Compression d'images par ondelettes - Code

Benjamin CATINAUD

Année 2017-2018

Fichier *array.ml*

```
let range n =
  let t = Array.make n 0 in
  for i = 1 to n - 1 do
    t.(i) <- i
  done;
  t

let ceil i d =
  if i mod d = 0 then i / d
  else i / d + 1

module type VECT =
  sig
    type 'a vect

    val vect_of_array : 'a array -> 'a vect
    val array_of_vect : 'a vect -> 'a array
    val length : 'a vect -> int
    val create : 'a -> int -> 'a vect
    val create_empty : 'a -> 'a vect
    val make_matrix : 'a -> int -> int -> 'a vect vect
    val alternate_id : int -> 'a -> ('a -> 'a) -> 'a vect
    val extend : 'a vect -> int -> 'a -> 'a vect

    val affect : 'a vect -> 'a -> int -> unit
    val value : 'a vect -> int -> 'a
    val reverse : 'a vect -> 'a vect
    val concat : 'a vect -> 'a vect -> 'a vect
    val sum : 'a vect -> 'a vect -> ('a -> 'a -> 'a) -> 'a vect
    val prod : 'a vect -> 'a vect -> ('a -> 'a -> 'a) -> 'a vect

    val sub_vect : 'a vect -> int -> int -> 'a vect
    val put_in : 'a vect -> 'a vect -> int -> unit
    val dilate : 'a vect -> int -> 'a -> 'a vect
    val extr_vect : 'a vect -> int -> int -> int -> 'a vect

    val filter : 'a vect -> 'a vect -> 'a vect -> ('a -> 'a -> 'a) -> ('a -> 'a -> 'a) -
  end;;
```

```
module Vect : VECT =
  struct
    type 'a vect = 'a array

    let vect_of_array t = t
    let array_of_vect v = v

    let length v = Array.length v

    let create x n = Array.make n x

    let create_empty x = Array.make 0 x

    let value v ind = v.(ind)

    let affect v x ind = v.(ind) <- x

    let make_matrix x n m = Array.make_matrix n m x

    let alternate_id n one opp =
      let v = create one n in
      for i = 0 to n - 1 do
        if i mod 2 = 1 then
          affect v (opp one) i
      done; v

    let reverse v =
      let n = length v in
      let v_rev = create (value v 0) n in
      for i = 0 to n - 1 do
        affect v_rev (value v (n - 1 - i)) i
      done; v_rev

    let concat v1 v2 =
      let n = length v1 and p = length v2 in
      if n = 0 then v2
      else if p = 0 then v1
      else
        let v = create (value v1 0) (n + p) in
        for i = 1 to (n + p - 1) do
          if i < n then
            affect v (value v1 i) i
          else
            affect v (value v2 (i - n)) i
        done; v

    let sum v1 v2 sum_elem =
      let n = length v1 in
      if not (n = length v2) then
        failwith "Vect.sum"
      else
        let v = create (value v1 0) n in
        for i = 0 to n - 1 do
          affect v (sum_elem (value v1 i) (value v2 i)) i
        done; v
```

```
let prod v1 v2 prod_elem =
  let n = length v1 in
  if not (n = length v2) then
    failwith "Vect.prod"
  else
    let v = create (value v1 0) n in
    for i = 0 to n - 1 do
      affect v (prod_elem (value v1 i) (value v2 i)) i
    done; v

let sub_vect v i j =
  let v_sub = create (value v i) (j - i) in
  for k = 1 to j - i - 1 do
    affect v_sub (value v (k + i)) k
  done; v_sub

let put_in v1 v2 i =
  let n = length v2 in
  if (i + n) > (length v1) then
    failwith "Vect.put_in"
  else
    for j = 0 to n - 1 do
      affect v1 (value v2 j) (j + i)
    done

let extend x m zero =
  let n = length x in
  if m = 0 then x
  else
    let y = create zero (n + m) in
    put_in y x 0;
    y

let dilate v s zero =
  let n = length v in
  let v_dil = create zero (n*s) in
  for i = 0 to n - 1 do
    affect v_dil (value v i) (s*i)
  done;
  v_dil

let extr_vect v ind_beg diff_ind ind_end =
  let n = ceil (ind_end - ind_beg) diff_ind in
  let v_ex = create (value v 0) n in
  let p = ref ind_beg and q = ref 0 in
  while !p < ind_end do
    affect v_ex (value v !p) !q;
    incr q; p := !p + diff_ind
  done; v_ex
```

```
    let filter b a x sum prod inv zero =
      let n = length x in
      let y = create (value x 0) n in
      let nb = length b in

      let b = (if n > nb then extend b (n - nb) zero else b) in

      for i = 0 to n - 1 do
        let s = ref zero in
        for j = 0 to i do
          s := sum !s (prod (value b j) (value x (i - j)))
        done;
        s := prod !s (inv (value a 0));
        affect y !s i
      done;
      y

  end

module type MATRIX =
  sig
    type 'a matrix

    val dim : 'a matrix -> (int * int)
    val create : 'a -> int -> int -> 'a matrix
    val matrix_of_vect : 'a Vect.vect -> 'a matrix

    val value : 'a matrix -> (int * int) -> 'a
    val affect : 'a matrix -> 'a -> (int * int) -> unit
    val vect : 'a matrix -> int -> 'a Vect.vect
    val affect_vect : 'a matrix -> 'a Vect.vect -> (int * int) -> unit
    val put_in : 'a matrix -> 'a matrix -> (int * int) -> unit
    val line : 'a matrix -> int -> 'a Vect.vect
    val affect_line : 'a matrix -> 'a Vect.vect -> (int * int) -> unit

    val sum : 'a matrix -> 'a matrix -> ('a -> 'a -> 'a) -> 'a matrix
    val prod_scal_cano : 'a matrix -> 'a matrix -> ('a -> 'a -> 'a) -> ('a -> 'a -> 'a)
    val norm_eucli : 'a matrix -> ('a -> 'a -> 'a) -> ('a -> 'a -> 'a) -> 'a -> 'a

    val extr_matrix : 'a matrix -> (int * int) -> (int * int) -> 'a matrix
  end;;

module Matrix : MATRIX =
  struct
    type 'a matrix = ('a Vect.vect) Vect.vect

    let value mat (i, j) = Vect.value (Vect.value mat i) j

    let affect mat x (i, j) =
      let vect = Vect.value mat i in
      Vect.affect vect x j; Vect.affect mat vect i

    let dim mat = (Vect.length mat, Vect.length (Vect.value mat 0))

    let create x n m = Vect.make_matrix x n m
```

```
let matrix_of_vect v = Vect.create v 1

let vect mat j =
  let (n, m) = dim mat in
  let vect_mat = Vect.create (value mat (0, j)) n in
  for i = 1 to n - 1 do
    Vect.affect vect_mat (value mat (i, j)) i
  done; vect_mat

let affect_vect mat vect (i, j) =
  for k = 0 to (Vect.length vect - 1) do
    affect mat (Vect.value vect k) (i + k, j)
  done

let line mat i = Vect.value mat i

let affect_line mat vect (i, ind_beg) =
  let line_mat = line mat i in
  Vect.put_in line_mat vect ind_beg;
  Vect.affect mat line_mat i

let put_in mat1 mat2 (i, j) =
  for k = 0 to fst (dim mat2) - 1 do
    affect_line mat1 (line mat2 k) (i + k, j)
  done

let sum mat1 mat2 sum_elem =
  let size = dim mat1 in
  if not (size = dim mat2) then
    failwith "Matrix.sum"
  else
    let mat = create (value mat1 (0, 0)) (fst size) (snd size) in
    for i = 0 to fst size - 1 do
      affect_line mat (Vect.sum (line mat1 i) (line mat2 i) sum_elem) (i, 0)
    done;
    mat

let prod_scal_cano mat1 mat2 sum_elem prod_elem zero =
  let (n, m) = dim mat1 in
  if (n, m) <> dim mat2 then
    failwith "Matrix.prod_scal_cano"
  else
    let prod_scal = ref zero in
    for i = 0 to n - 1 do
      for j = 0 to m - 1 do
        prod_scal := sum_elem !prod_scal (prod_elem (value mat1 (i, j)) (value mat2
      done
    done;
    !prod_scal

let norm_eucli mat sum_elem prod_elem zero =
  prod_scal_cano mat mat sum_elem prod_elem zero
```

```
    let extr_matrix mat (i, j) (n2, m2) =
      let (n, m) = dim mat in
      if (i + n2 > n || j + m2 > m) then
        failwith "Matrix.extr_matrix";
      let mat_extr = create (value mat (i, j)) n2 m2 in
      for k = 0 to n2 - 1 do
        for l = 0 to m2 - 1 do
          affect mat_extr (value mat (i + k, j + l)) (k, l)
        done
      done;
      mat_extr

  end;;

#use "bmp.ml"

module type PIXEL_MATRIX =
  sig
    type pixel
    type pixel_matrix

    val barycenter : pixel -> float

    val read_pixels : bitmapFileHeader -> bitmapInfoHeader -> in_channel -> pixel_matrix
    val write_pixels : out_channel -> pixel_matrix -> unit
    val read_bmp : string -> pixel_matrix
    val write_bmp : string -> pixel_matrix -> unit

    val intmatrix_of_matrix : float Matrix.matrix -> int Matrix.matrix

    val gray_levels : pixel_matrix -> float Matrix.matrix
    val pick_color : pixel -> int -> int
    val extr_uplet : pixel_matrix -> int -> float Matrix.matrix
    val red_filter : pixel_matrix -> float Matrix.matrix
    val blue_filter : pixel_matrix -> float Matrix.matrix
    val green_filter : pixel_matrix -> float Matrix.matrix
    val combine_filters : int Matrix.matrix -> int Matrix.matrix -> int Matrix.matrix ->
  end;;

module Pixel_matrix : PIXEL_MATRIX =
  struct
    type pixel = (int * int * int)
    type pixel_matrix = pixel Matrix.matrix

    let barycenter (r, g, b) =
      ( (float_of_int r) +. (float_of_int g) +. (float_of_int b)) /. 3.
```

```
let read_pixels fh ih channel =
  let w = ih.biWidth
  and h = ih.biHeight in
  let offs = offset w in
  let m = Matrix.create (0, 0, 0) w h in
  for j = 0 to h - 1 do
    for i = 0 to w - 1 do
      let b = input_byte channel in
      let g = input_byte channel in
      let r = input_byte channel in
      Matrix.affect m (r, g, b) (i, j)
    done;
    for i = 1 to offs do
      let _ = input_byte channel in ()
    done
  done;
  m

let write_pixels channel m =
  let (w, h) = Matrix.dim m in
  let offs = offset w in
  for j = 0 to h - 1 do
    for i = 0 to w - 1 do
      let r, g, b = Matrix.value m (i, j) in
      output_byte channel b;
      output_byte channel g;
      output_byte channel r;
    done;
    for i = 1 to offs do
      output_byte channel 0
    done
  done

let read_bmp filename =
  let channel = open_in_bin filename in
  let fh = read_file_header channel in
  let ih = read_info_header channel in
  let m = read_pixels fh ih channel in
  close_in channel;
  m

let write_bmp filename m =
  let channel = open_out_bin filename in
  let (w, h) = Matrix.dim m in
  let fh = make_file_header w h
  and ih = make_info_header w h in
  write_file_header channel fh;
  write_info_header channel ih;
  write_pixels channel m;
  close_out channel
```

```
let intmatrix_of_matrix matrix =
  let (w, h) = Matrix.dim matrix in
  let intmatrix = Matrix.create 0 w h in
  for i = 0 to w - 1 do
    for j = 0 to h - 1 do
      let matrix_i_j = Matrix.value matrix (i, j) in
      if matrix_i_j > 255. then
        Matrix.affect intmatrix 255 (i, j)
      else if matrix_i_j < 0. then
        Matrix.affect intmatrix 0 (i, j)
      else
        Matrix.affect intmatrix (int_of_float matrix_i_j) (i, j)
    done
  done;
  intmatrix

let gray_levels m =
  let (w, h) = Matrix.dim m in
  let m1 = Matrix.create (0.) w h in
  for i = 0 to w - 1 do
    for j = 0 to h - 1 do
      let p = Matrix.value m (i, j) in
      let c = barycenter p in
      Matrix.affect m1 c (i, j)
    done
  done;
  m1

let pick_color (r, g, b) i =
  match i with
    | 1 -> r
    | 2 -> g
    | 3 -> b
    | _ -> failwith "Pixel_matrix.pick_uplet"

let extr_uplet mat c =
  let (w, h) = Matrix.dim mat in
  let m1 = Matrix.create (0.) w h in
  for i = 0 to w - 1 do
    for j = 0 to h - 1 do
      let p = Matrix.value mat (i, j) in
      let u = float_of_int (pick_color p c) in
      Matrix.affect m1 u (i, j)
    done
  done;
  m1

let red_filter mat = extr_uplet mat 1
let green_filter mat = extr_uplet mat 2
let blue_filter mat = extr_uplet mat 3
```

```
    let combine_filters r_f g_f b_f =
      let (w, h) = Matrix.dim r_f in
      if (not ((w,h) = Matrix.dim g_f)) || (not ((w,h) = Matrix.dim b_f)) then
        failwith "Pixel_matrix.combine_filters"
      else
        let m = Matrix.create (0, 0, 0) w h in
        for i = 0 to w - 1 do
          for j = 0 to h - 1 do
            let r = Matrix.value r_f (i, j)
            and g = Matrix.value g_f (i, j)
            and b = Matrix.value b_f (i, j)
            in
            Matrix.affect m (r, g, b) (i, j)
          done
        done;
        m

  end
```

Fichier *fwt.ml*

```
#use "array.ml"

let op x = -. x
let inv x = 1. /. x
let prod = ( *. )
let sum = ( +. )
let zero = 0.
let one = 1.
let two = sum one one
let vect_1 = Vect.vect_of_array [|one|]

let round_2 n =
  if n mod 2 = 0 then n/2
  else n/2 + 1

let rec pow x n =
  if n = 0 then 1
  else
    let y = pow x (n / 2) in
    if n mod 2 = 0 then
      y * y
    else
      x * y * y

let normalise x (n, m) pow_2 =
  let offs_n = (if (n mod pow_2 = 0) then 0 else pow_2 - (n mod pow_2))
  and offs_m = (if (m mod pow_2 = 0) then 0 else pow_2 - (m mod pow_2)) in
  let y = Matrix.create zero (n + offs_n) (m + offs_m) in
  Matrix.put_in y x (0, 0);
  y, (n + offs_n , m + offs_m)
```

```
let denormalise x (n, m) =
  Matrix.extr_matrix x (0, 0) (n, m)

let aco f x =
  let n = Vect.length x
  and p = Vect.length f in
  let xpadded = ref (Vect.create_empty (Vect.value x 0)) in
  (if p < n then
    xpadded := (Vect.concat x (Vect.sub_vect x 0 p))
  else
    let z = Vect.create zero p in
    for i = 0 to p - 1 do
      let imod = (i mod n) in
      Vect.affect z (Vect.value x imod) i
    done;
    xpadded := (Vect.concat x z));
  let fflip = Vect.reverse f in
  let ypadded = Vect.filter fflip vect_1 !xpadded sum prod inv zero in
  Vect.sub_vect ypadded (p - 1) (n + p - 1)

let hi_up x g0 =
  let tmp = ref (Vect.dilate x 2 zero) in
  let len_tmp = Vect.length !tmp - 1 in
  tmp := Vect.concat (Vect.create (Vect.value !tmp len_tmp) 1)
        (Vect.sub_vect !tmp 0 len_tmp);
  aco g0 !tmp

let lo_conv x qmf =
  let d = aco qmf x in
  Vect.extr_vect d 0 2 (Vect.length d - 1)

let ico f x =
  let n = Vect.length x
  and p = Vect.length f in
  let xpadded = ref (Vect.create (Vect.value x 0) 1) in
  (if p <= n then
    xpadded := (Vect.concat (Vect.sub_vect x (n - p) n) x)
  else
    let z = Vect.create zero p in
    for i = 0 to p - 1 do
      let imod = ( (p * n - p + i) mod n ) in
      Vect.affect z (Vect.value x imod) i
    done;
    xpadded := (Vect.concat z x));
  let ypadded = Vect.filter f vect_1 !xpadded sum prod inv zero in
  Vect.sub_vect ypadded p (n + p)

let hi_conv s qmf =
  let s2 = Vect.concat (Vect.sub_vect s 1 (Vect.length s))
          (Vect.create (Vect.value s 0) 1) in
  let d = ico qmf s2 in
  Vect.extr_vect d 0 2 (Vect.length d - 1)
```

```
let fwt sens x l h0 =
  let len_h0 = Vect.length h0 in
  let g0 = Vect.prod (Vect.alternate_id len_h0 one op) h0 prod in

  if sens = 0 then
  (let wc, (n, m) = normalise x (Matrix.dim x) (pow 2 l) in
  let nc = ref n
  and mc = ref m in

  for jsqual = 1 to l do
    for ix = 0 to !nc - 1 do
      let row = Vect.sub_vect (Matrix.line wc ix) 0 !mc in
      Matrix.affect_line wc (lo_conv row h0) (ix, 0);
      Matrix.affect_line wc (hi_conv row g0) (ix, (!mc / 2));
    done;

    for iy = 0 to !mc - 1 do
      let row = Vect.sub_vect (Matrix.vect wc iy) 0 !nc in
      Matrix.affect_vect wc (hi_conv row g0) (!nc / 2, iy);
      Matrix.affect_vect wc (lo_conv row h0) (0, iy);
    done;

    nc := !nc / 2; mc := !mc / 2
  done; wc)

  else
    (let (n, m) = Matrix.dim x in
    let pow_2 = pow 2 (l - 1) in
    let nc = ref (n / pow_2)
    and mc = ref (m / pow_2) in

    for jsqual = 1 to l do
      for iy = 0 to !mc - 1 do
        Matrix.affect_vect x (Vect.sum
                (ico h0 (Vect.dilate
                            (Vect.sub_vect (Matrix.vect x iy) 0 (!nc / 2))
                            2 zero))
                (hi_up (Vect.sub_vect (Matrix.vect x iy) (!nc / 2) !nc) g0)
                sum) (0, iy)
      done;

      for ix = 0 to !nc - 1 do
        Matrix.affect_line x (Vect.sum
                (ico h0 (Vect.dilate
                            (Vect.sub_vect (Matrix.line x ix) 0 (!mc / 2))
                            2 zero))
                (hi_up (Vect.sub_vect (Matrix.line x ix) (!mc / 2) !mc) g0)
                sum) (ix, 0)
      done;

      nc := !nc * 2; mc := !mc * 2;
    done; x)
```

Fichier *compression_wt.ml*

```
#use "array.ml";;
#use "fwt.ml";;

let abs x =
  if x < zero then op x
  else x

let abs2 x =
  if x < 0 then - x
  else x

let haar_filter = Vect.prod_scal (Vect.vect_of_array [|1. ; 1.|]) 2. ( /. );;
let haar_filter2 = Vect.vect_of_array [|1. ; 1.|];;

let decompo_wt_matrix img_m it =
  fwt 0 img_m it haar_filter;;

let decompo_wt img it =
  let img_matrix = Pixel_matrix.read_bmp img in
  let gray_matrix = Pixel_matrix.gray_levels img_matrix in
  let decompo = decompo_wt_matrix gray_matrix it in
  decompo;;

let rec calc_indice_div it (n, m) =
  if it = 0 then
    (n, m)
  else
    calc_indice_div (it - 1) (n / 2, m / 2)

let rec calc_indice_mult it (n, m) =
  if it = 0 then
    (n, m)
  else
    calc_indice_mult (it - 1) (n * 2, m * 2)

let txt_of_matrix mat it filename =
  let (n, m) = Matrix.dim mat in
  let (size_n, size_m) = calc_indice_div it (n, m) in
  let extr_mat = Matrix.extr_matrix mat (0, 0) (size_n, size_m) in
  let file = open_out filename in
  output_string file ((string_of_int it) ^ "\n");
  for i = 0 to n - 1 do
    for j = 0 to m - 1 do
      let mat_i_j = Matrix.value mat (i, j) in
      if (i >= size_n || j >= size_m) && mat_i_j <> 0 then
        let str = (string_of_int i) ^ " " ^ (string_of_int j) ^
                  " " ^ (string_of_int mat_i_j) ^ "\n" in
        output_string file str
    done
  done;
  close_out file;
  extr_mat
```

```
let decompo_line line =
  let tab = Array.make 3 0 in
  let n = String.length line in
  let str = ref "" in
  let j = ref 0 in
  for i = 0 to n - 1 do
    if line.[i] <> ' ' then
      (str := !str ^ (String.make 1 line.[i]);
      if i = n - 1 then
        tab.(!j) <- int_of_string !str)
    else
      (tab.(!j) <- int_of_string !str;
      str := "";
      incr j)
  done;
  (tab.(0), tab.(1), tab.(2))

let matrix_of_txt mat filename =
  let file = open_in filename in
  let mult = int_of_string (input_line file) in
  let (size_n, size_m) = Matrix.dim mat in
  let (n, m) = calc_indice_mult mult (size_n, size_m) in

  let complete_mat = Matrix.create 0. n m in
  Matrix.put_in complete_mat mat (0, 0);

  let end_of_file = ref false in

  while (not !end_of_file) do
    try
      let line = input_line file in
      let (i, j, value) = decompo_line line in
      Matrix.affect complete_mat (float_of_int value) (i, j)
    with
      | End_of_file -> end_of_file := true
  done;
  close_in file;

  (complete_mat, mult);;

let compresser_img img it filename_target =
  let mat = decompo_wt img it in
  let intmat = Pixel_matrix.intmatrix_of_matrix mat in
  let extr_mat = txt_of_matrix intmat it filename_target in
  Pixel_matrix.write_bmp (filename_target ^ ".bmp")
      (Pixel_matrix.combine_filters extr_mat extr_mat extr_mat);;

let decompresser_img img file target =
  let mat = Pixel_matrix.read_bmp img in
  let gray_mat = Pixel_matrix.gray_levels mat in
  let (mat_recomp, it) = matrix_of_txt gray_mat file in
  let decomp = fwt 1 mat_recomp it haar_filter2 in
  let intdecomp = Pixel_matrix.intmatrix_of_matrix decomp in
  let combine = Pixel_matrix.combine_filters intdecomp intdecomp intdecomp in
  Pixel_matrix.write_bmp target combine;;
```

```
let error img it i =
  let mat = Pixel_matrix.read_bmp img in
  let original = Pixel_matrix.gray_levels mat in
  let comp = decompo_wt_matrix original it in

  let intcomp = Pixel_matrix.intmatrix_of_matrix comp in
  let extr = txt_of_matrix intcomp it ("text" ^ (string_of_int (i + 1))) in

  let pixextr = Pixel_matrix.combine_filters extr extr extr in
  let extrfloat = Pixel_matrix.gray_levels pixextr in

  let (recomp, _) = matrix_of_txt extrfloat ("text" ^ (string_of_int (i + 1))) in

  let decomp = fwt 1 recomp it haar_filter2 in

  let denorm_decomp = denormalise decomp (Matrix.dim original) in

  let intoriginal = Pixel_matrix.intmatrix_of_matrix original
  and intdecomp = Pixel_matrix.intmatrix_of_matrix denorm_decomp in

  let mat_diff = Matrix.sum intoriginal intdecomp ( - ) in
  Matrix.norm_eucli mat_diff ( + ) ( * ) 0
```

Fichier *bmp.ml*

```
#load "graphics.cma";;
open Graphics;;

type word  = int;;
type dword = int;;

type bitmapFileHeader = {
  bfType      : string;
  bfSize      : dword;
  bfReserved1 : word;
  bfReserved2 : word;
  bfOffBits   : dword;
};;

type bitmapInfoHeader = {
  biSize          : dword;
  biWidth         : dword;
  biHeight        : dword;
  biPlanes        : word;
  biBitCount      : word;
  biCompression   : dword;
  biSizeImage     : dword;
  biXPelsPerMeter : dword;
  biYPelsPerMeter : dword;
  biClrUsed       : dword;
  biClrImportant  : dword;
};;
let read_type channel =
```

```
  let s = "  " in
  s.[0] <- input_char channel;
  s.[1] <- input_char channel;
  s;;

let write_type channel =
  output_char channel 'B';
  output_char channel 'M';;

let read_dword channel =
  let a = input_byte channel in
  let b = input_byte channel in
  let c = input_byte channel in
  let d = input_byte channel in
  (d lsl 24) lor (c lsl 16) lor (b lsl 8) lor a;;

let write_dword channel x =
  let a = x lsr 24
  and b = (x lsr 16) land 255
  and c = (x lsr 8) land 255
  and d = x land 255 in
  output_byte channel d;
  output_byte channel c;
  output_byte channel b;
  output_byte channel a;;

let read_word channel =
  let a = input_byte channel in
  let b = input_byte channel in
  (b lsl 8) lor a;;

let write_word channel x =
  let c = (x lsr 8) land 255
  and d = x land 255 in
  output_byte channel d;
  output_byte channel c;;
```

```
let read_file_header channel =
  let t = read_type channel in
  let sz = read_dword channel in
  let r1 = read_word channel in
  let r2 = read_word channel in
  let off = read_dword channel in
  {
    bfType = t;
    bfSize = sz;
    bfReserved1 = r1;
    bfReserved2 = r2;
    bfOffBits = off;
  };;

let write_file_header channel fh =
  write_type channel;
  write_dword channel fh.bfSize;
  write_word channel fh.bfReserved1;
  write_word channel fh.bfReserved2;
  write_dword channel fh.bfOffBits;;

let read_info_header channel =
  let sz = read_dword channel in
  let w = read_dword channel in
  let h = read_dword channel in
  let pl = read_word channel in
  let bc = read_word channel in
  let compr = read_dword channel in
  let szim = read_dword channel in
  let xpm = read_dword channel in
  let ypm = read_dword channel in
  let clru = read_dword channel in
  let clri = read_dword channel in
  {
    biSize = sz;
    biWidth = w;
    biHeight = h;
    biPlanes = pl;
    biBitCount = bc;
    biCompression = compr;
    biSizeImage = szim;
    biXPelsPerMeter= xpm;
    biYPelsPerMeter= ypm;
    biClrUsed = clru;
    biClrImportant = clri;
  };;
```

```
let write_info_header channel ih =
  write_dword channel ih.biSize;
  write_dword channel ih.biWidth;
  write_dword channel ih.biHeight;
  write_word channel ih.biPlanes;
  write_word channel ih.biBitCount;
  write_dword channel ih.biCompression;
  write_dword channel ih.biSizeImage;
  write_dword channel ih.biXPelsPerMeter;
  write_dword channel ih.biYPelsPerMeter;
  write_dword channel ih.biClrUsed;
  write_dword channel ih.biClrImportant;;

let offset w =
  let r = (3 * w) mod 4 in
  if r = 0 then 0
  else 4 - r;;

let make_file_header w h =
  let off = offset w in
  {
    bfType = "BM";
    bfSize = (w + off) * h * 3 + 54;
    bfReserved1  = 0;
    bfReserved2 = 0;
    bfOffBits = 54;
  };;

let make_info_header w h =
  let off = offset w in
  {
    biSize = 40;
    biWidth = w;
    biHeight  = h;
    biPlanes  = 1;
    biBitCount = 24;
    biCompression = 0;
    biSizeImage = (w + off) * h * 3;
    biXPelsPerMeter = 0;
    biYPelsPerMeter = 0;
    biClrUsed = 0;
    biClrImportant = 0;
  };;
```