MATLAB implementation of the alpha scheme for the heat transfer problem

Below I present the MATLAB code performing simulations for the alpha scheme for the two-dimensional heat transport problem.

```
format long;

% Build cartesian product of specified vectors.
% Vector orientation is arbitrary.
%
% Order: first component changes fastest
%
% a1, a2, ... - sequence of n vectors
%
% returns - array of n-columns containing all the combinations of values in aj
function c = cartesian(varargin)
  n = nargin;
```

```
    [F{1:n}] = ndgrid(varargin{:});
    for i = n:-1:1
      c(i,:) = F{i}(:);
    end
  end

  % Create a row vector of size n filled with val
  function r = row_of(val, n)
    r = val * ones(1, n);
  end


  % Index conventions
  %------------------
  %
  % DoFs             - zero-based
  % Elements         - zero-based
  % Knot elements    - zero-based
  % Linear indices   - one-based


  % Create an one-dimensional basis object from specified data.
  % Performs some simple input validation.
  %
  % For a standard, clamped B-spline basis first and last elements of the knot vector
  % should be repeated (p+1) times.
  %
  % p       - polynomial order
```

```
% points  - increasing sequence of values defining the mesh
% knot    - knot vector containing integer indices of mesh points (starting from 0)
%
% returns - structure describing the basis
function b = basis1d(p, points, knot)
  validateattributes(points, {}, {'increasing'});
  validateattributes(knot, {}, {'nondecreasing'});
  assert(max(knot) == length(points) - 1, sprintf('Invalid knot index: %d, points: %d)', max(knot), lengt

  b.p = p;
  b.points = points;
  b.knot = knot;
endfunction

% Number of basis functions (DoFs) in the 1D basis
function n = number_of_dofs(b)
  n = length(b.knot) - b.p - 1;
endfunction

% Number of elements the domain is subdivided into
function n = number_of_elements(b)
  n = length(b.points) - 1;
endfunction

% Domain point corresponding to i-th element of the knot vector
function x = knot_point(b, i)
  x = b.points(b.knot(i) + 1);
endfunction
```

```
% Row vector containing indices of all the DoFs
function idx = dofs1d(b)
  n = number_of_dofs(b);
  idx = 0 : n-1;
endfunction

% Enumerate degrees of freedom in a tensor product of 1D bases
%
% b1, b2, ...  - sequence of n 1D bases
%
% returns - array of indices (n-columns) of basis functions
function idx = dofs(varargin)
  if (nargin == 1)
    idx = dofs1d(varargin{:});
  else
    ranges = cellfun(@(b) dofs1d(b), varargin, 'UniformOutput', false);
    idx = cartesian(ranges{:});
  endif
endfunction

% Row vector containing indices of all the elements
function idx = elements1d(b)
  n = number_of_elements(b);
  idx = 0 : n-1;
endfunction

% Enumerate element indices for a tensor product of 1D bases
```

```
%
% b1, b2, ...  - sequence of n 1D bases
%
% returns - array of indices (n-columns) of element indices
function idx = elements(varargin)
  if (nargin == 1)
    idx = elements1d(varargin{:});
  else
    ranges = cellfun(@(b) elements1d(b), varargin, 'UniformOutput', false);
    idx = cartesian(ranges{:});
  endif
endfunction

% Index of the first DoF that is non-zero over the specified element
function idx = first_dof_on_element(e, b)
  idx = lookup(b.knot, e) - b.p - 1;
endfunction

% Row vector containing indices of DoFs that are non-zero over the specified element
%
% e - element index (scalar)
% b - 1D basis
function idx = dofs_on_element1d(e, b)
  a = first_dof_on_element(e, b);
  idx = a : a + b.p;
endfunction

% Row vector containing indices (columns) of DoFs that are non-zero over the specified element
```

```
%
% e      - element index (pair)
% bx, by - 1D bases
function idx = dofs_on_element2d(e, bx, by)
  rx = dofs_on_element1d(e(1), bx);
  ry = dofs_on_element1d(e(2), by);
  idx = cartesian(rx, ry);
endfunction

% Compute 1-based, linear index of tensor product DoF.
% Column-major order - first index component changes fastest.
%
% dof            - n-tuple index
% b1, b2,, ...   - sequence of n 1D bases
%
% returns - linearized scalar index
function idx = linear_index(dof, varargin)
  n = length(varargin);

  idx = dof(n);
  for i = n-1 : -1 : 1
    ni = number_of_dofs(varargin{i});
    idx = dof(i) + idx * ni;
  endfor

  idx += 1;
endfunction
```

```
% Assuming clamped B-spline basis, compute the polynomial order based on the knot
function p = degree_from_knot(knot)
  p = find(knot > 0, 1) - 2;
endfunction

% Create a knot without interior repeated nodes
%
% elems - number of elements to subdivide domain into
% p     - polynomial degree
function knot = simple_knot(elems, p)
  pad = ones(1, p);
  knot = [0 * pad, 0:elems, elems * pad];
endfunction


% Spline evaluation functions are based on:
%
%    The NURBS Book, L. Piegl, W. Tiller, Springer 1995


% Find index i such that x lies between points corresponding to knot(i) and knot(i+1)
function span = find_span(x, b)
  low  = b.p + 1;
  high = number_of_dofs(b) + 1;

  if (x >= knot_point(b, high))
    span = high - 1;
  elseif (x <= knot_point(b, low))
```

```
      span = low;
    else
      span = floor((low + high) / 2);
      while (x < knot_point(b, span) || x >= knot_point(b, span + 1))
        if (x < knot_point(b, span))
          high = span;
        else
          low = span;
        endif
        span = floor((low + high) / 2);
      endwhile
    endif
  endfunction

  % Compute values at point x of (p+1) basis functions that are nonzero over the element
  % corresponding to specified span.
  %
  % span  - span containing x, as computed by function find_span
  % x     - point of evaluation
  % b     - basis
  %
  % returns - vector of size (p+1)
  function out = evaluate_bspline_basis(span, x, b)
    p = b.p;
    out = zeros(p + 1, 1);
    left = zeros(p, 1);
    right = zeros(p, 1);
    out(1) = 1;
    for j = 1:p
      left(j)  = x - knot_point(b, span + 1 - j);
      right(j) = knot_point(b, span + j) - x;
      saved = 0;

      for r = 1:j
        tmp = out(r) / (right(r) + left(j - r + 1));
        out(r) = saved + right(r) * tmp;
        saved = left(j - r + 1) * tmp;
      endfor
      out(j + 1) = saved;
    endfor
  endfunction

  % Compute values and derivatives of order up to der at point x of (p+1) basis functions
  % that are nonzero over the element corresponding to specified span.
  %
  % span  - span containing x, as computed by function find_span
  % x     - point of evaluation
  % b     - basis
  %
  % returns - array of size (p+1) x (der + 1) containing values and derivatives
  function out = evaluate_bspline_basis_ders(span, x, b, der)
    p = b.p;
    out = zeros(p + 1, der + 1);
    left = zeros(p, 1);
    right = zeros(p, 1);
```

```
ndu = zeros(p + 1, p + 1);
a = zeros(2, p + 1);

ndu(1, 1) = 1;
for j = 1:p
  left(j)  = x - knot_point(b, span + 1 - j);
  right(j) = knot_point(b, span + j) - x;
  saved = 0;

  for r = 1:j
    ndu(j + 1, r) = right(r) + left(j - r + 1);
    tmp = ndu(r, j) / ndu(j + 1, r);
    ndu(r, j + 1) = saved + right(r) * tmp;
    saved = left(j - r + 1) * tmp;
  endfor
  ndu(j + 1, j + 1) = saved;
endfor

out(:, 1) = ndu(:, p + 1);

for r = 0:p
  s1 = 1;
  s2 = 2;
  a(1, 1) = 1;

  for k = 1:der
    d = 0;
    rk = r - k;
```

```
      pk = p - k;
      if (r >= k)
        a(s2, 1) = a(s1, 1) / ndu(pk + 2, rk + 1);
        d = a(s2, 1) * ndu(rk + 1, pk + 1);
      endif
      j1 = max(-rk, 1);
      if (r - 1 <= pk)
        j2 = k - 1;
      else
        j2 = p - r;
      endif
      for j = j1:j2
        a(s2, j + 1) = (a(s1, j + 1) - a(s1, j)) / ndu(pk + 2, rk + j + 1);
        d = d + a(s2, j + 1) * ndu(rk + j + 1, pk + 1);
      endfor
      if (r <= pk)
        a(s2, k + 1) = -a(s1, k) / ndu(pk + 2, r + 1);
        d = d + a(s2, k + 1) * ndu(r + 1, pk + 1);
      endif
      out(r + 1, k + 1) = d;
      t = s1;
      s1 = s2;
      s2 = t;
    endfor
  endfor

  r = p;
  for k = 1:der
```

```
      for j = 1:p+1
        out(j, k + 1) = out(j, k + 1) * r;
      endfor
      r = r * (p - k);
    endfor

  endfunction

  % Evaluate combination of 2D B-splines at point x
  function val = evaluate2d(u, x, bx, by)
    sx = find_span(x(1), bx);
    sy = find_span(x(2), by);

    valsx = evaluate_bspline_basis(sx, x(1), bx);
    valsy = evaluate_bspline_basis(sy, x(2), by);

    offx = sx - bx.p;
    offy = sy - by.p;

    val = 0;
    for i = 0:bx.p
      for j = 0:by.p
        val = val + u(offx + i, offy + j) * valsx(i + 1) * valsy(j + 1);
      endfor
    endfor
  endfunction

  % Compute value and gradient of combination of 1D B-splines at point x
```

```
function [val, grad] = evaluate_with_grad2d(u, x, bx, by)
  sx = find_span(x(1), bx);
  sy = find_span(x(2), by);

  valsx = evaluate_bspline_basis_ders(sx, x(1), bx, 1);
  valsy = evaluate_bspline_basis_ders(sy, x(2), by, 1);

  offx = sx - bx.p;
  offy = sy - by.p;

  val = 0;
  grad = [0 0];
  for i = 0:bx.p
    for j = 0:by.p
      c = u(offx + i, offy + j);
      val     += c * valsx(i + 1, 1) * valsy(j + 1, 1);
      grad(1) += c * valsx(i + 1, 2) * valsy(j + 1, 1);
      grad(2) += c * valsx(i + 1, 1) * valsy(j + 1, 2);
    endfor
  endfor
endfunction

% Returns a structure containing information about 1D basis functions that can be non-zero at x,
% with the following fields:
%   offset - difference between global DoF numbers and indices into vals array
%   vals   - array of size (p+1) x (der + 1) containing values and derivatives of basis functions at x
function data = eval_local_basis(x, b, ders)
  span = find_span(x, b);
  first = span - b.p - 1;
  data.offset = first - 1;
  data.vals = evaluate_bspline_basis_ders(span, x, b, ders);
endfunction

% Compute value and derivative of specified 1D basis function, given data computed
% by function eval_local_basis
function [v, dv] = eval_dof1d(dof, data, b)
  v = data.vals(dof - data.offset, 1);
  dv = data.vals(dof - data.offset, 2);
endfunction

% Compute value and gradient of specified 2D basis function, given data computed
% by function eval_local_basis
function [v, dv] = eval_dof2d(dof, datax, datay, bx, by)
  [a, da] = eval_dof1d(dof(1), datax, bx);
  [b, db] = eval_dof1d(dof(2), datay, by);
  v = a * b;
  dv = [da * b, a * db];
endfunction

% Creates a wrapper function that takes 2D basis function index as argument and returns
% its value and gradient
function f = basis_evaluator2d(x, bx, by, ders)
  datax = eval_local_basis(x(1), bx, 1);
  datay = eval_local_basis(x(2), by, 1);
  f = @(i) eval_dof2d(i, datax, datay, bx, by);
endfunction
```

```
% Value of 1D element mapping jacobian (size of the element)
function a = jacobian1d(e, b)
  a = b.points(e + 2) - b.points(e + 1);
endfunction

% Value of 2D element mapping jacobian (size of the element)
function a = jacobian2d(e, bx, by)
  a = jacobian1d(e(1), bx) * jacobian1d(e(2), by);
endfunction

% Row vector of points of the k-point Gaussian quadrature on [a, b]
function xs = quad_points(a, b, k)
  % Affine mapping [-1, 1] -> [a, b]
  map = @(x) 0.5 * (a * (1 - x) + b * (x + 1));
  switch (k)
    case 1
      xs = [0];
    case 2
      xs = [-0.5773502691896257645, ...
             0.5773502691896257645];
    case 3
      xs = [-0.7745966692414833770, ...
             0,                     ...
             0.7745966692414833770];
    case 4
      xs = [-0.8611363115940525752, ...
```

```
              -0.3399810435848562648, ...
               0.3399810435848562648, ...
               0.8611363115940525752];
        case 5
          xs = [-0.9061798459386639928, ...
              -0.5384693101056830910, ...
               0,                     ...
               0.5384693101056830910, ...
               0.9061798459386639928];
      endswitch
      xs = map(xs);
    endfunction

    % Row vector of weights of the k-point Gaussian quadrature on [a, b]
    function ws = quad_weights(k)
      switch (k)
        case 1
          ws = [2];
        case 2
          ws = [1, 1];
        case 3
          ws = [0.5555555555555555556, ...
               0.8888888888888888889, ...
               0.5555555555555555556];
        case 4
          ws = [0.3478548451374538573, ...
               0.6521451548625461426, ...
               0.6521451548625461426, ...
```

```
            0.3478548451374538573 7];
      case 5
        ws = [0.2369268850561890875 1, ...
              0.4786286704993664680 4, ...
              0.5688888888888888888 9, ...
              0.4786286704993664680 4, ...
              0.2369268850561890875 1]
    endswitch
    % Gaussian quadrature is defined on [-1, 1], we use [0, 1]
    ws = ws / 2;
  endfunction

  % Create array of structures containing quadrature data for integrating over 2D element
  %
  % e      - element index (pair)
  % k      - quadrature order
  % bx, by - 1D bases
  %
  % returns - array of structures with fields
  %             x - point
  %             w - weight
  function qs = quad_data2d(e, k, bx, by)
    xs = quad_points(bx.points(e(1) + 1), bx.points(e(1) + 2), k);
    ys = quad_points(by.points(e(2) + 1), by.points(e(2) + 2), k);
    ws = quad_weights(k);

    for i = 1:k
      for j = 1:k
        qs(i, j).x = [xs(i), ys(j)];
        qs(i, j).w = ws(i) * ws(j);
      endfor
    endfor
    qs = reshape(qs, 1, []);

  endfunction

  % Row vector containing indices (columns) of DoFs non-zero on the left edge
  function ds = boundary_dofs_left(bx, by)
    ny = number_of_dofs(by);

    ds = [row_of(0, ny); dofs(by)];
  endfunction

  % Row vector containing indices (columns) of DoFs non-zero on the right edge
  function ds = boundary_dofs_right(bx, by)
    nx = number_of_dofs(bx);
    ny = number_of_dofs(by);

    ds = [row_of(nx - 1, ny); dofs(by)];
  endfunction

  % Row vector containing indices (columns) of DoFs non-zero on the bottom edge
  function ds = boundary_dofs_bottom(bx, by)
    nx = number_of_dofs(bx);

    ds = [dofs(bx); row_of(0, nx)];
```

```
endfunction

% Row vector containing indices (columns) of DoFs non-zero on the top edge
function ds = boundary_dofs_top(bx, by)
  nx = number_of_dofs(bx);
  ny = number_of_dofs(by);

  ds = [dofs(bx); row_of(ny - 1, nx)];
endfunction

% Row vector containing indices (columns) of DoFs non-zero on some part of the boundary
function ds = boundary_dofs2d(bx, by)
  left   = boundary_dofs_left(bx, by);
  right  = boundary_dofs_right(bx, by);
  bottom = boundary_dofs_bottom(bx, by);
  top    = boundary_dofs_top(bx, by);

  ds = [left, right, top(:,2:end-1), bottom(:,2:end-1)];
endfunction

% Modify matrix and right-hand side to enforce uniform (zero) Dirichlet boundary conditions
%
% M      - matrix
% F      - right-hand side
% dofs   - degrees of freedom to be fixed
% bx, by - 1D bases
%
% returns - modified M and F
```

```
function [M, F] = dirichlet_bc_uniform(M, F, dofs, bx, by)
  for d = dofs
    i = linear_index(d, bx, by);
    M(i, :) = 0;
    M(i, i) = 1;
    F(i) = 0;
  endfor
endfunction



% Evaluate function on a 2D cartesian product grid
%
% f      - function accepting 2D point as a two-element vector
% xs, ys - 1D arrays of coordinates
%
% returns - 2D array of values with (i, j) -> f( xs(j), ys(i) )
%           (this order is compatible with plotting functions)
function vals = evaluate_on_grid(f, xs, ys)
  [X, Y] = meshgrid(xs, ys);
  vals = arrayfun(@(x, y) f([x y]), X, Y);
endfunction

% Subdivide xr and yr into N equal size elements
function [xs, ys] = make_grid(xr, yr, N)
  xs = linspace(xr(1), xr(2), N + 1);
  ys = linspace(yr(1), yr(2), N + 1);
endfunction
```

```
% Plot 2D B-spline with coefficients u on a square given as product of xr and yr
%
% u      - matrix of coefficients
% xr, yr - intervals specifying the domain, given as two-element vectors
% N      - number of plot 'pixels' in each direction
% bx, by - 1D bases
%
% Domain given by xr and yr should be contained in the domain of the B-spline bases
function surface_plot_spline(u, xr, yr, N, bx, by)
  [xs, ys] = make_grid(xr, yr, N);
  vals = evaluate_on_grid(@(x) evaluate2d(u, x, bx, by), xs, ys);
  surface_plot_values(vals, xs, ys);
endfunction

% Plot array of values
%
% vals   - 2D array of size [length(ys), length(xs)]
% xs, ys - 1D arrays of coordinates
function surface_plot_values(vals, xs, ys)
  surf(xs, ys, vals);
  xlabel('x');
  ylabel('y');
endfunction



% Compute L2-projection of f onto 2D B-spline space spanned by the tensor product
% of bases bx and by
%
```

```
% f       - real-valued function taking two-element vector argument
% bx, by - 1D basis
%
% returns - matrix of coefficients
function u = project2d(f, bx, by)
  nx = number_of_dofs(bx);
  ny = number_of_dofs(by);
  n = nx * ny;
  k = max([bx.p, by.p]) + 1;
  idx = @(dof) linear_index(dof, bx, by);

  M = sparse(n, n);
  F = zeros(n, 1);

  for e = elements(bx, by)
    J = jacobian2d(e, bx, by);
    for q = quad_data2d(e, k, bx, by)
      basis = basis_evaluator2d(q.x, bx, by);

      for i = dofs_on_element2d(e, bx, by)
        v = basis(i);
        for j = dofs_on_element2d(e, bx, by)
          u = basis(j);
          M(idx(i), idx(j)) += u * v * q.w * J;
        endfor

        F(idx(i)) += f(q.x) * v * q.w * J;
      endfor
    endfor
  endfor

  u = reshape(M \ F, nx, ny);
endfunction





% Auxiliary function saving plot to a file with name including iteration number
function save_plot(u, iter, bx, by)
  N = 50;
  h = figure('visible', 'off');
  surface_plot_spline(u, [0 1], [0 1], N, bx, by);
  zlim([0 0.8]);
  saveas(h, sprintf('out_%d.png', iter));
endfunction



% Input data
knot = simple_knot(5, 2);      % knot vector
dt = 0.01;                     % time step size
alpha = 0.5;                   % scheme parameter (0 - explicit Euler, 1 - implicit Euler, 1/2 - Crank-Nic
K = 20;                        % number of time steps

% Problem formulation
f = @(t, x) 1;
init_state = @(x) 0;
```

```
% Setup
p = degree_from_knot(knot);
k = p + 1;

points = linspace(0, 1, max(knot) + 1);

bx = basis1d(p, points, knot);
by = basis1d(p, points, knot);

nx = number_of_dofs(bx);
ny = number_of_dofs(by);
n = nx * ny;

M = sparse(n, n);
F = zeros(n, 1);

idx = @(dof) linear_index(dof, bx, by);

% Assemble the matrix
for e = elements(bx, by)
  J = jacobian2d(e, bx, by);
  for q = quad_data2d(e, k, bx, by)
    basis = basis_evaluator2d(q.x, bx, by);

    for i = dofs_on_element2d(e, bx, by)
      [v, dv] = basis(i);
      for j = dofs_on_element2d(e, bx, by)
```

```
        [u, du] = basis(j);
        val = u * v + dt * alpha * dot(du, dv);
        M(idx(i), idx(j)) += val * q.w * J;
      endfor
    endfor
  endfor
endfor

% Modify the matrix to account for uniform Dirichlet boundary conditions
fixed_dofs = boundary_dofs2d(bx, by);
[M, F] = dirichlet_bc_uniform(M, F, fixed_dofs, bx, by);

% Put the initial state into u
u = project2d(init_state, bx, by);

% Plot the initial state
save_plot(u, 0, bx, by);

% Time stepping loop
for m = 1:K
  t = m * dt;
  printf('Iter %d, t = %f\n', m, t);

  % Assemble the right-hand side
  F(:) = 0;
  for e = elements(bx, by)
    J = jacobian2d(e, bx, by);
    for q = quad_data2d(e, k, bx, by)
```

```
      basis = basis_evaluator2d(q.x, bx, by);

      % u - solution from the previous time step
      [U, dU] = evaluate_with_grad2d(u, q.x, bx, by);
      fval = alpha * f(t, q.x) + (1 - alpha) * f(t - dt, q.x);

      for i = dofs_on_element2d(e, bx, by)
        [v, dv] = basis(i);

        rhs = U * v - dt * (1 - alpha) * dot(dU, dv) + dt * fval;
        F(idx(i)) += rhs * q.w * J;
      endfor
    endfor
  endfor

  % Impose boundary conditions
  for d = fixed_dofs
    F(idx(d)) = 0;
  endfor

  % Solve
  u = reshape(M \ F, nx, ny);

  % Plot the solution
  save_plot(u, m, bx, by);

endfor
```

Listing 1 (Pobierz): Kod MATLABa rozwiązujący dwuwymiarowy problem transportu ciepła za pomocą schematu alfa.

In line 621 we enter the time step size $dt = 0.01$, in line 622 we enter the parameter
$alpha = 0.5$, where alpha=0 means explicit Euler, alpha=1 implicit Euler, and alpha=0.5 means Cranck-Nicolson scheme.
In line 623 we specify the time steps number
$K = 20;$
The code can be run in the free Octave environment.
The code is activated by opening it in Octave and typing a command
$heat\_time$
During operation, the code prints successive time steps

```
Iter 1, t = 0.010000
Iter 2, t = 0.020000
Iter 3, t = 0.030000
...
```

The code generates a file at any time in the current directory out_*.png, e.g.

```
out_0.png
out_1.png
out_2.png
...
```

containing solutions from individual time steps.