The Sturm-Tarski Theorem

Wenda Li

April 20, 2020

Abstract

We have formalised the Sturm-Tarski theorem (also referred as the Tarski theorem): Given polynomials $p, q \in \mathbb{R}[x]$, the Sturm-Tarski theorem computes the sum of the signs of q over the roots of p by calculating some remainder sequences. Note, the better-known Sturm theorem is an instance of the Sturm-Tarski theorem when $q = 1$. The proof follows the classic book by Basu et al. [1] and Cyril Cohen's work in Coq [2]. With the Sturm-Tarski theorem proved, it is possible to further build a quantifier elimination procedure for real numbers as Cohen did in Coq. Another application of the Sturm-Tarski theorem is to build sign determination procedures for polynomials at real algebraic points, as described in our formalisation of real algebraic numbers [3].

theory *PolyMisc* **imports**
  HOL−Computational-Algebra.Polynomial-Factorial
begin

lemma *coprime-poly-0*:
  poly p x ≠ *0* ∨ *poly q x* ≠ *0* **if** *coprime p q*
  for *x* :: *'a* :: *field*
proof (*rule ccontr*)
  assume ¬ (*poly p x* ≠ *0* ∨ *poly q x* ≠ *0*)
  then have [:−*x*, *1*:] *dvd p* [:−*x*, *1*:] *dvd q*
    by (*simp-all add*: *poly-eq-0-iff-dvd*)
  with *that* **have** *is-unit* [:−*x*, *1*:]
    by (*rule coprime-common-divisor*)
  then show *False*
    by (*auto simp add*: *is-unit-pCons-iff*)
qed

lemma *smult-cancel*:
  fixes *p*::*'a*::*idom poly*
  assumes *c*≠*0* **and** *smult*: *smult c p* = *smult c q*
  shows *p*=*q*
proof −
  have *smult c* (*p*−*q*)=*0* **using** *smult* **by** (*metis diff-self smult-diff-right*)

thus *?thesis* **using** ‹*c≠0*› **by** *auto*
qed

lemma *dvd-monic*:
fixes *p q*:: *'a* :: *idom poly*
assumes *monic*:*lead-coeff p=1* **and** *p dvd* (*smult c q*) **and** *c≠0*
shows *p dvd q* **using** *assms*
proof (*cases q=0* ∨ *degree p=0*)
case *True*
thus *?thesis* **using** *assms*
by (*auto elim*!: *degree-eq-zeroE simp add*: *const-poly-dvd-iff*)
next
case *False*
hence *q≠0* **and** *degree p≠0* **by** *auto*
obtain *k* **where** *k*:*smult c q* = *p*∗*k* **using** *assms dvd-def* **by** *metis*
hence *k≠0* **by** (*metis False assms*(*3*) *mult-zero-right smult-eq-0-iff*)
hence *deg-eq*:*degree q*=*degree p* + *degree k*
by (*metis False assms*(*3*) *degree-0 degree-mult-eq degree-smult-eq k*)
have *c-dvd*:∀ *n*≤*degree k*. *c dvd coeff k* (*degree k* − *n*)
proof (*rule*,*rule*)
fix *n* **assume** *n* ≤ *degree k*
thus *c dvd coeff k* (*degree k* − *n*)
proof (*induct n rule*:*nat-less-induct*)
case (*1 n*)
define *T* **where** *T*≡(λ*i*. *coeff p i* ∗ *coeff k* (*degree p*+*degree k* − *n* − *i*))
have *c* ∗ *coeff q* (*degree q* − *n*) = (∑ *i*≤*degree q* − *n*. *coeff p i* ∗ *coeff k* (*degree q* − *n* − *i*))
using *coeff-mult*[*of p k degree q* − *n*] *k coeff-smult*[*of c q degree q* −*n*] **by** *auto*
also have ...=(∑ *i*≤*degree p*+*degree k* − *n*. *T i*)
using *deg-eq* **unfolding** *T-def* **by** *auto*
also have ...=(∑ *i*∈{*0*..<*degree p*}. *T i*) + *sum T* {(*degree p*)}+
sum T {*degree p* + *1*..*degree p* + *degree k* − *n*}
proof −
define *C* **where** *C*≡{{*0*..<*degree p*}, {*degree p*},{*degree p*+*1*..*degree p*+*degree k*−*n*}}
have ∀ *A*∈*C*. *finite A* **unfolding** *C-def* **by** *auto*
moreover have ∀ *A*∈*C*. ∀ *B*∈*C*. *A* ≠ *B* ⟶ *A* ∩ *B* = {}
unfolding *C-def* **by** *auto*
ultimately have *sum T* (⋃ *C*) = *sum* (*sum T*) *C*
using *sum.Union-disjoint* **by** *auto*
moreover have ⋃ *C*={..*degree p* + *degree k* − *n*}
using ‹*n* ≤ *degree k*› **unfolding** *C-def* **by** *auto*
moreover have *sum* (*sum T*) *C*= *sum T* {*0*..<*degree p*} + *sum T* {(*degree p*)} +
sum T {*degree p* + *1*..*degree p* + *degree k* − *n*}
proof −
have {*0*..<*degree p*}≠{*degree p*}
by (*metis atLeast0LessThan insertI1 lessThan-iff less-imp-not-eq*)

```
      moreover have {degree p}≠{degree p + 1..degree p + degree k − n}
       by (metis add.commute add-diff-cancel-right' atLeastAtMost-singleton-iff

            diff-self-eq-0 eq-imp-le not-one-le-zero)
      moreover have {0..<degree p}≠{degree p + 1..degree p + degree k − n}

        using ‹degree k≥n› ‹degree p≠0› by fastforce
       ultimately show ?thesis unfolding C-def by auto
     qed
     ultimately show ?thesis by auto
   qed
   also have ...=(∑ i∈{0..<degree p}. T i) +  coeff k (degree k − n)
   proof −
     have ∀ x∈{degree p + 1..degree p + degree k − n}. T x=0
       using coeff-eq-0[of p] unfolding T-def by simp
     hence sum T {degree p + 1..degree p + degree k − n}=0 by auto
     moreover have T (degree p)=coeff k (degree k − n)
       using monic by (simp add: T-def)
     ultimately show ?thesis by auto
   qed
   finally have c-coeff: c ∗ coeff q (degree q − n) = sum T {0..<degree p}
         + coeff k (degree k − n) .
   moreover have n≠0⟹c dvd sum T {0..<degree p}
   proof (rule dvd-sum)
     fix i assume i:i ∈ {0..<degree p} and n≠0
     hence (n+i−degree p)≤degree k using ‹n ≤ degree k› by auto
     moreover have n + i − degree p <n using i ‹n≠0› by auto
     ultimately have c dvd coeff k (degree k − (n+i−degree p))
       using 1(1) by auto
     hence c dvd coeff k (degree p + degree k − n − i)
       by (metis add-diff-cancel-left' deg-eq diff-diff-left dvd-0-right le-degree
         le-diff-conv add.commute ordered-cancel-comm-monoid-diff-class.diff-diff-right)
     thus c dvd T i unfolding T-def by auto
   qed
   moreover have n=0 ⟹?case
   proof −
     assume n=0
     hence ∀ i∈{0..<degree p}. coeff k (degree p + degree k − n − i) =0
       using coeff-eq-0[of k] by simp
     hence c ∗ coeff q (degree q − n) = coeff k (degree k − n)
       using c-coeff unfolding T-def by auto
     thus ?thesis by (metis dvdI)
   qed
   ultimately show ?case by (metis dvd-add-right-iff dvd-triv-left)
  qed
qed
hence ∀ n. c dvd coeff k n
  by (metis diff-diff-cancel dvd-0-right le-add2 le-add-diff-inverse le-degree)
then obtain f where f:∀ n. c ∗ f n=coeff k n unfolding dvd-def by metis
```

have $\forall_\infty$ *n*. *f n* = *0*
by (*metis* (*mono-tags*, *lifting*) *MOST-coeff-eq-0 MOST-mono assms*(*3*) *f mult-eq-0-iff*)
hence *smult c* (*Abs-poly f*)=*k*
using *f smult.abs-eq*[*of c Abs-poly f*] *Abs-poly-inverse*[*of f*] *coeff-inverse*[*of k*]
by *simp*
hence *q*=*p*∗ *Abs-poly f* **using** *k* ‹*c*≠*0*› *smult-cancel* **by** *auto*
thus *?thesis* **unfolding** *dvd-def* **by** *auto*
qed

lemma *poly-power-n-eq*:
fixes *x*::′*a* :: *idom*
assumes *n*≠*0*
shows *poly* ([:−*a*,*1*:] ^*n*) *x*=*0* ⟷ (*x*=*a*) **using** *assms*
by (*induct n*,*auto*)

lemma *poly-power-n-odd*:
fixes *x a*:: *real*
assumes *odd n*
shows *poly* ([:−*a*,*1*:] ^*n*) *x*>*0* ⟷ (*x*>*a*) **using** *assms*
proof −
have *poly* ([:−*a*,*1*:] ^*n*) *x*≥*0* = (*poly* [:− *a*, *1*:] *x* ≥*0*)
unfolding *poly-power* **using** *zero-le-odd-power*[*OF* ‹*odd n*›] **by** *blast*
also have (*poly* [:− *a*, *1*:] *x* ≥*0*) = (*x*≥*a*) **by** *fastforce*
finally have *poly* ([:−*a*,*1*:] ^*n*) *x*≥*0* = (*x*≥*a*) **.**
moreover have *poly* ([:−*a*,*1*:] ^*n*) *x*=*0* = (*x*=*a*) **by**(*rule poly-power-n-eq*, *metis assms even-zero*)
ultimately show *?thesis* **by** *linarith*
qed

lemma *gcd-coprime-poly*:
fixes *p q*::′*a*::{*factorial-ring-gcd*,*semiring-gcd-mult-normalize*} *poly*
assumes *nz*: *p* ≠ *0* ∨ *q* ≠ *0* **and** *p*′: *p* = *p*′ ∗ *gcd p q* **and**
q′: *q* = *q*′ ∗ *gcd p q*
shows *coprime p*′ *q*′
using *gcd-coprime nz p*′ *q*′ **by** *auto*

lemma *poly-mod*:
poly (*p mod q*) *x* = *poly p x* **if** *poly q x* = *0*
proof −
from *that* **have** *poly* (*p mod q*) *x* = *poly* (*p div q* ∗ *q*) *x* + *poly* (*p mod q*) *x*
by *simp*
also have ... = *poly p x*
by (*simp only*: *poly-add* [*symmetric*]) *simp*
finally show *?thesis* **.**
qed

end

1 Sturm-Tarski Theorem

theory *Sturm-Tarski*
  imports *Complex-Main PolyMisc HOL−Computational-Algebra.Field-as-Ring*
begin

2 Misc

lemma *eventually-at-right*:
  fixes *x*::*'a*::{*archimedean-field*,*linorder-topology*}
  shows *eventually P* (*at-right x*) ⟷ (∃ *b*>*x*. ∀ *y*>*x*. *y* < *b* ⟶ *P y*)
proof −
  obtain *y* **where** *y*>*x* **using** *ex-less-of-int* **by** *auto*
  thus *?thesis* **using** *eventually-at-right*[*OF* ‹*y*>*x*›] **by** *auto*
qed

lemma *eventually-at-left*:
  fixes *x*::*'a*::{*archimedean-field*,*linorder-topology*}
  shows *eventually P* (*at-left x*) ⟷ (∃ *b*<*x*. ∀ *y*>*b*. *y* < *x* ⟶ *P y*)
proof −
  obtain *y* **where** *y*<*x*
    using *linordered-field-no-lb* **by** *auto*
  thus *?thesis* **using** *eventually-at-left*[*OF* ‹*y*<*x*›] **by** *auto*
qed

lemma *eventually-neg*:
  assumes *F*≠*bot* **and** *eve*:*eventually* (λ*x*. *P x*) *F*
  shows ¬ *eventually* (λ*x*. ¬ *P x*) *F*
proof (*rule ccontr*)
  assume ¬ ¬ *eventually* (λ*x*. ¬ *P x*) *F*
  hence *eventually* (λ*x*. ¬ *P x*) *F* **by** *auto*
  hence *eventually* (λ*x*. *False*) *F* **using** *eventually-conj*[*OF eve*,*of* (λ*x*. ¬ *P x*)]
by *auto*
  thus *False* **using** ‹*F*≠*bot*› *eventually-False* **by** *auto*
qed

lemma *poly-tendsto*[*simp*]:
    (*poly p* ⟶ *poly p x*) (*at* (*x*::*real*))
    (*poly p* ⟶ *poly p x*) (*at-left* (*x*::*real*))
    (*poly p* ⟶ *poly p x*) (*at-right* (*x*::*real*))
  using *isCont-def*[**where** *f*=*poly p*] **by** (*auto simp add*:*filterlim-at-split*)

lemma *not-eq-pos-or-neg-iff-1*:
  fixes *p*::*real poly*
  shows (∀ *z*. *lb*<*z*∧*z*≤*ub*⟶*poly p z*≠*0*) ⟷
    (∀ *z*. *lb*<*z*∧*z*≤*ub*⟶*poly p z*>*0*)∨(∀ *z*. *lb*<*z*∧*z*≤*ub*⟶*poly p z*<*0*) (**is** *?Q* ⟷
?P)
proof (*rule*,*rule ccontr*)
  assume *?Q* ¬*?P*

```
  then obtain z1 z2 where z1:lb<z1 z1≤ub poly p z1≤0
                      and z2:lb<z2 z2≤ub poly p z2≥0
    by auto
  hence ∃z. lb<z∧z≤ub∧poly p z=0
  proof (cases poly p z1 = 0 ∨ poly p z2 =0 ∨ z1=z2)
    case True
    thus ?thesis using z1 z2 by auto
  next
    case False
    hence poly p z1<0 and poly p z2>0 and z1≠z2 using z1(3) z2(3) by auto
    hence (∃z>z1. z < z2 ∧ poly p z = 0) ∨ (∃z>z2. z < z1 ∧ poly p z = 0)
     using poly-IVT-neg poly-IVT-pos by (subst (asm) linorder-class.neq-iff,auto)

   thus ?thesis using z1(1,2) z2(1,2) by (metis less-eq-real-def order.strict-trans2)
  qed
  thus False using ‹?Q› by auto
next
  assume ?P
  thus ?Q by auto
qed

lemma not-eq-pos-or-neg-iff-2:
  fixes p::real poly
  shows (∀z. lb≤z∧z<ub⟶poly p z≠0)
      ⟷(∀z. lb≤z∧z<ub⟶poly p z>0)∨(∀z. lb≤z∧z<ub⟶poly p z<0) (is
?Q⟷?P)
proof (rule,rule ccontr)
  assume ?Q ¬?P
  then obtain z1 z2 where z1:lb≤z1 z1<ub poly p z1≤0
                      and z2:lb≤z2 z2<ub poly p z2≥0
    by auto
  hence ∃z. lb≤z∧z<ub∧poly p z=0
  proof (cases poly p z1 = 0 ∨ poly p z2 =0 ∨ z1=z2)
    case True
    thus ?thesis using z1 z2 by auto
  next
    case False
    hence poly p z1<0 and poly p z2>0 and z1≠z2 using z1(3) z2(3) by auto
    hence (∃z>z1. z < z2 ∧ poly p z = 0) ∨ (∃z>z2. z < z1 ∧ poly p z = 0)
     using poly-IVT-neg poly-IVT-pos by (subst (asm) linorder-class.neq-iff,auto)

    thus ?thesis using z1(1,2) z2(1,2) by (meson dual-order.strict-trans not-le)
  qed
  thus False using ‹?Q› by auto
next
  assume ?P
  thus ?Q by auto
qed
```

lemma *next-non-root-interval*:
fixes *p*::*real poly*
assumes *p*≠*0*
obtains *ub* **where** *ub*>*lb* **and** (∀ *z*. *lb*<*z*∧*z*≤*ub*⟶*poly p z*≠*0*)
proof (*cases* (∃ *r*. *poly p r*=*0* ∧ *r*>*lb*))
case *False*
thus *?thesis* **by** (*intro that*[*of lb*+*1*],*auto*)
next
case *True*
define *lr* **where** *lr*≡*Min* {*r* . *poly p r*=*0* ∧ *r*>*lb*}
have ∀ *z*. *lb*<*z*∧*z*<*lr*⟶*poly p z*≠*0* **and** *lr*>*lb*
using *True lr-def poly-roots-finite*[*OF* ‹*p*≠*0*›] **by** *auto*
thus *?thesis* **using** *that*[*of* (*lb*+*lr*)/*2*] **by** *auto*
qed

lemma *last-non-root-interval*:
fixes *p*::*real poly*
assumes *p*≠*0*
obtains *lb* **where** *lb*<*ub* **and** (∀ *z*. *lb*≤*z*∧*z*<*ub*⟶*poly p z*≠*0*)
proof (*cases* (∃ *r*. *poly p r*=*0* ∧ *r*<*ub*))
case *False*
thus *?thesis* **by** (*intro that*[*of ub* − *1*]) *auto*
next
case *True*
define *mr* **where** *mr*≡*Max* {*r* . *poly p r*=*0* ∧ *r*<*ub*}
have ∀ *z*. *mr*<*z*∧*z*<*ub*⟶*poly p z*≠*0* **and** *mr*<*ub*
using *True mr-def poly-roots-finite*[*OF* ‹*p*≠*0*›] **by** *auto*
thus *?thesis* **using** *that*[*of* (*mr*+*ub*)/*2*] ‹*mr*<*ub*› **by** *auto*
qed

3 Bound of polynomials

definition *sgn-pos-inf* :: ('*a* ::*linordered-idom*) *poly* ⇒ '*a* **where**
sgn-pos-inf p ≡ *sgn* (*lead-coeff p*)
definition *sgn-neg-inf* :: ('*a* ::*linordered-idom*) *poly* ⇒ '*a* **where**
sgn-neg-inf p ≡ *if even* (*degree p*) *then sgn* (*lead-coeff p*) *else* −*sgn* (*lead-coeff p*)

lemma *sgn-inf-sym*:
fixes *p*::*real poly*
shows *sgn-pos-inf* (*pcompose p* [:*0*,−*1*:]) = *sgn-neg-inf p* (**is** *?L*=*?R*)
proof −
have *?L*= *sgn* (*lead-coeff p* ∗ (− *1*) ^ *degree p*)
unfolding *sgn-pos-inf-def* **by** (*subst lead-coeff-comp*,*auto*)
thus *?thesis* **unfolding** *sgn-neg-inf-def*
by (*metis mult.right-neutral mult-minus1-right neg-one-even-power neg-one-odd-power sgn-minus*)
qed

lemma *poly-pinfty-gt-lc*:

```
  fixes p:: real poly
  assumes  lead-coeff p > 0
  shows ∃ n. ∀ x ≥ n. poly p x ≥ lead-coeff p using assms
proof (induct p)
  case 0
  thus ?case by auto
next
  case (pCons a p)
  have ⟦a≠0;p=0⟧ ⟹ ?case by auto
  moreover have p≠0 ⟹ ?case
  proof −
    assume p≠0
    then obtain n1 where gte-lcoeff:∀ x≥n1. lead-coeff p ≤ poly p x using that
pCons by auto
    have gt-0:lead-coeff p >0 using pCons(3) ‹p≠0› by auto
    define n where n≡max n1 (1+ |a|/(lead-coeff p))
    show ?thesis
    proof (rule-tac x=n in exI,rule,rule)
      fix x assume n ≤ x
      hence lead-coeff p ≤ poly p x
        using gte-lcoeff unfolding n-def by auto
      hence  |a|/(lead-coeff p) ≥ |a|/(poly p x) and poly p x>0 using gt-0
        by (intro frac-le,auto)
      hence x≥1+ |a|/(poly p x) using ‹n≤x›[unfolded n-def] by auto
      thus lead-coeff (pCons a p) ≤ poly (pCons a p) x
        using ‹lead-coeff p ≤ poly p x› ‹poly p x>0› ‹p≠0›
        by (auto simp add:field-simps)
      qed
    qed
  ultimately show ?case by fastforce
qed

lemma poly-sgn-eventually-at-top:
  fixes p::real poly
  shows eventually (λx. sgn (poly p x) = sgn-pos-inf p) at-top
proof (cases p=0)
  case True
  thus ?thesis unfolding sgn-pos-inf-def by auto
next
  case False
  obtain ub where ub:∀ x≥ub. sgn (poly p x) = sgn-pos-inf p
  proof (cases lead-coeff p>0)
    case True
    thus ?thesis
      using that poly-pinfty-gt-lc[of p] unfolding sgn-pos-inf-def by fastforce
  next
    case False
    hence lead-coeff (−p) > 0 and lead-coeff p < 0 unfolding lead-coeff-minus
      using leading-coeff-neq-0[OF ‹p≠0›]
```

```
    by (auto simp add:not-less-iff-gr-or-eq)
  then obtain n where ∀x≥n. lead-coeff p ≥ poly p x
    using poly-pinfty-gt-lc[of −p] unfolding lead-coeff-minus by auto
  thus ?thesis using ‹lead-coeff p<0› that[of n] unfolding sgn-pos-inf-def by
fastforce
 qed
 thus ?thesis unfolding eventually-at-top-linorder by auto
qed

lemma poly-sgn-eventually-at-bot:
 fixes p::real poly
 shows eventually (λx. sgn (poly p x) = sgn-neg-inf p) at-bot
using
 poly-sgn-eventually-at-top[of pcompose p [:0,−1:],unfolded poly-pcompose sgn-inf-sym,simplified]
  eventually-filtermap[of - uminus at-bot::real filter,folded at-top-mirror]
by auto

lemma root-ub:
 fixes p:: real poly
 assumes p≠0
 obtains ub where ∀x. poly p x=0 ⟶ x<ub
  and ∀x≥ub. sgn (poly p x) = sgn-pos-inf p
proof −
 obtain ub1 where ub1:∀x. poly p x=0 ⟶ x<ub1
 proof (cases ∃ r. poly p r=0)
  case False
  thus ?thesis using that by auto
 next
  case True
  define max-r where max-r≡Max {x . poly p x=0}
  hence ∀x. poly p x=0 ⟶ x≤max-r
   using poly-roots-finite[OF ‹p≠0›] True by auto
  thus ?thesis using that[of max-r+1]
   by (metis add.commute add-strict-increasing zero-less-one)
 qed
 obtain ub2 where ub2:∀x≥ub2. sgn (poly p x) = sgn-pos-inf p
  using poly-sgn-eventually-at-top[unfolded eventually-at-top-linorder] by auto
 define ub where ub≡max ub1 ub2
 have ∀x. poly p x=0 ⟶ x<ub using ub1 ub-def
  by (metis eq-iff less-eq-real-def less-linear max.bounded-iff)
 thus ?thesis using that[of ub] ub2 ub-def by auto
qed

lemma root-lb:
 fixes p:: real poly
 assumes p≠0
 obtains lb where ∀x. poly p x=0 ⟶ x>lb
  and ∀x≤lb. sgn (poly p x) = sgn-neg-inf p
proof −
```

obtain *lb1* **where** *lb1*:$\forall x.\ poly\ p\ x{=}0 \longrightarrow x{>}lb1$
proof (*cases* $\exists\ r.\ poly\ p\ r{=}0$)
case *False*
thus *?thesis* **using** *that* **by** *auto*
next
case *True*
define *min-r* **where** *min-r*≡*Min* $\{x\ .\ poly\ p\ x{=}0\}$
hence $\forall x.\ poly\ p\ x{=}0 \longrightarrow x{\geq}min\text{-}r$
using *poly-roots-finite*[*OF* ‹$p{\neq}0$›] *True* **by** *auto*
thus *?thesis* **using** *that*[*of min-r − 1*] **by** (*metis lt-ex order.strict-trans2 that*)

qed
obtain *lb2* **where** *lb2*:$\forall x{\leq}lb2.\ sgn\ (poly\ p\ x) = sgn\text{-}neg\text{-}inf\ p$
using *poly-sgn-eventually-at-bot*[*unfolded eventually-at-bot-linorder*] **by** *auto*
define *lb* **where** *lb*≡*min lb1 lb2*
have $\forall x.\ poly\ p\ x{=}0 \longrightarrow x{>}lb$ **using** *lb1 lb-def*
by (*metis* (*poly-guards-query*) *less-not-sym min-less-iff-conj neq-iff*)
thus *?thesis* **using** *that*[*of lb*] *lb2 lb-def* **by** *auto*
qed

4 Sign

definition *sign*:: *'a*::{*zero*,*linorder*} ⇒ *int* **where**
sign x≡(*if x>0 then 1 else if x=0 then 0 else −1*)

lemma *sign-simps*[*simp*]:
$x{>}0 \Longrightarrow sign\ x{=}1$
$x{=}0 \Longrightarrow sign\ x{=}0$
$x{<}0 \Longrightarrow sign\ x{=}{-}1$
unfolding *sign-def* **by** *auto*

lemma *sign-cases* [*case-names neg zero pos*]:
$(sign\ x = -1 \Longrightarrow P) \Longrightarrow (sign\ x = 0 \Longrightarrow P) \Longrightarrow (sign\ x = 1 \Longrightarrow P) \Longrightarrow P$
unfolding *Sturm-Tarski.sign-def* **by** *argo*

lemma *sign-times*:
fixes *x*::*'a*::*linordered-ring-strict*
shows *sign* $(x{*}y) = sign\ x * sign\ y$
unfolding *Sturm-Tarski.sign-def*
by (*auto simp add*:*zero-less-mult-iff*)

lemma *sign-power*:
fixes *x*::*'a*::*linordered-idom*
shows *sign* $(x\hat{}n) = (if\ n{=}0\ then\ 1\ else\ if\ even\ n\ then\ |sign\ x|\ else\ sign\ x)$
by (*simp add*: *Sturm-Tarski.sign-def zero-less-power-eq*)

lemma *sgn-sign-eq*:
fixes *x*::*'a*::{*linordered-idom*}
shows *sgn x* = *of-int* (*sign x*)

unfolding *sgn-if* **by** *auto*

5 Variation and cross

definition *variation* :: *real* ⇒ *real* ⇒ *int* **where**
variation x y=(*if x*∗*y*≥*0 then 0 else if x*<*y then 1 else* −*1*)

definition *cross* :: *real poly* ⇒ *real* ⇒ *real* ⇒ *int* **where**
cross p a b=*variation* (*poly p a*) (*poly p b*)

lemma *variation-0*[*simp*]: *variation 0 y*=*0 variation x 0*=*0*
unfolding *variation-def* **by** *auto*

lemma *variation-comm*: *variation x y*= − *variation y x* **unfolding** *variation-def*
by (*auto simp add*: *mult.commute*)

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

lemma *variation-cases*:
⟦*x*>*0*;*y*>*0*⟧⟹*variation x y* = *0*
⟦*x*>*0*;*y*<*0*⟧⟹*variation x y* = −*1*
⟦*x*<*0*;*y*>*0*⟧⟹*variation x y* = *1*
⟦*x*<*0*;*y*<*0*⟧⟹*variation x y* = *0*
proof −
show ⟦*x*>*0*;*y*>*0*⟧⟹*variation x y* = *0* **unfolding** *variation-def* **by** *auto*
show ⟦*x*>*0*;*y*<*0*⟧⟹*variation x y* = −*1* **unfolding** *variation-def*
using *mult-pos-neg* **by** *fastforce*
show ⟦*x*<*0*;*y*>*0*⟧⟹*variation x y* = *1* **unfolding** *variation-def*
using *mult-neg-pos* **by** *fastforce*
show ⟦*x*<*0*;*y*<*0*⟧⟹*variation x y* = *0* **unfolding** *variation-def*
using *mult-neg-neg* **by** *fastforce*
qed

lemma *variation-congr*:
assumes *sgn x*=*sgn x*′ *sgn y*=*sgn y*′
shows *variation x y*=*variation x*′ *y*′ **using** *assms*
proof −
have *0* ≤ *x* ∗ *y* = (*0*≤ *x*′ ∗ *y*′) **using** *assms* **by** (*metis Real-Vector-Spaces.sgn-mult zero-le-sgn-iff*)
moreover hence ¬ *0*≤*x* ∗ *y* ⟹ *x* < *y* = (*x*′ < *y*′) **using** *assms*
by (*metis less-eq-real-def mult-nonneg-nonneg mult-nonpos-nonpos not-le order.strict-trans2*
zero-le-sgn-iff)
ultimately show *?thesis* **unfolding** *variation-def* **by** *auto*
qed

lemma *variation-mult-pos*:
assumes *c*>*0*
shows *variation* (*c*∗*x*) *y* =*variation x y* **and** *variation x* (*c*∗*y*) =*variation x y*

proof −
have *sgn* (*c*∗*x*) = *sgn x* **using** ‹*c*>*0*›
by (*simp add*: *Real-Vector-Spaces.sgn-mult*)
thus *variation* (*c*∗*x*) *y* =*variation x y* **using** *variation-congr* **by** *blast*
next
have *sgn* (*c*∗*y*) = *sgn y* **using** ‹*c*>*0*›
by (*simp add*: *Real-Vector-Spaces.sgn-mult*)
thus *variation x* (*c*∗*y*) =*variation x y* **using** *variation-congr* **by** *blast*
qed

lemma *variation-mult-neg-1*:
assumes *c*<*0*
shows *variation* (*c*∗*x*) *y* =*variation x y* + (*if y*=*0 then 0 else sign x*)
apply (*cases x rule*:*linorder-cases*[*of 0*])
apply (*cases y rule*:*linorder-cases*[*of 0*], *auto simp add*:
variation-cases mult-neg-pos[*OF* ‹*c*<*0*›,*of x*] *mult-neg-neg*[*OF* ‹*c*<*0*›,*of x*])+
done

lemma *variation-mult-neg-2*:
assumes *c*<*0*
shows *variation x* (*c*∗*y*) = *variation x y* + (*if x*=*0 then 0 else* − *sign y*)
unfolding *variation-comm*[*of x c*∗*y*, *unfolded variation-mult-neg-1*[*OF* ‹*c*<*0*›, *of y x*]]
by (*auto*,*subst variation-comm*,*simp*)

lemma *cross-no-root*:
assumes *a*<*b* **and** *no-root*:∀ *x*. *a*<*x*∧*x*<*b* ⟶ *poly p x*≠*0*
shows *cross p a b*=*0*
proof −
have ⟦*poly p a*>*0*;*poly p b*<*0*⟧ ⟹ *False* **using** *poly-IVT-neg*[*OF* ‹*a*<*b*›] *no-root*
by *auto*
moreover have ⟦*poly p a*<*0*;*poly p b*>*0*⟧ ⟹ *False* **using** *poly-IVT-pos*[*OF* ‹*a*<*b*›] *no-root* **by** *auto*
ultimately have *0* ≤ *poly p a* ∗ *poly p b*
by (*metis less-eq-real-def linorder-neqE-linordered-idom mult-less-0-iff*)
thus *?thesis* **unfolding** *cross-def variation-def* **by** *simp*
qed

6 Tarski query

definition *taq* :: *'a*::*linordered-idom set* ⇒ *'a poly* ⇒ *int* **where**
taq s q ≡ ∑ *x*∈*s*. *sign* (*poly q x*)

7 Sign at the right

definition *sign-r-pos* :: *real poly* ⇒ *real* ⇒ *bool*
where
sign-r-pos p x≡ (*eventually* (λ*x*. *poly p x*>*0*) (*at-right x*))

lemma *sign-r-pos-rec*:
fixes *p*:: *real poly*
assumes *p≠0*
shows *sign-r-pos p x*= (*if poly p x=0 then sign-r-pos* (*pderiv p*) *x else poly p x>0*
)
proof (*cases poly p x=0*)
case *True*
have *sign-r-pos* (*pderiv p*) *x* ⟹ *sign-r-pos p x*
proof (*rule ccontr*)
assume *sign-r-pos* (*pderiv p*) *x* ¬ *sign-r-pos p x*
obtain *b* **where** *b*>*x* **and** *b*:∀ *z*. *x* < *z* ∧ *z* < *b* ⟶ 0 < *poly* (*pderiv p*) *z*
using ‹*sign-r-pos* (*pderiv p*) *x*› **unfolding** *sign-r-pos-def eventually-at-right*
by *auto*
have ∀ *b*>*x*. ∃ *z*>*x*. *z* < *b* ∧ ¬ 0 < *poly p z* **using** ‹¬ *sign-r-pos p x*›
unfolding *sign-r-pos-def eventually-at-right* **by** *auto*
then obtain *z* **where** *z*>*x* **and** *z*<*b* **and** *poly p z*≤*0*
using ‹*b*>*x*› *b* **by** *auto*
hence ∃ *z'*>*x*. *z'* < *z* ∧ *poly p z* = (*z* − *x*) ∗ *poly* (*pderiv p*) *z'*
using *poly-MVT*[*OF* ‹*z*>*x*›] *True* **by** (*metis diff-0-right*)
hence ∃ *z'*>*x*. *z'* < *z* ∧ *poly* (*pderiv p*) *z'* ≤*0*
using ‹*poly p z*≤*0*›‹*z*>*x*› **by** (*metis leD le-iff-diff-le-0 mult-le-0-iff*)
thus *False* **using** *b* **by** (*metis* ‹*z* < *b*› *dual-order.strict-trans not-le*)
qed
moreover have *sign-r-pos p x* ⟹ *sign-r-pos* (*pderiv p*) *x*
proof −
assume *sign-r-pos p x*
have *pderiv p≠0* **using** ‹*poly p x=0*› ‹*p≠0*›
by (*metis monoid-add-class.add.right-neutral monom-0 monom-eq-0 mult-zero-right*

pderiv-iszero poly-0 poly-pCons)
obtain *ub* **where** *ub*>*x* **and** *ub*: (∀ *z*. *x*<*z*∧*z*<*ub*⟶*poly* (*pderiv p*) *z*>*0*)
∨ (∀ *z*. *x*<*z*∧*z*<*ub*⟶*poly* (*pderiv p*) *z*<*0*)
using *next-non-root-interval*[*OF* ‹*pderiv p≠0*›, *of x*,*unfolded not-eq-pos-or-neg-iff-1*]

by (*metis order.strict-implies-order*)
have ∀ *z*. *x*<*z*∧*z*<*ub*⟶*poly* (*pderiv p*) *z*<*0* ⟹ *False*
proof −
assume *assm*:∀ *z*. *x*<*z*∧*z*<*ub*⟶*poly* (*pderiv p*) *z*<*0*
obtain *ub'* **where** *ub'*>*x* **and** *ub'*:∀ *z*. *x* < *z* ∧ *z* < *ub'* ⟶ 0 < *poly p z*
using ‹*sign-r-pos p x*› **unfolding** *sign-r-pos-def eventually-at-right* **by** *auto*
obtain *z'* **where** *x*<*z'* **and** *z'* < (*x*+(*min ub' ub*))/*2*
and *z'*:*poly p* ((*x*+*min ub' ub*)/*2*) = ((*x*+*min ub' ub*)/*2* − *x*) ∗ *poly*
(*pderiv p*) *z'*
using *poly-MVT*[*of x* (*x*+*min ub' ub*)/*2 p*] ‹*ub'*>*x*› ‹*ub*>*x*› *True* **by** *auto*
moreover have *0* < *poly p* ((*x*+*min ub' ub*)/*2*)
using *ub'*[*THEN HOL.spec*,*of* (*x*+(*min ub' ub*))/*2*] ‹*z'* < (*x*+*min ub' ub*)/*2*›
‹*x*<*z'*›
by *auto*

```
    moreover have (x+min ub' ub)/2 − x>0 using ‹ub'>x› ‹ub>x› by auto
    ultimately have poly (pderiv p) z'>0 by (metis zero-less-mult-pos)
    thus False using assm[THEN spec,of z'] ‹x<z'› ‹z' < (x+(min ub' ub))/2›
by auto
  qed
  hence ∀ z. x<z∧z<ub⟶poly (pderiv p) z>0 using ub by auto
  thus sign-r-pos (pderiv p) x unfolding sign-r-pos-def eventually-at-right
    using ‹ub>x› by auto
 qed
 ultimately show ?thesis using True by auto
next
 case False
 have sign-r-pos p x ⟹ poly p x>0
 proof (rule ccontr)
  assume sign-r-pos p x ¬ 0 < poly p x
  then obtain ub where ub>x and ub: ∀ z. x < z ∧ z < ub ⟶ 0 < poly p z
    unfolding sign-r-pos-def eventually-at-right by auto
  hence poly p ((ub+x)/2) > 0 by auto
  moreover have poly p x<0 using ‹¬ 0 < poly p x› False by auto
  ultimately have ∃ z>x. z < (ub + x) / 2 ∧ poly p z = 0
    using poly-IVT-pos[of x ((ub + x) / 2) p] ‹ub>x› by auto
  thus False using ub by auto
 qed
 moreover have poly p x>0 ⟹ sign-r-pos p x
  unfolding sign-r-pos-def
   using order-tendstoD(1)[OF poly-tendsto(1),of 0 p x] eventually-at-split by
auto
 ultimately show ?thesis using False by auto
qed

lemma sign-r-pos-0[simp]:¬ sign-r-pos 0 (x::real)
 using eventually-False[of at-right x] unfolding sign-r-pos-def by auto

lemma sign-r-pos-minus:
 fixes p:: real poly
 assumes p≠0
 shows sign-r-pos p x = (¬ sign-r-pos (−p) x)
proof −
 have sign-r-pos p x ∨ sign-r-pos (−p) x
  unfolding sign-r-pos-def eventually-at-right
  using next-non-root-interval[OF ‹p≠0›,unfolded not-eq-pos-or-neg-iff-1]
  by (metis (erased, hide-lams) le-less minus-zero neg-less-iff-less poly-minus)
 moreover have sign-r-pos p x ⟹ ¬ sign-r-pos (−p) x unfolding sign-r-pos-def

    using eventually-neg[OF trivial-limit-at-right-real, of λx. poly p x > 0 x]
poly-minus
  by (metis (lifting) eventually-mono less-asym neg-less-0-iff-less)
 ultimately show ?thesis by auto
qed
```

lemma *sign-r-pos-smult*:
fixes *p* :: *real poly*
assumes *c≠0 p≠0*
shows *sign-r-pos* (*smult c p*) *x*= (*if c>0 then sign-r-pos p x else* ¬ *sign-r-pos p x*)
(**is** *?L=?R*)
proof (*cases c>0*)
assume *c>0*
hence ∀ *x*. (*0* < *poly* (*smult c p*) *x*) = (*0* < *poly p x*)
by (*subst poly-smult*,*metis mult-pos-pos zero-less-mult-pos*)
thus *?thesis* **unfolding** *sign-r-pos-def* **using** ‹*c>0*› **by** *auto*
next
assume ¬(*c>0*)
hence ∀ *x*. (*0* < *poly* (*smult c p*) *x*) = (*0* < *poly* (−*p*) *x*)
by (*subst poly-smult*, *metis assms*(*1*) *linorder-neqE-linordered-idom mult-neg-neg mult-zero-right*
neg-0-less-iff-less poly-minus zero-less-mult-pos2)
hence *sign-r-pos* (*smult c p*) *x*=*sign-r-pos* (−*p*) *x*
unfolding *sign-r-pos-def* **using** ‹¬ *c>0*› **by** *auto*
thus *?thesis* **using** *sign-r-pos-minus*[*OF* ‹*p≠0*›, *of x*] ‹¬ *c>0*› **by** *auto*
qed

lemma *sign-r-pos-mult*:
fixes *p q* :: *real poly*
assumes *p≠0 q≠0*
shows *sign-r-pos* (*p*∗*q*) *x*= (*sign-r-pos p x* ⟷ *sign-r-pos q x*)
proof −
obtain *ub* **where** *ub>x*
and *ub*:(∀ *z*. *x* < *z* ∧ *z* < *ub* ⟶ *0* < *poly p z*) ∨ (∀ *z*. *x* < *z* ∧ *z* < *ub* ⟶ *poly p z* < *0*)
using *next-non-root-interval*[*OF* ‹*p≠0*›,*of x*,*unfolded not-eq-pos-or-neg-iff-1*]
by (*metis order.strict-implies-order*)
obtain *ub′* **where** *ub′>x*
and *ub′*:(∀ *z*. *x* < *z* ∧ *z* < *ub′* ⟶ *0* < *poly q z*) ∨ (∀ *z*. *x* < *z* ∧ *z* < *ub′* ⟶ *poly q z* < *0*)
using *next-non-root-interval*[*OF* ‹*q≠0*›,*unfolded not-eq-pos-or-neg-iff-1*]
by (*metis order.strict-implies-order*)
have (∀ *z*. *x* < *z* ∧ *z* < *ub* ⟶ *0* < *poly p z*) ⟹ (∀ *z*. *x* < *z* ∧ *z* < *ub′* ⟶ *0* < *poly q z*) ⟹ *?thesis*
proof −
assume (∀ *z*. *x* < *z* ∧ *z* < *ub* ⟶ *0* < *poly p z*) (∀ *z*. *x* < *z* ∧ *z* < *ub′* ⟶ *0* < *poly q z*)
hence *sign-r-pos p x* **and** *sign-r-pos q x* **unfolding** *sign-r-pos-def eventually-at-right*
using ‹*ub>x*› ‹*ub′>x*› **by** *auto*
moreover hence *eventually* (λ*z*. *poly p z>0* ∧ *poly q z>0*) (*at-right x*)
unfolding *sign-r-pos-def* **using** *eventually-conj-iff*[*of - - at-right x*] **by** *auto*
hence *sign-r-pos* (*p*∗*q*) *x*
unfolding *sign-r-pos-def poly-mult*

by (*metis* (*lifting*, *mono-tags*) *eventually-mono mult-pos-pos*)
ultimately show *?thesis* **by** *auto*
qed
moreover have $(\forall z.\ x < z \wedge z < ub \longrightarrow 0 > poly\ p\ z) \Longrightarrow (\forall z.\ x < z \wedge z < ub' \longrightarrow 0 < poly\ q\ z) \Longrightarrow ?thesis$
proof −
assume $(\forall z.\ x < z \wedge z < ub \longrightarrow 0 > poly\ p\ z)$ $(\forall z.\ x < z \wedge z < ub' \longrightarrow 0 < poly\ q\ z)$
hence *sign-r-pos* $(-p)\ x$ **and** *sign-r-pos* $q\ x$ **unfolding** *sign-r-pos-def eventually-at-right*
using ‹$ub>x$› ‹$ub'>x$› **by** *auto*
moreover hence *eventually* $(\lambda z.\ poly\ (-p)\ z>0 \wedge poly\ q\ z>0)$ (*at-right x*)
unfolding *sign-r-pos-def* **using** *eventually-conj-iff*[*of - - at-right x*] **by** *auto*
hence *sign-r-pos* $(-\ p*q)\ x$
unfolding *sign-r-pos-def poly-mult*
by (*metis* (*lifting*, *mono-tags*) *eventually-mono mult-pos-pos*)
ultimately show *?thesis*
using *sign-r-pos-minus* ‹$p\neq 0$› ‹$q\neq 0$› **by** (*metis minus-mult-left no-zero-divisors*)
qed
moreover have $(\forall z.\ x < z \wedge z < ub \longrightarrow 0 < poly\ p\ z) \Longrightarrow (\forall z.\ x < z \wedge z < ub' \longrightarrow 0 > poly\ q\ z) \Longrightarrow ?thesis$
proof −
assume $(\forall z.\ x < z \wedge z < ub \longrightarrow 0 < poly\ p\ z)$ $(\forall z.\ x < z \wedge z < ub' \longrightarrow 0 > poly\ q\ z)$
hence *sign-r-pos p x* **and** *sign-r-pos* $(-q)\ x$ **unfolding** *sign-r-pos-def eventually-at-right*
using ‹$ub>x$› ‹$ub'>x$› **by** *auto*
moreover hence *eventually* $(\lambda z.\ poly\ p\ z>0 \wedge poly\ (-q)\ z>0)$ (*at-right x*)
unfolding *sign-r-pos-def* **using** *eventually-conj-iff*[*of - - at-right x*] **by** *auto*
hence *sign-r-pos* $(\ p * (-\ q))\ x$
unfolding *sign-r-pos-def poly-mult*
by (*metis* (*lifting*, *mono-tags*) *eventually-mono mult-pos-pos*)
ultimately show *?thesis*
using *sign-r-pos-minus* ‹$p\neq 0$› ‹$q\neq 0$›
by (*metis minus-mult-right no-zero-divisors*)
qed
moreover have $(\forall z.\ x < z \wedge z < ub \longrightarrow 0 > poly\ p\ z) \Longrightarrow (\forall z.\ x < z \wedge z < ub' \longrightarrow 0 > poly\ q\ z) \Longrightarrow ?thesis$
proof −
assume $(\forall z.\ x < z \wedge z < ub \longrightarrow 0 > poly\ p\ z)$ $(\forall z.\ x < z \wedge z < ub' \longrightarrow 0 > poly\ q\ z)$
hence *sign-r-pos* $(-p)\ x$ **and** *sign-r-pos* $(-q)\ x$
unfolding *sign-r-pos-def eventually-at-right* **using** ‹$ub>x$› ‹$ub'>x$› **by** *auto*
moreover hence *eventually* $(\lambda z.\ poly\ (-p)\ z>0 \wedge poly\ (-q)\ z>0)$ (*at-right x*)
unfolding *sign-r-pos-def* **using** *eventually-conj-iff*[*of - - at-right x*] **by** *auto*
hence *sign-r-pos* $(p * q)\ x$
unfolding *sign-r-pos-def poly-mult poly-minus*

```
    apply (elim eventually-mono[of - at-right x])
    by (auto intro:mult-neg-neg)
  ultimately show ?thesis
    using sign-r-pos-minus ⟨p≠0⟩ ⟨q≠0⟩ by metis
 qed
 ultimately show ?thesis using ub ub' by auto
qed

lemma sign-r-pos-add:
 fixes p q :: real poly
 assumes poly p x=0 poly q x≠0
 shows sign-r-pos (p+q) x=sign-r-pos q x
proof (cases poly (p+q) x=0)
 case False
 hence p+q≠0 by (metis poly-0)
 have sign-r-pos (p+q) x = (poly q x > 0)
  using sign-r-pos-rec[OF ⟨p+q≠0⟩] False poly-add ⟨poly p x=0⟩ by auto
 moreover have sign-r-pos q x=(poly q x > 0)
  using sign-r-pos-rec[of q x] ⟨poly q x≠0⟩ poly-0 by force
 ultimately show ?thesis by auto
next
 case True
 hence False using ⟨poly p x=0⟩ ⟨poly q x≠0⟩ poly-add by auto
 thus ?thesis by auto
qed

lemma sign-r-pos-mod:
 fixes p q :: real poly
 assumes poly p x=0 poly q x≠0
 shows sign-r-pos (q mod p) x=sign-r-pos q x
proof −
 have poly (q div p * p) x=0 using ⟨poly p x=0⟩ poly-mult by auto
 moreover hence poly (q mod p) x ≠ 0 using ⟨poly q x≠0⟩
  by (simp add: assms(1) poly-mod)
 ultimately show ?thesis
  by (metis div-mult-mod-eq sign-r-pos-add)
qed

lemma sign-r-pos-pderiv:
 fixes p:: real poly
 assumes poly p x=0 p≠0
 shows sign-r-pos (pderiv p * p) x
proof −
 have pderiv p ≠0
  by (metis assms(1) assms(2) monoid-add-class.add.right-neutral mult-zero-right
pCons-0-0
    pderiv-iszero poly-0 poly-pCons)
 have ?thesis = (sign-r-pos (pderiv p) x ⟷ sign-r-pos p x)
  using sign-r-pos-mult[OF ⟨pderiv p ≠ 0⟩ ⟨p≠0⟩] by auto
```

```
  also have ...=((sign-r-pos (pderiv p) x ⟷ sign-r-pos (pderiv p) x))
    using sign-r-pos-rec[OF ‹p≠0›] ‹poly p x=0› by auto
  finally show ?thesis by auto
qed

lemma sign-r-pos-power:
  fixes p:: real poly and a::real
  shows sign-r-pos ([:−a,1:] ^n) a
proof (induct n)
  case 0
  thus ?case unfolding sign-r-pos-def eventually-at-right by (simp,metis gt-ex)
next
  case (Suc n)
  have pderiv ([:−a,1:] ^Suc n) = smult (Suc n) ([:−a,1:] ^n)
  proof −
    have pderiv [:− a, 1::real:] = 1 by (simp add: pderiv.simps)
    thus ?thesis unfolding pderiv-power-Suc by (metis mult-cancel-left1)
  qed
  moreover have  poly ([:− a, 1:] ^ Suc n) a=0 by (metis old.nat.distinct(2)
poly-power-n-eq)
  hence sign-r-pos ([:− a, 1:] ^ Suc n) a = sign-r-pos (smult (Suc n) ([:−a,1:] ^n))
a
    using sign-r-pos-rec by (metis (erased, hide-lams) calculation pderiv-0)
  hence sign-r-pos ([:− a, 1:] ^ Suc n) a = sign-r-pos  ([:−a,1:] ^n) a
    using sign-r-pos-smult by auto
  ultimately show ?case using Suc.hyps by auto
qed
```

8 Jump

```
definition jump-poly :: real poly ⇒ real poly ⇒real ⇒ int
 where
  jump-poly q p x≡ (if p≠0 ∧ q≠0 ∧ odd((order x p) − (order x q) ) then
                 if sign-r-pos (q∗p) x then 1 else −1
               else 0 )

lemma jump-poly-not-root:poly p x≠0⟹ jump-poly q p x=0
  unfolding  jump-poly-def by (metis even-zero order-root zero-diff)

lemma jump-poly0[simp]:
    jump-poly 0 p x = 0
    jump-poly q 0 x = 0
  unfolding jump-poly-def by auto

lemma jump-poly-smult-1:
  fixes p q::real poly and c::real
  shows jump-poly (smult c q) p x= sign c ∗ jump-poly q p x (is ?L=?R)
proof (cases c=0∨ q=0)
  case True
```

```
  thus ?thesis unfolding jump-poly-def by auto
next
  case False
  hence c≠0 and q≠0 by auto
  thus ?thesis unfolding jump-poly-def
    using order-smult[OF ‹c≠0›] sign-r-pos-smult[OF ‹c≠0›, of q*p x] ‹q≠0›
    by auto
qed

lemma jump-poly-mult:
  fixes p q p'::real poly
  assumes p'≠0
  shows jump-poly (p'*q) (p'*p) x= jump-poly q p x
proof (cases q=0 ∨ p=0)
  case True
  thus ?thesis unfolding jump-poly-def by fastforce
next
  case False
  then have q≠0 p≠0 by auto
  have sign-r-pos (p' * q * (p' * p)) x=sign-r-pos (q * p) x
  proof (unfold sign-r-pos-def,rule eventually-subst,unfold eventually-at-right)
    obtain b where b>x and b:∀ z. x < z ∧ z < b ⟶ poly (p' * p') z >0
    proof (cases ∃ z. poly p' z=0 ∧ z>x)
      case True
      define lr where lr≡Min {r . poly p' r=0 ∧ r>x}
      have ∀ z. x<z∧z<lr⟶poly p' z≠0 and lr>x
        using True lr-def poly-roots-finite[OF ‹p'≠0›] by auto
      hence ∀ z. x < z ∧ z < lr ⟶ 0 < poly (p' * p') z
        by (metis not-real-square-gt-zero poly-mult)
      thus ?thesis using that[OF ‹lr>x›] by auto
    next
      case False
      have ∀ z. x<z∧z<x+1⟶poly p' z≠0 and x+1>x
        using False poly-roots-finite[OF ‹p'≠0›] by auto
      hence ∀ z. x < z ∧ z < x+1 ⟶ 0 < poly (p' * p') z
        by (metis not-real-square-gt-zero poly-mult)
      thus ?thesis using that[OF ‹x+1>x›] by auto
    qed
    show ∃ b>x. ∀ z>x. z < b ⟶ (0 < poly (p' * q * (p' * p)) z) = (0 < poly
(q * p) z)
    proof (rule-tac x=b in exI, rule conjI[OF ‹b>x›],rule allI,rule impI,rule impI)
      fix z assume x < z  z < b
      hence 0<poly (p'*p') z using b by auto
      have  (0 < poly (p' * q * (p' * p)) z)=(0<poly (p'*p') z * poly (q*p) z)
        by (simp add: mult.commute mult.left-commute)
      also have ...=(0<poly (q*p) z)
        using ‹0<poly (p'*p') z› by (metis mult-pos-pos zero-less-mult-pos)
      finally show (0 < poly (p' * q * (p' * p)) z) = (0 < poly (q * p) z) .
    qed
```

```
  qed
  moreover have  odd (order x (p' * p) - order x (p' * q)) = odd (order x p -
order x q)
    using  False ‹p'≠0› ‹p≠0› mult-eq-0-iff order-mult
    by (metis add-diff-cancel-left)
  moreover have  p' * q ≠ 0 ⟷ q ≠ 0
    by (metis ‹p'≠0› mult-eq-0-iff)
  ultimately show jump-poly (p' * q) (p' * p) x = jump-poly q p x unfolding
jump-poly-def by auto
qed

lemma jump-poly-1-mult:
  fixes p1 p2::real poly
  assumes poly p1 x≠0 ∨ poly p2 x≠0
  shows jump-poly 1 (p1*p2) x= sign (poly p2 x) * jump-poly 1 p1 x
          + sign (poly p1 x) * jump-poly 1 p2 x (is ?L=?R)
proof (cases p1=0 ∨ p2 =0)
  case True
  then show ?thesis by auto
next
  case False
  then have p1≠0 p2≠0 p1*p2≠0 by auto
  have ?thesis when poly p1 x≠0
  proof −
    have [simp]:order x p1 = 0 using that order-root by blast
     define simpL where simpL≡(if p2≠0 ∧ odd (order x p2) then if (poly p1
x>0)
    ⟷ sign-r-pos p2 x then 1::int else −1 else 0)
    have ?L=simpL
      unfolding simpL-def jump-poly-def
      using order-mult[OF ‹p1*p2≠0›]
       sign-r-pos-mult[OF ‹p1≠0› ‹p2≠0›] sign-r-pos-rec[OF ‹p1≠0›] ‹poly p1 x≠0›
      by auto
    moreover have poly p1 x>0 ⟹ simpL =?R
       unfolding simpL-def jump-poly-def using jump-poly-not-root[OF ‹poly p1
x≠0›]
      by auto
    moreover have poly p1 x<0 ⟹ simpL =?R
       unfolding simpL-def jump-poly-def using jump-poly-not-root[OF ‹poly p1
x≠0›]
      by auto
   ultimately show ?L=?R using ‹poly p1 x≠0› by (metis linorder-neqE-linordered-idom)
  qed
  moreover have ?thesis when poly p2 x≠0
  proof −
    have [simp]:order x p2 = 0 using that order-root by blast
     define simpL where simpL≡(if p1≠0 ∧ odd (order x p1) then if (poly p2
x>0)
    ⟷ sign-r-pos p1 x then 1::int else −1 else 0)
```

```
    have ?L=simpL
      unfolding simpL-def jump-poly-def
      using order-mult[OF ‹p1*p2≠0›]
       sign-r-pos-mult[OF ‹p1≠0› ‹p2≠0›] sign-r-pos-rec[OF ‹p2≠0›] ‹poly p2 x≠0›
      by auto
    moreover have poly p2 x>0 ⟹ simpL =?R
       unfolding simpL-def jump-poly-def using jump-poly-not-root[OF ‹poly p2
x≠0›]
      by auto
    moreover have poly p2 x<0 ⟹ simpL =?R
       unfolding simpL-def jump-poly-def using jump-poly-not-root[OF ‹poly p2
x≠0›]
      by auto
   ultimately show ?L=?R using ‹poly p2 x≠0› by (metis linorder-neqE-linordered-idom)
  qed
  ultimately show ?thesis using assms by auto
qed

lemma jump-poly-mod:
  fixes p q::real poly
  shows jump-poly q p x= jump-poly (q mod p) p x
proof (cases q=0 ∨ p=0)
  case True
  thus ?thesis by fastforce
next
  case False
  then have p≠0 q≠0 by auto
  define n where n≡min (order x q) (order x p)
  obtain q' where q':q=[:−x,1:] ^n * q'
    using n-def  power-le-dvd[OF order-1[of x q], of n]
    by (metis dvdE min.cobounded2 min.commute)
  obtain p' where p':p=[:−x,1:] ^n * p'
    using n-def  power-le-dvd[OF order-1[of x p], of n]
    by (metis dvdE min.cobounded2)
  have q'≠0 and p'≠0 using q' p' ‹p≠0› ‹q≠0› by auto
  have order x q'=0 ∨ order x p'=0
  proof (rule ccontr)
    assume ¬ (order x q' = 0 ∨ order x p' = 0)
    hence order x q' > 0 and order x p' > 0 by auto
    hence order x q>n and order x p>n unfolding q' p'
      using order-mult[OF ‹q≠0›[unfolded q'],of x] order-mult[OF ‹p≠0›[unfolded
p'],of x]
          order-power-n-n[of x n]
      by auto
    thus False using n-def by auto
  qed
  have cond:q' ≠ 0 ∧ odd (order x p' − order x q')
    = (q' mod p' ≠0 ∧ odd(order x p' − order x (q' mod p')))
  proof (cases order x p'=0)
```

```
    case True
    thus ?thesis by (metis ⟨q' ≠ 0⟩ even-zero zero-diff)
  next
    case False
    hence order x q'=0 using ⟨order x q'=0 ∨ order x p'=0⟩ by auto
    hence ¬ [:−x,1:] dvd q'
      by (metis ⟨q' ≠ 0⟩ order-root poly-eq-0-iff-dvd)
    moreover have [:−x,1:] dvd p' using False
      by (metis order-root poly-eq-0-iff-dvd)
    ultimately have ¬ [:−x,1:] dvd (q' mod p')
      by (metis dvd-mod-iff)
    hence order x (q' mod p') = 0 and q' mod p' ≠0
       apply (metis order-root poly-eq-0-iff-dvd)
      by (metis ⟨¬ [:− x, 1:] dvd q' mod p'⟩ dvd-0-right)
    thus ?thesis using ⟨order x q'=0⟩ by auto
  qed
  moreover have q' mod p'≠0 ⟹ poly p' x = 0
     ⟹ sign-r-pos (q' * p') x= sign-r-pos (q' mod p' * p') x
  proof −
    assume q' mod p'≠0 poly p' x = 0
    hence poly q' x≠0 using ⟨order x q'=0 ∨ order x p'=0⟩
      by (metis ⟨p' ≠ 0⟩ ⟨q' ≠ 0⟩ order-root)
    hence sign-r-pos q' x= sign-r-pos (q' mod p') x
      using sign-r-pos-mod[OF ⟨poly p' x=0⟩] by auto
    thus ?thesis
       unfolding sign-r-pos-mult[OF ⟨q'≠0⟩ ⟨p'≠0⟩] sign-r-pos-mult[OF ⟨q' mod
p'≠0⟩ ⟨p'≠0⟩]
      by auto
  qed
  moreover have q' mod p' = 0 ∨ poly p' x ≠ 0 ⟹ jump-poly q' p' x = jump-poly
(q' mod p') p' x
  proof −
    assume assm:q' mod p' = 0 ∨ poly p' x ≠ 0
    have q' mod p' = 0 ⟹ ?thesis unfolding jump-poly-def
      using cond by auto
    moreover have poly p' x ≠ 0
       ⟹ ¬ odd (order x p' − order x q') ∧ ¬ odd(order x p' − order x (q' mod
p'))
      by (metis even-zero order-root zero-diff)
    hence poly p' x ≠ 0 ⟹ ?thesis unfolding jump-poly-def by auto
    ultimately show ?thesis using assm by auto
  qed
  ultimately have jump-poly q' p' x = jump-poly (q' mod p') p' x unfolding
jump-poly-def by force
  thus ?thesis using p' q' jump-poly-mult by auto
qed

lemma jump-poly-coprime:
  fixes p q:: real poly
```

assumes *poly p x=0 coprime p q*
shows *jump-poly q p x= jump-poly 1 (q∗p) x*
proof (*cases p=0 ∨ q=0*)
case *True*
then show *?thesis* **by** *auto*
next
case *False*
then have *p≠0 q≠0* **by** *auto*
then have *poly p x≠0 ∨ poly q x≠0* **using** *coprime-poly-0*[*OF* ‹*coprime p q*›]
by *auto*
then have *poly q x≠0* **using** ‹*poly p x=0*› **by** *auto*
then have *order x q=0* **using** *order-root* **by** *blast*
then have *order x p − order x q = order x (q ∗ p)*
using ‹*p≠0*› ‹*q≠0*› *order-mult* [*of q p x*] **by** *auto*
then show *?thesis* **unfolding** *jump-poly-def* **using** ‹*q≠0*› **by** *auto*
qed

lemma *jump-poly-sgn*:
fixes *p q*:: *real poly*
assumes *p≠0 poly p x=0*
shows *jump-poly (pderiv p ∗ q) p x = sign (poly q x)*
proof (*cases q=0*)
case *True*
thus *?thesis* **by** *auto*
next
case *False*
have *pderiv p≠0* **using** ‹*p≠0*› ‹*poly p x=0*›
by (*metis mult-poly-0-left sign-r-pos-0 sign-r-pos-pderiv*)
have *elim-p-order*: *order x p − order x (pderiv p ∗ q)=1 − order x q*
proof −
have *order x p − order x (pderiv p ∗ q) = order x p − order x (pderiv p) −*
order x q
using *order-mult* ‹*pderiv p≠0*› *False* **by** (*metis diff-diff-left mult-eq-0-iff*)
moreover have *order x p − order x (pderiv p) = 1*
using *order-pderiv*[*OF* ‹*pderiv p≠0*›, *of x*] ‹*poly p x=0*› *order-root*[*of p x*]
‹*p≠0*› **by** *auto*
ultimately show *?thesis* **by** *auto*
qed
have *elim-p-sign-r-pos*:*sign-r-pos (pderiv p ∗ q ∗ p) x=sign-r-pos q x*
proof −
have *sign-r-pos (pderiv p ∗ q ∗ p) x = (sign-r-pos (pderiv p∗ p) x ⟷ sign-r-pos*
q x)
by (*metis* ‹*q ≠ 0*› ‹*pderiv p ≠ 0*› *assms*(*1*) *no-zero-divisors sign-r-pos-mult*)
thus *?thesis* **using** *sign-r-pos-pderiv*[*OF* ‹*poly p x=0*› ‹*p≠0*›] **by** *auto*
qed
define *simpleL* **where** *simpleL≡if pderiv p ∗ q ≠ 0 ∧ odd (1 − order x q) then*
if sign-r-pos q x then 1::*int else − 1 else 0*
have *jump-poly (pderiv p ∗ q) p x =simpleL*
unfolding *simpleL-def jump-poly-def* **by** (*subst elim-p-order*, *subst elim-p-sign-r-pos*,*simp*)

moreover have *poly q x=0 ⟹ simpleL=sign (poly q x)*
proof −
assume *poly q x=0*
hence *1−order x q = 0* **using** ‹*q≠0*› **by** (*metis less-one not-gr0 order-root zero-less-diff*)
hence *simpleL=0* **unfolding** *simpleL-def* **by** *auto*
moreover have *sign (poly q x)=0* **using** ‹*poly q x=0*› **by** *auto*
ultimately show *?thesis* **by** *auto*
qed
moreover have *poly q x≠0⟹ simpleL=sign (poly q x)*
proof −
assume *poly q x≠0*
hence *odd (1 − order x q)* **by** (*simp add*: *order-root*)
moreover have *pderiv p ∗ q ≠ 0* **by** (*metis False* ‹*pderiv p ≠ 0*› *no-zero-divisors*)
moreover have *sign-r-pos q x = (poly q x > 0)*
using *sign-r-pos-rec*[*OF False*] ‹*poly q x≠0*› **by** *auto*
ultimately have *simpleL=(if poly q x>0 then 1 else − 1)* **unfolding** *simpleL-def* **by** *auto*
thus *?thesis* **using** ‹*poly q x≠0*› **by** *auto*
qed
ultimately show *?thesis* **by** *force*
qed

9 Cauchy index

definition *cindex-poly*:: *real ⇒ real ⇒ real poly ⇒ real poly ⇒ int*
where
cindex-poly a b q p≡ (∑ *x*∈{*x*. *poly p x=0 ∧ a< x∧ x< b*}. *jump-poly q p x*)

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

lemma *cindex-poly-cross*:
fixes *p*::*real poly* **and** *a b*::*real*
assumes *a<b poly p a≠0 poly p b≠0*
shows *cindex-poly a b 1 p = cross p a b*
using ‹*poly p a≠0*› ‹*poly p b≠0*›
proof (*cases* {*x*. *poly p x=0 ∧ a< x∧ x< b*}≠{}, *induct degree p arbitrary*:*p rule*:*nat-less-induct*)
case *1*
then have *p≠0* **by** *force*
define *roots* **where** *roots*≡{*x*. *poly p x=0 ∧ a< x∧ x< b*}
have *finite roots* **unfolding** *roots-def* **using** *poly-roots-finite*[*OF* ‹*p≠0*›] **by** *auto*
define *max-r* **where** *max-r*≡*Max roots*
hence *poly p max-r=0* **and** *a<max-r* **and** *max-r<b*
using *Max-in*[*OF* ‹*finite roots*›] *1.prems* **unfolding** *roots-def* **by** *auto*
define *max-rp* **where** *max-rp*≡[:−*max-r*,*1*:] ^*order max-r p*
then obtain *p′* **where** *p′*:*p=p′∗max-rp* **and** *not-dvd*:¬ [:−*max-r*,*1*:] *dvd p′*
by (*metis* ‹*p≠0*› *mult.commute order-decomp*)

```
  hence p'≠0 and max-rp≠0 and poly p' a≠0 and poly p' b≠0
     and  poly max-rp a≠0 and poly max-rp b≠0
    using ‹p≠0› ‹poly p a≠0› ‹poly p b≠0›  by auto
  define max-r-sign where max-r-sign≡if odd(order max-r p) then −1 else 1::int
  define roots' where roots'≡{x.  a< x∧ x< b ∧ poly p' x=0}
  have (∑ x∈roots. jump-poly 1 p x)= (∑ x∈roots'. jump-poly 1 p x)+ jump-poly
1 p max-r
  proof −
    have roots=roots' ∪ {x.  a< x∧ x< b ∧ poly max-rp x=0 }
      unfolding roots-def roots'-def p' by auto
    moreover have {x.  a < x ∧ x < b ∧  poly max-rp x = 0 }={max-r}
      unfolding max-rp-def using ‹poly p max-r=0›
     by (auto simp add: ‹a<max-r› ‹max-r<b›,metis 1.prems(1) neq0-conv order-root)
    moreover hence roots' ∩ {x. a< x∧ x< b ∧ poly max-rp x=0} ={}
      unfolding roots'-def  using ‹¬ [:−max-r,1:] dvd p'›
     by (metis (mono-tags) Int-insert-right-if0 inf-bot-right mem-Collect-eq poly-eq-0-iff-dvd)
    moreover have finite roots'
     using  p' ‹p≠0› by (metis ‹finite roots› calculation(1) calculation(2) finite-Un)
    ultimately show ?thesis using sum.union-disjoint by auto
  qed
  moreover have (∑ x∈roots'. jump-poly 1 p x) = max-r-sign ∗ cross p' a b
  proof −
    have (∑ x∈roots'. jump-poly 1 p x) = (∑ x∈roots'. max-r-sign ∗ jump-poly 1
p' x)
    proof (rule sum.cong,rule refl)
      fix x assume x ∈ roots'
      hence x≠max-r using not-dvd unfolding roots'-def
        by (metis (mono-tags, lifting) mem-Collect-eq poly-eq-0-iff-dvd )
       hence poly max-rp x≠0 using poly-power-n-eq unfolding max-rp-def by
auto
      hence order x max-rp=0  by (metis order-root)
      moreover have jump-poly 1 max-rp x=0
        using ‹poly max-rp x≠0› by (metis jump-poly-not-root)
      moreover have x∈roots
        using ‹x ∈ roots'› unfolding roots-def roots'-def p' by auto
      hence x<max-r
        using Max-ge[OF ‹finite roots›,of x] ‹x≠max-r› by (fold max-r-def,auto)
      hence sign (poly max-rp x) = max-r-sign
        using ‹poly max-rp x ≠ 0› unfolding max-r-sign-def max-rp-def sign-def
        by (subst poly-power,simp add:linorder-class.not-less zero-less-power-eq)
      ultimately show jump-poly 1 p x = max-r-sign ∗ jump-poly 1 p' x
        using jump-poly-1-mult[of p' x max-rp]  unfolding p'
        by (simp add: ‹poly max-rp x ≠ 0›)
      qed
      also have ... = max-r-sign ∗ (∑ x∈roots'. jump-poly 1 p' x)
         by (simp add: sum-distrib-left)
      also have ... = max-r-sign ∗ cross p' a b
      proof (cases roots'={})
        case True
```

```
      hence cross p' a b=0 unfolding roots'-def using cross-no-root[OF ‹a<b›]
by auto
      thus ?thesis using True by simp
    next
      case False
      moreover have degree max-rp≠0
        unfolding max-rp-def degree-linear-power
        by (metis 1.prems(1) ‹poly p max-r = 0› order-root)
         hence  degree p' < degree p unfolding p' degree-mult-eq[OF ‹p'≠0›
‹max-rp≠0›]
        by auto
      ultimately have cindex-poly a b 1 p' = cross p' a b
        unfolding roots'-def
          using 1.hyps[rule-format,of degree p' p'] ‹p'≠0› ‹poly p' a≠0› ‹poly p'
b≠0›
        by auto
      moreover have cindex-poly a b 1 p' = sum (jump-poly 1 p') roots'
        unfolding cindex-poly-def roots'-def
        apply simp
        by (metis (no-types, lifting) )
      ultimately show ?thesis by auto
    qed
    finally show ?thesis .
  qed
  moreover have max-r-sign * cross p' a b + jump-poly 1 p max-r = cross p a
b (is ?L=?R)
  proof (cases odd (order max-r p))
    case True
    have poly max-rp a < 0
      using poly-power-n-odd[OF True,of max-r a] ‹poly max-rp a≠0› ‹max-r>a›

      unfolding max-rp-def by linarith
    moreover have poly max-rp b>0
      using poly-power-n-odd[OF True,of max-r b] ‹max-r<b›
      unfolding max-rp-def by linarith
    ultimately have ?R=cross p' a b + sign (poly p' a)
      unfolding p' cross-def poly-mult
      using variation-mult-neg-1[of poly max-rp a, simplified mult.commute]
        variation-mult-pos(2)[of poly max-rp b, simplified mult.commute] ‹poly p'
b≠0›
      by auto
    moreover have ?L=- cross p' a b + sign (poly p' b)
    proof -
      have  sign-r-pos p' max-r = (poly p' max-r >0)
        using sign-r-pos-rec[OF ‹p'≠0›] not-dvd by (metis poly-eq-0-iff-dvd)
      moreover have (poly p' max-r>0) = (poly p' b>0)
      proof (rule ccontr)
        assume (0 < poly p' max-r) ≠ (0 < poly p' b)
        hence poly p' max-r * poly p' b <0
```

```
        using ‹poly p' b≠0› not-dvd[folded poly-eq-0-iff-dvd]
      by (metis (poly-guards-query) linorder-neqE-linordered-idom mult-less-0-iff)
       then obtain r where r>max-r and r<b and poly p' r=0
        using poly-IVT[OF ‹max-r<b›] by auto
       hence r∈roots unfolding roots-def p' using ‹max-r>a› by auto
        thus False using ‹r>max-r› Max-ge[OF ‹finite roots›,of r] unfolding
max-r-def by auto
      qed
      moreover have sign-r-pos max-rp max-r
        using sign-r-pos-power unfolding max-rp-def by auto
      ultimately show ?thesis
        using True ‹poly p' b≠0› ‹max-rp≠0› ‹p'≠0› sign-r-pos-mult[OF ‹p'≠0›
‹max-rp≠0›]
       unfolding max-r-sign-def  p' jump-poly-def
       by simp
    qed
    moreover have variation (poly p' a) (poly p' b) + sign (poly p' a)
      = − variation (poly p' a) (poly p' b) + sign (poly p' b) unfolding cross-def
    by (cases poly p' b rule:linorder-cases[of 0], (cases poly p' a rule:linorder-cases[of
0],
       auto simp add:variation-cases ‹poly p' a ≠ 0› ‹poly p' b ≠ 0›)+)
    ultimately show ?thesis unfolding cross-def by auto
  next
    case False
    hence poly max-rp a > 0 and poly max-rp b > 0
     unfolding max-rp-def poly-power
     using ‹poly max-rp a≠0› ‹poly max-rp b ≠ 0›  1.prems(1−2) ‹poly p max-r
= 0›
      apply (unfold zero-less-power-eq)
      by auto
    moreover have poly max-rp b > 0
     unfolding max-rp-def poly-power
     using ‹poly max-rp b ≠ 0› False max-rp-def poly-power
       zero-le-even-power[of order max-r p b − max-r]
     by (auto simp add: le-less)
    ultimately have ?R=cross p' a b
     apply (simp only: p' mult.commute cross-def) using variation-mult-pos
     by auto
    thus ?thesis unfolding max-r-sign-def jump-poly-def using False by auto
  qed
  ultimately have sum (jump-poly 1 p) roots = cross p a b  by auto
  then show ?case unfolding roots-def cindex-poly-def by simp
next
 case False
 hence cross p a b=0 using cross-no-root[OF ‹a<b›] by auto
 thus ?thesis using False unfolding cindex-poly-def by (metis sum.empty)
qed

lemma cindex-poly-mult:
```

fixes $p\ q\ p'$::*real poly*
assumes $p' \neq 0$
shows *cindex-poly a b* $(p' * q\ )\ (p' * p) =$ *cindex-poly a b q p*
proof (*cases* $p=0$)
case *True*
then show *?thesis* **by** *auto*
next
case *False*
show *?thesis* **unfolding** *cindex-poly-def*
apply (*rule sum.mono-neutral-cong-right*)
subgoal using ‹$p\neq 0$› ‹$p'\neq 0$› **by** (*simp add*: *poly-roots-finite*)
subgoal by *auto*
subgoal using *jump-poly-mult jump-poly-not-root assms* **by** *fastforce*
subgoal for x **using** *jump-poly-mult*[*OF* ‹$p'\neq 0$›] **by** *auto*
done
qed

lemma *cindex-poly-smult-1*:
fixes $p\ q$::*real poly* **and** c::*real*
shows *cindex-poly a b* (*smult c q*) $p =$ (*sign c*) $*$ *cindex-poly a b q p*
unfolding *cindex-poly-def*
using *sum-distrib-left*[*THEN sym*, *of sign c* $\lambda x.$ *jump-poly q p x*
$\{x.$ *poly p x* = (*0*::*real*) $\wedge$ $a < x \wedge x < b\}$] *jump-poly-smult-1*
by *auto*

lemma *cindex-poly-mod*:
fixes $p\ q$::*real poly*
shows *cindex-poly a b q p* = *cindex-poly a b* (*q mod p*) *p*
unfolding *cindex-poly-def* **using** *jump-poly-mod* **by** *auto*

lemma *cindex-poly-inverse-add*:
fixes $p\ q$::*real poly*
assumes *coprime p q*
shows *cindex-poly a b q p* + *cindex-poly a b p q*=*cindex-poly a b 1* (*q*$*$*p*)
(**is** *?L*=*?R*)
proof (*cases* $p=0 \vee q=0$)
case *True*
then show *?thesis* **by** *auto*
next
case *False*
then have $p\neq 0\ q\neq 0$ **by** *auto*
define *A* **where** $A \equiv \{x.$ *poly p x* = $0 \wedge a < x \wedge x < b\}$
define *B* **where** $B \equiv \{x.$ *poly q x* = $0 \wedge a < x \wedge x < b\}$
have *?L* = *sum* ($\lambda x.$ *jump-poly 1* (*q*$*$*p*) *x*) *A* + *sum* ($\lambda x.$ *jump-poly 1* (*q*$*$*p*) *x*)
B
proof −
have *cindex-poly a b q p* = *sum* ($\lambda x.$ *jump-poly 1* (*q*$*$*p*) *x*) *A* **unfolding** *A-def*
cindex-poly-def
using *jump-poly-coprime*[*OF* - ‹*coprime p q*›] **by** *auto*

```
    moreover have coprime q p using ‹coprime p q›
      by (simp add: ac-simps)
    hence cindex-poly a b p q = sum (λx. jump-poly 1 (q*p) x) B unfolding B-def
cindex-poly-def
      using jump-poly-coprime [of q - p] by (auto simp add: ac-simps)
    ultimately show ?thesis by auto
  qed
   moreover have A ∪ B= {x. poly (q*p) x=0 ∧ a<x ∧ x<b } unfolding
poly-mult A-def B-def by auto
  moreover have A ∩ B={}
  proof (rule ccontr)
    assume A ∩ B≠{}
    then obtain x where x∈A and x∈B by auto
    hence poly p x=0 and poly q x=0 unfolding A-def B-def by auto
    hence gcd p q≠1 by (metis poly-1 poly-eq-0-iff-dvd gcd-greatest zero-neq-one)
    thus False using ‹coprime p q› by auto
  qed
  moreover have finite A and finite B
    unfolding A-def B-def using poly-roots-finite ‹p≠0› ‹q≠0› by fast+
  ultimately have cindex-poly a b q p + cindex-poly a b p q
      = sum (jump-poly 1 (q * p)) {x. poly (q*p) x=0 ∧ a<x ∧ x<b}
    using sum.union-disjoint by metis
  then show ?thesis unfolding cindex-poly-def by auto
qed

lemma cindex-poly-inverse-add-cross:
  fixes p q::real poly
  assumes a < b poly (p * q) a ≠0 poly (p * q) b ≠0
  shows cindex-poly a b q p + cindex-poly a b p q = cross (p * q) a b (is ?L=?R)
proof −
  have p≠0 and q≠0 using ‹poly (p * q) a ≠0› by auto
  define g where g≡gcd p q
  obtain p' q' where p':p= p'*g and q':q=q'*g
    using gcd-dvd1 gcd-dvd2 dvd-def[of gcd p q, simplified mult.commute] g-def by
metis
  hence coprime p' q' using gcd-coprime ‹p≠0› unfolding g-def by auto
  have p'≠0 q'≠0 g ≠0 using p' q' ‹p≠0› ‹q≠0› by auto
  have ?L=cindex-poly a b q' p' + cindex-poly a b p' q'
    apply (simp only: p' q' mult.commute)
    using cindex-poly-mult[OF ‹g≠0›] cindex-poly-mult[OF ‹g≠0›]
    by auto
  also have ... = cindex-poly a b 1 (q' * p')
    using  cindex-poly-inverse-add[OF ‹coprime p' q'›, of a b] .
  also have ... = cross (p' * q') a b
    using cindex-poly-cross[OF ‹a<b›, of q'*p'] ‹p'≠0› ‹q'≠0›
      ‹poly (p * q) a ≠0› ‹poly (p * q) b ≠0›
    unfolding p' q'
    apply (subst (2) mult.commute)
    by auto
```

```
  also have ... = ?R
  proof −
    have poly (p ∗ q) a = poly (g∗g) a ∗ poly (p′ ∗ q′) a
       and poly (p ∗ q) b = poly (g∗g) b ∗ poly (p′ ∗ q′) b
      unfolding p′ q′ by auto
    moreover have poly g a≠0 using ‹poly (p ∗ q) a ≠0›
      unfolding p′ by auto
    hence poly (g∗g) a>0
      by (metis (poly-guards-query) not-real-square-gt-zero poly-mult)
    moreover have poly g b≠0 using ‹poly (p ∗ q) b ≠0›
      unfolding p′ by auto
     hence poly (g∗g) b>0 by (metis (poly-guards-query) not-real-square-gt-zero
poly-mult)
    ultimately show ?thesis
      unfolding cross-def using variation-mult-pos by auto
  qed
  finally show ?L = ?R .
qed

lemma cindex-poly-rec:
  fixes p q::real poly
  assumes a < b poly (p ∗ q) a ≠0 poly (p ∗ q) b ≠0
  shows cindex-poly a b q p  = cross (p ∗ q) a b  +  cindex-poly a b (− (p mod
q)) q (is ?L=?R)
proof −
  have q≠0 using ‹poly (p ∗ q) a ≠0› by auto
  note cindex-poly-inverse-add-cross[OF assms]
  moreover have − cindex-poly a b p q = cindex-poly a b (− (p mod q)) q
    using cindex-poly-mod cindex-poly-smult-1[of a b −1]
    by auto
  ultimately show ?thesis by auto
qed

lemma cindex-poly-congr:
  fixes p q:: real poly
  assumes a<a′ a′<b′ b′<b
  assumes ∀ x. ((a<x∧x≤a′) ∨ (b′≤x ∧ x<b)) ⟶ poly p x ≠0
  shows cindex-poly a b q p=cindex-poly a′ b′ q p
proof (cases p=0)
  case True
  then show ?thesis by auto
next
  case False
  show ?thesis unfolding cindex-poly-def
    apply (rule sum.mono-neutral-right)
    subgoal using poly-roots-finite[OF ‹p≠0›] by auto
    subgoal using assms by auto
    subgoal using assms(4) by fastforce
    done
```

qed

lemma *greaterThanLessThan-unfold*:$\{a<..<b\} = \{x.\ a<x \wedge x<b\}$
by *fastforce*

lemma *cindex-poly-taq*:
fixes *p q*::*real poly*
shows *taq* $\{x.\ poly\ p\ x = 0 \wedge a < x \wedge x < b\}$ *q*=*cindex-poly a b* (*pderiv p* $*$ *q*) *p*
proof (*cases p=0*)
case *True*
define *S* **where** *S*=$\{x.\ poly\ p\ x = 0 \wedge a < x \wedge x < b\}$
have *?thesis* **when** $a \geq b$
proof −
have $S = \{\}$ **using** *that* **unfolding** *S-def* **by** *auto*
then show *?thesis* **using** *True* **unfolding** *taq-def* **by** (*fold S-def*,*simp*)
qed
moreover have *?thesis* **when** $a<b$
proof −
have *infinite* $\{x.\ a<x \wedge x<b\}$ **using** *infinite-Ioo*[*OF* ‹$a<b$›]
unfolding *greaterThanLessThan-unfold* **by** *simp*
then have *infinite S* **unfolding** *S-def* **using** *True* **by** *auto*
then show *?thesis* **using** *True* **unfolding** *taq-def* **by** (*fold S-def*,*simp*)
qed
ultimately show *?thesis* **by** *fastforce*
next
case *False*
show *?thesis*
unfolding *cindex-poly-def taq-def*
by (*rule sum.cong*,*auto simp add*:*jump-poly-sgn*[*OF* ‹$p\neq 0$›])
qed

10 Signed remainder sequence

function *smods*:: *real poly* ⇒ *real poly* ⇒ (*real poly*) *list* **where**
smods p q= (*if p=0 then* [] *else Cons p* (*smods q* (−(*p mod q*))))
by *auto*
termination
apply (*relation measure* (λ(*p*,*q*).*if p=0 then 0 else if q=0 then 1 else 2+degree q*))
apply *simp-all*
apply (*metis degree-mod-less*)
done

lemma *smods-nil-eq*:*smods p q* = [] ⟷ (*p=0*) **by** *auto*
lemma *smods-singleton*:[*x*] = *smods p q* ⟹ ($p\neq 0 \wedge q=0 \wedge x=p$)
by (*metis list.discI list.inject smods.elims*)

lemma *smods-0*[*simp*]:

```
  smods 0 q = []
  smods p 0 = (if p=0 then [] else [p])
by auto

lemma no-0-in-smods: 0∉set (smods p q)
  apply (induct smods p q arbitrary:p q)
  by (simp,metis list.inject neq-Nil-conv set-ConsD smods.elims)

fun changes:: ('a ::linordered-idom) list ⇒ int where
  changes [] = 0|
  changes [-] = 0 |
  changes (x1#x2#xs) = (if x1*x2<0 then 1+changes (x2#xs)
                        else if x2=0 then changes (x1#xs)
                        else changes (x2#xs))

lemma changes-map-sgn-eq:
  changes xs = changes (map sgn xs)
proof (induct xs rule:changes.induct)
  case 1
  show ?case by simp
next
  case 2
  show ?case by simp
next
  case (3 x1 x2 xs)
  moreover have x1*x2<0 ⟷ sgn x1 * sgn x2 < 0
    by (unfold mult-less-0-iff sgn-less sgn-greater,simp)
  moreover have x2=0 ⟷ sgn x2 =0 by (rule sgn-0-0[symmetric])
  ultimately show ?case by auto
qed

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

definition changes-poly-pos-inf:: ('a ::linordered-idom) poly list ⇒ int where
  changes-poly-pos-inf ps = changes (map sgn-pos-inf ps)

definition changes-poly-neg-inf:: ('a ::linordered-idom) poly list ⇒ int where
  changes-poly-neg-inf ps = changes (map sgn-neg-inf ps)

lemma changes-poly-at-0[simp]:
  changes-poly-at [] a =0
  changes-poly-at [p] a=0
unfolding changes-poly-at-def by auto

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

definition *changes-gt-smods*:: *real* ⇒*real poly* ⇒ *real poly* ⇒ *int* **where**
changes-gt-smods a p q= (*let ps*= *smods p q in changes-poly-at ps a* − *changes-poly-pos-inf ps*)

definition *changes-le-smods*:: *real* ⇒*real poly* ⇒ *real poly* ⇒ *int* **where**
changes-le-smods b p q= (*let ps*= *smods p q in changes-poly-neg-inf ps* − *changes-poly-at ps b*)

definition *changes-R-smods*:: *real poly* ⇒ *real poly* ⇒ *int* **where**
changes-R-smods p q= (*let ps*= *smods p q in changes-poly-neg-inf ps* − *changes-poly-pos-inf ps*)

lemma *changes-R-smods-0*[*simp*]:
changes-R-smods 0 q = *0*
changes-R-smods p 0 = *0*
unfolding *changes-R-smods-def changes-poly-neg-inf-def changes-poly-pos-inf-def*
by *auto*

lemma *changes-itv-smods-0*[*simp*]:
changes-itv-smods a b 0 q = *0*
changes-itv-smods a b p 0 = *0*
unfolding *changes-itv-smods-def*
by *auto*

lemma *changes-itv-smods-rec*:
assumes *a*<*b poly* (*p*∗*q*) *a*≠*0 poly* (*p*∗*q*) *b*≠*0*
shows *changes-itv-smods a b p q* = *cross* (*p*∗*q*) *a b* + *changes-itv-smods a b q* (−(*p mod q*))
proof (*cases p*=*0* ∨ *q*=*0* ∨ *p mod q* = *0*)
case *True*
moreover have *p*=*0* ∨ *q*=*0* ⟹ *?thesis*
unfolding *changes-itv-smods-def changes-poly-at-def* **by** (*erule HOL.disjE*,*auto*)
moreover have *p mod q* = *0* ⟹ *?thesis*
unfolding *changes-itv-smods-def changes-poly-at-def cross-def*
apply (*insert assms*(*2*,*3*))
apply (*subst* (*asm*) (*1 2*) *neq-iff*)
by (*auto simp add*: *variation-cases*)
ultimately show *?thesis* **by** *auto*
next
case *False*
hence *p*≠*0 q*≠*0 p mod q*≠*0* **by** *auto*
then obtain *ps* **where** *ps*:*smods p q*=*p*#*q*#−(*p mod q*)#*ps smods q* (−(*p mod q*)) = *q*#−(*p mod q*)#*ps*
by *auto*
define *changes-diff* **where** *changes-diff*≡λ*x*. *changes-poly-at* (*p*#*q*#−(*p mod q*)#*ps*) *x*
− *changes-poly-at* (*q*#−(*p mod q*)#*ps*) *x*
have ⋀*x*. *poly p x*∗*poly q x*<*0* ⟹ *changes-diff x*=*1*
unfolding *changes-diff-def changes-poly-at-def* **by** *auto*

```
  moreover have ⋀x. poly p x*poly q x>0 ⟹ changes-diff x=0
    unfolding changes-diff-def changes-poly-at-def  by auto
  ultimately have changes-diff a − changes-diff b=cross (p*q) a b
    unfolding cross-def
    apply (cases rule:neqE[OF ‹poly (p*q) a≠0›])
    by (cases rule:neqE[OF ‹poly (p*q) b≠0›],auto simp add:variation-cases)+
 thus ?thesis unfolding changes-itv-smods-def changes-diff-def changes-poly-at-def

    using ps by auto
qed

lemma changes-smods-congr:
  fixes p q:: real poly
  assumes a≠a' poly p a≠0
  assumes ∀p∈set (smods p q). ∀x. ((a<x∧x≤a') ∨ (a'≤x ∧ x<a)) ⟶ poly p x
≠0
  shows changes-poly-at (smods p q) a = changes-poly-at (smods p q) a'
    using assms(2−3)
proof (induct smods p q arbitrary:p q rule:length-induct)
  case 1
  have p≠0 using ‹poly p a ≠0› by auto
  define r1 where r1≡− (p mod q)
  have a-a'-rel:∀pp∈set (smods p q). poly pp a * poly pp a' ≥0
  proof (rule ccontr)
    assume ¬ (∀pp∈set (smods p q). 0 ≤ poly pp a * poly pp a')
    then obtain pp where pp:pp∈set (smods p q)  poly pp a * poly pp a'<0
      using ‹p≠0› by (metis less-eq-real-def linorder-neqE-linordered-idom)
    hence a<a' ⟹ False using 1.prems(2) poly-IVT[of a a' pp] by auto
    moreover have a'<a⟹False
      using pp[unfolded mult.commute[of poly pp a]] 1.prems(2) poly-IVT[of a' a
pp] by auto
    ultimately show False using ‹a≠a'› by force
  qed
  have q=0 ⟹ ?case by auto
  moreover have ⟦q≠0;poly q a=0⟧ ⟹ ?case
  proof −
    assume q≠0 poly q a=0
    define r2 where r2≡− (q mod r1)
    have − poly r1 a = poly p a
      by (metis ‹poly q a = 0› add.inverse-inverse add.left-neutral div-mult-mod-eq
          mult-zero-right poly-add poly-minus poly-mult r1-def)
    hence r1≠0 and poly r1 a≠0 and poly p a*poly r1 a<0 using ‹poly p a≠0›
      apply auto
      using mult-less-0-iff by fastforce
    then obtain ps where ps:smods p q=p#q#r1#ps smods r1 r2=r1#ps
      by (metis ‹p≠0› ‹q ≠ 0› r1-def r2-def smods.simps)
    hence length (smods r1 r2)<length (smods p q) by auto
    moreover have (∀p∈set (smods r1 r2). ∀x. a < x ∧ x ≤ a' ∨ a' ≤ x ∧ x <
a ⟶ poly p x ≠ 0)
```

using *1.prems(2)* **unfolding** *ps* **by** *auto*
ultimately have *changes-poly-at (smods r1 r2) a = changes-poly-at (smods r1 r2) a'*
using *1.hyps ‹r1≠0› ‹poly r1 a≠0›* **by** *metis*
moreover have *changes-poly-at (smods p q) a = 1+changes-poly-at (smods r1 r2) a*
unfolding *ps changes-poly-at-def* **using** *‹poly q a=0› ‹poly p a*poly r1 a<0›* **by** *auto*
moreover have *changes-poly-at (smods p q) a' = 1+changes-poly-at (smods r1 r2) a'*
proof −
have *poly p a * poly p a' ≥0* **and** *poly r1 a*poly r1 a'≥0*
using *a-a'-rel* **unfolding** *ps* **by** *auto*
moreover have *poly p a'≠0* **and** *poly q a'≠0* **and** *poly r1 a'≠0*
using *1.prems(2)[unfolded ps] ‹a≠a'›* **by** *auto*
ultimately show *?thesis* **using** *‹poly p a*poly r1 a<0›* **unfolding** *ps changes-poly-at-def*
by (*auto simp add*: *zero-le-mult-iff*, *auto simp add*: *mult-less-0-iff*)
qed
ultimately show *?thesis* **by** *simp*
qed
moreover have ⟦*q≠0*;*poly q a≠0*⟧ ⟹ *?case*
proof −
assume *q≠0 poly q a≠0*
then obtain *ps* **where** *ps*:*smods p q=p#q#ps smods q r1=q#ps*
by (*metis ‹p≠0› r1-def smods.simps*)
hence *length (smods q r1) < length (smods p q)* **by** *auto*
moreover have (∀ *p*∈*set (smods q r1)*. ∀ *x*. *a < x ∧ x ≤ a' ∨ a' ≤ x ∧ x < a ⟶ poly p x ≠ 0*)
using *1.prems(2)* **unfolding** *ps* **by** *auto*
ultimately have *changes-poly-at (smods q r1) a = changes-poly-at (smods q r1) a'*
using *1.hyps ‹q≠0› ‹poly q a≠0›* **by** *metis*
moreover have *poly p a'≠0* **and** *poly q a'≠0*
using *1.prems(2)[unfolded ps] ‹a≠a'›* **by** *auto*
moreover have *poly p a * poly p a' ≥0* **and** *poly q a*poly q a'≥0*
using *a-a'-rel* **unfolding** *ps* **by** *auto*
ultimately show *?thesis* **unfolding** *ps changes-poly-at-def* **using** *‹poly q a≠0› ‹poly p a≠0›*
by (*auto simp add*: *zero-le-mult-iff*,*auto simp add*: *mult-less-0-iff*)
qed
ultimately show *?case* **by** *blast*
qed

lemma *changes-itv-smods-congr*:
fixes *p q*:: *real poly*
assumes *a<a' a'<b' b'<b poly p a≠0 poly p b≠0*
assumes *no-root*:∀ *p*∈*set (smods p q)*. ∀ *x*. ((*a<x∧x≤a'*) ∨ (*b'≤x ∧ x<b*)) ⟶ *poly p x ≠0*

shows *changes-itv-smods a b p q=changes-itv-smods a′ b′ p q*
proof −
have *changes-poly-at (smods p q) a = changes-poly-at (smods p q) a′*
apply (*rule changes-smods-congr*[*OF order.strict-implies-not-eq*[*OF* ‹*a*<*a′*›] ‹*poly p a*≠*0*›])
by (*metis assms*(*1*) *less-eq-real-def less-irrefl less-trans no-root*)
moreover have *changes-poly-at (smods p q) b = changes-poly-at (smods p q) b′*
apply (*rule changes-smods-congr*[*OF order.strict-implies-not-eq*[*OF* ‹*b′*<*b*›, *symmetric*] ‹*poly p b*≠*0*›])
by (*metis assms*(*3*) *less-eq-real-def less-trans no-root*)
ultimately show *?thesis* **unfolding** *changes-itv-smods-def Let-def* **by** *auto*
qed

lemma *cindex-poly-changes-itv-mods*:
assumes *a*<*b poly p a*≠*0 poly p b*≠*0*
shows *cindex-poly a b q p = changes-itv-smods a b p q* **using** *assms*
proof (*induct smods p q arbitrary*:*p q a b*)
case *Nil*
hence *p*=*0* **by** (*metis smods-nil-eq*)
thus *?case* **using** ‹*poly p a* ≠ *0*› **by** *simp*
next
case (*Cons x1 xs*)
have *p*≠*0* **using** ‹*poly p a* ≠ *0*› **by** *auto*
obtain *a′ b′* **where** *a*<*a′ a′*<*b′ b′*<*b*
and *no-root*:∀ *p*∈*set* (*smods p q*). ∀ *x*. ((*a*<*x*∧*x*≤*a′*) ∨ (*b′*≤*x* ∧ *x*<*b*)) ⟶ *poly p x* ≠*0*
proof (*induct smods p q arbitrary*:*p q thesis*)
case *Nil*
define *a′ b′* **where** *a′*≡*2*/*3* ∗ *a* + *1*/*3* ∗ *b* **and** *b′*≡*1*/*3*∗*a* + *2*/*3*∗*b*
have *a* < *a′* **and** *a′* < *b′* **and** *b′* < *b* **unfolding** *a′-def b′-def* **using** ‹*a*<*b*› **by** *auto*
moreover have ∀ *p*∈*set* (*smods p q*). ∀ *x*. *a* < *x* ∧ *x* ≤ *a′* ∨ *b′* ≤ *x* ∧ *x* < *b* ⟶ *poly p x* ≠ *0*
unfolding ‹[] = *smods p q*›[*symmetric*] **by** *auto*
ultimately show *?case* **using** *Nil* **by** *auto*
next
case (*Cons x1 xs*)
define *r* **where** *r*≡− (*p mod q*)
then have *smods p q* = *p* # *xs* **and** *smods q r* = *xs* **and** *p* ≠ *0*
using ‹*x1* # *xs* = *smods p q*›
by (*auto simp del*: *smods.simps simp add*: *smods.simps* [*of p q*] *split*: *if-splits*)
obtain *a1 b1* **where**
a < *a1* *a1* < *b1* *b1* < *b* **and**
a1-b1-no-root:∀ *p*∈*set xs*. ∀ *x*. *a* < *x* ∧ *x* ≤ *a1* ∨ *b1* ≤ *x* ∧ *x* < *b* ⟶ *poly p x* ≠ *0*
using *Cons*(*1*)[*OF* ‹*smods q r*=*xs*›[*symmetric*]] ‹*smods q r*=*xs*› **by** *auto*
obtain *a2 b2* **where**
a<*a2* **and** *a2*:∀ *x*. *a*<*x* ∧ *x*≤*a2* ⟶ *poly p x* ≠ *0*
b2<*b* **and** *b2*:∀ *x*. *b2*≤*x* ∧ *x*<*b* ⟶ *poly p x* ≠ *0*

```
    using next-non-root-interval[OF ‹p≠0›] last-non-root-interval[OF ‹p≠0›]
    by (metis less-numeral-extra(3))
  define a' b' where a'≡ if b2>a then Min{a1, b2, a2} else min a1 a2
    and b'≡if a2 <b then Max{ b1, a2, b2} else max b1 b2
  have a < a' a' < b' b' < b unfolding a'-def b'-def
    using ‹a < a1› ‹a1 < b1› ‹b1 < b› ‹a<a2› ‹b2<b› ‹a<b› by auto
  moreover have ∀ p∈set xs. ∀ x. a < x ∧ x ≤ a' ∨ b' ≤ x ∧ x < b ⟶ poly p
x ≠ 0
    using a1-b1-no-root unfolding a'-def b'-def by auto
  moreover have ∀ x. a < x ∧ x ≤ a' ∨ b' ≤ x ∧ x < b ⟶ poly p x ≠ 0
    using a2 b2 unfolding a'-def b'-def by auto
  ultimately show ?case using Cons(3)[unfolded ‹smods p q=p#xs›] by auto
 qed
 have q=0 ⟹ ?case by simp
 moreover have q≠0 ⟹ ?case
 proof −
  assume q≠0
  define r where r≡− (p mod q)
  obtain ps where ps:smods p q=p#q#ps smods q r=q#ps and xs=q#ps
   unfolding r-def using ‹q≠0› ‹p≠0› ‹x1 # xs = smods p q›
   by (metis list.inject smods.simps)
  have poly p a' ≠ 0 poly p b' ≠ 0 poly q a' ≠ 0 poly q b' ≠ 0
   using no-root[unfolded ps] ‹a'>a› ‹b'<b› by auto
  moreover hence
      changes-itv-smods a' b' p q = cross (p * q) a' b' + changes-itv-smods a'
b' q r
      cindex-poly a' b' q p = cross (p * q) a' b' + cindex-poly a' b' r q
   using changes-itv-smods-rec[OF ‹a'<b'›,of p q,folded r-def]
     cindex-poly-rec[OF ‹a'<b'›,of p q,folded r-def] by auto
  moreover have changes-itv-smods a' b' q r = cindex-poly a' b' r q
    using Cons.hyps(1)[of q r a' b'] ‹a' < b'› ‹q ≠ 0› ‹xs = q # ps› ps(2)
     ‹poly q a' ≠ 0› ‹poly q b' ≠ 0› by simp
  ultimately have changes-itv-smods a' b' p q = cindex-poly a' b' q p by auto
  thus ?thesis
   using
     changes-itv-smods-congr[OF ‹a<a'› ‹a'<b'› ‹b'<b› Cons(4,5),of q]
     no-root cindex-poly-congr[OF ‹a<a'› ‹a'<b'› ‹b'<b› ] ps
   by (metis insert-iff list.set(2))
 qed
 ultimately show ?case by metis
qed

lemma root-list-ub:
 fixes ps:: (real poly) list and a::real
 assumes 0∉set ps
 obtains ub where ∀ p∈set ps. ∀ x. poly p x=0 ⟶ x<ub
  and ∀ x≥ub. ∀ p∈set ps. sgn (poly p x) = sgn-pos-inf p and ub>a
  using assms
proof (induct ps arbitrary:thesis)
```

case *Nil*
show *?case* **using** *Nil(1)[of a+1]* **by** *auto*
next
case (*Cons p ps*)
hence *p≠0* **and** *0∉set ps* **by** *auto*
then obtain *ub1* **where** *ub1*:∀ *p*∈*set ps*. ∀ *x*. *poly p x* = *0* ⟶ *x* < *ub1* **and**
ub1-sgn:∀ *x*≥*ub1*. ∀ *p*∈*set ps*. *sgn* (*poly p x*) = *sgn-pos-inf p* **and** *ub1*>*a*
using *Cons.hyps* **by** *auto*
obtain *ub2* **where** *ub2*:∀ *x*. *poly p x* = *0* ⟶ *x* < *ub2*
and *ub2-sgn*: ∀ *x*≥*ub2*. *sgn* (*poly p x*) = *sgn-pos-inf p*
using *root-ub*[*OF* ‹*p≠0*›] **by** *auto*
define *ub* **where** *ub*≡*max ub1 ub2*
have ∀ *p*∈*set* (*p* # *ps*). ∀ *x*. *poly p x* = *0* ⟶ *x* < *ub* **using** *ub1 ub2 ub-def* **by** *force*
moreover have ∀ *x*≥*ub*. ∀ *p*∈*set* (*p* # *ps*). *sgn* (*poly p x*) = *sgn-pos-inf p*
using *ub1-sgn ub2-sgn ub-def* **by** *auto*
ultimately show *?case* **using** *Cons(2)*[*of ub*] ‹*ub1*>*a*› *ub-def* **by** *auto*
qed

lemma *root-list-lb*:
fixes *ps*:: (*real poly*) *list* **and** *b*::*real*
assumes *0∉set ps*
obtains *lb* **where** ∀ *p*∈*set ps*. ∀ *x*. *poly p x*=*0* ⟶ *x*>*lb*
and ∀ *x*≤*lb*. ∀ *p*∈*set ps*. *sgn* (*poly p x*) = *sgn-neg-inf p* **and** *lb*<*b*
using *assms*
proof (*induct ps arbitrary*:*thesis*)
case *Nil*
show *?case* **using** *Nil(1)*[*of b − 1*] **by** *auto*
next
case (*Cons p ps*)
hence *p≠0* **and** *0∉set ps* **by** *auto*
then obtain *lb1* **where** *lb1*:∀ *p*∈*set ps*. ∀ *x*. *poly p x* = *0* ⟶ *x* > *lb1* **and**
lb1-sgn:∀ *x*≤*lb1*. ∀ *p*∈*set ps*. *sgn* (*poly p x*) = *sgn-neg-inf p* **and** *lb1*<*b*
using *Cons.hyps* **by** *auto*
obtain *lb2* **where** *lb2*:∀ *x*. *poly p x* = *0* ⟶ *x* > *lb2*
and *lb2-sgn*: ∀ *x*≤*lb2*. *sgn* (*poly p x*) = *sgn-neg-inf p*
using *root-lb*[*OF* ‹*p≠0*›] **by** *auto*
define *lb* **where** *lb*≡*min lb1 lb2*
have ∀ *p*∈*set* (*p* # *ps*). ∀ *x*. *poly p x* = *0* ⟶ *x* > *lb* **using** *lb1 lb2 lb-def* **by** *force*
moreover have ∀ *x*≤*lb*. ∀ *p*∈*set* (*p* # *ps*). *sgn* (*poly p x*) = *sgn-neg-inf p*
using *lb1-sgn lb2-sgn lb-def* **by** *auto*
ultimately show *?case* **using** *Cons(2)*[*of lb*] ‹*lb1*<*b*› *lb-def* **by** *auto*
qed

theorem *sturm-tarski-interval*:
assumes *a*<*b poly p a*≠*0 poly p b*≠*0*
shows *taq* {*x*. *poly p x*=*0* ∧ *a*<*x* ∧ *x*<*b*} *q* = *changes-itv-smods a b p* (*pderiv p* ∗ *q*)

proof −
have *p≠0* **using** ‹*poly p a≠0*› **by** *auto*
thus *?thesis* **using** *cindex-poly-taq cindex-poly-changes-itv-mods*[*OF assms*] **by** *auto*
qed

theorem *sturm-tarski-above*:
assumes *poly p a≠0*
shows *taq* {*x*. *poly p x=0* ∧ *a<x*} *q* = *changes-gt-smods a p* (*pderiv p* ∗ *q*)
proof −
define *ps* **where** *ps≡smods p* (*pderiv p* ∗ *q*)
have *p≠0* **and** *p∈set ps* **using** ‹*poly p a≠0*› *ps-def* **by** *auto*
obtain *ub* **where** *ub*:∀ *p∈set ps*. ∀ *x*. *poly p x=0* ⟶ *x<ub*
and *ub-sgn*:∀ *x≥ub*. ∀ *p∈set ps*. *sgn* (*poly p x*) = *sgn-pos-inf p*
and *ub>a*
using *root-list-ub*[*OF no-0-in-smods*,*of p pderiv p* ∗ *q*,*folded ps-def*]
by *auto*
have *taq* {*x*. *poly p x=0* ∧ *a<x*} *q* = *taq* {*x*. *poly p x=0* ∧ *a<x* ∧ *x<ub*} *q*
unfolding *taq-def* **by** (*rule sum.cong*,*insert ub* ‹*p∈set ps*›,*auto*)
moreover have *changes-gt-smods a p* (*pderiv p* ∗ *q*) = *changes-itv-smods a ub p* (*pderiv p* ∗ *q*)
proof −
have *map* (*sgn* ∘ (λ*p*. *poly p ub*)) *ps* = *map sgn-pos-inf ps*
using *ub-sgn*[*THEN spec*,*of ub*,*simplified*]
by (*metis* (*mono-tags*, *lifting*) *comp-def list.map-cong0*)
hence *changes-poly-at ps ub=changes-poly-pos-inf ps*
unfolding *changes-poly-pos-inf-def changes-poly-at-def*
by (*subst changes-map-sgn-eq*,*metis map-map*)
thus *?thesis* **unfolding** *changes-gt-smods-def changes-itv-smods-def ps-def*
by *metis*
qed
moreover have *poly p ub≠0* **using** *ub* ‹*p∈set ps*› **by** *auto*
ultimately show *?thesis* **using** *sturm-tarski-interval*[*OF* ‹*ub>a*› *assms*] **by** *auto*
qed

theorem *sturm-tarski-below*:
assumes *poly p b≠0*
shows *taq* {*x*. *poly p x=0* ∧ *x<b*} *q* = *changes-le-smods b p* (*pderiv p* ∗ *q*)
proof −
define *ps* **where** *ps≡smods p* (*pderiv p* ∗ *q*)
have *p≠0* **and** *p∈set ps* **using** ‹*poly p b≠0*› *ps-def* **by** *auto*
obtain *lb* **where** *lb*:∀ *p∈set ps*. ∀ *x*. *poly p x=0* ⟶ *x>lb*
and *lb-sgn*:∀ *x≤lb*. ∀ *p∈set ps*. *sgn* (*poly p x*) = *sgn-neg-inf p*
and *lb<b*
using *root-list-lb*[*OF no-0-in-smods*,*of p pderiv p* ∗ *q*,*folded ps-def*]
by *auto*
have *taq* {*x*. *poly p x=0* ∧ *x<b*} *q* = *taq* {*x*. *poly p x=0* ∧ *lb<x* ∧ *x<b*} *q*
unfolding *taq-def* **by** (*rule sum.cong*,*insert lb* ‹*p∈set ps*›,*auto*)
moreover have *changes-le-smods b p* (*pderiv p* ∗ *q*) = *changes-itv-smods lb b p*

```
(pderiv p * q)
  proof −
    have map (sgn ∘ (λp. poly p lb)) ps = map sgn-neg-inf ps
      using lb-sgn[THEN spec,of lb,simplified]
      by (metis (mono-tags, lifting) comp-def list.map-cong0)
    hence changes-poly-at ps lb=changes-poly-neg-inf ps
      unfolding changes-poly-neg-inf-def changes-poly-at-def
      by (subst changes-map-sgn-eq,metis map-map)
    thus ?thesis unfolding changes-le-smods-def changes-itv-smods-def ps-def
      by metis
  qed
  moreover have poly p lb≠0 using lb ‹p∈set ps› by auto
  ultimately show ?thesis using sturm-tarski-interval[OF ‹lb<b› - assms] by
auto
qed

theorem sturm-tarski-R:
  shows taq {x. poly p x=0} q = changes-R-smods p (pderiv p * q)
proof (cases p=0)
  case True
  then show ?thesis
    unfolding taq-def using infinite-UNIV-char-0 by (auto intro!:sum.infinite)
next
  case False
  define ps where ps≡smods p (pderiv p * q)
  have p∈set ps using ps-def ‹p≠0› by auto
  obtain lb where lb:∀ p∈set ps. ∀ x. poly p x=0 ⟶ x>lb
      and lb-sgn:∀ x≤lb. ∀ p∈set ps. sgn (poly p x) = sgn-neg-inf p
      and lb<0
    using root-list-lb[OF no-0-in-smods,of p pderiv p * q,folded ps-def]
    by auto
  obtain ub where ub:∀ p∈set ps. ∀ x. poly p x=0 ⟶ x<ub
      and ub-sgn:∀ x≥ub. ∀ p∈set ps. sgn (poly p x) = sgn-pos-inf p
      and ub>0
    using root-list-ub[OF no-0-in-smods,of p pderiv p * q,folded ps-def]
    by auto
  have taq {x. poly p x=0} q = taq {x. poly p x=0 ∧ lb<x ∧ x<ub} q
    unfolding taq-def by (rule sum.cong,insert lb ub ‹p∈set ps›,auto)
  moreover have changes-R-smods p (pderiv p * q) = changes-itv-smods lb ub p
(pderiv p * q)
  proof −
    have map (sgn ∘ (λp. poly p lb)) ps = map sgn-neg-inf ps
       and map (sgn ∘ (λp. poly p ub)) ps = map sgn-pos-inf ps
      using lb-sgn[THEN spec,of lb,simplified] ub-sgn[THEN spec,of ub,simplified]
      by (metis (mono-tags, lifting) comp-def list.map-cong0)+
    hence changes-poly-at ps lb=changes-poly-neg-inf ps
        ∧ changes-poly-at ps ub=changes-poly-pos-inf ps
     unfolding changes-poly-neg-inf-def changes-poly-at-def changes-poly-pos-inf-def
      by (subst (1 3)  changes-map-sgn-eq,metis map-map)
```

```
    thus ?thesis unfolding changes-R-smods-def changes-itv-smods-def ps-def
      by metis
  qed
  moreover have poly p lb≠0 and poly p ub≠0 using lb ub ‹p∈set ps› by auto
  moreover have lb<ub using ‹lb<0› ‹0<ub› by auto
  ultimately show ?thesis using sturm-tarski-interval by auto
qed

theorem sturm-interval:
  assumes a < b poly p a ≠ 0 poly p b ≠ 0
  shows card {x. poly p x = 0 ∧ a < x ∧ x < b} = changes-itv-smods a b p
(pderiv p)
using sturm-tarski-interval[OF assms, unfolded taq-def,of 1] by force

theorem sturm-above:
  assumes poly p a ≠ 0
  shows card {x. poly p x = 0 ∧ a < x} = changes-gt-smods a p (pderiv p)
using sturm-tarski-above[OF assms, unfolded taq-def,of 1] by force

theorem sturm-below:
  assumes poly p b ≠ 0
  shows card {x. poly p x = 0 ∧ x < b} = changes-le-smods b p (pderiv p)
using sturm-tarski-below[OF assms, unfolded taq-def,of 1] by force

theorem sturm-R:
  shows card {x. poly p x=0} =  changes-R-smods p (pderiv p)
using sturm-tarski-R[of - 1,unfolded taq-def] by force

end
```

References

[1] S. Basu, R. Pollack, and M.-F. Roy. *Algorithms in Real Algebraic Geometry (Algorithms and Computation in Mathematics)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.

[2] C. Cohen. *Formalized algebraic numbers: construction and first-order theory*. PhD thesis, École polytechnique, Nov 2012.

[3] W. Li and L. C. Paulson. A modular, efficient formalisation of real algebraic numbers. In *Proceedings of the 5th ACM SIGPLAN Conference on Certified Programs and Proofs*, CPP 2016, pages 66–75, New York, NY, USA, 2016. ACM.