Stage 1 — Derivation of Finite Difference Scheme

To derive a **second-order accurate Finite Difference (FD)** scheme for the **2D steady-state heat equation** with **constant thermal conductivity** λ.
Assume **Dirichlet boundary conditions** are applied on all sides of the domain.

The task:

1. Start from the continuous governing equation
   $$ \frac{\partial}{\partial x}!\left(\lambda \frac{\partial T}{\partial x}\right)
   - \frac{\partial}{\partial y}!\left(\lambda \frac{\partial T}{\partial y}\right) = 0
   $$
2. Apply **central differencing** to discretize spatial derivatives.
3. The discrete equation is presented below:

SOLUTION

$$\frac{\lambda}{\Delta x^2}\left(T_{i+1,j} - 2T_{i,j} + T_{i-1,j}\right) \; + \; \frac{\lambda}{\Delta y^2}\left(T_{i,j+1} - 2T_{i,j} + T_{i,j-1}\right) \; = \; 0.$$

Stage 2 — Implementation of 2D Steady-State Heat Conduction Solver

Now, to implement a **2D steady-state heat conduction solver** with **Dirichlet, Robin and Neumann boundary conditions**.
The solver should assemble and solve the linear system corresponding to the discretized finite difference equations derived in **Stage 1**.

Note that: The corner nodes are assigned as follows:

SouthWest, NorthWest -> West

SouthEast, NorthEast -> East

In [91]:

```python
import numpy as np
import matplotlib.pyplot as plt


class Source:
    """Lightweight container for a single point source."""
    def __init__(self, location, is_active, q_dot):
        self.location = location          # (x, y) in physical coords
        self.is_active = is_active        # toggle source on/off
        self.q_dot = q_dot                # intensity (interpreted same as your code)


class SteadyHeat2D:
    """
    2D steady-state heat conduction on a uniform Cartesian grid.

    Discrete operator uses the -∇² convention in the interior:
        (2/dx² + 2/dy²) T_P
      - (T_E + T_W)/dx²
      - (T_N + T_S)/dy² = RHS

    Boundary rows are overwritten by the corresponding BC setter.
    """
    # -------------------- init & utilities --------------------
    def __init__(self, Lx, Ly, dimX, dimY, source, k=1.0):
        # domain & grid
        self.l = float(Lx)
        self.h = float(Ly)
        self.dimX = int(dimX)
        self.dimY = int(dimY)
        if self.dimX < 2 or self.dimY < 2:
            raise ValueError("Need at least 2 nodes in each direction.")

        self.dx = self.l / (self.dimX - 1)
        self.dy = self.h / (self.dimY - 1)

        # material
        self.k = float(k)

        # source
        self.source_location = tuple(source.location)
        self.source_active = bool(source.is_active)
        self.source_strength = float(source.q_dot)

        # system (lazy-assembled)
        self.A = None
        self.b = None
        self.T = None

    def _index(self, i, j):
        """(i, j) -> linear index."""
        return j * self.dimX + i

    def _clear_row(self, row):
        """Zero a row for both dense and LIL sparse matrices."""
        if hasattr(self.A, "rows"):  # scipy.sparse.lil_matrix
            self.A.rows[row] = []
            self.A.data[row] = []
        else:                        # dense numpy
            self.A[row, :] = 0.0

    def _ensure_system(self):
        if self.A is None or self.b is None:
            self.set_inner()

    # -------------------- assembly --------------------
    def set_inner(self):
        """
        Build interior stencil and identity rows on boundaries.
        Also adds the source contribution (kept same interpretation).
        """
```

```python
        try:
            from scipy.sparse import lil_matrix
        except Exception:
            lil_matrix = None

        Nx, Ny = self.dimX, self.dimY
        dx, dy, k = self.dx, self.dy, self.k
        N = Nx * Ny

        A = lil_matrix((N, N), dtype=float) if lil_matrix else np.zeros((N, N), dtype=float)
        b = np.zeros(N, dtype=float)

        ax = 1.0 / (dx * dx)
        ay = 1.0 / (dy * dy)

        # stencil for interior, identity on edges
        for j in range(Ny):
            for i in range(Nx):
                n = self._index(i, j)
                if 0 < i < Nx - 1 and 0 < j < Ny - 1:
                    A[n, n] = 2.0 * ax + 2.0 * ay
                    A[n, self._index(i + 1, j)] = -ax
                    A[n, self._index(i - 1, j)] = -ax
                    A[n, self._index(i, j + 1)] = -ay
                    A[n, self._index(i, j - 1)] = -ay
                    b[n] = 0.0
                else:
                    A[n, n] = 1.0
                    b[n] = 0.0

        # ----- point source: patch distribution (same concept preserved) -----
        if self.source_active and self.source_location is not None:
            x0, y0 = self.source_location

            # map physical -> nearest grid index
            i0 = int(round(x0 / dx))
            j0 = int(round(y0 / dy))
            i0 = np.clip(i0, 0, Nx - 1)
            j0 = np.clip(j0, 0, Ny - 1)

            r = 2  # half-width in cells
            patch = [
                (ii, jj)
                for jj in range(j0 - r, j0 + r + 1)
                for ii in range(i0 - r, i0 + r + 1)
                if 0 <= ii < Nx and 0 <= jj < Ny
            ]

            Q = float(self.source_strength)  # kept as in your code
            # keep same formula you used (interpreting q_dot as total power per area basis)
            share = (Q / (dx * dy) / k) / len(patch)

            for ii, jj in patch:
                b[self._index(ii, jj)] += share

        self.A = A
        self.b = b

    # -------------------- boundary conditions --------------------
    def set_south(self, bc_type, T_d=0.0, q=0.0, alpha=0.0, T_inf=0.0):
        """
        Bottom edge (j=0). Uses your original first-order one-sided forms.
        """
        self._ensure_system()
        Nx, Ny, dy, k = self.dimX, self.dimY, self.dy, self.k

        def idx(i, j): return self._index(i, j)

        for i in range(Nx):
            n = idx(i, 0)
            self._clear_row(n)

            bct = bc_type.upper()
            if bct == 'D':
                self.A[n, n] = 1.0
                self.b[n] = T_d
            elif bct == 'N':
                self.A[n, n] = -k / dy
                if Ny > 1:
                    self.A[n, idx(i, 1)] = k / dy
                self.b[n] = q
            elif bct == 'R':
                self.A[n, n] = (k / dy) + alpha
                if Ny > 1:
                    self.A[n, idx(i, 1)] = -k / dy
                self.b[n] = alpha * T_inf
            else:
                raise ValueError("bc_type must be 'D', 'N', or 'R' in set_south()")

    def set_west(self, bc_type, T_d=0.0, q=0.0, alpha=0.0, T_inf=0.0):
        """
        Left edge (i=0). Corners will be written by WEST when you call this first.
        """
        self._ensure_system()
        Nx, Ny, dx, k = self.dimX, self.dimY, self.dx, self.k

        def idx(i, j): return self._index(i, j)

        for j in range(Ny):
            n = idx(0, j)
            self._clear_row(n)

            bct = bc_type.upper()
            if bct == 'D':
                self.A[n, n] = 1.0
                self.b[n] = T_d
            elif bct == 'N':
                self.A[n, n] = -k / dx
                if Nx > 1:
                    self.A[n, idx(1, j)] = k / dx
                self.b[n] = q
            elif bct == 'R':
```

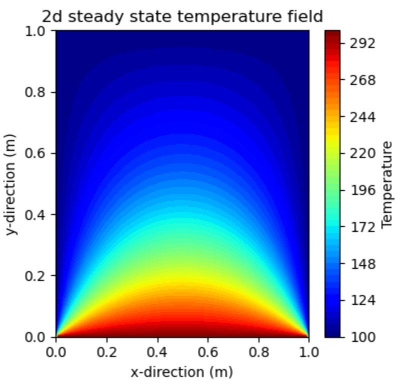
```
            self.A[n, n] = (-k / dx) - alpha
            if Nx > 1:
                self.A[n, idx(1, j)] = k / dx
            self.b[n] = -alpha * T_inf
        else:
            raise ValueError("bc_type must be 'D','N','R' in set_west()")

    def set_east(self, bc_type, T_d=0.0, q=0.0, alpha=0.0, T_inf=0.0):
        """
        Right edge (i=Nx-1). Corners will be written by EAST when you call this before S/N.
        """
        self._ensure_system()
        Nx, Ny, dx, k = self.dimX, self.dimY, self.dx, self.k

        def idx(i, j): return self._index(i, j)

        i = Nx - 1
        for j in range(Ny):
            n = idx(i, j)
            self._clear_row(n)

            bct = bc_type.upper()
            if bct == 'D':
                self.A[n, n] = 1.0
                self.b[n] = T_d
            elif bct == 'N':
                self.A[n, n] = k / dx
                if Nx > 1:
                    self.A[n, idx(i - 1, j)] = -k / dx
                self.b[n] = q
            elif bct == 'R':
                self.A[n, n] = (-k / dx) - alpha
                if Nx > 1:
                    self.A[n, idx(i - 1, j)] = k / dx
                self.b[n] = -alpha * T_inf
            else:
                raise ValueError("bc_type must be 'D','N','R' in set_east()")

    def set_north(self, bc_type, T_d=0.0, q=0.0, alpha=0.0, T_inf=0.0):
        """
        Top edge (j=Ny-1). Uses your original one-sided forms.
        """
        self._ensure_system()
        Nx, Ny, dy, k = self.dimX, self.dimY, self.dy, self.k

        def idx(i, j): return self._index(i, j)

        j = Ny - 1
        for i in range(Nx):
            n = idx(i, j)
            self._clear_row(n)

            bct = bc_type.upper()
            if bct == 'D':
                self.A[n, n] = 1.0
                self.b[n] = T_d
            elif bct == 'N':
                self.A[n, n] = k / dy
                if Ny > 1:
                    self.A[n, idx(i, j - 1)] = -k / dy
                self.b[n] = q
            elif bct == 'R':
                self.A[n, n] = (-k / dy) - alpha
                if Ny > 1:
                    self.A[n, idx(i, j - 1)] = k / dy
                self.b[n] = -alpha * T_inf
            else:
                raise ValueError("bc_type must be 'D','N','R' in set_north()")

    # -------------------- solve & visualize --------------------
    def solve(self):
        if self.A is None or self.b is None:
            raise ValueError("System not assembled. Call set_inner() first.")

        try:
            from scipy.sparse import csr_matrix
            from scipy.sparse.linalg import spsolve
            if hasattr(self.A, "tocsr"):
                T_flat = spsolve(csr_matrix(self.A), self.b)
            else:
                T_flat = np.linalg.solve(self.A, self.b)
        except Exception:
            A_mat = self.A.toarray() if hasattr(self.A, "toarray") else self.A
            T_flat = np.linalg.solve(A_mat, self.b)

        self.T = T_flat.reshape(self.dimY, self.dimX)
        return self.T

    def plot(self, TemperatureField, levels=50, cmap='jet'):
        x = np.linspace(0, self.l, self.dimX)
        y = np.linspace(0, self.h, self.dimY)
        X, Y = np.meshgrid(x, y)

        plt.figure(figsize=(4.2, 4.0))
        hm = plt.contourf(X, Y, TemperatureField, levels=levels, cmap=cmap)
        plt.colorbar(hm, label="Temperature")
        plt.xlabel("x")
        plt.ylabel("y")
        plt.title("2d steady state temperature field")
        plt.tight_layout()
        plt.show()
```

Stage 3 — Test Case 1: All Dirichlet Boundary Conditions

Validate your solver by testing a simple case with **Dirichlet boundaries on all sides**.
No internal heat source is active. The temperature field should show a smooth gradient between the hot and cold boundaries.

In [81]:
```
# ===============================================================
# Test Case 1 — All Dirichlet Boundary Conditions
```

```
# ==============================================================

# Domain and grid setup
Lx = 1.0
Ly = 1.0
dimX = 101
dimY = 101

# Point source (inactive for this test)
source = Source(location=[0.5, 0.5], is_active=False, q_dot=0.0)

# Initialize solver
heat = SteadyHeat2D(Lx, Ly, dimX, dimY, source)

# Apply boundary conditions
heat.set_south('D', T_d=300)   # Bottom boundary (hot)
heat.set_north('D', T_d=100)   # Top boundary
heat.set_east('D',  T_d=100)   # Right boundary
heat.set_west('D',  T_d=100)   # Left boundary

# Solve for temperature field
T = heat.solve() # expect shape = (dimX, dimY)

# Visualize result
heat.plot(T)
```

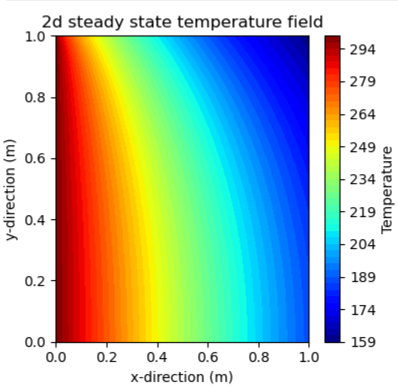


Stage 3 — Test Case 2 : Mixed Boundary Conditions

To validate the solver I run a case with **mixed boundary types** to verify correct handling of **Dirichlet**, **Neumann**, and **Robin** conditions simultaneously.
No internal heat source is active.

The temperature field should smoothly transition from the fixed west boundary to the other sides.

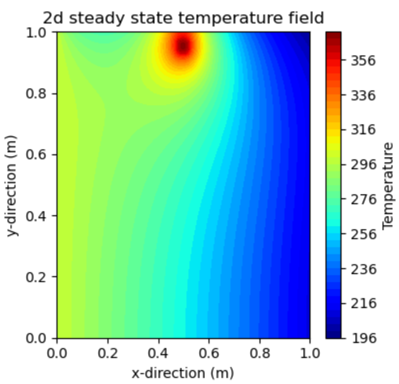
In [82]:
```
# ==============================================================
# Test Case 2 — Mixed Boundary Conditions - Original Case
# ==============================================================

# Domain and grid setup
Lx = 1.0
Ly = 1.0
dimX = 101
dimY = 101

# Point source (inactive for this test)
source = Source(location=[0.5, 0.5], is_active=False, q_dot=0.0)

# Apply boundary conditions
heat = SteadyHeat2D(Lx, Ly, dimX, dimY, source)
heat.set_south('N', q = 0.0)
heat.set_north('R', alpha = 1.0, T_inf = 100)
heat.set_east('R', alpha = 1.0, T_inf = 100)
heat.set_west('D', T_d = 300)

# Solve for temperature field
T = heat.solve() # expect shape = (dimX, dimY)

# Visualize result
heat.plot(T)
```



Stage 3 — Test Case 3 : Mixed Boundary Conditions with Point Source is Active

Validate the solver by running a case with **mixed boundary types** to verify correct handling of **Dirichlet**, **Neumann**, and **Robin** conditions simultaneously.
Internal heat source is active.

In [83]:
```
# ================================================================
# Test Case 3 — Mixed Boundary Conditions with Point Source is Active
# ================================================================
# Domain and grid setup
Lx = 1.0
Ly = 1.0
dimX = 101
dimY = 101

# Point source (inactive for this test)
source = Source(location=[0.5, 0.95], is_active=True, q_dot=200) #inner nodes only

# Apply boundary conditions
heat = SteadyHeat2D(Lx, Ly, dimX, dimY, source)
heat.set_south('N', q = 0.0)
heat.set_north('R', alpha = 1.0, T_inf = 100)
heat.set_east('R', alpha = 1.0, T_inf = 100)
heat.set_west('D', T_d = 300)

# Solve for temperature field
T = heat.solve() # expect shape = (dimX, dimY)

# Visualize result
heat.plot(T)
```



Stage 4 — Variable Thermal Conductivity (Optional)

Extending the steady-state heat conduction solver to handle **spatially varying thermal conductivity** λ(x, y).

Eq. (2.50) should now be discretized by considering λ depending on space.

In [ ]:
```
# New class that handles λ(x,y)

# Test Case for new class
class SteadyHeat2D_VarK:
    """
    2D steady heat conduction with spatially varying conductivity k(x,y).

    Discrete interior operator (finite volume / conservative FD on uniform grid):
        [(k_e (T_E - T_P) - k_w (T_P - T_W)) / dx] / dx
      + [(k_n (T_N - T_P) - k_s (T_S - T_S)) / dy] / dy = S_P

    where k_e,k_w,k_n,k_s are face values (harmonic averages of neighboring cell/node k).
    Supports:
      - k_func(x,y)   : callable returning scalar k at physical coords
      - k_field[j,i] : 2D array of k at grid nodes (shape (Ny, Nx))

    Source can be:
      - total power Q (W) spread over a small patch (mode='total')
      - areal source s (W/m^2) added directly (mode='areal')
    """

    def __init__(self, Lx, Ly, dimX, dimY,
                 k_func=None, k_field=None,
                 source=None, source_mode='total'):
        # domain & grid
        self.l = float(Lx)
        self.h = float(Ly)
        self.dimX = int(dimX)
        self.dimY = int(dimY)
        if self.dimX < 2 or self.dimY < 2:
            raise ValueError("dimX and dimY must be >= 2")

        self.dx = self.l / (self.dimX - 1)
        self.dy = self.h / (self.dimY - 1)

        # build conductivity field K[j,i]
        self.K = self._build_k_field(k_func, k_field)

        # optional source
        self.source_active = False
        self.source_mode = source_mode  # 'total' or 'areal'
        self.source_location = None
        self.source_strength = 0.0
        if source is not None:
            self.source_active   = bool(getattr(source, "is_active", False))
```

```python
        self.source_location = tuple(getattr(source, "location", (None, None)))
        self.source_strength = float(getattr(source, "q_dot", 0.0))

    # linear system
    self.A = None
    self.b = None
    self.T = None

# ---------------- helpers ----------------
def _index(self, i, j):
    return j * self.dimX + i

def _clear_row(self, r):
    if hasattr(self.A, "rows"):  # LIL sparse
        self.A.rows[r] = []
        self.A.data[r] = []
    else:
        self.A[r, :] = 0.0

def _coords(self):
    x = np.linspace(0.0, self.l, self.dimX)
    y = np.linspace(0.0, self.h, self.dimY)
    return np.meshgrid(x, y)  # X[i], Y[j] as 2D arrays (Y,X)

def _build_k_field(self, k_func, k_field):
    Ny, Nx = self.dimY, self.dimX
    if k_field is not None:
        K = np.array(k_field, dtype=float)
        if K.shape != (Ny, Nx):
            raise ValueError(f"k_field must have shape ({Ny},{Nx})")
        return K
    if k_func is None:
        # default: constant k=1 everywhere
        return np.ones((Ny, Nx), dtype=float)
    # sample callable on nodes
    x = np.linspace(0.0, self.l, Nx)
    y = np.linspace(0.0, self.h, Ny)
    X, Y = np.meshgrid(x, y)
    K = np.empty((Ny, Nx), dtype=float)
    for j in range(Ny):
        for i in range(Nx):
            K[j, i] = float(k_func(X[j, i], Y[j, i]))
    return K
@staticmethod
def _harmonic(a, b):
    """
    Element-wise harmonic mean for arrays/scalars:
        h = 2ab/(a+b)  (with safe handling where a+b == 0)
    """
    a = np.asarray(a, dtype=float)
    b = np.asarray(b, dtype=float)
    denom = a + b
    h = np.zeros_like(denom, dtype=float)
    mask = denom != 0.0
    h[mask] = (2.0 * a[mask] * b[mask]) / denom[mask]
    return h


def _ensure_system(self):
    if self.A is None or self.b is None:
        self.set_inner()

# ---------------- assembly ----------------
def set_inner(self):
    """
    Assemble A and b for ∇·(k ∇T) = S.
    Interior uses face-k (harmonic). Boundaries are identity rows here;
    call set_* BCs afterwards to overwrite boundary rows.
    """
    try:
        from scipy.sparse import lil_matrix
    except Exception:
        lil_matrix = None

    Nx, Ny = self.dimX, self.dimY
    dx, dy  = self.dx, self.dy
    N = Nx * Ny

    A = lil_matrix((N, N), dtype=float) if lil_matrix else np.zeros((N, N), dtype=float)
    b = np.zeros(N, dtype=float)

    K = self.K

    # precompute face k using harmonic averages
    # k_e[i,j] between (i,j) and (i+1,j), size (Ny, Nx-1)
    k_e = self._harmonic(K[:, :-1], K[:, 1:])
    k_w = k_e  # same array, but will be accessed shifted
    # k_n[i,j] between (i,j) and (i,j+1), size (Ny-1, Nx)
    k_n = self._harmonic(K[:-1, :], K[1:, :])
    k_s = k_n  # same array, accessed shifted

    inv_dx2 = 1.0 / (dx * dx)
    inv_dy2 = 1.0 / (dy * dy)

    for j in range(Ny):
        for i in range(Nx):
            n = self._index(i, j)

            # interior node
            if 0 < i < Nx - 1 and 0 < j < Ny - 1:
                ke = k_e[j, i]       # face east between (i,j) and (i+1,j)
                kw = k_w[j, i - 1]   # face west between (i-1,j) and (i,j)
                kn = k_n[j, i]       # face north between (i,j) and (i,j+1)
                ks = k_s[j - 1, i]   # face south between (i,j-1) and (i,j)

                aE = ke * inv_dx2
                aW = kw * inv_dx2
                aN = kn * inv_dy2
                aS = ks * inv_dy2
                aP = aE + aW + aN + aS

                A[n, n] = aP
```

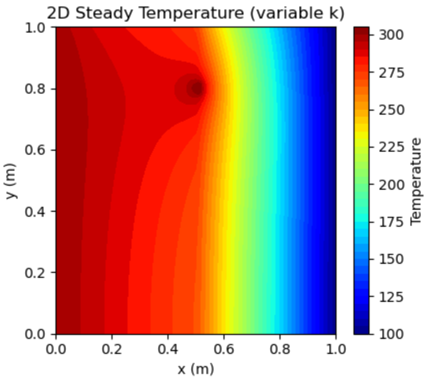
```python
                A[n, self._index(i + 1, j)] = -aE
                A[n, self._index(i - 1, j)] = -aW
                A[n, self._index(i, j + 1)] = -aN
                A[n, self._index(i, j - 1)] = -aS
                b[n] = 0.0

            else:
                # boundary placeholder (identity) to be overwritten by BC setters
                A[n, n] = 1.0
                b[n] = 0.0

        # ----- source term S -----
        if self.source_active and (self.source_location is not None):
            x0, y0 = self.source_location
            # map physical → nearest node index
            i0 = int(round(x0 / self.l * (Nx - 1))) if self.l > 0 else 0
            j0 = int(round(y0 / self.h * (Ny - 1))) if self.h > 0 else 0
            i0 = np.clip(i0, 0, Nx - 1)
            j0 = np.clip(j0, 0, Ny - 1)

            # patch (kept small; clip to interior to avoid BC overwrite)
            r = 2
            patch = [(ii, jj)
                     for jj in range(j0 - r, j0 + r + 1)
                     for ii in range(i0 - r, i0 + r + 1)
                     if 0 < ii < Nx - 1 and 0 < jj < Ny - 1]
            if not patch:
                # fallback to nearest interior
                ii = min(max(i0, 1), Nx - 2)
                jj = min(max(j0, 1), Ny - 2)
                patch = [(ii, jj)]

            if self.source_mode.lower() == 'total':
                # total power Q (W): split equally → RHS add per cell = Q/area_per_cell/num_cells
                Q = self.source_strength
                add = (Q / (self.dx * self.dy)) / len(patch)
            else:
                # areal density s (W/m^2): integrate over cell → s * dx * dy per cell
                s = self.source_strength
                add = (s)  # since operator already divides by dx,dy; put s directly on RHS


            for ii, jj in patch:
                b[self._index(ii, jj)] += add

        self.A = A
        self.b = b

    # ---------------- boundary conditions ----------------
    # We use first-order one-sided forms with *local face conductivity*.

    def set_west(self, bc_type, T_d=0.0, q=0.0, alpha=0.0, T_inf=0.0):
        self._ensure_system()
        Nx, Ny, dx = self.dimX, self.dimY, self.dx

        for j in range(Ny):
            r = self._index(0, j)
            self._clear_row(r)

            # face conductivity at west boundary (between (0,j) and (1,j))
            k_face = self._harmonic(self.K[j, 0], self.K[j, 1]) if Nx > 1 else self.K[j, 0]

            bct = bc_type.upper()
            if bct == 'D':
                self.A[r, r] = 1.0; self.b[r] = T_d
            elif bct == 'N':  # k_face*(T_1 - T_0)/dx = q
                self.A[r, r] = -k_face / dx
                if Nx > 1: self.A[r, self._index(1, j)] =  k_face / dx
                self.b[r] = q
            elif bct == 'R':  # k_face*(T_1 - T_0)/dx = alpha*(T_0 - T_inf)
                self.A[r, r] = (-k_face / dx) - alpha
                if Nx > 1: self.A[r, self._index(1, j)] =  k_face / dx
                self.b[r] = -alpha * T_inf
            else:
                raise ValueError("bc_type must be 'D','N','R' in set_west()")

    def set_east(self, bc_type, T_d=0.0, q=0.0, alpha=0.0, T_inf=0.0):
        self._ensure_system()
        Nx, Ny, dx = self.dimX, self.dimY, self.dx
        i = Nx - 1

        for j in range(Ny):
            r = self._index(i, j)
            self._clear_row(r)

            # face conductivity at east boundary (between (Nx-2,j) and (Nx-1,j))
            k_face = self._harmonic(self.K[j, i - 1], self.K[j, i]) if Nx > 1 else self.K[j, i]

            bct = bc_type.upper()
            if bct == 'D':
                self.A[r, r] = 1.0; self.b[r] = T_d
            elif bct == 'N':  # -k_face*(T_0 - T_1)/dx = q
                self.A[r, r] =  k_face / dx
                if Nx > 1: self.A[r, self._index(i - 1, j)] = -k_face / dx
                self.b[r] = q
            elif bct == 'R':  # -k_face*(T_0 - T_1)/dx = alpha*(T_0 - T_inf)
                self.A[r, r] = (-k_face / dx) - alpha
                if Nx > 1: self.A[r, self._index(i - 1, j)] =  k_face / dx
                self.b[r] = -alpha * T_inf
            else:
                raise ValueError("bc_type must be 'D','N','R' in set_east()")

    def set_south(self, bc_type, T_d=0.0, q=0.0, alpha=0.0, T_inf=0.0):
        self._ensure_system()
        Nx, Ny, dy = self.dimX, self.dimY, self.dy

        for i in range(Nx):
            r = self._index(i, 0)
            self._clear_row(r)

            # face conductivity at south boundary (between (i,0) and (i,1))
            k_face = self._harmonic(self.K[0, i], self.K[1, i]) if Ny > 1 else self.K[0, i]
```

```python
            bct = bc_type.upper()
            if bct == 'D':
                self.A[r, r] = 1.0; self.b[r] = T_d
            elif bct == 'N':  # k_face*(T_1 - T_0)/dy = q
                self.A[r, r] = -k_face / dy
                if Ny > 1: self.A[r, self._index(i, 1)] =  k_face / dy
                self.b[r] = q
            elif bct == 'R':  # k_face*(T_1 - T_0)/dy = alpha*(T_0 - T_inf)
                self.A[r, r] = (k_face / dy) + alpha
                if Ny > 1: self.A[r, self._index(i, 1)] = -k_face / dy
                self.b[r] = alpha * T_inf
            else:
                raise ValueError("bc_type must be 'D','N','R' in set_south()")

    def set_north(self, bc_type, T_d=0.0, q=0.0, alpha=0.0, T_inf=0.0):
        self._ensure_system()
        Nx, Ny, dy = self.dimX, self.dimY, self.dy
        j = Ny - 1

        for i in range(Nx):
            r = self._index(i, j)
            self._clear_row(r)

            # face conductivity at north boundary (between (i,Ny-2) and (i,Ny-1))
            k_face = self._harmonic(self.K[j - 1, i], self.K[j, i]) if Ny > 1 else self.K[j, i]

            bct = bc_type.upper()
            if bct == 'D':
                self.A[r, r] = 1.0; self.b[r] = T_d
            elif bct == 'N':  # -k_face*(T_0 - T_1)/dy = q
                self.A[r, r] =  k_face / dy
                if Ny > 1: self.A[r, self._index(i, j - 1)] = -k_face / dy
                self.b[r] = q
            elif bct == 'R':  # -k_face*(T_0 - T_1)/dy = alpha*(T_0 - T_inf)
                self.A[r, r] = (-k_face / dy) - alpha
                if Ny > 1: self.A[r, self._index(i, j - 1)] =  k_face / dy
                self.b[r] = -alpha * T_inf
            else:
                raise ValueError("bc_type must be 'D','N','R' in set_north()")

    # ---------------- solve & visualize ----------------
    def solve(self):
        if self.A is None or self.b is None:
            raise ValueError("System not assembled. Call set_inner() first.")
        try:
            from scipy.sparse import csr_matrix
            from scipy.sparse.linalg import spsolve
            if hasattr(self.A, "tocsr"):
                T_flat = spsolve(csr_matrix(self.A), self.b)
            else:
                T_flat = np.linalg.solve(self.A, self.b)
        except Exception:
            A_mat = self.A.toarray() if hasattr(self.A, "toarray") else self.A
            T_flat = np.linalg.solve(A_mat, self.b)
        self.T = T_flat.reshape(self.dimY, self.dimX)
        return self.T

    def plot(self, TemperatureField, levels=50, cmap='jet'):
        x = np.linspace(0, self.l, self.dimX)
        y = np.linspace(0, self.h, self.dimY)
        X, Y = np.meshgrid(x, y)
        plt.figure(figsize=(4.5, 4.0))
        cf = plt.contourf(X, Y, TemperatureField, levels=levels, cmap=cmap)
        plt.colorbar(cf, label='Temperature')
        plt.xlabel('x (m)'); plt.ylabel('y (m)')
        plt.title('2D Steady Temperature (variable k)')
        plt.tight_layout(); plt.show()
```

In [ ]:

```python
# ---- Test A: k(x,y) as a callable; jump in the middle; point source near top ----
def k_func(x, y):
    # left half conducts better than right half
    return 5.0 if x <= 0.5 else 1.0

# reusing Source container (location in meters)
src = Source(location=(0.50, 0.80), is_active=True, q_dot=200.0)

solverA = SteadyHeat2D_VarK(
    Lx=1.0, Ly=1.0,
    dimX=121, dimY=121,
    k_func=k_func,      # <= callable conductivity
    k_field=None,
    source=src,
    source_mode='total' # interpret q_dot as total power Q
)

solverA.set_inner()
solverA.set_west('D',  T_d=300.0)
solverA.set_east('D',  T_d=100.0)
solverA.set_south('N', q=0.0)
solverA.set_north('R', alpha=1.0, T_inf=100.0)

TA = solverA.solve()
print("Test A: T(min), T(max) =", float(TA.min()), float(TA.max()))
solverA.plot(TA)
```

```
Test A: T(min), T(max) = 100.0 304.8930927739237
```


2D Steady Temperature (variable k)
y (m)
x (m)
Temperature
1.0
0.8
0.6
0.4
0.2
0.0
0.0
0.2
0.4
0.6
0.8
1.0
300
275
250
225
200
175
150
125
100

In [ ]: