The Sum-of-Squares Function and Jacobi's Two-Square Theorem

Manuel Eberl

March 10, 2025

Abstract

This entry defines the *sum-of-squares function* $r_k(n)$, which counts the number of ways to write a natural number n as a sum of k squares of integers. Signs and permutations of these integers are taken into account, such that e.g. $1^2 + 2^2$, $2^2 + 1^2$, and $(-1)^2 + 2^2$ are all different decompositions of 5.

Using this, I then formalise the main result: Jacobi's two-square theorem, which states that for $n > 0$ we have $r_2(n) = 4(d_1(3) - d_3(n))$, where $d_i(n)$ denotes the number of divisors m of n such that $m = i \pmod 4$.

Corollaries include the identities $r_2(2n) = r_2(n)$ and $r_2(p^2 n) = r_2(n)$ if $p = 3 \pmod 4$ and the well-known theorem that $r_2(n) = 0$ iff n has a prime factor p of odd multiplicity with $p = 3 \pmod 4$.

Contents

1 Sum-of-square decompositions and Jacobi's two-squares Theorem

```
theory Sum_Of_Squares_Count
imports
  "HOL-Library.Discrete_Functions"
  "HOL-Library.FuncSet"
  "Gaussian_Integers.Gaussian_Integers"
  "Dirichlet_Series.Multiplicative_Function"
  "List-Index.List_Index"
begin
```

1.1 Auxiliary material

```
lemma is_square_conv_sqrt: "is_square n ⟷ floor_sqrt n ^ 2 = n"
  by (metis is_nth_power_def floor_sqrt_inverse_power2)

lemma sum_replicate_mset_count_eq: "(∑x∈set_mset X. replicate_mset
(count X x) x) = X"
  by (rule multiset_eqI) (auto simp: count_sum Multiset.not_in_iff)

lemma coprime_crossproduct_strong:
  fixes a b c d :: "'a :: semiring_gcd"
  assumes "coprime a d" "coprime b c"
  shows   "normalize (a * b) = normalize (c * d) ⟷
             normalize a = normalize c ∧ normalize b = normalize d"
proof
  assume *: "normalize (a * b) = normalize (c * d)"
  show "normalize a = normalize c ∧ normalize b = normalize d"
  proof
    have "a dvd c"
      by (metis assms(1) * coprime_dvd_mult_left_iff dvd_mult_left dvd_refl
normalize_dvd_iff)
    moreover have "c dvd a"
      by (metis assms(2) * coprime_commute coprime_dvd_mult_left_iff
            dvd_mult_left dvd_refl normalize_dvd_iff)
    ultimately show "normalize a = normalize c"
      by (intro associatedI)
  next
    have "b dvd d"
      by (metis assms(2) * coprime_dvd_mult_left_iff dvd_mult_left dvd_refl

            mult.commute normalize_dvd_iff)
    moreover have "d dvd b"
      by (metis assms(1) * coprime_commute coprime_dvd_mult_right_iff
dvd_normalize_iff
            dvd_triv_right)
    ultimately show "normalize b = normalize d"
      by (intro associatedI)
```

```
  qed
next
  assume "normalize a = normalize c ∧ normalize b = normalize d"
  thus "normalize (a * b) = normalize (c * d)"
    by (meson associated_iff_dvd mult_dvd_mono)
qed

lemma divisor_coprime_product_decomp_normalize:
  fixes d n1 n2 :: "'a :: factorial_semiring_gcd"
  assumes "d dvd n1 * n2" "coprime n1 n2"
  shows   "normalize d = normalize (gcd d n1 * gcd d n2)"
proof -
  obtain d3 d4 where d34: "d = d3 * d4" "d3 dvd n1" "d4 dvd n2"
    using division_decomp[of d n1 n2] assms by auto
  have "gcd d n1 = normalize d3"
    using d34 assms
    by (metis coprime_mult_right_iff dvd_div_mult_self gcd_mult_left_right_cancel
gcd_proj1_iff)
  moreover have "gcd d n2 = normalize d4"
    using d34 assms
    by (metis coprime_commute coprime_mult_right_iff dvd_div_mult_self

            gcd_mult_left_left_cancel gcd_proj1_iff)
  ultimately show ?thesis
    using d34 by simp
qed

lemma divisor_coprime_product_decomp:
  fixes d n1 n2 :: nat
  assumes "d dvd n1 * n2" "coprime n1 n2"
  shows   "d = gcd d n1 * gcd d n2"
  using divisor_coprime_product_decomp_normalize[of d n1 n2] assms
  by simp
```

1.2 Decompositions into squares of integers

The following definition gives the set of all the different ways to decompose a natural number n into a sum of k squares of integers. The signs and permutation of these integers is taken into account, i.e. $1^2 + 2^2$, $2^2 + 1^2$, and $1^2 + (-2)^2$ are all counted as different decompositions of 5.

```
definition sos_decomps :: "nat ⇒ nat ⇒ int list set" where
  "sos_decomps k n = {xs. length xs = k ∧ int n = (∑x←xs. x ^ 2)}"
```

The following function that counts the number of such decompositions is known as the "sum-of-squares function" in the literature, and frequently denoted with $r_k(n)$.

```
definition count_sos :: "nat ⇒ nat ⇒ nat" where
  "count_sos k n = card (sos_decomps k n)"
```

```
lemma finite_sos_decomps [simp, intro]: "finite (sos_decomps k n)"
proof (rule finite_subset)
  show "sos_decomps k n ⊆ {xs. set xs ⊆ {-int n..int n} ∧ length xs
= k}"
  proof safe
    fix xs x assume xs: "xs ∈ sos_decomps k n" and x: "x ∈ set xs"
    have "|x| ≤ x ^ 2"
      using self_le_power[of "|x|" 2] by (cases "x = 0") auto
    also have "x ^ 2 ≤ (∑x←xs. x ^ 2)"
      by (rule member_le_sum_list) (use x in auto)
    finally show "x ∈ {- int n..int n}"
      using xs by (auto simp: sos_decomps_def)
  qed (auto simp: sos_decomps_def)
next
  show "finite {xs. set xs ⊆ {-int n..int n} ∧ length xs = k}"
    by (rule finite_lists_length_eq) auto
qed

lemma sos_decomps_0_right [simp]: "sos_decomps k 0 = {replicate k 0}"
proof -
  have "xs = replicate k 0" if "xs ∈ sos_decomps k 0" for xs
  proof -
    have xs: "length xs = k" "(∑x←xs. x ^ 2) = 0"
      using that by (auto simp: sos_decomps_def)
    have "∀x∈set xs. x = 0"
      using xs by (subst (asm) sum_list_nonneg_eq_0_iff) auto
    thus ?thesis
      using xs(1) by (intro replicate_eqI) auto
  qed
  thus ?thesis
    by (auto simp: sos_decomps_def sum_list_replicate)
qed

lemma sos_decomps_0: "sos_decomps 0 n = (if n = 0 then {[]} else {})"
  by (auto simp: sos_decomps_def)

lemma sos_decomps_1:
  "sos_decomps (Suc 0) n = (if is_square n then {[floor_sqrt n], [-floor_sqrt
n]} else {})"
  (is "?lhs = ?rhs")
proof (intro equalityI subsetI)
  fix xs assume "xs ∈ ?lhs"
  then obtain x where [simp]: "xs = [x]" and x: "int n = x ^ 2"
    by (auto simp: sos_decomps_def length_Suc_conv)
  have "int n = x ^ 2"
    by fact
  also have "x ^ 2 = int (nat |x| ^ 2)"
    by auto
```

```
  finally have n_eq: "n = nat |x| ^ 2"
    by linarith
  show "xs ∈ ?rhs"
    using x by (auto simp: n_eq)
qed (auto simp: sos_decomps_def split: if_splits elim!: is_nth_powerE)

lemma bij_betw_sos_decomps_2: "bij_betw (λ(x,y). [x,y]) {(i,j). i² +
j² = int n} (sos_decomps 2 n)"
  by (rule bij_betwI[of _ _ _ "λxs. (xs ! 0, xs ! 1)"])
     (auto simp: length_Suc_conv eval_nat_numeral sos_decomps_def)

lemma sos_decomps_Suc:
  "sos_decomps (Suc k) n =
     (#) 0 ` sos_decomps k n ∪
     (⋃i∈{1..floor_sqrt n}. ⋃xs∈sos_decomps k (n - i ^ 2). {int i #
xs, (-int i) # xs})"
  (is "?A = ?B ∪ ?C")
proof (intro equalityI subsetI)
  fix xs assume "xs ∈ ?B ∪ ?C"
  thus "xs ∈ ?A"
    by (auto simp: sos_decomps_def of_nat_diff le_floor_sqrt_iff)
next
  fix xs assume "xs ∈ ?A"
  hence xs: "length xs = Suc k" "int n = (∑x←xs. x ^ 2)"
    by (auto simp: sos_decomps_def)
  then obtain x xs' where xs_eq: "xs = x # xs'"
    by (cases xs) auto
  show "xs ∈ ?B ∪ ?C"
  proof (cases "x = 0")
    case True
    hence "xs ∈ ?B"
      using xs by (auto simp: sos_decomps_def xs_eq)
    thus ?thesis ..
  next
    case False
    define y where "y = nat |x|"
    have "y ∈ {1..floor_sqrt n}" and "y ^ 2 ≤ n"
    proof -
      have *: "x ^ 2 = int y ^ 2"
        by (auto simp: y_def)
      have "int y ^ 2 ≤ int n"
        using xs by (auto simp: xs_eq * intro!: sum_list_nonneg)
      thus "y ^ 2 ≤ n"
        unfolding of_nat_power [symmetric] by linarith
      moreover have "y ≥ 1"
        using False by (auto simp: y_def)
      ultimately show "y ∈ {1..floor_sqrt n}"
        by (simp add: le_floor_sqrt_iff)
    qed
```

```
    have x_disj: "x = int y ∨ x = -int y"
      by (auto simp: y_def)
    hence "xs ∈ ?C"
      using xs False ‹y ∈ _› ‹y ^ 2 ≤ n› x_disj
      by (auto simp: sos_decomps_def xs_eq of_nat_diff intro!: bexI[of
_ "nat |x|"] exI[of _ xs'])
    thus ?thesis ..
  qed
qed

lemma count_sos_0_right [simp]: "count_sos k 0 = 1"
  unfolding count_sos_def by simp

lemma count_sos_0 [simp]: "n > 0 ⟹ count_sos 0 n = 0"
  unfolding count_sos_def by (subst sos_decomps_0) auto

lemma count_sos_1: "n > 0 ⟹ count_sos (Suc 0) n = (if is_square n then
2 else 0)"
  unfolding count_sos_def by (subst sos_decomps_1) auto

lemma count_sos_2: "count_sos 2 n = card {(i,j). i² + j² = int n}"
  using bij_betw_same_card[OF bij_betw_sos_decomps_2[of n]] by (simp add:
count_sos_def)
```

The following obvious recurrence for $r_k(n)$ allows us to compute $r_k(n)$ for concrete k, n – albeit rather inefficiently:

$$r_{k+1}(n) = r_k(n) + 2\sum_{i=1}^{\lfloor\sqrt{n}\rfloor} r_k(n - i^2)$$

```
lemma count_sos_Suc:
  "count_sos (Suc k) n = count_sos k n + 2 * (∑i=1..floor_sqrt n. count_sos
k (n - i ^ 2))"
proof -
  have "count_sos (Suc k) n = card ((#) 0 ` sos_decomps k n ∪
          (⋃i∈{1..floor_sqrt n}. ⋃xs∈sos_decomps k (n - i²). {int i
# xs, - int i # xs}))"
    (is "_ = card (?A ∪ ?B)") unfolding count_sos_def sos_decomps_Suc
..
  also have "... = card ?A + card ?B"
    by (subst card_Un_disjoint) auto
  also have "card ?A = count_sos k n"
    unfolding count_sos_def by (subst card_image) auto
  also have "card ?B = (∑i=1..floor_sqrt n. card (⋃xs∈sos_decomps k
(n - i²). {int i # xs, - int i # xs}))"
    by (rule card_UN_disjoint) auto
  also have "... = (∑i=1..floor_sqrt n. 2 * count_sos k (n - i ^ 2))"
    by (rule sum.cong) (auto simp: card_UN_disjoint count_sos_def)
  finally show ?thesis
```

```
    by (simp add: sum_distrib_left)
qed

lemma count_sos_code [code]:
  "count_sos k n = (if n = 0 then 1
     else if k = 0 then 0
     else if k = 1 then (if floor_sqrt n ^ 2 = n then 2 else 0)
     else count_sos (k-1) n + 2 * (∑i=1..floor_sqrt n. count_sos (k-1)
(n-i^2)))"
  unfolding is_square_conv_sqrt [symmetric] using count_sos_Suc[of "k-1"
n]
  by (auto simp: count_sos_1)
```

1.3 Decompositions into squares of positive integers

It seems somewhat unnatural to allow $(-x)^n$ and x^n as two different squares (for nonzero x), and it may also seem strange to allow 0^2 in the decomposition. However, as we will see later, this notion of square decomposition has some nice properties.

Still, we now introduce the perhaps more intuitively sensible definition of the different ways to decompose n into k squares of *positive* integers, and relate it to what we introduced above.

```
definition pos_sos_decomps :: "nat ⇒ nat ⇒ nat list set" where
  "pos_sos_decomps k n = {xs. length xs = k ∧ 0 ∉ set xs ∧ n = (∑x←xs.
x ^ 2)}"

definition count_pos_sos :: "nat ⇒ nat ⇒ nat" where
  "count_pos_sos k n = card (pos_sos_decomps k n)"

lemma finite_pos_sos_decomps [simp, intro]: "finite (pos_sos_decomps
k n)"
proof -
  have "map int ` pos_sos_decomps k n ⊆ sos_decomps k n"
    by (auto simp: pos_sos_decomps_def sos_decomps_def o_def simp flip:
sum_list_of_nat)
  moreover have "finite (sos_decomps k n)"
    by blast
  ultimately have "finite (map int ` pos_sos_decomps k n)"
    using finite_subset by blast
  also have "?this ⟷ finite (pos_sos_decomps k n)"
    by (subst finite_image_iff) (auto intro!: inj_onI)
  finally show ?thesis .
qed

lemma pos_sos_decomps_0_right: "pos_sos_decomps k 0 = (if k = 0 then
{[]} else {})"
proof (intro equalityI subsetI)
  fix xs assume "xs ∈ pos_sos_decomps k 0"
```

```
  hence "xs = [] ∧ k = 0"
    by (cases xs) (auto simp: pos_sos_decomps_def)
  thus "xs ∈ (if k = 0 then {[]} else {})"
    by auto
qed (auto simp: pos_sos_decomps_def split: if_splits)

lemma pos_sos_decomps_0: "pos_sos_decomps 0 n = (if n = 0 then {[]} else
{})"
  by (auto simp: pos_sos_decomps_def)

lemma pos_sos_decomps_1:
  "pos_sos_decomps (Suc 0) n = (if is_square n ∧ n > 0 then {[floor_sqrt
n]} else {})"
  (is "?lhs = ?rhs")
proof (intro equalityI subsetI)
  fix xs assume "xs ∈ ?lhs"
  then obtain x where [simp]: "xs = [x]" and n_eq: "n = x ^ 2" and "x
> 0"
    unfolding pos_sos_decomps_def length_Suc_conv by force
  show "xs ∈ ?rhs"
    using ‹x > 0› by (auto simp: n_eq)
qed (auto simp: pos_sos_decomps_def split: if_splits elim!: is_nth_powerE)

lemma bij_betw_pos_sos_decomps_2:
  "bij_betw (λ(x,y). [x,y]) {(i,j). i² + j² = n ∧ i > 0 ∧ j > 0} (pos_sos_decomps
2 n)"
  by (rule bij_betwI[of _ _ _ "λxs. (xs ! 0, xs ! 1)"])
     (auto simp: length_Suc_conv eval_nat_numeral pos_sos_decomps_def)

lemma pos_sos_decomps_Suc:
  "pos_sos_decomps (Suc k) n =
     (⋃i∈{1..floor_sqrt n}. ((#) i) ` pos_sos_decomps k (n - i ^ 2))"
  (is "?A = ?B")
proof (intro equalityI subsetI)
  fix xs assume "xs ∈ ?B"
  thus "xs ∈ ?A"
    by (auto simp: pos_sos_decomps_def of_nat_diff le_floor_sqrt_iff)
next
  fix xs assume "xs ∈ ?A"
  hence xs: "length xs = Suc k" "n = (∑x←xs. x ^ 2)" "0 ∉ set xs"
    by (auto simp: pos_sos_decomps_def)
  then obtain x xs' where xs_eq: "xs = x # xs'"
    by (cases xs) auto

  have "x ∈ {1..floor_sqrt n}" and "x ^ 2 ≤ n"
  proof -
    have "x ^ 2 ≤ int n"
      using xs by (auto simp: xs_eq intro!: sum_list_nonneg)
    thus "x ^ 2 ≤ n"
```

```
      unfolding of_nat_power [symmetric] by linarith
    moreover have "x ≥ 1"
      using xs by (auto simp: xs_eq)
    ultimately show "x∈ {1..floor_sqrt n}"
      by (simp add: le_floor_sqrt_iff)
  qed
  thus "xs ∈ ?B"
    using xs ‹x ∈ _› ‹x ^ 2 ≤ n›
    by (auto simp: pos_sos_decomps_def xs_eq of_nat_diff intro!: bexI[of
_ x] exI[of _ xs'])
qed

lemma count_pos_sos_0_right: "count_pos_sos k 0 = (if k = 0 then 1 else
0)"
  unfolding count_pos_sos_def by (simp add: pos_sos_decomps_0_right)

lemma count_pos_sos_0: " count_pos_sos 0 n = (if n = 0 then 1 else 0)"
  unfolding count_pos_sos_def by (subst pos_sos_decomps_0) auto

lemma count_pos_sos_0_0 [simp]: "count_pos_sos 0 0 = 1"
  and count_pos_sos_0_right' [simp]: "k > 0 ⟹ count_pos_sos k 0 = 0"
  and count_pos_sos_0' [simp]: "n > 0 ⟹ count_pos_sos 0 n = 0"
  by (simp_all add: count_pos_sos_0 count_pos_sos_0_right)

lemma count_pos_sos_1: "count_pos_sos (Suc 0) n = (if is_square n ∧
n > 0 then 1 else 0)"
  unfolding count_pos_sos_def by (subst pos_sos_decomps_1) auto

lemma count_pos_sos_2: "count_pos_sos 2 n = card {(i,j). i² + j² = n
∧ i > 0 ∧ j > 0}"
  using bij_betw_same_card[OF bij_betw_pos_sos_decomps_2[of n]]
  by (simp add: count_pos_sos_def)
```

We get a similar recurrence for *count_pos_sos* as earlier:

```
lemma count_pos_sos_Suc:
  "count_pos_sos (Suc k) n = (∑i=1..floor_sqrt n. count_pos_sos k (n
- i ^ 2))"
proof -
  have "count_pos_sos (Suc k) n =
          card ((⋃i∈{1..floor_sqrt n}. (#) i ` pos_sos_decomps k (n -
i²)))"
    unfolding count_pos_sos_def pos_sos_decomps_Suc ..
  also have "... = (∑i=1..floor_sqrt n. card ((#) i ` pos_sos_decomps
k (n - i²)))"
    by (rule card_UN_disjoint) auto
  also have "... = (∑i=1..floor_sqrt n. count_pos_sos k (n - i ^ 2))"
    by (rule sum.cong) (auto simp: card_UN_disjoint count_pos_sos_def
card_image)
  finally show ?thesis
```

```
  by (simp add: sum_distrib_left)
qed

lemma count_pos_sos_code [code]:
  "count_pos_sos k n = (if k = 0 ∧ n = 0 then 1
     else if k = 0 ∨ n = 0 then 0
     else if k = 1 then (if floor_sqrt n ^ 2 = n then 1 else 0)
     else (∑i=1..floor_sqrt n. count_pos_sos (k-1) (n-i^2)))"
  unfolding is_square_conv_sqrt [symmetric] using count_pos_sos_Suc[of
"k-1" n]
  by (auto simp: count_pos_sos_1)
```

If we denote the number of decompositions of n into k squares of integers as $r_k(n)$ and the number of decompositions of n into k *positive* integers as $r_k^+(n)$, we can show the following formula:

$$r_k(n) = \sum_{j=0}^{k} 2^j \binom{k}{j} r_j^+(n)$$

There is a simple combinatorial argument for this: any decomposition of n into k squares of integers can be produced by picking

- an integer j between 0 and k determining how many of the squares in the decomposition will be non-zero
- a set $X \subseteq [k]$ with $|X| = j$ of their indices
- a function $s : X \to \{-1, 1\}$ determining the sign of each of the j non-zero integers
- a decomposition of n into j squares, which determines the absolute values of each of the j integers

The inverse of this process is also clear: given a decomposition of n into k squares of integers, j is the number of non-zero integers in it, X is the set of all indices with a non-zero integer, $s(i)$ is the sign of the i-th integer, and the absolute values of the j non-zero integers in the decomposition form a decomposition of n into j squares of positive integers.

However, this proof is somewhat tedious to write down because it is not so easy to, given a list `xs` with k elements and a set $X \subseteq [k]$ of indices, construct a list that has the elements of `xs` at the indices X left-to-right and 0 everywhere else.

Therefore, we simply use a straightforward induction on k instead, which is also simple to do, albeit perhaps less insightful.

```
lemma count_sos_conv_count_pos_sos:
  "count_sos k n = (∑j≤k. 2 ^ j * (k choose j) * count_pos_sos j n)"
```

```
proof (induction k arbitrary: n)
  case (Suc k n)
  define m where "m = floor_sqrt n"
  have "(∑j≤Suc k. 2 ^ j * (Suc k choose j) * count_pos_sos j n) =
        count_pos_sos 0 n +
        (∑j≤k. 2 ^ Suc j * (k choose j) * count_pos_sos (Suc j) n) +
        (∑j≤k. 2 ^ Suc j * (k choose Suc j) * count_pos_sos (Suc j)
n)"
    by (subst sum.atMost_Suc_shift) (simp_all add: ring_distribs sum.distrib)
  also have "(∑j≤k. 2 ^ Suc j * (k choose Suc j) * count_pos_sos (Suc
j) n) =
             (∑j∈{1..Suc k}. 2 ^ j * (k choose j) * count_pos_sos j
n)"
    by (intro sum.reindex_bij_witness[of _ "λj. j - 1" Suc]) auto
  also have "(∑j≤k. 2 ^ Suc j * (k choose j) * count_pos_sos (Suc j)
n) =
             (∑j≤k. ∑i=1..m. 2 ^ Suc j * (k choose j) * count_pos_sos
j (n - i²))"
    by (simp add: count_pos_sos_Suc sum_distrib_left mult.assoc m_def)
  also have "... = (∑i=1..m. ∑j≤k. 2 ^ Suc j * (k choose j) * count_pos_sos
j (n - i²))"
    by (rule sum.swap)
  finally have "(∑j≤Suc k. 2 ^ j * (Suc k choose j) * count_pos_sos j
n) =
                count_pos_sos 0 n + (∑j=1..Suc k. 2 ^ j * (k choose j)
* count_pos_sos j n) +
                (∑i=1..m. ∑j≤k. 2 ^ Suc j * (k choose j) * count_pos_sos
j (n - i²))"
    by Groebner_Basis.algebra
  also have "count_pos_sos 0 n + (∑j=1..Suc k. 2 ^ j * (k choose j) *
count_pos_sos j n) =
             (∑j∈insert 0 {1..Suc k}. 2 ^ j * (k choose j) * count_pos_sos
j n)"
    by (subst sum.insert) auto
  also have "insert 0 {1..Suc k} = {..Suc k}"
    by auto
  also have "(∑j≤Suc k. 2 ^ j * (k choose j) * count_pos_sos j n) =
             (∑j≤k. 2 ^ j * (k choose j) * count_pos_sos j n)"
    by (rule sum.mono_neutral_right) auto
  also have "(∑j≤k. 2 ^ j * (k choose j) * count_pos_sos j n) +
             (∑i = 1..m. ∑j≤k. 2 ^ Suc j * (k choose j) * count_pos_sos
j (n - i²)) =
             count_sos (Suc k) n"
    by (simp add: count_sos_Suc Suc.IH sum_distrib_left mult.assoc m_def)
  finally show ?case ..
qed (auto simp: count_pos_sos_0)
```

We can however, just for illustration, easily establish a bijection between the the set of decompositions of n into k squares of integers and the set of pairs consisting of a decomposition of n into k squares of positive integers and a subset of $[k]$ (indicating which of the integers were originally negative).

This shows that $r_k(n) \geq 2^k r_k^+(n)$ (although we could easily have derived that fact from our identity relating $r_k(n)$ and $r_k^+(n)$ as well).

```
lemma
  fixes k n :: nat
  fixes f :: "nat list × nat set ⇒ int list"
  defines "f ≡ (λ(xs, X). map_index (λi x. if i ∈ X then -int x else int
x) xs)"
  defines "A ≡ pos_sos_decomps k n × Pow {..<k}"
  defines "B ≡ {xs∈sos_decomps k n. 0 ∉ set xs}"
  shows bij_betw_pos_sos_deocmps_nonzero_sos_decomps: "bij_betw f A B"
    and count_sos_ge_twopow_pos_sos: "count_sos k n ≥ 2 ^ k * count_pos_sos
k n"
proof -
  define g :: "int list ⇒ nat list × nat set"
    where "g = (λxs. (map (nat ∘ abs) xs, {i∈{..<k}. xs ! i < 0}))"

  show "bij_betw f A B"
  proof (rule bij_betwI[of _ _ _ g])
    show "f ∈ A → B"
      by (force simp: f_def A_def B_def sos_decomps_def pos_sos_decomps_def
                      map_map_index set_conv_nth sum_list_sum_nth
                intro!: sum.cong split: if_splits)
  next
    show "g ∈ B → A"
    proof
      fix xs assume "xs ∈ B"
      hence xs: "int n = (∑x←xs. x ^ 2)" "0 ∉ set xs" "length xs =
k"
        by (simp_all add: B_def sos_decomps_def)
      note xs(1)
      also have "(∑x←xs. x ^ 2) = int (∑x←xs. (nat |x|) ^ 2)"
        by (subst sum_list_of_nat [symmetric]) (simp_all add: o_def)
      finally have "n = (∑x←xs. (nat |x|) ^ 2)"
        by linarith
      thus "g xs ∈ A"
        using xs by (auto simp: A_def pos_sos_decomps_def g_def o_def)
    qed
  next
    show "g (f xs_X) = xs_X" if "xs_X ∈ A" for xs_X
    proof -
      obtain X xs where [simp]: "xs_X = (xs, X)"
        by (cases xs_X)
      have "X = {i ∈ {..<k}. f (xs, X) ! i < 0}" using that
        by (force simp: f_def A_def pos_sos_decomps_def set_conv_nth split:
if_splits)
      moreover have "xs = map (nat ∘ abs) (f (xs, X))"
```

```
        by (rule nth_equalityI) (use that in ‹auto simp: f_def A_def›)
      ultimately show ?thesis
        by (simp add: g_def)
    qed
  next
    show "f (g xs) = xs" if "xs ∈ B" for xs using that
      by (auto simp: f_def g_def map_index_map B_def sos_decomps_def intro!:
nth_equalityI)
  qed

  have "2 ^ k * count_pos_sos k n = card A"
    by (simp add: A_def card_Pow count_pos_sos_def)
  also have "... = card B"
    using bij_betw_same_card[OF ‹bij_betw f _ _›] .
  also have "... = card {xs∈sos_decomps k n. 0 ∉ set xs}"
    by (simp add: B_def count_sos_def)
  also have "... ≤ card (sos_decomps k n)"
    by (rule card_mono) auto
  also have "... = count_sos k n"
    by (simp add: count_sos_def)
  finally show "count_sos k n ≥ 2 ^ k * count_pos_sos k n" .
qed

value "map (count_pos_sos 2) [0..<100]"
```

1.4 Decompositions into two squares

For the rest of this development, we will focus on $k = 2$, i.e. decompositions of n into two squares. There is an obvious relationship between these and Gaussian integers with norm n.

To that end, recall that the Gaussian integers $\mathbb{Z}[i]$ are the subring of the complex numbers of the form $a + bi$ with $a, b \in \mathbb{Z}$. Their integer-valued norm is defined as $N(a + bi) = a^2 + b^2$ (which is the square of the distance of the complex number $a + bi$ to the origin).

```
lemma in_sos_decomps_2_conv_gauss_int_norm:
   "[x, y] ∈ sos_decomps 2 n ⟷ gauss_int_norm (of_int x + of_int y
* iℤ) = n"
  by (auto simp: sos_decomps_def gauss_int_norm_def)

lemma sos_decomps_2_conv_gauss_int_norm:
   "bij_betw (λz. [ReZ z, ImZ z]) {z. gauss_int_norm z = n} (sos_decomps
2 n)"
  by (rule bij_betwI[of _ _ _ "λxs. of_int (xs ! 0) + of_int (xs ! 1)
* iℤ"])
     (auto simp: sos_decomps_def length_Suc_conv gauss_int_norm_def
                 eval_nat_numeral gauss_int_eq_iff)
```

To make use of this connection, we will now develop some more theory on

Gaussian integers with a given norm n.

1.4.1 Gaussian integers on a circle

We define the set of all Gaussian integers with norm n, i.e. all complex numbers with integer real and imaginary part that lie on a circle of radius n^2 around the origin.

```
definition gauss_ints_with_norm :: "nat ⇒ gauss_int set" where
  "gauss_ints_with_norm n = gauss_int_norm -` {n}"

lemma gauss_ints_with_norm_0 [simp]: "gauss_ints_with_norm 0 = {0}"
  by (auto simp: gauss_ints_with_norm_def)

lemma card_gauss_ints_with_norm_conv_count_sos: "card (gauss_ints_with_norm
n) = count_sos 2 n"
  using bij_betw_same_card[OF sos_decomps_2_conv_gauss_int_norm[of n]]
  by (simp add: gauss_ints_with_norm_def vimage_def count_sos_def)
```

For convenience, we also define the following variant where we restrict the above set to the "standard" quadrant where the real part is positive and the imaginary part is non-negative.

In other words: if we have a Gaussian integer z, there are three more copies of it with the same norm in the other three quadrants, differing from z by one of the unit factors -1, i, or $-i$. It makes sense to therefore only look at the copy in the first quadrant as the "canonical" representative.

```
definition gauss_ints_with_norm' :: "nat ⇒ gauss_int set" where
  "gauss_ints_with_norm' n = gauss_int_norm -` {n} ∩ {z. z ≠ 0 ∧ normalize
z = z}"

lemma gauss_ints_with_norm'_subset:
  "gauss_ints_with_norm' n ⊆ (λ(a,b). of_int a + of_int b * iℤ) ` ({0..int
n} × {0..int n})"
proof
  fix z assume "z ∈ gauss_ints_with_norm' n"
  hence *: "gauss_int_norm z = n" "normalize z = z"
    by (auto simp: gauss_ints_with_norm'_def)
  have nonneg: "ReZ z ≥ 0 ∧ ImZ z ≥ 0"
    using *(2) by (simp add: normalized_gauss_int)
  moreover {
    have "ReZ z ≤ ReZ z ^ 2"
      using self_le_power[of "ReZ z" 2] nonneg by (cases "ReZ z = 0")
auto
    also have "ReZ z ^ 2 ≤ gauss_int_norm z"
      by (simp add: gauss_int_norm_def)
    finally have "ReZ z ≤ int n"
      using * by simp
  } moreover {
```

```
    have "ImZ z ≤ ImZ z ^ 2"
      using self_le_power[of "ImZ z" 2] nonneg by (cases "ImZ z = 0")
auto
    also have "ImZ z ^ 2 ≤ gauss_int_norm z"
      by (simp add: gauss_int_norm_def)
    finally have "ImZ z ≤ int n"
      using * by simp
  }
  ultimately have "ReZ z ∈ {0..int n}" "ImZ z ∈ {0..int n}"
    by auto
  thus "z ∈ (λ(a,b). of_int a + of_int b * iℤ) ` ({0..int n} × {0..int
n})"
    by (intro rev_image_eqI[of "(ReZ z, ImZ z)"]) (simp_all add: gauss_int_eq_iff)
qed

lemma finite_gauss_ints_with_norm' [simp, intro]: "finite (gauss_ints_with_norm'
n)"
  using gauss_ints_with_norm'_subset by (rule finite_subset) auto

lemma gauss_ints_with_norm'_0 [simp]: "gauss_ints_with_norm' 0 = {}"
  by (auto simp: gauss_ints_with_norm'_def)

lemma gauss_ints_with_norm'_1 [simp]: "gauss_ints_with_norm' (Suc 0)
= {1}"
  by (auto simp: gauss_ints_with_norm'_def gauss_int_norm_eq_Suc_0_iff
is_unit_normalize)

lemma unit_factor_eq_1_iff: "unit_factor x = 1 ⟷ normalize x = x ∧
x ≠ 0"
  by (metis unit_factor_0 unit_factor_1_imp_normalized unit_factor_normalize
zero_neq_one)

lemma gauss_ints_with_norm_conv_norm':
  assumes "n > 0"
  shows   "bij_betw (λ(c,z). c * z)
             ({z. is_unit z} × gauss_ints_with_norm' n) (gauss_ints_with_norm
n)"
  by (rule bij_betwI[of _ _ _ "λz. (unit_factor z, normalize z)"])
     (use assms in ‹auto simp: gauss_ints_with_norm'_def gauss_ints_with_norm_def
                               gauss_int_norm_mult gauss_int_norm_eq_Suc_0_iff
is_unit_normalize
                               unit_factor_eq_1_iff›)

lemma finite_gauss_ints_with_norm [simp, intro]: "finite (gauss_ints_with_norm
n)"
proof -
  have "{z. is_unit z} = {1, -1, iℤ, -iℤ}"
    by (auto simp: is_unit_gauss_int_iff)
  thus ?thesis
```

```
    using bij_betw_finite[OF gauss_ints_with_norm_conv_norm'[of n]]
    by (cases "n = 0") auto
qed

lemma card_gauss_ints_with_norm_conv_norm':
  assumes "n > 0"
  shows   "card (gauss_ints_with_norm n) = 4 * card (gauss_ints_with_norm'
n)"
proof -
  define U where "U = {z :: gauss_int. is_unit z}"
  have U_eq: "U = {1, -1, iℤ, -iℤ}"
    by (auto simp: is_unit_gauss_int_iff U_def)
  have [simp]: "finite U" "card U = 4"
    by (auto simp: U_eq gauss_int_eq_iff)
  have "card (gauss_ints_with_norm n) = card (U × gauss_ints_with_norm'
n)"
    unfolding U_def
    by (rule sym, rule bij_betw_same_card, rule gauss_ints_with_norm_conv_norm')
fact
  thus ?thesis
    by simp
qed
```

It now turns out that the number $G(n)$ of Gaussian integers (up to units) with norm n is a multiplicative function in n, meaning that $G(0) = 0$, $G(1) = 1$, and $G(mn) = G(m)G(n)$ if m and n are coprime.

```
lemma gauss_ints_with_norm'_mult_coprime:
  assumes "coprime n1 n2"
  shows   "bij_betw (λ(x1,x2). normalize (x1 * x2))
             (gauss_ints_with_norm' n1 × gauss_ints_with_norm' n2)
             (gauss_ints_with_norm' (n1 * n2))"
  unfolding bij_betw_def
proof
  show "(λ(x, y). normalize (x * y)) ` (gauss_ints_with_norm' n1 × gauss_ints_with_norm'
n2) =
        gauss_ints_with_norm' (n1 * n2)"
  proof safe
    fix z assume z: "z ∈ gauss_ints_with_norm' (n1 * n2)"
    define x1 x2 where "x1 = gcd z (of_nat n1)" and "x2 = gcd z (of_nat
n2)"
    have eq: "of_nat n1 * of_nat n2 = z * gauss_cnj z"
      using z by (simp add: self_mult_gauss_cnj gauss_ints_with_norm'_def)
    hence "z dvd of_nat n1 * of_nat n2"
      by auto
    hence z_eq: "normalize z = normalize (x1 * x2)"
      unfolding x1_def x2_def
      by (rule divisor_coprime_product_decomp_normalize)
         (use assms in ‹simp_all add: coprime_of_nat_gauss_int›)
    then obtain c where c: "is_unit c" "z = c * x1 * x2"
```

```
      by (elim associatedE1) (simp add: algebra_simps)
    have "(of_nat (n1 * n2) :: gauss_int) =
            (c * gauss_cnj c) * (x1 * gauss_cnj x1) * (x2 * gauss_cnj
x2)"
      by (simp add: eq c mult_ac)
    also have "... = of_nat (gauss_int_norm x1 * gauss_int_norm x2)"
      unfolding self_mult_gauss_cnj using c(1)
      by (simp add: is_unit_gauss_int_iff')
    finally have "n1 * n2 = gauss_int_norm x1 * gauss_int_norm x2"
      by (simp only: of_nat_eq_iff)
    moreover have "gauss_int_norm x1 dvd gauss_int_norm (of_nat n1)"
      unfolding x1_def by (rule gauss_int_norm_dvd_mono) auto
    hence "gauss_int_norm x1 dvd n1 ^ 2"
      by simp
    moreover have "gauss_int_norm x2 dvd gauss_int_norm (of_nat n2)"
      unfolding x2_def by (rule gauss_int_norm_dvd_mono) auto
    hence "gauss_int_norm x2 dvd n2 ^ 2"
      by simp
    ultimately have "n1 = gauss_int_norm x1 ∧ n2 = gauss_int_norm x2"
      by (metis assms coprime_crossproduct_nat coprime_mult_left_iff coprime_mult_right_iff
               dvd_div_mult_self power2_eq_square)
    moreover have "x1 ≠ 0" "normalize x1 = x1" "x2 ≠ 0" "normalize x2
= x2"
      using z by (auto simp: x1_def x2_def gauss_ints_with_norm'_def)
    ultimately have "x1 ∈ gauss_ints_with_norm' n1" "x2 ∈ gauss_ints_with_norm'
n2"
                    "z = normalize (x1 * x2)"
      using z_eq z unfolding gauss_ints_with_norm'_def by auto
    thus "z ∈ (λ(x, y). normalize (x * y)) ` (gauss_ints_with_norm' n1
× gauss_ints_with_norm' n2)"
      by fast
  qed (auto simp: gauss_ints_with_norm'_def gauss_int_norm_mult)
next
  show "inj_on (λ(x1, x2). normalize (x1 * x2)) (gauss_ints_with_norm'
n1 × gauss_ints_with_norm' n2)"
  proof (safe intro!: inj_onI)
    fix x1 x2 y1 y2 :: gauss_int
    assume eq: "normalize (x1 * x2) = normalize (y1 * y2)"
    assume x12: "x1 ∈ gauss_ints_with_norm' n1" "y1 ∈ gauss_ints_with_norm'
n1"
                "x2 ∈ gauss_ints_with_norm' n2" "y2 ∈ gauss_ints_with_norm'
n2"

    from eq have "normalize x1 = normalize y1 ∧ normalize x2 = normalize
y2"
    proof (subst (asm) coprime_crossproduct_strong)
      have "coprime (of_nat n1) (of_nat n2 :: gauss_int)"
        using assms by (simp add: coprime_of_nat_gauss_int)
      hence "coprime (x1 * gauss_cnj x1) (y2 * gauss_cnj y2)"
```

```
        using x12 unfolding self_mult_gauss_cnj gauss_ints_with_norm'_def
by simp_all
      thus "coprime x1 y2"
        by simp
    next
      have "coprime (of_nat n2) (of_nat n1 :: gauss_int)"
        using assms by (simp add: coprime_of_nat_gauss_int coprime_commute)
      hence "coprime (x2 * gauss_cnj x2) (y1 * gauss_cnj y1)"
        using x12 unfolding self_mult_gauss_cnj gauss_ints_with_norm'_def
by simp_all
      thus "coprime x2 y1"
        by simp
    qed auto
    thus "x1 = y1" "x2 = y2"
      using x12 by (auto simp: gauss_ints_with_norm'_def)
  qed
qed

interpretation gauss_ints_with_norm': multiplicative_function "λn. card
(gauss_ints_with_norm' n)"
proof
  fix m n :: nat
  assume coprime: "coprime m n"
  show "card (gauss_ints_with_norm' (m * n)) =
        card (gauss_ints_with_norm' m) * card (gauss_ints_with_norm' n)"
    using bij_betw_same_card[OF gauss_ints_with_norm'_mult_coprime[OF
coprime]] by simp
qed auto
```

A similar multiplicativity result for $r_2(n)$ follows, namely

$$r_2(mn) = \frac{1}{4} r_2(m) r_2(n)$$

for m, n positive and coprime.

```
corollary count_sos_2_mult_coprime:
  "m > 0 ⟹ n > 0 ⟹ coprime m n ⟹ 4 * count_sos 2 (m * n) = count_sos
2 m * count_sos 2 n"
  using gauss_ints_with_norm'.mult_coprime[of m n]
  by (cases "m = 0 ∨ n = 0")
     (auto simp: card_gauss_ints_with_norm_conv_norm'
           simp flip: card_gauss_ints_with_norm_conv_count_sos)
```

Since $G(n)$ is multiplicative, it is determined completely by the values it takes on prime powers. We will therefore determine the value of $G(p^k)$ for p being a (rational) prime next, and we distinguish the three cases $p = 2$, $p \equiv 1 \pmod 1$, and $p \equiv 3 \pmod 3$, corresponding to the different ways in which a rational prime p factors in $\mathbb{Z}[i]$

The integer 2 factors into the prime factors into $-i(1+i)^2$ in $\mathbb{Z}[i]$ (where

$1 + i$ is prime and $-i$ is a unit), there is exactly one Gaussian integer with norm 2^n (up to units), namely $(1 + i)^n$.

```
lemma gauss_ints_with_norm'_2_power: "gauss_ints_with_norm' (2 ^ n) =
{normalize ((1 + iℤ) ^ n)}"
proof -
  define p where "p = 1 + iℤ"
  have p: "p ≠ 0" "gauss_int_norm p = 2" "prime p"
    by (auto simp: p_def gauss_int_eq_iff gauss_int_norm_def prime_one_plus_i_gauss_int)
  show ?thesis
  proof (intro equalityI subsetI; (elim singletonE; hypsubst)?)
    show "normalize ((1 + iℤ) ^ n) ∈ gauss_ints_with_norm' (2 ^ n)"
      unfolding p_def [symmetric]
      using p by (auto simp: gauss_ints_with_norm'_def gauss_int_norm_power)
  next
    fix z assume "z ∈ gauss_ints_with_norm' (2 ^ n)"
    hence z: "gauss_int_norm z = 2 ^ n" "normalize z = z"
      by (auto simp: gauss_ints_with_norm'_def)
    from z have "2 ^ n = z * gauss_cnj z"
      by (simp add: self_mult_gauss_cnj)
    also have "2 = -iℤ * p ^ 2"
      by (auto simp: p_def power2_eq_square algebra_simps)
    also have "... ^ n = (-iℤ) ^ n * p ^ (2 * n)"
      by (simp add: algebra_simps power_minus' flip: power_mult)
    finally have "z dvd (-iℤ) ^ n * p ^ (2 * n)"
      by auto
    moreover have "is_unit ((-iℤ) ^ n)"
      by auto
    ultimately have "z dvd p ^ (2 * n)"
      using dvd_mult_unit_iff' by blast
    with ‹prime p› obtain i where i: "i ≤ 2 * n" "z = normalize (p ^
i)"
      using divides_primepow_weak[of p z "2*n"] z by auto
    with z p have "i = n"
      by (simp add: gauss_int_norm_power)
    with i show "z ∈ {normalize ((1 + gauss_i) ^ n)}"
      by (simp add: p_def)
  qed
qed
```

Rational primes p with $p \equiv 3 \pmod 4$ are inert in $\mathbb{Z}[i]$, i.e. they are also prime in $\mathbb{Z}[i]$. Using this, we can show that there is no Gaussian integers with norm p^{2n+1} and exactly one Gaussian integer (up to units) with norm p^{2n}, namely p^n.

```
lemma gauss_ints_with_norm'_prime_power_cong_3:
  assumes "prime p" "[p = 3] (mod 4)"
  shows   "gauss_ints_with_norm' (p ^ n) =
             (if odd n then {} else {of_nat (p ^ (n div 2))})"
  (is "?lhs = ?rhs")
```

```
proof (intro equalityI subsetI)
  fix z assume "z ∈ ?rhs"
  thus "z ∈ ?lhs" using assms
    by (auto split: if_splits simp: gauss_ints_with_norm'_def gauss_int_norm_power

             simp flip: power_mult of_nat_power)
next
  fix z assume "z ∈ ?lhs"
  hence z: "gauss_int_norm z = p ^ n" "normalize z = z"
    by (auto simp: gauss_ints_with_norm'_def)
  from z have "of_nat p ^ n = z * gauss_cnj z"
    by (simp add: self_mult_gauss_cnj)
  hence "z dvd of_nat p ^ n"
    by simp
  then obtain i where i: "i ≤ n" "z = of_nat p ^ i"
    using divides_primepow_weak[of "of_nat p" z n] z assms prime_gauss_int_of_nat[of
p]
    by (auto simp flip: of_nat_power)
  with z have "p ^ (2 * i) = p ^ n"
    by (simp add: gauss_int_norm_power flip: power_mult)
  hence "n = 2 * i"
    using assms prime_power_inj by blast
  with i show "z ∈ ?rhs"
    by auto
qed
```

Any rational prime p with $p \equiv 1 \pmod 4$ factor into two conjugate prime factors q and $\bar{q}$ in $\mathbb{Z}[i]$, just like it was the case for 2. But unlike for 2, where $q = \bar{q} = 1 + i$, we now have $q = \bar{q}$.

Thus a Gaussian integer z has norm p^n iff we have $z\bar{z} = p^n = q^n\bar{q}^n$, which means that z must be of the form $q^i\bar{q}^{n-i}$. This leaves us with $n+1$ choices for i and therefore $n+1$ such Gaussian integers z.

```
lemma gauss_ints_with_norm'_prime_power_cong_1:
  assumes "prime p" "[p = 1] (mod 4)"
  obtains q :: gauss_int where "prime q" "gauss_int_norm q = p"
    "bij_betw (λi. normalize (q ^ i * gauss_cnj q ^ (n - i))) {0..n} (gauss_ints_with_norm'
(p ^ n))"
proof -
  interpret p: noninert_gauss_int_prime p
    by standard fact+
  obtain q q' where q: "prime q" "prime q'" "gauss_int_norm q = p" "gauss_int_norm
q' = p"
                    "prime_factorization (of_nat p) = {#q, q'#}"
    and q'_def: "q' = iℤ * gauss_cnj q"
    using p.prime_factorization by metis
  have neq: "q' ≠ q"
  proof
    assume "q' = q"
```

```
    hence "ReZ q = ImZ q"
      unfolding q'_def gauss_int_eq_iff times_gauss_int.sel gauss_i.sel
gauss_cnj.sel
      by linarith
    hence "even (gauss_int_norm q)"
      by (simp add: gauss_int_norm_def nat_mult_distrib)
    thus False
      using q by (simp add: p.odd_p)
  qed

  have not_q_dvd: "¬q dvd gauss_cnj q"
    using neq q by (metis prime_elem_dvd_mult_iff prime_imp_prime_elem
primes_dvd_imp_eq q'_def)
  have [simp]: "multiplicity q (gauss_cnj q ^ i) = 0" for i
    by (rule not_dvd_imp_multiplicity_0) (use not_q_dvd prime_dvd_power
q(1) in auto)
  have [simp]: "multiplicity q' (gauss_cnj q) = 1"
  proof -
    have "multiplicity q' (gauss_cnj q) = multiplicity q' q'"
      unfolding q'_def by (subst multiplicity_times_unit_right) auto
    thus ?thesis
      using q by simp
  qed

  show ?thesis
  proof (rule that[of q])
    show "bij_betw (λi. normalize (q ^ i * gauss_cnj q ^ (n - i))) {0..n}
(gauss_ints_with_norm' (p ^ n))"
    proof (rule bij_betwI[of _ _ _ "multiplicity q"])
      from q show "(λi. normalize (q ^ i * gauss_cnj q ^ (n - i))) ∈
{0..n} → gauss_ints_with_norm' (p ^ n)"
        by (auto simp: gauss_ints_with_norm'_def gauss_int_norm_mult gauss_int_norm_power
                 simp flip: power_add)
    next
      show "multiplicity q ∈ gauss_ints_with_norm' (p ^ n) → {0..n}"
      proof
        fix z assume z: "z ∈ gauss_ints_with_norm' (p ^ n)"
        from z have [simp]: "z ≠ 0"
          by (auto simp: gauss_ints_with_norm'_def)
        from z have "gauss_int_norm z = p ^ n"
          by (auto simp: gauss_ints_with_norm'_def)
        hence "of_nat (p ^ n) = z * gauss_cnj z"
          by (simp add: self_mult_gauss_cnj)
        also have "of_nat (p ^ n) = ((q * gauss_cnj q) ^ n :: gauss_int)"
          using q by (simp add: self_mult_gauss_cnj)
        also have "... = q ^ n * gauss_cnj (q ^ n)"
          by (simp add: algebra_simps)
        finally have "q ^ n * gauss_cnj (q ^ n) = z * gauss_cnj z" .
        hence "multiplicity q (q ^ n * gauss_cnj (q ^ n)) = multiplicity
```

```
q (z * gauss_cnj z)"
          by (rule arg_cong)
        also have "multiplicity q (q ^ n * gauss_cnj (q ^ n)) = n"
          using q by (simp add: prime_elem_multiplicity_mult_distrib)
        also have "multiplicity q (z * gauss_cnj z) = multiplicity q z
+ multiplicity q (gauss_cnj z)"
          using q by (subst prime_elem_multiplicity_mult_distrib) auto
        finally show "multiplicity q z ∈ {0..n}"
          by simp
      qed
    next
      fix i assume "i ∈ {0..n}"
      thus "multiplicity q (normalize (q ^ i * gauss_cnj q ^ (n - i)))
= i"
        using q by (simp add: prime_elem_multiplicity_mult_distrib)
    next
      fix z assume "z ∈ gauss_ints_with_norm' (p ^ n)"
      hence [simp]: "z ≠ 0" and z: "normalize z = z" "gauss_int_norm
z = p ^ n"
        by (auto simp: gauss_ints_with_norm'_def)
      define i where "i = multiplicity q z"
      have subset: "prime_factors z ⊆ {q, q'}"
      proof -
        have "prime_factors z ⊆ prime_factors (z * gauss_cnj z)"
          by (simp add: dvd_prime_factors)
        also have "z * gauss_cnj z = of_nat p ^ n"
          by (simp add: self_mult_gauss_cnj z)
        also have "prime_factors ... ⊆ prime_factors (of_nat p)"
          by (cases "n = 0") (simp_all add: prime_factors_power)
        also have "... = {q, q'}"
          using q by simp
        finally show ?thesis .
      qed
      have "normalize z = normalize (prod_mset (prime_factorization z))"
        using ‹z ≠ 0› by (rule prod_mset_prime_factorization_weak [symmetric])
      also have "prod_mset (prime_factorization z) = (∏r∈prime_factors
z. r ^ multiplicity r z)"
        by (subst prod_mset_multiplicity, rule prod.cong)
           (auto simp: count_prime_factorization_prime prime_factors_multiplicity)
      also have "(∏r∈prime_factors z. r ^ multiplicity r z) = (∏r∈{q,
q'}. r ^ multiplicity r z)"
        by (rule prod.mono_neutral_left) (use subset q in ‹auto simp:
prime_factors_multiplicity›)
      also have "... = q ^ i * q' ^ multiplicity q' z"
        using q neq by (simp add: i_def)
      finally have z_eq: "z = normalize (q ^ i * q' ^ multiplicity q' z)"
        by (simp add: z(1))
      have "gauss_int_norm z = p ^ (i + multiplicity q' z)"
        by (subst z_eq) (use q in ‹simp_all add: gauss_int_norm_mult gauss_int_norm_power
```

```
power_add›)
      also have "gauss_int_norm z = p ^ n"
        using z by simp
      finally have "n = i + multiplicity q' z"
        using ‹prime p› prime_power_inj by blast
      hence "multiplicity q' z = n - i"
        by linarith
      with z_eq have "z = normalize (q ^ i * q' ^ (n - i))"
        by simp
      also have "... = normalize (iℤ ^ (n - i) * (q ^  i * gauss_cnj q
^ (n - i)))"
        by (simp add: q'_def mult_ac power_mult_distrib)
      also have "... = normalize (q ^ i * gauss_cnj q ^ (n - i))"
        by (rule normalize_mult_unit_left) auto
      finally show "normalize (q ^ i * gauss_cnj q ^ (n - i)) = z" ..
    qed
  qed fact+
qed
```

Combining all of these results, we now know the value of $G(p^n)$ for any rational prime p:

```
theorem card_gauss_ints_with_norm'_prime_power:
  assumes "prime p"
  shows   "card (gauss_ints_with_norm' (p ^ n)) =
             (if [p = 3] (mod 4) ∧ odd n then 0
              else if [p = 1] (mod 4) then n + 1 else 1)"
  using assms
proof (cases p rule: prime_cong_4_nat_cases)
  case 2
  thus ?thesis
    using gauss_ints_with_norm'_2_power[of n]
    by (simp add: cong_def)
next
  case cong_1
  then obtain q where q: "prime q" "gauss_int_norm q = p"
    "bij_betw (λi. normalize (q ^ i * gauss_cnj q ^ (n - i))) {0..n} (gauss_ints_with_norm'
(p ^ n))"
    using gauss_ints_with_norm'_prime_power_cong_1[of p n] assms by blast
  have "card {0..n} = card (gauss_ints_with_norm' (p ^ n))"
    by (rule bij_betw_same_card[OF q(3)])
  thus ?thesis
    using cong_1 by (simp add: cong_def)
next
  case cong_3
  thus ?thesis
    using gauss_ints_with_norm'_prime_power_cong_3[of p n] assms
    by (auto simp: cong_def)
qed
```

This allows us to compute $G(n)$ efficiently given a prime factorisation of n.

1.4.2 The number of divisors in a given congruence class

Next, we introduce a variant of the divisor counting function $\sigma_0(n)$ that will turn out to be useful for computing $r_k(n)$. This function counts the number of divisors d of n with $d \cong i \pmod{m}$ for fixed i and m.

It is not quite a multiplicative function (unless $i = 1$) since it does not necessarily return 1 for $n = 1$ (unless $i = 1$), but it is *somewhat* multiplicative since it does distribute over coprime factors in a more general sense, as we will see below.

```
definition divisor_count_cong :: "nat ⇒ nat ⇒ nat ⇒ nat" where
  "divisor_count_cong i m n = card {d. d dvd n ∧ [d = i] (mod m)}"

lemma divisor_count_cong_0 [simp]:
  assumes "m > 0"
  shows   "divisor_count_cong i m 0 = 0"
proof -
  have "range (λk. m * k + i) ⊆ {d. [d = i] (mod m)}"
    by (auto simp: cong_def)
  moreover have "infinite (range (λk. m * k + i))"
    by (subst finite_image_iff) (use assms in ‹auto intro!: injI›)
  ultimately have "infinite {d. [d = i] (mod m)}"
    using finite_subset by blast
  thus ?thesis
    by (simp add: divisor_count_cong_def)
qed

lemma divisor_count_cong_1:
  "divisor_count_cong i m (Suc 0) = (if [i = 1] (mod m) then 1 else 0)"
proof -
  have "{d. d dvd 1 ∧ [d = i] (mod m)} = (if [i = 1] (mod m) then {1}
else {})"
    by (auto simp: divisor_count_cong_def cong_sym_eq)
  thus ?thesis
    by (simp add: divisor_count_cong_def)
qed
```

The following is an obvious but very helpful lemma that allows us to determine the value of the function on a prime power by determining the number of exponents k such that $p^k \equiv i \pmod{m}$, which is quite easy for concrete i, m, p.

```
lemma divisor_count_cong_prime_power:
  assumes "prime p"
  shows   "divisor_count_cong i m (p ^ n) = card {k∈{0..n}. [p ^ k =
i] (mod m)}"
proof -
```

```
  have "divisor_count_cong i m (p ^ n) = card {d. d dvd p ^ n ∧ [d =
i] (mod m)}"
    by (simp add: divisor_count_cong_def)
  also have bij: "bij_betw (λi. p ^ i) {k∈{0..n}. [p ^ k = i] (mod m)}
{d. d dvd p ^ n ∧ [d = i] (mod m)}"
    by (rule bij_betwI[of _ _ _ "multiplicity p"])
       (use assms in ‹auto simp: dvd_power_iff divides_primepow_nat›)
  have "card {d. d dvd p ^ n ∧ [d = i] (mod m)} = card {k∈{0..n}. [p
^ k = i] (mod m)}"
    using bij_betw_same_card[OF bij] by simp
  finally show ?thesis .
qed
```

The following is a variant of the above lemma for the particular case where p divides the modulus m but not i.

```
lemma divisor_count_cong_prime_power_dvd:
  assumes "p dvd m" "prime p" "¬p dvd i"
  shows   "divisor_count_cong i m (p ^ n) = (if [i = 1] (mod m) then 1
else 0)"
proof -
  have "divisor_count_cong i m (p ^ n) = card {k∈{0..n}. [p ^ k = i]
(mod m)}"
    by (rule divisor_count_cong_prime_power) fact
  also have "{k∈{0..n}. [p ^ k = i] (mod m)} = (if [i = 1] (mod m) then
{0} else {})"
  proof (intro equalityI subsetI)
    fix k assume k: "k ∈ {k∈{0..n}. [p ^ k = i] (mod m)}"
    show "k ∈ (if [i = 1] (mod m) then {0} else {})"
    proof (cases "k = 0")
      case True
      thus ?thesis
        using k by (auto simp: cong_def)
    next
      case False
      have "[p ^ k ≠ i] (mod m)"
        using False assms by (meson bot_nat_0.not_eq_extremum cong_dvd_iff
cong_dvd_modulus_nat dvd_power)
      hence False
        using k by auto
      thus ?thesis ..
    qed
  qed (use assms in ‹auto split: if_splits simp: cong_sym›)
  finally show ?thesis
    by simp
qed
```

Next, we explore the way in which our function distributes over coprime factors.

```
context
```

```
  fixes D :: "nat ⇒ nat ⇒ nat set" and m :: nat
    and F :: "nat ⇒ (nat × nat) set"
    and count :: "nat ⇒ nat ⇒ nat"
  assumes m: "m > 0"
  defines "D ≡ (λi n. {d. d dvd n ∧ [d = i] (mod m)})"
  defines "F ≡ (λi. {(j1,j2). j1 < m ∧ j2 < m ∧ [j1 * j2 = i] (mod m)})"
  defines "count ≡ (λi. divisor_count_cong i m)"
begin

lemma finite_divisors_cong:
  assumes "n > 0"
  shows   "finite (D i n)"
proof (rule finite_subset)
  show "D i n ⊆ {..n}"
    using assms by (auto simp: D_def)
qed auto

lemma bij_betw_divisors_cong_nat:
  assumes "coprime n1 n2"
  shows   "bij_betw (λ(d1, d2). d1 * d2) (⋃(j1,j2)∈F i. D j1 n1 × D
j2 n2) (D i (n1 * n2))"
proof (rule bij_betwI[of _ _ _ "λd. (gcd d n1, gcd d n2)"])
  show "(λ(d1, d2). d1 * d2) ∈ (⋃(j1, j2)∈F i. D j1 n1 × D j2 n2) →
D i (n1 * n2)"
    unfolding F_def D_def
  proof safe
    fix a b j1 j2 :: nat
    assume j12: "j1 < m" "j2 < m" "[j1 * j2 = i] (mod m)"
    assume a: "a dvd n1" "[a = j1] (mod m)" and b: "b dvd n2" "[b = j2]
(mod m)"
    show "a * b dvd n1 * n2"
      using a b by auto
    have "[a * b = j1 * j2] (mod m)"
      by (intro cong_mult a b)
    also have "[j1 * j2 = i] (mod m)"
      by fact
    finally show "[a * b = i] (mod m)" .
  qed
next
  show "(λd. (gcd d n1, gcd d n2)) ∈ D i (n1 * n2) → (⋃(j1, j2)∈F i.
D j1 n1 × D j2 n2)"
  proof safe
    fix d assume "d ∈ D i (n1 * n2)"
    hence d: "d dvd n1 * n2" "[d = i] (mod m)"
      by (auto simp: D_def)
    define d1 d2 where "d1 = gcd d n1" and "d2 = gcd d n2"
    have d_eq: "d = d1 * d2"
      using divisor_coprime_product_decomp[of d n1 n2] d assms
      by (simp_all add: d1_def d2_def)
```

```
    have "[(d1 mod m) * (d2 mod m) = i] (mod m)"
    proof -
      have "[(d1 mod m) * (d2 mod m) = d1 * d2] (mod m)"
        by (intro cong_mult) (auto simp: cong_def)
      also have "[d1 * d2 = i] (mod m)"
        using d_eq d by simp
      finally show ?thesis .
    qed
    hence "(d1 mod m, d2 mod m) ∈ F i"
      using m by (auto simp: F_def)
    moreover have "d1 ∈ D (d1 mod m) n1" "d2 ∈ D (d2 mod m) n2"
      using d_eq by (auto simp: D_def d1_def d2_def)
    ultimately show "(d1, d2) ∈ (⋃(j1, j2)∈F i. D j1 n1 × D j2 n2)"
      by blast
  qed
next
  fix d assume d: "d ∈ (⋃(j1, j2)∈F i. D j1 n1 × D j2 n2)"
  obtain d1 d2 where [simp]: "d = (d1, d2)"
    by (cases d)
  have d12: "d1 dvd n1" "d2 dvd n2"
    using d by (auto simp: D_def)
  have "gcd (d1 * d2) n1 = d1"
    using assms d12
    by (metis coprime_mult_right_iff dvd_mult_div_cancel gcd_mult_left_right_cancel
gcd_nat.absorb_iff1)
  moreover have "gcd (d1 * d2) n2 = d2"
    using assms d12
    by (metis coprime_commute coprime_mult_right_iff dvd_div_mult_self
gcd_mult_left_left_cancel gcd_nat.orderE)
  ultimately show "(gcd (case d of (d1, d2) ⇒ d1 * d2) n1, gcd (case
d of (d1, d2) ⇒ d1 * d2) n2) = d"
    by (auto simp: D_def)
next
  fix d assume "d ∈ D i (n1 * n2)"
  hence "d dvd n1 * n2"
    by (auto simp: D_def)
  hence "gcd d n1 * gcd d n2 = d"
    using assms using divisor_coprime_product_decomp[of d n1 n2] by simp
  thus "(case (gcd d n1, gcd d n2) of (d1, d2) ⇒ d1 * d2) = d"
    using assms by (auto simp: D_def)
qed

lemma divisor_count_cong_mult_coprime:
  assumes "coprime n1 n2"
  shows   "count i (n1 * n2) = (∑(j1,j2)∈F i. count j1 n1 * count j2
n2)"
proof (cases "n1 = 0 ∨ n2 = 0")
  case False
```

```
  hence [simp]: "n1 > 0" "n2 > 0"
    by auto
  have [intro]: "finite (F i)"
    by (rule finite_subset[of _ "{..<m}×{..<m}"]) (auto simp: F_def)
  have D_disjoint: "D j1 n ∩ D j2 n = {}" if "j1 ≠ j2" "j1 < m" "j2 <
m" for j1 j2 n
    using that by (auto simp: D_def cong_def)

  have "count i (n1 * n2) = card (D i (n1 * n2))"
    unfolding count_def divisor_count_cong_def D_def ..
  also have "... = card (⋃(j1,j2)∈F i. D j1 n1 × D j2 n2)"
    by (rule sym, rule bij_betw_same_card, rule bij_betw_divisors_cong_nat)
fact
  also have "... = (∑(j1,j2)∈F i. card (D j1 n1 × D j2 n2))"
  proof (subst card_UN_disjoint)
    show "∀ia∈F i. ∀j∈F i. ia ≠ j ⟶
            (case ia of (j1, j2) ⇒ D j1 n1 × D j2 n2) ∩
            (case j of (j1, j2) ⇒ D j1 n1 × D j2 n2) = {}"
      using D_disjoint[of _ _ n1] D_disjoint[of _ _ n2]
      unfolding F_def by blast
  qed (auto simp: case_prod_unfold intro!: finite_divisors_cong finite_cartesian_product)
  also have "... = (∑(j1,j2)∈F i. count j1 n1 * count j2 n2)"
    by (simp add: count_def D_def divisor_count_cong_def)
  finally show ?thesis .
qed (use m in ‹auto simp: count_def›)

end
```

We now specialise the above relation to the particularly simple (but important) cases of $m = 4$ and $i = 1, 3$.

```
context
  fixes d :: "nat ⇒ nat ⇒ nat"
  defines "d ≡ (λi. divisor_count_cong i 4)"
begin

lemma divisor_count_cong_1_mult_coprime:
  assumes "coprime n1 n2"
  shows   "d 1 (n1 * n2) = d 1 n1 * d 1 n2 + d 3 n1 * d 3 n2"
proof -
  have "{(j1 :: nat, j2). j1 < 4 ∧ j2 < 4 ∧ [j1 * j2 = 1] (mod 4)} =
          Set.filter (λ(j1,j2). (j1 * j2) mod 4 = 1) ({..<4} × {..<4})"
    by (auto simp: Set.filter_def cong_def)
  also have "... = {(1,1), (3,3)}"
    by code_simp
  finally have *: "{(j1 :: nat, j2). j1 < 4 ∧ j2 < 4 ∧ [j1 * j2 = 1] (mod
4)} = {(1,1), (3,3)}" .
  show ?thesis
    unfolding d_def using assms
    by (subst divisor_count_cong_mult_coprime) (use * in simp_all)
```

```
qed

lemma divisor_count_cong_3_mult_coprime:
  assumes "coprime n1 n2"
  shows   "d 3 (n1 * n2) = d 1 n1 * d 3 n2 + d 3 n1 * d 1 n2"
proof -
  have "{(j1 :: nat, j2). j1 < 4 ∧ j2 < 4 ∧ [j1 * j2 = 3] (mod 4)} =
          Set.filter (λ(j1,j2). (j1 * j2) mod 4 = 3) ({..<4} × {..<4})"
    by (auto simp: Set.filter_def cong_def)
  also have "... = {(1,3), (3,1)}"
    by code_simp
  finally have *: "{(j1 :: nat, j2). j1 < 4 ∧ j2 < 4 ∧ [j1 * j2 = 3] (mod
4)} = {(1,3), (3,1)}" .
  show ?thesis
    unfolding d_def using assms
    by (subst divisor_count_cong_mult_coprime) (use * in simp_all)
qed
```

1.4.3 Jacobi's two-square Theorem

We are now ready to prove Jacobi's two-square theorem, namely that the number of ways in which a number $n > 0$ can be written as a sum of two squares of integers is equal to $4(d_1(n) - d_3(n))$, where $d_i(n)$ denotes the number of divisors of n that are congruent i modulo 4.

To that end, we first define the function $f(n)$ as the number of divisors congruent 1 modulo 4 minus the divisors congruent 3 modulo 4. This function $f(n)$ turns out to be multiplicative.

```
context
  fixes f :: "nat ⇒ int"
  defines "f ≡ (λn. int (d 1 n) - int (d 3 n))"
begin

interpretation f: multiplicative_function f
proof
  show "f 0 = 0"
    by (simp add: f_def d_def)
next
  show "f 1 = 1"
    by (simp add: f_def d_def divisor_count_cong_1 cong_def)
next
  fix n1 n2 :: nat
  assume n12: "n1 > 1" "n2 > 1" "coprime n1 n2"
  show "f (n1 * n2) = f n1 * f n2"
    unfolding f_def
    by (simp only: divisor_count_cong_1_mult_coprime divisor_count_cong_3_mult_coprime
n12(3))
       (simp add: algebra_simps)
qed
```

Next, we prove that in fact the number of Gaussian integers (up to units) with norm n is exactly $f(n)$. Since both functions are multiplicative, it suffices to show that this holds for n being a prime power.

Since we have already done all the hard work for $G(p^k)$, it only remains to evaluate $f(p^k)$ in each of the three cases.

```
lemma card_gauss_ints_with_norm': "int (card (gauss_ints_with_norm' n))
= f n"
proof -
  define G where "G = (λn. card (gauss_ints_with_norm' n))"
  have "multiplicative_function G"
    unfolding G_def ..

  have "int (G n) = f n"
  proof (rule multiplicative_function_eqI)
    show "multiplicative_function (λn. int (G n))"
      unfolding G_def by (rule multiplicative_function_of_natI) standard
  next
    show "multiplicative_function f" ..
  next
    fix p k :: nat
    assume p: "prime p" and k: "k > 0"
    thus "int (G (p ^ k)) = f (p ^ k)"
    proof (cases p rule: prime_cong_4_nat_cases)
      case [simp]: 2
      have "f (2 ^ k) = 1"
        by (simp add: f_def d_def divisor_count_cong_prime_power_dvd cong_def)
      thus ?thesis
        by (simp add: G_def gauss_ints_with_norm'_2_power)
    next
      case cong_1
      have mod: "(p ^ i) mod 4 = 1" for i
      proof -
        have "[p ^ i = 1 ^ i] (mod 4)"
          by (intro cong_pow cong_1)
        thus ?thesis
          by (simp add: cong_def)
      qed

      have "d 1 (p ^ k) = Suc k"
        using p by (simp add: d_def divisor_count_cong_prime_power cong_def
mod)
      moreover have "d 3 (p ^ k) = 0"
        using p by (simp add: d_def divisor_count_cong_prime_power cong_def
mod)
      ultimately have "f (p ^ k) = Suc k"
        by (simp add: f_def)
      moreover have "G (p ^ k) = Suc k"
        using cong_1 p by (simp add: G_def cong_def card_gauss_ints_with_norm'_prime_power)
```

```
      ultimately show ?thesis
        by simp
    next
      case cong_3
      have mod: "(p ^ i) mod 4 = (if even i then 1 else 3)" for i
      proof -
        have "[int (p ^ i) = int (3 ^ i)] (mod (int 4))"
          unfolding cong_int_iff using cong_3 by (intro cong_pow) auto
        hence "[int p ^ i = 3 ^ i] (mod 4)"
          by simp
        also have "[3 ^ i = (-1 :: int) ^ i] (mod 4)"
          by (intro cong_pow) (auto simp: cong_def)
        also have "(-1) ^ i = (if even i then 1 else -1 :: int)"
          by (auto simp: uminus_power_if)
        also have "[... = (if even i then 1 else 3 :: int)] (mod 4)"
          by (auto simp: cong_def)
        finally have "[int (p ^ i) = int (if even i then 1 else 3)] (mod
(int 4))"
          by (auto split: if_splits)
        hence "[p ^ i = (if even i then 1 else 3)] (mod 4)"
          unfolding cong_int_iff .
        thus ?thesis
          by (auto simp: cong_def)
      qed

      have "d 1 (p ^ k) = k div 2 + 1"
      proof -
        have "d 1 (p ^ k) = card {i. i ≤ k ∧ (p ^ i) mod 4 = 1}" us-
ing p
          by (simp add: d_def divisor_count_cong_prime_power cong_def)
        also have "{i. i ≤ k ∧ (p ^ i) mod 4 = 1} = {i. i ≤ k ∧ even
i}"
          by (auto simp: mod)
        also have "bij_betw (λi. i div 2) {i. i ≤ k ∧ even i} {0..k div
2}"
          by (rule bij_betwI[of _ _ _ "λi. i * 2"]) auto
        hence "card {i. i ≤ k ∧ even i} = card {0..k div 2}"
          by (rule bij_betw_same_card)
        finally show ?thesis
          by simp
      qed
      moreover have "d 3 (p ^ k) = (k+1) div 2"
      proof -
        have "d 3 (p ^ k) = card {i. i ≤ k ∧ (p ^ i) mod 4 = 3}" us-
ing p
          by (simp add: d_def divisor_count_cong_prime_power cong_def)
        also have "{i. i ≤ k ∧ (p ^ i) mod 4 = 3} = {i. i ≤ k ∧ odd
i}"
          by (auto simp: mod)
```

```
      also have "bij_betw (λi. (i+1) div 2) {i. i ≤ k ∧ odd i} {1..(k+1)
div 2}"
        by (rule bij_betwI[of _ _ _ "λi. i * 2 - 1"])  (auto elim!:
oddE)
      hence "card {i. i ≤ k ∧ odd i} = card {1..(k+1) div 2}"
        by (rule bij_betw_same_card)
      finally show ?thesis
        by simp
    qed
    ultimately have "f (p ^ k) = (if even k then 1 else 0)"
      by (auto simp: f_def elim!: evenE oddE)
    moreover have "G (p ^ k) = (if even k then 1 else 0)"
      using cong_3 p by (simp add: G_def cong_def card_gauss_ints_with_norm'_prime_power)
    ultimately show ?thesis
      by simp
  qed
qed
thus ?thesis
  by (simp add: G_def)
qed

corollary card_gauss_ints_with_norm:
  assumes "n > 0"
  shows   "int (card (gauss_ints_with_norm n)) = 4 * f n"
  using card_gauss_ints_with_norm'[of n] assms
  by (simp add: card_gauss_ints_with_norm_conv_norm')

end
end
```

We get the "Sum of Two Squares" Theorem as a simply corollary.

```
theorem sum_of_two_squares_eq:
  assumes "n > 0"
  shows   "count_sos 2 n = 4 * (int (divisor_count_cong 1 4 n) - int (divisor_count_cong
3 4 n))"
  unfolding card_gauss_ints_with_norm_conv_count_sos [symmetric]
  using card_gauss_ints_with_norm[OF assms] .
```

The number of decompositions into two squares of positive numbers can be computed similarly, but we need a "correction term" for the case that n itself is a square.

```
corollary count_pos_sos_2_eq:
  assumes "n > 0"
  shows   "count_pos_sos 2 n =
             (int (divisor_count_cong 1 4 n) - int (divisor_count_cong
3 4 n) -
             (if is_square n then 1 else 0))"
proof -
```

```
  have "int (count_sos 2 n) = 4 * (count_pos_sos 2 n + (if is_square n
then 1 else 0))"
    using assms by (auto simp: eval_nat_numeral count_sos_conv_count_pos_sos
count_pos_sos_1)
  also have "int (count_sos 2 n) = 4 * (int (divisor_count_cong 1 4 n)
- int (divisor_count_cong 3 4 n))"
    by (rule sum_of_two_squares_eq) fact
  finally show ?thesis
    by auto
qed
```

As a simple corollary, it follows that if $p = 2$ (for any k) or $p \equiv 3 \pmod 4$ (for even k), the numbers n and $p^k n$ have the same number of decompositions into two squares.

```
corollary count_sos_times_prime_power:
  assumes "p = 2 ∨ (prime p ∧ [p = 3] (mod 4) ∧ even k)"
  shows   "count_sos 2 (p ^ k * n) = count_sos 2 n"
proof (cases "n = 0")
  case False
  define i where "i = multiplicity p n"
  define m where "m = n div p ^ i"
  have 1: "n = p ^ i * m"
    using False unfolding i_def m_def by (simp add: multiplicity_dvd)
  have "p > 0" "p ≠ Suc 0"
    using assms by (auto intro: Nat.gr0I)
  hence 2: "¬p dvd m"
    using False multiplicity_decompose[of n p] unfolding m_def i_def by
auto
  have "gauss_ints_with_norm' (p ^ k * n) = gauss_ints_with_norm' (p ^
(i + k) * m)"
    by (simp add: 1 power_add mult_ac)
  also have "card ... = card (gauss_ints_with_norm' (p ^ (i + k))) * card
(gauss_ints_with_norm' m)"
    by (rule gauss_ints_with_norm'.mult_coprime) (use 2 assms in ‹auto
simp: prime_imp_coprime›)
  also have "card (gauss_ints_with_norm' (p ^ (i + k))) = card (gauss_ints_with_norm'
(p ^ i))"
    by (subst (1 2) card_gauss_ints_with_norm'_prime_power) (use assms
in ‹auto simp: cong_def›)
  also have "... * card (gauss_ints_with_norm' m) = card (gauss_ints_with_norm'
(p ^ i * m))"
    by (rule gauss_ints_with_norm'.mult_coprime [symmetric])
       (use 2 assms in ‹auto simp: prime_imp_coprime›)
  finally show ?thesis using False ‹p > 0›
    by (simp add: 1 card_gauss_ints_with_norm_conv_norm'
             flip: card_gauss_ints_with_norm_conv_count_sos)
qed auto

corollary count_sos_2_double: "count_sos 2 (2 * n) = count_sos 2 n"
```

```
  using count_sos_times_prime_power[of 2 1 n] by simp
```

And as yet another corollary, the following well-known fact follows: a positive integer n can be written as a sum of two squares iff all the prime factors congruent 3 modulo 4 have odd multiplicity.

```
corollary count_sos_2_eq_0_iff:
  "count_sos 2 n = 0 ⟷ (∃p. prime p ∧ [p = 3] (mod 4) ∧ odd (multiplicity
p n))"
proof (cases "n = 0")
  case False
  define G where "G = (λn. card (gauss_ints_with_norm' n))"
  define a where "a = (λp. multiplicity p n)"
  have "count_sos 2 n = 4 * G n"
    using False by (simp add: card_gauss_ints_with_norm_conv_norm' G_def
                         flip: card_gauss_ints_with_norm_conv_count_sos)
  also have "... = 0 ⟷ G n = 0"
    by simp
  also have "G n = (∏p∈prime_factors n. G (p ^ a p))"
    using False gauss_ints_with_norm'.prod_prime_factors[of n] by (simp
add: G_def a_def)
  also have "... = 0 ⟷ (∃p∈prime_factors n. G (p ^ a p) = 0)"
    by simp
  also have "... ⟷ (∃p∈prime_factors n. [p = 3] (mod 4) ∧ odd (a p))"
    by (intro bex_cong refl)
       (auto simp: prime_factors_multiplicity G_def card_gauss_ints_with_norm'_prime_power)
  also have "... ⟷ (∃p. prime p ∧ [p = 3] (mod 4) ∧ odd (a p))" un-
folding Bex_def
    by (intro arg_cong[of _ _ Ex]) (auto simp: prime_factors_multiplicity
fun_eq_iff a_def odd_pos)
  finally show ?thesis unfolding a_def .
qed auto

end
```

References

[1] E. Grosswald. *Representations of Integers as Sums of Squares.* Springer New York, 2012.