

Vector Calculus

Brian Huffman

August 27, 2008

Abstract

This development ...

Contents

1	Constructing Bounded Linear Operators	3
2	Frechet Derivative	5
2.1	Addition	5
2.2	Subtraction	6
2.3	Continuity	6
2.4	Composition	7
2.5	Product Rule	9
2.6	Powers	11
2.7	Inverse	11
2.8	Alternate definition	12
3	Inner Product Spaces	13
3.1	Real inner product spaces	13
3.2	Instances	16
4	Finite-Dimensional Vectors	17
4.1	Type definition	17
4.2	Vector arithmetic	17
4.3	Vector is a real vector space	19
4.4	Square root of sum of squares	19
4.5	Vectors are a real normed vector space	23
4.6	Vectors are an inner product space	24
4.7	Vector is a functor	25
4.8	Vector dot product	25
4.9	Single-component vectors	27

5	Square Matrices	28
5.1	Matrix arithmetic	28
5.2	Matrix multiplication	30
5.3	Numerical syntax for matrices	31
5.4	Matrices are a real vector space	32
5.5	Applying a matrix to a vector	32
5.6	Matrix norm	34
5.7	Vector outer product	38
6	Gradient Derivatives	39
6.1	Gradient Derivative	40
7	Pairs as Vector Spaces	41
7.1	Vector arithmetic	42
7.2	Product is a normed vector space	45
7.3	Product is an inner product space	46
8	Cross Products	47
9	Quaternions	48
9.1	Definition	48
9.2	Addition and Subtraction	49
9.3	Multiplication	49
9.4	Vector space	50
9.5	Norm and inner product	51
9.6	Conjugate	52
9.7	Inverse	53

1 Constructing Bounded Linear Operators

theory *BoundedLinear*

imports *Lim*

begin

lemma *bounded-linear-zero*: *bounded-linear* ($\lambda x. 0$)
by (*unfold-locales*, *rule-tac* [β] $x=0$ **in** *exI*, *simp-all*)

lemma *bounded-linear-ident*: *bounded-linear* ($\lambda x. x$)
by (*unfold-locales*, *rule-tac* [β] $x=1$ **in** *exI*, *simp-all*)

lemma *bounded-linear-uminus*: *bounded-linear* *uminus*
by (*unfold-locales*, *rule-tac* [β] $x=1$ **in** *exI*, *simp-all*)

lemma *bounded-linear-compose*:

includes *bounded-linear* *f*

includes *bounded-linear* *g*

shows *bounded-linear* ($\lambda x. f (g x)$)

proof (*unfold-locales*)

fix *x y* **show** $f (g (x + y)) = f (g x) + f (g y)$

by (*simp only*: *f.add g.add*)

next

fix *r x* **show** $f (scaleR r x) = scaleR r (f (g x))$

by (*simp only*: *f.scaleR g.scaleR*)

next

obtain *Kf* **where** $f: \bigwedge x. norm (f x) \leq norm x * Kf$ **and** *Kf*: $0 \leq Kf$

using *f.nonneg-bounded* **by** *fast*

obtain *Kg* **where** $g: \bigwedge x. norm (g x) \leq norm x * Kg$

using *g.bounded* **by** *fast*

show $\exists K. \forall x. norm (f (g x)) \leq norm x * K$

proof (*intro exI allI*)

fix *x*

have $norm (f (g x)) \leq norm (g x) * Kf$

using *f .*

also have $norm (g x) * Kf \leq (norm x * Kg) * Kf$

using *g Kf* **by** (*rule mult-right-mono*)

finally show $norm (f (g x)) \leq norm x * (Kg * Kf)$

by (*simp only*: *mult-assoc*)

qed

qed

lemma *bounded-linear-o*:

$\llbracket bounded-linear f; bounded-linear g \rrbracket \implies bounded-linear (f \circ g)$

unfolding *o-def* **by** (*rule bounded-linear-compose*)

lemma *bounded-linear-add*:

includes *bounded-linear* *f*

includes *bounded-linear* *g*

```

shows bounded-linear ( $\lambda x. f x + g x$ )
proof (unfold-locates)
  fix  $x y$  show  $f (x + y) + g (x + y) = (f x + g x) + (f y + g y)$ 
    by (simp only: f.add g.add add-ac)
next
  fix  $r x$  show  $f (scaleR r x) + g (scaleR r x) = scaleR r (f x + g x)$ 
    by (simp only: f.scaleR g.scaleR scaleR-right-distrib)
next
  obtain  $Kf$  where  $f: \bigwedge x. norm (f x) \leq norm x * Kf$ 
    using  $f.bounded$  by fast
  obtain  $Kg$  where  $g: \bigwedge x. norm (g x) \leq norm x * Kg$ 
    using  $g.bounded$  by fast
  show  $\exists K. \forall x. norm (f x + g x) \leq norm x * K$ 
  proof (intro exI allI)
    fix  $x$ 
    have  $norm (f x + g x) \leq norm (f x) + norm (g x)$ 
      by (rule norm-triangle-ineq)
    also have  $norm (f x) + norm (g x) \leq norm x * Kf + norm x * Kg$ 
      using  $f g$  by (rule add-mono)
    finally show  $norm (f x + g x) \leq norm x * (Kf + Kg)$ 
      by (simp only: right-distrib)
  qed
qed

```

```

lemma bounded-linear-minus:
  bounded-linear  $f \implies bounded-linear (\lambda x. - f x)$ 
by (rule bounded-linear-uminus [THEN bounded-linear-compose])

```

```

lemma bounded-linear-diff:
  [bounded-linear  $f$ ; bounded-linear  $g$ ]  $\implies bounded-linear (\lambda x. f x - g x)$ 
by (simp only: diff-minus bounded-linear-add bounded-linear-minus)

```

```

lemma bounded-linear-setsum:
   $\forall i \in A. bounded-linear (f i) \implies bounded-linear (\lambda x. \sum_{i \in A} f i x)$ 
apply (cases finite A)
apply (erule rev-mp, erule finite-induct)
apply (simp add: bounded-linear-zero)
apply (simp add: bounded-linear-add)
apply (simp add: bounded-linear-zero)
done

```

```

lemmas bounded-linear-scaleR =
  scaleR.bounded-linear-right [THEN bounded-linear-compose]

```

```

lemmas bounded-linear-mult-const =
  mult.bounded-linear-left [THEN bounded-linear-compose]

```

```

lemmas bounded-linear-const-mult =
  mult.bounded-linear-right [THEN bounded-linear-compose]

```

end

2 Frechet Derivative

theory *FrechetDeriv*
imports *BoundedLinear*
begin

definition

fderiv ::
[*'a::real-normed-vector* \Rightarrow *'b::real-normed-vector*, *'a*, *'a* \Rightarrow *'b*] \Rightarrow *bool*
— Frechet derivative: *D* is derivative of function *f* at *x*
 $((FDERIV (-) / (-) / :> (-)) [1000, 1000, 60] 60)$ **where**
 $FDERIV f x :> D = (bounded-linear D \wedge$
 $(\lambda h. norm (f (x + h) - f x - D h) / norm h) -- 0 --> 0)$

lemma *FDERIV-I*:

$\llbracket bounded-linear D; (\lambda h. norm (f (x + h) - f x - D h) / norm h) -- 0 --> 0 \rrbracket$
 $\implies FDERIV f x :> D$
by (*simp add: fderiv-def*)

lemma *FDERIV-D*:

$FDERIV f x :> D \implies (\lambda h. norm (f (x + h) - f x - D h) / norm h) -- 0 --> 0$
by (*simp add: fderiv-def*)

lemma *FDERIV-bounded-linear*: $FDERIV f x :> D \implies bounded-linear D$

by (*simp add: fderiv-def*)

lemma *FDERIV-const*: $FDERIV (\lambda x. k) x :> (\lambda h. 0)$

by (*simp add: fderiv-def bounded-linear-zero*)

lemma *FDERIV-ident*: $FDERIV (\lambda x. x) x :> (\lambda h. h)$

by (*simp add: fderiv-def bounded-linear-ident*)

2.1 Addition

lemma *norm-ratio-ineq*:

fixes *x y* :: *'a::real-normed-vector*

fixes *h* :: *'b::real-normed-vector*

shows $norm (x + y) / norm h \leq norm x / norm h + norm y / norm h$

apply (*rule ord-le-eq-trans*)

apply (*rule divide-right-mono*)

apply (*rule norm-triangle-ineq*)

apply (*rule norm-ge-zero*)

apply (*rule add-divide-distrib*)

done

lemma *FDERIV-add*:

assumes $f: FDERIV f x :=> F$

assumes $g: FDERIV g x :=> G$

shows $FDERIV (\lambda x. f x + g x) x :=> (\lambda h. F h + G h)$

proof (rule *FDERIV-I*)

show bounded-linear $(\lambda h. F h + G h)$

using $f g$ by (intro bounded-linear-add *FDERIV-bounded-linear*)

next

have $f': (\lambda h. norm (f (x + h) - f x - F h) / norm h) \dashrightarrow 0 \dashrightarrow 0$

using f by (rule *FDERIV-D*)

have $g': (\lambda h. norm (g (x + h) - g x - G h) / norm h) \dashrightarrow 0 \dashrightarrow 0$

using g by (rule *FDERIV-D*)

have $(\lambda h. norm (f (x + h) - f x - F h) / norm h$
 $+ norm (g (x + h) - g x - G h) / norm h) \dashrightarrow 0 \dashrightarrow 0$

using $f' g'$ by (rule *LIM-add-zero*)

thus $(\lambda h. norm (f (x + h) + g (x + h) - (f x + g x) - (F h + G h))$
 $/ norm h) \dashrightarrow 0 \dashrightarrow 0$

apply (rule *real-LIM-sandwich-zero*)

apply (simp add: *divide-nonneg-pos*)

apply (simp only: *add-diff-add*)

apply (rule *norm-ratio-ineq*)

done

qed

2.2 Subtraction

lemma *FDERIV-minus*:

$FDERIV f x :=> F \implies FDERIV (\lambda x. - f x) x :=> (\lambda h. - F h)$

apply (rule *FDERIV-I*)

apply (rule *bounded-linear-minus*)

apply (erule *FDERIV-bounded-linear*)

apply (simp only: *fderv-def minus-diff-minus norm-minus-cancel*)

done

lemma *FDERIV-diff*:

$\llbracket FDERIV f x :=> F; FDERIV g x :=> G \rrbracket$

$\implies FDERIV (\lambda x. f x - g x) x :=> (\lambda h. F h - G h)$

by (simp only: *diff-minus FDERIV-add FDERIV-minus*)

2.3 Continuity

lemma *FDERIV-isCont*:

assumes $f: FDERIV f x :=> F$

shows *isCont* $f x$

proof -

from f interpret $F: bounded-linear [F]$ by (rule *FDERIV-bounded-linear*)

have $(\lambda h. norm (f (x + h) - f x - F h) / norm h) \dashrightarrow 0 \dashrightarrow 0$

using f by (rule *FDERIV-D*)

hence $(\lambda h. \text{norm } (f (x + h) - f x - F h) / \text{norm } h * \text{norm } h) \text{ -- } 0 \text{ --} > 0$
by (*intro LIM-mult-zero LIM-norm-zero LIM-ident*)
hence $(\lambda h. \text{norm } (f (x + h) - f x - F h)) \text{ -- } 0 \text{ --} > 0$
by (*simp cong: LIM-cong*)
hence $(\lambda h. f (x + h) - f x - F h) \text{ -- } 0 \text{ --} > 0$
by (*rule LIM-norm-zero-cancel*)
hence $(\lambda h. f (x + h) - f x - F h + F h) \text{ -- } 0 \text{ --} > 0$
by (*intro LIM-add-zero F.LIM-zero LIM-ident*)
hence $(\lambda h. f (x + h) - f x) \text{ -- } 0 \text{ --} > 0$
by *simp*
thus *isCont f x*
unfolding *isCont-iff* **by** (*rule LIM-zero-cancel*)
qed

2.4 Composition

lemma *real-divide-cancel-lemma*:

fixes $a b c :: 'a::\{\text{field}, \text{division-by-zero}\}$
shows $(b = 0 \implies a = 0) \implies (a / b) * (b / c) = a / c$
by *simp*

lemma *FDERIV-compose*:

fixes $f :: 'a::\text{real-normed-vector} \Rightarrow 'b::\text{real-normed-vector}$
fixes $g :: 'b::\text{real-normed-vector} \Rightarrow 'c::\text{real-normed-vector}$
assumes $f: \text{FDERIV } f \ x \ :> \ F$
assumes $g: \text{FDERIV } g \ (f \ x) \ :> \ G$
shows $\text{FDERIV } (\lambda x. g \ (f \ x)) \ x \ :> \ (\lambda h. G \ (F \ h))$
proof (*rule FDERIV-I*)
from *FDERIV-bounded-linear* $[OF \ g]$ *FDERIV-bounded-linear* $[OF \ f]$
show *bounded-linear* $(\lambda h. G \ (F \ h))$
by (*rule bounded-linear-compose*)
next
let $?Rf = \lambda h. f \ (x + h) - f \ x - F \ h$
let $?Rg = \lambda k. g \ (f \ x + k) - g \ (f \ x) - G \ k$
let $?k = \lambda h. f \ (x + h) - f \ x$
let $?Nf = \lambda h. \text{norm } (?Rf \ h) / \text{norm } h$
let $?Ng = \lambda h. \text{norm } (?Rg \ (?k \ h)) / \text{norm } (?k \ h)$
from f **interpret** $F: \text{bounded-linear } [F]$ **by** (*rule FDERIV-bounded-linear*)
from g **interpret** $G: \text{bounded-linear } [G]$ **by** (*rule FDERIV-bounded-linear*)
from $F.\text{bounded}$ **obtain** kF **where** $kF: \bigwedge x. \text{norm } (F \ x) \leq \text{norm } x * kF$ **by** *fast*
from $G.\text{bounded}$ **obtain** kG **where** $kG: \bigwedge x. \text{norm } (G \ x) \leq \text{norm } x * kG$ **by** *fast*

let $?fun2 = \lambda h. ?Nf \ h * kG + ?Ng \ h * (?Nf \ h + kF)$

show $(\lambda h. \text{norm } (g \ (f \ (x + h)) - g \ (f \ x) - G \ (F \ h)) / \text{norm } h) \text{ -- } 0 \text{ --} > 0$

proof (*rule real-LIM-sandwich-zero*)

have $Nf: ?Nf \text{ -- } 0 \text{ --} > 0$

using *FDERIV-D* $[OF \ f]$.

```

have Ng1: isCont (λk. norm (?Rg k) / norm k) 0
  by (simp add: isCont-def FDERIV-D [OF g])
have Ng2: ?k -- 0 --> 0
  apply (rule LIM-zero)
  apply (fold isCont-iff)
  apply (rule FDERIV-isCont [OF f])
  done
have Ng: ?Ng -- 0 --> 0
  using isCont-LIM-compose [OF Ng1 Ng2] by simp

have (λh. ?Nf h * kG + ?Ng h * (?Nf h + kF))
  -- 0 --> 0 * kG + 0 * (0 + kF)
  by (intro LIM-add LIM-mult LIM-const Nf Ng)
thus (λh. ?Nf h * kG + ?Ng h * (?Nf h + kF)) -- 0 --> 0
  by simp
next
fix h::'a assume h: h ≠ 0
thus 0 ≤ norm (g (f (x + h)) - g (f x) - G (F h)) / norm h
  by (simp add: divide-nonneg-pos)
next
fix h::'a assume h: h ≠ 0
have g (f (x + h)) - g (f x) - G (F h) = G (?Rf h) + ?Rg (?k h)
  by (simp add: G.diff)
hence norm (g (f (x + h)) - g (f x) - G (F h)) / norm h
  = norm (G (?Rf h) + ?Rg (?k h)) / norm h
  by (rule arg-cong)
also have ... ≤ norm (G (?Rf h)) / norm h + norm (?Rg (?k h)) / norm h
  by (rule norm-ratio-ineq)
also have ... ≤ ?Nf h * kG + ?Ng h * (?Nf h + kF)
proof (rule add-mono)
  show norm (G (?Rf h)) / norm h ≤ ?Nf h * kG
    apply (rule ord-le-eq-trans)
    apply (rule divide-right-mono [OF kG norm-ge-zero])
    apply simp
  done
next
have norm (?Rg (?k h)) / norm h = ?Ng h * (norm (?k h) / norm h)
  apply (rule real-divide-cancel-lemma [symmetric])
  apply (simp add: G.zero)
  done
also have ... ≤ ?Ng h * (?Nf h + kF)
proof (rule mult-left-mono)
  have norm (?k h) / norm h = norm (?Rf h + F h) / norm h
    by simp
  also have ... ≤ ?Nf h + norm (F h) / norm h
    by (rule norm-ratio-ineq)
  also have ... ≤ ?Nf h + kF
    apply (rule add-left-mono)

```

```

    apply (subst pos-divide-le-eq, simp add: h)
    apply (subst mult-commute)
    apply (rule kF)
    done
  finally show norm (?k h) / norm h ≤ ?Nf h + kF .
next
  show 0 ≤ ?Ng h
  apply (case-tac f (x + h) - f x = 0, simp)
  apply (rule divide-nonneg-pos [OF norm-ge-zero])
  apply simp
  done
qed
  finally show norm (?Rg (?k h)) / norm h ≤ ?Ng h * (?Nf h + kF) .
qed
  finally show norm (g (f (x + h)) - g (f x) - G (F h)) / norm h
    ≤ ?Nf h * kG + ?Ng h * (?Nf h + kF) .
qed
qed

```

2.5 Product Rule

lemma (in *bounded-bilinear*) *FDERIV-lemma*:

```

  a' ** b' - a ** b - (a ** B + A ** b)
  = a ** (b' - b - B) + (a' - a - A) ** b' + A ** (b' - b)
by (simp add: diff-left diff-right)

```

lemma (in *bounded-bilinear*) *FDERIV*:

```

  fixes x :: 'd::real-normed-vector
  assumes f: FDERIV f x :> F
  assumes g: FDERIV g x :> G
  shows FDERIV (λx. f x ** g x) x :> (λh. f x ** G h + F h ** g x)
proof (rule FDERIV-I)
  show bounded-linear (λh. f x ** G h + F h ** g x)
    apply (rule bounded-linear-add)
    apply (rule bounded-linear-compose [OF bounded-linear-right])
    apply (rule FDERIV-bounded-linear [OF g])
    apply (rule bounded-linear-compose [OF bounded-linear-left])
    apply (rule FDERIV-bounded-linear [OF f])
  done

```

next

```

  from bounded-linear.bounded [OF FDERIV-bounded-linear [OF f]]
  obtain KF where norm-F: λx. norm (F x) ≤ norm x * KF by fast

```

from *pos-bounded* **obtain** *K* **where** $0 < K$ **and** *norm-prod*:

```

  λa b. norm (a ** b) ≤ norm a * norm b * K by fast

```

```

  let ?Rf = λh. f (x + h) - f x - F h
  let ?Rg = λh. g (x + h) - g x - G h

```

```

let ?fun1 = λh.
  norm (f x ** ?Rg h + ?Rf h ** g (x + h) + F h ** (g (x + h) - g x)) /
  norm h

let ?fun2 = λh.
  norm (f x) * (norm (?Rg h) / norm h) * K +
  norm (?Rf h) / norm h * norm (g (x + h)) * K +
  KF * norm (g (x + h) - g x) * K

have ?fun1 -- 0 --> 0
proof (rule real-LIM-sandwich-zero)
  from f g isCont-iff [THEN iffD1, OF FDERIV-isCont [OF g]]
  have ?fun2 -- 0 -->
    norm (f x) * 0 * K + 0 * norm (g x) * K + KF * norm (0::'b) * K
  by (intro LIM-add LIM-mult LIM-const LIM-norm LIM-zero FDERIV-D)
  thus ?fun2 -- 0 --> 0
  by simp
next
  fix h::'d assume h ≠ 0
  thus 0 ≤ ?fun1 h
  by (simp add: divide-nonneg-pos)
next
  fix h::'d assume h ≠ 0
  have ?fun1 h ≤ (norm (f x) * norm (?Rg h) * K +
    norm (?Rf h) * norm (g (x + h)) * K +
    norm h * KF * norm (g (x + h) - g x) * K) / norm h
  by (intro
    divide-right-mono mult-mono'
    order-trans [OF norm-triangle-ineq add-mono]
    order-trans [OF norm-prod mult-right-mono]
    mult-nonneg-nonneg order-refl norm-ge-zero norm-F
    K [THEN order-less-imp-le]
  )
  also have ... = ?fun2 h
  by (simp add: add-divide-distrib)
  finally show ?fun1 h ≤ ?fun2 h .
qed
thus (λh.
  norm (f (x + h) ** g (x + h) - f x ** g x - (f x ** G h + F h ** g x))
  / norm h) -- 0 --> 0
  by (simp only: FDERIV-lemma)
qed

```

lemmas FDERIV-mult = mult.FDERIV

lemmas FDERIV-scaleR = scaleR.FDERIV

2.6 Powers

lemma *FDERIV-power-Suc*:

```

fixes  $x :: 'a::\{\text{real-normed-algebra,recpower,comm-ring-1}\}$ 
shows FDERIV  $(\lambda x. x \wedge \text{Suc } n) x := (\lambda h. (1 + \text{of-nat } n) * x \wedge n * h)$ 
apply (induct  $n$ )
apply (simp add: power-Suc FDERIV-ident)
apply (drule FDERIV-mult [OF FDERIV-ident])
apply (simp only: of-nat-Suc left-distrib mult-1-left)
apply (simp only: power-Suc right-distrib mult-ac add-ac)
done

```

lemma *FDERIV-power*:

```

fixes  $x :: 'a::\{\text{real-normed-algebra,recpower,comm-ring-1}\}$ 
shows FDERIV  $(\lambda x. x \wedge n) x := (\lambda h. \text{of-nat } n * x \wedge (n - 1) * h)$ 
by (cases  $n$ , simp add: FDERIV-const, simp add: FDERIV-power-Suc)

```

2.7 Inverse

lemma *FDERIV-inverse*:

```

fixes  $x :: 'a::\text{real-normed-div-algebra}$ 
assumes  $x: x \neq 0$ 
shows FDERIV inverse  $x := (\lambda h. - (inverse\ x * h * inverse\ x))$ 
  (is FDERIV ?inv - := -)
proof (rule FDERIV-I)
  show bounded-linear  $(\lambda h. - (?inv\ x * h * ?inv\ x))$ 
    apply (rule bounded-linear-minus)
    apply (rule bounded-linear-mult-const)
    apply (rule bounded-linear-const-mult)
    apply (rule bounded-linear-ident)
  done
next
  show  $(\lambda h. \text{norm } (?inv\ (x + h) - ?inv\ x - - (?inv\ x * h * ?inv\ x)) / \text{norm } h)$ 
     $-- 0 --> 0$ 
  proof (rule LIM-equal2)
    show  $0 < \text{norm } x$  using  $x$  by simp
  next
  fix  $h::'a$ 
  assume  $1: h \neq 0$ 
  assume  $\text{norm } (h - 0) < \text{norm } x$ 
  hence  $h \neq -x$  by clarsimp
  hence  $2: x + h \neq 0$ 
    apply (rule contrapos-nn)
    apply (rule sym)
    apply (erule equals-zero-I)
  done
  show  $\text{norm } (?inv\ (x + h) - ?inv\ x - - (?inv\ x * h * ?inv\ x)) / \text{norm } h$ 
     $= \text{norm } ((?inv\ (x + h) - ?inv\ x) * h * ?inv\ x) / \text{norm } h$ 
    apply (subst inverse-diff-inverse [OF 2 x])
    apply (subst minus-diff-minus)

```

```

    apply (subst norm-minus-cancel)
    apply (simp add: left-diff-distrib)
  done
next
show (λh. norm ((?inv (x + h) - ?inv x) * h * ?inv x) / norm h)
  -- 0 --> 0
proof (rule real-LIM-sandwich-zero)
  show (λh. norm (?inv (x + h) - ?inv x) * norm (?inv x))
    -- 0 --> 0
    apply (rule LIM-mult-left-zero)
    apply (rule LIM-norm-zero)
    apply (rule LIM-zero)
    apply (rule LIM-offset-zero)
    apply (rule LIM-inverse)
    apply (rule LIM-ident)
    apply (rule x)
  done
next
fix h::'a assume h: h ≠ 0
show 0 ≤ norm ((?inv (x + h) - ?inv x) * h * ?inv x) / norm h
  apply (rule divide-nonneg-pos)
  apply (rule norm-ge-zero)
  apply (simp add: h)
done
next
fix h::'a assume h: h ≠ 0
have norm ((?inv (x + h) - ?inv x) * h * ?inv x) / norm h
  ≤ norm (?inv (x + h) - ?inv x) * norm h * norm (?inv x) / norm h
  apply (rule divide-right-mono [OF - norm-ge-zero])
  apply (rule order-trans [OF norm-mult-ineq])
  apply (rule mult-right-mono [OF - norm-ge-zero])
  apply (rule norm-mult-ineq)
done
also have ... = norm (?inv (x + h) - ?inv x) * norm (?inv x)
  by simp
finally show norm ((?inv (x + h) - ?inv x) * h * ?inv x) / norm h
  ≤ norm (?inv (x + h) - ?inv x) * norm (?inv x) .
qed
qed
qed

```

2.8 Alternate definition

lemma *field-fderiv-def*:

fixes $x :: 'a :: \text{real-normed-field}$ shows

$FDERIV f x :> (\lambda h. h * D) = (\lambda h. (f (x + h) - f x) / h) \text{ -- } 0 \text{ --> } D$

unfolding *fderiv-def*

apply (simp add: mult.bounded-linear-left)

apply (simp cong: LIM-cong add: nonzero-norm-divide [symmetric])

```

apply (subst diff-divide-distrib)
apply (subst times-divide-eq-right [symmetric])
apply (simp cong: LIM-cong add: divide-self)
apply (simp add: LIM-norm-zero-iff LIM-zero-iff)
done

end

```

3 Inner Product Spaces

```

theory InnerProduct
imports Complex FrechetDeriv
begin

```

3.1 Real inner product spaces

```

class inner = type +
  fixes inner :: 'a ⇒ 'a ⇒ real

class real-inner = real-vector + inner + sgn-div-norm +
  assumes inner-commute:  $inner\ x\ y = inner\ y\ x$ 
  and inner-left-distrib:  $inner\ (x + y)\ z = inner\ x\ z + inner\ y\ z$ 
  and inner-scaleR-left:  $inner\ (scaleR\ r\ x)\ y = r * (inner\ x\ y)$ 
  and inner-ge-zero:  $0 \leq inner\ x\ x$ 
  and inner-eq-zero:  $(inner\ x\ x = 0) = (x = 0)$ 
  and norm-eq-sqrt-inner:  $norm\ x = sqrt\ (inner\ x\ x)$ 

lemma inner-right-distrib:
  fixes  $x\ y\ z :: 'a::real-inner$ 
  shows  $inner\ x\ (y + z) = inner\ x\ y + inner\ x\ z$ 
proof –
  have  $inner\ x\ (y + z) = inner\ (y + z)\ x$ 
    by (rule inner-commute)
  also have  $\dots = inner\ y\ x + inner\ z\ x$ 
    by (rule inner-left-distrib)
  finally show ?thesis
    by (simp add: inner-commute)
qed

lemma inner-scaleR-right:
  fixes  $x\ y\ z :: 'a::real-inner$ 
  shows  $inner\ x\ (scaleR\ r\ y) = r * (inner\ x\ y)$ 
proof –
  have  $inner\ x\ (scaleR\ r\ y) = inner\ (scaleR\ r\ y)\ x$ 
    by (rule inner-commute)
  also have  $\dots = r * (inner\ y\ x)$ 
    by (rule inner-scaleR-left)
  finally show ?thesis

```

by (*simp add: inner-commute*)
 qed

interpretation *inner-left: additive* $[(\lambda x :: 'a :: \text{real-inner}. \text{inner } x \ y)]$
 by *unfold-locales (rule inner-left-distrib)*

interpretation *inner-right: additive* $[(\lambda y :: 'a :: \text{real-inner}. \text{inner } x \ y)]$
 by *unfold-locales (rule inner-right-distrib)*

lemmas *inner-zero-left [simp] = inner-left.zero*
lemmas *inner-zero-right [simp] = inner-right.zero*
lemmas *inner-left-diff-distrib = inner-left.diff*
lemmas *inner-right-diff-distrib = inner-right.diff*

lemmas *inner-distrib = inner-left-distrib inner-right-distrib*
lemmas *inner-diff = inner-left-diff-distrib inner-right-diff-distrib*
lemmas *inner-scaleR = inner-scaleR-left inner-scaleR-right*

lemma *zero-less-inner-iff*: $(0 < \text{inner } x \ x) = (x \neq (0 :: 'a :: \text{real-inner}))$
 by (*simp add: order-less-le inner-ge-zero inner-eq-zero*)

lemma *Cauchy-Schwartz-ineq*:
 fixes $x \ y :: 'a :: \text{real-inner}$
 shows $(\text{inner } x \ y)^2 \leq \text{inner } x \ x * \text{inner } y \ y$
proof (*cases*)
 assume $y = 0$
 thus *?thesis* by *simp*

next
 assume $y \neq 0$
 let $?r = \text{inner } x \ y / \text{inner } y \ y$
 have $0 \leq \text{inner } (x - \text{scaleR } ?r \ y) (x - \text{scaleR } ?r \ y)$
 by (*rule inner-ge-zero*)
 also have $\dots = \text{inner } x \ x - \text{inner } y \ x * ?r$
 by (*simp add: inner-left-diff-distrib inner-right-diff-distrib inner-scaleR-left inner-scaleR-right*)
 also have $\dots = \text{inner } x \ x - (\text{inner } x \ y)^2 / \text{inner } y \ y$
 by (*simp add: power2-eq-square inner-commute*)
 finally have $0 \leq \text{inner } x \ x - (\text{inner } x \ y)^2 / \text{inner } y \ y$.
 hence $(\text{inner } x \ y)^2 / \text{inner } y \ y \leq \text{inner } x \ x$
 by (*simp add: le-diff-eq*)
 thus $(\text{inner } x \ y)^2 \leq \text{inner } x \ x * \text{inner } y \ y$
 by (*simp add: pos-divide-le-eq zero-less-inner-iff y*)
 qed

lemma *norm-squared-eq-inner*: $(\text{norm } x)^2 = \text{inner } x \ (x :: 'a :: \text{real-inner})$
 by (*simp add: norm-eq-sqrt-inner inner-ge-zero*)

lemma *Cauchy-Schwartz-ineq2*:
 fixes $x \ y :: 'a :: \text{real-inner}$

```

shows |inner x y| ≤ norm x * norm y
proof (rule power2-le-imp-le)
  have (inner x y)2 ≤ inner x x * inner y y
    using Cauchy-Schwartz-ineq .
  thus |inner x y|2 ≤ (norm x * norm y)2
    by (simp add: power-mult-distrib norm-squared-eq-inner)
  show 0 ≤ norm x * norm y
    unfolding norm-eq-sqrt-inner
    by (intro mult-nonneg-nonneg real-sqrt-ge-zero inner-ge-zero)
qed

```

instance real-inner ⊆ real-normed-vector

```

proof
  fix a :: real and x y :: 'a::real-inner
  show 0 ≤ norm x
    unfolding norm-eq-sqrt-inner by (simp add: inner-ge-zero)
  show (norm x = 0) = (x = 0)
    unfolding norm-eq-sqrt-inner by (simp add: inner-eq-zero)
  show norm (x + y) ≤ norm x + norm y
    proof (rule power2-le-imp-le)
      have inner x y ≤ norm x * norm y
        by (rule order-trans [OF abs-ge-self Cauchy-Schwartz-ineq2])
      thus (norm (x + y))2 ≤ (norm x + norm y)2
        unfolding power2-sum norm-squared-eq-inner
        by (simp add: inner-distrib inner-commute)
      show 0 ≤ norm x + norm y
        unfolding norm-eq-sqrt-inner
        by (simp add: add-nonneg-nonneg inner-ge-zero)
    qed
  show norm (a *R x) = |a| * norm x
    unfolding norm-eq-sqrt-inner
    apply (simp add: inner-scaleR)
    apply (simp add: mult-assoc [symmetric] power2-eq-square [symmetric])
    apply (simp add: real-sqrt-mult-distrib)
    done
qed

```

interpretation inner:

```

  bounded-bilinear [inner::'a::real-inner ⇒ 'a ⇒ real]
  apply (unfold-locales)
  apply (unfold real-scaleR-def)
  apply (rule inner-left-distrib)
  apply (rule inner-right-distrib)
  apply (rule inner-scaleR-left)
  apply (rule inner-scaleR-right)
  apply (rule-tac x=1 in exI)
  apply (simp add: Cauchy-Schwartz-ineq2)
  done

```

interpretation *inner-left*:
bounded-linear [$\lambda x::'a::\text{real-inner. inner } x \ y$]
by (*rule inner.bounded-linear-left*)

interpretation *inner-right*:
bounded-linear [$\lambda y::'a::\text{real-inner. inner } x \ y$]
by (*rule inner.bounded-linear-right*)

3.2 Instances

instantiation *real* :: *real-inner*
begin

definition
inner-real-def [*simp*]: *inner* = *op* *

instance
apply (*intro-classes, unfold inner-real-def real-scaleR-def*)
apply (*rule mult-commute*)
apply (*rule left-distrib*)
apply (*rule mult-assoc*)
apply *simp*
apply *simp*
apply *simp*
done

end

lemma *add-nonneg-eq-0-iff*:
fixes $x \ y :: 'a::\text{pordered-ab-group-add}$
assumes $x: 0 \leq x$ **and** $y: 0 \leq y$
shows $x + y = 0 \longleftrightarrow x = 0 \wedge y = 0$
proof (*auto*)
have $x = x + 0$ **by** *simp*
also have $x + 0 \leq x + y$ **using** y **by** (*rule add-left-mono*)
also assume $x + y = 0$
also have $0 \leq x$ **using** x .
finally show $x = 0$.
next
have $y = 0 + y$ **by** *simp*
also have $0 + y \leq x + y$ **using** x **by** (*rule add-right-mono*)
also assume $x + y = 0$
also have $0 \leq y$ **using** y .
finally show $y = 0$.
qed

instantiation *complex* :: *real-inner*
begin

definition

inner-complex-def: $inner\ x\ y = Re\ x * Re\ y + Im\ x * Im\ y$

instance

apply (*intro-classes*, *unfold inner-complex-def*)
apply (*simp add: mult-commute*)
apply (*simp add: left-distrib*)
apply (*simp add: right-distrib*)
apply (*simp add: power2-eq-square [symmetric]*)
apply (*simp add: add-nonneg-eq-0-iff complex-Re-Im-cancel-iff*)
apply (*simp add: complex-norm-def power2-eq-square*)
done

end

end

4 Finite-Dimensional Vectors

theory *VectorType*
imports *InnerProduct*
begin

4.1 Type definition

typedef (**open**) (*'a, 'n*) *vec* (**infixl** $\wedge 80$) = *UNIV* :: (*'n* \Rightarrow *'a*) *set* ..

declare *Abs-vec-inverse* [*simplified, iff*]

lemma *expand-vec-eq*:

$x = y \longleftrightarrow (\forall i. Rep-vec\ x\ i = Rep-vec\ y\ i)$

by (*simp add: Rep-vec-inject [symmetric] expand-fun-eq*)

lemma *vec-ext*:

$(\bigwedge i. Rep-vec\ x\ i = Rep-vec\ y\ i) \Longrightarrow x = y$

by (*simp add: expand-vec-eq*)

4.2 Vector arithmetic

instantiation *vec* :: (*zero, type*) *zero*
begin

definition

zero-vec-def: $0 = Abs-vec\ (\lambda i. 0)$

instance ..

end

instantiation *vec* :: (*plus*, *type*) *plus*
begin

definition

plus-vec-def: $x + y = \text{Abs-vec } (\lambda i. \text{Rep-vec } x \ i + \text{Rep-vec } y \ i)$

instance ..
end

instantiation *vec* :: (*minus*, *type*) *minus*
begin

definition

minus-vec-def: $x - y = \text{Abs-vec } (\lambda i. \text{Rep-vec } x \ i - \text{Rep-vec } y \ i)$

instance ..
end

instantiation *vec* :: (*uminus*, *type*) *uminus*
begin

definition

uminus-vec-def: $- x = \text{Abs-vec } (\lambda i. - \text{Rep-vec } x \ i)$

instance ..
end

lemma *Rep-vec-zero* [*simp*]: $\text{Rep-vec } 0 \ i = 0$
unfolding *zero-vec-def* **by** *simp*

lemma *Rep-vec-add* [*simp*]:
 $\text{Rep-vec } (x + y) \ i = \text{Rep-vec } x \ i + \text{Rep-vec } y \ i$
unfolding *plus-vec-def* **by** *simp*

lemma *Rep-vec-diff* [*simp*]:
 $\text{Rep-vec } (x - y) \ i = \text{Rep-vec } x \ i - \text{Rep-vec } y \ i$
unfolding *minus-vec-def* **by** *simp*

lemma *Rep-vec-uminus* [*simp*]:
 $\text{Rep-vec } (- x) \ i = - \text{Rep-vec } x \ i$
unfolding *uminus-vec-def* **by** *simp*

Addition is associative

instance *vec* :: (*semigroup-add*, *type*) *semigroup-add*
by (*intro-classes*, *simp add: expand-vec-eq add-assoc*)

Addition is commutative

instance *vec* :: (*ab-semigroup-add*, *type*) *ab-semigroup-add*
by (*intro-classes*, *simp add: expand-vec-eq add-commute*)

Zero is additive identity

```
instance vec :: (comm-monoid-add, type) comm-monoid-add  
by (intro-classes, simp add: expand-vec-eq)
```

Addition has cancellation property

```
instance vec :: (cancel-semigroup-add, type) cancel-semigroup-add  
by (intro-classes, simp-all add: expand-vec-eq)
```

```
instance vec :: (cancel-ab-semigroup-add, type) cancel-ab-semigroup-add  
by (intro-classes, simp add: expand-vec-eq)
```

Negation is additive inverse

```
instance vec :: (ab-group-add, type) ab-group-add  
by (intro-classes, simp-all add: expand-vec-eq)
```

4.3 Vector is a real vector space

```
instantiation vec :: (scaleR, type) scaleR  
begin
```

definition

```
scaleR-vec-def: scaleR r x = Abs-vec ( $\lambda i$ . scaleR r (Rep-vec x i))
```

```
instance ..  
end
```

lemma *Rep-vec-scaleR* [*simp*]:

```
Rep-vec (scaleR r x) i = scaleR r (Rep-vec x i)  
unfolding scaleR-vec-def by simp
```

```
instance vec :: (real-vector, type) real-vector
```

```
apply (intro-classes)
```

```
apply (simp-all add: expand-vec-eq)
```

```
apply (simp add: scaleR-right-distrib)
```

```
apply (simp add: scaleR-left-distrib)
```

```
done
```

4.4 Square root of sum of squares

definition

```
set-l2 f A = sqrt ( $\sum_{i \in A} (f i)^2$ )
```

lemma *set-l2-infinite* [*simp*]: \neg *finite A* \implies *set-l2 f A = 0*

```
unfolding set-l2-def by simp
```

lemma *set-l2-empty* [*simp*]: *set-l2 f {} = 0*

```
unfolding set-l2-def by simp
```

lemma *set-l2-insert* [*simp*]:

$\llbracket \text{finite } F; a \notin F \rrbracket \implies$
 $\text{set-l2 } f (\text{insert } a \ F) = \text{sqrt } ((f \ a)^2 + (\text{set-l2 } f \ F)^2)$
unfolding *set-l2-def* **by** (*simp add: setsum-nonneg*)

lemma *set-l2-nonneg* [*simp*]: $0 \leq \text{set-l2 } f \ A$
unfolding *set-l2-def* **by** (*simp add: setsum-nonneg*)

lemma *set-l2-0'*: $\forall a \in A. f \ a = 0 \implies \text{set-l2 } f \ A = 0$
unfolding *set-l2-def* **by** *simp*

lemma *set-l2-mono*:
 assumes $\bigwedge i. i \in K \implies f \ i \leq g \ i$
 assumes $\bigwedge i. i \in K \implies 0 \leq f \ i$
 shows $\text{set-l2 } f \ K \leq \text{set-l2 } g \ K$
unfolding *set-l2-def*
by (*simp add: setsum-nonneg setsum-mono power-mono prems*)

lemma *set-l2-right-distrib*:
 $0 \leq r \implies r * \text{set-l2 } f \ A = \text{set-l2 } (\lambda x. r * f \ x) \ A$
unfolding *set-l2-def*
apply (*simp add: power-mult-distrib*)
apply (*simp add: setsum-right-distrib [symmetric]*)
apply (*simp add: real-sqrt-mult setsum-nonneg*)
done

lemma *set-l2-left-distrib*:
 $0 \leq r \implies \text{set-l2 } f \ A * r = \text{set-l2 } (\lambda x. f \ x * r) \ A$
unfolding *set-l2-def*
apply (*simp add: power-mult-distrib*)
apply (*simp add: setsum-left-distrib [symmetric]*)
apply (*simp add: real-sqrt-mult setsum-nonneg*)
done

lemma *setsum-nonneg-eq-0-iff*:
 fixes $f :: 'a \Rightarrow 'b::\text{pordered-ab-group-add}$
 shows $\llbracket \text{finite } A; \forall x \in A. 0 \leq f \ x \rrbracket \implies \text{setsum } f \ A = 0 \iff (\forall x \in A. f \ x = 0)$
apply (*induct set: finite, simp*)
apply (*simp add: add-nonneg-eq-0-iff setsum-nonneg*)
done

lemma *set-l2-eq-0-iff*: $\text{finite } A \implies \text{set-l2 } f \ A = 0 \iff (\forall x \in A. f \ x = 0)$
unfolding *set-l2-def*
by (*simp add: setsum-nonneg setsum-nonneg-eq-0-iff*)

lemma *set-l2-triangle-ineq*:
 shows $\text{set-l2 } (\lambda i. f \ i + g \ i) \ A \leq \text{set-l2 } f \ A + \text{set-l2 } g \ A$
proof (*cases finite A*)
 case *False*
thus *?thesis* **by** *simp*

```

next
  case True
  thus ?thesis
  proof (induct set: finite)
    case empty
    show ?case by simp
  next
    case (insert x F)
    hence sqrt ((f x + g x)2 + (set-l2 (λi. f i + g i) F)2) ≤
      sqrt ((f x + g x)2 + (set-l2 f F + set-l2 g F)2)
    by (intro real-sqrt-le-mono add-left-mono power-mono insert
        set-l2-nonneg add-increasing zero-le-power2)
    also have
      ... ≤ sqrt ((f x)2 + (set-l2 f F)2) + sqrt ((g x)2 + (set-l2 g F)2)
    by (rule real-sqrt-sum-squares-triangle-ineq)
    finally show ?case
      using insert by simp
  qed
qed

```

```

lemma sqrt-sum-squares-le-sum:
  [|0 ≤ x; 0 ≤ y|] ⇒ sqrt (x2 + y2) ≤ x + y
  apply (rule power2-le-imp-le)
  apply (simp add: power2-sum)
  apply (simp add: mult-nonneg-nonneg)
  apply (simp add: add-nonneg-nonneg)
  done

```

```

lemma set-l2-le-setsum [rule-format]:
  (∀ i ∈ A. 0 ≤ f i) → set-l2 f A ≤ setsum f A
  apply (cases finite A)
  apply (induct set: finite)
  apply simp
  apply clarsimp
  apply (erule order-trans [OF sqrt-sum-squares-le-sum])
  apply simp
  apply simp
  apply simp
  done

```

```

lemma sqrt-sum-squares-le-sum-abs: sqrt (x2 + y2) ≤ |x| + |y|
  apply (rule power2-le-imp-le)
  apply (simp add: power2-sum)
  apply (simp add: mult-nonneg-nonneg)
  apply (simp add: add-nonneg-nonneg)
  done

```

```

lemma set-l2-le-setsum-abs: set-l2 f A ≤ (∑ i ∈ A. |f i|)
  apply (cases finite A)

```

```

apply (induct set: finite)
apply simp
apply simp
apply (rule order-trans [OF sqrt-sum-squares-le-sum-abs])
apply simp
apply simp
done

```

```

lemma set-l2-mult-ineq-lemma:
  fixes a b c d :: real
  shows  $2 * (a * c) * (b * d) \leq a^2 * d^2 + b^2 * c^2$ 
proof -
  have  $0 \leq (a * d - b * c)^2$  by simp
  also have  $\dots = a^2 * d^2 + b^2 * c^2 - 2 * (a * d) * (b * c)$ 
    by (simp only: power2-diff power-mult-distrib)
  also have  $\dots = a^2 * d^2 + b^2 * c^2 - 2 * (a * c) * (b * d)$ 
    by simp
  finally show  $2 * (a * c) * (b * d) \leq a^2 * d^2 + b^2 * c^2$ 
    by simp
qed

```

```

lemma set-l2-mult-ineq:  $(\sum i \in A. |f i| * |g i|) \leq \text{set-l2 } f A * \text{set-l2 } g A$ 
apply (cases finite A)
apply (induct set: finite)
apply simp
apply (rule power2-le-imp-le, simp)
apply (rule order-trans)
apply (rule power-mono)
apply (erule add-left-mono)
apply (simp add: add-nonneg-nonneg mult-nonneg-nonneg setsum-nonneg)
apply (simp add: power2-sum)
apply (simp add: power-mult-distrib)
apply (simp add: right-distrib left-distrib)
apply (rule ord-le-eq-trans)
apply (rule set-l2-mult-ineq-lemma)
apply simp
apply (intro mult-nonneg-nonneg set-l2-nonneg)
apply simp
done

```

```

lemma member-le-set-l2:  $\llbracket \text{finite } A; i \in A \rrbracket \implies f i \leq \text{set-l2 } f A$ 
apply (rule-tac s=insert i (A - {i}) and t=A in subst)
apply fast
apply (subst set-l2-insert)
apply simp
apply simp
apply simp
done

```

4.5 Vectors are a real normed vector space

instantiation *vec* :: (norm, finite) norm
begin

definition

norm-vec-def: $\text{norm } x = \text{set-l2 } (\lambda i. \text{norm } (\text{Rep-vec } x \ i)) \text{ UNIV}$

instance ..
end

lemma *vec-norm-ge-zero*:

fixes $x :: 'a::\text{real-normed-vector} \wedge 'n::\text{finite}$
shows $0 \leq \text{norm } x$

unfolding *norm-vec-def* **by** (rule *set-l2-nonneg*)

lemma *vec-norm-eq-zero*:

fixes $x :: 'a::\text{real-normed-vector} \wedge 'n::\text{finite}$
shows $(\text{norm } x = 0) = (x = 0)$

unfolding *norm-vec-def*

by (*simp add: set-l2-eq-0-iff expand-vec-eq*)

lemma *vec-norm-triangle-ineq*:

fixes $x \ y :: 'a::\text{real-normed-vector} \wedge 'n::\text{finite}$
shows $\text{norm } (x + y) \leq \text{norm } x + \text{norm } y$

unfolding *norm-vec-def*

apply (rule *order-trans [OF - set-l2-triangle-ineq]*)

apply (*simp add: set-l2-mono norm-triangle-ineq*)

done

lemma *vec-norm-scaleR*:

fixes $x :: 'a::\text{real-normed-vector} \wedge 'n::\text{finite}$
shows $\text{norm } (\text{scaleR } r \ x) = |r| * \text{norm } x$

unfolding *norm-vec-def*

by (*simp add: norm-scaleR set-l2-right-distrib*)

instantiation *vec* :: (real-normed-vector, finite) real-normed-vector
begin

definition

sgn-vec-def: $\text{sgn } (x::'a \wedge 'b) = \text{scaleR } (\text{inverse } (\text{norm } x)) \ x$

instance

apply (*intro-classes*)

apply (rule *sgn-vec-def*)

apply (rule *vec-norm-ge-zero*)

apply (rule *vec-norm-eq-zero*)

apply (rule *vec-norm-triangle-ineq*)

apply (rule *vec-norm-scaleR*)

done

end

lemma *norm-Rep-vec-le*: $\text{norm } (\text{Rep-vec } x \ i) \leq \text{norm } x$
unfolding *norm-vec-def*
by (*rule member-le-set-l2* [*OF finite*], *simp*)

interpretation *Rep-vec*: *bounded-linear* [$\lambda x. \text{Rep-vec } x \ i$]
apply (*unfold-locale*)
apply (*rule Rep-vec-add*)
apply (*rule Rep-vec-scaleR*)
apply (*rule-tac x=1 in exI*)
apply (*simp add: norm-Rep-vec-le*)
done

4.6 Vectors are an inner product space

instantiation *vec* :: (*inner*, *finite*) *inner*
begin

definition

inner-vec-def: $\text{inner } x \ y = (\sum_{i \in \text{UNIV}} \text{inner } (\text{Rep-vec } x \ i) \ (\text{Rep-vec } y \ i))$

instance ..

end

lemma *vec-inner-commute*:

fixes $x \ y :: 'a::\text{real-inner} \ ^n::\text{finite}$

shows $\text{inner } x \ y = \text{inner } y \ x$

unfolding *inner-vec-def* **by** (*simp add: inner-commute*)

lemma *vec-inner-left-distrib*:

fixes $x \ y \ z :: 'a::\text{real-inner} \ ^n::\text{finite}$

shows $\text{inner } (x + y) \ z = \text{inner } x \ z + \text{inner } y \ z$

unfolding *inner-vec-def*

by (*simp add: inner-left-distrib setsum-addf*)

lemma *vec-inner-scaleR-left*:

fixes $x \ y :: 'a::\text{real-inner} \ ^n::\text{finite}$

shows $\text{inner } (\text{scaleR } r \ x) \ y = r * \text{inner } x \ y$

unfolding *inner-vec-def*

by (*simp add: inner-scaleR-left setsum-right-distrib*)

lemma *vec-inner-ge-zero*:

fixes $x :: 'a::\text{real-inner} \ ^n::\text{finite}$

shows $0 \leq \text{inner } x \ x$

unfolding *inner-vec-def*

by (*simp add: setsum-nonneg inner-ge-zero*)

```

lemma vec-inner-eq-zero:
  fixes  $x :: 'a::\text{real-inner} \wedge 'n::\text{finite}$ 
  shows  $(\text{inner } x \ x = 0) = (x = 0)$ 
unfolding inner-vec-def
apply (simp add: setsum-nonneg-eq-0-iff finite inner-ge-zero)
apply (simp add: inner-eq-zero expand-vec-eq)
done

```

```

lemma vec-norm-eq-sqrt-inner:
  fixes  $x :: 'a::\text{real-inner} \wedge 'n::\text{finite}$ 
  shows  $\text{norm } x = \text{sqrt } (\text{inner } x \ x)$ 
unfolding norm-vec-def inner-vec-def
unfolding set-l2-def
by (simp add: norm-squared-eq-inner)

```

```

instance vec ::  $(\text{real-inner}, \text{finite}) \text{ real-inner}$ 
apply (intro-classes)
apply (rule vec-inner-commute)
apply (rule vec-inner-left-distrib)
apply (rule vec-inner-scaleR-left)
apply (rule vec-inner-ge-zero)
apply (rule vec-inner-eq-zero)
apply (rule vec-norm-eq-sqrt-inner)
done

```

4.7 Vector is a functor

definition

```

vec-map ::  $('a \Rightarrow 'b) \Rightarrow 'a \wedge 'n::\text{finite} \Rightarrow 'b \wedge 'n$  where
vec-map  $f \ x = \text{Abs-vec } (\lambda i. f (\text{Rep-vec } x \ i))$ 

```

```

lemma Rep-vec-vec-map [simp]:  $\text{Rep-vec } (\text{vec-map } f \ x) \ i = f (\text{Rep-vec } x \ i)$ 
unfolding vec-map-def by simp

```

```

lemma vec-map-id:  $\text{vec-map } \text{id} = \text{id}$ 
by (rule ext, rule vec-ext, simp)

```

```

lemma vec-map-o:  $\text{vec-map } (f \circ g) = \text{vec-map } f \circ \text{vec-map } g$ 
by (rule ext, rule vec-ext, simp)

```

```

lemma vec-map-ident [simp]:  $\text{vec-map } (\lambda a. a) = (\lambda x. x)$ 
by (rule ext, rule vec-ext, simp)

```

```

lemma vec-map-map:  $\text{vec-map } f (\text{vec-map } g \ x) = \text{vec-map } (\lambda a. f (g \ a)) \ x$ 
by (rule vec-ext, simp)

```

4.8 Vector dot product

definition

```

dot ::  $'a \wedge 'n \Rightarrow 'a \wedge 'n::\text{finite} \Rightarrow 'a::\text{semiring-0}$  where

```

$$\text{dot } x \ y = (\sum_{i \in \text{UNIV}} \text{Rep-vec } x \ i * \text{Rep-vec } y \ i)$$

lemma *dot-right-distrib*: $\text{dot } x \ (y + z) = \text{dot } x \ y + \text{dot } x \ z$
unfolding *dot-def* **by** (*simp add: right-distrib setsum-addf*)

lemma *dot-left-distrib*: $\text{dot } (x + y) \ z = \text{dot } x \ z + \text{dot } y \ z$
unfolding *dot-def* **by** (*simp add: left-distrib setsum-addf*)

lemma *dot-right-scaleR*:
fixes $x \ y :: 'a::\text{real-algebra} \wedge 'n::\text{finite}$
shows $\text{dot } x \ (\text{scaleR } r \ y) = \text{scaleR } r \ (\text{dot } x \ y)$
unfolding *dot-def* **by** (*simp add: scaleR-right.setsum*)

lemma *dot-left-scaleR*:
fixes $x \ y :: 'a::\text{real-algebra} \wedge 'n::\text{finite}$
shows $\text{dot } (\text{scaleR } r \ x) \ y = \text{scaleR } r \ (\text{dot } x \ y)$
unfolding *dot-def* **by** (*simp add: scaleR-right.setsum*)

lemma *norm-dot-ineq*:
fixes $x \ y :: 'a::\text{real-normed-algebra} \wedge 'n::\text{finite}$
shows $\text{norm } (\text{dot } x \ y) \leq \text{norm } x * \text{norm } y$
unfolding *norm-vec-def dot-def*
apply (*rule order-trans*)
apply (*rule norm-setsum*)
apply (*rule order-trans*)
apply (*rule setsum-mono*)
apply (*rule norm-mult-ineq*)
apply (*rule ord-eq-le-trans [OF - set-l2-mult-ineq]*)
apply *simp*
done

interpretation *dot*:
bounded-bilinear [$\lambda x \ y. \text{dot } x \ y :: 'a::\text{real-normed-algebra}$]
apply (*unfold-locales*)
apply (*rule dot-left-distrib*)
apply (*rule dot-right-distrib*)
apply (*rule dot-left-scaleR*)
apply (*rule dot-right-scaleR*)
apply (*rule-tac x=1 in exI*)
apply (*simp add: norm-dot-ineq*)
done

lemma *dot-right-zero*: $\text{dot } x \ 0 = 0$
unfolding *dot-def* **by** *simp*

lemma *dot-left-zero*: $\text{dot } 0 \ y = 0$
unfolding *dot-def* **by** *simp*

4.9 Single-component vectors

definition

$vec1 :: 'n::finite \Rightarrow 'a::zero \Rightarrow 'a \wedge 'n$ **where**
 $vec1\ j\ x = Abs\text{-}vec\ (\lambda i. \text{if } i = j \text{ then } x \text{ else } 0)$

lemma *Rep-vec-vec1* [simp]:

$Rep\text{-}vec\ (vec1\ j\ x)\ i = (\text{if } i = j \text{ then } x \text{ else } 0)$

unfolding *vec1-def* **by** *simp*

lemma *vec1-zero* [simp]: $vec1\ j\ 0 = 0$

by (rule *vec-ext*, *simp*)

lemma *vec1-add*:

fixes $x\ y :: 'a::comm\text{-}monoid\text{-}add$

shows $vec1\ j\ (x + y) = vec1\ j\ x + vec1\ j\ y$

by (rule *vec-ext*, *simp*)

lemma *vec1-diff*:

fixes $x\ y :: 'a::ab\text{-}group\text{-}add$

shows $vec1\ j\ (x - y) = vec1\ j\ x - vec1\ j\ y$

by (rule *vec-ext*, *simp*)

lemma *vec1-scaleR*:

fixes $x :: 'a::real\text{-}vector$

shows $vec1\ j\ (scaleR\ r\ x) = scaleR\ r\ (vec1\ j\ x)$

by (rule *vec-ext*, *simp*)

interpretation *vec1*: *additive* [$\lambda x::'a::ab\text{-}group\text{-}add. vec1\ j\ x$]

by *unfold-locales* (rule *vec1-add*)

lemma *norm-vec1* [simp]:

fixes $a :: 'a::real\text{-}normed\text{-}vector$

shows $norm\ (vec1\ j\ a) = norm\ a$

unfolding *norm-vec-def*

apply (rule-tac $s=insert\ j\ (UNIV - \{j\})$ **and** $t=UNIV$ **in** *subst*)

apply *fast*

apply (*subst set-l2-insert*)

apply (rule *finite*)

apply *simp*

apply (*simp add: set-l2-0'*)

done

interpretation *vec1*:

bounded-linear [$\lambda x::'a::real\text{-}normed\text{-}vector. vec1\ j\ x$]

apply (*unfold-locales*)

apply (rule *vec1-scaleR*)

apply (rule-tac $x=1$ **in** *exI*, *simp*)

done

lemma *Rep-vec-setsum*: $Rep\text{-vec } (setsum\ f\ A)\ i = (\sum_{x \in A}. Rep\text{-vec } (f\ x)\ i)$
by (*cases finite A, induct set: finite, simp-all*)

lemma *setsum-vec1*: $(\sum_{i \in UNIV}. vec1\ i\ (f\ i)) = Abs\text{-vec } f$
apply (*rule vec-ext, simp*)
apply (*simp add: Rep-vec-setsum*)
apply (*cut-tac x=i in UNIV-I*)
apply (*erule setsum-diff1' [OF finite, THEN ssubst]*)
apply *simp*
done

end

5 Square Matrices

theory *SquareMatrix*
imports *VectorType*
begin

typedef (**open**) (*'a, 'n*) *smatrix* = *UNIV* :: (*'n* \Rightarrow *'n* \Rightarrow *'a*) *set* ..

declare *Abs-smatrix-inverse* [*simplified, iff*]

lemma *expand-smatrix-eq*:
 $A = B \iff (\forall i\ j. Rep\text{-smatrix } A\ i\ j = Rep\text{-smatrix } B\ i\ j)$
by (*simp add: Rep-smatrix-inject [symmetric] expand-fun-eq*)

lemma *smatrix-ext*:
 $(\bigwedge i\ j. Rep\text{-smatrix } A\ i\ j = Rep\text{-smatrix } B\ i\ j) \implies A = B$
by (*simp add: expand-smatrix-eq*)

5.1 Matrix arithmetic

instantiation *smatrix* :: (*zero, type*) *zero*
begin

definition
zero-smatrix-def: $0 = Abs\text{-smatrix } (\lambda i\ j. 0)$

instance ..
end

instantiation *smatrix* :: (*plus, type*) *plus*
begin

definition
plus-smatrix-def:
 $A + B = Abs\text{-smatrix } (\lambda i\ j. Rep\text{-smatrix } A\ i\ j + Rep\text{-smatrix } B\ i\ j)$

instance ..
end

instantiation *smatrix* :: (*minus*, *type*) *minus*
begin

definition

minus-smatrix-def:

$$A - B = \text{Abs-smatrix } (\lambda i j. \text{Rep-smatrix } A \ i \ j - \text{Rep-smatrix } B \ i \ j)$$

instance ..
end

instantiation *smatrix* :: (*uminus*, *type*) *uminus*
begin

definition

uminus-smatrix-def:

$$- A = \text{Abs-smatrix } (\lambda i j. - \text{Rep-smatrix } A \ i \ j)$$

instance ..
end

lemma *Rep-smatrix-zero* [*simp*]: *Rep-smatrix* 0 *i j* = 0
unfolding *zero-smatrix-def* **by** *simp*

lemma *Rep-smatrix-add* [*simp*]:
Rep-smatrix (*A* + *B*) *i j* = *Rep-smatrix* *A* *i j* + *Rep-smatrix* *B* *i j*
unfolding *plus-smatrix-def* **by** *simp*

lemma *Rep-smatrix-diff* [*simp*]:
Rep-smatrix (*A* - *B*) *i j* = *Rep-smatrix* *A* *i j* - *Rep-smatrix* *B* *i j*
unfolding *minus-smatrix-def* **by** *simp*

lemma *Rep-smatrix-uminus* [*simp*]:
Rep-smatrix (- *A*) *i j* = - *Rep-smatrix* *A* *i j*
unfolding *uminus-smatrix-def* **by** *simp*

instance *smatrix* :: (*semigroup-add*, *type*) *semigroup-add*
by (*intro-classes*, *simp* *add*: *expand-smatrix-eq* *add-assoc*)

instance *smatrix* :: (*ab-semigroup-add*, *type*) *ab-semigroup-add*
by (*intro-classes*, *simp* *add*: *expand-smatrix-eq* *add-commute*)

instance *smatrix* :: (*comm-monoid-add*, *type*) *comm-monoid-add*
by (*intro-classes*, *simp* *add*: *expand-smatrix-eq*)

instance *smatrix* :: (*cancel-semigroup-add*, *type*) *cancel-semigroup-add*

by (*intro-classes, simp-all add: expand-smatrix-eq*)

instance *smatrix* :: (*cancel-ab-semigroup-add, type*) *cancel-ab-semigroup-add*
by (*intro-classes, simp-all add: expand-smatrix-eq*)

instance *smatrix* :: (*ab-group-add, type*) *ab-group-add*
by (*intro-classes, simp-all add: expand-smatrix-eq*)

5.2 Smatrix multiplication

instantiation *smatrix* :: (*semiring-0, finite*) *semiring-0*
begin

definition

times-smatrix-def:

$A * B = \text{Abs-smatrix}$

$(\lambda i j. \sum_{k \in \text{UNIV}} \text{Rep-smatrix } A \ i \ k * \text{Rep-smatrix } B \ k \ j)$

lemma *Rep-smatrix-mult* [*simp*]:

$\text{Rep-smatrix } (A * B) \ i \ j =$

$(\sum_{k \in \text{UNIV}} \text{Rep-smatrix } A \ i \ k * \text{Rep-smatrix } B \ k \ j)$

unfolding *times-smatrix-def* **by** *simp*

instance proof

fix *A B C* :: (*'a, 'b*) *smatrix*

show $0 * A = 0$

by (*rule smatrix-ext*) *simp*

show $A * 0 = 0$

by (*rule smatrix-ext*) *simp*

show $(A * B) * C = A * (B * C)$

apply (*rule smatrix-ext, simp*)

apply (*subst setsum-left-distrib*)

apply (*subst setsum-right-distrib*)

apply (*subst mult-assoc*)

apply (*rule setsum-commute*)

done

show $(A + B) * C = A * C + B * C$

by (*rule smatrix-ext*)

(*simp add: left-distrib setsum-addr*)

show $A * (B + C) = A * B + A * C$

by (*rule smatrix-ext*)

(*simp add: right-distrib setsum-addr*)

qed

end

instance *smatrix* :: (*semiring-0-cancel, finite*) *semiring-0-cancel* ..

instance *smatrix* :: (*ring, finite*) *ring* ..

instantiation *smatrix* :: (*semiring-1*, *finite*) *semiring-1*
begin

definition

one-smatrix-def:
 $1 = \text{Abs-smatrix } (\lambda i j. \text{ if } i = j \text{ then } 1 \text{ else } 0)$

lemma *Rep-smatrix-one* [*simp*]:
 $\text{Rep-smatrix } 1 \ i \ j = (\text{ if } i = j \text{ then } 1 \text{ else } 0)$
unfolding *one-smatrix-def* **by** *simp*

instance proof

fix *A* :: ('a, 'b) *smatrix*
show $1 * A = A$
 apply (*rule smatrix-ext*, *simp*)
 apply (*cut-tac x=i in UNIV-I*)
 apply (*erule setsum-diff1'* [*OF finite*, *THEN ssubst*])
 apply *simp*
 done
show $A * 1 = A$
 apply (*rule smatrix-ext*, *simp*)
 apply (*cut-tac x=j in UNIV-I*)
 apply (*erule setsum-diff1'* [*OF finite*, *THEN ssubst*])
 apply *simp*
 done
show $(0::('a, 'b) \text{ smatrix}) \neq 1$
 by (*simp add: expand-smatrix-eq*)

qed

end

instance *smatrix* :: (*semiring-1-cancel*, *finite*) *semiring-1-cancel* ..

instance *smatrix* :: (*ring-1*, *finite*) *ring-1* ..

5.3 Numeral syntax for matrices

instantiation *smatrix* :: (*ring-1*, *finite*) *number*
begin

definition

number-of-smatrix-def: (*number-of w::(-, -)smatrix*) = *of-int w*

instance ..

end

Unfortunately class *number-ring* requires commutativity of multiplication.

5.4 Matrices are a real vector space

instantiation *smatrix* :: (*scaleR*, *finite*) *scaleR*
begin

definition

scaleR-smatrix-def:
 $scaleR\ r\ A = Abs-smatrix\ (\lambda i\ j.\ scaleR\ r\ (Rep-smatrix\ A\ i\ j))$

instance ..
end

lemma *Rep-smatrix-scaleR* [*simp*]:

$Rep-smatrix\ (scaleR\ r\ A)\ i\ j = scaleR\ r\ (Rep-smatrix\ A\ i\ j)$
unfolding *scaleR-smatrix-def* **by** *simp*

instance *smatrix* :: (*real-vector*, *finite*) *real-vector*

apply (*intro-classes*)

apply (*simp-all add: expand-smatrix-eq*)

apply (*simp add: scaleR-right-distrib*)

apply (*simp add: scaleR-left-distrib*)

done

instance *smatrix* :: (*real-algebra*, *finite*) *real-algebra*

by *intro-classes (simp-all add: expand-smatrix-eq scaleR-right.setsum)*

instance *smatrix* :: (*real-algebra-1*, *finite*) *real-algebra-1* ..

5.5 Applying a matrix to a vector

definition

$mmult\ ::\ ('a::semiring-0,\ 'n::finite)\ smatrix\ \Rightarrow\ 'a\ ^\ 'n\ \Rightarrow\ 'a\ ^\ 'n$

where

$mmult\ A\ x = Abs-vec\ (\lambda i.\ \sum_{j \in UNIV} Rep-smatrix\ A\ i\ j * Rep-vec\ x\ j)$

lemma *Rep-vec-mmult* [*simp*]:

$Rep-vec\ (mmult\ A\ x)\ i = (\sum_{j \in UNIV} Rep-smatrix\ A\ i\ j * Rep-vec\ x\ j)$
unfolding *mmult-def* **by** *simp*

lemma *mmult-mmult*: $mmult\ A\ (mmult\ B\ x) = mmult\ (A * B)\ x$

apply (*rule vec-ext*)

apply (*simp add: setsum-right-distrib setsum-left-distrib mult-assoc*)

apply (*rule setsum-commute*)

done

lemma *mmult-right-distrib*: $mmult\ A\ (x + y) = mmult\ A\ x + mmult\ A\ y$

by (*rule vec-ext, simp add: right-distrib setsum-addr*)

lemma *mmult-left-distrib*: $mmult\ (A + B)\ x = mmult\ A\ x + mmult\ B\ x$

by (*rule vec-ext, simp add: left-distrib setsum-addr*)

interpretation *mmult-right*:
additive [$\lambda x. \text{mmult } A \ x :: 'a::\text{ring} \wedge 'n::\text{finite}$]
by *unfold-locales* (rule *mmult-right-distrib*)

interpretation *mmult-left*:
additive [$\lambda A. \text{mmult } A \ x :: 'a::\text{ring} \wedge 'n::\text{finite}$]
by *unfold-locales* (rule *mmult-left-distrib*)

declare *mmult-right.zero* [*simp*]
declare *mmult-left.zero* [*simp*]

lemma *mmult-vec1*: $\text{mmult } A \ (\text{vec1 } j \ a) = \text{Abs-vec } (\lambda i. \text{Rep-smatrix } A \ i \ j * a)$
apply (rule *vec-ext*, *simp*)
apply (*cut-tac* $x=j$ **in** *UNIV-I*)
apply (*erule* *setsum-diff1'* [*OF* *finite*, *THEN* *ssubst*])
apply *simp*
done

Class *semiring-1* is required, since class *semiring-0* would allow multiplication in the ring to always return zero.

lemma *smatrix-mmult-ext*:
fixes $A \ B :: ('a::\text{semiring-1}, 'n::\text{finite}) \text{smatrix}$
assumes *mmult-eq*: $\bigwedge x. \text{mmult } A \ x = \text{mmult } B \ x$
shows $A = B$
proof (rule *smatrix-ext*)
fix $i \ j$
have $\text{mmult } A \ (\text{vec1 } j \ 1) = \text{mmult } B \ (\text{vec1 } j \ 1)$
by (rule *mmult-eq*)
thus $\text{Rep-smatrix } A \ i \ j = \text{Rep-smatrix } B \ i \ j$
by (*simp* *add: mmult-vec1*, *simp* *add: expand-vec-eq*)
qed

lemma *expand-smatrix-eq-mmult*:
fixes $A \ B :: ('a::\text{semiring-1}, 'n::\text{finite}) \text{smatrix}$
shows $A = B \iff (\forall x. \text{mmult } A \ x = \text{mmult } B \ x)$
by (*auto* *intro!*: *smatrix-mmult-ext*)

lemma *mmult-right-scaleR*:
fixes $A :: ('a::\text{real-algebra}, 'n::\text{finite}) \text{smatrix}$
shows $\text{mmult } A \ (\text{scaleR } r \ x) = \text{scaleR } r \ (\text{mmult } A \ x)$
by (rule *vec-ext*, *simp* *add: scaleR-right.setsum*)

lemma *mmult-left-scaleR*:
fixes $A :: ('a::\text{real-algebra}, 'n::\text{finite}) \text{smatrix}$
shows $\text{mmult } (\text{scaleR } r \ A) \ x = \text{scaleR } r \ (\text{mmult } A \ x)$
by (rule *vec-ext*, *simp* *add: scaleR-right.setsum*)

lemma *mmult-1-left* [*simp*]:

```

fixes x :: 'a::semiring-1 ^ 'n::finite
shows mmult 1 x = x
apply (rule vec-ext)
apply simp
apply (cut-tac x=i in UNIV-I)
apply (erule setsum-diff1' [OF finite, THEN ssubst])
apply simp
done

```

lemma *bounded-mmult:*

```

fixes A :: ('a::real-normed-algebra, 'n::finite) smatrix
shows  $\exists K. \forall x. \text{norm} (\text{mmult } A \ x) \leq \text{norm } x * K$ 
proof (intro exI allI)
fix x
have  $\text{norm} (\text{mmult } A \ x) = \text{set-l2} (\lambda i. \text{norm} (\text{Rep-vec} (\text{mmult } A \ x) \ i)) \ \text{UNIV}$ 
unfolding norm-vec-def ..
also have  $\dots \leq (\sum_{i \in \text{UNIV}} |\text{norm} (\text{Rep-vec} (\text{mmult } A \ x) \ i)|)$ 
by (rule set-l2-le-setsum-abs)
also have  $\dots = (\sum_{i \in \text{UNIV}} \text{norm} (\sum_{j \in \text{UNIV}} \text{Rep-smatrix } A \ i \ j * \text{Rep-vec} \ x \ j))$ 
by simp
also have  $\dots \leq (\sum_{i \in \text{UNIV}} \sum_{j \in \text{UNIV}} \text{norm} (\text{Rep-smatrix } A \ i \ j) * \text{norm } x)$ 
by (intro setsum-mono order-trans [OF norm-setsum]
order-trans [OF norm-mult-ineq] mult-left-mono
norm-Rep-vec-le norm-ge-zero)
finally show  $\text{norm} (\text{mmult } A \ x) \leq \text{norm } x * (\sum_{i \in \text{UNIV}} \sum_{j \in \text{UNIV}} \text{norm} (\text{Rep-smatrix } A \ i \ j))$ 
by (simp add: setsum-right-distrib mult-commute)
qed

```

lemma *bounded-linear-mmult-right:*

```

fixes A :: ('a::real-normed-algebra, 'n::finite) smatrix
shows bounded-linear (mmult A)
apply (unfold-locales)
apply (rule mmult-right-scaleR)
apply (rule bounded-mmult)
done

```

interpretation *mmult-right:*

```

bounded-linear [mmult (A::('a::real-normed-algebra, 'n::finite) smatrix)]
by (rule bounded-linear-mmult-right)

```

5.6 Matrix norm

instantiation *smatrix* :: (real-normed-algebra, finite) norm
begin

definition

```

norm-smatrix-def:

```

$norm\ A = (THE\ r.\ isLub\ UNIV\ \{norm\ (mmult\ A\ x)\ |\ x.\ norm\ x \leq 1\}\ r)$

instance ..
end

lemma *reals-complete1*:

assumes *notempty-S*: $\exists x. x \in S$
and *exists-Ub*: $\exists y. isUb\ (UNIV::real\ set)\ S\ y$
shows $\exists!t. isLub\ (UNIV::real\ set)\ S\ t$
apply (*rule ex-ex1I*)
apply (*rule reals-complete [OF notempty-S exists-Ub]*)
apply (*erule (1) real-isLub-unique*)
done

lemma *isLub-smatrix-norm*:

$isLub\ UNIV\ \{norm\ (mmult\ A\ x)\ |\ x.\ norm\ x \leq 1\}\ (norm\ A)$
unfolding *norm-smatrix-def*
apply (*rule theI'*)
apply (*rule reals-complete1*)
apply (*rule-tac x=0 in exI, simp*)
apply (*rule-tac x=0 in exI, simp*)
apply (*cut-tac mmult-right.nonneg-bounded [of A]*)
apply (*erule exE, rule-tac x=K in exI*)
apply (*rule isUbI*)
apply (*rule settleI*)
apply *clarify*
apply (*drule spec*)
apply (*erule order-trans*)
apply (*rule ord-le-eq-trans*)
apply (*erule (1) mult-right-mono*)
apply *simp*
apply *simp*
done

lemma *smatrix-norm-geI*:

fixes *A* :: (*'a::real-normed-algebra, 'n::finite*) *smatrix*
shows $\exists x. y = norm\ (mmult\ A\ x) \wedge norm\ x \leq 1 \implies y \leq norm\ A$
by (*rule isLub-smatrix-norm [THEN isLubD2], simp*)

lemma *smatrix-norm-leI*:

fixes *A* :: (*'a::real-normed-algebra, 'n::finite*) *smatrix*
assumes $\bigwedge x. norm\ x \leq 1 \implies norm\ (mmult\ A\ x) \leq y$
shows $norm\ A \leq y$
apply (*rule isLub-smatrix-norm [THEN isLub-le-isUb]*)
apply (*rule isUbI [OF settleI]*)
apply (*clarify elim!: prems*)
apply *simp*
done

```

lemma norm-mmult-ineq: norm (mmult A x) ≤ norm A * norm x
apply (case-tac x = 0, simp)
apply (subgoal-tac norm (mmult A x) / norm x ≤ norm A)
apply (simp add: pos-divide-le-eq)
apply (rule smatrix-norm-geI)
apply (rule-tac x=scaleR (inverse (norm x)) x in exI)
apply (simp add: norm-scaleR)
apply (simp add: mmult-right-scaleR norm-scaleR)
apply (simp add: nonzero-divide-eq-eq)
done

```

```

lemma smatrix-norm-ge-zero:
  fixes A :: ('a::real-normed-algebra, 'n::finite) smatrix
  shows 0 ≤ norm A
apply (rule smatrix-norm-geI)
apply (rule-tac x=0 in exI, simp)
done

```

```

lemma smatrix-norm-eq-zero:
  fixes A :: ('a::real-normed-algebra-1, 'n::finite) smatrix
  shows norm A = 0 ↔ A = 0
proof (safe)
  show norm (0::('a, 'n) smatrix) = 0
    apply (rule order-antisym)
    apply (rule smatrix-norm-leI, simp)
    apply (rule smatrix-norm-ge-zero)
    done
  next
  assume A: norm A = 0
  have ∀ x. norm (mmult A x) ≤ norm A * norm x
    by (simp add: norm-mmult-ineq)
  with A have ∀ x. mmult A x = 0 by simp
  thus A = 0
    by (simp add: smatrix-mmult-ext)
qed

```

```

lemma smatrix-norm-triangle-ineq:
  fixes A B :: ('a::real-normed-algebra, 'n::finite) smatrix
  shows norm (A + B) ≤ norm A + norm B
apply (rule smatrix-norm-leI)
apply (simp only: mmult-left-distrib)
apply (rule order-trans [OF norm-triangle-ineq])
apply (rule add-mono)
apply (rule smatrix-norm-geI, fast)
apply (rule smatrix-norm-geI, fast)
done

```

```

lemma smatrix-norm-scaleR:
  fixes A :: ('a::real-normed-algebra, 'n::finite) smatrix

```

```

  shows  $\text{norm} (\text{scaleR } r \ A) = |r| * \text{norm } A$ 
  apply (rule order-antisym)
  apply (rule smatrix-norm-leI)
  apply (simp add: mmult-left-scaleR norm-scaleR)
  apply (rule mult-left-mono [OF - abs-ge-zero])
  apply (rule smatrix-norm-geI, fast)
  apply (case-tac  $r = 0$ , simp add: smatrix-norm-ge-zero)
  apply (subgoal-tac  $\text{norm } A \leq \text{norm} (\text{scaleR } r \ A) / |r|$ )
  apply (simp add: pos-le-divide-eq mult-commute)
  apply (rule smatrix-norm-leI)
  apply (simp add: pos-le-divide-eq mult-commute)
  apply (rule smatrix-norm-geI)
  apply (rule-tac  $x=x$  in  $exI$ )
  apply (simp add: mmult-left-scaleR norm-scaleR)
done

```

```

lemma smatrix-norm-mult-ineq:
  fixes  $A \ B :: ('a::\text{real-normed-algebra}, 'n::\text{finite}) \text{matrix}$ 
  shows  $\text{norm} (A * B) \leq \text{norm } A * \text{norm } B$ 
  apply (rule smatrix-norm-leI)
  apply (simp only: mmult-mmult [symmetric])
  apply (rule order-trans [OF norm-mmult-ineq])
  apply (rule mult-left-mono)
  apply (rule smatrix-norm-geI, fast)
  apply (rule smatrix-norm-ge-zero)
done

```

```

lemma smatrix-norm-1:
   $\text{norm} (1::('a::\text{real-normed-algebra-1}, 'n::\text{finite}) \text{matrix}) = 1$ 
  apply (rule order-antisym)
  apply (rule smatrix-norm-leI)
  apply simp
  apply (rule smatrix-norm-geI)
  apply simp
  apply (rule-tac  $x=\text{vec1 } i \ 1$  in  $exI$ , simp)
done

```

```

instantiation smatrix :: ( $\text{real-normed-algebra-1}$ ,  $\text{finite}$ )  $\text{real-normed-algebra-1}$ 
begin

```

definition

```

   $\text{sgn-smatrix-def}: \text{sgn} (x::(-, -)\text{matrix}) = \text{scaleR} (\text{inverse} (\text{norm } x)) \ x$ 

```

instance

```

  apply intro-classes
  apply (rule sgn-smatrix-def)
  apply (rule smatrix-norm-ge-zero)
  apply (rule smatrix-norm-eq-zero)
  apply (rule smatrix-norm-triangle-ineq)

```

```

apply (rule smatrix-norm-scaleR)
apply (rule smatrix-norm-mult-ineq)
apply (rule smatrix-norm-1)
done

end

```

5.7 Vector outer product

definition

$outer :: 'a \hat{ } 'n \Rightarrow 'a \hat{ } 'n \Rightarrow ('a::semiring, 'n::finite) smatrix$ **where**
 $outer\ x\ y = Abs-smatrix\ (\lambda i\ j. Rep-vec\ x\ i * Rep-vec\ y\ j)$

lemma *Rep-smatrix-outer* [simp]:

$Rep-smatrix\ (outer\ x\ y)\ i\ j = Rep-vec\ x\ i * Rep-vec\ y\ j$

unfolding *outer-def* **by** *simp*

lemma *outer-right-distrib*: $outer\ x\ (y + z) = outer\ x\ y + outer\ x\ z$
by (rule *smatrix-ext*, *simp add: right-distrib*)

lemma *outer-left-distrib*: $outer\ (x + y)\ z = outer\ x\ z + outer\ y\ z$
by (rule *smatrix-ext*, *simp add: left-distrib*)

interpretation *outer-right*:

additive [$\lambda y. outer\ x\ y :: ('a::ring, 'n::finite) smatrix$]

by *unfold-locales* (rule *outer-right-distrib*)

interpretation *outer-left*:

additive [$\lambda x. outer\ x\ y :: ('a::ring, 'n::finite) smatrix$]

by *unfold-locales* (rule *outer-left-distrib*)

lemma *outer-right-scaleR*:

fixes $x\ y :: 'a::real-algebra \hat{ } 'n::finite$

shows $outer\ x\ (scaleR\ r\ y) = scaleR\ r\ (outer\ x\ y)$

by (rule *smatrix-ext*, *simp*)

lemma *outer-left-scaleR*:

fixes $x\ y :: 'a::real-algebra \hat{ } 'n::finite$

shows $outer\ (scaleR\ r\ x)\ y = scaleR\ r\ (outer\ x\ y)$

by (rule *smatrix-ext*, *simp*)

lemma *mmult-outer*: $mmult\ (outer\ x\ y)\ z = vec-map\ (\lambda a. a * dot\ y\ z)\ x$

unfolding *dot-def*

apply (rule *vec-ext*)

apply (*simp add: mult-assoc*)

apply (*simp add: setsum-right-distrib [symmetric]*)

done

lemma *norm-mmult-outer-ineq*:

```

fixes  $x y z :: 'a::\text{real-normed-algebra} \wedge 'n::\text{finite}$ 
shows  $\text{norm} (\text{mmult} (\text{outer } x y) z) \leq \text{norm } x * \text{norm } y * \text{norm } z$ 
apply (subst mmult-outer)
apply (subst norm-vec-def)
apply simp
apply (rule order-trans)
apply (rule set-l2-mono)
apply (rule norm-mult-ineq)
apply (rule norm-ge-zero)
apply (subst set-l2-left-distrib [symmetric])
apply (rule norm-ge-zero)
apply (subst norm-vec-def [symmetric])
apply (subst mult-assoc)
apply (rule mult-left-mono)
apply (rule norm-dot-ineq)
apply (rule norm-ge-zero)
done

```

```

lemma norm-outer-ineq:
fixes  $x y z :: 'a::\text{real-normed-algebra} \wedge 'n::\text{finite}$ 
shows  $\text{norm} (\text{outer } x y) \leq \text{norm } x * \text{norm } y$ 
apply (rule smatrix-norm-leI)
apply (rule order-trans)
apply (rule norm-mmult-outer-ineq)
apply (rule order-trans)
apply (rule mult-left-mono)
apply (simp add: mult-nonneg-nonneg)
apply simp
done

```

```

interpretation outer: bounded-bilinear [outer]
apply (unfold-locales)
apply (rule outer-left-distrib)
apply (rule outer-right-distrib)
apply (rule outer-left-scaleR)
apply (rule outer-right-scaleR)
apply (rule-tac x=1 in exI)
apply (simp add: norm-outer-ineq)
done

```

end

6 Gradient Derivatives

```

theory GradientDeriv
imports InnerProduct FrechetDeriv
begin

```

6.1 Gradient Derivative

definition

gderiv ::
 [*'a*::*real-inner* \Rightarrow *real*, *'a*, *'a*] \Rightarrow *bool*
 ((*GDERIV* (-)/ (-)/ \Rightarrow (-)) [1000, 1000, 60] 60) **where**
GDERIV *f x* \Rightarrow *D* = *FDERIV* *f x* \Rightarrow (λh . *inner* *h D*)

lemma *deriv-fderiv*: *DERIV* *f x* \Rightarrow *D* = *FDERIV* *f x* \Rightarrow (λh . *h * D*)
by (*simp* *only*: *deriv-def* *field-fderiv-def*)

lemma *gderiv-deriv* [*simp*]: *GDERIV* *f x* \Rightarrow *D* = *DERIV* *f x* \Rightarrow *D*
by (*simp* *only*: *gderiv-def* *deriv-fderiv* *inner-real-def*)

lemma *GDERIV-DERIV-compose*:

\llbracket *GDERIV* *f x* \Rightarrow *D*; *DERIV* *g (f x)* \Rightarrow *E* \rrbracket
 \implies *GDERIV* (λx . *g (f x)*) *x* \Rightarrow *scaleR E D*

unfolding *gderiv-def* *deriv-fderiv*

apply (*drule* (1) *FDERIV-compose*)

apply (*simp* *add*: *inner-scaleR-right* *mult-ac*)

done

lemma *FDERIV-subst*: \llbracket *FDERIV* *f x* \Rightarrow *D*; *D* = *E* $\rrbracket \implies$ *FDERIV* *f x* \Rightarrow *E*
by *simp*

lemma *GDERIV-subst*: \llbracket *GDERIV* *f x* \Rightarrow *D*; *D* = *E* $\rrbracket \implies$ *GDERIV* *f x* \Rightarrow *E*
by *simp*

lemma *GDERIV-const*: *GDERIV* (λx . *k*) *x* \Rightarrow 0

unfolding *gderiv-def* *inner-right.zero* **by** (*rule* *FDERIV-const*)

lemma *GDERIV-add*:

\llbracket *GDERIV* *f x* \Rightarrow *D*; *GDERIV* *g x* \Rightarrow *E* \rrbracket
 \implies *GDERIV* (λx . *f x* + *g x*) *x* \Rightarrow *D* + *E*

unfolding *gderiv-def* *inner-right.add* **by** (*rule* *FDERIV-add*)

lemma *GDERIV-minus*:

GDERIV *f x* \Rightarrow *D* \implies *GDERIV* (λx . - *f x*) *x* \Rightarrow - *D*

unfolding *gderiv-def* *inner-right.minus* **by** (*rule* *FDERIV-minus*)

lemma *GDERIV-diff*:

\llbracket *GDERIV* *f x* \Rightarrow *D*; *GDERIV* *g x* \Rightarrow *E* \rrbracket
 \implies *GDERIV* (λx . *f x* - *g x*) *x* \Rightarrow *D* - *E*

unfolding *gderiv-def* *inner-right.diff* **by** (*rule* *FDERIV-diff*)

lemma *GDERIV-scaleR*:

\llbracket *DERIV* *f x* \Rightarrow *D*; *GDERIV* *g x* \Rightarrow *E* \rrbracket
 \implies *GDERIV* (λx . *scaleR* (*f x*) (*g x*)) *x*
 \Rightarrow (*scaleR* (*f x*) *E* + *scaleR* *D* (*g x*))

unfolding *gderiv-def* *deriv-fderiv* *inner-right.add* *inner-right.scaleR*

```

apply (rule FDERIV-subst)
apply (erule (1) scaleR.FDERIV)
apply (simp add: mult-ac)
done

lemma GDERIV-mult:
  [[GDERIV f x := D; GDERIV g x := E]]
  ==> GDERIV (λx. f x * g x) x := scaleR (f x) E + scaleR (g x) D
unfolding gderiv-def
apply (rule FDERIV-subst)
apply (erule (1) FDERIV-mult)
apply (simp add: inner-distrib inner-scaleR mult-ac)
done

lemma GDERIV-inverse:
  [[GDERIV f x := D; f x ≠ 0]]
  ==> GDERIV (λx. inverse (f x)) x := - (inverse (f x))2 *R D
apply (erule GDERIV-DERIV-compose)
apply (erule DERIV-inverse [folded numeral-2-eq-2])
done

lemma GDERIV-norm: x ≠ 0 ==> GDERIV (λx. norm x) x := sgn x
apply (rule GDERIV-subst)
apply (subst norm-eq-sqrt-inner)
apply (rule GDERIV-DERIV-compose [where g=sqrt and D=scaleR 2 x])
apply (subst gderiv-def)
apply (rule FDERIV-subst)
apply (rule inner.FDERIV [OF FDERIV-ident FDERIV-ident])
apply (simp add: expand-fun-eq inner-scaleR inner-commute)
apply (rule DERIV-real-sqrt)
apply (simp add: zero-less-inner-iff)
apply (subst norm-eq-sqrt-inner [symmetric])
apply (simp add: sgn-div-norm)
done

lemmas FDERIV-norm = GDERIV-norm [unfolded gderiv-def]

lemma isCont-sgn: x ≠ 0 ==> isCont (λx. sgn x) x
unfolding sgn-div-norm
by (simp add: isCont-scaleR isCont-inverse isCont-norm)

end

```

7 Pairs as Vector Spaces

```

theory PairVector
imports GradientDeriv
begin

```

7.1 Vector arithmetic

instantiation * :: (*zero*, *zero*) *zero*
begin

definition

zero-prod-def: $0 = (0, 0)$

instance ..
end

instantiation * :: (*plus*, *plus*) *plus*
begin

definition

plus-prod-def: $A + B = (fst\ A + fst\ B, snd\ A + snd\ B)$

instance ..
end

instantiation * :: (*minus*, *minus*) *minus*
begin

definition

minus-prod-def: $A - B = (fst\ A - fst\ B, snd\ A - snd\ B)$

instance ..
end

instantiation * :: (*uminus*, *uminus*) *uminus*
begin

definition

uminus-prod-def: $- A = (-\ fst\ A, -\ snd\ A)$

instance ..
end

instantiation * :: (*scaleR*, *scaleR*) *scaleR*
begin

definition

scaleR-prod-def: $scaleR\ r\ A = (scaleR\ r\ (fst\ A), scaleR\ r\ (snd\ A))$

instance ..
end

instantiation * :: (*norm*, *norm*) *norm*
begin

definition

norm-prod-def: $\text{norm } A = \text{sqrt } ((\text{norm } (\text{fst } A))^2 + (\text{norm } (\text{snd } A))^2)$

instance ..

end

instantiation * :: (*inner*, *inner*) *inner*

begin

definition

inner-prod-def: $\text{inner } A \ B = \text{inner } (\text{fst } A) (\text{fst } B) + \text{inner } (\text{snd } A) (\text{snd } B)$

instance ..

end

instantiation * :: (*sgn-div-norm*, *sgn-div-norm*) *sgn-div-norm*

begin

definition

sgn-prod-def: $\text{sgn } (x::'a \times 'b) = \text{scaleR } (\text{inverse } (\text{norm } x)) \ x$

instance

by *default* (*rule* *sgn-prod-def*)

end

lemma *fst-zero* [*simp*]: $\text{fst } 0 = 0$

unfolding *zero-prod-def* **by** *simp*

lemma *snd-zero* [*simp*]: $\text{snd } 0 = 0$

unfolding *zero-prod-def* **by** *simp*

lemma *fst-add* [*simp*]: $\text{fst } (A + B) = \text{fst } A + \text{fst } B$

unfolding *plus-prod-def* **by** *simp*

lemma *snd-add* [*simp*]: $\text{snd } (A + B) = \text{snd } A + \text{snd } B$

unfolding *plus-prod-def* **by** *simp*

lemma *add-Pair* [*simp*]: $(a, b) + (c, d) = (a + c, b + d)$

unfolding *plus-prod-def* **by** *simp*

lemma *fst-diff* [*simp*]: $\text{fst } (A - B) = \text{fst } A - \text{fst } B$

unfolding *minus-prod-def* **by** *simp*

lemma *snd-diff* [*simp*]: $\text{snd } (A - B) = \text{snd } A - \text{snd } B$

unfolding *minus-prod-def* **by** *simp*

lemma *diff-Pair* [*simp*]: $(a, b) - (c, d) = (a - c, b - d)$

unfolding *minus-prod-def* **by** *simp*

lemma *fst-uminus* [*simp*]: $\text{fst } (- A) = - \text{fst } A$
unfolding *uminus-prod-def* **by** *simp*

lemma *snd-uminus* [*simp*]: $\text{snd } (- A) = - \text{snd } A$
unfolding *uminus-prod-def* **by** *simp*

lemma *uminus-Pair* [*simp*]: $-(a, b) = (- a, - b)$
unfolding *uminus-prod-def* **by** *simp*

lemma *fst-scaleR* [*simp*]: $\text{fst } (\text{scaleR } r A) = \text{scaleR } r (\text{fst } A)$
unfolding *scaleR-prod-def* **by** *simp*

lemma *snd-scaleR* [*simp*]: $\text{snd } (\text{scaleR } r A) = \text{scaleR } r (\text{snd } A)$
unfolding *scaleR-prod-def* **by** *simp*

lemma *scaleR-Pair* [*simp*]: $\text{scaleR } r (a, b) = (\text{scaleR } r a, \text{scaleR } r b)$
unfolding *scaleR-prod-def* **by** *simp*

lemma *norm-Pair* [*simp*]: $\text{norm } (a, b) = \text{sqrt } ((\text{norm } a)^2 + (\text{norm } b)^2)$
unfolding *norm-prod-def* **by** *simp*

lemma *inner-Pair* [*simp*]: $\text{inner } (a, b) (c, d) = \text{inner } a c + \text{inner } b d$
unfolding *inner-prod-def* **by** *simp*

lemmas *expand-prod-eq = Pair-fst-snd-eq*

instance * :: (*semigroup-add*, *semigroup-add*) *semigroup-add*
by (*intro-classes*, *simp add: expand-prod-eq add-assoc*)

instance * :: (*ab-semigroup-add*, *ab-semigroup-add*) *ab-semigroup-add*
by (*intro-classes*, *simp add: expand-prod-eq add-commute*)

instance * :: (*comm-monoid-add*, *comm-monoid-add*) *comm-monoid-add*
by (*intro-classes*, *simp add: expand-prod-eq*)

instance * :: (*cancel-semigroup-add*, *cancel-semigroup-add*) *cancel-semigroup-add*
by (*intro-classes*, *simp-all add: expand-prod-eq*)

instance * :: (*cancel-ab-semigroup-add*, *cancel-ab-semigroup-add*) *cancel-ab-semigroup-add*
by (*intro-classes*, *simp add: expand-prod-eq*)

instance * :: (*ab-group-add*, *ab-group-add*) *ab-group-add*
by (*intro-classes*, *simp-all add: expand-prod-eq*)

instance * :: (*real-vector*, *real-vector*) *real-vector*
apply (*intro-classes*)
apply (*simp-all add: expand-prod-eq*)
apply (*simp add: scaleR-right-distrib*)

```

apply (simp add: scaleR-left-distrib)
done

```

7.2 Product is a normed vector space

```

instance * :: (real-normed-vector, real-normed-vector) real-normed-vector
proof

```

```

  fix r :: real
  fix A B :: 'a::real-normed-vector × 'b::real-normed-vector
  show 0 ≤ norm A
    unfolding norm-prod-def by simp
  show (norm A = 0) = (A = 0)
    by (induct A) (simp add: expand-prod-eq)
  show norm (A + B) ≤ norm A + norm B
    unfolding norm-prod-def
    apply (rule order-trans [OF - real-sqrt-sum-squares-triangle-ineq])
    apply (simp add: add-mono power-mono norm-triangle-ineq)
    done
  show norm (scaleR r A) = |r| * norm A
    unfolding norm-prod-def
    apply (simp add: norm-scaleR power-mult-distrib)
    apply (simp add: right-distrib [symmetric])
    apply (simp add: real-sqrt-mult-distrib)
    done

```

qed

```

interpretation fst: bounded-linear [fst]

```

```

  apply (unfold-locales)
  apply (rule fst-add)
  apply (rule fst-scaleR)
  apply (rule-tac x=1 in exI, simp)
  done

```

```

interpretation snd: bounded-linear [snd]

```

```

  apply (unfold-locales)
  apply (rule snd-add)
  apply (rule snd-scaleR)
  apply (rule-tac x=1 in exI, simp)
  done

```

lemma LIMSEQ-Pair:

```

   $\llbracket X \text{ ----} > a; Y \text{ ----} > b \rrbracket \implies (\lambda n. (X\ n, Y\ n)) \text{ ----} > (a, b)$ 
  apply (rule LIMSEQ-I)
  apply (subgoal-tac 0 < r / sqrt 2)
  apply (drule-tac r=r / sqrt 2 in LIMSEQ-D, safe)
  apply (drule-tac r=r / sqrt 2 in LIMSEQ-D, safe)
  apply (rename-tac M N, rule-tac x=max M N in exI, safe)
  apply (simp add: real-sqrt-sum-squares-less)
  apply (simp add: divide-pos-pos)

```

done

lemma *Cauchy-Pair*:

```
[[Cauchy X; Cauchy Y]] ==> Cauchy (λn. (X n, Y n))
apply (rule CauchyI)
apply (subgoal-tac 0 < e / sqrt 2)
apply (drule-tac e=e / sqrt 2 in CauchyD, safe)
apply (drule-tac e=e / sqrt 2 in CauchyD, safe)
apply (rename-tac M N, rule-tac x=max M N in exI, safe)
apply (simp add: real-sqrt-sum-squares-less)
apply (simp add: divide-pos-pos)
done
```

instance * :: (banach, banach) banach

proof

```
fix X :: nat => 'a × 'b
assume X: Cauchy X
have 1: (λn. fst (X n)) -----> lim (λn. fst (X n))
  using fst.Cauchy [OF X]
  by (simp add: Cauchy-convergent-iff convergent-LIMSEQ-iff)
have 2: (λn. snd (X n)) -----> lim (λn. snd (X n))
  using snd.Cauchy [OF X]
  by (simp add: Cauchy-convergent-iff convergent-LIMSEQ-iff)
have X -----> (lim (λn. fst (X n)), lim (λn. snd (X n)))
  using LIMSEQ-Pair [OF 1 2] by simp
thus convergent X
  by (rule convergentI)
```

qed

7.3 Product is an inner product space

instance * :: (real-inner, real-inner) real-inner

proof

```
fix r :: real
fix A B C :: 'a::real-inner * 'b::real-inner
show inner A B = inner B A
  unfolding inner-prod-def
  by (simp add: inner-commute)
show inner (A + B) C = inner A C + inner B C
  unfolding inner-prod-def
  by (simp add: inner-left-distrib)
show inner (scaleR r A) B = r * inner A B
  unfolding inner-prod-def
  by (simp add: inner-scaleR-left right-distrib)
show 0 ≤ inner A A
  unfolding inner-prod-def
  by (intro add-nonneg-nonneg inner-ge-zero)
show (inner A A = 0) = (A = 0)
  unfolding inner-prod-def
```

```

    apply (simp add: add-nonneg-eq-0-iff inner-ge-zero)
    apply (simp add: inner-eq-zero expand-prod-eq)
  done
show norm A = sqrt (inner A A)
  unfolding norm-prod-def inner-prod-def
  by (simp add: norm-squared-eq-inner)
qed

end

```

8 Cross Products

```

theory CrossProduct
imports PairVector
begin

```

```

types 'a triple = 'a * 'a * 'a

```

```

types R3 = real * real * real

```

```

lemma norm ((2,3,6)::R3) = 7
by (simp add: real-sqrt-unique)

```

```

lemma inner ((1,2,3)::R3) (4,5,6) = ?x
by simp

```

definition

```

cross :: real triple ⇒ real triple ⇒ real triple where
cross = (λ(a, b, c). λ(x, y, z).
  (b*z - c*y, c*x - a*z, a*y - b*x))

```

lemma cross-Pair:

```

cross (a, b, c) (x, y, z) = (b*z - c*y, c*x - a*z, a*y - b*x)
unfolding cross-def by simp

```

```

lemma left-cross-distrib: cross (A + B) C = cross A C + cross B C
by (cases A, cases B, cases C, simp add: cross-Pair left-distrib)

```

```

lemma right-cross-distrib: cross A (B + C) = cross A B + cross A C
by (cases A, cases B, cases C, simp add: cross-Pair right-distrib)

```

lemma cross-scaleR-left:

```

cross (scaleR r A) B = scaleR r (cross A B)
by (cases A, cases B, simp add: cross-Pair right-diff-distrib)

```

lemma cross-scaleR-right:

```

cross A (scaleR r B) = scaleR r (cross A B)
by (cases A, cases B, simp add: cross-Pair right-diff-distrib)

```

```

lemma Lagrange-identity:
  (norm A)2 * (norm B)2 = (norm (inner A B))2 + (norm (cross A B))2
  apply (simp add: norm-squared-eq-inner)
  apply (cases A, cases B, rename-tac a b c x y z)
  apply (simp add: power2-eq-square)
  apply (simp add: cross-Pair)
  apply (simp add: left-diff-distrib right-diff-distrib)
  apply (simp add: ring-simps)
  done

lemma norm-cross-ineq: norm (cross A B) ≤ norm A * norm B
  apply (rule power2-le-imp-le)
  prefer 2 apply (simp add: mult-nonneg-nonneg real-sqrt-ge-zero add-nonneg-nonneg)
  apply (simp add: power-mult-distrib)
  apply (simp only: Lagrange-identity)
  apply simp
  done

interpretation bounded-bilinear-cross: bounded-bilinear [cross]
  apply (unfold-locales)
  apply (rule left-cross-distrib)
  apply (rule right-cross-distrib)
  apply (rule cross-scaleR-left)
  apply (rule cross-scaleR-right)
  apply (rule-tac x=1 in exI)
  apply (simp add: norm-cross-ineq)
  done

lemma cross-anticommutate: cross A B = - cross B A
  apply (cases A, cases B, rename-tac a b c x y z)
  apply (simp add: cross-Pair)
  done

end

```

9 Quaternions

```

theory Quaternion
imports InnerProduct
begin

```

9.1 Definition

```

datatype quaternion = Quaternion complex complex

```

9.2 Addition and Subtraction

instantiation *quaternion* :: *ab-group-add*
begin

definition

zero-quaternion-def: $0 = \text{Quaternion } 0 \ 0$

fun *plus-quaternion* **where**

quaternion-plus:

$\text{Quaternion } a \ b + \text{Quaternion } c \ d = \text{Quaternion } (a + c) \ (b + d)$

fun *minus-quaternion* **where**

quaternion-minus:

$\text{Quaternion } a \ b - \text{Quaternion } c \ d = \text{Quaternion } (a - c) \ (b - d)$

fun *uminus-quaternion* **where**

quaternion-uminus:

$- \text{Quaternion } a \ b = \text{Quaternion } (- a) \ (- b)$

instance proof

fix *a b c* :: *quaternion*

show $(a + b) + c = a + (b + c)$

by (*induct a, induct b, induct c, simp*)

show $a + b = b + a$

by (*induct a, induct b, simp*)

show $0 + a = a$

unfolding *zero-quaternion-def*

by (*induct a, simp*)

show $- a + a = 0$

unfolding *zero-quaternion-def*

by (*induct a, simp*)

show $a - b = a + - b$

by (*induct a, induct b, simp*)

qed

end

lemma *quaternion-eq-zero* [*simp*]:

$\text{Quaternion } a \ b = 0 \longleftrightarrow a = 0 \wedge b = 0$

by (*simp add: zero-quaternion-def*)

9.3 Multiplication

lemma *complex-cnj-scaleR*: $\text{cnj } (\text{scaleR } r \ x) = \text{scaleR } r \ (\text{cnj } x)$

by (*simp add: scaleR-conv-of-real complex-cnj-mult*)

lemmas *complex-cnj-simps* =

complex-cnj-add

complex-cnj-diff

complex-cnj-minus
complex-cnj-mult
complex-cnj-divide
complex-cnj-inverse
complex-cnj-scaleR

lemmas *ring-distrib* =
left-distrib
right-distrib
left-diff-distrib
right-diff-distrib

instantiation *quaternion* :: *ring-1*
begin

fun *times-quaternion* **where**
quaternion-times:
*Quaternion a b * Quaternion c d =*
*Quaternion (a * c - d * cnj b) (cnj a * d + c * b)*

definition
one-quaternion-def: $1 = \text{Quaternion } 1 \ 0$

instance proof
fix *x y z* :: *quaternion*
show $(x * y) * z = x * (y * z)$
by (*induct x, induct y, induct z,*
simp add: complex-cnj-simps ring-distrib)
show $1 * x = x$
unfolding *one-quaternion-def* **by** (*induct x, simp*)
show $x * 1 = x$
unfolding *one-quaternion-def* **by** (*induct x, simp*)
show $(x + y) * z = x * z + y * z$
by (*induct x, induct y, induct z,*
simp add: complex-cnj-simps ring-distrib)
show $x * (y + z) = x * y + x * z$
by (*induct x, induct y, induct z,*
simp add: complex-cnj-simps ring-distrib)
show $(0::\text{quaternion}) \neq 1$
unfolding *one-quaternion-def zero-quaternion-def* **by** *simp*
qed
end

9.4 Vector space

lemmas *scaleR-distrib* =
scaleR-left-distrib
scaleR-right-distrib

scaleR-left-diff-distrib
scaleR-right-diff-distrib

instantiation *quaternion* :: *real-algebra-1*
begin

fun *scaleR-quaternion* **where**
quaternion-scaleR:
 $\text{scaleR } r \text{ (Quaternion } a \text{ b)} = \text{Quaternion (scaleR } r \text{ a) (scaleR } r \text{ b)}$

instance proof

fix $r \ s :: \text{real}$ **and** $x \ y :: \text{quaternion}$
show $\text{scaleR } r \ (x + y) = \text{scaleR } r \ x + \text{scaleR } r \ y$
by (*induct x, induct y, simp add: scaleR-right-distrib*)
show $\text{scaleR } (r + s) \ x = \text{scaleR } r \ x + \text{scaleR } s \ x$
by (*induct x, induct y, simp add: scaleR-left-distrib*)
show $\text{scaleR } r \ (\text{scaleR } s \ x) = \text{scaleR } (r * s) \ x$
by (*induct x, induct y, simp*)
show $\text{scaleR } 1 \ x = x$
by (*induct x, simp*)
show $\text{scaleR } r \ x * y = \text{scaleR } r \ (x * y)$
by (*induct x, induct y, simp add: complex-cnj-simps scaleR-distrib*)
show $x * \text{scaleR } r \ y = \text{scaleR } r \ (x * y)$
by (*induct x, induct y, simp add: complex-cnj-simps scaleR-distrib*)
qed

end

lemma *quaternion-of-real-eq*:
 $\text{of-real } r = \text{Quaternion (of-real } r) \ 0$
unfolding *of-real-def one-quaternion-def* **by** *simp*

9.5 Norm and inner product

instantiation *quaternion* :: *real-inner*
begin

fun *inner-quaternion* **where**
quaternion-inner:
 $\text{inner (Quaternion } a \text{ b) (Quaternion } c \text{ d)} = \text{inner } a \ c + \text{inner } b \ d$

fun *norm-quaternion* **where**
quaternion-norm:
 $\text{norm (Quaternion } a \text{ b)} = \text{sqrt ((norm } a)^2 + (\text{norm } b)^2)$

definition

sgn-quaternion-def:
 $\text{sgn } (x :: \text{quaternion}) = \text{scaleR (inverse (norm } x)) \ x$

```

instance proof
  fix  $r :: \text{real}$  and  $x\ y\ z :: \text{quaternion}$ 
  show  $\text{sgn } x = \text{scaleR } (\text{inverse } (\text{norm } x))\ x$ 
    by (rule sgn-quaternion-def)
  show  $\text{inner } x\ y = \text{inner } y\ x$ 
    by (induct x, induct y)
      (simp add: inner-commute)
  show  $\text{inner } (x + y)\ z = \text{inner } x\ z + \text{inner } y\ z$ 
    by (induct x, induct y, induct z)
      (simp add: inner-left-distrib)
  show  $\text{inner } (\text{scaleR } r\ x)\ y = r * \text{inner } x\ y$ 
    by (induct x, induct y)
      (simp add: inner-scaleR-left right-distrib)
  show  $0 \leq \text{inner } x\ x$ 
    by (induct x)
      (simp add: add-nonneg-nonneg inner-ge-zero)
  show  $\text{inner } x\ x = 0 \iff x = 0$ 
    by (induct x)
      (simp add: add-nonneg-eq-0-iff inner-ge-zero inner-eq-zero)
  show  $\text{norm } x = \text{sqrt } (\text{inner } x\ x)$ 
    by (induct x)
      (simp add: norm-eq-sqrt-inner inner-ge-zero)
qed

end

```

9.6 Conjugate

```

fun
   $qcnj :: \text{quaternion} \Rightarrow \text{quaternion}$ 
where
  quaternion-cnj:
     $qcnj\ (\text{Quaternion } a\ b) = \text{Quaternion } (\text{cnj } a)\ (-\ b)$ 

lemma complex-cnj-mult:  $\text{cnj } x * x = \text{of-real } ((\text{norm } x)^2)$ 
unfolding complex-of-real-def
by (induct x, simp add: power2-eq-square [symmetric])

lemma complex-mult-cnj:  $x * \text{cnj } x = \text{of-real } ((\text{norm } x)^2)$ 
unfolding complex-of-real-def
by (induct x, simp add: power2-eq-square [symmetric])

lemma quaternion-cnj-mult:  $qcnj\ x * x = \text{of-real } ((\text{norm } x)^2)$ 
unfolding quaternion-of-real-eq
by (induct x, simp add: complex-cnj-simps complex-cnj-mult complex-mult-cnj)

lemma quaternion-mult-cnj:  $x * qcnj\ x = \text{of-real } ((\text{norm } x)^2)$ 
unfolding quaternion-of-real-eq
by (induct x, simp add: complex-cnj-simps complex-cnj-mult complex-mult-cnj)

```

9.7 Inverse

instantiation *quaternion* :: *division-ring*
begin

definition

inverse-quaternion-def:
 $inverse\ x = scaleR\ (inverse\ ((norm\ x)^2))\ (qcnj\ x)$

instance proof

fix *x* :: *quaternion*
assume $x \neq 0$
thus $inverse\ x * x = 1$
 unfolding *inverse-quaternion-def*
 by (*simp add: quaternion-cnj-mult of-real-def*)

next

fix *x* :: *quaternion*
assume $x \neq 0$
thus $x * inverse\ x = 1$
 unfolding *inverse-quaternion-def*
 by (*simp add: quaternion-mult-cnj of-real-def*)

qed

end

instance *quaternion* :: *real-normed-div-algebra*

proof

fix *x y* :: *quaternion*
show $norm\ (x * y) = norm\ x * norm\ y$
 apply (*induct x, induct y, simp*)
 apply (*rename-tac a b c d*)
 apply (*simp add: real-sqrt-mult [symmetric]*)
 apply (*simp add: norm-squared-eq-inner*)
 apply (*simp add: inner-diff inner-distrib*)
 apply (*simp add: ring-distrib*)
 apply (*simp add: norm-squared-eq-inner [symmetric]*)
 apply (*simp only: norm-mult power-mult-distrib*)
 apply (*simp add: inner-commute*)
 apply (*simp add: inner-complex-def*)
 apply (*simp add: ring-simps*)
 done

qed

end