b1, cnj b1, cnj a1) mat-det (a1, b1, cnj b1, cnj a1) ≠ 0
          k2 ≠ 0 M2 = k2 $*_{sm}$ (a2, b2, cnj b2, cnj a2) mat-det (a2, b2, cnj b2, cnj a2) ≠ 0
  shows ∃ k a b. k ≠ 0 ∧ M1 $*_{mm}$ M2 = k $*_{sm}$ (a, b, cnj b, cnj a) ∧ mat-det (a, b, cnj b, cnj a) ≠ 0
using *assms*
apply (rule-tac x=k1*k2 **in** *exI*)
apply (rule-tac x=a1*a2 + b1*cnj b2 **in** *exI*)
apply (rule-tac x=a1*b2 + b1*cnj a2 **in** *exI*)
apply (*auto simp add*: *field-simps complex-cnj*)
apply *algebra*
done

lemma *unitary11-gen-mat-inv*:
  assumes *unitary11-gen M mat-det M* ≠ *0*

shows *unitary11-gen* (*mat-inv M*)
proof −
obtain *k a b* **where** $k \neq 0 \wedge$ *mat-det* (*a*, *b*, *cnj b*, *cnj a*) $\neq 0 \wedge M = k *_{sm}$ (*a*, *b*, *cnj b*, *cnj a*)
using *assms unitary11-gen-iff* [*of M*]
by *auto*
then obtain *k′ a′ b′* **where** $k' \neq 0 \wedge$ *mat-inv M* $= k' *_{sm}$ (*a′*, *b′*, *cnj b′*, *cnj a′*) $\wedge$ *mat-det* (*a′*, *b′*, *cnj b′*, *cnj a′*) $\neq 0$
using *unitary11-inv* [*of k M a b*]
by *auto*
thus *?thesis*
using *unitary11-gen-iff* [*of mat-inv M*]
by *auto*
qed

lemma *unitary11-gen-comp*:
assumes *unitary11-gen M1 mat-det M1* $\neq 0$ *unitary11-gen M2 mat-det M2* $\neq 0$
shows *unitary11-gen* ($M1 *_{mm} M2$)
proof −
from *assms* **obtain** *k1 k2 a1 a2 b1 b2* **where**
$k1 \neq 0 \wedge$ *mat-det* (*a1*, *b1*, *cnj b1*, *cnj a1*) $\neq 0 \wedge M1 = k1 *_{sm}$ (*a1*, *b1*, *cnj b1*, *cnj a1*)
$k2 \neq 0 \wedge$ *mat-det* (*a2*, *b2*, *cnj b2*, *cnj a2*) $\neq 0 \wedge M2 = k2 *_{sm}$ (*a2*, *b2*, *cnj b2*, *cnj a2*)
using *unitary11-gen-iff* [*of M1*] *unitary11-gen-iff* [*of M2*]
by *blast*
then obtain *k a b* **where** $k \neq 0 \wedge M1 *_{mm} M2 = k *_{sm}$ (*a*, *b*, *cnj b*, *cnj a*) $\wedge$ *mat-det* (*a*, *b*, *cnj b*, *cnj a*) $\neq 0$
using *unitary11-comp*[*of k1 M1 a1 b1 k2 M2 a2 b2*]
by *blast*
thus *?thesis*
using *unitary11-gen-iff* [*of* $M1 *_{mm} M2$]
by *blast*
qed

lemma *unitary11-sgn-det-orientation*:
assumes $k \neq 0$ *mat-det* (*a*, *b*, *cnj b*, *cnj a*) $\neq 0$ $M = k *_{sm}$ (*a*, *b*, *cnj b*, *cnj a*)
shows ∃ *k′*. *sgn k′* = *sgn* (*Re* (*mat-det* (*a*, *b*, *cnj b*, *cnj a*))) ∧ *congruence M* (*1*, *0*, *0*, −*1*) = *cor* $k' *_{sm}$ (*1*, *0*, *0*, −*1*)
proof −
let *?x* = *cnj k* ∗ *cnj a* ∗ (*k* ∗ *a*) − (*cnj k* ∗ *b* ∗ (*k* ∗ *cnj b*))
have ∗: *?x* = *k* ∗ *cnj k* ∗ (*a* ∗ *cnj a* − *b* ∗ *cnj b*)
by (*auto simp add*: *field-simps*)
hence *is-real ?x*
by *auto*
hence *cor* (*Re ?x*) = *?x*
by (*rule complex-of-real-Re*)

moreover
have *sgn (Re ?x) = sgn (Re (a ∗ cnj a − b ∗ cnj b))*
proof−
have ∗: *Re ?x =* $(cmod\ k)^2$ ∗ *Re (a ∗ cnj a − b ∗ cnj b)*
by (*subst* ∗, *subst complex-mult-cnj-cmod*, *subst Re-mult-real*) (*metis Im-complex-of-real*, *metis Re-complex-of-real*)
show *?thesis*
using ‹*k ≠ 0*›
by (*subst* ∗) (*simp add*: *sgn-mult*)
qed
ultimately
show *?thesis*
using *assms*(*3*)
by (*rule-tac x=Re ?x* **in** *exI*) (*auto simp add*: *mat-adj-def mat-cnj-def complex-cnj*)
qed

lemma *unitary11-sgn-det*:
assumes *k ≠ 0 mat-det (a, b, cnj b, cnj a) ≠ 0 M = k* $*_{sm}$ *(a, b, cnj b, cnj a)* *M = (A, B, C, D)*
shows *sgn (Re (mat-det (a, b, cnj b, cnj a))) = (if b = 0 then 1 else sgn (Re ((A∗D)/(B∗C)) − 1))*
proof (*cases b = 0*)
case *True*
thus *?thesis*
using *assms*
by (*simp only*: *mat-det.simps*, *subst complex-mult-cnj-cmod*, *subst complex-Re-diff*, *subst Re-complex-of-real*, *simp*)
next
case *False*
from *assms* **have** ∗: *A = k ∗ a B = k ∗ b C = k ∗ cnj b D = k ∗ cnj a*
by *auto*
hence ∗: *(A∗D)/(B∗C) = (a∗cnj a)/(b∗cnj b)*
using ‹*k ≠ 0*›
by *simp*
show *?thesis*
using ‹*b ≠ 0*›
apply (*subst* ∗, *subst Re-divide-real*, *simp*, *simp*)
apply (*simp only*: *mat-det.simps*)
apply (*subst complex-mult-cnj-cmod*)+
apply ((*subst Re-complex-of-real*)+, *subst complex-Re-diff*, (*subst Re-complex-of-real*)+, *simp add*: *field-simps sgn-if*)
done
qed

lemma *unitary11-orientation*:
assumes *unitary11-gen M M = (A, B, C, D)*
shows ∃ *k'. sgn k' = sgn (if B = 0 then 1 else sgn (Re ((A∗D)/(B∗C)) − 1))* ∧ *congruence M (1, 0, 0, −1) = cor k'* $*_{sm}$ *(1, 0, 0, −1)*
proof−

```
  from ‹unitary11-gen M›
  obtain k a b where ∗: k ≠ 0 mat-det (a, b, cnj b, cnj a) ≠ 0 M = k∗sm (a, b, cnj b, cnj a)
    using unitary11-gen-iff[of M]
    by auto
  moreover
  have b = 0 ⟷ B = 0
    using ‹M = (A, B, C, D)› ∗
    by auto
  ultimately
  show ?thesis
    using unitary11-sgn-det-orientation[OF ∗] unitary11-sgn-det[OF ∗ ‹M = (A, B, C, D)›]
    by auto
qed

lemma unitary11-sgn-det-orientation':
  assumes congruence M (1, 0, 0, −1) = cor k' ∗sm (1, 0, 0, −1) k' ≠ 0
  shows ∃ a b k. k ≠ 0 ∧ M = k ∗sm (a, b, cnj b, cnj a) ∧ sgn k' = sgn (Re (mat-det (a, b, cnj b, cnj a)))
proof−
  obtain a b k where
    k ≠ 0 mat-det (a, b, cnj b, cnj a) ≠ 0 M = k ∗sm (a, b, cnj b, cnj a)
    using assms
    using unitary11-gen-iff[of M]
    unfolding unitary11-gen-def
    by auto
  moreover
  have sgn k' = sgn (Re (mat-det (a, b, cnj b, cnj a)))
  proof−
    let ?x = cnj k ∗ cnj a ∗ (k ∗ a) − (cnj k ∗ b ∗ (k ∗ cnj b))
    have ∗: ?x = k ∗ cnj k ∗ (a ∗ cnj a − b ∗ cnj b)
      by (auto simp add: field-simps)
    hence is-real ?x
      by auto
    hence cor (Re ?x) = ?x
      by (rule complex-of-real-Re)

    have ∗∗: sgn (Re ?x) = sgn (Re (a ∗ cnj a − b ∗ cnj b))
    proof−
      have ∗: Re ?x = (cmod k)^2 ∗ Re (a ∗ cnj a − b ∗ cnj b)
        by (subst ∗, subst complex-mult-cnj-cmod, subst Re-mult-real) (metis Im-complex-of-real, metis Re-complex-of-real)
      show ?thesis
        using ‹k ≠ 0›
        by (subst ∗) (simp add: sgn-mult)
    qed
    moreover
    have ?x = cor k'
```

```
      using ‹M = k *sm (a, b, cnj b, cnj a)› assms
      by (simp add: mat-adj-def mat-cnj-def complex-cnj complex-diff-def)
    hence sgn (Re ?x) = sgn k'
      using ‹cor (Re ?x) = ?x›
      unfolding complex-of-real-def
      by simp
    ultimately
    show ?thesis
      by simp
  qed
  ultimately
  show ?thesis
    by (rule-tac x=a in exI, rule-tac x=b in exI, rule-tac x=k in exI) simp
qed

end
```

6 Hermitean matrices

```
theory HermiteanMatrices
imports UnitaryMatrices
begin
```

Hermitean matrices

```
definition hermitean :: complex-mat ⇒ bool where
 hermitean A ⟷ mat-adj A = A

lemma hermitean A ⟷ mat-transpose A = mat-cnj A
unfolding hermitean-def
by (cases A) (auto simp add: mat-adj-def mat-cnj-def)

lemma hermitean-mat-cnj: hermitean H ⟷ hermitean (mat-cnj H)
by (cases H) (auto simp add: hermitean-def mat-adj-def mat-cnj-def)

lemma hermitean-mult-real:
  assumes hermitean H
  shows hermitean ((cor k) *sm H)
using assms
unfolding hermitean-def
by simp

lemma hermitean-congruence:
  assumes hermitean H
  shows hermitean (congruence M H)
using assms
unfolding hermitean-def
by (auto simp add: mult-mm-assoc)

lemma hermitean-elems:
  assumes hermitean (A, B, C, D)
  shows is-real A is-real D B = cnj C cnj B = C
using assms eq-cnj-iff-real[of A] eq-cnj-iff-real[of D]
by (auto simp add: hermitean-def mat-adj-def mat-cnj-def)

lemma mat-det-hermitean-real:
  assumes hermitean A
  shows is-real (mat-det A)
using assms
unfolding hermitean-def
by (cases A, auto simp add: mat-adj-def mat-cnj-def) (metis add-0-iff eq-cnj-iff-real
mult-eq-0-iff)

lemma Re-det-sgn-congruence:
  assumes hermitean H mat-det M ≠ 0
  shows sgn (Re (mat-det (congruence M H))) = sgn (Re (mat-det H))
proof -
  have *: mat-det (mat-adj M *_mm H *_mm M) =
    (cor ((cmod (mat-det M))^2)) * mat-det H
    using complex-mult-cnj-cmod[of mat-det M]
    by (auto simp add: mat-det-adj field-simps)
  have *: Re (mat-det (mat-adj M *_mm H *_mm M)) =
    (cmod (mat-det M))^2 * Re (mat-det H)
   by (subst *, subst Re-mult-real, rule is-real-complex-of-real) (subst Re-complex-of-real,
simp)
  show ?thesis
    using assms
    by (subst *) (auto simp add: sgn-mult)
qed

lemma det-sgn-congruence:
  assumes hermitean H mat-det M ≠ 0
  shows sgn (mat-det (congruence M H)) = sgn (mat-det H)
proof -
  have *: mat-det (mat-adj M *_mm H *_mm M) =
    (cor ((cmod (mat-det M))^2)) * mat-det H
    using complex-mult-cnj-cmod[of mat-det M]
    by (auto simp add: mat-det-adj field-simps)
  thus ?thesis
    using assms
      by (subst *, auto simp add: sgn-mult power2-eq-square) (smt mult-eq-0-iff
norm-divide norm-mult norm-sgn of-real-1 of-real-divide of-real-mult sgn-eq times-divide-times-eq)
qed

lemma bilinear-form-hermitean-commute:
  assumes hermitean H
  shows bilinear-form v1 v2 H = cnj (bilinear-form v2 v1 H)
proof -
  have v2 *_vm mat-cnj H *_vv vec-cnj v1 = vec-cnj v1 *_vv (mat-adj H *_mv v2)
```

by (*subst mult-vv-commute*, *subst mult-mv-mult-vm*, *simp add*: *mat-adj-def mat-transpose-mat-cnj*)
also
have . . . = *bilinear-form v1 v2 H*
using *assms*
by (*simp add*: *mult-vv-mv hermitean-def*)
finally
show *?thesis*
by (*simp add*: *cnj-mult-vv vec-cnj-mult-vm*)
qed

lemma *quad-form-hermitean-real*:
assumes *hermitean H*
shows *is-real* (*quad-form z H*)
using *assms*
by (*subst eq-cnj-iff-real*[*symmetric*]) (*simp del*: *quad-form-def add*: *hermitean-def*)

Eigenvalues, eigenvectors and diagonalization of Hermitean matrices

lemma *hermitean-eigenval-real*:
assumes *hermitean H eigenval k H*
shows *is-real k*
proof−
from *assms* **obtain** *v* **where** $v \neq$ *vec-zero* $H *_{mv} v = k *_{sv} v$
unfolding *eigenval-def*
by *blast*
have $k * (v *_{vv}$ *vec-cnj* $v) = (k *_{sv} v) *_{vv}$ (*vec-cnj* v)
by (*simp add*: *mult-vv-scale-sv1*)
also have ... = $(H *_{mv} v) *_{vv}$ (*vec-cnj* v)
using ‹$H *_{mv} v = k *_{sv} v$›
by *simp*
also have ... = $v *_{vv}$ (*mat-transpose* $H *_{mv}$ (*vec-cnj* v))
by (*simp add*: *mult-mv-vv*)
also have ... = $v *_{vv}$ (*vec-cnj* (*mat-cnj* (*mat-transpose* H) $*_{mv} v$))
by (*simp add*: *vec-cnj-mult-mv*)
also have ... = $v *_{vv}$ (*vec-cnj* $(H *_{mv} v)$)
using ‹*hermitean H*›
by (*simp add*: *hermitean-def mat-adj-def*)
also have ... = $v *_{vv}$ (*vec-cnj* $(k *_{sv} v)$)
using ‹$H *_{mv} v = k *_{sv} v$›
by *simp*
finally have $k * (v *_{vv}$ *vec-cnj* $v)$ = *cnj* $k * (v *_{vv}$ *vec-cnj* $v)$
by (*simp add*: *mult-vv-scale-sv2*)
hence k = *cnj* k
using ‹$v \neq$ *vec-zero*›
using *scalsquare-vv-zero*[*of v*]
by (*simp add*: *mult-vv-commute*)
thus *?thesis*
by (*metis eq-cnj-iff-real*)
qed

```
lemma hermitean-distinct-eigenvals:
  assumes hermitean H
  shows (∃ k_1 k_2. k_1 ≠ k_2 ∧ eigenval k_1 H ∧ eigenval k_2 H) ∨ mat-diagonal H
proof−
  obtain A B C D where HH: H = (A, B, C, D)
    by (cases H) auto
  show ?thesis
  proof (cases B = 0)
    case True
    thus ?thesis
      using ‹hermitean H› hermitean-elems[of A B C D] HH
      by auto
  next
    case False
    have (mat-trace H)^2 ≠ 4 * mat-det H
    proof (rule ccontr)
      have C = cnj B is-real A is-real D
        using hermitean-elems HH ‹hermitean H›
        by auto
      assume ¬ ?thesis
      hence (A + D)^2 = 4*(A*D − B*C)
        using HH
        by auto
      hence (A − D)^2 = − 4*B*cnj B
        using ‹C = cnj B›
        by (auto simp add: power2-eq-square field-simps) algebra
      hence (A − D)^2 / cor ((cmod B)^2) = −4
        using ‹B ≠ 0› complex-mult-cnj-cmod[of B]
        by (auto simp add: field-simps)
      hence (Re A − Re D)^2 / (cmod B)^2 = −4
        using ‹is-real A› ‹is-real D› ‹B ≠ 0›
        using Re-divide-real[of cor ((cmod B)^2) (A − D)^2]
        by (auto simp add: power2-eq-square)
      thus False
      by (metis abs-neg-numeral abs-power2 neg-numeral-neq-numeral power-divide)
    qed
    show ?thesis
      apply (rule disjI1)
      apply (subst eigen-equation)+
      using complex-quadratic-two-solutions[of −mat-trace H mat-det H] ‹(mat-trace H)^2 ≠ 4 * mat-det H›
      apply auto
      apply (rule-tac x=k_1 in exI, rule-tac x=k_2 in exI)
      apply (simp add: complex-diff-def)
      done
  qed
qed
```

```
lemma hermitean-ortho-eigenvecs:
  assumes hermitean H
  assumes eigenpair k1 v1 H eigenpair k2 v2 H k1 ≠ k2
  shows vec-cnj v2 ∗vv v1 = 0 vec-cnj v1 ∗vv v2 = 0
proof−
  from assms
  have v1 ≠ vec-zero H ∗mv v1 = k1 ∗sv v1
       v2 ≠ vec-zero H ∗mv v2 = k2 ∗sv v2
    unfolding eigenpair-def
    by auto
  have real-k: is-real k1 is-real k2
    using assms
    using hermitean-eigenval-real[of H k1]
    using hermitean-eigenval-real[of H k2]
    unfolding eigenpair-def eigenval-def
    by blast+

  have vec-cnj (H ∗mv v2) = vec-cnj (k2 ∗sv v2)
    using ‹H ∗mv v2 = k2 ∗sv v2›
    by auto
  hence vec-cnj v2 ∗vm H = k2 ∗sv vec-cnj v2
    using ‹hermitean H› real-k eq-cnj-iff-real[of k1] eq-cnj-iff-real[of k2]
    unfolding hermitean-def
     by (cases H, cases v2) (auto simp add: mat-adj-def mat-cnj-def vec-cnj-def complex-cnj)
  have k2 ∗ (vec-cnj v2 ∗vv v1) = k1 ∗ (vec-cnj v2 ∗vv v1)
    using ‹H ∗mv v1 = k1 ∗sv v1›
    using ‹vec-cnj v2 ∗vm H = k2 ∗sv vec-cnj v2›
   by (cases v1, cases v2, cases H) (auto simp add: vec-cnj-def field-simps, algebra)
  thus vec-cnj v2 ∗vv v1 = 0
    using ‹k1 ≠ k2›
    by simp
  hence cnj (vec-cnj v2 ∗vv v1) = 0
    by simp
  thus vec-cnj v1 ∗vv v2 = 0
    by (simp add: cnj-mult-vv mult-vv-commute)
qed

lemma hermitean-diagonizable:
  assumes hermitean H
  shows ∃ k1 k2 M. mat-det M ≠ 0 ∧ unitary M ∧ congruence M H = (k1, 0, 0, k2) ∧
                    is-real k1 ∧ is-real k2 ∧ sgn (Re k1 ∗ Re k2) = sgn (Re (mat-det H))
proof−
  from assms
  have (∃ k₁ k₂. k₁ ≠ k₂ ∧ eigenval k₁ H ∧ eigenval k₂ H) ∨ mat-diagonal H
    using hermitean-distinct-eigenvals[of H]
    by simp
```

```
thus ?thesis
proof
  assume ∃k_1 k_2. k_1 ≠ k_2 ∧ eigenval k_1 H ∧ eigenval k_2 H
  then obtain k1 k2 where k1 ≠ k2 eigenval k1 H eigenval k2 H
    using hermitean-distinct-eigenvals
    by blast
  then obtain v1 v2 where eigenpair k1 v1 H eigenpair k2 v2 H
    v1 ≠ vec-zero v2 ≠ vec-zero
    unfolding eigenval-def eigenpair-def
    by blast
  hence *: vec-cnj v2 *_vv v1 = 0 vec-cnj v1 *_vv v2 = 0
    using ‹k1 ≠ k2› hermitean-ortho-eigenvecs ‹hermitean H›
    by auto
  obtain v11 v12 v21 v22 where vv: v1 = (v11, v12) v2 = (v21, v22)
    by (cases v1, cases v2) auto
  let ?nv1' = vec-cnj v1 *_vv v1 and ?nv2' = vec-cnj v2 *_vv v2
  let ?nv1 = cor (sqrt (Re ?nv1'))
  let ?nv2 = cor (sqrt (Re ?nv2'))
  have ?nv1' ≠ 0 ?nv2' ≠ 0
    using ‹v1 ≠ vec-zero› ‹v2 ≠ vec-zero› vv
    by (simp add: scalsquare-vv-zero)+
  moreover
  have is-real ?nv1' is-real ?nv2'
    using vv
    by (auto simp add: vec-cnj-def)
  ultimately
  have ?nv1 ≠ 0 ?nv2 ≠ 0
    by − (cases ?nv1', cases ?nv2', auto)+
  have Re (?nv1') ≥ 0 Re (?nv2') ≥ 0
    using vv
    by (auto simp add: vec-cnj-def)
  obtain nv1 nv2 where nv1 = ?nv1 nv1 ≠ 0 nv2 = ?nv2 nv2 ≠ 0
    using ‹?nv1 ≠ 0› ‹?nv2 ≠ 0›
    by auto
  let ?M = (1/nv1 * v11, 1/nv2 * v21, 1/nv1 * v12, 1/nv2 * v22)

  have is-real k1 is-real k2
    using ‹eigenval k1 H› ‹eigenval k2 H› ‹hermitean H›
    by (auto simp add: hermitean-eigenval-real)
  moreover
  have mat-det ?M ≠ 0
  proof (rule ccontr)
    assume ¬ ?thesis
    hence v11 * v22 = v12 * v21
      using ‹nv1 ≠ 0› ‹nv2 ≠ 0›
      by (auto simp add: field-simps)
    hence ∃ k. k ≠ 0 ∧ v2 = k *_sv v1
      using vv ‹v1 ≠ vec-zero› ‹v2 ≠ vec-zero›
      apply auto
```

```
        apply (rule-tac x=v21/v11 in exI, force simp add: field-simps)
        apply (rule-tac x=v21/v11 in exI, force simp add: field-simps)
        apply (rule-tac x=v22/v12 in exI, force simp add: field-simps)
        apply (rule-tac x=v22/v12 in exI, force simp add: field-simps)
        done
      thus False
        using ‹vec-cnj v1 *vv v2 = 0› vv ‹?nv1' ≠ 0›
      by (auto simp add: vec-cnj-def field-simps) (metis comm-semiring-1-class.normalizing-semiring-rules(34)
mult-eq-0-iff)
    qed
    moreover
    have unitary ?M
    proof-
      have **: cnj nv1 * nv1 = ?nv1'  cnj nv2 * nv2 = ?nv2'
          using ‹nv1 = ?nv1› ‹nv1 ≠ 0›  ‹nv2 = ?nv2› ‹nv2 ≠ 0› ‹is-real ?nv1'›
‹is-real ?nv2'›
        using ‹Re (?nv1') ≥ 0›  ‹Re (?nv2') ≥ 0›
        by (auto simp add: complex-of-real-Re)
      have ***: cnj nv1 * nv2 ≠ 0  cnj nv2 * nv1 ≠ 0
         using vv ‹nv1 = ?nv1› ‹nv1 ≠ 0›  ‹nv2 = ?nv2› ‹nv2 ≠ 0› ‹is-real ?nv1'›
‹is-real ?nv2'›
        by auto

      show ?thesis
        unfolding unitary-def
        using vv ** ‹?nv1' ≠ 0› ‹?nv2' ≠ 0› * ***
        apply (auto simp add: mat-adj-def mat-cnj-def vec-cnj-def complex-cnj)
        apply (metis add-divide-distrib divide-self-if)
        apply (metis add-divide-distrib divide-zero-left)
        apply (metis add-divide-distrib divide-zero-left)
        apply (metis add-divide-distrib divide-self-if)
        done
    qed
    moreover
    have congruence ?M H = (k1, 0, 0, k2)
    proof-
      have mat-inv ?M *mm H *mm ?M = (k1, 0, 0, k2)
      proof-
        have *: H *mm ?M = ?M *mm (k1, 0, 0, k2)
          using ‹eigenpair k1 v1 H› ‹eigenpair k2 v2 H› vv ‹?nv1 ≠ 0› ‹?nv2 ≠ 0›
          unfolding eigenpair-def
          apply (cases H)
          apply (auto simp add: vec-cnj-def)
          apply (metis add-divide-distrib mult.commute)+
          done
        show ?thesis
          using mult-mm-inv-l[of ?M (k1, 0, 0, k2) H *mm ?M, OF ‹mat-det ?M
≠ 0› *[symmetric], symmetric]
          by (simp add: mult-mm-assoc)
      qed
      moreover
      have mat-inv ?M = mat-adj ?M
         using ‹mat-det ?M ≠ 0› ‹unitary ?M› mult-mm-inv-r[of ?M mat-adj ?M
eye]
        by (simp add: unitary-def)
      ultimately
      show ?thesis
        by simp
    qed
    moreover
    have sgn (Re k1 ∗ Re k2) = sgn (Re (mat-det H))
      using ‹congruence ?M H = (k1, 0, 0, k2)› ‹is-real k1› ‹is-real k2›
      using Re-det-sgn-congruence[of H ?M] ‹mat-det ?M ≠ 0› ‹hermitean H›
      by simp
    ultimately
    show ?thesis
      by (rule-tac x=k1 in exI, rule-tac x=k2 in exI, rule-tac x=?M in exI) simp
  next
    assume mat-diagonal H
    then obtain A D where H = (A, 0, 0, D)
      by (cases H) auto
    moreover
    hence is-real A is-real D
      using ‹hermitean H› hermitean-elems[of A 0 0 D]
      by auto
    ultimately
    show ?thesis
      by (rule-tac x=A in exI, rule-tac x=D in exI, rule-tac x=eye in exI) (simp
add: unitary-def mat-adj-def mat-cnj-def)
  qed
qed

end
```

7 Elementary complex geometry

```
theory ElementaryComplexGeometry
imports MoreComplex LinearSystems
begin



definition colinear :: complex ⇒ complex ⇒ complex ⇒ bool where
  colinear z1 z2 z3 ⟷ z1 = z2 ∨ Im ((z3 − z1)/(z2 − z1)) = 0

lemma colinear-ex-real:
  colinear z1 z2 z3 ⟷ (∃ k::real. z1 = z2 ∨ z3 − z1 = complex-of-real k ∗ (z2
− z1))
```

```
unfolding colinear-def
by (auto split: split-if-asm) (metis Im.simps complex.exhaust complex-of-real-def
eq-iff-diff-eq-0 nonzero-divide-eq-eq)

lemma colinear-sym1:
  colinear z1 z2 z3 ⟷ colinear z1 z3 z2
unfolding colinear-def
using div-reals[of 1 (z3 − z1)/(z2 − z1)]  div-reals[of 1 (z2 − z1)/(z3 − z1)]
by auto

lemma colinear-sym2':
  assumes colinear z1 z2 z3
  shows colinear z2 z1 z3
proof−
  obtain k where z1 = z2 ∨ z3 − z1 = complex-of-real k ∗ (z2 − z1)
    using assms
    unfolding colinear-ex-real
    by auto
  thus ?thesis
  proof
    assume z3 − z1 = complex-of-real k ∗ (z2 − z1)
    thus ?thesis
      unfolding colinear-ex-real
      by (rule-tac x=1−k in exI) (auto simp add: field-simps)
  qed (simp add: colinear-def)
qed

lemma colinear-sym2:
  colinear z1 z2 z3 ⟷ colinear z2 z1 z3
  using colinear-sym2'[of z1 z2 z3] colinear-sym2'[of z2 z1 z3]
  by auto

lemma colinear-trans1:
  assumes colinear z0 z2 z1 colinear z0 z3 z1 z0 ≠ z1
  shows colinear z0 z2 z3
using assms
unfolding colinear-ex-real
by (cases z0 = z2, auto) (rule-tac x=k/ka in exI, case-tac ka = 0, auto simp
add: field-simps)

lemma colinear-det:
  assumes ¬ colinear z2 z3 z1
  shows det2 (z1 − z2) (cnj (z1 − z2)) (z2 − z3) (cnj (z2 − z3)) ≠ 0
proof−
  from assms have ((z1 − z2) / (z3 − z2)) − cnj ((z1 − z2) / (z3 − z2)) ≠ 0
z3 ≠ z2
    unfolding colinear-def
    using im-complex[of (z1 − z2) / (z3 − z2)]
    by auto
  thus ?thesis
   by (auto simp add: field-simps complex-cnj-divide complex-cnj-add complex-cnj-diff)
qed



definition line :: complex ⇒ complex ⇒ complex set where
  line z1 z2 = {z. colinear z1 z2 z}

lemma line-points-colinear:
  assumes z1 ∈ line z z′ z2 ∈ line z z′ z3 ∈ line z z′ z ≠ z′
  shows colinear z1 z2 z3
using assms
unfolding line-def
by auto  (smt colinear-sym1 colinear-sym2 colinear-trans1)

lemma line-param:
  shows z1 + complex-of-real k ∗ (z2 − z1) ∈ line z1 z2
  unfolding line-def
  by (auto simp add: colinear-def)

definition circle :: complex ⇒ real ⇒ complex set where
  circle μ r = {z. cmod (z − μ) = r}



lemma line-equation:
  assumes z1 ≠ z2 μ = rot90 (z2 − z1)
  shows line z1 z2 = {z. cnj μ∗z + μ∗cnj z − (cnj μ ∗ z1 + μ ∗ cnj z1)  = 0}
proof−
  {
    fix z
    have z ∈ line z1 z2 ⟷ Im ((z − z1)/(z2 − z1)) = 0
      using assms
      by (simp add: line-def colinear-def)
    also have ... ⟷ (z − z1)/(z2 − z1) = cnj ((z − z1)/(z2 − z1))
      using complex-diff-cnj[of (z − z1)/(z2 − z1)]
      by auto
    also have ... ⟷ (z − z1)∗(cnj z2 − cnj z1) = (cnj z − cnj z1)∗(z2 − z1)
      using assms(1)
     by auto (metis (lifting) complex-cnj-cancel-iff complex-cnj-diff complex-cnj-divide
frac-eq-eq right-minus-eq)+
     also have ... ⟷ cnj(z2 − z1)∗z − (z2 − z1)∗cnj z − (cnj(z2 − z1)∗z1 −
(z2 − z1)∗cnj z1) = 0
      by (simp add: complex-cnj-diff field-simps)
    also have ... ⟷ cnj μ ∗ z + μ ∗ cnj z  − (cnj μ ∗ z1 + μ ∗ cnj z1) = 0
      using assms cnj-mix-minus
      by simp
    finally have z ∈ line z1 z2 ⟷ cnj μ ∗ z + μ ∗ cnj z  − (cnj μ ∗ z1 + μ ∗
cnj z1) = 0
   .
 }
 thus ?thesis
   by auto
qed

lemma circle-equation:
 assumes r ≥ 0
 shows circle μ r = {z. z∗cnj z − z∗cnj μ − cnj z∗μ + μ∗cnj μ − complex-of-real
(r∗r) = 0}
proof (safe)
 fix z

 assume z ∈ circle μ r
 hence (z − μ)∗cnj (z − μ) = complex-of-real (r∗r)
   unfolding circle-def
   using complex-mult-cnj-cmod[of z − μ]
   by (auto simp add: power2-eq-square)
 thus z ∗ cnj z − z ∗ cnj μ − cnj z ∗ μ + μ ∗ cnj μ − complex-of-real (r ∗ r)
= 0
   by (auto simp add: field-simps complex-cnj-diff)
next
 fix z
 assume z ∗ cnj z − z ∗ cnj μ − cnj z ∗ μ + μ ∗ cnj μ − complex-of-real (r ∗
r) = 0
 hence (z − μ)∗cnj (z − μ) = complex-of-real (r∗r)
   by (auto simp add: field-simps complex-cnj-diff)
 thus z ∈ circle μ r
   using assms
   using complex-mult-cnj-cmod[of z − μ]
   using power2-eq-imp-eq[of cmod (z − μ) r]
   unfolding circle-def power2-eq-square[symmetric] complex-of-real-def
   by auto
qed



definition circline where
 circline A BC D = {z. cor A∗z∗cnj z + cnj BC∗z + BC∗cnj z + cor D = 0}

lemma circline-circle:
 assumes A ≠ 0  A ∗ D ≤ (cmod BC)^2
 cl = circline A BC D
 μ = −BC/complex-of-real A r2 = ((cmod BC)^2 − A∗D) / A^2 r = sqrt r2
 shows cl = circle μ r
proof−
  have ∗: cl = {z. z ∗ cnj z + cnj (BC / complex-of-real A) ∗ z + (BC /
complex-of-real A) ∗ cnj z + complex-of-real (D / A) = 0}
```

```
    using ‹cl = circline A BC D› ‹A ≠ 0›
    by (auto simp add: circline-def complex-cnj-divide field-simps)

  have r2 ≥ 0
  proof−
    have (cmod BC)^2 − A ∗ D ≥  0
      using ‹A ∗ D ≤ (cmod BC)^2›
      by auto
    thus ?thesis
      using ‹A ≠ 0› ‹r2 = ((cmod BC)^2 − A∗D) / A^2›
      by (metis zero-le-divide-iff zero-le-power2)
  qed
  hence ∗∗: r ∗ r = r2 r ≥ 0
    using ‹r = sqrt r2›
    by (auto simp add: real-sqrt-mult[symmetric])

  have ∗∗∗: − μ ∗ − cnj μ − complex-of-real r2 = complex-of-real (D / A)
    using ‹μ = − BC / complex-of-real A› ‹r2 = ((cmod BC)^2 − A ∗ D) / A^2›
   by (auto simp add: complex-cnj-divide complex-cnj-minus complex-mult-cnj-cmod
power2-eq-square complex-of-real-def complex-divide-def div-reals field-simps intro!:
complex-eqI)
  thus ?thesis
    using ‹r2 = ((cmod BC)^2 − A∗D) / A^2› ‹μ = − BC / complex-of-real A›
     by (subst ∗, subst circle-equation[of r μ, OF ‹r ≥ 0›], subst ∗∗) (auto simp
add: complex-cnj-minus complex-cnj-divide field-simps power2-eq-square)
qed

lemma circline-ex-circle:
  assumes A ≠ 0  A ∗ D ≤ (cmod BC)^2
  cl = circline A BC D
  shows ∃ μ r. cl = circle μ r
using circline-circle[OF assms]
by auto

lemma circle-circline:
  assumes cl = circle μ r r ≥ 0
  shows cl = circline 1 (−μ) ((cmod μ)^2 − r^2)
proof−
  have complex-of-real ((cmod μ)^2 − r^2) = μ ∗ cnj μ − complex-of-real (r^2)
    by (auto simp add: complex-mult-cnj-cmod)
  thus cl = circline 1 (− μ) ((cmod μ)^2 − r^2)
    using assms
    using circle-equation[of r μ]
    unfolding circline-def power2-eq-square
    by (simp add: complex-cnj-minus field-simps)
qed

lemma circle-ex-circline:
  assumes cl = circle μ r r ≥ 0
```

shows ∃ *A BC D*. *A* ≠ *0* ∧ *A*∗*D* ≤ (*cmod BC*)^2 ∧ *cl* = *circline A BC D*
using *circle-circline*[*OF assms*]
using ‹*r* ≥ *0*›
by (*rule-tac x*=*1* **in** *exI*, *rule-tac x*=−*μ* **in** *exI*, *rule-tac x*=*Re* (*μ* ∗ *cnj μ*) − (*r* ∗ *r*) **in** *exI*) (*simp add*: *complex-mult-cnj-cmod power2-eq-square*)

lemma *circline-line*:
assumes
A = *0 BC* ≠ *0*
cl = *circline A BC D*
z1 = − *cor D* ∗ *BC* / (*2* ∗ *BC* ∗ *cnj BC*)
z2 = *z1* + *ii* ∗ *sgn* (*if arg BC* > *0 then* −*BC else BC*)
shows
cl = *line z1 z2*
proof−
have *cl* = {*z*. *cnj BC*∗*z* + *BC*∗*cnj z* + *complex-of-real D* = *0*}
using *assms*
by (*simp add*: *circline-def*)
have {*z*. *cnj BC*∗*z* + *BC*∗*cnj z* + *complex-of-real D* = *0*} =
{*z*. *cnj BC*∗*z* + *BC*∗*cnj z* − (*cnj BC*∗*z1* + *BC*∗*cnj z1*) = *0*}
using ‹*BC* ≠ *0*› *assms*
by (*auto simp add*: *complex-cnj-minus complex-cnj-divide complex-cnj-mult*)
moreover
have *z1* ≠ *z2*
using ‹*BC* ≠ *0*› *assms*
by (*auto simp add*: *sgn-eq*)
moreover
have ∃ *k*. *k* ≠ *0* ∧ *BC* = *cor k*∗*rot90* (*z2* − *z1*)
using *assms*
apply *auto*
apply (*rule-tac x*=(*cmod BC*) **in** *exI*, *simp*, *metis Complex.Re-sgn Im-sgn cmod-cis mult.commute complex-surj eq-divide-eq mult-zero-left sgn-eq*)
apply (*rule-tac x*=−(*cmod BC*) **in** *exI*, *simp*,*metis Complex.Re-sgn Im-sgn cis-arg cmod-cis mult.commute complex-minus-def minus-minus minus-mult-left*)
done
then obtain *k* **where** *cor k* ≠ *0 BC* = *cor k*∗*rot90* (*z2* − *z1*)
by *auto*
moreover
have ∗: ⋀ *z*. *cnj-mix* (*BC* / *cor k*) *z* − *cnj-mix* (*BC* / *cor k*) *z1* = (*1*/*cor k*) ∗ (*cnj-mix BC z* − *cnj-mix BC z1*)
using ‹*cor k* ≠ *0*›
by (*simp add*: *complex-cnj field-simps*)
hence {*z*. *cnj-mix BC z* − *cnj-mix BC z1* = *0*} = {*z*. *cnj-mix* (*BC* / *cor k*) *z* − *cnj-mix* (*BC* / *cor k*) *z1* = *0*}
using ‹*cor k* ≠ *0*›
by *auto*
ultimately
have *cl* = *line z1 z2*

```
      using line-equation[of z1 z2 BC/cor k] ‹cl = {z. cnj BC*z + BC*cnj z + complex-of-real D = 0}›
      by auto
    thus ?thesis
      using ‹z1 ≠ z2›
      by blast
qed


lemma circline-ex-line:
assumes
  A = 0 BC ≠ 0
  cl = circline A BC D
shows ∃ z1 z2. z1 ≠ z2 ∧ cl = line z1 z2
proof−
  let ?z1 = − cor D * BC / (2 * BC * cnj BC)
  let ?z2 = ?z1 + i * sgn (if 0 < arg BC then − BC else BC)
  have ?z1 ≠ ?z2
    using ‹BC ≠ 0›
    by (simp add: sgn-eq)
  thus ?thesis
    using circline-line[OF assms, of ?z1 ?z2] ‹BC ≠ 0›
    by (rule-tac x=?z1 in exI, rule-tac x=?z2 in exI, simp)
qed

lemma line-ex-circline:
  assumes cl = line z1 z2 z1 ≠ z2
  shows ∃ BC D. BC ≠ 0 ∧ cl = circline 0 BC D
proof−
  let ?BC = rot90 (z2 − z1)
  let ?D = Re (− 2 * scalprod z1 ?BC)
  show ?thesis
  proof (rule-tac x=?BC in exI, rule-tac x=?D in exI, rule conjI)
    show ?BC ≠ 0
      using ‹z1 ≠ z2›
        by (metis complex-minus-def eq-iff-diff-eq-0 i-mult-Complex minus-diff-eq mult-zero-right)
  next
     have *: complex-of-real (Re (− 2 * scalprod z1 (rot90 (z2 − z1)))) = −(cnj-mix z1 (rot90 (z2 − z1)))
      by (cases z1, cases z2, auto simp add: complex-of-real-def field-simps)
    show cl = circline 0 ?BC ?D
      apply (subst assms, subst line-equation[of z1 z2 ?BC])
      unfolding circline-def
      by (fact, simp, subst *, simp add: field-simps)
  qed
qed

end
```

theory *Angles*
imports *MoreComplex*
begin

definition *ang-vec* () **where**
  [*simp*]: *z1 z2* ≡ ⇂*arg z2* − *arg z1*⇃

definition *ang-vec-c* (*c*) **where**
[*simp*]:*c z1 z2* ≡ *abs* (*z1 z2*)

definition *acute-ang* **where**
  [*simp*]: *acute-ang* α = (*if* α > *pi* / *2 then pi* − α *else* α)

definition *ang-vec-a* (*a*) **where**
[*simp*]: *a z1 z2* ≡ *acute-ang* (*c z1 z2*)

lemma *ang-vec-sym*:
  assumes *z1 z2* ≠ *pi*
  shows *z1 z2* = − *z2 z1*
using *assms*
unfolding *ang-vec-def*
using *canon-ang-uminus*[*of arg z2* − *arg z1*]
by *simp*

lemma *ang-vec-sym-pi*:
  assumes *z1 z2* = *pi*
  shows *z1 z2* = *z2 z1*
using *assms*
unfolding *ang-vec-def*
using *canon-ang-uminus-pi*[*of arg z2* − *arg z1*]
by *simp*

lemma *ang-vec-c-sym*:
  shows *c z1 z2* = *c z2 z1*
unfolding *ang-vec-c-def*
using *ang-vec-sym-pi*[*of z1 z2*] *ang-vec-sym*[*of z1 z2*]
by (*cases* *z1 z2* = *pi*) *auto*

lemma *ang-vec-a-sym*:

```
  a z1 z2 = a z2 z1
  unfolding ang-vec-a-def
  using ang-vec-c-sym
  by auto

lemma ang-vec-c-bounded: 0 ≤ c z1 z2 ∧ c z1 z2 ≤ pi
using canon-ang(1)[of arg z2 − arg z1] canon-ang(2)[of arg z2 − arg z1]
by auto


lemma ortho-c-scalprod0:
  assumes z1 ≠ 0 z2 ≠ 0
  shows c z1 z2 = pi/2 ⟷ scalprod z1 z2 = 0
proof
  assume c z1 z2 = pi / 2
  have ⌊arg z2 − arg z1⌋ = arg (z2 / z1)
    using arg-div[of z2 z1] assms
    by auto
  hence arg (z2 / z1) = pi/2 ∨ arg (z2 / z1) = −pi/2
    using ‹c z1 z2 = pi/2›
    unfolding ang-vec-c-def
    unfolding ang-vec-def
    by auto
  hence Re (z2 / z1) = 0
    using re-complex-zero-arg1[of z2/z1]
    by auto
  hence z2 / z1 + cnj (z2 / z1) = 0
    using re-complex[of z2/z1]
    by (auto simp add: complex-of-real-def[symmetric])
  thus scalprod z1 z2 = 0
    using assms complex-cnj-divide[of z2 z1]
    using add-frac-eq[of z1 cnj z1 z2 cnj z2]
    using divide-eq-0-iff[of z2 ∗ cnj z1 + cnj z2 ∗ z1  z1 ∗ cnj z1]
    by (auto simp add:field-simps)
next
  assume scalprod z1 z2 = 0
  hence z2 ∗ cnj z1 + cnj z2 ∗ z1 = 0
    by (simp add:field-simps)
  hence z2 / z1 + cnj (z2 / z1) = 0
    using assms complex-cnj-divide[of z2 z1]
    using add-frac-eq[of z1 cnj z1 z2 cnj z2]
    using divide-eq-0-iff[of z2 ∗ cnj z1 + cnj z2 ∗ z1  z1 ∗ cnj z1]
    by auto
  hence Re (z2 / z1) = 0
    using re-complex[of z2/z1]
    by auto
  have z2 / z1 ≠ 0
    using assms
    by auto
  hence arg (z2 / z1) = pi/2 ∨ arg (z2 / z1) = −pi/2
    using ‹Re (z2 / z1) = 0› re-complex-zero-arg2[of z2/z1]
    by auto
  have ⇂arg z2 − arg z1⇃ = arg (z2 / z1)
    using arg-div[of z2 z1] assms
    by auto
  thus c z1 z2 = pi / 2
    using ‹arg (z2 / z1) = pi/2 ∨ arg (z2 / z1) = −pi/2›
    unfolding ang-vec-c-def
    unfolding ang-vec-def
    by (metis abs-minus-cancel abs-of-nonneg minus-divide-left pi-half-ge-zero)
qed

lemma ortho-a-scalprod0:
  assumes z1 ≠ 0 z2 ≠ 0
  shows a z1 z2 = pi/2 ⟷ scalprod z1 z2 = 0
unfolding ang-vec-a-def
using assms ortho-c-scalprod0[of z1 z2]
by auto



lemma canon-ang-plus-pi1:
  assumes  z1 z2 > 0
  shows ⇂ z1 z2 + pi⇃ =  z1 z2 − pi
proof (rule canon-ang-eqI)
  show ∃ x::int.  z1 z2 − pi − ( z1 z2 + pi) = 2 ∗ real x ∗ pi
    by (rule-tac x=−1 in exI) auto
next
  show − pi <  z1 z2 − pi ∧  z1 z2 − pi ≤ pi
    using assms
    unfolding ang-vec-def
    using canon-ang(1)[of arg z2 − arg z1] canon-ang(2)[of arg z2 − arg z1]
    by auto
qed

lemma canon-ang-plus-pi2:
  assumes  z1 z2 ≤ 0
  shows ⇂ z1 z2 + pi⇃ =  z1 z2 + pi
proof (rule canon-ang-id)
  show − pi <  z1 z2 + pi ∧  z1 z2 + pi ≤ pi
    using assms
    unfolding ang-vec-def
    using canon-ang(1)[of arg z2 − arg z1] canon-ang(2)[of arg z2 − arg z1]
    by auto
qed


lemma ang-vec-opposite1:
  assumes z1 ≠ 0
  shows  (−z1) z2 = ⇂ z1 z2 − pi⇃
unfolding ang-vec-def
apply (subst arg-uminus[OF assms])
apply (subst canon-ang-arg[of z2, symmetric])
apply (subst canon-ang-diff[of arg z2 arg z1 + pi, symmetric])
apply (subst canon-ang-id[of pi, symmetric]) back
apply simp
apply (subst canon-ang-diff[of arg z2 − arg z1  pi, symmetric])
apply (simp add: field-simps)
done

lemma ang-vec-opposite2:
  assumes z2 ≠ 0
  shows  z1 (−z2) = ⇂ z1 z2 + pi⇃
unfolding ang-vec-def
using arg-mult[of −1 z2] assms
using arg-complex-of-real-negative[of −1]
using canon-ang-diff[of arg −1 + arg z2 arg z1]
using canon-ang-sum[of arg z2 − arg z1  pi]
using canon-ang-id[of pi] canon-ang-arg[of z1]
by auto (metis (hide-lams, no-types) ab-diff-minus ab-semigroup-add-class.add-ac(1)
minus-add minus-add-distrib minus-minus)

lemma ang-vec-opposite-opposite:
  assumes z1 ≠ 0 z2 ≠ 0
  shows  (−z1) (−z2) =  z1 z2
apply (subst ang-vec-opposite1[OF assms(1)])
apply (subst ang-vec-opposite2[OF assms(2)])
apply (subst canon-ang-id[of pi, symmetric]) back
apply simp
apply (subst canon-ang-diff[symmetric])
apply (simp del: ang-vec-def)
by (metis ang-vec-def canon-ang(1) canon-ang(2) canon-ang-id)

lemma ang-vec-a-opposite2:
  a z1 z2 = a z1 (−z2)
proof(cases z2 = 0)
  case True
  thus ?thesis
    by (metis minus-zero)
next
  case False
  thus ?thesis
  proof(cases  z1 z2 < −pi / 2)
    case True
    hence  z1 z2 < 0
      by auto (metis less-trans minus-pi-half-less-zero)
    have a z1 z2 = pi +  z1 z2
      using True ‹ z1 z2 < 0›
      unfolding ang-vec-a-def ang-vec-c-def ang-vec-a-def abs-real-def
      by auto
    moreover
    have a z1 (−z2) = pi +  z1 z2
      unfolding ang-vec-a-def ang-vec-c-def abs-real-def
      using canon-ang(1)[of arg z2 − arg z1] canon-ang(2)[of arg z2 − arg z1]
      using canon-ang-plus-pi2[of z1 z2] True ‹ z1 z2 < 0› ‹z2 ≠ 0›
      using ang-vec-opposite2[of z2 z1]
      by auto
    ultimately
    show ?thesis
      by auto
  next
    case False
    show ?thesis
    proof (cases  z1 z2 ≤ 0)
      case True
      have a z1 z2 = −  z1 z2
        using ‹¬  z1 z2 < − pi / 2› True
        unfolding ang-vec-a-def ang-vec-c-def ang-vec-a-def abs-real-def
        by auto
      moreover
      have a z1 (−z2) = −  z1 z2
        using ‹¬  z1 z2 < − pi / 2› True
        unfolding ang-vec-a-def ang-vec-c-def abs-real-def
        using canon-ang-plus-pi2[of z1 z2]
        using canon-ang(1)[of arg z2 − arg z1] canon-ang(2)[of arg z2 − arg z1]
        using ‹z2 ≠ 0› ang-vec-opposite2[of z2 z1]
        by auto
      ultimately
      show ?thesis
        by simp
    next
      case False
      show ?thesis
      proof (cases  z1 z2 < pi / 2)
        case True
        have a z1 z2 =  z1 z2
          using ‹¬  z1 z2 ≤ 0› True
          unfolding ang-vec-a-def ang-vec-c-def ang-vec-a-def abs-real-def
          by auto
        moreover
        have a z1 (−z2) =  z1 z2
          using ‹¬  z1 z2 ≤ 0› True
          unfolding ang-vec-a-def ang-vec-c-def abs-real-def
          using canon-ang-plus-pi1[of z1 z2]
          using canon-ang(1)[of arg z2 − arg z1] canon-ang(2)[of arg z2 − arg z1]
          using ‹z2 ≠ 0› ang-vec-opposite2[of z2 z1]
```

```
          by auto
        ultimately
        show ?thesis
          by simp
      next
        case False
        have  z1 z2 > 0
          using False
          by (metis less-linear less-trans pi-half-gt-zero)
        have a z1 z2 = pi −  z1 z2
          using False ‹ z1 z2 > 0›
          unfolding ang-vec-a-def ang-vec-c-def ang-vec-a-def abs-real-def
          by auto
        moreover
        have a z1 (−z2) = pi −  z1 z2
          unfolding ang-vec-a-def ang-vec-c-def abs-real-def
          using False ‹ z1 z2 > 0›
          using canon-ang-plus-pi1[of z1 z2]
          using canon-ang(1)[of arg z2 − arg z1] canon-ang(2)[of arg z2 − arg z1]
          using ‹z2 ≠ 0› ang-vec-opposite2[of z2 z1]
          by auto
        ultimately
        show ?thesis
          by auto
      qed
    qed
  qed
qed

lemma ang-vec-a-opposite1:
  a z1 z2 = a (−z1) z2
using ang-vec-a-sym[of −z1 z2] ang-vec-a-opposite2[of z2 z1] ang-vec-a-sym[of z2
z1]
by auto

lemma ang-vec-a-scale1:
  assumes k ≠ 0
  shows a (complex-of-real k ∗ z1) z2 = a z1 z2
proof (cases k > 0)
  case True
  thus ?thesis
    unfolding ang-vec-a-def ang-vec-c-def ang-vec-def
    using arg-mult-real-positive[of k z1]
    by auto
next
  case False
  hence k < 0
    using assms
    by auto
  thus ?thesis
    using arg-mult-real-negative[of k z1]
    using ang-vec-a-opposite1[of z1 z2]
    unfolding ang-vec-a-def ang-vec-c-def ang-vec-def
    by simp
qed

lemma ang-vec-a-scale2:
  assumes k ≠ 0
  shows a z1 (complex-of-real k ∗ z2) = a z1 z2
using ang-vec-a-sym[of z1 complex-of-real k ∗ z2]
using ang-vec-a-scale1[OF assms, of z2 z1]
using ang-vec-a-sym[of z1 z2]
by auto

lemma ang-vec-a-scale:
  assumes k1 ≠ 0 k2 ≠ 0
  shows a (complex-of-real k1 ∗ z1) (complex-of-real k2 ∗ z2) = a z1 z2
using ang-vec-a-scale1[OF assms(1)] ang-vec-a-scale2[OF assms(2)]
by auto

lemma ang-a-cnj-cnj:
  shows a z1 z2 = a (cnj z1) (cnj z2)
unfolding ang-vec-a-def ang-vec-c-def ang-vec-def
proof(cases arg z1 ≠ pi ∧ arg z2 ≠ pi)
  case True
  thus acute-ang |⇂arg z2 − arg z1⇃| = acute-ang |⇂arg (cnj z2) − arg (cnj z1)⇃|
    using arg-cnj2[of z1] arg-cnj2[of z2]
    apply (auto simp del:acute-ang-def)
    proof(cases ⇂arg z2 − arg z1⇃ = pi)
      case True
      thus acute-ang |⇂arg z2 − arg z1⇃| = acute-ang |⇂− arg z2 + arg z1⇃|
        using canon-ang-uminus-pi[of arg z2 − arg z1]
        by (auto simp add:field-simps del:acute-ang-def)
    next
      case False
      thus acute-ang |⇂arg z2 − arg z1⇃| = acute-ang |⇂− arg z2 + arg z1⇃|
        using canon-ang-uminus[of arg z2 − arg z1]
        by (auto simp add:field-simps del:acute-ang-def)
    qed
  next
    case False
    thus acute-ang |⇂arg z2 − arg z1⇃| = acute-ang |⇂arg (cnj z2) − arg (cnj z1)⇃|
    proof(cases arg z1 = pi)
      case False
      hence arg z2 = pi
        using ‹ ¬ (arg z1 ≠ pi ∧ arg z2 ≠ pi)›
        by auto
      thus ?thesis
```

```
       using False
       using arg-cnj2[of z1] arg-cnj1[of z2]
       apply (auto simp del:acute-ang-def)
     proof(cases arg z1 > 0)
         case True
         hence −arg z1 ≤ 0
           by auto
         thus acute-ang |⇂pi − arg z1⇃| = acute-ang |⇂pi + arg z1⇃|
           using True MoreComplex.canon-ang-plus-pi1[of arg z1]
           using arg-bounded[of z1] MoreComplex.canon-ang-plus-pi2[of −arg z1]
           by (auto simp add:field-simps del:acute-ang-def)
       next
         case False
         hence −arg z1 ≥ 0
            by simp
         thus acute-ang |⇂pi − arg z1⇃| = acute-ang |⇂pi + arg z1⇃|
         proof(cases arg z1 = 0)
           case True
           thus ?thesis
             by (auto simp del:acute-ang-def)
         next
           case False
           hence −arg z1 > 0
             using ‹−arg z1 ≥ 0›
             by auto
           thus ?thesis
           using False MoreComplex.canon-ang-plus-pi1[of −arg z1]
           using arg-bounded[of z1] MoreComplex.canon-ang-plus-pi2[of arg z1]
           by (auto simp add:field-simps del:acute-ang-def)
       qed
     qed
   next
     case True
     thus ?thesis
       using arg-cnj1[of z1]
       apply (auto simp del:acute-ang-def)
     proof(cases arg z2 = pi)
       case True
       thus acute-ang |⇂arg z2 − pi⇃| = acute-ang |⇂arg (cnj z2) − pi⇃|
         using arg-cnj1[of z2]
         by auto
     next
       case False
       thus acute-ang |⇂arg z2 − pi⇃| = acute-ang |⇂arg (cnj z2) − pi⇃|
using arg-cnj2[of z2]
         apply (auto simp del:acute-ang-def)
       proof(cases arg z2 > 0)
         case True
         hence −arg z2 ≤ 0
```

```
            by auto
          thus acute-ang |⇂arg z2 − pi⇃| = acute-ang |⇂− arg z2 − pi⇃|
            using True canon-ang-minus-pi1[of arg z2]
            using arg-bounded[of z2] canon-ang-minus-pi2[of −arg z2]
            by (auto simp add:field-simps del:acute-ang-def)
        next
          case False
          hence −arg z2 ≥ 0
             by simp
          thus acute-ang |⇂arg z2 − pi⇃| = acute-ang |⇂− arg z2 − pi⇃|
          proof(cases arg z2 = 0)
            case True
            thus ?thesis
              by (auto simp del:acute-ang-def)
          next
            case False
            hence −arg z2 > 0
              using ‹−arg z2 ≥ 0›
              by auto
            thus ?thesis
            using False canon-ang-minus-pi1[of −arg z2]
            using arg-bounded[of z2] canon-ang-minus-pi2[of arg z2]
            by (auto simp add:field-simps del:acute-ang-def)
        qed
      qed
    qed
  qed
qed

abbreviation sgn-bool where
  sgn-bool p ≡ if p then 1 else −1


definition circ-tang-vec :: complex ⇒ complex ⇒ bool ⇒ complex where
  circ-tang-vec μ E p = sgn-bool p ∗ ii ∗ (E − μ)

lemma circ-tang-vec-ortho:
  scalprod (E − μ) (circ-tang-vec μ E p) = 0
unfolding circ-tang-vec-def Let-def
by (auto simp add: complex-cnj-mult)

lemma circ-tang-vec-opposite-orient:
  circ-tang-vec μ E p = − circ-tang-vec μ E (¬ p)
unfolding circ-tang-vec-def
by auto

definition ang-circ where
  ang-circ E μ1 μ2 p1 p2 =  (circ-tang-vec μ1 E p1) (circ-tang-vec μ2 E p2)
```

```
definition ang-circ-c where
  ang-circ-c E μ1 μ2 p1 p2 = c (circ-tang-vec μ1 E p1) (circ-tang-vec μ2 E p2)

definition ang-circ-a where
  ang-circ-a E μ1 μ2 p1 p2 = a (circ-tang-vec μ1 E p1) (circ-tang-vec μ2 E p2)

lemma ang-circ-simp:
  assumes E ≠ μ1 E ≠ μ2
  showsang-circ E μ1 μ2 p1 p2 = canon-ang (arg (E − μ2) − arg (E − μ1) +
sgn-bool p1 ∗ pi / 2 − sgn-bool p2 ∗ pi / 2)
unfolding ang-circ-def ang-vec-def circ-tang-vec-def
apply (rule canon-ang-eq)
using assms
using arg-mult-2kpi[of sgn-bool p2∗ii E − μ2]
using arg-mult-2kpi[of sgn-bool p1∗ii E − μ1]
apply auto
apply (rule-tac x=x−xa in exI, auto simp add: field-simps)
apply (rule-tac x=−1+x−xa in exI, auto simp add: field-simps)
apply (rule-tac x=1+x−xa in exI, auto simp add: field-simps)
apply (rule-tac x=x−xa in exI, auto simp add: field-simps)
done

lemma ang-circ-c-simp:
  assumes E ≠ μ1 E ≠ μ2
  shows ang-circ-c E μ1 μ2 p1 p2 = abs (canon-ang (arg(E − μ2) − arg(E −
μ1) + (sgn-bool p1) ∗ pi/2 − (sgn-bool p2) ∗ pi/2))
  unfolding ang-circ-c-def ang-vec-c-def
using ang-circ-simp[OF assms]
unfolding ang-circ-def
by auto

lemma ang-circ-a-simp:
  assumes E ≠ μ1 E ≠ μ2
  shows ang-circ-a E μ1 μ2 p1 p2 = acute-ang (abs (canon-ang (arg(E − μ2) −
arg(E − μ1) + (sgn-bool p1) ∗ pi/2 − (sgn-bool p2) ∗ pi/2)))
unfolding ang-circ-a-def ang-vec-a-def
using ang-circ-c-simp[OF assms]
unfolding ang-circ-c-def
by auto

lemma ang-circ-a-pTrue:
  assumes E ≠ μ1 E ≠ μ2
  shows ang-circ-a E μ1 μ2 p1 p2 = ang-circ-a E μ1 μ2 True True
proof (cases p1)
  case True
  show ?thesis
  proof (cases p2)
    case True
    show ?thesis
```

```
      using ‹p1› ‹p2›
      by simp
  next
    case False
    show ?thesis
      using ‹p1› ‹¬ p2›
      unfolding ang-circ-a-def
      using circ-tang-vec-opposite-orient[of μ2 E p2]
      using ang-vec-a-opposite2
      by simp
  qed
next
  case False
  show ?thesis
  proof (cases p2)
    case True
    show ?thesis
      using ‹¬ p1› ‹p2›
      unfolding ang-circ-a-def
      using circ-tang-vec-opposite-orient[of μ1 E p1]
      using ang-vec-a-opposite1
      by simp
  next
    case False
    show ?thesis
      using ‹¬ p1› ‹¬ p2›
      unfolding ang-circ-a-def
     using circ-tang-vec-opposite-orient[of μ1 E p1] circ-tang-vec-opposite-orient[of
μ2 E p2]
      using ang-vec-a-opposite1  ang-vec-a-opposite2
      by simp
  qed
qed

lemma ang-circ-a-simp1:
  assumes E ≠ μ1 E ≠ μ2
  shows ang-circ-a E μ1 μ2 p1 p2 = a (E − μ1) (E − μ2)
unfolding ang-vec-a-def ang-vec-c-def ang-vec-def
by (subst ang-circ-a-pTrue[OF assms, of p1 p2], subst ang-circ-a-simp[OF assms,
of True True]) (metis add-diff-cancel)

abbreviation ang-circ-a′ where
  ang-circ-a′ E μ1 μ2 ≡ ang-circ-a E μ1 μ2 True True

lemma ang-circ-a′-simp:
  assumes z ≠ μ1 z ≠ μ2
  shows ang-circ-a′ z μ1 μ2 = a (z − μ1) (z − μ2)
by (rule ang-circ-a-simp1[OF assms])
```

```
lemma cos-cmod-scalprod:
  shows cmod b * cmod c * (cos ( b c)) = Re (scalprod b c)
proof (cases b = 0 ∨ c = 0)
  case True
  thus ?thesis
    by auto
next
  case False
  thus ?thesis
    by (simp add: cos-diff cos-arg sin-arg field-simps)
qed

lemma law-of-cosines:
  shows (cdist B C)^2 = (cdist A C)^2 + (cdist A B)^2 − 2*(cdist A C)*(cdist A
B)*(cos ( (C−A) (B−A)))
proof−
  let ?a = C−B and ?b = C−A and ?c = B−A
  have ?a = ?b − ?c
    by simp
  hence (cmod ?a)^2 = (cmod (?b − ?c))^2
    by metis
  also have ... = Re (scalprod (?b−?c) (?b−?c))
    by (simp add: cmod-square)
  also have ... = (cmod ?b)^2 + (cmod ?c)^2 − 2*Re (scalprod ?b ?c)
    by (simp add: cmod-square field-simps)
  finally
  show ?thesis
    using cos-cmod-scalprod[of ?b ?c]
    by simp
qed

declare ang-vec-c-def[simp del]

lemma cos-c-: cos (c z1 z2) = cos ( z1 z2)
unfolding ang-vec-c-def
by (smt cos-minus)

lemma cos-a-c: cos (a z1 z2) = abs (cos (c z1 z2))
unfolding ang-vec-a-def
using ang-vec-c-bounded[of z1 z2] cos-lt-zero[of c z1 z2] cos-gt-zero-pi[of c z1 z2]
by (cases c z1 z2 = pi/2) (auto, smt cos-minus cos-periodic-pi3)

end
```

8 Homogeneous coordinates in extended complex plane

```
theory HomogeneousCoordinates
imports MoreComplex Matrices
begin

typedef homo-coords = {v. v ≠ vec-zero}
by (rule-tac x=(1, 0) in exI, simp)

lemma obtain-homo-coords:
  fixes x::homo-coords
  obtains A B where
  Rep-homo-coords x = (A, B) A ≠ 0 ∨ B ≠ 0
by (cases x) (auto simp add: Abs-homo-coords-inverse)

definition homo-coords-eq :: homo-coords ⇒ homo-coords ⇒ bool (infix ≈ 50)
where
 [simp]: z1 ≈ z2 ⟷
     (let z1 = Rep-homo-coords z1;
          z2 = Rep-homo-coords z2
       in (∃ k. k ≠ (0::complex) ∧ z2 = k *sv z1))

lemma homo-coords-eq-reflp:
  reflp homo-coords-eq
by (auto simp add: reflp-def, rule-tac x=1 in exI, simp)

lemma homo-coords-eq-symp:
  symp homo-coords-eq
by (auto simp add: symp-def, rule-tac x=1/k in exI, simp)

lemma homo-coords-eq-transp:
transp homo-coords-eq
by (auto simp add: transp-def, rule-tac x=ka*k in exI, simp)

lemma homo-coords-eq-equivp:
  equivp homo-coords-eq
 by (auto intro: equivpI homo-coords-eq-reflp homo-coords-eq-symp homo-coords-eq-transp)

lemma homo-coords-eq-refl [simp]:
  z ≈ z
using homo-coords-eq-reflp
by (auto simp add: reflp-def refl-on-def)

lemma homo-coords-eq-trans:
  assumes z1 ≈ z2  z2 ≈ z3
  shows z1 ≈ z3
using assms homo-coords-eq-transp
unfolding transp-def
```

by *blast*

lemma *homo-coords-eq-sym*:
  assumes *z1* ≈ *z2*
  shows *z2* ≈ *z1*
using *assms homo-coords-eq-symp*
unfolding *symp-def*
by *blast*

lemma *homo-coords-eq-mix*:
  assumes *Rep-homo-coords z1* = (*z1'*, *z1''*) *Rep-homo-coords z2* = (*z2'*, *z2''*)
  shows *z1* ≈ *z2* ⟷ *z2'*∗*z1''* = *z1'*∗*z2''*
using *assms*
proof (*cases z1''* ≠ *0* ∨ *z2''* ≠ *0*)
  case *False*
  thus *?thesis*
    using *assms* **using** *Rep-homo-coords*[*of z1*] *Rep-homo-coords*[*of z2*]
    by *auto*
next
  case *True*
  thus *?thesis*
    using *assms*
    apply *auto*
    apply (*rule-tac x*=*z2''*/*z1''* **in** *exI*)
    using *Rep-homo-coords*[*of z2*]
    apply (*auto simp add*: *field-simps*)
    apply (*rule-tac x*=*z2''*/*z1''* **in** *exI*)
    using *Rep-homo-coords*[*of z1*]
    apply (*auto simp add*: *field-simps*)
    done
qed

lemma [*simp*]: *Rep-homo-coords* (*Abs-homo-coords* (*Rep-homo-coords x*)) = *Rep-homo-coords x*
using *Rep-homo-coords*[*of x*]
by (*simp add*: *Abs-homo-coords-inverse*)

Quotient of homogeneous coordinates

quotient-type
  complex-homo = *homo-coords* / *homo-coords-eq*
by (*rule homo-coords-eq-equivp*)

Infinite point

definition *inf-homo-rep* **where** [*simp*]: *inf-homo-rep* = *Abs-homo-coords* (*1*, *0*)
lift-definition *inf-homo* :: *complex-homo* (∞_h) **is** *inf-homo-rep*
done

lemma [*simp*]: *Rep-homo-coords* (*Abs-homo-coords* (*1*, *0*)) = (*1*, *0*)
by (*simp add*: *Abs-homo-coords-inverse*)

lemma [*simp*]: *Rep-homo-coords inf-homo-rep* = (*1*, *0*)
by *simp*

lemma *inf-snd-0*: $z \approx$ *inf-homo-rep* $\longleftrightarrow$ (*let* (*z1*, *z2*) = *Rep-homo-coords z in z1* $\neq 0 \wedge z2 = 0$)
using *Rep-homo-coords*[*of z*]
by *auto*

lemma *not-inf-snd-not0*:
  assumes $\neg$ *z* $\approx$ *inf-homo-rep*
  shows *let* (*z1*, *z2*) = *Rep-homo-coords z in z2* $\neq 0$
using *assms Rep-homo-coords*[*of z*] *inf-snd-0*[*of z*]
by *auto*

Zero

definition *zero-homo-rep* **where** [*simp*]: *zero-homo-rep* = *Abs-homo-coords* (*0*, *1*)
lift-definition *zero-homo* :: *complex-homo* (0_h) **is** *zero-homo-rep*
done

lemma [*simp*]: *Rep-homo-coords* (*Abs-homo-coords* (*0*, *1*)) = (*0*, *1*)
by (*simp add*: *Abs-homo-coords-inverse*)

lemma [*simp*]: *Rep-homo-coords zero-homo-rep* = (*0*, *1*)
by *simp*

lemma *zero-fst-0*: $z \approx$ *zero-homo-rep* $\longleftrightarrow$ (*let* (*z1*, *z2*) = *Rep-homo-coords z in* $z1 = 0 \wedge z2 \neq 0$)
using *Rep-homo-coords*[*of z*]
by *auto*

One

definition *one-homo-rep* **where** [*simp*]: *one-homo-rep* = *Abs-homo-coords* (*1*, *1*)
lift-definition *one-homo* :: *complex-homo* (1_h) **is** *one-homo-rep*
done

lemma [*simp*]: *Rep-homo-coords* (*Abs-homo-coords* (*1*, *1*)) = (*1*, *1*)
by (*simp add*: *Abs-homo-coords-inverse*)

lemma [*simp*]: *Rep-homo-coords one-homo-rep* = (*1*, *1*)
by *simp*

lemma [*simp*]: $1_h \neq \infty_h$ $0_h \neq \infty_h$ $0_h \neq 1_h$ $1_h \neq 0_h$ $\infty_h \neq 0_h$ $\infty_h \neq 1_h$
by (*transfer*, *auto*)+

definition *ii-homo-rep* **where** *ii-homo-rep* = *Abs-homo-coords* (*ii*, *1*)

```
lift-definition ii-homo :: complex-homo  (ii_h)  is ii-homo-rep
done

lemma [simp]: Rep-homo-coords (Abs-homo-coords (ii, 1)) = (ii, 1)
  by (simp add: Abs-homo-coords-inverse)

lemma [simp]: Rep-homo-coords ii-homo-rep = (ii, 1)
  by (simp add: ii-homo-rep-def)



lemma ex-3-different-points:
  fixes z::complex-homo
  shows ∃ z1 z2. z ≠ z1 ∧ z1 ≠ z2 ∧ z ≠ z2
proof (cases z ≠ 0_h ∧ z ≠ 1_h)
  case True
  thus ?thesis
    by (rule-tac x=0_h in exI, rule-tac x=1_h in exI, auto)
next
  case False
  hence z = 0_h ∨ z = 1_h
    by simp
  thus ?thesis
  proof
    assume z = 0_h
    thus ?thesis
      by (rule-tac x=∞_h in exI, rule-tac x=1_h in exI, auto)
  next
    assume z = 1_h
    thus ?thesis
      by (rule-tac x=∞_h in exI, rule-tac x=0_h in exI, auto)
  qed
qed
```

Conversion from complex

```
definition of-complex-coords where
  of-complex-coords z = Abs-homo-coords (z, 1)

lemma [simp]: Rep-homo-coords (of-complex-coords z) = (z, 1)
by (simp add: of-complex-coords-def Abs-homo-coords-inverse)

lift-definition of-complex :: complex ⇒ complex-homo is of-complex-coords
by (simp del: homo-coords-eq-def)

lemma of-complex-inj:
  assumes of-complex x = of-complex y
  shows x = y
using assms
by transfer simp
```

lemma *of-complex-image-inj*:
  assumes *of-complex ‘ A = of-complex ‘ B*
  shows *A = B*
using *assms*
using *of-complex-inj*
by *auto*

lemma [*simp*]: *of-complex x* ≠ ∞_h
by *transfer simp*

lemma [*simp*]: ∞_h ≠ *of-complex x*
by *transfer simp*

lemma *inf-homo-or-complex-homo*:
   z = ∞_h ∨ (∃ *x*. *z* = *of-complex x*)
proof(*transfer*)
  fix *z*
  obtain *a b* **where** ∗: *Rep-homo-coords z* = (*a*, *b*)
    by (*rule obtain-homo-coords*)
  show *z* ≈ *inf-homo-rep* ∨ (∃ *x*. *z* ≈ *of-complex-coords x*)
    using ∗ *Rep-homo-coords*[*of z*]
    by (*cases b* = *0*) *auto*
qed

lemma *zero-of-complex* [*simp*]: *of-complex 0* = 0_h
by *transfer simp*

lemma *one-of-complex* [*simp*]: *of-complex 1* = 1_h
by *transfer simp*

lemma
  [*simp*]: *of-complex a* = 0_h ⟷ *a* = *0*
by (*subst zero-of-complex*[*symmetric*]) (*auto simp add*: *of-complex-inj*)

lemma
  [*simp*]: *of-complex a* = 1_h ⟷ *a* = *1*
by (*subst one-of-complex*[*symmetric*]) (*auto simp add*: *of-complex-inj*)

Coercion to complex

definition *to-complex-homo-coords* :: *homo-coords* ⇒ *complex* **where**
  to-complex-homo-coords z = (*let* (*z1*, *z2*) = *Rep-homo-coords z in z1*/*z2*)

lift-definition *to-complex* :: *complex-homo* ⇒ *complex* **is** *to-complex-homo-coords*
proof−
  fix *x y*
  assume *x* ≈ *y*
  thus *to-complex-homo-coords x* = *to-complex-homo-coords y*
    by (*auto simp add*: *to-complex-homo-coords-def split-def Let-def*)

```
qed

lemma [simp]: to-complex (of-complex z) = z
by (transfer) (simp add: of-complex-coords-def to-complex-homo-coords-def Abs-homo-coords-inverse)

lemma [simp]: z ≠ ∞h ⟹ (of-complex (to-complex z)) = z
proof (transfer)
  fix z
  obtain z1 z2 where zz: Rep-homo-coords z = (z1, z2)
    by (rule obtain-homo-coords)
  assume ¬ z ≈ inf-homo-rep
  hence z2 ≠ 0
    using zz Rep-homo-coords[of z]
    by auto (erule-tac x=1/z1 in allE, simp)
  thus of-complex-coords (to-complex-homo-coords z) ≈ z
    using zz
   by (auto simp add: of-complex-coords-def to-complex-homo-coords-def Abs-homo-coords-inverse)
qed
```

Addition

```
definition add-homo-coords :: homo-coords ⇒ homo-coords ⇒ homo-coords (infixl
+hc 100) where
  z +hc w = (let (z1, z2) = Rep-homo-coords z;
                 (w1, w2) = Rep-homo-coords w in
      Abs-homo-coords (z1∗w2 + w1∗z2, z2∗w2))

lemma add-homo-coords-Rep:
  assumes Rep-homo-coords z = (z1, z2) Rep-homo-coords w = (w1, w2) z2 ≠ 0
∨ w2 ≠ 0
  shows Rep-homo-coords (z +hc w) = (z1∗w2 + w1∗z2, z2∗w2)
proof−
  from assms
  have (z1∗w2 + w1∗z2, z2∗w2) ≠ vec-zero
    using Rep-homo-coords[of z] Rep-homo-coords[of w]
    by auto
  thus ?thesis
    using assms(1−2)
   by (auto simp add: add-homo-coords-def split-def Let-def Abs-homo-coords-inverse)
qed

lemma add-homo-coords-00:
  assumes Rep-homo-coords z = (z1, z2) Rep-homo-coords w = (w1, w2) z2 = 0
w2 = 0
  shows z +hc w = Abs-homo-coords (0, 0)
using assms unfolding add-homo-coords-def
by simp

lemma add-coords-well-defined-lemma:
  assumes x ≈ y x′ ≈ y′
  shows x +hc x' ≈ y +hc y'
using assms
proof−
  obtain Ax Bx where xx: Rep-homo-coords x = (Ax, Bx)
    by (rule obtain-homo-coords)
  obtain Ax' Bx' where xx': Rep-homo-coords x' = (Ax', Bx')
    by (rule obtain-homo-coords)
  obtain Ay By where yy: Rep-homo-coords y = (Ay, By)
    by (rule obtain-homo-coords)
  obtain Ay' By' where yy': Rep-homo-coords y' = (Ay', By')
    by (rule obtain-homo-coords)
  from assms obtain k k' where
    ∗: k ≠ 0 Ay = k∗Ax By = k∗Bx k' ≠ 0 Ay' = k'∗Ax' By' = k'∗Bx'
    using xx xx' yy yy'
    by auto
  show ?thesis
  proof (cases Bx = 0 ∧ Bx' = 0)
    case True
    thus ?thesis
      using add-homo-coords-00[of x Ax 0 x' Ax' 0] add-homo-coords-00[of y Ay 0
y' Ay' 0] xx yy xx' yy' ∗
      by (auto, rule-tac x=1 in exI, simp)
  next
    case False
    thus ?thesis
      using xx xx' yy yy' ∗
      using Rep-homo-coords[of x] Rep-homo-coords[of x'] ‹k ≠ 0› ‹k' ≠ 0›
      using add-homo-coords-Rep[of x Ax Bx x' Ax' Bx'] add-homo-coords-Rep[of
y k ∗ Ax k ∗ Bx y' k' ∗ Ax' k' ∗ Bx']
      by simp (rule-tac x=k∗k' in exI, auto simp add: field-simps)
  qed
qed

lift-definition add-homo :: complex-homo ⇒ complex-homo ⇒ complex-homo (infixl
+h 100) is add-homo-coords
by (rule add-coords-well-defined-lemma, simp-all)

lemma add-homo-commute: x +h y = y +h x
proof (transfer)
  fix x y
  obtain Ax Bx where xx: Rep-homo-coords x = (Ax, Bx)
    by (rule obtain-homo-coords)
  obtain Ay By where yy: Rep-homo-coords y = (Ay, By)
    by (rule obtain-homo-coords)

  show x +hc y ≈ y +hc x
  proof (cases Bx ≠ 0 ∨ By ≠ 0)
    case True
    thus ?thesis
```

using *add-homo-coords-Rep*[*of x Ax Bx y Ay By*, *OF xx yy*]
using *add-homo-coords-Rep*[*of y Ay By x Ax Bx*, *OF yy xx*]
by *auto* (*rule-tac x=1* **in** *exI*, *simp*)+
next
case *False*
thus *?thesis*
using *xx yy add-homo-coords-00*
by (*auto*, *rule-tac x=1* **in** *exI*, *simp*)
qed
qed

lemma *of-complex-add*: (*of-complex za*) $+_h$ (*of-complex zb*) = *of-complex* (*za* + *zb*)
proof (*transfer*)
fix *za zb*
have *Rep-homo-coords* (*Abs-homo-coords* (*za*, *1*)) = (*za*, *1*) *Rep-homo-coords* (*Abs-homo-coords* (*zb*, *1*)) = (*zb*, *1*)
by (*auto simp add*: *Abs-homo-coords-inverse*)
thus *of-complex-coords za* $+_{hc}$ *of-complex-coords zb* ≈ *of-complex-coords* (*za* + *zb*)
unfolding *of-complex-coords-def*
using *add-homo-coords-Rep*[*of Abs-homo-coords* (*za*, *1*) *za 1 Abs-homo-coords* (*zb*, *1*) *zb 1*]
by (*simp add*: *Abs-homo-coords-inverse*)
qed

lemma [*simp*]: (*of-complex z*) $+_h$ ∞_h = ∞_h
proof (*transfer*)
fix *z*
show *of-complex-coords z* $+_{hc}$ *inf-homo-rep* ≈ *inf-homo-rep*
using *add-homo-coords-Rep*[*of Abs-homo-coords* (*z*, *1*) *z 1 Abs-homo-coords* (*1*, *0*) *1 0*]
unfolding *of-complex-coords-def*
by (*simp add*: *Abs-homo-coords-inverse*)
qed

lemma [*simp*]: ∞_h $+_h$ (*of-complex z*) = ∞_h
by (*subst add-homo-commute*) *simp*

lemma *add-homo-zero-right* [*simp*]: *z* $+_h$ 0_h = *z*
proof (*transfer*)
fix *z*
obtain *z1 z2* **where** *zz*: *Rep-homo-coords z* = (*z1*, *z2*)
by (*rule obtain-homo-coords*)
thus *z* $+_{hc}$ *zero-homo-rep* ≈ *z*
using *add-homo-coords-Rep*[*of z z1 z2 zero-homo-rep 0 1*]
by *auto* (*metis zero-neq-one*)
qed

lemma *add-homo-zero-left* [*simp*]: $0_h +_h z = z$
  by (*subst add-homo-commute*) *simp*

uminus

definition *uminus-homo-coords* **where**
  uminus-homo-coords z = (*let* (*z1*, *z2*) = *Rep-homo-coords z in Abs-homo-coords*
(−*z1*, *z2*))

lemma *uminus-homo-coords-Rep* [*simp*]: *Rep-homo-coords* (*uminus-homo-coords*
z) = (*let* (*z1*, *z2*) = *Rep-homo-coords z in* (−*z1*, *z2*))
unfolding *uminus-homo-coords-def Let-def*
apply (*cases Rep-homo-coords z*)
using *Rep-homo-coords*[*of z*]
by (*auto simp add*: *Abs-homo-coords-inverse*)

lift-definition *uminus-homo* :: *complex-homo* ⇒ *complex-homo* **is** *uminus-homo-coords*
by (*auto simp add*: *split-def Let-def*)

lemma *of-complex-uminus* [*simp*]: *uminus-homo* (*of-complex z*) = *of-complex* (−*z*)
by (*transfer*) *auto*

Subtraction

definition *minus-homo* :: *complex-homo* ⇒ *complex-homo* ⇒ *complex-homo* (**infixl**
$-_h$ *100*) **where**
  $z1 -_h z2 = z1 +_h$ (*uminus-homo z2*)

lemma *minus-homo-coords-Rep*:
  assumes *Rep-homo-coords z* = (*z1*, *z2*) *Rep-homo-coords w* = (*w1*, *w2*) $z2 \neq 0$
$\vee$ $w2 \neq 0$
  shows *Rep-homo-coords* ($z +_{hc}$ (*uminus-homo-coords w*)) = ($z1{*}w2 - w1{*}z2$,
$z2{*}w2$)
using *assms*
using *add-homo-coords-Rep*[*of z z1 z2 uminus-homo-coords w* −*w1 w2*] *uminus-homo-coords-Rep*[*of*
w]
by *simp*

lemma *of-complex-minus*:
  (*of-complex z1*) $-_h$ (*of-complex z2*) = *of-complex* (*z1* − *z2*)
unfolding *minus-homo-def complex-diff-def*
by (*simp add*: *of-complex-add*)

lemma [*simp*]:
  assumes $z \neq \infty_h$
  shows $z -_h z = 0_h$
proof−
  from *assms* **obtain** z' **where** z = *of-complex* z'
    using *inf-homo-or-complex-homo*[*of z*]
    by *auto*
  thus *?thesis*

```
    by (simp add: of-complex-minus)
qed

lemma diff-zero-homo:
  assumes z1 −_h z2 = 0_h z1 ≠ ∞_h ∨ z2 ≠ ∞_h
  shows z1 = z2
using assms
unfolding minus-homo-def
proof transfer
  fix z w
  obtain z1 z2 where zz: Rep-homo-coords z = (z1, z2)
    by (rule obtain-homo-coords)
  obtain w1 w2 where ww: Rep-homo-coords w = (w1, w2)
    by (rule obtain-homo-coords)
  have mww: Rep-homo-coords (uminus-homo-coords w) = (−w1, w2)
    using ww
    by simp
  assume ∗: z +_hc uminus-homo-coords w ≈ zero-homo-rep  and
         ¬ z ≈ inf-homo-rep ∨ ¬ w ≈ inf-homo-rep
  have z2 ≠ 0 ∨ w2 ≠ 0
    using Rep-homo-coords[of z] Rep-homo-coords[of w]
    using ‹¬ z ≈ inf-homo-rep ∨ ¬ w ≈ inf-homo-rep›
    using inf-snd-0[of z] inf-snd-0[of w] zz ww
    by auto
  thus z ≈ w
    using ∗ zz ww
    apply simp
    apply (subst (asm) minus-homo-coords-Rep[of z z1 z2 w w1 w2])
    apply auto
    apply (rule-tac x=w2/z2 in exI, auto simp add: field-simps)
    apply (rule-tac x=w2/z2 in exI, auto)
    done
qed
```

Multiplication

```
definition mult-homo-coords :: homo-coords ⇒ homo-coords ⇒ homo-coords (infixl
∗_hc 100) where
  x ∗_hc y = (let (x1, y1) = Rep-homo-coords x;
                  (x2, y2) = Rep-homo-coords y in
     Abs-homo-coords (x1∗x2, y1∗y2))

lemma mult-homo-coords-Rep:
  assumes Rep-homo-coords x = (Ax, Bx) Rep-homo-coords x' = (Ax', Bx') (Bx
≠ 0 ∨ Ax' ≠ 0) ∧ (Bx' ≠ 0 ∨ Ax ≠ 0)
  shows Rep-homo-coords (x ∗_hc x') = (Ax∗Ax', Bx∗Bx')
using assms Rep-homo-coords[of x] Rep-homo-coords[of x']
by (auto simp add: mult-homo-coords-def split-def Let-def Abs-homo-coords-inverse)

lemma mult-homo-coords-00:
```

assumes *Rep-homo-coords* x = (Ax, Bx) *Rep-homo-coords* x' = (Ax', Bx') (Bx = *0* $\wedge$ Ax' = *0*) $\vee$ (Bx' = *0* $\wedge$ Ax = *0*)
shows $x *_{hc} x'$ = *Abs-homo-coords* (*0*, *0*)
using *assms* **unfolding** *mult-homo-coords-def*
by *auto*

lemma *mult-coords-well-defined-lemma*:
assumes $x \approx y$ $x' \approx y'$
shows $x *_{hc} x' \approx y *_{hc} y'$
proof−
obtain *Ax Bx* **where** *xx*: *Rep-homo-coords* x = (Ax, Bx)
by (*rule obtain-homo-coords*)
obtain Ax' Bx' **where** *xx'*: *Rep-homo-coords* x' = (Ax', Bx')
by (*rule obtain-homo-coords*)
obtain *Ay By* **where** *yy*: *Rep-homo-coords* y = (Ay, By)
by (*rule obtain-homo-coords*)
obtain Ay' By' **where** *yy'*: *Rep-homo-coords* y' = (Ay', By')
by (*rule obtain-homo-coords*)
from *assms* **obtain** k k' **where**
∗: $k \neq$ *0* $Ay = k*Ax$ $By = k*Bx$ $k' \neq$ *0* $Ay' = k'*Ax'$ $By' = k'*Bx'$
using *xx xx' yy yy'*
by *auto*
show *?thesis*
proof (*cases* ($Bx \neq$ *0* $\vee$ $Ax' \neq$ *0*) $\wedge$ ($Bx' \neq$ *0* $\vee$ $Ax \neq$ *0*))
case *False*
thus *?thesis*
using *mult-homo-coords-00*[*of x Ax Bx x' Ax' Bx'*] *mult-homo-coords-00*[*of y Ay By y' Ay' By'*] *xx yy xx' yy'* ∗
by *auto* (*rule-tac x=1* **in** *exI*, *simp*)+
next
case *True*
thus *?thesis*
using *xx xx' yy yy'* ∗
using *Rep-homo-coords*[*of x*] *Rep-homo-coords*[*of x'*] ‹$k \neq$ *0*› ‹$k' \neq$ *0*›
using *mult-homo-coords-Rep*[*of x Ax Bx x' Ax' Bx'*]
mult-homo-coords-Rep[*of y k* ∗ *Ax k* ∗ *Bx y' k'* ∗ *Ax' k'* ∗ *Bx'*]
by *simp* (*rule-tac x=k∗k'* **in** *exI*, *auto simp add*: *field-simps*)
qed
qed

lift-definition *mult-homo* :: *complex-homo* ⇒ *complex-homo* ⇒ *complex-homo* (**infixl** $*_h$ *100*) **is** *mult-homo-coords*
by (*rule mult-coords-well-defined-lemma*, *simp-all*)

lemma *mult-of-complex*:
shows (*of-complex z1*) $*_h$ (*of-complex z2*) = *of-complex* (*z1* ∗ *z2*)
proof (*transfer*)
fix *z1 z2*
show *of-complex-coords z1* $*_{hc}$ *of-complex-coords z2* $\approx$ *of-complex-coords* (*z1* ∗

```
z2)
    using mult-homo-coords-Rep[of of-complex-coords z1 - - of-complex-coords z2]
    by simp
qed

lemma mult-homo-commute:
  shows z1 *_h z2 = z2 *_h z1
proof transfer
  fix z1 z2
  obtain z11 z12 where z1: Rep-homo-coords z1 = (z11, z12)
    by (rule obtain-homo-coords)
  obtain z21 z22 where z2: Rep-homo-coords z2 = (z21, z22)
    by (rule obtain-homo-coords)
  show z1 *_hc z2 ≈ z2 *_hc z1
  proof (cases (z12 ≠ 0 ∨ z21 ≠ 0) ∧ (z22 ≠ 0 ∨ z11 ≠ 0))
    case True
    thus ?thesis
      using mult-homo-coords-Rep[of z1 z11 z12 z2 z21 z22] z1 z2
      using mult-homo-coords-Rep[of z2 z21 z22 z1 z11 z12]
      by simp (rule-tac x=1 in exI, simp)
  next
    case False
    thus ?thesis
      using mult-homo-coords-00[of z1 z11 z12 z2 z21 z22] z1 z2
      using mult-homo-coords-00[of z2 z21 z22 z1 z11 z12]
      by auto (rule-tac x=1 in exI, simp)+
  qed
qed

lemma mult-homo-zero-left [simp]:
  assumes z ≠ ∞_h
  shows 0_h *_h z = 0_h
using assms
proof−
  obtain z' where z = of-complex z'
    using inf-homo-or-complex-homo[of z] assms
    by auto
  thus ?thesis
    using zero-of-complex
    using mult-of-complex[of 0 z']
    by simp
qed

lemma mult-homo-zero-right [simp]:
  assumes z ≠ ∞_h
  shows z *_h 0_h = 0_h
using mult-homo-zero-left[OF assms]
by (simp add: mult-homo-commute)
```

lemma *mult-homo-inf-right* [*simp*]:
assumes $z \neq 0_h$
shows $z *_h \infty_h = \infty_h$
using *assms*
proof (*transfer*)
fix *z*
obtain *z1 z2* **where** *zz*: *Rep-homo-coords z* = (*z1*, *z2*)
by (*rule obtain-homo-coords*)
assume ¬ *z* ≈ *zero-homo-rep*
hence *z1* ≠ *0*
using *Rep-homo-coords*[*of z*] *zz*
by *auto* (*metis divide-self-if eq-divide-eq mult-divide-mult-cancel-right*)
thus $z *_{hc}$ *inf-homo-rep* ≈ *inf-homo-rep*
using *zz mult-homo-coords-Rep*[*of z z1 z2 Abs-homo-coords* (*1*, *0*) *1 0*]
by *auto*
qed

lemma *mult-homo-inf-left* [*simp*]:
assumes $z \neq 0_h$
shows $\infty_h *_h z = \infty_h$
using *mult-homo-inf-right*[*OF assms*]
by (*simp add*: *mult-homo-commute*)

lemma *mult-homo-one-left* [*simp*]:
shows $1_h *_h z = z$
proof (*transfer*)
fix *z*
obtain *z1 z2* **where** *Rep-homo-coords z* = (*z1*, *z2*)
by (*rule obtain-homo-coords*)
thus *one-homo-rep* $*_{hc}$ *z* ≈ *z*
using *mult-homo-coords-Rep*[*of Abs-homo-coords* (*1*, *1*) *1 1 z z1 z2*]
by *auto* (*metis zero-neq-one*)
qed

lemma *mult-homo-one-right* [*simp*]:
shows $z *_h 1_h = z$
using *mult-homo-one-left*[*of z*]
by (*simp add*: *mult-homo-commute*)

Reciprocal

definition *reciprocal-homo-coords* :: *homo-coords* ⇒ *homo-coords* **where**
reciprocal-homo-coords x = (*let* (*x1*, *y1*) = *Rep-homo-coords x in Abs-homo-coords* (*y1*, *x1*))

lemma *reciprocal-homo-coords-Rep*: *Rep-homo-coords* (*reciprocal-homo-coords x*) = (*let* (*x1*, *y1*) = *Rep-homo-coords x in* (*y1*, *x1*))
apply (*cases Rep-homo-coords x*)
unfolding *reciprocal-homo-coords-def Let-def*
using *Rep-homo-coords*[*of x*]

by (*auto simp add*: *Abs-homo-coords-inverse*)

lift-definition *reciprocal-homo* :: *complex-homo* ⇒ *complex-homo* **is** *reciprocal-homo-coords*
proof−
  fix *x y*
  assume $x \approx y$
  thus *reciprocal-homo-coords* $x \approx$ *reciprocal-homo-coords* y
    by (*cases Rep-homo-coords x*, *cases Rep-homo-coords y*) (*auto simp add*: *reciprocal-homo-coords-Rep*)
qed

lemma [*simp*]: *reciprocal-homo-coords* (*reciprocal-homo-coords z*) = *z*
  unfolding *reciprocal-homo-coords-def*[*of reciprocal-homo-coords z*]
  by (*cases Rep-homo-coords z*) (*auto simp add*: *reciprocal-homo-coords-Rep*, *metis Rep-homo-coords-inverse*)

lemma [*simp*]: *reciprocal-homo* (*reciprocal-homo z*) = *z*
by (*transfer*) (*auto*, *rule-tac x=1* **in** *exI*, *simp*)

lemma [*simp*]: *reciprocal-homo* $0_h = \infty_h$
by (*transfer*) (*simp add*: *reciprocal-homo-coords-Rep*)

lemma [*simp*]: *reciprocal-homo* $\infty_h = 0_h$
by (*transfer*) (*simp add*: *reciprocal-homo-coords-Rep*)

lemma [*simp*]: *reciprocal-homo* $1_h = 1_h$
by (*transfer*) (*simp add*: *reciprocal-homo-coords-Rep*)

Division

definition *divide-homo* :: *complex-homo* ⇒ *complex-homo* ⇒ *complex-homo* (**infixl** $:_h$ *100*) **where**
  $x :_h y = x *_h$ (*reciprocal-homo y*)

lemma [*simp*]:
  assumes $z \neq 0_h$
  shows $z :_h 0_h = \infty_h$
using *assms*
unfolding *divide-homo-def*
by *simp*

lemma [*simp*]:
  assumes $z \neq \infty_h$
  shows $z :_h \infty_h = 0_h$
using *assms*
unfolding *divide-homo-def*
by *simp*

lemma [*simp*]: $\infty_h :_h 0_h = \infty_h$
unfolding *divide-homo-def*

by (*transfer*) (*simp add*: *reciprocal-homo-coords-def mult-homo-coords-def*)

lemma [*simp*]: $0_h :_h \infty_h = 0_h$
unfolding *divide-homo-def*
by (*transfer*) (*simp add*: *mult-homo-coords-def reciprocal-homo-coords-def*)

lemma *divide-homo-one* [*simp*]:
  shows $z :_h 1_h = z$
unfolding *divide-homo-def*
by *simp*

lemma *of-complex-divide*:
  assumes $z2 \neq 0$
  shows (*of-complex z1*) $:_h$ (*of-complex z2*) = *of-complex* (*z1* / *z2*)
using *assms*
unfolding *divide-homo-def*
proof (*transfer*)
  fix *z1 z2* :: *complex*
  assume $z2 \neq 0$
  thus *of-complex-coords z1* $*_{hc}$ *reciprocal-homo-coords* (*of-complex-coords z2*) ≈
      of-complex-coords (*z1* / *z2*)
   by (*auto simp add*: *of-complex-coords-def Abs-homo-coords-inverse mult-homo-coords-def reciprocal-homo-coords-def*)
      (*rule-tac x=1/z2* **in** *exI*, *auto*)
qed

lemma *divide-homo-coords-Rep* [*simp*]:
  assumes *Rep-homo-coords z* = (*z1*, *z2*) *Rep-homo-coords w* = (*w1*, *w2*)
      $(z2 \neq 0 \vee w2 \neq 0) \wedge (w1 \neq 0 \vee z1 \neq 0)$
  shows *Rep-homo-coords* (*z* $*_{hc}$ (*reciprocal-homo-coords w*)) = (*z1*∗*w2*, *z2*∗*w1*)
using *assms*
using *mult-homo-coords-Rep*[*of z z1 z2 reciprocal-homo-coords w w2 w1*] *reciprocal-homo-coords-Rep*[*of w*]
by *simp*

Conjugate

definition *cnj-homo-coords* **where**
  cnj-homo-coords z = (*let* (*z1*, *z2*) = *Rep-homo-coords z in Abs-homo-coords* (*cnj z1*, *cnj z2*))

lemma [*simp*]: *Rep-homo-coords* (*cnj-homo-coords z*) = *vec-cnj* (*Rep-homo-coords z*)
apply (*cases Rep-homo-coords z*)
using *Rep-homo-coords*[*of z*]
by (*simp add*: *cnj-homo-coords-def Abs-homo-coords-inverse vec-cnj-def*)

lift-definition *cnj-homo* :: *complex-homo* ⇒ *complex-homo* **is** *cnj-homo-coords*
by *auto*

lemma *cnj-homo* (*of-complex z*) = *of-complex* (*cnj z*)
by (*transfer*) (*simp add*: *vec-cnj-def*)

lemma *cnj-homo* $\infty_h = \infty_h$
by (*transfer*) (*simp add*: *vec-cnj-def*)

lemma *cnj-homo-coords-involution* [*simp*]:
  cnj-homo-coords (*cnj-homo-coords z*) = *z*
unfolding *cnj-homo-coords-def*[*of cnj-homo-coords z*] *Let-def*
by (*cases Rep-homo-coords z*, *auto simp add*: *Let-def split-def vec-cnj-def*) (*metis Rep-homo-coords-inverse*)

lemma *cnj-homo-involution* [*simp*]: *cnj-homo* (*cnj-homo z*) = *z*
by (*transfer*) (*auto*, *rule-tac x=1* **in** *exI*, *simp*)

lemma [*simp*]:
  cnj-homo $\infty_h = \infty_h$
by (*transfer*) (*auto simp add*: *vec-cnj-def*)

lemma [*simp*]:
  cnj-homo $0_h = 0_h$
by (*transfer*) (*auto simp add*: *vec-cnj-def*)

Inversion

definition *inversion-homo* **where**
  inversion-homo = *cnj-homo* ∘ *reciprocal-homo*

lemma *inversion-homo-sym*:
  inversion-homo = *reciprocal-homo* ∘ *cnj-homo*
unfolding *inversion-homo-def*
by (*rule ext*, *simp*) (*transfer*, *case-tac Rep-homo-coords x*, *auto simp add*: *reciprocal-homo-coords-Rep split-def Let-def vec-cnj-def*, *metis zero-neq-one*)

lemma *inversion-homo-involution* [*simp*]: *inversion-homo* (*inversion-homo z*) = *z*
proof−
  have ∗: *cnj-homo* ∘ *reciprocal-homo* = *reciprocal-homo* ∘ *cnj-homo*
    using *inversion-homo-sym*
    by (*simp add*: *inversion-homo-def*)
  show *?thesis*
    unfolding *inversion-homo-def*
    by (*subst* ∗) *simp*
qed

lemma [*simp*]:
  inversion-homo $0_h = \infty_h$
by (*simp add*: *inversion-homo-def*)

lemma [*simp*]:
  inversion-homo $\infty_h = 0_h$

```
by (simp add: inversion-homo-def)
```

8.1 Ratio and crossratio

```
definition ratio-rep where
  ratio-rep z1 z2 z3 =
      (let (z1x, z1y) = Rep-homo-coords z1;
           (z2x, z2y) = Rep-homo-coords z2;
           (z3x, z3y) = Rep-homo-coords z3 in
        Abs-homo-coords ((z1x∗z2y − z2x∗z1y)∗z3y, (z1x∗z3y − z3x∗z1y)∗z2y))

lemma ratio-rep-Rep [simp]:
 assumes (¬ z1 ≈ z2 ∧ ¬ z3 ≈ inf-homo-rep) ∨ (¬ z1 ≈ z3 ∧ ¬ z2 ≈ inf-homo-rep)
 shows Rep-homo-coords (ratio-rep z1 z2 z3) = (let (z1x, z1y) = Rep-homo-coords
z1;
          (z2x, z2y) = Rep-homo-coords z2;
          (z3x, z3y) = Rep-homo-coords z3 in ((z1x∗z2y − z2x∗z1y)∗z3y, (z1x∗z3y
− z3x∗z1y)∗z2y))
proof−
  obtain z1′ z1′′ where zz1: Rep-homo-coords z1 = (z1′, z1′′)
    by (rule obtain-homo-coords)
  obtain z2′ z2′′ where zz2: Rep-homo-coords z2 = (z2′, z2′′)
    by (rule obtain-homo-coords)
  obtain z3′ z3′′ where zz3: Rep-homo-coords z3 = (z3′, z3′′)
    by (rule obtain-homo-coords)
  have ((z1′ ∗ z2′′ − z2′ ∗ z1′′) ∗ z3′′, (z1′ ∗ z3′′ − z3′ ∗ z1′′) ∗ z2′′) ≠ vec-zero
    using assms
   using homo-coords-eq-mix[OF zz1 zz2] homo-coords-eq-mix[OF zz3, of inf-homo-rep
1 0]
   using homo-coords-eq-mix[OF zz1 zz3] homo-coords-eq-mix[OF zz2, of inf-homo-rep
1 0]
    by auto
  thus ?thesis
    using zz1 zz2 zz3
    unfolding ratio-rep-def Let-def
    by (simp add: Abs-homo-coords-inverse)
qed

lemma ratio-rep-Rep′ [simp]:
  assumes (z1 ≈ z2 ∨ z3 ≈ inf-homo-rep) ∧ (z1 ≈ z3 ∨ z2 ≈ inf-homo-rep)
  shows ratio-rep z1 z2 z3 = Abs-homo-coords (0, 0)
using assms
unfolding ratio-rep-def
by (cases Rep-homo-coords z1, cases Rep-homo-coords z2, cases Rep-homo-coords
z3) auto

lift-definition ratio :: complex-homo ⇒ complex-homo ⇒ complex-homo ⇒ complex-homo
is ratio-rep
proof−
```

fix *z1 z2 z3 w1 w2 w3*
assume ∗: *z1* ≈ *w1 z2* ≈ *w2 z3* ≈ *w3*
obtain *z1′ z1″* **where** *zz1*: *Rep-homo-coords z1* = (*z1′*, *z1″*)
by (*rule obtain-homo-coords*)
obtain *z2′ z2″* **where** *zz2*: *Rep-homo-coords z2* = (*z2′*, *z2″*)
by (*rule obtain-homo-coords*)
obtain *z3′ z3″* **where** *zz3*: *Rep-homo-coords z3* = (*z3′*, *z3″*)
by (*rule obtain-homo-coords*)
obtain *w1′ w1″* **where** *ww1*: *Rep-homo-coords w1* = (*w1′*, *w1″*)
by (*rule obtain-homo-coords*)
obtain *w2′ w2″* **where** *ww2*: *Rep-homo-coords w2* = (*w2′*, *w2″*)
by (*rule obtain-homo-coords*)
obtain *w3′ w3″* **where** *ww3*: *Rep-homo-coords w3* = (*w3′*, *w3″*)
by (*rule obtain-homo-coords*)

show *ratio-rep z1 z2 z3* ≈ *ratio-rep w1 w2 w3*
proof (*cases* ¬ *z1* ≈ *z2* ∧ ¬ *z3* ≈ *inf-homo-rep* ∨ ¬ *z1* ≈ *z3* ∧ ¬ *z2* ≈ *inf-homo-rep*)
case *True*
hence ¬ *w1* ≈ *w2* ∧ ¬ *w3* ≈ *inf-homo-rep* ∨ ¬ *w1* ≈ *w3* ∧ ¬ *w2* ≈ *inf-homo-rep*
using ∗ *homo-coords-eq-sym homo-coords-eq-trans*
by *metis*
thus *?thesis*
apply (*subst homo-coords-eq-def*, *unfold Let-def*)
using *ratio-rep-Rep*[*OF* ‹¬ *z1* ≈ *z2* ∧ ¬ *z3* ≈ *inf-homo-rep* ∨ ¬ *z1* ≈ *z3* ∧ ¬ *z2* ≈ *inf-homo-rep*›]
using *ratio-rep-Rep*[*OF* ‹¬ *w1* ≈ *w2* ∧ ¬ *w3* ≈ *inf-homo-rep* ∨ ¬ *w1* ≈ *w3* ∧ ¬ *w2* ≈ *inf-homo-rep*›]
using *zz1 zz2 zz3 ww1 ww2 ww3* ∗
by (*simp add*: *Let-def field-simps*, (*erule-tac exE*)+) (*rule-tac x*=*k*∗*ka*∗*kb* **in** *exI*, *simp*)
next
case *False*
hence ¬ (¬ *w1* ≈ *w2* ∧ ¬ *w3* ≈ *inf-homo-rep* ∨ ¬ *w1* ≈ *w3* ∧ ¬ *w2* ≈ *inf-homo-rep*)
using ∗ *homo-coords-eq-sym homo-coords-eq-trans*
by *metis*
thus *?thesis*
using *False*
by (*simp del*: *homo-coords-eq-def*)
qed
qed

lemma *ratio-is-ratio*:
assumes *z1* ≠ *z2* ∨ *z1* ≠ *z3 z1* ≠ ∞_h *z2* ≠ ∞_h ∨ *z3* ≠ ∞_h
shows *ratio z1 z2 z3* = (*z1* $-_h$ *z2*) $:_h$ (*z1* $-_h$ *z3*)
unfolding *minus-homo-def divide-homo-def*
using *assms*
proof *transfer*

```
  fix z w v
  obtain z1 z2 where zz: Rep-homo-coords z = (z1, z2)
    by (rule obtain-homo-coords)
  obtain w1 w2 where ww: Rep-homo-coords w = (w1, w2)
    by (rule obtain-homo-coords)
  obtain v1 v2 where vv: Rep-homo-coords v = (v1, v2)
    by (rule obtain-homo-coords)
  assume *: ¬ z ≈ w ∨ ¬ z ≈ v ¬ z ≈ inf-homo-rep
           ¬ w ≈ inf-homo-rep ∨ ¬ v ≈ inf-homo-rep
  hence **: ¬ z ≈ w ∧ ¬ v ≈ inf-homo-rep ∨ ¬ z ≈ v ∧ ¬ w ≈ inf-homo-rep
    by (metis homo-coords-eq-trans)
  have z2 ≠ 0 w2 ≠ 0 ∨ v2 ≠ 0 z1*w2 ≠ z2*w1 ∨ z1*v2 ≠ z2*v1
   using zz vv ww not-inf-snd-not0[of v] not-inf-snd-not0[of z] not-inf-snd-not0[of
w] homo-coords-eq-mix[of z z1 z2 w w1 w2]  homo-coords-eq-mix[of z z1 z2 v v1 v2]
*
    by auto
  thus ratio-rep z w v ≈
    z +_hc uminus-homo-coords w *_hc
    reciprocal-homo-coords (z +_hc uminus-homo-coords v)
    using zz ww vv **
     using divide-homo-coords-Rep[of z +_hc uminus-homo-coords w z1 * w2 + -
w1 * z2 z2 * w2 (z +_hc uminus-homo-coords v) z1 * v2 + - v1 * z2 z2 * v2 ]
    using minus-homo-coords-Rep[of z z1 z2 w w1 w2]
    using minus-homo-coords-Rep[of z z1 z2 v v1 v2]
    by (auto simp add: field-simps)
qed

lemma
  assumes z2 ≠ ∞_h z3 ≠ ∞_h
  shows ratio ∞_h z2 z3 = 1_h
using assms
proof transfer
  fix z2 z3
  obtain z2x z2y where zz2: Rep-homo-coords z2 = (z2x, z2y)
    by (rule obtain-homo-coords)
  obtain z3x z3y where zz3: Rep-homo-coords z3 = (z3x, z3y)
    by (rule obtain-homo-coords)
  assume ¬ z2 ≈ inf-homo-rep ¬ z3 ≈ inf-homo-rep
  have z2y ≠ 0 z3y ≠ 0
    using not-inf-snd-not0[OF ‹¬ z2 ≈ inf-homo-rep›] zz2
    using not-inf-snd-not0[OF ‹¬ z3 ≈ inf-homo-rep›] zz3
    by auto
  thus ratio-rep inf-homo-rep z2 z3 ≈ one-homo-rep
    using ‹¬ z2 ≈ inf-homo-rep› ‹¬ z3 ≈ inf-homo-rep› zz2 zz3
   by (subst homo-coords-eq-def, subst ratio-rep-Rep, simp-all) (rule-tac x=1/(z2y*z3y)
in exI, auto)
qed
```

```
lemma
  assumes z1 ≠ ∞_h z3 ≠ ∞_h
  shows ratio z1 ∞_h z3 = ∞_h
using assms
proof transfer
  fix z1 z3
  obtain z1x z1y where zz1: Rep-homo-coords z1 = (z1x, z1y)
    by (rule obtain-homo-coords)
  obtain z3x z3y where zz3: Rep-homo-coords z3 = (z3x, z3y)
    by (rule obtain-homo-coords)
  assume ¬ z1 ≈ inf-homo-rep ¬ z3 ≈ inf-homo-rep
  have z1y ≠ 0 z3y ≠ 0
    using not-inf-snd-not0[OF ‹¬ z1 ≈ inf-homo-rep›] zz1
    using not-inf-snd-not0[OF ‹¬ z3 ≈ inf-homo-rep›] zz3
    by auto
  thus ratio-rep z1 inf-homo-rep z3 ≈ inf-homo-rep
    using ‹¬ z1 ≈ inf-homo-rep› ‹¬ z3 ≈ inf-homo-rep› zz1 zz3
   by (subst homo-coords-eq-def, subst ratio-rep-Rep, simp-all) (rule-tac x=−1/(z1y*z3y)
in exI, auto)
qed


lemma
  assumes z1 ≠ ∞_h z2 ≠ ∞_h
  shows ratio z1 z2 ∞_h = 0_h
using assms
proof transfer
  fix z1 z2
  obtain z1x z1y where zz1: Rep-homo-coords z1 = (z1x, z1y)
    by (rule obtain-homo-coords)
  obtain z2x z2y where zz2: Rep-homo-coords z2 = (z2x, z2y)
    by (rule obtain-homo-coords)
  assume ¬ z1 ≈ inf-homo-rep ¬ z2 ≈ inf-homo-rep
  have z1y ≠ 0 z2y ≠ 0
    using not-inf-snd-not0[OF ‹¬ z1 ≈ inf-homo-rep›] zz1
    using not-inf-snd-not0[OF ‹¬ z2 ≈ inf-homo-rep›] zz2
    by auto
  thus ratio-rep z1 z2 inf-homo-rep ≈ zero-homo-rep
    using ‹¬ z1 ≈ inf-homo-rep› ‹¬ z2 ≈ inf-homo-rep› zz1 zz2
   by (subst homo-coords-eq-def, subst ratio-rep-Rep, simp-all) (rule-tac x=−1/(z1y*z2y)
in exI, auto)
qed


lemma
  assumes z1 ≠ z2 z1 ≠ ∞_h
  shows ratio z1 z2 z1 = ∞_h
proof−
  have z1 −_h z2 ≠ 0_h
```

using *diff-zero-homo*[*of z1 z2*] ‹*z1* ≠ *z2*› ‹*z1* ≠ ∞_h›
by *auto*
thus *?thesis*
using *assms*
using *ratio-is-ratio*[*of z1 z2 z1*]
by *simp*
qed

definition *cross-ratio-rep* **where**
cross-ratio-rep z u v w =
(*let* (*z'*, *z''*) = *Rep-homo-coords z*;
(*u'*, *u''*) = *Rep-homo-coords u*;
(*v'*, *v''*) = *Rep-homo-coords v*;
(*w'*, *w''*) = *Rep-homo-coords w*
in Abs-homo-coords ((*z'*∗*u''* − *u'*∗*z''*)∗(*v'*∗*w''* − *w'*∗*v''*),
(*z'*∗*w''* − *w'*∗*z''*)∗(*v'*∗*u''* − *u'*∗*v''*)))

lemma *cross-ratio-rep-Rep* [*simp*]:
assumes (¬ *z1* ≈ *z2* ∧ ¬ *z3* ≈ *z4*) ∨ (¬ *z1* ≈ *z4* ∧ ¬ *z2* ≈ *z3*)
shows *Rep-homo-coords* (*cross-ratio-rep z1 z2 z3 z4*) =
(*let* (*z1'*, *z1''*) = *Rep-homo-coords z1*;
(*z2'*, *z2''*) = *Rep-homo-coords z2*;
(*z3'*, *z3''*) = *Rep-homo-coords z3*;
(*z4'*, *z4''*) = *Rep-homo-coords z4*
in ((*z1'*∗*z2''* − *z2'*∗*z1''*)∗(*z3'*∗*z4''* − *z4'*∗*z3''*), (*z1'*∗*z4''* − *z4'*∗*z1''*)∗(*z3'*∗*z2''*
− *z2'*∗*z3''*)))
proof−
obtain *z1' z1''* **where** *zz1*: *Rep-homo-coords z1* = (*z1'*, *z1''*)
by (*rule obtain-homo-coords*)
obtain *z2' z2''* **where** *zz2*: *Rep-homo-coords z2* = (*z2'*, *z2''*)
by (*rule obtain-homo-coords*)
obtain *z3' z3''* **where** *zz3*: *Rep-homo-coords z3* = (*z3'*, *z3''*)
by (*rule obtain-homo-coords*)
obtain *z4' z4''* **where** *zz4*: *Rep-homo-coords z4* = (*z4'*, *z4''*)
by (*rule obtain-homo-coords*)
show *?thesis*
using *zz1 zz2 zz3 zz4*
using *assms*
unfolding *cross-ratio-rep-def Let-def*
using *homo-coords-eq-mix*[*OF zz1 zz2*] *homo-coords-eq-mix*[*OF zz3 zz4*]
using *homo-coords-eq-mix*[*OF zz1 zz4*] *homo-coords-eq-mix*[*OF zz2 zz3*]
by (*auto simp add*: *Abs-homo-coords-inverse*)
qed

lift-definition *cross-ratio* :: *complex-homo* ⇒ *complex-homo* ⇒ *complex-homo* ⇒
complex-homo ⇒ *complex-homo* **is** *cross-ratio-rep*
proof−

```
fix z1 z2 z3 z4 w1 w2 w3 w4
obtain z1' z1'' where zz1: Rep-homo-coords z1 = (z1', z1'')
  by (rule obtain-homo-coords)
obtain z2' z2'' where zz2: Rep-homo-coords z2 = (z2', z2'')
  by (rule obtain-homo-coords)
obtain z3' z3'' where zz3: Rep-homo-coords z3 = (z3', z3'')
  by (rule obtain-homo-coords)
obtain z4' z4'' where zz4: Rep-homo-coords z4 = (z4', z4'')
  by (rule obtain-homo-coords)
obtain w1' w1'' where ww1: Rep-homo-coords w1 = (w1', w1'')
  by (rule obtain-homo-coords)
obtain w2' w2'' where ww2: Rep-homo-coords w2 = (w2', w2'')
  by (rule obtain-homo-coords)
obtain w3' w3'' where ww3: Rep-homo-coords w3 = (w3', w3'')
  by (rule obtain-homo-coords)
obtain w4' w4'' where ww4: Rep-homo-coords w4 = (w4', w4'')
  by (rule obtain-homo-coords)
let ?w12 = w1' ∗ w2'' − w2' ∗ w1''
let ?w34 = w3' ∗ w4'' − w4' ∗ w3''
let ?w14 = w1' ∗ w4'' − w4' ∗ w1''
let ?w32 = w3' ∗ w2'' − w2' ∗ w3''
let ?z12 = z1' ∗ z2'' − z2' ∗ z1''
let ?z34 = z3' ∗ z4'' − z4' ∗ z3''
let ?z14 = z1' ∗ z4'' − z4' ∗ z1''
let ?z32 = z3' ∗ z2'' − z2' ∗ z3''

assume ∗: z1 ≈ w1 z2 ≈ w2 z3 ≈ w3 z4 ≈ w4
hence ∗∗:
   ?w12 ∗ ?w34 = 0 ←→ ?z12 ∗ ?z34 = 0 ?w14 ∗ ?w32 = 0 ←→ ?z14 ∗ ?z32
= 0
  using zz1 zz2 zz3 zz4 ww1 ww2 ww3 ww4
  by auto

show cross-ratio-rep z1 z2 z3 z4 ≈ cross-ratio-rep w1 w2 w3 w4
proof (cases ?z12 ∗ ?z34 = 0 ∧ ?z14 ∗ ?z32 = 0)
  case True
  thus ?thesis
    using zz1 zz2 zz3 zz4 ww1 ww2 ww3 ww4 ∗∗
   by (simp add: cross-ratio-rep-def split-def Let-def) (rule-tac x=1 in exI, auto)
next
  case False
  have ¬ z1 ≈ z2 ∧ ¬ z3 ≈ z4 ∨ ¬ z1 ≈ z4 ∧ ¬ z2 ≈ z3
    using False
    using homo-coords-eq-mix[OF zz1 zz2] homo-coords-eq-mix[OF zz3 zz4]
    using homo-coords-eq-mix[OF zz1 zz4] homo-coords-eq-mix[OF zz2 zz3]
    by (simp del: homo-coords-eq-def) metis
  moreover
  have ¬ w1 ≈ w2 ∧ ¬ w3 ≈ w4 ∨ ¬ w1 ≈ w4 ∧ ¬ w2 ≈ w3
    using ∗∗ False
```

```
      using homo-coords-eq-mix[OF ww1 ww2] homo-coords-eq-mix[OF ww3 ww4]
      using homo-coords-eq-mix[OF ww1 ww4] homo-coords-eq-mix[OF ww2 ww3]
      by (simp del: homo-coords-eq-def) metis
    ultimately
    show ?thesis
      using *
      using cross-ratio-rep-Rep[of z1 z2 z3 z4]
      using cross-ratio-rep-Rep[of w1 w2 w3 w4]
      using zz1 zz2 zz3 zz4 ww1 ww2 ww3 ww4
      apply simp
      apply (erule exE)+
      apply simp
      apply (rule-tac x=k*ka*kb*kc in exI)
      apply (simp add: field-simps)
      done
  qed
qed

lemma cross-ratio z 0_h 1_h ∞_h = z
proof (transfer)
  fix z
   have *: ¬ z ≈ zero-homo-rep ∧ ¬ one-homo-rep ≈ inf-homo-rep ∨ ¬ z ≈ inf-homo-rep ∧ ¬ zero-homo-rep ≈ one-homo-rep
    by (cases Rep-homo-coords z) auto
  show cross-ratio-rep z zero-homo-rep one-homo-rep inf-homo-rep ≈ z
    using cross-ratio-rep-Rep[OF *]
    by (simp add: split-def Let-def) (rule-tac x=−1 in exI, simp)
qed

lemma cross-ratio-0:
  assumes z1 ≠ z2 z1 ≠ z3
  shows cross-ratio z1 z1 z2 z3 = 0_h
using assms
proof (transfer)
  fix z1 z2 z3
   let ?z1 = Rep-homo-coords z1 and ?z2 = Rep-homo-coords z2 and ?z3 = Rep-homo-coords z3
  assume ¬ z1 ≈ z2 ¬ z1 ≈ z3
  thus cross-ratio-rep z1 z1 z2 z3 ≈ zero-homo-rep
    using cross-ratio-rep-Rep[of z1 z1 z2 z3]
    homo-coords-eq-mix[of z1 fst ?z1 snd ?z1 z2 fst ?z2 snd ?z2] homo-coords-eq-mix[of z1 fst ?z1 snd ?z1 z3 fst ?z3 snd ?z3]
    by (cases ?z1, cases ?z2, cases ?z3, simp add: split-def Let-def)
qed

lemma cross-ratio-1:
  assumes z1 ≠ z2 z2 ≠ z3
  shows cross-ratio z2 z1 z2 z3 = 1_h
using assms
```

proof (*transfer*)
fix *z1 z2 z3*
obtain *z1′ z1″* **where** *zz1*: *Rep-homo-coords z1* = (*z1′*, *z1″*)
by (*rule obtain-homo-coords*)
obtain *z2′ z2″* **where** *zz2*: *Rep-homo-coords z2* = (*z2′*, *z2″*)
by (*rule obtain-homo-coords*)
obtain *z3′ z3″* **where** *zz3*: *Rep-homo-coords z3* = (*z3′*, *z3″*)
by (*rule obtain-homo-coords*)
assume ¬ *z1* ≈ *z2* ¬ *z2* ≈ *z3*
thus *cross-ratio-rep z2 z1 z2 z3* ≈ *one-homo-rep*
using *zz1 zz2 zz3*
using *homo-coords-eq-mix*[*of z1 z1′ z1″ z2 z2′ z2″*] *homo-coords-eq-mix*[*of z2 z2′ z2″ z3 z3′ z3″*]
by (*auto simp add*: *cross-ratio-rep-def split-def Let-def Abs-homo-coords-inverse*) (*rule-tac x=1* / ((*z2′* ∗ *z3″* − *z3′* ∗ *z2″*) ∗ (*z2′* ∗ *z1″* − *z1′* ∗ *z2″*)) **in** *exI*, *simp*)
qed

lemma *cross-ratio-inf*:
assumes *z1* ≠ *z3 z2* ≠ *z3*
shows *cross-ratio z3 z1 z2 z3* = ∞_h
using *assms*
proof (*transfer*)
fix *z1 z2 z3*
obtain *z1′ z1″* **where** *zz1*: *Rep-homo-coords z1* = (*z1′*, *z1″*)
by (*rule obtain-homo-coords*)
obtain *z2′ z2″* **where** *zz2*: *Rep-homo-coords z2* = (*z2′*, *z2″*)
by (*rule obtain-homo-coords*)
obtain *z3′ z3″* **where** *zz3*: *Rep-homo-coords z3* = (*z3′*, *z3″*)
by (*rule obtain-homo-coords*)
assume ¬ *z1* ≈ *z3* ¬ *z2* ≈ *z3*
thus *cross-ratio-rep z3 z1 z2 z3* ≈ *inf-homo-rep*
using *zz1 zz2 zz3*
using *homo-coords-eq-mix*[*of z1 z1′ z1″ z3 z3′ z3″*] *homo-coords-eq-mix*[*of z2 z2′ z2″ z3 z3′ z3″*]
by (*auto simp add*: *cross-ratio-rep-def split-def Let-def Abs-homo-coords-inverse*)
qed

lemma
assumes (*z* ≠ *u* ∧ *v* ≠ *w*) ∨ (*z* ≠ *w* ∧ *u* ≠ *v*) *z* ≠ ∞_h *u* ≠ ∞_h *v* ≠ ∞_h *w* ≠ ∞_h
shows *cross-ratio z u v w* = $((z -_h u) *_h (v -_h w)) :_h ((z -_h w) *_h (v -_h u))$
using *assms*
unfolding *minus-homo-def divide-homo-def*
proof *transfer*
fix *z u v w*
obtain *z1 z2* **where** *zz*: *Rep-homo-coords z* = (*z1*, *z2*)
by (*rule obtain-homo-coords*)
obtain *u1 u2* **where** *uu*: *Rep-homo-coords u* = (*u1*, *u2*)
by (*rule obtain-homo-coords*)

obtain *v1 v2* **where** *vv*: *Rep-homo-coords v* = (*v1*, *v2*)
  by (*rule obtain-homo-coords*)
obtain *w1 w2* **where** *ww*: *Rep-homo-coords w* = (*w1*, *w2*)
  by (*rule obtain-homo-coords*)

assume ∗: ¬ *z* ≈ *u* ∧ ¬ *v* ≈ *w* ∨ ¬ *z* ≈ *w* ∧ ¬ *u* ≈ *v* **and**
  ∗∗: ¬ *z* ≈ *inf-homo-rep* ¬ *u* ≈ *inf-homo-rep* ¬ *v* ≈ *inf-homo-rep* ¬ *w* ≈ *inf-homo-rep*
have *z2* ≠ *0 u2* ≠ *0 v2* ≠ *0 w2* ≠ *0*
  using ∗∗ *zz uu vv ww*
  using *not-inf-snd-not0*[*of z*] *not-inf-snd-not0*[*of u*] *not-inf-snd-not0*[*of v*] *not-inf-snd-not0*[*of w*]
  by *simp-all*
moreover
have ((*z1*∗*u2* − *z2*∗*u1* ≠ *0*) ∧ (*v1*∗*w2* − *v2*∗*w1* ≠ *0*)) ∨ ((*z1*∗*w2* − *z2*∗*w1* ≠ *0*) ∧ (*v1*∗*u2* − *v2*∗*u1* ≠ *0*))
  using ∗
  apply (*subst* (*asm*) *homo-coords-eq-mix*[*OF zz uu*])
  apply (*subst* (*asm*) *homo-coords-eq-mix*[*OF vv ww*])
  apply (*subst* (*asm*) *homo-coords-eq-mix*[*OF zz ww*])
  apply (*subst* (*asm*) *homo-coords-eq-mix*[*OF uu vv*])
  by (*auto simp add*: *field-simps*)
moreover
hence *z1* ∗ *w2* ≠ *w1* ∗ *z2* ∧ *v1* ∗ *u2* ≠ *u1* ∗ *v2* ∨ *z1* ∗ *u2* ≠ *u1* ∗ *z2* ∧ *v1* ∗ *w2* ≠ *w1* ∗ *v2*
  by *auto*
ultimately
show *cross-ratio-rep z u v w* ≈
    z $+_{hc}$ *uminus-homo-coords u* $*_{hc}$ (*v* $+_{hc}$ *uminus-homo-coords w*) $*_{hc}$
    reciprocal-homo-coords
     (*z* $+_{hc}$ *uminus-homo-coords w* $*_{hc}$ (*v* $+_{hc}$ *uminus-homo-coords u*))
  using *uu vv ww zz* ∗
  apply *simp*
  apply (*subst divide-homo-coords-Rep*[*of* (*z* $+_{hc}$ *uminus-homo-coords u*) $*_{hc}$ (*v* $+_{hc}$ *uminus-homo-coords w*) (*z1* ∗ *u2* − *u1* ∗ *z2*) ∗ (*v1* ∗ *w2* − *w1* ∗ *v2*) *z2* ∗ *u2* ∗ (*v2* ∗ *w2*) (*z* $+_{hc}$ *uminus-homo-coords w*) $*_{hc}$ (*v* $+_{hc}$ *uminus-homo-coords u*) (*z1* ∗ *w2* − *w1* ∗ *z2*) ∗ (*v1* ∗ *u2* − *u1* ∗ *v2*)
      z2 ∗ *w2* ∗ (*v2* ∗ *u2*)])
  using *mult-homo-coords-Rep*[*of z* $+_{hc}$ *uminus-homo-coords u z1* ∗ *u2* − *u1* ∗ *z2 z2* ∗ *u2 v* $+_{hc}$ *uminus-homo-coords w v1* ∗ *w2* − *w1* ∗ *v2 v2* ∗ *w2*]
  using *minus-homo-coords-Rep*[*of z z1 z2 u u1 u2*]
  using *minus-homo-coords-Rep*[*of v v1 v2 w w1 w2*]
  using *mult-homo-coords-Rep*[*of z* $+_{hc}$ *uminus-homo-coords w z1* ∗ *w2* − *w1* ∗ *z2 z2* ∗ *w2 v* $+_{hc}$ *uminus-homo-coords u v1* ∗ *u2* − *u1* ∗ *v2 v2* ∗ *u2*]
  using *minus-homo-coords-Rep*[*of z z1 z2 w w1 w2*]
  using *minus-homo-coords-Rep*[*of v v1 v2 u u1 u2*]
  using *mult-homo-coords-Rep*[*of z* $+_{hc}$ *uminus-homo-coords u z1* ∗ *u2* − *u1* ∗ *z2 z2* ∗ *u2 v* $+_{hc}$ *uminus-homo-coords w v1* ∗ *w2* − *w1* ∗ *v2 v2* ∗ *w2*]
  using *minus-homo-coords-Rep*[*of z z1 z2 u u1 u2*]

using *minus-homo-coords-Rep*[*of v v1 v2 w w1 w2*]
by *simp-all* (*rule-tac x=z2∗u2∗*(*v2∗w2*) **in** *exI*, *simp*)
qed

8.2 Distance

definition *inprod-homo-rep* **where**
inprod-homo-rep z w =
(*let* (*z1*, *z2*) = *Rep-homo-coords z*;
(*w1*, *w2*) = *Rep-homo-coords w*
in vec-cnj (*z1*, *z2*) $*_{vv}$ (*w1*, *w2*))
syntax
-inprod-homo-rep :: *homo-coords* ⇒ *homo-coords* ⇒ *complex* (⟨-,-⟩)
translations
⟨*z*,*w*⟩ == *CONST inprod-homo-rep z w*

lemma [*simp*]: *is-real* ⟨*z*,*z*⟩
unfolding *inprod-homo-rep-def*
by (*cases Rep-homo-coords z*, *simp add*: *vec-cnj-def*)

lemma [*simp*]: *Re* ⟨*z*,*z*⟩ ≥ *0*
unfolding *inprod-homo-rep-def*
by (*cases Rep-homo-coords z*, *simp add*: *vec-cnj-def*)

lemma *inprod-homo-bilinear1*:
assumes *Rep-homo-coords z'* = *k* $*_{sv}$ *Rep-homo-coords z*
shows ⟨*z'*,*w*⟩ = *cnj k* ∗ ⟨*z*,*w*⟩
using *assms*
unfolding *inprod-homo-rep-def Let-def*
by (*cases Rep-homo-coords z*, *cases Rep-homo-coords z'*, *cases Rep-homo-coords w*)
(*simp add*: *vec-cnj-def complex-cnj field-simps*)

lemma *inprod-homo-bilinear2*:
assumes *Rep-homo-coords w'* = *k* $*_{sv}$ *Rep-homo-coords w*
shows ⟨*z*,*w'*⟩ = *k* ∗ ⟨*z*,*w*⟩
using *assms*
unfolding *inprod-homo-rep-def Let-def*
by (*cases Rep-homo-coords z*, *cases Rep-homo-coords z'*, *cases Rep-homo-coords w*)
(*simp add*: *vec-cnj-def complex-cnj field-simps*)

definition *norm-homo-rep* **where**
norm-homo-rep z = *sqrt* (*Re* ⟨*z*,*z*⟩)
syntax
-norm-homo-rep :: *homo-coords* ⇒ *complex* (⟨-⟩)
translations
⟨*z*⟩ == *CONST norm-homo-rep z*

lemma
norm-homo-rep-square: $\langle z\rangle^2$ = *Re* (⟨*z*,*z*⟩)

```
unfolding norm-homo-rep-def
by simp

lemma norm-homo-gt-0: ⟨z⟩ > 0
proof−
  obtain z1 z2 where Rep-homo-coords z = (z1, z2)
    by (rule obtain-homo-coords)
  thus ?thesis
   using complex-mult-cnj-cmod[of z1] complex-mult-cnj-cmod[of z2] Rep-homo-coords[of
z]
    unfolding norm-homo-rep-def inprod-homo-rep-def
    by (simp add: vec-cnj-def split-def Let-def field-simps power2-eq-square) (metis
norm-eq-zero sum-squares-gt-zero-iff)
qed

lemma norm-homo-scale:
  assumes Rep-homo-coords z' = k *_sv Rep-homo-coords z
  shows ⟨z'⟩^2 = Re (cnj k * k) * ⟨z⟩^2
apply (subst norm-homo-rep-square)+
apply (subst inprod-homo-bilinear1[OF assms])
apply (subst inprod-homo-bilinear2[OF assms])
apply (simp add: field-simps)
done

definition dist-homo-rep where
  dist-homo-rep z1 z2 =
     (let (z1x, z1y) = Rep-homo-coords z1;
          (z2x, z2y) = Rep-homo-coords z2;
          num = (z1x*z2y − z2x*z1y) * (cnj z1x*cnj z2y − cnj z2x*cnj z1y);
          den = (z1x*cnj z1x + z1y*cnj z1y) * (z2x*cnj z2x + z2y*cnj z2y)
       in 2*sqrt(Re num / Re den))

lemma dist-homo-rep-iff: dist-homo-rep z w = 2*sqrt(1 − (cmod ⟨z,w⟩)^2 / (⟨z⟩^2
* ⟨w⟩^2))
proof−
  obtain z1 z2 w1 w2 where *: Rep-homo-coords z = (z1, z2) Rep-homo-coords
w = (w1, w2)
    by (cases Rep-homo-coords z, cases Rep-homo-coords w) auto
  have 1: 2*sqrt(1 − (cmod ⟨z,w⟩)^2 / (⟨z⟩^2 * ⟨w⟩^2)) = 2*sqrt((⟨z⟩^2 * ⟨w⟩^2 −
(cmod ⟨z,w⟩)^2) / (⟨z⟩^2 * ⟨w⟩^2))
    using norm-homo-gt-0[of z] norm-homo-gt-0[of w]
    by (simp add: field-simps)

  have 2: ⟨z⟩^2 * ⟨w⟩^2 = Re ((z1*cnj z1 + z2*cnj z2) * (w1*cnj w1 + w2*cnj
w2))
    using *
    by (simp add: norm-homo-rep-def inprod-homo-rep-def vec-cnj-def)

  have 3: ⟨z⟩^2 * ⟨w⟩^2 − (cmod ⟨z,w⟩)^2 = Re ((z1*w2 − w1*z2) * (cnj z1*cnj w2
```

− *cnj w1*∗*cnj z2*))
  apply (*subst cmod-square*, (*subst norm-homo-rep-square*)+)
  using ∗
  by (*simp add*: *inprod-homo-rep-def vec-cnj-def field-simps*)

 thus *?thesis*
  using *1 2 3*
  using ∗
  unfolding *dist-homo-rep-def Let-def*
  by *simp*
qed

lift-definition *dist-homo* :: *complex-homo* ⇒ *complex-homo* ⇒ *real* **is** *dist-homo-rep*
proof−
 fix *z1 z2 z1' z2'*
 obtain *z1x z1y z2x z2y z1'x z1'y z2'x z2'y* **where**
  zz: *Rep-homo-coords z1* = (*z1x*, *z1y*) *Rep-homo-coords z2* = (*z2x*, *z2y*) *Rep-homo-coords z1'* = (*z1'x*, *z1'y*) *Rep-homo-coords z2'* = (*z2'x*, *z2'y*)
  by (*cases Rep-homo-coords z1*, *cases Rep-homo-coords z2*, *cases Rep-homo-coords z1'*, *cases Rep-homo-coords z2'*) *blast*

 assume *z1* ≈ *z1' z2* ≈ *z2'*
 then obtain *k1 k2* **where**
  ∗: *k1* ≠ *0 Rep-homo-coords z1'* = *k1* $*_{sv}$ *Rep-homo-coords z1* **and**
  ∗∗: *k2* ≠ *0 Rep-homo-coords z2'* = *k2* $*_{sv}$ *Rep-homo-coords z2*
  by *auto*
 have (*cmod* ⟨*z1*,*z2*⟩)^2 / (⟨*z1*⟩^{2} ∗ ⟨*z2*⟩^{2}) = (*cmod* ⟨*z1'*,*z2'*⟩)^2 / (⟨*z1'*⟩^{2} ∗ ⟨*z2'*⟩^{2})
  using ‹*k1* ≠ *0*› ‹*k2* ≠ *0*›
  using *cmod-square*[*symmetric*, *of k1*] *cmod-square*[*symmetric*, *of k2*]
  apply (*subst norm-homo-scale*[*OF* ∗(*2*)])
  apply (*subst norm-homo-scale*[*OF* ∗∗(*2*)])
  apply (*subst inprod-homo-bilinear1*[*OF* ∗(*2*)])
  apply (*subst inprod-homo-bilinear2*[*OF* ∗∗(*2*)])
  by (*simp add*: *power2-eq-square*)
 thus *dist-homo-rep z1 z2* = *dist-homo-rep z1' z2'*
  by (*subst dist-homo-rep-iff*)+ *simp*
qed

lemma *dist-homo-finite*:
 dist-homo (*of-complex z1*) (*of-complex z2*) = *2* ∗ *cmod*(*z1* − *z2*) / (*sqrt* (*1*+(*cmod z1*)^2) ∗ *sqrt* (*1*+(*cmod z2*)^2))
apply *transfer*
apply (*subst cmod-square*)+
apply (*simp add*: *dist-homo-rep-def real-sqrt-divide cmod-def power2-eq-square*)
by (*smt ab-diff-minus comm-semiring-1-class.normalizing-semiring-rules*(*24*) *minus-diff-eq minus-mult-right real-sqrt-mult-distrib2*)

lemma *dist-homo-infinite1*:
 dist-homo (*of-complex z1*) ∞_h = *2* / *sqrt* (*1*+(*cmod z1*)^2)

by *transfer* (*subst cmod-square*, *simp add*: *dist-homo-rep-def real-sqrt-divide*)

lemma *dist-homo-infinite2*:
dist-homo ∞_h (*of-complex z1*) = *2* / *sqrt* (*1*+(*cmod z1*)^2)
by *transfer* (*subst cmod-square*, *simp add*: *dist-homo-rep-def real-sqrt-divide*)

lemma *dist-homo-rep-zero*:
dist-homo-rep z w = *0* $\longleftrightarrow$ (*cmod* $\langle z,w\rangle$)^2 = ($\langle z\rangle^2$ ∗ $\langle w\rangle^2$)
using *norm-homo-gt-0*[*of z*] *norm-homo-gt-0*[*of w*]
by (*subst dist-homo-rep-iff*) *auto*

lemma *dist-homo-zero1* [*simp*]: *dist-homo z z* = *0*
by *transfer* (*subst dist-homo-rep-zero*, ((*subst norm-homo-rep-square*)+), *subst cmod-square*, *simp*)

lemma *dist-homo-zero2* [*simp*]:
assumes *dist-homo z1 z2* = *0*
shows *z1* = *z2*
using *assms*
proof *transfer*
fix *z w*
obtain *z1 z2 w1 w2* **where** ∗: *Rep-homo-coords z* = (*z1*, *z2*) *Rep-homo-coords w* = (*w1*, *w2*)
by (*cases Rep-homo-coords z*, *cases Rep-homo-coords w*, *auto*)
let *?x* = (*z1*∗*w2* − *w1*∗*z2*) ∗ (*cnj z1*∗*cnj w2* − *cnj w1*∗*cnj z2*)
assume *dist-homo-rep z w* = *0*
hence (*cmod* $\langle z,w\rangle$)^2 = $\langle z\rangle^2$ ∗ $\langle w\rangle^2$
by (*subst* (*asm*) *dist-homo-rep-zero*)
hence *Re ?x* = *0*
using ∗
by (*subst* (*asm*) *cmod-square*) ((*subst* (*asm*) *norm-homo-rep-square*)+, *simp add*: *inprod-homo-rep-def vec-cnj-def field-simps*)
hence *?x* = *0*
using *complex-mult-cnj-cmod*[*of z1*∗*w2* − *w1*∗*z2*]
by (*subst complex-eq-if-Re-eq*[*of ?x 0*]) (*simp add*: *complex-cnj power2-eq-square*, *auto*)
thus *z* ≈ *w*
using *homo-coords-eq-mix*[*OF* ∗]
by (*auto simp del*: *homo-coords-eq-def*) (*metis complex-cnj-cnj complex-cnj-mult*)
qed

lemma *dist-homo-sym* [*simp*]:
shows *dist-homo z1 z2* = *dist-homo z2 z1*
by *transfer* (*simp add*: *dist-homo-rep-def split-def Let-def field-simps*)

Triangle inequality

lemma *dist-homo-triangle-finite*: *cmod*(*a* − *b*) / (*sqrt* (*1*+(*cmod a*)^2) ∗ *sqrt* (*1*+(*cmod b*)^2)) ≤ *cmod* (*a* − *c*) / (*sqrt* (*1*+(*cmod a*)^2) ∗ *sqrt* (*1*+(*cmod c*)^2)) + *cmod* (*c* − *b*) / (*sqrt* (*1*+(*cmod b*)^2) ∗ *sqrt* (*1*+(*cmod c*)^2))

proof−
let $?cc = 1+(cmod\ c)^2$ **and** $?bb = 1+(cmod\ b)^2$ **and** $?aa = 1+(cmod\ a)^2$
have $sqrt\ ?cc > 0\ sqrt\ ?aa > 0\ sqrt\ ?bb > 0$
by (*auto simp add*: *power2-eq-square*) (*metis add-strict-increasing norm-ge-zero norm-mult zero-less-one*)+

have $(a - b)*(1+cnj\ c*c) = (a-c)*(1+cnj\ c*b) + (c-b)*(1 + cnj\ c*a)$
by (*simp add*: *field-simps*)
moreover
have $cmod\ ((a - b)*(1+cnj\ c*c)) = cmod(a - b) * (1+(cmod\ c)^2)$
using *complex-mult-cnj-cmod*[*of cnj c*]
by (*auto simp add*: *power2-eq-square*) (*metis abs-add-abs abs-one abs-power2 norm-of-real of-real-1 of-real-add of-real-mult power2-eq-square*)
ultimately
have $cmod(a - b) * (1+(cmod\ c)^2) \le cmod\ (a-c) * cmod\ (1+cnj\ c*b) + cmod\ (c-b) * cmod(1 + cnj\ c*a)$
using *complex-mod-triangle-ineq2*[*of* $(a-c)*(1+cnj\ c*b)$ $(c-b)*(1 + cnj\ c*a)$]
by *simp*
moreover
have $*$: $\bigwedge a\ b\ c\ d\ b'\ d'.\ [\![b \le b';\ d \le d';\ a \ge (0::real);\ c \ge 0]\!] \Longrightarrow a*b + c*d \le a*b' + c*d'$
by (*metis add-mono comm-mult-left-mono*)
have $cmod\ (a-c) * cmod\ (1+cnj\ c*b) + cmod\ (c-b) * cmod(1 + cnj\ c*a) \le cmod\ (a - c) * (sqrt\ (1+(cmod\ c)^2) * sqrt\ (1+(cmod\ b)^2)) + cmod\ (c - b) * (sqrt\ (1+(cmod\ c)^2) * sqrt\ (1+(cmod\ a)^2))$
using $*$[*OF cmod-1-plus-mult-le*[*of cnj c b*] *cmod-1-plus-mult-le*[*of cnj c a*], *of cmod* $(a-c)$ *cmod* $(c-b)$]
by (*simp add*: *field-simps real-sqrt-mult*[*symmetric*])
ultimately
have $cmod(a - b) * ?cc \le cmod\ (a - c) * sqrt\ ?cc * sqrt\ ?bb + cmod\ (c - b) * sqrt\ ?cc * sqrt\ ?aa$
by *simp*
moreover
hence $0 \le ?cc * sqrt\ ?aa * sqrt\ ?bb$
using *mult-right-mono*[*of* $0\ sqrt\ ?aa\ \ sqrt\ ?bb$]
using *mult-right-mono*[*of* $0\ ?cc\ sqrt\ ?aa * sqrt\ ?bb$]
by *simp*
moreover
have $sqrt\ ?cc\ /\ ?cc = 1\ /\ sqrt\ ?cc$
using ‹$sqrt\ ?cc > 0$›
by (*simp add*: *field-simps*) (*metis abs-of-pos real-sqrt-abs2 real-sqrt-mult-distrib2*)
hence $sqrt\ ?cc\ /\ (?cc * sqrt\ ?aa) = 1\ /\ (sqrt\ ?aa * sqrt\ ?cc)$
using *times-divide-eq-right*[*of* $1/sqrt\ ?aa\ sqrt\ ?cc\ ?cc$]
using ‹$sqrt\ ?aa > 0$›
by *simp*
hence $cmod\ (a - c) * sqrt\ ?cc\ /\ (?cc * sqrt\ ?aa) = cmod\ (a - c)\ /\ (sqrt\ ?aa * sqrt\ ?cc)$
using *times-divide-eq-right*[*of cmod* $(a - c)\ sqrt\ ?cc\ (?cc * sqrt\ ?aa)$]
by *simp*

moreover
have *sqrt ?cc / ?cc = 1 / sqrt ?cc*
using ‹*sqrt ?cc > 0*›
by (*simp add*: *field-simps*) (*metis abs-of-pos real-sqrt-abs2 real-sqrt-mult-distrib2*)
hence *sqrt ?cc / (?cc ∗ sqrt ?bb) = 1 / (sqrt ?bb ∗ sqrt ?cc)*
using *times-divide-eq-right*[*of 1/sqrt ?bb sqrt ?cc ?cc*]
using ‹*sqrt ?bb > 0*›
by *simp*
hence *cmod (c − b) ∗ sqrt ?cc / (?cc ∗ sqrt ?bb) = cmod (c − b) / (sqrt ?bb ∗ sqrt ?cc)*
using *times-divide-eq-right*[*of cmod (c − b) sqrt ?cc ?cc ∗ sqrt ?bb*]
by *simp*
ultimately
show *?thesis*
using *divide-right-mono*[*of cmod (a − b) ∗ ?cc cmod (a − c) ∗ sqrt ?cc ∗ sqrt ?bb + cmod (c − b) ∗ sqrt ?cc ∗ sqrt ?aa ?cc ∗ sqrt ?aa ∗ sqrt ?bb*] ‹*sqrt ?aa > 0*› ‹*sqrt ?bb > 0*› ‹*sqrt ?cc > 0*›
by (*simp add*: *add-divide-distrib*)
qed

lemma *dist-homo-triangle-infinite1*: *1 / sqrt(1 + (cmod b)^2) ≤ 1 / sqrt(1 + (cmod c)^2) + cmod (b − c) / (sqrt(1 + (cmod b)^2) ∗ sqrt(1 + (cmod c)^2))*
proof−
let *?bb = sqrt (1 + (cmod b)^2)* **and** *?cc = sqrt (1 + (cmod c)^2)*
have *?bb > 0 ?cc > 0*
by (*metis add-strict-increasing real-sqrt-gt-0-iff zero-le-power2 zero-less-one*)+
hence ∗: *?bb ∗ ?cc ≥ 0*
by (*metis one-power2 real-sqrt-mult-distrib2 real-sqrt-sum-squares-mult-ge-zero*)
have ∗∗: *(?cc − ?bb) / (?bb ∗ ?cc) = 1 / ?bb − 1 / ?cc*
using ‹*sqrt (1 + (cmod b)^2) > 0*› ‹*sqrt (1 + (cmod c)^2) > 0*›
by (*simp add*: *field-simps*)
show *1 / ?bb ≤ 1 / ?cc + cmod (b − c) / (?bb ∗ ?cc)*
using *divide-right-mono*[*OF cmod-diff-ge*[*of c b*] ∗]
by (*subst* (*asm*) ∗∗) (*simp add*: *field-simps norm-minus-commute*)
qed

lemma *dist-homo-triangle-infinite2*:
1 / sqrt(1 + (cmod a)^2) ≤ cmod (a − c) / (sqrt (1+(cmod a)^2) ∗ sqrt (1+(cmod c)^2)) + 1 / sqrt(1 + (cmod c)^2)
using *dist-homo-triangle-infinite1*[*of a c*]
by *simp*

lemma *dist-homo-triangle-infinite3*:
cmod(a − b) / (sqrt (1+(cmod a)^2) ∗ sqrt (1+(cmod b)^2)) ≤ 1 / sqrt(1 + (cmod a)^2) + 1 / sqrt(1 + (cmod b)^2)
proof−
let *?aa = sqrt (1 + (cmod a)^2)* **and** *?bb = sqrt (1 + (cmod b)^2)*
have *?aa > 0 ?bb > 0*
by (*metis add-strict-increasing real-sqrt-gt-0-iff zero-le-power2 zero-less-one*)+

```
  hence *: ?aa * ?bb ≥ 0
   by (metis one-power2 real-sqrt-mult-distrib2 real-sqrt-sum-squares-mult-ge-zero)
  have **: (?aa + ?bb) / (?aa * ?bb) = 1 / ?aa + 1 / ?bb
    using ‹?aa > 0› ‹?bb > 0›
    by (simp add: field-simps)
  show cmod (a − b) / (?aa * ?bb) ≤ 1 / ?aa + 1 / ?bb
    using divide-right-mono[OF cmod-diff-le[of a b] *]
    by (subst (asm) **) (simp add: field-simps norm-minus-commute)
qed

lemma dist-homo-triangle:
  shows dist-homo A B ≤ dist-homo A C + dist-homo C B
proof (cases A = ∞_h)
  case True
  show ?thesis
  proof (cases B = ∞_h)
    case True
    show ?thesis
    proof (cases C = ∞_h)
      case True
      show ?thesis
        using ‹A = ∞_h› ‹B = ∞_h› ‹C = ∞_h›
        by simp
    next
      case False
      then obtain c where C = of-complex c
        using inf-homo-or-complex-homo[of C]
        by auto
      show ?thesis
        using ‹A = ∞_h› ‹B = ∞_h› ‹C = of-complex c›
        by (simp add: dist-homo-infinite2)
    qed
  next
    case False
    then obtain b where B = of-complex b
      using inf-homo-or-complex-homo[of B]
      by auto
    show ?thesis
    proof (cases C = ∞_h)
      case True
      show ?thesis
        using ‹A = ∞_h› ‹C = ∞_h› ‹B = of-complex b›
        by simp
    next
      case False
      then obtain c where C = of-complex c
        using inf-homo-or-complex-homo[of C]
        by auto
      show ?thesis
```

```
        using ‹A = ∞_h› ‹B = of-complex b› ‹C = of-complex c›
        using mult-left-mono[OF dist-homo-triangle-infinite1[of b c], of 2]
        by (simp add: dist-homo-finite dist-homo-infinite1 dist-homo-infinite2)
    qed
  qed
next
  case False
  then obtain a where A = of-complex a
    using inf-homo-or-complex-homo[of A]
    by auto
  show ?thesis
  proof (cases B = ∞_h)
    case True
    show ?thesis
    proof (cases C = ∞_h)
      case True
      show ?thesis
        using ‹B = ∞_h› ‹C = ∞_h› ‹A = of-complex a›
        by (simp add: dist-homo-infinite2)
    next
      case False
      then obtain c where C = of-complex c
        using inf-homo-or-complex-homo[of C]
        by auto
      show ?thesis
        using ‹B = ∞_h› ‹C = of-complex c› ‹A = of-complex a›
        using mult-left-mono[OF dist-homo-triangle-infinite2[of a c], of 2]
        by (simp add: dist-homo-finite dist-homo-infinite1 dist-homo-infinite2)
    qed
  next
    case False
    then obtain b where B = of-complex b
      using inf-homo-or-complex-homo[of B]
      by auto
    show ?thesis
    proof (cases C = ∞_h)
      case True
      thus ?thesis
        using ‹C = ∞_h› ‹B = of-complex b› ‹A = of-complex a›
        using mult-left-mono[OF dist-homo-triangle-infinite3[of a b], of 2]
        by (simp add: dist-homo-finite dist-homo-infinite1 dist-homo-infinite2)
    next
      case False
      then obtain c where C = of-complex c
        using inf-homo-or-complex-homo[of C]
        by auto
      show ?thesis
        using ‹A = of-complex a› ‹B = of-complex b› ‹C = of-complex c›
        using mult-left-mono[OF dist-homo-triangle-finite[of a b c], of 2]
      by (simp add: dist-homo-finite norm-minus-commute)
    qed
  qed
qed

instantiation complex-homo :: metric-space
begin
definition dist-complex-homo = dist-homo
definition open-complex-homo S = (∀ x∈S. ∃ e>0. ∀ y. dist-homo y x < e ⟶ y
∈ S)
instance
proof
  fix x y :: complex-homo
  show (dist x y = 0) = (x = y)
    unfolding dist-complex-homo-def
    using dist-homo-zero1[of x] dist-homo-zero2[of x y]
    by auto
next
  fix S :: complex-homo set
  show open S = (∀ x∈S. ∃ e>0. ∀ y. dist y x < e ⟶ y ∈ S)
    unfolding open-complex-homo-def dist-complex-homo-def
    by simp
next
  fix x y z :: complex-homo
  show dist x y ≤ dist x z + dist y z
    unfolding dist-complex-homo-def
    using dist-homo-triangle[of x y z]
    by simp
qed
end

end

theory RiemannSphere
imports HomogeneousCoordinates ~~/src/HOL/Library/Product-Vector
begin


lemma Lim-within: (f ---> l) (at a within S) ⟷
    (∀ e >0. ∃ d>0. ∀ x ∈ S. 0 < dist x a ∧ dist x a  < d ⟶ dist (f x) l < e)
  by (auto simp add: tendsto-iff eventually-at dist-nz)

lemma continuous-on-iff:
  continuous-on s f ⟷
    (∀ x∈s. ∀ e>0. ∃ d>0. ∀ x'∈s. dist x' x < d ⟶ dist (f x') (f x) < e)
  unfolding continuous-on-def Lim-within
  apply (intro ball-cong [OF refl] all-cong ex-cong)
  apply (rename-tac y, case-tac y = x)
  apply simp
  apply (simp add: dist-nz)
  done
```

9 Riemann sphere

```
typedef riemann-sphere = {(x::real, y::real, z::real). x∗x + y∗y + z∗z = 1}
by (rule-tac x=(1, 0, 0) in exI) simp

lemma sphere-bounds':
  assumes x∗x + y∗y + z∗z = (1::real)
  shows −1 ≤ x ∧ x ≤ 1
proof−
  from assms have x∗x ≤ 1
    by (smt real-minus-mult-self-le)
  hence x^2 ≤ 1^2 (− x)^2 ≤ 1^2
    by (auto simp add: power2-eq-square)
  show −1 ≤ x ∧ x ≤ 1
  proof (cases x ≥ 0)
    case True
    thus ?thesis
      using square-cancel[OF ‹x^2 ≤ 1^2›]
      by simp
  next
    case False
    thus ?thesis
      using square-cancel[OF ‹(−x)^2 ≤ 1^2›]
      by simp
  qed
qed

lemma sphere-bounds:
  assumes x∗x + y∗y + z∗z = (1::real)
  shows −1 ≤ x ∧ x ≤ 1  −1 ≤ y ∧ y ≤ 1  −1 ≤ z ∧ z ≤ 1
using assms
using sphere-bounds'[of x y z] sphere-bounds'[of y x z] sphere-bounds'[of z x y]
by (auto simp add: field-simps)
```

Polar coords parametrization

```
lemma sphere-params-on-sphere:
  assumes x = cos α ∗ cos β y = cos α ∗ sin β z = sin α
  shows x∗x + y∗y + z∗z = 1
proof−
  have x∗x + y∗y = (cos α ∗ cos α) ∗ (cos β ∗ cos β) + (cos α ∗ cos α) ∗ (sin
β ∗ sin β)
    using assms
    by simp
  hence x∗x + y∗y = cos α ∗ cos α
    using sin-cos-squared-add3[of β]
    by (subst (asm) distrib-left[symmetric]) (simp add: field-simps)
  thus ?thesis
    using assms
    using sin-cos-squared-add3[of α]
    by simp
qed

lemma sphere-params:
  assumes x∗x + y∗y + z∗z = 1
  shows x = cos (arcsin z) ∗ cos (atan2 y x) ∧ y = cos (arcsin z) ∗ sin (atan2 y
x) ∧ z = sin (arcsin z)
proof (cases z=1 ∨ z = −1)
  case True
  hence x = 0 ∧ y = 0
    using assms
    by auto
  thus ?thesis
    using ‹z = 1 ∨ z = −1›
    by (auto simp add: cos-arcsin)
next
  case False
  hence x ≠ 0 ∨ y ≠ 0
    using assms
    by auto (metis minus-one square-eq-1-iff)
  thus ?thesis
    using sphere-bounds[OF assms] assms
   by (auto simp add: cos-arcsin cos-arctan sin-arctan power2-eq-square field-simps
real-sqrt-divide atan2-def cos-periodic-pi2 cos-periodic-pi3 sin-periodic-pi3) (smt real-sqrt-abs2)+
qed

lemma ex-sphere-params:
  assumes x∗x + y∗y + z∗z = 1
  shows ∃ α β. x = cos α ∗ cos β ∧ y = cos α ∗ sin β ∧ z = sin α ∧ −pi / 2
≤ α ∧ α ≤ pi / 2 ∧ −pi ≤ β ∧ β < pi
using assms arcsin-bounded[of z] sphere-bounds[of x y z]
by (rule-tac x=arcsin z in exI, rule-tac x=atan2 y x in exI) (simp add: sphere-params
arcsin-bounded atan2-bounded)
```

Stereographic and inverse stereographic projection

```
definition stereographic-coords :: riemann-sphere ⇒ homo-coordswhere
stereographic-coords M = (let (x, y, z) = Rep-riemann-sphere M in
    (if (x, y, z) ≠ (0, 0, 1) then
         Abs-homo-coords (Complex x y, complex-of-real (1 − z))
     else
         Abs-homo-coords (1, 0)
    ))

lemma stereographic-coords-rep:
  Rep-homo-coords (stereographic-coords M) = (let (x, y, z) = Rep-riemann-sphere
M in
```

```
    (if (x, y, z) ≠ (0, 0, 1) then
         (Complex x y, complex-of-real (1 − z))
     else
         (1, 0)
    ))
proof−
  obtain x y z where MM: (x, y, z) = Rep-riemann-sphere M
    by (cases Rep-riemann-sphere M) auto
  show ?thesis
  proof (cases (x, y, z) ≠ (0, 0, 1) )
    case True
    thus ?thesis
     using MM[symmetric] Abs-homo-coords-inverse[of (Complex x y, 1 − cor z)]
      using Rep-riemann-sphere[of M]
     by (cases x = 0 ∧ y = 0, cases z=1) (auto simp add: stereographic-coords-def,
metis Complex-eq-1 complex-of-real-def)
  next
    case False
    thus ?thesis
      using MM
      by (simp add: stereographic-coords-def)
  qed
qed

lift-definition stereographic :: riemann-sphere ⇒ complex-homo is stereographic-coords
by (simp del: homo-coords-eq-def)

definition inv-stereographic-coords :: homo-coords ⇒ riemann-sphere where
  inv-stereographic-coords z = (
    let (z1, z2) = Rep-homo-coords z
      in if z2 = 0 then
             Abs-riemann-sphere (0, 0, 1)
         else
           let z = z1/z2;
               X = Re (2∗z / (1 + z∗cnj z));
               Y = Im (2∗z / (1 + z∗cnj z));
               Z = ((cmod z)^2 − 1) / (1 + (cmod z)^2)
             in Abs-riemann-sphere (X, Y, Z))

lift-definition inv-stereographic :: complex-homo ⇒ riemann-sphere is inv-stereographic-coords
by (auto simp add: inv-stereographic-coords-def split-def Let-def)

lemma one-plus-square-neq-zero [simp]:
  fixes x :: real
  shows 1 + (cor x)^2 ≠ 0
  by (metis (hide-lams, no-types) of-real-1 of-real-add of-real-eq-0-iff of-real-power
power-one sum-power2-eq-zero-iff zero-neq-one)

lemma Re-stereographic: Re (2 ∗ z / (1 + z ∗ cnj z)) = 2 ∗ Re z / (1 + (cmod
z)^2)
using one-plus-square-neq-zero
by (subst complex-mult-cnj-cmod, subst Re-divide-real) (auto simp add: power2-eq-square)

lemma Im-stereographic: Im (2 * z / (1 + z * cnj z)) = 2 * Im z / (1 + (cmod
z)^2)
using one-plus-square-neq-zero
by (subst complex-mult-cnj-cmod, subst Im-divide-real) (auto simp add: power2-eq-square)


lemma inv-stereographic-on-sphere:
  assumes X = Re (2*z / (1 + z*cnj z)) Y = Im (2*z / (1 + z*cnj z)) Z =
((cmod z)^2 − 1) / (1 + (cmod z)^2)
  shows X*X + Y*Y + Z*Z = 1
proof−
  have 1 + (cmod z)^2 ≠ 0
    by (metis power-one realpow-two-sum-zero-iff zero-neq-one)
  thus ?thesis
    using assms
   by (simp add: Re-stereographic Im-stereographic) (cases z, simp add: power2-eq-square
real-sqrt-mult[symmetric] add-divide-distrib[symmetric], simp add: field-simps)
qed

lemma inv-stereographic-coords-Rep:
  Rep-riemann-sphere (inv-stereographic-coords z) =
  (let (z1, z2) = Rep-homo-coords z
       in if z2 = 0 then
             (0, 0, 1)
          else
             let z = z1/z2;
                 X = Re (2*z / (1 + z*cnj z));
                 Y = Im (2*z / (1 + z*cnj z));
                 Z = ((cmod z)^2 − 1) / (1 + (cmod z)^2)
               in (X, Y, Z))
proof−
  obtain z1 z2 where zz: Rep-homo-coords z = (z1, z2)
    by (rule obtain-homo-coords)
  show ?thesis
    proof (cases z2 = 0)
      case True
      thus ?thesis
        using zz
      by (simp add: Let-def inv-stereographic-coords-def Abs-riemann-sphere-inverse)
    next
      case False
      thus ?thesis
        using inv-stereographic-on-sphere[of - z1/z2] zz
      by (simp add: Let-def inv-stereographic-coords-def Abs-riemann-sphere-inverse)
    qed
```

```
qed

definition [simp]: North = Abs-riemann-sphere (0, 0, 1)

lemma stereographic-North: stereographic x = ∞_h ⟷ x = North
proof (transfer)
  fix x
  show stereographic-coords x ≈ inf-homo-rep ⟷ x = North
  proof
    assume x = North
    thus stereographic-coords x ≈ inf-homo-rep
     by (simp add: stereographic-coords-def Abs-riemann-sphere-inverse Abs-homo-coords-inverse)
  next
    assume ∗: stereographic-coords x ≈ inf-homo-rep
    show x = North
    proof (cases Rep-riemann-sphere x = (0, 0, 1))
      case True
      thus ?thesis
        by auto (metis Rep-riemann-sphere-inverse)
    next
      case False
      thus ?thesis
        using ∗
        using Rep-riemann-sphere[of x]
      by (auto simp add: stereographic-coords-def split-def Let-def Abs-homo-coords-inverse
complex-of-real-def split: split-if-asm) (metis pair-collapse)
    qed
  qed
qed

lemma stereographic-inv-stereographic':
  assumes
  z: z = z1/z2 and z2 ≠ 0 and
  X: X = Re (2∗z / (1 + z∗cnj z)) and Y: Y = Im (2∗z / (1 + z∗cnj z)) and
Z: Z = ((cmod z)^2 − 1) / (1 + (cmod z)^2)
  shows ∃ k. k ≠ 0 ∧ (Complex X Y, complex-of-real (1 − Z)) = k ∗_sv (z1, z2)
proof−
  have 1 + (cmod z)^2 ≠ 0
    by (metis one-power2 sum-power2-eq-zero-iff zero-neq-one)
  hence cor (1 − Z) = 2 / cor (1 + (cmod z)^2)
    using Z
    by (simp add: field-simps complex-of-real-def)
  moreover
  have X = 2 ∗ Re(z) / (1 + (cmod z)^2)
    using X
    by (simp add: Re-stereographic)
  have Y = 2 ∗ Im(z) / (1 + (cmod z)^2)
    using Y
    by (simp add: Im-stereographic)
  have Complex X Y = 2 * z / cor (1 + (cmod z)^2)
    using ‹1 + (cmod z)^2 ≠ 0›
     by (subst ‹X = 2*Re(z) / (1 + (cmod z)^2)›, subst ‹Y = 2*Im(z) / (1 +
(cmod z)^2)›, simp add: Complex-scale4 Complex-scale1 of-real-numeral)
  moreover
  have 1 + (cor (cmod (z1 / z2)))^2 ≠ 0
    by (rule one-plus-square-neq-zero)
  ultimately
  show ?thesis
    using ‹z2 ≠ 0› ‹1 + (cmod z)^2 ≠ 0›
    by (simp, subst z)+
       (rule-tac x=(2 / (1 + (cor (cmod (z1 / z2)))^2)) / z2 in exI, auto)
qed

lemma
  stereographic-inv-stereographic:
  stereographic (inv-stereographic z) = z
proof transfer
  fix z
  obtain z1 z2 where zz: Rep-homo-coords z = (z1, z2)
    by (rule obtain-homo-coords)
  have z ≈ stereographic-coords (inv-stereographic-coords z)
  proof (cases z2 = 0)
    case True
    thus ?thesis
      using zz Rep-homo-coords[of z]
      by (simp add: stereographic-coords-def inv-stereographic-coords-Rep)
  next
    case False
    thus ?thesis
      using zz stereographic-inv-stereographic'[of z1/z2 z1 z2]
      by (simp add: stereographic-coords-rep inv-stereographic-coords-Rep Let-def)
  qed
  thus stereographic-coords (inv-stereographic-coords z) ≈ z
    by (rule homo-coords-eq-sym)
qed

lemma bij-stereographic: bij stereographic
unfolding bij-def inj-on-def surj-def
proof (safe)
  fix x y
  assume stereographic x = stereographic y
  thus x = y
  proof (transfer)
    fix a b
    assume *: stereographic-coords a ≈ stereographic-coords b
     obtain xa ya za xb yb zb where **: Rep-riemann-sphere a = (xa, ya, za)
Rep-riemann-sphere b = (xb, yb, zb)
      by (metis prod-cases3)
    show a = b
    proof (subst Rep-riemann-sphere-inject[symmetric])
      show Rep-riemann-sphere a = Rep-riemann-sphere b
      proof (cases Rep-riemann-sphere a = (0, 0, 1))
        case True
        thus ?thesis
          using ∗ ∗∗ Rep-riemann-sphere[of b]
          unfolding stereographic-coords-def
       by (cases zb=1) (auto simp add: Abs-homo-coords-inverse complex-of-real-def)
      next
        {
          fix k
          assume xa ∗ xa + (ya ∗ ya + za ∗ za) = 1
                 zb ∗ zb + (k ∗ (k ∗ (xa ∗ xa)) + k ∗ (k ∗ (ya ∗ ya))) = 1
                 zb ≠ 1 za ≠ 1 k ≠ 0 1 + k ∗ za = k + zb k ≠ 1
          hence False
            by algebra
        } note ∗∗∗ = this

        case False
        thus ?thesis
          using ∗ ∗∗ Rep-riemann-sphere[of a]  Rep-riemann-sphere[of b]
          unfolding stereographic-coords-def
          apply (case-tac[!] zb = 1, case-tac[!] za = 1)
          apply (auto simp add: Abs-homo-coords-inverse complex-of-real-def)
          apply (case-tac[!] k)
          using ∗∗∗
          apply (auto simp add: field-simps)
          apply (case-tac real1 = 1)
          by auto
      qed
    qed
  qed
next
  fix a
  show ∃ b. a = stereographic b
   by (rule-tac x=inv-stereographic a in exI) (simp add: stereographic-inv-stereographic)
qed

lemma inv-stereographic-stereographic:
  inv-stereographic (stereographic x) = x
using stereographic-inv-stereographic[of stereographic x]
using bij-stereographic
unfolding bij-def inj-on-def
by simp

lemma inv-stereographic-is-inv:
  inv-stereographic = inv stereographic
```

```
by (rule inv-equality[symmetric], simp-all add: inv-stereographic-stereographic stereographic-inv-stereographic)
```

Circles on the sphere

```
type-synonym real-vec-4 = real × real × real × real

fun mult-sv :: real ⇒ real-vec-4 ⇒ real-vec-4 (infixl *_{sv4} 100) where
  k *_{sv4} (a, b, c, d) = (k∗a, k∗b, k∗c, k∗d)

typedef plane-vec = {(a::real, b::real, c::real, d::real). a ≠ 0 ∨ b ≠ 0 ∨ c ≠ 0 ∨
d ≠ 0}
by (rule-tac x=(1, 1, 1, 1) in exI) simp

definition plane-vec-eq where
  plane-vec-eq v1 v2 ⟷ (∃ k. k ≠ 0 ∧ Rep-plane-vec v2 = k *_{sv4} Rep-plane-vec
v1)

lemma [simp]: 1 *_{sv4} x = x
by (cases x) simp

lemma [simp]: x *_{sv4} (y *_{sv4} v) = (x∗y) *_{sv4} v
by (cases v) simp

quotient-type plane = plane-vec / plane-vec-eq
proof (rule equivpI)
  show reflp plane-vec-eq
    unfolding reflp-def
    by (auto simp add: plane-vec-eq-def) (rule-tac x=1 in exI, simp)
next
  show symp plane-vec-eq
    unfolding symp-def
    by (auto simp add: plane-vec-eq-def) (rule-tac x=1/k in exI, simp)
next
  show transp plane-vec-eq
    unfolding transp-def
    by (auto simp add: plane-vec-eq-def) (rule-tac x=ka∗k in exI, simp)
qed

definition on-sphere-circle-rep where
  on-sphere-circle-rep α A ⟷
      (let (X, Y, Z) = Rep-riemann-sphere A;
           (a, b, c, d) = Rep-plane-vec α
        in a∗X + b∗Y + c∗Z + d = 0)

lift-definition on-sphere-circle :: plane ⇒ riemann-sphere ⇒ bool is on-sphere-circle-rep
proof−
  fix v1 v2
  obtain a1 b1 c1 d1 where vv1: Rep-plane-vec v1 = (a1, b1, c1, d1)
    by (cases Rep-plane-vec v1) auto
  obtain a2 b2 c2 d2 where vv2: Rep-plane-vec v2 = (a2, b2, c2, d2)
```

```
    by (cases Rep-plane-vec v2) auto
  assume plane-vec-eq v1 v2
  then obtain k where *: a2 = k*a1 b2 = k*b1 c2 = k*c1 d2 = k*d1 k ≠ 0
    using vv1 vv2
    by (auto simp add: plane-vec-eq-def)
  show on-sphere-circle-rep v1 = on-sphere-circle-rep v2
  proof (rule ext)
    fix M
    obtain x y z where MM: Rep-riemann-sphere M = (x, y, z)
      by (cases Rep-riemann-sphere M) auto
    have k * a1 * x + k * b1 * y + k * c1 * z + k * d1 = k*(a1*x + b1*y +
c1*z + d1)
      by (simp add: field-simps)
    thus on-sphere-circle-rep v1 M = on-sphere-circle-rep v2 M
      using vv1 vv2 MM *
      by (auto simp add: plane-vec-eq-def on-sphere-circle-rep-def split-def Let-def)
  qed
qed

definition sphere-circle-set where
  sphere-circle-set α = {A. on-sphere-circle α A}
```

Distance on the Riemann sphere

```
definition dist-riemann-sphere' where
  dist-riemann-sphere' M1 M2 =
     (let (x1, y1, z1) = Rep-riemann-sphere M1;
          (x2, y2, z2) = Rep-riemann-sphere M2
       in norm (x1 - x2, y1 - y2, z1 - z2))

lemma dist-riemann-sphere'-inner:
  (dist-riemann-sphere' M1 M2)^2 = 2 - 2 * inner (Rep-riemann-sphere M1)
(Rep-riemann-sphere M2)
using Rep-riemann-sphere[of M1] Rep-riemann-sphere[of M2]
unfolding dist-riemann-sphere'-def
by (auto simp add: norm-prod-def) (simp add: power2-eq-square field-simps)

lemma xxx [simp]:
  Re (2 * m1 / (1 + cor ((cmod m1)^2))) = 2 * Re m1 / (1 + (cmod m1)^2)
apply (subst Re-divide-real)
apply (simp add: power2-eq-square)
apply (metis numeral-One of-real-1 of-real-add of-real-eq-0-iff power-one sum-power2-eq-zero-iff
zero-neq-numeral)
apply (simp add: power2-eq-square)
done

lemma yyy [simp]:
  Im (2 * m1 / (1 + cor ((cmod m1)^2))) = 2 * Im m1 / (1 + (cmod m1)^2)
apply (subst Im-divide-real)
apply (simp add: power2-eq-square)
apply (metis numeral-One of-real-1 of-real-add of-real-eq-0-iff power-one sum-power2-eq-zero-iff
zero-neq-numeral)
apply (simp add: power2-eq-square)
done

lemma dist-riemann-sphere'-ge-0 [simp]: dist-riemann-sphere' M1 M2 ≥ 0
using norm-ge-zero
unfolding dist-riemann-sphere'-def
by (simp add: split-def Let-def)

lemma dist-homo-stereographic-finite:
  assumes stereographic M1 = of-complex m1  stereographic M2 = of-complex m2
  shows dist-riemann-sphere' M1 M2 = 2 ∗ cmod (m1 − m2) / (sqrt (1 + (cmod
m1)^2) ∗ sqrt (1 + (cmod m2)^2))
proof−
  obtain x1 y1 z1 x2 y2 z2 where MM: (x1, y1, z1) = Rep-riemann-sphere M1
(x2, y2, z2) = Rep-riemann-sphere M2
    by (cases Rep-riemann-sphere M1, cases Rep-riemann-sphere M2, auto, blast)
  have ∗: M1 = inv-stereographic (of-complex m1)  M2 = inv-stereographic (of-complex
m2)
    using inv-stereographic-is-inv assms
    by (metis inv-stereographic-stereographic)+
  have (1 + (cmod m1)^2) ≠ 0  (1 + (cmod m2)^2) ≠ 0
    by (metis power-one realpow-two-sum-zero-iff zero-neq-one)+
  have (1 + (cmod m1)^2) > 0  (1 + (cmod m2)^2) > 0
    by (smt realpow-square-minus-le)+
  hence (1 + (cmod m1)^2) ∗ (1 + (cmod m2)^2) > 0
    by (metis norm-mult-less norm-zero power2-eq-square zero-power2)
  hence sqrt ((1 + cmod m1 ∗ cmod m1) ∗ (1 + cmod m2 ∗ cmod m2)) > 0
    using real-sqrt-gt-0-iff
    by (simp add: power2-eq-square)
  hence ∗∗: (2 ∗ cmod (m1 − m2) / sqrt ((1 + cmod m1 ∗ cmod m1) ∗ (1 +
cmod m2 ∗ cmod m2))) ≥ 0 ⟷ cmod (m1 − m2) ≥ 0
    by (metis diff-self divide-nonneg-pos mult-2 norm-ge-zero norm-triangle-ineq4
norm-zero)

  have (dist-riemann-sphere' M1 M2)^2 ∗ (1 + (cmod m1)^2) ∗ (1 + (cmod m2)^2)
= 4 ∗ (cmod (m1 − m2))^2
    apply (subst ∗)+
  proof transfer
    fix m1 m2
    have (1 + (cmod m1)^2) ≠ 0  (1 + (cmod m2)^2) ≠ 0
      by (metis power-one realpow-two-sum-zero-iff zero-neq-one)+
     thus (dist-riemann-sphere' (inv-stereographic-coords (of-complex-coords m1))
(inv-stereographic-coords (of-complex-coords m2)))^2  ∗ (1 + (cmod m1)^2) ∗ (1 +
(cmod m2)^2)= 4 ∗ (cmod (m1 − m2))^2
        apply (simp add: dist-riemann-sphere'-inner inv-stereographic-coords-Rep
complex-mult-cnj-cmod)
      apply (subst cor-squared)+
      apply (subst xxx)+
      apply (subst yyy)+
      apply (subst left-diff-distrib[of 2])
      apply (subst left-diff-distrib[of 2*(1+(cmod m1)^2)])
      apply (subst distrib-right[of - - (1 + (cmod m1)^2)])
      apply (subst distrib-right[of - - (1 + (cmod m1)^2)])
      apply (subst distrib-right[of 2 * (2 * Re m1 / (1 + (cmod m1)^2) * (2 * Re
m2 / (1 + (cmod m2)^2))) * (1 + (cmod m1)^2) - (1 + (cmod m2)^2)])
      apply (subst distrib-right[of 2 * (2 * Im m1 / (1 + (cmod m1)^2) * (2 * Im
m2 / (1 + (cmod m2)^2))) * (1 + (cmod m1)^2) - (1 + (cmod m2)^2)])
      apply simp
      apply (subst (asm) cmod-square)+
      apply (subst cmod-square)+
      apply (simp add: field-simps)
      done
  qed
  hence (dist-riemann-sphere' M1 M2)^2 = 4 * (cmod (m1 − m2))^2 / ((1 + (cmod
m1)^2) * (1 + (cmod m2)^2))
      using ‹(1 + (cmod m1)^2) ≠ 0› ‹(1 + (cmod m2)^2) ≠ 0›
      using eq-divide-imp[of (1 + (cmod m1)^2) * (1 + (cmod m2)^2) (dist-riemann-sphere'
M1 M2)^2 4 * (cmod (m1 − m2))^2]
    by simp
  thus dist-riemann-sphere' M1 M2 = 2 * cmod (m1 − m2) / (sqrt (1 + (cmod
m1)^2) * sqrt (1 + (cmod m2)^2))
    using power2-eq-iff[of dist-riemann-sphere' M1 M2 2 * (cmod (m1 − m2)) /
sqrt ((1 + (cmod m1)^2) * (1 + (cmod m2)^2))]
    using ‹(1 + (cmod m1)^2) * (1 + (cmod m2)^2) > 0› ‹(1 + (cmod m1)^2) > 0›
‹(1 + (cmod m2)^2) > 0›
    apply (auto simp add: power2-eq-square real-sqrt-mult[symmetric])
    using dist-riemann-sphere'-ge-0[of M1 M2] **
    by simp
qed

lemma dist-homo-stereographic-infinite:
  assumes stereographic M1 = ∞_h  stereographic M2 = of-complex m2
  shows dist-riemann-sphere' M1 M2 = 2 / sqrt (1 + (cmod m2)^2)
proof−
  obtain x2 y2 z2 where MM: (0, 0, 1) = Rep-riemann-sphere M1 (x2, y2, z2)
= Rep-riemann-sphere M2
    using ‹stereographic M1 = ∞_h›
    using stereographic-North[of M1]
    by (cases Rep-riemann-sphere M2, auto simp add: Abs-riemann-sphere-inverse)
  have *: M1 = inv-stereographic ∞_h  M2 = inv-stereographic (of-complex m2)
    using inv-stereographic-is-inv assms
    by (metis inv-stereographic-stereographic)+
  have (1 + (cmod m2)^2) ≠ 0
    by (metis power-one realpow-two-sum-zero-iff zero-neq-one)+
  have (1 + (cmod m2)^2) > 0
    by (smt realpow-square-minus-le)+
  hence sqrt (1 + cmod m2 * cmod m2) > 0
    using real-sqrt-gt-0-iff
    by (simp add: power2-eq-square)
  hence **: 2 / sqrt (1 + cmod m2 * cmod m2) > 0
    by simp

  have (dist-riemann-sphere' M1 M2)^2 * (1 + (cmod m2)^2) = 4
    apply (subst *)+
  proof transfer
    fix m2
    have (1 + (cmod m2)^2) ≠ 0
      by (metis power-one realpow-two-sum-zero-iff zero-neq-one)
    thus (dist-riemann-sphere' (inv-stereographic-coords inf-homo-rep) (inv-stereographic-coords
(of-complex-coords m2)))^2 * (1 + (cmod m2)^2) = 4
      by (simp add: dist-riemann-sphere'-inner inv-stereographic-coords-Rep complex-mult-cnj-cmod)
         (subst left-diff-distrib[of 2], simp)
  qed
  hence (dist-riemann-sphere' M1 M2)^2 = 4 / (1 + (cmod m2)^2)
    using ‹(1 + (cmod m2)^2) ≠ 0›
    by (simp add: field-simps)
  thus dist-riemann-sphere' M1 M2 = 2 / sqrt (1 + (cmod m2)^2)
    using power2-eq-iff[of dist-riemann-sphere' M1 M2 2 / sqrt (1 + (cmod m2)^2)]
    using ‹(1 + (cmod m2)^2) > 0›
    apply (auto simp add: power2-eq-square real-sqrt-mult[symmetric])
    using dist-riemann-sphere'-ge-0[of M1 M2] **
    by simp
qed

lemma dist-riemann-sphere'-sym: dist-riemann-sphere' M1 M2 = dist-riemann-sphere'
M2 M1
proof-
  obtain x1 y1 z1 x2 y2 z2 where MM: (x1, y1, z1) = Rep-riemann-sphere M1
(x2, y2, z2) = Rep-riemann-sphere M2
    by (cases Rep-riemann-sphere M1, cases Rep-riemann-sphere M2, auto, blast)
  show ?thesis
    unfolding dist-riemann-sphere'-def
    using norm-minus-cancel[of (x1 - x2, y1 - y2, z1 - z2)] MM[symmetric]
    by simp
qed

lemma dist-homo-stereographic: dist-riemann-sphere' M1 M2 = dist-homo (stereographic
M1) (stereographic M2)
proof (cases M1 = North)
  case True
  hence stereographic M1 = ∞_h
    by (simp add: stereographic-North)
  show ?thesis
  proof (cases M2 = North)
    case True
```

show *?thesis*
using ⟨*M1* = *North*⟩ ⟨*M2* = *North*⟩
by (*auto simp add*: *Abs-riemann-sphere-inverse dist-riemann-sphere′-def norm-prod-def*)
next
case *False*
hence *stereographic M2* $\neq \infty_h$
using *stereographic-North*[*of M2*]
by *simp*
then obtain *m2* **where** *stereographic M2* = *of-complex m2*
using *inf-homo-or-complex-homo*[*of stereographic M2*]
by *auto*
show *?thesis*
using ⟨*stereographic M2* = *of-complex m2*⟩ ⟨*stereographic M1* = ∞_h⟩
using *dist-homo-infinite1 dist-homo-stereographic-infinite*
by *simp*
qed
next
case *False*
hence *stereographic M1* $\neq \infty_h$
by (*simp add*: *stereographic-North*)
then obtain *m1* **where** *stereographic M1* = *of-complex m1*
using *inf-homo-or-complex-homo*[*of stereographic M1*]
by *auto*
show *?thesis*
proof (*cases M2* = *North*)
case *True*
hence *stereographic M2* = ∞_h
by (*simp add*: *stereographic-North*)
show *?thesis*
using ⟨*stereographic M1* = *of-complex m1*⟩ ⟨*stereographic M2* = ∞_h⟩
using *dist-homo-infinite2 dist-homo-stereographic-infinite*
by (*subst dist-riemann-sphere′-sym*, *simp*)
next
case *False*
hence *stereographic M2* $\neq \infty_h$
by (*simp add*: *stereographic-North*)
then obtain *m2* **where** *stereographic M2* = *of-complex m2*
using *inf-homo-or-complex-homo*[*of stereographic M2*]
by *auto*
show *?thesis*
using ⟨*stereographic M1* = *of-complex m1*⟩ ⟨*stereographic M2* = *of-complex m2*⟩
using *dist-homo-finite dist-homo-stereographic-finite*
by *simp*
qed
qed

lemma *dist-homo-stereographic′*:

```
  dist-homo A B = dist-riemann-sphere′ (inv-stereographic A) (inv-stereographic
B)
by (subst dist-homo-stereographic) (metis stereographic-inv-stereographic)

instantiation riemann-sphere :: metric-space
begin
definition dist-riemann-sphere = dist-riemann-sphere′
definition open-riemann-sphere S = (∀x∈S. ∃e>0. ∀y. dist-riemann-sphere′ y x
< e ⟶ y ∈ S)
instance
proof
  fix x y :: riemann-sphere
  show (dist x y = 0) = (x = y)
  proof−
    obtain x1 y1 z1 x2 y2 z2 where MM: (x1, y1, z1) = Rep-riemann-sphere x
(x2, y2, z2) = Rep-riemann-sphere y
      by (cases Rep-riemann-sphere x, cases Rep-riemann-sphere y, auto, blast)
    show ?thesis
      unfolding dist-riemann-sphere-def
        using norm-eq-zero[of (x1 − y2, y1 − y2, z1 − z2)] MM[symmetric]
Rep-riemann-sphere-inject[of x y]
      by (simp add: dist-riemann-sphere′-def) (smt prod.inject zero-prod-def)
  qed
next
  fix S :: riemann-sphere set
  show open S = (∀x∈S. ∃e>0. ∀y. dist y x < e ⟶ y ∈ S)
    unfolding open-riemann-sphere-def dist-riemann-sphere-def
    by simp
next
  fix x y z :: riemann-sphere
  show dist x y ≤ dist x z + dist y z
  proof−
   obtain x1 y1 z1 x2 y2 z2 x3 y3 z3 where MM: (x1, y1, z1) = Rep-riemann-sphere
x (x2, y2, z2) = Rep-riemann-sphere y (x3, y3, z3) = Rep-riemann-sphere z
     by (cases Rep-riemann-sphere x, cases Rep-riemann-sphere y, cases Rep-riemann-sphere
z, auto, blast)
    show ?thesis
      unfolding dist-riemann-sphere-def
      using MM[symmetric] norm-minus-cancel[of (x3 − x2, y3 − y2, z3 − z2)]
norm-triangle-ineq[of (x1 − x3, y1 − y3, z1 − z3) (x3 − x2, y3 − y2, z3 − z2)]
      by (simp add: dist-riemann-sphere′-def field-simps)
  qed
qed

end

lemma ex-cos-gt′:
  assumes a ≥ 0 a < 1 −pi/2 ≤ α ∧ α ≤ pi/2
  shows ∃ α′. −pi/2 ≤ α′ ∧ α′ ≤ pi/2 ∧ α′ ≠ α ∧ cos (α − α′) = a
```

```
proof−
  have arccos a > 0 arccos a ≤ pi/2
    using ‹a ≥ 0› ‹a < 1›
    using arccos-lt-bounded arccos-le-pi2
    by auto

  show ?thesis
  proof (cases α − arccos a ≥ − pi/2)
    case True
    thus ?thesis
      using assms ‹arccos a > 0› ‹arccos a ≤ pi/2›
      by (rule-tac x = α − arccos a in exI) auto
  next
    case False
    thus ?thesis
      using assms ‹arccos a > 0› ‹arccos a ≤ pi/2›
      by (rule-tac x = α + arccos a in exI) auto
  qed
qed

lemma ex-cos-gt:
  assumes a < 1 −pi/2 ≤ α ∧ α ≤ pi/2
  shows ∃ α′. −pi/2 ≤ α′ ∧ α′ ≤ pi/2 ∧ α′ ≠ α ∧ cos (α − α′) > a
proof−
  have ∃ a′. a′ ≥ 0 ∧ a′ > a ∧ a′ < 1
    using ‹a < 1›
    using divide-strict-right-mono[of 2∗a + (1 − a) 2 2]
     by (rule-tac x=if a < 0 then 0 else a + (1−a)/2 in exI) (auto simp add: field-simps)
  then obtain a′ where a′ ≥ 0 a′ > a a′ < 1
    by auto
  thus ?thesis
    using ex-cos-gt′[of a′ α] assms
    by auto
qed

instantiation riemann-sphere :: perfect-space
begin
instance proof
  fix M :: riemann-sphere
  obtain x y z where MM: Rep-riemann-sphere M = (x, y, z)
    by (cases Rep-riemann-sphere M) auto
  then obtain α β where ∗: x = cos α ∗ cos β y = cos α ∗ sin β z = sin α −pi / 2 ≤ α ∧ α ≤ pi / 2
    using Rep-riemann-sphere[of M]
    using ex-sphere-params[of x y z]
    by auto
  show ¬ open {M}
    unfolding open-riemann-sphere-def
  proof auto
    fix e :: real
    assume e > 0
    then obtain α' where 1 − (e∗e/2) < cos (α − α') α ≠ α' −pi/2 ≤ α' α' ≤
pi/2
      using ex-cos-gt[of 1 − (e∗e/2) α] ‹− pi / 2 ≤ α ∧ α ≤ pi / 2›
      by (auto simp add: mult-pos-pos)
    hence sin α ≠ sin α'
      using ‹−pi / 2 ≤ α ∧ α ≤ pi / 2› sin-inj[of α α']
      by auto

    have 2 − 2 ∗ cos (α − α') < e∗e
      using mult-strict-right-mono[OF ‹1 − (e∗e/2) < cos (α − α')›, of 2]
      by (simp add: field-simps)
    have 2 − 2 ∗ cos (α − α') ≥ 0
      using cos-le-one[of α − α']
      by (simp add: sign-simps)
    let ?M' = Abs-riemann-sphere (cos α' ∗ cos β,  cos α' ∗ sin β, sin α')
     have dist-riemann-sphere' M ?M' = sqrt ((cos α − cos α')^2 + (sin α − sin
α')^2)
      using MM ∗ sphere-params-on-sphere[of - α' β]
      using sin-cos-squared-add[of β]
     apply (simp add: dist-riemann-sphere'-def Abs-riemann-sphere-inverse norm-prod-def)
      apply (subst left-diff-distrib[symmetric])+
      apply (subst power-mult-distrib)+
      apply (subst distrib-left[symmetric])
      apply simp
      done
    also have ... = sqrt (2 − 2∗cos (α − α'))
      by (simp add: power2-eq-square field-simps cos-diff)
    finally
    have (dist-riemann-sphere' M ?M')^2 = 2 − 2∗cos (α − α')
      using ‹2 − 2 ∗ cos (α − α') ≥ 0›
      by simp
    hence (dist-riemann-sphere' M ?M')^2 < e^2
      using ‹2 − 2 ∗ cos (α − α') < e∗e›
      by (simp add: power2-eq-square)
    hence dist-riemann-sphere' M ?M' < e
      apply (rule power2-less-imp-less)
      using ‹e > 0›
      by simp
    moreover
    have M ≠ ?M'
      apply (subst Rep-riemann-sphere-inverse[symmetric])
       using Abs-riemann-sphere-inject[of Rep-riemann-sphere M (cos α' ∗ cos β,
cos α' ∗ sin β, sin α') ]
     using MM MM[symmetric] ∗ sphere-params-on-sphere[of - α' β] Rep-riemann-sphere[of
M] ‹sin α ≠ sin α'›
      by (simp add: Abs-riemann-sphere-inverse)
    ultimately
    show ∃ y. dist-riemann-sphere' y M < e ∧ y ≠ M
      by (rule-tac x=?M' in exI) (simp add: dist-riemann-sphere'-sym)
  qed
qed
end

instantiation complex-homo :: perfect-space
begin
instance proof
  fix x::complex-homo
  show ¬ open {x}
    unfolding open-complex-homo-def[of {x}]
  proof (auto)
    fix e::real
    assume e > 0
    thus ∃ y. dist-homo y x < e ∧ y ≠ x
      using not-open-singleton[of inv-stereographic x]
      unfolding open-riemann-sphere-def[of {inv-stereographic x}]
      apply (subst dist-homo-stereographic', auto)
      apply (erule-tac x=e in allE, auto)
     apply (rule-tac x=stereographic y in exI, auto simp add: inv-stereographic-stereographic)
      done
  qed
qed

end

lemma continuous-on UNIV stereographic
unfolding continuous-on-iff
unfolding dist-complex-homo-def dist-riemann-sphere-def
by (subst dist-homo-stereographic', auto simp add: inv-stereographic-stereographic)

lemma continuous-on UNIV inv-stereographic
unfolding continuous-on-iff
unfolding dist-complex-homo-def dist-riemann-sphere-def
by (subst dist-homo-stereographic) (auto simp add: stereographic-inv-stereographic)

end
```

10 Moebius transformations

```
theory Moebius
imports HomogeneousCoordinates
begin

typedef moebius-mat = {M::complex-mat. mat-det M ≠ 0}
by (rule-tac x=eye in exI, simp)
```

definition *moebius-mat-eq* **where**
[*simp*]: *moebius-mat-eq A B* $\longleftrightarrow$ ($\exists$ *k*::*complex*. $k \neq 0 \wedge$ *Rep-moebius-mat B* = $k *_{sm}$ (*Rep-moebius-mat A*))

lemma [*simp*]: *moebius-mat-eq x x*
by (*simp*, *rule-tac x=1* **in** *exI*, *simp*)

quotient-type *moebius* = *moebius-mat* / *moebius-mat-eq*
proof (*rule equivpI*)
show *reflp moebius-mat-eq*
by (*auto simp add*: *reflp-def*, *rule-tac x=1* **in** *exI*, *simp*)
next
show *symp moebius-mat-eq*
by (*auto simp add*: *symp-def*, *rule-tac x=1/k* **in** *exI*, *simp*)
next
show *transp moebius-mat-eq*
by (*auto simp add*: *transp-def*, *rule-tac x=ka*k* **in** *exI*, *simp*)
qed

definition *mk-moebius-rep* **where**
mk-moebius-rep a b c d = *Abs-moebius-mat* (*a*, *b*, *c*, *d*)

lift-definition *mk-moebius* :: *complex* ⇒ *complex* ⇒ *complex* ⇒ *complex* ⇒ *moebius* **is** *mk-moebius-rep*
by (*simp del*: *moebius-mat-eq-def*)

lemma *mk-moebius-rep-Rep*:
assumes *mat-det* (*a*, *b*, *c*, *d*) $\neq 0$
shows *Rep-moebius-mat* (*mk-moebius-rep a b c d*) = (*a*, *b*, *c*, *d*)
using *assms*
by (*simp add*: *mk-moebius-rep-def Abs-moebius-mat-inverse*)

lemma *ex-mk-moebius*:
shows $\exists$ *a b c d*. *M* = *mk-moebius a b c d* $\wedge$ *mat-det* (*a*, *b*, *c*, *d*) $\neq 0$
proof *transfer*
fix *M*
obtain *a b c d* **where** *Rep-moebius-mat M* = (*a*, *b*, *c*, *d*)
by (*cases Rep-moebius-mat M*) *auto*
hence *moebius-mat-eq M* (*mk-moebius-rep a b c d*) $\wedge$ *mat-det* (*a*, *b*, *c*, *d*) $\neq 0$
using *Rep-moebius-mat*[*of M*]
by (*simp add*: *mk-moebius-rep-Rep*, *rule-tac x=1* **in** *exI*, *simp*)
thus $\exists$ *a b c d*. *moebius-mat-eq M* (*mk-moebius-rep a b c d*) $\wedge$ *mat-det* (*a*, *b*, *c*, *d*) $\neq 0$
by *blast*
qed

10.1 Action on points

definition *moebius-pt-rep* :: *moebius-mat* ⇒ *homo-coords* ⇒ *homo-coords* **where**

moebius-pt-rep M z =
  (*let z* = *Rep-homo-coords z*;
    M = *Rep-moebius-mat M*
   in Abs-homo-coords (*M* $*_{mv}$ *z*))

lemma [*simp*]: *Rep-homo-coords* (*Abs-homo-coords* (*Rep-moebius-mat M* $*_{mv}$ *Rep-homo-coords x*)) = *Rep-moebius-mat M* $*_{mv}$ *Rep-homo-coords x*
 using *Rep-moebius-mat*[*of M*] *Rep-homo-coords*[*of x*] *mult-mv-nonzero*[*of Rep-homo-coords x Rep-moebius-mat M*]
by (*simp add*: *Abs-homo-coords-inverse*)

lemma [*simp*]: *Rep-homo-coords* (*moebius-pt-rep M z*) = *Rep-moebius-mat M* $*_{mv}$ *Rep-homo-coords z*
by (*simp add*: *moebius-pt-rep-def*)

lift-definition *moebius-pt* :: *moebius* ⇒ *complex-homo* ⇒ *complex-homo* **is** *moebius-pt-rep*
proof−
 fix *M M' x x'*
 assume *moebius-mat-eq M M' x* ≈ *x'*
 thus *moebius-pt-rep M x* ≈ *moebius-pt-rep M' x'*
  by (*cases Rep-moebius-mat M*, *cases Rep-homo-coords x*, *auto simp add*: *field-simps*) (*rule-tac x*=*k*∗*ka* **in** *exI*, *simp*)
qed

lemma *bij-moebius-pt*:
 shows *bij* (*moebius-pt M*)
unfolding *bij-def inj-on-def surj-def*
proof (*simp*, *transfer*, *safe*)
 fix *M x y*
 assume *moebius-pt-rep M x* ≈ *moebius-pt-rep M y*
 thus *x* ≈ *y*
  using *Rep-moebius-mat*[*of M*]
  apply *auto*
  apply (*subst* (*asm*) *mult-sv-mv*)
  using *mult-mv-cancel-l*
  by *blast*
next
 fix *M y*
 let *?M* = *Rep-moebius-mat M*
 let *?iM* = *mat-inv ?M*
 let *?y* = *Rep-homo-coords y*
 show ∃ *x*. *y* ≈ *moebius-pt-rep M x*
  using *Rep-moebius-mat*[*of M*] *mat-det-inv*[*of ?M*] *Rep-homo-coords*[*of y*] *mult-mv-nonzero*[*of ?y ?iM*]
  using *mat-inv-r*[*of ?M*] *eye-mv-l*[*of ?y*]
  by (*auto*, *rule-tac x*=*Abs-homo-coords* ((*mat-inv* (*Rep-moebius-mat M*)) $*_{mv}$

```
Rep-homo-coords y) in exI, rule-tac x=1 in exI)
      (auto simp add: Abs-homo-coords-inverse)
qed

definition is-moebius where
  is-moebius f ⟷ (∃ M. f = moebius-pt M)
```

Bilinear and linear expressions

```
lemma moebius-bilinear:
  assumes mat-det (a, b, c, d) ≠ 0
  shows moebius-pt (mk-moebius a b c d) z =
          (if z ≠ ∞_h then
               ((of-complex a) *_h z +_h (of-complex b)) :_h
               ((of-complex c) *_h z +_h (of-complex d))
           else
               (of-complex a) :_h
               (of-complex c))
unfolding divide-homo-def
using assms
proof (transfer)
  fix a b c d :: complex and z
  obtain z1 z2 where zz: Rep-homo-coords z = (z1, z2)
    by (rule obtain-homo-coords)
  assume *: mat-det (a, b, c, d) ≠ 0
  let ?oc = of-complex-coords
  show moebius-pt-rep (mk-moebius-rep a b c d) z ≈
       (if ¬ z ≈ inf-homo-rep
        then ?oc a *_hc z +_hc ?oc b *_hc
             reciprocal-homo-coords (?oc c *_hc z +_hc ?oc d)
        else ?oc a *_hc
             reciprocal-homo-coords (of-complex-coords c))
  proof (cases z ≈ inf-homo-rep)
    case True
    thus ?thesis
      using zz *
      using mult-homo-coords-Rep[of ?oc a a 1 reciprocal-homo-coords (?oc c) 1 c]
      using reciprocal-homo-coords-Rep[of ?oc c]
      by (force simp add: mk-moebius-rep-Rep field-simps)
  next
    case False
    hence z2 ≠ 0
      using zz Rep-homo-coords[of z]
     by auto (metis mult.commute complex-divide-def mult-zero-right right-inverse-eq)
    thus ?thesis
      using zz * False
      using regular-homogenous-system[of a d b c z1 z2]
      apply simp
      apply (subst mult-homo-coords-Rep[of ?oc a *_hc z +_hc ?oc b a*z1+b*z2 z2
reciprocal-homo-coords (?oc c *_hc z +_hc ?oc d) z2 c*z1+d*z2])
```

using *add-homo-coords-Rep[of ?oc a* $*_{hc}$ *z a*∗*z1 z2 ?oc b b 1]*
using *mult-homo-coords-Rep[of ?oc a a 1 z z1 z2]*
using *reciprocal-homo-coords-Rep[of ?oc c* $*_{hc}$ *z* $+_{hc}$ *?oc d]*
using *add-homo-coords-Rep[of ?oc c* $*_{hc}$ *z c*∗*z1 z2 ?oc d d 1]*
using *mult-homo-coords-Rep[of ?oc c c 1 z z1 z2]*
by (*auto simp add*: *mk-moebius-rep-Rep*)
qed
qed

10.2 Moebius group

definition *moebius-inv-rep* **where**
moebius-inv-rep M =
(*let M* = *Rep-moebius-mat M*
in Abs-moebius-mat (*mat-inv M*))

lemma [*simp*]: *Rep-moebius-mat* (*Abs-moebius-mat* (*mat-inv* (*Rep-moebius-mat M*))) = *mat-inv* (*Rep-moebius-mat M*)
using *Rep-moebius-mat[of M] mat-det-inv[of Rep-moebius-mat M]*
by (*auto simp add*: *Abs-moebius-mat-inverse*)

lemma [*simp*]: *Rep-moebius-mat* (*moebius-inv-rep M*) = *mat-inv* (*Rep-moebius-mat M*)
by (*simp add*: *moebius-inv-rep-def*)

lift-definition *moebius-inv* :: *moebius* ⇒ *moebius* **is** *moebius-inv-rep*
proof−
fix *x y*
assume *moebius-mat-eq x y*
thus *moebius-mat-eq* (*moebius-inv-rep x*) (*moebius-inv-rep y*)
by (*auto simp add*: *mat-inv-mult-sm*) (*rule-tac x*=*1*/*k* **in** *exI*, *simp*)
qed

lemma *moebius-inv*: *moebius-pt* (*moebius-inv M*) = *inv* (*moebius-pt M*)
proof (*rule inv-equality[symmetric]*)
fix *x*
show *moebius-pt* (*moebius-inv M*) (*moebius-pt M x*) = *x*
proof (*transfer*)
fix *M x*
show *moebius-pt-rep* (*moebius-inv-rep M*) (*moebius-pt-rep M x*) ≈ *x*
using *Rep-moebius-mat[of M] Rep-homo-coords[of x] eye-mv-l*
by (*simp add*: *mat-inv-l*) (*rule-tac x*=*1* **in** *exI*, *simp*)
qed
next
fix *y*
show *moebius-pt M* (*moebius-pt* (*moebius-inv M*) *y*) = *y*
proof (*transfer*)
fix *M y*
show *moebius-pt-rep M* (*moebius-pt-rep* (*moebius-inv-rep M*) *y*) ≈ *y*

```
    using Rep-moebius-mat[of M] eye-mv-l
    by (simp add: mat-inv-r) (rule-tac x=1 in exI, simp)
  qed
qed

lemma is-moebius-inv:
  assumes is-moebius m
  shows is-moebius (inv m)
  using assms
  unfolding is-moebius-def
  using moebius-inv[symmetric]
  by auto


definition moebius-comp-rep where
  moebius-comp-rep M1 M2 =
    (let M1 = Rep-moebius-mat M1;
         M2 = Rep-moebius-mat M2 in
         Abs-moebius-mat (M1 *_{mm} M2))

lemma [simp]: Rep-moebius-mat (Abs-moebius-mat ((Rep-moebius-mat M1) *_{mm}
(Rep-moebius-mat M2))) = (Rep-moebius-mat M1) *_{mm} (Rep-moebius-mat M2)
using Rep-moebius-mat[of M1] Rep-moebius-mat[of M2]
by (simp add: Abs-moebius-mat-inverse)

lemma [simp]: Rep-moebius-mat (moebius-comp-rep M1 M2) = (Rep-moebius-mat
M1) *_{mm} (Rep-moebius-mat M2)
by (simp add: moebius-comp-rep-def)

lift-definition moebius-comp :: moebius ⇒ moebius ⇒ moebius is moebius-comp-rep
by auto (rule-tac x=ka*k in exI, simp)

lemma moebius-comp: moebius-pt M1 ∘ moebius-pt M2 = moebius-pt (moebius-comp
M1 M2)
unfolding comp-def
by (rule ext, transfer) (simp, rule-tac x=1 in exI, simp)

lemma is-moebius-comp:
  assumes is-moebius m1 is-moebius m2
  shows is-moebius (m1 ∘ m2)
  using assms
  unfolding is-moebius-def
  using moebius-comp
  by auto

definition [simp]: id-moebius-rep = Abs-moebius-mat eye

lift-definition id-moebius :: moebius is id-moebius-rep
done
```

```
lemma [simp]: Rep-moebius-mat (Abs-moebius-mat (1, 0, 0, 1)) = eye
by (simp add: Abs-moebius-mat-inverse)

lemma [simp]: Rep-moebius-mat (id-moebius-rep) = eye
by simp

lemma moebius-pt id-moebius = id
unfolding id-def
apply (rule ext, transfer)
using eye-mv-l
by simp (rule-tac x=1 in exI, simp)

instantiation moebius :: group-add
begin
definition plus-moebius :: moebius ⇒ moebius ⇒ moebius where
  [simp]: plus-moebius = moebius-comp

definition uminus-moebius :: moebius ⇒ moebius where
  [simp]: uminus-moebius = moebius-inv

definition zero-moebius :: moebius where
  [simp]: zero-moebius = id-moebius

definition minus-moebius :: moebius ⇒ moebius ⇒ moebius where
  [simp]: minus-moebius A B = A + (−B)

instance proof
  fix a b c :: moebius
  show a + b + c = a + (b + c)
    unfolding plus-moebius-def
  proof (transfer)
    fix a b c
    show moebius-mat-eq (moebius-comp-rep (moebius-comp-rep a b) c) (moebius-comp-rep
a (moebius-comp-rep b c))
      using Rep-moebius-mat[of a] Rep-moebius-mat[of b] Rep-moebius-mat[of c]
      by simp (rule-tac x=1 in exI, simp add: mult-mm-assoc)
  qed
next
  fix a :: moebius
  show a + 0 = a
    unfolding plus-moebius-def zero-moebius-def
  proof (transfer)
    fix A
    show moebius-mat-eq (moebius-comp-rep A id-moebius-rep) A
      using mat-eye-r
      by simp (rule-tac x=1 in exI, simp)
  qed
next
```

```
  fix a :: moebius
  show 0 + a = a
    unfolding plus-moebius-def zero-moebius-def
  proof (transfer)
    fix A
    show moebius-mat-eq (moebius-comp-rep id-moebius-rep A) A
      using mat-eye-l
      by simp (rule-tac x=1 in exI, simp)
  qed
next
  fix a :: moebius
  show − a + a = 0
    unfolding plus-moebius-def uminus-moebius-def zero-moebius-def
  proof (transfer)
    fix a
   show moebius-mat-eq (moebius-comp-rep (moebius-inv-rep a) a) id-moebius-rep
      using Rep-moebius-mat[of a]
      by (simp add: mat-inv-l)
  qed
next
  fix a b :: moebius
  show a − b = a + − b
    unfolding minus-moebius-def
    by simp
qed
end

lemma [simp]: moebius-comp (moebius-inv M) M = id-moebius
by (metis left-minus plus-moebius-def uminus-moebius-def zero-moebius-def)

lemma [simp]: moebius-comp M (moebius-inv M) = id-moebius
by (metis right-minus plus-moebius-def uminus-moebius-def zero-moebius-def)

lemma moebius-pt-moebius-id [simp]: moebius-pt (id-moebius) = id
by (rule ext) (transfer, case-tac Rep-homo-coords x, auto, rule-tac x=1 in exI,
simp)

lemma [simp]: moebius-pt (moebius-inv M) (moebius-pt M z) = z
proof−
  have moebius-pt (moebius-inv M) (moebius-pt M z) = (moebius-pt (moebius-inv
M) ∘ moebius-pt M) z
    by simp
  thus ?thesis
    using moebius-comp[of moebius-inv M M]
    by simp
qed

lemma moebius-pt-invert:
  assumes w = moebius-pt M z
```

shows *z* = *moebius-pt* (*moebius-inv M*) *w*
using *assms*
by *auto*

10.3 Special kinds of Moebius transformations

Reciprocal (1/z) as a moebius transformation

definition *reciprocal-moebius* :: *moebius* **where**
reciprocal-moebius = *mk-moebius 0 1 1 0*

lemma [*simp*]: *Rep-moebius-mat* (*Abs-moebius-mat* (*0*, *1*, *1*, *0*)) = (*0*, *1*, *1*, *0*)
by (*simp add*: *Abs-moebius-mat-inverse*)

lemma [*simp*]: *Rep-moebius-mat* (*mk-moebius-rep 0 1 1 0*) = (*0*, *1*, *1*, *0*)
by (*simp add*: *mk-moebius-rep-def*)

lemma [*simp*]: *Rep-homo-coords* (*reciprocal-homo-coords z*) = (*let* (*x*, *y*) = *Rep-homo-coords z in* (*y*, *x*))
unfolding *reciprocal-homo-coords-def Let-def*
apply (*cases Rep-homo-coords z*)
using *Rep-homo-coords*[*of z*]
by (*auto simp add*: *Abs-homo-coords-inverse*)

lemma *reciprocal-moebius*:
reciprocal-homo = *moebius-pt reciprocal-moebius*
unfolding *reciprocal-moebius-def*
by (*rule ext*, *transfer*) (*auto simp add*: *split-def Let-def*, *case-tac Rep-homo-coords x*, *rule-tac x=1* **in** *exI*, *auto*)

lemma *reciprocal-moebius-inv* [*simp*]:
moebius-inv reciprocal-moebius = *reciprocal-moebius*
unfolding *reciprocal-moebius-def*
by *transfer simp*

lemma *reciprocal-homo-only-0-to-inf*:
assumes *reciprocal-homo z* = ∞_h
shows *z* = 0_h
using *assms*
unfolding *reciprocal-moebius*
using *moebius-pt-invert*[*of* ∞_h *reciprocal-moebius z*]
by (*simp add*: *reciprocal-moebius*[*symmetric*])

lemma *reciprocal-homo-only-inf-to-0*:
assumes *reciprocal-homo z* = 0_h
shows *z* = ∞_h
using *assms*
unfolding *reciprocal-moebius*
using *moebius-pt-invert*[*of* 0_h *reciprocal-moebius z*]
by (*simp add*: *reciprocal-moebius*[*symmetric*])

Euclidean similarity as a Moebius transform

definition *similarity-moebius* :: *complex* ⇒ *complex* ⇒ *moebius* **where**
similarity-moebius a b = *mk-moebius a b 0 1*

lemma *moebius-similarity-linear*:
assumes $a \neq 0$
shows *moebius-pt* (*similarity-moebius a b*) z = (*of-complex a*) $*_h$ z $+_h$ (*of-complex b*)
unfolding *similarity-moebius-def*
using *assms*
using *mult-homo-inf-right*[*of of-complex a*]
by (*subst moebius-bilinear*, *auto*)

lemma *moebius-similarity'*:
assumes $a \neq 0$
shows *moebius-pt* (*similarity-moebius a b*) = (λ *z*. (*of-complex a*) $*_h$ z $+_h$ (*of-complex b*))
using *moebius-similarity-linear*[*OF assms*, *symmetric*]
by *simp*

lemma *is-moebius-similarity'*:
assumes $a \neq 0_h$ $a \neq \infty_h$ $b \neq \infty_h$
shows (λ *z*. $a *_h z +_h b$) = *moebius-pt* (*similarity-moebius* (*to-complex a*) (*to-complex b*))
proof−
obtain *ka kb* **where** ∗: *a* = *of-complex ka* $ka \neq 0$ *b* = *of-complex kb*
using *assms*
using *inf-homo-or-complex-homo*[*of a*] *inf-homo-or-complex-homo*[*of b*]
by *auto*
thus *?thesis*
unfolding *is-moebius-def*
using *moebius-similarity'*[*of ka kb*]
by *simp*
qed

lemma *is-moebius-similarity*:
assumes $a \neq 0_h$ $a \neq \infty_h$ $b \neq \infty_h$
shows *is-moebius* (λ *z*. $a *_h z +_h b$)
using *is-moebius-similarity'*[*OF assms*]
unfolding *is-moebius-def*
by *auto*

lemma *similarity-moebius-comp*:
assumes $a \neq 0$ $c \neq 0$
shows *similarity-moebius a b* + *similarity-moebius c d* = *similarity-moebius* ($a*c$) ($a*d+b$)
using *assms*
unfolding *similarity-moebius-def plus-moebius-def*
by *transfer* (*simp add*: *mk-moebius-rep-def Abs-moebius-mat-inverse*)

```
lemma similarity-moebius-inv:
  assumes a ≠ 0
  shows − similarity-moebius a b = similarity-moebius (1/a) (−b/a)
using assms
unfolding similarity-moebius-def uminus-moebius-def
by transfer (simp add: mk-moebius-rep-def Abs-moebius-mat-inverse)

lemma similarity-moebius-id: id-moebius = similarity-moebius 1 0
unfolding similarity-moebius-def
by transfer (simp add: mk-moebius-rep-def)

lemma similarity-inf-fixed:
  assumes a ≠ 0
  shows moebius-pt (similarity-moebius a b) ∞h = ∞h
using assms
unfolding similarity-moebius-def
by transfer (simp add: mk-moebius-rep-def Abs-moebius-mat-inverse)

lemma similarity-only-inf-to-inf:
  assumes a ≠ 0  moebius-pt (similarity-moebius a b) z = ∞h
  shows z = ∞h
using assms moebius-pt-invert[of ∞h similarity-moebius a b z] similarity-inf-fixed[of
1/a −b/a]
using similarity-moebius-inv[of a b]
by simp

lemma inf-fixed-similarity:
  assumes moebius-pt M ∞h = ∞h
  shows ∃ a b. a ≠ 0 ∧ M = similarity-moebius a b
using assms
unfolding similarity-moebius-def
proof transfer
  fix M
  obtain a b c d where MM: Rep-moebius-mat M = (a, b, c, d)
    by (cases M) (auto simp add: Abs-moebius-mat-inverse)
  assume moebius-pt-rep M inf-homo-rep ≈ inf-homo-rep
  hence c = 0
    using MM
    by auto
  hence ∗: a ≠ 0 ∧ d ≠ 0
    using Rep-moebius-mat[of M] MM
    by auto
  show ∃ a b. a ≠ 0 ∧ moebius-mat-eq M (mk-moebius-rep a b 0 1)
  proof (rule-tac x=a/d in exI, rule-tac x=b/d in exI)
    show a/d ≠ 0 ∧ moebius-mat-eq M (mk-moebius-rep (a / d) (b / d) 0 1)
      using MM ‹c = 0› ‹a ≠ 0 ∧ d ≠ 0›
    by simp (rule-tac x=1/d in exI, simp add: mk-moebius-rep-def Abs-moebius-mat-inverse)
  qed
```

qed

Translation

definition *translation-moebius* **where**
translation-moebius v = similarity-moebius 1 v

lemma *translation-moebius-comp*:
(*translation-moebius v1*) + (*translation-moebius v2*) = *translation-moebius* (*v1* + *v2*)
unfolding *translation-moebius-def similarity-moebius-def plus-moebius-def*
by (*transfer*) (*auto simp add*: *mk-moebius-rep-Rep*)

lemma *translation-moebius-zero*:
translation-moebius 0 = id-moebius
unfolding *translation-moebius-def similarity-moebius-def*
by (*transfer*) (*auto simp add*: *mk-moebius-rep-Rep*)

lemma *moebius-translation-inv*:
− (*translation-moebius v1*) = *translation-moebius* (−*v1*)
using *translation-moebius-comp*[*of v1* −*v1*] *translation-moebius-zero uminus-moebius-def*
using *equals-zero-I*[*of translation-moebius v1 translation-moebius* (−*v1*)]
by *simp*

lemma *moebius-pt-translation* [*simp*]: *moebius-pt* (*translation-moebius v*) (*of-complex z*) = *of-complex* (*v* + *z*)
unfolding *translation-moebius-def similarity-moebius-def*
by *transfer* (*simp add*: *mk-moebius-rep-Rep*)

Rotation

definition *rotation-moebius* **where**
rotation-moebius φ = similarity-moebius (*cis φ*) *0*

lemma *rotation-moebius-comp*:
(*rotation-moebius φ1*) + (*rotation-moebius φ2*) = *rotation-moebius* (*φ1* + *φ2*)
unfolding *rotation-moebius-def similarity-moebius-def plus-moebius-def*
by *transfer* (*simp add*: *mk-moebius-rep-Rep cis-mult*)

lemma *rotation-moebius-zero*:
rotation-moebius 0 = id-moebius
unfolding *rotation-moebius-def similarity-moebius-def*
by *transfer* (*simp add*: *mk-moebius-rep-Rep*)

lemma *rotation-moebius-inverse*:
− (*rotation-moebius φ*) = *rotation-moebius* (− *φ*)
using *rotation-moebius-comp*[*of φ* −*φ*] *rotation-moebius-zero*
using *equals-zero-I*[*of rotation-moebius φ rotation-moebius* (−*φ*)]
by *simp*

lemma *moebius-pt-rotation* [*simp*]: *moebius-pt* (*rotation-moebius φ*) (*of-complex*

z) = *of-complex* (*cis* φ $*$ *z*)
unfolding *rotation-moebius-def similarity-moebius-def*
by *transfer* (*simp add*: *mk-moebius-rep-Rep*)

Dilatation

definition *dilatation-moebius* **where**
dilatation-moebius a = *similarity-moebius* (*cor a*) *0*

lemma *dilatation-moebius-comp*:
assumes *a1* > *0* *a2* > *0*
shows (*dilatation-moebius a1*) + (*dilatation-moebius a2*) = *dilatation-moebius* (*a1* $*$ *a2*)
using *assms*
unfolding *dilatation-moebius-def similarity-moebius-def plus-moebius-def*
by *transfer* (*simp add*: *mk-moebius-rep-def Abs-moebius-mat-inverse*)

lemma *dilatation-moebius-zero*:
dilatation-moebius 1 = *id-moebius*
unfolding *dilatation-moebius-def similarity-moebius-def*
by *transfer* (*simp add*: *mk-moebius-rep-Rep*)

lemma *dilatation-moebius-inverse*:
assumes *a* > *0*
shows − (*dilatation-moebius a*) = *dilatation-moebius* (*1*/*a*)
using *assms*
using *dilatation-moebius-comp*[*of a 1*/*a*] *dilatation-moebius-zero*
using *equals-zero-I*[*of dilatation-moebius a dilatation-moebius* (*1*/*a*)]
by *simp*

lemma *moebius-pt-dilatation* [*simp*]: *a* ≠ *0* ⟹ *moebius-pt* (*dilatation-moebius a*) (*of-complex z*) = *of-complex* (*cor a* $*$ *z*)
unfolding *dilatation-moebius-def similarity-moebius-def*
by *transfer* (*simp add*: *mk-moebius-rep-Rep*)

rotation-dilation-moebius

definition *rotation-dilatation-moebius* **where**
rotation-dilatation-moebius a = *similarity-moebius a 0*

lemma *rot-dil*:
assumes *a* ≠ *0*
shows *rotation-dilatation-moebius a* = *rotation-moebius* (*arg a*) + *dilatation-moebius* (*cmod a*)
using *assms*
unfolding *rotation-dilatation-moebius-def rotation-moebius-def dilatation-moebius-def similarity-moebius-def plus-moebius-def*
by *transfer* (*simp add*: *mk-moebius-rep-Rep*)

10.4 Decomposition

lemma *similarity-decomposition*:
assumes *a ≠ 0*
shows *similarity-moebius a b* = (*translation-moebius b*) + (*rotation-moebius* (*arg a*)) + (*dilatation-moebius* (*cmod a*))
proof−
have *similarity-moebius a b* = (*translation-moebius b*) + *rotation-dilatation-moebius a*
unfolding *rotation-dilatation-moebius-def translation-moebius-def similarity-moebius-def plus-moebius-def*
using *assms*
by *transfer* (*simp add*: *mk-moebius-rep-Rep*)
thus *?thesis*
using *rot-dil*[*OF assms*]
by (*auto simp add*: *add-assoc simp del*: *plus-moebius-def*)
qed

lemma *moebius-decomposition*:
assumes *c ≠ 0 a∗d − b∗c ≠ 0*
shows *mk-moebius a b c d* =
translation-moebius (*a/c*) +
rotation-dilatation-moebius ((*b∗c − a∗d*)/(*c∗c*)) +
reciprocal-moebius +
translation-moebius (*d/c*)
using *assms*
unfolding *rotation-dilatation-moebius-def translation-moebius-def similarity-moebius-def plus-moebius-def reciprocal-moebius-def*
by *transfer* (*simp add*: *mk-moebius-rep-Rep*, *rule-tac x=1/c* **in** *exI*, *simp add*: *field-simps*)

lemma *wlog-moebius-decomposition*:
assumes
trans: ⋀ *v*. *P* (*translation-moebius v*) **and** *rot*: ⋀ *α*. *P* (*rotation-moebius α*) **and** *dil*: ⋀ *k*. *P* (*dilatation-moebius k*) **and** *recip*: *P* (*reciprocal-moebius*) **and**
comp: ⋀ *M1 M2*. ⟦*P M1*; *P M2*⟧ ⟹ *P* (*M1* + *M2*)
shows *P M*
proof−
obtain *a b c d* **where** *M* = *mk-moebius a b c d mat-det* (*a*, *b*, *c*, *d*) ≠ *0*
using *ex-mk-moebius*[*of M*]
by *auto*
show *?thesis*
proof (*cases c* = *0*)
case *False*
show *?thesis*
using *moebius-decomposition*[*of c a d b*] ‹*mat-det* (*a*, *b*, *c*, *d*) ≠ *0*› ‹*c* ≠ *0*› ‹*M* = *mk-moebius a b c d*›
using *rot-dil*[*of* (*b∗c − a∗d*) / (*c∗c*)]
using *trans*[*of a/c*] *rot*[*of arg* ((*b∗c − a∗d*) / (*c∗c*))] *dil*[*of cmod* ((*b∗c − a∗d*) / (*c∗c*))] *recip*

```
      using comp
      by simp (metis trans)
  next
    case True
    hence M = similarity-moebius (a/d) (b/d)
      using ‹M = mk-moebius a b c d› ‹mat-det (a, b, c, d) ≠ 0›
      unfolding similarity-moebius-def
       by transfer (auto simp add: mk-moebius-rep-Rep, rule-tac x=k/d in exI,
case-tac Rep-moebius-mat M, simp)
    thus ?thesis
      using ‹c = 0› ‹mat-det (a, b, c, d) ≠ 0›
      using similarity-decomposition[of a/d b/d]
      using trans[of b/d] rot[of arg (a/d)] dil[of cmod (a/d)] comp
      by simp
  qed
qed
```

10.5 Cross ratio and moebius existence

```
lemma is-moebius-cross-ratio:
  assumes z1 ≠ z2 z2 ≠ z3 z1 ≠ z3
  shows is-moebius (λ z. cross-ratio z z1 z2 z3)
proof–
  have ∃ M. ∀ z. cross-ratio z z1 z2 z3 = moebius-pt M z
    using assms
  proof (transfer)
    fix z1 z2 z3
    obtain z1' z1'' where zz1: Rep-homo-coords z1 = (z1', z1'')
      by (rule obtain-homo-coords)
    obtain z2' z2'' where zz2: Rep-homo-coords z2 = (z2', z2'')
      by (rule obtain-homo-coords)
    obtain z3' z3'' where zz3: Rep-homo-coords z3 = (z3', z3'')
      by (rule obtain-homo-coords)

    let ?m23 = z2'*z3''−z3'*z2''
    let ?m21 = z2'*z1''−z1'*z2''
    let ?m13 = z1'*z3''−z3'*z1''
    let ?M = (z1''*?m23, −z1'*?m23, z3''*?m21, −z3'*?m21)
    assume ¬ z1 ≈ z2 ¬ z2 ≈ z3 ¬ z1 ≈ z3
    hence ∗: ?m23 ≠ 0 ?m21 ≠ 0 ?m13 ≠ 0
      using zz1 zz2 zz3
     using homo-coords-eq-mix[of z1 z1' z1'' z2 z2' z2''] homo-coords-eq-mix[of z1
z1' z1'' z3 z3' z3''] homo-coords-eq-mix[of z2 z2' z2'' z3 z3' z3'']
      by auto
    have mat-det ?M = ?m21∗?m23∗?m13
      by (simp add: field-simps)
    hence mat-det ?M ≠ 0
      using ∗
      by simp
    show ∃M. ∀z. cross-ratio-rep z z1 z2 z3 ≈ moebius-pt-rep M z
    proof (rule-tac x=Abs-moebius-mat ?M in exI, rule)
      fix z
      obtain z′ z″ where zz: Rep-homo-coords z = (z′, z″)
        by (rule obtain-homo-coords)

      let ?m01 = z′*z1″−z1′*z″
      let ?m03 = z′*z3″−z3′*z″

      have ?m01 ≠ 0 ∨ ?m03 ≠ 0
        using ∗ Rep-homo-coords[of z] zz
        apply (cases z″ = 0 ∨ z1″ = 0 ∨ z3″ = 0)
        apply (auto simp add: field-simps)
        apply (subgoal-tac z1′/z1″ = z3′/z3″)
        by (simp add: field-simps) (metis eq-divide-imp mult-divide-mult-cancel-left
times-divide-eq-right times-divide-times-eq)
      note ∗ = ∗ this

      show cross-ratio-rep z z1 z2 z3 ≈ moebius-pt-rep (Abs-moebius-mat ?M) z
      using zz1 zz2 zz3 zz ∗ Rep-homo-coords[of z] mult-mv-nonzero[of Rep-homo-coords
z ?M] ‹mat-det ?M ≠ 0›
      by (simp add: cross-ratio-rep-def moebius-pt-rep-def split-def Let-def Abs-moebius-mat-inverse
Abs-homo-coords-inverse)
           (rule-tac x=1 in exI, simp add: field-simps)
    qed
  qed
  thus ?thesis
    by (auto simp add: is-moebius-def)
qed

lemma ex-moebius-01inf:
  assumes z1 ≠ z2 z1 ≠ z3 z2 ≠ z3
  shows ∃ M. ((moebius-pt M z1 = 0_h) ∧ (moebius-pt M z2 = 1_h) ∧ (moebius-pt
M z3 = ∞_h))
using assms
using is-moebius-cross-ratio[OF ‹z1 ≠ z2› ‹z2 ≠ z3› ‹z1 ≠ z3›]
using cross-ratio-0[OF ‹z1 ≠ z2› ‹z1 ≠ z3›] cross-ratio-1[OF ‹z1 ≠ z2› ‹z2 ≠
z3›] cross-ratio-inf[OF ‹z1 ≠ z3› ‹z2 ≠ z3›]
by (auto simp add: is-moebius-def) metis

lemma ex-moebius:
  assumes z1 ≠ z2 z1 ≠ z3 z2 ≠ z3  w1 ≠ w2 w1 ≠ w3 w2 ≠ w3
  shows ∃ M. ((moebius-pt M z1 = w1) ∧ (moebius-pt M z2 = w2) ∧ (moebius-pt
M z3 = w3))
proof−
  obtain M1 where ∗: moebius-pt M1 z1 = 0_h ∧ moebius-pt M1 z2 = 1_h ∧
moebius-pt M1 z3 = ∞_h
    using ex-moebius-01inf[OF assms(1−3)]
    by auto
  obtain M2 where **: moebius-pt M2 w1 = 0_h ∧ moebius-pt M2 w2 = 1_h ∧ moebius-pt M2 w3 = ∞_h
    using ex-moebius-01inf[OF assms(4-6)]
    by auto
  let ?M = moebius-comp (moebius-inv M2) M1
  show ?thesis
    using * ** bij-moebius-pt[of M2]
   by (rule-tac x=?M in exI, (subst moebius-comp[symmetric])+, (subst moebius-inv)+) (simp add: bij-def inv-f-eq)
qed

lemma ex-moebius-1:
  shows ∃ M. moebius-pt M z1 = w1
proof-
  obtain z2 z3 where z1 ≠ z2 z1 ≠ z3 z2 ≠ z3
    using ex-3-different-points[of z1]
    by auto
  moreover
  obtain w2 w3 where w1 ≠ w2 w1 ≠ w3 w2 ≠ w3
    using ex-3-different-points[of w1]
    by auto
  ultimately
  show ?thesis
    using ex-moebius[of z1 z2 z3 w1 w2 w3]
    by auto
qed

lemma wlog-moebius-01inf:
  fixes M::moebius
  assumes P 0_h 1_h ∞_h z1 ≠ z2 z2 ≠ z3 z1 ≠ z3
  ⋀ M a b c. P a b c ⟹ P (moebius-pt M a) (moebius-pt M b) (moebius-pt M c)
  shows P z1 z2 z3
proof-
  from assms obtain M where *:
    moebius-pt M z1 = 0_h  moebius-pt M z2 = 1_h   moebius-pt M z3 = ∞_h
    using ex-moebius-01inf[of z1 z2 z3]
    by auto
  have **: moebius-pt (moebius-inv M) 0_h = z1  moebius-pt (moebius-inv M) 1_h = z2 moebius-pt (moebius-inv M) ∞_h = z3
    by (subst *[symmetric], simp)+
  thus ?thesis
    using assms
    by auto
qed
```

10.6 Fixed points and moebius uniqueness

```
lemma three-fixed-points-01inf:
  assumes moebius-pt M 0_h = 0_h  moebius-pt M 1_h = 1_h moebius-pt M ∞_h = ∞_h
  shows M = id-moebius
using assms
by transfer (case-tac Rep-moebius-mat M, auto)

lemma three-fixed-points:
  assumes z1 ≠ z2 z1 ≠ z3 z2 ≠ z3
  assumes moebius-pt M z1 = z1 moebius-pt M z2 = z2 moebius-pt M z3 = z3
  shows M = id-moebius
proof−
  from assms obtain M' where ∗: moebius-pt M' z1 = 0_h  moebius-pt M' z2 = 1_h   moebius-pt M' z3 = ∞_h
    using ex-moebius-01inf[of z1 z2 z3]
    by auto
  have ∗∗: moebius-pt (moebius-inv M') 0_h = z1  moebius-pt (moebius-inv M') 1_h = z2 moebius-pt (moebius-inv M') ∞_h = z3
    by (subst ∗[symmetric], simp)+

  have M' + M + (−M') = 0
    unfolding zero-moebius-def
    apply (rule three-fixed-points-01inf)
    using ∗ ∗∗ assms
    by (simp add: moebius-comp[symmetric])+
  thus ?thesis
    by (metis eq-neg-iff-add-eq-0 minus-add-cancel zero-moebius-def)
qed

lemma unique-moebius-three-points:
  assumes z1 ≠ z2 z1 ≠ z3 z2 ≠ z3
  assumes moebius-pt M1 z1 = w1  moebius-pt M1 z2 = w2 moebius-pt M1 z3 = w3
          moebius-pt M2 z1 = w1  moebius-pt M2 z2 = w2 moebius-pt M2 z3 = w3
  shows M1 = M2
proof−
  let ?M = moebius-comp (moebius-inv M2) M1
  have moebius-pt ?M z1 = z1
    using ‹moebius-pt M1 z1 = w1› ‹moebius-pt M2 z1 = w1›
    using bij-moebius-pt[of M2]
   by (subst moebius-comp[symmetric], subst moebius-inv, simp add: bij-def inv-f-eq)
  moreover
  have moebius-pt ?M z2 = z2
    using ‹moebius-pt M1 z2 = w2› ‹moebius-pt M2 z2 = w2›
    using bij-moebius-pt[of M2]
   by (subst moebius-comp[symmetric], subst moebius-inv, simp add: bij-def inv-f-eq)
  moreover
  have moebius-pt ?M z3 = z3
    using ‹moebius-pt M1 z3 = w3› ‹moebius-pt M2 z3 = w3›
    using bij-moebius-pt[of M2]
    by (subst moebius-comp[symmetric], subst moebius-inv, simp add: bij-def inv-f-eq)
  ultimately
  have ?M = id-moebius
    using assms three-fixed-points
    by auto
  thus ?thesis
      by (metis add-minus-cancel left-minus plus-moebius-def uminus-moebius-def
zero-moebius-def)
qed

lemma ex-unique-moebius-three-points:
  assumes z1 ≠ z2 z1 ≠ z3 z2 ≠ z3  w1 ≠ w2 w1 ≠ w3 w2 ≠ w3
  shows ∃! M. ((moebius-pt M z1 = w1) ∧ (moebius-pt M z2 = w2) ∧ (moebius-pt
M z3 = w3))
proof−
 obtain M where ∗: moebius-pt M z1 = w1 ∧ moebius-pt M z2 = w2 ∧ moebius-pt
M z3 = w3
    using ex-moebius[OF assms]
    by auto
  show ?thesis
    unfolding Ex1-def
  proof (rule-tac x=M in exI, rule)
    show ∀ y. moebius-pt y z1 = w1 ∧ moebius-pt y z2 = w2 ∧ moebius-pt y z3 =
w3 ⟶ y = M
      using ∗
      using unique-moebius-three-points[OF assms(1−3)]
      by simp
  qed (simp add: ∗)
qed

lemma ex-unique-moebius-three-points-fun:
  assumes z1 ≠ z2 z1 ≠ z3 z2 ≠ z3 w1 ≠ w2 w1 ≠ w3 w2 ≠ w3
  shows ∃! f. is-moebius f ∧ (f z1 = w1) ∧ (f z2 = w2) ∧ (f z3 = w3)
proof−
  obtain M where moebius-pt M z1 = w1 moebius-pt M z2 = w2 moebius-pt M
z3 = w3
    using ex-unique-moebius-three-points[OF assms]
    by auto
  thus ?thesis
    using ex-unique-moebius-three-points[OF assms]
    unfolding Ex1-def
    by (rule-tac x=moebius-pt M in exI) (auto simp add: is-moebius-def)
qed

lemma is-cross-ratio-01inf:
  assumes z1 ≠ z2 z1 ≠ z3 z2 ≠ z3 is-moebius f
  assumes f z1 = 0_h f z2 = 1_h f z3 = ∞_h
  shows f = (λ z. cross-ratio z z1 z2 z3)
  using assms
```

using *cross-ratio-0*[*OF* ‹*z1* ≠ *z2*› ‹*z1* ≠ *z3*›] *cross-ratio-1*[*OF* ‹*z1* ≠ *z2*› ‹*z2* ≠ *z3*›] *cross-ratio-inf*[*OF* ‹*z1* ≠ *z3*› ‹*z2* ≠ *z3*›]
using *is-moebius-cross-ratio*[*OF* ‹*z1* ≠ *z2*› ‹*z2* ≠ *z3*› ‹*z1* ≠ *z3*›]
using *ex-unique-moebius-three-points-fun*[*OF* ‹*z1* ≠ *z2*› ‹*z1* ≠ *z3*› ‹*z2* ≠ *z3*›, *of* 0_h 1_h ∞_h]
by *auto*

lemma *moebius-preserve-cross-ratio*:
assumes *z1* ≠ *z2* *z1* ≠ *z3* *z2* ≠ *z3*
shows *cross-ratio z z1 z2 z3* = *cross-ratio* (*moebius-pt M z*) (*moebius-pt M z1*) (*moebius-pt M z2*) (*moebius-pt M z3*)
proof−
let *?f* = λ *z*. *cross-ratio z z1 z2 z3*
let *?M* = *moebius-pt M*
let *?iM* = *inv ?M*
have (*?f* ∘ *?iM*) (*?M z1*) = 0_h
using *bij-moebius-pt*[*of M*] *cross-ratio-0*[*OF* ‹*z1* ≠ *z2*› ‹*z1* ≠ *z3*›]
by (*simp add*: *bij-def*)
moreover
have (*?f* ∘ *?iM*) (*?M z2*) = 1_h
using *bij-moebius-pt*[*of M*] *cross-ratio-1*[*OF* ‹*z1* ≠ *z2*› ‹*z2* ≠ *z3*›]
by (*simp add*: *bij-def*)
moreover
have (*?f* ∘ *?iM*) (*?M z3*) = ∞_h
using *bij-moebius-pt*[*of M*] *cross-ratio-inf*[*OF* ‹*z1* ≠ *z3*› ‹*z2* ≠ *z3*›]
by (*simp add*: *bij-def*)
moreover
have *is-moebius* (*?f* ∘ *?iM*)
by (*rule is-moebius-comp*, *rule is-moebius-cross-ratio*[*OF* ‹*z1* ≠ *z2*› ‹*z2* ≠ *z3*› ‹*z1* ≠ *z3*›], *rule is-moebius-inv*, *auto simp add*: *is-moebius-def*)
moreover
have *?M z1* ≠ *?M z2* *?M z1* ≠ *?M z3* *?M z2* ≠ *?M z3*
using *assms*
using *bij-moebius-pt*[*of M*]
unfolding *bij-def inj-on-def*
by *blast+*
ultimately
have *?f* ∘ *?iM* = (λ *z*. *cross-ratio z* (*?M z1*) (*?M z2*) (*?M z3*))
using *assms*
using *is-cross-ratio-01inf*[*of ?M z1 ?M z2 ?M z3 ?f* ∘ *?iM*]
by *simp*
moreover
have (*?f* ∘ *?iM*) (*?M z*) = *cross-ratio z z1 z2 z3*
using *bij-moebius-pt*[*of M*]
by (*simp add*: *bij-def*)
moreover
have (λ *z*. *cross-ratio z* (*?M z1*) (*?M z2*) (*?M z3*)) (*?M z*) = *cross-ratio* (*?M z*) (*?M z1*) (*?M z2*) (*?M z3*)

```
    by simp
  ultimately
  show ?thesis
    by simp
qed

lemma fixed-points-0inf':
  assumes moebius-pt M 0_h = 0_h moebius-pt M ∞_h = ∞_h
  shows ∃ k::complex-homo. (k ≠ 0_h ∧ k ≠ ∞_h) ∧ (∀ z. moebius-pt M z = k *_h z)
using assms
proof (transfer)
  fix M
  obtain a b c d where MM: Rep-moebius-mat M = (a, b, c, d)
    by (cases M) (auto simp add: Abs-moebius-mat-inverse)
   assume moebius-pt-rep M zero-homo-rep ≈ zero-homo-rep moebius-pt-rep M inf-homo-rep ≈ inf-homo-rep
  hence b = 0 c = 0
    using MM
    by auto
  hence *: a ≠ 0 ∧ d ≠ 0
    using Rep-moebius-mat[of M] MM
    by auto
  show ∃k. (¬ k ≈ zero-homo-rep ∧ ¬ k ≈ inf-homo-rep) ∧ (∀z. moebius-pt-rep M z ≈ k *_hc z)
  proof (rule-tac x=Abs-homo-coords (a, d) in exI, rule conjI)
    show ¬ Abs-homo-coords (a, d) ≈ zero-homo-rep ∧ ¬ Abs-homo-coords (a, d) ≈ inf-homo-rep
      using *
      by (auto simp add: Abs-homo-coords-inverse)
  next
    show ∀z. moebius-pt-rep M z ≈ Abs-homo-coords (a, d) *_hc z
    proof
      fix z
      obtain z1 z2 where zz: Rep-homo-coords z = (z1, z2)
        by (rule obtain-homo-coords)
      thus moebius-pt-rep M z ≈ Abs-homo-coords (a, d) *_hc z
       using MM * ‹b = 0› ‹c = 0› mult-homo-coords-Rep[of Abs-homo-coords (a, d) a d z z1 z2] Rep-homo-coords[of z]
        by (simp add: Abs-homo-coords-inverse) (rule-tac x=1 in exI, simp)
    qed
  qed
qed

lemma fixed-points-0inf:
  assumes moebius-pt M 0_h = 0_h moebius-pt M ∞_h = ∞_h
  shows ∃ k::complex-homo. (k ≠ 0_h ∧ k ≠ ∞_h) ∧ moebius-pt M = (λ z. k *_h z)
using fixed-points-0inf'[OF assms]
by auto
```

10.7 Pole

```
definition is-pole where
  is-pole M z ⟷ moebius-pt M z = ∞h

lemma ex1-pole:
  ∃! z. is-pole M z
using bij-moebius-pt[of M]
unfolding is-pole-def bij-def inj-on-def surj-def
unfolding Ex1-def
by (metis UNIV-I)

definition pole where pole M = (THE z. is-pole M z)

lemma pole-mk-moebius:
  assumes is-pole (mk-moebius a b c d) z c ≠ 0 a∗d − b∗c ≠ 0
  shows z = of-complex (−d/c)
proof−
  let ?t1 = translation-moebius (a / c)
  let ?rd = rotation-dilatation-moebius ((b ∗ c − a ∗ d) / (c ∗ c))
  let ?r = reciprocal-moebius
  let ?t2 = translation-moebius (d / c)
  have moebius-pt (?rd + ?r + ?t2) z = ∞h
    using assms
    unfolding is-pole-def
    apply (subst (asm) moebius-decomposition)
    apply (auto simp add: moebius-comp[symmetric] translation-moebius-def)
    apply (subst similarity-only-inf-to-inf[of 1 a/c], auto)
    done
  hence moebius-pt (?r + ?t2) z = ∞h
    using ‹a∗d − b∗c ≠ 0› ‹c ≠ 0›
    unfolding rotation-dilatation-moebius-def
    apply (simp add: moebius-comp[symmetric])
    apply (subst similarity-only-inf-to-inf[of (b∗c−a∗d)/(c∗c) 0], auto)
    done
  hence moebius-pt ?t2 z = 0h
    apply (simp add: moebius-comp[symmetric])
    apply (subst (asm) reciprocal-moebius[symmetric])
    apply (subst reciprocal-homo-only-0-to-inf, auto)
    done
  thus ?thesis
    using moebius-pt-invert[of 0h ?t2 z] moebius-translation-inv[of d/c]
    by simp (subst zero-of-complex[symmetric], simp del: zero-of-complex)
qed

lemma pole-similarity:
  assumes is-pole (similarity-moebius a b) z a ≠ 0
  shows z = ∞h
using assms
unfolding is-pole-def
```

using *similarity-only-inf-to-inf*[*of a b z*]
by *simp*

10.8 Antihomographies

definition *is-antihomography* **where**
is-antihomography f $\longleftrightarrow$ ($\exists$ *f'*. *is-moebius f'* $\wedge$ *f* = *f'* $\circ$ *cnj-homo*)

lemma *is-antihomography inversion-homo*
using *reciprocal-moebius*
unfolding *inversion-homo-sym is-antihomography-def*
by (*auto simp add*: *is-moebius-def*)

10.9 Classification

lemma *similarity-scale-1*:
assumes *k* $\neq$ *0*
shows *similarity* (*k* $*_{sm}$ *I*) *M* = *similarity I M*
using *assms*
unfolding *similarity-def*
using *mat-inv-mult-sm*[*of k I*]
by *simp*

lemma *similarity-scale-2*:
shows *similarity I* (*k* $*_{sm}$ *M*) = *k* $*_{sm}$ (*similarity I M*)
unfolding *similarity-def*
by *auto*

lemma [*simp*]: *mat-trace* (*k* $*_{sm}$ *M*) = *k* $*$ *mat-trace M*
by (*cases M*) (*simp add*: *field-simps*)

definition *moebius-mb-rep* **where**
moebius-mb-rep I M = *Abs-moebius-mat* (*similarity* (*Rep-moebius-mat I*) (*Rep-moebius-mat M*))

lemma *moebius-mb-rep-Rep* [*simp*]:
Rep-moebius-mat (*moebius-mb-rep I M*) = *similarity* (*Rep-moebius-mat I*) (*Rep-moebius-mat M*)
using *Rep-moebius-mat*[*of I*] *Rep-moebius-mat*[*of M*]
unfolding *moebius-mb-rep-def*
by (*simp add*: *mat-det-similarity Abs-moebius-mat-inverse*)

lift-definition *moebius-mb* :: *moebius* $\Rightarrow$ *moebius* $\Rightarrow$ *moebius* **is** *moebius-mb-rep*
proof−
fix *M M' I I'*
assume *moebius-mat-eq M M' moebius-mat-eq I I'*
thus *moebius-mat-eq* (*moebius-mb-rep I M*) (*moebius-mb-rep I' M'*)
by (*auto simp add*: *similarity-scale-1 similarity-scale-2*)
qed

```
definition similarity-invar-rep where
  similarity-invar-rep M =
    (let M = Rep-moebius-mat M
      in (mat-trace M)^2 / mat-det M − 4)

lift-definition similarity-invar :: moebius ⇒ complex is similarity-invar-rep
by (auto simp add: similarity-invar-rep-def Let-def power2-eq-square)

lemma
  similarity-invar (moebius-mb I M) = similarity-invar M
proof transfer
  fix I M
  show similarity-invar-rep (moebius-mb-rep I M) = similarity-invar-rep M
    using Rep-moebius-mat[of I] Rep-moebius-mat[of M]
   by (simp add: similarity-invar-rep-def Let-def mat-trace-similarity mat-det-similarity)
qed

definition similar where
 similar M1 M2 ⟷ (∃ I. moebius-mb I M1 = M2)

lemma [simp]: similarity eye M = M
unfolding similarity-def
by simp (metis eye-def mat-eye-l mat-eye-r)

lemma [simp]: similarity (1, 0, 0, 1) M = M
unfolding eye-def[symmetric]
by (simp del: eye-def)

lemma similarity-comp:
  assumes mat-det I1 ≠ 0 mat-det I2 ≠ 0
  shows similarity I1 (similarity I2 M) = similarity (I2*mmI1) M
using assms
unfolding similarity-def
by (simp add: mult-mm-assoc mat-inv-mult-mm)

lemma similarity-inv:
  assumes similarity I M1 = M2 mat-det I ≠ 0
  shows similarity (mat-inv I) M2 = M1
using assms
unfolding similarity-def
by simp (metis mat-eye-l mult-mm-assoc mult-mm-inv-r)

lemma similar-refl [simp]: similar M M
unfolding similar-def
by (rule-tac x=id-moebius in exI) (transfer, simp, rule-tac x=1 in exI, auto)

lemma similar-sym:
  assumes similar M1 M2
  shows similar M2 M1
```

proof−
from *assms* **obtain** *I* **where** *M2* = *moebius-mb I M1*
unfolding *similar-def*
by *auto*
hence *M1* = *moebius-mb* (*moebius-inv I*) *M2*
proof *transfer*
fix *M2 I M1*
assume *moebius-mat-eq M2* (*moebius-mb-rep I M1*)
then obtain *k* **where** *k* ≠ *0 similarity* (*Rep-moebius-mat I*) (*Rep-moebius-mat M1*) = *k* $*_{sm}$ *Rep-moebius-mat M2*
by *auto*
thus *moebius-mat-eq M1* (*moebius-mb-rep* (*moebius-inv-rep I*) *M2*)
using *Rep-moebius-mat*[*of I*] *similarity-inv*[*of Rep-moebius-mat I Rep-moebius-mat M1 k* $*_{sm}$ *Rep-moebius-mat M2*]
by (*auto simp add*: *similarity-scale-2*) (*rule-tac x*=*1*/*k* **in** *exI*, *simp*, *metis mult-sm-inv-l*)
qed
thus *?thesis*
unfolding *similar-def*
by *auto*
qed

lemma *similar-trans*:
assumes *similar M1 M2 similar M2 M3*
shows *similar M1 M3*
proof−
obtain *I1 I2* **where** *moebius-mb I1 M1* = *M2 moebius-mb I2 M2* = *M3*
using *assms*
by (*auto simp add*: *similar-def*)
thus *?thesis*
unfolding *similar-def*
proof (*rule-tac x*=*moebius-comp I1 I2* **in** *exI*, *transfer*)
fix *I1 I2 M1 M2 M3*
assume *moebius-mat-eq* (*moebius-mb-rep I1 M1*) *M2*
moebius-mat-eq (*moebius-mb-rep I2 M2*) *M3*
thus *moebius-mat-eq* (*moebius-mb-rep* (*moebius-comp-rep I1 I2*) *M1*) *M3*
using *Rep-moebius-mat*[*of I1*] *Rep-moebius-mat*[*of I2*]
by (*auto simp add*: *similarity-scale-2 similarity-comp*) (*rule-tac x*=*ka*∗*k* **in** *exI*, *simp*)
qed
qed

end

11 Circline

theory *Circline*

```
imports Moebius HermiteanMatrices ElementaryComplexGeometry RiemannSphere
Angles
begin
```

11.1 Circline definition

```
typedef circline-mat = {H. hermitean H ∧ H ≠ mat-zero}
by (rule-tac x=eye in exI) (auto simp add: hermitean-def mat-adj-def mat-cnj-def)

lemma circline-mat-mult-sm-Rep [simp]:
  assumes k ≠ 0
  shows Rep-circline-mat (Abs-circline-mat ((cor k) *sm (Rep-circline-mat H)))
= (cor k) *sm (Rep-circline-mat H)
using assms Rep-circline-mat[of H]
using hermitean-mult-real[of Rep-circline-mat H k] nonzero-mult-real[of Rep-circline-mat
H cor k]
by (simp add: Abs-circline-mat-inverse)

definition circline-mat-eq where
  [simp]: circline-mat-eq A B ⟷ (∃ k::real. k ≠ 0 ∧ Rep-circline-mat B =
complex-of-real k *sm (Rep-circline-mat A))

lemma [simp]: circline-mat-eq H H
  by (simp, rule-tac x=1 in exI, simp)

quotient-type circline = circline-mat / circline-mat-eq
proof (rule equivpI)
  show reflp circline-mat-eq
    unfolding reflp-def
    by (auto, rule-tac x=1 in exI, simp)
next
  show symp circline-mat-eq
    unfolding symp-def
    by (auto, rule-tac x=1/k in exI, simp)
next
  show transp circline-mat-eq
    unfolding transp-def
    by (auto, rule-tac x=ka*k in exI, simp)
qed
```

Circline with specified matrix

```
definition mk-circline-rep where
 mk-circline-rep A B C D = Abs-circline-mat (A, B, C, D)

lift-definition mk-circline :: complex ⇒ complex ⇒ complex ⇒ complex ⇒ cir-
cline is mk-circline-rep
by (simp del: circline-mat-eq-def)

lemma ex-mk-circline:
```

shows ∃ *A B C D*. *H* = *mk-circline A B C D* ∧ *hermitean* (*A*, *B*, *C*, *D*) ∧ (*A*, *B*, *C*, *D*) ≠ *mat-zero*
proof *transfer*
fix *H*
obtain *A B C D* **where** *Rep-circline-mat H* = (*A*, *B*, *C*, *D*)
by (*cases Rep-circline-mat H*, *auto*)
hence *circline-mat-eq H* (*mk-circline-rep A B C D*) ∧ *hermitean* (*A*, *B*, *C*, *D*) ∧ (*A*, *B*, *C*, *D*) ≠ *mat-zero*
using *Rep-circline-mat*[*of H*]
by (*auto simp add*: *mk-circline-rep-def Abs-circline-mat-inverse*) (*rule-tac x=1* **in** *exI*, *simp*)+
thus ∃ *A B C D*. *circline-mat-eq H* (*mk-circline-rep A B C D*) ∧ *hermitean* (*A*, *B*, *C*, *D*) ∧ (*A*, *B*, *C*, *D*) ≠ *mat-zero*
by *blast*
qed

circline type

definition *circline-type-rep* **where**
circline-type-rep H = *sgn* (*Re* (*mat-det* (*Rep-circline-mat H*)))

lift-definition *circline-type* :: *circline* ⇒ *real* **is** *circline-type-rep*
proof−
fix *H H'*
assume *circline-mat-eq H H'*
thus *circline-type-rep H* = *circline-type-rep H'*
by (*auto simp add*: *circline-type-rep-def sgn-mult*) (*metis not-real-square-gt-zero real-sgn-pos sgn-mult*)
qed

lemma *circline-type*: *circline-type H* = −*1* ∨ *circline-type H* = *0* ∨ *circline-type H* = *1*
proof *transfer*
fix *H*
show *circline-type-rep H* = −*1* ∨ *circline-type-rep H* = *0* ∨ *circline-type-rep H* = *1*
unfolding *circline-type-rep-def*
using *Rep-circline-mat*[*of H*]
by (*metis linorder-cases real-sgn-neg real-sgn-pos sgn-zero-iff*)
qed

on-circline, *circline-set*

definition *on-circline-rep* **where**
on-circline-rep H z ⟷
(*let z* = *Rep-homo-coords z*;
H = *Rep-circline-mat H*
in quad-form z H = *0*)

lift-definition *on-circline* :: *circline* ⇒ *complex-homo* ⇒ *bool* **is** *on-circline-rep*

by (*auto simp add*: *on-circline-rep-def quad-form-scale-m quad-form-scale-v Let-def simp del*: *vec-cnj-sv quad-form-def*)

definition *circline-set* :: *circline* ⇒ *complex-homo set* **where**
circline-set H = {*z*. *on-circline H z*}

Circlines trough 0 and inf

definition *circline-A0-rep* **where**
circline-A0-rep H ⟷
(*let* (*A*, *B*, *C*, *D*) = *Rep-circline-mat H in A* = *0*)

lift-definition *circline-A0* :: *circline* ⇒ *bool* **is** *circline-A0-rep*
by (*auto simp add*: *circline-A0-rep-def*)

definition *circline-D0-rep* **where**
circline-D0-rep H ⟷
(*let* (*A*, *B*, *C*, *D*) = *Rep-circline-mat H in D* = *0*)

abbreviation *is-line* **where**
is-line H ≡ *circline-A0 H*

abbreviation *is-circle* **where**
is-circle H ≡ ¬ *circline-A0 H*

lift-definition *circline-D0* :: *circline* ⇒ *bool* **is** *circline-D0-rep*
by (*auto simp add*: *circline-D0-rep-def*)

lemma *inf-on-circline-rep*: *on-circline-rep H inf-homo-rep* ⟷ *circline-A0-rep H*
by (*simp add*: *on-circline-rep-def Let-def circline-A0-rep-def split-def*) (*cases Rep-circline-mat H*, *simp add*: *vec-cnj-def*)

lemma
inf-in-circline-set: ∞_h ∈ *circline-set H* ⟷ *is-line H*
unfolding *circline-set-def*
apply *simp*
apply (*transfer*)
using *inf-on-circline-rep*
by *simp*

lemma *zero-on-circline-rep*: *on-circline-rep H zero-homo-rep* ⟷ *circline-D0-rep H*
using *Rep-circline-mat*[*of H*]
by (*simp add*: *circline-D0-rep-def on-circline-rep-def split-def Let-def Abs-homo-coords-inverse Abs-circline-mat-inverse vec-cnj-def*) (*cases Rep-circline-mat H*, *simp*)

lemma *zero-in-circline-set*: 0_h ∈ *circline-set H* ⟷ *circline-D0 H*
unfolding *circline-set-def*
apply *simp*
apply (*transfer*)

```
using zero-on-circline-rep
by simp
```

Connection with circlines in classic complex plane

```
lemma classic-circline:
  assumes H = mk-circline A B C D hermitean (A, B, C, D) ∧ (A, B, C, D) ≠
mat-zero
  shows circline-set H − {∞h} = of-complex ‘ circline (Re A) B (Re D)
using assms
unfolding circline-set-def
proof (safe)
  fix z
  assume hermitean (A, B, C, D) (A, B, C, D) ≠ mat-zero z ∈ circline (Re A)
B (Re D)
    thus on-circline (mk-circline A B C D) (of-complex z)
      using hermitean-elems[of A B C D]
     by (transfer) (simp del: mat-zero-def add: on-circline-rep-def Let-def mk-circline-rep-def
Abs-circline-mat-inverse circline-def vec-cnj-def field-simps complex-of-real-Re)
next
  fix z
  assume of-complex z = ∞h
  thus False
    by simp
next
  fix z
 assume hermitean (A, B, C, D) (A, B, C, D) ≠ mat-zero on-circline (mk-circline
A B C D) z z ∉ of-complex ‘ circline (Re A) B (Re D)
  moreover
  have z ≠ ∞h ⟶ z ∈ of-complex ‘ circline (Re A) B (Re D)
  proof
    assume z ≠ ∞h
    show z ∈ of-complex ‘ circline (Re A) B (Re D)
    proof
      show z = of-complex (to-complex z)
        using ‹z ≠ ∞h›
        by simp
    next
      show to-complex z ∈ circline (Re A) B (Re D)
         using ‹on-circline (mk-circline A B C D) z› ‹z ≠ ∞h› ‹hermitean (A, B,
C, D)› ‹(A, B, C, D) ≠ mat-zero›
      proof (transfer)
        fix A B C D z
        obtain z1 z2 where zz: Rep-homo-coords z = (z1, z2)
          by (rule obtain-homo-coords)
        assume ∗: ¬ z ≈ inf-homo-rep on-circline-rep (mk-circline-rep A B C D) z
hermitean (A, B, C, D) (A, B, C, D) ≠ mat-zero
        have z2 ≠ 0
          using ‹¬ z ≈ inf-homo-rep› Rep-homo-coords[of z] zz
          by auto (erule-tac x=1/z1 in allE, simp)
        thus to-complex-homo-coords z ∈ circline (Re A) B (Re D)
          using ∗ zz
          using hermitean-elems[of A B C D]
        by (simp add: mk-circline-rep-def on-circline-rep-def to-complex-homo-coords-def
Let-def Abs-circline-mat-inverse  vec-cnj-def complex-cnj circline-def complex-of-real-Re
field-simps del: mat-zero-def)
      qed
    qed
  qed
  ultimately
  show z = ∞h
    by simp
qed

definition mk-circle-rep where
  mk-circle-rep a r = Abs-circline-mat (1, −a, −cnj a, a∗cnj a − cor r∗cor r)
lift-definition mk-circle :: complex ⇒ real ⇒ circline is mk-circle-rep
by (simp del: circline-mat-eq-def)

lemma mk-circle-rep-Rep
  [simp]: Rep-circline-mat (mk-circle-rep a r) = (1, −a, −cnj a, a∗cnj a − cor
r∗cor r)
by (simp add: mk-circle-rep-def Abs-circline-mat-inverse hermitean-def mat-adj-def
mat-cnj-def complex-cnj)

lemma is-circle-mk-circle: is-circle (mk-circle a r)
by transfer (simp add: circline-A0-rep-def)

lemma
  assumes r ≥ 0
  shows circline-set (mk-circle a r) = of-complex ‘ {z. cmod (z − a) = r}
proof−
  let ?A = 1 and ?B = −a and ?C = −cnj a and ?D = a∗cnj a − cor r∗cor r
  have ∗: (?A, ?B, ?C, ?D) ∈ {H. hermitean H ∧ H ≠ mat-zero}
    by (simp add: hermitean-def mat-adj-def mat-cnj-def complex-cnj)
  have mk-circle a r = mk-circline ?A ?B ?C ?D
    using ∗
     by transfer (simp add: mk-circline-rep-def Abs-circline-mat-inverse, rule-tac
x=1 in exI, simp)
   hence circline-set (mk-circle a r) − {∞h} = of-complex ‘ circline ?A ?B (Re
?D)
    using classic-circline[of mk-circle a r ?A ?B ?C ?D] ∗
    by simp
  moreover
  have circline ?A ?B (Re ?D) = circle a r
   by (rule circline-circle[of ?A Re ?D ?B circline ?A ?B (Re ?D) a r∗r r], simp-all
add: cmod-square ‹r ≥ 0›)
  moreover
  have ∞h ∉ circline-set (mk-circle a r)
```

```
    using inf-in-circline-set[of mk-circle a r] is-circle-mk-circle[of a r]
    by auto
  ultimately
  show ?thesis
    unfolding circle-def
    by simp
qed

definition mk-line-rep where mk-line-rep z1 z2 =
  (let B = ii∗(z2−z1) in Abs-circline-mat (0, B, cnj B, −cnj-mix B z1))
lift-definition mk-line :: complex ⇒ complex ⇒ circline is mk-line-rep
by (simp del: circline-mat-eq-def)

lemma mk-line-rep-Rep [simp]:
  assumes z1 ≠ z2
  shows Rep-circline-mat (mk-line-rep z1 z2) =
     (let B = ii∗(z2−z1) in (0, B, cnj B, −cnj-mix B z1))
using assms
by (simp add: mk-line-rep-def Let-def Abs-circline-mat-inverse hermitean-def mat-adj-def
mat-cnj-def complex-cnj)

lemma circline-line′:
  assumes z1 ≠ z2
  shows circline 0 (i ∗ (z2 − z1)) (Re (− cnj-mix (i ∗ (z2 − z1)) z1)) = line z1
z2
proof−
  let ?B = ii ∗ (z2 − z1)
  let ?D = Re (− cnj-mix ?B z1)
  have circline 0 ?B ?D = {z. cnj ?B∗z + ?B∗cnj z + complex-of-real ?D = 0}
    using assms
    by (simp add: circline-def)
  moreover
  have is-real (− cnj-mix (i ∗ (z2 − z1)) z1)
    using cnj-mix-real[of ?B z1]
    by auto
  hence {z. cnj ?B∗z + ?B∗cnj z + complex-of-real ?D = 0} =
        {z. cnj ?B∗z + ?B∗cnj z − (cnj ?B∗z1 + ?B∗cnj z1) = 0}
    by (subst complex-of-real-Re, simp, simp add: complex-diff-def)
  moreover
  have line z1 z2 = {z. cnj-mix (i ∗ (z2 − z1)) z − cnj-mix (i ∗ (z2 − z1)) z1 =
0}
    using line-equation[of z1 z2 ?B] assms
    unfolding rot90-ii
    by simp
  ultimately
  show ?thesis
    by simp
qed
```

lemma
  assumes z1 ≠ z2
  shows circline-set (mk-line z1 z2) − {∞$_h$} = of-complex ‘ line z1 z2
proof−
  let ?A = 0 **and** ?B = ii*(z2 − z1)
  let ?C = cnj ?B **and** ?D = −cnj-mix ?B z1
  have *: (?A, ?B, ?C, ?D) ∈ {H. hermitean H ∧ H ≠ mat-zero}
    using assms
    by (simp add: hermitean-def mat-adj-def mat-cnj-def complex-cnj)
  have mk-line z1 z2 = mk-circline ?A ?B ?C ?D
    using * assms
    by transfer (simp add: mk-circline-rep-def Abs-circline-mat-inverse Let-def, rule-tac x=1 **in** exI, simp)
  hence circline-set (mk-line z1 z2) − {∞$_h$} = of-complex ‘ circline ?A ?B (Re ?D)
    using classic-circline[of mk-line z1 z2 ?A ?B ?C ?D] *
    by simp
  moreover
  have circline ?A ?B (Re ?D) = line z1 z2
    using ‹z1 ≠ z2›
    using circline-line′
    by simp
  ultimately
  show ?thesis
    by simp
qed

definition euclidean-circle-rep **where**
  euclidean-circle-rep H = (let (A, B, C, D) = Rep-circline-mat H in (−B/A, sqrt(Re ((B*C − A*D)/(A*A)))))

lift-definition euclidean-circle :: circline ⇒ complex × real **is** euclidean-circle-rep
proof−
  fix H1 H2
  obtain A1 B1 C1 D1 **where** HH1: Rep-circline-mat H1 = (A1, B1, C1, D1)
    by (cases Rep-circline-mat H1) auto
  obtain A2 B2 C2 D2 **where** HH2: Rep-circline-mat H2 = (A2, B2, C2, D2)
    by (cases Rep-circline-mat H2) auto
  assume circline-mat-eq H1 H2
  then obtain k **where** k ≠ 0 **and** *: A2 = cor k * A1 B2 = cor k * B1 C2 = cor k * C1 D2 = cor k * D1
    using HH1 HH2
    by auto
  have (cor k * B1 * (cor k * C1) − cor k * A1 * (cor k * D1)) = $(\mathit{cor}\ k)^2$ * (B1*C1 − A1*D1)
    (cor k * A1 * (cor k * A1)) = $(\mathit{cor}\ k)^2$ * (A1*A1)
    by (auto simp add: field-simps power2-eq-square)
  hence (cor k * B1 * (cor k * C1) − cor k * A1 * (cor k * D1)) /
        (cor k * A1 * (cor k * A1)) = (B1*C1 − A1*D1) / (A1*A1)

```
    using ‹k ≠ 0›
    by (simp add: power2-eq-square)
  thus euclidean-circle-rep H1 = euclidean-circle-rep H2
    using HH1 HH2 ∗ Rep-circline-mat[of H2]
    by (auto simp add: euclidean-circle-rep-def)
qed

lemma classic-circle:
  assumes is-circle H (a, r) = euclidean-circle H circline-type H ≤ 0
  shows circline-set H = of-complex ‘ circle a r
proof−
  obtain A B C D where ∗: H = mk-circline A B C D hermitean (A, B, C, D)
(A, B, C, D) ≠ mat-zero
    using ex-mk-circline[of H]
    by auto
  have is-real A is-real D C = cnj B
    using ∗ hermitean-elems
    by auto

  have Re (A∗D − B∗C) ≤ 0
    using ‹circline-type H ≤ 0› ∗
   by simp (transfer, simp add: circline-type-rep-def mk-circline-rep-def Abs-circline-mat-inverse,
smt real-sgn-pos)

  hence ∗∗: Re A ∗ Re D ≤ (cmod B)^2
    using ‹is-real A› ‹is-real D› ‹C = cnj B›
    by (simp add: cmod-square)

  have A ≠ 0
    using ‹is-circle H› ∗ ‹is-real A›
   by simp (transfer, simp add: circline-A0-rep-def mk-circline-rep-def Abs-circline-mat-inverse)
  hence Re A ≠ 0
    using ‹is-real A›
    by (cases A, simp)

  have ∗∗∗: ∞h ∉ circline-set H
    using ∗ inf-in-circline-set[of H] ‹is-circle H›
    by simp

  let ?a = −B/A
  let ?r2 = ((cmod B)^2 − Re A ∗ Re D) / (Re A)^2
  let ?r = sqrt ?r2

  have ?a = a ∧ ?r = r
    using ‹(a, r) = euclidean-circle H›
    using ∗ ‹is-real A› ‹is-real D› ‹C = cnj B› ‹A ≠ 0›
    apply simp
    apply transfer
   apply (simp add: euclidean-circle-rep-def mk-circline-rep-def Abs-circline-mat-inverse)
```

```
    apply (subst Re-divide-real)
    apply (simp-all add: cmod-square, simp add: power2-eq-square)
    done

  show ?thesis
    using * ** *** ‹Re A ≠ 0› ‹is-real A› ‹C = cnj B› ‹?a = a ∧ ?r = r›
    using classic-circline[of H A B C D] assms circline-circle[of Re A Re D B
circline (Re A) B (Re D) ?a ?r2 ?r]
    by (simp add: complex-of-real-Re circle-def)
qed

definition
  euclidean-line-rep H =
    (let (A, B, C, D) = Rep-circline-mat H;
         z1 = −(D∗B)/(2∗B∗C);
         z2 = z1 + ii∗sgn (if arg B > 0 then −B else B)
     in (z1, z2))

lift-definition euclidean-line :: circline ⇒ complex × complex is euclidean-line-rep
proof−
  fix H1 H2
  obtain A1 B1 C1 D1 where HH1: Rep-circline-mat H1 = (A1, B1, C1, D1)
    by (cases Rep-circline-mat H1) auto
  obtain A2 B2 C2 D2 where HH2: Rep-circline-mat H2 = (A2, B2, C2, D2)
    by (cases Rep-circline-mat H2) auto
  assume circline-mat-eq H1 H2
  then obtain k where k ≠ 0 and ∗: A2 = cor k ∗ A1 B2 = cor k ∗ B1 C2 =
cor k ∗ C1 D2 = cor k ∗ D1
    using HH1 HH2
    by auto
  have 1: B1 ≠ 0 ∧ 0 < arg B1 ⟶ ¬ 0 < arg (− B1)
    using MoreComplex.canon-ang-plus-pi1[of arg B1] arg-bounded[of B1]
    by (auto simp add: arg-uminus)
  have 2: B1 ≠ 0 ∧ ¬ 0 < arg B1 ⟶ 0 < arg (− B1)
    using MoreComplex.canon-ang-plus-pi2[of arg B1] arg-bounded[of B1]
    by (auto simp add: arg-uminus)

  show euclidean-line-rep H1 = euclidean-line-rep H2
    using HH1 HH2 ∗ ‹k ≠ 0›
    by (cases k > 0) (auto simp add: euclidean-line-rep-def Let-def, simp-all add:
sgn-eq arg-mult-real-positive arg-mult-real-negative 1 2)
qed

lemma classic-line:
  assumes is-line H (z1, z2) = euclidean-line H circline-type H < 0
  shows circline-set H − {∞h} = of-complex ‘ line z1 z2
proof−
  obtain A B C D where ∗: H = mk-circline A B C D hermitean (A, B, C, D)
(A, B, C, D) ≠ mat-zero
```

  using *ex-mk-circline*[*of H*]
  by *auto*
 have *is-real A is-real D C* = *cnj B*
  using ∗ *hermitean-elems*
  by *auto*
 have *Re A* = *0*
  using ‹*is-line H*›∗ ‹*is-real A*› ‹*is-real D*› ‹*C* = *cnj B*›
 by *transfer* (*auto simp add*: *circline-A0-rep-def mk-circline-rep-def Abs-circline-mat-inverse*)
 have *B* ≠ *0*
  using ‹*Re A* = *0*› ‹*is-real A*› ‹*is-real D*› ‹*C* = *cnj B*› ∗ ‹*circline-type H* < *0*›
 by *transfer* (*auto simp add*: *circline-type-rep-def mk-circline-rep-def Abs-circline-mat-inverse*, (*case-tac Rep-circline-mat H*, *simp*)+)

 let *?z1* = − *cor* (*Re D*) ∗ *B* / (*2* ∗ *B* ∗ *cnj B*)
 let *?z2* = *?z1* + i ∗ *sgn* (*if 0* < *arg B then* − *B else B*)
 have *z1* = *?z1* ∧ *z2* = *?z2*
  using ‹(*z1*, *z2*) = *euclidean-line H*› ∗ ‹*is-real A*› ‹*is-real D*› ‹*C* = *cnj B*›
 by *simp* (*transfer*, *simp add*: *euclidean-line-rep-def mk-circline-rep-def Abs-circline-mat-inverse Let-def complex-of-real-Re*)
 thus *?thesis*
  using ∗
  using *classic-circline*[*of H A B C D*] *circline-line*[*of Re A B circline* (*Re A*) *B* (*Re D*) *Re D ?z1 ?z2*] ‹*Re A* = *0*› ‹*B* ≠ *0*›
  by *simp*
qed

11.2 Connections with circles on the Riemann sphere

definition *inv-stereographic-circline-rep* **where**
 inv-stereographic-circline-rep H =
  (*let* (*A*, *B*, *C*, *D*) = *Rep-circline-mat H in*
  Abs-plane-vec (*Re* (*B*+*C*), *Re*(*ii*∗(*C*−*B*)), *Re*(*A*−*D*), *Re*(*D*+*A*)))

lemma *inv-stereographic-circline-rep-Rep* [*simp*]:
 Rep-plane-vec (*inv-stereographic-circline-rep H*) =
  (*let* (*A*, *B*, *C*, *D*) = *Rep-circline-mat H in* (*Re* (*B*+*C*), *Re*(*ii*∗(*C*−*B*)), *Re*(*A*−*D*), *Re*(*D*+*A*)))
proof−
 obtain *A B C D* **where** *HH*: *Rep-circline-mat H* = (*A*, *B*, *C*, *D*)
  by (*cases Rep-circline-mat H*) *auto*
 have ∗: *is-real A is-real D C* = *cnj B*
  using *HH Rep-circline-mat*[*of H*] *hermitean-elems*[*of A B C D*]
  by *auto*
 have *Re B* + *Re C* = *0* ∧ *Im B* − *Im C* = *0* ∧ *Re A* − *Re D* = *0* ∧ *Re A* + *Re D* = *0* ⟶ (*A*, *B*, *C*, *D*) = *mat-zero*
  using ∗
  by *auto* (*metis complex-of-real-Re of-real-0*)+
 hence ∗∗: *Re B* + *Re C* ≠ *0* ∨ *Im B* − *Im C* ≠ *0* ∨ *Re A* − *Re D* ≠ *0* ∨ *Re D* + *Re A* ≠ *0*

```
    using Rep-circline-mat[of H] HH
    by auto
  thus ?thesis
    using HH
    by (simp add: Abs-plane-vec-inverse inv-stereographic-circline-rep-def)
qed

lift-definition inv-stereographic-circline :: circline ⇒ plane is inv-stereographic-circline-rep
proof−
  fix H1 H2
  obtain A1 B1 C1 D1 where HH1: Rep-circline-mat H1 = (A1, B1, C1, D1)
    by (cases Rep-circline-mat H1) auto
  obtain A2 B2 C2 D2 where HH2: Rep-circline-mat H2 = (A2, B2, C2, D2)
    by (cases Rep-circline-mat H2) auto
  have ∗: is-real A1 is-real A2 is-real D1 is-real D2 C1 = cnj B1 C2 = cnj B2
   using HH1 HH2 Rep-circline-mat[of H1] Rep-circline-mat[of H2] hermitean-elems[of A1 B1 C1 D1]  hermitean-elems[of A2 B2 C2 D2]
    by auto

  assume circline-mat-eq H1 H2
 thus plane-vec-eq (inv-stereographic-circline-rep H1) (inv-stereographic-circline-rep H2)
    using HH1 HH2 ∗
     by (simp add: plane-vec-eq-def) (erule exE, rule-tac x=k in exI, simp add: field-simps)
qed

definition stereographic-circline-rep where
stereographic-circline-rep α =
     (let (a, b, c, d) = Rep-plane-vec α in
         Abs-circline-mat (cor ((c+d)/2) , ((cor a+ii∗ cor b)/2), ((cor a−ii∗cor b)/2), cor ((d−c)/2)))

lemma stereographic-circline-rep-Rep:
  Rep-circline-mat (stereographic-circline-rep α) =
      (let (a, b, c, d) = Rep-plane-vec α in
            (cor ((c+d)/2) , ((cor a+ii∗ cor b)/2), ((cor a−ii∗cor b)/2), cor ((d−c)/2)))
proof−
  obtain a b c d where AA: (a, b, c, d) = Rep-plane-vec α
    by (cases Rep-plane-vec α) auto
   let ?M = (cor ((c+d)/2) , ((cor a+ii∗ cor b)/2), ((cor a−ii∗cor b)/2), cor ((d−c)/2))
  have ?M ∈ {M. hermitean M ∧ M ≠ mat-zero}
    using Rep-plane-vec[of α] AA
   by (auto simp add: hermitean-def mat-adj-def mat-cnj-def complex-cnj complex-of-real-def)
  thus ?thesis
    using AA[symmetric]
    by (simp add: Abs-circline-mat-inverse stereographic-circline-rep-def)
```

```
qed

lift-definition stereographic-circline :: plane ⇒ circline is stereographic-circline-rep
proof−
  fix α1 α2
  assume plane-vec-eq α1 α2
   thus circline-mat-eq (stereographic-circline-rep α1) (stereographic-circline-rep
α2)
    apply (cases Rep-plane-vec α2, cases Rep-plane-vec α1)
    apply (auto simp add: plane-vec-eq-def stereographic-circline-rep-Rep)
    apply (rule-tac x=k in exI, simp add: field-simps)
   by (metis (hide-lams, mono-tags) comm-semiring-1-class.normalizing-semiring-rules(19)
complex-of-real-mult-Complex mult-zero-right)
qed

lemma stereographic-circline-inv-stereographic-circline:
  stereographic-circline ∘ inv-stereographic-circline = id
proof (rule ext, simp)
  fix H
  show stereographic-circline (inv-stereographic-circline H) = H
  proof transfer
    fix H
    obtain A B C D where HH: Rep-circline-mat H = (A, B, C, D)
      by (cases Rep-circline-mat H) auto
    have is-real A is-real D C = cnj B
      using HH Rep-circline-mat[of H] hermitean-elems[of A B C D]
      by auto
     thus circline-mat-eq (stereographic-circline-rep (inv-stereographic-circline-rep
H)) H
      using HH
      apply (simp add: stereographic-circline-rep-Rep)
      apply (rule-tac x=1 in exI)
      apply (auto simp add: complex-of-real-Re of-real-numeral)
      apply (cases B, simp)
        apply (cases B, simp add: complex-of-real-def, metis Im.simps Re.simps
comm-semiring-1-class.normalizing-semiring-rules(4) complex-diff-def complex-minus-def
complex-of-real-add-Complex complex-of-real-def minus-zero monoid-add-class.add.right-neutral
one-add-one)
      done
  qed
qed

lemma [simp]: Im (z / 2) = Im z / 2
by (subst Im-divide-real, auto)

lemma [simp]: (Complex a b) / 2 = Complex (a/2) (b/2)
by (subst complex-eq-iff) auto

lemma [simp]: Complex 2 0 = 2
```

```
by simp

lemma inv-stereographic-circline-stereographic-circline:
  inv-stereographic-circline ∘ stereographic-circline = id
proof (rule ext, simp)
  fix α
  show inv-stereographic-circline (stereographic-circline α) = α
  proof transfer
    fix α
    obtain a b c d where AA: Rep-plane-vec α = (a, b, c, d)
      by (cases Rep-plane-vec α) auto
    thus plane-vec-eq (inv-stereographic-circline-rep (stereographic-circline-rep α))
α
      using AA
      by (simp add: plane-vec-eq-def stereographic-circline-rep-Rep) (rule-tac x=1
in exI, auto simp add: field-simps complex-of-real-def)
  qed
qed

lemma stereographic-sphere-circle-set'':
  on-sphere-circle (inv-stereographic-circline H) z ⟷ on-circline H (stereographic
z)
proof
  assume on-sphere-circle (inv-stereographic-circline H) z
  thus on-circline H (stereographic z)
  proof transfer
    fix M H
    obtain A B C D where HH: Rep-circline-mat H = (A, B, C, D)
      by (cases Rep-circline-mat H) auto
    have ∗: is-real A is-real D C = cnj B
      using Rep-circline-mat[of H] HH hermitean-elems[of A B C D]
      by auto
    obtain x y z where MM: Rep-riemann-sphere M = (x, y, z)
      by (cases Rep-riemann-sphere M) auto
    assume ∗∗: on-sphere-circle-rep (inv-stereographic-circline-rep H) M
    show on-circline-rep H (stereographic-coords M)
    proof (cases z=1)
      case True
      hence x = 0 y = 0
        using MM Rep-riemann-sphere[of M]
        by auto
      thus ?thesis
        using ∗ ∗∗ HH MM ‹z=1›
      by (cases A, simp add: on-circline-rep-def stereographic-coords-rep on-sphere-circle-rep-def
vec-cnj-def Let-def)
    next
      case False
      hence Re A∗(1+z) + 2∗Re B∗x + 2∗Im B∗y + Re D∗(1−z) = 0
        using ∗ ∗∗ HH MM
        by (simp add: on-sphere-circle-rep-def Let-def field-simps)
      hence (Re A∗(1+z) + 2∗Re B∗x + 2∗Im B∗y + Re D∗(1−z))∗(1−z) = 0
        by simp
      hence Re A∗(1+z)∗(1−z) + 2∗Re B∗x∗(1−z) + 2∗Im B∗y∗(1−z) + Re
D∗(1−z)∗(1−z) = 0
        by (simp add: field-simps)
      moreover
      have x∗x+y∗y = (1+z)∗(1−z)
        using MM Rep-riemann-sphere[of M]
        by (simp add: field-simps)
      ultimately
      have Re A∗(x∗x+y∗y) + 2∗Re B∗x∗(1−z) + 2∗Im B∗y∗(1−z) + Re
D∗(1−z)∗(1−z) = 0
        by simp
      hence (x ∗ Re A + (1 − z) ∗ Re B) ∗ x − (− (y ∗ Re A) + − ((1 − z) ∗
Im B)) ∗ y + (x ∗ Re B + y ∗ Im B + (1 − z) ∗ Re D) ∗ (1 − z) = 0
        by (simp add: field-simps)
      thus ?thesis
        using ⟨z ≠ 1⟩ HH MM ∗ ⟨Re A∗(1+z) + 2∗Re B∗x + 2∗Im B∗y + Re
D∗(1−z) = 0⟩
      apply (simp add: on-circline-rep-def stereographic-coords-rep Let-def vec-cnj-def
complex-cnj)
        apply (subst complex-eq-iff)
        apply (simp add: field-simps)
        done
    qed
  qed
next
  assume on-circline H (stereographic z)
  thus on-sphere-circle (inv-stereographic-circline H) z
  proof transfer
    fix H M
    fix M H
    obtain A B C D where HH: Rep-circline-mat H = (A, B, C, D)
      by (cases Rep-circline-mat H) auto
    have ∗: is-real A is-real D C = cnj B
      using Rep-circline-mat[of H] HH hermitean-elems[of A B C D]
      by auto
    obtain x y z where MM: Rep-riemann-sphere M = (x, y, z)
      by (cases Rep-riemann-sphere M) auto
    assume ∗∗: on-circline-rep H (stereographic-coords M)
    show on-sphere-circle-rep (inv-stereographic-circline-rep H) M
    proof (cases z=1)
      case True
      hence x = 0 y = 0
        using MM Rep-riemann-sphere[of M]
        by auto
      thus ?thesis
        using HH MM ∗∗ ⟨z = 1⟩
        by (simp add: on-sphere-circle-rep-def on-circline-rep-def Let-def vec-cnj-def
stereographic-coords-rep)
    next
      case False
      hence (x ∗ Re A + (1 − z) ∗ Re B) ∗ x − (− (y ∗ Re A) + − ((1 − z) ∗
Im B)) ∗ y + (x ∗ Re B + y ∗ Im B + (1 − z) ∗ Re D) ∗ (1 − z) = 0
        using HH MM ∗ ∗∗
        by (simp add: on-circline-rep-def Let-def vec-cnj-def stereographic-coords-rep
complex-eq-iff)
        hence Re A∗(x∗x+y∗y) + 2∗Re B∗x∗(1−z) + 2∗Im B∗y∗(1−z) + Re
D∗(1−z)∗(1−z) = 0
        by (simp add: field-simps)
      moreover
      have x∗x + y∗y = (1+z)∗(1−z)
        using MM Rep-riemann-sphere[of M]
        by (simp add: field-simps)
      ultimately
        have Re A∗(1+z)∗(1−z) + 2∗Re B∗x∗(1−z) + 2∗Im B∗y∗(1−z) + Re
D∗(1−z)∗(1−z) = 0
        by simp
      hence (Re A∗(1+z) + 2∗Re B∗x + 2∗Im B∗y + Re D∗(1−z))∗(1−z) = 0
        by (simp add: field-simps)
      hence Re A∗(1+z) + 2∗Re B∗x + 2∗Im B∗y + Re D∗(1−z) = 0
        using ‹z ≠ 1›
        by simp
      thus ?thesis
        using MM HH ∗
        by (simp add: on-sphere-circle-rep-def field-simps)
    qed
  qed
qed

lemma stereographic-sphere-circle-set′:
  stereographic ‘ sphere-circle-set (inv-stereographic-circline H) = circline-set H
unfolding sphere-circle-set-def circline-set-def
apply safe
proof−
  fix x
  assume on-sphere-circle (inv-stereographic-circline H) x
  thus on-circline H (stereographic x)
    using stereographic-sphere-circle-set′′
    by simp
next
  fix x
  assume on-circline H x
  show x ∈ stereographic ‘ {z. on-sphere-circle (inv-stereographic-circline H) z}
  proof
    show x = stereographic (inv-stereographic x)
      by (simp add: stereographic-inv-stereographic)
  next
    show inv-stereographic x ∈ {z. on-sphere-circle (inv-stereographic-circline H) z}
      using stereographic-sphere-circle-set''[of H inv-stereographic x] ‹on-circline H x›
      by (simp add: stereographic-inv-stereographic)
  qed
qed

lemma stereographic-sphere-circle-set:
  shows stereographic ‘ sphere-circle-set H = circline-set (stereographic-circline H)
using stereographic-sphere-circle-set'[of stereographic-circline H]
using inv-stereographic-circline-stereographic-circline
unfolding comp-def
by (metis id-apply)

lemma bij stereographic-circline
using stereographic-circline-inv-stereographic-circline inv-stereographic-circline-stereographic-circline
by (metis bij-def image-compose inj-iff inj-imp-surj-inv inj-on-imageI2 inv-id surj-id surj-iff)

lemma bij inv-stereographic-circline
using stereographic-circline-inv-stereographic-circline inv-stereographic-circline-stereographic-circline
by (metis bij-def image-compose inj-iff inj-imp-surj-inv inj-on-imageI2 inv-id surj-id surj-iff)
```

11.3 Some special circlines

Unit circle

```
definition unit-circle-rep where
  [simp]: unit-circle-rep = Abs-circline-mat (1, 0, 0, −1)

lemma [simp]: Rep-circline-mat (Abs-circline-mat (1, 0, 0, −1)) = (1, 0, 0, −1)
by (auto simp add: Abs-circline-mat-inverse hermitean-def mat-adj-def mat-cnj-def)

lemma [simp]: Rep-circline-mat unit-circle-rep = (1, 0, 0, −1)
by simp

lift-definition unit-circle :: circline is unit-circle-rep
done

lemma one-on-unit-circle: 1_h ∈ circline-set unit-circle
unfolding circline-set-def
by (simp, transfer, simp add: on-circline-rep-def Let-def vec-cnj-def)
```

x-axis

```
definition x-axis-rep where x-axis-rep = Abs-circline-mat (0, ii, −ii, 0)
lift-definition x-axis :: circline is x-axis-rep
done
```

lemma [*simp*]: *Rep-circline-mat* (*Abs-circline-mat* (*0*, *ii*, $-$*ii*, *0*)) = (*0*, *ii*, $-$*ii*, *0*)
using *Abs-circline-mat-inverse*[*of* (*0*, *ii*, $-$*ii*, *0*)]
by (*simp add*: *hermitean-def mat-adj-def mat-cnj-def complex-cnj*)

lemma [*simp*]: *Rep-circline-mat x-axis-rep* = (*0*, *ii*, $-$*ii*, *0*)
unfolding *x-axis-rep-def*
by *simp*

lemma [*simp*]: 0_h ∈ *circline-set x-axis* 1_h ∈ *circline-set x-axis* ∞_h ∈ *circline-set x-axis*
  unfolding *circline-set-def*
  by *auto* (*transfer*, *simp add*: *on-circline-rep-def Let-def vec-cnj-def*)+

Point 0_h as a circline

definition *circline-point-0h-rep* **where** *circline-point-0h-rep* = *Abs-circline-mat* (*1*, *0*, *0*, *0*)

lift-definition *circline-point-0h* :: *circline* **is** *circline-point-0h-rep*
done

lemma [*simp*]: *Rep-circline-mat* (*Abs-circline-mat* (*1*, *0*, *0*, *0*)) = (*1*, *0*, *0*, *0*)
using *Abs-circline-mat-inverse*
by (*simp add*: *hermitean-def mat-adj-def mat-cnj-def*)

lemma [*simp*]: *Rep-circline-mat circline-point-0h-rep* = (*1*, *0*, *0*, *0*)
unfolding *circline-point-0h-rep-def*
by *simp*

imaginary unit circle

definition *imag-unit-circle-rep* **where**
  [*simp*]: *imag-unit-circle-rep* = *Abs-circline-mat* (*1*, *0*, *0*, *1*)

lemma [*simp*]: *Rep-circline-mat* (*Abs-circline-mat* (*1*, *0*, *0*, *1*)) = (*1*, *0*, *0*, *1*)
by (*auto simp add*: *Abs-circline-mat-inverse hermitean-def mat-adj-def mat-cnj-def*)

lemma [*simp*]: *Rep-circline-mat imag-unit-circle-rep* = (*1*, *0*, *0*, *1*)
by *simp*

lift-definition *imag-unit-circle* :: *circline* **is** *imag-unit-circle-rep*
done

11.4 Moebius action on circlines

definition *moebius-circline-rep* :: *moebius-mat* ⇒ *circline-mat* ⇒ *circline-mat* **where**

  moebius-circline-rep M H =
    (*let M* = *Rep-moebius-mat M*;

```
      H = Rep-circline-mat H
    in Abs-circline-mat (congruence (mat-inv M) H))

lemma [simp]: Rep-circline-mat (Abs-circline-mat (congruence (mat-inv (Rep-moebius-mat
M)) (Rep-circline-mat H))) = congruence (mat-inv (Rep-moebius-mat M)) (Rep-circline-mat
H)
proof (rule Abs-circline-mat-inverse, safe)
  show hermitean (congruence (mat-inv (Rep-moebius-mat M)) (Rep-circline-mat
H))
    using Rep-circline-mat[of H]
    using hermitean-congruence
    by simp
next
  assume congruence (mat-inv (Rep-moebius-mat M)) (Rep-circline-mat H) =
mat-zero
  thus False
    using Rep-circline-mat[of H] Rep-moebius-mat[of M] mat-det-inv
    using congruence-nonzero
    by auto
qed

lemma moebius-circline-rep-Rep [simp]: Rep-circline-mat (moebius-circline-rep M
H) = congruence (mat-inv (Rep-moebius-mat M)) (Rep-circline-mat H)
by (simp add: moebius-circline-rep-def Let-def)

lift-definition moebius-circline :: moebius ⇒ circline ⇒ circline is moebius-circline-rep
proof−
  fix M M′ H H′
  assume moebius-mat-eq M M′ circline-mat-eq H H′
  thus circline-mat-eq (moebius-circline-rep M H) (moebius-circline-rep M′ H′)
    by (auto simp add: mat-inv-mult-sm complex-cnj) (rule-tac x=ka / Re (k ∗ cnj
k) in exI, auto simp add: complex-mult-cnj-cmod power2-eq-square)
qed

lemma moebius-preserve-circline-type:
  shows circline-type (moebius-circline M H) = circline-type H
proof (transfer)
  fix M H
  show circline-type-rep (moebius-circline-rep M H) = circline-type-rep H
    unfolding circline-type-rep-def Let-def
    apply simp
    using Re-det-sgn-congruence[of Rep-circline-mat H mat-inv (Rep-moebius-mat
M)]
   using Rep-circline-mat[of H] Rep-moebius-mat[of M] mat-det-inv[of Rep-moebius-mat
M]
    by simp
qed

lemma moebius-circline-rep:
 shows moebius-pt-rep M ‘ {z. on-circline-rep H z} = {z. on-circline-rep (moebius-circline-rep M H) z}
proof (safe)
  fix z
  let ?M = Rep-moebius-mat M
  let ?H = Rep-circline-mat H
  let ?z = Rep-homo-coords z
  let ?H' = Rep-circline-mat H'
  let ?z' = Rep-moebius-mat M *_{mv} Rep-homo-coords z
  let ?H'' = mat-adj (mat-inv ?M) *_{mm} ?H *_{mm} (mat-inv ?M)
  assume on-circline-rep H z
  hence quad-form ?z ?H = 0
    by (simp add: on-circline-rep-def Let-def)
  hence quad-form ?z' ?H'' = 0
    using quad-form-congruence[of ?M ?z ?H] Rep-moebius-mat[of M]
    by simp
  thus on-circline-rep (moebius-circline-rep M H) (moebius-pt-rep M z)
   by (auto simp add: moebius-circline-rep-def on-circline-rep-def moebius-pt-rep-def Let-def)
next
  fix z
  let ?z = Rep-homo-coords z
  let ?M = Rep-moebius-mat M
  let ?H = Rep-circline-mat H
  let ?iM = mat-inv ?M
  let ?z' = mat-inv ?M *_{mv} ?z

  assume on-circline-rep (moebius-circline-rep M H) z
  hence quad-form ?z (congruence (mat-inv ?M) ?H) = 0
    unfolding on-circline-rep-def Let-def
    by simp

  have ?z' ≠ (0, 0)
     using Rep-homo-coords[of z] mult-mv-nonzero[of ?z ?iM] Rep-moebius-mat[of M] mat-det-inv[of ?M]
    by simp
  hence *: Rep-homo-coords (Abs-homo-coords ?z') = ?z'
    by (simp add: Abs-homo-coords-inverse)

  show z ∈ moebius-pt-rep M ‘ {z. on-circline-rep H z}
  proof
    show z = moebius-pt-rep M (Abs-homo-coords ?z')
      using * Rep-moebius-mat[of M] eye-mv-l[of ?z]
      unfolding moebius-pt-rep-def Let-def
      by (simp add: mat-inv-r Rep-homo-coords-inverse)
  next
   have Rep-moebius-mat M *_{mm} mat-inv (Rep-moebius-mat M) *_{mv} Rep-homo-coords z = ?z
      using Rep-moebius-mat[of M]
```

by (*subst mat-inv-r*) (*auto simp add*: *simp del*: *eye-def*)
thus *Abs-homo-coords ?z′* ∈ {*z*. *on-circline-rep H z*}
using ∗
using ‹*quad-form ?z* (*congruence* (*mat-inv ?M*) *?H*) = *0*› *Rep-moebius-mat*[*of M*]
by (*auto simp add*: *on-circline-rep-def Let-def simp del*: *quad-form-def*) (*subst quad-form-congruence*[*of ?M ?iM* $*_{mv}$ *?z ?H*, *symmetric*], *auto*)
qed
qed

lemma *moebius-circline-set*:
shows *moebius-pt M* ‘ *circline-set H* = *circline-set* (*moebius-circline M H*) (**is** *?lhs* = *?rhs*)
proof
show *?lhs* ⊆ *?rhs*
proof (*safe*)
fix *z*::*complex-homo*
assume *z* ∈ *circline-set H*
thus *moebius-pt M z* ∈ *circline-set* (*moebius-circline M H*)
unfolding *circline-set-def*
using *moebius-circline-rep*
by *simp* (*transfer*, *auto*)
qed
next
show *?rhs* ⊆ *?lhs*
proof
fix *z*
assume *z* ∈ *circline-set* (*moebius-circline M H*)
thus *z* ∈ *moebius-pt M* ‘ *circline-set H*
using *assms*
unfolding *circline-set-def*
apply (*simp add*: *image-def*)
proof (*transfer*)
fix *M H z*
assume *on-circline-rep* (*moebius-circline-rep M H*) *z*
then obtain *z′* **where** *on-circline-rep H z′ z* = *moebius-pt-rep M z′*
using *moebius-circline-rep*[*of M H*]
by *auto*
thus ∃ *z′*. *on-circline-rep H z′* ∧ *z* ≈ *moebius-pt-rep M z′*
by (*rule-tac x*=*z′* **in** *exI*, *simp*, *rule-tac x*=*1* **in** *exI*, *simp*)
qed
qed
qed

lemma
inj-moebius-circline: *inj* (*moebius-circline M*)
unfolding *inj-on-def*
proof (*safe*)
fix *H H′*

assume *moebius-circline M H* = *moebius-circline M H′*
thus *H* = *H′*
proof (*transfer*)
fix *M H H′*
let *?M* = *Rep-moebius-mat M*
let *?iM* = *mat-inv ?M*
let *?H* = *Rep-circline-mat H* **and** *?H′* = *Rep-circline-mat H′*
assume *circline-mat-eq* (*moebius-circline-rep M H*) (*moebius-circline-rep M H′*)
then obtain *k* **where** *congruence ?iM ?H′* = *congruence ?iM* (*cor k* $*_{sm}$ *?H*) *k* ≠ *0*
by *auto*
thus *circline-mat-eq H H′*
using *Rep-moebius-mat*[*of M*] *inj-congruence*[*of ?iM ?H′ cor k* $*_{sm}$ *?H*] *mat-det-inv*[*of ?M*]
by *auto*
qed
qed

lemma [*simp*]:
moebius-circline id-moebius H = *H*
proof *transfer*
fix *H*
show *circline-mat-eq* (*moebius-circline-rep id-moebius-rep H*) *H*
by (*cases Rep-circline-mat H*, *simp*) (*rule-tac x=1* **in** *exI*, *simp add*: *mat-adj-def mat-cnj-def*)
qed

lemma *moebius-circline-comp*:
moebius-circline M1 (*moebius-circline M2 H*) = *moebius-circline* (*moebius-comp M1 M2*) *H*
proof (*transfer*)
fix *M1 M2 H*
show *circline-mat-eq* (*moebius-circline-rep M1* (*moebius-circline-rep M2 H*)) (*moebius-circline-rep* (*moebius-comp-rep M1 M2*) *H*)
using *congruence-congruence Rep-moebius-mat*[*of M1*] *Rep-moebius-mat*[*of M2*]
by (*simp add*: *mat-inv-mult-mm*, *rule-tac x=1* **in** *exI*, *simp*)
qed

lemma *moebius-circline-comp-inv* [*simp*]:
moebius-circline (*moebius-inv M*) (*moebius-circline M H*) = *H*
by (*subst moebius-circline-comp*) *simp*

lemma *moebius-circline-comp-inv′* [*simp*]:
moebius-circline M (*moebius-circline* (*moebius-inv M*) *H*) = *H*
by (*subst moebius-circline-comp*) *simp*

lemma
moebius-circline-set-mem:

```
  moebius-pt M z ∈ circline-set (moebius-circline M H) ⟷ z ∈ circline-set H
using moebius-circline-set[of M H, symmetric] bij-moebius-pt[of M]
by (auto simp add: bij-def inj-on-def)
```

11.5 Conjugation, recpiprocation and inversion of circlines

Conjugation of circlines

```
definition circline-cnj-rep where
  circline-cnj-rep H = Abs-circline-mat (mat-cnj (Rep-circline-mat H))

lemma [simp]: Rep-circline-mat (Abs-circline-mat (mat-cnj (Rep-circline-mat H)))
= mat-cnj (Rep-circline-mat H)
using Rep-circline-mat[of H] hermitean-mat-cnj nonzero-mat-cnj
by (auto simp add: Abs-circline-mat-inverse)

lemma [simp]: Rep-circline-mat (circline-cnj-rep H) = mat-cnj (Rep-circline-mat
H)
by (simp add: circline-cnj-rep-def)

lift-definition circline-cnj :: circline ⇒ circline is circline-cnj-rep
proof−
  fix H H'
  assume circline-mat-eq H H'
  thus circline-mat-eq (circline-cnj-rep H) (circline-cnj-rep H')
    using Rep-circline-mat[of H] Rep-circline-mat[of H']
    by auto
qed

lemma cnj-homo-circline-set':
  shows cnj-homo ` circline-set H ⊆ circline-set (circline-cnj H)
proof (safe)
  fix z
  assume z ∈ circline-set H
  thus cnj-homo z ∈ circline-set (circline-cnj H)
    unfolding circline-set-def
    apply simp
  proof (transfer)
    fix z H
    assume on-circline-rep H z
    obtain z1 z2 where zz: Rep-homo-coords z = (z1, z2)
      by (rule obtain-homo-coords)

    have (cnj z1, cnj z2) *vm Rep-circline-mat H *vv (z1, z2) = 0
      using ‹on-circline-rep H z› zz
      unfolding on-circline-rep-def Let-def
      by (simp add: vec-cnj-def)
    hence cnj ((cnj z1, cnj z2) *vm Rep-circline-mat H *vv (z1, z2)) = 0
      by simp
    hence (z1, z2) *vm mat-cnj (Rep-circline-mat H) *vv (cnj z1, cnj z2) = 0
```

by (*subst* (*asm*) *cnj-mult-vv*) (*cases Rep-circline-mat H*, *simp add*: *vec-cnj-def mat-cnj-def complex-cnj*)
thus *on-circline-rep* (*circline-cnj-rep H*) (*cnj-homo-coords z*)
unfolding *on-circline-rep-def Let-def*
using *zz*
by (*simp add*: *vec-cnj-def*)
qed
qed

lemma [*simp*]: *circline-cnj* (*circline-cnj H*) = *H*
by (*transfer*) (*auto simp add*: *circline-cnj-rep-def Rep-circline-mat-inverse*, *rule-tac x=1* **in** *exI*, *simp*)

lemma *cnj-homo-circline-set*:
shows *cnj-homo* ‘ *circline-set H* = *circline-set* (*circline-cnj H*) (**is** *?lhs* = *?rhs*)
proof (*safe*)
fix *z*
assume *z* ∈ *circline-set* (*circline-cnj H*)
show *z* ∈ *cnj-homo* ‘ *circline-set H*
proof
show *z* = *cnj-homo* (*cnj-homo z*)
by *simp*
next
show *cnj-homo z* ∈ *circline-set H*
using ‹*z* ∈ *circline-set* (*circline-cnj H*)›
using *cnj-homo-circline-set'*[*of circline-cnj H*]
by *auto*
qed
next
fix *z*
assume *z* ∈ *circline-set H*
thus *cnj-homo z* ∈ *circline-set* (*circline-cnj H*)
using *cnj-homo-circline-set'*[*of H*]
by *auto*
qed

Reciprocal and inversion of circlines

definition *circline-swap-AD-rep* **where**
circline-swap-AD-rep H =
(*let* (*A*, *B*, *C*, *D*) = *Rep-circline-mat H*
in Abs-circline-mat (*D*, *B*, *C*, *A*))

lemma
shows [*simp*]: *Rep-circline-mat* (*circline-swap-AD-rep H*) = (*let* (*A*, *B*, *C*, *D*) = *Rep-circline-mat H in* (*D*, *B*, *C*, *A*))
proof−
obtain *A B C D* **where** *hh*: *Rep-circline-mat H* = (*A*, *B*, *C*, *D*)
by (*cases Rep-circline-mat H*) *auto*
have *hermitean* (*D*, *B*, *C*, *A*) ∧ (*D*, *B*, *C*, *A*) ≠ *mat-zero*

using *Rep-circline-mat*[*of H*] *hh*
by (*auto simp add*: *hermitean-def mat-adj-def mat-cnj-def*)
thus *?thesis*
using *hh*
unfolding *circline-swap-AD-rep-def Let-def*
by (*cases Rep-circline-mat H*) (*simp add*: *Abs-circline-mat-inverse*)
qed

lift-definition *circline-swap-AD* :: *circline* ⇒ *circline* **is** *circline-swap-AD-rep*
proof−
fix *H H'* :: *circline-mat*
assume *circline-mat-eq H H'*
thus *circline-mat-eq* (*circline-swap-AD-rep H*) (*circline-swap-AD-rep H'*)
by (*cases Rep-circline-mat H*, *cases Rep-circline-mat H'*) *auto*
qed

lemma *reciprocal-circline-set*:
shows *reciprocal-homo* ‘ *circline-set H* = *circline-set* ((*circline-cnj* ∘ *circline-swap-AD*) *H*)
proof (*subst reciprocal-moebius*, *subst moebius-circline-set*)
have *moebius-circline reciprocal-moebius H* = (*circline-cnj* ∘ *circline-swap-AD*) *H*
unfolding *reciprocal-moebius-def*
proof (*transfer*)
fix *H* :: *circline-mat*
obtain *A B C D* **where** *H*: *Rep-circline-mat H* = (*A*, *B*, *C*, *D*)
by (*cases Rep-circline-mat H*) *blast*
thus *circline-mat-eq* (*moebius-circline-rep* (*mk-moebius-rep 0 1 1 0*) *H*) ((*circline-cnj-rep* ∘ *circline-swap-AD-rep*) *H*)
using *Rep-circline-mat*[*of H*]
by (*simp add*: *mat-adj-def mat-cnj-def hermitean-def*) (*rule-tac x=1* **in** *exI*, *simp*)
qed
thus *circline-set* (*moebius-circline reciprocal-moebius H*) = *circline-set* ((*circline-cnj* ∘ *circline-swap-AD*) *H*)
by *simp*
qed

lemma *inversion-circline-set*:
shows *inversion-homo* ‘ *circline-set H* = *circline-set* (*circline-swap-AD H*)
unfolding *inversion-homo-def image-comp*
by (*subst reciprocal-circline-set*, *subst cnj-homo-circline-set*, *rule arg-cong*[**where** *f=circline-set*]) *simp*

11.6 Circline uniqueness

11.6.1 Zero type circline uniqueness

lemma *unique-circline-type-zero-0h'*:
shows (*circline-type circline-point-0h* = *0* ∧ 0_h ∈ *circline-set circline-point-0h*)

```
∧
      (∀ H. circline-type H = 0 ∧ 0_h ∈ circline-set H ⟶ H = circline-point-0h)
unfolding circline-set-def
proof (safe)
  show circline-type circline-point-0h = 0
    by (transfer) (simp add: circline-type-rep-def circline-point-0h-rep-def)
next
  show on-circline circline-point-0h 0_h
    by (transfer) (simp add: on-circline-rep-def Let-def vec-cnj-def)
next
  fix H
  assume circline-type H = 0 on-circline H 0_h
  thus H = circline-point-0h
  proof (transfer)
    fix H
    obtain A B C D where HH: Rep-circline-mat H = (A, B, C, D)
      by (cases Rep-circline-mat H) auto
    hence ∗: C = cnj B is-real A
      using Rep-circline-mat[of H] hermitean-elems[of A B C D]
      by auto
    assume circline-type-rep H = 0 on-circline-rep H zero-homo-rep
    thus circline-mat-eq H circline-point-0h-rep
      using ∗ Rep-circline-mat[of H] HH
      by (simp add: on-circline-rep-def Let-def Abs-circline-mat-inverse vec-cnj-def
circline-type-rep-def sgn-minus sgn-mult sgn-zero-iff)
        (rule-tac x=1/Re A in exI, cases A, cases B, simp add: complex-of-real-Re
sgn-zero-iff)
  qed
qed

lemma unique-circline-type-zero-0h:
  shows ∃! H. circline-type H = 0 ∧ 0_h ∈ circline-set H
using unique-circline-type-zero-0h'
by auto

lemma unique-circline-type-zero:
  shows ∃! H. circline-type H = 0 ∧ z ∈ circline-set H
proof−
  obtain M where ++: moebius-pt M z = 0_h
    using ex-moebius-1[of z]
    by auto
  have +++: z = moebius-pt (moebius-inv M) 0_h
    by (subst ++[symmetric]) simp
  then obtain H0 where ∗: circline-type H0 = 0 ∧ 0_h ∈ circline-set H0 and
    ∗∗: ∀ H'. circline-type H' = 0 ∧ 0_h ∈ circline-set H' ⟶ H' = H0
    using unique-circline-type-zero-0h
    by auto
  let ?H' = moebius-circline (moebius-inv M) H0
  show ?thesis
```

```
    unfolding Ex1-def
    using * +++
  proof (rule-tac x=?H' in exI, simp add: moebius-preserve-circline-type moebius-circline-set[symmetric],
safe)
    fix H'
    assume circline-type H' = 0 moebius-pt (moebius-inv M) 0_h ∈ circline-set H'
    hence 0_h ∈ circline-set (moebius-circline M H')
      by (metis ++ +++ imageI moebius-circline-set)
    hence moebius-circline M H' = H0
      using **[rule-format, of moebius-circline M H']
      using moebius-preserve-circline-type[of M H'] ‹circline-type H' = 0›
      by simp
    thus H' = moebius-circline (moebius-inv M) H0
      by auto
  qed
qed
```

11.6.2 Negative type circline uniqueness

```
lemma unique-circline-01inf':
  0_h ∈ circline-set x-axis ∧ 1_h ∈ circline-set x-axis ∧ ∞_h ∈ circline-set x-axis ∧
   (∀ H. 0_h ∈ circline-set H ∧ 1_h ∈ circline-set H ∧ ∞_h ∈ circline-set H ⟶ H
= x-axis)
proof safe
  fix H
  assume 0_h ∈ circline-set H 1_h ∈ circline-set H ∞_h ∈ circline-set H
  thus H = x-axis
    unfolding circline-set-def
    apply simp
  proof (transfer)
    fix H
    obtain A B C D where HH: Rep-circline-mat H = (A, B, C, D)
      by (cases Rep-circline-mat H) auto
    have *: C = cnj B A = 0 ∧ D = 0 ⟶ B ≠ 0
      using hermitean-elems[of A B C D] Rep-circline-mat[of H] HH
      by auto
    obtain Bx By where B = Complex Bx By
      by (cases B) auto
    assume on-circline-rep H zero-homo-rep on-circline-rep H one-homo-rep on-circline-rep
H inf-homo-rep
    thus circline-mat-eq H x-axis-rep
      using * HH ‹C = cnj B› ‹B = Complex Bx By›
     by (simp add: on-circline-rep-def Let-def mk-circline-rep-def Abs-circline-mat-inverse
vec-cnj-def complex-of-real-def i-def, rule-tac x=1/By in exI, auto )
  qed
qed simp-all

lemma unique-circline-set:
  assumes A ≠ B A ≠ C B ≠ C
  shows ∃! H. A ∈ circline-set H ∧ B ∈ circline-set H ∧ C ∈ circline-set H
proof−
  let ?P = λ A B C. A ≠ B ∧ A ≠ C ∧ B ≠ C ⟶ (∃! H. A ∈ circline-set H ∧ B ∈ circline-set H ∧ C ∈ circline-set H)
  have ?P A B C
  proof (rule wlog-moebius-01inf[of ?P])
    fix M a b c
    let ?M = moebius-pt M
    assume ?P a b c
    show ?P (?M a) (?M b) (?M c)
    proof
      assume ?M a ≠ ?M b ∧ ?M a ≠ ?M c ∧ ?M b ≠ ?M c
      hence a ≠ b b ≠ c a ≠ c
        using bij-moebius-pt[of M]
        by (auto simp add: bij-def inj-on-def)
      hence ∃!H. a ∈ circline-set H ∧ b ∈ circline-set H ∧ c ∈ circline-set H
        using ‹?P a b c›
        by simp
      then obtain H where
        ∗: a ∈ circline-set H ∧ b ∈ circline-set H ∧ c ∈ circline-set H and
        ∗∗: ∀H'. a ∈ circline-set H' ∧ b ∈ circline-set H' ∧ c ∈ circline-set H' ⟶ H' = H
        unfolding Ex1-def
        by auto
      let ?H' = moebius-circline M H
       show ∃! H. ?M a ∈ circline-set H ∧ moebius-pt M b ∈ circline-set H ∧ moebius-pt M c ∈ circline-set H
        unfolding Ex1-def
      proof (rule-tac x=?H' in exI, rule)
      show ?M a ∈ circline-set ?H' ∧ ?M b ∈ circline-set ?H' ∧ ?M c ∈ circline-set ?H'
          using ∗ moebius-circline-set-mem[of M - H]
          by blast
      next
         show ∀H'. ?M a ∈ circline-set H' ∧ ?M b ∈ circline-set H' ∧ ?M c ∈ circline-set H' ⟶ H' = ?H'
        proof (safe)
          fix H'
          let ?iH' = moebius-circline (moebius-inv M) H'
         assume ?M a ∈ circline-set H' ?M b ∈ circline-set H' ?M c ∈ circline-set H'
         hence a ∈ circline-set ?iH' ∧ b ∈ circline-set ?iH' ∧ c ∈ circline-set ?iH'
            using moebius-circline-set-mem[of M - ?iH', simplified]
            by simp
          hence H = ?iH'
            using ∗∗
            by simp
          thus H' = moebius-circline M H
            by simp
        qed
      qed
    qed
  next
    show ?P 0_h 1_h ∞_h
      using unique-circline-01inf'
      unfolding Ex1-def
      by (safe, rule-tac x=x-axis in exI) auto
  qed fact+
  thus ?thesis
    using assms
    by simp
qed
```

11.7 Circline set cardinality

11.7.1 Diagonal circlines

```
definition circline-diag-rep where
  circline-diag-rep H ⟷ mat-diagonal (Rep-circline-mat H)
lemma [simp]: mat-diagonal H ⟷ (let (A, B, C, D) = H in B = 0 ∧ C = 0)
by (cases H) simp

lift-definition circline-diag :: circline ⇒ bool is circline-diag-rep
by (auto simp add: circline-diag-rep-def)


lemma det-zero-trace-zero:
  assumes mat-det A = 0 mat-trace A = (0::complex) hermitean A
  shows A = mat-zero
using assms
proof−
  {
    fix a d c
    assume a ∗ d = cnj c ∗ c a + d = 0 cnj a = a
    from ‹a + d = 0› have d = −a
      by (metis add-eq-0-iff)
    hence − (cor (Re a))^2 = (cor (cmod c))^2
      using ‹cnj a = a› eq-cnj-iff-real[of a]
      using ‹a∗d = cnj c ∗ c›
      using complex-mult-cnj-cmod[of cnj c]
      by (simp add: complex-of-real-Re power2-eq-square)
    hence − (Re a)^2 ≥ 0
      using zero-le-power2[of cmod c]
      by (metis Re-complex-of-real cor-squared of-real-minus)
    hence a = 0
      using zero-le-power2[of Re a]
      using ‹cnj a = a› eq-cnj-iff-real[of a]
      by (cases a) simp
  } note ∗ = this
  obtain a b c d where A = (a, b, c, d)
    by (cases A) auto
  thus ?thesis
    using ∗[of a d c]  ∗[of d a c]
    using assms ‹A = (a, b, c, d)›
    by (auto simp add: hermitean-def mat-adj-def mat-cnj-def)
qed


lemma circline-diagonalize:
  shows ∃ M H'. moebius-circline M H = H' ∧ circline-diag H'
using assms
proof transfer
  fix H
  obtain A B C D where HH: Rep-circline-mat H = (A, B, C, D)
    by (cases Rep-circline-mat H) auto
  hence HH-elems: is-real A is-real D C = cnj B
    using hermitean-elems[of A B C D]  Rep-circline-mat[of H]
    by auto
 obtain M k1 k2 where ∗: mat-det M ≠ 0 unitary M congruence M (Rep-circline-mat
H) = (k1, 0, 0, k2) is-real k1 is-real k2
    using hermitean-diagonizable[of Rep-circline-mat H] Rep-circline-mat[of H]
    by auto
  have k1 ≠ 0 ∨ k2 ≠ 0
   using ‹congruence M (Rep-circline-mat H) = (k1, 0, 0, k2)› Rep-circline-mat[of
H] congruence-nonzero[of Rep-circline-mat H M] ‹mat-det M ≠ 0›
    by auto

  have ∗∗: Rep-circline-mat (Abs-circline-mat (k1, 0, 0, k2)) = (k1, 0, 0, k2)
    apply (rule Abs-circline-mat-inverse)
    using ‹is-real k1› ‹is-real k2› ‹k1 ≠ 0 ∨ k2 ≠ 0›
   by (auto simp add: hermitean-def mat-adj-def mat-cnj-def eq-cnj-iff-real[symmetric])

  thus ∃M H'. circline-mat-eq (moebius-circline-rep M H) H' ∧ circline-diag-rep
H'
    using ∗ mat-det-inv[of M]
   by (rule-tac x=Abs-moebius-mat (mat-inv M) in exI, rule-tac x=Abs-circline-mat
(k1, 0, 0, k2) in exI)
       (simp add: Abs-moebius-mat-inverse circline-diag-rep-def, rule-tac x=1 in
exI, simp)
qed

lemma wlog-circline-diag:
  assumes ⋀ H. circline-diag H ⟹ P H
         ⋀ M H. P H ⟹ P (moebius-circline M H)
  shows P H
proof−
  obtain M H' where moebius-circline M H = H' circline-diag H'
    using circline-diagonalize[of H]
    by auto
  hence P (moebius-circline M H)
    using assms(1)
    by simp
  thus ?thesis
    using assms(2)[of moebius-circline M H moebius-inv M]
    by simp
qed
```

11.7.2 Zero type circline set cardinality

```
lemma circline-type-zero-card-eq1-0h:
  assumes circline-type H = 0 0_h ∈ circline-set H
  shows circline-set H = {0_h}
using assms
unfolding circline-set-def
proof(safe)
  fix z
  assume on-circline H z circline-type H = 0 on-circline H 0_h
  hence H = circline-point-0h
    using unique-circline-type-zero-0h'
    unfolding circline-set-def
    by simp
  thus z = 0_h
    using ‹on-circline H z›
  proof transfer
    fix H z
    assume circline-mat-eq H circline-point-0h-rep on-circline-rep H z
    thus z ≈ zero-homo-rep
      using Rep-homo-coords[of z]
     by (cases Rep-homo-coords z, cases Rep-circline-mat H) (simp add: circline-point-0h-rep-def
on-circline-rep-def Let-def vec-cnj-def, rule-tac x=1/b in exI, auto)
  qed
qed

lemma bij-image-singleton:
  ⟦f ‘ A = {b}; f a = b; bij f⟧ ⟹ A = {a}
by (metis (mono-tags) bij-betw-imp-inj-on image-empty image-insert inj-vimage-image-eq)

lemma circline-type-zero-card-eq1:
  assumes circline-type H = 0
  shows ∃ z. circline-set H = {z}
proof−
  have ∃ z. on-circline H z
    using assms
  proof transfer
    fix H
    obtain A B C D where HH: Rep-circline-mat H = (A, B, C, D)
      by (cases Rep-circline-mat H) auto
    hence C = cnj B is-real A is-real D
      using Rep-circline-mat[of H] hermitean-elems[of A B C D]
      by auto
    assume circline-type-rep H = 0
    hence mat-det (Rep-circline-mat H) = 0
      using Rep-circline-mat[of H] mat-det-hermitean-real[of Rep-circline-mat H]
        by (auto simp add: circline-type-rep-def sgn-zero-iff) (metis complex-surj
complex-zero-def)
    hence A∗D = B∗C
      using HH
      by simp
    show Ex (on-circline-rep H)
    proof (cases A ≠ 0 ∨ B ≠ 0)
      case True
      thus ?thesis
        using HH ‹A∗D = B∗C›
      by (rule-tac x=Abs-homo-coords (−B, A) in exI) (auto simp add: on-circline-rep-def
Let-def Abs-homo-coords-inverse vec-cnj-def complex-cnj field-simps)
    next
      case False
      thus ?thesis
        using HH ‹C = cnj B›
      by (rule-tac x=Abs-homo-coords (1, 0) in exI) (simp add: Abs-homo-coords-inverse
on-circline-rep-def Let-def vec-cnj-def)
    qed
  qed
  then obtain z where on-circline H z
    by auto
  obtain M where moebius-pt M z = 0_h
    using ex-moebius-1[of z]
    by auto
  hence 0_h ∈ circline-set (moebius-circline M H)
    using ‹on-circline H z›
   by (subst moebius-circline-set[of M H, symmetric]) (force simp add: circline-set-def)
  hence circline-set (moebius-circline M H) = {0_h}
     using circline-type-zero-card-eq1-0h[of moebius-circline M H] ‹circline-type H
= 0›
    by (auto simp add: moebius-preserve-circline-type)
  hence circline-set H = {z}
    using ‹moebius-pt M z = 0_h›
    using bij-moebius-pt[of M] bij-image-singleton[of moebius-pt M circline-set H -
z]
    by (subst (asm) moebius-circline-set[symmetric]) simp
  thus ?thesis
    by auto
qed
```

11.7.3 Negative type circline set cardinality

```
lemma quad-form-diagonal-iff:
  assumes k1 ≠ 0 is-real k1 is-real k2 Re k1 ∗ Re k2 < 0
   shows quad-form (z1, 1) (k1, 0, 0, k2) = 0 ⟷ (∃ φ. z1 = rcis (sqrt (Re (−k2 /k1))) φ)
proof−
  have Re (−k2/k1) ≥ 0
    using ‹Re k1 ∗ Re k2 < 0› ‹is-real k1› ‹is-real k2› ‹k1 ≠ 0›
    by (auto simp add: Re-divide-real) (metis less-asym mult-neg-neg mult-pos-pos not-less zero-less-divide-iff)

  have quad-form (z1, 1) (k1, 0, 0, k2) = 0 ⟷ (cor (cmod z1))^2 = −k2 / k1
    using assms add-eq-0-iff[of k2 k1∗(cor (cmod z1))^2]
    using eq-divide-imp[of k1 (cor (cmod z1))^2 −k2]
    by (auto simp add: vec-cnj-def field-simps complex-mult-cnj-cmod)
  also have ... ⟷ (cmod z1)^2 = Re (−k2 /k1)
    using assms
    apply (subst complex-eq-if-Re-eq)
    using Re-complex-of-real[of (cmod z1)^2]
    by auto (metis is-real-complex-of-real of-real-power, metis div-reals)
  also have ... ⟷ cmod z1 = sqrt (Re (−k2 /k1))
    by (metis norm-ge-zero real-sqrt-ge-0-iff real-sqrt-pow2 real-sqrt-power)
  also have ... ⟷ (∃ φ. z1 = rcis (sqrt (Re (−k2 /k1))) φ)
   using rcis-cmod-arg[of z1, symmetric] assms abs-of-nonneg[of sqrt (Re (−k2/k1))]
    using ‹Re (−k2/k1) ≥ 0›
    by auto
  finally show ?thesis
    .
qed

lemma circline-type-neg-card-gt3-diag:
  assumes circline-type H < 0 circline-diag H
  shows ∃ A B C. A ≠ B ∧ A ≠ C ∧ B ≠ C ∧ {A, B, C} ⊆ circline-set H
using assms
unfolding circline-set-def
apply (simp del: HOL.ex-simps)
proof (transfer)
  fix H
  obtain A B C D where HH: Rep-circline-mat H = (A, B, C, D)
    by (cases Rep-circline-mat H) auto
  hence HH-elems: is-real A is-real D C = cnj B
    using hermitean-elems[of A B C D]  Rep-circline-mat[of H]
    by auto
  assume circline-diag-rep H circline-type-rep H < 0
  hence B = 0 C = 0 Re A ∗ Re D < 0 A ≠ 0
    using HH ‹is-real A› ‹is-real D›
    unfolding circline-diag-rep-def circline-type-rep-def
    by auto
  let ?x = sqrt (Re (− D / A))
  let ?A = (rcis ?x 0, 1)
  let ?B = (rcis ?x (pi/2), 1)
  let ?C = (rcis ?x pi, 1)
  from quad-form-diagonal-iff[OF ‹A ≠ 0› ‹is-real A› ‹is-real D› ‹Re A * Re D < 0›]
  have quad-form ?A (A, 0, 0, D) = 0  quad-form ?B (A, 0, 0, D) = 0  quad-form ?C (A, 0, 0, D) = 0
    by (auto simp del: rcis-zero-arg)
  moreover
  have Re (D / A) < 0
    using ‹Re A * Re D < 0› ‹A ≠ 0› ‹is-real A› ‹is-real D›
    by (subst Re-divide-real) (auto, metis divide-less-0-iff mult-eq-0-iff mult-neg-neg mult-pos-pos not-less-iff-gr-or-eq)
  hence ¬ Abs-homo-coords ?A ≈ Abs-homo-coords ?B ∧ ¬ Abs-homo-coords ?A ≈ Abs-homo-coords ?C ∧ ¬ Abs-homo-coords ?B ≈ Abs-homo-coords ?C
    unfolding rcis-def
    by (auto simp add: Abs-homo-coords-inverse cis-def)
  ultimately
  show ∃ A B C. ¬ A ≈ B ∧ ¬ A ≈ C ∧ ¬ B ≈ C ∧ (on-circline-rep H A ∧ on-circline-rep H B ∧ on-circline-rep H C)
    using HH ‹B = 0› ‹C = 0›
    by (rule-tac x=Abs-homo-coords ?A in exI, rule-tac x=Abs-homo-coords ?B in exI, rule-tac x=Abs-homo-coords ?C in exI)
       (simp add: on-circline-rep-def Abs-homo-coords-inverse Let-def)
qed

lemma circline-type-neg-card-gt3:
  assumes circline-type H < 0
  shows ∃ A B C. A ≠ B ∧ A ≠ C ∧ B ≠ C ∧ {A, B, C} ⊆ circline-set H
proof−
  obtain M H' where moebius-circline M H = H' circline-diag H'
    using circline-diagonalize[of H] assms
    by auto
  moreover
  hence circline-type H' < 0
    using assms moebius-preserve-circline-type
    by auto
  ultimately
  obtain A B C where A ≠ B A ≠ C B ≠ C {A, B, C} ⊆ circline-set H'
    using circline-type-neg-card-gt3-diag[of H']
    by auto
  let ?iM = moebius-inv M
  have moebius-circline ?iM H' = H
    using ‹moebius-circline M H = H'›[symmetric]
    by simp
  let ?A = moebius-pt ?iM A and ?B= moebius-pt ?iM B and ?C = moebius-pt ?iM C
  have ?A ∈ circline-set H  ?B ∈ circline-set H  ?C ∈ circline-set H
```

```
    using ‹moebius-circline ?iM H' = H›[symmetric] ‹{A, B, C} ⊆ circline-set H'›
    by (simp-all add: moebius-circline-set[symmetric])
  moreover
  have ?A ≠ ?B ?A ≠ ?C ?B ≠ ?C
    using bij-moebius-pt[of moebius-inv M] ‹A ≠ B› ‹A ≠ C› ‹B ≠ C›
    unfolding bij-def inj-on-def
    by blast+
  ultimately
  show ?thesis
    by auto
qed
```

11.7.4 Positive type circline set cardinality

```
lemma circline-type-pos-card-eq0-diag:
  assumes circline-diag H circline-type H > 0
  shows circline-set H = {}
using assms
unfolding circline-set-def
apply simp
proof transfer
  fix H
  obtain A B C D where HH: Rep-circline-mat H = (A, B, C, D)
    by (cases Rep-circline-mat H) auto
  hence HH-elems: is-real A is-real D C = cnj B
    using hermitean-elems[of A B C D] Rep-circline-mat[of H]
    by auto
  assume circline-diag-rep H 0 < circline-type-rep H
  hence B = 0 C = 0 Re A * Re D > 0 A ≠ 0
    using ‹circline-diag-rep H› HH ‹is-real A› ‹is-real D›
    unfolding circline-diag-rep-def circline-type-rep-def
    by auto
  show ∀ x. ¬ on-circline-rep H x
  proof
    fix x
    obtain x1 x2 where xx:Rep-homo-coords x = (x1, x2)
      by (rule obtain-homo-coords)
    have (Re A > 0 ∧ Re D > 0) ∨ (Re A < 0 ∧ Re D < 0)
      using ‹Re A * Re D > 0›
     by (metis linorder-neqE-linordered-idom mult-eq-0-iff zero-less-mult-pos zero-less-mult-pos2)
    moreover
    have (Re (x1 * cnj x1) ≥ 0 ∧ Re (x2 * cnj x2) > 0) ∨ (Re (x1 * cnj x1) > 0 ∧ Re (x2 * cnj x2) ≥ 0)
      using Rep-homo-coords[of x] xx
      by auto (metis complex-surj complex-zero-def sum-squares-gt-zero-iff)+
    ultimately
    have Re A * Re (x1 * cnj x1) + Re D * Re (x2 * cnj x2) ≠ 0
      apply auto
     apply (metis add-less-cancel-left add-pos-pos mult-eq-0-iff mult-pos-pos sum-squares-eq-zero-iff
sum-squares-gt-zero-iff)
    apply (metis (lifting, no-types) add-pos-pos comm-semiring-1-class.normalizing-semiring-rules(6)
mult-eq-0-iff mult-pos-pos sum-squares-gt-zero-iff)
         apply (metis add-less-cancel-left add-neg-neg mult-eq-0-iff mult-pos-neg2
sum-squares-eq-zero-iff sum-squares-gt-zero-iff)
    apply (metis (lifting, no-types) add-neg-neg comm-semiring-1-class.normalizing-semiring-rules(6)
mult-eq-0-iff mult-pos-neg2 sum-squares-gt-zero-iff)
      done
    hence A * (x1 * cnj x1) + D * (x2 * cnj x2) ≠ 0
      using ‹is-real A› ‹is-real D›
      by (metis Re-mult-real complex-Re-add complex-Re-zero)
    thus ¬ on-circline-rep H x
      using HH ‹B = 0› ‹C = 0› xx
      unfolding on-circline-rep-def Let-def
      by (simp add: vec-cnj-def field-simps)
  qed
qed

lemma circline-type-pos-card-eq0:
  assumes circline-type H > 0
  shows circline-set H = {}
proof-
  obtain M H' where moebius-circline M H = H' circline-diag H'
    using circline-diagonalize[of H] assms
    by auto
  moreover
  hence circline-type H' > 0
    using assms moebius-preserve-circline-type
    by auto
  ultimately
  have circline-set H' = {}
    using circline-type-pos-card-eq0-diag[of H']
    by auto
  let ?iM = moebius-inv M
  have moebius-circline ?iM H' = H
    using ‹moebius-circline M H = H'›[symmetric]
    by simp
  thus ?thesis
    using ‹circline-set H' = {}›
    by (auto simp add: moebius-circline-set[symmetric])
qed
```

11.7.5 Cardinality determines type

```
lemma card-eq1-circline-type-zero:
  assumes ∃ z. circline-set H = {z}
  shows circline-type H = 0
proof (cases circline-type H < 0)
  case True
  thus ?thesis
    using circline-type-neg-card-gt3[of H] assms
    by auto
next
  case False
  show ?thesis
  proof (cases circline-type H > 0)
    case True
    thus ?thesis
      using circline-type-pos-card-eq0[of H] assms
      by auto
  next
    case False
    thus ?thesis
      using ‹¬ (circline-type H) < 0›
      by simp
  qed
qed
```

11.7.6 Circline set is injective

```
lemma inj-circline-set:
  assumes circline-set H = circline-set H' circline-set H ≠ {}
  shows H = H'
proof (cases circline-type H < 0)
  case True
  then obtain A B C where A ≠ B A ≠ C B ≠ C {A, B, C} ⊆ circline-set H
    using circline-type-neg-card-gt3[of H]
    by auto
  hence ∃!H. A ∈ circline-set H ∧ B ∈ circline-set H ∧ C ∈ circline-set H
    using unique-circline-set[of A B C]
    by simp
  thus ?thesis
    using ‹circline-set H = circline-set H'› ‹{A, B, C} ⊆ circline-set H›
    by auto
next
  case False
  show ?thesis
  proof (cases circline-type H = 0)
    case True
    moreover
    then obtain A where {A} = circline-set H
      using circline-type-zero-card-eq1[of H]
      by auto
    moreover
    hence circline-type H' = 0
      using ‹circline-set H = circline-set H'› card-eq1-circline-type-zero[of H']
      by auto
    ultimately
    show ?thesis
      using unique-circline-type-zero[of A] ‹circline-set H = circline-set H'›
      by auto
  next
    case False
    hence circline-type H > 0
      using ‹¬ (circline-type H < 0)›
      by auto
    thus ?thesis
      using ‹circline-set H ≠ {}›  circline-type-pos-card-eq0[of H]
      by auto
  qed
qed
```

11.8 Symmetric points wrt. circline

```
definition circline-symmetric-rep where
  circline-symmetric-rep z1 z2 H ⟷
    (let z1 = Rep-homo-coords z1;
         z2 = Rep-homo-coords z2;
          H = Rep-circline-mat H in
       bilinear-form z1 z2 H = 0)

lift-definition circline-symmetric :: complex-homo ⇒ complex-homo ⇒ circline
⇒ bool is circline-symmetric-rep
by (auto simp add: circline-symmetric-rep-def bilinear-form-scale-m bilinear-form-scale-v1
bilinear-form-scale-v2 simp del: vec-cnj-sv quad-form-def bilinear-form-def)

lemma symmetry-principle:
  assumes circline-symmetric z1 z2 H
  shows circline-symmetric (moebius-pt M z1) (moebius-pt M z2) (moebius-circline
M H)
using assms
proof (transfer)
  fix z1 z2 H M
  assume circline-symmetric-rep z1 z2 H
  thus circline-symmetric-rep (moebius-pt-rep M z1) (moebius-pt-rep M z2) (moebius-circline-rep
M H)
    apply (auto simp add: circline-symmetric-rep-def simp del: bilinear-form-def)
    using Rep-moebius-mat[of M]
    by (subst bilinear-form-congruence[symmetric]) simp-all
qed
```

Symmetry wrt. *unit-circle*

```
lemma circline-symmetric-0inf-disc: circline-symmetric 0_h ∞_h unit-circle
by (transfer) (simp add: circline-symmetric-rep-def vec-cnj-def)

lemma circline-symmetric-inv-homo-disc: circline-symmetric a (inversion-homo
a) unit-circle
```

```
unfolding inversion-homo-def
by (transfer) (case-tac Rep-homo-coords a, auto simp add: circline-symmetric-rep-def vec-cnj-def split-def Let-def)

lemma circline-symmetric-inv-homo-disc':
  assumes circline-symmetric a a' unit-circle
  shows a' = inversion-homo a
unfolding inversion-homo-def
using assms
proof (transfer)
  fix a a'
  obtain a1 a2 where aa: Rep-homo-coords a = (a1, a2)
    by (rule obtain-homo-coords)
  obtain a1' a2' where aa': Rep-homo-coords a' = (a1', a2')
    by (rule obtain-homo-coords)
  assume *: circline-symmetric-rep a a' unit-circle-rep
  show a' ≈ (cnj-homo-coords ∘ reciprocal-homo-coords) a
  proof (cases a1' = 0)
    case True
    thus ?thesis
      using aa aa' * Rep-homo-coords[of a'] Rep-homo-coords[of a]
      by (auto simp add: circline-symmetric-rep-def vec-cnj-def field-simps)
  next
    case False
    show ?thesis
    proof (cases a2 = 0)
      case True
      thus ?thesis
        using ‹a1' ≠ 0›
        using aa aa' * Rep-homo-coords[of a]
        by (simp add: circline-symmetric-rep-def vec-cnj-def field-simps)
    next
      case False
      thus ?thesis
        using ‹a1' ≠ 0› aa aa' *
         by (simp add: circline-symmetric-rep-def vec-cnj-def field-simps) (rule-tac x=cnj a2 / a1' in exI, simp add: field-simps)
    qed
  qed
qed
```

11.9 Oriented circlines; discs

```
definition ocircline-mat-eq where
  [simp]: ocircline-mat-eq A B ⟷ (∃ k::real. k > 0 ∧ Rep-circline-mat B = complex-of-real k *sm (Rep-circline-mat A))

lemma [simp]: ocircline-mat-eq H H
by (simp, rule-tac x=1 in exI, simp)
```

quotient-type *ocircline* = *circline-mat* / *ocircline-mat-eq*
proof (*rule equivpI*)
  show *reflp ocircline-mat-eq*
    unfolding *reflp-def*
    by (*auto*, *rule-tac x=1* **in** *exI*, *simp*)
next
  show *symp ocircline-mat-eq*
    unfolding *symp-def*
    by (*auto*, *rule-tac x=1/k* **in** *exI*, *simp*)
next
  show *transp ocircline-mat-eq*
    unfolding *transp-def*
    by (*auto*, *rule-tac x=ka∗k* **in** *exI*, *simp add*: *mult-pos-pos*)
qed

lift-definition *on-ocircline* :: *ocircline* ⇒ *complex-homo* ⇒ *bool* **is** *on-circline-rep*
by (*auto simp add*: *on-circline-rep-def quad-form-scale-m quad-form-scale-v Let-def simp del*: *vec-cnj-sv quad-form-def*)

definition *ocircline-set* :: *ocircline* ⇒ *complex-homo set* **where**
  ocircline-set H = {*z*. *on-ocircline H z*}

disc and disc complement

definition *in-ocircline-rep* **where**
  in-ocircline-rep H z ⟷
          (*let z* = *Rep-homo-coords z*;
              H = *Rep-circline-mat H*
            in Re (*quad-form z H*) < *0*)

lift-definition *in-ocircline* :: *ocircline* ⇒ *complex-homo* ⇒ *bool* **is** *in-ocircline-rep*
proof−
  fix *H H′ z z′*
  assume *ocircline-mat-eq H H′ z* ≈ *z′*
  then obtain *k k′* **where**
   ∗: *0* < *k Rep-circline-mat H′* = *cor k* $*_{sm}$ *Rep-circline-mat H k′* ≠ *0 Rep-homo-coords z′* = *k′* $*_{sv}$ *Rep-homo-coords z*
    by *auto*
   hence *quad-form* (*Rep-homo-coords z′*) (*Rep-circline-mat H′*) = *cor k* ∗ *cor* ((*cmod k′*)^2) ∗ *quad-form* (*Rep-homo-coords z*) (*Rep-circline-mat H*)
   by (*simp add*: *quad-form-scale-v quad-form-scale-m del*: *vec-cnj-sv quad-form-def*)
  hence *Re* (*quad-form* (*Rep-homo-coords z′*) (*Rep-circline-mat H′*)) =
    k ∗ (*cmod k′*)^2 ∗ *Re* (*quad-form* (*Rep-homo-coords z*) (*Rep-circline-mat H*))
    using *Rep-circline-mat*[*of H*] *quad-form-hermitean-real*[*of Rep-circline-mat H*]
    by (*simp add*: *complex-of-real-Re power2-eq-square*)
  thus *in-ocircline-rep H z* = *in-ocircline-rep H′ z′*
    unfolding *in-ocircline-rep-def Let-def*
    using ‹*k* > *0*› ‹*k′* ≠ *0*›
    apply *auto*

apply (*metis mult-pos-neg mult-pos-pos norm-eq-zero zero-less-power2*)
apply (*metis comm-semiring-1-class.normalizing-semiring-rules*(*10*) *mult-less-cancel-left-pos zero-less-norm-iff zero-less-power*)
done
qed

definition *disc* **where**
disc H = {*z. in-ocircline H z*}

definition *out-ocircline-rep* **where**
out-ocircline-rep H z ⟷
(*let z* = *Rep-homo-coords z*;
H = *Rep-circline-mat H*
in Re (*quad-form z H*) > *0*)

lift-definition *out-ocircline* :: *ocircline* ⇒ *complex-homo* ⇒ *bool* **is** *out-ocircline-rep*
proof−
fix *H H′ z z′*
assume *ocircline-mat-eq H H′ z* ≈ *z′*
then obtain *k k′* **where**
∗: *0* < *k Rep-circline-mat H′* = *cor k* $*_{sm}$ *Rep-circline-mat H k′* ≠ *0 Rep-homo-coords z′* = *k′* $*_{sv}$ *Rep-homo-coords z*
by *auto*
hence *quad-form* (*Rep-homo-coords z′*) (*Rep-circline-mat H′*) = *cor k* ∗ *cor* ((*cmod k′*)^2) ∗ *quad-form* (*Rep-homo-coords z*) (*Rep-circline-mat H*)
by (*simp add*: *quad-form-scale-v quad-form-scale-m del*: *vec-cnj-sv quad-form-def*)
hence *Re* (*quad-form* (*Rep-homo-coords z′*) (*Rep-circline-mat H′*)) =
k ∗ (*cmod k′*)^2 ∗ *Re* (*quad-form* (*Rep-homo-coords z*) (*Rep-circline-mat H*))
using *Rep-circline-mat*[*of H*] *quad-form-hermitean-real*[*of Rep-circline-mat H*]
by (*simp add*: *complex-of-real-Re power2-eq-square*)
thus *out-ocircline-rep H z* = *out-ocircline-rep H′ z′*
unfolding *out-ocircline-rep-def Let-def*
using ‹*k* > *0*› ‹*k′* ≠ *0*›
apply *auto*
apply (*metis mult-pos-pos norm-eq-zero zero-less-power2*)
apply (*metis comm-semiring-1-class.normalizing-semiring-rules*(*10*) *mult-less-cancel-left-pos zero-less-norm-iff zero-less-power*)
done
qed

definition *disc-compl* **where**
disc-compl H = {*z. out-ocircline H z*}

lemma *in-on-out*: *in-ocircline H z* ∨ *on-ocircline H z* ∨ *out-ocircline H z*
proof *transfer*
fix *z H*
show *in-ocircline-rep H z* ∨ *on-circline-rep H z* ∨ *out-ocircline-rep H z*
using *Rep-circline-mat*[*of H*] *quad-form-hermitean-real*[*of Rep-circline-mat H Rep-homo-coords z*]

by (*simp add*: *in-ocircline-rep-def on-circline-rep-def out-ocircline-rep-def Let-def*) (*metis complex-Im-zero complex-Re-zero complex-equality linorder-cases*)
qed

lemma *disc H* ∪ *disc-compl H* ∪ *ocircline-set H* = *UNIV*
unfolding *disc-def disc-compl-def ocircline-set-def*
using *in-on-out*[*of H*]
by *auto*

lemma
disc-inter-disc-compl: *disc H* ∩ *disc-compl H* = {}
unfolding *disc-def disc-compl-def*
by *auto* (*transfer*, *auto simp add*: *in-ocircline-rep-def out-ocircline-rep-def Let-def*)

lemma
disc-inter-ocircline-set: *disc H* ∩ *ocircline-set H* = {}
unfolding *disc-def ocircline-set-def*
by *auto* (*transfer*, *simp add*: *in-ocircline-rep-def on-circline-rep-def Let-def*)

lemma
disc-compl-inter-ocircline-set: *disc-compl H* ∩ *ocircline-set H* = {}
unfolding *disc-compl-def ocircline-set-def*
by *auto* (*transfer*, *simp add*: *out-ocircline-rep-def on-circline-rep-def Let-def*)

Opposite orientation

definition *opposite-ocircline-rep* **where**
opposite-ocircline-rep H =
(*let H* = *Rep-circline-mat H in*
Abs-circline-mat ($-1 *_{sm}$ *H*))

lemma *circline-mat-mult-m1* [*simp*]: *Rep-circline-mat* (*Abs-circline-mat* ($-1 *_{sm}$ *Rep-circline-mat H*)) = ($-1 *_{sm}$ *Rep-circline-mat H*)
proof−
have −*1* = *cor* (−*1*)
by (*simp add*: *complex-of-real-def*)
thus *?thesis*
using *circline-mat-mult-sm-Rep*[*of* −*1 H*]
by *auto*
qed

lemma [*simp*]: *Rep-circline-mat* (*opposite-ocircline-rep H*) = ($-1 *_{sm}$ *Rep-circline-mat H*)
unfolding *opposite-ocircline-rep-def*
by *auto*

lift-definition *opposite-ocircline* :: *ocircline* ⇒ *ocircline* **is** *opposite-ocircline-rep*
by *auto*

lemma *opposite-ocircline-rep-opposite-ocircline-rep*

[*simp*]: *opposite-ocircline-rep* (*opposite-ocircline-rep H*) = *H*
by (*simp add*: *opposite-ocircline-rep-def Rep-circline-mat-inverse*)

lemma *opposite-ocircline-opposite-ocircline*
[*simp*]: *opposite-ocircline* (*opposite-ocircline H*) = *H*
by (*transfer*) (*auto*, *rule-tac x=1* **in** *exI*, *simp*)

lemma *ocircline-set-opposite-ocircline*
[*simp*]: *ocircline-set* (*opposite-ocircline H*) = *ocircline-set H*
unfolding *ocircline-set-def*
by *auto* (*transfer*, *auto simp add*: *on-circline-rep-def quad-form-scale-m simp del*: *quad-form-def*)+

lemma *disc-compl-opposite*: *disc-compl* (*opposite-ocircline H*) = *disc H*
unfolding *disc-def disc-compl-def*
apply *auto*
apply (*transfer*)
apply (*auto simp add*: *in-ocircline-rep-def out-ocircline-rep-def quad-form-scale-m simp del*: *quad-form-def*)
apply (*transfer*)
apply (*auto simp add*: *in-ocircline-rep-def out-ocircline-rep-def quad-form-scale-m simp del*: *quad-form-def*)
done

lemma *disc-opposite*:
disc (*opposite-ocircline H*) = *disc-compl H*
using *disc-compl-opposite*[*of opposite-ocircline H*]
by *simp*

of-ocircline, *pos-oriented*, *of-circline*

lift-definition *of-ocircline* :: *ocircline* ⇒ *circline* **is** *id*::*circline-mat* ⇒ *circline-mat*
by *auto* (*rule-tac x=k* **in** *exI*, *simp*)

lemma *of-ocircline-opposite-ocircline* [*simp*]:
of-ocircline (*opposite-ocircline H*) = *of-ocircline H*
by (*transfer*) (*auto*, *rule-tac x=−1* **in** *exI*, *simp*)

lemma *circline-set-ocircline-set* [*simp*]:
circline-set (*of-ocircline H*) = *ocircline-set H*
unfolding *ocircline-set-def circline-set-def*
by (*safe*) (*transfer*, *simp*)+

lemma *inj-of-ocircline*:
assumes *of-ocircline H* = *of-ocircline H'*
shows *H* = *H'* ∨ *H* = *opposite-ocircline H'*
using *assms*
by (*transfer*) (*auto*, *metis linorder-neqE-linordered-idom neg-0-less-iff-less of-real-minus*)

```
lemma
  inj-ocircline-set:
  assumes ocircline-set H = ocircline-set H' ocircline-set H ≠ {}
  shows H = H' ∨ H = opposite-ocircline H'
proof−
   from assms have circline-set (of-ocircline H) = circline-set (of-ocircline H')
circline-set (of-ocircline H') ≠ {}
   using circline-set-ocircline-set[symmetric, of H] circline-set-ocircline-set[symmetric,
of H']
    by blast+
  hence of-ocircline H = of-ocircline H'
    by (simp add: inj-circline-set)
  thus ?thesis
    by (rule inj-of-ocircline)
qed
```

```
definition pos-oriented-rep where
 pos-oriented-rep H ⟷
   (let (A, B, C, D) = Rep-circline-mat H
     in (Re A > 0 ∨ (Re A = 0 ∧ ((B ≠ 0 ∧ arg B > 0) ∨ (B = 0 ∧ Re D >
0)))))

lemma pos-oriented-rep: pos-oriented-rep H ∨ pos-oriented-rep (opposite-ocircline-rep
H)
proof−
  obtain A B C D where HH: Rep-circline-mat H = (A, B, C, D)
    by (cases Rep-circline-mat H) auto
  moreover
  hence Re A = 0 ∧ Re D = 0 ⟶ B ≠ 0
    using Rep-circline-mat[of H] hermitean-elems[of A B C D]
    by (cases A, cases D) auto
  moreover
  have B ≠ 0 ∧ ¬ 0 < arg B ⟶ 0 < arg (− B)
    using MoreComplex.canon-ang-plus-pi2[of arg B] arg-bounded[of B]
    by (auto simp add: arg-uminus)
  ultimately
  show ?thesis
    by (auto simp add: pos-oriented-rep-def)
qed

lift-definition pos-oriented :: ocircline ⇒ bool is pos-oriented-rep
apply (auto simp add: pos-oriented-rep-def mult-pos-pos)
apply (metis arg-mult-real-positive)
apply (metis arg-mult-real-positive)
apply (metis zero-less-mult-pos)+
apply (metis arg-mult-real-positive)
apply (metis arg-mult-real-positive)
apply (metis zero-less-mult-pos)+
done

lemma pos-oriented: pos-oriented H ∨ pos-oriented (opposite-ocircline H)
by (transfer) (rule pos-oriented-rep)

lemma pos-oriented-opposite-ocircline:
  pos-oriented (opposite-ocircline H) ⟷ ¬ pos-oriented H
proof transfer
  fix H
  obtain A B C D where HH: Rep-circline-mat H = (A, B, C, D)
    by (cases Rep-circline-mat H) auto
  moreover
  hence Re A = 0 ∧ Re D = 0 ⟶ B ≠ 0
    using Rep-circline-mat[of H] hermitean-elems[of A B C D]
    by (cases A, cases D) auto
  moreover
  have B ≠ 0 ∧ ¬ 0 < arg B ⟶ 0 < arg (− B)
    using MoreComplex.canon-ang-plus-pi2[of arg B] arg-bounded[of B]
    by (auto simp add: arg-uminus)
  moreover
  have B ≠ 0 ∧ 0 < arg B ⟶ ¬ 0 < arg (− B)
    using MoreComplex.canon-ang-plus-pi1[of arg B] arg-bounded[of B]
    by (auto simp add: arg-uminus)
  ultimately
  show pos-oriented-rep (opposite-ocircline-rep H) = (¬ pos-oriented-rep H)
    unfolding pos-oriented-rep-def
    by simp (metis not-less-iff-gr-or-eq)
qed

lemma pos-oriented-circle-inf:
  assumes ∞h ∉ ocircline-set H
  shows pos-oriented H ⟷ ∞h ∉ disc H
using assms
unfolding ocircline-set-def disc-def
apply simp
proof transfer
  fix H
  obtain A B C D where HH: Rep-circline-mat H = (A, B, C, D)
    by (cases Rep-circline-mat H) auto
  hence is-real A
    using Rep-circline-mat[of H] hermitean-elems
    by auto
  assume ¬ on-circline-rep H inf-homo-rep
  thus pos-oriented-rep H = (¬ in-ocircline-rep H inf-homo-rep)
    using HH ‹is-real A›
     by (cases A) (auto simp add: on-circline-rep-def in-ocircline-rep-def Let-def
pos-oriented-rep-def vec-cnj-def)
qed
```

```
lemma
   assumes is-circle (of-ocircline H) (a, r) = euclidean-circle (of-ocircline H)
circline-type (of-ocircline H) < 0
  shows pos-oriented H ⟷ of-complex a ∈ disc H
using assms
unfolding disc-def
apply simp
proof transfer
  fix H a r
  obtain A B C D where HH: Rep-circline-mat H = (A, B, C, D)
    by (cases Rep-circline-mat H) auto
  hence is-real A is-real D C = cnj B
    using Rep-circline-mat[of H] hermitean-elems
    by auto

 assume ∗: ¬ circline-A0-rep (id H) (a, r) = euclidean-circle-rep (id H) circline-type-rep
(id H) < 0
  hence A ≠ 0 Re A ≠ 0
    using HH ‹is-real A›
    by (case-tac[!] A) (auto simp add: circline-A0-rep-def)

  have Re (A∗D − B∗C) < 0
    using ‹circline-type-rep (id H) < 0› HH
    by (simp add: circline-type-rep-def)

  have (A ∗ (D ∗ cnj A) − B ∗ (C ∗ cnj A)) / (A ∗ cnj A) = (A∗D − B∗C) / A
    using ‹A ≠ 0›
    by (simp add: field-simps)
  hence 0 < Re A ⟷ Re ((A ∗ (D ∗ cnj A) − B ∗ (C ∗ cnj A)) / (A ∗ cnj A))
< 0
    using ‹is-real A› ‹A ≠ 0› ‹Re (A∗D − B∗C) < 0›
   by (auto simp add: Re-divide-real, metis divide-less-0-iff less-iff-diff-less-0, metis
divide-less-0-iff less-iff-diff-less-0 mult-neg-neg zero-less-mult-pos)
  thus pos-oriented-rep H = in-ocircline-rep H (of-complex-coords a)
    using HH ‹Re A ≠ 0› ∗ ‹is-real A›
     by (simp add: circline-A0-rep-def euclidean-circle-rep-def pos-oriented-rep-def
in-ocircline-rep-def Let-def vec-cnj-def complex-cnj field-simps)
qed

definition of-circline-rep :: circline-mat ⇒ circline-mat where
 of-circline-rep H = (if pos-oriented-rep H then H else opposite-ocircline-rep H)

lift-definition of-circline :: circline ⇒ ocircline is of-circline-rep
proof−
  fix H H'
  assume circline-mat-eq H H'
 then obtain k where ∗: k ≠ 0 Rep-circline-mat H' = cor k ∗sm Rep-circline-mat
H
```

```
    by auto
  obtain A B C D where HH: Rep-circline-mat H = (A, B, C, D)
    by (cases Rep-circline-mat H) auto
  obtain A′ B′ C′ D′ where HH′: Rep-circline-mat H′ = (A′, B′, C′, D′)
    by (cases Rep-circline-mat H′) auto

  show ocircline-mat-eq (of-circline-rep H) (of-circline-rep H′)
  proof (cases Re A > 0)
    case True
    show ?thesis
    proof (cases k > 0)
      case True
      hence Re A′ > 0
        using ‹Re A > 0› ∗ HH HH′
        by (auto simp add: mult-pos-pos)
      thus ?thesis
        using ‹Re A > 0›
        using ∗ ‹k > 0› HH HH′
        by (auto simp add: pos-oriented-rep-def of-circline-rep-def Let-def split-def)
    next
      case False
      hence k < 0
        using ‹k ≠ 0›
        by auto
      hence Re A′ < 0
        using ‹Re A > 0› ∗ HH HH′
        by (auto simp add: mult-neg-pos)
      thus ?thesis
        using ‹Re A > 0›
        using ∗ ‹k < 0› HH HH′
        using circline-mat-mult-sm-Rep[of −k H]
        by (auto simp add: pos-oriented-rep-def of-circline-rep-def Let-def split-def)
(rule-tac x=−k in exI, simp)
    qed
  next
    case False
    show ?thesis
    proof (cases Re A < 0)
      case True
      show ?thesis
      proof (cases k > 0)
        case True
        hence Re A′ < 0
          using ‹Re A < 0›
          using ∗ HH HH′
          by (auto simp add: mult-pos-neg)
        moreover
        have −1 = cor (−1)
          by (simp add: complex-of-real-def)
        ultimately
        show ?thesis
          using ‹Re A < 0›
          using ∗ ‹k > 0› HH HH′
          using circline-mat-mult-sm-Rep[of −k H]
          using circline-mat-mult-sm-Rep[of −1 H]
         by (auto simp add: pos-oriented-rep-def of-circline-rep-def Let-def split-def)
      next
        case False
        hence k < 0
          using ‹k ≠ 0›
          by simp
        hence Re A′ > 0
          using ‹Re A < 0›
          using ∗ HH HH′
          by (auto simp add: mult-neg-neg)
        moreover
        have −1 = cor (−1)
          by (simp add: complex-of-real-def)
        ultimately
        show ?thesis
          using ‹Re A < 0›
          using ∗ ‹k < 0› HH HH′
          using circline-mat-mult-sm-Rep[of −k H]
          using circline-mat-mult-sm-Rep[of −1 H]
         by (auto simp add: pos-oriented-rep-def of-circline-rep-def Let-def split-def)
(rule-tac x=−k in exI, simp)
      qed
    next
      case False
      hence Re A = 0
        using ‹¬ Re A > 0›
        by auto
      hence Re A′ = 0
        using ∗ HH HH′
        by auto

      show ?thesis
      proof (cases B ≠ 0)
        case True
        show ?thesis
        proof (cases arg B > 0)
          case True
          show ?thesis
          proof (cases arg B′ > 0)
            case True
            hence k > 0
              using ‹arg B > 0›
              using ∗ HH HH′ arg-mult[of cor k B] ‹B ≠ 0› ‹k ≠ 0›
```

```
           using arg-complex-of-real-negative[of k] arg-complex-of-real-positive[of
k]
          using MoreComplex.canon-ang-plus-pi1[of arg B] arg-bounded[of B]
          by (cases k > 0) (auto simp add: arg-mult field-simps)
        thus ?thesis
          using ‹arg B > 0› ‹arg B′ > 0› ‹B ≠ 0› ‹Re A = 0› ‹Re A′ = 0› HH
HH′ ∗
          by (auto simp add: pos-oriented-rep-def of-circline-rep-def)
      next
        case False
        hence k < 0
          using ‹arg B > 0›
          using ∗ HH HH′ arg-mult[of cor k B] ‹B ≠ 0› ‹k ≠ 0›
           using arg-complex-of-real-negative[of k] arg-complex-of-real-positive[of
k]
          by (cases k > 0) (auto simp add: arg-mult field-simps canon-ang-arg)
        thus ?thesis
         using ‹arg B > 0› ‹¬ arg B′ > 0› ‹Re A = 0› ‹Re A′ = 0› ‹B ≠ 0› HH
HH′ ∗
          using circline-mat-mult-sm-Rep[of −k H]
            by (auto simp add: pos-oriented-rep-def of-circline-rep-def) (rule-tac
x=−k in exI, simp)+
      qed
    next
      case False
      show ?thesis
      proof (cases arg B′ > 0)
        case True
        hence k < 0
          using ‹¬ arg B > 0›
          using ∗ HH HH′ arg-mult[of cor k B] ‹B ≠ 0› ‹k ≠ 0›
           using arg-complex-of-real-negative[of k] arg-complex-of-real-positive[of
k]
          by (cases k > 0) (auto simp add: arg-mult field-simps canon-ang-arg)
        thus ?thesis
         using ‹¬ arg B > 0› ‹arg B′ > 0› ‹B ≠ 0› ‹Re A = 0› ‹Re A′ = 0› HH
HH′ ∗
            by (auto simp add: pos-oriented-rep-def of-circline-rep-def) (rule-tac
x=−k in exI, simp)+
      next
        case False
        hence k > 0
          using ‹¬ arg B > 0›
          using ∗ HH HH′ arg-mult[of cor k B] ‹B ≠ 0› ‹k ≠ 0›
           using arg-complex-of-real-negative[of k] arg-complex-of-real-positive[of
k]
          using MoreComplex.canon-ang-plus-pi2[of arg B] arg-bounded[of B]
          by (cases k > 0) (auto simp add: arg-mult field-simps canon-ang-arg)
        thus ?thesis
```

```
              using ⟨¬ arg B > 0⟩ ⟨¬ arg B' > 0⟩ ⟨Re A = 0⟩ ⟨Re A' = 0⟩ ⟨B ≠ 0⟩ HH HH' *
              using circline-mat-mult-sm-Rep[of −k H]
              by (auto simp add: pos-oriented-rep-def of-circline-rep-def)
          qed
        qed
      next
        case False
        hence B' = 0
          using * HH HH'
          by simp
        have Re D ≠ 0
          using ⟨Re A = 0⟩ ⟨¬ B ≠ 0⟩
          using Rep-circline-mat[of H] HH hermitean-elems[of A B C D]
          by (cases A, cases D) auto
        show ?thesis
         using ⟨¬ B ≠ 0⟩ ⟨B' = 0⟩ ⟨Re A = 0⟩ ⟨Re A' = 0⟩ ⟨Re D ≠ 0⟩ HH HH' *
          apply (auto simp add: of-circline-rep-def pos-oriented-rep-def)
          apply (metis zero-less-mult-pos2)
          apply (rule-tac x=−k in exI, simp, metis linorder-cases mult-pos-pos)
        apply (rule-tac x=−k in exI, simp, metis linorder-cases zero-less-mult-pos)
          apply (rule-tac x=k in exI, simp, metis mult-neg-neg neqE)
          done
      qed
    qed
  qed
qed

lemma pos-oriented-of-circline: pos-oriented (of-circline H)
proof (transfer)
  fix H
  show pos-oriented-rep (of-circline-rep H)
    using pos-oriented-rep[of H]
    unfolding of-circline-rep-def
    by auto
qed

lemma of-ocircline-of-circline [simp]: of-ocircline (of-circline H) = H
apply (transfer)
apply (auto simp add: of-circline-rep-def)
by (rule-tac x=1 in exI, simp) (rule-tac x=−1 in exI, auto simp add: pos-oriented-rep-def complex-of-real-def)

lemma of-circline-of-ocircline-pos-oriented [simp]:
  assumes pos-oriented H
  shows of-circline (of-ocircline H) = H
using assms
by (transfer) (simp add: of-circline-rep-def, rule-tac x=1 in exI, simp)
```

```
lemma ocircline-set-circline-set[simp]: ocircline-set (of-circline H) = circline-set
H
  unfolding ocircline-set-def circline-set-def
proof (safe)
  fix z
  assume on-ocircline (of-circline H) z
  thus on-circline H z
     by (transfer) (auto simp add: on-circline-rep-def of-circline-rep-def Let-def
quad-form-scale-m simp del: quad-form-def split: split-if-asm)
next
  fix z
  assume on-circline H z
  thus on-ocircline (of-circline H) z
     by (transfer) (auto simp add: on-circline-rep-def of-circline-rep-def Let-def
quad-form-scale-m simp del: quad-form-def)
qed

lemma inj-of-circline:
  assumes of-circline H = of-circline H'
  shows H = H'
using assms
proof (transfer)
  fix H H'
  assume ocircline-mat-eq (of-circline-rep H) (of-circline-rep H')
  then obtain k where k > 0 Rep-circline-mat (of-circline-rep H') = cor k *_sm
Rep-circline-mat (of-circline-rep H)
    by auto
  thus circline-mat-eq H H'
    using mult-sm-inv-l[of -1 Rep-circline-mat H' cor k *_sm Rep-circline-mat H]
   using mult-sm-inv-l[of -1 Rep-circline-mat H' (- (cor k)) *_sm Rep-circline-mat
H]
    apply (auto simp add: of-circline-rep-def split: split-if-asm)
    apply (rule-tac x=k in exI, simp)
    apply (rule-tac x=-k in exI, simp)
    apply (rule-tac x=-k in exI, simp)
    apply (rule-tac x=k in exI, simp)
    done
qed

lemma of-circline-of-ocircline:
 shows of-circline (of-ocircline H') = H' ∨ of-circline (of-ocircline H') = opposite-ocircline
H'
proof (cases pos-oriented H')
  case True
  thus ?thesis
    by auto
next
  case False
  hence pos-oriented (opposite-ocircline H')
```

using *pos-oriented*
by *auto*
thus *?thesis*
using *of-ocircline-opposite-ocircline*[*of H'*]
using *of-circline-of-ocircline-pos-oriented* [*of opposite-ocircline H'*]
by *auto*
qed

11.10 Some special oriented circlines and discs

lift-definition *mk-ocircline* :: *complex* ⇒ *complex* ⇒ *complex* ⇒ *complex* ⇒ *ocircline* **is** *mk-circline-rep*
by (*simp add*: *mk-circline-rep-def*, *rule-tac x=1* **in** *exI*, *simp*)

oriented unit circle and unit disc

lift-definition *ounit-circle* :: *ocircline* **is** *unit-circle-rep*
done

definition *unit-disc* = *disc ounit-circle*

lemma *zero-in-unit-disc*: $0_h \in$ *unit-disc*
unfolding *unit-disc-def disc-def*
by (*simp*, *transfer*) (*simp add*: *in-ocircline-rep-def Let-def vec-cnj-def*)

lemma *inf-notin-unit-disc*: $\infty_h \notin$ *unit-disc*
unfolding *unit-disc-def disc-def*
by (*simp*, *transfer*) (*simp add*: *in-ocircline-rep-def Let-def vec-cnj-def*)

lemma *of-ocircline-ounit-circle* [*simp*]: *of-ocircline ounit-circle* = *unit-circle*
by (*transfer*) (*auto*, *rule-tac x=1* **in** *exI*, *simp*)

lemma *of-circline-unit-circline* [*simp*]: *of-circline* (*unit-circle*) = *ounit-circle*
by (*transfer*) (*auto simp add*: *pos-oriented-rep-def of-circline-rep-def*, *rule-tac x=1* **in** *exI*, *simp*)

Oriented x axis and lower half plane

lift-definition *o-x-axis* :: *ocircline* **is** *x-axis-rep*
done

lemma *o-x-axis-pos-oriented*: *pos-oriented o-x-axis*
by *transfer* (*simp add*: *pos-oriented-rep-def*)

lemma *of-ocircline-o-x-axis* [*simp*]: *of-ocircline o-x-axis* = *x-axis*
by *transfer* (*simp del*: *circline-mat-eq-def*)

lemma *of-circline-x-axis* [*simp*]: *of-circline x-axis* = *o-x-axis*
using *of-circline-of-ocircline-pos-oriented*[*of o-x-axis*]
using *o-x-axis-pos-oriented*
by *simp*

lemma *ocircline-set-circline-set-x-axis*: *ocircline-set o-x-axis* = *circline-set x-axis*
by (*subst of-circline-x-axis*[*symmetric*], *subst ocircline-set-circline-set*, *simp*)

lemma [*simp*]: $ii_h \notin$ *disc o-x-axis*
unfolding *disc-def*
by *simp* (*transfer*, *simp add*: *in-ocircline-rep-def Let-def vec-cnj-def*)

lemma [*simp*]: $ii_h \in$ *disc* (*opposite-ocircline o-x-axis*)
unfolding *disc-def*
by *simp* (*transfer*, *simp add*: *in-ocircline-rep-def Let-def vec-cnj-def*)

11.11 Moebius action on oriented circlines and discs

lift-definition *moebius-ocircline* :: *moebius* ⇒ *ocircline* ⇒ *ocircline* **is** *moebius-circline-rep*
proof−
  fix *M M' H H'*
  assume *moebius-mat-eq M M' ocircline-mat-eq H H'*
  thus *ocircline-mat-eq* (*moebius-circline-rep M H*) (*moebius-circline-rep M' H'*)
    by (*auto simp add*: *mat-inv-mult-sm complex-cnj*) (*rule-tac x=ka / Re* (*k* ∗ *cnj k*) **in** *exI*, *auto simp add*: *complex-mult-cnj-cmod power2-eq-square*, *metis divide-pos-pos mult-eq-0-iff norm-mult zero-less-norm-iff*)
qed

lemma *moebius-circline-ocircline*:
  moebius-circline M H = *of-ocircline* (*moebius-ocircline M* (*of-circline H*))
apply (*transfer*)
apply (*auto simp add*: *of-circline-rep-def*)
apply (*rule-tac x=1* **in** *exI*, *simp*)
apply (*rule-tac x=−1* **in** *exI*, *simp add*: *of-real-neg-numeral*)
done

lemma *moebius-ocircline-circline*:
  moebius-ocircline M H = *of-circline* (*moebius-circline M* (*of-ocircline H*)) ∨
    moebius-ocircline M H = *opposite-ocircline* (*of-circline* (*moebius-circline M* (*of-ocircline H*)))
apply (*transfer*)
apply (*auto simp add*: *of-circline-rep-def*)
apply (*rule-tac x=1* **in** *exI*, *simp*)
apply (*erule-tac x=1* **in** *allE*, *simp*)
done

lemma
  inj-moebius-ocircline: *inj* (*moebius-ocircline M*)
unfolding *inj-on-def*
proof (*safe*)
  fix *H H'*
  assume *moebius-ocircline M H* = *moebius-ocircline M H'*
  thus *H* = *H'*

proof (*transfer*)
fix *M H H'*
let *?M* = *Rep-moebius-mat M*
let *?iM* = *mat-inv ?M*
let *?H* = *Rep-circline-mat H* **and** *?H'* = *Rep-circline-mat H'*
assume *ocircline-mat-eq* (*moebius-circline-rep M H*) (*moebius-circline-rep M H'*)
then obtain *k* **where** *congruence ?iM ?H'* = *congruence ?iM* (*cor k* $*_{sm}$ *?H*) *k* > *0*
by *auto*
thus *ocircline-mat-eq H H'*
using *Rep-moebius-mat*[*of M*] *inj-congruence*[*of ?iM ?H' cor k* $*_{sm}$ *?H*] *mat-det-inv*[*of ?M*]
by *auto*
qed
qed

lemma *moebius-ocircline-comp*:
moebius-ocircline M1 (*moebius-ocircline M2 H*) = *moebius-ocircline* (*moebius-comp M1 M2*) *H*
proof (*transfer*)
fix *M1 M2 H*
show *ocircline-mat-eq* (*moebius-circline-rep M1* (*moebius-circline-rep M2 H*)) (*moebius-circline-rep* (*moebius-comp-rep M1 M2*) *H*)
using *congruence-congruence Rep-moebius-mat*[*of M1*] *Rep-moebius-mat*[*of M2*]
by (*simp add*: *mat-inv-mult-mm*, *rule-tac x=1* **in** *exI*, *simp*)
qed

lemma [*simp*]:
moebius-ocircline id-moebius H = *H*
proof *transfer*
fix *H*
show *ocircline-mat-eq* (*moebius-circline-rep id-moebius-rep H*) *H*
by (*cases Rep-circline-mat H*, *simp*) (*rule-tac x=1* **in** *exI*, *simp add*: *mat-adj-def mat-cnj-def*)
qed

lemma *moebius-ocircline-comp-inv*[*simp*]:
moebius-ocircline (*moebius-inv M*) (*moebius-ocircline M H*) = *H*
by (*subst moebius-ocircline-comp*) *simp*

lemma *moebius-circline-opposite-ocircline* [*simp*]:
moebius-ocircline M (*opposite-ocircline H*) = *opposite-ocircline* (*moebius-ocircline M H*)
by *transfer* (*auto*, *rule-tac x=1* **in** *exI*, *simp*)

lemma *moebius-ocircline-set*:
shows *moebius-pt M ‘ ocircline-set H* = *ocircline-set* (*moebius-ocircline M H*)

```
(is ?lhs = ?rhs)
proof−
 have moebius-pt M ‘ ocircline-set H = circline-set (moebius-circline M (of-ocircline
H))
    by (subst moebius-circline-set[symmetric]) simp
  thus ?thesis
    using moebius-ocircline-circline[of M H]
    by auto
qed

lemma moebius-disc:
  moebius-pt M ‘ (disc H) = disc (moebius-ocircline M H)
proof (safe)
  fix z
  assume z ∈ disc H
  thus moebius-pt M z ∈ disc (moebius-ocircline M H)
    unfolding disc-def
  proof (safe)
    assume in-ocircline H z
    thus in-ocircline (moebius-ocircline M H) (moebius-pt M z)
    proof (transfer)
      fix H z M
      assume in-ocircline-rep H z
      thus in-ocircline-rep (moebius-circline-rep M H) (moebius-pt-rep M z)
       using Rep-moebius-mat[of M] quad-form-congruence[of Rep-moebius-mat M
Rep-homo-coords z]
        by (simp add: in-ocircline-rep-def moebius-circline-rep-def Let-def)
    qed
  qed
next
  fix z
  assume z ∈ disc (moebius-ocircline M H)
  thus z ∈ moebius-pt M ‘ disc H
    unfolding disc-def
  proof(safe)
    assume in-ocircline (moebius-ocircline M H) z
    show z ∈ moebius-pt M ‘ Collect (in-ocircline H)
    proof
      show z = moebius-pt M (moebius-pt (moebius-inv M) z)
        using moebius-inv[of M]  bij-moebius-pt[of M]
        by (simp add: bij-def) (metis surj-f-inv-f)
    next
      show moebius-pt (moebius-inv M) z ∈ Collect (in-ocircline H)
        using ‹in-ocircline (moebius-ocircline M H) z›
      proof (safe, transfer)
        fix M H z
         have congruence (mat-inv (mat-inv (Rep-moebius-mat M))) (congruence
(mat-inv (Rep-moebius-mat M)) (Rep-circline-mat H)) =
          Rep-circline-mat H
```

```
     using Rep-moebius-mat[of M]
     by (simp add: congruence-congruence-inv)
   hence quad-form (Rep-homo-coords z) (congruence (mat-inv (Rep-moebius-mat
M)) (Rep-circline-mat H)) =
        quad-form (mat-inv (Rep-moebius-mat M) *mv Rep-homo-coords z)
(Rep-circline-mat H)
    using quad-form-congruence[of mat-inv (Rep-moebius-mat M) Rep-homo-coords
z congruence (mat-inv (Rep-moebius-mat M)) (Rep-circline-mat H)]
     using Rep-moebius-mat[of M] mat-det-inv[of Rep-moebius-mat M]
     by simp
    moreover
    assume in-ocircline-rep (moebius-circline-rep M H) z
    ultimately
    show in-ocircline-rep H (moebius-pt-rep (moebius-inv-rep M) z)
     by (auto simp add: in-ocircline-rep-def Let-def)
   qed
  qed
 qed
qed

lemma moebius-disc-compl:
 moebius-pt M ‘ (disc-compl H) = disc-compl (moebius-ocircline M H)
proof (safe)
 fix z
 assume z ∈ disc-compl H
 thus moebius-pt M z ∈ disc-compl (moebius-ocircline M H)
  unfolding disc-compl-def
 proof (safe)
  assume out-ocircline H z
  thus out-ocircline (moebius-ocircline M H) (moebius-pt M z)
  proof (transfer)
   fix H z M
   assume out-ocircline-rep H z
   thus out-ocircline-rep (moebius-circline-rep M H) (moebius-pt-rep M z)
    using Rep-moebius-mat[of M] quad-form-congruence[of Rep-moebius-mat M
Rep-homo-coords z]
    by (simp add: out-ocircline-rep-def moebius-circline-rep-def Let-def)
  qed
 qed
next
 fix z
 assume z ∈ disc-compl (moebius-ocircline M H)
 thus z ∈ moebius-pt M ‘ disc-compl H
  unfolding disc-compl-def
 proof(safe)
  assume out-ocircline (moebius-ocircline M H) z
  show z ∈ moebius-pt M ‘ Collect (out-ocircline H)
  proof
   show z = moebius-pt M (moebius-pt (moebius-inv M) z)
```

```
        using moebius-inv[of M]  bij-moebius-pt[of M]
        by (simp add: bij-def) (metis surj-f-inv-f)
    next
      show moebius-pt (moebius-inv M) z ∈ Collect (out-ocircline H)
        using ‹out-ocircline (moebius-ocircline M H) z›
      proof (safe, transfer)
        fix M H z
          have congruence (mat-inv (mat-inv (Rep-moebius-mat M))) (congruence
(mat-inv (Rep-moebius-mat M)) (Rep-circline-mat H)) =
          Rep-circline-mat H
          using Rep-moebius-mat[of M]
          by (simp add: congruence-congruence-inv)
      hence quad-form (Rep-homo-coords z) (congruence (mat-inv (Rep-moebius-mat
M)) (Rep-circline-mat H)) =
                quad-form (mat-inv (Rep-moebius-mat M) *_mv Rep-homo-coords z)
(Rep-circline-mat H)
        using quad-form-congruence[of mat-inv (Rep-moebius-mat M) Rep-homo-coords
z congruence (mat-inv (Rep-moebius-mat M)) (Rep-circline-mat H)]
          using Rep-moebius-mat[of M] mat-det-inv[of Rep-moebius-mat M]
          by simp
        moreover
        assume out-ocircline-rep (moebius-circline-rep M H) z
        ultimately
        show out-ocircline-rep H (moebius-pt-rep (moebius-inv-rep M) z)
          by (auto simp add: out-ocircline-rep-def Let-def)
      qed
    qed
  qed
qed



lemma similarity-preserves-lines:
  assumes a ≠ 0
 shows ∞_h ∈ ocircline-set H ⟷ ∞_h ∈ ocircline-set (moebius-ocircline (similarity-moebius
a b) H) (is ?lhs = ?rhs)
proof
  assume ?lhs
  thus ?rhs
    using similarity-inf-fixed[OF ‹a ≠ 0›, of b]
    by (subst moebius-ocircline-set[symmetric]) force
next
  assume ?rhs
  thus ?lhs
    using similarity-only-inf-to-inf[OF ‹a ≠ 0›, of b]
    by (subst (asm) moebius-ocircline-set[symmetric]) (auto, metis)
qed

lemma similarity-preserve-orientation':
```

assumes *a* $\neq$ *0 M* = *similarity-moebius a b H'* = *moebius-ocircline M H* $\infty_h \notin$ *ocircline-set H*
shows *pos-oriented H* $\longrightarrow$ *pos-oriented H'*
proof
have $\infty_h \notin$ *ocircline-set H'*
using *assms similarity-preserves-lines*
by *auto*
assume *pos-oriented H*
hence $\infty_h \in$ *disc-compl H*
using ‹$\infty_h \notin$ *ocircline-set H*› *pos-oriented-circle-inf*[*of H*] *in-on-out*
unfolding *disc-def disc-compl-def ocircline-set-def*
by *auto*
hence $\infty_h \in$ *disc-compl H'*
using ‹*M* = *similarity-moebius a b*› ‹*H'* = *moebius-ocircline M H*›
using *similarity-inf-fixed*[*OF* ‹*a* $\neq$ *0*›, *of b*]
by (*simp*, *subst moebius-disc-compl*[*symmetric*], *force*)
thus *pos-oriented H'*
using *pos-oriented-circle-inf*[*of H'*] *disc-inter-disc-compl*[*of H'*] ‹$\infty_h \notin$ *ocircline-set H'*›
by *auto*
qed

lemma *similarity-preserve-orientation*:
assumes *a* $\neq$ *0 M* = *similarity-moebius a b H'* = *moebius-ocircline M H* $\infty_h \notin$ *ocircline-set H*
shows *pos-oriented H* $\longleftrightarrow$ *pos-oriented H'*
proof−
have $\infty_h \notin$ *ocircline-set H'*
using *assms similarity-preserves-lines*
by *auto*

have ∗: *H* = *moebius-ocircline* (− *similarity-moebius a b*) *H'*
using ‹*H'* = *moebius-ocircline M H*› ‹*M* = *similarity-moebius a b*›
by *simp*
thus *?thesis*
using ‹*a* $\neq$ *0*›
using *similarity-preserve-orientation'*[*OF* ‹*a* $\neq$ *0*› ‹*M* = *similarity-moebius a b*› ‹*H'* = *moebius-ocircline M H*› ‹$\infty_h \notin$ *ocircline-set H*›]
using *similarity-preserve-orientation'*[*OF* - *similarity-moebius-inv*[*of a b*, *OF* ‹*a* $\neq$ *0*›] ∗ ‹$\infty_h \notin$ *ocircline-set H'*›]
by *auto*
qed

lemma $0_h \in$ *disc-compl* (*mk-ocircline* −*1* (*2*∗*ii*) (−*2*∗*ii*) *1*)
unfolding *disc-compl-def*
by *simp* (*transfer*, *simp add*: *out-ocircline-rep-def mk-circline-rep-def Abs-homo-coords-inverse Let-def Abs-circline-mat-inverse hermitean-def mat-adj-def mat-cnj-def vec-cnj-def complex-cnj*)
lemma ¬ *pos-oriented* (*mk-ocircline* −*1* (*2*∗*ii*) (−*2*∗*ii*) *1*)

by *transfer* (*simp add*: *mk-circline-rep-def Abs-circline-mat-inverse hermitean-def mat-adj-def mat-cnj-def complex-cnj pos-oriented-rep-def*)
lemma *circline-type* (*mk-circline* −*1* (*2*∗*ii*) (−*2*∗*ii*) *1*) = −*1*
by *transfer* (*simp add*: *mk-circline-rep-def Abs-circline-mat-inverse hermitean-def mat-adj-def mat-cnj-def complex-cnj circline-type-rep-def*)
lemma 0_h ∈ *disc-compl* (*mk-ocircline 1* (*2*∗*ii*) (−*2*∗*ii*) *1*)
unfolding *disc-compl-def*
by *simp* (*transfer*, *simp add*: *out-ocircline-rep-def mk-circline-rep-def Abs-homo-coords-inverse Let-def Abs-circline-mat-inverse hermitean-def mat-adj-def mat-cnj-def vec-cnj-def complex-cnj*)
lemma *pos-oriented* (*mk-ocircline 1* (*2*∗*ii*) (−*2*∗*ii*) *1*)
by *transfer* (*simp add*: *mk-circline-rep-def Abs-circline-mat-inverse hermitean-def mat-adj-def mat-cnj-def complex-cnj pos-oriented-rep-def*)
lemma *circline-type* (*mk-circline 1* (*2*∗*ii*) (−*2*∗*ii*) *1*) = −*1*
by *transfer* (*simp add*: *mk-circline-rep-def Abs-circline-mat-inverse hermitean-def mat-adj-def mat-cnj-def complex-cnj circline-type-rep-def*)

lemma *reciprocal-preserve-orientation*:
  assumes 0_h ∈ *disc-compl H M* = *reciprocal-moebius H'* = *moebius-ocircline M H*
  shows *pos-oriented H'*
proof−
  have ∞_h ∈ *disc-compl H'*
    using *assms*
    by *simp* (*subst moebius-disc-compl*[*symmetric*], *subst reciprocal-moebius*[*symmetric*], *force*)
  thus *pos-oriented H'*
    using *pos-oriented-circle-inf*[*of H'*] *disc-inter-disc-compl*[*of H'*] *disc-compl-inter-ocircline-set*[*of H'*]
    by *auto*
qed

lemma *reciprocal-not-preserve-orientation*:
  assumes 0_h ∈ *disc H M* = *reciprocal-moebius H'* = *moebius-ocircline M H*
  shows ¬ *pos-oriented H'*
proof−
  have ∞_h ∈ *disc H'*
    using *assms*
    by *simp* (*subst moebius-disc*[*symmetric*], *subst reciprocal-moebius*[*symmetric*], *force*)
  thus ¬ *pos-oriented H'*
    using *pos-oriented-circle-inf*[*of H'*] *disc-inter-ocircline-set*[*of H'*]
    by *auto*
qed

lemma *pole-in-disc*:
  assumes *M* = *mk-moebius a b c d c* ≠ *0 a*∗*d* − *b*∗*c* ≠ *0*
  assumes *is-pole M z z* ∈ *disc H H'* = *moebius-ocircline M H*
  shows ¬ *pos-oriented H'*

```
proof–
  let ?t1 = translation-moebius (a / c)
  let ?rd = rotation-dilatation-moebius ((b * c − a * d) / (c * c))
  let ?r = reciprocal-moebius
  let ?t2 = translation-moebius (d / c)

  have 0_h = moebius-pt (translation-moebius (d/c)) z
    using pole-mk-moebius[of a b c d z] assms
    by simp

  have z ∉ ocircline-set H
    using ‹z ∈ disc H› disc-inter-ocircline-set[of H]
    by auto
  hence 0_h ∉ ocircline-set (moebius-ocircline ?t2 H)
    using ‹0_h = moebius-pt ?t2 z›
    using inj-image-mem-iff[of moebius-pt ?t2 z ocircline-set H] bij-moebius-pt
    by (subst moebius-ocircline-set[symmetric]) (simp add: bij-def)
  hence *: ∞_h ∉ ocircline-set (moebius-ocircline (?r + ?t2) H)
    using reciprocal-homo-only-0-to-inf
   by (simp add: moebius-ocircline-comp[symmetric]) (subst moebius-ocircline-set[symmetric],
subst reciprocal-moebius[symmetric], auto, metis)
  hence **: ∞_h ∉ ocircline-set (moebius-ocircline (?rd + ?r + ?t2) H)
    using ‹a*d − b*c ≠ 0› ‹c ≠ 0›
    using similarity-preserves-lines
    unfolding rotation-dilatation-moebius-def
    by (simp add: moebius-ocircline-comp[symmetric])

  have ¬ pos-oriented (moebius-ocircline (?r + ?t2) H)
    using pole-mk-moebius[of a b c d z] assms
    apply (simp add: moebius-ocircline-comp[symmetric])
    apply (subst reciprocal-not-preserve-orientation, simp-all)
    apply (subst moebius-disc[symmetric])
    apply force
    done
  hence ¬ pos-oriented (moebius-ocircline (?rd + ?r + ?t2) H)
    using *
    using ‹a*d − b*c ≠ 0› ‹c ≠ 0›
    unfolding rotation-dilatation-moebius-def
   by (simp add: moebius-ocircline-comp[symmetric]) (subst similarity-preserve-orientation[symmetric],
simp-all)
  hence ¬ pos-oriented (moebius-ocircline (?t1 + ?rd + ?r + ?t2) H)
    using **
    unfolding translation-moebius-def
   by (simp add: moebius-ocircline-comp[symmetric]) (subst similarity-preserve-orientation[symmetric],
simp-all)

  thus ?thesis
    using assms
   by simp (subst moebius-decomposition, auto simp add: moebius-ocircline-comp[symmetric])
```

```
qed

lemma pole-in-disc-compl:
  assumes M = mk-moebius a b c d c ≠ 0 a∗d − b∗c ≠ 0
  assumes is-pole M z z ∈ disc-compl H H′ = moebius-ocircline M H
  shows pos-oriented H′
proof−
  let ?t1 = translation-moebius (a / c)
  let ?rd = rotation-dilatation-moebius ((b ∗ c − a ∗ d) / (c ∗ c))
  let ?r = reciprocal-moebius
  let ?t2 = translation-moebius (d / c)

  have 0_h = moebius-pt (translation-moebius (d/c)) z
    using pole-mk-moebius[of a b c d z] assms
    by simp

  have z ∉ ocircline-set H
    using ‹z ∈ disc-compl H› disc-compl-inter-ocircline-set[of H]
    by auto
  hence 0_h ∉ ocircline-set (moebius-ocircline ?t2 H)
    using ‹0_h = moebius-pt ?t2 z›
    using inj-image-mem-iff[of moebius-pt ?t2 z ocircline-set H] bij-moebius-pt
    by (subst moebius-ocircline-set[symmetric]) (simp add: bij-def)
  hence ∗: ∞_h ∉ ocircline-set (moebius-ocircline (?r + ?t2) H)
    using reciprocal-homo-only-0-to-inf
   by (simp add: moebius-ocircline-comp[symmetric]) (subst moebius-ocircline-set[symmetric],
subst reciprocal-moebius[symmetric], auto, metis)
  hence ∗∗: ∞_h ∉ ocircline-set (moebius-ocircline (?rd + ?r + ?t2) H)
    using ‹a∗d − b∗c ≠ 0› ‹c ≠ 0›
    using similarity-preserves-lines
    unfolding rotation-dilatation-moebius-def
    by (simp add: moebius-ocircline-comp[symmetric])

  have pos-oriented (moebius-ocircline (?r + ?t2) H)
    using pole-mk-moebius[of a b c d z] assms
    apply (simp add: moebius-ocircline-comp[symmetric])
    apply (subst reciprocal-preserve-orientation, simp-all)
    apply (subst moebius-disc-compl[symmetric])
    apply force
    done
  hence pos-oriented (moebius-ocircline (?rd + ?r + ?t2) H)
    using ∗
    using ‹a∗d − b∗c ≠ 0› ‹c ≠ 0›
    unfolding rotation-dilatation-moebius-def
   by (simp add: moebius-ocircline-comp[symmetric]) (subst similarity-preserve-orientation[symmetric],
simp-all)
  hence pos-oriented (moebius-ocircline (?t1 + ?rd + ?r + ?t2) H)
    using ∗∗
    unfolding translation-moebius-def
```

by (*simp add*: *moebius-ocircline-comp*[*symmetric*]) (*subst similarity-preserve-orientation*[*symmetric*], *simp-all*)

thus *?thesis*
using *assms*
by *simp* (*subst moebius-decomposition*, *auto simp add*: *moebius-ocircline-comp*[*symmetric*])
qed

11.12 Oriented circlines uniqueness

lemma *ocircline-01inf*:
assumes 0_h ∈ *ocircline-set H* ∧ 1_h ∈ *ocircline-set H* ∧ ∞_h ∈ *ocircline-set H*
shows *H* = *o-x-axis* ∨ *H* = *opposite-ocircline o-x-axis*
proof−
have 0_h ∈ *circline-set* (*of-ocircline H*) ∧ 1_h ∈ *circline-set* (*of-ocircline H*) ∧ ∞_h ∈ *circline-set* (*of-ocircline H*)
using *assms*
by *simp*
hence *of-ocircline H* = *x-axis*
using *unique-circline-01inf'*
by *auto*
thus *H* = *o-x-axis* ∨ *H* = *opposite-ocircline o-x-axis*
by (*metis inj-of-ocircline of-ocircline-o-x-axis*)
qed

lemma *unique-ocircline-01inf*: ∃! *H*. 0_h ∈ *ocircline-set H* ∧ 1_h ∈ *ocircline-set H* ∧ ∞_h ∈ *ocircline-set H* ∧ ii_h ∉ *disc H*
proof
show 0_h ∈ *ocircline-set o-x-axis* ∧ 1_h ∈ *ocircline-set o-x-axis* ∧ ∞_h ∈ *ocircline-set o-x-axis* ∧ ii_h ∉ *disc o-x-axis*
by (*simp add*: *ocircline-set-circline-set-x-axis*)
next
fix *H*
assume 0_h ∈ *ocircline-set H* ∧ 1_h ∈ *ocircline-set H* ∧ ∞_h ∈ *ocircline-set H* ∧ ii_h ∉ *disc H*
hence 0_h ∈ *ocircline-set H* ∧ 1_h ∈ *ocircline-set H* ∧ ∞_h ∈ *ocircline-set H* ii_h ∉ *disc H*
by *auto*
hence *H* = *o-x-axis* ∨ *H* = *opposite-ocircline o-x-axis*
using *ocircline-01inf*
by *simp*
thus *H* = *o-x-axis*
using ‹ii_h ∉ *disc H*›
by *auto*
qed

lemma *unique-ocircline-set*:
assumes *A* ≠ *B* *A* ≠ *C* *B* ≠ *C*
shows ∃! *H*. *pos-oriented H* ∧ (*A* ∈ *ocircline-set H* ∧ *B* ∈ *ocircline-set H* ∧ *C*

```
∈ ocircline-set H)
proof−
  obtain M where ∗: moebius-pt M A = 0ₕ  moebius-pt M B = 1ₕ moebius-pt M
C = ∞ₕ
    using ex-moebius-01inf[OF assms]
    by auto
  let ?iM = moebius-pt (moebius-inv M)
  have ∗∗: ?iM 0ₕ = A  ?iM 1ₕ = B  ?iM ∞ₕ = C
    using bij-moebius-pt[of moebius-inv M] ∗
    by (auto simp add: moebius-inv) (metis bij-def bij-moebius-pt inv-f-eq)+
  let ?H = moebius-ocircline (moebius-inv M) o-x-axis
  have 1: A ∈ ocircline-set ?H B ∈ ocircline-set ?H C ∈ ocircline-set ?H
     by − (subst moebius-ocircline-set[symmetric], subst∗∗[symmetric], simp add:
ocircline-set-circline-set-x-axis)+

  have 2: ⋀ H'. A ∈ ocircline-set H' ∧ B ∈ ocircline-set H' ∧ C ∈ ocircline-set
H' ⟹ H' = ?H ∨ H' = opposite-ocircline ?H
  proof−
    fix H'
    let ?H' = ocircline-set H' and ?H'' = ocircline-set (moebius-ocircline M H')
    assume A ∈ ocircline-set H' ∧ B ∈ ocircline-set H' ∧ C ∈ ocircline-set H'
    hence moebius-pt M A ∈ ?H'' moebius-pt M B ∈ ?H'' moebius-pt M C ∈ ?H''
      using moebius-ocircline-set[of M H']
      by auto
    hence 0ₕ ∈ ?H'' 1ₕ ∈ ?H''  ∞ₕ ∈ ?H''
      using ∗
      by auto
    hence moebius-ocircline M H' = o-x-axis ∨ moebius-ocircline M H' = opposite-ocircline
o-x-axis
      using ocircline-01inf
      by auto
     hence o-x-axis = moebius-ocircline M H' ∨  o-x-axis = opposite-ocircline
(moebius-ocircline M H')
      by auto
    thus H' = ?H ∨ H' = opposite-ocircline ?H
    proof
      assume ∗: o-x-axis = moebius-ocircline M H'
     show H' = moebius-ocircline (moebius-inv M) o-x-axis ∨ H' = opposite-ocircline
(moebius-ocircline (moebius-inv M) o-x-axis)
        by (rule disjI1) (subst ∗, simp)
    next
      assume ∗: o-x-axis = opposite-ocircline (moebius-ocircline M H')
     show H' = moebius-ocircline (moebius-inv M) o-x-axis ∨ H' = opposite-ocircline
(moebius-ocircline (moebius-inv M) o-x-axis)
        by (rule disjI2) (subst ∗, simp)
    qed
  qed

  show ?thesis
  proof (cases pos-oriented ?H)
    case True
    thus ?thesis
      unfolding Ex1-def
     proof (rule-tac x=moebius-ocircline (moebius-inv M) o-x-axis in exI, simp
add: 1, safe)
      fix y
     assume pos-oriented y A ∈ ocircline-set y B ∈ ocircline-set y C ∈ ocircline-set
y
      thus y = moebius-ocircline (moebius-inv M) o-x-axis
        using 2[of y] True
        by (auto simp add: pos-oriented-opposite-ocircline)
    qed
  next
    case False
    thus ?thesis
      unfolding Ex1-def
   proof (rule-tac x=opposite-ocircline (moebius-ocircline (moebius-inv M) o-x-axis)
in exI, simp add: 1 pos-oriented-opposite-ocircline, safe)
      fix y
     assume pos-oriented y A ∈ ocircline-set y B ∈ ocircline-set y C ∈ ocircline-set
y
      thus y = opposite-ocircline (moebius-ocircline (moebius-inv M) o-x-axis)
        using 2[of y] False
        by (auto simp add: pos-oriented-opposite-ocircline)
    qed
  qed
qed


definition chordal-circle-rep where
 chordal-circle-rep a r =
   (let (a1, a2) = Rep-homo-coords a in
     mk-circline-rep (4*a2*cnj a2 − (cor r)^2*(a1*cnj a1 + a2*cnj a2)) (−4*a1*cnj
a2) (−4*cnj a1*a2) (4*a1*cnj a1 − (cor r)^2*(a1*cnj a1 + a2*cnj a2)))

lemma [simp]: Rep-circline-mat (chordal-circle-rep a r) = (let (a1, a2) = Rep-homo-coords
a in
  (4*a2*cnj a2 − (cor r)^2*(a1*cnj a1 + a2*cnj a2), −4*a1*cnj a2, −4*cnj
a1*a2, 4*a1*cnj a1 − (cor r)^2*(a1*cnj a1 + a2*cnj a2)))
proof−
  obtain a1 a2 where aa: Rep-homo-coords a = (a1, a2)
    by (rule obtain-homo-coords)
  hence (4 * a2 * cnj a2 − (cor r)^2 * (a1 * cnj a1 + a2 * cnj a2), −4 * a1 *
cnj a2, −4 * cnj a1 * a2, 4 * a1 * cnj a1 − (cor r)^2 * (a1 * cnj a1 + a2 * cnj
a2)) ∈ {H. hermitean H ∧ H ≠ mat-zero}
    using Rep-homo-coords[of a]
   by (auto simp add: hermitean-def mat-adj-def mat-cnj-def complex-cnj power2-eq-square)
  thus ?thesis
```

```
    using aa
   by (simp add: chordal-circle-rep-def split-def Let-def mk-circline-rep-def Abs-circline-mat-inverse)
qed

lift-definition chordal-circle :: complex-homo ⇒ real ⇒ circline is chordal-circle-rep
proof−
  fix a b r
  obtain a1 a2 where aa: Rep-homo-coords a = (a1, a2)
    by (rule obtain-homo-coords)
  obtain b1 b2 where bb: Rep-homo-coords b = (b1, b2)
    by (rule obtain-homo-coords)
  assume a ≈ b
  then obtain k where Rep-homo-coords b = (k ∗ a1, k ∗ a2) k ≠ 0
    using aa bb
    by auto
  moreover
  have cor (Re (k ∗ cnj k)) = k ∗ cnj k
    by (metis complex-In-mult-cnj-zero complex-of-real-Re)
  ultimately
  show circline-mat-eq (chordal-circle-rep a r) (chordal-circle-rep b r)
    using aa bb
     by (auto simp add: complex-cnj) (rule-tac x=Re (k∗cnj k) in exI, auto simp
add: field-simps)
qed

lemma sqrt-1-plus-square: sqrt (1 + a^2) ≠ 0
  by (smt real-sqrt-less-mono real-sqrt-zero realpow-square-minus-le)

lemma
  assumes dist-homo z a = r
  shows z ∈ circline-set (chordal-circle a r)
proof (cases a ≠ ∞_h)
  case True
  then obtain a' where  a = of-complex a'
    using inf-homo-or-complex-homo[of a]
    by auto
  let ?A = 4 − (cor r)^2 ∗ (1 + (a'∗cnj a')) and ?B = −4∗a' and ?C=−4∗cnj
a' and ?D = 4∗a'∗cnj a' −  (cor r)^2 ∗ (1 + (a'∗cnj a'))
  have hh: (?A, ?B, ?C, ?D) ∈ {H. hermitean H ∧ H ≠ mat-zero}
   by (auto simp add: hermitean-def mat-adj-def mat-cnj-def complex-cnj power2-eq-square)
  hence ∗: chordal-circle (of-complex a') r = mk-circline ?A ?B ?C ?D
    by (transfer, simp add: mk-circline-rep-def Abs-circline-mat-inverse) (rule-tac
x=1 in exI, simp)

  show ?thesis
  proof (cases z ≠ ∞_h)
    case True
    then obtain z' where z = of-complex z'
      using inf-homo-or-complex-homo[of z] inf-homo-or-complex-homo[of a]
      by auto
    have 2 * cmod (z' - a') / (sqrt (1 + (cmod z')^2) * sqrt (1 + (cmod a')^2)) = r
      using dist-homo-finite[of z' a'] assms ‹z = of-complex z'› ‹a = of-complex a'›
      by auto
    hence 4 * (cmod (z' - a'))^2 / ((1 + (cmod z')^2) * (1 + (cmod a')^2)) = r^2
      by (auto simp add: power-mult-distrib power-divide field-simps)
    moreover
    have sqrt (1 + (cmod z')^2) ≠ 0 sqrt (1 + (cmod a')^2) ≠ 0
      using sqrt-1-plus-square
      by simp+
    hence (1 + (cmod z')^2) * (1 + (cmod a')^2) ≠ 0
      by simp
    ultimately
    have 4 * (cmod (z' - a'))^2  = r^2 * ((1 + (cmod z')^2) * (1 + (cmod a')^2))
      by (simp add: field-simps)
    hence 4 * Re ((z' - a')*cnj (z' - a')) = r^2 * (1 + Re (z'*cnj z')) * (1 + Re (a'*cnj a'))
      by ((subst cmod-square[symmetric])+, simp)
    hence 4 * (Re(z'*cnj z') - Re(a'*cnj z') - Re(cnj a'*z') + Re(a'*cnj a')) = r^2 * (1 + Re (z'*cnj z')) * (1 + Re (a'*cnj a'))
      by (simp add: complex-cnj field-simps)
    hence Re (?A * z' * cnj z' + ?B * cnj z' + ?C * z' + ?D) = 0
      by (simp add: power2-eq-square field-simps)
    hence ?A * z' * cnj z' + ?B * cnj z' + ?C * z' + ?D = 0
      by (subst complex-eq-if-Re-eq) (auto simp add: power2-eq-square)
    hence (cnj z' * ?A + ?C) * z' + (cnj z' * ?B + ?D) = 0
      by algebra
    hence z ∈ circline-set (mk-circline ?A ?B ?C ?D)
      using ‹z = of-complex z'› hh
      unfolding circline-set-def
      by simp (transfer, simp add: of-complex-coords-def Abs-homo-coords-inverse on-circline-rep-def Let-def Abs-circline-mat-inverse mk-circline-rep-def vec-cnj-def)
    thus ?thesis
      using *
      by (subst ‹a = of-complex a'›) simp
  next
    case False
    hence 2 / sqrt (1 + (cmod a')^2) = r
      using assms ‹a = of-complex a'›
      using dist-homo-infinite2[of a']
      by simp
    moreover
    have sqrt (1 + (cmod a')^2) ≠ 0
      using sqrt-1-plus-square
      by simp
    ultimately
    have 2 = r * sqrt (1 + (cmod a')^2)
      by (simp add: field-simps)
    hence 4 = (r ∗ sqrt (1 + (cmod a')^2))^2
      by simp
    hence 4 = r^2 ∗ (1 + (cmod a')^2)
      by (simp add: power-mult-distrib)
    hence Re (4 − (cor r)^2 ∗ (1 + (a' ∗ cnj a'))) = 0
      by (subst (asm) cmod-square) (simp add: field-simps power2-eq-square)
    hence 4 − (cor r)^2 ∗ (1 + (a'∗cnj a')) = 0
      by (subst complex-eq-if-Re-eq) (auto simp add: power2-eq-square)
    hence circline-A0 (mk-circline ?A ?B ?C ?D)
      using hh
     by simp (transfer, simp add: circline-A0-rep-def mk-circline-rep-def Abs-circline-mat-inverse)
    hence z ∈ circline-set (mk-circline ?A ?B ?C ?D)
      using inf-in-circline-set[of mk-circline ?A ?B ?C ?D]
      using ‹¬ z ≠ ∞h›
      by simp
    thus ?thesis
      using ∗
      by (subst ‹a = of-complex a'›) simp
  qed
next
  case False
  let ?A = −(cor r)^2 and ?B = 0 and ?C = 0 and ?D = 4 −(cor r)^2
  have hh: (?A, ?B, ?C, ?D) ∈ {H. hermitean H ∧ H ≠ mat-zero}
   by (auto simp add: hermitean-def mat-adj-def mat-cnj-def complex-cnj power2-eq-square)
  hence ∗: chordal-circle a r = mk-circline ?A ?B ?C ?D
    using ‹¬ a ≠ ∞h›
      by simp (transfer, simp add: mk-circline-rep-def Abs-circline-mat-inverse,
rule-tac x=1 in exI, simp)

  show ?thesis
  proof (cases z = ∞h)
    case True
    show ?thesis
      using assms ‹z = ∞h› ‹¬ a ≠ ∞h›
      using ∗ hh
        by (simp, subst inf-in-circline-set, transfer, simp add: circline-A0-rep-def
mk-circline-rep-def Abs-circline-mat-inverse)
  next
    case False
    then obtain z' where z = of-complex z'
      using inf-homo-or-complex-homo[of z]
      by auto
    have 2 / sqrt (1 + (cmod z')^2) = r
      using assms ‹z = of-complex z'›‹¬ a ≠ ∞h›
      using dist-homo-infinite2[of z']
      by simp
    moreover
    have sqrt (1 + (cmod z')^2) ≠ 0
      using sqrt-1-plus-square
      by simp
    ultimately
    have 2 = r ∗ sqrt (1 + (cmod z')^2)
      by (simp add: field-simps)
    hence 4 = (r ∗ sqrt (1 + (cmod z')^2))^2
      by simp
    hence 4 = r^2 ∗ (1 + (cmod z')^2)
      by (simp add: power-mult-distrib)
    hence Re (4 − (cor r)^2 ∗ (1 + (z' ∗ cnj z'))) = 0
      by (subst (asm) cmod-square) (simp add: field-simps power2-eq-square)
    hence − (cor r)^2 ∗ z'∗cnj z' + 4 − (cor r)^2 = 0
      by (subst complex-eq-if-Re-eq) (auto simp add: power2-eq-square field-simps)
    hence z ∈ circline-set (mk-circline ?A ?B ?C ?D)
      using hh
      unfolding circline-set-def
    by (subst ‹z = of-complex z'›, simp) (transfer, auto simp add: on-circline-rep-def
Let-def mk-circline-rep-def Abs-circline-mat-inverse vec-cnj-def field-simps)
    thus ?thesis
      using ∗
      by simp
  qed
qed

lemma [simp]: sqrt 4 = 2
proof−
  have sqrt (2^2) = 2
    by (metis abs-numeral real-sqrt-abs)
  thus ?thesis
    by simp
qed

lemma
  assumes z ∈ circline-set (chordal-circle a r) r ≥ 0
  shows dist-homo z a = r
proof (cases a = ∞_h)
  case False
  then obtain a' where a = of-complex a'
    using inf-homo-or-complex-homo[of a]
    by auto

  let ?A = 4 − (cor r)^2 ∗ (1 + (a'∗cnj a')) and ?B = −4∗a' and ?C=−4∗cnj
a' and ?D = 4∗a'∗cnj a' −  (cor r)^2 ∗ (1 + (a'∗cnj a'))
  have hh: (?A, ?B, ?C, ?D) ∈ {H. hermitean H ∧ H ≠ mat-zero}
   by (auto simp add: hermitean-def mat-adj-def mat-cnj-def complex-cnj power2-eq-square)
  hence ∗: chordal-circle (of-complex a') r = mk-circline ?A ?B ?C ?D
    by (transfer, simp add: mk-circline-rep-def Abs-circline-mat-inverse) (rule-tac
x=1 in exI, simp)

  show ?thesis
  proof (cases z = ∞h)
    case False
    then obtain z' where z = of-complex z'
      using inf-homo-or-complex-homo[of z] inf-homo-or-complex-homo[of a]
      by auto
    hence z ∈ circline-set (mk-circline ?A ?B ?C ?D)
      using assms ‹a = of-complex a'› *
      by simp
    hence (cnj z' * ?A + ?C) * z' + (cnj z' * ?B + ?D) = 0
      using hh
      unfolding circline-set-def
     by (subst (asm) ‹z = of-complex z'›, simp) (transfer, simp add: on-circline-rep-def
Let-def mk-circline-rep-def Abs-circline-mat-inverse vec-cnj-def)
    hence ?A * z' * cnj z' + ?B * cnj z' + ?C * z' + ?D = 0
      by algebra
    hence Re (?A * z' * cnj z' + ?B * cnj z' +?C * z' +?D) = 0
      by (simp add: power2-eq-square field-simps)
     hence 4 * Re ((z' - a')*cnj (z' - a')) = r^2 * (1 + Re (z'*cnj z')) * (1 +
Re (a'*cnj a'))
      by (simp add: complex-cnj field-simps power2-eq-square)
    hence 4 * (cmod (z' - a'))^2  = r^2 * ((1 + (cmod z')^2) * (1 + (cmod a')^2))
      by (subst cmod-square)+ simp
    moreover
    have sqrt (1 + (cmod z')^2) ≠ 0 sqrt (1 + (cmod a')^2) ≠ 0
      using sqrt-1-plus-square
      by simp+
    hence (1 + (cmod z')^2) * (1 + (cmod a')^2) ≠ 0
      by simp
    ultimately
    have 4 * (cmod (z' - a'))^2 / ((1 + (cmod z')^2) * (1 + (cmod a')^2)) = r^2
      by (simp add: field-simps)
     hence 2 * cmod (z' - a') / (sqrt (1 + (cmod z')^2) * sqrt (1 + (cmod a')^2))
= r
      using ‹r ≥ 0›
     by (subst (asm) real-sqrt-eq-iff[symmetric]) (simp add: real-sqrt-mult real-sqrt-divide)
    thus ?thesis
      using ‹z = of-complex z'› ‹a = of-complex a'›
      using dist-homo-finite[of z' a']
      by simp
  next
    case True
    have z ∈ circline-set (mk-circline ?A ?B ?C ?D)
      using assms ‹a = of-complex a'› *
      by simp
    hence circline-A0 (mk-circline ?A ?B ?C ?D)
      using inf-in-circline-set[of mk-circline ?A ?B ?C ?D]
      using ‹z = ∞h›
      by simp
    hence 4 - (cor r)^2 * (1 + (a'*cnj a')) = 0
```

```
      using hh
     by transfer (simp add: circline-A0-rep-def mk-circline-rep-def Abs-circline-mat-inverse)
    hence Re (4 − (cor r)^2 ∗ (1 + (a' ∗ cnj a'))) = 0
      by simp
    hence 4 = r^2 ∗ (1 + (cmod a')^2)
      by (subst cmod-square) (simp add: power2-eq-square)
    hence 2 = r ∗ sqrt (1 + (cmod a')^2)
      using ‹r ≥ 0›
      by (subst (asm) real-sqrt-eq-iff [symmetric]) (simp add: real-sqrt-mult)
    moreover
    have sqrt (1 + (cmod a')^2) ≠ 0
      using sqrt-1-plus-square
      by simp
    ultimately
    have 2 / sqrt (1 + (cmod a')^2) = r
      by (simp add: field-simps)
    thus ?thesis
      using ‹a = of-complex a'› ‹z = ∞_h›
      using dist-homo-infinite2 [of a']
      by simp
  qed
next
  case True
  let ?A = −(cor r)^2 and ?B = 0 and ?C = 0 and ?D = 4 −(cor r)^2
  have hh: (?A, ?B, ?C, ?D) ∈ {H. hermitean H ∧ H ≠ mat-zero}
   by (auto simp add: hermitean-def mat-adj-def mat-cnj-def complex-cnj power2-eq-square)
  hence ∗: chordal-circle a r = mk-circline ?A ?B ?C ?D
    using ‹a = ∞_h›
      by simp (transfer, simp add: mk-circline-rep-def Abs-circline-mat-inverse,
rule-tac x=1 in exI, simp)

  show ?thesis
  proof (cases z = ∞_h)
    case True
    thus ?thesis
      using ‹a = ∞_h› assms ∗ hh
     by simp (subst (asm) inf-in-circline-set, transfer, simp add: circline-A0-rep-def
mk-circline-rep-def Abs-circline-mat-inverse)
  next
    case False
    then obtain z' where z = of-complex z'
      using inf-homo-or-complex-homo[of z] inf-homo-or-complex-homo[of a]
      by auto
    hence z ∈ circline-set (mk-circline ?A ?B ?C ?D)
      using assms ∗
      by simp
    hence − (cor r)^2 ∗ z'∗cnj z' + 4 − (cor r)^2 = 0
      using hh
      unfolding circline-set-def
      apply (subst (asm) ‹z = of-complex z'›)
      by (simp, transfer) (simp add: on-circline-rep-def mk-circline-rep-def Let-def
vec-cnj-def Abs-circline-mat-inverse, algebra)
    hence 4 − (cor r)² ∗ (1 + (z'∗cnj z')) = 0
      by (simp add: field-simps)
    hence Re (4 − (cor r)² ∗ (1 + (z' ∗ cnj z'))) = 0
      by simp
    hence 4 = r² ∗ (1 + (cmod z')²)
      by (subst cmod-square) (simp add: power2-eq-square)
    hence 2 = r ∗ sqrt (1 + (cmod z')²)
      using ‹r ≥ 0›
      by (subst (asm) real-sqrt-eq-iff[symmetric]) (simp add: real-sqrt-mult)
    moreover
    have sqrt (1 + (cmod z')²) ≠ 0
      using sqrt-1-plus-square
      by simp
    ultimately
    have 2 / sqrt (1 + (cmod z')²) = r
      by (simp add: field-simps)
    thus ?thesis
      using ‹z = of-complex z'› ‹a = ∞_h›
      using dist-homo-infinite2[of z']
      by simp
  qed
qed

lemma chordal-circle-radius-positive:
  assumes hermitean (A, B, C, D) Re (mat-det (A, B, C, D)) ≤ 0 B ≠ 0
  dsc = sqrt(Re ((D−A)² + 4 ∗ (B∗cnj B))) a1 = (A − D + cor dsc) / (2 ∗ C)
a2 = (A − D − cor dsc) / (2 ∗ C)
  shows Re (A∗a1/B) ≥ −1 ∧ Re (A∗a2/B) ≥ −1
proof−
  from assms have is-real A is-real D C = cnj B
    using hermitean-elems
    by auto
  have ∗: A∗a1/B = ((A − D + cor dsc) / (2 ∗ (B ∗ cnj B))) ∗ A
    using ‹B ≠ 0› ‹C = cnj B› ‹a1 = (A − D + cor dsc) / (2 ∗ C)›
    by (simp add: field-simps) algebra
  have ∗∗: A∗a2/B = ((A − D − cor dsc) / (2 ∗ (B ∗ cnj B))) ∗ A
    using ‹B ≠ 0› ‹C = cnj B› ‹a2 = (A − D − cor dsc) / (2 ∗ C)›
    by (simp add: field-simps) algebra
  have dsc ≥ 0
  proof−
    have 0 ≤ Re ((D − A)²) + 4 ∗ Re ((cor (cmod B))²)
      using ‹is-real A› ‹is-real D›
     by (subst cor-squared, subst Re-complex-of-real) (simp add: power2-eq-square)
    thus ?thesis
      using ‹dsc = sqrt(Re ((D−A)² + 4∗(B∗cnj B)))›
      by (subst (asm) complex-mult-cnj-cmod) simp
  qed
  hence Re (A − D − cor dsc) ≤ Re (A − D + cor dsc)
    by simp
  moreover
  have Re (2 ∗ (B ∗ cnj B)) > 0
    using ‹B ≠ 0›
   by (subst complex-mult-cnj-cmod, simp add: power2-eq-square) (metis norm-eq-zero
not-real-square-gt-zero)
  ultimately
  have xxx: Re (A − D + cor dsc) / Re (2 ∗ (B ∗ cnj B)) ≥ Re (A − D − cor
dsc) / Re (2 ∗ (B ∗ cnj B)) (is ?lhs ≥ ?rhs)
    by (metis divide-right-mono less-eq-real-def)

  have Re A ∗ Re D ≤ Re (B∗cnj B)
    using ‹Re (mat-det (A, B, C, D)) ≤ 0› ‹C = cnj B› ‹is-real A› ‹is-real D›
    by simp

  show ?thesis
  proof (cases Re A > 0)
    case True
    hence Re (A∗a1/B) ≥ Re (A∗a2/B)
      using ∗ ∗∗ ‹Re (2 ∗ (B ∗ cnj B)) > 0› ‹B ≠ 0› ‹is-real A› ‹is-real D› xxx
      using mult-right-mono[of ?rhs ?lhs Re A]
      apply simp
      apply (subst Re-divide-real, simp, simp)
      apply (subst Re-divide-real, simp, simp)
      apply (subst Re-mult-real, simp)+
      apply simp
      done
    moreover
    have Re (A∗a2/B) ≥ −1
    proof−
      from ‹Re A ∗ Re D ≤ Re (B∗cnj B)›
      have Re (A^2) ≤ Re (B∗cnj B) + Re ((A − D)∗A)
        using ‹Re A > 0› ‹is-real A› ‹is-real D›
        by (simp add: power2-eq-square field-simps)
      have 1 ≤ Re (B∗cnj B) / Re (A^2) + Re (A − D) / Re A
        using ‹Re A > 0› ‹is-real A› ‹is-real D›
       using divide-right-mono[OF ‹Re (A^2) ≤ Re (B∗cnj B) + Re ((A − D)∗A)›,
of Re (A^2)]
        by (simp add: power2-eq-square add-divide-distrib)
      have 4 ∗ Re(B∗cnj B) ≤ 4 ∗ (Re (B∗cnj B))^2 / Re (A^2)  + 2∗Re (A − D)
/ Re A ∗ 2 ∗ Re(B∗cnj B)
        using mult-right-mono[OF ‹1 ≤ Re (B∗cnj B) / Re (A^2) + Re (A − D) /
Re A›, of 4 ∗ Re (B∗cnj B)]
        by (simp add: distrib-right) (simp add: power2-eq-square field-simps)
      moreover
      have A ≠ 0
        using ‹Re A > 0›
        by auto
      hence 4 * (Re (B*cnj B))^2 / Re (A^2) = Re (4 * (B*cnj B)^2 / A^2)
        using Re-divide-real[of A^2 4 * (B*cnj B)^2] ‹Re A > 0› ‹is-real A›
        by (auto simp add: power2-eq-square)
      moreover
      have 2*Re (A − D) / Re A * 2 * Re(B*cnj B) = Re (2 * (A − D) / A *
2 * B * cnj B)
        using ‹is-real A› ‹is-real D› ‹A ≠ 0›
        using Re-divide-real[of A (4 * A − 4 * D) * B * cnj B]
        by (simp add: field-simps)
      ultimately
      have Re ((D − A)^2 + 4 * B*cnj B) ≤ Re((A − D)^2 + 4 * (B*cnj B)^2 /
A^2  + 2*(A − D) / A * 2 * B*cnj B)
        by (simp add: field-simps power2-eq-square)
      hence Re ((D − A)^2 + 4 * B*cnj B) ≤ Re(((A − D) +  2 * B*cnj B /
A)^2)
        using ‹A ≠ 0›
        by (subst power2-sum) (simp add: power2-eq-square field-simps)
      hence dsc ≤ sqrt (Re(((A − D) +  2 * B*cnj B / A)^2))
        using ‹dsc = sqrt(Re ((D−A)^2 + 4*(B*cnj B)))›
        by simp
      moreover
      have Re(((A − D) +  2 * B*cnj B / A)^2) = (Re((A − D) +  2 * B*cnj B
/ A))^2
        using ‹is-real A› ‹is-real D› div-reals
        by (simp add: power2-eq-square)
      ultimately
      have dsc ≤ |Re (A − D + 2 * B * cnj B / A)|
        by simp
      moreover
      have Re (A − D + 2 * B * cnj B / A) ≥ 0
      proof−
        have Re (A^2 + B*cnj B) ≥ 0
          using ‹is-real A›
          by (simp add: power2-eq-square)
        hence Re (A^2 + 2*B*cnj B − A*D) ≥ 0
          using ‹Re A * Re D ≤ Re (B*cnj B)›
          using ‹is-real A› ‹is-real D›
          by simp
        show ?thesis
          using divide-right-mono[OF ‹Re (A^2 + 2*B*cnj B − A*D) ≥ 0›, of Re
A] ‹Re A > 0› ‹is-real A› ‹A ≠ 0›
          by (simp add: add-divide-distrib diff-divide-distrib del: complex-Re-mult)
(subst Re-divide-real, auto simp add: power2-eq-square field-simps)
      qed
      ultimately
      have dsc ≤ Re (A − D + 2 * B * cnj B / A)
        by simp
      hence − Re (2 * (B * cnj B) / A) ≤ Re ((A − D − cor dsc))
```

```
        by (simp add: field-simps)
      hence − (Re (2 ∗ (B ∗ cnj B)) / Re A) ≤ Re (A − D − cor dsc)
        using ‹is-real A› ‹A ≠ 0›
        by (subst (asm) Re-divide-real, auto)
      from divide-right-mono[OF this, of Re (2 ∗ B ∗ cnj B)]
      have − 1 / Re A ≤ Re (A − D − cor dsc) / Re (2 ∗ B ∗ cnj B)
        using ‹Re A > 0› ‹B ≠ 0› ‹A ≠ 0› ‹0 < Re (2 ∗ (B ∗ cnj B))›
        by (simp add: field-simps del: complex-Re-mult)
      from mult-right-mono[OF this, of Re A]
      show ?thesis
        using ‹is-real A› ‹is-real D› ‹B ≠ 0› ‹Re A > 0› ‹A ≠ 0›
        apply (subst ∗∗)
        apply (subst Re-mult-real, simp add: div-reals)
        apply (subst Re-divide-real, simp, simp)
        apply (simp add: field-simps)
        done
    qed
    ultimately
    show ?thesis
      by simp
  next
    case False
    show ?thesis
    proof (cases Re A < 0)
      case True
      hence Re (A∗a1/B) ≤ Re (A∗a2/B)
        using ∗ ∗∗ ‹Re (2 ∗ (B ∗ cnj B)) > 0› ‹B ≠ 0› ‹is-real A› ‹is-real D› xxx
        using mult-right-mono-neg[of ?rhs ?lhs Re A]
        apply simp
        apply (subst Re-divide-real, simp, simp)
        apply (subst Re-divide-real, simp, simp)
        apply (subst Re-mult-real, simp)+
        apply simp
        done
      moreover
      have Re (A∗a1/B) ≥ −1
      proof−
        from ‹Re A ∗ Re D ≤ Re (B∗cnj B)›
        have Re (A^2) ≤ Re (B∗cnj B) − Re ((D − A)∗A)
          using ‹Re A < 0› ‹is-real A› ‹is-real D›
          by (simp add: power2-eq-square field-simps)
        hence 1 ≤ Re (B∗cnj B) / Re (A^2) − Re (D − A) / Re A
          using ‹Re A < 0› ‹is-real A› ‹is-real D›
        using divide-right-mono[OF ‹Re (A^2) ≤ Re (B∗cnj B) − Re ((D − A)∗A)›,
of Re (A^2)]
          by (simp add: power2-eq-square diff-divide-distrib)
        have 4 ∗ Re(B∗cnj B) ≤ 4 ∗ (Re (B∗cnj B))^2 / Re (A^2)  − 2∗Re (D −
A) / Re A ∗ 2 ∗ Re(B∗cnj B)
          using mult-right-mono[OF ‹1 ≤ Re (B∗cnj B) / Re (A^2) − Re (D − A)
/ Re A›, of 4 * Re (B*cnj B)]
          by (simp add: left-diff-distrib) (simp add: power2-eq-square field-simps)
        moreover
        have A ≠ 0
          using ‹Re A < 0›
          by auto
        hence 4 * (Re (B*cnj B))^2 / Re (A^2) = Re (4 * (B*cnj B)^2 / A^2)
          using Re-divide-real[of A^2 4 * (B*cnj B)^2] ‹Re A < 0› ‹is-real A›
          by (auto simp add: power2-eq-square)
        moreover
        have 2*Re (D − A) / Re A * 2 * Re(B*cnj B) = Re (2 * (D − A) / A *
2 * B * cnj B)
          using ‹is-real A› ‹is-real D› ‹A ≠ 0›
          using Re-divide-real[of A (4 * D − 4 * A) * B * cnj B]
          by (simp add: field-simps)
        ultimately
        have Re ((D − A)^2 + 4 * B*cnj B) ≤ Re((D − A)^2 + 4 * (B*cnj B)^2 /
A^2 − 2*(D − A) / A * 2 * B*cnj B)
          by (simp add: field-simps power2-eq-square)
        hence Re ((D − A)^2 + 4 * B*cnj B) ≤ Re(((D − A) − 2 * B*cnj B /
A)^2)
          using ‹A ≠ 0›
          by (subst power2-diff) (simp add: power2-eq-square field-simps)
        hence dsc ≤ sqrt (Re(((D − A) − 2 * B*cnj B / A)^2))
          using ‹dsc = sqrt(Re ((D−A)^2 + 4*(B*cnj B)))›
          by simp
        moreover
        have Re(((D − A) − 2 * B*cnj B / A)^2) = (Re((D − A) − 2 * B*cnj
B / A))^2
          using ‹is-real A› ‹is-real D› div-reals
          by (simp add: power2-eq-square)
        ultimately
        have dsc ≤ |Re (D − A − 2 * B * cnj B / A)|
          by simp
        moreover
        have Re (D − A − 2 * B * cnj B / A) ≥ 0
        proof−
          have Re (A^2 + B*cnj B) ≥ 0
            using ‹is-real A›
            by (simp add: power2-eq-square)
          hence Re (A^2 + 2*B*cnj B − A*D) ≥ 0
            using ‹Re A * Re D ≤ Re (B*cnj B)›
            using ‹is-real A› ‹is-real D›
            by simp
          show ?thesis
            using divide-right-mono-neg[OF ‹Re (A^2 + 2*B*cnj B − A*D) ≥ 0›,
of Re A] ‹Re A < 0› ‹is-real A› ‹A ≠ 0›
           by (simp add: add-divide-distrib diff-divide-distrib del: complex-Re-mult)
(subst Re-divide-real, auto simp add: power2-eq-square field-simps)
```

```
        qed
        ultimately
        have dsc ≤ Re (D − A − 2 ∗ B ∗ cnj B / A)
          by simp
        hence − Re (2 ∗ (B ∗ cnj B) / A) ≥ Re ((A − D + cor dsc))
          by (simp add: field-simps)
        hence − (Re (2 ∗ (B ∗ cnj B)) / Re A) ≥ Re (A − D + cor dsc)
          using ‹is-real A› ‹A ≠ 0›
          by (subst (asm) Re-divide-real, auto)
        from divide-right-mono[OF this, of Re (2 ∗ B ∗ cnj B)]
        have − 1 / Re A ≥ Re (A − D + cor dsc) / Re (2 ∗ B ∗ cnj B)
          using ‹Re A < 0› ‹B ≠ 0› ‹A ≠ 0› ‹0 < Re (2 ∗ (B ∗ cnj B))›
          by (simp add: field-simps del: complex-Re-mult)
        from mult-right-mono-neg[OF this, of Re A]
        show ?thesis
          using ‹is-real A› ‹is-real D› ‹B ≠ 0› ‹Re A < 0› ‹A ≠ 0›
          apply (subst ∗)
          apply (subst Re-mult-real, simp add: div-reals)
          apply (subst Re-divide-real, simp, simp)
          apply (simp add: field-simps)
          done
      qed
      ultimately
      show ?thesis
        by simp
    next
      case False
      hence A = 0
        using ‹¬ Re A > 0› ‹is-real A›
        by (cases A) simp
      thus ?thesis
        by simp
    qed
  qed
qed

definition chordal-circles-rep where
  chordal-circles-rep H =
    (let (A, B, C, D) = Rep-circline-mat H;
         dsc = sqrt(Re ((D−A)^2 + 4 ∗ (B∗cnj B)));
         a1 = (A − D + cor dsc) / (2 ∗ C);
         r1 = sqrt((4 − Re((−4 ∗ a1/B) ∗ A)) / (1 + Re (a1∗cnj a1)));
         a2 = (A − D − cor dsc) / (2 ∗ C);
         r2 = sqrt((4 − Re((−4 ∗ a2/B) ∗ A)) / (1 + Re (a2∗cnj a2)))
      in ((a1, r1), (a2, r2)))

lift-definition chordal-circles :: ocircline ⇒ (complex × real) × (complex × real)
is chordal-circles-rep
proof−
```

```
  fix H1 H2
  obtain A1 B1 C1 D1 where hh1: (A1, B1, C1, D1) = Rep-circline-mat H1
    by (cases Rep-circline-mat H1) auto
  obtain A2 B2 C2 D2 where hh2: (A2, B2, C2, D2) = Rep-circline-mat H2
    by (cases Rep-circline-mat H2) auto

  assume ocircline-mat-eq H1 H2
  then obtain k where ∗: k > 0 A2 = cor k ∗ A1 B2 = cor k ∗ B1 C2 = cor k
∗ C1 D2 = cor k ∗ D1
    using hh1[symmetric] hh2[symmetric]
    by auto
  let ?dsc1 = sqrt (Re ((D1 − A1)^2 + 4 ∗ (B1 ∗ cnj B1))) and ?dsc2 = sqrt
(Re ((D2 − A2)^2 + 4 ∗ (B2 ∗ cnj B2)))
  let ?a11 = (A1 − D1 + cor ?dsc1) / (2 ∗ C1) and ?a12 = (A2 − D2 + cor
?dsc2) / (2 ∗ C2)
  let ?a21 = (A1 − D1 − cor ?dsc1) / (2 ∗ C1) and ?a22 = (A2 − D2 − cor
?dsc2) / (2 ∗ C2)
  let ?r11 = sqrt((4 − Re((−4 ∗ ?a11/B1) ∗ A1)) / (1 + Re (?a11∗cnj ?a11)))
  let ?r12 = sqrt((4 − Re((−4 ∗ ?a12/B2) ∗ A2)) / (1 + Re (?a12∗cnj ?a12)))
  let ?r21 = sqrt((4 − Re((−4 ∗ ?a21/B1) ∗ A1)) / (1 + Re (?a21∗cnj ?a21)))
  let ?r22 = sqrt((4 − Re((−4 ∗ ?a22/B2) ∗ A2)) / (1 + Re (?a22∗cnj ?a22)))

  have Re ((D2 − A2)^2 + 4 ∗ (B2 ∗ cnj B2)) = k^2 ∗ Re ((D1 − A1)^2 + 4 ∗
(B1 ∗ cnj B1))
    using ∗
    by (simp add: power2-eq-square field-simps)
  hence ?dsc2 = k ∗ ?dsc1
    using ‹k > 0›
    by (simp add: real-sqrt-mult)
  hence A2 − D2 + cor ?dsc2 = cor k ∗ (A1 − D1 + cor ?dsc1) A2 − D2 −
cor ?dsc2 = cor k ∗ (A1 − D1 − cor ?dsc1) 2∗C2 = cor k ∗ (2∗C1)
    using ∗
    by (auto simp add: field-simps)
  hence ?a12 = ?a11 ?a22 = ?a21
    using ‹k > 0›
    by simp-all
  moreover
  have Re((−4 ∗ ?a12/B2) ∗ A2) = Re((−4 ∗ ?a11/B1) ∗ A1)
    using ∗
    by (subst ‹?a12 = ?a11›) (simp, simp add: field-simps)
  have ?r12 = ?r11
    by (subst ‹Re((−4 ∗ ?a12/B2) ∗ A2) = Re((−4 ∗ ?a11/B1) ∗ A1)›, (subst
‹?a12 = ?a11›)+) simp
  moreover
  have Re((−4 ∗ ?a22/B2) ∗ A2) = Re((−4 ∗ ?a21/B1) ∗ A1)
    using ∗
    by (subst ‹?a22 = ?a21›) (simp, simp add: field-simps)
  have ?r22 = ?r21
    by (subst ‹Re((−4 ∗ ?a22/B2) ∗ A2) = Re((−4 ∗ ?a21/B1) ∗ A1)›, (subst
‹?a22 = ?a21›)+) simp
  moreover
  have chordal-circles-rep H1 = ((?a11, ?r11), (?a21, ?r21))
    using hh1[symmetric] hh2[symmetric]
    unfolding chordal-circles-rep-def Let-def
    by simp
  moreover
  have chordal-circles-rep H1 = ((?a11, ?r11), (?a21, ?r21))
    using hh1[symmetric]
    unfolding chordal-circles-rep-def Let-def
    by simp
  moreover
  have chordal-circles-rep H2 = ((?a12, ?r12), (?a22, ?r22))
    using hh2[symmetric]
    unfolding chordal-circles-rep-def Let-def
    by simp
  ultimately
  show chordal-circles-rep H1 = chordal-circles-rep H2
    by metis
qed

lemma chordal-circle-det-positive:
  fixes x y :: real
  assumes x ∗ y < 0
  shows x / (x − y) > 0
proof (cases x > 0)
  case True
  hence y < 0
    using ‹x ∗ y < 0›
    by (metis mult-less-cancel-left-pos mult-zero-right)
  have x − y > 0
    using ‹x > 0› ‹y < 0›
    by auto
  thus ?thesis
    using ‹x > 0›
    by (metis zero-less-divide-iff)
next
  case False
  hence y > 0 x < 0
    using ‹x ∗ y < 0›
   by − (metis mult-less-cancel-left-disj mult-zero-right, metis less-linear mult-zero-left)
  have x − y < 0
    using ‹x < 0› ‹y > 0›
    by auto
  thus ?thesis
    using ‹x < 0›
    by (metis zero-less-divide-iff)
qed
```

```
lemma chordal-circle1:
  assumes is-real A is-real D Re (A ∗ D) < 0 r = sqrt(Re ((4∗A)/(A−D)))
  shows mk-circline A 0 0 D = chordal-circle ∞_h r
using assms
proof transfer
  fix A D r
  assume ∗: is-real A is-real D Re (A ∗ D) < 0 r = sqrt (Re ((4∗A)/(A−D)))
  hence A ≠ 0 ∨ D ≠ 0
    by auto
  hence (A, 0, 0, D) ∈ {H. hermitean H ∧ H ≠ mat-zero}
    using eq-cnj-iff-real[of A] eq-cnj-iff-real[of D] ∗
    unfolding hermitean-def
    by (simp add: mat-adj-def mat-cnj-def)
  moreover
  have (− (cor r)^2, 0, 0, 4 − (cor r)^2) ∈ {H. hermitean H ∧ H ≠ mat-zero}
   by (simp add: hermitean-def mat-adj-def mat-cnj-def complex-cnj power2-eq-square)
  moreover
  have A ≠ D
    using ‹Re (A ∗ D) < 0› ‹is-real A› ‹is-real D›
    by auto
  have Re ((4∗A)/(A−D)) ≥ 0
  proof−
    have Re A / Re (A − D) ≥ 0
      using ‹Re (A ∗ D) < 0› ‹is-real A› ‹is-real D›
      using chordal-circle-det-positive[of Re A Re D]
      by simp
    thus ?thesis
      using ‹is-real A› ‹is-real D› ‹A ≠ D›
       by (subst Re-divide-real, auto) (metis mult-nonneg-nonpos zero-le-divide-iff
zero-le-mult-iff zero-le-numeral)
  qed
  moreover
  have − (cor (sqrt (Re (4 ∗ A / (A − D)))))^2 = cor (Re (4 / (D − A))) ∗ A
    using ‹Re ((4∗A)/(A−D)) ≥ 0› ‹is-real A› ‹is-real D› ‹A ≠ D›
      by (subst cor-squared, subst real-sqrt-power[symmetric], simp) (simp add:
Re-divide-real Re-mult-real complex-of-real-Re of-real-numeral minus-divide-right)
  moreover
  have 4 ∗ (A − D) − 4 ∗ A  = 4 ∗ −D
    by (simp add: field-simps)
  hence 4 − 4 ∗ A / (A − D) = −4 ∗ D / (A − D)
    using ‹A ≠ D›
   by (smt ab-semigroup-mult-class.mult-ac(1) diff-divide-eq-iff eq-iff-diff-eq-0 mult-minus1
mult-minus1-right mult-numeral-1-right neg-numeral-def right-diff-distrib-numeral
times-divide-eq-right)
  hence 4 − 4 ∗ A / (A − D) = 4 ∗ D / (D − A)
   by (metis (hide-lams, no-types) minus-diff-eq minus-divide-left minus-divide-right
minus-mult-left neg-numeral-def)
  hence 4 − (cor (sqrt (Re (4 ∗ A / (A − D)))))^2 = cor (Re (4 / (D − A))) ∗
D
```

using ‹*Re* ((*4*∗*A*)/(*A*−*D*)) ≥ *0*› ‹*is-real A*› ‹*is-real D*› ‹*A* ≠ *D*›
by (*subst cor-squared*, *subst real-sqrt-power*[*symmetric*], *simp*) (*simp add*: *Re-divide-real Re-mult-real complex-of-real-Re of-real-numeral*)
ultimately
show *circline-mat-eq* (*mk-circline-rep A 0 0 D*) (*chordal-circle-rep inf-homo-rep r*)
using ‹*is-real A*› ‹*is-real D*› ‹*A* ≠ *D*› ‹*r* = *sqrt*(*Re* ((*4*∗*A*)/(*A*−*D*)))›
by (*simp add*: *chordal-circle-rep-def mk-circline-rep-def Abs-circline-mat-inverse*) (*rule-tac x*=*Re*(*4*/(*D*−*A*)) **in** *exI*, *auto*)
qed

lemma *chordal-circle2*:
assumes *is-real A is-real D Re* (*A* ∗ *D*) < *0 r* = *sqrt*(*Re* ((*4*∗*D*)/(*D*−*A*)))
shows *mk-circline A 0 0 D* = *chordal-circle 0*$_h$ *r*
using *assms*
proof *transfer*
fix *A D r*
assume ∗: *is-real A is-real D Re* (*A* ∗ *D*) < *0 r* = *sqrt* (*Re* ((*4*∗*D*)/(*D*−*A*)))
hence *A* ≠ *0* ∨ *D* ≠ *0*
by *auto*
hence (*A*, *0*, *0*, *D*) ∈ {*H*. *hermitean H* ∧ *H* ≠ *mat-zero*}
using *eq-cnj-iff-real*[*of A*] *eq-cnj-iff-real*[*of D*] ∗
unfolding *hermitean-def*
by (*simp add*: *mat-adj-def mat-cnj-def*)
moreover
have (*4* − (*cor r*)^2, *0*, *0*, − (*cor r*)^2) ∈ {*H*. *hermitean H* ∧ *H* ≠ *mat-zero*}
by (*simp add*: *hermitean-def mat-adj-def mat-cnj-def complex-cnj power2-eq-square*) (*metis mult-zero-right of-real-0 zero-neq-numeral*)
moreover
have *A* ≠ *D*
using ‹*Re* (*A* ∗ *D*) < *0*› ‹*is-real A*› ‹*is-real D*›
by *auto*
have *Re*((*4*∗*D*)/(*D*−*A*)) ≥ *0*
proof−
have *Re D* / *Re* (*D* − *A*) ≥ *0*
using ‹*Re* (*A* ∗ *D*) < *0*› ‹*is-real A*› ‹*is-real D*›
using *chordal-circle-det-positive*[*of Re D Re A*]
by (*simp add*: *field-simps*)
thus *?thesis*
using ‹*is-real A*› ‹*is-real D*› ‹*A* ≠ *D*›
by (*subst Re-divide-real*, *auto*) (*metis mult-nonneg-nonpos zero-le-divide-iff zero-le-mult-iff zero-le-numeral*)
qed
have *4* ∗ (*D* − *A*) − *4* ∗ *D* = *4* ∗ −*A*
by (*simp add*: *field-simps*)
hence *4* − *4* ∗ *D* / (*D* − *A*) = −*4* ∗ *A* / (*D* − *A*)
using ‹*A* ≠ *D*›
by (*smt ab-semigroup-mult-class.mult-ac*(*1*) *diff-divide-eq-iff eq-iff-diff-eq-0 mult-minus1 mult-minus1-right mult-numeral-1-right neg-numeral-def right-diff-distrib-numeral*

times-divide-eq-right)
hence $4 - 4 * D / (D - A) = 4 * A / (A - D)$
by (*metis* (*hide-lams*, *no-types*) *minus-diff-eq minus-divide-left minus-divide-right minus-mult-left neg-numeral-def*)
hence $4 - (cor\ (sqrt\ (Re\ ((4*D)/(D-A)))))^2 = cor\ (Re\ (4\ /\ (A - D))) * A$
using ‹*is-real A*› ‹*is-real D*› ‹$A \neq D$› ‹$Re\ (4 * D / (D - A)) \geq 0$›
by (*subst cor-squared*, *subst real-sqrt-power*[*symmetric*], *simp*) (*simp add*: *Re-divide-real complex-of-real-Re of-real-numeral*)

moreover
have $-\ (cor\ (sqrt\ (Re\ ((4*D)/(D-A)))))^2 = cor\ (Re\ (4\ /\ (A - D))) * D$
using ‹*is-real A*› ‹*is-real D*› ‹$A \neq D$› ‹$Re\ ((4*D)/(D-A)) \geq 0$›
by (*subst cor-squared*, *subst real-sqrt-power*[*symmetric*], *simp*) (*simp add*: *Re-divide-real complex-of-real-Re of-real-numeral minus-divide-right*)

ultimately
show *circline-mat-eq* (*mk-circline-rep A 0 0 D*) (*chordal-circle-rep zero-homo-rep r*)
using ‹*is-real A*› ‹*is-real D*› ‹$A \neq 0 \lor D \neq 0$› ‹$r = sqrt\ (Re\ ((4*D)/(D-A)))$›
by (*simp add*: *chordal-circle-rep-def mk-circline-rep-def Abs-circline-mat-inverse*) (*rule-tac* $x{=}Re\ (4/(A-D))$ **in** *exI*, *auto*)
qed

lemma *chordal-circle'*:
assumes $B \neq 0$ $(A, B, C, D) \in \{H.\ hermitean\ H \land H \neq mat\text{-}zero\}$ $Re\ (mat\text{-}det\ (A, B, C, D)) \leq 0$
$C * a^2 + (D - A) * a - B = 0$ $r = sqrt((4 - Re((-4 * a/B) * A))\ /\ (1 + Re\ (a*cnj\ a)))$
shows *mk-circline A B C D* = *chordal-circle* (*of-complex a*) *r*
using *assms*
proof *transfer*
fix *A B C D a* :: *complex* **and** *r* :: *real*

let $?k = -4*a\ /\ B$

assume ∗: $(A, B, C, D) \in \{H.\ hermitean\ H \land H \neq mat\text{-}zero\}$ **and** ∗∗: $B \neq 0$ $C * a^2 + (D - A) * a - B = 0$ **and** *rr*: $r = sqrt\ ((4 - Re\ (?k * A))\ /\ (1 + Re\ (a * cnj\ a)))$ **and** *det*: $Re\ (mat\text{-}det\ (A, B, C, D)) \leq 0$

have *is-real A is-real D* $C = cnj\ B$
using ∗ *hermitean-elems*
by *auto*

from ∗∗ **have** *a12*: *let dsc* = $sqrt(Re\ ((D-A)^2 + 4 * (B*cnj\ B)))$
in $a = (A - D + cor\ dsc)\ /\ (2 * C) \lor a = (A - D - cor\ dsc)\ /\ (2 * C)$
proof−
have $Re\ ((D-A)^2 + 4 * (B*cnj\ B)) \geq 0$
using ‹*is-real A*› ‹*is-real D*›

by (*subst complex-mult-cnj-cmod*) (*simp add*: *power2-eq-square*)
hence $csqrt\ ((D - A)^2 - 4 * C * - B) = cor\ (sqrt\ (Re\ ((D - A)^2 + 4 * (B * cnj\ B))))$
using $csqrt\text{-}real[of\ ((D - A)^2 + 4 * (B * cnj\ B))]$ ‹*is-real A*› ‹*is-real D*› ‹*C = cnj B*›
by (*auto simp add*: *power2-eq-square field-simps*)
thus *?thesis*
using $complex\text{-}quadratic\text{-}equation\text{-}full[of\ C\ a\ D - A\ {-B}]$ ‹$C * a^2 + (D - A) * a - B = 0$› ‹*B* ≠ *0*› ‹*C = cnj B*›
by (*simp add*: *Let-def*)
qed

have *is-real ?k*
using *a12* ‹*C = cnj B*› ‹*is-real A*› ‹*is-real D*›
by (*auto simp add*: *Let-def div-reals*)
have *a* ≠ *0*
using ∗∗
by *auto*
hence *Re ?k* ≠ *0*
using ‹*is-real* (−*4*∗*a* / *B*)› ‹*B* ≠ *0*›
by (*metis complex-surj complex-zero-def mult-eq-0-iff nonzero-divide-eq-eq zero-neq-neg-numeral*)

moreover
have −*4* ∗ *a* = *cor* (*Re ?k*) ∗ *B*
using *complex-of-real-Re*[*OF* ‹*is-real* (−*4*∗*a*/*B*)›] ‹*B* ≠ *0*›
by *simp*
moreover
have *is-real* (*a*/*B*)
using ‹*is-real ?k*›
by (*metis Im-mult-real complex-Im-neg-numeral complex-Re-neg-numeral mult-eq-0-iff times-divide-eq-right zero-neq-neg-numeral*)
hence *is-real* (*B* ∗ *cnj a*)
by (*smt mult.commute complex-In-mult-cnj-zero complex-cnj-divide complex-cnj-zero-iff eq-cnj-iff-real eq-divide-eq times-divide-eq-right*)
hence *B* ∗ *cnj a* = *cnj B* ∗ *a*
using *eq-cnj-iff-real*[*of B* ∗ *cnj a*]
by (*simp add*: *complex-cnj*)
hence −*4* ∗ *cnj a* = *cor* (*Re ?k*) ∗ *C*
using ‹*C = cnj B*›
using *complex-of-real-Re*[*OF* ‹*is-real ?k*›] ‹*B* ≠ *0*›
by (*simp*, *simp add*: *field-simps*)
moreover
have *1* + *a* ∗ *cnj a* ≠ *0*
by (*subst complex-mult-cnj-cmod*) (*smt cor-add of-real-0 of-real-1 of-real-eq-iff realpow-square-minus-le*)
have $r^2 = (4 - Re\ (?k * A)) / (1 + Re\ (a * cnj\ a))$
proof−
have *Re* (*a* / *B* ∗ *A*) ≥ −*1*
using *a12 chordal-circle-radius-positive*[*of A B C D*] ∗ ‹*B* ≠ *0*› *det*

```
      by (auto simp add: Let-def field-simps)
    from mult-right-mono-neg[OF this, of −4]
    have 4 − Re (?k ∗ A) ≥ 0
      using Re-mult-real[of −4 a / B ∗ A]
      by (simp add: field-simps)
    moreover
    have 1 + Re (a ∗ cnj a) > 0
     by (subst complex-mult-cnj-cmod) (smt Re-complex-of-real ‹a ≠ 0› norm-eq-zero
zero-less-power2)
    ultimately
    have (4 − Re (?k ∗ A)) / (1 + Re (a ∗ cnj a)) ≥ 0
      by (metis divide-nonneg-pos)
    thus ?thesis
      using rr
      by simp
  qed
  hence r^2 = Re ((4 − ?k ∗ A) / (1 + a ∗ cnj a))
    using ‹is-real ?k› ‹is-real A› ‹1 + a ∗ cnj a ≠ 0›
    by (subst Re-divide-real, auto)
  hence (cor r)^2 =  (4 − ?k ∗ A) / (1 + a ∗ cnj a)
    using ‹is-real ?k› ‹is-real A›
    using mult-reals[of ?k A]
    by (simp add: cor-squared) (subst complex-of-real-Re, subst div-reals, auto)
  hence 4 − (cor r)^2 ∗ (a ∗ cnj a + 1) = cor (Re ?k) ∗ A
    using complex-of-real-Re[OF ‹is-real (−4∗a/B)›]
    using ‹1 + a ∗ cnj a ≠ 0›
    by (simp add: field-simps)
  moreover

  have ?k = cnj ?k
    using ‹is-real ?k›
    by (subst eq-cnj-iff-real) simp
  have ?k^2 = cor ((cmod ?k)^2)
    using  cor-cmod-real[OF ‹is-real ?k›]
    unfolding power2-eq-square
    by (subst cor-mult) (metis minus-mult-minus)
  hence ?k^2 = ?k ∗ cnj ?k
    using complex-mult-cnj-cmod[of ?k]
    by simp
  hence ∗∗∗: a ∗ cnj a = (cor ((Re ?k)^2) ∗ B ∗ C) / 16
   using complex-of-real-Re[OF ‹is-real (−4∗a/B)›] ‹C = cnj B› ‹is-real (−4∗a/B)›
‹B ≠ 0›
    by (simp add: complex-cnj)
  from ∗∗ have cor ((Re ?k)^2) ∗ B ∗ C − 4 ∗ cor (Re ?k) ∗ (D−A) − 16 = 0
    using complex-of-real-Re[OF ‹is-real ?k›]
    by (simp add: power2-eq-square, simp add: field-simps, algebra)
  hence ?k ∗ (D−A) = 4 ∗ (cor ((Re ?k)^2) ∗ B ∗ C / 16 − 1)
    by (subst (asm) complex-of-real-Re[OF ‹is-real ?k›]) algebra
  hence ?k ∗ (D−A) = 4 ∗ (a∗cnj a − 1)
```

by (*subst* (*asm*) ∗∗∗[*symmetric*]) *simp*

hence *4* ∗ *a* ∗ *cnj a* − (*cor r*)^2 ∗ (*a* ∗ *cnj a* + *1*) = *cor* (*Re ?k*) ∗ *D*
using ‹*4* − (*cor r*)^2 ∗ (*a* ∗ *cnj a* + *1*) = *cor* (*Re ?k*) ∗ *A*›
using *complex-of-real-Re*[*OF* ‹*is-real* (−*4*∗*a*/*B*)›]
by *simp algebra*
ultimately
show *circline-mat-eq* (*mk-circline-rep A B C D*) (*chordal-circle-rep* (*of-complex-coords a*) *r*)
using ∗ ‹*a* ≠ *0*›
by (*simp add*: *mk-circline-rep-def Abs-circline-mat-inverse*) (*rule-tac x*=*Re* (−*4*∗*a* / *B*) **in** *exI*, *simp*)
qed

lift-definition *o-circline-point-0h* :: *ocircline* **is** *circline-point-0h-rep*
done

lemma *of-ocircline-o-circline-point-0h* [*simp*]: *of-ocircline o-circline-point-0h* = *circline-point-0h*
by (*metis circline-point-0h-def o-circline-point-0h-def of-ocircline.abs-eq*)

lemma *ocircline-set-0h*:
assumes *ocircline-set H* = {0_h}
shows *H* = *o-circline-point-0h* ∨ *H* = *opposite-ocircline* (*o-circline-point-0h*)
proof−
have *of-ocircline H* = *circline-point-0h*
using *assms*
using *circline-set-ocircline-set*[*of H*, *symmetric*]
using *unique-circline-type-zero-0h′ card-eq1-circline-type-zero*[*of of-ocircline H*]
by *blast*
thus *?thesis*
by (*metis inj-of-ocircline of-ocircline-o-circline-point-0h*)
qed

11.13 Disc automorphisms

lemma *circline-set-fix-iff-circline-fix*:
assumes *circline-set H′* ≠ {}
shows (*moebius-pt M*) ‘ (*circline-set H*) = *circline-set H′* ⟷ *moebius-circline M H* = *H′*
using *assms*
by (*subst moebius-circline-set*, *auto*) (*rule inj-circline-set*[*of - H′*], *auto*)

lemma *ocircline-set-fix-iff-ocircline-fix*:
assumes *ocircline-set H′* ≠ {}
shows (*moebius-pt M*) ‘ (*ocircline-set H*) = *ocircline-set H′* ⟷
moebius-ocircline M H = *H′* ∨ *moebius-ocircline M H* = *opposite-ocircline H′*
using *assms inj-ocircline-set*[*of - H′*]
by (*subst moebius-ocircline-set*, *auto*)

definition *Unitary11-gen-rep* **where**
Unitary11-gen-rep M ⟷ *unitary11-gen* (*Rep-moebius-mat M*)

lift-definition *Unitary11-gen* :: *moebius* ⇒ *bool* **is** *Unitary11-gen-rep*
apply (*auto simp add*: *Unitary11-gen-rep-def*)
apply (*simp add*: *unitary11-gen-mult-sm*)
apply (*simp add*: *unitary11-gen-div-sm*)
done

lemma *unit-circle-fix-iff-Unitary11-gen*:
  shows *moebius-circline M unit-circle* = *unit-circle* ⟷ *Unitary11-gen M* (**is** *?lhs* = *?rhs*)
proof
  assume *?lhs*
  thus *?rhs*
  proof (*transfer*)
    fix *M*
    assume *circline-mat-eq* (*moebius-circline-rep M unit-circle-rep*) *unit-circle-rep*
    then obtain *k* **where** *k* ≠ *0* (*1*, *0*, *0*, −*1*) = *cor k* $*_{sm}$ *congruence* (*mat-inv* (*Rep-moebius-mat M*)) (*1*, *0*, *0*, −*1*)
      by *auto*
    hence (*1*/*cor k*, *0*, *0*, −*1*/*cor k*) = *congruence* (*mat-inv* (*Rep-moebius-mat M*)) (*1*, *0*, *0*, −*1*)
      using *mult-sm-inv-l*[*of cor k congruence* (*mat-inv* (*Rep-moebius-mat M*)) (*1*, *0*, *0*, −*1*)]
      by *simp*
    hence *congruence* (*Rep-moebius-mat M*) (*1*/*cor k*, *0*, *0*, −*1*/*cor k*) = (*1*, *0*, *0*, −*1*)
      using *Rep-moebius-mat*[*of M*] *mat-det-inv*[*of Rep-moebius-mat M*]
      using *congruence-inv*[*of mat-inv* (*Rep-moebius-mat M*) (*1*, *0*, *0*, −*1*) (*1*/*cor k*, *0*, *0*, −*1*/*cor k*)]
      by *simp*
    hence *congruence* (*Rep-moebius-mat M*) (*1*, *0*, *0*, −*1*) = *cor k* $*_{sm}$ (*1*, *0*, *0*, −*1*)
      using *congruence-scale-m*[*of Rep-moebius-mat M 1*/*cor k* (*1*, *0*, *0*, −*1*)]
      using *mult-sm-inv-l*[*of 1*/ *cor k congruence* (*Rep-moebius-mat M*) (*1*, *0*, *0*, −*1*) (*1*, *0*, *0*, −*1*)] ‹*k* ≠ *0*›
      by *simp*
    thus *Unitary11-gen-rep M*
      using ‹*k* ≠ *0*›
      unfolding *Unitary11-gen-rep-def unitary11-gen-def*
      by *simp*
  qed
next
  assume *?rhs*
  thus *?lhs*
  proof (*transfer*)

```
    fix M
    assume Unitary11-gen-rep M
    hence unitary11-gen (mat-inv (Rep-moebius-mat M))
      using Rep-moebius-mat[of M]
      using unitary11-gen-mat-inv
      by (simp add: Unitary11-gen-rep-def)
    thus circline-mat-eq (moebius-circline-rep M unit-circle-rep) unit-circle-rep
      unfolding unitary11-gen-real
      by auto (rule-tac x=1/k in exI, simp)
  qed
qed

lemma unit-circle-set-fix-iff-Unitary11-gen:
  shows (moebius-pt M ` (circline-set unit-circle) = (circline-set unit-circle)) ⟷
Unitary11-gen M (is ?lhs ⟷ ?rhs)
using unit-circle-fix-iff-Unitary11-gen[of M] circline-set-fix-iff-circline-fix[of unit-circle
M unit-circle]
using one-on-unit-circle
by auto

definition Unitary11-gen-direct-rep where
  Unitary11-gen-direct-rep M ⟷
     (let (A, B, C, D) = Rep-moebius-mat M
       in unitary11-gen (A, B, C, D) ∧ (B = 0 ∨ Re ((A*D)/(B*C)) > 1))

lift-definition Unitary11-gen-direct :: moebius ⇒ bool is Unitary11-gen-direct-rep
proof−
  fix M M′
  let ?M = Rep-moebius-mat M and ?M′ = Rep-moebius-mat M′
  assume moebius-mat-eq M M′
  then obtain k where *: k ≠ 0 Rep-moebius-mat M′ = k *sm Rep-moebius-mat
M
    by auto
 hence **: unitary11-gen (Rep-moebius-mat M) ⟷ unitary11-gen (Rep-moebius-mat
M′)
    using unitary11-gen-mult-sm[of k ?M] unitary11-gen-div-sm[of k ?M]
    by auto
  obtain A B C D where MM: (A, B, C, D) = Rep-moebius-mat M
    by (cases Rep-moebius-mat M) auto
  obtain A′ B′ C′ D′ where MM′: (A′, B′, C′, D′) = Rep-moebius-mat M′
    by (cases Rep-moebius-mat M′) auto

  show Unitary11-gen-direct-rep M = Unitary11-gen-direct-rep M′
    using * ** MM MM′
    unfolding Unitary11-gen-direct-rep-def Let-def
    by auto
qed

lemma ounit-circle-fix-iff-Unitary11-gen-direct:
  shows moebius-ocircline M ounit-circle = ounit-circle ⟷ Unitary11-gen-direct M (is ?lhs ⟷ ?rhs)
proof
  assume *: ?lhs
  have moebius-circline M unit-circle = unit-circle
    apply (subst moebius-circline-ocircline[of M unit-circle])
    apply (subst of-circline-unit-circline)
    apply (subst *)
    by simp

  hence Unitary11-gen M
    by (simp add: unit-circle-fix-iff-Unitary11-gen)
  thus ?rhs
    using *
  proof (transfer)
    fix M
    let ?M = Rep-moebius-mat M
    let ?H = (1, 0, 0, −1)
    obtain A B C D where MM: (A, B, C, D) = ?M
      by (cases ?M) auto
   assume Unitary11-gen-rep M ocircline-mat-eq (moebius-circline-rep M unit-circle-rep) unit-circle-rep
    then obtain k where 0 < k ?H = cor k *_sm congruence (mat-inv ?M) ?H
      by auto
    hence congruence ?M ?H = cor k *_sm ?H
    using congruence-inv[of mat-inv ?M ?H (1/cor k) *_sm ?H] Rep-moebius-mat[of M]
      using mult-sm-inv-l[of cor k congruence (mat-inv ?M) ?H ?H]
      using mult-sm-inv-l[of 1/cor k congruence ?M ?H]
      using congruence-scale-m[of ?M 1/cor k ?H]
      by (auto simp add: mat-det-inv)
    then obtain a b k' where k' ≠ 0 ?M = k' *_sm (a, b, cnj b, cnj a) sgn (Re (mat-det (a, b, cnj b, cnj a))) = 1
      using unitary11-sgn-det-orientation'[of ?M k] ‹k > 0›
      by auto
    moreover
    have mat-det (a, b, cnj b, cnj a) ≠ 0
      using ‹sgn (Re (mat-det (a, b, cnj b, cnj a))) = 1›
      by (metis complex-Re-zero sgn-zero zero-neq-one)
    ultimately
    show Unitary11-gen-direct-rep M
      using unitary11-sgn-det[of k' a b ?M A B C D]
      using MM[symmetric] ‹k > 0› ‹Unitary11-gen-rep M›
      by (simp add: Unitary11-gen-rep-def Unitary11-gen-direct-rep-def sgn-1-pos split: split-if-asm)
  qed
next
  assume ?rhs
  thus ?lhs
  proof (transfer)
    fix M
    let ?M = Rep-moebius-mat M
    obtain A B C D where MM: (A, B, C, D) = ?M
      by (cases ?M) auto
    assume Unitary11-gen-direct-rep M
    hence unitary11-gen ?M B = 0 ∨ 1 < Re (A ∗ D / (B ∗ C))
      using MM[symmetric]
      by (auto simp add: Unitary11-gen-direct-rep-def)
    have sgn (if B = 0 then 1 else sgn (Re (A ∗ D / (B ∗ C)) − 1)) = 1
      using ‹B = 0 ∨ 1 < Re (A ∗ D / (B ∗ C))›
      by auto
   then obtain k' where k' > 0 congruence (Rep-moebius-mat M) (1, 0, 0, −1)
= cor k' ∗sm (1, 0, 0, −1)
      using unitary11-orientation[OF ‹unitary11-gen ?M› MM[symmetric]]
      by (auto simp add: sgn-1-pos)
    thus ocircline-mat-eq (moebius-circline-rep M unit-circle-rep) unit-circle-rep
        using congruence-inv[of ?M (1, 0, 0, −1) cor k' ∗sm (1, 0, 0, −1)]
Rep-moebius-mat[of M]
      using congruence-scale-m[of mat-inv ?M cor k' (1, 0, 0, −1)]
      by auto
  qed
qed
```

Blaschke factor

```
definition blaschke-rep where
  blaschke-rep a = Abs-moebius-mat (1, −a, −cnj a, 1)

lemma blaschke-rep-Rep1:
  assumes cmod a ≠ 1
  shows Rep-moebius-mat (blaschke-rep a) = (1, −a, −cnj a, 1)
  using assms
  by (simp add: blaschke-rep-def Abs-moebius-mat-inverse)

lemma blaschke-rep-Rep2:
  assumes a ∗ cnj a ≠ 1
  shows Rep-moebius-mat (blaschke-rep a) = (1, −a, −cnj a, 1)
  using assms
  by (simp add: blaschke-rep-def Abs-moebius-mat-inverse)

lift-definition blaschke :: complex ⇒ moebius is blaschke-rep
by (simp del: moebius-mat-eq-def)

lemma blaschke-a-to-zero:
  assumes cmod a ≠ 1
  shows moebius-pt (blaschke a) (of-complex a) = 0h
proof−
  from assms have a ∗ cnj a ≠ 1
    by simp
  thus ?thesis
     by (transfer) (simp add: blaschke-rep-Rep2, rule-tac x=1/(1 − a∗cnj a) in
exI, simp add: field-simps)
qed

lemma blaschke-inv-a-inf:
  assumes cmod a ≠ 1
  shows moebius-pt (blaschke a) (inversion-homo (of-complex a)) = ∞_h
proof−
  from assms have a ∗ cnj a ≠ 1
    by simp
  thus ?thesis
    unfolding inversion-homo-def
     by (transfer) (simp add: blaschke-rep-Rep2 vec-cnj-def, rule-tac x=1/(1 −
a∗cnj a) in exI, simp add: field-simps)
qed

lemma blaschke-Unitary11-gen-rep:
  assumes a ∗ cnj a ≠ 1
  shows Unitary11-gen-rep (blaschke-rep a)
proof−
  have is-real (1 − a∗cnj a)
    by auto
  moreover
  hence cor (Re (1 − a∗cnj a)) = 1 − a∗cnj a
    by (rule complex-of-real-Re)
  moreover
  have Re (a∗cnj a) ≠ 1
    using ‹is-real (1 − a∗cnj a)› assms
    by (metis complex-In-mult-cnj-zero complex-of-real-Re of-real-1)
  ultimately
  show ?thesis
    using assms
    using blaschke-rep-Rep2
     by (auto simp add: blaschke-rep-def Unitary11-gen-rep-def unitary11-gen-real
mat-adj-def mat-cnj-def complex-cnj field-simps simp del: complex-Re-mult) (rule-tac
x=Re (1 − a∗cnj a) in exI, simp del: complex-Re-mult)
qed

lemma blaschke-unitary11-gen-direct-rep:
  assumes Re (a ∗ cnj a) < 1
  shows Unitary11-gen-direct-rep (blaschke-rep a)
proof−
  have a ∗ cnj a ≠ 1
    using assms
    by (cases a, simp)
  show ?thesis
  proof (cases a = 0)
    case True
    thus ?thesis
      using blaschke-Unitary11-gen-rep[of a]
     by (simp add: Unitary11-gen-direct-rep-def Unitary11-gen-rep-def blaschke-rep-def)
  next
    case False
    hence Re (a ∗ cnj a) > 0
     by (subst complex-mult-cnj-cmod) (metis Re-complex-of-real zero-less-norm-iff
zero-less-power)
    thus ?thesis
      using assms ‹a ∗ cnj a ≠ 1› ‹a ≠ 0›
     using blaschke-Unitary11-gen-rep[of a] blaschke-rep-Rep2[of a] Re-divide-real[of
a∗cnj a 1]
     by (auto simp add: Unitary11-gen-direct-rep-def blaschke-rep-def Unitary11-gen-rep-def
simp del: complex-Re-mult)
  qed
qed

lemma blaschke-Unitary11-gen:
  assumes a ∗ cnj a ≠ 1
  shows Unitary11-gen (blaschke a)
using assms
by (transfer) (rule blaschke-Unitary11-gen-rep)

lemma blaschke-Unitary11-gen-direct:
  assumes Re (a ∗ cnj a) < 1
  shows Unitary11-gen-direct (blaschke a)
using assms
by transfer (simp add: blaschke-unitary11-gen-direct-rep)

lemma blaschke-unit-circle-fix:
  assumes cmod a ≠ 1
  shows moebius-circline (blaschke a) unit-circle = unit-circle
  using assms
  using blaschke-Unitary11-gen  unit-circle-fix-iff-Unitary11-gen
  by simp

lemma blaschke-ounit-circle-fix:
  assumes cmod a < 1
  shows moebius-ocircline (blaschke a) ounit-circle = ounit-circle
proof−
  have Re (a ∗ cnj a) < 1
    using assms
    by (metis complex-mod-sqrt-Re-mult-cnj real-sqrt-lt-1-iff)
  thus ?thesis
    using assms
    using blaschke-Unitary11-gen-direct  ounit-circle-fix-iff-Unitary11-gen-direct
    by simp
qed
```

lemma [*simp*]: *hermitean* (*1*, *0*, *0*, −*1*)
by (*auto simp add*: *hermitean-def mat-adj-def mat-cnj-def*)

definition *is-disc-aut* **where** *is-disc-aut f* ⟷ *bij-betw f unit-disc unit-disc*

lemma *is-disc-aut-iff-unit-disc-fix*:
  shows *is-disc-aut* (*moebius-pt M*) ⟷ (*moebius-pt M*) ‘ *unit-disc* = *unit-disc*
using *bij-moebius-pt*[*of M*]
unfolding *is-disc-aut-def is-moebius-def*
unfolding *bij-betw-def*
by *auto* (*metis injD inj-onI*)

lemma *comp-inv-l*:
  assumes *f* ∘ *inv g* = *h bij g*
  shows *f* = *h* ∘ *g*
using *assms*
by (*metis bij-def o-inv-o-cancel*)

lemma *in-unit-disc-cmod-lt-1*:
  assumes *of-complex a* ∈ *unit-disc*
  shows *cmod a* < *1*
using *assms*
unfolding *unit-disc-def disc-def*
apply *auto*
proof (*transfer*)
  fix *a*
  assume *in-ocircline-rep unit-circle-rep* (*of-complex-coords a*)
  hence *Re a* ∗ *Re a* + *Im a* ∗ *Im a* < *1*
    by (*simp add*: *in-ocircline-rep-def Let-def vec-cnj-def*)
  hence (*cmod a*)^2 < *1*
    unfolding *cmod-def*
    by (*simp*, *simp add*: *power2-eq-square*)
  thus *cmod a* < *1*
    by (*metis less-1-mult not-less-iff-gr-or-eq one-power2 power2-eq-square*)
qed

11.14 Angle between circlines

fun *mat-det-12* :: *complex-mat* ⇒ *complex-mat* ⇒ *complex* **where**
 mat-det-12 (*A1*, *B1*, *C1*, *D1*) (*A2*, *B2*, *C2*, *D2*) = *A1*∗*D2* + *A2*∗*D1* − *B1*∗*C2* − *B2*∗*C1*

lemma *mat-det-12-mm-l* [*simp*]: *mat-det-12* (*M* $*_{mm}$ *A*) (*M* $*_{mm}$ *B*) = *mat-det M* ∗ *mat-det-12 A B*
by (*cases M*, *cases A*, *cases B*) (*simp add*: *field-simps*)

lemma *mat-det-12-mm-r* [*simp*]: *mat-det-12* (*A* $*_{mm}$ *M*) (*B* $*_{mm}$ *M*) = *mat-det M* ∗ *mat-det-12 A B*

by (*cases M*, *cases A*, *cases B*) (*simp add*: *field-simps*)

lemma *mat-det-12-sm-l* [*simp*]: *mat-det-12* ($k *_{sm}$ *A*) *B* = *k* ∗ *mat-det-12 A B*
by (*cases A*, *cases B*) (*simp add*: *field-simps*)

lemma *mat-det-12-sm-r* [*simp*]: *mat-det-12 A* ($k *_{sm}$ *B*) = *k* ∗ *mat-det-12 A B*
by (*cases A*, *cases B*) (*simp add*: *field-simps*)

lemma *mat-det-12-congruence* [*simp*]:
mat-det-12 (*congruence M A*) (*congruence M B*) = (*cor* ((*cmod* (*mat-det M*))^2))
∗ *mat-det-12 A B*
by ((*subst mult-mm-assoc*[*symmetric*])+, *subst mat-det-12-mm-l*, *subst mat-det-12-mm-r*, *subst mat-det-adj*) (*auto simp add*: *field-simps complex-mult-cnj-cmod*)

lemma *mat-det-congruence* [*simp*]:
mat-det (*congruence M A*) = (*cor* ((*cmod* (*mat-det M*))^2)) ∗ *mat-det A*
by (*simp add*: *mat-det-adj complex-mult-cnj-cmod field-simps*)

definition *cos-angle-rep* **where**
cos-angle-rep H1 H2 =
(*let H1* = *Rep-circline-mat H1*;
H2 = *Rep-circline-mat H2 in*
− *Re* (*mat-det-12 H1 H2*) / (*2* ∗ (*sqrt* (*Re* (*mat-det H1* ∗ *mat-det H2*)))))

lemma [*simp*]: *is-real* (*mat-det* (*Rep-circline-mat H*))
using *Rep-circline-mat*[*of H*]
by (*simp add*: *mat-det-hermitean-real*)

lift-definition *cos-angle* :: *ocircline* ⇒ *ocircline* ⇒ *real* **is** *cos-angle-rep*
by (*auto simp add*: *cos-angle-rep-def Let-def real-sqrt-mult*)

lemma *ang-vec-opposite-opposite′*:
assumes *a1* ≠ *E a2* ≠ *E*
shows (*E* − *a1*) (*E* − *a2*) = (*a1* − *E*) (*a2* − *E*)
using *ang-vec-opposite-opposite*[*of E* − *a1 E* − *a2*] *assms*
by (*simp add*: *field-simps del*: *ang-vec-def*)

lemma *cos-ang-circ-simp*:
assumes *E* ≠ *μ1 E* ≠ *μ2*
shows*cos* (*ang-circ E μ1 μ2 p1 p2*) = *sgn-bool* (*p1* = *p2*) ∗ *cos* (*arg* (*E* − *μ2*)
− *arg* (*E* − *μ1*))
using *assms*
using *cos-periodic-pi2*[*of arg* (*E* − *μ2*) − *arg* (*E* − *μ1*)]
using *cos-periodic-pi3*[*of arg* (*E* − *μ2*) − *arg* (*E* − *μ1*)]
using *ang-circ-simp*[*OF assms*, *of p1 p2*]
by *auto* (*auto simp add*: *field-simps*)

lemma *Re-sgn*:
assumes *is-real A A* ≠ *0*

```
  shows Re (sgn A) = sgn-bool (Re A > 0)
using assms
by (cases A) simp

lemma Re-mult-real3:
  assumes is-real z1 is-real z2 is-real z3
  shows Re (z1 * z2 * z3) = Re z1 * Re z2 * Re z3
using assms
by (metis Re-mult-real mult-reals)

lemma [simp]: sgn (sqrt x) = sgn x
by (smt real-sqrt-eq-zero-cancel-iff real-sqrt-lt-0-iff sgn-real-def)

lemma sgn-divide:
  fixes x y :: real
  shows sgn (x / y) = sgn x / sgn y
by (metis divide-inverse inverse-sgn real-scaleR-def sgn-scaleR)

lemma real-circle-sgn-r:
  assumes is-circle H (a, r) = euclidean-circle H
  shows sgn r = − circline-type H
using assms
proof transfer
  fix H a r
  obtain A B C D where HH: Rep-circline-mat H = (A, B, C, D)
    by (cases Rep-circline-mat H) auto
  hence is-real A is-real D
    using hermitean-elems Rep-circline-mat[of H]
    by auto
  assume ¬ circline-A0-rep H (a, r) = euclidean-circle-rep H
  hence A ≠ 0
    using ⟨¬ circline-A0-rep H⟩ HH
    by (simp add: circline-A0-rep-def)
  hence Re A * Re A > 0
    using ⟨is-real A⟩
   by (metis complex-Im-zero complex-Re-zero complex-equality not-real-square-gt-zero)

  thus sgn r = − circline-type-rep H
    using HH ⟨(a, r) = euclidean-circle-rep H⟩ ⟨is-real A⟩ ⟨is-real D⟩ ⟨A ≠ 0⟩
   by (simp add: euclidean-circle-rep-def circline-type-rep-def Re-divide-real sgn-minus[symmetric]
sgn-divide)
qed

lemma
  assumes
  is-circle (of-ocircline H1) is-circle (of-ocircline H2)
  circline-type (of-ocircline H1) < 0 circline-type (of-ocircline H2) < 0
 (a1, r1) = euclidean-circle (of-ocircline H1) (a2, r2) = euclidean-circle (of-ocircline
H2)
```

```
      of-complex E ∈ ocircline-set H1 ∩ ocircline-set H2
  shows cos-angle H1 H2 = cos (ang-circ E a1 a2 (pos-oriented H1) (pos-oriented H2))
proof−
  let ?p1 = pos-oriented H1 and ?p2 = pos-oriented H2
  have E ∈ circle a1 r1 E ∈ circle a2 r2
   using classic-circle[of of-ocircline H1 a1 r1] classic-circle[of of-ocircline H2 a2 r2]
    using assms of-complex-inj
    by auto
  hence ∗: cdist E a1 = r1 cdist E a2 = r2
    unfolding circle-def
    by (simp-all add: norm-minus-commute)
  have r1 > 0 r2 > 0
   using assms(1−6) real-circle-sgn-r[of of-ocircline H1 a1 r1] real-circle-sgn-r[of of-ocircline H2 a2 r2]
    by auto (metis neg-0-less-iff-less sgn-1-pos sgn-sgn)+
  hence E ≠ a1 E ≠ a2
    using ‹cdist E a1 = r1› ‹cdist E a2 = r2›
    by auto

  let ?k = sgn-bool (?p1 = ?p2)
  let ?xx = ?k ∗ (r1² + r2² − (cdist a2 a1)²) / (2 ∗ r1 ∗ r2)

  have cos (ang-circ E a1 a2 ?p1 ?p2) = ?xx
    using law-of-cosines[of a2 a1 E] ∗ ‹r1 > 0› ‹r2 > 0› cos-ang-circ-simp[OF ‹E ≠ a1› ‹E ≠ a2›]
     by (subst (asm) ang-vec-opposite-opposite'[OF ‹E ≠ a1›[symmetric] ‹E ≠ a2›[symmetric], symmetric]) simp
  moreover
  have cos-angle H1 H2 = ?xx
    using ‹r1 > 0› ‹r2 > 0›
   using ‹(a1, r1) = euclidean-circle (of-ocircline H1)› ‹(a2, r2) = euclidean-circle (of-ocircline H2)›
    using ‹is-circle (of-ocircline H1)› ‹is-circle (of-ocircline H2)›
    using ‹circline-type (of-ocircline H1) < 0› ‹circline-type (of-ocircline H2) < 0›
  proof transfer
    fix a1 r1 H1 H2 a2 r2
    obtain A1 B1 C1 D1 where HH1: Rep-circline-mat H1 = (A1, B1, C1, D1)
      by (cases Rep-circline-mat H1) auto
    obtain A2 B2 C2 D2 where HH2: Rep-circline-mat H2 = (A2, B2, C2, D2)
      by (cases Rep-circline-mat H2) auto
    have ∗: is-real A1 is-real A2 is-real D1 is-real D2 cnj B1 = C1 cnj B2 = C2
     using Rep-circline-mat[of H1] Rep-circline-mat[of H2] hermitean-elems[of A1 B1 C1 D1] hermitean-elems[of A2 B2 C2 D2] HH1 HH2
      by auto
    have cnj A1 = A1 cnj A2 = A2
      using ‹is-real A1› ‹is-real A2›
      by (case-tac[!] A1, case-tac[!] A2, auto)
    assume ¬ circline-A0-rep (id H1) ¬ circline-A0-rep (id H2)
    hence A1 ≠ 0 A2 ≠ 0
      using HH1 HH2
      by (auto simp add: circline-A0-rep-def)
    hence Re A1 ≠ 0 Re A2 ≠ 0
      using ‹is-real A1› ‹is-real A2›
      by (metis complex-Im-zero complex-Re-zero complex-equality)+

    assume circline-type-rep (id H1) < 0 circline-type-rep (id H2) < 0
    assume (a1, r1) = euclidean-circle-rep (id H1) (a2, r2) = euclidean-circle-rep (id H2)
    assume r1 > 0 r2 > 0

    let ?D12 = mat-det-12 (Rep-circline-mat H1) (Rep-circline-mat H2) and ?D1 = mat-det (Rep-circline-mat H1) and ?D2 = mat-det (Rep-circline-mat H2)
    let ?x1 = (cdist a2 a1)^2 − r1^2 − r2^2 and ?x2 = 2∗r1∗r2
    let ?x = ?x1 / ?x2
    have ∗: Re (?D12) / (2 ∗ (sqrt (Re (?D1 ∗ ?D2)))) = Re (sgn A1) ∗ Re (sgn A2) ∗ ?x
    proof−
      let ?M1 = (A1, B1, C1, D1) and ?M2 = (A2, B2, C2, D2)
      let ?d1 = B1 ∗ C1 − A1 ∗ D1 and ?d2 = B2 ∗ C2 − A2 ∗ D2
      have Re ?d1 > 0 Re ?d2 > 0
        using HH1 HH2 ‹circline-type-rep (id H1) < 0› ‹circline-type-rep (id H2) < 0›
        by (auto simp add: circline-type-rep-def)
      hence ∗∗: Re (?d1 / (A1 ∗ A1)) > 0 Re (?d2 / (A2 ∗ A2)) > 0
        using ‹is-real A1› ‹is-real A2› ‹A1 ≠ 0› ‹A2 ≠ 0›
           by − (simp add: Re-divide-real, metis Re-divide-real complex-Re-mult divide-pos-pos eq-divide-imp mult-eq-0-iff not-real-square-gt-zero)+
      have ∗∗∗: is-real (?d1 / (A1 ∗ A1)) ∧ is-real (?d2 / (A2 ∗ A2))
       using ‹is-real A1› ‹is-real A2› ‹A1 ≠ 0› ‹A2 ≠ 0› ‹cnj B1 = C1›[symmetric] ‹cnj B2 = C2›[symmetric] ‹is-real D1› ‹is-real D2›
        by (subst div-reals, simp, simp, simp)+

     have cor ?x = mat-det-12 ?M1 ?M2 / (2 ∗ sgn A1 ∗ sgn A2 ∗ cor (sqrt (Re ?d1) ∗ sqrt (Re ?d2)))
      proof−
        have A1∗A2∗cor ?x1 = mat-det-12 ?M1 ?M2
        proof−
          have 1: A1∗A2∗(cor ((cdist a2 a1)^2)) = ((B2∗A1 − A2∗B1)∗(C2∗A1 − C1∗A2)) / (A1∗A2)
         using ‹(a1, r1) = euclidean-circle-rep (id H1)› ‹(a2, r2) = euclidean-circle-rep (id H2)›
            unfolding cdist-def cmod-square
            using HH1 HH2 ∗ ‹A1 ≠ 0› ‹A2 ≠ 0› ‹cnj A1 = A1› ‹cnj A2 = A2›
            apply (subst complex-of-real-Re)
            apply (simp add: complex-mult-cnj-cmod power2-eq-square)
```

```
            apply (simp add: euclidean-circle-rep-def complex-cnj power2-eq-square
field-simps)
            done
          have 2: A1*A2*cor (−r1^2) = A2*D1 − B1*C1*A2/A1
            using ‹(a1, r1) = euclidean-circle-rep (id H1)›
            using HH1 ** * *** ‹A1 ≠ 0›
            apply (simp add: euclidean-circle-rep-def power2-eq-square)
            apply (subst complex-of-real-Re, simp)
            apply (simp add: field-simps)
            done
          have 3: A1*A2*cor (−r2^2) = A1*D2 − B2*C2*A1/A2
            using ‹(a2, r2) = euclidean-circle-rep (id H2)›
            using HH2 ** * *** ‹A2 ≠ 0›
            apply (simp add: euclidean-circle-rep-def power2-eq-square)
            apply (subst complex-of-real-Re, simp)
            apply (simp add: field-simps)
            done
        have A1*A2*cor((cdist a2 a1)^2) + A1*A2*cor(−r1^2) + A1*A2*cor(−r2^2)
= mat-det-12 ?M1 ?M2
            using ‹A1 ≠ 0› ‹A2 ≠ 0›
            by (subst 1, subst 2, subst 3) (simp add: field-simps)
          thus ?thesis
            by (simp add: field-simps)
        qed

        moreover

        have A1 * A2 * cor (?x2) = 2 * sgn A1 * sgn A2 * cor (sqrt (Re ?d1) *
sqrt (Re ?d2))
        proof−
          have 1: sqrt (Re (?d1/ (A1 * A1))) = sqrt (Re ?d1) / |Re A1|
            using ‹A1 ≠ 0› ‹is-real A1›
            by (subst Re-divide-real, simp, simp, subst real-sqrt-divide, simp)

          have 2: sqrt (Re (?d2/ (A2 * A2))) = sqrt (Re ?d2) / |Re A2|
            using ‹A2 ≠ 0› ‹is-real A2›
            by (subst Re-divide-real, simp, simp, subst real-sqrt-divide, simp)
          have sgn A1 = A1 / cor |Re A1|
            using ‹is-real A1›
            unfolding sgn-eq
            by (cases A1, simp)
          moreover
          have sgn A2 = A2 / cor |Re A2|
            using ‹is-real A2›
            unfolding sgn-eq
            by (cases A2, simp)
          ultimately
          show ?thesis
        using ‹(a1, r1) = euclidean-circle-rep (id H1)› ‹(a2, r2) = euclidean-circle-rep
```

```
(id H2)⟩  HH1 HH2
            using ∗∗∗ ⟨is-real A1⟩ ⟨is-real A2⟩
               by (simp add: euclidean-circle-rep-def) (subst 1, subst 2, simp add:
of-real-numeral)
        qed

        ultimately

        have (A1 ∗ A2 ∗ cor ?x1) / (A1 ∗ A2 ∗ (cor ?x2)) =
               mat-det-12 ?M1 ?M2 / (2 ∗ sgn A1 ∗ sgn A2 ∗ cor (sqrt (Re ?d1) ∗
sqrt (Re ?d2)))
          by simp
        thus ?thesis
          using ⟨A1 ≠ 0⟩ ⟨A2 ≠ 0⟩
          by simp
      qed
      hence cor ?x ∗ sgn A1 ∗ sgn A2 = mat-det-12 ?M1 ?M2 / (2 ∗ cor (sqrt
(Re ?d1) ∗ sqrt (Re ?d2)))
        using ⟨A1 ≠ 0⟩ ⟨A2 ≠ 0⟩
        by (simp add: sgn-zero-iff)
      moreover
      have Re (cor ?x ∗ sgn A1 ∗ sgn A2) = Re (sgn A1) ∗ Re (sgn A2) ∗ ?x
      proof−
        have is-real (cor ?x) is-real (sgn A1) is-real (sgn A2)
          using ⟨is-real A1⟩ ⟨is-real A2⟩ Im-complex-of-real[of ?x]
          by auto
        thus ?thesis
          using Re-complex-of-real[of ?x]
          by (subst Re-mult-real3, auto simp add: field-simps)
      qed
      moreover
      have ∗: sqrt (Re ?D1) ∗ sqrt (Re ?D2) = sqrt (Re ?d1) ∗ sqrt (Re ?d2)
        using HH1 HH2
        by (subst real-sqrt-mult[symmetric])+ (simp add: field-simps)
      have 2 ∗ (sqrt (Re (?D1 ∗ ?D2))) ≠ 0
       using ⟨Re ?d1 > 0⟩ ⟨Re ?d2 > 0⟩ HH1 HH2 ⟨is-real A1⟩ ⟨is-real A2⟩ ⟨is-real
D1⟩ ⟨is-real D2⟩
         using Rep-circline-mat[of H1] mat-det-hermitean-real[of Rep-circline-mat
H1]
        by (subst Re-mult-real, auto)
      hence ∗∗: Re (?D12 / (2 ∗ cor (sqrt (Re (?D1 ∗ ?D2))))) = Re (?D12) /
(2 ∗ (sqrt (Re (?D1 ∗ ?D2))))
       using ⟨Re ?d1 > 0⟩ ⟨Re ?d2 > 0⟩ HH1 HH2 ⟨is-real A1⟩ ⟨is-real A2⟩ ⟨is-real
D1⟩ ⟨is-real D2⟩
        by (subst Re-divide-real) (auto simp add: Im-complex-of-real)
      have Re (mat-det-12 ?M1 ?M2 / (2 ∗ cor (sqrt (Re ?d1) ∗ sqrt (Re ?d2))))
= Re (?D12) / (2 ∗ (sqrt (Re (?D1 ∗ ?D2))))
      using HH1 HH2 Rep-circline-mat[of H1] mat-det-hermitean-real[of Rep-circline-mat
H1]
```

```
       by (subst **[symmetric], subst Re-mult-real, simp, subst real-sqrt-mult, subst
*, simp)
      ultimately
      show ?thesis
        by simp
    qed
    have **: pos-oriented-rep H1 ⟷ Re A1 > 0  pos-oriented-rep H2 ⟷ Re A2
> 0
      using ‹Re A1 ≠ 0› HH1  ‹Re A2 ≠ 0› HH2
      by (auto simp add: pos-oriented-rep-def)
    show cos-angle-rep H1 H2 = sgn-bool (pos-oriented-rep H1 = pos-oriented-rep
H2) * (r1^2 + r2^2 − (cdist a2 a1)^2) /  (2 * r1 * r2)
      unfolding cos-angle-rep-def Let-def
      using ‹r1 > 0› ‹r2 > 0›
      by (subst divide-minus-left, subst *, subst Re-sgn[OF ‹is-real A1› ‹A1 ≠ 0›],
subst Re-sgn[OF ‹is-real A2› ‹A2 ≠ 0›], subst **, subst **, simp add: field-simps)
  qed
  ultimately
  show ?thesis
    by simp
qed

lemma [simp]: sqrt a * sqrt a = |a|
by (subst real-sqrt-mult[symmetric]) simp

lemma cos-angle H1 H2 = cos-angle (moebius-ocircline M H1) (moebius-ocircline
M H2)
proof transfer
  fix H1 H2 M
  show cos-angle-rep H1 H2 = cos-angle-rep (moebius-circline-rep M H1) (moebius-circline-rep
M H2)
    unfolding cos-angle-rep-def Let-def moebius-circline-rep-Rep mat-det-12-congruence
mat-det-congruence
    using Rep-moebius-mat[of M] mat-det-inv[of Rep-moebius-mat M]
    by (auto simp add: power2-eq-square real-sqrt-mult field-simps)
qed



lemma
  assumes mat-det (A, B, C, D) ≠ 0
  shows moebius-circline (mk-moebius A B C D) imag-unit-circle = imag-unit-circle
⟷
        unitary-gen (A, B, C, D) (is ?lhs = ?rhs)
proof
  assume ?lhs
  thus ?rhs
    using assms
  proof transfer
```

```
    fix A B C D :: complex
    let ?M = (A, B, C, D) and ?E = (1, 0, 0, 1)
    assume circline-mat-eq (moebius-circline-rep (mk-moebius-rep A B C D) imag-unit-circle-rep) imag-unit-circle-rep mat-det ?M ≠ 0
    then obtain k where k ≠ 0 ?E = cor k *_{sm} congruence (mat-inv ?M) ?E
      by (auto simp add: mk-moebius-rep-Rep)
    hence unitary-gen (mat-inv ?M)
      using mult-sm-inv-l[of cor k congruence (mat-inv ?M) ?E ?E]
      unfolding unitary-gen-def
        by (rule-tac x=1/cor k in exI, simp del: mat-inv.simps, metis eye-def mat-eye-r)
    thus unitary-gen ?M
      using unitary-gen-inv[of mat-inv ?M] ‹mat-det ?M ≠ 0›
      by (simp add: mat-inv-inv del: mat-inv.simps)
  qed
next
  assume ?rhs
  thus ?lhs
    using assms
  proof transfer
    fix A B C D :: complex
    let ?M = (A, B, C, D) and ?E = (1, 0, 0, 1)
    assume unitary-gen ?M mat-det ?M ≠ 0
    hence unitary-gen (mat-inv ?M)
      using unitary-gen-inv[of ?M]
      by simp
    then obtain k where k ≠ 0 mat-adj (mat-inv ?M) *_{mm} (mat-inv ?M) = cor k *_{sm} eye
      using unitary-gen-real[of mat-inv ?M] mat-det-inv[of ?M]
      by auto
    hence *: ?E = (1 / cor k) *_{sm} (mat-adj (mat-inv ?M) *_{mm} (mat-inv ?M))
      using mult-sm-inv-l[of cor k eye mat-adj (mat-inv ?M) *_{mm} (mat-inv ?M)]
      by simp
    show circline-mat-eq (moebius-circline-rep (mk-moebius-rep A B C D) imag-unit-circle-rep) imag-unit-circle-rep
      using ‹mat-det ?M ≠ 0› ‹k ≠ 0›
      by (simp add: mk-moebius-rep-Rep del: mat-inv.simps) (rule-tac x=1/k in exI, subst *, simp del: mat-inv.simps, metis eye-def mat-eye-r)
  qed
qed

end
```