Wendland Radial Basis Function Used in SPLINE4

PRESENTED BY CHRISTIAN APARICIO

MARCH 5, 2026

Purpose

- Review how to use the Wendland C0 and C2 Functions
- Compare results with hand calculations and MSC Nastran output.
  - The SPLINE4 and SPLINE5 entries support Wendland radial basis functions.

Coupling Matrix H

- From reference 1, consider equation (5).
  - This is rewritten as (5.b).
- Equation (5.b) aligns to equations (16) and (17) in reference 2.
- Per reference 2, the coupling matrix H is:

$$\boldsymbol{H} = \boldsymbol{A}_{fs} \cdot \boldsymbol{C}_{ss}^{-1}. \qquad (11)$$

- Matrix H is also generated by MSC Nastran and stored in matrix GDGK or GPKG.

$$s|Y = \begin{pmatrix} \widetilde{A} & \widetilde{P} \end{pmatrix} \begin{pmatrix} A & P \\ P^T & 0 \end{pmatrix}^{-1} \begin{pmatrix} d \\ 0 \end{pmatrix} =: C \begin{pmatrix} d \\ 0 \end{pmatrix}. \qquad (5)$$

$$s|Y = \begin{pmatrix} \widetilde{P} & \widetilde{A} \end{pmatrix} \begin{pmatrix} 0 & P \\ P^T & A \end{pmatrix}^{-1} \begin{pmatrix} 0 \\ d \end{pmatrix} =: C \begin{pmatrix} 0 \\ d \end{pmatrix} \qquad (5.b)$$

$$\boldsymbol{C}_{ss} = \begin{array}{c} \\ 0 \\ \\ \\ \\ P^T \end{array} \begin{pmatrix} 0 & 0 & 0 & 0 & 1 & 1 & \cdots & 1 \\ 0 & 0 & 0 & 0 & x_{s_1} & x_{s_2} & \cdots & x_{s_{N_s}} \\ 0 & 0 & 0 & 0 & y_{s_1} & y_{s_2} & \cdots & y_{s_{N_s}} \\ 0 & 0 & 0 & 0 & z_{s_1} & z_{s_2} & \cdots & z_{s_{N_s}} \\ 1 & x_{s_1} & y_{s_1} & z_{s_1} & \phi_{s_1 s_1} & \phi_{s_1 s_2} & \cdots & \phi_{s_1 s_{N_s}} \\ 1 & x_{s_2} & y_{s_2} & z_{s_2} & \phi_{s_2 s_1} & \phi_{s_2 s_2} & \cdots & \phi_{s_2 s_{N_s}} \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_{s_{N_s}} & y_{s_{N_s}} & z_{s_{N_s}} & \phi_{s_{N_s} s_1} & \phi_{s_{N_s} s_2} & \cdots & \phi_{s_{N_s} s_{N_s}} \end{pmatrix} \begin{array}{l} P \\ \\ \\ \\ \\ \\ \\ A \end{array} . \qquad (16)$$

$$\boldsymbol{A}_{fs} = \begin{array}{c} \widetilde{P} \\ \\ \\ \\ \end{array} \begin{pmatrix} 1 & x_{f_1} & y_{f_1} & z_{f_1} & \phi_{f_1 s_1} & \phi_{f_1 s_2} & \cdots & \phi_{f_1 s_{N_s}} \\ 1 & x_{f_2} & y_{f_2} & z_{f_2} & \phi_{f_2 s_1} & \phi_{f_2 s_2} & \cdots & \phi_{f_2 s_{N_s}} \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_{f_{N_f}} & y_{f_{N_f}} & z_{f_{N_f}} & \phi_{f_{N_f} s_1} & \phi_{f_{N_f} s_2} & \cdots & \phi_{f_{N_f} s_{N_s}} \end{pmatrix} \begin{array}{l} \\ \\ \\ \\ \end{array} \widetilde{A} . \qquad (17)$$

Example

Training Data

- The inputs are the x, y, z coordinates.
- The outputs are the displacements.
- The training data consist of 4 samples.
- The prediction points consist of 2 samples. These are the points where the interpolation is performed, i.e. we want to know the interpolated displacements or forces.

Training Data

GRID ID	x	y	z	Displacement
1009	0	8	0	16
1011	0	10	0	0
1031	2	8	0	32
1033	2	10	0	16

Prediction Points

AEGRID ID	x	y	z	Displacement
2023	0.908477	9.09152	0	Unknown
2044	0.908477	8.31274	0	Unknown

```
MSC Nastran Bulk Data Entries
SET1    1001    1009    1011    1031    1033
AELIST  2002    2023    2044
SPLINE4 21      10      2002            1001            RIS     DISP
                         WF0    4.
GRID     1009            0.     8.       0.
GRID     1011            0.     10.      0.
GRID     1031           2.      8.       0.
GRID     1033           2.      10.      0.
AEGRID   2023           .908477 9.09152  0.
AEGRID   2044           .908477 8.31274  0.
DMIG, UG,   1, 1,,          1001, 3,     0.00
    , 1009, 3,    16.00
    , 1011, 3,     0.00
    , 1031, 3,    32.00
    , 1033, 3,    16.00
```

Calculation of $||s_i - s_j||$

$$||s_i - s_j|| = \begin{bmatrix} \sqrt{(0-0)^2+(8-8)^2} = 0 & \sqrt{(0-0)^2+(8-10)^2} = 2 & \sqrt{(0-2)^2+(8-8)^2} = 2 & \sqrt{(0-2)^2+(8-10)^2} = 2.828 \\ 2 & \sqrt{(0-0)^2+(10-10)^2} = 0 & \sqrt{(0-2)^2+(10-8)^2} = 2.828 & \sqrt{(0-2)^2+(10-10)^2} = 2 \\ 2 & 2.828 & \sqrt{(2-2)^2+(8-8)^2} = 0 & \sqrt{(2-2)^2+(8-10)^2} = 2 \\ 2.828 & 2 & 2 & \sqrt{(2-2)^2+(10-10)^2} = 0 \end{bmatrix}$$

Calculation of $||f_i - s_j||$

$$||f_i - s_j|| = \begin{bmatrix} \sqrt{(0.908477-0)^2+(9.09152-8)^2} = 1.4201 & \sqrt{(0.908477-0)^2+(9.09152-10)^2} = 1.2848 & \sqrt{(0.908477-2)^2+(9.09152-8)^2} = 1.5436 & \sqrt{(0.908477-2)^2+(9.09152-10)^2} = 1.4201 \\ \sqrt{(0.908477-0)^2+(8.31274-8)^2} = 0.9608 & \sqrt{(0.908477-0)^2+(8.31274-10)^2} = 1.9163 & \sqrt{(0.908477-2)^2+(8.31274-8)^2} = 1.1354 & \sqrt{(0.908477-2)^2+(8.31274-10)^2} = 2.0095 \end{bmatrix}$$

Calculation of $\Phi s_i s_j$

Matrix $||s_i - s_j||$ contains the radius for each $s_i s_j$

Let the core radius be 4, or rc = 4.

The C0 Wendland Function is used.

- If 0 < (1 – r / rc),
  - $\Phi s_i s_j$=(1 – r / rc)^2
- else,
  - $\Phi s_i s_j$ = 0

Examples

- If r=10, then 0 < (1 – r / rc)=(1-10/4)=-1.5 is false, so $\Phi s_i s_j$ = 0.
- If r=-10, then 0 < 3.5 is true, so $\Phi s_i s_j$=(1 – -10 / 4)^2=12.25.

$$||s_i - s_j|| = \begin{pmatrix} r=0 & r=2 & r=2 & r=2.828 \\ r=2 & r=0 & r=2.828 & r=2 \\ r=2 & r=2.828 & r=0 & r=2 \\ r=2.828 & r=2 & r=2 & r=0 \end{pmatrix}$$

$$\Phi s_i s_j = \begin{pmatrix} \Phi s_1 s_1 & \Phi s_1 s_2 & \Phi s_1 s_3 & \Phi s_1 s_4 \\ \Phi s_2 s_1 & \Phi s_2 s_2 & \Phi s_2 s_3 & \Phi s_2 s_4 \\ \Phi s_3 s_1 & \Phi s_3 s_2 & \Phi s_3 s_3 & \Phi s_3 s_4 \\ \Phi s_4 s_1 & \Phi s_4 s_2 & \Phi s_4 s_3 & \Phi s_4 s_4 \end{pmatrix}$$

$$= \begin{pmatrix} 1 & 0.25 & 0.25 & 0.085786 \\ 0.25 & 1 & 0.085786 & 0.25 \\ 0.25 & 0.085786 & 1 & 0.25 \\ 0.085786 & 0.25 & 0.25 & 1 \end{pmatrix}$$

Calculation of C_{ss}

$$C_{ss}= \begin{pmatrix} 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & x_{s_1} & x_{s_2} & x_{s_3} & x_{s_4} \\ 0 & 0 & 0 & 0 & y_{s_1} & y_{s_2} & y_{s_3} & y_{s_4} \\ 0 & 0 & 0 & 0 & z_{s_1} & z_{s_2} & z_{s_3} & z_{s_4} \\ 1 & x_{s_1} & y_{s_1} & z_{s_1} & \Phi s_1 s_1 & \Phi s_1 s_2 & \Phi s_1 s_3 & \Phi s_1 s_4 \\ 1 & x_{s_2} & y_{s_2} & z_{s_2} & \Phi s_2 s_1 & \Phi s_2 s_2 & \Phi s_2 s_3 & \Phi s_2 s_4 \\ 1 & x_{s_3} & y_{s_3} & z_{s_3} & \Phi s_3 s_1 & \Phi s_3 s_2 & \Phi s_3 s_3 & \Phi s_3 s_4 \\ 1 & x_{s_4} & y_{s_4} & z_{s_4} & \Phi s_4 s_1 & \Phi s_4 s_2 & \Phi s_4 s_3 & \Phi s_4 s_4 \end{pmatrix} = \begin{pmatrix} 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 2 & 2 \\ 0 & 0 & 0 & 0 & 8 & 10 & 8 & 10 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 8 & 0 & 1 & 0.25 & 0.25 & 0.085786 \\ 1 & 0 & 10 & 0 & 0.25 & 1 & 0.085786 & 0.25 \\ 1 & 2 & 8 & 0 & 0.25 & 0.085786 & 1 & 0.25 \\ 1 & 2 & 10 & 0 & 0.085786 & 0.25 & 0.25 & 1 \end{pmatrix}$$

Calculation of $\Phi \mathrm{f}_i s_j$

Matrix $||\mathrm{f}_i - s_j||$ contains the radius for each $\mathrm{f}_i s_j$

Use the same core radius rc = 4.

$$||\mathrm{f}_i - s_j|| = \begin{pmatrix} \text{r=1.4201} & \text{r=1.2848} & \text{r=1.5436} & \text{r=1.4201} \\ \text{r=0.9608} & \text{r=1.9163} & \text{r=1.1354} & \text{r=2.0095} \end{pmatrix}$$

$$\Phi \mathrm{f}_i s_j = \begin{pmatrix} \Phi f_1 s_1 & \Phi f_1 s_2 & \Phi f_1 s_3 & \Phi f_1 s_4 \\ \Phi f_2 s_1 & \Phi f_2 s_2 & \Phi f s_2 s_3 & \Phi f_2 s_4 \end{pmatrix}$$

$$= \begin{pmatrix} 0.4159857 & 0.4607753 & 0.3771051 & 0.4159843 \\ 0.5772960 & 0.2713648 & 0.5128557 & 0.2476196 \end{pmatrix}$$

Calculation of A_{fs}

$$A_{fs} = \begin{pmatrix} 1 & x_{f_1} & y_{f_1} & z_{f_1} & \Phi f_1 s_1 & \Phi f_1 s_2 & \Phi f_1 s_3 & \Phi f_1 s_4 \\ 1 & x_{f_2} & y_{f_2} & z_{f_2} & \Phi f_2 s_1 & \Phi f_2 s_2 & \Phi f s_2 s_3 & \Phi f_2 s_4 \end{pmatrix} = \begin{pmatrix} 1 & 0.908477 & 9.09152 & 0 & 0.4159857 & 0.4607753 & 0.3771051 & 0.4159843 \\ 1 & 0.908477 & 8.31274 & 0 & 0.5772960 & 0.2713648 & 0.5128557 & 0.2476196 \end{pmatrix}$$

Ignoring the z coordinate

The inverse of C_{ss} must be determined, but zero columns/rows exist and prevent finding the inverse. The z coordinate is ignored so the zero columns/rows are absent and the inverse of C_{ss} may be found.

$C_{ss}=$

0	0	0	0	1	1	1	1
0	0	0	0	0	0	2	2
0	0	0	0	8	10	8	10
0	0	0	0	0	0	0	0
1	0	8	0	1	0.25	0.25	0.085786
1	0	10	0	0.25	1	0.085786	0.25
1	2	8	0	0.25	0.085786	1	0.25
1	2	10	0	0.085786	0.25	0.25	1

$A_{fs}=$

1	0.908477	9.09152	0	0.4159857	0.4607753	0.3771051	0.4159843
1	0.908477	8.31274	0	0.5772960	0.2713648	0.5128557	0.2476196

Determine the interpolated displacements

u_s=[0, 0, 0, 16, 0, 32, 16]^T

- These are the known displacements at the structural points. Equation (5) shows 4 leading zeros are required. Since coordinate z is ignored on the previous slide, 3 leading zeros are required.

Recall from reference 2, equations (5) and (11). A_{fs}, C_{ss} and u_s are all available. H and u_f may now be determined.

$$\boldsymbol{H} = \boldsymbol{A}_{fs} \cdot \boldsymbol{C}_{ss}^{-1}. \qquad (11)$$

$$\boldsymbol{u}_f = \boldsymbol{H} \cdot \boldsymbol{u}_s. \qquad (5)$$

The interpolated displacements u_f are

u_f = 14.53566

20.76590

Comparison of Hand Calculated Displacements with MSC Nastran

Hand Calculation

u_f = **14.53566**

20.76590

MSC Nastran .pch file

```
DMI      UKF                0        2        1        0             1323        1
DMI*     UKF                                  1                 69  1.45356560E+01
*                         132  2.07658960E+01
```

Comparison of Hand Calculated H with MSC Nastran

Hand Calculated H

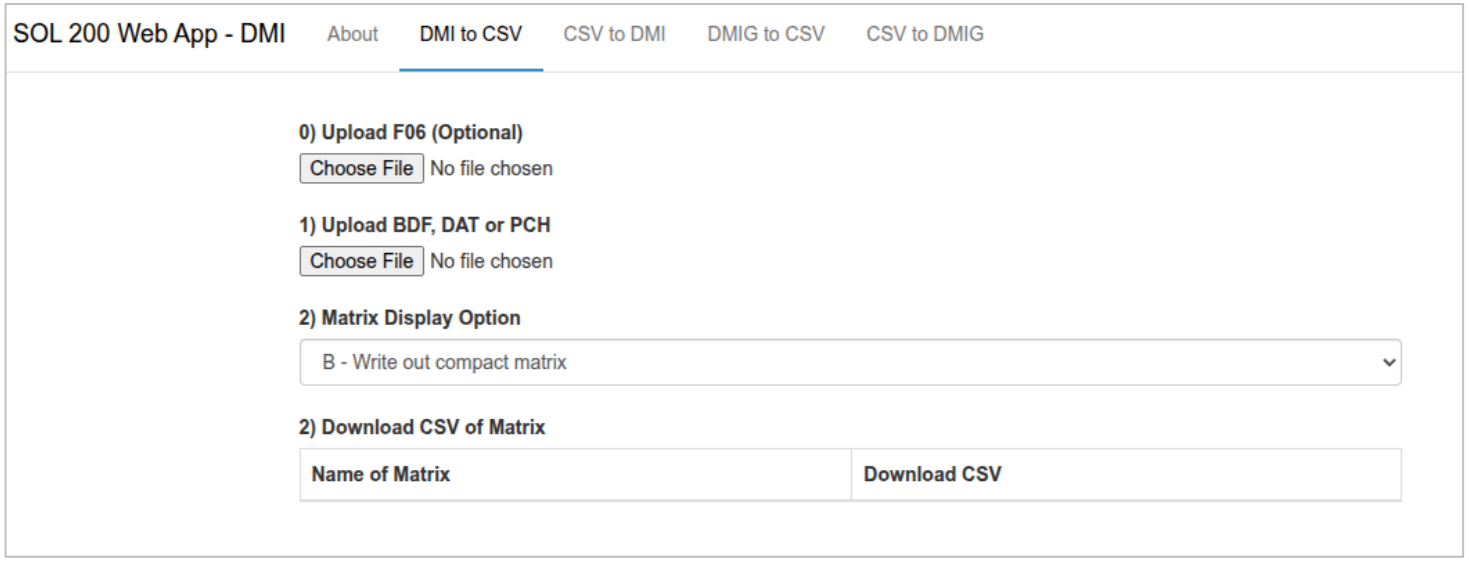
```
             [,1]           [,2]          [,3]      [,4]      [,5]      [,6]      [,7]
[1,]  0.02101598  2.207252e-09 2.207178e-09 0.2474783 0.29828322 0.2067617 0.24747678
[2,] -0.12158797 -1.128483e-03 1.428377e-02 0.4620634 0.08369806 0.3815666 0.07267194
```

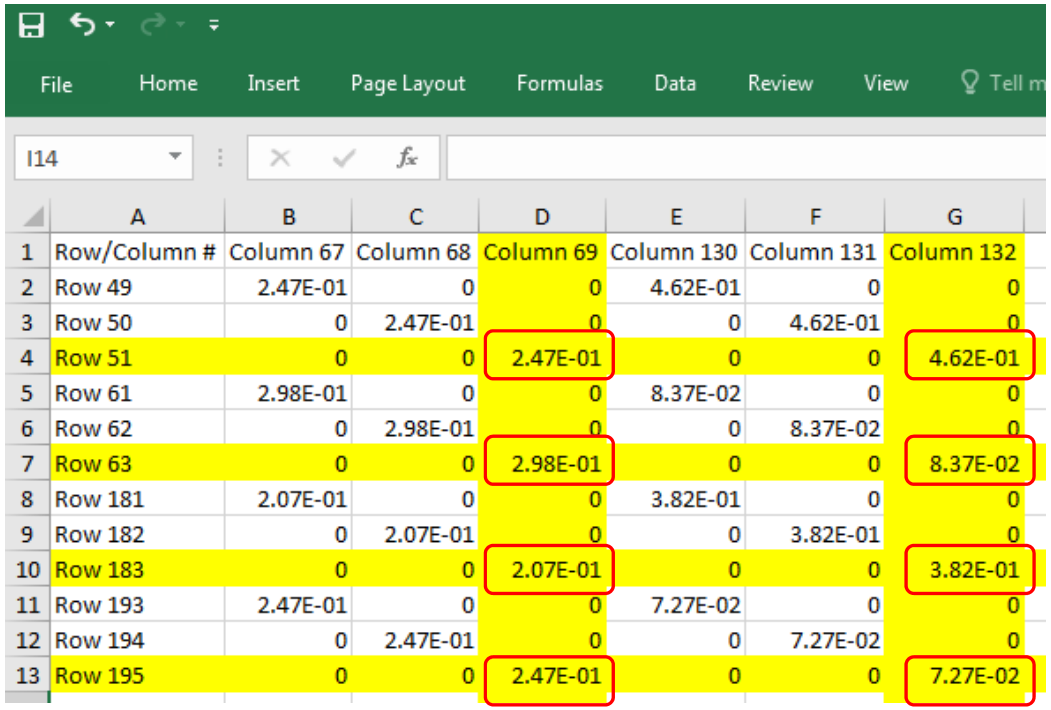
MSC Nastran H (GDGK) in the `.pch file`

```
DMI      GDKGF          0        2        1        0              726    1323
DMI*     GDKGF                            67               49  2.47478284E-01
*                      61  2.98283216E-01                181  2.06761716E-01
*                     193  2.47476784E-01
DMI*     GDKGF                            68               50  2.47478284E-01
*                      62  2.98283216E-01                182  2.06761716E-01
*                     194  2.47476784E-01
DMI*     GDKGF                            69               51  2.47478284E-01
*                      63  2.98283216E-01                183  2.06761716E-01
*                     195  2.47476784E-01
DMI*     GDKGF                           130               49  4.62063443E-01
*                      61  8.36980571E-02                181  3.81566557E-01
*                     193  7.26719429E-02
DMI*     GDKGF                           131               50  4.62063443E-01
*                      62  8.36980571E-02                182  3.81566557E-01
*                     194  7.26719429E-02
DMI*     GDKGF                           132               51  4.62063443E-01
*                      63  8.36980571E-02                183  3.81566557E-01
*                     195  7.26719429E-02
```

DMI Web App

- The DMI web app is used to convert DMI or DMIG entries to CSV files.
- The CSV files may then be opened in a spreadsheet application.





	A	B	C	D	E	F	G
1	Row/Column #	Column 67	Column 68	Column 69	Column 130	Column 131	Column 132
2	Row 49	2.47E-01	0	0	4.62E-01	0	0
3	Row 50	0	2.47E-01	0	0	4.62E-01	0
4	Row 51	0	0	2.47E-01	0	0	4.62E-01
5	Row 61	2.98E-01	0	0	8.37E-02	0	0
6	Row 62	0	2.98E-01	0	0	8.37E-02	0
7	Row 63	0	0	2.98E-01	0	0	8.37E-02
8	Row 181	2.07E-01	0	0	3.82E-01	0	0
9	Row 182	0	2.07E-01	0	0	3.82E-01	0
10	Row 183	0	0	2.07E-01	0	0	3.82E-01
11	Row 193	2.47E-01	0	0	7.27E-02	0	0
12	Row 194	0	2.47E-01	0	0	7.27E-02	0
13	Row 195	0	0	2.47E-01	0	0	7.27E-02

MSC Nastran's GDGK/GPGK matrix is opened in Excel.

Values circled in red are also found in the H matrix.

Comments about R

The example previously shown is recreated in R.

- “R is a programming language and free software environment for statistical computing and graphics supported by the R Foundation for Statistical Computing. The R language is widely used among statisticians and data miners for developing statistical software and data analysis.” Source: Wikipedia

The scripts determine all the matrices shown in this example.

Script A – This script considers only coordinates x and y.

Script B – This script considers coordinates x, y and z.

Script C – Same as script A, but the RBF (reference 3) Python package is used.

Code to replicate this example in R Script A

```r
library(plgp)

eps = sqrt(.Machine$double.eps)

generate_matrix_c_ss <- function(X, matrix_sigma_xn_xn) {
  rows <- nrow(matrix_sigma_xn_xn)
  cols <- ncol(matrix_sigma_xn_xn)

  # Initialize an array with zeros
  outgoing_matrix <- matrix(0, nrow = cols * 2 - 1, ncol = cols * 2 - 1)

  # Populate specific elements with 1
  for (i in 1:nrow(outgoing_matrix)) {
    # For rows 4, 5, 6, ...., column 1, use 1
    if ((rows - 1) < i) {
      outgoing_matrix[i] = 1
    }

    # For row 1, for columns 4, 5, 6, ..., use 1
    if (i == 1) {
      for (j in 4:(3 + cols)) {
        outgoing_matrix[i, j] = 1
      }
    }
  }

  # Populate the elements with positions (x, y, z)
  # For rows 4, 5, 6, etc.
  for (i in 4:nrow(outgoing_matrix)) {
    # While you might be in row 4 (i=4),
    # you need to access training data point 1 (j=1=4-3)
    j = i - 3

    # For column 2, 3, use x, y, respectively.
    outgoing_matrix[i, 2] = X[j, 1]
    outgoing_matrix[i, 3] = X[j, 2]
  }

  # For columns 4, 5, 6, etc.
  for (i in 4:nrow(outgoing_matrix)) {
    # While you might be in column 4 (i=4),
    # you need to access training data point 1 (j=1=4-3)
    j = i - 3

    # For row 2, 3, use x, y, respectively.
    outgoing_matrix[2, i] = X[j, 1]
    outgoing_matrix[3, i] = X[j, 2]
  }

  # Populate the elements corresponding to the covariance function Φ_si_si
  for (i in 1:nrow(matrix_sigma_xn_xn)) {
    new_i = i + 3

    for (j in 1:ncol(matrix_sigma_xn_xn)) {
      new_j = j + 3

      outgoing_matrix[new_i, new_j] = matrix_sigma_xn_xn[i, j]
    }
  }

  return (outgoing_matrix)
}

generate_matrix_a_fs <- function(X, matrix_sigma) {
  rows <- nrow(matrix_sigma)
  cols <- ncol(matrix_sigma)

  # Initialize an array with zeros
  # This matrix is nxm, where n is the number of prediction points
  # and m is the sum of the number of training points, 2 coordiantes (x, y)
  # and a leading column (with only 1 values).
  outgoing_matrix <- matrix(0, nrow = rows, ncol = cols + 3)

  # Populate specific elements with 1
  for (i in 1:nrow(outgoing_matrix)) {
    # For rows 1, 2, 3, ...., column 1, use 1
    outgoing_matrix[i] = 1
  }

  # Populate the elements with positions (x, y)
  # For rows 1, 2, 3, etc.
  for (i in 1:nrow(outgoing_matrix)) {
    # For column 2, 3, use x, y, respectively.
    outgoing_matrix[i, 2] = X[i, 1]
    outgoing_matrix[i, 3] = X[i, 2]
  }

  # Populate the elements corresponding to the covariance function Φ_fi_sj
  for (i in 1:nrow(matrix_sigma)) {
    for (j in 1:ncol(matrix_sigma)) {
      new_j = j + 3

      outgoing_matrix[i, new_j] = matrix_sigma[i, j]
    }
  }

  return (outgoing_matrix)
}

generate_wendland_covariance_matrix <- function(D, rc) {

  # Create any empty matrix
  rows <- nrow(D)
  cols <- ncol(D)
  outgoing_matrix <- matrix(0, nrow = rows, ncol = cols)

  # Wendland RBF
  for (i in 1:nrow(D)) {
    for (j in 1:ncol(D)) {
      # Use the square root of the L1 distance
      # Oddly, in R version 3.6.3, distance() is calculating the
      # L1 norm distance ((x2 - x1)^2 + (y2 - y1)^2), but the
      # L2 norm distance (sqrt((x2 - x1)^2 + (y2 - y1)^2)) is used
      # in the Wendland RBF. To get the radius, take the square
      # root of the L1 distance.
      r = sqrt(D[i, j])

      # Decide if the term is 0 or the Wendland expression
      y = 1 - r / rc

      if (y > 0) {
        # C0 Wendland Function
        outgoing_matrix[i, j] = (1 - r / rc)**2

        # C2 Wendland Function
        # outgoing_matrix[i, j] = (1 - r / rc)^4 * (4 * r / rc + 1)

      } else {
        # Zero
        outgoing_matrix[i, j] = 0
      }
    }
  }

  return(outgoing_matrix)
}

rc = 4 # Core radius

# Training and Prediction Data
# ####################################################
# Training points
X = rbind(c(0.0, 8.0), c(0.0, 10.0), c(2.0, 8.0), c(2.0, 10.0))
y = c(16.0, 0.0, 32.0, 16.0)

# Prediction points
XX = rbind(c(0.908477, 9.09152), c(0.908477, 8.31274))

# c_ss
# ####################################################
# Oddly, in R version 3.6.3, distance() is calculating the
# L1 norm distance ((x2 - x1)^2 + (y2 - y1)^2), where typically
# the L2 norm distance (sqrt((x2 - x1)^2 + (y2 - y1)^2)) is used
# in the Wendland RBF. To get the radius, take the square
# root of the L1 distance.
D = distance(X) # Distance among the Training Data
Sigma = generate_wendland_covariance_matrix(D, rc) # This is Φsi,sj
c_ss = generate_matrix_c_ss(X, Sigma)
print('c_ss')
c_ss

# a_fs
# ####################################################
DX = distance(XX, X) # Distance between training and testing data
SX = generate_wendland_covariance_matrix(DX, rc) # This is Φfi,sj
a_fs = generate_matrix_a_fs(XX, SX)
print('a_fs')
a_fs

# Calculate the interpolated value
# ####################################################
inverse_of_c_ss = solve(c_ss)
y1 <- c(c(0, 0, 0), y) # Add 3 leading zeros to the training data responses
mup = a_fs %*% inverse_of_c_ss %*% y1
mup
#           [,1]
# [1,] 14.53566
# [2,] 20.76590

# Print GDKG
# ####################################################
print(a_fs %*% inverse_of_c_ss)
# R Output GDKG
#           [,4]      [,5]      [,6]      [,7]
# [1,] 0.2474783 0.29828322 0.2067617 0.24747678
# [2,] 0.4620634 0.08369806 0.3815666 0.07267194
#
# MSC Nastran GDKG
# 0.247478284   0.298283216    0.206761716    0.247476784
# 0.462063443   0.083698057    0.381566557    0.072671943
```

Code to replicate this example in R Script B

```r
library(plgp)

eps = sqrt(.Machine$double.eps)

generate_matrix_c_ss <- function(X, matrix_sigma_xn_xn) {
  rows <- nrow(matrix_sigma_xn_xn)
  cols <- ncol(matrix_sigma_xn_xn)

  # Initialize an array with zeros
  outgoing_matrix <- matrix(0, nrow = cols * 2, ncol = cols * 2)

  # Populate specific elements with 1
  for (i in 1:nrow(outgoing_matrix)) {
    # For rows 5, 6, 7, ...., column 1, use 1
    if (rows < i) {
      outgoing_matrix[i] = 1
    }

    # For row 1, for columns 5, 6, 7, ..., use 1
    if (i == 1) {
      for (j in 5:(4 + cols)) {
        outgoing_matrix[i, j] = 1
      }
    }
  }

  # Populate the elements with positions (x, y, z)
  # For rows 5, 6, 7, etc.
  for (i in 5:nrow(outgoing_matrix)) {
    # While you might be in row 5 (i=5),
    # you need to access training data point 1 (j=1=5-4)
    j = i - 4

    # For column 2, 3, 4, use x, y, z, respectively.
    outgoing_matrix[i, 2] = X[j, 1]
    outgoing_matrix[i, 3] = X[j, 2]
    outgoing_matrix[i, 4] = X[j, 3]
  }

  # For columns 5, 6, 7, etc.
  for (i in 5:nrow(outgoing_matrix)) {
    # While you might be in column 5 (i=5),
    # you need to access training data point 1 (j=1=5-4)
    j = i - 4

    # For row 2, 3, 4, use x, y, z, respectively.
    outgoing_matrix[2, i] = X[j, 1]
    outgoing_matrix[3, i] = X[j, 2]
    outgoing_matrix[4, i] = X[j, 3]
  }

  # Populate the elements corresponding to the covariance function Φ_si_si
  for (i in 1:nrow(matrix_sigma_xn_xn)) {
    new_i = i + 4

    for (j in 1:ncol(matrix_sigma_xn_xn)) {
      new_j = j + 4

      outgoing_matrix[new_i, new_j] = matrix_sigma_xn_xn[i, j]
    }
  }

  return (outgoing_matrix)
}

generate_matrix_a_fs <- function(X, matrix_sigma) {
  rows <- nrow(matrix_sigma)
  cols <- ncol(matrix_sigma)

  # Initialize an array with zeros
  # This matrix is nxm, where n is the number of prediction points
  # and m is the sum of the number of training points, 3 coordiantes x, y, z,
  # and a leading column (with only 1 values).
  outgoing_matrix <- matrix(0, nrow = rows, ncol = cols + 4)

  # Populate specific elements with 1
  for (i in 1:nrow(outgoing_matrix)) {
    # For rows 5, 6, 7, ...., column 1, use 1
    outgoing_matrix[i] = 1
  }

  # Populate the elements with positions (x, y, z)
  # For rows 5, 6, 7, etc.
  for (i in 1:nrow(outgoing_matrix)) {
    # For column 2, 3, 4, use x, y, z, respectively.
    outgoing_matrix[i, 2] = X[i, 1]
    outgoing_matrix[i, 3] = X[i, 2]
    outgoing_matrix[i, 4] = X[i, 3] # z-coordinates are currently not considered
  }


  # Populate the elements corresponding to the covariance function Φ_fi_sj
  for (i in 1:nrow(matrix_sigma)) {
    for (j in 1:ncol(matrix_sigma)) {
      new_j = j + 4

      outgoing_matrix[i, new_j] = matrix_sigma[i, j]
    }
  }

  return (outgoing_matrix)
}

generate_wendland_covariance_matrix <- function(D, rc) {

  # Create any empty matrix
  rows <- nrow(D)
  cols <- ncol(D)
  outgoing_matrix <- matrix(0, nrow = rows, ncol = cols)

  # Wendland RBF
  for (i in 1:nrow(D)) {
    for (j in 1:ncol(D)) {
      # Use the square root of the L1 distance
      # Oddly, in R version 3.6.3, distance() is calculating the
      # L1 norm distance ((x2 - x1)^2 + (y2 - y1)^2), but the
      # L2 norm distance (sqrt((x2 - x1)^2 + (y2 - y1)^2)) is used
      # in the Wendland RBF. To get the radius, take the square
      # root of the L1 distance.
      r = sqrt(D[i, j])
      # r = D[i, j]^2 # NOT OK
      # r = D[i, j] * 2 # NOT OK
      # r = D[i, j] / 2 # NOT OK

      # Decide if the term is 0 or the Wendland expression
      y = 1 - r / rc

      if (y > 0) {
        # C0 Wendland Function
        outgoing_matrix[i, j] = (1 - r / rc)**2

        # C2 Wendland Function
        # outgoing_matrix[i, j] = (1 - r / rc)^4 * (4 * r / rc + 1)

      } else {
        # Zero
        outgoing_matrix[i, j] = 0
      }
    }
  }

  return(outgoing_matrix)
}

rc = 4 # Core radius

# Training and Prediction Data
# ####################################################
# If z is constant for all, I get an error that c_ss inverse cannot
# be found. I had to use 1.0E-6 for one z coordinate
# Training points
X = rbind(c(0.0, 8.0, 1.0E-6), c(0.0, 10.0, 0), c(2.0, 8.0, 0), c(2.0, 10.0, 0))
y = c(16.0, 0.0, 32.0, 16.0)

# Prediction points
XX = rbind(c(0.908477, 9.09152, 0), c(0.908477, 8.31274, 0))

# c_ss
# ####################################################
# Oddly, in R version 3.6.3, distance() is calculating the
# L1 norm distance ((x2 - x1)^2 + (y2 - y1)^2), where typically
# the L2 norm distance (sqrt((x2 - x1)^2 + (y2 - y1)^2)) is used
# in the Wendland RBF. To get the radius, take the square
# root of the L1 distance.
D = distance(X) # Distance among the Training Data
Sigma = generate_wendland_covariance_matrix(D, rc) # This is Φsi,sj
c_ss = generate_matrix_c_ss(X, Sigma)
print('c_ss')

# a_fs
# ####################################################
DX = distance(XX, X) # Distance between training and testing data
SX = generate_wendland_covariance_matrix(DX, rc) # This is Φfi,sj
SX
a_fs = generate_matrix_a_fs(XX, SX)
print('a_fs')
a_fs

# Calculate the interpolated value
# ####################################################
inverse_of_c_ss = solve(c_ss)
y1 <- c(0, 0, 0, 0), y) # Add 4 leading zeros to the training data responses
mup = a_fs %*% inverse_of_c_ss %*% y1
mup
#           [,1]
# [1,] 14.53566
# [2,] 20.76590

# Print GDKG
# ####################################################
# Why do these values differ from MSC Nastran?
# c_ss^-1 is signular because z is constant for all points.
# I had to add small perturbation to z for one training data point
# to ensure c_ss^-1 is obtainable. I suspect that MSC Nastran
# encounters the same issue and has its own way to account
# for this.
print(a_fs %*% inverse_of_c_ss)
# R Output GDKG
#       [,5]      [,6]     [,7]       [,8]
# [1,]    0 0.5457615 0.45424 -0.0000015
# [2,]    0 0.5457615 0.84363 -0.3893915
#
# MSC Nastran GDKG
# 0.247478284  0.298283216    0.206761716    0.247476784
# 0.462063443  0.083698057    0.381566557    0.072671943
```

Code to replicate this example in Python Script C

```python
import numpy as np
from rbf.interpolate import RBFInterpolant
import matplotlib
import matplotlib.pyplot as plt
import re
import csv

# Source: https://rbf.readthedocs.io/en/latest/basis.html


def convert_csv_to_list(incoming_text):
    reader = csv.DictReader(incoming_text.splitlines())

    list_of_all_variables = []

    for field_name in reader.fieldnames:
        if re.match(r'[XYZxyz]', field_name):
            list_of_all_variables.append(field_name)

    # Loop through each sample and construct the design matrix X
    # Re-read the CSV file. For some reason, the reader is no longer accessible
before this.
    reader = csv.DictReader(incoming_text.splitlines())
    x_train = []

    for row in reader:
        new_row = []

        for variable_name in list_of_all_variables:
            new_row.append(row[variable_name])

        x_train.append(new_row)

    # Convert to a numpy array
    x_train = np.array(x_train)

    # Force its elements to be floats
    x_train = x_train.astype(float)

    return x_train


# Training data
# ##############################################################################
# Acquire training data
file = open('./training_data_x.csv', 'r', encoding='utf8', errors='replace')
text_in_file = file.read()
file.close()
training_data_x = convert_csv_to_list(text_in_file)

file = open('./training_data_y.csv', 'r', encoding='utf8', errors='replace')
text_in_file = file.read()
file.close()
training_data_y = convert_csv_to_list(text_in_file)

# Prediction Points
# ##############################################################################
# Acquire prediction points
file = open('./prediction_data_x.csv', 'r', encoding='utf8', errors='replace')
text_in_file = file.read()
file.close()
prediction_points = convert_csv_to_list(text_in_file)

# Use tkinter for interactive plots
matplotlib.use('TkAgg')

# Establish a seed to reproduce the same random number.
# When epsilon (core radius) is being automatically determined,
# I suspect a global optimization is performed. This involves
# randomly selecting initial points for the initial variables.
# Use a seed value of your liking to enable the same
# randomly selected initial points.
np.random.seed(123)

# Construct the surrogate model (interpolation function)
# wen30 - C0 - (1 - r / rc)^2
# wen31 - C2 - (1 - r / rc)^4 (4 * r / rc + 1)
interp = RBFInterpolant(training_data_x, training_data_y, phi='wen30', eps='4',
order=1)
print(interp.eps)  # This is the core radius used in the Wendland kernel

# Interpolated values at the evaluation points
u_itp = interp(prediction_points)
print(u_itp)

# Prepare plot container
fig = plt.figure()
ax = fig.add_subplot(projection='3d')

# Scatter plot of training data
ax.scatter(training_data_x[:, 0], training_data_x[:, 1], training_data_y,
c='orange', marker='o')
ax.set_xlabel('X')
ax.set_ylabel('Y')
ax.set_zlabel('Response')

# Scatter plot of interpolated points
ax.scatter(prediction_points[:, 0], prediction_points[:, 1], u_itp[:, 0], c='blue',
marker='o')

# # Interpolated surface
# # low:high:increment_size
# prediction_points = np.mgrid[0:3.5:20j, 0:6:20j].reshape(2, -1).T
# u_itp = interp(prediction_points)
# surf = ax.plot_trisurf(prediction_points[:, 0], prediction_points[:, 1], u_itp[:,
0], cmap='viridis', linewidth=0, antialiased=True, alpha=0.5)

# Keep the plot open
plt.show()
```

Code to replicate this example in Python Script C, Continued

training_data_x.csv

```
Sample,x1,x2,x3
1009,0,8,0
1011,0,10,0
1031,2,8,0
1033,2,10,0
```

training_data_y.csv

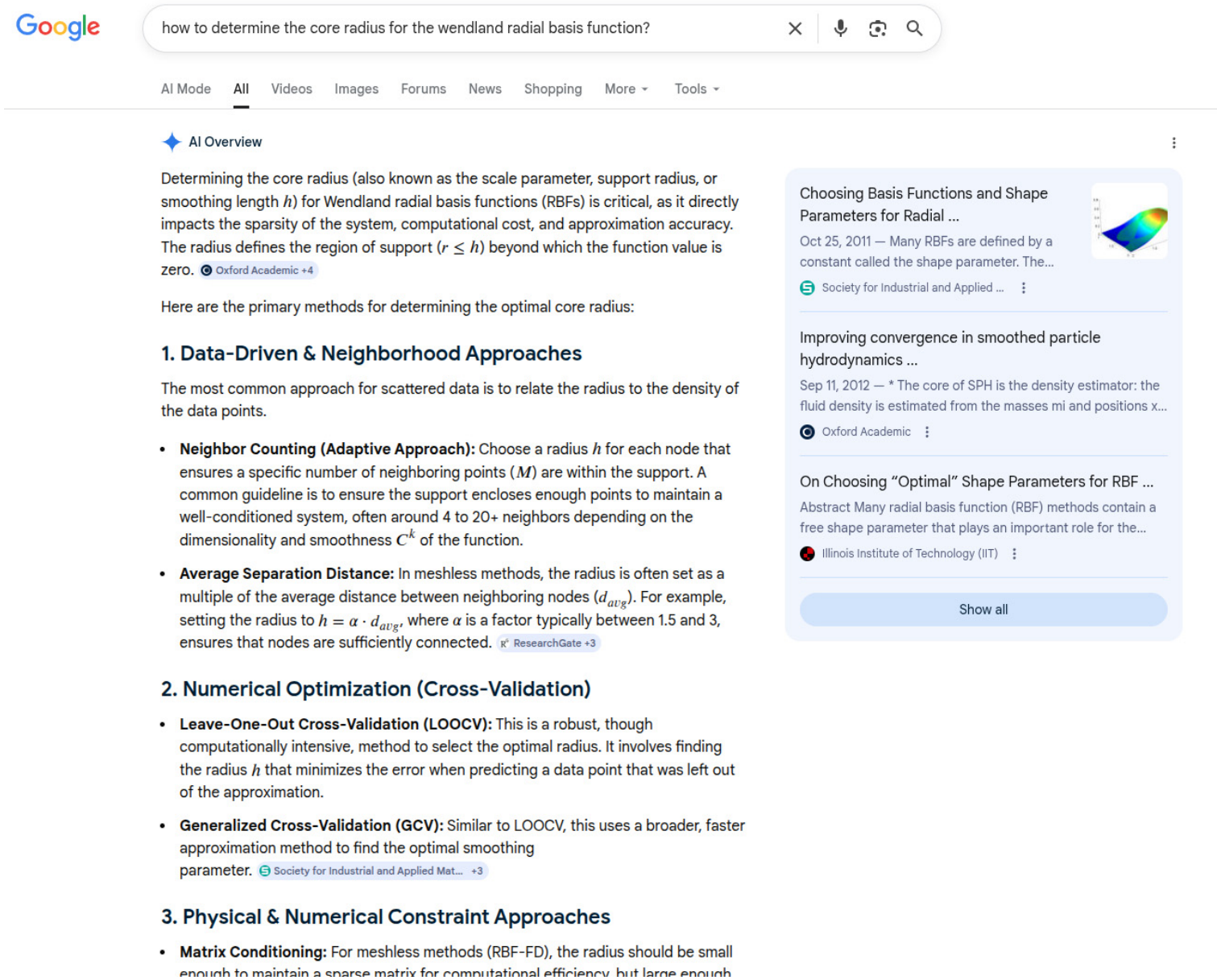
```
SAMPLE,y1
1009,16.0
1011,0.0
1031,32.0
1033,16.0
```

prediction_data_x.csv

```
Sample,x1,x2,x3
2023,0.908477,9.09152,0
2044,0.908477,8.31274,0
```

How to determine a core radius?

- Often, there is an automated procedure to determine the ideal parameters for radial basis functions.
- The Leave-One-Out Cross-Validation (LOOCV) method is one such automated procedure. The RBF Python package in reference 3 employs the LOOCV procedure to determine an ideal core radius.
- Note that for tens of thousands or millions of training points, e.g. structural or aerodynamic points, LOOCV may be computationally prohibitive. For a few dozen or hundred training points, LOOCV is practical.



References

1. Ahrem, R., Beckert, A., Wendland, H.: A New Multivariate Interpolation Method for Large-Scale Spatial Coupling Problems in Aeroelasticity. In: Conference Proceedings to the International Forum on Aeroelasticity and Structural Dynamics (IFASD), Munich (2005)
2. Beckert, A., Wendland, H.: Multivariate Interpolation for Fluid-Structure-Interaction Problems using Radial Basis Functions. AST - Aerospace Science and Technology, Art. No. 5125 (2001)
3. https://rbf.readthedocs.io/en/latest/index.html