The Detour Calculus

Manuel Eberl

September 9, 2026

Abstract

This entry provides the *detour calculus*, a relational framework to construct deformed versions of integration contours by adding small indentations around problematic points (typically poles). It consists of:

- the definition of a relation that relates the original contour γ to a family of deformed contours $(\gamma_\epsilon)_{\epsilon>0}$
- theorems that allow transferring geometric properties from γ to γ_ϵ for sufficiently small ϵ
- a library of "basic avoidance patterns" to add indentations to lines, circular arcs, or the corners between them, and prove that they are in relation with the original contour
- a calculus of inference rules which, in particular, allows dealing with composite contours by proving the relation for its individual parts separately

The motivating example for this calculus was the proof of the valence formula for modular forms (not included here), which requires such a complicated set of indentations that I deemed it beyond the reach of formalisation without a systematic approach such as the present one.

Contents

1 Auxiliary material **3**

2 Neighbourhoods of a path **3**
2.1 Nearby paths . . . 5
2.2 Piecewise smooth paths in the neighbourhood . . . 8
2.3 Lipschitz-continuity and paths . . . 11

3 Prerequisites for the Detour Calculus **14**
3.1 Miscellaneous . . . 15
3.2 Lipschitz continuity . . . 19
3.3 Homotopy . . . 22
3.4 Winding numbers . . . 23
3.5 Continuous transformations that preserve the winding number in some way . . . 33
3.6 Paths . . . 35
3.7 Topology . . . 41

4 The Detour Calculus **42**
4.1 Local deformations of a path . . . 42
4.2 The left/right detour relation . . . 50
4.3 Inference rules . . . 64
4.4 Basic avoidance patterns . . . 71
4.4.1 Generic helper lemmas . . . 71
4.4.2 Straight line . . . 80
4.4.3 Circular arc . . . 85
4.4.4 Line–line corner . . . 103
4.4.5 Line–arc corner . . . 110
4.5 Consequences of the detour relation . . . 117

1 Auxiliary material

2 Neighbourhoods of a path

```
theory Path_Nhds
  imports "HOL-Complex_Analysis.Complex_Analysis"
begin

lemma le_filterD_frequently [trans]: "F ≤ G ⟹ frequently P F ⟹
frequently P G"
  unfolding le_filter_def frequently_def by blast

lemma le_filterD_frequently' [trans]: "frequently P F ⟹ F ≤ G ⟹
frequently P G"
  unfolding le_filter_def frequently_def by blast

lemma frequently_filtermap: "frequently P (filtermap f F) ⟷ frequently
(λx. P (f x)) F"
  by (simp add: frequently_def eventually_filtermap)


lemma eventually_uniformly_on_iff:
  "eventually P (uniformly_on A f) ⟷ (∃e>0. ∀g. (∀y∈A. dist (g y)
(f y) < e) ⟶ P g)"
  (is "?lhs = ?rhs")
proof -
  have "eventually P (uniformly_on A f) ⟷
          (∃X⊆{0<..}. finite X ∧ eventually P (INF b∈X. principal {fa.
∀x∈A. dist (fa x) (f x) < b}))"
    unfolding uniformly_on_def by (rule eventually_INF)
  also have "... ⟷ (∃X⊆{(0::real)<..}. finite X ∧ (∃Q. (∀e∈X. ∀g.
(∀y∈A. dist (g y) (f y) < e) ⟶ Q e g) ∧
               (∀y. (∀x∈X. Q x y) ⟶ P y)))"
  proof (intro ex_cong1 conj_cong refl, goal_cases)
    case (1 X)
    have "eventually P (INF b∈X. principal {fa. ∀x∈A. dist (fa x) (f
x) < b}) ⟷
          (∃Q. (∀e∈X. ∀g. (∀y∈A. dist (g y) (f y) < e) ⟶ Q e g) ∧
               (∀y. (∀x∈X. Q x y) ⟶ P y))"
      using 1 by (subst eventually_INF_finite) (simp_all add: eventually_principal)
    thus ?case .
  qed
  finally have eq: "eventually P (uniformly_on A f) =
                      (∃X⊆{0<..}. finite X ∧
                         (∃Q. (∀e∈X. ∀g. (∀y∈A. dist (g y) (f y) < e)
⟶ Q e g) ∧
                         (∀y. (∀x∈X. Q x y) ⟶ P y)))" .

  show ?thesis
```

```
  proof
    assume ?rhs
    then obtain e where e: "e > 0" "⋀g. (∀y∈A. dist (g y) (f y) < e)
⟹ P g"
      by blast
    let ?Q = "λe g. ∀y∈A. dist (g y) (f y) < e"
    show "eventually P (uniformly_on A f)"
      by (subst eq, rule exI[of _ "{e}"], safe intro!: e exI[of _ ?Q])
(use e in auto)
  next
    assume ?lhs
    then obtain X Q where XQ: "X ⊆ {0<..}" "finite X"
      "⋀e g. e ∈ X ⟹ (∀y∈A. dist (g y) (f y) < e) ⟹ Q e g"
      "⋀g. (⋀x. x ∈ X ⟹ Q x g) ⟹ P g"
      by (subst (asm) eq) metis

    show ?rhs
    proof (cases "X = {}")
      case True
      thus ?thesis using XQ
        by (auto intro!: exI[of _ 1])
    next
      case False
      define e where "e = Min X"
      have e: "e > 0" "e ∈ X"
        using False XQ(1,2) by (auto simp: e_def)

      show ?rhs
      proof (rule exI[of _ e], safe)
        show "e > 0"
          by fact
      next
        fix g assume g: "∀y∈A. dist (g y) (f y) < e"
        show "P g"
        proof (intro XQ ballI)
          fix e' y assume e': "e' ∈ X" and y: "y ∈ A"
          have "dist (g y) (f y) < e"
            using g y by blast
          also have "e ≤ e'"
            using e' False ‹finite X› by (simp add: e_def)
          finally show "dist (g y) (f y) < e'" .
        qed auto
      qed
    qed
  qed
qed

lemma eventually_uniformly_onI [intro?]:
  "e > 0 ⟹ (⋀g. (⋀y. y ∈ A ⟹ dist (g y) (f y) < e) ⟹ P g) ⟹
```

```
    eventually P (uniformly_on A f)"
  unfolding eventually_uniformly_on_iff by blast

abbreviation same_ends :: "(real ⇒ 'a :: topological_space) ⇒ (real
⇒ 'a) ⇒ bool"
  where "same_ends p q ≡ pathstart p = pathstart q ∧ pathfinish p =
pathfinish q"
```

2.1 Nearby paths

```
definition path_nhds :: "(real ⇒ 'a :: real_normed_vector) ⇒ (real ⇒
'a) filter" where
  "path_nhds γ = inf (uniformly_on {0..1} γ) (principal {p. path p ∧
same_ends p γ})"

lemma eventually_path_nhds_iff:
  "eventually P (path_nhds γ) ⟷
    (∃e>0. ∀p. path p ⟶ same_ends p γ ⟶ (∀y∈{0..1}. dist (p y)
(γ y) < e) ⟶ P p)"
  unfolding path_nhds_def eventually_uniformly_on_iff eventually_inf_principal
  by blast

lemma frequently_path_nhds_iff:
  "frequently P (path_nhds γ) ⟷
    (∀e>0. ∃p. path p ∧ same_ends p γ ∧ (∀y∈{0..1}. dist (p y) (γ
y) < e) ∧ P p)"
  unfolding frequently_def eventually_path_nhds_iff by blast

lemma eventually_path_nhdsI [intro?]:
  "e > 0 ⟹ (⋀p. path p ⟹ same_ends p γ ⟹ (⋀y. y ∈ {0..1} ⟹
dist (p y) (γ y) < e) ⟹ P p)
    ⟹ eventually P (path_nhds γ)"
  unfolding eventually_path_nhds_iff by blast

lemma eventually_path_path_nhds: "eventually (λp. path p) (path_nhds
γ)"
  by (rule eventually_path_nhdsI[OF zero_less_one])

lemma path_nhds_neq_bot [simp]: "path γ ⟹ path_nhds γ ≠ bot"
  by (auto simp: trivial_limit_def eventually_path_nhds_iff intro!: exI[of
_ γ])

lemma eventually_dist_less_path_nhds:
  assumes "e > 0"
  shows   "eventually (λp. ∀t∈{0..1}. dist (p t) (γ t) < e) (path_nhds
γ)"
  using assms by (intro eventually_path_nhdsI[of e]) auto

lemma eventually_winding_number_eq_path_nhds:
```

```
  assumes "path γ" "z ∉ path_image γ"
  shows   "eventually (λp. winding_number p z = winding_number γ z) (path_nhds
γ)"
proof -
  define e where "e = setdist {z} (path_image γ)"
  show ?thesis
  proof (rule eventually_path_nhdsI; safe?)
    show "e > 0"
      using assms unfolding e_def
      by (subst setdist_gt_0_compact_closed) (auto intro!: closed_path_image)
  next
    fix p assume p: "path p" "pathstart p = pathstart γ" "pathfinish
p = pathfinish γ"
                    "⋀y. y ∈ {0..1} ⟹ dist (p y) (γ y) < e"
    show "winding_number p z = winding_number γ z"
    proof (rule winding_number_nearby_paths_eq)
      fix t :: real assume t: "t ∈ {0..1}"
      have "norm (p t - γ t) < e"
        using p(4)[OF t] by (simp add: dist_norm)
      also have "... ≤ dist z (γ t)"
        unfolding e_def by (rule setdist_le_dist) (use t in ‹auto simp:
path_image_def›)
      finally show "cmod (p t - γ t) < cmod (γ t - z)"
        by (simp add: dist_norm norm_minus_commute)
    qed (use p assms in auto)
  qed
qed

lemma eventually_path_image_subset_path_nhds:
  assumes "path γ" "open A" "path_image γ ⊆ A"
  shows   "eventually (λp. path_image p ⊆ A) (path_nhds γ)"
proof -
  have "compact (path_image γ)"
    by (intro compact_path_image assms)
  then obtain e where e: "e > 0" "(⋃x∈path_image γ. cball x e) ⊆ A"
    using compact_subset_open_imp_cball_epsilon_subset[of "path_image
γ" A] assms ‹open A›
    by blast
  have "eventually (λp. ∀t∈{0..1}. dist (p t) (γ t) < e) (path_nhds γ)"
    by (intro eventually_dist_less_path_nhds assms e)
  thus ?thesis
  proof eventually_elim
    case (elim p)
    show "path_image p ⊆ A"
      unfolding path_image_def
    proof safe
      fix t :: real assume t: "t ∈ {0..1}"
      have "dist (p t) (γ t) < e"
        using elim t by blast
```

```
      hence "p t ∈ (⋃x∈path_image γ. cball x e)"
        unfolding path_image_def using t by (auto simp: dist_commute
intro!: less_imp_le)
      also have "... ⊆ A"
        using e by blast
      finally show "p t ∈ A" .
    qed
  qed
qed

lemma eventually_path_nhds_avoid:
  assumes "path γ" "closed A" "A ∩ path_image γ = {}"
  shows   "eventually (λp. path_image p ∩ A = {}) (path_nhds γ)"
proof -
  have "eventually (λp. path_image p ⊆ -A) (path_nhds γ)"
    by (rule eventually_path_image_subset_path_nhds) (use assms in auto)
  thus ?thesis
    by eventually_elim blast
qed
```

If we have a path `p` and transform it with a function that is continuous in some open neighbourhood of `p`, then all the paths that are close to `p` are also transformed to paths close to the image of `p`.

```
lemma continuous_path_image:
  fixes p :: "real ⇒ 'a :: euclidean_space"
  assumes "path p" "continuous_on A f" "open A" "path_image p ⊆ A"
  shows "filterlim (λp. f ∘ p) (path_nhds (f ∘ p)) (path_nhds p)"
  unfolding filterlim_def le_filter_def eventually_filtermap
proof safe
  fix P assume P: "eventually P (path_nhds (f ∘ p))"
  then obtain ε :: real where ε: "ε > 0"
    "⋀p'. path p' ⟹ same_ends p' (f ∘ p) ⟹
           (⋀t. t ∈ {0..1} ⟹ dist (p' t) ((f ∘ p) t) < ε) ⟹ P p'"
    unfolding eventually_path_nhds_iff by blast

  obtain r where r: "r > 0" "(⋃x∈path_image p. cball x r) ⊆ A"
    using compact_subset_open_imp_cball_epsilon_subset[of "path_image
p" A] assms
    by auto
  define B where "B = (⋃x∈path_image p. cball x r)"
  have "B ⊆ A"
    using r unfolding B_def by blast
  have "compact B"
    unfolding B_def by (intro compact_minkowski_sum_cball compact_path_image
assms)
  have "uniformly_continuous_on B f"
    by (intro compact_uniformly_continuous continuous_on_subset[OF assms(2)])
fact+
  then obtain δ' where δ': "δ' > 0" "⋀x y. x ∈ B ⟹ y ∈ B ⟹ dist
```

```
x y < δ' ⟹ dist (f x) (f y) < ε"
    using ‹ε > 0› unfolding uniformly_continuous_on_def by fast
  define δ where "δ = min r δ'"
  have δ: "δ > 0" "δ ≤ r" "δ ≤ δ'"
    using ‹δ' > 0› ‹r > 0› unfolding δ_def by auto

  show "∀F x in path_nhds p. P (f ∘ x)"
  proof
    show "δ > 0"
      by fact
  next
    fix p'
    assume p': "path p'" "same_ends p' p"
      "⋀t. t ∈ {0..1} ⟹ dist (p' t) (p t) < δ"
    have "path_image p ⊆ B"
      unfolding B_def using ‹r > 0› by fastforce
    have "path_image p' ⊆ B"
      using p'(3) δ unfolding B_def
      by (force simp: path_image_def dist_commute)
    show "P (f ∘ p')"
    proof (rule ε(2))
      show "path (f ∘ p')"
        using assms ‹path_image p' ⊆ B› ‹B ⊆ A›
        by (intro path_continuous_image p' continuous_on_subset[OF assms(2)])
auto
      show "same_ends (f ∘ p') (f ∘ p)"
        using p' by (simp add: pathstart_compose pathfinish_compose)
      show "dist ((f ∘ p') t) ((f ∘ p) t) < ε" if "t ∈ {0..1}" for t
      proof -
        have "dist ((f ∘ p') t) ((f ∘ p) t) = dist (f (p' t)) (f (p t))"
          by simp
        moreover have "dist (p' t) (p t) < δ'"
          using δ_def min_less_iff_conj p'(3) that by blast
        ultimately show "dist ((f ∘ p') t) ((f ∘ p) t) < ε"
          unfolding o_def using p' ‹path_image p ⊆ B› ‹path_image p'
⊆ B› δ that
          by (intro δ') (auto simp: path_image_def)
      qed
    qed
  qed
qed
```

2.2 Piecewise smooth paths in the neighbourhood

```
definition valid_path_nhds :: "(real ⇒ 'a :: real_normed_vector) ⇒ (real
⇒ 'a) filter" where
  "valid_path_nhds γ = inf (uniformly_on {0..1} γ) (principal {p. valid_path
p ∧ same_ends p γ})"
```

```
lemma eventually_valid_path_nhds_iff:
  "eventually P (valid_path_nhds γ) ⟷
     (∃e>0. ∀p. valid_path p ⟶ same_ends p γ ⟶ (∀y∈{0..1}. dist
(p y) (γ y) < e) ⟶ P p)"
  unfolding valid_path_nhds_def eventually_uniformly_on_iff eventually_inf_principal
  by blast

lemma frequently_valid_path_nhds_iff:
  "frequently P (valid_path_nhds γ) ⟷
     (∀e>0. ∃p. valid_path p ∧ same_ends p γ ∧ (∀y∈{0..1}. dist (p
y) (γ y) < e) ∧ P p)"
  unfolding frequently_def eventually_valid_path_nhds_iff by blast

lemma eventually_valid_path_nhdsI [intro?]:
  "e > 0 ⟹ (⋀p. valid_path p ⟹ same_ends p γ ⟹ (⋀y. y ∈ {0..1}
⟹ dist (p y) (γ y) < e) ⟹ P p)
     ⟹ eventually P (valid_path_nhds γ)"
  unfolding eventually_valid_path_nhds_iff by blast

lemma eventually_valid_path_valid_path_nhds: "eventually (λp. valid_path
p) (valid_path_nhds γ)"
  by (rule eventually_valid_path_nhdsI [OF zero_less_one])

lemma path_nhds_le_valid_path_nhds: "valid_path_nhds γ ≤ path_nhds γ"
  by (rule filter_leI)
     (auto simp: eventually_path_nhds_iff eventually_valid_path_nhds_iff
valid_path_imp_path)

lemma valid_path_nhds_neq_bot [simp]: "valid_path γ ⟹ valid_path_nhds
γ ≠ bot"
  by (auto simp: trivial_limit_def eventually_valid_path_nhds_iff intro!:
exI[of _ γ])

lemma valid_path_nhds_eq_bot' [simp]:
  assumes "path (γ :: real ⇒ 'a :: euclidean_space)"
  shows   "valid_path_nhds γ ≠ bot"
proof -
  {
    fix e :: real assume e: "e > 0"
    obtain p where p: "polynomial_function p" "pathstart p = pathstart
γ"
       "pathfinish p = pathfinish γ" "⋀t. t ∈ {0..1} ⟹ norm (p t -
γ t) < e"
       using path_approx_polynomial_function[OF assms(1) e] by blast
    hence "∃p. valid_path p ∧ same_ends p γ ∧ (∀t∈{0..1}. dist (p t)
(γ t) < e)"
       using valid_path_polynomial_function by (intro exI[of _ p]) (auto
simp: dist_norm)
  }
```

```
  thus ?thesis
    unfolding trivial_limit_def eventually_valid_path_nhds_iff by blast
qed

lemma eventually_dist_less_valid_path_nhds:
  assumes "e > 0"
  shows   "eventually (λp. ∀t∈{0..1}. dist (p t) (γ t) < e) (valid_path_nhds
γ)"
  using assms by (intro eventually_valid_path_nhdsI[of e]) auto

lemma eventually_same_ends_path_nhds:
        "eventually (λp. same_ends p γ) (path_nhds γ)"
  and eventually_same_ends_valid_path_nhds:
        "eventually (λp. same_ends p γ) (valid_path_nhds γ)"
  by (rule eventually_path_nhdsI[of 1] eventually_valid_path_nhdsI[of
1]; simp)+

lemma eventually_valid_path_nhds_avoid:
  assumes "path γ" "closed A" "A ∩ path_image γ = {}"
  shows   "eventually (λp. path_image p ∩ A = {}) (valid_path_nhds γ)"
  using eventually_path_nhds_avoid[OF assms]
        le_filter_def path_nhds_le_valid_path_nhds by blast

lemma winding_number_unique':
  assumes "frequently (λp. winding_number p z = n) (valid_path_nhds γ)"
  assumes "path γ" "z ∉ path_image γ"
  shows   "winding_number γ z = n"
proof (rule winding_number_unique)
  fix e :: real
  assume e: "e > 0"
  have "frequently (λp. path_image p ∩ {z} = {} ∧ winding_number p z
= n) (valid_path_nhds γ)"
    using assms by (intro frequently_eventually_conj eventually_valid_path_nhds_avoid)
auto
  then obtain p
    where p: "valid_path p" "z ∉ path_image p" "same_ends p γ" "winding_number
p z = n"
             "(∀y∈{0..1}. dist (p y) (γ y) < e)"
    using e unfolding frequently_valid_path_nhds_iff by fast
  have "contour_integral p (λw. 1 / (w - z)) = 2 * complex_of_real pi
* i * winding_number p z"
    using p by (subst winding_number_valid_path) auto
  with p show "∃p. winding_number_prop γ z e p n"
    by (intro exI[of _ p]) (auto simp: winding_number_prop_def dist_norm
norm_minus_commute)
qed (use assms in auto)

lemma eventually_path_image_subset_valid_path_nhds:
  assumes "path γ" "open A" "path_image γ ⊆ A"
```

```
  shows   "eventually (λp. path_image p ⊆ A) (valid_path_nhds γ)"
  using eventually_path_image_subset_path_nhds[OF assms]
        le_filter_def path_nhds_le_valid_path_nhds by blast
```

A set is defined to be path-connected if any two points in it are connected by a continuous path. The following shows that for open sets, one can also take the paths to be piecewise C1.

```
lemma path_connected_open_has_valid_path:
  fixes A :: "'a :: euclidean_space set"
  assumes "path_connected A" "open A" "x ∈ A" "y ∈ A"
  obtains p where "valid_path p" "path_image p ⊆ A" "pathstart p = x"
"pathfinish p = y"
proof -
  from assms obtain p where p: "path p" "path_image p ⊆ A" "pathstart
p = x" "pathfinish p = y"
    by (force simp: path_connected_def)
  have "eventually (λp'. valid_path p' ∧ path_image p' ⊆ A ∧ same_ends
p p') (valid_path_nhds p)"
    using eventually_valid_path_valid_path_nhds eventually_same_ends_valid_path_nhds
          eventually_path_image_subset_valid_path_nhds[OF p(1) assms(2)
p(2)]
    by eventually_elim auto
  moreover have "valid_path_nhds p ≠ bot"
    using p by auto
  ultimately show ?thesis using that
    using eventually_happens' p(3) p(4) by force
qed
```

A path `p` always has arbitrarily close smooth paths in its vicinity. (i.e. it can be approximated by smooth paths to arbitrary precision)

```
lemma frequently_valid_path:
  assumes "path (p :: real ⇒ 'a :: euclidean_space)"
  shows   "frequently (λp'. valid_path p') (path_nhds p)"
proof -
  have "valid_path_nhds p ≠ bot"
    using assms by simp
  thus ?thesis
    by (meson eventually_frequently eventually_valid_path_valid_path_nhds

              frequently_def le_filter_def path_nhds_le_valid_path_nhds)
qed
```

2.3 Lipschitz-continuity and paths

```
lemma path_nhds_compose:
  assumes "uniformly_continuous_on A f" "path γ" "path_image γ ⊆ A"
"open A"
  shows   "filterlim ((∘) f) (path_nhds (f ∘ γ)) (path_nhds γ)"
proof -
```

```
  have 1: "uniform_limit {0..1} (λg. g) γ (path_nhds γ)"
    unfolding path_nhds_def using filterlim_ident filterlim_inf by blast
  have 2: "∀F h in path_nhds γ. path_image h ⊆ A"
    by (rule eventually_path_image_subset_path_nhds) (use assms in auto)
  have 3: "uniform_limit {0..1} ((∘) f) (f ∘ γ) (path_nhds γ)"
    using uniform_limit_compose[OF 1 assms(1)] 2 assms by (simp add: o_def[abs_def]
path_image_def)
  have 4: "∀F x in path_nhds γ. path (f ∘ x) ∧ same_ends (f ∘ x) (f
∘ γ)"
    using eventually_path_path_nhds[of γ] eventually_same_ends_path_nhds[of
γ] 2
  proof eventually_elim
    case (elim h)
    have "continuous_on (path_image h) f"
      using uniformly_continuous_imp_continuous continuous_on_subset elim
assms by blast
    hence "path (f ∘ h)"
      using elim by (auto intro!: path_continuous_image)
    moreover have "same_ends (f ∘ h) (f ∘ γ)"
      using elim by (auto simp: pathstart_compose pathfinish_compose)
    ultimately show ?case
      by blast
  qed

  from 3 and 4 show ?thesis
    unfolding path_nhds_def filterlim_inf filterlim_principal by simp_all
qed

lemma valid_path_nhds_compose:
  assumes "f analytic_on A" "uniformly_continuous_on A f" "path γ" "path_image
γ ⊆ A" "open A"
  shows   "filterlim ((∘) f) (valid_path_nhds (f ∘ γ)) (valid_path_nhds
γ)"
proof -
  have 1: "uniform_limit {0..1} (λg. g) γ (valid_path_nhds γ)"
    unfolding valid_path_nhds_def using filterlim_ident filterlim_inf
by blast
  have 2: "∀F h in valid_path_nhds γ. path_image h ⊆ A"
    by (rule eventually_path_image_subset_valid_path_nhds) (use assms
in auto)
  have 3: "uniform_limit {0..1} ((∘) f) (f ∘ γ) (valid_path_nhds γ)"
    using uniform_limit_compose[OF 1 assms(2)] 2 assms by (simp add: o_def[abs_def]
path_image_def)
  have 4: "∀F x in valid_path_nhds γ. valid_path (f ∘ x) ∧ same_ends
(f ∘ x) (f ∘ γ)"
    using eventually_valid_path_valid_path_nhds[of γ] eventually_same_ends_valid_path_nhds[
γ] 2
  proof eventually_elim
    case (elim h)
```

```
    have "continuous_on (path_image h) f"
      using uniformly_continuous_imp_continuous continuous_on_subset elim
assms by blast
    hence "valid_path (f ∘ h)"
      using elim by (auto intro!: valid_path_compose_analytic analytic_on_subset[OF
assms(1)])
    moreover have "same_ends (f ∘ h) (f ∘ γ)"
      using elim by (auto simp: pathstart_compose pathfinish_compose)
    ultimately show ?case
      by blast
  qed

  from 3 and 4 show ?thesis
    unfolding valid_path_nhds_def filterlim_inf filterlim_principal by
simp_all
qed

lemma winding_number_comp':
  assumes f: "f holomorphic_on A" "uniformly_continuous_on A f" "inj_on
f A" "open A"
  assumes γ: "path γ" "path_image γ ⊆ A"
  assumes z: "z ∈ A" "z ∉ path_image γ"
  assumes int: "∃F p in valid_path_nhds γ.
                  contour_integral p (λw. deriv f w / (f w - f z)) = 2
* pi * ı * c"
  shows   "winding_number (f ∘ γ) (f z) = c"
proof -
  have cont: "continuous_on A f"
    using f(1) by (intro holomorphic_on_imp_continuous_on)

  have *: "f z ∉ f ` X" if "z ∉ X" "X ⊆ A" for X
  proof
    assume "f z ∈ f ` X"
    then obtain z' where z': "z' ∈ X" "f z' = f z"
      by force
    have "z' = z"
      using inj_onD[OF f(3) z'(2)] z'(1) z γ(2) interior_subset that by
auto
    with z'(1) and that show False
      by simp
  qed

  show ?thesis
  proof (rule winding_number_unique')
    show "path (f ∘ γ)"
      using assms
      by (intro path_continuous_image γ continuous_on_subset[OF cont])
  next
    show "f z ∉ path_image (f ∘ γ)"
```

```
      unfolding path_image_compose using assms interior_subset by (intro
*) auto
  next
    have ev: "∀F p in valid_path_nhds γ. path_image p ∩ {z} = {} ∧ path_image
p ⊆ A ∧ valid_path p"
      by (intro eventually_conj eventually_valid_path_nhds_avoid
              eventually_path_image_subset_valid_path_nhds eventually_valid_path_valid_pa
         (use assms in auto)
    have freq: "∃Fp in valid_path_nhds γ. winding_number (f ∘ p) (f
z) = c"
      using frequently_eventually_conj[OF int ev]
    proof (rule frequently_elim1, goal_cases)
      case (1 p)
      have "f z ∉ path_image (f ∘ p)"
        unfolding path_image_compose using * 1 by blast
      hence "winding_number (f ∘ p) (f z) =
               contour_integral (f ∘ p) (λw. 1 / (w - f z)) / (2 * pi
* i)" using 1
        by (subst winding_number_valid_path)
           (auto simp: path_image_compose intro!: valid_path_compose_holomorphic
                       holomorphic_on_subset[OF f(1)] <open A>)
      also have "... = c"
      proof (subst contour_integral_comp_analyticW)
        show "f analytic_on A"
          using assms by (simp add: analytic_on_open)
        show "valid_path p" "path_image p ⊆ A"
          using 1 by auto
      qed (use 1 <open A> in auto)
      finally show ?case .
    qed

    show "∃F p in valid_path_nhds (f ∘ γ). winding_number p (f z) =
c"
      using valid_path_nhds_compose unfolding filterlim_def
    proof (rule le_filterD_frequently)
      show "f analytic_on A"
        using assms by (simp_all add: analytic_on_open)
    qed (use assms freq in <auto simp: frequently_filtermap>)
  qed
qed

end
```

3 Prerequisites for the Detour Calculus

```
theory Detour_Prerequisites
  imports
    "HOL-Complex_Analysis.Residue_Theorem"
    "Winding_Number_Eval.Missing_Analysis"
```

```
    "Winding_Number_Eval.Missing_Transcendental"
    "Path_Automation.Path_Automation"
    Path_Nhds
    "HOL-Library.Real_Mod"
begin
```

3.1 Miscellaneous

```
lemma inverse_conv_cnj: "norm z = 1 ⟹ inverse z = cnj z"
  by (simp add: divide_conv_cnj inverse_eq_divide)

lemma dist_cnj: "dist (cnj x) (cnj y) = dist x y"
  by (simp add: dist_norm norm_complex_def power2_commute)

lemma closed_segment_cnj: "closed_segment (cnj w) (cnj z) = cnj ‘ closed_segment
w z"
  using closed_segment_linear_image linear_cnj by blast

lemma closed_segment_mult:
  "(*) (c :: 'a :: real_algebra_1) ‘ closed_segment a b = closed_segment
(c * a) (c * b)"
  by (rule closed_segment_linear_image [symmetric]) auto

lemma arcsin_pos: "x ∈ {0<..1} ⟹ arcsin x > 0"
  by (metis arcsin_0 arcsin_less_arcsin greaterThanAtMost_iff neg_le_0_iff_le
zero_less_one_class.zero_le_one)

lemma sin_lt_zero': "x ∈ {-pi<..<0} ⟹ sin x < 0"
  by (metis greaterThanLessThan_iff minus_less_iff neg_0_less_iff_less
sin_gt_zero sin_minus)

lemma tan_arcsin: "x ∈ {-1..1} ⟹ tan (arcsin x) = x / sqrt (1 - x
^ 2)"
  by (simp add: tan_def cos_arcsin)

lemma Arg_neg_real [simp]: "Im x = 0 ⟹ Re x < 0 ⟹ Arg x = pi"
  using Arg_eq_pi by blast

lemma cis_eq_1_iff':
  assumes "|x| < 2 * pi"
  shows   "cis x = 1 ⟷ x = 0"
proof
  assume "cis x = 1"
  then obtain n where n: "x = of_int n * 2 * pi"
    by (auto simp: cis_eq_1_iff)
  with assms have "n = 0"
    by (auto simp: abs_mult simp flip: of_int_abs)
  with n show "x = 0"
    by simp
```

```
qed auto

lemma DeMoivre_int: "cis x powi n = cis (of_int n * x)"
proof (cases "n ≥ 0")
  case True
  hence "cis x powi n = cis x ^ nat n"
    by (simp add: power_int_def)
  also have "... = cis (real (nat n) * x)"
    by (rule Complex.DeMoivre)
  finally show ?thesis
    using True by simp
next
  case False
  hence "cis x powi n = cis (-x) ^ nat (-n)"
    by (simp add: power_int_def)
  also have "... = cis (-real (nat (-n)) * x)"
    by (subst Complex.DeMoivre) auto
  finally show ?thesis
    using False by simp
qed

lemma cis_of_int_times_pi_half: "cis (of_int n * pi / 2) = i powi n"
 using DeMoivre_int[of "pi / 2" n] by simp

lemma cis_of_nat_times_pi_half: "cis (real n * pi / 2) = i ^ n"
  using cis_of_int_times_pi_half[of "int n"] by simp

lemma cis_of_int_times_pi: "cis (of_int n * pi) = (-1) powi n"
 using DeMoivre_int[of pi n] by simp

lemma cis_of_nat_times_pi: "cis (real n * pi) = (-1) ^ n"
  using cis_of_int_times_pi[of "int n"] by simp

lemma power3_i [simp]: "i ^ 3 = -i"
  and power4_i [simp]: "i ^ 4 = 1"
  by (simp_all add: eval_nat_numeral)

lemma i_power_mod4: "i ^ n = i ^ (n mod 4)"
proof -
  have "i ^ n = i ^ (4 * (n div 4) + n mod 4)"
    by simp
  also have "... = i ^ (n mod 4)"
    by (subst power_add) (simp add: power_mult)
  finally show ?thesis .
qed

lemma cis_numeral_times_pi_half [simp]:
  "cis (numeral num * pi / 2) = i ^ (numeral num mod 4)"
proof -
```

```
  have "cis (numeral num * pi / 2) = i ^ numeral num"
    using cis_of_nat_times_pi_half[of "numeral num"] by simp
  also have "... = i ^ (numeral num mod 4)"
    by (rule i_power_mod4)
  finally show ?thesis .
qed

lemma cis_numeral_pi_times_numeral_half [simp]:
  "cis (pi * numeral num / 2) = i ^ (numeral num mod 4)"
  by (subst mult.commute) simp

lemma cis_numeral_times_pi [simp]:
  "cis (numeral num * pi) = (-1) ^ (numeral num mod 2)"
proof -
  have "cis (numeral num * pi) = (-1) ^ numeral num"
    using cis_of_nat_times_pi[of "numeral num"] by simp
  also have "... = (-1) ^ (numeral num mod 2)"
    by (metis even_mod_2_iff neg_one_even_power neg_one_odd_power)
  finally show ?thesis .
qed

lemma cis_pi_times_numeral [simp]:
  "cis (pi * numeral num) = (-1) ^ (numeral num mod 2)"
  by (subst mult.commute) simp

lemma cis_minus_numeral_times_pi_half [simp]:
  "cis (-(numeral num * pi / 2)) = (-i) ^ (numeral num mod 4)"
  by (subst cis_cnj [symmetric]) auto

lemma cis_minus_numeral_times_pi [simp]:
  "cis (-(numeral num * pi)) = (-1) ^ (numeral num mod 2)"
  by (subst cis_cnj [symmetric]) auto

lemma cis_minus_pi_times_numeral_half [simp]:
  "cis (-(pi * numeral num / 2)) = (-i) ^ (numeral num mod 4)"
  by (subst cis_cnj [symmetric]) auto

lemma cis_minus_pi_times_numeral_pi [simp]:
  "cis (-(pi * numeral num)) = (-1) ^ (numeral num mod 2)"
  by (subst cis_cnj [symmetric]) auto

lemma cis_minus_pi [simp]: "cis (-pi) = -1"
  by (simp flip: cis_cnj)

lemma inner_mult_both_complex: "(z * x :: complex) · (z * y) = norm z
^ 2 * (x · y)"
  unfolding cmod_power2 by (simp add: inner_complex_def algebra_simps
power2_eq_square)
```

```
lemma orthogonal_transformation_mult_complex [intro]:
  "norm z = 1 ⟹ orthogonal_transformation ((*) (z :: complex))"
  unfolding orthogonal_transformation_def
  by (auto intro!: linear_times simp: inner_mult_both_complex)

lemma contour_integral_affine:
  assumes "valid_path γ" "c ≠ 0"
  shows "contour_integral ((λx. c * x + b) ∘ γ) f = contour_integral
γ (λw. c * f (c * w + b))"
proof -
  define ff where "ff=(λx. c*x+b)"
  have "contour_integral (ff ∘ γ) f = contour_integral γ (λw. deriv ff
w * f (ff w))"
  proof (rule contour_integral_comp_analyticW)
    show "ff analytic_on UNIV" "path_image γ ⊆ UNIV" "valid_path γ"
    unfolding ff_def using ‹valid_path γ›
    by (auto intro: analytic_intros)
  qed
  also have "... = contour_integral γ (λw. c * f (c * w + b))"
  proof -
    have "deriv ff  x = c" "ff x = c*x+b" for x
      unfolding ff_def by auto
    then show ?thesis by auto
  qed
  finally show ?thesis unfolding ff_def .
qed


lemma finite_imp_eventually_sparse_at_0:
  assumes "finite X"
  shows   "eventually (λε. sparse ε X) (at_right 0)"
proof (cases "X = {} ∨ is_singleton X")
  case False
  hence ne: "Sigma X (λx. X - {x}) ≠ {}"
    unfolding is_singleton_def by auto
  define m where "m = Min ((λ(x,y). dist x y) ` (Sigma X (λx. X - {x})))"
  have "m > 0"
    unfolding m_def using ne by (subst Min_gr_iff) (use assms in auto)
  show ?thesis
    using eventually_at_right_real[OF ‹m > 0›]
  proof eventually_elim
    case (elim ε)
    show ?case
    proof (intro sparseI)
      fix x y assume xy: "x ∈ X" "y ∈ X" "x ≠ y"
      have "ε < m"
        using elim by simp
      also have "m ≤ dist x y"
        using xy unfolding m_def by (intro Min.coboundedI) (use assms
```

```
in auto)
      finally show "dist x y > ε" .
    qed
  qed
qed (auto elim!: is_singletonE)
```

3.2 Lipschitz continuity

```
lemma complex_derivative_on_convex_imp_lipschitz:
  fixes f' :: "complex ⇒ complex"
  assumes deriv: "⋀z. z ∈ A ⟹ (f has_field_derivative f' z) (at z
within A)"
  assumes A: "convex A" and C: "⋀x. x ∈ A ⟹ norm (f' x) ≤ C" "C ≥
0"
  shows   "C-lipschitz_on A f"
proof (rule lipschitz_onI)
  fix x y assume xy: "x ∈ A" "y ∈ A"
  let ?l = "linepath y x"
  have "(f' has_contour_integral f (pathfinish ?l) - f (pathstart ?l))
?l"
  proof (rule contour_integral_primitive)
    show "(f has_field_derivative f' z) (at z within A)" if "z ∈ A" for
z
      using deriv[OF that] .
    show "path_image (linepath y x) ⊆ A"
      unfolding path_image_linepath by (intro closed_segment_subset xy
A)
  qed auto
  hence "norm (f (pathfinish ?l) - f (pathstart ?l)) ≤ C * norm (x -
y)"
  proof (rule has_contour_integral_bound_linepath)
    fix z assume "z ∈ closed_segment y x"
    also have "closed_segment y x ⊆ A"
      by (intro closed_segment_subset xy A)
    finally show "norm (f' z) ≤ C"
      using C(1)[of z] by auto
  qed fact+
  thus "dist (f x) (f y) ≤ C * dist x y"
    by (simp add: dist_norm norm_minus_commute)
qed fact+


lemma analytic_on_compact_convex_imp_lipschitz:
  assumes "f analytic_on A" "convex A" "compact A"
  obtains C where "C-lipschitz_on A f"
proof -
  have "deriv f analytic_on A"
    by (intro analytic_intros assms)
  define C where "C = Sup (norm ‘ (insert 0 (deriv f ‘ A)))"
```

```
  have "compact (insert 0 (deriv f ‘ A))"
    by (intro compact_insert compact_continuous_image analytic_imp_holomorphic
              holomorphic_on_imp_continuous_on analytic_intros assms)
  hence "bounded (insert 0 (deriv f ‘ A))"
    by (rule compact_imp_bounded)
  hence bdd: "bdd_above (norm ‘ insert 0 (deriv f ‘ A))"
    unfolding bdd_above_norm .

  show ?thesis
  proof (rule that[of C], rule complex_derivative_on_convex_imp_lipschitz)
    show "C ≥ 0"
      unfolding C_def by (intro cSup_upper bdd) auto
  next
    show "(f has_field_derivative deriv f z) (at z within A)" if "z ∈
A" for z
      using that assms(1) analytic_on_holomorphic holomorphic_derivI by
blast
  next
    show "norm (deriv f z) ≤ C" if "z ∈ A" for z
    proof -
      have "norm (deriv f z) ∈ norm ‘ insert 0 (deriv f ‘ A)"
        using that by auto
      moreover have "norm ‘ insert 0 (deriv f ‘ A) ≠ {}"
        by blast
      ultimately show "norm (deriv f z) ≤ C"
        unfolding C_def using bdd by (intro cSup_upper)
    qed
  qed fact+
qed

lemma lipschitz_on_complex_inverse:
  assumes "C > 0"
  shows   "(1/C^2)-lipschitz_on {z. Im z ≥ C} (λz. inverse z :: complex)"
proof (rule complex_derivative_on_convex_imp_lipschitz)
  show "((λz. inverse z :: complex) has_field_derivative (-1 / z ^ 2))
(at z within {z. Im z ≥ C})"
    if "z ∈ {z. Im z ≥ C}" for z
    using that assms by (auto intro!: derivative_eq_intros simp: power2_eq_square
field_simps)
  show "convex {z. Im z ≥ C}"
    by (simp add: convex_halfspace_Im_ge)
  show "cmod (- 1 / z^2) ≤ 1 / C^2" if "z ∈ {z. Im z ≥ C}" for z
  proof -
    from that and assms have [simp]: "z ≠ 0"
      by auto
    have "C ^ 2 ≤ Im z ^ 2"
      using assms that by (intro power_mono) auto
    also have "... ≤ norm z ^ 2"
```

```
    unfolding cmod_power2 by simp
  finally show ?thesis
    using assms by (simp add: field_simps norm_divide norm_power)
 qed
qed (use assms in auto)

lemma lipschitz_on_cnj [lipschitz_intros]:
 fixes f::"'a::metric_space ⇒ complex"
 assumes "C-lipschitz_on U f"
 shows "C-lipschitz_on U (λx. cnj (f x))"
proof (rule lipschitz_onI)
 fix x y assume xy: "x ∈ U" "y ∈ U"
 have "dist (cnj (f x)) (cnj (f y)) = norm (cnj (f x) - cnj (f y))"
   by (simp add: dist_norm)
 also have "cnj (f x) - cnj (f y) = cnj (f x - f y)"
   by simp
 also have "norm ... = dist (f x) (f y)"
   by (subst complex_mod_cnj) (auto simp: dist_norm)
 also have "... ≤ C * dist x y"
   by (rule lipschitz_onD[OF assms]) fact+
 finally show "dist (cnj (f x)) (cnj (f y)) ≤ C * dist x y"
   by (simp add: mult_ac)
qed (use assms lipschitz_on_nonneg in blast)

lemma lipschitz_on_cmult_complex [lipschitz_intros]:
 fixes f::"'a::metric_space ⇒ complex"
 assumes "C-lipschitz_on U f"
 shows "(norm c * C)-lipschitz_on U (λx. c * f x)"
proof (rule lipschitz_onI)
 have "C ≥ 0"
   using assms lipschitz_on_nonneg by blast
 thus "norm c * C ≥ 0"
   by simp
next
 fix x y assume xy: "x ∈ U" "y ∈ U"
 have "dist (c * f x) (c * f y) = norm c * dist (f x) (f y)"
   by (metis dist_norm norm_mult vector_space_over_itself.scale_right_diff_distrib)
 also have "... ≤ norm c * (C * dist x y)"
   by (intro mult_left_mono lipschitz_onD[OF assms]) (use xy in auto)
 finally show "dist (c * f x) (c * f y) ≤ cmod c * C * dist x y"
   by (simp add: mult_ac)
qed

lemma lipschitz_on_cmult_complex' [lipschitz_intros]:
 fixes f::"'a::metric_space ⇒ complex"
 assumes "C-lipschitz_on U f" "C' ≥ norm c * C"
 shows "C'-lipschitz_on U (λx. c * f x)"
 using lipschitz_on_cmult_complex[OF assms(1), of c] assms(2) lipschitz_on_le
by blast
```

```
lemma lipschitz_on_cadd_left [lipschitz_intros]:
  fixes f :: "_ ⇒ 'b :: real_normed_vector"
  assumes "C-lipschitz_on A f"
  shows   "C-lipschitz_on A (λx. c + f x)"
proof (rule lipschitz_onI)
  fix x y assume "x ∈ A" "y ∈ A"
  thus "dist (c + f x) (c + f y) ≤ C * dist x y"
    using lipschitz_onD[OF assms, of x y] by (simp add: dist_norm)
qed (use assms lipschitz_on_nonneg in blast)

lemma lipschitz_on_cadd_right [lipschitz_intros]:
  fixes f :: "_ ⇒ 'b :: real_normed_vector"
  assumes "C-lipschitz_on A f"
  shows   "C-lipschitz_on A (λx. f x + c)"
  by (subst add.commute, rule lipschitz_on_cadd_left [OF assms])
```

3.3 Homotopy

```
lemma simply_connected_imp_homotopic_paths:
  fixes S :: "'a :: real_normed_vector set"
  assumes "simply_connected S" "path p" "path_image p ⊆ S" "path q" "path_image
q ⊆ S"
  assumes "pathstart q = pathstart p ∧ pathfinish q = pathfinish p"
  shows   "homotopic_paths S p q"
  using assms unfolding simply_connected_eq_homotopic_paths by blast

lemma homotopic_loops_part_circlepath_circlepath:
  assumes "b = a + 2 * pi" "sphere x r ⊆ A" "r ≥ 0"
  shows   "homotopic_loops A (part_circlepath x r a b) (circlepath x r)"
proof -
  have "homotopic_loops A (shiftpath' (a / (2 * pi)) (circlepath x r))
(circlepath x r)"
    using assms by (intro homotopic_loops_shiftpath'_left) auto
  also have "shiftpath' (a / (2 * pi)) (circlepath x r) = part_circlepath
x r a ((a / pi + 2) * pi)"
    by (simp add: shiftpath'_circlepath)
  also have "(a / pi + 2) * pi = b"
    using assms by (simp add: divide_simps del: div_mult_self3 div_mult_self4
div_mult_self2 div_mult_self1)
  finally show ?thesis .
qed

lemma homotopic_loops_reversepath_D:
  "homotopic_loops A p q ⟹ homotopic_loops A (reversepath p) (reversepath
q)"
  apply (simp add: homotopic_loops_def homotopic_with_def, clarify)
  apply (rule_tac x="h ∘ (λx. (fst x, 1 - snd x))" in exI)
  apply (rule conjI continuous_intros)+
```

```
  apply (auto simp: reversepath_def pathstart_def pathfinish_def elim!:
continuous_on_subset)
  done

lemma homotopic_loops_reversepath:
  "homotopic_loops A (reversepath p) (reversepath q) ⟷ homotopic_loops
A p q"
  using homotopic_loops_reversepath_D reversepath_reversepath by metis

lemmas [trans] = homotopic_loops_trans



lemma homotopic_paths_split:
  assumes p: "path p" and A: "path_image p ⊆ A"
  assumes a: "a ∈ {0..1}"
  assumes eq1: "⋀x. x ∈ {0..1} ⟹ p1 x = subpath 0 a p x"
  assumes eq2: "⋀x. x ∈ {0..1} ⟹ p2 x = subpath a 1 p x"
  shows   "homotopic_paths A p (p1 +++ p2)"
proof -
  have "homotopic_paths (path_image p) (subpath 0 a p +++ subpath a 1
p) (subpath 0 1 p)"
    by (rule homotopic_join_subpaths) (use a p in auto)
  hence "homotopic_paths (path_image p) (subpath 0 1 p) (subpath 0 a p
+++ subpath a 1 p)"
    by (simp add: homotopic_paths_sym_eq)
  also have "homotopic_paths (path_image p) (subpath 0 a p +++ subpath
a 1 p) (p1 +++ p2)"
    using assms
    apply (intro homotopic_paths_eq)
      apply auto
     apply (subst (asm) path_image_join)
    apply auto
      apply (auto simp: joinpaths_def path_image_def split: if_splits)
      apply (metis (no_types, opaque_lifting) atLeastAtMost_iff image_subset_iff
le_numeral_extra(3) path_image_def path_image_subpath_subset zero_less_one_class.zero_le_on
    apply (metis (full_types) atLeastAtMost_iff image_subset_iff le_numeral_extra(4)
path_image_def path_image_subpath_subset zero_less_one_class.zero_le_one)
    done
  also have "subpath 0 1 p = p"
    by simp
  finally show ?thesis
    by (rule homotopic_paths_subset) fact
qed
```

3.4 Winding numbers

```
lemma winding_number_comp_plus:
  assumes "path γ" "z ∉ path_image γ"
```

```
  shows "winding_number ((+) c ∘ γ) (z + c) = winding_number γ z"
proof (rule winding_number_unique)
  define f where "f = (λx. c+x)"
  have f_alt:"f = (λx. x+c)" unfolding f_def by auto

  show "∃p. winding_number_prop (f ∘ γ)
              (z + c) e p (winding_number γ z)"
    if "e>0" for e
  proof -
    obtain p where "winding_number_prop γ z e p (winding_number γ z)"
      using ‹0 < e› assms winding_number by blast
    then have p:"valid_path p" "z ∉ path_image p"
        "pathstart p = pathstart γ"
        "pathfinish p = pathfinish γ"
        "(∀t∈{0..1}. cmod (γ t - p t) < e)" and
        p_cont:"contour_integral p (λw. 1 / (w - z)) =
          complex_of_real (2 * pi) * i * winding_number γ z"
      unfolding winding_number_prop_def by auto

    have "valid_path (f ∘ p)"
      using p(1) unfolding f_def valid_path_translation_eq by simp
    moreover have "z + c ∉ path_image (f ∘ p)"
      using p(2) unfolding f_def by (auto simp add:path_image_def)
    moreover have "pathstart (f ∘ p) = pathstart (f ∘ γ)"
      using p(3) by (simp add:pathstart_compose)
    moreover have "pathfinish (f ∘ p) = pathfinish (f ∘ γ)"
      using p(4) by (simp add:pathfinish_compose)
    moreover have "(∀t∈{0..1}. cmod ((f ∘ γ) t
          - (f ∘ p) t) < e)"
      using p(5) unfolding f_def by simp
    moreover have "contour_integral (f ∘ p)
            (λw. 1 / (w - (z + c))) =
              complex_of_real (2 * pi) * i * winding_number γ z"
      unfolding f_alt
      apply (subst contour_integral_affine[where c=1,simplified])
      using p_cont p(1,2) by auto
    ultimately show ?thesis
      apply (rule_tac x="f ∘ p" in exI)
      unfolding winding_number_prop_def by auto
  qed
  show "path (f ∘ γ)"
    using ‹path γ› unfolding f_def by (simp add:path_translation_eq)
  show "z + c ∉ path_image (f ∘ γ)"
    using ‹z ∉ path_image γ› unfolding path_image_def f_def by auto
qed

lemma winding_number_comp_times:
  assumes "path γ"
      and "z ∉ path_image γ"
```

```
      and "c ≠ 0"
    shows "winding_number ((*) c ∘ γ) (z * c) = winding_number γ z"
proof (rule winding_number_unique)
  define f where "f = (λx. c*x)"
  have f_alt:"f = (λx. x*c)" unfolding f_def by auto

  show "∃p. winding_number_prop (f ∘ γ)
              (z * c) e p (winding_number γ z)"
    if "e>0" for e
  proof -
    have "cmod c>0" using ‹c≠0› by simp
    then have "(1/cmod c) * e>0"
      using that by auto
    from winding_number[OF assms(1,2) this]
    obtain p where "winding_number_prop γ z ((1/cmod c) * e)
      p (winding_number γ z)"
      by blast
    then have p:"valid_path p" "z ∉ path_image p"
      "pathstart p = pathstart γ"
      "pathfinish p = pathfinish γ"
      "(∀t∈{0..1}. cmod (γ t - p t) < ((1/cmod c) * e))" and
      p_cont:"contour_integral p (λw. 1 / (w - z)) =
        complex_of_real (2 * pi) * i * winding_number γ z"
      unfolding winding_number_prop_def by auto

    have "valid_path (f ∘ p)"
      using p(1) ‹c≠0› valid_path_times
      unfolding f_def by auto
    moreover have "z * c ∉ path_image (f ∘ p)"
      using p(2) ‹c≠0› unfolding f_def
      by (auto simp add:path_image_def)
    moreover have "pathstart (f ∘ p) = pathstart (f ∘ γ)"
      using p(3) by (simp add:pathstart_compose)
    moreover have "pathfinish (f ∘ p) = pathfinish (f ∘ γ)"
      using p(4) by (simp add:pathfinish_compose)
    moreover have "cmod ((f ∘ γ) t - (f ∘ p) t) < e"
      if "t∈{0..1}" for t
    proof -
      have "cmod ((f ∘ γ) t - (f ∘ p) t) = cmod (c*(γ t - p t))"
        unfolding f_def by (auto simp:algebra_simps)
      also have "... = cmod c * cmod (γ t - p t)"
        by (auto simp:norm_mult)
      also have "... < cmod c * (1 / cmod c * e)"
        using p(5)[rule_format,OF that] ‹cmod c>0›
        using mult_less_cancel_left_pos by blast
      also have "... = e"
        using ‹cmod c>0› by auto
      finally show ?thesis .
    qed
```

```
    moreover have "contour_integral (f ∘ p)
              (λw. 1 / (w - (z * c))) =
          complex_of_real (2 * pi) * i * winding_number γ z" (is "?L=?R")
    proof -
      have "?L = contour_integral p (λw. c / (c * (w - z)))"
        unfolding f_def
        apply (subst contour_integral_affine[where b=0,simplified])
        using p(1,2) <c≠0> by (auto simp:algebra_simps)
      also have "... = contour_integral p (λw. 1 / ( w - z))"
        using <c≠0> by simp
      also have "... = ?R" using p_cont .
      finally show ?thesis .
    qed
    ultimately show ?thesis
      apply (rule_tac x="f ∘ p" in exI)
      unfolding winding_number_prop_def by auto
  qed
  show "path (f ∘ γ)"
    using path_mult[OF path_const <path γ>] unfolding f_def comp_def
    by simp
  show "z * c ∉ path_image (f ∘ γ)"
    using <z ∉ path_image γ> <c≠0>
    unfolding path_image_def f_def by auto
qed

lemma winding_number_part_circlepath_full:
  assumes "y ∈ ball x r" "α + 2 * pi = β"
  shows   "winding_number (part_circlepath x r α β) y = 1"
proof -
  have "0 ≤ dist x y"
    by auto
  also have "... < r"
    using assms by auto
  finally have "r > 0" .
  have "homotopic_loops (-{y}) (part_circlepath x r α (α + 2 * pi)) (circlepath
x r)"
    by (rule homotopic_loops_part_circlepath_circlepath) (use assms <r
> 0> in auto)
  hence "winding_number (part_circlepath x r α (α + 2 * pi)) y = winding_number
(circlepath x r) y"
    by (rule winding_number_homotopic_loops)
  also have "... = 1"
    by (intro winding_number_circlepath) (use assms in <auto simp: dist_norm
norm_minus_commute>)
  also have "α + 2 * pi = β"
    using assms(2) by (simp add: algebra_simps)
  finally show ?thesis .
qed
```

```
lemma winding_number_part_circlepath_full':
  assumes "y ∈ ball x r" "α - 2 * pi = β"
  shows   "winding_number (part_circlepath x r α β) y = -1"
proof -
  have "0 ≤ dist x y"
    by simp
  also from assms have "dist x y < r"
    by auto
  finally have r: "r > 0"
    by simp
  have "winding_number (part_circlepath x r (α - 2 * pi) α) y = 1"
    by (rule winding_number_part_circlepath_full) (use assms in auto)
  also have "part_circlepath x r (α - 2 * pi) α = reversepath (part_circlepath
x r α (α - 2 * pi))"
    by simp
  also have "winding_number ... y = -winding_number (part_circlepath x
r α (α - 2 * pi)) y"
    using path_image_part_circlepath_subset'[of r x α "α - 2 * pi"] r
assms
    by (intro winding_number_reversepath) auto
  also have "α - 2 * pi = β"
    using assms(2) by (simp add: algebra_simps)
  finally show ?thesis
    by Groebner_Basis.algebra
qed

lemma winding_number_inverse_valid_path:
  assumes "valid_path γ" "0 ∉ path_image γ" "z ∉ path_image γ" "z ≠
0"
  shows "winding_number (inverse ∘ γ) (inverse z) = winding_number γ
z - winding_number γ 0"
proof -
  define C where "C = 1 / (complex_of_real (2 * pi) * i)"
  have "winding_number (inverse ∘ γ) (inverse z)
        = C * contour_integral γ (λw. deriv inverse w / (inverse w - inverse
z))"
    unfolding C_def
  proof (rule winding_number_comp[of "UNIV - {0,z}"])
    show "open (UNIV - {0, z})"
      by (metis Diff_insert open_UNIV open_delete)
    show "inverse holomorphic_on UNIV - {0, z}"
      by (auto intro:holomorphic_intros)
    show "path_image γ ⊆ UNIV - {0, z}" "inverse z ∉ path_image (inverse
∘ γ)"
      using ‹0 ∉ path_image γ› ‹z ∉ path_image γ› unfolding path_image_def
      by auto
  qed fact
  also have "... = C * (contour_integral γ (λw. 1 / (w - z) - 1 / w))"
  proof -
```

```
    have "contour_integral γ (λw. deriv inverse w / (inverse w - inverse
z)) =
          contour_integral γ (λw. 1 / (w - z) - 1 / w)"
    proof (rule contour_integral_cong)
      fix x assume "x ∈ path_image γ"
      then have "x≠0" "x≠z"
        using <0 ∉ path_image γ> <z ∉ path_image γ> unfolding path_image_def
        by auto
      then show "deriv inverse x / (inverse x - inverse z) = 1 / (x -
z) - 1 / x"
        using <z≠0>
        by (auto simp:divide_simps power2_eq_square algebra_simps)
    qed simp
    then show ?thesis by simp
  qed
  also have "... = C * (contour_integral γ (λw. 1 / (w - z)) - contour_integral
γ (λw. 1 / w))"
  proof (subst contour_integral_diff)
    show "(λw. 1 / (w - z)) contour_integrable_on γ"
      by (rule contour_integrable_inversediff) fact+
    show "(/) 1 contour_integrable_on γ"
      using contour_integrable_inversediff[OF <valid_path γ> <0 ∉ path_image
γ>]
      by simp
  qed simp
  also have "... =  winding_number γ z - winding_number γ 0"
    unfolding C_def by (subst (1 2) winding_number_valid_path) (use assms
in <auto simp: algebra_simps>)
  finally show ?thesis .
qed

lemma winding_number_inverse:
  assumes "path γ" "path_image γ ⊆ {z. Im z > 0}" "z ∉ path_image γ"
"Im z > 0"
  shows   "winding_number (inverse ∘ γ) (inverse z) = winding_number
γ z - winding_number γ 0"
proof -
  have compact: "compact (Im ` ({z} ∪ (path_image γ)))"
    by (intro compact_continuous_image compact_Un compact_path_image assms
continuous_intros) auto
  hence bdd: "bdd_below (Im ` ({z} ∪ path_image γ))"
    by (meson bounded_imp_bdd_below compact_imp_bounded)
  define c' where "c' = Inf (Im ` ({z} ∪ path_image γ))"
  have c'_le: "Im u ≥ c'" if "u ∈ {z} ∪ path_image γ" for u
    using that bdd unfolding c'_def by (meson cINF_lower)
  have "c' ∈ Im ` ({z} ∪ path_image γ)"
    unfolding c'_def using compact by (intro closed_contains_Inf compact_imp_closed
bdd) auto
  with assms have "c' > 0"
```

```
    by auto
  define c where "c = c' / 2"
  have "c > 0" "c < c'"
    using ‹c' > 0› by (simp_all add: c_def)
  have c: "c > 0" "Im z > c" "⋀u. u ∈ path_image γ ⟹ Im u > c"
    using ‹c > 0› ‹c < c'› c'_le by force+

  show ?thesis
  proof (rule winding_number_comp')
    show "inverse holomorphic_on {z. Im z > c}"
      using c by (intro holomorphic_intros) auto
    have "(1/c^2)-lipschitz_on {z. Im z > c} inverse"
      by (rule lipschitz_on_subset[OF lipschitz_on_complex_inverse[of
c]]) (use c in auto)
    thus "uniformly_continuous_on {z. c < Im z} inverse"
      by (rule lipschitz_on_uniformly_continuous)
    show "inj_on inverse {z. Im z > c}"
      using assms by (auto simp: inj_on_def)
    show "open {z. Im z > c}"
      by (simp add: open_halfspace_Im_gt)
    show "path γ" "path_image γ ⊆ {z. Im z > c}" "z ∈ {z. c < Im z}"
"z ∉ path_image γ"
      using c assms by auto
    have "∀F p in valid_path_nhds γ. valid_path p ∧ path_image p ∩ ({z.
Im z ≤ c} ∪ {z}) = {}"
      by (intro eventually_conj eventually_valid_path_valid_path_nhds
               eventually_valid_path_nhds_avoid closed_Un closed_halfspace_Im_le)
         (use assms c in force)+
    moreover have "∀F p in path_nhds γ. winding_number p 0 = winding_number
γ 0 ∧
                                  winding_number p z = winding_number γ z"
      by (intro eventually_conj eventually_winding_number_eq_path_nhds)
(use assms in auto)
    hence "∀F p in valid_path_nhds γ. winding_number p 0 = winding_number
γ 0 ∧
                                       winding_number p z = winding_number
γ z"
      by (meson filter_leD path_nhds_le_valid_path_nhds)
    ultimately have "∀F p in valid_path_nhds γ.
           contour_integral p (λw. deriv inverse w / (inverse w - inverse
z)) =
           complex_of_real (2 * pi) * i * (winding_number γ z - winding_number
γ 0)"
    proof eventually_elim
      case (elim p)
      have "contour_integral p (λw. deriv inverse w / (inverse w - inverse
z)) =
           contour_integral p (λw. 1 / (w - z) - 1 / w)"
      proof (rule contour_integral_cong)
```

```
      fix x assume "x ∈ path_image p"
      then have "x ≠ 0" "x ≠ z"
        using elim c by auto
      then show "deriv inverse x / (inverse x - inverse z) = 1 / (x
- z) - 1 / x"
        using assms by (auto simp: divide_simps power2_eq_square)
    qed simp
    also have "... = contour_integral p (λw. 1 / (w - z)) - contour_integral
p ((/) 1)"
    proof (rule contour_integral_diff)
      show "(λw. 1 / (w - z)) contour_integrable_on p"
        by (rule contour_integrable_inversediff) (use elim in auto)
      have "(λw. 1 / (w - 0)) contour_integrable_on p"
        by (rule contour_integrable_inversediff) (use elim c in auto)
      thus "(λw. 1 / w) contour_integrable_on p"
        by simp
    qed
    also have "... = (2 * pi * i) * (winding_number p z - winding_number
p 0)"
      by (subst (1 2) winding_number_valid_path) (use elim c in ‹auto
simp: algebra_simps›)
    finally show ?case
      using elim by (simp add: algebra_simps)
  qed
  thus "∃F p in valid_path_nhds γ.
          contour_integral p (λw. deriv inverse w / (inverse w - inverse
z)) =
          complex_of_real (2 * pi) * i * (winding_number γ z - winding_number
γ 0)"
    by (rule eventually_frequently [rotated]) (use assms in auto)
 qed
qed

lemma winding_number_inverse_valid_path_0:
 assumes "valid_path γ" "pathstart γ = pathfinish γ" "0 ∉ path_image
γ"
 shows   "winding_number (inverse ∘ γ) 0 = -winding_number γ 0"
proof -
 have val: "valid_path (inverse ∘ γ)" using assms
   by (intro valid_path_compose_holomorphic[of _ _ "-{0}"] assms)
      (auto intro!: holomorphic_intros)

 obtain B where B: "⋀w. norm w ≥ B ⟹ winding_number γ w = 0"
   using winding_number_zero_at_infinity[of γ] val assms
   by (auto simp: valid_path_imp_path)

 have "compact (path_image γ ∪ path_image (inverse ∘ γ))"
   using assms by (intro compact_Un compact_path_image valid_path_imp_path
val) auto
```

```
  hence "open (-(path_image γ ∪ path_image (inverse ∘ γ)))"
    by (intro open_Compl compact_imp_closed)
  moreover have "0 ∈ -(path_image γ ∪ path_image (inverse ∘ γ))"
    using assms by (auto simp: path_image_compose)
  ultimately obtain ε where ε: "ε > 0" "cball 0 ε ⊆ -(path_image γ ∪
path_image (inverse ∘ γ))"
    unfolding open_contains_cball by blast
  define w where "w = complex_of_real (min ε (1 / max 1 B))"

  have pos: "min ε (1 / max 1 B) > 0"
    using ε by (auto simp: min_less_iff_disj)
  hence norm_w: "norm w = min ε (1 / max 1 B)"
    unfolding w_def norm_of_real by simp
  from pos have "norm w ≠ 0"
    unfolding norm_w by linarith
  hence "w ≠ 0"
    by auto
  have "w ∈ cball 0 ε"
    by (auto simp: norm_w)

  have "B ≤ max 1 B"
    by simp
  also have "max 1 B = 1 / (1 / max 1 B)"
    by (auto simp: field_simps)
  also have "... ≤ 1 / min ε (1 / max 1 B)"
    using ε by (intro divide_left_mono) auto
  also have "... = norm (inverse w)"
    by (simp add: ‹norm w = _› norm_inverse field_simps norm_divide)
  finally have "B ≤ norm (inverse w)" .

  have "w ∉ inverse ‘ path_image γ"
    using ‹w ∈ cball 0 ε› ε unfolding path_image_compose by blast
  hence "inverse w ∉ path_image γ"
    using ‹w ≠ 0› by (metis image_iff inverse_inverse_eq)

  have "winding_number (inverse ∘ γ) 0 = winding_number (inverse ∘ γ)
w"
  proof (rule winding_number_eq)
    show "w ∈ cball 0 ε" "0 ∈ cball 0 ε" "cball 0 ε ∩ path_image (inverse
∘ γ) = {}"
         "connected (cball 0 ε :: complex set)"
      using ε by (auto simp: w_def)
  qed (use assms val in ‹auto intro: valid_path_imp_path simp: pathfinish_def
pathstart_def›)
  also have "winding_number (inverse ∘ γ) w = winding_number (inverse
∘ γ) (inverse (inverse w))"
    using ‹w ≠ 0› by simp
  also have "... = winding_number γ (inverse w) - winding_number γ 0"
    using assms ‹w ∈ cball 0 ε› ‹w ≠ 0› ε ‹inverse w ∉ path_image γ›
```

```
    by (subst winding_number_inverse_valid_path) (auto simp: path_image_compose)
  also have "winding_number γ (inverse w) = 0"
    using ε ‹B ≤ norm (inverse w)› by (intro B)
  finally show ?thesis by (simp add: o_def)
qed
```

The following allows us to compute the winding number of a non-closed circular arc with respect to its centre.

```
lemma winding_number_part_circlepath_centre:
  assumes "r > 0"
  shows   "winding_number (part_circlepath z r a b) z = (b - a) / (2 *
pi)"
proof -
  have "winding_number (part_circlepath 0 r a b) 0 = (b - a) / (2 * pi)"
  proof (induction a b rule: linorder_wlog)
    case (le a b)
    show ?case
    proof (cases "a = b")
      case True
      thus ?thesis
        using assms by (simp add: part_circlepath_empty winding_number_zero_const)
    next
      case False
      let ?p = "part_circlepath 0 r a b"
      have "((λt. i) has_integral (b - a) *R i) {a..b}"
        using has_integral_const[of i a b] assms False le by (simp add:
scaleR_conv_of_real mult_ac)
      hence "((λz. 1 / z) has_contour_integral (b - a) *R i) ?p"
        by (subst has_contour_integral_part_circlepath_iff) (use assms
False le in simp_all)
      moreover have "0 ∉ path_image ?p"
        using assms by (auto simp: path_image_part_circlepath')
      hence "((λz. 1 / z) has_contour_integral 2 * i * pi * winding_number
?p 0) ?p"
        using has_contour_integral_winding_number[of ?p 0] assms by (auto
simp add: mult_ac)
      ultimately have "(b - a) *R i = 2 * i * pi * winding_number ?p 0"
        using has_contour_integral_unique by blast
      thus ?thesis
        by (simp add: scaleR_conv_of_real)
    qed
  next
    case (sym a b)
    from assms have "0 ∉ path_image (part_circlepath 0 r a b)"
      by (auto simp: path_image_part_circlepath')
    thus ?case using assms sym
      by (simp add: reversepath_part_circlepath [symmetric, of 0 r b a]
                    winding_number_reversepath field_simps del: reversepath_part_circlepath
  qed
```

```
  hence "winding_number ((+) z ∘ part_circlepath 0 r a b) (0 + z) = (b
- a) / (2 * pi)"
    by (subst winding_number_comp_plus) (use assms in ‹auto simp: path_image_part_circlepat
  thus ?thesis
    by (subst (asm) part_circlepath_translate) auto
qed
```

3.5 Continuous transformations that preserve the winding number in some way

```
locale winding_preserving =
  fixes A :: "complex set" and f :: "complex ⇒ complex" and g :: "complex
⇒ complex"
  assumes inj: "inj_on f A"
  assumes cont: "continuous_on A f"
  assumes winding_number_eq:
    "⋀p x. path p ⟹ path_image p ⊆ A ⟹ pathstart p = pathfinish
p ⟹ x ∈ A - path_image p ⟹
       winding_number (f ∘ p) (f x) = g (winding_number p x)"

lemma winding_preserving_comp:
  assumes "winding_preserving B f2 g2"
  assumes "winding_preserving A f1 g1"
  assumes subset: "f1 ‘ A ⊆ B"
  shows   "winding_preserving A (f2 ∘ f1) (g2 ∘ g1)"
proof
  interpret f1: winding_preserving A f1 g1
    by fact
  interpret f2: winding_preserving B f2 g2
    by fact
  show "inj_on (f2 ∘ f1) A"
    by (intro comp_inj_on f1.inj inj_on_subset[OF f2.inj] assms)
  show "continuous_on A (f2 ∘ f1)"
    by (intro continuous_on_compose f1.cont continuous_on_subset[OF f2.cont]
subset)
  show "winding_number (f2 ∘ f1 ∘ p) ((f2 ∘ f1) x) = (g2 ∘ g1) (winding_number
p x)"
    if p: "path p" "path_image p ⊆ A" "x ∈ A - path_image p" "pathstart
p = pathfinish p" for p x
  proof -
    have [simp]: "f1 x = f1 (p t) ⟷ x = p t" if "t ∈ {0..1}" for t
      using f1.inj that p unfolding inj_on_def path_image_def by blast
    have "winding_number (f2 ∘ f1 ∘ p) ((f2 ∘ f1) x) =
          winding_number (f2 ∘ (f1 ∘ p)) (f2 (f1 x))"
      by (simp add: o_def)
    also have "... = g2 (winding_number (f1 ∘ p) (f1 x))"
    proof (rule f2.winding_number_eq)
      show "path (f1 ∘ p)"
        using that by (intro path_continuous_image continuous_on_subset[OF
```

```
f1.cont])
      show "path_image (f1 ∘ p) ⊆ B"
        using that subset by (auto simp: path_image_def)
      show "f1 x ∈ B - path_image (f1 ∘ p)"
        using that subset f1.inj by (auto simp: path_image_def)
    qed (use p in ‹auto simp: pathstart_compose pathfinish_compose›)
    also have "winding_number (f1 ∘ p) (f1 x) = g1 (winding_number p x)"
      by (rule f1.winding_number_eq) (use p in auto)
    finally show ?thesis
      by (simp add: o_def)
  qed
qed

lemmas winding_preserving_comp' = winding_preserving_comp [unfolded o_def]

lemma winding_preserving_subset:
  assumes "winding_preserving A f g" "B ⊆ A"
  shows   "winding_preserving B f g"
proof -
  interpret winding_preserving A f g
    by fact
  show ?thesis
  proof
    show "inj_on f B"
      by (rule inj_on_subset[OF inj]) (use assms(2) in auto)
    show "continuous_on B f"
      by (rule continuous_on_subset[OF cont]) (use assms(2) in auto)
    show "winding_number (f ∘ p) (f x) = g (winding_number p x)"
      if "path p" "path_image p ⊆ B" "x ∈ B - path_image p" "pathstart
p = pathfinish p" for p x
      using that winding_number_eq[of p x] assms(2) by auto
  qed
qed

lemma winding_preserving_translate: "winding_preserving A (λx. c + x)
(λx. x)"
proof
  show "winding_number ((+) c ∘ p) (c + x) = winding_number p x"
    if "path p" "path_image p ⊆ A" "x ∈ A - path_image p" for p x
    using that winding_number_comp_plus[of p x c] by (auto simp: algebra_simps)
qed (auto intro!: inj_onI continuous_intros)

lemma winding_preserving_mult: "c ≠ 0 ⟹ winding_preserving A (λx.
c * x) (λx. x)"
proof
  assume "c ≠ 0"
  show "winding_number ((*) c ∘ p) (c * x) = winding_number p x"
    if "path p" "path_image p ⊆ A" "x ∈ A - path_image p" for p x
    using that winding_number_comp_times[of p x c] ‹c ≠ 0› by (auto simp:
```

```
algebra_simps)
qed (auto intro!: inj_onI continuous_intros)

lemma winding_preserving_cnj: "winding_preserving A cnj (λx. -cnj x)"
proof
  show "winding_number (cnj ∘ p) (cnj x) = -cnj (winding_number p x)"
    if "path p" "path_image p ⊆ A" "x ∈ A - path_image p" for p x
    using that winding_number_cnj[of p x] by (auto simp: algebra_simps)
qed (auto intro!: inj_onI continuous_intros)

lemma winding_preserving_uminus: "winding_preserving A (λx. -x) (λx.
x)"
  using winding_preserving_mult[of "-1" A] by simp
```

3.6 Paths

```
lemma simple_path_cnj [simp]: "simple_path (cnj ∘ p) ⟷ simple_path
p"
  by (rule simple_path_linear_image_eq) (auto intro!: linear_cnj simp:
inj_def)

lemma part_circlepath_cnj': "cnj ∘ part_circlepath c r a b = part_circlepath
(cnj c) r (-a) (-b)"
  unfolding o_def by (intro ext part_circlepath_cnj)

lemma linepath_minus: "linepath (-a) (-b) x = -linepath a b x"
  by (simp add: linepath_def algebra_simps)
```

The following lemma is very difficult to bypass some nasty geometric reasoning: If a path only touches the frontier of a set at its beginning or end, then it is either fully inside the set or fully outside the set.

```
lemma path_fully_inside_or_fully_outside:
  fixes p :: "real ⇒ 'a :: euclidean_space"
  assumes "path p" "⋀x. x ∈ {0<..<1} ⟹ p x ∉ frontier A"
  shows   "path_image p ⊆ closure A ∨ path_image p ∩ interior A = {}"
proof (rule ccontr)
  assume *: "¬(path_image p ⊆ closure A ∨ path_image p ∩ interior A
= {})"
  from * obtain x y where xy: "x ∈ {0..1}" "y ∈ {0..1}" "p x ∉ closure
A" "p y ∈ interior A"
    unfolding path_image_def by blast
  define x' y' where "x' = min x y" and "y' = max x y"
  have xy': "x' ∈ {0..1}" "y' ∈ {0..1}" "x' ≤ y'"
    using xy by (auto simp: x'_def y'_def)

  define q where "q = subpath x' y' p"
  have [simp]: "path q"
    using xy' by (auto simp: q_def assms)
```

```
  have "path_image q ∩ frontier A ≠ {}"
  proof (rule connected_Int_frontier)
    show "connected (path_image q)"
      by auto
  next
    have "q (if x ≤ y then 0 else 1) ∈ path_image q"
      by (auto simp: path_image_def)
    moreover have "q (if x ≤ y then 0 else 1) ∉ A"
      using xy closure_subset by (auto simp: q_def subpath_def x'_def
y'_def)
    ultimately show "path_image q - A ≠ {}"
      by blast
  next
    have "q (if x ≤ y then 1 else 0) ∈ path_image q"
      by (auto simp: path_image_def)
    moreover have "q (if x ≤ y then 1 else 0) ∈ A"
      using xy interior_subset by (auto simp: q_def subpath_def x'_def
y'_def)
    ultimately show "path_image q ∩ A ≠ {}"
      by blast
  qed
  then obtain w where w: "w ∈ {x'..y'}" "p w ∈ frontier A"
    unfolding q_def path_image_subpath using xy' by (force split: if_splits)

  have "w ≠ x"
    using xy w by (metis DiffE frontier_def)
  moreover have "w ≠ y"
    using xy w unfolding frontier_def by auto
  ultimately have "w ∈ {x'<..<y'}"
    using w(1) by (auto simp: x'_def y'_def)
  also have "... ⊆ {0<..<1}"
    using xy' by auto
  finally have "w ∈ {0<..<1}" .
  with assms(2)[of w] have "p w ∉ frontier A"
    by blast
  with ‹p w ∈ frontier A› show False
    by contradiction
qed

lemma simple_path_joinE':
  assumes "simple_path (g1 +++ g2)" and "pathfinish g1 = pathstart g2"
  shows   "path_image g1 ∩ path_image g2 ⊆
             (insert (pathstart g2) (if pathstart g1 = pathfinish g2 then
{pathstart g1} else {}))"
  using assms
  by (smt (verit, del_insts) arc_join_eq insert_commute pathfinish_join
pathstart_join simple_path_cases simple_path_joinE)

lemma simple_path_joinE'':
```

```
  assumes "simple_path (g1 +++ g2)" and "pathfinish g1 = pathstart g2"
          "x ∈ path_image g1" "x ∈ path_image g2"
  shows   "x = pathstart g2 ∨ x = pathfinish g2 ∧ pathstart g1 = pathfinish
g2"
  using simple_path_joinE'[OF assms(1,2)] assms(3,4) by (auto split: if_splits)

lemma arc_continuous_image:
  assumes "arc p" "inj_on f (path_image p)" "continuous_on (path_image
p) f"
  shows   "arc (f ∘ p)"
  using assms by (auto simp: arc_def inj_on_def path_image_def intro!:
path_continuous_image)

lemma eventually_path_image_cball_subset:
  fixes p :: "real ⇒ 'a :: real_normed_vector"
  assumes "path p" "path_image p ⊆ interior A"
  shows "eventually (λε. (⋃x∈path_image p. cball x ε) ⊆ A) (at_right
0)"
proof -
  from assms have "compact ( path_image p)"
    by (intro compact_path_image) auto
  moreover have "path_image p ∩ frontier A = {}"
    using assms by (simp add: disjoint_iff frontier_def subset_eq)
  ultimately have "eventually (λε. setdist_gt ε (path_image p) (frontier
A)) (at_right 0)"
    by (intro compact_closed_imp_eventually_setdist_gt_at_right_0) auto
  moreover have "eventually (λε. ε > 0) (at_right (0 :: real))"
    by (simp add: eventually_at_right_less)
  ultimately have "eventually (λε. (⋃x∈path_image p. cball x ε) ⊆ A)
(at_right 0)"
  proof eventually_elim
    case (elim ε)
    show "(⋃x∈path_image p. cball x ε) ⊆ A"
    proof (intro subsetI, elim UN_E)
      fix x y assume xy: "x ∈ path_image p" "y ∈ cball x ε"
      show "y ∈ A"
      proof (rule ccontr)
        assume "y ∉ A"
        have "cball x ε ∩ frontier A ≠ {}"
        proof (rule connected_Int_frontier)
          have "x ∈ cball x ε"
            using ‹ε > 0› by simp
          moreover have "x ∈ A"
            using ‹x ∈ path_image p› assms interior_subset by blast
          ultimately show "cball x ε ∩ A ≠ {}"
            by blast
        next
          from xy show "cball x ε - A ≠ {}"
            using ‹y ∉ A› by blast
```

```
        qed auto
        hence "¬setdist_gt ε {x} (frontier A)"
          by (force simp: setdist_gt_def)
        moreover have "setdist_gt ε {x} (frontier A)"
          by (rule setdist_gt_mono[OF elim(1)]) (use xy in auto)
        ultimately show False by contradiction
      qed
    qed
  qed
  thus ?thesis
    by eventually_elim (use assms interior_subset in auto)
qed

lemma eventually_path_image_cball_subset':
  fixes p :: "real ⇒ 'a :: real_normed_vector"
  assumes "path p" "path_image p ⊆ interior A" "X ⊆ path_image p"
  shows "eventually (λε. path_image p ∪ (⋃x∈X. cball x ε) ⊆ A) (at_right
0)"
  using eventually_path_image_cball_subset[OF assms(1-2)]
        eventually_at_right_less[of 0]
proof eventually_elim
  case (elim ε)
  have "path_image p ∪ (⋃x∈X. cball x ε) = (⋃x∈path_image p. {x})
∪ (⋃x∈X. cball x ε)"
    by blast
  also have "... ⊆ (⋃x∈path_image p. cball x ε) ∪ (⋃x∈X. cball x ε)"
    using elim(2) by (intro Un_mono UN_mono) auto
  also have "... = (⋃x∈path_image p ∪ X. cball x ε)"
    by blast
  also have "path_image p ∪ X = path_image p"
    using assms by blast
  also have "(⋃x∈path_image p. cball x ε) ⊆ A"
    by fact
  finally show ?case .
qed
```

We say that a path does not cross a set `A` if it enters `A` at most at its beginning and end, and never inbetween.

```
definition does_not_cross :: "(real ⇒ 'a :: real_vector) ⇒ 'a set ⇒
bool" where
  "does_not_cross p A ⟷ (∀x∈{0<..<1}. p x ∉ A)"

lemma does_not_cross_simple_path:
  assumes "simple_path p"
  shows   "does_not_cross p A ⟷ path_image p ∩ A ⊆ {pathstart p, pathfinish
p}"
proof
  assume "does_not_cross p A"
  thus "path_image p ∩ A ⊆ {pathstart p, pathfinish p}" using assms
```

```
    by (force simp: does_not_cross_def simple_path_def path_image_def
pathstart_def pathfinish_def)
next
  assume *: "path_image p ∩ A ⊆ {pathstart p, pathfinish p}"
  show "does_not_cross p A"
    unfolding does_not_cross_def
  proof safe
    fix x :: real assume x: "x ∈ {0<..<1}" "p x ∈ A"
    hence "p x ∉ {pathstart p, pathfinish p}"
      using assms by (force simp: simple_path_def loop_free_def pathstart_def
pathfinish_def)+
    moreover from x have "p x ∈ path_image p"
      by (auto simp: path_image_def)
    ultimately show False
      using * and x by blast
  qed
qed

lemma path_fully_inside_or_fully_outside':
  fixes p :: "real ⇒ 'a :: euclidean_space"
  assumes "path p" "does_not_cross p (frontier A)"
  shows   "path_image p ⊆ closure A ∨ path_image p ∩ interior A = {}"
  using path_fully_inside_or_fully_outside[OF assms(1), of A] assms un-
folding does_not_cross_def
  by auto

lemma in_path_image_part_circlepathI:
  assumes "y = x + rcis r u" "u ∈ closed_segment a b"
  shows   "y ∈ path_image (part_circlepath x r a b)"
proof -
  have "u ∈ path_image (linepath a b)"
    using assms by simp
  then obtain v where v: "v ∈ {0..1}" "u = linepath a b v"
    unfolding path_image_def by blast
  have "y = part_circlepath x r a b v"
    by (simp add: part_circlepath_altdef v assms)
  with v(1) show ?thesis
    unfolding path_image_def by blast
qed

lemma in_path_image_part_circlepathI':
  assumes "Arg (y - x) ∈ closed_segment a b" "dist y x = r"
  shows   "y ∈ path_image (part_circlepath x r a b)"
proof (rule in_path_image_part_circlepathI)
  show "y = x + rcis r (Arg (y - x))"
    using assms
    by (cases "x = y")
       (auto simp: rcis_def cis_Arg complex_sgn_def dist_commute scaleR_conv_of_real
                   field_simps dist_norm norm_minus_commute)
```

```
qed fact

lemma path_image_part_circlepath':
  "path_image (part_circlepath x r a b) = (λt. x + rcis r t) ‘ closed_segment
a b"
proof -
  have "path_image (part_circlepath x r a b) = (λt. x + rcis r t) ‘ path_image
(linepath a b)"
    by (simp add: path_image_def part_circlepath_altdef image_image)
  thus ?thesis
    by simp
qed

lemma path_image_part_circlepath:
  assumes "a ∈ {-pi<..pi}" "b ∈ {-pi<..pi}" "r > 0"
  shows   "path_image (part_circlepath x r a b) = {y∈sphere x r. Arg
(y - x) ∈ closed_segment a b}"
proof safe
  fix y assume y: "y ∈ path_image (part_circlepath x r a b)"
  show "y ∈ sphere x r"
    using y path_image_part_circlepath_subset'[of r x a b] assms by auto

  from y obtain u where u: "u ∈ {0..1}" "y = part_circlepath x r a b
u"
    by (auto simp: path_image_def)
  have "linepath a b u ∈ closed_segment a b"
    using linepath_in_path u(1) by blast
  also have "... ⊆ {-pi<..pi}"
    using assms by (intro closed_segment_subset) auto
  finally have *: "linepath a b u ∈ {-pi<..pi}" .

  have "Arg (y - x) = Arg (rcis r (linepath a b u))"
    by (simp add: u part_circlepath_altdef)
  also have "... = linepath a b u"
    using * assms by (subst Arg_rcis) auto
  also have "... ∈ closed_segment a b"
    by fact
  finally show "Arg (y - x) ∈ closed_segment a b" .
next
  fix y assume y: "y ∈ sphere x r" "Arg (y - x) ∈ closed_segment a b"
  show "y ∈ path_image (part_circlepath x r a b)"
    using y by (intro in_path_image_part_circlepathI') (auto simp: dist_commute)
qed

lemma path_image_part_circlepath_mono:
  assumes "min a' b' ≤ min a b" "max a' b' ≥ max a b"
  shows   "path_image (part_circlepath x r a b) ⊆ path_image (part_circlepath
x r a' b')"
proof -
```

```
  have "path_image (part_circlepath x r a b) = (λt. x + rcis r t) ‘ path_image
(linepath a b)"
    by (simp add: path_image_def part_circlepath_altdef image_image)
  also have "... ⊆ (λt. x + rcis r t) ‘ path_image (linepath a' b')"
    unfolding path_image_linepath using assms
    by (intro image_mono) (auto simp: closed_segment_eq_real_ivl split:
if_splits)
  also have "... = path_image (part_circlepath x r a' b')"
    by (simp add: path_image_def part_circlepath_altdef image_image)
  finally show ?thesis .
qed
```

3.7 Topology

```
lemma eventually_at_within_in_open:
  assumes "open X" "x ∈ X"
  shows   "eventually (λz. z ∈ X ∩ A - {x}) (at x within A)"
  using eventually_nhds_in_open[OF assms] unfolding eventually_at_filter
  by eventually_elim auto

lemma filterlim_at_withinD:
  assumes "filterlim f (at L within A) F" "open X" "L ∈ X"
  shows   "eventually (λx. f x ∈ X ∩ A - {L}) F"
proof -
  have "eventually (λz. z ∈ X ∩ A - {L}) (at L within A)"
    by (rule eventually_at_within_in_open) (use assms in auto)
  moreover from assms(1) have "filtermap f F ≤ at L within A"
    unfolding filterlim_def .
  ultimately have "eventually (λz. z ∈ X ∩ A - {L}) (filtermap f F)"
    unfolding le_filter_def by blast
  thus ?thesis
    by (simp add: eventually_filtermap)
qed

lemma filterlim_at_withinD':
  assumes "filterlim f (at L within A) F" "open X" "L ∈ X"
  shows   "eventually (λx. f x ∈ X ∩ A) F"
  using filterlim_at_withinD[OF assms] by eventually_elim auto

lemma filterlim_at_rightD:
  assumes "filterlim f (at_right L) F" "a > L"
  shows   "eventually (λx. f x ∈ {L<..<a}) F"
  using filterlim_at_withinD'[OF assms(1), of "{..<a}"] assms(2)
  by (auto elim!: eventually_mono)

lemma filterlim_at_leftD:
  assumes "filterlim f (at_left L) F" "a < L"
  shows   "eventually (λx. f x ∈ {a<..<L}) F"
  using filterlim_at_withinD'[OF assms(1), of "{a<..}"] assms(2)
```

by *(auto elim!: eventually_mono)*

lemma *eventually_Ball_at_right_0_real:*
assumes *"eventually P (at_right (0 :: real))"*
shows *"eventually (λx. ∀y∈{0<..x}. P y) (at_right 0)"*
using *assms* **unfolding** *eventually_at_right_field* **by** *force*

lemma *open_segment_same_Re:*
assumes *"Re a = Re b"*
shows *"open_segment a b = {z. Re z = Re a ∧ Im z ∈ open_segment (Im a) (Im b)}"*
using *assms* **by** *(auto simp: open_segment_def closed_segment_same_Re complex_eq_iff)*

lemma *open_segment_same_Im:*
assumes *"Im a = Im b"*
shows *"open_segment a b = {z. Im z = Im a ∧ Re z ∈ open_segment (Re a) (Re b)}"*
using *assms* **by** *(auto simp: open_segment_def closed_segment_same_Im complex_eq_iff)*

lemma *interior_halfspace_Re_ge [simp]: "interior {z. Re z ≥ x} = {z. Re z > x}"*
and *interior_halfspace_Re_le [simp]: "interior {z. Re z ≤ x} = {z. Re z < x}"*
and *interior_halfspace_Im_ge [simp]: "interior {z. Im z ≥ x} = {z. Im z > x}"*
and *interior_halfspace_Im_le [simp]: "interior {z. Im z ≤ x} = {z. Im z < x}"*
using *interior_halfspace_ge[of "1::complex" x] interior_halfspace_le[of "1::complex" x]*
interior_halfspace_ge[of i x] interior_halfspace_le[of i x]
by *(simp_all add: inner_complex_def)*

thm *arc_def*
thm *simple_path_def*

end

4 The Detour Calculus

theory *Detour_Calculus*
imports *"HOL-Complex_Analysis.Complex_Analysis" "Path_Automation.Path_Automation" Detour_Prerequisites*
begin

4.1 Local deformations of a path

locale *detour_rel_aux_locale =*
fixes *ε :: real* **and** *X :: "complex set"* **and** *p :: "real ⇒ complex"* **and** *p' :: "real ⇒ complex"*

```
  assumes ε_pos: "ε > 0"
  assumes p'_simple [simp, intro]: "simple_path p'"
  assumes p'_valid [simp, intro]: "valid_path p'"
  assumes X_subset: "X ⊆ path_image p"
  assumes X_disjoint: "X ∩ path_image p' = {}"
  assumes homotopic: "homotopic_paths (path_image p ∪ (⋃x∈X. cball
x ε)) p p'"
begin

lemma X_disjoint' [simp]: "x ∈ X ⟹ x ∉ path_image p'"
  and X_subset' [simp]:   "x ∈ X ⟹ x ∈ path_image p"
  using X_subset X_disjoint by auto

lemma p'_path [simp, intro]: "path p'"
  using p'_simple by (rule simple_path_imp_path)

lemma path_image_p': "path_image p' ⊆ path_image p ∪ (⋃x∈X. cball
x ε)"
  by (metis homotopic homotopic_paths_imp_subset)

lemma same_ends [simp]: "pathstart p' = pathstart p" "pathfinish p' =
pathfinish p"
  using homotopic by (simp_all add: homotopic_paths_imp_pathstart homotopic_paths_imp_pathf

lemma ends_not_in_X: "pathstart p ∉ X" "pathfinish p ∉ X"
  using X_subset X_disjoint
  by (metis X_disjoint' pathstart_in_path_image pathfinish_in_path_image
same_ends)+

lemma translate:
  "detour_rel_aux_locale ε ((+) a ‘ X) ((+) a ∘ p) ((+) a ∘ p')"
proof
  show "simple_path ((+) a ∘ p')"
    by (subst simple_path_translation_eq) (rule p'_simple)
  show "(+) a ‘ X ⊆ path_image ((+) a ∘ p)" and "(+) a ‘ X ∩ path_image
((+) a ∘ p') = {}"
    by (auto simp: path_image_compose)
  show "homotopic_paths (path_image ((+) a ∘ p) ∪ (⋃x∈(+) a ‘ X. cball
x ε)) ((+) a ∘ p) ((+) a ∘ p')"
    by (intro homotopic_paths_continuous_image[OF homotopic] continuous_intros)
       (auto simp: path_image_compose)
qed (use ε_pos in ‹auto simp: valid_path_translation_eq›)

lemma orthogonal_transformation:
  assumes f [intro]: "orthogonal_transformation f"
  shows   "detour_rel_aux_locale ε (f ‘ X) (f ∘ p) (f ∘ p')"
proof
  from f(1) have f': "linear f" "inj f"
    using orthogonal_transformation f orthogonal_transformation_inj by
```

```
blast+
  have [simp]: "f x = f y ⟷ x = y" for x y
    using f'(2) by (auto simp: inj_def)
  note [continuous_intros] = linear_continuous_on_compose[OF _ f'(1)]


  show "simple_path (f ∘ p')"
    by (subst simple_path_linear_image_eq[OF f'(1,2)]) (rule p'_simple)
  show "valid_path (f ∘ p')"
    by (intro linear_image_valid_path f') blast
  show "f ‘ X ⊆ path_image (f ∘ p)" and "f ‘ X ∩ path_image (f ∘ p')
= {}"
    by (auto simp: path_image_compose)
  show "homotopic_paths (path_image (f ∘ p) ∪ (⋃x∈f ‘ X. cball x ε))
(f ∘ p) (f ∘ p')"
  proof (rule homotopic_paths_continuous_image[OF homotopic] continuous_intros)
    have "f ‘ (path_image p ∪ (⋃x∈X. cball x ε)) = f ‘ path_image p
∪ (⋃x∈X. f ‘ cball x ε)"
      by blast
    also have "(λx. f ‘ cball x ε) = (λx. cball (f x) ε)"
      by (intro ext image_orthogonal_transformation_cball f)
    also have "(⋃x∈X. cball (f x) ε) = (⋃x∈f‘X. cball x ε)"
      by blast
    also have "f ‘ path_image p ∪ ... = path_image (f ∘ p) ∪ (⋃x∈f ‘
X. cball x ε)"
      by (simp add: path_image_compose)
    finally show "f ∈ path_image p ∪ (⋃x∈X. cball x ε) → path_image
(f ∘ p) ∪ (⋃x∈f ‘ X. cball x ε)"
      by auto
  qed (intro continuous_intros)
qed (use ε_pos in auto)

lemma rotate:
  assumes "norm z = 1"
  shows   "detour_rel_aux_locale ε ((*) z ‘ X) ((*) z ∘ p) ((*) z ∘ p')"
  by (rule orthogonal_transformation) (use assms in auto)

lemma analytic_image:
  assumes inj: "inj_on f (path_image p ∪ (⋃x∈X. cball x ε))"
  assumes ana: "f analytic_on (path_image p ∪ (⋃x∈X. cball x ε))"
  assumes cball_image: "⋀x. x ∈ X ⟹ f ‘ cball x ε ⊆ cball (f x) ε'"
  assumes "ε' > 0"
  shows   "detour_rel_aux_locale ε' (f ‘ X) (f ∘ p) (f ∘ p')"
proof
  show "ε' > 0"
    by fact
  show "valid_path (f ∘ p')"
    using path_image_p' by (intro valid_path_compose_analytic[OF _ ana])
auto
```

```
  have cont: "continuous_on (path_image p ∪ (⋃x∈X. cball x ε)) f"
    by (intro holomorphic_on_imp_continuous_on analytic_imp_holomorphic
ana)
  show "simple_path (f ∘ p')"
    using p'_simple path_image_p'
    by (intro simple_path_continuous_image inj_on_subset[OF inj] continuous_on_subset[OF
cont])
  have "inj_on f (path_image p')"
    by (rule inj_on_subset) (use inj path_image_p' in auto)
  show "f ‘ X ⊆ path_image (f ∘ p)" and "f ‘ X ∩ path_image (f ∘ p')
= {}"
    using path_image_p' inj X_subset by (fastforce simp: path_image_compose
inj_on_def)+
  show "homotopic_paths (path_image (f ∘ p) ∪ (⋃x∈f ‘ X. cball x ε'))
(f ∘ p) (f ∘ p')"
  proof (rule homotopic_paths_continuous_image[OF homotopic cont])
    have "f ‘ (path_image p ∪ (⋃x∈X. cball x ε)) = path_image (f ∘ p)
∪ (⋃x∈X. f ‘ cball x ε)"
      by (auto simp: path_image_compose)
    also have "(⋃x∈X. f ‘ cball x ε) ⊆ (⋃x∈f‘X. cball x ε')"
      using cball_image by fast
    finally show "f ∈ path_image p ∪ (⋃x∈X. cball x ε) →
        path_image (f ∘ p) ∪ (⋃x∈f ‘ X. cball x ε')"
      by blast
  qed
qed

lemma reverse [intro!]:
  "detour_rel_aux_locale ε X (reversepath p) (reversepath p')"
  by unfold_locales
     (use ε_pos homotopic in <auto simp: homotopic_paths_reversepath simple_path_reversepat

theorem winding_number_unchanged:
  assumes "z ∉ path_image p ∪ (⋃x∈X. cball x ε)"
  shows   "winding_number p' z = winding_number p z"
proof -
  from homotopic have "homotopic_paths (-{z}) p p'"
    by (rule homotopic_paths_subset) (use assms in auto)
  hence "winding_number p z = winding_number p' z"
    by (intro winding_number_homotopic_paths)
  thus ?thesis
    by simp
qed

lemma congI:
  assumes "p ≡p p2" "p' ≡p p'2" "valid_path p'2"
  shows   "detour_rel_aux_locale ε X p2 p'2"
proof
  note [simp] = assms(1,2)[THEN eq_paths_imp_path_image_eq, symmetric]
```

```
  show "simple_path p'2"
    using eq_paths_imp_simple_path_iff[of p' p'2] assms p'_simple by simp
  show "X ⊆ path_image p2"
    using X_subset by simp
  show "X ∩ path_image p'2 = {}"
    using X_disjoint by simp
  let ?S = "(path_image p ∪ (⋃x∈X. cball x ε))"
  have "homotopic_paths ?S p2 p"
    by (intro eq_paths_imp_homotopic) (use assms in ‹simp_all add: eq_paths_sym_iff›)
  also have "homotopic_paths ?S p p'"
    by (rule homotopic)
  also have "homotopic_paths ?S p' p'2" using path_image_p'
    by (intro eq_paths_imp_homotopic) (use assms in ‹simp_all add: eq_paths_sym_iff›)
  finally show "homotopic_paths (path_image p2 ∪ (⋃x∈X. cball x ε)) p2
p'2"
    by simp
qed (use ε_pos assms in auto)

lemma mono:
  assumes "ε ≤ ε'"
  shows   "detour_rel_aux_locale ε' X p p'"
  by standard
     (use assms ε_pos X_subset X_disjoint p'_simple
      in ‹auto intro!: homotopic_paths_subset[OF homotopic]›)

lemma cnj: "detour_rel_aux_locale ε (cnj ‘ X) (cnj ∘ p) (cnj ∘ p')"
  by unfold_locales
     (auto simp: valid_path_cnj ε_pos path_image_compose dist_cnj
           intro!: homotopic_paths_continuous_image [OF homotopic])

end

lemma detour_rel_aux_locale_refl [intro!]:
  "ε > 0 ⟹ simple_path p ⟹ valid_path p ⟹ detour_rel_aux_locale
ε {} p p"
  by unfold_locales (auto dest: simple_path_imp_path)

lemma eq_paths_imp_detour_rel_aux_locale:
  assumes "ε > 0" "simple_path p" "valid_path q" "eq_paths p q"
  shows   "detour_rel_aux_locale ε {} p q"
  by unfold_locales
     (use assms eq_paths_imp_simple_path_iff[OF assms(4)] in ‹auto intro!:
eq_paths_imp_homotopic›)


lemma detour_rel_aux_join_aux1:
  assumes setdist: "setdist_gt ε X1 (path_image p2)"
  shows "(⋃x∈X1. cball x ε) ∩ path_image p2 = {}"
  using setdist unfolding setdist_gt_def by force
```

```
lemma detour_rel_aux_join_aux2:
  assumes "detour_rel_aux_locale ε X2 p2 p2'"
  assumes setdist: "setdist_gt ε X1 (path_image p2)"
  shows   "X1 ∩ path_image p2' = {}"
proof safe
  fix x assume x: "x ∈ X1" "x ∈ path_image p2'"
  interpret p2: detour_rel_aux_locale ε X2 p2 p2'
    by fact
  from x(2) have "x ∈ path_image p2 ∪ (⋃y∈X2. cball y ε)"
    using p2.path_image_p' by blast
  thus "x ∈ {}"
  proof safe
    assume x': "x ∈ path_image p2"
    from x and p2.ε_pos have "x ∈ (⋃y∈X1. cball y ε)"
      by force
    with x' show "x ∈ {}"
      using detour_rel_aux_join_aux1[OF setdist] by blast
  next
    fix z assume z: "z ∈ X2" "x ∈ cball z ε"
    have "ε < dist x z"
      using z(1) x p2.X_subset by (intro setdist_gtD[OF setdist]) auto
    with z(2) show "x ∈ {}"
      by (auto simp: dist_commute)
  qed
qed

locale detour_rel_aux_locale_join =
  p1: detour_rel_aux_locale ε X1 p1 p1' +
  p2: detour_rel_aux_locale ε X2 p2 p2' for ε X1 p1 p1' X2 p2 p2' +
  assumes p12_simple: "simple_path (p1 +++ p2)"
  assumes pathfinish_p1 [simp]: "pathfinish p1 = pathstart p2"
  assumes setdist: "setdist_gt (2*ε) X1 (path_image p2)" "setdist_gt
(2*ε) X2 (path_image p1)"
begin

lemma cball_X1_inter_p2: "(⋃x∈X1. cball x ε) ∩ path_image p2 = {}"
  and cball_X2_inter_p1: "(⋃x∈X2. cball x ε) ∩ path_image p1 = {}"
  by (intro detour_rel_aux_join_aux1 setdist[THEN setdist_gt_mono]; use
p1.ε_pos in simp)+

lemma X1_inter_p2: "X1 ∩ path_image p2' = {}"
  and X2_inter_p1: "X2 ∩ path_image p1' = {}"
proof -
  show "X1 ∩ path_image p2' = {}"
  proof (rule detour_rel_aux_join_aux2)
    show "detour_rel_aux_locale ε X2 p2 p2'"
      by (rule p2.detour_rel_aux_locale_axioms)
    show "setdist_gt ε X1 (path_image p2)"
```

```
    by (intro setdist[THEN setdist_gt_mono]) (use p1.ε_pos in auto)
  qed
next
  show "X2 ∩ path_image p1' = {}"
  proof (rule detour_rel_aux_join_aux2)
    show "detour_rel_aux_locale ε X1 p1 p1'"
      by (rule p1.detour_rel_aux_locale_axioms)
    show "setdist_gt ε X2 (path_image p1)"
      by (intro setdist[THEN setdist_gt_mono]) (use p1.ε_pos in auto)
  qed
qed

lemma X1_inter_p2' [simp]: "x ∈ X1 ⟹ x ∉ path_image p2'"
  and X2_inter_p1' [simp]: "x ∈ X2 ⟹ x ∉ path_image p1'"
  using X1_inter_p2 X2_inter_p1 by blast+

lemma X1_X2_disjoint: "X1 ∩ X2 = {}"
proof -
  have "X1 ∩ X2 ⊆ (⋃x∈X1. cball x ε) ∩ path_image p2"
    using p1.ε_pos by (intro Int_mono) auto
  also have "... = {}"
    by (intro cball_X1_inter_p2)
  finally show ?thesis by blast
qed

lemma X1_X2_disjoint' [simp]: "x ∈ X1 ⟹ x ∉ X2"
  using X1_X2_disjoint by auto

lemma cball_X1_inter_cball_X2:
  assumes "x ∈ X1" "y ∈ X2"
  shows   "cball x ε ∩ cball y ε = {}"
proof safe
  fix z assume z: "z ∈ cball x ε" "z ∈ cball y ε"
  have "dist x y ≤ dist x z + dist y z"
    using dist_triangle2 by blast
  also from z have "dist x z + dist y z ≤ 2 * ε"
    by (auto simp: dist_commute)
  finally have "dist x y ≤ 2 * ε" .
  moreover have "dist x y > 2 * ε"
    by (intro setdist_gtD[OF setdist(1)]) (use z p2.X_subset assms in
auto)
  ultimately show "z ∈ {}"
    by simp
qed

lemma cball_X1_inter_cball_X2':
  shows   "(⋃y∈X1. cball y ε) ∩ (⋃y∈X2. cball y ε) = {}"
  using cball_X1_inter_cball_X2 by blast
```

```
sublocale p12: detour_rel_aux_locale ε "X1 ∪ X2" "p1 +++ p2" "p1' +++
p2'"
proof
  show "ε > 0"
    by (rule p1.ε_pos)

  have [simp]: "arc p1" "arc p2"
    using p12_simple by (elim simple_path_joinE; simp)+
  have [simp]: "arc p1'"
    by (metis ‹arc p1› arc_distinct_ends p1.p'_simple p1.same_ends simple_path_eq_arc)
  have [simp]: "arc p2'"
    by (metis ‹arc p2› arc_distinct_ends p2.p'_simple p2.same_ends simple_path_eq_arc)

  show "homotopic_paths (path_image (p1 +++ p2) ∪ (⋃x∈X1 ∪ X2. cball
x ε))
          (p1 +++ p2) (p1' +++ p2')"
    by (intro homotopic_paths_join homotopic_paths_subset[OF p1.homotopic]
             homotopic_paths_subset[OF p2.homotopic] Un_mono)
       (auto simp: path_image_join)

  show "(X1 ∪ X2) ∩ path_image (p1' +++ p2') = {}"
    by (auto simp: path_image_join)

  show "X1 ∪ X2 ⊆ path_image (p1 +++ p2)"
    using p1.X_subset p2.X_subset by (subst path_image_join) auto

  show "valid_path (p1' +++ p2')"
    by (intro valid_path_join) auto

  show "simple_path (p1' +++ p2')"
  proof (rule simple_path_joinI)
    show "path_image p1' ∩ path_image p2' ⊆
          insert (pathstart p2') (if pathstart p1' = pathfinish p2' then
{pathstart p1'} else {})"
    proof (intro subsetI)
      fix x assume "x ∈ path_image p1' ∩ path_image p2'"
      also have "... ⊆ (path_image p1 ∪ (⋃x∈X1. cball x ε)) ∩ (path_image
p2 ∪ (⋃x∈X2. cball x ε))"
        using p1.path_image_p' p2.path_image_p' by blast
      also have "... ⊆ path_image p1 ∩ path_image p2"
        using cball_X1_inter_p2 cball_X2_inter_p1 cball_X1_inter_cball_X2'
by fast
      also have "... ⊆ insert (pathstart p2) (if pathstart p1 = pathfinish
p2 then {pathstart p1} else {})"
        using p12_simple by (rule simple_path_joinE') auto
      finally show "x ∈ insert (pathstart p2') (if pathstart p1' = pathfinish
p2' then
                           {pathstart p1'} else {})"
        by (simp cong: if_cong)
```

```
    qed
  qed auto
qed

end



locale detour_rel_aux_loop = detour_rel_aux_locale +
  assumes simple_loop [simp, intro]: "simple_loop p"
begin

lemma loop [simp]: "pathfinish p = pathstart p"
  and p_simple [simp, intro]: "simple_path p"
  and p_path [simp, intro]: "path p"
  and p'_simple_loop [simp, intro]: "simple_loop p'"
  using simple_loop unfolding simple_loop_def by (auto intro: simple_path_imp_path)

theorem same_orientation:
  assumes "winding_number p z ≠ 0" "z ∉ path_image p ∪ (⋃x∈X. cball
x ε)"
  shows   "simple_loop_orientation p' = simple_loop_orientation p"
proof -
  from assms have "z ∈ inside (path_image p)"
    using simple_loop_winding_number_cases[of p z] by (auto split: if_splits)
  with assms have winding_number: "winding_number p z ∈ {-1, 1}"
    using simple_closed_path_winding_number_inside[of p] by auto
  have p'_image: "path_image p' ⊆ path_image p ∪ (⋃x∈X. cball x ε)"
    using homotopic_paths_imp_subset[OF homotopic] by auto
  from homotopic have "homotopic_paths (-{z}) p p'"
    by (rule homotopic_paths_subset) (use assms in auto)
  hence 1: "winding_number p z = winding_number p' z"
    by (intro winding_number_homotopic_paths)
  have 2: "simple_loop_orientation p = winding_number p z"
    by (intro simple_loop_orientation_eqI) (use assms winding_number in
auto)
  have 3: "simple_loop_orientation p' = winding_number p' z"
    using p'_image assms 1 winding_number by (intro simple_loop_orientation_eqI)
auto
  from 1 2 3 show ?thesis
    by (metis of_int_eq_iff)
qed

end
```

4.2 The left/right detour relation

```
locale detour_rel_locale =
  pl: detour_rel_aux_locale ε "L ∪ R" p pl +
```

```
  pr: detour_rel_aux_locale ε "L ∪ R" p pr
  for ε L R p pl pr +
  assumes L_closed [intro]: "closed L" and R_closed [intro]: "closed
R"
      and winding_number_L: "x ∈ L ⟹ winding_number pl x - winding_number
pr x = -1"
      and winding_number_R: "x ∈ R ⟹ winding_number pl x - winding_number
pr x = 1"
begin

lemma L_R_disjoint: "L ∩ R = {}"
  using winding_number_L winding_number_R by force

lemma ends_not_in_L: "pathstart p ∉ L" "pathfinish p ∉ L"
  and ends_not_in_R: "pathstart p ∉ R" "pathfinish p ∉ R"
  using pl.ends_not_in_X pr.ends_not_in_X by blast+

lemma ε_pos: "ε > 0"
  using pl.ε_pos .

lemma swap: "detour_rel_locale ε R L p pr pl"
proof -
  interpret pl': detour_rel_aux_locale ε "R ∪ L" p pr
    using pr.detour_rel_aux_locale_axioms by (simp add: Un_commute)
  interpret pr': detour_rel_aux_locale ε "R ∪ L" p pl
    using pl.detour_rel_aux_locale_axioms by (simp add: Un_commute)
  show ?thesis
  proof
    show "winding_number pr x - winding_number pl x = -1" if "x ∈ R"
for x
      using winding_number_R[of x] that by Groebner_Basis.algebra
    show "winding_number pr x - winding_number pl x = 1" if "x ∈ L" for
x
      using winding_number_L[of x] that by Groebner_Basis.algebra
  qed auto
qed

lemma reverse [intro!]:
  "detour_rel_locale ε R L (reversepath p) (reversepath pl) (reversepath
pr)"
proof -
  interpret revpl: detour_rel_aux_locale ε "R ∪ L" "reversepath p" "reversepath
pl"
    by (subst Un_commute, rule pl.reverse)
  interpret revpr: detour_rel_aux_locale ε "R ∪ L" "reversepath p" "reversepath
pr"
    by (subst Un_commute, rule pr.reverse)
  show ?thesis
  proof unfold_locales
```

```
    show "winding_number (reversepath pl) x - winding_number (reversepath
pr) x = -1"
      if "x ∈ R" for x
      using winding_number_R[OF that] that by (simp add: winding_number_reversepath
algebra_simps)
  next
    show "winding_number (reversepath pl) x - winding_number (reversepath
pr) x = 1"
      if "x ∈ L" for x
      using winding_number_L[OF that] that by (simp add: winding_number_reversepath
algebra_simps)
  qed auto
qed


lemma winding_preserving:
  assumes ana: "f analytic_on (path_image p ∪ (⋃x∈L∪R. cball x ε))"
  assumes "winding_preserving (path_image p ∪ (⋃x∈L∪R. cball x ε))
f (λx. x)"
  assumes cball_image: "⋀x. x ∈ L ∪ R ⟹ f ‘ cball x ε ⊆ cball (f
x) ε'"
  assumes "path p"
  assumes "ε' > 0"
  shows "detour_rel_locale ε' (f ‘ L) (f ‘ R) (f ∘ p) (f ∘ pl) (f ∘ pr)"
proof -
  interpret f: winding_preserving "path_image p ∪ (⋃x∈L∪R. cball x ε)"
f "λx. x"
    by fact
  have cont: "continuous_on (path_image p ∪ (⋃x∈L∪R. cball x ε)) f"
    by (intro holomorphic_on_imp_continuous_on analytic_imp_holomorphic
ana)

  have "detour_rel_aux_locale ε' (f ‘ (L ∪ R)) (f ∘ p) (f ∘ pl)"
    by (rule pl.analytic_image)
       (use cball_image ‹ε' > 0›
        in ‹auto intro!: analytic_on_subset[OF ana] inj_on_subset[OF
f.inj]›)
  then interpret pl': detour_rel_aux_locale ε' "f ‘ L ∪ f ‘ R" "f ∘ p"
"f ∘ pl"
    by (simp add: image_Un)

  have "detour_rel_aux_locale ε' (f ‘ (L ∪ R)) (f ∘ p) (f ∘ pr)"
    by (rule pr.analytic_image)
       (use cball_image ‹ε' > 0›
        in ‹auto intro!: analytic_on_subset[OF ana] inj_on_subset[OF
f.inj]›)
  then interpret pr': detour_rel_aux_locale ε' "f ‘ L ∪ f ‘ R" "f ∘ p"
"f ∘ pr"
    by (simp add: image_Un)
```

```
  have eq: "winding_number (f ∘ pl) (f x) - winding_number (f ∘ pr) (f
x) =
            winding_number pl x - winding_number pr x" if "x ∈ L ∪ R"
for x
  proof -
    have *: "f x ∉ path_image (f ∘ pr)" "f x ∉ path_image (f ∘ pl)"
      by (metis image_Un image_eqI pr'.X_disjoint' pl'.X_disjoint' that)+
    hence "winding_number (f ∘ pl) (f x) - winding_number (f ∘ pr) (f
x) =
          winding_number (f ∘ pl) (f x) + winding_number (reversepath
(f ∘ pr)) (f x)"
      by (subst winding_number_reversepath) auto
    also have "... = winding_number ((f ∘ pl) +++ reversepath (f ∘ pr))
(f x)"
      using * by (subst winding_number_join) auto
    also have "... = winding_number (f ∘ (pl +++ reversepath pr)) (f x)"
      by (simp add: path_compose_join path_compose_reversepath)
    also have "... = winding_number (pl +++ reversepath pr) x"
      using that pl.X_subset pl.path_image_p' pr.path_image_p'
      by (intro f.winding_number_eq) (auto simp: path_image_join)
    also have "... = winding_number pl x - winding_number pr x"
      using pr.X_disjoint pl.X_disjoint that
      by (simp add: winding_number_reversepath winding_number_join)
    finally show ?thesis .
  qed

  show ?thesis
  proof
    have "compact (path_image p)"
      using ‹path p› by auto
    hence "compact L" "compact R"
      using L_closed R_closed pl.X_subset by (metis compact_Int_closed
inf.absorb2 le_sup_iff)+
    hence "compact (f ‘ L)" "compact (f ‘ R)"
      by (intro compact_continuous_image[OF continuous_on_subset[OF cont]];
          use pl.X_subset in force)+
    thus "closed (f ‘ L)" "closed (f ‘ R)"
      by (blast intro: compact_imp_closed)+
  next
    show "winding_number (f ∘ pl) x - winding_number (f ∘ pr) x = -1"
if "x ∈ f ‘ L" for x
      using that winding_number_L eq by (auto simp: f.winding_number_eq)
  next
    show "winding_number (f ∘ pl) x - winding_number (f ∘ pr) x = 1"
if "x ∈ f ‘ R" for x
      using that winding_number_R eq by (auto simp: f.winding_number_eq)
  qed
qed
```

```
lemma winding_preserving_flip:
  assumes ana: "f analytic_on (path_image p ∪ (⋃x∈L∪R. cball x ε))"
  assumes "winding_preserving (path_image p ∪ (⋃x∈L∪R. cball x ε))
f (λx. -cnj x)"
  assumes cball_image: "⋀x. x ∈ L ∪ R ⟹ f ‘ cball x ε ⊆ cball (f
x) ε'"
  assumes "path p"
  assumes "ε' > 0"
  shows "detour_rel_locale ε' (f ‘ R) (f ‘ L) (f ∘ p) (f ∘ pl) (f ∘ pr)"
proof -
  interpret f: winding_preserving "path_image p ∪ (⋃x∈L∪R. cball x ε)"
f "λx. -cnj x"
    by fact
  have cont: "continuous_on (path_image p ∪ (⋃x∈L∪R. cball x ε)) f"
    by (intro holomorphic_on_imp_continuous_on analytic_imp_holomorphic
ana)

  have "detour_rel_aux_locale ε' (f ‘ (R ∪ L)) (f ∘ p) (f ∘ pl)"
    by (subst Un_commute, rule pl.analytic_image)
       (use cball_image ‹ε' > 0›
        in ‹auto intro!: analytic_on_subset[OF ana] inj_on_subset[OF
f.inj]›)
  then interpret pl': detour_rel_aux_locale ε' "f ‘ R ∪ f ‘ L" "f ∘ p"
"f ∘ pl"
    by (simp add: image_Un)

  have "detour_rel_aux_locale ε' (f ‘ (R ∪ L)) (f ∘ p) (f ∘ pr)"
    by (subst Un_commute, rule pr.analytic_image)
       (use cball_image ‹ε' > 0›
        in ‹auto intro!: analytic_on_subset[OF ana] inj_on_subset[OF
f.inj]›)
  then interpret pr': detour_rel_aux_locale ε' "f ‘ R ∪ f ‘ L" "f ∘ p"
"f ∘ pr"
    by (simp add: image_Un)

  have eq: "winding_number (f ∘ pl) (f x) - winding_number (f ∘ pr) (f
x) =
            -cnj (winding_number pl x - winding_number pr x)" if "x ∈
L ∪ R" for x
  proof -
    have *: "f x ∉ path_image (f ∘ pr)" "f x ∉ path_image (f ∘ pl)"
      by (metis Un_ac(3) image_Un image_eqI pl'.X_disjoint' pr'.X_disjoint'
that)+
    hence "winding_number (f ∘ pl) (f x) - winding_number (f ∘ pr) (f
x) =
          winding_number (f ∘ pl) (f x) + winding_number (reversepath
(f ∘ pr)) (f x)"
      by (subst winding_number_reversepath) auto
```

```
    also have "... = winding_number ((f ∘ pl) +++ reversepath (f ∘ pr))
(f x)"
      using * by (subst winding_number_join) auto
    also have "... = winding_number (f ∘ (pl +++ reversepath pr)) (f x)"
      by (simp add: path_compose_join path_compose_reversepath)
    also have "... = -cnj (winding_number (pl +++ reversepath pr) x)"
      using that pl.X_subset pl.path_image_p' pr.path_image_p'
      by (intro f.winding_number_eq) (auto simp: path_image_join)
    also have "winding_number (pl +++ reversepath pr) x =
               winding_number pl x - winding_number pr x"
      using pr.X_disjoint pl.X_disjoint that
      by (simp add: winding_number_reversepath winding_number_join)
    finally show ?thesis .
  qed

  show ?thesis
  proof
    have "compact (path_image p)"
      using ‹path p› by auto
    hence "compact L" "compact R"
      using L_closed R_closed pl.X_subset by (metis compact_Int_closed
inf.absorb2 le_sup_iff)+
    hence "compact (f ‘ L)" "compact (f ‘ R)"
      by (intro compact_continuous_image[OF continuous_on_subset[OF cont]];
          use pl.X_subset in force)+
    thus "closed (f ‘ L)" "closed (f ‘ R)"
      by (blast intro: compact_imp_closed)+
  next
    show "winding_number (f ∘ pl) x - winding_number (f ∘ pr) x = -1"
if "x ∈ f ‘ R" for x
    proof -
      from that obtain x' where x': "x' ∈ R" "x = f x'"
        by auto
      have "winding_number (f ∘ pl) x - winding_number (f ∘ pr) x =
             -cnj (winding_number pl x' - winding_number pr x')"
        unfolding x' using x' by (subst eq) auto
      also have "winding_number pl x' - winding_number pr x' = 1"
        by (rule winding_number_R) fact
      also have "-cnj 1 = -1"
        by simp
      finally show ?thesis .
    qed
  next
     show "winding_number (f ∘ pl) x - winding_number (f ∘ pr) x = 1"
if "x ∈ f ‘ L" for x
    proof -
      from that obtain x' where x': "x' ∈ L" "x = f x'"
        by auto
      have "winding_number (f ∘ pl) x - winding_number (f ∘ pr) x =
```

```
          -cnj (winding_number pl x' - winding_number pr x')"
      unfolding x' using x' by (subst eq) auto
    also have "winding_number pl x' - winding_number pr x' = -1"
      by (rule winding_number_L) fact
    also have "-cnj (-1) = 1"
      by simp
    finally show ?thesis .
  qed
 qed
qed

lemma congI:
 assumes "p ≡p p'" "pl ≡p pl'" "pr ≡p pr'" "valid_path pl'" "valid_path
pr'"
 shows   "detour_rel_locale ε L R p' pl' pr'"
proof -
 interpret pl': detour_rel_aux_locale ε "L ∪ R" p' pl'
   by (rule detour_rel_aux_locale.congI[OF _ assms(1,2)])
      (fact pl.detour_rel_aux_locale_axioms assms)+
 interpret pr': detour_rel_aux_locale ε "L ∪ R" p' pr'
   by (rule detour_rel_aux_locale.congI[OF _ assms(1,3)])
      (fact pr.detour_rel_aux_locale_axioms assms)+

 have [simp]: "winding_number pl' x = winding_number pl x" if "x ∈ L
∪ R" for x
   using that by (intro winding_number_homotopic_paths
                    eq_paths_imp_homotopic[OF eq_paths_sym] assms) auto
 have [simp]: "winding_number pr' x = winding_number pr x" if "x ∈ L
∪ R" for x
   using that by (intro winding_number_homotopic_paths
                    eq_paths_imp_homotopic[OF eq_paths_sym] assms) auto
 show ?thesis
 proof
   show "winding_number pl' x - winding_number pr' x = -1" if "x ∈ L"
for x
     using winding_number_L[OF that] that by simp
   show "winding_number pl' x - winding_number pr' x = 1" if "x ∈ R"
for x
     using winding_number_R[OF that] that by simp
 qed auto
qed

lemma mono:
 assumes "ε ≤ ε'"
 shows   "detour_rel_locale ε' L R p pl pr"
proof -
 interpret pl': detour_rel_aux_locale ε' "L ∪ R" p pl
   using pl.mono[OF assms] .
 interpret pr': detour_rel_aux_locale ε' "L ∪ R" p pr
```

```
    using pr.mono[OF assms] .
  show ?thesis
    by standard (auto simp: winding_number_L winding_number_R)
qed

lemma cnj: "detour_rel_locale ε (cnj ‘ R) (cnj ‘ L) (cnj ∘ p) (cnj ∘
pl) (cnj ∘ pr)"
proof -
  interpret pl': detour_rel_aux_locale ε "cnj ‘ R ∪ cnj ‘ L" "cnj ∘ p"
"cnj ∘ pl"
    unfolding image_Un [symmetric] Un_commute[of R] by (rule pl.cnj)
  interpret pr': detour_rel_aux_locale ε "cnj ‘ R ∪ cnj ‘ L" "cnj ∘ p"
"cnj ∘ pr"
    unfolding image_Un [symmetric] Un_commute[of R] by (rule pr.cnj)
  show ?thesis
  proof
    have eq: "winding_number (cnj ∘ pl) (cnj z) - winding_number (cnj
∘ pr) (cnj z) =
              -cnj (winding_number pl z - winding_number pr z)" if "z ∈
L ∪ R" for z
      by (subst (1 2) winding_number_cnj) (use that in auto)
    show "winding_number (cnj ∘ pl) z - winding_number (cnj ∘ pr) z =
-1" if "z ∈ cnj ‘ R" for z
      using that winding_number_R by (auto simp: eq)
    show "winding_number (cnj ∘ pl) z - winding_number (cnj ∘ pr) z =
1" if "z ∈ cnj ‘ L" for z
      using that winding_number_L by (auto simp: eq)
  qed (auto intro!: closed_injective_linear_image linear_cnj simp: inj_def)
qed

end


locale detour_rel_locale_join =
  p1: detour_rel_locale ε L1 R1 p1 pl1 pr1 +
  p2: detour_rel_locale ε L2 R2 p2 pl2 pr2 for ε L1 R1 p1 pl1 pr1 L2 R2
p2 pl2 pr2 +
  assumes p12_simple: "simple_path (p1 +++ p2)"
  assumes pathfinish_p1 [simp]: "pathfinish p1 = pathstart p2"
  assumes setdist: "setdist_gt (2*ε) (L1 ∪ R1) (path_image p2)"
                   "setdist_gt (2*ε) (L2 ∪ R2) (path_image p1)"
begin

sublocale pl: detour_rel_aux_locale_join ε "L1 ∪ R1" p1 pl1 "L2 ∪ R2"
p2 pl2
  by unfold_locales (use p12_simple setdist in ‹simp_all add: Un_ac›)

sublocale pr: detour_rel_aux_locale_join ε "L1 ∪ R1" p1 pr1 "L2 ∪ R2"
p2 pr2
```

```
  by unfold_locales (use p12_simple setdist in <simp_all add: Un_ac>)

sublocale p12: detour_rel_locale ε "L1 ∪ L2" "R1 ∪ R2" "p1 +++ p2" "pl1
+++ pl2" "pr1 +++ pr2"
proof -
  write winding_number ("ind")
  interpret pl': detour_rel_aux_locale ε "(L1 ∪ L2) ∪ (R1 ∪ R2)" "p1
+++ p2" "pl1 +++ pl2"
    using p1.p12.detour_rel_aux_locale_axioms by (simp add: Un_ac)

  interpret pr': detour_rel_aux_locale ε "(L1 ∪ L2) ∪ (R1 ∪ R2)" "p1
+++ p2" "pr1 +++ pr2"
    using pr.p12.detour_rel_aux_locale_axioms by (simp add: Un_ac)

  show "detour_rel_locale ε (L1 ∪ L2) (R1 ∪ R2) (p1 +++ p2) (pl1 +++
pl2) (pr1 +++ pr2)"
  proof
    show "closed (L1 ∪ L2)" "closed (R1 ∪ R2)"
      by auto

    show "ind (pl1 +++ pl2) x - ind (pr1 +++ pr2) x = -1" if "x ∈ L1
∪ L2" for x
      using that
    proof
      assume x: "x ∈ L1"
      have "x ∈ cball x ε" "x ∈ path_image p1"
        using p1.ε_pos p1.pl.X_subset x by auto
      with x have "x ∉ path_image p2 ∪ (⋃x∈L2 ∪ R2. cball x ε)"
        using p1.cball_X1_inter_cball_X2[of x] p1.cball_X1_inter_p2 by
blast
      thus ?thesis using p1.winding_number_L[of x]
        p2.pr.winding_number_unchanged[of x] p2.pl.winding_number_unchanged[of
x] x
        by (simp add: winding_number_join algebra_simps)
    next
      assume x: "x ∈ L2"
      have "x ∈ cball x ε" "x ∈ path_image p2"
        using p1.ε_pos p1.pl.X_subset x by auto
      with x have "x ∉ path_image p1 ∪ (⋃x∈L1 ∪ R1. cball x ε)"
        using p1.cball_X1_inter_cball_X2[of _ x] p1.cball_X2_inter_p1
by blast
      thus ?thesis using p2.winding_number_L[of x]
        p1.pr.winding_number_unchanged[of x] p1.pl.winding_number_unchanged[of
x] x
        by (simp add: winding_number_join algebra_simps)
    qed

    show "ind (pl1 +++ pl2) x - ind (pr1 +++ pr2) x = 1" if "x ∈ R1 ∪
R2" for x
```

```
      using that
    proof
      assume x: "x ∈ R1"
      have "x ∈ cball x ε" "x ∈ path_image p1"
        using p1.ε_pos p1.pl.X_subset x by auto
      with x have "x ∉ path_image p2 ∪ (⋃x∈L2 ∪ R2. cball x ε)"
        using p1.cball_X1_inter_cball_X2[of x] p1.cball_X1_inter_p2 by
blast
      thus ?thesis using p1.winding_number_R[of x]
        p2.pr.winding_number_unchanged[of x] p2.pl.winding_number_unchanged[of
x] x
        by (simp add: winding_number_join algebra_simps)
    next
      assume x: "x ∈ R2"
      have "x ∈ cball x ε" "x ∈ path_image p2"
        using p1.ε_pos p1.pl.X_subset x by auto
      with x have "x ∉ path_image p1 ∪ (⋃x∈L1 ∪ R1. cball x ε)"
        using p1.cball_X1_inter_cball_X2[of _ x] p1.cball_X2_inter_p1
by blast
      thus ?thesis using p2.winding_number_R[of x]
        p1.pr.winding_number_unchanged[of x] p1.pl.winding_number_unchanged[of
x] x
        by (simp add: winding_number_join algebra_simps)
    qed
  qed
qed

end


locale detour_rel_loop = detour_rel_locale +
  assumes simple_loop [simp, intro]: "simple_loop p"
  assumes nontrivial: "∃z. winding_number p z ≠ 0 ∧ z ∉ path_image
p ∪ (⋃x∈L∪R. cball x ε)"
begin

sublocale pl: detour_rel_aux_loop ε "L ∪ R" p pl
  by unfold_locales (fact simple_loop)

sublocale pr: detour_rel_aux_loop ε "L ∪ R" p pr
  by unfold_locales (fact simple_loop)

lemma same_orientation:
  "simple_loop_orientation pl = simple_loop_orientation p"
  "simple_loop_orientation pr = simple_loop_orientation p"
proof -
  from nontrivial obtain z
    where z: "winding_number p z ≠ 0" "z ∉ path_image p ∪ (⋃x∈L∪R.
cball x ε)"
```

```
    by auto
  show "simple_loop_orientation pl = simple_loop_orientation p"
    using pl.same_orientation[OF z] .
  show "simple_loop_orientation pr = simple_loop_orientation p"
    using pr.same_orientation[OF z] .
qed

lemma reverse_loop [intro!]:
  "detour_rel_loop ε R L (reversepath p) (reversepath pl) (reversepath
pr)"
proof -
  interpret rev: detour_rel_locale ε R L "reversepath p" "reversepath
pl" "reversepath pr"
    by (rule reverse)
  show ?thesis
  proof
    from nontrivial obtain z
      where z: "winding_number p z ≠ 0" "z ∉ path_image p ∪ (⋃x∈L∪R.
cball x ε)"
      by blast
    show "∃z. winding_number (reversepath p) z ≠ 0 ∧
              z ∉ path_image (reversepath p) ∪ (⋃x∈R ∪ L. cball x ε)"
      using z by (intro exI[of _ z]) (auto simp: winding_number_reversepath)
  qed auto
qed

theorem
  assumes x: "x ∈ L"
  shows winding_number_L_left:  "winding_number pl x = (if simple_loop_cw
p then -1 else 0)"
    and winding_number_L_right: "winding_number pr x = (if simple_loop_ccw
p then 1 else 0)"
    and inside_pl_L_iff:        "x ∈ inside (path_image pl) ⟷ simple_loop_cw
p"
    and inside_pr_L_iff:        "x ∈ inside (path_image pr) ⟷ simple_loop_ccw
p"
proof -
  have [simp]: "x ∉ path_image pl" "x ∉ path_image pr"
    using x by auto

  define ort where "ort ≡ simple_loop_orientation p"
  have ort: "simple_loop_orientation pl = ort" "simple_loop_orientation
pr = ort"
    unfolding ort_def by (fact same_orientation)+
  have ort_cases: "ort ∈ {-1, 1}"
    using simple_loop_orientation_cases[of p] by (auto simp: ort_def)

  have *: "winding_number pl x ∈ {0, ort}" "winding_number pr x ∈ {0,
ort}"
```

```
    using simple_loop_winding_number_cases[of pl x] simple_loop_winding_number_cases[of
pr x] ort
    by auto

  show l: "winding_number pl x = (if simple_loop_cw p then -1 else 0)"
   and r: "winding_number pr x = (if simple_loop_ccw p then 1 else 0)"
    using ort_cases using simple_path_not_cw_and_ccw[of p] * winding_number_L[OF
x]
     by (auto simp: ort_def simple_loop_orientation_def split: if_splits)

  show "x ∈ inside (path_image pl) ⟷ simple_loop_cw p"
   and "x ∈ inside (path_image pr) ⟷ simple_loop_ccw  p"
    using l r simple_loop_winding_number_cases[of pl x]
          simple_loop_winding_number_cases[of pr x] ort by (auto split:
if_splits)
qed

corollary
  assumes x: "x ∈ R"
  shows winding_number_R_left:  "winding_number pl x = (if simple_loop_ccw
p then 1 else 0)"
    and winding_number_R_right: "winding_number pr x = (if simple_loop_cw
p then -1 else 0)"
    and inside_pl_R_iff:        "x ∈ inside (path_image pl) ⟷ simple_loop_ccw
p"
   and  inside_pr_R_iff:        "x ∈ inside (path_image pr) ⟷ simple_loop_cw
p"
proof -
  interpret rev: detour_rel_loop ε R L "reversepath p" "reversepath pl"
"reversepath pr"
    by (rule reverse_loop)
  show "winding_number pl x = (if simple_loop_ccw p then 1 else 0)"
    using rev.winding_number_L_left[OF x] x
    by (auto simp: winding_number_reversepath minus_equation_iff)
  show "winding_number pr x = (if simple_loop_cw p then -1 else 0)"
    using rev.winding_number_L_right[OF x] x
    by (auto simp: winding_number_reversepath minus_equation_iff)
  show "x ∈ inside (path_image pl) ⟷ simple_loop_ccw p"
   and "x ∈ inside (path_image pr) ⟷ simple_loop_cw  p"
    using rev.inside_pl_L_iff[OF x] rev.inside_pr_L_iff[OF x] by auto
qed

end


lemma detour_rel_locale_swap: "detour_rel_locale ε L R p pl pr ⟷ detour_rel_locale
ε R L p pr pl"
  using detour_rel_locale.swap by blast
```

```
lemma eq_paths_imp_detour_rel_locale:
  assumes "ε > 0" "simple_path p" "eq_paths p pl" "eq_paths p pr" "valid_path
pl" "valid_path pr"
  shows   "detour_rel_locale ε {} {} p pl pr"
proof -
  interpret pl: detour_rel_aux_locale ε "{} ∪ {}" p pl
    using assms by (auto intro!: eq_paths_imp_detour_rel_aux_locale)
  interpret pr: detour_rel_aux_locale ε "{} ∪ {}" p pr
    using assms by (auto intro!: eq_paths_imp_detour_rel_aux_locale)
  show ?thesis
    by standard auto
qed

lemma detour_rel_localeI [intro?]:
  assumes "detour_rel_aux_locale ε (L ∪ R) p pl" "detour_rel_aux_locale
ε (L ∪ R) p pr"
          "closed L" "closed R"
          "⋀x. x ∈ L ⟹ winding_number pl x - winding_number pr x =
-1"
          "⋀x. x ∈ R ⟹ winding_number pl x - winding_number pr x =
1"
  shows "detour_rel_locale ε L R p pl pr"
proof -
  interpret pl: detour_rel_aux_locale ε "L ∪ R" p pl by fact
  interpret pr: detour_rel_aux_locale ε "L ∪ R" p pr by fact
  show ?thesis using assms(3-)
    by unfold_locales auto
qed



definition detour_rel where
  "detour_rel L R p pl pr ⟷
     (simple_path p ∧ valid_path p ⟶
      eventually (λε. detour_rel_locale ε L R p (pl ε) (pr ε)) (at_right
0))"

lemma detour_rel_congI:
  assumes "detour_rel L R p pl pr" "p ≡ₚ p'" "valid_path p" "L = L'"
"R = R'"
          "eventually (λε. pl ε ≡ₚ pl' ε) (at_right 0)"
          "eventually (λε. pr ε ≡ₚ pr' ε) (at_right 0)"
          "eventually (λε. valid_path (pl' ε)) (at_right 0)"
          "eventually (λε. valid_path (pr' ε)) (at_right 0)"
  shows   "detour_rel L' R' p' pl' pr'"
  unfolding detour_rel_def
proof safe
  assume "simple_path p'" "valid_path p'"
  hence "simple_path p"
```

```
    using assms(2) eq_paths_imp_simple_path_iff by blast
  with assms have "eventually (λε. detour_rel_locale ε L R p (pl ε) (pr
ε)) (at_right 0)"
    by (auto simp: detour_rel_def)
  thus "eventually (λε. detour_rel_locale ε L' R' p' (pl' ε) (pr' ε))
(at_right 0)"
    using assms(6-9)
  proof eventually_elim
    case (elim ε)
    show ?case
      using detour_rel_locale.congI[OF elim(1) assms(2) elim(2-5)] assms
by auto
  qed
qed

lemma detour_rel_eq_paths_trans [trans]:
  assumes "p ≡p p'" "detour_rel L R p' pl pr" "valid_path p'"
  shows   "detour_rel L R p pl pr"
  unfolding detour_rel_def
proof safe
  assume "simple_path p" "valid_path p"
  with assms(1) have "simple_path p'"
    by (simp add: eq_paths_imp_simple_path_iff)
  with assms have "eventually (λε. detour_rel_locale ε L R p' (pl ε)
(pr ε)) (at_right 0)"
    by (auto simp: detour_rel_def)
  thus "eventually (λε. detour_rel_locale ε L R p (pl ε) (pr ε)) (at_right
0)"
  proof eventually_elim
    case (elim ε)
    interpret detour_rel_locale ε L R p' "pl ε" "pr ε"
      by fact
    show ?case using assms
      by (intro congI) (auto simp: eq_paths_sym_iff)
  qed
qed

lemma detour_rel_eq_paths_trans' [trans]:
  assumes "detour_rel L R p pl pr" "p ≡p p'" "valid_path p"
  shows   "detour_rel L R p' pl pr"
  using detour_rel_eq_paths_trans[of p' p L R pl pr] assms
  by (simp add: eq_paths_sym_iff)

lemma detour_rel_imp_valid_simple_path:
  assumes "detour_rel L R p pl pr" "simple_path p" "valid_path p"
  shows   "eventually (λε. simple_path (pl ε) ∧ simple_path (pr ε) ∧

             valid_path (pl ε) ∧ valid_path (pr ε)) (at_right 0)"
proof -
```

```
  from assms have "eventually (λε. detour_rel_locale ε L R p (pl ε) (pr
ε)) (at_right 0)"
    by (auto simp: detour_rel_def)
  thus ?thesis
  proof eventually_elim
    case (elim ε)
    then interpret detour_rel_locale ε L R p "pl ε" "pr ε" .
    show ?case
      by blast
  qed
qed
```

4.3 Inference rules

```
lemma detour_rel_image:
  assumes ind: "winding_preserving A f (λx. x)"
  assumes ana: "f analytic_on A"
  assumes lipschitz: "M-lipschitz_on A f" "M > 0"
  assumes rel: "detour_rel L R p pl pr"
  assumes p: "valid_path (f ∘ p) ⟹ valid_path p" "path_image p ⊆ interior
A" and "closed A"
  shows   "detour_rel (f ‘ L) (f ‘ R) (f ∘ p) (λε. f ∘ pl (ε / M)) (λε.
f ∘ pr (ε / M))"
  unfolding detour_rel_def
proof safe
  assume "simple_path (f ∘ p)" "valid_path (f ∘ p)"
  hence [simp]: "valid_path p"
    using p by (auto dest: simple_path_imp_path)
  hence [simp]: "path p"
    by (rule valid_path_imp_path)
  interpret f: winding_preserving A f "λx. x"
    by fact
  have cont: "continuous_on A f"
    using lipschitz by (simp add: lipschitz_on_continuous_on)
  have "simple_path p"
    using ‹simple_path (f ∘ p)› by (auto simp: simple_path_def loop_free_def)

  have ev: "∀F ε in at_right 0. detour_rel_locale ε L R p (pl ε) (pr
ε)"
    using ‹simple_path p› ‹valid_path p› rel unfolding detour_rel_def
by blast
  then obtain ε0 :: real where "detour_rel_locale ε0 L R p (pl ε0) (pr
ε0)"
    by fastforce
  interpret eps0: detour_rel_locale ε0 L R p "pl ε0" "pr ε0"
    by fact

  have "eventually (λε. (⋃x∈path_image p. cball x ε) ⊆ A) (at_right
0)"
```

```
    by (rule eventually_path_image_cball_subset) (use assms in auto)
  moreover have "eventually (λε. ε > 0) (at_right (0 :: real))"
    by (simp add: eventually_at_right_less)
  ultimately have "∀F ε in at_right 0.
                    detour_rel_locale (M * ε) (f ‘ L) (f ‘ R) (f ∘ p)
(f ∘ pl ε) (f ∘ pr ε)"
    using ev
  proof eventually_elim
    case (elim ε)
    interpret detour_rel_locale ε L R p "pl ε" "pr ε"
      by fact
    show "detour_rel_locale (M * ε) (f ‘ L) (f ‘ R) (f ∘ p) (f ∘ pl ε)
(f ∘ pr ε)"
    proof (rule winding_preserving)
      fix x assume x: "x ∈ L ∪ R"
      show "f ‘ cball x ε ⊆ cball (f x) (M * ε)"
      proof safe
        fix u assume u: "u ∈ cball x ε"
        have "u ∈ A"
          using elim u x pl.X_subset by blast
        hence "dist (f u) (f x) ≤ M * dist u x"
          by (intro lipschitz_onD[OF lipschitz(1)])
             (use u x elim pl.X_subset p interior_subset in blast)+
        also have "... ≤ M * ε"
          by (intro mult_left_mono) (use u elim lipschitz(2) in ‹auto
simp: dist_commute›)
        finally show "f u ∈ cball (f x) (M * ε)"
          by (simp add: dist_commute)
      qed
    next
      show "f analytic_on (path_image p ∪ (⋃x∈L ∪ R. cball x ε))"
        by (intro analytic_on_subset[OF ana]) (use elim(1,2) in fastforce)+
    next
      show "winding_preserving (path_image p ∪ (⋃x∈L ∪ R. cball x ε))
f (λx. x)"
        by (rule winding_preserving_subset[OF ind])
           (use elim(1,2) in fastforce)
    qed (use ‹path p› ‹M > 0› ε_pos in auto)
  qed
  also have "at_right 0 = filtermap ((*) (1 / M)) (at_right 0)"
    using filtermap_times_pos_at_right[of "1/M" 0] ‹M > 0› by simp
  finally show "∀F ε in at_right 0. detour_rel_locale ε
                                    (f ‘ L) (f ‘ R) (f ∘ p) (f ∘ pl (ε
/ M)) (f ∘ pr (ε / M))"
    using ‹M > 0› by (simp add: eventually_filtermap o_def)
qed

lemma detour_rel_mult:
  assumes "detour_rel L R p pl pr" "c ≠ 0"
```

```
  shows   "detour_rel ((*) c ‘ L) ((*) c ‘ R) ((*) c ∘ p)
             (λε. (*) c ∘ pl (ε / norm c)) (λε. (*) c ∘ pr (ε / norm c))"
proof (rule detour_rel_image)
  show "winding_preserving UNIV ((*) c) (λx. x)"
    by (rule winding_preserving_mult) fact
  show "(cmod c)-lipschitz_on UNIV ((*) c)"
    by (rule lipschitz_intros)+ auto
  show "valid_path p" if "valid_path ((*) c ∘ p)"
  proof -
    from that have "valid_path ((*) (inverse c) ∘ ((*) c ∘ p))"
      by (rule valid_path_compose_analytic) auto
    also have "... = p"
      using ‹c ≠ 0› by (simp add: fun_eq_iff)
    finally show "valid_path p" .
  qed
qed (use assms in auto)

lemma detour_rel_uminus:
  assumes "detour_rel L R p pl pr"
  shows   "detour_rel ((λx. -x) ‘ L) ((λx. -x) ‘ R) ((λx. -x) ∘ p)
             (λε. (λx. -x) ∘ pl ε) (λε. (λx. -x) ∘ pr ε)"
  using detour_rel_mult[OF assms, of "-1"] by (simp add: o_def)

lemma detour_rel_cnj:
  assumes "detour_rel L R p pl pr"
  shows   "detour_rel (cnj ‘ R) (cnj ‘ L) (cnj ∘ p) (λε. cnj ∘ pl ε)
(λε. cnj ∘ pr ε)"
  unfolding detour_rel_def
proof safe
  assume "simple_path (cnj ∘ p)" "valid_path (cnj ∘ p)"
  with assms have "eventually (λε. detour_rel_locale ε L R p (pl ε) (pr
ε)) (at_right 0)"
    by (simp add: detour_rel_def valid_path_cnj)
  thus "eventually (λε. detour_rel_locale ε (cnj ‘ R) (cnj ‘ L)
          (cnj ∘ p) (cnj ∘ pl ε) (cnj ∘ pr ε)) (at_right 0)"
  proof eventually_elim
    case (elim ε)
    thus ?case by (rule detour_rel_locale.cnj)
  qed
qed

lemma detour_rel_translate:
  assumes "detour_rel L R p pl pr"
  shows   "detour_rel ((+) c ‘ L) ((+) c ‘ R) ((+) c ∘ p)  (λε. (+) c
∘ pl ε) (λε. (+) c ∘ pr ε)"
proof -
  have "detour_rel ((+) c ‘ L) ((+) c ‘ R) ((+) c ∘ p)  (λε. (+) c ∘
pl (ε / 1)) (λε. (+) c ∘ pr (ε / 1))"
  proof (rule detour_rel_image)
```

```
    show "winding_preserving UNIV ((+) c) (λx. x)"
      by (rule winding_preserving_translate)
    show "1-lipschitz_on UNIV ((+) c)"
      by (rule lipschitz_intros)+ auto?
    show "valid_path p" if "valid_path ((+) c ∘ p)"
      using that by (simp add: valid_path_translation_eq)
  qed (use assms in ‹auto intro!: analytic_intros›)
  thus ?thesis
    by simp
qed

lemma detour_relI:
  assumes "eps > 0" "closed L" "closed R"
  assumes "⋀ε. ε > 0 ⟹ ε < eps ⟹ simple_path p ⟹ valid_path p
⟹ detour_rel_aux_locale ε (L ∪ R) p (pl ε)"
  assumes "⋀ε. ε > 0 ⟹ ε < eps ⟹ simple_path p ⟹ valid_path p⟹
detour_rel_aux_locale ε (L ∪ R) p (pr ε)"
  assumes "⋀ε x. ε > 0 ⟹ ε < eps ⟹ x ∈ L ⟹ simple_path p ⟹
valid_path p⟹
            winding_number (pl ε) x - winding_number (pr ε) x = -1"
  assumes "⋀ε x. ε > 0 ⟹ ε < eps ⟹ x ∈ R ⟹ simple_path p ⟹
valid_path p⟹
            winding_number (pl ε) x - winding_number (pr ε) x = 1"
  shows   "detour_rel L R p pl pr"
  unfolding detour_rel_def eventually_at
proof (intro impI exI[of _ eps], safe)
  fix ε :: real assume ε: "ε > 0" "dist ε 0 < eps"
  assume simple: "simple_path p" and valid: "valid_path p"
  interpret pl: detour_rel_aux_locale ε "L ∪ R" p "pl ε"
    by (rule assms) (use ε simple valid in auto)
  interpret pr: detour_rel_aux_locale ε "L ∪ R" p "pr ε"
    by (rule assms) (use ε simple valid in auto)
  show "detour_rel_locale ε L R p (pl ε) (pr ε)"
    by unfold_locales (use ε assms(1,2,3,6,7) simple valid in auto)
qed (rule ‹eps > 0›)

lemma detour_rel_swap: "detour_rel L R p pl pr ⟹ detour_rel R L p
pr pl"
  unfolding detour_rel_def by (simp add: detour_rel_locale_swap)

lemma detour_rel_swap_iff: "detour_rel L R p pl pr ⟷ detour_rel R
L p pr pl"
  unfolding detour_rel_def by (simp add: detour_rel_locale_swap)


named_theorems detour_rel_intros

lemma detour_rel_refl [detour_rel_intros, simp, intro!]: "detour_rel
{} {} p (λ_. p) (λ_. p)"
```

```
  unfolding detour_rel_def using eventually_at_right_less[of 0]
  by unfold_locales
     (auto simp: simple_path_imp_path
           intro!: eventually_mono[OF eventually_at_right_less[of 0]]
detour_rel_localeI)

lemma detour_rel_rescale:
  assumes "detour_rel L R p pl pr" "c ≥ 1"
  shows   "detour_rel L R p (λε. pl (ε / c)) (λε. pr (ε / c))"
  unfolding detour_rel_def
proof safe
  assume "simple_path p" "valid_path p"
  with assms have "eventually (λε. detour_rel_locale ε L R p (pl ε) (pr
ε)) (at_right 0)"
    by (auto simp: detour_rel_def)
  also have "at_right 0 = filtermap ((*) (inverse c)) (at_right 0)"
    using ‹c ≥ 1› by (simp add: filtermap_times_pos_at_right)
  finally show "eventually (λε. detour_rel_locale ε L R p (pl (ε / c))
(pr (ε / c))) (at_right 0)"
    unfolding eventually_filtermap using eventually_at_right_less[of "0
:: real"]
  proof eventually_elim
    case (elim ε)
    have "inverse c * ε ≤ 1 * ε"
      using elim assms by (intro mult_right_mono) (auto simp: field_simps)
    hence "detour_rel_locale ε L R p (pl (inverse c * ε)) (pr (inverse
c * ε))"
      by (intro detour_rel_locale.mono [OF elim(1)]) auto
    thus ?case
      by (simp add: field_simps)
  qed
qed

lemma eq_paths_imp_detour_rel:
  assumes "eq_paths p pl" "eq_paths p pr" "valid_path pl" "valid_path
pr"
  shows   "detour_rel {} {} p (λ_. pl) (λ_. pr)"
proof -
  have "eventually (λε::real. ε > 0) (at_right 0)"
    by (simp add: eventually_at_right_less)
  hence "eventually (λε. detour_rel_locale ε {} {} p pl pr) (at_right
0)" if "simple_path p" "valid_path p"
    by eventually_elim (use assms that in ‹auto intro!: eq_paths_imp_detour_rel_locale›)
  thus ?thesis
    by (auto simp: detour_rel_def)
qed

lemma detour_rel_reverse [detour_rel_intros, intro!]:
  assumes "detour_rel L R p pl pr"
```

```
  shows   "detour_rel R L (reversepath p) (λε. reversepath (pl ε)) (λε.
reversepath (pr ε))"
  unfolding detour_rel_def
proof safe
  assume "simple_path (reversepath p)" "valid_path (reversepath p)"
  with assms have "eventually (λε. detour_rel_locale ε L R p (pl ε) (pr
ε)) (at_right 0)"
    by (auto simp: detour_rel_def simple_path_reversepath_iff)
  thus "eventually (λε. detour_rel_locale ε R L (reversepath p)
                          (reversepath (pl ε)) (reversepath (pr ε))) (at_right
0)"
    by eventually_elim (auto intro!: detour_rel_locale.reverse)
qed

lemma detour_rel_join [detour_rel_intros, intro?]:
  assumes "detour_rel L1 R1 p1 pl1 pr1"
  assumes "detour_rel L2 R2 p2 pl2 pr2"
  assumes [simp]: "pathfinish p1 = pathstart p2"
  shows   "detour_rel (L1 ∪ L2) (R1 ∪ R2) (p1 +++ p2) (λε. pl1 ε +++
pl2 ε) (λε. pr1 ε +++ pr2 ε)"
  unfolding detour_rel_def
proof safe
  assume simple: "simple_path (p1 +++ p2)" and valid: "valid_path (p1
+++ p2)"
  have arc [intro, simp]: "arc p1" "arc p2"
    using simple_path_joinE[OF simple] by auto
  hence simple' [intro, simp]: "simple_path p1" "simple_path p2"
    by (auto intro: arc_imp_simple_path)
  hence path' [intro, simp]: "path p1" "path p2"
    by (auto intro: simple_path_imp_path)
  have [intro, simp]: "valid_path p1" "valid_path p2"
    using valid by (auto dest!: valid_path_join_D1)

  from assms(1) obtain ε0 where "detour_rel_locale ε0 L1 R1 p1 (pl1 ε0)
(pr1 ε0)"
    unfolding detour_rel_def by fastforce
  then interpret p1: detour_rel_locale ε0 L1 R1 p1 "pl1 ε0" "pr1 ε0" .
  from assms(2) obtain ε0' where "detour_rel_locale ε0' L2 R2 p2 (pl2
ε0') (pr2 ε0')"
    unfolding detour_rel_def by fastforce
  then interpret p2: detour_rel_locale ε0' L2 R2 p2 "pl2 ε0'" "pr2 ε0'"
.

  have p1_LR2: False if x: "x ∈ path_image p1" "x ∈ L2 ∪ R2" for x
  proof -
    have "x ∈ path_image p2"
      using x(2) by auto
    hence "x = pathstart p2 ∨ x = pathfinish p2 ∧ pathstart p1 = pathfinish
p2"
```

```
    using x(1) using simple_path_joinE''[OF simple, of x] by simp
  thus False
    using x(2) p2.ends_not_in_L p2.ends_not_in_R by auto
qed

have p2_LR1: False if x: "x ∈ path_image p2" "x ∈ L1 ∪ R1" for x
proof -
  have x': "x ∈ path_image p1"
    using x(2) by auto
  hence "x = pathstart p2 ∨ x = pathfinish p2 ∧ pathstart p1 = pathfinish
p2"
    using x(1) using simple_path_joinE''[OF simple, of x] by simp
  thus False
    using x(2) p1.ends_not_in_L p1.ends_not_in_R by auto
qed

have "(2 :: real) > 0"
  by simp
note ev_lift = eventually_setdist_gt_at_right_0_mult_iff[OF this, symmetric]

have "eventually (λε. setdist_gt ε (L1 ∪ R1) (path_image p2)) (at_right
0)"
proof (subst setdist_gt_sym, rule compact_closed_imp_eventually_setdist_gt_at_right_0)
  show "compact (path_image p2)"
    by (simp add: compact_arc_image)
  show "path_image p2 ∩ (L1 ∪ R1) = {}"
    using p2_LR1 by blast
qed (auto intro: finite_imp_closed)

moreover have "eventually (λε. setdist_gt ε (L2 ∪ R2) (path_image
p1)) (at_right 0)"
proof (subst setdist_gt_sym, rule compact_closed_imp_eventually_setdist_gt_at_right_0)
  show "compact (path_image p1)"
    by (simp add: compact_arc_image)
  show "path_image p1 ∩ (L2 ∪ R2) = {}"
          using p1_LR2 by blast
      qed (auto intro: finite_imp_closed)

moreover have "eventually (λε. detour_rel_locale ε L1 R1 p1 (pl1 ε)
(pr1 ε)) (at_right 0)"
              "eventually (λε. detour_rel_locale ε L2 R2 p2 (pl2 ε)
(pr2 ε)) (at_right 0)"
  using assms(1,2) unfolding detour_rel_def by auto
ultimately show "eventually (λε. detour_rel_locale ε (L1 ∪ L2) (R1
∪ R2)
                    (p1 +++ p2) (pl1 ε +++ pl2 ε) (pr1 ε +++ pr2 ε))
(at_right 0)"
  unfolding ev_lift
proof eventually_elim
```

```
    case (elim ε)
    interpret p1: detour_rel_locale ε L1 R1 p1 "pl1 ε" "pr1 ε"
      by fact
    interpret p2: detour_rel_locale ε L2 R2 p2 "pl2 ε" "pr2 ε"
      by fact
    interpret detour_rel_locale_join ε L1 R1 p1 "pl1 ε" "pr1 ε" L2 R2
p2 "pl2 ε" "pr2 ε"
      by unfold_locales (use elim in ‹simp_all add: simple›)
    show ?case
      using p12.detour_rel_locale_axioms .
  qed
qed
```

4.4 Basic avoidance patterns

4.4.1 Generic helper lemmas

```
lemma detour_rel_avoid_basic:
  fixes p :: "real ⇒ complex" and L R :: "complex set"
  defines "S ≡ (λε. path_image p ∪ (⋃y∈L∪R. cball y ε))"
  assumes simple: "⋀ε. ε > 0 ⟹ ε < eps ⟹ simple_path p ⟹ simple_path
(p1 ε +++ pl ε +++ p2 ε)"
                  "⋀ε. ε > 0 ⟹ ε < eps ⟹ simple_path p ⟹ simple_path
(p1 ε +++ pr ε +++ p2 ε)"
  assumes LR_subset: "simple_path p ⟹ L ∪ R ⊆ path_image p"
  assumes homo: "⋀ε. ε > 0 ⟹ ε < eps ⟹ simple_path p ⟹ homotopic_paths
(S ε) p (p1 ε +++ pl ε +++ p2 ε)"
                "⋀ε. ε > 0 ⟹ ε < eps ⟹ simple_path p ⟹ homotopic_paths
(S ε) p (p1 ε +++ pr ε +++ p2 ε)"
  assumes ind: "⋀ε x. ε > 0 ⟹ ε < eps ⟹ simple_path p ⟹ x ∈ L
⟹ winding_number (pl ε +++ reversepath (pr ε)) x = -1"
               "⋀ε x. ε > 0 ⟹ ε < eps ⟹ simple_path p ⟹ x ∈ R
⟹ winding_number (pl ε +++ reversepath (pr ε)) x = 1"
  assumes ends: "⋀ε. ε > 0 ⟹ ε < eps ⟹ simple_path p ⟹ pathfinish
(pl ε) = pathstart (p2 ε)"
                "⋀ε. ε > 0 ⟹ ε < eps ⟹ simple_path p ⟹ pathfinish
(pr ε) = pathstart (p2 ε)"
                "⋀ε. ε > 0 ⟹ ε < eps ⟹ simple_path p ⟹ pathstart
(pl ε) = pathfinish (p1 ε)"
                "⋀ε. ε > 0 ⟹ ε < eps ⟹ simple_path p ⟹ pathstart
(pr ε) = pathfinish (p1 ε)"
  assumes LR: "⋀ε. ε > 0 ⟹ ε < eps ⟹ simple_path p ⟹ path_image
(pl ε) ∩ (L ∪ R) = {}"
              "⋀ε. ε > 0 ⟹ ε < eps ⟹ simple_path p ⟹ path_image
(pr ε) ∩ (L ∪ R) = {}"
              "⋀ε. ε > 0 ⟹ ε < eps ⟹ simple_path p ⟹ path_image
(p1 ε) ∩ (L ∪ R) = {}"
              "⋀ε. ε > 0 ⟹ ε < eps ⟹ simple_path p ⟹ path_image
(p2 ε) ∩ (L ∪ R) = {}"
  assumes valid: "⋀ε. ε > 0 ⟹ ε < eps ⟹ valid_path p ⟹ valid_path
```

```
(pl ε)"
                "⋀ε. ε > 0 ⟹ ε < eps ⟹ valid_path p ⟹ valid_path
(pr ε)"
                "⋀ε. ε > 0 ⟹ ε < eps ⟹ valid_path p ⟹ valid_path
(p1 ε)"
                "⋀ε. ε > 0 ⟹ ε < eps ⟹ valid_path p ⟹ valid_path
(p2 ε)"
  assumes "eps > 0" and closed: "closed L" "closed R"
  shows   "detour_rel L R p (λε. p1 ε +++ pl ε +++ p2 ε) (λε. p1 ε +++
pr ε +++ p2 ε)"
proof (rule detour_relI[OF <eps > 0>])
  fix ε assume ε: "ε > 0" "ε < eps"
  assume simple_p: "simple_path p"
  hence [simp]: "path p"
    by (auto dest: simple_path_imp_path)
  assume valid_p: "valid_path p"
  note LR' = LR[OF ε]
  note simple' = simple[OF ε]
  note homo' = homo[OF ε]
  note valid = valid[OF ε valid_p]
  note [simp] = ends[OF ε]
  have [simp]: "path (pl ε)" "path (pr ε)" "path (p1 ε)" "path (p2 ε)"
    using simple' simple_path_imp_path simple_p by force+

  show pl: "detour_rel_aux_locale ε (L ∪ R) p (p1 ε +++ pl ε +++ p2 ε)"
    by unfold_locales (use <ε > 0> simple' simple_p valid LR_subset homo'
LR' in <auto simp: S_def path_image_join>)
  show pr: "detour_rel_aux_locale ε (L ∪ R) p (p1 ε +++ pr ε +++ p2 ε)"
    by unfold_locales (use <ε > 0> simple' simple_p valid LR_subset homo'
LR' in <auto simp: S_def path_image_join>)

  write winding_number ("ind")
  have ind_eq: "(x ∈ L ⟶ ind (pl ε) x - ind (pr ε) x = -1) ∧
                (x ∈ R ⟶ ind (pl ε) x - ind (pr ε) x = 1)" if "x ∈
L ∪ R" for x
  proof -
    have "ind (pl ε +++ reversepath (pr ε)) x = ind (pl ε) x + ind (reversepath
(pr ε)) x"
      using that LR' simple_p by (subst winding_number_join) auto
    also have "ind (reversepath (pr ε)) x = -ind (pr ε) x"
      using that LR' simple_p by (subst winding_number_reversepath) auto
    finally show ?thesis
      using ind[OF ε, of x] that simple_p by auto
  qed

  show "ind (p1 ε +++ pl ε +++ p2 ε) x - ind (p1 ε +++ pr ε +++ p2 ε)
x = -1" if "x ∈ L" for x
    using that LR' ind_eq[of x] simple_p
    by (subst winding_number_join winding_number_reversepath; auto simp:
```

```
path_image_join)+
  show "ind (p1 ε +++ pl ε +++ p2 ε) x - ind (p1 ε +++ pr ε +++ p2 ε)
x = 1" if "x ∈ R" for x
    using that LR' ind_eq[of x] simple_p
    by (subst winding_number_join winding_number_reversepath; auto simp:
path_image_join)+
qed fact+
```

This is the main rule for proving the common avoidance pattern where we avoid a single bad point `bad` on a curve by replacing some segment of it with a circular arc with radius ε around `bad`. We only need to show that the original path `p` splits into segments p_1, p_2, and p_3 such that p_1 and p_2 do not overlap and none of the segments crosses the ε-sphere around `bad`.

```
lemma detour_rel_avoid_basic_part_circlepath_left:
  fixes p :: "real ⇒ complex" and bad :: complex and a b a' b' :: "real
⇒ real"
  defines "S  ≡ (λε. path_image p ∪ cball bad ε)"
  defines "cl ≡ (λε. part_circlepath bad ε (a' ε) (b' ε))"
  defines "cr ≡ (λε. part_circlepath bad ε (a ε) (b ε))"
  assumes "bad ∈ path_image p - {pathstart p, pathfinish p}"
  assumes eq_paths: "⋀ε. ε > 0 ⟹ ε < eps ⟹ simple_path p ⟹ eq_paths
p (p1 ε +++ pm ε +++ p2 ε)"
  assumes p12_disjoint:
     "⋀ε. ε > 0 ⟹ ε < eps ⟹ simple_path p ⟹
           path_image (p1 ε) ∩ path_image (p2 ε) ⊆ {pathstart p} ∩
{pathfinish p}"
  assumes p1_dnc: "⋀ε. ε > 0 ⟹ ε < eps ⟹ simple_path p ⟹ does_not_cross
(p1 ε) (sphere bad ε)"
  assumes p2_dnc: "⋀ε. ε > 0 ⟹ ε < eps ⟹ simple_path p ⟹ does_not_cross
(p2 ε) (sphere bad ε)"
  assumes pm_dnc: "⋀ε. ε > 0 ⟹ ε < eps ⟹ simple_path p ⟹ does_not_cross
(pm ε) (sphere bad ε)"
  assumes finish_p1: "⋀ε. ε > 0 ⟹ ε < eps ⟹ simple_path p ⟹ pathfinish
(p1 ε) = bad + rcis ε (a ε)"
  assumes start_p2:  "⋀ε. ε > 0 ⟹ ε < eps ⟹ simple_path p ⟹ pathstart
(p2 ε) = bad + rcis ε (b ε)"
  assumes valid:     "⋀ε. ε > 0 ⟹ ε < eps ⟹ valid_path p ⟹ valid_path
(p1 ε)"
                     "⋀ε. ε > 0 ⟹ ε < eps ⟹ valid_path p ⟹ valid_path
(p2 ε)"
  assumes pm_ends:   "⋀ε. ε > 0 ⟹ ε < eps ⟹ simple_path p ⟹ pathstart
(pm ε) = pathfinish (p1 ε)"
                     "⋀ε. ε > 0 ⟹ ε < eps ⟹ simple_path p ⟹ pathfinish
(pm ε) = pathstart (p2 ε)"
  assumes ab:  "⋀ε. ε > 0 ⟹ ε < eps ⟹ simple_path p ⟹ a ε ≠ b
ε   ∧ |b ε - a ε|   < 2 * pi"
  assumes ab': "⋀ε. ε > 0 ⟹ ε < eps ⟹ simple_path p ⟹ a' ε ≠ b'
ε ∧ |b' ε - a' ε| < 2 * pi ∧
                 cis (a' ε) = cis (a ε) ∧ cis (b' ε) = cis (b ε) ∧
```

```
                b ε - a ε + a' ε - b' ε = 2 * pi"
  assumes "eps > 0"
  shows   "detour_rel {bad} {} p (λε. p1 ε +++ cl ε +++ p2 ε) (λε. p1
ε +++ cr ε +++ p2 ε)"
proof (rule detour_rel_avoid_basic[of "Min {eps, dist bad (pathstart p),
dist bad (pathfinish p)}"])
  show "Min {eps, dist bad (pathstart p), dist bad (pathfinish p)} >
0"
    using ‹eps > 0› ‹bad ∈ _› by auto
  show "closed {bad}" "closed ({} :: complex set)"
    by auto
  show "{bad} ∪ {} ⊆ path_image p"
    using ‹bad ∈ _› by auto
next
  fix ε :: real
  assume "ε > 0" "ε < Min {eps, dist bad (pathstart p), dist bad (pathfinish
p)}"
  hence ε: "ε > 0" "ε < eps" and ε': "ε < dist bad (pathstart p)" "ε
< dist bad (pathfinish p)"
    by auto
  show "valid_path (p1 ε)" "valid_path (p2 ε)" if "valid_path p"
    using valid that ε by auto
  note [simp] = pathstart_part_circlepath' pathfinish_part_circlepath'

  assume simple_p: "simple_path p"
  note eq_paths = eq_paths[OF ε simple_p]

  have [simp]: "pathstart (p1 ε) = pathstart p" "pathfinish (p2 ε) = pathfinish
p"
    using eq_paths_imp_same_ends[OF eq_paths] by simp_all
  note [simp] = pm_ends[OF ε simple_p] finish_p1[OF ε simple_p] start_p2[OF
ε simple_p]

  have *: "simple_path (p1 ε +++ pm ε +++ p2 ε)"
    using simple_p eq_paths eq_paths_imp_simple_path_iff by blast
  hence **: "arc (p1 ε) ∧ arc (pm ε) ∧ arc (p2 ε)"
    by (elim simple_path_joinE arc_join; auto simp: arc_join_eq)
  have "path (p1 ε +++ pm ε +++ p2 ε)"
    using eq_paths by (simp add: eq_paths_imp_path)
  hence [simp]: "path (pm ε)" "path (p1 ε)" "path (p2 ε)"
    using pm_ends[OF ε] by (auto simp: arc_imp_path)

  note ab = ab[OF ε simple_p]
  note ab' = ab'[OF ε simple_p]

  show "pathfinish (cl ε) = pathstart (p2 ε)" "pathfinish (cr ε) = pathstart
(p2 ε)"
       "pathstart (cl ε) = pathfinish (p1 ε)" "pathstart (cr ε) = pathfinish
(p1 ε)"
```

```
    using ab' by (auto simp: cl_def cr_def rcis_def pathfinish_part_circlepath'
                             pathstart_part_circlepath' exp_eq_polar)

  show "path_image (cl ε) ∩ ({bad} ∪ {}) = {}" "path_image (cr ε) ∩
({bad} ∪ {}) = {}"
    using ε by (auto simp: cl_def cr_def path_image_def part_circlepath_def)

  have p1_cball: "path_image (p1 ε) ∩ ball bad ε = {}"
  proof -
    have "path_image (p1 ε) ⊆ cball bad ε ∨ path_image (p1 ε) ∩ ball
bad ε = {}"
      using path_fully_inside_or_fully_outside'[of "p1 ε" "cball bad ε"]
p1_dnc[OF ε simple_p]
      by simp
    moreover have "pathstart (p1 ε) ∈ path_image (p1 ε)" "pathstart (p1
ε) ∉ cball bad ε"
      by (intro pathstart_in_path_image) (use ε' in auto)
    ultimately show "path_image (p1 ε) ∩ ball bad ε = {}"
      by auto
  qed

  have p1_dnc': "path_image (p1 ε) ∩ cball bad ε ⊆ {pathfinish (p1 ε)}"
  proof
    fix x assume x: "x ∈ path_image (p1 ε) ∩ cball bad ε"
    then obtain t where t: "t ∈ {0..1}" "p1 ε t = x"
      by (auto simp: path_image_def)
    from x p1_cball have "x ∈ sphere bad ε"
      by auto
    with t and p1_dnc[OF ε simple_p] have "t ∈ {0, 1}"
      by (auto simp: does_not_cross_def)
    moreover have "pathstart (p1 ε) ∉ cball bad ε"
      using ε' by auto
    ultimately have "t = 1"
      using t x by (auto simp: pathstart_def)
    thus "x ∈ {pathfinish (p1 ε)}"
      using t x by (auto simp: pathfinish_def)
  qed

  have p2_cball: "path_image (p2 ε) ∩ ball bad ε = {}"
  proof -
    have "path_image (p2 ε) ⊆ cball bad ε ∨ path_image (p2 ε) ∩ ball
bad ε = {}"
      using path_fully_inside_or_fully_outside'[of "p2 ε" "cball bad ε"]
p2_dnc[OF ε simple_p]
      by simp
    moreover have "pathfinish (p2 ε) ∈ path_image (p2 ε)" "pathfinish
(p2 ε) ∉ cball bad ε"
      by (intro pathfinish_in_path_image) (use ε' in auto)
    ultimately show "path_image (p2 ε) ∩ ball bad ε = {}"
```

```
    by auto
  qed

  have p2_dnc': "path_image (p2 ε) ∩ cball bad ε ⊆ {pathstart (p2 ε)}"
  proof
    fix x assume x: "x ∈ path_image (p2 ε) ∩ cball bad ε"
    then obtain t where t: "t ∈ {0..1}" "p2 ε t = x"
      by (auto simp: path_image_def)
    from x p2_cball have "x ∈ sphere bad ε"
      by auto
    with t and p2_dnc[OF ε simple_p] have "t ∈ {0, 1}"
      by (auto simp: does_not_cross_def)
    moreover have "pathfinish (p2 ε) ∉ cball bad ε"
      using ε' by auto
    ultimately have "t = 0"
      using t x by (auto simp: pathfinish_def)
    thus "x ∈ {pathstart (p2 ε)}"
      using t x by (auto simp: pathstart_def)
  qed

  have "bad ∈ path_image p"
    using ‹bad ∈ _› by blast
  also have "path_image p = path_image (p1 ε +++ pm ε +++ p2 ε)"
    using eq_paths by (rule eq_paths_imp_path_image_eq)
  also have "... = path_image (p1 ε) ∪ path_image (pm ε) ∪ path_image
(p2 ε)"
    by (simp add: path_image_join Un_assoc)
  also have "... ⊆ -ball bad ε ∪ path_image (pm ε) ∪ -ball bad ε"
    by (intro Un_mono) (use p1_cball p2_cball in auto)
  finally have "bad ∈ path_image (pm ε)"
    using ε by auto

  have pm_cball: "path_image (pm ε) ⊆ cball bad ε"
  proof -
    have "path_image (pm ε) ⊆ cball bad ε ∨ path_image (pm ε) ∩ ball
bad ε = {}"
      using path_fully_inside_or_fully_outside'[of "pm ε" "cball bad ε"]
pm_dnc[OF ε simple_p]
      by simp
    moreover have "bad ∈ ball bad ε"
      using ε by auto
    ultimately show "path_image (pm ε) ⊆ cball bad ε"
      using ‹bad ∈ path_image (pm ε)› by blast
  qed

  show "path_image (p1 ε) ∩ ({bad} ∪ {}) = {}"
  proof -
    have "bad ∈ ball bad ε"
      using ε by auto
```

```
    moreover have "pathfinish (p1 ε) ∈ sphere bad ε"
      using ε by (auto simp: dist_norm)
    hence "path_image (p1 ε) ∩ ball bad ε = {}"
      using p1_cball by force
    ultimately show ?thesis
      by auto
  qed

  show "path_image (p2 ε) ∩ ({bad} ∪ {}) = {}"
  proof -
    have "bad ∈ ball bad ε"
      using ε by auto
    moreover have "pathstart (p2 ε) ∈ sphere bad ε"
      using ε by (auto simp: dist_norm)
    hence "path_image (p2 ε) ∩ ball bad ε = {}"
      using p2_cball by force
    ultimately show ?thesis
      by auto
  qed

  have simple_circ: "simple_path (p1 ε +++ part_circlepath bad ε a'' b''
+++ p2 ε)"
    if "a'' ≠ b''" "|b'' - a''| < 2 * pi" "pathfinish (p1 ε) = bad + rcis
ε a''"
       "pathstart (p2 ε) = bad + rcis ε b''" for a'' b''
  proof (intro simple_path_join3I)
    show "arc (p1 ε)" "arc (p2 ε)"
      using ** by simp_all
    show "arc (part_circlepath bad ε a'' b'')"
      using that ε by (auto intro!: arc_part_circlepath)
    show "path_image (p1 ε) ∩ path_image (p2 ε) ⊆ {pathstart (p1 ε)}
∩ {pathfinish (p2 ε)}"
      using p12_disjoint[OF ε simple_p] by simp
    show "pathfinish (p1 ε) = pathstart (part_circlepath bad ε a'' b'')"
         "pathfinish (part_circlepath bad ε a'' b'') = pathstart (p2 ε)"
      using that by (auto simp: rcis_def exp_eq_polar)
    show "path_image (p1 ε) ∩ path_image (part_circlepath bad ε a'' b'')
⊆
            {pathstart (part_circlepath bad ε a'' b'')}"
      using path_image_part_circlepath_subset'[of ε bad a'' b''] p1_dnc'
ε that
      by (force simp: does_not_cross_def rcis_def exp_eq_polar)
    show "path_image (part_circlepath bad ε a'' b'') ∩ path_image (p2
ε) ⊆ {pathstart (p2 ε)}"
      using path_image_part_circlepath_subset'[of ε bad a'' b''] p2_dnc'
ε that
      by (force simp: does_not_cross_def)
  qed
```

```
  show "simple_path (p1 ε +++ cl ε +++ p2 ε)" "simple_path (p1 ε +++
cr ε +++ p2 ε)"
    unfolding cl_def cr_def
    by (rule simple_circ; use ab finish_p1[OF ε] start_p2[OF ε] ab ab'
in ‹simp add: rcis_def›)+

  have homo_circ: "homotopic_paths (S ε) p (p1 ε +++ part_circlepath bad
ε a'' b'' +++ p2 ε)"
    if "cis a'' = cis (a ε)" "cis b'' = cis (b ε)" for a'' b''
  proof -
    have "homotopic_paths (S ε) p (p1 ε +++ pm ε +++ p2 ε)"
      by (intro eq_paths_imp_homotopic[OF eq_paths]) (auto simp: S_def)
    also have "homotopic_paths (S ε) (pm ε) (part_circlepath bad ε a''
b'')"
    proof (rule homotopic_paths_subset[OF simply_connected_imp_homotopic_paths])
      show "simply_connected (cball bad ε)"
        by (rule convex_imp_simply_connected) auto
    next
      have "path_image (part_circlepath bad ε a'' b'') ⊆ sphere bad ε"
        using ε by (intro path_image_part_circlepath_subset') auto
      also have "... ⊆ cball bad ε"
        by auto
      finally show "path_image (part_circlepath bad ε a'' b'') ⊆ cball
bad ε" .
    next
      show "same_ends (part_circlepath bad ε a'' b'') (pm ε)"
        using that by (auto simp: rcis_def rcis_def exp_eq_polar)
    next
      show "path_image (pm ε) ⊆ cball bad ε"
        by fact
    qed (auto simp: S_def)
    hence "homotopic_paths (S ε) (p1 ε +++ pm ε +++ p2 ε)
             (p1 ε +++ part_circlepath bad ε a'' b'' +++ p2 ε)"
      using ‹path_image p = _›
      by (intro homotopic_paths_join) (auto simp: S_def path_image_join)
    finally show ?thesis .
  qed

  have "homotopic_paths (S ε) p (p1 ε +++ cl ε +++ p2 ε)"
       "homotopic_paths (S ε) p (p1 ε +++ cr ε +++ p2 ε)"
    unfolding cl_def cr_def by (rule homo_circ; use ab' in simp)+
  thus "homotopic_paths (path_image p ∪ (⋃y∈{bad} ∪ {}. cball y ε))
p (p1 ε +++ cl ε +++ p2 ε)"
       "homotopic_paths (path_image p ∪ (⋃y∈{bad} ∪ {}. cball y ε))
p (p1 ε +++ cr ε +++ p2 ε)"
    by (simp_all add: S_def)

  show "winding_number (cl ε +++ reversepath (cr ε)) x = -1" if "x ∈
{bad}" for x
```

```
  proof -
    let ?γ = "part_circlepath bad ε"
    have "cl ε +++ reversepath (cr ε) = ?γ (a' ε) (b' ε) +++ ?γ (b ε)
(a ε)"
      by (simp add: cl_def cr_def)
    also have "?γ (a' ε) (b' ε) = ?γ (a ε + 2 * pi) (b ε)"
      using ab' by (intro part_circlepath_cong) (auto simp flip: cis_mult)
    also have "... +++ ?γ (b ε) (a ε) ≡_p ?γ (a ε + 2 * pi) (a ε)"
      by (intro eq_paths_part_circlepaths) (use ab ab' in <auto simp:
closed_segment_eq_real_ivl>)
    also have "... = reversepath (?γ (a ε) (a ε + 2 * pi))"
      by simp
    also have "... ≡○ reversepath (circlepath bad ε)"
      by (intro eqloops_reversepath_cong eq_loops_full_part_circlepath)
auto
    also have "winding_number ... bad = -1"
      using ε by (simp add: winding_number_reversepath winding_number_circlepath)
    finally show ?thesis
      using ε that by auto
  qed
qed (auto simp: cl_def cr_def)

lemma detour_rel_avoid_basic_part_circlepath_right:
  fixes p :: "real ⇒ complex" and bad :: complex and a b a' b' :: "real
⇒ real"
  defines "S ≡ (λε. path_image p ∪ cball bad ε)"
  defines "cl ≡ (λε. part_circlepath bad ε (a' ε) (b' ε))"
  defines "cr ≡ (λε. part_circlepath bad ε (a ε) (b ε))"
  assumes "bad ∈ path_image p - {pathstart p, pathfinish p}"
  assumes eq_paths:     "⋀ε. ε > 0 ⟹ ε < eps ⟹ simple_path p ⟹
eq_paths p (p1 ε +++ pm ε +++ p2 ε)"
  assumes p12_disjoint: "⋀ε. ε > 0 ⟹ ε < eps ⟹ simple_path p ⟹
path_image (p1 ε) ∩ path_image (p2 ε) ⊆ {pathstart p} ∩ {pathfinish p}"
  assumes p1_dnc: "⋀ε. ε > 0 ⟹ ε < eps ⟹ simple_path p ⟹ does_not_cross
(p1 ε) (sphere bad ε)"
  assumes p2_dnc: "⋀ε. ε > 0 ⟹ ε < eps ⟹ simple_path p ⟹ does_not_cross
(p2 ε) (sphere bad ε)"
  assumes pm_dnc: "⋀ε. ε > 0 ⟹ ε < eps ⟹ simple_path p ⟹ does_not_cross
(pm ε) (sphere bad ε)"
  assumes finish_p1: "⋀ε. ε > 0 ⟹ ε < eps ⟹ simple_path p ⟹ pathfinish
(p1 ε) = bad + rcis ε (a ε)"
  assumes start_p2:  "⋀ε. ε > 0 ⟹ ε < eps ⟹ simple_path p ⟹ pathstart
(p2 ε) = bad + rcis ε (b ε)"
  assumes pm_ends:   "⋀ε. ε > 0 ⟹ ε < eps ⟹ simple_path p ⟹ pathstart
(pm ε) = pathfinish (p1 ε)"
                     "⋀ε. ε > 0 ⟹ ε < eps ⟹ simple_path p ⟹ pathfinish
(pm ε) = pathstart (p2 ε)"
  assumes valid:     "⋀ε. ε > 0 ⟹ ε < eps ⟹ valid_path p ⟹ valid_path
(p1 ε)"
```

```
                    "⋀ε. ε > 0 ⟹ ε < eps ⟹ valid_path p ⟹ valid_path
(p2 ε)"
  assumes ab:  "⋀ε. ε > 0 ⟹ ε < eps ⟹ simple_path p ⟹ a ε ≠ b
ε ∧ |b ε - a ε| < 2 * pi"
  assumes ab': "⋀ε. ε > 0 ⟹ ε < eps ⟹ simple_path p ⟹ a' ε ≠ b'
ε ∧ |b' ε - a' ε| < 2 * pi ∧
                cis (a' ε) = cis (a ε) ∧ cis (b' ε) = cis (b ε) ∧
                b ε - a ε + a' ε - b' ε = 2 * pi"
  assumes "eps > 0"
  shows   "detour_rel {} {bad} p (λε. p1 ε +++ cr ε +++ p2 ε) (λε. p1
ε +++ cl ε +++ p2 ε)"
  using detour_rel_avoid_basic_part_circlepath_left[of bad p eps p1 pm
p2 a b a' b'] assms
  by (simp add: detour_rel_swap)
```

4.4.2 Straight line

```
definition avoid_linepath where
  "avoid_linepath left a b bad ε =
     linepath a (bad - (ε / dist a b) *R (b - a)) +++
     part_circlepath bad ε (Arg (b - a) + (if left then pi else -pi))
(Arg (b - a)) +++
     linepath (bad + (ε / dist a b) *R (b - a)) b"

lemma valid_path_avoid_linepath:
  assumes "a ≠ b"
  shows   "valid_path (avoid_linepath left a b bad ε)"
  unfolding avoid_linepath_def using assms
  by (intro valid_path_join valid_path_linepath valid_path_part_circlepath)
     (auto simp: scaleR_conv_of_real rcis_def cis_Arg complex_sgn_def
dist_norm norm_minus_commute
                field_simps rcis_def exp_eq_polar simp flip: cis_divide
cis_mult)

lemma avoid_linepath_translate:
  "(+) c ∘ avoid_linepath left a b bad ε =
   avoid_linepath left (c + a) (c + b) (c + bad) ε"
  unfolding avoid_linepath_def path_compose_join linepath_translate
            part_circlepath_translate by (simp add: algebra_simps)

lemma avoid_linepath_mult:
  assumes "a ≠ b"
  shows   "(*) c ∘ avoid_linepath left a b 0 ε =
           avoid_linepath left (c * a) (c * b) 0 (norm c * ε)"
proof (cases "c = 0")
  assume "c = 0"
  thus ?thesis
    by (auto simp: avoid_linepath_def joinpaths_def
                   fun_eq_iff part_circlepath_def linepath_def)
```

```
next
  assume [simp]: "c ≠ 0"
  define β where "β = (if left then pi else -pi)"
  have *: "part_circlepath 0 (ε * cmod c) (β + Arg (b * c - a * c)) (Arg
(b * c - a * c)) =
           part_circlepath 0 (ε * cmod c) (β + (Arg c + Arg (b - a)))
(Arg c + Arg (b - a))"
    by (rule part_circlepath_cong)
       (use assms in ‹simp_all flip: cis_mult cis_divide sgn_mult
                                add: cis_Arg ring_distribs mult_ac›)
  show ?thesis
  unfolding avoid_linepath_def path_compose_join linepath_mult_complex

            part_circlepath_mult_complex β_def [symmetric] dist_mult_left
  using assms * by (simp add: algebra_simps)
qed

lemma detour_rel_linepath_semicircle_left_aux1:
  assumes "a < 0" "b > 0"
  shows "detour_rel {0} {} (linepath (complex_of_real a) (complex_of_real
b))
           (avoid_linepath True  (complex_of_real a) (complex_of_real
b) 0)
           (avoid_linepath False (complex_of_real a) (complex_of_real
b) 0)"
proof -
  define p where "p ≡ linepath (of_real a) (of_real b :: complex)"
  define p1 where "p1 ≡ (λε. linepath (of_real a) (-of_real ε :: complex))"
  define p2 where "p2 ≡ (λε. linepath (of_real ε :: complex) (of_real
b))"
  define pm where "pm ≡ (λε. linepath (-of_real ε) (of_real ε :: complex))"
  define cl where "cl ≡ (λε. part_circlepath 0 ε pi 0)"
  define cr where "cr ≡ (λε. part_circlepath 0 ε (-pi) 0)"
  note [simp] = closed_segment_eq_real_ivl1 closed_segment_same_Im
  note homoI = homotopic_paths_subset[OF simply_connected_imp_homotopic_paths]

  note [[goals_limit = 30]]
  have "detour_rel {0} {} p (λε. p1 ε +++ cl ε +++ p2 ε) (λε. p1 ε +++
cr ε +++ p2 ε)"
    unfolding cl_def cr_def
  proof (rule detour_rel_avoid_basic_part_circlepath_left [where eps
= "min (-a) b"])
    show "0 ∈ path_image p - {pathstart p, pathfinish p}"
      using assms by (auto simp: p_def)
  next
    fix ε assume ε: "ε > 0" "ε < min (-a) b"
    define S where
      "S ≡ path_image (linepath (complex_of_real a) (of_real b)) ∪ (⋃y∈{0}
∪ {}. cball y ε)"
```

```
    show "eq_paths p (p1 ε +++ pm ε +++ p2 ε)"
      unfolding p1_def pm_def p2_def p_def using ε
      by (auto intro!: eq_paths_joinpaths_linepath')

    show "path_image (p1 ε) ∩ path_image (p2 ε) ⊆ {pathstart p} ∩ {pathfinish
p}"
      using assms ε by (auto simp: p1_def p2_def)

    have *: "path_image (p1 ε) ∩ sphere 0 ε ⊆ {pathfinish (p1 ε)}"
            "path_image (p2 ε) ∩ sphere 0 ε ⊆ {pathstart (p2 ε)}" us-
ing ε
      by (auto simp: p1_def p2_def complex_eq_iff simp flip: complex_is_Real_iff
elim!: Reals_cases)
    show "does_not_cross (p1 ε) (sphere 0 ε)" "does_not_cross (p2 ε)
(sphere 0 ε)"
      by (subst does_not_cross_simple_path; use ε * in ‹force simp: p1_def
p2_def›)+
    have "path_image (pm ε) ⊆ cball 0 ε"
      unfolding pm_def path_image_linepath using ε by (intro closed_segment_subset)
auto
    thus "does_not_cross (pm ε) (sphere 0 ε)"
    proof (subst does_not_cross_simple_path)
      show "path_image (pm ε) ∩ sphere 0 ε ⊆ {pathstart (pm ε), pathfinish
(pm ε)}"
      proof
        fix z assume z: "z ∈ path_image (pm ε) ∩ sphere 0 ε"
        then obtain x where x: "x ∈ {-ε..ε}" "z = of_real x"
          using ε by (auto simp: pm_def complex_eq_iff)
        with z have "x ∈ {-ε, ε}"
          by auto
        thus "z ∈ {pathstart (pm ε), pathfinish (pm ε)}"
          using x by (auto simp: pm_def)
      qed
    qed (use ε in ‹auto simp: pm_def›)
  qed (use assms in ‹auto simp: p1_def rcis_def p2_def pm_def split: if_splits›)

  thus ?thesis using assms
    by (auto simp: p_def p1_def cl_def cr_def p2_def avoid_linepath_def
[abs_def]
                   scaleR_conv_of_real dist_real_def simp flip: of_real_diff)
qed

lemma detour_rel_linepath_semicircle_left_aux2:
  assumes "0 ∈ open_segment a b"
  shows "detour_rel {0} {} (linepath a b)
           (avoid_linepath True  a b 0)
           (avoid_linepath False a b 0)"
proof -
```

```
  define α where "α = sgn (b - a)"
  have [simp]: "α ≠ 0" "a ≠ 0" "b ≠ 0"
    using assms by (auto simp: open_segment_def α_def sgn_eq_0_iff)
  define a' b' where "a' = complex_of_real (-norm a)" and "b' = complex_of_real
(norm b)"
  let ?γ = "λleft a b. avoid_linepath left a b 0"

  obtain u where u: "u ∈ {0<..<1}" "(1 - u) *R a + u *R b = 0"
    using assms by (auto simp: in_segment)
  hence b_eq: "b = (-(1 - u) / u) *R a"
    by (simp add: field_simps scaleR_conv_of_real)
  have "-norm a = norm b ⟷ a = 0 ∧ b = 0"
  proof safe
    assume "-norm a = norm b"
    hence "norm a = 0 ∧ norm b = 0"
      using norm_ge_zero[of a] norm_ge_zero[of b] by linarith
    thus "a = 0" "b = 0"
      by auto
  qed auto
  hence ne: "a ≠ b" "a' ≠ b'"
    using assms by (auto simp: a'_def b'_def in_segment)

  have "detour_rel {0} {} (linepath a' b') (?γ True a' b') (?γ False a'
b')"
    unfolding a'_def b'_def
    by (rule detour_rel_linepath_semicircle_left_aux1) (use assms in <auto
simp: open_segment_def>)
  hence "detour_rel ((*) α ` {0}) ((*) α ` {}) ((*) α ∘ linepath a' b')
          (λε. (*) α ∘ ?γ True a' b' (ε / norm α)) (λε. (*) α ∘ ?γ False
a' b' (ε / norm α))"
    by (rule detour_rel_mult) auto
  hence "detour_rel {0} {} (linepath (α * a') (α * b'))
          (?γ True (α * a') (α * b')) (?γ False (α * a') (α * b'))"
    by (auto simp: linepath_mult_complex avoid_linepath_mult ne)
  also have "α * a' = a"
    using ne u(1) by (simp add: a'_def b'_def b_eq α_def complex_sgn_def

                                field_simps scaleR_conv_of_real norm_divide)
  also have "α * b' = b"
  proof -
    have "α * b' = sgn (b - a) * complex_of_real (cmod b)"
      unfolding α_def a'_def b'_def by simp
    also have "b - a = -(1 / u) *R a"
      using u(1) by (simp add: b_eq scaleR_conv_of_real field_simps)
    also have "sgn ... = -sgn a"
      unfolding scaleR_conv_of_real using u(1) by (subst sgn_mult) (auto
simp: sgn_of_real)
    also have "-sgn a * complex_of_real (norm b) = -(norm a * sgn a) *
of_real ((1 - u) / u)"
```

```
    using u(1) by (simp add: b_eq field_simps)
  also have "norm a * sgn a = a"
    using ‹a ≠ 0› by (simp add: complex_sgn_def scaleR_conv_of_real)
  also have "-a * of_real ((1 - u) / u) = b"
    using u(1) by (simp add: b_eq scaleR_conv_of_real field_simps)
  finally show ?thesis .
 qed
 finally show ?thesis .
qed

lemma detour_rel_linepath_semicircle_left [detour_rel_intros]:
 assumes "bad ∈ open_segment a b"
 shows   "detour_rel {bad} {} (linepath a b)
           (avoid_linepath True  a b bad)
           (avoid_linepath False a b bad)"
proof -
 let ?γ = "avoid_linepath"
 show ?thesis
 proof -
  have "detour_rel {0} {} (linepath (a - bad) (b - bad))
         (?γ True  (a - bad) (b - bad) 0) (?γ False (a - bad) (b -
bad) 0)"
  proof (rule detour_rel_linepath_semicircle_left_aux2)
    have "bad ∈ open_segment a b ⟷ -bad + bad ∈ open_segment (-bad
+ a) (-bad + b)"
      by (rule open_segment_translation_eq [symmetric])
    also have "-bad + bad = 0"
      by simp
    finally show "0 ∈ open_segment (a - bad) (b - bad)"
      using assms by simp
  qed
  hence "detour_rel ((+) bad ‘ {0}) ((+) bad ‘ {})
         ((+) bad ∘ linepath (a - bad) (b - bad))
         (λε. (+) bad ∘ ?γ True  (a - bad) (b - bad) 0 ε)
         (λε. (+) bad ∘ ?γ False (a - bad) (b - bad) 0 ε)"
    by (rule detour_rel_translate)
  thus ?thesis
    by (auto simp: linepath_translate avoid_linepath_translate)
 qed
qed

lemma detour_rel_linepath_semicircle_right [detour_rel_intros]:
 assumes "bad ∈ open_segment a b"
 shows   "detour_rel {} {bad} (linepath a b)
           (avoid_linepath False a b bad)
           (avoid_linepath True  a b bad)"
 using detour_rel_linepath_semicircle_left[OF assms] by (simp add: detour_rel_swap)
```

4.4.3 Circular arc

```
definition avoid_part_circlepath ::
    "bool ⇒ complex ⇒ real ⇒ real ⇒ real ⇒ real ⇒ real
⇒ complex" where
  "avoid_part_circlepath left x r a b φ ε = (
     let s = sgn (b - a);
         bad = x + rcis r φ;
         α = 2 * arcsin (ε / (2 * r));
         β = arcsin (ε / (2 * r)) + pi / 2
     in  part_circlepath x r a (φ - s * α) +++
         part_circlepath bad ε (φ - s * β + (if left = (a > b) then 0
else 2 * s * pi)) (φ + s * β) +++
         part_circlepath x r (φ + s * α) b)"

lemma avoid_part_circlepath_translate:
  "(+) c ∘ avoid_part_circlepath left x r a b φ ε =
   avoid_part_circlepath left (c + x) r a b φ ε"
proof -
  define δ where "δ = (if left then 0 else 2 * pi)"
  show ?thesis
  unfolding avoid_part_circlepath_def path_compose_join linepath_translate

            part_circlepath_translate δ_def [symmetric] Let_def  by (simp
add: add_ac)
qed

lemma avoid_part_circlepath_mult_scaleR_0:
  assumes [simp]: "bad ≠ 0" and "c > 0"
  shows   "(*R) c ∘ avoid_part_circlepath left 0 r a b φ ε =
             avoid_part_circlepath left 0 (norm c * r) a b φ (norm c *
ε)"
proof -
  define δ where "δ = (if left then 0 else 2 * pi)"
  show ?thesis
  unfolding avoid_part_circlepath_def path_compose_join linepath_scaleR
            part_circlepath_scaleR δ_def [symmetric] Let_def
  by (intro arg_cong2[of _ _ _ _ "(+++)"] part_circlepath_cong refl)
     (use ‹c > 0› in ‹simp_all flip: cis_mult cis_divide
                                add: cis_Arg sgn_mult scaleR_conv_of_real
rcis_def›)
qed

lemma avoid_part_circlepath_mult:
  assumes [simp]: "c ≠ 0"
  shows   "(*) c ∘ avoid_part_circlepath left 0 r a b φ ε =
             avoid_part_circlepath left 0 (norm c * r)
               (a + Arg c) (b + Arg c) (φ + Arg c) (norm c * ε)"
proof -
  define δ where "δ = (if left then 0 else 2 * pi)"
```

```
  show ?thesis
  unfolding avoid_part_circlepath_def path_compose_join linepath_mult_complex
            part_circlepath_mult_complex δ_def [symmetric] Let_def
  by (intro arg_cong2[of _ _ _ _ "(+++)"] part_circlepath_cong refl)
     (simp_all flip: cis_mult cis_divide cis_cnj
                          add: cis_Arg sgn_mult divide_conv_cnj norm_mult
rcis_def
                               complex_sgn_def scaleR_conv_of_real)
qed

lemma avoid_part_circlepath_cong:
  assumes "cis a = cis a'" "b' = b + a' - a" "φ' = φ + a' - a"
  shows "avoid_part_circlepath left x r a b φ =
         avoid_part_circlepath left x r a' b' φ'"
  unfolding avoid_part_circlepath_def Let_def
  by (rule ext, intro arg_cong2[of _ _ _ _ "(+++)"] part_circlepath_cong
refl)
     (use assms in ‹simp_all flip: cis_mult cis_divide add: rcis_def›)

lemma avoid_part_circlepath_wf_aux1:
  assumes "r ≥ 0" "r ≤ 2"
  shows    "(1 + of_real r * cis (arcsin (r / 2) + pi / 2)) = cis (2 *
arcsin (r / 2))"
proof -
  have "(r / 2)^2 ≤ (2 / 2) ^ 2"
    using assms by (intro power_mono) auto
  hence "(r / 2) ^ 2 ≤ 1"
    by (simp add: power2_eq_square)
  thus ?thesis using assms
    by (simp add: complex_eq_iff cos_arcsin sin_double cos_double sin_add
cos_add power2_eq_square)
qed


locale avoid_part_circlepath_locale =
  fixes x :: complex and r a b φ ε :: real
  assumes ε: "ε > 0" "ε < 2 * r" and ab: "a ≠ b"
begin

definition s where "s = sgn (b - a)"
definition bad where "bad = x + rcis r φ"
definition "α = 2 * arcsin (ε / (2 * r))"
definition "β = arcsin (ε / (2 * r)) + pi / 2"

lemma ends:
  "x + rcis r (φ - s * α) = bad +
     rcis ε (φ - s * β + (if left = (a > b) then 0 else 2 * s * pi))"
(is ?th1)
  "x + rcis r (φ + s * α) = bad + rcis ε (φ + s * β)" (is ?th2)
```

```
proof -
  have *: "x + rcis r (φ - s * α) = bad + rcis ε (φ - s * β)" if s: "s
∈ {-1, 1}" for s
  proof -
    have "bad + rcis ε (φ - s * β) = x + (rcis r φ + rcis ε (φ - s *
β))"
      by (auto simp: rcis_def bad_def simp flip: cis_divide cis_mult)
    also have "rcis ε (φ - s * β) = ε *R cis φ * cis (-s * β)" using
ab
      by (auto simp: rcis_def scaleR_conv_of_real s_def divide_conv_cnj
sgn_if
               simp flip: cis_divide cis_cnj)
    also have "rcis r φ + ε *R cis φ * cis (-s * β) =
                 (cis φ * of_real r) * (1 + of_real (ε / r) * cis (-s
* β))"
      using ε by (simp add: field_simps scaleR_conv_of_real rcis_def)
    also have *: "(1 + of_real (ε / r) * cis β) = cis α"
      using avoid_part_circlepath_wf_aux1[of "ε / r"] ε
      by (simp add: β_def α_def mult_ac divide_simps
               del: div_mult_self3 div_mult_self4 div_mult_self2 div_mult_self1)
    hence "(1 + of_real (ε / r) * cis (-s*β)) = cis (-s*α)"
    proof (cases "s = 1")
      case [simp]: True
      have "cnj ((1 + of_real (ε / r) * cis (-s*β))) = cnj (cis (-s*α))"
        using * by (simp add: cis_cnj)
      thus ?thesis
        using complex_cnj_cancel_iff by blast
    qed (use * s ab in auto)
    also have "x + cis φ * of_real r * ... = x + rcis r (φ - s * α)" us-
ing ab
      by (auto simp: s_def sgn_if scaleR_conv_of_real rcis_def divide_conv_cnj
               simp flip: cis_divide cis_cnj cis_mult)
    finally show ?thesis
      by (simp add: rcis_def scaleR_conv_of_real divide_conv_cnj mult_ac
               flip: cis_divide cis_cnj)
  qed

  have "rcis ε (φ - s * β + (if left = (a > b) then 0 else 2 * s * pi))
= rcis ε (φ - s * β)"
    by (auto simp: rcis_def s_def sgn_if simp flip: cis_divide cis_mult)
  with *[of s] show ?th1 using ab
    by (auto simp: s_def sgn_if)

  from *[of "-s"] show ?th2 using ab
    by (auto simp: s_def sgn_if)
qed

lemma valid_path: "valid_path (avoid_part_circlepath left x r a b φ
ε)"
```

```
  unfolding avoid_part_circlepath_def Let_def s_def [symmetric]
            bad_def [symmetric] α_def [symmetric] β_def [symmetric]
  by (intro valid_path_join valid_path_linepath valid_path_part_circlepath)
     (use ends in ‹simp_all add: rcis_def exp_eq_polar›)

end

lemma valid_path_avoid_part_circlepath:
  assumes "ε ∈ {0<..<2*r}" "a ≠ b"
  shows   "valid_path (avoid_part_circlepath left x r a b φ ε)"
proof -
  interpret avoid_part_circlepath_locale x r a b φ ε
    by standard (use assms in auto)
  show ?thesis
    by (rule valid_path)
qed

lemma valid_path_avoid_part_circlepath':
  assumes "a ≠ b" "r > 0"
  shows   "eventually (λε. valid_path (avoid_part_circlepath left x r
a b φ ε)) (at_right 0)"
proof -
  have "eventually (λε. ε ∈ {0<..<2*r}) (at_right 0)"
    using assms(2) eventually_at_right_real by force
  thus ?thesis
    by eventually_elim (use assms in ‹auto intro!: valid_path_avoid_part_circlepath›)
qed

lemma sphere_inter_sphere_pos_real_line_aux1:
  assumes r: "r ≥ 0" "r ≤ 1"
  shows   "dist 1 (cis (2 * arcsin (r / 2))) = r"
proof -
  have "r ^ 2 / 4 ≤ 1 ^ 2 / 4"
    using r by (intro divide_right_mono power_mono) auto
  also have "... ≤ 1"
    by simp
  finally have *: "r ^ 2 / 4 ≤ 1" .
  thus "dist 1 (cis (2 * arcsin (r / 2))) = r"
    using r * by (simp add: dist_norm norm_complex_def cos_arcsin power2_eq_square
                            algebra_simps cos_double sin_double)
qed

lemma sphere_inter_sphere_pos_real_line_aux2':
  assumes r: "r ≥ 0" "r ≤ 1"
  assumes dist: "dist 1 (cis x) = r"
  shows "[x = 2 * arcsin (r / 2)] (rmod 2 * pi) ∨ [x = -2 * arcsin (r
/ 2)] (rmod 2 * pi)"
proof -
  have "r ^ 2 ≤ 1"
```

```
    using power_mono[of r 1 2] assms by simp
  have "r ^ 2 = norm (cis x - 1) ^ 2"
    by (subst dist [symmetric]) (auto simp: dist_norm norm_minus_commute)
  also have "... = (cos x - 1) ^ 2 + (sin x)^2"
    unfolding cmod_power2 by simp
  also have "... = sin x ^ 2 + cos x ^ 2 - 2 * cos x + 1"
    by (simp add: power2_eq_square algebra_simps)
  also have "sin x ^ 2 + cos x ^ 2 = 1"
    by (rule sin_cos_squared_add)
  finally have "cos x = 1 - r ^ 2 / 2"
    by (simp add: field_simps)
  also have "cos x = cos |x|"
    by simp
  also have "1 - r ^ 2 / 2 = sqrt (1 - (r / 2) ^ 2) ^ 2 - (r / 2) ^ 2"
    using r ‹r ^ 2 ≤ 1› by (simp add: power2_eq_square algebra_simps)
  also have "... = cos (2 * arcsin (r / 2))"
    using r by (simp add: cos_double cos_arcsin)
  finally have *: "cos |x| = cos (2 * arcsin (r / 2))" .
  from cos_eq_cos_conv_rmod [OF this] show ?thesis
    by (cases "x ≥ 0") (auto simp: rcong_uminus_left_iff)
qed

lemma sphere_inter_sphere_pos_real_line_aux2:
  assumes r: "r ≥ 0" "r ≤ 1" and x: "x ∈ {-pi..pi}"
  assumes dist: "dist 1 (cis x) = r"
  shows "|x| = 2 * arcsin (r / 2)"
proof -
  have "arcsin (r / 2) ≥ arcsin 0" "arcsin (r / 2) ≤ arcsin (1 / 2)"
    using assms by (intro arcsin_le_arcsin; simp)+
  hence arcsin: "arcsin (r / 2) ∈ {0..pi / 6}"
    by auto

  have "[x = 2 * arcsin (r / 2)] (rmod 2 * pi) ∨ [x = -2 * arcsin (r
/ 2)] (rmod 2 * pi)"
    by (intro sphere_inter_sphere_pos_real_line_aux2') (use assms in auto)
  hence "x = 2 * arcsin (r / 2) ∨ x = -2 * arcsin (r / 2)"
  proof (rule disj_forward)
    assume "[x = 2 * arcsin (r / 2)] (rmod 2 * pi)"
    moreover have "|x - 2 * arcsin (r / 2)| < |2 * pi|"
      using pi_gt_zero arcsin x unfolding atLeastAtMost_iff by linarith
    ultimately show "x = 2 * arcsin (r / 2)"
      by (rule rcong_imp_eq)
  next
    assume "[x = -2 * arcsin (r / 2)] (rmod 2 * pi)"
    moreover have "|x - (-2 * arcsin (r / 2))| < |2 * pi|"
      using pi_gt_zero arcsin x unfolding atLeastAtMost_iff by linarith
    ultimately show "x = -2 * arcsin (r / 2)"
      by (rule rcong_imp_eq)
  qed
```

```
  thus ?thesis using arcsin
    by (auto simp: abs_if)
qed

lemma sphere_inter_sphere_aux1:
  assumes r: "r ≥ 0" "r ≤ 1"
  shows   "dist (cis φ) (cis (φ + 2 * arcsin (r / 2))) = r"
proof -
  have "dist (cis φ * 1) (cis φ * cis (2 * arcsin (r / 2))) =
        dist 1 (cis (2 * arcsin (r / 2)))"
    by (subst dist_mult_left) auto
  also have "... = r"
    by (rule sphere_inter_sphere_pos_real_line_aux1) (use assms in auto)
  finally show ?thesis
    by (simp add: cis_mult)
qed

lemma sphere_inter_sphere_aux2:
  assumes r: "r ≥ 0" "r ≤ 1"
  assumes dist: "dist (cis φ) (cis x) = r"
  shows "[x = φ + 2 * arcsin (r / 2)] (rmod 2 * pi) ∨ [x = φ - 2 * arcsin
(r / 2)] (rmod 2 * pi)"
proof -
  have "dist (cis φ * 1) (cis φ * cis (x - φ)) =
           dist 1 (cis (x - φ))"
    by (subst dist_mult_left) auto
  also have "cis φ * cis (x - φ) = cis x"
    by (simp add: cis_mult)
  finally have *: "dist 1 (cis (x - φ)) = r"
    using dist by simp
  have "[x - φ = 2 * arcsin (r / 2)] (rmod 2 * pi) ∨
        [x - φ = -2 * arcsin (r / 2)] (rmod 2 * pi)"
    by (intro sphere_inter_sphere_pos_real_line_aux2') (use * r in auto)
  hence "[x - φ + φ = 2 * arcsin (r / 2) + φ] (rmod 2 * pi) ∨
         [x - φ + φ = -2 * arcsin (r / 2) + φ] (rmod 2 * pi)"
    by (rule disj_forward; intro rcong_intros)
  thus ?thesis
    by (simp add: algebra_simps)
qed

lemma detour_rel_part_circlepath_semicircle_left_aux1:
  assumes ab: "a < 0" "0 < b" "a ≥ -pi" "b ≤ pi"
  shows "detour_rel {1} {} (part_circlepath 0 1 a b)
           (avoid_part_circlepath True  0 1 a b 0)
           (avoid_part_circlepath False 0 1 a b 0)"
proof -
  define α where "α = (λε. 2 * arcsin (ε/2))"
  define β where "β = (λε. arcsin (ε/2) + pi / 2)"
  define p where "p ≡ part_circlepath 0 1 a b"
```

```
  define p1 where "p1 ≡ (λε. part_circlepath 0 1 a (-α ε))"
  define p2 where "p2 ≡ (λε. part_circlepath 0 1 (α ε) b)"
  define pm where "pm ≡ (λε. part_circlepath 0 1 (-α ε) (α ε))"

  define cl :: "real ⇒ real ⇒ complex"
    where "cl = (λε. part_circlepath 1 ε (-β ε + 2 * pi) (β ε))"
  define cr :: "real ⇒ real ⇒ complex"
    where "cr = (λε. part_circlepath 1 ε (-β ε) (β ε))"
  note [simp] = closed_segment_eq_real_ivl1 closed_segment_same_Im
  note homoI = homotopic_paths_subset[OF simply_connected_imp_homotopic_paths]

  define eps where "eps = Min {1, - (sin (a / 2) * 2), sin (b / 2) *
2}"

  note [[goals_limit = 30]]
  have "detour_rel {1} {} p (λε. p1 ε +++ cl ε +++ p2 ε) (λε. p1 ε +++
cr ε +++ p2 ε)"
    unfolding cl_def cr_def
  proof (rule detour_rel_avoid_basic_part_circlepath_left [where eps
= eps])
    have "1 = p (a / (a - b))" "a / (a - b) ∈ {0..1}"
      using ab by (auto simp: p_def part_circlepath_altdef linepath_def
field_simps)
    hence "1 ∈ path_image p"
      unfolding path_image_def by auto
    moreover have "1 ∉ {pathstart p, pathfinish p}"
      using ab by (auto simp: p_def rcis_def cis_eq_1_iff' rcis_def exp_eq_polar)
    ultimately show "1 ∈ path_image p - {pathstart p, pathfinish p}"
      by blast
    show "eps > 0"
      unfolding eps_def using ab
      by (subst Min_gr_iff) (auto intro!: sin_lt_zero' sin_gt_zero)
  next
    fix ε :: real assume ε: "ε > 0" "ε < eps"
    have "eps ≤ x" if "x ∈ {1, -sin (a/2) * 2, sin (b/2) * 2}" for x
      using that unfolding eps_def by (intro Min.coboundedI) auto
    with ε have ε: "ε > 0" "ε < 1" "ε < -sin (a/2) * 2" "ε < sin (b/2)
* 2"
      unfolding insert_iff empty_iff simp_thms by force+

    have "arcsin (ε / 2) > 0"
      using ε by (intro arcsin_pos) auto
    moreover have "arcsin (ε / 2) < arcsin 1"
      using ε by (subst arcsin_less_mono) auto
    ultimately have arcsin: "arcsin (ε / 2) > 0" "arcsin (ε / 2) < pi
/ 2"
      using ε by simp_all

    define S where "S ≡ path_image p ∪ (cball 1 ε)"
```

```
    have [simp]: "path (p1 ε)" "path (p2 ε)" "path (pm ε)"
      by (simp_all add: p1_def p2_def pm_def)
    have "α ε > 0"
      using ε by (simp add: α_def arcsin_pos)
    have "arcsin (ε / 2) < arcsin (sin (-a/2))"
      using ε ab by (subst arcsin_less_mono) auto
    hence "a < -α ε"
      using ε ab by (simp add: α_def field_simps arcsin_sin del: sin_minus)
    have "arcsin (ε / 2) < arcsin (sin (b/2))"
      using ε ab by (subst arcsin_less_mono) auto
    hence "b > α ε"
      using ε ab by (simp add: α_def field_simps arcsin_sin del: sin_minus)
    note α = ‹a < -α ε› ‹α ε < b› ‹α ε > 0›

    have "(ε / 2) ^ 2 ≤ (1 / 2) ^ 2"
      using ε by (intro power_mono divide_right_mono) auto
    hence *: "1 - (ε / 2)^2 ≥ 0"
      by (simp add: power_divide)
    have **: "cis (α ε) = 1 + complex_of_real ε * cis (β ε)"
      using α ε ab * unfolding β_def α_def
      by (auto simp add: complex_eq_iff diff_divide_distrib
                    cos_diff sin_diff add_divide_distrib cos_add sin_add
cos_arcsin
                    cos_double sin_double power_divide power2_eq_square
[symmetric])

    have ends: "pathstart (p1 ε) = rcis 1 a"
               "pathfinish (p1 ε) = pathstart (pm ε)"
               "pathfinish (pm ε) = pathstart (p2 ε)"
               "pathstart (pm ε) = 1 + rcis ε (- β ε)"
               "pathstart (p2 ε) = 1 + rcis ε (β ε)"
               "pathfinish (p2 ε) = rcis 1 b"
               "pathstart (cl ε) = pathstart (pm ε)" "pathstart (cr ε)
= pathstart (pm ε)"
               "pathfinish (cl ε) = pathstart (p2 ε)" "pathfinish (cr
ε) = pathstart (p2 ε)"
      by (auto simp: p1_def p2_def pm_def cl_def cr_def rcis_def ** divide_conv_cnj
rcis_def exp_eq_polar
               simp flip: cis_cnj cis_divide)

    have "eq_paths p (p1 ε +++ part_circlepath 0 1 (-α ε) b)"
      unfolding p1_def p_def
      by (rule eq_paths_sym[OF eq_paths_part_circlepaths]) (use ε α in
auto)
    also have "eq_paths ... (p1 ε +++ pm ε +++ p2 ε)"
      unfolding p1_def p2_def pm_def
      by (intro eq_paths_join eq_paths_sym[OF eq_paths_part_circlepaths])
(use ε α in auto)
    finally show "eq_paths p (p1 ε +++ pm ε +++ p2 ε)" .
```

```
    show "path_image (p1 ε) ∩ path_image (p2 ε) ⊆ {pathstart p} ∩ {pathfinish
p}"
    proof (intro subsetI; elim IntE)
      fix x assume x: "x ∈ path_image (p1 ε)" "x ∈ path_image (p2 ε)"
      from x(1) obtain u where u: "u ∈ {a..-α ε}" "x = cis u"
        unfolding p1_def using ab α
        by (subst (asm) path_image_part_circlepath') (auto simp: rcis_def)
      from x(2) obtain v where v: "v ∈ {α ε..b}" "x = cis v"
        unfolding p2_def using ab α
        by (subst (asm) path_image_part_circlepath') (auto simp: rcis_def)
      from u v have "cis (u - v) = 1"
        by (simp flip: cis_divide)
      then obtain n where n': "u - v = 2 * pi * of_int n"
        by (auto simp: cis_eq_1_iff)
      hence n: "of_int n = (u - v) / (2 * pi)"
        by simp

      have "of_int n < (1 :: real)"
        unfolding n using u v ab α by simp
      hence "n < 1"
        by linarith
      moreover have "of_int n ≥ (-1 :: real)"
        unfolding n using u v ab α by (simp add: field_simps)
      hence "n ≥ -1"
        by linarith
      ultimately have "n = 0 ∨ n = -1"
        by linarith
      thus "x ∈ {pathstart p} ∩ {pathfinish p}"
      proof
        assume "n = 0"
        thus ?thesis using u v α ab n' by auto
      next
        assume "n = -1"
        hence "u ≥ -pi" "v ≤ pi" "u - v = -2 * pi"
          using u v ab n' α by simp_all
        hence "u = -pi ∧ v = pi ∧ a = -pi ∧ b = pi"
          using u v ab unfolding atLeastAtMost_iff by linarith
        thus ?thesis using u v
          by (auto simp: p_def rcis_def rcis_def exp_eq_polar)
      qed
    qed

    show "does_not_cross (p1 ε) (sphere 1 ε)"
    proof (subst does_not_cross_simple_path)
      show "path_image (p1 ε) ∩ sphere 1 ε ⊆ {pathstart (p1 ε), pathfinish
(p1 ε)}"
      proof (intro subsetI; elim IntE)
        fix x assume x: "x ∈ path_image (p1 ε)" "x ∈ sphere 1 ε"
```

```
        obtain t where t: "t ∈ {a..-α ε}" "x = cis t"
          using x(1) α unfolding p1_def path_image_part_circlepath' by
(auto simp: rcis_def)
        with x(2) ε ab α have "|t| = α ε"
          unfolding α_def by (intro sphere_inter_sphere_pos_real_line_aux2)
auto
        moreover have "t < 0"
          using t(1) α ab by auto
        ultimately have "t = -α ε"
          by simp
        thus "x ∈ {pathstart (p1 ε), pathfinish (p1 ε)}"
          by (auto simp: p1_def rcis_def t rcis_def exp_eq_polar)
      qed
    qed (use ε α ab in ‹auto simp: p1_def simple_path_part_circlepath›)

    show "does_not_cross (p2 ε) (sphere 1 ε)"
    proof (subst does_not_cross_simple_path)
      show "path_image (p2 ε) ∩ sphere 1 ε ⊆ {pathstart (p2 ε), pathfinish
(p2 ε)}"
      proof (intro subsetI; elim IntE)
        fix x assume x: "x ∈ path_image (p2 ε)" "x ∈ sphere 1 ε"
        obtain t where t: "t ∈ {α ε..b}" "x = cis t"
          using x(1) α unfolding p2_def path_image_part_circlepath' by
(auto simp: rcis_def)
        with x(2) ε ab α have "|t| = α ε"
          unfolding α_def by (intro sphere_inter_sphere_pos_real_line_aux2)
auto
        moreover have "t > 0"
          using t(1) α ab by auto
        ultimately have "t = α ε"
          by simp
        thus "x ∈ {pathstart (p2 ε), pathfinish (p2 ε)}"
          by (auto simp: p2_def rcis_def t rcis_def exp_eq_polar)
      qed
    qed (use ε α ab in ‹auto simp: p2_def simple_path_part_circlepath›)

    show "does_not_cross (pm ε) (sphere 1 ε)"
    proof (subst does_not_cross_simple_path)
      show "path_image (pm ε) ∩ sphere 1 ε ⊆ {pathstart (pm ε), pathfinish
(pm ε)}"
      proof (intro subsetI; elim IntE)
        fix x assume x: "x ∈ path_image (pm ε)" "x ∈ sphere 1 ε"
        obtain t where t: "t ∈ {-α ε..α ε}" "x = cis t"
          using x(1) α unfolding pm_def path_image_part_circlepath' by
(auto simp: rcis_def)
        with x(2) ε ab α have "|t| = α ε"
          unfolding α_def by (intro sphere_inter_sphere_pos_real_line_aux2)
auto
        hence "t = α ε ∨ t = -α ε"
```

```
      by (cases "t ≥ 0") auto
    thus "x ∈ {pathstart (pm ε), pathfinish (pm ε)}"
      by (auto simp: pm_def rcis_def t rcis_def exp_eq_polar)
  qed
 qed (use ε α ab in ‹auto simp: pm_def simple_path_part_circlepath›)

 show "- β ε ≠ β ε ∧ |β ε - - β ε| < 2 * pi"
  using arcsin by (auto simp: β_def)

 show "- β ε + 2 * pi ≠ β ε ∧
       |β ε - (- β ε + 2 * pi)| < 2 * pi ∧
       cis (- β ε + 2 * pi) = cis (- β ε) ∧
       cis (β ε) = cis (β ε) ∧ β ε - - β ε + (- β ε + 2 * pi) - β ε
= 2 * pi"
  using arcsin by (auto simp: β_def divide_conv_cnj simp flip: cis_divide
cis_cnj cis_mult)

 show "pathfinish (p1 ε) = 1 + rcis ε (- β ε)"
      "pathstart  (p2 ε) = 1 + rcis ε (β ε)"
      "pathstart  (pm ε) = pathfinish (p1 ε)"
      "pathfinish (pm ε) = pathstart  (p2 ε)"
  by (auto simp: ends)
qed (auto simp: p1_def p2_def)
thus ?thesis using ab
 by (simp add: p_def p1_def cl_def p2_def cr_def α_def β_def
               avoid_part_circlepath_def [abs_def] Let_def)
qed

lemma avoid_part_circlepath_extend_right:
 assumes "a < b ∧ b < c ∨ c < b ∧ b < a" "φ ∈ open_segment a b" "r
> 0"
 assumes ε: "ε > 0" "ε < r * (2 * sin (|b - φ| / 2))"
 shows "avoid_part_circlepath left x r a b φ ε +++ part_circlepath x
r b c ≡p
      avoid_part_circlepath left x r a c φ ε"
proof -
 note ‹ε < r * (2 * sin (|b - φ| / 2))›
 also have "r * (2 * sin (|b - φ| / 2)) ≤ r * (2 * 1)"
  using ‹r > 0› by (intro mult_left_mono sin_le_one) (use assms in auto)
 finally have ε': "ε < 2 * r"
  by simp
 then interpret avoid_part_circlepath_locale x r a b φ ε
  by unfold_locales (use assms in auto)

 have s': "sgn (c - a) = s"
  using assms by (auto simp add: s_def)
 define δ where "δ = (if left = (b < a) then 0 else 2 * s * pi)"
 have eq: "left = (c < a) ⟷ left = (b < a)"
  using assms by auto
```

```
  note defs = s_def [symmetric] s' α_def [symmetric] β_def [symmetric]
δ_def [symmetric] eq

  have "arcsin (ε / (2 * r)) < arcsin (sin (min (pi/2) (|b - φ| / 2)))"
    using assms ε' by (intro arcsin_less_arcsin) (auto simp: field_simps
min_def)
  also have "... = min (pi/2) (|b - φ| / 2)"
    by (subst arcsin_sin) (use pi_gt_zero in ‹auto simp del: pi_gt_zero›)
  also have "... ≤ |b - φ| / 2"
    by linarith
  finally have "α < |b - φ|"
    by (simp add: α_def)
  hence **: "if b < c then φ + s * α < b else φ + s * α > b"
    using assms(1,2) by (auto simp: s_def abs_if open_segment_eq_real_ivl
split: if_splits)

  have *: "rcis r φ + rcis ε (φ + s * β) = rcis r (φ + s * α)"
          "rcis r (φ - s * α) = rcis r φ + rcis ε (φ - s * β + δ)"
    using ends(1)[of left] ends(2) unfolding δ_def bad_def by (simp_all
add: δ_def)

  have "avoid_part_circlepath left x r a b φ ε +++ part_circlepath x
r b c ≡p
        part_circlepath x r a (φ - s * α) +++ part_circlepath (x + rcis
r φ) ε (φ - s * β + δ) (φ + s * β) +++
        part_circlepath x r (φ + s * α) b +++ part_circlepath x r b c"
    unfolding avoid_part_circlepath_def Let_def defs by path (use * in
simp_all)
  also have "... ≡p part_circlepath x r a (φ - s * α) +++
                   part_circlepath (x + rcis r φ) ε (φ - s * β + δ) (φ
+ s * β) +++
                   part_circlepath x r (φ + s * α) c"
    using * ** assms
    by (intro eq_paths_join eq_paths_refl eq_paths_part_circlepaths)
       (auto simp: closed_segment_eq_real_ivl rcis_def exp_eq_polar)
  also have "... = avoid_part_circlepath left x r a c φ ε"
    unfolding avoid_part_circlepath_def Let_def defs by simp
  finally show ?thesis .
qed

lemma avoid_part_circlepath_extend_left:
  assumes "c < a ∧ a < b ∨ c > a ∧ a > b" "φ ∈ open_segment a b" "r
> 0"
  assumes ε: "ε > 0" "ε < r * (2 * sin (|a - φ| / 2))"
  shows   "part_circlepath x r c a +++ avoid_part_circlepath left x r
a b φ ε ≡p
          avoid_part_circlepath left x r c b φ ε"
proof -
  note ‹ε < r * (2 * sin (|a - φ| / 2))›
```

```
  also have "r * (2 * sin (|a - φ| / 2)) ≤ r * (2 * 1)"
    using ‹r > 0› by (intro mult_left_mono sin_le_one) (use assms in auto)
  finally have ε': "ε < 2 * r"
    by simp
  then interpret avoid_part_circlepath_locale x r a b φ ε
    by unfold_locales (use assms in auto)

  have s': "sgn (b - c) = s"
    using assms by (auto simp: s_def)
  define δ where "δ = (if left = (b < a) then 0 else 2 * s * pi)"
  have eq: "left = (b < c) ⟷ left = (b < a)"
    using assms by auto
  note defs = s_def [symmetric] s' α_def [symmetric] β_def [symmetric]
δ_def [symmetric] eq

  have "arcsin (ε / (2 * r)) < arcsin (sin (min (pi/2) (|a - φ| / 2)))"
    using assms ε' by (intro arcsin_less_arcsin) (auto simp: field_simps
min_def)
  also have "... = min (pi/2) (|a - φ| / 2)"
    by (subst arcsin_sin) (use pi_gt_zero in ‹auto simp del: pi_gt_zero›)
  also have "... ≤ |a - φ| / 2"
    by linarith
  finally have "α < |a - φ|"
    by (simp add: α_def)
  hence **: "if c < a then φ - s * α > a else φ - s * α < a"
    using assms(1,2) by (auto simp: s_def abs_if open_segment_eq_real_ivl
split: if_splits)

  have *: "rcis r φ + rcis ε (φ + s * β) = rcis r (φ + s * α)"
          "rcis r (φ - s * α) = rcis r φ + rcis ε (φ - s * β + δ)"
    using ends(1)[of left] ends(2) unfolding δ_def bad_def by (simp_all
add: δ_def)

  have "part_circlepath x r c a +++ avoid_part_circlepath left x r a b
φ ε ≡p
        (part_circlepath x r c a +++ part_circlepath x r a (φ - s * α))
+++
        (part_circlepath (x + rcis r φ) ε (φ - s * β + δ) (φ + s * β)
+++
         part_circlepath x r (φ + s * α) b)"
    unfolding avoid_part_circlepath_def Let_def defs by path (use * in
simp_all)
  also have "... ≡p part_circlepath x r c (φ - s * α) +++
                   part_circlepath (x + rcis r φ) ε (φ - s * β + δ) (φ
+ s * β) +++
                   part_circlepath x r (φ + s * α) b"
    using * ** assms
    by (intro eq_paths_join eq_paths_refl eq_paths_part_circlepaths)
       (auto simp: closed_segment_eq_real_ivl rcis_def exp_eq_polar)
```

```
  also have "... = avoid_part_circlepath left x r c b φ ε"
    unfolding avoid_part_circlepath_def Let_def defs by simp
  finally show ?thesis .
qed

lemma avoid_part_circlepath_reverse:
  assumes "a ≠ b" "ε > 0" "ε < 2 * r"
  shows "reversepath (avoid_part_circlepath left x r a b φ ε) ≡p
         avoid_part_circlepath (¬left) x r b a φ ε"
proof -
  interpret avoid_part_circlepath_locale x r a b φ ε
    by unfold_locales (use assms in auto)

  have s: "sgn (a - b) = -s"
    using assms by (auto simp: s_def sgn_if)
  define δ where "δ = (if left = (b < a) then 0 else 2 * s * pi)"
  define δ' where "δ' = (if (¬left) ≠ (a < b) then 0 else 2 * (-s) *
pi)"
  note defs = s_def [symmetric] s α_def [symmetric] β_def [symmetric]
              δ_def [symmetric] δ'_def [symmetric]

  have *: "rcis r (φ - s * α) = rcis r φ + rcis ε (φ - s * β + δ)"
          "rcis r φ + rcis ε (φ + s * β) = rcis r (φ + s * α)"
    using ends(1)[of left] ends(2) unfolding δ_def bad_def by (simp_all
add: δ_def)
  have "reversepath (avoid_part_circlepath left x r a b φ ε) ≡p
          part_circlepath x r b (φ + s * α) +++
          part_circlepath (x + rcis r φ) ε (φ + s * β) (φ - s * β + δ)
+++
          part_circlepath x r (φ - s * α) a"
    unfolding avoid_part_circlepath_def Let_def defs by path (use * in
simp_all)
  also have "... = avoid_part_circlepath (¬left) x r b a φ ε"
    unfolding avoid_part_circlepath_def Let_def defs
    by (intro arg_cong2[of _ _ _ _ "(+++)"] part_circlepath_cong refl)
       (simp_all flip: cis_mult add: δ'_def δ_def s_def sgn_if)
  finally show ?thesis .
qed


lemma detour_rel_part_circlepath_semicircle_left_aux2:
  assumes ab: "a < 0" "0 < b"
  shows "detour_rel {1} {} (part_circlepath 0 1 a b)
           (avoid_part_circlepath True  0 1 a b 0)
           (avoid_part_circlepath False 0 1 a b 0)"
proof -
  let ?γ = "λleft a b. avoid_part_circlepath left 0 1 a b 0"
  define a' where "a' = max a (-pi/2)"
  define b' where "b' = min b (pi/2)"
```

```
  define p1 where "p1 = part_circlepath 0 1 a' b'"
  define p2 where "p2 = part_circlepath 0 1 a b'"
  define p3 where "p3 = part_circlepath 0 1 a b"

  have less: "a' < b'" "a < b'"
    using ab pi_gt_zero by (auto simp: a'_def b'_def simp del: pi_gt_zero)

  have 1: "detour_rel {1} {} p1
             (λε. ?γ True a' b' ε) (λε. ?γ False a' b' ε)"
    using ab unfolding p1_def a'_def b'_def
    by (intro detour_rel_part_circlepath_semicircle_left_aux1)
       (auto simp: min_le_iff_disj le_max_iff_disj)

  have 2: "detour_rel {1} {} p2 (λε. ?γ True a b' ε) (λε. ?γ False a b'
ε)"
  proof (cases "a = a'")
    case False
    hence "a < a'"
      by (auto simp: a'_def)

    have "detour_rel ({} ∪ {1}) ({} ∪ {}) (part_circlepath 0 1 a a' +++
p1)
            (λε. part_circlepath 0 1 a a' +++ ?γ True a' b' ε)
            (λε. part_circlepath 0 1 a a' +++ ?γ False a' b' ε)"
      by (intro detour_rel_join 1 detour_rel_refl) (auto simp: p1_def)
    thus ?thesis
    proof (rule detour_rel_congI)
      have "2 * sin (|a'| / 2) > 0" using assms
        by (intro mult_pos_pos sin_gt_zero) (auto simp: a'_def max_def)
      hence "eventually (λx. x < 2 * sin (|a'| / 2)) (at_right 0)"
        using eventually_at_right_field by blast
      moreover have "eventually (λx :: real. x > 0) (at_right 0)"
        by (simp add: eventually_at_right_less)
      ultimately have "∀_F ε in at_right 0. part_circlepath 0 1 a a' +++
?γ left a' b' ε ≡_p ?γ left a b' ε"
        (is "?P left") for left
      proof eventually_elim
        case (elim ε)
        have "a' < 0" "b' > 0"
          using False a'_def b'_def pi_gt_zero assms by (auto simp: min_def
max_def)
        moreover have "b' - a' ≤ pi"
          by (auto simp: b'_def a'_def)
        ultimately show ?case using ‹a < a'› ‹a' < b'› elim
          by (intro avoid_part_circlepath_extend_left)
             (use assms in ‹auto simp: open_segment_eq_real_ivl›)
      qed
      from this[of True] and this[of False] show "?P True" "?P False"
```

```
.
      show "part_circlepath 0 1 a a' +++ p1 ≡p p2"
        unfolding p1_def p2_def
        by (intro eq_paths_part_circlepaths)
           (use ‹a < a'› ‹a' < b'› in ‹auto simp: closed_segment_eq_real_ivl›)
    qed (use less in ‹auto simp: p1_def intro!: valid_path_avoid_part_circlepath'›)
  qed (use 1 in ‹simp_all add: a'_def p1_def p2_def›)

  show "detour_rel {1} {} p3 (λε. ?γ True a b ε) (λε. ?γ False a b ε)"
  proof (cases "b = b'")
    case False
    hence "b > b'"
      by (auto simp: b'_def)
    have "detour_rel ({1} ∪ {}) ({} ∪ {}) (p2 +++ part_circlepath 0 1
b' b)
            (λε. ?γ True a b' ε +++ part_circlepath 0 1 b' b)
            (λε. ?γ False a b' ε +++ part_circlepath 0 1 b' b)"
      by (intro detour_rel_join 2 detour_rel_refl) (auto simp: p2_def)
    thus ?thesis
    proof (rule detour_rel_congI)
      have "2 * sin (|b'| / 2) > 0" using assms
        by (intro mult_pos_pos sin_gt_zero) (auto simp: b'_def min_def)
      hence "eventually (λx. x < 2 * sin (|b'| / 2)) (at_right 0)"
        using eventually_at_right_field by blast
      moreover have "eventually (λx :: real. x > 0) (at_right 0)"
        by (simp add: eventually_at_right_less)
      ultimately have "∀F ε in at_right 0. ?γ left a b' ε +++ part_circlepath
0 1 b' b ≡p ?γ left a b ε"
        (is "?P left") for left
      proof eventually_elim
        case (elim ε)
        have "b' > 0"
          using assms by (auto simp: b'_def)
        thus ?case using ‹b > b'› ‹a < b'› elim
          by (intro avoid_part_circlepath_extend_right)
             (use assms in ‹auto simp: open_segment_eq_real_ivl›)
      qed
      from this[of True] and this[of False] show "?P True" "?P False"
.
      show "p2 +++ part_circlepath 0 1 b' b ≡p p3"
        unfolding p2_def p3_def
        by (intro eq_paths_part_circlepaths)
           (use ‹b > b'› ‹a < b'› in ‹auto simp: closed_segment_eq_real_ivl›)
    qed (use assms in ‹auto simp: p2_def intro!: valid_path_avoid_part_circlepath'›)
  qed (use 2 in ‹simp_all add: a'_def p2_def p3_def›)
qed

lemma detour_rel_part_circlepath_semicircle_left_aux3:
  assumes ab: "0 ∈ open_segment a b"
```

```
  shows "detour_rel {1} {} (part_circlepath 0 1 a b)
           (avoid_part_circlepath True  0 1 a b 0)
           (avoid_part_circlepath False 0 1 a b 0)"
proof (cases "a < b")
  case True
  thus ?thesis
    using detour_rel_part_circlepath_semicircle_left_aux2[of a b] assms
    by (simp add: open_segment_eq_real_ivl)
next
  case False
  hence "a > b"
    using ab by (auto simp: open_segment_eq_real_ivl split: if_splits)
  let ?γ = "λleft a b. avoid_part_circlepath left 0 1 a b 0"
  have "detour_rel {1} {} (part_circlepath 0 1 b a) (?γ True b a) (?γ
False b a)"
    using detour_rel_part_circlepath_semicircle_left_aux2[of b a] assms
<a > b>
    by (simp add: open_segment_eq_real_ivl)
  note detour_rel_swap[OF detour_rel_reverse [OF this]]
  thus ?thesis
  proof (rule detour_rel_congI)
    have "eventually (λε :: real. ε > 0 ∧ ε < 2) (at_right 0)"
      using eventually_at_right_field zero_less_numeral by blast
    hence *: "∀F ε in at_right 0. reversepath (?γ left b a ε) ≡p ?γ
(¬left) a b ε"
      (is "?P left") for left
    proof eventually_elim
      case (elim ε)
      thus ?case using <a > b>
        by (intro avoid_part_circlepath_reverse) (use assms in auto)
    qed

    show "∀F ε in at_right 0.
       reversepath (avoid_part_circlepath True 0 1 b a 0 ε) ≡p
       avoid_part_circlepath False 0 1 a b 0 ε"
      using *[of True] by simp
    show "∀F ε in at_right 0.
       reversepath (avoid_part_circlepath False 0 1 b a 0 ε) ≡p
       avoid_part_circlepath True 0 1 a b 0 ε"
      using *[of False] by simp
  qed (use ab in <auto intro!: valid_path_avoid_part_circlepath'>)
qed

lemma detour_rel_part_circlepath_semicircle_left_aux4:
  assumes "φ ∈ open_segment a b" "bad = rcis r φ" "r > 0"
  shows "detour_rel {bad} {} (part_circlepath 0 r a b)
           (avoid_part_circlepath True  0 r a b φ)
           (avoid_part_circlepath False 0 r a b φ)"
proof -
```

```
  {
    fix left :: bool
    have "(λε. (*) bad ∘ avoid_part_circlepath left 0 1 (a - φ) (b -
φ) 0 (ε / norm bad)) =
               avoid_part_circlepath left 0 r (a - φ + Arg (rcis r φ))
               (b - φ + Arg (rcis r φ)) (Arg (rcis r φ))"
      (is "?l = _") using assms by (subst avoid_part_circlepath_mult)
simp_all
    also have "... = avoid_part_circlepath left 0 r a b φ"
      using ‹r > 0›
      by (intro avoid_part_circlepath_cong)
         (auto simp: rcis_def cis_Arg simp flip: cis_mult cis_divide)
    finally have "?l = ..." .
  } note eq = this

  have "0 ∈ open_segment (a - φ) (b - φ)"
    using assms by (auto simp: open_segment_eq_real_ivl)
  have "detour_rel {1} {} (part_circlepath 0 1 (a - φ) (b - φ))
          (avoid_part_circlepath True 0 1 (a - φ) (b - φ) 0)
          (avoid_part_circlepath False 0 1 (a - φ) (b - φ) 0)"
    by (rule detour_rel_part_circlepath_semicircle_left_aux3) fact
  hence "detour_rel ((*) bad ‘ {1}) ((*) bad ‘ {}) ((*) bad ∘ part_circlepath
0 1 (a - φ) (b - φ))
          (λε. (*) bad ∘ avoid_part_circlepath True 0 1 (a - φ) (b - φ)
0 (ε / norm bad))
          (λε. (*) bad ∘ avoid_part_circlepath False 0 1 (a - φ) (b -
φ) 0 (ε / norm bad))"
    by (rule detour_rel_mult) (use assms in auto)
  also have "(*) bad ∘ part_circlepath 0 1 (a - φ) (b - φ) =
             part_circlepath 0 (cmod bad) (a - φ + Arg bad) (b - φ +
Arg bad)"
    by (simp add: part_circlepath_mult_complex)
  also have "... = part_circlepath 0 r a b"
    using assms by (intro part_circlepath_cong)
                   (auto simp: rcis_def norm_mult cis_Arg simp flip: cis_mult
cis_divide)
  also note eq[of True]
  also note eq[of False]
  finally show ?thesis by simp
qed




lemma detour_rel_part_circlepath_semicircle_left [detour_rel_intros]:
  assumes "φ ∈ open_segment a b" "bad = x + rcis r φ" "r > 0"
  shows "detour_rel {bad} {} (part_circlepath x r a b)
          (avoid_part_circlepath True  x r a b φ)
```

```
          (avoid_part_circlepath False x r a b φ)"
proof -
  have "detour_rel {bad - x} {} (part_circlepath 0 r a b)
          (avoid_part_circlepath True  0 r a b φ)
          (avoid_part_circlepath False 0 r a b φ)"
    by (rule detour_rel_part_circlepath_semicircle_left_aux4) (use assms
in auto)
  hence "detour_rel ((+) x ‘ {bad - x}) ((+) x ‘ {}) ((+) x ∘ part_circlepath
0 r a b)
          (λε. (+) x ∘ avoid_part_circlepath True  0 r a b φ ε)
          (λε. (+) x ∘ avoid_part_circlepath False 0 r a b φ ε)"
    by (rule detour_rel_translate)
  thus ?thesis
    by (simp add: part_circlepath_translate avoid_part_circlepath_translate)
qed

lemma detour_rel_part_circlepath_semicircle_right [detour_rel_intros]:
  assumes "φ ∈ open_segment a b" "bad = x + rcis r φ" "r > 0"
  shows "detour_rel {} {bad} (part_circlepath x r a b)
          (avoid_part_circlepath False x r a b φ)
          (avoid_part_circlepath True  x r a b φ)"
  using detour_rel_part_circlepath_semicircle_left[OF assms] by (rule
detour_rel_swap)
```

4.4.4 Line–line corner

```
definition avoid_linepath_linepath where
  "avoid_linepath_linepath left a b c ε = (
     let s = sgn (Arg (c - b) - Arg (a - b));
         δ = (if left ⟷ Arg (a - b) ≥ Arg (c - b) then 0 else 2 * s
* pi)
     in  linepath a (b + ε *R sgn (a - b)) +++
         part_circlepath b ε (Arg (a - b) + δ) (Arg (c - b)) +++
         linepath (b + ε *R sgn (c - b)) c)"

lemma avoid_linepath_linepath_translate:
  "(+) d ∘ avoid_linepath_linepath left a b c ε =
   avoid_linepath_linepath left (d + a) (d + b) (d + c) ε"
  unfolding avoid_linepath_linepath_def path_compose_join linepath_translate

            part_circlepath_translate Let_def by (simp add: algebra_simps)

lemma avoid_linepath_linepath_mult:
  assumes "a ≠ 0" "b ≠ 0"
  shows   "(*) c ∘ avoid_linepath_linepath left a 0 b ε =
          avoid_linepath_linepath left (c * a) 0 (c * b) (norm c * ε)"
proof (cases "c = 0")
  assume "c = 0"
  thus ?thesis
```

```
    by (auto simp: avoid_linepath_linepath_def joinpaths_def
                   fun_eq_iff part_circlepath_def linepath_def)
next
  assume [simp]: "c ≠ 0"
  define β where "β = (if left = (Arg b ≤ Arg a) then 0 else 2 * sgn
(Arg b - Arg a) * pi)"
  define β' where "β' = (if left = (Arg (c * b) ≤ Arg (c * a)) then 0
else 2 * sgn (Arg (c * b) - Arg (c * a)) * pi)"
  have [simp]: "cis β = 1" "cis β' = 1"
    by (auto simp: β_def β'_def sgn_if)

  have "(*) c ∘ avoid_linepath_linepath left a 0 b ε =
          linepath (c * a) (c * (ε *R sgn a)) +++
          part_circlepath 0 (norm c * ε) (Arg a + β + Arg c) (Arg b +
Arg c) +++
          linepath (c * (ε *R sgn b)) (c * b)"
    unfolding avoid_linepath_linepath_def path_compose_join linepath_mult_complex

              part_circlepath_mult_complex Let_def β_def by simp
  also have "part_circlepath 0 (norm c * ε) (Arg a + β + Arg c) (Arg b
+ Arg c) =
             part_circlepath 0 (norm c * ε) (Arg (c * a) + β') (Arg (c
* b))"
  proof (rule part_circlepath_cong)
    show "cis (Arg (c * a) + β') = cis (Arg a + β + Arg c)"
      using assms by (auto simp flip: cis_mult simp: cis_Arg sgn_mult)
    have "Arg (c * b) - Arg (c * a) = Arg b - Arg a + β' - β"
      using assms Arg_bounded[of a] Arg_bounded[of b] Arg_bounded[of c]
      by (simp add: Arg_times' β'_def β_def)
    thus "Arg (c * b) = Arg (c * a) + β' + (Arg b + Arg c) - (Arg a +
β + Arg c)"
      by simp
  qed auto
  also have "linepath (c * a) (c * (ε *R sgn a)) +++ ... +++ linepath
(c * (ε *R sgn b)) (c * b) =
               linepath (c * a) ((norm c * ε) *R sgn (c * a)) +++
               part_circlepath 0 (norm c * ε) (Arg (c * a) + β') (Arg
(c * b)) +++
               linepath ((norm c * ε) *R sgn (c * b)) (c * b)"
    by (simp add: algebra_simps scaleR_conv_of_real complex_sgn_def norm_mult)
  also have "... = avoid_linepath_linepath left (c * a) 0 (c * b) (norm
c * ε)"
     unfolding avoid_linepath_linepath_def path_compose_join linepath_mult_complex

               part_circlepath_mult_complex Let_def β'_def by simp
   finally show ?thesis .
qed
```

```
lemma norm_linepath_0: "norm (linepath 0 a x) = |x| * norm a"
  by (simp add: linepath_def)

lemma norm_linepath_0': "norm (linepath a 0 x) = |1 - x| * norm a"
  by (simp add: linepath_def)

lemma detour_rel_avoid_linepath_linepath_aux1:
  assumes "1 ∉ closed_segment 0 c" "c ∉ closed_segment 0 1" "Arg c >
0"
  shows   "detour_rel {0} {} (linepath 1 0 +++ linepath 0 c)
             (avoid_linepath_linepath True 1 0 c) (avoid_linepath_linepath
False 1 0 c)"
proof -
  define p1 where "p1 = (λε. linepath 1 (of_real ε) :: real ⇒ complex)"
  define p2 where "p2 = (λε. linepath (ε *R sgn c) c :: real ⇒ complex)"
  define pm1 where "pm1 = (λε. linepath (of_real ε) 0 :: real ⇒ complex)"
  define pm2 where "pm2 = (λε. linepath 0 (ε *R sgn c) :: real ⇒ complex)"
  define cl where "cl = (λε. part_circlepath 0 ε (2 * pi) (Arg c))"
  define cr where "cr = (λε. part_circlepath 0 ε 0 (Arg c))"

  have [simp]: "c ≠ 0"
    using assms by auto
  hence [simp]: "sgn c ≠ 0"
    by (simp add: sgn_zero_iff)
  note [simp] = closed_segment_same_Im closed_segment_eq_real_ivl

  have "detour_rel {0} {} (linepath 1 0 +++ linepath 0 c)
          (λε. p1 ε +++ cl ε +++ p2 ε) (λε. p1 ε +++ cr ε +++ p2 ε)"
    unfolding cl_def cr_def
  proof (rule detour_rel_avoid_basic_part_circlepath_left [where eps
= "min 1 (norm c)"])
    fix ε assume ε: "ε > 0" "ε < min 1 (norm c)"

    have "ε *R sgn c ≠ c"
      using ε by (auto simp: scaleR_conv_of_real complex_sgn_def field_simps)
    have "ε *R sgn c = linepath 0 c (ε / norm c)"
      by (auto simp: linepath_def complex_sgn_def field_simps)
    also have "... ∈ closed_segment 0 c"
      using ε by (subst in_segment) auto
    finally have "ε *R sgn c ∈ closed_segment 0 c" .
    hence "linepath 1 0 +++ linepath 0 c ≡p (p1 ε +++ pm1 ε) +++ (pm2
ε +++ p2 ε)"
      unfolding p1_def pm1_def pm2_def p2_def using ε ‹ε *R sgn c ≠ c›
      by (intro eq_paths_join eq_paths_linepaths') auto
    also have "... ≡p p1 ε +++ (pm1 ε +++ pm2 ε) +++ p2 ε"
      unfolding p1_def pm1_def pm2_def p2_def by path
    finally show "linepath 1 0 +++ linepath 0 c ≡p p1 ε +++ (pm1 ε +++
pm2 ε) +++ p2 ε" .
```

```
    have *: "Im z ≠ 0 ∨ Re z < 0" if "z ∈ closed_segment (ε *R sgn c)
c" for z
    proof (cases "Im c = 0")
      case False
      thus ?thesis using ε
        using in_closed_segment_imp_Im_in_closed_segment[OF that]
        by (auto split: if_splits simp: field_simps mult_le_0_iff zero_le_mult_iff)
    next
      case True
      hence "Re c < 0"
        using assms(3) by (simp add: Arg_pos_iff)
      have "Re z ∈ closed_segment (Re (ε *R sgn c)) (Re c)"
        by (intro in_closed_segment_imp_Re_in_closed_segment that)
      have "closed_segment (ε *R sgn c) c ⊆ {z. Re z < 0}"
        by (intro closed_segment_subset)
           (use ‹Re c < 0› ε in ‹auto simp: scaleR_conv_of_real field_simps
                                  mult_pos_neg mult_neg_pos convex_halfspace_Re_lt›)
      thus ?thesis
        using that by auto
    qed

    show "path_image (p1 ε) ∩ path_image (p2 ε)
         ⊆ {pathstart (linepath 1 0 +++ linepath 0 c)} ∩
            {pathfinish (linepath 1 0 +++ linepath 0 c)}"
      using ε by (auto simp: p1_def p2_def complex_eq_iff dest: *)

    show "0 ≠ Arg c ∧ |Arg c - 0| < 2 * pi"
         "2 * pi ≠ Arg c ∧ |Arg c - 2 * pi| < 2 * pi ∧ cis (2 * pi) =
cis 0 ∧
         cis (Arg c) = cis (Arg c) ∧ Arg c - 0 + 2 * pi - Arg c = 2 *
pi"
      using assms Arg_bounded[of c] by (auto)

    show "does_not_cross (p1 ε) (sphere 0 ε)"
      using ε by (subst does_not_cross_simple_path) (auto simp: p1_def
complex_eq_iff cmod_eq_Re)
    show "does_not_cross (pm1 ε +++ pm2 ε) (sphere 0 ε)" using ε
      by (auto simp: does_not_cross_def joinpaths_def pm1_def pm2_def

                     abs_mult norm_sgn norm_linepath_0')
    show "does_not_cross (p2 ε) (sphere 0 ε)"
      unfolding does_not_cross_def
    proof
      fix t :: real assume t: "t ∈ {0<..<1}"
      have "norm (p2 ε t) = norm (linepath (ε *R sgn c) c t)"
        by (simp add: p2_def)
      also have "linepath (ε *R sgn c) c t = ((1 - t) * (ε * inverse (cmod
c)) + t) *R c"
        by (simp add: linepath_def complex_sgn_def algebra_simps)
```

```
      also have "norm ... = ((1 - t) * (ε * inverse (norm c)) + t) * norm
c"
        using t ε by (subst norm_scaleR) simp_all
      also have "... = ε + t * (norm c - ε)"
        by (auto simp: algebra_simps)
      also have "... ≠ ε"
        using ε t by auto
      finally show "p2 ε t ∉ sphere 0 ε"
        by simp
    qed
  qed (auto simp: p1_def p2_def pm1_def pm2_def path_image_join complex_sgn_def

                  rcis_def scaleR_conv_of_real field_simps cis_Arg)
  thus ?thesis
    using ‹Arg c > 0›
    by (simp add: p1_def p2_def cl_def cr_def avoid_linepath_linepath_def
[abs_def] scaleR_conv_of_real)
qed


lemma detour_rel_avoid_linepath_linepath_aux2:
  assumes "1 ∉ closed_segment 0 c" "c ∉ closed_segment 0 1"
  shows   "detour_rel {0} {} (linepath 1 0 +++ linepath 0 c)
             (avoid_linepath_linepath True 1 0 c) (avoid_linepath_linepath
False 1 0 c)"
proof (cases "Arg c > 0")
  case True
  thus ?thesis
    using detour_rel_avoid_linepath_linepath_aux1[of c] assms by simp
next
  case False
  have "Arg c ≠ 0"
  proof
    assume "Arg c = 0"
    hence "c ∈ ℝ" "Re c ≥ 0"
      by (auto simp: Arg_eq_0)
    thus False using assms
      by (auto simp: complex_is_Real_iff closed_segment_same_Im closed_segment_eq_real_ivl)
  qed
  with False have "Arg c < 0"
    by auto
  have [simp]: "Arg (cnj c) = -Arg c"
    using ‹Arg c < 0› by (auto simp: Arg_cnj elim!: Reals_cases split:
if_splits)
  have [simp]: "cnj (sgn (cnj c)) = sgn c"
    by (auto simp: complex_sgn_def)

  have *: "detour_rel {0} {} (linepath 1 0 +++ linepath 0 (cnj c))
             (avoid_linepath_linepath True 1 0 (cnj c)) (avoid_linepath_linepath
```

```
False 1 0 (cnj c))"
  proof (rule detour_rel_avoid_linepath_linepath_aux1)
    have "cnj 1 ∉ closed_segment (cnj 0) (cnj c)"
      by (subst closed_segment_cnj) (use assms in auto)
    thus "1 ∉ closed_segment 0 (cnj c)"
      by simp
  next
    have "cnj c ∉ closed_segment (cnj 0) (cnj 1)"
      by (subst closed_segment_cnj) (use assms in auto)
    thus "cnj c ∉ closed_segment 0 1"
      by simp
  qed (use ‹Arg c < 0› in auto)

  have "detour_rel {} {0} (linepath 1 0 +++ linepath 0 c)
        (λε. cnj ∘ avoid_linepath_linepath True 1 0 (cnj c) ε)
        (λε. cnj ∘ avoid_linepath_linepath False 1 0 (cnj c) ε)"
    using detour_rel_cnj[OF *] by (simp add: path_compose_join linepath_cnj')
  also have "(λε. cnj ∘ avoid_linepath_linepath True 1 0 (cnj c) ε) =

              avoid_linepath_linepath False 1 0 c" using ‹Arg c < 0›
    by (auto simp: avoid_linepath_linepath_def path_compose_join linepath_cnj'
                   part_circlepath_cnj' fun_eq_iff)
  also have "(λε. cnj ∘ avoid_linepath_linepath False 1 0 (cnj c) ε) =

              avoid_linepath_linepath True 1 0 c" using ‹Arg c < 0›
    by (auto simp: avoid_linepath_linepath_def path_compose_join linepath_cnj'
                   part_circlepath_cnj' fun_eq_iff)
  finally have "detour_rel {} {0} (linepath 1 0 +++ linepath 0 c)
                (avoid_linepath_linepath False 1 0 c)
                (avoid_linepath_linepath True 1 0 c)" .
  from detour_rel_swap[OF this] show ?thesis
    by simp
qed

lemma detour_rel_avoid_linepath_linepath_aux3:
  assumes "a ∉ closed_segment 0 c" "c ∉ closed_segment a 0"
  shows   "detour_rel {0} {} (linepath a 0 +++ linepath 0 c)
             (avoid_linepath_linepath True a 0 c) (avoid_linepath_linepath
False a 0 c)"
proof -
  define c' where "c' = c / a"
  have "a ≠ 0" "c ≠ 0"
    using assms by auto
  hence c_eq: "c = a * c'"
    by (auto simp: c'_def)

  have *: "detour_rel {0} {} (linepath 1 0 +++ linepath 0 c')
             (avoid_linepath_linepath True 1 0 c') (avoid_linepath_linepath
False 1 0 c')"
```

```
  proof (rule detour_rel_avoid_linepath_linepath_aux2)
    show "1 ∉ closed_segment 0 c'"
      using assms by (simp add: c_eq in_segment(1) scaleR_conv_of_real)
    show "c' ∉ closed_segment 0 1"
    proof
      assume "c' ∈ closed_segment 0 1"
      hence "a * c' ∈ closed_segment (a * 0) (a * 1)"
        by (subst closed_segment_mult [symmetric]) auto
      thus False
        using assms by (simp add: c_eq closed_segment_commute)
    qed
  qed

  show ?thesis
    using detour_rel_mult[OF *, of a] ‹a ≠ 0› ‹c ≠ 0›
    by (simp add: path_compose_join linepath_mult_complex c_eq avoid_linepath_linepath_mult
qed

lemma detour_rel_avoid_linepath_linepath_left [detour_rel_intros]:
  assumes "a ∉ closed_segment b c" "c ∉ closed_segment a b"
  shows   "detour_rel {b} {} (linepath a b +++ linepath b c)
             (avoid_linepath_linepath True a b c) (avoid_linepath_linepath
False a b c)"
proof -
  have *: "detour_rel {0} {} (linepath (a - b) 0 +++ linepath 0 (c - b))
               (avoid_linepath_linepath True (a - b) 0 (c - b))
               (avoid_linepath_linepath False (a - b) 0 (c - b))"
  proof (rule detour_rel_avoid_linepath_linepath_aux3)
    show "a - b ∉ closed_segment 0 (c - b)"
      using assms(1) closed_segment_translation_eq[of b "a - b" 0 "c -
b"] by auto
    show "c - b ∉ closed_segment (a - b) 0"
      using assms(2) closed_segment_translation_eq[of b "c - b" "a - b"
0] by auto
  qed
  show ?thesis
    using detour_rel_translate[OF *, of b]
    by (simp add: path_compose_join linepath_translate avoid_linepath_linepath_translate)
qed

lemma detour_rel_avoid_linepath_linepath_right [detour_rel_intros]:
  assumes "a ∉ closed_segment b c" "c ∉ closed_segment a b"
  shows   "detour_rel {} {b} (linepath a b +++ linepath b c)
             (avoid_linepath_linepath False a b c) (avoid_linepath_linepath
True a b c)"
  using detour_rel_swap[OF detour_rel_avoid_linepath_linepath_left[OF
assms]] by simp
```

4.4.5 Line–arc corner

Avoidance pattern for a bad point lying on the junction between a straight vertical line coming from above to the point $e^{2i\pi/3}$ and a circular arc with radius 1 around the origin that continues from there to the right.

The bad point is avoided by cutting out a circle of radius ε around it from the path and replacing the removed section with a circular arc of radius ε around the bad point.

```
definition avoid_linepath_circlepath where
  "avoid_linepath_circlepath left y a ε =
     (let α = 2 * arcsin (ε / 2);
          bad = cis (2*pi/3);
          β = pi / 3 + arcsin (ε / 2)
      in  linepath (-1/2 + y *R i) (bad + ε *R i) +++
          part_circlepath bad ε (pi/2) (pi/2 - β + (if left then 0 else
2 * pi)) +++
          part_circlepath 0 1 (2*pi/3 - α) a)"

lemma cis_double: "cis (2 * x) = cis x ^ 2"
  by (simp add: complex_eq_iff cos_double sin_double power2_eq_square)

lemma valid_path_avoid_linepath_circlepath:
  assumes "ε ≥ 0" "ε ≤ 2"
  shows   "valid_path (avoid_linepath_circlepath left y a ε)"
proof -
  have "(ε / 2) ^ 2 ≤ (2 / 2) ^ 2"
    by (intro power_mono) (use assms in auto)
  thus ?thesis using assms
  unfolding avoid_linepath_circlepath_def Let_def
  apply (intro valid_path_join valid_path_linepath valid_path_part_circlepath)
   apply (auto simp: rcis_def scaleR_conv_of_real cis_double divide_conv_cnj
norm_power rcis_def exp_eq_polar
               simp flip: cis_divide cis_mult)
  apply (auto simp: complex_eq_iff cos_arcsin power2_eq_square cos_120
sin_120 sin_30 cos_30)
  apply (auto simp: field_simps simp del: div_mult_self3 div_mult_self4
div_mult_self2 div_mult_self1)
  done
qed

locale avoid_linepath_circlepath_locale =
  fixes y a ε :: real
  assumes ε: "ε ∈ {0<..<2}"
begin

definition α where "α = 2 * arcsin (ε / 2)"
definition bad where "bad = cis (2 * pi / 3)"
definition β where "β = pi / 3 + arcsin (ε / 2)"
```

```
lemma ends: "bad + ε *R i = bad + rcis ε (pi / 2)"
            "bad + rcis ε (pi/2 - β + (if left then 2 * pi else 0)) =
cis (2*pi/3 - α)"
proof -
  note [simp del] = div_mult_self3 div_mult_self4 div_mult_self2 div_mult_self1
  show "bad + ε *R i = bad + rcis ε (pi / 2)"
    by (auto simp: rcis_def scaleR_conv_of_real)
  have "(ε / 2) ^ 2 ≤ 1 ^ 2"
    using ε by (intro power_mono) auto
  hence "cis (2 * pi / 3) + complex_of_real ε * (cis (pi / 6) * cnj (cis
(arcsin (ε / 2)))) =
          cis (2 * pi / 3) * cnj (cis (2 * arcsin (ε / 2)))" using ε
    by (auto simp: complex_eq_iff cos_120 sin_120 cos_60 sin_60 cos_30
sin_30 cos_double sin_double cos_arcsin real_sqrt_divide
                   field_simps power2_eq_square sin_120' cos_120')
  hence "bad + rcis ε (pi/2 - β) = cis (2*pi/3 - α)"
    by (auto simp: rcis_def bad_def divide_conv_cnj α_def β_def simp
flip: cis_divide)
  thus "bad + rcis ε (pi/2 - β + (if left then 2 * pi else 0)) = cis (2*pi/3
- α)"
    unfolding rcis_def by (subst cis_mult [symmetric]) auto
qed

end


lemma detour_rel_avoid_linepath_circlepath_left:
  fixes a b c :: real
  assumes "y > sqrt 3 / 2" "a > pi / 2" "a < 2 * pi / 3"
  defines "bad ≡ cis (2 * pi / 3)"
  shows   "detour_rel {bad} {} (linepath (-1/2 + y *R i) bad +++ part_circlepath
0 1 (2*pi/3) a)
             (avoid_linepath_circlepath True y a) (avoid_linepath_circlepath
False y a)"
proof -
  define α where "α = (λε. 2 * arcsin (ε/2))"
  define β where "β = (λε. pi / 3 + arcsin (ε / 2))"
  define p where "p = linepath (-1/2 + y *R i) bad +++ part_circlepath
0 1 (2*pi/3) a"
  define p1 where "p1 = (λε. linepath (-1/2 + y *R i) (bad + ε *R i))"
  define pm1 where "pm1 = (λε. linepath (bad + ε *R i) bad)"
  define pm2 where "pm2 = (λε. part_circlepath 0 1 (2*pi/3) (2*pi/3 -
α ε))"
  define p2 where "p2 = (λε. part_circlepath 0 1 (2*pi/3 - α ε) a)"
  define cl where "cl = (λε. part_circlepath bad ε (pi/2) (pi/2 - β ε))"
  define cr where "cr = (λε. part_circlepath bad ε (pi/2) (pi/2 - β ε
+ 2 * pi))"
```

```
  note [simp] = closed_segment_eq_real_ivl open_segment_eq_real_ivl closed_segment_same_Re
               sin_30 cos_30 sin_60 cos_60 sin_120 cos_120 sin_120' cos_120'
  have [simp]: "rcis 1 = cis"
    by (simp add: rcis_def fun_eq_iff)

  have "sin (pi - a) < sin (pi / 2)"
    by (subst sin_mono_less_eq) (use assms in auto)
  hence "sin a < 1"
    by simp
  have "sin (pi - a) > sin (pi / 3)"
    by (subst sin_mono_less_eq) (use assms in auto)
  hence "sin a > sqrt 3 / 2"
    by simp

  define eps where "eps = Min {1, y - sqrt 3 / 2, sin (pi / 3 - a / 2)
* 2}"

  note [[goals_limit = 20]]
  have "detour_rel {bad} {} p (λε. p1 ε +++ cl ε +++ p2 ε) (λε. p1 ε +++
cr ε +++ p2 ε)"
    unfolding cl_def cr_def
  proof (rule detour_rel_avoid_basic_part_circlepath_left [where eps
= eps])
    show "eps > 0"
      unfolding eps_def using assms by (auto intro!: sin_gt_zero)
  next
    show "bad ∈ path_image p - {pathstart p, pathfinish p}"
      using assms ‹sin a > sqrt 3 / 2›
      by (auto simp: p_def path_image_join rcis_def complex_eq_iff rcis_def
exp_eq_polar)
  next
    fix ε :: real
    assume "ε > 0" "ε < eps"
    hence ε: "ε > 0" "ε < 1" "ε < y - sqrt 3 / 2" "ε < sin (pi / 3 -
a / 2) * 2"
      unfolding eps_def by (auto simp: min_less_iff_disj)
    assume simple: "simple_path p"

    have "arcsin (ε / 2) < arcsin (1 / 2)"
      using ε by (intro arcsin_less_arcsin) auto
    hence arcsin: "arcsin (ε / 2) > 0" "arcsin (ε / 2) < pi / 6"
      using ε by (auto intro!: arcsin_pos)

    have "arcsin (ε / 2) < arcsin (sin (pi / 3 - a / 2))"
      using ε by (intro arcsin_less_arcsin) (auto simp:)
    also have "... = pi / 3 - a / 2"
      using assms by (subst arcsin_sin) auto
    finally have α: "α ε > 0" "a < 2 * pi / 3 - α ε"
      using assms ε by (auto simp: α_def intro!: arcsin_pos)
```

```
    have "(ε / 2) ^ 2 ≤ (1 / 2) ^ 2"
      using ε by (intro power_mono divide_right_mono) auto
    hence ε': "(ε / 2)^2 < 1"
      by (simp add: power_divide)

    have "cis (pi * 13 / 6) = cis (pi / 6 + 2 * pi)"
      by (simp add: field_simps)
    also have "... = cis (pi / 6)"
      by (subst cis_mult [symmetric]) auto
    finally have [simp]: "cos (pi * 13 / 6) = sqrt 3 / 2" "sin (pi * 13
/ 6) = 1/2"
      by (simp_all add: complex_eq_iff)

    have ends: "pathstart (p1 ε) = -1 / 2 + of_real y * ii"
               "pathfinish (p1 ε) = pathstart (pm1 ε)"
               "pathfinish (pm1 ε) = pathstart (pm2 ε)"
               "pathfinish (pm2 ε) = pathstart (p2 ε)"
               "pathfinish (p2 ε) = cis a"
               "pathstart (cl ε) = pathfinish (p1 ε)" "pathstart (cr ε)
= pathfinish (p1 ε)"
               "pathfinish (cl ε) = pathstart (p2 ε)" "pathfinish (cr
ε) = pathstart (p2 ε)"
          using ε ε'
      by (auto simp: complex_eq_iff α_def β_def cos_double sin_double
sin_arccos rcis_def
                     bad_def cos_diff sin_diff cos_arcsin divide_conv_cnj
sin_add cos_add
                     algebra_simps power2_eq_square cl_def cr_def p1_def
p2_def pm1_def pm2_def
                     rcis_def exp_eq_polar
                simp del: div_mult_self3 div_mult_self4 div_mult_self2
div_mult_self1)

    show eq: "p ≡p p1 ε +++ (pm1 ε +++ pm2 ε) +++ p2 ε"
    proof -
      have "p ≡p (p1 ε +++ pm1 ε) +++ (pm2 ε +++ p2 ε)"
        unfolding p_def p1_def p2_def pm1_def pm2_def using assms ε ε'
α
        by (intro eq_paths_join eq_paths_linepaths' eq_paths_part_circlepaths')
           (auto simp: complex_eq_iff rcis_def exp_eq_polar)
      also have "... ≡p p1 ε +++ (pm1 ε +++ pm2 ε +++ p2 ε)"
        by (rule eq_paths_join_assoc1)
           (auto simp: pm1_def pm2_def p1_def p2_def bad_def rcis_def
exp_eq_polar)
      also have "... ≡p p1 ε +++ (pm1 ε +++ pm2 ε) +++ p2 ε"
        by (intro eq_paths_join eq_paths_join_assoc2)
           (auto simp: pm1_def pm2_def p1_def p2_def bad_def rcis_def
exp_eq_polar)
```

```
    finally show ?thesis .
  qed

  show "pi / 2 ≠ pi / 2 - β ε + 2 * pi ∧ |pi / 2 - β ε + 2 * pi - pi
/ 2| < 2 * pi"
    using ε ε' arcsin by (auto simp: β_def abs_if)

  show "pi / 2 ≠ pi / 2 - β ε ∧
      |pi / 2 - β ε - pi / 2| < 2 * pi ∧
      cis (pi / 2) = cis (pi / 2) ∧
      cis (pi / 2 - β ε) = cis (pi / 2 - β ε + 2 * pi) ∧
      pi / 2 - β ε + 2 * pi - pi / 2 + pi / 2 - (pi / 2 - β ε) = 2
* pi"
    using arcsin unfolding cis_divide [symmetric] cis_mult [symmetric]
by (auto simp add: β_def)

  show "pathfinish (p1 ε) = bad + rcis ε (pi / 2)"
      "pathstart (p2 ε) = bad + rcis ε (pi / 2 - β ε + 2 * pi)"
      "pathstart (pm1 ε +++ pm2 ε) = pathfinish (p1 ε)"
      "pathfinish (pm1 ε +++ pm2 ε) = pathstart (p2 ε)"
    using ends unfolding cl_def cr_def by (simp_all add: rcis_def exp_eq_polar)

  show "path_image (p1 ε) ∩ path_image (p2 ε) ⊆ {pathstart p} ∩ {pathfinish
p}"
  proof (intro subsetI; elim IntE)
    fix x assume x: "x ∈ path_image (p1 ε)" "x ∈ path_image (p2 ε)"
    have *: "Re x = -1/2" "Im x ∈ {sqrt 3 / 2 + ε..y}"
      using ε ε' x(1) by (auto simp: p1_def bad_def)
    have "1 = (-1 / 2) ^ 2 + (sqrt 3 / 2) ^ 2"
      by (simp add: field_simps)
    also have "... < (-1 / 2) ^ 2 + (sqrt 3 / 2) ^ 2 + ε ^ 2"
      using ε by simp
    also have "... ≤ Re x ^ 2 + (sqrt 3 / 2) ^ 2 + 2 * ε * (sqrt 3 /
2) + ε ^ 2"
      using ε * by simp
    also have "... = Re x ^ 2 + (sqrt 3 / 2 + ε) ^ 2"
      by (simp add: power2_eq_square algebra_simps)
    also have "... ≤ Re x ^ 2 + Im x ^ 2"
      using ε * by (intro add_left_mono power_mono) auto
    also have "... = norm x ^ 2"
      unfolding cmod_power2 ..
    also have "... = 1"
      using x(2) by (auto simp: p2_def path_image_part_circlepath')
    finally show "x ∈ {pathstart p} ∩ {pathfinish p}"
      by simp
  qed

  show "does_not_cross (p1 ε) (sphere bad ε)"
  proof (subst does_not_cross_simple_path)
```

```
      show "path_image (p1 ε) ∩ sphere bad ε ⊆ {pathstart (p1 ε), pathfinish
(p1 ε)}"
      proof (intro subsetI; elim IntE)
        fix x assume x: "x ∈ path_image (p1 ε)" "x ∈ sphere bad ε"
        from x have "x = pathfinish (p1 ε)"
          using ε by (auto simp: p1_def dist_norm norm_complex_def complex_eq_iff
bad_def)
        thus "x ∈ {pathstart (p1 ε), pathfinish (p1 ε)}"
          by blast
      qed
    qed (use ε in ‹auto simp: p1_def bad_def complex_eq_iff intro!: simple_path_linepath›)

    show "does_not_cross (p2 ε) (sphere bad ε)"
    proof (subst does_not_cross_simple_path)
      show "path_image (p2 ε) ∩ sphere bad ε ⊆ {pathstart (p2 ε), pathfinish
(p2 ε)}"
      proof (intro subsetI; elim IntE)
        fix x assume x: "x ∈ path_image (p2 ε)" "x ∈ sphere bad ε"
        from x obtain t where t: "t ∈ {a..2 * pi / 3 - α ε}" "x = cis
t"
          using α by (auto simp: p2_def path_image_part_circlepath')
        have "[2 * pi / 3 = t + α ε] (rmod 2 * pi) ∨ [2 * pi / 3 = t
- α ε] (rmod 2 * pi)"
          unfolding α_def by (intro sphere_inter_sphere_aux2)
                             (use ε x(2) t(2) in ‹auto simp: bad_def dist_commute›)
        hence "(2 * pi / 3 - t - α ε) rmod (2 * pi) = 0 ∨ (2 * pi / 3
- t + α ε) rmod (2 * pi) = 0"
          by (auto simp: rcong_conv_diff_rmod_eq_0 algebra_simps)
        also have "(2 * pi / 3 - t - α ε) rmod (2 * pi) = 2 * pi / 3 -
t - α ε"
          using α assms t(1) by (intro rmod_idem) auto
        also have "(2 * pi / 3 - t + α ε) rmod (2 * pi) = 2 * pi / 3 -
t + α ε"
          using α assms t(1) by (intro rmod_idem) auto
        finally have "t = 2 * pi / 3 + α ε ∨ t = 2 * pi / 3 - α ε"
          by (auto simp: algebra_simps)
        with t(1) assms α have "t = 2 * pi / 3 - α ε"
          by auto
        hence "x = pathstart (p2 ε)"
          using t by (auto simp: p2_def rcis_def exp_eq_polar)
        thus "x ∈ {pathstart (p2 ε), pathfinish (p2 ε)}"
          by blast
      qed
    next
      have "a + α ε ≥ -4 / 3 * pi"
        unfolding α_def using assms(2) arcsin(1) pi_gt3 by linarith
      thus "simple_path (p2 ε)"
        unfolding p2_def using α ε
        by (subst simple_path_part_circlepath) (auto simp: abs_if)
```

```
    qed

    show "does_not_cross (pm1 ε +++ pm2 ε) (sphere bad ε)"
    proof (subst does_not_cross_simple_path)
      show "path_image (pm1 ε +++ pm2 ε) ∩ sphere bad ε ⊆
              {pathstart (pm1 ε +++ pm2 ε), pathfinish (pm1 ε +++ pm2
ε)}"
      proof (intro subsetI; elim IntE)
        fix x assume x: "x ∈ path_image (pm1 ε +++ pm2 ε)" "x ∈ sphere
bad ε"
        hence "x ∈ path_image (pm1 ε) ∪ path_image (pm2 ε)"
          by (subst (asm) path_image_join) (auto simp: pm1_def pm2_def
bad_def rcis_def exp_eq_polar)
        thus "x ∈ {pathstart (pm1 ε +++ pm2 ε), pathfinish (pm1 ε +++
pm2 ε)}"
        proof
          assume "x ∈ path_image (pm1 ε)"
          hence x: "Re x = Re bad" "Im x ∈ {Im bad..Im bad + ε}"
            using ε by (auto simp: pm1_def )
          have "Im x = Im bad + dist x bad"
            using x by (auto simp: dist_norm norm_complex_def)
          also have "dist x bad = ε"
            using ‹x ∈ sphere bad ε› by (auto simp: dist_commute)
          finally show ?thesis using x
            by (auto simp: complex_eq_iff pm1_def)
        next
          assume "x ∈ path_image (pm2 ε)"
          then obtain t where t: "t ∈ {2*pi/3 - α ε..2*pi/3}" "x = cis
t"
            unfolding pm2_def path_image_part_circlepath' using α by auto
          have "t = 2 * pi / 3 - α ε"
          proof (rule ccontr)
            assume "t ≠ 2 * pi / 3 - α ε"
            with t(1) have t': "t ∈ {2*pi/3 - α ε<..2*pi/3}"
              by auto
            have "[2 * pi / 3 = t + α ε] (rmod 2 * pi) ∨ [2 * pi / 3
= t - α ε] (rmod 2 * pi)"
              unfolding α_def by (intro sphere_inter_sphere_aux2)
                                 (use ε x(2) t(2) in ‹auto simp: bad_def
dist_commute›)
            hence "(2 * pi / 3 - t - α ε) rmod (2 * pi) = 0 ∨ (2 * pi
/ 3 - t + α ε) rmod (2 * pi) = 0"
              by (auto simp: rcong_conv_diff_rmod_eq_0 algebra_simps)
            also have "(2 * pi / 3 - t - α ε) rmod (2 * pi) = 8 * pi /
3 - t - α ε"
              using α assms t(1) t'
              by (intro rmod_unique[where n = "-1"]) (auto simp: algebra_simps)

            also have "(2 * pi / 3 - t + α ε) rmod (2 * pi) = 2 * pi /
```

```
3 - t + α ε"
              using α assms t(1) by (intro rmod_idem) auto
            finally have "t = 2 * pi / 3 + α ε ∨ t = 8 * pi / 3 - α ε"
              by (auto simp: algebra_simps)
            with t' t(1) α assms show False
              by auto
          qed
          thus ?thesis using t
            by (auto simp: pm2_def rcis_def exp_eq_polar)
        qed
      qed
    next
      have "pm1 ε +++ pm2 ε ≤p (pm1 ε +++ pm2 ε) +++ p2 ε"
        by (rule is_subpath_joinI1) (auto simp: pm1_def pm2_def p2_def
bad_def rcis_def exp_eq_polar)
      also have "... ≤p p1 ε +++ (pm1 ε +++ pm2 ε) +++ p2 ε"
        by (rule is_subpath_joinI2)
           (auto simp: p1_def pm1_def pm2_def p2_def bad_def rcis_def
exp_eq_polar)
      also have "... ≡p p"
        using eq by (simp add: eq_paths_sym_iff)
      finally show "simple_path (pm1 ε +++ pm2 ε)"
      using simple subpath_imp_simple_path by blast
    qed
  qed (auto simp: p1_def p2_def)
  thus ?thesis
    by (simp add: p_def p1_def cl_def p2_def avoid_linepath_circlepath_def
[abs_def]
                  Let_def bad_def α_def β_def cr_def)
qed

lemmas detour_rel_avoid_linepath_circlepath_right =
  detour_rel_swap[OF detour_rel_avoid_linepath_circlepath_left]
```

4.5 Consequences of the detour relation

```
locale detour =
  aux: detour_rel_aux_locale ε "I ∪ X" p p'
  for ε :: real and I X P :: "complex set" and p_orig p p' :: "real ⇒
complex" +
  assumes eq_loops: "eq_loops p_orig p"
  assumes simple_loop_original: "simple_loop p_orig"
  assumes same_orientation': "simple_loop_orientation p' = simple_loop_orientation
p"
  assumes I_inside:  "I ⊆ inside (path_image p')"
  assumes X_outside: "X ⊆ outside (path_image p')"
  assumes disjoint:  "P ∩ (path_image p_orig ∪ (⋃x∈I∪X. cball x ε))
= {}"
begin
```

```
lemma same_orientation: "simple_loop_orientation p' = simple_loop_orientation
p_orig"
  using same_orientation' eq_loops_imp_same_orientation [OF eq_loops]
by simp
```

`p'` is a simple loop that can be obtained from `p` through local deformations in ε-neighbourhoods of `I ∪ X`.

```
lemma same_ends: "pathstart p' = pathstart p" "pathfinish p' = pathfinish
p"
  by simp_all

lemmas valid = aux.p'_valid

lemma simple_loop: "simple_loop p'"
  by (metis local.same_orientation simple_loop_orientation_eq_0_iff simple_loop_original)

lemma path_image_eq [simp]: "path_image p = path_image p_orig"
  by (rule sym, rule eq_loops_imp_path_image_eq, rule eq_loops)

lemma homotopic': "homotopic_paths (path_image p_orig ∪ (⋃x∈I ∪ X.
cball x ε)) p p'"
  using aux.homotopic by simp

lemma homotopic'': "homotopic_loops (path_image p_orig ∪ (⋃x∈I ∪ X.
cball x ε)) p p'"
  by (intro homotopic_paths_imp_homotopic_loops homotopic')
     (use same_ends eq_loops in ‹auto dest: eq_loops_imp_loop›)

lemma homotopic: "homotopic_loops (path_image p_orig ∪ (⋃x∈I ∪ X.
cball x ε)) p_orig p'"
proof -
  have "homotopic_loops (path_image p_orig ∪ (⋃x∈I ∪ X. cball x ε))
p_orig p"
    by (intro eq_loops_imp_homotopic [OF eq_loops]) auto
  with homotopic'' show ?thesis
    using homotopic_loops_trans by blast
qed
```

```
lemma path_image_subset: "path_image p' ⊆ path_image p_orig ∪ (⋃x∈I∪X.
cball x ε)"
  using aux.path_image_p' by simp
```

All the points in `I` are strictly inside `p'`, all the points in `X` are strictly outside, and all the points in `P` are unchanged (i.e. if they were outside

before they still are and if they were inside before, they still are).

```
lemma I_not_on_path: "I ∩ path_image p' = {}"
  using aux.X_disjoint by blast

lemma X_not_on_path: "X ∩ path_image p' = {}"
  using aux.X_disjoint by blast

lemma P_not_on_path: "P ∩ path_image p' = {}"
  using disjoint aux.path_image_p' by auto

lemma I_X_disjoint: "I ∩ X = {}"
proof -
  have "inside (path_image p') ∩ outside (path_image p') = {}"
    using simple_loop by simp
  thus ?thesis
    using I_inside X_outside by blast
qed

lemma I_P_disjoint: "I ∩ P = {}"
  using path_image_eq aux.X_subset disjoint path_image_subset by blast

lemma X_P_disjoint: "X ∩ P = {}"
  using path_image_eq aux.X_subset disjoint path_image_subset by blast

lemma winding_number_unchanged:
  assumes "z ∈ P"
  shows   "winding_number p' z = winding_number p_orig z"
  using disjoint aux.winding_number_unchanged[of z]
        eq_loops_imp_winding_number_eq [OF eq_loops, of z] assms by (auto
simp: disjoint_iff)

lemma P_inside_iff:
  assumes "z ∈ P"
  shows   "z ∈ inside (path_image p') ⟷ z ∈ inside (path_image p_orig)"
proof -
  from assms have [simp]: "z ∉ path_image p_orig" "z ∉ path_image p'"
    using disjoint P_not_on_path by auto
  show ?thesis
    using winding_number_unchanged [OF assms] simple_loop simple_loop_original
    by (simp add: inside_simple_loop_iff)
qed

lemma P_outside_iff:
  assumes "z ∈ P"
  shows   "z ∈ outside (path_image p') ⟷ z ∈ outside (path_image p_orig)"
proof -
  from assms have [simp]: "z ∉ path_image p_orig" "z ∉ path_image p'"
    using disjoint P_not_on_path by auto
  show ?thesis
```

```
    using winding_number_unchanged [OF assms] simple_loop simple_loop_original
    by (simp add: outside_simple_loop_iff)
qed

lemma winding_number_I:
  assumes "z ∈ I"
  shows   "winding_number p' z = simple_loop_orientation p_orig"
  using assms I_inside same_orientation simple_loop simple_loop_winding_number_cases
        same_orientation' by auto

lemma winding_number_X:
  assumes "z ∈ X"
  shows   "winding_number p' z = 0"
  using assms X_outside
  by (metis aux.p'_path simple_loop simple_loop_def subsetD winding_number_zero_in_outside)

end
```

Our final result: If `p` is a simple closed curve that is in detour relation with a family of deformed versions `p'` of itself, then for any closed set `P` of points of interest not on `p` there is an ε `> 0` such that all curves `p'(`ε`)` for ε `'` `<` ε have basically the same properties as `p`; namely:

- `p'(`ε`)` is `pr(`ε`)` also a closed curve with the same orientation and the same start/end as `p`.
- `p'(`ε`)` is homotopic to `p` with a homotopy transformation that only passes through the original path image of `p` plus ε-balls around `I` $\cup$ `X`. In other words, `p` is identical to `p'(`ε`)` except for small deformations of size ε around `I` $\cup$ `X`.
- All points in `I` ("include") are inside `p'(`ε`)` and all points in `X` ("exclude") are outside.
- Each point in `P` ("preserve") is in `p'(`ε`)` if and only if it was already in `p`.

```
theorem detour_generic:
  assumes rel: "detour_rel I X p pl pr"
  assumes eq: "eq_loops p_orig p"
  assumes p_orig: "simple_loop p_orig" and valid: "valid_path p"
  assumes P: "closed P" "P ∩ path_image p_orig = {}"
  defines "p' ≡ (if simple_loop_ccw p_orig then pr else pl)"
  shows   "eventually (λε. detour ε I X P p_orig p (p' ε)) (at_right 0)"
proof -
  have [simp, intro]: "path p_orig"
    using p_orig by (auto simp: simple_loop_def simple_path_imp_path)
  note [simp] = simple_loop_ccw_conv_cw[OF p_orig]
```

```
  note [simp] = eq_loops_imp_path_image_eq [OF eq]
  have p: "simple_loop p"
    using p_orig eq_loops_imp_simple_loop_iff [OF eq] by simp
  hence [simp, intro]: "path p"
    by (auto simp: simple_loop_def simple_path_imp_path)
  from rel p valid have ev1: "eventually (λε. detour_rel_locale ε I X
p (pl ε) (pr ε)) (at_right 0)"
    by (auto simp: detour_rel_def simple_loop_def)
  then obtain eps0 where "detour_rel_locale eps0 I X p (pl eps0) (pr
eps0)"
    by fastforce
  then interpret eps0: detour_rel_locale eps0 I X p "pl eps0" "pr eps0"
.

  obtain z0 where z0: "z0 ∈ inside (path_image p)"
    using p by (metis simple_closed_path_wn3 simple_loop_def)
  define P' where "P' = insert z0 P"
  have P': "closed P'" "P' ∩ (path_image p ∪ (I ∪ X)) = {}" "z0 ∈ P'"
    using P eps0.pl.X_subset z0 inside_no_overlap[of "path_image p"]
    by (auto simp: P'_def simp del: inside_no_overlap)

  have "compact (path_image p ∩ (I ∪ X))"
    by (intro compact_Int_closed compact_path_image) auto
  also have "path_image p ∩ (I ∪ X) = I ∪ X"
    by auto
  finally have "compact (I ∪ X)" .
  with P' have ev2: "eventually (λε. setdist_gt ε (I ∪ X) P') (at_right
0)"
    by (intro compact_closed_imp_eventually_setdist_gt_at_right_0) auto

  show ?thesis
    using ev1 ev2
  proof eventually_elim
    case (elim ε)
    interpret detour_rel_locale ε I X p "pl ε" "pr ε"
      by (rule elim)
    have disjoint: "P' ∩ (path_image p ∪ (⋃x∈I∪X. cball x ε)) = {}"
    proof (intro equalityI subsetI, elim IntE UnE)
      fix x assume x: "x ∈ P'" "x ∈ (⋃x∈I∪X. cball x ε)"
      then obtain y where y: "y ∈ I ∪ X" "dist y x ≤ ε"
        by auto
      moreover from y(1) x(1) have "dist y x > ε"
        by (rule setdist_gtD[OF elim(2)])
      ultimately show "x ∈ {}"
        by simp
    qed (use P' in auto)

    interpret detour_rel_loop ε I X p "pl ε" "pr ε"
    proof
```

```
      show "simple_loop p"
        by fact
    next
      have "z0 ∉ path_image p ∪ (⋃x∈I∪X. cball x ε)"
        using disjoint P' by blast
      moreover from this have "winding_number p z0 ≠ 0"
        using p z0 by (simp add: simple_loop_winding_number_cases)
      ultimately show "∃z. winding_number p z ≠ 0 ∧ z ∉ path_image
p ∪ (⋃x∈I∪X. cball x ε)"
        using z0 P' by (intro exI[of _ z0]) auto
    qed
    note [[goals_limit = 30]]
    show ?case
    proof
      show "ε > 0" "simple_path (p' ε)" "simple_loop p_orig"
        using ε_pos p_orig by (auto simp: p'_def simple_loop_def)
      show "I ∪ X ⊆ path_image p" "(I ∪ X) ∩ path_image (p' ε) = {}"
           "homotopic_paths (path_image p ∪ (⋃x∈I ∪ X. cball x ε)) p
(p' ε)"
        using pl.homotopic pr.homotopic pl.X_disjoint pr.X_disjoint pl.X_subset
        unfolding p'_def by (simp_all add: Un_commute)
      show "simple_loop_orientation (p' ε) = simple_loop_orientation
p"
        unfolding p'_def by (simp add: same_orientation)
      show "P ∩ (path_image p_orig ∪ (⋃x∈I ∪ X. cball x ε)) = {}"
        using disjoint unfolding P'_def by auto
      show "I ⊆ inside (path_image (p' ε))"
        unfolding p'_def using inside_pl_L_iff inside_pr_L_iff
        using eq eq_loops_imp_ccw_iff eq_loops_imp_cw_iff by fastforce
      show "X ⊆ outside (path_image (p' ε))"
      proof
        fix x assume x: "x ∈ X"
        hence "x ∉ path_image (p' ε)"
          unfolding p'_def by auto
        moreover have "x ∉ inside (path_image (p' ε))"
          using x inside_pl_R_iff inside_pr_R_iff unfolding p'_def
          using eq eq_loops_imp_ccw_iff eq_loops_imp_cw_iff by fastforce
        ultimately show "x ∈ outside (path_image (p' ε))"
          by (simp add: inside_outside)
      qed
      show "valid_path (p' ε)"
        using p'_def pl.p'_valid pr.p'_valid by simp
    qed fact+
  qed
qed

corollary detour_ccw:
  assumes "detour_rel I X p pl pr" "p_orig ≡○ p"
  assumes "simple_loop_ccw p_orig" "valid_path p"
```

```
  assumes "closed P" "P ∩ path_image p_orig = {}"
  shows   "eventually (λε. detour ε I X P p_orig p (pr ε)) (at_right
0)"
  using detour_generic[OF assms(1), of p_orig P] assms by (auto simp:
simple_loop_ccw_def)

corollary detour_cw:
  assumes "detour_rel I X p pl pr" "p_orig ≡○ p"
  assumes "simple_loop_cw p_orig" "valid_path p" "closed P" "P ∩ path_image
p_orig = {}"
  shows   "eventually (λε. detour ε I X P p_orig p (pl ε)) (at_right
0)"
proof -
  from assms(3) have "¬simple_loop_ccw p_orig"
    using simple_path_not_cw_and_ccw by blast
  moreover have "simple_loop p_orig"
    using assms(3) by (auto simp: simple_loop_cw_def)
  ultimately show ?thesis
    using detour_generic[OF assms(1), of p_orig P] assms by simp
qed

end
```