```
    1: import numpy as np
    2: import matplotlib.pyplot as plt
    3: import scipy.integrate as integrate
    4:
    5: # Basis functions
    6: # ---- TODO: order p? ----
    7: def phi(k, mesh, x):
    8:     if x > mesh[k-1] and x < mesh[k]:
    9:         return (x-mesh[k-1]) / (mesh[k] - mesh[k-1])
   10:     elif x > mesh[k] and x < mesh[k+1]:
   11:         return (mesh[k+1]-x) / (mesh[k+1] - mesh[k])
   12:     elif x == mesh[k]:
   13:         return 1
   14:     else:
   15:         return 0
   16: def phi_prime(k, mesh, x):
   17:     if x > mesh[k-1] and x < mesh[k]:
   18:         return 1 / (mesh[k] - mesh[k-1])
   19:     elif x > mesh[k] and x < mesh[k+1]:
   20:         return -1 / (mesh[k+1] - mesh[k])
   21:     elif x == mesh[k]:
   22:         return print("oh no")
   23:     else:
   24:         return 0
   25:
   26: # vertex flux jump between elements
   27: def flux_jumps(mesh, u):
   28:     m = len(mesh)
   29:
   30:     r = (u[2:]-u[1:-1]) / (mesh[2:]-mesh[1:-1])
   31:     l = (u[1:-1]-u[:-2]) / (mesh[1:-1]-mesh[:-2])
   32:     jumps = np.zeros(m)                          # 0 jump at bnd?
   33:     jumps[1:-1] = r - l
   34:
   35:     jumps[0] = jumps[1]                          # or same jump at bnd?
   36:     jumps[-1] = jumps[-2]                        # or same jump at bnd?
   37:     return jumps
   38:
   39: # h-adaptivity (refine neighboring elments, if flux jump over certain threshold)
   40: def adapt_h(mesh, jumps):
   41:     new_mesh = [mesh[0]]
   42:     jumps = jumps[1:-1]                          # only interior jumps (excluding bnd no
des needed for De Boor)
   43:
   44:     # define threshold
   45:     threshold = 0.5 * np.max(np.abs(jumps))
   46:     # mark elements
   47:     marked_nodes = np.abs(jumps) > threshold
   48:     marked_el = np.zeros(len(mesh)-1, dtype=bool)
   49:     for i, refine in enumerate(marked_nodes):
   50:         if refine:
   51:             marked_el[i] = True
   52:             marked_el[i+1] = True
   53:
   54:     # make new mesh
   55:     for k, refine in enumerate(marked_el):
   56:         l = mesh[k]
   57:         r = mesh[k+1]
   58:         if refine:
   59:             new_mesh.extend(np.linspace(l, r, 3)[1:])
   60:         else:
   61:             new_mesh.append(r)
   62:
   63:     return np.asarray(new_mesh, dtype=float)
   64:
   65: # r-adaptivity (one iteration of De Boor's algorithm, moving mesh nodes for equidist
ributing mesh)
   66: def adapt_r(mesh, rho):
```

```
    m = len(mesh)

    p = 0.5 * (rho[:-1] + rho[1:])                  # piecewise constant function on
 elements

    P = np.zeros(m)
    for i in range(1,m):
        P[i] = P[i-1] + (mesh[i]-mesh[i-1])*p[i-1]  # approx integral of p, from nod
e 0 to node i
    Pb = P[-1]                                      # integral of p, over whole mesh

    new_mesh = mesh.copy()
    for j in range(1, m-1):
        xi_j = (j)/(m-1)
        k = np.searchsorted(P, xi_j*Pb)             # search k s.t.: P[node k-1] < x
i_j*Pb < P[node k]
        k = max(k,1)                                # if k=0
        new_mesh[j] = mesh[k-1] + 2*(xi_j*Pb - P[k-1]) / (rho[k-1]+rho[k])    # new
node  j
    return new_mesh

```

```python
import numpy as np
from adapt import *

# PDE:
#  -u''(x) = f(x)        x in (-1,1)
#    u(-1) = -arctan(p)
#    u'(1) = p / (p^2 + 1)
#
# weak form
# int u'v' dx = int f(x) * v(x) dx + p/(p^2+1) * v(1)
#

# rhs
def f(x):
    return 2 * p**3 * x / (p**2 * x**2 + 1)**2

# Stiffness and Load
def K_loc(k, mesh):
    K_loc = np.zeros((2,2))
    K_loc[0,0] = integrate.quad(lambda x: phi_prime(k-1, mesh, x)**2, mesh[k-1], mesh[k])[0]
    K_loc[1,0] = integrate.quad(lambda x: phi_prime(k-1, mesh, x)*phi_prime(k, mesh, x), mesh[k-1], mesh[k])[0]
    K_loc[0,1] = integrate.quad(lambda x: phi_prime(k, mesh, x)*phi_prime(k-1, mesh, x), mesh[k-1], mesh[k])[0]
    K_loc[1,1] = integrate.quad(lambda x: phi_prime(k, mesh, x)**2, mesh[k-1], mesh[k])[0]
    return K_loc
def F_loc(k, mesh):
    F_loc = np.zeros(2)
    F_loc[0] = integrate.quad(lambda x: f(x) * phi(k-1, mesh, x), mesh[k-1], mesh[k])[0]
    F_loc[1] = integrate.quad(lambda x: f(x) * phi(k, mesh, x), mesh[k-1], mesh[k])[0]
    return F_loc

# Assembling
def Assemble(mesh):
    m = len(mesh)
    n = m-1

    K = np.zeros((m,m))
    F = np.zeros(m)
    for k in range(1,m):
        K[k-1:k+1,k-1:k+1] += K_loc(k, mesh)
        F[k-1:k+1] += F_loc(k, mesh)

    # Boundary conditions
    # Dirichlet: u(-1) = -arctan(p)
    K[0,:] = 0
    K[0,0] = 1
    F[0]   = -np.arctan(p)
    # Neumann: u'(1) = p / (p^2 + 1)
    F[-1] += p/(p**2+1) * phi(n, mesh, 1)
    return K,F

def plotting(mesh, u, comment):
    exact_x = np.linspace(-1,1,1000)
    exact = np.arctan(p*exact_x)
    plt.plot(exact_x, exact, "--", linewidth=1, color="red", label="exact")
    plt.title(f"p = {p} | n = {n} | {comment}")
    plt.xlabel("x")
    plt.ylabel("u(x)")
    plt.plot(mesh, u, "-o", label="u_h")
    plt.xticks(mesh, labels=[])
    plt.legend()
    plt.grid(True)
    plt.tight_layout()
```

```
    plt.savefig("task_a.png", dpi=300)
    plt.show()
    return 0

##############################################################################

# parameters
p_list = [5,10,20,100]
for p in p_list:

    # h-adaptivity
    mesh = np.array([-1.0, -0.2, 0, 0.7, 1.0])            # with 0 as node
    # mesh = np.array([-1.0,-0.131,0.372,1.0])              # without 0 as node

    # r-adaptivity
    # mesh = np.linspace(-1, 1, 11)                         # with 0 as node
    # mesh = np.linspace(-1, 1, 10)                         # without 0 as node

    m = len(mesh)
    n = m-1

    K, F = Assemble(mesh)                                   # assemble
    u = np.linalg.solve(K, F)                               # solve
    jumps = flux_jumps(mesh, u)                             # flux jumps
    plotting(mesh, u, "before adapting")                    # plotting

    iterations = 6
    for it in range(iterations):
        mesh = adapt_h(mesh, jumps)                         # h-adaptivity
        # mesh = adapt_r(mesh, np.abs(jumps))                 # r-adaptivity (positive density (jumps)!)
        m = len(mesh)
        n = m-1

        K, F = Assemble(mesh)                               # assemble
        u = np.linalg.solve(K, F)                           # solve
        jumps = flux_jumps(mesh, u)                         # flux jumps
        # plotting(mesh, u, f"iteration {it+1}")              # plotting each iteration

    # print(jumps)
    plotting(mesh, u, f"after {iterations} iterations")     # plotting
```

```
import numpy as np
from adapt import *

# PDE:
#  -(lambda(x)u'(x))' = 0          x in (0,1)
#                u(0) = 0
#                u(1) = 1
#           lambda(x) = |  1      x in (0,1/sqrt(2))
#                       | 10      x in (1/sqrt(2),1)

# rhs
def f(x):
    return 0

def lam(x):
    if x >= 0 and x <= 1/np.sqrt(2):
        return 1
    elif x <= 1 and x > 1/np.sqrt(2):
        return 10
    else:
        return print("lambda undefined")

# Stiffness and Load
def K_loc(k, mesh):
    K_loc = np.zeros((2,2))
    K_loc[0,0] = integrate.quad(lambda x: lam(x)*phi_prime(k-1, mesh, x)**2, mesh[k-1], mesh[k])[0]
    K_loc[1,0] = integrate.quad(lambda x: lam(x)*phi_prime(k-1, mesh, x)*phi_prime(k, mesh, x), mesh[k-1], mesh[k])[0]
    K_loc[0,1] = integrate.quad(lambda x: lam(x)*phi_prime(k, mesh, x)*phi_prime(k-1, mesh, x), mesh[k-1], mesh[k])[0]
    K_loc[1,1] = integrate.quad(lambda x: lam(x)*phi_prime(k, mesh, x)**2, mesh[k-1], mesh[k])[0]
    return K_loc
def F_loc(k, mesh):
    F_loc = np.zeros(2)
    F_loc[0] = integrate.quad(lambda x: f(x) * phi(k-1, mesh, x), mesh[k-1], mesh[k])[0]
    F_loc[1] = integrate.quad(lambda x: f(x) * phi(k, mesh, x), mesh[k-1], mesh[k])[0]
    return F_loc

# Assembling
def Assemble(mesh):
    m = len(mesh)
    n = m-1

    K = np.zeros((m,m))
    F = np.zeros(m)
    for k in range(1,m):
        K[k-1:k+1,k-1:k+1] += K_loc(k, mesh)
        F[k-1:k+1] += F_loc(k, mesh)

    # Boundary conditions
    # Dirichlet: u(0) = 0
    K[0,:] = 0
    K[0,0] = 1
    F[0]   = 0
    # Dirichlet: u(1) = 1
    K[-1,:] = 0
    K[-1,-1] = 1
    F[-1]   = 1
    return K,F

def plotting(mesh, u, comment):
    exact_x = [0, 1/np.sqrt(2), 1]
    exact = [0, 10/(np.sqrt(2)+9), 1]
    plt.plot(exact_x, exact, "--", linewidth=1, color="red", label="exact")
```

```
    63:     plt.title(f"n = {n} | {comment}")
    64:     plt.xlabel("x")
    65:     plt.ylabel("u(x)")
    66:     plt.plot(mesh, u, "-o", label="u_h")
    67:     plt.xticks(mesh, labels=[])
    68:     plt.legend()
    69:     plt.grid(True)
    70:     plt.tight_layout()
    71:     plt.savefig("task_b.png", dpi=300)
    72:     plt.show()
    73:     return 0
    74:
    75: ###############################################################################
    76:
    77: mesh = np.linspace(0, 1, 10)
    78: m = len(mesh)
    79: n = m-1
    80:
    81: K, F = Assemble(mesh)                                  # assemble
    82: u = np.linalg.solve(K, F)                              # solve
    83: jumps = flux_jumps(mesh, u)                            # flux jumps
    84: plotting(mesh, u, "before adapting")                   # plotting
    85:
    86: iterations = 3
    87: for it in range(iterations):
    88:     # mesh = adapt_h(mesh, jumps)                       # h-adaptivity
    89:     mesh = adapt_r(mesh, np.abs(jumps))               # r-adaptivity (positive densi
ty (jumps)!)
    90:     m = len(mesh)
    91:     n = m-1
    92:
    93:     K, F = Assemble(mesh)                             # assemble
    94:     u = np.linalg.solve(K, F)                         # solve
    95:     jumps = flux_jumps(mesh, u)                       # flux jumps
    96:     # plotting(mesh, u, f"iteration {it+1}")            # plotting each iteration
    97:
    98: # print(jumps)
    99: plotting(mesh, u, f"after {iterations} iterations")   # plotting
   100:
```

```
    1: import numpy as np
    2: from adapt import *
    3:
    4: # Peclet problem:
    5: #  -u''(x) + pu'(x) = 0    x in (0,1)
    6: #              u(0) = 0
    7: #              u(1) = 1
    8:
    9: # rhs
   10: def f(x):
   11:     return 0
   12:
   13: # Stiffness and Load
   14: def K_loc(k, mesh):
   15:     K_loc = np.zeros((2,2))
   16:     K_loc[0,0] = integrate.quad(lambda x: phi_prime(k-1, mesh, x)**2, mesh[k-1], mes
h[k])[0]
   17:     K_loc[1,0] = integrate.quad(lambda x: phi_prime(k-1, mesh, x)*phi_prime(k, mesh,
 x), mesh[k-1], mesh[k])[0]
   18:     K_loc[0,1] = integrate.quad(lambda x: phi_prime(k, mesh, x)*phi_prime(k-1, mesh,
 x), mesh[k-1], mesh[k])[0]
   19:     K_loc[1,1] = integrate.quad(lambda x: phi_prime(k, mesh, x)**2, mesh[k-1], mesh[
k])[0]
   20:
   21:     K_loc[0,0] += p*integrate.quad(lambda x: phi_prime(k-1, mesh, x) * phi(k-1, mesh
, x), mesh[k-1], mesh[k])[0]
   22:     K_loc[1,0] += p*integrate.quad(lambda x: phi_prime(k-1, mesh, x) * phi(k, mesh,
x), mesh[k-1], mesh[k])[0]
   23:     K_loc[0,1] += p*integrate.quad(lambda x: phi_prime(k, mesh, x) * phi(k-1, mesh,
x), mesh[k-1], mesh[k])[0]
   24:     K_loc[1,1] += p*integrate.quad(lambda x: phi_prime(k, mesh, x) * phi(k, mesh, x)
, mesh[k-1], mesh[k])[0]
   25:     return K_loc
   26: def F_loc(k, mesh):
   27:     F_loc = np.zeros(2)
   28:     F_loc[0] = integrate.quad(lambda x: f(x) * phi(k-1, mesh, x), mesh[k-1], mesh[k]
)[0]
   29:     F_loc[1] = integrate.quad(lambda x: f(x) * phi(k, mesh, x), mesh[k-1], mesh[k])[
0]
   30:     return F_loc
   31:
   32: # Assembling
   33: def Assemble(mesh):
   34:     m = len(mesh)
   35:     n = m-1
   36:
   37:     K = np.zeros((m,m))
   38:     F = np.zeros(m)
   39:     for k in range(1,m):
   40:         K[k-1:k+1,k-1:k+1] += K_loc(k, mesh)
   41:         F[k-1:k+1] += F_loc(k, mesh)
   42:
   43:     # Boundary conditions
   44:     # Dirichlet: u(0) = 0
   45:     K[0,:] = 0
   46:     K[0,0] = 1
   47:     F[0]   = 0
   48:     # Dirichlet: u(1) = 1
   49:     K[-1,:] = 0
   50:     K[-1,-1] = 1
   51:     F[-1]   = 1
   52:     return K,F
   53:
   54: def plotting(mesh, u, comment):
   55:     exact_x = np.linspace(0,1,1000)
   56:     exact = (np.exp(p*exact_x)-1)/(np.exp(p)-1)
   57:     plt.plot(exact_x, exact, "--", linewidth=1, color="red", label="exact")
   58:     plt.title(f"p = {p} | n = {n} | {comment}")
```

```
    59:     plt.xlabel("x")
    60:     plt.ylabel("u(x)")
    61:     plt.plot(mesh, u, "-o", label="u_h")
    62:     plt.xticks(mesh, labels=[])
    63:     plt.legend()
    64:     plt.grid(True)
    65:     plt.tight_layout()
    66:     plt.savefig("task_c.png", dpi=300)
    67:     plt.show()
    68:     return 0
    69:
    70: ###############################################################################
    71:
    72: # parameters
    73: p_list = [-70, 70]
    74: for p in p_list:
    75:     mesh = np.linspace(0, 1, 30)
    76:     m = len(mesh)
    77:     n = m-1
    78:
    79:     K, F = Assemble(mesh)                                   # assemble
    80:     u = np.linalg.solve(K, F)                               # solve
    81:     jumps = flux_jumps(mesh, u)                             # flux jumps
    82:     plotting(mesh, u, "before adapting")                    # plotting
    83:
    84:     # NOTE: mesh very different every iteration for adapt_r()
    85:     iterations = 21
    86:     for it in range(iterations):
    87:         # mesh = adapt_h(mesh, jumps)                         # h-adaptivity
    88:         mesh = adapt_r(mesh, np.abs(jumps))                 # r-adaptivity (positive d
ensity (jumps)!)
    89:         m = len(mesh)
    90:         n = m-1
    91:
    92:         K, F = Assemble(mesh)                               # assemble
    93:         u = np.linalg.solve(K, F)                           # solve
    94:         jumps = flux_jumps(mesh, u)                         # flux jumps
    95:         # plotting(mesh, u, f"iteration {it+1}")              # plotting each iteratio
n
    96:
    97:     # print(jumps)
    98:     plotting(mesh, u, f"after {iterations} iterations")     # plotting
    99:
   100:
```