The Incomplete Gamma Function

Manuel Eberl

July 8, 2026

Abstract

This entry provides two special functions, the lower and upper incomplete gamma functions. Similarly to the "complete" gamma function $\Gamma(s)$, which is defined as $\Gamma(s) = \int_0^\infty t^{s-1}e^{-t}\,\mathrm{d}t$ for $\mathrm{Re}(s) > 0$ and by analytic continuation elsewhere, these are defined as $\gamma(s,z) = \int_0^z t^{s-1}e^{-t}\,\mathrm{d}t$ and $\Gamma(s,z) = \int_z^\infty t^{s-1}e^{-t}\,\mathrm{d}t$, respectively, for $\mathrm{Re}(s) > 0$ and analytically continued to the entire complex plane.

$\gamma(s,z)$ is constructed using the regularised hypergeometric series and $\Gamma(s,z)$ via its contour integral representation. Various results are provided, including:

- holomorphicity, continuity, limits, and derivatives
- series and integral representations
- shift identities such as $\Gamma(s+1,z) = s\Gamma(s,z) + z^s e^{-z}$
- the identity $\gamma(s,z) + \Gamma(s,z) = \Gamma(s)$
- the fact that $\Gamma(s,z) \to \Gamma(s)$ as $z \to 0$ within a certain region
- closed forms for $\Gamma(n,z)$ and $\gamma(n,z)$, where n is a positive integer
- the connection to the error function via $\Gamma(\frac{1}{2},z) = \sqrt{\pi}\cdot\mathrm{erf}(\sqrt{z})$

Contents

1 A "safe" power operator for real and complex numbers

theory *Safe_Power*
  imports *"HOL-Complex_Analysis.Complex_Analysis"*
begin

Isabelle/HOL currently has three different power operators:

- *(^)*, denoted x^n, where *n* is a natural number and *x* is an element of a multiplicatively written monoid (*monoid_mult*).
- *(powi)*, denoted *x powi n*, where *n* is an integer and *x* is an element of a multiplicatively written monoid with some kind of inverse operation (typically a *division_ring*).
- *(powr)*, denoted *x powr y*, where *x* and *y* are complete normed real algebra with a 1 and some kind of natural logarithm. In practice, the only reasonable examples of such a structure are probably *real* and *complex*. It is defined as *x powr y = exp (ln x * y)*, except when *x = 0*, in which case it returns *0* by convention.

All three of these operators are required, since none of them is a generalisation of another. For example, the *(^)* operator is the most restrictive in what the right argument can be, but it is the most general in terms of what the left argument can be.

For the most part, the three operators agree in all the cases where they are simultaneously defined, but there are some caveats.

First of all, note that different conventions apply for 0^0: We have x^n *= 1* and *x powr n = 1* for any *x*, including for *x = 0*, whereas *0 powr y = 0* for any *y*, including *y = 0*.

Second, for negative real *x*, the *(powr)* operator inherits somewhat strange behaviour from the real-valued *ln* operator, which in Isabelle/HOL is defined symmetrically, i.e. *ln (- x) = ln x*, so we also have *(- x) powr y = x powr y* for real *x* and *y*. This means that while $(-1)^3 = -1$ as expected, we have *(- 1) powr 3 = 1*. It is therefore better to consider *x powr y* to be undefined for negative real *x*.

In the following, we will define a "safe" version of the *(powr)* operator that coincides with *(^)* and *(powi)* whenever applicable.

First, we define the inverse of the canonical ring homomorphism $\mathbb{Z} \to R$ using the choice operator, i.e. for any $x \in \mathbb{Z}$, the operator *the_int* gives us an integer n such that $x = n = \underbrace{1 + \ldots + 1}_{n \text{ times}}$. For $x \notin \mathbb{Z}$, the operator's behaviour is unspecified.

definition *the_int :: "'a :: ring_char_0 ⇒ int"* **where**
"the_int x = (THE n. x = of_int n)"

lemma *the_int_of_int [simp]: "the_int (of_int n :: 'a :: ring_char_0) = n"*
unfolding *the_int_def* **using** *theI'[of "λm. of_int n = (of_int m :: 'a)"]* **by** *simp*

lemma *the_int_eqI: "of_int n = x ⟹ the_int x = n"*
by *(metis the_int_of_int)*

lemma *the_int_0 [simp]: "the_int 0 = 0"*
using *the_int_of_int[of 0]* **by** *(simp del: the_int_of_int)*

lemma *the_int_1 [simp]: "the_int 1 = 1"*
using *the_int_of_int[of 1]* **by** *(simp del: the_int_of_int)*

lemma *the_int_of_nat [simp]: "the_int (of_nat n) = int n"*
using *the_int_of_int[of "int n"]* **by** *(simp del: the_int_of_int)*

lemma *the_int_numeral [simp]: "the_int (numeral n) = numeral n"*
using *the_int_of_int[of "numeral n"]* **by** *(simp del: the_int_of_int)*

lemma *the_int_uminus [simp]: "x ∈ ℤ ⟹ the_int (-x) = -the_int x"*
by *(metis Ints_cases of_int_minus the_int_of_int)*

lemma *of_int_the_int: "x ∈ ℤ ⟹ of_int (the_int x) = x"*
by *(elim Ints_cases) auto*

lemma *the_int_add: "x ∈ ℤ ⟹ y ∈ ℤ ⟹ the_int (x + y) = the_int x + the_int y"*
by *(rule the_int_eqI) (auto simp: of_int_the_int)*

lemma *the_int_diff: "x ∈ ℤ ⟹ y ∈ ℤ ⟹ the_int (x - y) = the_int x - the_int y"*
by *(rule the_int_eqI) (auto simp: of_int_the_int)*

lemma *the_int_mult: "x ∈ ℤ ⟹ y ∈ ℤ ⟹ the_int (x * y) = the_int x * the_int y"*
by *(rule the_int_eqI) (auto simp: of_int_the_int)*

lemma *the_int_power: "x ∈ ℤ ⟹ the_int (x ^ n) = the_int x ^ n"*
by *(rule the_int_eqI) (auto simp: of_int_the_int)*

Now we simply define the *powr'* operator as a version of the *(powr)* operator that falls back to the *(powi)* operator whenever possible.

definition *powr' :: "'a :: {division_ring, ln} ⇒ 'a ⇒ 'a"* (**infixr** *‹powr''› 80)* **where**
"x powr' y = (if y ∈ ℤ then x powi (the_int y) else x powr y)"

```
lemma powr'_powr: "y ∉ ℤ ⟹ x powr' y = x powr y"
  by (simp add: powr'_def)

lemma powr'_0_left [simp]: "y ≠ 0 ⟹ 0 powr' y = 0"
  by (auto simp: powr'_def elim!: Ints_cases)

lemma powr'_0_left_if: "0 powr' y = (if y = 0 then 1 else 0)"
  by (auto simp: powr'_def elim!: Ints_cases)

lemma powr'_of_int [simp]: "x powr' of_int n = x powi n"
  by (simp add: powr'_def)

lemma powr'_of_nat [simp]: "x powr' of_nat n = x ^ n"
  by (simp add: powr'_def)

lemma powr'_0 [simp]: "x powr' 0 = 1"
  by (simp add: powr'_def)

lemma powr'_1 [simp]: "x powr' 1 = x"
  by (simp add: powr'_def)

lemma powr'_numeral [simp]: "x powr' numeral n = x ^ numeral n"
  by (simp add: powr'_def)
```

If x is positive or if x is non-negative and y is positive, the safe power operator and the normal power operator agree..

```
lemma powr'_real: "x ≥ 0 ⟹ x ≠ 0 ∨ y > 0 ⟹ x powr' y = x powr (y
:: real)"
  by (auto simp: powr'_def powr_real_of_int' elim!: Ints_cases)

lemma powr'_real_pos: "x > 0 ⟹ x powr' y = x powr (y :: real)"
  by (auto simp: powr'_def powr_real_of_int' elim!: Ints_cases)
```

For complex inputs, the two operators always agree except in the case of 0^0.

```
lemma powr'_complex: "x ≠ 0 ∨ y ≠ 0 ⟹ x powr' y = x powr (y :: complex)"
  by (auto simp: powr'_def complex_powr_of_int elim!: Ints_cases)

lemma powr'_complex_of_real:
  "x ≥ 0 ∨ y ∈ ℤ ⟹ complex_of_real x powr' of_real y = (of_real (x
powr' y))"
  by (auto simp: powr'_def powr_Reals_eq elim!: Ints_cases)

lemma powr'_Reals_eq: "⟦x ∈ ℝ; y ∈ ℝ; Re x ≥ 0⟧ ⟹ x powr' y = of_real
(Re x powr' Re y)"
  by (cases "x = 0"; cases "y = 0")
     (auto elim!: Reals_cases simp: powr'_complex powr'_real powr_Reals_eq)
```

For this reason, the *(powr')* operator is holomorphic in both inputs except

for a branch cut along the non-positive reals.

```
lemma holomorphic_on_powr' [holomorphic_intros]:
  assumes "f holomorphic_on A" "g holomorphic_on A" "⋀x. x ∈ A ⟹ f
x ∉ ℝ≤0"
  shows   "(λx. f x powr' g x) holomorphic_on A"
proof -
  have "(λx. f x powr g x) holomorphic_on A"
    by (intro holomorphic_intros assms)
  also have "(λx. f x powr g x) holomorphic_on A ⟷ (λx. f x powr' g
x) holomorphic_on A"
    by (intro holomorphic_cong refl) (subst powr'_complex, auto dest:
assms(3))
  finally show ?thesis .
qed

lemma analytic_on_powr' [analytic_intros]:
  assumes "f analytic_on A" "g analytic_on A" "⋀x. x ∈ A ⟹ f x ∉
ℝ≤0"
  shows   "(λx. f x powr' g x) analytic_on A"
proof -
  from assms(1) obtain B where B: "open B" "A ⊆ B" "f holomorphic_on
B"
    using analytic_on_holomorphic by blast
  from assms(2) obtain C where C: "open C" "A ⊆ C" "g holomorphic_on
C"
    using analytic_on_holomorphic by blast
  note [holomorphic_intros] = holomorphic_on_subset[OF B(3)] holomorphic_on_subset[OF
C(3)]
  have "(λx. f x powr' g x) holomorphic_on ((B ∩ C) ∩ f -` (-ℝ≤0))"
    by (intro holomorphic_intros) auto
  moreover have "open ((B ∩ C) ∩ f -` (-ℝ≤0))" using B(1) C(1)
    by (intro continuous_open_preimage holomorphic_on_imp_continuous_on
holomorphic_intros) auto
  moreover have "A ⊆ (B ∩ C) ∩ f -` (-ℝ≤0)"
    using assms(3) B C by auto
  ultimately show ?thesis
    using analytic_on_holomorphic by blast
qed


lemma has_field_derivative_powr_complex [derivative_intros]:
  assumes "(f has_field_derivative f') (at x within A)"
  assumes "(g has_field_derivative g') (at x within A)"
  assumes "f x ∉ (ℝ≤0 :: complex set)"
  shows   "((λx. f x powr g x) has_field_derivative
             (f x powr g x * (f' / f x * g x + g' * ln (f x)))) (at x
within A)"
proof -
  have "((λx. exp (ln (f x) * g x)) has_field_derivative
```

```
          (exp (ln (f x) * g x) * (f' / f x * g x + g' * ln (f x))) (at
x within A)"
    by (auto intro!: derivative_eq_intros assms simp: field_simps)
  also have "(exp (ln (f x) * g x) * (f' / f x * g x + g' * ln (f x)))
=
             (f x powr g x * (f' / f x * g x + g' * ln (f x)))"
    by (use assms(3) in ‹auto simp: powr_def mult_ac›)
  also have "((λx. exp (ln (f x) * g x)) has_field_derivative ...) (at
x within A) ⟷ ?thesis"
  proof (intro has_field_derivative_cong_eventually)
    have "eventually (λz. z ∈ -{0}) (nhds (f x))"
      by (intro eventually_nhds_in_open) (use assms(3) in auto)
    moreover have "continuous (at x within A) f"
      using assms(1) DERIV_continuous by blast
    hence "filterlim f (nhds (f x)) (at x within A)"
      by (simp add: continuous_within)
    ultimately have "eventually (λx. f x ∈ -{0}) (at x within A)"
      by (rule eventually_compose_filterlim)
    thus "∀F x in at x within A. exp (ln (f x) * g x) = f x powr g x"
      by eventually_elim (auto simp: powr_def mult_ac)
  next
    show "exp (Ln (f x) * g x) = f x powr g x"
      by (use assms(3) in ‹auto simp: powr_def mult_ac›)
  qed
  finally show ?thesis .
qed


lemma has_field_derivative_powr'_complex [derivative_intros]:
  assumes "(f has_field_derivative f') (at x within A)"
  assumes "(g has_field_derivative g') (at x within A)"
  assumes "f x ∉ (ℝ≤0 :: complex set)"
  shows   "((λx. f x powr' g x) has_field_derivative
             (f x powr' g x * (f' / f x * g x + g' * ln (f x)))) (at x
within A)"
proof -
  have "((λx. exp (ln (f x) * g x)) has_field_derivative
          (exp (ln (f x) * g x) * (f' / f x * g x + g' * ln (f x)))) (at
x within A)"
    by (auto intro!: derivative_eq_intros assms simp: field_simps)
  also have "(exp (ln (f x) * g x) * (f' / f x * g x + g' * ln (f x)))
=
             (f x powr' g x * (f' / f x * g x + g' * ln (f x)))"
    by (subst powr'_complex) (use assms(3) in ‹auto simp: powr_def mult_ac›)
  also have "((λx. exp (ln (f x) * g x)) has_field_derivative ...) (at
x within A) ⟷ ?thesis"
  proof (intro has_field_derivative_cong_eventually)
    have "eventually (λz. z ∈ -{0}) (nhds (f x))"
      by (intro eventually_nhds_in_open) (use assms(3) in auto)
```

```
    moreover have "continuous (at x within A) f"
      using assms(1) DERIV_continuous by blast
    hence "filterlim f (nhds (f x)) (at x within A)"
      by (simp add: continuous_within)
    ultimately have "eventually (λx. f x ∈ -{0}) (at x within A)"
      by (rule eventually_compose_filterlim)
    thus "∀F x in at x within A. exp (ln (f x) * g x) = f x powr' g x"
      by eventually_elim (auto simp: powr'_complex powr_def mult_ac)
  next
    show "exp (Ln (f x) * g x) = f x powr' g x"
      by (subst powr'_complex) (use assms(3) in ‹auto simp: powr_def mult_ac›)
  qed
  finally show ?thesis .
qed

lemma has_field_derivative_powr'_Ints:
  assumes "(f has_field_derivative f') (at x within A)"
  assumes "c ∈ (ℤ :: 'a :: {real_normed_field, ln} set)" "c ∈ ℕ ∨ f
x ≠ 0"
  shows   "((λx. f x powr' c) has_field_derivative (c * f x powr' (c-1)
* f')) (at x within A)"
proof -
  from assms(2) obtain n where c: "c = of_int n"
    by (elim Ints_cases)
  have "n ≥ 0 ∨ f x ≠ 0"
    using assms(3) c
    by (metis Nats_cases dual_order.refl int_nat_eq nat_int of_int_eq_iff
of_int_of_nat_eq)
  hence "((λx. f x powi n) has_field_derivative (of_int n * f x powi (n
- 1) * f')) (at x within A)"
    by (auto intro!: derivative_eq_intros assms(1))
  also have "of_int n * f x powi (n - 1) * f' = c * f x powr' (of_int
(n - 1)) * f'"
    by (subst powr'_of_int) (use c in auto)
  also have "of_int (n - 1) = c - 1"
    using c by simp
  finally show ?thesis
    unfolding c by (subst powr'_of_int) auto
qed

lemma continuous_on_powr'_complex [continuous_intros]:
  assumes "A ⊆ {z. Re (f z) ≥ 0 ∨ Im (f z) ≠ 0}"
  assumes "⋀z. z ∈ A ⟹ f z = 0 ⟹ Re (g z) > 0"
  assumes "continuous_on A f" "continuous_on A g"
  shows   "continuous_on A (λz. f z powr' g z)"
proof -
  have "continuous_on A (λz. f z powr g z)"
    by (intro continuous_intros assms)
  also have "?this ⟷ ?thesis"
```

```
  proof (rule continuous_on_cong)
    fix x assume "x ∈ A"
    hence "f x ≠ 0 ∨ g x ≠ 0"
      using assms(2)[of x] assms(1) by (auto simp: complex_eq_iff)
    thus "f x powr g x = f x powr' g x"
      by (simp add: powr'_complex)
  qed auto
  finally show ?thesis .
qed

lemma tendsto_powr'_complex [tendsto_intros]:
  fixes f g :: "_ ⇒ complex"
  assumes "a ∉ ℝ≤0 ∨ (a = 0 ∧ Re b > 0)" and "(f ⟶ a) F" "(g ⟶
b) F"
  shows   "((λz. f z powr' g z) ⟶ a powr' b) F"
proof -
  have "((λz. f z powr g z) ⟶ a powr b) F"
    by (rule tendsto_intros) (use assms in auto)
  also have "a ≠ 0 ∨ b ≠ 0"
    using assms(1) by auto
  hence "a powr b = a powr' b"
    by (simp add: powr'_complex)
  also have "eventually (λz. f z ≠ 0 ∨ g z ≠ 0) F"
  proof (cases "a = 0")
    case True
    have "eventually (λz. g z ∈ -{0}) F"
      by (rule eventually_compose_filterlim[OF eventually_nhds_in_open
assms(3)])
         (use True assms(1) in auto)
    thus ?thesis
      by eventually_elim auto
  next
    case False
    have "eventually (λz. f z ∈ -{0}) F"
      by (rule eventually_compose_filterlim[OF eventually_nhds_in_open
assms(2)]) (use False in auto)
    thus ?thesis
      by eventually_elim auto
  qed
  hence "eventually (λz. f z powr g z = f z powr' g z) F"
    by eventually_elim (auto simp: powr'_complex)
  hence "((λz. f z powr g z) ⟶ a powr' b) F ⟷ ?thesis"
    by (intro filterlim_cong) auto
  finally show ?thesis .
qed


lemma ln_at_0': "filterlim (ln :: real ⇒ real) at_bot (at 0)"
proof -
```

```
  have "filterlim abs (at_right 0) (at (0::real))"
  proof (rule tendsto_imp_filterlim_at_right)
    show "abs ─0→ (0::real)"
      by (rule tendsto_rabs_zero) auto
  next
    show "eventually (λx. |x| > (0::real)) (at 0)"
      using eventually_neq_at_within by eventually_elim auto
  qed
  hence "filterlim (λx. ln |x| :: real) at_bot (at 0)"
    by (rule filterlim_compose[OF ln_at_0])
  also have "?this ⟷ filterlim (λx. ln x :: real) at_bot (at 0)"
    by (simp add: ln_real_def cong: if_cong)
  finally show ?thesis .
qed


lemma tendsto_powr_real [tendsto_intros]:
  fixes f g :: "_ ⇒ real"
  assumes "(f ⟶ a) F" "(g ⟶ b) F"
  assumes "a = 0 ⟶ b > 0"
  shows   "((λz. f z powr g z) ⟶ a powr b) F"
proof (cases "a = 0")
  case False
  thus ?thesis
    using assms by (auto intro!: tendsto_intros)
next
  case [simp]: True
  have "((λz. if f z = 0 then 0 else exp (g z * ln (f z))) ⟶ 0) F"
  proof (rule filterlim_If)
    show "((λz. exp (g z * ln (f z))) ⟶ 0) (inf F (principal {z. f
z ≠ 0}))"
    proof (rule filterlim_compose[OF exp_at_bot])
      show "filterlim (λz. g z * ln (f z)) at_bot (inf F (principal {z.
f z ≠ 0}))"
      proof (rule filterlim_tendsto_pos_mult_at_bot)
        show "(g ⟶ b) (inf F (principal {z. f z ≠ 0}))"
          using assms(2) by (rule filterlim_mono) auto
      next
        show "filterlim (λx. ln (f x)) at_bot (inf F (principal {z. f
z ≠ 0}))"
        proof (rule filterlim_compose[OF ln_at_0'])
          show "filterlim f (at 0) (inf F (principal {z. f z ≠ 0}))"
          proof (rule filterlim_atI)
            have "∀F x in principal {z. f z ≠ 0}. f x ≠ 0"
              by (auto simp: eventually_principal)
            thus "∀F x in inf F (principal {z. f z ≠ 0}). f x ≠ 0"
              by (rule filter_leD[rotated]) auto
          next
            show "(f ⟶ 0) (inf F (principal {z. f z ≠ 0}))"
```

```
            using assms(1) by (rule filterlim_mono) auto
        qed
      qed
    qed (use assms(3) in auto)
  qed
  qed auto
  thus ?thesis
    by (simp add: powr_def)
qed

lemmas [tendsto_intros del] = tendsto_powr tendsto_powr'

lemma tendsto_powr'_real [tendsto_intros]:
  fixes f g :: "_ ⇒ real"
  assumes "(f ⟶ a) F" "(g ⟶ b) F"
  assumes "a > 0 ∨ (a = 0 ∧ b > 0)"
  shows   "((λz. f z powr' g z) ⟶ a powr' b) F"
proof (cases "a = 0")
  case False
  with assms have "a > 0"
    by auto
  have ev: "eventually (λx. f x ∈ {0<..}) F"
    by (rule eventually_compose_filterlim[OF eventually_nhds_in_open assms(1)])

       (use ‹a > 0› in auto)
  have "((λz. f z powr g z) ⟶ a powr b) F"
    by (rule tendsto_powr_real) (use assms in auto)
  also have "?this ⟷ ((λz. f z powr' g z) ⟶ a powr' b) F"
    using ‹a > 0› by (intro filterlim_cong) (auto simp: powr'_real intro!:
eventually_mono[OF ev])
  finally show ?thesis .
next
  case [simp]: True
  with assms have "b > 0"
    by auto
  have "((λz. f z powr' g z) ⟶ 0) F"
    unfolding powr'_def
  proof (rule filterlim_If)
    have "((λz. f z powr g z) ⟶ 0) F"
      by (rule tendsto_eq_intros assms(1,2))+ (use ‹b > 0› in auto)
    thus "((λz. f z powr g z) ⟶ 0) (inf F (principal {z. g z ∉ ℤ}))"
      by (rule filterlim_mono) auto
  next
    show "((λz. f z powi the_int (g z)) ⟶ 0) (inf F (principal {z.
g z ∈ ℤ}))"
    proof (cases "inf F (principal {z. g z ∈ ℤ}) = bot")
      case False
      define n' where "n' = round b"
      define n where "n = nat n'"
```

```
    have "b = n'"
    proof -
      from False have "∃⇩F x in F. g x ∈ ℤ"
        by (auto simp: trivial_limit_def eventually_inf_principal frequently_def)
      moreover have "eventually (λx. g x ∈ ball b (1/2)) F"
        by (rule eventually_compose_filterlim[OF eventually_nhds_in_open
assms(2)]) auto
      hence "eventually (λx. dist (g x) n' < 1) F"
      proof eventually_elim
        case (elim x)
        have "dist (g x) n' ≤ dist (g x) b + dist b n'"
          by (rule dist_triangle)
        also have "dist (g x) b < 1/2"
          using elim by (simp add: dist_commute)
        also have "dist b n' ≤ 1 / 2"
          using of_int_round_abs_le[of b] by (auto simp: n'_def dist_norm
abs_minus_commute)
        finally show ?case
          by simp
      qed
      hence "eventually (λx. g x ∈ ℤ ⟶ g x = n') F"
        by eventually_elim (auto elim!: Ints_cases simp: dist_of_int)
      ultimately have "∃⇩F x in F. g x = n'"
        by (rule frequently_rev_mp)
      moreover have "F ≠ bot"
        using ‹_ ≠ bot› by auto
      ultimately show "b = n'"
        using assms(2) limit_frequently_eq ‹F ≠ bot› by blast
    qed

    have "b = n" "n > 0"
      using ‹b = n'› assms by (auto simp: n_def)

    have ev: "eventually (λx. g x = n) (inf F (principal {z. g z ∈
ℤ}))"
    proof -
      have "eventually (λx. g x ∈ ball b 1) F"
        by (rule eventually_compose_filterlim[OF eventually_nhds_in_open
assms(2)]) auto
      hence "eventually (λx. g x ∈ ℤ ⟶ g x = n') F"
        by eventually_elim (auto simp: ‹b = n'› dist_of_int elim!: Ints_cases)
      thus ?thesis
        by (simp add: eventually_inf_principal ‹b = n'› flip: ‹b = n›)
    qed

    have "((λz. f z ^ n) ⟶⟶ 0) F"
      using assms(1) by (rule tendsto_eq_intros) (use ‹n > 0› in auto)
    hence "((λz. f z ^ n) ⟶⟶ 0) (inf F (principal {z. g z ∈ ℤ}))"
      by (rule filterlim_mono) auto
```

```
      also have "?this ⟷ ((λz. f z powi the_int (g z)) ⟶ 0) (inf
F (principal {z. g z ∈ ℤ}))"
      proof (rule filterlim_cong)
        show "∀⇩F x in inf F (principal {z. g z ∈ ℤ}). f x ^ n = f x
powi the_int (g x)"
          using ev by eventually_elim auto
      qed auto
      finally show ?thesis .
    qed auto
  qed
  thus ?thesis
    using ‹b > 0› by simp
qed

lemma continuous_powr'_complex [continuous_intros]:
  assumes "continuous F f" "continuous F g"
  assumes "Re (f (netlimit F)) ≥ 0 ∨ Im (f (netlimit F)) ≠ 0"
  assumes "f (netlimit F) = 0 ⟶ Re (g (netlimit F)) > 0"
  shows   "continuous F (λz. f z powr' g z :: complex)"
  using assms unfolding continuous_def
  by (intro tendsto_intros) (auto simp: complex_nonpos_Reals_iff complex_eq_iff)

end
```

2 Auxiliary material on filters and dominated convergence

```
theory More_Dominated_Convergence
  imports "HOL-Complex_Analysis.Complex_Analysis" "HOL-Library.Going_To_Filter"
begin
```

2.1 Linear orders with unbounded sequences

We define type classes for linear orders that contain sequences that "tend to infinity". These are also known as "countably cofinite".

```
class seq_at_top = linorder +
  assumes seq_at_top_aux: "∃f::nat ⇒ 'a. ∀x. eventually (λn. f n ≥
x) sequentially"

lemma seq_at_top: "∃f::nat ⇒ 'a :: seq_at_top. filterlim f at_top sequentially"
  using seq_at_top_aux unfolding filterlim_at_top by blast

lemma seq_at_topI [intro?]:
  "filterlim (f :: nat ⇒ 'a :: linorder) at_top sequentially ⟹ OFCLASS('a,
seq_at_top_class)"
  by intro_classes (auto simp: filterlim_at_top)
```

```
class seq_at_bot = linorder +
  assumes seq_at_bot_aux: "∃f::nat ⇒ 'a. ∀x. eventually (λn. f n ≤
x) sequentially"

lemma seq_at_bot: "∃f::nat ⇒ 'a :: seq_at_bot. filterlim f at_bot sequentially"
  using seq_at_bot_aux unfolding filterlim_at_bot by blast

lemma seq_at_botI [intro?]:
  "filterlim (f :: nat ⇒ 'a :: linorder) at_bot sequentially ⟹ OFCLASS('a,
seq_at_bot_class)"
  by intro_classes (auto simp: filterlim_at_bot)
```

An archimedian field is countably cofinite, i.e. there exist sequences that tend to $\pm\infty$.

```
context archimedean_field
begin

lemma eventually_of_nat_ge: "eventually (λn. of_nat n ≥ x) sequentially"
proof -
  obtain m where "of_int m ≥ x"
    using ex_le_of_int[of x] by blast
  hence m: "of_nat (nat m) ≥ x"
    by (metis le_cases of_int_0_le_iff of_nat_0_le_iff of_nat_nat order.trans)
  show ?thesis
    using eventually_ge_at_top[of "nat m"]
  proof eventually_elim
    case (elim n)
    thus ?case
      using order.trans of_nat_le_iff m by blast
  qed
qed

subclass seq_at_top
proof
  show "∃f. ∀x::'a. ∀F n in sequentially. x ≤ f n" using eventually_of_nat_ge
    by metis
qed

subclass seq_at_bot
proof
  have "eventually (λn. -of_nat n ≤ x) sequentially" for x :: 'a
    using eventually_of_nat_ge[of "-x"] by eventually_elim (simp add:
minus_le_iff)
  thus "∃f. ∀x::'a. ∀F n in sequentially. x ≥ f n"
    by metis
qed

end
```

Complete linear orders are obviously countably cofinite, since even the singleton set *{top}* is cofinite.

```
context complete_linorder
begin

subclass seq_at_top
proof
  show "∃f. ∀x::'a. ∀F n in sequentially. x ≤ f n"
    by (rule exI[of _ "λn. top_class.top"]) auto
qed

subclass seq_at_bot
proof
  show "∃f. ∀x::'a. ∀F n in sequentially. f n ≤ x"
    by (rule exI[of _ "λn. bot_class.bot"]) auto
qed

end




instance nat :: seq_at_bot
proof
  show "filterlim (λn::nat. 0 :: nat) at_bot sequentially"
    by (subst filterlim_at_bot) simp_all
qed

instance nat :: seq_at_top
proof
  show "filterlim (λn. n) at_top sequentially"
    by (rule filterlim_ident)
qed

instance int :: seq_at_top
proof
  show "filterlim int at_top sequentially"
    by (metis filterlim_int_sequentially)
qed

instance int :: seq_at_bot
proof
  show "filterlim (λn. -int n) at_bot sequentially"
    by (rule filterlim_compose[OF _ filterlim_int_sequentially])
       (simp add: at_bot_mirror eventually_filtermap filterlim_iff)
qed
```

2.2 Countably generated filters

For convenience, we show that if we have a countably generated filter, we can assume w.l.o.g. that the sequence of sets generating it is decreasing.

```
lemma countably_generated_filterI_decseq:
  assumes "antimono_on UNIV B" "⋀P. eventually P F ⟷ (∃i::nat. ∀x∈B
i. P x)"
  shows   "countably_generated_filter F"
  unfolding countably_generated_filter_def
proof
  show "F = (INF n. principal (B n))"
  proof (rule filter_eqI)
    fix P :: "'a ⇒ bool"
    have *: "∃x. B x ⊆ B a ∧ B x ⊆ B b" for a b
      by (rule exI[of _ "max a b"]) (use monotoneD[OF assms(1)] in auto)
    show "eventually P F ⟷ eventually P (INF n. principal (B n))"
      by (subst eventually_INF_base) (use * assms(2) in ‹auto simp: eventually_principal›)
  qed
qed

lemma countably_generated_filter_iff_decseq:
  "countably_generated_filter F ⟷
    (∃B. antimono_on UNIV B ∧ (∀P. eventually P F ⟷ (∃i::nat. ∀x∈B
i. P x)))"
  using countably_generated_filterI_decseq countably_generated_filter_has_antimono_basis
  by metis

lemma countably_generated_filter_altdef:
  "countably_generated_filter F ⟷ (∃U. countable U ∧ F = (INF X∈U.
principal X))"
proof
  assume "countably_generated_filter F"
  then obtain U where "F = (INF n::nat. principal (U n))"
    unfolding countably_generated_filter_def by blast
  thus "(∃U. countable U ∧ F = (INF X∈U. principal X))"
    by (intro exI[of _ "range U"]) (simp_all add: image_image)
next
  assume "∃U. countable U ∧ F = (INF X∈U. principal X)"
  then obtain U where U: "countable U" "F = (INF X∈U. principal X)"
    by blast
  define B where "B = from_nat_into (insert UNIV U)"
  have "(INF n. principal (B n)) = Inf ((λX. principal X) ‘ range B)"
    by (simp add: image_image)
  also have "range B = insert UNIV U"
    unfolding B_def using range_from_nat_into[of "insert {} U"] U by
simp
  also have "Inf (principal ‘ insert UNIV U) = F"
    using U by simp
  finally have *: "F = (INF n. principal (B n))" ..
```

```
  show "countably_generated_filter F"
    unfolding countably_generated_filter_def by (rule exI, rule *)
qed
```

Countably generated filters are sequential, i.e. if any sequence that tends to the filter is eventually contained in some set, then that set is in the filter.

```
lemma countably_generated_filter_sequential:
  assumes "countably_generated_filter F"
  assumes "(⋀f. filterlim f F sequentially ⟹ eventually (λn. P (f
n)) sequentially)"
  shows   "eventually P F"
proof -
  obtain B where B: "antimono_on UNIV B" "⋀P. eventually P F ⟷ (∃i::nat.
∀x∈B i. P x)"
    using countably_generated_filter_has_antimono_basis[OF assms(1)] by
metis
  have *: "∀P. (∃i. Ball (B i) P) ⟶ (∃N. ∀n≥N. P (f n)) ⟹ ∃n::nat.
P (f n)" for f
    using assms unfolding B(2) filterlim_iff eventually_at_top_linorder
    by blast
  from * show "eventually P F"
    using decseqD[OF B(1)] unfolding B(2) filterlim_iff eventually_at_top_linorder
subset_iff
    by metis
qed


named_theorems countably_generated_filter_intros

lemma countably_generated_filter_top [countably_generated_filter_intros]:

  "countably_generated_filter top_class.top"
  unfolding countably_generated_filter_altdef by (rule exI[of _ "{}"])
simp_all

lemma countably_generated_filter_bot [countably_generated_filter_intros]:
  "countably_generated_filter bot_class.bot"
  unfolding countably_generated_filter_altdef by (rule exI[of _ "{{}}"])
simp_all

lemma countably_generated_filter_principal [countably_generated_filter_intros]:
  "countably_generated_filter (principal A)"
  unfolding countably_generated_filter_altdef by (rule exI[of _ "{A}"])
simp_all

lemma countably_based_filtermap [countably_generated_filter_intros]:
  assumes "countably_generated_filter F"
  shows   "countably_generated_filter (filtermap f F)"
proof -
```

```
  obtain B where B: "antimono_on UNIV B" "⋀P. eventually P F ⟷ (∃i::nat.
∀x∈B i. P x)"
    using countably_generated_filter_has_antimono_basis[OF assms] by metis
  show ?thesis
  proof (rule countably_generated_filterI_decseq)
    show "antimono_on UNIV (λi. f ‘ B i)"
      by (intro monotoneI image_mono monotoneD[OF B(1)])
  next
    fix P show "eventually P (filtermap f F) ⟷ (∃i. ∀x∈f ‘ B i. P
x)"
      unfolding eventually_filtermap B(2) by blast
  qed
qed

lemma countably_based_filtercomap [countably_generated_filter_intros]:

  assumes "countably_generated_filter F"
  shows   "countably_generated_filter (filtercomap f F)"
proof -
  obtain B where B: "antimono_on UNIV B" "⋀P. eventually P F ⟷ (∃i::nat.
∀x∈B i. P x)"
    using countably_generated_filter_has_antimono_basis[OF assms] by metis
  show ?thesis
  proof (rule countably_generated_filterI_decseq)
    show "antimono_on UNIV (λi. f -‘ B i)"
      by (intro monotoneI vimage_mono monotoneD[OF B(1)])
  next
    fix P show "eventually P (filtercomap f F) ⟷ (∃i. ∀x∈f -‘ B i.
P x)"
      unfolding eventually_filtercomap B(2) by fast
  qed
qed

lemma countably_generated_filter_inf [countably_generated_filter_intros]:
  assumes "countably_generated_filter F" "countably_generated_filter
G"
  shows   "countably_generated_filter (inf F G)"
proof -
  obtain BF where BF: "antimono_on UNIV BF" "F = (INF n::nat. principal
(BF n))"
    using countably_generated_filter_has_antimono_basis[OF assms(1)] by
metis
  obtain BG where BG: "antimono_on UNIV BG" "G = (INF n::nat. principal
(BG n))"
    using countably_generated_filter_has_antimono_basis[OF assms(2)] by
metis
  have "inf F G = Inf (range (λn. principal (BF n)) ∪ range (λn. principal
(BG n)))"
    unfolding BF BG by (rule Inf_union_distrib [symmetric])
```

```
  also have "range (λn. principal (BF n)) ∪ range (λn. principal (BG n))
=
                principal ‘ (range BF ∪ range BG)"
    by blast
  finally have "inf F G = Inf (principal ‘ (range BF ∪ range BG))" .
  moreover have "countable (range BF ∪ range BG)"
    by blast
  ultimately show ?thesis
    unfolding countably_generated_filter_altdef by blast
qed

lemma countably_generated_filter_sup [countably_generated_filter_intros]:
  assumes "countably_generated_filter F" "countably_generated_filter
G"
  shows   "countably_generated_filter (sup F G)"
proof -
  obtain BF where BF: "antimono_on UNIV BF" "⋀P. eventually P F ⟷
(∃i::nat. ∀x∈BF i. P x)"
    using countably_generated_filter_has_antimono_basis[OF assms(1)] by
metis
  obtain BG where BG: "antimono_on UNIV BG" "⋀P. eventually P G ⟷
(∃i::nat. ∀x∈BG i. P x)"
    using countably_generated_filter_has_antimono_basis[OF assms(2)] by
metis
  show ?thesis
  proof (rule countably_generated_filterI_decseq)
    show "antimono_on UNIV (λi. BF i ∪ BG i)"
      by (intro monotone_onI Un_mono antimonoD[OF BF(1)] antimonoD[OF
BG(1)])
  next
    show "eventually P (sup F G) ⟷ (∃i. ∀x∈BF i∪BG i. P x)" for P
    proof
      assume "(∃i. ∀x∈BF i∪BG i. P x)"
      thus "eventually P (sup F G)"
        by (auto simp: eventually_sup BF BG)
    next
      assume "eventually P (sup F G)"
      then obtain i j where ij: "∀x∈BF i. P x" "∀x∈BG j. P x"
        by (auto simp: BF BG eventually_sup)
      define k where "k = max i j"
      have "BF k ⊆ BF i" "BG k ⊆ BG j"
        by (rule antimonoD[OF BF(1)] antimonoD[OF BG(1)]; simp add: k_def;
fail)+
      thus "(∃i. ∀x∈BF i∪BG i. P x)"
        by (intro exI[of _ k]) (use ij in auto)
    qed
  qed
qed
```

```
lemma countably_generated_filter_prod [countably_generated_filter_intros]:
  assumes "countably_generated_filter F" "countably_generated_filter
G"
  shows   "countably_generated_filter (F ×F G)"
proof -
  obtain BF where BF: "countable BF" "F = Inf (principal ‘ BF)"
    using assms(1) unfolding countably_generated_filter_altdef by metis
  obtain BG where BG: "countable BG" "G = Inf (principal ‘ BG)"
    using assms(2) unfolding countably_generated_filter_altdef by metis
  define BF' where "BF' = insert UNIV BF"
  define BG' where "BG' = insert UNIV BG"
  have [simp]: "BF' ≠ {}" "BG' ≠ {}"
    by (auto simp: BF'_def BG'_def)
  define B where "B = ((λ(X,Y). X × Y) ‘ (BF'×BG'))"

  have "F ×F G = Inf (principal ‘ BF') ×F Inf (principal ‘ BG')"
    by (simp_all add: BF BG BF'_def BG'_def)
  also have "... = (INF X∈BF'. INF Y∈BG'. principal (X × Y))"
    by (subst prod_filter_INF) (simp_all add: principal_prod_principal)
  also have "... = (INF (X,Y)∈BF'×BG'. principal (X × Y))"
    by (subst INF_pair) (simp add: case_prod_unfold)
  finally have "F ×F G = Inf (principal ‘ B)"
    by (simp add: image_image case_prod_unfold B_def)
  moreover have "countable B"
    unfolding B_def BF'_def BG'_def using BF(1) BG(1) by auto
  ultimately show ?thesis
    unfolding countably_generated_filter_altdef by blast
qed

lemma countably_generated_filter_Inf [countably_generated_filter_intros]:
  assumes "countable F" "⋀G. G ∈ F ⟹ countably_generated_filter G"
  shows   "countably_generated_filter (Inf F)"
proof -
  have "∀G∈F. ∃B. antimono_on UNIV B ∧ G = (INF n::nat. principal (B
n))"
    using countably_generated_filter_has_antimono_basis[OF assms(2)] by
metis
  then obtain B where B: "⋀G. G ∈ F ⟹ antimono_on UNIV (B G)"
                         "⋀G. G ∈ F ⟹ G = (INF n::nat. principal (B
G n))"
    by metis
  define B' where "B' = (λ(G,n). B G n) ‘ (F × UNIV)"

  have "F = (λG. (INF n::nat. principal (B G n))) ‘ F"
    using B(2) by auto
  also have "Inf ... = (INF p∈F×UNIV. principal (B (fst p) (snd p)))"
    by (rule INF_pair)
  also have "... = (INF X∈B'. principal X)"
    by (simp add: B'_def image_image case_prod_unfold)
```

```
  finally have "Inf F = (INF X∈B'. principal X)" .
  moreover have "countable B'"
    unfolding B'_def using assms(1) by blast
  ultimately show ?thesis
    unfolding countably_generated_filter_altdef by blast
qed

lemma countably_generated_filter_Sup_finite [countably_generated_filter_intros]:
  assumes "finite F" "⋀G. G ∈ F ⟹ countably_generated_filter G"
  shows   "countably_generated_filter (Sup F)"
  using assms
  by (induction rule: finite_induct) (auto intro!: countably_generated_filter_intros)

lemma countably_generated_filter_going_to [countably_generated_filter_intros]:
  assumes "countably_generated_filter F"
  shows   "countably_generated_filter (f going_to F within A)"
  unfolding going_to_within_def by (intro countably_generated_filter_intros
assms)

lemma countably_generated_filter_at_top [countably_generated_filter_intros]:
  "countably_generated_filter (at_top :: 'a :: seq_at_top filter)"
proof -
  from seq_at_top obtain f :: "nat ⇒ 'a" where f: "filterlim f at_top
sequentially"
    by blast
  define B where "B = (λn. {f n..})"
  have *: "at_top = (INF n. principal (B n))"
  proof (rule filter_eqI)
    fix P :: "'a ⇒ bool"
    have "∃x. f a ≤ f x ∧ f b ≤ f x" for a b
      by (rule exI[of _ "if f a ≤ f b then b else a"]) auto
    hence "eventually P (INF n. principal (B n)) ⟷ (∃i. ∀x∈B i. P
x)"
      by (subst eventually_INF_base) (auto simp: eventually_principal
B_def)
    also have "... ⟷ eventually P at_top"
      using f unfolding eventually_at_top_linorder filterlim_at_top B_def
      by (metis le_left_mono le_refl atLeast_iff)
    finally show "eventually P at_top ⟷ eventually P (INF n. principal
(B n))"
      by (simp add: B_def)
  qed
  show "countably_generated_filter (at_top :: 'a filter)"
    unfolding countably_generated_filter_def by (rule exI, rule *)
qed

lemma countably_generated_filter_at_top [countably_generated_filter_intros]:
  "countably_generated_filter (at_top :: 'a :: {second_countable_topology,linorder_topology
filter)"
```

```
  oops

lemma countably_generated_filter_at_bot [countably_generated_filter_intros]:
  "countably_generated_filter (at_bot :: 'a :: seq_at_bot filter)"
proof -
  from seq_at_bot obtain f :: "nat ⇒ 'a" where f: "filterlim f at_bot
sequentially"
    by blast
  define B where "B = (λn. {..f n})"
  have *: "at_bot = (INF n. principal (B n))"
  proof (rule filter_eqI)
    fix P :: "'a ⇒ bool"
    have "∃x. f a ≥ f x ∧ f b ≥ f x" for a b
      by (rule exI[of _ "if f a ≤ f b then a else b"]) auto
    hence "eventually P (INF n. principal (B n)) ⟷ (∃i. ∀x∈B i. P
x)"
      by (subst eventually_INF_base) (auto simp: eventually_principal
B_def)
    also have "... ⟷ eventually P at_bot"
      using f unfolding eventually_at_bot_linorder eventually_at_top_linorder
filterlim_at_bot B_def
      by (metis atMost_iff atMost_subset_iff subset_iff)
    finally show "eventually P at_bot ⟷ eventually P (INF n. principal
(B n))"
      by (simp add: B_def)
  qed
  show "countably_generated_filter (at_bot :: 'a filter)"
    unfolding countably_generated_filter_def by (rule exI, rule *)
qed

lemma countably_generated_filter_nhds [countably_generated_filter_intros]:
  "countably_generated_filter (nhds (x :: 'a :: first_countable_topology))"
  unfolding countably_generated_filter_def using nhds_countable[of x]
by metis

lemma countably_generated_filter_at_within [countably_generated_filter_intros]:
  "countably_generated_filter (at (x :: 'a :: first_countable_topology)
within A)"
  unfolding at_within_def by (intro countably_generated_filter_intros)


lemma at_infinity_eq_filtercomap: "at_infinity = filtercomap norm at_top"
  by (rule filter_eqI) (simp_all add: eventually_filtercomap_at_top_linorder
eventually_at_infinity)

lemma countably_generated_filter_at_infinity [countably_generated_filter_intros]:
  "countably_generated_filter at_infinity"
  unfolding at_infinity_eq_filtercomap by (intro countably_generated_filter_intros)
```

```
lemmas [countably_generated_filter_intros] = countably_generated_uniformity
```

2.3 Sequential filters

We call a filter *sequential* if it can be "approached" by sequences in the sense that if a a properties holds eventually on all sequences that approach the filter, then it also holds eventually for the filter.

Importantly, any countably generated filter is sequential.

```
locale sequential_filter =
  fixes F :: "'a filter"
  assumes approachable:
    "(⋀f. filterlim f F sequentially ⟹ eventually (λn. P (f n)) sequentially)
⟹ eventually P F"
begin

lemma filterlim_sequentially_imp_filterlim:
  assumes "⋀X. filterlim X F sequentially ⟹ filterlim (λn. f (X n))
G sequentially"
  shows   "filterlim f G F"
  unfolding filterlim_iff
proof safe
  fix P assume P: "eventually P G"
  show "eventually (λx. P (f x)) F"
  proof (rule approachable)
    fix X assume "filterlim X F sequentially"
    hence "filterlim (λn. f (X n)) G sequentially"
      by (rule assms)
    thus "eventually (λn. P (f (X n))) sequentially"
      unfolding filterlim_iff using P by blast
  qed
qed

end


lemma sequential_filtermap:
  assumes "sequential_filter F"
  shows   "sequential_filter (filtermap (g :: 'a ⇒ 'b) F)"
proof
  interpret sequential_filter F by fact
  fix P
  assume *: "(⋀f. filterlim f (filtermap g F) sequentially ⟹ ∀⇩F n
in sequentially. P (f n))"
  show "eventually P (filtermap g F)"
    unfolding eventually_filtermap
  proof (rule approachable)
    fix f assume "filterlim f F sequentially"
    hence "filtermap g (filtermap f sequentially) ≤ filtermap g F"
```

```
      by (intro filtermap_mono) (auto simp: filterlim_def)
    hence "filterlim (λn. g (f n)) (filtermap g F) sequentially"
      unfolding filterlim_def by (simp add: filtermap_filtermap)
    thus "eventually (λn. P (g (f n))) sequentially"
      by (rule *)
  qed
qed

lemma countably_generated_filter_imp_sequential_filter:
  assumes "countably_generated_filter F"
  shows   "sequential_filter F"
  by standard (use countably_generated_filter_sequential[OF assms] in
blast)


interpretation bot: sequential_filter "bot_class.bot"
  by (intro countably_generated_filter_imp_sequential_filter countably_generated_filter_int

interpretation top: sequential_filter "top_class.top"
  by (intro countably_generated_filter_imp_sequential_filter countably_generated_filter_int

interpretation principal: sequential_filter "principal A"
  by (intro countably_generated_filter_imp_sequential_filter countably_generated_filter_int

interpretation nhds: sequential_filter "nhds (x :: 'a :: first_countable_topology)"
  by (intro countably_generated_filter_imp_sequential_filter countably_generated_filter_int

interpretation at_within: sequential_filter
  "at (x :: 'a :: first_countable_topology) within A"
  by (intro countably_generated_filter_imp_sequential_filter countably_generated_filter_int

interpretation at_top: sequential_filter "at_top :: 'a :: seq_at_top filter"
  by (intro countably_generated_filter_imp_sequential_filter countably_generated_filter_int

interpretation at_bot: sequential_filter "at_bot :: 'a :: seq_at_bot filter"
  by (intro countably_generated_filter_imp_sequential_filter countably_generated_filter_int

interpretation at_infinity: sequential_filter at_infinity
  by (intro countably_generated_filter_imp_sequential_filter countably_generated_filter_int
```

2.4 Set-wise monotone convergence

We now introduce the notion of a family of sets $A(x)$ converging to another set B "from the inside" as $x \to F$, in the sense that eventually $A(x) \subseteq B$ as $x \to F$ and, for every y, eventually $y \in A(x) \longleftrightarrow y \in B$ as $x \to F$.

That is, $A(x)$ converges to B pointwise from within B in the discrete topology. Typical examples include that e.g. $A(x) = [l(x), r(x)]$ converges to (a, b) if $l(x) \to a^+$ and $r(x) \to b^-$, or $A(x) = [l(x), r(x)]$ converges to $\mathbb{R}$ if

$l(x) \to -\infty$ and $r(x) \to \infty$.

```
definition tendsto_set :: "'b measure ⇒ ('a ⇒ 'b set) ⇒ 'b set ⇒ 'a
filter ⇒ bool" where
  "tendsto_set M A B F ⟷
     (∃C. C ∈ null_sets M ∧ (∀x. x ∉ C ⟶ eventually (λy. x ∈ A y
⟷ x ∈ B) F))"

named_theorems tendsto_set_intros

lemma tendsto_set_null_sets_transfer:
  assumes "tendsto_set M f A F" "sym_diff A B ∈ null_sets M"
  shows   "tendsto_set M f B F"
proof -
  from assms(1) obtain C where C:
    "C ∈ null_sets M" "⋀x. x ∉ C ⟹ ∀_F y in F. (x ∈ f y) ⟷ (x ∈
A)"
    by (auto simp: tendsto_set_def)
  show ?thesis
    unfolding tendsto_set_def
  proof (rule exI[of _ "C ∪ sym_diff A B"], intro conjI allI impI)
    show "C ∪ sym_diff A B ∈ null_sets M"
      using C(1) assms(2) by auto
  next
    fix x assume x: "x ∉ C ∪ sym_diff A B"
    hence "x ∉ C"
      by auto
    show "∀_F y in F. x ∈ f y ⟷ x ∈ B"
      using C(2)[OF ‹x ∉ C›] by eventually_elim (use x in auto)
  qed
qed

lemma tendsto_set_cong:
  assumes "null_sets M = null_sets N" "eventually (λx. f x = g x) F"
          "sym_diff A B ∈ null_sets M" "F = F'"
  shows   "tendsto_set M f A F ⟷ tendsto_set N g B F'"
proof -
  have "tendsto_set M f A F ⟷ tendsto_set N g A F"
    unfolding tendsto_set_def assms(4)[symmetric]
    by (intro arg_cong[of _ _ Ex] ext conj_cong eventually_cong[OF assms(2)]
all_cong)
       (simp_all add: assms(1))
  also have "... ⟷ tendsto_set N g B F'"
    using tendsto_set_null_sets_transfer[of N g A F' B]
          tendsto_set_null_sets_transfer[of N g B F' A] assms by (auto
simp: Un_commute)
  finally show ?thesis .
qed

lemma tendsto_set_Icc_Icc [tendsto_set_intros]:
```

```
  fixes a b :: "'a :: linorder_topology"
  assumes lim: "filterlim l (nhds a) F" "filterlim r (nhds b) F"
  assumes "{a, b} ∈ null_sets M"
  shows   "tendsto_set M (λx. {l x..r x}) {a..b} F"
  unfolding tendsto_set_def
proof (rule exI[of _ "{a, b}"], intro conjI allI impI)
  show "{a, b} ∈ null_sets M"
    by fact
next
  fix x assume x: "x ∉ {a, b}"
  consider "x < a" | "x ∈ {a<..<b}" | "x > b"
    using x by force
  thus "∀⇩F y in F. (x ∈ {l y..r y}) ⟷ (x ∈ {a..b})"
  proof cases
    assume x: "x < a"
    have "eventually (λy. l y ∈ {x<..}) F"
      by (rule eventually_compose_filterlim[OF eventually_nhds_in_open
lim(1)]) (use x in auto)
    thus ?thesis
      by eventually_elim (use x in auto)
  next
    assume x: "x > b"
    have "eventually (λy. r y ∈ {..<x}) F"
      by (rule eventually_compose_filterlim[OF eventually_nhds_in_open
lim(2)]) (use x in auto)
    thus ?thesis
      by eventually_elim (use x in auto)
  next
    assume x: "x ∈ {a<..<b}"
    have "eventually (λy. l y ∈ {..<x}) F"
      by (rule eventually_compose_filterlim[OF eventually_nhds_in_open
lim(1)]) (use x in auto)
    moreover have "eventually (λy. r y ∈ {x<..}) F"
      by (rule eventually_compose_filterlim[OF eventually_nhds_in_open
lim(2)]) (use x in auto)
    ultimately show "∀⇩F y in F. (x ∈ {l y..r y}) = (x ∈ {a..b})"
      by eventually_elim (use x in auto)
  qed
qed

lemma tendsto_set_Icc_Ici [tendsto_set_intros]:
  fixes a :: "'a :: linorder_topology"
  assumes lim: "filterlim l (nhds a) F" "filterlim r at_top F"
  assumes "{a} ∈ null_sets M"
  shows   "tendsto_set M (λx. {l x..r x}) {a..} F"
  unfolding tendsto_set_def
proof (rule exI[of _ "{a}"], intro conjI allI impI)
  show "{a} ∈ null_sets M"
    by fact
```

```
next
  fix x assume x: "x ∉ {a}"
  consider "x < a" | "x > a"
    using x by force
  thus "∀F y in F. (x ∈ {l y..r y}) ⟷ (x ∈ {a..})"
  proof cases
    assume x: "x < a"
    have "eventually (λy. l y ∈ {x<..}) F"
      by (rule eventually_compose_filterlim[OF eventually_nhds_in_open
lim(1)]) (use x in auto)
    thus ?thesis
      by eventually_elim (use x in auto)
  next
    assume x: "x > a"
    have "eventually (λy. l y ∈ {..<x}) F"
      by (rule eventually_compose_filterlim[OF eventually_nhds_in_open
lim(1)]) (use x in auto)
    moreover have "eventually (λy. r y ≥ x) F"
      by (rule eventually_compose_filterlim[OF eventually_ge_at_top lim(2)])
    ultimately show "∀F y in F. (x ∈ {l y..r y}) = (x ∈ {a..})"
      by eventually_elim (use x in auto)
  qed
qed

lemma tendsto_set_Icc_Iic [tendsto_set_intros]:
  fixes b :: "'a :: linorder_topology"
  assumes lim: "filterlim l at_bot F" "filterlim r (nhds b) F"
  assumes "{b} ∈ null_sets M"
  shows   "tendsto_set M (λx. {l x..r x}) {..b} F"
  unfolding tendsto_set_def
proof (rule exI[of _ "{b}"], intro conjI allI impI)
  show "{b} ∈ null_sets M"
    by fact
next
  fix x assume x: "x ∉ {b}"
  consider "x < b" | "x > b"
    using x by force
  thus "∀F y in F. (x ∈ {l y..r y}) ⟷ (x ∈ {..b})"
  proof cases
    assume x: "x > b"
    have "eventually (λy. r y ∈ {..<x}) F"
      by (rule eventually_compose_filterlim[OF eventually_nhds_in_open
lim(2)]) (use x in auto)
    thus ?thesis
      by eventually_elim (use x in auto)
  next
    assume x: "x < b"
    have "eventually (λy. l y ≤ x) F"
      by (rule eventually_compose_filterlim[OF eventually_le_at_bot lim(1)])
```

```
    moreover have "eventually (λy. r y ∈ {x<..}) F"
      by (rule eventually_compose_filterlim[OF eventually_nhds_in_open
lim(2)]) (use x in auto)
    ultimately show "∀F y in F. (x ∈ {l y..r y}) = (x ∈ {..b})"
      by eventually_elim (use x in auto)
  qed
qed

lemma tendsto_set_Icc_UNIV [tendsto_set_intros]:
  fixes l r :: "_ ⇒ 'a :: linorder_topology"
  assumes lim: "filterlim l at_bot F" "filterlim r at_top F"
  shows   "tendsto_set M (λx. {l x..r x}) UNIV F"
  unfolding tendsto_set_def
proof (rule exI[of _ "{}"], intro conjI allI impI)
  show "{} ∈ null_sets M"
    by simp
next
  fix x
  have "eventually (λy. l y ≤ x) F"
    by (rule eventually_compose_filterlim[OF eventually_le_at_bot lim(1)])
  moreover have "eventually (λy. r y ≥ x) F"
    by (rule eventually_compose_filterlim[OF eventually_ge_at_top lim(2)])
  ultimately show "∀F y in F. (x ∈ {l y..r y}) ⟷ (x ∈ UNIV)"
    by eventually_elim auto
qed

lemma tendsto_set_Ici_Ici [tendsto_set_intros]:
  fixes a :: "'a :: linorder_topology"
  assumes lim: "filterlim l (nhds a) F"
  assumes "{a} ∈ null_sets M"
  shows   "tendsto_set M (λx. {l x..}) {a..} F"
  unfolding tendsto_set_def
proof (rule exI[of _ "{a}"], intro conjI allI impI)
  show "{a} ∈ null_sets M"
    by fact
next
  fix x assume x: "x ∉ {a}"
  consider "x < a" | "x > a"
    using x by force
  thus "∀F y in F. (x ∈ {l y..}) ⟷ (x ∈ {a..})"
  proof cases
    assume x: "x < a"
    have "eventually (λy. l y ∈ {x<..}) F"
      by (rule eventually_compose_filterlim[OF eventually_nhds_in_open
lim(1)]) (use x in auto)
    thus ?thesis
      by eventually_elim (use x in auto)
  next
    assume x: "x > a"
```

```
    have "eventually (λy. l y ∈ {..<x}) F"
      by (rule eventually_compose_filterlim[OF eventually_nhds_in_open
lim(1)]) (use x in auto)
    thus "∀F y in F. (x ∈ {l y..}) = (x ∈ {a..})"
      by eventually_elim (use x in auto)
  qed
qed

lemma tendsto_set_Iic_Iic [tendsto_set_intros]:
  fixes b :: "'a :: linorder_topology"
  assumes lim: "filterlim r (nhds b) F"
  assumes "{b} ∈ null_sets M"
  shows   "tendsto_set M (λx. {..r x}) {..b} F"
  unfolding tendsto_set_def
proof (rule exI[of _ "{b}"], intro conjI allI impI)
  show "{b} ∈ null_sets M"
    by fact
next
  fix x assume x: "x ∉ {b}"
  consider "x < b" | "x > b"
    using x by force
  thus "∀F y in F. (x ∈ {..r y}) ⟷ (x ∈ {..b})"
  proof cases
    assume x: "x > b"
    have "eventually (λy. r y ∈ {..<x}) F"
      by (rule eventually_compose_filterlim[OF eventually_nhds_in_open
lim(1)]) (use x in auto)
    thus ?thesis
      by eventually_elim (use x in auto)
  next
    assume x: "x < b"
    have "eventually (λy. r y ∈ {x<..}) F"
      by (rule eventually_compose_filterlim[OF eventually_nhds_in_open
lim(1)]) (use x in auto)
    thus "∀F y in F. (x ∈ {..r y}) = (x ∈ {..b})"
      by eventually_elim (use x in auto)
  qed
qed

lemma tendsto_set_Ici_UNIV [tendsto_set_intros]:
  fixes l :: "_ ⇒ 'a :: linorder_topology"
  assumes lim: "filterlim l at_bot F"
  shows   "tendsto_set M (λx. {l x..}) UNIV F"
  unfolding tendsto_set_def
proof (rule exI[of _ "{}"], intro conjI allI impI)
  show "{} ∈ null_sets M"
    by simp
next
  fix x :: 'a
```

```
  have "eventually (λy. l y ≤ x) F"
    by (rule eventually_compose_filterlim[OF eventually_le_at_bot lim])
  thus "∀F y in F. (x ∈ {l y..}) = (x ∈ UNIV)"
    by eventually_elim auto
qed

lemma tendsto_set_Iic_UNIV [tendsto_set_intros]:
  fixes r :: "_ ⇒ 'a :: linorder_topology"
  assumes lim: "filterlim r at_top F"
  shows   "tendsto_set M (λx. {..r x}) UNIV F"
  unfolding tendsto_set_def
proof (rule exI[of _ "{}"], intro conjI allI impI)
  show "{} ∈ null_sets M"
    by simp
next
  fix x :: 'a
  have "eventually (λy. r y ≥ x) F"
    by (rule eventually_compose_filterlim[OF eventually_ge_at_top lim])
  thus "∀F y in F. (x ∈ {..r y}) = (x ∈ UNIV)"
    by eventually_elim auto
qed
```

The next version of the lemma is slightly stronger in the sense that we can also show $\lim_{x \to a^-} \int_0^x = \int_0^a$ (i.e. the endpoint is also included).

```
lemma (in sequential_filter) filterlim_set_lebesgue_integral_set:
  fixes f :: "real ⇒ 'b::{banach, second_countable_topology}"
  assumes integrable: "set_integrable M B f"
  assumes measurable: "set_borel_measurable M A f" "eventually (λx. set_borel_measurable
M (X x) f) F"
  assumes X: "tendsto_set M X A F" "eventually (λx. X x ∈ sets M) F"
  assumes subset: "eventually (λx. X x ⊆ B) F"
  shows   "((λx. set_lebesgue_integral M (X x) f) ⟶ set_lebesgue_integral
M A f) F"
proof (rule filterlim_sequentially_imp_filterlim)
  fix g assume g: "filterlim g F sequentially"
  obtain C where C: "C ∈ null_sets M" "(∀x. eventually (λy. x ∉ C ⟶
x ∈ X y ⟷ x ∈ A) F)"
    using X(1) by (auto simp: tendsto_set_def)
  have "eventually (λn. X (g n) ∈ sets M ∧ set_borel_measurable M (X
(g n)) f ∧ X (g n) ⊆ B) sequentially"
    by (intro eventually_conj eventually_compose_filterlim[OF _ g])
       (use C X(2) measurable(2) subset in ‹auto simp: tendsto_set_def›)
  then obtain N where N:
    "⋀n. n ≥ N ⟹ X (g n) ∈ sets M" "⋀n. n ≥ N ⟹ set_borel_measurable
M (X (g n)) f"
    "⋀n. n ≥ N ⟹ X (g n) ⊆ B"
    unfolding eventually_at_top_linorder by blast
  have g': "filterlim (λn. g (n + N)) F sequentially"
    by (rule filterlim_compose[OF g filterlim_add_const_nat_at_top])
```

```
  have "(λn. set_lebesgue_integral M (X (g (n + N))) f) ⟶ set_lebesgue_integral
M A f"
    unfolding set_lebesgue_integral_def
  proof (rule integral_dominated_convergence)
    show "(λx. indicator A x *R f x) ∈ borel_measurable M"
      using measurable(1) by (simp add: set_borel_measurable_def)
  next
    fix n :: nat
    show "(λx. indicator (X (g (n + N))) x *R f x) ∈ borel_measurable
M"
      using N(2)[of "n + N"] by (simp add: set_borel_measurable_def)
  next
    show "integrable M (λx. norm (indicator B x *R f x))"
      using integrable by (intro integrable_norm) (simp add: set_integrable_def)
  next
    have "AE x in M. x ∉ C"
      by (intro AE_not_in) (use C in auto)
    thus "AE x in M. (λn. indicat_real (X (g (n + N))) x *R f x) ⟶
indicat_real A x *R f x"
    proof eventually_elim
      case (elim x)
      have "eventually (λn. x ∈ X (g (n + N)) ⟷ x ∈ A) sequentially"
        by (rule eventually_compose_filterlim[OF _ g'])
           (use C(2) elim in ‹auto simp: tendsto_set_def›)
      hence "eventually (λn. indicat_real (X (g (n + N))) x *R f x =

                                indicat_real A x *R f x) sequentially"
        by eventually_elim (auto simp: indicator_def)
      thus "(λn. indicat_real (X (g (n + N))) x *R f x) ⟶ indicat_real
A x *R f x"
        by (rule tendsto_eventually)
    qed
  next
    fix n :: nat
    have "AE x in M. x ∉ C"
      by (intro AE_not_in) (use C in auto)
    thus "AE x in M. norm (indicator (X (g (n + N))) x *R f x) ≤ norm
(indicator B x *R f x)"
    proof eventually_elim
      case (elim x)
      show "norm (indicator (X (g (n + N))) x *R f x) ≤ norm (indicator
B x *R f x)"
        using N[of "n + N"] elim by (auto simp: indicator_def)
    qed
  qed
  thus "(λn. set_lebesgue_integral M (X (g n)) f) ⟶ set_lebesgue_integral
M A f"
    by (rule LIMSEQ_offset)
```

qed

end

3 Integral forms of the Beta function

theory *More_Beta*
  imports *"HOL-Complex_Analysis.Complex_Analysis" More_Dominated_Convergence*
begin

lemma *Gamma_legendre_duplication_real:*
  fixes *z :: real*
  assumes *"z ∉ ℤ≤0" "z + 1/2 ∉ ℤ≤0"*
  shows *"Gamma z * Gamma (z + 1/2) = 2 powr (1 - 2 * z) * sqrt pi * Gamma (2*z)"*
proof -
  have *"complex_of_real (Gamma z * Gamma (z + 1/2)) = Gamma (of_real z) * Gamma (of_real z + 1/2)"*
    by *(simp flip: Gamma_complex_of_real)*
  also have *"... = of_real (exp ((1 - 2 * z) * (ln 2)) * sqrt pi * Gamma (2 * z))"*
    using *assms of_real_in_nonpos_Ints_iff[of z,* **where** *?'a = complex]*
          of_real_in_nonpos_Ints_iff[of "z + 1/2", **where** *?'a = complex]*
    by *(subst Gamma_legendre_duplication) (auto simp: of_real_exp simp flip: Gamma_complex_of_real)*
  also have *"exp ((1 - 2 * z) * (ln 2)) = 2 powr (1 - 2 * z)"*
    by *(simp add: powr_def)*
  finally show *?thesis*
    by *(simp only: of_real_eq_iff)*
qed

lemma *Gamma_int_plus_half:*
  *"Gamma (real n + 1 / 2) = sqrt pi * fact (2 * n) / (4 ^ n * fact n)"*
proof *(cases "n = 0")*
  case *False*
  hence *"real n ∉ ℤ≤0" "real n + 1 / 2 ∉ ℤ≤0"*
    by *auto*
  hence *"Gamma (real n) * Gamma (real n + 1 / 2) =*
          *2 powr (1 - 2 * real n) * sqrt pi * Gamma (2 * real n)"*
    by *(rule Gamma_legendre_duplication_real)*
  also have *"Gamma (real n) = fact (n - 1)"*
    using *False* **by** *(simp flip: Gamma_fact add: of_nat_diff)*
  also have *"Gamma (2 * real n) = fact (2 * n - 1)"*
    using *False* **by** *(simp flip: Gamma_fact add: of_nat_diff)*
  finally have *"Gamma (real n + 1 / 2) = 2 powr (1 - 2 * real n) * sqrt pi * (fact (2 * n - 1) / fact (n - 1))"*
    by *(simp add: field_simps)*

```
  also have "fact (2 * n - 1) / fact (n - 1) = fact (2 * n) / fact n /
(2 :: real)"
    using False by (simp add: fact_reduce[of n] fact_reduce[of "n * 2"]
field_simps)
  also have "2 powr (1 - 2 * real n) * sqrt pi * (fact (2 * n) / fact
n / 2) =
               sqrt pi * fact (2 * n) / (4 ^ n * fact n)"
    by (simp add: powr_diff powr_realpow flip: powr_powr)
  finally show ?thesis .
qed (auto simp: Gamma_one_half_real)



lemma has_field_derivative_complex_powr_right:
  "w ≠ 0 ⟹ ((λz. w powr z) has_field_derivative Ln w * w powr z) (at
z within A)"
  by (rule DERIV_subset, rule has_field_derivative_powr_right) auto


lemmas has_field_derivative_complex_powr_right' =
  has_field_derivative_complex_powr_right[THEN DERIV_chain2]


lemma uniform_limit_set_lebesgue_integral:
  fixes f :: "'a ⇒ 'b :: euclidean_space ⇒ 'c :: {banach, second_countable_topology}"
  assumes "set_integrable lborel X' g"
  assumes [measurable]: "X' ∈ sets borel"
  assumes [measurable]: "⋀y. y ∈ Y ⟹ set_borel_measurable borel X'
(f y)"
  assumes "⋀y. y ∈ Y ⟹ (AE t∈X' in lborel. norm (f y t) ≤ g t)"
  assumes "eventually (λx. X x ∈ sets borel ∧ X x ⊆ X') F"
  assumes "filterlim (λx. set_lebesgue_integral lborel (X x) g)
             (nhds (set_lebesgue_integral lborel X' g)) F"
  shows "uniform_limit Y
           (λx y. set_lebesgue_integral lborel (X x) (f y))
           (λy. set_lebesgue_integral lborel X' (f y)) F"
proof (rule uniform_limitI, goal_cases)
  case (1 ε)
  have integrable_g: "set_integrable lborel U g"
    if "U ∈ sets borel" "U ⊆ X'" for U
    by (rule set_integrable_subset[OF assms(1)]) (use that in auto)
  have "eventually (λx. dist (set_lebesgue_integral lborel (X x) g)
                             (set_lebesgue_integral lborel X' g) < ε)
F"
    using ‹ε > 0› assms by (auto simp: tendsto_iff)
  from this show ?case using ‹eventually (λ_. _ ∧ _) F›
  proof eventually_elim
    case (elim x)
    hence [measurable]:"X x ∈ sets borel" and "X x ⊆ X'" by auto
```

```
    have integrable: "set_integrable lborel U (f y)"
      if "y ∈ Y" "U ∈ sets borel" "U ⊆ X'" for y U
      apply (rule set_integrable_subset)
        apply (rule set_integrable_bound[OF assms(1)])
         apply (use assms(3) that in ‹simp add: set_borel_measurable_def›)
      using assms(4)[OF ‹y ∈ Y›] apply eventually_elim apply force
      using that apply simp_all
      done
    show ?case
    proof
      fix y assume "y ∈ Y"
      have "dist (set_lebesgue_integral lborel (X x) (f y))
                 (set_lebesgue_integral lborel X' (f y)) =
            norm (set_lebesgue_integral lborel X' (f y) -
                  set_lebesgue_integral lborel (X x) (f y))"
        by (simp add: dist_norm norm_minus_commute)
      also have "set_lebesgue_integral lborel X' (f y) -
                    set_lebesgue_integral lborel (X x) (f y) =
                 set_lebesgue_integral lborel (X' - X x) (f y)"
        unfolding set_lebesgue_integral_def
        apply (subst Bochner_Integration.integral_diff [symmetric])
        unfolding set_integrable_def [symmetric]
          apply (rule integrable; (fact | simp))
         apply (rule integrable; fact)
        apply (intro Bochner_Integration.integral_cong)
         apply (use ‹X x ⊆ X'› in ‹auto simp: indicator_def›)
        done
      also have "norm ... ≤ (∫ t∈X'-X x. norm (f y t) ∂lborel)"
        by (intro set_integral_norm_bound integrable) (fact | simp)+
      also have "AE t∈X' - X x in lborel. norm (f y t) ≤ g t"
        using assms(4)[OF ‹y ∈ Y›] by eventually_elim auto
      with ‹y ∈ Y› have "(∫ t∈X'-X x. norm (f y t) ∂lborel) ≤ (∫ t∈X'-X
x. g t ∂lborel)"
        by (intro set_integral_mono_AE set_integrable_norm integrable
integrable_g) auto
      also have "... = (∫ t∈X'. g t ∂lborel) - (∫ t∈X x. g t ∂lborel)"
        unfolding set_lebesgue_integral_def
        apply (subst Bochner_Integration.integral_diff [symmetric])
        unfolding set_integrable_def [symmetric]
          apply (rule integrable_g; (fact | simp))
         apply (rule integrable_g; fact)
        apply (intro Bochner_Integration.integral_cong)
         apply (use ‹X x ⊆ X'› in ‹auto simp: indicator_def›)
        done
      also have "... ≤ dist (∫ t∈X x. g t ∂lborel) (∫ t∈X'. g t ∂lborel)"
        by (simp add: dist_norm)
      also have "... < ε" by fact
      finally show "dist (set_lebesgue_integral lborel (X x) (f y))
                         (set_lebesgue_integral lborel X' (f y)) < ε"
```

```
.
    qed
  qed
qed

lemma Beta_nonneg_real:
  assumes "a > 0" "b > 0"
  shows   "Beta a (b::real) ≥ 0"
  by (rule has_integral_nonneg[OF has_integral_Beta_real]) (use assms
in auto)
```

The Beta function is given by the following integral:

$$B(a,b) = \int_0^1 t^{a-1}(1-t)^{b-1}\,\mathrm{d}t$$

```
lemma has_integral_Beta_complex:
  assumes a: "Re a > 0" and b: "Re b > 0"
  shows "((λt. of_real t powr (a - 1) * of_real (1 - t) powr (b - 1))

           has_integral Beta a b) {0<..<1}"
    and "(λt. of_real t powr (a - 1) * of_real (1 - t) powr (b - 1))
           absolutely_integrable_on {0<..<1}"
proof -
  define f :: "complex ⇒ complex ⇒ real ⇒ complex"
    where "f = (λa b t. of_real t powr (a - 1) * of_real (1 - t) powr
(b - 1))"
  define F where "F = (λa b. integral {0..1} (f a b))"

  have integrable: "f w z absolutely_integrable_on A"
    if wz: "Re w > 0" "Re z > 0" and A: "A ⊆ {0..1}" "A ∈ sets lebesgue"
for A w z
  proof (rule set_integrable_subset)
    have "(λt. t powr (Re w - 1) * (1 - t) powr (Re z - 1)) integrable_on
{0..1}"
      using integrable_Beta'[of "Re w" "Re z"] wz by simp
    also have "?this ⟷ (λx. norm (f w z x)) integrable_on {0..1}"
      by (intro integrable_cong)
         (auto simp: f_def norm_mult norm_powr_complex simp del: of_real_diff)
    finally have "(λx. norm (f w z x)) absolutely_integrable_on {0..1}"
      by (rule nonnegative_absolutely_integrable_1) auto
    hence "integrable lebesgue (λx. norm (indicator {0..1} x *_R f w z
x))"
      by (simp add: set_integrable_def)
    thus "f w z absolutely_integrable_on {0..1}" unfolding set_integrable_def

      by (rule Bochner_Integration.integrable_norm_cancel) (simp add:
f_def measurable_completion)
  qed (use A in auto)
```

```
  show "(λt. of_real t powr (a - 1) * of_real (1 - t) powr (b - 1))
          absolutely_integrable_on {0<..<1}"
    using integrable[of a b "{0<..<1}"] a b
    by (simp add: f_def greaterThanLessThan_subseteq_atLeastAtMost_iff)

  have integral_eq: "integral A (f w z) = set_lebesgue_integral lborel
A (f w z)"
    if wz: "Re w > 0" "Re z > 0" and A: "A ⊆ {0..1}" "A ∈ sets lborel"
for A w z
  proof -
    have "integral A (f w z) = set_lebesgue_integral lebesgue A (f w z)"
      using integrable[OF wz, of A] A
      by (intro set_lebesgue_integral_eq_integral(2) [symmetric]) auto
    also have "... = set_lebesgue_integral lborel A (f w z)"
      unfolding set_lebesgue_integral_def
      by (rule integral_completion) (use A in ‹auto simp: f_def›)
    finally show ?thesis .
  qed

  have ana: "(λw. F w z) analytic_on {w}" "(λz. F w z) analytic_on {z}"
       if wz: "Re w > 0" "Re z > 0" for w z
  proof -
    define a where "a = Re w / 2"
    define b where "b = Re z / 2"
    have ab: "a > 0" "b > 0" "a < Re w" "b < Re z"
      using wz by (auto simp: a_def b_def)

    define A :: "(complex × complex) set" where "A = {w. Re w > a} ×
{z. Re z > b}"
    have A: "open A" "(w,z) ∈ A"
      using ab by (auto simp: A_def open_Times open_halfspace_Re_gt)

    have lim: "uniform_limit A (λx (a,b). LBINT t:{x..1-x}. f a b t)
                (λ(a,b). LBINT t:{0..1}. f a b t) (at_right 0)"
      unfolding case_prod_unfold
    proof (intro uniform_limit_set_lebesgue_integral)
      have "(λt. t powr (a - 1) * (1 - t) powr (b - 1)) integrable_on
{0..1}"
        using integrable_Beta'[of a b] ab by simp
      hence "(λt. t powr (a - 1) * (1 - t) powr (b - 1)) absolutely_integrable_on
{0..1}"
        by (rule nonnegative_absolutely_integrable_1) auto
      thus "set_integrable lborel {0..1} (λt. t powr (a - 1) * (1 - t)
powr (b - 1))"
        by (simp add: set_integrable_def integrable_completion)
    next
      fix wz assume "wz ∈ A"
      then obtain w z where wz: "wz = (w, z)" "Re w > a" "Re z > b"
        by (auto simp: A_def)
```

```
      show "AE x∈{0..1} in lborel. norm (f (fst wz) (snd wz) x) ≤ x
powr (a - 1) * (1 - x) powr (b - 1)"
      proof (intro always_eventually impI allI)
        fix x :: real assume x: "x ∈ {0..1}"
        have "norm (f (fst wz) (snd wz) x) = x powr (Re w - 1) * (1 -
x) powr (Re z - 1)"
          using x wz by (simp add: f_def norm_mult norm_powr_complex del:
of_real_diff)
        also have "... ≤ x powr (a - 1) * (1 - x) powr (b - 1)"
          by (intro mult_mono powr_mono') (use x ab wz in auto)
        finally show "norm (f (fst wz) (snd wz) x) ≤ x powr (a - 1) *
(1 - x) powr (b - 1)" .
      qed
    next
      show "((λx. LBINT t:{x..1-x}. t powr (a - 1) * (1 - t) powr (b
- 1)) ⟶
              (LBINT t:{0..1}. t powr (a - 1) * (1 - t) powr (b - 1)))
(at_right 0)"
      proof (rule at_within.filterlim_set_lebesgue_integral_set)
        show "set_integrable lborel {0..1} (λt. t powr (a - 1) * (1 -
t) powr (b - 1))"
          using integrable_Beta[OF ab(1,2)] by simp
      next
        show "tendsto_set lborel (λx::real. {x..1 - x}) {0..1} (at_right
0)"
        proof (rule tendsto_set_intros)
          show "((λx::real. 1 - x) ⟶ 1) (at_right 0)"
            by real_asymp
        qed (auto intro: finite_imp_null_set_lborel)
      next
        show "∀F x in at_right 0. {x::real..1 - x} ⊆ {0..1}"
          using eventually_at_right_less by eventually_elim auto
      qed (auto simp: set_borel_measurable_def)
    next
      have "eventually (λx::real. x > 0) (at_right 0)"
        by real_asymp
      thus "∀F (x::real) in at_right 0. {x..1 - x} ∈ sets borel ∧ {x..1
- x} ⊆ {0..1}"
        by eventually_elim auto
    qed (auto simp: f_def set_borel_measurable_def)

    show "(λw. F w z) analytic_on {w}"
    proof -
      obtain R where R: "R > 0" "cball w R ⊆ {w. Re w > a}"
        using ab open_halfspace_Re_gt open_contains_cball by blast
      have "uniform_limit {w. Re w > a} (λx w. LBINT t:{x..1-x}. f w z
t)
              (λw. LBINT t:{0..1}. f w z t) (at_right 0)"
        using uniform_limit_compose'[OF lim, of "λw. (w,z)" "{w. Re w
```

```
> a}"] wz ab
        by (auto simp: Pi_def A_def)
      hence "uniform_limit (cball w R) (λx w. LBINT t:{x..1-x}. f w z
t)
              (λw. LBINT t:{0..1}. f w z t) (at_right 0)"
        by (rule uniform_limit_on_subset) fact
      also have "?this ⟷ uniform_limit (cball w R) (λx w. integral
{x..1-x} (f w z))
                            (λw. integral {0..1} (f w z)) (at_right 0)"
      proof (rule uniform_limit_cong)
        have "eventually (λy. y > 0) (at_right (0::real))"
          by real_asymp
        thus "∀F y in at_right 0. ∀x∈cball w R.
                set_lebesgue_integral lborel {y..1-y} (f x z) = integral
{y..1-y} (f x z)"
          by eventually_elim
             (intro ballI integral_eq [symmetric], use R ab in ‹auto simp:
subset_iff›)
      next
        show "set_lebesgue_integral lborel {0..1} (f x z) = integral
{0..1} (f x z)"
          if x: "x ∈ cball w R" for x
          by (subst integral_eq [symmetric])  (use wz x R ab in auto)
      qed
      finally have 1: ... .
      have 2: "(λw. integral {x..1-x} (f w z)) holomorphic_on cball w
R" if x: "x > 0" for x
      proof -
        note [derivative_intros] = has_field_derivative_complex_powr_right'
        define f' where "f' = (λx t. of_real (ln t) * f x z t)"
        have "(λw. integral (cbox x (1-x)) (f w z)) holomorphic_on cball
w R"
        proof (rule leibniz_rule_holomorphic)
          show "((λw. f w z t) has_field_derivative f' w' t) (at w' within
cball w R)"
            if "w' ∈ cball w R" "t ∈ cbox x (1 - x)" for w' t
            unfolding f_def f'_def using x that
            by (auto intro!: derivative_eq_intros simp: Ln_of_real)
        next
          show "f w' z integrable_on cbox x (1 - x)" if "w' ∈ cball w
R" for w'
            by (intro set_lebesgue_integral_eq_integral(1) integrable)
               (use that ab R x in auto)
        qed (use x in ‹auto simp: f_def f'_def case_prod_unfold intro!:
continuous_intros›)
        thus ?thesis
          by simp
      qed
      have "eventually (λx::real. x > 0) (at_right 0)"
```

```
      by real_asymp
    hence 3: "∀F n in at_right 0. continuous_on (cball w R)
            (λw. integral {n..1 - n} (f w z)) ∧
            (λw. integral {n..1 - n} (f w z)) holomorphic_on ball w
R"
      by eventually_elim
         (intro conjI holomorphic_on_imp_continuous_on holomorphic_on_subset[OF
2], auto)
    have "(λw. integral {0..1} (f w z)) holomorphic_on ball w R"
      using holomorphic_uniform_limit[OF 3 1] by auto
    thus "(λw. F w z) analytic_on {w}" unfolding F_def
      using ‹R > 0› analytic_at_ball by blast
  qed

  show "(λz. F w z) analytic_on {z}"
  proof -
    obtain R where R: "R > 0" "cball z R ⊆ {z. Re z > b}"
      using ab open_halfspace_Re_gt open_contains_cball by blast
    have "uniform_limit {z. Re z > b} (λx z. LBINT t:{x..1-x}. f w z
t)
            (λz. LBINT t:{0..1}. f w z t) (at_right 0)"
      using uniform_limit_compose'[OF lim, of "λz. (w,z)" "{z. Re z
> b}"] wz ab
      by (auto simp: Pi_def A_def)
    hence "uniform_limit (cball z R) (λx z. LBINT t:{x..1-x}. f w z
t)
            (λz. LBINT t:{0..1}. f w z t) (at_right 0)"
      by (rule uniform_limit_on_subset) fact
    also have "?this ⟷ uniform_limit (cball z R) (λx z. integral
{x..1-x} (f w z))
                        (λz. integral {0..1} (f w z)) (at_right 0)"
    proof (rule uniform_limit_cong)
      have "eventually (λy. y > 0) (at_right (0::real))"
        by real_asymp
      thus "∀F y in at_right 0. ∀x∈cball z R.
              set_lebesgue_integral lborel {y..1-y} (f w x) = integral
{y..1-y} (f w x)"
        by eventually_elim
           (intro ballI integral_eq [symmetric], use R ab in ‹auto simp:
subset_iff›)
    next
      show "set_lebesgue_integral lborel {0..1} (f w x) = integral
{0..1} (f w x)"
        if x: "x ∈ cball z R" for x
        by (subst integral_eq [symmetric])  (use wz x R ab in auto)
    qed
    finally have 1: ... .
    have 2: "(λz. integral {x..1-x} (f w z)) holomorphic_on cball z
R" if x: "x > 0" for x
```

```
      proof -
        note [derivative_intros] = has_field_derivative_complex_powr_right'
        define f' where "f' = (λx t. of_real (ln (1 - t)) * f w x t)"
        have "(λz. integral (cbox x (1-x)) (f w z)) holomorphic_on cball
z R"
        proof (rule leibniz_rule_holomorphic)
          show "((λz. f w z t) has_field_derivative f' z' t) (at z' within
cball z R)"
            if "z' ∈ cball z R" "t ∈ cbox x (1 - x)" for z' t
            unfolding f_def f'_def using x that
            by (auto intro!: derivative_eq_intros simp: Ln_of_real mult_ac
simp del: of_real_diff)
        next
          show "f w z' integrable_on cbox x (1 - x)" if "z' ∈ cball z
R" for z'
            by (intro set_lebesgue_integral_eq_integral(1) integrable)
               (use that ab R x in auto)
        qed (use x in ‹auto simp: f_def f'_def case_prod_unfold intro!:
continuous_intros›)
        thus ?thesis
          by simp
      qed
      have "eventually (λx::real. x > 0) (at_right 0)"
        by real_asymp
      hence 3: "∀F n in at_right 0. continuous_on (cball z R)
                (λz. integral {n..1 - n} (f w z)) ∧
                (λz. integral {n..1 - n} (f w z)) holomorphic_on ball z
R"
        by eventually_elim
           (intro conjI holomorphic_on_imp_continuous_on holomorphic_on_subset[OF
2], auto)
      have "(λz. integral {0..1} (f w z)) holomorphic_on ball z R"
        using holomorphic_uniform_limit[OF 3 1] by auto
      thus "(λz. F w z) analytic_on {z}" unfolding F_def
        using ‹R > 0› analytic_at_ball by blast
    qed
  qed

  have [holomorphic_intros]: "(λa. F a b) holomorphic_on {a. Re a > 0}"
if "Re b > 0" for b
    by (rule analytic_imp_holomorphic, subst analytic_on_analytic_at)

       (use that in ‹auto intro!: ana(1)›)
  have [holomorphic_intros]: "(λb. F a b) holomorphic_on {b. Re b > 0}"
if "Re a > 0" for a
    by (rule analytic_imp_holomorphic, subst analytic_on_analytic_at)

       (use that in ‹auto intro!: ana(2)›)
```

```
  have integrable: "f a b absolutely_integrable_on {0..1}" if ab: "Re
a > 0" "Re b > 0" for a b
  proof (rule set_integrable_bound)
    show "(λt. t powr (Re a - 1) * (1 - t) powr (Re b - 1)) absolutely_integrable_on
{0..1}"
      using integrable_Beta[of "Re a" "Re b"] ab
      by (simp add: set_integrable_def integrable_completion)
  qed (simp_all add: f_def norm_mult norm_powr_complex set_borel_measurable_def
measurable_completion
                del: of_real_diff)

  have 1: "F a b - Beta a b = 0" if ab: "a ∈ ℝ" "Re a > 0" "Re b > 0"
for a b
  proof (rule analytic_continuation[of "λz. F a z - Beta a z"])
    show "(λz. F a z - Beta a z) holomorphic_on {b. Re b > 0}"
      using that by (auto intro!: holomorphic_intros elim!: Reals_cases
nonpos_Ints_cases)
  next
    show "of_real 1 islimpt complex_of_real ` {0<..}"
      by (rule islimpt_isCont_image) (auto simp: eventually_at_filter
intro: open_imp_islimpt)
  next
    fix z assume "z ∈ complex_of_real ` {0<..}"
    then obtain y where y: "z = of_real y" "y > 0"
      by auto
    from ab obtain x where x: "a = of_real x" "x > 0"
      by (auto elim!: Reals_cases)
    have "((λt. complex_of_real (t powr (x - 1) * (1 - t) powr (y - 1)))
            has_integral (of_real (Beta x y))) {0..1}"
      by (intro has_integral_of_real has_integral_Beta_real x y)
    also have "?this ⟷ (f (of_real x) (of_real y) has_integral (of_real
(Beta x y))) {0..1}"
      by (intro has_integral_cong) (auto simp: f_def powr_Reals_eq)
    also have "complex_of_real (Beta x y) = Beta (of_real x) (of_real
y)"
      by (simp add: Beta_complex_of_real)
    finally show "F a z - Beta a z = 0"
      using x y by (simp add: F_def has_integral_iff)
  qed (use ab in ‹auto simp: open_halfspace_Re_gt connected_halfspace_Re_gt›)

  have 2: "F a b - Beta a b = 0" if ab: "Re a > 0" "Re b > 0" for a b
  proof (rule analytic_continuation[of "λz. F z b - Beta z b"])
    show "(λz. F z b - Beta z b) holomorphic_on {a. Re a > 0}"
      using that by (auto intro!: holomorphic_intros elim!: Reals_cases
nonpos_Ints_cases)
  next
    show "of_real 1 islimpt complex_of_real ` {0<..}"
      by (rule islimpt_isCont_image) (auto simp: eventually_at_filter
intro: open_imp_islimpt)
```

```
  next
    fix z assume "z ∈ complex_of_real ` {0<..}"
    thus "F z b - Beta z b = 0"
      using 1[of z b] ab by auto
  qed (use ab in ‹auto simp: open_halfspace_Re_gt connected_halfspace_Re_gt›)

  have "f a b integrable_on {0..1}"
    using assms integrable[of a b] set_lebesgue_integral_eq_integral(1)
by blast
  hence "(f a b has_integral F a b) {0..1}"
    by (simp add: has_integral_iff F_def)
  also have "F a b = Beta a b"
    using 2[of a b] assms by simp
  finally show "((λt. of_real t powr (a - 1) * of_real (1 - t) powr (b
- 1))
                  has_integral Beta a b) {0<..<1}"
    by (simp add: f_def has_integral_Icc_iff_Ioo)
qed
```

By change of variables, we can also derive the following integral:

$$\frac{1}{2}B(a,b) = \int_0^{\frac{\pi}{2}} \sin^{2a-1} t \cos^{2b-1} t \,\mathrm{d}t$$

```
lemma has_integral_Beta_sin_cos_complex:
  assumes "Re a > 0" "Re b > 0"
  shows "(λt. of_real (sin t) powr (2*a-1) * of_real (cos t) powr (2*b-1))
          absolutely_integrable_on {0..pi/2}" (is "?thesis1")
    and "((λt. of_real (sin t) powr (2*a-1) * of_real (cos t) powr (2*b-1))
          has_integral (Beta a b / 2)) {0..pi/2}" (is "?thesis2")
proof -
  define f where "f = (λt. (1/2) * (of_real t powr (a-1) * of_real (1
- t) powr (b-1)))"
  define I where "I = (1/2) * Beta a b"
  define g where "g = (λt. sin t ^ 2 :: real)"
  define g' where "g' = (λt. 2 * sin t * cos t :: real)"
  define h where "h = (λt. of_real (sin t) powr (2*a-1) * of_real (cos
t) powr (2*b-1))"

  have *: "sqrt x ≥ (-1::real)" if "x ≥ 0" for x
    by (rule order.trans[of _ 0]) (use that in auto)
  have **: "arcsin (sqrt x) * 2 ≤ pi" if "x ∈ {0..1}" for x :: real
    using arcsin_bounded[of "sqrt x"] that *[of x] by simp
  have bij: "bij_betw g {0..pi/2} {0..1}"
    by (rule bij_betwI[of _ _ _ "λx. arcsin (sqrt x)"])
       (auto simp: g_def power_le_one_iff sin_ge_zero arcsin_nonneg arcsin_sin
* **)

  have eq: "|g' t| *R f (g t) = h t"
```

```
    if t: "t ∈ {0..pi/2}" for t
  proof -
    have "|g' t| *R f (g t) = of_real (sin t * cos t) *
              of_real (sin t ^ 2) powr (a - 1) * of_real (cos t ^ 2) powr
(b - 1)" using t
      by (simp add: g'_def f_def g_def scaleR_conv_of_real
                    abs_mult sin_ge_zero cos_ge_zero cos_squared_eq flip:
of_real_power)
    also have "... = of_real (sin t) powr ((a-1) + (a-1) + 1) * of_real
(cos t) powr ((b-1) + (b-1) + 1)"
      unfolding powr_add using t
      by (simp add: power2_eq_square powr_times_real sin_ge_zero cos_ge_zero)
    finally show ?thesis
      by (simp add: algebra_simps h_def)
  qed

  have "f absolutely_integrable_on (g ` {0..pi/2})"
    unfolding f_def using has_integral_Beta_complex[of a b] bij assms
    by (intro set_integrable_mult_right)
       (simp add: bij_betw_def f_def I_def absolutely_integrable_on_Icc_iff_Ioo
has_integral_iff)
  moreover have "integral (g ` {0..pi/2}) f = I"
    unfolding f_def using has_integral_Beta_complex[of a b] bij assms
    by (subst integral_mult_right)
       (simp add: bij_betw_def f_def I_def integral_open_interval_real
has_integral_iff)
  ultimately have "f absolutely_integrable_on (g ` {0..pi/2}) ∧ integral
(g ` {0..pi/2}) f = I"
    by blast
  also have "?this ⟷ (λx. |g' x| *R f (g x)) absolutely_integrable_on
{0..pi/2} ∧
                       integral {0..pi/2} (λx. |g' x| *R f (g x)) = I"
    by (subst eq_commute, rule has_absolute_integral_change_of_variables_real)
       (use bij in ‹auto simp: g_def g'_def bij_betw_def intro!: derivative_eq_intros›)
  also have "(λx. |g' x| *R f (g x)) absolutely_integrable_on {0..pi/2}
⟷
            h absolutely_integrable_on {0..pi/2}"
    by (rule set_integrable_cong) (use eq in auto)
  also have "integral {0..pi/2} (λx. |g' x| *R f (g x)) = integral {0..pi/2}
h"
    by (rule integral_cong) (use eq in auto)
  finally show ?thesis1 ?thesis2
    unfolding h_def I_def by (simp_all add: has_integral_iff set_lebesgue_integral_eq_integ
qed

lemma has_integral_Beta_sin_cos_real:
  assumes "a > 0" "b > 0"
  shows "((λt. sin t powr (2*a-1) * cos t powr (2*b-1)) has_integral
(Beta a b / 2)) {0..pi/2}"
```

```
proof -
  have "((λt. Re (of_real (sin t) powr (2 * of_real a - 1) * of_real (cos
t) powr (2 * of_real b - 1)))
           has_integral (Re (Beta (of_real a) (of_real b) / 2))) {0..pi/2}"
    by (intro has_integral_Re has_integral_Beta_sin_cos_complex) (use
assms in auto)
  also have "?this ⟷ ((λt. sin t powr (2 * a - 1) * cos t powr (2 *
b - 1)) has_integral
                        (Re (Beta (of_real a) (of_real b) / 2))) {0..pi/2}"
    by (intro has_integral_cong)
       (simp_all flip: sin_of_real cos_of_real add: powr_Reals_eq sin_ge_zero
cos_ge_zero)
  also have "Re (Beta (of_real a) (of_real b) / 2) = Beta a b / 2"
    by (subst Beta_complex_of_real) auto
  finally show ?thesis .
qed

lemma sin_power_integral_0_pi_half:
  "((λt. sin t ^ n) has_integral (Beta ((real n + 1) / 2) (1/2)) / 2)
{0..pi/2}"
proof -
  have "((λt. sin t powr real n * (if cos t = 0 then 0 else 1))
          has_integral Beta ((real n + 1) / 2) (1 / 2) / 2) {0..pi/2}"
    using has_integral_Beta_sin_cos_real[of "real (n+1) / 2" "1 / 2"]

    by (simp add: add_divide_distrib add_ac)
  also have "?this ⟷ ((λt. sin t ^ n) has_integral Beta ((real n +
1) / 2) (1 / 2) / 2) {0..pi/2}"
  proof (rule has_integral_spike_eq)
    fix t assume t: "t ∈ {0..pi/2} - {0, pi/2}"
    have "sin t > 0" "cos t > 0"
      using t by (auto simp: sin_gt_zero cos_gt_zero)
    thus "sin t ^ n = sin t powr real n * (if cos t = 0 then 0 else 1)"
      by (auto simp: powr_realpow)
  qed auto
  finally show ?thesis
    by simp
qed

lemma cos_power_integral_0_pi_half:
  "((λt. cos t ^ n) has_integral (Beta (1/2) ((real n + 1) / 2)) / 2)
{0..pi/2}"
proof -
  have "((λt. cos t powr real n * (if sin t = 0 then 0 else 1))
          has_integral Beta (1 / 2) ((real n + 1) / 2) / 2) {0..pi/2}"
    using has_integral_Beta_sin_cos_real[of  "1 / 2" "real (n+1) / 2"]

    by (simp add: add_divide_distrib add_ac mult_ac)
  also have "?this ⟷ ((λt. cos t ^ n) has_integral Beta (1 / 2) ((real
```

```
n + 1) / 2) / 2) {0..pi/2}"
  proof (rule has_integral_spike_eq)
    fix t assume t: "t ∈ {0..pi/2} - {0, pi/2}"
    have "cos t > 0" "sin t > 0"
      using t by (auto simp: sin_gt_zero cos_gt_zero)
    thus "cos t ^ n = cos t powr real n * (if sin t = 0 then 0 else 1)"
      by (auto simp: powr_realpow)
  qed auto
  finally show ?thesis
    by simp
qed

lemma sin_power_even_integral_0_pi_half_real:
  "((λt. sin t ^ (2*n)) has_integral (pi / 2 * fact (2 * n) / (fact n
^ 2 * 4 ^ n))) {0..pi/2}"
proof -
  have "((λt. sin t ^ (2*n)) has_integral (Beta ((real (2*n) + 1) / 2)
(1/2)) / 2) {0..pi/2}"
    by (rule sin_power_integral_0_pi_half)
  also have "Beta ((real (2*n) + 1) / 2) (1/2) = Gamma (real n + 1 / 2)
* sqrt pi / fact n"
    by (simp add: Beta_def Gamma_one_half_real add_divide_distrib Gamma_fact)
  also have "... / 2 = pi / 2 * fact (2 * n) / (fact n ^ 2 * 4 ^ n)"
    by (subst Gamma_int_plus_half) (auto simp: algebra_simps power2_eq_square)
  finally show ?thesis .
qed

lemma cos_power_even_integral_0_pi_half_real:
  "((λt. cos t ^ (2*n)) has_integral (pi / 2 * fact (2 * n) / (fact n
^ 2 * 4 ^ n))) {0..pi/2}"
proof -
  have "((λt. cos t ^ (2*n)) has_integral (Beta (1/2) ((real (2*n) + 1)
/ 2)) / 2) {0..pi/2}"
    by (rule cos_power_integral_0_pi_half)
  also have "Beta (1/2) ((real (2*n) + 1) / 2) = Gamma (real n + 1 / 2)
* sqrt pi / fact n"
    by (simp add: Beta_def Gamma_one_half_real add_divide_distrib Gamma_fact)
  also have "... / 2 = pi / 2 * fact (2 * n) / (fact n ^ 2 * 4 ^ n)"
    by (subst Gamma_int_plus_half) (auto simp: algebra_simps power2_eq_square)
  finally show ?thesis .
qed
```

Lastly, we derive the following integral:

$$B(a,b) = \int_0^\infty \frac{t^{a-1}}{(1+t)^{a+b}}\,\mathrm{d}t$$

```
lemma has_integral_Beta_0_infinity_complex:
  assumes "Re a > 0" "Re b > 0"
  shows "(λt. of_real t powr (a - 1) / of_real (1 + t) powr (a + b))
```

```
          absolutely_integrable_on {0<..}" (is "?thesis1")
    and "((λt. of_real t powr (a - 1) / of_real (1 + t) powr (a + b))
          has_integral (Beta a b)) {0<..}" (is "?thesis2")
proof -
  define f where "f = (λt. of_real t powr (a - 1) / of_real (1 + t) powr
(a + b))"
  define g where "g = (λx. tan x ^ 2 :: real)"
  define g' where "g' = (λx. 2 * tan x / cos x ^ 2 :: real)"
  define I where "I = Beta a b"
  define h where "h = (λx. 2 * (of_real (sin x) powr (2 * a - 1) * of_real
(cos x) powr (2 * b - 1)))"

  have bij: "bij_betw g {0<..<pi/2} {0<..}"
  proof (rule bij_betwI[of _ _ _ "λt. arctan (sqrt t)"])
    have "tan x ≠ 0" if "x ∈ {0<..<pi/2}" for x :: real
      using tan_gt_zero[of x] that by auto
    thus "g ∈ {0<..<pi / 2} → {0<..}"
      by (auto simp: g_def)
  next
    have "arctan (sqrt x) * 2 < pi" if "x > 0" for x
      using arctan_bounded[of "sqrt x"] that by auto
    thus "(λt. arctan (sqrt t)) ∈ {0<..} → {0<..<pi / 2}"
      by auto
  next
    show "arctan (sqrt (g x)) = x" if "x ∈ {0<..<pi/2}" for x
      using that tan_gt_zero[of x] by (auto simp: g_def arctan_tan)
  next
    show "g (arctan (sqrt y)) = y" if "y ∈ {0<..}" for y
      using that by (auto simp: g_def tan_arctan)
  qed

  have eq: "|g' x| *_R f (g x) = h x" if x: "x ∈ {0<..<pi/2}" for x
  proof -
    define s where "s = ln (sin x)"
    define c where "c = ln (cos x)"
    have sc: "sin x = exp s" "cos x = exp c"
      using x sin_gt_zero[of x] cos_gt_zero[of x] by (simp_all add: s_def
c_def)
    have "|g' x| *_R f (g x) =
           2 * of_real (tan x) * of_real (tan x ^ 2) powr (a - 1) /
             (of_real (cos x ^ 2) * of_real (1 + tan x ^ 2) powr (a +
b))"
      using x by (simp add: g'_def f_def g_def scaleR_conv_of_real tan_pos_pi2_le)
    also have "of_real (cos x ^ 2) * of_real (1 + tan x ^ 2) powr (a +
b) =
                 exp (2 * (1 - a - b) * c)"
      using exp_double[of "complex_of_real c"]
      by (subst tan_sec)
         (auto simp: sc powr_def Ln_Reals_eq ring_distribs exp_add exp_diff
```

```
exp_minus
                    field_simps ln_div ln_realpow exp_of_real)
    also have "2 * of_real (tan x) * of_real ((tan x)^2) powr (a - 1) =
               2 * exp ((2 * a - 1) * s) * exp ((1 - 2 * a) * c)"
      using x cos_gt_zero[of x] sin_gt_zero[of x]
            exp_double[of "complex_of_real s"] exp_double[of "complex_of_real
c"]
      by (simp add: sc tan_def powr_def Ln_Reals_eq ring_distribs ln_realpow
ln_div
                    exp_diff exp_add exp_minus field_simps exp_of_real
power2_eq_square ln_mult)
    also have "2 * exp ((2 * a - 1) * s) * exp ((1 - 2 * a) * c) / exp
(2 * (1 - a - b) * c) =
               2 * exp ((2 * a - 1) * s) * exp ((2 * b - 1) * c)"
      by (simp add: field_simps exp_add exp_diff exp_minus flip: power2_eq_square
exp_double)
    also have "exp ((2 * a - 1) * s) = of_real (sin x) powr (2 * a - 1)"
      using sin_gt_zero[of x] x by (auto simp: powr_def s_def Ln_Reals_eq)
    also have "exp ((2 * b - 1) * c) = of_real (cos x) powr (2 * b - 1)"
      using cos_gt_zero[of x] x by (auto simp: powr_def c_def Ln_Reals_eq)
    finally show ?thesis
      by (simp add: h_def)
  qed

  have cos_nz: "cos x ≠ 0" if "x ∈ {0<..<pi/2}" for x :: real
    using cos_gt_zero[of x] that by simp

  have "h absolutely_integrable_on {0<..<pi/2}"
    using set_integrable_mult_right[OF has_integral_Beta_sin_cos_complex(1)[of
a b], of 2] assms
    by (simp add: h_def absolutely_integrable_on_Icc_iff_Ioo)
  moreover have "integral {0<..<pi/2} h = I"
    using has_integral_mult_right[OF has_integral_Beta_sin_cos_complex(2)[of
a b], of 2] assms
    unfolding has_integral_Icc_iff_Ioo by (simp add: h_def I_def has_integral_iff)
  ultimately have "h absolutely_integrable_on {0<..<pi/2} ∧ integral {0<..<pi/2}
h = I"
    by blast

  also have "h absolutely_integrable_on {0<..<pi/2} ⟷
               (λx. |g' x| *_R f (g x)) absolutely_integrable_on {0<..<pi/2}"

    by (rule set_integrable_cong) (use eq in auto)
  also have "integral {0<..<pi/2} h = integral {0<..<pi/2} (λx. |g' x|
*_R f (g x))"
    by (rule integral_cong) (use eq in auto)
  also have "(λx. |g' x| *_R f (g x)) absolutely_integrable_on {0<..<pi/2}
∧
             integral {0<..<pi/2} (λx. |g' x| *_R f (g x)) = I ⟷
```

```
        f absolutely_integrable_on (g ‘ {0<..<pi/2}) ∧ integral (g ‘ {0<..<pi/2})
f = I"
    using bij
    by (intro has_absolute_integral_change_of_variables_real)
       (auto simp: g_def g'_def field_simps bij_betw_def cos_nz intro!:
derivative_eq_intros)
  also have "g ‘ {0<..<pi/2} = {0<..}"
    using bij by (simp add: bij_betw_def)
  finally show ?thesis1 ?thesis2
    unfolding f_def I_def by (simp_all add: has_integral_iff set_lebesgue_integral_eq_integ
qed

lemma has_integral_Beta_0_infinity_real:
  assumes "a > 0" "b > (0::real)"
  shows "((λt. t powr (a - 1) / (1 + t) powr (a + b)) has_integral (Beta
a b)) {0<..}"
proof -
  have "((λt. Re (of_real t powr (of_real a - 1) / of_real (1 + t) powr
(of_real a + of_real b)))
          has_integral (Re (Beta (of_real a) (of_real b)))) {0<..}"
    by (intro has_integral_Re has_integral_Beta_0_infinity_complex) (use
assms in auto)
  also have "?this ⟷ ((λt. t powr (a - 1) / (1 + t) powr (a + b)) has_integral

                        (Re (Beta (of_real a) (of_real b)))) {0<..}"
    by (intro has_integral_cong)
       (simp_all flip: sin_of_real cos_of_real add: powr_Reals_eq sin_ge_zero
cos_ge_zero)
  also have "Re (Beta (of_real a) (of_real b)) = Beta a b"
    by (subst Beta_complex_of_real) auto
  finally show ?thesis .
qed

end
```

4 A proof method for computing derivatives

```
theory Derivative_Method
  imports Complex_Main
begin



ML ‹

signature DERIVATIVE_METHOD =
sig

val tac : bool -> Proof.context -> int -> tactic
```

```
end

structure Derivative_Method : DERIVATIVE_METHOD =
struct

fun is_deriv_prop prop = (
  case head_of (HOLogic.dest_Trueprop (Logic.strip_imp_concl prop)) of
    Const (const_name‹has_derivative›, _) => true
  | Const (const_name‹has_field_derivative›, _) => true
  | Const (const_name‹has_vector_derivative›, _) => true
  | _ => false
  ) handle TERM _ => false

fun tac nofail ctxt =
  let
    val eq_thms =
      @{thms has_derivative_eq_rhs[rotated]
             DERIV_cong[rotated]
             has_vector_derivative_eq_rhs[rotated]}
    val intros = Named_Theorems.get ctxt named_theorems‹derivative_intros›
    val MYTRY = if nofail then TRY else I
    fun tac' (t, i) =
      if is_deriv_prop t then MYTRY (
        (resolve_tac ctxt intros
         THEN_ALL_NEW SUBGOAL tac') i)
      else all_tac
    fun tac i =
      resolve_tac ctxt eq_thms i
      THEN SUBGOAL tac' (i + 1)
  in
    tac
  end

val args_parser =
  Scan.lift (Scan.optional (Args.parens (Args.$$$ "nofail") >> K true)
false) --|
  Method.sections [
    Args.add -- Args.colon >>
      K (Method.modifier (Named_Theorems.add named_theorems‹derivative_intros›)
here),
    Args.del -- Args.colon >>
      K (Method.modifier (Named_Theorems.del named_theorems‹derivative_intros›)
here)
  ]

val method = args_parser >> (SIMPLE_METHOD' oo tac)

val _ =
```

```
  Theory.setup (Method.setup binding‹derivative› method "automation for
computing derivatives")

end

›

end
```

5 Auxiliary material

```
theory Incomplete_Gamma_Lemma_Bucket
  imports
  "HOL-Complex_Analysis.Complex_Analysis"
  Safe_Power
  More_Dominated_Convergence
  Derivative_Method
begin

lemma sgn_sqrt [simp]: "sgn (sqrt x) = sgn x"
  by (auto simp: sgn_if)


lemma countably_generated_imp_convergent_sequence:
  assumes "countably_generated_filter F" "F ≠ bot"
  obtains X where "filterlim X F sequentially"
proof -
  obtain B where B: "antimono_on UNIV B" "⋀P. eventually P F = (∃i::nat.
∀x∈B i. P x)"
    using countably_generated_filter_iff_decseq[of F] assms by blast
  have "B i ≠ {}" for i
    using B(2)[of "λ_. False"] assms by (auto simp: trivial_limit_def)
  hence "∀i. ∃x. x ∈ B i"
    by blast
  then obtain X where X: "X i ∈ B i" for i
    by metis
  have "filterlim X F sequentially"
    unfolding filterlim_def le_filter_def eventually_filtermap
  proof safe
    fix P assume "eventually P F"
    with B obtain i where i: "∀x∈B i. P x"
      by blast
    hence "P (X j)" if j: "j ≥ i" for j
      using monotone_onD[OF B(1) _ _ j] X[of j] by auto
    thus "eventually (λj. P (X j)) sequentially"
      by (auto simp: eventually_at_top_linorder)
  qed
  thus ?thesis
    by (rule that)
```

qed

lemma (**in** *sequential_filter*) *dominated_convergence'*:
fixes *f* :: "'*a* ⇒ '*n*::*euclidean_space* ⇒ '*m*::*euclidean_space*"
assumes *f*: "*eventually* (λ*k*. (*f k*) *integrable_on S*) *F*" **and** *h*: "*h integrable_on S*"
and *le*: "*eventually* (λ*k*. ∀*x*∈*S*. *norm* (*f k x*) ≤ *h x*) *F*"
and *conv*: "⋀*x*. *x* ∈ *S* ⟹ ((λ*k*. *f k x*) ⟶ *g x*) *F*"
shows "((λ*k*. *integral S* (*f k*)) ⟶ *integral S g*) *F*"
proof (*rule filterlim_sequentially_imp_filterlim*)
fix *X* **assume** *X*: "*filterlim X F sequentially*"
have "*eventually* (λ*k*. (∀*x*∈*S*. *norm* (*f k x*) ≤ *h x*) ∧ *f k integrable_on S*) *F*"
using *f le* **by** *eventually_elim auto*
hence "*eventually* (λ*k*. (∀*x*∈*S*. *norm* (*f* (*X k*) *x*) ≤ *h x*) ∧ *f* (*X k*) *integrable_on S*) *sequentially*"
by (*rule eventually_compose_filterlim*) *fact*
then obtain *N* **where** *N*: "⋀*k*. *k* ≥ *N* ⟹ (∀*x*∈*S*. *norm* (*f* (*X k*) *x*) ≤ *h x*) ∧ *f* (*X k*) *integrable_on S*"
by (*auto simp*: *eventually_at_top_linorder*)
define *X'* **where** "*X'* = *X* ∘ ((+) *N*)"
have *X'*: "∀*x*∈*S*. *norm* (*f* (*X'* *k*) *x*) ≤ *h x*" "*f* (*X'* *k*) *integrable_on S*" **for** *k*
using *N*[*of* "*N* + *k*"] **by** (*simp_all add*: *X'_def*)
have *lim_X'*: "*filterlim X' F sequentially*" **unfolding** *X'_def o_def*
by (*rule filterlim_compose*[*OF X*])
(*use filterlim_add_const_nat_at_top*[*of N*] **in** ‹*simp_all add*: *add_ac*›)

have "(λ*n*. *integral S* (*f* (*X'* *n*))) ⟶ *integral S g*"
proof (*rule dominated_convergence*)
show "*h integrable_on S*"
by *fact*
show "*f* (*X'* *n*) *integrable_on S*" **for** *n*
by (*rule X'*)
show "*norm* (*f* (*X'* *k*) *x*) ≤ *h x*" **if** "*x* ∈ *S*" **for** *x k*
using *X'*[*of k*] *that* **by** *auto*
show "(λ*n*. *f* (*X'* *n*) *x*) ⟶ *g x*" **if** "*x* ∈ *S*" **for** *x*
by (*rule filterlim_compose*[*OF conv lim_X'*]) *fact*
qed
hence "(λ*n*. *integral S* (*f* (*X'* (*n* - *N*)))) ⟶ *integral S g*"
by (*rule filterlim_compose*) (*rule filterlim_minus_const_nat_at_top*)
also have "?*this* ⟷ (λ*n*. *integral S* (*f* (*X n*))) ⟶ *integral S g*"
proof (*rule filterlim_cong*)
show "*eventually* (λ*n*. *integral S* (*f* (*X'* (*n* - *N*))) = *integral S* (*f* (*X n*))) *at_top*"
using *eventually_ge_at_top*[*of N*] **by** *eventually_elim* (*auto simp*: *X'_def*)
qed *auto*

```
  finally show "(λn. integral S (f (X n))) ⟶ integral S g" .
qed


lemma dominated_convergence_countably_generated_filter:
  fixes f :: "'a ⇒ 'n::euclidean_space ⇒ 'm::euclidean_space"
  assumes cg: "countably_generated_filter F"
  assumes f: "eventually (λk. (f k) integrable_on S) F" and h: "h integrable_on
S"
    and le: "eventually (λk. ∀x∈S. norm (f k x) ≤ h x) F"
    and conv: "⋀x. x ∈ S ⟹ ((λk. f k x) ⟶ g x) F"
  shows "F ≠ bot ⟹ g integrable_on S" "((λk. integral S (f k)) ⟶
integral S g) F"
proof -
  interpret sequential_filter F
    by (rule countably_generated_filter_imp_sequential_filter) fact
  show "((λk. integral S (f k)) ⟶ integral S g) F"
    using f h le conv by (rule dominated_convergence')

  assume "F ≠ bot"
  with cg obtain X where X: "filterlim X F sequentially"
    using countably_generated_imp_convergent_sequence by blast
  have "eventually (λk. (∀x∈S. norm (f k x) ≤ h x) ∧ f k integrable_on
S) F"
    using f le by eventually_elim auto
  hence "eventually (λk. (∀x∈S. norm (f (X k) x) ≤ h x) ∧ f (X k) integrable_on
S) sequentially"
    by (rule eventually_compose_filterlim) fact
  then obtain N where N: "⋀k. k ≥ N ⟹ (∀x∈S. norm (f (X k) x) ≤
h x) ∧ f (X k) integrable_on S"
    by (auto simp: eventually_at_top_linorder)
  define X' where "X' = X ∘ ((+) N)"
  have X': "∀x∈S. norm (f (X' k) x) ≤ h x" "f (X' k) integrable_on S"
for k
    using N[of "N + k"] by (simp_all add: X'_def)
  have lim_X': "filterlim X' F sequentially" unfolding X'_def o_def
    by (rule filterlim_compose[OF X])
       (use filterlim_add_const_nat_at_top[of N] in ‹simp_all add: add_ac›)

  show "g integrable_on S"
  proof (rule dominated_convergence)
    show "h integrable_on S"
      by fact
    show "f (X' n) integrable_on S" for n
      by (rule X')
    show "norm (f (X' k) x) ≤ h x" if "x ∈ S" for x k
      using X'[of k] that by auto
    show "(λn. f (X' n) x) ⟶ g x" if "x ∈ S" for x
      by (rule filterlim_compose[OF conv lim_X']) fact
```

```
  qed
qed
```

```
lemma nonpos_Reals_real_eq: "ℝ≤0 = {..(0::real)}"
  by (auto simp: nonpos_Reals_def)
```

```
lemma has_integral_complex_of_real_iff:
  "((λx. of_real (f x) :: complex) has_integral (of_real I)) A ⟷ (f
has_integral I) A"
proof
  assume "((λx. of_real (f x) :: complex) has_integral (of_real I)) A"
  thus "(f has_integral I) A"
    by (subst (asm) has_integral_componentwise_iff) (auto simp: Basis_complex_def)
qed (auto intro: has_integral_of_real)
```

```
lemma integrable_on_complex_of_real_iff:
  "(λx. of_real (f x) :: complex) integrable_on A ⟷ f integrable_on
A"
  using has_integral_complex_of_real_iff[of f _ A]
  by (metis (lifting) ext Re_complex_of_real complex_inner_1_right integrable_component
      integrable_on_def)
```

```
lemma integral_complex_of_real:
  "integral A (λx. complex_of_real (f x)) = of_real (integral A f)"
proof (cases "f integrable_on A")
  case True
  thus ?thesis
    using integral_linear[of f A complex_of_real]
    by (auto simp: bounded_linear_of_real o_def)
next
  case False
  hence "¬(λx. complex_of_real (f x)) integrable_on A"
    by (subst integrable_on_complex_of_real_iff)
  with False show ?thesis
    by (simp add: not_integrable_integral)
qed
```

```
lemma countable_nonpos_Ints [intro]: "countable ℤ≤0"
  by (rule countable_subset[of _ "ℤ"]) (auto intro: countable_int)
```

lemma
fixes *f :: "_ ⇒ _ ⇒ 'a :: {banach, second_countable_topology}"*
assumes *integrable: "⋀i. set_integrable M A (f i)"*
and *summable: "AE x∈A in M. summable (λi. norm (f i x))"*
and *sums: "summable (λi. (∫ x∈A. norm (f i x) ∂M))"*
shows *set_integrable_suminf: "set_integrable M A (λx. (∑ i. f i x))"*
and *sums_set_integral: "(λi. set_lebesgue_integral M A (f i)) sums (∫ x∈A. (∑ i. f i x) ∂M)"*
and *set_integral_suminf: "(∫ x∈A. (∑ i. f i x) ∂M) = (∑ i. set_lebesgue_integral M A (f i))"*
and *summable_set_integral: "summable (λi. set_lebesgue_integral M A (f i))"*
proof -
have *1: "integrable M (λx. indicat_real A x *$_R$ f i x)"* **for** *i*
using *integrable[of i]* **unfolding** *set_integrable_def* .
have *2: "AE x in M. summable (λi. norm (indicat_real A x *$_R$ f i x))"*
using *summable* **by** *eventually_elim auto*
have *3: "summable (λi. lebesgue_integral M (λx. norm (indicat_real A x *$_R$ f i x)))"*
using *sums* **unfolding** *set_lebesgue_integral_def* **by** *simp*

have *"integrable M (λx. (∑ i. indicator A x *$_R$ f i x))"*
using *1 2 3* **by** *(rule integrable_suminf)*
also have *"(λx. (∑ i. indicator A x *$_R$ f i x)) = (λx. indicator A x *$_R$ (∑ i. f i x))"*
by *(auto simp: indicator_def)*
finally show *"set_integrable M A (λx. (∑ i. f i x))"*
unfolding *set_integrable_def* .

have *"(λi. set_lebesgue_integral M A (f i)) sums (∫ x. (∑ i. indicator A x *$_R$ f i x) ∂M)"*
unfolding *set_lebesgue_integral_def* **using** *1 2 3* **by** *(rule sums_integral)*
also have *"(∫ x. (∑ i. indicator A x *$_R$ f i x) ∂M) = (∫ x. indicator A x *$_R$ (∑ i. f i x) ∂M)"*
by *(rule Bochner_Integration.integral_cong) (simp_all add: indicator_def)*
finally show *"(λi. set_lebesgue_integral M A (f i)) sums (∫ x∈A. (∑ i. f i x) ∂M)"*
unfolding *set_lebesgue_integral_def* .

have *"(∫ x∈A. ∑ i. f i x∂M) = (∫ x. (∑ i. indicator A x *$_R$ f i x) ∂M)"*
unfolding *set_lebesgue_integral_def*
by *(rule Bochner_Integration.integral_cong) (auto simp: indicator_def)*
also have *"... = (∑ i. (∫ x∈A. f i x ∂M))"*
unfolding *set_lebesgue_integral_def* **using** *1 2 3* **by** *(rule integral_suminf)*
finally show *"(∫ x∈A. (∑ i. f i x) ∂M) = (∑ i. set_lebesgue_integral M A (f i))"* .

show *"summable (λi. set_lebesgue_integral M A (f i))"*
unfolding *set_lebesgue_integral_def* **using** *1 2 3* **by** *(rule summable_integral)*

qed

lemma *leibniz_rule_field_derivative_real:*
fixes *f::"'a::{real_normed_field, banach}* ⇒ *real* ⇒ *'a"*
assumes *fx: "*⋀*x t. x* ∈ *U* ⟹ *t* ∈ *{a..b}* ⟹ *((*λ*x. f x t) has_field_derivative fx x t) (at x within U)"*
assumes *integrable_f2: "*⋀*x. x* ∈ *U* ⟹ *(f x) integrable_on {a..b}"*
assumes *cont_fx: "continuous_on (U* × *{a..b}) (*λ*(x, t). fx x t)"*
assumes *U: "x0* ∈ *U" "convex U"*
shows *"((*λ*x. integral {a..b} (f x)) has_field_derivative integral {a..b} (fx x0)) (at x0 within U)"*
using *leibniz_rule_field_derivative[of U a b f fx x0] assms* **by** *simp*

lemma *convex_minkowski_sum_ball:*
fixes *A :: "'a :: euclidean_space set"*
assumes *"convex A"*
shows *"convex (*⋃*x*∈*A. ball x r)"*
proof -
have *"convex (*⋃*x*∈*A.* ⋃*y*∈*ball 0 r. {x + y})"*
by *(rule convex_sums) (use assms* **in** *auto)*
also have *"(*⋃*x*∈*A.* ⋃*y*∈*ball 0 r. {x + y}) = (*⋃*x*∈*A. ball x r)"*
proof *(intro equalityI subsetI)*
fix *z* **assume** *"z* ∈ *(*⋃*x*∈*A.* ⋃*y*∈*ball 0 r. {x + y})"*
thus *"z* ∈ *(*⋃*x*∈*A. ball x r)"*
by *(force simp: dist_norm)*
next
fix *z* **assume** *"z* ∈ *(*⋃*x*∈*A. ball x r)"*
then obtain *x* **where** *"x* ∈ *A" "z* ∈ *ball x r"*
by *auto*
hence *"z = x + (z - x)" "z - x* ∈ *ball 0 r"*
by *(auto simp: dist_norm)*
thus *"z* ∈ *(*⋃*x*∈*A.* ⋃*y*∈*ball 0 r. {x + y})"*
using ‹*x* ∈ *A*› **by** *blast*
qed
finally show *?thesis* .
qed

lemma *convex_minkowski_sum_cball:*
fixes *A :: "'a :: euclidean_space set"*
assumes *"convex A"*
shows *"convex (*⋃*x*∈*A. cball x r)"*
proof -
have *"convex (*⋃*x*∈*A.* ⋃*y*∈*cball 0 r. {x + y})"*
by *(rule convex_sums) (use assms* **in** *auto)*
also have *"(*⋃*x*∈*A.* ⋃*y*∈*cball 0 r. {x + y}) = (*⋃*x*∈*A. cball x r)"*
proof *(intro equalityI subsetI)*
fix *z* **assume** *"z* ∈ *(*⋃*x*∈*A.* ⋃*y*∈*cball 0 r. {x + y})"*
thus *"z* ∈ *(*⋃*x*∈*A. cball x r)"*

```
    by (force simp: dist_norm)
  next
    fix z assume "z ∈ (⋃x∈A. cball x r)"
    then obtain x where "x ∈ A" "z ∈ cball x r"
      by auto
    hence "z = x + (z - x)" "z - x ∈ cball 0 r"
      by (auto simp: dist_norm)
    thus "z ∈ (⋃x∈A. ⋃y∈cball 0 r. {x + y})"
      using ‹x ∈ A› by blast
  qed
  finally show ?thesis .
qed

lemma continuous_on_linepath [continuous_intros]:
  assumes "continuous_on A f" "continuous_on A g" "continuous_on A h"
  shows   "continuous_on A (λx. linepath (f x) (g x) (h x))"
  unfolding linepath_def by (intro continuous_intros assms)

lemma contour_integral_linepath_has_field_derivative:
  assumes A: "open A" "a ∈ A" "z ∈ A" "closed_segment a z ⊆ A"
  assumes holo: "f holomorphic_on A"
  shows   "((λz. contour_integral (linepath a z) f) has_field_derivative
f z) (at z within B)"
proof -
  define e where "e = (if A = UNIV then 1 else setdist (closed_segment
a z) (-A))"
  have "e > 0"
  proof (cases "A = UNIV")
    case False
    hence "setdist (closed_segment a z) (-A) > 0"
      using assms unfolding e_def
      by (subst setdist_gt_0_compact_closed compact_path_image) auto
    thus "e > 0"
      using False by (auto simp: e_def)
  qed (auto simp: e_def)

  have dist_ge_e: "dist x y ≥ e" if "x ∈ closed_segment a z" "y ∈ -A"
for x y
    using setdist_le_dist[OF that] that by (auto simp: e_def)

  define A' where "A' = (⋃x∈closed_segment a z. ball x e)"
  have "A' ⊆ A"
  proof
    fix x assume "x ∈ A'"
    then obtain y where "y ∈ closed_segment a z" "dist y x < e"
      by (auto simp: A'_def)
    thus "x ∈ A"
      using dist_ge_e[of y x] by auto
  qed
```

```
  have "closed_segment a z ⊆ A'" "open A'"
    using <e > 0> unfolding A'_def by fastforce+
  have "convex A'"
    unfolding A'_def by (intro convex_minkowski_sum_ball) auto
  have "a ∈ A'" "z ∈ A'"
    using <closed_segment a z ⊆ A'> by auto
  note A' = <A' ⊆ A> <closed_segment a z ⊆ A'> <open A'> <convex A'>
<a ∈ A'> <z ∈ A'>
  note holo = holomorphic_on_subset[OF holo <A' ⊆ A>]

  have "(f has_field_derivative deriv f z) (at z)" if "z ∈ A'" for z
    using that A' A by (auto intro!: holomorphic_derivI holo)
  note [derivative_intros] = DERIV_chain2[OF this]
  note [continuous_intros] =
    continuous_on_compose2[OF holomorphic_on_imp_continuous_on [OF holo]]
    continuous_on_compose2[OF holomorphic_on_imp_continuous_on [OF holomorphic_deriv[OF
holo]]]
  have [derivative_intros]:
      "((λx. linepath a x t) has_field_derivative of_real t) (at x within
A')" for t x
    by (auto simp: linepath_def scaleR_conv_of_real intro!: derivative_eq_intros)

  have *: "linepath a b t ∈ A'" if "a ∈ A'" "b ∈ A'" "t ∈ {0..1}" for
a b t
    using that linepath_in_convex_hull[of a A' b t] A' by (simp add: hull_same)

  have "((λz. integral {0..1} (λx. f (linepath a z x)) * (z - a)) has_field_derivative

      integral {0..1} (λt. deriv f (linepath a z t) * of_real t) * (z
- a) +
      integral {0..1} (λx. f (linepath a z x))) (at z within A')"
      (is "(_ has_field_derivative ?I) _")
    by (rule derivative_eq_intros leibniz_rule_field_derivative_real *
A')+
       (insert assms A' *,
        auto intro!: derivative_eq_intros leibniz_rule_field_derivative_real
                     integrable_continuous_real continuous_intros
             simp:   split_beta scaleR_conv_of_real)
  also have "(λz. integral {0..1} (λx. f (linepath a z x)) * (z - a))
=
               (λz. contour_integral (linepath a z) f)"
    by (simp add: contour_integral_integral)
  also have "?I = integral {0..1} (λx. deriv f (linepath a z x) * of_real
x * (z - a) +
                     f (linepath a z x))" (is "_ = integral _ ?g")
    by (subst integral_mult_left [symmetric], subst integral_add [symmetric])
       (insert assms A', auto intro!: integrable_continuous_real continuous_intros
simp: *)
  also have "(?g has_integral of_real 1 * f (linepath a z 1) - of_real
```

```
0 * f (linepath a z 0)) {0..1}"
    using * A A'
    by (intro fundamental_theorem_of_calculus)
       (auto intro!: derivative_eq_intros has_vector_derivative_real_field

             simp: linepath_def scaleR_conv_of_real)
  hence "integral {0..1} ?g = f (linepath a z 1)" by (simp add: has_integral_iff)
  also have "linepath a z 1 = z" by (simp add: linepath_def)
  also from ‹z ∈ A'› and ‹open A'› have "at z within A' = at z" by (rule
at_within_open)
  finally show ?thesis by (rule DERIV_subset) simp_all
qed



lemma set_integrable_bigo:
  fixes f g :: "real ⇒ 'a :: {banach, real_normed_field, second_countable_topology}"
  assumes "f ∈ O(λx. g x)" and "set_integrable lborel {a..} g"
  assumes "⋀b. b ≥ a ⟹ set_integrable lborel {a..<b} f"
  assumes [measurable]: "set_borel_measurable borel {a..} f"
  shows   "set_integrable lborel {a..} f"
proof -
  from assms(1) obtain C x0 where C: "C > 0" "⋀x. x ≥ x0 ⟹ norm (f
x) ≤ C * norm (g x)"
    by (fastforce elim!: landau_o.bigE simp: eventually_at_top_linorder)
  define x0' where "x0' = max a x0"

  have "set_integrable lborel {a..<x0'} f"
    by (intro assms) (auto simp: x0'_def)
  moreover have "set_integrable lborel {x0'..} f" unfolding set_integrable_def
  proof (rule Bochner_Integration.integrable_bound)
    from assms(2) have "set_integrable lborel {x0'..} g"
      by (rule set_integrable_subset) (auto simp: x0'_def)
    thus "integrable lborel (λx. C *R (indicator {x0'..} x *R g x))"
unfolding set_integrable_def
      by (intro integrable_scaleR_right) (simp add: abs_mult norm_mult)
  next
    from assms(4) have "set_borel_measurable borel {x0'..} f"
      by (rule set_borel_measurable_subset) (auto simp: x0'_def)
    thus "(λx. indicator {x0'..} x *R f x) ∈ borel_measurable lborel"
      by (simp add: set_borel_measurable_def)
  next
    show "AE x in lborel. norm (indicator {x0'..} x *R f x)
                            ≤ norm (C *R (indicator {x0'..} x *R g x))"
      using C by (intro AE_I2) (auto simp: abs_mult indicator_def x0'_def)
  qed
  ultimately have "set_integrable lborel ({a..<x0'} ∪ {x0'..}) f"
    by (rule set_integrable_Un) auto
```

```
  also have "{a..<x0'} ∪ {x0'..} = {a..}" by (auto simp: x0'_def)
  finally show ?thesis .
qed
```

5.1 Miscellaneous preliminary material

```
lemma uniform_limit_analytic_at:
  assumes "uniform_limit A f g F"
  assumes "eventually (λx. f x holomorphic_on A) F"
  assumes "z ∈ A" "open A" "F ≠ bot"
  shows   "g analytic_on {z}"
proof -
  obtain r where r: "r > 0" "cball z r ⊆ A"
    using ‹z ∈ A› ‹open A› by (meson open_contains_cball)
  show ?thesis
  proof (rule holomorphic_uniform_limit[of z r f F g])
    show "∀⇩F x in F. continuous_on (cball z r) (f x) ∧ f x holomorphic_on
ball z r"
      using assms(2)
    proof eventually_elim
      case (elim x)
      have "ball z r ⊆ cball z r"
        by auto
      also have "... ⊆ A"
        by fact
      finally have "f x holomorphic_on ball z r"
        using elim holomorphic_on_subset r by blast
      moreover have "continuous_on (cball z r) (f x)"
        using elim r by (auto intro: holomorphic_on_imp_continuous_on)
      ultimately show ?case by blast
    qed
  next
    assume "g holomorphic_on ball z r"
    thus "g analytic_on {z}"
      using ‹z ∈ A› ‹open A› analytic_at_ball r(1) by blast
  next
    show "uniform_limit (cball z r) f g F"
      using assms(1) by (rule uniform_limit_on_subset) (use r in auto)
  qed (use assms in auto)
qed

lemma uniform_limit_holomorphic:
  assumes "uniform_limit A f g F"
  assumes "eventually (λx. f x holomorphic_on A) F"
  assumes "open A" "F ≠ bot"
  shows   "g holomorphic_on A"
proof -
  have "g analytic_on A"
    using uniform_limit_analytic_at[OF assms(1,2) _ assms(3,4)] analytic_on_analytic_at
```

```
    by blast
  thus ?thesis
    using ‹open A› by (simp add: analytic_on_open)
qed


lemma Re_euler_mascheroni [simp]: "Re euler_mascheroni = euler_mascheroni"
  and Im_euler_mascheroni [simp]: "Im euler_mascheroni = 0"
  by (simp_all add: euler_mascheroni_def)


lemma has_real_derivative_imp_has_vector_derivative [derivative_intros]:
  assumes "(f has_real_derivative f') (at x)"
  shows   "(f has_vector_derivative f') (at x)"
  using assms by (simp add: has_real_derivative_iff_has_vector_derivative)


lemma DERIV_real_sqrt_generic':
  assumes "x ≠ 0"
  shows "(sqrt has_field_derivative (sgn x * inverse (sqrt x) / 2)) (at
x within A)"
  unfolding sqrt_def
  by (rule has_field_derivative_at_within) (use assms in ‹auto intro!:
DERIV_real_root_generic›)


lemma DERIV_real_root_generic':
  assumes "x ≠ 0"
  shows "(root n has_real_derivative (inverse (real n * (sgn x * root
n x) ^ (n - Suc 0)))) (at x within A)"
proof (cases "n = 0")
  case False
  show ?thesis
    by (rule has_field_derivative_at_within)
       (use False assms in ‹auto intro!: DERIV_real_root_generic simp:
field_simps sgn_if›)
next
  case True
  have "root 0 = (λx. 0)"
    by (auto simp: root_def fun_eq_iff)
  thus ?thesis
    using True by auto
qed

declare
  DERIV_real_sqrt_generic[THEN DERIV_chain2, derivative_intros del]
  DERIV_real_root_generic[THEN DERIV_chain2, derivative_intros del]
  DERIV_real_sqrt_generic'[THEN DERIV_chain2, derivative_intros]
  DERIV_real_root_generic'[THEN DERIV_chain2, derivative_intros]
```

lemmas *[derivative_intros] = has_vector_derivative_real_field*

lemma *DERIV_complex_norm [derivative_intros]:*
fixes *f :: "real ⇒ complex"*
assumes *"f x ≠ 0"* **and** *[derivative_intros]: "(f has_vector_derivative D') (at x)"*
shows *"((λx. norm (f x)) has_field_derivative ((D' · f x) / norm (f x))) (at x)"*
proof -
have *"norm (f x) > 0"*
using *assms* **by** *simp*
hence **: "(Re (f x))² + (Im (f x))² > 0"*
unfolding *cmod_def* **by** *simp*
hence ***: "(Re (f x))² + (Im (f x))² ≠ 0"*
by *auto*
from *assms* **have** ****: "Re (f x) ≠ 0 ∨ Im (f x) ≠ 0"*
by *(auto simp: complex_eq_iff)*
show *?thesis* **unfolding** *cmod_def*
using *** **by** *derivative (auto simp: divide_simps inner_complex_def)*
qed

5.2 Preliminary facts about the Gamma and Digamma function

lemma *cnj_Digamma:*
assumes *[simp]: "s ≠ 0"*
shows *"cnj (Digamma s) = Digamma (cnj s)"*
proof *(rule tendsto_unique)*
have *"(λx. (complex_of_real (ln (real x)) - (∑n<x. inverse (cnj s + of_nat n)))) ⟶ Digamma (cnj s)"*
(is *"?f ⟶ _")* **by** *(rule tendsto_cnj Digamma_LIMSEQ)+ auto*
also have *"?f = (λx. cnj (complex_of_real (ln (real x)) - (∑n<x. inverse (s + of_nat n))))"*
by *simp*
finally show *"... ⟶ Digamma (cnj s)"* **.**
show *"(λx. cnj (complex_of_real (ln (real x)) - (∑n<x. inverse (s + of_nat n)))) ⟶ cnj (Digamma s)"*
by *(rule tendsto_cnj Digamma_LIMSEQ)+ auto*
qed *auto*

lemma *cnj_Polygamma:*
assumes *[simp]: "s ≠ 0"*
shows *"cnj (Polygamma n s) = Polygamma n (cnj s)"*
proof *(cases "n = 0")*
case *True*
thus *?thesis* **using** *cnj_Digamma[of s]* **by** *simp*

```
next
  case False
  have "(λk. inverse ((cnj s + of_nat k) ^ Suc n)) sums
          ((-1) ^ Suc n * Polygamma n (cnj s) / fact n)"
    by (rule Polygamma_LIMSEQ) (use False in auto)
  also have "(λk. inverse ((cnj s + of_nat k) ^ Suc n)) =
             (λk. cnj (inverse ((s + of_nat k) ^ Suc n)))"
    by simp
  finally have "(λk. cnj (inverse ((s + of_nat k) ^ Suc n))) sums
                  ((-1) ^ Suc n * Polygamma n (cnj s) / fact n)" .
  moreover have "(λk. cnj (inverse ((s + of_nat k) ^ Suc n))) sums
                   (cnj ((-1) ^ Suc n * Polygamma n s / fact n))"
    unfolding sums_cnj by (intro Polygamma_LIMSEQ) (use False in auto)
  ultimately have "(-1) ^ Suc n * Polygamma n (cnj s) / fact n =
                   cnj ((-1) ^ Suc n * Polygamma n s / fact n)"
    by (rule sums_unique2)
  thus ?thesis by simp
qed

lemma norm_rGamma_Im_mono:
  assumes "Re z1 = Re z2" "|Im z1| ≤ |Im z2|"
  shows   "norm (rGamma z1) ≤ norm (rGamma z2)"
proof -
  define a b1 b2 where "a = Re z1" and "b1 = |Im z1|" and "b2 = |Im z2|"
  from assms have b12: "0 ≤ b1" "b1 ≤ b2"
    by (simp_all add: b1_def b2_def)
  have *: "norm (rGamma_series_Weierstrass (Complex a b1) n) ≤ norm (rGamma_series_Weierstr
(Complex a b2) n)" for n
    unfolding rGamma_series_Weierstrass_def norm_mult prod_norm [symmetric]
  proof (intro mult_mono mult_nonneg_nonneg norm_ge_zero exp_ge_zero prod_nonneg
ballI prod_mono conjI)
    from b12 show "norm (Complex a b1) ≤ norm (Complex a b2)"
      by (auto simp: cmod_def simp flip: abs_le_square_iff)
  next
    fix i assume i: "i ∈ {1..n}"
    have *: "norm (1 + z / of_nat i) = norm (of_nat i + z) / real i" for
z :: complex
    proof -
      have "norm (of_nat i + z) / real i = norm ((of_nat i + z) / of_nat
i)"
        by (subst norm_divide) auto
      also have "(of_nat i + z) / of_nat i = 1 + z / of_nat i"
        using i by (simp add: field_simps)
      finally show ?thesis ..
    qed
    show "cmod (1 + Complex a b1 / of_nat i) ≤ cmod (1 + Complex a b2
/ of_nat i)"
      using i b12 unfolding *
      by (intro divide_right_mono) (auto simp: cmod_def simp flip: abs_le_square_iff)
```

```
  qed auto
  have "norm (rGamma (Complex a b1)) ≤ norm (rGamma (Complex a b2))"
    by (rule tendsto_le sequentially_bot tendsto_norm tendsto_norm rGamma_Weierstrass_compl
       (rule always_eventually allI *)+
  also have "Complex a b1 ∈ {z1, cnj z1}"
    by (auto simp: a_def b1_def complex_eq_iff)
  hence "norm (rGamma (Complex a b1)) = norm (rGamma z1)"
    by (auto simp flip: cnj_rGamma)
  also have "Complex a b2 ∈ {z2, cnj z2}"
    by (auto simp: assms a_def b2_def complex_eq_iff)
  hence "norm (rGamma (Complex a b2)) = norm (rGamma z2)"
    by (auto simp flip: cnj_rGamma)
  finally show ?thesis .
qed

lemma Re_Digamma_Im_mono:
  assumes "Re z1 = Re z2" "Re z1 > 0" "|Im z1| ≤ |Im z2|"
  shows "Re (Digamma z1) ≤ Re (Digamma z2)"
proof -
  have [simp]: "z1 ≠ 0" "z2 ≠ 0"
    using assms by (auto simp: complex_eq_iff)
  define a b1 b2 where "a = Re z1" and "b1 = |Im z1|" and "b2 = |Im z2|"
  have "Re (Digamma (Complex a b1)) ≤ Re (Digamma (Complex a b2))"
  proof (rule tendsto_le[OF _ _ _ always_eventually[OF allI]])
    fix k :: nat
    have "(a + real n) / (b2^2 + (a + real n)^2) ≤ (a + real n) / (b1^2
+ (a + real n)^2)" for n
      using assms
      by (intro divide_left_mono mult_pos_pos add_nonneg_pos)
         (auto simp: a_def b1_def b2_def abs_le_square_iff)
    hence "(∑x<k. Re (1 / (of_nat x + Complex a b2))) ≤ (∑x<k. Re (1
/ (of_nat x + Complex a b1)))"
      using assms by (intro sum_mono) (auto simp: Re_divide field_simps)
    thus "Re (complex_of_real (ln (real k)) - (∑n<k. inverse (Complex
a b1 + of_nat n))) ≤
          Re (complex_of_real (ln (real k)) - (∑n<k. inverse (Complex
a b2 + of_nat n)))"
      by (simp add: field_simps)
  next
    show "sequentially ≠ bot" by simp
  qed ((rule Digamma_LIMSEQ tendsto_Re)+; use assms in ‹simp add: a_def
complex_eq_iff›)+
  also have "Complex a b1 ∈ {z1, cnj z1}"
    by (auto simp: complex_eq_iff a_def b1_def b2_def)
  hence "Re (Digamma (Complex a b1)) = Re (Digamma z1)"
    by (auto simp flip: cnj_Digamma)
  also have "Complex a b2 ∈ {z2, cnj z2}"
    by (auto simp: complex_eq_iff a_def b1_def b2_def assms)
  hence "Re (Digamma (Complex a b2)) = Re (Digamma z2)"
```

```
    by (auto simp flip: cnj_Digamma)
  finally show ?thesis .
qed

lemma Digamma_Re_le_Re_Digamma:
  assumes "Re z > 0"
  shows "Digamma (Re z) ≤ Re (Digamma z)"
proof -
  have "Digamma (Re z) = Re (Digamma (of_real (Re z)))"
    using assms by (subst Polygamma_of_real) auto
  also have "... ≤ Re (Digamma z)"
    by (rule Re_Digamma_Im_mono) (use assms in auto)
  finally show ?thesis .
qed

lemma Re_Digamma_nonneg:
  assumes "Re z > 0" "Digamma (Re z) ≥ 0"
  shows "Re (Digamma z) ≥ 0"
proof -
  have "0 ≤ Digamma (Re z)"
    using assms by simp
  also have "... ≤ Re (Digamma z)"
    by (rule Digamma_Re_le_Re_Digamma) (use assms in auto)
  finally show ?thesis .
qed

lemma norm_Gamma_Re_mono:
  assumes "Im z1 = Im z2" "Re z1 ≤ Re z2" "Re z1 > 0" "Digamma (Re z1)
≥ 0"
  shows   "norm (Gamma z1) ≤ norm (Gamma z2)"
proof -
  define a1 a2 b where "a1 = Re z1" and "a2 = Re z2" and "b = Im z1"

  have z1: "z1 = complex_of_real a1 + i * complex_of_real b"
   and z2: "z2 = complex_of_real a2 + i * complex_of_real b"
    by (simp_all add: a1_def a2_def b_def complex_eq_iff assms)

  show "norm (Gamma z1) ≤ norm (Gamma z2)"
    unfolding z1 z2
  proof (rule deriv_nonneg_imp_mono[where g = "λa. norm (Gamma (complex_of_real
a + i * b))"])
    fix x :: real
    assume x: "x ∈ {a1..a2}"
    with assms have "x > 0" by (auto simp: a1_def)
    define z where "z = complex_of_real x + i * complex_of_real b"
    have *: "complex_of_real x + i * complex_of_real b ∉ ℤ≤0"
      using ‹x > 0› by (auto elim!: nonpos_Ints_cases simp: complex_eq_iff)
    hence **: "Gamma (complex_of_real x + i * complex_of_real b) ≠ 0"
      by (subst Gamma_eq_zero_iff) auto
```

```
    have "((λx. norm (Gamma (of_real x + i * b))) has_field_derivative
            (Gamma z * Digamma z) · Gamma z / norm (Gamma z)) (at x)"
      using * ** by derivative (auto simp: z_def)
    also have "(Gamma z * Digamma z) · Gamma z = Re (Digamma z) * norm
(Gamma z) ^ 2"
      by (simp add: inner_complex_def cmod_def power2_eq_square field_simps)
    also have "... / norm (Gamma z) = Re (Digamma z) * norm (Gamma z)"
      by (simp add: power2_eq_square)
    finally show "((λx. norm (Gamma (of_real x + i * b)))
                    has_field_derivative Re (Digamma z) * norm (Gamma
z)) (at x)" .

    from x have "Re z > 0"
      using assms by (auto simp: z_def a1_def)
    moreover {
      have "0 ≤ Digamma a1" using assms by (simp add: a1_def)
      also have "... ≤ Digamma (Re x)"
        using assms x by (intro Digamma_real_mono) (auto simp: a1_def)
      finally have "Digamma (Re x) ≥ 0" .
    }
    ultimately have "Re (Digamma z) ≥ 0"
      by (intro Re_Digamma_nonneg) (auto simp: z_def)
    thus "Re (Digamma z) * norm (Gamma z) ≥ 0"
      using assms x by (intro mult_nonneg_nonneg) auto
  qed (use assms in ‹auto simp: a1_def a2_def›)
qed

lemma norm_Gamma_Im_mono:
  assumes "Re z1 = Re z2" "|Im z1| ≤ |Im z2|" "z1 ∈ ℤ≤0 ⟶ z2 ∈ ℤ≤0"
  shows   "norm (Gamma z1) ≥ norm (Gamma z2)"
proof (cases "z2 ∈ ℤ≤0")
  case True
  hence "Gamma z2 = 0" by (auto simp: Gamma_eq_zero_iff)
  thus ?thesis by simp
next
  case False
  with assms have "z1 ∉ ℤ≤0"
    by auto
  with assms False norm_rGamma_Im_mono[of z1 z2] show ?thesis
    by (auto simp: Gamma_def norm_divide field_simps rGamma_eq_zero_iff)
qed

lemma norm_Gamma_bound:
  assumes "Re z ∉ ℤ≤0"
  shows   "norm (Gamma z) ≤ |Gamma (Re z)|"
proof -
  from assms have "z ∉ ℤ≤0"
    by (auto simp: nonpos_Ints_def)
  with norm_Gamma_Im_mono[of "of_real (Re z)" z] assms show ?thesis
```

```
    by (auto simp: Gamma_complex_of_real of_real_in_nonpos_Ints_iff)
qed

lemma norm_Gamma_bound':
  assumes "Re z > 0"
  shows   "norm (Gamma z) ≤ Gamma (Re z)"
proof -
  from assms have "Re z ∉ ℤ≤0"
    by (auto elim: nonpos_Ints_cases)
  with assms show ?thesis
    using norm_Gamma_bound[of z] by simp
qed

end
```

6 The Incomplete Gamma Function

```
theory Incomplete_Gamma
imports
  "HOL-Complex_Analysis.Complex_Analysis" "HOL-Library.Going_To_Filter"
  Generalized_Hypergeometric_Series.Generalized_Hypergeometric_Series
  Safe_Power More_Beta More_Dominated_Convergence
  Derivative_Method Incomplete_Gamma_Lemma_Bucket
begin
```

6.1 Construction of the auxiliary entire function

For an overview of the functions we will define, see §8.2 of the NIST Digital Library of Mathematical Functions [1].

We first use the regularised hypergeometric series to define a version of the regularised lower gamma function that is entire in both variables and therefore particularly pleasant to handle. The NIST DLMF calls this function $\gamma^*(s, z)$.

```
definition pre_Gamma_rincl :: "'a :: Gamma ⇒ 'a ⇒ 'a" where
  "pre_Gamma_rincl s z = exp (-z) * reg_hypergeo_F [1] [1+s] z"

lemma pre_Gamma_rincl_complex_of_real:
  "pre_Gamma_rincl (complex_of_real s) (of_real z) = of_real (pre_Gamma_rincl
s z)"
  by (simp add: pre_Gamma_rincl_def complex_of_real_reg_hypergeo_F flip:
exp_of_real)

lemma pre_Gamma_rincl_0_left [simp]: "pre_Gamma_rincl 0 z = 1"
  by (simp add: pre_Gamma_rincl_def reg_hypergeo_F_conv_hypergeo_F hypergeo_F_cancel
exp_minus)

lemma pre_Gamma_rincl_0_right [simp]: "pre_Gamma_rincl s 0 = rGamma (s
```

```
+ 1)"
  by (simp add: pre_Gamma_rincl_def reg_hypergeo_F_0 add_ac)

lemma pre_Gamma_rincl_1_left:
  assumes [simp]: "z ≠ 0"
  shows   "pre_Gamma_rincl 1 z = (1 - exp (-z)) / z"
proof -
  have "pre_Gamma_rincl 1 z = exp (-z) * hypergeo_F [1] [2] z"
    by (simp add: pre_Gamma_rincl_def reg_hypergeo_F_conv_hypergeo_F
                  rGamma_inverse_Gamma Gamma_numeral)
  also have "hypergeo_F [1] [2] z = (exp z - 1) / z"
    by (subst hypergeo_F_1_2) auto
  also have "exp (-z) * ... = (1 - exp (-z)) / z"
    by (simp add: field_simps exp_minus)
  finally show ?thesis .
qed

lemma analytic_pre_Gamma_rincl [analytic_intros]:
  assumes [analytic_intros]: "s analytic_on X" "z analytic_on X"
  shows   "(λx. pre_Gamma_rincl (s x) (z x)) analytic_on X"
  unfolding pre_Gamma_rincl_def
  by (intro analytic_intros analytic_reg_hypergeo_F[where as = "[λ_.
1]" and bs = "[λx. 1 + s x]"])
     (auto intro!: analytic_intros)

lemma continuous_on_pre_Gamma_rincl [continuous_intros]:
  fixes s z :: "_ ⇒ 'a :: {Gamma, heine_borel}"
  assumes [continuous_intros]: "continuous_on X s" "continuous_on X z"
  shows   "continuous_on X (λx. pre_Gamma_rincl (s x) (z x))"
proof -
  have "continuous_on ((λx. (s x, z x)) ` X) (λ(s,z). pre_Gamma_rincl
s z :: 'a)"
    unfolding pre_Gamma_rincl_def case_prod_unfold
    by (intro continuous_intros
              continuous_on_reg_hypergeo_F[where as = "[λ_. 1]" and bs
= "[λx. 1 + fst x :: 'a]"])
       (auto intro!: continuous_intros)
  hence "continuous_on X ((λ(s,z). pre_Gamma_rincl s z) ∘ (λx. (s x,
z x)))"
    by (intro continuous_on_compose continuous_intros)
  thus ?thesis
    by (simp add: o_def)
qed

lemma tendsto_pre_Gamma_rincl [tendsto_intros]:
  fixes s z :: "'a :: {Gamma, heine_borel}"
  assumes "(f ⟶ s) F" "(g ⟶ z) F"
  shows   "((λx. pre_Gamma_rincl (f x) (g x)) ⟶ pre_Gamma_rincl s
z) F"
```

proof -
  have *"continuous_on UNIV (λ(s,z). pre_Gamma_rincl s z :: 'a)"*
    by *(auto simp: case_prod_unfold intro!: continuous_intros)*
  hence *"((λx. case (f x, g x) of (s, z) ⇒ pre_Gamma_rincl s z) ⟶*

(case (s, z) of (s, z) ⇒ pre_Gamma_rincl s z)) F"
    by *(rule continuous_on_tendsto_compose) (use assms* **in** *‹auto intro!: tendsto_intros›)*
  thus *?thesis*
    by *simp*
qed

lemma *continuous_pre_Gamma_rincl [continuous_intros]:*
  fixes *s z :: "_ ⇒ 'a :: {Gamma, heine_borel}"*
  assumes *[continuous_intros]: "continuous (at x within A) s" "continuous (at x within A) z"*
  shows *"continuous (at x within A) (λx. pre_Gamma_rincl (s x) (z x))"*
  using *assms* **unfolding** *continuous_def*
  by *(cases "at x within A = bot") (auto simp: Lim_ident_at intro: tendsto_intros)*

lemma *sums_pre_Gamma_rincl:*
  *"(λn. exp (-z) * rGamma (s + of_nat n + 1) * z ^ n) sums (pre_Gamma_rincl s z)"*
  unfolding *pre_Gamma_rincl_def*
  using *sums_mult[OF sums_reg_hypergeo_F[of "[1]" "[1+s]" z], of "exp (-z)"]*
  by *(simp flip: pochhammer_fact add: add_ac mult_ac)*

lemma *erf_conv_pre_Gamma_rincl_complex:*
  *"erf (z :: complex) = z * pre_Gamma_rincl (1 / 2) (z^2)"*
proof -
  have *[simp]: "1 / (2::complex) ∉ $\mathbb{Z}_{\leq 0}$"*
    by *force*
  have *"z * pre_Gamma_rincl (1/2) (z^2) = exp (-z^2) * z * reg_hypergeo_F [1] [3/2] (z^2)"*
    by *(simp add: pre_Gamma_rincl_def)*
  also have *"reg_hypergeo_F [1] [3/2] (z^2) = rGamma (3/2) * hypergeo_F [1] [3/2] (z^2)"*
    by *(subst reg_hypergeo_F_conv_hypergeo_F) auto*
  also have *"... = 1 / Gamma (1/2 + 1) * exp (z^2) * hypergeo_F [1/2] [3/2] (-z^2)"*
    by *(subst hypergeo_F_kummer_transform) (auto simp: rGamma_inverse_Gamma field_simps)*
  also have *"... = exp (z^2) * (of_real (2 / sqrt pi) * hypergeo_F [1/2] [3/2] (-z^2))"*
    by *(subst Gamma_plus1) (auto simp: Gamma_one_half_complex)*
  also have *"exp (-z^2) * z * (exp (z^2) * (of_real (2 / sqrt pi) * hypergeo_F [1 / 2] [3 / 2] (- z^2))) =*

```
              of_real (2 / sqrt pi) * z * hypergeo_F [1 / 2] [3 / 2]
(- z^2)"
    by (simp add: exp_minus)
  also have "... = erf z"
    by (rule erf_conv_hypergeo_F [symmetric])
  finally show ?thesis ..
qed

lemma erf_conv_pre_Gamma_rincl_real:
  "erf (z :: real) = z * pre_Gamma_rincl (1 / 2) (z^2)"
  using erf_conv_pre_Gamma_rincl_complex[of "of_real z"]
        pre_Gamma_rincl_complex_of_real[of "1/2" "z^2"] by (simp add:
complex_eq_iff)
```

6.2 The regularised lower Gamma function

We now add the factor z^s, which contributes a branch cut, to obtain the regularised lower gamma function $P(s,z)$ (again using the NIST DLMF notation).

This is essentially $\gamma(s,z)/\Gamma(s)$, but with the benefit that it is defined even when $\gamma(s,z)$ and $\Gamma(s)$ are not (i.e. when s is a non-negative integer).

```
definition Gamma_rincl :: "'a :: {Gamma, ln} ⇒ 'a ⇒ 'a" where
  "Gamma_rincl s z = z powr' s * pre_Gamma_rincl s z"

lemma Gamma_rincl_0_left [simp]: "Gamma_rincl 0 z = 1"
  by (simp add: Gamma_rincl_def)

lemma Gamma_rincl_0_right [simp]: "s ≠ 0 ⟹ Gamma_rincl s 0 = 0"
  by (auto simp: Gamma_rincl_def)

lemma Gamma_rincl_1_left: "Gamma_rincl 1 z = 1 - exp (-z)"
  by (cases "z = 0") (auto simp: Gamma_rincl_def pre_Gamma_rincl_1_left)

lemma Gamma_rincl_complex_of_real:
  assumes "s ∉ ℤ ⟹ 0 ≤ z"
  shows   "Gamma_rincl (complex_of_real s) (of_real z) = of_real (Gamma_rincl
s z)"
  unfolding Gamma_rincl_def of_real_mult
  by (subst powr'_complex_of_real) (use assms in ‹auto simp: pre_Gamma_rincl_complex_of_rea
```

$P(s,z)$ is holomorphic in s and z apart from a branch cut along the negative real axis for z. Alternatively, if s is a fixed integer, $P(s,z)$ is entire in z if $s \geq 0$ and meromorphic with a pole at $z = 0$ if $s < 0$.

```
lemma analytic_Gamma_rincl [analytic_intros]:
  assumes [analytic_intros]: "s analytic_on X" "z analytic_on X"
  assumes "⋀x. x ∈ X ⟹ z x ∉ ℝ_{≤0}"
  shows   "(λx. Gamma_rincl (s x) (z x)) analytic_on X"
  unfolding Gamma_rincl_def using assms(3) by (auto intro!: analytic_intros)
```

```
lemma analytic_Gamma_rincl':
  assumes [analytic_intros]: "f analytic_on X"
  assumes "⋀x. x ∈ X ⟹ s ∈ (ℤ_{≤0} - {0}) ⟹ f x ≠ 0"
  assumes "⋀x. x ∈ X ⟹ s ∉ ℤ ⟹ f x ∉ ℝ_{≤0}"
  shows   "(λx. Gamma_rincl s (f x)) analytic_on X"
proof (cases "s ∈ ℤ")
  case False
  thus ?thesis
    using assms(3) by (auto intro!: analytic_intros)
next
  case True
  then obtain n where s: "s = of_int n"
    by (elim Ints_cases)
  show ?thesis
    using assms(2,3) unfolding s Gamma_rincl_def powr'_of_int of_int_in_nonpos_Ints_iff
    by (auto intro!: analytic_intros simp: of_int_in_nonpos_Ints_iff)
qed
```

$P(s, z)$ is continuous away from the nonpositive reals. It is additionally also continuous at $z = 0$ if $\mathrm{Re}(s) > 0$.

```
lemma continuous_on_Gamma_rincl_complex [continuous_intros]:
  fixes x z :: "'a :: topological_space ⇒ complex"
  assumes [continuous_intros]: "continuous_on X s" "continuous_on X z"
  assumes "⋀x. x ∈ X ⟹ Re (z x) ≥ 0 ∨ Im (z x) ≠ 0"
  assumes "⋀x. x ∈ X ⟹ z x = 0 ⟹ Re (s x) > 0"
  shows   "continuous_on X (λx. Gamma_rincl (s x) (z x))"
proof -
  have "continuous_on X (λx. z x powr s x * pre_Gamma_rincl (s x) (z x))"
    unfolding Gamma_rincl_def using assms(3-)
    by (intro continuous_intros) auto
  also have "?this ⟷ ?thesis"
  proof (intro continuous_on_cong)
    fix x assume "x ∈ X"
    show "z x powr s x * pre_Gamma_rincl (s x) (z x) = Gamma_rincl (s
x) (z x)"
    proof (cases "z x = 0")
      case True
      hence "s x ≠ 0"
        using assms(3,4)[OF ‹x ∈ X›] by auto
      thus ?thesis using True
        by (auto simp: Gamma_rincl_def)
    next
      case False
      thus ?thesis using assms(3,4)[OF ‹x ∈ X›]
        by (auto simp: Gamma_rincl_def powr'_complex)
    qed
  qed auto
  finally show ?thesis .
```

qed

lemma *continuous_on_Gamma_rincl_real [continuous_intros]:*
fixes *x z :: "'a :: topological_space ⇒ real"*
assumes *[continuous_intros]: "continuous_on X s" "continuous_on X z"*
assumes *"⋀x. x ∈ X ⟹ z x ≥ 0"*
assumes *"⋀x. x ∈ X ⟹ z x = 0 ⟹ s x > 0"*
shows *"continuous_on X (λx. Gamma_rincl (s x) (z x))"*
proof -
note *[continuous_intros del] = continuous_on_powr*
note *[continuous_intros] = continuous_on_powr'*
have *"continuous_on X (λx. z x powr s x * pre_Gamma_rincl (s x) (z x))"*
unfolding *Gamma_rincl_def* **using** *assms(3-)*
by *(intro continuous_intros) auto*
also have *"?this ⟷ ?thesis"*
proof *(intro continuous_on_cong)*
fix *x* **assume** *"x ∈ X"*
show *"z x powr s x * pre_Gamma_rincl (s x) (z x) = Gamma_rincl (s x) (z x)"*
proof *(cases "z x = 0")*
case *True*
hence *"s x ≠ 0"*
using *assms(3,4)[OF ‹x ∈ X›]* **by** *auto*
thus *?thesis* **using** *True*
by *(auto simp: Gamma_rincl_def)*
next
case *False*
thus *?thesis* **using** *assms(3,4)[OF ‹x ∈ X›]*
by *(auto simp: Gamma_rincl_def powr'_real)*
qed
qed *auto*
finally show *?thesis* .
qed

lemma *tendsto_Gamma_rincl_complex [tendsto_intros]:*
fixes *s z :: complex*
assumes *"(f ⟶ s) F" "(g ⟶ z) F" "z ∉ ℝ≤0 ∨ (z = 0 ∧ Re s > 0)"*
shows *"((λx. Gamma_rincl (f x) (g x)) ⟶ Gamma_rincl s z) F"*
unfolding *Gamma_rincl_def* **using** *assms* **by** *(auto intro!: tendsto_intros)*

lemma *tendsto_Gamma_rincl_real [tendsto_intros]:*
fixes *s z :: real*
assumes *"(f ⟶ s) F" "(g ⟶ z) F" "z > 0 ∨ (z = 0 ∧ s > 0)"*
shows *"((λx. Gamma_rincl (f x) (g x)) ⟶ Gamma_rincl s z) F"*
unfolding *Gamma_rincl_def* **using** *assms* **by** *(auto intro!: tendsto_intros)*

lemma *continuous_Gamma_rincl_complex [continuous_intros]:*
fixes *s z :: "_ ⇒ complex"*

```
  assumes "continuous (at x within A) s" "continuous (at x within A)
z"
  assumes "z x ∉ ℝ≤0 ∨ (z x = 0 ∧ Re (s x) > 0)"
  shows   "continuous (at x within A) (λx. Gamma_rincl (s x) (z x))"
  using assms unfolding continuous_def
  by (cases "at x within A = bot") (auto simp: Lim_ident_at intro: tendsto_intros)

lemma continuous_Gamma_rincl_real [continuous_intros]:
  fixes s z :: "_ ⇒ real"
  assumes "continuous (at x within A) s" "continuous (at x within A)
z"
  assumes "z x > 0 ∨ (z x = 0 ∧ s x > 0)"
  shows   "continuous (at x within A) (λx. Gamma_rincl (s x) (z x))"
  using assms unfolding continuous_def
  by (cases "at x within A = bot") (auto simp: Lim_ident_at intro: tendsto_intros)
```

Writing $P(s, z)$ as a series:

```
lemma sums_Gamma_rincl:
  "(λn. z powr' s * z ^ n * exp (-z) * rGamma (s + of_nat n + 1)) sums
(Gamma_rincl s z)"
  using sums_mult[OF sums_pre_Gamma_rincl[of z s], of "z powr' s"]
  by (simp add: mult_ac Gamma_rincl_def)

lemma sums_Gamma_rincl_complex:
  assumes "z ≠ 0 ∨ s ∉ ℤ≤0"
  shows   "(λn. z powr (s + of_nat n) * exp (-z) * rGamma (s + of_nat
n + 1 :: complex))
             sums (Gamma_rincl s z)"
proof -
  have "(λn. z powr' s * z ^ n * exp (-z) * rGamma (s + of_nat n + 1))
sums (Gamma_rincl s z)"
    by (rule sums_Gamma_rincl)
  also have "(λn. z powr' s * z ^ n * exp (-z) * rGamma (s + of_nat n
+ 1)) =
             (λn. z powr (s + of_nat n) * exp (-z) * rGamma (s + of_nat
n + 1))"
    using assms
    by (cases "z = 0"; cases "s = 0")
       (auto simp: fun_eq_iff powr'_complex power_0_left powr_add)
  finally show ?thesis .
qed

lemma sums_Gamma_rincl_real:
  assumes "z > 0 ∨ s ∈ ℤ - ℤ≤0"
  shows   "(λn. z powr' (s + of_nat n) * exp (-z) * rGamma (s + of_nat
n + 1 :: real))
             sums (Gamma_rincl s z)"
proof -
  have "(λn. z powr' s * z ^ n * exp (-z) * rGamma (s + of_nat n + 1))
```

```
sums (Gamma_rincl s z)"
    by (rule sums_Gamma_rincl)
  also have "(λn. z powr' s * z ^ n * exp (-z) * rGamma (s + of_nat n
+ 1)) =
             (λn. z powr' (s + of_nat n) * exp (-z) * rGamma (s + of_nat
n + 1))"
  proof
    fix n :: nat
    show "z powr' s * z ^ n * exp (-z) * rGamma (s + of_nat n + 1) =
          z powr' (s + of_nat n) * exp (-z) * rGamma (s + of_nat n + 1)"
    proof (cases "s ∈ ℤ")
      case True
      then obtain m where s: "s = of_int m"
        by (elim Ints_cases)
      have *: "s + of_nat n = of_int (m + int n)"
        using s by simp
      have "z powr' (s + of_nat n) = z powi (m + int n)"
        by (subst *, subst powr'_of_int) auto
      also have "... = z powi m * z ^ n"
        by (subst power_int_add) (use assms s in ‹auto simp: of_int_in_nonpos_Ints_iff›)
      finally show ?thesis
        by (simp add: s)
    next
      case False
      with assms have "z > 0"
        by blast
      thus ?thesis
        using assms False
        by (auto simp: powr'_def powr_add powr_realpow)
    qed
  qed
  finally show ?thesis .
qed
```

The recurrence relation for $P(s, z)$:

```
lemma Gamma_rincl_plus1_complex:
  assumes "s ≠ -1"
  shows   "Gamma_rincl (s+1) z = Gamma_rincl s z - z powr' s * exp (-z)
* rGamma (s+1 :: complex)"
proof (cases "z = 0")
  case [simp]: True
  show ?thesis
    by (cases "s = 0"; cases "s + 1 = 0"; use assms in ‹auto simp: add_eq_0_iff2›)
next
  case [simp]: False
  let ?f = "(λn. z powr (s + of_nat n) * exp (-z) * rGamma (s + of_nat
n + 1))"
  have "(λn. ?f (Suc n)) sums (Gamma_rincl (s+1) z)"
    using sums_Gamma_rincl_complex[of z "s + 1"] by (simp add: add_ac)
```

```
  hence "?f sums (Gamma_rincl (s+1) z + ?f 0)"
    by (rule sums_Suc)
  moreover have "?f sums (Gamma_rincl s z)"
    by (rule sums_Gamma_rincl_complex) auto
  ultimately have "Gamma_rincl (s+1) z + ?f 0 = Gamma_rincl s z"
    by (rule sums_unique2)
  thus ?thesis
    by (simp add: algebra_simps powr'_complex)
qed

lemma Gamma_rincl_plus1_real:
  assumes "z ≥ 0 ∨ s ∈ ℤ" "s ≠ -1"
  shows   "Gamma_rincl (s+1) z = Gamma_rincl s z - z powr' s * exp (-z)
* rGamma (s+1 :: real)"
proof (cases "z = 0")
  case [simp]: False
  have *: "complex_of_real s = -1 ⟷ s = -1"
    by (auto simp: complex_eq_iff)
  have "complex_of_real (Gamma_rincl (s+1) z) = Gamma_rincl (of_real s
+ 1) (of_real z)"
    by (subst Gamma_rincl_complex_of_real [symmetric]) (use assms(1) in
auto)
  also have "... = Gamma_rincl (of_real s) (of_real z) -
                   of_real z powr' of_real s * exp (-of_real z) *  rGamma
(of_real s + 1)"
    by (subst Gamma_rincl_plus1_complex) (use assms in ‹auto simp: *›)
  also have "... = complex_of_real (Gamma_rincl s z - z powr' s * exp
(-z) * rGamma (s+1))"
    unfolding of_real_diff
    by (subst Gamma_rincl_complex_of_real)
       (use assms in ‹auto simp flip: exp_of_real rGamma_complex_of_real
simp: powr'_complex_of_real›)
  finally show ?thesis
    by (simp only: of_real_eq_iff)
qed (cases "s = 0"; use assms in auto)
```

We can now also easily show a closed form for the case where s is a positive integer:

```
lemma Gamma_rincl_of_nat_left_complex:
  fixes z :: complex
  shows "Gamma_rincl (of_nat (Suc n)) z = 1 - exp (-z) * (∑k≤n. z ^
k / fact k)"
proof (induction n)
  case 0
  thus ?case by (simp add: Gamma_rincl_1_left)
next
  case (Suc n)
  have "of_nat (Suc n) ≠ (-1::complex)"
    by (auto simp: complex_eq_iff)
```

```
  hence "Gamma_rincl (of_nat (Suc (Suc n))) z =
           1 - exp (-z) * (∑k≤n. z ^ k / fact k) -
           z powr' of_nat (Suc n) * exp (-z) * rGamma (of_int (int n +
2))"
    using Gamma_rincl_plus1_complex[of "of_nat (Suc n)" z] Suc.IH by (simp
add: add_ac)
  also have "z powr' of_nat (Suc n) = z powr of_nat (Suc n)"
    by (subst powr'_complex) (auto simp: complex_eq_iff)
  also have "... = z ^ Suc n"
    by (subst powr_nat) auto
  also have "rGamma (of_int (int n + 2) :: complex) = 1 / fact (Suc n)"
    by (subst rGamma_of_int) (auto simp: nat_add_distrib divide_simps)
  also have "1 - exp (- z) * (∑k≤n. z ^ k / fact k) - z ^ Suc n * exp
(- z) * (1 / fact (Suc n)) =
             1 - exp (- z) * (∑k∈insert (Suc n) {..n}. z ^ k / fact
k)"
    by (subst sum.insert) (auto simp: field_simps simp del: fact_Suc)
  also have "insert (Suc n) {..n} = {..Suc n}"
    by auto
  finally show ?case .
qed

lemma Gamma_rincl_of_nat_left_real:
  fixes z :: real
  shows "Gamma_rincl (of_nat (Suc n)) z = 1 - exp (-z) * (∑k≤n. z ^
k / fact k)"
proof -
  have "Gamma_rincl (of_nat (Suc n)) z = Gamma_rincl (of_nat (Suc n))
(complex_of_real z)"
    using Gamma_rincl_complex_of_real[of "of_nat (Suc n)" z] by simp
  also have "... = of_real (1 - exp (-z) * (∑k≤n. z ^ k / fact k))"
    by (subst Gamma_rincl_of_nat_left_complex) (simp_all flip: exp_of_real)
  finally show ?thesis
    by (simp only: of_real_eq_iff)
qed
```

Lastly, when $s = \frac{1}{2}$, the reduced lower incomplete Gamma function is related to the error function (via their hypergeometric representations):

```
lemma erf_conv_Gamma_rincl_real: "erf z = sgn z * Gamma_rincl (1/2) (z
^ 2 :: real)"
proof -
  have "Gamma_rincl (1/2) (z ^ 2) = sgn z * (z * pre_Gamma_rincl (1 /
2) (z²))"
    by (simp add: Gamma_rincl_def powr'_real sgn_if)
  also have "... = sgn z * erf z"
    by (subst erf_conv_pre_Gamma_rincl_real [symmetric]) auto
  finally show ?thesis by (auto simp: sgn_if)
qed
```

```
lemma Gamma_rincl_one_half_left_real:
  assumes "z ≥ 0"
  shows   "Gamma_rincl (1/2) (z :: real) = erf (sqrt z)"
  using assms by (subst erf_conv_Gamma_rincl_real) (auto simp: sgn_if)

lemma Gamma_rincl_one_half_left_complex:
  assumes z: "z ∉ ℝ≤0"
  shows "Gamma_rincl (1/2) (z :: complex) = erf (csqrt z)"
proof -
  have "Gamma_rincl (1/2) z - erf (csqrt z) = 0"
  proof (rule analytic_continuation[of "λz. Gamma_rincl (1/2) z - erf
(csqrt z)"])
    show "complex_of_real 1 islimpt of_real ‘ {0<..}"
      by (intro islimpt_isCont_image continuous_intros open_imp_islimpt)
         (auto simp: eventually_at_topological)
  next
    show "(λz. Gamma_rincl (1 / 2) z - erf (csqrt z)) holomorphic_on
(-ℝ≤0)"
      by (auto intro!: analytic_imp_holomorphic analytic_intros)
  next
    have "connected (-complex_of_real ‘ {..0})"
      by (intro starlike_imp_connected starlike_slotted_complex_plane_left)
    also have "(-complex_of_real ‘ {..0}) = -ℝ≤0"
      by (auto simp: nonpos_Reals_def)
    finally show "connected (-ℝ≤0 :: complex set)" .
  next
    show "Gamma_rincl (1/2) z - erf (csqrt z) = 0" if "z ∈ complex_of_real
‘ {0<..}" for z
    proof -
      from that obtain x where [simp]: "z = of_real x" and x: "x > 0"
        by auto
      have "Gamma_rincl (1/2) z - erf (csqrt z) = of_real (Gamma_rincl
(1/2) x - erf (sqrt x))"
        using x by (simp flip: Gamma_rincl_complex_of_real)
      also have "... = 0"
        by (subst Gamma_rincl_one_half_left_real) (use x in auto)
      finally show ?thesis
        by simp
    qed
  qed (use assms in auto)
  thus ?thesis
    by simp
qed
```

6.3 The lower incomplete Gamma function

```
definition Gamma_incl :: "'a :: {Gamma, ln} ⇒ 'a ⇒ 'a" where
  "Gamma_incl s z = Gamma s * Gamma_rincl s z"
```

```
lemma Gamma_incl_complex_of_real:
  assumes "s ∉ ℤ ⟹ 0 ≤ z"
  shows   "Gamma_incl (complex_of_real s) (of_real z) = of_real (Gamma_incl
s z)"
  unfolding Gamma_incl_def of_real_mult
  by (subst Gamma_rincl_complex_of_real)
     (use assms in ‹auto simp: Gamma_complex_of_real Gamma_rincl_complex_of_real›)
```

The lower incomplete Gamma function $\gamma(s, z)$ is analytic away from $z \leq 0$ and $s \in \{0, -1, -2, \ldots\}$.

```
lemma analytic_Gamma_incl [analytic_intros]:
  assumes [analytic_intros]: "s analytic_on X" "z analytic_on X"
  assumes "⋀x. x ∈ X ⟹ z x ∉ ℝ≤0" "⋀x. x ∈ X ⟹ s x ∉ ℤ≤0"
  shows   "(λx. Gamma_incl (s x) (z x)) analytic_on X"
  unfolding Gamma_incl_def using assms(3,4) by (auto intro!: analytic_intros)
```

When s is a positive integer, $\gamma(s, z)$ is entire in z.

```
lemma analytic_Gamma_incl_nonneg_int [analytic_intros]:
  assumes [analytic_intros]: "z analytic_on X" and "n ≥ 0"
  shows   "(λx. Gamma_incl (of_int n) (z x)) analytic_on X"
proof -
  note [analytic_intros del] = analytic_Gamma_rincl
  show ?thesis
    unfolding Gamma_incl_def using ‹n ≥ 0›
    by (intro analytic_intros analytic_Gamma_rincl') (auto simp: of_int_in_nonpos_Ints_iff)
qed

lemma analytic_Gamma_incl':
  assumes [analytic_intros]: "f analytic_on X"
  assumes "⋀x. x ∈ X ⟹ s ∈ ℤ≤0 ⟹ f x ≠ 0"
  assumes "⋀x. x ∈ X ⟹ s ∉ ℤ ⟹ f x ∉ ℝ≤0"
  shows   "(λx. Gamma_incl s (f x)) analytic_on X"
proof -
  note [analytic_intros del] = analytic_Gamma_rincl
  note [analytic_intros] = analytic_Gamma_rincl'
  show ?thesis
    unfolding Gamma_incl_def using assms
    by (auto intro!: analytic_intros)
qed

lemma continuous_on_Gamma_incl_complex [continuous_intros]:
  fixes x z :: "'a :: topological_space ⇒ complex"
  assumes [continuous_intros]: "continuous_on X s" "continuous_on X z"
  assumes "⋀x. x ∈ X ⟹ Re (z x) ≥ 0 ∨ Im (z x) ≠ 0"
  assumes "⋀x. x ∈ X ⟹ z x = 0 ⟹ Re (s x) > 0"
  assumes "⋀x. x ∈ X ⟹ s x ∉ ℤ≤0"
  shows   "continuous_on X (λx. Gamma_incl (s x) (z x))"
  unfolding Gamma_incl_def
  using assms by (auto intro!: continuous_intros)
```

lemma *continuous_on_Gamma_incl_real [continuous_intros]:*
fixes *x z :: "'a :: topological_space ⇒ real"*
assumes *[continuous_intros]: "continuous_on X s" "continuous_on X z"*
assumes *"⋀x. x ∈ X ⟹ z x ≥ 0"*
assumes *"⋀x. x ∈ X ⟹ z x = 0 ⟹ s x > 0"*
assumes *"⋀x. x ∈ X ⟹ s x ∉ ℤ≤0"*
shows *"continuous_on X (λx. Gamma_incl (s x) (z x))"*
unfolding *Gamma_incl_def*
using *assms* **by** *(auto intro!: continuous_intros)*

lemma *tendsto_Gamma_incl_complex [tendsto_intros]:*
fixes *s z :: complex*
assumes *"(f ⟶ s) F" "(g ⟶ z) F"*
assumes *"Re z ≥ 0 ∨ Im z ≠ 0" "z = 0 ⟹ Re s > 0" "s ∉ ℤ≤0"*
shows *"((λx. Gamma_incl (f x) (g x)) ⟶ Gamma_incl s z) F"*
thm *Gamma_incl_def*
unfolding *Gamma_incl_def* **using** *assms*
by *(auto intro!: tendsto_intros simp: complex_nonpos_Reals_iff complex_eq_iff)*

lemma *tendsto_Gamma_incl_real [tendsto_intros]:*
fixes *s z :: real*
assumes *"(f ⟶ s) F" "(g ⟶ z) F" "z ≥ 0" "z = 0 ⟹ s > 0" "s*
∉ ℤ≤0"
shows *"((λx. Gamma_incl (f x) (g x)) ⟶ Gamma_incl s z) F"*
unfolding *Gamma_incl_def* **using** *assms* **by** *(auto intro!: tendsto_intros)*

lemma *continuous_Gamma_incl_complex [continuous_intros]:*
fixes *s z :: "_ ⇒ complex"*
assumes *"continuous (at x within A) s" "continuous (at x within A)*
z"
assumes *"Re (z x) ≥ 0 ∨ Im (z x) ≠ 0" "z x = 0 ⟹ Re (s x) > 0" "s*
x ∉ ℤ≤0"
shows *"continuous (at x within A) (λx. Gamma_incl (s x) (z x))"*
using *assms* **unfolding** *continuous_def*
by *(cases "at x within A = bot") (auto simp: Lim_ident_at intro: tendsto_intros)*

lemma *continuous_Gamma_incl_real [continuous_intros]:*
fixes *s z :: "_ ⇒ real"*
assumes *"continuous (at x within A) s" "continuous (at x within A)*
z"
assumes *"z x ≥ 0" "z x = 0 ⟹ s x > 0" "s x ∉ ℤ≤0"*
shows *"continuous (at x within A) (λx. Gamma_incl (s x) (z x))"*
using *assms* **unfolding** *continuous_def*
by *(cases "at x within A = bot") (auto simp: Lim_ident_at intro: tendsto_intros)*

$\gamma(s, z)$ as a series:

lemma *sums_Gamma_incl:*
*"(λn. z powr' s * z ^ n * exp (-z) * Gamma s * rGamma (s + of_nat n*

```
+ 1)) sums (Gamma_incl s z)"
  using sums_mult[OF sums_Gamma_rincl[of z s], of "Gamma s"]
  by (simp add: mult_ac Gamma_incl_def)

lemma sums_Gamma_incl_complex:
  "(λn. z powr (s + complex_of_nat n) * exp (-z) * Gamma s * rGamma (s
+ of_nat n + 1))
      sums (Gamma_incl s z)"
  using sums_Gamma_incl[of z s]
  by (cases "z = 0"; cases "s = 0") (auto simp: powr'_complex powr_add)

lemma sums_Gamma_incl_real_nonneg:
  assumes "z ≥ 0"
  shows   "(λn. z powr (s + real n) * exp (-z) * Gamma s * rGamma (s
+ of_nat n + 1))
             sums (Gamma_incl s z)"
  using sums_Gamma_incl[of z s] assms
  by (cases "z = 0"; cases "s = 0") (auto simp: powr'_real powr_add powr_realpow)
```

The recurrence for $\gamma(s, z)$:

```
lemma Gamma_incl_plus1_complex:
  assumes "s ∉ ℤ_{≤0}"
  shows   "Gamma_incl (s+1) z = s * Gamma_incl s z - z powr' s * exp (-z
:: complex)"
proof -
  from assms have "Gamma s ≠ 0"
    by (auto simp: Gamma_eq_zero_iff)
  hence "Gamma_incl (s+1) z = s * Gamma_incl s z - z powr' s * exp (-z)"
    unfolding Gamma_incl_def using rGamma_plus1[of s]
    by (subst Gamma_rincl_plus1_complex)
       (use assms in ‹auto simp: rGamma_inverse_Gamma field_simps›)
  thus ?thesis .
qed

lemma Gamma_incl_plus1_real:
  assumes "s ∈ ℤ ∨ z ≥ 0" "s ∉ ℤ_{≤0}"
  shows   "Gamma_incl (s+1) z = s * Gamma_incl s z - z powr' s * exp (-z
:: real)"
proof -
  from assms have "Gamma s ≠ 0"
    by (auto simp: Gamma_eq_zero_iff)
  hence "Gamma_incl (s+1) z = s * Gamma_incl s z - z powr' s * exp (-z)"
    unfolding Gamma_incl_def using rGamma_plus1[of s]
    by (subst Gamma_rincl_plus1_real)
       (use assms in ‹auto simp: rGamma_inverse_Gamma field_simps›)
  thus ?thesis .
qed
```

For $\mathrm{Re}(s) > 0$, $\Gamma(s, z)$ has a representation as a contour integral.

```
theorem has_contour_integral_Gamma_incl:
  fixes s z :: complex
  assumes s: "Re s > 0"
  shows   "((λu. u powr (s-1) * exp (-u)) has_contour_integral Gamma_incl
s z) (linepath 0 z)"
proof (cases "z = 0")
  case [simp]: True
  from assms have "s ≠ 0"
    by auto
  thus ?thesis
    by (simp add: Gamma_incl_def)
next
  case [simp]: False
  define f where "f = (λk t. z^k / fact k * of_real t powr (s-1) * (1-t)^k)"
  from s have [simp]: "s ∉ ℤ≤0"
    by (auto elim!: nonpos_Ints_cases)

  have sums: "(λk. z^k * Gamma s / Gamma (s + of_nat (Suc k))) sums
                    ((z powr (-s) * exp z) * Gamma_incl s z)"
    using sums_mult[OF sums_Gamma_incl[of z s], of "z powr (-s) * exp
z"]
    by (simp add: powr_minus field_simps powr'_complex exp_minus rGamma_inverse_Gamma)

  have 1: "set_integrable lborel {0..1} (f k)" for k
  proof -
    have "set_integrable lebesgue {0<..<1}
            (λt. z ^ k / fact k * (of_real t powr (s - 1) * of_real (1
- t) powr (of_nat (Suc k) - 1)))"
      by (intro set_integrable_mult_right has_integral_Beta_complex) (use
s in auto)
    also have "?this ⟷ f k absolutely_integrable_on {0<..<1}"
      by (intro set_integrable_cong) (auto simp: f_def powr_nat)
    also have "... ⟷ set_integrable lborel {0..1} (f k)"
      unfolding absolutely_integrable_on_Icc_iff_Ioo [symmetric] unfold-
ing set_integrable_def
      by (subst integrable_completion) (auto simp: f_def)
    finally show ?thesis .
  qed

  have 2: "AE t∈{0..1} in lborel. summable (λk. norm (f k t))"
  proof -
    have *: "summable (λk. norm (f k t))" if t: "t ∈ {0<..<1}" for t
    proof -
      have "summable (λk. t powr (Re s - 1) * (inverse (fact k) * (norm
z * (1 - t)) ^ k))"
        by (intro summable_mult summable_exp)
      also have "(λk. t powr (Re s - 1) * (inverse (fact k) * (norm z
* (1 - t)) ^ k)) =
                 (λk. norm (f k t))"
```

```
      using t by (simp add: f_def norm_mult norm_divide norm_power norm_powr_complex
                            divide_simps mult_ac flip: of_real_diff)
    finally show ?thesis .
  qed
  have "AE t in lborel. t ≠ 0 ∧ t ≠ (1::real)"
    by (simp add: AE_lborel_singleton)
  thus ?thesis
    by eventually_elim (use * in auto)
qed

have 3: "summable (λk. LBINT t:{0..1}. norm (f k t))"
proof (rule summable_comparison_test')
  show "summable (λk. Beta (Re s) 1 * (inverse (fact k) * norm z ^
k))"
    by (intro summable_mult summable_exp)
next
  fix k :: nat assume "k ≥ 0"
  have "(LBINT t:{0..1}. norm (f k t)) =
          (LBINT t:{0..1}. (norm z ^ k / fact k) *
            (of_real t powr (Re s - 1) * of_real (1 - t) powr (of_nat
k)))"
    (is "set_lebesgue_integral _ _ ?lhs = set_lebesgue_integral _ _ ?rhs")
  proof (rule set_lebesgue_integral_cong_AE)
    have "AE t in lborel. t ≠ (1::real)"
      by (simp add: AE_lborel_singleton)
    thus "AE t∈{0..1} in lborel. ?lhs t = ?rhs t"
    proof eventually_elim
      case (elim t)
      thus ?case
        by (auto simp: f_def norm_mult norm_divide norm_powr_complex
norm_power powr_realpow
                 simp flip: of_real_diff)
    qed
  qed (auto simp: f_def)
  also have "... = (norm z ^ k / fact k) *
          (LBINT t:{0..1}. of_real t powr (Re s - 1) * of_real (1-t)
powr (of_nat k))"
    by (rule set_integral_mult_right)
  also have "(LBINT t:{0..1}. of_real t powr (Re s - 1) * of_real (1-t)
powr (of_nat k)) =
               set_lebesgue_integral lebesgue {0..1}
                    (λt. of_real t powr (Re s - 1) * of_real (1-t) powr
(of_nat k))"
    unfolding set_lebesgue_integral_def by (subst integral_completion)
auto
  also have "... = integral {0..1} (λt. of_real t powr (Re s - 1) *
of_real (1-t) powr (of_nat k))"
    using integrable_Beta'[of "Re s" "of_nat (Suc k)"] s
```

```
      by (intro set_lebesgue_integral_eq_integral(2) nonnegative_absolutely_integrable_1)
simp_all
    also have "... = Beta (Re s) (real k + 1)"
      using has_integral_Beta_real[of "Re s" "of_nat (Suc k)"] s
      by (simp add: has_integral_iff add_ac)
    also have "norm z ^ k / fact k * Beta (Re s) (real k + 1) ≤
               norm z ^ k / fact k * Beta (Re s) 1"
      by (intro mult_left_mono Beta_real_mono) (use s in auto)
    also have "... = Beta (Re s) 1 * (inverse (fact k) * norm z ^ k)"
      by (simp add: field_simps)
    finally have "(LBINT t:{0..1}. norm (f k t)) ≤ Beta (Re s) 1 * (inverse
(fact k) * norm z ^ k)" .
    moreover have "(LBINT t:{0..1}. norm (f k t)) ≥ 0"
      unfolding set_lebesgue_integral_def by (rule Bochner_Integration.integral_nonneg)
auto
    ultimately show "norm (LBINT t:{0..1}. norm (f k t)) ≤
                       Beta (Re s) 1 * (inverse (fact k) * norm z ^ k)"
      by simp
  qed

  have sum_eq: "(∑k. f k t) = of_real t powr (s-1) * exp (z*(1-t))"
for t
  proof -
    have "(λk. of_real t powr (s - 1) * (((1 - t) * z) ^ k /R fact k))
sums
            (of_real t powr (s - 1) * exp ((1-t)*z))"
      by (intro sums_mult exp_converges)
    also have "(λk. of_real t powr (s - 1) * (((1 - t) * z) ^ k /R fact
k)) = (λk. f k t)"
      by (auto simp: f_def divide_simps scaleR_conv_of_real)
    finally show ?thesis
      by (simp add: sums_iff mult_ac)
  qed

  have integrable':
    "(λt. of_real t powr (s - 1) * exp (z * of_real (1 - t))) absolutely_integrable_on
{0..1}"
  proof -
    have "set_integrable lborel {0..1} (λk. ∑i. f i k)"
      using set_integrable_suminf[OF 1 2 3] by simp
    also have "?this ⟷ set_integrable lborel {0..1} (λt. of_real t
powr (s-1) * exp (z*(1-t)))"
      by (intro set_integrable_cong sum_eq refl)
    finally show ?thesis
      unfolding set_integrable_def by (subst integrable_completion) auto
  qed

  have integrand_eq: "of_real t powr (s - 1) * exp (- (z * of_real t))
=
```

```
                    z powr -s * (t *R z) powr (s - 1) * exp (- (t *R
z)) * z"
    if t: "t ∈ {0..1}" for t
    using powr_times_real_left[of t z "s - 1"] t
    by (auto simp: scaleR_conv_of_real powr_diff powr_minus field_simps)


  have "(λt. of_real t powr (s - 1) * exp (z * of_real (1 - t))) integrable_on
{0..1}"
    using integrable' by (rule set_lebesgue_integral_eq_integral(1))
  hence "(λt. z powr s * exp (-z) * (of_real t powr (s - 1) * exp (z *
of_real (1 - t)))) integrable_on {0..1}"
    by (rule integrable_on_mult_right)
  also have "?this ⟷ (λz. z powr (s-1) * exp (-z)) contour_integrable_on
linepath 0 z"
    unfolding contour_integrable_on
  proof (rule integrable_cong, goal_cases)
    case (1 t)
    thus ?case
      using integrand_eq[of t]
      by (auto simp: scaleR_conv_of_real field_simps powr_minus exp_diff
exp_minus)
  qed
  finally have contour_integrable:
    "(λz. z powr (s-1) * exp (-z)) contour_integrable_on linepath 0 z"
.

  have "(∑k. LBINT t:{0..1}. f k t) = (LBINT t:{0..1}. ∑k. f k t)"
    using 1 2 3 by (rule set_integral_suminf [symmetric])
  also have "(λk. LBINT t:{0..1}. f k t) = (λk. z^k * Gamma s / Gamma
(s + of_nat (Suc k)))"
  proof
    fix k :: nat
    have "(λt. of_real t powr (s-1) * of_real (1-t) powr of_nat k) absolutely_integrable_on
{0<..<1}"
      using has_integral_Beta_complex[of s "of_nat (Suc k)"] s by simp
    also have "?this ⟷ (λt. of_real t powr (s-1) * of_real (1-t) ^
k) absolutely_integrable_on {0<..<1}"
      by (intro set_integrable_cong) auto
    finally have integrable:
      "(λt. of_real t powr (s-1) * of_real (1-t) ^ k) absolutely_integrable_on
{0<..<1}" .
    have *: "1 + complex_of_nat k ∉ ℤ≤0"
      by (auto elim!: nonpos_Ints_cases simp: complex_eq_iff)

    have "(LBINT t:{0..1}. f k t) =
          z ^ k / fact k * (LBINT t:{0..1}. of_real t powr (s - 1) *
of_real (1 - t) ^ k)"
      by (simp add: f_def mult_ac flip: set_integral_mult_right set_integral_mult_left)
```

```
    also have "(LBINT t:{0..1}. of_real t powr (s - 1) * of_real (1 -
t) ^ k) =
                set_lebesgue_integral lebesgue {0..1} (λt. of_real t powr
(s - 1) * of_real (1 - t) ^ k)"
      unfolding set_lebesgue_integral_def by (rule integral_completion
[symmetric]) auto
    also have "... = integral {0..1} (λt. of_real t powr (s-1) * of_real
(1-t) ^ k)"
      using integrable
      by (intro set_lebesgue_integral_eq_integral(2) nonnegative_absolutely_integrable_1)
         (simp_all add: absolutely_integrable_on_Icc_iff_Ioo)
    also have "... = integral {0<..<1} (λt. of_real t powr (s-1) * of_real
(1-t) ^ k)"
      by (simp add: integral_open_interval_real)
    also have "... = integral {0<..<1} (λt. of_real t powr (s-1) * of_real
(1-t) powr of_nat k)"
      by (intro integral_cong) auto
    also have "... = Beta s (of_nat (Suc k))"
      using has_integral_Beta_complex[of s "of_nat (Suc k)"] s by (simp
add: has_integral_iff)
    also have "z ^ k / fact k * Beta s (of_nat (Suc k)) = z ^ k * Gamma
s / Gamma (s + of_nat (Suc k))"
      using * by (auto simp: Beta_def add_ac Gamma_eq_zero_iff simp flip:
Gamma_fact)
    finally show "(LBINT t:{0..1}. f k t) = z^k * Gamma s / Gamma (s +
of_nat (Suc k))" .
  qed
  also have "(∑k. z^k * Gamma s / Gamma (s + of_nat (Suc k))) =
               z powr (-s) * exp z * Gamma_incl s z"
    using sums by (simp add: sums_iff)
  also have "(LBINT t:{0..1}. ∑k. f k t) = (LBINT t:{0..1}. of_real t
powr (s-1) * exp (z*(1-t)))"
    by (intro set_lebesgue_integral_cong) (use sum_eq in auto)
  also have "... = set_lebesgue_integral lebesgue {0..1} (λt. of_real
t powr (s-1) * exp (z*(1-t)))"
    unfolding set_lebesgue_integral_def by (rule integral_completion [symmetric])
auto
  also have "... = integral {0..1} (λt. of_real t powr (s-1) * exp (z*(1-t)))"
    using integrable' by (rule set_lebesgue_integral_eq_integral(2))
  also have "... = exp z * integral {0..1} (λt. of_real t powr (s-1) *
exp (-z*t))"
    by (simp add: exp_diff ring_distribs exp_minus field_simps
             flip: integral_mult_right set_integral_mult_left)
  also have "integral {0..1} (λt. of_real t powr (s-1) * exp (-z*t)) =

             contour_integral (linepath 0 z) (λu. z powr (-s) * u powr
(s-1) * exp (-u))"
    unfolding contour_integral_integral
  proof (rule integral_cong, goal_cases)
```

```
    case (1 t)
    thus ?case using integrand_eq[of t] by simp
  qed
  also have "... = z powr (-s) * contour_integral (linepath 0 z) (λu.
u powr (s-1) * exp (-u))"
    by (simp flip: integral_mult_right integral_mult_left add: mult_ac
contour_integral_integral)
  finally have "Gamma_incl s z = contour_integral (linepath 0 z) (λu. u
powr (s-1) * exp (-u))"
    by simp

  thus "((λu. u powr (s-1) * exp (-u)) has_contour_integral Gamma_incl
s z) (linepath 0 z)"
    using has_contour_integral_integral[OF contour_integrable] by simp
qed

lemma has_integral_Gamma_incl_complex_of_real:
  assumes s: "Re s > (0::real)"
  assumes "x ≥ 0"
  shows "((λt. of_real t powr (s - 1) * of_real (exp (-t)))
          has_integral Gamma_incl s (of_real x)) {0..x}"
proof (cases "x = 0")
  case True
  have [simp]: "s ≠ 0"
    using s by auto
  have "((λt. of_real t powr (s - 1) * of_real (exp (-t))) has_integral
0) {0}"
    by (rule has_integral_refl)
  thus ?thesis using True
    by (simp add: Gamma_incl_def)
next
  case False
  with assms have x: "x > 0"
    by auto
  have "((λt. t powr (s - 1) * exp (-t)) has_contour_integral Gamma_incl
s (of_real x))
         (linepath 0 (of_real x))"
    by (rule has_contour_integral_Gamma_incl) (use ‹x > 0› s in auto)
  thus ?thesis using x
    by (simp add: has_contour_integral_linepath_Reals_iff exp_of_real
flip: of_real_minus)
qed

lemma has_integral_Gamma_incl_complex_of_real':
  assumes s: "Re s > (0::real)"
  assumes "x ≤ 0"
  shows "((λt. of_real t powr (s - 1) * of_real (exp (-t)))
          has_integral (-Gamma_incl s (of_real x))) {x..0}"
proof (cases "x = 0")
```

```
  case True
  have [simp]: "s ≠ 0"
    using s by auto
  have "((λt. of_real t powr (s - 1) * of_real (exp (-t))) has_integral
0) {0}"
    by (rule has_integral_refl)
  thus ?thesis using True
    by (simp add: Gamma_incl_def)
next
  case False
  with assms have x: "x < 0"
    by auto
  have *: "((λt. t powr (s - 1) * exp (-t)) has_contour_integral Gamma_incl
s (of_real x))
         (linepath 0 (of_real x))"
    by (rule has_contour_integral_Gamma_incl) (use x s in auto)
  have "((λt. t powr (s - 1) * exp (-t)) has_contour_integral (-Gamma_incl
s (of_real x)))
         (linepath (of_real x) 0)"
    using has_contour_integral_reversepath[OF _ *] by simp
  thus ?thesis using x
    by (simp add: has_contour_integral_linepath_Reals_iff exp_of_real
flip: of_real_minus)
qed

lemma has_integral_Gamma_incl_real:
  assumes s: "s > (0::real)"
  assumes "x ≥ 0"
  shows "((λt. t powr (s - 1) * exp (-t)) has_integral Gamma_incl s x)
{0..x}"
proof -
  have "((λt. of_real t powr (of_real s - 1) * of_real (exp (-t))) has_integral

         (Gamma_incl (complex_of_real s) (of_real x))) {0..x}"
    by (rule has_integral_Gamma_incl_complex_of_real) (use assms in auto)
  also have "?this ⟷ ((λt. of_real (t powr (s - 1) * exp (-t))) has_integral

                        (Gamma_incl (complex_of_real s) (of_real x)))
{0..x}"
    by (intro has_integral_cong) (auto simp: powr_Reals_eq)
  also have "Gamma_incl (complex_of_real s) (of_real x) = of_real (Gamma_incl
s x)"
    by (rule Gamma_incl_complex_of_real) (use assms in auto)
  finally show ?thesis
    by (subst (asm) has_integral_complex_of_real_iff)
qed
```

6.4 The upper incomplete Gamma function

To make the definition work on as big a domain as possible, we do not define the upper incomplete Gamma function $\Gamma(s,z)$ as $\Gamma(s,z) = \Gamma(s) - \gamma(s,z)$, since there are values of s where the left-hand side exists but the two parts on the right-hand side blow up. Rather, we express $\Gamma(s,z)$ as a contour integral starting at z and going to ∞. The precise path does not matter much, so for convenience, we go from z straight to 1 (the first auxiliary function below) and then from 1 straight to ∞.

To make the first definition work for both the *real* and *complex* type, we express the contour integral in a more explicit fashion.

```
definition Gamma_incu_aux1 :: "'a :: {banach, real_inner, real_normed_field,
ln} ⇒ 'a ⇒ 'a" where
  "Gamma_incu_aux1 s z =
     (if (z + 1 ∈ ℝ≤0 ∧ (z ≠ -1 ∨ s · 1 ≤ -1)) ∧ s ∉ ℤ then 0 else

        integral {0..1} (λx. (1 + x *R z) powr' s * exp (- (1 + x *R
z))))"

lemma Gamma_incu_aux1_complex_of_real:
  "Gamma_incu_aux1 (complex_of_real s) (complex_of_real z) = of_real (Gamma_incu_aux1
s z)"
proof (cases "(z + 1 ∈ ℝ≤0 ∧ (z ≠ -1 ∨ s · 1 ≤ -1)) ∧ s ∉ ℤ")
  case True
  thus ?thesis
    by (auto simp: Gamma_incu_aux1_def nonpos_Reals_def complex_eq_iff)
next
  case False
  have *: "s ∈ ℤ ∨ complex_of_real z + 1 ∉ ℝ≤0 ∨ z = -1 ∧ s > -1"
    using False
    by (auto simp: complex_nonpos_Reals_iff nonpos_Reals_real_eq)

  have **: "1 + x * z ≥ 0" if x: "x ≥ 0" "x ≤ 1" and s: "s ∉ ℤ" for
x
  proof (cases "z ≤ 0")
    case True
    have "-1 ≤ 1 * z"
      using False s by (auto simp: nonpos_Reals_real_eq)
    also have "... ≤ x * z"
      by (intro mult_right_mono_neg) (use x ‹z ≤ 0› in auto)
    finally show ?thesis
      by simp
  next
    case False
    have "-1 < (0::real)"
      by simp
    also have "... ≤ x * z"
      by (intro mult_nonneg_nonneg) (use x False in auto)
```

```
    finally show ?thesis
      by simp
  qed

  have "Gamma_incu_aux1 (complex_of_real s) (complex_of_real z) =
        integral {0..1} (λx. (1 + x *R complex_of_real z) powr' complex_of_real
s *
                                exp (- 1 - x *R complex_of_real z))"
    using False * by (auto simp add: Gamma_incu_aux1_def)
  also have "... = of_real (integral {0..1} (λx. (1 + x *R z) powr' s
* exp (- 1 - x *R z)))"
    unfolding integral_complex_of_real [symmetric]
  proof (intro integral_cong)
    fix x assume x: "x ∈ {0..1::real}"
    show "(1 + x *R complex_of_real z) powr' complex_of_real s *
           exp (- 1 - x *R complex_of_real z) =
           complex_of_real ((1 + x *R z) powr' s * exp (- 1 - x *R z))"
      unfolding of_real_mult using **[of x] x False
      by (subst powr'_complex_of_real [symmetric])
         (auto simp: nonpos_Reals_real_eq scaleR_conv_of_real simp flip:
exp_of_real)
  qed
  also have "... = complex_of_real (Gamma_incu_aux1 s z)"
    using False by (auto simp: Gamma_incu_aux1_def)
  finally show ?thesis .
qed

lemma Gamma_incu_aux1_conv_contour_integral:
  assumes "z ∉ ℝ≤0 ∨ z = 0 ∧ Re s > 0 ∨ s ∈ ℤ"
  shows   "(z-1) * Gamma_incu_aux1 (s-1) (z-1) =
             contour_integral (linepath 1 z) (λu. u powr' (s - 1) * exp
(-u))"
proof -
  define I where "I = integral {0..1} (λx. (1 + x *R (z - 1)) powr' (s
- 1) * exp (-(1 + x *R (z - 1))))"
  have "Gamma_incu_aux1 (s-1) (z-1) = I"
    using assms by (auto simp: Gamma_incu_aux1_def I_def)
  also have "(z - 1) * ... = contour_integral (linepath 1 z) (λu. u powr'
(s - 1) * exp (-u))"
    unfolding contour_integral_integral integral_mult_right [symmetric]
I_def
    by (rule integral_cong) (simp_all add: linepath_def algebra_simps)
  finally show ?thesis .
qed

lemma analytic_Gamma_incu_aux1 [analytic_intros]:
  assumes "f analytic_on A" "g analytic_on A" "⋀x. x ∈ A ⟹ 1 + g x
∉ ℝ≤0"
  shows   "(λx. Gamma_incu_aux1 (f x) (g x)) analytic_on A"
```

```
proof -
  define f' where "f' = deriv f"
  define g' where "g' = deriv g"
  define h where "h = (λy x. exp (f y * ln (1 + x *R g y) - (1 + x *R
g y)))"
  define h' where
    "h' = (λy x. exp (f y * ln (1 + x *R g y) - (1 + x *R g y)) * (f'
y * ln (1 + x *R g y) +
                x *R (g' y * f y) / (1 + x *R g y) - x *R g' y))"
  have "(λx. Gamma_incu_aux1 (f x) (g x)) analytic_on {z}" if z: "z ∈
A" for z
  proof -
    from assms(1) obtain A1 where A1: "open A1" "f holomorphic_on A1"
"A ⊆ A1"
      using analytic_on_holomorphic by auto
    from assms(2) obtain A2 where A2: "open A2" "g holomorphic_on A2"
"A ⊆ A2"
      using analytic_on_holomorphic by auto
    note [holomorphic_intros] = holomorphic_on_subset[OF A1(2)] holomorphic_on_subset[OF
A2(2)]
    have [derivative_intros]: "(f has_field_derivative deriv f x) (at
x within X)"
      if "x ∈ A1" for x X
      by (rule holomorphic_derivI[OF A1(2,1) that])
    have [derivative_intros]: "(g has_field_derivative deriv g x) (at
x within X)"
      if "x ∈ A2" for x X
      by (rule holomorphic_derivI[OF A2(2,1) that])

    have cont: "continuous_on A1 f" "continuous_on A2 g" "continuous_on
A1 f'" "continuous_on A2 g'"
      unfolding f'_def g'_def
      by (intro holomorphic_on_imp_continuous_on holomorphic_intros order.refl
A1 A2)+
    note [continuous_intros] = cont[THEN continuous_on_compose2]

    have "open ((A1 ∩ A2) ∩ (λx. 1 + g x) -‘ (-ℝ≤0))"
      by (intro continuous_open_preimage holomorphic_on_imp_continuous_on
holomorphic_intros)
         (use A1(1) A2(1) in auto)
    moreover have "z ∈ (A1 ∩ A2) ∩ (λx. 1 + g x) -‘ (-ℝ≤0)"
      using z assms(3) A1(3) A2(3) by auto
    ultimately obtain r where r: "r > 0" "cball z r ⊆ ((A1 ∩ A2) ∩ (λx.
1 + g x) -‘ (-ℝ≤0))"
      unfolding open_contains_cball by blast

    have *: "1 + x *R g y ∉ ℝ≤0" if x: "x ∈ {0..1}" and y: "y ∈ cball
z r" for x y
    proof
```

```
      assume "1 + x *_R g y ∈ ℝ_≤0"
      hence "Im (g y) = 0" "1 + x * Re (g y) ≤ 0"
        by (auto simp: complex_nonpos_Reals_iff)
      from this(2) have "x ≠ 0"
        by auto
      with x have "x > 0"
        by auto
      with ‹1 + x * Re (g y) ≤ 0› have "1 + Re (g y) ≤ 1 - 1 / x"
        by (auto simp: field_simps)
      also have "... ≤ 0"
        using x ‹x > 0› by (auto simp: field_simps)
      finally have "1 + g y ∈ ℝ_≤0"
        using ‹Im (g y) = 0› by (auto simp: complex_nonpos_Reals_iff)
      with y r show False
        by auto
    qed
    have **: "1 + x *_R g y ≠ 0" if x: "x ∈ {0..1}" and y: "y ∈ cball
z r" for x y
      using *[OF x y] by auto

    have "(λy. integral (cbox 0 1) (h y)) holomorphic_on cball z r"
    proof (rule leibniz_rule_holomorphic)
      fix y :: complex assume y: "y ∈ cball z r"
      fix x :: real assume x: "x ∈ cbox 0 1"
      from ‹y ∈ cball z r› have y': "y ∈ A1" "y ∈ A2" "1 + g y ∉ ℝ_≤0"
        using r by auto
      show "((λy. h y x) has_field_derivative h' y x) (at y within cball
z r)"
        unfolding h_def using y' x *[OF _ y, of x]
        by (auto intro!: derivative_eq_intros simp: f'_def g'_def h'_def
field_simps)
    next
      fix y assume "y ∈ cball z r"
      thus "h y integrable_on cbox 0 1"
        unfolding h_def by (intro integrable_continuous continuous_intros
*) auto
    next
      show "continuous_on (cball z r × cbox 0 1) (λ(y, t). h' y t)"
        unfolding h'_def case_prod_unfold
        by (intro continuous_intros ballI * **) (use r in auto)
    qed auto
    also have "?this ⟷ (λx. Gamma_incu_aux1 (f x) (g x)) holomorphic_on
cball z r"
    proof (intro holomorphic_cong)
      fix y assume y: "y ∈ cball z r"
      have "integral {0..1} (λx. h y x) =
            integral {0..1} (λx. (1 + x *_R g y) powr' f y * exp (- 1 -
x *_R g y))"
      proof (rule integral_cong)
```

```
      fix t assume t: "t ∈ {0..1::real}"
      show "h y t = (1 + t *R g y) powr' f y * exp (- 1 - t *R g y)"
        by (subst powr'_complex)
           (use *[OF t y] in ‹auto simp: powr_def exp_diff exp_minus
field_simps exp_add h_def›)
    qed
    also have "... = Gamma_incu_aux1 (f y) (g y)"
      unfolding Gamma_incu_aux1_def using y r by (auto simp: add_ac)
    finally show "integral (cbox 0 1) (h y) = Gamma_incu_aux1 (f y) (g
y)"
      by simp
  qed auto
  finally show "(λx. Gamma_incu_aux1 (f x) (g x)) analytic_on {z}"
    using ‹r > 0› by (meson analytic_at_ball ball_subset_cball holomorphic_on_subset)
qed
thus ?thesis
  by (subst analytic_on_analytic_at) auto
qed

lemma analytic_Gamma_incu_aux1_nat [analytic_intros]:
  assumes "g analytic_on A"
  shows   "(λx. Gamma_incu_aux1 (of_nat n) (g x)) analytic_on A"
proof -
  define g' where "g' = deriv g"
  define h where "h = (λy x. (1 + x *R g y) ^ n * exp (-(1 + x *R g y)))"
  define h' where
    "h' = (λy x. (of_nat n * (1 + x *R g y) ^ (n-1) - (1 + x *R g y)
^ n) * exp (-(1 + x *R g y)) * of_real x * g' y)"
  have "(λx. Gamma_incu_aux1 (of_nat n) (g x)) analytic_on {z}" if z:
"z ∈ A" for z
  proof -
    from assms(1) obtain A2 where A2: "open A2" "g holomorphic_on A2"
"A ⊆ A2"
      using analytic_on_holomorphic by auto
    note [holomorphic_intros] = holomorphic_on_subset[OF A2(2)]
    have [derivative_intros]: "(g has_field_derivative deriv g x) (at
x within X)"
      if "x ∈ A2" for x X
      by (rule holomorphic_derivI[OF A2(2,1) that])

    have cont: "continuous_on A2 g" "continuous_on A2 g'"
      unfolding g'_def
      by (intro holomorphic_on_imp_continuous_on holomorphic_intros order.refl
A2)+
    note [continuous_intros] = cont[THEN continuous_on_compose2]
    from A2 z obtain r where r: "r > 0" "cball z r ⊆ A2"
      unfolding open_contains_cball by blast

    have "(λy. integral (cbox 0 1) (h y)) holomorphic_on cball z r"
```

```
    proof (rule leibniz_rule_holomorphic)
      fix y :: complex assume y: "y ∈ cball z r"
      fix x :: real assume x: "x ∈ cbox 0 1"
      from ‹y ∈ cball z r› have y': "y ∈ A2"
        using r by auto
      show "((λy. h y x) has_field_derivative h' y x) (at y within cball
z r)"
        unfolding h_def using y' x
        by (auto intro!: derivative_eq_intros simp: g'_def h'_def field_simps
scaleR_conv_of_real)
    next
      fix y assume "y ∈ cball z r"
      thus "h y integrable_on cbox 0 1"
        unfolding h_def by (intro integrable_continuous continuous_intros)
    next
      show "continuous_on (cball z r × cbox 0 1) (λ(y, t). h' y t)"
        unfolding h'_def case_prod_unfold
        by (intro continuous_intros ballI) (use r in auto)
    qed auto
    also have "?this ⟷ (λx. Gamma_incu_aux1 (of_nat n) (g x)) holomorphic_on
cball z r"
    proof (intro holomorphic_cong)
      fix y assume y: "y ∈ cball z r"
      have "integral {0..1} (λx. h y x) =
            integral {0..1} (λx. (1 + x *R g y) ^ n * exp (- 1 - x *R
g y))"
      proof (rule integral_cong)
        fix t assume t: "t ∈ {0..1::real}"
        show "h y t = (1 + t *R g y) ^ n * exp (- 1 - t *R g y)"
          by (auto simp: powr_def exp_diff exp_minus field_simps exp_add
h_def)
      qed
      also have "... = Gamma_incu_aux1 (of_nat n) (g y)"
        unfolding Gamma_incu_aux1_def using y r by (auto simp: add_ac)
      finally show "integral (cbox 0 1) (h y) = Gamma_incu_aux1 (of_nat
n) (g y)"
        by simp
    qed auto
    finally show "(λx. Gamma_incu_aux1 (of_nat n) (g x)) analytic_on {z}"
      using ‹r > 0› by (meson analytic_at_ball ball_subset_cball holomorphic_on_subset)
  qed
  thus ?thesis
    by (subst analytic_on_analytic_at) auto
qed


lemma continuous_on_Gamma_incu_aux1_complex':
  fixes A :: "(complex × complex) set"
  assumes A: "⋀s z. (s, z) ∈ A ⟹ (Re z ≥ -1 ∨ Im z ≠ 0) ∧ z ≠ -1"
```

```
  shows   "continuous_on A (λsz. Gamma_incu_aux1 (fst sz) (snd sz))"
proof -
  define h where "h = (λs z x. (1 + x *R z) powr' s * exp (- (1 + x *R
z)) :: complex)"
  define B :: "(complex × complex) set" where "B = {(s,z). (Re z ≥ -1
∨ Im z ≠ 0) ∧ z ≠ -1}"

  have cont: "continuous_on B (λsz. integral (cbox 0 1) (h (fst sz) (snd
sz)))"
  proof (rule integral_continuous_on_param)
    have 1: "B × cbox 0 1 ⊆ {x. 0 ≤ Re (1 + snd x *R snd (fst x)) ∨
Im (1 + snd x *R snd (fst x)) ≠ 0}"
    proof (intro subsetI CollectI, elim SigmaE, simp only: fst_conv snd_conv)
      fix sz :: "complex × complex" and t :: real
      assume sz: "sz ∈ B" and t: "t ∈ cbox 0 1"
      consider "t = 0" | "t > 0" "Im (snd sz) = 0" | "t > 0" "Im (snd
sz) ≠ 0"
        using t by force
      thus "0 ≤ Re (1 + t *R snd sz) ∨ Im (1 + t *R snd sz) ≠ 0"
      proof cases
        case 2
        have "-1 ≤ t * (-1)"
          using t by simp
        also have "t * (-1) ≤ t * Re (snd sz)"
          by (rule mult_left_mono) (use 2 sz in ‹auto simp: B_def›)
        finally show ?thesis
          using 2 by simp
      qed (use sz t in auto)
    qed

    have 2: False if "((s,z), t) ∈ B × cbox 0 1" "1 + t *R z = 0" for
s z t
    proof -
      from that have "Re z ≥ -1" "1 + t * Re z = 0"
        by (auto simp: complex_eq_iff B_def)
      have "t > 0"
        using that by (cases "t = 0") auto
      with ‹1 + t * Re z = 0› have Re_z: "Re z = -1 / t"
        by (auto simp: field_simps)
      also have "-1/t ≤ -1"
        using that ‹t > 0› by (auto simp: field_simps)
      finally have "Re z = -1"
        using ‹Re z ≥ -1› by linarith
      with that show False
        by (auto simp: complex_eq_iff B_def)
    qed

    show "continuous_on (B × cbox 0 1) (λ(sz, t). h (fst sz) (snd sz)
t)"
```

```
    unfolding case_prod_unfold h_def by (intro continuous_intros) (use
1 2 in force)+
  qed
  also have "?this ⟷ continuous_on B (λsz. Gamma_incu_aux1 (fst sz)
(snd sz))"
  proof (rule continuous_on_cong)
    fix sz assume sz: "sz ∈ B"
    show "integral (cbox 0 1) (h (fst sz) (snd sz)) = Gamma_incu_aux1
(fst sz) (snd sz)"
      using sz
      by (auto simp: Gamma_incu_aux1_def h_def complex_nonpos_Reals_iff
complex_eq_iff B_def)
  qed auto
  finally show ?thesis
    by (rule continuous_on_subset) (use A in ‹auto simp: B_def›)
qed

lemma continuous_on_Gamma_incu_aux1_complex [continuous_intros]:
  assumes "continuous_on A f" "continuous_on A g"
  assumes A: "⋀x. x ∈ A ⟹ (Re (g x) ≥ -1 ∨ Im (g x) ≠ 0) ∧ g x ≠
-1"
  shows   "continuous_on A (λx. Gamma_incu_aux1 (f x) (g x :: complex))"
proof -
  have "continuous_on A ((λx. Gamma_incu_aux1 (fst x) (snd x)) ∘ (λx.
(f x, g x)))"
    by (intro continuous_on_compose continuous_on_Gamma_incu_aux1_complex'
continuous_intros assms)
       (use A in auto)
  thus ?thesis
    by (simp add: o_def)
qed

lemma continuous_Gamma_incu_aux1_at_neg1_aux:
  defines "A ≡ {(s,z). Re s > -1 ∧ Re z > -1}"
  assumes "Re s > -1"
  shows   "((λ(s,z). Gamma_incu_aux1 s z) ⟶ Gamma_incu_aux1 s (-1))
(at (s, -1) within A)"
proof -
  define h where "h = (λs z x. (1 + x *R z) powr' s * exp (- (1 + x *R
z)) :: complex)"
  define c where "c = |Im s| + 1"
  define d where "d = (Re s - 1) / 2"
  define e where "e = Re s + 1"
  define H where "H = (λt::real. (1 - t) powr d + (1 + 2 * t) powr e)"

  have *: "Re (1 + t *R z) > 0" if t: "t ∈ {0..1}" and z: "Re z > -1"
for t z
  proof (cases "t = 0")
    case False
```

```
    hence "t * (-Re z) < t * 1"
      by (intro mult_strict_left_mono) (use z t in ‹auto simp: A_def›)
    also have "... ≤ 1"
      using t by simp
    finally show ?thesis
      by simp
  qed auto

  have "((λ(s,z). integral {0..<1} (h s z)) ⟶ integral {0..<1} (h
s (-1))) (at (s,-1) within A)"
    unfolding case_prod_unfold
  proof (rule at_within.dominated_convergence')
    have "∀F (s,z) in at (s, -1) within A. (s,z) ∈ A"
      by (auto simp: eventually_at_topological)
    moreover have "∀F sz in at (s, -1::complex).
                    sz ∈ ({s. Im s < c} ∩ {s. Im s > -c} ∩ {s. Re s
> d} ∩ {s. Re s < e}) × ball 0 2"
      using assms
      by (intro eventually_at_in_open' open_Times open_Int open_halfspace_Re_gt
               open_halfspace_Re_lt open_halfspace_Im_gt open_halfspace_Im_lt)
         (auto simp: c_def d_def e_def dist_norm)
    hence "∀F sz in at (s, -1) within A.
             sz ∈ ({s. Im s < c} ∩ {s. Im s > -c} ∩ {s. Re s > d} ∩ {s.
Re s < e}) × ball 0 2"
      by (rule filter_leD [OF at_within_le_at])
    ultimately show "∀F sz in at (s, -1) within A. ∀t∈{0..<1}. norm
(h (fst sz) (snd sz) t) ≤
                       exp (c * pi) * H t"
    proof eventually_elim
      case (elim sz)
      obtain s z where [simp]: "sz = (s, z)"
        by (cases sz)
      show ?case
      proof safe
        fix t assume t: "t ∈ {0..<1::real}"
        have nz: "1 + t *R z ≠ 0"
          using *[of t z] t elim by (auto simp: A_def complex_eq_iff)
        have "norm (h (fst sz) (snd sz) t) = norm ((1 + t *R z) powr'
s) * exp (- 1 - t * Re z)"
          by (auto simp: h_def norm_mult)
        also have "... = norm ((1 + t *R z) powr s) * exp (- 1 - t * Re
z)"
          by (subst powr'_complex) (use nz in auto)
        also have "... = norm (1 + t *R z) powr Re s * exp (- (Im s *
Arg (1 + t *R z))) * exp (- 1 - t * Re z)"
          by (simp add: norm_powr_complex)
        also have "... ≤ norm (1 + t *R z) powr Re s * exp (pi * c) *
exp 0"
        proof (intro mult_mono mult_nonneg_nonneg exp_mono)
```

```
      have "1 * (-1) ≤ t * (-1)"
        by (rule mult_right_mono_neg) (use t in auto)
      also have "t * (-1) ≤ t * Re z"
        by (rule mult_left_mono) (use t elim in ‹auto simp: A_def›)
      finally show "- 1 - t * Re z ≤ 0"
        by simp
    next
      have "-(Im s * Arg (1 + t *R z)) ≤ |Im s * Arg (1 + t *R z)|"
        by linarith
      also have "... = |Im s| * |Arg (1 + t *R z)|"
        by (simp add: abs_mult)
      also have "... ≤ c * pi"
        by (intro mult_mono) (use elim Arg_bounded[of "1 + t *R z"]
in auto)
      finally show "-(Im s * Arg (1 + t *R z)) ≤ pi * c"
        by (simp add: mult_ac)
    qed auto

    also have "... ≤ H t * exp (pi * c) * exp 0"
    proof (intro mult_right_mono)
      show "norm (1 + t *R z) powr Re s ≤ H t"
      proof (cases "Re s ≤ 0")
        case True
        have "norm (1 + t *R z) powr Re s ≤ (1 - t) powr Re s"
        proof (intro powr_mono2')
          have "1 + t * (-1) ≤ 1 + t * Re z"
            by (intro add_mono mult_left_mono) (use t elim in ‹auto
simp: A_def›)
          also have "t * Re z ≥ t * (-1)"
            by (intro mult_left_mono) (use t elim in ‹auto simp: A_def›)
          hence "1 + t * Re z ≤ |Re (1 + t *R z)|"
            by simp
          also have "|Re (1 + t *R z)| ≤ norm (1 + t *R z)"
            by (rule abs_Re_le_cmod)
          finally show "norm (1 + t *R z) ≥ 1 - t"
            by simp
        qed (use ‹Re s ≤ 0› t in auto)
        also have "... ≤ (1 - t) powr d"
          by (intro powr_mono') (use t elim True in auto)
        also have "... ≤ H t"
          using True by (simp add: H_def)
        finally show ?thesis .
      next
        case False
        have "norm (1 + t *R z) powr Re s ≤ (1 + 2 * t) powr Re s"
        proof (intro powr_mono2)
          have "norm (1 + t *R z) ≤ norm (1::complex) + norm (t *R
z)"
            by (rule norm_triangle_ineq)
```

```
            also have "... = 1 + t * norm z"
              using t by simp
            also have "... ≤ 1 + t * 2"
              by (intro add_mono mult_left_mono) (use elim t in auto)
            finally show "norm (1 + t *R z) ≤ (1 + 2 * t)"
              by (simp add: mult_ac)
          qed (use False in auto)
          also have "... ≤ (1 + 2 * t) powr e"
            by (intro powr_mono) (use t elim in auto)
          also have "... ≤ H t"
            by (simp add: H_def)
          finally show ?thesis .
        qed
      qed auto
      finally show "norm (h (fst sz) (snd sz) t) ≤ exp (c * pi) * H
t"
        by (simp add: mult_ac)
    qed
  qed
next
  have "∀F (s,z) in at (s, -1) within A. (s,z) ∈ A"
    by (auto simp: eventually_at_topological)
  thus "∀F sz in at (s, -1) within A. h (fst sz) (snd sz) integrable_on
{0..<1}"
  proof eventually_elim
    case (elim sz)
    obtain s z where sz [simp]: "sz = (s, z)"
      by (cases sz)
    have z: "Re z > -1"
      using elim by (auto simp: A_def)
    have "continuous_on {0..1} (h (fst sz) (snd sz))"
      unfolding h_def sz snd_conv fst_conv using *[OF _ z]
      by (intro continuous_intros subsetI CollectI) fastforce+
    hence "h (fst sz) (snd sz) integrable_on {0..1}"
      by (rule integrable_continuous_real)
    also have "?this ⟷ ?case"
      by (rule integrable_spike_set_eq[OF negligible_subset[of "{1}"]])
auto
    finally show ?case .
  qed
next
  have d: "d > -1"
    using assms by (auto simp: d_def)
  have "(λx. x powr d) integrable_on {0..1}"
    by (rule integrable_on_powr_from_0) (use d in auto)
  also have "?this ⟷ ((λx. (1 + x) powr d) integrable_on {- 1..0})
"
    by (subst Henstock_Kurzweil_Integration.integrable_shift_real_ivl_iff
[of _ "-1", symmetric]) simp
```

```
    also have "... ⟷ (λx. (1 - x) powr d) integrable_on {0..1}"
      by (subst Henstock_Kurzweil_Integration.integrable_reflect_real
[symmetric]) simp
    also have "... ⟷ (λx. (1 - x) powr d) integrable_on {0..<1}"
      by (rule integrable_spike_set_eq[OF negligible_subset[of "{1}"]])
auto
    finally have I1: "(λt. (1 - t) powr d) integrable_on {0..<1}" .

    have "(λt. (1 + 2 * t) powr e) integrable_on {0..1}"
      by (rule integrable_continuous_real) (auto intro!: continuous_intros)
    also have "?this ⟷ (λt. (1 + 2 * t) powr e) integrable_on {0..<1}"
      by (rule integrable_spike_set_eq[OF negligible_subset[of "{1}"]])
auto
    finally have I2: "(λt. (1 + 2 * t) powr e) integrable_on {0..<1}" .

    show "(λt. exp (c * pi) * H t) integrable_on {0..<1}"
      unfolding H_def by (intro integrable_on_mult_right integrable_add
I1 I2)
  next
    fix t assume t: "t ∈ {0..<1::real}"
    have "1 + t *_R (-1 :: complex) ∉ ℝ_{≤0}"
      using t by (auto simp: complex_nonpos_Reals_iff)
    thus "((λp. h (fst p) (snd p) t) ⟶ h s (- 1) t) (at (s, - 1) within
A)"
      unfolding h_def using t by (intro tendsto_intros) (auto intro!:
tendsto_eq_intros)
  qed

  also have "integral {0..<1} (h s (-1)) = integral {0..1} (h s (-1))"
    by (rule integral_subset_negligible) auto
  also have "... = Gamma_incu_aux1 s (-1)"
    using assms(2) by (auto simp: Gamma_incu_aux1_def h_def)
  also have "eventually (λsz. sz ∈ A) (at (s, -1) within A)"
    by (auto simp: eventually_at_topological)
  hence "eventually (λ(s,z). integral {0..<1} (h s z) = Gamma_incu_aux1
s z) (at (s, -1) within A)"
  proof eventually_elim
    case (elim sz)
    obtain s z where [simp]: "sz = (s, z)"
      by (cases sz)
    have "integral {0..<1} (h s z) = integral {0..1} (h s z)"
      by (rule integral_subset_negligible) auto
    also have "... = Gamma_incu_aux1 s z"
      using elim by (auto simp: Gamma_incu_aux1_def h_def complex_nonpos_Reals_iff
A_def)
    finally show ?case
      by simp
  qed
  hence "((λ(s,z). integral {0..<1} (h s z)) ⟶ Gamma_incu_aux1 s (-
```

```
1)) (at (s, -1) within A) ⟷
        ((λ(s,z). Gamma_incu_aux1 s z) ⟶ Gamma_incu_aux1 s (-1))
(at (s, -1) within A)"
    by (intro filterlim_cong) (auto simp: case_prod_unfold)
  finally show ?thesis .
qed



definition Gamma_incu_aux2 :: "'a :: {banach, real_normed_algebra_1} ⇒
'a" where
  "Gamma_incu_aux2 s = integral {1..} (λt. exp ((s-1) * of_real (ln t)
- of_real t))"

lemma Gamma_incu_aux2_complex_of_real:
  "Gamma_incu_aux2 (complex_of_real s) = of_real (Gamma_incu_aux2 s)"
  unfolding Gamma_incu_aux2_def integral_complex_of_real [symmetric]
  by (rule integral_cong) (auto simp flip: exp_of_real)

lemma absolutely_integrable_incomplete_Gamma:
  fixes s :: complex
  assumes r: "r > 0"
  shows   "(λt. exp (of_real (-t) - s * of_real (ln t))) absolutely_integrable_on
{r..}"
proof -
  have "set_integrable lborel {r..} (λt. exp (of_real (-t) - s * of_real
(ln t)))"
  proof (rule set_integrable_bigo)
    have "(λt. norm (exp (of_real (-t) - s * of_real (ln t)))) = (λt.
exp (-t - Re s * ln t))"
      by simp
    also have "... ∈ O(λt. exp (-t/2))"
      by real_asymp
    finally have "(λt. norm (exp (of_real (-t) - s * of_real (ln t))))
∈
                    O(λt. norm (complex_of_real (exp (-t/2))))"
      by simp
    thus "(λt. exp (of_real (-t) - s * of_real (ln t))) ∈ O(λt. complex_of_real
(exp (-t/2)))"
      unfolding landau_o.big.norm_iff .
  next
    have "(λt. exp (-t/2)) integrable_on {r..}"
      using integrable_on_exp_minus_to_infinity[of "1/2" r] by simp
    hence "(λt. exp (-t/2)) absolutely_integrable_on {r..}"
      by (simp add: nonnegative_absolutely_integrable_1)
    hence "set_integrable lborel {r..} (λt. exp (-t/2))"
      unfolding set_integrable_def by (subst (asm) integrable_completion)
auto
    hence "integrable lborel (λt. complex_of_real (indicator {r..} t *
```

```
exp (-t/2)))"
      by (intro integrable_of_real) (simp_all add: set_integrable_def)
    thus "set_integrable lborel {r..} (λt. complex_of_real (exp (- t /
2)))"
      by (simp add: set_integrable_def of_real_indicator scaleR_conv_of_real)
  next
    show "set_integrable lborel {r..<b} (λt. exp (of_real (-t) - s *
of_real (ln t)))" if "r ≤ b" for b
    proof (rule set_integrable_subset)
      show "set_integrable lborel {r..b} (λt. exp (of_real (-t) - s *
of_real (ln t)))"
        by (rule borel_integrable_atLeastAtMost')
           (use ‹r ≤ b› ‹r > 0› in ‹auto intro!: continuous_intros›)
    qed auto
  qed (auto simp: set_borel_measurable_def)
  thus ?thesis
    unfolding set_integrable_def by (subst integrable_completion) auto
qed

lemma analytic_Gamma_incu_aux2_aux:
  fixes r :: real and f :: "complex ⇒ complex"
  assumes r: "r > 0"
  defines "f ≡ (λz. integral {r..} (λt. exp (-of_real t - z * of_real
(ln t))))"
  shows "f holomorphic_on UNIV"
proof -
  define g where "g = (λx z. LBINT t:{r..x}. exp (-of_real t - z * complex_of_real
(ln t)))"
  define f' where "f' = (λz. LBINT t:{r..}. exp (-of_real t - z * complex_of_real
(ln t)))"

  have "f' analytic_on {s}" for s
  proof -
    define A where "A = {z. Re z > -|Re s| - 1} ∩ {z. Re z < |Re s| + 1}"
    define c where "c = (|Re s| + 1)"
    define h :: "real ⇒ real" where "h = (λt. exp (-t + c * |ln t|))"

    have "open A"
      unfolding A_def by (intro open_Int open_halfspace_Re_lt open_halfspace_Re_gt)
    moreover have "s ∈ A"
      by (auto simp: A_def)
    ultimately obtain R where R: "R > 0" "cball s R ⊆ A"
      using open_contains_cball by blast

    have "uniform_limit A g f' at_top"
      unfolding g_def f'_def
    proof (rule uniform_limit_set_lebesgue_integral_at_top)
      show "set_integrable lborel {r..} h"
      proof (rule set_integrable_bigo)
```

```
      show "h ∈ O(λt. exp (-t/2))"
        unfolding h_def by real_asymp
    next
      have "(λt. exp (-t/2)) integrable_on {r..}"
        using integrable_on_exp_minus_to_infinity[of "1/2" r] by simp
      hence "(λt. exp (-t/2)) absolutely_integrable_on {r..}"
        by (simp add: nonnegative_absolutely_integrable_1)
      thus "set_integrable lborel {r..} (λt. exp (-t/2))"
        unfolding set_integrable_def by (subst (asm) integrable_completion)
auto
    next
      fix b assume "b ≥ r"
      have "set_integrable lborel {r..b} h" unfolding h_def
        by (intro borel_integrable_atLeastAtMost' continuous_intros)
(use r in auto)
      thus "set_integrable lborel {r..<b} h"
        by (rule set_integrable_subset) auto
    qed (auto simp: h_def set_borel_measurable_def)
  next
    fix z t assume z: "z ∈ A" and t: "t ≥ r"
    have "|Re z| < |Re s| + 1"
      using z unfolding A_def by auto
    have "norm (exp (-of_real t - z * of_real (ln t))) = exp (-t - Re
z * ln t)"
      by simp
    also have "-t - Re z * ln t ≤ -t + (|Re s| + 1) * |ln t|"
    proof -
      have "-Re z * ln t ≤ |Re z * ln t|"
        by linarith
      have "-Re z * ln t ≤ |Re z * ln t|"
        by linarith
      also have "... ≤ (|Re s| + 1) * |ln t|"
        unfolding abs_mult by (intro mult_right_mono) (use <|Re z| <
|Re s| + 1> in auto)
      finally show ?thesis
        by simp
    qed
    finally show "norm (exp (-of_real t - z * of_real (ln t))) ≤ h t"
      by (simp add: h_def c_def)
  qed (simp_all add: set_borel_measurable_def)
  hence lim: "uniform_limit (cball s R) g f' at_top"
    by (rule uniform_limit_on_subset) (use R in auto)

  have holo: "g b holomorphic_on UNIV" for b
  proof -
    define h :: "complex ⇒ real ⇒ complex"
      where "h = (λz t. exp (-of_real t - z * of_real (ln t)))"
    define h' where "h' = (λz t. -h z t * ln t)"
```

```
      have "(λz. integral (cbox r b) (h z)) holomorphic_on UNIV"
      proof (rule leibniz_rule_holomorphic)
        show "h z integrable_on cbox r b" for z
          by (rule integrable_continuous) (use r in ‹auto simp: h_def
intro!: continuous_intros›)
      next
        show "((λz. h z t) has_field_derivative h' z t) (at z)"
          if t: "t ∈ cbox r b" for z t
          by (auto intro!: derivative_eq_intros simp: h_def h'_def)
      next
        show "continuous_on (UNIV × cbox r b) (λ(z, t). h' z t)" us-
ing r
          by (auto simp: h'_def h_def case_prod_unfold intro!: continuous_intros)
      qed auto
      also have "(λz. integral (cbox r b) (h z)) = g b"
      proof
        fix z :: complex
        have "set_integrable lborel {r..b} (h z)" unfolding h_def
          by (intro borel_integrable_atLeastAtMost') (use r in ‹auto intro!:
continuous_intros›)
        thus "integral (cbox r b) (h z) = g b z" unfolding g_def
          by (subst set_borel_integral_eq_integral(2)) (simp_all add:
h_def)
      qed
      finally show ?thesis
        by (simp add: g_def)
    qed

    have wf: "∀F b in at_top. continuous_on (cball s R) (g b) ∧ g b holomorphic_on
ball s R"
      by (intro always_eventually allI impI conjI holomorphic_on_subset[OF
holo]
                holomorphic_on_imp_continuous_on) auto
    have "f' holomorphic_on ball s R"
      using holomorphic_uniform_limit[OF wf lim] by auto
    with ‹R > 0› show "f' analytic_on {s}"
      using analytic_at_ball by blast
  qed
  hence "f' holomorphic_on UNIV"
    using analytic_imp_holomorphic analytic_on_analytic_at by blast
  also have "?this ⟷ f holomorphic_on UNIV"
  proof (intro holomorphic_cong)
    fix z :: complex
    have "set_integrable lborel {r..} (λx. exp (-complex_of_real x - z
* complex_of_real (ln x)))"
      using absolutely_integrable_incomplete_Gamma[of r] r
      by (simp add: set_integrable_def integrable_completion)
    thus "f' z = f z"
      unfolding f_def f'_def by (rule set_borel_integral_eq_integral(2))
```

```
  qed auto
  finally show ?thesis .
qed

lemma analytic_Gamma_incu_aux2 [analytic_intros]:
  assumes "f analytic_on A"
  shows   "(λx. Gamma_incu_aux2 (f x)) analytic_on A"
proof -
  have "(Gamma_incu_aux2 ∘ f) analytic_on A"
  proof (rule analytic_on_compose_gen)
    have "((λz. integral {1..} (λt. exp (-complex_of_real t - z * of_real
(ln t)))) ∘ (λz. 1 - z))
            holomorphic_on UNIV"
      by (rule holomorphic_on_compose_gen[OF _ analytic_Gamma_incu_aux2_aux])

         (auto intro!: holomorphic_intros)
    thus "Gamma_incu_aux2 analytic_on UNIV"
      by (simp add: Gamma_incu_aux2_def [abs_def] o_def algebra_simps
analytic_on_open)
  qed (use assms in auto)
  thus ?thesis
    by (simp add: o_def)
qed

lemma continuous_on_Gamma_incu_aux2_complex [continuous_intros]:
  assumes "continuous_on A f"
  shows   "continuous_on A (λx. Gamma_incu_aux2 (f x :: complex))"
proof (rule continuous_on_compose2[OF _ assms])
  show "continuous_on (UNIV :: complex set) Gamma_incu_aux2"
    by (intro analytic_imp_holomorphic holomorphic_on_imp_continuous_on
analytic_intros)
qed auto

lemma continuous_Gamma_incu_aux2_complex [continuous_intros]:
  assumes "continuous (at x within A) f"
  shows   "continuous (at x within A) (λx. Gamma_incu_aux2 (f x :: complex))"
proof (rule continuous_within_compose3[OF _ assms])
  show "isCont Gamma_incu_aux2 (f x)"
    by (intro analytic_at_imp_isCont analytic_intros)
qed
```

Finally, we can define the upper incomplete Gamma function $\Gamma(s, z)$:

```
definition Gamma_incu :: "'a :: {banach, real_inner, real_normed_field,
ln} ⇒ 'a ⇒ 'a"
  where "Gamma_incu s z = Gamma_incu_aux2 s - (z - 1) * Gamma_incu_aux1
(s - 1) (z - 1)"

lemma Gamma_incu_complex_of_real:
  "Gamma_incu (complex_of_real s) (of_real z) = of_real (Gamma_incu s
```

```
z)"
  by (simp add: Gamma_incu_def flip: Gamma_incu_aux1_complex_of_real Gamma_incu_aux2_comple
```

In general, $\Gamma(s,z)$ is analytic away from $z \leq 0$ (where it has a branch cut).

```
lemma analytic_Gamma_incu [analytic_intros]:
  assumes "f analytic_on A" "g analytic_on A" "⋀x. x ∈ A ⟹ g x ∉
ℝ≤0"
  shows   "(λx. Gamma_incu (f x) (g x)) analytic_on A"
  unfolding Gamma_incu_def by (intro analytic_intros assms(1,2)) (use
assms(3) in auto)
```

For s a positive integer, $\Gamma(s,z)$ is entire in z:

```
lemma analytic_Gamma_incu_pos_int [analytic_intros]:
  assumes "g analytic_on A" "n > 0"
  shows   "(λx. Gamma_incu (of_int n) (g x)) analytic_on A"
proof -
  have "(λx. Gamma_incu_aux2 (complex_of_int n) -
         (g x - 1) * Gamma_incu_aux1 (of_nat (nat n - 1)) (g x - 1))
analytic_on A"
    by (intro analytic_intros assms(1))
  thus ?thesis
    using assms(2) by (simp add: Gamma_incu_def)
qed

lemma continuous_on_Gamma_incu_complex [continuous_intros]:
  fixes x z :: "'a :: topological_space ⇒ complex"
  assumes [continuous_intros]: "continuous_on X s" "continuous_on X z"
  assumes "⋀x. x ∈ X ⟹ Re (z x) > 0 ∨ Im (z x) ≠ 0"
  shows   "continuous_on X (λx. Gamma_incu (s x) (z x))"
  unfolding Gamma_incu_def by (auto intro!: continuous_intros dest!: assms(3))

lemma tendsto_Gamma_incu_complex [tendsto_intros]:
  fixes s z :: complex
  assumes "(f ⟶ s) F" "(g ⟶ z) F" "Re z > 0 ∨ Im z ≠ 0"
  shows   "((λx. Gamma_incu (f x) (g x)) ⟶ Gamma_incu s z) F"
proof -
  have "continuous_on {(s,z). Re z > 0 ∨ Im z ≠ 0} (λ(s,z). Gamma_incu
s z)"
    by (auto simp: case_prod_unfold intro!: continuous_intros)
  hence "((λx. case (f x, g x) of (s, z) ⇒ Gamma_incu s z) ⟶
          (case (s, z) of (s, z) ⇒ Gamma_incu s z)) F"
  proof (rule continuous_on_tendsto_compose)
    have "∀F x in F. g x ∈ {z. Re z > 0} ∪ {z. Im z > 0} ∪ {z. Im z
< 0}"
      by (intro eventually_compose_filterlim[OF eventually_nhds_in_open
assms(2)] open_Un)
         (use assms(3) in ‹auto simp: open_halfspace_Im_gt open_halfspace_Im_lt
open_halfspace_Re_gt›)
    thus "∀F x in F. (f x, g x) ∈ {(s, z). 0 < Re z ∨ Im z ≠ 0}"
```

```
    by eventually_elim auto
  qed (use assms in ‹auto simp: case_prod_unfold intro: tendsto_intros›)
  thus ?thesis
    by simp
qed

lemma tendsto_Gamma_incu_real [tendsto_intros]:
  fixes s z :: real
  assumes "(f ⟶ s) F" "(g ⟶ z) F" "z > 0"
  shows   "((λx. Gamma_incu (f x) (g x)) ⟶ Gamma_incu s z) F"
proof -
  have "((λx. Re (Gamma_incu (of_real (f x)) (of_real (g x)))) ⟶
        Re (Gamma_incu (of_real s) (of_real z))) F"
    by (rule tendsto_intros assms(1,2))+ (use assms in auto)
  thus ?thesis
    by (simp add: Gamma_incu_complex_of_real)
qed

lemma continuous_Gamma_incu_complex [continuous_intros]:
  fixes s z :: "_ ⇒ complex"
  assumes "continuous (at x within A) s" "continuous (at x within A)
z"
  assumes "Re (z x) > 0 ∨ Im (z x) ≠ 0"
  shows   "continuous (at x within A) (λx. Gamma_incu (s x) (z x))"
  using assms unfolding continuous_def
  by (cases "at x within A = bot") (auto simp: Lim_ident_at intro: tendsto_intros)

lemma continuous_Gamma_incu_real [continuous_intros]:
  fixes s z :: "_ ⇒ real"
  assumes "continuous (at x within A) s" "continuous (at x within A)
z"
  assumes "z x > 0"
  shows   "continuous (at x within A) (λx. Gamma_incu (s x) (z x))"
  using assms unfolding continuous_def
  by (cases "at x within A = bot") (auto simp: Lim_ident_at intro: tendsto_intros)
```

The behaviour at the branch point $z = 0$ is also interesting: when approaching from the right (i.e. $\mathrm{Re}(s) > 0$ and $\mathrm{Re}(z) > 0$, the function $\Gamma(s, z)$ converges as $z \to 0$. We will later see that it converges to $\Gamma(s)$.

```
lemma continuous_Gamma_incu_0_strong_complex:
  assumes "Re s > 0"
  defines "F ≡ (at (s,0) within ({s. Re s > 0} × {z. Re z > 0}))"
  shows "continuous F (λ(s,z). Gamma_incu s z)"
proof -
  define f where "f = (λ(s,z). Gamma_incu_aux1 s z :: complex)"
  define g where "g = (λ(s,z). (s - 1 :: complex, z - 1 :: complex))"
  define F' where "F' = (at (s-1, -1) within ({s. Re s > -1} × {z. Re
z > -1}))"
  have 1: "filterlim f (nhds (Gamma_incu_aux1 (s-1) (-1))) F'"
```

```
    using continuous_Gamma_incu_aux1_at_neg1_aux[of "s-1"] assms(1) by
(simp add: f_def F'_def)
  have 2: "filterlim g F' F"
    unfolding F'_def
  proof (rule filterlim_at_withinI)
    show "(g ⟶ (s - 1, - 1)) F"
      by (auto simp: g_def case_prod_unfold F_def intro!: tendsto_eq_intros)
  next
    have "∀⇩F x in F. x ∈ {s. 0 < Re s} × {z. 0 < Re z} - {(s, 0)}"
      by (auto simp: F_def eventually_at_topological)
    thus "∀⇩F x in F. g x ∈ {s. - 1 < Re s} × {z. - 1 < Re z} - {(s -
1, - 1)}"
      by eventually_elim (auto simp: g_def)
  qed
  have "continuous F (λx. Gamma_incu_aux1 (fst x - 1) (snd x - 1))"
    using filterlim_compose[OF 1 2]
    by (cases "F = bot")
       (simp_all add: continuous_def f_def g_def o_def case_prod_unfold
F_def Lim_ident_at)
  thus ?thesis
    unfolding Gamma_incu_def case_prod_unfold F_def by (intro continuous_intros)
qed

lemma continuous_Gamma_incu_0_complex:
  assumes "Re s > 0"
  shows "continuous (at 0 within {z. Re z > 0}) (λz. Gamma_incu s z)"
proof -
  define f where "f = (λ(s,z). Gamma_incu s z :: complex)"
  define g where "g = (λz::complex. (s, z::complex))"
  have "continuous (at 0 within {z. Re z > 0}) (f ∘ g)"
  proof (rule continuous_within_compose)
    show "continuous (at 0 within {z. 0 < Re z}) g"
      by (auto simp: g_def intro!: continuous_intros)
  next
    have "continuous (at (s,0) within ({s. Re s > 0} × {z. Re z > 0}))
f"
      unfolding f_def by (rule continuous_Gamma_incu_0_strong_complex)
fact
    also have "(s, 0) = g 0"
      by (simp add: g_def)
    finally show "continuous (at (g 0) within g ‘ {z. 0 < Re z}) f"
      by (rule continuous_within_subset) (use assms(1) in ‹auto simp:
g_def›)
  qed
  thus ?thesis
    by (simp add: f_def g_def case_prod_unfold o_def)
qed
```

It is also straightforward to show that:

$$\frac{\mathrm{d}}{\mathrm{d}z}\,\Gamma(s,z) = -z^{s-1}\exp(-z)$$

This is, unsurprisingly, the opposite of the derivative of $\gamma(s,z)$.

```
lemma has_field_derivative_Gamma_incu_complex:
  assumes z: "s ∈ (ℤ-ℤ≤0) ∨ (z :: complex) ∉ ℝ≤0"
  shows   "((λz. Gamma_incu s z) has_field_derivative (-(z powr' (s-1)
* exp (-z)))) (at z within A)"
proof -
  define A where "A = (if s ∈ ℤ-ℤ≤0 then UNIV else -ℝ≤0 :: complex
set)"
  show ?thesis
  proof (rule has_field_derivative_at_within)
    define h where "h = (λu. u powr' (s - 1) * exp (-u))"
    have "((λz. contour_integral (linepath 1 z) h) has_field_derivative
h z) (at z)"
    proof (rule contour_integral_linepath_has_field_derivative)
      show "open A" "z ∈ A" "(1::complex) ∈ A"
        using z by (auto simp: A_def)
    next
      show "closed_segment 1 z ⊆ A"
      proof (cases "s ∈ (ℤ-ℤ≤0)")
        case False
        have "closed_segment (complex_of_real 1) z ⊆ - complex_of_real
` {..0}"
          by (rule starlike_slotted_complex_plane_left_aux) (use False
z in auto)
        also have "complex_of_real ` {..0} = ℝ≤0"
          by (auto simp: nonpos_Reals_def)
        finally show ?thesis
          using False by (simp add: A_def)
      qed (auto simp: A_def)
    next
      show "h holomorphic_on A"
      proof (cases "s ∈ (ℤ-ℤ≤0)")
        case False
        thus ?thesis
          unfolding h_def by (auto intro!: holomorphic_intros simp: A_def)
      next
        case True
        then obtain n where n: "s = of_int n" "n > 0"
          by (auto elim!: Ints_cases simp: of_int_in_nonpos_Ints_iff)
        have *: "s - 1 = of_nat (nat (n - 1))"
          using n by auto
        show ?thesis
          unfolding h_def * powr'_of_nat by (intro holomorphic_intros)
      qed
    qed
```

```
    also have "?this ⟷ ((λz. (z - 1) * Gamma_incu_aux1 (s - 1) (z -
1)) has_field_derivative h z) (at z)"
    proof (intro DERIV_cong_ev)
      have "eventually (λx. x ∈ A) (nhds z)"
        by (rule eventually_nhds_in_open) (use z in ‹auto simp: A_def›)
      thus "∀F x in nhds z. contour_integral (linepath 1 x) h = (x -
1) * Gamma_incu_aux1 (s - 1) (x - 1)"
      proof eventually_elim
        case (elim x)
        thus ?case
          unfolding h_def
          by (intro ext Gamma_incu_aux1_conv_contour_integral [symmetric])
             (auto simp: A_def split: if_splits)
      qed
    qed auto
    finally have [derivative_intros]:
      "((λz. (z - 1) * Gamma_incu_aux1 (s - 1) (z - 1)) has_field_derivative
h z) (at z)" .
    have "((λz. Gamma_incu s z) has_field_derivative (0 - h z)) (at z)"
      unfolding Gamma_incu_def by (rule derivative_eq_intros refl)+
    also have "0 - h z = -(z powr' (s - 1) * exp (-z))"
      using assms by (auto simp: h_def exp_diff exp_add field_simps powr_def
exp_minus)
    finally show "(Gamma_incu s has_field_derivative - (z powr' (s - 1)
* exp (- z))) (at z)" .
  qed
qed

lemma has_field_derivative_Gamma_incu_complex' [derivative_intros]:
  assumes "(f has_field_derivative f') (at z within A)" "s ∈ (ℤ-ℤ≤0)
∨ (f z :: complex) ∉ ℝ≤0"
  shows   "((λz. Gamma_incu s (f z))
             has_field_derivative (-(f z powr' (s-1) * exp (-f z)) * f'))
(at z within A)"
  using DERIV_chain[OF has_field_derivative_Gamma_incu_complex[OF assms(2)]
assms(1)]
  by (simp add: o_def)

lemma has_field_derivative_Gamma_incu_real:
  assumes "(f has_field_derivative f') (at x within A)" "s ∈ (ℤ-ℤ≤0)
∨ (f x :: real) > 0"
  shows   "((λx. Gamma_incu s (f x)) has_field_derivative
             (-(f x powr' (s-1) * exp (-f x) * f'))) (at x within A)"
proof -
  have *: "(Gamma_incu s has_real_derivative - (x powr' (s - 1) * exp
(- x))) (at x)"
    if x: "s ∈ (ℤ-ℤ≤0) ∨ x > 0" for x :: real
  proof -
    have "((λx. Re (Gamma_incu (of_real s) (of_real x))) has_field_derivative
```

```
            (-(x powr' (s-1) * exp (-x)))) (at x)"
    proof -
      have x': "complex_of_real s ∈ ℤ - ℤ_{≤0} ∨ complex_of_real x ∉ ℝ_{≤0}"
        using x by (auto simp: of_real_in_nonpos_Ints_iff)
      have "((λx. Re (Gamma_incu (of_real s) (of_real x))) has_real_derivative

              Re (-(of_real x powr' of_real (s - 1) * exp (-of_real x))))
(at x)"
        by (rule derivative_eq_intros refl x')+ simp
      also have "-(of_real x powr' of_real (s - 1) * exp (-of_real x))
=
                  complex_of_real (-(x powr' (s - 1)) * exp (-x))"
        by (subst powr'_complex_of_real) (use x in ‹auto simp flip: exp_of_real›)
      also have "Re ... = -(x powr' (s - 1)) * exp (-x)"
        by (subst Re_complex_of_real) auto
      finally show ?thesis by simp
    qed
    also have "(λx. Re (Gamma_incu (of_real s) (of_real x))) = Gamma_incu
s"
      by (subst Gamma_incu_complex_of_real) auto
    finally show ?thesis .
  qed
  show ?thesis
    using DERIV_chain[OF *[OF assms(2)] assms(1)] by (simp add: o_def)
qed


lemma has_integral_Gamma_incu_complex_of_real':
  assumes x: "x > (0 :: real)"
  shows   "((λz. exp ((s-1) * of_real (ln z) - complex_of_real z)) has_integral

             (Gamma_incu s (of_real x))) {x..}"
proof -
  define h2 where "h2 = (λz. exp ((s-1) * of_real (ln z) - of_real z))"
  define h where "h = (λz. exp ((s-1) * ln (of_real z) - of_real z))"
  have "(h2 has_integral (Gamma_incu_aux2 s)) {1..}"
    unfolding Gamma_incu_aux2_def h2_def
    by (rule integrable_integral, rule set_lebesgue_integral_eq_integral(1))
       (use absolutely_integrable_incomplete_Gamma[of 1 "1 - s"] in ‹simp
add: algebra_simps›)
  also have "?this ⟷ (h has_integral (Gamma_incu_aux2 s)) {1..}"
    by (intro has_integral_cong) (auto simp: h2_def h_def Ln_of_real)
  finally have 1: "(h has_integral (Gamma_incu s 1)) {1..}"
    by (simp add: Gamma_incu_def)

  have "(h has_integral (Gamma_incu s (of_real x))) {x..}"
  proof (cases "x < 1")
    case True
```

```
    have 2: "(h has_integral (-(Gamma_incu s (of_real 1)) - (-Gamma_incu
s (of_real x)))) {x..1}"
      by (rule fundamental_theorem_of_calculus)
         (use x True in
           ‹auto simp: h_def Ln_of_real powr_def exp_diff exp_minus field_simps
exp_add
                        powr'_complex intro!: derivative_eq_intros›)
    have "(h has_integral
             (Gamma_incu s 1 + (-(Gamma_incu s (of_real 1)) - (-Gamma_incu
s (of_real x)))))
             ({1..} ∪ {x..1})"
      by (intro has_integral_Un 1 2) (use True in auto)
    also have "{1..} ∪ {x..1} = {x..}"
      using True by auto
    finally show ?thesis
      by (simp add: h_def)
  next
    case False
    have "(h has_integral (-(Gamma_incu s (of_real x)) - (-Gamma_incu
s (of_real 1)))) {1..x}"
      by (rule fundamental_theorem_of_calculus)
         (use x False in ‹auto simp: h_def Ln_of_real powr_def exp_diff
exp_minus field_simps
                                    exp_add powr'_complex intro!: derivative_eq_intros›)
    also have "?this ⟷ (h has_integral (-(Gamma_incu s (of_real x))
- (-Gamma_incu s (of_real 1)))) {1..<x}"
      by (rule has_integral_spike_set_eq) (rule negligible_subset[of "{x}"];
force; fail)+
    finally have 2: "(h has_integral (-(Gamma_incu s (of_real x)) - (-Gamma_incu
s (of_real 1)))) {1..<x}" .

    have "{1..<x} - {1..} = {}"
      by auto
    hence "(h has_integral
             (Gamma_incu s 1 - (-(Gamma_incu s (of_real x)) - (-Gamma_incu
s (of_real 1)))))
             ({1..} - {1..<x})"
      by (intro has_integral_setdiff 1 2) (use False x in auto)
    also have "{1..} - {1..<x} = {x..}"
      using x False by auto
    finally show ?thesis
      by simp
  qed
  also have "?this ⟷ ?thesis"
    by (intro has_integral_cong) (use x in ‹auto simp: h_def Ln_of_real›)
  finally show ?thesis .
qed

lemma has_integral_Gamma_incu_complex_of_real:
```

```
  assumes x: "x > (0 :: real)"
  shows   "((λt. complex_of_real t powr (s - 1) * of_real (exp (-t)))

             has_integral (Gamma_incu s (of_real x))) {x..}"
proof -
  have "((λz. exp ((s-1) * of_real (ln z) - complex_of_real z)) has_integral

           (Gamma_incu s (of_real x))) {x..}"
    using x by (rule has_integral_Gamma_incu_complex_of_real')
  also have "?this ⟷ ?thesis"
    using x by (intro has_integral_cong)
               (auto simp: powr_def field_simps exp_minus exp_diff exp_add
Ln_of_real exp_of_real)
  finally show ?thesis .
qed

lemma has_integral_Gamma_incu_real:
  fixes x s :: real
  assumes "x > (0::real)"
  shows "((λt. t powr (s - 1) * exp (-t)) has_integral Gamma_incu s x)
{x..}"
proof -
  have "((λt. of_real t powr (of_real s - 1) * of_real (exp (-t))) has_integral

          (Gamma_incu (complex_of_real s) (of_real x))) {x..}"
    by (rule has_integral_Gamma_incu_complex_of_real) (use assms in auto)
  also have "?this ⟷ ((λt. of_real (t powr (s - 1) * exp (-t))) has_integral

                           (Gamma_incu (complex_of_real s) (of_real x)))
{x..}"
    by (intro has_integral_cong) (use assms in ‹auto simp: powr_Reals_eq›)
  also have "Gamma_incu (complex_of_real s) (of_real x) = of_real (Gamma_incu
s x)"
    by (rule Gamma_incu_complex_of_real)
  finally show ?thesis
    by (subst (asm) has_integral_complex_of_real_iff)
qed
```

6.5 Identities

All the facts we have collected so far now allow us to prove that $\gamma(s,z) + \Gamma(s,z) = \Gamma(s)$ if $\mathrm{Re}(s) > 0$ (where the integral expressions for γ and Γ are valid). Then we use analytic continuation to lift this result to the rest of the domain.

```
lemma Gamma_incl_plus_incu_complex_aux:
  assumes "s ∉ ℤ≤0" "s - 1 ∈ ℕ ∨ z ∉ ℝ≤0"
  shows   "Gamma_incl s z + Gamma_incu s z = Gamma (s :: complex)"
proof -
```

```
  have nonpos_Reals_eq: "ℝ≤0 = complex_of_real ` {..0}"
    by (auto simp: nonpos_Reals_def)
  define A where "A = (λs::complex. if s - 1 ∈ ℕ then UNIV else -ℝ≤0
:: complex set)"
  have [simp, intro]: "open (A s)" for s
    by (auto simp: A_def)
  have conn: "connected (A s)" for s
    unfolding A_def nonpos_Reals_eq
    by (auto intro: starlike_imp_connected starlike_slotted_complex_plane_left)

  have eq1: "Gamma_incl s z + Gamma_incu s z = Gamma s" if s: "Re s >
0" and z: "z ∈ A s" for s z
  proof -
    from s have s': "s ∉ ℤ≤0"
      by (auto elim!: nonpos_Ints_cases)
    define f where "f = (λz. Gamma_incl s z + Gamma_incu s z - Gamma
s)"

    have "f z = 0"
    proof (rule analytic_continuation[where f = f])
      show "f holomorphic_on A s"
      proof (cases "s - 1 ∈ ℕ")
        case False
        thus ?thesis unfolding f_def A_def
          by (intro analytic_imp_holomorphic analytic_intros)
             (use assms s' in ‹auto simp: nonpos_Reals_def›)
      next
        case True
        note [analytic_intros del] = analytic_Gamma_incl analytic_Gamma_incu
        from True obtain n' where n': "s - 1 = of_nat n'"
          by (elim Nats_cases)
        define n where "n = int (Suc n')"
        have n: "n > 0" "s = of_int n"
          using n' by (simp_all add: n_def algebra_simps)
        show ?thesis
          unfolding f_def n(2) by (intro analytic_imp_holomorphic analytic_intros)
(use n in auto)
      qed
    next
      show "connected (A s)"
        by (rule conn)
    next
      show "complex_of_real ` {0<..} ⊆ A s"
        by (auto simp: complex_nonpos_Reals_iff A_def)
    next
      show "complex_of_real 1 islimpt complex_of_real ` {0<..}"
        by (intro islimpt_isCont_image)
           (auto intro: continuous_intros open_imp_islimpt eventually_neq_at_within)
    next
```

```
    fix z assume "z ∈ complex_of_real ` {0<..}"
    then obtain x where x: "z = of_real x" "x > 0"
      by auto
    have 1: "((λt. of_real t powr (s - 1) * of_real (exp (-t)))
              has_integral (Gamma_incu s (of_real x))) {x..}"
      by (rule has_integral_Gamma_incu_complex_of_real) (use x in auto)
    have 2: "((λt. of_real t powr (s - 1) * of_real (exp (-t)))
              has_integral Gamma_incl s (of_real x)) {0..x}"
      by (rule has_integral_Gamma_incl_complex_of_real) (use x s in
auto)
    have "((λt. of_real t powr (s - 1) * of_real (exp (-t)))
              has_integral (Gamma_incl s (of_real x) + Gamma_incu s
(of_real x))) ({0..x} ∪ {x..})"
      by (intro has_integral_Un 1 2) (use x in auto)
    also have "{0..x} ∪ {x..} = {0..}"
      using x by auto
    finally have "((λt. of_real t powr (s - 1) * of_real (exp (-t)))

                    has_integral (Gamma_incl s z + Gamma_incu s z))
{0..}" by (simp add: x)
    moreover have "((λt. of_real t powr (s - 1) * of_real (exp (-t)))

                    has_integral (Gamma s)) {0..}"
      using Gamma_integral_complex[of s] s by (simp add: exp_minus field_simps)
    ultimately have "Gamma_incl s z + Gamma_incu s z = Gamma s"
      by (rule has_integral_unique)
    thus "f z = 0"
      by (simp add: f_def)
  qed (use z in ‹auto simp: A_def›)
  thus ?thesis
    by (simp add: f_def)
qed

have eq2: "Gamma_incl s z + Gamma_incu s z = Gamma s"
  if s: "s ∉ ℤ≤0" and z: "z ∉ ℝ≤0" for s z :: complex
proof -
  define g where "g = (λs. Gamma_incl s z + Gamma_incu s z - Gamma
s)"
  have "g s = 0"
  proof (rule analytic_continuation_open[where f = g])
    show "g holomorphic_on (-ℤ≤0)" unfolding g_def using z
      by (intro analytic_imp_holomorphic analytic_intros)
         (use assms s in ‹auto simp: nonpos_Reals_def›)
  next
    show "{s. Re s > 0} ⊆ -ℤ≤0"
      by (auto elim!: nonpos_Ints_cases)
  next
    have "connected (UNIV - ℤ≤0 :: complex set)"
      by (rule connected_open_diff_countable) auto
```

```
    thus "connected (-ℤ≤0 :: complex set)"
      by (simp add: Compl_eq_Diff_UNIV)
  next
    have "1 ∈ {s. Re s > 0}"
      by auto
    thus "{s. Re s > 0} ≠ {}"
      by blast
  next
    fix s assume "s ∈ {s. Re s > 0}"
    thus "g s = 0"
      using eq1[of s z] z by (simp add: g_def A_def)
  qed (use s in ‹auto simp: open_halfspace_Re_gt›)
  thus ?thesis
    by (simp add: g_def)
qed

show ?thesis
proof (cases "s - 1 ∈ ℕ")
  case True
  then obtain n' where n': "s - 1 = of_nat n'"
    by (elim Nats_cases)
  define n where "n = int (Suc n')"
  have n: "n > 0" "s = of_int n"
    using n' by (simp_all add: n_def algebra_simps)
  show ?thesis
    using eq1[of s z] n unfolding A_def n' by (auto simp: A_def n')
next
  case False
  thus ?thesis
    using eq2[of s z] assms by auto
qed
qed
```

The recurrence for $\Gamma(s, z)$:

```
lemma Gamma_incu_plus1_complex_aux:
  assumes z: "z ∉ ℝ≤0 ∨ s - 1 ∈ ℕ"
  shows "Gamma_incu (s+1) z = s * Gamma_incu s z + z powr' s * exp (-z
:: complex)"
proof -
  have eq1: "Gamma_incu (s+1) z = s * Gamma_incu s z + z powr' s * exp
(-z :: complex)"
    if z: "z ∉ ℝ≤0" for s z
  proof (rule analytic_continuation_open[where f = "λs. Gamma_incu (s+1)
z"])
    show "(λs. Gamma_incu (s+1) z) holomorphic_on UNIV"
      by (intro analytic_imp_holomorphic analytic_intros) (use z in auto)
    show "(λs. s * Gamma_incu s z + z powr' s * exp (-z)) holomorphic_on
UNIV"
      by (intro analytic_imp_holomorphic analytic_intros) (use z in auto)
```

```
  next
    have "1 ∈ (-ℤ≤0 :: complex set)"
      by auto
    thus "(-ℤ≤0 :: complex set) ≠ {}"
      by blast
  next
    fix s :: complex assume s: "s ∈ -ℤ≤0"
    have "Gamma_incu (s + 1) z = Gamma (s+1) - Gamma_incl (s + 1) z"
      using Gamma_incl_plus_incu_complex_aux[of "s+1" z] plus_one_in_nonpos_Ints_imp[of
s] s z
      by (auto simp: algebra_simps)
    also have "... = s * (Gamma s - Gamma_incl s z) + z powr' s * exp
(- z)"
      by (subst Gamma_incl_plus1_complex) (use s in ‹auto simp: Gamma_plus1
ring_distribs›)
    also have "Gamma s - Gamma_incl s z = Gamma_incu s z"
      using Gamma_incl_plus_incu_complex_aux[of s z ]s z by (auto simp:
algebra_simps)
    finally show "Gamma_incu (s + 1) z = s * Gamma_incu s z + z powr' s
* exp (- z)" .
  qed (auto simp: open_halfspace_Re_gt)

  have eq2: "Gamma_incu (s+1) z = s * Gamma_incu s z + z powr' s * exp
(-z :: complex)"
    if "s - 1 ∈ ℕ" for s z
  proof -
    note [analytic_intros del] = analytic_Gamma_incu
    from that obtain n' where n': "s - 1 = of_nat n'"
      by (elim Nats_cases)
    define n where "n = int (Suc n')"
    have n: "n > 0" "s = of_int n"
      using n' by (simp_all add: n_def algebra_simps)
    show ?thesis
    proof (rule analytic_continuation_open[where f = "λz. Gamma_incu
(s+1) z"])
      show "Gamma_incu (s + 1) holomorphic_on UNIV"
        using analytic_Gamma_incu_pos_int[OF analytic_on_ident[of UNIV],
of "n + 1"] n
        by (auto intro: analytic_imp_holomorphic)
      show "(λa. s * Gamma_incu s a + a powr' s * exp (- a)) holomorphic_on
UNIV"
        unfolding n(2) using n(1) by (auto intro!: analytic_imp_holomorphic
analytic_intros)
      show "Gamma_incu (s+1) z = s * Gamma_incu s z + z powr' s * exp
(-z)" if "z ∈ -ℝ≤0" for z
        using that eq1[of z s] by simp
    qed auto
  qed
```

```
  show ?thesis
    using eq1[of z s] eq2[of s z] assms by auto
qed


theorem Gamma_incu_plus1_complex:
  assumes z: "z ∉ ℝ≤0 ∨ s - 1 ∈ ℕ ∨ (z = 0 ∧ Re s > 0)"
  shows "Gamma_incu (s+1) z = s * Gamma_incu s z + z powr' s * exp (-z
:: complex)"
proof (cases "z = 0 ∧ s - 1 ∉ ℕ")
  case True
  define f where "f = (λz. Gamma_incu (s+1) (of_real z) - s * Gamma_incu
s z - z powr' s * exp (-z))"
  have [continuous_intros]:
    "continuous (at_right 0) (λx. Gamma_incu s (of_real x))" if "Re s
> 0" for s
  proof -
    have "continuous (at_right 0) (Gamma_incu s ∘ complex_of_real)"
    proof (rule continuous_within_compose)
      show "continuous (at (complex_of_real 0) within complex_of_real
` {0<..}) (Gamma_incu s)"
        using continuous_Gamma_incu_0_complex unfolding of_real_0
        by (rule continuous_within_subset) (use ‹Re s > 0› in auto)
    qed (auto intro!: continuous_intros)
    thus ?thesis
      by (simp add: o_def)
  qed

  have "continuous (at_right 0) f"
    unfolding f_def using z True by (intro continuous_intros) (auto simp:
Lim_ident_at)
  hence "(f ⟶ f 0) (at_right 0)"
    by (cases "at 0 within {z. Re z > 0} = bot") (auto simp: continuous_def
Lim_ident_at)
  also have "?this ⟷ ((λ_::real. 0) ⟶ f 0) (at_right 0)"
  proof (rule filterlim_cong)
    have "eventually (λz::real. z > 0) (at_right 0)"
      by (auto simp: eventually_at_topological)
    thus "eventually (λz. f z = 0) (at_right 0)" unfolding f_def
      by eventually_elim (auto simp: Gamma_incu_plus1_complex_aux complex_nonpos_Reals_iff)
  qed auto
  finally have "f 0 = 0"
    by (subst (asm) tendsto_const_iff) auto
  thus ?thesis
    using True by (simp add: f_def field_simps)
qed (use Gamma_incu_plus1_complex_aux[of z s] assms in auto)

hide_fact Gamma_incu_plus1_complex_aux
```

```
lemma Gamma_incu_plus1_real:
  assumes z: "z > 0 ∨ (z = 0 ∧ s > 0)"
  shows "Gamma_incu (s+1) z = s * Gamma_incu s z + z powr' s * exp (-z
:: real)"
proof -
  have "complex_of_real (Gamma_incu (s+1) z) = Gamma_incu (of_real s +
1) (of_real z)"
    by (subst Gamma_incu_complex_of_real [symmetric]) auto
  also have "... = complex_of_real (s * Gamma_incu s z + z powr' s * exp
(-z))"
    by (subst Gamma_incu_plus1_complex)
       (use assms in ‹auto simp flip: exp_of_real Gamma_incu_complex_of_real

                                simp: powr'_complex_of_real›)
  finally show ?thesis
    by (simp only: of_real_eq_iff)
qed


theorem Gamma_incl_plus_incu_complex:
  assumes "s ∉ ℤ_{≤0}" "z ∉ ℝ_{≤0} ∨ s - 1 ∈ ℕ ∨ (z = 0 ∧ Re s > 0)"
  shows   "Gamma_incl s z + Gamma_incu s z = Gamma (s :: complex)"
proof (cases "z = 0 ∧ s - 1 ∉ ℕ")
  case True
  define f where "f = (λz. Gamma_incl s z + Gamma_incu s z)"
  from True and assms have s: "Re s > 0"
    by auto

  have [continuous_intros]:
    "continuous (at_right 0) (λx. Gamma_incu s (of_real x))" if "Re s
> 0" for s
  proof -
    have "continuous (at_right 0) (Gamma_incu s ∘ complex_of_real)"
    proof (rule continuous_within_compose)
      show "continuous (at (complex_of_real 0) within complex_of_real
‘ {0<..}) (Gamma_incu s)"
        using continuous_Gamma_incu_0_complex unfolding of_real_0
        by (rule continuous_within_subset) (use ‹Re s > 0› in auto)
    qed (auto intro!: continuous_intros)
    thus ?thesis
      by (simp add: o_def)
  qed

  have "continuous_on {x. x ≥ 0} (λx. Gamma_incl s (complex_of_real x))"
    using assms s by (auto intro!: continuous_intros)
  hence [continuous_intros]: "continuous (at_right 0) (λx. Gamma_incl
s (complex_of_real x))"
    by (rule continuous_on_imp_continuous_within) auto
```

```
  have "continuous (at_right 0) f"
    unfolding f_def using s by (intro continuous_intros) auto
  hence "(f ⟶ f 0) (at_right 0)"
    by (cases "at 0 within {z. Re z > 0} = bot") (auto simp: continuous_def
Lim_ident_at)
  also have "?this ⟷ ((λ_::real. Gamma s) ⟶ f 0) (at_right 0)"
  proof (rule filterlim_cong)
    have "eventually (λz::real. z > 0) (at_right 0)"
      by (auto simp: eventually_at_topological)
    thus "eventually (λz. f z = Gamma s) (at_right 0)" unfolding f_def
      by eventually_elim (subst Gamma_incl_plus_incu_complex_aux, use
assms in auto)
  qed auto
  finally have "f 0 = Gamma s"
    by (subst (asm) tendsto_const_iff) auto
  thus ?thesis
    using True by (simp add: f_def field_simps)
qed (use Gamma_incl_plus_incu_complex_aux[of s z] assms in auto)

hide_fact Gamma_incl_plus_incu_complex_aux

lemma Gamma_incl_plus_incu_real:
  assumes "s ∉ ℤ≤0" "z > 0 ∨ (z = 0 ∧ s > 0)"
  shows   "Gamma_incl s z + Gamma_incu s z = Gamma (s :: real)"
proof (cases "z = 0")
  case True
  thus ?thesis
    using Gamma_incl_plus_incu_complex[of s z] assms
          Gamma_incu_complex_of_real[of s z] Gamma_complex_of_real[of
s]
    by (auto simp: Gamma_incl_def of_real_in_nonpos_Ints_iff)
next
  case False
  thus ?thesis
    using Gamma_incl_plus_incu_complex[of s z] assms
    by (auto simp: Gamma_incl_complex_of_real Gamma_incu_complex_of_real
Gamma_complex_of_real
                   of_real_in_nonpos_Ints_iff simp flip: of_real_add)
qed
```

6.6 Derivative of $\gamma(s, z)$

Via the relationship with $\Gamma(s)$ and $\Gamma(s, z)$, it is now also straightforward to prove the derivative of $\gamma(s, z)$:

```
lemma has_field_derivative_Gamma_incl_complex:
  fixes s z :: complex
  assumes "s ∉ ℤ≤0" "s - 1 ∈ ℕ ∨ z ∉ ℝ≤0"
  shows   "((λx. Gamma_incl s x) has_field_derivative (z powr' (s-1)
* exp (-z))) (at z within A)"
```

```
proof (rule has_field_derivative_at_within)
  have "s ∈ ℤ" if "s - 1 ∈ ℕ"
    using that Ints_1 diff_in_Ints_iff_right minus_in_Ints_iff uminus_in_nonpos_Ints_iff
by blast
  hence "((λz. Gamma s - Gamma_incu s z) has_field_derivative
         (z powr' (s - 1) * exp (-z))) (at z)"
    using assms by (auto intro!: derivative_eq_intros)
  also have "?this ⟷ ((λx. Gamma_incl s x) has_field_derivative (z
powr' (s-1) * exp (-z))) (at z)"
  proof (rule DERIV_cong_ev)
    have "eventually (λx. x ∈ (if s - 1 ∈ ℕ then UNIV else -ℝ≤0)) (nhds
z)"
      by (rule eventually_nhds_in_open) (use assms in auto)
    thus "∀F x in nhds z. Gamma s - Gamma_incu s x = Gamma_incl s x"
    proof eventually_elim
      case (elim x)
      thus ?case
        using Gamma_incl_plus_incu_complex[of s x, symmetric] assms
        by (auto simp: algebra_simps split: if_splits)
    qed
  qed (auto simp: powr'_complex)
  finally show ... .
qed

lemma has_field_derivative_Gamma_incl_complex' [derivative_intros]:
  fixes f :: "_ ⇒ complex"
  assumes "(f has_field_derivative f') (at x within A)" "s ∉ ℤ≤0" "s
- 1 ∈ ℕ ∨ f x ∉ ℝ≤0"
  shows   "((λx. Gamma_incl s (f x)) has_field_derivative
             (f x powr' (s-1) * exp (-f x) * f')) (at x within A)"
  using DERIV_chain[OF has_field_derivative_Gamma_incl_complex[OF assms(2,3)]
assms(1)]
  by (simp add: o_def)

lemma has_field_derivative_Gamma_incl_real [derivative_intros]:
  fixes f :: "_ ⇒ real"
  assumes "(f has_field_derivative f') (at x within A)" and s: "s ∉
ℤ≤0" and fx: "f x > 0"
  shows   "((λx. Gamma_incl s (f x)) has_field_derivative
             (f x powr (s-1) * exp (-f x) * f')) (at x within A)"
proof -
  have *: "(Gamma_incl s has_real_derivative (x powr (s - 1) * exp (-
x))) (at x)"
    if x: "x > 0" for x :: real
  proof -
    have "((λx. Re (Gamma_incl (of_real s) (of_real x))) has_field_derivative

           ((x powr' (s-1) * exp (-x)))) (at x)"
    proof -
```

```
      have x': "complex_of_real x ∉ ℝ≤0" and s': "complex_of_real s
∉ ℤ≤0"
        using x s by (auto simp: of_real_in_nonpos_Ints_iff)
      show ?thesis
        by (rule derivative_eq_intros refl x' s')+
           (use x in ‹auto simp: powr'_Reals_eq exp_of_real simp flip:
of_real_minus›)
    qed
    also have "?this ⟷ (Gamma_incl s has_field_derivative (x powr (s-1)
* exp (-x))) (at x)"
    proof (rule DERIV_cong_ev)
      have "eventually (λt. t ∈ {0<..}) (nhds x)"
        by (rule eventually_nhds_in_open) (use x in auto)
      thus "∀F x in nhds x. Re (Gamma_incl (complex_of_real s) (complex_of_real
x)) =
              Gamma_incl s x"
        by eventually_elim (auto simp: Gamma_incl_complex_of_real)
    qed (use x in ‹auto simp: powr'_real›)
    finally show ?thesis .
  qed
  show ?thesis
    using DERIV_chain[OF *[OF fx] assms(1)] by (simp add: o_def)
qed
```

6.7 Special values

Lastly, we examine the values of $\Gamma(s, z)$ specifically for $z = 0$, s is a positive integer, $s = \frac{1}{2}$, and $z \to \infty$.

```
lemma Gamma_incl_0_left: "Gamma_incl 0 z = 0"
  by (simp add: Gamma_incl_def)

lemma Gamma_incl_0_right [simp]: "s ≠ 0 ⟹ Gamma_incl s 0 = 0"
  by (auto simp: Gamma_incl_def)

lemma Gamma_incu_0_right_complex [simp]: "Re s > 0 ⟹ Gamma_incu s 0
= Gamma (s::complex)"
  by (cases "s ∈ ℤ≤0"; cases "s = 0")
     (use Gamma_incl_plus_incu_complex[of s 0] in ‹auto elim!: nonpos_Ints_cases›)

lemma Gamma_incu_0_right_real [simp]: "s > 0 ⟹ Gamma_incu s 0 = Gamma
(s::real)"
  by (cases "s ∈ ℤ≤0"; cases "s = 0")
     (use Gamma_incl_plus_incu_real[of s 0] in ‹auto elim!: nonpos_Ints_cases›)
```

The following theorem now summarises the behaviour of $\Gamma(s, z)$ at $z = 0$ and $\mathrm{Re}(s) > 0$: when approaching from direction $\mathrm{Re}(z) > 0$, $\Gamma(s, z) \to \Gamma(s')$ as $s \to s'$ and $z \to 0$.

```
theorem tendsto_Gamma_incu_0_right_complex:
```

```
  assumes "(f ⟶ s) F" "(g ⟶ 0) F"
  assumes "Re s > 0" "eventually (λx. Re (g x) > 0) F"
  shows   "((λx. Gamma_incu (f x) (g x)) ⟶ Gamma s) F"
proof -
  have 1: "((λx. (f x, g x)) ⟶ (s, 0)) F"
    by (auto intro!: tendsto_intros assms(1,2))
  have "eventually (λx. f x ∈ {s. Re s > 0}) F"
    by (intro eventually_compose_filterlim[OF _ assms(1)] eventually_nhds_in_open)
       (use assms(3) in ‹auto simp: open_halfspace_Re_gt›)
  hence 2: "∀F x in F. (f x, g x) ∈ {s. 0 < Re s} × {z. 0 < Re z}"
    using assms(4) by eventually_elim auto
  show ?thesis
    using continuous_within_tendsto_compose[OF continuous_Gamma_incu_0_strong_complex
2 1] assms(3)
    by simp
qed



lemma Gamma_incl_1_left: "Gamma_incl 1 z = 1 - exp (-z)"
  by (auto simp: Gamma_incl_def Gamma_rincl_1_left)

lemma Gamma_incu_1_left_complex: "Gamma_incu 1 (z::complex) = exp (-z)"
  using Gamma_incl_plus_incu_complex[of 1 z] by (simp add: Gamma_incl_1_left)

lemma Gamma_incu_1_left_real: "z ≥ 0 ⟹ Gamma_incu 1 (z::real) = exp
(-z)"
  using Gamma_incl_plus_incu_real[of 1 z] by (cases "z = 0") (auto simp:
Gamma_incl_1_left)


theorem Gamma_incl_of_nat_left_complex:
  fixes z :: complex
  shows "Gamma_incl (of_nat (Suc n)) z = fact n * (1 - exp (-z) * (∑k≤n.
z ^ k / fact k))"
proof -
  have "Gamma_incl (of_nat (Suc n)) z =
          Gamma (1 + complex_of_nat n) * Gamma_rincl (of_nat (Suc n))
z"
    by (simp add: Gamma_incl_def)
  also have "Gamma (1 + complex_of_nat n) = fact n"
    by (rule Gamma_fact)
  also have "Gamma_rincl (of_nat (Suc n)) z = 1 - exp (- z) * (∑k≤n.
z ^ k / fact k)"
    by (rule Gamma_rincl_of_nat_left_complex)
  finally show ?thesis .
qed

lemma Gamma_incl_of_nat_left_real:
```

```
  fixes z :: real
  shows "Gamma_incl (of_nat (Suc n)) z = fact n * (1 - exp (-z) * (∑k≤n.
z ^ k / fact k))"
proof -
  have "Gamma_incl (of_nat (Suc n)) z = Gamma (1 + real n) * Gamma_rincl
(of_nat (Suc n)) z"
    by (simp add: Gamma_incl_def)
  also have "Gamma (1 + real n) = fact n"
    by (rule Gamma_fact)
  also have "Gamma_rincl (real (Suc n)) z = 1 - exp (- z) * (∑k≤n. z
^ k / fact k)"
    by (rule Gamma_rincl_of_nat_left_real)
  finally show ?thesis .
qed

theorem Gamma_incu_of_nat_left_complex:
  fixes z :: complex
  shows "Gamma_incu (of_nat (Suc n)) z = fact n * exp (-z) * (∑k≤n.
z ^ k / fact k)"
proof -
  have "complex_of_nat (Suc n) ∉ ℤ≤0"
    unfolding of_nat_in_nonpos_Ints_iff by simp
  hence "Gamma_incu (of_nat (Suc n)) z = Gamma (of_nat (Suc n)) - Gamma_incl
(of_nat (Suc n)) z"
    by (subst Gamma_incl_plus_incu_complex [of _ z, symmetric]) auto
  also have "Gamma (of_nat (Suc n)) = fact n"
    by (simp add: Gamma_fact)
  finally show ?thesis
    by (subst (asm) Gamma_incl_of_nat_left_complex) (auto simp: algebra_simps)
qed

lemma Gamma_incu_of_nat_left_real:
  fixes z :: real
  shows "Gamma_incu (of_nat (Suc n)) z = fact n * exp (-z) * (∑k≤n.
z ^ k / fact k)"
proof -
  have "complex_of_real (Gamma_incu (of_nat (Suc n)) z) = Gamma_incu (of_nat
(Suc n)) (of_real z)"
    by (simp flip: Gamma_incu_complex_of_real)
  also have "... = complex_of_real (fact n * exp (-z) * (∑k≤n. z ^ k
/ fact k))"
    by (subst Gamma_incu_of_nat_left_complex) (simp_all flip: exp_of_real)
  finally show ?thesis
    by (simp only: of_real_eq_iff)
qed
```

Via the hypergeometric representation, it is easy to see that for $\gamma(\frac{1}{2}, z)$ and $\Gamma(\frac{1}{2}, z)$ have representations in terms of $\mathrm{erf}(\sqrt{z})$ and $texterfc(\sqrt{z})$, respectively:

```
theorem Gamma_incl_one_half_left_complex:
  assumes "z = 0 ∨ z ∉ ℝ≤0"
  shows   "Gamma_incl (1/2) (z :: complex) = sqrt pi * erf (csqrt z)"
proof (cases "z = 0")
  case False
  thus ?thesis
    unfolding Gamma_incl_def
    by (subst Gamma_rincl_one_half_left_complex) (use assms in ‹auto simp:
Gamma_one_half_complex›)
qed auto

lemma Gamma_incl_one_half_left_real:
  assumes "z ≥ 0"
  shows   "Gamma_incl (1/2) (z :: real) = sqrt pi * erf (sqrt z)"
  unfolding Gamma_incl_def
  by (subst Gamma_rincl_one_half_left_real) (use assms in ‹auto simp:
Gamma_one_half_real›)

lemma Gamma_incu_one_half_left_complex:
  assumes "z = 0 ∨ z ∉ ℝ≤0"
  shows   "Gamma_incu (1/2) (z :: complex) = sqrt pi * erfc (csqrt z)"
proof -
  have "Gamma_incl (1 / 2) z + Gamma_incu (1 / 2) z = Gamma (1 / 2)"
    by (rule Gamma_incl_plus_incu_complex) (use assms in auto)
  thus ?thesis using assms
    by (auto simp: Gamma_incl_one_half_left_complex Gamma_one_half_complex
erfc_def field_simps)
qed

lemma Gamma_incu_one_half_left_real:
  assumes "z ≥ 0"
  shows   "Gamma_incu (1/2) (z :: real) = sqrt pi * erfc (sqrt z)"
proof -
  have "Gamma_incl (1 / 2) z + Gamma_incu (1 / 2) z = Gamma (1 / 2)"
    by (rule Gamma_incl_plus_incu_real) (use assms in auto)
  thus ?thesis using assms
    by (auto simp: Gamma_incl_one_half_left_real Gamma_one_half_real erfc_def
field_simps)
qed
```

$\Gamma(s, x)$ vanishes as $x \to \infty$.

```
lemma Gamma_incu_at_top_complex: "((λz. Gamma_incu s (complex_of_real
z)) ⟶ 0) at_top"
proof -
  define h where "h = (λz. exp ((s-1) * of_real (ln z) - complex_of_real
z))"
  have smallo: "(λt. exp ((Re s - 1) * ln t - t)) ∈ o(λt. exp (-t/2))"
    by (real_asymp simp: field_simps)
  have "eventually (λt. exp ((Re s - 1) * ln t - t) ≤ exp (-t/2)) at_top"
```

```
    using landau_o.smallD[OF smallo, of 1] by simp
  then obtain x0 where x0: "⋀t. t ≥ x0 ⟹ exp ((Re s - 1) * ln t -
t) ≤ exp (-t/2)"
    unfolding eventually_at_top_linorder by blast

  have "eventually (λx. norm (Gamma_incu s (of_real x)) ≤ 2 * exp (-x/2))
at_top"
    using eventually_gt_at_top[of x0] eventually_gt_at_top[of 1]
  proof eventually_elim
    case x: (elim x)
    have h: "(h has_integral (Gamma_incu s (of_real x))) {x..}"
      unfolding h_def by (rule has_integral_Gamma_incu_complex_of_real')
(use x in auto)
    have bound: "((λt. exp (-t/2)) has_integral (2 * exp (-x/2))) {x..}"
      using has_integral_exp_minus_to_infinity[of "1/2" x] x by (simp
add: mult_ac)
    have "norm (integral {x..} h) ≤ integral {x..} (λt. exp (-t/2))"
    proof (rule integral_norm_bound_integral)
      fix t assume t: "t ∈ {x..}"
      have "norm (h t) = exp ((Re s - 1) * ln t - t)"
        by (simp add: h_def)
      also have "... ≤ exp (-t/2)"
        by (rule x0) (use x t in auto)
      finally show "norm (h t) ≤ exp (-t/2)" .
    qed (use bound h in ‹simp_all add: has_integral_iff›)
    thus "norm (Gamma_incu s (of_real x)) ≤ 2 * exp (-x/2)"
      using bound h by (auto simp: has_integral_iff)
  qed
  moreover have "((λx. 2 * exp (-x/2::real)) ⟶ 0) at_top"
    by real_asymp
  ultimately show ?thesis
    by (rule Lim_null_comparison)
qed

lemma Gamma_incu_at_top_real: "((λz. Gamma_incu s (z::real)) ⟶ 0)
at_top"
proof -
  have "((λz. Re (Gamma_incu s (complex_of_real z))) ⟶ Re 0) at_top"
    by (rule tendsto_Re Gamma_incu_at_top_complex)+
  also have "(λz. Re (Gamma_incu s (complex_of_real z))) = Gamma_incu
s"
    by (simp add: Gamma_incu_complex_of_real)
  finally show ?thesis
    by simp
qed
```

Consequently, $\gamma(s, z) \to \Gamma(s)$ as $z \to \infty$:

```
lemma Gamma_incl_at_top_complex:
  assumes "s ∉ ℤ≤0"
```

```
  shows   "((λz. Gamma_incl s (complex_of_real z)) ⟶ Gamma s) at_top"
proof -
  have "((λz. Gamma s - Gamma_incu s (complex_of_real z)) ⟶ Gamma
s - 0) at_top"
    by (intro tendsto_intros Gamma_incu_at_top_complex)
  also have "eventually (λz. Gamma s - Gamma_incu s (of_real z) = Gamma_incl
s (of_real z)) at_top"
    using eventually_gt_at_top[of 0]
  proof eventually_elim
    case (elim z)
    thus ?case
      by (subst Gamma_incl_plus_incu_complex [of s z, symmetric]) (use
assms in auto)
  qed
  finally show ?thesis
    by simp
qed

lemma Gamma_incl_at_top_real:
  assumes "s ∉ ℤ≤0"
  shows   "((λz. Gamma_incl s (z::real)) ⟶ Gamma s) at_top"
proof -
  have "((λz. Re (Gamma_incl (of_real s) (complex_of_real z))) ⟶ Re
(Gamma (of_real s))) at_top"
    by (rule tendsto_Re Gamma_incl_at_top_complex)+
       (use assms in ‹auto simp: of_real_in_nonpos_Ints_iff›)
  also have "?this ⟷ ?thesis"
  proof (intro filterlim_cong)
    show "∀F x in at_top. Re (Gamma_incl (complex_of_real s) (complex_of_real
x)) = Gamma_incl s x"
      using eventually_gt_at_top[of 0] by eventually_elim (auto simp:
Gamma_incl_complex_of_real)
  qed (auto simp: Gamma_complex_of_real)
  finally show ?thesis .
qed

end
```

References

[1] NIST Digital Library of Mathematical Functions. https://dlmf.nist.gov/, Release 1.2.4 of 2025-03-15. F. W. J. Olver, A. B. Olde Daalhuis, D. W. Lozier, B. I. Schneider, R. F. Boisvert, C. W. Clark, B. R. Miller, B. V. Saunders, H. S. Cohl, and M. A. McClain, eds.