how U doin'?

$$🐳^{🐳^{🐳}} = \int_{🎃}^{🎅} 🐏 \, d🍀$$

# FEM for TPBVP *

10170437 Mark Taylor

December 18, 2020

## Contents

**1 Problems** 3

**2 Theory** 3

**3 Solutions** 3

**4 Code** 5

4.1 Scripts (Main Functions) . . . . . 5

4.2 Error Estimates . . . . . 6

4.3 PLRR and PQRR . . . . . 9

4.4 PLRR_intpol and PQRR_intpol . . . . . 13

4.5 Gaussquad . . . . . 14

4.6 GauEli . . . . . 15

**References** 16

**Appendices** 16

*This article was typeset by Mark Taylor using the LaTeX document processing system.

# 1 Problems

问题：

$$-\frac{\mathrm{d}}{\mathrm{d}x}(p(x)\frac{\mathrm{d}u}{\mathrm{d}x}) + q(x)u = f(x), \quad x \in (a,b), \tag{1}$$

$$u(a) = 0, \quad u'(b) = 0. \tag{2}$$

其中 $p(x) \in C^1(\bar{I}), p(x) \geq p_{\min} > 0, q \in C(\bar{I}), q(x) \geq 0, f \in H^0(I)$, $\bar{I} = [a,b]$. 等式两边乘以 $v$ 在 $I$ 上积分，并运用分部积分可得

$$a(u,v) = (f,v),$$

其中，

$$a(u,v) = \int_a^b \left( p \frac{\mathrm{d}u}{\mathrm{d}x} \frac{\mathrm{d}v}{\mathrm{d}x} + q\,u\,v \right) \mathrm{d}x, \quad (f,v) = \int_a^b f\,v\,\mathrm{d}x.$$

对两点边值问题 (1)-(2)，在等距网格上（区间个数为 $N$ 个），给出其相应的一次有限元程序。
要求：

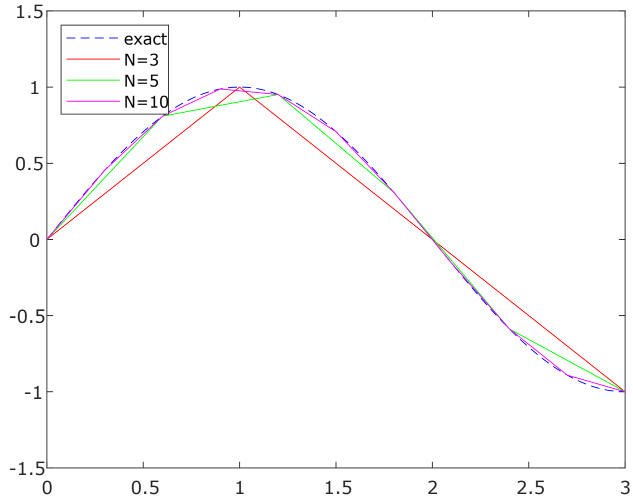
1. 要求网格点个数 $N$ 及区间 $[a,b]$ 在程序中可以变化；
2. 要求程序中可以很容易更换函数 $p, q$ 和 $f$；
3. 先运行 $[0,1]$ 之间，$p=1, q=0, f=\pi^2 sin(\pi x), N=10$ 的情形，输出所有结点的值；

# 2 Theory

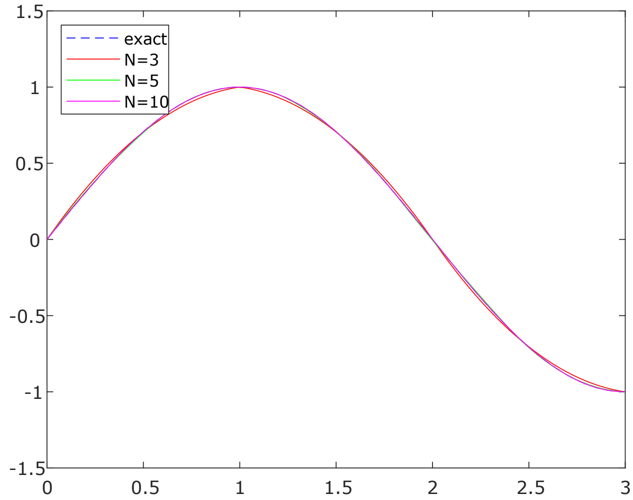
See reference.mlx [1] and error_analysis.mlx [3] in the *src* folder. Our text [2] also has a very detailed analysis on FEM error estimates.

# 3 Solutions

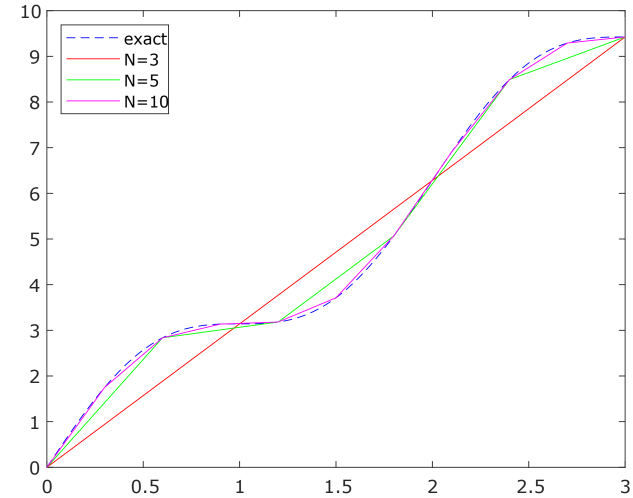
See graphs of those two test problems as follows. View code here.



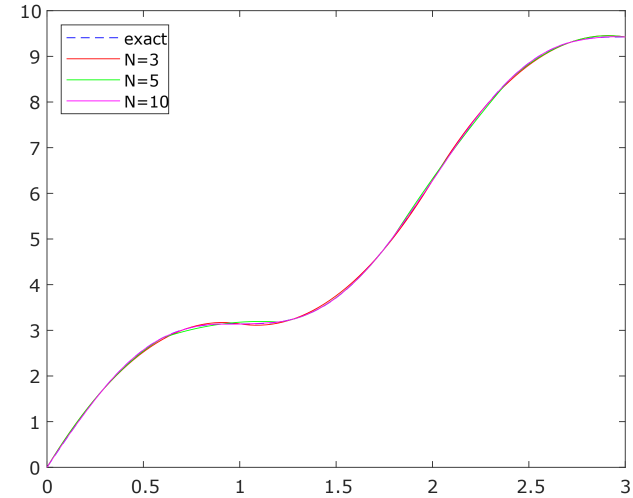

(a) Problem 1 by PLRR




(b) Problem 1 by PQRR




(c) Problem 2 by PLRR




(d) Problem 2 by PQRR

Figure 1: Fittings via PLRR & PQRR

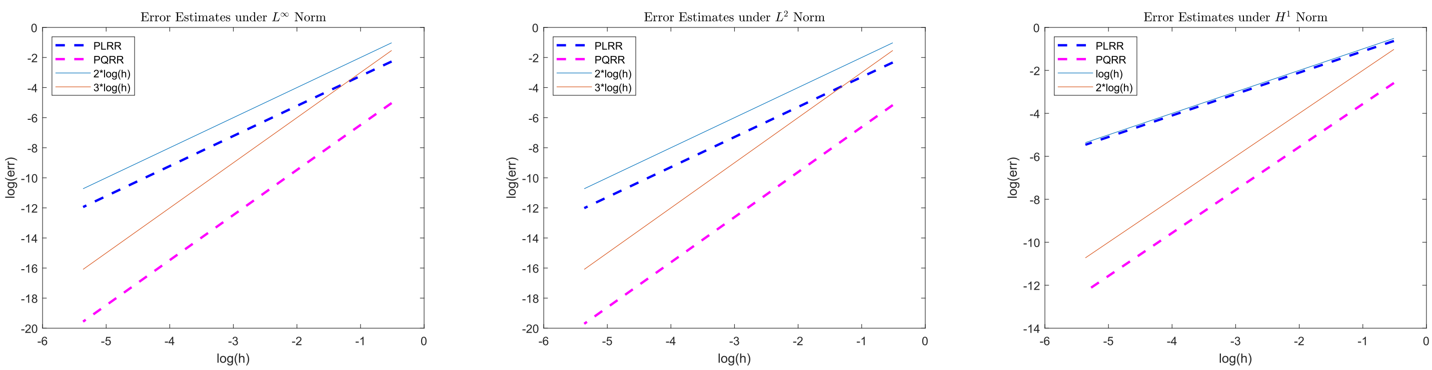



Figure 2: 1-d FEM error estimates under different norms

# 4 Code

## 4.1 Scripts (Main Functions)

```
% ============================================================
% Solve a linear two-point boundary-value problem
%                   -(p(x)u'(x))'+q(x)u(x)=f(x),    a<=x<=b,
% with initial conditions
%                           u(a)=0, u'(b)=0.
% ============================================================

% ************************************************************
% Test 1
isTest1 = true;
f=@(x)pi^2/4*sin(pi/2*x);
u=@(x)sin(pi/2*x);

% Test 2
% isTest1 = false;
% f=@(x)pi^2*sin(pi*x);
% u=@(x)sin(pi*x)+pi*x;
% ************************************************************

p=@(x)x.^0; % p(x)=1
q=@(x)x.*0; % q(x)=0
a=0;
b=3;
% b=1;
N=[3,5,10];
% N=2*[3,5,10]; % compare with quadratic element

% Since the dimensions are different, we can't
% simply use a matrix for storage. List?
uc1=PLRR(f,p,q,a,b,N(1));
uc2=PLRR(f,p,q,a,b,N(2));
uc3=PLRR(f,p,q,a,b,N(3));

% output coef vector of approx solution in command window
disp(uc3)

% plot graphs of exact solution & approximate solutions
t1=linspace(a,b,N(1)+1);
t2=linspace(a,b,N(2)+1);
t3=linspace(a,b,N(3)+1);
tt=linspace(a,b,100+1);

% approx solutions
u1=PLRR_intpol(t1,[0;uc1],tt); % Note that these interpolating points are also on
u2=PLRR_intpol(t2,[0;uc2],tt); % those line segments, which means we can actually
u3=PLRR_intpol(t3,[0;uc3],tt); % skip this phase, and replace with [0;uc] instead.

figure
plot(tt,u(tt),'--b',tt,u1,'r',...
    tt,u2,'g',tt,u3,'m')
legend('exact',sprintf('N=%d',N(1)),sprintf('N=%d',N(2))...
    ,sprintf('N=%d',N(3)),'Location','northwest')

if isTest1 && b==3
    ylim([-1.5,1.5])
end
```

Listing 1: PLRR_test for solving two-point BVP

```
% ============================================================
% Solve a linear two-point boundary-value problem
%                   -(p(x)u'(x))'+q(x)u(x)=f(x),    a<=x<=b,
```

```
% with initial conditions
%                            u(a)=0, u'(b)=0.
% ==============================================================

% **************************************************************
% Test 1
isTest1 = true;
f=@(x)pi^2/4*sin(pi/2*x);
u=@(x)sin(pi/2*x);

% Test 2
% isTest1 = false;
% f=@(x)pi^2*sin(pi*x);
% u=@(x)sin(pi*x)+pi*x;
% **************************************************************

p=@(x)x.^0; % p(x)=1
q=@(x)x.*0; % q(x)=0
a=0;
b=3;
% b=1;
N=[3,5,10];

% Since the dimensions are different, we can't
% simply use a matrix for storage. List?
uc1=PQRR(f,p,q,a,b,N(1));
uc2=PQRR(f,p,q,a,b,N(2));
uc3=PQRR(f,p,q,a,b,N(3));

% output coef vector of approx solution in command window
disp(uc3)

% plot graphs of exact solution & approximate solutions
t1=linspace(a,b,N(1)+1);
t2=linspace(a,b,N(2)+1);
t3=linspace(a,b,N(3)+1);
tt=linspace(a,b,100+1);

% approx solutions
u1=PQRR_intpol(t1,[0;uc1],tt);
u2=PQRR_intpol(t2,[0;uc2],tt);
u3=PQRR_intpol(t3,[0;uc3],tt);

figure
plot(tt,u(tt),'--b',tt,u1,'r',...
    tt,u2,'g',tt,u3,'m')
legend('exact',sprintf('N=%d',N(1)),sprintf('N=%d',N(2))...
    ,sprintf('N=%d',N(3)),'Location','northwest')

if isTest1 && b==3
    ylim([-1.5,1.5])
end
```

Listing 2: PQRR_test for solving two-point BVP

## 4.2 Error Estimates

```
% Error Estimates for PLRR and PQRR
% See error analysis at <error_analysis.mlx>

% (1)  infinity norm --> infinity norm
% ||u-u_h||_{L^{inf}}=sqrt(int((u-u_h)^2,a,b))
% p.w. linear: O(h^2);   cubic spline: O(h^4)
% So we can guess quadratic element is O(h^3)

% (2)  L^2 norm --> L^2 norm
% ||u-u_h||_{L^2}=sqrt(int((u-u_h)^2,a,b))
```

```
% p.w. linear: O(h^2);   cubic spline: O(h^4)
% So we can guess quadratic element is O(h^3)

% (3)  H^1 norm --> L^2 norm
% ||u-u_h||_{H^1}=sqrt(int((u'-u_h')^2,a,b)+int((u-u_h)^2,a,b))
% p.w. linear: O(h);     cubic spline: O(h^3)
% So we can guess quadratic element is O(h^2)


% ==============================================================
% Solve a linear two-point boundary-value problem
%                   -(p(x)u'(x))'+q(x)u(x)=f(x),    a<=x<=b,
% with initial conditions
%                           u(a)=0, u'(b)=0.
% ==============================================================

% **************************************************************
% Test 1
f=@(x)pi^2/4*sin(pi/2*x);
u=@(x)sin(pi/2*x);
u1=@(x)pi/2*cos(pi/2*x); % direvative of u

% Test 2
% f=@(x)pi^2*sin(pi*x);
% u=@(x)sin(pi*x)+pi*x;
% u1=@(x)pi*(1+cos(pi*x)); % direvative of u
% **************************************************************

p=@(x)x.^0; % p(x)=1
q=@(x)x.*0; % q(x)=0
a=0;
b=3;


% err = O(h^n), log(err)=n*log(h), slope k = n, which represents
% its convergence order. So we need to let log(h) be the x axis,
% log(err) be the y axis, and see how the slopes of those norms
% look like.
n=5;
N=8;    % no. of points of log(h)
h=zeros(N,1);
err_inf=zeros(N,2);  % infinity norm
err_L2 = zeros(N,2); % L^2 norm
err_H1 = zeros(N,2); % H^1 norm
for i=1:N
    h(i)=(b-a)/n;
    t =linspace(a,b,n+1).';
    tt=linspace(a,b,5*n+1).'; % interpolation every 5 points

    uc1=PLRR(f,p,q,a,b,n); % try also 2*n parts. See footnote for analysis
    uc2=PQRR(f,p,q,a,b,n);
    [y1, y1_1]=PLRR_intpol(t,[0;uc1],tt);
    [y2, y2_1]=PQRR_intpol(t,[0;uc2],tt);

    e = (u(tt)-y1).^2;
    e1 = (u1(tt)-y1_1).^2;
    err_inf(i,1)=max(abs(u(tt)-y1)); % norm(u(tt)-y1, inf)
    err_L2(i,1)=sqrt(sum(e(1:end-1))*h(i)/5); % approx equal to:
                                              % norm(u(tt)-y1)*sqrt(h(i)/5)
    err_H1(i,1)=sqrt(sum(e(1:end-1)+e1(1:end-1))*h(i)/5);

    e = (u(tt)-y2).^2;
    e1 = (u1(tt)-y2_1).^2;
    err_inf(i,2)=max(abs(u(tt)-y2));
    err_L2(i,2)=sqrt(sum(e(1:end-1))*h(i)/5);
    err_H1(i,2)=sqrt(sum(e(1:end-1))*h(i)/5+sum(e1(1:end-1))*h(i)/5);

```

```
    n=2*n; % delta(log(h)) = -log(2) = -0.6931
end

logh=log(h);
loge_inf=log(err_inf);
loge_L2=log(err_L2);
loge_H1=log(err_H1);

% L^{inf} norm
figure
plot(logh,loge_inf(:,1),'b--',logh,loge_inf(:,2),'m--','LineWidth',2)
hold on
plot(logh,2*logh,logh,3*logh)
legend('PLRR','PQRR','2*log(h)','3*log(h)','Location','northwest')
xlabel('log(h)')
ylabel('log(err)')
title('Error Estimates under $L^\infty$ Norm','interpreter','latex')

% Get convergence orders of these methods (i.e. their slopes)
% Or use built-in func polyfit instead
P1_inf=lsq(logh,loge_inf(:,1),1);     % use linear polynomial to fit, i.e. y = a1*x + a0
    , and P1 = [a0, a1]
k_PLRR_inf = P1_inf(2)                % its convergence order should amount to its slope
                                      % it may not be an integer, but it is pretty
                                      % much close to one, say 1.9961 or whatever

P2_inf=lsq(logh,loge_inf(:,2),1);
k_PQRR_inf = P2_inf(2)

% L^2 norm
figure
plot(logh,loge_L2(:,1),'b--',logh,loge_L2(:,2),'m--','LineWidth',2)
hold on
plot(logh,2*logh,logh,3*logh)
legend('PLRR','PQRR','2*log(h)','3*log(h)','Location','northwest')
xlabel('log(h)')
ylabel('log(err)')
title('Error Estimates under $L^2$ Norm','interpreter','latex')

P1_L2=lsq(logh,loge_L2(:,1),1);
k_PLRR_L2 = P1_L2(2)

P2_L2=lsq(logh,loge_L2(:,2),1);
k_PQRR_L2 = P2_L2(2)

% H^1 norm
figure
plot(logh,loge_H1(:,1),'b--',logh,loge_H1(:,2),'m--','LineWidth',2)
hold on
plot(logh,logh,logh,2*logh)
legend('PLRR','PQRR','log(h)','2*log(h)','Location','northwest')
xlabel('log(h)')
ylabel('log(err)')
title('Error Estimates under $H^1$ Norm','interpreter','latex')

P1_H1=lsq(logh,loge_H1(:,1),1);
k_PLRR_H1 = P1_H1(2)

P2_H1=lsq(logh,loge_H1(:,2),1);
k_PQRR_H1 = P2_H1(2)

%%%%%%%%%%%%% footnote %%%%%%%%%%%%%
% err = O(h^2) = C*h^2
% log(err') = log(C*(h/2)^2)
%           = log(err) - log(4)
% So doubling the no. of intervals
% just makes 'PLRR' translate down,
```

```
% remaining its slope unchanged.
%%%%%%%%%%%%% footnote %%%%%%%%%%%%%
```

Listing 3: Error Estimates for FEM

## 4.3 PLRR and PQRR

These are the exact methods we used for solving this two-point boundary value problem.

```
% Piecewise Linear Rayleigh-Ritz method
% ==============================================================
% Solve a linear two-point boundary-value problem
%                   -(p(x)u'(x))'+q(x)u(x)=f(x),     a<=x<=b,
% with initial conditions
%                           u(a)=0, u'(b)=0.
% ==============================================================
% See the theory part at <reference.mlx>
function uc=PLRR(f,p,q,a,b,N)
% INPUT:
%   f: f(x)
%   p: p(x); optional, by default p(x)=1
%   q: q(x); optional, by default q(x)=0
%   a, b: interval [a,b]; optional, by default [a,b]=[0,1]
%   N: # of evenly spaced intervals; optional, by default N=10
% OUTPUT:
%   uc: coef vector of approx solution

% ==============================================================
% *****  NOTE that f, p, q must be element-wise functions  *****
% e.g. f(x)=@(x)x.^2+1 (NOT x^2+1)
% ==============================================================

% set default args
if nargin<6
    N=10;
    if nargin<5
        b=1;
        if nargin<4
            a=0;
            if nargin<3
                q=@(x)x.*0;
                if nargin<2
                    p=@(x)x.^0;
                    if nargin<1
                        error('Error! Function f is not set.')
                    end
                end
            end
        end
    end
end
if b<a
    tmp=a; a=b; b=tmp;
end

h=(b-a)/N;  % step length. no check for N, hhh
A=zeros(N); % coef matrix, symmetric tridiagonal
c=zeros(N,1); % right-hand side (RHS) vector
xx=linspace(a,b,N+1);

% determine tridiagonal elements of A, and RHS vector c
for i=1:N-1
    % Or use MATLAB func <integral> instead of <Gaussquad>.
    % --> <Ctrl + F> --> Replace All. Then go to PLRR_test
    % script file & run it again, it shall get same graph.
    A(i,i)=Gaussquad(p,xx(i),xx(i+2))+...
        Gaussquad(@(x)(x-xx(i)).^2.*q(x),xx(i),xx(i+1))+...
        Gaussquad(@(x)(xx(i+2)-x).^2.*q(x),xx(i+1),xx(i+2));
```

```
    A(i,i+1)=-Gaussquad(p,xx(i+1),xx(i+2))+...
        Gaussquad(@(x)(x-xx(i+1)).*(xx(i+2)-x).*q(x),xx(i+1),xx(i+2));

    c(i)=Gaussquad(@(x)(x-xx(i)).*f(x),xx(i),xx(i+1))+...
        Gaussquad(@(x)(xx(i+2)-x).*f(x),xx(i+1),xx(i+2));
end
A(N,N)=Gaussquad(p,xx(N),xx(N+1))+...
       Gaussquad(@(x)(x-xx(N)).^2.*q(x),xx(N),xx(N+1));
c(N)=Gaussquad(@(x)(x-xx(N)).*f(x),xx(N),xx(N+1));

A=A+diag(diag(A,1),-1); % let A_{i,i-1}=A_{i-1,i}
A=A/h^2;
c=c/h;

uc = solveTridiag(A,c); % O(n) operations
% uc=A\c; % coef vector of approx solution
% or use user-defined function <GauEli>
% uc=GauEli(A,c);       % O(n^3) operations

end
```

Listing 4: Piecewise Linear Rayleigh-Ritz method

```
% Piecewise Quadratic element Rayleigh-Ritz method
% ===============================================================
% Solve a linear two-point boundary-value problem
%                    -(p(x)u'(x))'+q(x)u(x)=f(x),    a<=x<=b,
% with initial conditions
%                            u(a)=0, u'(b)=0.
% ===============================================================
% See the theory part under folder <quadratic_element_img>
function uc=PQRR(f,p,q,a,b,N)
% INPUT:
%   f: f(x)
%   p: p(x); optional, by default p(x)=1
%   q: q(x); optional, by default q(x)=0
%   a, b: interval [a,b]; optional, by default [a,b]=[0,1]
%   N: # of evenly spaced intervals; optional, by default N=10
% OUTPUT:
%   uc: coef vector of approx solution

% ===============================================================
% *****  NOTE that f, p, q must be element-wise functions  *****
% e.g. f(x)=@(x)x.^2+1 (NOT x^2+1)
% ===============================================================

% set default args
if nargin<6
    N=10;
    if nargin<5
        b=1;
        if nargin<4
            a=0;
            if nargin<3
                q=@(x)x.*0;
                if nargin<2
                    p=@(x)x.^0;
                    if nargin<1
                        error('Error! Function f is not set.')
                    end
                end
            end
        end
    end
end
if b<a
    tmp=a; a=b; b=tmp;
```

```
end

h=(b-a)/N; % step length. no check for N, hhh
A=zeros(2*N);   % coef matrix, symmetric
c=zeros(2*N,1); % right-hand side (RHS) vector
xx=linspace(a,b,N+1);

% k = ( x- x_{j-1} ) / h_j, 0 <= k <=1
% u_h = u_{j-1}*(2k-1)(k-1) + u_{j-1/2}*4k(1-k) + u_j*k(2k-1)
% u_h' = u_{j-1}*(4k-3)/h_j + u_{j-1/2}*(-8k+4)/h_j + u_j*(4k-1)/h_j

% hopefully inlined
phi1 = @(k)(2*k-1).*(k-1);
phi2 = @(k)4*k.*(1-k);
phi3 = @(k)k.*(2*k-1);
k1 = @(k)(4*k-3)/h;
k2 = @(k)(4-8*k)/h;
k3 = @(k)(4*k-1)/h;

% determine tridiagonal elements of A, and RHS vector c
for j = 1 : N
    % Or use MATLAB func <integral> instead of <Gaussquad>.
    % --> <Ctrl + F> --> Replace All. Then go to PQRR_test
    % script file & run it again, it shall get same graph.
    A(2*j-1,2*j-1) = Gaussquad(@(k)(p(k*h+xx(j)).*k2(k).^2+...
        q(k*h+xx(j)).*phi2(k).^2)*h, 0, 1); % omit *h in each item for optimization
    A(2*j-1,2*j) = Gaussquad(@(k)(p(k*h+xx(j)).*k2(k).*k3(k)+...
        q(k*h+xx(j)).*phi2(k).*phi3(k))*h, 0, 1);
    A(2*j,2*j-1) =  A(2*j-1, 2*j ); % symmetric

    A(2*j,2*j) = Gaussquad(@(k)(p(k*h+xx(j)).*k3(k).^2+...
        q(k*h+xx(j)).*phi3(k).^2)*h, 0, 1);
    if(j<N)
        A(2*j,2*j)=A(2*j,2*j)+Gaussquad(@(k)(p(k*h+xx(j+1)).*k1(k).^2+...
            q(k*h+xx(j+1)).*phi1(k).^2)*h, 0, 1);

        A(2*j,2*j+1) = Gaussquad(@(k)(p(k*h+xx(j+1)).*k1(k).*k2(k)+...
            q(k*h+xx(j+1)).*phi1(k).*phi2(k))*h, 0, 1);

        A(2*j,2*j+2) = Gaussquad(@(k)(p(k*h+xx(j+1)).*k1(k).*k3(k)+...
            q(k*h+xx(j+1)).*phi1(k).*phi3(k))*h, 0, 1);

        A(2*j+1,2*j) = A( 2*j ,2*j+1);
        A(2*j+2,2*j) = A( 2*j ,2*j+2);
    end

    c(2*j-1) = Gaussquad(@(k)(f(k*h+xx(j)).*phi2(k))*h, 0, 1);
    c( 2*j ) = Gaussquad(@(k)(f(k*h+xx(j)).*phi3(k))*h, 0, 1);
    if(j<N)
        c(2*j)=c(2*j)+Gaussquad(@(k)(f(k*h+xx(j+1)).*phi1(k))*h, 0, 1);
    end
end

% uc=A\c; % coef vector of approx solution
% or use user-defined function <GauEli>
uc=GauEli(A,c);

end
```

Listing 5: Piecewise Qudratic element Rayleigh-Ritz method global stiffness matrix version

```
% Piecewise Quadratic element Rayleigh-Ritz method
% element stiffness matrix version
% Inspired by Miss White ;)
function u = PQRR_es(f,p,q,a,b,n)

h = (b-a)/n; % step length. no check for N, hhh
```

```
A = zeros(2*n+1);   % coef matrix, symmetric
F = zeros(2*n+1,1); % right-hand side (RHS) vector
x = linspace(a,b,n+1);

% k = ( x- x_{j-1} ) / h_j, 0 <= k <=1
% u_h = u_{j-1}*(2k-1)(k-1) + u_{j-1/2}*4k(1-k) + u_j*k(2k-1)
% u_h' = u_{j-1}*(4k-3)/h_j + u_{j-1/2}*(-8k+4)/h_j + u_j*(4k-1)/h_j

% hopefully inlined
phi1 = @(k)(2*k-1).*(k-1);
phi2 = @(k)4*k.*(1-k);
phi3 = @(k)k.*(2*k-1);
k1 = @(k)(4*k-3)/h;
k2 = @(k)(4-8*k)/h;
k3 = @(k)(4*k-1)/h;

K = zeros(3); % element stiffness matrix
for j = 1 : n
    K(1,1) = integral(@(k)(p(k*h+x(j)).*k1(k).^2+...
        q(k*h+x(j)).*phi1(k).^2)*h, 0, 1); % omit *h in each item for optimization

    K(1,2) = integral(@(k)(p(k*h+x(j)).*k1(k).*k2(k)+...
        q(k*h+x(j)).*phi1(k).*phi2(k))*h, 0, 1);

    K(1,3) = integral(@(k)(p(k*h+x(j)).*k1(k).*k3(k)+...
        q(k*h+x(j)).*phi1(k).*phi3(k))*h, 0, 1);

    K(2,2) = integral(@(k)(p(k*h+x(j)).*k2(k).^2+...
        q(k*h+x(j)).*phi2(k).^2)*h, 0, 1);

    K(2,3) = integral(@(k)(p(k*h+x(j)).*k2(k).*k3(k)+...
        q(k*h+x(j)).*phi2(k).*phi3(k))*h, 0, 1);

    K(3,3) = integral(@(k)(p(k*h+x(j)).*k3(k).^2+...
        q(k*h+x(j)).*phi3(k).^2)*h, 0, 1);

    % using symmetricity
    K(2,1) = K(1,2);
    K(3,1) = K(1,3);
    K(3,2) = K(2,3);

    % assemble the element stiffness matrix
    A(2*j-1,2*j-1) = A(2*j-1,2*j-1) + K(1,1);
    A(2*j-1, 2*j ) = A(2*j-1, 2*j ) + K(1,2);
    A(2*j-1,2*j+1) = A(2*j-1,2*j+1) + K(1,3);
    A( 2*j ,2*j-1) = A( 2*j ,2*j-1) + K(2,1);
    A( 2*j , 2*j ) = A( 2*j , 2*j ) + K(2,2);
    A( 2*j ,2*j+1) = A( 2*j ,2*j+1) + K(2,3);
    A(2*j+1,2*j-1) = A(2*j+1,2*j-1) + K(3,1);
    A(2*j+1, 2*j ) = A(2*j+1, 2*j ) + K(3,2);
    A(2*j+1,2*j+1) = A(2*j+1,2*j+1) + K(3,3);

    % calculate & (implicitly) assemble right-hand side vector
    F(2*j-1) = F(2*j-1) + integral(@(k)(f(k*h+x(j)).*phi1(k))*h, 0, 1);
    F( 2*j ) = F( 2*j ) + integral(@(k)(f(k*h+x(j)).*phi2(k))*h, 0, 1);
    F(2*j+1) = F(2*j+1) + integral(@(k)(f(k*h+x(j)).*phi3(k))*h, 0, 1);
end

A = A(2:2*n+1,2:2*n+1);
F = F(2:2*n+1);

u = A\F;
end
```

Listing 6: Piecewise Qudratic element Rayleigh-Ritz method element stiffness matrix version

## 4.4 PLRR_intpol and PQRR_intpol

Interpolations for PLRR and PQRR.

```
% Interpolation for PLRR
function [u, u1]=PLRR_intpol(xx,uc,xq)
N = length(xx)-1;
m = length(xq);
h = (xx(end)-xx(1))/N; % evenly spaced step length
u = zeros(m,1);
u1 = zeros(m,1); % derivative of u

% a lazzy solution
%{
for i=1:N
    for j=1:m
        if xq(j)>=xx(i) && xq(j)<=xx(i+1)
            u(j)=uc(i)*(xx(i+1)-xq(j))/h+uc(i+1)*(xq(j)-xx(i))/h;
            u1(j)=(uc(i+1)-uc(i))/h;
        end
    end
end
%}

% When xq is an *ordered* further division based on xx,
% for example, xq = linspace(a,b,10*N+1), then an eager
% version can be developed like this:
% {
k = (m-1)/N;
for i=1:N
    % Note that, we are better off using floor, since floor(k * N)
    % might NOT be equal to m-1, in turn, xq(m) remains what it is.
    k1 = 1 + ceil(k*(i-1)); % Get upper bounds, at most 1 element of
    k2 = 1 + ceil(k*i);     % xq(k1:k2) NOT in [xx(i), xx(i+1)].
    for j=k1:k2
        if xq(j)>=xx(i) && xq(j)<=xx(i+1)
            u(j)=uc(i)*(xx(i+1)-xq(j))/h+uc(i+1)*(xq(j)-xx(i))/h;
            u1(j)=(uc(i+1)-uc(i))/h;
        end
    end
end
%}

% **********************FOOTNOTE*************************
% Say, m=101, N=3, floor(k*N)=100?? Luckily, this is true in
% MATLAB (C, C++, Python, etc.) due to the fact that:
% >> k=100/3; % k = 33.333333333333336
% >> k*3==100
% ans = (logical) 1
% That's how double-precision numbers store and operate.
% Notice that floor(33.3333*3) = 99, NOT 100.
% **********************FOOTNOTE*************************
end
```

Listing 7: Interpolation for PLRR

```
% Interpolation for PQRR
function [u, u1]=PQRR_intpol(xx,uc,xq)
N=(length(uc)-1)/2;  % No. of coefs of integer points
m=length(xq);        % No. of query points
h=(xx(end)-xx(1))/N; % evenly spaced step length
u=zeros(m,1);        % approximations at these m query points
u1=zeros(m,1);       % derivative of u

phi1 = @(k)(2*k-1).*(k-1);
phi2 = @(k)4*k.*(1-k);
phi3 = @(k)k.*(2*k-1);
```

```
kk1 = @(k)(4*k-3)/h;
kk2 = @(k)(4-8*k)/h;
kk3 = @(k)(4*k-1)/h;

% a lazzy solution
%{
for i=1:N
    for j=1:m
        if xq(j)>=xx(i) && xq(j)<=xx(i+1)
            u(j)=uc(2*i)*phi2((xq(j)-xx(i))/h)+...  % central part
                uc(2*i+1)*phi3((xq(j)-xx(i))/h)+... % left half of current
                uc(2*i-1)*phi1((xq(j)-xx(i))/h);    % right half of previous

            u1(j)=uc(2*i)*kk2((xq(j)-xx(i))/h)+...
                uc(2*i+1)*kk3((xq(j)-xx(i))/h)+...
                uc(2*i-1)*kk1((xq(j)-xx(i))/h);
        end
    end
end
%}

% When xq is an *ordered* further division based on xx,
% for example, xq = linspace(a,b,10*N+1), then an eager
% version can be developed like this:
% {
k = (m-1)/N;
for i=1:N
    % Note that, we are better off using floor, since floor(k * N)
    % might NOT be equal to m-1, in turn, xq(m) remains what it is.
    k1 = 1 + ceil(k*(i-1)); % Get upper bounds, at most 1 element of
    k2 = 1 + ceil(k*i);     % xq(k1:k2) NOT in [xx(i), xx(i+1)].
    for j=k1:k2
        if xq(j)>=xx(i) && xq(j)<=xx(i+1)
            u(j)=uc(2*i)*phi2((xq(j)-xx(i))/h)+...  % central part
                uc(2*i+1)*phi3((xq(j)-xx(i))/h)+... % left half of current
                uc(2*i-1)*phi1((xq(j)-xx(i))/h);    % right half of previous

            u1(j)=uc(2*i)*kk2((xq(j)-xx(i))/h)+...
                uc(2*i+1)*kk3((xq(j)-xx(i))/h)+...
                uc(2*i-1)*kk1((xq(j)-xx(i))/h);
        end
    end
end
%}

% ***********************FOOTNOTE**************************
% Say, m=101, N=3, floor(k*N)=100?? Luckily, this is true in
% MATLAB (C, C++, Python, etc.) due to the fact that:
% >> k=100/3; % k = 33.333333333333336
% >> k*3==100
% ans = (logical) 1
% That's how double-precision numbers store and operate.
% Notice that floor(33.3333*3) = 99, NOT 100.
% ***********************FOOTNOTE**************************
end
```

Listing 8: Interpolation for PQRR

## 4.5 Gaussquad

This is a usr-defined function that servers the purpose of numerical integration, which can be simply identically substituted by MATLAB build-in function *integral*.

```
% Gaussian quadrature
function T = Gaussquad(f,a,b,n)
if nargin<4
```

```
    n=8;
end
% Gaussian quarature points & weights on [-1,1]
if n == 2
    w = [1,1];                          % weights
    p = [-1/sqrt(3),1/sqrt(3)];         % points
elseif n == 4
    w = [0.3478548451,0.3478548451,0.6521451549,0.6521451549];
    p = [0.8611363116,-0.8611363116,0.3399810436,-0.3399810436];
elseif n == 8
    w = [0.1012285363,0.1012285363,0.2223810345,0.2223810345,0.3137066459,...
           0.3137066459,0.3626837834,0.3626837834];
    p = [0.9602898565,-0.9602898565,0.7966664774,-0.7966664774,0.5255324099,...
      -0.5255324099,0.1834346425,-0.1834346425];
else
    error('n must be either 2 or 4 or 8')
end

% take a linear transform: x = t*(b-a)/2 + (a+b)/2, where t belongs to [-1,1]
% int(f,[a,b]) = (b-a)/2 * int(f(t*(b-a)/2 + (a+b)/2), [-1,1])
w = 0.5*(b-a)*w;
p = 0.5*(b-a)*p+0.5*(a+b);

T = w*f(p).';
end
```

Listing 9: Gaussian quadrature

## 4.6 GauEli

A routine for solving linear equations of the form $Ax = b$.

```
% Gaussian Elimination: solve linear systems
% of algebraic equations of the form Ax=b.
% 10170437 Mark Taylor

function [x,U] = GauEli(A, b)

[m,n]=size(A);
if m ~= n
    error('A must be a square matrix!')
end

U=[A,b];        % Augemented matrix of A

for j=1:n
    k=maxIndex(U(:,j),j,n);
    if abs(U(k,j))>eps
        if k~=j
            temp=U(j,j:n+1);
            U(j,j:n+1)=U(k,j:n+1);
            U(k,j:n+1)=temp;
        end
    else
        x='The system has infinitely many solutions!';
        U=NaN;
        return;
    end

    % Perform Gauss elimination.
    for i=j+1:n
        if abs(U(i,j))>eps
            t=U(i,j)/U(j,j);
            U(i,j:n+1)=U(i,j:n+1)-t*U(j,j:n+1);
        end
    end
```

```
end

x=zeros(n,1);
% Sovle upper diagonal linear system, i.e. U(1:n,1:n)x=U(:,n+1).
x(n)=U(n,n+1)/U(n,n);
for i=n-1:-1:1
    x(i)=(U(i,n+1)-U(i,i+1:n)*x(i+1:n))/U(i,i);
end

end
```

Listing 10: Gaussian Elimination

## References


[1] R. L. Burden and J. D. Faires. *Numerical Analysis*. Cengage Learning, 9th edition, 2010.

[2] L. Ronghua. *Numerical Methods for Partial Differential Equations*. Higher Education Press, 2th edition, 2010.

[3] C. University. Chapter 5: Error estimates for the finite element method. page 4, 2002. URL http://pi.math.cornell.edu/~demlow/425/chap5.pdf.


## Appendices

Hello from the Beatles.😃