The Detour Calculus

Manuel Eberl

June 18, 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

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"*
  ⟨*proof*⟩

lemma *le_filterD_frequently' [trans]: "frequently P F ⟹ F ≤ G ⟹ frequently P G"*
  ⟨*proof*⟩

lemma *frequently_filtermap: "frequently P (filtermap f F) ⟷ frequently (λx. P (f x)) F"*
  ⟨*proof*⟩

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*⟩

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)"
  ⟨*proof*⟩

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)"
  ⟨*proof*⟩

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)"
⟨*proof*⟩

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)"
⟨*proof*⟩

lemma *eventually_path_path_nhds: "eventually (λp. path p) (path_nhds γ)"*
⟨*proof*⟩

lemma *path_nhds_neq_bot [simp]: "path γ ⟹ path_nhds γ ≠ bot"*
⟨*proof*⟩

lemma *eventually_dist_less_path_nhds:*
assumes *"e > 0"*
shows *"eventually (λp. ∀t∈{0..1}. dist (p t) (γ t) < e) (path_nhds γ)"*
⟨*proof*⟩

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*⟩

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*⟩

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*⟩

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)"*

⟨*proof*⟩

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)"
⟨*proof*⟩

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)"
⟨*proof*⟩

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)"
⟨*proof*⟩

lemma *eventually_valid_path_valid_path_nhds: "eventually (λp. valid_path p) (valid_path_nhds γ)"*
⟨*proof*⟩

lemma *path_nhds_le_valid_path_nhds: "valid_path_nhds γ ≤ path_nhds γ"*
⟨*proof*⟩

lemma *valid_path_nhds_neq_bot [simp]: "valid_path γ ⟹ valid_path_nhds γ ≠ bot"*
⟨*proof*⟩

lemma *valid_path_nhds_eq_bot' [simp]:*
assumes *"path (γ :: real ⇒ 'a :: euclidean_space)"*
shows *"valid_path_nhds γ ≠ bot"*
⟨*proof*⟩

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)"*
⟨*proof*⟩

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)"
⟨*proof*⟩

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)"
⟨*proof*⟩

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*⟩

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)"
⟨*proof*⟩

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*⟩

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*⟩

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*⟩

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*⟩

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*⟩

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"*
⟨*proof*⟩

lemma *dist_cnj: "dist (cnj x) (cnj y) = dist x y"*
⟨*proof*⟩

lemma *closed_segment_cnj: "closed_segment (cnj w) (cnj z) = cnj ‘ closed_segment w z"*
⟨*proof*⟩

lemma *closed_segment_mult:*
*"(*) (c :: 'a :: real_algebra_1) ‘ closed_segment a b = closed_segment (c * a) (c * b)"*
⟨*proof*⟩

lemma *arcsin_pos: "x ∈ {0<..1} ⟹ arcsin x > 0"*

⟨*proof*⟩

lemma *sin_lt_zero'*: "x ∈ {-pi<..<0} ⟹ sin x < 0"
⟨*proof*⟩

lemma *tan_arcsin*: "x ∈ {-1..1} ⟹ tan (arcsin x) = x / sqrt (1 - x ^ 2)"
⟨*proof*⟩

lemma *Arg_neg_real [simp]*: "Im x = 0 ⟹ Re x < 0 ⟹ Arg x = pi"
⟨*proof*⟩

lemma *cis_eq_1_iff'*:
assumes "|x| < 2 * pi"
shows "cis x = 1 ⟷ x = 0"
⟨*proof*⟩

lemma *DeMoivre_int*: "cis x powi n = cis (of_int n * x)"
⟨*proof*⟩

lemma *cis_of_int_times_pi_half*: "cis (of_int n * pi / 2) = i powi n"
⟨*proof*⟩

lemma *cis_of_nat_times_pi_half*: "cis (real n * pi / 2) = i ^ n"
⟨*proof*⟩

lemma *cis_of_int_times_pi*: "cis (of_int n * pi) = (-1) powi n"
⟨*proof*⟩

lemma *cis_of_nat_times_pi*: "cis (real n * pi) = (-1) ^ n"
⟨*proof*⟩

lemma *power3_i [simp]*: "i ^ 3 = -i"
and *power4_i [simp]*: "i ^ 4 = 1"
⟨*proof*⟩

lemma *i_power_mod4*: "i ^ n = i ^ (n mod 4)"
⟨*proof*⟩

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

lemma *cis_numeral_pi_times_numeral_half [simp]*:
"cis (pi * numeral num / 2) = i ^ (numeral num mod 4)"
⟨*proof*⟩

lemma *cis_numeral_times_pi [simp]*:
"cis (numeral num * pi) = (-1) ^ (numeral num mod 2)"

⟨*proof*⟩

lemma *cis_pi_times_numeral [simp]:*
*"cis (pi * numeral num) = (-1) ^ (numeral num mod 2)"*
⟨*proof*⟩

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

lemma *cis_minus_numeral_times_pi [simp]:*
*"cis (-(numeral num * pi)) = (-1) ^ (numeral num mod 2)"*
⟨*proof*⟩

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

lemma *cis_minus_pi_times_numeral_pi [simp]:*
*"cis (-(pi * numeral num)) = (-1) ^ (numeral num mod 2)"*
⟨*proof*⟩

lemma *cis_minus_pi [simp]: "cis (-pi) = -1"*
⟨*proof*⟩

lemma *inner_mult_both_complex: "(z * x :: complex) · (z * y) = norm z ^ 2 * (x · y)"*
⟨*proof*⟩

lemma *orthogonal_transformation_mult_complex [intro]:*
"norm z = 1 ⟹ orthogonal_transformation ((*) (z :: complex))"
⟨*proof*⟩

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*⟩

lemma *finite_imp_eventually_sparse_at_0:*
assumes *"finite X"*
shows *"eventually (λε. sparse ε X) (at_right 0)"*
⟨*proof*⟩

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*⟩

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*⟩

lemma *lipschitz_on_complex_inverse:*
assumes *"C > 0"*
shows *"(1/C^2)-lipschitz_on {z. Im z ≥ C} (λz. inverse z :: complex)"*
⟨*proof*⟩

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*⟩

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*⟩

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)"*
⟨*proof*⟩

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*⟩

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)"*
⟨*proof*⟩

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"*
⟨*proof*⟩

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*⟩

lemma *homotopic_loops_reversepath_D:*
"homotopic_loops A p q ⟹ homotopic_loops A (reversepath p) (reversepath q)"
⟨*proof*⟩

lemma *homotopic_loops_reversepath:*
"homotopic_loops A (reversepath p) (reversepath q) ⟷ homotopic_loops A p q"
⟨*proof*⟩

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*⟩

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*⟩

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*⟩

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*⟩

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*⟩

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*⟩

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*⟩

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*⟩

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*⟩

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*⟩

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*⟩

lemma *winding_preserving_translate: "winding_preserving A (λx. c + x) (λx. x)"*
⟨*proof*⟩

lemma *winding_preserving_mult: "c ≠ 0 ⟹ winding_preserving A (λx. c * x) (λx. x)"*
⟨*proof*⟩

lemma *winding_preserving_cnj: "winding_preserving A cnj (λx. -cnj x)"*
⟨*proof*⟩

lemma *winding_preserving_uminus: "winding_preserving A (λx. -x) (λx. x)"*
  ⟨*proof*⟩

3.6 Paths

lemma *simple_path_cnj [simp]: "simple_path (cnj ∘ p) ⟷ simple_path p"*
  ⟨*proof*⟩

lemma *part_circlepath_cnj': "cnj ∘ part_circlepath c r a b = part_circlepath (cnj c) r (-a) (-b)"*
  ⟨*proof*⟩

lemma *linepath_minus: "linepath (-a) (-b) x = -linepath a b x"*
  ⟨*proof*⟩

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*⟩

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 {}))"
⟨*proof*⟩

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"*
⟨*proof*⟩

lemma *arc_continuous_image:*
assumes *"arc p" "inj_on f (path_image p)" "continuous_on (path_image p) f"*
shows *"arc (f ∘ p)"*
⟨*proof*⟩

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*⟩

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)"*
⟨*proof*⟩

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*⟩

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 = {}"*
⟨*proof*⟩

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*⟩

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*⟩

lemma *path_image_part_circlepath':*
"path_image (part_circlepath x r a b) = (λt. x + rcis r t) ‘ closed_segment a b"
⟨*proof*⟩

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*⟩

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*⟩

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)"*
⟨*proof*⟩

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*⟩

lemma *filterlim_at_withinD':*
assumes *"filterlim f (at L within A) F" "open X" "L ∈ X"*

shows *"eventually (λx. f x ∈ X ∩ A) F"*
⟨*proof*⟩

lemma *filterlim_at_rightD:*
assumes *"filterlim f (at_right L) F" "a > L"*
shows *"eventually (λx. f x ∈ {L<..<a}) F"*
⟨*proof*⟩

lemma *filterlim_at_leftD:*
assumes *"filterlim f (at_left L) F" "a < L"*
shows *"eventually (λx. f x ∈ {a<..<L}) F"*
⟨*proof*⟩

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)"*
⟨*proof*⟩

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)}"*
⟨*proof*⟩

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)}"*
⟨*proof*⟩

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}"*
⟨*proof*⟩

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

lemma *shiftpath_reversepath_loop:*
assumes *"x ∈ {0..1}" "pathstart p = pathfinish p"*
shows *"shiftpath c (reversepath p) x = reversepath (shiftpath (1-c) p) x"*
⟨*proof*⟩

lemma *eqloops_reversepath_cong:*
assumes *"p ≡○ q"*
shows *"reversepath p ≡○ reversepath q"*
⟨*proof*⟩

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"*
⟨*proof*⟩

lemma *p'_path [simp, intro]: "path p'"*
⟨*proof*⟩

lemma *path_image_p': "path_image p' ⊆ path_image p ∪ (⋃x∈X. cball x ε)"*
⟨*proof*⟩

lemma *same_ends [simp]: "pathstart p' = pathstart p" "pathfinish p' = pathfinish p"*
⟨*proof*⟩

lemma *ends_not_in_X: "pathstart p ∉ X" "pathfinish p ∉ X"*
⟨*proof*⟩

lemma *translate:*

```
  "detour_rel_aux_locale ε ((+) a ‘ X) ((+) a ∘ p) ((+) a ∘ p')"
```
⟨*proof*⟩

lemma *orthogonal_transformation:*
```
  assumes f [intro]: "orthogonal_transformation f"
  shows   "detour_rel_aux_locale ε (f ‘ X) (f ∘ p) (f ∘ p')"
```
⟨*proof*⟩

lemma *rotate:*
```
  assumes "norm z = 1"
  shows   "detour_rel_aux_locale ε ((*) z ‘ X) ((*) z ∘ p) ((*) z ∘ p')"
```
⟨*proof*⟩

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*⟩

lemma *reverse [intro!]:*
```
  "detour_rel_aux_locale ε X (reversepath p) (reversepath p')"
```
⟨*proof*⟩

theorem *winding_number_unchanged:*
```
  assumes "z ∉ path_image p ∪ (⋃x∈X. cball x ε)"
  shows   "winding_number p' z = winding_number p z"
```
⟨*proof*⟩

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*⟩

lemma *mono:*
```
  assumes "ε ≤ ε'"
  shows   "detour_rel_aux_locale ε' X p p'"
```
⟨*proof*⟩

lemma *cnj:* `"detour_rel_aux_locale ε (cnj ‘ X) (cnj ∘ p) (cnj ∘ p')"`
⟨*proof*⟩

end

lemma *detour_rel_aux_locale_refl [intro!]:*
```
  "ε > 0 ⟹ simple_path p ⟹ valid_path p ⟹ detour_rel_aux_locale
ε {} p p"
```
⟨*proof*⟩

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"*
⟨*proof*⟩

lemma *detour_rel_aux_join_aux1:*
assumes *setdist: "setdist_gt ε X1 (path_image p2)"*
shows *"(⋃x∈X1. cball x ε) ∩ path_image p2 = {}"*
⟨*proof*⟩

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*⟩

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 = {}"*
⟨*proof*⟩

lemma *X1_inter_p2: "X1 ∩ path_image p2' = {}"*
and *X2_inter_p1: "X2 ∩ path_image p1' = {}"*
⟨*proof*⟩

lemma *X1_inter_p2' [simp]: "x ∈ X1 ⟹ x ∉ path_image p2'"*
and *X2_inter_p1' [simp]: "x ∈ X2 ⟹ x ∉ path_image p1'"*
⟨*proof*⟩

lemma *X1_X2_disjoint: "X1 ∩ X2 = {}"*
⟨*proof*⟩

lemma *X1_X2_disjoint' [simp]: "x ∈ X1 ⟹ x ∉ X2"*
⟨*proof*⟩

lemma *cball_X1_inter_cball_X2:*
assumes *"x ∈ X1" "y ∈ X2"*
shows *"cball x ε ∩ cball y ε = {}"*
⟨*proof*⟩

lemma *cball_X1_inter_cball_X2':*
shows *"(⋃y∈X1. cball y ε) ∩ (⋃y∈X2. cball y ε) = {}"*
⟨*proof*⟩

sublocale *p12: detour_rel_aux_locale ε "X1 ∪ X2" "p1 +++ p2" "p1' +++ p2'"*
⟨*proof*⟩

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'"*
⟨*proof*⟩

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*⟩

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 = {}"*
⟨*proof*⟩

lemma *ends_not_in_L: "pathstart p ∉ L" "pathfinish p ∉ L"*

and *ends_not_in_R: "pathstart p ∉ R" "pathfinish p ∉ R"*
⟨*proof*⟩

lemma *ε_pos: "ε > 0"*
⟨*proof*⟩

lemma *swap: "detour_rel_locale ε R L p pr pl"*
⟨*proof*⟩

lemma *reverse [intro!]:*
"detour_rel_locale ε R L (reversepath p) (reversepath pl) (reversepath pr)"
⟨*proof*⟩

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*⟩

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*⟩

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*⟩

lemma *mono:*
assumes *"ε ≤ ε’"*
shows *"detour_rel_locale ε’ L R p pl pr"*
⟨*proof*⟩

lemma *cnj: "detour_rel_locale ε (cnj ‘ R) (cnj ‘ L) (cnj ∘ p) (cnj ∘*

pl) (cnj ∘ pr)"
⟨proof⟩

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*
  ⟨proof⟩

sublocale *pr: detour_rel_aux_locale_join ε "L1 ∪ R1" p1 pr1 "L2 ∪ R2" p2 pr2*
  ⟨proof⟩

sublocale *p12: detour_rel_locale ε "L1 ∪ L2" "R1 ∪ R2" "p1 +++ p2" "pl1 +++ pl2" "pr1 +++ pr2"*
⟨proof⟩

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*
  ⟨proof⟩

sublocale *pr: detour_rel_aux_loop ε "L ∪ R" p pr*
  ⟨proof⟩

lemma *same_orientation:*
  "simple_loop_orientation pl = simple_loop_orientation p"
  "simple_loop_orientation pr = simple_loop_orientation p"
⟨proof⟩

lemma *reverse_loop [intro!]:*

"detour_rel_loop ε R L (reversepath p) (reversepath pl) (reversepath pr)"
⟨proof⟩

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⟩

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⟩

end

lemma detour_rel_locale_swap: "detour_rel_locale ε L R p pl pr ⟷ detour_rel_locale ε R L p pr pl"
⟨proof⟩

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⟩

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*⟩

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'"*
⟨*proof*⟩

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"*
⟨*proof*⟩

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"*
⟨*proof*⟩

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*⟩

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))"*

⟨*proof*⟩

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*⟩

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)"
⟨*proof*⟩

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)"
⟨*proof*⟩

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*⟩

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"*
⟨*proof*⟩

lemma *detour_rel_swap: "detour_rel L R p pl pr ⟹ detour_rel R L p*
pr pl"
⟨*proof*⟩

lemma *detour_rel_swap_iff: "detour_rel L R p pl pr ⟷ detour_rel R*
L p pr pl"
⟨*proof*⟩

named_theorems *detour_rel_intros*

lemma *detour_rel_refl [detour_rel_intros, simp, intro!]: "detour_rel {} {} p (λ_. p) (λ_. p)"*
⟨*proof*⟩

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))"*
⟨*proof*⟩

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*⟩

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))"*
⟨*proof*⟩

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)"*
⟨*proof*⟩

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⟩
```

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*⟩

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)"
  ⟨proof⟩
```

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)"
  ⟨proof⟩
```

lemma *avoid_linepath_translate:*

```
  "(+) c ∘ avoid_linepath left a b bad ε =
   avoid_linepath left (c + a) (c + b) (c + bad) ε"
  ⟨proof⟩
```

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⟩
```

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*⟩

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*⟩

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*⟩

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)"*
  ⟨*proof*⟩

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*⟩

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*⟩

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*⟩

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'"*
⟨*proof*⟩

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*⟩

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*⟩

lemma *valid_path: "valid_path (avoid_part_circlepath left x r a b φ ε)"*
⟨*proof*⟩

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*⟩

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*⟩

lemma *sphere_inter_sphere_pos_real_line_aux1:*
assumes *r: "r ≥ 0" "r ≤ 1"*
shows *"dist 1 (cis (2 * arcsin (r / 2))) = r"*
⟨*proof*⟩

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*⟩

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*⟩

lemma *sphere_inter_sphere_aux1:*
assumes *r: "r ≥ 0" "r ≤ 1"*
shows *"dist (cis φ) (cis (φ + 2 * arcsin (r / 2))) = r"*
⟨*proof*⟩

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*⟩

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*⟩

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* $\equiv_p$
        avoid_part_circlepath left x r a c φ ε"
⟨*proof*⟩

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* $\equiv_p$
          avoid_part_circlepath left x r c b φ ε"
⟨*proof*⟩

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)* $\equiv_p$
        avoid_part_circlepath (¬left) x r b a φ ε"
⟨*proof*⟩

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*⟩

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*⟩

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*⟩

lemma *detour_rel_part_circlepath_semicircle_left [detour_rel_intros]:*

assumes *"$\varphi \in$ 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*⟩

lemma *detour_rel_part_circlepath_semicircle_right [detour_rel_intros]:*
assumes *"$\varphi \in$ 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)"
⟨*proof*⟩

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 $\longleftrightarrow$ Arg (a - b) $\geq$ 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"
⟨*proof*⟩

lemma *avoid_linepath_linepath_mult:*
assumes *"a $\neq$ 0" "b $\neq$ 0"*
shows *"(*) c ∘ avoid_linepath_linepath left a 0 b ε =*
*avoid_linepath_linepath left (c * a) 0 (c * b) (norm c * ε)"*
⟨*proof*⟩

lemma *norm_linepath_0: "norm (linepath 0 a x) = |x| * norm a"*
⟨*proof*⟩

lemma *norm_linepath_0': "norm (linepath a 0 x) = |1 - x| * norm a"*
⟨*proof*⟩

lemma *detour_rel_avoid_linepath_linepath_aux1:*
assumes *"1 $\notin$ closed_segment 0 c" "c $\notin$ 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*⟩

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*⟩

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*⟩

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*⟩

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)"
  ⟨*proof*⟩

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"*
⟨*proof*⟩

lemma *valid_path_avoid_linepath_circlepath:*
assumes *"ε ≥ 0" "ε ≤ 2"*
shows *"valid_path (avoid_linepath_circlepath left y a ε)"*
⟨*proof*⟩

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*⟩

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*⟩

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"*
⟨*proof*⟩

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"*
⟨*proof*⟩

lemmas *valid = aux.p'_valid*

lemma *simple_loop: "simple_loop p'"*
⟨*proof*⟩

lemma *path_image_eq [simp]: "path_image p = path_image p_orig"*
⟨*proof*⟩

lemma *homotopic': "homotopic_paths (path_image p_orig ∪ (⋃x∈I ∪ X. cball x ε)) p p'"*
⟨*proof*⟩

lemma *homotopic'': "homotopic_loops (path_image p_orig ∪ (⋃x∈I ∪ X. cball x ε)) p p'"*
⟨*proof*⟩

lemma *homotopic: "homotopic_loops (path_image p_orig ∪ (⋃x∈I ∪ X. cball x ε)) p_orig p'"*
⟨*proof*⟩

lemma *path_image_subset: "path_image p' ⊆ path_image p_orig ∪ (⋃x∈I∪X. cball x ε)"*
⟨*proof*⟩

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' = {}"*
⟨*proof*⟩

lemma *X_not_on_path: "X ∩ path_image p' = {}"*
⟨*proof*⟩

lemma *P_not_on_path: "P ∩ path_image p' = {}"*
⟨*proof*⟩

lemma *I_X_disjoint: "I ∩ X = {}"*
⟨*proof*⟩

lemma *I_P_disjoint: "I ∩ P = {}"*
⟨*proof*⟩

lemma *X_P_disjoint: "X ∩ P = {}"*
⟨*proof*⟩

lemma *winding_number_unchanged:*
assumes *"z ∈ P"*
shows *"winding_number p' z = winding_number p_orig z"*
⟨*proof*⟩

lemma *P_inside_iff:*
assumes *"z ∈ P"*
shows *"z ∈ inside (path_image p') ⟷ z ∈ inside (path_image p_orig)"*
⟨*proof*⟩

lemma *P_outside_iff:*
assumes *"z ∈ P"*
shows *"z ∈ outside (path_image p') ⟷ z ∈ outside (path_image p_orig)"*
⟨*proof*⟩

lemma *winding_number_I:*
assumes *"z ∈ I"*
shows *"winding_number p' z = simple_loop_orientation p_orig"*
⟨*proof*⟩

lemma *winding_number_X:*
assumes *"z ∈ X"*
shows *"winding_number p' z = 0"*
⟨*proof*⟩

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 $\varepsilon > 0$ such that all curves *p'(ε)* for $\varepsilon' < \varepsilon$ 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*⟩

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)"*
⟨*proof*⟩

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*⟩

end