Twelfth Lesson

Formalising a LUP Decomposition

Motivation

Take a classic algorithm

Cormen LUP algorithm

How difficult to formalise?

Test our linear algebra (`matrix.v`)

Linear equations

$$\begin{array}{rcrcrcr} x & + & 2y & + & 3z & = & 5 \\ 2x & - & 4y & + & 6z & = & 18 \\ 3x & - & 9y & - & 3z & = & 6 \end{array}$$

Linear equations

$$\begin{bmatrix} 1 & 2 & 3 \\ 2 & -4 & 6 \\ 3 & -9 & -3 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} 5 \\ 18 \\ 6 \end{bmatrix}$$

$$\begin{bmatrix} 1 & 2 & 3 \\ 2 & -4 & 6 \\ 3 & -9 & -3 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 2 & -8 & 0 \\ 3 & -15 & -12 \end{bmatrix} \begin{bmatrix} 1 & 2 & 3 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Linear equations

$$\begin{bmatrix}1 & 0 & 0\\ 2 & -8 & 0\\ 3 & -15 & -12\end{bmatrix}\begin{bmatrix}1 & 2 & 3\\ 0 & 1 & 0\\ 0 & 0 & 1\end{bmatrix}\begin{bmatrix}x\\ y\\ z\end{bmatrix} = \begin{bmatrix}5\\ 18\\ 6\end{bmatrix}$$

$$\begin{bmatrix}1 & 0 & 0\\ 2 & -8 & 0\\ 3 & -15 & -12\end{bmatrix}\begin{bmatrix}x_1\\ y_1\\ z_1\end{bmatrix} = \begin{bmatrix}5\\ 18\\ 6\end{bmatrix} \qquad \begin{bmatrix}1 & 2 & 3\\ 0 & 1 & 0\\ 0 & 0 & 1\end{bmatrix}\begin{bmatrix}x\\ y\\ z\end{bmatrix} = \begin{bmatrix}x_1\\ y_1\\ z_1\end{bmatrix}$$

$$x_1 = 5, \quad y_1 = -1, \quad z_1 = 2 \qquad\qquad z = 2, \quad y = -1, \quad x = 1$$

Linear equations

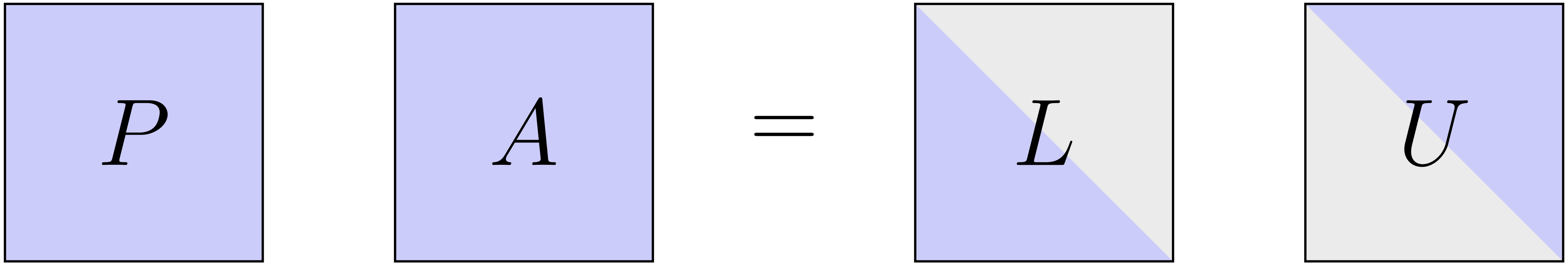
$$\begin{bmatrix} 1 & 2 & 3 \\ 2 & -4 & 6 \\ 3 & -9 & -3 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 2 & -8 & 0 \\ 3 & -15 & -12 \end{bmatrix} \begin{bmatrix} 1 & 2 & 3 \\ 0 & 1 & 0 \\ 3 & 0 & 1 \end{bmatrix}$$

$$P \quad A = L \quad U$$

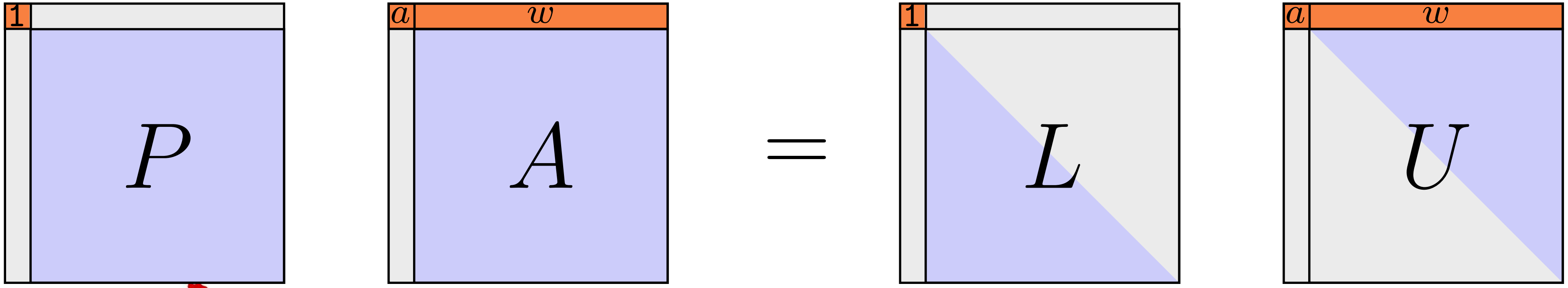
$$P = \begin{array}{cc} & \begin{array}{ccc} & j & \end{array} \\ i & \begin{bmatrix} & & \\ & 1 & \\ & & \end{bmatrix} \end{array}$$

Algorithm

Easy case: recursive construction

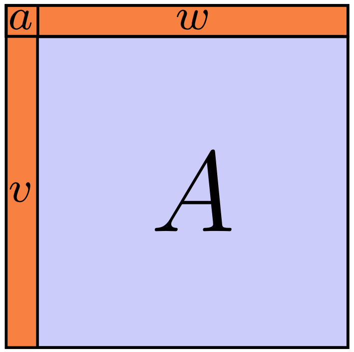


Add to A one line and a column of zeros:

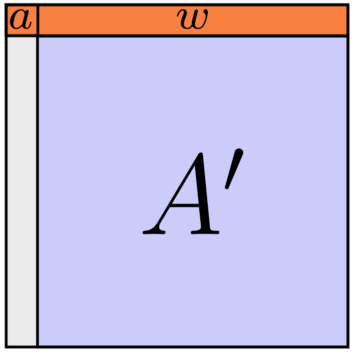


Algorithm

Main case ($a \neq 0$)

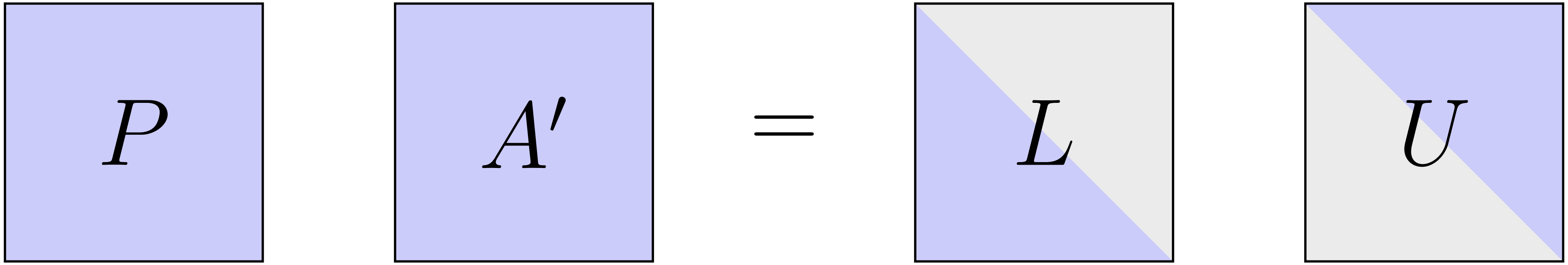


Schur complement $A' = A - 1/a(v * w)$

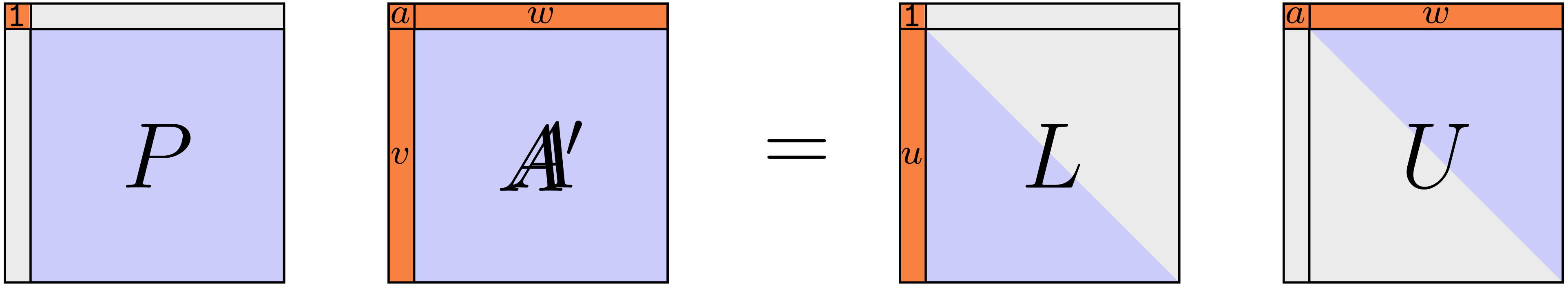


Algorithm

Main case: recursive construction ($A' = A - 1/a(v * w)$)

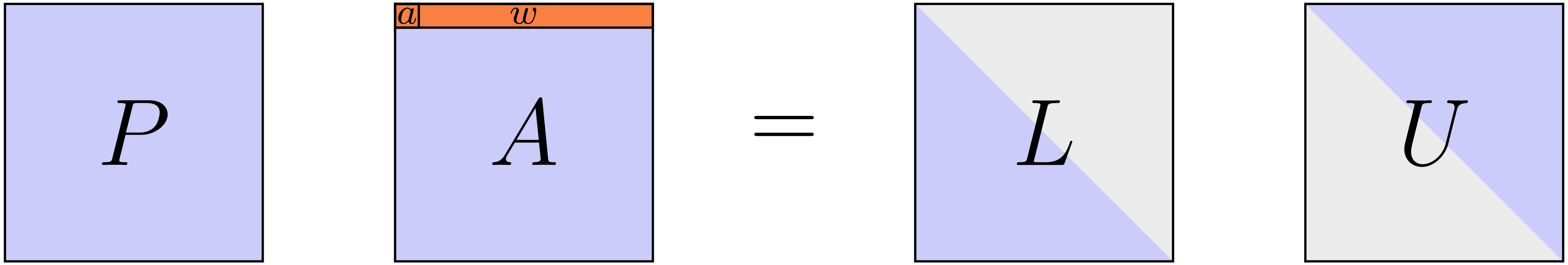


$u = 1/a(P * v)$

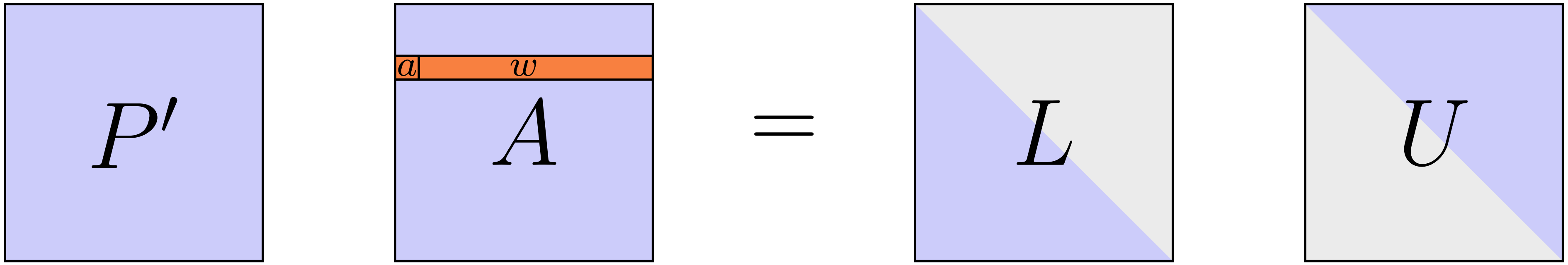


Algorithm

Special case



Line permutation



Algorithm

1. Put a non-zero element at (0,0) if needed by line permutation

2. Perform the recursive call on the Schur complement

3. Recompose the result of the recursive call

Formalisation

Write the algorithm

Write the specification

Prove that the specification is met

What is a matrix?

```
Inductive matrix (m n :  nat) R :=
  Matrix of {ffun 'I_m * 'I_n → R}.

Notation "'M[R]_(m,n)" := (matrix m n R).
Notation "'M[R]_(n)" := 'M[R]_(n,n).

Definition A : 'M[R]_(m,n) :=
    \matrix(i,j) E.
```

$\mathcal{A}\ i\ j\ :\ R$

Getting blocks

$$\left[\begin{array}{c|c} A_{ul} & A_{ur} \\ \hline A_{dl} & A_{dr} \end{array}\right]$$

Using ordinal :

```
lshift n (i : 'I_m) : 'I_(m + n)
rshift m (i : 'I_n) : 'I_(m + n)
split (i : 'I_(m + n)) : 'I_m + 'I_n
```

where

```
Inductive A + B := inl of A | inr of B.
```

Row block

$$[A_l|A_r]$$

```
Definition row_mx A_l A_r : 'M[R]_(m, n_1 + n_2) :=
  \matrix_(i, j) match split j with
     inl j_1 ⇒ A_l i j_1 | inr j_2 ⇒ A_r i j_2
   end.

Definition lsubmx (A :  'M[R]_(m, n_1 + n_2)) :=
   \matrix_(i, j) A i (lshift n_2 j).

Definition rsubmx (A :  'M[R]_(m, n_1 + n_2)) :=
   \matrix_(i, j) A i (rshift n_1 j).
```

Column block

$$\left[\begin{array}{c} A_u \\ \hline A_d \end{array}\right]$$

```
Definition col_mx $A_u$ $A_d$ : 'M[$R$]_($m_1$ + $m_2$ , $n$) :=
  \matrix_($i$, $j$) match split $i$ with
      inl $i_1$ $\Rightarrow$ $A_u$ $i_1$ $j$ | inr $i_2$ $\Rightarrow$ $A_d$ $i_2$ $j$
    end.

Definition usubmx ($A$ :  'M[$R$]_($m_1$ + $m_2$, $n$)) :=
    \matrix_($i$, $j$) $A$ (lshift $m_2$ $i$) $j$.

Definition dsubmx ($A$ :  'M[$R$]_($m_1$ + $m_2$, $n$)) :=
    \matrix_($i$, $j$) $A$ (rshift $m_1$ $i$) $j$.
```

Block

$$\left[\begin{array}{c|c} A_{ul} & A_{ur} \\ \hline A_{dl} & A_{dr} \end{array}\right]$$

`Definition col_mx` A_{ul} A_{ur} A_{dl} A_{dr} `:=`
    `col_mx (row_mx` A_{ul} A_{ur}`) (row_mx` A_{dl} A_{dr}`).`

`Definition ulsubmx` A `:= lsubmx (usubmx` A`).`
`Definition ursubmx` A `:= rsubmx (usubmx` A`).`
`Definition dlsubmx` A `:= lsubmx (dsubmx` A`).`

`Definition drsubmx` A `:= rsubmx (dsubmx` A`).`

Permutation

$$\begin{array}{l} i_1 \\ \\ i_2 \end{array}\left[\begin{array}{c} \\ \mathrm{A} \\ \\ \end{array}\right]$$

```
Definition row_perm (s : 'S_m) A :=
  \matrix_(i, j) A (s i) j.

Definition perm_mx n s : 'M[R]_n := row_perm s 1%:M.

Definition xrow i1 i2 A := row_perm (tperm i1 i2) A.

Definition tperm_mx n i1 i2 : 'M[R]_n :=
  perm_mx (tperm i1 i2).
```

Picking

$${}_{i}[\mathrm{A}]$$

Let P a predicate on a finite type T

```
[pick x | P x] : option T
```

where

```
Inductive option (A : Type) := Some of A | None.
```

If there is a default value:

```
odflt a [pick x | P x] :  T
```

where

```
Definition odflt (A : Type) (a : A) (o : option A) :=
  if o is Some v then v else a.
```

Algorithm

1. Put a non-zero element at (0,0) if needed by line permutation

2. Perform the recursive call on the Schur complement

3. Recompose the result of the recursive call

Algorithm

```
Let M n :=  ’M[F]_n.
Fixpoint cormen_lup n : M n.+1 -> M n.+1 * M n.+1 * M n.+1 :=
 if n is _.+1 then
 fun A ⇒
  let k :=  odflt 0 [pick k | A k 0 != 0] in
  let A1 : ’M_(1 + _) :=  xrow 0 k A in
  let P1 : ’M_(1 + _) :=  tperm_mx 0 k in
  let Schur := drsubmx A1 - ((A k 0)^-1 *: dlsubmx A1) *m ursubmx A1 in
  let: (P2, L2, U2) := cormen_lup Schur in
  let P :=  block_mx 1 0 0 P2 * P1 in
  let L :=  block_mx 1 0 ((A k 0)^-1 *: (P2 *m dlsubmx A1)) L2 in
  let U :=  block_mx (ulsubmx A1) (ursubmx A1) 0 U2 in
  (P, L, U)
 else fun A ⇒ (1, 1, A) .
```

Specification

We have a decomposition:

```
Lemma cormen_lup_correct n (A : 'M_n.+1) :
  let: (P, L, U) := cormen_lup A in P * A = L * U.
```

The first component is a permutation:

```
Definition perm_mx n s : 'M[R]_n := row_perm s 1%:M.
Definition is_perm_mx n (A :  'M[R]_n) :=
  existsb s: 'S_n, A == perm_mx s.
```

```
Lemma cormen_lup_perm n (A : 'M[F]_n.+1) :
  let: (P, _, _) := cormen_lup A in is_perm_mx P.
```

Specification

The second component is a lower triangular matrix with one of the diagonal

```
Lemma cormen_lup_lower n (A : 'M[F]_n.+1) (i j : 'I_n.+1) :
  let: (_, L, _) := cormen_lup A in
  i <= j → L i j = (i == j)%:R.
```

The third component is an uppper triangular matrix

```
Lemma cormen_lup_upper n (A : 'M[F]_n.+1) (i j : 'I_n.+1) :
  let: (_, _, U) := cormen_lup A in
  j < i → U i j = 0.
```

Proving I

```
Lemma cormen_lup_correct n (A : 'M[F]_n.+1) :
  let: (P, L, U) := cormen_lup A in P * A = L * U.

Lemma mulmx_block (m1 m2 n1 n2 p1 p2 : nat)
   (Aul : 'M[R]_(m1, n1)) (Aur : 'M[R]_(m1, n2))
   (Adl : 'M[R]_(m2, n1)) (Adr : 'M[R]_(m2, n2))
   (Bul : 'M_(n1, p1)) (Bur : 'M_(n1, p2))
   (Bdl : 'M_(n2, p1)) (Bdr : 'M_(n2, p2)),
block_mx Aul Aur Adl Adr *m block_mx Bul Bur Bdl Bdr =
block_mx (Aul *m Bul + Aur *m Bdl) (Aul *m Bur + Aur *m Bdr)
   (Adl *m Bul + Adr *m Bdl) (Adl *m Bur + Adr *m Bdr)

Lemma mxE F : matrix_of_fun F =2 F.

Lemma mx11_scalar (A : 'M[R]_1) : A = (A 0 0)%:M.
```

Proving I

```
Lemma cormen_lup_correct n (A : 'M[F]_n.+1) :
  let: (P, L, U) := cormen_lup A in P * A = L * U.
Proof.
elim: n ⇒ [|n IHn] /= in A *; first by rewrite !mul1r.
set k := odflt _ _; set A1 : 'M_(1 + _) := xrow _ _ _.
set A' := _ - _; move/(_ A'): IHn; case: cormen_lup ⇒ [[P' L' U']] /= IHn.
rewrite -mulrA -!mulmxE -xrowE -/A1 /= -[n.+2]/(1 + n.+1)%N -{1}(submxK A1).
rewrite !mulmx_block !mul0mx !mulmx0 !add0r !addr0 !mul1mx -{L' U'}[L' *m _]IHn.
rewrite -scalemxAl !scalemxAr -!mulmxA addrC -mulrDr {A'} subrK.
congr (block_mx _ _ (_ *m _) _).
rewrite [_ *: _]mx11_scalar !mxE lshift0 tpermL {}/A1 {}/k.
case: pickP ⇒ /= [k nzAk0 | no_k]; first by rewrite mulVf ?mulmx1.
rewrite (_ : dlsubmx _ = 0) ?mul0mx //; apply/colP⇒ i.
by rewrite !mxE lshift0 (elimNf eqP (no_k _)).
Qed.
```

Proving II

```
Lemma cormen_lup_perm n (A : 'M[F]_n.+1) :
  let: (P, _, _) ≔ cormen_lup A in is_perm_mx P.


Lemma perm_mx_is_perm (n : nat) (s : 'S_n) : is_perm_mx (perm_mx s).

Lemma is_perm_mxMr (n : nat) (A B : 'M[R]_n) :
  is_perm_mx B → is_perm_mx (A *m B) = is_perm_mx A.

Definition lift0_mx A : 'M[R]_(1 + n) := block_mx 1 0 0 A.

Lemma lift0_mx_is_perm (n : nat) (s : 'S_n) :
  is_perm_mx (lift0_mx (perm_mx s)).
```

Proving II

```
Lemma cormen_lup_perm n (A : 'M[F]_n.+1) :
  let: (P, _, _) := cormen_lup A in is_perm_mx P.
Proof.
elim: n ⇒ [|n IHn] /= in A *; first exact: is_perm_mx1.
set k := odflt _ _; set A₁ : 'M_(1 + _) := xrow _ _ _.
set A′ := _ - _; move/(_ A′): IHn; case: cormen_lup ⇒ [[P L U]] {A′}/=.
rewrite (is_perm_mxMr _ (perm_mx_is_perm _ _)).
case/is_perm_mxP ⇒ s →; exact: lift0_mx_is_perm.
Qed.
```

Proving III

```
Lemma cormen_lup_lower n (A : 'M[F]_n.+1) (i j : 'I_n.+1) :
  let: (_, L, _) := cormen_lup A in
  i <= j → L i j = (i == j)%:R.
```

```
(block_mx a w v P) i j     where i j : 'I_n.+1
```

```
split (i : 'I_(1 + n)) : 'I_1 + 'I_n

                          where Inductive A + B := inl of A | inr of B.
```

```
lift: 'I_n → 'I_n.-1 → 'I_n
```

`lift i j`

```
unlift: 'I_n → 'I_n → option 'I_n.-1
liftK:  forall n (h : 'I_n), pcancel (lift h) (unlift h).
```

```
split1:   forall n (i: 'I_(1 + n)), split i = oapp inr (inl 0) (unlift 0 i).
```

Proving III

```
Lemma cormen_lup_lower n (A : 'M[F]_n.+1) (i j : 'I_n.+1) :
  let: (_, L, _) := cormen_lup A in
  i <= j → L i j = (i == j)%:R.
Proof.
elim: n ⇒ [|n IHn] /= in A i j *; first by rewrite [i]ord1 [j ]ord1 mxE.
set A′ := _ - _; move/(_ A′): IHn; case: cormen_lup ⇒ [[P L U]] {A′}/= IHn.
rewrite !mxE split1; case: unliftP ⇒ [i′|] → /=; rewrite !mxE split1.
  by case: unliftP ⇒ [j′|] → //; exact: IHn.
by case: unliftP ⇒ [j′|] →; rewrite /= mxE.
Qed.
```

Proving IV

```
Lemma cormen_lup_upper n (A : 'M[F]_n.+1) (i j : 'I_n.+1) :
  let: (_, _, U) := cormen_lup A in
  j < i → U i j = 0.
Proof.
elim: n ⇒ [|n IHn] /= in A i j *; first by rewrite [i]ord1.
set A' := _ - _; move/(_ A'): IHn; case: cormen_lup ⇒ [[P L U]] {A'}/= IHn.
rewrite !mxE split1; case: unliftP ⇒ [i'|] → //=; rewrite !mxE split1.
by case: unliftP ⇒ [j'|] →; [exact: IHn | rewrite /= mxE].
Qed.
```

Conclusions

Definition : 15 lines
Proof : 60 lines

Key point : library