The Budan–Fourier Theorem and Counting Real Roots with Multiplicity

Wenda Li

May 26, 2024

Abstract

This entry is mainly about counting and approximating real roots (of a polynomial) with multiplicity. We have first formalised the Budan–Fourier theorem: given a polynomial with real coefficients, we can calculate sign variations on Fourier sequences to over-approximate the number of real roots (counting multiplicity) within an interval. When all roots are known to be real, the over-approximation becomes tight: we can utilise this theorem to count real roots exactly. It is also worth noting that Descartes' rule of sign is a direct consequence of the Budan–Fourier theorem, and has been included in this entry. In addition, we have extended previous formalised Sturm's theorem to count real roots with multiplicity, while the original Sturm's theorem only counts distinct real roots. Compared to the Budan–Fourier theorem, our extended Sturm's theorem always counts roots exactly but may suffer from greater computational cost.

Many problems in real algebraic geometry is about counting or approximating roots of a polynomial. Previous formalised results are mainly about counting distinct real roots (i.e. Sturm's theorem in Isabelle/HOL [5, 2], HOL Light [4], PVS [9] and Coq [8]) and limited support for multiple real roots (i.e. Descartes' rule of signs in Isabelle/HOL [3], HOL Light and ProofPower[1]). In comparison, this entry provides more comprehensive support for reasoning about multiple real roots.

The main motivation of this entry is to cope with the roots-on-the-border issue when counting complex roots [7, 6], but the results here should be beneficial to other developments.

Our proof of the Budan–Fourier theorem mainly follows Theorem 2.35 in the book by Basu et al. [1] and that of the extended Sturm's theorem is inspired by Theorem 10.5.6 in Rahman and Schmeisser's book [10].

[1] According to Freek Wiedijk's "Formalising 100 Theorems" (http://www.cs.ru.nl/~freek/100/index.html)

1 Misc results for polynomials and sign variations

theory *BF-Misc* **imports**
HOL−Computational-Algebra.Polynomial-Factorial
HOL−Computational-Algebra.Fundamental-Theorem-Algebra
Sturm-Tarski.Sturm-Tarski
begin

1.1 Induction on polynomial roots

lemma *poly-root-induct-alt* [*case-names 0 no-proots root*]:
fixes p :: $'a$ :: *idom poly*
assumes $Q\ 0$
assumes $\bigwedge p.\ (\bigwedge a.\ poly\ p\ a \neq 0) \Longrightarrow Q\ p$
assumes $\bigwedge a\ p.\ Q\ p \Longrightarrow Q\ ([:-a,\ 1:] * p)$
shows $Q\ p$
proof (*induction degree p arbitrary*: *p rule*: *less-induct*)
case (*less p*)
have *?case* **when** $p{=}0$ **using** ‹$Q\ 0$› *that* **by** *auto*
moreover have *?case* **when** $\nexists\, a.\ poly\ p\ a = 0$
using *assms*(*2*) *that* **by** *blast*
moreover have *?case* **when** $\exists\, a.\ poly\ p\ a = 0\ p{\neq}0$
proof −
obtain *a* **where** $poly\ p\ a = 0$ **using** ‹$\exists\, a.\ poly\ p\ a = 0$› **by** *auto*
then obtain *q* **where** $pq{:}p{=}\ [:-a{,}1:] * q$ **by** (*meson dvdE poly-eq-0-iff-dvd*)
then have $q{\neq}0$ **using** ‹$p{\neq}0$› **by** *auto*
then have $degree\ q{<}degree\ p$ **unfolding** *pq* **by** (*subst degree-mult-eq*,*auto*)
then have $Q\ q$ **using** *less* **by** *auto*
then show *?case* **using** *assms*(*3*) **unfolding** *pq* **by** *auto*
qed
ultimately show *?case* **by** *auto*
qed

1.2 Misc

lemma *lead-coeff-pderiv*:
fixes p :: $'a$::{*comm-semiring-1*,*semiring-no-zero-divisors*,*semiring-char-0*} *poly*
shows *lead-coeff* (*pderiv p*) = *of-nat* (*degree p*) ∗ *lead-coeff p*
apply (*auto simp*:*degree-pderiv coeff-pderiv*)
apply (*cases degree p*)
by (*auto simp add*: *coeff-eq-0*)

lemma *gcd-degree-le-min*:
assumes $p{\neq}0\ q{\neq}0$
shows *degree* (*gcd p q*) ≤ *min* (*degree p*) (*degree q*)
by (*simp add*: *assms*(*1*) *assms*(*2*) *dvd-imp-degree-le*)

lemma *lead-coeff-normalize-field*:
fixes p::$'a$::{*field*,*semidom-divide-unit-factor*} *poly*
assumes $p{\neq}0$

shows *lead-coeff* (*normalize p*) = *1*
by (*metis* (*no-types*, *lifting*) *assms coeff-normalize divide-self-if dvd-field-iff is-unit-unit-factor leading-coeff-0-iff normalize-eq-0-iff normalize-idem*)

lemma *smult-normalize-field-eq*:
fixes *p*::*'a*::{*field*,*semidom-divide-unit-factor*} *poly*
shows *p* = *smult* (*lead-coeff p*) (*normalize p*)
proof (*rule poly-eqI*)
fix *n*
have *unit-factor* (*lead-coeff p*) = *lead-coeff p*
by (*metis dvd-field-iff is-unit-unit-factor unit-factor-0*)
then show *coeff p n* = *coeff* (*smult* (*lead-coeff p*) (*normalize p*)) *n*
by *simp*
qed

lemma *lead-coeff-gcd-field*:
fixes *p q*::*'a*::*field-gcd poly*
assumes *p*≠*0* ∨ *q*≠*0*
shows *lead-coeff* (*gcd p q*) = *1*
using *assms* **by** (*metis gcd.normalize-idem gcd-eq-0-iff lead-coeff-normalize-field*)

lemma *poly-gcd-0-iff*:
poly (*gcd p q*) *x* = *0* ⟷ *poly p x*=*0* ∧ *poly q x*=*0*
by (*simp add*:*poly-eq-0-iff-dvd*)

lemma *degree-eq-oneE*:
fixes *p* :: *'a*::*zero poly*
assumes *degree p* = *1*
obtains *a b* **where** *p* = [:*a*,*b*:] *b*≠*0*
proof −
obtain *a b q* **where** *p*:*p*=*pCons a* (*pCons b q*)
by (*metis pCons-cases*)
with *assms* **have** *q*=*0* **by** (*cases q* = *0*) *simp-all*
with *p* **have** *p*=[:*a*,*b*:] **by** *auto*
moreover then have *b*≠*0* **using** *assms* **by** *auto*
ultimately show *?thesis* **..**
qed

1.3 More results about sign variations (i.e. *changes*

lemma *changes-0*[*simp*]:*changes* (*0*#*xs*) = *changes xs*
by (*cases xs*) *auto*

lemma *changes-Cons*:*changes* (*x*#*xs*) = (*if filter* (λ*x*. *x*≠*0*) *xs* = [] *then*
0
else if x∗ *hd* (*filter* (λ*x*. *x*≠*0*) *xs*) < *0 then*
1 + *changes xs*
else changes xs)
apply (*induct xs*)

```
  by auto

lemma changes-filter-eq:
  changes (filter (λx. x≠0) xs) = changes xs
  apply (induct xs)
  by (auto simp add:changes-Cons)

lemma changes-filter-empty:
  assumes filter (λx. x≠0) xs = []
  shows changes xs = 0 changes (a#xs) = 0 using assms
  apply (induct xs)
  apply auto
  by (metis changes-0 neq-Nil-conv)

lemma changes-append:
  assumes xs≠ [] ∧ ys≠ [] ⟶ (last xs = hd ys ∧ last xs≠0)
  shows changes (xs@ys) = changes xs + changes ys
  using assms
proof (induct xs)
  case Nil
  then show ?case by simp
next
  case (Cons a xs)
  have ?case when xs=[]
    using that Cons
    apply (cases ys)
    by auto
  moreover have ?case when ys=[]
    using that Cons by auto
  moreover have ?case when xs≠[] ys≠[]
  proof -
    have filter (λx. x ≠ 0) xs ≠[]
      using that Cons
      apply auto
        by (metis (mono-tags, lifting) filter.simps(1) filter.simps(2) filter-append
snoc-eq-iff-butlast)
    then have changes (a # xs @ ys) = changes (a # xs) + changes ys
      apply (subst (1 2) changes-Cons)
      using that Cons by auto
    then show ?thesis by auto
  qed
  ultimately show ?case by blast
qed

lemma changes-drop-dup:
  assumes xs≠ [] ys≠ [] ⟶ last xs=hd ys
  shows changes (xs@ys) = changes (xs@ tl ys)
  using assms
proof (induct xs)
```

```
  case Nil
  then show ?case by simp
next
  case (Cons a xs)
  have ?case when ys=[]
    using that by simp
  moreover have ?case when ys≠[] xs=[]
    using that Cons
    apply auto
    by (metis changes.simps(3) list.exhaust-sel not-square-less-zero)
  moreover have ?case when ys≠[] xs≠[]
  proof −
    define ts ts' where ts = filter (λx. x ≠ 0) (xs @ ys)
      and ts' = filter (λx. x ≠ 0) (xs @ tl ys)
    have (ts = [] ⟷ ts' = []) ∧ hd ts = hd ts'
    proof (cases filter (λx. x ≠ 0) xs = [])
      case True
      then have last xs = 0 using ‹xs≠[]›
        by (metis (mono-tags, lifting) append-butlast-last-id append-is-Nil-conv
            filter.simps(2) filter-append list.simps(3))
      then have hd ys=0 using Cons(3)[rule-format, OF ‹ys≠[]›] ‹xs≠[]› by auto
      then have filter (λx. x ≠ 0) ys = filter (λx. x ≠ 0) (tl ys)
        by (metis (mono-tags, lifting) filter.simps(2) list.exhaust-sel that(1))
      then show ?thesis unfolding ts-def ts'-def by auto
    next
      case False
      then show ?thesis unfolding ts-def ts'-def by auto
    qed
    moreover have changes (xs @ ys) = changes (xs @ tl ys)
      apply (rule Cons(1))
      using that Cons(3) by auto
    moreover have changes (a # xs @ ys) = (if ts = [] then 0 else if a * hd ts <
0
            then 1 + changes (xs @ ys) else changes (xs @ ys))
      using changes-Cons[of a xs @ ys,folded ts-def] .
    moreover have changes (a # xs @ tl ys) = (if ts' = [] then 0 else if a * hd ts'
< 0
            then 1 + changes (xs @ tl ys) else changes (xs @ tl ys))
      using changes-Cons[of a xs @ tl ys,folded ts'-def] .
    ultimately show ?thesis by auto
  qed
  ultimately show ?case by blast
qed
```

```
lemma Im-poly-of-real:
  Im (poly p (of-real x)) = poly (map-poly Im p) x
  apply (induct p)
```

by (*auto simp add*:*map-poly-pCons*)

lemma *Re-poly-of-real*:
Re (*poly p* (*of-real x*)) = *poly* (*map-poly Re p*) *x*
apply (*induct p*)
by (*auto simp add*:*map-poly-pCons*)

1.4 More about *map-poly* and *of-real*

lemma *of-real-poly-map-pCons*[*simp*]:*map-poly of-real* (*pCons a p*) = *pCons* (*of-real a*) (*map-poly of-real p*)
by (*simp add*: *map-poly-pCons*)

lemma *of-real-poly-map-plus*[*simp*]: *map-poly of-real* (*p* + *q*) = *map-poly of-real p* + *map-poly of-real q*
apply (*rule poly-eqI*)
by (*auto simp add*: *coeff-map-poly*)

lemma *of-real-poly-map-smult*[*simp*]:*map-poly of-real* (*smult s p*) = *smult* (*of-real s*) (*map-poly of-real p*)
apply (*rule poly-eqI*)
by (*auto simp add*: *coeff-map-poly*)

lemma *of-real-poly-map-mult*[*simp*]:*map-poly of-real* (*p*∗*q*) = *map-poly of-real p* ∗ *map-poly of-real q*
by (*induct p*,*intro poly-eqI*,*auto*)

lemma *of-real-poly-map-poly*:
of-real (*poly p x*) = *poly* (*map-poly of-real p*) (*of-real x*)
by (*induct p*,*auto*)

lemma *of-real-poly-map-power*:*map-poly of-real* (*p*^*n*) = (*map-poly of-real p*) ^ *n*
by (*induct n*,*auto*)

lemma *of-real-poly-eq-iff* [*simp*]: *map-poly of-real p* = *map-poly of-real q* ⟷ *p* = *q*
by (*auto simp*: *poly-eq-iff coeff-map-poly*)

lemma *of-real-poly-eq-0-iff* [*simp*]: *map-poly of-real p* = *0* ⟷ *p* = *0*
by (*auto simp*: *poly-eq-iff coeff-map-poly*)

1.5 More about *order*

lemma *order-multiplicity-eq*:
assumes *p*≠*0*
shows *order a p* = *multiplicity* [:−*a*,*1*:] *p*
by (*metis assms multiplicity-eqI order-1 order-2*)

```
lemma order-gcd:
  assumes p≠0 q≠0
  shows order x (gcd p q) = min (order x p) (order x q)
proof −
  have prime [:− x, 1:]
   apply (auto simp add: prime-elem-linear-poly normalize-poly-def  intro!:primeI)
    by (simp add: pCons-one)
  then show ?thesis
    using assms
    by (auto simp add:order-multiplicity-eq intro:multiplicity-gcd)
qed

lemma order-linear[simp]: order x [:−a,1:] = (if x=a then 1 else 0)
  by (auto simp add:order-power-n-n[where n=1,simplified] order-0I)

lemma map-poly-order-of-real:
  assumes p≠0
  shows order (of-real t) (map-poly of-real p) = order t p using assms
proof (induct p rule:poly-root-induct-alt)
  case 0
  then show ?case by simp
next
  case (no-proots p)
  then have order t p = 0 using order-root by blast
  moreover have poly (map-poly of-real p) (of-real x) ≠0 for x
    apply (subst of-real-poly-map-poly[symmetric])
    using no-proots order-root by simp
  then have order (of-real t) (map-poly of-real p) = 0
    using order-root by blast
  ultimately show ?case by auto
next
  case (root a p)
  define a1 where a1=[:−a,1:]
  have [simp]:a1≠0 p≠0 unfolding a1-def using root(2) by auto
  have order (of-real t) (map-poly of-real a1) = order t a1
    unfolding a1-def by simp
  then show ?case
    apply (fold a1-def)
    by (simp add:order-mult root)
qed

lemma order-pcompose:
  assumes pcompose p q≠0
  shows order x (pcompose p q) = order x (q−[:poly q x:]) ∗ order (poly q x) p
  using ‹pcompose p q≠0›
proof (induct p rule:poly-root-induct-alt)
  case 0
  then show ?case by simp
next
```

case (*no-proots p*)
have *order x* (*p* $\circ_p$ *q*) = *0*
apply (*rule order-0I*)
using *no-proots* **by** (*auto simp*:*poly-pcompose*)
moreover have *order* (*poly q x*) *p* = *0*
apply (*rule order-0I*)
using *no-proots* **by** (*auto simp*:*poly-pcompose*)
ultimately show *?case* **by** *auto*
next
case (*root a p*)
define *a1* **where** *a1*=[:−*a*,*1*:]
have [*simp*]: *a1*≠*0 p*≠*0 a1* $\circ_p$ *q* ≠*0 p* $\circ_p$ *q* ≠ *0*
subgoal using *root*(*2*) **unfolding** *a1-def* **by** *simp*
subgoal using *root*(*2*) **by** *auto*
using *root*(*2*) **by** (*fold a1-def*,*auto simp*:*pcompose-mult*)
have *order x* ((*a1* ∗ *p*) $\circ_p$ *q*) = *order x* (*a1* $\circ_p$ *q*) + *order x* (*p* $\circ_p$ *q*)
unfolding *pcompose-mult* **by** (*auto simp*: *order-mult*)
also have ... = *order x* (*q*−[:*poly q x*:]) ∗ (*order* (*poly q x*) *a1* + *order* (*poly q x*) *p*)
proof −
have *order x* (*a1* $\circ_p$ *q*) = *order x* (*q*−[:*poly q x*:]) ∗ *order* (*poly q x*) *a1*
unfolding *a1-def*
apply (*auto simp*: *pcompose-pCons algebra-simps diff-conv-add-uminus*)
by (*simp add*: *order-0I*)
moreover have *order x* (*p* $\circ_p$ *q*) = *order x* (*q* − [:*poly q x*:]) ∗ *order* (*poly q x*) *p*
apply (*rule root.hyps*)
by *auto*
ultimately show *?thesis* **by** (*auto simp*:*algebra-simps*)
qed
also have ... = *order x* (*q* − [:*poly q x*:]) ∗ *order* (*poly q x*) (*a1* ∗ *p*)
by (*auto simp*:*order-mult*)
finally show *?case* **unfolding** *a1-def* **.**
qed

1.6 Polynomial roots / zeros

definition *proots-within*::′*a*::*comm-semiring-0 poly* ⇒ ′*a set* ⇒ ′*a set* **where**
proots-within p s = {*x*∈*s*. *poly p x*=*0*}

abbreviation *proots*::′*a*::*comm-semiring-0 poly* ⇒ ′*a set* **where**
proots p ≡ *proots-within p UNIV*

lemma *proots-def*: *proots p* = {*x*. *poly p x*=*0*}
unfolding *proots-within-def* **by** *auto*

lemma *proots-within-empty*[*simp*]:
proots-within p {} = {} **unfolding** *proots-within-def* **by** *auto*

lemma *proots-within-0*[*simp*]:
proots-within 0 s = s **unfolding** *proots-within-def* **by** *auto*

lemma *proots-withinI*[*intro*,*simp*]:
poly p x=0 ⟹ *x*∈*s* ⟹ *x*∈*proots-within p s*
unfolding *proots-within-def* **by** *auto*

lemma *proots-within-iff*[*simp*]:
x∈*proots-within p s* ⟷ *poly p x=0* ∧ *x*∈*s*
unfolding *proots-within-def* **by** *auto*

lemma *proots-within-union*:
proots-within p A ∪ *proots-within p B* = *proots-within p* (*A* ∪ *B*)
unfolding *proots-within-def* **by** *auto*

lemma *proots-within-times*:
fixes *s*::′*a*::{*semiring-no-zero-divisors*,*comm-semiring-0*} *set*
shows *proots-within* (*p*∗*q*) *s* = *proots-within p s* ∪ *proots-within q s*
unfolding *proots-within-def* **by** *auto*

lemma *proots-within-gcd*:
fixes *s*::′*a*::{*factorial-ring-gcd*,*semiring-gcd-mult-normalize*} *set*
shows *proots-within* (*gcd p q*) *s*= *proots-within p s* ∩ *proots-within q s*
unfolding *proots-within-def*
by (*auto simp add*: *poly-eq-0-iff-dvd*)

lemma *proots-within-inter*:
NO-MATCH UNIV s ⟹ *proots-within p s* = *proots p* ∩ *s*
unfolding *proots-within-def* **by** *auto*

lemma *proots-within-proots*[*simp*]:
proots-within p s ⊆ *proots p*
unfolding *proots-within-def* **by** *auto*

lemma *finite-proots*[*simp*]:
fixes *p* :: ′*a*::*idom poly*
shows *p*≠*0* ⟹ *finite* (*proots-within p s*)
unfolding *proots-within-def* **using** *poly-roots-finite* **by** *fast*

lemma *proots-within-pCons-1-iff*:
fixes *a*::′*a*::*idom*
shows *proots-within* [:−*a*,*1*:] *s* = (*if a*∈*s then* {*a*} *else* {})
proots-within [:*a*,−*1*:] *s* = (*if a*∈*s then* {*a*} *else* {})
by (*cases a*∈*s*,*auto*)

lemma *proots-within-uminus*[*simp*]:
fixes *p* :: ′*a*::*comm-ring poly*
shows *proots-within* (− *p*) *s* = *proots-within p s*
by *auto*

```
lemma proots-within-smult:
  fixes a::'a::{semiring-no-zero-divisors,comm-semiring-0}
  assumes a≠0
  shows proots-within (smult a p) s = proots-within p s
  unfolding proots-within-def using assms by auto
```

1.7 Polynomial roots counting multiplicities.

```
definition proots-count::'a::idom poly ⇒ 'a set ⇒ nat where
  proots-count p s = (∑ r∈proots-within p s. order r p)

lemma proots-count-emtpy[simp]:proots-count p {} = 0
  unfolding proots-count-def by auto

lemma proots-count-times:
  fixes s :: 'a::idom set
  assumes p*q≠0
  shows proots-count (p*q) s = proots-count p s + proots-count q s
proof -
  define pts where pts=proots-within p s
  define qts where qts=proots-within q s
  have [simp]: finite pts finite qts
    using ‹p*q≠0› unfolding pts-def qts-def by auto
  have (∑ r∈pts ∪ qts. order r p) = (∑ r∈pts. order r p)
  proof (rule comm-monoid-add-class.sum.mono-neutral-cong-right,simp-all)
    show ∀ i∈pts ∪ qts − pts. order i p = 0
      unfolding pts-def qts-def proots-within-def using order-root by fastforce
  qed
  moreover have (∑ r∈pts ∪ qts. order r q) = (∑ r∈qts. order r q)
  proof (rule comm-monoid-add-class.sum.mono-neutral-cong-right,simp-all)
    show ∀ i∈pts ∪ qts − qts. order i q = 0
      unfolding pts-def qts-def proots-within-def using order-root by fastforce
  qed
  ultimately show ?thesis unfolding proots-count-def
    apply (simp add:proots-within-times order-mult[OF ‹p*q≠0›] sum.distrib)
    apply (fold pts-def qts-def)
    by auto
qed

lemma proots-count-power-n-n:
  shows proots-count ([:− a, 1:]^n) s = (if a∈s ∧ n>0 then n else 0)
proof -
  have proots-within ([:− a, 1:] ^ n) s= (if a∈s ∧ n>0 then {a} else {})
    unfolding proots-within-def by auto
  thus ?thesis unfolding proots-count-def using order-power-n-n by auto
qed

lemma degree-proots-count:
```

```
  fixes p::complex poly
  shows degree p = proots-count p UNIV
proof (induct degree p arbitrary:p )
  case 0
  then obtain c where c-def:p=[:c:] using degree-eq-zeroE by auto
  then show ?case unfolding proots-count-def
    apply (cases c=0)
    by (auto intro!:sum.infinite simp add:infinite-UNIV-char-0 order-0I)
next
  case (Suc n)
  then have degree p≠0 and p≠0 by auto
  obtain z where poly p z = 0
     using Fundamental-Theorem-Algebra.fundamental-theorem-of-algebra ‹degree
p≠0› constant-degree[of p]
    by auto
  define onez where onez=[:−z,1:]
  have [simp]: onez≠0 degree onez = 1 unfolding onez-def by auto
  obtain q where q-def:p= onez ∗ q
    using poly-eq-0-iff-dvd ‹poly p z = 0› dvdE unfolding onez-def by blast
  hence q≠0 using ‹p≠0› by auto
  hence n=degree q using degree-mult-eq[of onez q] ‹Suc n = degree p›
    apply (fold q-def)
    by auto
  hence degree q = proots-count q UNIV using Suc.hyps(1) by simp
  moreover have  Suc 0 = proots-count onez UNIV
    unfolding onez-def using proots-count-power-n-n[of z 1 UNIV]
    by auto
  ultimately show ?case
     unfolding q-def using degree-mult-eq[of onez q] proots-count-times[of onez q
UNIV] ‹q≠0›
    by auto
qed

lemma proots-count-smult:
  fixes a::'a::{semiring-no-zero-divisors,idom}
  assumes a≠0
  shows proots-count (smult a p) s= proots-count p s
proof (cases p=0)
  case True
  then show ?thesis by auto
next
  case False
  then show ?thesis
    unfolding proots-count-def
    using order-smult[OF assms] proots-within-smult[OF assms] by auto
qed


lemma proots-count-pCons-1-iff:
```

```
  fixes a::'a::idom
  shows proots-count [:−a,1:] s = (if a∈s then 1 else 0)
  unfolding proots-count-def
 by (cases a∈s,auto simp add:proots-within-pCons-1-iff order-power-n-n[of - 1,simplified])

lemma proots-count-uminus[simp]:
  proots-count (− p) s = proots-count p s
  unfolding proots-count-def by simp

lemma card-proots-within-leq:
  assumes p≠0
  shows proots-count p s ≥ card (proots-within p s) using assms
proof (induct rule:poly-root-induct[of - λx. x∈s])
  case 0
  then show ?case unfolding proots-within-def proots-count-def by auto
next
  case (no-roots p)
  then have proots-within p s = {} by auto
  then show ?case unfolding proots-count-def by auto
next
  case (root a p)
  have card (proots-within ([:− a, 1:] ∗ p) s)
      ≤ card (proots-within [:− a, 1:] s)+card (proots-within p s)
    unfolding proots-within-times by (auto simp add:card-Un-le)
  also have ... ≤ 1+ proots-count p s
  proof −
    have card (proots-within [:− a, 1:] s) ≤ 1
    proof (cases a∈s)
      case True
      then have proots-within [:− a, 1:] s = {a} by auto
      then show ?thesis by auto
    next
      case False
      then have proots-within [:− a, 1:] s = {} by auto
      then show ?thesis by auto
    qed
    moreover have card (proots-within p s) ≤ proots-count p s
      apply (rule root.hyps)
      using root by auto
    ultimately show ?thesis by auto
  qed
  also have ... = proots-count ([:− a,1:] ∗ p) s
    apply (subst proots-count-times)
    subgoal by (metis mult-eq-0-iff pCons-eq-0-iff root.prems zero-neq-one)
    using root by (auto simp add:proots-count-pCons-1-iff)
  finally have card (proots-within ([:− a, 1:] ∗ p) s) ≤ proots-count ([:− a, 1:] ∗
p) s .
  then show ?case
   by (metis (no-types, opaque-lifting) add.inverse-inverse add.inverse-neutral mi-
```

nus-pCons
    mult-minus-left proots-count-uminus proots-within-uminus)
qed

lemma *proots-count-0-imp-empty*:
  assumes *proots-count p s=0 p≠0*
  shows *proots-within p s* = {}
proof −
  have *card* (*proots-within p s*) = *0*
    using *card-proots-within-leq*[*OF* ‹*p≠0*›,*of s*] ‹*proots-count p s=0*› **by** *auto*
  moreover have *finite* (*proots-within p s*) **using** ‹*p≠0*› **by** *auto*
  ultimately show *?thesis* **by** *auto*
qed

lemma *proots-count-leq-degree*:
  assumes *p≠0*
  shows *proots-count p s≤ degree p* **using** *assms*
proof (*induct rule:poly-root-induct*[*of - λx. x∈s*])
  case *0*
  then show *?case* **by** *auto*
next
  case (*no-roots p*)
  then have *proots-within p s* = {} **by** *auto*
  then show *?case* **unfolding** *proots-count-def* **by** *auto*
next
  case (*root a p*)
  have *proots-count* ([:*a*, − *1*:] ∗ *p*) *s* = *proots-count* [:*a*, − *1*:] *s* + *proots-count p*
s
    apply (*subst proots-count-times*)
    using *root* **by** *auto*
  also have ... = *1* + *proots-count p s*
  proof −
    have *proots-count* [:*a*, − *1*:] *s* =*1*
      by (*metis* (*no-types*, *lifting*) *add.inverse-inverse add.inverse-neutral mi-*
nus-pCons
      proots-count-pCons-1-iff proots-count-uminus root.hyps(*1*))
    then show *?thesis* **by** *auto*
  qed
  also have ... ≤ *degree* ([:*a*,−*1*:] ∗ *p*)
    apply (*subst degree-mult-eq*)
    subgoal by *auto*
    subgoal using *root* **by** *auto*
    subgoal using *root* **by** (*simp add*: ‹*p* ≠ *0*›)
    done
  finally show *?case* **.**
qed

```
lemma proots-count-union-disjoint:
  assumes A ∩ B = {} p≠0
  shows proots-count p (A ∪ B) = proots-count p A + proots-count p B
  unfolding proots-count-def
  apply (subst proots-within-union[symmetric])
  apply (subst sum.union-disjoint)
  using assms by auto

lemma proots-count-cong:
  assumes order-eq:∀ x∈s. order x p = order x q and p≠0 and q≠0
  shows proots-count p s = proots-count q s unfolding proots-count-def
proof (rule sum.cong)
  have poly p x = 0 ⟷ poly q x = 0 when x∈s for x
    using order-eq that by (simp add: assms(2) assms(3) order-root)
  then show proots-within p s = proots-within q s by auto
  show ⋀x. x ∈ proots-within q s ⟹ order x p = order x q
    using order-eq by auto
qed

lemma proots-count-of-real:
  assumes p≠0
  shows proots-count (map-poly of-real p) ((of-real::-⇒'a::{real-algebra-1,idom}) `
s)
            = proots-count p s
proof –
  define k where k=(of-real::-⇒'a)
  have proots-within (map-poly of-real p) (k ` s) =k ` (proots-within p s)
   unfolding proots-within-def k-def by (auto simp add:of-real-poly-map-poly[symmetric])
  then have proots-count (map-poly of-real p) (k ` s)
                = (∑ r∈k ` (proots-within p s). order r (map-poly of-real p))
    unfolding proots-count-def by simp
  also have ... = sum ((λr. order r (map-poly of-real p)) ∘ k) (proots-within p s)
    apply (subst sum.reindex)
    unfolding k-def by (auto simp add: inj-on-def)
  also have ... = proots-count p s unfolding proots-count-def
    apply (rule sum.cong)
   unfolding k-def comp-def using ‹p≠0› by (auto simp add:map-poly-order-of-real)

  finally show ?thesis unfolding k-def .
qed


lemma proots-pcompose:
  fixes p q::'a::field poly
  assumes p≠0 degree q=1
  shows proots-count (pcompose p q) s = proots-count p (poly q ` s)
proof –
  obtain a b where ab:q=[:a,b:] b≠0
    using ‹degree q=1› degree-eq-oneE by metis
```

define *f* **where** $f=(\lambda y.\ (y-a)/b)$
have *f-eq*:*f* (*poly q x*) = *x poly q* (*f x*) = *x* **for** *x*
unfolding *f-def* **using** *ab* **by** *auto*
have *proots-count* ($p \circ_p q$) s = ($\sum r\in f$ ‘ *proots-within p* (*poly q* ‘ *s*). *order r* ($p \circ_p q$))
unfolding *proots-count-def*
apply (*rule arg-cong2*[**where** *f* =*sum*])
apply (*auto simp*:*poly-pcompose proots-within-def f-eq*)
by (*metis* (*mono-tags*, *lifting*) *f-eq*(*1*) *image-eqI mem-Collect-eq*)
also have ... = ($\sum x\in$*proots-within p* (*poly q* ‘ *s*). *order* (*f x*) ($p \circ_p q$))
apply (*subst sum.reindex*)
subgoal unfolding *f-def inj-on-def* **using** ‹$b\neq 0$› **by** *auto*
by *simp*
also have ... = ($\sum x\in$*proots-within p* (*poly q* ‘ *s*). *order x p*)
proof −
have $p \circ_p q \neq 0$ **using** *assms*(*1*) *assms*(*2*) *pcompose-eq-0* **by** *force*
moreover have *order* (*f x*) (q − [:*x*:]) = *1* **for** *x*
proof −
have *order* (*f x*) (q − [:*x*:]) = *order* (*f x*) (*smult b* [:−((*x* − *a*) / *b*),*1*:])
unfolding *f-def* **using** *ab* **by** *auto*
also have ... = *1*
apply (*subst order-smult*)
using ‹$b\neq 0$› **unfolding** *f-def* **by** *auto*
finally show *?thesis* **.**
qed
ultimately have *order* (*f x*) ($p \circ_p q$) = *order x p* **for** *x*
apply (*subst order-pcompose*)
using *f-eq* **by** *auto*
then show *?thesis* **by** *auto*
qed
also have ... = *proots-count p* (*poly q* ‘ *s*)
unfolding *proots-count-def* **by** *auto*
finally show *?thesis* **.**
qed

1.8 Composition of a polynomial and a rational function

definition *fcompose*::′*a* ::*field poly* ⇒ ′*a poly* ⇒ ′*a poly* ⇒ ′*a poly* **where**
fcompose p q r = *fst* (*fold-coeffs* (λa (*c*,*d*). (*d*∗[:*a*:] + *q* ∗ *c*,*r*∗*d*)) *p* (*0*,*1*))

lemma *fcompose-0* [*simp*]: *fcompose 0 q r* = *0*
by (*simp add*: *fcompose-def*)

lemma *fcompose-const*[*simp*]:*fcompose* [:*a*:] *q r* = [:*a*:]
unfolding *fcompose-def* **by** (*cases a*=*0*) *auto*

lemma *fcompose-pCons*:
fcompose (*pCons a p*) *q1 q2* = *smult a* (*q2*^(*degree* (*pCons a p*))) + *q1* ∗ *fcompose p q1 q2*

proof (*cases p=0*)
  case *False*
  define *ff* **where** *ff=(λa (c, d). (d ∗ [:a:] + q1 ∗ c, q2 ∗ d))*
  define *fc* **where** *fc=fold-coeffs ff p (0, 1)*
  have *snd-ff:snd fc = (if p=0 then 1 else q2^(degree p + 1))* **unfolding** *fc-def*
    apply (*induct p*)
    subgoal by *simp*
    subgoal for *a p*
      by (*auto simp add:ff-def split:if-splits prod.splits*)
    done

  have *fcompose (pCons a p) q1 q2 = fst (fold-coeffs ff (pCons a p) (0, 1))*
    unfolding *fcompose-def ff-def* **by** *simp*
  also have ... = *fst (ff a fc)*
    using *False* **unfolding** *fc-def* **by** *auto*
  also have ... = *snd fc ∗ [:a:] + q1 ∗ fst fc*
    unfolding *ff-def* **by** (*auto split:prod.splits*)
  also have ... = *smult a (q2^(degree (pCons a p))) + q1 ∗ fst fc*
    using *snd-ff False* **by** *auto*
  also have ... = *smult a (q2^(degree (pCons a p))) + q1 ∗ fcompose p q1 q2*
    unfolding *fc-def ff-def fcompose-def* **by** *simp*
  finally show *?thesis* **.**
qed *simp*

lemma *fcompose-uminus*:
  fcompose (−p) q r = − fcompose p q r
  by (*induct p*) (*auto simp:fcompose-pCons*)

lemma *fcompose-add-less*:
  assumes *degree p1 > degree p2*
  shows *fcompose (p1+p2) q1 q2*
          = fcompose p1 q1 q2 + q2^(degree p1−degree p2) ∗ fcompose p2 q1 q2
  using *assms*
proof (*induction p1 p2 rule: poly-induct2*)
  case (*pCons a1 p1 a2 p2*)
  have *?case* **when** *p2=0*
    using *that* **by** (*simp add:fcompose-pCons smult-add-left*)
  moreover have *?case* **when** *p2≠0 ¬ degree p2 < degree p1*
    using *that pCons(2)* **by** *auto*
  moreover have *?case* **when** *p2≠0 degree p2< degree p1*
  proof −
    define *d1 d2* **where** *d1=degree (pCons a1 p1)* **and** *d2=degree (pCons a2 p2)*
    define *fp1 fp2* **where** *fp1= fcompose p1 q1 q2* **and** *fp2=fcompose p2 q1 q2*

    have *fcompose (pCons a1 p1 + pCons a2 p2) q1 q2*
          = fcompose (pCons (a1+a2) (p1+p2)) q1 q2
      by *simp*
    also have ... = *smult (a1 + a2) (q2 ^ d1) + q1 ∗ fcompose (p1 + p2) q1 q2*
    proof −

```
      have degree (pCons (a1 + a2) (p1 + p2)) = d1
        unfolding d1-def using that degree-add-eq-left by fastforce
      then show ?thesis unfolding fcompose-pCons by simp
    qed
    also have ... = smult (a1 + a2) (q2 ^ d1) + q1 * (fp1 + q2 ^(d1 − d2) *
fp2)
    proof −
      have degree p1 − degree p2 = d1 − d2
        unfolding d1-def d2-def using that by simp
      then show ?thesis
        unfolding pCons(1)[OF that(2),folded fp1-def fp2-def] by simp
    qed
    also have ... = fcompose (pCons a1 p1) q1 q2 + q2 ^(d1 − d2)
                    * fcompose (pCons a2 p2) q1 q2
    proof −
      have d1 > d2 unfolding d1-def d2-def using that by auto
      then show ?thesis
        unfolding fcompose-pCons
        apply (fold d1-def d2-def fp1-def fp2-def)
        by (simp add:algebra-simps smult-add-left power-add[symmetric])
    qed
    finally show ?thesis unfolding d1-def d2-def .
  qed
  ultimately show ?case by blast
qed simp

lemma fcompose-add-eq:
  assumes degree p1 = degree p2
  shows q2^(degree p1 − degree (p1+p2)) * fcompose (p1+p2) q1 q2
          = fcompose p1 q1 q2 + fcompose p2 q1 q2
  using assms
proof (induction p1 p2 rule: poly-induct2)
  case (pCons a1 p1 a2 p2)
  have ?case when p1+p2=0
  proof −
    have p2=−p1 using that by algebra
    then show ?thesis by (simp add:fcompose-pCons fcompose-uminus smult-add-left)
  qed
  moreover have ?case when p1=0
  proof −
    have p2=0
      using pCons(2) that by (auto split:if-splits)
    then show ?thesis using that by simp
  qed
  moreover have ?case when p1≠0 p1+p2≠0
  proof −
    define d1 d2 dp where d1=degree (pCons a1 p1) and d2=degree (pCons a2
p2)
                      and dp = degree p1 − degree (p1+p2)
```

```
    define fp1 fp2 where fp1= fcompose p1 q1 q2 and fp2=fcompose p2 q1 q2

    have q2 ^(degree (pCons a1 p1) − degree (pCons a1 p1 + pCons a2 p2)) ∗
            fcompose (pCons a1 p1 + pCons a2 p2) q1 q2
              = q2 ^ dp ∗ fcompose (pCons (a1+a2) (p1 +p2)) q1 q2
      unfolding dp-def using that by auto
    also have ... = smult (a1 + a2) (q2 ^ d1) + q1 ∗ (q2 ^ dp ∗ fcompose (p1 +
p2) q1 q2)
    proof −
      have degree p1 ≥ degree (p1 + p2)
      by (metis degree-add-le degree-pCons-eq-if not-less-eq-eq order-refl pCons.prems
zero-le)
      then show ?thesis
        unfolding fcompose-pCons dp-def d1-def using that
        by (simp add:algebra-simps power-add[symmetric])
    qed
    also have ... =  smult (a1 + a2) (q2 ^ d1) + q1 ∗ (fp1 + fp2)
      apply (subst pCons(1)[folded dp-def fp1-def fp2-def])
        subgoal by (metis degree-pCons-eq-if diff-Suc-Suc diff-zero not-less-eq-eq
pCons.prems zero-le)
      subgoal by simp
      done
    also have ... = fcompose (pCons a1 p1) q1 q2 + fcompose (pCons a2 p2) q1
q2
    proof −
      have ∗:d1 = degree (pCons a2 p2)
        unfolding d1-def using pCons(2) by simp
      show ?thesis
        unfolding fcompose-pCons
        apply (fold d1-def fp1-def fp2-def ∗)
        by (simp add:smult-add-left algebra-simps)
    qed
    finally show ?thesis .
  qed
  ultimately show ?case by blast
qed simp

lemma fcompose-add-const:
  fcompose ([:a:] + p) q1 q2 = smult a (q2 ^ degree p) + fcompose p q1 q2
  apply (cases p)
  by (auto simp add:fcompose-pCons smult-add-left)

lemma fcompose-smult: fcompose (smult a p) q1 q2 = smult a (fcompose p q1 q2)
  by (induct p) (simp-all add:fcompose-pCons smult-add-right)

lemma fcompose-mult: fcompose (p1∗p2) q1 q2 = fcompose p1 q1 q2 ∗ fcompose
p2 q1 q2
proof (induct p1)
  case 0
```

```
  then show ?case by simp
next
  case (pCons a p1)
  have ?case when p1=0 ∨ p2=0
    using that by (auto simp add:fcompose-smult)
  moreover have ?case when p1≠0 p2≠0 a=0
    using that by (simp add:fcompose-pCons pCons)
  moreover have ?case when p1≠0 p2≠0 a≠0
  proof −
    have fcompose (pCons a p1 ∗ p2) q1 q2
            = fcompose (pCons 0 (p1 ∗ p2) + smult a p2) q1 q2
      by (simp add:algebra-simps)
    also have ... =  fcompose (pCons 0 (p1 ∗ p2)) q1 q2
                        + q2 ^ (degree p1 +1) ∗ fcompose (smult a p2) q1 q2
    proof −
      have degree (pCons 0 (p1 ∗ p2)) > degree (smult a p2)
        using that by (simp add: degree-mult-eq)
      from fcompose-add-less[OF this,of q1 q2] that
      show ?thesis by (simp add:degree-mult-eq)
    qed
    also have ... = fcompose (pCons a p1) q1 q2 ∗ fcompose p2 q1 q2
     using that by (simp add:fcompose-pCons fcompose-smult pCons algebra-simps)
    finally show ?thesis .
  qed
  ultimately show ?case by blast
qed

lemma fcompose-poly:
  assumes poly q2 x≠0
  shows poly p (poly q1 x/poly q2 x) = poly (fcompose p q1 q2) x / poly (q2^(degree
p)) x
  apply (induct p)
  using assms by (simp-all add:fcompose-pCons field-simps)

lemma poly-fcompose:
   assumes poly q2 x≠0
    shows poly (fcompose p q1 q2) x = poly p (poly q1 x/poly q2 x) ∗ (poly q2
x)^(degree p)
  using fcompose-poly[OF assms] assms by (auto simp add:field-simps)
lemma poly-fcompose-0-denominator:
  assumes poly q2 x=0
  shows poly (fcompose p q1 q2) x = poly q1 x ^ degree p ∗ lead-coeff p
  apply (induct p)
  using assms by (auto simp add:fcompose-pCons)

lemma fcompose-0-denominator:fcompose p q1 0 = smult (lead-coeff p) (q1^degree
p)
  apply (induct p)
  by (auto simp:fcompose-pCons)
```

```
lemma fcompose-nzero:
  fixes p::'a::field poly
  assumes p≠0 and q2≠0 and nconst:∀ c. q1 ≠ smult c q2
      and infi:infinite (UNIV::'a set)
  shows fcompose p q1 q2 ≠ 0 using ‹p≠0›
proof (induct p rule:poly-root-induct-alt)
  case 0
  then show ?case by simp
next
  case (no-proots p)
  have False when fcompose p q1 q2 = 0
  proof -
    obtain x where poly q2 x≠0
    proof -
      have finite (proots q2) using ‹q2≠0› by auto
      then have ∃ x. poly q2 x≠0
        by (meson UNIV-I ex-new-if-finite infi proots-withinI)
      then show ?thesis using that by auto
    qed
    define y where y = poly q1 x / poly q2 x
    have poly p y = 0
     using ‹fcompose p q1 q2 = 0› fcompose-poly[OF ‹poly q2 x≠0›,of p q1,folded
y-def]
      by simp
    then show False using no-proots(1) by auto
  qed
  then show ?case by auto
next
  case (root a p)
  have fcompose [:− a, 1:] q1 q2 ≠ 0
    unfolding fcompose-def using nconst[rule-format,of a]
    by simp
  moreover have fcompose p q1 q2 ≠ 0
    using root by fastforce
  ultimately show ?case unfolding fcompose-mult by auto
qed
```

1.9 Bijection (*bij-betw*) and the number of polynomial roots

```
lemma proots-fcompose-bij-eq:
  fixes p::'a::field poly
  assumes bij:bij-betw (λx. poly q1 x/poly q2 x) A B and p≠0
      and nzero:∀ x∈A. poly q2 x≠0
      and max-deg: max (degree q1) (degree q2) ≤ 1
      and nconst:∀ c. q1 ≠ smult c q2
      and infi:infinite (UNIV::'a set)
  shows proots-count p B = proots-count (fcompose p q1 q2) A
  using ‹p≠0›
```

```
proof (induct p rule:poly-root-induct-alt)
  case 0
  then show ?case by simp
next
  case (no-proots p)
  have proots-count p B = 0
  proof -
    have proots-within p B = {}
      using no-proots by auto
    then show ?thesis unfolding proots-count-def by auto
  qed
  moreover have proots-count (fcompose p q1 q2) A = 0
  proof -
    have proots-within (fcompose p q1 q2) A = {}
      using no-proots unfolding proots-within-def
      by (smt (verit) div-0 empty-Collect-eq fcompose-poly nzero)
    then show ?thesis unfolding proots-count-def by auto
  qed
  ultimately show ?case by auto
next
  case (root b p)
  have proots-count ([:− b, 1:] ∗ p) B = proots-count [:− b, 1:] B + proots-count
p B
    using proots-count-times[OF ‹[:− b, 1:] ∗ p ≠ 0›] by simp
  also have ... = proots-count (fcompose [:− b, 1:] q1 q2) A
                  + proots-count (fcompose p q1 q2) A
  proof -
    define g where g=(λx. poly q1 x/poly q2 x)

    have proots-count [:− b, 1:] B = proots-count (fcompose [:− b, 1:] q1 q2) A
    proof (cases b∈B)
      case True
      then have proots-count [:− b, 1:] B = 1
        unfolding proots-count-pCons-1-iff by simp
      moreover have proots-count (fcompose [:− b, 1:] q1 q2) A = 1
      proof -
        obtain a where b=g a a∈A
          using bij[folded g-def] True
          by (metis bij-betwE bij-betw-the-inv-into f-the-inv-into-f-bij-betw)
        define qq where qq=q1 − smult b q2
        have qq-0:poly qq a=0 and qq-deg: degree qq≤1 and ‹qq≠0›
          unfolding qq-def
           subgoal using ‹b=g a› nzero[rule-format,OF ‹a∈A›] unfolding g-def by
auto
          subgoal using max-deg by (simp add: degree-diff-le)
          subgoal using nconst[rule-format,of b] by auto
          done
        have proots-within qq A = {a}
        proof -
```

```
        have a∈proots-within qq A
          using qq-0 ‹a∈A› by auto
        moreover have card (proots-within qq A) = 1
        proof –
          have finite (proots-within qq A) using ‹qq≠0› by simp
          moreover have proots-within qq A ≠ {}
            using ‹a∈proots-within qq A› by auto
          ultimately have card (proots-within qq A) ≠0 by auto
          moreover have card (proots-within qq A) ≤ 1
         by (meson ‹qq ≠ 0› card-proots-within-leq le-trans proots-count-leq-degree
qq-deg)
          ultimately show ?thesis by auto
        qed
        ultimately show ?thesis by (metis card-1-singletonE singletonD)
      qed
      moreover have order a qq=1
         by (metis One-nat-def ‹qq ≠ 0› le-antisym le-zero-eq not-less-eq-eq or-
der-degree
            order-root qq-0 qq-deg)
      ultimately show ?thesis unfolding fcompose-def proots-count-def qq-def
        by auto
    qed
    ultimately show ?thesis by auto
  next
    case False
    then have proots-count [:− b, 1:] B  = 0
      unfolding proots-count-pCons-1-iff by simp
    moreover have proots-count (fcompose [:− b, 1:] q1 q2) A = 0
    proof –
      have proots-within (fcompose [:− b, 1:] q1 q2) A = {}
      proof (rule ccontr)
        assume proots-within (fcompose [:− b, 1:] q1 q2) A ≠ {}
        then obtain a where a∈A poly q1 a = b ∗ poly q2 a
          unfolding fcompose-def proots-within-def by auto
        then have b = g a
          unfolding g-def using nzero[rule-format,OF ‹a∈A›] by auto
        then have b∈B using ‹a∈A› bij[folded g-def] using bij-betwE by blast
        then show False using False by auto
      qed
      then show ?thesis unfolding proots-count-def by auto
    qed
    ultimately show ?thesis by simp
  qed
  moreover have proots-count p B = proots-count (fcompose p q1 q2) A
    apply (rule root.hyps)
    using mult-eq-0-iff root.prems by blast
  ultimately show ?thesis by auto
 qed
 also have ... = proots-count (fcompose ([:− b, 1:] ∗ p) q1 q2) A
```

```
  proof (cases A={})
    case False
    have fcompose [:− b, 1:] q1 q2 ≠0
      using nconst[rule-format,of b] unfolding fcompose-def by auto
    moreover have fcompose p q1 q2 ≠ 0
      apply (rule fcompose-nzero[OF - - nconst infi])
      subgoal using ‹[:− b, 1:] ∗ p ≠ 0› by auto
      subgoal using nzero False by auto
      done
    ultimately show ?thesis unfolding fcompose-mult
      apply (subst proots-count-times)
      by auto
  qed auto
  finally show ?case .
qed

lemma proots-card-fcompose-bij-eq:
  fixes p::'a::field poly
  assumes bij:bij-betw (λx. poly q1 x/poly q2 x) A B and p≠0
      and nzero:∀ x∈A. poly q2 x≠0
      and max-deg: max (degree q1) (degree q2) ≤ 1
      and nconst:∀ c. q1 ≠ smult c q2
      and infi:infinite (UNIV::'a set)
  shows card (proots-within p B) = card (proots-within (fcompose p q1 q2) A)
  using ‹p≠0›
proof (induct p rule:poly-root-induct-alt)
  case 0
  then show ?case by simp
next
  case (no-proots p)
  have proots-within p B = {} using no-proots by auto
  moreover have proots-within (fcompose p q1 q2) A = {}
    using no-proots fcompose-poly
    by (smt (verit) Collect-empty-eq divide-eq-0-iff nzero proots-within-def)
  ultimately show ?case by auto
next
  case (root b p)
  then have [simp]:p≠0 by auto

  have ?case when b∉B ∨ poly p b=0
  proof −
    have proots-within ([:− b, 1:] ∗ p) B = proots-within p B
      using that by auto
    moreover have proots-within (fcompose ([:− b, 1:] ∗ p) q1 q2) A
        = proots-within (fcompose p q1 q2) A
      using that nzero unfolding fcompose-mult proots-within-times
      apply (auto simp add: poly-fcompose)
      using bij bij-betwE by blast
    ultimately show ?thesis using root by auto
```

```
  qed
  moreover have ?case when b∈B poly p b≠0
  proof −
    define bb where bb=[:− b, 1:]
    have card (proots-within (bb ∗ p) B) = card {b} + card (proots-within p B)
    proof −
      have proots-within bb B = {b}
        using that unfolding bb-def by auto
      then show ?thesis unfolding proots-within-times
        apply (subst card-Un-disjoint)
        by (use that in auto)
    qed
    also have ... = 1 + card (proots-within (fcompose p q1 q2) A)
      using root.hyps by simp
    also have ... = card (proots-within (fcompose (bb ∗ p) q1 q2) A)
      unfolding proots-within-times fcompose-mult
    proof (subst card-Un-disjoint)
      obtain a where b-poly:b=poly q1 a / poly q2 a and a∈A
        by (metis (no-types, lifting) ‹b ∈ B› bij bij-betwE bij-betw-the-inv-into
            f-the-inv-into-f-bij-betw)
      define bbq pq where bbq=fcompose bb q1 q2 and pq=fcompose p q1 q2
      have bbq-0:poly bbq a=0 and bbq-deg: degree bbq≤1 and bbq≠0
        unfolding bbq-def bb-def
        subgoal using ‹a ∈ A› b-poly nzero poly-fcompose by fastforce
         subgoal by (metis (no-types, lifting) degree-add-le degree-pCons-eq-if de-
gree-smult-le
            dual-order.trans fcompose-const fcompose-pCons max.boundedE max-deg
mult-cancel-left2
           one-neq-zero one-poly-eq-simps(1) power.simps)
          subgoal by (metis ‹a ∈ A› ‹poly (fcompose [:− b, 1:] q1 q2) a = 0›
fcompose-nzero infi
                nconst nzero one-neq-zero pCons-eq-0-iff)
        done
      show finite (proots-within bbq A) using ‹bbq≠0› by simp
      show finite (proots-within pq A) unfolding pq-def
         by (metis ‹a ∈ A› ‹p ≠ 0› fcompose-nzero finite-proots infi nconst nzero
poly-0 pq-def)
      have bbq-a:proots-within bbq A = {a}
      proof −
        have a∈proots-within bbq A
          by (simp add: ‹a ∈ A› bbq-0)
        moreover have card (proots-within bbq A) = 1
        proof −
          have card (proots-within bbq A) ≠0
            using ‹a∈proots-within bbq A› ‹finite (proots-within bbq A)›
            by auto
          moreover have card (proots-within bbq A) ≤ 1
          by (meson ‹bbq ≠ 0› card-proots-within-leq le-trans proots-count-leq-degree
bbq-deg)
```

ultimately show *?thesis* **by** *auto*
qed
ultimately show *?thesis* **by** (*metis card-1-singletonE singletonD*)
qed
show *proots-within* (*bbq*) *A* ∩ *proots-within* (*pq*) *A* = {}
using *b-poly bbq-a fcompose-poly nzero pq-def that*(*2*) **by** *fastforce*
show *1* + *card* (*proots-within pq A*) = *card* (*proots-within bbq A*) + *card* (*proots-within pq A*)
using *bbq-a* **by** *simp*
qed
finally show *?thesis* **unfolding** *bb-def* **.**
qed
ultimately show *?case* **by** *auto*
qed

lemma *proots-pcompose-bij-eq*:
fixes *p*::′*a*::*idom poly*
assumes *bij*:*bij-betw* (*λx. poly q x*) *A B* **and** *p≠0*
and *q-deg*: *degree q* = *1*
shows *proots-count p B* = *proots-count* ($p \circ_p q$) *A* **using** ‹*p≠0*›
proof (*induct p rule*:*poly-root-induct-alt*)
case *0*
then show *?case* **by** *simp*
next
case (*no-proots p*)
have *proots-count p B* = *0*
proof −
have *proots-within p B* = {}
using *no-proots* **by** *auto*
then show *?thesis* **unfolding** *proots-count-def* **by** *auto*
qed
moreover have *proots-count* ($p \circ_p q$) *A* = *0*
proof −
have *proots-within* ($p \circ_p q$) *A* = {}
using *no-proots* **unfolding** *proots-within-def*
by (*auto simp*:*poly-pcompose*)
then show *?thesis* **unfolding** *proots-count-def* **by** *auto*
qed
ultimately show *?case* **by** *auto*
next
case (*root b p*)
have *proots-count* ([:− *b*, *1*:] ∗ *p*) *B* = *proots-count* [:− *b*, *1*:] *B* + *proots-count p B*
using *proots-count-times*[*OF* ‹[:− *b*, *1*:] ∗ *p* ≠ *0*›] **by** *simp*
also have ... = *proots-count* ([:− *b*, *1*:] $\circ_p$ *q*) *A* + *proots-count* ($p \circ_p q$) *A*
proof −
have *proots-count* [:− *b*, *1*:] *B* = *proots-count* ([:− *b*, *1*:] $\circ_p$ *q*) *A*
proof (*cases b∈B*)
case *True*

```
    then have proots-count [:− b, 1:] B = 1
      unfolding proots-count-pCons-1-iff by simp
    moreover have proots-count ([:− b, 1:] ∘p q) A = 1
    proof −
      obtain a where b=poly q a a∈A
     using True bij by (metis bij-betwE bij-betw-the-inv-into f-the-inv-into-f-bij-betw)
      define qq where qq=[:− b:] + q
      have qq-0:poly qq a=0 and qq-deg: degree qq≤1 and ‹qq≠0›
        unfolding qq-def
        subgoal using ‹b=poly q a› by auto
        subgoal using q-deg by (simp add: degree-add-le)
        subgoal using q-deg add.inverse-unique by force
        done
      have proots-within qq A = {a}
      proof −
        have a∈proots-within qq A
          using qq-0 ‹a∈A› by auto
        moreover have card (proots-within qq A) = 1
        proof −
          have finite (proots-within qq A) using ‹qq≠0› by simp
          moreover have proots-within qq A ≠ {}
            using ‹a∈proots-within qq A› by auto
          ultimately have card (proots-within qq A) ≠0 by auto
          moreover have card (proots-within qq A) ≤ 1
          by (meson ‹qq ≠ 0› card-proots-within-leq le-trans proots-count-leq-degree
qq-deg)
          ultimately show ?thesis by auto
        qed
        ultimately show ?thesis by (metis card-1-singletonE singletonD)
      qed
      moreover have order a qq=1
          by (metis One-nat-def ‹qq ≠ 0› le-antisym le-zero-eq not-less-eq-eq or-
der-degree
              order-root qq-0 qq-deg)
      ultimately show ?thesis unfolding pcompose-def proots-count-def qq-def
        by auto
    qed
    ultimately show ?thesis by auto
  next
    case False
    then have proots-count [:− b, 1:] B  = 0
      unfolding proots-count-pCons-1-iff by simp
    moreover have proots-count ([:− b, 1:] ∘p q) A = 0
    proof −
      have proots-within ([:− b, 1:] ∘p q) A = {}
        unfolding pcompose-def
        apply auto
        using False bij bij-betwE by blast
      then show ?thesis unfolding proots-count-def by auto
```

```
      qed
      ultimately show ?thesis by simp
    qed
    moreover have proots-count p B = proots-count (p ∘p q) A
      apply (rule root.hyps)
      using ‹[:− b, 1:] ∗ p ≠ 0› by auto
    ultimately show ?thesis by auto
  qed
  also have ... = proots-count (([:− b, 1:] ∗ p) ∘p q) A
    unfolding pcompose-mult
    apply (subst proots-count-times)
     subgoal by (metis (no-types, lifting) One-nat-def add.right-neutral degree-0
degree-mult-eq
     degree-pCons-eq-if degree-pcompose mult-eq-0-iff one-neq-zero one-pCons pcom-
pose-mult
      q-deg root.prems)
    by simp
  finally show ?case .
qed

lemma proots-card-pcompose-bij-eq:
  fixes p::'a::idom poly
  assumes bij:bij-betw (λx. poly q x) A B and p≠0
      and q-deg: degree q = 1
  shows card (proots-within p B) = card (proots-within (p ∘p q) A) using ‹p≠0›
proof (induct p rule:poly-root-induct-alt)
  case 0
  then show ?case by auto
next
  case (no-proots p)
  have proots-within p B = {} using no-proots by auto
  moreover have proots-within (p ∘p q) A = {} using no-proots
    by (simp add: poly-pcompose proots-within-def)
  ultimately show ?case by auto
next
  case (root b p)
  then have [simp]:p≠0 by auto
  have ?case when b∉B ∨ poly p b=0
  proof −
    have proots-within ([:− b, 1:] ∗ p) B = proots-within p B
      using that by auto
     moreover have proots-within (([:− b, 1:] ∗ p) ∘p q) A = proots-within (p ∘p
q) A
      using that unfolding pcompose-mult proots-within-times
      apply (auto simp add: poly-pcompose)
      using bij bij-betwE by blast
    ultimately show ?thesis using root.hyps[OF ‹p≠0›] by auto
  qed
  moreover have ?case when b∈B poly p b≠0
```

proof −
  define *bb* **where** *bb*=[:− *b*, *1*:]
  have *card* (*proots-within* (*bb* ∗ *p*) *B*) = *card* {*b*} + *card* (*proots-within p B*)
  proof −
    have *proots-within bb B* = {*b*}
      using *that* **unfolding** *bb-def* **by** *auto*
    then show *?thesis* **unfolding** *proots-within-times*
      apply (*subst card-Un-disjoint*)
      by (*use that* **in** *auto*)
  qed
  also have ... = *1* + *card* (*proots-within* (*p* $\circ_p$ *q*) *A*)
    using *root.hyps* **by** *simp*
  also have ... = *card* (*proots-within* ((*bb* ∗ *p*) $\circ_p$ *q*) *A*)
    unfolding *proots-within-times pcompose-mult*
  proof (*subst card-Un-disjoint*)
    obtain *a* **where** *b*=*poly q a a*∈*A*
     by (*metis* ‹*b* ∈ *B*› *bij bij-betwE bij-betw-the-inv-into f-the-inv-into-f-bij-betw*)
    define *bbq pq* **where** *bbq*=*bb* $\circ_p$ *q* **and** *pq*=*p* $\circ_p$ *q*
    have *bbq-0*:*poly bbq a*=*0* **and** *bbq-deg*: *degree bbq*≤*1* **and** *bbq*≠*0*
      unfolding *bbq-def bb-def poly-pcompose*
      subgoal using ‹*b*=*poly q a*› **by** *auto*
      subgoal using *q-deg* **by** (*simp add*: *degree-add-le degree-pcompose*)
      subgoal using *pcompose-eq-0 q-deg* **by** *fastforce*
      done
    show *finite* (*proots-within bbq A*) **using** ‹*bbq*≠*0*› **by** *simp*
    show *finite* (*proots-within pq A*) **unfolding** *pq-def*
      by (*metis* ‹*p* ≠ *0*› *finite-proots pcompose-eq-0 q-deg zero-less-one*)
    have *bbq-a*:*proots-within bbq A* = {*a*}
    proof −
      have *a*∈*proots-within bbq A*
        unfolding *bb-def proots-within-def poly-pcompose bbq-def*
        using ‹*b*=*poly q a*› ‹*a*∈*A*› **by** *simp*
      moreover have *card* (*proots-within bbq A*) = *1*
      proof −
        have *card* (*proots-within bbq A*) ≠*0*
          using ‹*a*∈*proots-within bbq A*› ‹*finite* (*proots-within bbq A*)›
          by *auto*
        moreover have *card* (*proots-within bbq A*) ≤ *1*
        by (*meson* ‹*bbq* ≠ *0*› *card-proots-within-leq le-trans proots-count-leq-degree bbq-deg*)
        ultimately show *?thesis* **by** *auto*
      qed
      ultimately show *?thesis* **by** (*metis card-1-singletonE singletonD*)
    qed
    show *proots-within* (*bbq*) *A* ∩ *proots-within* (*pq*) *A* = {}
    using *bbq-a* ‹*b* = *poly q a*› *that*(*2*) **unfolding** *pq-def* **by** (*simp add*:*poly-pcompose*)
     show *1* + *card* (*proots-within pq A*) = *card* (*proots-within bbq A*) + *card* (*proots-within pq A*)
      using *bbq-a* **by** *simp*

```
    qed
    finally show ?thesis unfolding bb-def .
  qed
  ultimately show ?case by auto
qed

end
```

2 Budan–Fourier theorem

```
theory Budan-Fourier imports
  BF-Misc
begin
```

The Budan–Fourier theorem is a classic result in real algebraic geometry to over-approximate real roots of a polynomial (counting multiplicity) within an interval. When all roots of the the polynomial are known to be real, the over-approximation becomes tight – the number of roots are counted exactly. Also note that Descartes' rule of sign is a direct consequence of the Budan–Fourier theorem.

The proof mainly follows Theorem 2.35 in Basu, S., Pollack, R., Roy, M.-F.: Algorithms in Real Algebraic Geometry. Springer Berlin Heidelberg, Berlin, Heidelberg (2006).

2.1 More results related to *sign-r-pos*

```
lemma sign-r-pos-nzero-right:
  assumes nzero:∀ x. c<x ∧ x≤d ⟶ poly p x ≠0 and c<d
  shows if sign-r-pos p c then poly p d>0 else poly p d<0
proof (cases sign-r-pos p c)
  case True
  then obtain d' where d'>c and d'-pos:∀ y>c. y < d' ⟶ 0 < poly p y
    unfolding sign-r-pos-def eventually-at-right by auto
  have False when ¬ poly p d>0
  proof –
    have ∃ x>(c + min d d') / 2. x < d ∧ poly p x = 0
      apply (rule poly-IVT-neg)
      using ‹d'>c› ‹c<d› that nzero[rule-format,of d,simplified]
      by (auto intro:d'-pos[rule-format])
    then show False using nzero ‹c < d'› by auto
  qed
  then show ?thesis using True by auto
next
  case False
  then have sign-r-pos (−p) c
    using sign-r-pos-minus[of p c] nzero[rule-format,of d,simplified] ‹c<d›
    by fastforce
  then obtain d' where d'>c and d'-neg:∀ y>c. y < d' ⟶ 0 > poly p y
```

```
    unfolding sign-r-pos-def eventually-at-right by auto
  have False when ¬ poly p d<0
  proof −
    have ∃x>(c + min d d') / 2. x < d ∧ poly p x = 0
      apply (rule poly-IVT-pos)
      using ‹d'>c› ‹c<d› that nzero[rule-format,of d,simplified]
      by (auto intro:d'-neg[rule-format])
    then show False using nzero ‹c < d'› by auto
  qed
  then show ?thesis using False by auto
qed

lemma sign-r-pos-at-left:
  assumes p≠0
   shows if even (order c p) ⟷sign-r-pos p c then eventually (λx. poly p x>0)
(at-left c)
          else eventually (λx. poly p x<0) (at-left c)
  using assms
proof (induct p rule:poly-root-induct-alt)
  case 0
  then show ?case by simp
next
  case (no-proots p)
  then have [simp]:order c p = 0 using order-root by blast
  have ?case when poly p c >0
  proof −
    have ∀F x in at c. 0 < poly p x
      using that
      by (metis (no-types, lifting) less-linear no-proots.hyps not-eventuallyD
          poly-IVT-neg poly-IVT-pos)
    then have ∀F x in at-left c. 0 < poly p x
      using eventually-at-split by blast
    moreover have sign-r-pos p c using sign-r-pos-rec[OF ‹p≠0›] that by auto
    ultimately show ?thesis by simp
  qed
  moreover have ?case when poly p c <0
  proof −
    have ∀F x in at c. poly p x < 0
      using that
      by (metis (no-types, lifting) less-linear no-proots.hyps not-eventuallyD
          poly-IVT-neg poly-IVT-pos)
    then have ∀F x in at-left c. poly p x < 0
      using eventually-at-split by blast
    moreover have ¬ sign-r-pos p c using sign-r-pos-rec[OF ‹p≠0›] that by auto
    ultimately show ?thesis by simp
  qed
  ultimately show ?case using no-proots(1)[of c] by argo
next
  case (root a p)
```

```
  define aa where aa=[:−a,1:]
  have [simp]:aa≠0 p≠0 using ‹[:− a, 1:] ∗ p ≠ 0› unfolding aa-def by auto
  have ?case when c>a
  proof −
    have ?thesis = (if even (order c p) = sign-r-pos p c
            then ∀F x in at-left c. 0 < poly (aa ∗ p) x
            else ∀F x in at-left c. poly (aa ∗ p) x < 0)
    proof −
      have order c aa=0 unfolding aa-def using order-0I that by force
      then have even (order c (aa ∗ p)) = even (order c p)
        by (subst order-mult) auto
      moreover have sign-r-pos aa c
        unfolding aa-def using that
        by (auto simp: sign-r-pos-rec)
      then have sign-r-pos (aa ∗ p) c = sign-r-pos p c
        by (subst sign-r-pos-mult) auto
      ultimately show ?thesis
        by (fold aa-def) auto
    qed
    also have ... = (if even (order c p) = sign-r-pos p c
            then ∀F x in at-left c. 0 < poly p x
            else ∀F x in at-left c. poly p x < 0)
    proof −
      have ∀F x in at-left c. 0 < poly aa x
        apply (simp add:aa-def)
        using that eventually-at-left-field by blast
      then have (∀F x in at-left c. 0 < poly (aa ∗ p) x) ⟷ (∀F x in at-left c. 0
< poly p x)
       (∀F x in at-left c. 0 > poly (aa ∗ p) x) ⟷ (∀F x in at-left c. 0 > poly p x)
        apply auto
       by (erule (1) eventually-elim2,simp add: zero-less-mult-iff mult-less-0-iff)+
      then show ?thesis by simp
    qed
    also have ... using root.hyps by simp
    finally show ?thesis .
  qed
  moreover have ?case when c<a
  proof −
    have ?thesis = (if even (order c p) = sign-r-pos p c
            then ∀F x in at-left c. poly (aa ∗ p) x < 0
            else ∀F x in at-left c. 0 < poly (aa ∗ p) x)
    proof −
      have order c aa=0 unfolding aa-def using order-0I that by force
      then have even (order c (aa ∗ p)) = even (order c p)
        by (subst order-mult) auto
      moreover have ¬ sign-r-pos aa c
        unfolding aa-def using that
        by (auto simp: sign-r-pos-rec)
      then have sign-r-pos (aa ∗ p) c = (¬ sign-r-pos p c)
```

```
      by (subst sign-r-pos-mult) auto
    ultimately show ?thesis
      by (fold aa-def) auto
  qed
  also have ... = (if even (order c p) = sign-r-pos p c
        then ∀F x in at-left c. 0 < poly p x
        else ∀F x in at-left c. poly p x < 0)
  proof -
    have ∀F x in at-left c. poly aa x < 0
      apply (simp add:aa-def)
      using that eventually-at-filter by fastforce
     then have (∀F x in at-left c. 0 < poly (aa * p) x) ⟷ (∀F x in at-left c.
poly p x < 0)
       (∀F x in at-left c. 0 > poly (aa * p) x) ⟷ (∀F x in at-left c. 0 < poly p x)
      apply auto
      by (erule (1) eventually-elim2,simp add: zero-less-mult-iff mult-less-0-iff)+
    then show ?thesis by simp
  qed
  also have ... using root.hyps by simp
  finally show ?thesis .
qed
moreover have ?case when c=a
proof -
  have ?thesis = (if even (order c p) = sign-r-pos p c
        then ∀F x in at-left c. 0 > poly (aa * p) x
        else ∀F x in at-left c. poly (aa * p) x > 0)
  proof -
    have order c aa=1 unfolding aa-def using that
      by (metis order-power-n-n power-one-right)
    then have even (order c (aa * p)) = odd (order c p)
      by (subst order-mult) auto
    moreover have sign-r-pos aa c
      unfolding aa-def using that
      by (auto simp: sign-r-pos-rec pderiv-pCons)
    then have sign-r-pos (aa * p) c = sign-r-pos p c
      by (subst sign-r-pos-mult) auto
    ultimately show ?thesis
      by (fold aa-def) auto
  qed
  also have ... = (if even (order c p) = sign-r-pos p c
        then ∀F x in at-left c. 0 < poly p x
        else ∀F x in at-left c. poly p x < 0)
  proof -
    have ∀F x in at-left c. 0 > poly aa x
      apply (simp add:aa-def)
      using that by (simp add: eventually-at-filter)
    then have (∀F x in at-left c. 0 < poly (aa * p) x) ⟷ (∀F x in at-left c. 0
> poly p x)
       (∀F x in at-left c. 0 > poly (aa * p) x) ⟷ (∀F x in at-left c. 0 < poly p x)
```

```
      apply auto
      by (erule (1) eventually-elim2,simp add: zero-less-mult-iff mult-less-0-iff)+
    then show ?thesis by simp
  qed
  also have ... using root.hyps by simp
  finally show ?thesis .
 qed
 ultimately show ?case by argo
qed

lemma sign-r-pos-nzero-left:
 assumes nzero:∀x. d≤x ∧ x<c ⟶ poly p x ≠0 and d<c
 shows if even (order c p) ⟷sign-r-pos p c then poly p d>0 else poly p d<0
proof (cases even (order c p) ⟷sign-r-pos p c)
 case True
 then have eventually (λx. poly p x>0) (at-left c)
  using nzero[rule-format,of d,simplified] ‹d<c› sign-r-pos-at-left
  by (simp add: order-root)
 then obtain d' where d'<c and d'-pos:∀y>d'. y < c ⟶ 0 < poly p y
  unfolding eventually-at-left by auto
 have False when ¬ poly p d>0
 proof −
  have ∃x>d. x < (c + max d d') / 2 ∧ poly p x = 0
   apply (rule poly-IVT-pos)
   using ‹d'<c› ‹c>d› that nzero[rule-format,of d,simplified]
   by (auto intro:d'-pos[rule-format])
  then show False using nzero ‹c > d'› by auto
 qed
 then show ?thesis using True by auto
next
 case False
 then have eventually (λx. poly p x<0) (at-left c)
  using nzero[rule-format,of d,simplified] ‹d<c› sign-r-pos-at-left
  by (simp add: order-root)
 then obtain d' where d'<c and d'-neg:∀y>d'. y < c ⟶ 0 > poly p y
  unfolding eventually-at-left by auto
 have False when ¬ poly p d<0
 proof −
  have ∃x>d. x < (c + max d d') / 2 ∧ poly p x = 0
   apply (rule poly-IVT-neg)
   using ‹d'<c› ‹c>d› that nzero[rule-format,of d,simplified]
   by (auto intro:d'-neg[rule-format])
  then show False using nzero ‹c > d'› by auto
 qed
 then show ?thesis using False by auto
qed
```

2.2 Fourier sequences

function *pders::real poly ⇒ real poly list* **where**
  pders p = (if p =0 then [] else Cons p (pders (pderiv p)))
  by *auto*
termination
  apply *(relation measure (λp. if p=0 then 0 else degree p + 1))*
  by *(auto simp:degree-pderiv pderiv-eq-0-iff)*

declare *pders.simps[simp del]*

lemma *set-pders-nzero*:
  assumes *p≠0 q∈set (pders p)*
  shows *q≠0*
  using *assms*
proof *(induct p rule:pders.induct)*
  case *(1 p)*
  then have *q ∈ set (p # pders (pderiv p))*
    by *(simp add: pders.simps)*
  then have *q=p ∨ q∈set (pders (pderiv p))* **by** *auto*
  moreover have *?case* **when** *q=p*
    using *that ‹p≠0›* **by** *auto*
  moreover have *?case* **when** *q∈set (pders (pderiv p))*
    using *1 pders.simps* **by** *fastforce*
  ultimately show *?case* **by** *auto*
qed

2.3 Sign variations for Fourier sequences

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

definition *changes-gt-der:: real ⇒real poly ⇒ int* **where**
  changes-gt-der a p= changes-poly-at (pders p) a

definition *changes-le-der:: real ⇒real poly ⇒ int* **where**
  changes-le-der b p= (degree p − changes-poly-at (pders p) b)

lemma *changes-poly-pos-inf-pders[simp]:changes-poly-pos-inf (pders p) = 0*
proof *(induct degree p arbitrary:p)*
  case *0*
  then obtain *a* **where** *p=[:a:]* **using** *degree-eq-zeroE* **by** *auto*
  then show *?case*
    apply *(cases a=0)*
    by *(auto simp:changes-poly-pos-inf-def pders.simps)*
next
  case *(Suc x)*
  then have *pderiv p≠0 p≠0* **using** *pderiv-eq-0-iff* **by** *force+*
  define *ps* **where** *ps=pders (pderiv (pderiv p))*

have *ps*:*pders p* = *p*# *pderiv p* #*ps pders* (*pderiv p*) = *pderiv p*#*ps*
unfolding *ps-def* **by** (*simp-all add*: ‹*p* ≠ *0*› ‹*pderiv p* ≠ *0*› *pders.simps*)
have *hyps*:*changes-poly-pos-inf* (*pders* (*pderiv p*)) = *0*
apply (*rule Suc*(*1*))
using ‹*Suc x* = *degree p*› **by** (*metis degree-pderiv diff-Suc-1*)
moreover have *sgn-pos-inf p* ∗ *sgn-pos-inf* (*pderiv p*) >*0*
unfolding *sgn-pos-inf-def lead-coeff-pderiv*
apply (*simp add*:*algebra-simps sgn-mult*)
using *Suc.hyps*(*2*) ‹*p* ≠ *0*› **by** *linarith*
ultimately show *?case* **unfolding** *changes-poly-pos-inf-def ps* **by** *auto*
qed

lemma *changes-poly-neg-inf-pders*[*simp*]: *changes-poly-neg-inf* (*pders p*) = *degree p*
proof (*induct degree p arbitrary*:*p*)
case *0*
then obtain *a* **where** *p*=[:*a*:] **using** *degree-eq-zeroE* **by** *auto*
then show *?case* **unfolding** *changes-poly-neg-inf-def* **by** (*auto simp*: *pders.simps*)
next
case (*Suc x*)
then have *pderiv p*≠*0 p*≠*0* **using** *pderiv-eq-0-iff* **by** *force*+
then have *changes-poly-neg-inf* (*pders p*)
= *changes-poly-neg-inf* (*p* # *pderiv p*#*pders* (*pderiv* (*pderiv p*)))
by (*simp add*:*pders.simps*)
also have ... = *1* + *changes-poly-neg-inf* (*pderiv p*#*pders* (*pderiv* (*pderiv p*)))
proof −
have *sgn-neg-inf p* ∗ *sgn-neg-inf* (*pderiv p*) < *0*
unfolding *sgn-neg-inf-def* **using** ‹*p*≠*0*› ‹*pderiv p*≠*0*›
by (*auto simp add*:*lead-coeff-pderiv degree-pderiv coeff-pderiv sgn-mult pderiv-eq-0-iff*)
then show *?thesis* **unfolding** *changes-poly-neg-inf-def* **by** *auto*
qed
also have ... = *1* + *changes-poly-neg-inf* (*pders* (*pderiv p*))
using ‹*pderiv p*≠*0*› **by** (*simp add*:*pders.simps*)
also have ... = *1* + *degree* (*pderiv p*)
apply (*subst Suc*(*1*))
using *Suc*(*2*) **by** (*auto simp add*: *degree-pderiv*)
also have ... = *degree p*
by (*metis Suc.hyps*(*2*) *degree-pderiv diff-Suc-1 plus-1-eq-Suc*)
finally show *?case* **.**
qed

lemma *pders-coeffs-sgn-eq*:*map* (λ*p*. *sgn*(*poly p 0*)) (*pders p*) = *map sgn* (*coeffs p*)
proof (*induct degree p arbitrary*:*p*)
case *0*
then obtain *a* **where** *p*=[:*a*:] **using** *degree-eq-zeroE* **by** *auto*
then show *?case* **by** (*auto simp*: *pders.simps*)
next
case (*Suc x*)
then have *pderiv p*≠*0 p*≠*0* **using** *pderiv-eq-0-iff* **by** *force*+

```
  have map (λp. sgn (poly p 0)) (pders p)
          = sgn (poly p 0)# map (λp. sgn (poly p 0)) (pders (pderiv p))
    apply (subst pders.simps)
    using ‹p≠0› by simp
  also have ... = sgn (coeff p 0) # map sgn (coeffs (pderiv p))
  proof -
    have sgn (poly p 0) = sgn (coeff p 0) by (simp add: poly-0-coeff-0)
    then show ?thesis
      apply (subst Suc(1))
      subgoal by (metis Suc.hyps(2) degree-pderiv diff-Suc-1)
      subgoal by auto
      done
  qed
  also have ... =  map sgn (coeffs p)
  proof (rule nth-equalityI)
    show p-length:length (sgn (coeff p 0) # map sgn (coeffs (pderiv p)))
                    = length (map sgn (coeffs p))
        by (metis Suc.hyps(2) ‹p ≠ 0› ‹pderiv p ≠ 0› degree-pderiv diff-Suc-1
length-Cons
          length-coeffs-degree length-map)
    show (sgn (coeff p 0) # map sgn (coeffs (pderiv p))) ! i = map sgn (coeffs p)
! i
      if i < length (sgn (coeff p 0) # map sgn (coeffs (pderiv p))) for i
    proof -
      show (sgn (coeff p 0) # map sgn (coeffs (pderiv p))) ! i = map sgn (coeffs p)
! i
      proof (cases i)
        case 0
        then show ?thesis
          by (simp add: ‹p ≠ 0› coeffs-nth)
      next
        case (Suc i')
        then show ?thesis
          using that p-length
          apply simp
          apply (subst (1 2) coeffs-nth)
          by (auto simp add: ‹p ≠ 0› ‹pderiv p ≠ 0› length-coeffs-degree coeff-pderiv
sgn-mult)
      qed
    qed
  qed
  finally show ?case .
qed

lemma changes-poly-at-pders-0:changes-poly-at (pders p) 0 = changes (coeffs p)
  unfolding changes-poly-at-def
  apply (subst (1 2) changes-map-sgn-eq)
  by (auto simp add:pders-coeffs-sgn-eq comp-def)
```

2.4 Budan–Fourier theorem

```
lemma budan-fourier-aux-right:
  assumes c<d2 and p≠0
  assumes ∀ x. c<x∧ x≤d2 ⟶ (∀ q∈set (pders p). poly q x≠0)
  shows changes-itv-der c d2 p=0
  using assms(2−3)
proof (induct degree p arbitrary:p)
  case 0
  then obtain a where p=[:a:] a≠0 by (metis degree-eq-zeroE pCons-0-0)
  then show ?case
    by (auto simp add:changes-itv-der-def pders.simps intro:order-0I)
next
  case (Suc n)
  then have [simp]:pderiv p≠0 by (metis nat.distinct(1) pderiv-eq-0-iff)
  note nzero=‹∀ x. c < x ∧ x ≤ d2 ⟶ (∀ q∈set (pders p). poly q x ≠ 0)›

  have hyps:changes-itv-der c d2 (pderiv p) = 0
    apply (rule Suc(1))
    subgoal by (metis Suc.hyps(2) degree-pderiv diff-Suc-1)
    subgoal by (simp add: Suc.prems(1) Suc.prems(2) pders.simps)
    subgoal by (simp add: Suc.prems(1) nzero pders.simps)
    done
   have pders-changes-c:changes-poly-at (r# pders q) c = (if sign-r-pos q c ⟷
poly r c>0
        then changes-poly-at (pders q) c else 1+changes-poly-at (pders q) c)
    when poly r c≠0 q≠0 for q r
    using ‹q≠0›
  proof (induct q rule:pders.induct)
    case (1 q)
    have ?case when pderiv q=0
    proof −
      have degree q=0 using that pderiv-eq-0-iff by blast
       then obtain a where q=[:a:] a≠0 using ‹q≠0› by (metis degree-eq-zeroE
pCons-0-0)
      then show ?thesis using ‹poly r c≠0›
     by (auto simp add:sign-r-pos-rec changes-poly-at-def mult-less-0-iff pders.simps)
    qed
    moreover have ?case when pderiv q≠0
    proof −
      obtain qs where qs:pders q=q#qs pders (pderiv q) = qs
        using ‹q≠0› by (simp add:pders.simps)
       have changes-poly-at (r # qs) c = (if sign-r-pos (pderiv q) c = (0 < poly r
c)
            then changes-poly-at qs c else 1 + changes-poly-at qs c)
        using 1 ‹pderiv q≠0› unfolding qs by simp
      then show ?thesis unfolding qs
        apply (cases poly q c=0)
         subgoal unfolding changes-poly-at-def by (auto simp:sign-r-pos-rec[OF
‹q≠0›,of c])
```

subgoal unfolding *changes-poly-at-def* **using** ‹*poly r c≠0*›
by (*auto simp*:*sign-r-pos-rec*[*OF* ‹*q≠0*›,*of c*] *mult-less-0-iff*)
done
qed
ultimately show *?case* **by** *blast*
qed
have *pders-changes-d2*:*changes-poly-at* (*r# pders q*) *d2* = (*if sign-r-pos q c* ⟷ *poly r c>0*
then changes-poly-at (*pders q*) *d2 else 1+changes-poly-at* (*pders q*) *d2*)
when *poly r c≠0 q≠0* **and** *qr-nzero*:∀ *x. c < x* ∧ *x* ≤ *d2* ⟶ *poly r x* ≠ *0* ∧ *poly q x≠0*
for *q r*
proof −
have *r≠0* **using** *that*(*1*) **using** *poly-0* **by** *blast*
obtain *qs* **where** *qs*:*pders q=q#qs pders* (*pderiv q*) = *qs*
using ‹*q≠0*› **by** (*simp add*:*pders.simps*)
have *if sign-r-pos r c then 0 < poly r d2 else poly r d2 < 0*
if sign-r-pos q c then 0 < poly q d2 else poly q d2 < 0
subgoal by (*rule sign-r-pos-nzero-right*[*of c d2 r*]) (*use qr-nzero* ‹*c<d2*› **in** *auto*)
subgoal by (*rule sign-r-pos-nzero-right*[*of c d2 q*]) (*use qr-nzero* ‹*c<d2*› **in** *auto*)
done
then show *?thesis* **unfolding** *qs changes-poly-at-def*
using ‹*poly r c≠0*› **by** (*auto split*:*if-splits simp*:*mult-less-0-iff sign-r-pos-rec*[*OF* ‹*r≠0*›])
qed
have *d2c-nzero*:∀ *x. c<x* ∧ *x≤d2* ⟶ *poly p x≠0* ∧ *poly* (*pderiv p*) *x ≠0*
and *p-cons*:*pders p* = *p#pders*(*pderiv p*)
subgoal by (*simp add*: *nzero Suc.prems*(*1*) *pders.simps*)
subgoal by (*simp add*: *Suc.prems*(*1*) *pders.simps*)
done

have *?case* **when** *poly p c=0*
proof −
define *ps* **where** *ps=pders* (*pderiv* (*pderiv p*))
have *ps-cons*:*p#pderiv p#ps* = *pders p pderiv p#ps=pders* (*pderiv p*)
unfolding *ps-def* **using** ‹*p≠0*› **by** (*auto simp*:*pders.simps*)

have *changes-poly-at* (*p # pderiv p # ps*) *c* = *changes-poly-at* (*pderiv p # ps*) *c*
unfolding *changes-poly-at-def* **using** *that* **by** *auto*
moreover have *changes-poly-at* (*p # pderiv p # ps*) *d2* = *changes-poly-at* (*pderiv p # ps*) *d2*
proof −
have *if sign-r-pos p c then 0 < poly p d2 else poly p d2 < 0*
apply (*rule sign-r-pos-nzero-right*[*OF -* ‹*c<d2*›])
using *nzero*[*folded ps-cons*] *assms*(*1*−*2*) **by** *auto*
moreover have *if sign-r-pos* (*pderiv p*) *c then 0 < poly* (*pderiv p*) *d2*

else poly (*pderiv p*) *d2* < *0*
apply (*rule sign-r-pos-nzero-right*[*OF* - ‹*c*<*d2*›])
using *nzero*[*folded ps-cons*] *assms*(*1*−*2*) **by** *auto*
ultimately have *poly p d2* ∗ *poly* (*pderiv p*) *d2* > *0*
unfolding *zero-less-mult-iff sign-r-pos-rec*[*OF* ‹*p*≠*0*›] **using** ‹*poly p c*=*0*›
by (*auto split*:*if-splits*)
then show *?thesis* **unfolding** *changes-poly-at-def* **by** *auto*
qed
ultimately show *?thesis* **using** *hyps* **unfolding** *changes-itv-der-def*
apply (*fold ps-cons*)
by (*auto simp*:*Let-def*)
qed
moreover have *?case* **when** *poly p c*≠*0 sign-r-pos* (*pderiv p*) *c* ⟷ *poly p c*>*0*
proof −
have *changes-poly-at* (*pders p*) *c* = *changes-poly-at* (*pders* (*pderiv p*)) *c*
unfolding *p-cons*
apply (*subst pders-changes-c*[*OF* ‹*poly p c*≠*0*›])
using *that* **by** *auto*
moreover have *changes-poly-at* (*pders p*) *d2* = *changes-poly-at* (*pders* (*pderiv p*)) *d2*
unfolding *p-cons*
apply (*subst pders-changes-d2*[*OF* ‹*poly p c*≠*0*› - *d2c-nzero*])
using *that* **by** *auto*
ultimately show *?thesis* **using** *hyps* **unfolding** *changes-itv-der-def Let-def*
by *auto*
qed
moreover have *?case* **when** *poly p c*≠*0* ¬ *sign-r-pos* (*pderiv p*) *c* ⟷ *poly p c*>*0*
proof −
have *changes-poly-at* (*pders p*) *c* = *changes-poly-at* (*pders* (*pderiv p*)) *c* +*1*
unfolding *p-cons*
apply (*subst pders-changes-c*[*OF* ‹*poly p c*≠*0*›])
using *that* **by** *auto*
moreover have *changes-poly-at* (*pders p*) *d2* = *changes-poly-at* (*pders* (*pderiv p*)) *d2* + *1*
unfolding *p-cons*
apply (*subst pders-changes-d2*[*OF* ‹*poly p c*≠*0*› - *d2c-nzero*])
using *that* **by** *auto*
ultimately show *?thesis* **using** *hyps* **unfolding** *changes-itv-der-def Let-def*
by *auto*
qed
ultimately show *?case* **by** *blast*
qed

lemma *budan-fourier-aux-left′*:
assumes *d1*<*c* **and** *p*≠*0*
assumes ∀ *x*. *d1*≤*x*∧ *x*<*c* ⟶ (∀ *q*∈*set* (*pders p*). *poly q x*≠*0*)
shows *changes-itv-der d1 c p* ≥ *order c p* ∧ *even* (*changes-itv-der d1 c p* − *order c p*)

```
  using assms(2−3)
proof (induct degree p arbitrary:p)
  case 0
  then obtain a where p=[:a:] a≠0 by (metis degree-eq-zeroE pCons-0-0)
  then show ?case
    apply (auto simp add:changes-itv-der-def pders.simps intro:order-0I)
    by (metis add.right-neutral dvd-0-right mult-zero-right order-root poly-pCons)
next
  case (Suc n)
  then have [simp]:pderiv p≠0 by (metis nat.distinct(1) pderiv-eq-0-iff)
  note nzero=‹∀x. d1 ≤ x ∧ x < c ⟶ (∀q∈set (pders p). poly q x ≠ 0)›
  define v where v=order c (pderiv p)

  have hyps:v ≤ changes-itv-der d1 c (pderiv p) ∧ even (changes-itv-der d1 c (pderiv
p) − v)
    unfolding v-def
    apply (rule Suc(1))
    subgoal by (metis Suc.hyps(2) degree-pderiv diff-Suc-1)
    subgoal by (simp add: Suc.prems(1) Suc.prems(2) pders.simps)
    subgoal by (simp add: Suc.prems(1) nzero pders.simps)
    done
  have pders-changes-c:changes-poly-at (r# pders q) c = (if sign-r-pos q c ⟷
poly r c>0
          then changes-poly-at (pders q) c else 1+changes-poly-at (pders q) c)
    when poly r c≠0 q≠0 for q r
    using ‹q≠0›
  proof (induct q rule:pders.induct)
    case (1 q)
    have ?case when pderiv q=0
    proof −
      have degree q=0 using that pderiv-eq-0-iff by blast
      then obtain a where q=[:a:] a≠0 using ‹q≠0› by (metis degree-eq-zeroE
pCons-0-0)
      then show ?thesis using ‹poly r c≠0›
      by (auto simp add:sign-r-pos-rec changes-poly-at-def mult-less-0-iff pders.simps)
    qed
    moreover have ?case when pderiv q≠0
    proof −
      obtain qs where qs:pders q=q#qs pders (pderiv q) = qs
        using ‹q≠0› by (simp add:pders.simps)
      have changes-poly-at (r # qs) c = (if sign-r-pos (pderiv q) c = (0 < poly r
c)
            then changes-poly-at qs c else 1 + changes-poly-at qs c)
        using 1 ‹pderiv q≠0› unfolding qs by simp
      then show ?thesis unfolding qs
        apply (cases poly q c=0)
          subgoal unfolding changes-poly-at-def by (auto simp:sign-r-pos-rec[OF
‹q≠0›,of c])
        subgoal unfolding changes-poly-at-def using ‹poly r c≠0›
```

```
        by (auto simp:sign-r-pos-rec[OF ‹q≠0›,of c] mult-less-0-iff)
      done
    qed
    ultimately show ?case by blast
  qed
  have pders-changes-d1:changes-poly-at (r# pders q) d1 = (if even (order c q) ⟷ sign-r-pos q c ⟷ poly r c>0
        then changes-poly-at (pders q) d1 else 1+changes-poly-at (pders q) d1)
    when poly r c≠0 q≠0 and qr-nzero:∀x. d1 ≤ x ∧ x < c ⟶ poly r x ≠ 0 ∧ poly q x≠0
    for q r
  proof −
    have r≠0 using that(1) using poly-0 by blast
    obtain qs where qs:pders q=q#qs pders (pderiv q) = qs
      using ‹q≠0› by (simp add:pders.simps)
    have if even (order c r) = sign-r-pos r c then 0 < poly r d1 else poly r d1 < 0
      if even (order c q) = sign-r-pos q c then 0 < poly q d1 else poly q d1 < 0
       subgoal by (rule sign-r-pos-nzero-left[of d1 c r]) (use qr-nzero ‹d1<c› in auto)
       subgoal by (rule sign-r-pos-nzero-left[of d1 c q]) (use qr-nzero ‹d1<c› in auto)
      done
    moreover have order c r=0 by (simp add: order-0I that(1))
    ultimately show ?thesis unfolding qs changes-poly-at-def
     using ‹poly r c≠0› by (auto split:if-splits simp:mult-less-0-iff sign-r-pos-rec[OF ‹r≠0›])
  qed
  have d1c-nzero:∀x. d1 ≤ x ∧ x < c ⟶ poly p x ≠ 0 ∧ poly (pderiv p) x ≠ 0
    and p-cons:pders p = p#pders(pderiv p)
    by (simp-all add: nzero Suc.prems(1) pders.simps)

  have ?case when poly p c=0
  proof −
    define ps where ps=pders (pderiv (pderiv p))
    have ps-cons:p#pderiv p#ps = pders p pderiv p#ps=pders (pderiv p)
      unfolding ps-def using ‹p≠0› by (auto simp:pders.simps)

    have p-order:order c p = Suc v
      apply (subst order-pderiv)
      using Suc.prems(1) order-root that unfolding v-def by auto
   moreover have changes-poly-at (p#pderiv p # ps) d1 = changes-poly-at (pderiv p#ps) d1 +1
    proof −
     have if even (order c p) = sign-r-pos p c then 0 < poly p d1 else poly p d1 < 0
        apply (rule sign-r-pos-nzero-left[OF - ‹d1<c›])
        using nzero[folded ps-cons] assms(1−2) by auto
      moreover have if even v = sign-r-pos (pderiv p) c
                  then 0 < poly (pderiv p) d1 else poly (pderiv p) d1 < 0
```

unfolding *v-def*
apply (*rule sign-r-pos-nzero-left*[*OF* - ‹*d1*<*c*›])
using *nzero*[*folded ps-cons*] *assms*(*1*−*2*) **by** *auto*
ultimately have *poly p d1* ∗ *poly* (*pderiv p*) *d1* < *0*
unfolding *mult-less-0-iff sign-r-pos-rec*[*OF* ‹*p*≠*0*›] **using** ‹*poly p c*=*0*› *p-order*
by (*auto split*:*if-splits*)
then show *?thesis*
unfolding *changes-poly-at-def* **by** *auto*
qed
moreover have *changes-poly-at* (*p* # *pderiv p* # *ps*) *c* = *changes-poly-at* (*pderiv p* # *ps*) *c*
unfolding *changes-poly-at-def* **using** *that* **by** *auto*
ultimately show *?thesis* **using** *hyps* **unfolding** *changes-itv-der-def*
apply (*fold ps-cons*)
by (*auto simp*:*Let-def*)
qed
moreover have *?case* **when** *poly p c*≠*0 odd v sign-r-pos* (*pderiv p*) *c* ⟷ *poly p c*>*0*
proof −
have *order c p*=*0* **by** (*simp add*: *order-0I that*(*1*))
moreover have *changes-poly-at* (*pders p*) *d1* = *changes-poly-at* (*pders* (*pderiv p*)) *d1* +*1*
unfolding *p-cons*
apply (*subst pders-changes-d1*[*OF* ‹*poly p c*≠*0*› - *d1c-nzero*])
using *that* **unfolding** *v-def* **by** *auto*
moreover have *changes-poly-at* (*pders p*) *c* = *changes-poly-at* (*pders* (*pderiv p*)) *c*
unfolding *p-cons*
apply (*subst pders-changes-c*[*OF* ‹*poly p c*≠*0*›])
using *that* **unfolding** *v-def* **by** *auto*
ultimately show *?thesis* **using** *hyps* ‹*odd v*› **unfolding** *changes-itv-der-def Let-def*
by *auto*
qed
moreover have *?case* **when** *poly p c*≠*0 odd v* ¬ *sign-r-pos* (*pderiv p*) *c* ⟷ *poly p c*>*0*
proof −
have *v*≥*1* **using** ‹*odd v*› **using** *not-less-eq-eq* **by** *auto*
moreover have *order c p*=*0* **by** (*simp add*: *order-0I that*(*1*))
moreover have *changes-poly-at* (*pders p*) *d1* = *changes-poly-at* (*pders* (*pderiv p*)) *d1*
unfolding *p-cons*
apply (*subst pders-changes-d1*[*OF* ‹*poly p c*≠*0*› - *d1c-nzero*])
using *that* **unfolding** *v-def* **by** *auto*
moreover have *changes-poly-at* (*pders p*) *c* = *changes-poly-at* (*pders* (*pderiv p*)) *c* + *1*
unfolding *p-cons*
apply (*subst pders-changes-c*[*OF* ‹*poly p c*≠*0*›])

using *that* **unfolding** *v-def* **by** *auto*
ultimately show *?thesis* **using** *hyps* ‹*odd v*› **unfolding** *changes-itv-der-def Let-def*
by *auto*
qed
moreover have *?case* **when** *poly p c≠0 even v sign-r-pos* (*pderiv p*) *c* ⟷ *poly p c>0*
proof −
have *order c p=0* **by** (*simp add*: *order-0I that*(*1*))
moreover have *changes-poly-at* (*pders p*) *d1* = *changes-poly-at* (*pders* (*pderiv p*)) *d1*
unfolding *p-cons*
apply (*subst pders-changes-d1*[*OF* ‹*poly p c≠0*› - *d1c-nzero*])
using *that* **unfolding** *v-def* **by** *auto*
moreover have *changes-poly-at* (*pders p*) *c* = *changes-poly-at* (*pders* (*pderiv p*)) *c*
unfolding *p-cons*
apply (*subst pders-changes-c*[*OF* ‹*poly p c≠0*›])
using *that* **unfolding** *v-def* **by** *auto*
ultimately show *?thesis* **using** *hyps* ‹*even v*› **unfolding** *changes-itv-der-def Let-def*
by *auto*
qed
moreover have *?case* **when** *poly p c≠0 even v* ¬ *sign-r-pos* (*pderiv p*) *c* ⟷ *poly p c>0*
proof −
have *order c p=0* **by** (*simp add*: *order-0I that*(*1*))
moreover have *changes-poly-at* (*pders p*) *d1* = *changes-poly-at* (*pders* (*pderiv p*)) *d1* + *1*
unfolding *p-cons*
apply (*subst pders-changes-d1*[*OF* ‹*poly p c≠0*› - *d1c-nzero*])
using *that* **unfolding** *v-def* **by** *auto*
moreover have *changes-poly-at* (*pders p*) *c* = *changes-poly-at* (*pders* (*pderiv p*)) *c* +*1*
unfolding *p-cons*
apply (*subst pders-changes-c*[*OF* ‹*poly p c≠0*›])
using *that* **unfolding** *v-def* **by** *auto*
ultimately show *?thesis* **using** *hyps* ‹*even v*› **unfolding** *changes-itv-der-def Let-def*
by *auto*
qed
ultimately show *?case* **by** *blast*
qed

lemma *budan-fourier-aux-left*:
assumes *d1<c* **and** *p≠0*
assumes *nzero*:∀ *x*. *d1<x*∧ *x<c* ⟶ (∀ *q*∈*set* (*pders p*). *poly q x≠0*)
shows *changes-itv-der d1 c p* ≥ *order c p even* (*changes-itv-der d1 c p* − *order c p*)

```
proof −
  define d where d=(d1+c)/2
  have d1<d d<c unfolding d-def using ‹d1<c› by auto

  have changes-itv-der d1 d p = 0
    apply (rule budan-fourier-aux-right[OF ‹d1<d› ‹p≠0›])
    using nzero ‹d1<d› ‹d<c› by auto
  moreover have order c p ≤ changes-itv-der d c p ∧ even (changes-itv-der d c p
− order c p)
    apply (rule budan-fourier-aux-left'[OF ‹d<c› ‹p≠0›])
    using nzero ‹d1<d› ‹d<c› by auto
  ultimately show changes-itv-der d1 c p ≥ order c p even (changes-itv-der d1 c
p − order c p)
    unfolding changes-itv-der-def Let-def by auto
qed

theorem budan-fourier-interval:
  assumes a<b p≠0
  shows changes-itv-der a b p ≥ proots-count p {x. a< x ∧ x≤ b} ∧
          even (changes-itv-der a b p − proots-count p {x. a< x ∧ x≤ b})
  using ‹a<b›
proof (induct card {x. ∃ p∈set (pders p). poly p x=0 ∧ a<x ∧ x<b} arbitrary:b)
  case 0
  have nzero:∀ x. a<x ∧ x<b ⟶ (∀ q∈set (pders p). poly q x≠0)
  proof −
    define S where S={x. ∃ p∈set (pders p). poly p x = 0 ∧ a < x ∧ x < b}
    have finite S
    proof −
      have S ⊆ (⋃ p∈set (pders p). proots p)
        unfolding S-def by auto
      moreover have finite (⋃ p∈set (pders p). proots p)
        apply (subst finite-UN)
        using set-pders-nzero[OF ‹p≠0›] by auto
      ultimately show ?thesis by (simp add: finite-subset)
    qed
    moreover have card S = 0 unfolding S-def using 0 by auto
    ultimately have S={} by auto
     then show ?thesis unfolding S-def using ‹a<b› assms(2) pders.simps by
fastforce
  qed
  from budan-fourier-aux-left[OF ‹a<b› ‹p≠0› this]
  have order b p ≤ changes-itv-der a b p even (changes-itv-der a b p − order b p)
by simp-all
  moreover have proots-count p {x. a< x ∧ x≤ b} = order b p
  proof −
    have p-cons:pders p=p#pders (pderiv p) by (simp add: assms(2) pders.simps)
    have proots-within p {x. a < x ∧ x ≤ b} = (if poly p b=0 then {b} else {})
      using nzero ‹a< b› unfolding p-cons
      apply auto
```

using *not-le* **by** *fastforce*
then show *?thesis* **unfolding** *proots-count-def* **using** *order-root* **by** *auto*
qed
ultimately show *?case* **by** *auto*
next
case $(Suc\ n)$
define P **where** $P=(\lambda x.\ \exists p\in set\ (pders\ p).\ poly\ p\ x = 0)$
define S **where** $S=(\lambda b.\ \{x.\ P\ x \wedge a < x \wedge x < b\})$
define b' **where** $b'=Max\ (S\ b)$
have *f-S*:$finite\ (S\ x)$ **for** x
proof –
have $S\ x \subseteq (\bigcup p\in set\ (pders\ p).\ proots\ p)$
unfolding *S-def P-def* **by** *auto*
moreover have $finite\ (\bigcup p\in set\ (pders\ p).\ proots\ p)$
apply (*subst finite-UN*)
using *set-pders-nzero*[*OF* ‹$p\neq 0$›] **by** *auto*
ultimately show *?thesis* **by** (*simp add*: *finite-subset*)
qed
have $b'\in S\ b$
unfolding *b′-def*
apply (*rule Max-in*[*OF f-S*])
using *Suc*(*2*) **unfolding** *S-def P-def* **by** *force*
then have $a<b'$ $b'<b$ **unfolding** *S-def* **by** *auto*
have *b′-nzero*:$\forall x.\ b'<x \wedge x<b \longrightarrow (\forall q\in set\ (pders\ p).\ poly\ q\ x\neq 0)$
proof (*rule ccontr*)
assume $\neg\ (\forall x.\ b' < x \wedge x < b \longrightarrow (\forall q\in set\ (pders\ p).\ poly\ q\ x \neq 0))$
then obtain bb **where** $P\ bb$ $b'<bb$ $bb<b$ **unfolding** *P-def* **by** *auto*
then have $bb\in S\ b$ **unfolding** *S-def* **using** ‹$a<b'$› ‹$b'<b$› **by** *auto*
from *Max-ge*[*OF f-S this*, *folded b′-def*] **have** $bb \leq b'$ **.**
then show *False* **using** ‹$b'<bb$› **by** *auto*
qed

have *hyps*:$proots\text{-}count\ p\ \{x.\ a < x \wedge x \leq b'\} \leq changes\text{-}itv\text{-}der\ a\ b'\ p \wedge$
$even\ (changes\text{-}itv\text{-}der\ a\ b'\ p - proots\text{-}count\ p\ \{x.\ a < x \wedge x \leq b'\})$
proof (*rule Suc*(*1*)[*OF* - ‹$a<b'$›])
have $S\ b= \{b'\} \cup S\ b'$
proof –
have $\{x.\ P\ x \wedge b' < x \wedge x < b\} = \{\}$
using *b′-nzero* **unfolding** *P-def* **by** *auto*
then have $\{x.\ P\ x\wedge\ b' \leq x \wedge x < b\} = \{b'\}$
using ‹$b'\in S\ b$› **unfolding** *S-def* **by** *force*
moreover have $S\ b= S\ b' \cup\ \{x.\ P\ x \wedge b' \leq x \wedge x < b\}$
unfolding *S-def* **using** ‹$a<b'$› ‹$b'<b$› **by** *auto*
ultimately show *?thesis* **by** *auto*
qed
moreover have $Suc\ n = card\ (S\ b)$ **using** *Suc*(*2*) **unfolding** *S-def P-def* **by**
simp
moreover have $b'\notin S\ b'$ **unfolding** *S-def* **by** *auto*
ultimately have $n=card\ (S\ b')$ **using** *f-S* **by** *auto*

then show $n = card\ \{x.\ \exists\, p{\in}set\ (pders\ p).\ poly\ p\ x = 0 \wedge a < x \wedge x < b'\}$
unfolding *S-def P-def* **by** *simp*
qed
moreover have $proots\text{-}count\ p\ \{x.\ a < x \wedge x \le b\}$
$= proots\text{-}count\ p\ \{x.\ a < x \wedge x \le b'\} + order\ b\ p$
proof −
have *p-cons*:*pders p*=*p*#*pders* (*pderiv p*) **by** (*simp add*: *assms*(*2*) *pders.simps*)
have $proots\text{-}within\ p\ \{x.\ b' < x \wedge x \le b\} = (\mathit{if}\ poly\ p\ b{=}0\ \mathit{then}\ \{b\}\ \mathit{else}\ \{\})$
using b'-*nzero* ‹$b' < b$› **unfolding** *p-cons*
apply *auto*
using *not-le* **by** *fastforce*
then have $proots\text{-}count\ p\ \{x.\ b' < x \wedge x \le b\} = order\ b\ p$
unfolding *proots-count-def* **using** *order-root* **by** *auto*
moreover have $proots\text{-}count\ p\ \{x.\ a < x \wedge x \le b\} = proots\text{-}count\ p\ \{x.\ a < x \wedge x \le b'\} +$
$proots\text{-}count\ p\ \{x.\ b' < x \wedge x \le b\}$
apply (*subst proots-count-union-disjoint*[*symmetric*])
using ‹$a{<}b'$› ‹$b'{<}b$› ‹$p{\neq}0$› **by** (*auto intro*:*arg-cong2*[**where** *f*=*proots-count*])
ultimately show *?thesis* **by** *auto*
qed
moreover note *budan-fourier-aux-left*[*OF* ‹$b'{<}b$› ‹$p{\neq}0$› b'-*nzero*]
ultimately show *?case* **unfolding** *changes-itv-der-def Let-def* **by** *auto*
qed

theorem *budan-fourier-gt*:
assumes $p{\neq}0$
shows $changes\text{-}gt\text{-}der\ a\ p \ge proots\text{-}count\ p\ \{x.\ a{<}\ x\} \wedge$
$even\ (changes\text{-}gt\text{-}der\ a\ p - proots\text{-}count\ p\ \{x.\ a{<}\ x\})$
proof −
define *ps* **where** *ps*=*pders p*
obtain *ub* **where** *ub-root*:$\forall\, p{\in}set\ ps.\ \forall\, x.\ poly\ p\ x = 0 \longrightarrow x < ub$
and *ub-sgn*:$\forall\, x{\ge}ub.\ \forall\, p{\in}set\ ps.\ sgn\ (poly\ p\ x) = sgn\text{-}pos\text{-}inf\ p$
and $a < ub$
using *root-list-ub*[*of ps a*] *set-pders-nzero*[*OF* ‹$p{\neq}0$›,*folded ps-def*] **by** *blast*
have $proots\text{-}count\ p\ \{x.\ a{<}\ x\} = proots\text{-}count\ p\ \{x.\ a{<}\ x \wedge x \le ub\}$
proof −
have $p{\in}set\ ps$ **unfolding** *ps-def* **by** (*simp add*: *assms pders.simps*)
then have $proots\text{-}within\ p\ \{x.\ a{<}\ x\} = proots\text{-}within\ p\ \{x.\ a{<}\ x \wedge x{\le}ub\}$
using *ub-root* **by** *fastforce*
then show *?thesis* **unfolding** *proots-count-def* **by** *auto*
qed
moreover have $changes\text{-}gt\text{-}der\ a\ p = changes\text{-}itv\text{-}der\ a\ ub\ p$
proof −
have $map\ (sgn \circ (\lambda p.\ poly\ p\ ub))\ ps = map\ sgn\text{-}pos\text{-}inf\ ps$
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*)

```
    then have changes-poly-at ps ub=0 unfolding ps-def by simp
    thus ?thesis unfolding changes-gt-der-def changes-itv-der-def ps-def
      by (simp add:Let-def)
  qed
  moreover have proots-count p {x. a < x ∧ x ≤ ub} ≤ changes-itv-der a ub p ∧
      even (changes-itv-der a ub p − proots-count p {x. a < x ∧ x ≤ ub})
    using budan-fourier-interval[OF ‹a<ub› ‹p≠0›] .
  ultimately show ?thesis by auto
qed
```

Descartes' rule of signs is a direct consequence of the Budan–Fourier theorem

```
theorem descartes-sign:
  fixes p::real poly
  assumes p≠0
  shows  changes (coeffs p) ≥ proots-count p {x. 0 < x} ∧
        even (changes (coeffs p) − proots-count p {x. 0< x})
  using budan-fourier-gt[OF ‹p≠0›,of 0] unfolding changes-gt-der-def
  by (simp add:changes-poly-at-pders-0)

theorem budan-fourier-le:
  assumes p≠0
  shows changes-le-der b p ≥ proots-count p {x. x ≤b} ∧
        even (changes-le-der b p − proots-count p {x. x ≤b})
proof −
  define ps where ps=pders p
  obtain lb where lb-root:∀ 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 ps b] set-pders-nzero[OF ‹p≠0›,folded ps-def] by blast
  have proots-count p {x. x ≤b} = proots-count p {x. lb< x ∧ x ≤ b}
  proof −
    have p∈set ps unfolding ps-def by (simp add: assms pders.simps)
    then have proots-within p {x. x ≤b} = proots-within p {x. lb< x ∧ x≤b}
      using lb-root by fastforce
    then show ?thesis unfolding proots-count-def by auto
  qed
  moreover have changes-le-der b p = changes-itv-der lb b p
  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)
    then have changes-poly-at ps lb=degree p unfolding ps-def by simp
    thus ?thesis unfolding changes-le-der-def changes-itv-der-def ps-def
      by (simp add:Let-def)
  qed
```

moreover have *proots-count p* $\{x.\ lb < x \wedge x \leq b\} \leq$ *changes-itv-der lb b p* $\wedge$
even (*changes-itv-der lb b p* $-$ *proots-count p* $\{x.\ lb < x \wedge x \leq b\}$)
using *budan-fourier-interval*[*OF* ‹*lb*<*b*› ‹*p*≠*0*›] **.**
ultimately show *?thesis* **by** *auto*
qed

2.5 Count exactly when all roots are real

definition *all-roots-real*:: *real poly* ⇒ *bool* **where**
all-roots-real p = (∀ *r*∈*proots* (*map-poly of-real p*). *Im r*=*0*)

lemma *all-roots-real-mult*[*simp*]:
all-roots-real (*p*∗*q*) ⟷ *all-roots-real p* ∧ *all-roots-real q*
unfolding *all-roots-real-def* **by** *auto*

lemma *all-roots-real-const-iff*:
assumes *all-real*:*all-roots-real p*
shows *degree p*≠*0* ⟷ (∃ *x*. *poly p x*=*0*)
proof
assume *degree p* ≠ *0*
moreover have *degree p*=*0* **when** ∀ *x*. *poly p x*≠*0*
proof −
define *pp* **where** *pp*=*map-poly complex-of-real p*
have ∀ *x*. *poly pp x*≠*0*
proof (*rule ccontr*)
assume ¬ (∀ *x*. *poly pp x* ≠ *0*)
then obtain *x* **where** *poly pp x*=*0* **by** *auto*
moreover have *Im x*=*0*
using *all-real*[*unfolded all-roots-real-def*, *rule-format*,*of x*,*folded pp-def*] ‹*poly pp x*=*0*›
by *auto*
ultimately have *poly pp* (*of-real* (*Re x*)) = *0*
by (*simp add*: *complex-is-Real-iff*)
then have *poly p* (*Re x*) = *0*
unfolding *pp-def*
by (*metis Re-complex-of-real of-real-poly-map-poly zero-complex.simps*(*1*))
then show *False* **using** *that* **by** *simp*
qed
then obtain *a* **where** *pp*=[:*of-real a*:] *a*≠*0*
by (*metis* ‹*degree p* ≠ *0*› *constant-degree degree-map-poly fundamental-theorem-of-algebra of-real-eq-0-iff pp-def*)
then have *p*=[:*a*:] **unfolding** *pp-def*
by (*metis map-poly-0 map-poly-pCons of-real-0 of-real-poly-eq-iff*)
then show *?thesis* **by** *auto*
qed
ultimately show ∃ *x*. *poly p x* = *0* **by** *auto*
next
assume ∃ *x*. *poly p x* = *0*
then show *degree p* ≠ *0*

```
    by (metis UNIV-I all-roots-real-def assms degree-pCons-eq-if
        imaginary-unit.sel(2) map-poly-0 nat.simps(3) order-root pCons-eq-0-iff
        proots-within-iff synthetic-div-eq-0-iff synthetic-div-pCons zero-neq-one)
qed

lemma all-roots-real-degree:
  assumes all-roots-real p
  shows proots-count p UNIV =degree p using assms
proof (induct p rule:poly-root-induct-alt)
  case 0
   then have False using imaginary-unit.sel(2) unfolding all-roots-real-def by
auto
  then show ?case by simp
next
  case (no-proots p)
  from all-roots-real-const-iff[OF this(2)] this(1)
  have degree p=0 by auto
  then obtain a where p=[:a:] a≠0
    by (metis degree-eq-zeroE no-proots.hyps poly-const-conv)
  then have proots p={} by auto
  then show ?case using ‹p=[:a:]› by (simp add:proots-count-def)
next
  case (root a p)
  define a1 where a1=[:− a, 1:]
  have p≠0 using root.prems
    apply auto
    using imaginary-unit.sel(2) unfolding all-roots-real-def by auto
  have a1≠0 unfolding a1-def by auto

   have proots-count (a1 ∗ p) UNIV = proots-count a1 UNIV + proots-count p
UNIV
    using ‹p≠0› ‹a1≠0› by (subst proots-count-times,auto)
  also have ... = 1 + degree p
  proof −
   have proots-count a1 UNIV = 1 unfolding a1-def by (simp add: proots-count-pCons-1-iff)
    moreover have hyps:proots-count p UNIV = degree p
      apply (rule root.hyps)
      using root.prems[folded a1-def] unfolding all-roots-real-def by auto
    ultimately show ?thesis by auto
  qed
  also have ... = degree (a1∗p)
    apply (subst degree-mult-eq)
    using ‹a1≠0› ‹p≠0› unfolding a1-def by auto
  finally show ?case unfolding a1-def .
qed

lemma all-real-roots-mobius:
  fixes a b::real
  assumes all-roots-real p and a<b
```

```
  shows all-roots-real (fcompose p [:a,b:] [:1,1:]) using assms(1)
proof (induct p rule:poly-root-induct-alt)
  case 0
  then show ?case by simp
next
  case (no-proots p)
  from all-roots-real-const-iff[OF this(2)] this(1)
  have degree p=0 by auto
  then obtain a where p=[:a:] a≠0
    by (metis degree-eq-zeroE no-proots.hyps poly-const-conv)
  then show ?case by (auto simp add:all-roots-real-def)
next
  case (root x p)
  define x1 where x1=[:- x, 1:]
  define fx where fx=fcompose x1 [:a, b:] [:1, 1:]

  have all-roots-real fx
  proof (cases x=b)
    case True
    then have fx = [:a-x:] a≠x
     subgoal unfolding fx-def by (simp add:fcompose-def smult-add-right x1-def)
      subgoal using ‹a<b› True by auto
      done
    then have proots (map-poly complex-of-real fx) = {}
      by auto
    then show ?thesis unfolding all-roots-real-def by auto
  next
    case False
    then have fx = [:a-x,b-x:]
      unfolding fx-def by (simp add:fcompose-def smult-add-right x1-def)
    then have proots (map-poly complex-of-real fx) = {of-real ((x-a)/(b-x))}
      using False by (auto simp add:field-simps)
    then show ?thesis unfolding all-roots-real-def by auto
  qed
  moreover have all-roots-real (fcompose p [:a, b:] [:1, 1:])
    using root[folded x1-def] all-roots-real-mult by auto
  ultimately show ?case
    apply (fold x1-def)
    by (auto simp add:fcompose-mult fx-def)
qed
```

If all roots are real, we can use the Budan–Fourier theorem to EXACTLY count the number of real roots.

```
corollary budan-fourier-real:
  assumes p≠0
  assumes all-roots-real p
  shows proots-count p {x. x ≤a} = changes-le-der a p
      a<b ⟹ proots-count p {x. a <x ∧ x ≤b} = changes-itv-der a b p
      proots-count p {x. b <x} = changes-gt-der b p
```

proof −
  have ∗:*proots-count p* {*x. x* ≤*a*} = *changes-le-der a p*
      ∧ *proots-count p* {*x. a* <*x* ∧ *x* ≤*b*} = *changes-itv-der a b p*
      ∧ *proots-count p* {*x. b* <*x*} = *changes-gt-der b p*
    when *a*<*b* **for** *a b*
  proof −
    define *c1 c2 c3* **where**
      c1=*changes-le-der a p* − *proots-count p* {*x. x* ≤*a*} **and**
      c2=*changes-itv-der a b p* − *proots-count p* {*x. a* <*x* ∧ *x* ≤*b*} **and**
      c3=*changes-gt-der b p* − *proots-count p* {*x. b* <*x*}

    have *c1*≥*0 c2*≥*0 c3*≥*0*
     using *budan-fourier-interval*[*OF* ‹*a*<*b*› ‹*p*≠*0*›] *budan-fourier-gt*[*OF* ‹*p*≠*0*›,*of*
b]
        budan-fourier-le[*OF* ‹*p*≠*0*›,*of a*]
      unfolding *c1-def c2-def c3-def* **by** *auto*
    moreover have *c1*+*c2*+*c3*=*0*
    proof −
      have *proots-deg*:*proots-count p UNIV* =*degree p*
        using *all-roots-real-degree*[*OF* ‹*all-roots-real p*›] **.**
      have *changes-le-der a p* + *changes-itv-der a b p* + *changes-gt-der b p* = *degree*
p
        unfolding *changes-le-der-def changes-itv-der-def changes-gt-der-def*
        by (*auto simp add*:*Let-def*)
      moreover have *proots-count p* {*x. x* ≤*a*} + *proots-count p* {*x. a* <*x* ∧ *x* ≤*b*}

          + *proots-count p* {*x. b* <*x*} = *degree p*
        using ‹*p*≠*0*› ‹*a*<*b*›
        apply (*subst proots-count-union-disjoint*[*symmetric*],*auto*)+
        apply (*subst proots-deg*[*symmetric*])
        by (*auto intro*!:*arg-cong2*[**where** *f*=*proots-count*])
      ultimately show *?thesis* **unfolding** *c1-def c2-def c3-def*
        by (*auto simp add*:*algebra-simps*)
    qed
    ultimately have *c1* =*0* ∧ *c2*=*0* ∧ *c3*=*0* **by** *auto*
    then show *?thesis* **unfolding** *c1-def c2-def c3-def* **by** *auto*
  qed
  show *proots-count p* {*x. x* ≤*a*} = *changes-le-der a p* **using** ∗[*of a a*+*1*] **by** *auto*
  show *proots-count p* {*x. a* <*x* ∧ *x* ≤*b*} = *changes-itv-der a b p* **when** *a*<*b*
    using ∗[*OF that*] **by** *auto*
  show *proots-count p* {*x. b* <*x*} = *changes-gt-der b p*
    using ∗[*of b*−*1 b*] **by** *auto*
qed

Similarly, Descartes' rule of sign counts exactly when all roots are real.

corollary *descartes-sign-real*:
  fixes *p*::*real poly* **and** *a b*::*real*
  assumes *p*≠*0*
  assumes *all-roots-real p*

```
  shows proots-count p {x. 0 < x} = changes (coeffs p)
  using budan-fourier-real(3)[OF ‹p≠0› ‹all-roots-real p›]
  unfolding changes-gt-der-def by (simp add:changes-poly-at-pders-0)

end
```

3 Extension of Sturm's theorem for multiple roots

```
theory Sturm-Multiple-Roots
  imports
    BF-Misc
begin
```

The classic Sturm's theorem is used to count real roots WITHOUT multiplicity of a polynomial within an interval. Surprisingly, we can also extend Sturm's theorem to count real roots WITH multiplicity by modifying the signed remainder sequence, which seems to be overlooked by many textbooks.

Our formal proof is inspired by Theorem 10.5.6 in Rahman, Q.I., Schmeisser, G.: Analytic Theory of Polynomials. Oxford University Press (2002).

3.1 More results for *smods*

```
lemma last-smods-gcd:
  fixes p q ::real poly
  defines pp ≡ last (smods p q)
  assumes p≠0
  shows pp = smult (lead-coeff pp) (gcd p q)
  using ‹p≠0› unfolding pp-def
proof (induct smods p q arbitrary:p q rule:length-induct)
  case 1
  have ?case when q=0
    using that smult-normalize-field-eq ‹p≠0› by auto
  moreover have ?case when q≠0
  proof −
    define r where r= − (p mod q)
    have smods-cons:smods p q = p # smods q r
      unfolding r-def using ‹p≠0› by simp
    have last (smods q r) = smult (lead-coeff (last (smods q r))) (gcd q r)
      apply (rule 1(1)[rule-format,of smods q r q r])
      using smods-cons ‹q≠0› by auto
    moreover have gcd p q = gcd q r
      unfolding r-def by (simp add: gcd.commute that)
    ultimately show ?thesis unfolding smods-cons using ‹q≠0›
      by simp
  qed
  ultimately show ?case by argo
qed
```

lemma *last-smods-nzero*:
  assumes *p≠0*
  shows *last* (*smods p q*) *≠0*
  by (*metis assms last-in-set no-0-in-smods smods-nil-eq*)

3.2 Alternative signed remainder sequences

function *smods-ext*::*real poly* ⇒ *real poly* ⇒ *real poly list* **where**
  smods-ext p q = (*if p=0 then* [] *else*
            (*if p mod q* ≠ *0*
              then Cons p (*smods-ext q* (−(*p mod q*)))
              else Cons p (*smods-ext q* (*pderiv 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*))
  using *degree-mod-less* **by** (*auto simp add*:*degree-pderiv pderiv-eq-0-iff*)

lemma *smods-ext-prefix*:
  fixes *p q*::*real poly*
  defines *pp* ≡ *last* (*smods p q*)
  assumes *p≠0 q≠0*
  shows *smods-ext p q* = *smods p q* @ *tl* (*smods-ext pp* (*pderiv pp*))
  unfolding *pp-def* **using** *assms*(*2*,*3*)
proof (*induct smods-ext p q arbitrary*:*p q rule*:*length-induct*)
  case *1*
  have *?case* **when** *p mod q ≠0*
  proof −
    define *pp* **where** *pp=last* (*smods q* (− (*p mod q*)))
    have *smods-cons*:*smods p q* = *p#* *smods q* (− (*p mod q*))
      using ‹*p≠0*› **by** *auto*
    then have *pp-last*:*pp=last* (*smods p q*) **unfolding** *pp-def*
      by (*simp add*: *1.prems*(*2*) *pp-def*)
    have *smods-ext-cons*:*smods-ext p q* = *p* # *smods-ext q* (− (*p mod q*))
      using *that* ‹*p≠0*› **by** *auto*
    have *smods-ext q* (− (*p mod q*)) = *smods q* (− (*p mod q*)) @ *tl* (*smods-ext pp* (*pderiv pp*))
     apply (*rule 1*(*1*)[*rule-format*,*of smods-ext q* (− (*p mod q*)) *q* − (*p mod q*),*folded pp-def*])
      using *smods-ext-cons* ‹*q≠0*› *that* **by** *auto*
    then show *?thesis* **unfolding** *pp-last*
      apply (*subst smods-cons*)
      apply (*subst smods-ext-cons*)
      by *auto*
  qed
  moreover have *?case* **when** *p mod q =0 pderiv q* = *0*
  proof −

```
    have smods p q = [p,q]
      using ‹p≠0› ‹q≠0› that by auto
    moreover have smods-ext p q = [p,q]
      using that ‹p≠0› by auto
    ultimately show ?case using ‹p≠0› ‹q≠0› that(1) by auto
  qed
  moreover have ?case when p mod q =0 pderiv q ≠ 0
  proof –
    have smods-cons:smods p q = [p,q]
      using ‹p≠0› ‹q≠0› that by auto
    have smods-ext-cons:smods-ext p q = p#smods-ext q (pderiv q)
      using that ‹p≠0› by auto
    show ?case unfolding smods-cons smods-ext-cons
      apply (simp del:smods-ext.simps)
      by (simp add: 1.prems(2))
  qed
  ultimately show ?case by argo
qed

lemma no-0-in-smods-ext: 0∉set (smods-ext p q)
  apply (induct smods-ext p q arbitrary:p q)
   apply simp
  by (metis list.distinct(1) list.inject set-ConsD smods-ext.simps)
```

3.3 Sign variations on the alternative signed remainder sequences

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

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

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

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

3.4 Extension of Sturm's theorem for multiple roots

```
theorem sturm-ext-interval:
  assumes a<b poly p a≠0 poly p b≠0
  shows proots-count p {x. a<x ∧ x<b} = changes-itv-smods-ext a b p (pderiv p)
```

```
  using assms(2,3)
proof (induct smods-ext p (pderiv p) arbitrary:p rule:length-induct)
  case 1
  have p≠0 using ‹poly p a ≠ 0› by auto
  have ?case when pderiv p=0
  proof −
    obtain c where p=[:c:] c≠0
      using ‹p≠0› ‹pderiv p = 0› pderiv-iszero by force
    then have proots-count p {x. a < x ∧ x < b} = 0
      unfolding proots-count-def by auto
    moreover have changes-itv-smods-ext a b p (pderiv p) = 0
      unfolding changes-itv-smods-ext-def using ‹p=[:c:]› ‹c≠0› by auto
    ultimately show ?thesis by auto
  qed
  moreover have ?case when pderiv p≠0
  proof −
    define pp where pp = last (smods p (pderiv p))
    define lp where lp = lead-coeff pp
    define S where S={x. a < x ∧ x< b}

     have prefix:smods-ext p (pderiv p) = smods p (pderiv p) @ tl (smods-ext pp
(pderiv pp))
      using smods-ext-prefix[OF ‹p≠0› ‹pderiv p≠0›,folded pp-def] .
    have pp-gcd:pp = smult lp (gcd p (pderiv p))
      using last-smods-gcd[OF ‹p≠0›,of pderiv p,folded pp-def lp-def] .
    have pp≠0 lp≠0 unfolding pp-def lp-def
      subgoal by (rule last-smods-nzero[OF ‹p≠0›])
      subgoal using ‹last (smods p (pderiv p)) ≠ 0› by auto
      done
    have poly pp a≠0 poly pp b ≠ 0
      unfolding pp-gcd using ‹poly p a≠0› ‹poly p b≠0› ‹lp≠0›
      by (simp-all add:poly-gcd-0-iff)

    have proots-count pp S = changes-itv-smods-ext a b pp (pderiv pp) unfolding
S-def
    proof (rule 1(1)[rule-format,of smods-ext pp (pderiv pp) pp])
      show length (smods-ext pp (pderiv pp)) < length (smods-ext p (pderiv p))
        unfolding prefix by (simp add: ‹p ≠ 0› that)
    qed (use ‹poly pp a≠0› ‹poly pp b≠0› in simp-all)
    moreover have proots-count p S = card (proots-within p S) + proots-count pp
S
    proof −
      have (∑r∈proots-within p S. order r p) = (∑r∈ proots-within p S. order r
pp + 1)
      proof (rule sum.cong)
        fix x assume x ∈ proots-within p S
        have order x pp = order x (gcd p (pderiv p))
          unfolding pp-gcd using ‹lp≠0› by (simp add:order-smult)
        also have ... = min (order x p) (order x (pderiv p))
```

```
        apply (subst order-gcd)
        using ‹p≠0› ‹pderiv p≠0› by simp-all
      also have ... = order x (pderiv p)
        apply (subst order-pderiv)
        using ‹pderiv p≠0› ‹p ≠ 0› ‹x ∈ proots-within p S› order-root by auto
      finally have order x pp = order x (pderiv p) .
      moreover have order x p = order x (pderiv p) + 1
        apply (subst order-pderiv)
        using ‹pderiv p≠0› ‹p ≠ 0› ‹x ∈ proots-within p S› order-root by auto
      ultimately show order x p = order x pp + 1 by auto
    qed simp
    also have ... = card (proots-within p S) + (∑ r∈ proots-within p S. order r
pp)
      apply (subst sum.distrib)
      by auto
    also have ... = card (proots-within p S) + (∑ r∈ proots-within pp S. order r
pp)
    proof −
      have (∑ r∈proots-within p S. order r pp) = (∑ r∈proots-within pp S. order
r pp)
        apply (rule sum.mono-neutral-right)
        subgoal using ‹p≠0› by auto
        subgoal unfolding pp-gcd using ‹lp≠0› by (auto simp:poly-gcd-0-iff)
        subgoal unfolding pp-gcd using ‹lp≠0›
          apply (auto simp:poly-gcd-0-iff order-smult)
          apply (subst order-gcd)
          by (auto simp add: order-root)
        done
      then show ?thesis by simp
    qed
    finally show ?thesis unfolding proots-count-def .
  qed
  moreover have card (proots-within p S) = changes-itv-smods a b p (pderiv p)
    using sturm-interval[OF ‹a<b› ‹poly p a≠0› ‹poly p b≠0›,symmetric]
    unfolding S-def proots-within-def
    by (auto intro!:arg-cong[where f=card])
  moreover have changes-itv-smods-ext a b p (pderiv p)
          = changes-itv-smods a b p (pderiv p) + changes-itv-smods-ext a b pp
(pderiv pp)
  proof −
    define xs ys where xs=smods p (pderiv p) and ys=smods-ext pp (pderiv pp)
    have xys: xs≠[] ys≠[] last xs=hd ys poly (last xs) a≠0 poly (last xs) b≠0
      subgoal unfolding xs-def using ‹p≠0› by auto
      subgoal unfolding ys-def using ‹pp≠0› by auto
      subgoal using ‹pp≠0› unfolding xs-def ys-def
        apply (fold pp-def)
        by auto
      subgoal using ‹poly pp a≠0› unfolding pp-def xs-def .
      subgoal using ‹poly pp b≠0› unfolding pp-def xs-def .
```

```
        done
      have changes-poly-at (xs @ tl ys) a = changes-poly-at xs a + changes-poly-at
ys a
      proof -
        have changes-poly-at (xs @ tl ys) a  = changes-poly-at (xs @ ys) a
          unfolding changes-poly-at-def
          apply (simp add:map-tl)
          apply (subst changes-drop-dup[symmetric])
          using that xys by (auto simp add: hd-map last-map)
        also have ... = changes-poly-at xs a + changes-poly-at ys a
          unfolding changes-poly-at-def
          apply (subst changes-append[symmetric])
          using xys by (auto simp add: hd-map last-map)
        finally show ?thesis .
      qed
      moreover have changes-poly-at (xs @ tl ys) b = changes-poly-at xs b +
changes-poly-at ys b
      proof -
        have changes-poly-at (xs @ tl ys) b  = changes-poly-at (xs @ ys) b
          unfolding changes-poly-at-def
          apply (simp add:map-tl)
          apply (subst changes-drop-dup[symmetric])
          using that xys by (auto simp add: hd-map last-map)
        also have ... = changes-poly-at xs b + changes-poly-at ys b
          unfolding changes-poly-at-def
          apply (subst changes-append[symmetric])
          using xys by (auto simp add: hd-map last-map)
        finally show ?thesis .
      qed
    ultimately show ?thesis unfolding changes-itv-smods-ext-def changes-itv-smods-def
        apply (fold xs-def ys-def,unfold prefix[folded xs-def ys-def] Let-def)
        by auto
    qed
    ultimately show proots-count p S = changes-itv-smods-ext a b p (pderiv p)
      by auto
  qed
  ultimately show ?case by argo
qed

theorem sturm-ext-above:
  assumes poly p a≠0
  shows proots-count p {x. a<x} = changes-gt-smods-ext a p (pderiv p)
proof -
  define ps where ps≡smods-ext p (pderiv p)
  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-ext,of p pderiv p,folded ps-def]
```

```
    by auto
  have proots-count p {x. a<x} = proots-count p {x. a<x ∧ x<ub}
    unfolding proots-count-def
    apply (rule sum.cong)
    by (use ub ‹p∈set ps› in auto)
  moreover have changes-gt-smods-ext a p (pderiv p) = changes-itv-smods-ext a
ub p (pderiv p)
  proof −
    have map (sgn ∘ (λp. 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-ext-def changes-itv-smods-ext-def
ps-def
      by metis
  qed
  moreover have poly p ub≠0 using ub ‹p∈set ps› by auto
  ultimately show ?thesis using sturm-ext-interval[OF ‹ub>a› assms] by auto
qed

theorem sturm-ext-below:
  assumes poly p b≠0
  shows proots-count p {x. x<b} = changes-le-smods-ext b p (pderiv p)
proof −
  define ps where ps≡smods-ext p (pderiv p)
  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-ext,of p pderiv p,folded ps-def]
    by auto
  have proots-count p {x. x<b} = proots-count p {x. lb<x ∧ x<b}
    unfolding proots-count-def by (rule sum.cong,insert lb ‹p∈set ps›,auto)
  moreover have changes-le-smods-ext b p (pderiv p) = changes-itv-smods-ext lb
b p (pderiv p)
  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-ext-def changes-itv-smods-ext-def ps-def
      by metis
  qed
  moreover have poly p lb≠0 using lb ‹p∈set ps› by auto
  ultimately show ?thesis using sturm-ext-interval[OF ‹lb<b› - assms] by auto
```

qed

theorem *sturm-ext-R*:
  assumes $p \neq 0$
  shows *proots-count p UNIV = changes-R-smods-ext p (pderiv p)*
proof –
  define *ps* **where** *ps≡smods-ext p (pderiv p)*
  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-ext,of p pderiv p,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-ext,of p pderiv p,folded ps-def]*
    by *auto*
  have *proots-count p UNIV = proots-count p {x. lb<x ∧ x<ub}*
    unfolding *proots-count-def* **by** *(rule sum.cong,insert lb ub ‹p∈set ps›,auto)*
  moreover have *changes-R-smods-ext p (pderiv p) = changes-itv-smods-ext lb ub p (pderiv p)*
  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-ext-def changes-itv-smods-ext-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-ext-interval* **by** *auto*
qed

end

4 Descartes Roots Test

theory *Descartes-Roots-Test* **imports** *Budan-Fourier*
begin

The Descartes roots test is a consequence of Descartes' rule of signs: through counting sign variations on coefficients of a base-transformed (i.e. Taylor shifted) polynomial, it can over-approximate the number of real roots (counting multiplicity) within an interval. Its ability is similar to the Budan–Fourier theorem, but is far more efficient in practice. Therefore, this test is widely used in modern root isolation procedures.

More information can be found in the wiki page about Vincent's theorem: https://en.wikipedia.org/wiki/Vincent%27s_theorem and Collins and Akritas's classic paper of root isolation: Collins, G.E., Akritas, A.G.: Polynomial real root isolation using Descarte's rule of signs. SYMSACC. 272–275 (1976). A more modern treatment is available from a recent implementation of isolating real roots: Kobel, A., Rouillier, F., Sagraloff, M.: Computing Real Roots of Real Polynomials ... and now For Real! Proceedings of ISSAC '16, New York, New York, USA (2016).

```
lemma bij-betw-pos-interval:
  fixes a b::real
  assumes a<b
  shows bij-betw (λx. (a+b ∗ x) / (1+x)) {x. x>0} {x. a<x ∧ x<b}
proof (rule bij-betw-imageI)
  show inj-on (λx. (a + b ∗ x) / (1 + x)) {x. 0 < x}
    unfolding inj-on-def
    apply (auto simp add:field-simps)
    using assms crossproduct-noteq by fastforce
  have x ∈ (λx. (a + b ∗ x) / (1 + x)) ‘ {x. 0 < x} when a < x x < b for x
  proof (rule rev-image-eqI[of (x−a)/(b−x)])
    define bx where bx=b−x
    have x:x=b−bx unfolding bx-def by auto
    have bx≠0 b>a unfolding bx-def using that by auto
    then show  x = (a + b ∗ ((x − a) / (b − x))) / (1 + (x − a) / (b − x))
      apply (fold bx-def,unfold x)
      by (auto simp add:field-simps)
    show (x − a) / (b − x) ∈ {x. 0 < x} using that by auto
  qed
  then show (λx. (a + b ∗ x) / (1 + x)) ‘ {x. 0 < x} = {x. a < x ∧ x < b}
    using assms by (auto simp add:divide-simps algebra-simps)
qed

lemma proots-sphere-pos-interval:
  fixes a b::real
  defines q1≡[:a,b:] and q2≡[:1,1:]
  assumes p≠0 a<b
  shows proots-count p {x. a < x ∧ x < b} = proots-count (fcompose p q1 q2) {x.
0 < x}
  apply (rule proots-fcompose-bij-eq[OF - ‹p≠0›])
  unfolding q1-def q2-def using bij-betw-pos-interval[OF ‹a<b›] ‹a<b›
  by (auto simp add:algebra-simps infinite-UNIV-char-0)

definition descartes-roots-test::real ⇒ real ⇒ real poly ⇒ nat where
  descartes-roots-test a b p = nat (changes (coeffs (fcompose p [:a,b:] [:1,1:])))

theorem descartes-roots-test:
```

fixes *p*::*real poly*
assumes $p{\neq}0$ $a{<}b$
shows *proots-count p* $\{x.\ a < x \wedge x < b\} \leq$ *descartes-roots-test a b p* $\wedge$
even (*descartes-roots-test a b p* $-$ *proots-count p* $\{x.\ a < x \wedge x < b\}$)
proof $-$
define *q* **where** *q=fcompose p* [:*a*,*b*:] [:*1*,*1*:]
have $q{\neq}0$
unfolding *q-def*
apply (*rule fcompose-nzero*[*OF* ‹$p{\neq}0$›])
using ‹$a{<}b$› *infinite-UNIV-char-0* **by** *auto*
have *proots-count p* $\{x.\ a < x \wedge x < b\}$ = *proots-count q* $\{x.\ 0 < x\}$
using *proots-sphere-pos-interval*[*OF* ‹$p{\neq}0$› ‹$a{<}b$›,*folded q-def*] **.**
moreover have *int* (*proots-count q* $\{x.\ 0 < x\}$) $\leq$ *changes* (*coeffs q*) $\wedge$
even (*changes* (*coeffs q*) $-$ *int* (*proots-count q* $\{x.\ 0 < x\}$))
by (*rule descartes-sign*[*OF* ‹$q{\neq}0$›])
then have *proots-count q* $\{x.\ 0 < x\} \leq$ *nat* (*changes* (*coeffs q*)) $\wedge$
even (*nat* (*changes* (*coeffs q*)) $-$ *proots-count q* $\{x.\ 0 < x\}$)
using *even-nat-iff* **by** *auto*
ultimately show *?thesis*
unfolding *descartes-roots-test-def*
apply (*fold q-def*)
by *auto*
qed

The roots test *descartes-roots-test* is exact if its result is 0 or 1.

corollary *descartes-roots-test-zero*:
fixes *p*::*real poly*
assumes $p{\neq}0$ $a{<}b$ *descartes-roots-test a b p* = *0*
shows $\forall x.\ a{<}x \wedge x{<}b \longrightarrow$ *poly p* $x{\neq}0$
proof $-$
have *proots-count p* $\{x.\ a < x \wedge x < b\}$ = *0*
using *descartes-roots-test*[*OF assms*(*1*,*2*)] *assms*(*3*) **by** *auto*
from *proots-count-0-imp-empty*[*OF this* ‹$p{\neq}0$›]
show *?thesis* **by** *auto*
qed

corollary *descartes-roots-test-one*:
fixes *p*::*real poly*
assumes $p{\neq}0$ $a{<}b$ *descartes-roots-test a b p* = *1*
shows *proots-count p* $\{x.\ a < x \wedge x < b\}$ = *1*
using *descartes-roots-test*[*OF* ‹$p{\neq}0$› ‹$a{<}b$›] ‹*descartes-roots-test a b p* = *1*›
by (*metis dvd-diffD even-zero le-neq-implies-less less-one odd-one*)

Similar to the Budan–Fourier theorem, the Descartes roots test result is exact when all roots are real.

corollary *descartes-roots-test-real*:
fixes *p*::*real poly*
assumes $p{\neq}0$ $a{<}b$
assumes *all-roots-real p*

```
  shows proots-count p {x. a < x ∧ x <b} = descartes-roots-test a b p
proof −
  define q where q=fcompose p [:a,b:] [:1,1:]
  have q≠0
    unfolding q-def
    apply (rule fcompose-nzero[OF ‹p≠0›])
    using ‹a<b› infinite-UNIV-char-0 by auto
  have proots-count p {x. a < x ∧ x <b} = proots-count q {x. 0 < x}
    using proots-sphere-pos-interval[OF ‹p≠0› ‹a<b›,folded q-def] .
  moreover have int (proots-count q {x. 0 < x}) = changes (coeffs q)
    apply (rule descartes-sign-real[OF ‹q≠0›])
    unfolding q-def by (rule all-real-roots-mobius[OF ‹all-roots-real p› ‹a<b›])
  then have proots-count q {x. 0 < x} = nat (changes (coeffs q))
    by simp
  ultimately show ?thesis unfolding descartes-roots-test-def
    apply (fold q-def)
    by auto
qed

end
```

5 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] S. Basu, R. Pollack, and M.-F. Roy. *Algorithms in Real Algebraic Geometry*, volume 10 of *Algorithms and Computation in Mathematics*. Springer Berlin Heidelberg, Berlin, Heidelberg, 2006.

[2] M. Eberl. Sturm's theorem. *Archive of Formal Proofs*, Jan. 2014. http://isa-afp.org/entries/Sturm_Sequences.html, Formal proof development.

[3] M. Eberl. Descartes' rule of signs. *Archive of Formal Proofs*, Dec. 2015. http://isa-afp.org/entries/Descartes_Sign_Rule.html, Formal proof development.

[4] J. Harrison. Verifying the accuracy of polynomial approximations in HOL. In E. L. Gunter and A. Felty, editors, *Theorem Proving in Higher Order Logics: 10th International Conference, TPHOLs'97*, volume 1275 of *Lecture Notes in Computer Science*, pages 137–152, Murray Hill, NJ, 1997. Springer-Verlag.

[5] W. Li. The Sturm–Tarski Theorem. *Archive of Formal Proofs*, Sept. 2014.

[6] W. Li. Count the Number of Complex Roots. *Archive of Formal Proofs*, Oct. 2017.

[7] W. Li and L. C. Paulson. Evaluating Winding Numbers and Counting Complex Roots through Cauchy Indices in Isabelle/HOL. *CoRR*, abs/1804.03922, 2018.

[8] A. Mahboubi and C. Cohen. Formal proofs in real algebraic geometry: from ordered fields to quantifier elimination. *Logical Methods in Computer Science*, 8(1), 2012.

[9] A. Narkawicz, C. A. Muñoz, and A. Dutle. Formally-Verified Decision Procedures for Univariate Polynomial Computation Based on Sturm's and Tarski's Theorems. *Journal of Automated Reasoning*, 54(4):285–326, 2015.

[10] Q. I. Rahman and G. Schmeisser. *Analytic Theory of Polynomials*. Oxford University Press, 2002.