A Formalisation of Sturm's Theorem

Manuel Eberl

March 19, 2025

Abstract

Sturm sequences are a method for computing the number of real roots of a real polynomial inside a given interval efficiently. In this project, this fact and a number of methods to construct Sturm sequences efficiently have been formalised with the interactive theorem prover Isabelle/HOL. Building upon this, an Isabelle/HOL proof method was then implemented to prove statements about the number of roots of a real polynomial and related properties.

Contents

1 Miscellaneous

theory *Misc-Polynomial*
imports *HOL−Computational-Algebra.Polynomial HOL−Computational-Algebra.Polynomial-Factorial Pure−ex.Guess*
begin

1.1 Analysis

lemma *fun-eq-in-ivl*:
  assumes $a \le b$ $\forall x::real.\ a \le x \wedge x \le b \longrightarrow eventually\ (\lambda\xi.\ f\ \xi = f\ x)\ (at\ x)$
  shows $f\ a = f\ b$
proof (*rule connected-local-const*)
  show *connected* $\{a..b\}$ $a \in \{a..b\}$ $b \in \{a..b\}$ **using** ‹$a \le b$› **by** (*auto intro*: *connected-Icc*)
  show $\forall aa \in \{a..b\}.\ eventually\ (\lambda b.\ f\ aa = f\ b)\ (at\ aa\ within\ \{a..b\})$
  proof
    fix x **assume** $x \in \{a..b\}$
    with *assms*(*2*)[*rule-format*, *of x*]
    show $eventually\ (\lambda b.\ f\ x = f\ b)\ (at\ x\ within\ \{a..b\})$
      by (*auto simp*: *eventually-at-filter elim*: *eventually-mono*)
  qed
qed

1.2 Polynomials

1.2.1 General simplification lemmas

lemma *pderiv-div*:
  assumes [*simp*]: $q\ dvd\ p$ $q \ne 0$
  shows $pderiv\ (p\ div\ q) = (q * pderiv\ p - p * pderiv\ q)\ div\ (q * q)$
      $q*q\ dvd\ (q * pderiv\ p - p * pderiv\ q)$
proof−
  from *assms* **obtain** r **where** $p = q * r$ **unfolding** *dvd-def* **by** *blast*
  hence $q * pderiv\ p - p * pderiv\ q = (q * q) * pderiv\ r$
    by (*simp add*: *algebra-simps pderiv-mult*)
  thus $q*q\ dvd\ (q * pderiv\ p - p * pderiv\ q)$ **by** *simp*
  note *0* = *pderiv-mult*[*of q p div q*]
  have *1*: $q * (p\ div\ q) = p$
    by (*metis assms*(*1*) *assms*(*2*) *dvd-def nonzero-mult-div-cancel-left*)
  have *f1*: $pderiv\ (p\ div\ q) * (q * q)\ div\ (q * q) = pderiv\ (p\ div\ q)$
    by *simp*
  have *f2*: $pderiv\ p = q * pderiv\ (p\ div\ q) + p\ div\ q * pderiv\ q$
    by (*metis 0 1*)
  have $p * pderiv\ q = pderiv\ q * (q * (p\ div\ q))$
    by (*metis 1 mult.commute*)
  then have $p * pderiv\ q = q * (p\ div\ q * pderiv\ q)$
    by *fastforce*
  then have $q * pderiv\ p - p * pderiv\ q = q * (q * pderiv\ (p\ div\ q))$

using *f2* **by** (*metis add-diff-cancel-right′ distrib-left*)
then show *pderiv* (*p div q*) = (*q* ∗ *pderiv p* − *p* ∗ *pderiv q*) *div* (*q* ∗ *q*)
using *f1* **by** (*metis mult.commute mult.left-commute*)
qed

1.2.2 Divisibility of polynomials

Two polynomials that are coprime have no common roots.

lemma *coprime-imp-no-common-roots*:
¬ (*poly p x* = *0* ∧ *poly q x* = *0*) **if** *coprime p q*
for *x* :: *′a* :: *field*
proof *clarify*
assume *poly p x* = *0 poly q x* = *0*
then have [:−*x*, *1*:] *dvd p* [:−*x*, *1*:] *dvd q*
by (*simp-all add*: *poly-eq-0-iff-dvd*)
with *that* **have** *is-unit* [:−*x*, *1*:]
by (*rule coprime-common-divisor*)
then show *False*
by (*auto simp add*: *is-unit-pCons-iff*)
qed

lemma *poly-div*:
assumes *poly q x* ≠ *0* **and** (*q*::*′a* :: *field poly*) *dvd p*
shows *poly* (*p div q*) *x* = *poly p x* / *poly q x*
proof−
from *assms* **have** [*simp*]: *q* ≠ *0* **by** *force*
have *poly q x* ∗ *poly* (*p div q*) *x* = *poly* (*q* ∗ (*p div q*)) *x* **by** *simp*
also have *q* ∗ (*p div q*) = *p*
using *assms* **by** (*simp add*: *div-mult-swap*)
finally show *poly* (*p div q*) *x* = *poly p x* / *poly q x*
using *assms* **by** (*simp add*: *field-simps*)
qed

lemma *poly-div-gcd-squarefree-aux*:
assumes *pderiv* (*p*::(*′a*::{*field-char-0*,*field-gcd*}) *poly*) ≠ *0*
defines *d* ≡ *gcd p* (*pderiv p*)
shows *coprime* (*p div d*) (*pderiv* (*p div d*)) **and**
⋀*x*. *poly* (*p div d*) *x* = *0* ⟷ *poly p x* = *0*
proof −
obtain *r s* **where** *bezout-coefficients p* (*pderiv p*) = (*r*, *s*)
by (*auto simp add*: *prod-eq-iff*)
then have *r* ∗ *p* + *s* ∗ *pderiv p* = *gcd p* (*pderiv p*)
by (*rule bezout-coefficients*)
then have *rs*: *d* = *r* ∗ *p* + *s* ∗ *pderiv p*
by (*simp add*: *d-def*)
define *t* **where** *t* = *p div d*
define *p′* **where** [*simp*]: *p′* = *pderiv p*
define *d′* **where** [*simp*]: *d′* = *pderiv d*

```
define u where u = p′ div d
have A: p = t * d and B: p′ = u * d
  by (simp-all add: t-def u-def d-def algebra-simps)
from poly-squarefree-decomp[OF assms(1) A B[unfolded p′-def] rs]
  show ⋀x. poly (p div d) x = 0 ⟷ poly p x = 0 by (auto simp: t-def)

from rs have C: s*t*d′ = d * (1 − r*t − s*pderiv t)
    by (simp add: A B algebra-simps pderiv-mult)
from assms have [simp]: p ≠ 0 d ≠ 0 t ≠ 0
    by (force, force, subst (asm) A, force)

have ⋀x. ⟦x dvd t; x dvd (pderiv t)⟧ ⟹ x dvd 1
proof −
  fix x assume x dvd t x dvd (pderiv t)
  then obtain v w where vw:
      t = x*v pderiv t = x*w unfolding dvd-def by blast
  define x′ v′ where [simp]: x′ = pderiv x and [simp]: v′ = pderiv v
  from vw have x*v′ + v*x′ = x*w by (simp add: pderiv-mult)
  hence v*x′ = x*(w − v′) by (simp add: algebra-simps)
  hence x dvd v*pderiv x by simp
  then obtain y where y: v*x′ = x*y unfolding dvd-def by force
  from ‹t ≠ 0› and vw have x ≠ 0 by simp

  have x-pow-n-dvd-d: ⋀n. x^n dvd d
  proof−
    fix n show x ^ n dvd d
    proof (induction n, simp, rename-tac n, case-tac n)
      fix n assume n = (0::nat)
      from vw and C have d = x*(d*r*v + d*s*w + s*v*d′)
          by (simp add: algebra-simps)
      with ‹n = 0› show x^Suc n dvd d by (force intro: dvdI)
    next
      fix n n′ assume IH: x^n dvd d and n = Suc n′
      hence [simp]: Suc n′ = n x * x^n′ = x^n by simp-all
      define c :: ′a poly where c = [:of-nat n:]
      from pderiv-power-Suc[of x n′]
          have [simp]: pderiv (x^n) = c*x^n′ * x′ unfolding c-def
          by simp

      from IH obtain z where d: d = x^n * z unfolding dvd-def by blast
      define z′ where [simp]: z′ = pderiv z
      from d ‹d ≠ 0› have x^n ≠ 0 z ≠ 0 by force+
      from C d have x^n*z = z*r*v*x^Suc n + z*s*c*x^n*(v*x′) +
                    s*v*z′*x^Suc n + s*z*(v*x′)*x^n + s*z*v′*x^Suc n
          by (simp add: algebra-simps vw pderiv-mult)
      also have ... = x^n*x * (z*r*v + z*s*c*y + s*v*z′ + s*z*y + s*z*v′)
          by (simp only: y, simp add: algebra-simps)
      finally have z = x*(z*r*v+z*s*c*y+s*v*z′+s*z*y+s*z*v′)
           using ‹x^n ≠ 0› by force
```

hence *x dvd z* **by** (*metis dvd-triv-left*)
with *d* **show** *x^Suc n dvd d* **by** *simp*
qed
qed

have *degree x = 0*
proof (*cases degree x*, *simp*)
case (*Suc n*)
hence *x ≠ 0* **by** *auto*
with *Suc* **have** *degree* (*x ^* (*Suc* (*degree d*))) > *degree d*
by (*subst degree-power-eq*, *simp-all*)
moreover from *x-pow-n-dvd-d*[*of Suc* (*degree d*)] **and** ‹*d ≠ 0*›
have *degree* (*x^Suc* (*degree d*)) ≤ *degree d*
by (*simp add*: *dvd-imp-degree-le*)
ultimately show *?thesis* **by** *simp*
qed
then obtain *c* **where** [*simp*]: *x* = [:*c*:] **by** (*cases x*, *simp split*: *if-split-asm*)
moreover from ‹*x ≠ 0*› **have** *c ≠ 0* **by** *simp*
ultimately show *x dvd 1* **using** *dvdI*[*of 1 x* [:*inverse c*:]]
by *simp*
qed

then show *coprime t* (*pderiv t*)
by (*rule coprimeI*)
qed

lemma *normalize-field*:
normalize (*x* :: *'a* :: {*field*,*normalization-semidom*}) = (*if x = 0 then 0 else 1*)
by (*auto simp*: *is-unit-normalize dvd-field-iff*)

lemma *normalize-field-eq-1* [*simp*]:
x ≠ 0 ⟹ *normalize* (*x* :: *'a* :: {*field*,*normalization-semidom*}) = *1*
by (*simp add*: *normalize-field*)

lemma *unit-factor-field* [*simp*]:
unit-factor (*x* :: *'a* :: {*field*,*normalization-semidom*}) = *x*
by (*cases x = 0*) (*auto simp*: *is-unit-unit-factor dvd-field-iff*)

Dividing a polynomial by its gcd with its derivative yields a squarefree polynomial with the same roots.

lemma *poly-div-gcd-squarefree*:
assumes (*p* :: (*'a*::{*field-char-0*,*field-gcd*}) *poly*) ≠ *0*
defines *d* ≡ *gcd p* (*pderiv p*)
shows *coprime* (*p div d*) (*pderiv* (*p div d*)) (**is** *?A*) **and**
⋀*x*. *poly* (*p div d*) *x* = *0* ⟷ *poly p x* = *0* (**is** ⋀*x*. *?B x*)
proof−
have *?A* ∧ (∀ *x*. *?B x*)
proof (*cases pderiv p = 0*)
case *False*

```
    from poly-div-gcd-squarefree-aux[OF this] show ?thesis
      unfolding d-def by auto
  next
    case True
    then obtain c where [simp]: p = [:c:] using pderiv-iszero by blast
    from assms(1) have c ≠ 0 by simp
    from True have d = smult (inverse c) p
      by (simp add: d-def normalize-poly-def map-poly-pCons field-simps)
    with ‹p ≠ 0› ‹c ≠ 0› have p div d = [:c:]
      by (simp add: pCons-one)
    with ‹c ≠ 0› show ?thesis
      by (simp add: normalize-const-poly is-unit-triv)
  qed
  thus ?A and ⋀x. ?B x by simp-all
qed
```

1.2.3 Sign changes of a polynomial

If a polynomial has different signs at two points, it has a root inbetween.

```
lemma poly-different-sign-imp-root:
  assumes a < b and sgn (poly p a) ≠ sgn (poly p (b::real))
  shows ∃x. a ≤ x ∧ x ≤ b ∧ poly p x = 0
proof (cases poly p a = 0 ∨ poly p b = 0)
  case True
    thus ?thesis using assms(1)
      by (elim disjE, rule-tac exI[of - a], simp,
                      rule-tac exI[of - b], simp)
next
  case False
    hence [simp]: poly p a ≠ 0 poly p b ≠ 0 by simp-all
    show ?thesis
    proof (cases poly p a < 0)
      case True
        hence sgn (poly p a) = −1 by simp
        with assms True have poly p b > 0
          by (auto simp: sgn-real-def split: if-split-asm)
        from poly-IVT-pos[OF ‹a < b› True this] guess x ..
        thus ?thesis by (intro exI[of - x], simp)
    next
      case False
        hence poly p a > 0 by (simp add: not-less less-eq-real-def)
        hence sgn (poly p a) = 1 by simp
        with assms False have poly p b < 0
          by (auto simp: sgn-real-def not-less
                         less-eq-real-def split: if-split-asm)
        from poly-IVT-neg[OF ‹a < b› ‹poly p a > 0› this] guess x ..
        thus ?thesis by (intro exI[of - x], simp)
    qed
qed
```

lemma *poly-different-sign-imp-root'*:
  assumes *sgn* (*poly p a*) $\neq$ *sgn* (*poly p* (*b*::*real*))
  shows $\exists$ *x*. *poly p x* = *0*
using *assms* **by** (*cases a* < *b*, *auto dest*!: *poly-different-sign-imp-root*
                                    simp: *less-eq-real-def not-less*)

lemma *no-roots-inbetween-imp-same-sign*:
  assumes *a* < *b* $\forall$ *x*. *a* $\leq$ *x* $\wedge$ *x* $\leq$ *b* $\longrightarrow$ *poly p x* $\neq$ (*0*::*real*)
  shows *sgn* (*poly p a*) = *sgn* (*poly p b*)
  using *poly-different-sign-imp-root assms* **by** *auto*

1.2.4 Limits of polynomials

lemma *poly-neighbourhood-without-roots*:
  assumes (*p* :: *real poly*) $\neq$ *0*
  shows *eventually* (λ*x*. *poly p x* $\neq$ *0*) (*at* x_0)
proof−
  {
    fix ε :: *real* **assume** ε > *0*
    have *fin*: *finite* {*x*. $|x-x_0|$ < ε $\wedge$ *x* $\neq$ x_0 $\wedge$ *poly p x* = *0*}
        using *poly-roots-finite*[*OF assms*] **by** *simp*
    with ‹ε > *0*›**have** $\exists\delta$>*0*. $\delta\leq\varepsilon$ $\wedge$ ($\forall$ *x*. $|x-x_0|$ < δ $\wedge$ *x* $\neq$ x_0 $\longrightarrow$ *poly p x* $\neq$ *0*)
    proof (*induction card* {*x*. $|x-x_0|$ < ε $\wedge$ *x* $\neq$ x_0 $\wedge$ *poly p x* = *0*}
          arbitrary: ε *rule*: *less-induct*)
    case (*less* ε)
    let *?A* = {*x*. $|x - x_0|$ < ε $\wedge$ *x* $\neq$ x_0 $\wedge$ *poly p x* = *0*}
    show *?case*
      proof (*cases card ?A*)
      case *0*
        hence *?A* = {} **using** *less* **by** *auto*
        thus *?thesis* **using** *less*(*2*) **by** (*rule-tac exI*[*of* - ε], *auto*)
      next
      case (*Suc* -)
       with *less*(*3*) **have** {*x*. $|x - x_0|$ < ε $\wedge$ *x* $\neq$ x_0 $\wedge$ *poly p x* = *0*} $\neq$ {} **by** *force*
        then obtain *x* **where** *x-props*: $|x - x_0|$ < ε *x* $\neq$ x_0 *poly p x* = *0* **by** *blast*
        define ε' **where** ε' = $|x - x_0|$ / *2*
        have ε' > *0* ε' < ε **unfolding** ε'*-def* **using** *x-props* **by** *simp-all*
        from *x-props*(*1*,*2*) **and** ‹ε > *0*›
            have *x* $\notin$ {*x'*. $|x' - x_0|$ < ε' $\wedge$ *x'* $\neq$ x_0 $\wedge$ *poly p x'* = *0*} (**is** - $\notin$ *?B*)
            by (*auto simp*: ε'*-def*)
        moreover from *x-props*
            have *x* $\in$ {*x*. $|x - x_0|$ < ε $\wedge$ *x* $\neq$ x_0 $\wedge$ *poly p x* = *0*} **by** *blast*
        ultimately have *?B* $\subset$ *?A* **by** *auto*
        hence *card ?B* < *card ?A finite ?B*
            by (*rule psubset-card-mono*[*OF less*(*3*)],
                blast intro: *finite-subset*[*OF* - *less*(*3*)])
        from *less*(*1*)[*OF this*(*1*) ‹ε' > *0*› *this*(*2*)]

```
          show ?thesis using ‹ε' < ε› by force
       qed
     qed
   }
   from this[of 1]
     show ?thesis by (auto simp: eventually-at dist-real-def)
qed


lemma poly-neighbourhood-same-sign:
  assumes poly p (x0 :: real) ≠ 0
  shows eventually (λx. sgn (poly p x) = sgn (poly p x0)) (at x0)
proof −
  have cont: isCont (λx. sgn (poly p x)) x0
      by (rule isCont-sgn, rule poly-isCont, rule assms)
  then have eventually (λx. |sgn (poly p x) − sgn (poly p x0)| < 1) (at x0)
      by (auto simp: isCont-def tendsto-iff dist-real-def)
  then show ?thesis
    by (rule eventually-mono) (simp add: sgn-real-def split: if-split-asm)
qed

lemma poly-lhopital:
  assumes poly p (x::real) = 0 poly q x = 0 q ≠ 0
  assumes (λx. poly (pderiv p) x / poly (pderiv q) x) −x→ y
  shows (λx. poly p x / poly q x) −x→ y
using assms
proof (rule-tac lhopital)
  have isCont (poly p) x isCont (poly q) x by simp-all
  with assms(1,2) show poly p −x→ 0 poly q −x→ 0
       by (simp-all add: isCont-def)
  from ‹q ≠ 0› and ‹poly q x = 0› have pderiv q ≠ 0
      by (auto dest: pderiv-iszero)
  from poly-neighbourhood-without-roots[OF this]
      show eventually (λx. poly (pderiv q) x ≠ 0) (at x) .
qed (auto intro: poly-DERIV poly-neighbourhood-without-roots)


lemma poly-roots-bounds:
  assumes p ≠ 0
  obtains l u
  where l ≤ (u :: real)
    and poly p l ≠ 0
    and poly p u ≠ 0
    and {x. x > l ∧ x ≤ u ∧ poly p x = 0 } = {x. poly p x = 0}
    and ⋀x. x ≤ l ⟹ sgn (poly p x) = sgn (poly p l)
    and ⋀x. x ≥ u ⟹ sgn (poly p x) = sgn (poly p u)
proof
  from assms have finite {x. poly p x = 0} (is finite ?roots)
      using poly-roots-finite by fast
```

```
  let ?roots' = insert 0 ?roots

  define l where l = Min ?roots' − 1
  define u where u = Max ?roots' + 1

  from ‹finite ?roots› have A: finite ?roots'  by auto
  from Min-le[OF this, of 0] and Max-ge[OF this, of 0]
    show l ≤  u by (simp add: l-def u-def)
  from Min-le[OF A] have l-props: ⋀x. x≤l ⟹ poly p x ≠ 0
    by (fastforce simp: l-def)
  from Max-ge[OF A] have u-props: ⋀x. x≥u ⟹ poly p x ≠ 0
    by (fastforce simp: u-def)
  from l-props u-props show [simp]: poly p l ≠ 0 poly p u ≠ 0 by auto

  from l-props have ⋀x. poly p x = 0 ⟹ x > l by (metis not-le)
  moreover from u-props have ⋀x. poly p x = 0 ⟹ x ≤ u by (metis linear)
  ultimately show {x. x > l ∧ x ≤ u ∧ poly p x = 0} = ?roots by auto

  {
    fix x assume A: x < l sgn (poly p x) ≠ sgn (poly p l)
    with poly-IVT-pos[OF A(1), of p] poly-IVT-neg[OF A(1), of p] A(2)
      have False by (auto split: if-split-asm
                      simp: sgn-real-def l-props not-less less-eq-real-def)
  }
  thus ⋀x. x ≤ l ⟹ sgn (poly p x) = sgn (poly p l)
    by (case-tac x = l, auto simp: less-eq-real-def)

  {
    fix x assume A: x > u sgn (poly p x) ≠ sgn (poly p u)
    with u-props poly-IVT-neg[OF A(1), of p] poly-IVT-pos[OF A(1), of p] A(2)
      have False by (auto split: if-split-asm
                      simp: sgn-real-def l-props not-less less-eq-real-def)
  }
  thus ⋀x. x ≥ u ⟹ sgn (poly p x) = sgn (poly p u)
    by (case-tac x = u, auto simp: less-eq-real-def)
qed


definition poly-inf :: ('a::real-normed-vector) poly ⇒ 'a where
  poly-inf p ≡ sgn (coeff p (degree p))
definition poly-neg-inf :: ('a::real-normed-vector) poly ⇒ 'a where
  poly-neg-inf p ≡ if even (degree p) then sgn (coeff p (degree p))
                                    else −sgn (coeff p (degree p))
lemma poly-inf-0-iff[simp]:
    poly-inf p = 0 ⟷ p = 0 poly-neg-inf p = 0 ⟷ p = 0
    by (auto simp: poly-inf-def poly-neg-inf-def sgn-zero-iff)

lemma poly-inf-mult[simp]:
```

fixes *p* :: (*'a*::*real-normed-field*) *poly*
shows *poly-inf* (*p*∗*q*) = *poly-inf* *p* ∗ *poly-inf* *q*
poly-neg-inf (*p*∗*q*) = *poly-neg-inf* *p* ∗ *poly-neg-inf* *q*
unfolding *poly-inf-def poly-neg-inf-def*
by ((*cases p* = *0* ∨ *q* = *0*,*auto simp*: *sgn-zero-iff*
degree-mult-eq[*of p q*] *coeff-mult-degree-sum Real-Vector-Spaces.sgn-mult*)[])+

lemma *poly-neq-0-at-infinity*:
assumes (*p* :: *real poly*) ≠ *0*
shows *eventually* (λ*x*. *poly p x* ≠ *0*) *at-infinity*
proof−
from *poly-roots-bounds*[*OF assms*] **guess** *l u* **.**
note *lu-props* = *this*
define *b* **where** *b* = *max* (−*l*) *u*
show *?thesis*
proof (*subst eventually-at-infinity*, *rule exI*[*of - b*], *clarsimp*)
fix *x* **assume** *A*: |*x*| ≥ *b* **and** *B*: *poly p x* = *0*
show *False*
proof (*cases x* ≥ *0*)
case *True*
with *A* **have** *x* ≥ *u* **unfolding** *b-def* **by** *simp*
with *lu-props*(*3*, *6*) **show** *False* **by** (*metis sgn-zero-iff B*)
next
case *False*
with *A* **have** *x* ≤ *l* **unfolding** *b-def* **by** *simp*
with *lu-props*(*2*, *5*) **show** *False* **by** (*metis sgn-zero-iff B*)
qed
qed
qed

lemma *poly-limit-aux*:
fixes *p* :: *real poly*
defines *n* ≡ *degree p*
shows ((λ*x*. *poly p x* / *x* ^ *n*) ⟶ *coeff p n*) *at-infinity*
proof (*subst filterlim-cong*, *rule refl*, *rule refl*)
show *eventually* (λ*x*. *poly p x* / *x*^*n* = (∑ *i*≤*n*. *coeff p i* / *x*^(*n*−*i*)))
at-infinity
proof (*rule eventually-mono*)
show *eventually* (λ*x*::*real*. *x* ≠ *0*) *at-infinity*
by (*simp add*: *eventually-at-infinity*, *rule exI*[*of - 1*], *auto*)
fix *x* :: *real* **assume** [*simp*]: *x* ≠ *0*
show *poly p x* / *x* ^ *n* = (∑ *i*≤*n*. *coeff p i* / *x* ^ (*n* − *i*))
by (*simp add*: *n-def sum-divide-distrib power-diff poly-altdef*)
qed

```
  let ?a = λi. if i = n then coeff p n else 0
  have ∀ i∈{..n}. ((λx. coeff p i / x ^ (n − i)) ⟶ ?a i) at-infinity
  proof
    fix i assume i ∈ {..n}
    hence i ≤ n by simp
    show ((λx. coeff p i / x ^ (n − i)) ⟶ ?a i) at-infinity
    proof (cases i = n)
      case True
        thus ?thesis by (intro tendstoI, subst eventually-at-infinity,
                         intro exI[of - 1], simp add: dist-real-def)
    next
      case False
        hence n − i > 0 using ‹i ≤ n› by simp
        from tendsto-inverse-0 and divide-real-def[of 1]
            have ((λx. 1 / x :: real) ⟶ 0) at-infinity by simp
        from tendsto-power[OF this, of n − i]
            have ((λx::real. 1 / x ^ (n − i)) ⟶ 0) at-infinity
                using ‹n − i > 0› by (simp add: power-0-left power-one-over)
        from tendsto-mult-right-zero[OF this, of coeff p i]
            have ((λx. coeff p i / x ^ (n − i)) ⟶ 0) at-infinity
                by (simp add: field-simps)
        thus ?thesis using False by simp
    qed
  qed
  hence ((λx. ∑ i≤n. coeff p i / x^(n−i)) ⟶ (∑ i≤n. ?a i)) at-infinity
    by (force intro!: tendsto-sum)
  also have (∑ i≤n. ?a i) = coeff p n by (subst sum.delta, simp-all)
  finally show ((λx. ∑ i≤n. coeff p i / x^(n−i)) ⟶ coeff p n) at-infinity .
qed


lemma poly-at-top-at-top:
  fixes p :: real poly
  assumes degree p ≥ 1 coeff p (degree p) > 0
  shows LIM x at-top. poly p x :> at-top
proof−
  let ?n = degree p
  define f g where f x = poly p x / x^?n and g x = x ^ ?n for x :: real

  from poly-limit-aux have (f ⟶ coeff p (degree p)) at-top
      using tendsto-mono at-top-le-at-infinity unfolding f-def by blast
  moreover from assms
      have LIM x at-top. g x :> at-top
        by (auto simp add: g-def intro!: filterlim-pow-at-top filterlim-ident)
  ultimately have LIM x at-top. f x ∗ g x :> at-top
      using filterlim-tendsto-pos-mult-at-top assms by simp
  also have eventually (λx. f x ∗ g x = poly p x) at-top
      unfolding f-def g-def
```

```
    by (subst eventually-at-top-linorder, rule exI[of - 1],
        simp add: poly-altdef field-simps sum-distrib-left power-diff)
  note filterlim-cong[OF refl refl this]
  finally show ?thesis .
qed

lemma poly-at-bot-at-top:
  fixes p :: real poly
  assumes degree p ≥ 1 coeff p (degree p) < 0
  shows LIM x at-top. poly p x :> at-bot
proof−
  from poly-at-top-at-top[of −p] and assms
    have LIM x at-top. −poly p x :> at-top by simp
  thus ?thesis by (simp add: filterlim-uminus-at-bot)
qed

lemma poly-lim-inf:
  eventually (λx::real. sgn (poly p x) = poly-inf p) at-top
proof (cases degree p ≥ 1)
  case False
    hence degree p = 0 by simp
    then obtain c where p = [:c:] by (cases p, auto split: if-split-asm)
    thus ?thesis
      by (simp add: eventually-at-top-linorder poly-inf-def)
next
  case True
    note deg = this
    let ?lc = coeff p (degree p)
    from True have ?lc ≠ 0 by force
    show ?thesis
    proof (cases ?lc > 0)
      case True
        from poly-at-top-at-top[OF deg this]
          obtain x0 where ⋀x. x ≥ x0 ⟹ poly p x ≥ 1
            by (fastforce simp: filterlim-at-top
                eventually-at-top-linorder less-eq-real-def)
        hence ⋀x. x ≥ x0 ⟹ sgn (poly p x) = 1 by force
        thus ?thesis by (simp only: eventually-at-top-linorder poly-inf-def,
                         intro exI[of - x0], simp add: True)
    next
      case False
        hence ?lc < 0 using ‹?lc ≠ 0› by linarith
        from poly-at-bot-at-top[OF deg this]
          obtain x0 where ⋀x. x ≥ x0 ⟹ poly p x ≤ −1
            by (fastforce simp: filterlim-at-bot
                eventually-at-top-linorder less-eq-real-def)
        hence ⋀x. x ≥ x0 ⟹ sgn (poly p x) = −1 by force
        thus ?thesis by (simp only: eventually-at-top-linorder poly-inf-def,
                         intro exI[of - x0], simp add: ‹?lc < 0›)
```

```
    qed
qed

lemma poly-at-top-or-bot-at-bot:
  fixes p :: real poly
  assumes degree p ≥ 1 coeff p (degree p) > 0
  shows LIM x at-bot. poly p x :> (if even (degree p) then at-top else at-bot)
proof−
  let ?n = degree p
  define f g where f x = poly p x / x ^ ?n and g x = x ^ ?n for x :: real

  from poly-limit-aux have (f ⟶ coeff p (degree p)) at-bot
      using tendsto-mono at-bot-le-at-infinity by (force simp: f-def [abs-def])
  moreover from assms
      have LIM x at-bot. g x :> (if even (degree p) then at-top else at-bot)
        by (auto simp add: g-def split: if-split-asm intro: filterlim-pow-at-bot-even
filterlim-pow-at-bot-odd filterlim-ident)
  ultimately have LIM x at-bot. f x ∗ g x :>
                    (if even ?n then at-top else at-bot)
      by (auto simp: assms intro: filterlim-tendsto-pos-mult-at-top
                                  filterlim-tendsto-pos-mult-at-bot)
  also have eventually (λx. f x ∗ g x = poly p x) at-bot
      unfolding f-def g-def
      by (subst eventually-at-bot-linorder, rule exI[of - −1],
          simp add: poly-altdef field-simps sum-distrib-left power-diff)
  note filterlim-cong[OF refl refl this]
  finally show ?thesis .
qed


lemma poly-at-bot-or-top-at-bot:
  fixes p :: real poly
  assumes degree p ≥ 1 coeff p (degree p) < 0
  shows LIM x at-bot. poly p x :> (if even (degree p) then at-bot else at-top)
proof−
  from poly-at-top-or-bot-at-bot[of −p] and assms
      have LIM x at-bot. −poly p x :>
              (if even (degree p) then at-top else at-bot) by simp
  thus ?thesis by (auto simp: filterlim-uminus-at-bot)
qed

lemma poly-lim-neg-inf:
  eventually (λx::real. sgn (poly p x) = poly-neg-inf p) at-bot
proof (cases degree p ≥ 1)
  case False
    hence degree p = 0 by simp
    then obtain c where p = [:c:] by (cases p, auto split: if-split-asm)
    thus ?thesis
        by (simp add: eventually-at-bot-linorder poly-neg-inf-def)
```

```
next
  case True
    note deg = this
    let ?lc = coeff p (degree p)
    from True have ?lc ≠ 0 by force
    show ?thesis
    proof (cases ?lc > 0)
      case True
        note lc-pos = this
        show ?thesis
        proof (cases even (degree p))
          case True
            from poly-at-top-or-bot-at-bot[OF deg lc-pos] and True
              obtain x0 where ⋀x. x ≤ x0 ⟹ poly p x ≥ 1
                by (fastforce simp add: filterlim-at-top filterlim-at-bot
                        eventually-at-bot-linorder less-eq-real-def)
                hence ⋀x. x ≤ x0 ⟹ sgn (poly p x) = 1 by force
              thus ?thesis
                by (simp add: True eventually-at-bot-linorder poly-neg-inf-def,
                    intro exI[of - x0], simp add: lc-pos)
        next
          case False
            from poly-at-top-or-bot-at-bot[OF deg lc-pos] and False
              obtain x0 where ⋀x. x ≤ x0 ⟹ poly p x ≤ −1
                by (fastforce simp add: filterlim-at-bot
                        eventually-at-bot-linorder less-eq-real-def)
                hence ⋀x. x ≤ x0 ⟹ sgn (poly p x) = −1 by force
              thus ?thesis
                by (simp add: False eventually-at-bot-linorder poly-neg-inf-def,
                            intro exI[of - x0], simp add: lc-pos)
      qed
    next
      case False
        hence lc-neg: ?lc < 0 using ‹?lc ≠ 0› by linarith
        show ?thesis
        proof (cases even (degree p))
          case True
            with poly-at-bot-or-top-at-bot[OF deg lc-neg]
              obtain x0 where ⋀x. x ≤ x0 ⟹ poly p x ≤ −1
                  by (fastforce simp: filterlim-at-bot
                          eventually-at-bot-linorder less-eq-real-def)
              hence ⋀x. x ≤ x0 ⟹ sgn (poly p x) = −1 by force
              thus ?thesis
                by (simp only: True eventually-at-bot-linorder poly-neg-inf-def,
                            intro exI[of - x0], simp add: lc-neg)
        next
          case False
            with poly-at-bot-or-top-at-bot[OF deg lc-neg]
              obtain x0 where ⋀x. x ≤ x0 ⟹ poly p x ≥ 1
```

```
          by (fastforce simp: filterlim-at-top
                eventually-at-bot-linorder less-eq-real-def)
        hence ⋀x. x ≤ x₀ ⟹ sgn (poly p x) = 1 by force
        thus ?thesis
          by (simp only: False eventually-at-bot-linorder poly-neg-inf-def,
                intro exI[of - x₀], simp add: lc-neg)
    qed
  qed
qed
```

1.2.5 Signs of polynomials for sufficiently large values

```
lemma polys-inf-sign-thresholds:
  assumes finite (ps :: real poly set)
  obtains l u
  where l ≤ u
    and ⋀p. ⟦p ∈ ps; p ≠ 0⟧ ⟹
          {x. l < x ∧ x ≤ u ∧ poly p x = 0} = {x. poly p x = 0}
    and ⋀p x. ⟦p ∈ ps; x ≥ u⟧ ⟹ sgn (poly p x) = poly-inf p
    and ⋀p x. ⟦p ∈ ps; x ≤ l⟧ ⟹ sgn (poly p x) = poly-neg-inf p
proof goal-cases
  case prems: 1
  have ∃ l u. l ≤ u ∧ (∀ p x. p ∈ ps ∧ x ≥ u ⟶ sgn (poly p x) = poly-inf p) ∧
          (∀ p x. p ∈ ps ∧ x ≤ l ⟶ sgn (poly p x) = poly-neg-inf p)
    (is ∃ l u. ?P ps l u)
  proof (induction rule: finite-subset-induct[OF assms(1), where A = UNIV])
    case 1
      show ?case by simp
  next
    case 2
      show ?case by (intro exI[of - 42], simp)
  next
    case prems: (3 p ps)
      from prems(4) obtain l u where lu-props: ?P ps l u by blast
      from poly-lim-inf obtain u'
          where u'-props: ∀ x≥u'. sgn (poly p x) = poly-inf p
          by (force simp add: eventually-at-top-linorder)
      from poly-lim-neg-inf obtain l'
          where l'-props: ∀ x≤l'. sgn (poly p x) = poly-neg-inf p
          by (force simp add: eventually-at-bot-linorder)
      show ?case
          by (rule exI[of - min l l'], rule exI[of - max u u'],
            insert lu-props l'-props u'-props, auto)
  qed
  then obtain l u where lu-props: l ≤ u
      ⋀p x. p ∈ ps ⟹ u ≤ x ⟹ sgn (poly p x) = poly-inf p
      ⋀p x. p ∈ ps ⟹ x ≤ l ⟹ sgn (poly p x) = poly-neg-inf p by blast
  moreover {
    fix p x assume A: p ∈ ps p ≠ 0 poly p x = 0
```

```
    from A have l < x x < u
      by (auto simp: not-le[symmetric] dest: lu-props(2,3))
  }
  note A = this
  have ⋀p. p ∈ ps ⟹ p ≠ 0 ⟹
           {x. l < x ∧ x ≤ u ∧ poly p x = 0} = {x. poly p x = 0}
    by (auto dest: A)

  from prems[OF lu-props(1) this lu-props(2,3)] show thesis .
qed
```

1.2.6 Positivity of polynomials

```
lemma poly-pos:
  (∀x::real. poly p x > 0) ⟷ poly-inf p = 1 ∧ (∀x. poly p x ≠ 0)
proof (intro iffI conjI)
  assume A: ∀x::real. poly p x > 0
  have ⋀x. poly p (x::real) > 0 ⟹ poly p x ≠ 0 by simp
  with A show ∀x::real. poly p x ≠ 0 by simp

  from poly-lim-inf obtain x where sgn (poly p x) = poly-inf p
    by (auto simp: eventually-at-top-linorder)
  with A show poly-inf p = 1
    by (simp add: sgn-real-def split: if-split-asm)
next
  assume poly-inf p = 1 ∧ (∀x. poly p x ≠ 0)
  hence A: poly-inf p = 1 and B: (∀x. poly p x ≠ 0) by simp-all
  from poly-lim-inf obtain x where C: sgn (poly p x) = poly-inf p
    by (auto simp: eventually-at-top-linorder)
  show ∀x. poly p x > 0
  proof (rule ccontr)
    assume ¬(∀x. poly p x > 0)
    then obtain x' where poly p x' ≤ 0 by (auto simp: not-less)
    with A and C have sgn (poly p x') ≠ sgn (poly p x)
      by (auto simp: sgn-real-def split: if-split-asm)
    from poly-different-sign-imp-root'[OF this] and B
      show False by blast
  qed
qed

lemma poly-pos-greater:
  (∀x::real. x > a ⟶ poly p x > 0) ⟷
    poly-inf p = 1 ∧ (∀x. x > a ⟶ poly p x ≠ 0)
proof (intro iffI conjI)
  assume A: ∀x::real. x > a ⟶ poly p x > 0
  have ⋀x. poly p (x::real) > 0 ⟹ poly p x ≠ 0 by simp
  with A show ∀x::real. x > a ⟶ poly p x ≠ 0 by auto

  from poly-lim-inf obtain x0 where
```

```
      ∀x≥x0. sgn (poly p x) = poly-inf p
      by (auto simp: eventually-at-top-linorder)
  hence poly-inf p = sgn (poly p (max x0 (a + 1))) by simp
  also from A have ... = 1 by force
  finally show poly-inf p = 1 .
next
  assume poly-inf p = 1 ∧ (∀x. x > a ⟶ poly p x ≠ 0)
  hence A: poly-inf p = 1 and
        B: (∀x. x > a ⟶ poly p x ≠ 0) by simp-all
  from poly-lim-inf obtain x0 where
      C: ∀x≥x0. sgn (poly p x) = poly-inf p
      by (auto simp: eventually-at-top-linorder)
  hence sgn (poly p (max x0 (a+1))) = poly-inf p by simp
  with A have D: sgn (poly p (max x0 (a+1))) = 1 by simp
  show ∀x. x > a ⟶ poly p x > 0
  proof (rule ccontr)
    assume ¬(∀x. x > a ⟶ poly p x > 0)
    then obtain x' where x' > a poly p x' ≤ 0 by (auto simp: not-less)
    with A and D have E: sgn (poly p x') ≠ sgn (poly p (max x0(a+1)))
        by (auto simp: sgn-real-def split: if-split-asm)
    show False
        apply (cases x' max x0 (a+1) rule: linorder-cases)
        using B E ‹x' > a›
            apply (force dest!: poly-different-sign-imp-root[of - - p])+
        done
  qed
qed

lemma poly-pos-geq:
  (∀x::real. x ≥ a ⟶ poly p x > 0) ⟷
    poly-inf p = 1 ∧ (∀x. x ≥ a ⟶ poly p x ≠ 0)
proof (intro iffI conjI)
  assume A: ∀x::real. x ≥ a ⟶ poly p x > 0
  hence ∀x::real. x > a ⟶ poly p x > 0 by simp
  also note poly-pos-greater
  finally have poly-inf p = 1 (∀x>a. poly p x ≠ 0) by simp-all
  moreover from A have poly p a > 0 by simp
  ultimately show poly-inf p = 1 ∀x≥a. poly p x ≠ 0
      by (auto simp: less-eq-real-def)
next
  assume poly-inf p = 1 ∧ (∀x. x ≥ a ⟶ poly p x ≠ 0)
  hence A: poly-inf p = 1 and
        B: poly p a ≠ 0 and C: ∀x>a. poly p x ≠ 0 by simp-all
  from A and C and poly-pos-greater have ∀x>a. poly p x > 0 by simp
  moreover with B C poly-IVT-pos[of a a+1 p] have poly p a > 0 by force
  ultimately show ∀x≥a. poly p x > 0 by (auto simp: less-eq-real-def)
qed

lemma poly-pos-less:
```

```
  (∀x::real. x < a ⟶ poly p x > 0) ⟷
    poly-neg-inf p = 1 ∧ (∀x. x < a ⟶ poly p x ≠ 0)
proof (intro iffI conjI)
  assume A: ∀x::real. x < a ⟶ poly p x > 0
  have ⋀x. poly p (x::real) > 0 ⟹ poly p x ≠ 0 by simp
  with A show ∀x::real. x < a ⟶ poly p x ≠ 0 by auto

  from poly-lim-neg-inf obtain x0 where
      ∀x≤x0. sgn (poly p x) = poly-neg-inf p
      by (auto simp: eventually-at-bot-linorder)
  hence poly-neg-inf p = sgn (poly p (min x0 (a − 1))) by simp
  also from A have ... = 1 by force
  finally show poly-neg-inf p = 1 .
next
  assume poly-neg-inf p = 1 ∧ (∀x. x < a ⟶ poly p x ≠ 0)
  hence A: poly-neg-inf p = 1 and
        B: (∀x. x < a ⟶ poly p x ≠ 0) by simp-all
  from poly-lim-neg-inf obtain x0 where
      C: ∀x≤x0. sgn (poly p x) = poly-neg-inf p
      by (auto simp: eventually-at-bot-linorder)
  hence sgn (poly p (min x0 (a − 1))) = poly-neg-inf p by simp
  with A have D: sgn (poly p (min x0 (a − 1))) = 1 by simp
  show ∀x. x < a ⟶ poly p x > 0
  proof (rule ccontr)
    assume ¬(∀x. x < a ⟶ poly p x > 0)
    then obtain x' where x' < a poly p x' ≤ 0 by (auto simp: not-less)
    with A and D have E: sgn (poly p x') ≠ sgn (poly p (min x0 (a − 1)))
        by (auto simp: sgn-real-def split: if-split-asm)
    show False
        apply (cases x' min x0 (a − 1) rule: linorder-cases)
        using B E ‹x' < a›
            apply (auto dest!: poly-different-sign-imp-root[of - - p])+
        done
  qed
qed

lemma poly-pos-leq:
  (∀x::real. x ≤ a ⟶ poly p x > 0) ⟷
    poly-neg-inf p = 1 ∧ (∀x. x ≤ a ⟶ poly p x ≠ 0)
proof (intro iffI conjI)
  assume A: ∀x::real. x ≤ a ⟶ poly p x > 0
  hence ∀x::real. x < a ⟶ poly p x > 0 by simp
  also note poly-pos-less
  finally have poly-neg-inf p = 1 (∀x<a. poly p x ≠ 0) by simp-all
  moreover from A have poly p a > 0 by simp
  ultimately show poly-neg-inf p = 1 ∀x≤a. poly p x ≠ 0
      by (auto simp: less-eq-real-def)
next
  assume poly-neg-inf p = 1 ∧ (∀x. x ≤ a ⟶ poly p x ≠ 0)
```

```
  hence A: poly-neg-inf p = 1 and
      B: poly p a ≠ 0 and C: ∀x<a. poly p x ≠ 0 by simp-all
  from A and C and poly-pos-less have ∀x<a. poly p x > 0 by simp
  moreover with B C poly-IVT-neg[of a − 1 a p] have poly p a > 0 by force
  ultimately show ∀x≤a. poly p x > 0 by (auto simp: less-eq-real-def)
qed

lemma poly-pos-between-less-less:
  (∀x::real. a < x ∧ x < b ⟶ poly p x > 0) ⟷
    (a ≥ b ∨ poly p ((a+b)/2) > 0) ∧ (∀x. a < x ∧ x < b ⟶ poly p x ≠ 0)
proof (intro iffI conjI)
  assume A: ∀x. a < x ∧ x < b ⟶ poly p x > 0
  have ⋀x. poly p (x::real) > 0 ⟹ poly p x ≠ 0 by simp
  with A show ∀x::real. a < x ∧ x < b ⟶ poly p x ≠ 0 by auto
  from A show a ≥ b ∨ poly p ((a+b)/2) > 0 by (cases a < b, auto)
next
  assume (b ≤ a ∨ 0 < poly p ((a+b)/2)) ∧ (∀x. a<x ∧ x<b ⟶ poly p x ≠ 0)
  hence A: b ≤ a ∨ 0 < poly p ((a+b)/2) and
      B: ∀x. a<x ∧ x<b ⟶ poly p x ≠ 0 by simp-all
  show ∀x. a < x ∧ x < b ⟶ poly p x > 0
  proof (cases a ≥ b, simp, clarify, rule-tac ccontr,
        simp only: not-le not-less)
    fix x assume a < b a < x x < b poly p x ≤ 0
    with B have poly p x < 0 by (simp add: less-eq-real-def)
    moreover from A and ‹a < b› have poly p ((a+b)/2) > 0 by simp
    ultimately have sgn (poly p x) ≠ sgn (poly p ((a+b)/2)) by simp
    thus False using B
      apply (cases x (a+b)/2 rule: linorder-cases)
      apply (drule poly-different-sign-imp-root[of - - p], assumption,
            insert ‹a < b› ‹a < x› ‹x < b› , force) []
      apply simp
      apply (drule poly-different-sign-imp-root[of - - p], simp,
            insert ‹a < b› ‹a < x› ‹x < b› , force)
      done
  qed
qed

lemma poly-pos-between-less-leq:
  (∀x::real. a < x ∧ x ≤ b ⟶ poly p x > 0) ⟷
    (a ≥ b ∨ poly p b > 0) ∧ (∀x. a < x ∧ x ≤ b ⟶ poly p x ≠ 0)
proof (intro iffI conjI)
  assume A: ∀x. a < x ∧ x ≤ b ⟶ poly p x > 0
  have ⋀x. poly p (x::real) > 0 ⟹ poly p x ≠ 0 by simp
  with A show ∀x::real. a < x ∧ x ≤ b ⟶ poly p x ≠ 0 by auto
  from A show a ≥ b ∨ poly p b > 0 by (cases a < b, auto)
next
  assume (b ≤ a ∨ 0 < poly p b) ∧ (∀x. a<x ∧ x≤b ⟶ poly p x ≠ 0)
  hence A: b ≤ a ∨ 0 < poly p b and B: ∀x. a<x ∧ x≤b ⟶ poly p x ≠ 0
    by simp-all
```

```
  show ∀x. a < x ∧ x ≤ b ⟶ poly p x > 0
  proof (cases a ≥ b, simp, clarify, rule-tac ccontr,
       simp only: not-le not-less)
    fix x assume a < b a < x x ≤ b poly p x ≤ 0
    with B have poly p x < 0 by (simp add: less-eq-real-def)
    moreover from A and ‹a < b› have poly p b > 0 by simp
    ultimately have x < b using ‹x ≤ b› by (auto simp: less-eq-real-def)
    from ‹poly p x < 0› and ‹poly p b > 0›
      have sgn (poly p x) ≠ sgn (poly p b) by simp
    from poly-different-sign-imp-root[OF ‹x < b› this] and B and ‹x > a›
      show False by auto
  qed
qed

lemma poly-pos-between-leq-less:
  (∀x::real. a ≤ x ∧ x < b ⟶ poly p x > 0) ⟷
    (a ≥ b ∨ poly p a > 0) ∧ (∀x. a ≤ x ∧ x < b ⟶ poly p x ≠ 0)
proof (intro iffI conjI)
  assume A: ∀x. a ≤ x ∧ x < b ⟶ poly p x > 0
  have ⋀x. poly p (x::real) > 0 ⟹ poly p x ≠ 0 by simp
  with A show ∀x::real. a ≤ x ∧ x < b ⟶ poly p x ≠ 0 by auto
  from A show a ≥ b ∨ poly p a > 0 by (cases a < b, auto)
next
  assume (b ≤ a ∨ 0 < poly p a) ∧ (∀x. a≤x ∧ x<b ⟶ poly p x ≠ 0)
  hence A: b ≤ a ∨ 0 < poly p a and B: ∀x. a≤x ∧ x<b ⟶ poly p x ≠ 0
    by simp-all
  show ∀x. a ≤ x ∧ x < b ⟶ poly p x > 0
  proof (cases a ≥ b, simp, clarify, rule-tac ccontr,
       simp only: not-le not-less)
    fix x assume a < b a ≤ x x < b poly p x ≤ 0
    with B have poly p x < 0 by (simp add: less-eq-real-def)
    moreover from A and ‹a < b› have poly p a > 0 by simp
    ultimately have x > a using ‹x ≥ a› by (auto simp: less-eq-real-def)
    from ‹poly p x < 0› and ‹poly p a > 0›
      have sgn (poly p a) ≠ sgn (poly p x) by simp
    from poly-different-sign-imp-root[OF ‹x > a› this] and B and ‹x < b›
      show False by auto
  qed
qed

lemma poly-pos-between-leq-leq:
  (∀x::real. a ≤ x ∧ x ≤ b ⟶ poly p x > 0) ⟷
    (a > b ∨ poly p a > 0) ∧ (∀x. a ≤ x ∧ x ≤ b ⟶ poly p x ≠ 0)
proof (intro iffI conjI)
  assume A: ∀x. a ≤ x ∧ x ≤ b ⟶ poly p x > 0
  have ⋀x. poly p (x::real) > 0 ⟹ poly p x ≠ 0 by simp
  with A show ∀x::real. a ≤ x ∧ x ≤ b ⟶ poly p x ≠ 0 by auto
  from A show a > b ∨ poly p a > 0 by (cases a ≤ b, auto)
next
```

assume $(b < a \lor 0 < poly\ p\ a) \land (\forall x.\ a \le x \land x \le b \longrightarrow poly\ p\ x \ne 0)$
hence *A*: $b < a \lor 0 < poly\ p\ a$ **and** *B*: $\forall x.\ a \le x \land x \le b \longrightarrow poly\ p\ x \ne 0$
by *simp-all*
show $\forall x.\ a \le x \land x \le b \longrightarrow poly\ p\ x > 0$
proof (*cases* $a > b$, *simp*, *clarify*, *rule-tac ccontr*,
simp only: *not-le not-less*)
fix *x* **assume** $a \le b$ $a \le x$ $x \le b$ $poly\ p\ x \le 0$
with *B* **have** $poly\ p\ x < 0$ **by** (*simp add*: *less-eq-real-def*)
moreover from *A* **and** ‹$a \le b$› **have** $poly\ p\ a > 0$ **by** *simp*
ultimately have $x > a$ **using** ‹$x \ge a$› **by** (*auto simp*: *less-eq-real-def*)
from ‹$poly\ p\ x < 0$› **and** ‹$poly\ p\ a > 0$›
have $sgn\ (poly\ p\ a) \ne sgn\ (poly\ p\ x)$ **by** *simp*
from *poly-different-sign-imp-root*[*OF* ‹$x > a$› *this*] **and** *B* **and** ‹$x \le b$›
show *False* **by** *auto*
qed
qed

end

2 Proof of Sturm's Theorem

theory *Sturm-Theorem*
imports *HOL−Computational-Algebra.Polynomial*
Lib/Sturm-Library HOL−Computational-Algebra.Field-as-Ring
begin

2.1 Sign changes of polynomial sequences

For a given sequence of polynomials, this function computes the number of sign changes of the sequence of polynomials evaluated at a given position x. A sign change is a change from a negative value to a positive one or vice versa; zeros in the sequence are ignored.

definition *sign-changes* **where**
sign-changes ps (*x*::*real*) =
length (*remdups-adj* (*filter* ($\lambda x.\ x \ne 0$) (*map* ($\lambda p.\ sgn\ (poly\ p\ x)$) *ps*))) − *1*

The number of sign changes of a sequence distributes over a list in the sense that the number of sign changes of a sequence $p_1, \ldots, p_i, \ldots, p_n$ at x is the same as the sum of the sign changes of the sequence $p_1, \ldots, p_i$ and $p_i, \ldots, p_n$ as long as $p_i(x) \ne 0$.

lemma *sign-changes-distrib*:
$poly\ p\ x \ne 0 \Longrightarrow$
sign-changes (ps_1 @ [*p*] @ ps_2) *x* =
sign-changes (ps_1 @ [*p*]) *x* + *sign-changes* ([*p*] @ ps_2) *x*
by (*simp add*: *sign-changes-def sgn-zero-iff*, *subst remdups-adj-append*, *simp*)

The following two congruences state that the number of sign changes is the same if all the involved signs are the same.

lemma *sign-changes-cong*:
  assumes *length ps* = *length ps'*
  assumes ∀ *i* < *length ps*. *sgn* (*poly* (*ps*!*i*) *x*) = *sgn* (*poly* (*ps'*!*i*) *y*)
  shows *sign-changes ps x* = *sign-changes ps' y*
proof−
 from *assms*(*2*) **have** *A*: *map* (λ*p*. *sgn* (*poly p x*)) *ps* = *map* (λ*p*. *sgn* (*poly p y*)) *ps'*
  proof (*induction rule*: *list-induct2*[*OF assms*(*1*)])
    case *1*
      then show *?case* **by** *simp*
  next
    case (*2 p ps p' ps'*)
      from *2*(*3*)
      have ∀ *i*<*length ps*. *sgn* (*poly* (*ps* ! *i*) *x*) =
                   sgn (*poly* (*ps'* ! *i*) *y*) **by** *auto*
      from *2*(*2*)[*OF this*] *2*(*3*) **show** *?case* **by** *auto*
  qed
  show *?thesis* **unfolding** *sign-changes-def* **by** (*simp add*: *A*)
qed

lemma *sign-changes-cong'*:
  assumes ∀ *p* ∈ *set ps*. *sgn* (*poly p x*) = *sgn* (*poly p y*)
  shows *sign-changes ps x* = *sign-changes ps y*
using *assms* **by** (*intro sign-changes-cong*, *simp-all*)

For a sequence of polynomials of length 3, if the first and the third polynomial have opposite and nonzero sign at some x, the number of sign changes is always 1, irrespective of the sign of the second polynomial.

lemma *sign-changes-sturm-triple*:
  assumes *poly p x* ≠ *0* **and** *sgn* (*poly r x*) = − *sgn* (*poly p x*)
  shows *sign-changes* [*p*,*q*,*r*] *x* = *1*
unfolding *sign-changes-def* **by** (*insert assms*, *auto simp*: *sgn-real-def*)

Finally, we define two additional functions that count the sign changes "at infinity".

definition *sign-changes-inf* **where**
sign-changes-inf ps =
    length (*remdups-adj* (*filter* (λ*x*. *x* ≠ *0*) (*map poly-inf ps*))) − *1*

definition *sign-changes-neg-inf* **where**
sign-changes-neg-inf ps =
    length (*remdups-adj* (*filter* (λ*x*. *x* ≠ *0*) (*map poly-neg-inf ps*))) − *1*

2.2 Definition of Sturm sequences locale

We first define the notion of a "Quasi-Sturm sequence", which is a weakening of a Sturm sequence that captures the properties that are fulfilled by a nonempty suffix of a Sturm sequence:

- The sequence is nonempty.
- The last polynomial does not change its sign.
- If the middle one of three adjacent polynomials has a root at x, the other two have opposite and nonzero signs at x.

locale *quasi-sturm-seq* =
  fixes *ps* :: (*real poly*) *list*
  assumes *last-ps-sgn-const*[*simp*]:
    $\bigwedge x\ y.\ sgn\ (poly\ (last\ ps)\ x) = sgn\ (poly\ (last\ ps)\ y)$
  assumes *ps-not-Nil*[*simp*]: $ps \neq []$
  assumes *signs*: $\bigwedge i\ x.\ [\![i < length\ ps - 2;\ poly\ (ps\ !\ (i{+}1))\ x = 0]\!]$
    $\Longrightarrow (poly\ (ps\ !\ (i{+}2))\ x) * (poly\ (ps\ !\ i)\ x) < 0$

Now we define a Sturm sequence $p_1, \ldots, p_n$ of a polynomial p in the following way:

- The sequence contains at least two elements.
- p is the first polynomial, i. e. $p_1 = p$.
- At any root x of p, p_2 and p have opposite sign left of x and the same sign right of x in some neighbourhood around x.
- The first two polynomials in the sequence have no common roots.
- If the middle one of three adjacent polynomials has a root at x, the other two have opposite and nonzero signs at x.

locale *sturm-seq* = *quasi-sturm-seq* +
  fixes *p* :: *real poly*
  assumes *hd-ps-p*[*simp*]: *hd ps* = *p*
  assumes *length-ps-ge-2*[*simp*]: *length ps* $\geq$ *2*
  assumes *deriv*: $\bigwedge x_0.\ poly\ p\ x_0 = 0 \Longrightarrow$
    $eventually\ (\lambda x.\ sgn\ (poly\ (p * ps!1)\ x) =$
      $(if\ x > x_0\ then\ 1\ else\ -1))\ (at\ x_0)$
  assumes *p-squarefree*: $\bigwedge x.\ \neg(poly\ p\ x = 0 \wedge poly\ (ps!1)\ x = 0)$
begin

Any Sturm sequence is obviously a Quasi-Sturm sequence.

  lemma *quasi-sturm-seq*: *quasi-sturm-seq ps* **..**
end

Any suffix of a Quasi-Sturm sequence is again a Quasi-Sturm sequence.

lemma *quasi-sturm-seq-Cons*:
  assumes *quasi-sturm-seq* (*p*#*ps*) **and** $ps \neq []$
  shows *quasi-sturm-seq ps*

proof (*unfold-locales*)
  show *ps* ≠ [] **by** *fact*
next
  from *assms*(*1*) **interpret** *quasi-sturm-seq p#ps* **.**
  fix *x y*
  from *last-ps-sgn-const* **and** ‹*ps* ≠ []›
    show *sgn* (*poly* (*last ps*) *x*) = *sgn* (*poly* (*last ps*) *y*) **by** *simp-all*
next
  from *assms*(*1*) **interpret** *quasi-sturm-seq p#ps* **.**
  fix *i x*
  assume *i* < *length ps* − *2* **and** *poly* (*ps* ! (*i*+*1*)) *x* = *0*
  with *signs*[*of i*+*1*]
    show *poly* (*ps* ! (*i*+*2*)) *x* ∗ *poly* (*ps* ! *i*) *x* < *0* **by** *simp*
qed

2.3 Auxiliary lemmas about roots and sign changes

lemma *sturm-adjacent-root-aux*:
  assumes *i* < *length* (*ps* :: *real poly list*) − *1*
  assumes *poly* (*ps* ! *i*) *x* = *0* **and** *poly* (*ps* ! (*i* + *1*)) *x* = *0*
  assumes ⋀*i x*. ⟦*i* < *length ps* − *2*; *poly* (*ps* ! (*i*+*1*)) *x* = *0*⟧
        ⟹ *sgn* (*poly* (*ps* ! (*i*+*2*)) *x*) = − *sgn* (*poly* (*ps* ! *i*) *x*)
  shows ∀ *j*≤*i*+*1*. *poly* (*ps* ! *j*) *x* = *0*
using *assms*
proof (*induction i*)
  case *0* **thus** *?case* **by** (*clarsimp*, *rename-tac j*, *case-tac j*, *simp-all*)
next
  case (*Suc i*)
    from *Suc.prems*(*1*,*2*)
      have *sgn* (*poly* (*ps* ! (*i* + *2*)) *x*) = − *sgn* (*poly* (*ps* ! *i*) *x*)
      by (*intro assms*(*4*)) *simp-all*
    with *Suc.prems*(*3*) **have** *poly* (*ps* ! *i*) *x* = *0* **by** (*simp add*: *sgn-zero-iff*)
    with *Suc.prems* **have** ∀ *j*≤*i*+*1*. *poly* (*ps* ! *j*) *x* = *0*
      by (*intro Suc.IH*, *simp-all*)
    with *Suc.prems*(*3*) **show** *?case*
      by (*clarsimp*, *rename-tac j*, *case-tac j* = *Suc* (*Suc i*), *simp-all*)
qed

This function splits the sign list of a Sturm sequence at a position x that is not a root of p into a list of sublists such that the number of sign changes within every sublist is constant in the neighbourhood of x, thus proving that the total number is also constant.

fun *split-sign-changes* **where**
split-sign-changes [*p*] (*x* :: *real*) = [[*p*]] |
split-sign-changes [*p*,*q*] *x* = [[*p*,*q*]] |
split-sign-changes (*p#q#r#ps*) *x* =
  (*if poly p x* ≠ *0* ∧ *poly q x* = *0 then*
    [*p*,*q*,*r*] # *split-sign-changes* (*r#ps*) *x*
   else

[p,q] # *split-sign-changes* (*q*#*r*#*ps*) *x*)

lemma (**in** *quasi-sturm-seq*) *split-sign-changes-subset*[*dest*]:
ps' ∈ *set* (*split-sign-changes ps x*) ⟹ *set ps'* ⊆ *set ps*
apply (*insert ps-not-Nil*)
apply (*induction ps x rule*: *split-sign-changes.induct*)
apply (*simp*, *simp*, *rename-tac p q r ps x*,
case-tac poly p x ≠ *0* ∧ *poly q x* = *0*, *auto*)
done

A custom induction rule for *split-sign-changes* that uses the fact that all the intermediate parameters in calls of *split-sign-changes* are quasi-Sturm sequences.

lemma (**in** *quasi-sturm-seq*) *split-sign-changes-induct*:
⟦⋀*p x*. *P* [*p*] *x*; ⋀*p q x*. *quasi-sturm-seq* [*p*,*q*] ⟹ *P* [*p*,*q*] *x*;
⋀*p q r ps x*. *quasi-sturm-seq* (*p*#*q*#*r*#*ps*) ⟹
⟦*poly p x* ≠ *0* ⟹ *poly q x* = *0* ⟹ *P* (*r*#*ps*) *x*;
poly q x ≠ *0* ⟹ *P* (*q*#*r*#*ps*) *x*;
poly p x = *0* ⟹ *P* (*q*#*r*#*ps*) *x*⟧
⟹ *P* (*p*#*q*#*r*#*ps*) *x*⟧ ⟹ *P ps x*
proof *goal-cases*
case *prems*: *1*
have *quasi-sturm-seq ps* **..**
with *prems* **show** *?thesis*
proof (*induction ps x rule*: *split-sign-changes.induct*)
case (*3 p q r ps x*)
show *?case*
proof (*rule 3*(*5*)[*OF 3*(*6*)])
assume *A*: *poly p x* ≠ *0 poly q x* = *0*
from *3*(*6*) **have** *quasi-sturm-seq* (*r*#*ps*)
by (*force dest*: *quasi-sturm-seq-Cons*)
with *3 A* **show** *P* (*r* # *ps*) *x* **by** *blast*
next
assume *A*: *poly q x* ≠ *0*
from *3*(*6*) **have** *quasi-sturm-seq* (*q*#*r*#*ps*)
by (*force dest*: *quasi-sturm-seq-Cons*)
with *3 A* **show** *P* (*q* # *r* # *ps*) *x* **by** *blast*
next
assume *A*: *poly p x* = *0*
from *3*(*6*) **have** *quasi-sturm-seq* (*q*#*r*#*ps*)
by (*force dest*: *quasi-sturm-seq-Cons*)
with *3 A* **show** *P* (*q* # *r* # *ps*) *x* **by** *blast*
qed
qed *simp-all*
qed

The total number of sign changes in the split list is the same as the number of sign changes in the original list.

lemma (**in** *quasi-sturm-seq*) *split-sign-changes-correct*:

```
  assumes poly (hd ps) x₀ ≠ 0
  defines sign-changes' ≡ λps x.
              ∑ps'←split-sign-changes ps x. sign-changes ps' x
  shows sign-changes' ps x₀ = sign-changes ps x₀
using assms(1)
proof (induction x₀ rule: split-sign-changes-induct)
case (3 p q r ps x₀)
  hence poly p x₀ ≠ 0 by simp
  note IH = 3(2,3,4)
  show ?case
  proof (cases poly q x₀ = 0)
    case True
      from 3 interpret quasi-sturm-seq p#q#r#ps by simp
      from signs[of 0] and True have
           sgn-r-x0: poly r x₀ ∗ poly p x₀ < 0 by simp
      with 3 have poly r x₀ ≠ 0 by force
      from sign-changes-distrib[OF this, of [p,q] ps]
        have sign-changes (p#q#r#ps) x₀ =
                 sign-changes ([p, q, r]) x₀ + sign-changes (r # ps) x₀ by simp
      also have sign-changes (r#ps) x₀ = sign-changes' (r#ps) x₀
          using ‹poly q x₀ = 0› ‹poly p x₀ ≠ 0› 3(5)‹poly r x₀ ≠ 0›
          by (intro IH(1)[symmetric], simp-all)
      finally show ?thesis unfolding sign-changes'-def
          using True ‹poly p x₀ ≠ 0› by simp
  next
    case False
      from sign-changes-distrib[OF this, of [p] r#ps]
          have sign-changes (p#q#r#ps) x₀ =
                 sign-changes ([p,q]) x₀ + sign-changes (q#r#ps) x₀ by simp
      also have sign-changes (q#r#ps) x₀ = sign-changes' (q#r#ps) x₀
          using ‹poly q x₀ ≠ 0› ‹poly p x₀ ≠ 0› 3(5)
          by (intro IH(2)[symmetric], simp-all)
      finally show ?thesis unfolding sign-changes'-def
          using False by simp
    qed
qed (simp-all add: sign-changes-def sign-changes'-def)
```

We now prove that if $p(x) \neq 0$, the number of sign changes of a Sturm sequence of p at x is constant in a neighbourhood of x.

```
lemma (in quasi-sturm-seq) split-sign-changes-correct-nbh:
  assumes poly (hd ps) x₀ ≠ 0
  defines sign-changes' ≡ λx₀ ps x.
              ∑ps'←split-sign-changes ps x₀. sign-changes ps' x
  shows eventually (λx. sign-changes' x₀ ps x = sign-changes ps x) (at x₀)
proof (rule eventually-mono)
  show eventually (λx. ∀p∈{p ∈ set ps. poly p x₀ ≠ 0}. sgn (poly p x) = sgn (poly
p x₀)) (at x₀)
      by (rule eventually-ball-finite, auto intro: poly-neighbourhood-same-sign)
next
```

fix *x*
show $(\forall p{\in}\{p \in set\ ps.\ poly\ p\ x_0 \neq 0\}.\ sgn\ (poly\ p\ x) = sgn\ (poly\ p\ x_0)) \Longrightarrow$
sign-changes′ x_0 *ps x* = *sign-changes ps x*
proof −
fix *x* **assume** *nbh*: $\forall p{\in}\{p \in set\ ps.\ poly\ p\ x_0 \neq 0\}.\ sgn\ (poly\ p\ x) = sgn\ (poly\ p\ x_0)$
thus *sign-changes′* x_0 *ps x* = *sign-changes ps x* **using** *assms*(*1*)
proof (*induction* x_0 *rule*: *split-sign-changes-induct*)
case (*3 p q r ps* x_0)
hence *poly p* $x_0 \neq 0$ **by** *simp*
note *IH* = *3*(*2*,*3*,*4*)
show *?case*
proof (*cases poly q* $x_0 = 0$)
case *True*
from *3* **interpret** *quasi-sturm-seq p#q#r#ps* **by** *simp*
from *signs*[*of 0*] **and** *True* **have**
sgn-r-x0: *poly r* x_0 ∗ *poly p* $x_0 < 0$ **by** *simp*
with *3* **have** *poly r* $x_0 \neq 0$ **by** *force*
with *nbh 3*(*5*) **have** *poly r x* ≠ *0* **by** (*auto simp*: *sgn-zero-iff*)
from *sign-changes-distrib*[*OF this*, *of* [*p*,*q*] *ps*]
have *sign-changes* (*p#q#r#ps*) *x* =
sign-changes ([*p*, *q*, *r*]) *x* + *sign-changes* (*r # ps*) *x* **by** *simp*
also have *sign-changes* (*r#ps*) *x* = *sign-changes′* x_0 (*r#ps*) *x*
using ‹*poly q* $x_0 = 0$› *nbh* ‹*poly p* $x_0 \neq 0$› *3*(*5*)‹*poly r* $x_0 \neq 0$›
by (*intro IH*(*1*)[*symmetric*], *simp-all*)
finally show *?thesis* **unfolding** *sign-changes′-def*
using *True* ‹*poly p* $x_0 \neq 0$›**by** *simp*
next
case *False*
with *nbh 3*(*5*) **have** *poly q x* ≠ *0* **by** (*auto simp*: *sgn-zero-iff*)
from *sign-changes-distrib*[*OF this*, *of* [*p*] *r#ps*]
have *sign-changes* (*p#q#r#ps*) *x* =
sign-changes ([*p*,*q*]) *x* + *sign-changes* (*q#r#ps*) *x* **by** *simp*
also have *sign-changes* (*q#r#ps*) *x* = *sign-changes′* x_0 (*q#r#ps*) *x*
using ‹*poly q* $x_0 \neq 0$› *nbh* ‹*poly p* $x_0 \neq 0$› *3*(*5*)
by (*intro IH*(*2*)[*symmetric*], *simp-all*)
finally show *?thesis* **unfolding** *sign-changes′-def*
using *False* **by** *simp*
qed
qed (*simp-all add*: *sign-changes-def sign-changes′-def*)
qed
qed

lemma (**in** *quasi-sturm-seq*) *hd-nonzero-imp-sign-changes-const-aux*:
assumes *poly* (*hd ps*) $x_0 \neq 0$ **and** *ps′* ∈ *set* (*split-sign-changes ps* x_0)
shows *eventually* (λx. *sign-changes ps′ x* = *sign-changes ps′* x_0) (*at* x_0)
using *assms*

proof (*induction* x_0 *rule*: *split-sign-changes-induct*)
  case (*1 p x*)
    thus *?case* **by** (*simp add*: *sign-changes-def*)
next
  case (*2 p q* x_0)
    hence [*simp*]: $ps' = [p,q]$ **by** *simp*
    from *2* **have** *poly p* $x_0 \neq 0$ **by** *simp*
    from *2*(*1*) **interpret** *quasi-sturm-seq* [*p*,*q*] **.**
    from *poly-neighbourhood-same-sign*[*OF* ‹*poly p* $x_0 \neq 0$›]
        have *eventually* ($\lambda x.$ *sgn* (*poly p x*) = *sgn* (*poly p* x_0)) (*at* x_0) **.**
    moreover from *last-ps-sgn-const*
        have *sgn-q*: $\bigwedge x.$ *sgn* (*poly q x*) = *sgn* (*poly q* x_0) **by** *simp*
    ultimately have *A*: *eventually* ($\lambda x.$ $\forall$ $p \in set[p,q]$. *sgn* (*poly p x*) =
                        sgn (*poly p* x_0)) (*at* x_0) **by** *simp*
    thus *?case* **by** (*force intro*: *eventually-mono*[*OF A*]
                        sign-changes-cong')
next
  case (*3 p q r ps''* x_0)
    hence *p-not-0*: *poly p* $x_0 \neq 0$ **by** *simp*
    note *sturm* = *3*(*1*)
    note *IH* = *3*(*2*,*3*)
    note *ps''-props* = *3*(*6*)
    show *?case*
    proof (*cases poly q* $x_0 = 0$)
      case *True*
        note *q-0* = *this*
        from *sturm* **interpret** *quasi-sturm-seq p#q#r#ps''* **.**
        from *signs*[*of 0*] **and** *q-0*
            have *signs'*: *poly r* x_0 $*$ *poly p* $x_0 < 0$ **by** *simp*
        with *p-not-0* **have** *r-not-0*: *poly r* $x_0 \neq 0$ **by** *force*
        show *?thesis*
        proof (*cases ps'* $\in$ *set* (*split-sign-changes* (*r* # *ps''*) x_0))
          case *True*
            show *?thesis* **by** (*rule IH*(*1*), *fact*, *fact*, *simp add*: *r-not-0*, *fact*)
        next
          case *False*
            with *ps''-props p-not-0 q-0* **have** *ps'-props*: $ps' = [p,q,r]$ **by** *simp*
            from *signs*[*of 0*] **and** *q-0*
                have *sgn-r*: *poly r* x_0 $*$ *poly p* $x_0 < 0$ **by** *simp*
            from *p-not-0 sgn-r*
              have *A*: *eventually* ($\lambda x.$ *sgn* (*poly p x*) = *sgn* (*poly p* x_0) $\wedge$
                              sgn (*poly r x*) = *sgn* (*poly r* x_0)) (*at* x_0)
                by (*intro eventually-conj poly-neighbourhood-same-sign*,
                    simp-all add: *r-not-0*)
            show *?thesis*
            proof (*rule eventually-mono*[*OF A*], *clarify*,
                  subst ps'-props, *subst sign-changes-sturm-triple*)
              fix *x* **assume** *A*: *sgn* (*poly p x*) = *sgn* (*poly p* x_0)
                    and *B*: *sgn* (*poly r x*) = *sgn* (*poly r* x_0)

```
          have prod-neg: ⋀a (b::real). ⟦a>0; b>0; a*b<0⟧ ⟹ False
                         ⋀a (b::real). ⟦a<0; b<0; a*b<0⟧ ⟹ False
            by (drule mult-pos-pos, simp, simp,
                drule mult-neg-neg, simp, simp)
          from A and ‹poly p x0 ≠ 0› show poly p x ≠ 0
            by (force simp: sgn-zero-iff)

          with sgn-r p-not-0 r-not-0 A B
            have poly r x * poly p x < 0 poly r x ≠ 0
            by (metis sgn-less sgn-mult, metis sgn-0-0)
          with sgn-r show sgn-r': sgn (poly r x) = - sgn (poly p x)
            apply (simp add: sgn-real-def not-le not-less
                       split: if-split-asm, intro conjI impI)
            using prod-neg[of poly r x poly p x] apply force+
            done

          show 1 = sign-changes ps' x0
            by (subst ps'-props, subst sign-changes-sturm-triple,
                fact, metis A B sgn-r', simp)
        qed
      qed
  next
    case False
      note q-not-0 = this
      show ?thesis
      proof (cases ps' ∈ set (split-sign-changes (q # r # ps'') x0))
        case True
          show ?thesis by (rule IH(2), fact, simp add: q-not-0, fact)
      next
        case False
          with ps''-props and q-not-0 have ps' = [p, q] by simp
          hence [simp]: ∀p∈set ps'. poly p x0 ≠ 0
              using q-not-0 p-not-0 by simp
          show ?thesis
          proof (rule eventually-mono)
            fix x assume ∀p∈set ps'. sgn (poly p x) = sgn (poly p x0)
            thus sign-changes ps' x = sign-changes ps' x0
                by (rule sign-changes-cong')
          next
            show eventually (λx. ∀p∈set ps'.
                    sgn (poly p x) = sgn (poly p x0)) (at x0)
                by (force intro: eventually-ball-finite
                                poly-neighbourhood-same-sign)
          qed
    qed
  qed
qed
```

lemma (**in** *quasi-sturm-seq*) *hd-nonzero-imp-sign-changes-const*:
assumes *poly* (*hd ps*) $x_0 \neq 0$
shows *eventually* (λx. *sign-changes ps x* = *sign-changes ps* x_0) (*at* x_0)
proof−
let *?pss* = *split-sign-changes ps* x_0
let *?f* = λ*pss x*. $\sum$ *ps'*←*pss*. *sign-changes ps' x*
{
fix *pss* **assume** $\bigwedge$*ps'*. *ps'*∈*set pss* ⟹
eventually (λx. *sign-changes ps' x* = *sign-changes ps'* x_0) (*at* x_0)
hence *eventually* (λx. *?f pss x* = *?f pss* x_0) (*at* x_0)
proof (*induction pss*)
case (*Cons ps' pss*)
then show *?case*
apply (*rule eventually-mono*[*OF eventually-conj*])
apply (*auto simp add*: *Cons.prems*)
done
qed *simp*
}
note *A* = *this*[*of ?pss*]
have *B*: *eventually* (λx. *?f ?pss x* = *?f ?pss* x_0) (*at* x_0)
by (*rule A*, *rule hd-nonzero-imp-sign-changes-const-aux*[*OF assms*], *simp*)
note *C* = *split-sign-changes-correct-nbh*[*OF assms*]
note *D* = *split-sign-changes-correct*[*OF assms*]
note *E* = *eventually-conj*[*OF B C*]
show *?thesis* **by** (*rule eventually-mono*[*OF E*], *auto simp*: *D*)
qed

lemma (**in** *sturm-seq*) *p-nonzero-imp-sign-changes-const*:
poly p $x_0 \neq 0$ ⟹
eventually (λx. *sign-changes ps x* = *sign-changes ps* x_0) (*at* x_0)
using *hd-nonzero-imp-sign-changes-const* **by** *simp*

If x is a root of p and p is not the zero polynomial, the number of sign changes of a Sturm chain of p decreases by 1 at x.

lemma (**in** *sturm-seq*) *p-zero*:
assumes *poly p* $x_0 = 0$ $p \neq 0$
shows *eventually* (λx. *sign-changes ps x* =
sign-changes ps x_0 + (*if* $x<x_0$ *then 1 else 0*)) (*at* x_0)
proof−
from *ps-first-two* **obtain** *q ps'* **where** [*simp*]: *ps* = *p#q#ps'* **.**
hence *ps*!*1* = *q* **by** *simp*
have *eventually* (λx. $x \neq x_0$) (*at* x_0)
by (*simp add*: *eventually-at*, *rule exI*[*of - 1*], *simp*)
moreover from *p-squarefree* **and** *assms*(*1*) **have** *poly q* $x_0 \neq 0$ **by** *simp*
{
have *A*: *quasi-sturm-seq ps* **..**
with *quasi-sturm-seq-Cons*[*of p q#ps'*]
interpret *quasi-sturm-seq q#ps'* **by** *simp*
from ‹*poly q* $x_0 \neq 0$› **have** *eventually* (λx. *sign-changes* (*q#ps'*) *x* =

$sign\text{-}changes\ (q\#ps')\ x_0)\ (at\ x_0)$
using *hd-nonzero-imp-sign-changes-const*[**where** $x_0=x_0$] **by** *simp*
}
moreover note *poly-neighbourhood-without-roots*[*OF assms*(*2*)] *deriv*[*OF assms*(*1*)]
ultimately
have *A*: $eventually\ (\lambda x.\ x \neq x_0 \wedge poly\ p\ x \neq 0 \wedge$
$sgn\ (poly\ (p{*}ps!1)\ x) = (if\ x > x_0\ then\ 1\ else\ -1) \wedge$
$sign\text{-}changes\ (q\#ps')\ x = sign\text{-}changes\ (q\#ps')\ x_0)\ (at\ x_0)$
by (*simp only*: ‹$ps!1 = q$›, *intro eventually-conj*)
show *?thesis*
proof (*rule eventually-mono*[*OF A*], *clarify*, *goal-cases*)
case *prems*: (*1 x*)
from *zero-less-mult-pos* **have** *zero-less-mult-pos'*:
$\bigwedge a\ b.\ [\![(0{::}real) < a{*}b;\ 0 < b]\!] \Longrightarrow 0 < a$
by (*subgoal-tac* $a{*}b = b{*}a$, *auto*)
from *prems* **have** $poly\ q\ x \neq 0$ **and** *q-sgn*: $sgn\ (poly\ q\ x) =$
$(if\ x < x_0\ then\ -sgn\ (poly\ p\ x)\ else\ sgn\ (poly\ p\ x))$
by (*auto simp add*: *sgn-real-def elim*: *linorder-neqE-linordered-idom*
dest: *mult-neg-neg zero-less-mult-pos*
zero-less-mult-pos' split: *if-split-asm*)
from *sign-changes-distrib*[*OF* ‹$poly\ q\ x \neq 0$›, *of* [*p*] *ps'*]
have $sign\text{-}changes\ ps\ x = sign\text{-}changes\ [p,q]\ x + sign\text{-}changes\ (q\#ps')\ x$
by *simp*
also from *q-sgn* **and** ‹$poly\ p\ x \neq 0$›
have $sign\text{-}changes\ [p,q]\ x = (if\ x{<}x_0\ then\ 1\ else\ 0)$
by (*simp add*: *sign-changes-def sgn-zero-iff split*: *if-split-asm*)
also note *prems*(*4*)
also from *assms*(*1*) **have** $sign\text{-}changes\ (q\#ps')\ x_0 = sign\text{-}changes\ ps\ x_0$
by (*simp add*: *sign-changes-def*)
finally show *?case* **by** *simp*
qed
qed

With these two results, we can now show that if p is nonzero, the number of roots in an interval of the form $(a;b]$ is the difference of the sign changes of a Sturm sequence of p at a and b.
First, however, we prove the following auxiliary lemma that shows that if a function $f : \mathbb{R} \to \mathbb{N}$ is locally constant at any $x \in (a;b]$, it is constant across the entire interval $(a;b]$:

lemma *count-roots-between-aux*:
assumes $a \leq b$
assumes $\forall x{::}real.\ a < x \wedge x \leq b \longrightarrow eventually\ (\lambda\xi.\ f\ \xi = (f\ x{::}nat))\ (at\ x)$
shows $\forall x.\ a < x \wedge x \leq b \longrightarrow f\ x = f\ b$
proof (*clarify*)
fix *x* **assume** $x > a\ x \leq b$
with *assms* **have** $\forall x'.\ x \leq x' \wedge x' \leq b \longrightarrow$
$eventually\ (\lambda\xi.\ f\ \xi = f\ x')\ (at\ x')$ **by** *auto*
from *fun-eq-in-ivl*[*OF* ‹$x \leq b$› *this*] **show** $f\ x = f\ b$ **.**
qed

Now we can prove the actual root-counting theorem:

```
theorem (in sturm-seq) count-roots-between:
  assumes [simp]: p ≠ 0 a ≤ b
  shows sign-changes ps a − sign-changes ps b =
            card {x. x > a ∧ x ≤ b ∧ poly p x = 0}
proof−
  have sign-changes ps a − int (sign-changes ps b) =
            card {x. x > a ∧ x ≤ b ∧ poly p x = 0} using ‹a ≤ b›
  proof (induction card {x. x > a ∧ x ≤ b ∧ poly p x = 0} arbitrary: a b
            rule: less-induct)
    case (less a b)
      show ?case
      proof (cases ∃x. a < x ∧ x ≤ b ∧ poly p x = 0)
        case False
          hence no-roots: {x. a < x ∧ x ≤ b ∧ poly p x = 0} = {}
            (is ?roots=-) by auto
          hence card-roots: card ?roots = (0::int) by (subst no-roots, simp)
          show ?thesis
          proof (simp only: card-roots eq-iff-diff-eq-0[symmetric] of-nat-eq-iff,
                cases poly p a = 0)
            case False
              with no-roots show sign-changes ps a = sign-changes ps b
                by (force intro: fun-eq-in-ivl ‹a ≤ b›
                                p-nonzero-imp-sign-changes-const)
          next
            case True
             have A: ∀x. a < x ∧ x ≤ b ⟶ sign-changes ps x = sign-changes ps b
                apply (rule count-roots-between-aux, fact, clarify)
                apply (rule p-nonzero-imp-sign-changes-const)
                apply (insert False, simp)
                done
              have eventually (λx. x > a ⟶
                    sign-changes ps x = sign-changes ps a) (at a)
                 apply (rule eventually-mono [OF p-zero[OF ‹poly p a = 0› ‹p ≠
0›]])
                apply force
                done
              then obtain δ where δ-props:
                δ > 0 ∀x. x > a ∧ x < a+δ ⟶
                                sign-changes ps x = sign-changes ps a
                by (auto simp: eventually-at dist-real-def)

              show sign-changes ps a = sign-changes ps b
              proof (cases a = b)
                case False
                  define x where x = min (a+δ/2) b
                  with False have a < x x < a+δ x ≤ b
                    using ‹δ > 0› ‹a ≤ b› by simp-all
                  from δ-props ‹a < x› ‹x < a+δ›
```

have *sign-changes ps a* = *sign-changes ps x* **by** *simp*
also from *A* ‹*a* < *x*› ‹*x* ≤ *b*› **have** ... = *sign-changes ps b*
by *blast*
finally show *?thesis* **.**
qed *simp*
qed

next
case *True*
from *poly-roots-finite*[*OF assms*(*1*)]
have *fin*: *finite* {*x*. *x* > *a* ∧ *x* ≤ *b* ∧ *poly p x* = *0*}
by (*force intro*: *finite-subset*)
from *True* **have** {*x*. *x* > *a* ∧ *x* ≤ *b* ∧ *poly p x* = *0*} ≠ {} **by** *blast*
with *fin* **have** *card-greater-0*:
card {*x*. *x* > *a* ∧ *x* ≤ *b* ∧ *poly p x* = *0*} > *0* **by** *fastforce*

define x_2 **where** x_2 = *Min* {*x*. *x* > *a* ∧ *x* ≤ *b* ∧ *poly p x* = *0*}
from *Min-in*[*OF fin*] **and** *True*
have x_2*-props*: x_2 > *a* x_2 ≤ *b* *poly p* x_2 = *0*
unfolding x_2*-def* **by** *blast*+
from *Min-le*[*OF fin*] x_2*-props*
have x_2*-le*: ⋀*x'*. ⟦*x'* > *a*; *x'* ≤ *b*; *poly p x'* = *0*⟧ ⟹ x_2 ≤ *x'*
unfolding x_2*-def* **by** *simp*

have *left*: {*x*. *a* < *x* ∧ *x* ≤ x_2 ∧ *poly p x* = *0*} = {x_2}
using x_2*-props* x_2*-le* **by** *force*
hence [*simp*]: *card* {*x*. *a* < *x* ∧ *x* ≤ x_2 ∧ *poly p x* = *0*} = *1* **by** *simp*

from *p-zero*[*OF* ‹*poly p* x_2 = *0*› ‹*p* ≠ *0*›,
unfolded eventually-at dist-real-def] **guess** ε **..**
hence ε*-props*: ε > *0*
∀ *x*. *x* ≠ x_2 ∧ |*x* − x_2| < ε ⟶
sign-changes ps x = *sign-changes ps* x_2 +
(*if x* < x_2 *then 1 else 0*) **by** *auto*
define x_1 **where** x_1 = *max* (x_2 − ε / *2*) *a*
have |x_1 − x_2| < ε **using** ‹ε > *0*› x_2*-props* **by** (*simp add*: x_1*-def*)
hence *sign-changes ps* x_1 =
(*if* x_1 < x_2 *then sign-changes ps* x_2 + *1 else sign-changes ps* x_2)
using ε*-props*(*2*) **by** (*cases* x_1 = x_2, *auto*)
hence *sign-changes ps* x_1 − *sign-changes ps* x_2 = *1*
unfolding x_1*-def* **using** x_2*-props* ‹ε > *0*› **by** *simp*

also have x_2 ∉ {*x*. *a* < *x* ∧ *x* ≤ x_1 ∧ *poly p x* = *0*}
unfolding x_1*-def* **using** ‹ε > *0*› **by** *force*
with *left* **have** {*x*. *a* < *x* ∧ *x* ≤ x_1 ∧ *poly p x* = *0*} = {} **by** *force*
with *less*(*1*)[*of a* x_1] **have** *sign-changes ps* x_1 = *sign-changes ps a*
unfolding x_1*-def* ‹ε > *0*› **by** (*force simp*: *card-greater-0*)

finally have *signs-left*:

sign-changes ps a − *int* (*sign-changes ps* x_2) = *1* **by** *simp*

have {*x*. *x* > *a* ∧ *x* ≤ *b* ∧ *poly p x* = *0*} =
{*x*. *a* < *x* ∧ *x* ≤ x_2 ∧ *poly p x* = *0*} ∪
{*x*. x_2 < *x* ∧ *x* ≤ *b* ∧ *poly p x* = *0*} **using** x_2*-props* **by** *auto*
also note *left*
finally have *A*: *card* {*x*. x_2 < *x* ∧ *x* ≤ *b* ∧ *poly p x* = *0*} + *1* =
card {*x*. *a* < *x* ∧ *x* ≤ *b* ∧ *poly p x* = *0*} **using** *fin* **by** *simp*
hence *card* {*x*. x_2 < *x* ∧ *x* ≤ *b* ∧ *poly p x* = *0*} <
card {*x*. *a* < *x* ∧ *x* ≤ *b* ∧ *poly p x* = *0*} **by** *simp*
from *less*(*1*)[*OF this* x_2*-props*(*2*)] **and** *A*
have *signs-right*: *sign-changes ps* x_2 − *int* (*sign-changes ps b*) + *1* =
card {*x*. *a* < *x* ∧ *x* ≤ *b* ∧ *poly p x* = *0*} **by** *simp*

from *signs-left* **and** *signs-right* **show** *?thesis* **by** *simp*
qed
qed
thus *?thesis* **by** *simp*
qed

By applying this result to a sufficiently large upper bound, we can effectively count the number of roots "between a and infinity", i. e. the roots greater than a:

lemma (**in** *sturm-seq*) *count-roots-above*:
assumes *p* ≠ *0*
shows *sign-changes ps a* − *sign-changes-inf ps* =
card {*x*. *x* > *a* ∧ *poly p x* = *0*}
proof−
have *p* ∈ *set ps* **using** *hd-in-set*[*OF ps-not-Nil*] **by** *simp*
have *finite* (*set ps*) **by** *simp*
from *polys-inf-sign-thresholds*[*OF this*] **guess** *l u* **.**
note *lu-props* = *this*
let *?u* = *max a u*
{**fix** *x* **assume** *poly p x* = *0* **hence** *x* ≤ *?u*
using *lu-props*(*3*)[*OF* ‹*p* ∈ *set ps*›, *of x*] ‹*p* ≠ *0*›
by (*cases u* ≤ *x*, *auto simp*: *sgn-zero-iff*)
} **note** [*simp*] = *this*

from *lu-props*
have *map* (λ*p*. *sgn* (*poly p ?u*)) *ps* = *map poly-inf ps* **by** *simp*
hence *sign-changes ps a* − *sign-changes-inf ps* =
sign-changes ps a − *sign-changes ps ?u*
by (*simp-all only*: *sign-changes-def sign-changes-inf-def*)
also from *count-roots-between*[*OF assms*] *lu-props*
have ... = *card* {*x*. *a* < *x* ∧ *x* ≤ *?u* ∧ *poly p x* = *0*} **by** *simp*
also have {*x*. *a* < *x* ∧ *x* ≤ *?u* ∧ *poly p x* = *0*} = {*x*. *a* < *x* ∧ *poly p x* = *0*}
using *lu-props* **by** *auto*
finally show *?thesis* **.**
qed

The same works analogously for the number of roots below a and the total number of roots.

```
lemma (in sturm-seq) count-roots-below:
  assumes p ≠ 0
  shows sign-changes-neg-inf ps − sign-changes ps a =
           card {x. x ≤ a ∧ poly p x = 0}
proof−
  have p ∈ set ps using hd-in-set[OF ps-not-Nil] by simp
  have finite (set ps) by simp
  from polys-inf-sign-thresholds[OF this] guess l u .
  note lu-props = this
  let ?l = min a l
  {fix x assume poly p x = 0 hence x > ?l
   using lu-props(4)[OF ‹p ∈ set ps›, of x] ‹p ≠ 0›
      by (cases l < x, auto simp: sgn-zero-iff)
  } note [simp] = this

  from lu-props
    have map (λp. sgn (poly p ?l)) ps = map poly-neg-inf ps by simp
  hence sign-changes-neg-inf ps − sign-changes ps a =
           sign-changes ps ?l − sign-changes ps a
     by (simp-all only: sign-changes-def sign-changes-neg-inf-def)
  also from count-roots-between[OF assms] lu-props
     have ... =  card {x. ?l < x ∧ x ≤ a ∧ poly p x = 0} by simp
  also have {x. ?l < x ∧ x ≤ a ∧ poly p x = 0} = {x. a ≥ x ∧ poly p x = 0}
     using lu-props by auto
  finally show ?thesis .
qed

lemma (in sturm-seq) count-roots:
  assumes p ≠ 0
  shows sign-changes-neg-inf ps − sign-changes-inf ps =
           card {x. poly p x = 0}
proof−
  have finite (set ps) by simp
  from polys-inf-sign-thresholds[OF this] guess l u .
  note lu-props = this

  from lu-props
    have map (λp. sgn (poly p l)) ps = map poly-neg-inf ps
         map (λp. sgn (poly p u)) ps = map poly-inf ps by simp-all
  hence sign-changes-neg-inf ps − sign-changes-inf ps =
           sign-changes ps l − sign-changes ps u
     by (simp-all only: sign-changes-def sign-changes-inf-def
                        sign-changes-neg-inf-def)
  also from count-roots-between[OF assms] lu-props
     have ... =  card {x. l < x ∧ x ≤ u ∧ poly p x = 0} by simp
  also have {x. l < x ∧ x ≤ u ∧ poly p x = 0} = {x. poly p x = 0}
     using lu-props assms by simp
```

finally show *?thesis* **.**
qed

2.4 Constructing Sturm sequences

2.5 The canonical Sturm sequence

In this subsection, we will present the canonical Sturm sequence construction for a polynomial p without multiple roots that is very similar to the Euclidean algorithm:

$$p_i = \begin{cases} p & \text{for } i = 1 \\ p' & \text{for } i = 2 \\ -p_{i-2} \bmod p_{i-1} & \text{otherwise} \end{cases}$$

We break off the sequence at the first constant polynomial.

function *sturm-aux* **where**
sturm-aux (*p* :: *real poly*) *q* =
(*if degree q* = *0 then* [*p*,*q*] *else p* # *sturm-aux q* (−(*p mod q*)))
by (*pat-completeness*, *simp-all*)
termination by (*relation measure* (*degree* ∘ *snd*),
simp-all add: *o-def degree-mod-less*′)

definition *sturm* **where** *sturm p* = *sturm-aux p* (*pderiv p*)

Next, we show some simple facts about this construction:

lemma *sturm-0*[*simp*]: *sturm 0* = [*0*,*0*]
by (*unfold sturm-def*, *subst sturm-aux.simps*, *simp*)

lemma [*simp*]: *sturm-aux p q* = [] ⟷ *False*
by (*induction p q rule*: *sturm-aux.induct*, *subst sturm-aux.simps*, *auto*)

lemma *sturm-neq-Nil*[*simp*]: *sturm p* ≠ [] **unfolding** *sturm-def* **by** *simp*

lemma [*simp*]: *hd* (*sturm p*) = *p*
unfolding *sturm-def* **by** (*subst sturm-aux.simps*, *simp*)

lemma [*simp*]: *p* ∈ *set* (*sturm p*)
using *hd-in-set*[*OF sturm-neq-Nil*] **by** *simp*

lemma [*simp*]: *length* (*sturm p*) ≥ *2*
proof−
{fix *q* **have** *length* (*sturm-aux p q*) ≥ *2*
by (*induction p q rule*: *sturm-aux.induct*, *subst sturm-aux.simps*, *auto*)
}
thus *?thesis* **unfolding** *sturm-def* **.**
qed

```
lemma [simp]: degree (last (sturm p)) = 0
proof-
  {fix q have degree (last (sturm-aux p q)) = 0
          by (induction p q rule: sturm-aux.induct, subst sturm-aux.simps, simp)
  }
  thus ?thesis unfolding sturm-def .
qed

lemma [simp]: sturm-aux p q ! 0 = p
    by (subst sturm-aux.simps, simp)
lemma [simp]: sturm-aux p q ! Suc 0 = q
    by (subst sturm-aux.simps, simp)

lemma [simp]: sturm p ! 0 = p
    unfolding sturm-def by simp
lemma [simp]: sturm p ! Suc 0 = pderiv p
    unfolding sturm-def by simp


lemma sturm-indices:
  assumes i < length (sturm p) − 2
  shows sturm p!(i+2) = −(sturm p!i mod sturm p!(i+1))
proof-
 {fix ps q
  have ⟦ps = sturm-aux p q; i < length ps − 2⟧
          ⟹ ps!(i+2) = −(ps!i mod ps!(i+1))
  proof (induction p q arbitrary: ps i rule: sturm-aux.induct)
    case (1 p q)
      show ?case
      proof (cases i = 0)
        case False
          then obtain i' where [simp]: i = Suc i' by (cases i, simp-all)
          hence length ps ≥ 4 using 1 by simp
          with 1(2) have deg: degree q ≠ 0
              by (subst (asm) sturm-aux.simps, simp split: if-split-asm)
          with 1(2) obtain ps' where [simp]: ps = p # ps'
              by (subst (asm) sturm-aux.simps, simp)
          with 1(2) deg have ps': ps' = sturm-aux q (−(p mod q))
              by (subst (asm) sturm-aux.simps, simp)
          from ‹length ps ≥ 4› and ‹ps = p # ps'› 1(3) False
              have i − 1 < length ps' − 2 by simp
          from 1(1)[OF deg ps' this]
              show ?thesis by simp
      next
        case True
          with 1(3) have length ps ≥ 3 by simp
          with 1(2) have degree q ≠ 0
              by (subst (asm) sturm-aux.simps, simp split: if-split-asm)
```

```
        with 1(2) have [simp]: sturm-aux p q ! Suc (Suc 0) = −(p mod q)
          by (subst sturm-aux.simps, simp)
        from True have ps!i = p ps!(i+1) = q ps!(i+2) = −(p mod q)
          by (simp-all add: 1(2))
        thus ?thesis by simp
    qed
  qed}
  from this[OF sturm-def assms] show ?thesis .
qed
```

If the Sturm sequence construction is applied to polynomials p and q, the greatest common divisor of p and q a divisor of every element in the sequence. This is obvious from the similarity to Euclid's algorithm for computing the GCD.

```
lemma sturm-aux-gcd: r ∈ set (sturm-aux p q) ⟹ gcd p q dvd r
proof (induction p q rule: sturm-aux.induct)
  case (1 p q)
    show ?case
    proof (cases r = p)
      case False
        with 1(2) have r: r ∈ set (sturm-aux q (−(p mod q)))
          by (subst (asm) sturm-aux.simps, simp split: if-split-asm,
              subst sturm-aux.simps, simp)
        show ?thesis
        proof (cases degree q = 0)
          case False
            hence q ≠ 0 by force
            with 1(1) [OF False r] show ?thesis
              by (simp add: gcd-mod-right ac-simps)
        next
          case True
            with 1(2) and ‹r ≠ p› have r = q
                by (subst (asm) sturm-aux.simps, simp)
            thus ?thesis by simp
        qed
    qed simp
qed

lemma sturm-gcd: r ∈ set (sturm p) ⟹ gcd p (pderiv p) dvd r
    unfolding sturm-def by (rule sturm-aux-gcd)
```

If two adjacent polynomials in the result of the canonical Sturm chain construction both have a root at some x, this x is a root of all polynomials in the sequence.

```
lemma sturm-adjacent-root-propagate-left:
  assumes i < length (sturm (p :: real poly)) − 1
  assumes poly (sturm p ! i) x = 0
      and poly (sturm p ! (i + 1)) x = 0
```

shows $\forall j \leq i+1.\ poly\ (sturm\ p\ !\ j)\ x = 0$
using *assms(2)*
proof (*intro sturm-adjacent-root-aux*[*OF assms(1,2,3)*], *goal-cases*)
case *prems*: (*1 i x*)
let $?p = sturm\ p\ !\ i$
let $?q = sturm\ p\ !\ (i + 1)$
let $?r = sturm\ p\ !\ (i + 2)$
from *sturm-indices*[*OF prems(2)*] **have** $?p = ?p\ div\ ?q * ?q - ?r$
by (*simp add*: *div-mult-mod-eq*)
hence $poly\ ?p\ x = poly\ (?p\ div\ ?q * ?q - ?r)\ x$ **by** *simp*
hence $poly\ ?p\ x = -poly\ ?r\ x$ **using** *prems(3)* **by** *simp*
thus *?case* **by** (*simp add*: *sgn-minus*)
qed

Consequently, if this is the case in the canonical Sturm chain of p, p must have multiple roots.

lemma *sturm-adjacent-root-not-squarefree*:
assumes $i < length\ (sturm\ (p :: real\ poly)) - 1$
$poly\ (sturm\ p\ !\ i)\ x = 0$ $poly\ (sturm\ p\ !\ (i + 1))\ x = 0$
shows $\neg rsquarefree\ p$
proof−
from *sturm-adjacent-root-propagate-left*[*OF assms*]
have $poly\ p\ x = 0$ $poly\ (pderiv\ p)\ x = 0$ **by** *auto*
thus *?thesis* **by** (*auto simp*: *rsquarefree-roots*)
qed

Since the second element of the sequence is chosen to be the derivative of p, p_1 and p_2 fulfil the property demanded by the definition of a Sturm sequence that they locally have opposite sign left of a root x of p and the same sign to the right of x.

lemma *sturm-firsttwo-signs-aux*:
assumes $(p :: real\ poly) \neq 0$ $q \neq 0$
assumes *q-pderiv*:
$eventually\ (\lambda x.\ sgn\ (poly\ q\ x) = sgn\ (poly\ (pderiv\ p)\ x))\ (at\ x_0)$
assumes *p-0*: $poly\ p\ (x_0::real) = 0$
shows $eventually\ (\lambda x.\ sgn\ (poly\ (p*q)\ x) = (if\ x > x_0\ then\ 1\ else\ -1))\ (at\ x_0)$
proof−
have *A*: $eventually\ (\lambda x.\ poly\ p\ x \neq 0 \land poly\ q\ x \neq 0 \land$
$sgn\ (poly\ q\ x) = sgn\ (poly\ (pderiv\ p)\ x))\ (at\ x_0)$
using ‹$p \neq 0$› ‹$q \neq 0$›
by (*intro poly-neighbourhood-same-sign q-pderiv*
poly-neighbourhood-without-roots eventually-conj)
then obtain ε **where** *ε-props*: $\varepsilon > 0$ $\forall x.\ x \neq x_0 \land |x - x_0| < \varepsilon \longrightarrow$
$poly\ p\ x \neq 0 \land poly\ q\ x \neq 0 \land sgn\ (poly\ (pderiv\ p)\ x) = sgn\ (poly\ q\ x)$
by (*auto simp*: *eventually-at dist-real-def*)
have *sqr-pos*: $\bigwedge x::real.\ x \neq 0 \Longrightarrow sgn\ x * sgn\ x = 1$
by (*auto simp*: *sgn-real-def*)

show *?thesis*

proof (*simp only*: *eventually-at dist-real-def*, *rule exI*[*of* - ε],
intro conjI, *fact* ‹$\varepsilon > 0$›, *clarify*)
fix x **assume** $x \neq x_0$ $|x - x_0| < \varepsilon$
with ε*-props* **have** [*simp*]: *poly p x* $\neq$ *0 poly q x* $\neq$ *0*
sgn (*poly* (*pderiv p*) *x*) = *sgn* (*poly q x*) **by** *auto*
show *sgn* (*poly* (*p*∗*q*) *x*) = (*if* $x > x_0$ *then 1 else* −*1*)
proof (*cases* $x \geq x_0$)
case *True*
with ‹$x \neq x_0$› **have** $x > x_0$ **by** *simp*
from *poly-MVT*[*OF this*, *of p*] **guess** ξ **..**
note ξ*-props* = *this*
with ‹$|x - x_0| < \varepsilon$› ‹*poly p* x_0 = *0*› ‹$x > x_0$› ε*-props*
have $|\xi - x_0| < \varepsilon$ *sgn* (*poly p x*) = *sgn* ($x - x_0$) ∗ *sgn* (*poly q* ξ)
by (*auto simp add*: *q-pderiv sgn-mult*)
moreover from ξ*-props* ε*-props* ‹$|x - x_0| < \varepsilon$›
have $\forall t.\ \xi \leq t \wedge t \leq x \longrightarrow$ *poly q t* $\neq$ *0* **by** *auto*
hence *sgn* (*poly q* ξ) = *sgn* (*poly q x*) **using** ξ*-props* ε*-props*
by (*intro no-roots-inbetween-imp-same-sign*, *simp-all*)
ultimately show *?thesis* **using** *True* ‹$x \neq x_0$› ε*-props* ξ*-props*
by (*auto simp*: *sgn-mult sqr-pos*)
next
case *False*
hence $x < x_0$ **by** *simp*
hence *sgn*: *sgn* ($x - x_0$) = −*1* **by** *simp*
from *poly-MVT*[*OF* ‹$x < x_0$›, *of p*] **guess** ξ **..**
note ξ*-props* = *this*
with ‹$|x - x_0| < \varepsilon$› ‹*poly p* x_0 = *0*› ‹$x < x_0$› ε*-props*
have $|\xi - x_0| < \varepsilon$ *poly p x* = ($x - x_0$) ∗ *poly* (*pderiv p*) ξ
poly p $\xi \neq$ *0* **by** (*auto simp*: *field-simps*)
hence *sgn* (*poly p x*) = *sgn* ($x - x_0$) ∗ *sgn* (*poly q* ξ)
using ε*-props* ξ*-props* **by** (*auto simp*: *q-pderiv sgn-mult*)
moreover from ξ*-props* ε*-props* ‹$|x - x_0| < \varepsilon$›
have $\forall t.\ x \leq t \wedge t \leq \xi \longrightarrow$ *poly q t* $\neq$ *0* **by** *auto*
hence *sgn* (*poly q* ξ) = *sgn* (*poly q x*) **using** ξ*-props* ε*-props*
by (*rule-tac sym*, *intro no-roots-inbetween-imp-same-sign*, *simp-all*)
ultimately show *?thesis* **using** *False* ‹$x \neq x_0$›
by (*auto simp*: *sgn-mult sqr-pos*)
qed
qed
qed

lemma *sturm-firsttwo-signs*:
fixes *ps* :: *real poly list*
assumes *squarefree*: *rsquarefree p*
assumes *p-0*: *poly p* (x_0::*real*) = *0*
shows *eventually* ($\lambda x.$ *sgn* (*poly* (*p* ∗ *sturm p* ! *1*) *x*) =
(*if* $x > x_0$ *then 1 else* −*1*)) (*at* x_0)
proof−
from *assms* **have** [*simp*]: *p* $\neq$ *0* **by** (*auto simp add*: *rsquarefree-roots*)

```
  with squarefree p-0 have [simp]: pderiv p ≠ 0
      by (auto simp  add:rsquarefree-roots)
  from assms show ?thesis
      by (intro sturm-firsttwo-signs-aux,
          simp-all add: rsquarefree-roots)
qed
```

The construction also obviously fulfils the property about three adjacent polynomials in the sequence.

```
lemma sturm-signs:
  assumes squarefree: rsquarefree p
  assumes i-in-range: i < length (sturm (p :: real poly)) − 2
  assumes q-0: poly (sturm p ! (i+1)) x = 0 (is poly ?q x = 0)
  shows poly (sturm p ! (i+2)) x ∗ poly (sturm p ! i) x < 0
           (is poly ?p x ∗ poly ?r x < 0)
proof−
  from sturm-indices[OF i-in-range]
      have sturm p ! (i+2) = − (sturm p ! i mod sturm p ! (i+1))
          (is ?r = − (?p mod ?q)) .
  hence −?r = ?p mod ?q by simp
  with div-mult-mod-eq[of ?p ?q] have ?p div ?q ∗ ?q − ?r = ?p by simp
  hence poly (?p div ?q) x ∗ poly ?q x − poly ?r x = poly ?p x
      by (metis poly-diff poly-mult)
  with q-0 have r-x: poly ?r x = −poly ?p x by simp
  moreover have sqr-pos: ⋀x::real. x ≠ 0 ⟹ x ∗ x > 0 apply (case-tac x ≥ 0)
      by (simp-all add: mult-neg-neg)
  from sturm-adjacent-root-not-squarefree[of i p] assms r-x
      have poly ?p x ∗ poly ?p x > 0 by (force intro: sqr-pos)
  ultimately show poly ?r x ∗ poly ?p x < 0 by simp
qed
```

Finally, if p contains no multiple roots, *sturm p*, i.e. the canonical Sturm sequence for p, is a Sturm sequence and can be used to determine the number of roots of p.

```
lemma sturm-seq-sturm[simp]:
   assumes rsquarefree p
   shows sturm-seq (sturm p) p
proof
  show sturm p ≠ [] by simp
  show hd (sturm p) = p by simp
  show length (sturm p) ≥ 2 by simp
  from assms show ⋀x. ¬(poly p x = 0 ∧ poly (sturm p ! 1) x = 0)
      by (simp add: rsquarefree-roots)
next
  fix x :: real and y :: real
  have degree (last (sturm p)) = 0 by simp
  then obtain c where last (sturm p) = [:c:]
      by (cases last (sturm p), simp split: if-split-asm)
  thus ⋀x y. sgn (poly (last (sturm p)) x) =
```

$sgn\ (poly\ (last\ (sturm\ p))\ y)$ **by** *simp*
next
from *sturm-firsttwo-signs*[*OF assms*]
show $\bigwedge x_0.\ poly\ p\ x_0 = 0 \Longrightarrow$
$eventually\ (\lambda x.\ sgn\ (poly\ (p*sturm\ p\ !\ 1)\ x) =$
$(if\ x > x_0\ then\ 1\ else\ -1))\ (at\ x_0)$ **by** *simp*
next
from *sturm-signs*[*OF assms*]
show $\bigwedge i\ x.\ [\![i < length\ (sturm\ p) - 2;\ poly\ (sturm\ p\ !\ (i + 1))\ x = 0]\!]$
$\Longrightarrow poly\ (sturm\ p\ !\ (i + 2))\ x * poly\ (sturm\ p\ !\ i)\ x < 0$ **by** *simp*
qed

2.5.1 Canonical squarefree Sturm sequence

The previous construction does not work for polynomials with multiple roots, but we can simply "divide away" multiple roots by dividing p by the GCD of p and p'. The resulting polynomial has the same roots as p, but with multiplicity 1, allowing us to again use the canonical construction.

definition *sturm-squarefree* **where**
$sturm\text{-}squarefree\ p = sturm\ (p\ div\ (gcd\ p\ (pderiv\ p)))$

lemma *sturm-squarefree-not-Nil*[*simp*]: $sturm\text{-}squarefree\ p \neq []$
by (*simp add*: *sturm-squarefree-def*)

lemma *sturm-seq-sturm-squarefree*:
assumes [*simp*]: $p \neq 0$
defines [*simp*]: $p' \equiv p\ div\ gcd\ p\ (pderiv\ p)$
shows $sturm\text{-}seq\ (sturm\text{-}squarefree\ p)\ p'$
proof
have $rsquarefree\ p'$
proof (*subst rsquarefree-roots*, *clarify*)
fix x **assume** $poly\ p'\ x = 0\ poly\ (pderiv\ p')\ x = 0$
hence $[:-x,1:]\ dvd\ gcd\ p'\ (pderiv\ p')$ **by** (*simp add*: *poly-eq-0-iff-dvd*)
also from *poly-div-gcd-squarefree*(*1*)[*OF assms*(*1*)]
have $gcd\ p'\ (pderiv\ p') = 1$ **by** *simp*
finally show *False* **by** (*simp add*: *poly-eq-0-iff-dvd*[*symmetric*])
qed

from *sturm-seq-sturm*[*OF* ‹*rsquarefree p'*›]
interpret *sturm-seq*: $sturm\text{-}seq\ sturm\text{-}squarefree\ p\ p'$
by (*simp add*: *sturm-squarefree-def*)

show $\bigwedge x\ y.\ sgn\ (poly\ (last\ (sturm\text{-}squarefree\ p))\ x) =$
$sgn\ (poly\ (last\ (sturm\text{-}squarefree\ p))\ y)$ **by** *simp*
show $sturm\text{-}squarefree\ p \neq []$ **by** *simp*
show $hd\ (sturm\text{-}squarefree\ p) = p'$ **by** (*simp add*: *sturm-squarefree-def*)
show $length\ (sturm\text{-}squarefree\ p) \geq 2$ **by** *simp*

have [*simp*]: *sturm-squarefree p* ! *0* = *p'*
sturm-squarefree p ! *Suc 0* = *pderiv p'*
by (*simp-all add*: *sturm-squarefree-def*)

from ‹*rsquarefree p'*›
show ⋀*x*. ¬ (*poly p' x* = *0* ∧ *poly* (*sturm-squarefree p* ! *1*) *x* = *0*)
by (*simp add*: *rsquarefree-roots*)

from *sturm-seq.signs* **show** ⋀*i x*. ⟦*i* < *length* (*sturm-squarefree p*) − *2*;
poly (*sturm-squarefree p* ! (*i* + *1*)) *x* = *0*⟧
⟹ *poly* (*sturm-squarefree p* ! (*i* + *2*)) *x* ∗
poly (*sturm-squarefree p* ! *i*) *x* < *0* **.**

from *sturm-seq.deriv* **show** ⋀x_0. *poly p'* x_0 = *0* ⟹
eventually (λ*x*. *sgn* (*poly* (*p'* ∗ *sturm-squarefree p* ! *1*) *x*) =
(*if x* > x_0 *then 1 else* −*1*)) (*at* x_0) **.**
qed

2.5.2 Optimisation for multiple roots

We can also define the following non-canonical Sturm sequence that is obtained by taking the canonical Sturm sequence of *p* (possibly with multiple roots) and then dividing the entire sequence by the GCD of *p* and its derivative.

definition *sturm-squarefree'* **where**
sturm-squarefree' p = (*let d* = *gcd p* (*pderiv p*)
in map (λ*p'*. *p' div d*) (*sturm p*))

This construction also has all the desired properties:

lemma *sturm-squarefree'-adjacent-root-propagate-left*:
assumes *p* ≠ *0*
assumes *i* < *length* (*sturm-squarefree'* (*p* :: *real poly*)) − *1*
assumes *poly* (*sturm-squarefree' p* ! *i*) *x* = *0*
and *poly* (*sturm-squarefree' p* ! (*i* + *1*)) *x* = *0*
shows ∀ *j*≤*i*+*1*. *poly* (*sturm-squarefree' p* ! *j*) *x* = *0*
proof (*intro sturm-adjacent-root-aux*[*OF assms*(*2*,*3*,*4*)], *goal-cases*)
case *prems*: (*1 i x*)
define *q* **where** *q* = *sturm p* ! *i*
define *r* **where** *r* = *sturm p* ! (*Suc i*)
define *s* **where** *s* = *sturm p* ! (*Suc* (*Suc i*))
define *d* **where** *d* = *gcd p* (*pderiv p*)
define *q' r' s'* **where** *q'* = *q div d* **and** *r'* = *r div d* **and** *s'* = *s div d*
from ‹*p* ≠ *0*› **have** *d* ≠ *0* **unfolding** *d-def* **by** *simp*
from *prems*(*1*) **have** *i-in-range*: *i* < *length* (*sturm p*) − *2*
unfolding *sturm-squarefree'-def Let-def* **by** *simp*
have [*simp*]: *d dvd q d dvd r d dvd s* **unfolding** *q-def r-def s-def d-def*
using *i-in-range* **by** (*auto intro*: *sturm-gcd*)

```
    hence qrs-simps: q = q′ ∗ d r = r′ ∗ d s = s′ ∗ d
        unfolding q′-def r′-def s′-def by (simp-all)
    with prems(2) i-in-range have r′-0: poly r′ x = 0
        unfolding r′-def r-def d-def sturm-squarefree′-def Let-def by simp
    hence r-0: poly r x = 0 by (simp add: ‹r = r′ ∗ d›)
    from sturm-indices[OF i-in-range] have q = q div r ∗ r − s
        unfolding q-def r-def s-def by (simp add: div-mult-mod-eq)
    hence q′ = (q div r ∗ r − s) div d by (simp add: q′-def)
    also have ... = (q div r ∗ r) div d − s′
      by (simp add: s′-def poly-div-diff-left)
    also have ... = q div r ∗ r′ − s′
        using dvd-div-mult[OF ‹d dvd r›, of q div r]
        by (simp add: algebra-simps r′-def)
    also have q div r = q′ div r′ by (simp add: qrs-simps ‹d ≠ 0›)
    finally have poly q′ x = poly (q′ div r′ ∗ r′ − s′) x by simp
    also from r′-0 have ... = −poly s′ x by simp
    finally have poly s′ x = −poly q′ x by simp
    thus ?case using i-in-range
        unfolding q′-def s′-def q-def s-def sturm-squarefree′-def Let-def
        by (simp add: d-def sgn-minus)
qed

lemma sturm-squarefree′-adjacent-roots:
  assumes p ≠ 0
          i < length (sturm-squarefree′ (p :: real poly)) − 1
          poly (sturm-squarefree′ p ! i) x = 0
          poly (sturm-squarefree′ p ! (i + 1)) x = 0
  shows False
proof−
  define d where d = gcd p (pderiv p)
  from sturm-squarefree′-adjacent-root-propagate-left[OF assms]
      have poly (sturm-squarefree′ p ! 0) x = 0
           poly (sturm-squarefree′ p ! 1) x = 0 by auto
  hence poly (p div d) x = 0 poly (pderiv p div d) x = 0
      using assms(2)
      unfolding sturm-squarefree′-def Let-def d-def by auto
  moreover from div-gcd-coprime assms(1)
      have coprime (p div d) (pderiv p div d) unfolding d-def by auto
  ultimately show False using coprime-imp-no-common-roots by auto
qed

lemma sturm-squarefree′-signs:
  assumes p ≠ 0
  assumes i-in-range: i < length (sturm-squarefree′ (p :: real poly)) − 2
  assumes q-0: poly (sturm-squarefree′ p ! (i+1)) x = 0 (is poly ?q x = 0)
  shows poly (sturm-squarefree′ p ! (i+2)) x ∗
        poly (sturm-squarefree′ p ! i) x < 0
          (is poly ?r x ∗ poly ?p x < 0)
proof−
```

```
  define d where d = gcd p (pderiv p)
  with ‹p ≠ 0› have [simp]: d ≠ 0 by simp
  from poly-div-gcd-squarefree(1)[OF ‹p ≠ 0›]
       coprime-imp-no-common-roots
    have rsquarefree: rsquarefree (p div d)
    by (auto simp: rsquarefree-roots d-def)

  from i-in-range have i-in-range': i < length (sturm p) − 2
    unfolding sturm-squarefree'-def by simp
  hence d dvd (sturm p ! i) (is d dvd ?p')
        d dvd (sturm p ! (Suc i)) (is d dvd ?q')
        d dvd (sturm p ! (Suc (Suc i))) (is d dvd ?r')
    unfolding d-def by (auto intro: sturm-gcd)
  hence pqr-simps: ?p' = ?p ∗ d ?q' = ?q ∗ d ?r' = ?r ∗ d
   unfolding sturm-squarefree'-def Let-def d-def using i-in-range'
   by (auto simp: dvd-div-mult-self)
  with q-0 have q'-0: poly ?q' x = 0 by simp
  from sturm-indices[OF i-in-range']
    have sturm p ! (i+2) = − (sturm p ! i mod sturm p ! (i+1)) .
  hence −?r' = ?p' mod ?q' by simp
  with div-mult-mod-eq[of ?p' ?q'] have ?p' div ?q' ∗ ?q' − ?r' = ?p' by simp
  hence d∗(?p div ?q ∗ ?q − ?r) = d∗ ?p by (simp add: pqr-simps algebra-simps)
  hence ?p div ?q ∗ ?q − ?r = ?p by simp
  hence poly (?p div ?q) x ∗ poly ?q x − poly ?r x = poly ?p x
    by (metis poly-diff poly-mult)
  with q-0 have r-x: poly ?r x = −poly ?p x by simp

  from sturm-squarefree'-adjacent-roots[OF ‹p ≠ 0›] i-in-range q-0
    have poly ?p x ≠ 0 by force
  moreover have sqr-pos: ⋀x::real. x ≠ 0 ⟹ x ∗ x > 0 apply (case-tac x ≥ 0)
    by (simp-all add: mult-neg-neg)
  ultimately show ?thesis using r-x by simp
qed
```

This approach indeed also yields a valid squarefree Sturm sequence for the polynomial $p/\gcd(p, p')$.

```
lemma sturm-seq-sturm-squarefree':
  assumes (p :: real poly) ≠ 0
  defines d ≡ gcd p (pderiv p)
  shows sturm-seq (sturm-squarefree' p) (p div d)
    (is sturm-seq ?ps' ?p')
proof
  show ?ps' ≠ [] hd ?ps' = ?p' 2 ≤ length ?ps'
    by (simp-all add: sturm-squarefree'-def d-def hd-map)

  from assms have d ≠ 0 by simp
  {
   have d dvd last (sturm p) unfolding d-def
     by (rule sturm-gcd, simp)
```

```
    hence ∗: last (sturm p) = last ?ps′ ∗ d
        by (simp add: sturm-squarefree′-def last-map d-def dvd-div-mult-self)
    then have last ?ps′ dvd last (sturm p) by simp
    with ∗ dvd-imp-degree-le[OF this] have degree (last ?ps′) ≤ degree (last (sturm
p))
        using ‹d ≠ 0› by (cases last ?ps′ = 0) auto
    hence degree (last ?ps′) = 0 by simp
    then obtain c where last ?ps′ = [:c:]
        by (cases last ?ps′, simp split: if-split-asm)
    thus ⋀x y. sgn (poly (last ?ps′) x) = sgn (poly (last ?ps′) y) by simp
  }

  have squarefree: rsquarefree ?p′ using ‹p ≠ 0›
    by (subst rsquarefree-roots, unfold d-def,
        intro allI coprime-imp-no-common-roots poly-div-gcd-squarefree)
  have [simp]: sturm-squarefree′ p ! Suc 0 = pderiv p div d
      unfolding sturm-squarefree′-def Let-def sturm-def d-def
        by (subst sturm-aux.simps, simp)
  have coprime: coprime ?p′ (pderiv p div d)
      unfolding d-def using div-gcd-coprime ‹p ≠ 0› by blast
  thus squarefree′:
      ⋀x. ¬ (poly (p div d) x = 0 ∧ poly (sturm-squarefree′ p ! 1) x = 0)
      using coprime-imp-no-common-roots by simp

  from sturm-squarefree′-signs[OF ‹p ≠ 0›]
      show ⋀i x. ⟦i < length ?ps′ − 2; poly (?ps′ ! (i + 1)) x = 0⟧
              ⟹ poly (?ps′ ! (i + 2)) x ∗ poly (?ps′ ! i) x < 0 .

  have [simp]: ?p′ ≠ 0 using squarefree by (simp add: rsquarefree-def)
  have A: ?p′ = ?ps′ ! 0 pderiv p div d = ?ps′ ! 1
      by (simp-all add: sturm-squarefree′-def Let-def d-def sturm-def,
          subst sturm-aux.simps, simp)
  have [simp]: ?ps′ ! 0 ≠ 0 using squarefree
      by (auto simp: A rsquarefree-def)

  fix x0 :: real
  assume poly ?p′ x0 = 0
  hence poly p x0 = 0 using poly-div-gcd-squarefree(2)[OF ‹p ≠ 0›]
      unfolding d-def by simp
  hence pderiv p ≠ 0 using ‹p ≠ 0› by (auto dest: pderiv-iszero)
  with ‹p ≠ 0› ‹poly p x0 = 0›
      have A: eventually (λx. sgn (poly (p ∗ pderiv p) x) =
                          (if x0 < x then 1 else −1)) (at x0)
      by (intro sturm-firsttwo-signs-aux, simp-all)
  note ev = eventually-conj[OF A poly-neighbourhood-without-roots[OF ‹d ≠ 0›]]

  show eventually (λx. sgn (poly (p div d ∗ sturm-squarefree′ p ! 1) x) =
                    (if x0 < x then 1 else −1)) (at x0)
  proof (rule eventually-mono[OF ev], goal-cases)
```

have [*intro*]:
  $\bigwedge a\ (b{::}real).\ b \neq 0 \Longrightarrow a < 0 \Longrightarrow a\ /\ (b * b) < 0$
  $\bigwedge a\ (b{::}real).\ b \neq 0 \Longrightarrow a > 0 \Longrightarrow a\ /\ (b * b) > 0$
  by ((*case-tac b* > *0*,
    auto simp: *mult-neg-neg field-simps*) [])+
case *prems*: (*1 x*)
hence [*simp*]: *poly d x* ∗ *poly d x* > *0*
  by (*cases poly d x* > *0*, *auto simp*: *mult-neg-neg*)
from *poly-div-gcd-squarefree-aux*(*2*)[*OF* ‹*pderiv p* ≠ *0*›]
  have *poly* (*p div d*) *x* = *0* ⟷ *poly p x* = *0* **by** (*simp add*: *d-def*)
moreover have *d dvd p d dvd pderiv p* **unfolding** *d-def* **by** *simp-all*
ultimately show *?case* **using** *prems*
  by (*auto simp*: *sgn-real-def poly-div not-less*[*symmetric*]
      zero-less-divide-iff split: *if-split-asm*)
qed
qed

This construction is obviously more expensive to compute than the one that *first* divides p by $\gcd(p, p')$ and *then* applies the canonical construction. In this construction, we *first* compute the canonical Sturm sequence of p as if it had no multiple roots and *then* divide by the GCD. However, it can be seen quite easily that unless x is a multiple root of p, i.e. as long as $\gcd(P, P') \neq 0$, the number of sign changes in a sequence of polynomials does not actually change when we divide the polynomials by $\gcd(p, p')$. Therefore we can use the canonical Sturm sequence even in the non-squarefree case as long as the borders of the interval we are interested in are not multiple roots of the polynomial.

lemma *sign-changes-mult-aux*:
assumes *d* ≠ (*0*::*real*)
shows *length* (*remdups-adj* (*filter* (λ*x*. *x* ≠ *0*) (*map* ((∗) *d* ∘ *f*) *xs*))) =
    length (*remdups-adj* (*filter* (λ*x*. *x* ≠ *0*) (*map f xs*)))
proof−
from *assms* **have** *inj*: *inj* ((∗) *d*) **by** (*auto intro*: *injI*)
from *assms* **have** [*simp*]: *filter* (λ*x*. ((∗) *d* ∘ *f*) *x* ≠ *0*) = *filter* (λ*x*. *f x* ≠ *0*)
      filter ((λ*x*. *x* ≠ *0*) ∘ *f*) = *filter* (λ*x*. *f x* ≠ *0*)
  by (*simp-all add*: *o-def*)
have *filter* (λ*x*. *x* ≠ *0*) (*map* ((∗) *d* ∘ *f*) *xs*) =
    map ((∗) *d* ∘ *f*) (*filter* (λ*x*. ((∗) *d* ∘ *f*) *x* ≠ *0*) *xs*)
  by (*simp add*: *filter-map o-def*)
thus *?thesis* **using** *remdups-adj-map-injective*[*OF inj*] *assms*
  by (*simp add*: *filter-map map-map*[*symmetric*] *del*: *map-map*)
qed

lemma *sturm-sturm-squarefree′-same-sign-changes*:
fixes *p* :: *real poly*
defines *ps* ≡ *sturm p* **and** *ps′* ≡ *sturm-squarefree′ p*
shows *poly p x* ≠ *0* ∨ *poly* (*pderiv p*) *x* ≠ *0* ⟹
    sign-changes ps′ x = *sign-changes ps x*

```
    p ≠ 0 ⟹ sign-changes-inf ps' = sign-changes-inf ps
    p ≠ 0 ⟹ sign-changes-neg-inf ps' = sign-changes-neg-inf ps
proof−
  define d where d = gcd p (pderiv p)
  define p' where p' = p div d
  define s' where s' = poly-inf d
  define s'' where s'' = poly-neg-inf d

  {
    fix x :: real and q :: real poly
    assume q ∈ set ps
    hence d dvd q unfolding d-def ps-def using sturm-gcd by simp
    hence q-prod: q = (q div d) ∗ d unfolding p'-def d-def
        by (simp add: algebra-simps dvd-mult-div-cancel)

    have poly q x = poly d x ∗ poly (q div d) x  by (subst q-prod, simp)
    hence s1: sgn (poly q x) = sgn (poly d x) ∗ sgn (poly (q div d) x)
        by (subst q-prod, simp add: sgn-mult)
    from poly-inf-mult have s2: poly-inf q = s' ∗ poly-inf (q div d)
        unfolding s'-def by (subst q-prod, simp)
    from poly-inf-mult have s3: poly-neg-inf q = s'' ∗ poly-neg-inf (q div d)
        unfolding s''-def by (subst q-prod, simp)
    note s1 s2 s3
  }
  note signs = this

  {
    fix f :: real poly ⇒ real and s :: real
    assume f: ⋀q. q ∈ set ps ⟹ f q = s ∗ f (q div d) and s: s ≠ 0
    hence inverse s ≠ 0 by simp
    {fix q assume q ∈ set ps
     hence f (q div d) = inverse s ∗ f q
         by (subst f[of q], simp-all add: s)
    } note f' = this
    have length (remdups-adj [x←map f (map (λq. q div d) ps). x ≠ 0]) − 1 =
            length (remdups-adj [x←map (λq. f (q div d)) ps . x ≠ 0]) − 1
        by (simp only: sign-changes-def o-def map-map)
    also have map (λq. q div d) ps = ps'
        by (simp add: ps-def ps'-def sturm-squarefree'-def Let-def d-def)
    also from f' have map (λq. f (q div d)) ps =
                      map (λx. ((∗)(inverse s) ∘ f) x) ps by (simp add: o-def)
    also note sign-changes-mult-aux[OF ‹inverse s ≠ 0›, of f ps]
    finally have
        length (remdups-adj [x←map f ps' . x ≠ 0]) − 1 =
         length (remdups-adj [x←map f ps . x ≠ 0]) − 1 by simp
  }
  note length-remdups-adj = this

  {
```

```
    fix x assume A: poly p x ≠ 0 ∨ poly (pderiv p) x ≠ 0
    have d dvd p d dvd pderiv p unfolding d-def by simp-all
    with A have sgn (poly d x) ≠ 0
        by (auto simp add: sgn-zero-iff elim: dvdE)
    thus sign-changes ps' x = sign-changes ps x using signs(1)
        unfolding sign-changes-def
        by (intro length-remdups-adj[of λq. sgn (poly q x)], simp-all)
  }

  assume p ≠ 0
  hence d ≠ 0 unfolding d-def by simp
  hence s' ≠ 0 s'' ≠ 0 unfolding s'-def s''-def by simp-all
  from length-remdups-adj[of poly-inf s', OF signs(2) ‹s' ≠ 0›]
      show sign-changes-inf ps' = sign-changes-inf ps
      unfolding sign-changes-inf-def .
  from length-remdups-adj[of poly-neg-inf s'', OF signs(3) ‹s'' ≠ 0›]
      show sign-changes-neg-inf ps' = sign-changes-neg-inf ps
      unfolding sign-changes-neg-inf-def .
qed
```

2.6 Root-counting functions

With all these results, we can now define functions that count roots in bounded and unbounded intervals:

```
definition count-roots-between where
count-roots-between p a b = (if a ≤ b ∧ p ≠ 0 then
  (let ps = sturm-squarefree p
    in sign-changes ps a − sign-changes ps b) else 0)

definition count-roots where
count-roots p = (if (p::real poly) = 0 then 0 else
  (let ps = sturm-squarefree p
    in sign-changes-neg-inf ps − sign-changes-inf ps))

definition count-roots-above where
count-roots-above p a = (if (p::real poly) = 0 then 0 else
  (let ps = sturm-squarefree p
    in sign-changes ps a − sign-changes-inf ps))

definition count-roots-below where
count-roots-below p a = (if (p::real poly) = 0 then 0 else
  (let ps = sturm-squarefree p
    in sign-changes-neg-inf ps − sign-changes ps a))


lemma count-roots-between-correct:
  count-roots-between p a b = card {x. a < x ∧ x ≤ b ∧ poly p x = 0}
proof (cases p ≠ 0 ∧ a ≤ b)
  case False
```

```
    note False' = this
    hence card {x. a < x ∧ x ≤ b ∧ poly p x = 0} = 0
    proof (cases a < b)
      case True
        with False have [simp]: p = 0 by simp
        have subset: {a<..<b} ⊆ {x. a < x ∧ x ≤ b ∧ poly p x = 0} by auto
        from infinite-Ioo[OF True] have ¬finite {a<..<b} .
        hence ¬finite {x. a < x ∧ x ≤ b ∧ poly p x = 0}
            using finite-subset[OF subset] by blast
        thus ?thesis by simp
    next
      case False
        with False' show ?thesis by (auto simp: not-less card-eq-0-iff)
    qed
    thus ?thesis unfolding count-roots-between-def Let-def using False by auto
next
  case True
  hence p ≠ 0 a ≤ b by simp-all
  define p' where p' = p div (gcd p (pderiv p))
  from poly-div-gcd-squarefree(1)[OF ‹p ≠ 0›] have p' ≠ 0
      unfolding p'-def by clarsimp

  from sturm-seq-sturm-squarefree[OF ‹p ≠ 0›]
      interpret sturm-seq sturm-squarefree p p'
      unfolding p'-def .
  from poly-roots-finite[OF ‹p' ≠ 0›]
      have finite {x. a < x ∧ x ≤ b ∧ poly p' x = 0} by fast
  have count-roots-between p a b = card {x. a < x ∧ x ≤ b ∧ poly p' x = 0}
      unfolding count-roots-between-def Let-def
      using True count-roots-between[OF ‹p' ≠ 0› ‹a ≤ b›] by simp
  also from poly-div-gcd-squarefree(2)[OF ‹p ≠ 0›]
      have {x. a < x ∧ x ≤ b ∧ poly p' x = 0} =
              {x. a < x ∧ x ≤ b ∧ poly p x = 0} unfolding p'-def by blast
  finally show ?thesis .
qed

lemma count-roots-correct:
  fixes p :: real poly
  shows count-roots p = card {x. poly p x = 0} (is - = card ?S)
proof (cases p = 0)
  case True
    with finite-subset[of {0<..<1} ?S]
    have ¬finite {x. poly p x = 0} by (auto simp: infinite-Ioo)
    thus ?thesis by (simp add: count-roots-def True)
next
  case False
  define p' where p' = p div (gcd p (pderiv p))
  from poly-div-gcd-squarefree(1)[OF ‹p ≠ 0›] have p' ≠ 0
      unfolding p'-def by clarsimp
```

from *sturm-seq-sturm-squarefree*[*OF* ‹*p* ≠ *0*›]
    interpret *sturm-seq sturm-squarefree p p*′
    unfolding *p*′*-def* **.**
  from *count-roots*[*OF* ‹*p*′ ≠ *0*›]
    have *count-roots p* = *card* {*x*. *poly p*′ *x* = *0*}
    unfolding *count-roots-def Let-def* **by** (*simp add*: ‹*p* ≠ *0*›)
  also from *poly-div-gcd-squarefree*(*2*)[*OF* ‹*p* ≠ *0*›]
    have {*x*. *poly p*′ *x* = *0*} = {*x*. *poly p x* = *0*} **unfolding** *p*′*-def* **by** *blast*
  finally show *?thesis* **.**
qed

lemma *count-roots-above-correct*:
  fixes *p* :: *real poly*
  shows *count-roots-above p a* = *card* {*x*. *x* > *a* ∧ *poly p x* = *0*}
      (**is** - = *card ?S*)
proof (*cases p* = *0*)
  case *True*
  with *finite-subset*[*of* {*a*<..<*a*+*1*} *?S*]
   have ¬*finite* {*x*. *x* > *a* ∧ *poly p x* = *0*} **by** (*auto simp*: *infinite-Ioo subset-eq*)
  thus *?thesis* **by** (*simp add*: *count-roots-above-def True*)
next
  case *False*
  define *p*′ **where** *p*′ = *p div* (*gcd p* (*pderiv p*))
  from *poly-div-gcd-squarefree*(*1*)[*OF* ‹*p* ≠ *0*›] **have** *p*′ ≠ *0*
    unfolding *p*′*-def* **by** *clarsimp*

  from *sturm-seq-sturm-squarefree*[*OF* ‹*p* ≠ *0*›]
    interpret *sturm-seq sturm-squarefree p p*′
    unfolding *p*′*-def* **.**
  from *count-roots-above*[*OF* ‹*p*′ ≠ *0*›]
    have *count-roots-above p a* = *card* {*x*. *x* > *a* ∧ *poly p*′ *x* = *0*}
    unfolding *count-roots-above-def Let-def* **by** (*simp add*: ‹*p* ≠ *0*›)
  also from *poly-div-gcd-squarefree*(*2*)[*OF* ‹*p* ≠ *0*›]
    have {*x*. *x* > *a* ∧ *poly p*′ *x* = *0*} = {*x*. *x* > *a* ∧ *poly p x* = *0*}
    unfolding *p*′*-def* **by** *blast*
  finally show *?thesis* **.**
qed

lemma *count-roots-below-correct*:
  fixes *p* :: *real poly*
  shows *count-roots-below p a* = *card* {*x*. *x* ≤ *a* ∧ *poly p x* = *0*}
      (**is** - = *card ?S*)
proof (*cases p* = *0*)
  case *True*
   with *finite-subset*[*of* {*a* − *1*<..<*a*} *?S*]
     have ¬*finite* {*x*. *x* ≤ *a* ∧ *poly p x* = *0*} **by** (*auto simp*: *infinite-Ioo subset-eq*)
   thus *?thesis* **by** (*simp add*: *count-roots-below-def True*)
next

case *False*
define p' **where** $p' = p\ div\ (gcd\ p\ (pderiv\ p))$
from *poly-div-gcd-squarefree*(*1*)[*OF* ‹$p \neq 0$›] **have** $p' \neq 0$
unfolding p'*-def* **by** *clarsimp*

from *sturm-seq-sturm-squarefree*[*OF* ‹$p \neq 0$›]
interpret *sturm-seq sturm-squarefree p* p'
unfolding p'*-def* **.**
from *count-roots-below*[*OF* ‹$p' \neq 0$›]
have $count\text{-}roots\text{-}below\ p\ a = card\ \{x.\ x \leq a \wedge poly\ p'\ x = 0\}$
unfolding *count-roots-below-def Let-def* **by** (*simp add*: ‹$p \neq 0$›)
also from *poly-div-gcd-squarefree*(*2*)[*OF* ‹$p \neq 0$›]
have $\{x.\ x \leq a \wedge poly\ p'\ x = 0\} = \{x.\ x \leq a \wedge poly\ p\ x = 0\}$
unfolding p'*-def* **by** *blast*
finally show *?thesis* **.**
qed

The optimisation explained above can be used to prove more efficient code equations that use the more efficient construction in the case that the interval borders are not multiple roots:

lemma *count-roots-between*[*code*]:
count-roots-between p a b =
(*let* $q = pderiv\ p$
in if $a > b \vee p = 0$ *then* *0*
else if $(poly\ p\ a \neq 0 \vee poly\ q\ a \neq 0) \wedge (poly\ p\ b \neq 0 \vee poly\ q\ b \neq 0)$
then (*let* $ps = sturm\ p$
in $sign\text{-}changes\ ps\ a - sign\text{-}changes\ ps\ b$)
else (*let* $ps = sturm\text{-}squarefree\ p$
in $sign\text{-}changes\ ps\ a - sign\text{-}changes\ ps\ b$))
proof (*cases* $a > b \vee p = 0$)
case *True*
thus *?thesis* **by** (*auto simp add*: *count-roots-between-def Let-def*)
next
case *False*
note *False1* = *this*
hence $a \leq b$ $p \neq 0$ **by** *simp-all*
thus *?thesis*
proof (*cases* $(poly\ p\ a \neq 0 \vee poly\ (pderiv\ p)\ a \neq 0) \wedge$
$(poly\ p\ b \neq 0 \vee poly\ (pderiv\ p)\ b \neq 0)$)
case *False*
thus *?thesis* **using** *False1*
by (*auto simp add*: *Let-def count-roots-between-def*)
next
case *True*
hence *A*: $poly\ p\ a \neq 0 \vee poly\ (pderiv\ p)\ a \neq 0$ **and**
B: $poly\ p\ b \neq 0 \vee poly\ (pderiv\ p)\ b \neq 0$ **by** *auto*
define *d* **where** $d = gcd\ p\ (pderiv\ p)$
from ‹$p \neq 0$› **have** [*simp*]: $p\ div\ d \neq 0$
using *poly-div-gcd-squarefree*(*1*)[*OF* ‹$p \neq 0$›] **by** (*auto simp add*: *d-def*)

from *sturm-seq-sturm-squarefree′*[*OF* ‹*p* ≠ *0*›]
interpret *sturm-seq sturm-squarefree′ p p div d*
unfolding *sturm-squarefree′-def Let-def d-def* **.**
note *count-roots-between-correct*
also have {*x*. *a* < *x* ∧ *x* ≤ *b* ∧ *poly p x* = *0*} =
{*x*. *a* < *x* ∧ *x* ≤ *b* ∧ *poly* (*p div d*) *x* = *0*}
unfolding *d-def* **using** *poly-div-gcd-squarefree*(*2*)[*OF* ‹*p* ≠ *0*›] **by** *simp*
also note *count-roots-between*[*OF* ‹*p div d* ≠ *0*› ‹*a* ≤ *b*›, *symmetric*]
also note *sturm-sturm-squarefree′-same-sign-changes*(*1*)[*OF A*]
also note *sturm-sturm-squarefree′-same-sign-changes*(*1*)[*OF B*]
finally show *?thesis* **using** *True False* **by** (*simp add*: *Let-def*)
qed
qed

lemma *count-roots-code*[*code*]:
count-roots (*p*::*real poly*) =
(*if p* = *0 then 0*
else let ps = *sturm p*
in sign-changes-neg-inf ps − *sign-changes-inf ps*)
proof (*cases p* = *0*, *simp add*: *count-roots-def*)
case *False*
define *d* **where** *d* = *gcd p* (*pderiv p*)
from ‹*p* ≠ *0*› **have** [*simp*]: *p div d* ≠ *0*
using *poly-div-gcd-squarefree*(*1*)[*OF* ‹*p* ≠ *0*›] **by** (*auto simp add*: *d-def*)
from *sturm-seq-sturm-squarefree′*[*OF* ‹*p* ≠ *0*›]
interpret *sturm-seq sturm-squarefree′ p p div d*
unfolding *sturm-squarefree′-def Let-def d-def* **.**

note *count-roots-correct*
also have {*x*. *poly p x* = *0*} = {*x*. *poly* (*p div d*) *x* = *0*}
unfolding *d-def* **using** *poly-div-gcd-squarefree*(*2*)[*OF* ‹*p* ≠ *0*›] **by** *simp*
also note *count-roots*[*OF* ‹*p div d* ≠ *0*›, *symmetric*]
also note *sturm-sturm-squarefree′-same-sign-changes*(*2*)[*OF* ‹*p* ≠ *0*›]
also note *sturm-sturm-squarefree′-same-sign-changes*(*3*)[*OF* ‹*p* ≠ *0*›]
finally show *?thesis* **using** *False* **unfolding** *Let-def* **by** *simp*
qed

lemma *count-roots-above-code*[*code*]:
count-roots-above p a =
(*let q* = *pderiv p*
in if p = *0 then 0*
else if poly p a ≠ *0* ∨ *poly q a* ≠ *0*
then (*let ps* = *sturm p*
in sign-changes ps a − *sign-changes-inf ps*)
else (*let ps* = *sturm-squarefree p*
in sign-changes ps a − *sign-changes-inf ps*))
proof (*cases p* = *0*)

```
  case True
    thus ?thesis by (auto simp add: count-roots-above-def Let-def)
next
  case False
    note False1 = this
    hence p ≠ 0 by simp-all
    thus ?thesis
    proof (cases (poly p a ≠ 0 ∨ poly (pderiv p) a ≠ 0))
    case False
      thus ?thesis using False1
          by (auto simp add: Let-def count-roots-above-def)
    next
    case True
      hence A: poly p a ≠ 0 ∨ poly (pderiv p) a ≠ 0 by simp
      define d where d = gcd p (pderiv p)
      from ‹p ≠ 0› have [simp]: p div d ≠ 0
          using poly-div-gcd-squarefree(1)[OF ‹p ≠ 0›] by (auto simp add: d-def)
      from sturm-seq-sturm-squarefree'[OF ‹p ≠ 0›]
          interpret sturm-seq sturm-squarefree' p p div d
          unfolding sturm-squarefree'-def Let-def d-def .
      note count-roots-above-correct
      also have {x. a < x ∧ poly p x = 0} =
                {x. a < x ∧ poly (p div d) x = 0}
          unfolding d-def using poly-div-gcd-squarefree(2)[OF ‹p ≠ 0›] by simp
      also note count-roots-above[OF ‹p div d ≠ 0›, symmetric]
      also note sturm-sturm-squarefree'-same-sign-changes(1)[OF A]
      also note sturm-sturm-squarefree'-same-sign-changes(2)[OF ‹p ≠ 0›]
      finally show ?thesis using True False by (simp add: Let-def)
    qed
qed

lemma count-roots-below-code[code]:
  count-roots-below p a =
     (let q = pderiv p
       in if p = 0 then 0
       else if poly p a ≠ 0 ∨ poly q a ≠ 0
            then (let ps = sturm p
                   in sign-changes-neg-inf ps − sign-changes ps a)
            else (let ps = sturm-squarefree p
                   in sign-changes-neg-inf ps − sign-changes ps a))
proof (cases p = 0)
  case True
    thus ?thesis by (auto simp add: count-roots-below-def Let-def)
next
  case False
    note False1 = this
    hence p ≠ 0 by simp-all
    thus ?thesis
    proof (cases (poly p a ≠ 0 ∨ poly (pderiv p) a ≠ 0))
```

```
    case False
      thus ?thesis using False1
          by (auto simp add: Let-def count-roots-below-def)
    next
    case True
      hence A: poly p a ≠ 0 ∨ poly (pderiv p) a ≠ 0 by simp
      define d where d = gcd p (pderiv p)
      from ‹p ≠ 0› have [simp]: p div d ≠ 0
          using poly-div-gcd-squarefree(1)[OF ‹p ≠ 0›] by (auto simp add: d-def)
      from sturm-seq-sturm-squarefree'[OF ‹p ≠ 0›]
          interpret sturm-seq sturm-squarefree' p p div d
          unfolding sturm-squarefree'-def Let-def d-def .
      note count-roots-below-correct
      also have {x. x ≤ a ∧ poly p x = 0} =
                {x. x ≤ a ∧ poly (p div d) x = 0}
          unfolding d-def using poly-div-gcd-squarefree(2)[OF ‹p ≠ 0›] by simp
      also note count-roots-below[OF ‹p div d ≠ 0›, symmetric]
      also note sturm-sturm-squarefree'-same-sign-changes(1)[OF A]
      also note sturm-sturm-squarefree'-same-sign-changes(3)[OF ‹p ≠ 0›]
      finally show ?thesis using True False by (simp add: Let-def)
    qed
qed

end
```

3 The "sturm" proof method

```
theory Sturm-Method
imports Sturm-Theorem
begin
```

3.1 Preliminary lemmas

In this subsection, we prove lemmas that reduce root counting and related statements to simple, computable expressions using the *count-roots* function family.

```
lemma poly-card-roots-less-leq:
  card {x. a < x ∧ x ≤ b ∧ poly p x = 0} = count-roots-between p a b
  by (simp add: count-roots-between-correct)

lemma poly-card-roots-leq-leq:
  card {x. a ≤ x ∧ x ≤ b ∧ poly p x = 0} =
      ( count-roots-between p a b +
      (if (a ≤ b ∧ poly p a = 0 ∧ p ≠ 0) ∨ (a = b ∧ p = 0) then 1 else 0))
proof (cases (a ≤ b ∧ poly p a = 0 ∧ p ≠ 0) ∨ (a = b ∧ p = 0))
  case False
    note False' = this
    thus ?thesis
```

```
    proof (cases p = 0)
      case False
        with False' have poly p a ≠ 0 ∨ a > b by auto
        hence {x. a ≤ x ∧ x ≤ b ∧ poly p x = 0} =
               {x. a < x ∧ x ≤ b ∧ poly p x = 0}
        by (auto simp: less-eq-real-def)
        thus ?thesis using poly-card-roots-less-leq False'
            by (auto split: if-split-asm)
    next
      case True
        have {x. a ≤ x ∧ x ≤ b} = {a..b}
             {x. a < x ∧ x ≤ b} = {a<..b} by auto
        with True False have card {x. a < x ∧ x ≤ b} = 0 card {x. a ≤ x ∧ x ≤
b} = 0
          by (auto simp add: card-eq-0-iff infinite-Ioc infinite-Icc)
        with True False show ?thesis
            using count-roots-between-correct by simp
    qed
next
  case True
    note True' = this
    have fin: finite {x. a ≤ x ∧ x ≤ b ∧ poly p x = 0}
    proof (cases p = 0)
      case True
        with True' have a = b by simp
        hence {x. a ≤ x ∧ x ≤ b ∧ poly p x = 0} = {b} using True by auto
        thus ?thesis by simp
    next
      case False
        from poly-roots-finite[OF this] show ?thesis by fast
    qed
    with True have {x. a ≤ x ∧ x ≤ b ∧ poly p x = 0} =
        insert a {x. a < x ∧ x ≤ b ∧ poly p x = 0} by auto
    hence card {x. a ≤ x ∧ x ≤ b ∧ poly p x = 0} =
           Suc (card {x. a < x ∧ x ≤ b ∧ poly p x = 0}) using fin by force
    thus ?thesis using True count-roots-between-correct by simp
qed

lemma poly-card-roots-less-less:
  card {x. a < x ∧ x < b ∧ poly p x = 0} =
      ( count-roots-between p a b −
              (if poly p b = 0 ∧ a < b ∧ p ≠ 0 then 1 else 0))
proof (cases poly p b = 0 ∧ a < b ∧ p ≠ 0)
  case False
    note False' = this
    show ?thesis
    proof (cases p = 0)
      case True
        have [simp]: {x. a < x ∧ x < b} = {a<..<b}
```

```
                {x. a < x ∧ x ≤ b} = {a<..b} by auto
      with True False have card {x. a < x ∧ x ≤ b} = 0 card {x. a < x ∧ x <
b} = 0
        by (auto simp add: card-eq-0-iff infinite-Ioo infinite-Ioc)
      with True False′ show ?thesis
          by (auto simp: count-roots-between-correct)
  next
    case False
      with False′ have {x. a < x ∧ x < b ∧ poly p x = 0} =
                      {x. a < x ∧ x ≤ b ∧ poly p x = 0}
        by (auto simp: less-eq-real-def)
    thus ?thesis using poly-card-roots-less-leq False by auto
  qed
next
  case True
  with poly-roots-finite
      have fin: finite {x. a < x ∧ x < b ∧ poly p x = 0} by fast
  from True have {x. a < x ∧ x ≤ b ∧ poly p x = 0} =
      insert b {x. a < x ∧ x < b ∧ poly p x = 0} by auto
  hence Suc (card {x. a < x ∧ x < b ∧ poly p x = 0}) =
          card {x. a < x ∧ x ≤ b ∧ poly p x = 0} using fin by force
  also note count-roots-between-correct[symmetric]
  finally show ?thesis using True by simp
qed

lemma poly-card-roots-leq-less:
  card {x::real. a ≤ x ∧ x < b ∧ poly p x = 0} =
      ( count-roots-between p a b +
      (if p ≠ 0 ∧ a < b ∧ poly p a = 0 then 1 else 0) −
      (if p ≠ 0 ∧ a < b ∧ poly p b = 0 then 1 else 0))
proof (cases p = 0 ∨ a ≥ b)
  case True
    note True′ = this
    show ?thesis
    proof (cases a ≥ b)
      case False
      hence {x. a < x ∧ x ≤ b} = {a<..b}
            {x. a ≤ x ∧ x < b} = {a..<b} by auto
      with True False have card {x. a < x ∧ x ≤ b} = 0 card {x. a ≤ x ∧ x <
b} = 0
        by (auto simp add: card-eq-0-iff infinite-Ico infinite-Ioc)
      with False True′ show ?thesis
          by (simp add: count-roots-between-correct)
  next
    case True
      with True′ have {x. a ≤ x ∧ x < b ∧ poly p x = 0} =
                      {x. a < x ∧ x ≤ b ∧ poly p x = 0}
        by (auto simp: less-eq-real-def)
    thus ?thesis using poly-card-roots-less-leq True by simp
```

qed
next
case *False*
let *?A* = {*x*. *a* ≤ *x* ∧ *x* < *b* ∧ *poly p x* = *0*}
let *?B* = {*x*. *a* < *x* ∧ *x* ≤ *b* ∧ *poly p x* = *0*}
let *?C* = {*x*. *x* = *b* ∧ *poly p x* = *0*}
let *?D* = {*x*. *x* = *a* ∧ *poly p a* = *0*}
have *CD-if*: *?C* = (*if poly p b* = *0 then* {*b*} *else* {})
?D = (*if poly p a* = *0 then* {*a*} *else* {}) **by** *auto*
from *False poly-roots-finite*
have [*simp*]: *finite ?A finite ?B finite ?C finite ?D*
by (*fast*, *fast*, *simp-all*)

from *False* **have** *?A* = (*?B* ∪ *?D*) − *?C* **by** (*auto simp*: *less-eq-real-def*)
with *False* **have** *card ?A* = *card ?B* + (*if poly p a* = *0 then 1 else 0*) −
(*if poly p b* = *0 then 1 else 0*) **by** (*auto simp*: *CD-if*)
also note *count-roots-between-correct*[*symmetric*]
finally show *?thesis* **using** *False* **by** *simp*
qed

lemma *poly-card-roots*:
card {*x*::*real*. *poly p x* = *0*} = *count-roots p*
using *count-roots-correct* **by** *simp*

lemma *poly-no-roots*:
(∀ *x*. *poly p x* ≠ *0*) ⟷ (*p* ≠ *0* ∧ *count-roots p* = *0*)
by (*auto simp*: *count-roots-correct dest*: *poly-roots-finite*)

lemma *poly-pos*:
(∀ *x*. *poly p x* > *0*) ⟷ (
p ≠ *0* ∧ *poly-inf p* = *1* ∧ *count-roots p* = *0*)
by (*simp only*: *Let-def poly-pos poly-no-roots*, *blast*)

lemma *poly-card-roots-greater*:
card {*x*::*real*. *x* > *a* ∧ *poly p x* = *0*} = *count-roots-above p a*
using *count-roots-above-correct* **by** *simp*

lemma *poly-card-roots-leq*:
card {*x*::*real*. *x* ≤ *a* ∧ *poly p x* = *0*} = *count-roots-below p a*
using *count-roots-below-correct* **by** *simp*

lemma *poly-card-roots-geq*:
card {*x*::*real*. *x* ≥ *a* ∧ *poly p x* = *0*} = (
count-roots-above p a + (*if poly p a* = *0* ∧ *p* ≠ *0 then 1 else 0*))
proof (*cases poly p a* = *0* ∧ *p* ≠ *0*)
case *False*
hence *card* {*x*. *x* ≥ *a* ∧ *poly p x* = *0*} = *card* {*x*. *x* > *a* ∧ *poly p x* = *0*}
proof (*cases rule*: *disjE*)

```
      assume p = 0
      have ¬finite {a<..<a+1}
        by (metis infinite-Ioo less-add-one)
      moreover have {a<..<a+1} ⊆ {x. x ≥ a ∧ poly p x = 0}
                    {a<..<a+1} ⊆ {x. x > a ∧ poly p x = 0}
          using ‹p = 0› by auto
      ultimately have ¬finite {x. x ≥ a ∧ poly p x = 0}
                      ¬finite {x. x > a ∧ poly p x = 0}
        by (auto dest!: finite-subset[of {a<..<a+1}] simp: infinite-Ioo)
      thus ?thesis by simp
    next
      assume poly p a ≠ 0
      hence {x. x ≥ a ∧ poly p x = 0} = {x. x > a ∧ poly p x = 0}
          by (auto simp: less-eq-real-def)
      thus ?thesis by simp
    qed auto
    thus ?thesis using False
        by (auto intro: poly-card-roots-greater)
next
  case True
    hence finite {x. x > a ∧ poly p x = 0} using poly-roots-finite by force
    moreover have {x. x ≥ a ∧ poly p x = 0} =
                      insert a {x. x > a ∧ poly p x = 0} using True by auto
    ultimately have card {x. x ≥ a ∧ poly p x = 0} =
                        Suc (card {x. x > a ∧ poly p x = 0})
        using card-insert-disjoint by auto
    thus ?thesis using True by (auto intro: poly-card-roots-greater)
qed

lemma poly-card-roots-less:
  card {x::real. x < a ∧ poly p x = 0} =
      (count-roots-below p a − (if poly p a = 0 ∧ p ≠ 0 then 1 else 0))
proof (cases poly p a = 0 ∧ p ≠ 0)
  case False
    hence card {x. x < a ∧ poly p x = 0} = card {x. x ≤ a ∧ poly p x = 0}
    proof (cases rule: disjE)
      assume p = 0
      have ¬finite {a − 1<..<a}
        by (metis infinite-Ioo diff-add-cancel less-add-one)
      moreover have {a − 1<..<a} ⊆ {x. x ≤ a ∧ poly p x = 0}
                    {a − 1<..<a} ⊆ {x. x < a ∧ poly p x = 0}
          using ‹p = 0› by auto
      ultimately have ¬finite {x. x ≤ a ∧ poly p x = 0}
                     ¬finite {x. x < a ∧ poly p x = 0}
          by (auto dest: finite-subset[of {a − 1<..<a}] simp: infinite-Ioo)
      thus ?thesis by simp
    next
      assume poly p a ≠ 0
      hence {x. x < a ∧ poly p x = 0} = {x. x ≤ a ∧ poly p x = 0}
```

```
        by (auto simp: less-eq-real-def)
      thus ?thesis by simp
    qed auto
    thus ?thesis using False
        by (auto intro: poly-card-roots-leq)
next
  case True
    hence finite {x. x < a ∧ poly p x = 0} using poly-roots-finite by force
    moreover have {x. x ≤ a ∧ poly p x = 0} =
                      insert a {x. x < a ∧ poly p x = 0} using True by auto
    ultimately have Suc (card {x. x < a ∧ poly p x = 0}) =
                    (card {x. x ≤ a ∧ poly p x = 0})
        using card-insert-disjoint by auto
    also note count-roots-below-correct[symmetric]
    finally show ?thesis using True by simp
qed


lemma poly-no-roots-less-leq:
  (∀x. a < x ∧ x ≤ b ⟶ poly p x ≠ 0) ⟷
   ((a ≥ b ∨ (p ≠ 0 ∧ count-roots-between p a b = 0)))
  by (auto simp: count-roots-between-correct card-eq-0-iff not-le
          dest: poly-roots-finite)

lemma poly-pos-between-less-leq:
  (∀x. a < x ∧ x ≤ b ⟶ poly p x > 0) ⟷
   ((a ≥ b ∨ (p ≠ 0 ∧ poly p b > 0 ∧ count-roots-between p a b = 0)))
  by (simp only: poly-pos-between-less-leq Let-def
                poly-no-roots-less-leq, blast)


lemma poly-no-roots-leq-leq:
  (∀x. a ≤ x ∧ x ≤ b ⟶ poly p x ≠ 0) ⟷
   ((a > b ∨ (p ≠ 0 ∧ poly p a ≠ 0 ∧ count-roots-between p a b = 0)))
apply (intro iffI)
apply (force simp add: count-roots-between-correct card-eq-0-iff)
apply (elim conjE disjE, simp, intro allI)
apply (rename-tac x, case-tac x = a)
apply (auto simp add: count-roots-between-correct card-eq-0-iff
            dest: poly-roots-finite)
done

lemma poly-pos-between-leq-leq:
  (∀x. a ≤ x ∧ x ≤ b ⟶ poly p x > 0) ⟷
   ((a > b ∨ (p ≠ 0 ∧ poly p a > 0 ∧
                count-roots-between p a b = 0)))
by (simp only: poly-pos-between-leq-leq Let-def poly-no-roots-leq-leq, force)
```

lemma *poly-no-roots-less-less*:
$(\forall x.\ a < x \wedge x < b \longrightarrow poly\ p\ x \neq 0) \longleftrightarrow$
$((a \geq b \vee p \neq 0 \wedge count\text{-}roots\text{-}between\ p\ a\ b =$
$(if\ poly\ p\ b = 0\ then\ 1\ else\ 0)))$
proof (*standard*, *goal-cases*)
case *A*: *1*
show *?case*
proof (*cases* $a \geq b$)
case *True*
with *A* **show** *?thesis* **by** *simp*
next
case *False*
with *A* **have** [*simp*]: $p \neq 0$ **using** *dense*[*of a b*] **by** *auto*
have *B*: $\{x.\ a < x \wedge x \leq b \wedge poly\ p\ x = 0\} =$
$\{x.\ a < x \wedge x < b \wedge poly\ p\ x = 0\} \cup$
$(if\ poly\ p\ b = 0\ then\ \{b\}\ else\ \{\})$ **using** *A False* **by** *auto*
have $count\text{-}roots\text{-}between\ p\ a\ b =$
$card\ \{x.\ a < x \wedge x < b \wedge poly\ p\ x = 0\} +$
$(if\ poly\ p\ b = 0\ then\ 1\ else\ 0)$
by (*subst count-roots-between-correct*, *subst B*, *subst card-Un-disjoint*,
rule finite-subset[*OF - poly-roots-finite*], *blast*, *simp-all*)
also from *A* **have** $\{x.\ a < x \wedge x < b \wedge poly\ p\ x = 0\} = \{\}$ **by** *simp*
finally show *?thesis* **by** *auto*
qed
next
case *prems*: *2*
hence $card\ \{x.\ a < x \wedge x < b \wedge poly\ p\ x = 0\} = 0$
by (*subst poly-card-roots-less-less*, *auto simp*: *count-roots-between-def*)
thus *?case* **using** *prems*
by (*cases* $p = 0$, *simp*, *subst* (*asm*) *card-eq-0-iff*,
auto dest: *poly-roots-finite*)
qed

lemma *poly-pos-between-less-less*:
$(\forall x.\ a < x \wedge x < b \longrightarrow poly\ p\ x > 0) \longleftrightarrow$
$((a \geq b \vee (p \neq 0 \wedge poly\ p\ ((a+b)/2) > 0 \wedge$
$count\text{-}roots\text{-}between\ p\ a\ b = (if\ poly\ p\ b = 0\ then\ 1\ else\ 0))))$
by (*simp only*: *poly-pos-between-less-less Let-def*
poly-no-roots-less-less, *blast*)

lemma *poly-no-roots-leq-less*:
$(\forall x.\ a \leq x \wedge x < b \longrightarrow poly\ p\ x \neq 0) \longleftrightarrow$
$((a \geq b \vee p \neq 0 \wedge poly\ p\ a \neq 0 \wedge count\text{-}roots\text{-}between\ p\ a\ b =$
$(if\ a < b \wedge poly\ p\ b = 0\ then\ 1\ else\ 0)))$
proof (*standard*, *goal-cases*)
case *prems*: *1*
hence $\forall x.\ a < x \wedge x < b \longrightarrow poly\ p\ x \neq 0$ **by** *simp*
thus *?case* **using** *prems* **by** (*subst* (*asm*) *poly-no-roots-less-less*, *auto*)

next
  case *prems*: *2*
    hence $(b \leq a \vee p \neq 0 \wedge$ *count-roots-between* $p\ a\ b =$
        (*if poly* $p\ b = 0$ *then 1 else 0*)) **by** *auto*
    thus *?case* **using** *prems* **unfolding** *Let-def*
      by (*subst* (*asm*) *poly-no-roots-less-less*[*symmetric*, *unfolded Let-def*],
      auto split: *if-split-asm simp*: *less-eq-real-def*)
qed

lemma *poly-pos-between-leq-less*:
  $(\forall x.\ a \leq x \wedge x < b \longrightarrow$ *poly* $p\ x > 0) \longleftrightarrow$
   $((a \geq b \vee (p \neq 0 \wedge$ *poly* $p\ a > 0 \wedge$ *count-roots-between* $p\ a\ b =$
     (*if* $a < b \wedge$ *poly* $p\ b = 0$ *then 1 else 0*))))
 by (*simp only*: *poly-pos-between-leq-less Let-def*
     poly-no-roots-leq-less, *force*)

lemma *poly-no-roots-greater*:
  $(\forall x.\ x > a \longrightarrow$ *poly* $p\ x \neq 0) \longleftrightarrow$
    $((p \neq 0 \wedge$ *count-roots-above* $p\ a = 0))$
proof−
  have $\forall x.\ \neg\ a < x \Longrightarrow$ *False* **by** (*metis gt-ex*)
  thus *?thesis* **by** (*auto simp*: *count-roots-above-correct card-eq-0-iff*
      intro: *poly-roots-finite*)
qed

lemma *poly-pos-greater*:
  $(\forall x.\ x > a \longrightarrow$ *poly* $p\ x > 0) \longleftrightarrow$ (
    $p \neq 0 \wedge$ *poly-inf* $p = 1 \wedge$ *count-roots-above* $p\ a = 0)$
  unfolding *Let-def*
  by (*subst poly-pos-greater*, *subst poly-no-roots-greater*, *force*)

lemma *poly-no-roots-leq*:
  $(\forall x.\ x \leq a \longrightarrow$ *poly* $p\ x \neq 0) \longleftrightarrow$
    $(\ (p \neq 0 \wedge$ *count-roots-below* $p\ a = 0))$
   by (*auto simp*: *Let-def count-roots-below-correct card-eq-0-iff*
     intro: *poly-roots-finite*)

lemma *poly-pos-leq*:
  $(\forall x.\ x \leq a \longrightarrow$ *poly* $p\ x > 0) \longleftrightarrow$
   $(\ p \neq 0 \wedge$ *poly-neg-inf* $p = 1 \wedge$ *count-roots-below* $p\ a = 0)$
  by (*simp only*: *poly-pos-leq Let-def poly-no-roots-leq*, *blast*)

lemma *poly-no-roots-geq*:
  $(\forall x.\ x \geq a \longrightarrow$ *poly* $p\ x \neq 0) \longleftrightarrow$
    $(\ (p \neq 0 \wedge$ *poly* $p\ a \neq 0 \wedge$ *count-roots-above* $p\ a = 0))$
proof (*standard*, *goal-cases*)

case *prems*: *1*
hence $\forall x{>}a.\ poly\ p\ x \neq 0$ **by** *simp*
thus *?case* **using** *prems* **by** (*subst* (*asm*) *poly-no-roots-greater*, *auto*)
next
case *prems*: *2*
hence ($p \neq 0 \wedge$ *count-roots-above* $p\ a = 0$) **by** *simp*
thus *?case* **using** *prems*
by (*subst* (*asm*) *poly-no-roots-greater*[*symmetric*, *unfolded Let-def*],
auto simp: *less-eq-real-def*)
qed

lemma *poly-pos-geq*:
($\forall x.\ x \geq a \longrightarrow poly\ p\ x > 0) \longleftrightarrow$
($p \neq 0 \wedge$ *poly-inf* $p = 1 \wedge poly\ p\ a \neq 0 \wedge$ *count-roots-above* $p\ a = 0$)
by (*simp only*: *poly-pos-geq Let-def poly-no-roots-geq*, *blast*)

lemma *poly-no-roots-less*:
($\forall x.\ x < a \longrightarrow poly\ p\ x \neq 0) \longleftrightarrow$
(($p \neq 0 \wedge$ *count-roots-below* $p\ a =$ (*if poly* $p\ a = 0$ *then* 1 *else* 0)))
proof (*standard*, *goal-cases*)
case *prems*: *1*
hence $\{x.\ x \leq a \wedge poly\ p\ x = 0\}$ = (*if poly* $p\ a = 0$ *then* $\{a\}$ *else* $\{\}$)
by (*auto simp*: *less-eq-real-def*)
moreover have $\forall x.\ \neg\ x < a \Longrightarrow False$ **by** (*metis lt-ex*)
ultimately show *?case* **using** *prems* **by** (*auto simp*: *count-roots-below-correct*)
next
case *prems*: *2*
have *A*: $\{x.\ x \leq a \wedge poly\ p\ x = 0\} = \{x.\ x < a \wedge poly\ p\ x = 0\} \cup$
(*if poly* $p\ a = 0$ *then* $\{a\}$ *else* $\{\}$) **by** (*auto simp*: *less-eq-real-def*)
have *count-roots-below* $p\ a$ = *card* $\{x.\ x < a \wedge poly\ p\ x = 0\}$ +
(*if poly* $p\ a = 0$ *then* 1 *else* 0) **using** *prems*
by (*subst count-roots-below-correct*, *subst A*, *subst card-Un-disjoint*,
auto intro: *poly-roots-finite*)
with *prems* **have** *card* $\{x.\ x < a \wedge poly\ p\ x = 0\} = 0$ **by** *simp*
thus *?case* **using** *prems*
by (*subst* (*asm*) *card-eq-0-iff*, *auto intro*: *poly-roots-finite*)
qed

lemma *poly-pos-less*:
($\forall x.\ x < a \longrightarrow poly\ p\ x > 0) \longleftrightarrow$
($p \neq 0 \wedge$ *poly-neg-inf* $p = 1 \wedge$ *count-roots-below* $p\ a$ =
(*if poly* $p\ a = 0$ *then* 1 *else* 0))
by (*simp only*: *poly-pos-less Let-def poly-no-roots-less*, *blast*)

lemmas *sturm-card-substs* = *poly-card-roots poly-card-roots-less-leq*
poly-card-roots-leq-less poly-card-roots-less-less poly-card-roots-leq-leq
poly-card-roots-less poly-card-roots-leq poly-card-roots-greater
poly-card-roots-geq

lemmas *sturm-prop-substs* = *poly-no-roots poly-no-roots-less-leq poly-no-roots-leq-leq poly-no-roots-less-less poly-no-roots-leq-less poly-no-roots-leq poly-no-roots-less poly-no-roots-geq poly-no-roots-greater poly-pos poly-pos-greater poly-pos-geq poly-pos-less poly-pos-leq poly-pos-between-leq-less poly-pos-between-less-leq poly-pos-between-leq-leq poly-pos-between-less-less*

3.2 Reification

This subsection defines a number of equations to automatically convert statements about roots of polynomials into a canonical form so that they can be proven using the above substitutions.

definition $PR\text{-}TAG\ x \equiv x$

lemma *sturm-id-PR-prio0*:
$\{x::real.\ P\ x\} = \{x::real.\ (PR\text{-}TAG\ P)\ x\}$
$(\forall x::real.\ f\ x < g\ x) = (\forall x::real.\ PR\text{-}TAG\ (\lambda x.\ f\ x < g\ x)\ x)$
$(\forall x::real.\ P\ x) = (\forall x::real.\ \neg(PR\text{-}TAG\ (\lambda x.\ \neg P\ x))\ x)$
by (*simp-all add*: *PR-TAG-def*)

lemma *sturm-id-PR-prio1*:
$\{x::real.\ x < a \wedge P\ x\} = \{x::real.\ x < a \wedge (PR\text{-}TAG\ P)\ x\}$
$\{x::real.\ x \leq a \wedge P\ x\} = \{x::real.\ x \leq a \wedge (PR\text{-}TAG\ P)\ x\}$
$\{x::real.\ x \geq b \wedge P\ x\} = \{x::real.\ x \geq b \wedge (PR\text{-}TAG\ P)\ x\}$
$\{x::real.\ x > b \wedge P\ x\} = \{x::real.\ x > b \wedge (PR\text{-}TAG\ P)\ x\}$
$(\forall x::real < a.\ f\ x < g\ x) = (\forall x::real < a.\ PR\text{-}TAG\ (\lambda x.\ f\ x < g\ x)\ x)$
$(\forall x::real \leq a.\ f\ x < g\ x) = (\forall x::real \leq a.\ PR\text{-}TAG\ (\lambda x.\ f\ x < g\ x)\ x)$
$(\forall x::real > a.\ f\ x < g\ x) = (\forall x::real > a.\ PR\text{-}TAG\ (\lambda x.\ f\ x < g\ x)\ x)$
$(\forall x::real \geq a.\ f\ x < g\ x) = (\forall x::real \geq a.\ PR\text{-}TAG\ (\lambda x.\ f\ x < g\ x)\ x)$
$(\forall x::real < a.\ P\ x) = (\forall x::real < a.\ \neg(PR\text{-}TAG\ (\lambda x.\ \neg P\ x))\ x)$
$(\forall x::real > a.\ P\ x) = (\forall x::real > a.\ \neg(PR\text{-}TAG\ (\lambda x.\ \neg P\ x))\ x)$
$(\forall x::real \leq a.\ P\ x) = (\forall x::real \leq a.\ \neg(PR\text{-}TAG\ (\lambda x.\ \neg P\ x))\ x)$
$(\forall x::real \geq a.\ P\ x) = (\forall x::real \geq a.\ \neg(PR\text{-}TAG\ (\lambda x.\ \neg P\ x))\ x)$
by (*simp-all add*: *PR-TAG-def*)

lemma *sturm-id-PR-prio2*:
$\{x::real.\ x > a \wedge x \leq b \wedge P\ x\} =$
$\{x::real.\ x > a \wedge x \leq b \wedge PR\text{-}TAG\ P\ x\}$
$\{x::real.\ x \geq a \wedge x \leq b \wedge P\ x\} =$
$\{x::real.\ x \geq a \wedge x \leq b \wedge PR\text{-}TAG\ P\ x\}$
$\{x::real.\ x \geq a \wedge x < b \wedge P\ x\} =$
$\{x::real.\ x \geq a \wedge x < b \wedge PR\text{-}TAG\ P\ x\}$
$\{x::real.\ x > a \wedge x < b \wedge P\ x\} =$
$\{x::real.\ x > a \wedge x < b \wedge PR\text{-}TAG\ P\ x\}$
$(\forall x::real.\ a < x \wedge x \leq b \longrightarrow f\ x < g\ x) =$
$(\forall x::real.\ a < x \wedge x \leq b \longrightarrow PR\text{-}TAG\ (\lambda x.\ f\ x < g\ x)\ x)$
$(\forall x::real.\ a \leq x \wedge x \leq b \longrightarrow f\ x < g\ x) =$

$(\forall x::real.\ a \le x \wedge x \le b \longrightarrow \mathit{PR\text{-}TAG}\ (\lambda x.\ f\,x < g\ x)\ x)$
$(\forall x::real.\ a < x \wedge x < b \longrightarrow f\,x < g\ x) =$
$(\forall x::real.\ a < x \wedge x < b \longrightarrow \mathit{PR\text{-}TAG}\ (\lambda x.\ f\,x < g\ x)\ x)$
$(\forall x::real.\ a \le x \wedge x < b \longrightarrow f\,x < g\ x) =$
$(\forall x::real.\ a \le x \wedge x < b \longrightarrow \mathit{PR\text{-}TAG}\ (\lambda x.\ f\,x < g\ x)\ x)$
$(\forall x::real.\ a < x \wedge x \le b \longrightarrow P\ x) =$
$(\forall x::real.\ a < x \wedge x \le b \longrightarrow \neg(\mathit{PR\text{-}TAG}\ (\lambda x.\ \neg P\ x))\ x)$
$(\forall x::real.\ a \le x \wedge x \le b \longrightarrow P\ x) =$
$(\forall x::real.\ a \le x \wedge x \le b \longrightarrow \neg(\mathit{PR\text{-}TAG}\ (\lambda x.\ \neg P\ x))\ x)$
$(\forall x::real.\ a \le x \wedge x < b \longrightarrow P\ x) =$
$(\forall x::real.\ a \le x \wedge x < b \longrightarrow \neg(\mathit{PR\text{-}TAG}\ (\lambda x.\ \neg P\ x))\ x)$
$(\forall x::real.\ a < x \wedge x < b \longrightarrow P\ x) =$
$(\forall x::real.\ a < x \wedge x < b \longrightarrow \neg(\mathit{PR\text{-}TAG}\ (\lambda x.\ \neg P\ x))\ x)$
by (*simp-all add*: *PR-TAG-def*)

lemma *PR-TAG-intro-prio0*:
fixes $P :: real \Rightarrow bool$ **and** $f :: real \Rightarrow real$
shows
$\mathit{PR\text{-}TAG}\ P = P' \Longrightarrow \mathit{PR\text{-}TAG}\ (\lambda x.\ \neg(\neg P\ x)) = P'$
$[\![\mathit{PR\text{-}TAG}\ P = (\lambda x.\ poly\ p\ x = 0);\ \mathit{PR\text{-}TAG}\ Q = (\lambda x.\ poly\ q\ x = 0)]\!]$
$\Longrightarrow \mathit{PR\text{-}TAG}\ (\lambda x.\ P\ x \wedge Q\ x) = (\lambda x.\ poly\ (gcd\ p\ q)\ x = 0)$ **and**
$[\![\mathit{PR\text{-}TAG}\ P = (\lambda x.\ poly\ p\ x = 0);\ \mathit{PR\text{-}TAG}\ Q = (\lambda x.\ poly\ q\ x = 0)]\!]$
$\Longrightarrow \mathit{PR\text{-}TAG}\ (\lambda x.\ P\ x \vee Q\ x) = (\lambda x.\ poly\ (p*q)\ x = 0)$ **and**

$[\![\mathit{PR\text{-}TAG}\ f = (\lambda x.\ poly\ p\ x);\ \mathit{PR\text{-}TAG}\ g = (\lambda x.\ poly\ q\ x)]\!]$
$\Longrightarrow \mathit{PR\text{-}TAG}\ (\lambda x.\ f\,x = g\ x) = (\lambda x.\ poly\ (p-q)\ x = 0)$
$[\![\mathit{PR\text{-}TAG}\ f = (\lambda x.\ poly\ p\ x);\ \mathit{PR\text{-}TAG}\ g = (\lambda x.\ poly\ q\ x)]\!]$
$\Longrightarrow \mathit{PR\text{-}TAG}\ (\lambda x.\ f\,x \ne g\ x) = (\lambda x.\ poly\ (p-q)\ x \ne 0)$
$[\![\mathit{PR\text{-}TAG}\ f = (\lambda x.\ poly\ p\ x);\ \mathit{PR\text{-}TAG}\ g = (\lambda x.\ poly\ q\ x)]\!]$
$\Longrightarrow \mathit{PR\text{-}TAG}\ (\lambda x.\ f\,x < g\ x) = (\lambda x.\ poly\ (q-p)\ x > 0)$
$[\![\mathit{PR\text{-}TAG}\ f = (\lambda x.\ poly\ p\ x);\ \mathit{PR\text{-}TAG}\ g = (\lambda x.\ poly\ q\ x)]\!]$
$\Longrightarrow \mathit{PR\text{-}TAG}\ (\lambda x.\ f\,x \le g\ x) = (\lambda x.\ poly\ (q-p)\ x \ge 0)$

$\mathit{PR\text{-}TAG}\ f = (\lambda x.\ poly\ p\ x) \Longrightarrow \mathit{PR\text{-}TAG}\ (\lambda x.\ -f\,x) = (\lambda x.\ poly\ (-p)\ x)$
$[\![\mathit{PR\text{-}TAG}\ f = (\lambda x.\ poly\ p\ x);\ \mathit{PR\text{-}TAG}\ g = (\lambda x.\ poly\ q\ x)]\!]$
$\Longrightarrow \mathit{PR\text{-}TAG}\ (\lambda x.\ f\,x + g\ x) = (\lambda x.\ poly\ (p+q)\ x)$
$[\![\mathit{PR\text{-}TAG}\ f = (\lambda x.\ poly\ p\ x);\ \mathit{PR\text{-}TAG}\ g = (\lambda x.\ poly\ q\ x)]\!]$
$\Longrightarrow \mathit{PR\text{-}TAG}\ (\lambda x.\ f\,x - g\ x) = (\lambda x.\ poly\ (p-q)\ x)$
$[\![\mathit{PR\text{-}TAG}\ f = (\lambda x.\ poly\ p\ x);\ \mathit{PR\text{-}TAG}\ g = (\lambda x.\ poly\ q\ x)]\!]$
$\Longrightarrow \mathit{PR\text{-}TAG}\ (\lambda x.\ f\,x * g\ x) = (\lambda x.\ poly\ (p*q)\ x)$
$\mathit{PR\text{-}TAG}\ f = (\lambda x.\ poly\ p\ x) \Longrightarrow \mathit{PR\text{-}TAG}\ (\lambda x.\ (f\ x)\hat{}n) = (\lambda x.\ poly\ (p\hat{}n)\ x)$
$\mathit{PR\text{-}TAG}\ (\lambda x.\ poly\ p\ x :: real) = (\lambda x.\ poly\ p\ x)$
$\mathit{PR\text{-}TAG}\ (\lambda x.\ x::real) = (\lambda x.\ poly\ [:0,1:]\ x)$
$\mathit{PR\text{-}TAG}\ (\lambda x.\ a::real) = (\lambda x.\ poly\ [:a:]\ x)$
by (*simp-all add*: *PR-TAG-def poly-eq-0-iff-dvd field-simps*)

lemma *PR-TAG-intro-prio1*:
fixes $f :: real \Rightarrow real$
shows
$PR\text{-}TAG\ f = (\lambda x.\ poly\ p\ x) \Longrightarrow PR\text{-}TAG\ (\lambda x.\ f\ x = 0) = (\lambda x.\ poly\ p\ x = 0)$
$PR\text{-}TAG\ f = (\lambda x.\ poly\ p\ x) \Longrightarrow PR\text{-}TAG\ (\lambda x.\ f\ x \neq 0) = (\lambda x.\ poly\ p\ x \neq 0)$
$PR\text{-}TAG\ f = (\lambda x.\ poly\ p\ x) \Longrightarrow PR\text{-}TAG\ (\lambda x.\ 0 = f\ x) = (\lambda x.\ poly\ p\ x = 0)$
$PR\text{-}TAG\ f = (\lambda x.\ poly\ p\ x) \Longrightarrow PR\text{-}TAG\ (\lambda x.\ 0 \neq f\ x) = (\lambda x.\ poly\ p\ x \neq 0)$
$PR\text{-}TAG\ f = (\lambda x.\ poly\ p\ x) \Longrightarrow PR\text{-}TAG\ (\lambda x.\ f\ x \geq 0) = (\lambda x.\ poly\ p\ x \geq 0)$
$PR\text{-}TAG\ f = (\lambda x.\ poly\ p\ x) \Longrightarrow PR\text{-}TAG\ (\lambda x.\ f\ x > 0) = (\lambda x.\ poly\ p\ x > 0)$
$PR\text{-}TAG\ f = (\lambda x.\ poly\ p\ x) \Longrightarrow PR\text{-}TAG\ (\lambda x.\ f\ x \leq 0) = (\lambda x.\ poly\ (-p)\ x \geq 0)$
$PR\text{-}TAG\ f = (\lambda x.\ poly\ p\ x) \Longrightarrow PR\text{-}TAG\ (\lambda x.\ f\ x < 0) = (\lambda x.\ poly\ (-p)\ x > 0)$
$PR\text{-}TAG\ f = (\lambda x.\ poly\ p\ x) \Longrightarrow$
$PR\text{-}TAG\ (\lambda x.\ 0 \leq f\ x) = (\lambda x.\ poly\ (-p)\ x \leq 0)$
$PR\text{-}TAG\ f = (\lambda x.\ poly\ p\ x) \Longrightarrow$
$PR\text{-}TAG\ (\lambda x.\ 0 < f\ x) = (\lambda x.\ poly\ (-p)\ x < 0)$
$PR\text{-}TAG\ f = (\lambda x.\ poly\ p\ x)$
$\Longrightarrow PR\text{-}TAG\ (\lambda x.\ a * f\ x) = (\lambda x.\ poly\ (smult\ a\ p)\ x)$
$PR\text{-}TAG\ f = (\lambda x.\ poly\ p\ x)$
$\Longrightarrow PR\text{-}TAG\ (\lambda x.\ f\ x * a) = (\lambda x.\ poly\ (smult\ a\ p)\ x)$
$PR\text{-}TAG\ f = (\lambda x.\ poly\ p\ x)$
$\Longrightarrow PR\text{-}TAG\ (\lambda x.\ f\ x\ /\ a) = (\lambda x.\ poly\ (smult\ (inverse\ a)\ p)\ x)$
$PR\text{-}TAG\ (\lambda x.\ x\hat{\ }n :: real) = (\lambda x.\ poly\ (monom\ 1\ n)\ x)$
by (*simp-all add*: *PR-TAG-def field-simps poly-monom*)

lemma *PR-TAG-intro-prio2*:
$PR\text{-}TAG\ (\lambda x.\ 1\ /\ b) = (\lambda x.\ inverse\ b)$
$PR\text{-}TAG\ (\lambda x.\ a\ /\ b) = (\lambda x.\ a\ /\ b)$
$PR\text{-}TAG\ (\lambda x.\ a\ /\ b * x\hat{\ }n :: real) = (\lambda x.\ poly\ (monom\ (a/b)\ n)\ x)$
$PR\text{-}TAG\ (\lambda x.\ x\hat{\ }n * a\ /\ b :: real) = (\lambda x.\ poly\ (monom\ (a/b)\ n)\ x)$
$PR\text{-}TAG\ (\lambda x.\ a * x\hat{\ }n :: real) = (\lambda x.\ poly\ (monom\ a\ n)\ x)$
$PR\text{-}TAG\ (\lambda x.\ x\hat{\ }n * a :: real) = (\lambda x.\ poly\ (monom\ a\ n)\ x)$
$PR\text{-}TAG\ (\lambda x.\ x\hat{\ }n\ /\ a :: real) = (\lambda x.\ poly\ (monom\ (inverse\ a)\ n)\ x)$

$PR\text{-}TAG\ (\lambda x.\ f\ x\hat{\ }(Suc\ (Suc\ 0)) :: real) = (\lambda x.\ poly\ p\ x)$
$\Longrightarrow PR\text{-}TAG\ (\lambda x.\ f\ x * f\ x :: real) = (\lambda x.\ poly\ p\ x)$
$PR\text{-}TAG\ (\lambda x.\ (f\ x)\hat{\ }Suc\ n :: real) = (\lambda x.\ poly\ p\ x)$
$\Longrightarrow PR\text{-}TAG\ (\lambda x.\ (f\ x)\hat{\ }n * f\ x :: real) = (\lambda x.\ poly\ p\ x)$
$PR\text{-}TAG\ (\lambda x.\ (f\ x)\hat{\ }Suc\ n :: real) = (\lambda x.\ poly\ p\ x)$
$\Longrightarrow PR\text{-}TAG\ (\lambda x.\ f\ x * (f\ x)\hat{\ }n :: real) = (\lambda x.\ poly\ p\ x)$
$PR\text{-}TAG\ (\lambda x.\ (f\ x)\hat{\ }(m+n) :: real) = (\lambda x.\ poly\ p\ x)$
$\Longrightarrow PR\text{-}TAG\ (\lambda x.\ (f\ x)\hat{\ }m * (f\ x)\hat{\ }n :: real) = (\lambda x.\ poly\ p\ x)$
by (*simp-all add*: *PR-TAG-def field-simps poly-monom power-add*)

lemma *sturm-meta-spec*: $(\bigwedge x::real.\ P\ x) \Longrightarrow P\ x$ **by** *simp*
lemma *sturm-imp-conv*:
$(a < x \longrightarrow x < b \longrightarrow c) \longleftrightarrow (a < x \wedge x < b \longrightarrow c)$
$(a \leq x \longrightarrow x < b \longrightarrow c) \longleftrightarrow (a \leq x \wedge x < b \longrightarrow c)$
$(a < x \longrightarrow x \leq b \longrightarrow c) \longleftrightarrow (a < x \wedge x \leq b \longrightarrow c)$
$(a \leq x \longrightarrow x \leq b \longrightarrow c) \longleftrightarrow (a \leq x \wedge x \leq b \longrightarrow c)$

$(x < b \longrightarrow a < x \longrightarrow c) \longleftrightarrow (a < x \wedge x < b \longrightarrow c)$
$(x < b \longrightarrow a \leq x \longrightarrow c) \longleftrightarrow (a \leq x \wedge x < b \longrightarrow c)$
$(x \leq b \longrightarrow a < x \longrightarrow c) \longleftrightarrow (a < x \wedge x \leq b \longrightarrow c)$
$(x \leq b \longrightarrow a \leq x \longrightarrow c) \longleftrightarrow (a \leq x \wedge x \leq b \longrightarrow c)$
by *auto*

3.3 Setup for the "sturm" method

ML-file ‹*sturm.ML*›

method-setup *sturm* = ‹
Scan.succeed (*fn ctxt* => *SIMPLE-METHOD′* (*Sturm.sturm-tac ctxt true*))
›

end

theory *Sturm*
imports *Sturm-Method*
begin

end

4 Example usage of the "sturm" method

theory *Sturm-Ex*
imports ../*Sturm*
begin

In this section, we give a variety of statements about real polynomials that can b proven by the *sturm* method.

lemma
$\forall x{::}real.\ x\hat{}2 + 1 \neq 0$
by *sturm*

lemma
fixes $x :: real$
shows $x\hat{}2 + 1 \neq 0$ **by** *sturm*

lemma $(x{::}real) > 1 \Longrightarrow x\hat{}3 > 1$ **by** *sturm*

lemma $\forall x{::}real.\ x{*}x \neq -1$ **by** *sturm*

schematic-goal *A*:
$card\ \{x{::}real.\ -0.010831 < x \wedge x < 0.010831 \wedge$
$1/120{*}x\hat{}5 + 1/24{*}x\hat{}4 + 1/6{*}x\hat{}3 - 49/16777216{*}x\hat{}2 - 17/2097152{*}x =$
$0\}$
$= ?n$
by *sturm*

lemma *card* {*x*::*real*. *x*^*3* + *x* = *2*∗*x*^*2* ∧ *x*^*3* − *6*∗*x*^*2* + *11*∗*x* = *6*} = *1*
by *sturm*

schematic-goal *card* {*x*::*real*. *x*^*3* + *x* = *2*∗*x*^*2* ∨ *x*^*3* − *6*∗*x*^*2* + *11*∗*x* = *6*} = *?n* **by** *sturm*

lemma
card {*x*::*real*. −*0.010831* < *x* ∧ *x* < *0.010831* ∧
poly [:*0*, −*17/2097152*, −*49/16777216*, *1/6*, *1/24*, *1/120*:] *x* = *0*} = *3*
by *sturm*

lemma ∀ *x*::*real*. *x*∗*x* ≠ *0* ∨ *x*∗*x* − *1* ≠ *2*∗*x* **by** *sturm*

lemma (*x*::*real*)∗*x*+*1* ≠ *0* ∧ (*x*^*2*+*1*)∗(*x*^*2*+*2*) ≠ *0* **by** *sturm*

3 examples related to continued fraction approximants to exp: LCP

lemma fixes *x*::*real*
shows −*7.29347719* ≤ *x* ⟹ *0* < *x*^*5* + *30*∗*x*^*4* + *420*∗*x*^*3* + *3360*∗*x*^*2* + *15120*∗*x* + *30240*
by *sturm*

lemma fixes *x*::*real*
shows *0* < *x*^*6* + *42*∗*x*^*5* + *840*∗*x*^*4* + *10080*∗*x*^*3* + *75600*∗x^2 + *332640*∗*x* + *665280*
by *sturm*

schematic-goal *card* {*x*::*real*. *x*^*7* + *56*∗*x*^*6* + *1512*∗*x*^*5* + *25200*∗*x*^*4* + *277200*∗*x*^*3* + *1995840*∗*x*^*2* + *8648640*∗*x* = −*17297280*} = *?n*
by *sturm*

end