Evaluate Winding Numbers through Cauchy Indices

Wenda Li

March 17, 2025

Abstract

In complex analysis, the winding number measures the number of times a path (counterclockwise) winds around a point, while the Cauchy index can approximate how the path winds. This entry provides a formalisation of the Cauchy index, which is then shown to be related to the winding number. In addition, this entry also offers a tactic that enables users to evaluate the winding number by calculating Cauchy indices. The connection between the winding number and the Cauchy index can be found in the literature [1] [2, Chapter 11].

1 Some useful lemmas in topology

```
theory Missing-Topology imports HOL−Analysis.Multivariate-Analysis
begin
```

1.1 Misc

```
lemma open-times-image:
  fixes S::'a::real-normed-field set
  assumes open S c≠0
  shows open (((∗) c) ‘ S)
proof −
  let ?f = λx. x/c and ?g=((∗) c)
  have continuous-on UNIV ?f using ‹c≠0› by (auto intro:continuous-intros)
  then have open (?f −‘ S) using ‹open S› by (auto elim:open-vimage)
  moreover have ?g ‘ S = ?f −‘ S using ‹c≠0›
    using image-iff by fastforce
  ultimately show ?thesis by auto
qed

lemma image-linear-greaterThan:
  fixes x::'a::linordered-field
  assumes c≠0
  shows ((λx. c∗x+b) ‘ {x<..}) = (if c>0 then {c∗x+b <..} else {..< c∗x+b})
using ‹c≠0›
  apply (auto simp add:image-iff field-simps)
  subgoal for y by (rule bexI[where x=(y−b)/c],auto simp add:field-simps)
```

subgoal for *y* **by** (*rule bexI*[**where** *x*=(*y*−*b*)/*c*],*auto simp add*:*field-simps*)
done

lemma *image-linear-lessThan*:
fixes *x*::′*a*::*linordered-field*
assumes *c*≠*0*
shows ((λ*x*. *c*∗*x*+*b*) ‘ {..<*x*}) = (*if c*>*0 then* {..<*c*∗*x*+*b*} *else* {*c*∗*x*+*b*<..})
using ‹*c*≠*0*›
apply (*auto simp add*:*image-iff field-simps*)
subgoal for *y* **by** (*rule bexI*[**where** *x*=(*y*−*b*)/*c*],*auto simp add*:*field-simps*)
subgoal for *y* **by** (*rule bexI*[**where** *x*=(*y*−*b*)/*c*],*auto simp add*:*field-simps*)
done

lemma *continuous-on-neq-split*:
fixes *f* :: ′*a*::*linear-continuum-topology* ⇒ ′*b*::*linorder-topology*
assumes ∀ *x*∈*s*. *f x*≠*y continuous-on s f connected s*
shows (∀ *x*∈*s*. *f x*>*y*) ∨ (∀ *x*∈*s*. *f x*<*y*)
by (*smt* (*verit*) *assms connectedD-interval connected-continuous-image imageE image-eqI leI*)

lemma
fixes *f*::′*a*::*linorder-topology* ⇒ ′*b*::*topological-space*
assumes *continuous-on* {*a*..*b*} *f a*<*b*
shows *continuous-on-at-left*:*continuous* (*at-left b*) *f*
and *continuous-on-at-right*:*continuous* (*at-right a*) *f*
using *assms continuous-on-Icc-at-leftD continuous-within* **apply** *blast*
using *assms continuous-on-Icc-at-rightD continuous-within* **by** *blast*

1.2 More about *eventually*

lemma *eventually-comp-filtermap*:
eventually (*P o f*) *F* ⟷ *eventually P* (*filtermap f F*)
unfolding *comp-def* **using** *eventually-filtermap* **by** *auto*

lemma *eventually-at-infinityI*:
fixes *P*::′*a*::*real-normed-vector* ⇒ *bool*
assumes ⋀*x*. *c* ≤ *norm x* ⟹ *P x*
shows *eventually P at-infinity*
unfolding *eventually-at-infinity* **using** *assms* **by** *auto*

lemma *eventually-at-bot-linorderI*:
fixes *c*::′*a*::*linorder*
assumes ⋀*x*. *x* ≤ *c* ⟹ *P x*
shows *eventually P at-bot*
using *assms* **by** (*auto simp*: *eventually-at-bot-linorder*)

1.3 More about *filtermap*

lemma *filtermap-linear-at-within*:
assumes *bij f* **and** *cont*: *isCont f a* **and** *open-map*: ⋀*S*. *open S* ⟹ *open* (*f*‘*S*)

shows *filtermap f* (*at a within S*) = *at* (*f a*) *within f'S*
unfolding *filter-eq-iff*
proof *safe*
fix *P*
assume *eventually P* (*filtermap f* (*at a within S*))
then obtain *T* **where** *open T a* ∈ *T* **and** *impP*:∀ *x*∈*T*. *x*≠*a* ⟶ *x*∈*S*⟶ *P* (*f x*)
by (*auto simp*: *eventually-filtermap eventually-at-topological*)
then show *eventually P* (*at* (*f a*) *within f ' S*)
unfolding *eventually-at-topological*
apply (*intro exI*[*of - f'T*])
using ‹*bij f*› *open-map* **by** (*metis bij-pointE imageE imageI*)
next
fix *P*
assume *eventually P* (*at* (*f a*) *within f ' S*)
then obtain *T1* **where** *open T1 f a* ∈ *T1* **and** *impP*:∀ *x*∈*T1*. *x*≠*f a* ⟶ *x*∈*f'S*⟶ *P* (*x*)
unfolding *eventually-at-topological* **by** *auto*
then obtain *T2* **where** *open T2 a* ∈ *T2* (∀ *x'*∈*T2*. *f x'* ∈ *T1*)
using *cont*[*unfolded continuous-at-open*,*rule-format*,*of T1*] **by** *blast*
then have ∀ *x*∈*T2*. *x*≠*a* ⟶ *x*∈*S*⟶ *P* (*f x*)
using *impP* **by** (*metis assms*(*1*) *bij-pointE imageI*)
then show *eventually P* (*filtermap f* (*at a within S*))
unfolding *eventually-filtermap eventually-at-topological*
apply (*intro exI*[*of - T2*])
using ‹*open T2*› ‹*a* ∈ *T2*› **by** *auto*
qed

lemma *filtermap-at-bot-linear-eq*:
fixes *c*::*'a*::*linordered-field*
assumes *c*≠*0*
shows *filtermap* (λ*x*. *x* ∗ *c* + *b*) *at-bot* = (*if c*>*0 then at-bot else at-top*)
proof (*cases c*>*0*)
case *True*
then have *filtermap* (λ*x*. *x* ∗ *c* + *b*) *at-bot* = *at-bot*
apply (*intro filtermap-fun-inverse*[*of* λ*x*. (*x*−*b*) / *c*])
subgoal unfolding *eventually-at-bot-linorder filterlim-at-bot*
by (*auto simp add*: *field-simps*)
subgoal unfolding *eventually-at-bot-linorder filterlim-at-bot*
by (*metis mult.commute real-affinity-le*)
by *auto*
then show *?thesis* **using** ‹*c*>*0*› **by** *auto*
next
case *False*
then have *c*<*0* **using** ‹*c*≠*0*› **by** *auto*
then have *filtermap* (λ*x*. *x* ∗ *c* + *b*) *at-bot* = *at-top*
apply (*intro filtermap-fun-inverse*[*of* λ*x*. (*x*−*b*) / *c*])
subgoal unfolding *eventually-at-top-linorder filterlim-at-bot*
by (*meson le-diff-eq neg-divide-le-eq*)

```
    subgoal unfolding eventually-at-bot-linorder filterlim-at-top
      using ‹c < 0› by (meson False diff-le-eq le-divide-eq)
    by auto
  then show ?thesis using ‹c<0› by auto
qed

lemma filtermap-linear-at-left:
  fixes c::'a::{linordered-field,linorder-topology,real-normed-field}
  assumes c≠0
  shows filtermap (λx. c∗x+b) (at-left x) = (if c>0 then at-left (c∗x+b) else
at-right (c∗x+b))
proof −
  let ?f = λx. c∗x+b
  have filtermap (λx. c∗x+b) (at-left x) = (at (?f x) within ?f ‘ {..<x})
  proof (subst filtermap-linear-at-within)
    show bij ?f using ‹c≠0›
      by (auto intro!: o-bij[of λx. (x−b)/c])
    show isCont ?f x by auto
    show ⋀S. open S ⟹ open (?f ‘ S)
      using open-times-image[OF - ‹c≠0›,THEN open-translation,of - b]
      by (simp add:image-image add.commute)
    show at (?f x) within ?f ‘ {..<x} = at (?f x) within ?f ‘ {..<x} by simp
  qed
  moreover have ?f ‘ {..<x} = {..<?f x} when c>0
    using image-linear-lessThan[OF ‹c≠0›,of b x] that by auto
  moreover have ?f ‘ {..<x} = {?f x<..} when ¬ c>0
    using image-linear-lessThan[OF ‹c≠0›,of b x] that by auto
  ultimately show ?thesis by auto
qed

lemma filtermap-linear-at-right:
  fixes c::'a::{linordered-field,linorder-topology,real-normed-field}
  assumes c≠0
  shows filtermap (λx. c∗x+b) (at-right x) = (if c>0 then at-right (c∗x+b) else
at-left (c∗x+b))
proof −
  let ?f = λx. c∗x+b
  have filtermap ?f (at-right x) = (at (?f x) within ?f ‘ {x<..})
  proof (subst filtermap-linear-at-within)
    show bij ?f using ‹c≠0›
      by (auto intro!: o-bij[of λx. (x−b)/c])
    show isCont ?f x by auto
    show ⋀S. open S ⟹ open (?f ‘ S)
      using open-times-image[OF - ‹c≠0›,THEN open-translation,of - b]
      by (simp add:image-image add.commute)
    show at (?f x) within ?f ‘ {x<..} = at (?f x) within ?f ‘ {x<..} by simp
  qed
  moreover have ?f ‘ {x<..} = {?f x<..} when c>0
    using image-linear-greaterThan[OF ‹c≠0›,of b x] that by auto
```

moreover have *?f ‘ {x<..} = {..<?f x}* **when** *¬ c>0*
using *image-linear-greaterThan[OF ‹c≠0›,of b x] that* **by** *auto*
ultimately show *?thesis* **by** *auto*
qed

lemma *filtermap-at-top-linear-eq*:
fixes *c::'a::linordered-field*
assumes *c≠0*
shows *filtermap (λx. x ∗ c + b) at-top = (if c>0 then at-top else at-bot)*
proof *(cases c>0)*
case *True*
then have *filtermap (λx. x ∗ c + b) at-top = at-top*
apply *(intro filtermap-fun-inverse[of λx. (x−b) / c])*
subgoal unfolding *eventually-at-top-linorder filterlim-at-top*
by *(meson le-diff-eq pos-le-divide-eq)*
subgoal unfolding *eventually-at-top-linorder filterlim-at-top*
apply *auto*
by *(metis mult.commute real-le-affinity)*
by *auto*
then show *?thesis* **using** *‹c>0›* **by** *auto*
next
case *False*
then have *c<0* **using** *‹c≠0›* **by** *auto*
then have *filtermap (λx. x ∗ c + b) at-top = at-bot*
apply *(intro filtermap-fun-inverse[of λx. (x−b) / c])*
subgoal unfolding *eventually-at-bot-linorder filterlim-at-top*
by *(auto simp add: field-simps)*
subgoal unfolding *eventually-at-top-linorder filterlim-at-bot*
by *(meson le-diff-eq neg-divide-le-eq)*
by *auto*
then show *?thesis* **using** *‹c<0›* **by** *auto*
qed

1.4 More about *filterlim*

lemma *filterlim-at-top-linear-iff*:
fixes *f::'a::linordered-field ⇒ 'b*
assumes *c≠0*
shows *(LIM x at-top. f (x ∗ c + b) :> F2) ⟷ (if c>0 then (LIM x at-top. f x :> F2)*
else (LIM x at-bot. f x :> F2))
unfolding *filterlim-def*
apply *(subst filtermap-filtermap[of f λx. x ∗ c + b,symmetric])*
using *assms* **by** *(auto simp add:filtermap-at-top-linear-eq)*

lemma *filterlim-at-bot-linear-iff*:
fixes *f::'a::linordered-field ⇒ 'b*
assumes *c≠0*
shows *(LIM x at-bot. f (x ∗ c + b) :> F2) ⟷ (if c>0 then (LIM x at-bot. f x*

:> *F2*)
    else (*LIM x at-top. f x* :> *F2*))
  unfolding *filterlim-def*
  apply (*subst filtermap-filtermap*[*of f* λ*x. x* ∗ *c* + *b*,*symmetric*])
  using *assms* **by** (*auto simp add*:*filtermap-at-bot-linear-eq*)

lemma *filterlim-tendsto-add-at-top-iff*:
  assumes *f*: (*f* ⟶ *c*) *F*
  shows (*LIM x F*. (*f x* + *g x* :: *real*) :> *at-top*) ⟷ (*LIM x F*. *g x* :> *at-top*)
proof
  assume *LIM x F*. *f x* + *g x* :> *at-top*
  moreover have ((λ*x*. − *f x*) ⟶ − *c*) *F*
    using *f* **by** (*intro tendsto-intros*,*simp*)
  ultimately show *filterlim g at-top F* **using** *filterlim-tendsto-add-at-top*
    by *fastforce*
qed (*auto simp add*:*filterlim-tendsto-add-at-top*[*OF f*])

lemma *filterlim-tendsto-add-at-bot-iff*:
  fixes *c*::*real*
  assumes *f*: (*f* ⟶ *c*) *F*
  shows (*LIM x F*. *f x* + *g x* :> *at-bot*) ⟷ (*LIM x F*. *g x* :> *at-bot*)
proof −
  have (*LIM x F*. *f x* + *g x* :> *at-bot*)
      ⟷ (*LIM x F*. − *f x* + (− *g x*) :> *at-top*)
    apply (*subst filterlim-uminus-at-top*)
    by (*rule filterlim-cong*,*auto*)
  also have ... = (*LIM x F*. − *g x* :> *at-top*)
    apply (*subst filterlim-tendsto-add-at-top-iff*[*of* - −*c*])
    by (*auto intro*:*tendsto-intros simp add*:*f*)
  also have ... = (*LIM x F*. *g x* :> *at-bot*)
    apply (*subst filterlim-uminus-at-top*)
    by (*rule filterlim-cong*,*auto*)
  finally show *?thesis* **.**
qed

lemma *tendsto-inverse-0-at-infinity*:
    LIM x F. *f x* :> *at-infinity* ⟹ ((λ*x*. *inverse* (*f x*) :: *real*) ⟶ *0*) *F*
  by (*metis filterlim-at filterlim-inverse-at-iff*)

end

2 Some useful lemmas in algebra

theory *Missing-Algebraic* **imports**
  HOL−*Computational-Algebra.Polynomial-Factorial*

HOL−Computational-Algebra.Fundamental-Theorem-Algebra
HOL−Complex-Analysis.Complex-Analysis
Missing-Topology
Budan-Fourier.BF-Misc
begin

2.1 Misc

lemma *poly-holomorphic-on*[*simp*]: (*poly p*) *holomorphic-on s*
by (*meson field-differentiable-def has-field-derivative-at-within holomorphic-onI poly-DERIV*)

```
lemma order-zorder:
  fixes p::complex poly and z::complex
  assumes p≠0
  shows order z p = nat (zorder (poly p) z)
proof −
  define n where n=nat (zorder (poly p) z)
  define h where h=zor-poly (poly p) z
  have ∃ w. poly p w ≠ 0 using assms poly-all-0-iff-0 by auto
  then obtain r where 0 < r cball z r ⊆ UNIV and
      h-holo: h holomorphic-on cball z r and
      poly-prod:(∀ w∈cball z r. poly p w = h w ∗ (w − z) ^ n ∧ h w ≠ 0)
    using zorder-exist-zero[of poly p UNIV z,folded h-def] poly-holomorphic-on
    unfolding n-def by auto
  then have h holomorphic-on ball z r
    and (∀ w∈ball z r. poly p w = h w ∗ (w − z) ^ n)
    and h z≠0
    by auto
  then have order z p = n using ‹p≠0›
  proof (induct n arbitrary:p h)
    case 0
    then have poly p z=h z using ‹r>0› by auto
    then have poly p z≠0 using ‹h z≠0› by auto
    then show ?case using order-root by blast
  next
    case (Suc n)
    define sn where sn=Suc n
    define h' where h'≡ λw. deriv h w ∗ (w−z)+ sn ∗ h w
    have (poly p has-field-derivative poly (pderiv p) w) (at w) for w
      using poly-DERIV[of p w] .
    moreover have (poly p has-field-derivative (h' w)∗(w−z)^n ) (at w) when
w∈ball z r for w
      proof (subst DERIV-cong-ev[of w w poly p λw.  h w ∗ (w − z) ^ Suc n
],simp-all)
      show ∀F x in nhds w. poly p x = h x ∗ ((x − z) ∗ (x − z) ^ n)
        unfolding eventually-nhds using Suc(3) ‹w∈ball z r›
        by (metis Elementary-Metric-Spaces.open-ball power-Suc)
      next
```

```
        have (h has-field-derivative deriv h w) (at w)
        using ‹h holomorphic-on ball z r› ‹w∈ball z r› holomorphic-on-imp-differentiable-at

          by (simp add: holomorphic-derivI)
        then have ((λw. h w * ((w − z) ^ sn))
                   has-field-derivative h′ w * (w − z) ^ (sn − 1)) (at w)
          unfolding h′-def
          apply (auto intro!: derivative-eq-intros simp add:field-simps)
          by (auto simp add:field-simps sn-def)
        then show ((λw. h w * ((w − z) * (w − z) ^ n))
                   has-field-derivative h′ w * (w − z) ^ n) (at w)
          unfolding sn-def by auto
      qed
    ultimately have ∀ w∈ball z r. poly (pderiv p) w = h′ w * (w − z) ^ n
      using DERIV-unique by blast
    moreover have h′ holomorphic-on ball z r
      unfolding h′-def using ‹h holomorphic-on ball z r›
      by (auto intro!: holomorphic-intros)
    moreover have h′ z≠0 unfolding h′-def sn-def using ‹h z ≠ 0› of-nat-neq-0
by auto
    moreover have pderiv p ≠ 0
    proof
      assume pderiv p = 0
      obtain c where p=[:c:] using ‹pderiv p = 0› using pderiv-iszero by blast
      then have c=0
        using Suc(3)[rule-format,of z] ‹r>0› by auto
      then show False using ‹p≠0› using ‹p=[:c:]› by auto
    qed
    ultimately have order z (pderiv p) = n by (auto elim: Suc.hyps)
    moreover have order z p ≠ 0
      using Suc(3)[rule-format,of z] ‹r>0› order-root ‹p≠0› by auto
    ultimately show ?case using order-pderiv[OF ‹pderiv p ≠ 0›] by auto
  qed
  then show ?thesis unfolding n-def .
qed

lemma pcompose-pCons-0:pcompose p [:a:] = [:poly p a:]
 by (metis (no-types, lifting) coeff-pCons-0 pcompose-0′ pcompose-assoc poly-0-coeff-0
poly-pcompose)

lemma pcompose-coeff-0:
  coeff (pcompose p q) 0 = poly p (coeff q 0)
  by (metis poly-0-coeff-0 poly-pcompose)

lemma poly-field-differentiable-at[simp]:
  poly p field-differentiable (at x within s)
  using field-differentiable-at-within field-differentiable-def poly-DERIV by blast

lemma deriv-pderiv:
```

deriv (*poly p*) = *poly* (*pderiv p*)
by (*meson ext DERIV-imp-deriv poly-DERIV*)

lemma *lead-coeff-map-poly-nz*:
assumes *f* (*lead-coeff p*) ≠*0 f 0*=*0*
shows *lead-coeff* (*map-poly f p*) = *f* (*lead-coeff p*)
by (*metis* (*no-types*, *lifting*) *antisym assms coeff-0 coeff-map-poly le-degree leading-coeff-0-iff*)

lemma *filterlim-poly-at-infinity*:
fixes *p*::′*a*::*real-normed-field poly*
assumes *degree p*>*0*
shows *filterlim* (*poly p*) *at-infinity at-infinity*
using *assms*
proof (*induct p*)
case *0*
then show *?case* **by** *auto*
next
case (*pCons a p*)
have *?case* **when** *degree p*=*0*
proof −
obtain *c* **where** *c-def*:*p*=[:*c*:] **using** ‹*degree p* = *0*› *degree-eq-zeroE* **by** *blast*
then have *c*≠*0* **using** ‹*0* < *degree* (*pCons a p*)› **by** *auto*
then show *?thesis* **unfolding** *c-def*
apply (*auto intro*!:*tendsto-add-filterlim-at-infinity*)
apply (*subst mult.commute*)
by (*auto intro*!:*tendsto-mult-filterlim-at-infinity filterlim-ident*)
qed
moreover have *?case* **when** *degree p*≠*0*
proof −
have *filterlim* (*poly p*) *at-infinity at-infinity*
using *that* **by** (*auto intro*:*pCons*)
then show *?thesis*
by (*auto intro*!:*tendsto-add-filterlim-at-infinity filterlim-at-infinity-times filterlim-ident*)
qed
ultimately show *?case* **by** *auto*
qed

lemma *poly-divide-tendsto-aux*:
fixes *p*::′*a*::*real-normed-field poly*
shows ((λ*x*. *poly p x*/*x*^(*degree p*)) ⟶ *lead-coeff p*) *at-infinity*
proof (*induct p*)
case *0*
then show *?case* **by** (*auto intro*:*tendsto-eq-intros*)
next
case (*pCons a p*)
have *?case* **when** *p*=*0*
using *that* **by** *auto*

```
  moreover have ?case when p≠0
  proof -
    define g where g=(λx. a/(x*x^degree p))
    define f where f=(λx. poly p x/x^degree p)
    have ∀F x in at-infinity. poly (pCons a p) x / x ^ degree (pCons a p) = g x +
f x
    proof (rule eventually-at-infinityI[of 1])
      fix x::'a assume norm x≥1
      then have x≠0 by auto
      then show poly (pCons a p) x / x ^ degree (pCons a p) = g x + f x
        using that unfolding g-def f-def by (auto simp add:field-simps)
    qed
    moreover have ((λx. g x+f x) ⟶ lead-coeff (pCons a p)) at-infinity
    proof -
      have (g ⟶ 0) at-infinity
          unfolding g-def using filterlim-poly-at-infinity[of monom 1 (Suc (degree
p))]
        apply (auto intro!:tendsto-intros tendsto-divide-0 simp add: degree-monom-eq)
        apply (subst filterlim-cong[where g=poly (monom 1 (Suc (degree p)))])
        by (auto simp add:poly-monom)
      moreover have (f ⟶ lead-coeff (pCons a p)) at-infinity
        using pCons ‹p≠0› unfolding f-def by auto
      ultimately show ?thesis by (auto intro:tendsto-eq-intros)
    qed
    ultimately show ?thesis by (auto dest:tendsto-cong)
  qed
  ultimately show ?case by auto
qed

lemma filterlim-power-at-infinity:
  assumes n≠0
  shows filterlim (λx::'a::real-normed-field. x^n) at-infinity at-infinity
  using filterlim-poly-at-infinity[of monom 1 n] assms
  apply (subst filterlim-cong[where g=poly (monom 1 n)])
  by (auto simp add:poly-monom degree-monom-eq)

lemma poly-divide-tendsto-0-at-infinity:
  fixes p::'a::real-normed-field poly
  assumes degree p > degree q
  shows ((λx. poly q x / poly p x) ⟶ 0 ) at-infinity
proof -
  define pp where pp=(λx. x^(degree p) / poly p x)
  define qq where qq=(λx. poly q x/x^(degree q))
  define dd where dd=(λx::'a. 1/x^(degree p - degree q))
  have ∀F x in at-infinity.  poly q x / poly p x = qq x * pp x * dd x
  proof (rule eventually-at-infinityI[of 1])
    fix x::'a assume norm x≥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 divide-simps power-diff*)
qed
moreover have $((\lambda x.\ qq\ x * pp\ x * dd\ x) \longrightarrow 0)$ *at-infinity*
proof −
have $(qq \longrightarrow$ *lead-coeff q*) *at-infinity*
unfolding *qq-def* **using** *poly-divide-tendsto-aux*[*of q*] **.**
moreover have $(pp \longrightarrow 1/$*lead-coeff p*) *at-infinity*
proof −
have $p\neq 0$ **using** *assms* **by** *auto*
then show *?thesis*
unfolding *pp-def* **using** *poly-divide-tendsto-aux*[*of p*]
apply (*drule-tac tendsto-inverse*)
by (*auto simp add*:*inverse-eq-divide*)
qed
moreover have $(dd \longrightarrow 0)$ *at-infinity*
unfolding *dd-def*
apply (*rule tendsto-divide-0*)
by (*auto intro*!: *filterlim-power-at-infinity simp add*:*assms*)
ultimately show *?thesis* **by** (*auto intro*:*tendsto-eq-intros*)
qed
ultimately show *?thesis* **by** (*auto dest*:*tendsto-cong*)
qed

lemma *lead-coeff-list-def*:
lead-coeff p= (*if coeffs p*=[] *then 0 else last* (*coeffs p*))
by (*simp add*: *last-coeffs-eq-coeff-degree*)

lemma *poly-linepath-comp*:
fixes *a*::′*a*::{*real-normed-vector*,*comm-semiring-0*,*real-algebra-1*}
shows *poly p o* (*linepath a b*) = *poly* ($p \circ_p$ [:*a*, *b*−*a*:]) *o of-real*
by (*force simp add*:*poly-pcompose linepath-def scaleR-conv-of-real algebra-simps*)

lemma *poly-eventually-not-zero*:
fixes *p*::*real poly*
assumes $p\neq 0$
shows *eventually* (λx. *poly p x*$\neq 0$) *at-infinity*
proof (*rule eventually-at-infinityI*[*of Max* (*norm* ‘ {*x*. *poly p x*=*0*}) + *1*])
fix *x*::*real* **assume** *asm*:*Max* (*norm* ‘ {*x*. *poly p x*=*0*}) + *1* ≤ *norm x*
have *False* **when** *poly p x*=*0*
proof −
define *S* **where** *S*=*norm* ‘{*x*. *poly p x* = *0*}
have *norm x*∈*S* **using** *that* **unfolding** *S-def* **by** *auto*
moreover have *finite S* **using** ‹*p*≠*0*› *poly-roots-finite* **unfolding** *S-def* **by** *blast*
ultimately have *norm x*≤*Max S* **by** *simp*
moreover have *Max S* + *1* ≤ *norm x* **using** *asm* **unfolding** *S-def* **by** *simp*
ultimately show *False* **by** *argo*
qed

then show *poly p x* ≠ *0* **by** *auto*
qed

2.2 More about *degree*

lemma *map-poly-degree-eq*:
  assumes *f* (*lead-coeff p*) ≠*0*
  shows *degree* (*map-poly f p*) = *degree p*
  using *assms*
  unfolding *map-poly-def degree-eq-length-coeffs coeffs-Poly lead-coeff-list-def*
  by (*metis* (*full-types*) *last-conv-nth-default length-map no-trailing-unfold nth-default-coeffs-eq*

    nth-default-map-eq strip-while-idem)

lemma *map-poly-degree-less*:
  assumes *f* (*lead-coeff p*) =*0 degree p*≠*0*
  shows *degree* (*map-poly f p*) < *degree p*
proof −
  have *length* (*coeffs p*) >*1*
    using ‹*degree p*≠*0*› **by** (*simp add*: *degree-eq-length-coeffs*)
  then obtain *xs x* **where** *xs-def*:*coeffs p*=*xs*@[*x*] *length xs*>*0*
   by (*metis One-nat-def add-0 append-Nil length-greater-0-conv list.size*(*4*) *nat-neq-iff not-less-zero rev-exhaust*)
  have *f x*=*0* **using** *assms*(*1*) **by** (*simp add*: *lead-coeff-list-def xs-def*(*1*))
  have *degree* (*map-poly f p*) = *length* (*strip-while* ((=) *0*) (*map f* (*xs*@[*x*]))) − *1*
    unfolding *map-poly-def degree-eq-length-coeffs coeffs-Poly*
    by (*subst xs-def*,*auto*)
  also have ... = *length* (*strip-while* ((=) *0*) (*map f xs*)) − *1*
    using ‹*f x*=*0*› **by** *simp*
  also have ... ≤ *length xs* −*1*
    using *length-strip-while-le* **by** (*metis diff-le-mono length-map*)
  also have ... < *length* (*xs*@[*x*]) − *1*
    using *xs-def*(*2*) **by** *auto*
  also have ... = *degree p*
    unfolding *degree-eq-length-coeffs xs-def* **by** *simp*
  finally show *?thesis* **.**
qed

lemma *map-poly-degree-leq*[*simp*]:
  shows *degree* (*map-poly f p*) ≤ *degree p*
  unfolding *map-poly-def degree-eq-length-coeffs*
  by (*metis coeffs-Poly diff-le-mono length-map length-strip-while-le*)

2.3 roots / zeros of a univariate function

definition *roots-within*::(′*a* ⇒ ′*b*::*zero*) ⇒ ′*a set* ⇒ ′*a set* **where**
  roots-within f s = {*x*∈*s*. *f x* = *0*}

abbreviation *roots*::(′*a* ⇒ ′*b*::*zero*) ⇒ ′*a set* **where**
  roots f ≡ *roots-within f UNIV*

2.4 The argument principle specialised to polynomials.

lemma *argument-principle-poly*:
assumes *p≠0* **and** *valid*:*valid-path g* **and** *loop*: *pathfinish g* = *pathstart g*
and *no-proots*:*path-image g* ⊆ − *proots p*
shows *contour-integral g* (λ*x*. *deriv* (*poly p*) *x* / *poly p x*) = *2* ∗ *of-real pi* ∗ i ∗
(∑ *x*∈*proots p*. *winding-number g x* ∗ *of-nat* (*order x p*))
proof −
have *contour-integral g* (λ*x*. *deriv* (*poly p*) *x* / *poly p x*) = *2* ∗ *of-real pi* ∗ i ∗
(∑ *x* | *poly p x* = *0*. *winding-number g x* ∗ *of-int* (*zorder* (*poly p*) *x*))
apply (*rule argument-principle*[*of UNIV poly p* {} λ-. *1 g*,*simplified*,*OF - valid loop*])
using *no-proots*[*unfolded proots-def*] **by** (*auto simp add*:*poly-roots-finite*[*OF* ‹*p≠0*›])
also have ... = *2* ∗ *of-real pi* ∗ i ∗ (∑ *x*∈*proots p*. *winding-number g x* ∗ *of-nat* (*order x p*))
proof −
have *nat* (*zorder* (*poly p*) *x*) = *order x p* **when** *x*∈*proots p* **for** *x*
using *order-zorder*[*OF* ‹*p≠0*›] *that* **unfolding** *proots-def* **by** *auto*
then show *?thesis* **unfolding** *proots-def*
apply (*auto intro*!: *sum.cong*)
by (*metis assms*(*1*) *nat-eq-iff2 of-nat-nat order-root*)
qed
finally show *?thesis* **.**
qed

end

3 Some useful lemmas about transcendental functions

theory *Missing-Transcendental* **imports**
Missing-Topology
Missing-Algebraic
begin

3.1 Misc

lemma *exp-Arg2pi2pi-multivalue*:
assumes *exp* (i ∗ *of-real x*) = *z*
shows ∃ *k*::*int*. *x* = *Arg2pi z* + *2*∗*k*∗*pi*
proof −
define *k* **where** *k*=*floor*(*x*/(*2*∗*pi*))
define *x*′ **where** *x*′= *x* − (*2*∗*k*∗*pi*)
have *x*′/(*2*∗*pi*) ≥*0* **unfolding** *x*′-*def k-def* **by** (*simp add*: *diff-divide-distrib*)
moreover have *x*′/(*2*∗*pi*) < *1*
proof −
have *x*/(*2*∗*pi*) − *k* < *1* **unfolding** *k-def* **by** *linarith*
thus *?thesis* **unfolding** *k-def x*′-*def* **by** (*auto simp add*:*field-simps*)

```
  qed
  ultimately have x′≥0 and x′<2∗pi by (auto simp add:field-simps)
  moreover have exp (i ∗ complex-of-real x′) = z
    using assms x′-def by  (auto simp add:field-simps )
  ultimately have Arg2pi z = x′ using Arg2pi-unique[of 1 x′ z,simplified] by auto
  hence  x = Arg2pi z + 2∗k∗pi unfolding x′-def by auto
  thus ?thesis by auto
qed

lemma uniform-discrete-tan-eq:
  uniform-discrete {x::real. tan x = y}
proof −
  have x1=x2 when dist:dist x1 x2<pi/2 and tan x1=y tan x2=y for x1 x2
  proof −
     obtain k1::int where x1:x1 = arctan y + k1∗pi ∨ (x1 = pi/2 + k1∗pi ∧
y=0)
      using tan-eq-arctan-Ex ‹tan x1=y› by auto
     obtain k2::int where x2:x2 = arctan y + k2∗pi ∨ (x2 = pi/2 + k2∗pi ∧
y=0)
      using tan-eq-arctan-Ex ‹tan x2=y› by auto
    let ?xk1=x1 = arctan y + k1∗pi and ?xk1′=x1 = pi/2 + k1∗pi ∧ y=0
    let ?xk2=x2 = arctan y + k2∗pi and ?xk2′=x2 = pi/2 + k2∗pi ∧ y=0
    have ?thesis when (?xk1 ∧ ?xk2) ∨ (?xk1′ ∧ ?xk2′)
    proof −
      have x1−x2= (k1 − k2) ∗pi when ?xk1 ?xk2
        using arg-cong2[where f=minus,OF ‹?xk1› ‹?xk2›]
        by (auto simp add:algebra-simps)
      moreover have x1−x2= (k1 − k2) ∗pi when ?xk1′ ?xk2′
        using arg-cong2[where f=minus,OF conjunct1[OF ‹?xk1′›] conjunct1[OF
‹?xk2′›]]
        by (auto simp add:algebra-simps)
      ultimately have x1−x2= (k1 − k2) ∗pi using that by auto
      then have |k1 − k2| < 1/2
        using dist[unfolded dist-real-def] by (auto simp add:abs-mult)
      then have k1=k2 by linarith
      then show ?thesis using that by auto
    qed
    moreover have ?thesis when ?xk1 ?xk2′
    proof −
      have x1 = k1∗pi x2 = pi / 2 + k2 ∗ pi using ‹?xk2′› ‹?xk1› by auto
       from arg-cong2[where f=minus,OF this] have x1 − x2 = (k1 − k2) ∗ pi
−pi/2
        by (auto simp add:algebra-simps)
      then have |(k1 − k2) ∗ pi −pi/2| < pi/2 using dist[unfolded dist-real-def]
by auto
      then have 0<k1−k2 k1−k2<1
        unfolding abs-less-iff  by (auto simp add: zero-less-mult-iff)
      then have False by simp
      then show ?thesis by auto
```

```
    qed
    moreover have ?thesis when ?xk1' ?xk2
    proof -
      have x1 = pi / 2 + k1*pi x2 = k2 * pi using ‹?xk2› ‹?xk1'› by auto
       from arg-cong2[where f=minus,OF this] have x1 − x2 = (k1 − k2) * pi
+ pi/2
        by (auto simp add:algebra-simps)
      then have |(k1 − k2) * pi + pi/2| < pi/2 using dist[unfolded dist-real-def]
by auto
      then have |(k1 − k2 + 1/2)*pi| < pi/2 by (auto simp add:algebra-simps)
      then have |(k1 − k2 + 1/2)| < 1/2 by (auto simp add:abs-mult)
      then have −1<k1−k2 ∧ k1−k2<0
        unfolding abs-less-iff by linarith
      then have False by auto
      then show ?thesis by auto
    qed
    ultimately show ?thesis using x1 x2 by blast
  qed
  then show ?thesis unfolding uniform-discrete-def
    apply (intro exI[where x=pi/2])
    by auto
qed

lemma get-norm-value:
  fixes a::'a::{floor-ceiling}
  assumes pp>0
  obtains k::int and a1 where a=(of-int k)*pp+a1 a0≤a1 a1<a0+pp
proof -
  define k where k=floor ((a−a0)/pp)
  define a1 where a1=a−(of-int k)*pp
  have of-int ⌊(a − a0) / pp⌋ * pp ≤ a− a0
    using assms by (meson le-divide-eq of-int-floor-le)
  moreover have a−a0 < of-int (⌊(a − a0) / pp⌋+1) * pp
    using assms by (meson divide-less-eq floor-correct)
  ultimately show ?thesis
    apply (intro that[of k a1])
    unfolding k-def a1-def using assms by (auto simp add:algebra-simps)
qed


lemma filtermap-tan-at-right:
  fixes a::real
  assumes cos a≠0
  shows filtermap tan (at-right a) = at-right (tan a)
proof -
  obtain k::int and a1 where aa1:a=k*pi+a1 and pi-a1: −pi/2≤a1 a1<pi/2
    using get-norm-value[of pi a −pi/2] by auto
  have −pi/2 < a1
  using assms
```

by (*smt* (*verit*, *ccfv-SIG*) *pi-a1 aa1 cos-2pi-minus cos-diff cos-pi-half cos-two-pi divide-minus-left mult-of-int-commute sin-add sin-npi-int sin-pi-half sin-two-pi*)
have *eventually P* (*at-right* (*tan a*))
when *eventually P* (*filtermap tan* (*at-right a*)) **for** *P*
proof −
obtain *b1* **where** *b1*>*a* **and** *b1-imp*: ∀ *y*>*a*. *y* < *b1* ⟶ *P* (*tan y*)
by (*metis Sturm-Tarski.eventually-at-right* ‹*eventually P* (*filtermap tan* (*at-right a*))› *eventually-filtermap*)
define *b2* **where** *b2*=*min b1* (*k*∗*pi*+*pi*/*4*+*a1*/*2*)
define *b3* **where** *b3*=*b2* − *k*∗*pi*
have −*pi*/*2* < *b3* *b3*<*pi*/*2*
proof −
have *a1*<*b3*
using ‹*b1*>*a*› *aa1* ‹*a1*<*pi*/*2*› **unfolding** *b2-def b3-def* **by** (*auto simp add*:*field-simps*)
then show −*pi*/*2* < *b3* **using** ‹−*pi*/*2*≤*a1*› **by** *auto*
show *b3* < *pi*/*2*
using *b2-def b3-def pi-a1*(*2*) **by** *linarith*
qed
have *tan b2* > *tan a*
proof −
have *tan a* = *tan a1*
using *aa1* **by** (*simp add*: *add.commute*)
also have ... < *tan b3*
proof −
have *a1*<*b3*
using ‹*b1*>*a*› *aa1* ‹*a1*<*pi*/*2*› **unfolding** *b2-def b3-def* **by** (*auto simp add*:*field-simps*)
then show *?thesis*
using *tan-monotone* ‹−*pi*/*2* < *a1*› ‹*b3* < *pi*/*2*› **by** *simp*
qed
also have ... = *tan b2* **unfolding** *b3-def*
by (*metis Groups.mult-ac*(*2*) *add-uminus-conv-diff mult-minus-right of-int-minus*

tan-periodic-int)
finally show *?thesis* **.**
qed
moreover have *P y* **when** *y*>*tan a* *y* < *tan b2* **for** *y*
proof −
define *y1* **where** *y1*=*arctan y*+ *k* ∗ *pi*
have *a*<*y1*
proof −
have *arctan* (*tan a*) < *arctan y* **using** ‹*y*>*tan a*› *arctan-monotone* **by** *auto*
then have *a1*<*arctan y*
using *arctan-tan* ‹−*pi*/*2* < *a1*› ‹*a1*<*pi*/*2*› **unfolding** *aa1* **by** (*simp add*: *add.commute*)
then show *?thesis* **unfolding** *y1-def aa1* **by** *auto*
qed
moreover have *y1*<*b2*

proof −
have *arctan y* < *arctan* (*tan b2*)
using ‹*y* < *tan b2*› *arctan-monotone* **by** *auto*
moreover have *arctan* (*tan b2*) = *b3*
using *arctan-tan*[*of b3*] ‹−*pi*/*2* < *b3*› ‹*b3*<*pi*/*2*› **unfolding** *b3-def*
by (*metis add.inverse-inverse diff-minus-eq-add divide-minus-left mult.commute*

mult-minus-right of-int-minus tan-periodic-int)
ultimately have *arctan y* < *b3* **by** *auto*
then show *?thesis* **unfolding** *y1-def b3-def* **by** *auto*
qed
moreover have ∀ *y*>*a*. *y* < *b2* ⟶ *P* (*tan y*)
using *b1-imp* **unfolding** *b2-def* **by** *auto*
moreover have *tan y1*=*y* **unfolding** *y1-def* **by** (*auto simp add*:*tan-arctan*)
ultimately show *?thesis* **by** *auto*
qed
ultimately show *eventually P* (*at-right* (*tan a*))
unfolding *eventually-at-right* **by** (*metis eventually-at-right-field*)
qed
moreover have *eventually P* (*filtermap tan* (*at-right a*))
when *eventually P* (*at-right* (*tan a*)) **for** *P*
proof −
obtain *b1* **where** *b1*>*tan a* **and** *b1-imp*:∀ *y*>*tan a*. *y* < *b1* ⟶ *P y*
using ‹*eventually P* (*at-right* (*tan a*))› **unfolding** *eventually-at-right*
by (*metis eventually-at-right-field*)
define *b2* **where** *b2*=*arctan b1* + *k*∗*pi*
have *a1* < *arctan b1*
by (*metis* ‹− *pi* / *2* < *a1*› ‹*a1* < *pi* / *2*› ‹*tan a* < *b1*› *aa1 add.commute*
arctan-less-iff
arctan-tan divide-minus-left tan-periodic-int)
then have *b2*>*a* **unfolding** *aa1 b2-def* **by** *auto*
moreover have *P* (*tan y*) **when** *y*>*a y* < *b2* **for** *y*
proof −
define *y1* **where** *y1* = *y* − *k*∗*pi*
have *a1* < *y1 y1* < *arctan b1* **unfolding** *y1-def*
subgoal using ‹*y*>*a*› **unfolding** *aa1* **by** *auto*
subgoal using *b2-def that*(*2*) **by** *linarith*
done
then have *tan a1* < *tan y1 tan y1*< *b1*
subgoal using ‹*a1*>−*pi*/*2*›
apply (*intro tan-monotone*,*simp*,*simp*)
using *arctan-ubound less-trans* **by** *blast*
subgoal
by (*metis* ‹− *pi* / *2* < *a1*› ‹*a1* < *y1*› ‹*y1* < *arctan b1*› *arctan-less-iff*
arctan-tan
arctan-ubound divide-minus-left less-trans)
done
have *tan y*>*tan a*
by (*metis* ‹*tan a1* < *tan y1*› *aa1 add.commute add-uminus-conv-diff*

mult.commute
mult-minus-right of-int-minus tan-periodic-int y1-def)
moreover have *tan y<b1*
by (*metis* ‹*tan y1 < b1*› *add-uminus-conv-diff mult.commute mult-minus-right*

of-int-minus tan-periodic-int y1-def)
ultimately show *?thesis* **using** *b1-imp* **by** *auto*
qed
ultimately show *?thesis* **unfolding** *eventually-filtermap eventually-at-right*
by (*metis eventually-at-right-field*)
qed
ultimately show *?thesis* **unfolding** *filter-eq-iff* **by** *blast*
qed

lemma *filtermap-tan-at-left*:
fixes *a*::*real*
assumes *cos a≠0*
shows *filtermap tan* (*at-left a*) = *at-left* (*tan a*)
proof −
have *filtermap tan* (*at-right* (− *a*)) = *at-right* (*tan* (− *a*))
using *filtermap-tan-at-right*[*of* −*a*] *assms* **by** *auto*
then have *filtermap* (*uminus o tan*) (*at-left a*) = *filtermap uminus* (*at-left* (*tan a*))
unfolding *at-right-minus filtermap-filtermap comp-def* **by** *auto*
then have *filtermap uminus* (*filtermap* (*uminus o tan*) (*at-left a*))
= *filtermap uminus* (*filtermap uminus* (*at-left* (*tan a*)))
by *auto*
then show *?thesis*
unfolding *filtermap-filtermap comp-def* **by** *auto*
qed

lemma *filtermap-tan-at-right-inf*:
fixes *a*::*real*
assumes *cos a=0*
shows *filtermap tan* (*at-right a*) = *at-bot*
proof −
obtain *k*::*int* **where** *ak*:*a=k∗pi + pi/2*
using *cos-zero-iff-int2 assms* **by** *auto*
have *eventually P at-bot* **when** *eventually P* (*filtermap tan* (*at-right a*)) **for** *P*
proof −
obtain *b1* **where** *b1>a* **and** *b1-imp*:∀ *y>a. y < b1* ⟶ *P* (*tan y*)
using ‹*eventually P* (*filtermap tan* (*at-right a*))›
unfolding *eventually-filtermap eventually-at-right*
by (*metis eventually-at-right-field*)
define *b2* **where** *b2=min* (*k∗pi+pi*) *b1*
have *P y* **when** *y<tan b2* **for** *y*
proof −
define *y1* **where** *y1=*(*k+1*)*∗pi+arctan y*
have *a < y1*

```
      unfolding ak y1-def using arctan-lbound[of y]
      by (auto simp add:field-simps)
    moreover have y1 < b2
    proof −
      define b3 where b3=b2−(k+1) ∗ pi
      have −pi/2 < b3 b3<pi/2
        using ‹b1>a› unfolding b3-def b2-def ak
     by (auto simp add:field-simps min-mult-distrib-left intro!:min.strict-coboundedI1)
      then have arctan (tan b3) = b3
        by (simp add: arctan-tan)
      then have arctan (tan b2) = b3
        unfolding b3-def by (metis diff-eq-eq tan-periodic-int)
      then have arctan y < b3
        using arctan-monotone[OF ‹y<tan b2›] by simp
      then show ?thesis
        unfolding y1-def b3-def by auto
    qed
    then have y1<b1 unfolding b2-def by auto
    ultimately have  P (tan y1) using b1-imp[rule-format,of y1,simplified] by
auto
    then show ?thesis unfolding y1-def by (metis add.commute arctan tan-periodic-int)
  qed
  then show ?thesis unfolding eventually-at-bot-dense by auto
 qed
 moreover have eventually P (filtermap tan (at-right a)) when eventually P
at-bot for P
 proof −
  obtain b1 where b1-imp:∀ n<b1. P n
    using ‹eventually P at-bot› unfolding eventually-at-bot-dense by auto
  define b2 where b2=arctan b1 + (k+1)∗pi
  have b2>a unfolding ak b2-def using arctan-lbound[of b1]
    by (auto simp add:algebra-simps)
  moreover have P (tan y) when a < y  y < b2  for y
  proof −
    define y1 where y1=y−(k+1)∗pi
    have tan y1 < tan (arctan b1)
      apply (rule tan-monotone)
     subgoal using ‹a<y› unfolding y1-def ak by (auto simp add:algebra-simps)
    subgoal using ‹y < b2› unfolding y1-def b2-def by (auto simp add:algebra-simps)
      subgoal using arctan-ubound by auto
      done
    then have tan y1<b1 by (simp add: arctan)
    then have tan y < b1 unfolding y1-def
      by (metis diff-eq-eq tan-periodic-int)
    then show ?thesis using b1-imp by auto
  qed
  ultimately show eventually P (filtermap tan (at-right a))
    unfolding eventually-filtermap eventually-at-right
    by (metis eventually-at-right-field)
```

qed
ultimately show *?thesis* **unfolding** *filter-eq-iff* **by** *auto*
qed

lemma *filtermap-tan-at-left-inf*:
fixes *a::real*
assumes *cos a=0*
shows *filtermap tan (at-left a) = at-top*
proof −
have *filtermap tan (at-right (− a)) = at-bot*
using *filtermap-tan-at-right-inf[of −a] assms* **by** *auto*
then have *filtermap (uminus o tan) (at-left a) = at-bot*
unfolding *at-right-minus filtermap-filtermap comp-def* **by** *auto*
then have *filtermap uminus (filtermap (uminus o tan) (at-left a)) = filtermap uminus at-bot*
by *auto*
then show *?thesis*
unfolding *filtermap-filtermap comp-def* **using** *at-top-mirror[***where** *'a=real]* **by** *auto*
qed

3.2 Periodic set

definition *periodic-set:: real set ⇒ real ⇒ bool* **where**
periodic-set S δ ⟷ (∃B. finite B ∧ (∀x∈S. ∃b∈B. ∃k::int. x =b + k ∗ δ))

lemma *periodic-set-multiple*:
assumes *k≠0*
shows *periodic-set S δ ⟷ periodic-set S (of-int k∗δ)*
proof
assume *asm:periodic-set S δ*
then obtain *B1* **where** *finite B1* **and** *B1-def:∀x∈S. ∃b∈B1. (∃k::int. x = b + k ∗ δ)*
unfolding *periodic-set-def* **by** *metis*
define *B* **where** *B = B1 ∪ {b+i∗δ | b i. b∈B1 ∧ i∈{0..<|k|}}*
have *∃b∈B. ∃k'. x = b + real-of-int k' ∗ (real-of-int k ∗ δ)* **when** *x∈S* **for** *x*
proof −
obtain *b1* **and** *k1::int* **where** *b1∈B1* **and** *x-δ:x = b1 + k1 ∗ δ*
using *B1-def[rule-format, OF ‹x∈S›]* **by** *auto*
define *r d* **where** *r= k1 mod |k|* **and** *d = k1 div |k|*
define *b kk* **where** *b=b1+r∗δ* **and** *kk = (if k>0 then d else −d)*
have *x = b1 + (r+|k|∗d)∗δ* **using** *x-δ* **unfolding** *r-def d-def* **by** *auto*
then have *x = b + kk∗(k∗δ)* **unfolding** *b-def kk-def* **using** *‹k≠0›*
by *(auto simp add:algebra-simps)*
moreover have *b∈B*
proof −
have *r ∈ {0..<|k|}* **unfolding** *r-def* **by** *(simp add: ‹k≠0›)*
then show *?thesis* **unfolding** *b-def B-def* **using** *‹b1∈B1›* **by** *blast*
qed

```
    ultimately show ?thesis by auto
  qed
  moreover have finite B unfolding B-def using ‹finite B1›
    by (simp add: finite-image-set2)
   ultimately show periodic-set S (real-of-int k ∗ δ) unfolding periodic-set-def
by auto
next
  assume periodic-set S (real-of-int k ∗ δ)
  then show periodic-set S δ unfolding periodic-set-def
    by (metis mult.commute mult.left-commute of-int-mult)
qed

lemma periodic-set-empty[simp]: periodic-set {} δ
  unfolding periodic-set-def by auto

lemma periodic-set-finite:
  assumes finite S
  shows periodic-set S δ
unfolding periodic-set-def using assms mult.commute by force

lemma periodic-set-subset[elim]:
  assumes periodic-set S δ T ⊆ S
  shows periodic-set T δ
using assms unfolding periodic-set-def by (meson subsetCE)

lemma periodic-set-union:
  assumes periodic-set S δ periodic-set T δ
  shows periodic-set (S ∪ T) δ
using assms unfolding periodic-set-def by (metis Un-iff infinite-Un)

lemma periodic-imp-uniform-discrete:
  assumes periodic-set S δ
  shows uniform-discrete S
proof −
  have ?thesis when S≠{} δ≠0
  proof −
    obtain B g where finite B and g-def:∀x∈S. g x∈B ∧ (∃k::int. x = g x + k
∗ δ)
      using assms unfolding periodic-set-def by metis
    define P where P = ((∗) δ) ‘ Ints
    define B-diff where B-diff = {|x−y| | x y.  x∈B ∧ y∈B} − P
    have finite B-diff unfolding B-diff-def using ‹finite B›
      by (simp add: finite-image-set2)
   define e where e = (if setdist B-diff P = 0 then |δ| else min (setdist B-diff P)
(|δ|))
    have e>0
      unfolding e-def using setdist-pos-le[unfolded order-class.le-less] ‹δ≠0›
      by auto
    moreover have x=y when x∈S y∈S dist x y<e for x y
```

```
    proof −
      obtain k1::int where k1:x = g x + k1 ∗ δ and g x∈B using g-def ‹x∈S›
by auto
      obtain k2::int where k2:y = g y + k2 ∗ δ and g y∈B using g-def ‹y∈S›
by auto
      have ?thesis when |g x − g y| ∈ P
      proof −
        obtain k::int where k:g x − g y = k ∗ δ
        proof −
          obtain k′ where k′∈Ints and ∗:|g x − g y| = δ ∗ k′
            using ‹|g x − g y| ∈ P› unfolding P-def image-iff by auto
          then obtain k where ∗∗:k′ = of-int k using Ints-cases by auto
          show ?thesis
            apply (cases g x − g y ≥ 0)
            subgoal using that[of k] ∗ ∗∗ by simp
            subgoal using that[of −k] ∗ ∗∗ by (auto simp add:algebra-simps)
            done
        qed
        have dist x y = |(g x − g y)+(k1−k2)∗δ|
          unfolding dist-real-def by (subst k1,subst k2,simp add:algebra-simps)
        also have ... = |(k+k1−k2)∗δ|
          by (subst k,simp add:algebra-simps)
        also have ... = |k+k1−k2|∗|δ| by (simp add: abs-mult)
        finally have ∗:dist x y = |k+k1−k2|∗|δ| .
        then have |k+k1−k2|∗|δ| < e using ‹dist x y<e› by auto
        then have |k+k1−k2|∗|δ| < |δ|
          by (simp add: e-def split: if-splits)
        then have |k+k1−k2| = 0 unfolding e-def using ‹δ≠0› by force
        then have dist x y=0 using ∗ by auto
        then show ?thesis by auto
      qed
      moreover have ?thesis when |g x − g y| ∉ P
      proof −
        have |g x − g y| ∈ B-diff unfolding B-diff-def using ‹g x∈B› ‹g y∈B›
that by auto
        have e ≤ ||g x − g y| − |(k1−k2)∗δ||
        proof −
          have |g x − g y| ∈ B-diff unfolding B-diff-def using ‹g x∈B› ‹g y∈B›
that by auto
          moreover have |(k1−k2)∗δ| ∈ P unfolding P-def
              apply (intro rev-image-eqI[of (if δ≥0 then |of-int(k1−k2)| else −
|of-int(k1−k2)|)])
            apply (metis Ints-minus Ints-of-int of-int-abs)
            by (auto simp add:abs-mult)
          ultimately have ||g x − g y| − |(k1−k2)∗δ|| ≥ setdist B-diff P
            using setdist-le-dist[of - B-diff - P] dist-real-def by auto
          moreover have setdist B-diff P ≠ 0
          proof −
            have compact B-diff using ‹finite B-diff› using finite-imp-compact by
```

```
blast
          moreover have closed P
            unfolding P-def using closed-scaling[OF closed-Ints[where 'a=real],
of δ] by auto
          moreover have P ≠ {} using Ints-0 unfolding P-def by blast
          moreover have B-diff ∩ P = {} unfolding B-diff-def by auto
            moreover have B-diff ≠{} unfolding B-diff-def using ‹g x∈B› ‹g
y∈B› that by auto
          ultimately show ?thesis using setdist-eq-0-compact-closed[of B-diff P]
by auto
        qed
        ultimately show ?thesis unfolding e-def by argo
      qed
      also have ... ≤ |(g x − g y) + (k1−k2)∗δ|
      proof −
        define t1 where t1=g x − g y
        define t2 where t2 = of-int (k1 − k2) ∗ δ
        show ?thesis
          apply (fold t1-def t2-def)
          by linarith
      qed
      also have ... = dist x y
        unfolding dist-real-def
        by (subst (2) k1,subst (2) k2,simp add:algebra-simps)
      finally have dist x y≥e .
      then have False using ‹dist x y<e› by auto
      then show ?thesis by auto
    qed
    ultimately show ?thesis by auto
  qed
  ultimately show ?thesis unfolding uniform-discrete-def by auto
 qed
 moreover have ?thesis when S={} using that by auto
 moreover have ?thesis when δ=0
 proof −
   obtain B g where finite B and g-def:∀ x∈S. g x∈B ∧ (∃ k::int. x = g x + k
∗ δ)
     using assms unfolding periodic-set-def by metis
   then have ∀ x∈S. g x∈B ∧ (x = g x) using that by fastforce
   then have S ⊆ g ‘ B by auto
   then have finite S using ‹finite B› by (auto elim:finite-subset)
   then show ?thesis using uniform-discrete-finite-iff by blast
 qed
 ultimately show ?thesis by blast
qed

lemma periodic-set-tan-linear:
 assumes a≠0 c≠0
 shows periodic-set (roots (λx. a∗tan (x/c) + b)) (c∗pi)
```

proof −
  define *B* **where** $B = \{ c*arctan\ (- b\ /\ a),\ c*pi/2\}$
  have $\exists\, b\in B.\ \exists\, k::int.\ x = b + k * (c*pi)$ **when** $x\in roots\ (\lambda x.\ a * tan\ (x/c) + b)$
for *x*
  proof −
    define *C1* **where** $C1 = (\exists\, k::int.\ x = c*arctan\ (- b\ /\ a) + k * (c*pi))$
    define *C2* **where** $C2 = (\exists\, k::int.\ x = c*pi\ /\ 2 + k\ * (c*pi) \wedge - b\ /\ a = 0)$
    have $tan\ (x/c) = - b/a$ **using** *that* ‹$a\neq 0$› **unfolding** *roots-within-def*
      by (*auto simp add*:*field-simps*)
    then have $C1 \vee C2$ **unfolding** *C1-def C2-def* **using** *tan-eq-arctan-Ex*[*of x/c*
$-b/a$] ‹$c\neq 0$›
      by (*auto simp add*:*field-simps*)
    moreover have *?thesis* **when** *C1* **using** *that* **unfolding** *C1-def B-def* **by** *blast*
    moreover have *?thesis* **when** *C2* **using** *that* **unfolding** *C2-def B-def* **by** *blast*
    ultimately show *?thesis* **by** *auto*
  qed
  moreover have *finite B* **unfolding** *B-def* **by** *auto*
  ultimately show *?thesis* **unfolding** *periodic-set-def* **by** *auto*
qed

lemma *periodic-set-cos-linear*:
  assumes $a\neq 0$ $c\neq 0$
  shows $periodic\text{-}set\ (roots\ (\lambda x.\ a*cos\ (x/c) + b))\ (2*c*pi)$
proof −
  define *B* **where** $B = \{ c*arccos\ (- b\ /\ a),\ - c*arccos\ (- b\ /\ a)\}$
  have $\exists\, b\in B.\ \exists\, k::int.\ x = b + k * (2*c*pi)$
    when $x\in roots\ (\lambda x.\ a * cos\ (x/c) + b)$ **for** *x*
  proof −
    define *C1* **where** $C1 = (\exists\, k::int.\ x = c*arccos\ (- b\ /\ a) + k * (2*c*pi))$
    define *C2* **where** $C2 = (\exists\, k::int.\ x = - c*arccos\ (- b\ /\ a) + k * (2*c*pi))$
    have $cos\ (x/c) = - b/a$ **using** *that* ‹$a\neq 0$› **unfolding** *roots-within-def*
      by (*auto simp add*:*field-simps*)
    then have $C1 \vee C2$
      unfolding *cos-eq-arccos-Ex ex-disj-distrib C1-def C2-def* **using** ‹$c\neq 0$›
      apply (*auto simp add*:*divide-simps*)
      by (*auto simp add*:*algebra-simps*)
    moreover have *?thesis* **when** *C1* **using** *that* **unfolding** *C1-def B-def* **by** *blast*
    moreover have *?thesis* **when** *C2* **using** *that* **unfolding** *C2-def B-def* **by** *blast*
    ultimately show *?thesis* **by** *auto*
  qed
  moreover have *finite B* **unfolding** *B-def* **by** *auto*
  ultimately show *?thesis* **unfolding** *periodic-set-def* **by** *auto*
qed

lemma *periodic-set-tan-poly*:
  assumes $p\neq 0$ $c\neq 0$
  shows $periodic\text{-}set\ (roots\ (\lambda x.\ poly\ p\ (tan\ (x/c))))\ (c*pi)$
  using *assms*
proof (*induct rule*:*poly-root-induct-alt*)

```
  case 0
  then show ?case by simp
next
  case (no-proots p)
  then show ?case unfolding roots-within-def by simp
next
  case (root a p)
  have roots (λx. poly ([:− a, 1:] ∗ p) (tan (x/c))) = roots (λx. tan (x/c) − a)
          ∪ roots (λx. poly p (tan (x/c)))
    unfolding roots-within-def by auto
  moreover have periodic-set (roots (λx. tan (x/c) − a)) (c∗pi)
    using periodic-set-tan-linear[OF - ‹c≠0› ,of 1 −a,simplified] .
  moreover have periodic-set (roots (λx. poly p (tan (x/c)))) (c∗pi) using root
by fastforce
  ultimately show ?case using periodic-set-union by simp
qed

lemma periodic-set-sin-cos-linear:
  fixes a b c ::real
  assumes a≠0 ∨ b≠0 ∨ c≠0
  shows periodic-set (roots (λx. a ∗ cos x + b ∗ sin x + c)) (4∗pi)
proof −
  define f where f x= a ∗ cos x + b ∗ sin x + c for x
  have roots f = (roots f ∩ {x. cos (x/2) = 0}) ∪ (roots f ∩ {x. cos (x/2) ≠ 0})
    by auto
  moreover have periodic-set (roots f ∩ {x. cos (x/2) = 0}) (4∗pi)
  proof −
    have periodic-set ({x. cos (x/2) = 0}) (4∗pi)
      using periodic-set-cos-linear[of 1 2 0,unfolded roots-within-def,simplified] by
simp
    then show ?thesis by auto
  qed
  moreover have periodic-set (roots f ∩ {x. cos (x/2) ≠ 0}) (4∗pi)
  proof −
    define p where p=[:a+c,2∗b,c−a:]
    have poly p (tan (x/2)) = 0 ⟷ f x=0 when cos (x/2) ≠0 for x
    proof −
      define t where t=tan (x/2)
      define tt where tt = 1+t^2
      have cos x = (1−t^2) / tt unfolding tt-def t-def
        using cos-tan-half[OF that,simplified] by simp
      moreover have sin x = 2∗t / tt unfolding tt-def t-def
        using sin-tan-half[of x/2,simplified] by simp
      moreover have tt≠0 unfolding tt-def
        by (metis power-one sum-power2-eq-zero-iff zero-neq-one)
      ultimately show ?thesis
        unfolding f-def p-def
        apply (fold t-def)
        apply simp
```

apply (*auto simp add*:*field-simps*)
by (*auto simp add*:*algebra-simps tt-def power2-eq-square*)
qed
then have *roots f* ∩ {*x*. *cos* (*x*/*2*) ≠ *0*} = *roots* (λ*x*. *poly p* (*tan* (*x*/*2*))) ∩ {*x*. *cos* (*x*/*2*) ≠ *0*}
unfolding *roots-within-def* **by** *auto*
moreover have *periodic-set* (*roots* (λ*x*. *poly p* (*tan* (*x*/*2*))) ∩ {*x*. *cos* (*x*/*2*) ≠ *0*}) (*4*∗*pi*)
proof −
have *p*≠*0* **unfolding** *p-def* **using** *assms* **by** *auto*
then have *periodic-set* (*roots* (λ*x*. *poly p* (*tan* (*x*/*2*)))) (*4*∗*pi*)
using *periodic-set-tan-poly*[*of p 2*,*simplified*]
periodic-set-multiple[*of 2 - 2*∗*pi*,*simplified*]
by *auto*
then show *?thesis* **by** *auto*
qed
ultimately show *?thesis* **by** *auto*
qed
ultimately show *periodic-set* (*roots f*) (*4*∗*pi*) **using** *periodic-set-union* **by** *metis*
qed

end

4 Some useful lemmas in analysis

theory *Missing-Analysis*
imports *HOL*−*Complex-Analysis*.*Complex-Analysis*
begin

4.1 More about paths

lemma *pathfinish-offset*[*simp*]:
pathfinish (λ*t*. *g t* − *z*) = *pathfinish g* − *z*
unfolding *pathfinish-def* **by** *simp*

lemma *pathstart-offset*[*simp*]:
pathstart (λ*t*. *g t* − *z*) = *pathstart g* − *z*
unfolding *pathstart-def* **by** *simp*

lemma *pathimage-offset*[*simp*]:
fixes *g* :: - ⇒ ′*b*::*topological-group-add*
shows *p* ∈ *path-image* (λ*t*. *g t* − *z*) ⟷ *p*+*z* ∈ *path-image g*
unfolding *path-image-def* **by** (*auto simp*:*algebra-simps*)

lemma *path-offset*[*simp*]:
fixes *g* :: - ⇒ ′*b*::*topological-group-add*
shows *path* (λ*t*. *g t* − *z*) ⟷ *path g*
unfolding *path-def*
proof

```
  assume continuous-on {0..1} (λt. g t − z)
  hence continuous-on {0..1} (λt. (g t − z) + z)
    using continuous-on-add continuous-on-const by blast
  then show continuous-on {0..1} g by auto
qed (auto intro:continuous-intros)

lemma not-on-circlepathI:
  assumes cmod (z−z0) ≠ |r|
  shows z ∉ path-image (part-circlepath z0 r st tt)
  using assms
  by (auto simp add: path-image-def image-def part-circlepath-def norm-mult)

lemma circlepath-inj-on:
  assumes r>0
  shows inj-on (circlepath z r) {0..<1}
proof (rule inj-onI)
  fix x y assume asm: x ∈ {0..<1} y ∈ {0..<1} circlepath z r x = circlepath z r
y
  define c where c=2 ∗ pi ∗ i
  have c≠0 unfolding c-def by auto
  from asm(3) have exp (c ∗ x) =exp (c ∗ y)
    unfolding circlepath c-def using ‹r>0› by auto
  then obtain n where c ∗ x =c ∗ (y + of-int n)
    by (auto simp add:exp-eq c-def algebra-simps)
  then have x=y+n using ‹c≠0›
    by (meson mult-cancel-left of-real-eq-iff)
  then show x=y using asm(1,2) by auto
qed
```

4.2 More lemmas related to *winding-number*

```
lemma winding-number-comp:
  assumes open s f holomorphic-on s path-image γ ⊆ s
    valid-path γ z ∉ path-image (f ∘ γ)
  shows winding-number (f ∘ γ) z = 1/(2∗pi∗i)∗ contour-integral γ (λw. deriv f
w / (f w − z))
proof −
  obtain spikes where finite spikes and γ-diff: γ C1-differentiable-on {0..1} −
spikes
   using ‹valid-path γ› unfolding valid-path-def piecewise-C1-differentiable-on-def
by auto
  have valid-path (f ∘ γ)
    using valid-path-compose-holomorphic assms by blast
  moreover have contour-integral (f ∘ γ) (λw. 1 / (w − z))
      = contour-integral γ (λw. deriv f w / (f w − z))
    unfolding contour-integral-integral
  proof (rule integral-spike[rule-format,OF negligible-finite[OF ‹finite spikes›]])
    fix t::real assume t:t ∈ {0..1} − spikes
    then have γ differentiable at t
```

using *γ-diff* **unfolding** *C1-differentiable-on-eq* **by** *auto*
moreover have *f field-differentiable at* (*γ t*)
proof −
have *γ t* ∈ *s* **using** ‹*path-image γ* ⊆ *s*› *t* **unfolding** *path-image-def* **by** *auto*
thus *?thesis*
using ‹*open s*› ‹*f holomorphic-on s*› *holomorphic-on-imp-differentiable-at*
by *blast*
qed
ultimately show *deriv f* (*γ t*) / (*f* (*γ t*) − *z*) ∗ *vector-derivative γ* (*at t*) =
1 / ((*f* ∘ *γ*) *t* − *z*) ∗ *vector-derivative* (*f* ∘ *γ*) (*at t*)
by (*simp add*: *vector-derivative-chain-at-general*)
qed
moreover note ‹*z* ∉ *path-image* (*f* ∘ *γ*)›
ultimately show *?thesis*
using *winding-number-valid-path* **by** *presburger*
qed

lemma *winding-number-uminus-comp*:
assumes *valid-path γ* − *z* ∉ *path-image γ*
shows *winding-number* (*uminus* ∘ *γ*) *z* = *winding-number γ* (−*z*)
proof −
define *c* **where** *c*= *2* ∗ *pi* ∗ i
have *winding-number* (*uminus* ∘ *γ*) *z* = *1*/*c* ∗ *contour-integral γ* (*λw. deriv uminus w* / (−*w*−*z*))
proof (*rule winding-number-comp*[*of UNIV*, *folded c-def*])
show *open UNIV uminus holomorphic-on UNIV path-image γ* ⊆ *UNIV valid-path γ*
using ‹*valid-path γ*› **by** (*auto intro*:*holomorphic-intros*)
show *z* ∉ *path-image* (*uminus* ∘ *γ*)
unfolding *path-image-compose* **using** ‹− *z* ∉ *path-image γ*› **by** *auto*
qed
also have . . . = *1*/*c* ∗ *contour-integral γ* (*λw. 1* / (*w*− (−*z*)))
by (*auto intro*!:*contour-integral-eq simp add*:*field-simps minus-divide-right*)
also have . . . = *winding-number γ* (−*z*)
using *winding-number-valid-path*[*OF* ‹*valid-path γ*› ‹− *z* ∉ *path-image γ*›,*folded c-def*]
by *simp*
finally show *?thesis* **by** *auto*
qed

lemma *winding-number-comp-linear*:
assumes *c*≠*0 valid-path γ* **and** *not-image*: (*z*−*b*)/*c* ∉ *path-image γ*
shows *winding-number* ((*λx. c*∗*x*+*b*) ∘ *γ*) *z* = *winding-number γ* ((*z*−*b*)/*c*) (**is** *?L* = *?R*)
proof −
define *cc* **where** *cc*=*1* / (*complex-of-real* (*2* ∗ *pi*) ∗ i)
define *zz* **where** *zz*=(*z*−*b*)/*c*
have *?L* = *cc* ∗ *contour-integral γ* (*λw. deriv* (*λx. c* ∗ *x* + *b*) *w* / (*c* ∗ *w* + *b* − *z*))

apply (*subst winding-number-comp*[*of UNIV*,*simplified*])
subgoal by (*auto intro*:*holomorphic-intros*)
subgoal using ‹*valid-path* γ› **.**
subgoal using *not-image* ‹$c\neq 0$› **unfolding** *path-image-compose* **by** *auto*
subgoal unfolding *cc-def* **by** *auto*
done
also have ... = *cc* ∗ *contour-integral* γ ($\lambda w.1$ / (w − *zz*))
proof −
have *deriv* ($\lambda x.$ $c * x + b$) = ($\lambda x.$ c)
by (*auto intro*:*derivative-intros*)
then show *?thesis*
unfolding *zz-def cc-def* **using** ‹$c\neq 0$›
by (*auto simp*:*field-simps*)
qed
also have ... = *winding-number* γ *zz*
using *winding-number-valid-path*[*OF* ‹*valid-path* γ› *not-image*,*folded zz-def cc-def*]
by *simp*
finally show *winding-number* (($\lambda x.$ $c * x + b$) ∘ γ) z = *winding-number* γ *zz* **.**
qed

end

5 Cauchy's index theorem

theory *Cauchy-Index-Theorem* **imports**
HOL−Complex-Analysis.Complex-Analysis
Sturm-Tarski.Sturm-Tarski
HOL−Computational-Algebra.Fundamental-Theorem-Algebra
Missing-Transcendental
Missing-Algebraic
Missing-Analysis
begin

This theory formalises Cauchy indices on the complex plane and relate them to winding numbers

5.1 Misc

lemma *atMostAtLeast-subset-convex*:
fixes C :: *real set*
assumes *convex* C
and $x \in C$ $y \in C$
shows $\{x .. y\} \subseteq C$
proof *safe*
fix z **assume** z: $z \in \{x .. y\}$
have $z \in C$ **if** ∗: $x < z$ $z < y$
proof −
let $?\mu = (y - z) / (y - x)$

have *0* ≤ *?μ* *?μ* ≤ *1*
using *assms* ∗ **by** (*auto simp*: *field-simps*)
then have *comb*: *?μ* ∗ *x* + (*1* − *?μ*) ∗ *y* ∈ *C*
using *assms iffD1*[*OF convex-alt*, *rule-format*, *of C y x ?μ*]
by (*simp add*: *algebra-simps*)
have *?μ* ∗ *x* + (*1* − *?μ*) ∗ *y* = (*y* − *z*) ∗ *x* / (*y* − *x*) + (*1* − (*y* − *z*) / (*y* − *x*)) ∗ *y*
by (*auto simp*: *field-simps*)
also have ... = ((*y* − *z*) ∗ *x* + (*y* − *x* − (*y* − *z*)) ∗ *y*) / (*y* − *x*)
using ∗ **by** (*simp only*: *add-divide-distrib*) (*auto simp*: *field-simps*)
also have ... = *z*
using *assms* ∗ **by** (*auto simp*: *field-simps*)
finally show *?thesis*
using *comb* **by** *auto*
qed
then show *z* ∈ *C*
using *z assms* **by** (*auto simp*: *le-less*)
qed

lemma *arg-elim*:
f x ⟹ *x*= *y* ⟹ *f y*
by *auto*

lemma *arg-elim2*:
f x1 x2 ⟹ *x1*= *y1* ⟹*x2*=*y2* ⟹ *f y1 y2*
by *auto*

lemma *arg-elim3*:
⟦*f x1 x2 x3*;*x1*= *y1*;*x2*=*y2*;*x3*=*y3* ⟧ ⟹ *f y1 y2 y3*
by *auto*

lemma *IVT-strict*:
fixes *f* :: ′*a*::*linear-continuum-topology* ⇒ ′*b*::*linorder-topology*
assumes (*f a* > *y* ∧ *y* > *f b*) ∨ (*f a* < *y* ∧ *y* < *f b*) *a*<*b continuous-on* {*a* .. *b*} *f*
shows ∃ *x*. *a* < *x* ∧ *x* < *b* ∧ *f x* = *y*
by (*metis IVT′ IVT2′ assms*(*1*) *assms*(*2*) *assms*(*3*) *linorder-neq-iff order-le-less order-less-imp-le*)

lemma (**in** *dense-linorder*) *atLeastAtMost-subseteq-greaterThanLessThan-iff*:
{*a* .. *b*} ⊆ { *c* <..< *d* } ⟷ (*a* ≤ *b* ⟶ *c* < *a* ∧ *b* < *d*)
using *dense*[*of a min c b*] *dense*[*of max a d b*]
by (*force simp*: *subset-eq Ball-def not-less*[*symmetric*])

lemma *Re-winding-number-half-right*:
assumes ∀ *p*∈*path-image* γ. *Re p* ≥ *Re z* **and** *valid-path* γ **and** *z*∉*path-image* γ
shows *Re*(*winding-number* γ *z*) = (*Im* (*Ln* (*pathfinish* γ − *z*)) − *Im* (*Ln* (*pathstart* γ − *z*)))/(*2*∗*pi*)
proof −
define *g* **where** *g*=(λ*t*. γ *t* − *z*)

define *st fi* **where** *st≡pathstart g* **and** *fi≡pathfinish g*
have *valid-path g 0∉path-image g* **and** *pos-img*:∀ *p∈path-image g. Re p ≥ 0*
unfolding *g-def*
subgoal using *assms*(*2*) **by** *auto*
subgoal using *assms*(*3*) **by** *auto*
subgoal using *assms*(*1*) **by** *fastforce*
done
have (*inverse has-contour-integral Ln fi* − *Ln st*) *g*
unfolding *fi-def st-def*
proof (*rule contour-integral-primitive*[*OF* - ‹*valid-path g*›,*of* − $\mathbb{R}_{\leq 0}$])
fix *x*::*complex* **assume** *x* ∈ − $\mathbb{R}_{\leq 0}$
then have (*Ln has-field-derivative inverse x*) (*at x*) **using** *has-field-derivative-Ln*
by *auto*
then show (*Ln has-field-derivative inverse x*) (*at x within* − $\mathbb{R}_{\leq 0}$)
using *has-field-derivative-at-within* **by** *auto*
next
show *path-image g* ⊆ − $\mathbb{R}_{\leq 0}$ **using** *pos-img* ‹*0∉path-image g*›
by (*metis ComplI antisym assms*(*3*) *complex-nonpos-Reals-iff complex-surj subsetI zero-complex.code*)
qed
then have *winding-eq*:*2*∗*pi*∗i∗*winding-number g 0* = (*Ln fi* − *Ln st*)
using *has-contour-integral-winding-number*[*OF* ‹*valid-path g*› ‹*0∉path-image g*›
,*simplified*,*folded inverse-eq-divide*] *has-contour-integral-unique*
by *auto*
have *Re*(*winding-number g 0*)
= (*Im* (*Ln fi*) − *Im* (*Ln st*))/(*2*∗*pi*)
(**is** *?L*=*?R*)
proof −
have *?L* = *Re*((*Ln fi* − *Ln st*)/(*2*∗*pi*∗i))
using *winding-eq*[*symmetric*] **by** *auto*
also have ... = *?R*
by (*metis Im-divide-of-real Im-i-times complex-i-not-zero minus-complex.simps*(*2*)

mult.commute mult-divide-mult-cancel-left-if times-divide-eq-right)
finally show *?thesis* **.**
qed
then show *?thesis* **unfolding** *g-def fi-def st-def* **using** *winding-number-offset*
by *simp*
qed

lemma *Re-winding-number-half-upper*:
assumes *pimage*:∀ *p∈path-image γ. Im p ≥ Im z* **and** *valid-path γ* **and** *z∉path-image γ*
shows *Re*(*winding-number γ z*) =
(*Im* (*Ln* (i∗*z* − i∗*pathfinish γ*)) − *Im* (*Ln* (i∗*z* − i∗*pathstart γ*)))/(*2*∗*pi*)
proof −
define *γ′* **where** *γ′*=(*λt.* − i ∗ (*γ t* − *z*) + *z*)
have *Re* (*winding-number γ′ z*) = (*Im* (*Ln* (*pathfinish γ′* − *z*)) − *Im* (*Ln* (*pathstart γ′* − *z*))) / (*2* ∗ *pi*)

unfolding *γ′-def*
apply (*rule Re-winding-number-half-right*)
subgoal using *pimage* **unfolding** *path-image-def* **by** *auto*
subgoal
apply (*rule valid-path-compose-holomorphic*[*OF* ‹*valid-path γ*›,*of λx.* −i ∗ (*x*−*z*) + *z UNIV*
, *unfolded comp-def*])
by (*auto intro*!:*holomorphic-intros*)
subgoal using ‹*z*∉*path-image γ*› **unfolding** *path-image-def* **by** *auto*
done
moreover have *winding-number γ′ z* = *winding-number γ z*
proof −
define *f* **where** *f*=(*λx.* −i ∗ (*x*−*z*) + *z*)
define *c* **where** *c*= *1* / (*complex-of-real* (*2* ∗ *pi*) ∗ i)
have *winding-number γ′ z* = *winding-number* (*f o γ*) *z*
unfolding *γ′-def comp-def f-def* **by** *auto*
also have ... = *c* ∗ *contour-integral γ* (*λw. deriv f w* / (*f w* − *z*)) **unfolding** *c-def*
proof (*rule winding-number-comp*[*of UNIV*])
show *z* ∉ *path-image* (*f* ∘ *γ*) **using** ‹*z*∉*path-image γ*› **unfolding** *f-def path-image-def* **by** *auto*
qed (*auto simp add*:*f-def* ‹*valid-path γ*› *intro*!:*holomorphic-intros*)
also have ... = *c* ∗ *contour-integral γ* (*λw. 1* / (*w* − *z*))
proof −
have *deriv f x* = −i **for** *x*
unfolding *f-def*
by (*auto intro*!:*derivative-eq-intros DERIV-imp-deriv*)
then show *?thesis*
unfolding *f-def c-def*
by (*auto simp add*:*field-simps divide-simps intro*!:*arg-cong2*[**where** *f*=*contour-integral*])
qed
also have ... = *winding-number γ z*
using *winding-number-valid-path*[*OF* ‹*valid-path γ*› ‹*z*∉*path-image γ*›,*folded c-def*] **by** *simp*
finally show *?thesis* **.**
qed
moreover have *pathfinish γ′* = *z*+ i∗*z* −i∗ *pathfinish γ pathstart γ′* = *z*+ i∗*z* −i∗*pathstart γ*
unfolding *γ′-def path-defs* **by** (*auto simp add*:*algebra-simps*)
ultimately show *?thesis* **by** *auto*
qed

lemma *Re-winding-number-half-lower*:
assumes *pimage*:∀ *p*∈*path-image γ*. *Im p* ≤ *Im z* **and** *valid-path γ* **and** *z*∉*path-image γ*
shows *Re*(*winding-number γ z*) =
(*Im* (*Ln* (i∗*pathfinish γ* − i∗*z*)) − *Im* (*Ln* (i∗*pathstart γ* − i∗*z*)))/(*2*∗*pi*)
proof −
define *γ′* **where** *γ′*=(*λt.* i ∗ (*γ t* − *z*) + *z*)

have *Re* (*winding-number* γ' *z*) = (*Im* (*Ln* (*pathfinish* γ' − *z*)) − *Im* (*Ln* (*pathstart* γ' − *z*))) / (*2* ∗ *pi*)
unfolding γ'-*def*
apply (*rule Re-winding-number-half-right*)
subgoal using *pimage* **unfolding** *path-image-def* **by** *auto*
subgoal
apply (*rule valid-path-compose-holomorphic*[*OF* ‹*valid-path* γ›,*of* λx. i ∗ (*x*−*z*) + *z UNIV*
, *unfolded comp-def*])
by (*auto intro*!:*holomorphic-intros*)
subgoal using ‹*z*∉*path-image* γ› **unfolding** *path-image-def* **by** *auto*
done
moreover have *winding-number* γ' *z* = *winding-number* γ *z*
proof −
define *f* **where** *f*=(λx. i ∗ (*x*−*z*) + *z*)
define *c* **where** *c*= *1* / (*complex-of-real* (*2* ∗ *pi*) ∗ i)
have *winding-number* γ' *z* = *winding-number* (*f o* γ) *z*
unfolding γ'-*def comp-def f-def* **by** *auto*
also have ... = *c* ∗ *contour-integral* γ (λw. *deriv f w* / (*f w* − *z*)) **unfolding** *c-def*
proof (*rule winding-number-comp*[*of UNIV*])
show *z* ∉ *path-image* (*f* ∘ γ) **using** ‹*z*∉*path-image* γ› **unfolding** *f-def path-image-def* **by** *auto*
qed (*auto simp add*:*f-def* ‹*valid-path* γ› *intro*!:*holomorphic-intros*)
also have ... = *c* ∗ *contour-integral* γ (λw. *1* / (*w* − *z*))
proof −
have *deriv f x* = i **for** *x*
unfolding *f-def*
by (*auto intro*!:*derivative-eq-intros DERIV-imp-deriv*)
then show *?thesis*
unfolding *f-def c-def*
by (*auto simp add*:*field-simps divide-simps intro*!:*arg-cong2*[**where** *f*=*contour-integral*])
qed
also have ... = *winding-number* γ *z*
using *winding-number-valid-path*[*OF* ‹*valid-path* γ› ‹*z*∉*path-image* γ›,*folded c-def*] **by** *simp*
finally show *?thesis* **.**
qed
moreover have *pathfinish* γ' = *z*+ i∗ *pathfinish* γ − i∗*z pathstart* γ' = *z*+ i∗*pathstart* γ − i∗*z*
unfolding γ'-*def path-defs* **by** (*auto simp add*:*algebra-simps*)
ultimately show *?thesis* **by** *auto*
qed

lemma *Re-winding-number-half-left*:
assumes *neg-img*:∀ *p*∈*path-image* γ. *Re p* ≤ *Re z* **and** *valid-path* γ **and** *z*∉*path-image* γ
shows *Re*(*winding-number* γ *z*) = (*Im* (*Ln* (*z* − *pathfinish* γ)) − *Im* (*Ln* (*z* −

pathstart γ)))/(*2*∗*pi*)
proof −
  define γ' **where** $\gamma'\equiv(\lambda t.\ 2*z - \gamma\ t)$
  have *Re* (*winding-number* γ' *z*) = (*Im* (*Ln* (*pathfinish* γ' − *z*)) − *Im* (*Ln* (*pathstart* γ' − *z*))) / (*2* ∗ *pi*)
    unfolding γ'*-def*
    apply (*rule Re-winding-number-half-right*)
    subgoal using *neg-img* **unfolding** *path-image-def* **by** *auto*
    subgoal
      apply (*rule valid-path-compose-holomorphic*[*OF* ‹*valid-path* γ›,*of* $\lambda t.\ 2*z-t$ *UNIV*,
          unfolded comp-def])
      by (*auto intro*:*holomorphic-intros*)
    subgoal using ‹*z*∉*path-image* γ› **unfolding** *path-image-def* **by** *auto*
    done
  moreover have *winding-number* γ' *z* = *winding-number* γ *z*
  proof −
    define *f* **where** $f=(\lambda t.\ 2*z-t)$
    define *c* **where** *c*= *1* / (*complex-of-real* (*2* ∗ *pi*) ∗ i)
    have *winding-number* γ' *z* = *winding-number* (*f o* γ) *z*
      unfolding γ'*-def comp-def f-def* **by** *auto*
    also have ... = *c* ∗ *contour-integral* γ ($\lambda w.$ *deriv f w* / (*f w* − *z*)) **unfolding** *c-def*
    proof (*rule winding-number-comp*[*of UNIV*])
      show *z* ∉ *path-image* (*f* ∘ γ) **using** ‹*z*∉*path-image* γ› **unfolding** *f-def path-image-def* **by** *auto*
    qed (*auto simp add*:*f-def* ‹*valid-path* γ› *intro*:*holomorphic-intros*)
    also have ... = *c* ∗ *contour-integral* γ ($\lambda w.$ *1* / (*w* − *z*))
      unfolding *f-def c-def*
    by (*auto simp add*:*field-simps divide-simps intro*!:*arg-cong2*[**where** *f*=*contour-integral*])
    also have ... = *winding-number* γ *z*
      using *winding-number-valid-path*[*OF* ‹*valid-path* γ› ‹*z*∉*path-image* γ›,*folded c-def*] **by** *simp*
    finally show *?thesis* **.**
  qed
  moreover have *pathfinish* γ' = *2*∗*z* − *pathfinish* γ *pathstart* γ' = *2*∗*z* − *pathstart* γ
    unfolding γ'*-def path-defs* **by** *auto*
  ultimately show *?thesis* **by** *auto*
qed

lemma *continuous-on-open-Collect-neq*:
  fixes *f g* :: *'a*::*topological-space* ⇒ *'b*::*t2-space*
  assumes *f*: *continuous-on S f* **and** *g*: *continuous-on S g* **and** *open S*
  shows *open* {*x*∈*S*. *f x* ≠ *g x*}
proof (*rule topological-space-class.openI*)
  fix *t*
  assume *t* ∈ {*x*∈*S*. *f x* ≠ *g x*}
  then obtain *U0 V0* **where** *open U0 open V0 f t* ∈ *U0 g t* ∈ *V0 U0* ∩ *V0* = {}

t∈*S*
  by (*auto simp add*: *separation-t2*)
 obtain *U1* **where** *open U1 t* ∈ *U1* ∀ *y*∈(*S* ∩ *U1*). *f y* ∈ *U0*
  using *f*[*unfolded continuous-on-topological*,*rule-format*,*OF* ‹*t*∈*S*› ‹*open U0*› ‹*f t* ∈*U0*›] **by** *auto*
 obtain *V1* **where** *open V1 t* ∈ *V1* ∀ *y*∈(*S* ∩ *V1*). *g y* ∈ *V0*
  using *g*[*unfolded continuous-on-topological*,*rule-format*,*OF* ‹*t*∈*S*› ‹*open V0*› ‹*g t* ∈*V0*›] **by** *auto*
 define *T* **where** *T*=*V1* ∩ *U1* ∩ *S*
 have *open T* **unfolding** *T-def* **using** ‹*open U1*› ‹*open V1*› ‹*open S*› **by** *auto*
 moreover have *t* ∈ *T* **unfolding** *T-def* **using** ‹*t*∈*U1*› ‹*t*∈*V1*› ‹*t*∈*S*› **by** *auto*
 moreover have *T* ⊆ {*x* ∈ *S*. *f x* ≠ *g x*} **unfolding** *T-def*
  using ‹*U0* ∩ *V0* = {}› ‹∀ *y*∈*S* ∩ *U1*. *f y* ∈ *U0*› ‹∀ *y*∈*S* ∩ *V1*. *g y* ∈ *V0*› **by** *auto*
 ultimately show ∃ *T*. *open T* ∧ *t* ∈ *T* ∧ *T* ⊆ {*x* ∈ *S*. *f x* ≠ *g x*} **by** *auto*
qed

5.2 Sign at a filter

definition *has-sgnx*::(*real* ⇒ *real*) ⇒ *real* ⇒ *real filter* ⇒ *bool*
  (**infixr** ‹*has′-sgnx*› *55*) **where**
 (*f has-sgnx c*) *F*= (*eventually* (λ*x*. *sgn*(*f x*) = *c*) *F*)

definition *sgnx-able* (**infixr** ‹*sgnx′-able*› *55*) **where**
 (*f sgnx-able F*) = (∃ *c*. (*f has-sgnx c*) *F*)

definition *sgnx* **where**
 sgnx f F = (*SOME c*. (*f has-sgnx c*) *F*)

lemma *has-sgnx-eq-rhs*: (*f has-sgnx x*) *F* ⟹ *x* = *y* ⟹ (*f has-sgnx y*) *F*
 by *simp*

named-theorems *sgnx-intros introduction rules for has-sgnx*
setup ‹
 Global-Theory.add-thms-dynamic (@{*binding sgnx-eq-intros*},
  fn context =>
   Named-Theorems.get (*Context.proof-of context*) @{*named-theorems sgnx-intros*}
    |> *map-filter* (*try* (*fn thm* => @{*thm has-sgnx-eq-rhs*} *OF* [*thm*])))
›

lemma *sgnx-able-sgnx*:*f sgnx-able F* ⟹ (*f has-sgnx* (*sgnx f F*)) *F*
 unfolding *sgnx-able-def sgnx-def* **using** *someI-ex* **by** *metis*

lemma *has-sgnx-imp-sgnx-able*[*elim*]:
 (*f has-sgnx c*) *F* ⟹ *f sgnx-able F*
unfolding *sgnx-able-def* **by** *auto*

lemma *has-sgnx-unique*:
 assumes *F*≠*bot* (*f has-sgnx c1*) *F* (*f has-sgnx c2*) *F*

```
  shows c1=c2
proof (rule ccontr)
  assume c1 ≠ c2
  have eventually (λx. sgn(f x) = c1 ∧ sgn(f x) = c2) F
    using assms  unfolding has-sgnx-def eventually-conj-iff by simp
  then have eventually (λ-. c1 = c2) F by (elim eventually-mono,auto)
  then have eventually (λ-. False) F using ‹c1 ≠ c2› by auto
  then show False using ‹F ≠ bot› eventually-False by auto
qed

lemma has-sgnx-imp-sgnx[elim]:
  (f has-sgnx c) F ⟹F≠bot ⟹ sgnx f F = c
  using has-sgnx-unique sgnx-def by auto

lemma has-sgnx-const[simp,sgnx-intros]:
  ((λ-. c) has-sgnx sgn c) F
by (simp add: has-sgnx-def)

lemma finite-sgnx-at-left-at-right:
  assumes finite {t. f t=0 ∧ a<t ∧ t<b} continuous-on ({a<..<b} − s) f finite s
      and x:x∈{a<..<b}
  shows f sgnx-able (at-left x) sgnx f (at-left x)≠0
        f sgnx-able (at-right x) sgnx f (at-right x)≠0
proof −
  define ls where ls ≡ {t. (f t=0 ∨ t∈s) ∧ a<t ∧t<x }
  define l where l≡(if ls = {} then (a+x)/2 else (Max ls + x)/2)
  have finite ls
  proof −
    have {t. f t=0 ∧ a<t ∧ t<x} ⊆ {t. f t=0 ∧ a<t ∧ t<b} using x by auto
    then have finite {t. f t=0 ∧ a<t ∧ t<x} using assms(1)
      using finite-subset by blast
    moreover have finite {t. t∈s ∧ a<t ∧ t<x} using assms(3) by auto
    moreover have ls = {t. f t=0 ∧ a<t ∧ t<x} ∪ {t. t∈s ∧ a<t ∧ t<x}
      unfolding ls-def by auto
    ultimately show ?thesis by auto
  qed
  have [simp]: l<x a<l l<b
  proof −
    have l<x ∧ a<l ∧ l<b when ls = {}
      using that x unfolding l-def by auto
    moreover have l<x ∧ a<l ∧ l<b when ls ≠{}
    proof −
      have Max ls ∈ ls using assms(1,3) that ‹finite ls›
        apply (intro linorder-class.Max-in)
        by auto
      then have a<Max ls ∧ Max ls < x unfolding ls-def by auto
      then show ?thesis unfolding l-def using that x by auto
    qed
    ultimately show l<x a<l l<b by auto
```

```
  qed
  have noroot:f t≠0 when t:t∈{l..<x} for t
  proof (cases ls = {})
    case True
    have False when f t=0
    proof -
      have t>a using t ‹l>a› by (meson atLeastLessThan-iff less-le-trans)
      then have t∈ls using that t unfolding ls-def by auto
      then show False using True by auto
    qed
    then show ?thesis by auto
  next
    case False
    have t>Max ls using that False ‹l<x› unfolding l-def by auto
    have False when f t=0
    proof -
      have t>a using t ‹l>a› by (meson atLeastLessThan-iff less-le-trans)
      then have t∈ls using that t unfolding ls-def by auto
      then have t≤Max ls using ‹finite ls› by auto
      then show False using ‹t>Max ls› by auto
    qed
    then show ?thesis by auto
  qed
  have (f has-sgnx sgn (f l)) (at-left x) unfolding has-sgnx-def
  proof (rule eventually-at-leftI[OF - ‹l<x›])
    fix t assume t:t∈{l<..<x}
    then have [simp]:t>a t<b using ‹l>a› x
      by (meson greaterThanLessThan-iff less-trans)+
    have False when f t = 0
      using noroot t that by auto
    moreover have False when f l=0
      using noroot t that by auto
    moreover have False when f l>0 ∧ f t<0 ∨ f l <0 ∧ f t >0
    proof -
      have False when {l..t} ∩ s ≠{}
      proof -
        obtain t' where t':t'∈{l..t} t'∈s
          using ‹{l..t} ∩ s ≠ {}› by blast
        then have a<t' ∧ t'<x
        by (metis ‹a < l› atLeastAtMost-iff greaterThanLessThan-iff le-less less-trans
t)
        then have t'∈ls unfolding ls-def using ‹t'∈s› by auto
        then have t'≤Max ls using ‹finite ls› by auto
        moreover have Max ls<l
          using ‹l<x› ‹t'∈ls› ‹finite ls› unfolding l-def by (auto simp add:ls-def )
        ultimately show False using t'(1) by auto
      qed
      moreover have {l..t} ⊆ {a<..<b}
        by (intro atMostAtLeast-subset-convex,auto)
```

```
      ultimately have continuous-on {l..t} f using assms(2)
        by (elim continuous-on-subset,auto)
      then have ∃x>l. x < t ∧ f x = 0
        apply (intro IVT-strict)
        using that t assms(2) by auto
      then obtain t′ where l<t′ t′<t f t′=0 by auto
      then have t′∈{l..<x} unfolding ls-def using t by auto
      then show False using noroot ‹f t′=0› by auto
    qed
    ultimately show sgn (f t) = sgn (f l)
      by (metis le-less not-less sgn-if)
  qed
  then show f sgnx-able (at-left x) by auto
  show sgnx f (at-left x)≠0
    using noroot[of l,simplified] ‹(f has-sgnx sgn (f l)) (at-left x)›
    by (simp add: has-sgnx-imp-sgnx sgn-if)
next
  define rs where rs ≡ {t. (f t=0 ∨ t∈s) ∧ x<t ∧ t<b}
  define r where r≡(if rs = {} then (x+b)/2 else (Min rs + x)/2)
  have finite rs
  proof −
    have {t. f t=0 ∧ x<t ∧ t<b} ⊆ {t. f t=0 ∧ a<t ∧ t<b} using x by auto
    then have finite {t. f t=0 ∧ x<t ∧ t<b} using assms(1)
      using finite-subset by blast
    moreover have finite {t. t∈s ∧ x<t ∧ t<b} using assms(3) by auto
    moreover have rs = {t. f t=0 ∧ x<t ∧ t<b} ∪ {t. t∈s ∧ x<t ∧ t<b}
      unfolding rs-def by auto
    ultimately show ?thesis by auto
  qed

  have [simp]: r>x a<r r<b
  proof −
    have r>x ∧ a<r ∧ r<b when rs = {}
      using that x unfolding r-def by auto
    moreover have r>x ∧ a<r ∧ r<b when rs ≠{}
    proof −
      have Min rs ∈ rs using assms(1,3) that ‹finite rs›
        apply (intro linorder-class.Min-in)
        by auto
      then have x<Min rs ∧ Min rs < b unfolding rs-def by auto
      then show ?thesis unfolding r-def using that x by auto
    qed
    ultimately show r>x a<r r<b by auto
  qed
  have noroot:f t≠0 when t:t∈{x<..r} for t
  proof (cases rs = {})
    case True
    have False when f t=0
    proof −
```

```
      have t<b using t ‹r<b›
        using greaterThanAtMost-iff by fastforce
      then have t∈rs using that t unfolding rs-def by auto
      then show False using True by auto
    qed
    then show ?thesis by auto
  next
    case False
    have t<Min rs using that False ‹r>x› unfolding r-def by auto
    have False when f t=0
    proof –
      have t<b using t ‹r<b› by (metis greaterThanAtMost-iff le-less less-trans)
      then have t∈rs using that t unfolding rs-def by auto
      then have t≥Min rs using ‹finite rs› by auto
      then show False using ‹t<Min rs› by auto
    qed
    then show ?thesis by auto
  qed
  have (f has-sgnx sgn (f r)) (at-right x) unfolding has-sgnx-def
  proof (rule eventually-at-rightI[OF - ‹r>x›])
    fix t assume t:t∈{x<..<r}
    then have [simp]:t>a t<b using ‹r<b› x
      by (meson greaterThanLessThan-iff less-trans)+
    have False when f t = 0
      using noroot t that by auto
    moreover have False when f r=0
      using noroot t that by auto
    moreover have False when f r>0 ∧ f t<0 ∨ f r <0 ∧ f t >0
    proof –
      have False when {t..r} ∩ s ≠{}
      proof –
        obtain t' where t':t'∈{t..r} t'∈s
          using ‹{t..r} ∩ s ≠ {}› by blast
        then have x<t' ∧ t'<b
         by (meson ‹r < b› atLeastAtMost-iff greaterThanLessThan-iff less-le-trans
not-le t)
        then have t'∈rs unfolding rs-def using t ‹t'∈s› by auto
        then have t'≥Min rs using ‹finite rs› by auto
        moreover have Min rs>r
          using ‹r>x› ‹t'∈rs› ‹finite rs› unfolding r-def by (auto simp add:rs-def
)
        ultimately show False using t'(1) by auto
      qed
      moreover have {t..r} ⊆ {a<..<b}
        by (intro atMostAtLeast-subset-convex,auto)
       ultimately have continuous-on {t..r} f using assms(2) by (elim continu-
ous-on-subset,auto)
      then have ∃ x>t. x < r ∧ f x = 0
        apply (intro IVT-strict)
```

```
      using that t assms(2) by auto
    then obtain t′ where t<t′ t′<r f t′=0 by auto
    then have t′∈{x<..r} unfolding rs-def using t by auto
    then show False using noroot ‹f t′=0› by auto
  qed
  ultimately show sgn (f t) = sgn (f r)
    by (metis le-less not-less sgn-if)
 qed
 then show f sgnx-able (at-right x) by auto
 show sgnx f (at-right x)≠0
  using noroot[of r,simplified] ‹(f has-sgnx sgn (f r)) (at-right x)›
  by (simp add: has-sgnx-imp-sgnx sgn-if)
qed

lemma sgnx-able-poly[simp]:
 (poly p) sgnx-able (at-right a)
 (poly p) sgnx-able (at-left a)
 (poly p) sgnx-able at-top
 (poly p) sgnx-able at-bot
proof −
 show (poly p) sgnx-able at-top
  using has-sgnx-def poly-sgn-eventually-at-top sgnx-able-def by blast
 show (poly p) sgnx-able at-bot
  using has-sgnx-def poly-sgn-eventually-at-bot sgnx-able-def by blast
 show (poly p) sgnx-able (at-right a)
 proof (cases p=0)
  case True
  then show ?thesis unfolding sgnx-able-def has-sgnx-def eventually-at-right
   using linordered-field-no-ub by force
 next
  case False
  obtain ub where ub>a and ub:∀ z. a<z∧z≤ub⟶poly p z≠0
   using next-non-root-interval[OF False] by auto
  have ∀ z. a<z∧z≤ub⟶sgn(poly p z) = sgn (poly p ub)
  proof (rule ccontr)
   assume ¬ (∀ z. a < z ∧ z ≤ ub ⟶ sgn (poly p z) = sgn (poly p ub))
   then obtain z where a<z z≤ub sgn(poly p z) ≠ sgn (poly p ub) by auto
    moreover then have poly p z≠0 poly p ub≠0 z≠ub using ub ‹ub>a› by
blast+
   ultimately have (poly p z>0 ∧ poly p ub<0) ∨ (poly p z<0 ∧ poly p ub>0)
    by (metis linorder-neqE-linordered-idom sgn-neg sgn-pos)
   then have ∃ x>z. x < ub ∧ poly p x = 0
     using poly-IVT-neg[of z ub p] poly-IVT-pos[of z ub p] ‹z≤ub› ‹z≠ub› by
argo
   then show False using ub ‹a < z› by auto
  qed
  then show ?thesis unfolding sgnx-able-def has-sgnx-def eventually-at-right
   apply (rule-tac exI[where x=sgn(poly p ub)])
   apply (rule-tac exI[where x=ub])
```

```
      using less-eq-real-def ‹ub>a› by blast
  qed
  show (poly p) sgnx-able (at-left a)
  proof (cases p=0)
    case True
    then show ?thesis unfolding sgnx-able-def has-sgnx-def eventually-at-right
      using linordered-field-no-ub by force
  next
    case False
    obtain lb where lb<a and ub:∀ z. lb≤z∧z<a⟶poly p z≠0
      using last-non-root-interval[OF False] by auto
    have ∀ z. lb≤z∧z<a⟶sgn(poly p z) = sgn (poly p lb)
    proof (rule ccontr)
      assume ¬ (∀ z. lb≤z∧z<a ⟶ sgn (poly p z) = sgn (poly p lb))
      then obtain z where lb≤z z<a sgn(poly p z) ≠ sgn (poly p lb) by auto
     moreover then have poly p z≠0 poly p lb≠0 z≠lb using ub ‹lb<a› by blast+
      ultimately have (poly p z>0 ∧ poly p lb<0) ∨ (poly p z<0 ∧ poly p lb>0)
        by (metis linorder-neqE-linordered-idom sgn-neg sgn-pos)
      then have ∃ x>lb. x < z ∧ poly p x = 0
        using poly-IVT-neg[of lb z p] poly-IVT-pos[of lb z p] ‹lb≤z› ‹z≠lb› by argo
      then show False using ub ‹z < a› by auto
    qed
    then show ?thesis unfolding sgnx-able-def has-sgnx-def eventually-at-left
      apply (rule-tac exI[where x=sgn(poly p lb)])
      apply (rule-tac exI[where x=lb])
      using less-eq-real-def ‹lb<a› by blast
  qed
qed

lemma has-sgnx-identity[intro,sgnx-intros]:
  shows x≥0 ⟹((λx. x) has-sgnx 1) (at-right x)
        x≤0 ⟹ ((λx. x) has-sgnx −1) (at-left x)
proof −
  show x≥0 ⟹ ((λx. x) has-sgnx 1) (at-right x)
    unfolding has-sgnx-def eventually-at-right
    apply (intro exI[where x=x+1])
    by auto
  show x≤0 ⟹ ((λx. x) has-sgnx −1) (at-left x)
    unfolding has-sgnx-def eventually-at-left
    apply (intro exI[where x=x−1])
    by auto
qed

lemma has-sgnx-divide[sgnx-intros]:
  assumes (f has-sgnx c1) F (g has-sgnx c2) F
  shows ((λx. f x / g x) has-sgnx c1 / c2) F
proof −
  have ∀F x in F. sgn (f x) = c1 ∧ sgn (g x) = c2
    using assms unfolding has-sgnx-def by (intro eventually-conj,auto)
```

then have ∀$_F$ *x in F. sgn* (*f x* / *g x*) = *c1* / *c2*
apply (*elim eventually-mono*)
by (*simp add*: *sgn-mult sgn-divide*)
then show ((λ*x*. *f x* / *g x*) *has-sgnx c1* / *c2*) *F* **unfolding** *has-sgnx-def* **by** *auto*
qed

lemma *sgnx-able-divide*[*sgnx-intros*]:
assumes *f sgnx-able F g sgnx-able F*
shows (λ*x*. *f x* / *g x*) *sgnx-able F*
using *has-sgnx-divide* **by** (*meson assms*(*1*) *assms*(*2*) *sgnx-able-def*)

lemma *sgnx-divide*:
assumes *F*≠*bot f sgnx-able F g sgnx-able F*
shows *sgnx* (λ*x*. *f x* / *g x*) *F* =*sgnx f F* / *sgnx g F*
proof −
obtain *c1 c2* **where** *c1*:(*f has-sgnx c1*) *F* **and** *c2*:(*g has-sgnx c2*) *F*
using *assms* **unfolding** *sgnx-able-def* **by** *auto*
have *sgnx f F*=*c1 sgnx g F*=*c2* **using** *c1 c2* ‹*F*≠*bot*› **by** *auto*
moreover have ((λ*x*. *f x* / *g x*) *has-sgnx c1* / *c2*) *F*
using *has-sgnx-divide*[*OF c1 c2*] **.**
ultimately show *?thesis* **using** *assms*(*1*) *has-sgnx-imp-sgnx* **by** *blast*
qed

lemma *has-sgnx-times*[*sgnx-intros*]:
assumes (*f has-sgnx c1*) *F* (*g has-sgnx c2*) *F*
shows ((λ*x*. *f x*∗ *g x*) *has-sgnx c1* ∗ *c2*) *F*
proof −
have ∀$_F$ *x in F. sgn* (*f x*) = *c1* ∧ *sgn* (*g x*) = *c2*
using *assms* **unfolding** *has-sgnx-def* **by** (*intro eventually-conj*,*auto*)
then have ∀$_F$ *x in F. sgn* (*f x* ∗ *g x*) = *c1* ∗ *c2*
apply (*elim eventually-mono*)
by (*simp add*: *sgn-mult*)
then show ((λ*x*. *f x*∗ *g x*) *has-sgnx c1* ∗ *c2*) *F* **unfolding** *has-sgnx-def* **by** *auto*
qed

lemma *sgnx-able-times*[*sgnx-intros*]:
assumes *f sgnx-able F g sgnx-able F*
shows (λ*x*. *f x* ∗ *g x*) *sgnx-able F*
using *has-sgnx-times* **by** (*meson assms*(*1*) *assms*(*2*) *sgnx-able-def*)

lemma *sgnx-times*:
assumes *F*≠*bot f sgnx-able F g sgnx-able F*
shows *sgnx* (λ*x*. *f x* ∗ *g x*) *F* =*sgnx f F* ∗ *sgnx g F*
proof −
obtain *c1 c2* **where** *c1*:(*f has-sgnx c1*) *F* **and** *c2*:(*g has-sgnx c2*) *F*
using *assms* **unfolding** *sgnx-able-def* **by** *auto*
have *sgnx f F*=*c1 sgnx g F*=*c2* **using** *c1 c2* ‹*F*≠*bot*› **by** *auto*
moreover have ((λ*x*. *f x*∗ *g x*) *has-sgnx c1* ∗ *c2*) *F*
using *has-sgnx-times*[*OF c1 c2*] **.**

```
  ultimately show ?thesis using assms(1) has-sgnx-imp-sgnx by blast
qed

lemma tendsto-nonzero-has-sgnx:
  assumes (f ⟶ c) F c≠0
  shows (f has-sgnx sgn c) F
proof (cases rule:linorder-cases[of c 0])
  case less
  then have ∀F x in F. f x<0
    using order-topology-class.order-tendstoD[OF assms(1),of 0] by auto
  then show ?thesis
    unfolding has-sgnx-def
    apply (elim eventually-mono)
    using less by auto
next
  case equal
  then show ?thesis using ‹c≠0› by auto
next
  case greater
  then have ∀F x in F. f x>0
    using order-topology-class.order-tendstoD[OF assms(1),of 0] by auto
  then show ?thesis
    unfolding has-sgnx-def
    apply (elim eventually-mono)
    using greater by auto
qed

lemma tendsto-nonzero-sgnx:
  assumes (f ⟶ c) F F≠bot c≠0
  shows sgnx f F = sgn c
  using tendsto-nonzero-has-sgnx
by (simp add: assms has-sgnx-imp-sgnx)


lemma filterlim-divide-at-bot-at-top-iff:
  assumes (f ⟶ c) F c≠0
  shows
    (LIM x F. f x / g x :> at-bot) ⟷ (g ⟶ 0) F
      ∧ ((λx. g x) has-sgnx − sgn c) F
    (LIM x F. f x / g x :> at-top) ⟷ (g ⟶ 0) F
      ∧ ((λx. g x) has-sgnx sgn c) F
proof −
  show (LIM x F. f x / g x :> at-bot) ⟷ ((g ⟶ 0) F )
    ∧ ((λx. g x) has-sgnx − sgn c) F
  proof
    assume asm:LIM x F. f x / g x :> at-bot
    then have filterlim g (at 0) F
      using filterlim-at-infinity-divide-iff[OF assms(1,2),of g]
      at-bot-le-at-infinity filterlim-mono by blast
```

```
  then have (g ⟶ 0) F using filterlim-at by blast
  moreover have (g has-sgnx − sgn c) F
  proof −
    have ((λx. sgn c ∗ inverse (f x)) ⟶ sgn c ∗ inverse c) F
      using assms(1,2) by (auto intro:tendsto-intros)
    then have LIM x F. sgn c ∗ inverse (f x) ∗ (f x / g x) :> at-bot
      apply (elim filterlim-tendsto-pos-mult-at-bot[OF - - asm])
      using ‹c≠0› sgn-real-def by auto
    then have LIM x F. sgn c / g x :> at-bot
      apply (elim filterlim-mono-eventually)
     using eventually-times-inverse-1[OF assms] by (auto elim:eventually-mono)
    then have ∀F x in F. sgn c / g x < 0
      using filterlim-at-bot-dense[of λx. sgn c/g x F] by auto
    then show ?thesis unfolding has-sgnx-def
      apply (elim eventually-mono)
      by (metis add.inverse-inverse divide-less-0-iff sgn-neg sgn-pos sgn-sgn)
  qed
  ultimately show (g ⟶ 0) F ∧ (g has-sgnx − sgn c) F by auto
next
  assume (g ⟶ 0) F ∧ (g has-sgnx − sgn c) F
  then have asm:(g ⟶ 0) F (g has-sgnx − sgn c) F by auto
  have LIM x F. inverse (g x ∗ sgn c) :> at-bot
  proof (rule filterlim-inverse-at-bot)
    show ((λx. g x ∗ sgn c) ⟶ 0) F
      apply (rule tendsto-mult-left-zero)
      using asm(1) by blast
  next
    show ∀F x in F. g x ∗ sgn c < 0 using asm(2) unfolding has-sgnx-def
      apply (elim eventually-mono)
    by (metis add.inverse-inverse assms(2) linorder-neqE-linordered-idom mult-less-0-iff

        neg-0-less-iff-less sgn-greater sgn-zero-iff)
  qed
  moreover have ((λx. f x ∗ sgn c) ⟶ c ∗ sgn c) F
    using ‹(f ⟶ c) F› ‹c≠0›
    apply (intro tendsto-intros)
    by (auto simp add:sgn-zero-iff)
  moreover have c ∗ sgn c >0 using ‹c≠0› by (simp add: sgn-real-def)
  ultimately have LIM x F. (f x ∗ sgn c) ∗inverse (g x ∗ sgn c) :> at-bot
    using filterlim-tendsto-pos-mult-at-bot by blast
  then show LIM x F. f x / g x :> at-bot
    using ‹c≠0› by (auto simp add:field-simps sgn-zero-iff)
qed
show (LIM x F. f x / g x :> at-top) ⟷ ((g ⟶ 0) F )
  ∧ ((λx. g x) has-sgnx sgn c) F
proof
  assume asm:LIM x F. f x / g x :> at-top
  then have filterlim g (at 0) F
    using filterlim-at-infinity-divide-iff[OF assms(1,2),of g]
```

```
      at-top-le-at-infinity filterlim-mono by blast
    then have (g ⟶ 0) F using filterlim-at by blast
    moreover have (g has-sgnx sgn c) F
    proof –
      have ((λx. sgn c ∗ inverse (f x)) ⟶ sgn c ∗ inverse c) F
        using assms(1,2) by (auto intro:tendsto-intros)
      then have LIM x F. sgn c ∗ inverse (f x) ∗ (f x / g x) :> at-top
        apply (elim filterlim-tendsto-pos-mult-at-top[OF - - asm])
        using ‹c≠0› sgn-real-def by auto
      then have LIM x F. sgn c / g x :> at-top
        apply (elim filterlim-mono-eventually)
       using eventually-times-inverse-1[OF assms] by (auto elim:eventually-mono)
      then have ∀F x in F. sgn c / g x > 0
        using filterlim-at-top-dense[of λx. sgn c/g x F] by auto
      then show ?thesis unfolding has-sgnx-def
        apply (elim eventually-mono)
        by (metis sgn-greater sgn-less sgn-neg sgn-pos zero-less-divide-iff)
    qed
    ultimately show (g ⟶ 0) F ∧ (g has-sgnx sgn c) F by auto
  next
    assume (g ⟶ 0) F ∧ (g has-sgnx sgn c) F
    then have asm:(g ⟶ 0) F (g has-sgnx sgn c) F by auto
    have LIM x F. inverse (g x ∗ sgn c) :> at-top
    proof (rule filterlim-inverse-at-top)
      show ((λx. g x ∗ sgn c) ⟶ 0) F
        apply (rule tendsto-mult-left-zero)
        using asm(1) by blast
    next
      show ∀F x in F. g x ∗ sgn c > 0 using asm(2) unfolding has-sgnx-def
        apply (elim eventually-mono)
        by (metis assms(2) sgn-1-neg sgn-greater sgn-if zero-less-mult-iff)
    qed
    moreover have ((λx. f x ∗ sgn c) ⟶ c ∗ sgn c) F
      using ‹(f ⟶ c) F› ‹c≠0›
      apply (intro tendsto-intros)
      by (auto simp add:sgn-zero-iff)
    moreover have c ∗ sgn c >0 using ‹c≠0› by (simp add: sgn-real-def)
    ultimately have LIM x F. (f x ∗ sgn c) ∗inverse (g x ∗ sgn c) :> at-top
      using filterlim-tendsto-pos-mult-at-top by blast
    then show LIM x F. f x / g x :> at-top
      using ‹c≠0› by (auto simp add:field-simps sgn-zero-iff)
  qed
qed


lemma poly-sgnx-left-right:
  fixes c a::real and p::real poly
  assumes p≠0
  shows sgnx (poly p) (at-left a) = (if even (order a p)
```

```
        then sgnx (poly p) (at-right a)
        else −sgnx (poly p) (at-right a))
  using assms
proof (induction degree p arbitrary: p rule: less-induct)
  case less
  have ?case when poly p a≠0
  proof −
    have sgnx (poly p) (at-left a) = sgn (poly p a)
      by (simp add: has-sgnx-imp-sgnx tendsto-nonzero-has-sgnx that)
    moreover have sgnx (poly p) (at-right a) = sgn (poly p a)
      by (simp add: has-sgnx-imp-sgnx tendsto-nonzero-has-sgnx that)
    moreover have order a p = 0 using that by (simp add: order-0I)
    ultimately show ?thesis by auto
  qed
  moreover have ?case when poly p a=0
  proof −
    obtain q where pq:p= [:−a,1:] ∗ q
      using ‹poly p a=0› by (meson dvdE poly-eq-0-iff-dvd)
    then have q≠0 using ‹p≠0› by auto
    then have degree q < degree p unfolding pq by (subst degree-mult-eq,auto)
    have sgnx (poly p) (at-left a) = − sgnx (poly q) (at-left a)
    proof −
      have sgnx (λx. poly p x) (at-left a)
          = sgnx (poly q) (at-left a) ∗ sgnx (poly [:−a,1:]) (at-left a)
        unfolding pq
        apply (subst poly-mult)
        apply (subst sgnx-times)
        by auto
      moreover have sgnx (λx. poly [:−a,1:] x) (at-left a) = −1
        apply (intro has-sgnx-imp-sgnx)
        unfolding has-sgnx-def eventually-at-left
        by (auto simp add: linordered-field-no-lb)
      ultimately show ?thesis by auto
    qed
    moreover have sgnx (poly p) (at-right a) = sgnx (poly q) (at-right a)
    proof −
      have sgnx (λx. poly p x) (at-right a)
          = sgnx (poly q) (at-right a) ∗ sgnx (poly [:−a,1:]) (at-right a)
        unfolding pq
        apply (subst poly-mult)
        apply (subst sgnx-times)
        by auto
      moreover have sgnx (λx. poly [:−a,1:] x) (at-right a) = 1
        apply (intro has-sgnx-imp-sgnx)
        unfolding has-sgnx-def eventually-at-right
        by (auto simp add: linordered-field-no-ub)
      ultimately show ?thesis by auto
    qed
    moreover have even (order a p) ⟷ odd (order a q)
```

```
      unfolding pq
      apply (subst order-mult[OF ‹p ≠ 0›[unfolded pq]])
      using ‹q≠0› by (auto simp add:order-power-n-n[of - 1, simplified])
    moreover note less.hyps[OF ‹degree q < degree p› ‹q≠0›]
    ultimately show ?thesis by auto
  qed
  ultimately show ?case by blast
qed

lemma poly-has-sgnx-left-right:
  fixes c a::real and p::real poly
  assumes p≠0
  shows (poly p has-sgnx c) (at-left a) ⟷ (if even (order a p)
          then (poly p has-sgnx c) (at-right a)
          else (poly p has-sgnx −c) (at-right a))
using poly-sgnx-left-right
by (metis (no-types, opaque-lifting) add.inverse-inverse assms has-sgnx-unique
    sgnx-able-poly sgnx-able-sgnx trivial-limit-at-left-real trivial-limit-at-right-real)


lemma sign-r-pos-sgnx-iff:
  sign-r-pos p a ⟷ sgnx (poly p) (at-right a) > 0
proof
  assume asm:0 < sgnx (poly p) (at-right a)
  obtain c where c-def:(poly p has-sgnx c) (at-right a)
    using sgnx-able-poly(1) sgnx-able-sgnx by blast
  then have c>0 using asm
    using has-sgnx-imp-sgnx trivial-limit-at-right-real by blast
  then show sign-r-pos p a using c-def unfolding sign-r-pos-def has-sgnx-def
    apply (elim eventually-mono)
    by force
next
  assume asm:sign-r-pos p a
  define c where c = sgnx (poly p) (at-right a)
  then have (poly p has-sgnx c) (at-right a)
    by (simp add: sgnx-able-sgnx)
  then have (∀F x in (at-right a). poly p x>0 ∧ sgn (poly p x) = c)
    using asm unfolding has-sgnx-def sign-r-pos-def
    by (simp add:eventually-conj-iff)
  then have ∀F x in (at-right a). c > 0
    apply (elim eventually-mono)
    by fastforce
  then show c>0 by auto
qed

lemma sgnx-values:
  assumes f sgnx-able F F ≠ bot
  shows sgnx f F = −1 ∨ sgnx f F = 0 ∨ sgnx f F = 1
```

proof −
  obtain *c* **where** *c-def*:(*f has-sgnx c*) *F*
    using *assms*(*1*) **unfolding** *sgnx-able-def* **by** *auto*
  then obtain *x* **where** *sgn*(*f x*) = *c*
    unfolding *has-sgnx-def* **using** *assms*(*2*) *eventually-happens*
    by *blast*
  then have *c*=−*1* ∨ *c*=*0* ∨ *c*=*1* **using** *sgn-if* **by** *metis*
  moreover have *sgnx f F* = *c* **using** *c-def* **by** (*simp add*: *assms*(*2*) *has-sgnx-imp-sgnx*)
  ultimately show *?thesis* **by** *auto*
qed

lemma *has-sgnx-poly-at-top*:
    (*poly p has-sgnx sgn-pos-inf p*) *at-top*
  using *has-sgnx-def poly-sgn-eventually-at-top* **by** *blast*

lemma *has-sgnx-poly-at-bot*:
    (*poly p has-sgnx sgn-neg-inf p*) *at-bot*
  using *has-sgnx-def poly-sgn-eventually-at-bot* **by** *blast*

lemma *sgnx-poly-at-top*:
  sgnx (*poly p*) *at-top* = *sgn-pos-inf p*
by (*simp add*: *has-sgnx-def has-sgnx-imp-sgnx poly-sgn-eventually-at-top*)

lemma *sgnx-poly-at-bot*:
  sgnx (*poly p*) *at-bot* = *sgn-neg-inf p*
by (*simp add*: *has-sgnx-def has-sgnx-imp-sgnx poly-sgn-eventually-at-bot*)

lemma *poly-has-sgnx-values*:
  assumes *p*≠*0*
  shows
    (*poly p has-sgnx 1*) (*at-left a*) ∨ (*poly p has-sgnx* − *1*) (*at-left a*)
    (*poly p has-sgnx 1*) (*at-right a*) ∨ (*poly p has-sgnx* − *1*) (*at-right a*)
    (*poly p has-sgnx 1*) *at-top* ∨ (*poly p has-sgnx* − *1*) *at-top*
    (*poly p has-sgnx 1*) *at-bot* ∨ (*poly p has-sgnx* − *1*) *at-bot*
proof −
  have *sgn-pos-inf p* = *1* ∨ *sgn-pos-inf p* = −*1*
    unfolding *sgn-pos-inf-def* **by** (*simp add*: *assms sgn-if*)
  then show (*poly p has-sgnx 1*) *at-top* ∨ (*poly p has-sgnx* − *1*) *at-top*
    using *has-sgnx-poly-at-top* **by** *metis*
next
  have *sgn-neg-inf p* = *1* ∨ *sgn-neg-inf p* = −*1*
    unfolding *sgn-neg-inf-def* **by** (*simp add*: *assms sgn-if*)
  then show (*poly p has-sgnx 1*) *at-bot* ∨ (*poly p has-sgnx* − *1*) *at-bot*
    using *has-sgnx-poly-at-bot* **by** *metis*
next
  obtain *c* **where** *c-def*:(*poly p has-sgnx c*) (*at-left a*)
    using *sgnx-able-poly*(*2*) *sgnx-able-sgnx* **by** *blast*
  then have *sgnx* (*poly p*) (*at-left a*) = *c* **using** *assms* **by** *auto*
  then have *c*=−*1* ∨ *c*=*0* ∨ *c*=*1*

```
    using sgnx-values sgnx-able-poly(2) trivial-limit-at-left-real by blast
  moreover have False when c=0
  proof −
    have (poly p has-sgnx 0) (at-left a) using c-def that by auto
    then obtain lb where lb<a ∀ y. (lb<y ∧ y < a) ⟶ poly p y = 0
      unfolding has-sgnx-def eventually-at-left sgn-if
      by (metis one-neq-zero zero-neq-neg-one)
    then have {lb<..<a} ⊆ proots p unfolding proots-within-def by auto
    then have infinite (proots p)
      apply (elim infinite-super)
      using ‹lb<a› by auto
    moreover have finite (proots p) using finite-proots[OF ‹p≠0›] by auto
    ultimately show False by auto
  qed
  ultimately have c=−1 ∨ c=1 by auto
  then show (poly p has-sgnx 1) (at-left a) ∨ (poly p has-sgnx − 1) (at-left a)
    using c-def by auto
next
  obtain c where c-def:(poly p has-sgnx c) (at-right a)
    using sgnx-able-poly(1) sgnx-able-sgnx by blast
  then have sgnx (poly p) (at-right a) = c using assms by auto
  then have c=−1 ∨ c=0 ∨ c=1
    using sgnx-values sgnx-able-poly(1) trivial-limit-at-right-real by blast
  moreover have False when c=0
  proof −
    have (poly p has-sgnx 0) (at-right a) using c-def that by auto
    then obtain ub where ub>a ∀ y. (a<y ∧ y < ub) ⟶ poly p y = 0
      unfolding has-sgnx-def eventually-at-right sgn-if
      by (metis one-neq-zero zero-neq-neg-one)
    then have {a<..<ub} ⊆ proots p unfolding proots-within-def by auto
    then have infinite (proots p)
      apply (elim infinite-super)
      using ‹ub>a› by auto
    moreover have finite (proots p) using finite-proots[OF ‹p≠0›] by auto
    ultimately show False by auto
  qed
  ultimately have c=−1 ∨ c=1 by auto
  then show (poly p has-sgnx 1) (at-right a) ∨ (poly p has-sgnx − 1) (at-right a)
    using c-def by auto
qed

lemma poly-sgnx-values:
  assumes p≠0
  shows sgnx (poly p) (at-left a) = 1 ∨ sgnx (poly p) (at-left a) = −1
      sgnx (poly p) (at-right a) = 1 ∨ sgnx (poly p) (at-right a) = −1
  using poly-has-sgnx-values[OF ‹p≠0›] has-sgnx-imp-sgnx trivial-limit-at-left-real

    trivial-limit-at-right-real by blast+
```

lemma *has-sgnx-inverse*: (*f has-sgnx c*) *F* ⟷ ((*inverse o f*) *has-sgnx* (*inverse c*)) *F*
  unfolding *has-sgnx-def comp-def*
  apply (*rule eventually-subst*)
  apply (*rule always-eventually*)
  by (*metis inverse-inverse-eq sgn-inverse*)

lemma *has-sgnx-derivative-at-left*:
  assumes *g-deriv*:(*g has-field-derivative c*) (*at x*) **and** *g x=0* **and** *c≠0*
  shows (*g has-sgnx* − *sgn c*) (*at-left x*)
proof −
  have (*g has-sgnx* −*1*) (*at-left x*) **when** *c>0*
  proof −
    obtain *d1* **where** *d1>0* **and** *d1-def*:∀ *h>0*. *h* < *d1* ⟶ *g* (*x* − *h*) < *g x*
      using *DERIV-pos-inc-left*[*OF g-deriv* ‹*c>0*›] ‹*g x=0*› **by** *auto*
    have (*g has-sgnx* −*1*) (*at-left x*)
      unfolding *has-sgnx-def eventually-at-left*
      apply (*intro exI*[**where** *x=x*−*d1*])
      using ‹*d1>0*› *d1-def*
     by (*metis* (*no-types*, *opaque-lifting*) *add.commute add-uminus-conv-diff assms*(*2*) *diff-add-cancel*
         diff-strict-left-mono diff-zero minus-diff-eq sgn-neg)
    thus *?thesis* **by** *auto*
  qed
  moreover have (*g has-sgnx 1*) (*at-left x*) **when** *c<0*
  proof −
    obtain *d1* **where** *d1>0* **and** *d1-def*:∀ *h>0*. *h* < *d1* ⟶ *g* (*x* − *h*) > *g x*
        using *DERIV-neg-dec-left*[*OF g-deriv* ‹*c<0*›] ‹*g x=0*› **by** *auto*
    have (*g has-sgnx 1*) (*at-left x*)
        unfolding *has-sgnx-def eventually-at-left*
        apply (*intro exI*[**where** *x=x*−*d1*])
        using ‹*d1>0*› *d1-def*
          by (*metis* (*no-types*, *opaque-lifting*) *add.commute add-uminus-conv-diff assms*(*2*) *diff-add-cancel*
          diff-zero less-diff-eq minus-diff-eq sgn-pos)
    thus *?thesis* **using** ‹*c<0*› **by** *auto*
  qed
  ultimately show *?thesis* **using** ‹*c≠0*› **using** *sgn-real-def* **by** *auto*
qed

lemma *has-sgnx-derivative-at-right*:
  assumes *g-deriv*:(*g has-field-derivative c*) (*at x*) **and** *g x=0* **and** *c≠0*
  shows (*g has-sgnx sgn c*) (*at-right x*)
proof −
  have (*g has-sgnx 1*) (*at-right x*) **when** *c>0*
  proof −
    obtain *d2* **where** *d2>0* **and** *d2-def*:∀ *h>0*. *h* < *d2* ⟶ *g x* < *g* (*x* + *h*)
        using *DERIV-pos-inc-right*[*OF g-deriv* ‹*c>0*›] ‹*g x=0*› **by** *auto*

have (*g has-sgnx 1*) (*at-right x*)
unfolding *has-sgnx-def eventually-at-right*
apply (*intro exI*[**where** *x=x+d2*])
using ‹*d2>0*› *d2-def*
by (*metis add.commute assms*(*2*) *diff-add-cancel diff-less-eq less-add-same-cancel1 sgn-pos*)
thus *?thesis* **using** ‹*c>0*› **by** *auto*
qed
moreover have (*g has-sgnx −1*) (*at-right x*) **when** *c<0*
proof −
obtain *d2* **where** *d2>0* **and** *d2-def*:∀ *h>0. h < d2* ⟶ *g x > g* (*x + h*)
using *DERIV-neg-dec-right*[*OF g-deriv* ‹*c<0*›] ‹*g x=0*› **by** *auto*
have (*g has-sgnx −1*) (*at-right x*)
unfolding *has-sgnx-def eventually-at-right*
apply (*intro exI*[**where** *x=x+d2*])
using ‹*d2>0*› *d2-def*
by (*metis* (*no-types, opaque-lifting*) *add.commute add.right-inverse add-uminus-conv-diff assms*(*2*)
diff-add-cancel diff-less-eq sgn-neg)
thus *?thesis* **using** ‹*c<0*› **by** *auto*
qed
ultimately show *?thesis* **using** ‹*c≠0*› **using** *sgn-real-def* **by** *auto*
qed

lemma *has-sgnx-split*:
(*f has-sgnx c*) (*at x*) ⟷ (*f has-sgnx c*) (*at-left x*) ∧ (*f has-sgnx c*) (*at-right x*)
unfolding *has-sgnx-def* **using** *eventually-at-split* **by** *auto*

lemma *sgnx-at-top-IVT*:
assumes *sgnx* (*poly p*) (*at-right a*) ≠ *sgnx* (*poly p*) *at-top*
shows ∃ *x>a. poly p x=0*
proof (*cases p=0*)
case *True*
then show *?thesis* **using** *gt-ex*[*of a*] **by** *simp*
next
case *False*
from *poly-has-sgnx-values*[*OF this*]
have (*poly p has-sgnx 1*) (*at-right a*) ∨ (*poly p has-sgnx − 1*) (*at-right a*)
(*poly p has-sgnx 1*) *at-top* ∨ (*poly p has-sgnx − 1*) *at-top*
by *auto*
moreover have *?thesis* **when** *has-r*:(*poly p has-sgnx 1*) (*at-right a*)
and *has-top*:(*poly p has-sgnx −1*) *at-top*
proof −
obtain *b* **where** *b>a poly p b>0*
proof −
obtain *a′* **where** *a′>a* **and** *a′-def*:∀ *y>a. y < a′* ⟶ *sgn* (*poly p y*) = *1*
using *has-r*[*unfolded has-sgnx-def eventually-at-right*] **by** *auto*
define *b* **where** *b=*(*a+a′*)*/2*
have *a<b b<a′* **unfolding** *b-def* **using** ‹*a′>a*› **by** *auto*

```
    moreover have poly p b>0
      using a'-def[rule-format,OF ‹b>a› ‹b<a'›] unfolding sgn-if by argo
    ultimately show ?thesis using that by auto
  qed
  moreover obtain c where c>b poly p c<0
  proof -
    obtain b' where b'-def:∀ n≥b'. sgn (poly p n) = - 1
      using has-top[unfolded has-sgnx-def eventually-at-top-linorder] by auto
    define c where c=1+max b b'
    have c>b c≥b' unfolding c-def using ‹b>a› by auto
    moreover have poly p c<0
      using b'-def[rule-format,OF ‹b'≤c›] unfolding sgn-if by argo
    ultimately show ?thesis using that by auto
  qed
  ultimately show ?thesis using poly-IVT-neg[of b c p] not-less by fastforce
qed
moreover have ?thesis when has-r:(poly p has-sgnx -1) (at-right a)
    and has-top:(poly p has-sgnx 1) at-top
proof -
  obtain b where b>a poly p b<0
  proof -
    obtain a' where a'>a and a'-def:∀ y>a. y < a' ⟶ sgn (poly p y) = -1
      using has-r[unfolded has-sgnx-def eventually-at-right] by auto
    define b where b=(a+a')/2
    have a<b b<a' unfolding b-def using ‹a'>a› by auto
    moreover have poly p b<0
      using a'-def[rule-format,OF ‹b>a› ‹b<a'›] unfolding sgn-if by argo
    ultimately show ?thesis using that by auto
  qed
  moreover obtain c where c>b poly p c>0
  proof -
    obtain b' where b'-def:∀ n≥b'. sgn (poly p n) = 1
      using has-top[unfolded has-sgnx-def eventually-at-top-linorder] by auto
    define c where c=1+max b b'
    have c>b c≥b' unfolding c-def using ‹b>a› by auto
    moreover have poly p c>0
      using b'-def[rule-format,OF ‹b'≤c›] unfolding sgn-if by argo
    ultimately show ?thesis using that by auto
  qed
  ultimately show ?thesis using poly-IVT-pos[of b c p] not-less by fastforce
qed
moreover have ?thesis when
  (poly p has-sgnx 1) (at-right a) ∧ (poly p has-sgnx 1) at-top
   ∨ (poly p has-sgnx - 1) (at-right a) ∧ (poly p has-sgnx -1) at-top
proof -
  have sgnx (poly p) (at-right a) = sgnx (poly p) at-top
    using that has-sgnx-imp-sgnx by auto
  then have False using assms by simp
  then show ?thesis by auto
```

```
  qed
  ultimately show ?thesis by blast
qed

lemma sgnx-at-left-at-right-IVT:
  assumes sgnx (poly p) (at-right a) ≠ sgnx (poly p) (at-left b) a<b
  shows ∃x. a<x ∧ x<b ∧ poly p x=0
proof (cases p=0)
  case True
  then show ?thesis using ‹a<b› by (auto intro:exI[where x=(a+b)/2])
next
  case False
  from poly-has-sgnx-values[OF this]
  have (poly p has-sgnx 1) (at-right a) ∨ (poly p has-sgnx − 1) (at-right a)
    (poly p has-sgnx 1) (at-left b) ∨ (poly p has-sgnx − 1) (at-left b)
    by auto
  moreover have ?thesis when has-r:(poly p has-sgnx 1) (at-right a)
      and has-l:(poly p has-sgnx −1) (at-left b)
  proof −
    obtain c where a<c c<b poly p c>0
    proof −
      obtain a' where a'>a and a'-def:∀y>a. y < a' ⟶ sgn (poly p y) = 1
        using has-r[unfolded has-sgnx-def eventually-at-right] by auto
      define c where c=(a+min a' b)/2
      have a<c c<a' c<b unfolding c-def using ‹a'>a› ‹b>a› by auto
      moreover have poly p c>0
        using a'-def[rule-format,OF ‹c>a› ‹c<a'›] unfolding sgn-if by argo
      ultimately show ?thesis using that by auto
    qed
    moreover obtain d where c<dd<b poly p d<0
    proof −
      obtain b' where b'<b and b'-def:∀y>b'. y < b ⟶ sgn (poly p y) = − 1
        using has-l[unfolded has-sgnx-def eventually-at-left] by auto
      define d where d=(b+max b' c)/2
      have b'<d d<b d>c
        unfolding d-def using ‹b>b'› ‹b>c› by auto
      moreover have poly p d<0
        using b'-def[rule-format, OF ‹b'<d› ‹d<b›] unfolding sgn-if by argo
      ultimately show ?thesis using that by auto
    qed
    ultimately obtain x where c<x x<d poly p x=0
      using poly-IVT-neg[of c d p] by auto
    then show ?thesis using ‹c>a› ‹d<b› by (auto intro: exI[where x=x])
  qed
  moreover have ?thesis when has-r:(poly p has-sgnx −1) (at-right a)
      and has-l:(poly p has-sgnx 1) (at-left b)
  proof −
    obtain c where a<c c<b poly p c<0
    proof −
```

```
      obtain a′ where a′>a and a′-def:∀ y>a. y < a′ ⟶ sgn (poly p y) = −1
        using has-r[unfolded has-sgnx-def eventually-at-right] by auto
      define c where c=(a+min a′ b)/2
      have a<c c<a′ c<b unfolding c-def using ‹a′>a› ‹b>a› by auto
      moreover have poly p c<0
        using a′-def[rule-format,OF ‹c>a› ‹c<a′›] unfolding sgn-if by argo
      ultimately show ?thesis using that by auto
    qed
    moreover obtain d where c<dd<b poly p d>0
    proof −
      obtain b′ where b′<b and b′-def:∀ y>b′. y < b ⟶ sgn (poly p y) = 1
        using has-l[unfolded has-sgnx-def eventually-at-left] by auto
      define d where d=(b+max b′ c)/2
      have b′<d d<b d>c
        unfolding d-def using ‹b>b′› ‹b>c› by auto
      moreover have poly p d>0
        using b′-def[rule-format, OF ‹b′<d› ‹d<b›] unfolding sgn-if by argo
      ultimately show ?thesis using that by auto
    qed
    ultimately obtain x where c<x x<d poly p x=0
      using poly-IVT-pos[of c d p] by auto
    then show ?thesis using ‹c>a› ‹d<b› by (auto intro: exI[where x=x])
  qed
  moreover have ?thesis when
    (poly p has-sgnx 1) (at-right a) ∧ (poly p has-sgnx 1) (at-left b)
     ∨ (poly p has-sgnx − 1) (at-right a) ∧ (poly p has-sgnx −1) (at-left b)
  proof −
    have sgnx (poly p) (at-right a) = sgnx (poly p) (at-left b)
      using that has-sgnx-imp-sgnx by auto
    then have False using assms by simp
    then show ?thesis by auto
  qed
  ultimately show ?thesis by blast
qed

lemma sgnx-at-bot-IVT:
  assumes sgnx (poly p) (at-left a) ≠ sgnx (poly p) at-bot
  shows ∃ x<a. poly p x=0
proof (cases p=0)
  case True
  then show ?thesis using lt-ex[of a] by simp
next
  case False
  from poly-has-sgnx-values[OF this]
  have (poly p has-sgnx 1) (at-left a) ∨ (poly p has-sgnx − 1) (at-left a)
    (poly p has-sgnx 1) at-bot ∨ (poly p has-sgnx − 1) at-bot
    by auto
  moreover have ?thesis when has-l:(poly p has-sgnx 1) (at-left a)
      and has-bot:(poly p has-sgnx −1) at-bot
```

```
  proof -
    obtain b where b<a poly p b>0
    proof -
      obtain a' where a'<a and a'-def:∀ y>a'. y < a ⟶ sgn (poly p y) = 1
        using has-l[unfolded has-sgnx-def eventually-at-left] by auto
      define b where b=(a+a')/2
      have a>b b>a' unfolding b-def using ‹a'<a› by auto
      moreover have poly p b>0
        using a'-def[rule-format,OF ‹b>a'› ‹b<a›] unfolding sgn-if by argo
      ultimately show ?thesis using that by auto
    qed
    moreover obtain c where c<b poly p c<0
    proof -
      obtain b' where b'-def:∀ n≤b'. sgn (poly p n) = - 1
        using has-bot[unfolded has-sgnx-def eventually-at-bot-linorder] by auto
      define c where c=min b b'- 1
      have c<b c≤b' unfolding c-def using ‹b<a› by auto
      moreover have poly p c<0
        using b'-def[rule-format,OF ‹b'≥c›] unfolding sgn-if by argo
      ultimately show ?thesis using that by auto
    qed
     ultimately show ?thesis using poly-IVT-pos[of c b p] using not-less by
fastforce
  qed
  moreover have ?thesis when has-l:(poly p has-sgnx -1) (at-left a)
      and has-bot:(poly p has-sgnx 1) at-bot
  proof -
    obtain b where b<a poly p b<0
    proof -
      obtain a' where a'<a and a'-def:∀ y>a'. y < a ⟶ sgn (poly p y) = -1
        using has-l[unfolded has-sgnx-def eventually-at-left] by auto
      define b where b=(a+a')/2
      have a>b b>a' unfolding b-def using ‹a'<a› by auto
      moreover have poly p b<0
        using a'-def[rule-format,OF ‹b>a'› ‹b<a›] unfolding sgn-if by argo
      ultimately show ?thesis using that by auto
    qed
    moreover obtain c where c<b poly p c>0
    proof -
      obtain b' where b'-def:∀ n≤b'. sgn (poly p n) = 1
        using has-bot[unfolded has-sgnx-def eventually-at-bot-linorder] by auto
      define c where c=min b b'- 1
      have c<b c≤b' unfolding c-def using ‹b<a› by auto
      moreover have poly p c>0
        using b'-def[rule-format,OF ‹b'≥c›] unfolding sgn-if by argo
      ultimately show ?thesis using that by auto
    qed
     ultimately show ?thesis using poly-IVT-neg[of c b p] using not-less by
fastforce
```

```
  qed
  moreover have ?thesis when
    (poly p has-sgnx 1) (at-left a) ∧ (poly p has-sgnx 1) at-bot
     ∨ (poly p has-sgnx − 1) (at-left a) ∧ (poly p has-sgnx −1) at-bot
  proof −
    have sgnx (poly p) (at-left a) = sgnx (poly p) at-bot
      using that has-sgnx-imp-sgnx by auto
    then have False using assms by simp
    then show ?thesis by auto
  qed
  ultimately show ?thesis by blast
qed

lemma sgnx-poly-nz:
  assumes poly p x≠0
  shows sgnx (poly p) (at-left x) = sgn (poly p x)
        sgnx (poly p) (at-right x) = sgn (poly p x)
proof −
  have (poly p has-sgnx sgn(poly p x)) (at x)
    apply (rule tendsto-nonzero-has-sgnx)
    using assms by auto
  then show sgnx (poly p) (at-left x) = sgn (poly p x)
        sgnx (poly p) (at-right x) = sgn (poly p x)
    unfolding has-sgnx-split by auto
qed
```

5.3 Finite predicate segments over an interval

```
inductive finite-Psegments::(real ⇒ bool) ⇒ real ⇒ real ⇒ bool for P where
  emptyI: a≥b ⟹ finite-Psegments P a b|
  insertI-1: ⟦s∈{a..<b};s=a∨P s;∀ t∈{s<..<b}. P t; finite-Psegments P a s⟧
        ⟹ finite-Psegments P a b|
  insertI-2: ⟦s∈{a..<b};s=a∨P s;(∀ t∈{s<..<b}. ¬P t);finite-Psegments P a s⟧
        ⟹ finite-Psegments P a b

lemma finite-Psegments-pos-linear:
  assumes finite-Psegments P (b∗lb+c) (b∗ub+c)  and b>0
  shows finite-Psegments (P o (λt. b∗t+c)) lb ub
proof −
  have [simp]:b≠0 using ‹b>0› by auto
  show ?thesis
  proof (rule finite-Psegments.induct[OF assms(1),
      of λlb′ ub′. finite-Psegments (P o (λt. b∗t+c)) ((lb′−c)/b) ((ub′−c)/b),simplified])

    fix lb ub f assume (lb::real)≤ub
    then have (lb − c) / b ≤ (ub − c) / b
      using ‹b>0› by (auto simp add:field-simps)
    then show finite-Psegments (f ∘ (λt. b ∗ t + c)) ((ub − c) / b) ((lb − c) / b)
      by (rule finite-Psegments.emptyI)
```

```
  next
    fix s lb ub P
    assume asm: lb ≤ s ∧ s < ub
      ∀ t∈{s<..<ub}. P t
      finite-Psegments (P ∘ (λt. b ∗ t + c)) ((lb − c) / b) ((s − c) / b)
      s = lb ∨ P s
    show finite-Psegments (P ∘ (λt. b ∗ t + c)) ((lb − c) / b) ((ub − c) / b)
      apply (rule finite-Psegments.insertI-1[of (s−c)/b])
      using asm ‹b>0› by (auto simp add:field-simps)
  next
    fix s lb ub P
    assume asm: lb ≤ s ∧ s < ub
      ∀ t∈{s<..<ub}. ¬ P t
      finite-Psegments (P ∘ (λt. b ∗ t + c)) ((lb − c) / b) ((s − c) / b)
      s=lb ∨ P s
    show finite-Psegments (P ∘ (λt. b ∗ t + c)) ((lb − c) / b) ((ub − c) / b)
      apply (rule finite-Psegments.insertI-2[of (s−c)/b])
      using asm ‹b>0› by (auto simp add:field-simps)
  qed
qed

lemma finite-Psegments-congE:
  assumes finite-Psegments Q lb ub
    ⋀t. ⟦lb<t;t<ub⟧ ⟹ Q t ⟷ P t
  shows finite-Psegments P lb ub using assms
proof (induct rule:finite-Psegments.induct)
  case (emptyI a b)
  then show ?case using finite-Psegments.emptyI by auto
next
  case (insertI-1 s a b)
  show ?case
  proof (rule finite-Psegments.insertI-1[of s])
    have P s when s≠a
    proof −
      have s∈{a<..<b} using ‹s ∈ {a..<b}› that by auto
      then show ?thesis using insertI-1 by auto
    qed
    then show s = a ∨ P s by auto
  next
     show s ∈ {a..<b}  ∀ t∈{s<..<b}. P t finite-Psegments P a s using insertI-1
by auto
  qed
next
  case (insertI-2 s a b)
  show ?case
  proof (rule finite-Psegments.insertI-2[of s])
    have P s when s≠a
    proof −
      have s∈{a<..<b} using ‹s ∈ {a..<b}› that by auto
```

```
      then show ?thesis using insertI-2 by auto
    qed
    then show s = a ∨ P s by auto
  next
    show s ∈ {a..<b}  ∀ t∈{s<..<b}. ¬ P t finite-Psegments P a s using insertI-2
by auto
  qed
qed

lemma finite-Psegments-constI:
  assumes ⋀t. ⟦a<t;t<b⟧ ⟹ P t = c
  shows finite-Psegments P a b
proof −
  have finite-Psegments (λ-. c) a b
  proof −
    have ?thesis when a≥b
      using that finite-Psegments.emptyI by auto
    moreover have ?thesis when a<b c
      apply (rule finite-Psegments.insertI-1[of a])
      using that by (auto intro: finite-Psegments.emptyI)
    moreover have ?thesis when a<b ¬c
      apply (rule finite-Psegments.insertI-2[of a])
      using that by (auto intro: finite-Psegments.emptyI)
    ultimately show ?thesis by argo
  qed
  then show ?thesis
    apply (elim finite-Psegments-congE)
    using assms by auto
qed

context
begin

private lemma finite-Psegments-less-eq1:
  assumes finite-Psegments P a c b≤c
  shows finite-Psegments P a b using assms
proof (induct arbitrary: b rule:finite-Psegments.induct)
  case (emptyI a c)
  then show ?case using finite-Psegments.emptyI by auto
next
  case (insertI-1 s a c)
  have ?case when b≤s using insertI-1 that by auto
  moreover have ?case when b>s
  proof −
    have s ∈ {a..<b} using that ‹s ∈ {a..<c}› ‹b ≤ c› by auto
    moreover have ∀ t∈{s<..<b}. P t using ‹∀ t∈{s<..<c}. P t› that ‹b ≤ c› by
auto
    ultimately show ?case
      using finite-Psegments.insertI-1[OF - - - ‹finite-Psegments P a s›] ‹ s = a ∨
```

P s› **by** *auto*
  qed
  ultimately show *?case* **by** *fastforce*
next
  case (*insertI-2 s a c*)
  have *?case* **when** *b*≤*s* **using** *insertI-2 that* **by** *auto*
  moreover have *?case* **when** *b*>*s*
  proof −
    have *s* ∈ {*a*..<*b*} **using** *that* ‹*s* ∈ {*a*..<*c*}› ‹*b* ≤ *c*› **by** *auto*
    moreover have ∀ *t*∈{*s*<..<*b*}. ¬ *P t* **using** ‹∀ *t*∈{*s*<..<*c*}. ¬ *P t*› *that* ‹*b* ≤ *c*› **by** *auto*
    ultimately show *?case*
      using *finite-Psegments.insertI-2*[*OF* - - - ‹*finite-Psegments P a s*›] ‹ *s* = *a* ∨ *P s*› **by** *auto*
  qed
  ultimately show *?case* **by** *fastforce*
qed

private lemma *finite-Psegments-less-eq2*:
  assumes *finite-Psegments P a c a*≤*b*
  shows *finite-Psegments P b c* **using** *assms*
proof (*induct arbitrary*: *rule*:*finite-Psegments.induct*)
  case (*emptyI a c*)
  then show *?case* **using** *finite-Psegments.emptyI* **by** *auto*
next
  case (*insertI-1 s a c*)
  have *?case* **when** *s*≤*b*
  proof −
    have ∀ *t*∈{*b*<..<*c*}. *P t* **using** *insertI-1 that* **by** *auto*
    then show *?thesis* **by** (*simp add*: *finite-Psegments-constI*)
  qed
  moreover have *?case* **when** *s*>*b*
    apply (*rule finite-Psegments.insertI-1*[**where** *s*=*s*])
    using *insertI-1 that* **by** *auto*
  ultimately show *?case* **by** *linarith*
next
  case (*insertI-2 s a c*)
  have *?case* **when** *s*≤*b*
  proof −
    have ∀ *t*∈{*b*<..<*c*}. ¬ *P t* **using** *insertI-2 that* **by** *auto*
    then show *?thesis* **by** (*metis finite-Psegments-constI greaterThanLessThan-iff*)
  qed
  moreover have *?case* **when** *s*>*b*
    apply (*rule finite-Psegments.insertI-2*[**where** *s*=*s*])
    using *insertI-2 that* **by** *auto*
  ultimately show *?case* **by** *linarith*
qed

```
lemma finite-Psegments-included:
  assumes finite-Psegments P a d a≤b c≤d
  shows finite-Psegments P b c
  using finite-Psegments-less-eq2 finite-Psegments-less-eq1 assms by blast
end

lemma finite-Psegments-combine:
  assumes finite-Psegments P a b finite-Psegments P b c b∈{a..c} closed ({x. P
x} ∩ {a..c})
  shows finite-Psegments P a c  using assms(2,1,3,4)
proof (induct rule:finite-Psegments.induct)
  case (emptyI b c)
  then show ?case using finite-Psegments-included by auto
next
  case (insertI-1 s b c)
  have P s
  proof –
    have s<c using insertI-1 by auto
    define S where S = {x. P x} ∩ {s..(s+c)/2}
    have closed S
    proof –
      have closed ({a. P a} ∩ {a..c}) using insertI-1(8) .
      moreover have S = ({a. P a} ∩ {a..c}) ∩ {s..(s+c)/2}
        using insertI-1(1,7) unfolding S-def by (auto simp add:field-simps)
      ultimately show ?thesis
        using closed-Int[of {a. P a} ∩ {a..c} {s..(s+c)/2}] by blast
    qed
    moreover have ∃ y∈S. dist y s < e when e>0 for e
    proof –
      define y where y = min ((s+c)/2) (e/2+s)
      have y∈S
      proof –
        have y∈{s..(s+c)/2} unfolding y-def
         using ‹e>0› ‹s<c› by (auto simp add:min-mult-distrib-left algebra-simps)
        moreover have P y
          apply (rule insertI-1(3)[rule-format])
          unfolding y-def
          using ‹e>0› ‹s<c›
          by (auto simp add:algebra-simps min-mult-distrib-left min-less-iff-disj)
        ultimately show ?thesis unfolding S-def by auto
      qed
      moreover have dist y s <e
        unfolding y-def using ‹e>0› ‹s<c›
      by (auto simp add:algebra-simps min-mult-distrib-left min-less-iff-disj dist-real-def)
      ultimately show ?thesis by auto
    qed
    ultimately have s∈S using closed-approachable by auto
    then show ?thesis unfolding S-def by auto
```

```
  qed
  show ?case
  proof (rule finite-Psegments.insertI-1[of s])
    show  s ∈ {a..<c} s = a ∨ P s ∀ t∈{s<..<c}. P t
      using insertI-1 ‹P s› by auto
  next
    have closed ({a. P a} ∩ {a..s})
      using closed-Int[OF ‹closed ({a. P a} ∩ {a..c})›,of {a..s},simplified]
      apply (elim arg-elim[of closed])
      using ‹s ∈ {b..<c}› ‹b ∈ {a..c}› by auto
    then show finite-Psegments P a s using insertI-1 by auto
  qed
next
  case (insertI-2 s b c)
  have ?case when P s
  proof (rule finite-Psegments.insertI-2[of s])
    show s ∈ {a..<c} s = a ∨ P s ∀ t∈{s<..<c}. ¬ P t using that insertI-2 by auto
  next
    have closed ({a. P a} ∩ {a..s})
      using closed-Int[OF ‹closed ({a. P a} ∩ {a..c})›,of {a..s},simplified]
      apply (elim arg-elim[of closed])
      using  ‹s ∈ {b..<c}› ‹b ∈ {a..c}› by auto
    then show finite-Psegments P a s using insertI-2 by auto
  qed
  moreover have ?case when ¬ P s s=b using ‹finite-Psegments P a b›
  proof (cases rule:finite-Psegments.cases)
    case emptyI
    then show ?thesis using insertI-2 that
      by (metis antisym-conv atLeastAtMost-iff finite-Psegments.insertI-2)
  next
    case (insertI-1 s0)
    have P s
    proof −
      have s0<s using insertI-1 atLeastLessThan-iff that(2) by blast
      define S where S = {x. P x} ∩ {(s0+s)/2..s}
      have closed S
       using closed-Int[OF ‹closed ({a. P a} ∩ {a..c})›,of {(s0+s)/2..s},simplified]

        apply (elim arg-elim[of closed])
       unfolding S-def using ‹s0 ∈ {a..<b}› ‹ s ∈ {b..<c}› ‹b ∈ {a..c}› by auto

      moreover have ∃ y∈S. dist y s < e when e>0 for e
      proof −
        define y where y = max ((s+s0)/2) (s−e/2)
        have y∈S
        proof −
          have y∈{(s0+s)/2..s} unfolding y-def
           using ‹e>0› ‹s0<s› by (auto simp add:field-simps min-mult-distrib-left)
```

```
        moreover have P y
          apply (rule insertI-1(3)[rule-format])
          unfolding y-def
          using ‹e>0› ‹s0<s› ‹s=b›
          by (auto simp add:field-simps max-mult-distrib-left less-max-iff-disj)
        ultimately show ?thesis unfolding S-def by auto
      qed
      moreover have dist y s <e
        unfolding y-def using ‹e>0› ‹s0<s›
          by (auto simp add:algebra-simps max-mult-distrib-left less-max-iff-disj
dist-real-def
            max-add-distrib-right)
      ultimately show ?thesis by auto
    qed
    ultimately have s∈S using closed-approachable by auto
    then show ?thesis unfolding S-def by auto
  qed
  then have False using ‹¬ P s› by auto
  then show ?thesis by simp
next
  case (insertI-2 s0)
  have *: ∀ t∈{s0<..<c}. ¬ P t
    using ‹ ∀ t∈{s<..<c}. ¬ P t› that ‹∀ t∈{s0<..<b}. ¬ P t›
    by force
  show ?thesis
    apply (rule finite-Psegments.insertI-2[of s0])
    subgoal using insertI-2.prems(2) local.insertI-2(1) by auto
    subgoal using ‹s0 = a ∨ P s0› .
    subgoal using * .
    subgoal using ‹finite-Psegments P a s0› .
    done
qed
moreover note ‹s = b ∨ P s›
ultimately show ?case by auto
qed
```

5.4 Finite segment intersection of a path with the imaginary axis

```
definition finite-ReZ-segments::(real ⇒ complex) ⇒ complex ⇒ bool where
  finite-ReZ-segments g z = finite-Psegments (λt. Re (g t − z) = 0) 0 1

lemma finite-ReZ-segments-joinpaths:
  assumes g1:finite-ReZ-segments g1 z and g2: finite-ReZ-segments g2 z and
    path g1 path g2 pathfinish g1=pathstart g2
  shows finite-ReZ-segments (g1+++g2) z
proof −
  define P where P = (λt. (Re ((g1 +++ g2) t − z) = 0 ∧ 0<t ∧ t<1) ∨ t=0
∨ t=1)
```

```
  have finite-Psegments P 0 (1/2)
  proof -
    have finite-Psegments (λt. Re (g1 t - z) = 0) 0 1
      using g1 unfolding finite-ReZ-segments-def .
    then have finite-Psegments (λt. Re (g1 (2 * t) - z) = 0) 0 (1/2)
      apply (drule-tac finite-Psegments-pos-linear[of - 2 0 0 1/2,simplified])
      by (auto simp add:comp-def)
    then show ?thesis
      unfolding P-def joinpaths-def
      by (elim finite-Psegments-congE,auto)
  qed
  moreover have finite-Psegments P (1/2) 1
  proof -
    have finite-Psegments (λt. Re (g2 t - z) = 0) 0 1
      using g2 unfolding finite-ReZ-segments-def .
    then have finite-Psegments (λt. Re (g2 (2 * t-1) - z) = 0) (1/2) 1
      apply (drule-tac finite-Psegments-pos-linear[of - 2 1/2 -1 1,simplified])
      by (auto simp add:comp-def)
    then show ?thesis
      unfolding P-def joinpaths-def
      apply (elim finite-Psegments-congE)
      by auto
  qed
  moreover have closed {x. P x}
  proof -
    define Q where Q=(λt. Re ((g1 +++ g2) t - z) = 0)
    have continuous-on {0<..<1} (g1+++g2)
      using path-join-imp[OF ‹path g1› ‹path g2› ‹pathfinish g1=pathstart g2›]
      unfolding path-def by (auto elim:continuous-on-subset)
     from continuous-on-Re[OF this] have continuous-on {0<..<1} (λx. Re ((g1
+++ g2) x)) .
   from continuous-on-open-Collect-neq[OF this,of λ-. Re z,OF continuous-on-const,simplified]
    have open {t. Re ((g1 +++ g2) t - z) ≠ 0 ∧ 0<t ∧ t<1}
       by (elim arg-elim[where f=open],auto)
    from closed-Diff[of {0::real..1},OF - this,simplified]
    show closed {x. P x}
      apply (elim arg-elim[where f=closed])
      by (auto simp add:P-def)
  qed
  ultimately have finite-Psegments P 0 1
    using finite-Psegments-combine[of - 0 1/2 1] by auto
  then show ?thesis
    unfolding finite-ReZ-segments-def P-def
    by (elim finite-Psegments-congE,auto)
qed

lemma finite-ReZ-segments-congE:
  assumes finite-ReZ-segments p1 z1
    ⋀t. ⟦0<t;t<1⟧ ⟹ Re(p1 t- z1) = Re(p2 t - z2)
```

shows *finite-ReZ-segments p2 z2*
using *assms* **unfolding** *finite-ReZ-segments-def*
apply (*elim finite-Psegments-congE*)
by *auto*

lemma *finite-ReZ-segments-constI*:
assumes ∀ *t*. *0*<*t*∧*t*<*1* ⟶ *g t* = *c*
shows *finite-ReZ-segments g z*
proof −
have *finite-ReZ-segments* (λ-. *c*) *z*
unfolding *finite-ReZ-segments-def*
by (*rule finite-Psegments-constI*,*auto*)
then show *?thesis* **using** *assms*
by (*elim finite-ReZ-segments-congE*,*auto*)
qed

lemma *finite-ReZ-segment-cases* [*consumes 1*, *case-names subEq subNEq*,*cases pred*:*finite-ReZ-segments*]:
assumes *finite-ReZ-segments g z*
and *subEq*:(⋀*s*. ⟦*s* ∈ {*0*..<*1*};*s*=*0*∨*Re* (*g s*) = *Re z*;
∀ *t*∈{*s*<..<*1*}. *Re* (*g t*) = *Re z*;*finite-ReZ-segments* (*subpath 0 s g*) *z*⟧ ⟹
P)
and *subNEq*:(⋀*s*. ⟦*s* ∈ {*0*..<*1*};*s*=*0*∨*Re* (*g s*) = *Re z*;
∀ *t*∈{*s*<..<*1*}. *Re* (*g t*) ≠ *Re z*;*finite-ReZ-segments* (*subpath 0 s g*) *z*⟧ ⟹
P)
shows *P*
using *assms*(*1*) **unfolding** *finite-ReZ-segments-def*
proof (*cases rule*:*finite-Psegments.cases*)
case *emptyI*
then show *?thesis* **by** *auto*
next
case (*insertI-1 s*)
have *finite-ReZ-segments* (*subpath 0 s g*) *z*
proof (*cases s*=*0*)
case *True*
show *?thesis*
apply (*rule finite-ReZ-segments-constI*)
using *True* **unfolding** *subpath-def* **by** *auto*
next
case *False*
then have *s*>*0* **using** ‹*s*∈{*0*..<*1*}› **by** *auto*
from *finite-Psegments-pos-linear*[*OF - this*,*of - 0 0 1*] *insertI-1*(*4*)
show *finite-ReZ-segments* (*subpath 0 s g*) *z*
unfolding *finite-ReZ-segments-def comp-def subpath-def* **by** *auto*
qed
then show *?thesis* **using** *subEq insertI-1* **by** *force*
next
case (*insertI-2 s*)
have *finite-ReZ-segments* (*subpath 0 s g*) *z*
proof (*cases s*=*0*)

```
    case True
    show ?thesis
      apply (rule finite-ReZ-segments-constI)
      using True unfolding subpath-def by auto
  next
    case False
    then have s>0 using ‹s∈{0..<1}› by auto
    from finite-Psegments-pos-linear[OF - this,of - 0 0 1] insertI-2(4)
    show finite-ReZ-segments (subpath 0 s g) z
      unfolding finite-ReZ-segments-def comp-def subpath-def by auto
  qed
  then show ?thesis using subNEq insertI-2 by force
qed

lemma finite-ReZ-segments-induct [case-names sub0 subEq subNEq, induct pred:finite-ReZ-segments]:
  assumes finite-ReZ-segments g z
  assumes  sub0:⋀g z. (P (subpath 0 0 g) z)
    and subEq:(⋀s g z. ⟦s ∈ {0..<1};s=0∨Re (g s) = Re z;
          ∀ t∈{s<..<1}. Re (g t) = Re z;finite-ReZ-segments (subpath 0 s g) z;
          P (subpath 0 s g) z⟧ ⟹ P g z)
    and subNEq:(⋀s g z. ⟦s ∈ {0..<1};s=0∨Re (g s) = Re z;
          ∀ t∈{s<..<1}. Re (g t) ≠ Re z;finite-ReZ-segments (subpath 0 s g) z;
          P (subpath 0 s g) z⟧ ⟹ P g z)
  shows P g z
proof –
  have finite-Psegments (λt. Re (g t − z) = 0) 0 1
    using assms(1) unfolding finite-ReZ-segments-def by auto
  then have (0::real)≤1 ⟶ P (subpath 0 1 g) z
  proof (induct rule: finite-Psegments.induct[of - 0 1 λa b. b≥a ⟶ P (subpath a
b g) z] )
    case (emptyI a b)
    then show ?case using sub0[of subpath a b g] unfolding subpath-def by auto

  next
    case (insertI-1 s a b)
    have ?case when a=b
      using sub0[of subpath a b g] that unfolding subpath-def by auto
    moreover have ?case when a≠b
    proof –
      have b>a using that ‹s ∈ {a..<b}› by auto
      define s'::real where s'=(s−a)/(b−a)
      have P (subpath a b g) z
      proof (rule subEq[of s' subpath a b g])
        show ∀ t∈{s'<..<1}. Re (subpath a b g t) = Re z
        proof
          fix t assume t ∈ {s'<..<1}
          then have (b − a) ∗ t + a∈{s<..<b}
            unfolding s'-def using ‹b>a› ‹s ∈ {a..<b}›
            apply (auto simp add:field-simps)
```

```
        by (sos ((((A<0 * (A<1 * A<2)) * R<1) + (((A<=1 * (A<0 * R<1))
* (R<1 * [1]^2))
            + ((A<=0 * (A<0 * (A<1 * R<1))) * (R<1 * [1]^2))))))
      then have Re (g ((b − a) * t + a) − z) = 0
        using insertI-1(3)[rule-format,of (b − a) * t + a] by auto
      then show Re (subpath a b g t) = Re z
        unfolding subpath-def by auto
    qed
    show finite-ReZ-segments (subpath 0 s′ (subpath a b g)) z
    proof (cases s=a)
      case True
      then show ?thesis unfolding s′-def subpath-def
        by (auto intro:finite-ReZ-segments-constI)
    next
      case False
      have finite-Psegments (λt. Re (g t − z) = 0) a s
        using insertI-1(4) unfolding finite-ReZ-segments-def by auto
      then have finite-Psegments ((λt. Re (g t − z) = 0) ∘ (λt. (s − a) * t +
a)) 0 1
        apply (elim finite-Psegments-pos-linear[of - s−a 0 a 1,simplified])
        using False ‹s∈{a..<b}› by auto
      then show ?thesis
     using ‹b>a› unfolding finite-ReZ-segments-def subpath-def s′-def comp-def
        by auto
    qed
    show s′ ∈ {0..<1}
      using ‹b>a› ‹s∈{a..<b}› unfolding s′-def
      by (auto simp add:field-simps)
    show P (subpath 0 s′ (subpath a b g)) z
    proof −
      have P (subpath a s g) z using insertI-1(1,5) by auto
      then show ?thesis
        using ‹b>a› unfolding s′-def subpath-def  by simp
    qed
    show s′ = 0 ∨ Re (subpath a b g s′) = Re z
    proof −
      have ?thesis when s=a
        using that unfolding s′-def by auto
      moreover have ?thesis when Re (g s − z) = 0
        using that unfolding s′-def subpath-def by auto
      ultimately show ?thesis using ‹s = a ∨ Re (g s − z) = 0› by auto
    qed
  qed
  then show ?thesis using ‹b>a› by auto
 qed
 ultimately show ?case by auto
next
 case (insertI-2 s a b)
 have ?case when a=b
```

```
    using sub0[of subpath a b g] that unfolding subpath-def by auto
  moreover have ?case when a≠b
  proof −
    have b>a using that ‹s ∈ {a..<b}› by auto
    define s'::real where s'=(s−a)/(b−a)
    have P (subpath a b g) z
    proof (rule subNEq[of s' subpath a b g])
      show ∀ t∈{s'<..<1}. Re (subpath a b g t) ≠ Re z
      proof
        fix t assume t ∈ {s'<..<1}
        then have (b − a) ∗ t + a∈{s<..<b}
          unfolding s'-def using ‹b>a› ‹s ∈ {a..<b}›
          apply (auto simp add:field-simps)
         by (sos ((((A<0 ∗ (A<1 ∗ A<2)) ∗ R<1) + (((A<=1 ∗ (A<0 ∗ R<1))
∗ (R<1 ∗ [1]^2)) +
           ((A<=0 ∗ (A<0 ∗ (A<1 ∗ R<1))) ∗ (R<1 ∗ [1]^2))))))
        then have Re (g ((b − a) ∗ t + a) − z) ≠ 0
          using insertI-2(3)[rule-format,of (b − a) ∗ t + a] by auto
        then show Re (subpath a b g t) ≠ Re z
          unfolding subpath-def by auto
      qed
      show finite-ReZ-segments (subpath 0 s' (subpath a b g)) z
      proof (cases s=a)
        case True
        then show ?thesis unfolding s'-def subpath-def
          by (auto intro:finite-ReZ-segments-constI)
      next
        case False
        have finite-Psegments (λt. Re (g t − z) = 0) a s
          using insertI-2(4) unfolding finite-ReZ-segments-def by auto
        then have finite-Psegments ((λt. Re (g t − z) = 0) ∘ (λt. (s − a) ∗ t +
a)) 0 1
          apply (elim finite-Psegments-pos-linear[of - s−a 0 a 1,simplified])
          using False ‹s∈{a..<b}› by auto
        then show ?thesis
         using ‹b>a› unfolding finite-ReZ-segments-def subpath-def s'-def comp-def
          by auto
      qed
      show s' ∈ {0..<1}
        using ‹b>a› ‹s∈{a..<b}› unfolding s'-def
        by (auto simp add:field-simps)
      show P (subpath 0 s' (subpath a b g)) z
      proof −
        have P (subpath a s g) z using insertI-2(1,5) by auto
        then show ?thesis
          using ‹b>a› unfolding s'-def subpath-def  by simp
      qed
      show s' = 0 ∨ Re (subpath a b g s') = Re z
      proof −
```

have *?thesis* **when** *s=a*
using *that* **unfolding** *s′-def* **by** *auto*
moreover have *?thesis* **when** *Re* (*g s* − *z*) = *0*
using *that* **unfolding** *s′-def subpath-def* **by** *auto*
ultimately show *?thesis* **using** ‹*s* = *a* ∨ *Re* (*g s* − *z*) = *0*› **by** *auto*
qed
qed
then show *?thesis* **using** ‹*b*>*a*› **by** *auto*
qed
ultimately show *?case* **by** *auto*
qed
then show *?thesis* **by** *auto*
qed

lemma *finite-ReZ-segments-shiftpah*:
assumes *finite-ReZ-segments g z s*∈{*0*..*1*} *path g* **and** *loop*:*pathfinish g* = *path-start g*
shows *finite-ReZ-segments* (*shiftpath s g*) *z*
proof −
have *finite-Psegments* (λ*t*. *Re* (*shiftpath s g t* − *z*) = *0*) *0* (*1*−*s*)
proof −
have *finite-Psegments* (λ*t*. *Re* (*g t*) = *Re z*) *s 1*
using *assms finite-Psegments-included*[*of - 0 1 s*] **unfolding** *finite-ReZ-segments-def*

by *force*
then have *finite-Psegments* (λ*t*. *Re* (*g* (*s* + *t*) − *z*) = *0*) *0* (*1*−*s*)
using *finite-Psegments-pos-linear*[*of* λ*t*. *Re* (*g t* − *z*) =*0 1 0 s 1*−*s*,*simplified*]
unfolding *comp-def* **by** (*auto simp add*:*algebra-simps*)
then show *?thesis* **unfolding** *shiftpath-def*
apply (*elim finite-Psegments-congE*)
using ‹*s*∈{*0*..*1*}› **by** *auto*
qed
moreover have *finite-Psegments* (λ*t*. *Re* (*shiftpath s g t* − *z*) = *0*) (*1*−*s*) *1*
proof −
have *finite-Psegments* (λ*t*. *Re* (*g t*) = *Re z*) *0 s*
using *assms finite-Psegments-included* **unfolding** *finite-ReZ-segments-def*
by *force*
then have *finite-Psegments* (λ*t*. *Re* (*g* (*s* + *t* − *1*) − *z*) = *0*) (*1*−*s*) *1*
using *finite-Psegments-pos-linear*[*of* λ*t*. *Re* (*g t* − *z*) =*0 1 1*−*s s*−*1 1*,*simplified*]
unfolding *comp-def* **by** (*auto simp add*:*algebra-simps*)
then show *?thesis* **unfolding** *shiftpath-def*
apply (*elim finite-Psegments-congE*)
using ‹*s*∈{*0*..*1*}› **by** *auto*
qed
moreover have *1* − *s* ∈ {*0*..*1*} **using** ‹*s*∈{*0*..*1*}› **by** *auto*
moreover have *closed* ({*x*. *Re* (*shiftpath s g x* − *z*) = *0*} ∩ {*0*..*1*})
proof −
let *?f* = λ*x*. *Re* (*shiftpath s g x* − *z*)
have *continuous-on* {*0*..*1*} *?f*

```
      using path-shiftpath[OF ‹path g› loop ‹s∈{0..1}›] unfolding path-def
      by (auto intro: continuous-intros)
    from continuous-closed-preimage-constant[OF this,of 0,simplified]
    show ?thesis
      apply (elim arg-elim[of closed])
      by force
  qed
  ultimately show ?thesis unfolding finite-ReZ-segments-def
    by (rule finite-Psegments-combine[where b=1−s])
qed

lemma finite-imp-finite-ReZ-segments:
  assumes finite {t. Re (g t − z) = 0 ∧ 0 ≤ t ∧ t≤1}
  shows finite-ReZ-segments g z
proof −
  define P where P = (λt. Re (g t − z) = 0)
  define rs where rs=(λb. {t. P t ∧ 0 < t ∧ t<b})
  have finite-Psegments P 0 b when finite (rs b) b>0 for b
  using that
  proof (induct card (rs b) arbitrary:b rule:nat-less-induct)
    case ind:1
    have ?case when rs b= {}
      apply (rule finite-Psegments.intros(3)[of 0])
      using that ‹0 < b› unfolding rs-def by (auto intro:finite-Psegments.intros)

    moreover have ?case when rs b≠{}
    proof −
      define lj where lj = Max (rs b)
      have 0<lj lj<b P lj
        using Max-in[OF ‹finite (rs b)› ‹rs b≠{}›,folded lj-def]
        unfolding rs-def by auto
      show ?thesis
      proof (rule finite-Psegments.intros(3)[of lj])
        show lj ∈ {0..<b}  lj = 0 ∨ P lj
          using ‹0<lj› ‹lj<b› ‹P lj› by auto
        show ∀ t∈{lj<..<b}. ¬ P t
        proof (rule ccontr)
          assume  ¬ (∀ t∈{lj<..<b}. ¬ P t)
          then obtain t where t:P t lj < t t < b by auto
          then have t∈rs b unfolding rs-def using ‹lj>0› by auto
         then have t≤lj using Max-ge[OF ‹finite (rs b)›,of t] unfolding lj-def by
auto
          then show False using ‹t>lj› by auto
        qed
        show finite-Psegments P 0 lj
        proof (rule ind.hyps[rule-format,of card (rs lj) lj,simplified])
          show finite (rs lj)
            using ‹finite (rs b)› unfolding rs-def using ‹lj<b›
            by (auto elim!:rev-finite-subset )
```

show *card* (*rs lj*) < *card* (*rs b*)
apply (*rule psubset-card-mono*[*OF* ‹*finite* (*rs b*)›])
using *Max-in* ‹*finite* (*rs lj*)› ‹*lj* < *b*› *lj-def rs-def that* **by** *fastforce*
show *0* < *lj* **using** ‹*0*<*lj*› **.**
qed
qed
qed
ultimately show *?case* **by** *auto*
qed
moreover have *finite* (*rs 1*)
using *assms* **unfolding** *rs-def P-def*
by (*auto elim*:*rev-finite-subset*)
ultimately have *finite-Psegments P 0 1* **by** *auto*
then show *?thesis* **unfolding** *P-def finite-ReZ-segments-def* **.**
qed

lemma *finite-ReZ-segments-poly-linepath*:
shows *finite-ReZ-segments* (*poly p o linepath a b*) *z*
proof −
define *P* **where** *P*=*map-poly Re* (*pcompose* (*p*−[:*z*:]) [:*a*,*b*−*a*:])
have ∗:*Re* ((*poly p* ∘ *linepath a b*) *t* − *z*) = *0* ⟷ *poly P t*=*0* **for** *t*
unfolding *inner-complex-def P-def linepath-def comp-def*
apply (*subst Re-poly-of-real*[*symmetric*])
by (*auto simp add*: *algebra-simps poly-pcompose scaleR-conv-of-real*)
have *?thesis* **when** *P*≠*0*
proof −
have *finite* {*t*. *poly P t*=*0*} **using** *that poly-roots-finite* **by** *auto*
then have *finite* {*t*. *Re* ((*poly p* ∘ *linepath a b*) *t* − *z*) = *0* ∧ *0* ≤ *t* ∧ *t* ≤ *1*}
using ∗
by *auto*
then show *?thesis*
using *finite-imp-finite-ReZ-segments*[*of poly p o linepath a b z*] **by** *auto*
qed
moreover have *?thesis* **when** *P*=*0*
unfolding *finite-ReZ-segments-def*
apply (*rule finite-Psegments-constI*[**where** *c*=*True*])
apply (*subst* ∗)
using *that* **by** *auto*
ultimately show *?thesis* **by** *auto*
qed

lemma *part-circlepath-half-finite-inter*:
assumes *st*≠*tt r*≠*0 c*≠*0*
shows *finite* {*t*. *part-circlepath z0 r st tt t* • *c* = *d* ∧ *0*≤ *t* ∧ *t*≤*1*} (**is** *finite ?T*)
proof −
let *?S* = {*ϑ*. (*z0*+*r*∗*exp* (i ∗ *ϑ*)) • *c* = *d* ∧ *ϑ* ∈ *closed-segment st tt*}
define *S* **where** *S* ≡ {*ϑ*. (*z0*+*r*∗*exp* (i ∗ *ϑ*)) • *c* = *d* ∧ *ϑ* ∈ *closed-segment st tt*}
have *S* = *linepath st tt* ‘ *?T*

```
  proof
    define g where g≡(λt. (t−st)/(tt −st))
    have 0≤g t g t≤1 when t ∈ closed-segment st tt  for t
      using that ‹st≠tt› closed-segment-eq-real-ivl unfolding  g-def real-scaleR-def
      by (auto simp add:divide-simps)
    moreover have linepath st tt (g t) =t g (linepath st tt t) = t for t
      unfolding linepath-def g-def real-scaleR-def using ‹st≠tt›
      apply (simp-all add:divide-simps)
      by (auto simp add:algebra-simps )
    ultimately have x∈linepath st tt ‘ ?T when x∈S for x
      using that unfolding S-def
      by (auto intro!:image-eqI[where x=g x] simp add:part-circlepath-def)
    then show S ⊆ linepath st tt ‘ ?T by auto
  next
    have x∈S when x∈linepath st tt ‘ ?T for x
      using that unfolding part-circlepath-def S-def
      by (auto simp add: linepath-in-path)
    then show linepath st tt ‘ ?T ⊆ S by auto
  qed
  moreover have finite S
  proof −
    define a′ b′ c′ where a′=r ∗ Re c and b′ = r∗ Im c and c′=Im c ∗ Im z0 +
Re z0 ∗ Re c − d
    define f where f ϑ= a′ ∗ cos ϑ + b′ ∗ sin ϑ + c′ for ϑ
    have (z0+r∗exp (i ∗ ϑ )) · c = d ⟷ f ϑ = 0 for ϑ
      unfolding exp-Euler inner-complex-def f-def a′-def b′-def c′-def
      by (auto simp add:algebra-simps cos-of-real sin-of-real)
    then have ∗:S = roots f ∩ closed-segment st tt
      unfolding S-def roots-within-def by auto
    have uniform-discrete S
    proof −
      have a′ ≠ 0 ∨ b′ ≠ 0 ∨ c′ ≠ 0
        using assms complex-eq-iff unfolding a′-def b′-def c′-def
        by auto
      then have periodic-set (roots f) (4 ∗ pi)
        using periodic-set-sin-cos-linear[of a′ b′ c′,folded f-def] by auto
      then have uniform-discrete (roots f) using periodic-imp-uniform-discrete by
auto
      then show ?thesis unfolding ∗ by auto
    qed
    moreover have bounded S unfolding ∗
      by (simp add: bounded-Int bounded-closed-segment)
    ultimately show ?thesis using uniform-discrete-finite-iff by auto
  qed
  moreover have inj-on (linepath st tt) ?T
  proof −
    have inj (linepath st tt)
      unfolding linepath-def using assms inj-segment by blast
    then show ?thesis by (auto elim:subset-inj-on)
```

```
  qed
  ultimately show ?thesis by (auto elim!: finite-imageD)
qed

lemma linepath-half-finite-inter:
  assumes a · c ≠ d ∨ b · c ≠ d
  shows finite {t. linepath a b t · c = d ∧ 0≤ t ∧ t≤1} (is finite ?S)
proof (rule ccontr)
  assume asm:infinite ?S
  obtain t1 t2 where u1u2:t1≠t2 t1∈?S t2∈?S
  proof −
    obtain t1 where t1∈?S using not-finite-existsD asm by blast
    moreover have ∃ u2. u2∈?S−{t1}
      using infinite-remove[OF asm,of t1]
      by (meson finite.emptyI rev-finite-subset subsetI)
    ultimately show ?thesis using that by auto
  qed
  have t1:(1−t1)∗(a · c) + t1 ∗ (b · c) = d
    using ‹t1∈?S› unfolding linepath-def by (simp add: inner-left-distrib)
  have t2:(1−t2)∗(a · c) + t2 ∗ (b · c) = d
    using ‹t2∈?S› unfolding linepath-def by (simp add: inner-left-distrib)
  have a · c = d
  proof −
    have t2∗((1−t1)∗(a · c) + t1 ∗ (b · c)) = t2∗d using t1 by auto
     then have ∗:(t2−t1∗t2)∗(a · c) + t1∗t2 ∗ (b · c) = t2∗d by (auto simp add:algebra-simps)
    have t1∗((1−t2)∗(a · c) + t2 ∗ (b · c)) = t1∗d using t2 by auto
     then have ∗∗:(t1−t1∗t2)∗(a · c) + t1∗t2 ∗ (b · c) = t1∗d by (auto simp add:algebra-simps)
    have (t2−t1)∗(a · c) = (t2−t1)∗d using arg-cong2[OF ∗ ∗∗,of minus]
      by (auto simp add:algebra-simps)
    then show ?thesis using ‹t1≠t2› by auto
  qed
  moreover have b · c = d
  proof −
    have (1−t2)∗((1−t1)∗(a · c) + t1 ∗ (b · c)) = (1−t2)∗d using t1 by auto
     then have ∗:(1−t1)∗(1−t2)∗(a · c) + (t1−t1∗t2) ∗ (b · c) = (1−t2)∗d by (auto simp add:algebra-simps)
    have (1−t1)∗((1−t2)∗(a · c) + t2 ∗ (b · c)) = (1−t1)∗d using t2 by auto
    then have ∗∗:(1−t1)∗(1−t2)∗(a · c) + (t2−t1∗t2) ∗ (b · c) = (1−t1)∗d by (auto simp add:algebra-simps)
    have (t2−t1)∗(b · c) = (t2−t1)∗d using arg-cong2[OF ∗∗ ∗,of minus]
      by (auto simp add:algebra-simps)
    then show ?thesis using ‹t1≠t2› by auto
  qed
  ultimately show False using assms by auto
qed

lemma finite-half-joinpaths-inter:
```

```
  assumes finite {t. l1 t • c = d ∧ 0≤ t ∧ t≤1} finite {t. l2 t • c = d ∧ 0≤ t ∧
t≤1}
  shows finite {t. (l1+++l2) t • c = d ∧ 0≤ t ∧ t≤1}
proof −
  let ?l1s = {t. l1 (2∗t) • c = d ∧ 0≤ t ∧ t≤1/2}
  let ?l2s = {t. l2 (2 ∗ t − 1) • c = d ∧ 1/2< t ∧ t≤1}
  let ?ls = λl. {t. l t • c = d ∧ 0≤ t ∧ t≤1}
  have {t. (l1+++l2) t • c = d ∧ 0≤ t ∧ t≤1} = ?l1s ∪ ?l2s
    unfolding joinpaths-def by auto
  moreover have finite ?l1s
  proof −
    have ?l1s = ((∗) (1/2)) ‘ ?ls l1 by (auto intro:rev-image-eqI)
    thus ?thesis using assms by simp
  qed
  moreover have finite ?l2s
  proof −
     have ?l2s ⊆ (λx. x/2 + 1/2) ‘ ?ls l2 by (auto intro:rev-image-eqI simp
add:field-simps)
    thus ?thesis using assms
      by (auto elim:finite-subset)
  qed
  ultimately show ?thesis by simp
qed

lemma finite-ReZ-segments-linepath:
  finite-ReZ-segments (linepath a b) z
proof −
  have ?thesis when Re a≠Re z ∨ Re b ≠Re z
  proof −
    let ?S1={t. Re (linepath a b t−z) = 0 ∧ 0 ≤ t ∧ t ≤ 1}
    have finite ?S1
      using linepath-half-finite-inter[of a Complex 1 0 Re z b] that
      using one-complex.code by auto
    from finite-imp-finite-ReZ-segments[OF this] show ?thesis .
  qed
  moreover have ?thesis when Re a=Re z Re b=Re z
    unfolding finite-ReZ-segments-def
    apply (rule finite-Psegments.intros(2)[of 0])
   using that unfolding linepath-def by (auto simp add:algebra-simps intro:finite-Psegments.intros)
  ultimately show ?thesis by blast
qed

lemma finite-ReZ-segments-part-circlepath:
  finite-ReZ-segments (part-circlepath z0 r st tt) z
proof −
  have ?thesis when st ≠ tt r ≠ 0
  proof −
    let ?S1={t. Re (part-circlepath z0 r st tt t−z) = 0 ∧ 0 ≤ t ∧ t ≤ 1}
    have finite ?S1
```

using *part-circlepath-half-finite-inter*[*of st tt r Complex 1 0 z0 Re z*] *that one-complex.code*
by (*auto simp add*:*inner-complex-def*)
from *finite-imp-finite-ReZ-segments*[*OF this*] **show** *?thesis* **.**
qed
moreover have *?thesis* **when** *st* =*tt* ∨ *r*=*0*
proof −
define *c* **where** *c* = *z0* + *r* ∗ *exp* (i ∗ *tt*)
have *part-circlepath z0 r st tt* = (λ*t*. *c*)
unfolding *part-circlepath-def c-def* **using** *that linepath-refl* **by** *auto*
then show *?thesis*
using *finite-ReZ-segments-linepath*[*of c c z*] *linepath-refl*[*of c*]
by *auto*
qed
ultimately show *?thesis* **by** *blast*
qed

lemma *finite-ReZ-segments-poly-of-real*:
shows *finite-ReZ-segments* (*poly p o of-real*) *z*
using *finite-ReZ-segments-poly-linepath*[*of p 0 1 z*] **unfolding** *linepath-def*
by (*auto simp add*:*scaleR-conv-of-real*)

lemma *finite-ReZ-segments-subpath*:
assumes *finite-ReZ-segments g z*
0≤*u u*≤*v v*≤*1*
shows *finite-ReZ-segments* (*subpath u v g*) *z*
proof (*cases u*=*v*)
case *True*
then show *?thesis*
unfolding *subpath-def* **by** (*auto intro*:*finite-ReZ-segments-constI*)
next
case *False*
then have *u*<*v* **using** ‹*u*≤*v*› **by** *auto*
define *P* **where** *P*=(λ*t*. *Re* (*g t* − *z*) = *0*)
have *finite-ReZ-segments* (*subpath u v g*) *z*
= *finite-Psegments* (*P o* (λ*t*. (*v* − *u*) ∗ *t* + *u*)) *0 1*
unfolding *finite-ReZ-segments-def subpath-def P-def comp-def* **by** *auto*
also have ...
apply (*rule finite-Psegments-pos-linear*)
using *assms False* **unfolding** *finite-ReZ-segments-def*
by (*fold P-def*,*auto elim*:*finite-Psegments-included*)
finally show *?thesis* **.**
qed

5.5 jump and jumpF

definition *jump*::(*real* ⇒ *real*) ⇒ *real* ⇒ *int* **where**
jump f a = (*if*
(*LIM x* (*at-left a*). *f x* :> *at-bot*) ∧ (*LIM x* (*at-right a*). *f x* :> *at-top*)

then 1 else if
(LIM x (at-left a). f x :> at-top) ∧ (LIM x (at-right a). f x :> at-bot)
then −1 else 0)

definition *jumpF::(real ⇒ real) ⇒ real filter ⇒ real* **where**
jumpF f F ≡ (if filterlim f at-top F then 1/2 else
if filterlim f at-bot F then −1/2 else (0::real))

lemma *jumpF-const[simp]*:
assumes *F≠bot*
shows *jumpF (λ-. c) F = 0*
proof −
have *False* **when** *LIM x F. c :> at-bot*
using *filterlim-at-bot-nhds[OF that - ‹F≠bot›]* **by** *auto*
moreover have *False* **when** *LIM x F. c :> at-top*
using *filterlim-at-top-nhds[OF that - ‹F≠bot›]* **by** *auto*
ultimately show *?thesis* **unfolding** *jumpF-def* **by** *auto*
qed

lemma *jumpF-not-infinity*:
assumes *continuous F g F≠bot*
shows *jumpF g F = 0*
proof −
have ¬ *filterlim g at-infinity F*
using *not-tendsto-and-filterlim-at-infinity[OF ‹F ≠bot› assms(1)[unfolded continuous-def]]*
by *auto*
then have ¬ *filterlim g at-bot F* ¬ *filterlim g at-top F*
using *at-bot-le-at-infinity at-top-le-at-infinity filterlim-mono* **by** *blast+*
then show *?thesis* **unfolding** *jumpF-def* **by** *auto*
qed

lemma *jumpF-linear-comp*:
assumes *c≠0*
shows
jumpF (f o (λx. c∗x+b)) (at-left x) =
(if c>0 then jumpF f (at-left (c∗x+b)) else jumpF f (at-right (c∗x+b)))
(**is** *?case1*)
jumpF (f o (λx. c∗x+b)) (at-right x) =
(if c>0 then jumpF f (at-right (c∗x+b)) else jumpF f (at-left (c∗x+b)))
(**is** *?case2*)
proof −
let *?g = λx. c∗x+b*
have *?case1 ?case2* **when** ¬ *c>0*
proof −
have *c<0* **using** *‹c≠0› that* **by** *auto*
have *filtermap ?g (at-left x) = at-right (?g x)*
filtermap ?g (at-right x) = at-left (?g x)
using *‹c<0›*

filtermap-linear-at-left[*OF* ‹*c*≠*0*›, *of b x*]
filtermap-linear-at-right[*OF* ‹*c*≠*0*›, *of b x*] **by** *auto*
then have
jumpF (*f* ∘ *?g*) (*at-left x*) = *jumpF f* (*at-right* (*?g x*))
jumpF (*f* ∘ *?g*) (*at-right x*) = *jumpF f* (*at-left* (*?g x*))
unfolding *jumpF-def filterlim-def comp-def*
by (*auto simp add*: *filtermap-filtermap*[*of f ?g*,*symmetric*])
then show *?case1 ?case2* **using** ‹*c*<*0*› **by** *auto*
qed
moreover have *?case1 ?case2* **when** *c*>*0*
proof −
have *filtermap ?g* (*at-left x*) = *at-left* (*?g x*)
filtermap ?g (*at-right x*) = *at-right* (*?g x*)
using *that*
filtermap-linear-at-left[*OF* ‹*c*≠*0*›, *of b x*]
filtermap-linear-at-right[*OF* ‹*c*≠*0*›, *of b x*] **by** *auto*
then have
jumpF (*f* ∘ *?g*) (*at-left x*) = *jumpF f* (*at-left* (*?g x*))
jumpF (*f* ∘ *?g*) (*at-right x*) = *jumpF f* (*at-right* (*?g x*))
unfolding *jumpF-def filterlim-def comp-def*
by (*auto simp add*: *filtermap-filtermap*[*of f ?g*,*symmetric*])
then show *?case1 ?case2* **using** *that* **by** *auto*
qed
ultimately show *?case1 ?case2* **by** *auto*
qed

lemma *jump-const*[*simp*]:*jump* (λ-. *c*) *a* = *0*
proof −
have *False* **when** *LIM x* (*at-left a*). *c* :> *at-bot*
apply (*rule not-tendsto-and-filterlim-at-infinity*[*of at-left a* λ-. *c c*])
apply *auto*
using *at-bot-le-at-infinity filterlim-mono that* **by** *blast*
moreover have *False* **when** *LIM x* (*at-left a*). *c* :> *at-top*
apply (*rule not-tendsto-and-filterlim-at-infinity*[*of at-left a* λ-. *c c*])
apply *auto*
using *at-top-le-at-infinity filterlim-mono that* **by** *blast*
ultimately show *?thesis* **unfolding** *jump-def* **by** *auto*
qed

lemma *jump-not-infinity*:
isCont f a ⟹ *jump f a* =*0*
by (*meson at-bot-le-at-infinity at-top-le-at-infinity filterlim-at-split*
filterlim-def isCont-def jump-def not-tendsto-and-filterlim-at-infinity
order-trans trivial-limit-at-left-real)

lemma *jump-jump-poly-aux*:
assumes *p*≠*0 coprime p q*
shows *jump* (λ*x*. *poly q x* / *poly p x*) *a* = *jump-poly q p a*
proof (*cases q*=*0*)

```
  case True
  then show ?thesis by auto
next
  case False
  define f where f ≡ (λx. poly q x / poly p x)
  have ?thesis when poly q a = 0
  proof −
    have poly p a≠0 using coprime-poly-0[OF ‹coprime p q›] that by blast
    then have isCont f a unfolding f-def by simp
    then have jump f a=0 using jump-not-infinity by auto
    moreover have jump-poly q p a=0
      using jump-poly-not-root[OF ‹poly p a≠0›] by auto
    ultimately show ?thesis unfolding f-def by auto
  qed
  moreover have ?thesis when poly q a≠0
  proof (cases even(order a p))
    case True
    define c where c≡sgn (poly q a)
    note
      filterlim-divide-at-bot-at-top-iff
        [OF - that,of poly q  at-left a poly p,folded f-def c-def,simplified]
      filterlim-divide-at-bot-at-top-iff
        [OF - that,of poly q  at-right a poly p,folded f-def c-def,simplified]
     moreover have (poly p has-sgnx − c) (at-left a) = (poly p has-sgnx − c)
(at-right a)
        (poly p has-sgnx c) (at-left a) = (poly p has-sgnx c) (at-right a)
      using poly-has-sgnx-left-right[OF ‹p≠0›] True by auto
    moreover have c≠0 by (simp add: c-def sgn-if that)
    then have False when
        (poly p has-sgnx − c) (at-right a)
        (poly p has-sgnx c) (at-right a)
      using has-sgnx-unique[OF - that] by auto
    ultimately have jump f a = 0
      unfolding jump-def by auto
    moreover have jump-poly q p a = 0 unfolding jump-poly-def
      using True by (simp add: order-0I that)
    ultimately show ?thesis unfolding f-def by auto
  next
    case False
    define c where c≡sgn (poly q a)
    have (poly p ⟶ 0) (at a) using False
      by (metis even-zero order-0I poly-tendsto(1))
    then have (poly p ⟶ 0) (at-left a) and (poly p ⟶ 0) (at-right a)
      by (auto simp add: filterlim-at-split)
    moreover note
      filterlim-divide-at-bot-at-top-iff
        [OF - that,of poly q - poly p,folded f-def c-def]
   moreover have (poly p has-sgnx c) (at-left a) = (poly p has-sgnx − c) (at-right
a)
```

```
      (poly p has-sgnx − c) (at-left a) = (poly p has-sgnx c) (at-right a)
    using poly-has-sgnx-left-right[OF ‹p≠0›] False by auto
  ultimately have jump f a = (if (poly p has-sgnx c) (at-right a) then 1
      else if (poly p has-sgnx − c) (at-right a) then −1 else 0)
    unfolding jump-def by auto
  also have ... = (if sign-r-pos (q ∗ p) a then 1 else − 1)
  proof −
    have (poly p has-sgnx c) (at-right a) ⟷ sign-r-pos (q ∗ p) a
    proof
      assume (poly p has-sgnx c) (at-right a)
      then have sgnx (poly p) (at-right a) = c by auto
      moreover have sgnx (poly q) (at-right a) = c
        unfolding c-def using that by (auto intro!: tendsto-nonzero-sgnx)
      ultimately have sgnx (λx. poly (q∗p) x) (at-right a) = c ∗ c
        by (simp add:sgnx-times)
      moreover have c≠0 by (simp add: c-def sgn-if that)
      ultimately have sgnx (λx. poly (q∗p) x) (at-right a) > 0
        using not-real-square-gt-zero by fastforce
      then show sign-r-pos (q ∗ p) a using sign-r-pos-sgnx-iff
        by blast
    next
      assume asm:sign-r-pos (q ∗ p) a
      let ?c1 = sgnx (poly p) (at-right a)
      let ?c2 = sgnx (poly q) (at-right a)
      have 0 < sgnx (λx. poly (q ∗ p) x) (at-right a)
        using asm sign-r-pos-sgnx-iff by blast
      then have ?c2 ∗ ?c1 >0
        apply (subst (asm) poly-mult)
        apply (subst (asm) sgnx-times)
        by auto
      then have ?c2>0 ∧ ?c1>0 ∨ ?c2<0 ∧ ?c1<0
        by (simp add: zero-less-mult-iff)
      then have ?c1=?c2
        using sgnx-values[OF sgnx-able-poly(1), of a,simplified]
        by (metis add.inverse-neutral less-minus-iff less-not-sym)
      moreover have sgnx (poly q) (at-right a) = c
        unfolding c-def using that by (auto intro!: tendsto-nonzero-sgnx)
      ultimately have ?c1 = c by auto
      then show (poly p has-sgnx c) (at-right a)
        using sgnx-able-poly(1) sgnx-able-sgnx by blast
    qed
    then show ?thesis
      unfolding jump-poly-def using poly-has-sgnx-values[OF ‹p≠0›]
      by (metis add.inverse-inverse c-def sgn-if that)
  qed
  also have ... = jump-poly q p a
   unfolding jump-poly-def using False order-root that by (simp add: order-root
assms(1))
  finally show ?thesis unfolding f-def by auto
```

qed
ultimately show *?thesis* **by** *auto*
qed

lemma *jump-jumpF*:
assumes *cont*:*isCont* (*inverse o f*) *a* **and**
sgnxl:(*f has-sgnx l*) (*at-left a*) **and** *sgnxr*:(*f has-sgnx r*) (*at-right a*) **and**
l≠*0* *r*≠*0*
shows *jump f a* = *jumpF f* (*at-right a*) − *jumpF f* (*at-left a*)
proof −
have *?thesis* **when** *filterlim f at-bot* (*at-left a*) *filterlim f at-top* (*at-right a*)
unfolding *jump-def jumpF-def*
using *that filterlim-at-top-at-bot*[*OF - - trivial-limit-at-left-real*]
by *auto*
moreover have *?thesis* **when** *filterlim f at-top* (*at-left a*) *filterlim f at-bot* (*at-right a*)
unfolding *jump-def jumpF-def*
using *that filterlim-at-top-at-bot*[*OF - - trivial-limit-at-right-real*]
by *auto*
moreover have *?thesis* **when**
¬ *filterlim f at-bot* (*at-left a*) ∨ ¬ *filterlim f at-top* (*at-right a*)
¬ *filterlim f at-top* (*at-left a*) ∨ ¬ *filterlim f at-bot* (*at-right a*)
proof (*cases f a*=*0*)
case *False*
have *jumpF f* (*at-right a*) = *0 jumpF f* (*at-left a*) = *0*
proof −
have *isCont* (*inverse o inverse o f*) *a* **using** *cont False* **unfolding** *comp-def*
by (*rule-tac continuous-at-within-inverse*, *auto*)
then have *isCont f a* **unfolding** *comp-def* **by** *auto*
then have (*f* ⟶ *f a*) (*at-right a*) (*f* ⟶ *f a*) (*at-left a*)
unfolding *continuous-at-split* **by** (*auto simp add*:*continuous-within*)
moreover note *trivial-limit-at-left-real trivial-limit-at-right-real*
ultimately show *jumpF f* (*at-right a*) = *0 jumpF f* (*at-left a*) = *0*
unfolding *jumpF-def* **using** *filterlim-at-bot-nhds filterlim-at-top-nhds*
by *metis*+
qed
then show *?thesis* **unfolding** *jump-def* **using** *that* **by** *auto*
next
case *True*
then have *tends0*:((λ*x*. *inverse* (*f x*)) ⟶ *0*) (*at a*)
using *cont* **unfolding** *isCont-def comp-def* **by** *auto*
have *jump f a* = *0* **using** *that* **unfolding** *jump-def* **by** *auto*
have *r-lim*:*if r*>*0 then filterlim f at-top* (*at-right a*) *else filterlim f at-bot* (*at-right a*)
proof (*cases r*>*0*)
case *True*
then have $\forall_F$ *x in* (*at-right a*). *0* < *f x*
using *sgnxr* **unfolding** *has-sgnx-def*
by (*auto elim*:*eventually-mono*)

then have *filterlim f at-top* (*at-right a*)
using *filterlim-inverse-at-top*[*of* λ*x. inverse* (*f x*), *simplified*] *tends0*
unfolding *filterlim-at-split* **by** *auto*
then show *?thesis* **using** *True* **by** *presburger*
next
case *False*
then have ∀$_F$ *x in* (*at-right a*). *0* > *f x*
using *sgnxr* ‹*r*≠*0*› *False* **unfolding** *has-sgnx-def*
apply (*elim eventually-mono*)
by (*meson linorder-neqE-linordered-idom sgn-less*)
then have *filterlim f at-bot* (*at-right a*)
using *filterlim-inverse-at-bot*[*of* λ*x. inverse* (*f x*), *simplified*] *tends0*
unfolding *filterlim-at-split* **by** *auto*
then show *?thesis* **using** *False* **by** *simp*
qed
have *l-lim:if l*>*0 then filterlim f at-top* (*at-left a*) *else filterlim f at-bot* (*at-left a*)
proof (*cases l*>*0*)
case *True*
then have ∀$_F$ *x in* (*at-left a*). *0* < *f x*
using *sgnxl* **unfolding** *has-sgnx-def*
by (*auto elim:eventually-mono*)
then have *filterlim f at-top* (*at-left a*)
using *filterlim-inverse-at-top*[*of* λ*x. inverse* (*f x*), *simplified*] *tends0*
unfolding *filterlim-at-split* **by** *auto*
then show *?thesis* **using** *True* **by** *presburger*
next
case *False*
then have ∀$_F$ *x in* (*at-left a*). *0* > *f x*
using *sgnxl* ‹*l*≠*0*› *False* **unfolding** *has-sgnx-def*
apply (*elim eventually-mono*)
by (*meson linorder-neqE-linordered-idom sgn-less*)
then have *filterlim f at-bot* (*at-left a*)
using *filterlim-inverse-at-bot*[*of* λ*x. inverse* (*f x*), *simplified*] *tends0*
unfolding *filterlim-at-split* **by** *auto*
then show *?thesis* **using** *False* **by** *simp*
qed

have *?thesis* **when** *l*>*0 r*>*0*
using *that l-lim r-lim* ‹*jump f a*=*0*› **unfolding** *jumpF-def* **by** *auto*
moreover have *?thesis* **when** ¬ *l*>*0* ¬ *r*>*0*
proof −
have *filterlim f at-bot* (*at-right a*) *filterlim f at-bot* (*at-left a*)
using *r-lim l-lim that* **by** *auto*
moreover then have ¬ *filterlim f at-top* (*at-right a*) ¬ *filterlim f at-top* (*at-left a*)
by (*auto elim*: *filterlim-at-top-at-bot*)
ultimately have *jumpF f* (*at-right a*) = −*1/2* *jumpF f* (*at-left a*) = −*1/2*
unfolding *jumpF-def* **by** *auto*

then show *?thesis* **using** ‹*jump f a=0*› **by** *auto*
qed
moreover have *?thesis* **when** *l>0* ¬ *r>0*
proof −
note ‹¬ *filterlim f at-top* (*at-left a*) ∨ ¬ *filterlim f at-bot* (*at-right a*)›
moreover have *filterlim f at-bot* (*at-right a*) *filterlim f at-top* (*at-left a*)
using *r-lim l-lim that* **by** *auto*
ultimately have *False* **by** *auto*
then show *?thesis* **by** *auto*
qed
moreover have *?thesis* **when** ¬ *l>0 r>0*
proof −
note ‹¬ *filterlim f at-bot* (*at-left a*) ∨ ¬ *filterlim f at-top* (*at-right a*)›
moreover have *filterlim f at-bot* (*at-left a*) *filterlim f at-top* (*at-right a*)
using *r-lim l-lim that* **by** *auto*
ultimately have *False* **by** *auto*
then show *?thesis* **by** *auto*
qed
ultimately show *?thesis* **by** *auto*
qed
ultimately show *?thesis* **by** *auto*
qed

lemma *jump-linear-comp*:
assumes *c≠0*
shows *jump* (*f o* (*λx. c∗x+b*)) *x* = (*if c>0 then jump f* (*c∗x+b*) *else* −*jump f* (*c∗x+b*))
proof (*cases c>0*)
case *False*
then have *c<0* **using** ‹*c≠0*› **by** *auto*
let *?g* = *λx. c∗x+b*
have *filtermap ?g* (*at-left x*) = *at-right* (*?g x*)
filtermap ?g (*at-right x*) = *at-left* (*?g x*)
using ‹*c<0*›
filtermap-linear-at-left[*OF* ‹*c≠0*›, *of b x*]
filtermap-linear-at-right[*OF* ‹*c≠0*›, *of b x*] **by** *auto*
then have *jump* (*f* ∘ *?g*) *x* = − *jump f* (*c* ∗ *x* + *b*)
unfolding *jump-def filterlim-def comp-def*
apply (*auto simp add*: *filtermap-filtermap*[*of f ?g*,*symmetric*])
apply (*fold filterlim-def*)
by (*auto elim*:*filterlim-at-top-at-bot*)
then show *?thesis* **using** ‹*c<0*› **by** *auto*
next
case *True*
let *?g* = *λx. c∗x+b*
have *filtermap ?g* (*at-left x*) = *at-left* (*?g x*)
filtermap ?g (*at-right x*) = *at-right* (*?g x*)
using *True*
filtermap-linear-at-left[*OF* ‹*c≠0*›, *of b x*]

```
      filtermap-linear-at-right[OF ‹c≠0›, of b x] by auto
  then have jump (f ∘ ?g) x = jump f (c ∗ x + b)
    unfolding jump-def filterlim-def comp-def
    by (auto simp add: filtermap-filtermap[of f ?g,symmetric])
  then show ?thesis using True by auto
qed

lemma jump-divide-derivative:
  assumes isCont f x g x = 0 f x≠0
    and g-deriv:(g has-field-derivative c) (at x) and c≠0
  shows jump (λt. f t/g t) x = (if sgn c = sgn ( f x) then 1 else −1)
proof −
  have g-tendsto:(g ⟶ 0) (at-left x) (g ⟶ 0) (at-right x)
    by (metis DERIV-isCont Lim-at-imp-Lim-at-within assms(2) assms(4) contin-
uous-at)+
  have f-tendsto:(f ⟶ f x) (at-left x) (f ⟶ f x) (at-right x)
    using Lim-at-imp-Lim-at-within assms(1) continuous-at by blast+

  have ?thesis when c>0 f x>0
  proof −
    have (g has-sgnx − sgn (f x)) (at-left x)
      using has-sgnx-derivative-at-left[OF g-deriv ‹g x=0›] that by auto
    moreover have (g has-sgnx sgn (f x)) (at-right x)
      using has-sgnx-derivative-at-right[OF g-deriv ‹g x=0›] that by auto
    ultimately have (LIM t at-left x. f t / g t :> at-bot) ∧ (LIM t at-right x. f t
/ g t :> at-top)
      using filterlim-divide-at-bot-at-top-iff[OF - ‹f x≠0›, of f]
      using f-tendsto(1) f-tendsto(2) g-tendsto(1) g-tendsto(2) by blast
    moreover have sgn c = sgn (f x) using that by auto
    ultimately show ?thesis unfolding jump-def by auto
  qed
  moreover have ?thesis when c>0 f x<0
  proof −
    have (g has-sgnx sgn (f x)) (at-left x)
      using has-sgnx-derivative-at-left[OF g-deriv ‹g x=0›] that by auto
    moreover have (g has-sgnx − sgn (f x)) (at-right x)
      using has-sgnx-derivative-at-right[OF g-deriv ‹g x=0›] that by auto
    ultimately have (LIM t at-left x. f t / g t :> at-top) ∧ (LIM t at-right x. f t
/ g t :> at-bot)
      using filterlim-divide-at-bot-at-top-iff[OF - ‹f x≠0›, of f]
      using f-tendsto(1) f-tendsto(2) g-tendsto(1) g-tendsto(2) by blast
    moreover from this have ¬ (LIM t at-left x. f t / g t :> at-bot)
      using filterlim-at-top-at-bot by fastforce
    moreover have sgn c ≠ sgn (f x) using that by auto
    ultimately show ?thesis unfolding jump-def by auto
  qed
  moreover have ?thesis when c<0 f x>0
  proof −
    have (g has-sgnx sgn (f x)) (at-left x)
```

using *has-sgnx-derivative-at-left*[*OF g-deriv* ‹*g x=0*›] *that* **by** *auto*
moreover have (*g has-sgnx* − *sgn* (*f x*)) (*at-right x*)
using *has-sgnx-derivative-at-right*[*OF g-deriv* ‹*g x=0*›] *that* **by** *auto*
ultimately have (*LIM t at-left x. f t* / *g t* :> *at-top*) ∧ (*LIM t at-right x. f t* / *g t* :> *at-bot*)
using *filterlim-divide-at-bot-at-top-iff*[*OF -* ‹*f x*≠*0*›, *of f*]
using *f-tendsto*(*1*) *f-tendsto*(*2*) *g-tendsto*(*1*) *g-tendsto*(*2*) **by** *blast*
moreover from *this* **have** ¬ (*LIM t at-left x. f t* / *g t* :> *at-bot*)
using *filterlim-at-top-at-bot* **by** *fastforce*
moreover have *sgn c* ≠ *sgn* (*f x*) **using** *that* **by** *auto*
ultimately show *?thesis* **unfolding** *jump-def* **by** *auto*
qed
moreover have *?thesis* **when** *c*<*0 f x*<*0*
proof −
have (*g has-sgnx* − *sgn* (*f x*)) (*at-left x*)
using *has-sgnx-derivative-at-left*[*OF g-deriv* ‹*g x=0*›] *that* **by** *auto*
moreover have (*g has-sgnx sgn* (*f x*)) (*at-right x*)
using *has-sgnx-derivative-at-right*[*OF g-deriv* ‹*g x=0*›] *that* **by** *auto*
ultimately have (*LIM t at-left x. f t* / *g t* :> *at-bot*) ∧ (*LIM t at-right x. f t* / *g t* :> *at-top*)
using *filterlim-divide-at-bot-at-top-iff*[*OF -* ‹*f x*≠*0*›, *of f*]
using *f-tendsto*(*1*) *f-tendsto*(*2*) *g-tendsto*(*1*) *g-tendsto*(*2*) **by** *blast*
moreover have *sgn c* =*sgn* (*f x*) **using** *that* **by** *auto*
ultimately show *?thesis* **unfolding** *jump-def* **by** *auto*
qed
ultimately show *?thesis* **using** ‹*c*≠*0*› ‹*f x*≠*0*› **by** *argo*
qed

lemma *jump-jump-poly*: *jump* (λ*x. poly q x* / *poly p x*) *a* = *jump-poly q p a*
proof (*cases p=0*)
case *True*
then show *?thesis* **by** *auto*
next
case *False*
obtain *p*′ *q*′ **where** *p*′:*p*= *p*′∗*gcd p q* **and** *q*′:*q*=*q*′∗*gcd p q*
using *gcd-dvd1 gcd-dvd2 dvd-def*[*of gcd p q*, *simplified mult.commute*] **by** *metis*
then have *coprime p*′ *q*′ *p*′≠*0 gcd p q*≠*0* **using** *gcd-coprime* ‹*p*≠*0*› **by** *auto*

define *f* **where** *f* ≡ (λ*x. poly q*′ *x* / *poly p*′ *x*)
define *g* **where** *g* ≡ (λ*x. if poly* (*gcd p q*) *x* = *0 then 0*::*real else 1*)

have *g-tendsto*:(*g* ⟶ *1*) (*at-left a*) (*g* ⟶ *1*) (*at-right a*)
proof −
have
(*poly* (*gcd p q*) *has-sgnx 1*) (*at-left a*)
∨ (*poly* (*gcd p q*) *has-sgnx* − *1*) (*at-left a*)
(*poly* (*gcd p q*) *has-sgnx 1*) (*at-right a*)
∨ (*poly* (*gcd p q*) *has-sgnx* − *1*) (*at-right a*)
using ‹*p*≠*0*› *poly-has-sgnx-values* **by** *auto*

```
    then have ∀F x in at-left a. g x = 1  ∀F x in at-right a. g x = 1
      unfolding has-sgnx-def g-def by (auto elim:eventually-mono)
    then show (g ⟶ 1) (at-left a) (g ⟶ 1) (at-right a)
      using tendsto-eventually by auto
  qed

  have poly q x / poly p x = g x * f x for x
    unfolding f-def g-def by (subst p',subst q',auto)
  then have jump (λx. poly q x / poly p x) a = jump (λx. g x * f x) a
    by auto
  also have ... = jump f a
    unfolding jump-def
    apply (subst (1 2) filterlim-tendsto-pos-mult-at-top-iff)
        prefer 5
        apply (subst (1 2) filterlim-tendsto-pos-mult-at-bot-iff)
    using g-tendsto by auto
  also have ... = jump-poly q' p' a
   using jump-jump-poly-aux[OF ‹p'≠0› ‹coprime p' q'›] unfolding f-def by auto
  also have ... = jump-poly q p a
    using jump-poly-mult[OF ‹gcd p q ≠ 0›, of q'] p' q'
    by (metis mult.commute)
  finally show ?thesis .
qed


lemma jump-Im-divide-Re-0:
  assumes path g Re (g x)≠0 0<x x<1
  shows jump (λt. Im (g t) / Re (g t)) x = 0
proof -
  have isCont g x
    using ‹path g›[unfolded path-def] ‹0<x› ‹x<1›
    apply (elim continuous-on-interior)
    by auto
  then have isCont (λt. Im(g t)/Re(g t)) x using ‹Re (g x)≠0›
    by (auto intro:continuous-intros isCont-Re isCont-Im)
  then show jump (λt. Im(g t)/Re(g t)) x=0
    using jump-not-infinity by auto
qed

lemma jumpF-im-divide-Re-0:
  assumes path g Re (g x)≠0
  shows ⟦0≤x;x<1⟧ ⟹ jumpF (λt. Im (g t) / Re (g t)) (at-right x) = 0
        ⟦0<x;x≤1⟧ ⟹ jumpF (λt. Im (g t) / Re (g t)) (at-left x) = 0
proof -
  define g' where g' = (λt. Im (g t) / Re (g t))

  show jumpF g' (at-right x) = 0 when 0≤x x<1
  proof -
    have (g' ⟶ g' x) (at-right x)
```

proof (*cases x=0*)
case *True*
have *continuous* (*at-right 0*) *g*
using ‹*path g*› **unfolding** *path-def*
by (*auto elim*:*continuous-on-at-right*)
then have *continuous* (*at-right x*) (λ*t*. *Im*(*g t*)) *continuous* (*at-right x*) (λ*t*. *Re*(*g t*))
using *continuous-Im continuous-Re True* **by** *auto*
moreover have *Re* (*g* (*netlimit* (*at-right x*))) ≠ *0*
using *assms*(*2*) **by** (*simp add*: *Lim-ident-at*)
ultimately have *continuous* (*at-right x*) (λ*t*. *Im* (*g t*)/*Re*(*g t*))
by (*auto intro*:*continuous-divide*)
then show *?thesis* **unfolding** *g′-def continuous-def*
by (*simp add*: *Lim-ident-at*)
next
case *False*
have *isCont* (λ*x*. *Im* (*g x*)) *x isCont* (λ*x*. *Re* (*g x*)) *x*
using ‹*path g*› **unfolding** *path-def*
by (*metis False atLeastAtMost-iff at-within-Icc-at continuous-Im continuous-Re*
continuous-on-eq-continuous-within less-le that)+
then have *isCont g′ x*
using *assms*(*2*) **unfolding** *g′-def*
by (*auto intro*:*continuous-intros*)
then show *?thesis* **unfolding** *isCont-def* **using** *filterlim-at-split* **by** *blast*
qed
then have ¬ *filterlim g′ at-top* (*at-right x*) ¬ *filterlim g′ at-bot* (*at-right x*)
using *filterlim-at-top-nhds*[*of g′ at-right x*] *filterlim-at-bot-nhds*[*of g′ at-right x*]
by *auto*
then show *?thesis* **unfolding** *jumpF-def* **by** *auto*
qed

show *jumpF g′* (*at-left x*) = *0* **when** *0<x x≤1*
proof −
have (*g′* ⟶ *g′ x*) (*at-left x*)
proof (*cases x=1*)
case *True*
have *continuous* (*at-left 1*) *g*
using ‹*path g*› **unfolding** *path-def*
by (*auto elim*:*continuous-on-at-left*)
then have *continuous* (*at-left x*) (λ*t*. *Im*(*g t*)) *continuous* (*at-left x*) (λ*t*. *Re*(*g t*))
using *continuous-Im continuous-Re True* **by** *auto*
moreover have *Re* (*g* (*netlimit* (*at-left x*))) ≠ *0*
using *assms*(*2*) **by** (*simp add*: *Lim-ident-at*)
ultimately have *continuous* (*at-left x*) (λ*t*. *Im* (*g t*)/*Re*(*g t*))
by (*auto intro*:*continuous-divide*)
then show *?thesis* **unfolding** *g′-def continuous-def*

by (*simp add*: *Lim-ident-at*)
next
case *False*
have *isCont* (*λx. Im* (*g x*)) *x isCont* (*λx. Re* (*g x*)) *x*
using ‹*path g*› **unfolding** *path-def*
by (*metis False atLeastAtMost-iff at-within-Icc-at continuous-Im continuous-Re*
continuous-on-eq-continuous-within less-le that)+
then have *isCont g′ x*
using *assms*(*2*) **unfolding** *g′-def*
by (*auto*)
then show *?thesis* **unfolding** *isCont-def* **using** *filterlim-at-split* **by** *blast*
qed
then have ¬ *filterlim g′ at-top* (*at-left x*) ¬ *filterlim g′ at-bot* (*at-left x*)
using *filterlim-at-top-nhds*[*of g′ at-left x*] *filterlim-at-bot-nhds*[*of g′ at-left x*]
by *auto*
then show *?thesis* **unfolding** *jumpF-def* **by** *auto*
qed
qed

lemma *jump-cong*:
assumes *x=y* **and** *eventually* (*λx. f x=g x*) (*at x*)
shows *jump f x* = *jump g y*
proof −
have *left*:*eventually* (*λx. f x=g x*) (*at-left x*)
and *right*:*eventually* (*λx. f x=g x*) (*at-right x*)
using *assms*(*2*) *eventually-at-split* **by** *blast+*
from *filterlim-cong*[*OF - - this*(*1*)] *filterlim-cong*[*OF - - this*(*2*)]
show *?thesis* **unfolding** *jump-def* **using** *assms*(*1*) **by** *fastforce*
qed

lemma *jumpF-cong*:
assumes *F=G* **and** *eventually* (*λx. f x=g x*) *F*
shows *jumpF f F* = *jumpF g G*
proof −
have $\forall_F$ *r in G. f r* = *g r*
using *assms*(*1*) *assms*(*2*) **by** *force*
then show *?thesis*
by (*simp add*: *assms*(*1*) *filterlim-cong jumpF-def*)
qed

lemma *jump-at-left-at-right-eq*:
assumes *isCont f x* **and** *f x* ≠ *0* **and** *sgnx-eq*:*sgnx g* (*at-left x*) = *sgnx g* (*at-right x*)
shows *jump* (*λt. f t/g t*) *x* = *0*
proof −
define *c* **where** *c* = *sgn* (*f x*)
then have *c≠0* **using** ‹*f x≠0*› **by** (*simp add*: *sgn-zero-iff*)
have *f-tendsto*:(*f* ⟶ *f x*) (*at-left x*) (*f* ⟶ *f x*) (*at-right x*)

using ‹*isCont f x*› *Lim-at-imp-Lim-at-within isCont-def* **by** *blast+*
have *False* **when** (*g has-sgnx* − *c*) (*at-left x*) (*g has-sgnx c*) (*at-right x*)
proof −
have *sgnx g* (*at-left x*) = −*c* **using** *that*(*1*) **by** *auto*
moreover have *sgnx g* (*at-right x*) = *c* **using** *that*(*2*) **by** *auto*
ultimately show *False* **using** *sgnx-eq* ‹*c*≠*0*› **by** *force*
qed
moreover have *False* **when** (*g has-sgnx c*) (*at-left x*) (*g has-sgnx* − *c*) (*at-right x*)
proof −
have *sgnx g* (*at-left x*) = *c* **using** *that*(*1*) **by** *auto*
moreover have *sgnx g* (*at-right x*) = − *c* **using** *that*(*2*) **by** *auto*
ultimately show *False* **using** *sgnx-eq* ‹*c*≠*0*› **by** *force*
qed
ultimately show *?thesis*
unfolding *jump-def*
by (*auto simp add*:*f-tendsto filterlim-divide-at-bot-at-top-iff*[*OF* - ‹*f x* ≠ *0*›] *c-def*)
qed

lemma *jumpF-pos-has-sgnx*:
assumes *jumpF f F* > *0*
shows (*f has-sgnx 1*) *F*
proof −
have *filterlim f at-top F* **using** *assms* **unfolding** *jumpF-def* **by** *argo*
then have *eventually* (λ*x*. *f x*>*0*) *F* **using** *filterlim-at-top-dense*[*of f F*] **by** *blast*
then show *?thesis* **unfolding** *has-sgnx-def*
apply (*elim eventually-mono*)
by *auto*
qed

lemma *jumpF-neg-has-sgnx*:
assumes *jumpF f F* < *0*
shows (*f has-sgnx* −*1*) *F*
proof −
have *filterlim f at-bot F* **using** *assms* **unfolding** *jumpF-def* **by** *argo*
then have *eventually* (λ*x*. *f x*<*0*) *F* **using** *filterlim-at-bot-dense*[*of f F*] **by** *blast*
then show *?thesis* **unfolding** *has-sgnx-def*
apply (*elim eventually-mono*)
by *auto*
qed

lemma *jumpF-IVT*:
fixes *f*::*real* ⇒ *real* **and** *a b*::*real*
defines *right*≡(λ(*R*::*real* ⇒ *real* ⇒ *bool*). *R* (*jumpF f* (*at-right a*)) *0*
∨ (*continuous* (*at-right a*) *f* ∧ *R* (*f a*) *0*))
and
left≡(λ(*R*::*real* ⇒ *real* ⇒ *bool*). *R* (*jumpF f* (*at-left b*)) *0*

∨ (*continuous* (*at-left* *b*) *f* ∧ *R* (*f* *b*) *0*))
assumes *a*<*b* **and** *cont*:*continuous-on* {*a*<..<*b*} *f* **and**
right-left:*right greater* ∧ *left less* ∨ *right less* ∧ *left greater*
shows ∃ *x*. *a*<*x* ∧ *x*<*b* ∧ *f* *x* =*0*
proof −
have *?thesis* **when** *right greater left less*
proof −
have (*f has-sgnx 1*) (*at-right a*)
proof −
have *?thesis* **when** *jumpF f* (*at-right a*)>*0* **using** *jumpF-pos-has-sgnx*[*OF that*] **.**
moreover have *?thesis* **when** *f a* > *0 continuous* (*at-right a*) *f*
proof −
have (*f* ⟶ *f a*) (*at-right a*) **using** *that*(*2*) **by** (*simp add*: *continuous-within*)
then show *?thesis*
using *tendsto-nonzero-has-sgnx*[*of f f a at-right a*] *that* **by** *auto*
qed
ultimately show *?thesis* **using** *that*(*1*) **unfolding** *right-def* **by** *auto*
qed
then obtain *a′* **where** *a*<*a′* **and** *a′-def*:∀ *y*. *a*<*y* ∧ *y* < *a′* ⟶ *f y* > *0*
unfolding *has-sgnx-def eventually-at-right* **using** *sgn-1-pos* **by** *auto*
have (*f has-sgnx* − *1*) (*at-left b*)
proof −
have *?thesis* **when** *jumpF f* (*at-left b*)<*0* **using** *jumpF-neg-has-sgnx*[*OF that*] **.**
moreover have *?thesis* **when** *f b* < *0 continuous* (*at-left b*) *f*
proof −
have (*f* ⟶ *f b*) (*at-left b*)
using *that*(*2*) **by** (*simp add*: *continuous-within*)
then show *?thesis*
using *tendsto-nonzero-has-sgnx*[*of f f b at-left b*] *that* **by** *auto*
qed
ultimately show *?thesis* **using** *that*(*2*) **unfolding** *left-def* **by** *auto*
qed
then obtain *b′* **where** *b′*<*b* **and** *b′-def*:∀ *y*. *b′*<*y* ∧ *y* < *b* ⟶ *f y* < *0*
unfolding *has-sgnx-def eventually-at-left* **using** *sgn-1-neg* **by** *auto*
have *a′* ≤ *b′*
proof (*rule ccontr*)
assume ¬ *a′* ≤ *b′*
then have {*a*<..<*a′*} ∩ {*b′*<..<*b*} ≠ {}
using ‹*a*<*a′*› ‹*b′*<*b*› ‹*a*<*b*› **by** *auto*
then obtain *c* **where** *c*∈{*a*<..<*a′*} *c*∈{*b′*<..<*b*} **by** *blast*
then have *f c*>*0* *f c*<*0*
using *a′-def b′-def* **by** *auto*
then show *False* **by** *auto*
qed
define *a0* **where** *a0*=(*a*+*a′*)/*2*
define *b0* **where** *b0*=(*b*+*b′*)/*2*

have [*simp*]:*a*<*a0* *a0*<*a′* *a0*<*b0* *b′*<*b0* *b0*<*b*
unfolding *a0-def b0-def* **using** ‹*a*<*a′*› ‹*b′*<*b*› ‹*a′*≤*b′*› **by** *auto*
have *f a0*>*0 f b0*<*0* **using** *a′-def*[*rule-format*,*of a0*] *b′-def*[*rule-format*,*of b0*]
by *auto*
moreover have *continuous-on* {*a0*..*b0*} *f*
using *cont* ‹*a* < *a0*› ‹*b0* < *b*›
by (*meson atLeastAtMost-subseteq-greaterThanLessThan-iff continuous-on-subset*)
ultimately have ∃ *x*>*a0*. *x* < *b0* ∧ *f x* = *0*
using *IVT-strict*[*of 0 f a0 b0*] **by** *auto*
then show *?thesis* **using** ‹*a* < *a0*› ‹*b0* < *b*›
by (*meson lessThan-strict-subset-iff psubsetE subset-psubset-trans*)
qed
moreover have *?thesis* **when** *right less left greater*
proof −
have (*f has-sgnx* −*1*) (*at-right a*)
proof −
have *?thesis* **when** *jumpF f* (*at-right a*)<*0* **using** *jumpF-neg-has-sgnx*[*OF*
that] **.**
moreover have *?thesis* **when** *f a* < *0 continuous* (*at-right a*) *f*
proof −
have (*f* ⟶ *f a*) (*at-right a*)
using *that*(*2*) **by** (*simp add*: *continuous-within*)
then show *?thesis*
using *tendsto-nonzero-has-sgnx*[*of f f a at-right a*] *that* **by** *auto*
qed
ultimately show *?thesis* **using** *that*(*1*) **unfolding** *right-def* **by** *auto*
qed
then obtain *a′* **where** *a*<*a′* **and** *a′-def*:∀ *y*. *a*<*y* ∧ *y* < *a′* ⟶ *f y* < *0*
unfolding *has-sgnx-def eventually-at-right* **using** *sgn-1-neg* **by** *auto*
have (*f has-sgnx 1*) (*at-left b*)
proof −
have *?thesis* **when** *jumpF f* (*at-left b*)>*0* **using** *jumpF-pos-has-sgnx*[*OF that*]
.
moreover have *?thesis* **when** *f b* > *0 continuous* (*at-left b*) *f*
proof −
have (*f* ⟶ *f b*) (*at-left b*)
using *that*(*2*) **by** (*simp add*: *continuous-within*)
then show *?thesis*
using *tendsto-nonzero-has-sgnx*[*of f f b at-left b*] *that* **by** *auto*
qed
ultimately show *?thesis* **using** *that*(*2*) **unfolding** *left-def* **by** *auto*
qed
then obtain *b′* **where** *b′*<*b* **and** *b′-def*:∀ *y*. *b′*<*y* ∧ *y* < *b* ⟶ *f y* > *0*
unfolding *has-sgnx-def eventually-at-left* **using** *sgn-1-pos* **by** *auto*
have *a′* ≤ *b′*
proof (*rule ccontr*)
assume ¬ *a′* ≤ *b′*
then have {*a*<..<*a′*} ∩ {*b′*<..<*b*} ≠ {}
using ‹*a*<*a′*› ‹*b′*<*b*› ‹*a*<*b*› **by** *auto*

then obtain *c* **where** *c*∈{*a*<..<*a'*} *c*∈{*b'*<..<*b*} **by** *blast*
then have *f c*>*0* *f c*<*0*
using *a'-def* *b'-def* **by** *auto*
then show *False* **by** *auto*
qed
define *a0* **where** *a0*=(*a*+*a'*)/*2*
define *b0* **where** *b0*=(*b*+*b'*)/*2*
have [*simp*]:*a*<*a0* *a0*<*a'* *a0*<*b0* *b'*<*b0* *b0*<*b*
unfolding *a0-def* *b0-def* **using** ‹*a*<*a'*› ‹*b'*<*b*› ‹*a'*≤*b'*› **by** *auto*
have *f a0*<*0* *f b0*>*0* **using** *a'-def*[*rule-format*,*of a0*] *b'-def*[*rule-format*,*of b0*]
by *auto*
moreover have *continuous-on* {*a0*..*b0*} *f*
using *cont* ‹*a* < *a0*› ‹*b0* < *b*›
by (*meson atLeastAtMost-subseteq-greaterThanLessThan-iff continuous-on-subset*)

ultimately have ∃ *x*>*a0*. *x* < *b0* ∧ *f x* = *0*
using *IVT-strict*[*of 0 f a0 b0*] **by** *auto*
then show *?thesis* **using** ‹*a* < *a0*› ‹*b0* < *b*›
by (*meson lessThan-strict-subset-iff psubsetE subset-psubset-trans*)
qed
ultimately show *?thesis* **using** *right-left* **by** *auto*
qed

lemma *jumpF-eventually-const*:
assumes *eventually* (λ*x*. *f x*=*c*) *F F*≠*bot*
shows *jumpF f F* = *0*
proof −
have *jumpF f F* = *jumpF* (λ-. *c*) *F*
apply (*rule jumpF-cong*)
using *assms*(*1*) **by** *auto*
also have ... = *0* **using** *jumpF-const*[*OF* ‹*F*≠*bot*›] **by** *simp*
finally show *?thesis* **.**
qed

lemma *jumpF-tan-comp*:
jumpF (*f o tan*) (*at-right x*) = (*if cos x* = *0*
then jumpF f at-bot else jumpF f (*at-right* (*tan x*)))
jumpF (*f o tan*) (*at-left x*) = (*if cos x* =*0*
then jumpF f at-top else jumpF f (*at-left* (*tan x*)))
proof −
have *filtermap* (*f* ∘ *tan*) (*at-right x*) =
(*if cos x* = *0 then filtermap f at-bot else filtermap f* (*at-right* (*tan x*)))
unfolding *comp-def*
apply (*subst filtermap-filtermap*[*of f tan*,*symmetric*])
using *filtermap-tan-at-right-inf filtermap-tan-at-right* **by** *auto*
then show *jumpF* (*f o tan*) (*at-right x*) = (*if cos x* = *0*
then jumpF f at-bot else jumpF f (*at-right* (*tan x*)))
unfolding *jumpF-def filterlim-def* **by** *auto*
next

have *filtermap* (*f* ∘ *tan*) (*at-left x*) =
(*if cos x* = *0 then filtermap f at-top else filtermap f* (*at-left* (*tan x*)))
unfolding *comp-def*
apply (*subst filtermap-filtermap*[*of f tan*,*symmetric*])
using *filtermap-tan-at-left-inf filtermap-tan-at-left* **by** *auto*
then show *jumpF* (*f o tan*) (*at-left x*) = (*if cos x* = *0*
then jumpF f at-top else jumpF f (*at-left* (*tan x*)))
unfolding *jumpF-def filterlim-def* **by** *auto*
qed

5.6 Finite jumpFs over an interval

definition *finite-jumpFs*::(*real* ⇒ *real*) ⇒ *real* ⇒ *real* ⇒ *bool* **where**
finite-jumpFs f a b = *finite* {*x*. (*jumpF f* (*at-left x*) ≠*0* ∨ *jumpF f* (*at-right x*) ≠*0*) ∧ *a*≤*x* ∧ *x*≤*b*}

lemma *finite-jumpFs-linear-pos*:
assumes *c*>*0*
shows *finite-jumpFs* (*f o* (λ*x*. *c* ∗ *x* + *b*)) *lb ub* ⟷ *finite-jumpFs f* (*c* ∗ *lb* +*b*) (*c* ∗ *ub* + *b*)
proof −
define *left* **where** *left* = (λ*f lb ub*. {*x*. *jumpF f* (*at-left x*) ≠ *0* ∧ *lb* ≤ *x* ∧ *x* ≤ *ub*})
define *right* **where** *right* = (λ*f lb ub*. {*x*. *jumpF f* (*at-right x*) ≠ *0* ∧ *lb* ≤ *x* ∧ *x* ≤ *ub*})
define *g* **where** *g*=(λ*x*. *c*∗*x*+*b*)
define *gi* **where** *gi* = (λ*x*. (*x*−*b*)/*c*)
have *finite-jumpFs* (*f o* (λ*x*. *c* ∗ *x* + *b*)) *lb ub*
= *finite* (*left* (*f o g*) *lb ub* ∪ *right* (*f o g*) *lb ub*)
unfolding *finite-jumpFs-def*
apply (*rule arg-cong*[**where** *f*=*finite*])
by (*auto simp add*:*left-def right-def g-def*)
also have ... = *finite* (*gi* ‘ (*left f* (*g lb*) (*g ub*) ∪ *right f* (*g lb*) (*g ub*)))
proof −
have *j-rw*:
jumpF (*f o g*) (*at-left x*) = *jumpF f* (*at-left* (*g x*))
jumpF (*f o g*) (*at-right x*) = *jumpF f* (*at-right* (*g x*))
for *x*
using *jumpF-linear-comp*[*of c f b x*] ‹*c*>*0*› **unfolding** *g-def* **by** *auto*
then have
left (*f o g*) *lb ub* = *gi* ‘ *left f* (*g lb*) (*g ub*)
right (*f o g*) *lb ub* = *gi* ‘ *right f* (*g lb*) (*g ub*)
unfolding *left-def right-def gi-def*
using ‹*c*>*0*› **by** (*auto simp add*:*g-def field-simps*)
then have *left* (*f o g*) *lb ub* ∪ *right* (*f o g*) *lb ub*
= *gi* ‘ (*left f* (*g lb*) (*g ub*) ∪ *right f* (*g lb*) (*g ub*))
by *auto*
then show *?thesis* **by** *auto*
qed

also have ... = *finite* (*left f* (*g lb*) (*g ub*) ∪ *right f* (*g lb*) (*g ub*))
apply (*rule finite-image-iff*)
unfolding *gi-def* **using** ‹*c* >*0*› *inj-on-def* **by** *fastforce*
also have ... = *finite-jumpFs f* (*c* ∗ *lb* +*b*) (*c* ∗ *ub* + *b*)
unfolding *finite-jumpFs-def*
apply (*rule arg-cong*[**where** *f*=*finite*])
by (*auto simp add*:*left-def right-def g-def*)
finally show *?thesis* **.**
qed

lemma *finite-jumpFs-consts*:
finite-jumpFs (λ- . *c*) *lb ub*
unfolding *finite-jumpFs-def* **using** *jumpF-const* **by** *auto*

lemma *finite-jumpFs-combine*:
assumes *finite-jumpFs f a b finite-jumpFs f b c*
shows *finite-jumpFs f a c*
proof −
define *P* **where** *P*=(λ*x*. *jumpF f* (*at-left x*) ≠ *0* ∨ *jumpF f* (*at-right x*) ≠ *0*)
have {*x*. *P x* ∧ *a* ≤ *x* ∧ *x* ≤ *c*} ⊆ {*x*. *P x* ∧ *a* ≤ *x* ∧ *x*≤*b*} ∪ {*x*. *P x* ∧ *b* ≤*x* ∧ *x*≤*c*}
by *auto*
moreover have *finite* ({*x*. *P x* ∧ *a* ≤ *x* ∧ *x*≤*b*} ∪ {*x*. *P x* ∧ *b* ≤*x* ∧ *x*≤*c*})
using *assms* **unfolding** *finite-jumpFs-def P-def* **by** *auto*
ultimately have *finite* {*x*. *P x* ∧ *a* ≤ *x* ∧ *x* ≤ *c*}
using *finite-subset* **by** *auto*
then show *?thesis* **unfolding** *finite-jumpFs-def P-def* **by** *auto*
qed

lemma *finite-jumpFs-subE*:
assumes *finite-jumpFs f a b a*≤*a*′ *b*′≤*b*
shows *finite-jumpFs f a*′ *b*′
using *assms* **unfolding** *finite-jumpFs-def*
apply (*elim rev-finite-subset*)
by *auto*

lemma *finite-Psegments-Re-imp-jumpFs*:
assumes *finite-Psegments* (λ*t*. *Re* (*g t* − *z*) = *0*) *a b continuous-on* {*a*..*b*} *g*
shows *finite-jumpFs* (λ*t*. *Im* (*g t* − *z*)/*Re* (*g t* − *z*)) *a b*
using *assms*
proof (*induct rule*:*finite-Psegments.induct*)
case (*emptyI a b*)
then show *?case* **unfolding** *finite-jumpFs-def*
by (*auto intro*:*rev-finite-subset*[*of* {*a*}])
next
case (*insertI-1 s a b*)
define *f* **where** *f*=(λ*t*. *Im* (*g t* − *z*) / *Re* (*g t* − *z*))
have *finite-jumpFs f a s*
proof −

```
    have continuous-on {a..s} g using ‹continuous-on {a..b} g› ‹s ∈ {a..<b}›
      by (auto elim:continuous-on-subset)
    then show ?thesis using insertI-1 unfolding f-def by auto
  qed
  moreover have finite-jumpFs f s b
  proof –
    have jumpF f (at-left x) =0 jumpF f (at-right x) = 0 when x∈{s<..<b} for x
    proof –
      show jumpF f (at-left x) =0
        apply (rule jumpF-eventually-const[of - 0])
        unfolding eventually-at-left
         apply (rule exI[where x=s])
        using that insertI-1 unfolding f-def by auto
      show jumpF f (at-right x) = 0
        apply (rule jumpF-eventually-const[of - 0])
        unfolding eventually-at-right
         apply (rule exI[where x=b])
        using that insertI-1 unfolding f-def by auto
    qed
    then have {x. (jumpF f (at-left x) ≠ 0 ∨ jumpF f (at-right x) ≠ 0) ∧ s ≤ x
∧ x ≤ b}
          = {x. (jumpF f (at-left x) ≠ 0 ∨ jumpF f (at-right x) ≠ 0) ∧ (x=s ∨ x
= b)}
      using ‹s∈{a..<b}› by force
    then show ?thesis unfolding finite-jumpFs-def by auto
  qed
  ultimately show ?case using finite-jumpFs-combine[of - a s b] unfolding f-def
by auto
next
  case (insertI-2 s a b)
  define f where f=(λt. Im (g t − z) / Re (g t − z))
  have finite-jumpFs f a s
  proof –
    have continuous-on {a..s} g using ‹continuous-on {a..b} g› ‹s ∈ {a..<b}›
      by (auto elim:continuous-on-subset)
    then show ?thesis using insertI-2 unfolding f-def by auto
  qed
  moreover have finite-jumpFs f s b
  proof –
    have jumpF f (at-left x) =0 jumpF f (at-right x) = 0 when x∈{s<..<b} for x
    proof –
      have isCont f x
        unfolding f-def
        apply (intro continuous-intros isCont-Im isCont-Re
            continuous-on-interior[OF ‹continuous-on {a..b} g›])
        using insertI-2.hyps(1) that
          apply auto[2]
        using insertI-2.hyps(3) that by blast
      then show jumpF f (at-left x) =0 jumpF f (at-right x) = 0
```

by (*simp-all add*: *continuous-at-split jumpF-not-infinity*)
qed
then have {*x*. (*jumpF f* (*at-left x*) ≠ *0* ∨ *jumpF f* (*at-right x*) ≠ *0*) ∧ *s* ≤ *x* ∧ *x* ≤ *b*}
= {*x*. (*jumpF f* (*at-left x*) ≠ *0* ∨ *jumpF f* (*at-right x*) ≠ *0*) ∧ (*x*=*s* ∨ *x* = *b*)}
using ‹*s*∈{*a*..<*b*}› by *force*
then show *?thesis* unfolding *finite-jumpFs-def* by *auto*
qed
ultimately show *?case* using *finite-jumpFs-combine*[*of - a s b*] unfolding *f-def* by *auto*
qed

lemma *finite-ReZ-segments-imp-jumpFs*:
assumes *finite-ReZ-segments g z path g*
shows *finite-jumpFs* (λ*t*. *Im* (*g t* − *z*)/*Re* (*g t* − *z*)) *0 1*
using *assms* **unfolding** *finite-ReZ-segments-def path-def*
by (*rule finite-Psegments-Re-imp-jumpFs*)

5.7 *jumpF* at path ends

definition *jumpF-pathstart*::(*real* ⇒ *complex*) ⇒ *complex* ⇒ *real* **where**
jumpF-pathstart g z= *jumpF* (λ*t*. *Im*(*g t*− *z*)/*Re*(*g t* − *z*)) (*at-right 0*)

definition *jumpF-pathfinish*::(*real* ⇒ *complex*) ⇒ *complex* ⇒ *real* **where**
jumpF-pathfinish g z= *jumpF* (λ*t*. *Im*(*g t* − *z*)/*Re*(*g t* −*z*)) (*at-left 1*)

lemma *jumpF-pathstart-eq-0*:
assumes *path g Re*(*pathstart g*)≠*Re z*
shows *jumpF-pathstart g z* = *0*
unfolding *jumpF-pathstart-def*
apply (*rule jumpF-im-divide-Re-0*)
using *assms*[*unfolded pathstart-def*] **by** *auto*

lemma *jumpF-pathfinish-eq-0*:
assumes *path g Re*(*pathfinish g*)≠*Re z*
shows *jumpF-pathfinish g z* = *0*
unfolding *jumpF-pathfinish-def*
apply (*rule jumpF-im-divide-Re-0*)
using *assms*[*unfolded pathfinish-def*] **by** *auto*

lemma
shows *jumpF-pathfinish-reversepath*: *jumpF-pathfinish* (*reversepath g*) *z* = *jumpF-pathstart g z*
and *jumpF-pathstart-reversepath*: *jumpF-pathstart* (*reversepath g*) *z* = *jumpF-pathfinish g z*
proof −
define *f* **where** *f*=(λ*t*. *Im* (*g t* − *z*) / *Re* (*g t* − *z*))
define *f'* **where** *f'*=(λ*t*. *Im* (*reversepath g t* − *z*) / *Re* (*reversepath g t* − *z*))

have $f\ o\ (\lambda t.\ 1 - t) = f'$
unfolding *f-def f'-def comp-def reversepath-def* **by** *auto*
then show *jumpF-pathfinish (reversepath g) z = jumpF-pathstart g z*
jumpF-pathstart (reversepath g) z = jumpF-pathfinish g z
unfolding *jumpF-pathstart-def jumpF-pathfinish-def*
using *jumpF-linear-comp(2)[of −1 f 1 0,simplified] jumpF-linear-comp(1)[of −1 f 1 1,simplified]*
apply *(fold f-def f'-def)*
by *auto*
qed

lemma *jumpF-pathstart-joinpaths[simp]*:
jumpF-pathstart (g1+++g2) z = jumpF-pathstart g1 z
proof −
let *?h=(λt. Im (g1 t − z) / Re (g1 t − z))*
let *?f=λt. Im ((g1 +++ g2) t − z) / Re ((g1 +++ g2) t − z)*
have *jumpF-pathstart g1 z = jumpF ?h (at-right 0)*
unfolding *jumpF-pathstart-def* **by** *simp*
also have ... = *jumpF (?h o (λt. 2∗t)) (at-right 0)*
using *jumpF-linear-comp[of 2 ?h 0 0,simplified]* **by** *auto*
also have ... = *jumpF ?f (at-right 0)*
proof *(rule jumpF-cong)*
show $\forall_F$ *x in at-right 0. (?h ∘ (∗) 2) x =?f x*
unfolding *eventually-at-right*
apply *(intro exI[***where** *x=1/2])*
by *(auto simp add:joinpaths-def)*
qed *simp*
also have ... =*jumpF-pathstart (g1+++g2) z*
unfolding *jumpF-pathstart-def* **by** *simp*
finally show *?thesis* **by** *simp*
qed

lemma *jumpF-pathfinish-joinpaths[simp]*:
jumpF-pathfinish (g1+++g2) z = jumpF-pathfinish g2 z
proof −
let *?h=(λt. Im (g2 t − z) / Re (g2 t − z))*
let *?f=λt. Im ((g1 +++ g2) t − z) / Re ((g1 +++ g2) t − z)*
have *jumpF-pathfinish g2 z = jumpF ?h (at-left 1)*
unfolding *jumpF-pathfinish-def* **by** *simp*
also have ... = *jumpF (?h o (λt. 2∗t−1)) (at-left 1)*
using *jumpF-linear-comp[of 2 - −1 1,simplified]* **by** *auto*
also have ... = *jumpF ?f (at-left 1)*
proof *(rule jumpF-cong)*
show $\forall_F$ *x in at-left 1. (?h ∘ (λt. 2 ∗ t − 1)) x =?f x*
unfolding *eventually-at-left*
apply *(intro exI[***where** *x=1/2])*
by *(auto simp add:joinpaths-def)*
qed *simp*
also have ... =*jumpF-pathfinish (g1+++g2) z*

unfolding *jumpF-pathfinish-def* **by** *simp*
finally show *?thesis* **by** *simp*
qed

5.8 Cauchy index

definition *cindex*::*real* ⇒ *real* ⇒ (*real* ⇒ *real*) ⇒ *int* **where**
cindex a b f = (∑ *x*∈{*x*. *jump f x*≠*0* ∧ *a*<*x* ∧ *x*<*b*}. *jump f x*)

definition *cindexE*::*real* ⇒ *real* ⇒ (*real* ⇒ *real*) ⇒ *real* **where**
cindexE a b f = (∑ *x*∈{*x*. *jumpF f* (*at-right x*) ≠*0* ∧ *a*≤*x* ∧ *x*<*b*}. *jumpF f* (*at-right x*))
− (∑ *x*∈{*x*. *jumpF f* (*at-left x*) ≠*0* ∧ *a*<*x* ∧ *x*≤*b*}. *jumpF f* (*at-left x*))

definition *cindexE-ubd*::(*real* ⇒ *real*) ⇒ *real* **where**
cindexE-ubd f = (∑ *x*∈{*x*. *jumpF f* (*at-right x*) ≠*0* }. *jumpF f* (*at-right x*))
− (∑ *x*∈{*x*. *jumpF f* (*at-left x*) ≠*0*}. *jumpF f* (*at-left x*))

lemma *cindexE-empty*:
cindexE a a f = *0*
unfolding *cindexE-def* **by** (*simp add*: *sum.neutral*)

lemma *cindex-const*: *cindex a b* (λ-. *c*) = *0*
unfolding *cindex-def*
apply (*rule sum.neutral*)
by *auto*

lemma *cindex-eq-cindex-poly*: *cindex a b* (λ*x*. *poly q x*/*poly p x*) = *cindex-poly a b q p*
proof (*cases p*=*0*)
case *True*
then show *?thesis* **using** *cindex-const* **by** *auto*
next
case *False*
have *cindex-poly a b q p* =
(∑ *x* |*jump-poly q p x* ≠*0* ∧ *a* < *x* ∧ *x* < *b*. *jump-poly q p x*)
unfolding *cindex-poly-def*
apply (*rule sum.mono-neutral-cong-right*)
using *jump-poly-not-root* **by** (*auto simp add*: ‹*p*≠*0*› *poly-roots-finite*)
also have ... = *cindex a b* (λ*x*. *poly q x*/*poly p x*)
unfolding *cindex-def*
apply (*rule sum.cong*)
using *jump-jump-poly*[*of q*] **by** *auto*
finally show *?thesis* **by** *auto*
qed

lemma *cindex-combine*:

assumes *finite*:*finite* {*x*. *jump f x*≠*0* ∧ *a*<*x* ∧ *x*<*c*} **and** *a*<*b b*<*c*
shows *cindex a c f* = *cindex a b f* + *jump f b* + *cindex b c f*
proof −
define *ssum* **where** *ssum* = (λ*s*. *sum* (*jump f*) ({*x*. *jump f x*≠*0* ∧ *a*<*x* ∧ *x*<*c*} ∩ *s*))
have *ssum-union*:*ssum* (*A* ∪ *B*) = *ssum A* + *ssum B* **when** *A* ∩ *B* ={} **for** *A B*
proof −
define *C* **where** *C*={*x*. *jump f x* ≠ *0* ∧ *a*<*x* ∧ *x*<*c*}
have *finite C* **using** *finite* **unfolding** *C-def* **.**
then show *?thesis*
unfolding *ssum-def*
apply (*fold C-def*)
using *sum-Un*[*of C* ∩ *A C* ∩ *B*] *that*
by (*simp add*: *inf-assoc inf-sup-aci*(*3*) *inf-sup-distrib1 sum.union-disjoint*)
qed
have *cindex a c f* = *ssum* ({*a*<..<*b*} ∪ {*b*} ∪ {*b*<..<*c*})
unfolding *ssum-def cindex-def*
apply (*rule sum.cong*[*of - - jump f jump f*,*simplified*])
using ‹*a*<*b*› ‹*b*<*c*› **by** *fastforce*
moreover have *cindex a b f* = *ssum* {*a*<..<*b*}
unfolding *cindex-def ssum-def* **using** ‹*a*<*b*› ‹*b*<*c*›
by (*intro sum.cong*,*auto*)
moreover have *jump f b* = *ssum* {*b*}
unfolding *ssum-def* **using** ‹*a*<*b*› ‹*b*<*c*› **by** (*cases jump f b*=*0*,*auto*)
moreover have *cindex b c f* = *ssum* {*b*<..<*c*}
unfolding *cindex-def ssum-def* **using** ‹*a*<*b*› ‹*b*<*c*› **by** (*intro sum.cong*,*auto*)
ultimately show *?thesis*
apply (*subst* (*asm*) *ssum-union*,*simp*)
by (*subst* (*asm*) *ssum-union*,*auto*)
qed

lemma *cindexE-combine*:
assumes *finite*:*finite-jumpFs f a c* **and** *a*≤*b b*≤*c*
shows *cindexE a c f* = *cindexE a b f* + *cindexE b c f*
proof −
define *S* **where** *S*={*x*. (*jumpF f* (*at-left x*) ≠ *0* ∨ *jumpF f* (*at-right x*) ≠ *0*) ∧ *a* ≤ *x* ∧ *x* ≤ *c*}
define *A0* **where** *A0*={*x*. *jumpF f* (*at-right x*) ≠ *0* ∧ *a* ≤ *x* ∧ *x* < *c*}
define *A1* **where** *A1*={*x*. *jumpF f* (*at-right x*) ≠ *0* ∧ *a* ≤ *x* ∧ *x* < *b*}
define *A2* **where** *A2*={*x*. *jumpF f* (*at-right x*) ≠ *0* ∧ *b* ≤ *x* ∧ *x* < *c*}
define *B0* **where** *B0*={*x*. *jumpF f* (*at-left x*) ≠ *0* ∧ *a* < *x* ∧ *x* ≤ *c*}
define *B1* **where** *B1*={*x*. *jumpF f* (*at-left x*) ≠ *0* ∧ *a* < *x* ∧ *x* ≤ *b*}
define *B2* **where** *B2*={*x*. *jumpF f* (*at-left x*) ≠ *0* ∧ *b* < *x* ∧ *x* ≤ *c*}
have [*simp*]:*finite A1 finite A2 finite B1 finite B2*
proof −
have *finite S* **using** *finite* **unfolding** *finite-jumpFs-def S-def* **by** *auto*
moreover have *A1* ⊆ *S A2* ⊆ *S B1* ⊆ *S B2* ⊆ *S*
unfolding *A1-def A2-def B1-def B2-def S-def* **using** ‹*a*≤*b*› ‹*b*≤*c*› **by** *auto*

ultimately show *finite A1 finite A2 finite B1 finite B2* **by** (*auto elim:finite-subset*)
qed
have *cindexE a c f* = *sum* (λ*x. jumpF f* (*at-right x*)) *A0*
− *sum* (λ*x. jumpF f* (*at-left x*)) *B0*
unfolding *cindexE-def A0-def B0-def* **by** *auto*
also have ... = *sum* (λ*x. jumpF f* (*at-right x*)) (*A1* ∪ *A2*)
− *sum* (λ*x. jumpF f* (*at-left x*)) (*B1* ∪ *B2*)
proof −
have *A0=A1∪A2* **unfolding** *A0-def A1-def A2-def* **using** *assms* **by** *auto*
moreover have *B0=B1∪B2* **unfolding** *B0-def B1-def B2-def* **using** *assms*
by *auto*
ultimately show *?thesis* **by** *auto*
qed
also have ... = *cindexE a b f* + *cindexE b c f*
proof −
have *A1* ∩ *A2* = {} **unfolding** *A1-def A2-def* **by** *auto*
moreover have *B1* ∩ *B2* = {} **unfolding** *B1-def B2-def* **by** *auto*
ultimately show *?thesis*
unfolding *cindexE-def*
apply (*fold A1-def A2-def B1-def B2-def*)
by (*auto simp add:sum.union-disjoint*)
qed
finally show *?thesis* **.**
qed

lemma *cindex-linear-comp*:
assumes *c≠0*
shows *cindex lb ub* (*f o* (λ*x. c∗x+b*)) = (*if c>0*
then cindex (*c∗lb+b*) (*c∗ub+b*) *f*
else − *cindex* (*c∗ub+b*) (*c∗lb+b*) *f*)
proof (*cases c>0*)
case *False*
then have *c<0* **using** ‹*c≠0*› **by** *auto*
have *cindex lb ub* (*f o* (λ*x. c∗x+b*)) = − *cindex* (*c∗ub+b*) (*c∗lb+b*) *f*
unfolding *cindex-def*
apply (*subst sum-negf*[*symmetric*])
apply (*intro sum.reindex-cong*[*of* λ*x.* (*x−b*)/*c*])
subgoal by (*simp add*: *inj-on-def*)
subgoal using *False*
apply (*subst jump-linear-comp*[*OF* ‹*c≠0*›])
by (*auto simp add*:‹*c<0*› ‹*c≠0*› *field-simps*)
subgoal for *x*
apply (*subst jump-linear-comp*[*OF* ‹*c≠0*›])
by (*auto simp add*:‹*c<0*› ‹*c≠0*› *False field-simps*)
done
then show *?thesis* **using** *False* **by** *auto*
next
case *True*
have *cindex lb ub* (*f o* (λ*x. c∗x+b*)) = *cindex* (*c∗lb+b*) (*c∗ub+b*) *f*

```
    unfolding cindex-def
    apply (intro sum.reindex-cong[of λx. (x−b)/c])
    subgoal by (simp add: inj-on-def)
    subgoal
      apply (subst jump-linear-comp[OF ‹c≠0›])
      by (auto simp add: True ‹c≠0› field-simps)
    subgoal for x
      apply (subst jump-linear-comp[OF ‹c≠0›])
      by (auto simp add: ‹c≠0› True field-simps)
    done
  then show ?thesis using True by auto
qed

lemma cindexE-linear-comp:
  assumes c≠0
  shows cindexE lb ub (f o (λx. c∗x+b)) = (if c>0
    then cindexE (c∗lb+b) (c∗ub+b) f
    else − cindexE (c∗ub+b) (c∗lb+b) f)
proof −
  define cright where cright = (λlb ub f. (∑x | jumpF f (at-right x) ≠ 0 ∧ lb ≤
x ∧ x < ub.
                        jumpF f (at-right x)))
  define cleft where cleft = (λlb ub f. (∑x | jumpF f (at-left x) ≠ 0 ∧ lb < x ∧
x ≤ ub.
                        jumpF f (at-left x)))
  have cindexE-unfold:cindexE lb ub f = cright lb ub f − cleft lb ub f
    for lb ub f unfolding cindexE-def cright-def cleft-def by auto
  have ?thesis when c<0
  proof −
    have cright lb ub (f ∘ (λx. c ∗ x + b)) = cleft (c ∗ ub + b) (c ∗ lb + b) f
      unfolding cright-def cleft-def
      apply (intro sum.reindex-cong[of λx. (x−b)/c])
      subgoal by (simp add: inj-on-def)
      subgoal using that
        by (subst jumpF-linear-comp[OF ‹c≠0›],auto simp add:field-simps)
      subgoal for x using that
        by (subst jumpF-linear-comp[OF ‹c≠0›],auto simp add: field-simps)
      done
    moreover have cleft lb ub (f ∘ (λx. c ∗ x + b)) = cright (c∗ub+b) (c∗lb + b)
f
      unfolding cright-def cleft-def
      apply (intro sum.reindex-cong[of λx. (x−b)/c])
      subgoal by (simp add: inj-on-def)
      subgoal using that
        by (subst jumpF-linear-comp[OF ‹c≠0›],auto simp add:field-simps)
      subgoal for x using that
        by (subst jumpF-linear-comp[OF ‹c≠0›],auto simp add: field-simps)
      done
    ultimately show ?thesis unfolding cindexE-unfold using that by auto
```

```
  qed
  moreover have ?thesis when c>0
  proof -
    have cright lb ub (f ∘ (λx. c * x + b)) = cright (c * lb + b) (c * ub + b) f
      unfolding cright-def cleft-def
      apply (intro sum.reindex-cong[of λx. (x−b)/c])
      subgoal by (simp add: inj-on-def)
      subgoal using that
        by (subst jumpF-linear-comp[OF ‹c≠0›],auto simp add:field-simps)
      subgoal for x using that
        by (subst jumpF-linear-comp[OF ‹c≠0›],auto simp add: field-simps)
      done
    moreover have cleft lb ub (f ∘ (λx. c * x + b)) = cleft (c*lb+b) (c*ub + b) f
      unfolding cright-def cleft-def
      apply (intro sum.reindex-cong[of λx. (x−b)/c])
      subgoal by (simp add: inj-on-def)
      subgoal using that
        by (subst jumpF-linear-comp[OF ‹c≠0›],auto simp add:field-simps)
      subgoal for x using that
        by (subst jumpF-linear-comp[OF ‹c≠0›],auto simp add: field-simps)
      done
    ultimately show ?thesis unfolding cindexE-unfold using that by auto
  qed
  ultimately show ?thesis using ‹c≠0› by auto
qed

lemma cindexE-cong:
  assumes finite s and fg-eq:⋀x. ⟦a<x;x<b;x∉s⟧ ⟹ f x = g x
  shows cindexE a b f = cindexE a b g
proof -
  define left where
     left=(λf. (∑x | jumpF f (at-left x) ≠ 0 ∧ a < x ∧ x ≤ b. jumpF f (at-left
x)))
  define right where
    right=(λf. (∑x | jumpF f (at-right x) ≠ 0 ∧ a ≤ x ∧ x < b. jumpF f (at-right
x)))
  have left f = left g
  proof -
    have jumpF f (at-left x) = jumpF g (at-left x) when a<x x≤b for x
    proof (rule jumpF-cong)
      define cs where cs ≡ {y∈s. a<y ∧ y<x}
      define c where c≡ (if cs = {} then (x+a)/2 else Max cs)
      have finite cs unfolding cs-def using assms(1) by auto
      have c<x ∧ (∀y. c<y ∧ y<x ⟶ f y=g y)
      proof (cases cs={})
        case True
        then have ∀y. c<y ∧ y<x ⟶ y ∉ s unfolding cs-def c-def by force
        moreover have c=(x+a)/2 using True unfolding c-def by auto
        ultimately show ?thesis using fg-eq using that by auto
```

```
    next
      case False
      then have c∈cs unfolding c-def using False ‹finite cs› by auto
      moreover have ∀ y. c<y ∧ y<x ⟶ y ∉ s
      proof (rule ccontr)
        assume ¬ (∀ y. c < y ∧ y < x ⟶ y ∉ s)
        then obtain y' where c<y' y'<x y'∈s by auto
        then have y'∈cs using ‹c∈cs› unfolding cs-def by auto
        then have y'≤c unfolding c-def using False ‹finite cs› by auto
        then show False using ‹c<y'› by auto
      qed
     ultimately show ?thesis unfolding cs-def using that by (auto intro!:fg-eq)
    qed
    then show ∀F x in at-left x. f x = g x
      unfolding eventually-at-left by auto
  qed simp
  then show ?thesis
    unfolding left-def
    by (auto intro: sum.cong)
qed
moreover have right f = right g
proof −
  have jumpF f (at-right x) = jumpF g (at-right x) when a≤x x<b for x
  proof (rule jumpF-cong)
    define cs where cs ≡ {y∈s. x<y ∧ y<b}
    define c where c≡ (if cs = {} then (x+b)/2 else Min cs)
    have finite cs unfolding cs-def using assms(1) by auto
    have x<c ∧ (∀ y. x<y ∧ y<c ⟶ f y=g y)
    proof (cases cs={})
      case True
      then have ∀ y. x<y ∧ y<c ⟶ y ∉ s unfolding cs-def c-def by force
      moreover have c=(x+b)/2 using True unfolding c-def by auto
      ultimately show ?thesis using fg-eq using that by auto
    next
      case False
      then have c∈cs unfolding c-def using False ‹finite cs› by auto
      moreover have ∀ y. x<y ∧ y<c ⟶ y ∉ s
      proof (rule ccontr)
        assume ¬ (∀ y. x < y ∧ y < c ⟶ y ∉ s)
        then obtain y' where x<y' y'<c y'∈s by auto
        then have y'∈cs using ‹c∈cs› unfolding cs-def by auto
        then have y'≥c unfolding c-def using False ‹finite cs› by auto
        then show False using ‹c>y'› by auto
      qed
     ultimately show ?thesis unfolding cs-def using that by (auto intro!:fg-eq)
    qed
    then show ∀F x in at-right x. f x = g x
      unfolding eventually-at-right by auto
  qed simp
```

    then show *?thesis*
      unfolding *right-def*
      by (*auto intro*: *sum.cong*)
  qed
  ultimately show *?thesis* **unfolding** *cindexE-def left-def right-def* **by** *presburger*
qed

lemma *cindexE-constI*:
  assumes $\bigwedge t.\ [\![a<t;t<b]\!] \Longrightarrow f\ t=c$
  shows *cindexE a b f = 0*
proof −
  define *left* **where**
    $left=(\lambda f.\ (\sum x \mid jumpF\ f\ (at\text{-}left\ x) \neq 0 \wedge a < x \wedge x \leq b.\ jumpF\ f\ (at\text{-}left\ x)))$
  define *right* **where**
    $right=(\lambda f.\ (\sum x \mid jumpF\ f\ (at\text{-}right\ x) \neq 0 \wedge a \leq x \wedge x < b.\ jumpF\ f\ (at\text{-}right\ x)))$
  have *left f = 0*
  proof −
    have *jumpF f (at-left x) = 0* **when** $a<x$ $x\leq b$ **for** *x*
      apply (*rule jumpF-eventually-const*[*of - c*])
      unfolding *eventually-at-left* **using** *assms that* **by** *auto*
    then show *?thesis* **unfolding** *left-def* **by** *auto*
  qed
  moreover have *right f = 0*
  proof −
    have *jumpF f (at-right x) = 0* **when** $a\leq x$ $x<b$ **for** *x*
      apply (*rule jumpF-eventually-const*[*of - c*])
      unfolding *eventually-at-right* **using** *assms that* **by** *auto*
    then show *?thesis* **unfolding** *right-def* **by** *auto*
  qed
  ultimately show *?thesis* **unfolding** *cindexE-def left-def right-def* **by** *auto*
qed

lemma *cindex-eq-cindexE-divide*:
  fixes *f g*::*real* ⇒ *real*
  defines $h \equiv (\lambda x.\ f\ x/g\ x)$
  assumes $a<b$ **and**
    finite-fg: $finite\ \{x.\ (f\ x=0 \vee g\ x=0) \wedge a\leq x \wedge x\leq b\}$ **and**
    g-imp-f: $\forall x\in\{a..b\}.\ g\ x=0 \longrightarrow f\ x\neq 0$ **and**
    f-cont:*continuous-on* {*a..b*} *f* **and**
    g-cont:*continuous-on* {*a..b*} *g*
  shows *cindexE a b h = jumpF h (at-right a) + cindex a b h − jumpF h (at-left b)*
proof −
  define *R* **where** $R=(\lambda S.\{x.\ jumpF\ h\ (at\text{-}right\ x) \neq 0 \wedge x\in S\})$
  define *L* **where** $L=(\lambda S.\{x.\ jumpF\ h\ (at\text{-}left\ x) \neq 0 \wedge x\in S\})$
  define *right* **where** $right = (\lambda S.\ (\sum x\in R\ S.\ jumpF\ h\ (at\text{-}right\ x)))$
  define *left* **where** $left = (\lambda S.\ (\sum x\in L\ S.\ jumpF\ h\ (at\text{-}left\ x)))$

have *jump-gnz*:*jumpF h* (*at-left x*) = *0 jumpF h* (*at-right x*) = *0 jump h x*=*0*
when *a*<*x x*<*b g x*≠*0* **for** *x*
proof −
have *isCont h x* **unfolding** *h-def* **using** *f-cont g-cont that*
by (*auto intro*!:*continuous-intros elim*:*continuous-on-interior*)
then show *jumpF h* (*at-left x*) = *0 jumpF h* (*at-right x*) = *0 jump h x*=*0*
using *jumpF-not-infinity jump-not-infinity* **unfolding** *continuous-at-split*
by *auto*
qed

have *finite-jFs*:*finite-jumpFs h a b*
proof −
define *S* **where** *S*=(λ*s*. {*x*. (*jumpF h* (*at-left x*) ≠ *0* ∨ *jumpF h* (*at-right x*) ≠ *0*) ∧ *x*∈*s*})
note *jump-gnz*
then have *S* {*a*<..<*b*} ⊆ {*x*. (*f x*=*0*∨*g x*=*0*) ∧ *a*≤*x*∧*x*≤*b*}
unfolding *S-def* **by** *auto*
then have *finite* (*S* {*a*<..<*b*})
using *rev-finite-subset*[*OF finite-fg*] **by** *auto*
moreover have *finite* (*S* {*a*,*b*}) **unfolding** *S-def* **by** *auto*
moreover have *S* {*a*..*b*} = *S* {*a*<..<*b*} ∪ *S* {*a*,*b*}
unfolding *S-def* **using** ‹*a*<*b*› **by** *auto*
ultimately have *finite* (*S* {*a*..*b*}) **by** *auto*
then show *?thesis* **unfolding** *S-def finite-jumpFs-def* **by** *auto*
qed
have *cindexE a b h* = *right* {*a*..<*b*} − *left* {*a*<..*b*}
unfolding *cindexE-def right-def left-def R-def L-def* **by** *auto*
also have ... = *jumpF h* (*at-right a*) + *right* {*a*<..<*b*} − *left* {*a*<..<*b*} − *jumpF h* (*at-left b*)
proof −
have *right* {*a*..<*b*} = *jumpF h* (*at-right a*) + *right* {*a*<..<*b*}
proof (*cases jumpF h* (*at-right a*) =*0*)
case *True*
then have *R* {*a*..<*b*} = *R* {*a*<..<*b*}
unfolding *R-def* **using** *less-eq-real-def* **by** *auto*
then have *right* {*a*..<*b*} = *right* {*a*<..<*b*}
unfolding *right-def* **by** *auto*
then show *?thesis* **using** *True* **by** *auto*
next
case *False*
have *finite* (*R* {*a*..<*b*})
using *finite-jFs* **unfolding** *R-def finite-jumpFs-def*
by (*auto elim*:*rev-finite-subset*)
moreover have *a* ∈ *R* {*a*..<*b*} **using** *False* ‹*a*<*b*› **unfolding** *R-def* **by** *auto*
moreover have *R* {*a*..<*b*} − {*a*} = *R* {*a*<..<*b*} **unfolding** *R-def* **by** *auto*
ultimately show *right* {*a*..<*b*}= *jumpF h* (*at-right a*)
\+ *right* {*a*<..<*b*}
using *sum.remove*[*of R* {*a*..<*b*} *a* λ*x*. *jumpF h* (*at-right x*)]
unfolding *right-def* **by** *simp*

```
  qed
  moreover have left {a<..b} = jumpF h (at-left b) +  left {a<..<b}
  proof (cases jumpF h (at-left b) =0)
    case True
    then have L {a<..b} = L {a<..<b}
      unfolding L-def using less-eq-real-def by auto
    then have left {a<..b} = left {a<..<b}
      unfolding left-def by auto
    then show ?thesis using True by auto
  next
    case False
    have finite (L {a<..b})
      using finite-jFs unfolding L-def finite-jumpFs-def
      by (auto elim:rev-finite-subset)
    moreover have b ∈ L {a<..b} using False ‹a<b› unfolding L-def by auto
    moreover have L {a<..b} − {b} = L {a<..<b} unfolding L-def by auto
    ultimately show left {a<..b}= jumpF h (at-left b) + left {a<..<b}
      using sum.remove[of L {a<..b} b λx. jumpF h (at-left x)]
      unfolding left-def by simp
  qed
  ultimately show ?thesis by simp
qed
also have ... = jumpF h (at-right a) + cindex a b h − jumpF h (at-left b)
proof −
  define S where S={x. g x=0 ∧ a < x ∧ x < b}
  have right {a<..<b} = sum (λx. jumpF h (at-right x)) S
    unfolding right-def S-def R-def
    apply (rule sum.mono-neutral-left)
    subgoal using finite-fg by (auto elim:rev-finite-subset)
    subgoal using jump-gnz by auto
    subgoal by auto
    done
  moreover have left {a<..<b} = sum (λx. jumpF h (at-left x)) S
    unfolding left-def S-def L-def
    apply (rule sum.mono-neutral-left)
    subgoal using finite-fg by (auto elim:rev-finite-subset)
    subgoal using jump-gnz by auto
    subgoal by auto
    done
  ultimately have right {a<..<b} − left {a<..<b}
      = sum (λx. jumpF h (at-right x) − jumpF h (at-left x)) S
    by (simp add: sum-subtractf)
  also have ... = sum (λx. of-int(jump h x)) S
  proof (rule sum.cong)
    fix x assume x∈S
    define hr where hr = sgnx h (at-right x)
    define hl where hl = sgnx h (at-left x)
    have h sgnx-able (at-left x) hr≠0 h sgnx-able (at-right x) hl≠0
    proof −
```

have *finite* {*t*. *h t* = *0* ∧ *a* < *t* ∧ *t* < *b*}
using *finite-fg* **unfolding** *h-def* **by** (*auto elim*!:*rev-finite-subset*)
moreover have *continuous-on* ({*a*<..<*b*} − {*x*. *g x* = *0* ∧ *a* < *x* ∧ *x* < *b*}) *h*
unfolding *h-def* **using** *f-cont g-cont*
by (*auto intro*!: *continuous-intros elim*:*continuous-on-subset*)
moreover have *finite* {*x*. *g x* = *0* ∧ *a* < *x* ∧ *x* < *b*}
using *finite-fg* **by** (*auto elim*!:*rev-finite-subset*)
moreover have *x* ∈ {*a*<..<*b*}
using ‹*x*∈*S*› **unfolding** *S-def* **by** *auto*
ultimately show *h sgnx-able* (*at-left x*) *hl*≠*0 h sgnx-able* (*at-right x*) *hr*≠*0*

using *finite-sgnx-at-left-at-right*[*of h a b* {*x*. *g x*=*0* ∧ *a*<*x*∧*x*<*b*} *x*]
unfolding *hl-def hr-def* **by** *blast*+
qed
then have (*h has-sgnx hl*) (*at-left x*) (*h has-sgnx hr*) (*at-right x*)
unfolding *hl-def hr-def* **using** *sgnx-able-sgnx* **by** *blast*+
moreover have *isCont* (*inverse* ∘ *h*) *x*
proof −
have *f x*≠*0* **using** ‹*x*∈*S*› *g-imp-f* **unfolding** *S-def* **by** *auto*
then show *?thesis* **using** *f-cont g-cont* ‹*x*∈*S*› **unfolding** *h-def S-def*
by (*auto simp add*:*comp-def intro*!:*continuous-intros elim*:*continuous-on-interior*)
qed
ultimately show *jumpF h* (*at-right x*) − *jumpF h* (*at-left x*) = *real-of-int* (*jump h x*)
using *jump-jumpF*[*of x h*] ‹*hr*≠*0*› ‹*hl*≠*0*› **by** *auto*
qed *auto*
also have ... = *cindex a b h*
unfolding *cindex-def of-int-sum S-def*
apply (*rule sum.mono-neutral-cong-right*)
using *jump-gnz finite-fg* **by** (*auto elim*:*rev-finite-subset*)
finally show *?thesis* **by** *simp*
qed
finally show *?thesis* **.**
qed

5.9 Cauchy index along a path

definition *cindex-path*::(*real* ⇒ *complex*) ⇒ *complex* ⇒ *int* **where**
cindex-path g z = *cindex 0 1* (λ*t*. *Im* (*g t* − *z*) / *Re* (*g t* − *z*))

definition *cindex-pathE*::(*real* ⇒ *complex*) ⇒ *complex* ⇒ *real* **where**
cindex-pathE g z = *cindexE 0 1* (λ*t*. *Im* (*g t* − *z*) / *Re* (*g t* − *z*))

lemma *cindex-pathE-point*: *cindex-pathE* (*linepath a a*) *b* = *0*
unfolding *cindex-pathE-def* **by** (*simp add*:*cindexE-constI*)

lemma *cindex-path-reversepath*:
cindex-path (*reversepath g*) *z* = − *cindex-path g z*

proof −
  define *f* **where** *f*=(*λt. Im* (*g t* − *z*) / *Re* (*g t* − *z*))
  define *f′* **where** *f′*=(*λt. Im* (*reversepath g t* − *z*) / *Re* (*reversepath g t* − *z*))
  have *f o* (*λt. 1* − *t*) = *f′*
    unfolding *f-def f′-def comp-def reversepath-def* **by** *auto*
  then have *cindex 0 1 f′* = − *cindex 0 1 f*
    using *cindex-linear-comp*[*of* −*1 0 1 f 1*,*simplified*] **by** *simp*
  then show *?thesis*
    unfolding *cindex-path-def*
    apply (*fold f-def f′-def*)
    by *simp*
qed

lemma *cindex-pathE-reversepath*: *cindex-pathE* (*reversepath g*) *z* = −*cindex-pathE g z*
  using *cindexE-linear-comp*[*of* −*1 0 1 λt.* (*Im* (*g t*) − *Im z*) / (*Re* (*g t*) − *Re z*) *1*]
  by (*simp add*: *cindex-pathE-def reversepath-def o-def*)

lemma *cindex-pathE-reversepath′*: *cindex-pathE g z* = −*cindex-pathE* (*reversepath g*) *z*
  using *cindexE-linear-comp*[*of* −*1 0 1 λt.* (*Im* (*g t*) − *Im z*) / (*Re* (*g t*) − *Re z*) *1*]
  by (*simp add*: *cindex-pathE-def reversepath-def o-def*)

lemma *cindex-pathE-joinpaths*:
  assumes *g1*:*finite-ReZ-segments g1 z* **and** *g2*: *finite-ReZ-segments g2 z* **and**
    path g1 path g2 pathfinish g1 = *pathstart g2*
  shows *cindex-pathE* (*g1*+++*g2*) *z* = *cindex-pathE g1 z* + *cindex-pathE g2 z*
proof −
  define *f* **where** *f* = (*λg* (*t*::*real*). *Im* (*g t* − *z*) / *Re* (*g t* − *z*))
  have *cindex-pathE* (*g1* +++ *g2*) *z* = *cindexE 0 1* (*f* (*g1*+++*g2*))
    unfolding *cindex-pathE-def f-def* **by** *auto*
  also have ... = *cindexE 0* (*1*/*2*) (*f* (*g1*+++*g2*)) + *cindexE* (*1*/*2*) *1* (*f* (*g1*+++*g2*))
  proof (*rule cindexE-combine*)
    show *finite-jumpFs* (*f* (*g1* +++ *g2*)) *0 1*
      unfolding *f-def*
      apply (*rule finite-ReZ-segments-imp-jumpFs*)
      subgoal using *finite-ReZ-segments-joinpaths*[*OF g1 g2*] *assms*(*3*−*5*) **.**
      subgoal using *path-join-imp*[*OF* ‹*path g1*› ‹*path g2*› ‹*pathfinish g1*=*pathstart g2*›] **.**
      done
  qed *auto*
  also have ... = *cindex-pathE g1 z* + *cindex-pathE g2 z*
  proof −
    have *cindexE 0* (*1*/*2*) (*f* (*g1*+++*g2*)) = *cindex-pathE g1 z*
    proof −
      have *cindexE 0* (*1*/*2*) (*f* (*g1*+++*g2*)) = *cindexE 0* (*1*/*2*) (*f g1 o* ((∗) *2*))
        apply (*rule cindexE-cong*)

    unfolding *comp-def joinpaths-def f-def* **by** *auto*
   also have ... = *cindexE 0 1* (*f g1*)
    using *cindexE-linear-comp*[*of 2 0 1/2 - 0*,*simplified*] **by** *simp*
   also have ... = *cindex-pathE g1 z*
    unfolding *cindex-pathE-def f-def* **by** *auto*
   finally show *?thesis* **.**
  qed
  moreover have *cindexE* (*1/2*) *1* (*f* (*g1+++g2*)) = *cindex-pathE g2 z*
  proof −
   have *cindexE* (*1/2*) *1* (*f* (*g1+++g2*)) = *cindexE* (*1/2*) *1* (*f g2 o* (*λx. 2∗x* − *1*))
    apply (*rule cindexE-cong*)
    unfolding *comp-def joinpaths-def f-def* **by** *auto*
   also have ... = *cindexE 0 1* (*f g2*)
    using *cindexE-linear-comp*[*of 2 1/2 1 - −1*,*simplified*] **by** *simp*
   also have ... = *cindex-pathE g2 z*
    unfolding *cindex-pathE-def f-def* **by** *auto*
   finally show *?thesis* **.**
  qed
  ultimately show *?thesis* **by** *simp*
 qed
 finally show *?thesis* **.**
qed

lemma *cindex-pathE-constI*:
 assumes ⋀*t*. ⟦*0<t*;*t<1*⟧ ⟹ *g t=c*
 shows *cindex-pathE g z* = *0*
 unfolding *cindex-pathE-def*
 apply (*rule cindexE-constI*)
 using *assms* **by** *auto*

lemma *cindex-pathE-subpath-combine*:
 assumes *g*:*finite-ReZ-segments g z***and** *path g* **and**
  0≤a a≤b b≤c c≤1
 shows *cindex-pathE* (*subpath a b g*) *z* + *cindex-pathE* (*subpath b c g*) *z*
    = *cindex-pathE* (*subpath a c g*) *z*
proof −
 define *f* **where** *f* = (*λt. Im* (*g t* − *z*) / *Re* (*g t* − *z*))
 have *?thesis* **when** *a=b*
 proof −
  have *cindex-pathE* (*subpath a b g*) *z* = *0*
   apply (*rule cindex-pathE-constI*)
   using *that* **unfolding** *subpath-def* **by** *auto*
  then show *?thesis* **using** *that* **by** *auto*
 qed
 moreover have *?thesis* **when** *b=c*
 proof −
  have *cindex-pathE* (*subpath b c g*) *z* = *0*
   apply (*rule cindex-pathE-constI*)

using *that* **unfolding** *subpath-def* **by** *auto*
then show *?thesis* **using** *that* **by** *auto*
qed
moreover have *?thesis* **when** $a\neq b$ $b\neq c$
proof −
have [*simp*]:$a<b$ $b<c$ $a<c$
using *that* ‹$a\leq b$› ‹$b\leq c$› **by** *auto*
have *cindex-pathE* (*subpath a b g*) *z* = *cindexE a b f*
proof −
have *cindex-pathE* (*subpath a b g*) *z* = *cindexE 0 1* ($f \circ (\lambda x.\ (b - a) * x + a)$)
unfolding *cindex-pathE-def f-def comp-def subpath-def* **by** *auto*
also have ... = *cindexE a b f*
using *cindexE-linear-comp*[*of b−a 0 1 f a*,*simplified*] *that*(*1*) **by** *auto*
finally show *?thesis* **.**
qed
moreover have *cindex-pathE* (*subpath b c g*) *z* = *cindexE b c f*
proof −
have *cindex-pathE* (*subpath b c g*) *z* = *cindexE 0 1* ($f \circ (\lambda x.\ (c - b) * x + b)$)
unfolding *cindex-pathE-def f-def comp-def subpath-def* **by** *auto*
also have ... = *cindexE b c f*
using *cindexE-linear-comp*[*of c−b 0 1 f b*,*simplified*] *that*(*2*) **by** *auto*
finally show *?thesis* **.**
qed
moreover have *cindex-pathE* (*subpath a c g*) *z* = *cindexE a c f*
proof −
have *cindex-pathE* (*subpath a c g*) *z* = *cindexE 0 1* ($f \circ (\lambda x.\ (c - a) * x + a)$)
unfolding *cindex-pathE-def f-def comp-def subpath-def* **by** *auto*
also have ... = *cindexE a c f*
using *cindexE-linear-comp*[*of c−a 0 1 f a*,*simplified*] ‹$a<c$› **by** *auto*
finally show *?thesis* **.**
qed
moreover have *cindexE a b f* + *cindexE b c f* = *cindexE a c f*
proof −
have *finite-jumpFs f a c*
using *finite-ReZ-segments-imp-jumpFs*[*OF g* ‹*path g*›] ‹$0\leq a$› ‹$c\leq 1$› **unfolding** *f-def*
by (*elim finite-jumpFs-subE*,*auto*)
then show *?thesis* **using** *cindexE-linear-comp cindexE-combine*[*OF -* ‹$a\leq b$› ‹$b\leq c$›] **by** *auto*
qed
ultimately show *?thesis* **by** *auto*
qed
ultimately show *?thesis* **by** *blast*
qed

lemma *cindex-pathE-shiftpath*:

assumes *finite-ReZ-segments g z s*∈{*0..1*} *path g* **and** *loop*:*pathfinish g* = *pathstart g*
shows *cindex-pathE* (*shiftpath s g*) *z* = *cindex-pathE g z*
proof −
define *f* **where** *f*=(λ*g t. Im* (*g* (*t*::*real*) − *z*) / *Re* (*g t* − *z*))
have *cindex-pathE* (*shiftpath s g*) *z* = *cindexE 0 1* (*f* (*shiftpath s g*))
unfolding *cindex-pathE-def f-def* **by** *simp*
also have ... = *cindexE 0* (*1*−*s*) (*f* (*shiftpath s g*)) + *cindexE* (*1*−*s*) *1* (*f* (*shiftpath s g*))
proof (*rule cindexE-combine*)
have *finite-ReZ-segments* (*shiftpath s g*) *z*
using *finite-ReZ-segments-shiftpah*[*OF assms*] **.**
from *finite-ReZ-segments-imp-jumpFs*[*OF this*] *path-shiftpath*[*OF* ‹*path g*› *loop* ‹*s*∈{*0..1*}›]
show *finite-jumpFs* (*f* (*shiftpath s g*)) *0 1* **unfolding** *f-def* **by** *simp*
show *0* ≤ *1* − *s 1* − *s* ≤ *1* **using** ‹*s*∈{*0..1*}› **by** *auto*
qed
also have ... = *cindexE 0 s* (*f g*) + *cindexE s 1* (*f g*)
proof −
have *cindexE 0* (*1*−*s*) (*f* (*shiftpath s g*)) = *cindexE s 1* (*f g*)
proof −
have *cindexE 0* (*1*−*s*) (*f* (*shiftpath s g*)) = *cindexE 0* (*1*−*s*) ((*f g*) *o* (λ*t. t*+*s*))
apply (*rule cindexE-cong*)
unfolding *shiftpath-def f-def* **using** ‹*s*∈{*0..1*}› **by** (*auto simp add*:*algebra-simps*)
also have ...= *cindexE s 1* (*f g*)
using *cindexE-linear-comp*[*of 1 0 1*−*s f g s*,*simplified*] **.**
finally show *?thesis* **.**
qed
moreover have *cindexE* (*1* − *s*) *1* (*f* (*shiftpath s g*)) = *cindexE 0 s* (*f g*)
proof −
have *cindexE* (*1* − *s*) *1* (*f* (*shiftpath s g*)) = *cindexE* (*1*−*s*) *1* ((*f g*) *o* (λ*t. t*+*s*−*1*))
apply (*rule cindexE-cong*)
unfolding *shiftpath-def f-def* **using** ‹*s*∈{*0..1*}› **by** (*auto simp add*:*algebra-simps*)
also have ... = *cindexE 0 s* (*f g*)
using *cindexE-linear-comp*[*of 1 1*−*s 1 f g s*−*1*,*simplified*]
by (*simp add*:*algebra-simps*)
finally show *?thesis* **.**
qed
ultimately show *?thesis* **by** *auto*
qed
also have ... = *cindexE 0 1* (*f g*)
proof (*rule cindexE-combine*[*symmetric*])
show *finite-jumpFs* (*f g*) *0 1*
using *finite-ReZ-segments-imp-jumpFs*[*OF assms*(*1*,*3*)] **unfolding** *f-def* **by** *simp*
show *0* ≤ *s s*≤*1* **using** ‹*s*∈{*0..1*}› **by** *auto*
qed

also have ... = *cindex-pathE g z*
unfolding *cindex-pathE-def f-def* **by** *simp*
finally show *?thesis* **.**
qed

5.10 Cauchy's Index Theorem

theorem *winding-number-cindex-pathE-aux*:
fixes *g*::*real* ⇒ *complex*
assumes *finite-ReZ-segments g z* **and** *valid-path g z* ∉ *path-image g* **and**
Re-ends:*Re* (*g 1*) = *Re z Re* (*g 0*) = *Re z*
shows *2* ∗ *Re*(*winding-number g z*) = − *cindex-pathE g z*
using *assms*
proof (*induct rule*:*finite-ReZ-segments-induct*)
case (*sub0 g z*)
have *winding-number* (*subpath 0 0 g*) *z* = *0*
using ‹*z* ∉ *path-image* (*subpath 0 0 g*)› **unfolding** *subpath-refl*
by (*auto intro*!: *winding-number-trivial*)
moreover have *cindex-pathE* (*subpath 0 0 g*) *z* = *0*
unfolding *subpath-def* **by** (*auto intro*:*cindex-pathE-constI*)
ultimately show *?case* **by** *auto*
next
case (*subEq s g z*)
have *Re-winding-0*:*Re*(*winding-number h z*) = *0*
when *Re-const*:∀ *t*∈{*0*..*1*}. *Re* (*h t*) = *Re z* **and** *valid-path h z*∉*path-image h*
for *h*
proof −
have *Re* (*winding-number* (λ*t*. *h t* − *z*) *0*) = (*Im* (*Ln* (*pathfinish* (λ*t*. *h t* −
z)))
− *Im* (*Ln* (*pathstart* (λ*t*. *h t* − *z*)))) / (*2* ∗ *pi*)
apply (*rule Re-winding-number-half-right*[*of* - *0*,*simplified*])
using *Re-const* ‹*valid-path h*› ‹*z* ∉ *path-image h*›
apply *auto*
by (*metis* (*no-types*, *opaque-lifting*) *add.commute imageE le-add-same-cancel1*
order-refl
path-image-def plus-complex.simps(*1*))
moreover have *Im* (*Ln* (*h 1* − *z*)) = *Im* (*Ln* (*h 0* − *z*))
proof −
define *z0* **where** *z0* = *h 0* − *z*
define *z1* **where** *z1* = *h 1* − *z*
have [*simp*]: *z0*≠*0 z1*≠*0 Re z0*=*0 Re z1*=*0*
using ‹*z* ∉ *path-image h*› *that*(*1*) **unfolding** *z1-def z0-def path-image-def*
by *auto*
have *?thesis* **when** [*simp*]: *Im z0*>*0 Im z1*>*0*
apply (*fold z1-def z0-def*)
using *Im-Ln-eq-pi-half*[*of z1*] *Im-Ln-eq-pi-half*[*of z0*] **by** *auto*
moreover have *?thesis* **when** [*simp*]: *Im z0*<*0 Im z1*<*0*
apply (*fold z1-def z0-def*)
using *Im-Ln-eq-pi-half*[*of z1*] *Im-Ln-eq-pi-half*[*of z0*] **by** *auto*

moreover have *False* **when** *Im z0≥0 Im z1≤0*
proof −
define *f* **where** *f*=(λ*t*. *Im* (*h t* − *z*))
have ∃ *x*≥*0*. *x* ≤ *1* ∧ *f x* = *0*
apply (*rule IVT2′*[*of f 1 0 0*])
using *that valid-path-imp-path*[*OF* ‹*valid-path h*›]
unfolding *f-def z0-def z1-def path-def*
by (*auto intro*:*continuous-intros*)
then show *False* **using** *Re-const* ‹*z* ∉ *path-image h*› **unfolding** *f-def*
by (*metis atLeastAtMost-iff complex-surj image-eqI minus-complex.simps*(*2*)

path-defs(*4*) *right-minus-eq*)
qed
moreover have *False* **when** *Im z0≤0 Im z1≥0*
proof −
define *f* **where** *f*=(λ*t*. *Im* (*h t* − *z*))
have ∃ *x*≥*0*. *x* ≤ *1* ∧ *f x* = *0*
apply (*rule IVT′*)
using *that valid-path-imp-path*[*OF* ‹*valid-path h*›]
unfolding *f-def z0-def z1-def path-def*
by (*auto intro*:*continuous-intros*)
then show *False* **using** *Re-const* ‹*z* ∉ *path-image h*› **unfolding** *f-def*
by (*metis atLeastAtMost-iff complex-surj image-eqI minus-complex.simps*(*2*)

path-defs(*4*) *right-minus-eq*)
qed
ultimately show *?thesis* **by** *argo*
qed
ultimately have *Re* (*winding-number* (λ*t*. *h t* − *z*) *0*) = *0*
unfolding *pathfinish-def pathstart-def* **by** *auto*
then show *?thesis* **using** *winding-number-offset* **by** *auto*
qed
have *?case* **when** *s* = *0*
proof −
have ∗: ∀ *t*∈{*0*..*1*}. *Re* (*g t*) = *Re z*
using ‹∀ *t*∈{*s*<..<*1*}. *Re* (*g t*) = *Re z*› ‹*Re* (*g 1*) = *Re z*› ‹*Re* (*g 0*) = *Re z*›
‹*s*=*0*›
by *force*
have *Re*(*winding-number g z*) = *0*
by (*rule Re-winding-0*[*OF* ∗ ‹*valid-path g*› ‹*z* ∉ *path-image g*›])
moreover have *cindex-pathE g z* = *0*
unfolding *cindex-pathE-def*
apply (*rule cindexE-constI*)
using ∗ **by** *auto*
ultimately show *?thesis* **by** *auto*
qed
moreover have *?case* **when** *s*≠*0*
proof −
define *g1* **where** *g1* = *subpath 0 s g*

```
define g2 where g2 = subpath s 1 g
have path g s>0
  using valid-path-imp-path[OF ‹valid-path g›] that ‹s∈{0..<1}› by auto
 have 2 ∗ Re (winding-number g z) = 2∗Re (winding-number g1 z) + 2∗Re
(winding-number g2 z)
    apply (subst winding-number-subpath-combine[OF ‹path g› ‹z∉path-image
g›,of 0 s 1
        ,simplified,symmetric])
  using ‹s∈{0..<1}› unfolding g1-def g2-def by auto
also have ... = − cindex-pathE g1 z − cindex-pathE g2 z
proof −
  have 2∗Re (winding-number g1 z) = − cindex-pathE g1 z
    unfolding g1-def
    apply (rule subEq.hyps(5))
  subgoal using subEq.hyps(1) subEq.prems(1) valid-path-subpath by fastforce

     subgoal by (meson Path-Connected.path-image-subpath-subset atLeastAt-
Most-iff
      atLeastLessThan-iff less-eq-real-def subEq(7) subEq.hyps(1) subEq.prems(1)

        subsetCE valid-path-imp-path zero-le-one)
     subgoal by (metis Groups.add-ac(2) add-0-left diff-zero mult.right-neutral
subEq(2)
        subEq(9) subpath-def)
    subgoal by (simp add: subEq.prems(4) subpath-def)
    done
  moreover have 2∗Re (winding-number g2 z) = − cindex-pathE g2 z
  proof −
    have ∗: ∀ t∈{0..1}. Re (g2 t) = Re z
    proof
      fix t::real assume t∈{0..1}
      have Re (g2 t) = Re z when t=0 ∨ t=1
        using that unfolding g2-def
           by (metis ‹s ≠ 0› add.left-neutral diff-add-cancel mult.commute
mult.left-neutral
          mult-zero-left subEq.hyps(2) subEq.prems(3) subpath-def)
      moreover have Re (g2 t) = Re z when t∈{0<..<1}
      proof −
        define t' where t'=(1 − s) ∗ t + s
        then have t'∈{s<..<1}
          using that ‹s∈{0..<1}› unfolding t'-def
          apply auto
        by (sos ((((A<0 ∗ (A<1 ∗ A<2)) ∗ R<1) + ((A<=1 ∗ (A<0 ∗ R<1))
∗ (R<1 ∗ [1]^2)))))
        then have Re (g t') = Re z
          using ‹∀ t∈{s<..<1}. Re (g t) = Re z› by auto
        then show ?thesis
          unfolding g2-def subpath-def t'-def .
      qed
```

```
        ultimately show Re (g2 t) = Re z using ‹t∈{0..1}› by fastforce
      qed
      have Re(winding-number g2 z) = 0
        apply (rule Re-winding-0[OF ∗])
         subgoal using g2-def subEq.hyps(1) subEq.prems(1) valid-path-subpath
by fastforce
       subgoal by (metis (no-types, opaque-lifting) Path-Connected.path-image-subpath-subset

          atLeastAtMost-iff atLeastLessThan-iff g2-def less-eq-real-def subEq.hyps(1)

                    subEq.prems(1) subEq.prems(2) subsetCE valid-path-imp-path
zero-le-one)
        done
      moreover have cindex-pathE g2 z = 0
        unfolding cindex-pathE-def
        apply (rule cindexE-constI)
        using ∗ by auto
      ultimately show ?thesis by auto
    qed
    ultimately show ?thesis by auto
  qed
  also have ... = − cindex-pathE g z
  proof −
    have finite-ReZ-segments g z
      unfolding finite-ReZ-segments-def
      apply (rule finite-Psegments.insertI-1[of s])
      subgoal using ‹s ∈ {0..<1}› by auto
      subgoal using ‹s = 0 ∨ Re (g s) = Re z› by auto
      subgoal using ‹∀ t∈{s<..<1}. Re (g t) = Re z› by auto
      subgoal
      proof −
        have finite-Psegments (λt. Re (g (s ∗ t)) = Re z) 0 1
          using ‹finite-ReZ-segments (subpath 0 s g) z›
          unfolding subpath-def finite-ReZ-segments-def by auto
        from finite-Psegments-pos-linear[of - 1/s 0 0 s,simplified,OF this]
        show finite-Psegments (λt. Re (g t − z) = 0) 0 s
          using ‹s>0› unfolding comp-def by auto
      qed
      done
    then show ?thesis
      using cindex-pathE-subpath-combine[OF - ‹path g›,of z 0 s 1,folded g1-def
g2-def,simplified]
      ‹s∈{0..<1}› by auto
  qed
  finally show ?thesis .
 qed
 ultimately show ?case by auto
next
 case (subNEq s g z)
```

```
 have Re-winding:2∗Re(winding-number h z) = jumpF-pathfinish h z − jumpF-pathstart
h z
    when Re-neq:∀ t∈{0<..<1}. Re (h t) ≠ Re z and Re (h 0) = Re z Re (h 1)
= Re z
        and valid-path h z∉path-image h for h
  proof −
    have Re-winding-pos:
        2∗Re(winding-number h0 0) = jumpF-pathfinish h0 0 − jumpF-pathstart h0
0
      when Re-gt:∀ t∈{0<..<1}. Re (h0 t) > 0 and Re (h0 0) = 0 Re (h0 1) = 0
        and valid-path h0 0∉path-image h0 for h0
    proof −
      define f where f ≡ (λ(t::real). Im(h0 t) / Re (h0 t))
      define ln0 where ln0 = Im (Ln (h0 0)) / pi
      define ln1 where ln1 = Im (Ln (h0 1)) / pi
      have path h0 using ‹valid-path h0› valid-path-imp-path by auto
      have h0 0≠0 h0 1≠0
        using path-defs(4) that(5) by fastforce+
      have ln1 = jumpF-pathfinish h0 0
      proof −
        have sgnx-at-left:((λx. Re (h0 x)) has-sgnx 1) (at-left 1)
            unfolding has-sgnx-def eventually-at-left using ‹∀ p∈{0<..<1}. Re (h0
p) > 0›
          by (intro exI[where x=0],auto)
        have cont:continuous (at-left 1) (λt. Im (h0 t))
              continuous (at-left 1) (λt. Re (h0 t))
          using ‹path h0› unfolding path-def
          by (auto intro:continuous-on-at-left[of 0 1] continuous-intros)
        have ?thesis when Im (h0 1) > 0
        proof −
          have ln1 = 1/2
            using Im-Ln-eq-pi-half[OF ‹h0 1≠0›] that ‹Re (h0 1) = 0› unfolding
ln1-def by auto
          moreover have jumpF-pathfinish h0 0 = 1/2
          proof −
            have filterlim f at-top (at-left 1) unfolding f-def
              apply (subst filterlim-divide-at-bot-at-top-iff[of - Im (h0 1)])
            using ‹Re(h0 1) = 0› sgnx-at-left cont that unfolding continuous-within
by auto
            then show ?thesis unfolding jumpF-pathfinish-def jumpF-def f-def by
auto
          qed
          ultimately show ?thesis by auto
        qed
        moreover have ?thesis when Im (h0 1) < 0
        proof −
          have ln1 = − 1/2
            using Im-Ln-eq-pi-half[OF ‹h0 1≠0›] that ‹Re (h0 1) = 0› unfolding
ln1-def by auto
```

moreover have *jumpF-pathfinish h0 0* = − *1/2*
proof −
have ((λ*x*. *Re* (*h0 x*)) *has-sgnx* − *sgn* (*Im* (*h0 1*))) (*at-left 1*)
using *sgnx-at-left that* **by** *auto*
then have *filterlim f at-bot* (*at-left 1*)
unfolding *f-def* **using** *cont that*
apply (*subst filterlim-divide-at-bot-at-top-iff*[*of - Im* (*h0 1*)])
unfolding *continuous-within* **using** ‹*Re*(*h0 1*) = *0*› **by** *auto*
then show *?thesis* **unfolding** *jumpF-pathfinish-def jumpF-def f-def* **by**
auto
qed
ultimately show *?thesis* **by** *auto*
qed
moreover have *Im* (*h0 1*)≠*0* **using** ‹*h0 1*≠*0*› ‹*Re* (*h0 1*) = *0*›
using *complex.expand* **by** *auto*
ultimately show *?thesis* **by** *linarith*
qed
moreover have *ln0* = *jumpF-pathstart h0 0*
proof −
have *sgnx-at-right*:((λ*x*. *Re* (*h0 x*)) *has-sgnx 1*) (*at-right 0*)
unfolding *has-sgnx-def eventually-at-right* **using** ‹∀ *p*∈{*0*<..<*1*}. *Re* (*h0*
p) > *0*›
by (*intro exI*[**where** *x*=*1*],*auto*)
have *cont*:*continuous* (*at-right 0*) (λ*t*. *Im* (*h0 t*))
continuous (*at-right 0*) (λ*t*. *Re* (*h0 t*))
using ‹*path h0*› **unfolding** *path-def*
by (*auto intro*:*continuous-on-at-right*[*of 0 1*] *continuous-intros*)
have *?thesis* **when** *Im* (*h0 0*) > *0*
proof −
have *ln0* = *1/2*
using *Im-Ln-eq-pi-half*[*OF* ‹*h0 0*≠*0*›] *that* ‹*Re* (*h0 0*) = *0*› **unfolding**
ln0-def **by** *auto*
moreover have *jumpF-pathstart h0 0* = *1/2*
proof −
have *filterlim f at-top* (*at-right 0*) **unfolding** *f-def*
apply (*subst filterlim-divide-at-bot-at-top-iff*[*of - Im* (*h0 0*)])
using ‹*Re*(*h0 0*) = *0*› *sgnx-at-right cont that* **unfolding** *continuous-within*
by *auto*
then show *?thesis* **unfolding** *jumpF-pathstart-def jumpF-def f-def* **by**
auto
qed
ultimately show *?thesis* **by** *auto*
qed
moreover have *?thesis* **when** *Im* (*h0 0*) < *0*
proof −
have *ln0* = − *1/2*
using *Im-Ln-eq-pi-half*[*OF* ‹*h0 0*≠*0*›] *that* ‹*Re* (*h0 0*) = *0*› **unfolding**
ln0-def **by** *auto*
moreover have *jumpF-pathstart h0 0* = − *1/2*

```
        proof -
          have filterlim f at-bot (at-right 0) unfolding f-def
            apply (subst filterlim-divide-at-bot-at-top-iff[of - Im (h0 0)])
        using ‹Re(h0 0) = 0› sgnx-at-right cont that unfolding continuous-within
by auto
          then show ?thesis unfolding jumpF-pathstart-def jumpF-def f-def by
auto
        qed
        ultimately show ?thesis by auto
      qed
      moreover have Im (h0 0)≠0 using ‹h0 0≠0› ‹Re (h0 0) = 0›
        using complex.expand by auto
      ultimately show ?thesis by linarith
    qed
    moreover have 2*Re(winding-number h0 0) = ln1 − ln0
    proof −
      have ∀ p∈path-image h0. 0 ≤ Re p
      proof
        fix p assume p ∈ path-image h0
        then obtain t where t:t∈{0..1} p = h0 t unfolding path-image-def by
auto
        have 0 ≤ Re p when t=0 ∨ t=1
          using that t ‹Re (h0 0) = 0› ‹Re (h0 1) = 0› by auto
        moreover have 0 ≤ Re p when t∈{0<..<1}
          using that t Re-gt[rule-format, of t] by fastforce
        ultimately show 0 ≤ Re p using t(1) by fastforce
      qed
      from Re-winding-number-half-right[of - 0,simplified,OF this ‹valid-path h0›
‹0 ∉ path-image h0›]
      show ?thesis unfolding ln1-def ln0-def pathfinish-def pathstart-def
        by (auto simp add:field-simps)
    qed
    ultimately show ?thesis by auto
  qed

  have ?thesis when ∀ t∈{0<..<1}. Re (h t) < Re z
  proof −
    let ?hu= λt. z − h t
    have 2*Re(winding-number ?hu 0) = jumpF-pathfinish ?hu 0 − jumpF-pathstart
?hu 0
      apply(rule Re-winding-pos)
      subgoal using that by auto
      subgoal using ‹Re (h 0) = Re z› by auto
      subgoal using ‹Re (h 1) = Re z› by auto
      subgoal using ‹valid-path h› valid-path-offset valid-path-uminus-comp
          unfolding comp-def by fastforce
      subgoal using ‹z∉path-image h› by (simp add: image-iff path-defs(4))
      done
    moreover have winding-number ?hu 0 = winding-number h z
```

using *winding-number-offset*[*of h z*]
winding-number-uminus-comp[*of* λ*t. h t*− *z 0* ,*unfolded comp-def* ,*simplified*]

‹*valid-path h*› ‹*z*∉*path-image h*› **by** *auto*
moreover have *jumpF-pathfinish ?hu 0* = *jumpF-pathfinish h z*
unfolding *jumpF-pathfinish-def*
apply (*auto intro*!:*jumpF-cong eventuallyI*)
by (*auto simp add*:*divide-simps algebra-simps*)
moreover have *jumpF-pathstart ?hu 0* = *jumpF-pathstart h z*
unfolding *jumpF-pathstart-def*
apply (*auto intro*!:*jumpF-cong eventuallyI*)
by (*auto simp add*:*divide-simps algebra-simps*)
ultimately show *?thesis* **by** *auto*
qed
moreover have *?thesis* **when** ∀ *t*∈{*0*<..<*1*}. *Re* (*h t*) > *Re z*
proof −
let *?hu*= λ*t. h t* − *z*
have *2*∗*Re*(*winding-number ?hu 0*) = *jumpF-pathfinish ?hu 0* − *jumpF-pathstart ?hu 0*
apply(*rule Re-winding-pos*)
subgoal using *that* **by** *auto*
subgoal using ‹*Re* (*h 0*) = *Re z*› **by** *auto*
subgoal using ‹*Re* (*h 1*) = *Re z*› **by** *auto*
subgoal using ‹*valid-path h*› *valid-path-offset valid-path-uminus-comp*
unfolding *comp-def* **by** *fastforce*
subgoal using ‹*z*∉*path-image h*› **by** *simp*
done
moreover have *winding-number ?hu 0* = *winding-number h z*
using *winding-number-offset*[*of h z*] ‹*valid-path h*› ‹*z*∉*path-image h*› **by** *auto*
moreover have *jumpF-pathfinish ?hu 0* = *jumpF-pathfinish h z*
unfolding *jumpF-pathfinish-def* **by** *auto*
moreover have *jumpF-pathstart ?hu 0* = *jumpF-pathstart h z*
unfolding *jumpF-pathstart-def* **by** *auto*
ultimately show *?thesis* **by** *auto*
qed
moreover have (∀ *t*∈{*0*<..<*1*}. *Re* (*h t*) > *Re z*) ∨ (∀ *t*∈{*0*<..<*1*}. *Re* (*h t*) < *Re z*)
proof (*rule ccontr*)
assume ¬ ((∀ *t*∈{*0*<..<*1*}. *Re z* < *Re* (*h t*)) ∨ (∀ *t*∈{*0*<..<*1*}. *Re* (*h t*) < *Re z*))
then obtain *t1 t2* **where** *t*:*t1*∈{*0*<..<*1*} *t2*∈{*0*<..<*1*} *Re* (*h t1*)≤*Re z Re* (*h t2*)≥*Re z*
unfolding *path-image-def* **by** *auto*
have *False* **when** *t1*≤*t2*
proof −
have *continuous-on* {*t1*..*t2*} (λ*t. Re* (*h t*))
using *valid-path-imp-path*[*OF* ‹*valid-path h*›] *t* **unfolding** *path-def*
by (*metis* (*full-types*) *atLeastatMost-subset-iff continuous-on-Re continuous-on-subset*

eucl-less-le-not-le greaterThanLessThan-iff)
then obtain *t′* **where** *t′*:*t′*≥*t1 t′*≤*t2 Re* (*h t′*) = *Re z*
using *IVT′*[*of* λ*t*. *Re* (*h t*) *t1* - *t2*] *t* ‹*t1*≤*t2*› **by** *auto*
then have *t′*∈{*0*<..<*1*} **using** *t* **by** *auto*
then have *Re* (*h t′*) ≠ *Re z* **using** *Re-neq* **by** *auto*
then show *False* **using** ‹*Re* (*h t′*) = *Re z*› **by** *simp*
qed
moreover have *False* **when** *t1*≥*t2*
proof −
have *continuous-on* {*t2*..*t1*} (λ*t*. *Re* (*h t*))
using *valid-path-imp-path*[*OF* ‹*valid-path h*›] *t* **unfolding** *path-def*
by (*metis* (*full-types*) *atLeastatMost-subset-iff continuous-on-Re continuous-on-subset*
eucl-less-le-not-le greaterThanLessThan-iff)
then obtain *t′* **where** *t′*:*t′*≤*t1 t′*≥*t2 Re* (*h t′*) = *Re z*
using *IVT2′*[*of* λ*t*. *Re* (*h t*) *t1* - *t2*] *t* ‹*t1*≥*t2*› **by** *auto*
then have *t′*∈{*0*<..<*1*} **using** *t* **by** *auto*
then have *Re* (*h t′*) ≠ *Re z* **using** *Re-neq* **by** *auto*
then show *False* **using** ‹*Re* (*h t′*) = *Re z*› **by** *simp*
qed
ultimately show *False* **by** *linarith*
qed
ultimately show *?thesis* **by** *blast*
qed

have *index-ends*:*cindex-pathE h z* = *jumpF-pathstart h z* − *jumpF-pathfinish h z*
when *Re-neq*:∀ *t*∈{*0*<..<*1*}. *Re* (*h t*) ≠ *Re z* **and** *valid-path h* **for** *h*
proof −
define *f* **where** *f* = (λ*t*. *Im* (*h t* − *z*) / *Re* (*h t* − *z*))
define *Ri* **where** *Ri* = {*x*. *jumpF f* (*at-right x*) ≠ *0* ∧ *0* ≤ *x* ∧ *x* < *1*}
define *Le* **where** *Le* = {*x*. *jumpF f* (*at-left x*) ≠ *0* ∧ *0* < *x* ∧ *x* ≤ *1*}
have *path h* **using** ‹*valid-path h*› *valid-path-imp-path* **by** *auto*
have *jumpF-eq0*: *jumpF f* (*at-left x*) = *0 jumpF f* (*at-right x*) = *0* **when** *x*∈{*0*<..<*1*} **for** *x*
proof −
have *Re* (*h x*) ≠ *Re z*
using ‹∀ *t*∈{*0*<..<*1*}. *Re* (*h t*) ≠ *Re z*› *that* **by** *blast*
then have *isCont f x*
unfolding *f-def* **using** *continuous-on-interior*[*OF* ‹*path h*›[*unfolded path-def*]] *that*
by (*auto intro*!: *continuous-intros isCont-Im isCont-Re*)
then show *jumpF f* (*at-left x*) = *0 jumpF f* (*at-right x*) = *0*
unfolding *continuous-at-split* **by** (*auto intro*: *jumpF-not-infinity*)
qed
have *cindex-pathE h z* = *cindexE 0 1 f*
unfolding *cindex-pathE-def f-def* **by** *simp*
also have ... = *sum* (λ*x*. *jumpF f* (*at-right x*)) *Ri* − *sum* (λ*x*. *jumpF f* (*at-left x*)) *Le*
unfolding *cindexE-def Ri-def Le-def* **by** *auto*

```
    also have ... = jumpF f (at-right 0) − jumpF f (at-left 1)
    proof −
      have sum (λx. jumpF f (at-right x)) Ri = jumpF f (at-right 0)
      proof (cases jumpF f (at-right 0) = 0)
        case True
        hence False if x ∈ Ri for x using that
          by (cases x = 0) (auto simp: jumpF-eq0 Ri-def)
        hence Ri = {} by blast
        then show ?thesis using True by auto
      next
        case False
        hence x ∈ Ri ⟷ x = 0 for x using that
          by (cases x = 0) (auto simp: jumpF-eq0 Ri-def)
        hence Ri = {0} by blast
        then show ?thesis by auto
      qed
      moreover have sum (λx. jumpF f (at-left x)) Le = jumpF f (at-left 1)
      proof (cases jumpF f (at-left 1) = 0)
        case True
        then have Le = {}
            unfolding Le-def using jumpF-eq0(1) greaterThanLessThan-iff by
fastforce
        then show ?thesis using True by auto
      next
        case False
        then have Le = {1}
            unfolding Le-def using jumpF-eq0(1) greaterThanLessThan-iff by
fastforce
        then show ?thesis by auto
      qed
      ultimately show ?thesis by auto
    qed
    also have ... = jumpF-pathstart h z − jumpF-pathfinish h z
      unfolding jumpF-pathstart-def jumpF-pathfinish-def f-def by simp
    finally show ?thesis .
  qed

  have ?case when s=0
  proof −
    have 2 ∗ Re (winding-number g z) = jumpF-pathfinish g z − jumpF-pathstart
g z
      apply (rule Re-winding)
      using subNEq that by auto
    moreover have cindex-pathE g z = jumpF-pathstart g z − jumpF-pathfinish g
z
      apply (rule index-ends)
      using subNEq that by auto
    ultimately show ?thesis by auto
  qed
```

moreover have *?case* **when** *s≠0*
proof −
define *g1* **where** *g1 = subpath 0 s g*
define *g2* **where** *g2 = subpath s 1 g*
have *path g s>0*
using *valid-path-imp-path[OF ‹valid-path g›] that ‹s∈{0..<1}›* **by** *auto*
have *2 ∗ Re (winding-number g z) = 2∗Re (winding-number g1 z) + 2∗Re (winding-number g2 z)*
apply *(subst winding-number-subpath-combine[OF ‹path g› ‹z∉path-image g›,of 0 s 1*
,simplified,symmetric])
using *‹s∈{0..<1}›* **unfolding** *g1-def g2-def* **by** *auto*
also have ... = − *cindex-pathE g1 z* − *cindex-pathE g2 z*
proof −
have *2∗Re (winding-number g1 z) = − cindex-pathE g1 z*
unfolding *g1-def*
apply *(rule subNEq.hyps(5))*
subgoal using *subNEq.hyps(1) subNEq.prems(1) valid-path-subpath* **by** *fastforce*
subgoal by *(meson Path-Connected.path-image-subpath-subset atLeastAt-Most-iff*
atLeastLessThan-iff less-eq-real-def subNEq(7) subNEq.hyps(1) sub-NEq.prems(1)
subsetCE valid-path-imp-path zero-le-one)
subgoal by *(metis Groups.add-ac(2) add-0-left diff-zero mult.right-neutral subNEq(2)*
subNEq(9) subpath-def)
subgoal by *(simp add: subNEq.prems(4) subpath-def)*
done
moreover have *2∗Re (winding-number g2 z) = − cindex-pathE g2 z*
proof −
have ∗:∀ *t∈{0<..<1}. Re (g2 t) ≠ Re z*
proof
fix *t::real* **assume** *t ∈ {0<..<1}*
define *t′* **where** *t′=(1 − s) ∗ t + s*
have *t′∈{s<..<1}* **unfolding** *t′-def* **using** *‹s∈{0..<1}› ‹t ∈ {0<..<1}›*
apply *(auto simp add:algebra-simps)*
by *(sos ((((A<0 ∗ (A<1 ∗ A<2)) ∗ R<1) + ((A<=1 ∗ (A<1 ∗ R<1)) ∗ (R<1 ∗ [1]^2)))))*
then have *Re (g t′) ≠ Re z* **using** *‹∀ t∈{s<..<1}. Re (g t) ≠ Re z›* **by** *auto*
then show *Re (g2 t) ≠ Re z* **unfolding** *g2-def subpath-def t′-def* **by** *auto*
qed
have *2∗Re (winding-number g2 z) = jumpF-pathfinish g2 z − jumpF-pathstart g2 z*
apply *(rule Re-winding[OF ∗])*
subgoal by *(metis add.commute add.right-neutral g2-def mult-zero-right subNEq.hyps(2)*
subpath-def that)

```
        subgoal by (simp add: ‹g2 ≡ subpath s 1 g› subNEq.prems(3) subpath-def)
        subgoal using g2-def subNEq.hyps(1) subNEq.prems(1) valid-path-subpath
by fastforce
        subgoal by (metis (no-types, opaque-lifting) Path-Connected.path-image-subpath-subset

                ‹path g› atLeastAtMost-iff atLeastLessThan-iff g2-def less-eq-real-def
subNEq.hyps(1)
                subNEq.prems(2) subsetCE zero-le-one)
          done
      moreover have cindex-pathE g2 z = jumpF-pathstart g2 z − jumpF-pathfinish
g2 z
          apply (rule index-ends[OF ∗])
            using g2-def subNEq.hyps(1) subNEq.prems(1) valid-path-subpath by
fastforce
        ultimately show ?thesis by auto
      qed
      ultimately show ?thesis by auto
    qed
    also have ... = − cindex-pathE g z
    proof −
      have finite-ReZ-segments g z
        unfolding finite-ReZ-segments-def
        apply (rule finite-Psegments.insertI-2[of s])
        subgoal using ‹s ∈ {0..<1}› by auto
        subgoal using ‹s = 0 ∨ Re (g s) = Re z› by auto
        subgoal using ‹∀ t∈{s<..<1}. Re (g t) ≠ Re z› by auto
        subgoal
        proof −
          have finite-Psegments (λt. Re (g (s ∗ t)) = Re z) 0 1
            using ‹finite-ReZ-segments (subpath 0 s g) z›
            unfolding subpath-def finite-ReZ-segments-def by auto
          from finite-Psegments-pos-linear[of - 1/s 0 0 s,simplified,OF this]
          show finite-Psegments (λt. Re (g t − z) = 0) 0 s
            using ‹s>0› unfolding comp-def by auto
        qed
        done
      then show ?thesis
        using cindex-pathE-subpath-combine[OF - ‹path g›,of z 0 s 1,folded g1-def
g2-def,simplified]
          ‹s∈{0..<1}› by auto
    qed
    finally show ?thesis .
  qed
  ultimately show ?case by auto
qed

theorem winding-number-cindex-pathE:
  fixes g::real ⇒ complex
  assumes finite-ReZ-segments g z and valid-path g z ∉ path-image g and
```

```
    loop: pathfinish g = pathstart g
  shows winding-number g z = − cindex-pathE g z / 2
proof (rule finite-ReZ-segment-cases[OF assms(1)])
  fix s assume s ∈ {0..<1} s = 0 ∨ Re (g s) = Re z
        and const:∀ t∈{s<..<1}. Re (g t) = Re z
        and finite:finite-ReZ-segments (subpath 0 s g) z
  have Re (g 1) = Re z
    apply(rule continuous-constant-on-closure[of {s<..<1} λt. Re(g t)])
    subgoal using valid-path-imp-path[OF ‹valid-path g›,unfolded path-def] ‹s∈{0..<1}›
      by (auto intro!:continuous-intros continuous-Re elim:continuous-on-subset)
    subgoal using const by auto
    subgoal using ‹s∈{0..<1}› by auto
    done
  moreover then have Re (g 0) = Re z using loop unfolding path-defs by auto
  ultimately have 2 ∗ Re (winding-number g z) = − cindex-pathE g z
    using winding-number-cindex-pathE-aux[of g z] assms(1−3) by auto
  moreover have winding-number g z ∈ ℤ
   using integer-winding-number[OF - loop ‹z∉path-image g›] valid-path-imp-path[OF
‹valid-path g›]
    by auto
  ultimately show winding-number g z = − cindex-pathE g z / 2
    by (metis add.right-neutral complex-eq complex-is-Int-iff mult-zero-right
        nonzero-mult-div-cancel-left of-real-0 zero-neq-numeral)
next
  fix s assume s ∈ {0..<1} s = 0 ∨ Re (g s) = Re z
        and Re-neq:∀ t∈{s<..<1}. Re (g t) ≠ Re z
        and finite:finite-ReZ-segments (subpath 0 s g) z
  have path g using ‹valid-path g› valid-path-imp-path by auto
  let ?goal = 2 ∗ Re (winding-number g z) = − cindex-pathE g z
  have ?goal when s=0
  proof −
    have index-ends:cindex-pathE h z = jumpF-pathstart h z − jumpF-pathfinish h
z
      when Re-neq:∀ t∈{0<..<1}. Re (h t) ≠ Re z and valid-path h for h
    proof −
      define f where f = (λt. Im (h t − z) / Re (h t − z))
      define Ri where Ri = {x. jumpF f (at-right x) ≠ 0 ∧ 0 ≤ x ∧ x < 1}
      define Le where Le = {x. jumpF f (at-left x) ≠ 0 ∧ 0 < x ∧ x ≤ 1}
      have path h using ‹valid-path h› valid-path-imp-path by auto
       have jumpF-eq0: jumpF f (at-left x) = 0 jumpF f (at-right x) = 0 when
x∈{0<..<1} for x
      proof −
        have Re (h x) ≠ Re z
          using ‹∀ t∈{0<..<1}. Re (h t) ≠ Re z› that by blast
        then have isCont f x
              unfolding f-def using continuous-on-interior[OF ‹path h›[unfolded
path-def]] that
          by (auto intro!: continuous-intros isCont-Im isCont-Re)
        then show jumpF f (at-left x) = 0 jumpF f (at-right x) = 0
```

unfolding *continuous-at-split* **by** (*auto intro*: *jumpF-not-infinity*)
qed
have *cindex-pathE h z* = *cindexE 0 1 f*
unfolding *cindex-pathE-def f-def* **by** *simp*
also have ... = *sum* (λx. *jumpF f* (*at-right x*)) *Ri* − *sum* (λx. *jumpF f* (*at-left x*)) *Le*
unfolding *cindexE-def Ri-def Le-def* **by** *auto*
also have ... = *jumpF f* (*at-right 0*) − *jumpF f* (*at-left 1*)
proof −
have *sum* (λx. *jumpF f* (*at-right x*)) *Ri* = *jumpF f* (*at-right 0*)
proof (*cases jumpF f* (*at-right 0*) = *0*)
case *True*
hence *False* **if** $x \in Ri$ **for** *x* **using** *that*
by (*cases x* = *0*) (*auto simp*: *jumpF-eq0 Ri-def*)
hence *Ri* = {} **by** *blast*
then show *?thesis* **using** *True* **by** *auto*
next
case *False*
hence $x \in Ri \longleftrightarrow x = 0$ **for** *x* **using** *that*
by (*cases x* = *0*) (*auto simp*: *jumpF-eq0 Ri-def*)
then have *Ri* = {*0*} **by** *blast*
then show *?thesis* **by** *auto*
qed
moreover have *sum* (λx. *jumpF f* (*at-left x*)) *Le* = *jumpF f* (*at-left 1*)
proof (*cases jumpF f* (*at-left 1*) = *0*)
case *True*
then have *Le* = {}
unfolding *Le-def* **using** *jumpF-eq0*(*1*) *greaterThanLessThan-iff* **by** *fastforce*
then show *?thesis* **using** *True* **by** *auto*
next
case *False*
then have *Le* = {*1*}
unfolding *Le-def* **using** *jumpF-eq0*(*1*) *greaterThanLessThan-iff* **by** *fastforce*
then show *?thesis* **by** *auto*
qed
ultimately show *?thesis* **by** *auto*
qed
also have ... = *jumpF-pathstart h z* − *jumpF-pathfinish h z*
unfolding *jumpF-pathstart-def jumpF-pathfinish-def f-def* **by** *simp*
finally show *?thesis* **.**
qed
define *fI* **where** *fI*=(λt. *Im* (*g t* − *z*))
define *fR* **where** *fR*=(λt. *Re* (*g t* − *z*))
have *fI*: (*fI* ⟶ *fI 0*) (*at-right 0*) (*fI* ⟶ *fI 1*) (*at-left 1*)
proof −
have *continuous* (*at-right 0*) *fI*
apply (*rule continuous-on-at-right*[*of* - *1*])

using ‹*path g*› **unfolding** *fI-def path-def* **by** (*auto intro*:*continuous-intros*)
then show ($fI \longrightarrow fI\ 0$) (*at-right 0*) **by** (*simp add*: *continuous-within*)
next
have *continuous* (*at-left 1*) *fI*
apply (*rule continuous-on-at-left*[*of 0*])
using ‹*path g*› **unfolding** *fI-def path-def* **by** (*auto intro*:*continuous-intros*)
then show ($fI \longrightarrow fI\ 1$) (*at-left 1*) **by** (*simp add*: *continuous-within*)
qed
have *fR*: ($fR \longrightarrow 0$) (*at-right 0*) ($fR \longrightarrow 0$) (*at-left 1*) **when** *Re* (*g 0*) = *Re z*
proof −
have *continuous* (*at-right 0*) *fR*
apply (*rule continuous-on-at-right*[*of - 1*])
using ‹*path g*› **unfolding** *fR-def path-def* **by** (*auto intro*:*continuous-intros*)

then show ($fR \longrightarrow 0$) (*at-right 0*) **using** *that* **unfolding** *fR-def* **by** (*simp add*: *continuous-within*)
next
have *continuous* (*at-left 1*) *fR*
apply (*rule continuous-on-at-left*[*of 0*])
using ‹*path g*› **unfolding** *fR-def path-def* **by** (*auto intro*:*continuous-intros*)

then show ($fR \longrightarrow 0$) (*at-left 1*)
using *that loop* **unfolding** *fR-def path-defs* **by** (*simp add*: *continuous-within*)
qed
have (∀ *t*∈{*0*<..<*1*}. *Re* (*g t*) > *Re z*) ∨ (∀ *t*∈{*0*<..<*1*}. *Re* (*g t*) < *Re z*)
proof (*rule ccontr*)
assume ¬ ((∀ *t*∈{*0*<..<*1*}. *Re z* < *Re* (*g t*)) ∨ (∀ *t*∈{*0*<..<*1*}. *Re* (*g t*) < *Re z*))
then obtain *t1 t2* **where** *t*:*t1*∈{*0*<..<*1*} *t2*∈{*0*<..<*1*} *Re* (*g t1*)≤*Re z Re* (*g t2*)≥*Re z*
unfolding *path-image-def* **by** *auto*
have *False* **when** *t1*≤*t2*
proof −
have *continuous-on* {*t1*..*t2*} (λ*t*. *Re* (*g t*))
using *valid-path-imp-path*[*OF* ‹*valid-path g*›] *t* **unfolding** *path-def*
by (*metis* (*full-types*) *atLeastatMost-subset-iff continuous-on-Re continuous-on-subset*
eucl-less-le-not-le greaterThanLessThan-iff)
then obtain *t′* **where** *t′*:*t′*≥*t1 t′*≤*t2 Re* (*g t′*) = *Re z*
using *IVT′*[*of* λ*t*. *Re* (*g t*) *t1* - *t2*] *t* ‹*t1*≤*t2*› **by** *auto*
then have *t′*∈{*0*<..<*1*} **using** *t* **by** *auto*
then have *Re* (*g t′*) ≠ *Re z* **using** *Re-neq* ‹*s=0*› **by** *auto*
then show *False* **using** ‹*Re* (*g t′*) = *Re z*› **by** *simp*
qed
moreover have *False* **when** *t1*≥*t2*
proof −
have *continuous-on* {*t2*..*t1*} (λ*t*. *Re* (*g t*))
using *valid-path-imp-path*[*OF* ‹*valid-path g*›] *t* **unfolding** *path-def*

by (*metis* (*full-types*) *atLeastatMost-subset-iff continuous-on-Re continuous-on-subset*
eucl-less-le-not-le greaterThanLessThan-iff)
then obtain t' **where** t':$t' \leq t1$ $t' \geq t2$ *Re* (g t') = *Re z*
using *IVT2'*[*of* λ*t*. *Re* (*g t*) *t1* - *t2*] *t* ‹*t1*≥*t2*› **by** *auto*
then have $t' \in \{0 <..< 1\}$ **using** *t* **by** *auto*
then have *Re* (g t') ≠ *Re z* **using** *Re-neq* ‹*s*=*0*› **by** *auto*
then show *False* **using** ‹*Re* (g t') = *Re z*› **by** *simp*
qed
ultimately show *False* **by** *linarith*
qed
moreover have *?thesis* **when** *Re-pos*:∀ *t*∈{*0*<..<*1*}. *Re* (*g t*) > *Re z*
proof −
have *Re* (*winding-number g z*) = *0*
proof −
have ∀ *p*∈*path-image g*. *Re z* ≤ *Re p*
proof
fix *p* **assume** *p* ∈ *path-image g*
then obtain *t* **where** *0*≤*t* *t*≤*1* *p* = *g t* **unfolding** *path-image-def* **by** *auto*
have *Re z* ≤ *Re* (*g t*)
apply (*rule continuous-ge-on-closure*[*of* {*0*<..<*1*} λ*t*. *Re* (*g t*) *t Re z*,*simplified*])
subgoal using *valid-path-imp-path*[*OF* ‹*valid-path g*›,*unfolded path-def*]
by (*auto intro*:*continuous-intros*)
subgoal using ‹*0*≤*t*› ‹*t*≤*1*› **by** *auto*
subgoal for *x* **using** *that*[*rule-format*,*of x*] **by** *auto*
done
then show *Re z* ≤ *Re p* **using** ‹*p* = *g t*› **by** *auto*
qed
from *Re-winding-number-half-right*[*OF this* ‹*valid-path g*› ‹*z*∉*path-image g*›] *loop*
show *?thesis* **by** *auto*
qed
moreover have *cindex-pathE g z* = *0*
proof −
have *cindex-pathE g z* = *jumpF-pathstart g z* − *jumpF-pathfinish g z*
using *index-ends*[*OF* - ‹*valid-path g*›] *Re-neq* ‹*s*=*0*› **by** *auto*
moreover have *jumpF-pathstart g z* = *jumpF-pathfinish g z* **when** *Re* (*g* *0*) ≠ *Re z*
proof −
have *jumpF-pathstart g z* = *0*
using *jumpF-pathstart-eq-0*[*OF* ‹*path g*›] *that* **unfolding** *path-defs* **by** *auto*
moreover have *jumpF-pathfinish g z*=*0*
using *jumpF-pathfinish-eq-0*[*OF* ‹*path g*›] *that loop* **unfolding** *path-defs* **by** *auto*
ultimately show *?thesis* **by** *auto*
qed
moreover have *jumpF-pathstart g z* = *jumpF-pathfinish g z* **when** *Re* (*g*

0) = *Re z*
proof −
have [*simp*]:(*fR has-sgnx 1*) (*at-right 0*)
unfolding *fR-def has-sgnx-def eventually-at-right*
apply (*rule exI*[**where** *x=1*])
using *Re-pos* **by** *auto*
have [*simp*]:(*fR has-sgnx 1*) (*at-left 1*)
unfolding *fR-def has-sgnx-def eventually-at-left*
apply (*rule exI*[**where** *x=0*])
using *Re-pos* **by** *auto*
have *fI 0≠0*
proof (*rule ccontr*)
assume ¬ *fI 0 ≠ 0*
then have *g 0 =z* **using** ‹*Re* (*g 0*) = *Re z*›
unfolding *fI-def* **by** (*simp add*: *complex.expand*)
then show *False* **using** ‹*z ∉ path-image g*› **unfolding** *path-image-def*
by *auto*
qed
moreover have *?thesis* **when** *fI 0>0*
proof −
have *jumpF-pathstart g z = 1/2*
proof −
have (*LIM x at-right 0. fI x / fR x :> at-top*)
apply (*subst filterlim-divide-at-bot-at-top-iff*[*of - fI 0*])
using *that fI fR*[*OF* ‹*Re* (*g 0*) = *Re z*›] **by** *simp-all*
then show *?thesis* **unfolding** *jumpF-pathstart-def fI-def fR-def*
jumpF-def **by** *auto*
qed
moreover have *jumpF-pathfinish g z = 1/2*
proof −
have *fI 1>0* **using** *loop that* **unfolding** *path-defs fI-def* **by** *auto*
then have (*LIM x at-left 1. fI x / fR x :> at-top*)
apply (*subst filterlim-divide-at-bot-at-top-iff*[*of - fI 1*])
using *that fI fR*[*OF* ‹*Re* (*g 0*) = *Re z*›] **by** *simp-all*
then show *?thesis* **unfolding** *jumpF-pathfinish-def fI-def fR-def*
jumpF-def **by** *auto*
qed
ultimately show *?thesis* **by** *simp*
qed
moreover have *?thesis* **when** *fI 0<0*
proof −
have *jumpF-pathstart g z = − 1/2*
proof −
have (*LIM x at-right 0. fI x / fR x :> at-bot*)
apply (*subst filterlim-divide-at-bot-at-top-iff*[*of - fI 0*])
using *that fI fR*[*OF* ‹*Re* (*g 0*) = *Re z*›] **by** *simp-all*
then show *?thesis* **unfolding** *jumpF-pathstart-def fI-def fR-def*
jumpF-def **by** *auto*
qed

moreover have *jumpF-pathfinish g z* = − *1/2*
proof −
have *fI 1*<*0* **using** *loop that* **unfolding** *path-defs fI-def* **by** *auto*
then have (*LIM x at-left 1. fI x / fR x* :> *at-bot*)
apply (*subst filterlim-divide-at-bot-at-top-iff* [*of - fI 1*])
using *that fI fR*[*OF* ‹*Re* (*g 0*) = *Re z*›] **by** *simp-all*
then show *?thesis* **unfolding** *jumpF-pathfinish-def fI-def fR-def jumpF-def* **by** *auto*
qed
ultimately show *?thesis* **by** *simp*
qed
ultimately show *?thesis* **by** *linarith*
qed
ultimately show *?thesis* **by** *auto*
qed
ultimately show *?thesis* **by** *auto*
qed
moreover have *?thesis* **when** *Re-neg*:∀ *t*∈{*0*<..<*1*}. *Re* (*g t*) < *Re z*
proof −
have *Re* (*winding-number g z*) = *0*
proof −
have ∀ *p*∈*path-image g*. *Re z* ≥ *Re p*
proof
fix *p* **assume** *p* ∈ *path-image g*
then obtain *t* **where** *0*≤*t t*≤*1 p* = *g t* **unfolding** *path-image-def* **by** *auto*
have *Re z* ≥ *Re* (*g t*)
apply (*rule continuous-le-on-closure*[*of* {*0*<..<*1*} λ*t*. *Re* (*g t*) *t Re z*,*simplified*])
subgoal using *valid-path-imp-path*[*OF* ‹*valid-path g*›,*unfolded path-def*]
by (*auto intro*:*continuous-intros*)
subgoal using ‹*0*≤*t*› ‹*t*≤*1*› **by** *auto*
subgoal for *x* **using** *that*[*rule-format*,*of x*] **by** *auto*
done
then show *Re z* ≥ *Re p* **using** ‹*p* = *g t*› **by** *auto*
qed
from *Re-winding-number-half-left*[*OF this* ‹*valid-path g*› ‹*z*∉*path-image g*›] *loop*
show *?thesis* **by** *auto*
qed
moreover have *cindex-pathE g z* = *0*
proof −
have *cindex-pathE g z* = *jumpF-pathstart g z* − *jumpF-pathfinish g z*
using *index-ends*[*OF -* ‹*valid-path g*›] *Re-neq* ‹*s*=*0*› **by** *auto*
moreover have *jumpF-pathstart g z* = *jumpF-pathfinish g z* **when** *Re* (*g 0*) ≠ *Re z*
proof −
have *jumpF-pathstart g z* = *0*
using *jumpF-pathstart-eq-0*[*OF* ‹*path g*›] *that* **unfolding** *path-defs* **by** *auto*

moreover have *jumpF-pathfinish g z=0*
using *jumpF-pathfinish-eq-0*[*OF* ‹*path g*›] *that loop* **unfolding** *path-defs* **by** *auto*
ultimately show *?thesis* **by** *auto*
qed
moreover have *jumpF-pathstart g z* = *jumpF-pathfinish g z* **when** *Re* (*g 0*) = *Re z*
proof −
have [*simp*]:(*fR has-sgnx* − *1*) (*at-right 0*)
unfolding *fR-def has-sgnx-def eventually-at-right*
apply (*rule exI*[**where** *x=1*])
using *Re-neg* **by** *auto*
have [*simp*]:(*fR has-sgnx* − *1*) (*at-left 1*)
unfolding *fR-def has-sgnx-def eventually-at-left*
apply (*rule exI*[**where** *x=0*])
using *Re-neg* **by** *auto*
have *fI 0≠0*
proof (*rule ccontr*)
assume ¬ *fI 0* ≠ *0*
then have *g 0* =*z* **using** ‹*Re* (*g 0*) = *Re z*›
unfolding *fI-def* **by** (*simp add*: *complex.expand*)
then show *False* **using** ‹*z* ∉ *path-image g*› **unfolding** *path-image-def* **by** *auto*
qed
moreover have *?thesis* **when** *fI 0>0*
proof −
have *jumpF-pathstart g z* = − *1/2*
proof −
have (*LIM x at-right 0. fI x / fR x :> at-bot*)
apply (*subst filterlim-divide-at-bot-at-top-iff*[*of - fI 0*])
using *that fI fR*[*OF* ‹*Re* (*g 0*) = *Re z*›] **by** *simp-all*
then show *?thesis* **unfolding** *jumpF-pathstart-def fI-def fR-def jumpF-def* **by** *auto*
qed
moreover have *jumpF-pathfinish g z* = − *1/2*
proof −
have *fI 1>0* **using** *loop that* **unfolding** *path-defs fI-def* **by** *auto*
then have (*LIM x at-left 1. fI x / fR x :> at-bot*)
apply (*subst filterlim-divide-at-bot-at-top-iff*[*of - fI 1*])
using *that fI fR*[*OF* ‹*Re* (*g 0*) = *Re z*›] **by** *simp-all*
then show *?thesis* **unfolding** *jumpF-pathfinish-def fI-def fR-def jumpF-def* **by** *auto*
qed
ultimately show *?thesis* **by** *simp*
qed
moreover have *?thesis* **when** *fI 0<0*
proof −
have *jumpF-pathstart g z* = *1/2*
proof −

have (*LIM x at-right 0. fI x / fR x :> at-top*)
apply (*subst filterlim-divide-at-bot-at-top-iff*[*of - fI 0*])
using *that fI fR*[*OF* ‹*Re* (*g 0*) = *Re z*›] **by** *simp-all*
then show *?thesis* **unfolding** *jumpF-pathstart-def fI-def fR-def jumpF-def* **by** *auto*
qed
moreover have *jumpF-pathfinish g z* = *1/2*
proof −
have *fI 1*<*0* **using** *loop that* **unfolding** *path-defs fI-def* **by** *auto*
then have (*LIM x at-left 1. fI x / fR x :> at-top*)
apply (*subst filterlim-divide-at-bot-at-top-iff*[*of - fI 1*])
using *that fI fR*[*OF* ‹*Re* (*g 0*) = *Re z*›] **by** *simp-all*
then show *?thesis* **unfolding** *jumpF-pathfinish-def fI-def fR-def jumpF-def* **by** *auto*
qed
ultimately show *?thesis* **by** *simp*
qed
ultimately show *?thesis* **by** *linarith*
qed
ultimately show *?thesis* **by** *auto*
qed
ultimately show *?thesis* **by** *auto*
qed
ultimately show *?thesis* **by** *auto*
qed
moreover have *?goal* **when** *s*≠*0*
proof −
have *Re* (*g s*) = *Re z* **using** ‹*s* = *0* ∨ *Re* (*g s*) = *Re z*› *that* **by** *auto*
define *g′* **where** *g′* = *shiftpath s g*
have *2* ∗ *Re* (*winding-number g′ z*) = − *cindex-pathE g′ z*
proof (*rule winding-number-cindex-pathE-aux*)
show *Re* (*g′ 1*) = *Re z Re* (*g′ 0*) = *Re z*
using ‹*Re* (*g s*) = *Re z*› ‹*s*∈{*0..*<*1*}› ‹*s*≠*0*›
unfolding *g′-def shiftpath-def* **by** *simp-all*
show *valid-path g′*
using *valid-path-shiftpath*[*OF* ‹*valid-path g*› *loop*,*of s*,*folded g′-def*] ‹*s*∈{*0..*<*1*}›
by *auto*
show *z* ∉ *path-image g′*
using ‹*s* ∈ {*0..*<*1*}› *assms*(*3*) *g′-def loop path-image-shiftpath* **by** *fastforce*
show *finite-ReZ-segments g′ z*
using *finite-ReZ-segments-shiftpah*[*OF* ‹*finite-ReZ-segments g z*› - ‹*path g*› *loop*] ‹*s*∈{*0..*<*1*}›
unfolding *g′-def* **by** *auto*
qed
moreover have *winding-number g′ z* = *winding-number g z*
unfolding *g′-def*
apply (*rule winding-number-shiftpath*[*OF* ‹*path g*› ‹*z* ∉ *path-image g*› *loop*])
using ‹*s*∈{*0..*<*1*}› **by** *auto*
moreover have *cindex-pathE g′ z* = *cindex-pathE g z*

unfolding *g'-def*
apply (*rule cindex-pathE-shiftpath*[*OF* ‹*finite-ReZ-segments g z*› - ‹*path g*› *loop*])
using ‹*s*∈{*0..<1*}› **by** *auto*
ultimately show *?thesis* **by** *auto*
qed
ultimately have *?goal* **by** *auto*
moreover have *winding-number g z* ∈ ℤ
using *integer-winding-number*[*OF - loop* ‹*z*∉*path-image g*›] *valid-path-imp-path*[*OF* ‹*valid-path g*›]
by *auto*
ultimately show *winding-number g z* = − *cindex-pathE g z* / *2*
by (*metis add.right-neutral complex-eq complex-is-Int-iff mult-zero-right nonzero-mult-div-cancel-left of-real-0 zero-neq-numeral*)
qed

REMARK: The usual statement of Cauchy's Index theorem (i.e. Analytic Theory of Polynomials (2002): Theorem 11.1.3) is about the equality between the number of polynomial roots and the Cauchy index, which is the joint application of ⟦*finite-ReZ-segments ?g ?z*; *valid-path ?g*; *?z* ∉ *path-image ?g*; *pathfinish ?g* = *pathstart ?g*⟧ ⟹ *winding-number ?g ?z* = *complex-of-real* (− *cindex-pathE ?g ?z* / *2*) and ⟦*open ?S*; *connected ?S*; *?f holomorphic-on ?S* − *?poles*; *?h holomorphic-on ?S*; *valid-path ?g*; *pathfinish ?g* = *pathstart ?g*; *path-image ?g* ⊆ *?S* − {*w* ∈ *?S*. *?f w* = *0* ∨ *w* ∈ *?poles*}; ∀ *z*. *z* ∉ *?S* ⟶ *winding-number ?g z* = *0*; *finite* {*w* ∈ *?S*. *?f w* = *0* ∨ *w* ∈ *?poles*}; ∀ *p*∈*?S* ∩ *?poles*. *is-pole ?f p*⟧ ⟹ *contour-integral ?g* (λ*x*. *deriv ?f x* ∗ *?h x* / *?f x*) = *complex-of-real* (*2* ∗ *pi*) ∗ i ∗ (∑ *p*∈{*w* ∈ *?S*. *?f w* = *0* ∨ *w* ∈ *?poles*}. *winding-number ?g p* ∗ *?h p* ∗ *complex-of-int* (*zorder ?f p*)).

end

6 Evaluate winding numbers by calculating Cauchy indices

theory *Winding-Number-Eval* **imports**
Cauchy-Index-Theorem
HOL−Eisbach.Eisbach-Tools
begin

6.1 Misc

lemma *not-on-closed-segmentI*:
fixes *z*::*'a*::*euclidean-space*
assumes *norm* (*z* − *a*) $*_R$ (*b* − *z*) ≠ *norm* (*b* − *z*) $*_R$ (*z* − *a*)
shows *z* ∉ *closed-segment a b*
using *assms* **by** (*auto simp add*:*between-mem-segment*[*symmetric*] *between-norm*)

lemma *not-on-closed-segmentI-complex*:
  fixes *z*::*complex*
  assumes (*Re b* − *Re z*) ∗ (*Im z* − *Im a*) ≠ (*Im b* − *Im z*) ∗ (*Re z* − *Re a*)
  shows *z* ∉ *closed-segment a b*
proof (*cases z*≠*a* ∧ *z*≠*b*)
  case *True*
  then have *cmod* (*z* − *a*)≠*0 cmod* (*b* − *z*)≠*0* **by** *auto*
  then have (*Re b* − *Re z*) ∗ (*Im z* − *Im a*) = (*Im b* − *Im z*) ∗ (*Re z* − *Re a*)
when
    cmod (*z* − *a*) ∗ (*Re b* − *Re z*) = *cmod* (*b* − *z*) ∗ (*Re z* − *Re a*)
    cmod (*z* − *a*) ∗ (*Im b* − *Im z*) = *cmod* (*b* − *z*) ∗ (*Im z* − *Im a*)
    using *that* **by** *algebra*
  then show *?thesis* **using** *assms*
    apply (*intro not-on-closed-segmentI*)
    by (*auto simp add*:*scaleR-complex.ctr simp del*:*Complex-eq*)
next
  case *False*
  then have (*Re b* − *Re z*) ∗ (*Im z* − *Im a*) = (*Im b* − *Im z*) ∗ (*Re z* − *Re a*)
by *auto*
  then have *False* **using** *assms* **by** *auto*
  then show *?thesis* **by** *auto*
qed

6.2 finite intersection with the two axes

definition *finite-axes-cross*::(*real* ⇒ *complex*) ⇒ *complex* ⇒ *bool* **where**
  finite-axes-cross g z = *finite* {*t*. (*Re* (*g t*−*z*) = *0* ∨ *Im* (*g t*−*z*) = *0*) ∧ *0* ≤ *t* ∧ *t* ≤ *1*}

lemma *finite-cross-intros*:
  ⟦*Re a*≠*Re z* ∨ *Re b* ≠*Re z*; *Im a*≠*Im z* ∨ *Im b*≠*Im z*⟧⟹*finite-axes-cross* (*linepath a b*) *z*
  ⟦*st* ≠ *tt*; *r* ≠ *0*⟧ ⟹ *finite-axes-cross* (*part-circlepath z0 r st tt*) *z*
  ⟦*finite-axes-cross g1 z*;*finite-axes-cross g2 z*⟧ ⟹ *finite-axes-cross* (*g1*+++*g2*) *z*
proof −
  assume *asm*:*Re a*≠*Re z* ∨ *Re b* ≠*Re z Im a*≠*Im z* ∨ *Im b*≠*Im z*
  let *?S1*={*t*. *Re* (*linepath a b t*−*z*) = *0* ∧ *0* ≤ *t* ∧ *t* ≤ *1*}
  and *?S2*={*t*. *Im* (*linepath a b t*−*z*) = *0* ∧ *0* ≤ *t* ∧ *t* ≤ *1*}
  have *finite ?S1*
    using *linepath-half-finite-inter*[*of a Complex 1 0 Re z b*] *asm*(*1*)
    by (*auto simp add*:*inner-complex-def*)
  moreover have *finite ?S2*
    using *linepath-half-finite-inter*[*of a Complex 0 1 Im z b*] *asm*(*2*)
    by (*auto simp add*:*inner-complex-def*)
  moreover have {*t*. (*Re* (*linepath a b t*−*z*) = *0* ∨ *Im* (*linepath a b t*−*z*) = *0*) ∧ *0* ≤ *t* ∧ *t* ≤ *1*}
      = *?S1* ∪ *?S2*
    by *fast*
  ultimately show *finite-axes-cross* (*linepath a b*) *z*

```
    unfolding finite-axes-cross-def by force
next
  assume asm: st ≠tt r≠0
  let ?S1={t. Re (part-circlepath z0 r st tt t−z) = 0 ∧ 0 ≤ t ∧ t ≤ 1}
  and ?S2={t. Im (part-circlepath z0 r st tt t−z) = 0 ∧ 0 ≤ t ∧ t ≤ 1}
  have finite ?S1
    using part-circlepath-half-finite-inter[of st tt r Complex 1 0 z0 Re z] asm
    by (auto simp add:inner-complex-def Complex-eq-0)
  moreover have finite ?S2
    using part-circlepath-half-finite-inter[of st tt r Complex 0 1 z0 Im z] asm
    by (auto simp add:inner-complex-def Complex-eq-0)
  moreover have {t. (Re (part-circlepath z0 r st tt t−z) = 0
      ∨ Im (part-circlepath z0 r st tt t−z) = 0) ∧ 0 ≤ t ∧ t ≤ 1} = ?S1 ∪ ?S2
    by fast
  ultimately show finite-axes-cross (part-circlepath z0 r st tt) z
    unfolding finite-axes-cross-def by auto
next
  assume asm:finite-axes-cross g1 z finite-axes-cross g2 z
  let ?g1R={t. Re (g1 t−z) = 0 ∧ 0 ≤ t ∧ t ≤ 1}
  and ?g1I={t. Im (g1 t−z) = 0 ∧ 0 ≤ t ∧ t ≤ 1}
  and ?g2R={t. Re (g2 t−z) = 0 ∧ 0 ≤ t ∧ t ≤ 1}
  and ?g2I={t. Im (g2 t−z) = 0 ∧ 0 ≤ t ∧ t ≤ 1}
  have finite ?g1R finite ?g1I
  proof −
    have {t. (Re (g1 t − z) = 0 ∨ Im (g1 t − z) = 0) ∧ 0 ≤ t ∧ t ≤ 1} = ?g1R
∪ ?g1I
      by force
    then have finite (?g1R ∪ ?g1I)
      using asm(1) unfolding finite-axes-cross-def by auto
    then show finite ?g1R finite ?g1I by blast+
  qed
  have finite ?g2R finite ?g2I
  proof −
    have {t. (Re (g2 t − z) = 0 ∨ Im (g2 t − z) = 0) ∧ 0 ≤ t ∧ t ≤ 1} = ?g2R
∪ ?g2I
      by force
    then have finite (?g2R ∪ ?g2I)
      using asm(2) unfolding finite-axes-cross-def by auto
    then show finite ?g2R finite ?g2I by blast+
  qed
  let ?S1 = {t. Re ((g1 +++ g2) t − z) = 0 ∧ 0 ≤ t ∧ t ≤ 1}
  and ?S2 = {t. Im ((g1 +++ g2) t − z) = 0 ∧ 0 ≤ t ∧ t ≤ 1}
  have finite ?S1
    using finite-half-joinpaths-inter[of g1 Complex 1 0 Re z g2,simplified]
      ‹finite ?g1R› ‹finite ?g2R›
    by (auto simp add:inner-complex-def)
  moreover have finite ?S2
    using finite-half-joinpaths-inter[of g1 Complex 0 1 Im z g2,simplified]
      ‹finite ?g1I› ‹finite ?g2I›
```

by (*auto simp add*:*inner-complex-def*)
moreover have {*t*. (*Re* ((*g1* +++ *g2*) *t* − *z*) = *0* ∨ *Im* ((*g1* +++ *g2*) *t* − *z*) = *0*) ∧ *0* ≤ *t* ∧ *t* ≤ *1*}
= *?S1* ∪ *?S2*
by *force*
ultimately show *finite-axes-cross* (*g1* +++ *g2*) *z*
unfolding *finite-axes-cross-def*
by *auto*
qed

lemma *cindex-path-joinpaths*:
assumes *finite-axes-cross g1 z finite-axes-cross g2 z*
and *path g1 path g2 pathfinish g1* = *pathstart g2 pathfinish g1*≠*z*
shows *cindex-path* (*g1*+++*g2*) *z* = *cindex-path g1 z* + *jumpF-pathstart g2 z*
− *jumpF-pathfinish g1 z* + *cindex-path g2 z*
proof −
define *h12* **where** *h12* = (λ*t*. *Im* ((*g1*+++*g2*) *t* − *z*) / *Re* ((*g1*+++*g2*) *t* − *z*))
let *?h* = λ*g*. λ*t*. *Im* (*g t* − *z*) / *Re* (*g t* − *z*)
have *cindex-path* (*g1*+++*g2*) *z* = *cindex 0 1 h12*
unfolding *cindex-path-def h12-def* **by** *simp*
also have ... = *cindex 0* (*1*/*2*) *h12* + *jump h12* (*1*/*2*) + *cindex* (*1*/*2*) *1 h12*
proof (*rule cindex-combine*)
have *finite-axes-cross* (*g1*+++*g2*) *z* **using** *assms* **by** (*auto intro*:*finite-cross-intros*)
then have *finite* {*t*. *Re* ((*g1*+++*g2*) *t* − *z*) = *0* ∧ *0*≤*t* ∧ *t*≤*1*}
unfolding *finite-axes-cross-def* **by** (*auto elim*:*rev-finite-subset*)
moreover have *jump h12 t* = *0* **when** *Re* ((*g1* +++ *g2*) *t* − *z*) ≠ *0 0* < *t t* < *1* **for** *t*
apply (*rule jump-Im-divide-Re-0*[*of* λ*t*. (*g1*+++*g2*) *t*− *z*,*folded h12-def*,*OF* - *that*])
using *assms* **by** (*auto intro*:*path-offset*)
ultimately show *finite* {*x*. *jump h12 x* ≠ *0* ∧ *0* < *x* ∧ *x* < *1*}
apply (*elim rev-finite-subset*)
by *auto*
qed *auto*
also have ... = *cindex-path g1 z* + *jumpF-pathstart g2 z*
− *jumpF-pathfinish g1 z* + *cindex-path g2 z*
proof −
have *jump h12* (*1*/*2*) = *jumpF-pathstart g2 z* − *jumpF-pathfinish g1 z*
proof −
have *jump h12* (*1* / *2*) = *jumpF h12* (*at-right* (*1* / *2*)) − *jumpF h12* (*at-left* (*1* / *2*))
proof (*cases Re* ((*g1*+++*g2*) (*1*/*2*) − *z*) = *0*)
case *False*
have *jump h12* (*1* / *2*) = *0*
unfolding *h12-def*
apply (*rule jump-Im-divide-Re-0*)
using *assms False* **by** (*auto intro*:*path-offset*)
moreover have *jumpF h12* (*at-right* (*1*/*2*)) = *0*

unfolding *h12-def*
apply (*intro jumpF-im-divide-Re-0*)
subgoal using *assms* **by** (*auto intro*:*path-offset*)
subgoal using *assms*(*5*−*6*) *False* **unfolding** *joinpaths-def pathfinish-def pathstart-def* **by** *auto*
by *auto*
moreover have *jumpF h12* (*at-left* (*1*/*2*)) = *0*
unfolding *h12-def*
apply (*intro jumpF-im-divide-Re-0*)
subgoal using *assms* **by** (*auto intro*:*path-offset*)
subgoal using *assms*(*5*−*6*) *False* **unfolding** *joinpaths-def pathfinish-def pathstart-def* **by** *auto*
by *auto*
ultimately show *?thesis* **by** *auto*
next
case *True*
then have *Im* ((*g1* +++ *g2*) (*1* / *2*) − *z*) ≠ *0*
using *assms*(*5*,*6*)
by (*metis* (*no-types*, *opaque-lifting*) *Re-divide-numeral complex-Re-numeral complex-eq*
divide-self-if joinpaths-def minus-complex.simps mult.commute mult.left-neutral
numeral-One pathfinish-def pathstart-def right-minus-eq times-divide-eq-left zero-neq-numeral)
show *?thesis*
proof (*rule jump-jumpF*[*of - h12 sgnx h12* (*at-left* (*1*/*2*)) *sgnx h12* (*at-right* (*1*/*2*))])
define *g* **where** *g*=(λ*t*. (*g1* +++ *g2*) *t* − *z*)
have *h12-def*:*h12* = (λ*t*. *Im*(*g t*)/*Re*(*g t*)) **unfolding** *h12-def g-def* **by** *simp*
have *path g* **using** *assms* **unfolding** *g-def* **by** (*auto intro*!:*path-offset*)
then have *isCont* (λ*t*. *Im* (*g t*)) (*1* / *2*) *isCont* (λ*t*. *Re* (*g t*)) (*1* / *2*)
unfolding *path-def* **by** (*auto intro*!:*continuous-intros continuous-on-interior*)
moreover have *Im* (*g* (*1*/*2*)) ≠*0*
using ‹*Im* ((*g1* +++ *g2*) (*1* / *2*) − *z*) ≠ *0*› **unfolding** *g-def* **.**
ultimately show *isCont* (*inverse* ∘ *h12*) (*1* / *2*)
unfolding *h12-def comp-def*
by (*auto intro*!: *continuous-intros*)

define *l* **where** *l* ≡ *sgnx h12* (*at-left* (*1*/*2*))
define *r* **where** *r* ≡ *sgnx h12* (*at-right* (*1*/*2*))
have ∗:*continuous-on* ({*0*<..<*1*}− {*t*. *h12 t* = *0* ∧ *0* < *t* ∧ *t* < *1*}) *h12*
using ‹*path g*›[*unfolded path-def*] **unfolding** *h12-def*
apply (*auto intro*!: *continuous-intros*)
by (*auto elim*:*continuous-on-subset*)
have ∗∗:*finite* {*t*. *h12 t* = *0* ∧ *0* < *t* ∧ *t* < *1*}
proof −
have *finite-axes-cross* (*g1* +++ *g2*) *z*
using *assms*(*1*,*2*) *finite-cross-intros*(*3*)[*of g1 z g2*] **by** *auto*

then have *finite* {*t*. (*Re* (*g t*) = *0* ∨ *Im* (*g t*) = *0*) ∧ *0* < *t* ∧ *t* < *1*}
unfolding *finite-axes-cross-def g-def*
apply (*elim rev-finite-subset*)
by *auto*
then show *?thesis* **unfolding** *h12-def*
by (*simp add*:*disj-commute*)
qed
have *h12 sgnx-able at-left* (*1/2*) *l* ≠ *0 h12 sgnx-able at-right* (*1/2*) *r* ≠ *0*
unfolding *l-def r-def* **using** *finite-sgnx-at-left-at-right*[*OF* ∗∗ ∗ ∗∗]
by *auto*
then show (*h12 has-sgnx l*) (*at-left* (*1/2*)) (*h12 has-sgnx r*) (*at-right* (*1/2*)) *l*≠*0 r*≠*0*
unfolding *l-def r-def* **by** (*auto elim*:*sgnx-able-sgnx*)
qed
qed
moreover have *jumpF h12* (*at-right* (*1/2*)) = *jumpF-pathstart g2 z*
proof −
have *jumpF h12* (*at-right* (*1* / *2*)) = *jumpF* (*h12* ∘ (λ*x*. *x* / *2* + *1* / *2*)) (*at-right 0*)
using *jumpF-linear-comp*[*of 1/2 h12 1/2 0*,*simplified*] **by** *simp*
also have *jumpF* (*h12* ∘ (λ*x*. *x* / *2* + *1* / *2*)) (*at-right 0*) = *jumpF-pathstart g2 z*
unfolding *h12-def jumpF-pathstart-def*
proof (*rule jumpF-cong*)
show ∀$_F$ *x in at-right 0*. ((λ*t*. *Im* ((*g1* +++ *g2*) *t* − *z*) / *Re* ((*g1* +++ *g2*) *t* − *z*))
∘ (λ*x*. *x* / *2* + *1* / *2*)) *x* = *Im* (*g2 x* − *z*) / *Re* (*g2 x* − *z*)
unfolding *eventually-at-right*
apply (*intro exI*[**where** *x*=*1/2*])
unfolding *joinpaths-def* **by** *auto*
qed *simp*
finally show *?thesis* **.**
qed
moreover have *jumpF h12* (*at-left* (*1* / *2*)) = *jumpF-pathfinish g1 z*
proof −
have *jumpF h12* (*at-left* (*1* / *2*)) = *jumpF* (*h12* ∘ (λ*x*. *x* / *2*)) (*at-left 1*)
using *jumpF-linear-comp*[*of 1/2 h12 0 1*,*simplified*] **by** *simp*
also have *jumpF* (*h12* ∘ (λ*x*. *x* / *2*)) (*at-left 1*) = *jumpF-pathfinish g1 z*
unfolding *h12-def jumpF-pathfinish-def*
proof (*rule jumpF-cong*)
show ∀$_F$ *x in at-left 1*. ((λ*t*. *Im* ((*g1* +++ *g2*) *t* − *z*) / *Re* ((*g1* +++ *g2*) *t* − *z*))
∘ (λ*x*. *x* / *2*)) *x* = *Im* (*g1 x* − *z*) / *Re* (*g1 x* − *z*)
unfolding *eventually-at-left*
apply (*intro exI*[**where** *x*=*1/2*])
unfolding *joinpaths-def* **by** *auto*
qed *simp*
finally show *?thesis* **.**
qed

ultimately show *?thesis* **by** *auto*
qed
moreover have *cindex 0 (1 / 2) h12 = cindex-path g1 z*
proof −
have *cindex 0 (1 / 2) h12 = cindex 0 1 (h12 ∘ (λx. x / 2))*
using *cindex-linear-comp[of 1/2 0 1 h12 0,simplified,symmetric]* **.**
also have *... = cindex-path g1 z*
proof −
let *?g =* (*λt. Im (g1 t − z) / Re (g1 t − z)*)
have ∗:*jump (h12 ∘ (λx. x / 2)) x = jump ?g x* **when** *0<x x<1* **for** *x*
unfolding *h12-def*
proof (*rule jump-cong*)
show *∀ $_F$ x in at x. ((λt. Im ((g1 +++ g2) t − z) / Re ((g1 +++ g2) t − z))*
∘ (λx. x / 2)) x = Im (g1 x − z) / Re (g1 x − z)
unfolding *eventually-at joinpaths-def comp-def* **using** *that*
apply (*intro exI*[**where** *x=(1−x)/2*])
by (*auto simp add: dist-norm*)
qed *simp*
then have *{x. jump (h12 ∘ (λx. x / 2)) x ≠ 0 ∧ 0 < x ∧ x < 1}*
= {x. jump ?g x ≠ 0 ∧ 0 < x ∧ x < 1}
by *auto*
then show *?thesis*
unfolding *cindex-def cindex-path-def*
apply (*elim sum.cong*)
by (*auto simp add:*∗)
qed
finally show *?thesis* **.**
qed
moreover have *cindex (1 / 2) 1 h12 = cindex-path g2 z*
proof −
have *cindex (1 / 2) 1 h12 = cindex 0 1 (h12 ∘ (λx. x / 2 + 1 / 2))*
using *cindex-linear-comp[of 1/2 0 1 h12 1/2,simplified,symmetric]* **.**
also have *... = cindex-path g2 z*
proof −
let *?g =* (*λt. Im (g2 t − z) / Re (g2 t − z)*)
have ∗:*jump (h12 ∘ (λx. x / 2+1/2)) x = jump ?g x* **when** *0<x x<1* **for** *x*
unfolding *h12-def*
proof (*rule jump-cong*)
show *∀ $_F$ x in at x. ((λt. Im ((g1 +++ g2) t − z) / Re ((g1 +++ g2) t − z))*
∘ (λx. x / 2+1/2)) x = Im (g2 x − z) / Re (g2 x − z)
unfolding *eventually-at joinpaths-def comp-def* **using** *that*
apply (*intro exI*[**where** *x=x/2*])
by (*auto simp add: dist-norm*)
qed *simp*
then have *{x. jump (h12 ∘ (λx. x / 2+1/2)) x ≠ 0 ∧ 0 < x ∧ x < 1}*
= {x. jump ?g x ≠ 0 ∧ 0 < x ∧ x < 1}

```
        by auto
      then show ?thesis
        unfolding cindex-def cindex-path-def
        apply (elim sum.cong)
        by (auto simp add:∗)
    qed
    finally show ?thesis .
  qed
  ultimately show ?thesis by simp
 qed
 finally show ?thesis .
qed
```

6.3 More lemmas related *cindex-pathE* / *jumpF-pathstart* / *jumpF-pathfinish*

```
lemma cindex-pathE-linepath:
 assumes z∉closed-segment a b
 shows cindex-pathE (linepath a b) z = (
  let c1 = Re a − Re z;
      c2 = Re b − Re z;
      c3 = Im a ∗ Re b + Re z ∗ Im b + Im z ∗ Re a − Im z ∗ Re b − Im b ∗
Re a − Re z ∗ Im a;
      d1 = Im a − Im z;
      d2 = Im b − Im z
  in if (c1>0 ∧ c2<0) ∨ (c1<0 ∧ c2>0) then
       (if c3>0 then 1 else −1)
     else
       (if (c1=0 ←→ c2≠0) ∧ (c1=0 −→d1≠0) ∧ (c2=0 −→ d2≠0) then
         if (c1=0 ∧ (c2 >0 ←→ d1>0)) ∨ (c2=0 ∧ (c1 >0 ←→ d2<0)) then
1/2 else −1/2
       else 0))
proof −
 define c1 c2 where c1=Re a − Re z and c2=Re b − Re z
 define d1 d2 where d1=Im a − Im z and d2=Im b − Im z
 let ?g = linepath a b
 have ?thesis when ¬ ((c1>0 ∧ c2<0) ∨ (c1<0 ∧ c2>0))
 proof −
  have Re a= Re z ∧ Re b=Re z
    when 0<t t<1 and asm:(1−t)∗Re a + t ∗ Re b = Re z for t
    unfolding c1-def c2-def using that
  proof −
    have ?thesis when c1≤0 c1≥0
    proof −
      have Re a=Re z using that unfolding c1-def by auto
      then show ?thesis using ‹0<t› ‹t<1› asm
        apply (cases Re b Re z rule:linorder-cases)
          apply (auto simp add:field-simps)
        done
    qed
```

moreover have *?thesis* **when** *c1≤0 c2≤0*
proof −
have *False* **when** *c1<0*
proof −
have (*1* − *t*) ∗ *Re a* < (*1* − *t*) ∗ *Re z*
using ‹*t<1*› ‹*c1<0*› **unfolding** *c1-def* **by** *auto*
moreover have *t* ∗ *Re b* ≤ *t*∗ *Re z* **using** ‹*t>0*› ‹*c2≤0*› **unfolding** *c2-def* **by** *auto*
ultimately have (*1* − *t*) ∗ *Re a* + *t* ∗ *Re b* < (*1* − *t*) ∗ *Re z* + *t* ∗ *Re z*
by *auto*
thus *False* **using** *asm* **by** (*auto simp add*:*algebra-simps*)
qed
moreover have *False* **when** *c2<0*
proof −
have (*1* − *t*) ∗ *Re a* ≤ (*1* − *t*) ∗ *Re z*
using ‹*t<1*› ‹*c1≤0*› **unfolding** *c1-def* **by** *auto*
moreover have *t* ∗ *Re b* < *t*∗ *Re z* **using** ‹*t>0*› ‹*c2<0*› **unfolding** *c2-def* **by** *auto*
ultimately have (*1* − *t*) ∗ *Re a* + *t* ∗ *Re b* < (*1* − *t*) ∗ *Re z* + *t* ∗ *Re z*
by *auto*
thus *False* **using** *asm* **by** (*auto simp add*:*algebra-simps*)
qed
ultimately show *?thesis* **using** *that* **unfolding** *c1-def c2-def* **by** *argo*
qed
moreover have *?thesis* **when** *c2≤0 c2≥0*
proof −
have *Re b=Re z* **using** *that* **unfolding** *c2-def* **by** *auto*
then have (*1* − *t*) ∗ *Re a* = (*1*−*t*)∗*Re z* **using** *asm* **by** (*auto simp add*:*field-simps*)
then have *Re a= Re z* **using** ‹*t<1*› **by** *auto*
then show *?thesis* **using** ‹*Re b=Re z*› **by** *auto*
qed
moreover have *?thesis* **when** *c1≥0 c2≥0*
proof −
have *False* **when** *c1>0*
proof −
have (*1* − *t*) ∗ *Re a* > (*1* − *t*) ∗ *Re z*
using ‹*t<1*› ‹*c1>0*› **unfolding** *c1-def* **by** *auto*
moreover have *t* ∗ *Re b* ≥ *t*∗ *Re z* **using** ‹*t>0*› ‹*c2≥0*› **unfolding** *c2-def* **by** *auto*
ultimately have (*1* − *t*) ∗ *Re a* + *t* ∗ *Re b* > (*1* − *t*) ∗ *Re z* + *t* ∗ *Re z*
by *auto*
thus *False* **using** *asm* **by** (*auto simp add*:*algebra-simps*)
qed
moreover have *False* **when** *c2>0*
proof −
have (*1* − *t*) ∗ *Re a* ≥ (*1* − *t*) ∗ *Re z*
using ‹*t<1*› ‹*c1≥0*› **unfolding** *c1-def* **by** *auto*
moreover have *t* ∗ *Re b* > *t*∗ *Re z* **using** ‹*t>0*› ‹*c2>0*› **unfolding** *c2-def*

by *auto*
ultimately have $(1 - t) * Re\ a + t * Re\ b > (1 - t) * Re\ z + t * Re\ z$
by *auto*
thus *False* using *asm* by (*auto simp add*:*algebra-simps*)
qed
ultimately show *?thesis* using *that* unfolding *c1-def c2-def* by *argo*
qed
moreover have $c1 \leq 0 \vee c2 \geq 0$ $c1 \geq 0 \vee c2 \leq 0$ using ‹¬ $((c1>0 \wedge c2<0) \vee (c1<0 \wedge c2>0))$› by *auto*
ultimately show *?thesis* by *fast*
qed
then have $(\forall t.\ 0<t \wedge t<1 \longrightarrow Re(linepath\ a\ b\ t - z) \neq 0) \vee (c1=0 \wedge c2=0)$

using *that* unfolding *linepath-def c1-def c2-def* by *auto*
moreover have *?thesis* when *asm*: $\forall t.\ 0<t \wedge t<1 \longrightarrow Re(linepath\ a\ b\ t - z) \neq 0$
and $\neg\ (c1=0 \wedge c2=0)$
proof −
have *cindex-ends*:*cindex-pathE ?g z = jumpF-pathstart ?g z − jumpF-pathfinish ?g z*
proof −
define *f* where $f=(\lambda t.\ Im\ (linepath\ a\ b\ t - z)\ /\ Re\ (linepath\ a\ b\ t - z))$
define *left* where $left = \{x.\ jumpF\ f\ (at\text{-}left\ x) \neq 0 \wedge 0 < x \wedge x \leq 1\}$
define *right* where $right = \{x.\ jumpF\ f\ (at\text{-}right\ x) \neq 0 \wedge 0 \leq x \wedge x < 1\}$
have *jumpF-nz*: $jumpF\ f\ (at\text{-}left\ x) = 0$ $jumpF\ f\ (at\text{-}right\ x) = 0$
when $0<x$ $x<1$ for *x*
proof −
have *isCont f x* unfolding *f-def*
using *asm*[*rule-format*,*of x*] *that*
by (*auto intro*!:*continuous-intros isCont-Im isCont-Re*)
then have *continuous* (*at-left x*) *f continuous* (*at-right x*) *f*
using *continuous-at-split* by *blast*+
then show $jumpF\ f\ (at\text{-}left\ x) = 0$ $jumpF\ f\ (at\text{-}right\ x) = 0$
using *jumpF-not-infinity* by *auto*
qed
have *cindex-pathE ?g z* = *sum* ($\lambda x.\ jumpF\ f\ (at\text{-}right\ x)$) *right*
− *sum* ($\lambda x.\ jumpF\ f\ (at\text{-}left\ x)$) *left*
unfolding *cindex-pathE-def cindexE-def right-def left-def*
by (*fold f-def*,*simp*)
moreover have *sum* ($\lambda x.\ jumpF\ f\ (at\text{-}right\ x)$) *right* = *jumpF-pathstart ?g z*
proof (*cases* $jumpF\ f\ (at\text{-}right\ 0) = 0$)
case *True*
hence *False* if $x \in right$ for *x* using *that*
by (*cases* $x = 0$) (*auto simp*: *jumpF-nz right-def*)
then have $right = \{\}$ by *blast*
then show *?thesis*
unfolding *jumpF-pathstart-def* using *True*

```
      apply (fold f-def)
      by auto
    next
      case False
      hence x ∈ right ⟷ x = 0 for x using that
        by (cases x = 0) (auto simp: jumpF-nz right-def)
      then have right = {0} by blast
      then show ?thesis
        unfolding jumpF-pathstart-def using False
        apply (fold f-def)
        by auto
    qed
    moreover have sum (λx. jumpF f (at-left x)) left = jumpF-pathfinish ?g z
    proof (cases jumpF f (at-left 1) = 0)
      case True
      then have left = {}
        unfolding left-def using jumpF-nz by force
      then show ?thesis
        unfolding jumpF-pathfinish-def using True
        apply (fold f-def)
        by auto
    next
      case False
      then have left = {1}
        unfolding left-def using jumpF-nz by force
      then show ?thesis
        unfolding jumpF-pathfinish-def using False
        apply (fold f-def)
        by auto
    qed
    ultimately show ?thesis by auto
  qed
  moreover have jF-start:jumpF-pathstart ?g z =
      (if c1=0 ∧ c2 ≠0 ∧ d1 ≠0 then
        if c2 >0 ⟷ d1 > 0 then 1/2 else −1/2
      else
        0)
  proof −
    define f where f=(λt. (Im b − Im a )∗ t + d1)
    define g where g=(λt. (Re b − Re a )∗ t + c1)
   have jump-eq:jumpF-pathstart (linepath a b) z = jumpF (λt. f t/g t) (at-right
0)
      unfolding jumpF-pathstart-def f-def linepath-def g-def d1-def c1-def
      by (auto simp add:algebra-simps)
    have ?thesis when ¬ (c1 =0 ∧ c2 ≠0 ∧ d1 ≠0)
    proof −
      have c2=0 ⟶ c1≠0 using ‹¬ (c1=0 ∧ c2=0)› by auto
      moreover have d1 =0 ⟶ c1≠0
      proof (rule ccontr)
```

```
        assume ¬ (d1 = 0 ⟶ c1 ≠ 0)
        then have a=z unfolding d1-def c1-def by (simp add: complex-eqI)
        then have z∈path-image (linepath a b) by auto
        then show False using ‹z∉closed-segment a b› by auto
      qed
      moreover have ?thesis when c1≠0
      proof −
        have jumpF (λt. f t/g t) (at-right 0) = 0
          apply (rule jumpF-not-infinity)
           apply (unfold f-def g-def)
          using that by (auto intro!: continuous-intros)
        then show ?thesis using jump-eq using that by auto
      qed
      ultimately show ?thesis using that by blast
    qed
    moreover have ?thesis when c1=0 c2 ≠0 d1 ≠0 c2 >0 ⟷ d1 > 0
    proof −
      have (LIM x at-right 0. f x / g x :> at-top)
      proof −
        have (f ⟶ d1) (at-right 0)
          unfolding f-def by (auto intro!: tendsto-eq-intros)
        moreover have (g ⟶ 0) (at-right 0)
          unfolding g-def using ‹c1=0› by (auto intro!: tendsto-eq-intros)
        moreover have (g has-sgnx sgn d1) (at-right 0)
        proof −
          have (g has-sgnx sgn (c2−c1)) (at-right 0)
            unfolding g-def
            apply (rule has-sgnx-derivative-at-right)
               subgoal unfolding c2-def c1-def d1-def by (auto intro!: deriva-
tive-eq-intros)
            subgoal using ‹c1=0› by auto
            subgoal using ‹c1=0› ‹c2≠0› by auto
            done
          moreover have sgn (c2−c1) = sgn d1 using that by fastforce
          ultimately show ?thesis by auto
        qed
        ultimately show ?thesis
           using filterlim-divide-at-bot-at-top-iff[of f d1 at-right 0 g] ‹d1≠0› by
auto
      qed
      then have jumpF (λt. f t/g t) (at-right 0) = 1/2 unfolding jumpF-def
by auto
      then show ?thesis using that jump-eq by auto
    qed
    moreover have ?thesis when c1=0 c2 ≠0 d1 ≠0 ¬ c2 >0 ⟷ d1 > 0
    proof −
      have (LIM x at-right 0. f x / g x :> at-bot)
      proof −
        have (f ⟶ d1) (at-right 0)
```

```
            unfolding f-def by (auto intro!: tendsto-eq-intros)
          moreover have (g ⟶ 0) (at-right 0)
            unfolding g-def using ‹c1=0› by (auto intro!: tendsto-eq-intros)
          moreover have (g has-sgnx − sgn d1) (at-right 0)
          proof −
            have (g has-sgnx sgn (c2−c1)) (at-right 0)
              unfolding g-def
              apply (rule has-sgnx-derivative-at-right)
                subgoal unfolding c2-def c1-def d1-def by (auto intro!: derivative-eq-intros)
              subgoal using ‹c1=0› by auto
              subgoal using ‹c1=0› ‹c2≠0› by auto
              done
            moreover have sgn (c2−c1) = − sgn d1 using that by fastforce
            ultimately show ?thesis by auto
          qed
          ultimately show ?thesis
             using filterlim-divide-at-bot-at-top-iff[of f d1 at-right 0 g] ‹d1≠0› by auto
        qed
       then have jumpF (λt. f t/g t) (at-right 0) = − 1/2 unfolding jumpF-def by auto
        then show ?thesis using that jump-eq by auto
      qed
      ultimately show ?thesis by fast
    qed
    moreover have jF-finish:jumpF-pathfinish ?g z =
        (if c2=0 ∧ c1 ≠0 ∧ d2 ≠0 then
          if c1 >0 ⟷ d2 > 0 then 1/2 else −1/2
        else
          0)
    proof −
      define f where f=(λt. (Im b − Im a )∗ t + (Im a − Im z))
      define g where g=(λt. (Re b − Re a )∗ t + (Re a − Re z))
     have jump-eq:jumpF-pathfinish (linepath a b) z = jumpF (λt. f t/g t) (at-left 1)
        unfolding jumpF-pathfinish-def f-def linepath-def g-def d1-def c1-def
        by (auto simp add:algebra-simps)
      have ?thesis when ¬ (c2 =0 ∧ c1 ≠0 ∧ d2 ≠0)
      proof −
        have c1=0 ⟶ c2≠0 using ‹¬ (c1=0 ∧ c2=0)› by auto
        moreover have d2 =0 ⟶ c2≠0
        proof (rule ccontr)
          assume ¬ (d2 = 0 ⟶ c2 ≠ 0)
          then have b=z unfolding d2-def c2-def by (simp add: complex-eqI)
          then have z∈path-image (linepath a b) by auto
          then show False using ‹z∉closed-segment a b› by auto
        qed
        moreover have ?thesis when c2≠0
```

proof −
have *jumpF* (λ*t*. *f t*/*g t*) (*at-left 1*) = *0*
apply (*rule jumpF-not-infinity*)
apply (*unfold f-def g-def*)
using *that* **unfolding** *c2-def* **by** (*auto intro*!: *continuous-intros*)
then show *?thesis* **using** *jump-eq* **using** *that* **by** *auto*
qed
ultimately show *?thesis* **using** *that* **by** *blast*
qed
moreover have *?thesis* **when** *c2=0 c1 ≠0 d2 ≠0 c1 >0 ⟷ d2 > 0*
proof −
have (*LIM x at-left 1*. *f x* / *g x* :> *at-top*)
proof −
have (*f* ⟶ *d2*) (*at-left 1*)
unfolding *f-def d2-def* **by** (*auto intro*!: *tendsto-eq-intros*)
moreover have (*g* ⟶ *0*) (*at-left 1*)
using ‹*c2=0*› **unfolding** *g-def c2-def* **by** (*auto intro*!: *tendsto-eq-intros*)
moreover have (*g has-sgnx sgn d2*) (*at-left 1*)
proof −
have (*g has-sgnx* − *sgn* (*c2*−*c1*)) (*at-left 1*)
unfolding *g-def*
apply (*rule has-sgnx-derivative-at-left*)
subgoal unfolding *c2-def c1-def d1-def* **by** (*auto intro*!: *derivative-eq-intros*)
subgoal using ‹*c2=0*› **unfolding** *c2-def* **by** *auto*
subgoal using ‹*c2=0*› ‹*c1≠0*› **by** *auto*
done
moreover have − *sgn* (*c2*−*c1*) = *sgn d2* **using** *that* **by** *fastforce*
ultimately show *?thesis* **by** *auto*
qed
ultimately show *?thesis*
using *filterlim-divide-at-bot-at-top-iff*[*of f d2 at-left 1 g*] ‹*d2≠0*› **by** *auto*
qed
then have *jumpF* (λ*t*. *f t*/*g t*) (*at-left 1*) = *1*/*2* **unfolding** *jumpF-def* **by** *auto*
then show *?thesis* **using** *that jump-eq* **by** *auto*
qed
moreover have *?thesis* **when** *c2=0 c1 ≠0 d2 ≠0* ¬ *c1 >0 ⟷ d2 > 0*
proof −
have (*LIM x at-left 1*. *f x* / *g x* :> *at-bot*)
proof −
have (*f* ⟶ *d2*) (*at-left 1*)
unfolding *f-def d2-def* **by** (*auto intro*!: *tendsto-eq-intros*)
moreover have (*g* ⟶ *0*) (*at-left 1*)
using ‹*c2=0*› **unfolding** *g-def c2-def* **by** (*auto intro*!: *tendsto-eq-intros*)
moreover have (*g has-sgnx* − *sgn d2*) (*at-left 1*)
proof −
have (*g has-sgnx* − *sgn* (*c2*−*c1*)) (*at-left 1*)

```
          unfolding g-def
          apply (rule has-sgnx-derivative-at-left)
            subgoal unfolding c2-def c1-def d1-def by (auto intro!: deriva-
tive-eq-intros)
          subgoal using ‹c2=0› unfolding c2-def by auto
          subgoal using ‹c2=0› ‹c1≠0› by auto
          done
        moreover have sgn (c2−c1) = sgn d2 using that by fastforce
        ultimately show ?thesis by auto
      qed
      ultimately show ?thesis
        using filterlim-divide-at-bot-at-top-iff[of f d2 at-left 1 g] ‹d2≠0› by
auto
    qed
    then have jumpF (λt. f t/g t) (at-left 1) = − 1/2 unfolding jumpF-def
by auto
    then show ?thesis using that jump-eq by auto
   qed
   ultimately show ?thesis by fast
  qed
  ultimately show ?thesis using ‹¬ ((c1>0 ∧ c2<0) ∨ (c1<0 ∧ c2>0))›
   apply (fold c1-def c2-def d1-def d2-def)
   by auto
 qed
 moreover have ?thesis when c1=0 c2=0
 proof −
  have (λt. Re (linepath a b t − z)) = (λ-. 0)
   using that unfolding linepath-def c1-def c2-def
   by (auto simp add:algebra-simps)
  then have (λt. Im (linepath a b t − z) / Re (linepath a b t − z)) = (λ-. 0)
   by (metis div-by-0)
  then have cindex-pathE (linepath a b) z = 0
   unfolding cindex-pathE-def
   by (auto intro: cindexE-constI)
  thus ?thesis using ‹¬ ((c1>0 ∧ c2<0) ∨ (c1<0 ∧ c2>0))› that
   apply (fold c1-def c2-def d1-def d2-def)
   by auto
 qed
 ultimately show ?thesis by fast
qed
moreover have ?thesis when c1c2-diff-sgn:(c1>0 ∧ c2<0) ∨ (c1<0 ∧ c2>0)
proof −
 define f where f=(λt. (Im b − Im a )∗ t + (Im a − Im z))
 define g where g=(λt. (Re b − Re a )∗ t + (Re a − Re z))
 define h where h=(λt. f t/ g t)
 define c3 where c3=Im(a)∗Re(b)+Re(z)∗Im(b)+Im(z)∗Re(a) −Im(z)∗Re(b)
− Im(b)∗Re(a) − Re(z)∗Im(a)
 define u where u = (Re z − Re a) / (Re b − Re a)
 let ?g = λt. linepath a b t − z
```

have *0<u u<1 Re b − Re a≠0* **using** *that* **unfolding** *u-def c1-def c2-def* **by** (*auto simp add*:*field-simps*)
have *Re*(*?g u*) = *0 g u=0* **unfolding** *linepath-def u-def g-def*
apply (*auto simp add*:*field-simps*)
using ‹*Re b − Re a≠0*› **by** (*auto simp add*:*field-simps*)
moreover have *u1* = *u2* **when** *Re*(*?g u1*) = *0 Re*(*?g u2*) = *0* **for** *u1 u2*
proof −
have (*u1* − *u2*) ∗ (*Re b* − *Re a*) = *Re*(*?g u1*) − *Re*(*?g u2*)
unfolding *linepath-def* **by** (*auto simp add*:*algebra-simps*)
also have ... = *0* **using** *that* **by** *auto*
finally have (*u1* − *u2*) ∗ (*Re b* − *Re a*) = *0* **.**
thus *?thesis* **using** ‹*Re b − Re a≠0*› **by** *auto*
qed
ultimately have *re-g-iff*:*Re*(*?g t*) = *0* ⟷ *t=u* **for** *t* **by** *blast*

have *cindex-pathE* (*linepath a b*) *z* = *jumpF h* (*at-right u*) − *jumpF h* (*at-left u*)
proof −
define *left* **where** *left* = {*x. jumpF h* (*at-left x*) ≠ *0* ∧ *0* < *x* ∧ *x* ≤ *1*}
define *right* **where** *right* = {*x. jumpF h* (*at-right x*) ≠ *0* ∧ *0* ≤ *x* ∧ *x* < *1*}
have *jumpF-nz*:*jumpF h* (*at-left x*) = *0 jumpF h* (*at-right x*) = *0*
when *0≤x x≤1 x≠u* **for** *x*
proof −
have *g x≠0*
using *re-g-iff* ‹*x≠u*› **unfolding** *g-def linepath-def*
by (*metis* ‹*Re b − Re a ≠ 0*› *add-diff-cancel-left′ diff-diff-eq2 diff-zero nonzero-mult-div-cancel-left u-def*)
then have *isCont h x*
unfolding *h-def f-def g-def*
by (*auto intro*!:*continuous-intros*)
then have *continuous* (*at-left x*) *h continuous* (*at-right x*) *h*
using *continuous-at-split* **by** *blast+*
then show *jumpF h* (*at-left x*) = *0 jumpF h*(*at-right x*) = *0*
using *jumpF-not-infinity* **by** *auto*
qed
have *cindex-pathE* (*linepath a b*) *z* = *sum* (λ*x. jumpF h* (*at-right x*)) *right* − *sum* (λ*x. jumpF h* (*at-left x*)) *left*
proof −
have *cindex-pathE* (*linepath a b*) *z* = *cindexE 0 1* (λ*t. Im* (*?g t*) / *Re* (*?g t*))
unfolding *cindex-pathE-def* **by** *auto*
also have ... = *cindexE 0 1 h*
proof −
have (λ*t. Im* (*?g t*) / *Re* (*?g t*)) = *h*
unfolding *h-def f-def g-def linepath-def*
by (*auto simp add*:*algebra-simps*)
then show *?thesis* **by** *auto*
qed
also have ... = *sum* (λ*x. jumpF h* (*at-right x*)) *right* − *sum* (λ*x. jumpF h*

```
(at-left x)) left
          unfolding cindexE-def left-def right-def by auto
        finally show ?thesis .
      qed
      moreover have sum (λx. jumpF h (at-right x)) right = jumpF h (at-right u)
      proof (cases jumpF h (at-right u) = 0)
        case True
        then have right = {}
          unfolding right-def using jumpF-nz by force
        then show ?thesis using True by auto
      next
        case False
        then have right = {u}
          unfolding right-def using jumpF-nz ‹0<u› ‹u<1› by fastforce
        then show ?thesis by auto
      qed
      moreover have sum (λx. jumpF h (at-left x)) left = jumpF h (at-left u)
      proof (cases jumpF h (at-left u) = 0)
        case True
        then have left = {}
          unfolding left-def
          apply safe
          apply (case-tac x=u)
          using jumpF-nz ‹0<u› ‹u<1› by auto
        then show ?thesis using True by auto
      next
        case False
        then have left = {u}
          unfolding left-def
          apply safe
            apply (case-tac x=u)
          using jumpF-nz ‹0<u› ‹u<1› by auto
        then show ?thesis by auto
      qed
      ultimately show ?thesis by auto
    qed
    moreover have jump h u = (if c3>0 then 1 else −1)
    proof −
      have Re b−Re a≠0 using c1c2-diff-sgn unfolding c1-def c2-def by auto
      have jump (λt. Im(?g t) / Re(?g t)) u = jump h u
        apply (rule arg-cong2[where f=jump])
        unfolding linepath-def h-def f-def g-def by (auto simp add:algebra-simps)
      moreover have jump (λt. Im(?g t) / Re(?g t)) u
          = (if sgn (Re b −Re a) = sgn (Im(?g u)) then 1 else − 1)
      proof (rule jump-divide-derivative)
        have path ?g using path-offset by auto
        then have continuous-on {0..1} (λt. Im(?g t))
          using continuous-on-Im path-def by blast
        then show isCont (λt. Im (?g t)) u
```

```
          unfolding path-def
          apply (elim continuous-on-interior)
          using ‹0<u› ‹u<1› by auto
      next
        show Re(?g u) = 0 Re b − Re a ≠ 0 using ‹Re(?g u) = 0› ‹Re b − Re a
≠ 0›
          by auto
        show Im(?g u) ≠ 0
        proof (rule ccontr)
          assume ¬ Im (linepath a b u − z) ≠ 0
          then have ?g u = 0 using ‹Re(?g u) = 0›
            by (simp add: complex-eq-iff)
          then have z ∈ closed-segment a b using ‹0<u› ‹u<1›
            by (auto intro:linepath-in-path)
          thus False using ‹z ∉ closed-segment a b› by simp
        qed
        show ((λt. Re (linepath a b t − z)) has-real-derivative Re b − Re a) (at u)
          unfolding linepath-def by (auto intro!:derivative-eq-intros)
      qed
      moreover have sgn (Re b − Re a) = sgn (Im(?g u)) ⟷ c3 > 0
      proof −
        have Im(?g u) = c3/(Re b−Re a)
        proof −
          define ba where ba = Re b−Re a
          have ba≠0  using ‹Re b − Re a ≠ 0› unfolding ba-def by auto
          then show ?thesis
            unfolding linepath-def u-def c3-def
            apply (fold ba-def)
            apply (auto simp add:field-simps)
            by (auto simp add:algebra-simps ba-def)
        qed
        then have sgn (Re b − Re a) = sgn (Im(?g u)) ⟷ sgn (Re b − Re a) =
sgn (c3/(Re b−Re a))
          by auto
        also have ... ⟷ c3>0
          using ‹Re b−Re a≠0›
          apply (cases 0::real c3 rule:linorder-cases)
          by (auto simp add:sgn-zero-iff)
        finally show ?thesis .
      qed
      ultimately show ?thesis by auto
    qed
    moreover have jump h u = jumpF h (at-right u) − jumpF h (at-left u)
    proof (rule jump-jumpF)
      have f u≠0
      proof (rule ccontr)
        assume ¬ f u ≠ 0
        then have z∈path-image (linepath a b)
          unfolding path-image-def
```

```
        apply (rule-tac rev-image-eqI[of u])
        using re-g-iff[of u,simplified] ‹0<u› ‹u<1›
        unfolding f-def linepath-def
        by (auto simp add:algebra-simps complex.expand)
      then show False using ‹z∉closed-segment a b› by simp
    qed
    then show isCont (inverse ∘ h) u
      unfolding h-def comp-def f-def g-def
      by (auto intro!: continuous-intros)
    define hs where hs = sgn ((f u) / (c2 −c1))
    show (h has-sgnx −hs) (at-left u) (h has-sgnx hs) (at-right u)
    proof −
      have ff:(f has-sgnx sgn (f u)) (at-left u) (f has-sgnx sgn (f u)) (at-right u)
      proof −
        have (f ⟶ f u) (at u)
          unfolding f-def by (auto intro!:tendsto-intros)
        then have  (f has-sgnx sgn (f u)) (at u)
          using tendsto-nonzero-has-sgnx[of f, OF - ‹f u≠0›] by auto
        then show (f has-sgnx sgn (f u)) (at-left u) (f has-sgnx sgn (f u)) (at-right
u)
          using has-sgnx-split by blast+
      qed
      have gg:(g has-sgnx − sgn (c2 − c1)) (at-left u) (g has-sgnx sgn (c2 − c1))
(at-right u)
      proof −
        have (g has-real-derivative c2 − c1) (at u) unfolding g-def c1-def c2-def
          by (auto intro!:derivative-eq-intros)
        moreover have c2 − c1 ≠ 0 using that by auto
        ultimately show (g has-sgnx sgn (c2 − c1)) (at-right u)
            (g has-sgnx − sgn (c2 − c1)) (at-left u)
          using has-sgnx-derivative-at-right[of g c2−c1 u]
              has-sgnx-derivative-at-left[of g c2−c1 u] ‹g u=0›
          by auto
      qed
      show (h has-sgnx − hs) (at-left u)
        using has-sgnx-divide[OF ff(1) gg(1)] unfolding h-def hs-def
        by auto
      show (h has-sgnx hs) (at-right u)
        using has-sgnx-divide[OF ff(2) gg(2)] unfolding h-def hs-def
        by auto
    qed
    show hs≠0 −hs≠0
      unfolding hs-def using ‹f u≠0› that by (auto simp add:sgn-if)
  qed
  ultimately show ?thesis using that
    apply (fold c1-def c2-def c3-def)
    by auto
qed
ultimately show ?thesis by fast
```

```
qed

lemma cindex-path-linepath:
  assumes z∉path-image (linepath a b)
  shows cindex-path (linepath a b) z = (
    let c1=Re(a)−Re(z) ; c2=Re(b)−Re(z) ;
      c3 = Im(a)∗Re(b)+Re(z)∗Im(b)+Im(z)∗Re(a) −Im(z)∗Re(b) − Im(b)∗Re(a)
− Re(z)∗Im(a)
    in if (c1>0 ∧ c2<0) ∨ (c1<0 ∧ c2>0) then (if c3>0 then 1 else −1) else 0)

proof −
  define c1 c2 where c1=Re(a)−Re(z) and c2=Re(b)−Re(z)
  let ?g = linepath a b
  have ?thesis when ¬ ((c1>0 ∧ c2<0) ∨ (c1<0 ∧ c2>0))
  proof −
    have Re a= Re z ∧ Re b=Re z
      when 0<t t<1 and asm:(1−t)∗Re a + t ∗ Re b = Re z for t
      unfolding c1-def c2-def using that
    proof −
      have ?thesis when c1≤0 c1≥0
      proof −
        have Re a=Re z using that unfolding c1-def by auto
        then show ?thesis using ‹0<t› ‹t<1› asm
          apply (cases Re b Re z rule:linorder-cases)
            apply (auto simp add:field-simps)
          done
      qed
      moreover have ?thesis when c1≤0 c2≤0
      proof −
        have False when c1<0
        proof −
          have (1 − t) ∗ Re a < (1 − t) ∗ Re z
            using ‹t<1› ‹c1<0› unfolding c1-def by auto
         moreover have t ∗ Re b ≤ t∗ Re z using ‹t>0› ‹c2≤0› unfolding c2-def
by auto
          ultimately have (1 − t) ∗ Re a + t ∗ Re b < (1 − t) ∗ Re z + t ∗ Re z
            by auto
          thus False using asm by (auto simp add:algebra-simps)
        qed
        moreover have False when c2<0
        proof −
          have (1 − t) ∗ Re a ≤ (1 − t) ∗ Re z
            using ‹t<1› ‹c1≤0› unfolding c1-def by auto
         moreover have t ∗ Re b < t∗ Re z using ‹t>0› ‹c2<0› unfolding c2-def
by auto
          ultimately have (1 − t) ∗ Re a + t ∗ Re b < (1 − t) ∗ Re z + t ∗ Re z
            by auto
          thus False using asm by (auto simp add:algebra-simps)
        qed
```

```
    ultimately show ?thesis using that unfolding c1-def c2-def by argo
  qed
  moreover have ?thesis when c2≤0 c2≥0
  proof -
    have Re b=Re z using that unfolding c2-def by auto
      then have (1 - t) * Re a = (1-t)*Re z using asm by (auto simp
add:field-simps)
    then have Re a= Re z using ‹t<1› by auto
    then show ?thesis using ‹Re b=Re z› by auto
  qed
  moreover have ?thesis when c1≥0 c2≥0
  proof -
    have False when c1>0
    proof -
      have (1 - t) * Re a > (1 - t) * Re z
        using ‹t<1› ‹c1>0› unfolding c1-def by auto
     moreover have t * Re b ≥ t* Re z using ‹t>0› ‹c2≥0› unfolding c2-def
by auto
      ultimately have (1 - t) * Re a + t * Re b > (1 - t) * Re z + t * Re z
        by auto
      thus False using asm by (auto simp add:algebra-simps)
    qed
    moreover have False when c2>0
    proof -
      have (1 - t) * Re a ≥ (1 - t) * Re z
        using ‹t<1› ‹c1≥0› unfolding c1-def by auto
     moreover have t * Re b > t* Re z using ‹t>0› ‹c2>0› unfolding c2-def
by auto
      ultimately have (1 - t) * Re a + t * Re b > (1 - t) * Re z + t * Re z
        by auto
      thus False using asm by (auto simp add:algebra-simps)
    qed
    ultimately show ?thesis using that unfolding c1-def c2-def by argo
  qed
  moreover have c1≤0 ∨ c2≥0 c1≥0 ∨ c2≤0 using ‹¬ ((c1>0 ∧ c2<0) ∨
(c1<0 ∧ c2>0))› by auto
  ultimately show ?thesis by fast
qed
then have (∀ t. 0<t ∧ t<1 ⟶ Re(linepath a b t - z) ≠ 0) ∨ (Re a= Re z ∧
Re b=Re z)
  using that unfolding linepath-def by auto
moreover have ?thesis when asm:∀ t. 0<t ∧ t<1 ⟶ Re(linepath a b t - z)
≠ 0
proof -
  have jump (λt. Im (linepath a b t - z) / Re (linepath a b t - z)) t = 0
    when 0<t t<1 for t
    apply (rule jump-Im-divide-Re-0[of λt. linepath a b t - z,
          OF - asm[rule-format]])
    by (auto simp add:path-offset that)
```

```
      then have cindex-path (linepath a b) z = 0
        unfolding cindex-path-def cindex-def by auto
      thus ?thesis using ‹¬ ((c1>0 ∧ c2<0) ∨ (c1<0 ∧ c2>0))›
        apply (fold c1-def c2-def)
        by auto
    qed
    moreover have ?thesis when Re a= Re z Re b=Re z
    proof –
      have (λt. Re (linepath a b t − z)) = (λ-. 0)
        unfolding linepath-def using ‹Re a= Re z› ‹Re b=Re z›
        by (auto simp add:algebra-simps)
      then have (λt. Im (linepath a b t − z) / Re (linepath a b t − z)) = (λ-. 0)
        by (metis div-by-0)
      then have jump (λt. Im (linepath a b t − z) / Re (linepath a b t − z)) t =
0 for t
        using jump-const by auto
      then have cindex-path (linepath a b) z = 0
        unfolding cindex-path-def cindex-def by auto
      thus ?thesis using ‹¬ ((c1>0 ∧ c2<0) ∨ (c1<0 ∧ c2>0))›
        apply (fold c1-def c2-def)
        by auto
    qed
    ultimately show ?thesis by auto
  qed
  moreover have ?thesis when c1c2-diff-sgn:(c1>0 ∧ c2<0) ∨ (c1<0 ∧ c2>0)
  proof –
    define c3 where c3=Im(a)*Re(b)+Re(z)*Im(b)+Im(z)*Re(a) −Im(z)*Re(b)
− Im(b)*Re(a) − Re(z)*Im(a)
    define u where u = (Re z − Re a) / (Re b − Re a)
    let ?g = λt. linepath a b t − z
    have 0<u u<1 Re b − Re a≠0 using that unfolding u-def c1-def c2-def by
(auto simp add:field-simps)
    have Re(?g u) = 0 unfolding linepath-def u-def
      apply (auto simp add:field-simps)
      using ‹Re b − Re a≠0› by (auto simp add:field-simps)
    moreover have u1 = u2 when Re(?g u1) = 0 Re(?g u2) = 0 for u1 u2
    proof –
      have (u1 − u2) * (Re b − Re a) = Re(?g u1) − Re(?g u2)
        unfolding linepath-def by (auto simp add:algebra-simps)
      also have ... = 0 using that by auto
      finally have (u1 − u2) * (Re b − Re a) = 0 .
      thus ?thesis using ‹Re b − Re a≠0› by auto
    qed
    ultimately have re-g-iff:Re(?g t) = 0 ⟷ t=u for t by blast
    have cindex-path (linepath a b) z = jump (λt. Im (?g t)/Re(?g t)) u
    proof –
      define f where f=(λt. Im (linepath a b t − z) / Re (linepath a b t − z))
      have jump f t =0 when t≠u 0<t t<1 for t
        unfolding f-def
```

apply (*rule jump-Im-divide-Re-0*)
using *that re-g-iff* **by** (*auto simp add*: *path-offset*)
then have {*x*. *jump f x* ≠ *0* ∧ *0* < *x* ∧ *x* < *1*} = (*if jump f u*=*0 then* {} *else* {*u*})
using ‹*0*<*u*› ‹*u*<*1*›
apply *auto*
by *fastforce*
then show *?thesis*
unfolding *cindex-path-def cindex-def*
apply (*fold f-def*)
by *auto*
qed
moreover have *jump* (λ*t*. *Im* (*?g t*)/*Re*(*?g t*)) *u* = (*if c3*>*0 then 1 else* −*1*)
proof −
have *Re b*−*Re a*≠*0* **using** *c1c2-diff-sgn* **unfolding** *c1-def c2-def* **by** *auto*
have *jump* (λ*t*. *Im*(*?g t*) / *Re*(*?g t*)) *u*
= (*if sgn* (*Re b* −*Re a*) = *sgn* (*Im*(*?g u*)) *then 1 else* − *1*)
proof (*rule jump-divide-derivative*)
have *path ?g* **using** *path-offset* **by** *auto*
then have *continuous-on* {*0*..*1*} (λ*t*. *Im*(*?g t*))
using *continuous-on-Im path-def* **by** *blast*
then show *isCont* (λ*t*. *Im* (*?g t*)) *u*
unfolding *path-def*
apply (*elim continuous-on-interior*)
using ‹*0*<*u*› ‹*u*<*1*› **by** *auto*
next
show *Re*(*?g u*) = *0 Re b* − *Re a* ≠ *0* **using** ‹*Re*(*?g u*) = *0*› ‹*Re b* − *Re a* ≠ *0*›
by *auto*
show *Im*(*?g u*) ≠ *0*
proof (*rule ccontr*)
assume ¬ *Im* (*linepath a b u* − *z*) ≠ *0*
then have *?g u* = *0* **using** ‹*Re*(*?g u*) = *0*›
by (*simp add*: *complex-eq-iff*)
thus *False* **using** *assms* ‹*0*<*u*› ‹*u*<*1*› **unfolding** *path-image-def* **by** *fastforce*
qed
show ((λ*t*. *Re* (*linepath a b t* − *z*)) *has-real-derivative Re b* − *Re a*) (*at u*)
unfolding *linepath-def* **by** (*auto intro*!:*derivative-eq-intros*)
qed
moreover have *sgn* (*Re b* − *Re a*) = *sgn* (*Im*(*?g u*)) ⟷ *c3* > *0*
proof −
have *Im*(*?g u*) = *c3*/(*Re b*−*Re a*)
proof −
define *ba* **where** *ba* = *Re b*−*Re a*
have *ba*≠*0* **using** ‹*Re b* − *Re a* ≠ *0*› **unfolding** *ba-def* **by** *auto*
then show *?thesis*
unfolding *linepath-def u-def c3-def*
apply (*fold ba-def*)

```
        apply (auto simp add:field-simps)
        by (auto simp add:algebra-simps ba-def)
      qed
      then have sgn (Re b − Re a) = sgn (Im(?g u)) ⟷ sgn (Re b − Re a) =
sgn (c3/(Re b−Re a))
        by auto
      also have ... ⟷ c3>0
        using ‹Re b−Re a≠0›
        apply (cases 0::real c3 rule:linorder-cases)
        by (auto simp add:sgn-zero-iff)
      finally show ?thesis .
    qed
    ultimately show ?thesis by auto
  qed
  ultimately show ?thesis using c1c2-diff-sgn
    apply (fold c1-def c2-def c3-def)
    by auto
 qed
 ultimately show ?thesis by blast
qed

lemma cindex-pathE-part-circlepath:
 assumes cmod (z−z0) ≠r and r>0 0≤st st<tt tt≤2∗pi
 shows cindex-pathE (part-circlepath z r st tt) z0 = (
  if |Re z − Re z0| < r then
   (let
     ϑ = arccos ((Re z0 − Re z)/r);
     β = 2∗pi − ϑ
    in
     jumpF-pathstart (part-circlepath z r st tt) z0
     +
     (if st<ϑ ∧ ϑ<tt then if r ∗ sin ϑ + Im z > Im z0 then −1 else 1 else 0)
     +
     (if st<β ∧ β < tt then if r ∗ sin β + Im z > Im z0 then 1 else −1 else 0)
     −
     jumpF-pathfinish (part-circlepath z r st tt) z0
   )
  else
   if |Re z − Re z0| = r then
    jumpF-pathstart (part-circlepath z r st tt) z0
    − jumpF-pathfinish (part-circlepath z r st tt) z0
   else 0
  )
proof −
 define f where f=(λi. r ∗ sin i + Im z − Im z0)
 define g where g=(λi. r ∗ cos i + Re z − Re z0)
 define h where h=(λt. f t / g t)
 have index-eq:cindex-pathE (part-circlepath z r st tt) z0 = cindexE st tt h
 proof −
```

have *cindex-pathE* (*part-circlepath z r st tt*) *z0*
= *cindexE 0 1* ((λ*i*. *f i*/*g i*) *o* (*linepath st tt*))
unfolding *cindex-pathE-def part-circlepath-def exp-Euler f-def g-def comp-def*
by (*simp add*:*cos-of-real sin-of-real algebra-simps*)
also have ... = *cindexE st tt* (λ*i*. *f i*/*g i*)
unfolding *linepath-def* **using** *cindexE-linear-comp*[*of tt*−*st 0 1 - st*] ‹*st*<*tt*›
by (*simp add*:*algebra-simps*)
also have ... = *cindexE st tt h* **unfolding** *h-def* **by** *simp*
finally show *?thesis* **.**
qed
have *jstart-eq*:*jumpF-pathstart* (*part-circlepath z r st tt*) *z0* = *jumpF h* (*at-right st*)
proof −
have *jumpF-pathstart* (*part-circlepath z r st tt*) *z0*
= *jumpF* ((λ*i*. *f i*/*g i*) *o* (*linepath st tt*)) (*at-right 0*)
unfolding *jumpF-pathstart-def part-circlepath-def exp-Euler f-def g-def comp-def*
by (*simp add*:*cos-of-real sin-of-real algebra-simps*)
also have ... = *jumpF* (λ*i*. *f i*/*g i*) (*at-right st*)
unfolding *linepath-def* **using** *jumpF-linear-comp*(*2*)[*of tt*−*st - st 0*] ‹*st*<*tt*›
by (*simp add*:*algebra-simps*)
also have ... = *jumpF h* (*at-right st*) **unfolding** *h-def* **by** *simp*
finally show *?thesis* **.**
qed
have *jfinish-eq*:*jumpF-pathfinish* (*part-circlepath z r st tt*) *z0* = *jumpF h* (*at-left tt*)
proof −
have *jumpF-pathfinish* (*part-circlepath z r st tt*) *z0*
= *jumpF* ((λ*i*. *f i*/*g i*) *o* (*linepath st tt*)) (*at-left 1*)
unfolding *jumpF-pathfinish-def part-circlepath-def exp-Euler f-def g-def comp-def*
by (*simp add*:*cos-of-real sin-of-real algebra-simps*)
also have ... = *jumpF* (λ*i*. *f i*/*g i*) (*at-left tt*)
unfolding *linepath-def* **using** *jumpF-linear-comp*(*1*)[*of tt*−*st - st 1*] ‹*st*<*tt*›
by (*simp add*:*algebra-simps*)
also have ... = *jumpF h* (*at-left tt*) **unfolding** *h-def* **by** *simp*
finally show *?thesis* **.**
qed
have *finite-jFs*:*finite-jumpFs h st tt*
proof −
note *finite-ReZ-segments-imp-jumpFs*[*OF finite-ReZ-segments-part-circlepath*
,*of z r st tt z0*,*simplified*]
then have *finite-jumpFs* ((λ*i*. *f i*/*g i*) *o* (*linepath st tt*)) *0 1*
unfolding *h-def f-def g-def part-circlepath-def exp-Euler comp-def*
by (*simp add*:*cos-of-real sin-of-real algebra-simps*)
then have *finite-jumpFs* (λ*i*. *f i*/*g i*) *st tt*
unfolding *linepath-def* **using** *finite-jumpFs-linear-pos*[*of tt*−*st - st 0 1*] ‹*st*<*tt*›
by (*simp add*:*algebra-simps*)

then show *?thesis* **unfolding** *h-def* **by** *auto*
qed
have *g-imp-f*:$g\ i = 0 \Longrightarrow f\ i \neq 0$ **for** *i*
proof (*rule ccontr*)
assume $g\ i = 0$ $\neg f\ i \neq 0$
then have $r * sin\ i = Im\ (z0 - z)$ $r * cos\ i = Re\ (z0 - z)$
unfolding *f-def g-def* **by** *auto*
then have $(r * sin\ i)\ \hat{}2 + (r * cos\ i)\hat{}2 = Im\ (z0 - z)\ \hat{}\ 2 + Re\ (z0 - z)$
$\hat{}2$
by *auto*
then have $r\hat{}2 * (sin\ i\ \hat{}2 + cos\ i\hat{}2) = Im\ (z0 - z)\ \hat{}\ 2 + Re\ (z0 - z)\ \hat{}2$
by (*auto simp only*:*algebra-simps power-mult-distrib*)
then have $r\hat{}2 = cmod\ (z0{-}z)\ \hat{}\ 2$
unfolding *cmod-def* **by** *auto*
then have $r = cmod\ (z0{-}z)$
using ‹$r{>}0$› *power2-eq-imp-eq* **by** *fastforce*
then show *False* **using** ‹$cmod\ (z{-}z0) \neq r$› **using** *norm-minus-commute* **by**
blast
qed
have *?thesis* **when** $|Re\ z - Re\ z0| > r$
proof −
have $jumpF\ h\ (at\text{-}right\ x) = 0$ $jumpF\ h\ (at\text{-}left\ x) = 0$ **for** *x*
proof −
have $g\ x \neq 0$
proof (*rule ccontr*)
assume $\neg\ g\ x \neq 0$
then have $cos\ x = (Re\ z0 - Re\ z)\ /\ r$ **unfolding** *g-def* **using** ‹$r{>}0$›
by (*auto simp add*:*field-simps*)
then have $|(Re\ z0 - Re\ z)/r| \leq 1$
by (*metis abs-cos-le-one*)
then have $|Re\ z0 - Re\ z| \leq r$
using ‹$r{>}0$› **by** (*auto simp add*:*field-simps*)
then show *False* **using** *that* **by** *auto*
qed
then have *isCont h x*
unfolding *h-def f-def g-def* **by** (*auto intro*:*continuous-intros*)
then show $jumpF\ h\ (at\text{-}right\ x) = 0$ $jumpF\ h\ (at\text{-}left\ x) = 0$
using *jumpF-not-infinity* **unfolding** *continuous-at-split* **by** *auto*
qed
then have $cindexE\ st\ tt\ h = 0$ **unfolding** *cindexE-def* **by** *auto*
then show *?thesis* **using** *index-eq that* **by** *auto*
qed
moreover have *?thesis* **when** $|Re\ z - Re\ z0| = r$
proof −
define *R* **where** $R{=}(\lambda S.\{x.\ jumpF\ h\ (at\text{-}right\ x) \neq 0 \wedge x{\in}S\})$
define *L* **where** $L{=}(\lambda S.\{x.\ jumpF\ h\ (at\text{-}left\ x) \neq 0 \wedge x{\in}S\})$
define *right* **where**
$right = (\lambda S.\ (\sum x{\in}R\ S.\ jumpF\ h\ (at\text{-}right\ x)))$
define *left* **where**

left = (λ*S*. (∑*x*∈*L S*. *jumpF h* (*at-left x*)))
have *cindex-pathE* (*part-circlepath z r st tt*) *z0* = *cindexE st tt h*
using *index-eq* **by** *simp*
also have ... = *right* {*st*..<*tt*} − *left* {*st*<..*tt*}
unfolding *cindexE-def right-def left-def R-def L-def* **by** *auto*
also have ... = *jumpF h* (*at-right st*) + *right* {*st*<..<*tt*} − *left* {*st*<..<*tt*} − *jumpF h* (*at-left tt*)
proof −
have *right* {*st*..<*tt*} = *jumpF h* (*at-right st*) + *right* {*st*<..<*tt*}
proof (*cases jumpF h* (*at-right st*) =*0*)
case *True*
then have *R* {*st*..<*tt*} = *R* {*st*<..<*tt*}
unfolding *R-def* **using** *less-eq-real-def* **by** *auto*
then have *right* {*st*..<*tt*} = *right* {*st*<..<*tt*}
unfolding *right-def* **by** *auto*
then show *?thesis* **using** *True* **by** *auto*
next
case *False*
have *finite* (*R* {*st*..<*tt*})
using *finite-jFs* **unfolding** *R-def finite-jumpFs-def*
by (*auto elim*:*rev-finite-subset*)
moreover have *st* ∈ *R* {*st*..<*tt*} **using** *False* ‹*st*<*tt*› **unfolding** *R-def* **by** *auto*
moreover have *R* {*st*..<*tt*} − {*st*} = *R* {*st*<..<*tt*} **unfolding** *R-def* **by** *auto*
ultimately show *right* {*st*..<*tt*}= *jumpF h* (*at-right st*)
+ *right* {*st*<..<*tt*}
using *sum.remove*[*of R* {*st*..<*tt*} *st* λ*x*. *jumpF h* (*at-right x*)]
unfolding *right-def* **by** *simp*
qed
moreover have *left* {*st*<..*tt*} = *jumpF h* (*at-left tt*) + *left* {*st*<..<*tt*}
proof (*cases jumpF h* (*at-left tt*) =*0*)
case *True*
then have *L* {*st*<..*tt*} = *L* {*st*<..<*tt*}
unfolding *L-def* **using** *less-eq-real-def* **by** *auto*
then have *left* {*st*<..*tt*} = *left* {*st*<..<*tt*}
unfolding *left-def* **by** *auto*
then show *?thesis* **using** *True* **by** *auto*
next
case *False*
have *finite* (*L* {*st*<..*tt*})
using *finite-jFs* **unfolding** *L-def finite-jumpFs-def*
by (*auto elim*:*rev-finite-subset*)
moreover have *tt* ∈ *L* {*st*<..*tt*} **using** *False* ‹*st*<*tt*› **unfolding** *L-def* **by** *auto*
moreover have *L* {*st*<..*tt*} − {*tt*} = *L* {*st*<..<*tt*} **unfolding** *L-def* **by** *auto*
ultimately show *left* {*st*<..*tt*}= *jumpF h* (*at-left tt*) + *left* {*st*<..<*tt*}
using *sum.remove*[*of L* {*st*<..*tt*} *tt* λ*x*. *jumpF h* (*at-left x*)]

```
      unfoldling left-def by simp
    qed
    ultimately show ?thesis by simp
  qed
  also have ... =  jumpF h (at-right st) − jumpF h (at-left tt)
  proof −
    define S where S={x. (jumpF h (at-left x) ≠ 0 ∨ jumpF h (at-right x) ≠
0) ∧ st < x ∧ x < tt}
    have right {st<..<tt} = sum (λx. jumpF h (at-right x)) S
      unfolding right-def S-def R-def
      apply (rule sum.mono-neutral-left)
    subgoal using finite-jFs unfolding finite-jumpFs-def by (auto elim:rev-finite-subset)
      subgoal by auto
      subgoal by auto
      done
    moreover have left {st<..<tt} = sum (λx. jumpF h (at-left x)) S
      unfolding left-def S-def L-def
      apply (rule sum.mono-neutral-left)
    subgoal using finite-jFs unfolding finite-jumpFs-def by (auto elim:rev-finite-subset)
      subgoal by auto
      subgoal by auto
      done
    ultimately have right {st<..<tt} − left {st<..<tt}
        = sum (λx. jumpF h (at-right x) − jumpF h (at-left x)) S
      by (simp add: sum-subtractf)
    also have ... = 0
    proof −
      have jumpF h (at-right i) − jumpF h (at-left i) = 0 when g i=0 for i
      proof −
        have (LIM x at i. f x / g x :> at-bot) ∨ (LIM x at i. f x / g x :> at-top)
        proof −
          have *: f −i→ f i g −i→ 0 f i ≠ 0
            using g-imp-f[OF ‹g i=0›] ‹g i=0› unfolding f-def g-def
            by (auto intro!:tendsto-eq-intros)
          have ?thesis when Re z > Re z0
          proof −
           have g-alt:g = (λt. r ∗ cos t + r) unfolding g-def using ‹|Re z − Re
z0| = r› that by auto
            have (g has-sgnx 1) (at i)
            proof −
              have sgn (g t) = 1 when t ≠ i  dist t i < pi for t
              proof −
                have cos i = − 1 using ‹g i =0› ‹r>0› unfolding g-alt
                by (metis add.inverse-inverse less-numeral-extra(3) mult-cancel-left

                      mult-minus1-right real-add-minus-iff)
                then obtain k::int where k-def:i = (2 ∗ k + 1) ∗ pi
                  using cos-eq-minus1[of i] by auto
                show ?thesis
```

proof (*rule ccontr*)
assume *sgn* (*g t*) ≠ *1*
then have *cos t* + *1*≤*0* **using** ‹*r*>*0*› **unfolding** *g-alt*
by (*metis* (*no-types*, *opaque-lifting*) *add-le-same-cancel1*
add-minus-cancel
mult-le-cancel-left1 mult-le-cancel-right1 mult-minus-right
mult-zero-left
sgn-pos zero-le-one)
then have *cos t* = −*1*
by (*metis add.commute cos-ge-minus-one le-less not-less*
real-add-le-0-iff)
then obtain *k*′::*int* **where** *k*′*-def*:*t* = (*2* ∗ *k*′ + *1*) ∗ *pi*
using *cos-eq-minus1*[*of t*] **by** *auto*
then have *t* − *i* = *2* ∗ *pi*∗(*k*′ − *k*)
using *k-def* **by** (*auto simp add*:*algebra-simps*)
then have *2* ∗ *pi* ∗ | (*k*′ − *k*)| < *pi*
using ‹*dist t i* < *pi*› **by** (*simp add*:*dist-norm abs-mult*)
from *divide-strict-right-mono*[*OF this*, *of 2*∗*pi*,*simplified*] **have** |*k*′
− *k* | < *1*/*2*
by *auto*
then have *k*=*k*′ **by** *linarith*
then have *t*=*i* **using** *k-def k*′*-def* **by** *auto*
then show *False* **using** ‹*t*≠*i*› **by** *auto*
qed
qed
then show *?thesis* **unfolding** *has-sgnx-def eventually-at*
apply(*intro exI*[**where** *x*=*pi*])
by *auto*
qed
then show *?thesis* **using** ∗ *filterlim-divide-at-bot-at-top-iff*[*of f f i at i*
g]
by (*simp add*: *sgn-if*)
qed
moreover have *?thesis* **when** *Re z* < *Re z0*
proof −
have *g-alt*:*g* = (λ*t*. *r* ∗ *cos t* − *r*) **unfolding** *g-def* **using** ‹|*Re z* −
Re z0| = *r*› *that* **by** *auto*
have (*g has-sgnx* − *1*) (*at i*)
proof −
have *sgn* (*g t*) = − *1* **when** *t* ≠ *i dist t i* < *pi* **for** *t*
proof −
have *cos i* = *1* **using** ‹*g i* =*0*› ‹*r*>*0*› **unfolding** *g-alt* **by** *simp*
then obtain *k*::*int* **where** *k-def*:*i* = (*2* ∗ *k* ∗ *pi*)
using *cos-one-2pi-int*[*of i*] **by** *auto*
show *?thesis*
proof (*rule ccontr*)
assume *sgn* (*g t*) ≠ − *1*
then have *cos t* − *1*≥*0* **using** ‹*r*>*0*› **unfolding** *g-alt*
using *mult-le-cancel-left1* **by** *fastforce*

```
                then have cos t = 1
                  by (meson cos-le-one diff-ge-0-iff-ge le-less not-less)
                then obtain k'::int where k'-def:t = 2 * k'* pi
                  using cos-one-2pi-int[of t] by auto
                then have t - i = 2 * pi*(k' - k)
                  using k-def by (auto simp add:algebra-simps)
                then have 2 * pi * | (k' - k)| < pi
                  using ‹dist t i < pi› by (simp add:dist-norm abs-mult)
                 from divide-strict-right-mono[OF this, of 2*pi,simplified] have
|k' - k | < 1/2
                  by auto
                then have k=k' by linarith
                then have t=i using k-def k'-def by auto
                then show False using ‹t≠i› by auto
              qed
            qed
            then show ?thesis unfolding has-sgnx-def eventually-at
              apply(intro exI[where x=pi])
              by auto
          qed
          then show ?thesis using * filterlim-divide-at-bot-at-top-iff[of f f i at
i g]
            by (simp add: sgn-if)
      qed
       moreover have Re z≠ Re z0 using ‹|Re z - Re z0| = r› ‹r>0› by
fastforce
      ultimately show ?thesis by fastforce
    qed
    moreover have ?thesis when (LIM x at i. f x / g x :> at-bot)
    proof -
      have jumpF h (at-right i) = - 1/2 jumpF h (at-left i) = -1/2
        using that unfolding jumpF-def h-def filterlim-at-split by auto
      then show ?thesis by auto
    qed
    moreover have ?thesis when (LIM x at i. f x / g x :> at-top)
    proof -
      have jumpF h (at-right i) =  1/2 jumpF h (at-left i) = 1/2
        using that unfolding jumpF-def h-def filterlim-at-split by auto
      then show ?thesis by auto
    qed
    ultimately show ?thesis by auto
  qed
   moreover have jumpF h (at-right i) - jumpF h (at-left i) = 0 when g
i≠0 for i
  proof -
    have isCont h i using that unfolding h-def f-def g-def
      by (auto intro!:continuous-intros)
    then have jumpF h (at-right i) = 0 jumpF h (at-left i) = 0
      using jumpF-not-infinity unfolding continuous-at-split by auto
```

```
      then show ?thesis by auto
    qed
    ultimately show ?thesis by (intro sum.neutral,auto)
  qed
  finally show ?thesis by simp
qed
also have ... = jumpF-pathstart (part-circlepath z r st tt) z0
    − jumpF-pathfinish (part-circlepath z r st tt) z0
  using jstart-eq jfinish-eq by auto
finally have cindex-pathE (part-circlepath z r st tt) z0 =
    jumpF-pathstart (part-circlepath z r st tt) z0
    − jumpF-pathfinish (part-circlepath z r st tt) z0
  .
then show ?thesis using that by auto
qed
moreover have ?thesis when |Re z − Re z0| < r
proof −
  define zr where zr= (Re z0 − Re z)/r
  define ϑ where ϑ = arccos zr
  define β where β = 2∗pi − ϑ
  have 0<ϑ ϑ<pi
  proof −
    have − 1 < zr zr < 1
      using that ‹r>0› unfolding zr-def by (auto simp add:field-simps)
    from arccos-lt-bounded[OF this] show 0<ϑ ϑ<pi
      unfolding ϑ-def by auto
  qed
  have g ϑ = 0 g β = 0
  proof −
    have |zr|≤1 using that unfolding zr-def by auto
    then have cos ϑ = zr cos β = cos ϑ
      unfolding ϑ-def[folded zr-def] β-def by auto
    then show g ϑ = 0 g β = 0 unfolding zr-def g-def using ‹r>0› by auto
  qed
  have g-sgnx-ϑ:(g has-sgnx 1) (at-left ϑ) (g has-sgnx −1) (at-right ϑ)
  proof −
    have (g has-real-derivative − r ∗ sin ϑ) (at ϑ)
      unfolding g-def by (auto intro!:derivative-eq-intros)
    moreover have − r ∗ sin ϑ <0
      using sin-gt-zero[OF ‹0<ϑ› ‹ϑ<pi›] ‹r>0› by auto
    ultimately show (g has-sgnx 1) (at-left ϑ) (g has-sgnx −1) (at-right ϑ)
      using has-sgnx-derivative-at-left[of g − r ∗ sin ϑ, OF - ‹g ϑ=0›]
            has-sgnx-derivative-at-right[of g − r ∗ sin ϑ, OF - ‹g ϑ=0›]
      by force+
  qed
  have g-sgnx-β:(g has-sgnx −1) (at-left β) (g has-sgnx 1) (at-right β)
  proof −
    have (g has-real-derivative − r ∗ sin β) (at β)
      unfolding g-def by (auto intro!:derivative-eq-intros)
```

```
      moreover have pi<β β<2*pi unfolding β-def using ‹0<ϑ› ‹ϑ<pi› by
auto
      from sin-lt-zero[OF this] ‹r>0› have − r ∗ sin β >0 by (simp add:
mult-pos-neg)
    ultimately show (g has-sgnx −1) (at-left β) (g has-sgnx 1) (at-right β)
      using has-sgnx-derivative-at-left[of g − r ∗ sin β, OF - ‹g β=0›]
          has-sgnx-derivative-at-right[of g − r ∗ sin β, OF - ‹g β=0›]
      by force+
  qed
  have f-tendsto: (f ⟶ f i) (at-left i) (f ⟶ f i) (at-right i)
   and g-tendsto: (g ⟶ g i) (at-left i) (g ⟶ g i) (at-right i) for i
  proof −
    have (f ⟶ f i) (at i)
      unfolding f-def by (auto intro!:tendsto-eq-intros)
    then show (f ⟶ f i) (at-left i) (f ⟶ f i) (at-right i)
      by (auto simp add: filterlim-at-split)
  next
    have (g ⟶ g i) (at i)
      unfolding g-def by (auto intro!:tendsto-eq-intros)
    then show (g ⟶ g i) (at-left i) (g ⟶ g i) (at-right i)
      by (auto simp add: filterlim-at-split)
  qed

  define ϑ-if::real where ϑ-if = (if r ∗ sin ϑ + Im z > Im z0 then −1 else 1)
  define β-if::real where β-if = (if r ∗ sin β + Im z > Im z0 then 1 else −1)
  have jump (λi. f i/g i) ϑ = ϑ-if
  proof −
    have ?thesis when r ∗ sin ϑ + Im z > Im z0
    proof −
      have f ϑ > 0 using that unfolding f-def by auto
      have (LIM x (at-left ϑ). f x / g x :> at-top)
        apply (subst filterlim-divide-at-bot-at-top-iff[of f f ϑ - g])
        using ‹f ϑ > 0› ‹g ϑ =0› f-tendsto g-tendsto[of ϑ] g-sgnx-ϑ by auto
      moreover then have ¬ (LIM x (at-left ϑ). f x / g x :> at-bot) by auto
      moreover have (LIM x (at-right ϑ). f x / g x :> at-bot)
        apply (subst filterlim-divide-at-bot-at-top-iff[of f f ϑ - g])
        using ‹f ϑ > 0› ‹g ϑ =0› f-tendsto g-tendsto[of ϑ] g-sgnx-ϑ by auto
      ultimately show ?thesis using that unfolding jump-def ϑ-if-def by auto
    qed
    moreover have ?thesis when r ∗ sin ϑ + Im z < Im z0
    proof −
      have f ϑ < 0 using that unfolding f-def by auto
      have (LIM x (at-left ϑ). f x / g x :> at-bot)
        apply (subst filterlim-divide-at-bot-at-top-iff[of f f ϑ - g])
        using ‹f ϑ < 0› ‹g ϑ =0› f-tendsto g-tendsto[of ϑ] g-sgnx-ϑ by auto
      moreover have (LIM x (at-right ϑ). f x / g x :> at-top)
        apply (subst filterlim-divide-at-bot-at-top-iff[of f f ϑ - g])
        using ‹f ϑ < 0› ‹g ϑ =0› f-tendsto g-tendsto[of ϑ] g-sgnx-ϑ by auto
      ultimately show ?thesis using that unfolding jump-def ϑ-if-def by auto
```

```
    qed
    moreover have r * sin ϑ + Im z ≠ Im z0
      using g-imp-f[OF ‹g ϑ=0›] unfolding f-def by auto
    ultimately show ?thesis by fastforce
  qed
  moreover have jump (λi. f i/g i) β = β-if
  proof –
    have ?thesis when r * sin β + Im z > Im z0
    proof –
      have f β > 0 using that unfolding f-def by auto
      have (LIM x (at-left β). f x / g x :> at-bot)
        apply (subst filterlim-divide-at-bot-at-top-iff[of f f β - g])
        using ‹f β > 0› ‹g β =0› f-tendsto g-tendsto[of β] g-sgnx-β by auto
      moreover have (LIM x (at-right β). f x / g x :> at-top)
        apply (subst filterlim-divide-at-bot-at-top-iff[of f f β - g])
        using ‹f β > 0› ‹g β =0› f-tendsto g-tendsto[of β] g-sgnx-β by auto
      ultimately show ?thesis using that unfolding jump-def β-if-def by auto
    qed
    moreover have ?thesis when r * sin β + Im z < Im z0
    proof –
      have f β < 0 using that unfolding f-def by auto
      have (LIM x (at-left β). f x / g x :> at-top)
        apply (subst filterlim-divide-at-bot-at-top-iff[of f f β - g])
        using ‹f β < 0› ‹g β =0› f-tendsto g-tendsto[of β] g-sgnx-β by auto
      moreover have (LIM x (at-right β). f x / g x :> at-bot)
        apply (subst filterlim-divide-at-bot-at-top-iff[of f f β - g])
        using ‹f β < 0› ‹g β =0› f-tendsto g-tendsto[of β] g-sgnx-β by auto
      ultimately show ?thesis using that unfolding jump-def β-if-def by auto
    qed
    moreover have r * sin β + Im z ≠ Im z0
      using g-imp-f[OF ‹g β=0›] unfolding f-def by auto
    ultimately show ?thesis by fastforce
  qed
  moreover have jump (λi. f i / g i) x ≠ 0 ⟷ x=ϑ ∨ x=β when st<x x<tt
for x
  proof
    assume x = ϑ ∨ x = β
    then show jump (λi. f i / g i) x ≠ 0
      using ‹jump (λi. f i/g i) ϑ = ϑ-if› ‹jump (λi. f i/g i) β = β-if›
      unfolding ϑ-if-def β-if-def
      by (metis add.inverse-inverse add.inverse-neutral of-int-0 one-neq-zero)
  next
    assume asm:jump (λi. f i / g i) x ≠ 0
    let ?thesis = x = ϑ ∨ x = β
    have g x=0
    proof (rule ccontr)
      assume g x ≠ 0
      then have isCont (λi. f i / g i) x
        unfolding f-def g-def by (auto intro:continuous-intros)
```

then have *jump* (λ*i*. *f i* / *g i*) *x* = *0* **using** *jump-not-infinity* **by** *simp*
then show *False* **using** *asm* **by** *auto*
qed
then have *cos x* = *zr* **unfolding** *g-def zr-def* **using** ‹*r*>*0*› **by** (*auto simp add*:*field-simps*)
have *?thesis* **when** *x*≤*pi*
proof−
have *x*≥*0* **using** ‹*st*<*x*› ‹*st*≥*0*› **by** *auto*
then have *arccos* (*cos x*) = *x* **using** *arccos-cos*[*of x*] *that* **by** *auto*
then have *x*=ϑ **unfolding** ϑ*-def* ‹*cos x*=*zr*› **by** *auto*
then show *?thesis* **by** *auto*
qed
moreover have *?thesis* **when** ¬ *x*≤*pi*
proof −
have *x*−*2*∗*pi*≤*0* −*pi*≤*x*−*2*∗*pi* **using** *that* ‹*x*<*tt*› ‹*tt*≤*2*∗*pi*› **by** *auto*
from *arccos-cos2*[*OF this*] **have** *arccos* (*cos* (*x* − *2* ∗ *pi*)) = *2*∗*pi*−*x* **by** *auto*
then have *arccos* (*cos x*) = *2*∗*pi*−*x*
by (*metis arccos cos-2pi-minus cos-ge-minus-one cos-le-one*)
then have *x*=β **unfolding** β*-def* ϑ*-def* **using** ‹*cos x* =*zr*› **by** *auto*
then show *?thesis* **by** *auto*
qed
ultimately show *?thesis* **by** *auto*
qed
then have {*x*. *jump* (λ*i*. *f i* / *g i*) *x* ≠ *0* ∧ *st* < *x* ∧ *x* < *tt*} = {ϑ,β} ∩ {*st*<..<*tt*}
by *force*
moreover have ϑ≠β **using** β*-def* ‹ϑ < *pi*› **by** *auto*
ultimately have *cindex st tt h* =
(*if st*<ϑ ∧ ϑ<*tt then* ϑ*-if else 0*)
+
(*if st*<β ∧ β < *tt then* β*-if else 0*)
unfolding *cindex-def h-def* **by** *fastforce*
moreover have *cindexE st tt h* = *jumpF h* (*at-right st*) + *cindex st tt h* − *jumpF h* (*at-left tt*)
proof (*rule cindex-eq-cindexE-divide*[*of st tt f g*,*folded h-def*])
show *st* < *tt* **using** ‹*st* < *tt*› **.**
show ∀ *x*∈{*st*..*tt*}. *g x* = *0* ⟶ *f x* ≠ *0* **using** *g-imp-f* **by** *auto*
show *continuous-on* {*st*..*tt*} *f continuous-on* {*st*..*tt*} *g*
unfolding *f-def g-def* **by** (*auto intro*!:*continuous-intros*)
next
let *?S1*={*t*. *Re* (*part-circlepath z r st tt t*−*z0*) = *0* ∧ *0* ≤ *t* ∧ *t* ≤ *1*}
let *?S2*={*t*. *Im* (*part-circlepath z r st tt t*−*z0*) = *0* ∧ *0* ≤ *t* ∧ *t* ≤ *1*}
define *G* **where** *G*={*t*. *g* (*linepath st tt t*) = *0* ∧ *0* ≤ *t* ∧ *t* ≤ *1*}
define *F* **where** *F*={*t*. *f* (*linepath st tt t*) = *0* ∧ *0* ≤ *t* ∧ *t* ≤ *1*}
define *vl* **where** *vl*=(λ*x*. (*x*−*st*)/(*tt*−*st*))
have *finite G finite F*
proof −
have *finite* {*t*. *Re* (*part-circlepath z r st tt t*−*z0*) = *0* ∧ *0* ≤ *t* ∧ *t* ≤ *1*}

finite {*t. Im* (*part-circlepath z r st tt t*−*z0*) = *0* ∧ *0* ≤ *t* ∧ *t* ≤ *1*}
using *part-circlepath-half-finite-inter*[*of st tt r Complex 1 0 z Re z0*]
part-circlepath-half-finite-inter[*of st tt r Complex 0 1 z Im z0*] ‹*st*<*tt*›
‹*r*>*0*›
by (*auto simp add*:*inner-complex-def Complex-eq-0*)
moreover have
Re (*part-circlepath z r st tt t*−*z0*) = *0* ⟷ *g* (*linepath st tt t*) = *0*
Im (*part-circlepath z r st tt t*−*z0*) = *0* ⟷ *f* (*linepath st tt t*) = *0*
for *t*
unfolding *cindex-pathE-def part-circlepath-def exp-Euler f-def g-def comp-def*
by (*auto simp add*:*cos-of-real sin-of-real algebra-simps*)
ultimately show *finite G finite F* **unfolding** *G-def F-def*
by *auto*
qed
then have *finite* (*linepath st tt ‘ F*) *finite* (*linepath st tt ‘ G*)
by *auto*
moreover have
{*x. f x* = *0* ∧ *st* ≤ *x* ∧ *x* ≤ *tt*} ⊆ *linepath st tt ‘ F*
{*x. g x* = *0* ∧ *st* ≤ *x* ∧ *x* ≤ *tt*} ⊆ *linepath st tt ‘ G*
proof −
have ∗: *linepath st tt* (*vl t*) = *t vl t*≥*0* ⟷ *t*≥*st vl t*≤*1* ⟷*t*≤*tt* **for** *t*
unfolding *linepath-def vl-def* **using** ‹*tt*>*st*›
apply (*auto simp add*:*divide-simps*)
by (*simp add*:*algebra-simps*)
then show
{*x. f x* = *0* ∧ *st* ≤ *x* ∧ *x* ≤ *tt*} ⊆ *linepath st tt ‘F*
{*x. g x* = *0* ∧ *st* ≤ *x* ∧ *x* ≤ *tt*} ⊆ *linepath st tt ‘G*
unfolding *F-def G-def*
by (*clarify*|*rule-tac x*=*vl x* **in** *rev-image-eqI*,*auto*)+
qed
ultimately have
finite {*x. f x* = *0* ∧ *st* ≤ *x* ∧ *x* ≤ *tt*}
finite {*x. g x* = *0* ∧ *st* ≤ *x* ∧ *x* ≤ *tt*}
by (*auto elim*:*rev-finite-subset*)
from *finite-UnI*[*OF this*] **show** *finite* {*x.* (*f x* = *0* ∨ *g x* = *0*) ∧ *st* ≤ *x* ∧ *x*
≤ *tt*}
by (*elim rev-finite-subset*,*auto*)
qed
ultimately show *?thesis*
unfolding *Let-def*
apply (*fold zr-def ϑ-def β-def ϑ-if-def β-if-def*)+
using *jstart-eq jfinish-eq index-eq that* **by** *auto*
qed
ultimately show *?thesis* **by** *fastforce*
qed

lemma *jumpF-pathstart-part-circlepath*:
assumes *st*<*tt r*>*0 cmod* (*z*−*z0*) ≠*r*
shows *jumpF-pathstart* (*part-circlepath z r st tt*) *z0* = (

```
        if r * cos st + Re z − Re z0 = 0 then
          (let
            Δ = r* sin st + Im z − Im z0
          in
            if (sin st > 0 ∨ cos st=1 ) ∧ Δ < 0
                ∨ (sin st < 0 ∨  cos st=−1 ) ∧ Δ > 0  then
              1/2
            else
              − 1/2)
        else 0)
proof −
  define f where f=(λi. r * sin i + Im z − Im z0)
  define g where g=(λi. r * cos i + Re z − Re z0)
  have jumpF-eq:jumpF-pathstart (part-circlepath z r st tt) z0 = jumpF (λi. f i/g
i) (at-right st)
  proof −
    have jumpF-pathstart (part-circlepath z r st tt) z0
        = jumpF ((λi. f i/g i) o linepath st tt) (at-right 0)
     unfolding jumpF-pathstart-def part-circlepath-def exp-Euler f-def g-def comp-def
      by (simp add:cos-of-real sin-of-real algebra-simps)
    also have ... = jumpF (λi. f i/g i) (at-right st)
      using jumpF-linear-comp(2)[of tt−st (λi. f i/g i) st 0,symmetric] ‹st<tt›
      unfolding linepath-def by (auto simp add:algebra-simps)
    finally show ?thesis .
  qed
  have g-has-sgnx1:(g has-sgnx 1) (at-right st) when g st=0 sin st < 0 ∨ cos
st=−1
  proof −
    have ?thesis when sin st<0
    proof −
      have (g has-sgnx sgn (− r * sin st)) (at-right st)
        apply (rule has-sgnx-derivative-at-right[of g − r * sin st st])
        subgoal unfolding g-def by (auto intro!:derivative-eq-intros)
        subgoal using ‹g st=0› .
        subgoal using ‹r>0› ‹sin st<0› by (simp add: mult-pos-neg)
        done
      then show ?thesis using ‹r>0› that by (simp add: sgn-mult)
    qed
    moreover have ?thesis when cos st = −1
    proof −
      have g i > 0 when st<i i<st+pi for i
      proof −
        obtain k where k-def:st = 2 * of-int k * pi+ pi
          using ‹cos st = −1› by (metis cos-eq-minus1 distrib-left mult.commute
mult.right-neutral)
        have cos (i−st) < 1 using cos-monotone-0-pi[of 0 i−st ] that by auto
        moreover have cos (i−st) = − cos i
          apply (rule cos-eq-neg-periodic-intro[of - - −k−1])
          unfolding k-def by (auto simp add:algebra-simps)
```

```
      ultimately have cos i>−1 by auto
      then have cos st<cos i using ‹cos st=−1› by auto
      have 0 = r ∗ cos st + Re z − Re z0
        using ‹g st = 0› unfolding g-def by auto
      also have ... < r ∗ cos i + Re z − Re z0
        using ‹cos st < cos i› ‹r>0› by auto
      finally show ?thesis unfolding g-def by auto
    qed
    then show ?thesis
      unfolding has-sgnx-def eventually-at-right
      apply (intro exI[where x=st+pi])
      by auto
  qed
  ultimately show ?thesis using that(2) by auto
qed
have g-has-sgnx2:(g has-sgnx −1) (at-right st) when g st=0 sin st > 0 ∨ cos
st=1
proof −
  have ?thesis when sin st>0
  proof −
    have (g has-sgnx sgn (− r ∗ sin st)) (at-right st)
      apply (rule has-sgnx-derivative-at-right[of - − r ∗ sin st])
      subgoal unfolding g-def by (auto intro!:derivative-eq-intros)
      subgoal using ‹g st=0› .
      subgoal using ‹r>0› ‹sin st>0› by (simp add: mult-pos-neg)
      done
    then show ?thesis using ‹r>0› that by (simp add: sgn-mult)
  qed
  moreover have ?thesis when cos st=1
  proof −
    have g i < 0 when st<i i<st+pi for i
    proof −
      obtain k where k-def:st = 2 ∗ of-int k ∗ pi
        using ‹cos st=1› cos-one-2pi-int by auto
      have cos (i−st) < 1 using cos-monotone-0-pi[of 0 i−st ] that by auto
      moreover have cos (i−st) = cos i
        apply (rule cos-eq-periodic-intro[of - - −k])
        unfolding k-def by (auto simp add:algebra-simps)
      ultimately have cos i<1 by auto
      then have cos st>cos i using ‹cos st=1› by auto
      have 0 = r ∗ cos st + Re z − Re z0
        using ‹g st = 0› unfolding g-def by auto
      also have ... > r ∗ cos i + Re z − Re z0
        using ‹cos st > cos i› ‹r>0› by auto
      finally show ?thesis unfolding g-def by auto
    qed
    then show ?thesis
      unfolding has-sgnx-def eventually-at-right
      apply (intro exI[where x=st+pi])
```

```
        by auto
    qed
    ultimately show ?thesis using that(2) by auto
  qed

  have ?thesis when r * cos st + Re z − Re z0 ≠ 0
  proof −
    have g st ≠0 using that unfolding g-def by auto
    then have continuous (at-right st) (λi. f i / g i)
      unfolding f-def g-def by (auto intro!:continuous-intros)
    then have jumpF (λi. f i/g i) (at-right st) = 0
      using jumpF-not-infinity[of at-right st (λi. f i/g i)] by auto
    then show ?thesis using jumpF-eq that by auto
  qed
  moreover have ?thesis when r * cos st + Re z − Re z0 = 0
    (sin st > 0 ∨ (cos st=1) ) ∧ f st < 0
                ∨ (sin st < 0 ∨  (cos st=−1) ) ∧ f st > 0
  proof −
   have g st = 0 f st≠0 and g-cont: continuous (at-right st) g and f-cont:continuous
(at-right st) f
      using that unfolding g-def f-def by (auto intro!:continuous-intros)
    have (g has-sgnx sgn (f st)) (at-right st)
      using g-has-sgnx1[OF ‹g st=0›] g-has-sgnx2[OF ‹g st=0›] that(2) by auto
    then have LIM x at-right st. f x / g x :> at-top
      apply (subst filterlim-divide-at-bot-at-top-iff[of f f st at-right st g])
     using ‹f st≠0› ‹g st = 0› g-cont f-cont by (auto simp add: continuous-within)
    then have jumpF (λi. f i/g i) (at-right st) = 1/2
      unfolding jumpF-def by auto
    then show ?thesis using jumpF-eq that unfolding f-def by auto
  qed
  moreover have ?thesis when r * cos st + Re z − Re z0 = 0
    ¬ ((sin st > 0 ∨ cos st=1 ) ∧ f st < 0
                ∨ (sin st < 0 ∨  cos st=−1 ) ∧ f st > 0)
  proof −
    define neq1 where neq1 = (∀ k::int. st ≠ 2*k*pi)
    define neq2 where neq2 = (∀ k::int. st ≠ 2*k*pi+pi)
     have g st = 0 and g-cont: continuous (at-right st) g and f-cont:continuous
(at-right st) f
      using that unfolding g-def f-def by (auto intro!:continuous-intros)
    have f st≠0
    proof (rule ccontr)
      assume ¬ f st ≠ 0
      then have f st = 0 by auto
     then have Im (z0 − z) =r * sin st  Re (z0 − z) = r * cos st using ‹g st=0›
        unfolding f-def g-def by (auto simp add:algebra-simps)
      then have cmod (z0 − z) = sqrt((r * sin st)^2 + (r * cos st)^2)
        unfolding cmod-def by auto
      also have ... = sqrt (r^2 * ((sin st)^2 + (cos st)^2))
        by (auto simp only:algebra-simps power-mult-distrib)
```

```
      also have ... = r
        using ‹r>0› by simp
      finally have cmod (z0 − z) = r .
    then show False using ‹cmod (z−z0) ≠r› by (simp add: norm-minus-commute)
    qed
    have (sin st > 0 ∨ (cos st=1) ) ∧ f st > 0 ∨ (sin st < 0 ∨ (cos st=−1) ) ∧
f st < 0
    proof −
      have sin st = 0 ⟷ cos st=−1 ∨ cos st=1
      by (metis (no-types, opaque-lifting) add.right-neutral cancel-comm-monoid-add-class.diff-cancel

          cos-diff cos-zero mult-eq-0-iff power2-eq-1-iff power2-eq-square sin-squared-eq)
      moreover have ((sin st ≤ 0 ∧ cos st ≠1 ) ∨ f st > 0) ∧ ((sin st ≥ 0 ∧ cos
st≠−1) ∨ f st < 0)
        using that(2) ‹f st≠0› by argo
      ultimately show ?thesis by (meson linorder-neqE-linordered-idom not-le)
    qed
    then have (g has-sgnx − sgn (f st)) (at-right st)
      using g-has-sgnx1[OF ‹g st=0›] g-has-sgnx2[OF ‹g st=0›] by auto
    then have LIM x at-right st. f x / g x :> at-bot
      apply (subst filterlim-divide-at-bot-at-top-iff[of f f st at-right st g])
     using ‹f st≠0› ‹g st = 0› g-cont f-cont by (auto simp add: continuous-within)
    then have jumpF (λi. f i/g i) (at-right st) = −1/2
      unfolding jumpF-def by auto
    then show ?thesis using jumpF-eq that unfolding f-def by auto
  qed
  ultimately show ?thesis by fast
qed

lemma jumpF-pathfinish-part-circlepath:
  assumes st<tt r>0 cmod (z−z0) ≠r
  shows jumpF-pathfinish (part-circlepath z r st tt) z0 = (
          if r ∗ cos tt + Re z − Re z0 = 0 then
            (let
              Δ = r∗ sin tt + Im z − Im z0
            in
              if (sin tt > 0 ∨ cos tt=−1 ) ∧ Δ < 0
                  ∨ (sin tt < 0 ∨ cos tt=1 ) ∧ Δ > 0 then
                − 1/2
              else
                1/2)
          else 0)
proof −
  define f where f=(λi. r ∗ sin i + Im z − Im z0)
  define g where g=(λi. r ∗ cos i + Re z − Re z0)
  have jumpF-eq:jumpF-pathfinish (part-circlepath z r st tt) z0 = jumpF (λi. f i/g
i) (at-left tt)
  proof −
    have jumpF-pathfinish (part-circlepath z r st tt) z0
```

```
      = jumpF ((λi. f i/g i) o linepath st tt) (at-left 1)
    unfolding jumpF-pathfinish-def part-circlepath-def exp-Euler f-def g-def comp-def
      by (simp add:cos-of-real sin-of-real algebra-simps)
    also have ... = jumpF (λi. f i/g i) (at-left tt)
      using jumpF-linear-comp(1)[of tt−st (λi. f i/g i) st 1,symmetric] ‹st<tt›
      unfolding linepath-def by (auto simp add:algebra-simps)
    finally show ?thesis .
  qed
  have g-has-sgnx1:(g has-sgnx −1) (at-left tt) when g tt=0 sin tt < 0 ∨ cos tt=1

  proof −
    have ?thesis when sin tt<0
    proof −
      have (g has-sgnx − sgn (− r ∗ sin tt)) (at-left tt)
        apply (rule has-sgnx-derivative-at-left[of - − r ∗ sin tt])
        subgoal unfolding g-def by (auto intro!:derivative-eq-intros)
        subgoal using ‹g tt=0› .
        subgoal using ‹r>0› ‹sin tt<0› by (simp add: mult-pos-neg)
        done
      then show ?thesis using ‹r>0› that by (simp add: sgn-mult)
    qed
    moreover have ?thesis when cos tt=1
    proof −
      have g i < 0 when tt−pi<i i<tt for i
      proof −
        obtain k where k-def:tt = 2 ∗ of-int k ∗ pi
          using ‹cos tt=1› cos-one-2pi-int by auto
        have cos (i−tt) < 1
          using cos-monotone-0-pi[of 0 tt−i ] that cos-minus[of tt−i,simplified] by
auto
        moreover have cos (i−tt) = cos i
          apply (rule cos-eq-periodic-intro[of - - −k])
          unfolding k-def by (auto simp add:algebra-simps)
        ultimately have cos i<1 by auto
        then have cos tt>cos i using ‹cos tt=1› by auto
        have 0 = r ∗ cos tt + Re z − Re z0
          using ‹g tt = 0› unfolding g-def by auto
        also have ... > r ∗ cos i + Re z − Re z0
          using ‹cos tt > cos i› ‹r>0› by auto
        finally show ?thesis unfolding g-def by auto
      qed
      then show ?thesis
        unfolding has-sgnx-def eventually-at-left
        apply (intro exI[where x=tt−pi])
        by auto
    qed
    ultimately show ?thesis using that(2) by auto
  qed
  have g-has-sgnx2:(g has-sgnx 1) (at-left tt) when g tt=0 sin tt > 0 ∨ cos tt=−1
```

```
proof -
  have ?thesis when sin tt>0
  proof -
    have (g has-sgnx - sgn (- r * sin tt)) (at-left tt)
      apply (rule has-sgnx-derivative-at-left[of - - r * sin tt])
      subgoal unfolding g-def by (auto intro!:derivative-eq-intros)
      subgoal using ‹g tt=0› .
      subgoal using ‹r>0› ‹sin tt>0› by (simp add: mult-pos-neg)
      done
    then show ?thesis using ‹r>0› that by (simp add: sgn-mult)
  qed
  moreover have ?thesis when cos tt = -1
  proof -
    have g i > 0 when tt-pi<i i<tt for i
    proof -
      obtain k where k-def:tt = 2 * of-int k * pi+ pi
          using ‹cos tt = -1› by (metis cos-eq-minus1 distrib-left mult.commute
mult.right-neutral)
      have cos (i-tt) < 1
          using cos-monotone-0-pi[of 0 tt-i ] that cos-minus[of tt-i,simplified]
          by auto
      moreover have cos (i-tt) = - cos i
        apply (rule cos-eq-neg-periodic-intro[of - - -k-1])
        unfolding k-def by (auto simp add:algebra-simps)
      ultimately have cos i>-1 by auto
      then have cos tt<cos i using ‹cos tt=-1› by auto
      have 0 = r * cos tt + Re z - Re z0
        using ‹g tt = 0› unfolding g-def by auto
      also have ... < r * cos i + Re z - Re z0
        using ‹cos tt < cos i› ‹r>0› by auto
      finally show ?thesis unfolding g-def by auto
    qed
    then show ?thesis
      unfolding has-sgnx-def eventually-at-left
      apply (intro exI[where x=tt-pi])
      by auto
  qed
  ultimately show ?thesis using that(2) by auto
qed

have ?thesis when r * cos tt + Re z - Re z0 ≠ 0
proof -
  have g tt ≠0 using that unfolding g-def by auto
  then have continuous (at-left tt) (λi. f i / g i)
    unfolding f-def g-def by (auto intro!:continuous-intros)
  then have jumpF (λi. f i/g i) (at-left tt) = 0
    using jumpF-not-infinity[of at-left tt (λi. f i/g i)] by auto
  then show ?thesis using jumpF-eq that by auto
```

qed
moreover have *?thesis* **when** *r* ∗ *cos tt* + *Re z* − *Re z0* = *0*
(*sin tt* > *0* ∨ *cos tt*=−*1*) ∧ *f tt* < *0*
∨ (*sin tt* < *0* ∨ *cos tt*=*1*) ∧ *f tt* > *0*
proof −
have *g tt* = *0 f tt*≠*0* **and** *g-cont*: *continuous* (*at-left tt*) *g* **and** *f-cont*:*continuous* (*at-left tt*) *f*
using *that* **unfolding** *g-def f-def* **by** (*auto intro*!:*continuous-intros*)
have (*g has-sgnx* − *sgn* (*f tt*)) (*at-left tt*)
using *g-has-sgnx1*[*OF* ‹*g tt*=*0*›] *g-has-sgnx2*[*OF* ‹*g tt*=*0*›] *that*(*2*) **by** *auto*
then have *LIM x at-left tt. f x* / *g x* :> *at-bot*
apply (*subst filterlim-divide-at-bot-at-top-iff*[*of f f tt at-left tt g*])
using ‹*f tt*≠*0*› ‹*g tt* = *0*› *g-cont f-cont* **by** (*auto simp add*: *continuous-within*)
then have *jumpF* (λ*i. f i*/*g i*) (*at-left tt*) = − *1*/*2*
unfolding *jumpF-def* **by** *auto*
then show *?thesis* **using** *jumpF-eq that* **unfolding** *f-def* **by** *auto*
qed
moreover have *?thesis* **when** *r* ∗ *cos tt* + *Re z* − *Re z0* = *0*
¬ ((*sin tt* > *0* ∨ *cos tt*=−*1*) ∧ *f tt* < *0*
∨ (*sin tt* < *0* ∨ *cos tt*=*1*) ∧ *f tt* > *0*)
proof −
have *g tt* = *0* **and** *g-cont*: *continuous* (*at-left tt*) *g* **and** *f-cont*:*continuous* (*at-left tt*) *f*
using *that* **unfolding** *g-def f-def* **by** (*auto intro*!:*continuous-intros*)
have *f tt*≠*0*
proof (*rule ccontr*)
assume ¬ *f tt* ≠ *0*
then have *f tt* = *0* **by** *auto*
then have *Im* (*z0* − *z*) =*r* ∗ *sin tt Re* (*z0* − *z*) = *r* ∗ *cos tt* **using** ‹*g tt*=*0*›
unfolding *f-def g-def* **by** (*auto simp add*:*algebra-simps*)
then have *cmod* (*z0* − *z*) = *sqrt*((*r* ∗ *sin tt*)^*2* + (*r* ∗ *cos tt*)^*2*)
unfolding *cmod-def* **by** *auto*
also have ... = *sqrt* (*r*^*2* ∗ ((*sin tt*)^*2* + (*cos tt*)^*2*))
by (*auto simp only*:*algebra-simps power-mult-distrib*)
also have ... = *r*
using ‹*r*>*0*› **by** *simp*
finally have *cmod* (*z0* − *z*) = *r* **.**
then show *False* **using** ‹*cmod* (*z*−*z0*) ≠*r*› **by** (*simp add*: *norm-minus-commute*)
qed
have (*sin tt* > *0* ∨ *cos tt*=−*1*) ∧ *f tt* > *0* ∨ (*sin tt* < *0* ∨ *cos tt*=*1*) ∧ *f tt* < *0*
proof −
have *sin tt* = *0* ⟷ *cos tt*=−*1* ∨ *cos tt*=*1*
by (*metis* (*no-types*, *opaque-lifting*) *add.right-neutral cancel-comm-monoid-add-class.diff-cancel*
cos-diff cos-zero mult-eq-0-iff power2-eq-1-iff power2-eq-square sin-squared-eq)
moreover have ((*sin tt* ≤ *0* ∧ *cos tt* ≠−*1*) ∨ *f tt* > *0*) ∧ ((*sin tt* ≥ *0* ∧ *cos tt*≠*1*) ∨ *f tt* < *0*)
using *that*(*2*) ‹*f tt*≠*0*› **by** *argo*

ultimately show *?thesis* **by** (*meson linorder-neqE-linordered-idom not-le*)
qed
then have (*g has-sgnx sgn* (*f tt*)) (*at-left tt*)
using *g-has-sgnx1*[*OF* ‹*g tt=0*›] *g-has-sgnx2*[*OF* ‹*g tt=0*›] **by** *auto*
then have *LIM x at-left tt. f x / g x :> at-top*
apply (*subst filterlim-divide-at-bot-at-top-iff*[*of f f tt at-left tt g*])
using ‹*f tt≠0*› ‹*g tt = 0*› *g-cont f-cont* **by** (*auto simp add*: *continuous-within*)
then have *jumpF* (*λi. f i/g i*) (*at-left tt*) = *1/2*
unfolding *jumpF-def* **by** *auto*
then show *?thesis* **using** *jumpF-eq that* **unfolding** *f-def* **by** *auto*
qed
ultimately show *?thesis* **by** *fast*
qed

lemma
fixes *z0 z*::*complex* **and** *r*::*real*
defines *upper* ≡ *cindex-pathE* (*part-circlepath z r 0 pi*) *z0*
and *lower* ≡ *cindex-pathE* (*part-circlepath z r pi* (*2∗pi*)) *z0*
shows *cindex-pathE-circlepath-upper*:
⟦*cmod* (*z0−z*) < *r*⟧ ⟹ *upper* = −*1*
⟦*Im* (*z0−z*) > *r*; |*Re* (*z0* − *z*)| < *r*⟧ ⟹ *upper* = *1*
⟦*Im* (*z0−z*) < −*r*; |*Re* (*z0* − *z*)| < *r*⟧ ⟹ *upper* = −*1*
⟦|*Re* (*z0* − *z*)| > *r*; *r>0*⟧ ⟹ *upper* = *0*
and *cindex-pathE-circlepath-lower*:
⟦*cmod* (*z0−z*) < *r*⟧ ⟹ *lower* = −*1*
⟦*Im* (*z0−z*) > *r*; |*Re* (*z0* − *z*)| < *r*⟧ ⟹ *lower* = −*1*
⟦*Im* (*z0−z*) < −*r*; |*Re* (*z0* − *z*)| < *r*⟧ ⟹ *lower* = *1*
⟦|*Re* (*z0* − *z*)| > *r*; *r>0*⟧ ⟹ *lower* = *0*
proof −
assume *assms*:*cmod* (*z0−z*) < *r*
have *zz-facts*:−*r<Re z* − *Re z0 Re z* − *Re z0<r r>0*
subgoal using *assms complex-Re-le-cmod le-less-trans* **by** *fastforce*
subgoal by (*metis assms complex-Re-le-cmod le-less-trans minus-complex.simps*(*1*) *norm-minus-commute*)
subgoal using *assms le-less-trans norm-ge-zero* **by** *blast*
done
define *ϑ* **where** *ϑ* = *arccos* ((*Re z0* − *Re z*) / *r*)
have *ϑ-bound*:*0* < *ϑ* ∧ *ϑ* < *pi*
unfolding *ϑ-def*
apply (*rule arccos-lt-bounded*)
using *zz-facts* **by** (*auto simp add*:*field-simps*)
have *Im-sin*:*abs* (*Im z0* − *Im z*) < *r* ∗ *sin ϑ*
proof −
define *zz* **where** *zz*=*z0*−*z*
have *sqrt* ((*Re zz*)^2 + (*Im zz*)^2) < *r*
using *assms* **unfolding** *zz-def cmod-def* **.**
then have (*Re zz*)^2 + (*Im zz*)^2 < *r^2*
by (*metis cmod-power2 dvd-refl linorder-not-le norm-complex-def power2-le-imp-le real-sqrt-power zero-le-power-eq-numeral*)

then have $(Im\ zz)^2 < r\hat{}2 - (Re\ zz)\hat{}2$ **by** *auto*
then have *abs* (*Im zz*) < *sqrt* (*r^2* − (*Re zz*)^*2*)
by (*simp add*: *real-less-rsqrt*)
then show *?thesis*
unfolding *ϑ-def zz-def*
apply (*subst sin-arccos-abs*)
subgoal using *zz-facts* **by** *auto*
subgoal using ‹*r*>*0*› **by** (*auto simp add*:*field-simps divide-simps real-sqrt-divide*)
done
qed
show *upper* = − *1*
proof −
have *jumpF-pathstart* (*part-circlepath z r 0 pi*) *z0* = *0*
apply (*subst jumpF-pathstart-part-circlepath*)
using *zz-facts assms* **by** (*auto simp add*: *norm-minus-commute*)
moreover have *jumpF-pathfinish* (*part-circlepath z r 0 pi*) *z0* = *0*
apply (*subst jumpF-pathfinish-part-circlepath*)
using *zz-facts assms* **by** (*auto simp add*: *norm-minus-commute*)
ultimately show *?thesis* **using** *assms zz-facts ϑ-bound Im-sin* **unfolding** *upper-def*
apply (*subst cindex-pathE-part-circlepath*)
by (*fold ϑ-def*,*auto simp add*: *norm-minus-commute*)
qed
show *lower* = − *1*
proof −
have *jumpF-pathstart* (*part-circlepath z r pi* (*2*∗*pi*)) *z0* = *0*
apply (*subst jumpF-pathstart-part-circlepath*)
using *zz-facts assms* **by** (*auto simp add*: *norm-minus-commute*)
moreover have *jumpF-pathfinish* (*part-circlepath z r pi* (*2*∗*pi*)) *z0* = *0*
apply (*subst jumpF-pathfinish-part-circlepath*)
using *zz-facts assms* **by** (*auto simp add*: *norm-minus-commute*)
ultimately show *?thesis* **using** *assms zz-facts ϑ-bound Im-sin* **unfolding** *lower-def*
apply (*subst cindex-pathE-part-circlepath*)
by (*fold ϑ-def*,*auto simp add*: *norm-minus-commute*)
qed
next
assume *assms*:|*Re* (*z0* − *z*)| > *r r*>*0*
show *upper* = *0* **using** *assms* **unfolding** *upper-def*
apply (*subst cindex-pathE-part-circlepath*)
apply *auto*
by (*metis abs-Re-le-cmod abs-minus-commute eucl-less-le-not-le minus-complex.simps*(*1*))
show *lower* = *0*
using *assms* **unfolding** *lower-def*
apply (*subst cindex-pathE-part-circlepath*)
apply *auto*
by (*metis abs-Re-le-cmod abs-minus-commute eucl-less-le-not-le minus-complex.simps*(*1*))
next
assume *assms*:|*Re* (*z0* − *z*)| < *r*

```
  then have r>0 by auto

  define ϑ where ϑ = arccos ((Re z0 − Re z) / r)
  have ϑ-bound:0 < ϑ ∧ ϑ < pi
    unfolding ϑ-def
    apply (rule arccos-lt-bounded)
    using assms by (auto simp add:field-simps)
  note norm-minus-commute[simp]
  have jumpFs:
      jumpF-pathstart (part-circlepath z r 0 pi) z0 = 0
      jumpF-pathfinish (part-circlepath z r 0 pi) z0 = 0
      jumpF-pathstart (part-circlepath z r pi (2∗pi)) z0 = 0
      jumpF-pathfinish (part-circlepath z r pi (2∗pi)) z0 = 0
      when cmod (z0 − z) ≠ r
    subgoal by (subst jumpF-pathstart-part-circlepath,use assms that in auto)
    subgoal by (subst jumpF-pathfinish-part-circlepath,use assms that in auto)
    subgoal by (subst jumpF-pathstart-part-circlepath,use assms that in auto)
    subgoal by (subst jumpF-pathfinish-part-circlepath,use assms that in auto)
    done
  show upper = 1 lower = −1 when Im (z0−z) > r
  proof −
    have cmod (z0 − z) ≠ r
      using that assms abs-Im-le-cmod abs-le-D1 not-le by blast
    moreover have Im z0 − Im z > r ∗ sin ϑ
    proof −
      have r ∗ sin ϑ ≤ r
        using ‹r>0› by auto
      also have ... < Im z0 − Im z using that by auto
      finally show ?thesis .
    qed
     ultimately show upper = 1  using assms jumpFs ϑ-bound that unfolding
upper-def
      apply (subst cindex-pathE-part-circlepath)
      by (fold ϑ-def,auto)
    have Im z − Im z0 < r ∗ sin ϑ
    proof −
      have Im z − Im z0  <0 using that ‹r>0› by auto
     moreover have r ∗ sin ϑ>0 using ‹r>0› ϑ-bound by (simp add: sin-gt-zero)
      ultimately show ?thesis by auto
    qed
    then show lower = −1 using ‹cmod (z0 − z) ≠ r› ‹Im z0 − Im z > r ∗ sin
ϑ›
        assms jumpFs ϑ-bound that unfolding lower-def
      apply (subst cindex-pathE-part-circlepath)
      by (fold ϑ-def,auto)
  qed
  show upper = − 1 lower = 1 when Im (z0−z) < −r
  proof −
    have cmod (z0 − z) ≠ r
```

using *that assms*
by (*metis abs-Im-le-cmod abs-le-D1 minus-complex.simps*(*2*) *minus-diff-eq neg-less-iff-less*
norm-minus-cancel not-le)
moreover have *Im z* − *Im z0* > *r* ∗ *sin* ϑ
proof −
have *r* ∗ *sin* ϑ ≤ *r*
using ‹*r*>*0*› **by** *auto*
also have ... < *Im z* − *Im z0* **using** *that* **by** *auto*
finally show *?thesis* **.**
qed
moreover have *Im z0* − *Im z* < *r* ∗ *sin* ϑ
proof −
have *Im z0* − *Im z*<*0* **using** *that* ‹*r*>*0*› **by** *auto*
moreover have *r* ∗ *sin* ϑ>*0* **using** ‹*r*>*0*› ϑ*-bound* **by** (*simp add*: *sin-gt-zero*)
ultimately show *?thesis* **by** *auto*
qed
ultimately show *upper* = − *1* **using** *assms jumpFs* ϑ*-bound that* **unfolding** *upper-def*
apply (*subst cindex-pathE-part-circlepath*)
by (*fold* ϑ*-def*,*auto*)
show *lower* = *1*
using ‹*Im z0* − *Im z* < *r* ∗ *sin* ϑ› ‹*Im z* − *Im z0* > *r* ∗ *sin* ϑ› ‹*cmod* (*z0* − *z*) ≠ *r*›
assms jumpFs ϑ*-bound that* **unfolding** *lower-def*
apply (*subst cindex-pathE-part-circlepath*)
by (*fold* ϑ*-def*,*auto*)
qed
qed

lemma *jumpF-pathstart-linepath*:
jumpF-pathstart (*linepath a b*) *z* =
(*if Re a* = *Re z* ∧ *Im a*≠*Im z* ∧ *Re b* ≠ *Re a then*
if (*Im a*>*Im z* ∧ *Re b* > *Re a*) ∨ (*Im a*<*Im z* ∧ *Re b* < *Re a*) *then 1*/*2 else* −*1*/*2*
else 0)
proof −
define *f* **where** *f*=(λ*t*. (*Im b* − *Im a*)∗ *t* + (*Im a* − *Im z*))
define *g* **where** *g*=(λ*t*. (*Re b* − *Re a*)∗ *t* + (*Re a* − *Re z*))
have *jump-eq*:*jumpF-pathstart* (*linepath a b*) *z* = *jumpF* (λ*t*. *f t*/*g t*) (*at-right 0*)
unfolding *jumpF-pathstart-def f-def linepath-def g-def*
by (*auto simp add*:*algebra-simps*)
have *?thesis* **when** *Re a*≠*Re z*
proof −
have *jumpF-pathstart* (*linepath a b*) *z* = *0*
unfolding *jumpF-pathstart-def*
apply (*rule jumpF-im-divide-Re-0*)
apply *auto*
by (*auto simp add*:*linepath-def that*)

then show *?thesis* **using** *that* **by** *auto*
qed
moreover have *?thesis* **when** *Re a=Re z Im a = Im z*
proof −
define *c* **where** *c=(Im b − Im a) / (Re b − Re a)*
have *jumpF (λt. f t/g t) (at-right 0) = jumpF (λ-. c) (at-right 0)*
proof (*rule jumpF-cong*)
show ∀ $_F$ *x in at-right 0. f x / g x = c*
unfolding *eventually-at-right f-def g-def c-def* **using** *that*
apply (*intro exI*[**where** *x=1*])
by *auto*
qed *simp*
then show *?thesis* **using** *jump-eq that* **by** *auto*
qed
moreover have *?thesis* **when** *Re a=Re z Re b = Re a*
proof −
have (*λt. f t/g t*) = (*λ-. 0*) **unfolding** *f-def g-def* **using** *that* **by** *auto*
then have *jumpF (λt. f t/g t) (at-right 0) = jumpF (λ-. 0) (at-right 0)* **by** *auto*
then show *?thesis* **using** *jump-eq that* **by** *auto*
qed
moreover have *?thesis* **when** *Re a = Re z (Im a>Im z ∧ Re b > Re a) ∨ (Im a<Im z ∧ Re b < Re a)*
proof −
have *LIM x at-right 0. f x / g x :> at-top*
apply (*subst filterlim-divide-at-bot-at-top-iff*[*of - Im a − Im z*])
unfolding *f-def g-def* **using** *that*
by (*auto intro*!:*tendsto-eq-intros sgnx-eq-intros*)
then have *jumpF (λt. f t/g t) (at-right 0) = 1/2*
unfolding *jumpF-def* **by** *simp*
then show *?thesis* **using** *jump-eq that* **by** *auto*
qed
moreover have *?thesis* **when** *Re a = Re z Im a≠Im z Re b ≠ Re a*
¬ ((*Im a>Im z ∧ Re b > Re a*) ∨ (*Im a<Im z ∧ Re b < Re a*))
proof −
have (*Im a>Im z ∧ Re b < Re a*) ∨ (*Im a<Im z ∧ Re b > Re a*)
using *that* **by** *argo*
then have *LIM x at-right 0. f x / g x :> at-bot*
apply (*subst filterlim-divide-at-bot-at-top-iff*[*of - Im a − Im z*])
unfolding *f-def g-def* **using** *that*
by (*auto intro*!:*tendsto-eq-intros sgnx-eq-intros*)
moreover then have ¬ (*LIM x at-right 0. f x / g x :> at-top*)
using *filterlim-at-top-at-bot* **by** *fastforce*
ultimately have *jumpF (λt. f t/g t) (at-right 0) = − 1/2*
unfolding *jumpF-def* **by** *simp*
then show *?thesis* **using** *jump-eq that* **by** *auto*
qed
ultimately show *?thesis* **by** *fast*
qed

```
lemma jumpF-pathfinish-linepath:
  jumpF-pathfinish (linepath a b) z =
    (if Re b = Re z ∧ Im b ≠Im z ∧ Re b ≠ Re a then
       if (Im b>Im z ∧ Re a > Re b) ∨ (Im b<Im z ∧ Re a < Re b) then 1/2 else
−1/2
     else 0)
proof −
  define f where f=(λt. (Im b − Im a )∗ t + (Im a − Im z))
  define g where g=(λt. (Re b − Re a )∗ t + (Re a − Re z))
  have jump-eq:jumpF-pathfinish (linepath a b) z = jumpF (λt. f t/g t) (at-left 1)
    unfolding jumpF-pathfinish-def f-def linepath-def g-def
    by (auto simp add:algebra-simps)
  have ?thesis when Re b≠Re z
  proof −
    have jumpF-pathfinish (linepath a b) z = 0
      unfolding jumpF-pathfinish-def
      apply (rule jumpF-im-divide-Re-0)
        apply auto
      by (auto simp add:linepath-def that)
    then show ?thesis using that by auto
  qed
  moreover have ?thesis when Re z=Re b Im z = Im b
  proof −
    define c where c=(Im a − Im b) / (Re a − Re b)
    have jumpF (λt. f t/g t) (at-left 1) = jumpF (λ-. c) (at-left 1)
    proof (rule jumpF-cong)
      have f x / g x = c when x<1 for x
      proof −
        have f x / g x = ((Im a − Im b)∗(1−x))/((Re a − Re b)∗(1−x))
          unfolding f-def g-def
          by (auto simp add:algebra-simps ‹Re z=Re b› ‹Im z = Im b›)
        also have ... = c
          using that unfolding c-def by auto
        finally show ?thesis .
      qed
      then show ∀F x in at-left 1. f x / g x = c
        unfolding eventually-at-left using that
        apply (intro exI[where x=0])
        by auto
    qed simp
    then show ?thesis using jump-eq that by auto
  qed
  moreover have ?thesis when Re a=Re z Re b = Re a
  proof −
    have (λt. f t/g t) = (λ-. 0) unfolding f-def g-def using that by auto
    then have jumpF (λt. f t/g t) (at-left 1) = jumpF (λ-. 0) (at-left 1) by auto
    then show ?thesis using jump-eq that by auto
  qed
```

```
  moreover have ?thesis when Re b = Re z (Im b>Im z ∧ Re a > Re b) ∨ (Im
b<Im z ∧ Re a < Re b)
  proof –
    have LIM x at-left 1. f x / g x :> at-top
    proof –
      have (g has-real-derivative Re b − Re a) (at 1)
        unfolding g-def by (auto intro!:derivative-eq-intros)
      from has-sgnx-derivative-at-left[OF this]
      have (g has-sgnx sgn (Im b − Im z)) (at-left 1)
        using that unfolding g-def by auto
      then show ?thesis
        apply (subst filterlim-divide-at-bot-at-top-iff[of - Im b − Im z])
        unfolding f-def g-def using that by (auto intro!:tendsto-eq-intros)
    qed
    then have jumpF (λt. f t/g t) (at-left 1) = 1/2
      unfolding jumpF-def by simp
    then show ?thesis using jump-eq that by auto
  qed
  moreover have ?thesis when Re b = Re z Im b≠Im z Re b ≠ Re a
      ¬ ((Im b>Im z ∧ Re a > Re b) ∨ (Im b<Im z ∧ Re a < Re b))
  proof –
    have (Im b>Im z ∧ Re a < Re b) ∨ (Im b<Im z ∧ Re a > Re b)
      using that by argo
    have LIM x at-left 1. f x / g x :> at-bot
    proof –
      have (g has-real-derivative Re b − Re a) (at 1)
        unfolding g-def by (auto intro!:derivative-eq-intros)
      from has-sgnx-derivative-at-left[OF this]
      have (g has-sgnx − sgn (Im b − Im z)) (at-left 1)
        using that unfolding g-def by auto
      then show ?thesis
        apply (subst filterlim-divide-at-bot-at-top-iff[of - Im b − Im z])
        unfolding f-def g-def using that by (auto intro!:tendsto-eq-intros )
    qed
    moreover then have ¬ (LIM x at-left 1. f x / g x :> at-top)
      using filterlim-at-top-at-bot by fastforce
    ultimately have jumpF (λt. f t/g t) (at-left 1) = − 1/2
      unfolding jumpF-def by simp
    then show ?thesis using jump-eq that by auto
  qed
  ultimately show ?thesis by argo
qed
```

6.4 Setting up the method for evaluating winding numbers

```
lemma pathfinish-pathstart-partcirclepath-simps:
  pathstart (part-circlepath z0 r (3∗pi/2) tt) = z0 − Complex 0 r
  pathstart (part-circlepath z0 r (2∗pi) tt) = z0 + r
  pathfinish (part-circlepath z0 r st (3∗pi/2)) = z0 − Complex 0 r
```

*pathfinish (part-circlepath z0 r st (2*pi)) = z0 + r*
pathstart (part-circlepath z0 r 0 tt) = z0 + r
pathstart (part-circlepath z0 r (pi/2) tt) = z0 + Complex 0 r
pathstart (part-circlepath z0 r (pi) tt) = z0 − r
pathfinish (part-circlepath z0 r st 0) = z0+r
pathfinish (part-circlepath z0 r st (pi/2)) = z0 + Complex 0 r
pathfinish (part-circlepath z0 r st (pi)) = z0 − r
unfolding *part-circlepath-def linepath-def pathstart-def pathfinish-def exp-Euler*
subgoal
  apply(*simp, subst sin.minus-1*[*symmetric*],*subst cos.minus-1*[*symmetric*])
  by (*simp add*: *complex-of-real-i*)
subgoal
  by (*simp add*: *complex-of-real-i*)
subgoal
  apply(*simp, subst sin.minus-1*[*symmetric*],*subst cos.minus-1*[*symmetric*])
  by (*simp add*: *complex-of-real-i*)
by (*simp-all add*: *complex-of-real-i*)

lemma *winding-eq-intro*:
finite-ReZ-segments g z ⟹
valid-path g ⟹
z ∉ path-image g ⟹
pathfinish g = pathstart g ⟹
*− of-real(cindex-pathE g z) =2*n ⟹*
winding-number g z = (n::complex)
apply (*subst winding-number-cindex-pathE*[*of g z*])
by (*auto simp add:field-simps*)

named-theorems *winding-intros* **and** *winding-simps*

lemmas [*winding-intros*] =
finite-ReZ-segments-joinpaths
valid-path-join
path-join-imp
not-in-path-image-join

lemmas [*winding-simps*] =
finite-ReZ-segments-linepath
finite-ReZ-segments-part-circlepath
jumpF-pathfinish-joinpaths
jumpF-pathstart-joinpaths
pathfinish-linepath
pathstart-linepath
pathfinish-join
pathstart-join
valid-path-linepath
valid-path-part-circlepath
path-part-circlepath
Re-complex-of-real

Im-complex-of-real
of-real-linepath
pathfinish-pathstart-partcirclepath-simps

method *rep-subst* =
(*subst cindex-pathE-joinpaths*; *rep-subst*)?

The method "eval_winding" *1* will try to simplify of the form *winding-number g z* = *n* where *n* is an integer and *g* is a closed path comprised of *linepath*, *part-circlepath* and (+++).

Suppose *g* = *l1* +++ *l2*, usually, the key behind the success of this framework is whether we can prove *z* ∉ *path-image l1*, *z* ∉ *path-image l2* and calculate *cindex-pathE l1 z* and *cindex-pathE l2 z*.

method *eval-winding* =
((*rule-tac winding-eq-intro*;
rep-subst
)
, *auto simp only*:*winding-simps del*:*notI intro*!:*winding-intros*
, *tactic* ‹*distinct-subgoals-tac*›)

end

7 Some examples of applying the method winding_eval

theory *Winding-Number-Eval-Examples* **imports** *Winding-Number-Eval*
begin

lemma *example1*:
assumes *R*>*1*
shows *winding-number* (*part-circlepath 0 R 0 pi* +++ *linepath* (−*R*) *R*) i = *1*
proof (*eval-winding*,*simp-all*)
define *CR* **where** *CR* ≡*part-circlepath 0 R 0 pi*
define *L* **where** *L*≡ *linepath* (− (*complex-of-real R*)) *R*
show i ∉ *path-image CR* **unfolding** *CR-def* **using** ‹*R*>*1*›
by (*intro not-on-circlepathI*,*auto*)
show ∗:i ∉ *closed-segment* (− (*of-real R*)) *R* **using** ‹*R*>*1*› *complex-eq-iff*
by (*intro not-on-closed-segmentI*,*auto*)
from *cindex-pathE-linepath*[*OF this*] **have** *cindex-pathE L* i = −*1*
unfolding *L-def* **using** ‹*R*>*1*› **by** *auto*
moreover have *cindex-pathE CR* i = −*1*
unfolding *CR-def* **using** ‹*R*>*1*›
apply (*subst cindex-pathE-part-circlepath*)
by (*simp-all add*:*jumpF-pathstart-part-circlepath jumpF-pathfinish-part-circlepath*)
ultimately show − *complex-of-real* (*cindex-pathE CR* i) − *cindex-pathE L* i =
2
unfolding *L-def CR-def* **by** *auto*
qed

```
lemma example2:
  assumes R>1
  shows winding-number (part-circlepath 0 R 0 pi +++ linepath (−R) R) (−i) =
0
proof (eval-winding,simp-all)
  define CR where  CR ≡part-circlepath 0 R 0 pi
  define L where L≡ linepath (− (complex-of-real R)) R
  show −i ∉ path-image CR unfolding CR-def using ‹R>1›
    by (intro not-on-circlepathI,auto)
  show ∗:−i ∉ closed-segment (− (of-real R)) R using ‹R>1› complex-eq-iff
    by (intro not-on-closed-segmentI,auto)
  from cindex-pathE-linepath[OF this] have cindex-pathE L (−i) = 1
    unfolding L-def using ‹R>1› by auto
  moreover have cindex-pathE CR (−i) = −1
    unfolding CR-def using ‹R>1›
    apply (subst cindex-pathE-part-circlepath)
   by (simp-all add:jumpF-pathstart-part-circlepath jumpF-pathfinish-part-circlepath)
  ultimately show −cindex-pathE CR (−i) = cindex-pathE L (−i)
    unfolding L-def CR-def by auto
qed

lemma example3:
  fixes lb ub z :: complex
  defines rec ≡  linepath lb (Complex (Re ub) (Im lb)) +++ linepath (Complex
(Re ub) (Im lb)) ub
              +++ linepath ub (Complex (Re lb) (Im ub)) +++ linepath (Complex
(Re lb) (Im ub)) lb
  assumes order-asms:Re lb < Re z Re z < Re ub Im lb < Im z Im z < Im ub
  shows winding-number rec z = 1
  unfolding rec-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: z ∉ path-image ?l1
    apply (auto intro!: not-on-closed-segmentI-complex)
    using order-asms by (simp add: algebra-simps crossproduct-eq)
  show l2:z ∉ path-image ?l2
    apply (auto intro!: not-on-closed-segmentI-complex)
    using order-asms by (simp add: algebra-simps crossproduct-eq)
  show l3:z ∉ path-image ?l3
    apply (auto intro!: not-on-closed-segmentI-complex)
    using order-asms by (simp add: algebra-simps crossproduct-eq)
  show l4:z ∉ path-image ?l4
    apply (auto intro!: not-on-closed-segmentI-complex)
    using order-asms by (simp add: algebra-simps crossproduct-eq)
 show − complex-of-real (cindex-pathE ?l1 z + (cindex-pathE ?l2 z + (cindex-pathE
```

```
?l3 z +
        cindex-pathE ?l4 z))) = 2 * 1
  proof -
    have (Im z - 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 z = -1
      apply (subst cindex-pathE-linepath)
      using l3 order-asms by (auto simp add:algebra-simps)
    moreover have (Im lb - Im z) * (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 z = -1
      apply (subst cindex-pathE-linepath)
      using l1 order-asms by (auto simp add:algebra-simps)
    moreover have cindex-pathE ?l2 z = 0
      apply (subst cindex-pathE-linepath)
      using l2 order-asms by (auto simp add:algebra-simps)
    moreover have cindex-pathE ?l4 z = 0
      apply (subst cindex-pathE-linepath)
      using l4 order-asms by (auto simp add:algebra-simps)
    ultimately show ?thesis by auto
  qed
qed

end
```

8 Acknowledgements

The work was supported by the ERC Advanced Grant ALEXANDRIA (Project 742178), funded by the European Research Council and led by Professor Lawrence Paulson at the University of Cambridge, UK.

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.