Formalization in Linear Algebra

Clea Bergsman, Katherine Buesing, Sahan Wijetunga, Mentor: George McNinch

VERSEIM REU

06/26/2025

What is a Bilinear Form?

Definition: a **bilinear form** is a map $\beta : V \times W \to K$, where V and W are K-vector spaces and K is a field, when

1. $\beta(v_1 + v_2, w) = \beta(v_1, w) + \beta(v_2, w)$
2. $\beta(v, w_1 + w_2) = \beta(v, w_1) + \beta(v, w_2)$
3. $\beta(\lambda v, w) = \beta(v, \lambda w) = \lambda\beta(v, w)$

hold for all $v \in V$, $w \in W$, and $\lambda \in K$.

Special Properties

A bilinear form β is **symmetric** if

1. $\beta(v, w) = \beta(w, v) \ \forall \ v, w$

A bilinear form β is **anti-symmetric** or **alternating** if

1. $\beta(v, v) = 0, \ \forall \ v$
2. $\beta(v, w) = -\beta(w, v), \ \forall \ v, w$

Otherwise, a bilinear form β is called **nonsymmetric**.

A Lemma on Anti-Symmetric Bilinear Forms

Lemma: If $\beta(v, v) = 0, \forall\ v$, then $\beta(v, w) = -\beta(w, v), \forall\ v, w$.
Proof:

- Let $\beta(v + w, v + w) = 0$
- $= \beta(v + w, v) + \beta(v + w, w)$
- $= \beta(v, v) + \beta(w, v) + \beta(v, w) + \beta(w, w)$
- Since $\beta(v, v) = 0$, we have $\beta(w, v) + \beta(v, w) = 0$
- Thus, $\beta(v, w) = -\beta(w, v)$

```
def Alt (β:V →l[k] V →l[k] k) : Prop :=
  ∀ v : V, β v v = 0

def Skew (β:V →l[k] V →⍰lk] k) : Prop :=
  ∀ v w : V, β v w = −β w v
```

```
lemma skew_of_alt (β:V →l[k] V →l[k] k) (ha : Alt β) :
  Skew β := by
  intro v w
  have h0 : β (v+w) = β v + β w := by simp
  have h : β (v+w) (v+w)
  = (β v) v + (β w) v + (β v) w + (β w) w :=
    calc
    (β (v+w)) (v+w) = (β v) (v+w) + (β w) (v+w) :=
    by rw [LinearMap.BilinForm.add_left]
    _ = (β v) v + (β w) v + (β v) w + (β w) w :=
    by rw [LinearMap.BilinForm.add_right v v w,
    LinearMap.BilinForm.add_right w v w, ← add_assoc]; ring
  have hv : β v v = 0 := by apply ha
  have hw : β w w = 0 := by apply ha
  have hvw : β (v+w) (v+w) = 0 := by apply ha
  rw [hv, hw, hvw, zero_add, add_zero] at h
  have h1 : (β v) w = −(β w) v := by
  exact Eq.symm (LinearMap.BilinForm.IsAlt.neg_eq ha w v)
  exact h1
```

Orthogonal Complement

The following is a definition of the orthogonal complement

It also proves each of the following characteristics of the orthogonal complement

- add_mem'
- zero_mem'
- smul_mem'

This definition and proof was written by Clea Bergsman

```
def OrthogComplement {k V: Type} [AddCommGroup V]
[Field k] [Module k V] (S : Set V)
{β:V →l [k] V →l [k] k} : Subspace k V where
  carrier := { v | ∀ (x:S), β v x = 0 }
  add_mem' := by
    simp
    intro h1 h2 h3 h4
    exact fun a b ↦ Linarith.eq_of_eq_of_eq
    (h3 a b) (h4 a b)
  zero_mem' := by
    simp
  smul_mem' := by
    simp
    intro h1 h2 h3 h4 h5
    right
    apply h3
    exact h5
```

Unique Representation of a Direct Sum

The following is a proof of the fact that if you have two disjoint subspaces W_1 and W_2 four vectors, $x_1, y_1 \in W_1$ and $x_2, y_2 \in W_2$, then $x_1 + x_2 = y_1 + y_2$ implies $x_1 = y_1$ and $x_2 = y_2$

This proof was written by Sahan Wijetunga

```
theorem direct_sum_unique_repr (k V : Type)
  [Field k] [AddCommGroup V] [Module k V]
  (W₁ W₂ : Submodule k V) (h_int : ⊥ = W₁ ⊓ W₂)
  (x₁ x₂ y₁ y₂ : V) (h₁ : x₁ ∈ W₁ ∧ y₁ ∈ W₁)
  (h₂ : x₂ ∈ W₂ ∧ y₂ ∈ W₂ ) :
 x₁ + x₂ = y₁ + y₂ → x₁ = y₁ ∧ x₂ = y₂ := by
   have <hx1, hy1> := h₁
   have <hx2, hy2> := h₂
   intro h
   have h': x₁-y₁ = y₂-x₂ :=
     calc
       x₁-y₁ = (x₁+x₂-x₂)-y₁ := by abel_nf
       _ = (y₁+y₂-x₂)-y₁ := by rw[h]
       _ = y₂-x₂ := by abel
   have hw1: (x₁-y₁) ∈ W₁ := by
     exact
      (Submodule.sub_mem_iff_left W₁ hy1).mpr
```

```
    hx1
  have hw2: (x₁-y₁) ∈ W₂ := by
    have: y₂-x₂ ∈ W₂ := by
      exact
      (Submodule.sub_mem_iff_left W₂ hx2).mpr
      hy2
    rw[h']
    assumption
  have hw: (x₁-y₁) ∈ (W₁: Set V) ∩ W₂ := by
    exact Set.mem_inter hw1 hw2
  have hw0: x₁-y₁ = 0 := by
    have: (W₁: Set V) ∩ W₂ = {(0: V)} := calc
      (W₁: Set V) ∩ W₂ = W₁ ⊓ W₂ := rfl
      _ = (⊥: Submodule k V) := by
        exact congrArg SetLike.coe
        (id (Eq.symm h_int))
      _ = ({0}: Set V) := rfl
    have: (x₁-y₁) ∈ ({0}: Set V) := by
      rw[<- this]
      assumption
    exact this
  have hxy1: x₁=y₁ := by
    calc
      _ = (x₁-y₁)+y₁ := by abel
      _ = 0+y₁ := by rw[hw0]
      _ = y₁ := by abel
  have hxy2: x₂=y₂ := by
    calc
      x₂ = x₂+ 0 := by abel
      _ = x₂+(x₁-y₁) := by rw[hw0]
      _ = x₂+(y₂-x₂) := by rw[h']
      _ = y₂ := by abel
  exact <hxy1, hxy2>
```

Disjoint Union of Functions

This is an (incomplete) proof that the disjoint union of two linearly independent bases b_1 and b_2 are linearly independent

This (incomplete) proof was written by Katherine Buesing

```
theorem lin_indep_by_transverse_subspaces
    (k V : Type) [Field k] [AddCommGroup V]
    [Module k V] (I₁ I₂ : Type)
    [Fintype I₁] [Fintype I₂]
    (b₁ : I₁ → V) (b₂ : I₂ → V)
    (b1_indep : LinearIndependent k b₁)
    (b2_indep : LinearIndependent k b₂)
    (W₁ W₂ : Submodule k V)
    (h_int : W₁ ⊓ W₂ = ⊥)
    (hbw1 : ∀ i, b₁ i ∈ W₁)
    (hbw2 : ∀ i, b₂ i ∈ W₂)
    [DecidableEq I₁] [DecidableEq I₂]
    : LinearIndependent k
    (disjointUnion_funs b₁ b₂) := by
     rw[linearIndependent_iff'']
     intro s a g h₁ h₂
     have k₀ :
```

```
  ∑ i ∈ s, a i • disjointUnion_funs b₁ b₂ i
  = ∑ i : (I₁ ⊕ I₂), a i •
  disjointUnion_funs b₁ b₂ i := by
    sorry
  have k₁ : ∑ i, (a (Sum.inl i)) • (b₁ i) =
  - ∑ j, (a (Sum.inr j)) • (b₂ j)  := by
    rw[k₀] at h₁
    simp at h₁
    sorry
  have k₂ : ∑ i, (a (Sum.inl i)) • (b₁ i)
  ∈ W₁ ⊓ W₂ := by
    simp
    have k₂₀ : ∑ i, (a (Sum.inl i)) • (b₁ i)
    ∈ W₁ := by
      exact Submodule.sum_smul_mem W₁
      (fun i ↦ a (Sum.inl i)) fun c a ↦ hbw1 c
    have k₂₁ : ∑ i, (a (Sum.inl i)) • (b₁ i)
```

```
      ∈ W₂ := by
        rw[k₁]
        apply Submodule.neg_mem
        exact Submodule.sum_smul_mem W₂
        (fun i ↦ a (Sum.inr i)) fun c a ↦ hbw2 c
      constructor
      · exact k₂₀
      · exact k₂₁
    have k₃ : - ∑ j, (a  (Sum.inr j)) • (b₂ j)
    ∈ W₁ ⊓ W₂ := by
      rw[k₁] at k₂
      exact k₂
    rw[linearIndependent_iff] at b1_indep
    rw[linearIndependent_iff] at b2_indep
    rw[h_int] at k₂
    rw[h_int] at k₃
    simp at k₂
    simp at k₃
    sorry
```

Some (More) Unsolved Proofs

- The linear independence of orthogonal bases
- The disjoint union of sets Fin n and Fin m is equal to Fin n + m
- And more properties of alternating and symmetric bilinear forms

Definite Form

Definition: a **positive definite** **(symmetric bilinear) form** is a map $\beta : V \times V \to \mathbb{R}$ with

1. $\beta(v_1 + v_2, w) = \beta(v_1, w) + \beta(v_2, w),\ \forall v_1, v_2, w \in V$
2. $\beta(\lambda v, w) = \lambda\beta(v, w),\ \forall \lambda \in \mathbb{R},\ \forall v, w \in V$
3. $\beta(v, w) = \beta(w, v),\ \forall v, w \in V$
4. $\beta(v, v) > 0,\ \forall v \neq 0$

Orthogonal

1. We say $(v_1, \dots, v_n)$ are *orthogonal* if $\beta(v_i, v_j) = 0$ for $i \neq j$.
2. We say $(v_1, \dots, v_n)$ are *orthonormal* if they are orthogonal and $\beta(v_i, v_i) = 1$ for all i.

Gram-Schmidt

1. Let V be a vector space over $\mathbb{R}$ with a positive definite form β and basis $(v_1, \ldots, v_n)$.
2. The Gram-Schmidt (orthonormalization) algorithm returns an *orthonormal* basis $(u_1, \ldots, u_n)$.

Gram-Schmidt Algorithm

1. Let $(v_1, \ldots, v_n)$ linearly independent. We define $(u_1, \ldots, u_n)$ (inductively) by

$$\begin{aligned} u_1 &:= v_1, \\ u_2 &:= v_2 - \operatorname{proj}_{u_1}(v_2), \\ u_3 &:= v_2 - \operatorname{proj}_{u_1}(v_3) - \operatorname{proj}_{u_2}(v_3), \\ &\vdots \\ u_n &:= v_n - \sum_{j=1}^{n-1} \operatorname{proj}_{u_j}(v_n). \end{aligned}$$

where $\operatorname{proj}_u(v) = \frac{\beta(v,u)}{\beta(u,u)} \cdot u$. These $(u_1, \ldots, u_n)$ are orthogonal.

Gram-Schmidt Algorithm

1. Let $(v_1, \ldots, v_n)$ linearly independent. We define $(u_1, \ldots, u_n)$ (inductively) by

$$\begin{aligned} u_1 &:= v_1, \\ u_2 &:= v_2 - \mathrm{proj}_{u_1}(v_2), \\ u_3 &:= v_2 - \mathrm{proj}_{u_1}(v_3) - \mathrm{proj}_{u_2}(v_3), \\ &\vdots \\ u_n &:= v_n - \sum_{j=1}^{n-1} \mathrm{proj}_{u_j}(v_n). \end{aligned}$$

where $\mathrm{proj}_u(v) = \frac{\beta(v,u)}{\beta(u,u)} \cdot u$. Note defining $v_i := \frac{1}{\sqrt{\beta(u_i,u_i)}}$ we have $\beta(v_i, v_i) = 1$. The $(v_1, \ldots, v_n)$ are orthonormal.

Calculation

Recall $v_1 = u_1,\ u_k = v_k - \sum_{i=1}^{k-1} \text{proj}_{u_i}(v_k)$. We have for $j < k$ that

$$\begin{aligned}
\beta\left(u_j, u_k\right) &= \beta\left(u_j, v_k - \sum_{i=1}^{n-1} \text{proj}_{u_j}(v_k)\right) \\
&= \beta(u_j, v_k) - \sum_{i=1}^{n-1} \beta(u_j, \text{proj}_{u_i}(v_k)) \\
&= \beta(u_j, v_k) - \sum_{i=1}^{n-1} \frac{\beta(u_i, u_k)}{\beta(u_i, u_i)} \beta(u_j, u_i) \\
&= \beta(u_j, v_k) - \beta(u_j, v_k) \\
&= 0
\end{aligned}$$

As β is symmetric, $\beta(u_k, u_j) = 0$ as well.

Lean Proof Structure

1. The proof took 478 lines in lean! I separated it into 15 lemmas, 10 theorems, and $\sim$15 definitions.

```
structure OrthogBasis'{V:Type}[AddCommGroup V]
  [Module ℝ V] (β:V → V → ℝ) (hp: Def β)
  (hs: Symm β) (n:ℕ) where

  basis : Basis (Fin n) ℝ V
  is_orthog : OrthogPred β basis
```

```
structure OrthogBasis'{V:Type}[AddCommGroup V]
  [Module ℝ V] (β:V → V → ℝ) (hp: Def β)
  (hs: Symm β) (n:ℕ) where

  basis : Basis (Fin n) ℝ V
  is_orthog : OrthogPred β basis

def orthog_basis {V:Type} [AddCommGroup V]
[Module ℝ V] (β:V → V → ℝ) (hp: Def β)
(hs : Symm β) {n:ℕ} (b: Basis (Fin n) ℝ V):
OrthogBasis' β hp hs n where

    basis := ...
    is_orthog :=
        orthog_span_gram_schmidt β hp hs b
```

```
theorem orthog_span_gram_schmidt {V:Type} [AddCommGroup V] [Module ℝ V]
  (β:V → V → ℝ) (hp : Def β) (hs : Symm β)
  {n:ℕ} (b:Fin n → V)
  : OrthogPred β (gram_schmidt β hp hs b):=
  match n with
  | Nat.zero => by
    intro i j h
    have: ↑i<(0: ℕ) := by exact i.isLt
    linarith
  | Nat.succ n => by
    let c := gram_schmidt β hp hs b
    let c' := gram_schmidt β hp hs (restrict b)
    let x := b (Fin.last n)
    have h: c = intermediate β c' x := rfl
    exact orthog_intermediate β hp hs c'
      (orthog_span_gram_schmidt β hp hs (restrict b)) x
```

```
theorem orthog_span_gram_schmidt {V:Type} [AddCommGroup V] [Module ℝ V]
  (β:V → V → ℝ) (hp : Def β) (hs : Symm β)
  {n:ℕ} (b:Fin n → V)
  : OrthogPred β (gram_schmidt β hp hs b):=
  match n with
  | Nat.zero => by
    intro i j h
    have: ↑i<(0: ℕ) := by exact i.isLt
    linarith
  | Nat.succ n => by
    let c := gram_schmidt β hp hs b
    let c' := gram_schmidt β hp hs (restrict b)
    let x := b (Fin.last n)
    have h: c = intermediate β c' x := rfl
    exact orthog_intermediate β hp hs c'
      (orthog_span_gram_schmidt β hp hs (restrict b)) x
theorem orthog_intermediate {V:Type} [AddCommGroup V] [Module ℝ V]
  (β:V → V →ℝ) (hp : Def β) (hs : Symm β)
  {n:ℕ} (b:Fin n → V) (hb: OrthogPred β b) (x: V):
    OrthogPred β (intermediate β b x) := by
    have case1 (...) (h₁: i=Fin.last n): (β (...) (...)) = 0 := by
      ...
      rw[this]
      exact orthog_sub_perp_align β hp hs b hb x (j.castPred h₂)
    have case2 (...): ... = 0 := by
      ...
    intro i j h
    by_cases h₁:(i=Fin.last n)
    . exact case1 i j h h₁
    by_cases h₂:(j=Fin.last n)
    . rw[hs]
      exact case1 j i h.symm h₂
    . exact case2 i j h h₁ h₂
```

```
@[simp]
def restrict {X:Type} {m:ℕ} (f:Fin (m+1) → X)
    : Fin m → X := fun i => f i.castSucc
```

```
@[simp]
def restrict {X:Type} {m:ℕ} (f:Fin (m+1) → X)
    : Fin m → X := fun i => f i.castSucc
theorem restrict_set_eq {X: Type} {m: ℕ}
    (f: Fin (m+1) → X) :
    f '' (Set.range (@Fin.castSucc m))
    = Set.range (restrict f)
  := by ...
```

```
@[simp]
def restrict {X:Type} {m:ℕ} (f:Fin (m+1) → X)
    : Fin m → X := fun i => f i.castSucc
theorem restrict_set_eq {X: Type} {m: ℕ}
    (f: Fin (m+1) → X) :
    f '' (Set.range (@Fin.castSucc m))
    = Set.range (restrict f)
  := by ...

lemma linear_independence_mem (...)
    (hb: LinearIndependent ℝ b) :
  ¬ (b (Fin.last n))∈
    Submodule.span ℝ (Set.range (restrict b))

    := by
    rw[<- restrict_set_eq]
    ...
```

References

1. Avigad, J. Buzzard, K. Lewis R. Y. Massot, P. (2020). *Mathematics in Lean*.
2. Liesen, J. Mehrmann, V. (2015). *Linear Algebra*.

Thank you!

Special thanks to Dr. George McNinch and the REU