Brownian Motion

Christian Pardillo-Laursen and Simon Foster

November 28, 2026

Abstract

Brownian motion is a mathematical model of continuous random movement. It starts at zero, has independent changes over disjoint time intervals, and the change over an interval of length t has a centred normal distribution with variance t. It is a fundamental example of a continuous-time stochastic process; see, for example, Klenke [1].

In this work, we construct Brownian motion in Isabelle/HOL. We first build consistent Gaussian joint distributions for its values at any finite collection of times, and use them to obtain a Gaussian process. We then apply the AFP formalisation of the Kolmogorov–Chentsov theorem [2] to construct a continuous modification of this process and prove that it starts at zero and has the required independent normal increments. In fact, its paths satisfy a stronger local Hölder-continuity property. We then construct Brownian motion in arbitrary finite-dimensional Euclidean spaces from independent coordinate processes, proving path continuity and independent centred multivariate-normal increments with covariance proportional to the elapsed time. Finally, for the real-valued process, we show that rescaling time by $c > 0$ and space by $1/\sqrt{c}$ leaves the distribution of the entire process unchanged.

Contents

1 Normal laws for increments **2**

2 Finite-dimensional laws from independent increments **6**

3 From consistent laws to a Gaussian process **27**

4 Brownian motion **48**

5 Brownian motion in finite-dimensional Euclidean space **60**

5.1 Centred Euclidean normal distributions 62

5.2 Independent families on a finite product 65

5.3 The Brownian-motion specification 70

5.4 Basic Brownian properties 70

theory *Gaussian-Increments*

imports *Kolmogorov-Chentsov.Kolmogorov-Chentsov HOL−Probability.Distributions*
begin

1 Normal laws for increments

HOL-Probability's *normal-density* takes a standard deviation, not a variance. The zero-variance case is a point mass; no zero-standard-deviation density is used. The negative-variance branch makes this a total measure-valued function, but subsequent theorems about Gaussian increments will assume nonnegative variances.

definition *centered-normal* :: *real* ⇒ *real measure* **where**
  centered-normal v =
    (*if v* ≤ *0 then return borel 0*
     else density lborel (*normal-density 0* (*sqrt v*)))

lemma *centered-normal-zero* [*simp*]: *centered-normal 0* = *return borel 0*
  by (*simp add*: *centered-normal-def*)

lemma *centered-normal-nonpositive* [*simp*]:
  v ≤ *0* ⟹ *centered-normal v* = *return borel 0*
  by (*simp add*: *centered-normal-def*)

lemma *prob-space-centered-normal* [*intro*]: *prob-space* (*centered-normal v*)
proof (*cases v* ≤ *0*)
  case *True*
  then show *?thesis* **by** (*simp add*: *centered-normal-def prob-space-return*)
next
  case *False*
  then have *sigma*: *0* < *sqrt v* **by** *simp*
  show *?thesis*
   using *False sigma* **by** (*simp add*: *centered-normal-def prob-space-normal-density*)
qed

lemma *sets-centered-normal* [*simp*]: *sets* (*centered-normal v*) = *sets borel*
  by (*simp add*: *centered-normal-def*)

lemma *centered-normal-scale*:
  assumes *c*: *0* < *c* **and** *v*: *0* ≤ *v*
  shows *distr* (*centered-normal* (*c* ∗ *v*)) *borel* (λ*x*. *x* / *sqrt c*) =
    centered-normal v
proof (*cases v* = *0*)
  case *True*
  show *?thesis* **using** *c True* **by** (*simp add*: *distr-return*)
next
  case *False*
  with *v* **have** *pos*: *0* < *v* **by** *simp*
  have *cv*: *0* < *c* ∗ *v* **using** *c pos* **by** *simp*
  have *prob*: *prob-space* (*centered-normal* (*c* ∗ *v*))

```
    by (rule prob-space-centered-normal)
  have sigma: 0 < sqrt (c * v) using cv by simp
  have alpha: 1 / sqrt c ≠ 0 using c by simp
  have identity: distr (centered-normal (c * v)) lborel id =
    centered-normal (c * v)
    unfolding id-def by (rule distr-id2) (simp add: centered-normal-def cv)
  have initial: distributed (centered-normal (c * v)) lborel id
    (normal-density 0 (sqrt (c * v)))
    using identity cv by (simp add: distributed-def centered-normal-def)
  have scaled: distributed (centered-normal (c * v)) lborel
    (λx. 0 + (1 / sqrt c) * id x)
    (normal-density (0 + (1 / sqrt c) * 0)
      (|1 / sqrt c| * sqrt (c * v)))
    by (rule prob-space.normal-density-affine[OF prob initial sigma alpha])
  have simplified: distributed (centered-normal (c * v)) lborel
    (λx. x / sqrt c) (normal-density 0 (sqrt v))
    using scaled c v by (simp add: real-sqrt-mult)
  have eq-borel: distr (centered-normal (c * v)) borel (λx. x / sqrt c) =
    distr (centered-normal (c * v)) lborel (λx. x / sqrt c)
    by (rule distr-cong) simp-all
  have density-eq: distr (centered-normal (c * v)) lborel
    (λx. x / sqrt c) = centered-normal v
    using distributed-distr-eq-density[OF simplified] pos
    by (simp add: centered-normal-def)
  show ?thesis using eq-borel density-eq by simp
qed

lemma integrable-centered-normal-moment:
  assumes 0 ≤ v
  shows integrable (centered-normal v) (λx. x^k)
proof (cases v = 0)
  case True
  then show ?thesis
    by (intro integrableI-bounded) (simp-all add: nn-integral-return)
next
  case False
  with assms have 0 < v by simp
  then show ?thesis
    using integrable-normal-moment[of sqrt v 0 k]
    by (simp add: centered-normal-def integrable-density)
qed

lemma centered-normal-mean:
  assumes 0 ≤ v
  shows (∫ x. x ∂centered-normal v) = 0
proof (cases v = 0)
  case True
  then show ?thesis by (simp add: integral-return)
next
```

```
  case False
  with assms have 0 < v by simp
  then have 0 < sqrt v by simp
  then show ?thesis
    using ‹0 < v› integral-normal-moment-nz-1[of sqrt v 0]
    by (simp add: centered-normal-def integral-density)
qed

lemma centered-normal-fourth-moment:
  assumes 0 ≤ v
  shows (∫ x. x^4 ∂centered-normal v) = 3 * v^2
proof (cases v = 0)
  case True
  then show ?thesis by (simp add: integral-return)
next
  case False
  with assms have v: 0 < v by simp
  then have sigma: 0 < sqrt v by simp
  have (∫ x. x^4 ∂centered-normal v) =
        (∫ x. normal-density 0 (sqrt v) x * x^4 ∂lborel)
    using v by (simp add: centered-normal-def integral-density)
  also have ... = 3 * v^2
    using integral-normal-moment-even[OF sigma, of 0 2]
    by (simp add: power2-eq-square fact-numeral)
  finally show ?thesis .
qed

lemma centered-normal-fourth-nn-integral:
  assumes v: 0 ≤ v
  shows (∫⁺x. ennreal (x^4) ∂centered-normal v) = ennreal (3 * v^2)
proof (cases v = 0)
  case True
  then show ?thesis
    by (simp add: centered-normal-def nn-integral-return)
next
  case False
  with v have pos: 0 < v by simp
  then have sigma: 0 < sqrt v by simp
  have integrable: integrable lborel (λx. normal-density 0 (sqrt v) x * x^4)
    using integrable-normal-moment[OF sigma, of 0 4] by simp
  have (∫⁺x. ennreal (x^4) ∂centered-normal v) =
    (∫⁺x. ennreal (normal-density 0 (sqrt v) x * x^4) ∂lborel)
   using pos by (simp add: centered-normal-def nn-integral-density ennreal-mult')
  also have ... = (∫ x. normal-density 0 (sqrt v) x * x^4 ∂lborel)
    by (rule nn-integral-eq-integral) (auto intro: integrable)
  also have ... = 3 * v^2
    using integral-normal-moment-even[OF sigma, of 0 2]
    by (simp add: power2-eq-square fact-numeral)
  finally show ?thesis .
```

qed

lemma *centered-normal-add-positive*:
assumes *prob*: *prob-space M*
and *indep*: *prob-space.indep-var M borel X borel Y*
and *measurable-X*: $X \in$ *borel-measurable M*
and *measurable-Y*: $Y \in$ *borel-measurable M*
and *v*: $0 < v$ **and** *w*: $0 < w$
and *X*: *distr M borel X* = *centered-normal v*
and *Y*: *distr M borel Y* = *centered-normal w*
shows *distr M borel* $(\lambda z.\ X\ z + Y\ z)$ = *centered-normal* $(v + w)$
proof −
interpret *P*: *prob-space M* **by** (*rule prob*)
have *eq-X*: *distr M lborel X* = *distr M borel X*
by (*rule distr-cong*) (*simp-all add*: *measurable-X*)
have *eq-Y*: *distr M lborel Y* = *distr M borel Y*
by (*rule distr-cong*) (*simp-all add*: *measurable-Y*)
have *distributed-X*: *distributed M lborel X* (*normal-density 0* (*sqrt v*))
using *X v measurable-X*
by (*simp add*: *distributed-def centered-normal-def eq-X*)
have *distributed-Y*: *distributed M lborel Y* (*normal-density 0* (*sqrt w*))
using *Y w measurable-Y*
by (*simp add*: *distributed-def centered-normal-def eq-Y*)
have *distributed-sum*: *distributed M lborel* $(\lambda z.\ X\ z + Y\ z)$
(*normal-density 0* (*sqrt* $(v + w)$))
using *P.add-indep-normal*[*OF indep - - distributed-X distributed-Y*]
v w
by (*simp add*: *power2-eq-square*)
have *eq-sum*: *distr M borel* $(\lambda z.\ X\ z + Y\ z)$ =
distr M lborel $(\lambda z.\ X\ z + Y\ z)$
by (*rule distr-cong*) (*simp-all add*: *measurable-X measurable-Y*)
show *?thesis*
using *distributed-distr-eq-density*[*OF distributed-sum*] *v w*
by (*simp add*: *centered-normal-def eq-sum*)
qed

lemma *centered-normal-add*:
assumes *prob*: *prob-space M*
and *indep*: *prob-space.indep-var M borel X borel Y*
and *measurable-X*: $X \in$ *borel-measurable M*
and *measurable-Y*: $Y \in$ *borel-measurable M*
and *v*: $0 \leq v$ **and** *w*: $0 \leq w$
and *X*: *distr M borel X* = *centered-normal v*
and *Y*: *distr M borel Y* = *centered-normal w*
shows *distr M borel* $(\lambda z.\ X\ z + Y\ z)$ = *centered-normal* $(v + w)$
proof (*cases* $v = 0$)
case *True*
have *ae*: *AE z in M*. $X\ z = 0$
proof (*rule AE-distrD*[*OF measurable-X*])

```
    have ae-zero: AE x in return borel (0::real). x = 0
      by (subst AE-return) (auto simp: Measurable.pred-def)
    have law0: distr M borel X = return borel 0 using X True by simp
    show AE x in distr M borel X. x = 0
      using ae-zero by (simp only: law0)
  qed
  have eq: distr M borel (λz. X z + Y z) = distr M borel Y
    by (rule distr-cong-AE[OF refl refl])
       (use ae measurable-X measurable-Y in ‹auto intro: borel-measurable-add›)
  show ?thesis using True Y eq by simp
next
  case v-pos: False
  show ?thesis
  proof (cases w = 0)
    case True
    have ae: AE z in M. Y z = 0
    proof (rule AE-distrD[OF measurable-Y])
      have ae-zero: AE y in return borel (0::real). y = 0
        by (subst AE-return) (auto simp: Measurable.pred-def)
      have law0: distr M borel Y = return borel 0 using Y True by simp
      show AE y in distr M borel Y. y = 0
        using ae-zero by (simp only: law0)
    qed
    have eq: distr M borel (λz. X z + Y z) = distr M borel X
      by (rule distr-cong-AE[OF refl refl])
         (use ae measurable-X measurable-Y in ‹auto intro: borel-measurable-add›)
    show ?thesis using True X eq by simp
  next
    case False
    with v-pos v w have 0 < v 0 < w by auto
    then show ?thesis
      by (rule centered-normal-add-positive[OF prob indep measurable-X measur-
able-Y - - X Y])
  qed
qed

end
theory Gaussian-Finite-Dimensional
  imports Gaussian-Increments
begin
```

2 Finite-dimensional laws from independent increments

The list of times will be assumed sorted, distinct, and nonnegative in subsequent results. Index 0 represents the increment from time 0 to ts!0. Out-of-range indices are assigned variance 0 to keep this definition total.

definition *increment-variance* :: *real list* ⇒ *nat* ⇒ *real* **where**
increment-variance ts i =
(*if i* ≥ *length ts then 0*
else if i = *0 then ts* ! *0*
else ts ! *i* − *ts* ! (*i* − *1*))

lemma *increment-variance-nonnegative*:
assumes *sorted ts* ∀ *t*∈*set ts*. *0* ≤ *t i* < *length ts*
shows *0* ≤ *increment-variance ts i*
using *assms* **by** (*cases i*) (*auto simp*: *increment-variance-def sorted-iff-nth-mono*)

definition *increment-law* :: *real list* ⇒ (*nat* ⇒ *real*) *measure* **where**
increment-law ts =
PiM {..<*length ts*} (λ*i*. *centered-normal* (*increment-variance ts i*))

lemma *prob-space-increment-law* [*intro*]: *prob-space* (*increment-law ts*)
unfolding *increment-law-def*
by (*rule prob-space-PiM*) (*simp add*: *prob-space-centered-normal*)

lemma *sets-increment-law* [*simp*]:
sets (*increment-law ts*) = *sets* (*PiM* {..<*length ts*} (λ-. *borel*))
unfolding *increment-law-def*
by (*rule sets-PiM-cong*) *simp-all*

lemma *increment-law-component*:
assumes *i* < *length ts*
shows *distr* (*increment-law ts*) (*centered-normal* (*increment-variance ts i*))
(λ*w*. *w i*) = *centered-normal* (*increment-variance ts i*)
unfolding *increment-law-def*
by (*rule distr-PiM-component*) (*auto simp*: *assms*)

definition *partial-sums* :: *nat* ⇒ (*nat* ⇒ *real*) ⇒ *nat* ⇒ *real* **where**
partial-sums n increments i =
(*if i* < *n then* (∑ *j*≤*i*. *increments j*) *else undefined*)

lemma *partial-sums-increment*:
assumes *0* < *i i* < *n*
shows *partial-sums n w i* − *partial-sums n w* (*i* − *1*) = *w i*
using *assms* **by** (*cases i*) (*simp-all add*: *partial-sums-def*)

definition *gaussian-finite-law* :: *real list* ⇒ (*nat* ⇒ *real*) *measure* **where**
gaussian-finite-law ts =
distr (*increment-law ts*) (*PiM* {..<*length ts*} (λ-. *borel*))
(*partial-sums* (*length ts*))

lemma *measurable-partial-sums*:
partial-sums (*length ts*) ∈
measurable (*increment-law ts*) (*PiM* {..<*length ts*} (λ-. *borel*))
proof (*rule measurable-PiM-single′*)

fix *i* **assume** *i*: $i \in \{..<length\ ts\}$
have *eq*: *borel-measurable* (*increment-law ts*) =
borel-measurable (*PiM* $\{..<length\ ts\}$ (λ-. *borel*))
by (*rule measurable-cong-sets*) *simp-all*
show (λ*w*. *partial-sums* (*length ts*) *w i*) ∈ *borel-measurable* (*increment-law ts*)
using *i* **by** (*auto simp*: *eq partial-sums-def intro*!: *borel-measurable-sum measurable-component-singleton*)
next
show *partial-sums* (*length ts*) ∈
space (*increment-law ts*) → (Π_E *i*∈$\{..<length\ ts\}$. *space borel*)
by (*auto simp*: *partial-sums-def space-PiM PiE-iff split*: *if-splits*)
qed

lemma *sets-gaussian-finite-law* [*simp*]:
sets (*gaussian-finite-law ts*) = *sets* (*PiM* $\{..<length\ ts\}$ (λ-. *borel*))
by (*simp add*: *gaussian-finite-law-def*)

lemma *prob-space-gaussian-finite-law* [*intro*]: *prob-space* (*gaussian-finite-law ts*)
unfolding *gaussian-finite-law-def*
by (*rule prob-space.prob-space-distr*[*OF prob-space-increment-law measurable-partial-sums*])

lemma *gaussian-finite-law-empty*:
gaussian-finite-law [] = *PiM* {} (λ-. *borel*)
proof −
have *gaussian-finite-law* [] =
distr (*count-space* {λ-. *undefined*}) (*count-space* {λ-. *undefined*}) (*partial-sums 0*)
by (*simp add*: *gaussian-finite-law-def increment-law-def PiM-empty*)
also have ... = *distr* (*count-space* {λ-. *undefined*}) (*count-space* {λ-. *undefined*}) *id*
by (*rule distr-cong*) (*auto simp*: *partial-sums-def*)
also have ... = *PiM* {} (λ-. *borel*)
by (*simp add*: *PiM-empty id-def*)
finally show *?thesis* **.**
qed

lemma *gaussian-finite-law-initial*:
assumes *ts*: *ts* ≠ []
shows *distr* (*gaussian-finite-law ts*) *borel* (λ*w*. *w 0*) =
centered-normal (*ts* ! *0*)
proof −
have *idx*: *0* < *length ts* **using** *ts* **by** *simp*
have *coord*: (λ*w*. *w 0*) ∈ *measurable* (*PiM* $\{..<length\ ts\}$ (λ-. *borel*)) *borel*
by (*rule measurable-component-singleton*) (*simp add*: *ts*)
have *distr* (*gaussian-finite-law ts*) *borel* (λ*w*. *w 0*) =
distr (*increment-law ts*) *borel* ((λ*w*. *w 0*) ∘ *partial-sums* (*length ts*))
unfolding *gaussian-finite-law-def*
by (*rule distr-distr*[*OF coord measurable-partial-sums*])
also have ... = *distr* (*increment-law ts*) *borel* (λ*w*. *w 0*)

```
    by (rule distr-cong) (simp-all add: partial-sums-def ts)
  also have ... = centered-normal (increment-variance ts 0)
    using increment-law-component[OF idx]
    by (simp cong: distr-cong)
  also have ... = centered-normal (ts ! 0)
    using idx by (simp add: increment-variance-def)
  finally show ?thesis .
qed

lemma gaussian-finite-law-starts-at-zero:
  assumes ts: ts ≠ [] ts ! 0 = 0
  shows AE w in gaussian-finite-law ts. w 0 = 0
proof −
  have coord: (λw. w 0) ∈ borel-measurable (gaussian-finite-law ts)
    by (simp add: gaussian-finite-law-def measurable-component-singleton ts)
  have law: distr (gaussian-finite-law ts) borel (λw. w 0) = return borel 0
    using gaussian-finite-law-initial[OF ts(1)] ts(2) by simp
  have ae-zero: AE x in return borel (0::real). x = 0
    by (subst AE-return) (auto simp: Measurable.pred-def)
  have AE x in distr (gaussian-finite-law ts) borel (λw. w 0). x = 0
    using ae-zero by (simp only: law)
  then show ?thesis by (rule AE-distrD[OF coord])
qed

lemma gaussian-finite-law-increment:
  assumes i: 0 < i i < length ts
  shows distr (gaussian-finite-law ts) borel (λw. w i − w (i − 1)) =
    centered-normal (increment-variance ts i)
proof −
  have first: (λw :: nat ⇒ real. w i) ∈
      borel-measurable (PiM {..<length ts} (λ-. borel))
    by (rule measurable-component-singleton) (use i in auto)
  have second: (λw :: nat ⇒ real. w (i − 1)) ∈
      borel-measurable (PiM {..<length ts} (λ-. borel))
    by (rule measurable-component-singleton) (use i in auto)
  have coord: (λw :: nat ⇒ real. w i − w (i − 1)) ∈
      borel-measurable (PiM {..<length ts} (λ-. borel))
    using first second by (rule borel-measurable-diff)
  have distr (gaussian-finite-law ts) borel (λw. w i − w (i − 1)) =
      distr (increment-law ts) borel
        ((λw. w i − w (i − 1)) ∘ partial-sums (length ts))
    unfolding gaussian-finite-law-def
    by (rule distr-distr[OF coord measurable-partial-sums])
  also have ... = distr (increment-law ts) borel (λw. w i)
   using partial-sums-increment[OF i] by (intro distr-cong) (auto simp: comp-def)
  also have ... = centered-normal (increment-variance ts i)
    using increment-law-component[OF i(2)] by (simp cong: distr-cong)
  finally show ?thesis .
qed
```

lemma *increment-law-independent*:
assumes *nonempty*: $ts \neq []$
shows *prob-space.indep-vars* (*increment-law ts*)
(λ*i*. *centered-normal* (*increment-variance ts i*)) (λ*i w*. *w i*)
{..<*length ts*}
proof −
let *?I* = {..<*length ts*}
let *?M* = λ*i*. *centered-normal* (*increment-variance ts i*)
have *prod*: *product-prob-space ?M*
by (*rule product-prob-spaceI*) (*rule prob-space-centered-normal*)
have *rv*: ⋀*i*. *i* ∈ *?I* ⟹
(λ*w*. *w i*) ∈ *measurable* (*increment-law ts*) (*?M i*)
by (*simp add*: *increment-law-def measurable-component-singleton*)
have *identity*: (λ*w*. λ*i*∈*?I*. *w i*) = (λ*w*. *restrict w ?I*)
by (*simp add*: *fun-eq-iff restrict-def*)
have *restricted*: *PiM ?I ?M* =
distr (*increment-law ts*) (*PiM ?I ?M*) (λ*w*. *restrict w ?I*)
unfolding *increment-law-def*
by (*rule product-prob-space.distr-restrict*[*OF prod*]) *simp-all*
have *law*: *distr* (*increment-law ts*) (*PiM ?I ?M*)
(λ*w*. λ*i*∈*?I*. *w i*) = *PiM ?I ?M*
by (*subst identity*) (*rule restricted*[*symmetric*])
have *iff*: *prob-space.indep-vars* (*increment-law ts*) *?M* (λ*i w*. *w i*) *?I* ⟷
distr (*increment-law ts*) (*PiM ?I ?M*) (λ*w*. λ*i*∈*?I*. *w i*) =
PiM ?I (λ*i*. *distr* (*increment-law ts*) (*?M i*) (λ*w*. *w i*))
by (*rule prob-space.indep-vars-iff-distr-eq-PiM'*[*OF prob-space-increment-law*])
(*use nonempty rv* **in** *auto*)
have *marginals*: *PiM ?I* (λ*i*. *distr* (*increment-law ts*) (*?M i*) (λ*w*. *w i*)) =
PiM ?I ?M
by (*rule PiM-cong*) (*auto intro*: *increment-law-component*)
show *?thesis* **using** *iff law marginals* **by** *simp*
qed

definition *coordinate-increments* :: *nat* ⇒ (*nat* ⇒ *real*) ⇒ *nat* ⇒ *real* **where**
coordinate-increments n w i =
(*if i* < *n then* (*if i* = *0 then w 0 else w i* − *w* (*i* − *1*)) *else undefined*)

lemma *coordinate-increments-partial-sums*:
coordinate-increments n (*partial-sums n w*) = *restrict w* {..<*n*}
proof (*rule ext*)
fix *i*
have *initial*: *0* < *n* ⟹ *partial-sums n w 0* = *w 0*
by (*simp add*: *partial-sums-def*)
show *coordinate-increments n* (*partial-sums n w*) *i* = *restrict w* {..<*n*} *i*
proof (*cases i* < *n*)
case *False*
then show *?thesis* **by** (*simp add*: *coordinate-increments-def restrict-def*)
next

```
    case True
    show ?thesis
    proof (cases i = 0)
      case True
      with ‹i < n› show ?thesis
        by (simp add: coordinate-increments-def restrict-def initial)
    next
      case False
      with True have pos: 0 < i by simp
      have step: partial-sums n w i − partial-sums n w (i − 1) = w i
        by (rule partial-sums-increment[OF pos True])
      with True False show ?thesis
        by (simp add: coordinate-increments-def restrict-def)
    qed
  qed
qed

lemma partial-sums-coordinate-increments:
  partial-sums n (coordinate-increments n w) = restrict w {..<n}
proof −
  have telescope: (∑ j≤i. coordinate-increments n w j) = w i
    if i: i < n for i
    using i
  proof (induction i)
    case 0
    then show ?case by (simp add: coordinate-increments-def)
  next
    case (Suc i)
    then show ?case by (simp add: coordinate-increments-def)
  qed
  show ?thesis using telescope
    by (intro ext) (auto simp: partial-sums-def restrict-def)
qed

lemma measurable-coordinate-increments:
  coordinate-increments (length ts) ∈
    measurable (PiM {..<length ts} (λ-. borel)) (increment-law ts)
proof −
  let ?I = {..<length ts}
  let ?B = λ-. borel
  have meas: coordinate-increments (length ts) ∈ measurable (PiM ?I ?B) (PiM ?I
?B)
  proof (rule measurable-PiM-single′)
    fix i assume i: i ∈ ?I
    have first: (λw :: nat ⇒ real. w i) ∈ borel-measurable (PiM ?I ?B)
      by (rule measurable-component-singleton) (use i in auto)
    show (λw. coordinate-increments (length ts) w i) ∈
      borel-measurable (PiM ?I ?B)
    proof (cases i = 0)
```

```
    case True
    with i first show ?thesis by (simp add: coordinate-increments-def)
  next
    case False
    have second: (λw :: nat ⇒ real. w (i − 1)) ∈
      borel-measurable (PiM ?I ?B)
      by (rule measurable-component-singleton) (use i False in auto)
    from i False first second show ?thesis
      by (simp add: coordinate-increments-def borel-measurable-diff)
  qed
 next
  show coordinate-increments (length ts) ∈
    space (PiM ?I ?B) → (Π_E i∈?I. space borel)
    by (auto simp: coordinate-increments-def space-PiM PiE-iff split: if-splits)
 qed
 show ?thesis using meas
  by (simp only: measurable-cong-sets[OF refl sets-increment-law[symmetric]])
qed

lemma gaussian-finite-law-coordinate-increments:
 distr (gaussian-finite-law ts) (increment-law ts)
  (coordinate-increments (length ts)) = increment-law ts
proof −
 let ?I = {..<length ts}
 have prod: product-prob-space (λi. centered-normal (increment-variance ts i))
  by (rule product-prob-spaceI) (rule prob-space-centered-normal)
 have distr (gaussian-finite-law ts) (increment-law ts)
    (coordinate-increments (length ts)) =
  distr (increment-law ts) (increment-law ts)
    (coordinate-increments (length ts) ∘ partial-sums (length ts))
  unfolding gaussian-finite-law-def
 by (rule distr-distr[OF measurable-coordinate-increments measurable-partial-sums])
 also have ... = distr (increment-law ts) (increment-law ts)
  (λw. restrict w ?I)
   using coordinate-increments-partial-sums by (intro distr-cong) (auto simp:
comp-def)
 also have ... = increment-law ts
  unfolding increment-law-def
  by (rule sym, rule product-prob-space.distr-restrict[OF prod]) simp-all
 finally show ?thesis .
qed

lemma gaussian-finite-law-independent-increments:
 assumes nonempty: ts ≠ []
 shows prob-space.indep-vars (gaussian-finite-law ts)
  (λi. centered-normal (increment-variance ts i))
  (λi w. coordinate-increments (length ts) w i) {..<length ts}
proof −
 let ?I = {..<length ts}
```

```
  let ?M = λi. centered-normal (increment-variance ts i)
  let ?D = coordinate-increments (length ts)
  have full: ?D ∈ measurable (gaussian-finite-law ts) (increment-law ts)
    using measurable-coordinate-increments
    by (simp only: measurable-cong-sets[OF sets-gaussian-finite-law refl])
  have component: (λw. w i) ∈ measurable (increment-law ts) (?M i)
    if i ∈ ?I for i
    unfolding increment-law-def
    by (rule measurable-component-singleton) (use that in auto)
  have rv: (λw. ?D w i) ∈ measurable (gaussian-finite-law ts) (?M i)
    if i ∈ ?I for i
   using measurable-compose[OF full component[OF that]] by (simp add: comp-def)
  have identity: (λw. λi∈?I. ?D w i) = ?D
    by (auto simp: fun-eq-iff coordinate-increments-def)
  have joint: distr (gaussian-finite-law ts) (PiM ?I ?M)
      (λw. λi∈?I. ?D w i) = PiM ?I ?M
    using gaussian-finite-law-coordinate-increments[of ts]
    by (simp only: identity increment-law-def)
  have marginals: distr (gaussian-finite-law ts) (?M i) (λw. ?D w i) = ?M i
    if i: i ∈ ?I for i
  proof −
    have distr (gaussian-finite-law ts) (?M i) (λw. ?D w i) =
      distr (distr (gaussian-finite-law ts) (increment-law ts) ?D)
        (?M i) (λw. w i)
      using distr-distr[OF component[OF i] full] by (simp add: comp-def)
    also have ... = ?M i
      using increment-law-component[of i ts] i
      by (simp add: gaussian-finite-law-coordinate-increments)
    finally show ?thesis .
  qed
  have iff: prob-space.indep-vars (gaussian-finite-law ts) ?M
      (λi w. ?D w i) ?I ⟷
    distr (gaussian-finite-law ts) (PiM ?I ?M)
      (λw. λi∈?I. ?D w i) =
    PiM ?I (λi. distr (gaussian-finite-law ts) (?M i) (λw. ?D w i))
   by (rule prob-space.indep-vars-iff-distr-eq-PiM'[OF prob-space-gaussian-finite-law])
      (use nonempty rv in auto)
  have product: PiM ?I (λi. distr (gaussian-finite-law ts) (?M i) (λw. ?D w i)) =
    PiM ?I ?M
    by (rule PiM-cong) (auto intro: marginals)
  show ?thesis using iff joint product by simp
qed

lemma gaussian-finite-law-independent-borel-increments:
  assumes ts ≠ []
  shows prob-space.indep-vars (gaussian-finite-law ts) (λ-. borel)
    (λi w. coordinate-increments (length ts) w i) {..<length ts}
proof −
 interpret P: prob-space gaussian-finite-law ts by (rule prob-space-gaussian-finite-law)
```

```
  have independent: P.indep-vars
    (λi. centered-normal (increment-variance ts i))
    (λi w. coordinate-increments (length ts) w i) {..<length ts}
    by (rule gaussian-finite-law-independent-increments[OF assms])
  show ?thesis
    using independent
    by (simp add: P.indep-vars-def2 cong: measurable-cong-sets)
qed

lemma gaussian-finite-law-coordinate-increment:
  assumes i: i < length ts
  shows distr (gaussian-finite-law ts) borel
    (λw. coordinate-increments (length ts) w i) =
    centered-normal (increment-variance ts i)
proof (cases i)
  case 0
  with i show ?thesis
    by (simp add: coordinate-increments-def increment-variance-def
      gaussian-finite-law-initial)
next
  case (Suc j)
  with i gaussian-finite-law-increment[of Suc j ts] show ?thesis
    by (simp add: coordinate-increments-def)
qed

lemma gaussian-finite-law-adjacent-independent:
  assumes bound: Suc i < length ts
  shows prob-space.indep-var (gaussian-finite-law ts)
    borel (λw. coordinate-increments (length ts) w i)
    borel (λw. coordinate-increments (length ts) w (Suc i))
proof −
 interpret P: prob-space gaussian-finite-law ts by (rule prob-space-gaussian-finite-law)
  let ?X = λi w. coordinate-increments (length ts) w i
  have nonempty: ts ≠ [] using bound by auto
  have independent: P.indep-vars (λ-. borel) ?X (insert i {Suc i})
   by (rule P.indep-vars-subset[OF gaussian-finite-law-independent-borel-increments[OF
nonempty]])
       (use bound in auto)
  have P.indep-var borel (?X i) borel (λw. ∑ j∈{Suc i}. ?X j w)
    by (rule P.indep-vars-sum[OF - - independent]) simp-all
  then show ?thesis by simp
qed

lemma gaussian-finite-law-adjacent-sum:
  assumes sorted: sorted ts and nonnegative: ∀ t∈set ts. 0 ≤ t
    and bound: Suc i < length ts
  shows distr (gaussian-finite-law ts) borel
    (λw. coordinate-increments (length ts) w i +
      coordinate-increments (length ts) w (Suc i)) =
```

```
    centered-normal (increment-variance ts i + increment-variance ts (Suc i))
proof −
 interpret P: prob-space gaussian-finite-law ts by (rule prob-space-gaussian-finite-law)
 let ?X = λw. coordinate-increments (length ts) w i
 let ?Y = λw. coordinate-increments (length ts) w (Suc i)
 have independent: P.indep-var borel ?X borel ?Y
   by (rule gaussian-finite-law-adjacent-independent[OF bound])
 have measurable-X: ?X ∈ borel-measurable (gaussian-finite-law ts)
   using independent unfolding P.indep-var-def P.indep-vars-def2
   by (auto split: bool.splits)
 have measurable-Y: ?Y ∈ borel-measurable (gaussian-finite-law ts)
   using independent unfolding P.indep-var-def P.indep-vars-def2
   by (auto split: bool.splits)
 have i: i < length ts using bound by simp
 show ?thesis
   by (rule centered-normal-add[OF prob-space-gaussian-finite-law independent
     measurable-X measurable-Y
     increment-variance-nonnegative[OF sorted nonnegative i]
     increment-variance-nonnegative[OF sorted nonnegative bound]
     gaussian-finite-law-coordinate-increment[OF i]
     gaussian-finite-law-coordinate-increment[OF bound]])
qed

definition delete-time :: real list ⇒ nat ⇒ real list where
  delete-time ts j = take j ts @ drop (Suc j) ts

lemma delete-time-length:
  assumes j < length ts
  shows length (delete-time ts j) = length ts − 1
  using assms by (simp add: delete-time-def)

lemma delete-time-nth:
  assumes j < length ts i < length (delete-time ts j)
  shows delete-time ts j ! i = (if i < j then ts ! i else ts ! Suc i)
  using assms by (auto simp: delete-time-def nth-append)

lemma increment-variance-delete-time-merged:
  assumes bound: Suc j < length ts
  shows increment-variance (delete-time ts j) j =
    increment-variance ts j + increment-variance ts (Suc j)
proof −
  have j: j < length ts using bound by simp
  have len: length (delete-time ts j) = length ts − 1
    by (rule delete-time-length[OF j])
  have idx: j < length (delete-time ts j) using bound len by simp
  have curr: delete-time ts j ! j = ts ! Suc j
    using delete-time-nth[OF j idx] by simp
  show ?thesis
  proof (cases j)
```

```
    case 0
    have nonempty: ts ≠ [] using bound by auto
    from 0 bound len curr nonempty show ?thesis
      by (simp add: increment-variance-def)
  next
    case (Suc k)
    have prev: delete-time ts j ! (j − 1) = ts ! (j − 1)
      using delete-time-nth[OF j, of j − 1] Suc idx by simp
    with Suc bound len curr idx show ?thesis
      by (simp add: increment-variance-def)
  qed
qed

definition delete-coordinate :: nat ⇒ nat ⇒ (nat ⇒ real) ⇒ nat ⇒ real where
  delete-coordinate n j w i =
    (if i < n − 1 then w (if i < j then i else Suc i) else undefined)

lemma delete-coordinate-merged-increment:
  assumes bound: Suc j < n
  shows coordinate-increments (n − 1) (delete-coordinate n j w) j =
    coordinate-increments n w j + coordinate-increments n w (Suc j)
proof (cases j)
  case 0
  have idx: j < n − 1 using bound by simp
  with 0 bound show ?thesis
    by (simp add: coordinate-increments-def delete-coordinate-def)
next
  case (Suc k)
  have idx: j < n − 1 using bound by simp
  with Suc bound show ?thesis
    by (simp add: coordinate-increments-def delete-coordinate-def)
qed

lemma increment-variance-delete-time-unaffected:
  assumes j: j < length ts and i: i < length (delete-time ts j)
    and different: i ≠ j
  shows increment-variance (delete-time ts j) i =
    increment-variance ts (if i < j then i else Suc i)
proof −
  have len: length (delete-time ts j) = length ts − 1
    by (rule delete-time-length[OF j])
  have bound: Suc i < length ts using i len by simp
  have curr: delete-time ts j ! i = (if i < j then ts ! i else ts ! Suc i)
    by (rule delete-time-nth[OF j i])
  show ?thesis
  proof (cases i)
    case 0
    have nonempty: ts ≠ [] using j by auto
    from 0 different i len bound curr nonempty show ?thesis
```

```
    by (simp add: increment-variance-def)
  next
    case (Suc k)
    have prev: delete-time ts j ! k = (if k < j then ts ! k else ts ! Suc k)
      by (rule delete-time-nth[OF j]) (use i Suc in auto)
    from Suc different i len bound curr prev show ?thesis
      by (auto simp: increment-variance-def)
  qed
qed

lemma delete-coordinate-unaffected-increment:
  assumes j: j < length ts and i: i < length (delete-time ts j)
    and different: i ≠ j
  shows coordinate-increments (length (delete-time ts j))
      (delete-coordinate (length ts) j w) i =
    coordinate-increments (length ts) w (if i < j then i else Suc i)
proof −
  have len: length (delete-time ts j) = length ts − 1
    by (rule delete-time-length[OF j])
  have bound: Suc i < length ts using i len by simp
  show ?thesis using i different len bound
    by (cases i) (auto simp: coordinate-increments-def delete-coordinate-def)
qed

lemma measurable-delete-coordinate:
  assumes j: j < length ts
  shows delete-coordinate (length ts) j ∈
    measurable (PiM {..<length ts} (λ-. borel))
      (PiM {..<length (delete-time ts j)} (λ-. borel))
proof (rule measurable-PiM-single')
  fix i assume i: i ∈ {..<length (delete-time ts j)}
  have len: length (delete-time ts j) = length ts − 1
    by (rule delete-time-length[OF j])
  have idx: (if i < j then i else Suc i) < length ts
    using i len by auto
  show (λw. delete-coordinate (length ts) j w i) ∈
    borel-measurable (PiM {..<length ts} (λ-. borel))
   using i len idx by (simp add: delete-coordinate-def measurable-component-singleton)
next
  show delete-coordinate (length ts) j ∈
    space (PiM {..<length ts} (λ-. borel)) →
      (Π_E i∈{..<length (delete-time ts j)}. space borel)
    using delete-time-length[OF j]
    by (auto simp: delete-coordinate-def space-PiM PiE-iff split: if-splits)
qed

lemma gaussian-finite-law-deleted-time-unaffected-increment:
  assumes j: j < length ts and i: i < length (delete-time ts j)
    and different: i ≠ j
```

```
  shows distr (gaussian-finite-law ts) borel
    (λw. coordinate-increments (length (delete-time ts j))
      (delete-coordinate (length ts) j w) i) =
    centered-normal (increment-variance (delete-time ts j) i)
proof −
  have old: (if i < j then i else Suc i) < length ts
    using i delete-time-length[OF j] by auto
  have distr (gaussian-finite-law ts) borel
    (λw. coordinate-increments (length (delete-time ts j))
      (delete-coordinate (length ts) j w) i) =
    distr (gaussian-finite-law ts) borel
      (λw. coordinate-increments (length ts) w
        (if i < j then i else Suc i))
    using delete-coordinate-unaffected-increment[OF j i different]
    by (intro distr-cong) auto
  also have ... = centered-normal
    (increment-variance ts (if i < j then i else Suc i))
    by (rule gaussian-finite-law-coordinate-increment[OF old])
  also have ... = centered-normal (increment-variance (delete-time ts j) i)
    by (simp only: increment-variance-delete-time-unaffected[OF j i different])
  finally show ?thesis .
qed

lemma gaussian-finite-law-deleted-time-merged-increment:
  assumes sorted: sorted ts and nonnegative: ∀ t∈set ts. 0 ≤ t
    and bound: Suc j < length ts
  shows distr (gaussian-finite-law ts) borel
    (λw. coordinate-increments (length (delete-time ts j))
      (delete-coordinate (length ts) j w) j) =
    centered-normal (increment-variance (delete-time ts j) j)
proof −
  have len: length (delete-time ts j) = length ts − 1
    by (rule delete-time-length) (use bound in auto)
  have distr (gaussian-finite-law ts) borel
    (λw. coordinate-increments (length (delete-time ts j))
      (delete-coordinate (length ts) j w) j) =
    distr (gaussian-finite-law ts) borel
      (λw. coordinate-increments (length ts) w j +
        coordinate-increments (length ts) w (Suc j))
    using delete-coordinate-merged-increment[OF bound]
    by (intro distr-cong) (auto simp: len)
  also have ... = centered-normal
    (increment-variance ts j + increment-variance ts (Suc j))
    by (rule gaussian-finite-law-adjacent-sum[OF sorted nonnegative bound])
  also have ... = centered-normal (increment-variance (delete-time ts j) j)
    by (simp only: increment-variance-delete-time-merged[OF bound])
  finally show ?thesis .
qed
```

```
definition deletion-block :: nat ⇒ nat ⇒ nat set where
  deletion-block j i =
    (if i = j then {j, Suc j} else {if i < j then i else Suc i})

lemma deletion-block-subset:
  assumes bound: Suc j < n and i: i < n − 1
  shows deletion-block j i ⊆ {..<n}
  using assms by (auto simp: deletion-block-def)

lemma deletion-blocks-disjoint:
  disjoint-family-on (deletion-block j) {..<n − 1}
  by (auto simp: disjoint-family-on-def deletion-block-def)

lemma gaussian-finite-law-deleted-time-independent-increments:
  assumes bound: Suc j < length ts
  shows prob-space.indep-vars (gaussian-finite-law ts) (λ-. borel)
    (λi w. coordinate-increments (length (delete-time ts j))
      (delete-coordinate (length ts) j w) i)
    {..<length (delete-time ts j)}
proof −
 interpret P: prob-space gaussian-finite-law ts by (rule prob-space-gaussian-finite-law)
  let ?n = length ts
  let ?I = {..<length (delete-time ts j)}
  let ?K = deletion-block j
  let ?X = λk w. coordinate-increments ?n w k
  let ?R = λi w. restrict (λk. ?X k w) (?K i)
  let ?Y = λi (u :: nat ⇒ real). if i = j then u j + u (Suc j)
    else u (if i < j then i else Suc i)
  have len: length (delete-time ts j) = ?n − 1
    by (rule delete-time-length) (use bound in auto)
  have nonempty: ts ≠ [] using bound by auto
  have independent: P.indep-vars (λ-. borel) ?X {..<?n}
    by (rule gaussian-finite-law-independent-borel-increments[OF nonempty])
  have blocks: P.indep-vars (λi. PiM (?K i) (λ-. borel)) ?R ?I
  proof (rule P.indep-vars-restrict[OF independent])
    fix i assume i: i ∈ ?I
    show ?K i ⊆ {..<?n}
      by (rule deletion-block-subset[OF bound]) (use i in ‹simp add: len›)
  next
    show disjoint-family-on ?K ?I
      using deletion-blocks-disjoint[of j ?n] by (simp only: len)
  qed
  have measurable: ?Y i ∈ borel-measurable (PiM (?K i) (λ-. borel))
    if i: i ∈ ?I for i
  proof (cases i = j)
    case True
    have a: (λu :: nat ⇒ real. u j) ∈
      borel-measurable (PiM (?K i) (λ-. borel))
      by (rule measurable-component-singleton) (simp add: deletion-block-def True)
```

```
    have b: (λu :: nat ⇒ real. u (Suc j)) ∈
      borel-measurable (PiM (?K i) (λ-. borel))
     by (rule measurable-component-singleton) (simp add: deletion-block-def True)
    have aj: (λu :: nat ⇒ real. u j) ∈
      borel-measurable (PiM (?K j) (λ-. borel))
      using a True by simp
    have bj: (λu :: nat ⇒ real. u (Suc j)) ∈
      borel-measurable (PiM (?K j) (λ-. borel))
      using b True by simp
    have sum-meas: (λ(u :: nat ⇒ real). u j + u (Suc j)) ∈
      borel-measurable (PiM (?K j) (λ-. borel))
      by (rule borel-measurable-add[OF aj bj])
    show ?thesis using sum-meas True by (simp only: True if-True)
  next
    case False
    have (if i < j then i else Suc i) ∈ ?K i
      by (simp add: deletion-block-def False)
    then show ?thesis using False
      by (simp add: measurable-component-singleton)
  qed
  have composed: P.indep-vars (λ-. borel) (λi w. ?Y i (?R i w)) ?I
    by (rule P.indep-vars-compose2[OF blocks measurable])
  have eq: ?Y i (?R i w) =
    coordinate-increments (length (delete-time ts j))
      (delete-coordinate ?n j w) i if i: i ∈ ?I for i w
  proof (cases i = j)
    case True
    from True len delete-coordinate-merged-increment[OF bound, of w]
    show ?thesis by (simp add: deletion-block-def)
  next
    case False
    with i bound show ?thesis
      by (simp add: deletion-block-def
        delete-coordinate-unaffected-increment[of j ts i w])
  qed
  have same: P.indep-vars (λ-. borel) (λi w. ?Y i (?R i w)) ?I =
    P.indep-vars (λ-. borel)
      (λi w. coordinate-increments (length (delete-time ts j))
        (delete-coordinate ?n j w) i) ?I
    by (rule P.indep-vars-cong) (use eq in auto)
  show ?thesis using composed same by simp
qed

lemma gaussian-finite-law-deleted-time-increment-law:
  assumes sorted: sorted ts and nonnegative: ∀ t∈set ts. 0 ≤ t
    and bound: Suc j < length ts
  shows distr (gaussian-finite-law ts) (increment-law (delete-time ts j))
    (λw. coordinate-increments (length (delete-time ts j))
      (delete-coordinate (length ts) j w)) = increment-law (delete-time ts j)
```

proof −
interpret P: *prob-space gaussian-finite-law ts* **by** (*rule prob-space-gaussian-finite-law*)
let *?ds* = *delete-time ts j*
let *?I* = {..<*length ?ds*}
let *?D* = λ*w*. *coordinate-increments* (*length ?ds*)
(*delete-coordinate* (*length ts*) *j w*)
have *j*: *j* < *length ts* **using** *bound* **by** *simp*
have *len*: *length ?ds* = *length ts* − *1* **by** (*rule delete-time-length*[*OF j*])
have *independent*: *P.indep-vars* (λ-. *borel*) (λ*i w*. *?D w i*) *?I*
by (*rule gaussian-finite-law-deleted-time-independent-increments*[*OF bound*])
have *rv*: (λ*w*. *?D w i*) ∈ *borel-measurable* (*gaussian-finite-law ts*)
if *i* ∈ *?I* **for** *i*
using *independent that* **by** (*simp add*: *P.indep-vars-def2*)
have *marginals*: *distr* (*gaussian-finite-law ts*) *borel* (λ*w*. *?D w i*) =
centered-normal (*increment-variance ?ds i*) **if** *i*: *i* ∈ *?I* **for** *i*
proof (*cases i* = *j*)
case *True*
then show *?thesis*
by (*simp only*: *gaussian-finite-law-deleted-time-merged-increment*
[*OF sorted nonnegative bound*])
next
case *False*
show *?thesis*
by (*rule gaussian-finite-law-deleted-time-unaffected-increment*[*OF j*])
(*use i False* **in** *auto*)
qed
have *positive*: *0* < *length ?ds* **using** *bound len* **by** *arith*
have *nonempty*: *?I* ≠ {} **using** *positive* **by** *auto*
have *identity*: (λ*w*. λ*i*∈*?I*. *?D w i*) = *?D*
by (*auto simp*: *fun-eq-iff coordinate-increments-def*)
have *iff*: *P.indep-vars* (λ-. *borel*) (λ*i w*. *?D w i*) *?I* ⟷
distr (*gaussian-finite-law ts*) (*PiM ?I* (λ-. *borel*))
(λ*w*. λ*i*∈*?I*. *?D w i*) =
PiM ?I (λ*i*. *distr* (*gaussian-finite-law ts*) *borel* (λ*w*. *?D w i*))
by (*rule P.indep-vars-iff-distr-eq-PiM′*) (*use nonempty rv* **in** *auto*)
have *product*: *PiM ?I* (λ*i*. *distr* (*gaussian-finite-law ts*) *borel* (λ*w*. *?D w i*)) =
increment-law ?ds
unfolding *increment-law-def*
by (*rule PiM-cong*) (*use marginals* **in** *auto*)
have *joint*: *distr* (*gaussian-finite-law ts*) (*PiM ?I* (λ-. *borel*)) *?D* =
increment-law ?ds
using *independent iff product* **by** (*simp only*: *identity*)
have *same*: *distr* (*gaussian-finite-law ts*) (*increment-law ?ds*) *?D* =
distr (*gaussian-finite-law ts*) (*PiM ?I* (λ-. *borel*)) *?D*
by (*rule distr-cong*) *auto*
show *?thesis* **using** *same joint* **by** *simp*
qed

theorem *gaussian-finite-law-delete-time*:

```
  assumes sorted: sorted ts and nonnegative: ∀ t∈set ts. 0 ≤ t
    and bound: Suc j < length ts
  shows distr (gaussian-finite-law ts)
    (PiM {..<length (delete-time ts j)} (λ-. borel))
    (delete-coordinate (length ts) j) = gaussian-finite-law (delete-time ts j)
proof −
  let ?ds = delete-time ts j
  let ?n = length ts
  let ?I = {..<length ?ds}
  let ?B = λ-. borel
  let ?del = delete-coordinate ?n j
  let ?D = coordinate-increments (length ?ds) ∘ ?del
  have j: j < ?n using bound by simp
  have len: length ?ds = ?n − 1 by (rule delete-time-length[OF j])
  have measurable-del: ?del ∈ measurable (gaussian-finite-law ts) (PiM ?I ?B)
    using measurable-delete-coordinate[OF j]
    by (simp only: measurable-cong-sets[OF sets-gaussian-finite-law refl])
 have measurable-D: ?D ∈ measurable (gaussian-finite-law ts) (increment-law ?ds)
  by (rule measurable-comp[OF measurable-del measurable-coordinate-increments[of
?ds]])
  have measurable-sums: partial-sums (length ?ds) ∈
    measurable (increment-law ?ds) (PiM ?I ?B)
    by (rule measurable-partial-sums)
  have inverse: partial-sums (length ?ds) (?D w) = ?del w for w
    by (auto simp: comp-def partial-sums-coordinate-increments len
      delete-coordinate-def restrict-def fun-eq-iff)
  have distr (gaussian-finite-law ts) (PiM ?I ?B) ?del =
    distr (gaussian-finite-law ts) (PiM ?I ?B)
      (partial-sums (length ?ds) ∘ ?D)
    using inverse by (intro distr-cong) (auto simp: comp-def)
  also have ... = distr (distr (gaussian-finite-law ts) (increment-law ?ds) ?D)
    (PiM ?I ?B) (partial-sums (length ?ds))
    by (rule distr-distr[OF measurable-sums measurable-D, symmetric])
  also have ... = gaussian-finite-law ?ds
      using gaussian-finite-law-deleted-time-increment-law[OF sorted nonnegative
bound]
    by (simp add: gaussian-finite-law-def comp-def)
  finally show ?thesis .
qed

lemma partial-sums-restrict:
  assumes k ≤ n
  shows restrict (partial-sums n w) {..<k} =
    partial-sums k (restrict w {..<k})
 using assms by (intro ext) (auto simp: restrict-def partial-sums-def intro!: sum.cong)

lemma increment-variance-take:
  assumes k ≤ length ts i < k
  shows increment-variance (take k ts) i = increment-variance ts i
```

proof −
have *len*: *length* (*take k ts*) = *k* **using** *assms* **by** *simp*
have *nth*: *take k ts* ! *j* = *ts* ! *j* **if** *j* < *k* **for** *j*
using *that assms* **by** *simp*
show *?thesis*
using *assms len nth* **by** (*cases i*) (*auto simp*: *increment-variance-def*)
qed

lemma *increment-law-prefix*:
assumes *k*: *k* ≤ *length ts*
shows *distr* (*increment-law ts*)
(*PiM* {..<*k*} (λ*i*. *centered-normal* (*increment-variance* (*take k ts*) *i*)))
(λ*w*. *restrict w* {..<*k*}) = *increment-law* (*take k ts*)
proof −
have *prod*: *product-prob-space* (λ*i*. *centered-normal* (*increment-variance ts i*))
by (*rule product-prob-spaceI*) (*rule prob-space-centered-normal*)
have *targets*: *PiM* {..<*k*} (λ*i*. *centered-normal* (*increment-variance* (*take k ts*)
i)) =
PiM {..<*k*} (λ*i*. *centered-normal* (*increment-variance ts i*))
by (*intro PiM-cong*) (*auto simp*: *k increment-variance-take*)
have *restricted*: *distr* (*PiM* {..<*length ts*}
(λ*i*. *centered-normal* (*increment-variance ts i*)))
(*PiM* {..<*k*} (λ*i*. *centered-normal* (*increment-variance ts i*)))
(λ*w*. *restrict w* {..<*k*}) =
PiM {..<*k*} (λ*i*. *centered-normal* (*increment-variance ts i*))
by (*rule sym*, *rule product-prob-space.distr-restrict*[*OF prod*]) (*use k* **in** *auto*)
show *?thesis*
unfolding *increment-law-def*
using *k* **by** (*simp only*: *length-take min-absorb2 targets restricted*)
qed

lemma *gaussian-finite-law-prefix*:
assumes *k*: *k* ≤ *length ts*
shows *distr* (*gaussian-finite-law ts*) (*PiM* {..<*k*} (λ-. *borel*))
(λ*w*. *restrict w* {..<*k*}) = *gaussian-finite-law* (*take k ts*)
proof −
let *?B* = λ-. *borel*
have *length*: *length* (*take k ts*) = *k* **using** *k* **by** *simp*
have *measurable-restrict*: (λ*w*. *restrict w* {..<*k*}) ∈
measurable (*PiM* {..<*length ts*} *?B*) (*PiM* {..<*k*} *?B*)
by (*rule measurable-restrict-subset*) (*use k* **in** *auto*)
have *measurable-sums*: *partial-sums k* ∈
measurable (*increment-law* (*take k ts*)) (*PiM* {..<*k*} *?B*)
using *measurable-partial-sums*[*of take k ts*] **by** (*simp only*: *length*)
have *target-sets*: *sets* (*increment-law* (*take k ts*)) = *sets* (*PiM* {..<*k*} *?B*)
by (*simp only*: *sets-increment-law length*)
have *measurable-increments*: (λ*w*. *restrict w* {..<*k*}) ∈
measurable (*increment-law ts*) (*increment-law* (*take k ts*))
using *measurable-restrict*

```
    by (simp only: measurable-cong-sets[OF sets-increment-law target-sets])
  have distr (gaussian-finite-law ts) (PiM {..<k} ?B)
      (λw. restrict w {..<k}) =
    distr (increment-law ts) (PiM {..<k} ?B)
      ((λw. restrict w {..<k}) ∘ partial-sums (length ts))
    unfolding gaussian-finite-law-def
    by (rule distr-distr[OF measurable-restrict measurable-partial-sums])
  also have ... = distr (increment-law ts) (PiM {..<k} ?B)
      (partial-sums k ∘ (λw. restrict w {..<k}))
    using partial-sums-restrict[OF k] by (intro distr-cong) (auto simp: comp-def)
  also have ... = distr (distr (increment-law ts) (increment-law (take k ts))
      (λw. restrict w {..<k})) (PiM {..<k} ?B) (partial-sums k)
    by (rule distr-distr[OF measurable-sums measurable-increments, symmetric])
  also have ... = gaussian-finite-law (take k ts)
  proof −
    have prefix: distr (increment-law ts) (increment-law (take k ts))
        (λw. restrict w {..<k}) = increment-law (take k ts)
      using increment-law-prefix[OF k]
      by (simp only: increment-law-def length)
    show ?thesis by (simp only: prefix gaussian-finite-law-def length)
  qed
  finally show ?thesis .
qed

corollary gaussian-finite-law-butlast:
  assumes ts ≠ []
  shows distr (gaussian-finite-law ts)
    (PiM {..<length (butlast ts)} (λ-. borel))
    (λw. restrict w {..<length (butlast ts)}) =
    gaussian-finite-law (butlast ts)
  using gaussian-finite-law-prefix[of length ts − 1 ts] assms
  by (simp add: butlast-conv-take)

corollary gaussian-finite-law-delete-time-any:
  assumes sorted: sorted ts and nonnegative: ∀ t∈set ts. 0 ≤ t
    and j: j < length ts
  shows distr (gaussian-finite-law ts)
    (PiM {..<length (delete-time ts j)} (λ-. borel))
    (delete-coordinate (length ts) j) = gaussian-finite-law (delete-time ts j)
proof (cases Suc j < length ts)
  case True
 show ?thesis by (rule gaussian-finite-law-delete-time[OF sorted nonnegative True])
next
  case False
  have last: Suc j = length ts using False j by arith
  have ds: delete-time ts j = take j ts
    using last by (simp add: delete-time-def)
  have dc: delete-coordinate (length ts) j = (λw. restrict w {..<j})
    using last by (auto simp: fun-eq-iff delete-coordinate-def restrict-def)
```

```
  have prefix: distr (gaussian-finite-law ts)
      (PiM {..<j} (λ-. borel)) (λw. restrict w {..<j}) =
      gaussian-finite-law (take j ts)
    by (rule gaussian-finite-law-prefix) (use j in auto)
  show ?thesis using prefix by (simp only: ds dc length-take min-absorb2[OF
less-imp-le[OF j]])
qed

lemma increment-variance-scale:
  increment-variance (map (λt. c * t) ts) i =
    c * increment-variance ts i
  by (auto simp: increment-variance-def algebra-simps)

lemma increment-law-scale:
  assumes c: 0 < c and sorted: sorted ts
    and nonnegative: ∀ t∈set ts. 0 ≤ t
  shows distr (increment-law (map (λt. c * t) ts)) (increment-law ts)
    (compose {..<length ts} (λx. x / sqrt c)) = increment-law ts
proof −
  let ?I = {..<length ts}
  let ?us = map (λt. c * t) ts
  have src: product-prob-space (λi. centered-normal (increment-variance ?us i))
    by (rule product-prob-spaceI) (rule prob-space-centered-normal)
  have tgt: product-prob-space (λi. centered-normal (increment-variance ts i))
    by (rule product-prob-spaceI) (rule prob-space-centered-normal)
  have base: (λx::real. x / sqrt c) ∈ borel-measurable borel
    by measurable
  have meas: (λx. x / sqrt c) ∈ measurable
    (centered-normal (increment-variance ?us i))
    (centered-normal (increment-variance ts i)) for i
    using base
   by (simp only: measurable-cong-sets[OF sets-centered-normal sets-centered-normal])
  have distr (increment-law ?us) (increment-law ts)
      (compose ?I (λx. x / sqrt c)) =
    PiM ?I (λi. distr (centered-normal (increment-variance ?us i))
      (centered-normal (increment-variance ts i)) (λx. x / sqrt c))
    unfolding increment-law-def
    by (simp only: length-map; rule distr-PiM-finite-prob-space[OF - src tgt meas];
simp)
  also have ... = increment-law ts
  proof −
    have component: distr (centered-normal (increment-variance ?us i))
      (centered-normal (increment-variance ts i)) (λx. x / sqrt c) =
      centered-normal (increment-variance ts i) if i ∈ ?I for i
    proof −
      have v: 0 ≤ increment-variance ts i
        by (rule increment-variance-nonnegative[OF sorted nonnegative]) (use that
in auto)
      have distr (centered-normal (increment-variance ?us i))
```

```
      (centered-normal (increment-variance ts i)) (λx. x / sqrt c) =
      distr (centered-normal (increment-variance ?us i)) borel
        (λx. x / sqrt c)
      by (rule distr-cong) simp-all
    also have ... = centered-normal (increment-variance ts i)
      by (simp only: increment-variance-scale centered-normal-scale[OF c v])
    finally show ?thesis .
  qed
  show ?thesis unfolding increment-law-def
    by (rule PiM-cong) (use component in auto)
 qed
 finally show ?thesis .
qed

lemma gaussian-finite-law-scale:
 assumes c: 0 < c and sorted: sorted ts
   and nonnegative: ∀ t∈set ts. 0 ≤ t
 shows distr (gaussian-finite-law (map (λt. c * t) ts))
   (PiM {..<length ts} (λ-. borel))
   (compose {..<length ts} (λx. x / sqrt c)) =
   gaussian-finite-law ts
proof −
 let ?I = {..<length ts}
 let ?F = compose ?I (λx. x / sqrt c)
 let ?us = map (λt. c * t) ts
 have map-meas: ?F ∈ measurable (PiM ?I (λ-. borel))
   (PiM ?I (λ-. borel))
 proof (rule measurable-PiM-single′)
   fix i assume i: i ∈ ?I
   have coord: (λw::nat ⇒ real. w i) ∈
     borel-measurable (PiM ?I (λ-. borel))
     by (rule measurable-component-singleton) (use i in auto)
   show (λw. ?F w i) ∈ borel-measurable (PiM ?I (λ-. borel))
     using coord i by (simp add: compose-def)
 next
   show ?F ∈ space (PiM ?I (λ-. borel)) →
     (Π_E i∈?I. space borel)
     by (auto simp: compose-def PiE-iff split: if-splits)
 qed
 have map-finite: ?F ∈ measurable (gaussian-finite-law ?us)
   (PiM ?I (λ-. borel))
   using map-meas by (simp only: length-map
     measurable-cong-sets[OF sets-gaussian-finite-law refl])
 have sums-meas: partial-sums (length ts) ∈
   measurable (increment-law ?us) (PiM ?I (λ-. borel))
   using measurable-partial-sums[of ?us] by (simp only: length-map)
 have inc-meas: ?F ∈ measurable (increment-law ?us) (increment-law ts)
   using map-meas by (simp only: length-map
     measurable-cong-sets[OF sets-increment-law sets-increment-law])
```

have *comm*: *partial-sums* (*length ts*) (*?F w*) =
?F (*partial-sums* (*length ts*) *w*) **for** *w*
by (*auto simp*: *fun-eq-iff partial-sums-def compose-def sum-divide-distrib*)
have *distr* (*gaussian-finite-law ?us*) (*PiM ?I* (λ-. *borel*)) *?F* =
distr (*increment-law ?us*) (*PiM ?I* (λ-. *borel*))
(*?F* ∘ *partial-sums* (*length ts*))
unfolding *gaussian-finite-law-def*
by (*simp only*: *length-map*; *rule distr-distr*[*OF map-meas sums-meas*])
also have ... = *distr* (*increment-law ?us*) (*PiM ?I* (λ-. *borel*))
(*partial-sums* (*length ts*) ∘ *?F*)
by (*intro distr-cong*) (*auto simp*: *comp-def comm*)
also have ... = *distr* (*distr* (*increment-law ?us*) (*increment-law ts*) *?F*)
(*PiM ?I* (λ-. *borel*)) (*partial-sums* (*length ts*))
by (*rule distr-distr*[*OF measurable-partial-sums inc-meas*, *symmetric*])
also have ... = *gaussian-finite-law ts*
using *increment-law-scale*[*OF c sorted nonnegative*]
by (*simp only*: *gaussian-finite-law-def*)
finally show *?thesis* **.**
qed

Single-time deletion consistency for the actual joint laws. AFP stochastic-process distributions form a product of marginals, not the joint law.

end
theory *Gaussian-Process*
imports *Gaussian-Finite-Dimensional HOL−Probability.Projective-Limit*
begin

3 From consistent laws to a Gaussian process

definition *ordered-times* :: *real set* ⇒ *real list* **where**
ordered-times T = *sorted-list-of-set T*

definition *time-index* :: *real set* ⇒ *real* ⇒ *nat* **where**
time-index T t = (*THE i*. *i* < *length* (*ordered-times T*) ∧ *ordered-times T* ! *i* = *t*)

lemma *ordered-times-finite*:
assumes *finite T*
shows *set* (*ordered-times T*) = *T distinct* (*ordered-times T*)
sorted (*ordered-times T*) *length* (*ordered-times T*) = *card T*
using *assms* **by** (*simp-all add*: *ordered-times-def*)

lemma *time-index-unique*:
assumes *finite*: *finite T* **and** *t*: *t* ∈ *T*
shows *time-index T t* < *length* (*ordered-times T*)
ordered-times T ! *time-index T t* = *t*
proof −
have *membership*: *t* ∈ *set* (*ordered-times T*)
using *ordered-times-finite*[*OF finite*] *t* **by** *simp*

```
  then obtain i where i: i < length (ordered-times T)
    ordered-times T ! i = t by (auto simp: in-set-conv-nth)
  have unique: j = i if j < length (ordered-times T)
    ordered-times T ! j = t for j
    using ordered-times-finite(2)[OF finite] that i
    by (auto simp: distinct-conv-nth)
  have equivalence: (j < length (ordered-times T) ∧ ordered-times T ! j = t) ⟷
j = i
    for j using i unique by blast
  have index: time-index T t = i
    unfolding time-index-def by (simp only: equivalence; rule the-equality; simp)
  show time-index T t < length (ordered-times T)
    ordered-times T ! time-index T t = t using index i by auto
qed

lemma ordered-times-delete:
  assumes finite: finite T and t: t ∈ T
  shows ordered-times (T − {t}) =
    delete-time (ordered-times T) (time-index T t)
proof −
  let ?xs = ordered-times T
  let ?j = time-index T t
  have j: ?j < length ?xs and nth: ?xs ! ?j = t
    using time-index-unique[OF finite t] by auto
  have absent: t ∉ set (take ?j ?xs)
  proof
    assume t ∈ set (take ?j ?xs)
    then obtain k where k: k < ?j ?xs ! k = t
      by (auto simp: in-set-conv-nth)
    have k = ?j
      using k nth j ordered-times-finite(2)[OF finite]
      by (auto simp: distinct-conv-nth)
    with k show False by simp
  qed
  have split: ?xs = take ?j ?xs @ t # drop (Suc ?j) ?xs
    using id-take-nth-drop[OF j] nth by simp
  have remove: remove1 t ?xs = delete-time ?xs ?j
    by (subst split) (simp add: delete-time-def remove1-append absent)
  have ordered-times (T − {t}) = remove1 t ?xs
    using finite by (simp add: ordered-times-def sorted-list-of-set-remove)
  with remove show ?thesis by simp
qed

definition set-coordinates :: real set ⇒ (nat ⇒ real) ⇒ real ⇒ real where
  set-coordinates T w = (λt∈T. w (time-index T t))

lemma set-coordinates-delete:
  assumes finite: finite T and t: t ∈ T
  shows set-coordinates (T − {t})
```

```
    (delete-coordinate (length (ordered-times T)) (time-index T t) w) =
   restrict (set-coordinates T w) (T − {t})
proof (rule ext)
 fix s
 show set-coordinates (T − {t})
    (delete-coordinate (length (ordered-times T)) (time-index T t) w) s =
   restrict (set-coordinates T w) (T − {t}) s
 proof (cases s ∈ T − {t})
  case False
  have lhs: set-coordinates (T − {t})
   (delete-coordinate (length (ordered-times T)) (time-index T t) w) s = undefined
    using False by (simp only: set-coordinates-def restrict-def if-False)
  have rhs: restrict (set-coordinates T w) (T − {t}) s = undefined
    using False by (simp only: restrict-def if-False)
  show ?thesis using lhs rhs by simp
 next
  case True
  let ?xs = ordered-times T
  let ?ys = ordered-times (T − {t})
  let ?j = time-index T t
  let ?i = time-index (T − {t}) s
  let ?k = if ?i < ?j then ?i else Suc ?i
  have j: ?j < length ?xs by (rule time-index-unique(1)[OF finite t])
  have i: ?i < length ?ys and y: ?ys ! ?i = s
    using time-index-unique[OF - True] finite by auto
  have deletion: ?ys = delete-time ?xs ?j
    by (rule ordered-times-delete[OF finite t])
  have len: length ?ys = length ?xs − 1
    by (simp add: deletion delete-time-length[OF j])
  have k: ?k < length ?xs using i len by auto
  have xy: ?ys ! ?i = ?xs ! ?k
    using delete-time-nth[OF j, of ?i] i deletion
    by (cases ?i < ?j) auto
  have x: ?xs ! ?k = s using xy y by simp
  have s-index: time-index T s < length ?xs ?xs ! time-index T s = s
    using time-index-unique[OF finite] True by auto
  have inj: inj-on (nth ?xs) {..<length ?xs}
    using ordered-times-finite(2)[OF finite]
    by (auto simp: distinct-conv-nth inj-on-def)
  have index: time-index T s = ?k
    by (rule inj-onD[OF inj]) (use s-index k x in auto)
  show ?thesis using True i len index
    by (simp add: set-coordinates-def delete-coordinate-def restrict-def)
 qed
qed

lemma ordered-times-set:
 assumes sorted: sorted ts and distinct: distinct ts
 shows ordered-times (set ts) = ts
```

using *assms* **by** (*simp add*: *ordered-times-def sorted-list-of-set-sort-remdups sorted-sort-id*)

lemma *time-index-nth*:
assumes *sorted*: *sorted ts* **and** *distinct*: *distinct ts* **and** *i*: $i < length\ ts$
shows *time-index* (*set ts*) (*ts* ! *i*) = *i*
proof −
have *finite*: *finite* (*set ts*) **by** *simp*
have *member*: *ts* ! *i* ∈ *set ts* **using** *i* **by** *simp*
have *nth*: *ordered-times* (*set ts*) ! *time-index* (*set ts*) (*ts* ! *i*) = *ts* ! *i*
using *time-index-unique*(*2*)[*OF finite member*] **.**
have *bound*: *time-index* (*set ts*) (*ts* ! *i*) < *length ts*
using *time-index-unique*(*1*)[*OF finite member*]
ordered-times-set[*OF sorted distinct*] **by** *simp*
show *?thesis* **using** *distinct nth bound i ordered-times-set*[*OF sorted distinct*]
by (*metis distinct-conv-nth*)
qed

definition *list-coordinates* :: *real list* ⇒ (*real* ⇒ *real*) ⇒ *nat* ⇒ *real* **where**
list-coordinates ts w = (λ*i*∈{..<*length ts*}. *w* (*ts* ! *i*))

lemma *measurable-list-coordinates*:
list-coordinates ts ∈
measurable (*PiM* (*set ts*) (λ-. *borel*))
(*PiM* {..<*length ts*} (λ-. *borel*))
proof (*rule measurable-PiM-single′*)
fix *i* **assume** *i*: *i* ∈ {..<*length ts*}
have *coord*: (λ*w* :: *real* ⇒ *real*. *w* (*ts* ! *i*)) ∈
borel-measurable (*PiM* (*set ts*) (λ-. *borel*))
by (*rule measurable-component-singleton*) (*use i* **in** *auto*)
show (λ*w*. *list-coordinates ts w i*) ∈
borel-measurable (*PiM* (*set ts*) (λ-. *borel*))
using *i coord* **by** (*simp add*: *list-coordinates-def*)
next
show *list-coordinates ts* ∈
space (*PiM* (*set ts*) (λ-. *borel*)) →
(Π$_E$ *i*∈{..<*length ts*}. *space borel*)
by (*auto simp*: *list-coordinates-def space-PiM PiE-iff split*: *if-splits*)
qed

lemma *list-coordinates-set-coordinates*:
assumes *sorted*: *sorted ts* **and** *distinct*: *distinct ts*
shows *list-coordinates ts* (*set-coordinates* (*set ts*) *w*) =
restrict w {..<*length ts*}
using *assms*
by (*auto simp*: *list-coordinates-def set-coordinates-def restrict-def*
fun-eq-iff time-index-nth)

definition *gaussian-set-law* :: *real set* ⇒ (*real* ⇒ *real*) *measure* **where**
gaussian-set-law T =

```
    distr (gaussian-finite-law (ordered-times T)) (PiM T (λ-. borel))
      (set-coordinates T)

lemma measurable-set-coordinates:
  assumes finite: finite T
  shows set-coordinates T ∈
    measurable (PiM {..<length (ordered-times T)} (λ-. borel))
      (PiM T (λ-. borel))
proof (rule measurable-PiM-single')
  fix t assume t: t ∈ T
  show (λw. set-coordinates T w t) ∈
    borel-measurable (PiM {..<length (ordered-times T)} (λ-. borel))
    using time-index-unique(1)[OF finite t]
    by (simp add: set-coordinates-def measurable-component-singleton t)
next
  show set-coordinates T ∈
    space (PiM {..<length (ordered-times T)} (λ-. borel)) →
      (Π_E t∈T. space borel)
    by (auto simp: set-coordinates-def space-PiM PiE-iff split: if-splits)
qed

lemma prob-space-gaussian-set-law:
  assumes finite T
  shows prob-space (gaussian-set-law T)
proof −
  have measurable: set-coordinates T ∈
    measurable (gaussian-finite-law (ordered-times T)) (PiM T (λ-. borel))
    using measurable-set-coordinates[OF assms]
    by (simp only: measurable-cong-sets[OF sets-gaussian-finite-law refl])
  show ?thesis unfolding gaussian-set-law-def
    by (rule prob-space.prob-space-distr[OF prob-space-gaussian-finite-law measur-
able])
qed

lemma sets-gaussian-set-law:
  sets (gaussian-set-law T) = sets (PiM T (λ-. borel))
  by (simp add: gaussian-set-law-def)

lemma gaussian-set-law-ordered-list:
  assumes sorted: sorted ts and distinct: distinct ts
  shows distr (gaussian-set-law (set ts))
    (PiM {..<length ts} (λ-. borel)) (list-coordinates ts) =
    gaussian-finite-law ts
proof −
  let ?I = {..<length ts}
  let ?B = λ-. borel
  let ?T = set ts
  have ordered: ordered-times ?T = ts
    by (rule ordered-times-set[OF sorted distinct])
```

```
  have set-meas: set-coordinates ?T ∈
    measurable (gaussian-finite-law ts) (PiM ?T ?B)
    using measurable-set-coordinates[of ?T]
    by (simp add: ordered measurable-cong-sets[OF sets-gaussian-finite-law refl])
  have list-meas: list-coordinates ts ∈
    measurable (PiM ?T ?B) (PiM ?I ?B)
    by (rule measurable-list-coordinates)
  have identity: restrict w ?I = w if w ∈ space (gaussian-finite-law ts) for w
  proof −
    have space: space (gaussian-finite-law ts) = space (PiM ?I ?B)
      by (rule sets-eq-imp-space-eq) (rule sets-gaussian-finite-law)
    show ?thesis using that by (auto simp: space space-PiM PiE-iff
      extensional-def restrict-def fun-eq-iff)
  qed
  have distr (gaussian-set-law ?T) (PiM ?I ?B) (list-coordinates ts) =
    distr (gaussian-finite-law ts) (PiM ?I ?B)
      (list-coordinates ts ∘ set-coordinates ?T)
    unfolding gaussian-set-law-def
    using distr-distr[OF list-meas set-meas] by (simp only: ordered)
  also have ... = distr (gaussian-finite-law ts) (PiM ?I ?B)
    (λw. restrict w ?I)
    using list-coordinates-set-coordinates[OF sorted distinct]
    by (intro distr-cong) (auto simp: comp-def)
  also have ... = distr (gaussian-finite-law ts) (PiM ?I ?B) id
    by (rule distr-cong) (use identity in auto)
  also have ... = gaussian-finite-law ts
    by (simp add: id-def distr-id2)
  finally show ?thesis .
qed

lemma gaussian-set-law-delete:
  assumes finite: finite T and nonnegative: T ⊆ {0..}
    and t: t ∈ T
  shows distr (gaussian-set-law T) (PiM (T − {t}) (λ-. borel))
    (λw. restrict w (T − {t})) = gaussian-set-law (T − {t})
proof −
  let ?S = T − {t}
  let ?xs = ordered-times T
  let ?ys = ordered-times ?S
  let ?j = time-index T t
  let ?B = λ-. borel
  let ?del = delete-coordinate (length ?xs) ?j
  have j: ?j < length ?xs by (rule time-index-unique(1)[OF finite t])
  have deletion: ?ys = delete-time ?xs ?j
    by (rule ordered-times-delete[OF finite t])
  have ordered: sorted ?xs using ordered-times-finite(3)[OF finite] .
  have nonneg: ∀ u∈set ?xs. 0 ≤ u
    using nonnegative ordered-times-finite(1)[OF finite] by auto
  have finite-S: finite ?S using finite by simp
```

```
  have coord-meas: set-coordinates T ∈
    measurable (gaussian-finite-law ?xs) (PiM T ?B)
    using measurable-set-coordinates[OF finite]
    by (simp only: measurable-cong-sets[OF sets-gaussian-finite-law refl])
  have restrict-meas: (λw. restrict w ?S) ∈
    measurable (PiM T ?B) (PiM ?S ?B)
    by (rule measurable-restrict-subset) auto
  have del-meas: ?del ∈ measurable (gaussian-finite-law ?xs)
      (PiM {..<length ?ys} ?B)
    using measurable-delete-coordinate[of ?j ?xs, OF j]
    by (simp only: deletion measurable-cong-sets[OF sets-gaussian-finite-law refl])
  have set-meas: set-coordinates ?S ∈
    measurable (PiM {..<length ?ys} ?B) (PiM ?S ?B)
    by (rule measurable-set-coordinates[OF finite-S])
  have distr (gaussian-set-law T) (PiM ?S ?B) (λw. restrict w ?S) =
    distr (gaussian-finite-law ?xs) (PiM ?S ?B)
      ((λw. restrict w ?S) ∘ set-coordinates T)
    unfolding gaussian-set-law-def
    by (rule distr-distr[OF restrict-meas coord-meas])
  also have ... = distr (gaussian-finite-law ?xs) (PiM ?S ?B)
    (set-coordinates ?S ∘ ?del)
    using set-coordinates-delete[OF finite t]
    by (intro distr-cong) (auto simp: comp-def)
  also have ... = distr (distr (gaussian-finite-law ?xs)
      (PiM {..<length ?ys} ?B) ?del) (PiM ?S ?B) (set-coordinates ?S)
    by (rule distr-distr[OF set-meas del-meas, symmetric])
  also have ... = gaussian-set-law ?S
    using gaussian-finite-law-delete-time-any[OF ordered nonneg j]
    by (simp only: deletion gaussian-set-law-def)
  finally show ?thesis .
qed

lemma gaussian-set-law-consistent:
  assumes subset: J ⊆ H and finite: finite H
    and nonnegative: H ⊆ {0..}
  shows gaussian-set-law J =
    distr (gaussian-set-law H) (PiM J (λ-. borel)) (λw. restrict w J)
proof −
  let ?B = λ-. borel
  have finite-J: finite J by (rule finite-subset[OF subset finite])
  have J-nonnegative: J ⊆ {0..} using subset nonnegative by auto
  have consistent: distr (gaussian-set-law (J ∪ K)) (PiM J ?B)
      (λw. restrict w J) = gaussian-set-law J
    if finK: finite K and nonnegK: K ⊆ {0..} and disjK: K ∩ J = {}
    for K
    using finK nonnegK disjK
  proof (induction K rule: finite-induct)
    case empty
    have space: space (gaussian-set-law J) = space (PiM J ?B)
```

```
    using sets-gaussian-set-law by (rule sets-eq-imp-space-eq)
  have identity: restrict w J = w if w ∈ space (gaussian-set-law J) for w
    using that by (auto simp: space space-PiM PiE-iff extensional-def
      restrict-def fun-eq-iff)
  have distr (gaussian-set-law J) (PiM J ?B) (λw. restrict w J) =
    distr (gaussian-set-law J) (PiM J ?B) id
    by (rule distr-cong) (use identity in auto)
  also have ... = gaussian-set-law J
    by (simp add: id-def distr-id2 sets-gaussian-set-law)
  finally show ?case by (simp only: Un-empty-right)
next
  case (insert t K)
  let ?M = J ∪ K
  let ?L = J ∪ insert t K
  have t: t ∉ ?M using insert by auto
  have nonnegM: ?M ⊆ {0..} using insert J-nonnegative by auto
  have nonnegL: ?L ⊆ {0..} using insert J-nonnegative by auto
  have finiteL: finite ?L using insert finite-J by simp
  have del: ?L − {t} = ?M using t by auto
  have one: distr (gaussian-set-law ?L) (PiM ?M ?B)
    (λw. restrict w ?M) = gaussian-set-law ?M
    using gaussian-set-law-delete[OF finiteL nonnegL, of t]
    by (simp only: del) simp
  have IH: distr (gaussian-set-law ?M) (PiM J ?B)
    (λw. restrict w J) = gaussian-set-law J
    by (rule insert.IH) (use insert in auto)
  have restrict-J: (λw. restrict w J) ∈
    measurable (PiM ?M ?B) (PiM J ?B)
    by (rule measurable-restrict-subset) auto
  have restrict-M-base: (λw. restrict w ?M) ∈
    measurable (PiM ?L ?B) (PiM ?M ?B)
    by (rule measurable-restrict-subset) auto
  have restrict-M: (λw. restrict w ?M) ∈
    measurable (gaussian-set-law ?L) (PiM ?M ?B)
    using restrict-M-base
    by (simp only: measurable-cong-sets[OF sets-gaussian-set-law refl])
  have distr (gaussian-set-law ?L) (PiM J ?B) (λw. restrict w J) =
    distr (gaussian-set-law ?L) (PiM J ?B)
      ((λw. restrict w J) ∘ (λw. restrict w ?M))
    by (intro distr-cong) (auto simp: comp-def restrict-def fun-eq-iff)
  also have ... = distr (distr (gaussian-set-law ?L) (PiM ?M ?B)
      (λw. restrict w ?M)) (PiM J ?B) (λw. restrict w J)
    by (rule distr-distr[OF restrict-J restrict-M, symmetric])
  also have ... = gaussian-set-law J using one IH by simp
  finally show ?case .
qed
have diff: finite (H − J) using finite by simp
have H-eq: H = J ∪ (H − J) using subset by auto
have law: distr (gaussian-set-law (J ∪ (H − J))) (PiM J ?B)
```

```
    (λw. restrict w J) = gaussian-set-law J
   by (rule consistent[OF diff]) (use nonnegative in auto)
  show ?thesis using law H-eq by metis
qed

interpretation gaussian: projective-family {0..} :: real set
  gaussian-set-law λ-. borel :: real measure
proof (intro projective-family.intro)
  fix J H :: real set
  assume J ⊆ H finite H H ⊆ {0..}
  then show gaussian-set-law J =
    distr (gaussian-set-law H) (PiM J (λ-. borel)) (λw. restrict w J)
    by (rule gaussian-set-law-consistent)
next
  fix J :: real set
  assume finite J J ⊆ {0..}
  show prob-space (gaussian-set-law J)
    by (rule prob-space-gaussian-set-law[OF ‹finite J›])
qed

interpretation gaussian-limit: polish-projective {0..} :: real set gaussian-set-law
  by (rule polish-projective.intro[OF gaussian.projective-family-axioms])

definition gaussian-path-law :: (real ⇒ real) measure where
  gaussian-path-law =
    projective-family.lim ({0..} :: real set) gaussian-set-law (λ-. borel)

lemma prob-space-gaussian-path-law: prob-space gaussian-path-law
  unfolding gaussian-path-law-def
  by (rule gaussian-limit.P.prob-space-axioms)

lemma sets-gaussian-path-law:
  sets gaussian-path-law = sets (PiM {0..} (λ-. borel :: real measure))
  by (simp only: gaussian-path-law-def gaussian.sets-lim)

lemma gaussian-limit-marginal:
  assumes finite: finite J and subset: J ⊆ {0..}
  shows distr gaussian-path-law (PiM J (λ-. borel))
      (λw. restrict w J) = gaussian-set-law J
proof (rule measure-eqI)
  show sets (distr gaussian-path-law (PiM J (λ-. borel))
      (λw. restrict w J)) = sets (gaussian-set-law J)
    by (simp add: sets-gaussian-set-law)
next
  fix A assume A: A ∈ sets (distr gaussian-path-law
    (PiM J (λ-. borel)) (λw. restrict w J))
  have A': A ∈ sets (PiM J (λ-. borel)) using A by simp
  have measurable: (λw. restrict w J) ∈
      measurable gaussian-path-law (PiM J (λ-. borel))
```

```
    using measurable-restrict-subset[OF subset, of λ-. borel]
    by (simp only: gaussian-path-law-def
      measurable-cong-sets[OF gaussian.sets-lim refl])
  have emb-eq: prod-emb ({0..} :: real set) (λ-. borel :: real measure) J A =
    (λw. restrict w J) -` A ∩ space gaussian-path-law
    by (simp add: prod-emb-def gaussian-path-law-def space-PiM)
  have emeasure (distr gaussian-path-law (PiM J (λ-. borel))
      (λw. restrict w J)) A =
    emeasure gaussian-path-law ((λw. restrict w J) -` A ∩ space gaussian-path-law)
    by (rule emeasure-distr[OF measurable A'])
  also have ... = emeasure gaussian-path-law
    (prod-emb {0..} (λ-. borel) J A)
    using emb-eq by simp
  also have ... = emeasure (gaussian-set-law J) A
    using gaussian-limit.emeasure-lim-emb[OF subset finite A']
    by (simp only: gaussian-path-law-def)
  finally show emeasure (distr gaussian-path-law (PiM J (λ-. borel))
      (λw. restrict w J)) A = emeasure (gaussian-set-law J) A .
qed

interpretation gaussian-path: prob-space gaussian-path-law
  by (rule prob-space-gaussian-path-law)

lemma gaussian-path-coordinate-measurable:
  assumes t ∈ {0..}
  shows (λw :: real ⇒ real. w t) ∈ borel-measurable gaussian-path-law
  using measurable-component-singleton[of t {0..} λ-. borel :: real measure]
    assms
  by (simp only: measurable-cong-sets[OF sets-gaussian-path-law refl])

lemma gaussian-path-coordinates-measurable:
  ∀ t∈{0..}. (λw :: real ⇒ real. w t) ∈ borel-measurable gaussian-path-law
  using gaussian-path-coordinate-measurable by auto

definition gaussian-process :: (real, real ⇒ real, real) stochastic-process where
  gaussian-process = prob-space.process-of gaussian-path-law borel {0..}
    (λt w. w t) 0

lemma gaussian-process-index: proc-index gaussian-process = {0..}
  by (simp add: gaussian-process-def)

lemma gaussian-process-source: proc-source gaussian-process = gaussian-path-law
  using gaussian-path-coordinates-measurable
  by (simp add: gaussian-process-def)

lemma gaussian-process-target: proc-target gaussian-process = borel
  using gaussian-path-coordinates-measurable
  by (simp add: gaussian-process-def)
```

lemma *gaussian-process-coordinate*:
assumes $t \in \{0..\}$
shows *process gaussian-process t* = $(\lambda w.\ w\ t)$
using *gaussian-path-coordinates-measurable assms*
by (*simp add*: *gaussian-process-def gaussian-path.process-of-apply*)

lemma *gaussian-process-joint-law*:
assumes *finite*: *finite J* **and** *subset*: $J \subseteq \{0..\}$
shows *distr* (*proc-source gaussian-process*)
(*PiM J* (λ-. *proc-target gaussian-process*))
$(\lambda w.\ \lambda t{\in}J.$ *process gaussian-process t w*) = *gaussian-set-law J*
proof −
have *equal*: $(\lambda w.\ \lambda t{\in}J.$ *process gaussian-process t w*) =
$(\lambda w.$ *restrict w J*)
using *subset* **by** (*auto simp*: *fun-eq-iff restrict-def gaussian-process-coordinate*)
show *?thesis* **using** *gaussian-limit-marginal*[*OF finite subset*]
by (*simp only*: *gaussian-process-source gaussian-process-target equal*)
qed

lemma *measurable-gaussian-process-ordered-coordinates*:
$(\lambda w.\ \lambda i{\in}\{..{<}$*length ts*$\}.$ *process gaussian-process* (*ts* ! *i*) *w*) ∈
measurable (*proc-source gaussian-process*)
(*PiM* $\{..{<}$*length ts*$\}$ (λ-. *borel*))
proof (*rule measurable-PiM-single′*)
fix *i* **assume** $i \in \{..{<}$*length ts*$\}$
show $(\lambda w.\ (\lambda i{\in}\{..{<}$*length ts*$\}.$ *process gaussian-process* (*ts* ! *i*) *w*) *i*) ∈
borel-measurable (*proc-source gaussian-process*)
using *stochastic-process-measurable*[*of gaussian-process ts* ! *i*]
by (*simp add*: *gaussian-process-target* ‹*i* ∈ $\{..{<}$*length ts*$\}$›)
next
show $(\lambda w.\ \lambda i{\in}\{..{<}$*length ts*$\}.$ *process gaussian-process* (*ts* ! *i*) *w*) ∈
space (*proc-source gaussian-process*) →
(Π_E *i*∈$\{..{<}$*length ts*$\}$. *space borel*)
by (*auto simp*: *PiE-iff split*: *if-splits*)
qed

lemma *gaussian-process-ordered-list-joint*:
assumes *sorted*: *sorted ts* **and** *distinct*: *distinct ts*
and *nonnegative*: ∀ *t*∈*set ts*. $0 \le t$
shows *distr* (*proc-source gaussian-process*)
(*PiM* $\{..{<}$*length ts*$\}$ (λ-. *borel*))
$(\lambda w.\ \lambda i{\in}\{..{<}$*length ts*$\}.$ *process gaussian-process* (*ts* ! *i*) *w*) =
gaussian-finite-law ts
proof −
let *?I* = $\{..{<}$*length ts*$\}$
let *?T* = *set ts*
let *?B* = λ-. *borel*
have *subset*: *?T* ⊆ $\{0..\}$ **using** *nonnegative* **by** *auto*
have *fin*: *finite ?T* **by** *simp*

```
  have restrict-base: (λw. restrict w ?T) ∈
    measurable (PiM ({0..} :: real set) ?B) (PiM ?T ?B)
    by (rule measurable-restrict-subset[OF subset])
  have restrict-meas: (λw. restrict w ?T) ∈
    measurable gaussian-path-law (PiM ?T ?B)
    using restrict-base
    by (simp only: measurable-cong-sets[OF sets-gaussian-path-law refl])
  have list-meas: list-coordinates ts ∈
    measurable (PiM ?T ?B) (PiM ?I ?B)
    by (rule measurable-list-coordinates)
  have proc: process gaussian-process (ts ! i) = (λw. w (ts ! i))
    if i < length ts for i
    by (rule gaussian-process-coordinate) (use nonnegative that in auto)
  have mapping: (λw. λi∈?I. process gaussian-process (ts ! i) w) =
    list-coordinates ts ∘ (λw. restrict w ?T)
    using proc by (auto simp: fun-eq-iff list-coordinates-def restrict-def)
  have distr (proc-source gaussian-process) (PiM ?I ?B)
      (λw. λi∈?I. process gaussian-process (ts ! i) w) =
    distr gaussian-path-law (PiM ?I ?B)
      (list-coordinates ts ∘ (λw. restrict w ?T))
    by (simp only: gaussian-process-source mapping)
  also have ... = distr (distr gaussian-path-law (PiM ?T ?B)
    (λw. restrict w ?T)) (PiM ?I ?B) (list-coordinates ts)
    by (rule distr-distr[OF list-meas restrict-meas, symmetric])
  also have ... = gaussian-finite-law ts
    using gaussian-limit-marginal[OF fin subset]
      gaussian-set-law-ordered-list[OF sorted distinct] by simp
  finally show ?thesis .
qed

lemma time-index-adjacent:
  assumes sorted: sorted ts and j: 0 < j j < length ts
    and jump: ts ! (j − 1) < ts ! j
  shows time-index (set ts) (ts ! j) =
    Suc (time-index (set ts) (ts ! (j − 1)))
proof −
  let ?T = set ts
  let ?u = ordered-times ?T
  let ?a = time-index ?T (ts ! (j − 1))
  let ?b = time-index ?T (ts ! j)
  have prev: ts ! (j − 1) ∈ ?T using j by simp
  have curr: ts ! j ∈ ?T using j by simp
  have ab: ?a < length ?u ?b < length ?u
    using time-index-unique(1)[of ?T ts ! (j − 1)]
      time-index-unique(1)[of ?T ts ! j] prev curr by auto
  have coords: ?u ! ?a = ts ! (j − 1) ?u ! ?b = ts ! j
    using time-index-unique(2)[of ?T ts ! (j − 1)]
      time-index-unique(2)[of ?T ts ! j] prev curr by auto
  have strict: sorted-wrt (<) ?u
```

```
    by (simp add: ordered-times-def strict-sorted-iff)
  have less: ?a < ?b
  proof (rule ccontr)
    assume ¬ ?a < ?b
    then have ?b ≤ ?a by simp
    then have ?u ! ?b ≤ ?u ! ?a
      using sorted-nth-mono[of ?u ?b ?a] ab strict
      by (simp add: strict-sorted-iff)
    with jump coords show False by simp
  qed
  have no-middle: ¬ Suc ?a < ?b
  proof
    assume mid: Suc ?a < ?b
    then have m: Suc ?a < length ?u using ab by linarith
    have bounds: ts ! (j − 1) < ?u ! Suc ?a
      ?u ! Suc ?a < ts ! j
      using sorted-wrt-nth-less[OF strict, of ?a Suc ?a]
        sorted-wrt-nth-less[OF strict, of Suc ?a ?b]
        m mid ab coords by auto
    have member: ?u ! Suc ?a ∈ set ts
      using nth-mem[OF m] ordered-times-finite(1)[of ?T] by simp
    then obtain k where k: k < length ts ts ! k = ?u ! Suc ?a
      by (auto simp: in-set-conv-nth)
    have k ≤ j − 1 ∨ j ≤ k by arith
    then show False
    proof
      assume k ≤ j − 1
      then have ts ! k ≤ ts ! (j − 1)
        by (rule sorted-nth-mono[OF sorted]) (use j in auto)
      with bounds k show False by simp
    next
      assume j ≤ k
      then have ts ! j ≤ ts ! k
        by (rule sorted-nth-mono[OF sorted]) (use k in auto)
      with bounds k show False by simp
    qed
  qed
  show ?thesis using less no-middle by arith
qed

lemma gaussian-process-ordered-increment-law:
  assumes sorted: sorted ts and distinct: distinct ts
    and nonnegative: ∀ t∈set ts. 0 ≤ t
  shows distr (proc-source gaussian-process) (increment-law ts)
    (λw. coordinate-increments (length ts)
      (λi∈{..<length ts}. process gaussian-process (ts ! i) w)) =
    increment-law ts
proof −
  let ?F = λw. λi∈{..<length ts}. process gaussian-process (ts ! i) w
```

have *meas-F*: *?F* ∈ *measurable* (*proc-source gaussian-process*)
(*PiM* {..<*length ts*} (λ-. *borel*))
by (*rule measurable-gaussian-process-ordered-coordinates*)
have *distr* (*proc-source gaussian-process*) (*increment-law ts*)
(*coordinate-increments* (*length ts*) ∘ *?F*) =
distr (*distr* (*proc-source gaussian-process*)
(*PiM* {..<*length ts*} (λ-. *borel*)) *?F*)
(*increment-law ts*) (*coordinate-increments* (*length ts*))
by (*rule distr-distr*[*OF measurable-coordinate-increments*[*of ts*] *meas-F*, *symmetric*])
also have ... = *increment-law ts*
by (*simp only*: *gaussian-process-ordered-list-joint*[*OF sorted distinct nonnegative*]
gaussian-finite-law-coordinate-increments)
finally show *?thesis* **by** (*simp only*: *comp-def*)
qed

lemma *gaussian-process-ordered-independent-increments*:
assumes *sorted*: *sorted ts* **and** *distinct*: *distinct ts*
and *nonnegative*: ∀ *t*∈*set ts*. *0* ≤ *t* **and** *nonempty*: *ts* ≠ []
shows *prob-space.indep-vars* (*proc-source gaussian-process*) (λ-. *borel*)
(λ*i w*. *coordinate-increments* (*length ts*)
(λ*k*∈{..<*length ts*}. *process gaussian-process* (*ts* ! *k*) *w*) *i*)
{..<*length ts*}
proof −
have *prob*: *prob-space* (*proc-source gaussian-process*)
by (*simp add*: *gaussian-process-source prob-space-gaussian-path-law*)
let *?I* = {..<*length ts*}
let *?F* = λ*w*. λ*i*∈*?I*. *process gaussian-process* (*ts* ! *i*) *w*
let *?D* = λ*w*. *coordinate-increments* (*length ts*) (*?F w*)
have *meas-F*: *?F* ∈ *measurable* (*proc-source gaussian-process*)
(*PiM ?I* (λ-. *borel*))
by (*rule measurable-gaussian-process-ordered-coordinates*)
have *meas-D*: *?D* ∈ *measurable* (*proc-source gaussian-process*) (*increment-law ts*)
using *measurable-compose*[*OF meas-F measurable-coordinate-increments*[*of ts*]]
by (*simp add*: *comp-def*)
have *component*: (λ*w* :: *nat* ⇒ *real*. *w i*) ∈ *borel-measurable* (*increment-law ts*)
if *i*: *i* ∈ *?I* **for** *i*
using *measurable-component-singleton*[*of i ?I* λ-. *borel* :: *real measure*] *i*
by (*simp only*: *measurable-cong-sets*[*OF sets-increment-law refl*])
have *rv*: (λ*w*. *?D w i*) ∈ *borel-measurable* (*proc-source gaussian-process*)
if *i*: *i* ∈ *?I* **for** *i*
using *measurable-compose*[*OF meas-D component*[*OF i*]]
by (*simp add*: *comp-def*)
have *law*: *distr* (*proc-source gaussian-process*) (*increment-law ts*) *?D* = *increment-law ts*
by (*rule gaussian-process-ordered-increment-law*[*OF sorted distinct nonnegative*])
have *joint*: *distr* (*proc-source gaussian-process*) (*PiM ?I* (λ-. *borel*)) *?D* =

```
    increment-law ts
  proof −
    have distr (proc-source gaussian-process) (PiM ?I (λ-. borel)) ?D =
      distr (proc-source gaussian-process) (increment-law ts) ?D
      by (rule distr-cong) auto
    then show ?thesis using law by simp
  qed
  have marg: distr (proc-source gaussian-process) borel (λw. ?D w i) =
    centered-normal (increment-variance ts i) if i: i ∈ ?I for i
  proof −
    have distr (proc-source gaussian-process) borel (λw. ?D w i) =
      distr (distr (proc-source gaussian-process) (increment-law ts) ?D)
        borel (λw. w i)
      using distr-distr[OF component[OF i] meas-D] by (simp add: comp-def)
    also have ... = centered-normal (increment-variance ts i)
    proof −
      have target: distr (increment-law ts) borel (λw. w i) =
        distr (increment-law ts) (centered-normal (increment-variance ts i))
          (λw. w i)
        by (rule distr-cong) simp-all
      show ?thesis using target increment-law-component[of i ts] i
        by (simp only: law) simp
    qed
    finally show ?thesis .
  qed
  have identity: (λw. λi∈?I. ?D w i) = ?D
    by (auto simp: fun-eq-iff coordinate-increments-def)
  have iff: prob-space.indep-vars (proc-source gaussian-process)
      (λ-. borel) (λi w. ?D w i) ?I ⟷
    distr (proc-source gaussian-process) (PiM ?I (λ-. borel))
      (λw. λi∈?I. ?D w i) =
    PiM ?I (λi. distr (proc-source gaussian-process) borel (λw. ?D w i))
    by (rule prob-space.indep-vars-iff-distr-eq-PiM'[OF prob])
      (use nonempty rv in auto)
  have product: PiM ?I (λi. distr (proc-source gaussian-process) borel (λw. ?D w
i)) =
    increment-law ts
    unfolding increment-law-def by (rule PiM-cong) (use marg in auto)
  show ?thesis using iff joint product by (simp only: identity)
qed

corollary gaussian-process-independent-successive-increments:
  assumes sorted: sorted ts and distinct: distinct ts
    and nonnegative: ∀ t∈set ts. 0 ≤ t and nonempty: ts ≠ []
  shows prob-space.indep-vars (proc-source gaussian-process) (λ-. borel)
    (λi w. if i = 0 then process gaussian-process (ts ! 0) w
      else process gaussian-process (ts ! i) w −
        process gaussian-process (ts ! (i − 1)) w) {..<length ts}
proof −
```

```
  let ?I = {..<length ts}
  let ?F = λw. λi∈?I. process gaussian-process (ts ! i) w
  have prob: prob-space (proc-source gaussian-process)
    by (simp add: gaussian-process-source prob-space-gaussian-path-law)
  have indep: prob-space.indep-vars (proc-source gaussian-process)
    (λ-. borel)
    (λi w. coordinate-increments (length ts) (?F w) i) ?I
    by (rule gaussian-process-ordered-independent-increments
      [OF sorted distinct nonnegative nonempty])
  have agree: (λw. coordinate-increments (length ts) (?F w) i) =
    (λw. if i = 0 then process gaussian-process (ts ! 0) w
      else process gaussian-process (ts ! i) w −
        process gaussian-process (ts ! (i − 1)) w)
    if i ∈ ?I for i
    using that by (auto simp: fun-eq-iff coordinate-increments-def)
  have same: prob-space.indep-vars (proc-source gaussian-process) (λ-. borel)
      (λi w. coordinate-increments (length ts) (?F w) i) ?I =
    prob-space.indep-vars (proc-source gaussian-process) (λ-. borel)
      (λi w. if i = 0 then process gaussian-process (ts ! 0) w
        else process gaussian-process (ts ! i) w −
          process gaussian-process (ts ! (i − 1)) w) ?I
    by (rule prob-space.indep-vars-cong[OF prob]) (use agree in auto)
  show ?thesis using indep same by simp
qed

corollary gaussian-process-jointly-independent-increments:
  assumes sorted: sorted ts and distinct: distinct ts
    and nonnegative: ∀t∈set ts. 0 ≤ t
    and nonempty: ts ≠ []
  shows prob-space.indep-vars (proc-source gaussian-process) (λ-. borel)
    (λi w. process gaussian-process (ts ! i) w −
      process gaussian-process (ts ! (i − 1)) w) {1..<length ts}
proof −
  have prob: prob-space (proc-source gaussian-process)
    by (simp add: gaussian-process-source prob-space-gaussian-path-law)
  have indep: prob-space.indep-vars (proc-source gaussian-process) (λ-. borel)
    (λi w. if i = 0 then process gaussian-process (ts ! 0) w
      else process gaussian-process (ts ! i) w −
        process gaussian-process (ts ! (i − 1)) w) {..<length ts}
    by (rule gaussian-process-independent-successive-increments
      [OF sorted distinct nonnegative nonempty])
  have restricted: prob-space.indep-vars (proc-source gaussian-process) (λ-. borel)
    (λi w. if i = 0 then process gaussian-process (ts ! 0) w
      else process gaussian-process (ts ! i) w −
        process gaussian-process (ts ! (i − 1)) w) {1..<length ts}
    by (rule prob-space.indep-vars-subset[OF prob indep]) auto
  have same: prob-space.indep-vars (proc-source gaussian-process) (λ-. borel)
    (λi w. if i = 0 then process gaussian-process (ts ! 0) w
      else process gaussian-process (ts ! i) w −
```

```
      process gaussian-process (ts ! (i − 1)) w) {1..<length ts} =
    prob-space.indep-vars (proc-source gaussian-process) (λ-. borel)
     (λi w. process gaussian-process (ts ! i) w −
      process gaussian-process (ts ! (i − 1)) w) {1..<length ts}
   by (rule prob-space.indep-vars-cong[OF prob]) auto
  show ?thesis using restricted same by simp
qed

theorem gaussian-process-indep-increments:
  indep-increments gaussian-process
proof (rule indep-incrementsI)
  fix ts :: real list
  assume times: set ts ⊆ proc-index gaussian-process
    and sorted: sorted ts and length: 2 ≤ length ts
  let ?T = set ts
  let ?u = ordered-times ?T
  let ?I = {1..<length ?u}
  let ?J = {1..<length ts}
  let ?X = λi w. process gaussian-process (?u ! i) w −
    process gaussian-process (?u ! (i − 1)) w
  let ?r = λj. time-index ?T (ts ! j)
  let ?K = λj. if ts ! (j − 1) = ts ! j then {} else {?r j}
  let ?Y = λj f :: nat ⇒ real. if ?K j = {} then 0 else f (?r j)
  have prob: prob-space (proc-source gaussian-process)
    by (simp add: gaussian-process-source prob-space-gaussian-path-law)
  have nonnegative: ∀ t∈?T. 0 ≤ t
    using times by (auto simp: gaussian-process-index)
  have fin: finite ?T by simp
  have u-sorted: sorted ?u and u-distinct: distinct ?u
    and u-set: set ?u = ?T
    using ordered-times-finite[OF fin] by auto
  have u-nonempty: ?u ≠ [] using u-set length by auto
  have indep: prob-space.indep-vars (proc-source gaussian-process)
    (λ-. borel) ?X ?I
    by (rule gaussian-process-jointly-independent-increments
      [OF u-sorted u-distinct - u-nonempty]) (use u-set nonnegative in auto)
  have jump: ts ! (j − 1) < ts ! j
    if j: j ∈ ?J and neq: ts ! (j − 1) ≠ ts ! j for j
    using sorted-nth-mono[OF sorted, of j − 1 j] j neq by auto
  have rank: ?r j = Suc (time-index ?T (ts ! (j − 1)))
    if j: j ∈ ?J and neq: ts ! (j − 1) ≠ ts ! j for j
    by (rule time-index-adjacent[OF sorted]) (use j jump[OF j neq] in auto)
  have sub: ?K j ⊆ ?I if j: j ∈ ?J for j
  proof (cases ts ! (j − 1) = ts ! j)
    case True then show ?thesis by simp
  next
    case False
    have member: ts ! j ∈ ?T using j by auto
    have bound: ?r j < length ?u
```

```
    by (rule time-index-unique(1)[OF fin member])
  show ?thesis using rank[OF j False] bound False by auto
qed
have ends: ts ! j ≠ ts ! k
  if j: j ∈ ?J and k: k ∈ ?J and neq: j ≠ k
    and jump-j: ts ! (j − 1) ≠ ts ! j
    and jump-k: ts ! (k − 1) ≠ ts ! k for j k
proof (cases j < k)
  case True
  have ts ! j ≤ ts ! (k − 1)
    by (rule sorted-nth-mono[OF sorted]) (use True k in auto)
  moreover have ts ! (k − 1) < ts ! k by (rule jump[OF k jump-k])
  ultimately show ?thesis by simp
next
  case False
  with neq have kj: k < j by simp
  have ts ! k ≤ ts ! (j − 1)
    by (rule sorted-nth-mono[OF sorted]) (use kj j in auto)
  moreover have ts ! (j − 1) < ts ! j by (rule jump[OF j jump-j])
  ultimately show ?thesis by simp
qed
have disjoint: disjoint-family-on ?K ?J
proof (unfold disjoint-family-on-def, intro ballI impI)
  fix j k assume j: j ∈ ?J and k: k ∈ ?J and neq: j ≠ k
  show ?K j ∩ ?K k = {}
  proof (cases ts ! (j − 1) = ts ! j ∨ ts ! (k − 1) = ts ! k)
    case True then show ?thesis by auto
  next
    case False
    then have diff: ts ! j ≠ ts ! k using ends[OF j k neq] by auto
    have members: ts ! j ∈ ?T ts ! k ∈ ?T using j k by auto
    have ranks: ?r j ≠ ?r k
      using time-index-unique(2)[OF fin members(1)]
        time-index-unique(2)[OF fin members(2)] diff by auto
    with False show ?thesis by auto
  qed
qed
have parts: prob-space.indep-vars (proc-source gaussian-process)
  (λj. PiM (?K j) (λ-. borel))
  (λj w. restrict (λi. ?X i w) (?K j)) ?J
  by (rule prob-space.indep-vars-restrict[OF prob indep sub disjoint])
have Y-meas: ?Y j ∈ borel-measurable (PiM (?K j) (λ-. borel))
  if j: j ∈ ?J for j
  by (cases ts ! (j − 1) = ts ! j)
     (auto intro: measurable-component-singleton)
have independent: prob-space.indep-vars (proc-source gaussian-process)
  (λ-. borel)
  (λj w. ?Y j (restrict (λi. ?X i w) (?K j))) ?J
  by (rule prob-space.indep-vars-compose2[OF prob parts Y-meas])
```

```
  have agree: (λw. ?Y j (restrict (λi. ?X i w) (?K j))) =
    (λw. process gaussian-process (ts ! j) w −
      process gaussian-process (ts ! (j − 1)) w)
    if j: j ∈ ?J for j
  proof (cases ts ! (j − 1) = ts ! j)
    case True then show ?thesis by simp
  next
    case False
    have prev: ts ! (j − 1) ∈ ?T and curr: ts ! j ∈ ?T
      using j by auto
    show ?thesis
      using time-index-unique(2)[OF fin prev]
        time-index-unique(2)[OF fin curr] rank[OF j False]
      by (simp add: False fun-eq-iff restrict-def)
  qed
  have eq: prob-space.indep-vars (proc-source gaussian-process) (λ-. borel)
    (λj w. ?Y j (restrict (λi. ?X i w) (?K j))) ?J =
    prob-space.indep-vars (proc-source gaussian-process) (λ-. borel)
      (λj w. process gaussian-process (ts ! j) w −
        process gaussian-process (ts ! (j − 1)) w) ?J
    by (rule prob-space.indep-vars-cong[OF prob]) (use agree in auto)
  show prob-space.indep-vars (proc-source gaussian-process)
      (λ-. proc-target gaussian-process)
      (λk v. process gaussian-process (ts ! k) v −
        process gaussian-process (ts ! (k − 1)) v) ?J
    using independent eq by (simp only: gaussian-process-target)
qed

lemma gaussian-process-increment-law:
  assumes s: 0 ≤ s and st: s ≤ t
  shows distr (proc-source gaussian-process) borel
    (λw. process gaussian-process t w − process gaussian-process s w) =
    centered-normal (t − s)
proof (cases s = t)
  case True
  have prob: prob-space (proc-source gaussian-process)
    by (simp add: gaussian-process-source prob-space-gaussian-path-law)
  have distr (proc-source gaussian-process) borel
    (λw. process gaussian-process t w − process gaussian-process s w) =
    distr (proc-source gaussian-process) borel (λ-. 0)
    using True by (intro distr-cong) auto
  also have ... = centered-normal (t − s)
    using True prob by (simp add: prob-space.distr-const)
  finally show ?thesis .
next
  case False
  with st have strict: s < t by simp
  let ?ts = [s,t]
  let ?I = {..<length ?ts}
```

```
  let ?F = λw. λi∈?I. process gaussian-process (?ts ! i) w
  let ?g = λw :: nat ⇒ real. w 1 − w 0
  have sorted: sorted ?ts using strict by simp
  have distinct: distinct ?ts using strict by simp
  have nonnegative: ∀ u∈set ?ts. 0 ≤ u using s strict by auto
  have F-meas: ?F ∈ measurable (proc-source gaussian-process)
    (PiM ?I (λ-. borel))
    by (rule measurable-gaussian-process-ordered-coordinates)
  have g-meas: ?g ∈ borel-measurable (PiM ?I (λ-. borel))
    by (intro borel-measurable-diff measurable-component-singleton) auto
  have distr (proc-source gaussian-process) borel (?g ∘ ?F) =
    distr (distr (proc-source gaussian-process) (PiM ?I (λ-. borel)) ?F)
      borel ?g
    by (rule distr-distr[OF g-meas F-meas, symmetric])
  also have ... = centered-normal (t − s)
    using gaussian-process-ordered-list-joint[OF sorted distinct nonnegative]
      gaussian-finite-law-increment[of 1 ?ts]
    by (simp add: increment-variance-def)
  finally show ?thesis by (simp add: comp-def)
qed

lemma gaussian-set-law-zero-component:
  distr (gaussian-set-law {0}) borel (λw. w 0) = return borel 0
proof −
  have ordered: ordered-times {0} = [0]
    by (simp add: ordered-times-def)
  have index: time-index {0} 0 = 0
    using time-index-unique(1)[of {0} 0] by (simp add: ordered)
  have coord: (λw :: real ⇒ real. w 0) ∈
    borel-measurable (PiM {0} (λ-. borel))
    by (rule measurable-component-singleton) simp
  have base: set-coordinates {0} ∈
    measurable (PiM {..<length (ordered-times {0})} (λ-. borel))
      (PiM {0} (λ-. borel))
    by (rule measurable-set-coordinates) simp
  have map: set-coordinates {0} ∈
    measurable (gaussian-finite-law [0]) (PiM {0} (λ-. borel))
    using base
    by (simp only: ordered measurable-cong-sets[OF sets-gaussian-finite-law refl])
  have comp: (λw. w 0) ∘ set-coordinates {0} = (λw. w 0)
    using index by (auto simp: fun-eq-iff set-coordinates-def)
  have distr (gaussian-set-law {0}) borel (λw. w 0) =
    distr (gaussian-finite-law [0]) borel ((λw. w 0) ∘ set-coordinates {0})
    unfolding gaussian-set-law-def
    using distr-distr[OF coord map] by (simp only: ordered)
  also have ... = return borel 0
  proof −
    have initial: distr (gaussian-finite-law [0]) borel (λw. w 0) =
      centered-normal 0
```

```
    using gaussian-finite-law-initial[of [0]] by simp
   show ?thesis using initial by (simp only: comp centered-normal-zero)
  qed
  finally show ?thesis .
qed

lemma gaussian-process-starts-at-zero:
  AE w in proc-source gaussian-process. process gaussian-process 0 w = 0
proof −
  have coord: (λw :: real ⇒ real. w 0) ∈
    borel-measurable (PiM {0} (λ-. borel))
    by (rule measurable-component-singleton) simp
  have subset-zero: ({0} :: real set) ⊆ {0..} by simp
  have restrict-base: (λ(w :: real ⇒ real). restrict w {0}) ∈
    measurable (PiM ({0..} :: real set) (λ-. borel :: real measure))
      (PiM {0} (λ-. borel))
    by (rule measurable-restrict-subset[OF subset-zero])
  have restrict: (λw. restrict w {0}) ∈
    measurable gaussian-path-law (PiM {0} (λ-. borel))
    using restrict-base
    by (simp only: measurable-cong-sets[OF sets-gaussian-path-law refl])
  have comp: (λw. w 0) ∘ (λw. restrict w {0}) = (λw. w 0)
    by (simp add: fun-eq-iff restrict-def)
  have law: distr gaussian-path-law borel (λw. w 0) = return borel 0
    using distr-distr[OF coord restrict] gaussian-limit-marginal[of {0}]
      gaussian-set-law-zero-component
    by (simp only: comp) simp
  have ae-zero: AE x in return borel (0::real). x = 0
    by (subst AE-return) (auto simp: Measurable.pred-def)
  have measurable-zero: (λw :: real ⇒ real. w 0) ∈
    borel-measurable gaussian-path-law
    by (rule gaussian-path-coordinate-measurable) simp
  have ae-law: AE x in distr gaussian-path-law borel (λw. w 0). x = 0
    using ae-zero by (simp only: law)
  have ae: AE w in gaussian-path-law. w 0 = 0
    by (rule AE-distrD[OF measurable-zero ae-law])
  have proc0: process gaussian-process 0 = (λw. w 0)
    by (rule gaussian-process-coordinate) simp
  show ?thesis using ae
    by (simp only: gaussian-process-source proc0)
qed
```

The projective-limit process starts at zero almost surely. Its independent normal increments are established above; the fourth-moment bound and continuous modification follow in the final theory.

```
end
theory Brownian-Motion
  imports Gaussian-Process
begin
```

4 Brownian motion

A Brownian motion is a real-valued process on nonnegative time, starting at zero almost surely, with jointly independent centred normal increments and almost surely continuous paths.

locale *brownian-motion* =
  fixes W :: (*real*, 'a, *real*) *stochastic-process*
  assumes *index*: *proc-index* $W = \{0..\}$
    and *init-0*: *AE w in proc-source W. process W 0 w = 0*
    and *indep-increments*: *indep-increments W*
    and *normal-increments*: $\bigwedge s\ t.\ 0 \le s \Longrightarrow s \le t \Longrightarrow$
      distr (*proc-source W*) *borel*
        ($\lambda w.$ *process W t w − process W s w*) = *centered-normal* ($t - s$)
    and *AE-continuous*: *AE w in proc-source W.*
      continuous-on $\{0..\}$ ($\lambda t.$ *process W t w*)

The existence argument starts from the canonical Gaussian process. Its fourth increment moment has quadratic order in the time difference, which is the estimate needed for Kolmogorov–Chentsov.

lemma *gaussian-process-fourth-increment-moment-ordered*:
  assumes *s*: $0 \le s$ **and** *st*: $s \le t$
  shows ($\int^+ w.$ *ennreal* (*dist* (*process gaussian-process t w*)
    (*process gaussian-process s w*) ^ 4) ∂*proc-source gaussian-process*) =
    ennreal (3 ∗ (*dist t s*)^2)
proof −
  have *meas*: ($\lambda w.$ *process gaussian-process t w −*
    process gaussian-process s w) ∈ *borel-measurable* (*proc-source gaussian-process*)
    using *stochastic-process-measurable*[*of gaussian-process t*]
      stochastic-process-measurable[*of gaussian-process s*]
    by (*simp add*: *gaussian-process-target*)
  have *law*: *distr* (*proc-source gaussian-process*) *borel*
    ($\lambda w.$ *process gaussian-process t w − process gaussian-process s w*) =
    centered-normal ($t - s$)
    by (*rule gaussian-process-increment-law*[*OF s st*])
  have *f-base*: (λx :: *real. ennreal* (x^4)) ∈ *borel-measurable borel*
    by *measurable*
  have *f-meas*: (λx :: *real. ennreal* (x^4)) ∈
    borel-measurable (*centered-normal* ($t - s$))
    using *f-base*
    by (*simp only*: *measurable-cong-sets*[*OF sets-centered-normal refl*])
  have *f-meas-distr*: (λx :: *real. ennreal* (x^4)) ∈
    borel-measurable (*distr* (*proc-source gaussian-process*) *borel*
      ($\lambda w.$ *process gaussian-process t w − process gaussian-process s w*))
    using *f-meas* **by** (*simp only*: *law*)
  have ($\int^+ w.$ *ennreal* ((*process gaussian-process t w −*
      process gaussian-process s w)^4) ∂*proc-source gaussian-process*) =
    ($\int^+ x.$ *ennreal* (x^4) ∂*centered-normal* ($t - s$))
    using *nn-integral-distr*[*OF meas f-meas-distr*] **by** (*simp only*: *law*)

```
  also have ... = ennreal (3 * (t - s)^2)
    by (rule centered-normal-fourth-nn-integral) (use st in simp)
  finally show ?thesis using st
    by (simp add: dist-real-def power-even-abs)
qed

lemma gaussian-process-fourth-increment-moment:
  assumes s: 0 ≤ s and t: 0 ≤ t
  shows (∫⁺ w. ennreal (dist (process gaussian-process t w)
    (process gaussian-process s w) ^ 4) ∂proc-source gaussian-process) =
    ennreal (3 * (dist t s)^2)
proof (cases s ≤ t)
  case True
 then show ?thesis by (rule gaussian-process-fourth-increment-moment-ordered[OF
s])
next
  case False
  then have st: t ≤ s by simp
  have (∫⁺ w. ennreal (dist (process gaussian-process t w)
      (process gaussian-process s w) ^ 4) ∂proc-source gaussian-process) =
    (∫⁺ w. ennreal (dist (process gaussian-process s w)
      (process gaussian-process t w) ^ 4) ∂proc-source gaussian-process)
    by (simp add: dist-commute)
  also have ... = ennreal (3 * (dist s t)^2)
    by (rule gaussian-process-fourth-increment-moment-ordered[OF t st])
  finally show ?thesis by (simp add: dist-commute)
qed

lemma gaussian-process-KC-bound:
  assumes 0 ≤ s 0 ≤ t
  shows (∫⁺ w. dist (process gaussian-process t w)
    (process gaussian-process s w) powr 4 ∂proc-source gaussian-process) ≤
    3 * dist t s powr 2
  using gaussian-process-fourth-increment-moment[OF assms]
  by (simp add: powr-realpow)
```

The fourth-moment estimate verifies the Kolmogorov–Chentsov hypotheses and therefore yields a modification whose paths are locally Holder continuous for every exponent below one quarter.

```
lemma gaussian-process-Kolmogorov-Chentsov:
  assumes 0 < γ γ < 1 / 4
  shows Kolmogorov-Chentsov gaussian-process 4 1 3 γ
proof (unfold-locales)
  fix s t :: real
  assume 0 ≤ s 0 ≤ t
  then show (∫⁺ w. dist (process gaussian-process t w)
      (process gaussian-process s w) powr 4 ∂proc-source gaussian-process) ≤
    3 * dist t s powr (1 + 1)
    using gaussian-process-KC-bound[OF ‹0 ≤ s› ‹0 ≤ t›] by simp
```

qed (*use assms* **in** ‹*auto simp*: *gaussian-process-index gaussian-process-target*›)

theorem *gaussian-process-continuous-modification*:
assumes $0 < \gamma\ \gamma < 1 / 4$
shows $\exists B.$ *modification gaussian-process B* $\wedge$
$(\forall w.$ *local-holder-on* γ $\{0..\}$ $(\lambda t.$ *process B t w*$))$
using *Kolmogorov-Chentsov.continuous-modification*
[*OF gaussian-process-Kolmogorov-Chentsov*[*OF assms*]] **.**

Modification preserves the finite-dimensional laws. In particular, the continuous version retains independent Gaussian increments and starts at zero almost surely, so it has all the Brownian motion properties.

theorem *brownian-continuous-modification*:
assumes *gamma*: $0 < \gamma\ \gamma < 1 / 4$
shows $\exists B.$ *modification gaussian-process B* $\wedge$
$(\forall w.$ *local-holder-on* γ $\{0..\}$ $(\lambda t.$ *process B t w*$))$ $\wedge$
indep-increments B $\wedge$
$(AE\ w$ *in proc-source B. process B 0 w* $= 0) \wedge$
$(\forall s\ t.\ 0 \leq s \wedge s \leq t \longrightarrow$
distr (*proc-source B*) *borel*
$(\lambda w.$ *process B t w* $-$ *process B s w*$)$ $=$ *centered-normal* $(t - s))$
proof $-$
obtain *B* **where** *mod*: *modification gaussian-process B*
and *holder*: $\forall w.$ *local-holder-on* γ $\{0..\}$ $(\lambda t.$ *process B t w*$)$
using *gaussian-process-continuous-modification*[*OF gamma*] **by** *blast*
have *compat*: *compatible gaussian-process B* **by** (*rule modificationD*(*1*)[*OF mod*])
have *source*: *proc-source B* $=$ *proc-source gaussian-process*
using *compatible-source*[*OF compat*] **by** *simp*
have *target*: *sets* (*proc-target B*) $=$ *sets borel*
using *compatible-target*[*OF compat*] **by** (*simp add*: *gaussian-process-target*)
have *indep*: *indep-increments B*
using *modification-indep-increments*[*OF mod target*] *gaussian-process-indep-increments*
by *blast*
have *zero*: $AE\ w$ *in proc-source B. process B 0 w* $= 0$
proof $-$
have *ae*: $AE\ w$ *in proc-source gaussian-process.*
process gaussian-process 0 w $=$ *process B 0 w*
by (*rule modificationD*(*2*)[*OF mod*]) (*simp add*: *gaussian-process-index*)
show *?thesis* **using** *ae gaussian-process-starts-at-zero*
by (*simp only*: *source*) *auto*
qed
have *inc*: *distr* (*proc-source B*) *borel*
$(\lambda w.$ *process B t w* $-$ *process B s w*$)$ $=$ *centered-normal* $(t - s)$
if *s*: $0 \leq s$ **and** *st*: $s \leq t$ **for** *s t*
proof $-$
have *s-index*: $s \in$ *proc-index gaussian-process*
using *s* **by** (*simp add*: *gaussian-process-index*)
have *t-index*: $t \in$ *proc-index gaussian-process*
using *s st* **by** (*simp add*: *gaussian-process-index*)

```
    have ae-s: AE w in proc-source gaussian-process.
      process gaussian-process s w = process B s w
      by (rule modificationD(2)[OF mod s-index])
    have ae-t: AE w in proc-source gaussian-process.
      process gaussian-process t w = process B t w
      by (rule modificationD(2)[OF mod t-index])
    have ae: AE w in proc-source gaussian-process.
      process B t w − process B s w =
        process gaussian-process t w − process gaussian-process s w
      using ae-s ae-t by eventually-elim simp
    have rv-gauss: (λw. process gaussian-process t w −
     process gaussian-process s w) ∈ borel-measurable (proc-source gaussian-process)
      using stochastic-process-measurable[of gaussian-process t]
        stochastic-process-measurable[of gaussian-process s]
      by (simp add: gaussian-process-target)
    have B-t: process B t ∈ borel-measurable (proc-source B)
      using stochastic-process-measurable[of B t]
      by (simp only: measurable-cong-sets[OF refl target])
    have B-s: process B s ∈ borel-measurable (proc-source B)
      using stochastic-process-measurable[of B s]
      by (simp only: measurable-cong-sets[OF refl target])
    have rv-B: (λw. process B t w − process B s w) ∈
      borel-measurable (proc-source B)
      using B-t B-s by simp
    have distr (proc-source B) borel
      (λw. process B t w − process B s w) =
      distr (proc-source gaussian-process) borel
        (λw. process gaussian-process t w − process gaussian-process s w)
      by (rule distr-cong-AE[OF source refl - rv-B rv-gauss])
        (use ae in ‹simp only: source›)
    also have ... = centered-normal (t − s)
      by (rule gaussian-process-increment-law[OF s st])
    finally show ?thesis .
  qed
  show ?thesis using mod holder indep zero inc by blast
qed
```

We now choose one such modification (with the concrete Holder exponent one eighth) and give it a permanent name. The subsequent interpretation packages the transferred properties and path continuity into the *brownian-motion* locale.

```
definition brownian-process :: (real, real ⇒ real, real) stochastic-process where
  brownian-process = (SOME B. modification gaussian-process B ∧
    (∀ w. local-holder-on (1/8) {0..} (λt. process B t w)) ∧
    indep-increments B ∧
    (AE w in proc-source B. process B 0 w = 0) ∧
    (∀ s t. 0 ≤ s ∧ s ≤ t ⟶
      distr (proc-source B) borel
        (λw. process B t w − process B s w) = centered-normal (t − s)))
```

lemma *brownian-process-properties*:
modification gaussian-process brownian-process ∧
(∀ *w*. *local-holder-on* (*1*/*8*) {*0*..} (λ*t*. *process brownian-process t w*)) ∧
indep-increments brownian-process ∧
(*AE w in proc-source brownian-process*. *process brownian-process 0 w* = *0*) ∧
(∀ *s t*. *0* ≤ *s* ∧ *s* ≤ *t* ⟶
distr (*proc-source brownian-process*) *borel*
(λ*w*. *process brownian-process t w* − *process brownian-process s w*) =
centered-normal (*t* − *s*))
proof −
have *existence*: ∃ *B* :: (*real*, *real* ⇒ *real*, *real*) *stochastic-process*.
modification gaussian-process B ∧
(∀ *w*. *local-holder-on* (*1*/*8*) {*0*..} (λ*t*. *process B t w*)) ∧
indep-increments B ∧
(*AE w in proc-source B*. *process B 0 w* = *0*) ∧
(∀ *s t*. *0* ≤ *s* ∧ *s* ≤ *t* ⟶
distr (*proc-source B*) *borel*
(λ*w*. *process B t w* − *process B s w*) = *centered-normal* (*t* − *s*))
using *brownian-continuous-modification*[*of 1*/*8*] **by** *simp*
show *?thesis* **unfolding** *brownian-process-def* **by** (*rule someI-ex*[*OF existence*])
qed

lemma *brownian-process-modification*:
modification gaussian-process brownian-process
using *brownian-process-properties* **by** *blast*

lemma *brownian-process-local-holder*:
local-holder-on (*1*/*8*) {*0*..} (λ*t*. *process brownian-process t w*)
using *brownian-process-properties* **by** *blast*

lemma *brownian-process-indep-increments*:
indep-increments brownian-process
using *brownian-process-properties* **by** *blast*

lemma *brownian-process-starts-at-zero*:
AE w in proc-source brownian-process. *process brownian-process 0 w* = *0*
using *brownian-process-properties* **by** *blast*

lemma *brownian-process-increment-law*:
assumes *0* ≤ *s s* ≤ *t*
shows *distr* (*proc-source brownian-process*) *borel*
(λ*w*. *process brownian-process t w* − *process brownian-process s w*) =
centered-normal (*t* − *s*)
using *brownian-process-properties assms* **by** *blast*

interpretation *constructed-brownian-motion*: *brownian-motion brownian-process*
proof
show *proc-index brownian-process* = {*0*..}

```
    using brownian-process-modification
    by (metis compatible-index gaussian-process-index modificationD(1))
  show AE w in proc-source brownian-process. process brownian-process 0 w = 0
    by (rule brownian-process-starts-at-zero)
  show indep-increments brownian-process
    by (rule brownian-process-indep-increments)
  show distr (proc-source brownian-process) borel
    (λw. process brownian-process t w − process brownian-process s w) =
    centered-normal (t − s) if 0 ≤ s s ≤ t for s t
    by (rule brownian-process-increment-law[OF that(1) that(2)])
  show AE w in proc-source brownian-process.
    continuous-on {0..} (λt. process brownian-process t w)
    using brownian-process-local-holder local-holder-imp-continuous by blast
qed

lemma modification-gaussian-joint-law:
  assumes mod: modification gaussian-process B
    and finite: finite T and subset: T ⊆ {0..}
  shows distr (proc-source B) (PiM T (λ-. borel))
    (λw. λt∈T. process B t w) = gaussian-set-law T
proof −
  have compat: compatible gaussian-process B
    by (rule modificationD(1)[OF mod])
  have source: proc-source B = proc-source gaussian-process
    using compatible-source[OF compat] by simp
  have target: sets (proc-target B) = sets borel
    using compatible-target[OF compat] by (simp add: gaussian-process-target)
  have measurable-coords: (λw. λt∈T. process X t w) ∈
    measurable (proc-source X) (PiM T (λ-. borel))
    if target-X: sets (proc-target X) = sets borel
    for X :: (real, real ⇒ real, real) stochastic-process
  proof (rule measurable-PiM-single′)
    fix t assume t: t ∈ T
    have coord: process X t ∈ borel-measurable (proc-source X)
      using stochastic-process-measurable[of X t]
      by (simp only: measurable-cong-sets[OF refl target-X])
    show (λw. (λt∈T. process X t w) t) ∈
      borel-measurable (proc-source X)
      using coord t by simp
  next
    show (λw. λt∈T. process X t w) ∈
      space (proc-source X) → (Π_E t∈T. space borel)
      by (auto simp: PiE-iff split: if-splits)
  qed
  have rv-B: (λw. λt∈T. process B t w) ∈
    measurable (proc-source B) (PiM T (λ-. borel))
    by (rule measurable-coords[OF target])
  have rv-gauss: (λw. λt∈T. process gaussian-process t w) ∈
    measurable (proc-source gaussian-process) (PiM T (λ-. borel))
```

```
    by (rule measurable-coords) (simp add: gaussian-process-target)
  have ae-coords: AE w in proc-source gaussian-process.
    ∀ t∈T. process B t w = process gaussian-process t w
  proof (rule AE-finite-allI[OF finite])
    fix t assume t ∈ T
    then have t ∈ proc-index gaussian-process
      using subset by (auto simp: gaussian-process-index)
    then have eq: AE w in proc-source gaussian-process.
      process gaussian-process t w = process B t w
      by (rule modificationD(2)[OF mod])
    show AE w in proc-source gaussian-process.
      process B t w = process gaussian-process t w
      using eq by eventually-elim simp
  qed
  have ae-maps: AE w in proc-source gaussian-process.
    (λt∈T. process B t w) =
    (λt∈T. process gaussian-process t w)
    using ae-coords by eventually-elim (auto simp: fun-eq-iff)
  have distr (proc-source B) (PiM T (λ-. borel))
      (λw. λt∈T. process B t w) =
    distr (proc-source gaussian-process) (PiM T (λ-. borel))
      (λw. λt∈T. process gaussian-process t w)
    by (rule distr-cong-AE[OF source refl - rv-B rv-gauss])
       (use ae-maps in ‹simp only: source›)
  also have ... = gaussian-set-law T
    using gaussian-process-joint-law[OF finite subset]
    by (simp only: gaussian-process-target)
  finally show ?thesis .
qed

lemma brownian-process-joint-law:
  assumes finite T T ⊆ {0..}
  shows distr (proc-source brownian-process) (PiM T (λ-. borel))
    (λw. λt∈T. process brownian-process t w) = gaussian-set-law T
 by (rule modification-gaussian-joint-law[OF brownian-process-modification assms])

lemma brownian-process-ordered-list-joint:
  assumes sorted: sorted ts and distinct: distinct ts
    and nonnegative: ∀ t∈set ts. 0 ≤ t
  shows distr (proc-source brownian-process)
    (PiM {..<length ts} (λ-. borel))
    (λw. λi∈{..<length ts}. process brownian-process (ts ! i) w) =
    gaussian-finite-law ts
proof −
  let ?I = {..<length ts}
  let ?F = λw. λi∈?I. process brownian-process (ts ! i) w
  let ?G = λw. λi∈?I. process gaussian-process (ts ! i) w
  have mod: modification gaussian-process brownian-process
    by (rule brownian-process-modification)
```

have *source*: *proc-source brownian-process = proc-source gaussian-process*
using *compatible-source*[*OF modificationD*(*1*)[*OF mod*]] **by** *simp*
have *target*: *sets* (*proc-target brownian-process*) = *sets borel*
using *compatible-target*[*OF modificationD*(*1*)[*OF mod*]]
by (*simp add*: *gaussian-process-target*)
have *rv*: *?F* ∈ *measurable* (*proc-source brownian-process*)
(*PiM ?I* (λ-. *borel*))
proof (*rule measurable-PiM-single′*)
fix *i* **assume** *i*: *i* ∈ *?I*
have *coord*: *process brownian-process* (*ts* ! *i*) ∈
borel-measurable (*proc-source brownian-process*)
using *stochastic-process-measurable*[*of brownian-process ts* ! *i*]
by (*simp only*: *measurable-cong-sets*[*OF refl target*])
show (λ*w*. *?F w i*) ∈ *borel-measurable* (*proc-source brownian-process*)
using *coord i* **by** *simp*
next
show *?F* ∈ *space* (*proc-source brownian-process*) →
(Π$_E$ *i*∈*?I*. *space borel*)
by (*auto simp*: *PiE-iff split*: *if-splits*)
qed
have *rv-gauss*: *?G* ∈ *measurable* (*proc-source gaussian-process*)
(*PiM ?I* (λ-. *borel*))
by (*rule measurable-gaussian-process-ordered-coordinates*)
have *ae-coords*: *AE w in proc-source gaussian-process*.
∀ *i*∈*?I*. *process brownian-process* (*ts* ! *i*) *w* =
process gaussian-process (*ts* ! *i*) *w*
proof (*rule AE-finite-allI*)
fix *i* **assume** *i*: *i* ∈ *?I*
have *t*: *ts* ! *i* ∈ *proc-index gaussian-process*
using *nonnegative i* **by** (*auto simp*: *gaussian-process-index*)
have *AE w in proc-source gaussian-process*.
process gaussian-process (*ts* ! *i*) *w* = *process brownian-process* (*ts* ! *i*) *w*
by (*rule modificationD*(*2*)[*OF mod t*])
then show *AE w in proc-source gaussian-process*.
process brownian-process (*ts* ! *i*) *w* = *process gaussian-process* (*ts* ! *i*) *w*
by *eventually-elim simp*
qed *simp*
have *ae*: *AE w in proc-source gaussian-process*. *?F w* = *?G w*
using *ae-coords* **by** *eventually-elim* (*auto simp*: *fun-eq-iff*)
have *distr* (*proc-source brownian-process*) (*PiM ?I* (λ-. *borel*)) *?F* =
distr (*proc-source gaussian-process*) (*PiM ?I* (λ-. *borel*)) *?G*
by (*rule distr-cong-AE*[*OF source refl - rv rv-gauss*])
(*use ae* **in** ‹*simp only*: *source*›)
also have ... = *gaussian-finite-law ts*
by (*rule gaussian-process-ordered-list-joint*[*OF sorted distinct nonnegative*])
finally show *?thesis* **.**
qed

lemma *brownian-process-ordered-list-scale*:

```
  assumes c: 0 < c and sorted: sorted ts and distinct: distinct ts
    and nonnegative: ∀ t∈set ts. 0 ≤ t
  shows distr (proc-source brownian-process)
    (PiM {..<length ts} (λ-. borel))
    (λw. λi∈{..<length ts}.
      process brownian-process (c * (ts ! i)) w / sqrt c) =
    gaussian-finite-law ts
proof −
  let ?I = {..<length ts}
  let ?us = map (λt. c * t) ts
  let ?F = compose ?I (λx. x / sqrt c)
  let ?G = λw. λi∈?I. process brownian-process (?us ! i) w
  have sorted-us: sorted ?us
    using sorted c by (simp add: sorted-map sorted-wrt-iff-nth-less
      sorted-iff-nth-mono-less mult-left-mono)
  have distinct-us: distinct ?us
    using distinct c by (simp add: distinct-map inj-on-def)
  have nonnegative-us: ∀ t∈set ?us. 0 ≤ t
    using nonnegative c by auto
  have target: sets (proc-target brownian-process) = sets borel
   using compatible-target[OF modificationD(1)[OF brownian-process-modification]]
    by (simp add: gaussian-process-target)
  have G-meas: ?G ∈ measurable (proc-source brownian-process)
    (PiM ?I (λ-. borel))
  proof (rule measurable-PiM-single′)
    fix i assume i: i ∈ ?I
    have coord: process brownian-process (?us ! i) ∈
      borel-measurable (proc-source brownian-process)
      using stochastic-process-measurable[of brownian-process ?us ! i]
      by (simp only: measurable-cong-sets[OF refl target])
    show (λw. ?G w i) ∈ borel-measurable (proc-source brownian-process)
      using coord i by simp
  next
    show ?G ∈ space (proc-source brownian-process) →
      (Π_E i∈?I. space borel)
      by (auto simp: PiE-iff split: if-splits)
  qed
  have F-meas: ?F ∈ measurable (PiM ?I (λ-. borel))
    (PiM ?I (λ-. borel))
  proof (rule measurable-PiM-single′)
    fix i assume i: i ∈ ?I
    have coord: (λw::nat ⇒ real. w i) ∈
      borel-measurable (PiM ?I (λ-. borel))
      by (rule measurable-component-singleton) (use i in auto)
    show (λw. ?F w i) ∈ borel-measurable (PiM ?I (λ-. borel))
      using coord i by (simp add: compose-def)
  next
    show ?F ∈ space (PiM ?I (λ-. borel)) →
      (Π_E i∈?I. space borel)
```

```
    by (auto simp: compose-def PiE-iff split: if-splits)
  qed
  have mapping: (λw. λi∈?I.
      process brownian-process (c * (ts ! i)) w / sqrt c) = ?F ∘ ?G
    by (auto simp: fun-eq-iff compose-def)
  have distr (proc-source brownian-process) (PiM ?I (λ-. borel))
      (?F ∘ ?G) =
    distr (distr (proc-source brownian-process) (PiM ?I (λ-. borel)) ?G)
      (PiM ?I (λ-. borel)) ?F
    by (rule distr-distr[OF F-meas G-meas, symmetric])
  also have ... = gaussian-finite-law ts
      using brownian-process-ordered-list-joint[OF sorted-us distinct-us nonnega-
tive-us]
      gaussian-finite-law-scale[OF c sorted nonnegative]
    by simp
  finally show ?thesis using mapping by simp
qed

theorem brownian-process-finite-scale-invariance:
  assumes c: 0 < c and finite: finite T and subset: T ⊆ {0..}
  shows distr (proc-source brownian-process) (PiM T (λ-. borel))
    (λw. λt∈T. process brownian-process (c * t) w / sqrt c) =
    gaussian-set-law T
proof −
  let ?ts = ordered-times T
  let ?I = {..<length ?ts}
  let ?F = λw. λi∈?I.
    process brownian-process (c * (?ts ! i)) w / sqrt c
  let ?H = λw. λt∈T.
    process brownian-process (c * t) w / sqrt c
  have sorted: sorted ?ts and distinct: distinct ?ts
    and times: set ?ts = T
    using ordered-times-finite[OF finite] by auto
  have nonnegative: ∀ t∈set ?ts. 0 ≤ t
    using times subset by auto
  have target: sets (proc-target brownian-process) = sets borel
   using compatible-target[OF modificationD(1)[OF brownian-process-modification]]
    by (simp add: gaussian-process-target)
  have F-meas: ?F ∈ measurable (proc-source brownian-process)
    (PiM ?I (λ-. borel))
  proof (rule measurable-PiM-single′)
    fix i assume i: i ∈ ?I
    have coord: process brownian-process (c * (?ts ! i)) ∈
      borel-measurable (proc-source brownian-process)
      using stochastic-process-measurable[of brownian-process c * (?ts ! i)]
      by (simp only: measurable-cong-sets[OF refl target])
    show (λw. ?F w i) ∈ borel-measurable (proc-source brownian-process)
      using coord i by simp
  next
```

```
    show ?F ∈ space (proc-source brownian-process) →
      (Π_E i∈?I. space borel)
      by (auto simp: PiE-iff split: if-splits)
  qed
  have set-meas: set-coordinates T ∈ measurable
    (PiM ?I (λ-. borel)) (PiM T (λ-. borel))
    by (rule measurable-set-coordinates[OF finite])
  have mapping: ?H = set-coordinates T ∘ ?F
   using time-index-unique[OF finite] by (auto simp: fun-eq-iff set-coordinates-def)
  have distr (proc-source brownian-process) (PiM T (λ-. borel)) ?H =
    distr (distr (proc-source brownian-process) (PiM ?I (λ-. borel)) ?F)
      (PiM T (λ-. borel)) (set-coordinates T)
    by (simp only: mapping distr-distr[OF set-meas F-meas, symmetric])
  also have ... = gaussian-set-law T
    using brownian-process-ordered-list-scale[OF c sorted distinct nonnegative]
    by (simp only: gaussian-set-law-def)
  finally show ?thesis .
qed

theorem brownian-process-scale-invariance:
  assumes 0 < c finite T T ⊆ {0..}
  shows distr (proc-source brownian-process) (PiM T (λ-. borel))
    (λw. λt∈T. process brownian-process (c * t) w / sqrt c) =
    distr (proc-source brownian-process) (PiM T (λ-. borel))
      (λw. λt∈T. process brownian-process t w)
  using brownian-process-finite-scale-invariance[OF assms]
    brownian-process-joint-law[OF assms(2,3)] by simp

theorem brownian-process-path-scale-invariance:
  assumes c: 0 < c
  shows distr (proc-source brownian-process)
    (PiM {0..} (λ-. borel :: real measure))
    (λw. λt∈{0..}. process brownian-process (c * t) w / sqrt c) =
    gaussian-path-law
proof −
  let ?I = {0..} :: real set
  let ?B = λ-. borel :: real measure
  let ?F = λw. λt∈?I. process brownian-process (c * t) w / sqrt c
  let ?P = distr (proc-source brownian-process) (PiM ?I ?B) ?F
  have target: sets (proc-target brownian-process) = sets borel
   using compatible-target[OF modificationD(1)[OF brownian-process-modification]]
    by (simp add: gaussian-process-target)
  have meas-F: ?F ∈ measurable (proc-source brownian-process) (PiM ?I ?B)
  proof (rule measurable-PiM-single′)
    fix t assume t: t ∈ ?I
    have coord: process brownian-process (c * t) ∈
      borel-measurable (proc-source brownian-process)
      using stochastic-process-measurable[of brownian-process c * t]
      by (simp only: measurable-cong-sets[OF refl target])
```

```
  show (λw. ?F w t) ∈ borel-measurable (proc-source brownian-process)
    using coord t by simp
next
  show ?F ∈ space (proc-source brownian-process) →
    (Π_E t∈?I. space borel)
    by (auto simp: PiE-iff split: if-splits)
qed
have prob-source: prob-space (proc-source brownian-process)
 using compatible-source[OF modificationD(1)[OF brownian-process-modification]]
    prob-space-gaussian-path-law by (simp add: gaussian-process-source)
have fin-P: finite-measure ?P
  using prob-space.prob-space-distr[OF prob-source meas-F]
  by (simp add: prob-space-def)
have sets-P: sets ?P = sets (PiM ?I ?B) by simp
have sets-Q: sets gaussian-path-law = sets (PiM ?I ?B)
  by (rule sets-gaussian-path-law)
have space-Q: space gaussian-path-law = space (PiM ?I ?B)
  by (rule sets-eq-imp-space-eq[OF sets-Q])
show ?thesis
proof (rule measure-eqI-PiM-infinite[OF sets-P sets-Q - fin-P])
  fix A J
  assume finite: finite J and subset: J ⊆ ?I
    and events: ⋀t. t ∈ J ⟹ A t ∈ sets (?B t)
  have restr-meas: (λf. restrict f J) ∈
    measurable (PiM ?I ?B) (PiM J ?B)
    by (rule measurable-restrict-subset[OF subset])
  have restr-P: (λf. restrict f J) ∈ measurable ?P (PiM J ?B)
    using restr-meas by (simp only: measurable-cong-sets[OF sets-P refl])
  have restr-Q: (λf. restrict f J) ∈
    measurable gaussian-path-law (PiM J ?B)
    using restr-meas by (simp only: measurable-cong-sets[OF sets-Q refl])
  have event: PiE J A ∈ sets (PiM J ?B)
    by (rule sets-PiM-I-finite[OF finite]) (use events in auto)
  have comp: (λf. restrict f J) ∘ ?F =
    (λw. λt∈J. process brownian-process (c * t) w / sqrt c)
    using subset by (auto simp: fun-eq-iff restrict-def)
  have law-P: distr ?P (PiM J ?B) (λf. restrict f J) = gaussian-set-law J
  proof −
    have distr ?P (PiM J ?B) (λf. restrict f J) =
      distr (proc-source brownian-process) (PiM J ?B)
        ((λf. restrict f J) ∘ ?F)
      by (rule distr-distr[OF restr-meas meas-F])
    also have ... = gaussian-set-law J
      using brownian-process-finite-scale-invariance[OF c finite subset]
      by (simp only: comp)
    finally show ?thesis .
  qed
  have law-Q: distr gaussian-path-law (PiM J ?B)
    (λf. restrict f J) = gaussian-set-law J
```

```
    by (rule gaussian-limit-marginal[OF finite subset])
  have emb-P: prod-emb ?I ?B J (PiE J A) =
    (λf. restrict f J) -‘ (PiE J A) ∩ space ?P
    by (simp add: prod-emb-def space-PiM)
  have emb-Q: prod-emb ?I ?B J (PiE J A) =
    (λf. restrict f J) -‘ (PiE J A) ∩ space gaussian-path-law
    by (simp add: prod-emb-def space-PiM space-Q)
  show emeasure ?P (prod-emb ?I ?B J (PiE J A)) =
    emeasure gaussian-path-law (prod-emb ?I ?B J (PiE J A))
    using emeasure-distr[OF restr-P event] emeasure-distr[OF restr-Q event]
      law-P law-Q emb-P emb-Q by simp
 qed
qed

corollary brownian-process-full-scale-invariance:
 assumes 0 < c
 shows distr (proc-source brownian-process)
   (PiM {0..} (λ-. borel :: real measure))
   (λw. λt∈{0..}. process brownian-process (c * t) w / sqrt c) =
   distr (proc-source brownian-process)
     (PiM {0..} (λ-. borel :: real measure))
     (λw. λt∈{0..}. process brownian-process t w)
 using brownian-process-path-scale-invariance[OF assms]
   brownian-process-path-scale-invariance[of 1] by simp
```

The named process has Gaussian joint laws, independent normal increments, locally Holder-continuous paths, and Brownian scale invariance as an equality of full path-space distributions.

```
end
theory Euclidean-Brownian-Motion
 imports Brownian-Motion
begin
```

5 Brownian motion in finite-dimensional Euclidean space

The construction uses one independent copy of the real Brownian motion for every coordinate. The dimension type is finite but otherwise arbitrary.

```
definition euclidean-brownian-source ::
   ('d::finite ⇒ real ⇒ real) measure where
 euclidean-brownian-source = PiM UNIV (λ-. proc-source brownian-process)

lemma prob-space-euclidean-brownian-source:
 prob-space (euclidean-brownian-source :: ('d::finite ⇒ real ⇒ real) measure)
 unfolding euclidean-brownian-source-def
 by (rule prob-space-PiM) (simp add: proc-source.prob-space-axioms)
```

```
interpretation euclidean-source: prob-space
  euclidean-brownian-source :: ('d::finite ⇒ real ⇒ real) measure
  by (rule prob-space-euclidean-brownian-source)

lemma borel-measurable-vec-lambda:
  fixes f :: 'a ⇒ 'd::finite ⇒ real
  assumes coord: ⋀i. (λx. f x i) ∈ borel-measurable M
  shows (λx. (χ i. f x i) :: real^'d) ∈ borel-measurable M
proof (rule borel-measurable-iff-halfspace-less[THEN iffD2])
  show ∀ b∈Basis. ∀ a. {x ∈ space M. (χ i. f x i) • b < a} ∈ sets M
  proof (intro ballI allI)
    fix b :: real^'d and a :: real
    have scalar: (λx. (χ i. f x i) • b) ∈ borel-measurable M
      unfolding inner-vec-def by (measurable; simp add: coord)
    show {x ∈ space M. (χ i. f x i) • b < a} ∈ sets M
      using scalar unfolding borel-measurable-iff-less by blast
  qed
qed

definition euclidean-brownian-path ::
    real ⇒ ('d::finite ⇒ real ⇒ real) ⇒ real^'d where
  euclidean-brownian-path t w = (χ i. process brownian-process t (w i))

lemma euclidean-brownian-path-measurable [measurable]:
  euclidean-brownian-path t ∈
    borel-measurable (euclidean-brownian-source ::
      ('d::finite ⇒ real ⇒ real) measure)
proof −
  have target: sets (proc-target brownian-process) = sets borel
   using compatible-target[OF modificationD(1)[OF brownian-process-modification]]
    by (simp add: gaussian-process-target)
  have process-meas: process brownian-process t ∈
      borel-measurable (proc-source brownian-process)
    using stochastic-process-measurable[of brownian-process t]
    by (simp only: measurable-cong-sets[OF refl target])
  have coord: (λw. process brownian-process t (w i)) ∈
      borel-measurable (PiM UNIV (λ-. proc-source brownian-process)) for i :: 'd
    by (rule measurable-PiM-component-rev) (simp-all add: process-meas)
  show ?thesis
    unfolding euclidean-brownian-path-def euclidean-brownian-source-def
    apply (rule borel-measurable-iff-halfspace-less[THEN iffD2])
    apply (intro ballI allI)
    subgoal for b a
    proof −
      have scalar: (λw. (χ i. process brownian-process t (w i)) • b) ∈
          borel-measurable (PiM UNIV (λ-. proc-source brownian-process))
        unfolding inner-vec-def by (measurable; simp add: coord)
      show {w ∈ space (PiM UNIV (λ-. proc-source brownian-process)).
          (χ i. process brownian-process t (w i)) • b < a} ∈
```

```
      sets (PiM UNIV (λ-. proc-source brownian-process))
    using scalar unfolding borel-measurable-iff-less by blast
  qed
  done
qed

definition euclidean-brownian-process ::
  (real, 'd::finite ⇒ real ⇒ real, real^'d) stochastic-process where
 euclidean-brownian-process =
  euclidean-source.process-of borel {0..} euclidean-brownian-path 0

lemma euclidean-brownian-process-index [simp]:
 proc-index (euclidean-brownian-process ::
  (real, 'd::finite ⇒ real ⇒ real, real^'d) stochastic-process) = {0..}
 unfolding euclidean-brownian-process-def by simp

lemma euclidean-brownian-process-source [simp]:
 proc-source (euclidean-brownian-process ::
  (real, 'd::finite ⇒ real ⇒ real, real^'d) stochastic-process) =
  euclidean-brownian-source
 unfolding euclidean-brownian-process-def
by (rule euclidean-source.source-process-of) (auto simp: euclidean-brownian-path-measurable)

lemma euclidean-brownian-process-target [simp]:
 proc-target (euclidean-brownian-process ::
  (real, 'd::finite ⇒ real ⇒ real, real^'d) stochastic-process) = borel
 unfolding euclidean-brownian-process-def
by (rule euclidean-source.target-process-of) (auto simp: euclidean-brownian-path-measurable)

lemma euclidean-brownian-process-apply [simp]:
 assumes 0 ≤ t
 shows process (euclidean-brownian-process ::
    (real, 'd::finite ⇒ real ⇒ real, real^'d) stochastic-process) t w =
  (χ i. process brownian-process t (w i))
proof −
 have eq: process (euclidean-brownian-process ::
    (real, 'd ⇒ real ⇒ real, real^'d) stochastic-process) t =
    euclidean-brownian-path t
  unfolding euclidean-brownian-process-def
  by (rule euclidean-source.process-of-apply) (use assms in
      ‹auto simp: euclidean-brownian-path-measurable›)
 show ?thesis
  using fun-cong[OF eq, of w]
  unfolding euclidean-brownian-path-def by simp
qed
```

5.1 Centred Euclidean normal distributions

```
definition euclidean-centered-normal ::
```

```
    real ⇒ (real^'d::finite) measure where
  euclidean-centered-normal v =
    distr (PiM UNIV (λ-::'d. centered-normal v)) borel (λx. χ i. x i)

lemma prob-space-euclidean-centered-normal:
  assumes 0 ≤ v
  shows prob-space (euclidean-centered-normal v :: (real^'d::finite) measure)
  unfolding euclidean-centered-normal-def
  apply (rule prob-space.prob-space-distr)
   apply (rule prob-space-PiM)
   apply (simp add: prob-space-centered-normal)
  apply (rule borel-measurable-vec-lambda)
  apply (rule measurable-PiM-component-rev)
   apply simp
  apply (rule measurable-ident-sets)
  by simp

lemma sets-euclidean-centered-normal [simp]:
  sets (euclidean-centered-normal v :: (real^'d::finite) measure) = sets borel
  unfolding euclidean-centered-normal-def by simp

lemma euclidean-centered-normal-coordinate:
  assumes 0 ≤ v
  shows distr (euclidean-centered-normal v :: (real^'d::finite) measure) borel
      (λx. x $ i) = centered-normal v
proof −
  let ?M = PiM UNIV (λ-::'d. centered-normal v)
  have ps: prob-space ?M
    by (rule prob-space-PiM) (simp add: prob-space-centered-normal)
  have comp': distr ?M (centered-normal v) (λx. x i) = centered-normal v
    by (rule distr-PiM-component) (use assms in
        ‹auto simp: prob-space-centered-normal›)
  have target-eq: distr ?M borel (λx. x i) =
      distr ?M (centered-normal v) (λx. x i)
    by (rule distr-cong) auto
  have comp: distr ?M borel (λx. x i) = centered-normal v
    using target-eq comp' by simp
  have Vmeas: (λx. χ i. x i :: real^'d) ∈ borel-measurable ?M
    apply (rule borel-measurable-vec-lambda)
    apply (rule measurable-PiM-component-rev)
     apply simp
    apply (rule measurable-ident-sets)
    by simp
  have distr (euclidean-centered-normal v :: (real^'d) measure) borel (λx. x $ i) =
      distr ?M borel ((λx. x $ i) ∘ (λx. χ i. x i))
    unfolding euclidean-centered-normal-def
    by (rule distr-distr) (measurable, rule Vmeas)
  also have ... = centered-normal v
    using comp by (simp add: comp-def)
```

```
  finally show ?thesis .
qed

lemma euclidean-centered-normal-independent-coordinates:
  assumes 0 ≤ v
  shows prob-space.indep-vars (euclidean-centered-normal v ::
      (real^'d::finite) measure) (λ-. borel) (λi x. x $ i) UNIV
proof −
  let ?N = PiM UNIV (λ-::'d. centered-normal v)
  let ?V = λx. χ i. x i :: real^'d
  have psN: prob-space ?N
    by (rule prob-space-PiM) (simp add: prob-space-centered-normal)
  have psE: prob-space (euclidean-centered-normal v :: (real^'d) measure)
    by (rule prob-space-euclidean-centered-normal[OF assms])
  have Vmeas: ?V ∈ borel-measurable ?N
    apply (rule borel-measurable-vec-lambda)
    apply (rule measurable-PiM-component-rev)
     apply simp
    apply (rule measurable-ident-sets)
    by simp
  have rv: (λx. restrict (λi. x $ i) UNIV) ∈
      measurable (euclidean-centered-normal v :: (real^'d) measure)
        (PiM UNIV (λ-::'d. borel))
    apply measurable
     by (simp only: measurable-cong-sets[OF sets-euclidean-centered-normal refl];
measurable)
  have joint: distr (euclidean-centered-normal v :: (real^'d) measure)
      (PiM UNIV (λ-::'d. borel))
      (λx. restrict (λi. x $ i) UNIV) = ?N
  proof −
    have distr (euclidean-centered-normal v :: (real^'d) measure)
        (PiM UNIV (λ-::'d. borel))
        (λx. restrict (λi. x $ i) UNIV) =
      distr ?N (PiM UNIV (λ-::'d. borel))
        ((λx. restrict (λi. x $ i) UNIV) ∘ ?V)
      unfolding euclidean-centered-normal-def
      by (rule distr-distr) (measurable, rule Vmeas)
    also have ... = distr ?N ?N id
    proof (rule distr-cong)
      show ?N = ?N by simp
      show sets (PiM UNIV (λ-::'d. borel)) = sets ?N
        by (rule sets-PiM-cong) auto
      show ⋀x. x ∈ space ?N ⟹
        ((λx. restrict (λi. x $ i) UNIV) ∘ ?V) x = id x
        by (auto simp: space-PiM fun-eq-iff)
    qed
    also have ... = ?N
      by (simp only: id-def distr-id)
    finally show ?thesis .
```

```
  qed
  show ?thesis
  proof (rule prob-space.indep-vars-iff-distr-eq-PiM'[OF psE, THEN iffD2])
    show UNIV ≠ ({}::'d set) by simp
    show ⋀i. i ∈ UNIV ⟹
      (λx::real^'d. x $ i) ∈ borel-measurable (euclidean-centered-normal v)
      by (simp only: measurable-cong-sets[OF sets-euclidean-centered-normal refl];
measurable)
    show distr (euclidean-centered-normal v :: (real^'d) measure)
      (PiM UNIV (λ-::'d. borel))
      (λx. restrict (λi. x $ i) UNIV) =
      PiM UNIV (λi. distr (euclidean-centered-normal v) borel
        (λx. x $ i))
      unfolding joint
       by (rule PiM-cong) (auto simp: euclidean-centered-normal-coordinate[OF
assms])
  qed
qed

lemma euclidean-centered-normal-zero:
  euclidean-centered-normal 0 = return borel (0 :: real^'d::finite)
  unfolding euclidean-centered-normal-def centered-normal-zero
  apply (subst PiM-return[where a=λ-. 0])
    apply simp
   apply measurable
  apply (subst distr-return)
   apply (rule borel-measurable-vec-lambda)
   apply measurable
   apply (auto simp: space-PiM)[1]
  apply (rule arg-cong[where f=return borel])
  by (simp add: vec-eq-iff)
```

5.2 Independent families on a finite product

```
lemma indep-vars-PiM-finite-family:
  fixes I :: 'i set and J :: 'j set
    and M :: 'i ⇒ 'a measure and N :: 'i ⇒ 'j ⇒ 'b measure
    and X :: 'i ⇒ 'j ⇒ 'a ⇒ 'b
  assumes finite-I: finite I and finite-J: finite J
    and nonempty-I: I ≠ {} and nonempty-J: J ≠ {}
    and ps-M: ⋀i. i ∈ I ⟹ prob-space (M i)
    and indep: ⋀i. i ∈ I ⟹
      prob-space.indep-vars (M i) (N i) (X i) J
  shows prob-space.indep-vars (PiM I M) (λ(i,j). N i j)
    (λ(i,j) w. X i j (w i)) (I × J)
proof −
  let ?S = PiM I M
  let ?K = I × J
  let ?N = λ(i,j). N i j
```

```
  let ?Q = λp. if p ∈ ?K then
    (case p of (i,j) ⇒ distr (M i) (N i j) (X i j))
    else return (count-space UNIV) undefined
  let ?Y = λ(i,j) w. X i j (w i)
  have psS: prob-space ?S
    by (rule prob-space-PiM) (simp add: ps-M)
  have meas-X: X i j ∈ measurable (M i) (N i j)
    if i ∈ I j ∈ J for i j
    using indep[OF that(1)] that(2)
      prob-space.indep-vars-def2[OF ps-M[OF that(1)]] by blast
  have psQ: ⋀p. prob-space (?Q p)
  proof −
    fix p
    show prob-space (?Q p)
      by (cases p) (auto intro!: prob-space.prob-space-distr prob-space-return ps-M
meas-X)
  qed
  have meas: ⋀p. p ∈ ?K ⟹ ?Y p ∈ measurable ?S (?N p)
    using meas-X by (auto intro!: measurable-PiM-component-rev)
  have sets-Q: sets (?Q (i,j)) = sets (N i j)
    if i ∈ I j ∈ J for i j
    using meas-X[OF that] by (simp add: that)
  have space-Q: space (?Q (i,j)) = space (N i j)
    if i ∈ I j ∈ J for i j
    by (rule sets-eq-imp-space-eq[OF sets-Q[OF that]])
  have joint: distr ?S (PiM ?K ?N)
      (λw. restrict (λp. ?Y p w) ?K) = PiM ?K ?Q
  proof −
    interpret K: product-prob-space ?Q ?K
      by (rule product-prob-spaceI) (rule psQ)
    show ?thesis
    proof (rule K.PiM-eq)
      show sets (distr ?S (PiM ?K ?N)
          (λw. restrict (λp. ?Y p w) ?K)) = sets (PiM ?K ?Q)
      proof −
        have sets (PiM ?K ?N) = sets (PiM ?K ?Q)
          by (rule sets-PiM-cong) (auto simp: meas-X)
        then show ?thesis by simp
      qed
      fix L F
      assume L: finite L L ⊆ ?K
        and F: ⋀p. p ∈ L ⟹ F p ∈ sets (?Q p)
      let ?L = λi. {j ∈ J. (i,j) ∈ L}
      let ?F = λi. prod-emb J (λj. ?Q (i,j)) (?L i)
        (PiE (?L i) (λj. F (i,j)))
      have F-eq: ?F i = prod-emb J (N i) (?L i)
          (PiE (?L i) (λj. F (i,j))) if i ∈ I for i
        unfolding prod-emb-def
        apply (rule arg-cong2[where f=(∩)])
```

```
  apply simp
 apply (rule set-eqI)
 using space-Q[OF that] by (auto simp: PiE-iff)
have FL: ?F i ∈ sets (PiM J (N i)) if iI: i ∈ I for i
 unfolding F-eq[OF iI]
proof (rule sets-PiM-I)
 show finite (?L i) using finite-J by auto
 show ?L i ⊆ J using L F sets-Q iI by auto
 show ∀ j∈?L i. F (i,j) ∈ sets (N i j)
 proof (intro ballI)
  fix j
  assume jL: j ∈ ?L i
  have jJ: j ∈ J using jL by simp
  have F (i,j) ∈ sets (?Q (i,j))
   by (rule F) (use jL in auto)
  then show F (i,j) ∈ sets (N i j)
   using sets-Q[OF iI jJ] by simp
 qed
qed
have event-eq: (λw. restrict (λp. ?Y p w) ?K) -‘
   prod-emb ?K ?Q L (PiE L F) ∩ space ?S =
 prod-emb I M I (PiE I (λi.
   (λx. restrict (λj. X i j x) J) -‘ ?F i ∩ space (M i)))
 apply (rule set-eqI)
 using L apply (auto simp: prod-emb-iff PiE-iff restrict-def
   space-PiM extensional-def)
 done
have F-N: F p ∈ sets (?N p) if p ∈ L for p
proof -
 obtain i j where p: p = (i,j) by (cases p)
 have iI: i ∈ I and jJ: j ∈ J
   using L that p by auto
 have Fij: F (i,j) ∈ sets (?Q (i,j))
   by (rule F) (use that p in simp)
 show ?thesis
   unfolding p using Fij sets-Q[OF iI jJ] by simp
qed
have emb-eq: prod-emb ?K ?Q L (PiE L F) =
   prod-emb ?K ?N L (PiE L F)
 unfolding prod-emb-def
 apply (rule arg-cong2[where f=(∩)])
  apply simp
 apply (rule set-eqI)
 using space-Q by (auto simp: PiE-iff)
have emb-sets: prod-emb ?K ?Q L (PiE L F) ∈ sets (PiM ?K ?N)
 unfolding emb-eq
 apply (rule measurable-prod-emb[OF L(2)])
 apply (rule sets-PiM-I-finite[OF L(1)])
 using F-N by auto
```

have *factor*: *emeasure* (*M i*)
  ((λ*x*. *restrict* (λ*j*. *X i j x*) *J*) −‘ *?F i* ∩ *space* (*M i*)) =
  (∏ *j*∈*?L i*. *emeasure* (*?Q* (*i*,*j*)) (*F* (*i*,*j*)))
 if *i* ∈ *I* **for** *i*
proof −
 have *d*: *distr* (*M i*) (*PiM J* (*N i*))
    (λ*x*. *restrict* (λ*j*. *X i j x*) *J*) =
    PiM J (λ*j*. *distr* (*M i*) (*N i j*) (*X i j*))
 proof (*rule prob-space.indep-vars-iff-distr-eq-PiM′*
    [*OF ps-M*[*OF that*] *nonempty-J*, *THEN iffD1*])
  show ⋀*j*. *j* ∈ *J* ⟹ *X i j* ∈ *measurable* (*M i*) (*N i j*)
   by (*rule meas-X*[*OF that*])
  show *prob-space.indep-vars* (*M i*) (*N i*) (*X i*) *J*
   by (*rule indep*[*OF that*])
 qed
 have *m*: (λ*x*. *restrict* (λ*j*. *X i j x*) *J*) ∈
    measurable (*M i*) (*PiM J* (*N i*))
  using *meas-X*[*OF that*] **by** *measurable*
 have *product-eq*: *PiM J* (λ*j*. *?Q* (*i*,*j*)) =
    PiM J (λ*j*. *distr* (*M i*) (*N i j*) (*X i j*))
  by (*rule PiM-cong*) (*auto simp*: *that*)
 have *emeasure* (*M i*)
    ((λ*x*. *restrict* (λ*j*. *X i j x*) *J*) −‘ *?F i* ∩ *space* (*M i*)) =
    emeasure (*distr* (*M i*) (*PiM J* (*N i*))
     (λ*x*. *restrict* (λ*j*. *X i j x*) *J*)) (*?F i*)
  using *FL*[*OF that*] *m* **by** (*subst emeasure-distr*) *auto*
 also have ... = *emeasure* (*PiM J*
    (λ*j*. *distr* (*M i*) (*N i j*) (*X i j*))) (*?F i*)
  by (*simp only*: *d*)
 also have ... = *emeasure* (*PiM J* (λ*j*. *?Q* (*i*,*j*))) (*?F i*)
  using *arg-cong*[*OF product-eq*, *of* λ*R*. *emeasure R* (*?F i*)] **by** *simp*
 finally have *emeasure* (*M i*)
    ((λ*x*. *restrict* (λ*j*. *X i j x*) *J*) −‘ *?F i* ∩ *space* (*M i*)) =
    emeasure (*PiM J* (λ*j*. *?Q* (*i*,*j*))) (*?F i*) **.**
 also have ... = (∏ *j*∈*?L i*. *emeasure* (*?Q* (*i*,*j*)) (*F* (*i*,*j*)))
  apply (*rule emeasure-PiM-emb*)
   apply (*rule psQ*)
  apply (*use finite-J* **in** *auto*)
  using *F sets-Q that* **by** *blast*
 finally show *?thesis* **.**
qed
have *event-sets*: (λ*x*. *restrict* (λ*j*. *X i j x*) *J*) −‘ *?F i* ∩
  space (*M i*) ∈ *sets* (*M i*) **if** *iI*: *i* ∈ *I* **for** *i*
proof −
 have *m*: (λ*x*. *restrict* (λ*j*. *X i j x*) *J*) ∈
    measurable (*M i*) (*PiM J* (*N i*))
  using *meas-X*[*OF iI*] **by** *measurable*
 show *?thesis* **by** (*rule measurable-sets*[*OF m FL*[*OF iI*]])
qed

```
    have emeasure (distr ?S (PiM ?K ?N)
        (λw. restrict (λp. ?Y p w) ?K))
        (prod-emb ?K ?Q L (PiE L F)) =
      emeasure ?S (prod-emb I M I
        (PiE I (λi. (λx. restrict (λj. X i j x) J) -‘ ?F i ∩
          space (M i))))
       using L F meas event-eq emb-sets by (subst emeasure-distr) (measurable,
auto)
    also have ... = (∏ i∈I. emeasure (M i)
        ((λx. restrict (λj. X i j x) J) -‘ ?F i ∩ space (M i)))
      apply (rule emeasure-PiM-emb)
       apply (rule ps-M, assumption)
      apply simp
      apply (rule finite-I)
      by (rule event-sets, assumption)
    also have ... = (∏ i∈I. ∏ j∈?L i.
        emeasure (?Q (i,j)) (F (i,j)))
      apply (rule prod.cong)
       apply simp
      by (rule factor) assumption
    also have ... = (∏ p∈L. emeasure (?Q p) (F p))
      using finite-I finite-J L
      by (subst prod.Sigma) (auto intro!: prod.cong)
    finally show emeasure (distr ?S (PiM ?K ?N)
        (λw. restrict (λp. ?Y p w) ?K))
        (prod-emb ?K ?Q L (PiE L F)) =
      (∏ p∈L. emeasure (?Q p) (F p)) .
  qed
 qed
 have marginal: distr ?S (N i j) (λw. X i j (w i)) =
     distr (M i) (N i j) (X i j) if iI: i ∈ I and jJ: j ∈ J
   for i j
 proof -
   have xm: X i j ∈ measurable (M i) (N i j)
     by (rule meas-X[OF iI jJ])
   have cm: (λw. w i) ∈ measurable ?S (M i)
     by (rule measurable-PiM-component-rev[OF iI]) measurable
   have component: distr ?S (M i) (λw. w i) = M i
     by (rule distr-PiM-component[OF ps-M iI])
   have distr ?S (N i j) (λw. X i j (w i)) =
       distr (distr ?S (M i) (λw. w i)) (N i j) (X i j)
     using distr-distr[OF xm cm] by (simp add: comp-def)
   also have ... = distr (M i) (N i j) (X i j)
     by (simp only: component)
   finally show ?thesis .
 qed
 show ?thesis
 proof (rule prob-space.indep-vars-iff-distr-eq-PiM'
     [OF psS, THEN iffD2])
```

show $I \times J \neq \{\}$ **using** *nonempty-I nonempty-J* **by** *auto*
show $\bigwedge p.\ p \in I \times J \Longrightarrow ?Y\ p \in measurable\ ?S\ (?N\ p)$
by (*rule meas*)
show *distr ?S* (*PiM ?K ?N*)
(λ*w*. *restrict* (λ*p*. *?Y p w*) *?K*) =
PiM ?K (λ*p*. *distr ?S* (*?N p*) (*?Y p*))
unfolding *joint*
apply (*rule PiM-cong*)
apply *simp*
using *marginal* **by** *auto*
qed
qed

5.3 The Brownian-motion specification

locale *euclidean-brownian-motion* =
fixes *W* :: (*real*, *'a*, *real^'d*::*finite*) *stochastic-process*
assumes *index*: *proc-index W* = {*0*..}
and *init-0*: *AE w in proc-source W*. *process W 0 w* = *0*
and *indep-increments*: *indep-increments W*
and *normal-increments*: $\bigwedge s\ t.\ 0 \leq s \Longrightarrow s \leq t \Longrightarrow$
distr (*proc-source W*) *borel*
(λ*w*. *process W t w* − *process W s w*) =
euclidean-centered-normal (*t* − *s*)
and *AE-continuous*: *AE w in proc-source W*.
continuous-on {*0*..} (λ*t*. *process W t w*)

5.4 Basic Brownian properties

lemma *euclidean-brownian-process-starts-at-zero*:
AE w in proc-source (*euclidean-brownian-process* ::
(*real*, *'d*::*finite* ⇒ *real* ⇒ *real*, *real^'d*) *stochastic-process*).
process euclidean-brownian-process 0 w = *0*
proof −
have *coordinate*: *AE w in* (*euclidean-brownian-source* ::
(*'d* ⇒ *real* ⇒ *real*) *measure*).
process brownian-process 0 (*w i*) = *0* **for** *i*
unfolding *euclidean-brownian-source-def*
by (*rule AE-PiM-component*) (*auto simp*: *proc-source.prob-space-axioms*
brownian-process-starts-at-zero)
have *AE w in* (*euclidean-brownian-source* ::
(*'d* ⇒ *real* ⇒ *real*) *measure*).
∀ *i*. *process brownian-process 0* (*w i*) = *0*
apply (*simp only*: *AE-all-countable*)
by (*intro allI*, *rule coordinate*)
show *?thesis*
apply (*simp only*: *euclidean-brownian-process-source*)
using ‹*AE w in euclidean-brownian-source*.
∀ *i*. *process brownian-process 0* (*w i*) = *0*›
by *eventually-elim* (*simp add*: *vec-eq-iff*)

qed

lemma *euclidean-brownian-increment-product-law*:
fixes *s t* :: *real*
assumes *s*: $0 \leq s$ **and** *st*: $s \leq t$
defines $X \equiv \lambda(\text{-}::'d::finite)\ p.$
process brownian-process t p − process brownian-process s p
shows *distr* (*euclidean-brownian-source* ::
($'d \Rightarrow real \Rightarrow real$) *measure*)
(*PiM UNIV* ($\lambda\text{-}::'d.$ *centered-normal* $(t - s)$))
($\lambda w.$ *restrict* ($\lambda i.$ *X i* (*w i*)) *UNIV*) =
PiM UNIV ($\lambda\text{-}::'d.$ *centered-normal* $(t - s)$)
proof −
let *?S* = $\lambda\text{-}::'d.$ *proc-source brownian-process*
let *?N* = $\lambda\text{-}::'d.$ *centered-normal* $(t - s)$
have *source*: *product-prob-space ?S*
by (*rule product-prob-spaceI*) (*simp add*: *proc-source.prob-space-axioms*)
have *target*: *product-prob-space ?N*
by (*rule product-prob-spaceI*) (*simp add*: *prob-space-centered-normal*)
have *target-sets*: *sets* (*proc-target brownian-process*) = *sets borel*
using *compatible-target*[*OF modificationD*(*1*)[*OF brownian-process-modification*]]
by (*simp add*: *gaussian-process-target*)
have *process-meas*: *process brownian-process u* ∈
borel-measurable (*proc-source brownian-process*) **for** *u*
using *stochastic-process-measurable*[*of brownian-process u*]
by (*simp only*: *measurable-cong-sets*[*OF refl target-sets*])
have *meas*: *X i* ∈ *measurable* (*?S i*) (*?N i*) **for** *i*
proof −
have *X i* ∈ *borel-measurable* (*proc-source brownian-process*)
unfolding *X-def* **by** (*measurable*; *rule process-meas*)
then show *?thesis*
by (*simp only*: *measurable-cong-sets*[*OF refl sets-centered-normal*])
qed
have *law*: *distr* (*?S i*) (*?N i*) (*X i*) = *?N i* **for** *i*
proof −
have *distr* (*proc-source brownian-process*) *borel* (*X i*) =
centered-normal $(t - s)$
unfolding *X-def* **by** (*rule brownian-process-increment-law*[*OF s st*])
moreover have *distr* (*proc-source brownian-process*) (*?N i*) (*X i*) =
distr (*proc-source brownian-process*) *borel* (*X i*)
by (*rule distr-cong*) *auto*
ultimately show *?thesis* **by** *simp*
qed
have *distr* (*PiM UNIV ?S*) (*PiM UNIV ?N*)
(*compose UNIV* (*X undefined*)) =
PiM UNIV ($\lambda i.$ *distr* (*?S i*) (*?N i*) (*X undefined*))
apply (*rule distr-PiM-finite-prob-space*)
apply *simp*
apply (*rule source*)

```
    apply (rule target)
   by (rule meas)
  also have ... = PiM UNIV ?N
   apply (rule PiM-cong)
    apply simp
   using law by (simp add: X-def)
  finally show ?thesis
   unfolding euclidean-brownian-source-def compose-def X-def by simp
qed

lemma euclidean-brownian-process-increment-law:
  assumes s: 0 ≤ s and st: s ≤ t
  shows distr (proc-source (euclidean-brownian-process ::
      (real, 'd::finite ⇒ real ⇒ real, real^'d) stochastic-process)) borel
      (λw. process euclidean-brownian-process t w −
        process euclidean-brownian-process s w) =
    euclidean-centered-normal (t − s)
proof −
  let ?S = euclidean-brownian-source :: ('d ⇒ real ⇒ real) measure
  let ?N = PiM UNIV (λ-::'d. centered-normal (t − s))
  let ?I = λw i. process brownian-process t (w i) −
    process brownian-process s (w i)
  let ?V = λx. χ i. x i :: real^'d
  have ps: prob-space ?S by (rule prob-space-euclidean-brownian-source)
  have nps: prob-space ?N
    by (rule prob-space-PiM) (simp add: prob-space-centered-normal)
  have target-sets: sets (proc-target brownian-process) = sets borel
   using compatible-target[OF modificationD(1)[OF brownian-process-modification]]
    by (simp add: gaussian-process-target)
  have scalar-meas: (λp. process brownian-process t p −
      process brownian-process s p) ∈
      borel-measurable (proc-source brownian-process)
    using stochastic-process-measurable[of brownian-process t]
      stochastic-process-measurable[of brownian-process s]
    by (simp only: measurable-cong-sets[OF refl target-sets]) measurable
  have coord-meas: (λw. ?I w i) ∈
      measurable ?S (centered-normal (t − s)) for i
  proof −
    have (λw. ?I w i) ∈ borel-measurable ?S
      unfolding euclidean-brownian-source-def
      by (rule measurable-PiM-component-rev) (simp-all add: scalar-meas)
    then show ?thesis
      by (simp only: measurable-cong-sets[OF refl sets-centered-normal])
  qed
  have Imeas: (λw. restrict (?I w) UNIV) ∈ measurable ?S ?N
    apply (rule measurable-PiM-single')
     apply (simp add: coord-meas)
    using measurable-space[OF coord-meas] by (auto simp: space-PiM)
  have Vmeas: ?V ∈ borel-measurable ?N
```

```
    apply (rule borel-measurable-vec-lambda)
    apply (rule measurable-PiM-component-rev)
     apply simp
    apply (rule measurable-ident-sets)
    by simp
  have inc: distr ?S ?N (λw. restrict (?I w) UNIV) = ?N
    by (rule euclidean-brownian-increment-product-law[OF s st, unfolded Let-def])
  have distr ?S borel (λw. ?V (restrict (?I w) UNIV)) =
      distr (distr ?S ?N (λw. restrict (?I w) UNIV)) borel ?V
    using distr-distr[OF Vmeas Imeas, symmetric]
    by (simp only: comp-def)
  also have ... = euclidean-centered-normal (t − s)
    unfolding euclidean-centered-normal-def inc by simp
  finally have raw-law: distr ?S borel
      (λw. ?V (restrict (?I w) UNIV)) =
      euclidean-centered-normal (t − s) .
  have coord-eq: (λw. ?V (restrict (?I w) UNIV)) =
      (λw. χ i. process brownian-process t (w i) −
        process brownian-process s (w i))
    by (simp add: fun-eq-iff vec-eq-iff)
  have law: distr ?S borel
      (λw. χ i. process brownian-process t (w i) −
        process brownian-process s (w i)) =
      euclidean-centered-normal (t − s)
    using raw-law by (simp only: coord-eq)
  have map-eq: (λw. process (euclidean-brownian-process ::
      (real, 'd ⇒ real ⇒ real, real^'d) stochastic-process) t w −
      process euclidean-brownian-process s w) =
    (λw. χ i. process brownian-process t (w i) −
      process brownian-process s (w i))
    using s st by (simp add: fun-eq-iff vec-eq-iff)
  show ?thesis
    by (simp only: euclidean-brownian-process-source map-eq law)
qed

lemma euclidean-brownian-process-local-holder:
  local-holder-on (1/8) {0..}
    (λt. process (euclidean-brownian-process ::
      (real, 'd::finite ⇒ real ⇒ real, real^'d) stochastic-process) t w)
proof −
  let ?g = (1/8 :: real)
  have scalar: local-holder-on ?g {0..}
      (λt. process brownian-process t (w i)) for i
    by (rule brownian-process-local-holder)
  have data: ∃ e C. 0 < e ∧ 0 ≤ C ∧
      (∀ r∈{0..}. ∀ s∈{0..}.
        dist s t < e ∧ dist r t < e ⟶
        dist (process brownian-process r (w i))
          (process brownian-process s (w i)) ≤ C * dist r s powr ?g)
```

```
    if t ∈ {0..} for i t
    using scalar that unfolding local-holder-on-def by blast
  show ?thesis
  proof (unfold local-holder-on-def, intro conjI ballI)
    show ?g ∈ {0<..1} by simp
    fix t :: real
    assume t: t ∈ {0..}
    obtain e C where e: ⋀i. 0 < e i and C: ⋀i. 0 ≤ C i
      and bound: ⋀i r s. r ∈ {0..} ⟹ s ∈ {0..} ⟹
        dist s t < e i ⟹ dist r t < e i ⟹
        dist (process brownian-process r (w i))
          (process brownian-process s (w i)) ≤ C i * dist r s powr ?g
      using data[OF t] by metis
    let ?e = Min (e ` (UNIV::'d set))
    let ?C = sum C (UNIV::'d set)
    have epos: 0 < ?e
      by (rule Min-gr-iff[THEN iffD2]) (auto simp: e)
    have e-le: ?e ≤ e i for i
      by (rule Min-le) auto
    have Cnonneg: 0 ≤ ?C by (simp add: sum-nonneg C)
    show ∃ e>0. ∃ K≥0. ∀ r∈{0..}. ∀ s∈{0..}.
        dist s t < e ∧ dist r t < e ⟶
        dist (process euclidean-brownian-process r w)
          (process euclidean-brownian-process s w) ≤ K * dist r s powr ?g
    proof (intro exI conjI ballI impI)
      show 0 < ?e by (rule epos)
      show 0 ≤ ?C by (rule Cnonneg)
      fix r s assume r: r ∈ {0..} and s: s ∈ {0..}
        and near: dist s t < ?e ∧ dist r t < ?e
      have component: dist (process brownian-process r (w i))
          (process brownian-process s (w i)) ≤
          C i * dist r s powr ?g for i
        by (rule bound[OF r s]) (use near e-le[of i] in linarith)+
      have dist (process euclidean-brownian-process r w)
          (process euclidean-brownian-process s w) ≤
          (∑ i∈(UNIV::'d set). dist (process brownian-process r (w i))
            (process brownian-process s (w i)))
        using r s unfolding dist-vec-def
        by (simp add: L2-set-le-sum)
      also have ... ≤ (∑ i∈(UNIV::'d set). C i * dist r s powr ?g)
        by (rule sum-mono) (rule component)
      also have ... = ?C * dist r s powr ?g
        by (simp add: sum-distrib-right)
      finally show dist (process euclidean-brownian-process r w)
          (process euclidean-brownian-process s w) ≤
          ?C * dist r s powr ?g .
    qed
  qed
qed
```

```
lemma euclidean-brownian-process-indep-increments:
  indep-increments (euclidean-brownian-process ::
    (real, 'd::finite ⇒ real ⇒ real, real^'d) stochastic-process)
proof (rule indep-incrementsI)
  fix l :: real list
  assume times: set l ⊆ proc-index (euclidean-brownian-process ::
      (real, 'd ⇒ real ⇒ real, real^'d) stochastic-process)
    and sorted: sorted l and length: 2 ≤ length l
  let ?J = {1..<length l}
  let ?D = λ(i,k) p. process brownian-process (l ! k) p −
    process brownian-process (l ! (k − 1)) p
  have scalar: prob-space.indep-vars (proc-source brownian-process)
      (λ-. borel) (λk p. ?D (i,k) p) ?J for i :: 'd
  proof −
    have times-scalar: set l ⊆ proc-index brownian-process
      using times constructed-brownian-motion.index by simp
    have iv0: prob-space.indep-vars (proc-source brownian-process)
        (λ-. proc-target brownian-process)
        (λk p. process brownian-process (l ! k) p −
          process brownian-process (l ! (k − 1)) p) ?J
      by (rule indep-incrementsE[OF brownian-process-indep-increments])
        (use times-scalar sorted length in auto)
    have iv: prob-space.indep-vars (proc-source brownian-process)
        (λ-. proc-target brownian-process) (λk p. ?D (i,k) p) ?J
      using iv0 by simp
    have target-sets: sets (proc-target brownian-process) = sets borel
    using compatible-target[OF modificationD(1)[OF brownian-process-modification]]
      by (simp add: gaussian-process-target)
    show ?thesis
      using iv
      unfolding prob-space.indep-vars-def2[OF proc-source.prob-space-axioms]
      by (simp only: measurable-cong-sets[OF refl target-sets] target-sets; blast)
  qed
  have flat0: prob-space.indep-vars (euclidean-brownian-source ::
      ('d ⇒ real ⇒ real) measure) (λ-. borel)
      (λ(i,k) w. process brownian-process (l ! k) (w i) −
        process brownian-process (l ! (k − 1)) (w i)) (UNIV × ?J)
  proof −
    let ?B = λ(-::'d) (-::nat). borel
    let ?E = λ(-::'d) k p. process brownian-process (l ! k) p −
      process brownian-process (l ! (k − 1)) p
    have raw: prob-space.indep-vars
        (PiM (UNIV::'d set) (λ-. proc-source brownian-process))
        (λ(i,k). ?B i k) (λ(i,k) w. ?E i k (w i))
        (UNIV × ?J)
      apply (rule indep-vars-PiM-finite-family)
        apply simp
       apply simp
```

```
      apply simp
     apply (use length in auto)
    apply (simp add: proc-source.prob-space-axioms)
   using scalar by simp
 have B-eq: (λ(i,k). ?B i k) = (λ-. borel)
  by (rule ext) (auto split: prod.splits)
 show ?thesis
  unfolding euclidean-brownian-source-def
  using raw by (simp only: B-eq)
qed
have flat: prob-space.indep-vars (euclidean-brownian-source ::
   ('d ⇒ real ⇒ real) measure) (λ-. borel)
   (λ(i,k) w. ?D (i,k) (w i)) (UNIV × ?J)
 using flat0 by simp
let ?K = λk. UNIV × {k} :: ('d × nat) set
have grouped: prob-space.indep-vars (euclidean-brownian-source ::
   ('d ⇒ real ⇒ real) measure)
   (λk. PiM (?K k) (λ-. borel))
   (λk w. restrict (λp. ?D p (w (fst p))) (?K k)) ?J
proof −
 let ?Z = λ(i,k) w. ?D (i,k) (w i)
 have raw: prob-space.indep-vars (euclidean-brownian-source ::
    ('d ⇒ real ⇒ real) measure)
    (λk. PiM (?K k) (λ-. borel))
    (λk w. restrict (λp. ?Z p w) (?K k)) ?J
 proof (rule prob-space.indep-vars-restrict[OF
      euclidean-source.prob-space-axioms flat])
  show ?K k ⊆ UNIV × ?J if k ∈ ?J for k
   using that by auto
  show disjoint-family-on ?K ?J
   by (auto simp: disjoint-family-on-def)
 qed
 have map-eq: (λw. restrict (λp. ?Z p w) (?K k)) =
    (λw. restrict (λp. ?D p (w (fst p))) (?K k)) for k
  by (rule ext, rule ext) (auto split: prod.splits)
 have transfer: prob-space.indep-vars (euclidean-brownian-source ::
    ('d ⇒ real ⇒ real) measure)
    (λk. PiM (?K k) (λ-. borel))
    (λk w. restrict (λp. ?Z p w) (?K k)) ?J =
   prob-space.indep-vars euclidean-brownian-source
    (λk. PiM (?K k) (λ-. borel))
    (λk w. restrict (λp. ?D p (w (fst p))) (?K k)) ?J
 proof (rule euclidean-source.indep-vars-cong)
  show ?J = ?J by simp
  show ⋀k. k ∈ ?J ⟹
   (λw. restrict (λp. ?Z p w) (?K k)) =
   (λw. restrict (λp. ?D p (w (fst p))) (?K k))
   by (rule map-eq)
  show ⋀k. k ∈ ?J ⟹
```

```
    PiM (?K k) (λ-. borel) = PiM (?K k) (λ-. borel)
    by simp
  qed
  show ?thesis using raw transfer by blast
qed
have vec-meas: (λf. (χ i. f (i,k)) :: real^'d) ∈
   measurable (PiM (?K k) (λ-. borel)) borel if kJ: k ∈ ?J for k
proof −
  have coord: (λf. f (i,k)) ∈
     borel-measurable (PiM (?K k) (λ-. borel)) for i
    by (rule measurable-PiM-component-rev) (use kJ in auto)
  show ?thesis
    by (rule borel-measurable-vec-lambda[where f=λx i. x (i,k)])
      (rule coord)
qed
have vectors: prob-space.indep-vars (euclidean-brownian-source ::
   ('d ⇒ real ⇒ real) measure) (λ-. borel)
   (λk w. χ i. ?D (i,k) (w i)) ?J
  using euclidean-source.indep-vars-compose2[OF grouped vec-meas]
  by (simp add: fun-eq-iff)
have nonneg: 0 ≤ l ! n if n < length l for n
  using times nth-mem[OF that] by auto
have map-eq: (λw. process (euclidean-brownian-process ::
   (real, 'd ⇒ real ⇒ real, real^'d) stochastic-process)
     (l ! k) w − process euclidean-brownian-process (l ! (k − 1)) w) =
  (λw. χ i. ?D (i,k) (w i)) if kJ: k ∈ ?J for k
proof −
  have klen: k < length l and km1len: k − 1 < length l
    using kJ by auto
  show ?thesis
    apply (rule ext)
    using nonneg[OF klen] nonneg[OF km1len]
    by (simp add: vec-eq-iff)
qed
have desired: prob-space.indep-vars (euclidean-brownian-source ::
   ('d ⇒ real ⇒ real) measure) (λ-. borel)
   (λk w. process (euclidean-brownian-process ::
     (real, 'd ⇒ real ⇒ real, real^'d) stochastic-process)
       (l ! k) w − process euclidean-brownian-process (l ! (k − 1)) w) ?J
proof −
  have transfer: prob-space.indep-vars (euclidean-brownian-source ::
     ('d ⇒ real ⇒ real) measure) (λ-. borel)
     (λk w. χ i. ?D (i,k) (w i)) ?J =
   prob-space.indep-vars euclidean-brownian-source (λ-. borel)
     (λk w. process (euclidean-brownian-process ::
       (real, 'd ⇒ real ⇒ real, real^'d) stochastic-process)
         (l ! k) w − process euclidean-brownian-process (l ! (k − 1)) w) ?J
  proof (rule euclidean-source.indep-vars-cong)
    show ?J = ?J by simp
```

```
    show ⋀k. k ∈ ?J ⟹
      (λw. χ i. ?D (i,k) (w i)) =
      (λw. process (euclidean-brownian-process ::
        (real, 'd ⇒ real ⇒ real, real^'d) stochastic-process)
          (l ! k) w − process euclidean-brownian-process (l ! (k − 1)) w)
      by (rule sym, rule map-eq) assumption
    show ⋀k. k ∈ ?J ⟹ borel = borel by simp
   qed
   show ?thesis by (rule transfer[THEN iffD1, OF vectors])
  qed
  show prob-space.indep-vars
     (proc-source (euclidean-brownian-process ::
       (real, 'd ⇒ real ⇒ real, real^'d) stochastic-process))
     (λ-. proc-target euclidean-brownian-process)
     (λk w. process euclidean-brownian-process (l ! k) w −
       process euclidean-brownian-process (l ! (k − 1)) w) ?J
   using desired by simp
qed

lemma euclidean-brownian-process-continuous:
  continuous-on {0..}
    (λt. process (euclidean-brownian-process ::
      (real, 'd::finite ⇒ real ⇒ real, real^'d) stochastic-process) t w)
 using euclidean-brownian-process-local-holder local-holder-imp-continuous by blast

interpretation constructed-euclidean-brownian-motion: euclidean-brownian-motion
  euclidean-brownian-process ::
    (real, 'd::finite ⇒ real ⇒ real, real^'d) stochastic-process
proof
  show proc-index (euclidean-brownian-process ::
      (real, 'd ⇒ real ⇒ real, real^'d) stochastic-process) = {0..}
    by simp
  show AE w in proc-source (euclidean-brownian-process ::
      (real, 'd ⇒ real ⇒ real, real^'d) stochastic-process).
      process euclidean-brownian-process 0 w = 0
    by (rule euclidean-brownian-process-starts-at-zero)
  show indep-increments (euclidean-brownian-process ::
      (real, 'd ⇒ real ⇒ real, real^'d) stochastic-process)
    by (rule euclidean-brownian-process-indep-increments)
  show distr (proc-source (euclidean-brownian-process ::
      (real, 'd ⇒ real ⇒ real, real^'d) stochastic-process)) borel
      (λw. process euclidean-brownian-process t w −
       process euclidean-brownian-process s w) = euclidean-centered-normal (t − s)
    if 0 ≤ s s ≤ t for s t
    by (rule euclidean-brownian-process-increment-law[OF that])
  show AE w in proc-source (euclidean-brownian-process ::
      (real, 'd ⇒ real ⇒ real, real^'d) stochastic-process).
      continuous-on {0..} (λt. process euclidean-brownian-process t w)
    by (rule AE-I2) (rule euclidean-brownian-process-continuous)
```

qed

end

References

[1] A. Klenke. *Probability Theory: A Comprehensive Course*. Springer, 3 edition, 2020.

[2] C. Pardillo-Laursen and S. Foster. The kolmogorov-chentsov theorem. *Archive of Formal Proofs*, Apr. 2025. https://isa-afp.org/entries/Kolmogorov_Chentsov.html, Formal proof development.