Count the Number of Complex Roots

Wenda Li

June 16, 2019

Abstract

Based on evaluating Cauchy indices through remainder sequences [1] [2, Chapter 11], this entry provides an effective procedure to count the number of complex roots (with multiplicity) of a polynomial within a rectangle box or a half-plane. Potential applications of this entry include certified complex root isolation (of a polynomial) and testing the Routh-Hurwitz stability criterion (i.e., to check whether all the roots of some characteristic polynomial have negative real parts).

1 An alternative Sturm sequences

theory *Extended-Sturm* **imports**
Sturm-Tarski.Sturm-Tarski
Winding-Number-Eval.Cauchy-Index-Theorem
begin

The main purpose of this theory is to provide an effective way to compute *cindexE a b f* when *f* is a rational function. The idea is similar to and based on the evaluation of *cindex-poly* through $[\![?a < ?b; \mathit{poly}\ ?p\ ?a \neq 0; \mathit{poly}\ ?p\ ?b \neq 0]\!] \Longrightarrow \mathit{cindex\text{-}poly}\ ?a\ ?b\ ?q\ ?p = \mathit{changes\text{-}itv\text{-}smods}\ ?a\ ?b\ ?p\ ?q$.

This alternative version of remainder sequences is inspired by the paper "The Fundamental Theorem of Algebra made effective: an elementary real-algebraic proof via Sturm chains" by Michael Eisermann.

hide-const *Permutations.sign*

1.1 Misc

lemma *is-unit-pCons-ex-iff*:
fixes $p::'a::field\ poly$
shows $\mathit{is\text{-}unit}\ p \longleftrightarrow (\exists\, a.\ a{\neq}0 \wedge p{=}[:a:])$
using *is-unit-poly-iff is-unit-triv* **by** *auto*

lemma *poly-gcd-iff*:
$\mathit{poly}\ (\mathit{gcd}\ p\ q)\ x{=}0 \longleftrightarrow \mathit{poly}\ p\ x{=}0 \wedge \mathit{poly}\ q\ x{=}0$
by (*simp add*: *poly-eq-0-iff-dvd*)

lemma *eventually-poly-nz-at-within*:
fixes x :: *'a*::{*idom*,*euclidean-space*}
assumes $p \neq 0$
shows *eventually* (λx. *poly p x*$\neq$*0*) (*at x within S*)
proof −
have *eventually* (λx. *poly p x*$\neq$*0*) (*at x within S*)
= ($\forall_F$ *x in* (*at x within S*). $\forall$ *y*$\in$*proots p*. *x* $\neq$ *y*)
apply (*rule eventually-subst*,*rule eventuallyI*)
by (*auto simp add*:*proots-def*)
also have ... = ($\forall$ *y*$\in$*proots p*. $\forall_F$ *x in* (*at x within S*). *x* $\neq$ *y*)
apply (*subst eventually-ball-finite-distrib*)
using ‹*p*$\neq$*0*› **by** *auto*
also have ...
unfolding *eventually-at*
by (*metis gt-ex not-less-iff-gr-or-eq zero-less-dist-iff*)
finally show *?thesis* **.**
qed

lemma *sgn-power*:
fixes *x*::*'a*::*linordered-idom*
shows *sgn* (*x*^*n*) = (*if n*=*0 then 1 else if even n then* |*sgn x*| *else sgn x*)
apply (*induct n*)
by (*auto simp add*:*sgn-mult*)

lemma *poly-divide-filterlim-at-top*:
fixes *p q*::*real poly*
defines *ll*≡(*if degree q*<*degree p then*
at 0
else if degree q=*degree p then*
nhds (*lead-coeff q* / *lead-coeff p*)
else if sgn-pos-inf q ∗ *sgn-pos-inf p* > *0 then*
at-top
else
at-bot)
assumes *p*$\neq$*0 q*$\neq$*0*
shows *filterlim* (λx. *poly q x* / *poly p x*) *ll at-top*
proof −
define *pp* **where** *pp*=(λx. *poly p x* / *x*^(*degree p*))
define *qq* **where** *qq*=(λx. *poly q x* / *x*^(*degree q*))
define *dd* **where** *dd*=(λx::*real*. *if degree p*>*degree q then 1*/*x*^(*degree p* − *degree q*) *else*
x^(*degree q* − *degree p*))
have *divide-cong*:$\forall_F x$ *in at-top*. *poly q x* / *poly p x* = *qq x* / *pp x* ∗ *dd x*
proof (*rule eventually-at-top-linorderI*[*of 1*])
fix *x* **assume** (*x*::*real*)≥*1*
then have *x*$\neq$*0* **by** *auto*
then show *poly q x* / *poly p x* = *qq x* / *pp x* ∗ *dd x*
unfolding *qq-def pp-def dd-def* **using** *assms*
by (*auto simp add*:*field-simps power-diff*)

```
  qed
  have qqpp-tendsto:((λx. qq x / pp x) ⟶ lead-coeff q / lead-coeff p) at-top
  proof −
    have (qq ⟶ lead-coeff q) at-top
      unfolding qq-def using poly-divide-tendsto-aux[of q]
      by (auto elim!:filterlim-mono simp:at-top-le-at-infinity)
    moreover have (pp ⟶ lead-coeff p) at-top
      unfolding pp-def using poly-divide-tendsto-aux[of p]
      by (auto elim!:filterlim-mono simp:at-top-le-at-infinity)
    ultimately show ?thesis using ‹p≠0› by (auto intro!:tendsto-eq-intros)
  qed

  have ?thesis when degree q<degree p
  proof −
    have filterlim (λx. poly q x / poly p x) (at 0) at-top
    proof (rule filterlim-atI)
      show ((λx. poly q x / poly p x) ⟶ 0) at-top
        using poly-divide-tendsto-0-at-infinity[OF that]
        by (auto elim:filterlim-mono simp:at-top-le-at-infinity)
      have ∀F x in at-top. poly q x ≠0 ∀F x in at-top. poly p x ≠0
      using poly-eventually-not-zero[OF ‹q≠0›] poly-eventually-not-zero[OF ‹p≠0›]
            filter-leD[OF at-top-le-at-infinity]
        by auto
      then show ∀F x in at-top. poly q x / poly p x ≠ 0
        apply eventually-elim
        by auto
    qed
    then show ?thesis unfolding ll-def using that by auto
  qed
  moreover have ?thesis when degree q=degree p
  proof −
    have ((λx. poly q x / poly p x) ⟶ lead-coeff q / lead-coeff p) at-top
      using divide-cong qqpp-tendsto that unfolding dd-def
      by (auto dest:tendsto-cong)
    then show ?thesis unfolding ll-def using that by auto
  qed
  moreover have ?thesis when degree q>degree p sgn-pos-inf q * sgn-pos-inf p >
0
  proof −
    have filterlim (λx. (qq x / pp x) * dd x) at-top at-top
    proof (subst filterlim-tendsto-pos-mult-at-top-iff[OF qqpp-tendsto])
      show 0 < lead-coeff q / lead-coeff p using that(2) unfolding sgn-pos-inf-def
        by (simp add: zero-less-divide-iff zero-less-mult-iff)
      show filterlim dd at-top at-top
        unfolding dd-def using that(1)
        by (auto intro!:filterlim-pow-at-top simp:filterlim-ident)
    qed
    then have LIM x at-top. poly q x / poly p x :> at-top
      using filterlim-cong[OF - - divide-cong] by blast
```

```
    then show ?thesis unfolding ll-def using that by auto
  qed
  moreover have ?thesis  when degree q>degree p ¬ sgn-pos-inf q * sgn-pos-inf
p > 0
  proof −
    have filterlim (λx. (qq x / pp x) * dd x) at-bot at-top
    proof (subst filterlim-tendsto-neg-mult-at-bot-iff[OF qqpp-tendsto])
      show lead-coeff q / lead-coeff p < 0
        using that(2) ‹p≠0› ‹q≠0› unfolding sgn-pos-inf-def
        by (metis divide-eq-0-iff divide-sgn leading-coeff-0-iff
            linorder-neqE-linordered-idom sgn-divide sgn-greater)
      show filterlim dd at-top at-top
        unfolding dd-def using that(1)
        by (auto intro!:filterlim-pow-at-top simp:filterlim-ident)
    qed
    then have LIM x at-top. poly q x / poly p x :> at-bot
      using filterlim-cong[OF - - divide-cong] by blast
    then show ?thesis unfolding ll-def using that by auto
  qed
  ultimately show ?thesis by linarith
qed

lemma poly-divide-filterlim-at-bot:
  fixes p q::real poly
  defines ll≡( if degree q<degree p then
                  at 0
               else if degree q=degree p then
                  nhds (lead-coeff q / lead-coeff p)
               else if sgn-neg-inf q * sgn-neg-inf p > 0 then
                  at-top
               else
                  at-bot)
  assumes p≠0 q≠0
  shows filterlim (λx. poly q x / poly p x) ll at-bot
proof −
  define pp where pp=(λx. poly p x / x^(degree p))
  define qq where qq=(λx. poly q x / x^(degree q))
  define dd where dd=(λx::real. if degree p>degree q then 1/x^(degree p − degree
q) else
                                  x^(degree q − degree p))
  have divide-cong:∀ F x in at-bot. poly q x / poly p x = qq x / pp x * dd x
  proof (rule eventually-at-bot-linorderI[of −1])
    fix x assume (x::real)≤−1
    then have x≠0 by auto
    then show poly q x / poly p x = qq x / pp x * dd x
      unfolding qq-def pp-def dd-def using assms
      by (auto simp add:field-simps power-diff)
  qed
  have qqpp-tendsto:((λx. qq x / pp x) ⟶ lead-coeff q / lead-coeff p) at-bot
```

```
  proof −
    have (qq ⟶ lead-coeff q) at-bot
      unfolding qq-def using poly-divide-tendsto-aux[of q]
      by (auto elim!:filterlim-mono simp:at-bot-le-at-infinity)
    moreover have (pp ⟶ lead-coeff p) at-bot
      unfolding pp-def using poly-divide-tendsto-aux[of p]
      by (auto elim!:filterlim-mono simp:at-bot-le-at-infinity)
    ultimately show ?thesis using ‹p≠0› by (auto intro!:tendsto-eq-intros)
  qed

  have ?thesis when degree q<degree p
  proof −
    have filterlim (λx. poly q x / poly p x) (at 0) at-bot
    proof (rule filterlim-atI)
      show ((λx. poly q x / poly p x) ⟶ 0) at-bot
        using poly-divide-tendsto-0-at-infinity[OF that]
        by (auto elim:filterlim-mono simp:at-bot-le-at-infinity)
      have ∀F x in at-bot. poly q x ≠0 ∀F x in at-bot. poly p x ≠0
      using poly-eventually-not-zero[OF ‹q≠0›] poly-eventually-not-zero[OF ‹p≠0›]
            filter-leD[OF at-bot-le-at-infinity]
        by auto
      then show ∀F x in at-bot. poly q x / poly p x ≠ 0
        by eventually-elim auto
    qed
    then show ?thesis unfolding ll-def using that by auto
  qed
  moreover have ?thesis when degree q=degree p
  proof −
    have ((λx. poly q x / poly p x) ⟶ lead-coeff q / lead-coeff p) at-bot
      using divide-cong qqpp-tendsto that unfolding dd-def
      by (auto dest:tendsto-cong)
    then show ?thesis unfolding ll-def using that by auto
  qed
  moreover have ?thesis when degree q>degree p sgn-neg-inf q * sgn-neg-inf p
> 0
  proof −
    define cc where cc=lead-coeff q / lead-coeff p
    have (cc > 0 ∧ even (degree q − degree p)) ∨ (cc<0 ∧ odd (degree q − degree
p))
    proof −
      have even (degree q − degree p) ⟷
            (even (degree q) ∧ even (degree p)) ∨ (odd (degree q) ∧ odd (degree p))
        using ‹degree q>degree p› by auto
      then show ?thesis
        using that ‹p≠0› ‹q≠0› unfolding sgn-neg-inf-def cc-def zero-less-mult-iff
          divide-less-0-iff zero-less-divide-iff
        apply (simp add:if-split[of (<) 0] if-split[of (>) 0])
        by argo
    qed
```

```
    moreover have filterlim (λx. (qq x / pp x) * dd x) at-top at-bot
      when cc>0 even (degree q − degree p)
    proof (subst filterlim-tendsto-pos-mult-at-top-iff[OF qqpp-tendsto])
      show 0 < lead-coeff q / lead-coeff p using ‹cc>0› unfolding cc-def by auto
      show filterlim dd at-top at-bot
        unfolding dd-def using ‹degree q>degree p› that(2)
        by (auto intro!:filterlim-pow-at-bot-even simp:filterlim-ident)
    qed
    moreover have filterlim (λx. (qq x / pp x) * dd x) at-top at-bot
      when cc<0 odd (degree q − degree p)
    proof (subst filterlim-tendsto-neg-mult-at-top-iff[OF qqpp-tendsto])
      show 0 > lead-coeff q / lead-coeff p using ‹cc<0› unfolding cc-def by auto
      show filterlim dd at-bot at-bot
        unfolding dd-def using ‹degree q>degree p› that(2)
        by (auto intro!:filterlim-pow-at-bot-odd simp:filterlim-ident)
    qed
    ultimately have filterlim (λx. (qq x / pp x) * dd x) at-top at-bot
      by blast
    then have LIM x at-bot. poly q x / poly p x :> at-top
      using filterlim-cong[OF - - divide-cong] by blast
    then show ?thesis unfolding ll-def using that by auto
  qed
  moreover have ?thesis  when degree q>degree p ¬ sgn-neg-inf q * sgn-neg-inf
p > 0
  proof −
    define cc where cc=lead-coeff q / lead-coeff p
    have (cc < 0 ∧ even (degree q − degree p)) ∨ (cc > 0 ∧ odd (degree q − degree
p))
    proof −
      have even (degree q − degree p) ⟷
            (even (degree q) ∧ even (degree p)) ∨ (odd (degree q) ∧ odd (degree p))
        using ‹degree q>degree p› by auto
      then show ?thesis
        using that ‹p≠0› ‹q≠0› unfolding sgn-neg-inf-def cc-def zero-less-mult-iff
          divide-less-0-iff zero-less-divide-iff
        apply (simp add:if-split[of (<) 0] if-split[of (>) 0])
        by (metis leading-coeff-0-iff linorder-neqE-linordered-idom)
    qed
    moreover have filterlim (λx. (qq x / pp x) * dd x) at-bot at-bot
      when cc<0 even (degree q − degree p)
    proof (subst filterlim-tendsto-neg-mult-at-bot-iff[OF qqpp-tendsto])
      show 0 > lead-coeff q / lead-coeff p using ‹cc<0› unfolding cc-def by auto
      show filterlim dd at-top at-bot
        unfolding dd-def using ‹degree q>degree p› that(2)
        by (auto intro!:filterlim-pow-at-bot-even simp:filterlim-ident)
    qed
    moreover have filterlim (λx. (qq x / pp x) * dd x) at-bot at-bot
      when cc>0 odd (degree q − degree p)
    proof (subst filterlim-tendsto-pos-mult-at-bot-iff[OF qqpp-tendsto])
```

```
    show 0 < lead-coeff q / lead-coeff p using ‹cc>0› unfolding cc-def by auto
    show filterlim dd at-bot at-bot
      unfolding dd-def using ‹degree q>degree p› that(2)
      by (auto intro!:filterlim-pow-at-bot-odd simp:filterlim-ident)
   qed
   ultimately have filterlim (λx. (qq x / pp x) * dd x) at-bot at-bot
    by blast
   then have LIM x at-bot. poly q x / poly p x :> at-bot
    using filterlim-cong[OF - - divide-cong] by blast
   then show ?thesis unfolding ll-def using that by auto
  qed
  ultimately show ?thesis by linarith
qed
```

1.2 Alternative definition of cross

```
definition cross-alt :: real poly ⇒real poly ⇒ real ⇒ real ⇒ int where
  cross-alt p q a b=|sign (poly p a) − sign (poly q a)| − |sign (poly p b) − sign(poly
q b)|

lemma cross-alt-coprime-0:
  assumes coprime p q p=0∨q=0
  shows cross-alt p q a b=0
proof −
  have ?thesis when p=0
  proof −
    have is-unit q using that ‹coprime p q›
      by simp
    then obtain a where a≠0 q=[:a:] using is-unit-pCons-ex-iff by blast
    then show ?thesis using that unfolding cross-alt-def by auto
  qed
  moreover have ?thesis when q=0
  proof −
    have is-unit p using that ‹coprime p q›
      by simp
    then obtain a where a≠0 p=[:a:] using is-unit-pCons-ex-iff by blast
    then show ?thesis using that unfolding cross-alt-def by auto
  qed
  ultimately show ?thesis using ‹p=0∨q=0› by auto
qed

lemma cross-alt-0[simp]: cross-alt 0 0 a b=0 unfolding cross-alt-def by auto

lemma cross-alt-poly-commute:
  cross-alt p q a b = cross-alt q p a b
  unfolding cross-alt-def by auto

lemma cross-alt-clear-n:
  assumes coprime p q
```

shows *cross-alt p q a b = cross-alt 1 (p∗q) a b*
proof −
have *|sign (poly p a) − sign (poly q a)| = |1 − sign (poly p a) ∗ sign (poly q a)|*
proof (*cases poly p a=0 ∧ poly q a=0*)
case *True*
then have *False* **using** *assms* **using** *coprime-poly-0* **by** *blast*
then show *?thesis* **by** *auto*
next
case *False*
then show *?thesis*
unfolding *Sturm-Tarski.sign-def*
by *force*
qed
moreover have *|sign (poly p b) − sign (poly q b)| = |1 − sign (poly p b) ∗ sign (poly q b)|*
proof (*cases poly p b=0 ∧ poly q b=0*)
case *True*
then have *False* **using** *assms* **using** *coprime-poly-0* **by** *blast*
then show *?thesis* **by** *auto*
next
case *False*
then show *?thesis*
unfolding *Sturm-Tarski.sign-def*
by *force*
qed
ultimately show *?thesis*
by (*simp add*: *cross-alt-def sign-times*)
qed

1.3 Alternative sign variation sequencse

fun *changes-alt*:: (*'a* ::*linordered-idom*) *list* ⇒ *int* **where**
changes-alt [] = 0|
changes-alt [-] = 0 |
changes-alt (x1#x2#xs) = abs(sign x1 − sign x2) + changes-alt (x2#xs)

definition *changes-alt-poly-at*::(*'a* ::*linordered-idom*) *poly list* ⇒ *'a* ⇒ *int* **where**
changes-alt-poly-at ps a= changes-alt (map (λp. poly p a) ps)

definition *changes-alt-itv-smods*:: *real* ⇒ *real* ⇒*real poly* ⇒ *real poly* ⇒ *int* **where**
changes-alt-itv-smods a b p q= (let ps= smods p q
in changes-alt-poly-at ps a − changes-alt-poly-at ps b)

lemma *changes-alt-itv-smods-rec*:
assumes *a<b coprime p q*
shows *changes-alt-itv-smods a b p q = cross-alt p q a b + changes-alt-itv-smods a b q (−(p mod q))*

proof (*cases* $p = 0 \vee q = 0 \vee q$ *dvd* p)
case *True*
moreover have $p{=}0 \vee q{=}0 \Longrightarrow$ *?thesis*
using *cross-alt-coprime-0*[*OF* ‹*coprime p q*›]
unfolding *changes-alt-itv-smods-def changes-alt-poly-at-def* **by** *fastforce*
moreover have $[\![p{\neq}0; q{\neq}0; p \bmod q = 0]\!] \Longrightarrow$ *?thesis*
unfolding *changes-alt-itv-smods-def changes-alt-poly-at-def cross-alt-def*
by (*simp add*:*sign-times*)
ultimately show *?thesis*
by *auto* (*auto elim*: *dvdE*)
next
case *False*
hence $p{\neq}0$ $q{\neq}0$ $p \bmod q{\neq}0$ **by** *auto*
then obtain *ps* **where** *ps*:*smods p q*=*p*#*q*#−(*p mod q*)#*ps smods q* (−(*p mod q*)) = *q*#−(*p mod q*)#*ps*
by *auto*
define *changes-diff* **where** *changes-diff*≡λ*x*. *changes-alt-poly-at* (*p*#*q*#−(*p mod q*)#*ps*) *x*
− *changes-alt-poly-at* (*q*#−(*p mod q*)#*ps*) *x*
have *changes-diff a* − *changes-diff b*=*cross-alt p q a b*
unfolding *changes-diff-def changes-alt-poly-at-def cross-alt-def* **by** *simp*
thus *?thesis* **unfolding** *changes-alt-itv-smods-def changes-diff-def changes-alt-poly-at-def ps*
by *force*
qed

1.4 jumpF on polynomials

definition *jumpF-polyR*:: *real poly* ⇒ *real poly* ⇒ *real* ⇒ *real* **where**
jumpF-polyR q p a = *jumpF* (λ*x*. *poly q x* / *poly p x*) (*at-right a*)

definition *jumpF-polyL*:: *real poly* ⇒ *real poly* ⇒ *real* ⇒ *real* **where**
jumpF-polyL q p a = *jumpF* (λ*x*. *poly q x* / *poly p x*) (*at-left a*)

definition *jumpF-poly-top*:: *real poly* ⇒ *real poly* ⇒ *real* **where**
jumpF-poly-top q p = *jumpF* (λ*x*. *poly q x* / *poly p x*) *at-top*

definition *jumpF-poly-bot*:: *real poly* ⇒ *real poly* ⇒ *real* **where**
jumpF-poly-bot q p = *jumpF* (λ*x*. *poly q x* / *poly p x*) *at-bot*

lemma *jumpF-polyR-0*[*simp*]: *jumpF-polyR 0 p a* = *0 jumpF-polyR q 0 a* = *0*
unfolding *jumpF-polyR-def* **by** *auto*

lemma *jumpF-polyL-0*[*simp*]: *jumpF-polyL 0 p a* = *0 jumpF-polyL q 0 a* = *0*
unfolding *jumpF-polyL-def* **by** *auto*

lemma *jumpF-polyR-mult-cancel*:
assumes $p'{\neq}0$

shows *jumpF-polyR* $(p' * q)$ $(p' * p)$ a = *jumpF-polyR q p a*
unfolding *jumpF-polyR-def*
proof (*rule jumpF-cong*)
obtain *ub* **where** $a < ub \; \forall z.\; a < z \wedge z \leq ub \longrightarrow poly\; p'\; z \neq 0$
using *next-non-root-interval*[*OF* ‹$p' \neq 0$›,*of a*] **by** *auto*
then show $\forall_F\; x\; in\; at\text{-}right\; a.\; poly\; (p' * q)\; x\; /\; poly\; (p' * p)\; x = poly\; q\; x\; /\; poly\; p\; x$
apply (*unfold eventually-at-right*)
apply (*intro exI*[**where** *x*=*ub*])
by *auto*
qed *simp*

lemma *jumpF-polyL-mult-cancel*:
assumes $p' \neq 0$
shows *jumpF-polyL* $(p' * q)$ $(p' * p)$ a = *jumpF-polyL q p a*
unfolding *jumpF-polyL-def*
proof (*rule jumpF-cong*)
obtain *lb* **where** $lb < a \; \forall z.\; lb \leq z \wedge z < a \longrightarrow poly\; p'\; z \neq 0$
using *last-non-root-interval*[*OF* ‹$p' \neq 0$›,*of a*] **by** *auto*
then show $\forall_F\; x\; in\; at\text{-}left\; a.\; poly\; (p' * q)\; x\; /\; poly\; (p' * p)\; x = poly\; q\; x\; /\; poly\; p\; x$
apply (*unfold eventually-at-left*)
apply (*intro exI*[**where** *x*=*lb*])
by *auto*
qed *simp*

lemma *jumpF-poly-noroot*:
assumes *poly p a*$\neq$*0*
shows *jumpF-polyL q p a* = *0 jumpF-polyR q p a* = *0*
subgoal unfolding *jumpF-polyL-def* **using** *assms*
apply (*intro jumpF-not-infinity*)
by (*auto intro*!:*continuous-intros*)
subgoal unfolding *jumpF-polyR-def* **using** *assms*
apply (*intro jumpF-not-infinity*)
by (*auto intro*!:*continuous-intros*)
done

lemma *jumpF-polyR-coprime*:
assumes *coprime p q*
shows *jumpF-polyR q p x* = (*if p* $\neq$ *0* $\wedge$ *q* $\neq$ *0* $\wedge$ *poly p x*=*0 then*
if sign-r-pos p x $\longleftrightarrow$ *poly q x*>*0 then 1/2 else* $-$ *1/2*
else 0)
proof (*cases p*=*0* $\vee$ *q*=*0* $\vee$ *poly p x*$\neq$*0*)
case *True*
then show *?thesis* **using** *jumpF-poly-noroot* **by** *fastforce*
next
case *False*
then have *asm*:*p*$\neq$*0 q*$\neq$*0 poly p x*=*0* **by** *auto*

```
  then have poly q x≠0 using assms using coprime-poly-0 by blast
  have ?thesis when sign-r-pos p x ⟷ poly q x>0
  proof −
    have (poly p has-sgnx sgn (poly q x)) (at-right x)
    by (metis False ‹poly q x ≠ 0› add.inverse-neutral has-sgnx-imp-sgnx less-not-sym

        neg-less-iff-less poly-has-sgnx-values(2) sgn-if sign-r-pos-sgnx-iff that
        trivial-limit-at-right-real zero-less-one)
    then have LIM x at-right x. poly q x / poly p x :> at-top
      apply (subst filterlim-divide-at-bot-at-top-iff[of - poly q x])
      apply (auto simp add:‹poly q x≠0›)
      by (metis asm(3) poly-tendsto(3))
    then have jumpF-polyR q p x = 1/2
      unfolding jumpF-polyR-def jumpF-def by auto
    then show ?thesis using that False by auto
  qed
  moreover have ?thesis when ¬ (sign-r-pos p x ⟷ poly q x>0)
  proof −
    have (poly p has-sgnx − sgn (poly q x)) (at-right x)
    proof −
      have (0::real) < 1 ∨ ¬ (1::real) < 0 ∧ sign-r-pos p x
          ∨ (poly p has-sgnx − sgn (poly q x)) (at-right x)
        by simp
      then show ?thesis
      by (metis (no-types) False ‹poly q x ≠ 0› add.inverse-inverse has-sgnx-imp-sgnx

          neg-less-0-iff-less poly-has-sgnx-values(2) sgn-if sgn-less sign-r-pos-sgnx-iff

            that trivial-limit-at-right-real)
    qed
    then have LIM x at-right x. poly q x / poly p x :> at-bot
      apply (subst filterlim-divide-at-bot-at-top-iff[of - poly q x])
      apply (auto simp add:‹poly q x≠0›)
      by (metis asm(3) poly-tendsto(3))
    then have jumpF-polyR q p x = − 1/2
      unfolding jumpF-polyR-def jumpF-def by auto
    then show ?thesis using that False by auto
  qed
  ultimately show ?thesis by auto
qed

lemma jumpF-polyL-coprime:
  assumes coprime p q
  shows jumpF-polyL q p x = (if p ≠ 0 ∧ q ≠ 0 ∧ poly p x=0 then
              if even (order x p) ⟷ sign-r-pos p x ⟷ poly q x>0 then 1/2 else
− 1/2 else 0)
proof (cases p=0 ∨ q=0 ∨ poly p x≠0)
  case True
  then show ?thesis using jumpF-poly-noroot by fastforce
```

next
case *False*
then have *asm:p≠0 q≠0 poly p x=0* **by** *auto*
then have *poly q x≠0* **using** *assms* **using** *coprime-poly-0* **by** *blast*
have *?thesis* **when** *even* (*order x p*) ⟷ *sign-r-pos p x* ⟷ *poly q x>0*
proof −
consider (*lt*) *poly q x>0* | (*gt*) *poly q x<0* **using** ‹*poly q x≠0*› **by** *linarith*
then have *sgnx* (*poly p*) (*at-left x*) = *sgn* (*poly q x*)
apply *cases*
subgoal using *that sign-r-pos-sgnx-iff poly-sgnx-values*[*OF* ‹*p≠0*›,*of x*]
apply (*subst poly-sgnx-left-right*[*OF* ‹*p≠0*›])
by *auto*
subgoal using *that sign-r-pos-sgnx-iff poly-sgnx-values*[*OF* ‹*p≠0*›,*of x*]
apply (*subst poly-sgnx-left-right*[*OF* ‹*p≠0*›])
by *auto*
done
then have (*poly p has-sgnx sgn* (*poly q x*)) (*at-left x*)
by (*metis sgnx-able-poly*(*2*) *sgnx-able-sgnx*)
then have *LIM x at-left x. poly q x / poly p x :> at-top*
apply (*subst filterlim-divide-at-bot-at-top-iff*[*of - poly q x*])
apply (*auto simp add:*‹*poly q x≠0*›)
by (*metis asm*(*3*) *poly-tendsto*(*2*))
then have *jumpF-polyL q p x = 1/2*
unfolding *jumpF-polyL-def jumpF-def* **by** *auto*
then show *?thesis* **using** *that False* **by** *auto*
qed
moreover have *?thesis* **when** ¬ (*even* (*order x p*) ⟷ *sign-r-pos p x* ⟷ *poly q x>0*)
proof −
consider (*lt*) *poly q x>0* | (*gt*) *poly q x<0* **using** ‹*poly q x≠0*› **by** *linarith*
then have *sgnx* (*poly p*) (*at-left x*) = − *sgn* (*poly q x*)
apply *cases*
subgoal using *that sign-r-pos-sgnx-iff poly-sgnx-values*[*OF* ‹*p≠0*›,*of x*]
apply (*subst poly-sgnx-left-right*[*OF* ‹*p≠0*›])
by *auto*
subgoal using *that sign-r-pos-sgnx-iff poly-sgnx-values*[*OF* ‹*p≠0*›,*of x*]
apply (*subst poly-sgnx-left-right*[*OF* ‹*p≠0*›])
by *auto*
done
then have (*poly p has-sgnx* − *sgn* (*poly q x*)) (*at-left x*)
by (*metis sgnx-able-poly*(*2*) *sgnx-able-sgnx*)
then have *LIM x at-left x. poly q x / poly p x :> at-bot*
apply (*subst filterlim-divide-at-bot-at-top-iff*[*of - poly q x*])
apply (*auto simp add:*‹*poly q x≠0*›)
by (*metis asm*(*3*) *poly-tendsto*(*2*))
then have *jumpF-polyL q p x* = − *1/2*
unfolding *jumpF-polyL-def jumpF-def* **by** *auto*
then show *?thesis* **using** *that False* **by** *auto*
qed

```
  ultimately show ?thesis by auto
qed

lemma jumpF-times:
  assumes tendsto:(f ⟶ c) F and c≠0 F≠bot
  shows jumpF (λx. f x * g x) F = sgn c * jumpF g F
proof −
  have c>0 ∨ c<0 using ‹c≠0› by auto
  moreover have ?thesis when c>0
  proof −
    note filterlim-tendsto-pos-mult-at-top-iff[OF tendsto ‹c>0›,of g]
    moreover note filterlim-tendsto-pos-mult-at-bot-iff[OF tendsto ‹c>0›,of g]
    moreover have sgn c = 1 using ‹c>0› by auto
    ultimately show ?thesis unfolding jumpF-def by auto
  qed
  moreover have ?thesis when c<0
  proof −
    define atbot where atbot = filterlim g at-bot F
    define attop where attop = filterlim g at-top F
    have jumpF (λx. f x * g x) F = (if atbot then 1 / 2 else if attop then − 1 /
2 else 0)
    proof −
      note filterlim-tendsto-neg-mult-at-top-iff[OF tendsto ‹c<0›,of g]
      moreover note filterlim-tendsto-neg-mult-at-bot-iff[OF tendsto ‹c<0›,of g]
      ultimately show ?thesis unfolding jumpF-def atbot-def attop-def by auto
    qed
    also have ... = − (if attop then 1 / 2 else if atbot then − 1 / 2 else 0)
    proof −
      have False when atbot attop
          using filterlim-at-top-at-bot[OF - - ‹F≠bot›] that unfolding atbot-def
attop-def by auto
      then show ?thesis by fastforce
    qed
    also have ... = sgn c * jumpF g F
      using ‹c<0› unfolding jumpF-def attop-def atbot-def by auto
    finally show ?thesis .
  qed
  ultimately show ?thesis by auto
qed


lemma jumpF-polyR-inverse-add:
  assumes coprime p q
  shows jumpF-polyR q p x + jumpF-polyR p q x = jumpF-polyR 1 (q*p) x
proof (cases p=0 ∨ q=0)
  case True
  then show ?thesis by auto
next
  case False
```

```
  have jumpF-add:
    jumpF-polyR q p x= jumpF-polyR 1 (q*p) x when poly p x=0 coprime p q for
p q
  proof (cases p=0)
    case True
    then show ?thesis by auto
  next
    case False
    have poly q x≠0 using that coprime-poly-0 by blast
    then have q≠0 by auto
    moreover have sign-r-pos p x = (0 < poly q x) ⟷ sign-r-pos (q * p) x
     using sign-r-pos-mult[OF ‹q≠0› ‹p≠0›] sign-r-pos-rec[OF ‹q≠0›] ‹poly q x≠0›
      by auto
    ultimately show ?thesis using ‹poly p x=0›
    unfolding jumpF-polyR-coprime[OF ‹coprime p q›,of x] jumpF-polyR-coprime[of
q*p 1 x,simplified]
      by auto
  qed
  have False when poly p x=0 poly q x=0
    using ‹coprime p q› that coprime-poly-0 by blast
  moreover have ?thesis when poly p x=0 poly q x≠0
  proof −
    have jumpF-polyR p q x = 0 using jumpF-poly-noroot[OF ‹poly q x≠0›] by
auto
    then show ?thesis using jumpF-add[OF ‹poly p x=0› ‹coprime p q›] by auto
  qed
  moreover have ?thesis when poly p x≠0 poly q x=0
  proof −
    have jumpF-polyR q p x = 0 using jumpF-poly-noroot[OF ‹poly p x≠0›] by
auto
    then show ?thesis using jumpF-add[OF ‹poly q x=0›,of p] ‹coprime p q›
      by (simp add: ac-simps)
  qed
  moreover have ?thesis when poly p x≠0 poly q x≠0
    by (simp add: jumpF-poly-noroot(2) that(1) that(2))
  ultimately show ?thesis by auto
qed

lemma jumpF-polyL-inverse-add:
  assumes coprime p q
  shows jumpF-polyL q p x + jumpF-polyL p q x = jumpF-polyL 1 (q*p) x
proof (cases p=0 ∨ q=0)
  case True
  then show ?thesis by auto
next
  case False
  have jumpF-add:
    jumpF-polyL q p x= jumpF-polyL 1 (q*p) x when poly p x=0 coprime p q for
p q
```

```
  proof (cases p=0)
    case True
    then show ?thesis by auto
  next
    case False
    have poly q x≠0 using that coprime-poly-0 by blast
    then have q≠0 by auto
    moreover have sign-r-pos p x = (0 < poly q x) ⟷ sign-r-pos (q * p) x
     using sign-r-pos-mult[OF ‹q≠0› ‹p≠0›] sign-r-pos-rec[OF ‹q≠0›] ‹poly q x≠0›
      by auto
    moreover have order x p = order x (q * p)
    by (metis ‹poly q x ≠ 0› add-cancel-right-left divisors-zero order-mult order-root)
    ultimately show ?thesis using ‹poly p x=0›
    unfolding jumpF-polyL-coprime[OF ‹coprime p q›,of x] jumpF-polyL-coprime[of
q*p 1 x,simplified]
      by auto
  qed
  have False when poly p x=0 poly q x=0
    using ‹coprime p q› that coprime-poly-0 by blast
  moreover have ?thesis when poly p x=0 poly q x≠0
  proof −
    have jumpF-polyL p q x = 0 using jumpF-poly-noroot[OF ‹poly q x≠0›] by
auto
    then show ?thesis using jumpF-add[OF ‹poly p x=0› ‹coprime p q›] by auto
  qed
  moreover have ?thesis when poly p x≠0 poly q x=0
  proof −
    have jumpF-polyL q p x = 0 using jumpF-poly-noroot[OF ‹poly p x≠0›] by
auto
    then show ?thesis using jumpF-add[OF ‹poly q x=0›,of p] ‹coprime p q›
      by (simp add: ac-simps)
  qed
  moreover have ?thesis when poly p x≠0 poly q x≠0
    by (simp add: jumpF-poly-noroot that(1) that(2))
  ultimately show ?thesis by auto
qed


lemma jumpF-polyL-smult-1:
  jumpF-polyL (smult c q) p x = sgn c * jumpF-polyL q p x
proof (cases c=0)
  case True
  then show ?thesis by auto
next
  case False
  then show ?thesis
    unfolding jumpF-polyL-def
    apply (subst jumpF-times[of λ-. c,symmetric])
    by auto
```

qed

lemma *jumpF-polyR-smult-1*:
jumpF-polyR (*smult c q*) *p x* = *sgn c* ∗ *jumpF-polyR q p x*
proof (*cases c=0*)
case *True*
then show *?thesis* **by** *auto*
next
case *False*
then show *?thesis*
unfolding *jumpF-polyR-def* **using** *False*
apply (*subst jumpF-times*[*of* λ-. *c*,*symmetric*])
by *auto*
qed

lemma
shows *jumpF-polyR-mod*:*jumpF-polyR q p x* = *jumpF-polyR* (*q mod p*) *p x* **and**
jumpF-polyL-mod:*jumpF-polyL q p x* = *jumpF-polyL* (*q mod p*) *p x*
proof −
define *f* **where** *f*=(λ*x*. *poly* (*q div p*) *x*)
define *g* **where** *g*=(λ*x*. *poly* (*q mod p*) *x* / *poly p x*)
have *jumpF-eq*:*jumpF* (λ*x*. *poly q x* / *poly p x*) (*at y within S*) = *jumpF g* (*at y within S*)
when *p*≠*0* **for** *y S*
proof −
let *?F* = *at y within S*
have ∀ *F x in at y within S*. *poly p x* ≠ *0*
using *eventually-poly-nz-at-within*[*OF* ‹*p*≠*0*›,*of y S*] .
then have *eventually* (λ*x*. (*poly q x* / *poly p x*) = (*f x*+ *g x*)) *?F*
proof (*rule eventually-mono*)
fix *x*
assume *P*: *poly p x* ≠ *0*
have *poly q x* = *poly* (*q div p* ∗ *p* + *q mod p*) *x*
by *simp*
also have ... = *f x* ∗ *poly p x* + *poly* (*q mod p*) *x*
by (*simp only*: *poly-add poly-mult f-def g-def*)
moreover have *poly* (*q mod p*) *x* = *g x* ∗ *poly p x*
using *P* **by** (*simp add*: *g-def*)
ultimately show *poly q x* / *poly p x* = *f x* + *g x*
using *P* **by** *simp*
qed
then have *jumpF* (λ*x*. *poly q x* / *poly p x*) *?F* = *jumpF* (λ*x*. *f x*+ *g x*) *?F*
by (*intro jumpF-cong*,*auto*)
also have ... = *jumpF g ?F*
proof −
have (*f* ⟶ *f y*) (*at y within S*)
unfolding *f-def* **by** (*intro tendsto-intros*)
from *filterlim-tendsto-add-at-bot-iff*[*OF this*,*of g*] *filterlim-tendsto-add-at-top-iff*[*OF*

```
this,of g]
      show ?thesis unfolding jumpF-def by auto
    qed
    finally show ?thesis .
  qed
  show jumpF-polyR q p x = jumpF-polyR (q mod p) p x
    apply (cases p=0)
    subgoal by auto
    subgoal using jumpF-eq unfolding g-def jumpF-polyR-def by auto
    done
  show jumpF-polyL q p x = jumpF-polyL (q mod p) p x
    apply (cases p=0)
    subgoal by auto
    subgoal using jumpF-eq unfolding g-def jumpF-polyL-def by auto
    done
qed


lemma jumpF-poly-top-0[simp]: jumpF-poly-top 0 p = 0 jumpF-poly-top q 0 = 0
  unfolding jumpF-poly-top-def by auto

lemma jumpF-poly-bot-0[simp]: jumpF-poly-bot 0 p = 0 jumpF-poly-bot q 0 = 0
  unfolding jumpF-poly-bot-def by auto

lemma jumpF-poly-top-code:
  jumpF-poly-top q p = (if p≠0 ∧ q≠0 ∧ degree q>degree p then
        if sgn-pos-inf q * sgn-pos-inf p > 0 then 1/2 else −1/2 else 0)
proof (cases p≠0 ∧ q≠0 ∧ degree q>degree p)
  case True
  have ?thesis when sgn-pos-inf q * sgn-pos-inf p > 0
  proof −
    have LIM x at-top. poly q x / poly p x :> at-top
      using poly-divide-filterlim-at-top[of p q] True that by auto
    then have jumpF (λx. poly q x / poly p x) at-top = 1/2
      unfolding jumpF-def by auto
    then show ?thesis unfolding jumpF-poly-top-def using that True by auto
  qed
  moreover have ?thesis when ¬ sgn-pos-inf q * sgn-pos-inf p > 0
  proof −
    have LIM x at-top. poly q x / poly p x :> at-bot
      using poly-divide-filterlim-at-top[of p q] True that by auto
    then have jumpF (λx. poly q x / poly p x) at-top = − 1/2
      unfolding jumpF-def by auto
    then show ?thesis unfolding jumpF-poly-top-def using that True by auto
  qed
  ultimately show ?thesis by auto
next
  case False
  define P where P= (¬ (LIM x at-top. poly q x / poly p x :> at-bot)
```

```
                    ∧ ¬ (LIM x at-top. poly q x / poly p x :> at-top))
  have P when p≠0 ∨ q≠0
    unfolding P-def using that
    by (auto elim!:filterlim-at-bot-nhds filterlim-at-top-nhds)
  moreover have P when p≠0 q≠0 degree p> degree q
  proof −
    have LIM x at-top. poly q x / poly p x :> at 0
      using poly-divide-filterlim-at-top[OF that(1,2)] that(3) by auto
    then show ?thesis unfolding P-def
      by (auto elim!:filterlim-at-bot-nhds filterlim-at-top-nhds simp:filterlim-at)
  qed
  moreover have P when p≠0 q≠0 degree p = degree q
  proof −
    have ((λx. poly q x / poly p x) ⟶ lead-coeff q / lead-coeff p) at-top
      using poly-divide-filterlim-at-top[OF that(1,2)] using that by auto
    then show ?thesis unfolding P-def
      by (auto elim!:filterlim-at-bot-nhds filterlim-at-top-nhds)
  qed
  ultimately have P using False by fastforce
  then have jumpF (λx. poly q x / poly p x) at-top = 0
    unfolding jumpF-def P-def by auto
  then show ?thesis unfolding jumpF-poly-top-def using False by presburger
qed

lemma jumpF-poly-bot-code:
  jumpF-poly-bot q p = (if p≠0 ∧ q≠0 ∧ degree q>degree p then
        if sgn-neg-inf q * sgn-neg-inf p > 0 then 1/2 else −1/2 else 0)
proof (cases p≠0 ∧ q≠0 ∧ degree q>degree p)
  case True
  have ?thesis when sgn-neg-inf q * sgn-neg-inf p > 0
  proof −
    have LIM x at-bot. poly q x / poly p x :> at-top
      using poly-divide-filterlim-at-bot[of p q] True that by auto
    then have jumpF (λx. poly q x / poly p x) at-bot = 1/2
      unfolding jumpF-def by auto
    then show ?thesis unfolding jumpF-poly-bot-def using that True by auto
  qed
  moreover have ?thesis when ¬ sgn-neg-inf q * sgn-neg-inf p > 0
  proof −
    have LIM x at-bot. poly q x / poly p x :> at-bot
      using poly-divide-filterlim-at-bot[of p q] True that by auto
    then have jumpF (λx. poly q x / poly p x) at-bot = − 1/2
      unfolding jumpF-def by auto
    then show ?thesis unfolding jumpF-poly-bot-def using that True by auto
  qed
  ultimately show ?thesis by auto
next
  case False
  define P where P= (¬ (LIM x at-bot. poly q x / poly p x :> at-bot)
```

$\wedge \neg$ (*LIM x at-bot. poly q x / poly p x :> at-top*))
have *P* **when** *p=0* $\vee$ *q=0*
unfolding *P-def* **using** *that*
by (*auto elim*!:*filterlim-at-bot-nhds filterlim-at-top-nhds*)
moreover have *P* **when** *p≠0 q≠0 degree p> degree q*
proof −
have *LIM x at-bot. poly q x / poly p x :> at 0*
using *poly-divide-filterlim-at-bot*[*OF that*(*1*,*2*)] *that*(*3*) **by** *auto*
then show *?thesis* **unfolding** *P-def*
by (*auto elim*!:*filterlim-at-bot-nhds filterlim-at-top-nhds simp*:*filterlim-at*)
qed
moreover have *P* **when** *p≠0 q≠0 degree p = degree q*
proof −
have ((λ*x. poly q x / poly p x*) ⟶ *lead-coeff q / lead-coeff p*) *at-bot*
using *poly-divide-filterlim-at-bot*[*OF that*(*1*,*2*)] **using** *that* **by** *auto*
then show *?thesis* **unfolding** *P-def*
by (*auto elim*!:*filterlim-at-bot-nhds filterlim-at-top-nhds*)
qed
ultimately have *P* **using** *False* **by** *fastforce*
then have *jumpF* (λ*x. poly q x / poly p x*) *at-bot = 0*
unfolding *jumpF-def P-def* **by** *auto*
then show *?thesis* **unfolding** *jumpF-poly-bot-def* **using** *False* **by** *presburger*
qed

1.5 The extended Cauchy index on polynomials

definition *cindex-polyE*:: *real* ⇒ *real* ⇒ *real poly* ⇒ *real poly* ⇒ *real* **where**
cindex-polyE a b q p = jumpF-polyR q p a + cindex-poly a b q p − jumpF-polyL q p b

definition *cindex-poly-ubd*::*real poly* ⇒ *real poly* ⇒ *int* **where**
cindex-poly-ubd q p = (*THE l*. (∀ $_F$ *r in at-top. cindexE* (−*r*) *r* (λ*x. poly q x*/*poly p x*) = *of-int l*))

lemma *cindex-polyE-0*[*simp*]: *cindex-polyE a b 0 p = 0 cindex-polyE a b q 0 = 0*
unfolding *cindex-polyE-def* **by** *auto*

lemma *cindex-polyE-mult-cancel*:
fixes *p q p'*::*real poly*
assumes *p'≠ 0*
shows *cindex-polyE a b* (*p' ∗ q*) (*p' ∗ p*) = *cindex-polyE a b q p*
unfolding *cindex-polyE-def*
using *cindex-poly-mult*[*OF* ‹*p'≠0*›] *jumpF-polyL-mult-cancel*[*OF* ‹*p'≠0*›]
jumpF-polyR-mult-cancel[*OF* ‹*p'≠0*›]
by *simp*

lemma *cindexE-eq-cindex-polyE*:
assumes *a<b*
shows *cindexE a b* (λ*x. poly q x*/*poly p x*) = *cindex-polyE a b q p*

proof (*cases p=0 ∨ q=0*)
case *True*
then show *?thesis* **by** (*auto simp add*: *cindexE-constI*)
next
case *False*
then have *p≠0 q≠0* **by** *auto*
define *g* **where** *g=gcd p q*
define *p′ q′* **where** *p′=p div g* **and** *q′ = q div g*
define *f′* **where** *f′=(λx. poly q′ x / poly p′ x)*
have *g≠0* **using** *False g-def* **by** *auto*
have *pq-f*:*p=g∗p′ q=g∗q′* **and** *coprime p′ q′*
unfolding *g-def p′-def q′-def*
apply *simp-all*
using *False div-gcd-coprime* **by** *blast*
have *cindexE a b* (*λx. poly q x/poly p x*) = *cindexE a b* (*λx. poly q′ x/poly p′ x*)
proof −
define *f* **where** *f=(λx. poly q x / poly p x)*
define *f′* **where** *f′=(λx. poly q′ x / poly p′ x)*
have *jumpF f* (*at-right x*) = *jumpF f′* (*at-right x*) **for** *x*
proof (*rule jumpF-cong*)
obtain *ub* **where** *x < ub* ∀ *z. x < z ∧ z ≤ ub ⟶ poly g z ≠ 0*
using *next-non-root-interval*[*OF* ‹*g≠0*›,*of x*] **by** *auto*
then show ∀ *F x in at-right x. f x = f′ x*
unfolding *eventually-at-right f-def f′-def pq-f*
apply (*intro exI*[**where** *x=ub*])
by *auto*
qed *simp*
moreover have *jumpF f* (*at-left x*) = *jumpF f′* (*at-left x*) **for** *x*
proof (*rule jumpF-cong*)
obtain *lb* **where** *lb < x* ∀ *z. lb ≤ z ∧ z < x ⟶ poly g z ≠ 0*
using *last-non-root-interval*[*OF* ‹*g≠0*›,*of x*] **by** *auto*
then show ∀ *F x in at-left x. f x = f′ x*
unfolding *eventually-at-left f-def f′-def pq-f*
apply (*intro exI*[**where** *x=lb*])
by *auto*
qed *simp*
ultimately show *?thesis* **unfolding** *cindexE-def*
apply (*fold f-def f′-def*)
by *auto*
qed
also have ... = *jumpF f′* (*at-right a*) + *real-of-int* (*cindex a b f′*) − *jumpF f′* (*at-left b*)
unfolding *f′-def*
apply (*rule cindex-eq-cindexE-divide*)
subgoal using ‹*a<b*› **.**
subgoal using *False poly-roots-finite pq-f*(*1*) *pq-f*(*2*) **by** *fastforce*
subgoal using ‹*coprime p′ q′*› *poly-gcd-iff* **by** *force*
subgoal by (*auto intro*:*continuous-intros*)

```
    subgoal by (auto intro:continuous-intros)
    done
  also have ... = cindex-polyE a b q' p'
   using cindex-eq-cindex-poly unfolding cindex-polyE-def jumpF-polyR-def jumpF-polyL-def
f'-def
    by auto
  also have ... = cindex-polyE a b q p
    using cindex-polyE-mult-cancel[OF ‹g≠0›] unfolding pq-f by auto
  finally show ?thesis .
qed

lemma cindex-polyE-cross:
  fixes p::real poly and a b::real
  assumes a<b
  shows cindex-polyE a b 1 p = cross-alt 1 p a b / 2
proof (induct degree p arbitrary:p rule:nat-less-induct)
  case induct:1
  have ?case when p=0
    using that unfolding cross-alt-def by auto
  moreover have ?case when p≠0 and noroot:{x.  a< x∧ x< b ∧ poly p x=0
} = {}
  proof −
    have cindex-polyE a b 1 p = jumpF-polyR 1 p a  − jumpF-polyL 1 p b
    proof −
      have cindex-poly a b 1 p = 0 unfolding cindex-poly-def
        apply (rule sum.neutral)
        using that by auto
      then show ?thesis unfolding cindex-polyE-def by auto
    qed
    also have ... = cross-alt 1 p a b / 2
    proof −
      define f where f = (λx. 1 / poly p x)
      define ja where ja = jumpF f (at-right a)
      define jb where jb = jumpF f (at-left b)
      define right where right = (λR. R ja (0::real) ∨ (continuous (at-right a) f
∧ R (poly p a) 0))
      define left where left = (λR. R jb (0::real) ∨ (continuous (at-left b) f ∧ R
(poly p b) 0))

    note ja-alt=jumpF-polyR-coprime[of p 1 a,unfolded jumpF-polyR-def,simplified,folded
f-def ja-def]
    note jb-alt=jumpF-polyL-coprime[of p 1 b,unfolded jumpF-polyL-def,simplified,folded
f-def jb-def]

     have [simp]:0 < ja ⟷ jumpF-polyR 1 p a = 1/2 0 > ja ⟷ jumpF-polyR
1 p a = −1/2
          0 < jb ⟷ jumpF-polyL 1 p b = 1/2 0 > jb ⟷ jumpF-polyL 1 p b =
−1/2
       unfolding ja-def jb-def jumpF-polyR-def jumpF-polyL-def f-def jumpF-def
```

```
    by auto
  have [simp]:
    poly p a ≠0 ⟹ continuous (at-right a) f
    poly p b ≠0 ⟹ continuous (at-left b) f
    unfolding f-def by (auto intro!: continuous-intros )
  have not-right-left: False when (right greater ∧ left less ∨ right less ∧ left
greater)
  proof −
    have [simp]: f a > 0 ⟷ poly p a > 0 f a < 0 ⟷ poly p a < 0
        f b > 0 ⟷ poly p b > 0 f b < 0 ⟷ poly p b < 0
      unfolding f-def by auto
    have continuous-on {a<..<b} f
      unfolding f-def using noroot by (auto intro!: continuous-intros)
    then have ∃x>a. x < b ∧ f x = 0
      apply (elim jumpF-IVT[OF ‹a<b›,of f])
      using that unfolding right-def left-def by (fold ja-def jb-def,auto)
    then show False using noroot using f-def by auto
  qed
  have ?thesis when poly p a>0 ∧ poly p b>0 ∨ poly p a<0 ∧ poly p b<0
    using that jumpF-poly-noroot unfolding cross-alt-def by auto
  moreover have False when poly p a>0 ∧ poly p b<0 ∨ poly p a<0 ∧ poly
p b>0
    apply (rule not-right-left)
    unfolding right-def left-def using that by auto
  moreover have ?thesis when poly p a=0 poly p b>0 ∨ poly p b <0
  proof −
    have ja>0 ∨ ja < 0 using ja-alt ‹p≠0› ‹poly p a=0› by argo
    moreover have False when ja > 0 ∧ poly p b<0 ∨ ja < 0 ∧ poly p b>0
      apply (rule not-right-left)
      unfolding right-def left-def using that by fastforce
    moreover have ?thesis when ja >0 ∧ poly p b>0 ∨ ja < 0 ∧ poly p b<0
      using that jumpF-poly-noroot ‹poly p a=0› unfolding cross-alt-def by
auto
  ultimately show ?thesis using that jumpF-poly-noroot unfolding cross-alt-def
by auto
  qed
  moreover have ?thesis when poly p b=0 poly p a>0 ∨ poly p a <0
  proof −
    have jb>0 ∨ jb < 0 using jb-alt ‹p≠0› ‹poly p b=0› by argo
    moreover have False when jb > 0 ∧ poly p a<0 ∨ jb < 0 ∧ poly p a>0
      apply (rule not-right-left)
      unfolding right-def left-def using that by fastforce
    moreover have ?thesis when jb >0 ∧ poly p a>0 ∨ jb < 0 ∧ poly p a<0
      using that jumpF-poly-noroot ‹poly p b=0› unfolding cross-alt-def by
auto
  ultimately show ?thesis using that jumpF-poly-noroot unfolding cross-alt-def
by auto
  qed
  moreover have ?thesis when poly p a=0 poly p b=0
```

```
      proof −
        have jb>0 ∨ jb < 0 using jb-alt ⟨p≠0⟩ ⟨poly p b=0⟩ by argo
        moreover have ja>0 ∨ ja < 0 using ja-alt ⟨p≠0⟩ ⟨poly p a=0⟩ by argo
        moreover have False when ja>0 ∧ jb<0 ∨ ja<0 ∧ jb>0
          apply (rule not-right-left)
          unfolding right-def left-def using that by fastforce
        moreover have ?thesis when ja>0 ∧ jb>0 ∨ ja<0 ∧ jb<0
          using that jumpF-poly-noroot ⟨poly p b=0⟩ ⟨poly p a=0⟩
          unfolding cross-alt-def by auto
        ultimately show ?thesis by blast
      qed
      ultimately show ?thesis by argo
    qed
    finally show ?thesis .
  qed
  moreover have ?case when p≠0 and no-empty:{x.  a< x∧ x< b ∧ poly p x=0
} ≠ {}
  proof −
    define roots where roots≡{x.  a< x∧ x< b ∧ poly p x=0 }
     have finite roots unfolding roots-def using poly-roots-finite[OF ⟨p≠0⟩] by
auto
    define max-r where max-r≡Max roots
    hence poly p max-r=0 and a<max-r and max-r<b
      using Max-in[OF ⟨finite roots⟩] no-empty  unfolding roots-def by auto
    define max-rp where max-rp≡[:−max-r,1:] ^order max-r p
    then obtain p' where p'-def:p=p'∗max-rp and ¬ [:−max-r,1:] dvd p'
      by (metis ⟨p≠0⟩ mult.commute order-decomp)
    hence p'≠0 and max-rp≠0 and max-r-nz:poly p' max-r≠0

      using ⟨p≠0⟩ by (auto simp add: dvd-iff-poly-eq-0)

   define max-r-sign where max-r-sign≡if odd(order max-r p) then −1 else 1::int
    define roots' where roots'≡{x.  a< x∧ x< b ∧ poly p' x=0}

    have cindex-polyE a b 1 p = jumpF-polyR 1 p a + (∑ x∈roots. jump-poly 1 p
x) − jumpF-polyL 1 p b
      unfolding cindex-polyE-def cindex-poly-def roots-def by (simp,meson)
    also have ... = max-r-sign ∗ cindex-poly a b 1 p' + jump-poly 1 p max-r
       + max-r-sign ∗ jumpF-polyR 1 p' a − jumpF-polyL 1 p' b
    proof −
      have (∑ x∈roots. jump-poly 1 p x) = max-r-sign ∗ cindex-poly a b 1 p' +
jump-poly 1 p max-r
      proof −
        have (∑ x∈roots. jump-poly 1 p x)= (∑ x∈roots'. jump-poly 1 p x)+
jump-poly 1 p max-r
       proof −
         have roots = insert max-r roots'
           unfolding roots-def roots'-def p'-def
           using ⟨poly p max-r=0⟩ ⟨a<max-r⟩ ⟨max-r<b⟩ ⟨p≠0⟩ order-root
```

```
        apply (subst max-rp-def)
        by auto
      moreover have finite roots'
        unfolding roots'-def using poly-roots-finite[OF ‹p'≠0›] by auto
      moreover have max-r ∉ roots'
        unfolding roots'-def using max-r-nz
        by auto
      ultimately show ?thesis using sum.insert[of roots' max-r] by auto
    qed
    moreover have (∑ x∈roots'. jump-poly 1 p x) = max-r-sign * cindex-poly
a b 1 p'
    proof −
     have (∑ x∈roots'. jump-poly 1 p x) = (∑ x∈roots'. max-r-sign * jump-poly
1 p' x)
      proof (rule sum.cong,rule refl)
        fix x assume x ∈ roots'
        hence x≠max-r using max-r-nz unfolding roots'-def
          by auto
         hence poly max-rp x≠0 using poly-power-n-eq unfolding max-rp-def
by auto
        hence order x max-rp=0  by (metis order-root)
        moreover have jump-poly 1 max-rp x=0
          using ‹poly max-rp x≠0› by (metis jump-poly-not-root)
        moreover have x∈roots
          using ‹x ∈ roots'› unfolding roots-def roots'-def p'-def by auto
        hence x<max-r
        using Max-ge[OF ‹finite roots›,of x] ‹x≠max-r› by (fold max-r-def,auto)
        hence sign (poly max-rp x) = max-r-sign
        using ‹poly max-rp x ≠ 0› unfolding max-r-sign-def max-rp-def sign-def
        by (subst poly-power,simp add:linorder-class.not-less zero-less-power-eq)
        ultimately show jump-poly 1 p x = max-r-sign * jump-poly 1 p' x
          using jump-poly-1-mult[of p' x max-rp]  unfolding p'-def
          by (simp add: ‹poly max-rp x ≠ 0›)
      qed
      also have ... = max-r-sign * (∑ x∈roots'. jump-poly 1 p' x)
        by (simp add: sum-distrib-left)
      also have ... = max-r-sign * cindex-poly a b 1 p'
        unfolding cindex-poly-def roots'-def by meson
      finally show ?thesis .
    qed
    ultimately show ?thesis by simp
  qed
  moreover have jumpF-polyR 1 p a = max-r-sign * jumpF-polyR 1 p' a
  proof −
    define f where f = (λx. 1 / poly max-rp x)
    define g where g = (λx. 1 / poly p' x)
    have jumpF-polyR 1 p a = jumpF (λx. f x * g x) (at-right a)
      unfolding jumpF-polyR-def f-def g-def p'-def
      by (auto simp add:field-simps)
```

```
    also have ... = sgn (f a) * jumpF g (at-right a)
    proof (rule jumpF-times)
      have [simp]: poly max-rp a ≠0
        unfolding max-rp-def using ‹max-r>a› by auto
      show (f ⟶ f a) (at-right a) f a ≠ 0
        unfolding f-def by (auto intro:tendsto-intros)
    qed auto
    also have ... = max-r-sign * jumpF-polyR 1 p' a
    proof −
      have sgn (f a) = max-r-sign
        unfolding max-r-sign-def f-def max-rp-def using ‹a<max-r›
        by (auto simp add:sgn-power)
      then show ?thesis unfolding jumpF-polyR-def g-def by auto
    qed
    finally show ?thesis .
  qed
  moreover have jumpF-polyL 1 p b =  jumpF-polyL 1 p' b
  proof −
    define f where f = (λx. 1 / poly max-rp x)
    define g where g = (λx. 1 / poly p' x)
    have jumpF-polyL 1 p b = jumpF (λx. f x * g x) (at-left b)
      unfolding jumpF-polyL-def f-def g-def p'-def
      by (auto simp add:field-simps)
    also have ... = sgn (f b) * jumpF g (at-left b)
    proof (rule jumpF-times)
      have [simp]: poly max-rp b ≠0
        unfolding max-rp-def using ‹max-r<b› by auto
      show (f ⟶ f b) (at-left b) f b ≠ 0
        unfolding f-def by (auto intro:tendsto-intros)
    qed auto
    also have ... = jumpF-polyL 1 p' b
    proof −
      have sgn (f b) = 1
        unfolding max-r-sign-def f-def max-rp-def using ‹b>max-r›
        by (auto simp add:sgn-power)
      then show ?thesis unfolding jumpF-polyL-def g-def by auto
    qed
    finally show ?thesis .
  qed
  ultimately show ?thesis by auto
qed
also have ... = max-r-sign * cindex-polyE a b 1 p' + jump-poly 1 p max-r
    + (max-r-sign − 1) * jumpF-polyL 1 p' b
  unfolding cindex-polyE-def roots'-def by (auto simp add:algebra-simps)
also have ... = max-r-sign * cross-alt 1 p' a b / 2 + jump-poly 1 p max-r
    + (max-r-sign − 1) * jumpF-polyL 1 p' b
proof −
  have degree max-rp>0 unfolding max-rp-def degree-linear-power
    using ‹poly p max-r=0› order-root ‹p≠0› by blast
```

```
    then have degree p'<degree p unfolding p'-def
      using degree-mult-eq[OF ‹p'≠0› ‹max-rp≠0›] by auto
    from induct[rule-format, OF this]
    have cindex-polyE a b 1 p' = real-of-int (cross-alt 1 p' a b) / 2 by auto
    then show ?thesis by auto
   qed
   also have ... = real-of-int (cross-alt 1 p a b) / 2
   proof −
    have sjump-p:jump-poly 1 p max-r = (if odd (order max-r p) then sign (poly
p' max-r) else 0)
    proof −
     note max-r-nz
     moreover then have poly max-rp max-r=0
      using ‹poly p max-r = 0› p'-def by auto
     ultimately have jump-poly 1 p max-r = sign (poly p' max-r) * jump-poly
1 max-rp max-r
      unfolding p'-def using jump-poly-1-mult[of p' max-r max-rp]
      by auto
     also have ... = (if odd (order max-r max-rp) then sign (poly p' max-r) else
0)
     proof −
      have sign-r-pos max-rp max-r
       unfolding max-rp-def using sign-r-pos-power by auto
      then show ?thesis using ‹max-rp≠0› unfolding jump-poly-def by auto
     qed
     also have ... = (if odd (order max-r p) then sign (poly p' max-r) else 0)
     proof −
      have order max-r p'=0 by (simp add: ‹poly p' max-r ≠ 0› order-0I)
      then have order max-r max-rp = order max-r p
       unfolding p'-def using ‹p'≠0› ‹max-rp≠0›
       apply (subst order-mult)
       by auto
      then show ?thesis by auto
     qed
     finally show ?thesis .
    qed
    have ?thesis when even (order max-r p)
    proof −
     have sign (poly p x) =  sign (poly p' x) when x≠max-r for x
     proof −
      have sign (poly max-rp x) = 1
       unfolding max-rp-def using ‹even (order max-r p)› that
       apply (simp add:sign-power )
       by (simp add: Sturm-Tarski.sign-def)
      then show ?thesis unfolding p'-def by (simp add:sign-times)
     qed
     from this[of a] this[of b] ‹a<max-r› ‹max-r<b›
     have cross-alt 1 p' a b = cross-alt 1 p a b
      unfolding cross-alt-def by auto
```

```
    then show ?thesis using that unfolding max-r-sign-def sjump-p by auto
  qed
  moreover have ?thesis when odd (order max-r p)
  proof −
  let ?thesis2 = sign (poly p' max-r) * 2 − cross-alt 1 p' a b − 4 * jumpF-polyL
1 p' b
        = cross-alt 1 p a b
    have ?thesis2 when poly p' b=0
    proof −
      have jumpF-polyL 1 p' b = 1/2 ∨ jumpF-polyL 1 p' b=−1/2
       using jumpF-polyL-coprime[of p' 1 b,simplified] ‹p'≠0› ‹poly p' b=0› by
auto
      moreover have poly p' max-r>0 ∨ poly p' max-r<0
       using max-r-nz by auto
     moreover have False when poly p' max-r>0 ∧ jumpF-polyL 1 p' b=−1/2

          ∨ poly p' max-r<0 ∧ jumpF-polyL 1 p' b=1/2
      proof −
       define f where f= (λx. 1/ poly p' x)
       have noroots:poly p' x≠0 when x∈{max-r<..<b} for x
       proof (rule ccontr)
         assume ¬ poly p' x ≠ 0
         then have poly p x =0 unfolding p'-def by auto
        then have x∈roots unfolding roots-def using that ‹a<max-r› by auto
            then have x≤max-r using Max-ge[OF ‹finite roots›] unfolding
max-r-def by auto
         moreover have x>max-r using that by auto
         ultimately show False by auto
       qed
       have continuous-on {max-r<..<b} f
         unfolding f-def using noroots by (auto intro!:continuous-intros)
       moreover have continuous (at-right max-r) f
         unfolding f-def using max-r-nz
         by (auto intro!:continuous-intros)
       moreover have f max-r>0 ∧ jumpF f (at-left b)<0
          ∨ f max-r<0 ∧ jumpF f (at-left b)>0
         using that unfolding f-def jumpF-polyL-def by auto
       ultimately have ∃ x>max-r. x < b ∧ f x = 0
         apply (intro jumpF-IVT[OF ‹max-r<b›])
         by auto
       then show False using noroots unfolding f-def by auto
      qed
     moreover have ?thesis when poly p' max-r>0 ∧ jumpF-polyL 1 p' b=1/2
         ∨ poly p' max-r<0 ∧ jumpF-polyL 1 p' b=−1/2
      proof −
       have poly max-rp a < 0 poly max-rp b>0
      unfolding max-rp-def using ‹odd (order max-r p)› ‹a<max-r› ‹max-r<b›
         by (simp-all add:zero-less-power-eq)
       then have cross-alt 1 p a b = − cross-alt 1 p' a b
```

```
          unfolding cross-alt-def p'-def using ‹poly p' b=0›
          apply (simp add:sign-times)
          by (simp add: Sturm-Tarski.sign-def)
        with that show ?thesis by auto
      qed
      ultimately show ?thesis by blast
    qed
    moreover have ?thesis2 when poly p' b≠0
    proof −
      have [simp]:jumpF-polyL 1 p' b = 0
        using jumpF-polyL-coprime[of p' 1 b,simplified] ‹poly p' b≠0› by auto
      have [simp]:poly max-rp a < 0 poly max-rp b>0
      unfolding max-rp-def using ‹odd (order max-r p)› ‹a<max-r› ‹max-r<b›
        by (simp-all add:zero-less-power-eq)
      have poly p' b>0 ∨ poly p' b<0
        using ‹poly p' b≠0› by auto
      moreover have poly p' max-r>0 ∨ poly p' max-r<0
        using max-r-nz by auto
      moreover have ?thesis when poly p' b>0 ∧ poly p' max-r>0
        using that unfolding cross-alt-def p'-def
        apply (simp add:sign-times)
        by (simp add: Sturm-Tarski.sign-def)
      moreover have ?thesis when poly p' b<0 ∧ poly p' max-r<0
        using that unfolding cross-alt-def p'-def
        apply (simp add:sign-times)
        by (simp add: Sturm-Tarski.sign-def)
       moreover have False when poly p' b>0 ∧ poly p' max-r<0 ∨ poly p'
b<0 ∧ poly p' max-r>0
      proof −
        have ∃ x>max-r. x < b ∧ poly p' x = 0
          apply (rule poly-IVT[OF ‹max-r<b›,of p'])
          using that mult-less-0-iff by blast
        then obtain x where max-r<x x<b poly p x=0 unfolding p'-def by
auto
        then have x∈roots using ‹a<max-r› unfolding roots-def by auto
          then have x≤max-r unfolding max-r-def using Max-ge[OF ‹finite
roots›] by auto
        then show False using ‹max-r<x› by auto
      qed
      ultimately show ?thesis by blast
    qed
    ultimately have ?thesis2 by auto
    then show ?thesis unfolding max-r-sign-def sjump-p using that by simp
  qed
  ultimately show ?thesis by auto
 qed
 finally show ?thesis .
qed
ultimately show ?case by fast
```

qed

lemma *cindex-polyE-inverse-add*:
fixes *p q*::*real poly*
assumes *cp*:*coprime p q*
shows *cindex-polyE a b q p* + *cindex-polyE a b p q*=*cindex-polyE a b 1* (*q*∗*p*)
unfolding *cindex-polyE-def*
using *cindex-poly-inverse-add*[*OF cp*,*symmetric*] *jumpF-polyR-inverse-add*[*OF cp*,*symmetric*]
jumpF-polyL-inverse-add[*OF cp*,*symmetric*]
by *auto*

lemma *cindex-polyE-inverse-add-cross*:
fixes *p q*::*real poly*
assumes *a* < *b coprime p q*
shows *cindex-polyE a b q p* + *cindex-polyE a b p q* = *cross-alt p q a b* / *2*
apply (*subst cindex-polyE-inverse-add*[*OF* ‹*coprime p q*›])
apply (*subst cindex-polyE-cross*[*OF* ‹*a*<*b*›])
apply (*subst mult.commute*)
apply (*subst cross-alt-clear-n*[*OF* ‹*coprime p q*›])
by *simp*

lemma *cindex-polyE-smult-1*:
fixes *p q*::*real poly* **and** *c*::*real*
shows *cindex-polyE a b* (*smult c q*) *p* = (*sgn c*) ∗ *cindex-polyE a b q p*
unfolding *cindex-polyE-def jumpF-polyL-smult-1 jumpF-polyR-smult-1 cindex-poly-smult-1*

by (*auto simp add*:*sgn-sign-eq*[*symmetric*] *algebra-simps*)

lemma *cindex-polyE-mod*:
fixes *p q*::*real poly*
shows *cindex-polyE a b q p* = *cindex-polyE a b* (*q mod p*) *p*
unfolding *cindex-polyE-def*
apply (*subst cindex-poly-mod*)
apply (*subst jumpF-polyR-mod*)
apply (*subst jumpF-polyL-mod*)
by *simp*

lemma *cindex-polyE-rec*:
fixes *p q*::*real poly*
assumes *a* < *b coprime p q*
shows *cindex-polyE a b q p* = *cross-alt q p a b*/*2* + *cindex-polyE a b* (− (*p mod q*)) *q*
proof −
note *cindex-polyE-inverse-add-cross*[*OF assms*]
moreover have *cindex-polyE a b* (− (*p mod q*)) *q* = − *cindex-polyE a b p q*
using *cindex-polyE-mod cindex-polyE-smult-1*[*of a b* −*1*]
by *auto*
ultimately show *?thesis* **by** (*auto simp add*:*field-simps cross-alt-poly-commute*)

qed

lemma *cindex-polyE-changes-alt-itv-mods*:
assumes *a<b coprime p q*
shows *cindex-polyE a b q p = changes-alt-itv-smods a b p q / 2* **using** ‹*coprime p q*›
proof (*induct smods p q arbitrary:p q*)
case *Nil*
then have *p=0* **by** (*metis smods-nil-eq*)
then show *?case* **by** (*simp add:changes-alt-itv-smods-def changes-alt-poly-at-def*)

next
case (*Cons x xs*)
then have *p≠0* **by** *auto*
have *?case* **when** *q=0*
using *that* **by** (*simp add:changes-alt-itv-smods-def changes-alt-poly-at-def*)
moreover have *?case* **when** *q≠0*
proof −
define *r* **where** *r≡−* (*p mod q*)
obtain *ps* **where** *ps:smods p q=p#q#ps smods q r=q#ps* **and** *xs=q#ps*
unfolding *r-def* **using** ‹*q≠0*› ‹*p≠0*› ‹*x # xs = smods p q*›
by (*metis list.inject smods.simps*)
from *Cons.prems* ‹*q ≠ 0*› **have** *coprime q r*
by (*simp add: r-def ac-simps*)
then have *cindex-polyE a b r q = real-of-int* (*changes-alt-itv-smods a b q r*) / *2*
apply (*rule-tac Cons.hyps*(*1*))
using *ps* ‹*xs=q#ps*› **by** *simp-all*
moreover have *changes-alt-itv-smods a b p q = cross-alt p q a b + changes-alt-itv-smods a b q r*
using *changes-alt-itv-smods-rec*[*OF* ‹*a<b*› ‹*coprime p q*›,*folded r-def*] **.**
moreover have *cindex-polyE a b q p = real-of-int* (*cross-alt q p a b*) / *2 + cindex-polyE a b r q*
using *cindex-polyE-rec*[*OF* ‹*a<b*› ‹*coprime p q*›,*folded r-def*] **.**
ultimately show *?case*
by (*auto simp add:field-simps cross-alt-poly-commute*)
qed
ultimately show *?case* **by** *blast*
qed

lemma *cindex-poly-ubd-eventually*:
shows ∀ *F r in at-top. cindexE* (*−r*) *r* (*λx. poly q x/poly p x*) = *of-int* (*cindex-poly-ubd q p*)
proof −
define *f* **where** *f=*(*λx. poly q x/poly p x*)
obtain *R* **where** *R-def*: *R>0 proots p ⊆* {*−R<..<R*}
if *p≠0*
proof (*cases p=0*)
case *True*

```
  then show ?thesis using that[of 1] by auto
next
  case False
  then have finite (proots p) by auto
  from finite-ball-include[OF this,of 0]
  obtain r where r>0 and r-ball:proots p ⊆ ball 0 r
    by auto
  have proots p ⊆ {−r<..<r}
  proof
    fix x assume x ∈ proots p
    then have x∈ball 0 r using r-ball by auto
    then have abs x<r using mem-ball-0 by auto
    then show x ∈ {− r<..<r} using ‹r>0› by auto
  qed
  then show ?thesis using that[of r] False ‹r>0› by auto
qed
define l where l=(if p=0  then 0 else cindex-poly (−R) R q p)
define P where P=(λl. (∀F r in at-top. cindexE (−r) r f = of-int l))
have P l
proof (cases p=0 )
  case True
  then show ?thesis
    unfolding P-def f-def l-def using True
    by (auto intro!: eventuallyI cindexE-constI)
next
  case False
  have P l unfolding P-def
  proof (rule eventually-at-top-linorderI[of R])
    fix r assume R ≤ r
    then have cindexE (− r) r f =  cindex-polyE (−r) r q p
    unfolding f-def using R-def[OF ‹p≠0›] by (auto intro: cindexE-eq-cindex-polyE)
    also have ... = of-int (cindex-poly (− r) r q p)
    proof −
      have jumpF-polyR q p (− r) = 0
        apply (rule jumpF-poly-noroot)
        using ‹R≤r› R-def[OF ‹p≠0›] by auto
      moreover have jumpF-polyL q p r = 0
        apply (rule jumpF-poly-noroot)
        using ‹R≤r› R-def[OF ‹p≠0›] by auto
      ultimately show ?thesis unfolding cindex-polyE-def by auto
    qed
    also have ... = of-int (cindex-poly (− R) R q p)
    proof −
      define rs where rs={x. poly p x = 0 ∧ − r < x ∧ x < r}
      define Rs where Rs={x. poly p x = 0 ∧ − R < x ∧ x < R}
      have rs=Rs
        using R-def[OF ‹p≠0›] ‹R≤r› unfolding rs-def Rs-def by force
      then show ?thesis
        unfolding cindex-poly-def by (fold rs-def Rs-def,auto)
```

```
    qed
    also have ... = of-int l unfolding l-def using False by auto
    finally show cindexE (− r) r f = real-of-int l .
   qed
   then show ?thesis unfolding P-def by auto
  qed
  moreover have x=l when P x for x
  proof −
   have ∀F r in at-top. cindexE (− r) r f = real-of-int x
       ∀F r in at-top. cindexE (− r) r f = real-of-int l
    using ‹P x› ‹P l› unfolding P-def by auto
   from eventually-conj[OF this]
   have ∀F r::real in at-top. real-of-int x = real-of-int l
    by (elim eventually-mono,auto)
   then have real-of-int x = real-of-int l by auto
   then show ?thesis by simp
  qed
  ultimately have P (THE x. P x) using theI[of P l] by blast
  then show ?thesis unfolding P-def f-def cindex-poly-ubd-def by auto
qed

lemma cindex-poly-ubd-0:
  assumes p=0 ∨ q=0
  shows cindex-poly-ubd q p = 0
proof −
  have ∀F r in at-top. cindexE (−r) r (λx. poly q x/poly p x) = 0
   apply (rule eventuallyI)
   using assms by (auto intro:cindexE-constI)
  from eventually-conj[OF this cindex-poly-ubd-eventually[of q p]]
  have ∀F r::real in at-top. (cindex-poly-ubd q p) = (0::int)
   apply (elim eventually-mono)
   by auto
  then show ?thesis by auto
qed

lemma cindex-poly-ubd-code:
  shows cindex-poly-ubd q p = changes-R-smods p q
proof (cases p=0)
  case True
  then show ?thesis using cindex-poly-ubd-0 by auto
next
  case False
  define ps where ps≡smods p q
  have p∈set ps using ps-def ‹p≠0› by auto
  obtain lb where lb:∀ p∈set ps. ∀ x. poly p x=0 ⟶ x>lb
     and lb-sgn:∀ x≤lb. ∀ p∈set ps. sgn (poly p x) = sgn-neg-inf p
     and lb<0
   using root-list-lb[OF no-0-in-smods,of p q,folded ps-def]
   by auto
```

obtain *ub* **where** *ub*:$\forall\, p{\in}set\ ps.\ \forall\, x.\ poly\ p\ x{=}0 \longrightarrow x{<}ub$
and *ub-sgn*:$\forall\, x{\geq}ub.\ \forall\, p{\in}set\ ps.\ sgn\ (poly\ p\ x) = sgn\text{-}pos\text{-}inf\ p$
and *ub*$>$*0*
using *root-list-ub*[*OF no-0-in-smods*,*of p q*,*folded ps-def*]
by *auto*
define *f* **where** *f*=($\lambda t.$ *poly q t*/ *poly p t*)
define *P* **where** *P*=($\lambda l.$ ($\forall_F$ *r in at-top. cindexE* ($-r$) *r f* = *of-int l*))
have *P* (*changes-R-smods p q*) **unfolding** *P-def*
proof (*rule eventually-at-top-linorderI*[*of max* |*lb*| |*ub*| + *1*])
fix *r* **assume** *r-asm*:*r*$\geq$*max* |*lb*| |*ub*| + *1*
have *cindexE* ($-$ *r*) *r f* = *cindex-polyE* ($-r$) *r q p*
unfolding *f-def* **using** *r-asm* **by** (*auto intro*: *cindexE-eq-cindex-polyE*)
also have ... = *of-int* (*cindex-poly* ($-$ *r*) *r q p*)
proof $-$
have *jumpF-polyR q p* ($-$ *r*) = *0*
apply (*rule jumpF-poly-noroot*)
using *r-asm lb*[*rule-format*,*OF* ‹*p*$\in$*set ps*›,*of* $-r$] **by** *linarith*
moreover have *jumpF-polyL q p r* = *0*
apply (*rule jumpF-poly-noroot*)
using *r-asm ub*[*rule-format*,*OF* ‹*p*$\in$*set ps*›,*of r*] **by** *linarith*
ultimately show *?thesis* **unfolding** *cindex-polyE-def* **by** *auto*
qed
also have ... = *of-int* (*changes-itv-smods* ($-$ *r*) *r p q*)
apply (*rule cindex-poly-changes-itv-mods*[*THEN arg-cong*])
using *r-asm lb*[*rule-format*,*OF* ‹*p*$\in$*set ps*›,*of* $-r$] *ub*[*rule-format*,*OF* ‹*p*$\in$*set ps*›,*of r*]
by *linarith*+
also have ... = *of-int* (*changes-R-smods p q*)
proof $-$
have *map* (*sgn* $\circ$ ($\lambda p.$ *poly p* ($-r$))) *ps* = *map sgn-neg-inf ps*
and *map* (*sgn* $\circ$ ($\lambda p.$ *poly p r*)) *ps* = *map sgn-pos-inf ps*
using *lb-sgn*[*THEN spec*,*of* $-r$,*simplified*] *ub-sgn*[*THEN spec*,*of r*,*simplified*] *r-asm*
by *auto*
hence *changes-poly-at ps* ($-r$)=*changes-poly-neg-inf ps*
$\wedge$ *changes-poly-at ps r*=*changes-poly-pos-inf ps*
unfolding *changes-poly-neg-inf-def changes-poly-at-def changes-poly-pos-inf-def*
by (*subst* (*1 3*) *changes-map-sgn-eq*,*metis map-map*)
thus *?thesis* **unfolding** *changes-R-smods-def changes-itv-smods-def ps-def*
by *metis*
qed
finally show *cindexE* ($-$ *r*) *r f* = *of-int* (*changes-R-smods p q*) **.**
qed
moreover have *x* = *changes-R-smods p q* **when** *P x* **for** *x*
proof $-$
have $\forall_F$ *r in at-top. cindexE* ($-$ *r*) *r f* = *real-of-int* (*changes-R-smods p q*)
$\forall_F$ *r in at-top. cindexE* ($-$ *r*) *r f* = *real-of-int x*
using ‹*P* (*changes-R-smods p q*)› ‹*P x*› **unfolding** *P-def* **by** *auto*
from *eventually-conj*[*OF this*]

```
    have ∀F (r::real) in at-top. of-int x = of-int (changes-R-smods p q)
      by (elim eventually-mono,auto)
    then have of-int x = of-int (changes-R-smods p q)
      using eventually-const-iff by auto
    then show ?thesis using of-int-eq-iff by blast
  qed
  ultimately have (THE x. P x) = changes-R-smods p q
    using the-equality[of P changes-R-smods p q] by blast
  then show ?thesis unfolding cindex-poly-ubd-def P-def f-def by auto
qed


lemma cindexE-ubd-poly: cindexE-ubd (λx. poly q x/poly p x) = cindex-poly-ubd
q p
proof (cases p=0)
  case True
  then show ?thesis using cindex-poly-ubd-0 unfolding cindexE-ubd-def
    by auto
next
  case False
  define mx mn where mx = Max {x. poly p x = 0} and mn = Min {x. poly p
x=0}
  define rr where rr= 1+ (max |mx| |mn|)
  have rr:−rr < x ∧ x< rr when poly p x = 0  for x
  proof −
    have finite {x. poly p x = 0} using ‹p≠0› poly-roots-finite by blast
    then have mn ≤ x x≤mx
      using Max-ge Min-le that unfolding mn-def mx-def by simp-all
    then show ?thesis unfolding rr-def by auto
  qed
  define f where f=(λx. poly q x / poly p x)
  have ∀F r in at-top. cindexE (− r) r f = cindexE-ubd f
  proof (rule eventually-at-top-linorderI[of rr])
    fix r assume r≥rr
    define R1 R2 where R1={x. jumpF f (at-right x) ≠ 0 ∧ − r ≤ x ∧ x < r}
                   and R2 = {x. jumpF f (at-right x) ≠ 0}
    define L1 L2 where L1={x. jumpF f (at-left x) ≠ 0 ∧ − r < x ∧ x ≤ r}
                   and L2={x. jumpF f (at-left x) ≠ 0}
    have R1=R2
    proof −
      have jumpF f (at-right x) = 0 when ¬ (− r ≤ x ∧ x < r) for x
      proof −
        have jumpF f (at-right x) = jumpF-polyR q p x
          unfolding f-def jumpF-polyR-def by simp
        also have ... = 0
          apply (rule jumpF-poly-noroot)
          using  that ‹r≥rr› by (auto dest:rr)
        finally show ?thesis .
      qed
```

```
    then show ?thesis unfolding R1-def R2-def by blast
   qed
   moreover have L1=L2
   proof −
    have jumpF f (at-left x) = 0 when ¬ (− r < x ∧ x ≤ r) for x
    proof −
     have jumpF f (at-left x) = jumpF-polyL q p x
      unfolding f-def jumpF-polyL-def by simp
     also have ... = 0
      apply (rule jumpF-poly-noroot)
      using that ‹r≥rr› by (auto dest:rr)
     finally show ?thesis .
    qed
    then show ?thesis unfolding L1-def L2-def by blast
   qed
   ultimately show cindexE (− r) r f = cindexE-ubd f
    unfolding cindexE-def cindexE-ubd-def
    apply (fold R1-def R2-def L1-def L2-def)
    by auto
  qed
  moreover have ∀F r in at-top. cindexE (− r) r f = cindex-poly-ubd q p
   using cindex-poly-ubd-eventually unfolding f-def by auto
  ultimately have ∀F r in at-top. cindexE (− r) r f = cindexE-ubd f
                                  ∧ cindexE (− r) r f = cindex-poly-ubd q p
   using eventually-conj by auto
  then have ∀F (r::real) in at-top. cindexE-ubd f = cindex-poly-ubd q p
   by (elim eventually-mono) auto
  then show ?thesis unfolding f-def by auto
qed

end
```

2 More useful lemmas related polynomials

```
theory More-Polynomials imports
  Winding-Number-Eval.Missing-Algebraic
  Winding-Number-Eval.Missing-Transcendental
  Sturm-Tarski.PolyMisc
  Budan-Fourier.BF-Misc
begin
```

2.1 More about *order*

```
lemma order-normalize[simp]:order x (normalize p) = order x p
 by (metis dvd-normalize-iff normalize-eq-0-iff order-1 order-2 order-unique-lemma)

lemma order-gcd:
  assumes p≠0 q≠0
  shows order x (gcd p q) = min (order x p) (order x q)
```

```
proof −
  define xx op oq where xx=[:− x, 1:] and op = order x p and oq = order x q
  obtain pp where pp:p = xx ^ op * pp ¬ xx dvd pp
    using order-decomp[OF ‹p≠0›,of x,folded xx-def op-def] by auto
  obtain qq where qq:q = xx ^ oq * qq ¬ xx dvd qq
    using order-decomp[OF ‹q≠0›,of x,folded xx-def oq-def] by auto
  define pq where pq = gcd pp qq

  have p-unfold:p = (pq * xx ^ (min op oq)) * ((pp div pq) * xx ^ (op − min op
oq))
        and [simp]:coprime xx (pp div pq) and pp≠0
  proof −
    have xx ^ op = xx ^ (min op oq) * xx ^ (op − min op oq)
      by (simp flip:power-add)
    moreover have pp = pq * (pp div pq)
      unfolding pq-def by simp
   ultimately show p = (pq * xx ^ (min op oq)) * ((pp div pq) * xx ^ (op − min
op oq))
      unfolding pq-def pp by(auto simp:algebra-simps)
    show coprime xx (pp div pq)
      apply (rule prime-elem-imp-coprime[OF
                  prime-elem-linear-poly[of 1 −x,simplified],folded xx-def])
      using ‹pp = pq * (pp div pq)› pp(2) by auto
  qed (use pp ‹p≠0› in auto)
  have q-unfold:q = (pq * xx ^ (min op oq)) * ((qq div pq) * xx ^ (oq − min op
oq))
        and [simp]:coprime xx (qq div pq)
  proof −
    have xx ^ oq = xx ^ (min op oq) * xx ^ (oq − min op oq)
      by (simp flip:power-add)
    moreover have qq = pq * (qq div pq)
      unfolding pq-def by simp
    ultimately show q = (pq * xx ^ (min op oq)) * ((qq div pq) * xx ^ (oq − min
op oq))
      unfolding pq-def qq by(auto simp:algebra-simps)
    show coprime xx (qq div pq)
      apply (rule prime-elem-imp-coprime[OF
                  prime-elem-linear-poly[of 1 −x,simplified],folded xx-def])
      using ‹qq = pq * (qq div pq)› qq(2) by auto
  qed

  have gcd p q=normalize (pq * xx ^ (min op oq))
  proof −
    have coprime (pp div pq * xx ^ (op − min op oq)) (qq div pq * xx ^ (oq − min
op oq))
    proof (cases op>oq)
      case True
      then have oq − min op oq = 0 by auto
      moreover have coprime (xx ^ (op − min op oq)) (qq div pq) by auto
```

```
    moreover have coprime (pp div pq) (qq div pq)
      apply (rule div-gcd-coprime[of pp qq,folded pq-def])
      using ‹pp≠0› by auto
    ultimately show ?thesis by auto
  next
    case False
    then have op − min op oq = 0 by auto
    moreover have coprime (pp div pq) (xx ^ (oq − min op oq))
      by (auto simp:coprime-commute)
    moreover have coprime (pp div pq) (qq div pq)
      apply (rule div-gcd-coprime[of pp qq,folded pq-def])
      using ‹pp≠0› by auto
    ultimately show ?thesis by auto
  qed
  then show ?thesis unfolding p-unfold q-unfold
    apply (subst gcd-mult-left)
    by auto
 qed
 then have order x (gcd p q) = order x pq + order x (xx ^ (min op oq))
  apply simp
  apply (subst order-mult)
  using assms(1) p-unfold by auto
 also have ... = order x (xx ^ (min op oq))
  using pp(2) qq(2) unfolding pq-def xx-def
  by (auto simp add: order-0I poly-eq-0-iff-dvd)
 also have ... = min op oq
  unfolding xx-def by (rule order-power-n-n)
 also have ... = min (order x p) (order x q) unfolding op-def oq-def by simp
 finally show ?thesis .
qed

lemma pderiv-power: pderiv (p ^ n) = smult (of-nat n) (p ^ (n−1)) ∗ pderiv p
 apply (cases n)
 using pderiv-power-Suc by auto


lemma order-pderiv:
 fixes p::'a::{idom,semiring-char-0} poly
 assumes p≠0 poly p x=0
 shows order x p = Suc (order x (pderiv p)) using assms
proof −
 define xx op where xx=[:− x, 1:] and op = order x p
 have op ≠0 unfolding op-def using assms order-root by blast
 obtain pp where pp:p = xx ^ op ∗ pp ¬ xx dvd pp
  using order-decomp[OF ‹p≠0›,of x,folded xx-def op-def] by auto
 have p-der:pderiv p = smult (of-nat op) (xx^(op −1)) ∗ pp + xx^op∗pderiv pp
  unfolding pp(1) by (auto simp:pderiv-mult pderiv-power xx-def algebra-simps
pderiv-pCons)
 have xx^(op −1) dvd (pderiv p)
```

unfolding *p-der*
by (*metis One-nat-def Suc-pred assms(1) assms(2) dvd-add dvd-mult-right dvd-triv-left*
neq0-conv op-def order-root power-Suc smult-dvd-cancel)
moreover have ¬ *xx^op dvd* (*pderiv p*)
proof
assume *xx ^ op dvd pderiv p*
then have *xx ^ op dvd smult* (*of-nat op*) (*xx^*(*op −1*) ∗ *pp*)
unfolding *p-der* **by** (*simp add*: *dvd-add-left-iff*)
then have *xx ^ op dvd* (*xx^*(*op −1*)) ∗ *pp*
apply (*elim dvd-monic*[*rotated*])
using ‹*op≠0*› **by** (*auto simp*:*lead-coeff-power xx-def*)
then have *xx ^* (*op−1*) ∗ *xx dvd* (*xx^*(*op −1*))
using ‹¬ *xx dvd pp*› **by** (*simp add*: ‹*op* ≠ *0*› *mult.commute power-eq-if*)
then have *xx dvd 1*
using *assms*(*1*) *pp*(*1*) **by** *auto*
then show *False* **unfolding** *xx-def* **by** (*meson assms*(*1*) *dvd-trans one-dvd order-decomp*)
qed
ultimately have *op − 1 = order x* (*pderiv p*)
using *order-unique-lemma*[*of x op−1 pderiv p*,*folded xx-def*] ‹*op≠0*›
by *auto*
then show *?thesis* **using** ‹*op≠0*› **unfolding** *op-def* **by** *auto*
qed

2.2 More about *rsquarefree*

lemma *rsquarefree-0*[*simp*]: ¬ *rsquarefree 0*
unfolding *rsquarefree-def* **by** *simp*

lemma *rsquarefree-times*:
assumes *rsquarefree* (*p∗q*)
shows *rsquarefree q* **using** *assms*
proof (*induct p rule*:*poly-root-induct-alt*)
case *0*
then show *?case* **by** *simp*
next
case (*no-proots p*)
then have [*simp*]:*p≠0 q≠0* ⋀*a. order a p = 0*
using *order-0I* **by** *auto*
have *order a* (*p ∗ q*) = *0* ⟷ *order a q* = *0*
order a (*p ∗ q*) = *1* ⟷ *order a q* = *1*
for *a*
subgoal by (*subst order-mult*) *auto*
subgoal by (*subst order-mult*) *auto*
done
then show *?case* **using** ‹*rsquarefree* (*p ∗ q*)›
unfolding *rsquarefree-def* **by** *simp*
next

```
  case (root a p)
  define pq aa where pq = p * q and aa = [:- a, 1:]
  have [simp]:pq≠0 aa≠0 order a aa=1
    subgoal using pq-def root.prems by auto
    subgoal by (simp add: aa-def)
    subgoal by (metis aa-def order-power-n-n power-one-right)
    done
  have rsquarefree (aa * pq)
    unfolding aa-def pq-def using root(2) by (simp add:algebra-simps)
  then have rsquarefree pq
    unfolding rsquarefree-def by (auto simp add:order-mult)
  from root(1)[OF this[unfolded pq-def]] show ?case .
qed

lemma rsquarefree-smult-iff:
  assumes s≠0
  shows rsquarefree (smult s p) ⟷ rsquarefree p
  unfolding rsquarefree-def using assms by (auto simp add:order-smult)

lemma card-proots-within-rsquarefree:
  assumes rsquarefree p
  shows proots-count p s = card (proots-within p s) using assms
proof (induct rule:poly-root-induct[of - λx. x∈s])
  case 0
  then have False by simp
  then show ?case by simp
next
  case (no-roots p)
  then show ?case
   by (metis all-not-in-conv card-empty proots-count-def proots-within-iff sum.empty)
next
  case (root a p)
  have proots-count ([:a, - 1:] * p) s = 1 + proots-count p s
    apply (subst proots-count-times)
    subgoal using root.prems rsquarefree-def by blast
   subgoal by (metis (no-types, hide-lams) add.inverse-inverse add.inverse-neutral

                        minus-pCons  proots-count-pCons-1-iff  proots-count-uminus
root.hyps(1))
    done
  also have ... = 1 + card (proots-within p s)
  proof -
    have rsquarefree p using ‹rsquarefree ([:a, - 1:] * p)›
      by (elim rsquarefree-times)
    from root(2)[OF this] show ?thesis by simp
  qed
 also have ... = card (proots-within ([:a, - 1:] * p) s) unfolding proots-within-times

  proof (subst card-Un-disjoint)
```

```
    have [simp]:p≠0 using root.prems by auto
    show finite (proots-within [:a, − 1:] s) finite (proots-within p s)
      by auto
    show  1 + card (proots-within p s) = card (proots-within [:a, − 1:] s)
            + card (proots-within p s)
      using ‹a ∈ s›
      apply (subst proots-within-pCons-1-iff)
      by simp
    have poly p a≠0
    proof (rule ccontr)
      assume ¬ poly p a ≠ 0
      then have order a p >0 by (simp add: order-root)
      moreover have order a [:a,−1:] = 1
          by (metis (no-types, hide-lams) add.inverse-inverse add.inverse-neutral
minus-pCons
           order-power-n-n order-uminus power-one-right)
      ultimately have order a  ([:a, − 1:] ∗ p) > 1
        apply (subst order-mult)
        subgoal using root.prems by auto
        subgoal by auto
        done
      then show False using ‹rsquarefree ([:a, − 1:] ∗ p)›
        unfolding rsquarefree-def using gr-implies-not0 less-not-refl2 by blast
    qed
    then show  proots-within [:a, − 1:] s ∩ proots-within p s = {}
      using proots-within-pCons-1-iff(2) by auto
  qed
  finally show ?case .
qed

lemma rsquarefree-gcd-pderiv:
  fixes p::'a::{factorial-ring-gcd,semiring-char-0} poly
  assumes p≠0
  shows rsquarefree (p div (gcd p (pderiv p)))
proof (cases pderiv p = 0)
  case True
  have poly (unit-factor p) x ≠0 for x
    using unit-factor-is-unit[OF ‹p≠0›]
    by (meson assms dvd-trans order-decomp poly-eq-0-iff-dvd unit-factor-dvd)
  then have order x (unit-factor p) = 0 for x
    using order-0I by blast
  then show ?thesis using True ‹p≠0› unfolding rsquarefree-def by simp
next
  case False
  define q where q = p div (gcd p (pderiv p))
  have q≠0 unfolding q-def by (simp add: assms dvd-div-eq-0-iff)

  have order-pq:order x p = order x q + min (order x p) (order x (pderiv p))
    for x
```

```
  proof −
    have ∗:p = q ∗ gcd p (pderiv p)
      unfolding q-def by simp
    show ?thesis
      apply (subst ∗)
      using ‹q≠0› ‹p≠0› ‹pderiv p≠0› by (simp add:order-mult order-gcd)
  qed
  have order x q = 0 ∨ order x q=1 for x
  proof (cases poly p x=0)
    case True
    from order-pderiv[OF ‹p≠0› this]
    have order x p = order x (pderiv p) + 1 by simp
    then show ?thesis using order-pq[of x] by auto
  next
    case False
    then have order x p = 0 by (simp add: order-0I)
    then have order x q = 0 using order-pq[of x] by simp
    then show ?thesis by simp
  qed
  then show ?thesis using ‹q≠0› unfolding rsquarefree-def q-def
    by auto
qed

lemma poly-gcd-pderiv-iff:
  fixes p::'a::{semiring-char-0,factorial-ring-gcd} poly
  shows poly (p div (gcd p (pderiv p))) x =0 ⟷ poly p x=0
proof (cases pderiv p=0)
  case True
  then obtain a where p=[:a:] using pderiv-iszero by auto
  then show ?thesis by (auto simp add: unit-factor-poly-def)
next
  case False
  then have p≠0 using pderiv-0 by blast
  define q where q = p div (gcd p (pderiv p))
  have q≠0 unfolding q-def by (simp add: ‹p≠0› dvd-div-eq-0-iff)

  have order-pq:order x p = order x q + min (order x p) (order x (pderiv p)) for
x
  proof −
    have ∗:p = q ∗ gcd p (pderiv p)
      unfolding q-def by simp
    show ?thesis
      apply (subst ∗)
      using ‹q≠0› ‹p≠0› ‹pderiv p≠0› by (simp add:order-mult order-gcd)
  qed

  have order x q =0 ⟷ order x p = 0
  proof (cases poly p x=0)
    case True
```

from *order-pderiv*[*OF* ‹*p*≠*0*› *this*]
have *order x p* = *order x* (*pderiv p*) + *1* **by** *simp*
then show *?thesis* **using** *order-pq*[*of x*] **by** *auto*
next
case *False*
then have *order x p* = *0* **by** (*simp add*: *order-0I*)
then have *order x q* = *0* **using** *order-pq*[*of x*] **by** *simp*
then show *?thesis* **using** ‹*order x p* = *0*› **by** *simp*
qed
then show *?thesis*
apply (*fold q-def*)
unfolding *order-root* **using** ‹*p*≠*0*› ‹*q*≠*0*› **by** *auto*
qed

2.3 Composition of a polynomial and a circular path

lemma *poly-circlepath-tan-eq*:
fixes *z0*::*complex* **and** *r*::*real* **and** *p*::*complex poly*
defines *q1*≡ *fcompose p* [:(*z0*+*r*)∗i,*z0*−*r*:] [:i,*1*:] **and** *q2* ≡ [:i,*1*:] ^ *degree p*
assumes *0*≤*t t*≤*1 t*≠*1*/*2*
shows *poly p* (*circlepath z0 r t*) = *poly q1* (*tan* (*pi*∗*t*)) / *poly q2* (*tan* (*pi*∗*t*))
(**is** *?L* = *?R*)
proof −
have *?L* = *poly p* (*z0*+ *r*∗*exp* (*2* ∗ *of-real pi* ∗ i ∗ *t*))
unfolding *circlepath* **by** *simp*
also have ... = *?R*
proof −
define *f* **where** *f* = (*poly p* ∘ (λ*x*::*real*. *z0* + *r* ∗ *exp* (i ∗ *x*)))
have *f-eq*:*f t* = ((λ*x*::*real*. *poly q1 x* / *poly q2 x*) *o* (λ*x*. *tan* (*x*/*2*))) *t*
when *cos* (*t* / *2*) ≠ *0* **for** *t*
proof −
have *f t* = *poly p* (*z0* + *r* ∗ (*cos t* + i ∗ *sin t*))
unfolding *f-def exp-Euler* **by** (*auto simp add*:*cos-of-real sin-of-real*)
also have ... = *poly p* ((λ*x*. ((*z0*−*r*)∗*x*+(*z0*+*r*)∗i) / (i+*x*)) (*tan* (*t*/*2*)))
proof −
define *tt* **where** *tt*=*complex-of-real* (*tan* (*t* / *2*))
define *rr* **where** *rr* = *complex-of-real r*
have *cos t* = (*1*−*tt*∗*tt*) / (*1* + *tt* ∗ *tt*)
sin t = *2*∗*tt* / (*1* + *tt* ∗ *tt*)
unfolding *sin-tan-half*[*of t*/*2*,*simplified*] *cos-tan-half*[*of t*/*2*,*OF that*, *simplified*] *tt-def*
by (*auto simp add*:*power2-eq-square*)
moreover have *1* + *tt* ∗ *tt* ≠ *0* **unfolding** *tt-def*
apply (*fold of-real-mult*)
by (*metis* (*no-types*, *hide-lams*) *mult-numeral-1 numeral-One of-real-add of-real-eq-0-iff*
of-real-numeral sum-squares-eq-zero-iff zero-neq-one)
ultimately have *z0* + *r* ∗ ((*cos t*) + i ∗ (*sin t*))
=(*z0*∗(*1*+*tt*∗*tt*)+*rr*∗(*1*−*tt*∗*tt*)+i∗*rr*∗*2*∗*tt*) / (*1* + *tt* ∗ *tt*)

```
      apply (fold rr-def,simp add:add-divide-distrib)
      by (simp add:algebra-simps)
    also have ... = ((z0−rr)∗tt+z0∗i+rr∗i) / (tt + i)
    proof −
      have tt + i ≠ 0
        using ‹1 + tt ∗ tt ≠ 0› by (metis i-squared neg-eq-iff-add-eq-0 square-eq-iff)
      then show ?thesis
        using ‹1 + tt ∗ tt ≠ 0› by (auto simp add:divide-simps algebra-simps)
    qed
    finally have z0 +  r ∗ ( (cos t) + i ∗ (sin t)) = ((z0−rr)∗tt+z0∗i+rr∗i)
/ (tt + i) .
    then show ?thesis unfolding tt-def rr-def
      by (auto simp add:algebra-simps power2-eq-square)
  qed
   also have ... = (poly p o ((λx. ((z0−r)∗x+(z0+r)∗i) / (i+x)) o (λx. tan
(x/2)) )) t
    unfolding comp-def by (auto simp:tan-of-real)
  also have ... = ((λx::real. poly q1 x / poly q2 x) o  (λx. tan (x/2)) ) t
    unfolding q2-def q1-def
    apply (subst fcompose-poly[symmetric])
    subgoal for x
      apply simp
          by (metis Re-complex-of-real add-cancel-right-left complex-i-not-zero
imaginary-unit.sel(1) plus-complex.sel(1) rcis-zero-arg rcis-zero-mod)
    subgoal by (auto simp:tan-of-real algebra-simps)
    done
  finally show ?thesis .
 qed

 have cos (pi ∗ t) ≠0 unfolding cos-zero-iff-int2
 proof
  assume ∃ i. pi ∗ t = real-of-int i ∗ pi + pi / 2
  then obtain i where pi ∗ t = real-of-int i ∗ pi + pi / 2 by auto
  then have pi ∗ t=pi ∗ (real-of-int i + 1 / 2) by (simp add:algebra-simps)
  then have t=real-of-int i + 1 / 2 by auto
  then show False using ‹0≤t› ‹t≤1› ‹t≠1/2› by auto
 qed
 from f-eq[of 2∗pi∗t,simplified,OF this]
 show ?thesis
  unfolding f-def comp-def by (auto simp add:algebra-simps)
qed
finally show ?thesis .
qed

end
```

3 Procedures to count the number of complex roots

theory *Count-Complex-Roots* **imports**
Winding-Number-Eval.Winding-Number-Eval
Extended-Sturm
More-Polynomials
Budan-Fourier.Sturm-Multiple-Roots
begin

3.1 Misc

corollary *path-image-part-circlepath-subset*:
assumes $r \geq 0$
shows *path-image*(*part-circlepath z r st tt*) $\subseteq$ *sphere z r*
proof (*cases st*$\leq$*tt*)
case *True*
then show *?thesis*
by (*auto simp*: *assms path-image-part-circlepath sphere-def dist-norm algebra-simps norm-mult*)
next
case *False*
then have *path-image*(*part-circlepath z r tt st*) $\subseteq$ *sphere z r*
by (*auto simp*: *assms path-image-part-circlepath sphere-def dist-norm algebra-simps norm-mult*)
moreover have *path-image*(*part-circlepath z r tt st*) = *path-image*(*part-circlepath z r st tt*)
using *path-image-reversepath* **by** *fastforce*
ultimately show *?thesis* **by** *auto*
qed

proposition *in-path-image-part-circlepath*:
assumes $w \in$ *path-image*(*part-circlepath z r st tt*) $0 \leq r$
shows *norm*$(w - z) = r$
proof −
have $w \in \{c.\ dist\ z\ c = r\}$
by (*metis* (*no-types*) *path-image-part-circlepath-subset sphere-def subset-eq assms*)
thus *?thesis*
by (*simp add*: *dist-norm norm-minus-commute*)
qed

lemma *infinite-ball*:
fixes *a* :: *'a*::*euclidean-space*
assumes $r > 0$
shows *infinite* (*ball a r*)
using *uncountable-ball*[*OF assms*, *THEN uncountable-infinite*] **.**

lemma *infinite-cball*:
fixes *a* :: *'a*::*euclidean-space*
assumes $r > 0$

shows *infinite* (*cball a r*)
using *uncountable-cball*[*OF assms*, *THEN uncountable-infinite*,*of a*] **.**

lemma *infinite-sphere*:
fixes *a* :: *complex*
assumes *r* > *0*
shows *infinite* (*sphere a r*)
proof −
have *uncountable* (*path-image* (*circlepath a r*))
apply (*rule simple-path-image-uncountable*)
using *simple-path-circlepath assms* **by** *simp*
then have *uncountable* (*sphere a r*)
using *assms* **by** *simp*
from *uncountable-infinite*[*OF this*] **show** *?thesis* **.**
qed

lemma *infinite-halfspace-Im-gt*: *infinite* {*x*. *Im x* > *b*}
apply (*rule connected-uncountable*[*THEN uncountable-infinite*,*of* - (*b*+*1*)∗ i (*b*+*2*)∗i])
by (*auto intro*!:*convex-connected simp add*: *convex-halfspace-Im-gt*)

lemma (**in** *ring-1*) *Ints-minus2*: − *a* ∈ ℤ ⟹ *a* ∈ ℤ
using *Ints-minus*[*of* −*a*] **by** *auto*

lemma *dvd-divide-Ints-iff*:
b dvd a ∨ *b*=*0* ⟷ *of-int a* / *of-int b* ∈ (ℤ :: *'a* :: {*field*,*ring-char-0*} *set*)
proof
assume *asm*:*b dvd a* ∨ *b*=*0*
let *?thesis* = *of-int a* / *of-int b* ∈ (ℤ :: *'a* :: {*field*,*ring-char-0*} *set*)
have *?thesis* **when** *b dvd a*
proof −
obtain *c* **where** *a*=*b* ∗ *c* **using** ‹*b dvd a*› **unfolding** *dvd-def* **by** *auto*
then show *?thesis* **by** (*auto simp add*:*field-simps*)
qed
moreover have *?thesis* **when** *b*=*0*
using *that* **by** *auto*
ultimately show *?thesis* **using** *asm* **by** *auto*
next
assume *of-int a* / *of-int b* ∈ (ℤ :: *'a* :: {*field*,*ring-char-0*} *set*)
from *Ints-cases*[*OF this*] **obtain** *c* **where** ∗:(*of-int*::- ⇒ *'a*) *c*= *of-int a* / *of-int*
b
by *metis*
have *b dvd a* **when** *b*≠*0*
proof −
have (*of-int*::- ⇒ *'a*) *a* = *of-int b* ∗ *of-int c* **using** *that* ∗ **by** *auto*
then have *a* = *b* ∗ *c* **using** *of-int-eq-iff* **by** *fastforce*
then show *?thesis* **unfolding** *dvd-def* **by** *auto*
qed
then show *b dvd a* ∨ *b* = *0* **by** *auto*

qed

lemma *of-int-div-field*:
assumes *d dvd n*
shows (*of-int*::-⇒'*a*::*field-char-0*) (*n div d*) = *of-int n* / *of-int d*
apply (*subst* (*2*) *dvd-mult-div-cancel*[*OF assms*,*symmetric*])
by (*auto simp add*:*field-simps*)

lemma *powr-eq-1-iff*:
assumes *a*>*0*
shows (*a*::*real*) *powr b* =*1* ⟷ *a*=*1* ∨ *b*=*0*
proof
assume *a powr b* = *1*
have *b* ∗ *ln a* = *0*
using ‹*a powr b* = *1*› *ln-powr*[*of a b*] *assms* **by** *auto*
then have *b*=*0* ∨ *ln a* =*0* **by** *auto*
then show *a* = *1* ∨ *b* = *0* **using** *assms* **by** *auto*
qed (*insert assms*, *auto*)

lemma *tan-inj-pi*:
− (*pi*/*2*) < *x* ⟹ *x* < *pi*/*2* ⟹ − (*pi*/*2*) < *y* ⟹ *y* < *pi*/*2* ⟹ *tan x* = *tan y* ⟹ *x* = *y*
by (*metis arctan-tan*)

lemma *finite-ReZ-segments-poly-circlepath*:
finite-ReZ-segments (*poly p* ∘ *circlepath z0 r*) *0*
proof (*cases* ∀ *t*∈({*0*..*1*} − {*1*/*2*}). *Re* ((*poly p* ∘ *circlepath z0 r*) *t*) = *0*)
case *True*
have *isCont* (*Re* ∘ *poly p* ∘ *circlepath z0 r*) (*1*/*2*)
by (*auto intro*!:*continuous-intros simp*:*circlepath*)
moreover have (*Re* ∘ *poly p* ∘ *circlepath z0 r*)− *1*/*2* → *0*
proof −
have ∀ *F x in at* (*1* / *2*). (*Re* ∘ *poly p* ∘ *circlepath z0 r*) *x* = *0*
unfolding *eventually-at-le*
apply (*rule exI*[**where** *x*=*1*/*2*])
unfolding *dist-real-def abs-diff-le-iff*
by (*auto intro*!:*True*[*rule-format*, *unfolded comp-def*])
then show *?thesis* **by** (*rule Lim-eventually*)
qed
ultimately have *Re* ((*poly p* ∘ *circlepath z0 r*) (*1*/*2*)) = *0*
unfolding *comp-def* **by** (*simp add*: *LIM-unique continuous-within*)
then have ∀ *t*∈{*0*..*1*}. *Re* ((*poly p* ∘ *circlepath z0 r*) *t*) = *0*
using *True* **by** *blast*
then show *?thesis*
apply (*rule-tac finite-ReZ-segments-constI*[*THEN finite-ReZ-segments-congE*])
by *auto*
next
case *False*

```
 define q1 q2 where q1=fcompose p [:(z0+r)*i,z0-r:] [:i,1:] and
                   q2=([:i, 1:] ^ degree p)
 define q1R q1I where q1R=map-poly Re q1 and q1I=map-poly Im q1
 define q2R q2I where q2R=map-poly Re q2 and q2I=map-poly Im q2
 define qq where qq=q1R*q2R + q1I*q2I

 have poly-eq:Re ((poly p ∘ circlepath z0 r) t) = 0 ⟷ poly qq (tan (pi * t)) =
0
   when 0≤t t≤1 t≠1/2 for t
 proof −
   define tt where tt=tan (pi * t)
   have Re ((poly p ∘ circlepath z0 r) t) = 0 ⟷ Re (poly q1 tt / poly q2 tt) = 0
     unfolding comp-def
     apply (subst poly-circlepath-tan-eq[of t p z0 r,folded q1-def q2-def tt-def])
     using that by simp-all
   also have ... ⟷ poly q1R tt * poly q2R tt + poly q1I tt * poly q2I tt = 0
     unfolding q1I-def q1R-def q2R-def q2I-def
     by (simp add:Re-complex-div-eq-0 Re-poly-of-real Im-poly-of-real)
   also have ... ⟷ poly qq tt = 0
     unfolding qq-def by simp
   finally show ?thesis unfolding tt-def .
 qed

 have finite {t. Re ((poly p ∘ circlepath z0 r) t) = 0 ∧ 0 ≤ t ∧ t ≤ 1}
 proof −
   define P where P=(λt. Re ((poly p ∘ circlepath z0 r) t) = 0)
   define A where A= ({0..1}::real set)
   define S where S={t∈A-{1,1/2}. P t}
   have finite {t. poly qq (tan (pi * t)) = 0 ∧ 0 ≤ t ∧ t < 1 ∧ t≠1/2}
   proof −
     define A where A={t::real. 0 ≤ t ∧ t < 1 ∧ t ≠ 1 / 2}
     have finite ((λt. tan (pi * t)) −‘ {x. poly qq x=0} ∩ A)
     proof (rule finite-vimage-IntI)
       have x = y when tan (pi * x) = tan (pi * y) x∈A y∈A for x y
       proof −
         define x' where x'=(if x<1/2 then x else x-1)
         define y' where y'=(if y<1/2 then y else y-1)
         have x'*pi = y'*pi
         proof (rule tan-inj-pi)
           have *:- 1 / 2 < x' x' < 1 / 2 - 1 / 2 < y' y' < 1 / 2
             using that(2,3) unfolding x'-def y'-def A-def by simp-all
           show - (pi / 2) < x' * pi x' * pi < pi / 2 - (pi / 2) < y' * pi
                 y'*pi < pi / 2
             using mult-strict-right-mono[OF *(1),of pi]
                   mult-strict-right-mono[OF *(2),of pi]
                   mult-strict-right-mono[OF *(3),of pi]
                   mult-strict-right-mono[OF *(4),of pi]
             by auto
         next
```

```
          have tan (x' * pi) = tan (x * pi)
            unfolding x'-def using tan-periodic-int[of - - 1,simplified]
            by (auto simp add:algebra-simps)
          also have ... = tan (y * pi)
            using ‹tan (pi * x) = tan (pi * y)› by (auto simp:algebra-simps)
          also have ... = tan (y' * pi)
            unfolding y'-def using tan-periodic-int[of - - 1,simplified]
            by (auto simp add:algebra-simps)
          finally show tan (x' * pi) = tan (y' * pi) .
        qed
        then have x'=y' by auto
        then show ?thesis
          using that(2,3) unfolding x'-def y'-def A-def by (auto split:if-splits)
      qed
      then show inj-on (λt. tan (pi * t)) A
        unfolding inj-on-def by blast
    next
      have qq≠0
      proof (rule ccontr)
        assume ¬ qq ≠ 0
       then have Re ((poly p ∘ circlepath z0 r) t) = 0 when t∈{0..1} - {1/2}
for t
          apply (subst poly-eq)
          using that by auto
        then show False using False by blast
      qed
      then show finite {x. poly qq x = 0} by (simp add: poly-roots-finite)
    qed
    then show ?thesis by (elim rev-finite-subset) (auto simp:A-def)
  qed
  moreover have {t. poly qq (tan (pi * t)) = 0 ∧ 0 ≤ t ∧ t < 1 ∧ t≠1/2} =
S
    unfolding S-def P-def A-def using poly-eq by force
  ultimately have finite S by blast
  then have finite (S ∪ (if P 1 then {1} else {}) ∪ (if P (1/2) then {1/2} else
{}))
    by auto
  moreover have (S ∪ (if P 1 then {1} else {}) ∪ (if P (1/2) then {1/2} else
{}))
                  = {t. P t ∧ 0 ≤ t ∧ t ≤ 1}
  proof −
    have 1∈A 1/2 ∈A unfolding A-def by auto
   then have (S ∪ (if P 1 then {1} else {}) ∪ (if P (1/2) then {1/2} else {}))
                  = {t∈A. P t}
      unfolding S-def
      apply auto
      by (metis eq-divide-eq-numeral1(1) zero-neq-numeral)+
    also have ... = {t. P t ∧ 0 ≤ t ∧ t ≤ 1}
      unfolding A-def by auto
```

finally show *?thesis* **.**
qed
ultimately have *finite* $\{t.\ P\ t \wedge 0 \leq t \wedge t \leq 1\}$ **by** *auto*
then show *?thesis* **unfolding** *P-def* **by** *simp*
qed
then show *?thesis*
apply (*rule-tac finite-imp-finite-ReZ-segments*)
by *auto*
qed

3.2 Some useful conformal/*bij-betw* properties

lemma *bij-betw-plane-ball*:*bij-betw* $(\lambda x.\ (\mathrm{i}-x)/(\mathrm{i}+x))\ \{x.\ Im\ x>0\}\ (ball\ 0\ 1)$
proof (*rule bij-betw-imageI*)
have *neq*:$\mathrm{i} + x \neq 0$ **when** $Im\ x>0$ **for** x
using *that*
by (*metis add-less-same-cancel2 add-uminus-conv-diff diff-0 diff-add-cancel imaginary-unit.simps*(2) *not-one-less-zero uminus-complex.sel*(2))
then show *inj-on* $(\lambda x.\ (\mathrm{i} - x)\ /\ (\mathrm{i} + x))\ \{x.\ 0 < Im\ x\}$
unfolding *inj-on-def* **by** (*auto simp add*:*divide-simps algebra-simps*)
have *cmod* $((\mathrm{i} - x)\ /\ (\mathrm{i} + x)) < 1$ **when** $0 < Im\ x$ **for** x
proof −
have *cmod* $(\mathrm{i} - x) <$ *cmod* $(\mathrm{i} + x)$
unfolding *norm-lt inner-complex-def* **using** *that*
by (*auto simp add*:*algebra-simps*)
then show *?thesis*
unfolding *norm-divide* **using** *neq*[*OF that*] **by** *auto*
qed
moreover have $x \in (\lambda x.\ (\mathrm{i} - x)\ /\ (\mathrm{i} + x))\ `\ \{x.\ 0 < Im\ x\}$ **when** *cmod* $x <$ *1* **for** x
proof (*rule rev-image-eqI*[*of* $\mathrm{i}*(1-x)/(1+x)$])
have $1 + x \neq 0\ \mathrm{i} * 2 + \mathrm{i} * (x * 2) \neq 0$
subgoal using *that* **by** (*metis complex-mod-triangle-sub norm-one norm-zero not-le pth-7*(1))
subgoal using *that* **by** (*metis* ‹$1 + x \neq 0$› *complex-i-not-zero div-mult-self4 mult-2*
mult-zero-right nonzero-mult-div-cancel-left nonzero-mult-div-cancel-right one-add-one zero-neq-numeral)
done
then show $x = (\mathrm{i} - \mathrm{i} * (1 - x)\ /\ (1 + x))\ /\ (\mathrm{i} + \mathrm{i} * (1 - x)\ /\ (1 + x))$
by (*auto simp add*:*field-simps*)
show $\mathrm{i} * (1 - x)\ /\ (1 + x) \in \{x.\ 0 < Im\ x\}$
apply (*auto simp*:*Im-complex-div-gt-0 algebra-simps*)
using *that* **unfolding** *cmod-def* **by** (*auto simp*:*power2-eq-square*)
qed
ultimately show $(\lambda x.\ (\mathrm{i} - x)\ /\ (\mathrm{i} + x))\ `\ \{x.\ 0 < Im\ x\} = ball\ 0\ 1$
by *auto*
qed

lemma *bij-betw-axis-sphere*:*bij-betw* ($\lambda x.$ (i−x)/(i+x)) {$x.$ *Im* x=*0*} (*sphere 0 1* − {−*1*})
proof (*rule bij-betw-imageI*)
have *neq*:i + $x \neq 0$ **when** *Im* x=*0* **for** x
using *that*
by (*metis add-diff-cancel-left' imaginary-unit.simps*(*2*) *minus-complex.simps*(*2*)
right-minus-eq zero-complex.simps(*2*) *zero-neq-one*)
then show *inj-on* ($\lambda x.$ (i − x) / (i + x)) {$x.$ *Im* x = *0*}
unfolding *inj-on-def* **by** (*auto simp add*:*divide-simps algebra-simps*)
have *cmod* ((i − x) / (i + x)) = *1* (i − x) / (i + x) ≠ − *1* **when** *Im* x = *0* **for** x
proof −
have *cmod* (i + x) = *cmod* (i − x)
using *that* **unfolding** *cmod-def* **by** *auto*
then show *cmod* ((i − x) / (i + x)) = *1*
unfolding *norm-divide* **using** *neq*[*OF that*] **by** *auto*
show (i − x) / (i + x) ≠ − *1* **using** *neq*[*OF that*] **by** (*auto simp add*:*divide-simps*)
qed
moreover have $x \in$ ($\lambda x.$ (i − x) / (i + x)) ‘ {$x.$ *Im* x = *0*}
when *cmod* x = *1* $x \neq -1$ **for** x
proof (*rule rev-image-eqI*[*of* i∗(*1*−x)/(*1*+x)])
have *1* + $x \neq 0$ i ∗ *2* + i ∗ (x ∗ *2*) ≠*0*
subgoal using *that*(*2*) **by** *algebra*
subgoal using *that* **by** (*metis* ‹*1* + x ≠ *0*› *complex-i-not-zero div-mult-self4 mult-2*
mult-zero-right nonzero-mult-div-cancel-left nonzero-mult-div-cancel-right one-add-one zero-neq-numeral)
done
then show x = (i − i ∗ (*1* − x) / (*1* + x)) / (i + i ∗ (*1* − x) / (*1* + x))
by (*auto simp add*:*field-simps*)
show i ∗ (*1* − x) / (*1* + x) ∈ {$x.$ *Im* x = *0*}
apply (*auto simp*:*algebra-simps Im-complex-div-eq-0*)
using *that*(*1*) **unfolding** *cmod-def* **by** (*auto simp*:*power2-eq-square*)
qed
ultimately show ($\lambda x.$ (i − x) / (i + x)) ‘ {$x.$ *Im* x = *0*} = *sphere 0 1* − {−*1*}
by *force*
qed

lemma *bij-betw-ball-uball*:
assumes r>*0*
shows *bij-betw* ($\lambda x.$ *complex-of-real* r∗x + *z0*) (*ball 0 1*) (*ball z0 r*)
proof (*rule bij-betw-imageI*)
show *inj-on* ($\lambda x.$ *complex-of-real* r ∗ x + *z0*) (*ball 0 1*)
unfolding *inj-on-def* **using** *assms* **by** *simp*
have *dist z0* (*complex-of-real* r ∗ x + *z0*) < r **when** *cmod* x<*1* **for** x
using *that assms* **by** (*auto simp*:*dist-norm norm-mult abs-of-pos*)
moreover have $x \in$ ($\lambda x.$ *complex-of-real* r ∗ x + *z0*) ‘ *ball 0 1* **when** *dist z0* x

$< r$ **for** x
  apply (*rule rev-image-eqI*[*of* $(x-z0)/r$])
 using *that assms* **by** (*auto simp add*: *dist-norm norm-divide norm-minus-commute*)
 ultimately show (λx. *complex-of-real r* $*$ x + *z0*) ‘ *ball 0 1* = *ball z0 r*
  by *auto*
qed

lemma *bij-betw-sphere-usphere*:
 assumes *r*>*0*
 shows *bij-betw* (λx. *complex-of-real r*$*x$ + *z0*) (*sphere 0 1*) (*sphere z0 r*)
proof (*rule bij-betw-imageI*)
 show *inj-on* (λx. *complex-of-real r* $*$ x + *z0*) (*sphere 0 1*)
  unfolding *inj-on-def* **using** *assms* **by** *simp*
 have *dist z0* (*complex-of-real r* $*$ x + *z0*) = r **when** *cmod x*=*1* **for** x
  using *that assms* **by** (*auto simp*:*dist-norm norm-mult abs-of-pos*)
 moreover have $x \in$ (λx. *complex-of-real r* $*$ x + *z0*) ‘ *sphere 0 1* **when** *dist z0* $x = r$ **for** x
  apply (*rule rev-image-eqI*[*of* $(x-z0)/r$])
 using *that assms* **by** (*auto simp add*: *dist-norm norm-divide norm-minus-commute*)
 ultimately show (λx. *complex-of-real r* $*$ x + *z0*) ‘ *sphere 0 1* = *sphere z0 r*
  by *auto*
qed

lemma *proots-ball-plane-eq*:
 defines *q1*≡[:i,−*1*:] **and** *q2*≡[:i,*1*:]
 assumes *p*≠*0*
 shows *proots-count p* (*ball 0 1*) = *proots-count* (*fcompose p q1 q2*) {x. *0* < *Im* x}
 unfolding *q1-def q2-def*
proof (*rule proots-fcompose-bij-eq*[*OF* - ‹*p*≠*0*›])
 show $\forall x \in \{x.\ 0 < Im\ x\}$. *poly* [:i, *1*:] $x \neq 0$
  apply *simp*
  by (*metis add-less-same-cancel2 imaginary-unit.simps*(*2*) *not-one-less-zero*
      plus-complex.simps(*2*) *zero-complex.simps*(*2*))
 show *infinite* (*UNIV*::*complex set*) **by** (*simp add*: *infinite-UNIV-char-0*)
qed (*use bij-betw-plane-ball* **in** *auto*)

lemma *proots-sphere-axis-eq*:
 defines *q1*≡[:i,−*1*:] **and** *q2*≡[:i,*1*:]
 assumes *p*≠*0*
 shows *proots-count p* (*sphere 0 1* − {− *1*}) = *proots-count* (*fcompose p q1 q2*) {x. *0* = *Im* x}
unfolding *q1-def q2-def*
proof (*rule proots-fcompose-bij-eq*[*OF* - ‹*p*≠*0*›])
  show $\forall x \in \{x.\ 0 = Im\ x\}$. *poly* [:i, *1*:] $x \neq 0$ **by** (*simp add*: *Complex-eq-0 plus-complex.code*)
 show *infinite* (*UNIV*::*complex set*) **by** (*simp add*: *infinite-UNIV-char-0*)
qed (*use bij-betw-axis-sphere* **in** *auto*)

lemma *proots-card-ball-plane-eq*:
defines *q1*≡[:i,−*1*:] **and** *q2*≡[:i,*1*:]
assumes *p*≠*0*
shows *card* (*proots-within p* (*ball 0 1*)) = *card* (*proots-within* (*fcompose p q1 q2*) {*x*. *0* < *Im x*})
unfolding *q1-def q2-def*
proof (*rule proots-card-fcompose-bij-eq*[*OF* - ‹*p*≠*0*›])
show ∀ *x*∈{*x*. *0* < *Im x*}. *poly* [:i, *1*:] *x* ≠ *0*
apply *simp*
by (*metis add-less-same-cancel2 imaginary-unit.simps*(*2*) *not-one-less-zero plus-complex.simps*(*2*) *zero-complex.simps*(*2*))
qed (*use bij-betw-plane-ball infinite-UNIV-char-0* **in** *auto*)

lemma *proots-card-sphere-axis-eq*:
defines *q1*≡[:i,−*1*:] **and** *q2*≡[:i,*1*:]
assumes *p*≠*0*
shows *card* (*proots-within p* (*sphere 0 1* − {− *1*}))
= *card* (*proots-within* (*fcompose p q1 q2*) {*x*. *0* = *Im x*})
unfolding *q1-def q2-def*
proof (*rule proots-card-fcompose-bij-eq*[*OF* - ‹*p*≠*0*›])
show ∀ *x*∈{*x*. *0* = *Im x*}. *poly* [:i, *1*:] *x* ≠ *0* **by** (*simp add*: *Complex-eq-0 plus-complex.code*)
qed (*use bij-betw-axis-sphere infinite-UNIV-char-0* **in** *auto*)

lemma *proots-uball-eq*:
fixes *z0*::*complex* **and** *r*::*real*
defines *q*≡[:*z0*, *of-real r*:]
assumes *p*≠*0* **and** *r*>*0*
shows *proots-count p* (*ball z0 r*) = *proots-count* (*p* ∘$_p$ *q*) (*ball 0 1*)
proof −
show *?thesis*
apply (*rule proots-pcompose-bij-eq*[*OF* - ‹*p*≠*0*›])
subgoal unfolding *q-def* **using** *bij-betw-ball-uball*[*OF* ‹*r*>*0*›,*of z0*] **by** (*auto simp*:*algebra-simps*)
subgoal unfolding *q-def* **using** ‹*r*>*0*› **by** *auto*
done
qed

lemma *proots-card-uball-eq*:
fixes *z0*::*complex* **and** *r*::*real*
defines *q*≡[:*z0*, *of-real r*:]
assumes *r*>*0*
shows *card* (*proots-within p* (*ball z0 r*)) = *card* (*proots-within* (*p* ∘$_p$ *q*) (*ball 0 1*))
proof −
have *?thesis*
when *p*=*0*
proof −
have *card* (*ball z0 r*) = *0 card* (*ball* (*0*::*complex*) *1*) = *0*

```
    using infinite-ball[OF ‹r>0›,of z0] infinite-ball[of 1 0::complex] by auto
   then show ?thesis using that by auto
 qed
 moreover have ?thesis
   when p≠0
   apply (rule proots-card-pcompose-bij-eq[OF - ‹p≠0›])
   subgoal unfolding q-def using bij-betw-ball-uball[OF ‹r>0›,of z0] by (auto
simp:algebra-simps)
   subgoal unfolding q-def using ‹r>0› by auto
   done
 ultimately show ?thesis
   by blast
qed

lemma proots-card-usphere-eq:
 fixes z0::complex and r::real
 defines q≡[:z0, of-real r:]
 assumes r>0
 shows card (proots-within p (sphere z0 r)) = card (proots-within (p ∘p q) (sphere
0 1))
proof −
 have ?thesis
   when p=0
 proof −
   have card (sphere z0 r) = 0 card (sphere (0::complex) 1) = 0
    using infinite-sphere[OF ‹r>0›,of z0] infinite-sphere[of 1 0::complex] by auto

   then show ?thesis using that by auto
 qed
 moreover have ?thesis
   when p≠0
   apply (rule proots-card-pcompose-bij-eq[OF - ‹p≠0›])
   subgoal unfolding q-def using bij-betw-sphere-usphere[OF ‹r>0›,of z0]
     by (auto simp:algebra-simps)
   subgoal unfolding q-def using ‹r>0› by auto
   done
 ultimately show card (proots-within p (sphere z0 r)) = card (proots-within (p
∘p q) (sphere 0 1))
   by blast
qed
```

3.3 Combining two real polynomials into a complex one

```
definition cpoly-of:: real poly ⇒ real poly ⇒ complex poly where
 cpoly-of pR pI = map-poly of-real pR + smult i (map-poly of-real pI)

lemma cpoly-of-eq-0-iff[iff]:
 cpoly-of pR pI = 0 ⟷ pR = 0 ∧ pI = 0
proof −
```

```
  have pR = 0 ∧ pI = 0 when cpoly-of pR pI = 0
  proof −
    have complex-of-real (coeff pR n) + i ∗ complex-of-real (coeff pI n) = 0 for n
      using that unfolding poly-eq-iff cpoly-of-def by (auto simp:coeff-map-poly)
    then have coeff pR n = 0 ∧ coeff pI n = 0 for n
      by (metis Complex-eq Im-complex-of-real Re-complex-of-real complex.sel(1)
complex.sel(2)
          of-real-0)
    then show ?thesis unfolding poly-eq-iff by auto
  qed
  then show ?thesis by (auto simp:cpoly-of-def)
qed

lemma cpoly-of-decompose:
    p = cpoly-of (map-poly Re p) (map-poly Im p)
  unfolding cpoly-of-def
  apply (induct p)
  by (auto simp add:map-poly-pCons map-poly-map-poly complex-eq)

lemma cpoly-of-dist-right:
    cpoly-of (pR∗g) (pI∗g) = cpoly-of pR pI ∗ (map-poly of-real g)
  unfolding cpoly-of-def by (simp add: distrib-right)

lemma poly-cpoly-of-real:
    poly (cpoly-of pR pI) (of-real x) = Complex (poly pR x) (poly pI x)
  unfolding cpoly-of-def by (simp add: Complex-eq of-real-poly-map-poly)

lemma poly-cpoly-of-real-iff:
  shows poly (cpoly-of pR pI) (of-real t) =0 ⟷ poly pR t = 0 ∧ poly pI t=0
  unfolding  poly-cpoly-of-real using Complex-eq-0 by blast

lemma order-cpoly-gcd-eq:
  assumes pR≠0 ∨ pI≠0
  shows order t (cpoly-of pR pI) = order t (gcd pR pI)
proof −
  define g where g = gcd pR pI
  have [simp]:g≠0 unfolding g-def using assms by auto
  obtain pr pi where pri: pR = pr ∗ g pI = pi ∗ g coprime pr pi
    unfolding g-def using assms(1) gcd-coprime-exists ‹g ≠ 0› g-def by blast
  then have pr ≠0 ∨ pi ≠0 using assms mult-zero-left by blast

  have order t (cpoly-of pR pI) = order t (cpoly-of pr pi ∗ (map-poly of-real g))
    unfolding pri cpoly-of-dist-right by simp
  also have ... = order t (cpoly-of pr pi) + order t g
    apply (subst order-mult)
    using ‹pr ≠0 ∨ pi ≠0› by (auto simp:map-poly-order-of-real)
  also have ... = order t g
  proof −
    have poly (cpoly-of pr pi) t ≠0 unfolding poly-cpoly-of-real-iff
```

using ‹*coprime pr pi*› *coprime-poly-0* **by** *blast*
then have *order t* (*cpoly-of pr pi*) = *0* **by** (*simp add*: *order-0I*)
then show *?thesis* **by** *auto*
qed
finally show *?thesis* **unfolding** *g-def* **.**
qed

3.4 Number of roots on a (bounded or unbounded) segment

— 1 dimensional hyperplane
definition *unbounded-line*::*'a*::*real-vector* ⇒ *'a* ⇒ *'a set* **where**
unbounded-line a b = ({*x*. ∃ *u*::*real*. *x*= (*1* − *u*) ∗$_R$ *a* + *u* ∗$_R$ *b*})

definition *proots-line-card*:: *complex poly* ⇒ *complex* ⇒ *complex* ⇒ *nat* **where**
proots-line-card p st tt = *card* (*proots-within p* (*open-segment st tt*))

definition *proots-unbounded-line-card*:: *complex poly* ⇒ *complex* ⇒ *complex* ⇒ *nat* **where**
proots-unbounded-line-card p st tt = *card* (*proots-within p* (*unbounded-line st tt*))

definition *proots-unbounded-line* :: *complex poly* ⇒ *complex* ⇒ *complex* ⇒ *nat* **where**
proots-unbounded-line p st tt = *proots-count p* (*unbounded-line st tt*)

lemma *card-proots-open-segments*:
assumes *poly p st* ≠*0 poly p tt* ≠ *0*
shows *card* (*proots-within p* (*open-segment st tt*)) =
(*let pc* = *pcompose p* [:*st*, *tt* − *st*:];
pR = *map-poly Re pc*;
pI = *map-poly Im pc*;
g = *gcd pR pI*
in changes-itv-smods 0 1 g (*pderiv g*)) (**is** *?L* = *?R*)
proof −
define *pc pR pI g* **where**
pc = *pcompose p* [:*st*, *tt*−*st*:] **and**
pR = *map-poly Re pc* **and**
pI = *map-poly Im pc* **and**
g = *gcd pR pI*
have *poly-iff*:*poly g t*=*0* ⟷ *poly pc t* =*0* **for** *t*
proof −
have *poly g t* = *0* ⟷ *poly pR t* =*0* ∧ *poly pI t* =*0*
unfolding *g-def* **using** *poly-gcd-iff* **by** *auto*
also have ... ⟷ *poly pc t* =*0*
proof −
have *cpoly-of pR pI* = *pc*
unfolding *pc-def pR-def pI-def* **using** *cpoly-of-decompose* **by** *auto*
then show *?thesis* **using** *poly-cpoly-of-real-iff* **by** *blast*
qed
finally show *?thesis* **by** *auto*

```
  qed

  have ?R = changes-itv-smods 0 1 g (pderiv g)
    unfolding pc-def g-def pI-def pR-def by (auto simp add:Let-def)
  also have ... = card {t. poly g t = 0 ∧ 0 < t ∧ t < 1}
  proof −
    have poly g 0 ≠ 0
     using poly-iff[of 0] assms unfolding pc-def by (auto simp add:poly-pcompose)
    moreover have poly g 1 ≠ 0
     using poly-iff[of 1] assms unfolding pc-def by (auto simp add:poly-pcompose)
    ultimately show ?thesis using sturm-interval[of 0 1 g] by auto
  qed
  also have ... = card {t::real. poly pc t = 0 ∧ 0 < t ∧ t < 1}
    unfolding poly-iff by simp
  also have ... = ?L
  proof (cases st=tt)
    case True
    then show ?thesis unfolding pc-def poly-pcompose using ‹poly p tt ≠ 0›
      by auto
  next
    case False
    define ff where ff = (λt::real. st + t∗(tt−st))
    define ll where ll = {t. poly pc (complex-of-real t) = 0 ∧ 0 < t ∧ t < 1}
    have ff ‘ ll = proots-within p (open-segment st tt)
    proof (rule equalityI)
      show ff ‘ ll ⊆ proots-within p (open-segment st tt)
        unfolding ll-def ff-def pc-def poly-pcompose
        by (auto simp add:in-segment False scaleR-conv-of-real algebra-simps)
    next
      show proots-within p (open-segment st tt) ⊆ ff ‘ ll
      proof clarify
        fix x assume asm:x ∈ proots-within p (open-segment st tt)
        then obtain u where 0<u and u < 1 and u:x = (1 − u) ∗R st + u ∗R
tt
          by (auto simp add:in-segment)
        then have poly p ((1 − u) ∗R st + u ∗R tt) = 0 using asm by simp
        then have u ∈ ll
          unfolding ll-def pc-def poly-pcompose
          by (simp add:scaleR-conv-of-real algebra-simps ‹0<u› ‹u<1›)
        moreover have x = ff u
       unfolding ff-def using u by (auto simp add:algebra-simps scaleR-conv-of-real)
        ultimately show x ∈ ff ‘ ll by (rule rev-image-eqI[of u])
      qed
    qed
    moreover have inj-on ff ll
      unfolding ff-def using False inj-on-def by fastforce
    ultimately show ?thesis unfolding ll-def
      using card-image[of ff] by fastforce
  qed
```

```
  finally show ?thesis by simp
qed

lemma unbounded-line-closed-segment: closed-segment a b ⊆ unbounded-line a b
  unfolding unbounded-line-def closed-segment-def by auto

lemma card-proots-unbounded-line:
  assumes st≠tt
  shows card (proots-within p (unbounded-line st tt)) =
              (let pc = pcompose p [:st, tt − st:];
                   pR = map-poly Re pc;
                   pI = map-poly Im pc;
                   g  = gcd pR pI
               in nat (changes-R-smods g (pderiv g))) (is ?L = ?R)
proof −
  define pc pR pI g where
     pc = pcompose p [:st, tt−st:] and
     pR = map-poly Re pc and
     pI = map-poly Im pc and
     g  = gcd pR pI
  have poly-iff:poly g t=0 ⟷ poly pc t =0 for t
  proof −
    have poly g t = 0 ⟷ poly pR t =0 ∧ poly pI t =0
      unfolding g-def using poly-gcd-iff by auto
    also have ... ⟷ poly pc t =0
    proof −
      have cpoly-of pR pI = pc
        unfolding pc-def pR-def pI-def using cpoly-of-decompose by auto
      then show ?thesis using poly-cpoly-of-real-iff by blast
    qed
    finally show ?thesis by auto
  qed

  have ?R = nat (changes-R-smods g (pderiv g))
    unfolding pc-def g-def pI-def pR-def by (auto simp add:Let-def)
  also have ... = card {t. poly g t = 0}
    using sturm-R[of g] by simp
  also have ... = card {t::real. poly pc t = 0}
    unfolding poly-iff by simp
  also have ... = ?L
  proof (cases st=tt)
    case True
    then show ?thesis unfolding pc-def poly-pcompose unbounded-line-def using
assms
      by (auto simp add:proots-within-def)
  next
    case False
    define ff where ff = (λt::real. st + t∗(tt−st))
    define ll where ll = {t. poly pc (complex-of-real t) = 0}
```

```
    have ff ‘ ll = proots-within p (unbounded-line st tt)
    proof (rule equalityI)
      show ff ‘ ll ⊆ proots-within p (unbounded-line st tt)
        unfolding ll-def ff-def pc-def poly-pcompose
      by (auto simp add:unbounded-line-def False scaleR-conv-of-real algebra-simps)
    next
      show proots-within p (unbounded-line st tt) ⊆ ff ‘ ll
      proof clarify
        fix x assume asm:x ∈ proots-within p (unbounded-line st tt)
        then obtain u where u:x = (1 − u) *R st + u *R tt
          by (auto simp add:unbounded-line-def)
        then have poly p ((1 − u) *R st + u *R tt) = 0 using asm by simp
        then have u ∈ ll
          unfolding ll-def pc-def poly-pcompose
          by (simp add:scaleR-conv-of-real algebra-simps unbounded-line-def)
        moreover have x = ff u
       unfolding ff-def using u by (auto simp add:algebra-simps scaleR-conv-of-real)
        ultimately show x ∈ ff ‘ ll by (rule rev-image-eqI[of u])
      qed
    qed
    moreover have inj-on ff ll
      unfolding ff-def using False inj-on-def by fastforce
    ultimately show ?thesis unfolding ll-def
      using card-image[of ff] by metis
  qed
  finally show ?thesis by simp
qed

lemma proots-unbounded-line:
  assumes st≠tt p≠0
  shows (proots-count p (unbounded-line st tt)) =
              (let pc = pcompose p [:st, tt − st:];
                   pR = map-poly Re pc;
                   pI = map-poly Im pc;
                   g  = gcd pR pI
               in nat (changes-R-smods-ext g (pderiv g))) (is ?L = ?R)
proof −
  define pc pR pI g where
     pc = pcompose p [:st, tt−st:] and
     pR = map-poly Re pc and
     pI = map-poly Im pc and
     g  = gcd pR pI
  have [simp]: g≠0 pc≠0
  proof −
    show pc≠0 using assms(1) assms(2) pc-def pcompose-eq-0 by fastforce
   then have pR≠0 ∨ pI≠0 unfolding pR-def pI-def by (metis cpoly-of-decompose
map-poly-0)
    then show g≠0 unfolding g-def by simp
  qed
```

```
  have order-eq:order t g = order t pc for t
    apply (subst order-cpoly-gcd-eq[of pR pI,folded g-def,symmetric])
    subgoal using ‹g≠0› unfolding g-def by simp
   subgoal unfolding pR-def pI-def by (simp add:cpoly-of-decompose[symmetric])
    done

  have ?R = nat (changes-R-smods-ext g (pderiv g))
    unfolding pc-def g-def pI-def pR-def by (auto simp add:Let-def)
  also have ... = proots-count g UNIV
    using sturm-ext-R[OF ‹g≠0›] by auto
  also have ... = proots-count (map-poly complex-of-real g) (of-real ‘ UNIV)
    apply (subst proots-count-of-real)
    by auto
  also have ... = proots-count (map-poly complex-of-real g) {x. Im x = 0}
    apply (rule arg-cong2[where f=proots-count])
    using Reals-def complex-is-Real-iff by auto
  also have ... = proots-count pc {x. Im x = 0}
    apply (rule proots-count-cong)
   apply (metis (mono-tags) Im-complex-of-real Re-complex-of-real ‹g ≠ 0› complex-surj

          map-poly-order-of-real mem-Collect-eq order-eq)
    by auto
  also have ... = proots-count p (unbounded-line st tt)
  proof −
    have poly [:st, tt − st:] ‘ {x. Im x = 0} = unbounded-line st tt
      unfolding unbounded-line-def
      apply safe
      subgoal for - x
        apply (rule-tac x=Re x in exI)
        apply (simp add:algebra-simps)
        by (simp add: mult.commute scaleR-complex.code times-complex.code)
      subgoal for - u
        apply (rule rev-image-eqI[of of-real u])
        by (auto simp:scaleR-conv-of-real algebra-simps)
      done
    then show ?thesis
      unfolding pc-def
      apply (subst proots-pcompose)
      using ‹p≠0› ‹st≠tt› by auto
  qed
  finally show ?thesis by simp
qed

lemma proots-unbounded-line-card-code[code]:
  proots-unbounded-line-card p st tt =
            (if st≠tt then
              (let pc = pcompose p [:st, tt − st:];
                   pR = map-poly Re pc;
                   pI = map-poly Im pc;
```

```
            g  = gcd pR pI
          in nat (changes-R-smods g (pderiv g)))
        else
            Code.abort (STR ''proots-unbounded-line-card fails due to invalid
hyperplanes.'')
              (λ-. proots-unbounded-line-card p st tt))
 unfolding proots-unbounded-line-card-def using card-proots-unbounded-line[of st
tt p] by auto

lemma proots-unbounded-line-code[code]:
 proots-unbounded-line p st tt =
        ( if st≠tt then
          if p≠0 then
           (let pc = pcompose p [:st, tt − st:];
            pR = map-poly Re pc;
            pI = map-poly Im pc;
            g  = gcd pR pI
           in nat (changes-R-smods-ext g (pderiv g)))
          else
           Code.abort (STR ''proots-unbounded-line fails due to p=0'')
            (λ-. proots-unbounded-line p st tt)
          else
              Code.abort (STR ''proots-unbounded-line fails due to invalid
hyperplanes.'')
              (λ-. proots-unbounded-line p st tt) )
 unfolding proots-unbounded-line-def using proots-unbounded-line by auto
```

3.5 Checking if there a polynomial root on a closed segment

```
definition no-proots-line::complex poly ⇒ complex ⇒ complex ⇒ bool where
 no-proots-line p st tt = (proots-within p (closed-segment st tt) = {})


lemma no-proots-line-code[code]: no-proots-line p st tt = (if poly p st ≠0 ∧ poly
p tt ≠ 0 then
          (let pc = pcompose p [:st, tt − st:];
            pR = map-poly Re pc;
            pI = map-poly Im pc;
            g  = gcd pR pI
           in if changes-itv-smods 0 1 g (pderiv g) = 0 then True else False)
else False)
       (is ?L = ?R)
proof (cases poly p st ≠0 ∧ poly p tt ≠ 0)
 case False
 thus ?thesis unfolding no-proots-line-def by auto
next
 case True
 then have poly p st ≠ 0 poly p tt ≠ 0 by auto
 define pc pR pI g where
```

```
    pc = pcompose p [:st, tt−st:] and
    pR = map-poly Re pc and
    pI = map-poly Im pc and
    g  = gcd pR pI
  have poly-iff:poly g t=0 ⟷ poly pc t =0 for t
  proof −
   have poly g t = 0 ⟷ poly pR t =0 ∧ poly pI t =0
    unfolding g-def using poly-gcd-iff by auto
   also have ... ⟷ poly pc t =0
   proof −
    have cpoly-of pR pI = pc
     unfolding pc-def pR-def pI-def using cpoly-of-decompose by auto
    then show ?thesis using poly-cpoly-of-real-iff by blast
   qed
   finally show ?thesis by auto
  qed
  have ?R = (changes-itv-smods 0 1 g (pderiv g) = 0)
   using True unfolding pc-def g-def pI-def pR-def
   by (auto simp add:Let-def)
  also have ... = (card {x. poly g x = 0 ∧ 0 < x ∧ x < 1} = 0)
  proof −
   have poly g 0 ≠ 0
    using poly-iff[of 0] True unfolding pc-def by (auto simp add:poly-pcompose)
   moreover have poly g 1 ≠ 0
    using poly-iff[of 1] True unfolding pc-def by (auto simp add:poly-pcompose)
   ultimately show ?thesis using sturm-interval[of 0 1 g] by auto
  qed
  also have ... = ({x. poly g x = 0 ∧ 0 < x ∧ x < 1} = {})
  proof −
   have g≠0
   proof (rule ccontr)
    assume ¬ g ≠ 0
    then have poly pc 0 =0
     using poly-iff[of 0] by auto
   then show False using True unfolding pc-def by (auto simp add:poly-pcompose)
   qed
   from poly-roots-finite[OF this] have finite {x. poly g x = 0 ∧ 0 < x ∧ x < 1}
    by auto
   then show ?thesis using card-eq-0-iff by auto
  qed
  also have ... = ?L
  proof −
   have (∃ t. poly g t = 0 ∧ 0 < t ∧ t < 1) ⟷ (∃ t::real. poly pc t = 0 ∧ 0 <
t ∧ t < 1)
    using poly-iff by auto
   also have ... ⟷ (∃ x. x ∈ closed-segment st tt ∧ poly p x = 0)
   proof
    assume ∃ t. poly pc (complex-of-real t) = 0 ∧ 0 < t ∧ t < 1
    then obtain t where ∗:poly pc (of-real t) = 0 and 0 < t t < 1 by auto
```

define x **where** x=*poly* [:*st*, *tt* − *st*:] *t*
have $x \in$ *closed-segment st tt* **using** ‹*0*<*t*› ‹*t*<*1*› **unfolding** *x-def in-segment*
by (*intro exI*[**where** x=*t*],*auto simp add*: *algebra-simps scaleR-conv-of-real*)
moreover have *poly p x=0* **using** ∗ **unfolding** *pc-def x-def*
by (*auto simp add*:*poly-pcompose*)
ultimately show $\exists x.\ x \in$ *closed-segment st tt* $\wedge$ *poly p x = 0* **by** *auto*
next
assume $\exists x.\ x \in$ *closed-segment st tt* $\wedge$ *poly p x = 0*
then obtain x **where** $x \in$ *closed-segment st tt poly p x = 0* **by** *auto*
then obtain *t*::*real* **where** ∗:$x = (1 - t) *_R$ *st* + *t* $*_R$ *tt* **and** *0*≤*t t*≤*1*
unfolding *in-segment* **by** *auto*
then have *x*=*poly* [:*st*, *tt* − *st*:] *t* **by** (*auto simp add*: *algebra-simps scaleR-conv-of-real*)
then have *poly pc* (*complex-of-real t*) = *0*
using ‹*poly p x=0*› **unfolding** *pc-def* **by** (*auto simp add*:*poly-pcompose*)
moreover have *t*≠*0 t*≠*1* **using** *True* ∗ ‹*poly p x=0*› **by** *auto*
then have *0*<*t t*<*1* **using** ‹*0*≤*t*› ‹*t*≤*1*› **by** *auto*
ultimately show $\exists t.$ *poly pc* (*complex-of-real t*) = *0* $\wedge$ *0* < *t* $\wedge$ *t* < *1* **by** *auto*
qed
finally show *?thesis*
unfolding *no-proots-line-def proots-within-def*
by *blast*
qed
finally show *?thesis* **by** *simp*
qed

3.6 Counting roots in a rectangle

definition *proots-rectangle* ::*complex poly* ⇒ *complex* ⇒ *complex* ⇒ *nat* **where**
proots-rectangle p lb ub = *proots-count p* (*box lb ub*)

lemma *closed-segment-imp-Re-Im*:
fixes *x*::*complex*
assumes *x*∈*closed-segment lb ub*
shows *Re lb* ≤ *Re ub* ⟹ *Re lb* ≤ *Re x* ∧ *Re x* ≤ *Re ub*
Im lb ≤ *Im ub* ⟹ *Im lb* ≤ *Im x* ∧ *Im x* ≤ *Im ub*
proof −
obtain *u* **where** *x-u*:x=$(1 - u) *_R$ *lb* + *u* $*_R$ *ub* **and** *0* ≤ *u u* ≤ *1*
using *assms* **unfolding** *closed-segment-def* **by** *auto*
have *Re lb* ≤ *Re x* **when** *Re lb* ≤ *Re ub*
proof −
have *Re x* = *Re* ((*1* − *u*) $*_R$ *lb* + *u* $*_R$ *ub*)
using *x-u* **by** *blast*
also have ... = *Re* (*lb* + *u* $*_R$ (*ub* − *lb*)) **by** (*auto simp add*:*algebra-simps*)
also have ... = *Re lb* + *u* ∗ (*Re ub* − *Re lb*) **by** *auto*
also have ... ≥ *Re lb* **using** ‹*u*≥*0*› ‹*Re lb* ≤ *Re ub*› **by** *auto*
finally show *?thesis* **.**
qed
moreover have *Im lb* ≤ *Im x* **when** *Im lb* ≤ *Im ub*

```
  proof −
    have Im x = Im ((1 − u) *R lb + u *R ub)
      using x-u by blast
    also have ... = Im (lb + u *R (ub − lb)) by (auto simp add:algebra-simps)
    also have ... = Im lb + u * (Im ub − Im lb) by auto
    also have ... ≥ Im lb using ‹u≥0› ‹Im lb ≤ Im ub› by auto
    finally show ?thesis .
  qed
  moreover have Re x ≤ Re ub when Re lb ≤ Re ub
  proof −
    have Re x = Re ((1 − u) *R lb + u *R ub)
      using x-u by blast
    also have ... = (1 − u) * Re lb + u * Re ub by auto
    also have ... ≤ (1 − u) * Re ub + u * Re ub
      using ‹u≤1› ‹Re lb ≤ Re ub› by (auto simp add: mult-left-mono)
    also have ... = Re ub by (auto simp add:algebra-simps)
    finally show ?thesis .
  qed
  moreover have Im x ≤ Im ub when Im lb ≤ Im ub
  proof −
    have Im x = Im ((1 − u) *R lb + u *R ub)
      using x-u by blast
    also have ... = (1 − u) * Im lb + u * Im ub by auto
    also have ... ≤ (1 − u) * Im ub + u * Im ub
      using ‹u≤1› ‹Im lb ≤ Im ub› by (auto simp add: mult-left-mono)
    also have ... = Im ub by (auto simp add:algebra-simps)
    finally show ?thesis .
  qed
  ultimately show
      Re lb ≤ Re ub ⟹ Re lb ≤ Re x ∧ Re x ≤ Re ub
      Im lb ≤ Im ub ⟹ Im lb ≤ Im x ∧ Im x ≤ Im ub
    by auto
qed

lemma closed-segment-degen-complex:
  ⟦Re lb = Re ub; Im lb ≤ Im ub ⟧
    ⟹ x ∈ closed-segment lb ub ⟷ Re x = Re lb ∧ Im lb ≤ Im x ∧ Im x ≤ Im
ub
  ⟦Im lb = Im ub; Re lb ≤ Re ub ⟧
    ⟹ x ∈ closed-segment lb ub ⟷ Im x = Im lb ∧ Re lb ≤ Re x ∧ Re x ≤ Re
ub
proof −
  show x ∈ closed-segment lb ub ⟷ Re x = Re lb ∧ Im lb ≤ Im x ∧ Im x ≤ Im
ub
    when Re lb = Re ub Im lb ≤ Im ub
  proof
    show Re x = Re lb ∧ Im lb ≤ Im x ∧ Im x ≤ Im ub when x ∈ closed-segment
lb ub
      using closed-segment-imp-Re-Im[OF that] ‹Re lb = Re ub› ‹Im lb ≤ Im ub›
```

by *fastforce*
next
assume *asm*:*Re x = Re lb ∧ Im lb ≤ Im x ∧ Im x ≤ Im ub*
define *u* **where** *u*=(*Im x − Im lb*)/ (*Im ub − Im lb*)
have *x = (1 − u) ∗R lb + u ∗R ub*
unfolding *u-def* **using** *asm* ‹*Re lb = Re ub*› ‹*Im lb ≤ Im ub*›
apply (*intro complex-eqI*)
apply (*auto simp add*:*field-simps*)
apply (*cases Im ub − Im lb =0*)
apply (*auto simp add*:*field-simps*)
done
moreover have *0≤u u≤1* **unfolding** *u-def*
using ‹*Im lb ≤ Im ub*› *asm*
by (*cases Im ub − Im lb =0*,*auto simp add*:*field-simps*)+
ultimately show *x ∈ closed-segment lb ub* **unfolding** *closed-segment-def* **by** *auto*
qed
show *x ∈ closed-segment lb ub ⟷ Im x = Im lb ∧ Re lb ≤ Re x ∧ Re x ≤ Re ub*
when *Im lb = Im ub Re lb ≤ Re ub*
proof
show *Im x = Im lb ∧ Re lb ≤ Re x ∧ Re x ≤ Re ub* **when** *x ∈ closed-segment lb ub*
using *closed-segment-imp-Re-Im*[*OF that*] ‹*Im lb = Im ub*› ‹*Re lb ≤ Re ub*›
by *fastforce*
next
assume *asm*:*Im x = Im lb ∧ Re lb ≤ Re x ∧ Re x ≤ Re ub*
define *u* **where** *u*=(*Re x − Re lb*)/ (*Re ub − Re lb*)
have *x = (1 − u) ∗R lb + u ∗R ub*
unfolding *u-def* **using** *asm* ‹*Im lb = Im ub*› ‹*Re lb ≤ Re ub*›
apply (*intro complex-eqI*)
apply (*auto simp add*:*field-simps*)
apply (*cases Re ub − Re lb =0*)
apply (*auto simp add*:*field-simps*)
done
moreover have *0≤u u≤1* **unfolding** *u-def*
using ‹*Re lb ≤ Re ub*› *asm*
by (*cases Re ub − Re lb =0*,*auto simp add*:*field-simps*)+
ultimately show *x ∈ closed-segment lb ub* **unfolding** *closed-segment-def* **by** *auto*
qed
qed

lemma *complex-box-ne-empty*:
fixes *a b*::*complex*
shows
cbox a b ≠ {} ⟷ (Re a ≤ Re b ∧ Im a ≤ Im b)
box a b ≠ {} ⟷ (Re a < Re b ∧ Im a < Im b)
by (*auto simp add*:*box-ne-empty Basis-complex-def*)

```
lemma proots-rectangle-code1:
  proots-rectangle p lb ub = (if Re lb < Re ub ∧ Im lb < Im ub then
          if p≠0 then
          if no-proots-line p lb (Complex (Re ub) (Im lb))
          ∧ no-proots-line p (Complex (Re ub) (Im lb)) ub
          ∧ no-proots-line p ub (Complex (Re lb) (Im ub))
          ∧ no-proots-line p (Complex (Re lb) (Im ub)) lb then
          (
          let p1 = pcompose p [:lb,  Complex (Re ub − Re lb) 0:];
              pR1 = map-poly Re p1; pI1 = map-poly Im p1; gc1 = gcd pR1 pI1;
             p2 = pcompose p [:Complex (Re ub) (Im lb), Complex 0 (Im ub − Im
lb):];
              pR2 = map-poly Re p2; pI2 = map-poly Im p2; gc2 = gcd pR2 pI2;
              p3 = pcompose p [:ub, Complex (Re lb − Re ub) 0:];
              pR3 = map-poly Re p3; pI3 = map-poly Im p3; gc3 = gcd pR3 pI3;
             p4 = pcompose p [:Complex (Re lb) (Im ub), Complex 0 (Im lb − Im
ub):];
              pR4 = map-poly Re p4; pI4 = map-poly Im p4; gc4 = gcd pR4 pI4
          in
            nat (− (changes-alt-itv-smods 0 1 (pR1 div gc1) (pI1 div gc1)
              + changes-alt-itv-smods 0 1 (pR2 div gc2) (pI2 div gc2)
              + changes-alt-itv-smods 0 1 (pR3 div gc3) (pI3 div gc3)
              + changes-alt-itv-smods 0 1 (pR4 div gc4) (pI4 div gc4))  div 4)
          )
          else Code.abort (STR ''proots-rectangle fails when there is a root on the
border.'')
          (λ-. proots-rectangle p lb ub)
          else Code.abort (STR ''proots-rectangle fails when p=0.'')
          (λ-. proots-rectangle p lb ub)
          else 0)
proof −
  have ?thesis when ¬ (Re lb < Re ub ∧ Im lb < Im ub)
  proof −
    have box lb ub = {} using complex-box-ne-empty[of lb ub] that by auto
   then have proots-rectangle p lb ub = 0 unfolding proots-rectangle-def by auto
    then show ?thesis by (simp add:that)
  qed
  moreover have ?thesis when Re lb < Re ub ∧ Im lb < Im ub p=0
    using that by simp
  moreover have ?thesis when
          Re lb < Re ub Im lb < Im ub p≠0
          and no-proots:
          no-proots-line p lb (Complex (Re ub) (Im lb))
          no-proots-line p (Complex (Re ub) (Im lb)) ub
          no-proots-line p ub (Complex (Re lb) (Im ub))
          no-proots-line p (Complex (Re lb) (Im ub)) lb
  proof −
    define l1 where l1 = linepath lb (Complex (Re ub) (Im lb))
```

define *l2* **where** *l2 = linepath (Complex (Re ub) (Im lb)) ub*
define *l3* **where** *l3 = linepath ub (Complex (Re lb) (Im ub))*
define *l4* **where** *l4 = linepath (Complex (Re lb) (Im ub)) lb*
define *rec* **where** *rec = l1 +++ l2 +++ l3 +++ l4*
have *valid*[*simp*]:*valid-path rec* **and** *loop*[*simp*]:*pathfinish rec = pathstart rec*
unfolding *rec-def l1-def l2-def l3-def l4-def* **by** *auto*
have *path-no-proots*:*path-image rec ∩ proots p = {}*
unfolding *rec-def l1-def l2-def l3-def l4-def*
apply (*subst path-image-join*,*simp-all del*:*Complex-eq*)+
using *no-proots*[*unfolded no-proots-line-def*] **by** (*auto simp del*:*Complex-eq*)

define *g1* **where** *g1 = poly p o l1*
define *g2* **where** *g2 = poly p o l2*
define *g3* **where** *g3 = poly p o l3*
define *g4* **where** *g4 = poly p o l4*
have [*simp*]: *path g1 path g2 path g3 path g4*
pathfinish g1 = pathstart g2 pathfinish g2 = pathstart g3 pathfinish g3 = pathstart g4
pathfinish g4 = pathstart g1
unfolding *g1-def g2-def g3-def g4-def l1-def l2-def l3-def l4-def*
by (*auto intro*!: *path-continuous-image continuous-intros*
simp add:*pathfinish-compose pathstart-compose*)
have [*simp*]: *finite-ReZ-segments g1 0 finite-ReZ-segments g2 0*
finite-ReZ-segments g3 0 finite-ReZ-segments g4 0
unfolding *g1-def l1-def g2-def l2-def g3-def l3-def g4-def l4-def poly-linepath-comp*

by (*rule finite-ReZ-segments-poly-of-real*)+
define *p1 pR1 pI1 gc1*
p2 pR2 pI2 gc2
p3 pR3 pI3 gc3
p4 pR4 pI4 gc4
where *p1 = pcompose p* [:*lb*, *Complex (Re ub − Re lb) 0*:]
and *pR1 = map-poly Re p1* **and** *pI1 = map-poly Im p1* **and** *gc1 = gcd pR1 pI1*
and *p2 = pcompose p* [:*Complex (Re ub) (Im lb)*, *Complex 0 (Im ub − Im lb)*:]
and *pR2 = map-poly Re p2* **and** *pI2 = map-poly Im p2* **and** *gc2 = gcd pR2 pI2*
and *p3 = pcompose p* [:*ub*, *Complex (Re lb − Re ub) 0*:]
and *pR3 = map-poly Re p3* **and** *pI3 = map-poly Im p3* **and** *gc3 = gcd pR3 pI3*
and *p4 = pcompose p* [:*Complex (Re lb) (Im ub)*, *Complex 0 (Im lb − Im ub)*:]
and *pR4 = map-poly Re p4* **and** *pI4 = map-poly Im p4* **and** *gc4 = gcd pR4 pI4*
have *gc1≠0 gc2≠0 gc3≠0 gc4≠0*
proof −
show *gc1≠0*
proof (*rule ccontr*)

```
      assume ¬ gc1 ≠ 0
      then have pI1 = 0 pR1 = 0 unfolding gc1-def by auto
      then have p1 = 0 unfolding pI1-def pR1-def
        by (metis cpoly-of-decompose map-poly-0)
      then have p=0 unfolding p1-def using ‹Re lb < Re ub›
        by (auto elim!:pcompose-eq-0 simp add:Complex-eq-0)
      then show False using ‹p≠0› by simp
    qed
    show gc2≠0
    proof (rule ccontr)
      assume ¬ gc2 ≠ 0
      then have pI2 = 0 pR2 = 0 unfolding gc2-def by auto
      then have p2 = 0 unfolding pI2-def pR2-def
        by (metis cpoly-of-decompose map-poly-0)
      then have p=0 unfolding p2-def using ‹Im lb < Im ub›
        by (auto elim!:pcompose-eq-0 simp add:Complex-eq-0)
      then show False using ‹p≠0› by simp
    qed
    show gc3≠0
    proof (rule ccontr)
      assume ¬ gc3 ≠ 0
      then have pI3 = 0 pR3 = 0 unfolding gc3-def by auto
      then have p3 = 0 unfolding pI3-def pR3-def
        by (metis cpoly-of-decompose map-poly-0)
      then have p=0 unfolding p3-def using ‹Re lb < Re ub›
        by (auto elim!:pcompose-eq-0 simp add:Complex-eq-0)
      then show False using ‹p≠0› by simp
    qed
    show gc4≠0
    proof (rule ccontr)
      assume ¬ gc4 ≠ 0
      then have pI4 = 0 pR4 = 0 unfolding gc4-def by auto
      then have p4 = 0 unfolding pI4-def pR4-def
        by (metis cpoly-of-decompose map-poly-0)
      then have p=0 unfolding p4-def using ‹Im lb < Im ub›
        by (auto elim!:pcompose-eq-0 simp add:Complex-eq-0)
      then show False using ‹p≠0› by simp
    qed
  qed
  define sms where
    sms = (changes-alt-itv-smods 0 1 (pR1 div gc1) (pI1 div gc1)
          + changes-alt-itv-smods 0 1 (pR2 div gc2) (pI2 div gc2)
          + changes-alt-itv-smods 0 1 (pR3 div gc3) (pI3 div gc3)
          + changes-alt-itv-smods 0 1 (pR4 div gc4) (pI4 div gc4))
  have proots-rectangle p lb ub = (∑ r∈proots p. winding-number rec r * (order
r p))
  proof −
    have winding-number rec x * of-nat (order x p) = 0
      when x∈proots p − proots-within p (box lb ub) for x
```

```
    proof −
      have ∗:cbox lb ub = box lb ub ∪ path-image rec
      proof −
        have x∈cbox lb ub when x∈box lb ub ∪ path-image rec for x
          using that ‹Re lb<Re ub› ‹Im lb<Im ub›
          unfolding box-def cbox-def Basis-complex-def rec-def l1-def l2-def l3-def
l4-def
            apply (auto simp add:path-image-join closed-segment-degen-complex)

           apply (subst (asm) closed-segment-commute,simp add: closed-segment-degen-complex)+
          done
        moreover have x∈box lb ub ∪ path-image rec when x∈cbox lb ub for x
          using that
          unfolding box-def cbox-def Basis-complex-def rec-def l1-def l2-def l3-def
l4-def
          apply (auto simp add:path-image-join closed-segment-degen-complex)
         apply (subst (asm) (1 2) closed-segment-commute,simp add:closed-segment-degen-complex)+
          done
        ultimately show ?thesis by auto
      qed
      moreover have x∉path-image rec
        using path-no-proots that by auto
      ultimately have x∉cbox lb ub using that by simp
      from winding-number-zero-outside[OF valid-path-imp-path[OF valid] - loop
this,simplified] ∗
      have winding-number rec x = 0 by auto
      then show ?thesis by auto
    qed
    moreover have of-nat (order x p) = winding-number rec x ∗ of-nat (order x
p) when
      x ∈ proots-within p (box lb ub) for x
    proof −
      have x∈box lb ub using that unfolding proots-within-def by auto
       then have order-asms:Re lb < Re x Re x < Re ub Im lb < Im x Im x <
Im ub
        by (auto simp add:box-def Basis-complex-def)
      have winding-number rec x = 1
        unfolding rec-def l1-def l2-def l3-def l4-def
      proof eval-winding
        let ?l1 = linepath lb (Complex (Re ub) (Im lb))
        and ?l2 = linepath (Complex (Re ub) (Im lb)) ub
        and ?l3 = linepath ub (Complex (Re lb) (Im ub))
        and ?l4 = linepath (Complex (Re lb) (Im ub)) lb
        show l1: x ∉ path-image ?l1 and l2: x ∉ path-image ?l2 and
            l3: x ∉ path-image ?l3 and l4:x ∉ path-image ?l4
          using no-proots that unfolding no-proots-line-def by auto
        show − of-real (cindex-pathE ?l1 x + (cindex-pathE ?l2 x +
          (cindex-pathE ?l3 x + cindex-pathE ?l4 x))) = 2 ∗ 1
        proof −
```

```
          have (Im x − Im ub) ∗ (Re ub − Re lb) < 0
            using mult-less-0-iff order-asms(1) order-asms(2) order-asms(4) by
fastforce
          then have cindex-pathE ?l3 x = −1
           apply (subst cindex-pathE-linepath)
           using l3 by (auto simp add:algebra-simps order-asms)
          moreover have (Im lb − Im x) ∗ (Re ub − Re lb) <0
            using mult-less-0-iff order-asms(1) order-asms(2) order-asms(3) by
fastforce
          then have cindex-pathE ?l1 x = −1
           apply (subst cindex-pathE-linepath)
           using l1 by (auto simp add:algebra-simps order-asms)
          moreover have cindex-pathE ?l2 x = 0
           apply (subst cindex-pathE-linepath)
           using l2 order-asms by auto
          moreover have cindex-pathE ?l4 x = 0
           apply (subst cindex-pathE-linepath)
           using l4 order-asms by auto
          ultimately show ?thesis by auto
        qed
       qed
       then show ?thesis by auto
     qed
     ultimately show ?thesis using ‹p≠0›
     unfolding proots-rectangle-def proots-count-def
      by (auto intro!: sum.mono-neutral-cong-left[of proots p proots-within p (box
lb ub)])
   qed
   also have ... = 1/(2 ∗ of-real pi ∗ i) ∗contour-integral rec (λx. deriv (poly p)
x / poly p x)
   proof −
     have contour-integral rec (λx. deriv (poly p) x / poly p x) = 2 ∗ of-real pi ∗
i
           ∗ (∑ x | poly p x = 0. winding-number rec x ∗ of-int (zorder (poly p)
x))
     proof (rule argument-principle[of UNIV poly p {} λ-. 1 rec,simplified])
     show connected (UNIV::complex set) using connected-UNIV[where 'a=complex]
.
       show path-image rec ⊆ UNIV − {x. poly p x = 0}
         using path-no-proots unfolding proots-within-def by auto
       show finite {x. poly p x = 0} by (simp add: poly-roots-finite that(3))
     qed
     also have  ... = 2 ∗ of-real pi ∗ i ∗ (∑ x∈proots p. winding-number rec x ∗
(order x p))
       unfolding proots-within-def
       apply (auto intro!:sum.cong simp add:order-zorder[OF ‹p≠0›] )
       by (metis nat-eq-iff2 of-nat-nat order-root order-zorder that(3))
     finally show ?thesis by auto
   qed
```

```
  also have ... = winding-number (poly p ∘ rec) 0
  proof −
    have 0 ∉ path-image (poly p ∘ rec)
      using path-no-proots unfolding path-image-compose proots-within-def by
fastforce
    from winding-number-comp[OF - poly-holomorphic-on - - this,of UNIV,simplified]
    show ?thesis by auto
  qed
  also have winding-eq:... = − cindex-pathE (poly p ∘ rec) 0 / 2
  proof (rule winding-number-cindex-pathE)
    show finite-ReZ-segments (poly p ∘ rec) 0
      unfolding rec-def path-compose-join
      apply (fold g1-def g2-def g3-def g4-def)
      by (auto intro!: finite-ReZ-segments-joinpaths path-join-imp)
    show valid-path (poly p ∘ rec)
      by (rule valid-path-compose-holomorphic[where S=UNIV]) auto
    show 0 ∉ path-image (poly p ∘ rec)
     using path-no-proots unfolding path-image-compose proots-def by fastforce
    show pathfinish (poly p ∘ rec) = pathstart (poly p ∘ rec)
       unfolding rec-def pathstart-compose pathfinish-compose  by (auto simp
add:l1-def l4-def)
  qed
  also have cindex-pathE-eq:... = of-int (− sms) / of-int 4
  proof −
  have cindex-pathE (poly p ∘ rec) 0 = cindex-pathE (g1+++g2+++g3+++g4)
0
      unfolding rec-def path-compose-join g1-def g2-def g3-def g4-def by simp
    also have ... = cindex-pathE g1 0 + cindex-pathE g2 0 + cindex-pathE g3 0
+ cindex-pathE g4 0
     by (subst cindex-pathE-joinpaths,auto intro!:finite-ReZ-segments-joinpaths)+
    also have ... = cindex-polyE 0 1 (pI1 div gc1) (pR1 div gc1)
                  + cindex-polyE 0 1 (pI2 div gc2) (pR2 div gc2)
                  + cindex-polyE 0 1 (pI3 div gc3) (pR3 div gc3)
                  + cindex-polyE 0 1 (pI4 div gc4) (pR4 div gc4)
    proof −
      have cindex-pathE g1 0 = cindex-polyE 0 1 (pI1 div gc1) (pR1 div gc1)
      proof −
        have ∗:g1 = poly p1 o of-real
          unfolding g1-def p1-def l1-def poly-linepath-comp
          by (subst (5) complex-surj[symmetric],simp)
       then have cindex-pathE g1 0 = cindexE 0 1 (λt. poly pI1 t / poly pR1 t)
          unfolding cindex-pathE-def pR1-def pI1-def
          by (simp add:Im-poly-of-real Re-poly-of-real)
       also have ... = cindex-polyE 0 1 pI1 pR1
          using cindexE-eq-cindex-polyE by auto
        also have ... = cindex-polyE 0 1 (pI1 div gc1) (pR1 div gc1)
          using ‹gc1≠0›
          apply (subst (2) cindex-polyE-mult-cancel[of gc1,symmetric])
          by (simp-all add: gc1-def)
```

```
    finally show ?thesis .
  qed
  moreover have cindex-pathE g2 0 = cindex-polyE 0 1 (pI2 div gc2) (pR2
div gc2)
  proof −
    have g2 = poly p2 o of-real
      unfolding g2-def p2-def l2-def poly-linepath-comp
      by (subst (5) complex-surj[symmetric],simp)
   then have cindex-pathE g2 0 = cindexE 0 1 (λt. poly pI2 t / poly pR2 t)
      unfolding cindex-pathE-def pR2-def pI2-def
      by (simp add:Im-poly-of-real Re-poly-of-real)
    also have ... = cindex-polyE 0 1 pI2 pR2
      using cindexE-eq-cindex-polyE by auto
    also have ... = cindex-polyE 0 1 (pI2 div gc2) (pR2 div gc2)
      using ‹gc2≠0›
      apply (subst (2) cindex-polyE-mult-cancel[of gc2,symmetric])
        by (simp-all add: gc2-def)
    finally show ?thesis .
  qed
  moreover have cindex-pathE g3 0 = cindex-polyE 0 1 (pI3 div gc3) (pR3
div gc3)
  proof −
    have g3 = poly p3 o of-real
      unfolding g3-def p3-def l3-def poly-linepath-comp
      by (subst (5) complex-surj[symmetric],simp)
   then have cindex-pathE g3 0 = cindexE 0 1 (λt. poly pI3 t / poly pR3 t)
      unfolding cindex-pathE-def pR3-def pI3-def
      by (simp add:Im-poly-of-real Re-poly-of-real)
    also have ... = cindex-polyE 0 1 pI3 pR3
      using cindexE-eq-cindex-polyE by auto
    also have ... = cindex-polyE 0 1 (pI3 div gc3) (pR3 div gc3)
      using ‹gc3≠0›
      apply (subst (2) cindex-polyE-mult-cancel[of gc3,symmetric])
      by (simp-all add: gc3-def)
    finally show ?thesis .
  qed
  moreover have cindex-pathE g4 0 = cindex-polyE 0 1 (pI4 div gc4) (pR4
div gc4)
  proof −
    have g4 = poly p4 o of-real
      unfolding g4-def p4-def l4-def poly-linepath-comp
      by (subst (5) complex-surj[symmetric],simp)
   then have cindex-pathE g4 0 = cindexE 0 1 (λt. poly pI4 t / poly pR4 t)
      unfolding cindex-pathE-def pR4-def pI4-def
      by (simp add:Im-poly-of-real Re-poly-of-real)
    also have ... = cindex-polyE 0 1 pI4 pR4
      using cindexE-eq-cindex-polyE by auto
    also have ... = cindex-polyE 0 1 (pI4 div gc4) (pR4 div gc4)
      using ‹gc4≠0›
```

```
        apply (subst (2) cindex-polyE-mult-cancel[of gc4,symmetric])
        by (simp-all add: gc4-def)
      finally show ?thesis .
    qed
    ultimately show ?thesis by auto
  qed
  also have ... = sms / 2
  proof −
    have cindex-polyE 0 1 (pI1 div gc1) (pR1 div gc1)
      = changes-alt-itv-smods 0 1 (pR1 div gc1) (pI1 div gc1) / 2
      apply (rule cindex-polyE-changes-alt-itv-mods)
      using ‹gc1≠0› unfolding gc1-def by (auto intro:div-gcd-coprime)
    moreover have cindex-polyE 0 1 (pI2 div gc2) (pR2 div gc2)
      = changes-alt-itv-smods 0 1 (pR2 div gc2) (pI2 div gc2) / 2
      apply (rule cindex-polyE-changes-alt-itv-mods)
      using ‹gc2≠0› unfolding gc2-def by (auto intro:div-gcd-coprime)
    moreover have cindex-polyE 0 1 (pI3 div gc3) (pR3 div gc3)
      = changes-alt-itv-smods 0 1 (pR3 div gc3) (pI3 div gc3) / 2
      apply (rule cindex-polyE-changes-alt-itv-mods)
      using ‹gc3≠0› unfolding gc3-def by (auto intro:div-gcd-coprime)
    moreover have cindex-polyE 0 1 (pI4 div gc4) (pR4 div gc4)
      = changes-alt-itv-smods 0 1 (pR4 div gc4) (pI4 div gc4) / 2
      apply (rule cindex-polyE-changes-alt-itv-mods)
      using ‹gc4≠0› unfolding gc4-def by (auto intro:div-gcd-coprime)
    ultimately show ?thesis unfolding sms-def by auto
  qed
  finally have ∗:cindex-pathE (poly p ∘ rec) 0 = real-of-int sms / 2 .
  show ?thesis
    apply (subst ∗)
    by auto
qed
finally have (of-nat::-⇒complex) (proots-rectangle p lb ub) = of-int (− sms)
/ of-int 4 .
moreover have 4 dvd sms
proof −
  have winding-number (poly p ∘ rec) 0 ∈ ℤ
  proof (rule integer-winding-number)
    show path (poly p ∘ rec)
   by (auto intro!:valid-path-compose-holomorphic[where S=UNIV] valid-path-imp-path)
    show pathfinish (poly p ∘ rec) = pathstart (poly p ∘ rec)
      unfolding rec-def path-compose-join
      by (auto simp add:l1-def l4-def pathfinish-compose pathstart-compose)
    show 0 ∉ path-image (poly p ∘ rec)
    using path-no-proots unfolding path-image-compose proots-def by fastforce
  qed
  then have of-int (− sms) / of-int 4 ∈ (ℤ::complex set)
    by (simp only: winding-eq cindex-pathE-eq)
  then show ?thesis by (subst (asm) dvd-divide-Ints-iff[symmetric],auto)
qed
```

```
  ultimately have proots-rectangle p lb ub = nat (− sms div 4)
    apply (subst (asm) of-int-div-field[symmetric])
    by (simp,metis nat-int of-int-eq-iff of-int-of-nat-eq)
  then show ?thesis
    unfolding Let-def
   apply (fold p1-def p2-def p3-def p4-def pI1-def pR1-def pI2-def pR2-def pI3-def
pR3-def
       pI4-def pR4-def gc1-def gc2-def gc3-def gc4-def)
    apply (fold sms-def)
    using that by auto
 qed
 ultimately show ?thesis by fastforce
qed


lemma proots-rectangle-code2[code]:
 proots-rectangle p lb ub = (if Re lb < Re ub ∧ Im lb < Im ub then
          if p≠0 then
          if poly p lb ≠ 0 ∧ poly p (Complex (Re ub) (Im lb)) ≠0
            ∧ poly p ub ≠0 ∧ poly p (Complex (Re lb) (Im ub)) ≠0
          then
            (let p1 = pcompose p [:lb,  Complex (Re ub − Re lb) 0:];
              pR1 = map-poly Re p1; pI1 = map-poly Im p1; gc1 = gcd pR1 pI1;
             p2 = pcompose p [:Complex (Re ub) (Im lb), Complex 0 (Im ub − Im
lb):];
              pR2 = map-poly Re p2; pI2 = map-poly Im p2; gc2 = gcd pR2 pI2;
              p3 = pcompose p [:ub, Complex (Re lb − Re ub) 0:];
              pR3 = map-poly Re p3; pI3 = map-poly Im p3; gc3 = gcd pR3 pI3;
             p4 = pcompose p [:Complex (Re lb) (Im ub), Complex 0 (Im lb − Im
ub):];
              pR4 = map-poly Re p4; pI4 = map-poly Im p4; gc4 = gcd pR4 pI4
            in
              if changes-itv-smods 0 1 gc1 (pderiv gc1) = 0
                ∧ changes-itv-smods 0 1 gc2 (pderiv gc2) = 0
                ∧ changes-itv-smods 0 1 gc3 (pderiv gc3) = 0
                ∧ changes-itv-smods 0 1 gc4 (pderiv gc4) = 0
              then
                nat (− (changes-alt-itv-smods 0 1 (pR1 div gc1) (pI1 div gc1)
                 + changes-alt-itv-smods 0 1 (pR2 div gc2) (pI2 div gc2)
                 + changes-alt-itv-smods 0 1 (pR3 div gc3) (pI3 div gc3)
                 + changes-alt-itv-smods 0 1 (pR4 div gc4) (pI4 div gc4)) div 4)
             else Code.abort (STR ''proots-rectangle fails when there is a root on
the border.'')
                   (λ-. proots-rectangle p lb ub))
           else Code.abort (STR ''proots-rectangle fails when there is a root on the
border.'')
            (λ-. proots-rectangle p lb ub)
          else Code.abort (STR ''proots-rectangle fails when p=0.'')
          (λ-. proots-rectangle p lb ub)
```

```
          else 0)
proof −
  define p1 pR1 pI1 gc1
         p2 pR2 pI2 gc2
         p3 pR3 pI3 gc3
         p4 pR4 pI4 gc4
      where p1 = pcompose p [:lb,  Complex (Re ub − Re lb) 0:]
            and pR1 = map-poly Re p1 and pI1 = map-poly Im p1 and gc1 =
gcd pR1 pI1
            and p2 = pcompose p [:Complex (Re ub) (Im lb), Complex 0 (Im ub −
Im lb):]
            and pR2 = map-poly Re p2 and pI2 = map-poly Im p2 and gc2 =
gcd pR2 pI2
            and p3 = pcompose p [:ub, Complex (Re lb − Re ub) 0:]
            and pR3 = map-poly Re p3 and pI3 = map-poly Im p3 and gc3 =
gcd pR3 pI3
            and p4 = pcompose p [:Complex (Re lb) (Im ub), Complex 0 (Im lb −
Im ub):]
            and pR4 = map-poly Re p4 and pI4 = map-poly Im p4 and gc4 =
gcd pR4 pI4
  define sms where
      sms = (− (changes-alt-itv-smods 0 1 (pR1 div gc1) (pI1 div gc1) +
                  changes-alt-itv-smods 0 1 (pR2 div gc2) (pI2 div gc2) +
                  changes-alt-itv-smods 0 1 (pR3 div gc3) (pI3 div gc3) +
                  changes-alt-itv-smods 0 1 (pR4 div gc4) (pI4 div gc4)) div
               4)
  have more-folds:p1 = p ∘p [:lb, Complex (Re ub) (Im lb) − lb:]
    p2 = p ∘p [:Complex (Re ub) (Im lb), ub − Complex (Re ub) (Im lb):]
    p3 = p ∘p [:ub, Complex (Re lb) (Im ub) − ub:]
    p4 = p ∘p [:Complex (Re lb) (Im ub), lb − Complex (Re lb) (Im ub):]
    subgoal unfolding p1-def
      by (subst (10) complex-surj[symmetric],auto simp add:minus-complex.code)
    subgoal unfolding p2-def by (subst (10) complex-surj[symmetric],auto)
    subgoal unfolding p3-def by (subst (10) complex-surj[symmetric],auto simp
add:minus-complex.code)
    subgoal unfolding p4-def by (subst (10) complex-surj[symmetric],auto)
    done
  show ?thesis
    apply (subst proots-rectangle-code1)
    apply (unfold no-proots-line-code Let-def)
    apply (fold p1-def p2-def p3-def p4-def pI1-def pR1-def pI2-def pR2-def pI3-def
pR3-def
          pI4-def pR4-def gc1-def gc2-def gc3-def gc4-def more-folds)
    apply (fold sms-def)
    by presburger
qed
```

3.7 Polynomial roots on the upper half-plane

— Roots counted WITH multiplicity
definition *proots-upper* ::*complex poly* ⇒ *nat* **where**
proots-upper p= *proots-count p* {*z*. *Im z*>*0*}

— Roots counted WITHOUT multiplicity
definition *proots-upper-card*::*complex poly* ⇒ *nat* **where**
proots-upper-card p = *card* (*proots-within p* {*x*. *Im x* >*0*})

lemma *Im-Ln-tendsto-at-top*: ((λ*x*. *Im* (*Ln* (*Complex a x*))) ⟶ *pi*/*2*) *at-top*
proof (*cases a*=*0*)
case *False*
define *f* **where** *f*=(λ*x*. *if a*>*0 then arctan* (*x*/*a*) *else arctan* (*x*/*a*) + *pi*)
define *g* **where** *g*=(λ*x*. *Im* (*Ln* (*Complex a x*)))
have (*f* ⟶ *pi* / *2*) *at-top*
proof (*cases a*>*0*)
case *True*
then have (*f* ⟶ *pi* / *2*) *at-top* ⟷ ((λ*x*. *arctan* (*x* ∗ *inverse a*)) ⟶ *pi* / *2*) *at-top*
unfolding *f-def field-class.field-divide-inverse* **by** *auto*
also have ... ⟷ (*arctan* ⟶ *pi* / *2*) *at-top*
apply (*subst filterlim-at-top-linear-iff*[*of inverse a arctan 0 nhds* (*pi*/*2*),*simplified*])
using *True* **by** *auto*
also have ... **using** *tendsto-arctan-at-top* **.**
finally show *?thesis* **.**
next
case *False*
then have (*f* ⟶ *pi* / *2*) *at-top* ⟷ ((λ*x*. *arctan* (*x* ∗ *inverse a*) + *pi*) ⟶ *pi* / *2*) *at-top*
unfolding *f-def field-class.field-divide-inverse* **by** *auto*
also have ... ⟷ ((λ*x*. *arctan* (*x* ∗ *inverse a*)) ⟶ − *pi* / *2*) *at-top*
apply (*subst tendsto-add-const-iff*[*of* −*pi*,*symmetric*])
by *auto*
also have ... ⟷ (*arctan* ⟶ − *pi* / *2*) *at-bot*
apply (*subst filterlim-at-top-linear-iff*[*of inverse a arctan 0*,*simplified*])
using *False* ‹*a*≠*0*› **by** *auto*
also have ... **using** *tendsto-arctan-at-bot* **by** *simp*
finally show *?thesis* **.**
qed
moreover have ∀ *F x in at-top*. *f x* = *g x*
unfolding *f-def g-def* **using** ‹*a*≠*0*›
apply (*subst Im-Ln-eq*)
subgoal for *x* **using** *Complex-eq-0* **by** *blast*
subgoal unfolding *eventually-at-top-linorder* **by** *auto*
done
ultimately show *?thesis*
using *tendsto-cong*[*of f g at-top*] **unfolding** *g-def* **by** *auto*
next
case *True*

```
  show ?thesis
    apply (rule Lim-eventually)
    apply (rule eventually-at-top-linorderI[of 1])
    using True by (subst Im-Ln-eq,auto simp add:Complex-eq-0)
qed

lemma Im-Ln-tendsto-at-bot: ((λx. Im (Ln (Complex a x))) ⟶ − pi/2 ) at-bot

proof (cases a=0)
  case False
  define f where f=(λx. if a>0 then arctan (x/a) else arctan (x/a) − pi)
  define g where g=(λx. Im (Ln (Complex a x)))
  have (f ⟶ − pi / 2) at-bot
  proof (cases a>0)
    case True
    then have (f ⟶ − pi / 2) at-bot ⟷ ((λx. arctan (x ∗ inverse a)) ⟶
− pi / 2) at-bot
      unfolding f-def field-class.field-divide-inverse by auto
    also have ... ⟷ (arctan ⟶ − pi / 2) at-bot
      apply (subst filterlim-at-bot-linear-iff[of inverse a arctan 0,simplified])
      using True by auto
    also have ... using tendsto-arctan-at-bot by simp
    finally show ?thesis .
  next
    case False
    then have (f ⟶ − pi / 2) at-bot ⟷ ((λx. arctan (x ∗ inverse a) − pi)
⟶ − pi / 2) at-bot
      unfolding f-def field-class.field-divide-inverse by auto
    also have ... ⟷ ((λx. arctan (x ∗ inverse a)) ⟶ pi / 2) at-bot
      apply (subst tendsto-add-const-iff[of pi,symmetric])
      by auto
    also have ... ⟷ (arctan ⟶ pi / 2) at-top
      apply (subst filterlim-at-bot-linear-iff[of inverse a arctan 0,simplified])
      using False ‹a≠0› by auto
    also have ... using tendsto-arctan-at-top by simp
    finally show ?thesis .
  qed
  moreover have ∀F x in at-bot. f x = g x
    unfolding f-def g-def using ‹a≠0›
    apply (subst Im-Ln-eq)
    subgoal for x using Complex-eq-0 by blast
   subgoal unfolding eventually-at-bot-linorder by (auto intro:exI[where x=−1])
    done
  ultimately show ?thesis
    using tendsto-cong[of f g at-bot] unfolding g-def by auto
next
  case True
  show ?thesis
    apply (rule Lim-eventually)
```

```
    apply (rule eventually-at-bot-linorderI[of −1])
    using True by (subst Im-Ln-eq,auto simp add:Complex-eq-0)
qed

lemma Re-winding-number-tendsto-part-circlepath:
  shows ((λr. Re (winding-number (part-circlepath z0 r 0 pi ) a)) ⟶ 1/2 )
at-top
proof (cases Im z0≤Im a)
  case True
  define g1 where g1=(λr. part-circlepath z0 r 0 pi)
  define g2 where g2=(λr. part-circlepath z0 r pi (2∗pi))
  define f1 where f1=(λr. Re (winding-number (g1 r ) a))
  define f2 where f2=(λr. Re (winding-number (g2 r) a))
  have (f2 ⟶ 1/2 ) at-top
  proof −
    define h1 where h1 = (λr. Im (Ln (Complex ( Im a−Im z0) (Re z0 − Re a
+ r))))
    define h2 where h2= (λr. Im (Ln (Complex (  Im a −Im z0) (Re z0 − Re
a − r))))
    have ∀F x in at-top. f2 x = (h1 x − h2 x) / (2 ∗ pi)
    proof (rule eventually-at-top-linorderI[of cmod (a−z0) + 1])
      fix r assume asm:r ≥ cmod (a − z0) + 1
      have Im p ≤ Im a when p∈path-image (g2 r) for p
      proof −
        obtain t where p-def:p=z0 + of-real r ∗ exp (i ∗ of-real t) and pi≤t
t≤2∗pi
          using ‹p∈path-image (g2 r)›
          unfolding g2-def path-image-part-circlepath[of pi 2∗pi,simplified]
          by auto
        then have Im p=Im z0 + sin t ∗ r by (auto simp add:Im-exp)
        also have ... ≤ Im z0
        proof −
          have sin t≤0 using ‹pi≤t› ‹t≤2∗pi› sin-le-zero by fastforce
          moreover have r≥0
         using asm by (metis add.inverse-inverse add.left-neutral add-uminus-conv-diff
              diff-ge-0-iff-ge norm-ge-zero order-trans zero-le-one)
          ultimately have sin t ∗ r≤0 using mult-le-0-iff by blast
          then show ?thesis by auto
        qed
        also have ... ≤ Im a using True .
        finally show ?thesis .
      qed
      moreover have valid-path (g2 r) unfolding g2-def by auto
      moreover have a ∉ path-image (g2 r)
        unfolding g2-def
        apply (rule not-on-circlepathI)
        using asm by auto
      moreover have [symmetric]:Im (Ln (i ∗ pathfinish (g2 r) − i ∗ a)) = h1 r
        unfolding h1-def g2-def
```

```
      apply (simp only:pathfinish-pathstart-partcirclepath-simps)
      apply (subst (4 10) complex-eq)
      by (auto simp add:algebra-simps Complex-eq)
    moreover have [symmetric]:Im (Ln (i * pathstart (g2 r) - i * a)) = h2 r
      unfolding h2-def g2-def
      apply (simp only:pathfinish-pathstart-partcirclepath-simps)
      apply (subst (4 10) complex-eq)
      by (auto simp add:algebra-simps Complex-eq)
    ultimately show f2 r = (h1 r - h2 r) / (2 * pi)
      unfolding f2-def
      apply (subst Re-winding-number-half-lower)
      by (auto simp add:exp-Euler algebra-simps)
  qed
  moreover have ((λx. (h1 x - h2 x) / (2 * pi)) ⟶ 1/2 ) at-top
  proof -
    have (h1 ⟶ pi/2) at-top
      unfolding h1-def
    apply (subst filterlim-at-top-linear-iff[of 1 - Re a - Re z0 ,simplified,symmetric])

      using Im-Ln-tendsto-at-top by (simp del:Complex-eq)
    moreover have (h2 ⟶ - pi/2) at-top
      unfolding h2-def
    apply (subst filterlim-at-bot-linear-iff[of - 1 - - Re a + Re z0 ,simplified,symmetric])

      using Im-Ln-tendsto-at-bot by (simp del:Complex-eq)
    ultimately have ((λx. h1 x- h2 x) ⟶ pi) at-top
      by (auto intro: tendsto-eq-intros)
    then show ?thesis
      by (auto intro: tendsto-eq-intros)
  qed
  ultimately show ?thesis by (auto dest:tendsto-cong)
qed
moreover have ∀F r in at-top. f2 r = 1 - f1 r
proof (rule eventually-at-top-linorderI[of cmod (a-z0) + 1])
  fix r assume asm:r ≥ cmod (a - z0) + 1
  have f1 r + f2 r = Re(winding-number (g1 r +++ g2 r) a)
    unfolding f1-def f2-def g1-def g2-def
    apply (subst winding-number-join)
    using asm by (auto intro!:not-on-circlepathI)
  also have ... = Re(winding-number (circlepath z0 r) a)
  proof -
    have g1 r +++ g2 r = circlepath z0 r
        unfolding circlepath-def g1-def g2-def joinpaths-def part-circlepath-def
linepath-def
      by (auto simp add:field-simps)
    then show ?thesis by auto
  qed
  also have ... = 1
  proof -
```

```
    have winding-number (circlepath z0 r) a = 1
      apply (rule winding-number-circlepath)
      using asm by auto
    then show ?thesis by auto
   qed
   finally have f1 r+f2 r=1 .
   then show  f2 r = 1 − f1 r by auto
  qed
  ultimately have ((λr. 1 − f1 r) ⟶ 1/2 ) at-top
   using tendsto-cong[of f2 λr. 1 − f1 r at-top] by auto
  then have (f1 ⟶ 1/2 ) at-top
   apply (rule-tac tendsto-minus-cancel)
   apply (subst tendsto-add-const-iff[of 1,symmetric])
   by auto
  then show ?thesis unfolding f1-def g1-def by auto
next
  case False
  define g where g=(λr. part-circlepath z0 r 0 pi)
  define f where f=(λr. Re (winding-number (g r) a))
  have (f ⟶ 1/2 ) at-top
  proof −
   define h1 where h1 = (λr. Im (Ln (Complex ( Im z0−Im a) (Re a − Re z0
+ r))))
   define h2 where h2= (λr. Im (Ln (Complex (  Im z0 −Im a ) (Re a − Re
z0 − r))))
   have ∀F x in at-top. f x = (h1 x − h2 x) / (2 ∗ pi)
   proof (rule eventually-at-top-linorderI[of cmod (a−z0) + 1])
    fix r assume asm:r ≥ cmod (a − z0) + 1
    have Im p ≥ Im a when p∈path-image (g r) for p
    proof −
     obtain t where p-def:p=z0 + of-real r ∗ exp (i ∗ of-real t) and 0≤t t≤pi
      using ‹p∈path-image (g r)›
      unfolding g-def path-image-part-circlepath[of 0 pi,simplified]
      by auto
     then have Im p=Im z0 + sin t ∗ r by (auto simp add:Im-exp)
     moreover have sin t ∗ r≥0
     proof −
      have sin t≥0 using ‹0≤t› ‹t≤pi› sin-ge-zero by fastforce
      moreover have r≥0
     using asm by (metis add.inverse-inverse add.left-neutral add-uminus-conv-diff
          diff-ge-0-iff-ge norm-ge-zero order-trans zero-le-one)
      ultimately have sin t ∗ r≥0 by simp
      then show ?thesis by auto
     qed
     ultimately show ?thesis using False by auto
    qed
    moreover have valid-path (g r) unfolding g-def by auto
    moreover have a ∉ path-image (g r)
     unfolding g-def
```

```
      apply (rule not-on-circlepathI)
      using asm by auto
    moreover have [symmetric]:Im (Ln (i * a - i * pathfinish (g r))) = h1 r
      unfolding h1-def g-def
      apply (simp only:pathfinish-pathstart-partcirclepath-simps)
      apply (subst (4 9) complex-eq)
      by (auto simp add:algebra-simps Complex-eq)
    moreover have [symmetric]:Im (Ln (i * a - i * pathstart (g r))) = h2 r
      unfolding h2-def g-def
      apply (simp only:pathfinish-pathstart-partcirclepath-simps)
      apply (subst (4 9) complex-eq)
      by (auto simp add:algebra-simps Complex-eq)
    ultimately show f r = (h1 r - h2 r) / (2 * pi)
      unfolding f-def
      apply (subst Re-winding-number-half-upper)
      by (auto simp add:exp-Euler algebra-simps)
  qed
  moreover have ((λx. (h1 x - h2 x) / (2 * pi)) ⟶ 1/2 ) at-top
  proof -
    have (h1 ⟶ pi/2) at-top
      unfolding h1-def
    apply (subst filterlim-at-top-linear-iff[of 1 - - Re a + Re z0 ,simplified,symmetric])

      using Im-Ln-tendsto-at-top by (simp del:Complex-eq)
    moreover have (h2 ⟶ - pi/2) at-top
      unfolding h2-def
    apply (subst filterlim-at-bot-linear-iff[of - 1 - Re a - Re z0 ,simplified,symmetric])

      using Im-Ln-tendsto-at-bot by (simp del:Complex-eq)
    ultimately have ((λx. h1 x- h2 x) ⟶ pi) at-top
      by (auto intro: tendsto-eq-intros)
    then show ?thesis
      by (auto intro: tendsto-eq-intros)
  qed
  ultimately show ?thesis by (auto dest:tendsto-cong)
 qed
 then show ?thesis unfolding f-def g-def by auto
qed

lemma not-image-at-top-poly-part-circlepath:
 assumes degree p>0
 shows ∀ F r in at-top. b∉path-image (poly p o part-circlepath z0 r st tt)
proof -
 have finite (proots (p-[:b:]))
   apply (rule finite-proots)
   using assms by auto
 from finite-ball-include[OF this]
 obtain R::real where R>0 and R-ball:proots (p-[:b:]) ⊆ ball z0 R by auto
 show ?thesis
```

```
  proof (rule eventually-at-top-linorderI[of R])
    fix r assume r≥R
    show  b∉path-image (poly p o part-circlepath z0 r st tt)
      unfolding path-image-compose
    proof clarify
      fix x assume asm:b = poly p x x ∈ path-image (part-circlepath z0 r st tt)
      then have x∈proots (p−[:b:]) unfolding proots-def by auto
      then have x∈ball z0 r using R-ball ‹r≥R› by auto
      then have cmod (x− z0) < r
        by (simp add: dist-commute dist-norm)
      moreover have cmod (x − z0) = r
        using asm(2) in-path-image-part-circlepath ‹R>0› ‹r≥R›  by auto
      ultimately show False by auto
    qed
  qed
qed

lemma not-image-poly-part-circlepath:
  assumes degree p>0
  shows ∃ r>0.  b∉path-image (poly p o part-circlepath z0 r st tt)
proof −
  have finite (proots (p−[:b:]))
    apply (rule finite-proots)
    using assms by auto
  from finite-ball-include[OF this]
  obtain r::real where r>0 and r-ball:proots (p−[:b:]) ⊆ ball z0 r by auto
  have b∉path-image (poly p o part-circlepath z0 r st tt)
    unfolding path-image-compose
  proof clarify
    fix x assume asm:b = poly p x x ∈ path-image (part-circlepath z0 r st tt)
    then have x∈proots (p−[:b:]) unfolding proots-def by auto
    then have x∈ball z0 r using r-ball by auto
    then have cmod (x− z0) < r
      by (simp add: dist-commute dist-norm)
    moreover have cmod (x − z0) = r
      using asm(2) in-path-image-part-circlepath ‹r>0› by auto
    ultimately show False by auto
  qed
  then show ?thesis using ‹r>0› by blast
qed

lemma Re-winding-number-poly-part-circlepath:
  assumes degree p>0
  shows ((λr. Re (winding-number (poly p o part-circlepath z0 r 0 pi) 0)) ⟶
degree p/2 ) at-top
using assms
proof (induct rule:poly-root-induct-alt)
  case 0
  then show ?case by auto
```

```
next
  case (no-proots p)
  then have False
   using Fundamental-Theorem-Algebra.fundamental-theorem-of-algebra constant-degree
neq0-conv
    by blast
  then show ?case by auto
next
  case (root a p)
  define g where g = (λr. part-circlepath z0 r 0 pi)
  define q where q=[:− a, 1:] ∗ p
  define w where w = (λr. winding-number (poly q ∘ g r) 0)
  have ?case when degree p=0
  proof −
   obtain pc where pc-def:p=[:pc:] using ‹degree p = 0› degree-eq-zeroE by blast
    then have pc≠0 using root(2) by auto
    have ∀F r in at-top. Re (w r) = Re (winding-number (g r) a)
    proof (rule eventually-at-top-linorderI[of cmod ((pc ∗ a) / pc − z0) + 1])
      fix r::real assume asm:cmod ((pc ∗ a) / pc − z0) + 1 ≤ r
      have w r =  winding-number ((λx. pc∗x − pc∗a) ∘ (g r)) 0
        unfolding w-def pc-def g-def q-def
        apply auto
      by (metis (no-types, hide-lams) add.right-neutral mult.commute mult-zero-right

            poly-0 poly-pCons uminus-add-conv-diff)
      also have ... =  winding-number (g r) a
        apply (subst winding-number-comp-linear[where b=−pc∗a,simplified])
        subgoal using ‹pc≠0› .
        subgoal unfolding g-def by auto
        subgoal unfolding g-def
          apply (rule not-on-circlepathI)
          using asm by auto
        subgoal using ‹pc≠0› by (auto simp add:field-simps)
        done
      finally have w r = winding-number (g r) a  .
      then show Re (w r) = Re (winding-number (g r) a) by simp
    qed
    moreover have ((λr. Re (winding-number (g r) a)) ⟶ 1/2) at-top
      using Re-winding-number-tendsto-part-circlepath unfolding g-def by auto
    ultimately have ((λr. Re (w r)) ⟶ 1/2) at-top
      by (auto dest!:tendsto-cong)
    moreover have degree ([:− a, 1:] ∗ p) = 1 unfolding pc-def using ‹pc≠0›
by auto
    ultimately show ?thesis unfolding w-def g-def comp-def q-def by simp
  qed
  moreover have ?case when degree p>0
  proof −
    have ∀F r in at-top. 0 ∉ path-image (poly q ∘ g r)
      unfolding g-def
```

```
    apply (rule not-image-at-top-poly-part-circlepath)
    unfolding q-def using root.prems by blast
  then have ∀F r in at-top. Re (w r) = Re (winding-number (g r) a)
        + Re (winding-number (poly p ∘ g r) 0)
  proof (rule eventually-mono)
    fix r assume asm:0 ∉ path-image (poly q ∘ g r)
    define cc where cc= 1 / (of-real (2 * pi) * i)
    define pf where pf=(λw. deriv (poly p) w / poly p w)
    define af where af=(λw. 1/(w−a))
    have w r = cc * contour-integral (g r) (λw. deriv (poly q) w / poly q w)
      unfolding w-def
      apply (subst winding-number-comp[of UNIV,simplified])
      using asm unfolding g-def cc-def by auto
    also have ... = cc * contour-integral (g r) (λw. deriv (poly p) w / poly p w
+ 1/(w−a))
    proof −
      have contour-integral (g r) (λw. deriv (poly q) w / poly q w)
         = contour-integral (g r) (λw. deriv (poly p) w / poly p w + 1/(w−a))
      proof (rule contour-integral-eq)
        fix x assume x ∈ path-image (g r)
        have deriv (poly q) x = deriv (poly p) x * (x−a) + poly p x
        proof −
          have poly q = (λx. (x−a) * poly p x)
            apply (rule ext)
            unfolding q-def by (auto simp add:algebra-simps)
          then show ?thesis
            apply simp
            apply (subst deriv-mult[of λx. x− a - poly p])
            by (auto intro:derivative-intros)
        qed
        moreover have poly p x≠0 ∧ x−a≠0
        proof (rule ccontr)
          assume ¬ (poly p x ≠ 0 ∧ x − a ≠ 0)
          then have poly q x=0 unfolding q-def by auto
          then have 0∈poly q ‘ path-image (g r)
            using ‹x ∈ path-image (g r)› by auto
          then show False using ‹0 ∉ path-image (poly q ∘ g r)›
            unfolding path-image-compose by auto
        qed
        ultimately show deriv (poly q) x / poly q x = deriv (poly p) x / poly p
x + 1 / (x − a)
          unfolding q-def by (auto simp add:field-simps)
      qed
      then show ?thesis by auto
    qed
    also have ... = cc * contour-integral (g r) (λw. deriv (poly p) w / poly p w)
       + cc * contour-integral (g r) (λw. 1/(w−a))
    proof (subst contour-integral-add)
      have continuous-on (path-image (g r)) (λw. deriv (poly p) w)
```

```
      unfolding deriv-pderiv by (intro continuous-intros)
    moreover have ∀ w∈path-image (g r). poly p w ≠ 0
      using asm unfolding q-def path-image-compose by auto
    ultimately show (λw. deriv (poly p) w / poly p w) contour-integrable-on
g r
      unfolding g-def
    by (auto intro!: contour-integrable-continuous-part-circlepath continuous-intros)
    show (λw. 1 / (w − a)) contour-integrable-on g r
      apply (rule contour-integrable-inversediff)
      subgoal unfolding g-def by auto
      subgoal using asm unfolding q-def path-image-compose by auto
      done
  qed (auto simp add:algebra-simps)
  also have ... = winding-number (g r) a + winding-number (poly p o g r) 0
  proof −
    have winding-number (poly p o g r) 0
       = cc ∗ contour-integral (g r) (λw. deriv (poly p) w / poly p w)
      apply (subst winding-number-comp[of UNIV,simplified])
     using ‹0 ∉ path-image (poly q ∘ g r)› unfolding path-image-compose q-def
g-def cc-def
      by auto
    moreover have winding-number (g r) a = cc ∗ contour-integral (g r) (λw.
1/(w−a))
      apply (subst winding-number-valid-path)
     using ‹0 ∉ path-image (poly q ∘ g r)› unfolding path-image-compose q-def
g-def cc-def
      by auto
    ultimately show ?thesis by auto
  qed
 finally show Re (w r) = Re (winding-number (g r) a) + Re (winding-number
(poly p ∘ g r) 0)
    by auto
qed
moreover have ((λr. Re (winding-number (g r) a)
        + Re (winding-number (poly p ∘ g r) 0)) ⟶ degree q / 2) at-top
proof −
  have ((λr. Re (winding-number (g r) a)) ⟶1 / 2) at-top
    unfolding g-def by (rule Re-winding-number-tendsto-part-circlepath)
  moreover have ((λr. Re (winding-number (poly p ∘ g r) 0)) ⟶ degree p
/ 2) at-top
    unfolding g-def by (rule root(1)[OF that])
  moreover have degree q = degree p + 1
    unfolding q-def
    apply (subst degree-mult-eq)
    using that by auto
  ultimately show ?thesis
    by (simp add: tendsto-add add-divide-distrib)
qed
ultimately have ((λr. Re (w r)) ⟶ degree q/2) at-top
```

```
    by (auto dest!:tendsto-cong)
   then show ?thesis unfolding w-def q-def g-def by blast
  qed
  ultimately show ?case by blast
qed

lemma Re-winding-number-poly-linepth:
  fixes pp::complex poly
  defines g ≡ (λr. poly pp o linepath (−r) (of-real r))
  assumes lead-coeff pp=1 and no-real-zero:∀ x∈proots pp. Im x≠0
  shows ((λr. 2∗Re (winding-number (g r) 0) + cindex-pathE (g r) 0 ) ⟶ 0
) at-top
proof −
  define p where p=map-poly Re pp
  define q where q=map-poly Im pp
  define f where f=(λt. poly q t / poly p t)
  have sgnx-top:sgnx (poly p) at-top = 1
   unfolding sgnx-poly-at-top sgn-pos-inf-def p-def using ‹lead-coeff pp=1›
   by (subst lead-coeff-map-poly-nz,auto)
  have not-g-image:0 ∉ path-image (g r) for r
  proof (rule ccontr)
   assume ¬ 0 ∉ path-image (g r)
   then obtain x where poly pp x=0 x∈closed-segment (− of-real r) (of-real r)
    unfolding g-def path-image-compose of-real-linepath by auto
   then have Im x=0 x∈proots pp
    using closed-segment-imp-Re-Im(2) unfolding proots-def by fastforce+
   then show False using ‹∀ x∈proots pp. Im x≠0› by auto
  qed
  have arctan-f-tendsto:((λr. (arctan (f r) − arctan (f (−r))) / pi) ⟶ 0)
at-top
  proof (cases degree p>0)
   case True
   have degree p>degree q
   proof −
    have degree p=degree pp
     unfolding p-def using ‹lead-coeff pp=1›
     by (auto intro:map-poly-degree-eq)
    moreover then have degree q<degree pp
     unfolding q-def using ‹lead-coeff pp=1› True
     by (auto intro!:map-poly-degree-less)
    ultimately show ?thesis by auto
   qed
   then have (f ⟶ 0) at-infinity
    unfolding f-def using poly-divide-tendsto-0-at-infinity by auto
   then have (f ⟶ 0) at-bot (f ⟶ 0) at-top
   by (auto elim!:filterlim-mono simp add:at-top-le-at-infinity at-bot-le-at-infinity)
   then have ((λr. arctan (f r))⟶ 0) at-top ((λr. arctan (f (−r)))⟶ 0)
at-top
    apply −
```

```
    subgoal by (auto intro:tendsto-eq-intros)
    subgoal
      apply (subst tendsto-compose-filtermap[of - uminus,unfolded comp-def])
      by (auto intro:tendsto-eq-intros simp add:at-bot-mirror[symmetric])
    done
  then show ?thesis
    by (auto intro:tendsto-eq-intros)
next
  case False
  obtain c where f=(λr. c)
  proof −
    have degree p=0 using False by auto
    moreover have degree q≤degree p
    proof −
      have degree p=degree pp
        unfolding p-def using ‹lead-coeff pp=1›
        by (auto intro:map-poly-degree-eq)
      moreover have degree q≤degree pp
        unfolding q-def by simp
      ultimately show ?thesis by auto
    qed
    ultimately have degree q=0 by simp
    then obtain pa qa where p=[:pa:] q=[:qa:]
      using ‹degree p=0› by (meson degree-eq-zeroE)
    then show ?thesis using that unfolding f-def by auto
  qed
  then show ?thesis by auto
qed
have [simp]:valid-path (g r) path (g r) finite-ReZ-segments (g r) 0 for r
proof −
  show valid-path (g r) unfolding g-def
    apply (rule valid-path-compose-holomorphic[where S=UNIV])
    by (auto simp add:of-real-linepath)
  then show path (g r) using valid-path-imp-path by auto
  show finite-ReZ-segments (g r) 0
    unfolding g-def of-real-linepath using finite-ReZ-segments-poly-linepath by
simp
qed
have g-f-eq:Im (g r t) / Re (g r t) = (f o (λx. 2∗r∗x − r)) t for r t
proof −
  have Im (g r t) / Re (g r t) = Im ((poly pp o of-real o (λx. 2∗r∗x − r)) t)
      / Re ((poly pp o of-real o (λx. 2∗r∗x − r)) t)
    unfolding g-def linepath-def comp-def
    by (auto simp add:algebra-simps)
  also have ... = (f o (λx. 2∗r∗x − r)) t
    unfolding comp-def
    by (simp only:Im-poly-of-real diff-0-right Re-poly-of-real f-def q-def p-def)
  finally show ?thesis .
qed
```

```
  have ?thesis when proots p={}
  proof −
    have ∀ᶠr in at-top. 2 * Re (winding-number (g r) 0) + cindex-pathE (g r) 0
        = (arctan (f r) −  arctan (f (−r))) / pi
    proof (rule eventually-at-top-linorderI[of 1])
      fix r::real assume r≥1
      have image-pos:∀ p∈path-image (g r).  0<Re p
      proof (rule ccontr)
        assume ¬ (∀ p∈path-image (g r). 0 < Re p)
        then obtain t where poly p t≤0
          unfolding g-def path-image-compose of-real-linepath p-def
          using Re-poly-of-real
          apply (simp add:closed-segment-def)
          by (metis not-less of-real-def real-vector.scale-scale scaleR-left-diff-distrib)

        moreover have False when poly p t<0
        proof −
          have sgnx (poly p) (at-right t) = −1
            using sgnx-poly-nz that by auto
          then obtain x where x>t poly p x = 0
            using sgnx-at-top-IVT[of p t] sgnx-top by auto
          then have x∈proots p unfolding proots-def by auto
          then show False using ‹proots p={}› by auto
        qed
        moreover have False when poly p t=0
          using ‹proots p={}› that unfolding proots-def by auto
        ultimately show False by linarith
      qed
      have Re (winding-number (g r) 0) = (Im (Ln (pathfinish (g r))) − Im (Ln
(pathstart (g r))))
          / (2 * pi)
        apply (rule Re-winding-number-half-right[of g r 0,simplified])
        subgoal using image-pos by auto
        subgoal by (auto simp add:not-g-image)
        done
      also have ... = (arctan (f r) − arctan (f (−r)))/(2*pi)
      proof −
        have Im (Ln (pathfinish (g r))) = arctan (f r)
        proof −
          have Re (pathfinish (g r)) > 0
            by (auto intro: image-pos[rule-format])
          then have Im (Ln (pathfinish (g r)))
              = arctan (Im (pathfinish (g r)) / Re (pathfinish (g r)))
            by (subst Im-Ln-eq,auto)
          also have ... = arctan (f r)
            unfolding path-defs by (subst g-f-eq,auto)
          finally show ?thesis .
        qed
```

```
      moreover have Im (Ln (pathstart (g r))) = arctan (f (−r))
      proof −
        have Re (pathstart (g r)) > 0
          by (auto intro: image-pos[rule-format])
        then have Im (Ln (pathstart (g r)))
            = arctan (Im (pathstart (g r)) / Re (pathstart (g r)))
          by (subst Im-Ln-eq,auto)
        also have ... = arctan (f (−r))
          unfolding path-defs by (subst g-f-eq,auto)
        finally show ?thesis .
      qed
      ultimately show ?thesis by auto
    qed
      finally have Re (winding-number (g r) 0) = (arctan (f r) − arctan (f
(−r)))/(2∗pi) .
    moreover have cindex-pathE (g r) 0 = 0
    proof −
      have cindex-pathE (g r) 0 = cindex-pathE (poly pp o of-real o (λx. 2∗r∗x
− r)) 0
        unfolding g-def linepath-def comp-def
        by (auto simp add:algebra-simps)
      also have ... = cindexE 0 1 (f o (λx. 2∗r∗x − r))
        unfolding cindex-pathE-def comp-def
        by (simp only:Im-poly-of-real diff-0-right Re-poly-of-real f-def q-def p-def)
      also have ... = cindexE (−r) r f
        apply (subst cindexE-linear-comp[of 2∗r 0 1 - −r,simplified])
        using ‹r≥1› by auto
      also have ... = 0
      proof −
        have jumpF f (at-left x) =0 jumpF f (at-right x) = 0 when x∈{−r..r}
for x
        proof −
          have poly p x≠0 using ‹proots p={}› unfolding proots-def by auto
          then show jumpF f (at-left x) =0 jumpF f (at-right x) = 0
            unfolding f-def by (auto intro!: jumpF-not-infinity continuous-intros)
        qed
        then show ?thesis unfolding cindexE-def by auto
      qed
      finally show ?thesis .
    qed
    ultimately show 2 ∗ Re (winding-number (g r) 0) + cindex-pathE (g r) 0
      = (arctan (f r) − arctan (f (−r))) / pi
      unfolding path-defs by (auto simp add:field-simps)
  qed
  with arctan-f-tendsto show ?thesis by (auto dest:tendsto-cong)
qed
moreover have ?thesis when proots p≠{}
proof −
  define max-r where max-r=Max (proots p)
```

define *min-r* **where** *min-r=Min* (*proots p*)
have *max-r* ∈*proots p min-r* ∈*proots p min-r*≤*max-r* **and**
min-max-bound:∀ *p*∈*proots p*. *p*∈{*min-r*..*max-r*}
proof −
have *p*≠*0*
proof −
have (*0*::*real*) ≠ *1*
by *simp*
then show *?thesis*
by (*metis* (*full-types*) ‹*p* ≡ *map-poly Re pp*› *assms*(*2*) *coeff-0 coeff-map-poly one-complex.simps*(*1*) *zero-complex.sel*(*1*))
qed
then have *finite* (*proots p*) **by** *auto*
then show *max-r* ∈*proots p min-r* ∈*proots p*
using *Min-in Max-in that* **unfolding** *max-r-def min-r-def* **by** *fast*+
then show ∀ *p*∈*proots p*. *p*∈{*min-r*..*max-r*}
using *Min-le Max-ge* ‹*finite* (*proots p*)› **unfolding** *max-r-def min-r-def* **by** *auto*
then show *min-r*≤*max-r* **using** ‹*max-r*∈*proots p*› **by** *auto*
qed
have ∀ *F r in at-top*. *2* ∗ *Re* (*winding-number* (*g r*) *0*) + *cindex-pathE* (*g r*) *0*
= (*arctan* (*f r*) − *arctan* (*f* (−*r*))) / *pi*
proof (*rule eventually-at-top-linorderI*[*of max* (*norm max-r*) (*norm min-r*) + *1*])
fix *r* **assume** *r-asm*:*max* (*norm max-r*) (*norm min-r*) + *1* ≤ *r*
then have *r*≠*0 min-r*>−*r max-r*<*r* **by** *auto*
define *u* **where** *u*=(*min-r* + *r*)/(*2*∗*r*)
define *v* **where** *v*=(*max-r* + *r*)/(*2*∗*r*)
have *uv*:*u*∈{*0..1*} *v*∈{*0..1*} *u*≤*v*
unfolding *u-def v-def* **using** *r-asm* ‹*min-r*≤*max-r*›
by (*auto simp add*:*field-simps*)
define *g1* **where** *g1*=*subpath 0 u* (*g r*)
define *g2* **where** *g2*=*subpath u v* (*g r*)
define *g3* **where** *g3*=*subpath v 1* (*g r*)
have *path g1 path g2 path g3 valid-path g1 valid-path g2 valid-path g3*
unfolding *g1-def g2-def g3-def* **using** *uv*
by (*auto intro*!:*path-subpath valid-path-subpath*)
define *wc-add* **where** *wc-add* = (λ*g*. *2*∗*Re* (*winding-number g 0*) + *cindex-pathE g 0*)
have *wc-add* (*g r*) = *wc-add g1* + *wc-add g2* + *wc-add g3*
proof −
have *winding-number* (*g r*) *0* = *winding-number g1 0* + *winding-number g2 0* + *winding-number g3 0*
unfolding *g1-def g2-def g3-def* **using** ‹*u*∈{*0..1*}› ‹*v*∈{*0..1*}› *not-g-image*
by (*subst winding-number-subpath-combine*,*simp-all*)+
moreover have *cindex-pathE* (*g r*) *0* = *cindex-pathE g1 0* + *cindex-pathE g2 0* + *cindex-pathE g3 0*
unfolding *g1-def g2-def g3-def* **using** ‹*u*∈{*0..1*}› ‹*v*∈{*0..1*}› ‹*u*≤*v*› *not-g-image*

```
      by (subst cindex-pathE-subpath-combine,simp-all)+
    ultimately show ?thesis unfolding wc-add-def by auto
  qed
  moreover have wc-add g2=0
  proof −
    have 2 ∗ Re (winding-number g2 0) = − cindex-pathE g2 0
      unfolding g2-def
      apply (rule winding-number-cindex-pathE-aux)
      subgoal using uv by (auto intro:finite-ReZ-segments-subpath)
      subgoal using uv by (auto intro:valid-path-subpath)
     subgoal using Path-Connected.path-image-subpath-subset ‹⋀r. path (g r)›
not-g-image uv
        by blast
    subgoal unfolding subpath-def v-def g-def linepath-def using r-asm ‹max-r
∈proots p›
        by (auto simp add:field-simps Re-poly-of-real p-def)
    subgoal unfolding subpath-def u-def g-def linepath-def using r-asm ‹min-r
∈proots p›
        by (auto simp add:field-simps Re-poly-of-real p-def)
      done
    then show ?thesis unfolding wc-add-def by auto
  qed
  moreover have wc-add g1=− arctan (f (−r)) / pi
  proof −
    have g1-pq:
      Re (g1 t) = poly p (min-r∗t+r∗t−r)
      Im (g1 t) = poly q (min-r∗t+r∗t−r)
      Im (g1 t)/Re (g1 t) = (f o (λx. (min-r+r)∗x − r)) t
      for t
    proof −
      have g1 t = poly pp (of-real (min-r∗t+r∗t−r))
       using ‹r≠0› unfolding g1-def g-def linepath-def subpath-def u-def p-def
        by (auto simp add:field-simps)
      then show
          Re (g1 t) = poly p (min-r∗t+r∗t−r)
          Im (g1 t) = poly q (min-r∗t+r∗t−r)
        unfolding p-def q-def
        by (simp only:Re-poly-of-real Im-poly-of-real)+
      then show Im (g1 t)/Re (g1 t) = (f o (λx. (min-r+r)∗x − r)) t
        unfolding f-def by (auto simp add:algebra-simps)
    qed
    have Re(g1 1)=0
      using ‹r≠0› Re-poly-of-real ‹min-r∈proots p›
      unfolding g1-def subpath-def u-def g-def linepath-def
      by (auto simp add:field-simps p-def)
    have 0 ∉ path-image g1
      by (metis (full-types) path-image-subpath-subset ‹⋀r. path (g r)›
        atLeastAtMost-iff g1-def le-less not-g-image subsetCE uv(1) zero-le-one)
```

```
      have wc-add-pos:wc-add h = − arctan (poly q (− r) / poly p (−r)) / pi
when
      Re-pos:∀ x∈{0..<1}. 0 < (Re ∘ h) x
      and hp:∀ t. Re (h t) = c∗poly p (min-r∗t+r∗t−r)
      and hq:∀ t. Im (h t) = c∗poly q (min-r∗t+r∗t−r)
      and [simp]:c≠0

      and Re (h 1) = 0
      and valid-path h
      and h-img:0 ∉ path-image h
      for h c
    proof −
      define f where f=(λt. c∗poly q t / (c∗poly p t))
      define farg where farg= (if 0 < Im (h 1) then pi / 2 else − pi / 2)
      have Re (winding-number h 0) = (Im (Ln (pathfinish h))
        − Im (Ln (pathstart h))) / (2 ∗ pi)
        apply (rule Re-winding-number-half-right[of h 0,simplified])
        subgoal using that ‹Re (h 1) = 0› unfolding path-image-def
          by (auto simp add:le-less)
        subgoal using ‹valid-path h› .
        subgoal using h-img .
        done
      also have ... = (farg − arctan (f (−r))) / (2 ∗ pi)
      proof −
        have Im (Ln (pathfinish h)) = farg
          using ‹Re(h 1)=0› unfolding farg-def path-defs
          apply (subst Im-Ln-eq)
          subgoal using h-img unfolding path-defs by fastforce
          subgoal by simp
          done
        moreover have Im (Ln (pathstart h)) = arctan (f (−r))
        proof −
          have pathstart h ≠ 0
            using h-img
            by (metis pathstart-in-path-image)
            then have Im (Ln (pathstart h)) = arctan (Im (pathstart h) / Re
(pathstart h))
            using Re-pos[rule-format,of 0]
            by (simp add: Im-Ln-eq path-defs)
          also have ... = arctan (f (−r))
            unfolding f-def path-defs hp[rule-format] hq[rule-format]
            by simp
          finally show ?thesis .
        qed
        ultimately show ?thesis by auto
      qed
      finally have Re (winding-number h 0) = (farg − arctan (f (−r))) / (2 ∗
pi) .
      moreover have cindex-pathE h 0 = (−farg/pi)
```

```
      proof −
        have cindex-pathE h 0 = cindexE 0 1 (f ∘ (λx. (min-r + r) ∗ x − r))
          unfolding cindex-pathE-def using ‹c≠0›
          by (auto simp add:hp hq f-def comp-def algebra-simps)
        also have ... = cindexE (−r) min-r f
          apply (subst cindexE-linear-comp[where b=−r,simplified])
          using r-asm by auto
        also have ... = − jumpF f (at-left min-r)
        proof −
          define right where right = {x. jumpF f (at-right x) ≠ 0 ∧ − r ≤ x
∧ x < min-r}
          define left where left = {x. jumpF f (at-left x) ≠ 0 ∧ − r < x ∧ x
≤ min-r}
            have ∗:jumpF f (at-right x) =0 jumpF f (at-left x) =0 when
x∈{−r..<min-r} for x
          proof −
            have False when poly p x =0
            proof −
              have x≥min-r
                using min-max-bound[rule-format,of x] that by auto
              then show False using ‹x∈{−r..<min-r}› by auto
            qed
            then show jumpF f (at-right x) =0 jumpF f (at-left x) =0
           unfolding f-def by (auto intro!:jumpF-not-infinity continuous-intros)

          qed
          then have right = {}
            unfolding right-def by force
           moreover have left = (if jumpF f (at-left min-r) = 0 then {} else
{min-r})
            unfolding left-def le-less using ∗ r-asm by force
          ultimately show ?thesis
            unfolding cindexE-def by (fold left-def right-def,auto)
        qed
        also have ... = (−farg/pi)
        proof −
          have p-pos:c∗poly p x > 0 when x ∈ {− r<..<min-r} for x
          proof −
            define hh where hh=(λt. min-r∗t+r∗t−r)
            have (x+r)/(min-r+r) ∈ {0..<1}
              using that r-asm by (auto simp add:field-simps)
            then have 0 < c∗poly p (hh ((x+r)/(min-r+r)))
              apply (drule-tac Re-pos[rule-format])
              unfolding comp-def hp[rule-format] hq[rule-format] hh-def .
            moreover have hh ((x+r)/(min-r+r)) = x
              unfolding hh-def using ‹min-r>−r›
              apply (auto simp add:divide-simps)
              by (auto simp add:algebra-simps)
            ultimately show ?thesis by simp
```

```
      qed

      have c∗poly q min-r ≠0
        using no-real-zero ‹c≠0›
      by (metis Im-complex-of-real UNIV-I ‹min-r ∈ proots p› cpoly-of-decompose

          mult-eq-0-iff p-def poly-cpoly-of-real-iff proots-within-iff q-def)

      moreover have ?thesis when c∗poly q min-r > 0
      proof −
       have 0 < Im (h 1) unfolding hq[rule-format] hp[rule-format] using
that by auto
        moreover have jumpF f (at-left min-r) = 1/2
        proof −
          have ((λt. c∗poly p t) has-sgnx 1) (at-left min-r)
            unfolding has-sgnx-def
            apply (rule eventually-at-leftI[of −r])
            using p-pos ‹min-r>−r› by auto
          then have filterlim f at-top (at-left min-r)
            unfolding f-def
            apply (subst filterlim-divide-at-bot-at-top-iff[of - c∗poly q min-r])
            using that ‹min-r∈proots p› by (auto intro!:tendsto-eq-intros)
          then show ?thesis unfolding jumpF-def by auto
        qed
        ultimately show ?thesis unfolding farg-def by auto
      qed
      moreover have ?thesis when c∗poly q min-r < 0
      proof −
       have 0 > Im (h 1) unfolding hq[rule-format] hp[rule-format] using
that by auto
        moreover have jumpF f (at-left min-r) = − 1/2
        proof −
          have ((λt. c∗poly p t) has-sgnx 1) (at-left min-r)
            unfolding has-sgnx-def
            apply (rule eventually-at-leftI[of −r])
            using p-pos ‹min-r>−r› by auto
          then have filterlim f at-bot (at-left min-r)
            unfolding f-def
            apply (subst filterlim-divide-at-bot-at-top-iff[of - c∗poly q min-r])
            using that ‹min-r∈proots p› by (auto intro!:tendsto-eq-intros)
          then show ?thesis unfolding jumpF-def by auto
        qed
        ultimately show ?thesis unfolding farg-def by auto
      qed
      ultimately show ?thesis by linarith
    qed
    finally show ?thesis .
  qed
    ultimately show ?thesis unfolding wc-add-def f-def by (auto simp
```

```
add:field-simps)
      qed

      have ∀ x∈{0..<1}. (Re ∘ g1) x ≠ 0
      proof (rule ccontr)
        assume ¬ (∀ x∈{0..<1}. (Re ∘ g1) x ≠ 0)
        then obtain t where t-def:Re (g1 t) =0 t∈{0..<1}
          unfolding path-image-def by fastforce
        define m where m=min-r∗t+r∗t−r
        have poly p m=0
        proof −
          have Re (g1 t) = Re (poly pp (of-real m))
          unfolding m-def g1-def g-def linepath-def subpath-def u-def using ‹r≠0›
            by (auto simp add:field-simps)
         then show ?thesis using t-def unfolding Re-poly-of-real p-def by auto

        qed
        moreover have m<min-r
        proof −
          have min-r+r>0 using r-asm by simp
          then have (min-r + r)∗(t−1)<0 using ‹t∈{0..<1}›
            by (simp add: mult-pos-neg)
          then show ?thesis unfolding m-def by (auto simp add:algebra-simps)
        qed
        ultimately show False using min-max-bound unfolding proots-def by
auto
      qed
      then have (∀ x∈{0..<1}. 0 < (Re ∘ g1) x) ∨ (∀ x∈{0..<1}. (Re ∘ g1) x
< 0)
        apply (elim continuous-on-neq-split)
        using ‹path g1› unfolding path-def
        by (auto intro!:continuous-intros elim:continuous-on-subset)
      moreover have ?thesis when ∀ x∈{0..<1}. (Re ∘ g1) x < 0
      proof −
        have wc-add (uminus o g1) = − arctan (f (− r)) / pi
          unfolding f-def
          apply (rule wc-add-pos[of - −1])
          using g1-pq that ‹min-r ∈proots p› ‹valid-path g1› ‹0 ∉ path-image g1›
          by (auto simp add:path-image-compose)
        moreover have wc-add (uminus ∘ g1) = wc-add g1
          unfolding wc-add-def cindex-pathE-def
          apply (subst winding-number-uminus-comp)
          using ‹valid-path g1› ‹0 ∉ path-image g1› by auto
        ultimately show ?thesis by auto
      qed
      moreover have ?thesis when ∀ x∈{0..<1}. (Re ∘ g1) x > 0
        unfolding f-def
        apply (rule wc-add-pos[of - 1])
        using g1-pq that ‹min-r ∈proots p› ‹valid-path g1› ‹0 ∉ path-image g1›
```

```
      by (auto simp add:path-image-compose)
    ultimately show ?thesis by blast
  qed
  moreover have wc-add g3 = arctan (f r) / pi
  proof −
    have g3-pq:
      Re (g3 t) = poly p ((r−max-r)∗t + max-r)
      Im (g3 t) = poly q ((r−max-r)∗t + max-r)
      Im (g3 t)/Re (g3 t) = (f o (λx. (r−max-r)∗x + max-r)) t
      for t
    proof −
      have g3 t = poly pp (of-real ((r−max-r)∗t + max-r))
        using ‹r≠0› ‹max-r<r› unfolding g3-def g-def linepath-def subpath-def
v-def p-def
        by (auto simp add:algebra-simps)
      then show
          Re (g3 t) = poly p ((r−max-r)∗t + max-r)
          Im (g3 t) = poly q ((r−max-r)∗t + max-r)
        unfolding p-def q-def
        by (simp only:Re-poly-of-real Im-poly-of-real)+
      then show Im (g3 t)/Re (g3 t) = (f o (λx. (r−max-r)∗x + max-r)) t
        unfolding f-def by (auto simp add:algebra-simps)
    qed
    have Re(g3 0)=0
      using ‹r≠0› Re-poly-of-real ‹max-r∈proots p›
      unfolding g3-def subpath-def v-def g-def linepath-def
      by (auto simp add:field-simps p-def)
    have 0 ∉ path-image g3
    proof −
      have (1::real) ∈ {0..1}
        by auto
      then show ?thesis
        using Path-Connected.path-image-subpath-subset ‹⋀r. path (g r)› g3-def
not-g-image uv(2) by blast
    qed

    have wc-add-pos:wc-add h = arctan (poly q r / poly p r) / pi when
      Re-pos:∀ x∈{0<..1}. 0 < (Re ∘ h) x
      and hp:∀ t. Re (h t) = c∗poly p ((r−max-r)∗t + max-r)
      and hq:∀ t. Im (h t) = c∗poly q ((r−max-r)∗t + max-r)
      and [simp]:c≠0

      and Re (h 0) = 0
      and valid-path h
      and h-img:0 ∉ path-image h
      for h c
    proof −
      define f where f=(λt. c∗poly q t / (c∗poly p t))
      define farg where farg= (if 0 < Im (h 0) then pi / 2 else − pi / 2)
```

```
        have Re (winding-number h 0) = (Im (Ln (pathfinish h))
          − Im (Ln (pathstart h))) / (2 ∗ pi)
         apply (rule Re-winding-number-half-right[of h 0,simplified])
         subgoal using that ‹Re (h 0) = 0› unfolding path-image-def
          by (auto simp add:le-less)
         subgoal using ‹valid-path h› .
         subgoal using h-img .
         done
        also have ... = (arctan (f r) − farg) / (2 ∗ pi)
        proof −
         have Im (Ln (pathstart h)) = farg
          using ‹Re(h 0)=0› unfolding farg-def path-defs
          apply (subst Im-Ln-eq)
          subgoal using h-img unfolding path-defs by fastforce
          subgoal by simp
          done
         moreover have Im (Ln (pathfinish h)) = arctan (f r)
         proof −
          have pathfinish h ≠ 0
           using h-img
           by (metis pathfinish-in-path-image)
           then have Im (Ln (pathfinish h)) = arctan (Im (pathfinish h) / Re
(pathfinish h))
           using Re-pos[rule-format,of 1]
           by (simp add: Im-Ln-eq path-defs)
          also have ... = arctan (f r)
           unfolding f-def path-defs hp[rule-format] hq[rule-format]
           by simp
          finally show ?thesis .
         qed
         ultimately show ?thesis by auto
        qed
       finally have Re (winding-number h 0) = (arctan (f r) − farg) / (2 ∗ pi) .
        moreover have cindex-pathE h 0 = farg/pi
        proof −
         have cindex-pathE h 0 = cindexE 0 1 (f ∘ (λx. (r−max-r)∗x + max-r))
          unfolding cindex-pathE-def using ‹c≠0›
          by (auto simp add:hp hq f-def comp-def algebra-simps)
         also have ... = cindexE max-r r f
          apply (subst cindexE-linear-comp)
          using r-asm by auto
         also have ... = jumpF f (at-right max-r)
         proof −
          define right where right = {x. jumpF f (at-right x) ≠ 0 ∧ max-r ≤
x ∧ x < r}
          define left where left = {x. jumpF f (at-left x) ≠ 0 ∧ max-r < x ∧
x ≤ r}
              have ∗:jumpF f (at-right x) =0 jumpF f (at-left x) =0 when
x∈{max-r<..r} for x
```

```
      proof −
        have False when poly p x =0
        proof −
          have x≤max-r
            using min-max-bound[rule-format,of x] that by auto
          then show False using ‹x∈{max-r<..r}› by auto
        qed
        then show jumpF f (at-right x) =0 jumpF f (at-left x) =0
       unfolding f-def by (auto intro!:jumpF-not-infinity continuous-intros)

      qed
      then have left = {}
        unfolding left-def by force
     moreover have right = (if jumpF f (at-right max-r) = 0 then {} else
{max-r})
        unfolding right-def le-less using ∗ r-asm by force
      ultimately show ?thesis
        unfolding cindexE-def by (fold left-def right-def,auto)
    qed
    also have ... = farg/pi
    proof −
      have p-pos:c∗poly p x > 0 when x ∈ {max-r<..<r} for x
      proof −
        define hh where hh=(λt. (r−max-r)∗t + max-r)
        have (x−max-r)/(r−max-r) ∈ {0<..1}
          using that r-asm by (auto simp add:field-simps)
        then have 0 < c∗poly p (hh ((x−max-r)/(r−max-r)))
          apply (drule-tac Re-pos[rule-format])
          unfolding comp-def hp[rule-format] hq[rule-format] hh-def .
        moreover have hh ((x−max-r)/(r−max-r)) = x
          unfolding hh-def using ‹max-r<r›
          by (auto simp add:divide-simps)
        ultimately show ?thesis by simp
      qed

      have c∗poly q max-r ≠0
        using no-real-zero ‹c≠0›
    by (metis Im-complex-of-real UNIV-I ‹max-r ∈ proots p› cpoly-of-decompose

          mult-eq-0-iff p-def poly-cpoly-of-real-iff proots-within-iff q-def)

      moreover have ?thesis when c∗poly q max-r > 0
      proof −
       have 0 < Im (h 0) unfolding hq[rule-format] hp[rule-format] using
that by auto
        moreover have jumpF f (at-right max-r) = 1/2
        proof −
          have ((λt. c∗poly p t) has-sgnx 1) (at-right max-r)
            unfolding has-sgnx-def
```

```
            apply (rule eventually-at-rightI[of - r])
            using p-pos ‹max-r<r› by auto
          then have filterlim f at-top (at-right max-r)
            unfolding f-def
            apply (subst filterlim-divide-at-bot-at-top-iff[of - c*poly q max-r])
            using that ‹max-r∈proots p› by (auto intro!:tendsto-eq-intros)
          then show ?thesis unfolding jumpF-def by auto
        qed
        ultimately show ?thesis unfolding farg-def by auto
      qed
      moreover have ?thesis when c*poly q max-r < 0
      proof −
       have 0 > Im (h 0) unfolding hq[rule-format] hp[rule-format] using
that by auto
        moreover have jumpF f (at-right max-r) = − 1/2
        proof −
          have ((λt. c*poly p t) has-sgnx 1) (at-right max-r)
            unfolding has-sgnx-def
            apply (rule eventually-at-rightI[of - r])
            using p-pos ‹max-r<r› by auto
          then have filterlim f at-bot (at-right max-r)
            unfolding f-def
            apply (subst filterlim-divide-at-bot-at-top-iff[of - c*poly q max-r])
            using that ‹max-r∈proots p› by (auto intro!:tendsto-eq-intros)
          then show ?thesis unfolding jumpF-def by auto
        qed
        ultimately show ?thesis unfolding farg-def by auto
      qed
      ultimately show ?thesis by linarith
    qed
    finally show ?thesis .
  qed
    ultimately show ?thesis unfolding wc-add-def f-def by (auto simp
add:field-simps)
  qed

  have ∀ x∈{0<..1}. (Re ∘ g3) x ≠ 0
  proof (rule ccontr)
    assume ¬ (∀ x∈{0<..1}. (Re ∘ g3) x ≠ 0)
    then obtain t where t-def:Re (g3 t) =0 t∈{0<..1}
      unfolding path-image-def by fastforce
    define m where m=(r−max-r)*t + max-r
    have poly p m=0
    proof −
      have Re (g3 t) = Re (poly pp (of-real m))
      unfolding m-def g3-def g-def linepath-def subpath-def v-def using ‹r≠0›
        by (auto simp add:algebra-simps)
     then show ?thesis using t-def unfolding Re-poly-of-real p-def by auto
```

```
      qed
      moreover have m>max-r
      proof −
        have r−max-r>0 using r-asm by simp
        then have (r − max-r)∗t>0 using ‹t∈{0<..1}›
          by (simp add: mult-pos-neg)
        then show ?thesis unfolding m-def by (auto simp add:algebra-simps)
      qed
      ultimately show False using min-max-bound unfolding proots-def by
auto
    qed
    then have (∀ x∈{0<..1}. 0 < (Re ∘ g3) x) ∨ (∀ x∈{0<..1}. (Re ∘ g3) x
< 0)
      apply (elim continuous-on-neq-split)
      using ‹path g3› unfolding path-def
      by (auto intro!:continuous-intros elim:continuous-on-subset)
    moreover have ?thesis when ∀ x∈{0<..1}. (Re ∘ g3) x < 0
    proof −
      have wc-add (uminus o g3) = arctan (f r) / pi
        unfolding f-def
        apply (rule wc-add-pos[of - −1])
       using g3-pq that ‹max-r ∈proots p› ‹valid-path g3› ‹0 ∉ path-image g3›
        by (auto simp add:path-image-compose)
      moreover have wc-add (uminus ∘ g3) = wc-add g3
        unfolding wc-add-def cindex-pathE-def
        apply (subst winding-number-uminus-comp)
        using ‹valid-path g3› ‹0 ∉ path-image g3› by auto
      ultimately show ?thesis by auto
    qed
    moreover have ?thesis when ∀ x∈{0<..1}. (Re ∘ g3) x > 0
      unfolding f-def
      apply (rule wc-add-pos[of - 1])
      using g3-pq that ‹max-r ∈proots p› ‹valid-path g3› ‹0 ∉ path-image g3›
      by (auto simp add:path-image-compose)
    ultimately show ?thesis by blast
  qed
  ultimately have wc-add (g r) = (arctan (f r) − arctan (f (−r))) / pi
    by (auto simp add:field-simps)
  then show 2 ∗ Re (winding-number (g r) 0) + cindex-pathE (g r) 0
      = (arctan (f r) − arctan (f (− r))) / pi
    unfolding wc-add-def .
 qed
 with arctan-f-tendsto show ?thesis by (auto dest:tendsto-cong)
qed
ultimately show ?thesis by auto
qed

lemma proots-upper-cindex-eq:
 assumes lead-coeff p=1 and no-real-roots:∀ x∈proots p. Im x≠0
```

shows *proots-upper p =*
(degree p − cindex-poly-ubd (map-poly Im p) (map-poly Re p)) /2
proof *(cases degree p = 0)*
case *True*
then obtain *c* **where** *p=[:c:]* **using** *degree-eq-zeroE* **by** *blast*
then have *p-def:p=[:1:]* **using** ‹*lead-coeff p=1*› **by** *simp*
have *proots-count p {x. Im x>0} = 0* **unfolding** *p-def proots-count-def* **by** *auto*

moreover have *cindex-poly-ubd (map-poly Im p) (map-poly Re p) = 0*
apply *(subst cindex-poly-ubd-code)*
unfolding *p-def*
by *(auto simp add:map-poly-pCons changes-R-smods-def changes-poly-neg-inf-def*

changes-poly-pos-inf-def)
ultimately show *?thesis* **using** *True* **unfolding** *proots-upper-def* **by** *auto*
next
case *False*
then have *degree p>0 p≠0* **by** *auto*
define *w1* **where** *w1=(λr. Re (winding-number (poly p ∘*
(λx. complex-of-real (linepath (− r) (of-real r) x))) 0))
define *w2* **where** *w2=(λr. Re (winding-number (poly p ∘ part-circlepath 0 r 0 pi) 0))*
define *cp* **where** *cp=(λr. cindex-pathE (poly p ∘ (λx.*
of-real (linepath (− r) (of-real r) x))) 0)
define *ci* **where** *ci=(λr. cindexE (−r) r (λx. poly (map-poly Im p) x/poly (map-poly Re p) x))*
define *cubd* **where** *cubd =cindex-poly-ubd (map-poly Im p) (map-poly Re p)*
obtain *R* **where** *proots p ⊆ ball 0 R* **and** *R>0*
using ‹*p≠0*› *finite-ball-include*[*of proots p 0*] **by** *auto*
have *((λr. w1 r +w2 r+ cp r / 2 −ci r/2)*
⟶ real (degree p) / 2 − of-int cubd / 2) at-top
proof −
have *t1:((λr. 2 ∗ w1 r + cp r) ⟶ 0) at-top*
using *Re-winding-number-poly-linepth*[*OF assms*] **unfolding** *w1-def cp-def* **by** *auto*
have *t2:(w2 ⟶ real (degree p) / 2) at-top*
using *Re-winding-number-poly-part-circlepath*[*OF* ‹*degree p>0*›,*of 0*] **unfolding** *w2-def* **by** *auto*
have *t3:(ci ⟶ of-int cubd) at-top*
apply *(rule Lim-eventually)*
using *cindex-poly-ubd-eventually*[*of map-poly Im p map-poly Re p*]
unfolding *ci-def cubd-def* **by** *auto*
from *tendsto-add*[*OF tendsto-add*[*OF tendsto-mult-left*[*OF t3*,*of −1/2*,*simplified*]

tendsto-mult-left[*OF t1*,*of 1/2*,*simplified*]]
t2]
show *?thesis* **by** *(simp add:algebra-simps)*
qed
moreover have *∀ F r in at-top. w1 r +w2 r+ cp r / 2 −ci r/2 = proots-count*

```
p {x. Im x>0}
  proof (rule eventually-at-top-linorderI[of R])
    fix r assume r≥R
    then have r-ball:proots p ⊆ ball 0 r and r>0
      using ‹R>0› ‹proots p ⊆ ball 0 R› by auto
    define ll where ll=linepath (− complex-of-real r) r
    define rr where rr=part-circlepath 0 r 0 pi
    define lr where lr = ll +++ rr
    have img-ll:path-image ll ⊆ − proots p and img-rr: path-image rr ⊆ − proots
p
      subgoal unfolding ll-def using ‹0 < r› closed-segment-degen-complex(2)
no-real-roots by auto
     subgoal unfolding rr-def using in-path-image-part-circlepath ‹0 < r› r-ball
by fastforce
      done
    have [simp]:valid-path (poly p o ll) valid-path (poly p o rr)
       valid-path ll valid-path rr
       pathfinish rr=pathstart ll pathfinish ll = pathstart rr
    proof −
      show valid-path (poly p o ll) valid-path (poly p o rr)
        unfolding ll-def rr-def by (auto intro:valid-path-compose-holomorphic)
      then show valid-path ll valid-path rr unfolding ll-def rr-def by auto
      show pathfinish rr=pathstart ll pathfinish ll = pathstart rr
        unfolding ll-def rr-def by auto
    qed
     have proots-count p {x. Im x>0} = (∑ x∈proots p. winding-number lr x ∗
of-nat (order x p))
    unfolding proots-count-def of-nat-sum
    proof (rule sum.mono-neutral-cong-left)
      show finite (proots p) proots-within p {x. 0 < Im x} ⊆ proots p
        using ‹p≠0› by auto
    next
      have winding-number lr x=0 when x∈proots p − proots-within p {x. 0 <
Im x} for x
      unfolding lr-def ll-def rr-def
      proof (eval-winding,simp-all)
        show ∗:x ∉ closed-segment (− complex-of-real r) (complex-of-real r)
          using img-ll that unfolding ll-def by auto
        show x ∉ path-image (part-circlepath 0 r 0 pi)
          using img-rr that unfolding rr-def by auto
        have xr-facts:0 > Im x −r<Re x Re x<r cmod x<r
        proof −
          have Im x≤0 using that by auto
          moreover have Im x≠0 using no-real-roots that by blast
          ultimately show 0 > Im x by auto
        next
          show cmod x<r using that r-ball by auto
          then have |Re x| < r
            using abs-Re-le-cmod[of x] by argo
```

```
      then show −r<Re x Re x<r  by linarith+
    qed
    then have cindex-pathE ll x = 1
      using ‹r>0› unfolding cindex-pathE-linepath[OF ∗] ll-def
      by (auto simp add: mult-pos-neg)
    moreover have cindex-pathE rr x=−1
      unfolding rr-def using r-ball that
      by (auto intro!: cindex-pathE-circlepath-upper)
    ultimately show −cindex-pathE (linepath (− of-real r) (of-real r)) x =
        cindex-pathE (part-circlepath 0 r 0 pi) x
      unfolding ll-def rr-def by auto
  qed
  then show ∀ i∈proots p − proots-within p {x. 0 < Im x}.
      winding-number lr i ∗ of-nat (order i p) = 0
    by auto
next
  fix x assume x-asm:x ∈ proots-within p {x. 0 < Im x}
  have winding-number lr x=1 unfolding lr-def ll-def rr-def
  proof (eval-winding,simp-all)
    show ∗:x ∉ closed-segment (− complex-of-real r) (complex-of-real r)
      using img-ll x-asm unfolding ll-def by auto
    show x ∉ path-image (part-circlepath 0 r 0 pi)
      using img-rr x-asm unfolding rr-def by auto
    have xr-facts:0 < Im x −r<Re x Re x<r cmod x<r
    proof −
      show 0 < Im x using x-asm by auto
    next
      show cmod x<r using x-asm r-ball by auto
      then have |Re x| < r
        using abs-Re-le-cmod[of x] by argo
      then show −r<Re x Re x<r  by linarith+
    qed
    then have cindex-pathE ll x = −1
      using ‹r>0› unfolding cindex-pathE-linepath[OF ∗] ll-def
      by (auto simp add: mult-less-0-iff)
    moreover have cindex-pathE rr x=−1
      unfolding rr-def using r-ball x-asm
      by (auto intro!: cindex-pathE-circlepath-upper)
   ultimately show − of-real (cindex-pathE (linepath (− of-real r) (of-real
r)) x) −
        of-real (cindex-pathE (part-circlepath 0 r 0 pi) x) = 2
      unfolding ll-def rr-def by auto
  qed
 then show of-nat (order x p) = winding-number lr x ∗ of-nat (order x p) by
auto
qed
also have ... = 1/(2∗pi∗i) ∗ contour-integral lr (λx. deriv (poly p) x / poly p
x)
 apply (subst argument-principle-poly[of p lr])
```

using ‹$p\neq 0$› *img-ll img-rr* **unfolding** *lr-def ll-def rr-def*
by (*auto simp add*:*path-image-join*)
also have ... = *winding-number* (*poly p* ∘ *lr*) *0*
apply (*subst winding-number-comp*[*of UNIV poly p lr 0*])
using ‹$p\neq 0$› *img-ll img-rr* **unfolding** *lr-def ll-def rr-def*
by (*auto simp add*:*path-image-join path-image-compose*)
also have ... = *Re* (*winding-number* (*poly p* ∘ *lr*) *0*)
proof −
have *winding-number* (*poly p* ∘ *lr*) *0* ∈ *Ints*
apply (*rule integer-winding-number*)
using ‹$p\neq 0$› *img-ll img-rr* **unfolding** *lr-def*
by (*auto simp add*:*path-image-join path-image-compose path-compose-join pathstart-compose pathfinish-compose valid-path-imp-path*)
then show *?thesis* **by** (*simp add*: *complex-eqI complex-is-Int-iff*)
qed
also have ... = *Re* (*winding-number* (*poly p* ∘ *ll*) *0*) + *Re* (*winding-number* (*poly p* ∘ *rr*) *0*)
unfolding *lr-def path-compose-join* **using** *img-ll img-rr*
apply (*subst winding-number-join*)
by (*auto simp add*:*valid-path-imp-path path-image-compose pathstart-compose pathfinish-compose*)
also have ... = *w1 r* +*w2 r*
unfolding *w1-def w2-def ll-def rr-def of-real-linepath* **by** *auto*
finally have *of-nat* (*proots-count p* {*x*. *0* < *Im x*}) = *complex-of-real* (*w1 r* + *w2 r*) **.**
then have *proots-count p* {*x*. *0* < *Im x*} = *w1 r* + *w2 r*
using *of-real-eq-iff* **by** *fastforce*
moreover have *cp r* = *ci r*
proof −
define *f* **where** *f*=(λ*x*. *Im* (*poly p* (*of-real x*)) / *Re* (*poly p x*))
have *cp r* = *cindex-pathE* (*poly p* ∘ (λ*x*. *2*∗*r*∗*x* − *r*)) *0*
unfolding *cp-def linepath-def* **by** (*auto simp add*:*algebra-simps*)
also have ... = *cindexE 0 1* (*f o* (λ*x*. *2*∗*r*∗*x* − *r*))
unfolding *cp-def ci-def cindex-pathE-def f-def comp-def* **by** *auto*
also have ... = *cindexE* (−*r*) *r f*
apply (*subst cindexE-linear-comp*[*of 2*∗*r 0 1 f* −*r*,*simplified*])
using ‹$r>0$› **by** *auto*
also have ... = *ci r*
unfolding *ci-def f-def Im-poly-of-real Re-poly-of-real* **by** *simp*
finally show *?thesis* **.**
qed
ultimately show *w1 r* + *w2 r* + *cp r* / *2* − *ci r* / *2* = *real* (*proots-count p* {*x*. *0* < *Im x*})
by *auto*
qed
ultimately have ((λ*r*::*real*. *real* (*proots-count p* {*x*. *0* < *Im x*}))
⟶ *real* (*degree p*) / *2* − *of-int cubd* / *2*) *at-top*
by (*auto dest*: *tendsto-cong*)
then show *?thesis*

```
    apply (subst (asm) tendsto-const-iff)
    unfolding cubd-def proots-upper-def by auto
qed

lemma cindexE-roots-on-horizontal-border:
  fixes a::complex and s::real
  defines g≡linepath a (a + of-real s)
  assumes pqr:p = q * r and r-monic:lead-coeff r=1 and r-proots:∀ x∈proots r.
Im x=Im a
  shows cindexE lb ub (λt. Im ((poly p ∘ g) t) / Re ((poly p ∘ g) t)) =
        cindexE lb ub (λt. Im ((poly q ∘ g) t) / Re ((poly q ∘ g) t))
  using assms
proof (induct r arbitrary:p rule:poly-root-induct-alt)
  case 0
  then have False
    by (metis Im-complex-of-real UNIV-I imaginary-unit.simps(2) proots-within-0
zero-neq-one)
  then show ?case by simp
next
  case (no-proots r)
  then obtain b where b≠0 r=[:b:]
    using fundamental-theorem-of-algebra-alt by blast
  then have r=1 using ‹lead-coeff r = 1› by simp
  with ‹p = q * r› show ?case by simp
next
  case (root b r)
  then have ?case when s=0
    using that(1) unfolding cindex-pathE-def by (simp add:cindexE-constI)
  moreover have ?case when s≠0
  proof −
    define qrg where qrg = poly (q*r) ∘ g
    have cindexE lb ub (λt. Im ((poly p ∘ g) t) / Re ((poly p ∘ g) t))
        = cindexE lb ub (λt. Im (qrg t * (g t − b)) / Re (qrg t * (g t − b)))
      unfolding qrg-def ‹p = q * ([:− b, 1:] * r)› comp-def
      by (simp add:algebra-simps)
    also have ... = cindexE lb ub
       (λt. ((Re a + t * s − Re b )* Im (qrg t)) /
         ((Re a + t * s − Re b )* Re (qrg t)))
    proof −
      have Im b = Im a
        using ‹∀ x∈proots ([:− b, 1:] * r). Im x = Im a› by auto
      then show ?thesis
        unfolding cindex-pathE-def g-def linepath-def
        by (simp add:algebra-simps)
    qed
    also have ... = cindexE lb ub (λt. Im (qrg t) / Re (qrg t))
    proof (rule cindexE-cong[of {t. Re a + t * s − Re b = 0}])
      show finite {t. Re a + t * s − Re b = 0}
      proof (cases Re a= Re b)
```

```
      case True
      then have {t. Re a + t * s - Re b = 0} = {0}
        using ‹s≠0› by auto
      then show ?thesis by auto
    next
      case False
      then have {t. Re a + t * s - Re b = 0} = {(Re b - Re a) / s}
        using ‹s≠0› by (auto simp add:field-simps)
      then show ?thesis by auto
    qed
  next
    fix x assume asm:x ∉ {t. Re a + t * s - Re b = 0}
    define tt where tt=Re a + x * s - Re b
    have tt≠0 using asm unfolding tt-def by auto
    then show tt * Im (qrg x) / (tt * Re (qrg x)) = Im (qrg x) / Re (qrg x)
      by auto
  qed
  also have ... = cindexE lb ub (λt. Im ((poly q ∘ g) t) / Re ((poly q ∘ g) t))
    unfolding qrg-def
  proof (rule root(1))
    show lead-coeff r = 1
    by (metis lead-coeff-mult lead-coeff-pCons(1) mult-cancel-left2 one-poly-eq-simps(2)

        root.prems(2) zero-neq-one)
  qed (use root in simp-all)
  finally show ?thesis .
 qed
 ultimately show ?case by auto
qed



lemma poly-decompose-by-proots:
 fixes p ::'a::idom poly
 assumes p≠0
 shows ∃ q r. p = q * r ∧ lead-coeff q=1 ∧ (∀ x∈proots q. P x) ∧ (∀ x∈proots r.
¬P x) using assms
proof (induct p rule:poly-root-induct-alt)
 case 0
 then show ?case by simp
next
 case (no-proots p)
 then show ?case
   apply (rule-tac x=1 in exI)
   apply (rule-tac x=p in exI)
   by (simp add:proots-def)
next
 case (root a p)
 then obtain q r where pqr:p = q * r and leadq:lead-coeff q=1
```

```
                    and qball:∀ a∈proots q. P a and rball:∀ x∈proots r. ¬ P x
    using mult-zero-right by blast
  have ?case when P a
    apply (rule-tac x=[:− a, 1:] ∗ q in exI)
    apply (rule-tac x=r in exI)
    using pqr qball rball that leadq unfolding lead-coeff-mult
    by (auto simp add:algebra-simps)
  moreover have ?case when ¬ P a
    apply (rule-tac x=q in exI)
    apply (rule-tac x=[:− a, 1:] ∗r in exI)
    using pqr qball rball that leadq unfolding lead-coeff-mult
    by (auto simp add:algebra-simps)
  ultimately show ?case by blast
qed

lemma proots-upper-cindex-eq':
  assumes lead-coeff p=1
  shows proots-upper p = (degree p − proots-count p {x. Im x=0}
            − cindex-poly-ubd (map-poly Im p) (map-poly Re p)) /2
proof −
  have p≠0 using assms by auto
  from poly-decompose-by-proots[OF this,of λx. Im x≠0]
  obtain q r where pqr:p = q ∗ r and leadq:lead-coeff q=1
            and qball: ∀ x∈proots q. Im x ≠0 and rball:∀ x∈proots r. Im x =0
    by auto
  have real-of-int (proots-upper p) = proots-upper q + proots-upper r
   using ‹p≠0› unfolding proots-upper-def pqr by (auto simp add:proots-count-times)
  also have ... = proots-upper q
  proof −
    have proots-within r {z. 0 < Im z} = {}
      using rball by auto
    then have proots-upper r =0
      unfolding proots-upper-def proots-count-def by simp
    then show ?thesis by auto
  qed
  also have ... =  (degree q − cindex-poly-ubd (map-poly Im q) (map-poly Re q))
/ 2
    by (rule proots-upper-cindex-eq[OF leadq qball])
  also have ... = (degree p − proots-count p {x. Im x=0}
                    − cindex-poly-ubd (map-poly Im p) (map-poly Re p)) /2
  proof −
    have degree q = degree p − proots-count p {x. Im x=0}
    proof −
      have degree p= degree q + degree r
        unfolding pqr
        apply (rule degree-mult-eq)
        using ‹p ≠ 0› pqr by auto
      moreover have degree r = proots-count p {x. Im x=0}
        unfolding degree-proots-count proots-count-def
```

```
    proof (rule sum.cong)
      fix x assume x ∈ proots-within p {x. Im x = 0}
      then have Im x=0 by auto
      then have order x q = 0
        using qball order-0I by blast
      then show order x r = order x p
        using ‹p≠0› unfolding pqr by (simp add: order-mult)
    next
      show proots r = proots-within p {x. Im x = 0}
        unfolding pqr proots-within-times using qball rball by auto
    qed
    ultimately show ?thesis by auto
  qed
  moreover have cindex-poly-ubd (map-poly Im q) (map-poly Re q)
        = cindex-poly-ubd (map-poly Im p) (map-poly Re p)
  proof −
    define iq rq ip rp where iq = map-poly Im q and rq=map-poly Re q
                    and ip=map-poly Im p and rp = map-poly Re p
    have cindexE (− x) x (λx. poly iq x / poly rq x)
          = cindexE (− x) x (λx. poly ip x / poly rp x) for x
    proof −
      have lead-coeff r = 1
        using pqr leadq ‹lead-coeff p=1› by (simp add: coeff-degree-mult)
      then have cindexE (− x) x (λt. Im (poly p (t *R 1)) / Re (poly p (t *R
1))) =
                  cindexE (− x) x (λt. Im (poly q (t *R 1)) / Re (poly q (t *R 1)))
        using cindexE-roots-on-horizontal-border[OF pqr,of 0 −x x 1
            ,unfolded linepath-def comp-def,simplified] rball by simp
      then show ?thesis
        unfolding scaleR-conv-of-real iq-def ip-def rq-def rp-def
        by (simp add:Im-poly-of-real Re-poly-of-real)
    qed
    then have ∀F r::real in at-top.
      real-of-int (cindex-poly-ubd iq rq) = cindex-poly-ubd ip rp
      using eventually-conj[OF cindex-poly-ubd-eventually[of iq rq]
            cindex-poly-ubd-eventually[of ip rp]]
      by (elim eventually-mono,auto)
    then show ?thesis
      apply (fold iq-def rq-def ip-def rp-def)
      by simp
  qed
  ultimately show ?thesis by auto
 qed
 finally show ?thesis by simp
qed


lemma proots-within-upper-squarefree:
 assumes rsquarefree p
```

```
  shows  card (proots-within p {x. Im x >0}) = (let
          pp = smult (inverse (lead-coeff p)) p;
          pI = map-poly Im pp;
          pR = map-poly Re pp;
          g = gcd pR pI
        in
          nat ((degree p − changes-R-smods g (pderiv g) − changes-R-smods pR
pI) div 2)
      )
proof −
  define pp where pp = smult (inverse (lead-coeff p)) p
  define pI where pI = map-poly Im pp
  define pR where pR = map-poly Re pp
  define g where  g = gcd pR pI
  have card (proots-within p {x. Im x >0}) = proots-upper p
   unfolding proots-upper-def using card-proots-within-rsquarefree[OF assms] by
auto
  also have ... = proots-upper pp
    unfolding proots-upper-def pp-def
    apply (subst proots-count-smult)
    using assms by auto
  also have ... = (degree pp − proots-count pp {x. Im x = 0} − cindex-poly-ubd
pI pR) div 2
  proof −
    define rr where rr = proots-count pp {x. Im x = 0}
    define cpp where cpp = cindex-poly-ubd pI pR
    have ∗:proots-upper pp = (degree pp − rr − cpp) / 2
     apply (rule proots-upper-cindex-eq'[of pp,folded rr-def cpp-def pR-def pI-def])
      unfolding pp-def using assms by auto
    also have ... = (degree pp − rr − cpp) div 2
    proof (subst real-of-int-div)
      define tt where tt=int (degree pp − rr) − cpp
      have real-of-int tt=2∗proots-upper pp
        by (simp add:∗[folded tt-def])
    then show even tt by (metis dvd-triv-left even-of-nat of-int-eq-iff of-int-of-nat-eq)
    qed simp
    finally show ?thesis unfolding rr-def cpp-def by simp
  qed
  also have ... = (degree pp − changes-R-smods g (pderiv g)
                      − cindex-poly-ubd pI pR) div 2
  proof −
    have rsquarefree pp
      using assms rsquarefree-smult-iff unfolding pp-def
      by (metis inverse-eq-imp-eq inverse-zero leading-coeff-neq-0 rsquarefree-0)
    from card-proots-within-rsquarefree[OF this]
    have proots-count pp {x. Im x = 0} = card (proots-within pp {x. Im x = 0})
      by simp
    also have ... = card (proots-within pp (unbounded-line 0 1))
    proof −
```

```
    have {x. Im x = 0} = unbounded-line 0 1
      unfolding unbounded-line-def
      apply auto
      subgoal for x
        apply (rule-tac x=Re x in exI)
        by (metis complex-is-Real-iff of-real-Re of-real-def)
      done
    then show ?thesis by simp
  qed
  also have ... = changes-R-smods g (pderiv g)
   unfolding card-proots-unbounded-line[of 0 1 pp,simplified,folded pI-def pR-def]
g-def
    by (auto simp add:Let-def sturm-R[symmetric])
  finally have proots-count pp {x. Im x = 0} = changes-R-smods g (pderiv g) .
  moreover have degree pp ≥ proots-count pp {x. Im x = 0}
    by (metis ‹rsquarefree pp› proots-count-leq-degree rsquarefree-0)
  ultimately show ?thesis
    by auto
 qed
 also have ... = (degree p − changes-R-smods g (pderiv g)
                    − changes-R-smods pR pI) div 2
  using cindex-poly-ubd-code unfolding pp-def by simp
 finally have card (proots-within p {x. 0 < Im x}) = (degree p − changes-R-smods
g (pderiv g) −
               changes-R-smods pR pI) div 2 .
 then show ?thesis unfolding Let-def
  apply (fold pp-def pR-def pI-def g-def)
  by (simp add: pp-def)
qed

lemma proots-upper-code1[code]:
 proots-upper p =
  (if p ≠ 0 then
     (let pp=smult (inverse (lead-coeff p)) p;
          pI=map-poly Im pp;
          pR=map-poly Re pp;
          g = gcd pI pR
      in
       nat ((degree p − nat (changes-R-smods-ext g (pderiv g)) − changes-R-smods
pR pI) div 2)
      )
   else
     Code.abort (STR ''proots-upper fails when p=0.'') (λ-. proots-upper p))
proof −
 define pp where pp = smult (inverse (lead-coeff p)) p
 define pI where pI = map-poly Im pp
 define pR where pR=map-poly Re pp
 define g where  g = gcd pI pR
 have ?thesis when p=0
```

```
    using that by auto
  moreover have ?thesis when p≠0
  proof −
    have pp≠0 unfolding pp-def using that by auto
     define rr where rr=int (degree pp − proots-count pp {x. Im x = 0}) −
cindex-poly-ubd pI pR
    have lead-coeff p≠0 using ‹p≠0› by simp
    then have proots-upper pp = rr / 2 unfolding rr-def
      apply (rule-tac proots-upper-cindex-eq'[of pp, folded pI-def pR-def])
      unfolding pp-def lead-coeff-smult by auto
    then have proots-upper pp = nat (rr div 2) by linarith
    moreover have
      rr = degree p − nat (changes-R-smods-ext g (pderiv g)) − changes-R-smods
pR pI
    proof −
      have degree pp = degree p unfolding pp-def by auto
      moreover have proots-count pp {x. Im x = 0} = nat (changes-R-smods-ext
g (pderiv g))
      proof −
        have {x. Im x = 0} = unbounded-line 0 1
          unfolding unbounded-line-def by (simp add: complex-eq-iff)
        then show ?thesis
            using proots-unbounded-line[of 0 1 pp,simplified, folded pI-def pR-def]
‹pp≠0›
          by (auto simp:Let-def g-def gcd.commute)
      qed
      moreover have cindex-poly-ubd pI pR = changes-R-smods pR pI
        using cindex-poly-ubd-code by auto
      ultimately show ?thesis unfolding rr-def by auto
    qed
    moreover have proots-upper pp = proots-upper p
      unfolding pp-def proots-upper-def
      apply (subst proots-count-smult)
      using that by auto
    ultimately show ?thesis
      unfolding Let-def using that
      apply (fold pp-def pI-def pR-def g-def)
      by argo
  qed
  ultimately show ?thesis by blast
qed

lemma proots-upper-card-code[code]:
  proots-upper-card p = (if p=0 then 0 else
      (let
            pf = p div (gcd p (pderiv p));
            pp = smult (inverse (lead-coeff pf)) pf;
            pI = map-poly Im pp;
            pR = map-poly Re pp;
```

```
        g = gcd pR pI
      in
        nat ((degree pf − changes-R-smods g (pderiv g) − changes-R-smods pR
pI) div 2)
    ))
proof (cases p=0)
  case True
 then show ?thesis unfolding proots-upper-card-def using infinite-halfspace-Im-gt
by auto
next
  case False
  define pf pp pI pR g where
      pf = p div (gcd p (pderiv p))
    and pp = smult (inverse (lead-coeff pf)) pf
    and pI = map-poly Im pp
    and pR = map-poly Re pp
    and g = gcd pR pI
  have proots-upper-card p = proots-upper-card pf
  proof −
    have proots-within p {x. 0 < Im x} = proots-within pf {x. 0 < Im x}
      unfolding proots-within-def using poly-gcd-pderiv-iff[of p,folded pf-def]
      by auto
    then show ?thesis unfolding proots-upper-card-def by auto
  qed
 also have ... = nat ((degree pf − changes-R-smods g (pderiv g) − changes-R-smods
pR pI) div 2)
    using proots-within-upper-squarefree[OF rsquarefree-gcd-pderiv[OF ‹p≠0›]
      ,unfolded Let-def,folded pf-def,folded pp-def pI-def pR-def g-def]
    unfolding proots-upper-card-def by blast
  finally show ?thesis unfolding Let-def
    apply (fold pf-def,fold pp-def pI-def pR-def g-def)
    using False by auto
qed
```

3.8 Polynomial roots on a general half-plane

the number of roots of polynomial p, counted with multiplicity, on the left half plane of the vector $b - a$.

```
definition proots-half ::complex poly ⇒ complex ⇒ complex ⇒ nat where
  proots-half p a b = proots-count p {w. Im ((w−a) / (b−a)) > 0}

lemma proots-half-empty:
  assumes a=b
  shows proots-half p a b = 0
unfolding proots-half-def using assms by auto


lemma proots-half-proots-upper:
  assumes a≠b p≠0
```

shows *proots-half p a b= proots-upper (pcompose p [:a, (b−a):])*
proof −
define *q* **where** *q=[:a, (b − a):]*
define *f* **where** *f=(λx. (b−a)∗x+ a)*
have *(∑ r∈proots-within p {w. Im ((w−a) / (b−a)) > 0}. order r p)*
= (∑ r∈proots-within (p ∘p q) {z. 0 < Im z}. order r (p ∘p q))
proof *(rule sum.reindex-cong[of f])*
have *inj f*
using *assms* **unfolding** *f-def inj-on-def* **by** *fastforce*
then show *inj-on f (proots-within (p ∘p q) {z. 0 < Im z})*
by *(elim inj-on-subset,auto)*
next
show *proots-within p {w. Im ((w−a) / (b−a)) > 0} = f ‘ proots-within (p ∘p q) {z. 0 < Im z}*
proof *safe*
fix *x* **assume** *x-asm:x ∈ proots-within p {w. Im ((w−a) / (b−a)) > 0}*
define *xx* **where** *xx=(x −a) / (b − a)*
have *poly (p ∘p q) xx = 0*
unfolding *q-def xx-def poly-pcompose* **using** *assms x-asm* **by** *auto*
moreover have *Im xx > 0*
unfolding *xx-def* **using** *x-asm* **by** *auto*
ultimately have *xx ∈ proots-within (p ∘p q) {z. 0 < Im z}* **by** *auto*
then show *x ∈ f ‘ proots-within (p ∘p q) {z. 0 < Im z}*
apply *(intro rev-image-eqI[of xx])*
unfolding *f-def xx-def* **using** *assms* **by** *auto*
next
fix *x* **assume** *x ∈ proots-within (p ∘p q) {z. 0 < Im z}*
then show *f x ∈ proots-within p {w. 0 < Im ((w−a) / (b − a))}*
unfolding *f-def q-def* **using** *assms*
apply *(auto simp add:poly-pcompose)*
by *(auto simp add:algebra-simps)*
qed
next
fix *x* **assume** *x ∈ proots-within (p ∘p q) {z. 0 < Im z}*
show *order (f x) p = order x (p ∘p q)* **using** ‹*p≠0*›
proof *(induct p rule:poly-root-induct-alt)*
case *0*
then show *?case* **by** *simp*
next
case *(no-proots p)*
have *order (f x) p = 0*
apply *(rule order-0I)*
using *no-proots* **by** *auto*
moreover have *order x (p ∘p q) = 0*
apply *(rule order-0I)*
unfolding *poly-pcompose q-def* **using** *no-proots* **by** *auto*
ultimately show *?case* **by** *auto*
next
case *(root c p)*

```
    have order (f x) ([:− c, 1:] ∗ p) = order (f x) [:−c,1:] + order (f x) p
      apply (subst order-mult)
      using root by auto
    also have ... =  order x ([:− c, 1:] ∘p q) +  order x (p ∘p q)
    proof −
      have order (f x) [:− c, 1:] = order x ([:− c, 1:] ∘p q)
      proof (cases f x=c)
        case True
        have [:− c, 1:] ∘p q = smult (b−a) [:−x,1:]
          using True unfolding q-def f-def pcompose-pCons by auto
        then have order x ([:− c, 1:] ∘p q) = order x (smult (b−a) [:−x,1:])
          by auto
        then have order x ([:− c, 1:] ∘p q) = 1
          apply (subst (asm) order-smult)
          using assms order-power-n-n[of - 1,simplified] by auto
        moreover have order (f x) [:− c, 1:] = 1
          using True order-power-n-n[of - 1,simplified] by auto
        ultimately show ?thesis by auto
      next
        case False
        have order (f x) [:− c, 1:] = 0
          apply (rule order-0I)
          using False unfolding f-def by auto
        moreover have order x ([:− c, 1:] ∘p q) = 0
          apply (rule order-0I)
          using False unfolding f-def q-def poly-pcompose by auto
        ultimately show ?thesis by auto
      qed
      moreover have order (f x) p = order x (p ∘p q)
        apply (rule root)
        using root by auto
      ultimately show ?thesis by auto
    qed
    also have ... = order x ((([:− c, 1:] ∗ p) ∘p q)
      unfolding pcompose-mult
      apply (subst order-mult)
       subgoal unfolding q-def using assms(1) pcompose-eq-0 root.prems by
fastforce
      by simp
    finally show ?case .
  qed
 qed
 then show ?thesis unfolding proots-half-def proots-upper-def proots-count-def
q-def
  by auto
qed

lemma proots-half-code1[code]:
 proots-half p a b = (if a≠b then
```

```
                    if p≠0 then proots-upper (p ∘p [:a, b − a:])
                    else Code.abort (STR ''proots-half fails when p=0.'')
                      (λ-. proots-half p a b)
                    else 0)
proof −
  have ?thesis when a=b
    using proots-half-empty that by auto
  moreover have ?thesis when a≠b p=0
    using that by auto
  moreover have ?thesis when a≠b p≠0
    using proots-half-proots-upper[OF that] that by auto
  ultimately show ?thesis by auto
qed
```

3.9 Polynomial roots within a circle (open ball)

```
— Roots counted WITH multiplicity
definition proots-ball::complex poly ⇒ complex ⇒ real ⇒ nat where
  proots-ball p z0 r = proots-count p (ball z0 r)

— Roots counted WITHOUT multiplicity
definition proots-ball-card ::complex poly ⇒ complex ⇒ real ⇒ nat where
  proots-ball-card p z0 r = card (proots-within p (ball z0 r))

lemma proots-ball-code1[code]:
  proots-ball p z0 r = ( if r ≤ 0 then
                           0
                         else if p≠0 then
                          proots-upper (fcompose (p ∘p [:z0, of-real r:]) [:i,−1:] [:i,1:])
                         else
                             Code.abort (STR ''proots-ball fails when p=0.'')
                               (λ-. proots-ball p z0 r)
                       )
proof (cases p=0 ∨ r≤0)
  case False
  have proots-ball p z0 r = proots-count (p ∘p [:z0, of-real r:]) (ball 0 1)
    unfolding proots-ball-def
    apply (rule proots-uball-eq[THEN arg-cong])
    using False by auto
  also have ... = proots-upper (fcompose (p ∘p [:z0, of-real r:]) [:i,−1:] [:i,1:])
    unfolding proots-upper-def
    apply (rule proots-ball-plane-eq[THEN arg-cong])
    using False pcompose-eq-0[of p [:z0, of-real r:]] by auto
  finally show ?thesis using False by auto
qed (auto simp:proots-ball-def ball-empty)

lemma proots-ball-card-code1[code]:
  proots-ball-card p z0 r =
              ( if r ≤ 0 ∨ p=0 then
```

```
                0
            else
              proots-upper-card (fcompose (p ∘p [:z0, of-real r:]) [:i,−1:] [:i,1:])
                )
proof (cases p=0 ∨ r≤0)
  case True
  moreover have ?thesis when r≤0
  proof −
    have proots-within p (ball z0 r) = {}
      by (simp add: ball-empty that)
    then show ?thesis unfolding proots-ball-card-def using that by auto
  qed
  moreover have ?thesis when r>0 p=0
    unfolding proots-ball-card-def using that infinite-ball[of r z0]
    by auto
  ultimately show ?thesis by argo
next
  case False
  then have p≠0 r>0 by auto

  have proots-ball-card p z0 r = card (proots-within (p ∘p [:z0, of-real r:]) (ball 0
1))
    unfolding proots-ball-card-def
    by (rule proots-card-uball-eq[OF ‹r>0›, THEN arg-cong])
  also have ... = proots-upper-card (fcompose (p ∘p [:z0, of-real r:]) [:i,−1:] [:i,1:])
    unfolding proots-upper-card-def
    apply (rule proots-card-ball-plane-eq[THEN arg-cong])
    using False pcompose-eq-0[of p [:z0, of-real r:]] by auto
  finally show ?thesis using False by auto
qed
```

3.10 Polynomial roots on a circle (sphere)

```
— Roots counted WITH multiplicity
definition proots-sphere::complex poly ⇒ complex ⇒ real ⇒ nat where
  proots-sphere p z0 r = proots-count p (sphere z0 r)

— Roots counted WITHOUT multiplicity
definition proots-sphere-card ::complex poly ⇒ complex ⇒ real ⇒ nat where
  proots-sphere-card p z0 r = card (proots-within p (sphere z0 r))

lemma proots-sphere-card-code1[code]:
  proots-sphere-card p z0 r =
              ( if r=0 then
                  (if poly p z0=0 then 1 else 0)
                else if r < 0 ∨ p=0 then
                  0
                else
                  (if poly p (z0−r) =0 then 1 else 0) +
```

```
                proots-unbounded-line-card (fcompose (p ∘p [:z0, of-real r:]) [:i,−1:]
[:i,1:])
                    0 1
              )
proof −
  have ?thesis when r=0
  proof −
    have proots-within p {z0} = (if poly p z0 = 0 then {z0} else {})
      by auto
    then show ?thesis unfolding proots-sphere-card-def using that by simp
  qed
  moreover have ?thesis when r≠0 r < 0 ∨ p=0
  proof −
    have ?thesis when r<0
    proof −
      have proots-within p (sphere z0 r) = {}
        by (auto simp add: ball-empty that)
      then show ?thesis unfolding proots-sphere-card-def using that by auto
    qed
    moreover have ?thesis when r>0 p=0
      unfolding proots-sphere-card-def using that infinite-sphere[of r z0]
      by auto
    ultimately show ?thesis using that by argo
  qed
  moreover have ?thesis when r>0 p≠0
  proof −
    define pp where pp = p ∘p [:z0, of-real r:]
    define ppp where ppp=fcompose pp [:i, − 1:] [:i, 1:]

    have pp≠0 unfolding pp-def using that pcompose-eq-0 by fastforce

    have proots-sphere-card p z0 r = card (proots-within pp (sphere 0 1))
      unfolding proots-sphere-card-def pp-def
      by (rule proots-card-usphere-eq[OF ‹r>0›, THEN arg-cong])
    also have ... = card (proots-within pp {−1} ∪ proots-within pp (sphere 0 1 −
{−1}))
      by (simp add: insert-absorb proots-within-union)
    also have ... = card (proots-within pp {−1}) + card (proots-within pp (sphere
0 1 − {−1}))
      apply (rule card-Un-disjoint)
      using ‹pp≠0› by auto
    also have ... = card (proots-within pp {−1}) + card (proots-within ppp {x. 0
= Im x})
      using proots-card-sphere-axis-eq[OF ‹pp≠0›,folded ppp-def] by simp
   also have ... = (if poly p (z0−r) =0 then 1 else 0) + proots-unbounded-line-card
ppp 0 1
    proof −
      have proots-within pp {−1} = (if poly p (z0−r) =0 then {−1} else {})
        unfolding pp-def by (auto simp:poly-pcompose)
```

```
    then have card (proots-within pp {−1}) = (if poly p (z0−r) =0 then 1 else
0)
      by auto
    moreover have {x. Im x = 0} = unbounded-line 0 1
      unfolding unbounded-line-def
      apply auto
      by (metis complex-is-Real-iff of-real-Re of-real-def)
    then have card (proots-within ppp {x. 0 = Im x})
                  = proots-unbounded-line-card ppp 0 1
      unfolding proots-unbounded-line-card-def by simp
    ultimately show ?thesis by auto
  qed
  finally show ?thesis
    apply (fold pp-def,fold ppp-def)
    using that by auto
 qed
 ultimately show ?thesis by auto
qed
```

3.11 Polynomial roots on a closed ball

```
— Roots counted WITH multiplicity
definition proots-cball::complex poly ⇒ complex ⇒ real ⇒ nat where
  proots-cball p z0 r = proots-count p (cball z0 r)

— Roots counted WITHOUT multiplicity
definition proots-cball-card ::complex poly ⇒ complex ⇒ real ⇒ nat where
  proots-cball-card p z0 r = card (proots-within p (cball z0 r))


lemma proots-cball-card-code1[code]:
  proots-cball-card p z0 r =
              ( if r=0 then
                   (if poly p z0=0 then 1 else 0)
                else if r < 0 ∨ p=0 then
                   0
                else
                  ( let pp=fcompose (p ∘p [:z0, of-real r:]) [:i,−1:] [:i,1:]
                    in
                      (if poly p (z0−r) =0 then 1 else 0)
                      + proots-unbounded-line-card pp 0 1
                      + proots-upper-card pp
                  )
              )
proof −
  have ?thesis when r=0
  proof −
    have proots-within p {z0} = (if poly p z0 = 0 then {z0} else {})
      by auto
```

```
  then show ?thesis unfolding proots-cball-card-def using that by simp
 qed
 moreover have ?thesis when r≠0 r < 0 ∨ p=0
 proof −
  have ?thesis when r<0
  proof −
   have proots-within p (cball z0 r) = {}
    by (auto simp add: ball-empty that)
   then show ?thesis unfolding proots-cball-card-def using that by auto
  qed
  moreover have ?thesis when r>0 p=0
   unfolding proots-cball-card-def using that infinite-cball[of r z0]
   by auto
  ultimately show ?thesis using that by argo
 qed
 moreover have ?thesis when p≠0 r>0
 proof −
  define pp where pp=fcompose (p ∘p [:z0, of-real r:]) [:i,−1:] [:i,1:]

  have proots-cball-card p z0 r = card (proots-within p (sphere z0 r)
                                  ∪ proots-within p (ball z0 r))
   unfolding proots-cball-card-def
   apply (simp add:proots-within-union)
   by (metis Diff-partition cball-diff-sphere sphere-cball)
  also have ... = card (proots-within p (sphere z0 r)) + card (proots-within p
(ball z0 r))
   apply (rule card-Un-disjoint)
   using ‹p≠0› by auto
 also have ... = (if poly p (z0−r) =0 then 1 else 0) + proots-unbounded-line-card
pp 0 1
                  + proots-upper-card pp
  using proots-sphere-card-code1[of p z0 r,folded pp-def,unfolded proots-sphere-card-def]

    proots-ball-card-code1[of p z0 r,folded pp-def,unfolded proots-ball-card-def]
    that
   by simp
  finally show ?thesis
   apply (fold pp-def)
   using that by auto
 qed
 ultimately show ?thesis by auto
qed

end
```

4 Some examples for complex root counting

```
theory Count-Complex-Roots-Examples
 imports Count-Complex-Roots
```

begin

value *proots-rectangle* [:*2*∗i,*0*,i:] (*Complex* (−*1*) *0*) (*Complex 2 2*)

value *proots-rectangle* [:−*1*,−*2*∗i,*1*:]
(*Complex* (−*1*) *0*) (*Complex 2 2*)

value *proots-half* [:*1*−i,*2*−i,*1*:]
0 (*Complex 0 1*)

value *proots-half* [:*1*−i,*2*−i,*1*:] (*Complex 0 1*) *0*

value [*code*] *proots-ball* ([:−*2*,*1*:]∗[:−*2*,*1*:]∗[:−*3*,*1*:]) *0 4*

value [*code*] *proots-ball-card* ([:−*2*,*1*:]∗[:−*2*,*1*:]∗[:−*3*,*1*:]) *0 3*

end

References

[1] M. Eisermann. The fundamental theorem of algebra made effective: An elementary real-algebraic proof via Sturm chains. *American Mathematical Monthly*, 119(9):715–752, 2012.

[2] Q. I. Rahman and G. Schmeisser. *Analytic theory of polynomials*. Number 26. Oxford University Press, 2002.