Congruences of Bernoulli Numbers

Manuel Eberl

March 29, 2024

Abstract

This entry provides proofs for two important congruences involving Bernoulli numbers. The proofs follow Cohen's textbook *Number Theory Volume II: Analytic and Modern Tools* [1]. In the following we write $\mathcal{B}_k = N_k/D_k$ for the k-th Bernoulli number (with $\gcd(N_k, D_k) = 1$).

The first result that I showed is *Voronoi's congruence*, which states that for any even integer $k \geq 2$ and all positive coprime integers a, n we have:

$$(a^k - 1)N_k \equiv k a^{k-1} D_k \sum_{m=1}^{n-1} m^{k-1} \left\lfloor \frac{ma}{n} \right\rfloor \pmod{n}$$

Building upon this, I then derive *Kummer's congruence*. In its common form, it states that for a prime p and even integers k, k' with $\min(k, k') \geq e + 1$ and $(p-1) \nmid k$ and $k \equiv k' \pmod{\varphi(p^e)}$, we have:

$$\frac{\mathcal{B}_k}{k} \equiv \frac{\mathcal{B}_{k'}}{k'} \pmod{p^e}$$

The version proved in my entry is slightly more general than this.

One application of these congruences is to prove that there are infinitely many irregular primes, which I formalised as well.

Contents

1 Preliminary facts **3**
- 1.1 Miscellaneous facts . . . 3
- 1.2 Facts about congruence . . . 7
- 1.3 Modular inverses . . . 10
- 1.4 Facts about Bernoulli numbers . . . 12

2 Congruence of rational numbers modulo an integer **14**
- 2.1 p-adic valuation of a rational . . . 14
- 2.2 Rational modulo operation . . . 18
- 2.3 Congruence relation . . . 20

3 The Voronoi congruence **29**

4 Kummer's Congruence **50**

5 Regular primes **59**

6 Infinitude of irregular primes **60**

1 Preliminary facts

theory *Kummer_Library*
imports
  "HOL-Number_Theory.Number_Theory"
  "Bernoulli.Bernoulli_Zeta"
begin

1.1 Miscellaneous facts

lemma *fact_ge_monomial:*
  fixes *k :: "'a :: {linordered_semidom, semiring_char_0}"*
  assumes *"n ≥ n0" "fact n0 ≥ c * k ^ n0" "of_nat n0 ≥ k" "k ≥ 0"*
  shows *"fact n ≥ c * k ^ n"*
proof -
  have *"fact n = (∏ i=1..n. of_nat i :: 'a)"*
    by *(simp add: fact_prod)*
  also have *"{1..n} = {1..n0} ∪ {n0<..n}"*
    using *assms* **by** *auto*
  also have *"(∏ i∈.... of_nat i :: 'a) = (∏ i=1..n0. of_nat i) * (∏ i∈{n0<..n}. of_nat i)"*
    by *(subst prod.union_disjoint) auto*
  also have *"(∏ i=1..n0. of_nat i :: 'a) = fact n0"*
    by *(simp add: fact_prod)*
  also have *"fact n0 * (∏ i∈{n0<..n}. of_nat i :: 'a) ≥ c * k ^ n0 * (∏ i∈{n0<..n}. k)"*
    using *assms* **by** *(intro mult_mono prod_mono conjI order.trans[OF assms(3)]) auto*
  also have *"(∏ i∈{n0<..n}. k) = k ^ (n - n0)"*
    using ‹*n ≥ n0*› **by** *simp*
  also have *"c * k ^ n0 * k ^ (n - n0) = c * (k ^ n0 * k ^ (n - n0))"*
    by *(simp add: algebra_simps)*
  also have *"k ^ n0 * k ^ (n - n0) = k ^ n"*
    using ‹*n ≥ n0*› **by** *(simp flip: power_add)*
  finally show *?thesis* **.**
qed

lemma *fact_ge_2pi_power:*
  assumes *"n ≥ 23"*
  shows *"fact n ≥ (2 * pi) ^ n * n"*
proof -
  define *m* **where** *"m = n - 1"*
  have *n_eq: "n = Suc m"*
    using *assms* **by** *(simp add: m_def)*
  have *"m ≥ 22"*
    using *assms* **by** *(simp add: n_eq)*
  hence **: "fact m ≥ 8 * (8 ^ m :: real)"*
    by *(rule fact_ge_monomial) (simp_all add: fact_numeral)*

  have *"(2 * pi) ^ n ≤ (2 * 4) ^ n"*

```
    by (intro power_mono mult_left_mono less_imp_le[OF pi_less_4]) auto
  also have "... = 8 * 8 ^ m"
    by (simp add: n_eq)
  also have "... ≤ fact m"
    using ‹m ≥ 22› by (rule fact_ge_monomial) (simp_all add: fact_numeral)
  also have "fact m = fact n / n"
    by (simp add: n_eq)
  finally show ?thesis
    using assms by (simp add: field_simps)
qed

lemma Rats_power_int: "x ∈ ℚ ⟹ x powi n ∈ ℚ"
  by (auto simp: power_int_def)

lemma coprimeI_via_bezout:
  fixes x y :: "'a :: algebraic_semidom"
  assumes "a * x + b * y = 1"
  shows   "coprime x y"
  by (metis assms coprime_def dvd_add dvd_mult)

lemma quotient_of_eqI:
  assumes "coprime a b" "b > 0" "x = of_int a / of_int b"
  shows   "quotient_of x = (a, b)"
  using Fract_of_int_quotient assms(1) assms(2) assms(3) normalize_stable
quotient_of_Fract
  by simp

lemma quotient_of_of_nat [simp]: "quotient_of (of_nat n) = (int n, 1)"
  by (intro quotient_of_eqI) auto

lemma quotient_of_of_int [simp]: "quotient_of (of_int n) = (n, 1)"
  by (intro quotient_of_eqI) auto

lemma quotient_of_fraction_conv_normalize:
  "quotient_of (of_int a / of_int b) = Rat.normalize (a, b)"
  using Fract_of_int_quotient quotient_of_Fract by presburger

lemma dvd_imp_div_dvd: "(b :: 'a :: algebraic_semidom) dvd a ⟹ a div
b dvd a"
  by (metis dvd_mult_div_cancel dvd_triv_right)

lemma dvd_rat_normalize:
  assumes "b ≠ 0"
  shows   "fst (Rat.normalize (a, b)) dvd a" "snd (Rat.normalize (a, b))
dvd b"
  using assms by (auto simp: Rat.normalize_def Let_def intro!: dvd_imp_div_dvd)

lemma of_int_div: "b dvd a ⟹ of_int (a div b) = of_int a / (of_int
b :: 'a :: field_char_0)"
```

```
  by (elim dvdE) auto

lemma coprime_lcm_left:
  fixes a b c :: "'a :: semiring_gcd"
  shows "coprime a c ⟹ coprime b c ⟹ coprime (lcm a b) c"
  by (meson coprime_divisors coprime_mult_left_iff dvd_refl dvd_triv_left
dvd_triv_right lcm_least)

lemma coprime_Lcm_left:
  fixes x y :: "'a :: semiring_Gcd"
  assumes "finite A" "⋀x. x ∈ A ⟹ coprime x y"
  shows   "coprime (Lcm A) y"
  using assms by (induction rule: finite_induct) (auto intro: coprime_lcm_left)

lemma coprimeI_by_prime_factors:
  fixes x y :: "'a :: factorial_semiring"
  assumes "⋀p. p ∈ prime_factors x ⟹ ¬p dvd y"
  assumes "x ≠ 0"
  shows   "coprime x y"
  using assms by (smt (verit, best) coprimeI dvd_0_right dvd_trans prime_divisor_exists
prime_factorsI)

lemma multiplicity_int: "multiplicity (int p) (int n) = multiplicity
p n"
proof -
  have "{k. p ^ k dvd n} = {k. int p ^ k dvd int n}"
    by (intro Collect_cong) (auto simp flip: of_nat_power)
  thus ?thesis
    by (simp add: multiplicity_def)
qed

lemma squarefree_int_iff [simp]: "squarefree (int n) ⟷ squarefree
n"
proof (cases "n = 0")
  case True
  thus ?thesis by auto
next
  case False
  show ?thesis
  proof
    assume "squarefree n"
    thus "squarefree (int n)"
      apply (subst squarefree_factorial_semiring, use False in ‹simp;
fail›)
      apply (subst (asm) squarefree_factorial_semiring, use False in ‹simp;
fail›)
      by (metis nat_dvd_iff nat_power_eq prime_int_nat_transfer zero_le_power_eq)
  next
    assume "squarefree (int n)"
```

```
    thus "squarefree n"
      apply (subst squarefree_factorial_semiring, use False in ‹simp;
fail›)
      apply (subst (asm) squarefree_factorial_semiring, use False in ‹simp;
fail›)
      by (metis of_nat_dvd_iff of_nat_power prime_nat_int_transfer)
  qed
qed

lemma squarefree_imp_multiplicity_prime_le_1:
  "squarefree n ⟹ n ≠ 0 ⟹ prime p ⟹ multiplicity p n ≤ 1"
  using squarefree_factorial_semiring''[of n] by auto

lemma residue_primroot_is_generator':
  assumes "m > 1" and "residue_primroot m g"
  shows   "bij_betw (λi. g ^ i mod m) {1..totient m} (totatives m)"
  unfolding bij_betw_def
proof
  show "inj_on (λi. g ^ i mod m) {1..totient m}"
  proof (rule inj_onI)
    fix i j assume ij: "i ∈ {1..totient m}" "j ∈ {1..totient m}" "g ^
i mod m = g ^ j mod m"
    hence "[g ^ i = g ^ j] (mod m)"
      by (simp add: Cong.cong_def)
    also have "?this ⟷ [i = j] (mod totient m)"
      using assms by (subst order_divides_expdiff) (auto simp: residue_primroot_def)
    also have "... ⟷ [int i = int j] (mod int (totient m))"
      by (simp add: cong_int_iff)
    also have "... ⟷ int (totient m) dvd (int i - int j)"
      by (rule cong_iff_dvd_diff)
    finally have dvd: "int (totient m) dvd |int i - int j|"
      by simp
    show "i = j"
    proof (rule ccontr)
      assume "i ≠ j"
      with dvd have "int (totient m) ≤ |int i - int j|"
        by (intro zdvd_imp_le) auto
      moreover have "|int i - int j| < totient m"
        using ij by auto
      ultimately show False
        by simp
    qed
  qed
next
  show "(λi. g ^ i mod m) ` {1..totient m} = totatives m"
  proof safe
    fix x assume "x ∈ totatives m"
    also have "totatives m = (λi. g ^ i mod m) ` {..<totient m}"
      using residue_primroot_is_generator[OF assms] unfolding bij_betw_def
```

```
by blast
    finally obtain i where i: "i < totient m" "g ^ i mod m = x"
      by auto
    have "coprime g m"
      using assms by (auto simp: residue_primroot_def coprime_commute)
    hence "g ^ totient m mod m = g ^ 0 mod m"
      using euler_theorem[of g m] by (auto simp: Cong.cong_def)
    with i have "x = g ^ (if i = 0 then totient m else i) mod m"
                "(if i = 0 then totient m else i) ∈ {1..totient m}"
      by auto
    thus "x ∈ (λi. g ^ i mod m) ` {1..totient m}"
      by blast
  qed (use assms in ‹auto simp: power_in_totatives residue_primroot_def›)
qed
```

1.2 Facts about congruence

```
lemma cong_modulus_mono:
  assumes "[a = b] (mod m)" "m' dvd m"
  shows   "[a = b] (mod m')"
  using assms by (metis mod_mod_cancel unique_euclidean_semiring_class.cong_def)

lemma cong_pow_totient:
  fixes x x' n k k' :: nat
  assumes "[x = x'] (mod n)" "[k = k'] (mod totient n)" "coprime x n"
  shows   "[x ^ k = x' ^ k'] (mod n)"
proof -
  have "[k = k'] (mod ord n x)"
    by (rule cong_modulus_mono[OF assms(2)])
       (use assms in ‹auto intro!: order_divides_totient simp: coprime_commute›)
  hence "[x ^ k = x ^ k'] (mod n)"
    by (subst order_divides_expdiff) (use assms in ‹auto simp: coprime_commute›)
  also have "[x ^ k' = x' ^ k'] (mod n)"
    by (intro cong_pow assms)
  finally show ?thesis .
qed

lemma cong_modulus_power:
  assumes "[a = b] (mod (n ^ k))" "k > 0"
  shows   "[a = b] (mod n)"
  using assms(1) by (rule cong_modulus_mono) (use assms(2) in auto)

lemma cong_mult_cancel:
  assumes "[n * a = n * b] (mod (n * m))" "n ≠ 0"
  shows   "[a = b] (mod m)"
  using assms by (auto simp: Cong.cong_def)

lemma cong_mult_square:
  assumes "[a = 0] (mod n)" "[b = b'] (mod n)"
```

```
  shows   "[a * b = a * b'] (mod (n^2))"
  using assms by (auto simp: Cong.cong_def power2_eq_square intro: mod_mult_cong
elim!: dvdE)

lemma sum_reindex_bij_betw_cong:
  assumes "⋀a. a ∈ S ⟹ i (j a) = a"
  assumes "⋀a. a ∈ S ⟹ j a ∈ T"
  assumes "⋀b. b ∈ T ⟹ j (i b) = b"
  assumes "⋀b. b ∈ T ⟹ i b ∈ S"
  assumes "⋀a. a ∈ S ⟹ [h (j a) = g a] (mod m)"
  shows   "[sum g S = sum h T] (mod m)"
proof -
  have "[sum g S = (∑x∈S. g x mod m)] (mod m)"
    by (intro cong_sum) (auto simp: Cong.cong_def)
  also have "(∑x∈S. g x mod m) = (∑x∈T. h x mod m)"
    using assms by (intro sum.reindex_bij_witness[of _ i j]) (auto simp:
Cong.cong_def)
  also have "[... = sum h T] (mod m)"
    by (intro cong_sum) (auto simp: Cong.cong_def)
  finally show ?thesis .
qed

lemma power_mult_cong:
  fixes a b :: "'a :: unique_euclidean_ring"
  assumes "[a = b] (mod n^k)" and "k' ≤ k + l"
  shows   "[n^l * a = n^l * b] (mod n^k')"
proof (cases "k' ≥ k")
  case True
  have "n^(k'-k) * n^k dvd n^l * (a - b)"
    by (intro mult_dvd_mono le_imp_power_dvd) (use assms in ‹auto simp:
cong_iff_dvd_diff›)
  also have "n^(k'-k) * n^k = n ^ k'"
    using True by (simp flip: power_add)
  finally show ?thesis
    by (simp add: cong_iff_dvd_diff algebra_simps)
next
  case False
  have "n ^ k' dvd n ^ k"
    using False by (intro le_imp_power_dvd) auto
  also have "[n^l * a = n^l * b] (mod n^k)"
    by (intro cong_mult cong_refl assms)
  hence "n ^ k dvd (n^l * a - n^l * b)"
    by (simp add: cong_iff_dvd_diff)
  finally show ?thesis
    by (simp add: cong_iff_dvd_diff)
qed

lemma residue_primroot_power_cong_neg1:
  fixes x :: nat and p :: nat
```

```
  assumes "prime p" "p ≠ 2" "residue_primroot p x"
  shows   "[int x ^ ((p - 1) div 2) = -1] (mod p)"
proof -
  have "x > 0"
    using assms by (intro Nat.gr0I) auto
  from assms have "p > 2"
    using prime_gt_1_nat[of p] by auto
  hence "odd p"
    using assms by (intro prime_odd_nat) auto
  have cong_1_iff: "[int x ^ k = 1] (mod p) ⟷ (p - 1) dvd k" for k
    using assms
    by (metis cong_int_iff of_nat_1 of_nat_power ord_divides residue_primroot.cases
totient_prime)

  have "[int x ^ ((p - 1) div 2) = 1] (mod p) ∨ [int x ^ ((p - 1) div
2) = -1] (mod p)"
  proof (rule cong_square)
    have "int x ^ ((p - 1) div 2) * int x ^ ((p - 1) div 2) = (int x ^
((p - 1) div 2)) ^ 2"
      by (simp add: power2_eq_square)
    also have "... = int x ^ ((p - 1) div 2 * 2)"
      by (simp add: power_mult)
    also have "(p - 1) div 2 * 2 = p - 1"
      using ‹odd p› by auto
    also have "[int x ^ (p - 1) = 1] (mod p)"
      by (subst cong_1_iff) auto
    finally show "[int x ^ ((p - 1) div 2) * int x ^ ((p - 1) div 2) =
1] (mod p)" .
  qed (use assms ‹x > 0› in auto)
  moreover have "¬p - 1 dvd (p - 1) div 2"
  proof
    assume "p - 1 dvd (p - 1) div 2"
    hence "p - 1 ≤ (p - 1) div 2"
      using assms ‹odd p› ‹p > 2› by (intro dvd_imp_le) (auto elim!: dvdE)
    also have "... < (p - 1)"
      by (rule div_less_dividend) (use ‹p > 2› in auto)
    finally show False
      by simp
  qed
  hence "[int x ^ ((p - 1) div 2) ≠ 1] (mod p)"
    by (subst cong_1_iff) auto
  ultimately show ?thesis
    by auto
qed

lemma cong_mod_left: "[a = b] (mod p) ⟹ [a mod p = b] (mod p)"
  by auto

lemma cong_mod_right: "[a = b] (mod p) ⟹ [a = b mod p] (mod p)"
```

```
  by auto

lemma cong_mod: "[a = b] (mod p) ⟹ [a mod p = b mod p] (mod p)"
  by auto
```

1.3 Modular inverses

```
definition modular_inverse where
  "modular_inverse p n = fst (bezout_coefficients n p) mod p"

lemma cong_modular_inverse1:
  assumes "coprime n p"
  shows   "[n * modular_inverse p n = 1] (mod p)"
proof -
  have "[fst (bezout_coefficients n p) * n + snd (bezout_coefficients
n p) * p =
        modular_inverse p n * n + 0] (mod p)"
    unfolding modular_inverse_def by (intro cong_add cong_mult) (auto
simp: Cong.cong_def)
  also have "fst (bezout_coefficients n p) * n + snd (bezout_coefficients
n p) * p = gcd n p"
    by (simp add: bezout_coefficients_fst_snd)
  also have "... = 1"
    using assms by simp
  finally show ?thesis
    by (simp add: cong_sym mult_ac)
qed

lemma cong_modular_inverse2:
  assumes "coprime n p"
  shows   "[modular_inverse p n * n = 1] (mod p)"
  using cong_modular_inverse1[OF assms] by (simp add: mult.commute)

lemma coprime_modular_inverse [simp, intro]:
  fixes n :: "'a :: {euclidean_ring_gcd,unique_euclidean_semiring}"
  assumes "coprime n p"
  shows   "coprime (modular_inverse p n) p"
  using cong_modular_inverse1[OF assms] assms
  by (meson cong_imp_coprime cong_sym coprime_1_left coprime_mult_left_iff)

lemma modular_inverse_int_nonneg: "p > 0 ⟹ modular_inverse p (n ::
int) ≥ 0"
  by (simp add: modular_inverse_def)

lemma modular_inverse_int_less: "p > 0 ⟹ modular_inverse p (n :: int)
< p"
  by (simp add: modular_inverse_def)

lemma modular_inverse_int_eqI:
```

```
  fixes x y :: int
  assumes "y ∈ {0..<m}" "[x * y = 1] (mod m)"
  shows   "modular_inverse m x = y"
proof -
  from assms have "coprime x m"
    using cong_gcd_eq by force
  have "[modular_inverse m x * 1 = modular_inverse m x * (x * y)] (mod
m)"
    by (rule cong_sym, intro cong_mult assms cong_refl)
  also have "modular_inverse m x * (x * y) = (modular_inverse m x * x)
* y"
    by (simp add: mult_ac)
  also have "[... = 1 * y] (mod m)"
    using ‹coprime x m› by (intro cong_mult cong_refl cong_modular_inverse2)
  finally have "[modular_inverse m x = y] (mod m)"
    by simp
  thus "modular_inverse m x = y"
    using assms by (simp add: Cong.cong_def modular_inverse_def)
qed

lemma modular_inverse_1 [simp]:
  assumes "m > (1 :: int)"
  shows   "modular_inverse m 1 = 1"
  by (rule modular_inverse_int_eqI) (use assms in auto)

lemma modular_inverse_int_mult:
  fixes x y :: int
  assumes "coprime x m" "coprime y m" "m > 0"
  shows "modular_inverse m (x * y) = (modular_inverse m y * modular_inverse
m x) mod m"
proof (rule modular_inverse_int_eqI)
  show "modular_inverse m y * modular_inverse m x mod m ∈ {0..<m}"
    using assms by auto
next
  have "[x * y * (modular_inverse m y * modular_inverse m x mod m) =
        x * y * (modular_inverse m y * modular_inverse m x)] (mod m)"
    by (intro cong_mult cong_refl) auto
  also have "x * y * (modular_inverse m y * modular_inverse m x) =
             (x * modular_inverse m x) * (y * modular_inverse m y)"
    by (simp add: mult_ac)
  also have "[... = 1 * 1] (mod m)"
    by (intro cong_mult cong_modular_inverse1 assms)
  finally show "[x * y * (modular_inverse m y * modular_inverse m x mod
m) = 1] (mod m)"
    by simp
qed

lemma bij_betw_int_remainders_mult:
  fixes a n :: int
```

```
  assumes a: "coprime a n"
  shows   "bij_betw (λm. a * m mod n) {1..<n} {1..<n}"
proof -
  define a' where "a' = modular_inverse n a"

  have *: "a' * (a * m mod n) mod n = m ∧ a * m mod n ∈ {1..<n}"
    if a: "[a * a' = 1] (mod n)" and m: "m ∈ {1..<n}" for m a a' :: int
  proof
    have "[a' * (a * m mod n) = a' * (a * m)] (mod n)"
      by (intro cong_mult cong_refl) (auto simp: Cong.cong_def)
    also have "a' * (a * m) = (a * a') * m"
      by (simp add: mult_ac)
    also have "[(a * a') * m = 1 * m] (mod n)"
      unfolding a'_def by (intro cong_mult cong_refl) (use a in auto)
    finally show "a' * (a * m mod n) mod n = m"
      using m by (simp add: Cong.cong_def)
  next
    have "coprime a n"
      using a coprime_iff_invertible_int by auto
    hence "¬n dvd (a * m)"
      using m by (simp add: coprime_commute coprime_dvd_mult_right_iff
zdvd_not_zless)
    hence "a * m mod n > 0"
      using m order_le_less by fastforce
    thus "a * m mod n ∈ {1..<n}"
      using m by auto
  qed

  have "[a * a' = 1] (mod n)" "[a' * a = 1] (mod n)"
    unfolding a'_def by (rule cong_modular_inverse1 cong_modular_inverse2;
fact)+
  from this[THEN *] show ?thesis
    by (intro bij_betwI[of _ _ _ "λm. a' * m mod n"]) auto
qed
```

1.4 Facts about Bernoulli numbers

```
definition bernoulli_rat :: "nat ⇒ rat"
  where "bernoulli_rat n = of_int (bernoulli_num n) / of_int (bernoulli_denom
n)"

bundle bernoulli_notation
begin
notation bernoulli_rat ("ℬ")
end

bundle no_bernoulli_notation
begin
no_notation bernoulli_rat ("ℬ")
```

end

lemma *bernoulli_num_eq_0_iff: "bernoulli_num n = 0 ⟷ odd n ∧ n ≠ 1"*
proof -
  have *"bernoulli_num n = 0 ⟷ real_of_int (bernoulli_num n) / real (bernoulli_denom n) = 0"*
    by *auto*
  also have *"real_of_int (bernoulli_num n) / real (bernoulli_denom n) = bernoulli n"*
    by *(rule bernoulli_conv_num_denom [symmetric])*
  also have *"bernoulli n = 0 ⟷ odd n ∧ n ≠ 1"*
    by *(rule bernoulli_zero_iff)*
  finally show *?thesis* **.**
qed

lemma *bernoulli_num_odd_eq_0: "odd k ⟹ k ≠ 1 ⟹ bernoulli_num k = 0"*
  by *(simp add: bernoulli_num_def bernoulli_odd_eq_0)*

lemma *prime_dvd_bernoulli_denom_iff:*
  assumes *"prime p" "even k" "k > 0"*
  shows *"p dvd bernoulli_denom k ⟷ (p - 1) dvd k"*
proof -
  have *fin: "finite {p. prime p ∧ p - Suc 0 dvd k}"*
    by *(rule finite_subset[of _ "{..k+1}"]) (use assms* **in** *‹auto dest!: dvd_imp_le›)*
  have *"bernoulli_denom k = ∏ {p. prime p ∧ p - 1 dvd k}"*
    unfolding *bernoulli_denom_def* **using** *assms* **by** *auto*
  also have *"p dvd ... ⟷ (p - 1) dvd k"*
    using *assms fin primes_dvd_imp_eq* **by** *(subst prime_dvd_prod_iff) auto*
  finally show *?thesis* **.**
qed

lemma *bernoulli_num_denom_eqI:*
  assumes *"bernoulli k = of_int a / of_nat b" "coprime a b" "b > 0"*
  shows *"bernoulli_num k = a" "bernoulli_denom k = b"*
proof -
  have *"bernoulli k = of_rat (of_int (bernoulli_num k) / of_nat (bernoulli_denom k))"*
    by *(simp add: bernoulli_conv_num_denom of_rat_divide)*
  also have *"bernoulli k = of_rat (of_int a / of_nat b)"*
    by *(simp add: assms(1) of_rat_divide)*
  finally have *∗: "of_int (bernoulli_num k) / of_nat (bernoulli_denom k) = (of_int a / of_nat b :: rat)"*
    by *simp*

  have *"quotient_of (of_int (bernoulli_num k) / of_nat (bernoulli_denom k)) =*

```
        (bernoulli_num k, int (bernoulli_denom k))"
    by (intro quotient_of_eqI coprime_bernoulli_num_denom) (auto simp:
bernoulli_denom_pos)
  also note *
  also have "quotient_of (of_int a / of_nat b) = (a, int b)"
    by (intro quotient_of_eqI) (use assms in auto)
  finally show "bernoulli_num k = a" "bernoulli_denom k = b"
    by simp_all
qed

lemma bernoulli_rat_eq_0_iff: "bernoulli_rat n = 0 ⟷ odd n ∧ n ≠
1"
  by (auto simp: bernoulli_rat_def bernoulli_num_eq_0_iff)

lemma bernoulli_rat_odd_eq_0: "odd n ⟹ n ≠ 1 ⟹ bernoulli_rat n =
0"
  by (auto simp: bernoulli_rat_def bernoulli_num_odd_eq_0)

lemma bernoulli_rat_conv_bernoulli: "of_rat (bernoulli_rat n) = bernoulli
n"
  unfolding bernoulli_rat_def by (simp add: bernoulli_conv_num_denom of_rat_divide)

lemma quotient_of_bernoulli_rat [simp]:
  "quotient_of (bernoulli_rat n) = (bernoulli_num n, int (bernoulli_denom
n))"
  unfolding bernoulli_rat_def
  by (rule quotient_of_eqI) (auto intro: bernoulli_denom_pos coprime_bernoulli_num_denom)

end
```

2 Congruence of rational numbers modulo an integer

```
theory Rat_Congruence
  imports Kummer_Library
begin
```

2.1 p-adic valuation of a rational

The notion of the multiplicity $\nu_p(n)$ of a prime p in an integer n can be generalised to rational numbers via $\nu_p(a/b) = \nu_p(a) - \nu_p(b)$. This is also called the p-adic valuation of a/b.

```
definition qmultiplicity :: "int ⇒ rat ⇒ int" where
  "qmultiplicity p x = (case quotient_of x of (a, b) ⇒ int (multiplicity
p a) - int (multiplicity p b))"

lemma qmultiplicity_of_int [simp]:
```

```
  "qmultiplicity p (of_int n) = int (multiplicity p n)"
proof -
  have "quotient_of (of_int n) = (n, 1)"
    by (intro quotient_of_eqI) auto
  thus ?thesis
    by (simp add: qmultiplicity_def)
qed

lemma qmultiplicity_of_nat [simp]:
  "qmultiplicity p (of_nat n) = int (multiplicity p n)"
  using qmultiplicity_of_int[of p "int n"] by (simp del: qmultiplicity_of_int)

lemma qmultiplicity_numeral [simp]:
  "qmultiplicity p (numeral n) = int (multiplicity p (numeral n))"
  using qmultiplicity_of_nat[of p "numeral n"] by (simp del: qmultiplicity_of_nat)

lemma qmultiplicity_0 [simp]: "qmultiplicity p 0 = 0"
  by (simp add: qmultiplicity_def)

lemma qmultiplicity_1 [simp]: "qmultiplicity p 1 = 0"
  by (simp add: qmultiplicity_def)

lemma qmultiplicity_minus [simp]: "qmultiplicity p (-x) = qmultiplicity
p x"
  by (auto simp: qmultiplicity_def rat_uminus_code case_prod_unfold Let_def)

lemma qmultiplicity_divide_of_int:
  assumes "x ≠ 0" "y ≠ 0" "prime_elem p"
  shows   "qmultiplicity p (of_int x / of_int y) = int (multiplicity p
x) - int (multiplicity p y)"
proof -
  define d where "d = sgn y * gcd x y"
  define x' y' where "x' = x div d" and "y' = y div d"
  have xy_eq: "x = x' * d" "y = y' * d"
    unfolding x'_def y'_def d_def using assms by (auto simp: sgn_if)
  have "sgn y = sgn y' * sgn d"
    using assms by (auto simp: xy_eq sgn_mult)
  also have "sgn d = sgn y"
    using assms by (auto simp: d_def sgn_mult)
  finally have "y' > 0"
    using assms by (auto simp: sgn_if split: if_splits)

  have "gcd x y = gcd x' y' * |d|"
    by (auto simp: xy_eq gcd_mult_right abs_mult gcd.commute)
  also have "|d| = gcd x y"
    using assms by (simp add: d_def abs_mult)
  finally have "gcd x' y' = 1"
    using assms by simp
  hence "coprime x' y'"
```

```
    by blast

  have "d ≠ 0" "x' ≠ 0" "y' ≠ 0"
    using assms by (auto simp: xy_eq)
  hence "(of_int x / of_int y :: rat) = (of_int x' / of_int y')"
    by (auto simp: xy_eq field_simps)
  also have "quotient_of ... = (x', y')"
    using ‹coprime x' y'› ‹y' > 0› by (intro quotient_of_eqI) (auto simp:
)
  hence "qmultiplicity p (of_int x' / of_int y') = int (multiplicity p
x') - int (multiplicity p y')"
    by (simp add: qmultiplicity_def)
  also have "... = int (multiplicity p x' + multiplicity p d) - int (multiplicity
p y' + multiplicity p d)"
    by simp
  also have "... = int (multiplicity p x) - int (multiplicity p y)"
    using assms unfolding xy_eq by (subst (1 2) prime_elem_multiplicity_mult_distrib)
auto
  finally show ?thesis .
qed

lemma qmultiplicity_mult [simp]:
  assumes "prime_elem p" "x ≠ 0" "y ≠ 0"
  shows   "qmultiplicity p (x * y) = qmultiplicity p x + qmultiplicity
p y"
proof -
  obtain a b where ab: "quotient_of x = (a, b)"
    using prod.exhaust by blast
  obtain c d where cd: "quotient_of y = (c, d)"
    using prod.exhaust by blast
  have x: "x = of_int a / of_int b ∧ b > 0"
    using ab by (simp add: quotient_of_denom_pos quotient_of_div)
  have y: "y = of_int c / of_int d ∧ d > 0"
    using cd by (simp add: quotient_of_denom_pos quotient_of_div)
  have [simp]: "a ≠ 0" "b ≠ 0" "c ≠ 0" "d ≠ 0"
    using assms x y by auto
  have "x * y = of_int (a * c) / of_int (b * d)"
    by (simp add: x y)
  also have "qmultiplicity p ... = int (multiplicity p (a * c)) - int (multiplicity
p (b * d))"
    using assms(1) by (subst qmultiplicity_divide_of_int) auto
  also have "... = qmultiplicity p x + qmultiplicity p y"
    using assms(1)
    by (subst (1 2) prime_elem_multiplicity_mult_distrib)
       (auto simp: x y qmultiplicity_divide_of_int)
  finally show ?thesis .
qed

lemma qmultiplicity_inverse [simp]:
```

```
  "qmultiplicity p (inverse x) = -qmultiplicity p x"
proof (cases "x = 0")
  case False
  hence "fst (quotient_of x) ≠ 0"
    by (metis div_0 fst_conv of_int_0 quotient_of_div surj_pair)
  thus ?thesis
    by (auto simp: qmultiplicity_def rat_inverse_code case_prod_unfold
Let_def sgn_if)
qed auto

lemma qmultiplicity_divide [simp]:
  assumes "prime_elem p" "x ≠ 0" "y ≠ 0"
  shows   "qmultiplicity p (x / y) = qmultiplicity p x - qmultiplicity
p y"
proof -
  have "qmultiplicity p (x / y) = qmultiplicity p (x * inverse y)"
    by (simp add: field_simps)
  also have "... = qmultiplicity p x - qmultiplicity p y"
    using assms by (subst qmultiplicity_mult) auto
  finally show ?thesis .
qed

lemma qmultiplicity_nonneg_iff:
  assumes "a ≠ 0" "b ≠ 0" "coprime a b" "prime p"
  shows   "qmultiplicity p (of_int a / of_int b) ≥ 0 ⟷ ¬p dvd b"
proof -
  have "qmultiplicity p (of_int a / of_int b) = int (multiplicity p a)
- int (multiplicity p b)"
    using assms by (subst qmultiplicity_divide_of_int) auto
  also have "... ≥ 0 ⟷ ¬p dvd b"
  proof (cases "p dvd b")
    case True
    hence "¬p dvd a"
      using ‹coprime a b› ‹prime p› by (meson coprime_common_divisor_int
not_prime_unit zdvd1_eq)
    hence "multiplicity p a = 0"
      using not_dvd_imp_multiplicity_0 by blast
    moreover have "multiplicity p b ≥ 1"
      using True assms by (intro multiplicity_geI) auto
    ultimately show ?thesis
      using True by simp
  next
    case False
    hence "multiplicity p b = 0"
      using not_dvd_imp_multiplicity_0 by blast
    thus ?thesis
      using False by simp
  qed
  finally show ?thesis .
```

```
qed

lemma qmultiplicity_nonneg_imp_not_dvd_denom:
  assumes "qmultiplicity p x ≥ 0" "|p| ≠ 1"
  shows   "¬p dvd snd (quotient_of x)"
proof -
  obtain a b where ab: "quotient_of x = (a, b)"
    using prod.exhaust by blast
  have "b > 0"
    using ab quotient_of_denom_pos by blast
  have "¬p dvd b"
  proof
    assume "p dvd b"
    hence "multiplicity p b ≥ 1"
      using assms(2) ‹b > 0› by (intro multiplicity_geI) auto
    moreover have "coprime a b"
      using ab by (simp add: quotient_of_coprime)
    hence "¬p dvd a"
      using ‹p dvd b› assms(2) coprime_common_divisor_int by blast
    hence "multiplicity p a = 0"
      by (intro not_dvd_imp_multiplicity_0)
    ultimately show False
      using assms by (simp add: qmultiplicity_def ab)
  qed
  thus ?thesis
    by (simp add: ab)
qed

lemma qmultiplicity_prime_nonneg_imp_coprime_denom:
  assumes "qmultiplicity p x ≥ 0" "prime p"
  shows   "coprime (snd (quotient_of x)) p"
  using qmultiplicity_nonneg_imp_not_dvd_denom[OF assms(1)] assms(2)
  by (simp add: coprime_commute prime_ge_0_int prime_imp_coprime_int)
```

2.2 Rational modulo operation

Similarly, we can define $(a/b) \bmod m$ whenever b and m are coprime by choosing to interpret $(1/b) \bmod m$ as the modular inverse of b modulo m:

```
definition qmod :: "rat ⇒ int ⇒ int" (infixl "qmod" 70) where
  "x qmod m = (let (a, b) = quotient_of x in if coprime b m then (a *
modular_inverse m b) mod m else 0)"

lemma qmod_mod_absorb [simp]: "x qmod m mod m = x qmod m"
  by (simp add: qmod_def case_prod_unfold Let_def)

lemma qmod_of_nat [simp]: "m > 1 ⟹ of_nat x qmod m = int x mod m"
  by (simp add: qmod_def)

lemma qmod_of_int [simp]: "m > 1 ⟹ of_int x qmod m = x mod m"
```

```
  by (simp add: qmod_def)

lemma qmod_numeral [simp]: "m > 1 ⟹ numeral n qmod m = numeral n mod
m"
  by (simp add: qmod_def)

lemma qmod_0 [simp]: "0 qmod m = 0"
  by (simp add: qmod_def)

lemma qmod_1 [simp]: "m > 1 ⟹ 1 qmod m = 1"
  by (simp add: qmod_def)

lemma qmod_fraction_eq:
  assumes "coprime b m" "b ≠ 0" "m > 0"
  shows   "(of_int a / of_int b) qmod m = a * modular_inverse m b mod
m"
proof -
  define d where "d = sgn b * gcd a b"
  define a' where "a' = a div d"
  define b' where "b' = b div d"
  have "d dvd a" "d dvd b"
    using assms unfolding d_def by auto
  hence a_eq: "a = a' * d" and b_eq: "b = b' * d"
    unfolding a'_def b'_def d_def by auto
  have "d ≠ 0"
    unfolding d_def using assms by (auto simp: sgn_if)
  hence "sgn d = sgn b"
    by (auto simp: d_def sgn_mult)
  moreover have "sgn b' = sgn d * sgn b"
    using ‹d ≠ 0› by (simp add: b_eq sgn_mult)
  ultimately have "sgn b' = 1"
    using assms by simp
  hence "b' > 0"
    by (auto simp: sgn_if split: if_splits)
  have "gcd a b = gcd a' b' * |d|"
    by (simp add: a_eq b_eq gcd_mult_right abs_mult gcd.commute)
  also have "|d| = gcd a b"
    using assms by (simp add: d_def abs_mult)
  finally have "gcd a' b' = 1"
    using assms(2) by simp
  hence "coprime a' b'"
    by auto
  have "coprime b' m" "coprime d m"
    using assms(1) b_eq by simp_all
  have ab': "quotient_of (of_int a / of_int b) = (a', b')"
    using ‹b' > 0› ‹coprime a' b'› ‹d ≠ 0› by (intro quotient_of_eqI)
(auto simp: a_eq b_eq)

  have "(of_int a / of_int b) qmod m = a' * modular_inverse m b' mod m"
```

```
    using ‹coprime b' m› by (simp add: qmod_def ab')
  also have "[... = a' * 1 * modular_inverse m b'] (mod m)"
    by simp
  also have "[a' * 1 * modular_inverse m b' = a' * (d * modular_inverse
m d) * modular_inverse m b'] (mod m)"
    by (intro cong_mult cong_refl cong_sym[OF cong_modular_inverse1] ‹coprime
d m›)
  also have "a' * (d * modular_inverse m d) * modular_inverse m b' =
             (a' * d) * (modular_inverse m d * modular_inverse m b')"
    by (simp add: mult_ac)
  also have "[(a' * d) * (modular_inverse m d * modular_inverse m b')
=
              (a' * d) * (modular_inverse m d * modular_inverse m b' mod
m)] (mod m)"
    by (intro cong_mult cong_refl) auto
  also have "modular_inverse m d * modular_inverse m b' mod m = modular_inverse
m (b' * d)"
    by (rule modular_inverse_int_mult [symmetric]) (use ‹coprime b' m›
‹coprime d m› ‹m > 0› in auto)
  also have "[(a' * d) * modular_inverse m (b' * d) = a * modular_inverse
m b mod m] (mod m)"
    by (simp add: a_eq b_eq)
  finally show ?thesis
    by (simp add: Cong.cong_def)
qed
```

2.3 Congruence relation

With this, it is now straightforward to define the congruence relation $x = y \pmod{m}$ for rational x, y:

```
definition qcong :: "rat ⇒ rat ⇒ int ⇒ bool" (‹(1[_ = _] '(' qmod _'))›)
where
  "[a = b] (qmod m) ⟷
     coprime (snd (quotient_of a)) m ∧ coprime (snd (quotient_of b)) m
∧ a qmod m = b qmod m"

lemma qcong_of_int_iff [simp]:
  assumes "m > 1"
  shows   "[of_int a = of_int b] (qmod m) ⟷ [a = b] (mod m)"
  using assms by (auto simp: qcong_def Cong.cong_def)

lemma cong_imp_qcong:
  assumes "[a = b] (mod m)" "m > 1"
  shows   "[of_int a = of_int b] (qmod m)"
  using assms by (auto simp: qcong_def Cong.cong_def)

lemma cong_imp_qcong_of_nat:
  assumes "[a = b] (mod m)" "m > 1"
  shows   "[of_nat a = of_nat b] (qmod m)"
```

```
  using cong_imp_qcong assms
  by (metis cong_int_iff of_int_of_nat_eq of_nat_1 of_nat_less_iff)

lemma qcong_refl [intro]: "coprime (snd (quotient_of q)) m ⟹ [q = q]
(qmod m)"
  by (auto simp: qcong_def)

lemma qcong_sym_eq: "[q1 = q2] (qmod m) ⟷ [q2 = q1] (qmod m)"
  by (simp add: qcong_def conj_ac eq_commute)

lemma qcong_sym: "[q1 = q2] (qmod m) ⟹ [q2 = q1] (qmod m)"
  using qcong_sym_eq by blast

lemma qcong_trans [trans]:
  assumes "[q1 = q2] (qmod m)" "[q2 = q3] (qmod m)"
  shows   "[q1 = q3] (qmod m)"
  using assms by (auto simp: qcong_def)

lemma qcong_0D:
  assumes "[x = 0] (qmod m)"
  shows   "m dvd fst (quotient_of x)"
proof -
  have 1: "coprime (snd (quotient_of x)) m"
   and 2: "m dvd fst (quotient_of x) * modular_inverse m (snd (quotient_of
x))"
    using assms by (auto simp: qcong_def qmod_def case_prod_unfold Let_def)
  have 3: "coprime (modular_inverse m (snd (quotient_of x))) m"
    using 1 by blast
  from 1 2 3 show ?thesis
    using coprime_commute coprime_dvd_mult_left_iff by blast
qed

lemma qcong_0_iff:
  "[x = 0] (qmod m) ⟷ m dvd fst (quotient_of x) ∧ coprime (snd (quotient_of
x)) m"
proof
  assume "m dvd fst (quotient_of x) ∧ coprime (snd (quotient_of x)) m"
  thus "[x = 0] (qmod m)"
    by (auto simp: qcong_def qmod_def case_prod_unfold)
qed (use qcong_0D[of x m] in ‹auto simp: qcong_def›)

lemma qcong_1 [simp]: "[a = b] (qmod 1)"
  by (simp_all add: qcong_def qmod_def)

lemma mod_minus_cong':
  fixes a b :: "'a :: euclidean_ring_cancel"
  assumes "(- a) mod b = (- a') mod b"
  shows "a mod b = a' mod b"
  using mod_minus_cong[OF assms] by simp
```

```
lemma qcong_minus_minus_iff:
  "[-b = -c] (qmod a) ⟷ [b = c] (qmod a)"
  by (auto simp: qcong_def rat_uminus_code case_prod_unfold Let_def qmod_def
          dest: mod_minus_cong' intro: mod_minus_cong)

lemma qcong_minus: "[b = c] (qmod a) ⟹ [-b = -c] (qmod a)"
  by (simp only: qcong_minus_minus_iff)

lemma qcong_fraction_iff:
  assumes "b ≠ 0" "d ≠ 0" "coprime b m" "coprime d m" "m > 0"
  shows   "[of_int a / of_int b = of_int c / of_int d] (qmod m) ⟷ [a
* d = b * c] (mod m)"
proof
  assume *: "[of_int a / of_int b = of_int c / of_int d] (qmod m)"
  have "[a * 1 * d = a * (modular_inverse m b * b) * d] (mod m)"
    by (rule cong_sym, intro cong_mult cong_modular_inverse2 cong_refl
assms)
  also from * have "rat_of_int a / rat_of_int b qmod m = rat_of_int c
/ rat_of_int d qmod m"
    by (auto simp: qcong_def)
  hence "[a * modular_inverse m b = c * modular_inverse m d] (mod m)"
    using assms by (auto simp: qmod_fraction_eq Cong.cong_def)
  hence "[a * modular_inverse m b * (b * d) = c * modular_inverse m d
* (b * d)] (mod m)"
    by (rule cong_mult) (rule cong_refl)
  hence "[a * (modular_inverse m b * b) * d = c * (modular_inverse m d
* d) * b] (mod m)"
    by (simp add: mult_ac)
  also have "[c * (modular_inverse m d * d) * b = c * 1 * b] (mod m)"
    by (intro cong_mult cong_modular_inverse2 cong_refl assms)
  finally show "[a * d = b * c] (mod m)"
    by (simp add: mult_ac)
next
  assume *: "[a * d = b * c] (mod m)"
  have "rat_of_int a / rat_of_int b qmod m = a * modular_inverse m b mod
m"
    using assms by (subst qmod_fraction_eq) auto
  have "rat_of_int c / rat_of_int d qmod m = c * modular_inverse m d mod
m"
    using assms by (subst qmod_fraction_eq) auto
  let ?b' = "modular_inverse m b" and ?d' = "modular_inverse m d"
  have "[a * ?b' mod m = a * 1 * ?b'] (mod m)"
    by auto
  also have "[a * 1 * ?b' = a * (d * ?d') * ?b'] (mod m)"
    by (rule cong_sym, intro cong_mult cong_modular_inverse1 cong_refl
assms)
  also have "[a * d * (?b' * ?d') = b * c * (?b' * ?d')] (mod m)"
    using * by (rule cong_mult) (rule cong_refl)
```

```
  hence "[a * (d * ?d') * ?b' = b * ?b' * c * ?d'] (mod m)"
    by (simp add: mult_ac)
  also have "[b * ?b' * c * ?d' = 1 * c * ?d'] (mod m)"
    by (intro cong_mult cong_modular_inverse1 cong_refl assms)
  also have "[1 * c * ?d' = c * ?d' mod m] (mod m)"
    by auto
  finally have "rat_of_int a / rat_of_int b qmod m = rat_of_int c / rat_of_int
d qmod m"
    using assms by (simp add: qmod_fraction_eq Cong.cong_def)
  moreover have "coprime (snd (Rat.normalize (a, b))) m" "coprime (snd
(Rat.normalize (c, d))) m"
    using dvd_rat_normalize assms by (meson coprime_divisors dvd_refl)+
  ultimately show "[of_int a / of_int b = of_int c / of_int d] (qmod m)"
    unfolding qcong_def by (auto simp: quotient_of_fraction_conv_normalize)
qed

lemma qcong_fractionI:
  assumes "x = of_int a / of_int b" "b ≠ 0" "coprime b m"
  shows   "[x = of_int a / of_int b] (qmod m)"
proof -
  obtain a' b' where ab: "quotient_of x = (a', b')"
    using prod.exhaust by blast
  have "(a', b') = Rat.normalize (a, b)"
    using assms ab by (metis Fract_of_int_quotient quotient_of_Fract)
  hence "b' dvd b"
    unfolding Rat.normalize_def
    by (metis assms(2) dvd_def dvd_div_mult_self gcd_dvd2 minus_dvd_iff
snd_eqD)
  with assms have "coprime b' m"
    by (meson coprime_divisors dvd_refl)
  thus ?thesis
    unfolding assms(1) using ab by (intro qcong_refl) (auto simp: assms(1))
qed

lemma qcong_add:
  assumes "[x = x'] (qmod m)" "[y = y'] (qmod m)" "m > 0"
  shows   "[x + y = x' + y'] (qmod m)"
proof -
  obtain a b where ab: "quotient_of x = (a, b)"
    using prod.exhaust by blast
  obtain c d where cd: "quotient_of y = (c, d)"
    using prod.exhaust by blast
  obtain a' b' where ab': "quotient_of x' = (a', b')"
    using prod.exhaust by blast
  obtain c' d' where cd': "quotient_of y' = (c', d')"
    using prod.exhaust by blast
  have x_eq: "x = of_int a / of_int b" and y_eq: "y = of_int c / of_int
d"
    using ab cd quotient_of_div by blast+
```

```
  have x'_eq: "x' = of_int a' / of_int b'" and y'_eq: "y' = of_int c'
/ of_int d'"
    using ab' cd' quotient_of_div by blast+
  have pos: "b > 0" "d > 0" "b' > 0" "d' > 0"
    using ab cd ab' cd' by (simp_all add: quotient_of_denom_pos)
  have coprime: "coprime b m" "coprime d m" "coprime b' m" "coprime d'
m"
    using ab cd ab' cd' assms unfolding qcong_def by auto

  have "[x + y = of_int (a * d + b * c) / of_int (b * d)] (qmod m)"
    using pos coprime by (intro qcong_fractionI) (auto simp: x_eq y_eq
field_simps)
  also have "[of_int (a * d + b * c) / of_int (b * d) = of_int (a' * d'
+ b' * c') / of_int (b' * d')] (qmod m)"
  proof (subst qcong_fraction_iff)
    have cong1: "[a * b' = b * a'] (mod m)"
      using assms(1) pos coprime ‹m > 0› unfolding x_eq x'_eq
      by (subst (asm) qcong_fraction_iff) auto
    have cong2:"[c * d' = d * c'] (mod m)"
      using assms(2) pos coprime ‹m > 0› unfolding y_eq y'_eq
      by (subst (asm) qcong_fraction_iff) auto
    have "[(a * d + b * c) * (b' * d') = (a * b') * d * d' + (c * d')
* b * b'] (mod m)"
      by (simp add: algebra_simps)
    also have "[(a * b') * d * d' + (c * d') * b * b' = (b * a') * d *
d' + (d * c') * b * b'] (mod m)"
      by (intro cong1 cong2 cong_mult[OF _ cong_refl] cong_add cong_refl)
    also have "(b * a') * d * d' + (d * c') * b * b' = b * d * (a' * d'
+ b' * c')"
      by (simp add: algebra_simps)
    finally show "[(a * d + b * c) * (b' * d') = b * d * (a' * d' + b'
* c')] (mod m)" .
  qed (use pos coprime ‹m > 0› in auto)
  also have "[of_int (a' * d' + b' * c') / of_int (b' * d') = x' + y']
(qmod m)"
    by (rule qcong_sym, rule qcong_fractionI) (use pos coprime in ‹auto
simp: x'_eq y'_eq field_simps›)
  finally show ?thesis .
qed

lemma qcong_diff:
  assumes "[x = x'] (qmod m)" "[y = y'] (qmod m)" "m > 0"
  shows   "[x - y = x' - y'] (qmod m)"
  using qcong_add[OF assms(1) qcong_minus[OF assms(2)]] ‹m > 0› by simp

lemma qcong_mult:
  assumes "[x = x'] (qmod m)" "[y = y'] (qmod m)" "m > 0"
  shows   "[x * y = x' * y'] (qmod m)"
proof -
```

```
  obtain a b where ab: "quotient_of x = (a, b)"
    using prod.exhaust by blast
  obtain c d where cd: "quotient_of y = (c, d)"
    using prod.exhaust by blast
  obtain a' b' where ab': "quotient_of x' = (a', b')"
    using prod.exhaust by blast
  obtain c' d' where cd': "quotient_of y' = (c', d')"
    using prod.exhaust by blast
  have x_eq: "x = of_int a / of_int b" and y_eq: "y = of_int c / of_int
d"
    using ab cd quotient_of_div by blast+
  have x'_eq: "x' = of_int a' / of_int b'" and y'_eq: "y' = of_int c'
/ of_int d'"
    using ab' cd' quotient_of_div by blast+
  have pos: "b > 0" "d > 0" "b' > 0" "d' > 0"
    using ab cd ab' cd' by (simp_all add: quotient_of_denom_pos)
  have coprime: "coprime b m" "coprime d m" "coprime b' m" "coprime d'
m"
    using ab cd ab' cd' assms unfolding qcong_def by auto

  have "[x * y = of_int (a * c) / of_int (b * d)] (qmod m)"
    using pos coprime by (intro qcong_fractionI) (auto simp: x_eq y_eq
field_simps)
  also have "[of_int (a * c) / of_int (b * d) = of_int (a' * c') / of_int
(b' * d')] (qmod m)"
  proof (subst qcong_fraction_iff)
    have cong1: "[a * b' = b * a'] (mod m)"
      using assms(1) pos coprime ‹m > 0› unfolding x_eq x'_eq
      by (subst (asm) qcong_fraction_iff) auto
    have cong2:"[c * d' = d * c'] (mod m)"
      using assms(2) pos coprime ‹m > 0› unfolding y_eq y'_eq
      by (subst (asm) qcong_fraction_iff) auto
    have "[a * c * (b' * d') = (a * b') * (c * d')] (mod m)"
      by (simp add: algebra_simps)
    also have "[(a * b') * (c * d') = (b * a') * (d * c')] (mod m)"
      by (intro cong1 cong2 cong_mult)
    also have "(b * a') * (d * c') = b * d * (a' * c')"
      by (simp add: algebra_simps)
    finally show "[a * c * (b' * d') = b * d * (a' * c')] (mod m)" .
  qed (use pos coprime ‹m > 0› in auto)
  also have "[of_int (a' * c') / of_int (b' * d') = x' * y'] (qmod m)"
    by (rule qcong_sym, rule qcong_fractionI) (use pos coprime in ‹auto
simp: x'_eq y'_eq field_simps›)
  finally show ?thesis .
qed

lemma qcong_divide_of_int:
  assumes "[x = x'] (qmod m)" "[c = c'] (mod m)" "coprime c m" "c ≠ 0"
"c' ≠ 0" "m > 0"
```

```
  shows   "[x / of_int c = x' / of_int c'] (qmod m)"
proof -
  obtain a b where ab: "quotient_of x = (a, b)"
    using prod.exhaust by blast
  obtain a' b' where ab': "quotient_of x' = (a', b')"
    using prod.exhaust by blast
  have x_eq: "x = of_int a / of_int b" and x'_eq: "x' = of_int a' / of_int
b'"
    using ab ab' quotient_of_div by blast+
  have pos: "b > 0" "b' > 0"
    using ab ab' by (simp_all add: quotient_of_denom_pos)
  have coprime: "coprime b m" "coprime b' m"
    using ab ab' assms unfolding qcong_def by auto
  from assms have coprime': "coprime c' m"
    using cong_imp_coprime by blast

  have "[x / of_int c = of_int a / of_int (b * c)] (qmod m)"
    using pos coprime assms by (intro qcong_fractionI) (auto simp: x_eq
field_simps)
  also have "[of_int a / of_int (b * c) = of_int a' / of_int (b' * c')]
(qmod m)"
  proof (subst qcong_fraction_iff)
    have cong: "[a * b' = b * a'] (mod m)"
      using assms(1) pos coprime ‹m > 0› unfolding x_eq x'_eq
      by (subst (asm) qcong_fraction_iff) auto
    have "[a * (b' * c') = (a * b') * c'] (mod m)"
      by (simp add: algebra_simps)
    also have "[(a * b') * c' = (b * a') * c] (mod m)"
      by (intro cong cong_sym[OF assms(2)] cong_mult)
    also have "(b * a') * c = b * c * a'"
      by (simp add: algebra_simps)
    finally show "[a * (b' * c') = b * c * a'] (mod m)" .
  qed (use pos coprime coprime' assms in auto)
  also have "[of_int a' / of_int (b' * c') = x' / of_int c'] (qmod m)"
    by (rule qcong_sym, rule qcong_fractionI)
       (use pos coprime coprime' assms in ‹auto simp: x'_eq field_simps›)
  finally show ?thesis .
qed

lemma qcong_mult_of_int_cancel_left:
  assumes "[of_int a * b = of_int a * c] (qmod m)" "coprime a m" "a ≠
0" "m > 0"
  shows   "[b = c] (qmod m)"
proof -
  have "[of_int a * b / of_int a = of_int a * c / of_int a] (qmod m)"
    by (rule qcong_divide_of_int) (use assms in auto)
  thus ?thesis
    using assms(3) by simp
qed
```

```
lemma qcong_pow:
  assumes "[a = b] (qmod m)" "m > 0"
  shows   "[a ^ n = b ^ n] (qmod m)"
  by (induction n) (auto intro!: qcong_mult assms)

lemma qcong_sum:
  "[sum f A = sum g A] (qmod m)" if "⋀x. x ∈ A ⟹ [f x = g x] (qmod m)"
"m > 0"
  using that by (induct A rule: infinite_finite_induct) (auto intro: qcong_add)

lemma qcong_prod:
  "[prod f A = prod g A] (qmod m)" if "(⋀x. x ∈ A ⟹ [f x = g x] (qmod
m))" "m > 0"
  using that by (induct A rule: infinite_finite_induct) (auto intro: qcong_mult)

lemma qcong_modulus_abs_1:
  assumes "|n| = 1"
  shows   "[a = b] (qmod n)"
  using assms by (auto simp: qcong_def qmod_def case_prod_unfold abs_if
split: if_splits)

lemma qcong_divide_of_int_left_iff:
  assumes "coprime c n" "c ≠ 0" ‹n > 0›
  shows   "[a / of_int c = b] (qmod n) ⟷ [a = b * of_int c] (qmod n)"
proof
  assume *: "[a / of_int c = b] (qmod n)"
  hence "[a / of_int c * of_int c = b * of_int c] (qmod n)"
    by (rule qcong_mult) (use assms in auto)
  also have "a / of_int c * of_int c = a"
    using assms by simp
  finally show "[a = b * of_int c] (qmod n)" .
next
  assume "[a = b * of_int c] (qmod n)"
  hence "[a / of_int c = b * of_int c / of_int c] (qmod n)"
    by (intro qcong_divide_of_int assms cong_refl)
  also have "b * of_int c / of_int c = b"
    using assms by simp
  finally show "[a / of_int c = b] (qmod n)" .
qed

lemma qcong_divide_of_nat_left_iff:
  assumes "coprime (int c) n" "c ≠ 0" "n > 0"
  shows   "[a / of_nat c = b] (qmod n) ⟷ [a = b * of_nat c] (qmod n)"
  using qcong_divide_of_int_left_iff[of "int c" n a b] assms by simp

lemma qcong_divide_of_int_right_iff:
  assumes "coprime c n" "c ≠ 0" "n > 0"
  shows   "[a = b / of_int c] (qmod n) ⟷ [a * of_int c = b] (qmod n)"
```

```
  using qcong_divide_of_int_left_iff[OF assms, of b a] by (simp add: qcong_sym_eq)

lemma qcong_divide_of_nat_right_iff:
  assumes "coprime (int c) n" "c ≠ 0" "n > 0"
  shows   "[a = b / of_nat c] (qmod n) ⟷ [a * of_nat c = b] (qmod n)"
  using qcong_divide_of_int_right_iff[of "int c" n a b] assms by simp

lemma qcong_qmultiplicity_pos_transfer:
  assumes "[x = y] (qmod m)" "qmultiplicity m x > 0"
  shows   "y = 0 ∨ qmultiplicity m y > 0"
proof -
  obtain a b where ab: "quotient_of x = (a, b)"
    using prod.exhaust by blast
  obtain c d where cd: "quotient_of y = (c, d)"
    using prod.exhaust by blast
  have "b > 0" "d > 0"
    using ab cd quotient_of_denom_pos by blast+
  have coprime: "coprime a b" "coprime c d"
    using ab cd quotient_of_coprime by blast+
  have *: "coprime b m" "coprime d m" "[a * modular_inverse m b = c *
modular_inverse m d] (mod m)"
    using assms(1) unfolding qcong_def ab cd qmod_def by (auto simp: Cong.cong_def)

  have x: "x = of_int a / of_int b" and y: "y = of_int c / of_int d"
    using ab cd by (simp_all add: quotient_of_div)
  from assms have "multiplicity m a > multiplicity m b"
    unfolding qmultiplicity_def ab cd by auto
  hence "m dvd a"
    using not_dvd_imp_multiplicity_0 by force
  hence "[0 = a * modular_inverse m b] (mod m)"
    by (auto simp: Cong.cong_def)
  also have "[a * modular_inverse m b = c * modular_inverse m d] (mod
m)"
    by fact
  finally have "m dvd c * modular_inverse m d"
    using cong_dvd_iff by blast
  moreover have "coprime (modular_inverse m d) m"
    using * by auto
  ultimately have "m dvd c"
    using coprime_commute coprime_dvd_mult_left_iff by blast
  hence "c = 0 ∨ multiplicity m c ≥ 1"
    by (metis <multiplicity m b < multiplicity m a> dual_order.refl less_one
linorder_not_le
          multiplicity_eq_zero_iff multiplicity_unit_left)
  hence "c = 0 ∨ multiplicity m c > multiplicity m d"
    using coprime(2) <m dvd c>
    by (metis Suc_le_eq coprime_common_divisor multiplicity_unit_left
            not_dvd_imp_multiplicity_0 One_nat_def)
  thus ?thesis
```

```
    unfolding qmultiplicity_def cd unfolding y by auto
qed

end
```

3 The Voronoi congruence

```
theory Voronoi_Congruence
  imports Kummer_Library Rat_Congruence
begin

unbundle bernoulli_notation

lemma sum_of_powers_mod_prime:
  assumes p: "prime p"
  shows   "[(∑x=1..<p. int x ^ m) = (if (p - 1) dvd m then -1 else 0)]
(mod p)"
proof -
  obtain g where g: "residue_primroot p g"
    using assms prime_gt_1_nat prime_primitive_root_exists by auto
  have "coprime g p"
    using g by (auto simp: residue_primroot_def coprime_commute)
  have bij: "bij_betw (λi. g ^ i mod p) {..<p-1} {0<..<p}"
    using residue_primroot_is_generator[OF _ g] p
    by (simp add: totient_prime totatives_prime prime_gt_Suc_0_nat)
  have "(∑x=1..<p. int x ^ m) = (∑x∈{0<..<p}. int x ^ m)"
    by (intro sum.cong) auto
  also have "... = (∑i<p-1. int (g ^ i mod p) ^ m)"
    by (subst sum.reindex_bij_betw[OF bij, symmetric]) auto
  also have "[... = (∑i<p-1. int (g ^ i) ^ m)] (mod p)"
    by (intro cong_sum cong_pow cong_int) auto
  also have "(∑i<p-1. int (g ^ i) ^ m) = (∑i<p-1. int (g ^ m) ^ i)"
    by (simp flip: power_mult add: mult.commute)
  also have "[... = (∑i<p-1. int (g ^ m mod p) ^ i)] (mod p)"
    by (intro cong_sum cong_pow cong_int) auto
  also have "[(∑i<p-1. int (g ^ m mod p) ^ i) = (if (p - 1) dvd m then
-1 else 0)] (mod p)"
  proof (cases "(p - 1) dvd m")
    case True
    have "[(∑i<p-1. int (g ^ m mod p) ^ i) = (∑i<p-1. int (g ^ 0) ^
i)] (mod p)"
      using ‹coprime g p› p True
      by (intro cong_sum cong_pow cong_pow_totient cong_mod_left cong_refl
cong_int)
         (auto simp: cong_0_iff totient_prime)
    also have "(∑i<p-1. int (g ^ 0) ^ i) = int p - 1"
      using prime_gt_1_nat[OF p] by (simp add: of_nat_diff)
    also have "[int p - 1 = 0 - 1] (mod int p)"
      by (intro cong_diff) (auto simp: Cong.cong_def)
```

```
    finally show ?thesis
      using True by auto
  next
    case False
    have not_cong: "[g ^ m ≠ 1] (mod p)"
      using False g by (metis assms ord_divides residue_primroot_def totient_prime)
    hence neq1: "g ^ m mod p ≠ 1"
      using prime_gt_1_nat[OF p] by (auto simp: Cong.cong_def)
    have "real_of_int (int (∑i<p-1. (g ^ m mod p) ^ i)) = (∑i<p-1.
real (g ^ m mod p) ^ i)"
      by simp
    also have "... = (1 - real (g ^ m mod p) ^ (p - 1)) / (1 - real (g
^ m mod p))"
      using prime_gt_1_nat[OF p] neq1 by (subst sum_gp_strict) (auto simp:
of_nat_diff)
    finally have "real_of_int (int (∑i<p-1. (g ^ m mod p) ^ i)) * (1 -
real (g ^ m mod p)) =
                  1 - real (g ^ m mod p) ^ (p - 1)"
      using neq1 by (simp add: field_simps)
    also have "real_of_int (int (∑i<p-1. (g ^ m mod p) ^ i)) * (1 - real
(g ^ m mod p)) =
               real_of_int ((∑i<p-1. int (g ^ m mod p) ^ i) * (1 - int
(g ^ m mod p)))"
      by simp
    also have "1 - real (g ^ m mod p) ^ (p - 1) = real_of_int (1 - int
(g ^ m mod p) ^ (p - 1))"
      by simp
    finally have "(∑i<p-1. int (g ^ m mod p) ^ i) * (1 - int (g ^ m mod
p)) = 1 - int (g ^ m mod p) ^ (p - 1)"
      by linarith
    also have "[1 - int (g ^ m mod p) ^ (p - 1) = 1 - int (g ^ m) ^ (p
- 1)] (mod p)"
      by (intro cong_diff cong_int cong_pow) auto
    also have "int (g ^ m) ^ (p - 1) = int ((g ^ (p - 1)) ^ m)"
      by (simp flip: power_mult add: mult.commute)
    also have "[1 - int ((g ^ (p - 1)) ^ m) = 1 - int ((g ^ 0) ^ m)] (mod
p)"
      using ‹coprime g p› p
      by (intro cong_diff cong_int cong_pow_totient cong_refl)
         (auto simp: Cong.cong_def totient_prime)
    finally have "[(∑i<p-1. int (g ^ m mod p) ^ i) * (1 - int (g ^ m mod
p)) = 0] (mod p)"
      by simp
    hence "p dvd (∑i<p-1. int (g ^ m mod p) ^ i) * (1 - int (g ^ m mod
p))"
      by (simp add: cong_0_iff)
    moreover from not_cong have "¬p dvd (1 - int (g ^ m mod p))"
      by (metis of_nat_1 cong_iff_dvd_diff mod_mod_trivial nat_int of_nat_mod
Cong.cong_def)
```

```
    ultimately have "p dvd (∑i<p-1. int (g ^ m mod p) ^ i)"
      using p by (subst (asm) prime_dvd_mult_iff) auto
    thus ?thesis
      using False by (simp add: Cong.cong_def)
  qed
  finally show ?thesis .
qed

lemma sum_of_powers_mod_prime':
  fixes p m :: nat
  assumes p: "prime p" "¬(p - 1) dvd m"
  shows   "[(∑x=1..<p. x ^ m) = 0] (mod p)"
proof -
  have "[(∑x=1..<p. int x ^ m) = int 0] (mod p)"
    using sum_of_powers_mod_prime[of p m] assms by simp
  also have "(∑x=1..<p. int x ^ m) = int (∑x=1..<p. x ^ m)"
    by simp
  finally show ?thesis
    using cong_int_iff by blast
qed

lemma voronoi_congruence_aux1:
  assumes "prime p" "j ≥ 4"
  shows   "multiplicity p (j + 1) ≤ (if p ∈ {2, 3} then 1 else 0) + j
- 2"
proof (cases "p ∈ {2, 3}")
  case True
  have "multiplicity p (j + 1) < j"
  proof (rule multiplicity_lessI)
    have "2 ^ (n + 2) > n + 3" for n
      by (induction n) auto
    from this[of "j - 2"] have "j + 1 < 2 ^ j"
      using assms(2) by (simp del: power_Suc add: Suc_diff_Suc eval_nat_numeral)
    also have "2 ^ j ≤ p ^ j"
      using True by (intro power_mono) auto
    finally have "p ^ j > j + 1" .
    thus "¬p ^ j dvd j + 1"
      using dvd_imp_le by force
  qed (use assms in auto)
  with True show ?thesis
    by simp
next
  case False
  have "p ≠ 0" "p ≠ 1" "p ≠ 4"
    using assms by auto
  with False have "p ≥ 5"
    by force
  have "multiplicity p (j + 1) < j - 1"
  proof (rule multiplicity_lessI)
```

```
    have "5 ^ (n + 1) > n + 3" for n
      by (induction n) auto
    from this[of "j - 2"] have "j + 1 < 5 ^ (j - 1)"
      using assms(2) by (simp del: power_Suc add: Suc_diff_Suc eval_nat_numeral)
    also have "5 ^ (j - 1) ≤ p ^ (j - 1)"
      using ‹p ≥ 5› by (intro power_mono) auto
    finally have "p ^ (j - 1) > j + 1" .
    thus "¬p ^ (j - 1) dvd j + 1"
      using dvd_imp_le by force
  qed (use assms in auto)
  with False show ?thesis
    by simp
qed

context
  fixes S :: "nat ⇒ nat ⇒ nat" and D :: "nat ⇒ nat" and N :: "nat ⇒
int"
  defines "S ≡ (λk n. ∑r<n. r ^ k)"
  defines "N ≡ bernoulli_num" and "D ≡ bernoulli_denom"
begin

lemma voronoi_congruence_aux2:
  fixes k n :: nat
  assumes k: "even k" "k ≥ 2" and n: "n > 0"
  shows   "real (S k n) = (∑j≤k. real (k choose j) / real (j + 1) *
bernoulli (k - j) * real (n ^ (j + 1)))"
proof -
  have "real (S k n) = (∑r≤n-1. real r ^ k)"
    using n unfolding S_def of_nat_sum by (intro sum.cong) auto
  also have "... = (bernpoly (Suc k) (real n) - bernoulli (Suc k)) / (real
k + 1)"
    using n by (subst sum_of_powers) (auto simp: of_nat_diff)
  also have "bernoulli (Suc k) = 0"
    using k by (intro bernoulli_odd_eq_0) auto
  also have "(bernpoly (Suc k) (real n) - 0) / (real k + 1) =
               bernpoly (Suc k) (real n) / real (k + 1)"
    by simp
  also have "bernpoly (Suc k) (real n) / real (k + 1) = (∑j≤Suc k. (Suc
k choose j) / (k + 1) * bernoulli (Suc k - j) * n ^ j)"
    by (subst bernpoly_altdef)
       (auto simp: sum_divide_distrib sum_distrib_left sum_distrib_right
field_simps simp del: of_nat_Suc)
  also have "... = (∑j=1..Suc k. (Suc k choose j) / (k+1) * bernoulli
(Suc k - j) * n ^ j)"
    using k by (intro sum.mono_neutral_right) (auto simp: not_le simp:
bernoulli_odd_eq_0)
  also have "... = (∑j≤k. (Suc k choose Suc j) / (k+1) * bernoulli (k
- j) * n ^ (j + 1))"
    by (intro sum.reindex_bij_witness[of _ "λj. j+1" "λj. j-1"]) (auto
```

```
simp: of_nat_diff)
  also have "... = (∑j≤k. (k choose j) / (j+1) * bernoulli (k - j) *
n ^ (j + 1))"
  proof (intro sum.cong refl, goal_cases)
    case (1 j)
    have "real (Suc j * (Suc k choose Suc j)) = real (Suc k * (k choose
j))"
      by (subst Suc_times_binomial_eq) (simp add: mult_ac)
    thus ?case
      unfolding of_nat_mult by (simp add: field_simps del: of_nat_Suc
binomial_Suc_Suc)
  qed
  finally show ?thesis .
qed

lemma voronoi_congruence_aux3:
  fixes k n :: nat
  assumes k: "even k" "k ≥ 2" and n: "n > 0"
  shows   "[D k * S k n = N k * n] (mod (n^2))"
proof -
  note [simp del] = div_mult_self3 div_mult_self4 div_mult_self2 div_mult_self1
  define A :: "nat ⇒ rat"
    where "A = (λj. of_nat (k choose j) * of_int (N (k - j)) / of_nat
(D (k - j)) * of_nat n powi (int j - 1) / of_nat (Suc j))"
  have "real (S k n) = (∑j≤k. real (k choose j) / real (j + 1) * bernoulli
(k - j) * real (n ^ (j + 1)))"
    by (rule voronoi_congruence_aux2) fact+
  also have "... = (∑j≤k. of_rat (A j) * n^2)"
    unfolding A_def using n
    by (intro sum.cong)
       (auto simp: power_int_diff power2_eq_square N_def D_def bernoulli_conv_num_denom

                   of_rat_mult of_rat_divide field_simps of_rat_power
simp del: of_nat_Suc)
  also have "... = (∑j≤k. of_rat (A j)) * real (n ^ 2)"
    by (simp add: sum_distrib_right)
  also have "(∑j≤k. of_rat (A j)) = (∑j∈insert 0 {1..k}. of_rat (A
j))"
    using k by (intro sum.cong) auto
  also have "... = of_rat (A 0) + (∑j=1..k. of_rat (A j))"
    by (subst sum.insert) auto
  also have "of_rat (A 0) = bernoulli k / n"
    using n by (auto simp: A_def bernoulli_conv_num_denom N_def D_def
field_simps
                           of_rat_mult of_rat_divide)
  also have "(bernoulli k / n + (∑j=1..k. of_rat (A j))) * real (n ^
2) =
             real n * bernoulli k + (∑j=1..k. of_rat (A j)) * real (n
^ 2)"
```

```
    using n by (simp add: field_simps power2_eq_square)
  finally have eq1: "real (S k n) = real n * bernoulli k + (∑j=1..k. of_rat
(A j)) * real (n ^ 2)" .

  have "∃ab. coprime (fst ab) (snd ab) ∧ coprime (snd ab) (int n) ∧ snd
ab > 0 ∧
          6 * A j = of_int (fst ab) / of_int (snd ab)" (is "∃ab. ?P j
ab") if j: "j ∈ {1..k}" for j
  proof (cases "A j = 0")
    case True
    thus ?thesis
      by (intro exI[of _ "(0, 1)"]) auto
  next
    case False
    obtain a b :: int where ab: "coprime a b" "b > 0" "6 * A j = of_int
a / of_int b"
      by (metis Fract_of_int_quotient Rat_cases)
    have *: "qmultiplicity p (6 * A j) ≥ 0" if p: "prime p" "p dvd n"
for p
    proof -
      consider "j = 1" | "j = 2" | "j = k - 1" | "odd j" "j ≠ k - 1" |
"j ≥ 3" "even j"
        using j by force
      thus ?thesis
      proof cases
        assume [simp]: "j = 1"
        show ?thesis
        proof (cases "k = 2")
          case False
          have "(of_rat (6 * A j) :: real) = 3 * real k * bernoulli (k
- 1)"
            by (simp add: A_def N_def D_def of_rat_mult of_rat_divide
bernoulli_conv_num_denom)
          also have "bernoulli (k - 1) = 0"
            using k False by (subst bernoulli_odd_eq_0) auto
          finally show ?thesis
            by simp
        next
          case [simp]: True
          have "qmultiplicity (int p) (6 * A j) = qmultiplicity (int p)
3"
            by (simp add: A_def N_def D_def bernoulli_num_def floor_minus)
          also have "... ≥ 0"
            by auto
          finally show ?thesis .
        qed
      next
        assume [simp]: "j = 2"
        have "6 * A j = rat_of_int (2 * n * (k choose 2) * N (k - 2))
```

```
/ rat_of_int (D (k - 2))"
          by (simp add: A_def)
        also have "qmultiplicity (int p) ... =
                     int (multiplicity (int p) (2 * int n * int (k choose
2) * N (k - 2))) -
                     int (multiplicity (int p) (D (k - 2)))" using k n
p
          by (subst qmultiplicity_divide_of_int) (auto simp: D_def N_def
bernoulli_num_eq_0_iff)
        also have "... ≥ int 1 - int 1"
        proof (intro diff_mono; unfold of_nat_le_iff)
          show "multiplicity (int p) (2 * int n * int (k choose 2) *
N (k - 2)) ≥ 1"
            using k n j p by (intro multiplicity_geI) (auto simp: N_def
bernoulli_num_eq_0_iff)
        next
          show "multiplicity (int p) (D (k - 2)) ≤ 1" using p
            by (intro squarefree_imp_multiplicity_prime_le_1)
               (auto simp: D_def squarefree_bernoulli_denom)
        qed
        finally show ?thesis
          by simp
      next
        assume [simp]: "j = k - 1"
        have "6 * A j = -rat_of_nat (3 * n ^ (k - 2))"
          using k binomial_symmetric[of 1 k, symmetric]
          by (auto simp: A_def D_def N_def bernoulli_num_def floor_minus
of_nat_diff
                         power_int_def nat_diff_distrib)
        also have "qmultiplicity (int p) ... ≥ 0"
          unfolding qmultiplicity_minus by (subst qmultiplicity_of_nat)
auto
        finally show ?thesis .
      next
        assume "odd j" "j ≠ k - 1"
        with k j have "odd (k - j)" "k - j ≠ 1"
          by auto
        hence "6 * A j = 0"
          by (auto simp: A_def bernoulli_num_odd_eq_0 N_def)
        thus ?thesis
          by simp
      next
        assume "j ≥ 3" "even j"
        with j have j: "j ∈ {3..k}"
          by auto
        have "6 * A j = rat_of_int (6 * int (k choose j) * N (k - j) *
n ^ (j - 1)) /
                        rat_of_int (int (D (k - j)) * int (j + 1))"
          using j by (simp add: A_def power_int_def nat_diff_distrib algebra_simps)
```

```
        also have "qmultiplicity p ... =
                     int (multiplicity (int p) (6 * int (k choose j) *
N (k - j) * n ^ (j - 1))) -
                     int (multiplicity (int p) (int (D (k - j)) * int
(j + 1)))" using k n p j ‹even j›
          by (subst qmultiplicity_divide_of_int) (auto simp: D_def N_def
bernoulli_num_eq_0_iff)
        also have "... ≥ ((if p ∈ {2,3} then 1 else 0) + int j - 1) -
((if p ∈ {2,3} then 1 else 0) + int j - 1)"
        proof (intro diff_mono, goal_cases)
          case 1
          have "multiplicity (int p) (6 * int (k choose j) * N (k - j)
* int (n ^ (j - 1))) =
                multiplicity (int p) (6 * (int (k choose j) * N (k - j)
* int (n ^ (j - 1))))"
            by (simp add: mult_ac)
          also have "multiplicity (int p) (6 * (int (k choose j) * N (k
- j) * int (n ^ (j - 1)))) =
                  multiplicity (int p) (2 * 3) + multiplicity (int p)
(int (k choose j) * N (k - j) * int (n ^ (j - 1)))"
            using k n p j ‹even j›
            by (subst prime_elem_multiplicity_mult_distrib) (auto simp:
N_def bernoulli_num_eq_0_iff)
          also have "multiplicity (int p) (2 * 3) = (if p ∈ {2, 3} then
1 else 0)"
            using p by (subst prime_elem_multiplicity_mult_distrib) (auto
simp: multiplicity_prime_prime)
          finally have "multiplicity (int p) (6 * int (k choose j) * N
(k - j) * int (n ^ (j - 1))) =
                        (if p ∈ {2, 3} then 1 else 0) +
                        multiplicity (int p) (int (k choose j) * N (k
- j) * int (n ^ (j - 1)))" .
          moreover have "p ^ (j - 1) dvd n ^ (j - 1)"
            using p by (intro dvd_power_same)
          hence "multiplicity (int p) (int (k choose j) * N (k - j) *
int (n ^ (j - 1))) ≥ j - 1"
            using j k n ‹even j› p by (intro multiplicity_geI) (auto simp:
N_def bernoulli_num_eq_0_iff)
          hence "int (multiplicity (int p) (int (k choose j) * N (k -
j) * int (n ^ (j - 1)))) ≥ int j - 1"
            using ‹j ≥ 3› by linarith
          ultimately show ?case
            by simp
        next
          case 2
          have "multiplicity (int p) (int (D (k - j)) * int (j + 1)) =
                multiplicity (int p) (int (D (k - j))) + multiplicity
(int p) (int (j + 1))"
            using p by (subst prime_elem_multiplicity_mult_distrib) (auto
```

```
simp: D_def)
          moreover have "multiplicity (int p) (int (D (k - j))) ≤ 1"
using p
            by (intro squarefree_imp_multiplicity_prime_le_1)
               (auto simp: D_def squarefree_bernoulli_denom)
          moreover have "multiplicity (int p) (int (j + 1)) ≤ (if p
∈ {2, 3} then 1 else 0) + j - 2"
            by (subst multiplicity_int, rule voronoi_congruence_aux1)
               (use p j ‹even j› in auto)
          hence "int (multiplicity (int p) (int (j + 1))) ≤ (if p ∈ {2,
3} then 1 else 0) + int j - 2"
            using j by auto
          ultimately show ?case
            using ‹j ≥ 3› by linarith
        qed
        finally show ?thesis
          by simp
      qed
    qed
    have "coprime b n"
    proof (subst coprime_commute, rule coprimeI_by_prime_factors)
      fix p assume p: "p ∈ prime_factors (int n)"
      hence "p > 0"
        by (auto simp: in_prime_factors_iff prime_gt_0_int)
      hence "qmultiplicity (nat p) (rat_of_int a / rat_of_int b) ≥ 0"
        using *[of "nat p"] p unfolding ab by (auto simp: in_prime_factors_iff
nat_dvd_iff)
      thus "¬p dvd b"
        using ab False ‹p > 0› p by (subst (asm) qmultiplicity_nonneg_iff)
auto
    qed (use n in auto)
    with ab show ?thesis
      by (intro exI[of _ "(a, b)"]) auto
  qed
  then obtain f where f: "⋀j. j ∈ {1..k} ⟹ ?P j (f j)"
    by metis

  define B :: int where "B = Lcm ((snd ∘ f) ` {1..k})"
  have B: "coprime B n"
    unfolding B_def using f by (auto intro!: coprime_Lcm_left)
  define b' where "b' = (λj. B div snd (f j))"
  define T where "T = (∑j=1..k. fst (f j) * b' j)"

  have "real_of_int (B * int (D k * S k n)) =
        real_of_int B * (real (D k) * bernoulli k) * real n +
        real (D k) / 6 * (∑j=1..k. real_of_rat (6 * A j * of_int B))
* real n ^ 2"
    unfolding of_int_mult of_nat_mult of_int_of_nat_eq
    by (subst eq1) (simp_all add: algebra_simps of_rat_mult sum_distrib_left)
```

```
  also have "real (D k) * bernoulli k = real_of_int (N k)"
    by (simp add: bernoulli_conv_num_denom N_def D_def)
  also have "(∑j=1..k. real_of_rat (6 * A j * of_int B)) = T"
    unfolding T_def of_int_sum
  proof (intro sum.cong refl, goal_cases)
    case j: (1 j)
    have "6 * A j = of_int (fst (f j)) / of_int (snd (f j))"
      using f[OF j] by auto
    also have "... * of_int B = of_int (fst (f j)) * (of_int B / of_int
(snd (f j)))"
      by simp
    also have "of_int B / of_int (snd (f j)) = (of_int (B div snd (f j))
:: rat)"
      by (subst of_int_div) (use j in ‹auto simp: B_def›)
    finally show ?case
      by (simp add: of_rat_mult b'_def)
  qed
  also have "real (D k) / 6 = real (D k div 6)"
    by (subst real_of_nat_div) (use k in ‹auto intro!: six_divides_bernoulli_denom
simp: D_def›)
  also have "of_int B * of_int (N k) * real n + real (D k div 6) * of_int
T * (real n)^2 =
             of_int (B * N k * int n + int (D k div 6) * T * int n ^ 2)"
    by (simp add: algebra_simps)
  finally have "B * int (D k * S k n) = B * N k * int n + int (D k div
6) * T * int n ^ 2"
    by linarith
  hence "[B * int (D k * S k n) = B * N k * int n + int (D k div 6) *
T * int n ^ 2] (mod (int n ^ 2))"
    by simp
  also have "[B * N k * int n + int (D k div 6) * T * int n ^ 2 =
              B * N k * int n + int (D k div 6) * T * 0] (mod (int n ^
2))"
    by (intro cong_add cong_mult cong_refl) (auto simp: Cong.cong_def)
  finally have "[B * int (D k * S k n) = B * (N k * int n)] (mod (int n)^2)"
    by (simp add: mult_ac)
  hence "[int (D k * S k n) = N k * int n] (mod (int n)^2)"
    by (subst (asm) cong_mult_lcancel) (use B in auto)
  thus ?thesis
    by simp
qed
```

Proposition 9.5.20

```
theorem voronoi_congruence:
  fixes k n :: nat and a :: int
  assumes k: "even k" "k ≥ 2" and n: "n > 0" and a: "coprime a n"
  shows   "[(a^k-1) * N k = k * a^(k-1) * D k * (∑m=1..<n. m^(k-1) *
((m * a) div n))] (mod n)"
proof -
```

```
  define a' where "a' = modular_inverse (int n) a"
  define q r where "q = (λm. (a * m) div n)" and "r = (λm. (a * m) mod
n)"

  have "S k n = (∑m=1..<n. m ^ k)"
    using k unfolding S_def by (intro sum.mono_neutral_right) auto
  hence "a ^ k * S k n = (∑m=1..<n. (a * m) ^ k)"
    by (simp add: sum_distrib_left power_mult_distrib)
  also have "[(∑m=1..<n. (a * m) ^ k) = (∑m=1..<n. r m ^ k + k * q
m * n * (m * a) ^ (k - 1))] (mod n^2)"
  proof (rule cong_sum)
    fix m :: nat
    have "[(∑j≤k. int (k choose j) * (q m * n) ^ j * r m ^ (k - j))
=
            (∑j≤k. if j > 1 then 0 else (k choose j) * q m ^ j * r m
^ (k - j) * n ^ j)] (mod (n^2))"
    proof (intro cong_sum, goal_cases)
      case (1 j)
      have dvd: "int n ^ 2 dvd int n ^ j" if "j > 1" for j
        using that by (intro le_imp_power_dvd) auto
      have eq: "int (k choose j) * (q m * n) ^ j * r m ^ (k - j) =
                  int (k choose j) * q m ^ j * r m ^ (k - j) * n ^ j"
        using 1 by (simp add: algebra_simps flip: power_add)
      have "[int (k choose j) * q m ^ j * r m ^ (k - j) * n ^ j =
                   (if j > 1 then 0 else int (k choose j) * q m ^ j * r
m ^ (k - j) * n ^ j)] (mod (n^2))"
        using dvd by (auto simp: cong_0_iff)
      thus ?case
        by (simp only: eq)
    qed
    also have "(∑j≤k. if j > 1 then 0 else (k choose j) * q m ^ j *
r m ^ (k - j) * n ^ j) =
                 (∑j∈{0,1}. (k choose j) * q m ^ j * r m ^ (k - j) * n
^ j)"
      using k by (intro sum.mono_neutral_cong_right) auto
    also have "... = r m ^ k + (k * q m * n) * r m ^ (k-1)"
      by simp
    also have "[(k * q m * n) * (q m * 0 + r m) ^ (k-1) =
                  (k * q m * n) * (q m * n + r m) ^ (k-1)] (mod (int n)^2)"
      by (intro cong_mult_square cong_pow cong_add cong_mult cong_refl)
(auto simp: Cong.cong_def)
    hence "[r m ^ k + (k * q m * n) * r m ^ (k-1) =
              r m ^ k + (k * q m * n) * (q m * n + r m) ^ (k-1)] (mod
n^2)"
      by (intro cong_add) simp_all
    also have "q m * n + r m = (m * a)"
      by (simp add: q_def r_def)
    also have "(∑j≤k. int (k choose j) * (q m * n) ^ j * r m ^ (k -
j)) = (q m * n + r m) ^ k"
```

```
    by (rule binomial_ring [symmetric])
  also have "q m * n + r m = a * m"
    by (simp add: q_def r_def)
  finally show "[(a * m) ^ k = r m ^ k + k * q m * n * (m * a) ^ (k -
1)] (mod n^2)" .
qed
also have "(∑m=1..<n. r m ^ k + k * q m * n * (m * a) ^ (k - 1)) =
           (∑m=1..<n. r m ^ k) + n * k * a^(k-1) * (∑m=1..<n. q m
* m^(k-1))"
  by (simp add: sum.distrib sum_distrib_left sum_distrib_right mult_ac
power_mult_distrib)
also have "(∑m=1..<n. r m ^ k) = (∑m=1..<int n. r m ^ k)"
  by (intro sum.reindex_bij_witness[of _ nat int]) auto
also have "... = (∑m=1..<int n. m ^ k)"
  by (rule sum.reindex_bij_betw)
     (use a bij_betw_int_remainders_mult[of a] in ‹simp_all add: r_def›)
also have "... = of_nat (∑m=1..<n. m ^ k)"
  unfolding of_nat_sum by (intro sum.reindex_bij_witness[of _ int nat])
auto
also have "(∑m=1..<n. m ^ k) = S k n"
  using k unfolding S_def by (intro sum.mono_neutral_left) auto
finally have "[a ^ k * S k n - S k n =
                (S k n + n * k * a^(k-1) * (∑m=1..<n. q m * m^(k-1)))
- S k n] (mod n^2)"
  by (intro cong_diff cong_refl)
hence "[D k * (n * k * a^(k-1) * (∑m=1..<n. q m * m^(k-1))) = D k
* ((a^k-1) * S k n)] (mod n^2)"
  by (intro cong_mult[OF cong_refl]) (simp_all add: algebra_simps cong_sym_eq)
also have "D k * ((a^k-1) * S k n) = (a^k-1) * (D k * S k n)"
  by (simp add: algebra_simps)
also have "[(a^k-1) * (D k * S k n) = (a^k-1) * (N k * int n)] (mod
n^2)"
  using k n by (intro cong_mult cong_refl voronoi_congruence_aux3)
finally have "[n * (k * a^(k-1) * D k * (∑m=1..<n. q m * m^(k-1))) =
n * ((a^k-1) * N k)] (mod n^2)"
  by (simp add: mult_ac)
hence "[k * a^(k-1) * D k * (∑m=1..<n. q m * m^(k-1)) = (a^k-1) *
N k] (mod n)"
  unfolding power2_eq_square of_nat_mult by (rule cong_mult_cancel)
(use n in auto)
thus ?thesis
  by (simp add: mult_ac cong_sym_eq q_def)
qed

corollary voronoi_congruence':
  fixes k p :: nat and a :: int
  assumes k: "even k" "k ≥ 2" and p: "prime p" "¬(p - 1) dvd k" and
a: "¬p dvd a" "[a ^ k ≠ 1] (mod p)"
  shows   "[ℬ k = of_int (k * a^(k-1)) / of_int (a^k - 1) *
```

```
                of_int (∑m=1..<p. m^(k-1) * ((m * a) div p))] (qmod
p)"
proof -
  have "¬p dvd D k"
    using p k by (auto simp: D_def prime_dvd_bernoulli_denom_iff)
  hence "coprime p (D k)"
    using p prime_imp_coprime by blast
  moreover have "¬p dvd (a ^ k - 1)"
    using a cong_iff_dvd_diff by blast
  hence "coprime p (a ^ k - 1)"
    using p prime_imp_coprime by (metis prime_nat_int_transfer)
  moreover have "coprime p a"
    using p prime_imp_coprime a by (metis prime_nat_int_transfer)
  ultimately have "[of_int (N k) / of_int (D k) =
        of_int (int k * a ^ (k-1) * (∑m=1..<p. int m^(k-1) * (int m
* a div p))) / of_int (a ^ k - 1)] (qmod p)"
    using assms voronoi_congruence[of k p a]
    by (subst qcong_fraction_iff) (auto simp: D_def coprime_commute prime_gt_0_nat
mult_ac)
  thus ?thesis
    by (simp add: bernoulli_rat_def mult_ac N_def D_def of_rat_divide)
qed

corollary voronoi_congruence_harvey:
  fixes k p :: nat and c a :: int and h :: "nat ⇒ rat"
  assumes k: "even k" "k ∈ {2..p-3}" and p: "prime p" "p ≥ 5" and c:
"c ∈ {0<..<p}" "[c^k ≠ 1] (mod p)"
  assumes a: "[a * c = 1] (mod p)"
  defines "h ≡ (λm. of_int (m - c * ((a * m) mod p)) / of_int p + of_int
(c - 1) / 2)"
  shows   "[ℬ k = rat_of_nat k / rat_of_int (1 - c ^ k) * (∑m=1..<p.
rat_of_nat m^(k-1) * h m)] (qmod p)"
proof -
  have [simp]: "p ≠ 0"
    using p by auto
  have "odd p"
    using p by (intro prime_odd_nat) auto
  note [simp del] = div_mult_self3 div_mult_self4 div_mult_self2 div_mult_self1
  have [simp]: "D k > 0"
    by (auto simp: D_def bernoulli_denom_pos)
  have "¬(p - 1) dvd k"
    using k p by (auto dest!: dvd_imp_le)
  hence [simp]: "coprime (D k) p"
    by (subst coprime_commute, intro prime_imp_coprime)
       (use p k in ‹auto simp: D_def prime_dvd_bernoulli_denom_iff›)
  have "¬p dvd c"
    using c p by (auto dest!: zdvd_imp_le)
  hence "coprime c p"
    using p prime_imp_coprime[of p c] by (auto simp: coprime_commute)
```

```
  have "coprime a p"
    using a coprime_iff_invertible_int by auto
  define n where "n = (c * a - 1) div p"
  have "[c * a = 1] (mod p)"
    using a by (simp add: mult_ac)
  hence "p dvd (c * a - 1)"
    by (simp add: cong_iff_dvd_diff)
  hence n: "c * a = n * p + 1"
    unfolding n_def by auto

  have "[c * a = 1] (mod int p)"
    using a by (simp add: mult_ac)
  hence "[(1 ^ k - c ^ k) * N k = ((c * a) ^ k - c ^ k) * N k] (mod p)"
    by (intro cong_mult cong_diff cong_refl cong_pow) (auto simp: cong_sym_eq)
  also have "((c * a) ^ k - c ^ k) * N k = c ^ k * (a ^ k - 1) * N k"
    by (simp add: algebra_simps)
  also have "[(a^k-1) * N k = k * a^(k-1) * D k * (∑m=1..<p. m^(k-1)
* (m * a div p))] (mod p)"
    by (rule voronoi_congruence) (use k p ‹coprime a p› in auto)
  hence "[c ^ k * ((a^k-1) * N k) =
         c ^ k * (k * a^(k-1) * D k * (∑m=1..<p. m^(k-1) * (m * a div
p)))] (mod p)"
    by (rule cong_mult[OF cong_refl])
  hence "[c ^ k * (a^k-1) * N k =
         k * (c ^ k * a^(k-1)) * D k * (∑m=1..<p. m^(k-1) * (m * a div
p))] (mod p)"
    by (simp add: mult_ac)
  also have "c ^ k * a ^ (k - 1) = c * (c * a) ^ (k - 1)"
    using k by (cases k) (auto simp: algebra_simps)
  also have "[k * (c * (c * a) ^ (k-1)) * D k * (∑m=1..<p. m^(k-1) *
(m * a div p)) =
             k * (c * 1 ^ (k-1)) * D k * (∑m=1..<p. m^(k-1) * (m * a
div p))] (mod p)"
    by (intro cong_mult cong_pow cong_refl cong_modular_inverse1) (use
a in ‹simp_all add: mult_ac›)
  also have "k * (c * 1 ^ (k-1)) * D k * (∑m=1..<p. m^(k-1) * (m * a
div p)) =
           D k * (k * (∑m=1..<p. m^(k-1) * c * (m * a div p)))"
    by (simp add: algebra_simps sum_distrib_left sum_distrib_right)
  finally have "[of_int ((1 - c ^ k) * N k) =
                of_int (D k * k * (∑m=1..<p. m^(k-1) * c * (m * a div
p)))] (qmod p)"
    using p by (intro cong_imp_qcong) (auto simp: mult_ac prime_gt_1_nat)
  hence "[of_int (1 - c ^ k) * of_int (N k) / of_int (D k) =
                of_int k * (∑m=1..<p. of_nat m^(k-1) * (of_int c * of_int
(m * a div p)))] (qmod p)"
    using p by (subst qcong_divide_of_int_left_iff) (auto simp: mult_ac)
  also have "(∑m=1..<p. of_nat m^(k-1) * (of_int c * of_int (m * a div
p))) =
```

```
            (∑m=1..<p. of_nat m^(k-1) * h m + of_int n * of_nat m^k
- of_int (c - 1) / 2 * of_nat m^(k-1))"
  proof (intro sum.cong, goal_cases)
    case (2 m)
    have *: "(m * a) div p * p = m * a - ((m * a) mod p)"
      by (metis minus_mod_eq_div_mult)
    have "c * ((m * a) div p * p) = c * a * m - c * ((m * a) mod p)"
      by (subst *) (simp_all add: algebra_simps)
    also have "c * a = n * p + 1"
      by fact
    finally have **: "c * ((m * a) div p) * p = n * m * p + m - c * ((m
* a) mod p)"
      by (simp add: algebra_simps)
    have "of_int (c * ((m * a) div p)) = of_int (n * m * p + m - c * ((m
* a) mod p)) / (of_int p :: rat)"
      by (subst ** [symmetric]) auto
    also have "... = h m + of_int (n * m) - of_int (c - 1) / 2"
      by (simp add: h_def field_simps)
    finally have ***: "of_int c * of_int (m * a div p) = h m + of_int (n
* m) - of_int (c - 1) / 2"
      by simp
    have "of_nat m^(k-1) * (of_int c * of_int (m * a div p)) =
          of_nat m^(k-1) * h m + of_int n * of_nat (m * m^(k-1)) - of_int
(c - 1) / 2 * of_nat m^(k-1)"
      by (subst ***) (auto simp: algebra_simps)
    also have "m * m ^ (k - 1) = m ^ k"
      using k by (cases k) auto
    finally show ?case
      by simp
  qed auto
  also have "... = (∑m=1..<p. of_nat m^(k-1) * h m) + of_int n * of_int
(int (∑m=1..<p. m ^ k)) -
                   of_int (c-1) / 2 * of_int (int (∑m=1..<p. m^(k-1)))"
    by (simp add: sum.distrib sum_subtractf sum_distrib_left sum_distrib_right)
  also have "(∑m=1..<p. m ^ k) = S k p"
    using k unfolding S_def by (intro sum.mono_neutral_left) auto
  also have "(∑m=1..<p. m ^ (k - 1)) = S (k - 1) p"
    using k unfolding S_def by (intro sum.mono_neutral_left) auto
  also have "[of_int k * ((∑m=1..<p. of_nat m^(k-1) * h m) + of_int n
* of_int (int (S k p)) -
                   of_int (c-1) / 2 * of_int (int (S (k-1) p))) =
              of_int k * ((∑m=1..<p. of_nat m^(k-1) * h m) + of_int n
* of_int (int 0) -
                   of_int (c-1) / 2 * of_int (int 0))] (qmod p)"
  proof (intro qcong_mult qcong_add qcong_diff cong_imp_qcong qcong_sum
qcong_pow qcong_refl cong_int cong_refl)
    have "quotient_of (rat_of_int (c - 1) / 2) = (if even c then c - 1
else c div 2, if even c then 2 else 1)"
      using c by (intro quotient_of_eqI) (auto elim!: oddE)
```

```
    thus "coprime (snd (quotient_of (rat_of_int (c - 1) / 2))) (int p)"
      using ‹odd p› by auto
  next
    fix m assume m: "m ∈ {1..<p}"
    have "[int m - c * (a * int m mod int p) = int m - c * (a * int m)]
(mod p)"
      by (intro cong_diff cong_mult cong_refl) auto
    also have "c * (a * int m) = (c * a) * int m"
      by (simp add: mult_ac)
    also have "[int m - (c * a) * int m = int m - 1 * int m] (mod p)"
      using a by (intro cong_diff cong_mult cong_refl) (auto simp: mult_ac)
    finally have "p dvd int m - c * (a * int m mod int p)"
      by (simp add: cong_0_iff)
    then obtain d where d: "int m - c * (a * int m mod int p) = int p
* d"
      by (elim dvdE)

    have "h m = of_int (int m - c * (a * int m mod int p)) / of_nat p
+ of_int (c - 1) / 2"
      by (auto simp: h_def)
    also note d
    also have "rat_of_int (int p * d) / of_nat p = of_int d"
      by simp
    finally have h_eq: "h m = Rat.Fract (2 * d + c - 1) 2"
      by (auto simp: Rat.Fract_of_int_quotient)

    have "snd (quotient_of (h m)) dvd 2"
      unfolding h_eq quotient_of_Fract using dvd_rat_normalize(2) by simp
    with ‹prime p› ‹odd p› have "¬p dvd snd (quotient_of (h m))"
      by (metis dvd_trans int_dvd_int_iff of_nat_numeral primes_dvd_imp_eq
two_is_prime_nat)
    show "coprime (snd (quotient_of (h m))) (int p)"
      by (subst coprime_commute, rule prime_imp_coprime)
         (use p ‹¬p dvd snd (quotient_of (h m))› in auto)
  next
    have *: "[S k p = 0] (mod p)" if "¬(p - 1) dvd k" for k
    proof -
      have "k > 0"
        using that by (intro Nat.gr0I) auto
      hence "S k p = (∑m=1..<p. m ^ k)"
        unfolding S_def by (intro sum.mono_neutral_right) auto
      thus ?thesis
        using sum_of_powers_mod_prime'[of p k] that p by simp
    qed
    show "[S k p = 0] (mod p)"
      using k by (intro *) (auto dest!: dvd_imp_le)
    show "[S (k - 1) p = 0] (mod p)"
      using k by (intro *) (auto dest!: dvd_imp_le)
  qed (use p in ‹auto simp: prime_gt_1_nat›)
```

```
  finally have "[rat_of_int (1 - c ^ k) * rat_of_int (N k) / rat_of_int
(int (D k)) =
                rat_of_nat k * (∑m=1..<p. rat_of_nat m^(k-1) * h m)]
(qmod p)"
    by simp
  moreover have "¬p dvd (1 - c ^ k)"
    using c by (auto simp: cong_iff_dvd_diff dvd_diff_commute)
  hence "coprime (1 - c ^ k) (int p)"
    using prime_imp_coprime[of p "1 - c ^ k"] p by (auto simp: coprime_commute)
  ultimately have "[rat_of_int (N k) / rat_of_int (int (D k)) =
                   rat_of_nat k * (∑m=1..<p. rat_of_nat m^(k-1) * h
m) / rat_of_int (1 - c ^ k)] (qmod p)"
    using c by (subst qcong_divide_of_int_right_iff) (auto simp: mult_ac)
  thus ?thesis
    by (simp add: bernoulli_rat_def mult_ac N_def D_def)
qed

corollary voronoi_congruence_harvey':
  fixes k p :: nat and g :: nat and h :: "nat ⇒ rat" and a :: int
  assumes k: "even k" "k ∈ {2..p-3}" and p: "prime p" "p ≥ 5"
  assumes g: "residue_primroot p g" "g ∈ {0<..<p}"
  assumes a: "[a * int g = 1] (mod int p)"
  defines "h' ≡ (λm. rat_of_int (int m mod p - g * ((a * m) mod p)) /
of_int p + of_int (int g - 1) / 2)"
  shows   "[ℬ k = 2 * of_nat k / of_int (1 - int g ^ k) *
                   (∑i=1..(p-1) div 2. of_nat (g^(i*(k-1))) * h' (g^i))]
(qmod int p)"
proof -
  define h :: "nat ⇒ rat" where "h = (λm. of_int (m - g * ((a * m) mod
p)) / of_int p + of_int (int g - 1) / 2)"
  define m1 where "m1 = g ^ ((p - 1) div 2)"
  have cong_m1: "[int m1 = - 1] (mod int p)"
    unfolding m1_def using residue_primroot_power_cong_neg1[of p g] g
p by simp
  from p have [simp]: "p ≠ 0"
    by auto
  from p have "odd p"
    by (intro prime_odd_nat) auto
  from g have "coprime g p"
    by (auto simp: residue_primroot_def coprime_commute)
  have "coprime a p"
    using a coprime_iff_invertible_int by auto
  have "¬(p - 1) dvd k"
    using k p by (auto dest!: dvd_imp_le)

  have neq1: "[int g ^ k ≠ 1] (mod p)"
  proof -
    have "[int g ^ k = 1] (mod p) ⟷ [g ^ k = 1] (mod p)"
      using cong_int_iff by force
```

```
    also have "[g ^ k = 1] (mod p) ⟷ ord p g dvd k"
      by (rule ord_divides)
    also have "ord p g = totient p"
      using g(1) unfolding residue_primroot_def by blast
    also have "... = p - 1"
      using p by (simp add: totient_prime)
    finally show ?thesis
      using ‹¬(p - 1) dvd k› by simp
  qed

  have coprime_h': "coprime (snd (quotient_of (h' (g ^ i)))) (int p)"
for i
  proof -
    define b where "b = (int (g ^ i) mod p - int g * (a * int (g ^ i)
mod int p)) div p"
    have "[int g * (a * int (g ^ i) mod int p) = int g * (a * int (g ^
i))] (mod p)"
      by (intro cong_mult cong_refl) auto
    also have "int g * (a * int (g ^ i)) = int g * a * int (g ^ i)"
      by (simp add: mult_ac)
    also have "[... = 1 * int (g ^ i) mod p] (mod p)"
      by (intro cong_mult cong_mod_right) (use a in ‹auto simp: mult_ac›)
    finally have "p dvd (int (g ^ i) mod p - int g * (a * int (g ^ i) mod
int p))"
      by (simp add: cong_iff_dvd_diff dvd_diff_commute)
    hence b: "(int (g ^ i) mod p - int g * (a * int (g ^ i) mod int p))
= p * b"
      unfolding b_def by simp

    have "h' (g ^ i) = Rat.Fract (2 * b + int g - 1) 2"
      unfolding h'_def b by (simp add: field_simps Rat.Fract_of_int_quotient)
    also have "snd (quotient_of ...) dvd 2"
      unfolding quotient_of_Fract by (simp add: dvd_rat_normalize(2))
    finally have "¬p dvd snd (quotient_of (h' (g ^ i)))" using ‹odd p›
      by (metis assms(3) dvd_trans int_dvd_int_iff of_nat_numeral primes_dvd_imp_eq
two_is_prime_nat)
    with p show ?thesis
      using prime_imp_coprime coprime_commute prime_nat_int_transfer by
metis
  qed

  have bij: "bij_betw (λi. g ^ i mod p) {1..p-1} {0<..<p}"
    using residue_primroot_is_generator'[of p g] g p
    by (simp add: totient_prime totatives_prime prime_gt_Suc_0_nat)

  have "[ℬ k = of_nat k / of_int (1 - int g ^ k) * (∑m=1..<p. of_nat
m ^ (k-1) * h m)] (qmod p)"
    unfolding h_def by (rule voronoi_congruence_harvey) (use k p g neq1
a in simp_all)
```

```
  also have "(∑m∈{1..<p}. of_nat m ^ (k-1) * h m) = (∑m∈{0<..<p}. of_nat
m ^ (k-1) * h (m mod p))"
    by (intro sum.cong) auto
  also have "(∑m∈{0<..<p}. of_nat m ^ (k-1) * h (m mod p)) =
             (∑i=1..p-1. of_nat (g^i mod p) ^ (k-1) * h (g^i mod p mod
p))"
    using bij by (intro sum.reindex_bij_betw [symmetric])
  also have "... = (∑i=1..p-1. of_nat (g^i mod p) ^ (k-1) * h (g^i mod
p))"
    by simp
  also have "... = (∑i=1..p-1. of_nat (g^i mod p) ^ (k-1) * h' (g^i))"
  proof (intro sum.cong, goal_cases)
    case (2 i)
    have "h' (g ^ i) - h (g ^ i mod p mod p) =
           of_nat g * of_int (a * int (g ^ i mod p) mod int p -  a *
int (g ^ i) mod int p) / of_int p"
      unfolding h_def h'_def by (simp add: field_simps flip: of_nat_power
of_nat_mod)
    also have "[a * int (g ^ i mod p) = a * int (g ^ i)] (mod p)"
      by (intro cong_mult) (auto simp flip: of_nat_power of_nat_mod intro!:
cong_int)
    hence "a * int (g ^ i mod p) mod int p = a * int (g ^ i) mod int p"
      by (auto simp: Cong.cong_def)
    finally show ?case
      by simp
  qed auto
  also have "{1..p-1} = {1..(p-1) div 2} ∪ {(p-1) div 2<..p-1}"
    by auto
  also have "(∑i∈.... of_nat (g^i mod p) ^ (k-1) * h' (g^i)) =
             (∑i=1..(p-1) div 2. of_nat (g^i mod p) ^ (k-1) * h' (g^i))
+
             (∑i∈{(p-1) div 2<..p-1}. of_nat (g^i mod p) ^ (k-1) * h'
(g^i))"
    by (subst sum.union_disjoint) auto
  also have "(∑i∈{(p-1) div 2<..p-1}. of_nat (g^i mod p) ^ (k-1) * h'
(g^i)) =
             (∑i=1..(p-1) div 2. of_nat (g^(i + (p-1) div 2) mod p) ^
(k-1) * h' (g^(i + (p-1) div 2)))"
    using ‹odd p›
    by (intro sum.reindex_bij_witness[of _ "λi. i + (p-1) div 2" "λi.
i - (p-1) div 2"])
       (auto elim!: oddE)
  also have "[rat_of_nat k / rat_of_int (1 - int g ^ k) *
                ((∑i=1..(p-1) div 2. of_nat (g^i mod p) ^ (k-1) * h'
(g^i)) + ...) =
              rat_of_nat k / rat_of_int (1 - int g ^ k) * (
                (∑i=1..(p-1) div 2. of_nat (g^(i*(k-1))) * h' (g^i))
+
                (∑i=1..(p-1) div 2. (-of_int (g^(i*(k-1)))) * (-h' (g^i))))]
```

```
(qmod p)"
  proof (intro qcong_divide_of_int qcong_add qcong_sum qcong_refl qcong_mult)
    fix i :: nat
    have "rat_of_int (int (g ^ (i + (p - 1) div 2) mod p) ^ (k - 1)) =
          of_int ((int g ^ i * m1 mod p) ^ (k - 1))"
      by (simp add: power_add m1_def flip: of_nat_power of_nat_mult of_nat_mod)
    also have "[... = of_int (- (int g ^ (i * (k - 1))))] (qmod p)"
    proof (intro cong_imp_qcong)
      have "[(int g ^ i * int m1 mod int p) ^ (k - 1) = (int g ^ i * (-1))
^ (k - 1)] (mod p)"
        by (intro cong_mult cong_pow cong_mod_left cong_refl cong_m1)
      also have "(int g ^ i * (-1)) ^ (k - 1) = -(int g ^ (i * (k - 1)))"
        using k by (simp add: power_minus' flip: power_mult)
      finally show "[(int g ^ i * m1 mod p) ^ (k - 1) = -(int g ^ (i *
(k - 1)))] (mod p)" .
    qed (use p in auto)
    finally show "[rat_of_nat (g ^ (i + (p - 1) div 2) mod p) ^ (k - 1)
=
                  -(rat_of_int (int (g ^ (i * (k - 1)))))] (qmod int
p)"
      by simp
  next
    fix i :: nat
    have "[int g ^ i * int m1 = int g ^ i * (-1)] (mod p)"
      by (intro cong_mult cong_m1 cong_refl)
    hence "(int g ^ i * int m1 mod p) = -(int g ^ i) mod p"
      by (simp add: Cong.cong_def)
    also have nz: "int g ^ i mod p ≠ 0" using ‹coprime g p›
      by (metis assms(3) coprime_absorb_right coprime_int_iff coprime_power_left_iff

          dvd_triv_left not_prime_unit prime_nat_int_transfer zmod_eq_0D)
    hence "-(int g ^ i) mod p = int p - (int g ^ i mod p)"
      by (subst zmod_zminus1_eq_if) auto
    finally have eq1: "int g ^ i * int m1 mod int p = int p - int g ^ i
mod int p" .

    have "[a * (int g ^ i * int m1) = a * (int g ^ i * (-1))] (mod p)"
      by (intro cong_mult cong_m1 cong_refl)
    hence "(a * (int g ^ i * int m1)) mod p = -(a * int g ^ i) mod p"
      by (simp add: Cong.cong_def)
    also have "(a * int g ^ i) mod p ≠ 0" using ‹coprime a p›
      by (metis nz coprime_commute mod_mod_trivial mult_mod_cancel_left
mult_not_zero)
    hence "-(a * int g ^ i) mod p = int p - (a * int g ^ i mod p)"
      by (subst zmod_zminus1_eq_if) auto
    finally have eq2: "a * (int g ^ i * int m1) mod int p = int p - a *
int g ^ i mod int p"
      by (simp add: mult_ac)
```

```
    have "h' (g ^ (i + (p - 1) div 2)) = h' (g ^ i * m1)"
      by (simp add: power_add m1_def)
    also have "... = of_int (int p - (int g ^ i mod int p) - int g * (int
p - a * int g ^ i mod int p)) /
                     rat_of_nat p + (rat_of_nat g - 1) / 2"
      by (simp add: h'_def eq1 eq2)
    also have "... = -h' (g ^ i)"
      by (simp add: h'_def field_simps)
    finally have "h' (g ^ (i + (p - 1) div 2)) = - h' (g ^ i)"
      by simp
    moreover have "coprime (snd (quotient_of (- h' (g ^ i)))) (int p)"
      by (metis calculation coprime_h')
    ultimately show "[h' (g ^ (i + (p - 1) div 2)) = -h' (g ^ i)] (qmod
int p)"
      by (auto intro!: qcong_refl)
  next
    fix i :: nat
    show "[rat_of_nat (g ^ i mod p) ^ (k - 1) = rat_of_nat (g ^ (i *
(k - 1)))] (qmod int p)"
      unfolding of_nat_power [symmetric]
    proof (rule cong_imp_qcong_of_nat)
      have "[(g ^ i mod p) ^ (k - 1) = (g ^ i) ^ (k - 1)] (mod p)"
        by (intro cong_pow) auto
      also have "(g ^ i) ^ (k - 1) = g ^ (i * (k - 1))"
        by (simp add: power_mult)
      finally show "[(g ^ i mod p) ^ (k - 1) = g ^ (i * (k - 1))] (mod
p)" .
    qed (use ‹p ≥ 5› in auto)
  next
    fix i :: nat
    show "coprime (snd (quotient_of (h' (g ^ i)))) (int p)"
      using coprime_h'[of i] by auto
  next
    show "coprime (1 - int g ^ k) (int p)"
      by (meson assms(3) cong_iff_dvd_diff dvd_diff_commute neq1 residues_prime.intro
            residues_prime.p_coprime_right_int)
    thus "1 - int g ^ k ≠ 0"
      using ‹p ≥ 5› by fastforce
    thus "1 - int g ^ k ≠ 0" .
  qed (use ‹p ≥ 5› in auto)
  finally show ?thesis
    by (simp add: mult_ac)
qed

unbundle no_bernoulli_notation

end

end
```

4 Kummer's Congruence

```
theory Kummer_Congruence
  imports Voronoi_Congruence
begin

unbundle bernoulli_notation

context
  fixes S :: "nat ⇒ nat ⇒ nat" and D :: "nat ⇒ nat" and N :: "nat ⇒
int"
  defines "S ≡ (λk n. ∑r<n. r ^ k)"
  defines "N ≡ bernoulli_num" and "D ≡ bernoulli_denom"
begin
```

Auxiliary lemma for Proposition 9.5.23: if k is even and $(p-1) \nmid k$, then $\nu_p(N_k) \geq \nu_p(k)$.

```
lemma multiplicity_prime_bernoulli_num_ge:
  fixes p k :: nat
  assumes p: "prime p" "¬(p - 1) dvd k" and k: "even k"
  shows   "multiplicity p (N k) ≥ multiplicity p k"
proof (cases "k ≥ 2")
  case True
  define e where "e = multiplicity p k"
  define k' where "k' = k div p ^ e"
  obtain a where a: "residue_primroot p a"
    using ‹prime p› prime_primitive_root_exists[of p] prime_gt_1_nat[of
p] by auto

  have "[(int a^k-1) * N k = k * int a ^ (k-1) * D k * (∑m=1..<p^e.
m^(k-1) * (m * int a div p^e))] (mod p^e)"
    unfolding D_def N_def
  proof (rule voronoi_congruence)
    show "coprime (int a) (int (p ^ e))"
      using a p unfolding residue_primroot_def by (auto simp: coprime_commute)
  qed (use p True k prime_gt_0_nat[of p] in auto)
  also have "p ^ e dvd k"
    unfolding e_def by (simp add: multiplicity_dvd)
  hence "[k * int a ^ (k - 1) * D k * (∑m=1..<p^e. m^(k-1) * (m * int
a div p^e)) = 0] (mod (p ^ e))"
    by (subst cong_0_iff) auto
  finally have "[(int a ^ k - 1) * N k = 0] (mod int (p ^ e))" .
  moreover have "coprime (int a ^ k - 1) (int p)"
  proof -
    have "[a ^ k = 1] (mod p) ⟷ (ord p a dvd k)"
      by (rule ord_divides)
    also have "ord p a = p - 1"
      using a p by (simp add: residue_primroot_def totient_prime)
    finally have "[a ^ k ≠ 1] (mod p)"
```

```
    using p by simp
  hence "[int (a ^ k) ≠ int 1] (mod int p)"
    using cong_int_iff by blast
  hence "¬int p dvd int a ^ k - 1"
    unfolding cong_iff_dvd_diff by auto
  thus "coprime (int a ^ k - 1) (int p)"
    using ‹prime p› by (simp add: coprime_commute prime_imp_coprime)
qed
ultimately have "[N k = 0] (mod int (p ^ e))"
  by (metis cong_mult_rcancel coprime_power_right_iff mult.commute mult_zero_left
of_nat_power)
hence "p ^ e dvd N k"
  by (simp add: cong_iff_dvd_diff)
thus "multiplicity p (N k) ≥ e"
  using p k by (intro multiplicity_geI) (auto simp: N_def bernoulli_num_eq_0_iff)
next
case False
with assms have "k = 0"
  by auto
thus ?thesis by auto
qed
```

Proposition 9.5.23: if k is even and $(p-1) \nmid k$, then B_k/k is p-integral.

```
lemma bernoulli_k_over_k_is_p_integral:
fixes p k :: nat
assumes p: "prime p" "¬(p - 1) dvd k" and k: "k ≠ 1"
shows   "qmultiplicity p (ℬ k / of_nat k) ≥ 0"
proof -
consider "odd k" | "k = 0" | "even k" "k ≥ 2"
  by fastforce
hence "qmultiplicity p (of_int (N k) / of_int (D k * k)) ≥ 0"
proof cases
  assume k: "odd k"
  hence "bernoulli_num k = 0" using ‹k ≠ 1›
    by (subst bernoulli_num_eq_0_iff) auto
  thus ?thesis by (auto simp: N_def)
next
  assume k: "even k" "k ≥ 2"
  from k have [simp]: "N k ≠ 0" "D k > 0"
    by (auto simp: N_def D_def bernoulli_num_eq_0_iff intro!: Nat.gr0I)
  have "qmultiplicity p (of_int (N k) / of_int (k * D k)) =
          int (multiplicity p (N k)) - int (multiplicity p (k * D k))"
using k p
    by (subst qmultiplicity_divide_of_int) (auto simp: multiplicity_int
simp del: of_nat_mult)
  also have "multiplicity p (k * D k) = multiplicity p k + multiplicity
p (D k)"
    using p k by (subst prime_elem_multiplicity_mult_distrib) auto
  also have "multiplicity p (D k) = 0"
```

```
      using p k by (intro not_dvd_imp_multiplicity_0) (auto simp: prime_dvd_bernoulli_denom
D_def)
    also have "int (multiplicity (int p) (N k)) - int (multiplicity p
k + 0) ≥ 0"
      using multiplicity_prime_bernoulli_num_ge[of p k] k p by auto
    finally show ?thesis by (simp add: mult_ac)
  qed auto
  also have "of_int (N k) / of_int (D k * k) = bernoulli_rat k / of_nat
k"
    by (simp add: bernoulli_rat_def N_def D_def)
  finally show ?thesis .
qed


lemma kummer_congruence_aux:
  fixes k p a :: nat
  assumes k: "even k" "k ≥ 2" and p: "¬(p - 1) dvd k" "prime p"
  assumes a: "¬p dvd a"
  assumes s: "s ≥ multiplicity p k"
  shows "[of_int ((1 - int p^(k-1)) * (int a^k - 1)) * ℬ k / of_nat k
=
            of_int (int a^(k-1) *
              (∑m∈{m∈{1..<p^(s+e)}. ¬p dvd m}. m^(k-1) * (int m * a
div p ^ (e + s))))] (qmod p^e)"
proof (cases "e > 0")
  case e: True
  from p have "p > 0"
    by (auto intro!: Nat.gr0I)
  have [simp]: "D k > 0"
    by (auto simp: D_def bernoulli_denom_pos)
  define s' where "s' = multiplicity p k"

  define k1 where "k1 = k div p ^ s'"
  have "p ^ s' dvd k"
    by (simp add: multiplicity_dvd' s'_def)
  hence k_eq: "k = k1 * p ^ s'"
    by (simp add: k1_def)
  have "coprime k1 p" unfolding k1_def s'_def using p
    by (metis coprime_commute dvd_0_right multiplicity_decompose not_prime_unit
prime_imp_coprime)
  have "k1 > 0"
    using k by (intro Nat.gr0I) (auto simp: k_eq)

  define N' where "N' = N k div p^s'"
  have "multiplicity p (N k) ≥ s'"
    using s s'_def p k multiplicity_prime_bernoulli_num_ge[of p k] by
linarith
  hence "p^s' dvd N k"
    by (simp add: multiplicity_dvd')
```

```
  hence N_eq: "N k = N' * p^s'"
    by (simp add: N'_def)
  have "coprime a p"
    using a p coprime_commute prime_imp_coprime by blast

  have "¬p dvd D k"
    unfolding D_def using k p by (subst prime_dvd_bernoulli_denom_iff)
auto
  hence "coprime (D k) p"
    using p coprime_commute prime_imp_coprime by blast

  have cong1: "[(int a ^ k - 1) * N' =
                 k1 * a ^ (k-1) * D k * (∑m=1..<p^(s+e). (m^(k-1)) *
(int m * a div p^(s+e)))]
               (mod int p ^ e)" for e
  proof -
    have "[(int a ^ k - 1) * N k =
           k * a ^ (k - 1) * D k * (∑m = 1..<p^(s+e). (m^(k-1)) * (int
m * a div p^(s+e)))] (mod p^(s+e))"
      using voronoi_congruence[of k "p^(s+e)" a] k ‹p > 0› ‹coprime a
p› unfolding D_def N_def
      by (simp_all add: residue_primroot_def coprime_commute)
    also have "N k = N' * p ^ s'"
      by (simp add: N_eq)
    also have "k * a ^ (k - 1) = k1 * p ^ s' * a ^ (k - 1)"
      by (simp add: k_eq)
    finally have "[int p ^ s' * ((int a ^ k - 1) * N') =
                   int p ^ s' * (k1 * a ^ (k-1) * D k * (∑m=1..<p^(s+e).
(m^(k-1)) * (int m * a div p^(s+e))))]
                  (mod (int p ^ s * int p ^ e))"
      by (simp add: mult_ac power_add)
    hence "[int p ^ s' * ((int a ^ k - 1) * N') =
                   int p ^ s' * (k1 * a ^ (k-1) * D k * (∑m=1..<p^(s+e).
(m^(k-1)) * (int m * a div p^(s+e))))]
                  (mod (int p ^ s' * int p ^ e))"
      by (rule cong_modulus_mono) (use s in ‹auto simp: le_imp_power_dvd
s'_def›)
    thus ?thesis
      by (rule cong_mult_cancel) (use p in auto)
  qed

  have cong2:
    "[int p^(k-1) * ((int a ^ k - 1) * N') =
      int p^(k-1) * (k1 * a ^ (k-1) * D k * (∑m=1..<p^(s+(e-1)). (m^(k-1))
* (int m * a div p^(s+(e-1)))))]
       (mod int p ^ e)"
    by (rule power_mult_cong[OF cong1]) (use k in auto)

  define M1 where "M1 = {m∈{1..<p^(s+e)}. ¬p dvd m}"
```

```
  define M2 where "M2 = {m∈{1..<p^(s+e)}. p dvd m}"
  define M2' where "M2' = {1..<p^(e+s-1)}"

  have "(∑m=1..<p^(s+e). (m^(k-1)) * (int m * a div p^(s+e))) =
        (∑m∈M1∪M2. (m^(k-1)) * (int m * a div p^(s+e)))"
    by (intro sum.cong) (auto simp: M1_def M2_def)
  also have "... = (∑m∈M1. (m^(k-1)) * (int m * a div p^(s+e))) +
                   (∑m∈M2. (m^(k-1)) * (int m * a div p^(s+e)))"
    by (intro sum.union_disjoint) (auto simp: M1_def M2_def)
  also have "(∑m∈M2. (m^(k-1)) * (int m * a div p^(s+e))) =
             (∑m∈M2'. ((p*m)^(k-1)) * (int m * a div p^(s+e-1)))"
    using e p prime_gt_1_nat[of p]
    by (intro sum.reindex_bij_witness[of _ "λm. p*m" "λm. m div p"]; cases
e)
       (auto simp: M2_def M2'_def add_ac elim!: dvdE)
  also have "... = p^(k-1) * (∑m∈M2'. (m^(k-1)) * (int m * a div p^(s+e-1)))"
    by (simp add: sum_distrib_left sum_distrib_right power_mult_distrib
mult_ac)
  finally have sum_eq:
     "(∑m=1..<p^(s+e). (m^(k-1)) * (int m * a div p^(s+e))) =
       (∑m∈M1. (m^(k-1)) * (int m * a div p^(s+e))) +
       p^(k-1) * (∑m∈M2'. (m^(k-1)) * (int m * a div p^(s+e-1)))" .


  have "[(1 - int p^(k-1)) * (int a^k - 1) * N' =
         k1 * a ^ (k-1) * D k * (
            (∑m=1..<p^(s+e). (m^(k-1)) * (int m * a div p^(s+e))) -
            int p^(k-1) * (∑m=1..<p^(s+(e-1)). (m^(k-1)) * (int m * a
div p^(s+(e-1)))))]
    (mod p^e)"
    using cong_diff[OF cong1[of e] cong2] by (simp add: algebra_simps)
  also have "(∑m=1..<p^(s+e). (m^(k-1)) * (int m * a div p^(s+e))) -

              int p^(k-1) * (∑m=1..<p^(s+(e-1)). (m^(k-1)) * (int m
* a div p^(s+(e-1)))) =
             (∑m∈M1. int m ^ (k - Suc 0) * (int m * int a div int p
^ (e + s)))"
    using e by (subst sum_eq) (simp_all add: M2'_def add_ac)
  finally have cong:
     "[(1 - int p^(k-1)) * (int a^k - 1) * N' =
         k1 * a^(k-1) * D k * (∑m∈M1. m^(k-1) * (int m * a div p ^ (e
+ s)))] (mod p^e)"
    by simp

  have "int p ^ e > 1"
    using p e by (metis of_nat_1 of_nat_less_iff one_less_power prime_gt_1_nat)
  hence "[of_int ((1 - int p^(k-1)) * (int a^k - 1)) * of_int N' =
           of_nat (k1 * D k) * of_int (int a^(k-1) * (∑m∈M1. m^(k-1)
* (int m * a div p ^ (e + s))))] (qmod p^e)"
```

```
    using cong_imp_qcong[OF cong] by (simp add: mult_ac)
  hence "[of_int ((1 - int p^(k-1)) * (int a^k - 1)) * of_int N' / of_nat
(k1 * D k) =
           of_int (int a^(k-1) * (∑m∈M1. m^(k-1) * (int m * a div p
^ (e + s))))] (qmod p^e)"
    using ‹coprime k1 p› ‹coprime (D k) p› ‹k1 > 0› p
    by (subst qcong_divide_of_nat_left_iff) (auto simp: mult_ac prime_gt_0_nat)
  hence "[of_int ((1 - int p^(k-1)) * (int a^k - 1)) * (of_int N' / of_nat
(k1 * D k)) =
           of_int (int a^(k-1) * (∑m∈M1. m^(k-1) * (int m * a div p
^ (e + s))))] (qmod p^e)"
    by simp
  also have "of_int N' / of_nat (k1 * D k) = of_int (N k) / (of_nat (k
* D k) :: rat)"
    unfolding N_eq unfolding k_eq using p by simp
  finally show ?thesis
    by (simp add: M1_def bernoulli_rat_def N_def D_def mult_ac)
qed auto

theorem kummer_congruence:
  fixes k k' p :: nat
  assumes k: "even k" "k ≥ 2" and k': "even k'" "k' ≥ 2" and p: "¬(p
- 1) dvd k" "prime p"
  assumes cong: "[k = k'] (mod totient (p ^ e))"
  shows   "[(of_nat p^(k-1)-1) * ℬ k / of_nat k =
           (of_nat p^(k'-1)-1) * ℬ k' / of_nat k'] (qmod (p^e))"
proof (cases "e > 0")
  case e: True
  from p have [simp]: "p ≠ 0"
    by auto
  obtain a where a: "residue_primroot p a"
    using ‹prime p› prime_primitive_root_exists[of p] prime_gt_1_nat[of
p] by auto
  define s where "s = max (multiplicity p k) (multiplicity p k')"
  have "¬p dvd a"
    using a p unfolding residue_primroot_def
    by (metis coprime_absorb_right coprime_commute not_prime_unit)
  have "coprime a p"
    using a by (auto simp: residue_primroot_def coprime_commute)

  have cong': "[k = k'] (mod (p - 1) * (p ^ (e - 1)))"
    using p cong e by (simp add: totient_prime_power mult_ac)
  hence "[k = k'] (mod (p - 1))"
    using cong_modulus_mult_nat by blast
  with ‹¬(p - 1) dvd k› have "¬(p - 1) dvd k'"
    using cong_dvd_iff by blast
  have "int p ^ e > 1"
    using p e by (metis of_nat_1 of_nat_less_iff one_less_power prime_gt_1_nat)
```

```
  define M1 where "M1 = {m∈{1..<p^(s+e)}. ¬p dvd m}"
  have M1: "coprime m p" if "m ∈ M1" for m
    using prime_imp_coprime[of p m] that p by (auto simp: coprime_commute
M1_def)

  have coprime: "coprime (snd (quotient_of (ℬ k' / of_nat k'))) (int
p)"
    by (rule qmultiplicity_prime_nonneg_imp_coprime_denom [OF bernoulli_k_over_k_is_p_integ
       (use k' p ‹¬(p-1) dvd k'› in auto)

  have "[of_int ((1 - int p^(k-1)) * (int a^k - 1)) * ℬ k / of_nat k
=
         of_int (int a^(k-1) * (∑m∈M1. int m^(k-1) * (int m * a div
p ^ (e + s))))] (qmod p^e)"
    unfolding M1_def using p ‹¬p dvd a› k
    by (intro kummer_congruence_aux) (auto simp: s_def)
  also have "of_int (int a^(k-1) * (∑m∈M1. int m^(k-1) * (int m * a
div p ^ (e + s)))) =
             of_int (int (a^(k-1)) * (∑m∈M1. int (m^(k-1)) * (int m
* a div p ^ (e + s))))"
    by simp
  also have "[... = of_int (int (a^(k'-1)) * (∑m∈M1. int (m^(k'-1)) *
(int m * a div p ^ (e + s))))] (qmod p^e)"
    using ‹int p ^ e > 1› ‹coprime a p› k k' cong p M1
    by (intro cong_imp_qcong cong_mult cong_sum cong_int cong_refl cong_pow_totient
cong_diff_nat)
       (auto simp: M1_def)
  also have "of_int (int (a^(k'-1)) * (∑m∈M1. int (m^(k'-1)) * (int
m * a div p ^ (e + s)))) =
             of_int (int a^(k'-1) * (∑m∈M1. int m^(k'-1) * (int m *
a div p ^ (e + s))))"
    by simp
  also have "[... = of_int ((1 - int p^(k'-1)) * (int a^k' - 1)) * ℬ k'
/ of_nat k'] (qmod p^e)"
    unfolding M1_def
    by (rule qcong_sym, rule kummer_congruence_aux)
       (use p ‹¬p dvd a› k' ‹¬(p - 1) dvd k'› in ‹auto simp: s_def›)
  also have "of_int ((1 - int p^(k'-1)) * (int a^k' - 1)) * ℬ k' / of_nat
k' =
             of_int ((1 - int p^(k'-1)) * (int (a^k') - 1)) * (ℬ k' /
of_nat k')"
    by simp
  also have "[of_int ((1 - int p^(k'-1)) * (int (a^k') - 1)) * (ℬ k' /
of_nat k') =
              of_int ((1 - int p^(k'-1)) * (int (a^k) - 1)) * (ℬ k' /
of_nat k')] (qmod p^e)"
    using ‹int p ^ e > 1› k k' cong p ‹coprime a p› coprime
    by (intro qcong_mult qcong_refl cong_imp_qcong cong_mult cong_refl
cong_diff cong_int
```

```
            cong_pow_totient cong_diff_nat)
      (auto simp: cong_sym_eq mult_ac prime_gt_0_nat)
  finally have "[of_int (int a ^ k - 1) * (of_int (1-int p ^ (k-1)) * ℬ
k / of_nat k) =
                of_int (int a ^ k - 1) * (of_int (1-int p ^ (k'-1)) *
ℬ k' / of_nat k')] (qmod int (p ^ e))"
    by (simp add: mult_ac)
  hence "[of_int (1-int p ^ (k-1)) * ℬ k / of_nat k =
         of_int (1-int p ^ (k'-1)) * ℬ k' / of_nat k'] (qmod int (p ^
e))"
  proof (rule qcong_mult_of_int_cancel_left)
    have "[a ^ k ≠ 1] (mod p)"
      using a p(1) assms(6) by (metis ord_divides residue_primroot_def
totient_prime)
    hence *: "¬int p dvd (int a ^ k - 1)"
      by (simp add: cong_altdef_nat')
    from * show "int a ^ k - 1 ≠ 0"
      by (intro notI) auto
    from * show "coprime (int a ^ k - 1) (int (p ^ e))"
      using p prime_imp_coprime[of p "int a ^ k - 1"] by (auto simp: coprime_commute)
  qed (use p in ‹auto simp: prime_gt_0_nat›)
  from qcong_minus[OF this] show ?thesis
    unfolding minus_divide_left minus_mult_left minus_diff_eq of_int_minus
[symmetric]
    by (simp add: mult_ac)
qed auto

corollary kummer_congruence':
  assumes kk': "even k" "even k'" "k ≥ e+1" "k' ≥ e+1"
  assumes cong: "[k = k'] (mod totient (p ^ e))"
  assumes p: "prime p" "¬(p-1) dvd k"
  shows   "[ℬ k / of_nat k = ℬ k' / of_nat k'] (qmod (p^e))"
proof (cases "e > 0")
  case e: True
  define z :: "nat ⇒ rat" where "z = (λk. ℬ k / of_nat k)"
  have "1 < int p ^ e"
    by (metis assms(6) e of_nat_1 less_imp_of_nat_less one_less_power
prime_nat_iff)
  have ge2: "k ≥ 2" "k' ≥ 2"
    using kk' by auto

  have cong': "[k = k'] (mod (p - 1) * (p ^ (e - 1)))"
    using p cong e by (simp add: totient_prime_power mult_ac)
  hence "[k = k'] (mod (p - 1))"
    using cong_modulus_mult_nat by blast
  with ‹¬(p - 1) dvd k› have "¬(p - 1) dvd k'"
    using cong_dvd_iff by blast

  have coprime: "coprime (snd (quotient_of (z l))) (int p)" if "l ∈ {k,
```

```
k'}" for l
  proof (rule qmultiplicity_prime_nonneg_imp_coprime_denom)
    have "qmultiplicity (int p) (ℬ l / of_nat l) ≥ 0"
      by (rule bernoulli_k_over_k_is_p_integral)  (use p kk' that <¬(p
- 1) dvd k'> in auto)
    thus "qmultiplicity (int p) (z l) ≥ 0"
      by (simp add: z_def)
  qed (use p in auto)

  have "[of_int (0 - 1) * z k = of_int (int p ^ (k-1) - 1) * z k] (qmod
(p ^ e))"
    using <int p ^ e > 1> kk' coprime[of k] p
    by (intro qcong_mult cong_imp_qcong qcong_refl cong_diff cong_refl)
       (auto simp: Cong.cong_def mod_eq_0_iff_dvd prime_gt_0_nat intro!:
le_imp_power_dvd)
  also have "[of_int (int p ^ (k-1) - 1) * z k = of_int (int p ^ (k'-1)
- 1) * z k'] (qmod (p ^ e))"
    unfolding z_def using kummer_congruence[of k k' p e] kk' cong p ge2
by simp
  also have "[of_int (int p ^ (k'-1) - 1) * z k' = of_int (0 - 1) * z
k'] (qmod (p ^ e))"
    using <int p ^ e > 1> kk' coprime[of k'] p
    by (intro qcong_mult cong_imp_qcong qcong_refl cong_diff cong_refl)
       (auto simp: Cong.cong_def mod_eq_0_iff_dvd prime_gt_0_nat intro!:
le_imp_power_dvd)
  finally show ?thesis
    by (simp add: z_def qcong_minus_minus_iff)
qed auto

corollary kummer_congruence'_prime:
  assumes kk': "even k" "even k'" "k > 0" "k' > 0"
  assumes cong: "[k = k'] (mod (p - 1))"
  assumes p: "prime p" "¬(p-1) dvd k"
  shows   "[ℬ k / of_nat k = ℬ k' / of_nat k'] (qmod p)"
proof -
  from kk' have "k ≥ 2" "k' ≥ 2"
    by auto
  thus ?thesis
    using kummer_congruence'[of k k' 1 p] assms by (auto simp: totient_prime)
qed

end

unbundle no_bernoulli_notation

end
```

5 Regular primes

```
theory Regular_Primes
  imports Kummer_Congruence Zeta_Function.Zeta_Function
begin


definition regular_prime :: "nat ⇒ bool" where
  "regular_prime p ⟷ prime p ∧ (p = 2 ∨ (∀k∈{2..p-3}. even k ⟶ ¬p
dvd bernoulli_num k))"

definition irregular_prime :: "nat ⇒ bool" where
  "irregular_prime p ⟷ prime p ∧ (p ≠ 2 ∧ (∃k∈{2..p-3}. even k ∧
p dvd bernoulli_num k))"

lemma irregular_primeI:
  assumes "prime p" "p ≠ 2" "p dvd bernoulli_num k" "even k" "k ∈ {2..p-3}"
  shows   "irregular_prime p"
  unfolding irregular_prime_def using assms by blast

lemma bernoulli_32: "bernoulli 32 = -7709321041217 / 510"
  by (simp add: eval_bernpoly)
```

The smallest irregular prime is 37.

```
lemma irregular_prime_37: "irregular_prime 37"
proof -
  have "(-217) * 7709321041217 + 3280240521459 * 510 = (1 :: int)"
    by simp
  hence "coprime 7709321041217 (510 :: int)"
    by (rule coprimeI_via_bezout)
  hence "bernoulli_num 32 = -7709321041217"
    using bernoulli_num_denom_eqI(1)[of 32 "-7709321041217" "510"] bernoulli_32
by simp
  thus ?thesis
    by (intro irregular_primeI[of _ 32]) simp_all
qed
```

Irregularity of primes can be certified relatively easily with the code generator:

```
experiment
begin

lemma irregular_59: "irregular_prime 59"
proof (rule irregular_primeI)
  show "int 59 dvd bernoulli_num 44"
    by eval
qed auto

lemma irregular_67: "irregular_prime 67"
```

```
proof (rule irregular_primeI)
  show "int 67 dvd bernoulli_num 58"
    by eval
qed auto

end

end
```

6 Infinitude of irregular primes

```
theory Irregular_Primes_Infinite
  imports Regular_Primes
begin
```

One consequence of Kummer's congruence is that there are infinitely many irregular primes. We shall derive this here.

```
lemma zeta_real_gt_1:
  assumes "x > 1"
  shows   "Re (zeta (of_real x)) > 1"
proof -
  have *: "(λn. Re (complex_of_nat (Suc n) powr -of_real x)) sums (Re
(zeta x))"
    using assms by (intro sums_Re sums_zeta) auto
  have **: "(Re (zeta x) - (1 + Re (2 powr -of_real x))) ≥ 0"
  proof (rule sums_le)
    show "(λn. Re (complex_of_nat (n+3) powr -of_real x)) sums (Re (zeta
x) - (1 + Re (2 powr -of_real x)))"
      using sums_split_initial_segment[OF *, of 2] by (simp add: eval_nat_numeral)
    show "(λ_. 0) sums (0 :: real)"
      by simp
  next
    fix n :: nat
    have "complex_of_nat (n + 3) powr -complex_of_real x = of_real (real
(n + 3) powr -x)"
      by (subst powr_of_real [symmetric]) auto
    also have "Re ... = real (n + 3) powr -x"
      by simp
    also have "... ≥ 0"
      by simp
    finally show "Re (complex_of_nat (n + 3) powr -complex_of_real x) ≥
0" .
  qed

  have "0 < Re (of_real (2 powr -x))"
    by simp
  also have "complex_of_real (2 powr -x) = 2 powr -complex_of_real x"
    by (subst powr_of_real [symmetric]) auto
  also have "Re (2 powr -of_real x) ≤ Re (zeta x) - 1"
```

```
    using ** by simp
  finally show ?thesis
    by linarith
qed

lemma zeta_real_gt_1':
  assumes "Re s > 1" "s ∈ ℝ"
  shows   "Re (zeta s) > 1"
  using assms by (elim Reals_cases) (auto simp: zeta_real_gt_1)

lemma bernoulli_even_conv_zeta:
  "complex_of_real (bernoulli (2*n)) = (-1)^Suc n * 2 * fact (2*n) / (2*pi)^(2*n)
* zeta (2 * of_nat n)"
  by (subst zeta_even_nat[of n]) (auto simp: field_simps)

lemma bernoulli_even_conv_zeta':
  "bernoulli (2*n) = (-1)^Suc n * 2 * fact (2*n) / (2*pi)^(2*n) * Re (zeta
(2 * of_nat n))"
proof -
  have "complex_of_real (bernoulli (2*n)) = (-1)^Suc n * 2 * fact (2*n)
/ (2*pi)^(2*n) * zeta (2 * of_nat n)"
    by (rule bernoulli_even_conv_zeta)
  also have "... = of_real ((-1)^Suc n * 2 * fact (2*n) / (2*pi)^(2*n)
* Re (zeta (2 * of_nat n)))"
    using zeta_real'[of "2 * of_nat n"] by simp
  finally show ?thesis
    by (subst (asm) of_real_eq_iff)
qed

lemma abs_bernoulli_even_conv_zeta:
  assumes "even n" "n > 0"
  shows   "|bernoulli n| = 2 * fact n / (2*pi)^n * Re (zeta (of_nat n))"
proof -
  from assms obtain k where n: "n = 2 * k"
    by (elim evenE)
  show ?thesis
    using zeta_real_gt_1'[of n] assms(2) unfolding n bernoulli_even_conv_zeta'
    by (simp add: abs_mult)
qed


lemma abs_bernoulli_over_n_ge_2:
  assumes "n ≥ 23" "even n"
  shows   "|bernoulli n / n| ≥ 2"
proof -
  have "2 * real n ≤ 2 * real n * 1"
    by simp
  also have "... ≤ 2 * (fact n / (2*pi)^n) * Re (zeta n)"
    using fact_ge_2pi_power[OF assms(1)] assms(1)
```

```
    by (intro mult_left_mono mult_mono less_imp_le[OF zeta_real_gt_1'])
(auto simp: field_simps)
  also have "... = abs (bernoulli n)"
    using assms by (subst abs_bernoulli_even_conv_zeta) auto
  finally show ?thesis
    using assms(1) by (simp add: field_simps)
qed

lemma infinite_irregular_primes_aux:
  assumes "finite P" "∀p∈P. irregular_prime p" "37 ∈ P"
  shows   "∃p. irregular_prime p ∧ p ∉ P"
proof -
  define n where "n = (∏p∈P. p - 1)"
  have nz: "(∏p∈P. p - 1) ≠ 0"
    using assms by (subst prod_zero_iff) (auto simp: irregular_prime_def)
  define N where "N = bernoulli_num"
  define D where "D = bernoulli_denom"

  have "37 - 1 dvd (∏p∈P. p - 1)"
    by (rule dvd_prodI) (use assms in auto)
  hence "n ≥ 36"
    unfolding n_def using nz by (intro dvd_imp_le) auto
  have "even n"
    unfolding n_def using assms by (auto simp: even_prod_iff intro!: bexI[of
_ 37])
  have [simp]: "N n ≠ 0"
    using ‹even n› by (auto simp: N_def bernoulli_num_eq_0_iff)

  have "abs (bernoulli n / n) > 1"
  proof -
    have "1 < (2 :: real)"
      by simp
    also have "... ≤ abs (bernoulli n / n)"
      by (rule abs_bernoulli_over_n_ge_2) (use ‹even n› ‹n ≥ 36› in auto)
    finally show ?thesis .
  qed

  obtain p where p: "prime p" "qmultiplicity (int p) (bernoulli_rat n
/ of_nat n) > 0"
  proof -
    obtain a b where ab: "quotient_of (bernoulli_rat n / of_nat n) =
(a, b)"
      using prod.exhaust by blast
    have "b > 0"
      using ab quotient_of_denom_pos by blast
    have eq': "of_int a / of_int b = (bernoulli_rat n / of_nat n :: rat)"
      using ab quotient_of_div by simp
    also have "(of_rat ... :: real) = bernoulli n / n"
      by (simp add: of_rat_divide bernoulli_rat_conv_bernoulli)
```

```
    finally have eq: "real_of_rat (rat_of_int a / rat_of_int b) = bernoulli
n / real n" .

    have [simp]: "a ≠ 0"
      using eq ‹abs (bernoulli n / n) > 1› by auto
    moreover have "¬is_unit a "
    proof
      assume "is_unit a"
      hence "abs (bernoulli n / n) = 1 / of_int b"
        unfolding eq [symmetric] using ‹b > 0› by (simp add: of_rat_divide
flip: of_int_abs)
      also have "... ≤ 1"
        using ‹b > 0› by simp
      finally show False
        using ‹abs (bernoulli n / n) > 1› by simp
    qed
    ultimately obtain p' where p': "prime p'" "p' dvd a"
      using prime_divisor_exists by blast
    define p where "p = nat p'"
    from p' have p: "prime p" "int p dvd a"
      unfolding p_def by (auto intro: prime_ge_0_int)

    show ?thesis
    proof (rule that[of p])
      from p have "multiplicity p a ≥ 1"
        by (intro multiplicity_geI) auto
      moreover have "coprime a b"
        using ab quotient_of_coprime by auto
      hence "¬p dvd b"
        using ab p by (meson not_coprimeI not_prime_unit prime_nat_int_transfer)
      hence "multiplicity p b = 0"
        by (intro not_dvd_imp_multiplicity_0) auto
      ultimately show "qmultiplicity (int p) (bernoulli_rat n / of_nat
n) > 0"
        using ‹b > 0› p unfolding eq'[symmetric] by (subst qmultiplicity_divide_of_int)
auto
    qed fact+
  qed
  hence "qmultiplicity p (of_int (N n) / of_int (int (D n * n))) > 0"
    by (simp add: N_def D_def bernoulli_rat_def)
  hence "multiplicity (int p) (N n) > 0"
    unfolding bernoulli_rat_def using p
    by (subst (asm) qmultiplicity_divide_of_int)
       (use ‹n ≥ 36› ‹even n› in ‹simp_all add: N_def D_def bernoulli_num_eq_0_iff›)
  hence "p dvd N n"
    using not_dvd_imp_multiplicity_0 by fastforce

  have "¬(p - 1) dvd n"
  proof
```

```
    assume "(p - 1) dvd n"
    hence "p dvd D n"
      using p(1) ‹even n› ‹n ≥ 36› unfolding D_def by (subst prime_dvd_bernoulli_denom_iff)
auto
    hence "¬p dvd N n" unfolding N_def D_def
      using coprime_bernoulli_num_denom[of n] p(1)
      by (metis coprime_def int_dvd_int_iff not_prime_unit of_nat_1)
    with ‹p dvd N n› show False
      by contradiction
  qed
  hence "p ∉ P" "p ≠ 2"
    unfolding n_def using assms by auto
  hence "p > 2"
    using prime_gt_1_nat[OF ‹prime p›] by linarith
  hence "odd p"
    using prime_odd_nat[of p] ‹prime p› by auto

  define r where "r = n mod (p - 1)"
  have "even (n - n div (p - 1) * (p - 1))"
    using ‹p > 2› ‹odd p› ‹even n› by (subst even_diff_nat) auto
  also have "... = r"
    unfolding r_def by (rule minus_div_mult_eq_mod)
  finally have "even r" .

  have "irregular_prime p"
  proof (rule irregular_primeI)
    have "r > 0" "r < p - 1"
      using p prime_gt_1_nat[of p] ‹¬(p - 1) dvd n›
      unfolding r_def by (auto intro!: pos_mod_bound Nat.gr0I)
    moreover from ‹even r› and ‹prime p› have "r ≠ 1" "r ≠ p - 2"
      using ‹odd p› ‹p > 2› by auto
    ultimately have "r ≥ 2" "r ≤ p - 3"
      by linarith+
    thus "r ∈ {2..p-3}"
      by auto

    have "bernoulli_rat r / of_nat r = (0 :: rat) ∨
          qmultiplicity p (bernoulli_rat r / of_nat r) > 0"
    proof (rule qcong_qmultiplicity_pos_transfer)
      show "qmultiplicity p (bernoulli_rat n / of_nat n) > 0"
        using p by simp
    next
      have "[n = r] (mod (p - 1))"
        by (auto simp: Cong.cong_def r_def)
      thus "[bernoulli_rat n / of_nat n = bernoulli_rat r / of_nat r]
(qmod p)"
        using p(1) ‹even n› ‹even r› ‹n ≥ _› ‹r > 0› ‹¬(p-1) dvd n› un-
folding D_def N_def
        by (intro kummer_congruence'_prime) auto
```

```
    qed
    moreover have "bernoulli_rat r / of_nat r ≠ (0 :: rat)"
      using ‹even r› ‹r > 0› by (auto simp: bernoulli_rat_eq_0_iff)
    ultimately have "qmultiplicity p (bernoulli_rat r / of_nat r) > 0"
      by simp
    hence "qmultiplicity p (of_int (N r) / of_int (int (D r * r))) > 0"
      by (simp add: bernoulli_rat_def N_def D_def)
    hence "multiplicity (int p) (N r) > 0"
      by (subst (asm) qmultiplicity_divide_of_int)
         (use ‹r > 0› ‹even r› ‹prime p› in ‹simp_all add: N_def D_def
bernoulli_num_eq_0_iff›)
    hence "p dvd N r"
      using not_dvd_imp_multiplicity_0 by fastforce
    thus "p dvd bernoulli_num r"
      by (simp add: N_def)
  qed (use p ‹p ≠ 2› ‹even r› in auto)
  with ‹p ∉ P› show ?thesis
    by blast
qed

theorem infinite_irregular_primes: "infinite {p. irregular_prime p}"
proof
  assume "finite {p. irregular_prime p}"
  hence "∃p. irregular_prime p ∧ p ∉ {p. irregular_prime p}"
    by (rule infinite_irregular_primes_aux) (use irregular_prime_37 in
auto)
  thus False
    by simp
qed

end
```

References

[1] H. Cohen. *Number Theory: Volume II: Analytic and Modern Tools.* Graduate Texts in Mathematics. Springer New York, 2007.