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

Manuel Eberl

November 30, 2024

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"
  "HOL-Library.FuncSet"
  "Gaussian_Integers.Gaussian_Integers"
  "Dirichlet_Series.Multiplicative_Function"
  "List-Index.List_Index"
begin

1.1 Auxiliary material

lemma *map_index_cong:*
  assumes *"length xs = length ys" "⋀i. i < length xs ⟹ f i (xs ! i) = g i (ys ! i)"*
  shows *"map_index f xs = map_index g ys"*
  ⟨*proof*⟩

lemma *map_index_idI: "(⋀i. f i (xs ! i) = xs ! i) ⟹ map_index f xs = xs"*
  ⟨*proof*⟩

lemma *map_index_transfer [transfer_rule]:*
  "rel_fun (rel_fun (=) (rel_fun R1 R2)) (rel_fun (list_all2 R1) (list_all2 R2))
     map_index map_index"
  ⟨*proof*⟩

lemma *map_index_Cons: "map_index f (x # xs) = f 0 x # map_index (λi x. f (Suc i) x) xs"*
  ⟨*proof*⟩

lemma *map_index_rev: "map_index f (rev xs) = rev (map_index (λi. f (length xs - i - 1)) xs)"*
  ⟨*proof*⟩

lemma *map_conv_map_index: "map f xs = map_index (λi x. f x) xs"*
  ⟨*proof*⟩

lemma *map_index_map_index: "map_index f (map_index g xs) = map_index (λi x. f i (g i x)) xs"*
  ⟨*proof*⟩

lemma *map_index_replicate [simp]: "map_index f (replicate n x) = map (λi. f i x) [0..<n]"*
  ⟨*proof*⟩

lemma *zip_map_index:*
  "zip (map_index f xs) (map_index g ys) = map_index (λi. map_prod (f i) (g i)) (zip xs ys)"
  ⟨*proof*⟩

lemma *map_index_conv_fold:*
  "map_index f xs = rev (snd (fold (λx (i,ys). (i+1, f i x # ys)) xs (0, [])))"
⟨*proof*⟩

lemma *map_index_code_conv_foldr:*
  "map_index f xs = snd (foldr (λx (i,ys). (i-1, f i x # ys)) xs (length xs - 1, []))"
⟨*proof*⟩

lemma *sum_list_of_nat: "sum_list (map of_nat xs) = of_nat (sum_list xs)"*
  ⟨*proof*⟩

lemma *sum_list_of_int: "sum_list (map of_int xs) = of_int (sum_list xs)"*
  ⟨*proof*⟩

lemma *sum_list_of_real: "sum_list (map of_real xs) = of_real (sum_list xs)"*
  ⟨*proof*⟩

lemma *prime_cong_4_nat_cases [consumes 1, case_names 2 cong_1 cong_3]:*
  assumes *"prime (p :: nat)"*
  obtains *"p = 2" | "[p = 1] (mod 4)" | "[p = 3] (mod 4)"*
⟨*proof*⟩

lemma *member_le_sum_list:*
  fixes *x :: "'a :: ordered_comm_monoid_add"*
  assumes *"x ∈ set xs" "⋀x. x ∈ set xs ⟹ x ≥ 0"*
  shows *"x ≤ sum_list xs"*
  ⟨*proof*⟩

lemma *is_square_conv_sqrt: "is_square n ⟷ Discrete.sqrt n ^ 2 = n"*
  ⟨*proof*⟩

lemma *sum_replicate_mset_count_eq: "(∑x∈set_mset X. replicate_mset (count X x) x) = X"*
  ⟨*proof*⟩

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⟩

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⟩

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"
  ⟨proof⟩



lemma gauss_int_norm_power: "gauss_int_norm (x ^ n) = gauss_int_norm
x ^ n"
  ⟨proof⟩

lemma gcd_gauss_cnj: "gcd (gauss_cnj x) (gauss_cnj y) = normalize (gauss_cnj
(gcd x y))"
⟨proof⟩

lemma gcd_gauss_cnj_left: "gcd (gauss_cnj x) y = normalize (gauss_cnj
(gcd x (gauss_cnj y)))"
  ⟨proof⟩

lemma gcd_gauss_cnj_right: "gcd x (gauss_cnj y) = normalize (gauss_cnj
(gcd (gauss_cnj x) y))"
  ⟨proof⟩
```

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⟩

lemma sos_decomps_0_right [simp]: "sos_decomps k 0 = {replicate k 0}"
⟨proof⟩

lemma sos_decomps_0: "sos_decomps 0 n = (if n = 0 then {[]} else {})"
  ⟨proof⟩

lemma sos_decomps_1:
  "sos_decomps (Suc 0) n = (if is_square n then {[Discrete.sqrt n], [-Discrete.sqrt
n]} else {})"
  (is "?lhs = ?rhs")
⟨proof⟩

lemma bij_betw_sos_decomps_2: "bij_betw (λ(x,y). [x,y]) {(i,j). i^2 +
j^2 = int n} (sos_decomps 2 n)"
  ⟨proof⟩

lemma sos_decomps_Suc:
  "sos_decomps (Suc k) n =
     (#) 0 ` sos_decomps k n ∪
     (⋃i∈{1..Discrete.sqrt n}. ⋃xs∈sos_decomps k (n - i ^ 2). {int i
# xs, (-int i) # xs})"
  (is "?A = ?B ∪ ?C")
⟨proof⟩

lemma count_sos_0_right [simp]: "count_sos k 0 = 1"
  ⟨proof⟩

lemma count_sos_0 [simp]: "n > 0 ⟹ count_sos 0 n = 0"
  ⟨proof⟩

lemma count_sos_1: "n > 0 ⟹ count_sos (Suc 0) n = (if is_square n then
2 else 0)"
  ⟨proof⟩

lemma count_sos_2: "count_sos 2 n = card {(i,j). i^2 + j^2 = int n}"
  ⟨proof⟩
```

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..Discrete.sqrt n.
count_sos k (n - i ^ 2))"
```

⟨*proof*⟩

```
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 Discrete.sqrt n ^ 2 = n then 2 else 0)
     else count_sos (k-1) n + 2 * (∑i=1..Discrete.sqrt n. count_sos (k-1)
(n-i^2)))"
  ⟨proof⟩
```

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⟩

lemma pos_sos_decomps_0_right: "pos_sos_decomps k 0 = (if k = 0 then
{[]} else {})"
⟨proof⟩

lemma pos_sos_decomps_0: "pos_sos_decomps 0 n = (if n = 0 then {[]} else
{})"
  ⟨proof⟩

lemma pos_sos_decomps_1:
  "pos_sos_decomps (Suc 0) n = (if is_square n ∧ n > 0 then {[Discrete.sqrt
n]} else {})"
  (is "?lhs = ?rhs")
⟨proof⟩

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)"
  ⟨proof⟩
```

```
lemma pos_sos_decomps_Suc:
  "pos_sos_decomps (Suc k) n =
     (⋃i∈{1..Discrete.sqrt n}. ((#) i) ` pos_sos_decomps k (n - i ^ 2))"
  (is "?A = ?B")
⟨proof⟩

lemma count_pos_sos_0_right: "count_pos_sos k 0 = (if k = 0 then 1 else
0)"
  ⟨proof⟩

lemma count_pos_sos_0: " count_pos_sos 0 n = (if n = 0 then 1 else 0)"
  ⟨proof⟩

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"
  ⟨proof⟩

lemma count_pos_sos_1: "count_pos_sos (Suc 0) n = (if is_square n ∧
n > 0 then 1 else 0)"
  ⟨proof⟩

lemma count_pos_sos_2: "count_pos_sos 2 n = card {(i,j). i² + j² = n
∧ i > 0 ∧ j > 0}"
  ⟨proof⟩
```

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

```
lemma count_pos_sos_Suc:
  "count_pos_sos (Suc k) n = (∑i=1..Discrete.sqrt n. count_pos_sos k
(n - i ^ 2))"
⟨proof⟩

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 Discrete.sqrt n ^ 2 = n then 1 else 0)
     else (∑i=1..Discrete.sqrt n. count_pos_sos (k-1) (n-i^2)))"
  ⟨proof⟩
```

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⟩
```

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⟩

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"
  ⟨proof⟩
```

```
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)"
  ⟨proof⟩
```

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}"
  ⟨proof⟩
```

```
lemma card_gauss_ints_with_norm_conv_count_sos: "card (gauss_ints_with_norm
n) = count_sos 2 n"
  ⟨proof⟩
```

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 * i_ℤ) ` ({0..int
n} × {0..int n})"
⟨proof⟩

lemma finite_gauss_ints_with_norm' [simp, intro]: "finite (gauss_ints_with_norm'
n)"
  ⟨proof⟩

lemma gauss_ints_with_norm'_0 [simp]: "gauss_ints_with_norm' 0 = {}"
  ⟨proof⟩

lemma gauss_ints_with_norm'_1 [simp]: "gauss_ints_with_norm' (Suc 0)
= {1}"
  ⟨proof⟩

lemma unit_factor_eq_1_iff: "unit_factor x = 1 ⟷ normalize x = x ∧
x ≠ 0"
  ⟨proof⟩

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)"
  ⟨proof⟩

lemma finite_gauss_ints_with_norm [simp, intro]: "finite (gauss_ints_with_norm
n)"
⟨proof⟩

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⟩
```

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))"
  ⟨proof⟩
```

interpretation *gauss_ints_with_norm': multiplicative_function "λn. card (gauss_ints_with_norm' n)"*
⟨*proof*⟩

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"*
  ⟨*proof*⟩

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 + i$_\mathbb{Z}$) ^ n)}"*
⟨*proof*⟩

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⟩
```

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⟩
```

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)"
  ⟨proof⟩
```

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⟩

lemma divisor_count_cong_1:
  "divisor_count_cong i m (Suc 0) = (if [i = 1] (mod m) then 1 else 0)"
⟨proof⟩
```

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*⟩

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⟩
```

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⟩

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⟩

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⟩

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⟩

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⟩
```

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⟩
```

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⟩

corollary card_gauss_ints_with_norm:
  assumes "n > 0"
  shows   "int (card (gauss_ints_with_norm n)) = 4 * f n"
  ⟨proof⟩

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))"
  ⟨proof⟩
```

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⟩
```

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⟩

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

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⟩

end
```

References

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