Vosper's Theorem and the Cauchy–Davenport Theorem via the Polynomial Method

Arthur F. Ramos David Barros Hulak Ruy J. G. B. de Queiroz

June 12, 2026

Abstract

This entry formalizes the Cauchy–Davenport lower bound and Vosper's theorem for sumsets over finite fields of prime cardinality. It combines a polynomial-method proof of Cauchy–Davenport with a Davenport-transform proof of Vosper's classification of the nontrivial equality case, and it provides modular-representative lemmas connecting the abstract prime-field results to normalized representatives of $\mathbb{Z}/p\mathbb{Z}$. AI assistance was used for proof engineering. The final definitions, statements, and proofs are checked by Isabelle.

1 Overview

For nonempty finite subsets A and B of a prime field $\mathbb{F}_p$, the Cauchy–Davenport theorem gives the lower bound

$$|A + B| \geq \min(p, |A| + |B| - 1).$$

Vosper's theorem refines the extremal case: if $|A|, |B| \geq 2$, the sumset is not all of $\mathbb{F}_p$, and equality holds in the Cauchy–Davenport bound, then both sets are arithmetic progressions with a common difference. These results are standard cornerstones of additive combinatorics.

2 Formalization

The Isabelle/HOL development first establishes generic finite-sumset facts for translations, complements, and cardinalities. The proof of Cauchy–Davenport is then carried out by the polynomial method, using ordinary Isabelle polynomials to package the bivariate degree argument over finite fields of prime cardinality. Vosper's theorem is proved via Davenport transforms: the formalization derives one-sided progression results under normalization hypotheses and combines them to obtain the final shared-difference classification. The entry also supplies concrete modular-translation facts over normalized representatives of $\mathbb{Z}/p\mathbb{Z}$.

3 Relation to existing AFP entries

The Cauchy–Davenport bound has been formalized in the AFP before. The entry *Kneser's Theorem and the Cauchy–Davenport Theorem* [3] derives it as a corollary of Kneser's addition theorem for abelian groups, and *A Generalization of the Cauchy–Davenport Theorem* [2] formalizes DeVos's generalization to (possibly non-abelian) groups. Both developments are set in an abstract group structure and build on further AFP material.

The present entry is intended to complement, not duplicate, this work. Its principal new contribution is *Vosper's theorem*, the critical-pair (inverse) theorem characterizing the equality case of the Cauchy–Davenport bound, which neither of the entries above provides. The Cauchy–Davenport bound itself is established here by a different route, the *polynomial method*

(Combinatorial Nullstellensatz) following Alon, Nathanson, and Ruzsa [1]: a single explicit polynomial is shown to be non-vanishing on a grid built from A and B. To the best of our knowledge this proof of Cauchy–Davenport is not otherwise present in the AFP. It is carried out directly for finite fields of prime cardinality, modelled by the `finite_field` type class together with a primality hypothesis on the field's cardinality, and it depends only on the Isabelle/HOL distribution (`HOL-Computational_Algebra` and `HOL-Number_Theory`). Keeping this self-contained prime-field bound as the base case lets the Vosper development proceed without translating between the type-class and locale-based group representations.

4 Sources

The Cauchy–Davenport proof follows the polynomial-method perspective of Alon, Nathanson, and Ruzsa [1]. The additive-combinatorics background follows standard sources such as Nathanson [4] and Tao and Vu [5]. The inverse theorem formalized here is Vosper's critical-pair theorem for groups of prime order [6].

Contents

theory *Sumset-Basics*
imports *Main*
begin

definition *sumset* :: ('*a*::*comm-monoid-add*) *set* => '*a set* => '*a set* **where**
sumset A B = {*x*. ∃ *a*∈*A*. ∃ *b*∈*B*. *x* = *a* + *b*}

lemma *sumset-iff*:
x ∈ *sumset A B* ⟷ (∃ *a*∈*A*. ∃ *b*∈*B*. *x* = *a* + *b*)
by (*auto simp*: *sumset-def*)

lemma *sumsetI* [*intro*]:

assumes $a \in A$ $b \in B$
shows $a + b \in sumset\ A\ B$
using *assms* **by** (*auto simp*: *sumset-def*)

lemma *sumsetE* [*elim*]:
assumes $x \in sumset\ A\ B$
obtains $a\ b$ **where** $a \in A$ $b \in B$ $x = a + b$
using *assms* **by** (*auto simp*: *sumset-def*)

lemma *sumset-as-image*:
$sumset\ A\ B = case\text{-}prod\ (+)\ `\ (A \times B)$
by (*auto simp*: *sumset-def*)

lemma *sumset-commute*:
$sumset\ A\ B = sumset\ B\ A$
unfolding *sumset-def* **using** *add.commute* **by** *blast*

lemma *empty-sumset-left* [*simp*]:
$sumset\ \{\}\ B = \{\}$
by (*auto simp*: *sumset-def*)

lemma *empty-sumset-right* [*simp*]:
$sumset\ A\ \{\} = \{\}$
by (*auto simp*: *sumset-def*)

lemma *sumset-assoc*:
fixes $A\ B\ C :: ('a::comm\text{-}monoid\text{-}add)\ set$
shows $sumset\ (sumset\ A\ B)\ C = sumset\ A\ (sumset\ B\ C)$
unfolding *sumset-def* **using** *add.assoc* **by** *force*

lemma *sumset-zero-right* [*simp*]:
fixes $A :: ('a::comm\text{-}monoid\text{-}add)\ set$
shows $sumset\ A\ \{0\} = A$
by (*auto simp*: *sumset-def*)

lemma *sumset-zero-left* [*simp*]:
fixes $A :: ('a::comm\text{-}monoid\text{-}add)\ set$
shows $sumset\ \{0\}\ A = A$
by (*auto simp*: *sumset-def*)

definition $translate :: 'a::comm\text{-}monoid\text{-}add \Rightarrow 'a\ set \Rightarrow 'a\ set$ **where**
$translate\ c\ A = ((+)\ c)\ `\ A$

lemma *translate-iff*:
$x \in translate\ c\ A \longleftrightarrow (\exists\, a{\in}A.\ x = c + a)$
by (*auto simp*: *translate-def*)

lemma *translate-empty* [*simp*]:
$translate\ c\ \{\} = \{\}$
by (*simp add*: *translate-def*)

lemma *translate-compose*:
$translate\ c\ (translate\ d\ A) = translate\ (c + d)\ A$
by (*simp add*: *translate-def image-image add-ac*)

lemma *translate-as-sumset*:
$translate\ c\ A = sumset\ \{c\}\ A$
by (*auto simp*: *translate-def sumset-def*)

lemma *finite-sumset* [*intro*]:
assumes *finite A finite B*
shows *finite* (*sumset A B*)
unfolding *sumset-as-image*
using *assms* **by** (*intro finite-imageI finite-cartesian-product*)

lemma *card-sumset-le*:
assumes *finite A finite B*
shows *card* (*sumset A B*) ≤ *card A* ∗ *card B*
proof −
have *card* (*sumset A B*) = *card* (*case-prod* (+) ‘ (*A* × *B*))
by (*simp add*: *sumset-as-image*)
also have ... ≤ *card* (*A* × *B*)
using *assms* **by** (*intro card-image-le finite-cartesian-product*)
also have ... = *card A* ∗ *card B*
using *assms* **by** *simp*
finally show *?thesis* **.**
qed

lemma *card-translate-eq*:
fixes *A* :: (′*a*::*cancel-comm-monoid-add*) *set*
assumes *finite A*
shows *card* (*translate c A*) = *card A*
proof −
have *inj-on* ((+) *c*) *A*
by (*rule inj-onI*) *simp*
with *assms* **show** *?thesis*
by (*simp add*: *translate-def card-image*)
qed

lemma *translate-Compl*:
fixes *A* :: (′*a*::*ab-group-add*) *set*
shows *translate c* (*UNIV* − *A*) = *UNIV* − *translate c A*
by (*simp add*: *translate-def translation-Compl flip*: *Compl-eq-Diff-UNIV*)

lemma *card-complement-translate-eq*:
fixes *A* :: (′*a*::{*finite*,*ab-group-add*}) *set*
shows *card* (*UNIV* − *translate c A*) = *card* (*UNIV* − *A*)
proof −
have *card* (*translate c* (*UNIV* − *A*)) = *card* (*UNIV* − *A*)
by (*simp add*: *card-translate-eq*)
then show *?thesis*
using *translate-Compl*[*of c A*] **by** *simp*
qed

lemma *sumset-pair-zero*:
fixes *A* :: (′*a*::*comm-monoid-add*) *set*
shows *sumset A* {*0*, *d*} = *A* ∪ *translate d A*
proof
show *sumset A* {*0*, *d*} ⊆ *A* ∪ *translate d A*
proof
fix *x*
assume *x* ∈ *sumset A* {*0*, *d*}
then obtain *a* **where** *a-in*: *a* ∈ *A* **and** *x-cases*: *x* = *a* ∨ *x* = *a* + *d*
by (*auto simp*: *sumset-def*)
then consider *x* = *a* | *x* = *a* + *d*
by *blast*
then show *x* ∈ *A* ∪ *translate d A*

```
    proof cases
      case 1
      then show ?thesis
        using a-in by auto
    next
      case 2
      have x = d + a
        using 2 by (simp add: add.commute)
      then show ?thesis
        using a-in by (auto simp: translate-def)
    qed
  qed
next
  show A ∪ translate d A ⊆ sumset A {0, d}
  proof
    fix x
    assume x ∈ A ∪ translate d A
    then show x ∈ sumset A {0, d}
    proof
      assume x ∈ A
      then show ?thesis
      proof −
        have 0 ∈ {0, d}
          by simp
        have x + 0 ∈ sumset A {0, d}
          using ‹x ∈ A› ‹0 ∈ {0, d}› by (rule sumsetI)
        then show ?thesis
          by simp
      qed
    next
      assume x ∈ translate d A
      then obtain a where a-in: a ∈ A and x-eq: x = d + a
        by (auto simp: translate-def)
      then show ?thesis
        by (auto intro!: sumsetI simp: add.commute)
    qed
  qed
qed

end
theory Modular-Sumsets
  imports
    Sumset-Basics
    HOL−Number-Theory.Number-Theory
begin
```

5 Modular sumsets over integer representatives of $\mathbb{Z}/p\mathbb{Z}$

This theory provides a concrete, self-contained companion to the abstract prime-field development. It works with the canonical integer representatives $0, \ldots, p - 1$ of $\mathbb{Z}/p\mathbb{Z}$ and records the basic cardinality behaviour of modular translations and singleton sumsets in that representation.

```
definition residues :: nat => int set where
  residues p = {0..<int p}

definition mod-translate :: nat => int => int set => int set where
  mod-translate p c A = ((λx. (x + c) mod int p) ‘ A)
```

definition *mod-sumset* :: *nat* => *int set* => *int set* => *int set* **where**
mod-sumset p A B = {*x*. ∃ *a*∈*A*. ∃ *b*∈*B*. *x* = (*a* + *b*) *mod int p*}

lemma *residues-finite* [*simp*]:
finite (*residues p*)
by (*simp add*: *residues-def*)

lemma *card-residues* [*simp*]:
assumes *0* < *p*
shows *card* (*residues p*) = *p*
using *assms* **by** (*simp add*: *residues-def*)

lemma *mod-translate-iff*:
x ∈ *mod-translate p c A* ⟷ (∃ *a*∈*A*. *x* = (*a* + *c*) *mod int p*)
by (*auto simp*: *mod-translate-def*)

lemma *mod-sumset-iff*:
x ∈ *mod-sumset p A B* ⟷ (∃ *a*∈*A*. ∃ *b*∈*B*. *x* = (*a* + *b*) *mod int p*)
by (*auto simp*: *mod-sumset-def*)

lemma *mod-sumset-as-image*:
mod-sumset p A B = ((λ*x*. *x mod int p*) ‘ *sumset A B*)
by (*auto simp*: *mod-sumset-def sumset-def*)

lemma *mod-translate-subset-residues*:
assumes *0* < *p*
shows *mod-translate p c A* ⊆ *residues p*
using *assms* **by** (*auto simp*: *mod-translate-def residues-def*)

lemma *mod-sumset-subset-residues*:
assumes *0* < *p*
shows *mod-sumset p A B* ⊆ *residues p*
using *assms* **by** (*auto simp*: *mod-sumset-def residues-def*)

lemma *mod-sumset-singleton-left*:
mod-sumset p {*a*} *B* = *mod-translate p a B*
proof
show *mod-sumset p* {*a*} *B* ⊆ *mod-translate p a B*
proof
fix *x*
assume *x* ∈ *mod-sumset p* {*a*} *B*
then obtain *b* **where** *b* ∈ *B x* = (*a* + *b*) *mod int p*
by (*auto simp*: *mod-sumset-def*)
then show *x* ∈ *mod-translate p a B*
by (*auto simp*: *mod-translate-def add.commute*)
qed
next
show *mod-translate p a B* ⊆ *mod-sumset p* {*a*} *B*
proof
fix *x*
assume *x* ∈ *mod-translate p a B*
then obtain *b* **where** *b* ∈ *B x* = (*b* + *a*) *mod int p*
by (*auto simp*: *mod-translate-def*)
then show *x* ∈ *mod-sumset p* {*a*} *B*
by (*auto simp*: *mod-sumset-def add.commute*)
qed
qed

lemma *mod-sumset-singleton-right*:

mod-sumset p A {b} = mod-translate p b A
by (*auto simp*: *mod-sumset-def mod-translate-def ac-simps*)

lemma *mod-translate-inj-on-residues*:
assumes *0 < p*
shows *inj-on* (λ*x*. (*x* + *c*) *mod int p*) (*residues p*)
proof (*rule inj-onI*)
fix *x y*
assume *x-in*: *x* ∈ *residues p*
assume *y-in*: *y* ∈ *residues p*
assume *eq*: (*x* + *c*) *mod int p* = (*y* + *c*) *mod int p*

have *cong-xyc*: [*x* + *c* = *y* + *c*] (*mod int p*)
using *eq* **by** (*simp add*: *cong-def*)
then have *cong-xy*: [*x* = *y*] (*mod int p*)
by (*simp add*: *cong-add-rcancel*)

from *x-in* **have** *x-bounds*: *0* ≤ *x x* < *int p*
by (*auto simp*: *residues-def*)
from *y-in* **have** *y-bounds*: *0* ≤ *y y* < *int p*
by (*auto simp*: *residues-def*)

show *x* = *y*
by (*rule cong-less-imp-eq-int*[*OF x-bounds y-bounds cong-xy*])
qed

lemma *card-mod-translate-eq*:
assumes *0 < p*
assumes *A* ⊆ *residues p*
shows *card* (*mod-translate p c A*) = *card A*
proof −
have *finite-A*: *finite A*
using *assms* **by** (*meson finite-subset residues-finite*)
have *inj-residues*: *inj-on* (λ*x*. (*x* + *c*) *mod int p*) (*residues p*)
by (*rule mod-translate-inj-on-residues*[*OF assms*(*1*)])
have *inj-A*: *inj-on* (λ*x*. (*x* + *c*) *mod int p*) *A*
by (*rule inj-on-subset*[*OF inj-residues assms*(*2*)])
show *?thesis*
unfolding *mod-translate-def*
by (*simp add*: *card-image finite-A inj-A*)
qed

lemma *card-mod-sumset-singleton-left*:
assumes *0 < p*
assumes *B* ⊆ *residues p*
shows *card* (*mod-sumset p {a} B*) = *card B*
using *assms* **by** (*simp add*: *mod-sumset-singleton-left card-mod-translate-eq*)

lemma *card-mod-sumset-singleton-right*:
assumes *0 < p*
assumes *A* ⊆ *residues p*
shows *card* (*mod-sumset p A {b}*) = *card A*
using *assms* **by** (*simp add*: *mod-sumset-singleton-right card-mod-translate-eq*)

These lemmas complement the abstract prime-field proofs of Cauchy-Davenport and Vosper by providing a convenient API for expressing the corresponding cardinality facts over the standard residue representatives of prime cyclic groups.

end
theory *Cauchy-Davenport-Prime-Field*

imports
Sumset-Basics
HOL−Computational-Algebra.Computational-Algebra
HOL−Number-Theory.Number-Theory
begin

6 Cauchy-Davenport over prime fields

This theory proves the prime-field Cauchy-Davenport theorem by a polynomial-method argument. The core technical work packages the bivariate degree bookkeeping needed to show that a suitably chosen polynomial cannot vanish on too large a grid unless the expected lower bound holds.

6.1 Polynomial infrastructure

definition *total-degree-le* :: *'a::comm-semiring-1 poly poly ⇒ nat ⇒ bool* **where**
total-degree-le P d ⟷
(∀ i. if i ≤ d then degree (poly.coeff P i) ≤ d − i else poly.coeff P i = 0)

definition *sum-factor* :: *'a::comm-ring-1 ⇒ 'a poly poly* **where**
sum-factor c = [: [:−c, 1:], [:1:] :]

lemma *coeff-poly-const*:
fixes *P* :: *'a::comm-semiring-1 poly poly*
shows *poly.coeff (poly P [:a:]) j =*
*(∑ i≤degree P. poly.coeff (poly.coeff P i) j * a ^ i)*
proof −
have *poly.coeff (poly P [:a:]) j =*
*poly.coeff (∑ i≤degree P. poly.coeff P i * [:a:] ^ i) j*
by (*simp add*: *poly-altdef*)
also have ... =
*(∑ i≤degree P. poly.coeff (poly.coeff P i * [:a:] ^ i) j)*
by (*simp add*: *coeff-sum*)
also have ... =
*(∑ i≤degree P. poly.coeff (poly.coeff P i) j * a ^ i)*
proof (*intro sum.cong refl*)
fix *i*
assume *i* ∈ {..*degree P*}
have [:*a*:] ^ *i* = [:*a* ^ *i*:]
by (*induction i*) (*simp-all add*: *mult.commute*)
then show *poly.coeff (poly.coeff P i * [:a:] ^ i) j =*
*poly.coeff (poly.coeff P i) j * a ^ i*
by (*simp add*: *coeff-mult mult.commute mult.left-commute mult.assoc*)
qed
finally show *?thesis* **.**
qed

lemma *total-degree-le-eval-const*:
fixes *P* :: *'a::comm-semiring-1 poly poly*
assumes *td*: *total-degree-le P d*
shows *degree (poly P [:a:]) ≤ d*
proof −
have *zero-above*: *poly.coeff (poly P [:a:]) j = 0* **if** *j*: *j > d* **for** *j*
proof −
have *eval-coeff*: *poly.coeff (poly P [:a:]) j =*
*(∑ i≤degree P. poly.coeff (poly.coeff P i) j * a ^ i)*
by (*rule coeff-poly-const*)
have *sum-zero*: *(∑ i≤degree P. poly.coeff (poly.coeff P i) j * a ^ i) = 0*
proof (*intro sum.neutral ballI*)

```
    fix i
    assume i: i ∈ {..degree P}
    show poly.coeff (poly.coeff P i) j * a ^ i = 0
    proof (cases i ≤ d)
      case True
      with td have degree (poly.coeff P i) ≤ d − i
        by (simp add: total-degree-le-def)
      moreover from True j have d − i < j
        by simp
      ultimately have poly.coeff (poly.coeff P i) j = 0
        by (meson coeff-eq-0 le-less-trans)
      then show ?thesis
        by simp
    next
      case False
      with td have poly.coeff P i = 0
        by (simp add: total-degree-le-def)
      then show ?thesis
        by simp
    qed
  qed
  show ?thesis
    using eval-coeff sum-zero by simp
 qed
 show ?thesis
   by (rule degree-le) (use zero-above in blast)
qed

lemma total-degree-le-eval-const-top:
 fixes P :: 'a::comm-semiring-1 poly poly
 assumes td: total-degree-le P d
 shows poly.coeff (poly P [:a:]) d = poly.coeff (poly.coeff P 0) d
proof −
 have poly.coeff (poly P [:a:]) d =
    (∑ i≤degree P. poly.coeff (poly.coeff P i) d * a ^ i)
   by (rule coeff-poly-const)
 also have ... =
    poly.coeff (poly.coeff P 0) d * a ^ 0 +
    (∑ i∈{..degree P} − {0}. poly.coeff (poly.coeff P i) d * a ^ i)
   by (subst sum.remove[of {..degree P} 0 λi. poly.coeff (poly.coeff P i) d * a ^ i]) simp-all
 also have ... = poly.coeff (poly.coeff P 0) d * a ^ 0
 proof −
   have (∑ i∈{..degree P} − {0}. poly.coeff (poly.coeff P i) d * a ^ i) = 0
   proof (intro sum.neutral ballI)
     fix i
     assume i: i ∈ {..degree P} − {0}
     then have i-pos: 0 < i
       by simp
     show poly.coeff (poly.coeff P i) d * a ^ i = 0
     proof (cases i ≤ d)
       case True
       with td have deg-i: degree (poly.coeff P i) ≤ d − i
       by (simp add: total-degree-le-def)
     from True i-pos have d − i < d
       by simp
     with deg-i have poly.coeff (poly.coeff P i) d = 0
       by (meson coeff-eq-0 le-less-trans)
       then show ?thesis
         by simp
```

```
    next
      case False
      with td have poly.coeff P i = 0
        by (simp add: total-degree-le-def)
      then show ?thesis
        by simp
    qed
  qed
  then show ?thesis
    by simp
qed

lemma coeff-sum-factor-mult-0:
  poly.coeff (sum-factor c * P) 0 = [:−c, 1:] * poly.coeff P 0
  by (simp add: sum-factor-def coeff-mult)

lemma coeff-sum-factor-mult-Suc:
  poly.coeff (sum-factor c * P) (Suc i) =
    [:−c, 1:] * poly.coeff P (Suc i) + poly.coeff P i
  by (simp add: sum-factor-def coeff-mult)

lemma total-degree-le-sum-factor-mult:
  fixes P :: 'a::field poly poly
  assumes td: total-degree-le P d
  shows total-degree-le (sum-factor c * P) (Suc d)
unfolding total-degree-le-def
proof
  fix i
  show (if i ≤ Suc d then degree (poly.coeff (sum-factor c * P) i) ≤ Suc d − i
        else poly.coeff (sum-factor c * P) i = 0)
  proof (cases i)
    case 0
    have degree (poly.coeff (sum-factor c * P) 0) =
       degree ([:−c, 1:] * poly.coeff P 0)
      by (simp add: coeff-sum-factor-mult-0)
    also have ... ≤ degree [:−c, 1:] + degree (poly.coeff P 0)
      by (rule degree-mult-le)
    also have ... ≤ Suc d
    proof −
      have degree (poly.coeff P 0) ≤ d
      proof −
        from td have if 0 ≤ d then degree (poly.coeff P 0) ≤ d − 0 else poly.coeff P 0 = 0
          unfolding total-degree-le-def by blast
        then show ?thesis
          by simp
      qed
      then show ?thesis
        by simp
    qed
    finally show ?thesis
      using 0 by simp
  next
    case (Suc k)
    show ?thesis
    proof (cases k ≤ d)
      case True
```

```
  have deg-Suc: degree (poly.coeff P (Suc k)) ≤ d − Suc k
  proof (cases Suc k ≤ d)
    case True
    with td show ?thesis
      by (simp add: total-degree-le-def)
  next
    case False
    with td have poly.coeff P (Suc k) = 0
      by (simp add: total-degree-le-def)
    then show ?thesis
      by simp
  qed
  from True td have deg-k: degree (poly.coeff P k) ≤ d − k
    by (simp add: total-degree-le-def)
  have degree (poly.coeff (sum-factor c ∗ P) (Suc k)) =
      degree ([:−c, 1:] ∗ poly.coeff P (Suc k) + poly.coeff P k)
    by (simp add: coeff-sum-factor-mult-Suc)
  also have ... ≤
      max (degree ([:−c, 1:] ∗ poly.coeff P (Suc k))) (degree (poly.coeff P k))
    by (rule degree-add-le-max)
  also have ... ≤ d − k
  proof −
    have deg-lin: degree ([:−c, 1:] ∗ poly.coeff P (Suc k)) ≤ d − k
    proof −
      show ?thesis
      proof (cases poly.coeff P (Suc k) = 0)
        case True
        then show ?thesis
          by simp
      next
        case False
        have suc-le: Suc k ≤ d
        proof (rule ccontr)
          assume ¬ Suc k ≤ d
          with td have poly.coeff P (Suc k) = 0
            by (simp add: total-degree-le-def)
          with False show False
            by contradiction
        qed
        have degree ([:−c, 1:] ∗ poly.coeff P (Suc k)) =
            degree [:−c, 1:] + degree (poly.coeff P (Suc k))
          by (rule degree-mult-eq) (use False in simp-all)
        also have ... ≤ 1 + (d − Suc k)
          using deg-Suc by simp
        also have ... = d − k
          using suc-le by simp
        finally show ?thesis .
      qed
    qed
    show ?thesis
      using deg-lin deg-k by simp
  qed
  finally show ?thesis
    using Suc True by simp
next
  case False
  with td have zero-k: poly.coeff P k = 0
    by (simp add: total-degree-le-def)
  from False have d < k
```

```
      by simp
    then have d < Suc k
      by simp
    with td have zero-Suc: poly.coeff P (Suc k) = 0
      by (simp add: total-degree-le-def)
    show ?thesis
      using Suc zero-k zero-Suc by (simp add: coeff-sum-factor-mult-Suc)
  qed
 qed
qed

lemma total-degree-le-prod-sum-factor:
 fixes C :: 'a::field set
 assumes finite C
 shows total-degree-le (prod sum-factor C) (card C)
 using assms
proof (induction rule: finite-induct)
 case empty
 show ?case
  by (simp add: total-degree-le-def)
next
 case (insert c C)
 then show ?case
  by (simp add: total-degree-le-sum-factor-mult)
qed

lemma coeff-coeff-sum-factor-mult-top:
 fixes P :: 'a::field poly poly
 assumes td: total-degree-le P d
 assumes ij: i + j = Suc d
 shows poly.coeff (poly.coeff (sum-factor c * P) i) j =
  (if i = 0 then 0 else poly.coeff (poly.coeff P (i - 1)) j) +
  (if j = 0 then 0 else poly.coeff (poly.coeff P i) (j - 1))
proof (cases i)
 case 0
 with ij have j: j = Suc d
  by simp
 have poly.coeff (poly.coeff (sum-factor c * P) 0) j =
    poly.coeff ([:-c, 1:] * poly.coeff P 0) j
  by (simp add: coeff-sum-factor-mult-0)
 also have ... =
    poly.coeff (poly.coeff P 0) (j - 1) - c * poly.coeff (poly.coeff P 0) j
  using j by (cases j) (simp-all add: coeff-mult)
 also have poly.coeff (poly.coeff P 0) j = 0
 proof -
  have degree (poly.coeff P 0) ≤ d
  proof -
   have if 0 ≤ d then degree (poly.coeff P 0) ≤ d - 0 else poly.coeff P 0 = 0
    using td unfolding total-degree-le-def by blast
   then show ?thesis
    by simp
  qed
  with j show ?thesis
   by (meson coeff-eq-0 less-Suc-eq-le)
 qed
 finally show ?thesis
  using 0 j by simp
next
 case (Suc k)
```

```
  have rec: poly.coeff (poly.coeff (sum-factor c * P) (Suc k)) j =
      (if j = 0 then 0 else poly.coeff (poly.coeff P (Suc k)) (j - 1)) -
      c * poly.coeff (poly.coeff P (Suc k)) j +
      poly.coeff (poly.coeff P k) j
  proof -
    have poly.coeff (poly.coeff (sum-factor c * P) (Suc k)) j =
        poly.coeff ([:-c, 1:] * poly.coeff P (Suc k) + poly.coeff P k) j
      by (simp add: coeff-sum-factor-mult-Suc)
    also have ... =
        poly.coeff ([:-c, 1:] * poly.coeff P (Suc k)) j +
        poly.coeff (poly.coeff P k) j
      by simp
    also have poly.coeff ([:-c, 1:] * poly.coeff P (Suc k)) j =
        (if j = 0 then 0 else poly.coeff (poly.coeff P (Suc k)) (j - 1)) -
        c * poly.coeff (poly.coeff P (Suc k)) j
    proof (cases j)
      case 0
      then show ?thesis
        by (simp add: coeff-mult)
    next
      case (Suc l)
      then show ?thesis
        by (simp add: coeff-mult)
    qed
    finally show ?thesis
      by simp
  qed
  have zero-term: poly.coeff (poly.coeff P (Suc k)) j = 0
  proof (cases Suc k ≤ d)
    case True
    with td have degree (poly.coeff P (Suc k)) ≤ d - Suc k
      by (simp add: total-degree-le-def)
    moreover have d - Suc k < j
    proof -
      from ij Suc have j = Suc d - Suc k
        by simp
      with True show ?thesis
        by simp
    qed
    ultimately show ?thesis
      by (meson coeff-eq-0 le-less-trans)
  next
    case False
    with td show ?thesis
      by (simp add: total-degree-le-def)
  qed
  show ?thesis
    using Suc rec zero-term by (cases j) simp-all
qed

lemma coeff-coeff-prod-sum-factor-top:
  fixes C :: 'a::field set
  assumes fin: finite C
  assumes ij: i + j = card C
  shows poly.coeff (poly.coeff (prod sum-factor C) i) j = of-nat (card C choose i)
  using fin ij
proof (induction C arbitrary: i j rule: finite-induct)
  case empty
  then have i = 0 j = 0
```

```
    by simp-all
  then show ?case
    by simp
next
  case (insert c C)
  let ?P = prod sum-factor C
  have td: total-degree-le ?P (card C)
    using insert.hyps by (simp add: total-degree-le-prod-sum-factor)
  have ij': i + j = Suc (card C)
    using insert.prems insert.hyps by simp
  have top-rec:
    poly.coeff (poly.coeff (prod sum-factor (insert c C)) i) j =
      (if i = 0 then 0 else poly.coeff (poly.coeff ?P (i − 1)) j) +
      (if j = 0 then 0 else poly.coeff (poly.coeff ?P i) (j − 1))
  proof −
    have poly.coeff (poly.coeff (prod sum-factor (insert c C)) i) j =
        poly.coeff (poly.coeff (sum-factor c ∗ ?P) i) j
      using insert.hyps by simp
    also have ... =
        (if i = 0 then 0 else poly.coeff (poly.coeff ?P (i − 1)) j) +
        (if j = 0 then 0 else poly.coeff (poly.coeff ?P i) (j − 1))
      using coeff-coeff-sum-factor-mult-top[OF td ij'] .
    finally show ?thesis .
  qed
  show ?case
  proof (cases i = 0 ∨ j = 0)
    case True
    then show ?thesis
    proof
      assume i = 0
      with ij' have j: j = Suc (card C)
        by simp
      have poly.coeff (poly.coeff (prod sum-factor (insert c C)) i) j =
          poly.coeff (poly.coeff ?P 0) (card C)
        using top-rec ‹i = 0› j by simp
      also have ... = of-nat (card C choose 0)
        using insert.IH[of 0 card C] by simp
      finally show ?thesis
        using ‹i = 0› by simp
    next
      assume j = 0
      with ij' have i: i = Suc (card C)
        by simp
      have poly.coeff (poly.coeff (prod sum-factor (insert c C)) i) j =
          poly.coeff (poly.coeff ?P (card C)) 0
        using top-rec ‹j = 0› i by simp
      also have ... = of-nat (card C choose card C)
        using insert.IH[of card C 0] by simp
      finally show ?thesis
        using i ‹j = 0› insert.hyps by simp
    qed
  next
    case False
    then have i-pos: i > 0 and j-pos: j > 0
      by auto
    have split1: i − 1 + j = card C and split2: i + (j − 1) = card C
      using ij' i-pos j-pos by simp-all
    have poly.coeff (poly.coeff (prod sum-factor (insert c C)) i) j =
        poly.coeff (poly.coeff ?P (i − 1)) j +
```

```
      poly.coeff (poly.coeff ?P i) (j − 1)
    using top-rec False by simp
  also have ... = of-nat (card C choose (i − 1)) + of-nat (card C choose i)
    using insert.IH[of i − 1 j] insert.IH[of i j − 1] split1 split2 by simp
  also have ... = of-nat (Suc (card C) choose i)
    using i-pos by (cases i) simp-all
  finally show ?thesis
    using insert.hyps by simp
 qed
qed

lemma poly-sum-factor:
 poly (poly (sum-factor c) [:x:]) y = x + y − c
 by (simp add: sum-factor-def)

lemma prime-not-dvd-binomial:
 assumes prime p k ≤ n n < p
 shows ¬ p dvd (n choose k)
proof
 assume dvd-choose: p dvd (n choose k)
 have fact-expand: fact n = fact k * fact (n − k) * (n choose k)
   using binomial-fact-lemma[OF assms(2)] by (simp add: mult-ac)
 have p dvd fact n
   using dvd-choose fact-expand by simp
 with assms show False
   by (simp add: prime-dvd-fact-iff)
qed

lemma finite-subset-with-card:
 assumes finite U n ≤ card U
 shows ∃ V. V ⊆ U ∧ card V = n
 using assms
proof (induction U arbitrary: n rule: finite-induct)
 case empty
 then show ?case
   by auto
next
 case (insert x U)
 show ?case
 proof (cases n ≤ card U)
   case True
   then obtain V where V ⊆ U card V = n
     using insert.IH by blast
   then show ?thesis
     using insert.hyps by auto
 next
   case False
   have n-le: n ≤ Suc (card U)
     using insert.prems insert.hyps by simp
   from False have Suc (card U) ≤ n
     by simp
   with n-le have n = Suc (card U)
     by simp
   then show ?thesis
     using insert.hyps by auto
 qed
qed

lemma prime-card-eq-char:
```

```
  assumes prime-card: prime (card (UNIV :: 'a::finite-field set))
  shows CHAR('a) = card (UNIV :: 'a set)
proof −
  have CHAR('a) dvd card (UNIV :: 'a set)
    by (rule CHAR-dvd-CARD)
  moreover from prime-card have ⋀m. m dvd card (UNIV :: 'a set) ⟹ m = 1 ∨ m = card (UNIV ::
'a set)
    by (simp add: prime-nat-iff)
  ultimately have CHAR('a) = 1 ∨ CHAR('a) = card (UNIV :: 'a set)
    by blast
  with CHAR-prime show ?thesis
    by auto
qed

lemma of-nat-binomial-ne-zero:
  assumes prime-card: prime (card (UNIV :: 'a::finite-field set))
  assumes k ≤ n n < card (UNIV :: 'a set)
  shows of-nat (n choose k) ≠ (0 :: 'a)
proof −
  have char-eq: CHAR('a) = card (UNIV :: 'a set)
    by (rule prime-card-eq-char[OF prime-card])
  have ¬ CHAR('a) dvd (n choose k)
    using assms by (simp add: char-eq prime-not-dvd-binomial prime-card-eq-char)
  then show ?thesis
    using of-nat-eq-0-iff-char-dvd by blast
qed

lemma coeff-x-factor-plus-const-0:
  poly.coeff ([:−[:a:], 1:] * Q + [:R:]) 0 = − [:a:] * poly.coeff Q 0 + R
  by (simp add: coeff-mult)

lemma coeff-x-factor-plus-const-Suc:
  poly.coeff ([:−[:a:], 1:] * Q + [:R:]) (Suc i) =
    − [:a:] * poly.coeff Q (Suc i) + poly.coeff Q i
  by (simp add: coeff-mult)

lemma total-degree-le-factor-out:
  fixes P Q :: 'a::field poly poly
  assumes td: total-degree-le P (Suc d)
  assumes decomp: P = [:−[:a:], 1:] * Q + [:R:]
  shows total-degree-le Q d
proof −
  have outer-zero: poly.coeff Q i = 0 if d < i for i
  proof (rule ccontr)
    assume nz: poly.coeff Q i ≠ 0
    let ?I = {k. poly.coeff Q k ≠ 0}
    have finI: finite ?I
    proof (rule finite-subset[of - {..degree Q}])
      show ?I ⊆ {..degree Q}
      proof
        fix k
        assume k-in: k ∈ ?I
        have nz-k: poly.coeff Q k ≠ 0
          using k-in by simp
        show k ∈ {..degree Q}
        proof (rule ccontr)
          assume k ∉ {..degree Q}
          then have degree Q < k
            by simp
```

```
      then have poly.coeff Q k = 0
        by (rule coeff-eq-0)
      with nz-k show False
        by contradiction
    qed
  qed
  qed simp
  have i-in: i ∈ ?I
    using nz by simp
  let ?k = Max ?I
  have k-in: ?k ∈ ?I
    by (rule Max-in[OF finI]) (use i-in in auto)
  have i-le-k: i ≤ ?k
    using finI i-in by (intro Max-ge) auto
  have q-suc-zero: poly.coeff Q (Suc ?k) = 0
  proof (rule ccontr)
    assume nz-suc: poly.coeff Q (Suc ?k) ≠ 0
    then have suc-in: Suc ?k ∈ ?I
      by simp
    have Suc ?k ≤ ?k
      using finI suc-in by (intro Max-ge) auto
    then show False
      by simp
  qed
  have poly.coeff P (Suc ?k) = poly.coeff Q ?k
    using decomp q-suc-zero by (simp add: coeff-x-factor-plus-const-Suc)
  moreover have poly.coeff P (Suc ?k) = 0
    using td that i-le-k by (simp add: total-degree-le-def)
  ultimately show False
    using k-in by simp
qed
have degree-bound: degree (poly.coeff Q i) ≤ d − i if i ≤ d for i
proof (rule ccontr)
  assume bad: ¬ degree (poly.coeff Q i) ≤ d − i
  let ?I = {k. k ≤ d ∧ degree (poly.coeff Q k) > d − k}
  have finI: finite ?I
    by simp
  have i-in: i ∈ ?I
    using that bad by simp
  let ?k = Max ?I
  have k-in: ?k ∈ ?I
    by (rule Max-in[OF finI]) (use i-in in auto)
  have k-le-d: ?k ≤ d and deg-k: degree (poly.coeff Q ?k) > d − ?k
    using k-in by auto
  have next-bound: degree (poly.coeff Q (Suc ?k)) ≤ d − Suc ?k
  proof (cases Suc ?k ≤ d)
    case True
    have Suc ?k ∉ ?I
    proof
      assume suc-in: Suc ?k ∈ ?I
      have Suc ?k ≤ ?k
        using finI suc-in by (intro Max-ge) auto
      then show False
        by simp
    qed
    then show ?thesis
      using True by simp
  next
    case False
```

```
    have poly.coeff Q (Suc ?k) = 0
      using outer-zero False by simp
    then show ?thesis
      by simp
  qed
  have deg-tail: degree (− [:a:] ∗ poly.coeff Q (Suc ?k)) < degree (poly.coeff Q ?k)
    using next-bound deg-k by (auto intro: le-less-trans simp: degree-mult-le)
  have poly.coeff P (Suc ?k) =
      poly.coeff Q ?k + (− [:a:] ∗ poly.coeff Q (Suc ?k))
    using decomp by (simp add: coeff-x-factor-plus-const-Suc algebra-simps)
  then have eq-p: poly.coeff P (Suc ?k) =
      poly.coeff Q ?k + (− [:a:] ∗ poly.coeff Q (Suc ?k)) .
  have deg-sum: degree (poly.coeff Q ?k + (− [:a:] ∗ poly.coeff Q (Suc ?k))) =
      degree (poly.coeff Q ?k)
    by (rule degree-add-eq-left[OF deg-tail])
  have deg-p: degree (poly.coeff P (Suc ?k)) = degree (poly.coeff Q ?k)
    using eq-p deg-sum by simp
  have degree (poly.coeff P (Suc ?k)) ≤ Suc d − Suc ?k
  proof −
    have Suc ?k ≤ Suc d
      using k-le-d by simp
    from td have step:
      if Suc ?k ≤ Suc d
       then degree (poly.coeff P (Suc ?k)) ≤ Suc d − Suc ?k
       else poly.coeff P (Suc ?k) = 0
      unfolding total-degree-le-def by blast
    with ‹Suc ?k ≤ Suc d› show ?thesis
      by simp
  qed
  then show False
    using deg-k deg-p by simp
 qed
 show ?thesis
  by (simp add: total-degree-le-def degree-bound outer-zero)
qed

lemma coeff-coeff-x-factor-top:
 fixes Q :: 'a::field poly poly
 assumes td: total-degree-le Q d
 assumes mn: m + n = d
 shows poly.coeff (poly.coeff ([:−[:a:], 1:] ∗ Q + [:R:]) (Suc m)) n =
   poly.coeff (poly.coeff Q m) n
proof −
 have poly.coeff (poly.coeff ([:−[:a:], 1:] ∗ Q + [:R:]) (Suc m)) n =
     poly.coeff (poly.coeff Q m) n − a ∗ poly.coeff (poly.coeff Q (Suc m)) n
   by (simp add: coeff-x-factor-plus-const-Suc coeff-mult)
 moreover have poly.coeff (poly.coeff Q (Suc m)) n = 0
 proof −
   show ?thesis
   proof (cases Suc m ≤ d)
     case True
     have deg-bound: degree (poly.coeff Q (Suc m)) ≤ d − Suc m
       using td True by (simp add: total-degree-le-def)
     have d − Suc m < n
     proof −
       from mn have n = d − m
         by simp
       with True show ?thesis
         by simp
```

```
    qed
    with deg-bound show ?thesis
      by (meson coeff-eq-0 le-less-trans)
  next
    case False
    with td show ?thesis
      by (simp add: total-degree-le-def)
  qed
 qed
 ultimately show ?thesis
  by simp
qed

lemma grid-nonzero-from-top-coeff:
 fixes P :: 'a::field poly poly
 assumes cardA: card A = Suc m
 assumes cardB: card B = Suc n
 assumes td: total-degree-le P (m + n)
 assumes top: poly.coeff (poly.coeff P m) n ≠ 0
 shows ∃ a∈A. ∃ b∈B. poly (poly P [:a:]) b ≠ 0
 using cardA td top cardB
proof (induction m arbitrary: A P)
 case 0
 show ?case
 proof −
  from 0.prems(1) have ∃ a. A = {a}
    by (simp add: card-1-singleton-iff)
  then obtain a where A: A = {a}
    by blast
  note td0 = 0.prems(2)
  note top0 = 0.prems(3)
  define R where R = poly P [:a:]
  have coeff-R: poly.coeff R (0 + n) = poly.coeff (poly.coeff P 0) (0 + n)
    unfolding R-def by (rule total-degree-le-eval-const-top[OF td0])
  have coeff-R-nz: poly.coeff R n ≠ 0
    using coeff-R top0 by simp
  then have nz-R: R ≠ 0
    by auto
  show ?thesis
  proof (rule ccontr)
    assume ¬ ?thesis
    then have vanish-B: ∀ b∈B. poly R b = 0
      using A by (simp add: R-def)
    have B ⊆ {b. poly R b = 0}
      using vanish-B by auto
    then have card B ≤ card {b. poly R b = 0}
      by (intro card-mono poly-roots-finite[OF nz-R]) auto
    also have ... ≤ degree R
      by (rule card-poly-roots-bound[OF nz-R])
    finally have cardB-le-degR: card B ≤ degree R .
    have degR': degree R ≤ 0 + n
      unfolding R-def by (rule total-degree-le-eval-const[OF td0])
    have degR: degree R ≤ n
      using degR' by simp
    show False
      using cardB cardB-le-degR degR by linarith
  qed
 qed
next
```

```
case (Suc m)
have 0 < card A
  using Suc.prems(1) by simp
then have A ≠ {}
  by auto
then obtain a where a-in: a ∈ A
  by auto
show ?case
proof (cases ∃ b∈B. poly (poly P [:a:]) b ≠ 0)
  case True
  then show ?thesis
    using a-in by blast
next
  case False
  define R where R = poly P [:a:]
  have vanish-R: poly R b = 0 if b ∈ B for b
    using False that by (simp add: R-def)
  have dvd: [:−[:a:], 1:] dvd (P − [:R:])
  proof −
    have eval0: poly (P − [:poly P [:a:]:]) [:a:] = 0
      by simp
    have dvd0: [:−[:a:], 1:] dvd (P − [:poly P [:a:]:])
      using eval0 by (rule iffD1[OF poly-eq-0-iff-dvd])
    show ?thesis
      using dvd0 R-def by simp
  qed
  then obtain Q where Q: P = [:−[:a:], 1:] * Q + [:R:]
    by (auto simp: dvd-def algebra-simps)
  let ?A' = A − {a}
  have cardA': card ?A' = Suc m
    using Suc.prems(1) a-in by (simp add: card-Diff-singleton-if)
  have tdQ: total-degree-le Q (m + n)
  proof −
    have tdP': total-degree-le P (Suc (m + n))
      using Suc.prems(2) by simp
    show ?thesis
      by (rule total-degree-le-factor-out[OF tdP' Q])
  qed
  have topQ: poly.coeff (poly.coeff Q m) n ≠ 0
  proof −
    have coeff-eq:
      poly.coeff (poly.coeff P (Suc m)) n = poly.coeff (poly.coeff Q m) n
    proof −
      have poly.coeff (poly.coeff ([:−[:a:], 1:] * Q + [:R:]) (Suc m)) n =
          poly.coeff (poly.coeff Q m) n
        by (rule coeff-coeff-x-factor-top[OF tdQ]) simp
      then show ?thesis
        by (simp add: Q)
    qed
    with Suc.prems(3) show ?thesis
      by simp
  qed
  obtain x b where xb: x ∈ ?A' b ∈ B poly (poly Q [:x:]) b ≠ 0
    using Suc.IH[OF cardA' tdQ topQ cardB] by blast
  have x-neq-a: x ≠ a
    using xb by simp
  have poly (poly P [:x:]) b = (x − a) * poly (poly Q [:x:]) b + poly R b
  proof −
    have poly (poly P [:x:]) b =
```

```
      poly (poly ([:−[:a:], 1:] * Q + [:R:]) [:x:]) b
    using Q by simp
  also have ... = poly (poly ([:−[:a:], 1:] * Q) [:x:]) b + poly R b
    by simp
  also have poly (poly ([:−[:a:], 1:] * Q) [:x:]) b =
      (x − a) * poly (poly Q [:x:]) b
    by (simp add: algebra-simps)
  finally show ?thesis .
 qed
 also have ... = (x − a) * poly (poly Q [:x:]) b
  using xb vanish-R by simp
 also have ... ≠ 0
  using xb x-neq-a by simp
 finally have poly (poly P [:x:]) b ≠ 0 .
 then show ?thesis
  using xb by blast
 qed
qed

lemma sumset-eq-UNIV-if-large:
 fixes A B :: 'a::finite-field set
 assumes card (UNIV :: 'a set) < card A + card B − 1
 assumes A ≠ {} B ≠ {}
 shows sumset A B = UNIV
proof
 show sumset A B ⊆ UNIV
  by simp
next
 show UNIV ⊆ sumset A B
 proof
  fix x :: 'a
  let ?T = translate x ((uminus) ‘ B)
  have card-negB: card ((uminus) ‘ B) = card B
   by (intro card-image inj-onI) auto
  have card-T: card ?T = card B
   by (simp add: card-negB card-translate-eq)
  have card A + card ?T = card (A ∪ ?T) + card (A ∩ ?T)
   using card-Un-Int[of A ?T] by simp
  moreover have card (A ∪ ?T) ≤ card (UNIV :: 'a set)
   by (rule card-mono) auto
  ultimately have card (A ∩ ?T) > 0
   using assms card-T by linarith
  then have A ∩ ?T ≠ {}
   by auto
  then obtain a where a: a ∈ A ∩ ?T
   by auto
  then obtain b where b: b ∈ B a = x + (− b)
   by (auto simp: translate-iff)
  have a-in: a ∈ A
   using a by simp
  have x = a + b
   using b by simp
  moreover have a + b ∈ sumset A B
   using a-in b(1) by (rule sumsetI)
  ultimately show x ∈ sumset A B
   by simp
 qed
qed
```

6.2 The main lower bound

The final statement is the abstract prime-field version of Cauchy-Davenport: for nonempty finite subsets of a field whose cardinality is prime, the sumset has size at least the expected truncated lower bound.

theorem *cauchy-davenport-prime-field*:
fixes *A B* :: *'a*::*finite-field set*
assumes *prime-card*: *prime* (*card* (*UNIV* :: *'a set*))
assumes $A \neq \{\}$ $B \neq \{\}$
shows *card* (*sumset A B*) $\geq$ *min* (*card* (*UNIV* :: *'a set*)) (*card A* + *card B* − *1*)
proof (*cases card* (*UNIV* :: *'a set*) < *card A* + *card B* − *1*)
case *True*
have *sumset A B* = *UNIV*
proof (*rule sumset-eq-UNIV-if-large*)
show *card* (*UNIV* :: *'a set*) < *card A* + *card B* − *1*
using *True* **by** *assumption*
show $A \neq \{\}$
using *assms*(*2*) **.**
show $B \neq \{\}$
using *assms*(*3*) **.**
qed
then show *?thesis*
by *simp*
next
case *False*
let *?m* = *card A* − *1*
let *?n* = *card B* − *1*
have *cardA-pos*: *0* < *card A* **and** *cardB-pos*: *0* < *card B*
using *assms*(*2*,*3*) **by** *auto*
have *cardA*: *card A* = *Suc ?m* **and** *cardB*: *card B* = *Suc ?n*
using *cardA-pos cardB-pos* **by** *simp-all*
have *mn-lt*: *?m* + *?n* < *card* (*UNIV* :: *'a set*)
proof −
have *le-card*: *card A* + *card B* − *1* ≤ *card* (*UNIV* :: *'a set*)
using *False* **by** *simp*
have *Suc* (*?m* + *?n*) = *card A* + *card B* − *1*
using *cardA cardB* **by** *simp*
with *le-card* **have** *Suc* (*?m* + *?n*) ≤ *card* (*UNIV* :: *'a set*)
by *simp*
then show *?thesis*
by *simp*
qed
show *?thesis*
proof (*rule ccontr*)
assume *bad*: ¬ *card* (*sumset A B*) ≥ *min* (*card* (*UNIV* :: *'a set*)) (*card A* + *card B* − *1*)
have *sumset-small*: *card* (*sumset A B*) ≤ *?m* + *?n*
using *False bad assms* **by** *simp*
obtain *C* **where** *C*: *sumset A B* ⊆ *C C* ⊆ *UNIV card C* = *?m* + *?n*
using *exists-subset-between*[*of sumset A B ?m* + *?n UNIV*] *sumset-small mn-lt* **by** *auto*
define *F* **where** *F* = *prod sum-factor C*
have *tdF*: *total-degree-le F* (*?m* + *?n*)
proof −
have *finC*: *finite C*
using *C* **by** *auto*
have *total-degree-le F* (*card C*)
unfolding *F-def* **by** (*rule total-degree-le-prod-sum-factor*[*OF finC*])
with *C* **show** *?thesis*
by *simp*

```
    qed
    have topF: poly.coeff (poly.coeff F ?m) ?n = of-nat ((?m + ?n) choose ?m)
    proof −
      have finC: finite C
        using C by auto
      have mn-card: ?m + ?n = card C
        using C by simp
      have poly.coeff (poly.coeff F ?m) ?n = of-nat (card C choose ?m)
        unfolding F-def by (rule coeff-coeff-prod-sum-factor-top[OF finC mn-card])
      with C show ?thesis
        by simp
    qed
    have topF-nz: poly.coeff (poly.coeff F ?m) ?n ≠ 0
      using of-nat-binomial-ne-zero[OF prime-card, of ?m ?m + ?n] topF mn-lt by simp
    obtain a b where ab: a ∈ A b ∈ B poly (poly F [:a:]) b ≠ 0
      using grid-nonzero-from-top-coeff[OF cardA cardB tdF topF-nz] by blast
    have ab-sum: a + b ∈ sumset A B
      using ab(1,2) by (rule sumsetI)
    have a + b ∈ C
      using ab-sum C(1) by auto
    then have poly (poly F [:a:]) b = 0
    proof −
      have finC: finite C
        using C by auto
      have poly (poly F [:a:]) b = (∏ c∈C. (a + b − c))
        using finC by (simp add: F-def poly-prod poly-sum-factor)
      also have ... = 0
        using finC ‹a + b ∈ C› by (simp add: prod-zero)
      finally show ?thesis .
    qed
    with ab show False
      by simp
  qed
qed

end
theory Vosper-Support
  imports Cauchy-Davenport-Prime-Field
begin
```

7 Arithmetic progressions in prime fields

Vosper's theorem is formulated in terms of arithmetic progressions with a shared common difference. This theory isolates the corresponding progression notation, translation lemmas, and the overlap criterion that recovers a progression from near-invariance under translation.

7.1 Basic progression infrastructure

```
definition ap-segment :: 'a::semiring-1 ⇒ 'a ⇒ nat ⇒ 'a set where
  ap-segment a d n = (λi. a + of-nat i * d) ‘ {..<n}

definition arithmetic-progression :: 'a::semiring-1 set ⇒ bool where
  arithmetic-progression A ⟷ (∃ a d n. A = ap-segment a d n)

lemma arithmetic-progressionI:
  A = ap-segment a d n ⟹ arithmetic-progression A
  unfolding arithmetic-progression-def by blast
```

lemma *arithmetic-progressionE*:
assumes *arithmetic-progression A*
obtains *a d n* **where** *A* = *ap-segment a d n*
using *assms* **by** (*auto simp*: *arithmetic-progression-def*)

lemma *finite-ap-segment* [*simp*]:
finite (*ap-segment a d n*)
by (*simp add*: *ap-segment-def*)

lemma *ap-segment-empty* [*simp*]:
ap-segment a d 0 = {}
by (*simp add*: *ap-segment-def*)

lemma *ap-segment-zero-step*:
fixes *a* :: *'a*::*semiring-1*
assumes *0* < *n*
shows *ap-segment a 0 n* = {*a*}
proof
show *ap-segment a 0 n* ⊆ {*a*}
by (*auto simp*: *ap-segment-def*)
next
have *a* + *of-nat 0* ∗ *0* ∈ *ap-segment a 0 n*
using *assms* **by** (*auto simp*: *ap-segment-def*)
then show {*a*} ⊆ *ap-segment a 0 n*
by *simp*
qed

lemma *ap-segment-translate*:
fixes *a d c* :: *'a*::*semiring-1*
shows *translate c* (*ap-segment a d n*) = *ap-segment* (*c* + *a*) *d n*
proof
show *translate c* (*ap-segment a d n*) ⊆ *ap-segment* (*c* + *a*) *d n*
by (*auto simp*: *translate-def ap-segment-def ac-simps*)
next
show *ap-segment* (*c* + *a*) *d n* ⊆ *translate c* (*ap-segment a d n*)
proof
fix *x*
assume *x* ∈ *ap-segment* (*c* + *a*) *d n*
then obtain *i* **where** *i-lt*: *i* < *n* **and** *x*: *x* = (*c* + *a*) + *of-nat i* ∗ *d*
by (*auto simp*: *ap-segment-def*)
have *a* + *of-nat i* ∗ *d* ∈ *ap-segment a d n*
using *i-lt* **by** (*auto simp*: *ap-segment-def*)
moreover have *x* = *c* + (*a* + *of-nat i* ∗ *d*)
using *x* **by** (*simp add*: *ac-simps*)
ultimately show *x* ∈ *translate c* (*ap-segment a d n*)
by (*auto simp*: *translate-def*)
qed
qed

lemma *arithmetic-progression-translate* [*simp*]:
fixes *c* :: *'a*::*ring-1*
shows *arithmetic-progression* (*translate c A*) ⟷ *arithmetic-progression A*
proof
assume *arithmetic-progression* (*translate c A*)
then obtain *a d n* **where** *eq*: *translate c A* = *ap-segment a d n*
unfolding *arithmetic-progression-def* **by** *blast*
have *A* = *translate* (− *c*) (*translate c A*)
proof
show *A* ⊆ *translate* (− *c*) (*translate c A*)

```
    proof
      fix x
      assume x ∈ A
      then have cx-in: c + x ∈ translate c A
        by (auto simp: translate-def)
      have x-eq: x = − c + (c + x)
        by simp
      show x ∈ translate (− c) (translate c A)
      proof (unfold translate-def, rule image-eqI[where x = c + x])
        show c + x ∈ (+) c ‘ A
          using cx-in unfolding translate-def by simp
        show x = − c + (c + x)
          using x-eq by simp
      qed
    qed
  next
    show translate (− c) (translate c A) ⊆ A
    proof
      fix x
      assume x ∈ translate (− c) (translate c A)
      then obtain y where y-in: y ∈ translate c A and x: x = − c + y
        by (auto simp: translate-def)
      then obtain z where z-in: z ∈ A and y: y = c + z
        by (auto simp: translate-def)
      from x y have x = z
        by simp
      with z-in show x ∈ A
        by simp
    qed
  qed
  also have ... = ap-segment ((− c) + a) d n
    using eq by (simp add: ap-segment-translate)
  finally show arithmetic-progression A
    unfolding arithmetic-progression-def by blast
next
  assume arithmetic-progression A
  then obtain a d n where A = ap-segment a d n
    unfolding arithmetic-progression-def by blast
  then show arithmetic-progression (translate c A)
    unfolding arithmetic-progression-def
    by (intro exI[of - c + a] exI[of - d] exI[of - n]) (simp add: ap-segment-translate)
qed

lemma sumset-ap-segment-pair-suc:
  fixes a d :: 'a::semiring-1
  shows sumset (ap-segment a d (Suc n)) {0, d} = ap-segment a d (Suc (Suc n))
proof
  show sumset (ap-segment a d (Suc n)) {0, d} ⊆ ap-segment a d (Suc (Suc n))
  proof
    fix x
    assume x ∈ sumset (ap-segment a d (Suc n)) {0, d}
    then obtain y b where
        y-in: y ∈ ap-segment a d (Suc n)
      and b-in: b ∈ {0, d}
      and x-eq: x = y + b
      by (rule sumsetE)
    from y-in obtain i where i-lt: i < Suc n and y-eq: y = a + of-nat i * d
      by (auto simp: ap-segment-def)
    show x ∈ ap-segment a d (Suc (Suc n))
```

```
    proof (cases b = 0)
      case True
      have i < Suc (Suc n)
        using i-lt by simp
      with True x-eq y-eq show ?thesis
        by (auto simp: ap-segment-def)
    next
      case False
      with b-in have b = d
        by simp
      with x-eq y-eq have x = a + of-nat (Suc i) * d
        by (simp add: algebra-simps)
      moreover have Suc i < Suc (Suc n)
        using i-lt by simp
      ultimately show ?thesis
      proof −
        have a + of-nat (Suc i) * d ∈ ap-segment a d (Suc (Suc n))
        proof (unfold ap-segment-def, rule image-eqI[where x = Suc i])
          show Suc i ∈ {..<Suc (Suc n)}
            using i-lt by simp
          show a + of-nat (Suc i) * d = a + of-nat (Suc i) * d
            by simp
        qed
        then show ?thesis
          using ‹x = a + of-nat (Suc i) * d› by simp
      qed
    qed
  qed
next
  show ap-segment a d (Suc (Suc n)) ⊆ sumset (ap-segment a d (Suc n)) {0, d}
  proof
    fix x
    assume x ∈ ap-segment a d (Suc (Suc n))
    then obtain j where j-lt: j < Suc (Suc n) and x-eq: x = a + of-nat j * d
      by (auto simp: ap-segment-def)
    show x ∈ sumset (ap-segment a d (Suc n)) {0, d}
    proof (cases j < Suc n)
      case True
      have a + of-nat j * d ∈ ap-segment a d (Suc n)
        using True by (auto simp: ap-segment-def)
      then have (a + of-nat j * d) + 0 ∈ sumset (ap-segment a d (Suc n)) {0, d}
        by (rule sumsetI) simp-all
      then show ?thesis
        using x-eq by simp
    next
      case False
      with j-lt have j-eq: j = Suc n
        by simp
      have a + of-nat n * d ∈ ap-segment a d (Suc n)
        by (auto simp: ap-segment-def)
      moreover have x = (a + of-nat n * d) + d
        using x-eq j-eq by (simp add: algebra-simps)
      ultimately show ?thesis
      proof −
        have (a + of-nat n * d) + d ∈ sumset (ap-segment a d (Suc n)) {0, d}
          using ‹a + of-nat n * d ∈ ap-segment a d (Suc n)›
          by (rule sumsetI) simp-all
        then show ?thesis
          using ‹x = (a + of-nat n * d) + d› by simp
```

```
    qed
  qed
 qed
qed

lemma of-nat-eq-below-prime-card:
 fixes i j :: nat
 assumes prime-card: prime (card (UNIV :: 'a::finite-field set))
 assumes i-lt: i < card (UNIV :: 'a set)
 assumes j-lt: j < card (UNIV :: 'a set)
 assumes eq: (of-nat i :: 'a) = of-nat j
 shows i = j
proof (cases i j rule: linorder-cases)
 case less
 have CHAR('a) dvd (j - i)
  using eq less by (simp add: of-nat-eq-iff-char-dvd)
 then have card-dvd: card (UNIV :: 'a set) dvd (j - i)
  by (simp add: prime-card-eq-char[OF prime-card])
 have j - i > 0
  using less by simp
 then have card (UNIV :: 'a set) ≤ j - i
  using card-dvd prime-card by (meson dvd-imp-le prime-gt-0-nat)
 moreover have j - i < card (UNIV :: 'a set)
  using i-lt j-lt less by simp
 ultimately show ?thesis
  by simp
next
 case greater
 have CHAR('a) dvd (i - j)
 proof -
  have (of-nat j :: 'a) = of-nat i
   using eq by simp
  with greater show ?thesis
   by (simp add: of-nat-eq-iff-char-dvd)
 qed
 then have card-dvd: card (UNIV :: 'a set) dvd (i - j)
  by (simp add: prime-card-eq-char[OF prime-card])
 have i - j > 0
  using greater by simp
 then have card (UNIV :: 'a set) ≤ i - j
  using card-dvd prime-card by (meson dvd-imp-le prime-gt-0-nat)
 moreover have i - j < card (UNIV :: 'a set)
  using i-lt j-lt greater by simp
 ultimately show ?thesis
  by simp
qed simp

lemma inj-on-ap-segment-index:
 fixes a d :: 'a::finite-field
 assumes prime-card: prime (card (UNIV :: 'a::finite-field set))
 assumes d-nonzero: d ≠ 0
 assumes n-le: n ≤ card (UNIV :: 'a set)
 shows inj-on (λi. a + of-nat i * d) {..<n}
proof (rule inj-onI)
 fix i j
 assume i-in: i ∈ {..<n}
 assume j-in: j ∈ {..<n}
 assume eq: a + of-nat i * d = a + of-nat j * d
 have (of-nat i :: 'a) = of-nat j
```

```
  proof −
    have ((of-nat i :: 'a) − of-nat j) ∗ d = 0
      using eq by (simp add: algebra-simps)
    with d-nonzero have (of-nat i :: 'a) − of-nat j = 0
      by auto
    then show ?thesis
      by simp
  qed
  then show i = j
    using i-in j-in n-le prime-card by (intro of-nat-eq-below-prime-card) auto
qed

lemma card-ap-segment:
  fixes a d :: 'a::finite-field
  assumes prime-card: prime (card (UNIV :: 'a::finite-field set))
  assumes d-nonzero: d ≠ 0
  assumes n-le: n ≤ card (UNIV :: 'a set)
  shows card (ap-segment a d n) = n
  by (simp add: ap-segment-def card-image inj-on-ap-segment-index[OF prime-card d-nonzero n-le])

lemma arithmetic-progression-obtain-card:
  fixes A :: 'a::finite-field set
  assumes prime-card: prime (card (UNIV :: 'a set))
  assumes ap: arithmetic-progression A
  assumes card-ge2: 2 ≤ card A
  assumes small: card A < card (UNIV :: 'a set)
  obtains a d where d ≠ 0 A = ap-segment a d (card A)
proof −
  obtain a d n where A-eq: A = ap-segment a d n
    using ap by (rule arithmetic-progressionE)
  have n-pos: 0 < n
  proof (rule ccontr)
    assume ¬ 0 < n
    then have n = 0
      by simp
    with A-eq card-ge2 show False
      by simp
  qed
  have d-nonzero: d ≠ 0
  proof
    assume d = 0
    with A-eq n-pos have A = {a}
      by (simp add: ap-segment-zero-step)
    with card-ge2 show False
      by simp
  qed
  let ?p = card (UNIV :: 'a set)
  have n-lt-p: n < ?p
  proof (rule ccontr)
    assume ¬ n < ?p
    then have p-le-n: ?p ≤ n
      by simp
    have ap-segment a d ?p ⊆ ap-segment a d n
      using p-le-n by (auto simp: ap-segment-def)
    moreover have ap-segment a d ?p = UNIV
    proof −
      have card (ap-segment a d ?p) = ?p
        by (rule card-ap-segment[OF prime-card d-nonzero]) simp
      moreover have ap-segment a d ?p ⊆ UNIV
```

```
      by simp
    ultimately show ?thesis
      using finite-UNIV by (intro card-subset-eq) auto
  qed
  ultimately have UNIV ⊆ A
    using A-eq by simp
  then have A = UNIV
    by auto
  with small show False
    by simp
 qed
 have n-le-p: n ≤ ?p
  using n-lt-p by simp
 have card A = n
  using A-eq card-ap-segment[OF prime-card d-nonzero n-le-p, of a] by simp
 then have A = ap-segment a d (card A)
  using A-eq by simp
 then show ?thesis
  using d-nonzero that by blast
qed

lemma step-image-UNIV:
 fixes a d :: 'a::finite-field
 assumes prime-card: prime (card (UNIV :: 'a::finite-field set))
 assumes d-nonzero: d ≠ 0
 shows (λi. a + of-nat i * d) ` {..<card (UNIV :: 'a set)} = UNIV
proof −
 have inj: inj-on (λi. a + of-nat i * d) {..<card (UNIV :: 'a set)}
  by (rule inj-on-ap-segment-index[OF prime-card d-nonzero]) simp
 have card ((λi. a + of-nat i * d) ` {..<card (UNIV :: 'a set)}) = card (UNIV :: 'a set)
  by (simp add: card-image inj)
 moreover have (λi. a + of-nat i * d) ` {..<card (UNIV :: 'a set)} ⊆ UNIV
  by simp
 ultimately show ?thesis
  using finite-UNIV by (intro card-subset-eq) auto
qed

lemma predecessor-closed-initial-segment:
 fixes S :: nat set
 assumes fin: finite S
 assumes zero: 0 ∈ S
 assumes pred: ⋀k. Suc k ∈ S ⟹ k ∈ S
 shows S = {..<card S}
proof
 have init: {..<Suc k} ⊆ S if k ∈ S for k
  using that
 proof (induction k)
  case 0
  then show ?case
   by auto
 next
  case (Suc k)
  have k ∈ S
   using pred Suc.prems by blast
  then have {..<Suc k} ⊆ S
   by (rule Suc.IH)
  show ?case
  proof
   fix x
```

```
      assume x-lt: x ∈ {..<Suc (Suc k)}
      show x ∈ S
      proof (cases x = Suc k)
        case True
        with Suc.prems show ?thesis
          by simp
      next
        case False
        with x-lt have x < Suc k
          by simp
        with ‹{..<Suc k} ⊆ S› show ?thesis
          by auto
      qed
    qed
  qed
  show S ⊆ {..<card S}
  proof
    fix k
    assume k ∈ S
    then have {..<Suc k} ⊆ S
      by (rule init)
    then have Suc k ≤ card S
    proof −
      have card {..<Suc k} = Suc k
        by simp
      moreover have card {..<Suc k} ≤ card S
        using fin ‹{..<Suc k} ⊆ S› by (intro card-mono) auto
      ultimately show ?thesis
        by simp
    qed
    then show k ∈ {..<card S}
      by simp
  qed
next
  show {..<card S} ⊆ S
  proof
    have down: i ∈ S if i ≤ j j ∈ S for i j
      using that
    proof (induction j arbitrary: i)
      case 0
      then show ?case
        by simp
    next
      case (Suc j)
      show ?case
      proof (cases i = Suc j)
        case True
        with Suc.prems show ?thesis
          by simp
      next
        case False
        with Suc.prems have i-le-j: i ≤ j
          by simp
        have j ∈ S
          using pred Suc.prems(2) by simp
        from Suc.IH[OF i-le-j this] show ?thesis .
      qed
    qed
    fix k
```

```
    assume k-lt: k ∈ {..<card S}
    show k ∈ S
    proof (rule ccontr)
      assume k-notin: k ∉ S
      have S ⊆ {..<k}
      proof
        fix j
        assume j-in: j ∈ S
        show j ∈ {..<k}
        proof (rule ccontr)
          assume j ∉ {..<k}
          then have k ≤ j
            by simp
          then have k ∈ S
            using j-in by (rule down)
          with k-notin show False
            by simp
        qed
      qed
      then have card S ≤ k
      proof −
        have card S ≤ card {..<k}
          using fin ‹S ⊆ {..<k}› by (intro card-mono) auto
        then show ?thesis
          by simp
      qed
      with k-lt show False
        by simp
    qed
  qed
qed
```

7.2 Recovering a progression from maximal overlap

The following lemmas package the structural step used repeatedly in the Vosper proof: if a set is almost preserved by translation with a nonzero step, then it must itself be an arithmetic progression with that step.

```
lemma ap-segment-from-maximal-shift-overlap:
  fixes A :: 'a::finite-field set
  assumes prime-card: prime (card (UNIV :: 'a set))
  assumes d-nonzero: d ≠ 0
  assumes small: card A < card (UNIV :: 'a set)
  assumes overlap: card (A ∩ translate d A) = card A − 1
  obtains a where A = ap-segment a d (card A)
proof (cases A = {})
  case True
  then show ?thesis
  proof −
    have A = ap-segment 0 d (card A)
      using True by simp
    then show ?thesis
      by (rule that)
  qed
next
  case False
  have finA: finite A
    using finite-subset[of A UNIV :: 'a set] by simp
  have cardA-pos: 0 < card A
```

```
  using False finA by auto
have one-start: card (A − translate d A) = 1
proof −
  have A ∩ translate d A ⊆ A
    by simp
  then have card (A − (A ∩ translate d A)) = card A − card (A ∩ translate d A)
    using finA by (simp add: card-Diff-subset)
  moreover have A − (A ∩ translate d A) = A − translate d A
    by auto
  ultimately show ?thesis
    using overlap cardA-pos by simp
qed
then obtain a where a-start: A − translate d A = {a}
  by (meson card-1-singletonE)
then have a-in: a ∈ A and a-notin: a ∉ translate d A
  by auto
define p where p = card (UNIV :: 'a set)
define f where f i = a + of-nat i * d for i
define S where S = {i ∈ {..<p}. f i ∈ A}
have p-pos: 0 < p
  unfolding p-def using prime-card prime-gt-0-nat by blast
have image-all: f ‘ {..<p} = UNIV
  unfolding p-def f-def by (rule step-image-UNIV[OF prime-card d-nonzero])
have A-as-image: A = f ‘ S
  unfolding S-def
  using image-all by (auto simp: f-def)
have inj-f: inj-on f {..<p}
  unfolding p-def f-def
  by (rule inj-on-ap-segment-index[OF prime-card d-nonzero]) simp
have finS: finite S
  unfolding S-def by simp
have cardS: card S = card A
proof −
  have S-subset: S ⊆ {..<p}
    unfolding S-def by auto
  have inj-fS: inj-on f S
    using inj-f S-subset by (rule inj-on-subset)
  have card (f ‘ S) = card S
    using finS inj-fS by (simp add: card-image)
  with A-as-image show ?thesis
    by simp
qed
have zero-in-S: 0 ∈ S
  using a-in p-pos by (simp add: S-def f-def)
have pred-closed: Suc k ∈ S ⟹ k ∈ S for k
proof −
  assume suc-in: Suc k ∈ S
  then have sk-lt: Suc k < p
    by (simp add: S-def)
  have fk-in: f (Suc k) ∈ A
    using suc-in by (simp add: S-def)
  have fk-neq-a: f (Suc k) ≠ a
  proof
    assume f (Suc k) = a
    then have f (Suc k) = f 0
      by (simp add: f-def)
    have suc-mem: Suc k ∈ {..<p}
      using sk-lt by simp
    have zero-mem: 0 ∈ {..<p}
```

```
      using p-pos by simp
    from inj-f suc-mem zero-mem ‹f (Suc k) = f 0› have Suc k = 0
      unfolding inj-on-def by blast
    then show False
      by simp
  qed
  have f (Suc k) ∈ A − translate d A ⟹ False
    using a-start fk-neq-a by auto
  then have fk-shift: f (Suc k) ∈ translate d A
    using fk-in by blast
  then obtain y where y-in: y ∈ A and fy: f (Suc k) = d + y
    by (auto simp: translate-iff)
  have y = f k
    using fy by (simp add: f-def algebra-simps)
  with y-in show k ∈ S
    using sk-lt by (simp add: S-def)
 qed
 have S-init: S = {..<card S}
  by (rule predecessor-closed-initial-segment[OF finS zero-in-S pred-closed])
 have A = f ‘ {..<card A}
  using A-as-image S-init cardS by simp
 also have ... = ap-segment a d (card A)
  by (simp add: ap-segment-def f-def)
 finally have A = ap-segment a d (card A) .
 then show ?thesis
  using that by blast
qed

lemma arithmetic-progression-from-maximal-shift-overlap:
 fixes A :: 'a::finite-field set
 assumes prime-card: prime (card (UNIV :: 'a set))
 assumes d-nonzero: d ≠ 0
 assumes small: card A < card (UNIV :: 'a set)
 assumes overlap: card (A ∩ translate d A) = card A − 1
 shows arithmetic-progression A
proof −
 obtain a where A = ap-segment a d (card A)
  by (rule ap-segment-from-maximal-shift-overlap[OF prime-card d-nonzero small overlap])
 then show ?thesis
  unfolding arithmetic-progression-def by blast
qed

end
theory Vosper-Prime-Field
 imports Vosper-Support
begin
```

8 Vosper over prime fields

This theory formalizes Vosper's theorem in the prime-field setting. The proof is organized around Davenport transforms, first deriving one-sided progression results under a normalization hypothesis and then combining the normalized arguments to obtain the full shared-difference conclusion.

8.1 Davenport transforms and translation lemmas

```
definition davenport-transform ::
 'a::ring-1 set ⇒ 'a set ⇒ 'a ⇒ 'a set
where
```

```
  davenport-transform A B e = {b ∈ B. e − b ∉ sumset A B}

definition davenport-remainder ::
  'a::ring-1 set ⇒ 'a set ⇒ 'a ⇒ 'a set
where
  davenport-remainder A B e = {b ∈ B. e − b ∈ sumset A B}

lemma sumset-mono-left:
  assumes A ⊆ A'
  shows sumset A B ⊆ sumset A' B
  using assms by (auto simp: sumset-def)

lemma sumset-mono-right:
  assumes B ⊆ B'
  shows sumset A B ⊆ sumset A B'
  using assms by (auto simp: sumset-def)

lemma sumset-translate-right:
  fixes A B :: ('a::comm-monoid-add) set
  shows sumset A (translate c B) = translate c (sumset A B)
proof
  show sumset A (translate c B) ⊆ translate c (sumset A B)
  proof
    fix x
    assume x ∈ sumset A (translate c B)
    then obtain a b where a-in: a ∈ A and b-in: b ∈ B and x-eq: x = a + (c + b)
      by (auto simp: sumset-def translate-def)
    have a + b ∈ sumset A B
      using a-in b-in by (rule sumsetI)
    then show x ∈ translate c (sumset A B)
    proof −
      have x = c + (a + b)
        using x-eq by (simp add: ac-simps)
      then show ?thesis
        using ‹a + b ∈ sumset A B› by (auto simp: translate-def)
    qed
  qed
next
  show translate c (sumset A B) ⊆ sumset A (translate c B)
  proof
    fix x
    assume x ∈ translate c (sumset A B)
    then obtain y where y-in: y ∈ sumset A B and x-eq: x = c + y
      by (auto simp: translate-def)
    then obtain a b where a-in: a ∈ A and b-in: b ∈ B and y-eq: y = a + b
      by (auto simp: sumset-def)
    have c + b ∈ translate c B
      using b-in by (auto simp: translate-def)
    then show x ∈ sumset A (translate c B)
      using a-in x-eq y-eq by (auto simp: sumset-def ac-simps)
  qed
qed

lemma sumset-translate-left:
  fixes A B :: ('a::comm-monoid-add) set
  shows sumset (translate c A) B = translate c (sumset A B)
proof
  show sumset (translate c A) B ⊆ translate c (sumset A B)
  proof
```

```
    fix x
    assume x ∈ sumset (translate c A) B
    then obtain a b where a-in: a ∈ A and b-in: b ∈ B and x-eq: x = (c + a) + b
      by (auto simp: sumset-def translate-def)
    have a + b ∈ sumset A B
      using a-in b-in by (rule sumsetI)
    then show x ∈ translate c (sumset A B)
    proof −
      have x = c + (a + b)
        using x-eq by (simp add: ac-simps)
      then show ?thesis
        using ‹a + b ∈ sumset A B› by (auto simp: translate-def)
    qed
  qed
next
  show translate c (sumset A B) ⊆ sumset (translate c A) B
  proof
    fix x
    assume x ∈ translate c (sumset A B)
    then obtain y where y-in: y ∈ sumset A B and x-eq: x = c + y
      by (auto simp: translate-def)
    then obtain a b where a-in: a ∈ A and b-in: b ∈ B and y-eq: y = a + b
      by (auto simp: sumset-def)
    have c + a ∈ translate c A
      using a-in by (auto simp: translate-def)
    then show x ∈ sumset (translate c A) B
    proof −
      have x = (c + a) + b
        using x-eq y-eq by (simp add: ac-simps)
      then show ?thesis
        using ‹c + a ∈ translate c A› b-in by (auto simp: sumset-def)
    qed
  qed
qed

lemma card-uminus-image-eq [simp]:
  fixes A :: ('a::ab-group-add) set
  assumes finite A
  shows card ((uminus) ` A) = card A
  using assms by (simp add: card-image)

lemma card-2-zero-obtain:
  fixes B :: ('a::zero) set
  assumes fin: finite B
  assumes zero-in: 0 ∈ B
  assumes card2: card B = 2
  obtains d where d ≠ 0 B = {0, d}
proof −
  have card (B − {0}) = 1
    using assms by simp
  then obtain d where diff: B − {0} = {d}
    by (meson card-1-singletonE)
  have B = {0, d}
    using diff zero-in by auto
  moreover have d ≠ 0
    using card2 calculation by auto
  ultimately show ?thesis
    using that by blast
qed
```

lemma *davenport-partition*:
davenport-transform A B e ∪ *davenport-remainder A B e* = *B*
davenport-transform A B e ∩ *davenport-remainder A B e* = {}
by (*auto simp*: *davenport-transform-def davenport-remainder-def*)

lemma *davenport-transform-subset* [*simp*]:
davenport-transform A B e ⊆ *B*
by (*auto simp*: *davenport-transform-def*)

lemma *davenport-remainder-subset* [*simp*]:
davenport-remainder A B e ⊆ *B*
by (*auto simp*: *davenport-remainder-def*)

lemma *card-lt-univ-if-not-univ*:
fixes *S* :: *'a*::*finite set*
assumes *S* ≠ *UNIV*
shows *card S* < *card* (*UNIV* :: *'a set*)
proof (*rule ccontr*)
assume ¬ *card S* < *card* (*UNIV* :: *'a set*)
moreover have *card S* ≤ *card* (*UNIV* :: *'a set*)
by (*intro card-mono*) *auto*
ultimately have *card S* = *card* (*UNIV* :: *'a set*)
by *linarith*
then have *S* = *UNIV*
using *finite-UNIV* **by** (*intro card-subset-eq*) *auto*
with *assms* **show** *False*
by *simp*
qed

8.2 One-sided progression results

The first main step shows that equality in Cauchy-Davenport, together with a normalization such as *0* ∈ *B*, forces the opposite set to be an arithmetic progression.

theorem *vosper-progression-left-zero*:
fixes *A B* :: *'a*::*finite-field set*
assumes *prime-card*: *prime* (*card* (*UNIV* :: *'a set*))
assumes *zero-in-B*: *0* ∈ *B*
assumes *B-ge2*: *2* ≤ *card B*
assumes *comp-ge2*: *2* ≤ *card* (*UNIV* − *sumset A B*)
assumes *eq*: *card* (*sumset A B*) = *card A* + *card B* − *1*
shows *arithmetic-progression A*
proof −
let *?p* = *card* (*UNIV* :: *'a set*)
have *aux*:
⋀*n*::*nat*. ⋀*A B*::*'a set*.
card B = *n* ⟹
0 ∈ *B* ⟹
2 ≤ *card B* ⟹
2 ≤ *card* (*UNIV* − *sumset A B*) ⟹
card (*sumset A B*) = *card A* + *card B* − *1* ⟹
arithmetic-progression A
proof −
fix *n* :: *nat*
fix *A B* :: *'a set*
show *card B* = *n* ⟹
0 ∈ *B* ⟹
2 ≤ *card B* ⟹

```
  2 ≤ card (UNIV − sumset A B) ⟹
  card (sumset A B) = card A + card B − 1 ⟹
  arithmetic-progression A
proof (induction n arbitrary: A B rule: less-induct)
  case (less n)
  note IH = less.IH
  assume cardB: card B = n
  assume zero-in-B: 0 ∈ B
  assume B-ge2: 2 ≤ card B
  assume comp-ge2: 2 ≤ card (UNIV − sumset A B)
  assume eq: card (sumset A B) = card A + card B − 1

  have finA: finite A and finB: finite B
    by simp-all
  have B-nonempty: B ≠ {}
    using B-ge2 by auto
  have A-nonempty: A ≠ {}
    using eq B-ge2 by auto
  have A-subset-sumset: A ⊆ sumset A B
  proof
    fix x
    assume x-in: x ∈ A
    have x + 0 ∈ sumset A B
      using x-in zero-in-B by (rule sumsetI)
    then show x ∈ sumset A B
      by simp
  qed
  have sumset-not-full: sumset A B ≠ UNIV
    using comp-ge2 by auto
  have sumset-small: card (sumset A B) < ?p
  proof (rule ccontr)
    assume ¬ card (sumset A B) < ?p
    moreover have card (sumset A B) ≤ ?p
      using finite-UNIV finite-sumset[OF finA finB] by (intro card-mono) auto
    ultimately have card (sumset A B) = ?p
      by linarith
    then have sumset A B = UNIV
      using finite-UNIV by (intro card-subset-eq) auto
    with sumset-not-full show False
      by simp
  qed

  show arithmetic-progression A
  proof (cases card B = 2)
    case True
    obtain d where d-nonzero: d ≠ 0 and B-eq: B = {0, d}
      by (rule card-2-zero-obtain[OF finB zero-in-B True])
    have sumset-eq: sumset A B = A ∪ translate d A
      using B-eq by (simp add: sumset-pair-zero)
    have card-overlap: card (A ∩ translate d A) = card A − 1
    proof −
      have fin-translate: finite (translate d A)
        by simp
      have card (sumset A B) = card (A ∪ translate d A)
        using sumset-eq by simp
      also have ... = card A + card (translate d A) − card (A ∩ translate d A)
      proof −
        have card A + card (translate d A) =
            card (A ∪ translate d A) + card (A ∩ translate d A)
```

```
        using finA fin-translate by (rule card-Un-Int)
      then show ?thesis
        by linarith
    qed
    also have ... = card A + card A − card (A ∩ translate d A)
      using finA by (simp add: card-translate-eq)
    finally show ?thesis
      using eq True by simp
  qed
  have smallA: card A < ?p
    using A-subset-sumset sumset-small finA finite-sumset[OF finA finB]
    by (meson card-mono le-less-trans)
  from arithmetic-progression-from-maximal-shift-overlap
    [OF prime-card d-nonzero smallA card-overlap]
  show ?thesis .
next
  case False
  have cardB-gt2: 2 < card B
    using B-ge2 False by linarith
  define S where S = sumset A B
  define T where T = sumset S B
  define E where E = T − S

  have finS: finite S and finT: finite T and finE: finite E
    unfolding S-def T-def E-def by auto
  have S-nonempty: S ≠ {}
    using A-nonempty B-nonempty unfolding S-def by auto
  have S-subset-T: S ⊆ T
  proof
    fix x
    assume x-in: x ∈ S
    have x + 0 ∈ T
      using x-in zero-in-B unfolding T-def by (rule sumsetI)
    then show x ∈ T
      by simp
  qed
  have T-lb: card T ≥ min ?p (card S + card B − 1)
  proof −
    have card (sumset S B) ≥ min ?p (card S + card B − 1)
      using cauchy-davenport-prime-field[where A = S and B = B, OF prime-card S-nonempty
B-nonempty]
      by simp
    then show ?thesis
      unfolding T-def .
  qed
  have min-gt: min ?p (card S + card B − 1) > card S
  proof (cases ?p < card S + card B − 1)
    case True
    then show ?thesis
      using sumset-small unfolding S-def by simp
  next
    case False
    then have min ?p (card S + card B − 1) = card S + card B − 1
      by simp
    with B-ge2 show ?thesis
      by simp
  qed
  have T-gt: card T > card S
    using T-lb min-gt by linarith
```

```
have cardE: card E = card T − card S
  using S-subset-T finT unfolding E-def by (simp add: card-Diff-subset)
have E-nonempty: E ≠ {}
  using T-gt cardE by auto

show ?thesis
proof (rule ccontr)
  assume not-ap: ¬ arithmetic-progression A

  have transform-singleton:
    davenport-transform A B e = {0} if e-in: e ∈ E for e
  proof −
    let ?D = davenport-transform A B e
    let ?R = davenport-remainder A B e

    have e-notin-S: e ∉ S
      using e-in unfolding E-def by auto
    have zero-in-D: 0 ∈ ?D
      using zero-in-B e-notin-S unfolding davenport-transform-def S-def by auto
    have D-nonempty: ?D ≠ {}
      using zero-in-D by auto
    have R-nonempty: ?R ≠ {}
    proof −
      from e-in have e ∈ T
        unfolding E-def by auto
      then obtain s b where s-in: s ∈ S and b-in: b ∈ B and e-eq: e = s + b
        unfolding T-def by (auto simp: sumset-def)
      have b ∈ ?R
        using s-in b-in e-eq unfolding davenport-remainder-def S-def by simp
      then show ?thesis
        by auto
    qed
    have part: ?D ∪ ?R = B ?D ∩ ?R = {}
      by (rule davenport-partition)+
    have finD: finite ?D and finR: finite ?R
      by auto
    have cardD-lt: card ?D < card B
    proof −
      have card (?D ∪ ?R) = card ?D + card ?R
        using finD finR part(2) by (simp add: card-Un-disjoint)
      then have card B = card ?D + card ?R
        using part(1) by simp
      moreover have 0 < card ?R
        using R-nonempty finR by auto
      ultimately show ?thesis
        by linarith
    qed
    have left-subset: sumset A ?D ⊆ S
    proof
      fix x
      assume x-in: x ∈ sumset A ?D
      then obtain a b where a-in: a ∈ A and b-in: b ∈ ?D and x-eq: x = a + b
        by (auto simp: sumset-def)
      have b ∈ B
        using b-in unfolding davenport-transform-def by auto
      then show x ∈ S
        using a-in x-eq unfolding S-def by (auto simp: sumset-def)
    qed
    have right-subset: (λb. e − b) ‘ ?R ⊆ S
```

unfolding *davenport-remainder-def S-def* **by** *auto*
have *disj-subset*: *sumset A ?D* ∩ (λ*b*. *e* − *b*) ‘ *?R* ⊆ {}
proof
fix *x*
assume *x-in*: *x* ∈ *sumset A ?D* ∩ (λ*b*. *e* − *b*) ‘ *?R*
then obtain *a d r* **where**
a-in: *a* ∈ *A* **and** *d-in*: *d* ∈ *?D* **and** *r-in*: *r* ∈ *?R*
and *x-eq1*: *x* = *a* + *d* **and** *x-eq2*: *x* = *e* − *r*
by (*auto simp*: *sumset-def*)
have *eq-ed*: *e* − *d* = *a* + *r*
using *x-eq1 x-eq2* **by** (*simp add*: *algebra-simps*)
have *ar-in-S*: *a* + *r* ∈ *S*
using *a-in r-in* **unfolding** *S-def davenport-remainder-def* **by** (*auto intro*: *sumsetI*)
have *ed-notin*: *e* − *d* ∉ *sumset A B*
using *d-in* **unfolding** *davenport-transform-def* **by** *auto*
have *ed-in*: *e* − *d* ∈ *sumset A B*
using *eq-ed ar-in-S* **unfolding** *S-def* **by** *simp*
show *x* ∈ {}
using *ed-notin ed-in* **by** *blast*
qed
have *disj*: *sumset A ?D* ∩ (λ*b*. *e* − *b*) ‘ *?R* = {}
using *disj-subset* **by** *blast*
have *card-right*: *card* ((λ*b*. *e* − *b*) ‘ *?R*) = *card ?R*
proof −
have *inj-on* (λ*b*. *e* − *b*) *?R*
by (*rule inj-onI*) *simp*
with *finR* **show** *?thesis*
by (*simp add*: *card-image*)
qed
have *cardS-lb*: *card S* ≥ *card* (*sumset A ?D*) + *card ?R*
proof −
have *card-union-le*: *card S* ≥ *card* (*sumset A ?D* ∪ (λ*b*. *e* − *b*) ‘ *?R*)
using *left-subset right-subset finS finite-sumset*[*OF finA finD*]
by (*intro card-mono*) *auto*
have *fin-image*: *finite* ((λ*b*. *e* − *b*) ‘ *?R*)
using *finR* **by** *auto*
have *card-union*:
card (*sumset A ?D* ∪ (λ*b*. *e* − *b*) ‘ *?R*) =
card (*sumset A ?D*) + *card* ((λ*b*. *e* − *b*) ‘ *?R*)
by (*rule card-Un-disjoint*[*OF finite-sumset*[*OF finA finD*] *fin-image disj*])
have *card-union′*:
card (*sumset A ?D* ∪ (λ*b*. *e* − *b*) ‘ *?R*) =
card (*sumset A ?D*) + *card ?R*
using *card-union card-right* **by** *simp*
from *card-union-le card-union′* **show** *?thesis*
by *simp*
qed
have *cardB-split*: *card B* = *card ?D* + *card ?R*
proof −
have *card* (*?D* ∪ *?R*) = *card ?D* + *card ?R*
using *finD finR part*(*2*) **by** (*simp add*: *card-Un-disjoint*)
then show *?thesis*
using *part*(*1*) **by** *simp*
qed
have *upperD*: *card* (*sumset A ?D*) ≤ *card A* + *card ?D* − *1*
using *cardS-lb eq cardB-split* **unfolding** *S-def* **by** *linarith*
have *sumsetD-le-S*: *card* (*sumset A ?D*) ≤ *card S*
using *left-subset finS finite-sumset*[*OF finA finD*] **by** (*intro card-mono*) *auto*
have *sumsetD-small*: *card* (*sumset A ?D*) < *?p*

```
    using sumsetD-le-S sumset-small unfolding S-def by linarith
  have eqD: card (sumset A ?D) = card A + card ?D − 1
  proof −
    have cdD: card (sumset A ?D) ≥ min ?p (card A + card ?D − 1)
      using cauchy-davenport-prime-field[OF prime-card A-nonempty D-nonempty]
      by simp
    have card A + card ?D − 1 < ?p
    proof (rule ccontr)
      assume not-lt: ¬ card A + card ?D − 1 < ?p
      then have min ?p (card A + card ?D − 1) = ?p
        by simp
      with cdD have card (sumset A ?D) ≥ ?p
        by simp
      with sumsetD-small show False
        by simp
    qed
    with cdD upperD show ?thesis
      by simp
  qed
  have compD-ge2: 2 ≤ card (UNIV − sumset A ?D)
  proof −
    have UNIV − S ⊆ UNIV − sumset A ?D
      using left-subset by auto
    then have card (UNIV − S) ≤ card (UNIV − sumset A ?D)
      by (intro card-mono) auto
    with comp-ge2 show ?thesis
      unfolding S-def by linarith
  qed

  show ?thesis
  proof (cases 2 ≤ card ?D)
    case True
    have cardD-lt-n: card ?D < n
      using cardD-lt cardB by simp
    have arithmetic-progression A
      using IH[of card ?D ?D A] cardD-lt-n zero-in-D True compD-ge2 eqD by blast
    with not-ap show ?thesis
      by simp
  next
    case False
    have cardD-pos: 0 < card ?D
      using D-nonempty finD by auto
    have card ?D = 1
      using cardD-pos False by linarith
    then obtain x where D-eq: ?D = {x}
      by (auto simp: card-1-singleton-iff)
    with zero-in-D show ?thesis
      by auto
  qed
qed

let ?Bx = B − {0}
have card-Bx: card ?Bx = card B − 1
  using finB zero-in-B by simp
have Bx-nonempty: ?Bx ≠ {}
proof −
  have 0 < card ?Bx
    using card-Bx cardB-gt2 by linarith
  then show ?thesis
```

```
    using finB by (simp add: card-gt-0-iff)
qed
have A-disj-subset: A ∩ sumset E ((uminus) ‘ ?Bx) ⊆ {}
proof
  fix x
  assume x-in: x ∈ A ∩ sumset E ((uminus) ‘ ?Bx)
  then obtain e b where e-in: e ∈ E and b-in: b ∈ ?Bx and x-eq: x = e − b
    by (auto simp: sumset-def)
  have e = x + b
    using x-eq by simp
  moreover have x + b ∈ S
    using x-in b-in unfolding S-def by (auto intro: sumsetI)
  ultimately have e-in-S: e ∈ S
    by simp
  have e-notin-S: e ∉ S
    using e-in unfolding E-def by auto
  show x ∈ {}
    using e-in-S e-notin-S by blast
qed
have A-disj: A ∩ sumset E ((uminus) ‘ ?Bx) = {}
  using A-disj-subset by blast
have E-minus-subset: sumset E ((uminus) ‘ ?Bx) ⊆ S
proof
  fix x
  assume x-in: x ∈ sumset E ((uminus) ‘ ?Bx)
  then obtain e b where e-in: e ∈ E and b-in: b ∈ ?Bx and x-eq: x = e − b
    by (auto simp: sumset-def)
  have b-notin-D: b ∉ davenport-transform A B e
    using b-in transform-singleton[OF e-in] by auto
  have b-in-B: b ∈ B
    using b-in by auto
  have b ∈ davenport-remainder A B e
    using b-notin-D b-in-B unfolding davenport-transform-def davenport-remainder-def
    by auto
  then have e − b ∈ S
    unfolding davenport-remainder-def S-def by auto
  then show x ∈ S
    using x-eq by simp
qed
have A-union-subset: A ∪ sumset E ((uminus) ‘ ?Bx) ⊆ S
  using A-subset-sumset E-minus-subset unfolding S-def by auto
have Eminus-not-full: sumset E ((uminus) ‘ ?Bx) ≠ UNIV
  using A-nonempty A-disj by auto
have finEminus: finite (sumset E ((uminus) ‘ ?Bx))
  by auto
have card-neg-Bx: card ((uminus) ‘ ?Bx) = card ?Bx
proof −
  have inj-on uminus ?Bx
    by (rule inj-onI) simp
  then show ?thesis
    by (simp add: card-image)
qed
have neg-Bx-nonempty: (uminus) ‘ ?Bx ≠ {}
  using Bx-nonempty by auto
have E-lb: card (sumset E ((uminus) ‘ ?Bx)) ≥ card E + card ?Bx − 1
proof −
  have cdE:
    card (sumset E ((uminus) ‘ ?Bx)) ≥ min ?p (card E + card ((uminus) ‘ ?Bx) − 1)
    using cauchy-davenport-prime-field[OF prime-card E-nonempty neg-Bx-nonempty]
```

```
    by simp
  then have cdE': card (sumset E ((uminus) ` ?Bx)) ≥ min ?p (card E + card ?Bx - 1)
    using card-neg-Bx by simp
  have card (sumset E ((uminus) ` ?Bx)) < ?p
  proof (rule ccontr)
    assume not-lt: ¬ card (sumset E ((uminus) ` ?Bx)) < ?p
    have le-p: card (sumset E ((uminus) ` ?Bx)) ≤ ?p
      using finEminus finite-UNIV by (intro card-mono) auto
    with not-lt have eq-p: card (sumset E ((uminus) ` ?Bx)) = ?p
      by linarith
    then have sumset E ((uminus) ` ?Bx) = UNIV
      using finEminus finite-UNIV by (intro card-subset-eq) auto
    with Eminus-not-full show False
      by simp
  qed
  with cdE' show ?thesis
    by simp
qed
have cardS-ge: card S ≥ card A + card (sumset E ((uminus) ` ?Bx))
proof -
  have card-union-le: card S ≥ card (A ∪ sumset E ((uminus) ` ?Bx))
    using A-union-subset finS finEminus
    by (intro card-mono) auto
  have card-union:
    card (A ∪ sumset E ((uminus) ` ?Bx)) =
     card A + card (sumset E ((uminus) ` ?Bx))
    by (rule card-Un-disjoint[OF finA finEminus A-disj])
  from card-union-le card-union show ?thesis
    by simp
qed
have cardE-le1: card E ≤ 1
  using cardS-ge E-lb eq cardB-gt2 card-Bx unfolding S-def by linarith
have cardE-pos: 0 < card E
  using E-nonempty finE by (simp add: card-gt-0-iff)
have cardE-one: card E = 1
  using cardE-pos cardE-le1 by linarith

show False
proof (cases T = UNIV)
  case True
  have card E = card (UNIV - S)
    using True unfolding E-def by auto
  also have ... ≥ 2
    using comp-ge2 unfolding S-def by simp
  finally show False
    using cardE-one by simp
next
  case False
  have cdT: card T ≥ min ?p (card S + card B - 1)
    unfolding T-def
    using cauchy-davenport-prime-field[OF prime-card S-nonempty B-nonempty]
    by simp
  have T-small: card T < ?p
  proof (rule ccontr)
    assume not-lt: ¬ card T < ?p
    have le-p: card T ≤ ?p
      using finT finite-UNIV by (intro card-mono) auto
    with not-lt have card T = ?p
      by linarith
```

```
        then have T = UNIV
          using finT finite-UNIV by (intro card-subset-eq) auto
        with False show False
          by simp
      qed
      have T-lb': card T ≥ card S + card B - 1
        using cdT T-small by simp
      have cardS-le-T: card S ≤ card T
        using S-subset-T finT by (intro card-mono) auto
      have card T = card S + card E
      proof -
        have card T = card S + (card T - card S)
          using cardS-le-T by simp
        also have ... = card S + card E
          using cardE by simp
        finally show ?thesis .
      qed
      with T-lb' cardE-one have card B ≤ 2
        by linarith
      with cardB-gt2 show False
        by simp
    qed
    qed
  qed
  qed
  qed
  from aux[where n = card B and A = A and B = B] zero-in-B B-ge2 comp-ge2 eq show ?thesis
    by simp
qed

theorem vosper-progression-right-zero:
  fixes A B :: 'a::finite-field set
  assumes prime-card: prime (card (UNIV :: 'a set))
  assumes zero-in-A: 0 ∈ A
  assumes A-ge2: 2 ≤ card A
  assumes comp-ge2: 2 ≤ card (UNIV - sumset A B)
  assumes eq: card (sumset A B) = card A + card B - 1
  shows arithmetic-progression B
proof -
  have comp-ge2': 2 ≤ card (UNIV - sumset B A)
    using comp-ge2 by (simp add: sumset-commute)
  have eq': card (sumset B A) = card B + card A - 1
    using eq by (simp add: sumset-commute add.commute)
  show ?thesis
    by (rule vosper-progression-left-zero[OF prime-card zero-in-A A-ge2 comp-ge2' eq'])
qed

lemma vosper-left-with-right-ap-zero-two:
  fixes A :: 'a::finite-field set
  assumes prime-card: prime (card (UNIV :: 'a set))
  assumes d-nonzero: d ≠ 0
  assumes A-ge2: 2 ≤ card A
  assumes comp-ge2: 2 ≤ card (UNIV - sumset A (ap-segment 0 d (Suc (Suc 0))))
  assumes eq: card (sumset A (ap-segment 0 d (Suc (Suc 0)))) = card A + Suc 0
  shows ∃ a. A = ap-segment a d (card A)
proof -
  let ?p = card (UNIV :: 'a set)
  let ?two = Suc (Suc 0)
  have finA: finite A
```

```
  by simp
have A-nonempty: A ≠ {}
proof
  assume A = {}
  then have card A = 0
    by simp
  then have 2 ≤ (0::nat)
    using A-ge2 by simp
  then show False
    by simp
qed
have sumset-not-full: sumset A (ap-segment 0 d ?two) ≠ UNIV
  using comp-ge2 by auto
have sumset-small: card (sumset A (ap-segment 0 d ?two)) < card (UNIV :: 'a set)
  using sumset-not-full card-lt-univ-if-not-univ[of sumset A (ap-segment 0 d ?two)] by simp
have B-eq: ap-segment 0 d ?two = {0, d}
proof
  show ap-segment 0 d ?two ⊆ {0, d}
  proof
    fix x
    assume x ∈ ap-segment 0 d ?two
    then obtain i where i-lt: i < ?two and x-eq: x = 0 + of-nat i * d
      by (auto simp: ap-segment-def)
    from i-lt have i = 0 ∨ i = 1
      by auto
    with x-eq show x ∈ {0, d}
      by auto
  qed
next
  show {0, d} ⊆ ap-segment 0 d ?two
  proof
    fix x
    assume x-in: x ∈ {0, d}
    show x ∈ ap-segment 0 d ?two
    proof (cases x = 0)
      case x0: True
      have 0 ∈ ap-segment 0 d ?two
      proof (unfold ap-segment-def, rule image-eqI[where x = 0])
        show 0 ∈ {..< ?two}
          by simp
        show 0 = 0 + of-nat 0 * d
          by simp
      qed
      then show ?thesis
        using x0 by simp
    next
      case False
      with x-in have x = d
        by simp
      then show ?thesis
      proof −
        have d ∈ ap-segment 0 d ?two
        proof (unfold ap-segment-def, rule image-eqI[where x = 1])
          show 1 ∈ {..< ?two}
            by simp
          show d = 0 + of-nat 1 * d
            by simp
        qed
        then show ?thesis
```

```
          using ‹x = d› by simp
        qed
      qed
    qed
  qed
  have zero-in-B: 0 ∈ ap-segment 0 d ?two
    using B-eq by simp
  have A-subset-sumset: A ⊆ sumset A (ap-segment 0 d ?two)
  proof
    fix x
    assume x-in: x ∈ A
    have x + 0 ∈ sumset A (ap-segment 0 d ?two)
      using x-in zero-in-B by (rule sumsetI)
    then show x ∈ sumset A (ap-segment 0 d ?two)
      by simp
  qed
  have smallA: card A < ?p
    using A-subset-sumset sumset-small finite-sumset[OF finA finite-ap-segment[of 0 d ?two]]
    by (meson card-mono le-less-trans)
  have sumset-eq: sumset A (ap-segment 0 d ?two) = A ∪ translate d A
    using B-eq by (simp add: sumset-pair-zero)
  have fin-translate: finite (translate d A)
    by simp
  have card-overlap: card (A ∩ translate d A) = card A − 1
  proof −
    have card (sumset A (ap-segment 0 d ?two)) = card (A ∪ translate d A)
      using sumset-eq by simp
    also have ... = card A + card (translate d A) − card (A ∩ translate d A)
    proof −
      have card A + card (translate d A) =
          card (A ∪ translate d A) + card (A ∩ translate d A)
        using finA fin-translate by (rule card-Un-Int)
      then show ?thesis
        by linarith
    qed
    also have ... = card A + card A − card (A ∩ translate d A)
      using finA by (simp add: card-translate-eq)
    finally show ?thesis
      using eq by simp
  qed
  show ?thesis
  proof (rule ap-segment-from-maximal-shift-overlap[OF prime-card d-nonzero smallA card-overlap])
    fix a
    assume A = ap-segment a d (card A)
    then show ∃ a. A = ap-segment a d (card A)
      by blast
  qed
qed

lemma vosper-left-with-right-ap-zero-aux:
  fixes A :: 'a::finite-field set
  assumes prime-card: prime (card (UNIV :: 'a set))
  assumes A-ge2: 2 ≤ card A
  assumes d-nonzero: d ≠ 0
  assumes comp-ge2: 2 ≤ card (UNIV − sumset A (ap-segment 0 d (Suc (Suc m))))
  assumes eq: card (sumset A (ap-segment 0 d (Suc (Suc m)))) = card A + Suc m
  shows ∃ a. A = ap-segment a d (card A)
proof −
  let ?P = λk X.
```

```
  2 ≤ card X ⟶
  2 ≤ card (UNIV − sumset X (ap-segment 0 d (Suc (Suc k)))) ⟶
  card (sumset X (ap-segment 0 d (Suc (Suc k)))) = card X + Suc k ⟶
  (∃ a. X = ap-segment a d (card X))
have aux: ?P k X for k X
proof (induction k arbitrary: X)
  case 0
  show ?case
  proof (intro impI)
    assume X-ge2: 2 ≤ card X
    assume X-comp-raw: 2 ≤ card (UNIV − sumset X (ap-segment 0 d (Suc (Suc 0))))
    assume X-eq-raw: card (sumset X (ap-segment 0 d (Suc (Suc 0)))) = card X + Suc 0
    show ∃ a. X = ap-segment a d (card X)
      by (rule vosper-left-with-right-ap-zero-two[OF prime-card d-nonzero X-ge2 X-comp-raw X-eq-raw])
  qed
next
  case (Suc k)
  show ?case
  proof (intro impI)
    let ?p = card (UNIV :: 'a set)
    let ?n = Suc (Suc (Suc k))
    let ?D = ap-segment 0 d (Suc (Suc k))
    assume X-ge2: 2 ≤ card X
    assume X-comp: 2 ≤ card (UNIV − sumset X (ap-segment 0 d ?n))
    assume X-eq: card (sumset X (ap-segment 0 d ?n)) = card X + Suc (Suc k)
    define C where C = sumset X ?D
    have finX: finite X
      by simp
    have X-nonempty: X ≠ {}
    proof
      assume X = {}
      then have card X = 0
        by simp
      then have 2 ≤ (0::nat)
        using X-ge2 by simp
      then show False
        by simp
    qed
    have sumset-not-full: sumset X (ap-segment 0 d ?n) ≠ UNIV
      using X-comp by auto
    have sumset-small: card (sumset X (ap-segment 0 d ?n)) < card (UNIV :: 'a set)
      using sumset-not-full card-lt-univ-if-not-univ[of sumset X (ap-segment 0 d ?n)] by simp
    have D-decomp: ap-segment 0 d ?n = sumset ?D {0, d}
      by (simp add: sumset-ap-segment-pair-suc)
    have final-as-C: sumset X (ap-segment 0 d ?n) = sumset C {0, d}
      unfolding C-def D-decomp by (simp add: sumset-assoc)
    have D-nonempty: ?D ≠ {}
    proof
      assume ?D = {}
      have 0 ∈ ?D
      proof (unfold ap-segment-def, rule image-eqI[where x = 0])
        show 0 ∈ {..<Suc (Suc k)}
          by simp
        show 0 = 0 + of-nat 0 * d
          by simp
      qed
      then show False
        using ‹?D = {}› by simp
    qed
```

have *C-nonempty*: $C \neq \{\}$
unfolding *C-def* **using** *X-nonempty D-nonempty* **by** *auto*
have *cardD*: *card ?D* = *Suc* (*Suc k*)
proof −
have *Suc* (*Suc k*) ≤ *?p*
using *X-eq sumset-small X-ge2* **by** *linarith*
then show *?thesis*
by (*rule card-ap-segment*[*OF prime-card d-nonzero*])
qed
have *C-lb*: *card C* ≥ *card X* + *Suc k*
proof −
have *cdC*: *card C* ≥ *min ?p* (*card X* + *card ?D* − *1*)
unfolding *C-def*
using *cauchy-davenport-prime-field*[*OF prime-card X-nonempty D-nonempty*]
by *simp*
have *card X* + *card ?D* − *1* < *?p*
using *X-eq sumset-small cardD* **by** *simp*
then show *?thesis*
using *cdC cardD* **by** *simp*
qed
have *final-eq′*: *card* (*sumset C* {*0*, *d*}) = *card X* + *Suc* (*Suc k*)
using *X-eq final-as-C* **by** *simp*
have *C-upper*: *card C* ≤ *card X* + *Suc k*
proof −
have *cd-pair0*: *card* (*sumset C* {*0*, *d*}) ≥ *min ?p* (*card C* + *card* {*0*, *d*} − *1*)
by (*rule cauchy-davenport-prime-field*[*OF prime-card C-nonempty*]) *simp*
have *cd-pair*: *card* (*sumset C* {*0*, *d*}) ≥ *min ?p* (*card C* + *1*)
using *cd-pair0 d-nonzero* **by** *simp*
have *C-small*: *card C* + *1* < *?p*
proof (*rule ccontr*)
assume ¬ *card C* + *1* < *?p*
then have *min ?p* (*card C* + *1*) = *?p*
by *simp*
with *cd-pair* **have** *card* (*sumset C* {*0*, *d*}) ≥ *?p*
by *simp*
moreover have *card* (*sumset C* {*0*, *d*}) < *?p*
using *final-as-C sumset-small* **by** *simp*
ultimately show *False*
by *linarith*
qed
from *cd-pair C-small* **have** *card* (*sumset C* {*0*, *d*}) ≥ *card C* + *1*
by *simp*
then show *?thesis*
using *final-eq′* **by** *linarith*
qed
have *eqC*: *card C* = *card X* + *Suc k*
using *C-lb C-upper* **by** *linarith*
have *D-subset*: *?D* ⊆ *ap-segment 0 d ?n*
by (*auto simp*: *ap-segment-def*)
have *C-subset*: *C* ⊆ *sumset X* (*ap-segment 0 d ?n*)
unfolding *C-def* **by** (*rule sumset-mono-right*[*OF D-subset*])
have *compC-ge2*: *2* ≤ *card* (*UNIV* − *C*)
proof −
have *UNIV* − *sumset X* (*ap-segment 0 d ?n*) ⊆ *UNIV* − *C*
using *C-subset* **by** *auto*
then have *card* (*UNIV* − *sumset X* (*ap-segment 0 d ?n*)) ≤ *card* (*UNIV* − *C*)
by (*intro card-mono*) *auto*
with *X-comp* **show** *?thesis*
by *linarith*

```
    qed
    show ∃ a. X = ap-segment a d (card X)
      using Suc.IH[of X] X-ge2 compC-ge2 eqC
      unfolding C-def by simp
  qed
 qed
 show ?thesis
  using aux[of A m] A-ge2 comp-ge2 eq by simp
qed

theorem vosper-left-with-right-ap-zero:
 fixes A :: 'a::finite-field set
 assumes prime-card: prime (card (UNIV :: 'a set))
 assumes A-ge2: 2 ≤ card A
 assumes d-nonzero: d ≠ 0
 assumes n-ge2: 2 ≤ n
 assumes comp-ge2: 2 ≤ card (UNIV − sumset A (ap-segment 0 d n))
 assumes eq: card (sumset A (ap-segment 0 d n)) = card A + n − 1
 shows ∃ a. A = ap-segment a d (card A)
proof −
 obtain n' where n'-eq: n = Suc n'
  using n-ge2 by (cases n) auto
 then obtain m where m-eq: n' = Suc m
  using n-ge2 by (cases n') auto
 have n-eq: n = Suc (Suc m)
  using n'-eq m-eq by simp
 show ?thesis
  using vosper-left-with-right-ap-zero-aux[OF prime-card A-ge2 d-nonzero, of m] comp-ge2 eq n-eq
  by simp
qed
```

8.3 The full Vosper theorem

After normalizing one side by translation and propagating the common difference through an explicit arithmetic progression on the other side, we recover the classical prime-field conclusion: both extremal sets are arithmetic progressions with the same nonzero step.

```
theorem vosper-prime-field:
 fixes A B :: 'a::finite-field set
 assumes prime-card: prime (card (UNIV :: 'a set))
 assumes A-ge2: 2 ≤ card A
 assumes B-ge2: 2 ≤ card B
 assumes comp-ge2: 2 ≤ card (UNIV − sumset A B)
 assumes eq: card (sumset A B) = card A + card B − 1
 shows ∃ a b d. d ≠ 0 ∧ A = ap-segment a d (card A) ∧ B = ap-segment b d (card B)
proof −
 have A-nonempty: A ≠ {}
  using A-ge2 by auto
 then obtain a0 where a0-in: a0 ∈ A
  by blast
 let ?A0 = translate (− a0) A
 have zero-in-A0: 0 ∈ ?A0
  using a0-in by (auto simp: translate-def)
 have A0-ge2: 2 ≤ card ?A0
  using A-ge2 by (simp add: card-translate-eq)
 have sumset-A0: sumset ?A0 B = translate (− a0) (sumset A B)
  by (simp add: sumset-translate-left)
 have comp-A0: 2 ≤ card (UNIV − sumset ?A0 B)
  using comp-ge2 sumset-A0 by (simp add: card-complement-translate-eq)
```

have *eq-A0*: *card* (*sumset ?A0 B*) = *card ?A0* + *card B* − *1*
proof −
have *card* (*sumset ?A0 B*) = *card* (*sumset A B*)
using *sumset-A0* **by** (*simp add*: *card-translate-eq*)
also have ... = *card A* + *card B* − *1*
using *eq* **by** *simp*
also have ... = *card ?A0* + *card B* − *1*
by (*simp add*: *card-translate-eq*)
finally show *?thesis* **.**
qed
have *apB*: *arithmetic-progression B*
using *vosper-progression-right-zero*[*OF prime-card zero-in-A0 A0-ge2 comp-A0 eq-A0*] **.**

have *sumset-A0-small*: *card* (*sumset ?A0 B*) < *card* (*UNIV* :: *'a set*)
proof (*rule ccontr*)
let *?p* = *card* (*UNIV* :: *'a set*)
assume ¬ *card* (*sumset ?A0 B*) < *?p*
moreover have *card* (*sumset ?A0 B*) ≤ *?p*
using *finite-UNIV finite-sumset*[*of ?A0 B*] **by** (*intro card-mono*) *auto*
ultimately have *card* (*sumset ?A0 B*) = *?p*
by *linarith*
then have *sumset ?A0 B* = *UNIV*
using *finite-UNIV* **by** (*intro card-subset-eq*) *auto*
with *comp-A0* **show** *False*
by *simp*
qed
have *B-subset-sumset*: *B* ⊆ *sumset ?A0 B*
proof
fix *x*
assume *x-in*: *x* ∈ *B*
have *0* + *x* ∈ *sumset ?A0 B*
using *zero-in-A0 x-in* **by** (*rule sumsetI*)
then show *x* ∈ *sumset ?A0 B*
by *simp*
qed
have *B-small*: *card B* < *card* (*UNIV* :: *'a set*)
proof −
have *card B* ≤ *card* (*sumset ?A0 B*)
using *B-subset-sumset finite-sumset*[*of ?A0 B*] **by** (*intro card-mono*) *auto*
with *sumset-A0-small* **show** *?thesis*
by *linarith*
qed
obtain *b d* **where** *d-nonzero*: *d* ≠ *0* **and** *B-eq*: *B* = *ap-segment b d* (*card B*)
by (*rule arithmetic-progression-obtain-card*[*OF prime-card apB B-ge2 B-small*])

let *?B0* = *translate* (− *b*) *B*
have *B0-eq*: *?B0* = *ap-segment 0 d* (*card B*)
proof −
have *?B0* = *translate* (− *b*) (*ap-segment b d* (*card B*))
using *B-eq* **by** *simp*
also have ... = *ap-segment* ((− *b*) + *b*) *d* (*card B*)
by (*simp add*: *ap-segment-translate*)
finally show *?thesis*
by *simp*
qed
have *sumset-B0*: *sumset A ?B0* = *translate* (− *b*) (*sumset A B*)
by (*simp add*: *sumset-translate-right*)
have *comp-B0*: *2* ≤ *card* (*UNIV* − *sumset A ?B0*)
using *comp-ge2 sumset-B0* **by** (*simp add*: *card-complement-translate-eq*)

```
  have eq-B0: card (sumset A ?B0) = card A + card B − 1
  proof −
    have card (sumset A ?B0) = card (sumset A B)
      using sumset-B0 by (simp add: card-translate-eq)
    also have ... = card A + card B − 1
      using eq by simp
    finally show ?thesis .
  qed
  have comp-B0': 2 ≤ card (UNIV − sumset A (ap-segment 0 d (card B)))
    using comp-B0 B0-eq by simp
  have eq-B0': card (sumset A (ap-segment 0 d (card B))) = card A + card B − 1
    using eq-B0 B0-eq by simp
  then obtain a where A-eq: A = ap-segment a d (card A)
    using vosper-left-with-right-ap-zero[OF prime-card A-ge2 d-nonzero B-ge2 comp-B0'] by blast

  show ?thesis
    using d-nonzero A-eq B-eq by blast
qed

end
theory Cauchy-Davenport-Entry
  imports
    Modular-Sumsets
    Cauchy-Davenport-Prime-Field
    Vosper-Prime-Field
begin
```

9 Overview

This section records the scope of the combined Cauchy-Davenport and Vosper development. The session combines abstract prime-field sumset theorems with concrete representative lemmas for modular sumsets. The resulting theory stack connects generic finite-sumset infrastructure, the polynomial-method Cauchy-Davenport bound, and the final Vosper classification theorem.

end

References

[1] N. Alon, M. B. Nathanson, and I. Z. Ruzsa. The polynomial method and restricted sums of congruence classes. *Journal of Number Theory*, 56(2):404–417, 1996. DOI: https://doi.org/10.1006/jnth.1996.0029.

[2] M. Bakšys. A generalization of the cauchy–davenport theorem. *Archive of Formal Proofs*, Nov. 2024. https://isa-afp.org/entries/Generalized_Cauchy_Davenport.html, Formal proof development.

[3] M. Bakšys and A. Koutsoukou-Argyraki. Kneser's theorem and the cauchy–davenport theorem. *Archive of Formal Proofs*, Nov. 2022. https://isa-afp.org/entries/Kneser_Cauchy_Davenport.html, Formal proof development.

[4] M. B. Nathanson. *Additive Number Theory: Inverse Problems and the Geometry of Sumsets*, volume 165 of *Graduate Texts in Mathematics*. Springer-Verlag, 1996.

[5] T. Tao and V. H. Vu. *Additive Combinatorics*, volume 105 of *Cambridge Studies in Advanced Mathematics*. Cambridge University Press, 2006. DOI: https://doi.org/10.1017/CBO9780511755149.

[6] A. G. Vosper. The critical pairs of subsets of a group of prime order. *Journal of the London Mathematical Society*, 31(2):200–205, 1956. DOI: https://doi.org/10.1112/jlms/s1-31.2.200.