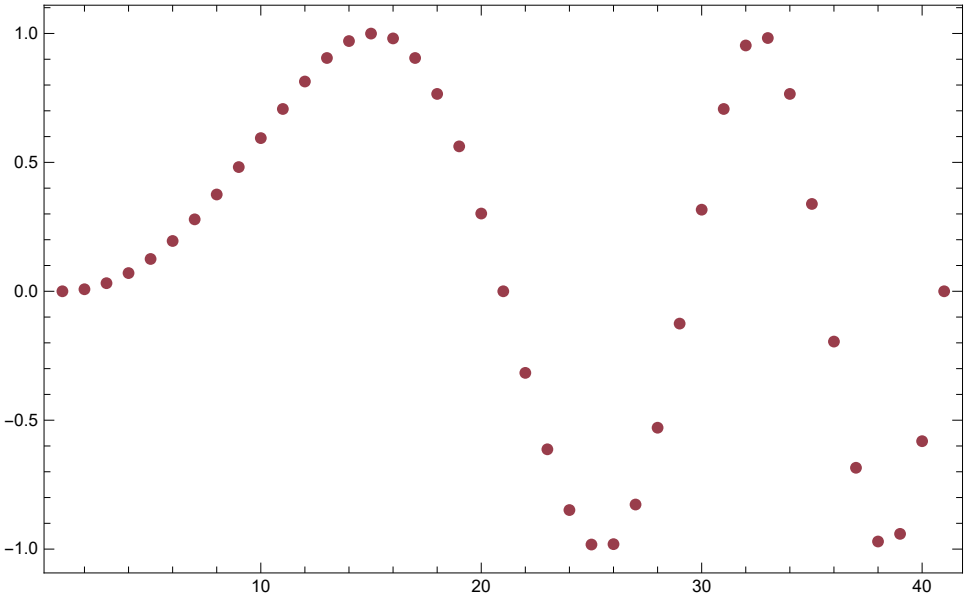
Data Analysis in Astronomy and Physics

Lecture 12: Interpolation and Smoothing

M. Röllig

Introduction

Suppose we know the value of a function $f(x)$ at a set of points $x_1, x_2, \ldots, x_N$ but we don't have an analytic expression for $f(x)$ that lets us calculate its value at an arbitrary point.



- Often the x_i are equally spaced, but not necessarily (see Fig.)
  - Usually the x_i are not of our own choosing!
  - If we are interested in $f(x_1 \leq x \leq x_N)$ this is called **interpolation**
  - If we are interested in $f(x \leq x_1 \text{ or } x \geq x_N)$ this is called **extrapolation**

Introduction

Inter/Extrapolation models the function between or beyond the known points by **some plausible form**.

General form to allow application to large classes of functions

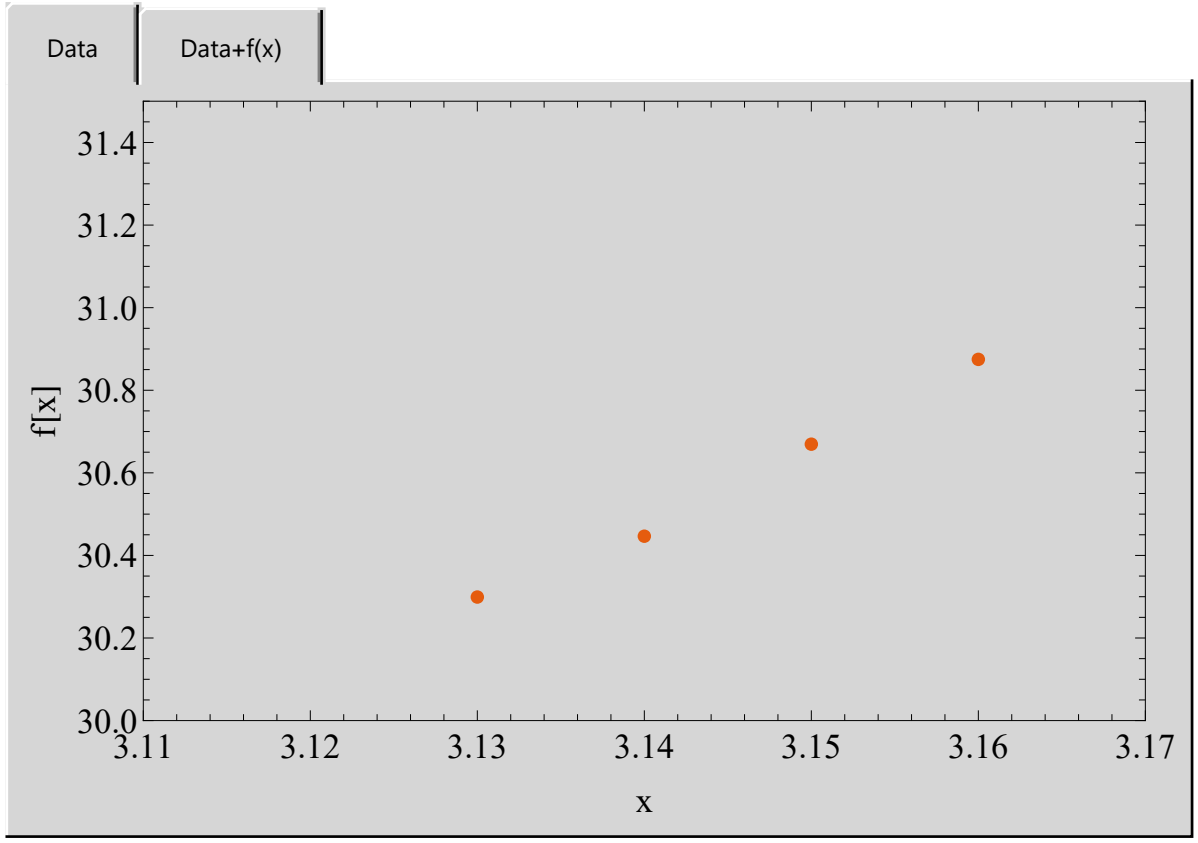
Most common form: polynomials

rational functions (quotients of polynomials)

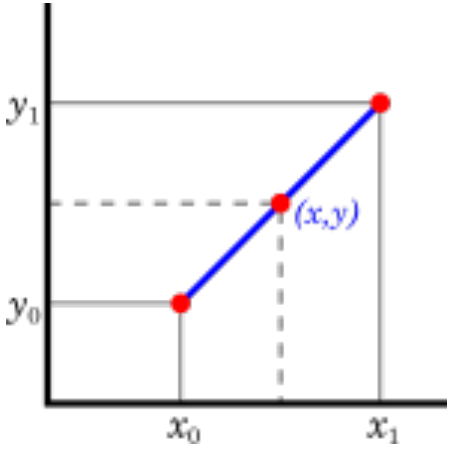
trigonometric functions

Every interpolation scheme has pathological cases where it fails to reproduce $f(x)$

Out[ ]=



Most Simple Case: Linear Interpolation



Given two points $\{x_0, y_0\}$ and $\{x_1, y_1\}$ the **linear interpolant** is the straight line between these points. For a value x in the interval $\{x_0, x_1\}$ the value y along the straight line is given from the equation

Out[•]//TraditionalForm=

$$\frac{y-y_0}{x-x_0} = \frac{y_1-y_0}{x_1-x_0}$$

Solving for *y* gives:

Out[•]//TraditionalForm=

$$p(x) = y = y_0 + (y_1 - y_0)\,\frac{x - x_0}{x_1 - x_0}$$

This is equivalent to a weighted average. Points further away have a smaller weight. The weights are $\frac{x-x_0}{x_1-x_0}$ and $\frac{x_1-x}{x_1-x_0}$

Out[•]//TraditionalForm=

$$p(x) = y = y_0\left(1 - \frac{x - x_0}{x_1 - x_0}\right) + y_1\left(1 - \frac{x_1 - x}{x_1 - x_0}\right) = y_0\left(1 - \frac{x - x_0}{x_1 - x_0}\right) + y_1\left(\frac{x - x_0}{x_1 - x_0}\right)$$

Most Simple Case: Linear Interpolation

The error of the approximation $p(x)$ to the underlying function $f(x)$ is

Out[•]//TraditionalForm=

$$R_T = f(x) - p(x) \quad \text{with} \quad p(x) = f(x_0) + \frac{f(x_1) - f(x_0)}{x_1 - x_0}(x - x_0)$$

It can be shown that the error is bounded by

Out[•]//TraditionalForm=

$$|R_T| \leq \frac{(x_1 - x_0)^2}{8} \max_{x_0 \leq x \leq x_1} |f''(x)|$$

As you see, the approximation between two points on a given function gets worse with the second derivative of the function that is approximated. This is intuitively correct as well: the "curvier" the function is, the worse the approximations made with simple linear interpolation.

Polynomial Interpolation

Through any two points there is a unique line. Through any three points there is a unique quadratic, etc. The interpolating polynomial of degree $N - 1$ through the N points $y_1 = f(x_1),\ y_2 = f(x_2),\ \ldots,\ y_N = f(x_N)$ is given explicitly by Lagrange's classical formula:

Out[•]//TraditionalForm=

$$P(x) = \frac{(x-x_2)(x-x_3)\ldots(x-x_N)}{(x_1-x_2)(x_1-x_3)\ldots(x_1-x_N)}y_1 + \frac{(x-x_1)(x-x_3)\ldots(x-x_N)}{(x_2-x_1)(x_2-x_3)\ldots(x_2-x_N)}y_2 + \ldots + \frac{(x-x_1)(x-x_2)\ldots(x-x_{N-1})}{(x_N-x_2)(x_N-x_3)\ldots(x_N-x_{N-1})}y_N = \sum_{i=0}^{N}\left(\prod_{\substack{0\le j\le N\\ j\neq i}}\frac{x-x_j}{x_i-x_j}\right)y_i$$

For matrix arguments, this formula is called Sylvester's formula and the matrix-valued Lagrange polynomials are the Frobenius covariants.

Polynomial Interpolation

There are N terms, each a polynomial of degree $N - 1$ and each constructed to be zero at all of the x_i except one, at which it is constructed to be y_i

Alternatively,suppose that the interpolation polynomial is in the form

Out[ ]//TraditionalForm=

$$P(x) = a_N x^N + a_{N-1} x^{N-1} + \ldots + a_2 x^2 + a_1 x^1 + a_0$$

The statement that P interpolates the data points means that

Out[ ]//TraditionalForm=

$$P(x_i) = y_i \text{ , for all i} \in \{0,1,...,N\}$$

Polynomial Interpolation

For a system of points we get a system of linear equations in the coefficients a_k

Out[ ]//TraditionalForm=

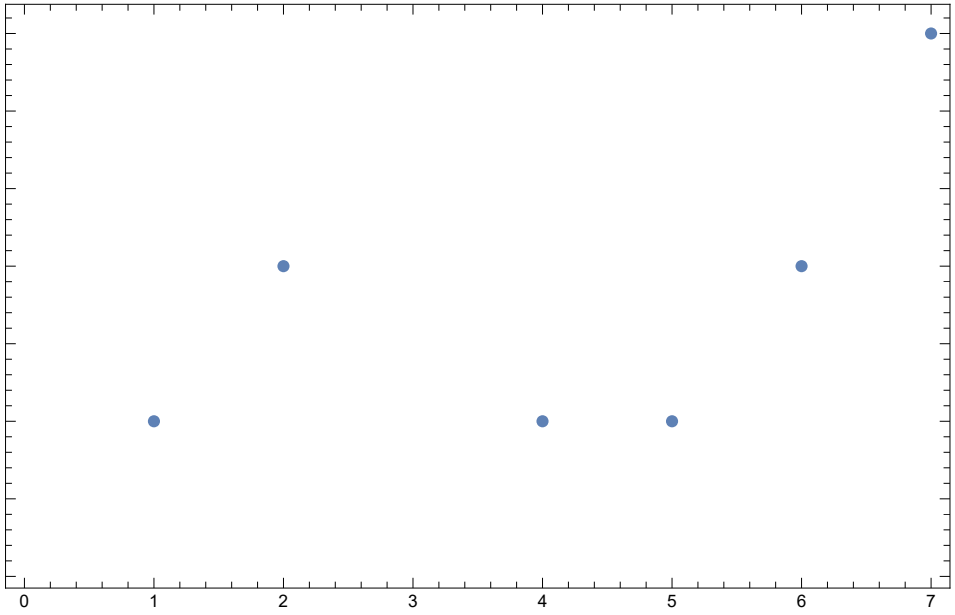
$$\begin{pmatrix} x_0^N & x_0^{N-1} & x_0^{N-2} & \dots & x_0 & 1 \\ x_1^N & x_1^{N-1} & x_1^{N-2} & \dots & x_2 & 1 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ x_N^N & x_N^{N-1} & x_N^{N-2} & \dots & x_N & 1 \end{pmatrix} \begin{pmatrix} a_N \\ a_{N-1} \\ \vdots \\ a_0 \end{pmatrix} = \begin{pmatrix} y_N \\ y_{N-1} \\ \vdots \\ y_0 \end{pmatrix}$$

We have to solve this system for a_k to construct $P(x)$. The matrix on the left is commonly referred to as a Vandermonde matrix.

Example - Polynomial Interpolation

```
data = {{1, 2}, {2, 4}, {4, 2}, {5, 2}, {6, 4}, {7, 7}};
ListPlot[data, ...]
```

Out[ ]=



In[ ]:=
```
polyTab = Evaluate[Table[#^n, {n, 0, Length[data] - 1}]] &
```

Out[ ]=
```
{1, #1, #1^2, #1^3, #1^4, #1^5} &
```

Example - Polynomial Interpolation

In[•]:=
```
mat = polyTab /@ data[[All, 1]];

MatrixForm[mat]
```

$$\begin{pmatrix} 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 2 & 4 & 8 & 16 & 32 \\ 1 & 4 & 16 & 64 & 256 & 1024 \\ 1 & 5 & 25 & 125 & 625 & 3125 \\ 1 & 6 & 36 & 216 & 1296 & 7776 \\ 1 & 7 & 49 & 343 & 2401 & 16\,807 \end{pmatrix}$$

In[•]:=
```
yVec = data[[All, 2]]
```

Out[•]=
```
{2, 4, 2, 2, 4, 7}
```

Example - Polynomial Interpolation

In[•]:=
```
coeffs = LinearSolve[mat, yVec]
```

Out[•]=
$$\left\{-\frac{14}{3}, \frac{85}{9}, -\frac{29}{10}, \frac{11}{180}, \frac{1}{15}, -\frac{1}{180}\right\}$$

```
mat.coeffs == yVec
```

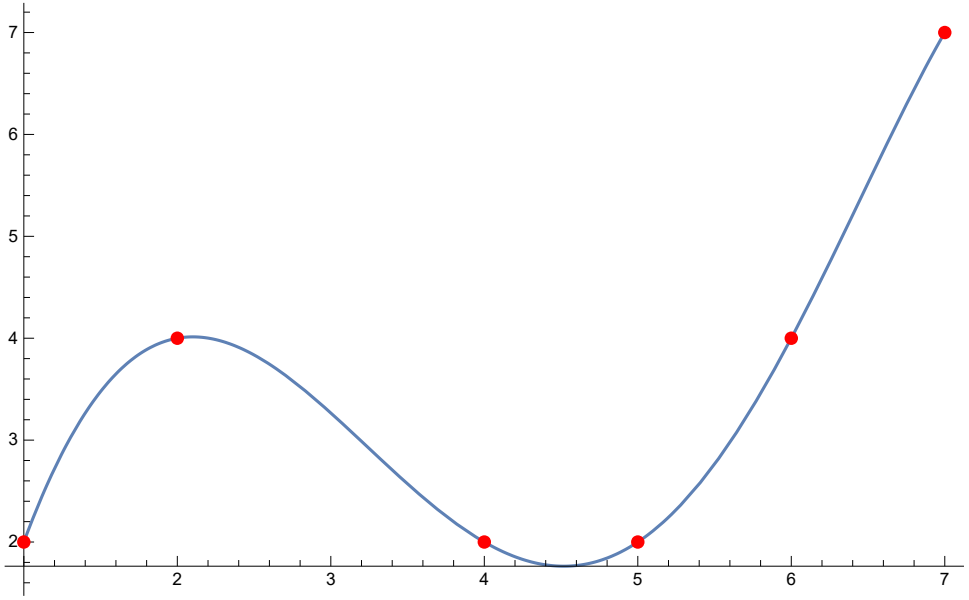
```
True
```

Example - Polynomial Interpolation

In[•]:=
```
poly[x_] := Evaluate[FromDigits[Reverse[coeffs], x]]
```

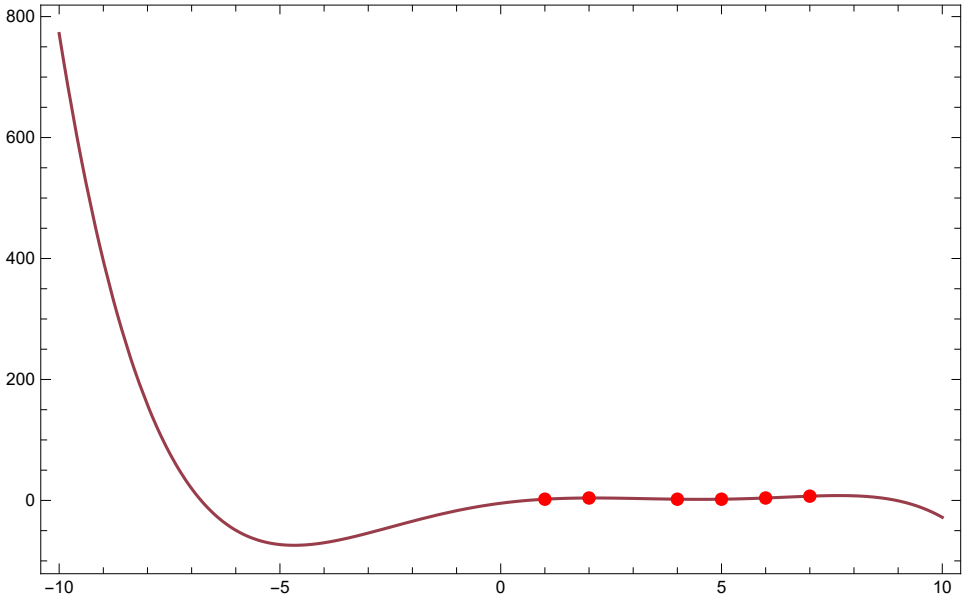
In[•]:=
```
Plot[poly[x], {x, 1, 7}, Epilog → {PointSize → Large, Red, Point /@ data}]
```

Out[•]=



Example - Polynomial Interpolation

```
Plot[poly[x], {x, -10, 10}, Epilog → {PointSize → Large, Red, Point /@ data}]
```



Example - Polynomial Interpolation

```
In[•]:= PolyInterpolate[data_] := Module[{n = Length[data], mat, y, polyTab, coeffs},
          polyTab = Evaluate[Table[#^n, {n, 0, Length[data] - 1}]] &;
          mat = polyTab /@ data[[All, 1]];
          y = data[[All, 2]];
          coeffs = LinearSolve[mat, y] // Quiet;
          FromDigits[Reverse[coeffs], x]]
```

```
In[•]:= PolyInterpolate[data]
```

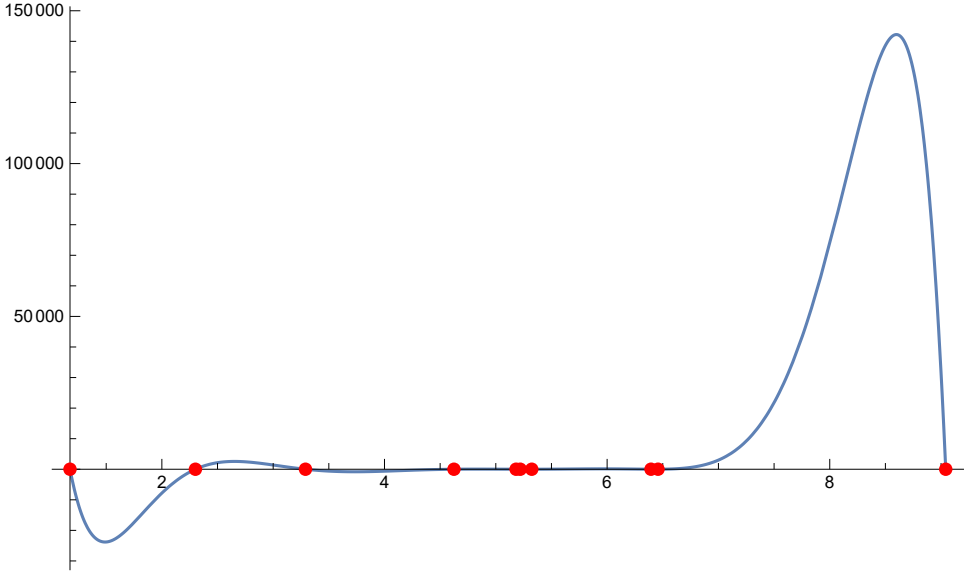
$$\text{Out}[\bullet]= -\frac{14}{3}+\frac{85\,x}{9}+\left(-\frac{29}{10}+\frac{11\,x}{180}\right)x^2+\left(\frac{1}{15}-\frac{x}{180}\right)x^4$$

Example - Polynomial Interpolation

In[ ]:=
```
plotDataInterp[data_, {xmin_, xmax_}, opts : OptionsPattern[{}]] :=
 Plot[Evaluate[PolyInterpolate[data]], {x, xmin, xmax}, Epilog → {PointSize → Large, Red, Point /@ data}, {opts}, PlotRange → All]
```

In[ ]:=
```
data = RandomVariate[UniformDistribution[{0, 10}], {10, 2}];
plotDataInterp[data, MinMax[data[[All, 1]]]]
```

Out[ ]=



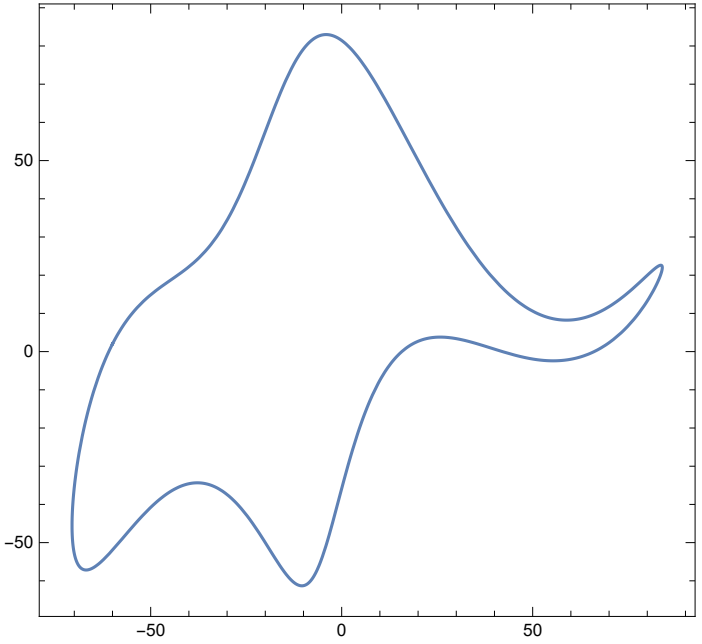
Fitting an elephant

"With four parameters I can fit an elephant, and with five I can make him wiggle his trunk."
(John von Neumann)

See also: https://publications.mpi-cbg.de/Mayer_2010_ 4314.pdf

In[•]:=
```
p = {50 + 30 I, 18 + 8 I, 12 + 10 I, -14 - 60 I, 40 + 20 I};
Ax = Ay = Bx = By = ConstantArray[0, 6];
Ax[[3 + 1]] = 12; Ax[[5 + 1]] = -14; Ay[[1 + 1]] = -60;
Bx[[1 + 1]] = 50; Bx[[2 + 1]] = 18; By[[1 + 1]] = -30; By[[2 + 1]] = 8; By[[3 + 1]] = -10;
x[t_] := Sum[Ax[[k + 1]] Cos[k t] + Bx[[k + 1]] Sin[k t], {k, 0, 5}]
y[t_] := Sum[Ay[[k + 1]] Cos[k t] + By[[k + 1]] Sin[k t], {k, 0, 5}]
```

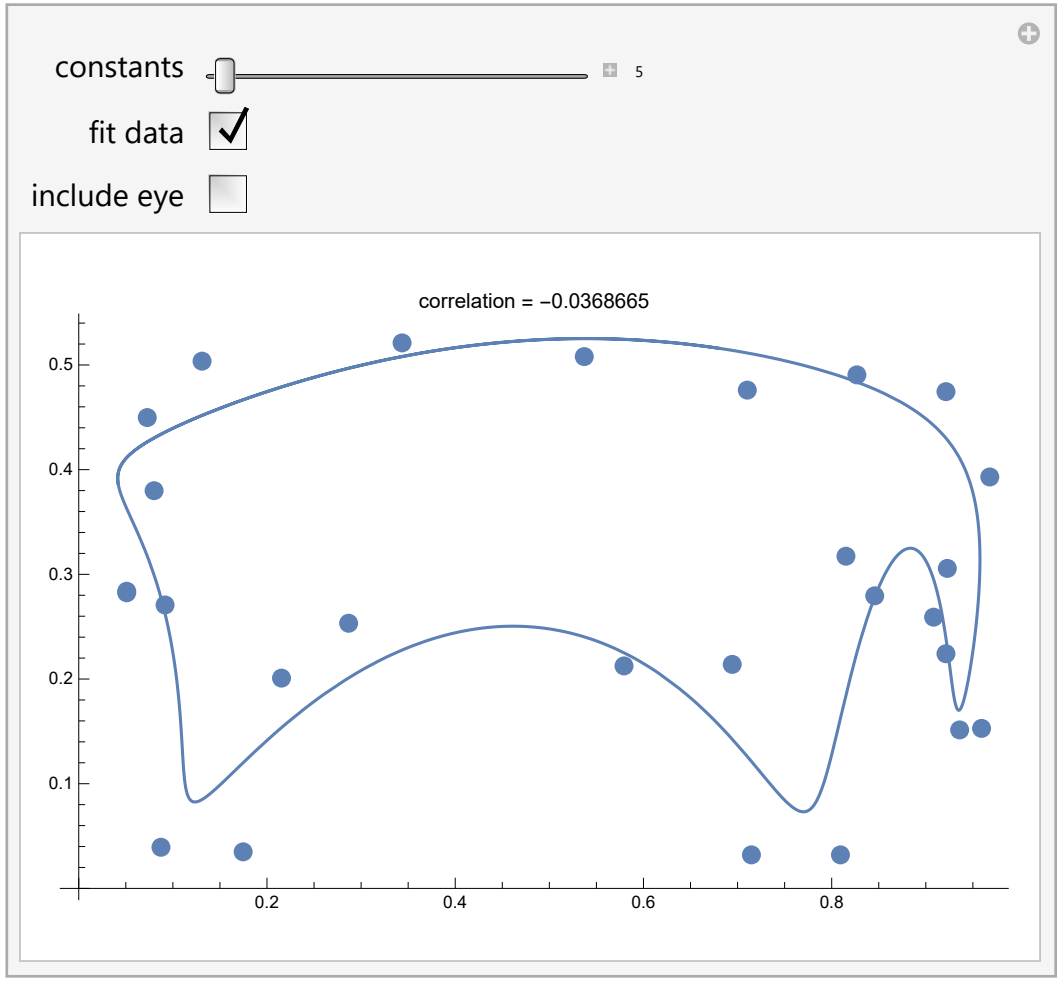
Out[•]=



Init

Fit

Out[ ]=



Contributed by : Roger J. Brown

Neville's Algorithm

Let P_1 be the value at x of the unique polynomial of degree zero (constant) passing through the point (x_1, y_1), so $P_1 = y_1$. Likewise define $P_2, P_3, \ldots, P_N$.

Now let P_{12} be the value at x of the unique polynomial of degree one passing through both (x_1, y_1) and (x_2, y_2). Likewise $P_{23}, P_{34}, \ldots, P_{(N-1)\,N}$. Similarly for higher order polynomial, up to $P_{123\ldots N}$ which is the value of the unique interpolating polynomial through all N points, i.e. the desired answer.

Neville's Algorithm

The various *P*'s form a "tableau" with "ancestors" on the left leading to a single "descendant" at the extreme right. For example for *N* = 4,

x_0	$P_0(x)$				
		$P_{01}(x)$			
x_1	$P_1(x)$		$P_{012}(x)$		
		$P_{12}(x)$		$P_{0123}(x)$	
x_2	$P_2(x)$		$P_{123}(x)$		$P_{01\,234}(x)$
		$P_{23}(x)$		$P_{1234}(x)$	
x_3	$P_3(x)$		$P_{234}(x)$		
		$P_{34}(x)$			
x_4	$P_4(x)$				

Out[•]=

$$P_i = y_i$$

$$P_{i\,(i+1)\,\ldots\,(i+m)} = \frac{(x - x_{i+m})\,P_{i\,(i+1)\,\ldots\,(i+m-1)} + (x_i - x)\,P_{(i+1)\,(i+2)\,\ldots\,(i+m)}}{x_i - x_{i+m}}$$

Example

Suppose we have data $f(x) = 1/x$ at the points $x = 2, 2.5, 4$. We want to compute $f(3)$.

i	x_i	$f(x_i)$
0	2	0.5
1	2.5	0.4
2	4	0.25

We start with the zero-order approximation

Out[•]=
$$f(3) \approx P_0(3) = f(x_0) = 0.5$$
$$f(3) \approx P_1(3) = f(x_1) = 0.4$$
$$f(3) \approx P_2(3) = f(x_2) = 0.25$$

Example

Now we construct the second generation:

Out[•]=

$$f(3) \approx P_{0,1}(3) = \frac{(3-x_1)\,P_0(3)-(3-x_0)\,P_1(3)}{x_0-x_1} = \frac{(3-2.5)\,0.5-(3-2)\,0.4}{2-2.5} = 0.3$$

$$f(3) \approx P_{1,2}(3) = \frac{(3-x_2)\,P_1(3)-(3-x_1)\,P_2(3)}{x_1-x_2} = \frac{(3-4)\,0.4-(3-2.5)\,0.25}{2.5-4} = 0.35$$

Example

Adding this to the table:

i	x_i	$f(x_i) = P_i$	$P_{i,i-1}$
0	2	0.5	
1	2.5	0.4	0.3
2	4	0.25	0.35

Out[•]= $f(3) \approx P_{0,1,2}(3) = \frac{(3-x_2)\,P_{0,1}(3)-(3-x_0)\,P_{1,2}(3)}{x_0-x_2} = \frac{(3-4)\,0.3-(3-2)\,0.35}{4-2} = 0.325$

i	x_i	$f(x_i) = P_i$	$P_{i,i-1}$	$P_{i,i-1,i-2}$
0	2	0.5		
1	2.5	0.4	0.3	
2	4	0.25	0.35	0.325

Example

0.1	−1.6228				
		−1.22230			
0.2	−0.8218		−1.18706		
		−1.08135		−1.17642	
0.3	−0.3027		−1.12320		−1.17186
		−0.91395		−1.13992	
0.4	0.1048		−1.02289		
		−0.76870			
0.5	0.4542				

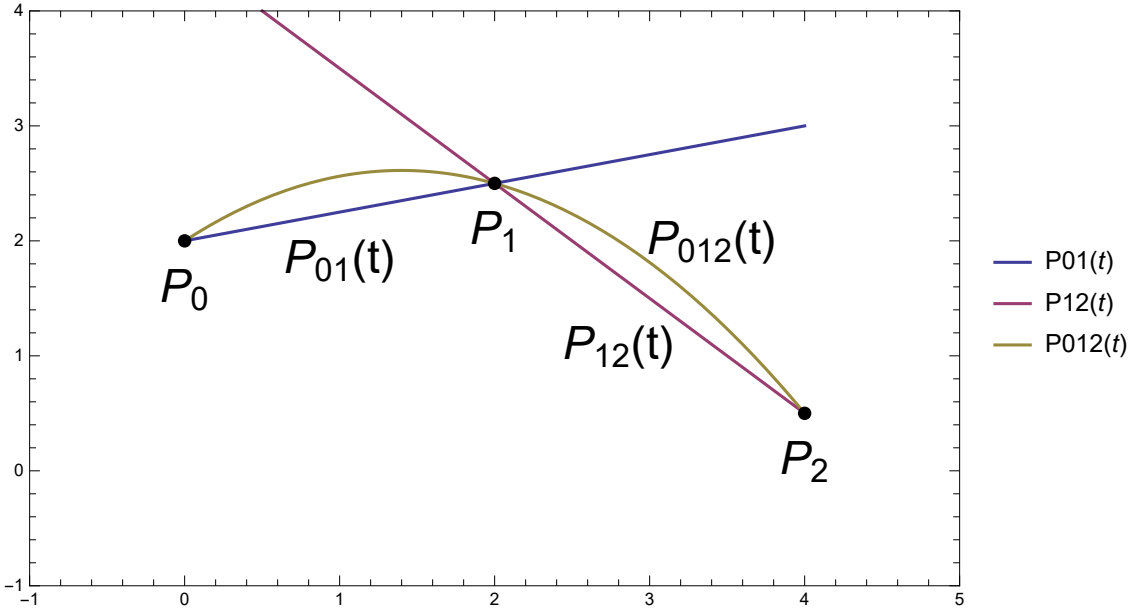
Out[ ]//TraditionalForm=

$$P_{123}(x) = \frac{(x - x_3)\, P_{12}(x) - (x - x_1)\, P_{23}(x)}{x_1 - x_3}$$

Quadratic Interpolation

Find a smooth curve $P_{012}(t)$ such that:

$P_{012}(t_0) = P_0 \quad P_{012}(t_1) = P_1 \quad P_{012}(t_2) = P_2$



Linear Interpolation

Out[ ]//TraditionalForm=

$$P_{01}(t) = \left(\frac{t_1 - t}{t_1 - t_0}\right) P_0 + \left(\frac{t - t_0}{t_1 - t_0}\right) P_1$$

Out[ ]//TraditionalForm=

$$P_{12}(t) = \left(\frac{t_2 - t}{t_2 - t_1}\right) P_0 + \left(\frac{t - t_1}{t_2 - t_1}\right) P_1$$

Quadratic Interpolation

Quadratic Interpolation

Out[ ]//TraditionalForm=

$$P_{012}(t) = \left(\frac{t_2 - t}{t_2 - t_0}\right) P_{01}(t) + \left(\frac{t - t_0}{t_2 - t_0}\right) P_{12}(t)$$

Verification

Out[ ]//TraditionalForm=

$$P_{012}(t) = \left(\frac{t_2 - t}{t_2 - t_0}\right) P_{01}(t) + \left(\frac{t - t_0}{t_2 - t_0}\right) P_{12}(t)$$

Out[ ]//TraditionalForm=

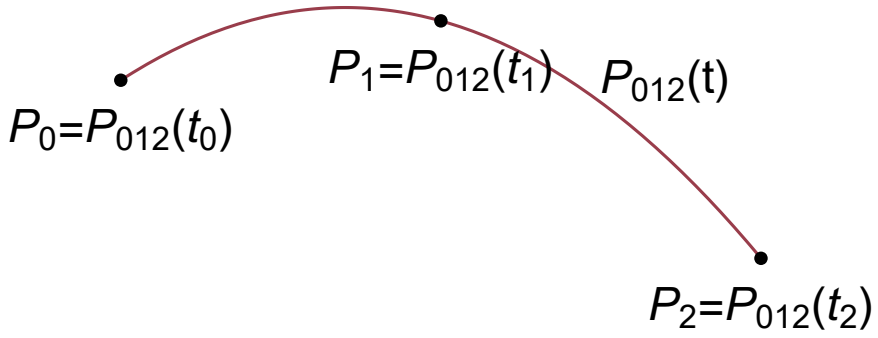
$$P_{012}(t_0) = \left(\frac{t_2 - t_0}{t_2 - t_0}\right) P_{01}(t_0) + \left(\frac{t_0 - t_0}{t_2 - t_0}\right) P_{12}(t_0) = P_1(t_0) = P_0$$

Out[ ]//TraditionalForm=

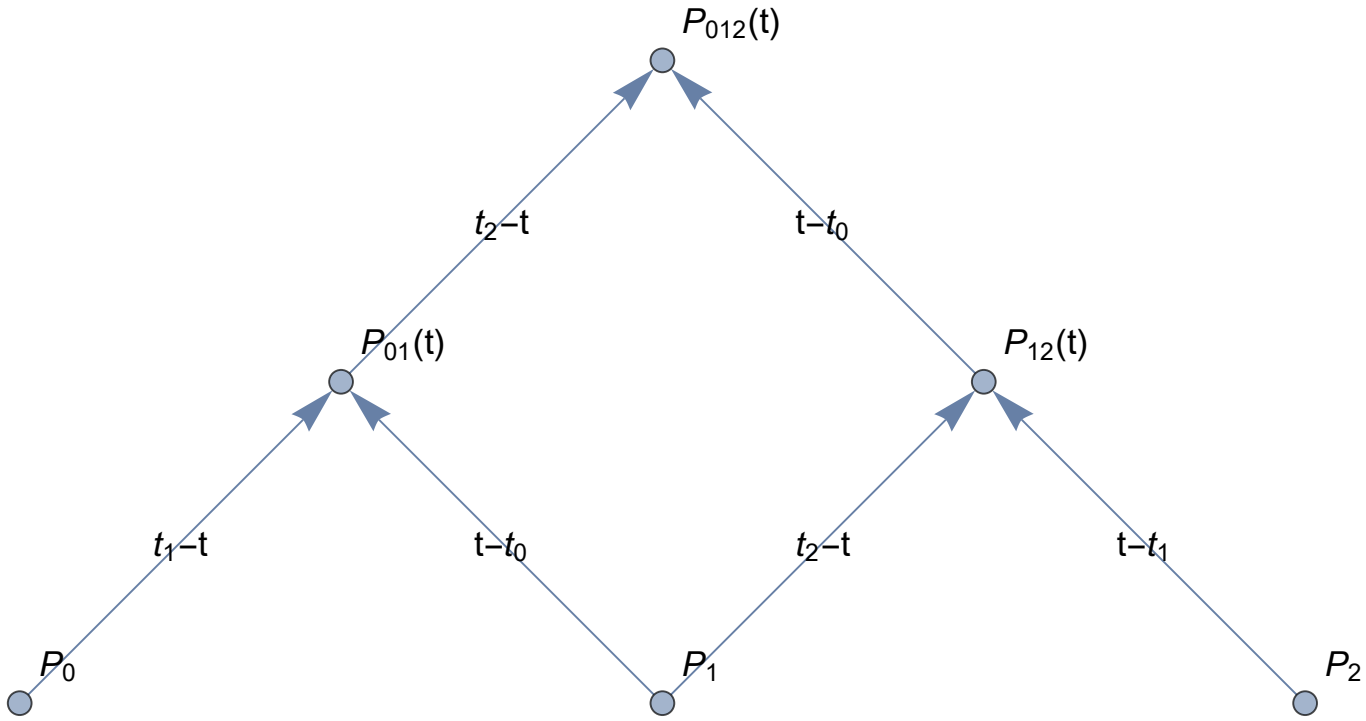
$$P_{012}(t_2) = \left(\frac{t_2 - t_2}{t_2 - t_0}\right) P_{01}(t_2) + \left(\frac{t_2 - t_0}{t_2 - t_0}\right) P_{12}(t_2) = P_{12}(t_2) = P_2$$

Out[ ]//TraditionalForm=

$$P_{012}(t_1) = \left(\frac{t_2 - t_1}{t_2 - t_0}\right) P_{01}(t_1) + \left(\frac{t_1 - t_0}{t_2 - t_0}\right) P_{12}(t_1) = \left(\frac{t_2 - t_1}{t_2 - t_0}\right) P_1 + \left(\frac{t_1 - t_0}{t_2 - t_0}\right) P_1 = P_1$$



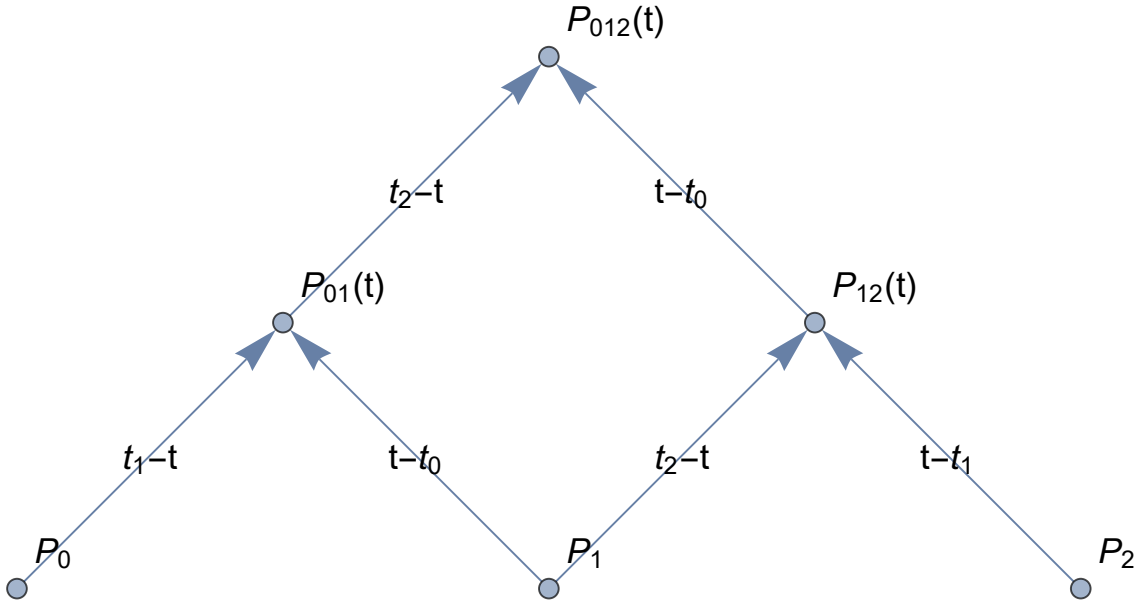
Neville's Algorithm for Quadratic Interpolation

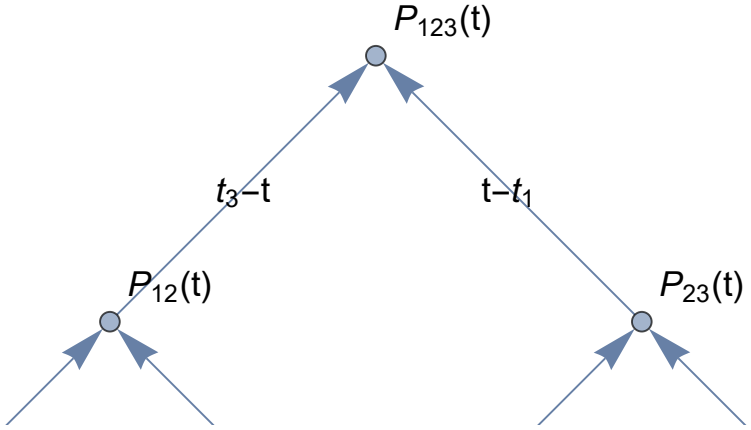


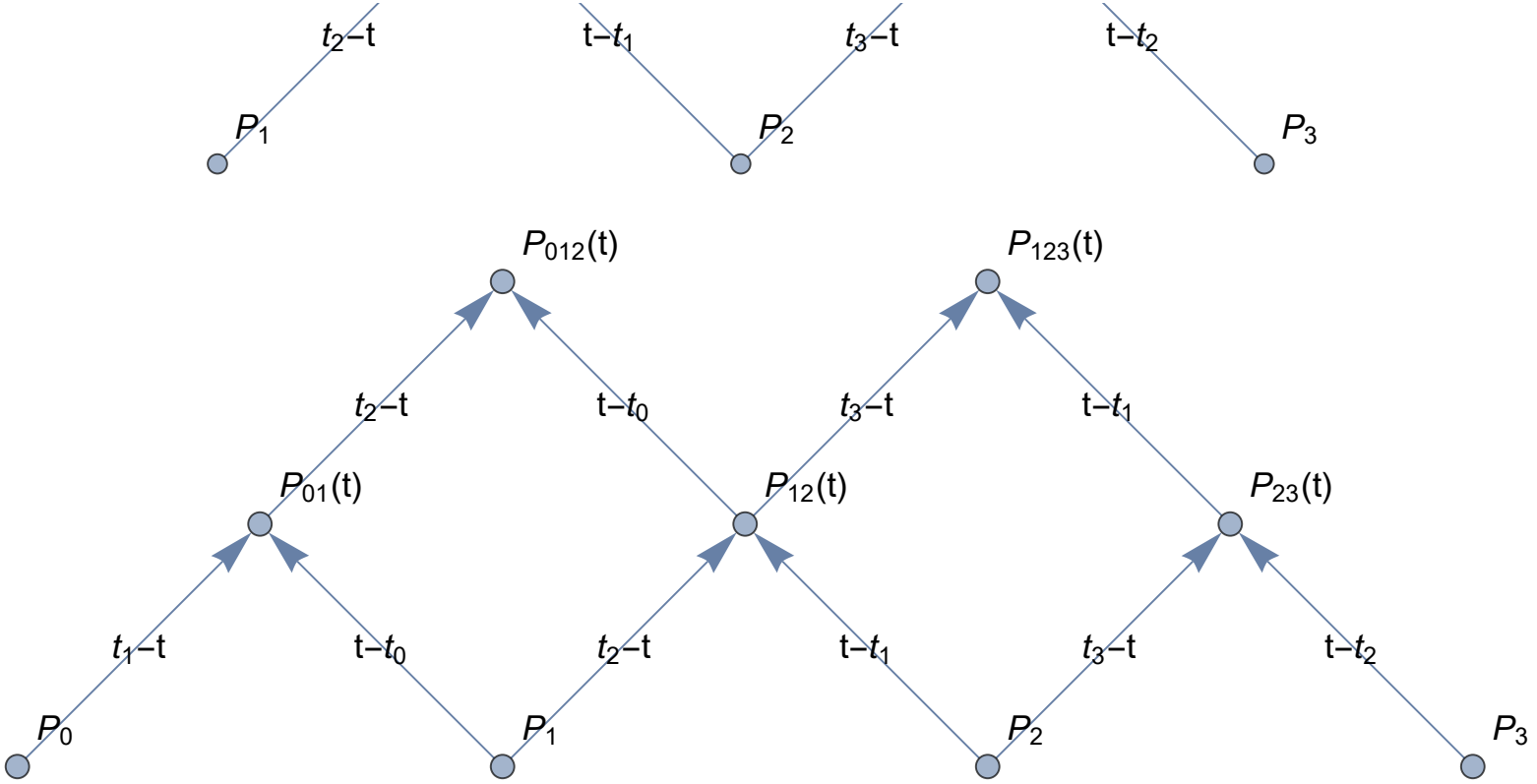
$$P_{012}(t_k) = P_k$$

Neville's Algorithm for Two Quadratic Interpolation

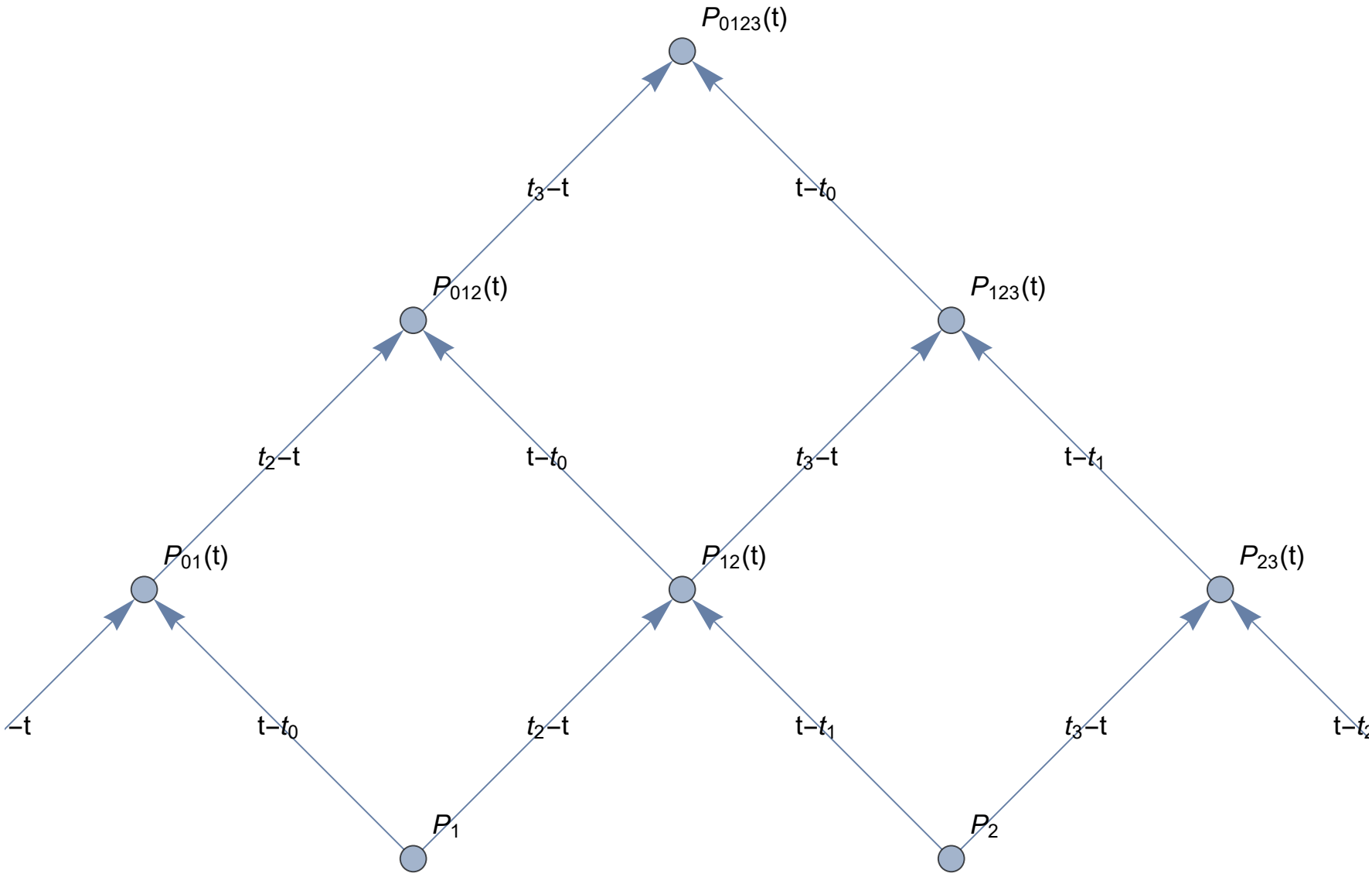
Out[ ]=






t_2–t
t–t_1
t_3–t
t–t_2
P_1
P_2
P_3
P_{012}(t)
P_{123}(t)
t_2–t
t–t_0
t_3–t
t–t_1
P_{01}(t)
P_{12}(t)
P_{23}(t)
t_1–t
t–t_0
t_2–t
t–t_1
t_3–t
t–t_2
P_0
P_1
P_2
P_3

Neville's Algorithm for Cubic Curves



Out[ ]//TraditionalForm=

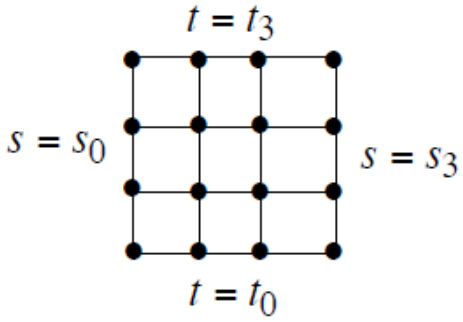
$$P_{123}(t) = \left(\frac{t_3 - t}{t_3 - t_0}\right) P_{12}(t) + \left(\frac{t - t_0}{t_3 - t_0}\right) P_{123}(t)$$

Advantages of Neville's Algorithm

- Numerically stable
  - Uses the given data directly
  - No need to represent the Polynomial in the basis 0, t, t^2, …
- Fast
  - Dynamic Programming
  - $O(n^2)$ vs. $O(2^n)$
- Simple Structure
  - parallel property
  - Easy to Update
- Add a computation of $O(n)$ instead of redoing work of $O(n^2)$

Neville's Algorithm for Tensor Product Surfaces

Setup



(a) Domain -- Rectangular Grid

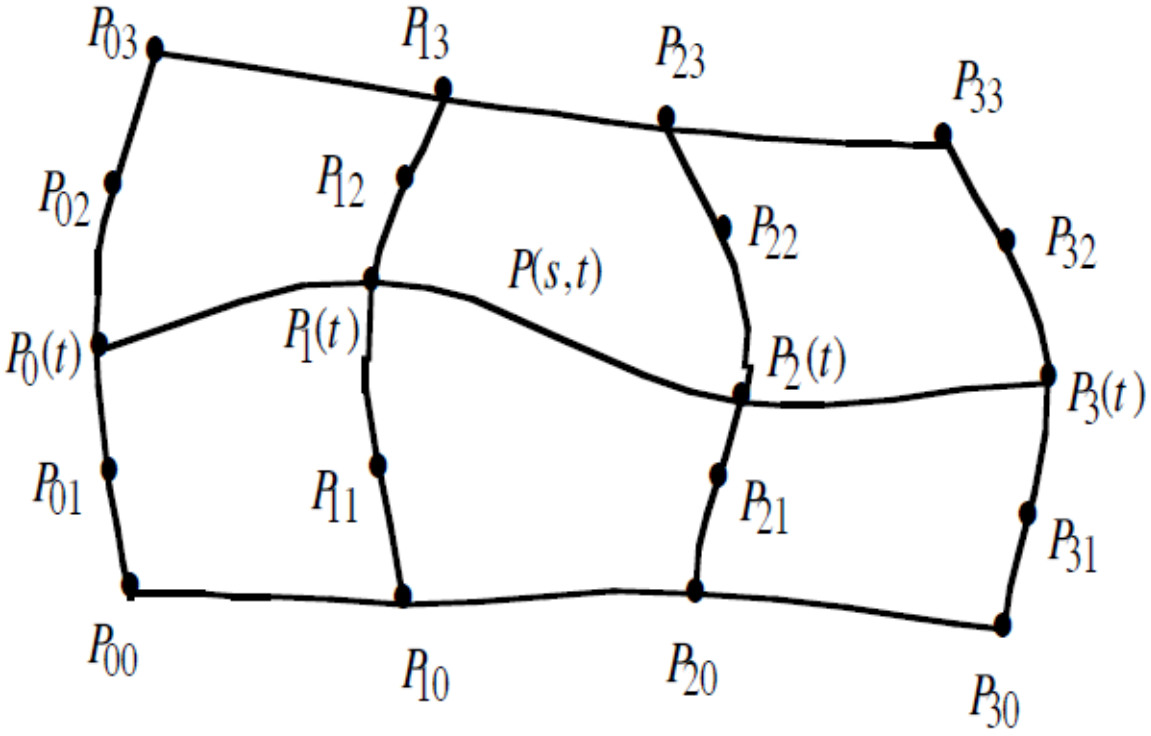
$$\begin{matrix} P_{03} & P_{13} & P_{23} & P_{33} \\ P_{02} & P_{12} & P_{22} & P_{32} \\ P_{01} & P_{11} & P_{21} & P_{31} \\ P_{00} & P_{10} & P_{20} & P_{30} \end{matrix}$$

(b) Range -- Rectangular Array of Points

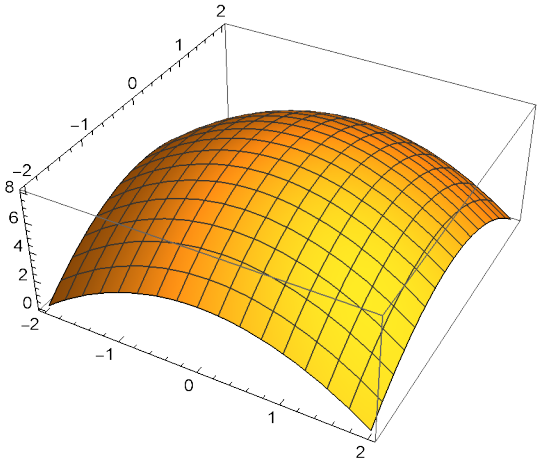
Find a smooth curve $P(s, t)$ such that: $P(s_i, t_j) = P_{ij}$

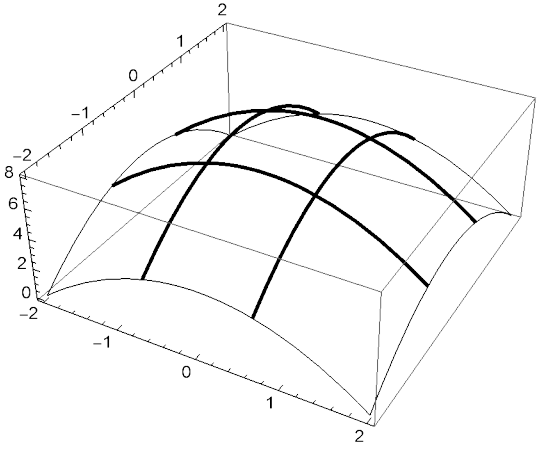
This interpolation works only on a regular grid!

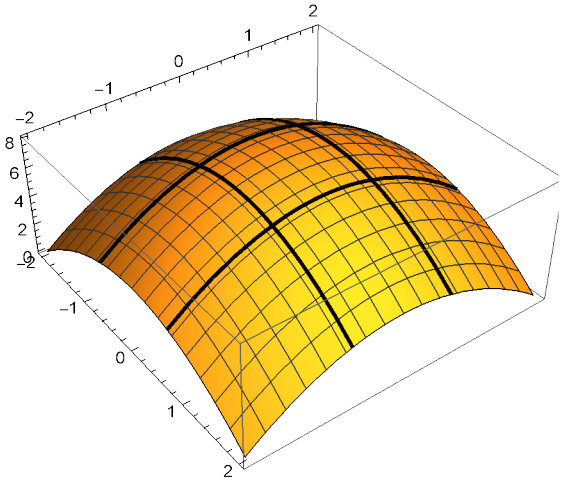
Surface Interpolation



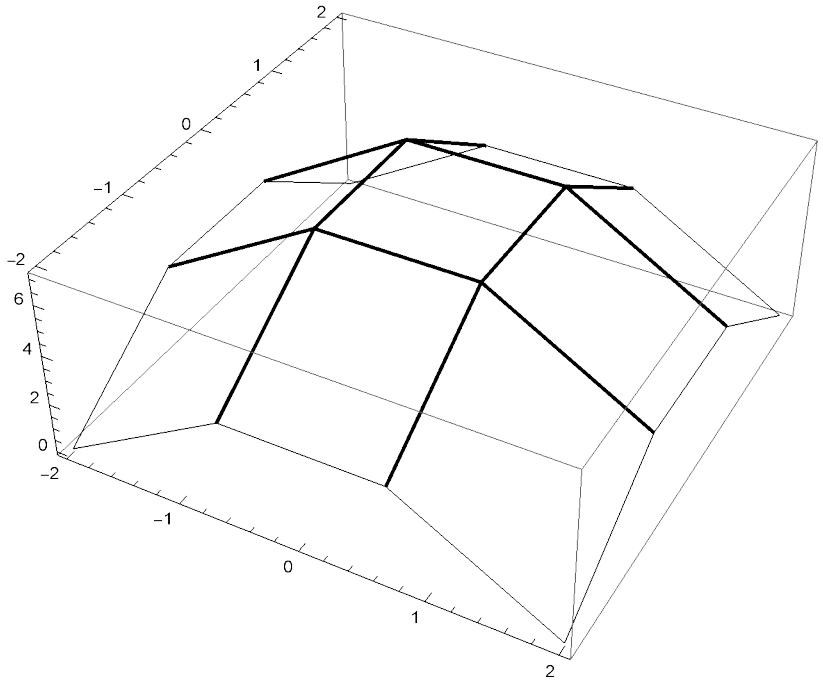
Surface Interpolation

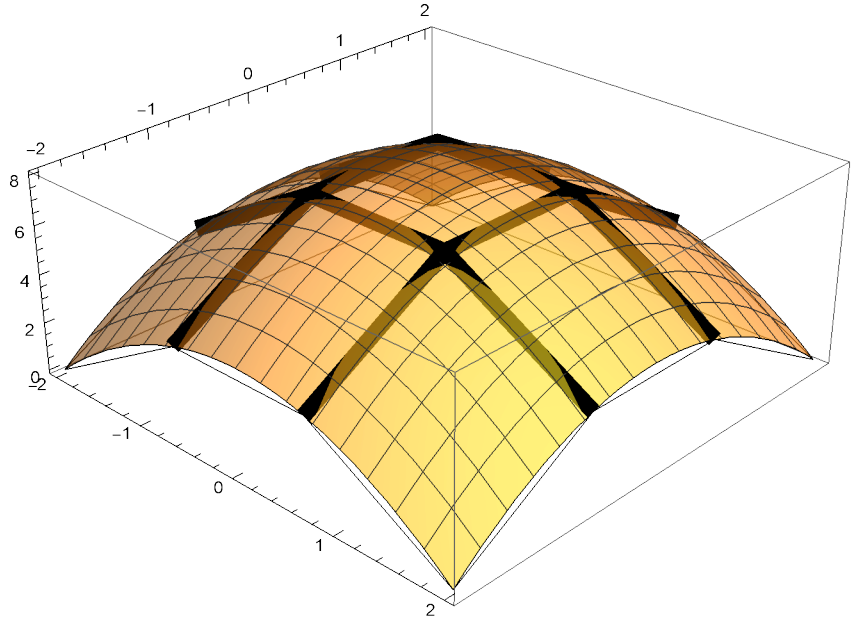




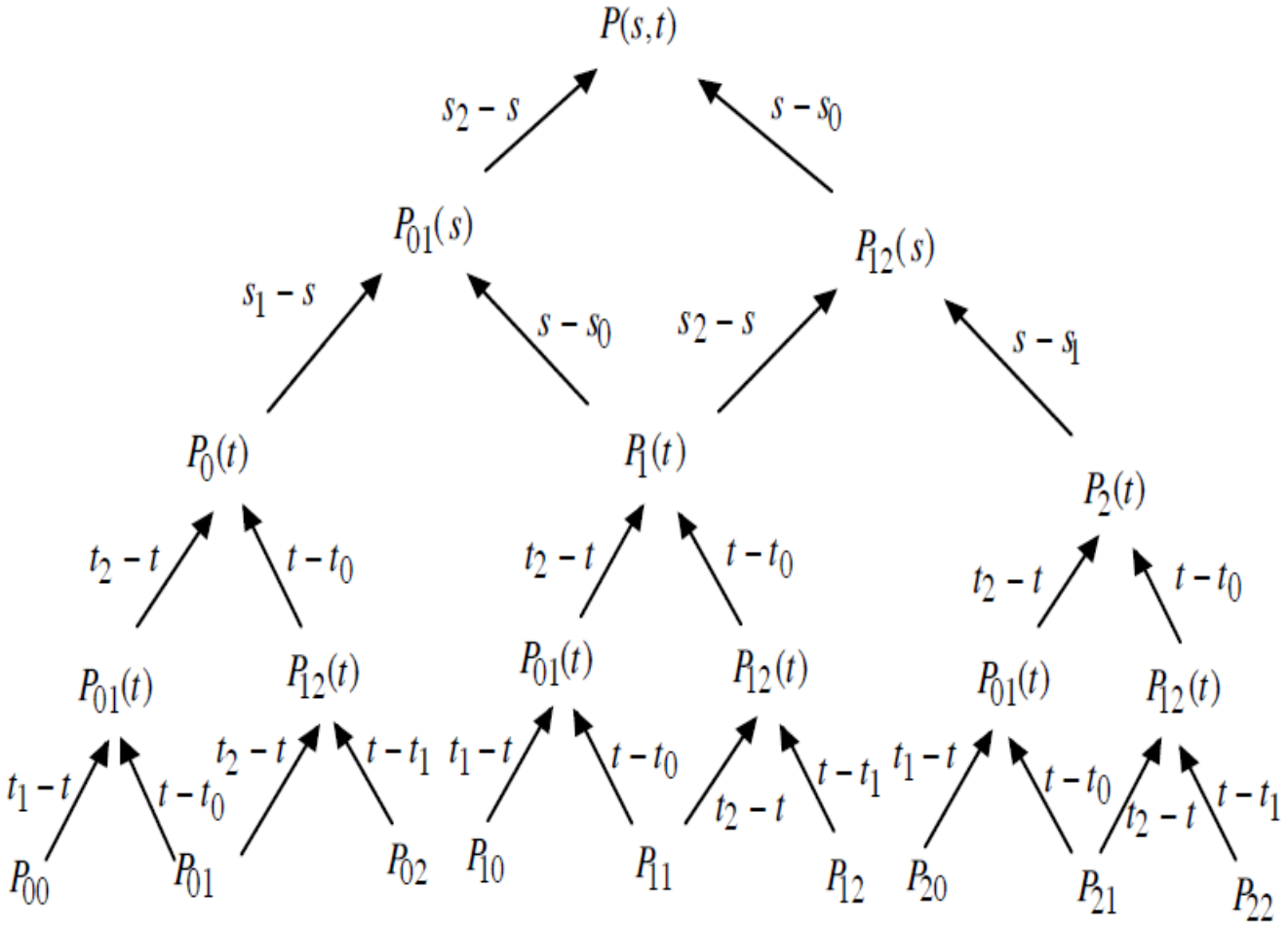


Surface Interpolation





Neville's Algorithm



Neville's Algorithm

Out[ ]//TraditionalForm=

$$\left\{\begin{array}{ll} P_{01,s=0}(t) := \frac{(t_1-t)\,P_{00}}{t_1-t_0} + \frac{(t-t_0)\,P_{01}}{t_1-t_0} & P_{01,s=1}(\mathrm{t\_}) := \frac{(t_1-t)\,P_{10}}{t_1-t_0} + \frac{(t-t_0)\,P_{11}}{t_1-t_0} \\ P_{12,s=0}(\mathrm{t\_}) := \frac{(t_2-t)\,P_{01}}{t_2-t_1} + \frac{(t-t_1)\,P_{02}}{t_2-t_1} & P_{12,s=1}(\mathrm{t\_}) := \frac{(t_2-t)\,P_{11}}{t_2-t_1} + \frac{(t-t_1)\,P_{12}}{t_2-t_1} \end{array}\right.\left.\begin{array}{l} P_{01,s=2}(\mathrm{t\_}) := \frac{(t_1-t)\,P_{20}}{t_1-t_0} + \frac{(t-t_0)\,P_{21}}{t_1-t_0} \\ P_{12,s=2}(\mathrm{t\_}) := \frac{(t_2-t)\,P_{21}}{t_2-t_1} + \frac{(t-t_1)\,P_{22}}{t_2-t_1} \end{array}\right\}$$

Out[ ]//TraditionalForm=

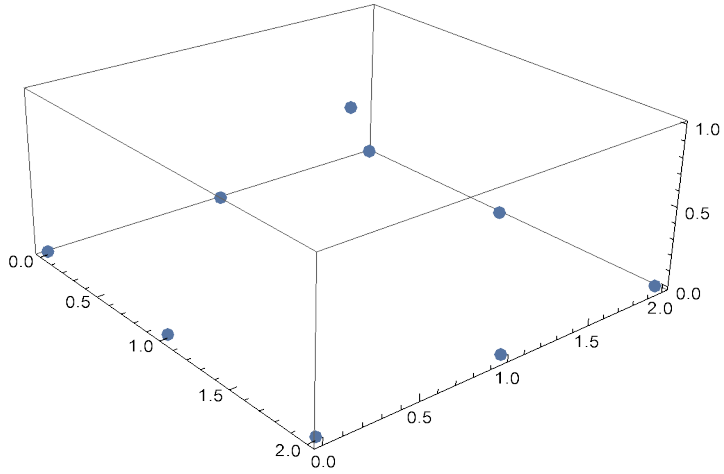
$$\left\{\begin{array}{l} P_{0,s=0}(\mathrm{t\_}) := \frac{(t_2-t)\,P_{01,s=0}(t)}{t_2-t_0} + \frac{(t-t_0)\,P_{12,s=0}(t)}{t_2-t_0} \\ P_{1,s=1}(\mathrm{t\_}) := \frac{(t_2-t)\,P_{01,s=1}(t)}{t_2-t_0} + \frac{(t-t_0)\,P_{12,s=1}(t)}{t_2-t_0} \\ P_{2,s=2}(\mathrm{t\_}) := \frac{(t_2-t)\,P_{01,s=2}(t)}{t_2-t_0} + \frac{(t-t_0)\,P_{12,s=2}(t)}{t_2-t_0} \end{array}\right., \left.\begin{array}{l} P_{01}(s,\,t) := \frac{(s_2-s)\,P_{0,s=0}(t)}{s_1-s_0} + \frac{(s-s_0)\,P_{1,s=1}(t)}{s_1-s_0} \\ P_{12}(s,\,t) := \frac{(s_2-s)\,P_{1,s=1}(t)}{s_2-s_1} + \frac{(s-s_1)\,P_{2,s=2}(t)}{s_2-s_1} \end{array}\right\}$$

Out[ ]//TraditionalForm=

$$P(s,\,t) := \frac{(s_2-s)\,P_{01}(s,\,t)}{s_2-s_0} + \frac{(s-s_0)\,P_{12}(s,\,t)}{s_2-s_0}$$

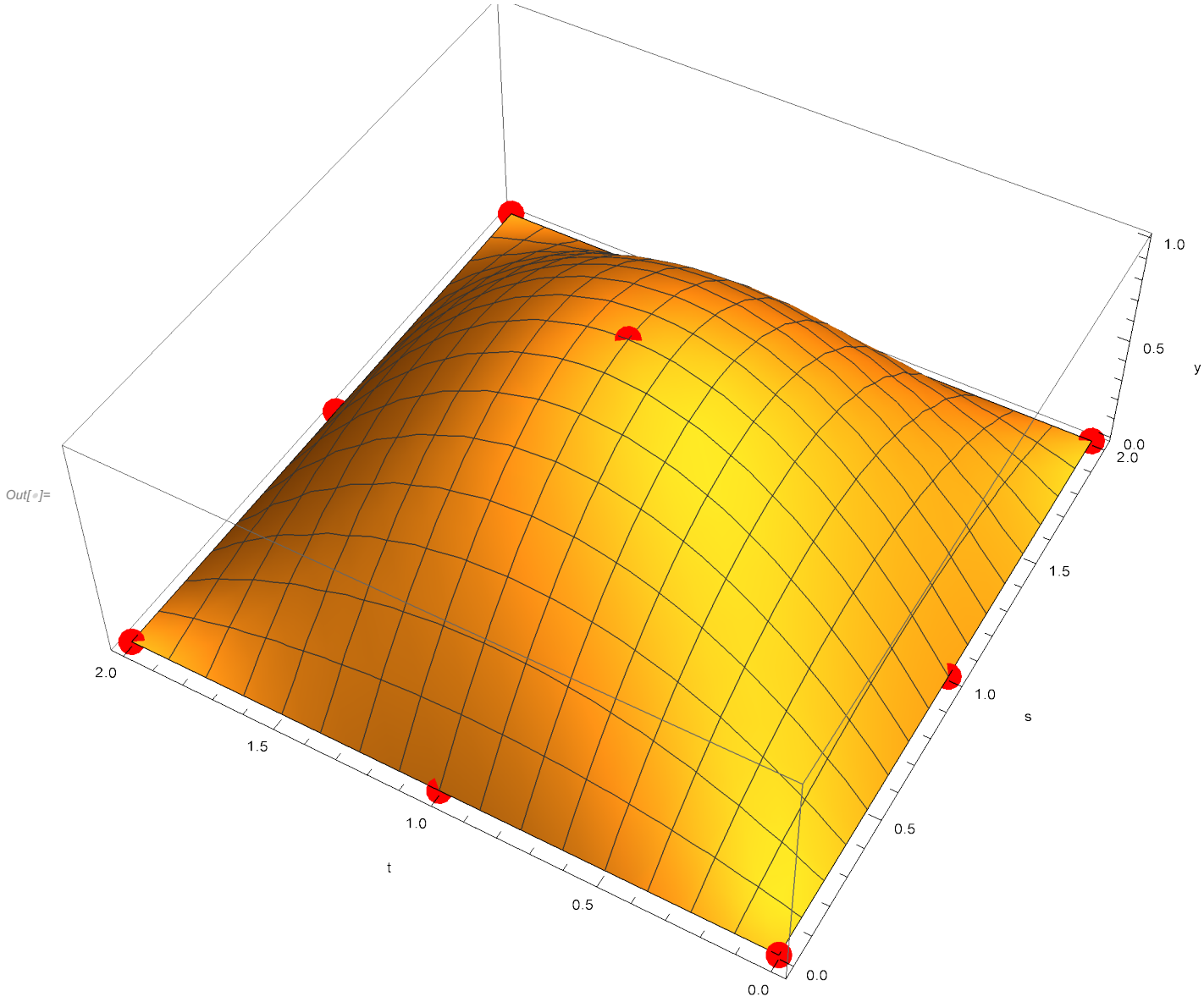
Example: Surface Interpolation

We start with a set of points:



Using the algorithm from the previous slides we construct a quadratic surface through this points.

Out[ ]=



Spline Interpolation

A spline is a polynomial between **each pair** of table points, but one whose coefficients are determined “slightly” non locally. The non-locality is designed to guarantee global smoothness in the interpolated function up to some order of derivative. Cubic splines are the most popular. They produce an interpolated function that is continuous through the second derivative. Splines tend to be stabler than polynomials, with less possibility of wild oscillation between the tabulated points.

In mathematics, a spline is a numeric function that is piecewise-defined by polynomial functions, and which possesses a high degree of smoothness at the places where the polynomial pieces connect (at the nodes).

Introductory Example

In[ ]:=
```
xdat = {1, 2, 3};
ydat = {2, 2, 4};
```

In[ ]:=
```
f[x_] := a0 + a1 x + a2 x^2 + a3 x^3
f'[x]
f''[x]
g[x_] := b0 + b1 x + b2 x^2 + b3 x^3
g'[x]
g''[x]
```

Out[ ]= $a1 + 2\,a2\,x + 3\,a3\,x^2$

Out[ ]= $2\,a2 + 6\,a3\,x$

Out[ ]= $b1 + 2\,b2\,x + 3\,b3\,x^2$

Out[ ]= $2\,b2 + 6\,b3\,x$

Introductory Example

In[•]:=
```
equations = {
    f[xdat[[1]]] == ydat[[1]],
    f[xdat[[2]]] == ydat[[2]],
    f'[xdat[[2]]] == g'[xdat[[2]]],
    f''[xdat[[2]]] == g''[xdat[[2]]],
    g[xdat[[2]]] == ydat[[2]],
    g[xdat[[3]]] == ydat[[3]],
    f''[xdat[[1]]] == 0,
    g''[xdat[[3]]] == 0
   };
Column[equations]
```

Out[•]=
```
a0 + a1 + a2 + a3 == 2
a0 + 2 a1 + 4 a2 + 8 a3 == 2
a1 + 4 a2 + 12 a3 == b1 + 4 b2 + 12 b3
2 a2 + 12 a3 == 2 b2 + 12 b3
b0 + 2 b1 + 4 b2 + 8 b3 == 2
b0 + 3 b1 + 9 b2 + 27 b3 == 4
2 a2 + 6 a3 == 0
2 b2 + 18 b3 == 0
```

Introductory Example

```
In[•]:= solution = Solve[equations, {a0, a1, a2, a3, b0, b1, b2, b3}]
```

$$\text{Out[•]= } \left\{\left\{a0 \to 2,\ a1 \to 1,\ a2 \to -\frac{3}{2},\ a3 \to \frac{1}{2},\ b0 \to 10,\ b1 \to -11,\ b2 \to \frac{9}{2},\ b3 \to -\frac{1}{2}\right\}\right\}$$

```
In[•]:= f[x] /. solution
        g[x] /. solution
```

$$\text{Out[•]= } \left\{2 + x - \frac{3\,x^2}{2} + \frac{x^3}{2}\right\}$$

$$\text{Out[•]= } \left\{10 - 11\,x + \frac{9\,x^2}{2} - \frac{x^3}{2}\right\}$$

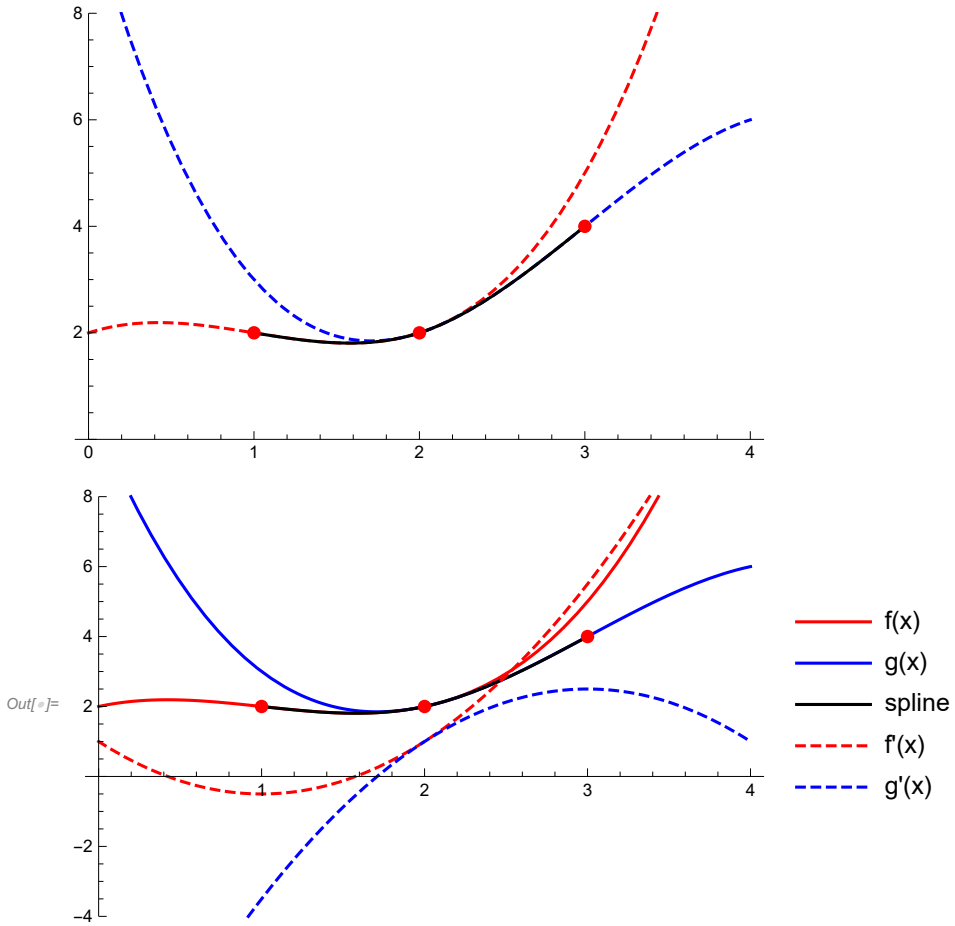
Introductory Example

In[•]:=
```
spline = Piecewise[{{f[x] /. solution, xdat[[1]] ≤ x ≤ xdat[[2]]}, {g[x] /. solution, xdat[[2]] ≤ x ≤ xdat[[3]]}}, Indeterminate]
```

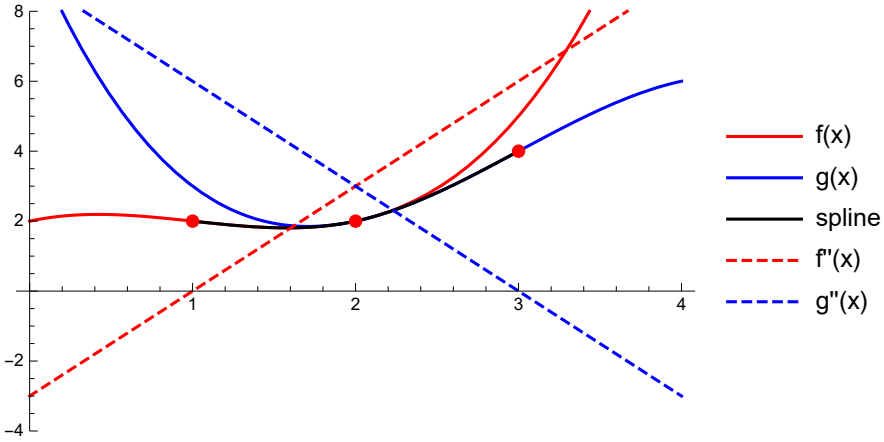
Out[•]=
$$\begin{cases} \left\{2 + x - \frac{3\,x^2}{2} + \frac{x^3}{2}\right\} & 1 \le x \le 2 \\ \left\{10 - 11\,x + \frac{9\,x^2}{2} - \frac{x^3}{2}\right\} & 2 \le x \le 3 \\ \text{Indeterminate} & \text{True} \end{cases}$$

Introductory Example

Out[ ]=



Out[ ]=



Cubic Spline Interpolation

Given a tabulated function $y_i = y(x_i)$, $i = 1, \ldots, N$, focus attention on one particular interval, between x_j and x_{j+1}. Linear interpolation gives the interpolation formula:

Out[•]=
$$y = A\, y_j + b\, y_{j+1} \quad \text{where} \quad A = \frac{x_{j+1} - x}{x_{j+1} - x_j} \quad B = 1 - A = \frac{x - x_j}{x_{j+1} - x_j}$$

Since it is (piecewise) linear, the equation has zero second derivative in the interior of each interval, and an undefined, or infinite, second derivative at the abscissas x_j . The goal of cubic spline interpolation is to get an interpolation formula that is smooth in the first derivative, and continuous in the second derivative, both within an interval and at its boundaries.

Suppose, we also have tabulated values for the function's second derivatives y'', that is a set of numbers y_i''. Then, within each interval, we can add to the right hand side of the equation above a cubic polynomial whose second derivative varies linearly from a value y_j'' on the left to a value y_{j+1}'' on the right.

Cubic Spline Interpolation

Doing so, we will have the desired continuous second derivative. If we also construct the cubic polynomial to have zero values at x_j and x_{j+1}, then adding it will not spoil the agreement with the tabulated functional values y_j and y_{j+1} at the endpoints x_j and x_{j+1}.

Out[•]//TraditionalForm=

$$y = A\,y_j + B\,y_{j+1} + C\,y_j'' + \mathrm{D}\,y_{j+1}''$$

with

Out[•]= $$C = \frac{1}{6}\left(A^3 - A\right)\,(x_{j+1} - x_j)^2 \qquad \mathrm{D} = \frac{1}{6}\left(B^3 - B\right)\,(x_{j+1} - x_j)^2$$

As a check we take the derivatives:

Out[•]= $$\frac{\mathrm{d}A}{\mathrm{d}x} = -\frac{1}{x_{j+1} - x_j} \qquad \frac{\mathrm{d}B}{\mathrm{d}x} = \frac{1}{x_{j+1} - x_j} \qquad \frac{\mathrm{d}C}{\mathrm{d}x} = \frac{\left(3A^2 - 1\right)\,(x_{j+1} - x_j)^2}{6\,(x_{j+1} - x_j)} \qquad \frac{\mathrm{d}\mathrm{D}}{\mathrm{d}x} = \frac{\left(3B^2 - 1\right)\,(x_{j+1} - x_j)^2}{6\,(x_{j+1} - x_j)}$$

Cubic Spline Interpolation

Out[•]//TraditionalForm=

$$\frac{dy}{dx} = \frac{y_{j+1} - y_j}{x_{j+1} - x_j} - \frac{3A^2 - 1}{6}(x_{j+1} - x_j)\, y''_j + \frac{3B^2 - 1}{6}(x_{j+1} - x_j)\, y''_{j+1}$$

Out[•]//TraditionalForm=

$$\frac{d^2y}{dx^2} = A\, y''_j + B\, y''_{j+1}$$

$A = 1$ at x_j and $A = 0$ at x_{j+1} while $B = 0$ at x_j and $B = 1$ at x_{j+1}, therefore y'' is just the tabulated second derivative, and it will be continuous across the boundary between the two intervals (x_{j-1}, x_j) and (x_j, x_{j+1}).

Cubic Spline Interpolation

Problem: we assume we know y'' but we don't!

Solution: Require that the first derivative is continuous across the boundary between the two intervals.

Set y'' for $x = x_j$ in the interval (x_{j-1}, x_j) equal to $x = x_j$ in the interval (x_j, x_{j+1}):

Out[ ]//TraditionalForm=

$$\frac{1}{6}(x_j - x_{j-1})\, y''_{j-1} + \frac{1}{3}(x_{j+1} - x_{j-1})\, y''_j + \frac{1}{6}(x_{j+1} - x_j)\, y''_{j+1} = \frac{x_{j+1} - y_j}{x_{j+1} - x_j} - \frac{y_j - y_{j-1}}{x_j - x_{j-1}}$$

$N - 2$ linear equations in the N unknowns y''_i, $i = 1, \ldots, N$.

Note, that $x_j - x_{j-1} = h_{j-1}$ is the spacing between data points, that $\frac{y_j - y_{j-1}}{x_j - x_{j-1}} = d_0$ is the slope (1st derivative).

For a unique solution we need to specify two further conditions, typically taken as boundary conditions at x_1 and x_N:

Cubic Spline Interpolation

- set one or both of y_1'' and y_N'' equal to zero ⇒ natural cubic spline
- set either of y_1'' and y_N'' to values calculated with the equation from above for a specified first derivative on the boundary ⇒ clamped spline
- requires that the third derivative of the spline is continuous at x_1 and x_{N-1}

Cubic Spline Interpolation

Tri-diagonal set of equations, i.e. each y_j'' is only coupled to its nearest neighbors at $j \pm 1$. Equations can be solved in $O(N)$ operations.

Implementation from Numerical Recipes: `spline` and `splint`. Call `spline` once to compute the polynomial coefficients and then use only `splint`.

Spline Interpolation - Worked Example

We would like to use a spline to approximate a function represented by the points (0,0), (1,0), (3,2), and (4,2). The first task is to determine the spacing between the points h_k, the slopes d_k and the second derivatives $y''(x_k) = m_k$.

$h_0 = x_1 - x_0 = 1 \qquad h_1 = x_2 - x_1 = 2 \qquad h_2 = x_3 - x_2 = 1$

Out[•]=

$d_0 = \frac{y_1 - y_0}{x_1 - x_0} = 0 \qquad d_1 = \frac{y_2 - y_1}{x_2 - x_1} = 1 \qquad d_2 = \frac{y_3 - y_2}{x_3 - x_2} = 0$

Next, set up the equations for the second derivatives (m_k) which have been derived by requiring the continuity of the first and second derivatives of the splines at x_1 and x_2:

$2\left(h_0 + h_1\right) m_1 + h_1 m_2 = 6\left(d_1 - d_0\right) - h_0 m_0$

Out[•]=

$h_1 m_1 + 2\left(h_1 + h_2\right) m_2 = 6\left(d_2 - d_1\right) - h_2 m_3$

Spline Interpolation - Worked Example

There will be $N - 1$ of these equations when the spline interpolates $N + 1$ points. Applying the end conditions:

Natural Spline (second derivative zero)

$$m_0 = 0$$

Out[•]=

$$h_0 m_0 + 2\left(h_0 + h_1\right) m_1 + h_1 m_1 = 6\left(d_1 - d_0\right)$$

$$h_1 m_1 + 2\left(h_1 + h_2\right) m_2 + h_2 m_3 = 6\left(d_2 - d_1\right)$$

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 1 & 6 & 2 & 0 \\ 0 & 2 & 6 & 1 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} m_0 \\ m_1 \\ m_2 \\ m_3 \end{pmatrix} = \begin{pmatrix} 0 \\ 6 \\ -6 \\ 0 \end{pmatrix}$$

$$m_3 = 0$$

Spline Interpolation - Worked Example

Clamped Spline (first derivatives set at end points)

Out[•]=

$$2\,h_0\,m_0 + h_0\,m_1 = 6\,\left(d_0 - y'(x_0)\right)$$

$$h_0\,m_0 + 2\,\left(h_0 + h_1\right)\,m_1 + h_1\,m_1 = 6\,\left(d_1 - d_0\right)$$

$$h_1\,m_1 + 2\,\left(h_1 + h_2\right)\,m_2 + h_2\,m_3 = 6\,\left(d_2 - d_1\right)$$

$$h_2\,m_2 + 2\,h_2\,m_3 = 6\,\left(y'(x_3) - d_2\right)$$

$$\begin{pmatrix} 2 & 1 & 0 & 0 \\ 1 & 8 & 0 & 0 \\ 0 & 2 & 6 & 1 \\ 0 & 0 & 1 & 2 \end{pmatrix} \begin{pmatrix} m_0 \\ m_1 \\ m_2 \\ m_3 \end{pmatrix} = \begin{pmatrix} 0 \\ -6 \\ 6 \\ 0 \end{pmatrix}$$

Spline Interpolation - Worked Example

Not-a-knot Spline (3rd derivative continuous, i.e. $y_0'''(x_1) = y_1'''(x_1)$

Out[•]=

$$h_1 m_0 + \left(h_0 + h_1\right) m_1 + h_0 m_2 = 0$$

$$h_0 m_0 + 2\left(h_0 + h_1\right) m_1 + h_1 m_1 = 6\left(d_1 - d_0\right)$$

$$h_1 m_1 + 2\left(h_1 + h_2\right) m_2 + h_2 m_3 = 6\left(d_2 - d_1\right)$$

$$h_2 m_1 + \left(h_1 + h_2\right) m_2 + h_1 m_3 = 0$$

$$\begin{pmatrix} 1 & -3 & 2 & 0 \\ 1 & 6 & 2 & 0 \\ 0 & 2 & 6 & 1 \\ 0 & 2 & -3 & 1 \end{pmatrix} \begin{pmatrix} m_0 \\ m_1 \\ m_2 \\ m_3 \end{pmatrix} = \begin{pmatrix} 0 \\ 6 \\ -6 \\ 0 \end{pmatrix}$$

Spline Interpolation - Worked Example

After solving the appropriate system for the end conditions that you have chosen, evaluate the coefficients of the splines

Out[•]//TraditionalForm=

$$y(x) = \begin{pmatrix} y_0(x) = y_{0,0} + y_{0,1}\,(x - x_0) + y_{0,2}\,(x - x_0)^2 + y_{0,3}\,(x - x_0)^3 & x \in \{x_0, x_1\} \\ y_1(x) = y_{1,0} + y_{1,1}\,(x - x_1) + y_{1,2}\,(x - x_1)^2 + y_{1,3}\,(x - x_1)^3 & x \in \{x_1, x_2\} \\ y_2(x) = y_{2,0} + y_{2,1}\,(x - x_2) + y_{2,2}\,(x - x_2)^2 + y_{2,3}\,(x - x_2)^3 & x \in \{x_2, x_3\} \end{pmatrix}$$

where

Out[•]= $y_{k,0} = y_k \qquad y_{k,1} = d_k - \frac{1}{6} h_k \left(2 m_k + m_{k+1}\right) \qquad y_{k,2} = \frac{m_k}{2} \qquad y_{k,3} = \frac{m_{k+1} - m_k}{6 h_k}$

Spline Interpolation - Worked Example

Natural Spline:

In[•]:=
$$\texttt{LinearSolve}\left[\begin{pmatrix}1&0&0&0\\1&6&2&0\\0&2&6&1\\0&0&0&1\end{pmatrix},\begin{pmatrix}0\\6\\-6\\0\end{pmatrix}\right]$$

Out[•]=
$$\left\{\{0\},\left\{\frac{3}{2}\right\},\left\{-\frac{3}{2}\right\},\{0\}\right\}$$

Spline Interpolation - Worked Example

Clamped Spline:

In[•]:=
```
LinearSolve[{{2, 1, 0, 0}, {1, 8, 0, 0}, {0, 2, 6, 1}, {0, 0, 1, 2}}, {{0}, {-6}, {6}, {0}}]
```

Out[•]=
$$\left\{\left\{\frac{2}{5}\right\}, \left\{-\frac{4}{5}\right\}, \left\{\frac{76}{55}\right\}, \left\{-\frac{38}{55}\right\}\right\}$$

Spline Interpolation - Worked Example

Not-A-knot Spline:

In[•]:=
```
LinearSolve[{{1, -3, 2, 0}, {1, 6, 2, 0}, {0, 2, 6, 1}, {0, 2, -3, 1}}, {0, 6, -6, 0}]
```

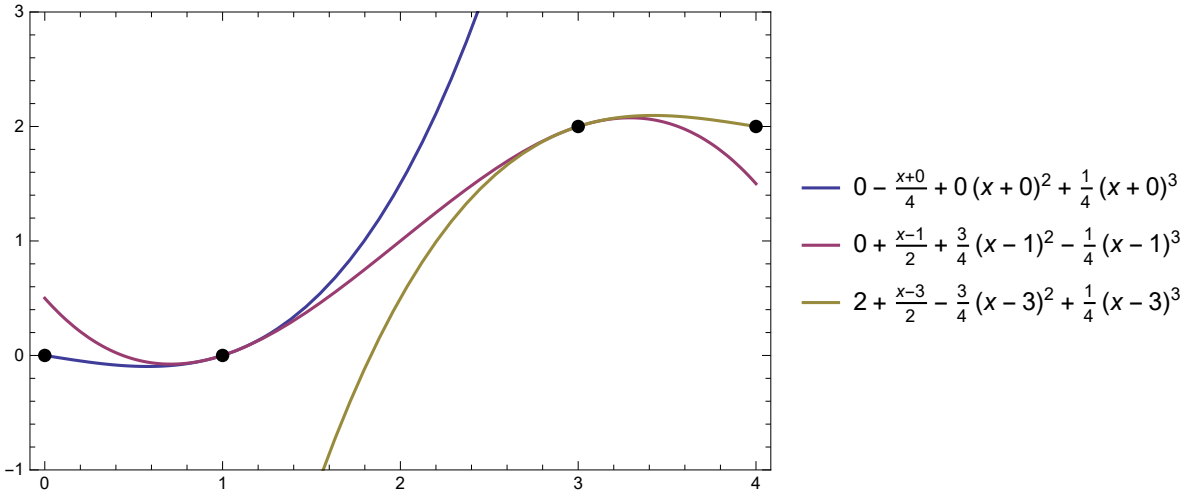
Out[•]=
$$\left\{\left\{\frac{10}{3}\right\}, \left\{\frac{2}{3}\right\}, \left\{-\frac{2}{3}\right\}, \left\{-\frac{10}{3}\right\}\right\}$$

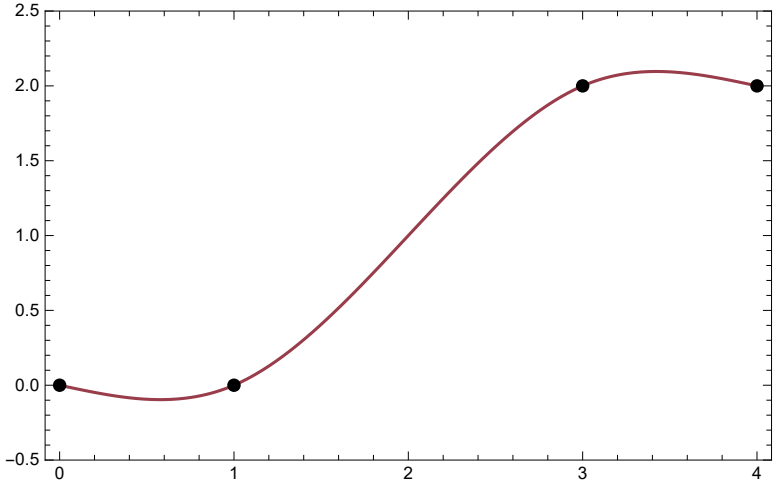
Spline Interpolation - Worked Example

So with: $m_0 = m_3 = 0,\ m_1 = 3/2,\ m_2 = -3/2$

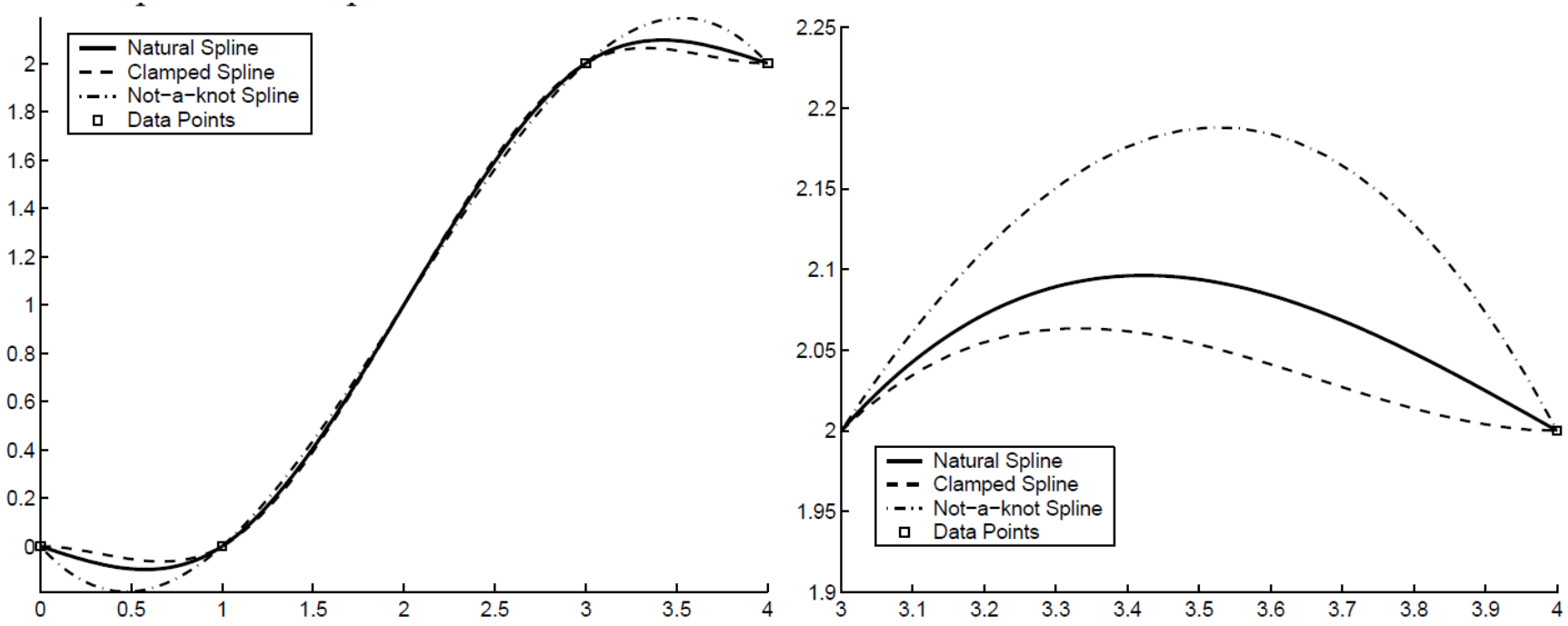
Out[•]//TraditionalForm=

$$y(x) = \begin{pmatrix} \begin{cases} y_0(x) = 0 - \frac{1}{4}(x-0) + 0(x-0)^2 + \frac{1}{4}(x-0)^3 & x \in \{x_0, x_1\} \\ y_1(x) = 0 + \frac{1}{2}(x-1) + \frac{3}{4}(x-1)^2 - \frac{1}{4}(x-1)^3 & x \in \{x_1, x_2\} \\ y_2(x) = 2 + \frac{1}{2}(x-3) - \frac{3}{4}(x-3)^2 + \frac{1}{4}(x-3)^3 & x \in \{x_2, x_3\} \end{cases} \end{pmatrix}$$


3
2
1
0
−1
0
1
2
3
4
$0 - \frac{x+0}{4} + 0\,(x+0)^2 + \frac{1}{4}(x+0)^3$
$0 + \frac{x-1}{2} + \frac{3}{4}(x-1)^2 - \frac{1}{4}(x-1)^3$
$2 + \frac{x-3}{2} - \frac{3}{4}(x-3)^2 + \frac{1}{4}(x-3)^3$


2.5
2.0
1.5
1.0
0.5
0.0
−0.5
0
1
2
3
4

Spline Interpolation - Worked Example



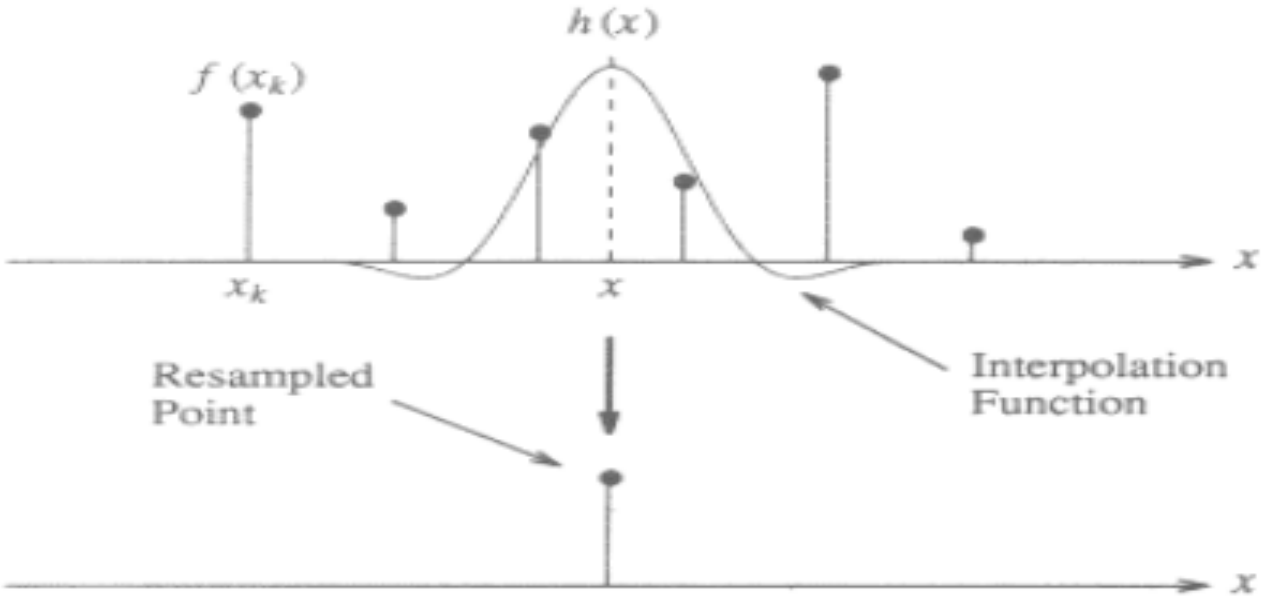
Interpolation as convolution

For equally spaced data, interpolation can be expressed as

Out[•]//TraditionalForm=

$$f(x) = \sum_{k=0}^{K-1} c_k \, h(x - x_k)$$

where h us the interpolation kernel, weighted by coefficients c_k and applied to K data samples, x_k. This formulates interpolation as a convolution operation. In practice h is nearly always a symmetric kernel, i.e. $h(-x) = h(x)$. In most cases, c_k are the data samples themselves.



In the examples we work in 1-D. Interpolation in 2-D is a simple extension of the 1-D case.

Interpolation Kernels

Nearest Neighbor

The simplest interpolation algorithm (from a computational standpoint) is the nearest neighbor algorithm, where each interpolated output is assigned the value of the nearest sample point in the input - also known as **point shift** algorithm.

Out[ ]//TraditionalForm=

$$f(x) = f(x_k) \qquad \frac{1}{2}(x_{k-1} + x_k) < x \leq \frac{1}{2}(x_k + x_{k+1})$$

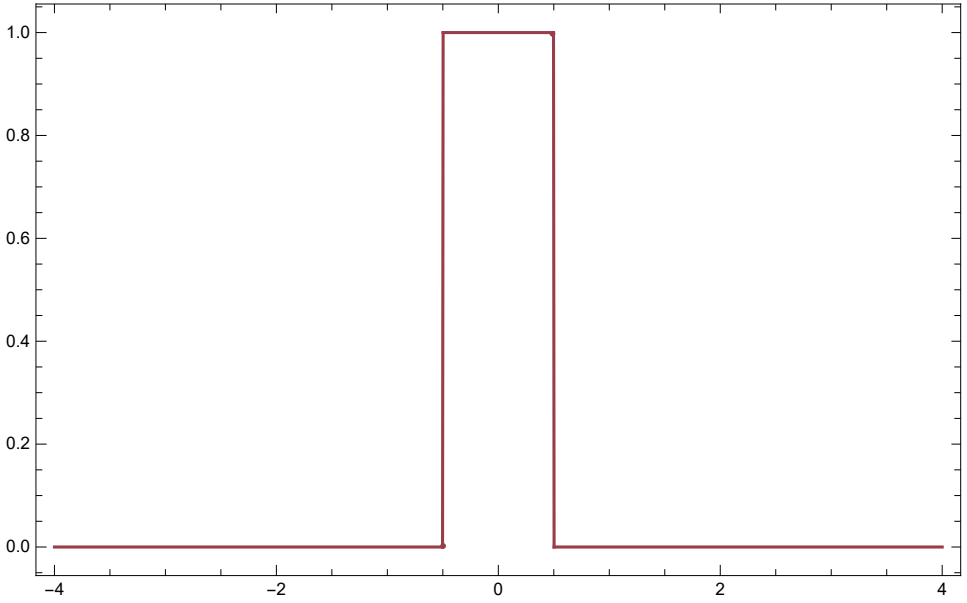
Nearest Neighbor

It can be achieved by convolving the data with a one-pixel width rectangle in the spatial domain, The interpolation kernel for the nearest neighbor is defined as:

Out[•]= $$h_{\Delta}(x) = \begin{cases} 1 & 0 \le |x| < \frac{\Delta}{2} \\ 0 & \frac{\Delta}{2} \le |x| \end{cases}$$

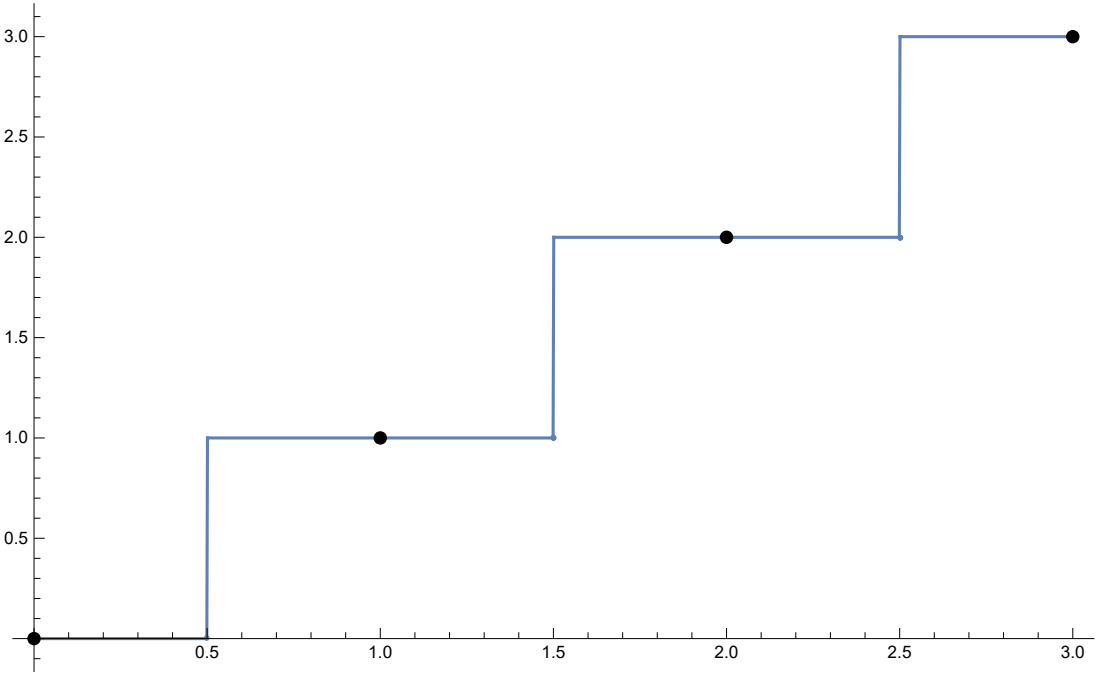
where Δ is the data spacing ($x_{k+1} - x_k$). Alternative names for this kernel are: box filter, sample-and-hold function, Fourier window.

Out[•]=



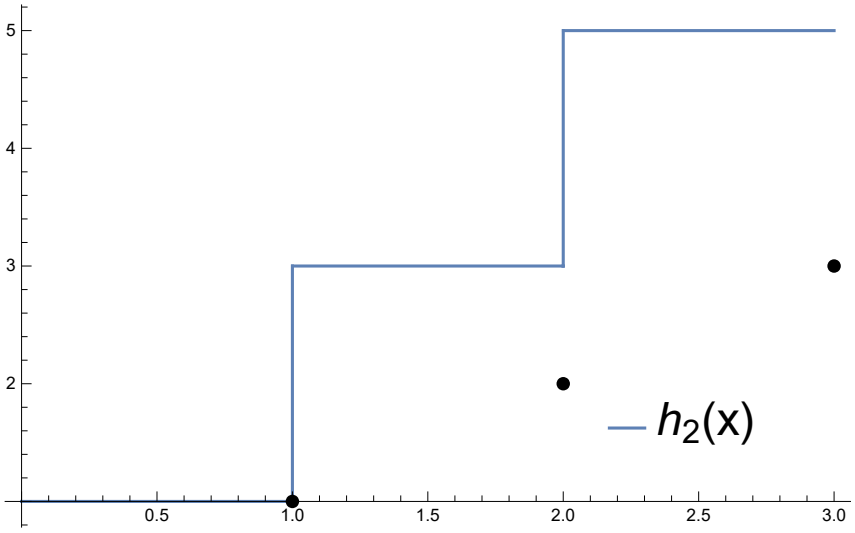
Nearest Neighbor

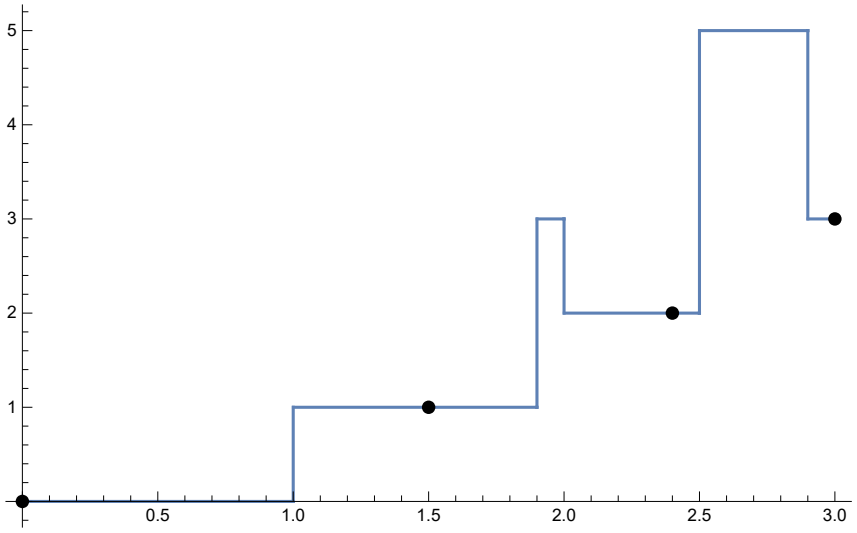
Out[ ]=



Nearest Neighbor

Danger ahead: the kernel $h(x)$ assumes a data spacing of 1! Applied to data with varying spacing gives strange results:





We will discuss interpolation of scattered data later.

Linear Interpolation (revisited)

Linear interpolation is a first-degree method that passes a straight line through every two consecutive points of the input signal. Given an interval (x_0, x_1) and function values f_0 and f_1 for the endpoints, the interpolating polynomial is

Out[•]//TraditionalForm=

$$f(x) = a_1 x + a_0$$

where a_0 and a_1 are determined by solving

Out[•]//TraditionalForm=

$$[f_0 \quad f_1] = [a_1 \quad a_0] \cdot \begin{pmatrix} x_0 & x_1 \\ 1 & 1 \end{pmatrix}$$

Linear Interpolation (revisited)

which gives

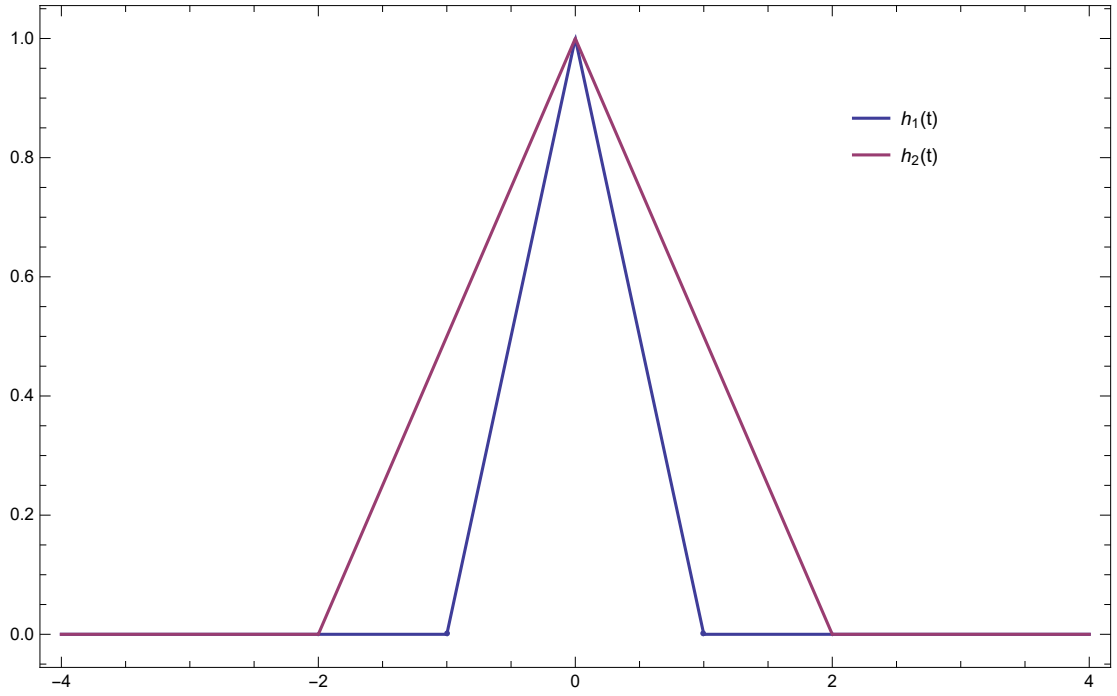
Out[ ]//TraditionalForm=

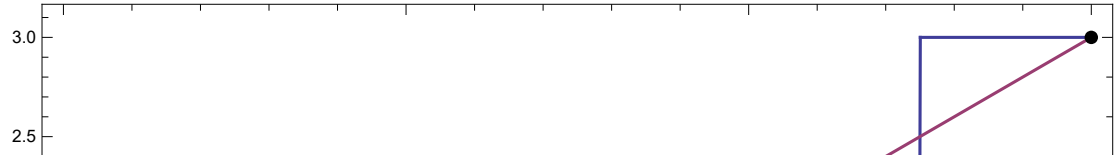
$$f(x) = f_0 + \left(\frac{x - x_0}{x_1 - x_0}\right)(f_1 - f_0)$$

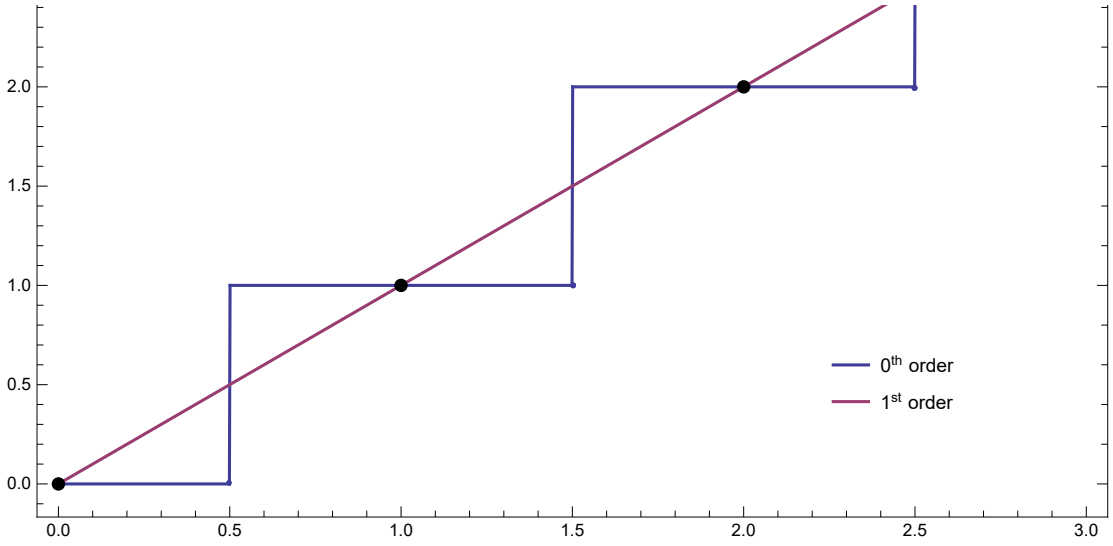
Out[ ]//TraditionalForm=

$$h_\Delta[x] = \begin{cases} 1 - \left|\frac{x}{\Delta}\right| & |x| \le \Delta \\ 0 & \text{True} \end{cases}$$

Out[ ]=

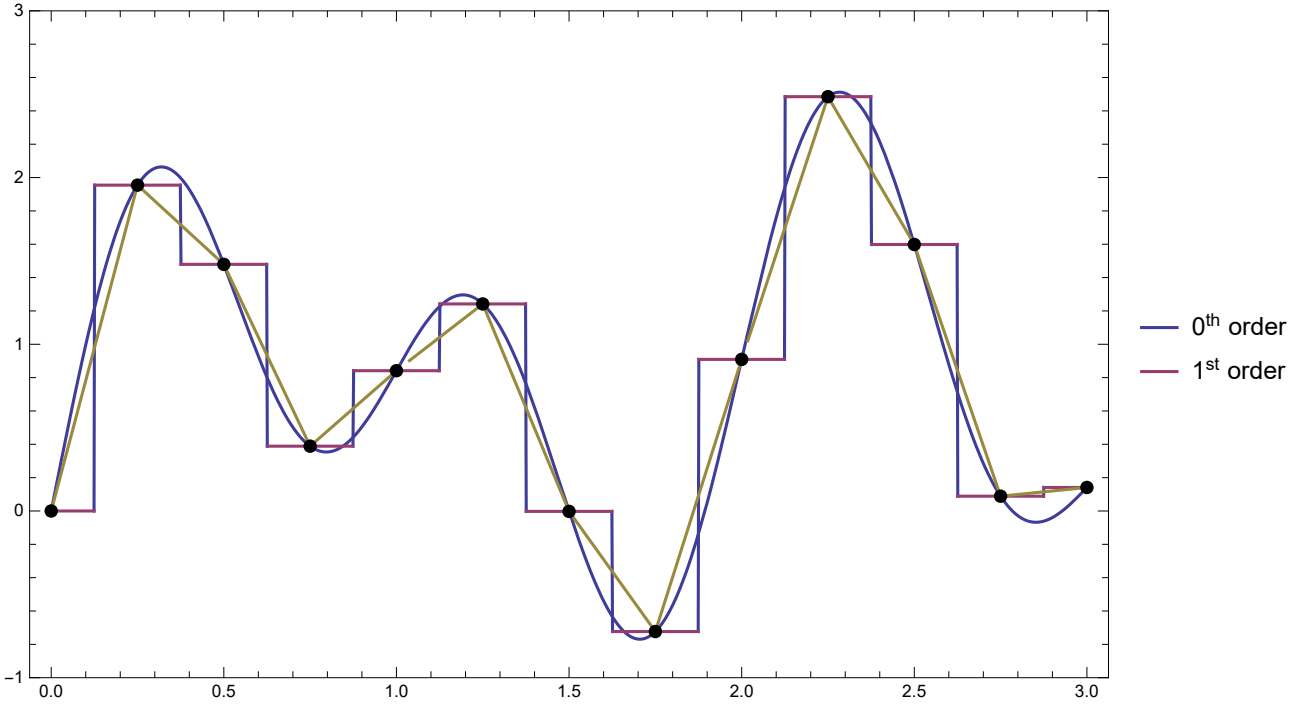





2.0
1.5
1.0
0.5
0.0
0.0
0.5
1.0
1.5
2.0
2.5
3.0
0th order
1st order

Linear Interpolation (revisited)

Other names for the kernel are: triangle filter, tent filter, roof function, Chateau function, or Bartlett window



Cubic Convolution

Out[•]//TraditionalForm=

$$h_{\Delta}[x]=\begin{cases} a_{30}|x|^3+a_{20}|x|^2+a_{10}|x|+a_0 & 0\le|x|<\Delta \\ a_{31}|x|^3+a_{21}|x|^2+a_{11}|x|+a_1 & \Delta\le|x|<2\Delta \\ 0 & 2\Delta\le|x| \end{cases}$$

The coefficients a are determined from the following constraints

- $h(0)=1$, $h(x)=0$ for $|x|=1$ and 2
- h must be continuous at $|x|=0$, 1, and 2
- h' must be continuous at $|x|=0$, 1, and 2

Cubic Convolution

The constraints given above have resulted in seven equations. However, there are eight unknown coefficients. This requires another constraint in order to obtain a unique solution. By allowing a = a_{31} to be a free parameter that may be controlled by the user, the family of solutions given below may be obtained.

Out[ ]//TraditionalForm=

$$h(x) = \begin{cases} (a+2)\,|x|^3 - (a+3)\,|x|^2 & 0 \le |x| < 1 \\ a\,|x|^3 + 5\,a\,|x|^2 + 8\,a\,|x| - 4\,a & 1 \le |x| < 2 \\ 0 & 2 \le |x| \end{cases}$$

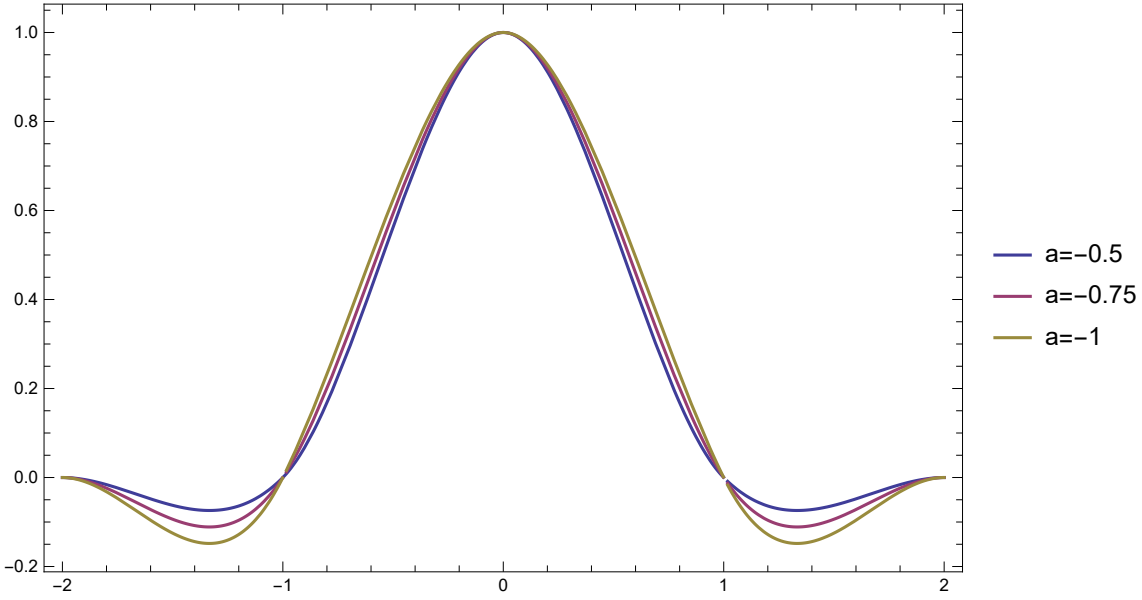
Cubic Convolution

By requiring h to be concave upward at |x| = 1, and concave downward at x = 0, we have

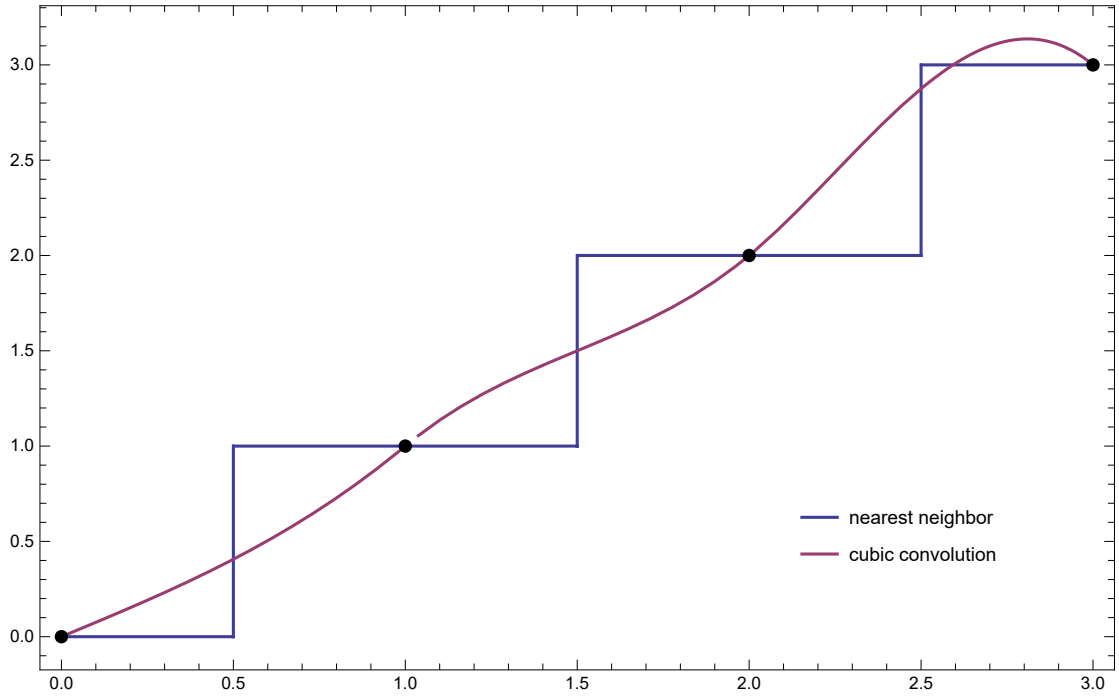
Out[ ]= $h''(0) == -2(a+4) < 0 \longrightarrow a > -3$

Out[ ]= $h''(1) == -4a > 0 \longrightarrow a < 0$

Bounding *a* to values between −3 and 0 makes *h* resemble the sinc function.

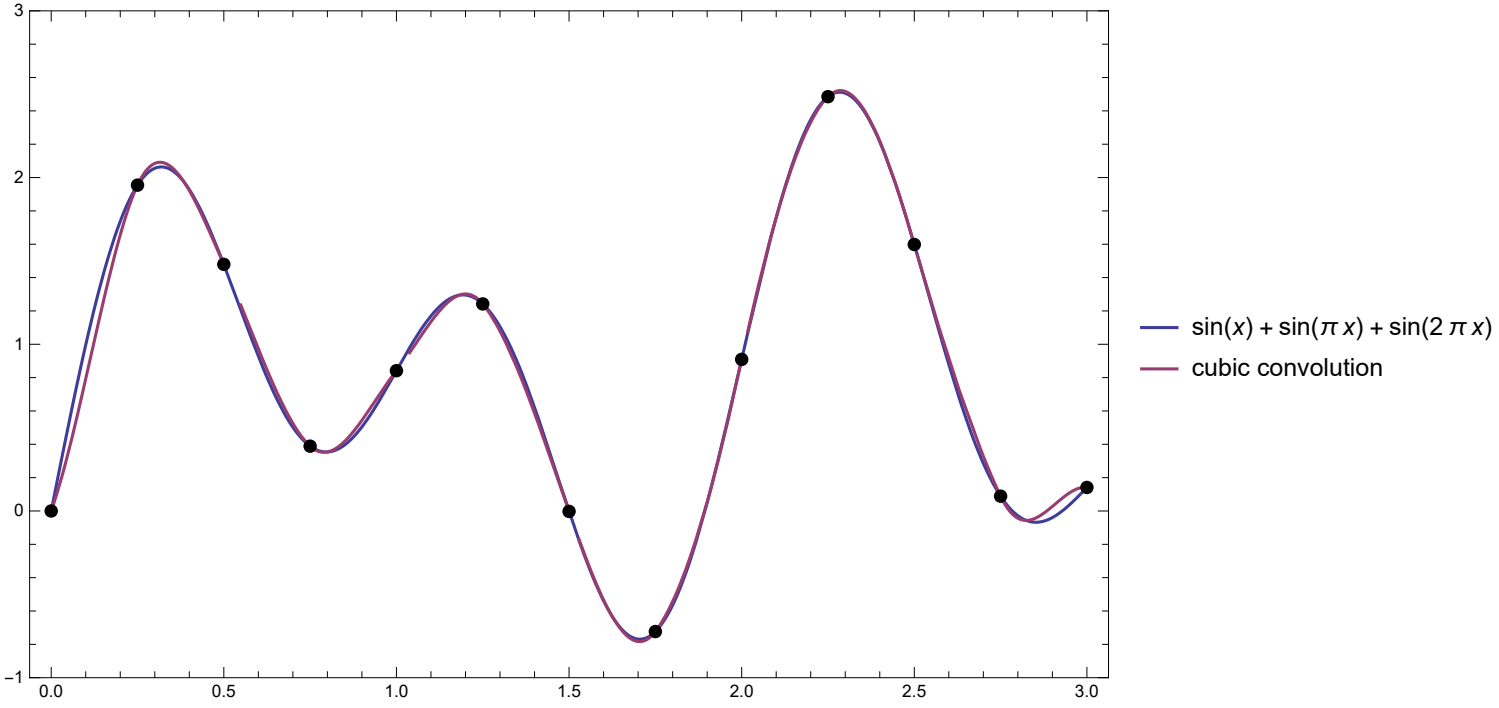


Cubic Convolution



Cubic Convolution

To apply the kernel to differently spaced data scale argument. E.g. a spacing of $\Delta x = 0.25 \rightarrow \Sigma c_k\, h(4\,(x - x_k))$



Radial Basis Functions

A radial basis function (RBF) is a real-valued function whose value depends only on the distance from the origin, so that $\phi(x) = \phi(\|x\|)$.

Out[•]//TraditionalForm=

$$f(x) = \sum_{k=1}^{N} c_k \, \phi(\|x - x_k\|)$$

Commonly used RBFs are (writing $r = \|x - x_k\|$)

- Gaussian

Out[•]//TraditionalForm=

$$\phi(r) = e^{-(\epsilon r)^2}$$

- Multi-quadratic

Out[•]//TraditionalForm=

$$\phi(r) = \sqrt{1 + (\epsilon r)^2}$$

Radial Basis Functions

- Inverse quadratic

Out[ ]//TraditionalForm=

$$\phi(r) = \frac{1}{1 + (\epsilon\, r)^2}$$

- Inverse multi-quadratic

Out[ ]//TraditionalForm=

$$\phi(r) = \frac{1}{\sqrt{1 + (\epsilon\, r)^2}}$$

- Poly-harmonic spline

Out[ ]//TraditionalForm=

$$\phi(r) = r^k \text{ , k=1,3,5,…}$$

Out[ ]//TraditionalForm=

$$\phi(r) = r^k \log(r) \text{ , k=2,4,6,…}$$

Radial Basis Functions

- Thin plate spline

Out[ ]//TraditionalForm=

$$\phi(r) = r^2 \log(r)$$

Radial Basis Functions

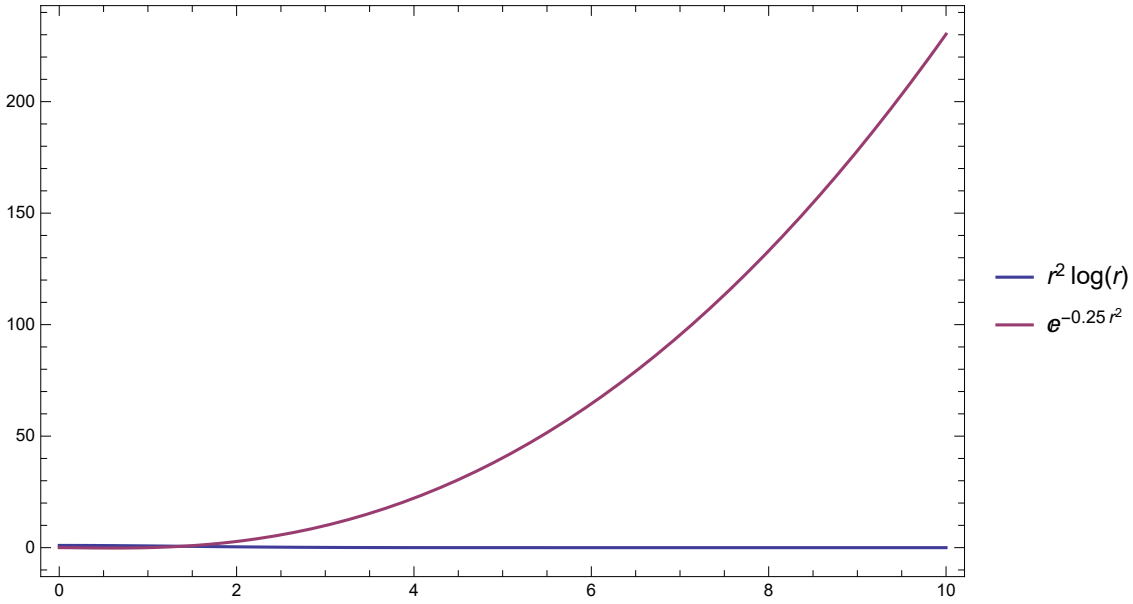
From the condition that the interpolation has to go through the data points follows:

Out[•]= $f(x) = \sum_{k=1}^{n} c_k \phi(\|x_k - x_i\|) = \sum_{k=1}^{n} c_k \phi(r_{ki})$ then $\begin{pmatrix} \phi(r_{11}) & \phi(r_{12}) & \dots & \phi(r_{1N}) \\ \phi(r_{21}) & \phi(r_{22}) & \dots & \phi(r_{2N}) \\ \vdots & \vdots & \ddots & \vdots \\ \phi(r_{N1}) & \phi(r_{N2}) & \dots & \phi(r_{NN}) \end{pmatrix} \begin{pmatrix} c_1 \\ c_2 \\ \vdots \\ c_n \end{pmatrix} = \begin{pmatrix} f(x_1) \\ f(x_2) \\ \vdots \\ f(x_N) \end{pmatrix}$

So computing the RBF interpolation requires the inversion of a n×n matrix and then summing all terms in a scalar product, i.e. it is of computational complexity $O(n^3)$ which basically prohibits its application on very large data sets.

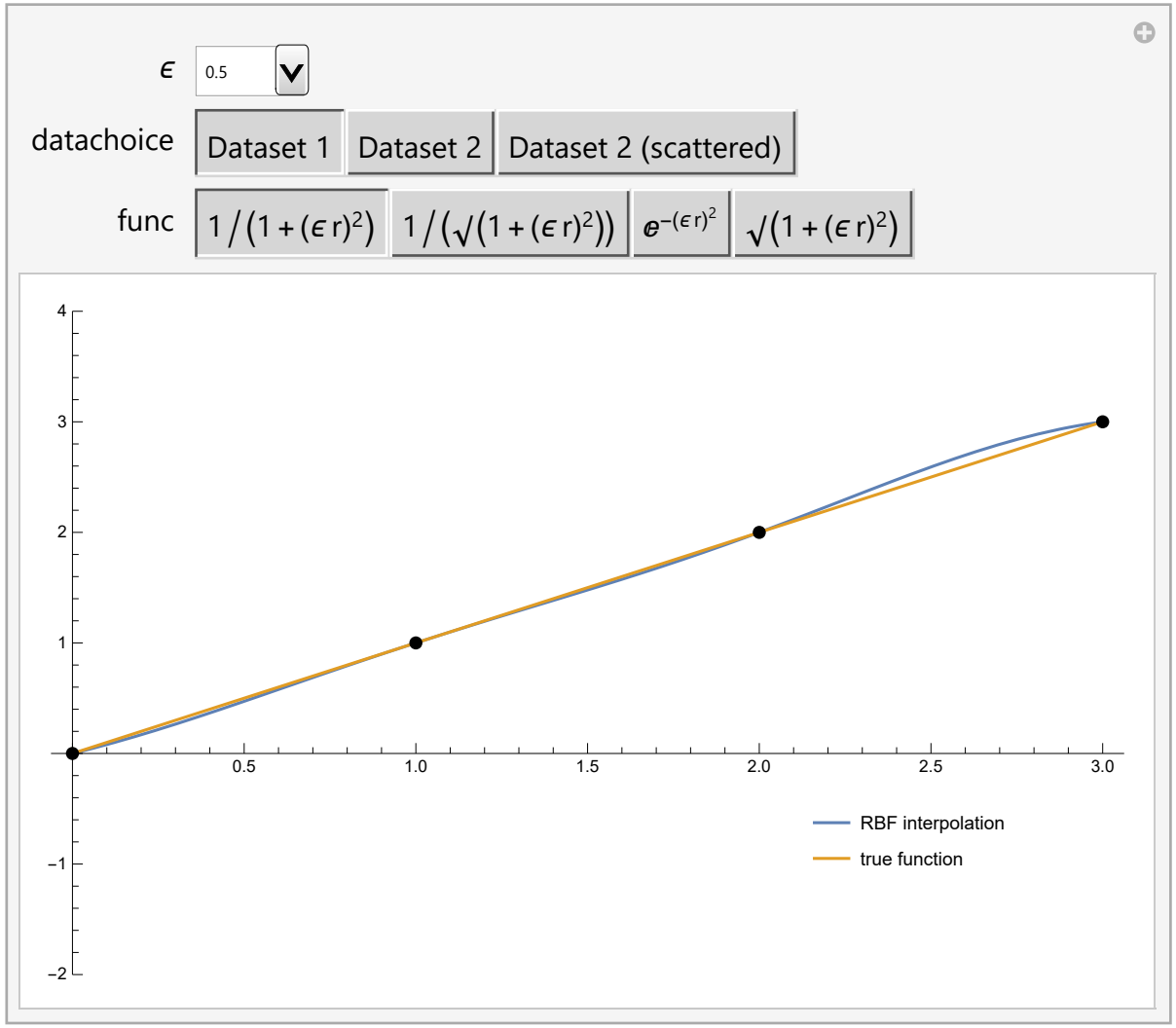
To circumvent this, one can use localization schemes, etc.

Example



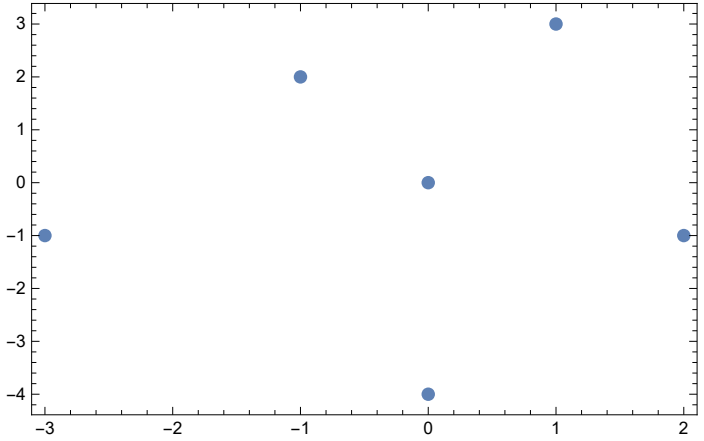
Choice of scale parameter ϵ and RBF form

Out[ ]=



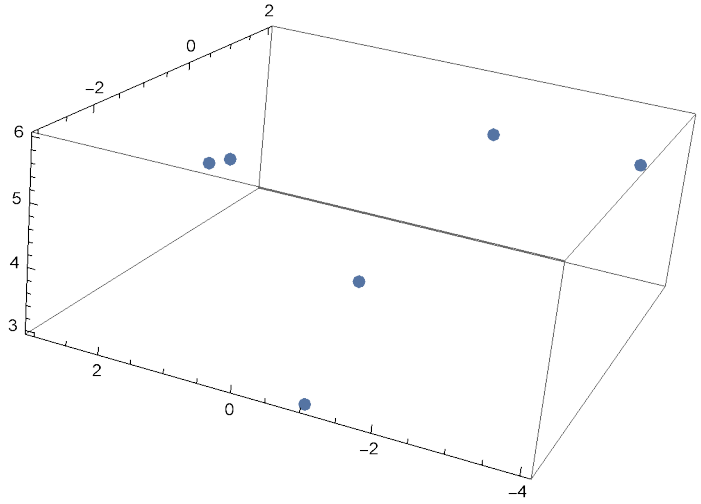
Radial Basis Function Interpolation - Example

In[•]:=
```
points = {{1, 3, 4}, {-1, 2, 5}, {-3, -1, 3}, {0, -4, 6}, {2, -1, 5}, {0, 0, 3}};

ListPlot[points[[All, {1, 2}]], Frame → True, Axes → False, PlotStyle → PointSize → Large]
```

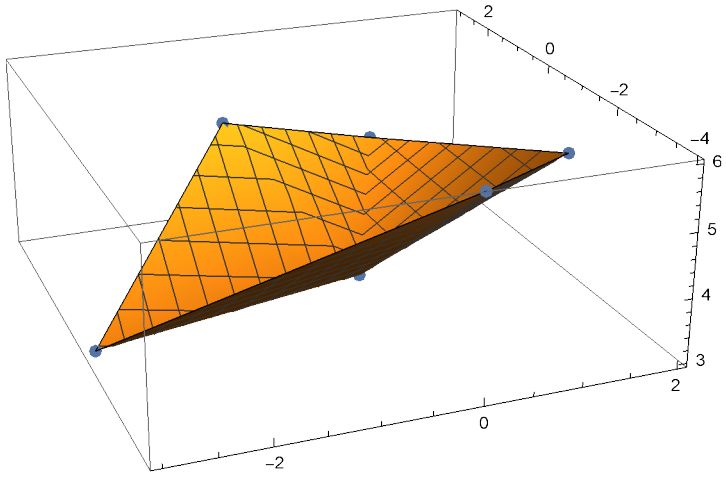


Radial Basis Functions

```
ListPointPlot3D[points, PlotStyle → PointSize → Large]
```



```
Show[{ListPlot3D[points, PlotStyle → PointSize → Large], ListPointPlot3D[points, PlotStyle → PointSize → Large]}]
```



Radial Basis Functions

```
In[•]:= rbf[x_] /; (x>0) := N[x^2 Log[x]]
        rbf[x_] /; (x == 0) := 0

In[•]:= unknownPoint = {-1, -1}

Out[•]= {-1, -1}

In[•]:= distances[{x_, y_}, data_] := EuclideanDistance[{x, y}, #] & /@ data[[All, {1, 2}]]

        distances[unknownPoint, points]

        {2 √5, 3, 2, √10, 3, √2}
```

Radial Basis Functions

```
rbf /@ distances[unknownPoint, points]
```

```
{29.9573, 9.88751, 2.77259, 11.5129, 9.88751, 0.693147}
```

In[ ]:=
```
inter[{x_, y_}, data_] := Total[rbf /@ distances[unknownPoint, points]]
```

In[ ]:=
```
inter[unknownPoint, points]
```

Out[ ]=
```
64.711
```

That looks wrong! But what we didn't force the interpolation through the data points!

Radial Basis Functions

In[•]:=
```
distMat = Table[EuclideanDistance[points[[i, {1, 2}]], points[[j, {1, 2}]]], {i, 1, Length[points]}, {j, 1, Length[points]}];

MatrixForm[distMat]
```

$$\begin{pmatrix} 0 & \sqrt{5} & 4\sqrt{2} & 5\sqrt{2} & \sqrt{17} & \sqrt{10} \\ \sqrt{5} & 0 & \sqrt{13} & \sqrt{37} & 3\sqrt{2} & \sqrt{5} \\ 4\sqrt{2} & \sqrt{13} & 0 & 3\sqrt{2} & 5 & \sqrt{10} \\ 5\sqrt{2} & \sqrt{37} & 3\sqrt{2} & 0 & \sqrt{13} & 4 \\ \sqrt{17} & 3\sqrt{2} & 5 & \sqrt{13} & 0 & \sqrt{5} \\ \sqrt{10} & \sqrt{5} & \sqrt{10} & 4 & \sqrt{5} & 0 \end{pmatrix}$$

Radial Basis Functions

```
Map[rbf, distMat, {2}]
```

```
{{0, 4.02359, 55.4518, 97.8006, 24.0823, 11.5129}, {4.02359, 0, 16.6722, 66.802, 26.0133, 4.02359},
 {55.4518, 16.6722, 0, 26.0133, 40.2359, 11.5129}, {97.8006, 66.802, 26.0133, 0, 16.6722, 22.1807},
 {24.0823, 26.0133, 40.2359, 16.6722, 0, 4.02359}, {11.5129, 4.02359, 11.5129, 22.1807, 4.02359, 0}}
```

In[ ]:=
```
coeffs = LinearSolve[Map[rbf, distMat, {2}], points[[All, -1]]]
```

Out[ ]=
```
{0.00643967, 0.243995, -0.0258816, 0.0925154, 0.0472315, -0.49788}
```

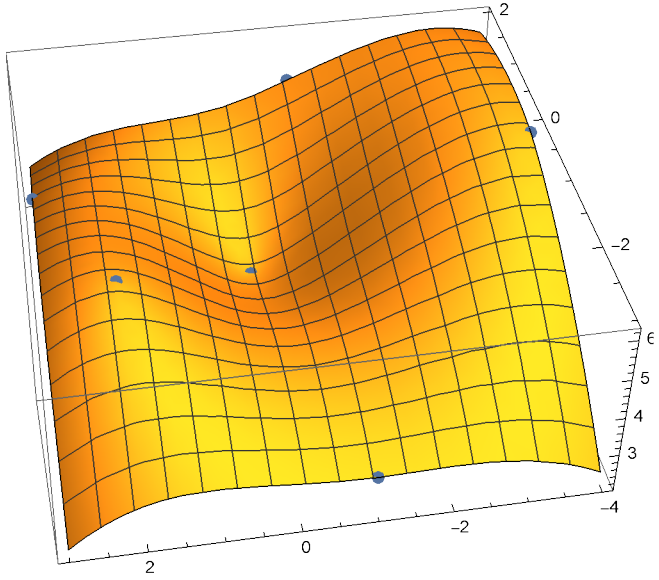
Radial Basis Functions

```
coeffs.rbf /@ distances[unknownPoint, points]
```

```
3.72068
```

In[ ]:=
```
Show[{
  Plot3D[coeffs.rbf /@ distances[{x, y}, points], {x, -3, 2}, {y, -4, 3}], ListPointPlot3D[points, PlotStyle → PointSize → Large]}]
```

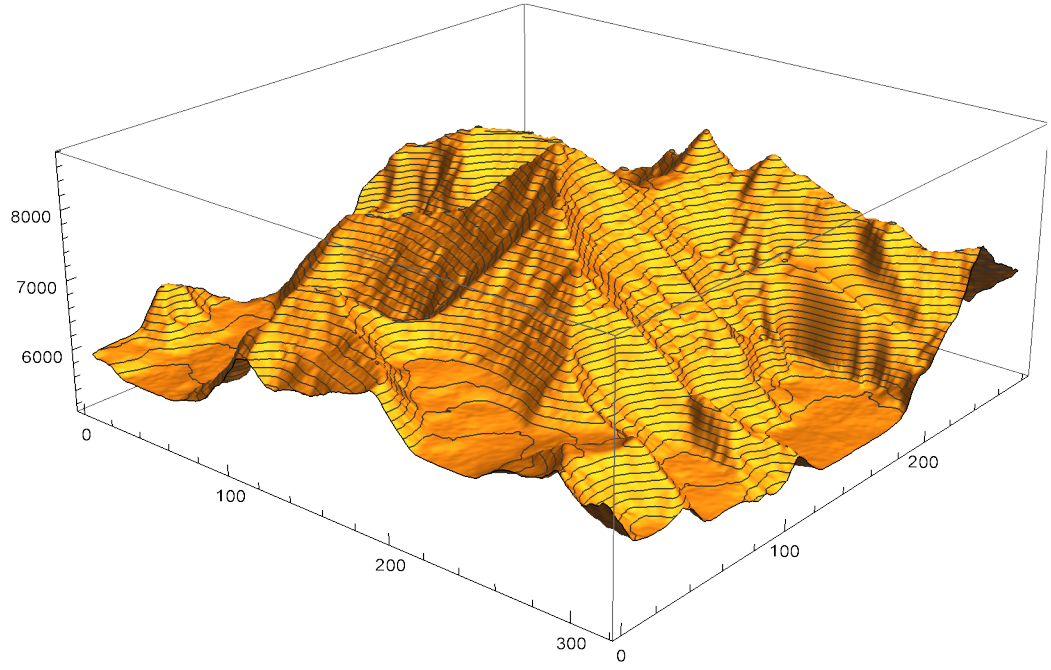
Out[ ]=



Radial Basis Functions

In[•]:=
```
data = GeoElevationData[GeoDisk[ Mount Everest MOUNTAIN , 3 mi]];
ListPlot3D[data, MeshFunctions → {#3 &}, Mesh → 40]
```

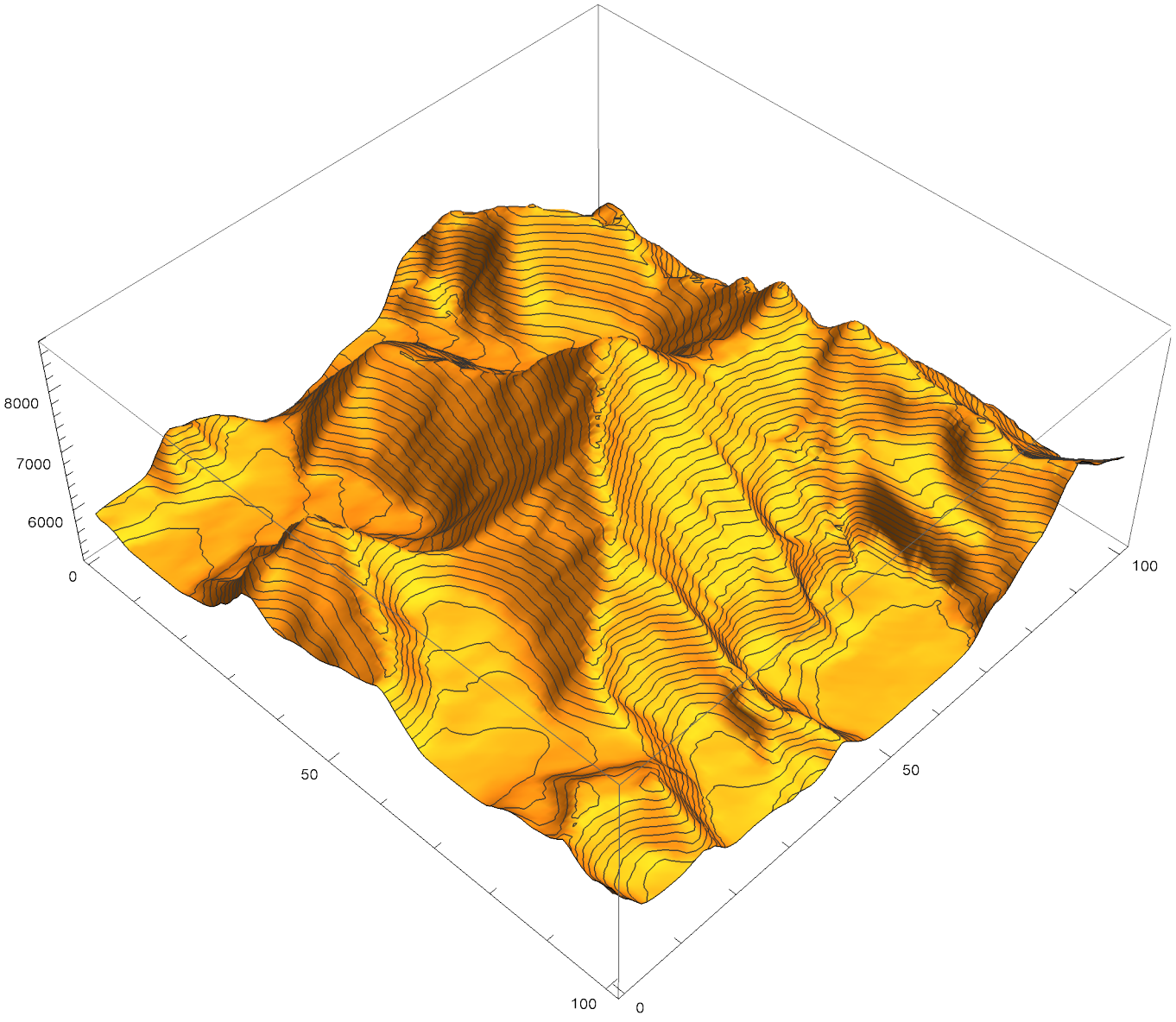
Out[•]=



Radial Basis Functions

In[•]:=
```
smallData = ArrayResample[QuantityMagnitude[data], {100, 100}];
ListPlot3D[smallData, MeshFunctions → {#3 &}, Mesh → 40]
```

Out[ ]=



Radial Basis Functions

```
In[•]:= SeedRandom[12345]; numPoints = 30;
        randomx = RandomInteger[{1, 100}, numPoints]
        randomy = RandomInteger[{1, 100}, numPoints]
```

```
Out[•]= {16, 67, 64, 91, 93, 47, 100, 36, 88, 20, 64, 76, 73, 50, 31, 77, 72, 40, 14, 98, 15, 40, 76, 92, 73, 61, 6, 31, 81, 94}
```

```
Out[•]= {82, 42, 75, 56, 94, 52, 56, 16, 53, 91, 74, 79, 49, 23, 80, 61, 84, 82, 23, 22, 20, 53, 28, 9, 4, 78, 43, 31, 13, 23}
```

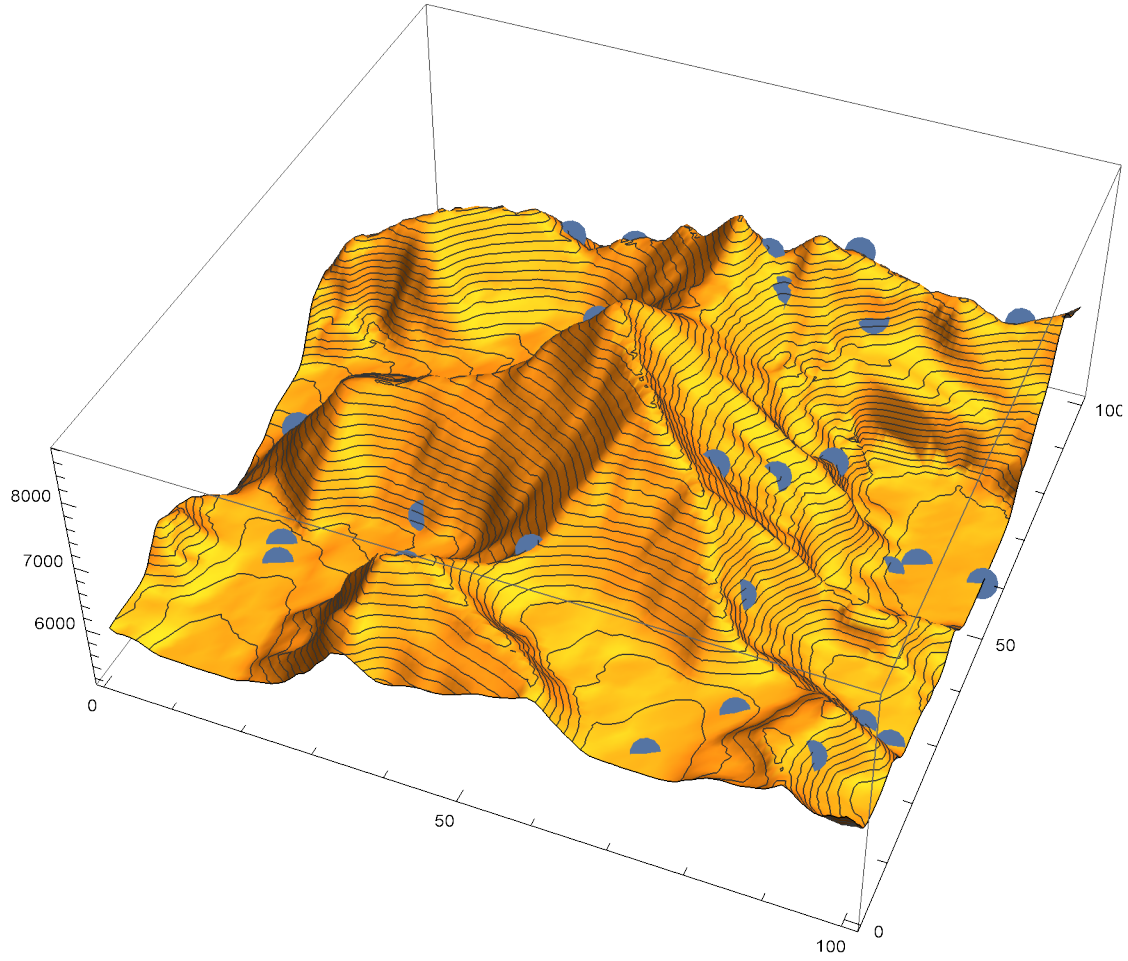
```
In[•]:= stuetzstellen =
         Table[{randomx[[i]], randomy[[i]], QuantityMagnitude[smallData[[randomy[[i]], randomx[[i]]]]]}, {i, 1, Length[randomx]}]
```

```
Out[•]= {{16, 82, 6878.54}, {67, 42, 7396.45}, {64, 75, 7614.07}, {91, 56, 5461.91}, {93, 94, 6623.29}, {47, 52, 8251.7},
         {100, 56, 5448.}, {36, 16, 6881.63}, {88, 53, 5480.75}, {20, 91, 5751.71}, {64, 74, 7564.5}, {76, 79, 7262.86},
         {73, 49, 6858.52}, {50, 23, 7032.7}, {31, 80, 7263.25}, {77, 61, 6278.99}, {72, 84, 7839.64}, {40, 82, 7210.63},
         {14, 23, 5906.3}, {98, 22, 5714.93}, {15, 20, 5880.4}, {40, 53, 7441.63}, {76, 28, 6797.01}, {92, 9, 6548.17},
         {73, 4, 6314.2}, {61, 78, 7923.76}, {6, 43, 6005.47}, {31, 31, 6295.69}, {81, 13, 6430.33}, {94, 23, 5798.97}}
```

Radial Basis Functions

In[•]:=
```
Show[{ListPlot3D[smallData, MeshFunctions → {#3 &}, Mesh → 40], ListPointPlot3D[stuetzstellen, PlotStyle → PointSize → 0.03]}]
```

Out[•]=



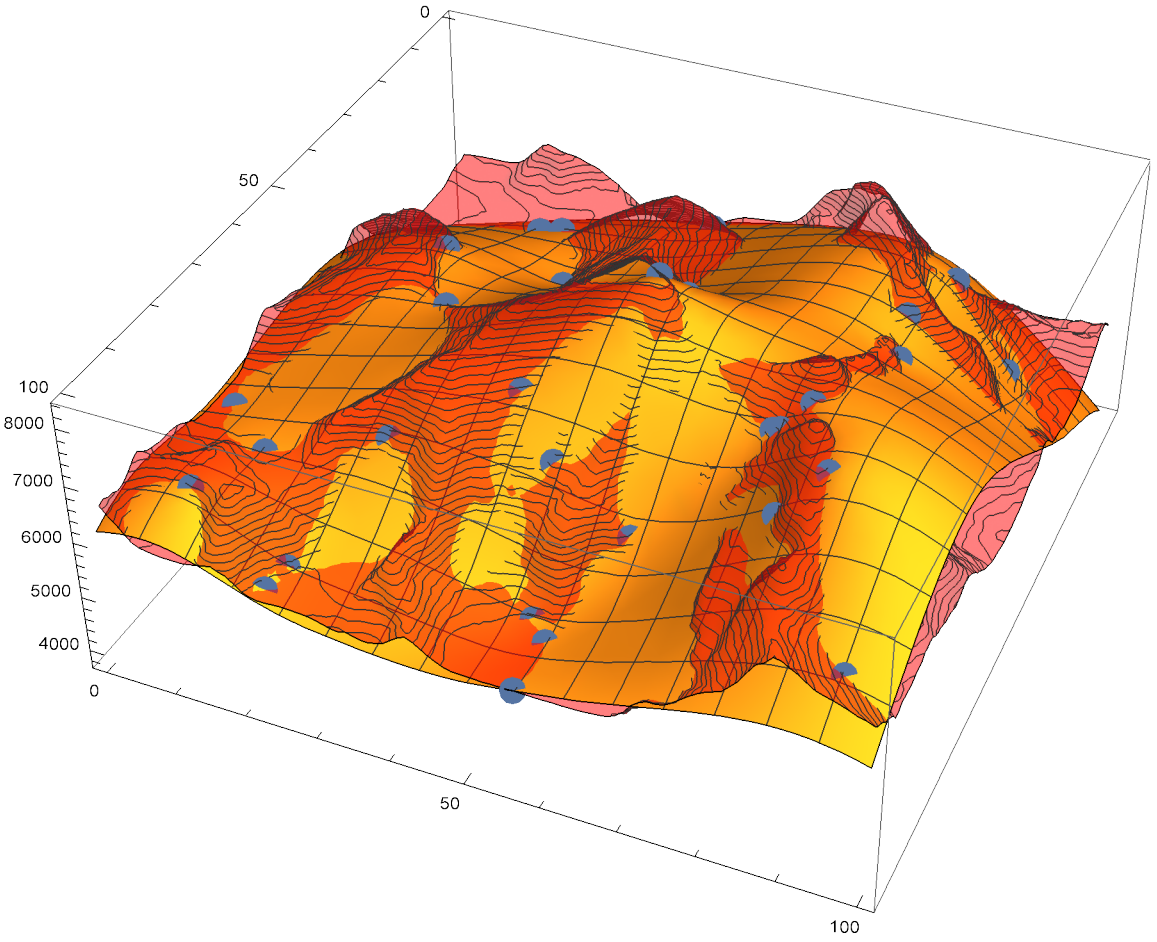
Radial Basis Functions

```
In[•]:= tabData = Flatten[Table[{i, j, QuantityMagnitude[smallData[[j, i]]]}, {i, 1, 100}, {j, 1, 100}], 1];

In[•]:= (*distMat=Table[EuclideanDistance[tabData[[i,{1,2}]],tabData[[j,{1,2}]]],{i,1,Length[tabData]},{j,1,Length[tabData]}];*)
        distMat = DistanceMatrix[stuetzstellen[[All, {1, 2}]]];

In[•]:= coeffs = LinearSolve[Map[rbf, distMat, {2}], stuetzstellen[[All, 3]]]

Out[•]= {2.74078, 0.00477582, -1.62365, 0.075806, 0.135378, 5.06804, 0.899392, 1.58204, -1.8815, -2.62986,
         -1.44493, -1.11841, 0.737765, -0.619094, 1.20662, -1.08216, 2.34799, -1.51531, 1.15964, -0.0369991,
         -1.02393, -4.02143, 0.945672, 2.2171, -0.7454, 2.70358, -0.0129552, -1.88567, -0.885455, -1.46586}
```

Radial Basis Functions

In[•]:=
```
Show[{
  Plot3D[coeffs.rbf /@ distances[{x, y}, stuetzstellen], {x, 0, 100}, {y, 0, 100}],
  ListPlot3D[smallData, MeshFunctions → {#3 &}, Mesh → 40, PlotStyle → Directive[Red, Opacity[0.5]]],
  ListPointPlot3D[stuetzstellen, PlotStyle → PointSize → 0.025]}]
```

Out[ ]=



Laplace Interpolation

Laplace interpolation is a specialized interpolation method for restoring missing data on a grid. It's simple, but sometimes works astonishingly well. It is based on the mean value theorem for solutions of Laplace's equation (harmonic functions):

If y satisfies $\nabla^2 y = 0$ in any number of dimensions, then for any sphere not intersecting a boundary condition

Out[ ]//TraditionalForm=

$$\frac{1}{\text{area}} \int_{\text{surface } \omega} y \, d\omega = y(\text{center})$$

So Laplace's equation is, in some sense, the perfect interpolator. It also turns out to be the one that minimizes the integrated square of the gradient,

Out[ ]//TraditionalForm=

$$\int_{\Omega} |\nabla y|^2 \, d\Omega$$

So the basic idea of Laplace interpolation is to set $y(x_i) = y_i$ at every known data point, and solve $\nabla^2 y = 0$ at every unknown point.

Laplace Interpolation

Consider a grid (array) with one missing pixel value:

y11	y12	y13	y14
y21	y22	y23	y24
y31	y32	?	y34
y41	y42	y43	y44

Using the Laplace approach, the value of y_{33} is determined by the average of its neighbors:

y11	y12	y13	y14
y21	y22	y23	y24
y31	y32	?	y34
y41	y42	y43	y44

Out[•]//TraditionalForm=

$$y_{33} = \frac{1}{4}(y_{23} + y_{43} + y_{32} + y_{34})$$

If the values of $y_{23}, y_{43}, y_{32},$ and y_{34} are known y_{33} can be computed.

Laplace Interpolation

Assume that two neighboring pixel are lost:

y11	y12	y13	y14
y21	y22	?	y24
y31	y32	?	y34
y41	y42	y43	y44

We have two unknowns y_{23} and y_{33} and two equations

Out[ ]//TraditionalForm=

$$y_{33} = \frac{1}{4}(y_{23} + y_{43} + y_{32} + y_{34})$$

Out[ ]//TraditionalForm=

$$y_{23} = \frac{1}{4}(y_{13} + y_{22} + y_{24} + y_{33})$$

Laplace Interpolation

Solving, gives:

Out[ ]//TraditionalForm=

$$y_{23} = \frac{1}{15}(4\,y_{13} + 4\,y_{22} + 4\,y_{24} + y_{32} + y_{34} + y_{43})$$

Out[ ]//TraditionalForm=

$$y_{33} = \frac{1}{15}(y_{13} + y_{22} + y_{24} + 4\,y_{32} + 4\,y_{34} + 4\,y_{43})$$

Laplace Interpolation

At the edges, on has to apply modified rules:

(left and right boundaries) $y_0 - \frac{1}{2} y_u - \frac{1}{2} y_d = 0$

(top and bottom boundaries) $y_0 - \frac{1}{2} y_l - \frac{1}{2} y_r = 0$

(top-left corner) $y_0 - \frac{1}{2} y_r - \frac{1}{2} y_d = 0$

(top-right corner) $y_0 - \frac{1}{2} y_l - \frac{1}{2} y_d = 0$

(bottom-left corner) $y_0 - \frac{1}{2} y_r - \frac{1}{2} y_u = 0$

(bottom-right corner) $y_0 - \frac{1}{2} y_l - \frac{1}{2} y_u = 0$

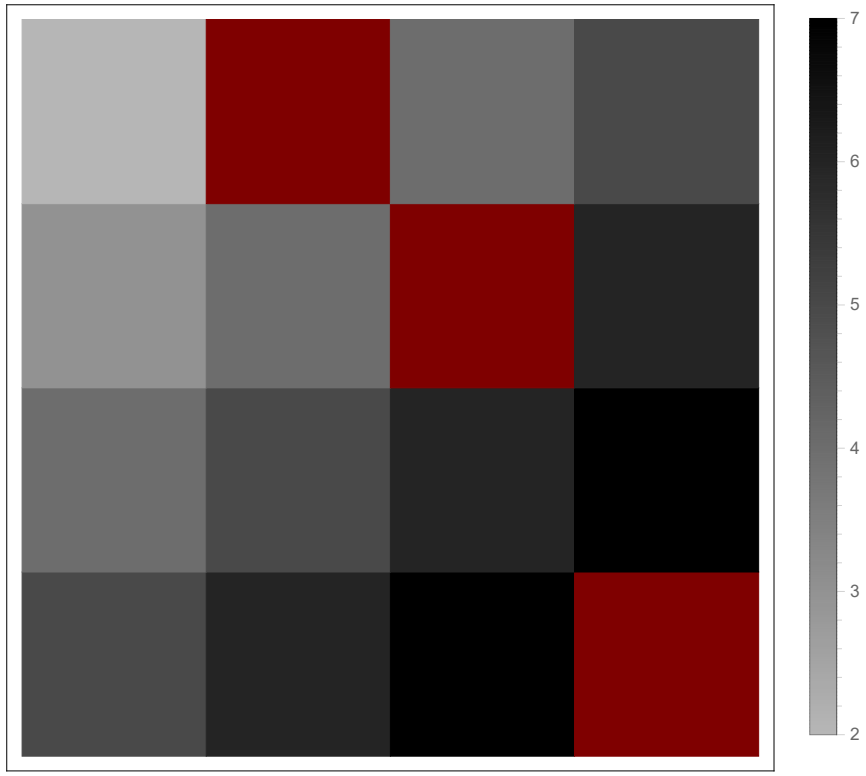
Here y_u, y_d, y_r, y_l denotes the y-values above, below, to the right and to the left of the respective position.

?	y12	y13	y14
y21	y22	y23	y24
y31	y32	y33	y34
y41	y42	y43	y44

y11	y12	y13	y14
y21	y22	y23	y24
?	y32	y33	y34
y41	y42	y43	y44

Example

$$\begin{pmatrix} 2 & \text{Missing}[\,] & 4 & 5 \\ 3 & 4 & \text{Missing}[\,] & 6 \\ 4 & 5 & 6 & 7 \\ 5 & 6 & 7 & \text{Missing}[\,] \end{pmatrix}$$



```
equations = {
    y12 - 1 / 2 × 4 - 1/2 × 2 == 0,
    y23 - 1/4 × 4 - 1/4 × 4 - 1/4 × 6 - 1/4 × 6 == 0,
    y44 - 1/2 × 7 - 1/2 × 7 == 0};
MatrixForm /@ Normal@CoefficientArrays[equations, {y12, y23, y44}]
```

$$\left\{\begin{pmatrix} -3 \\ -5 \\ -7 \end{pmatrix}, \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}\right\}$$

Example

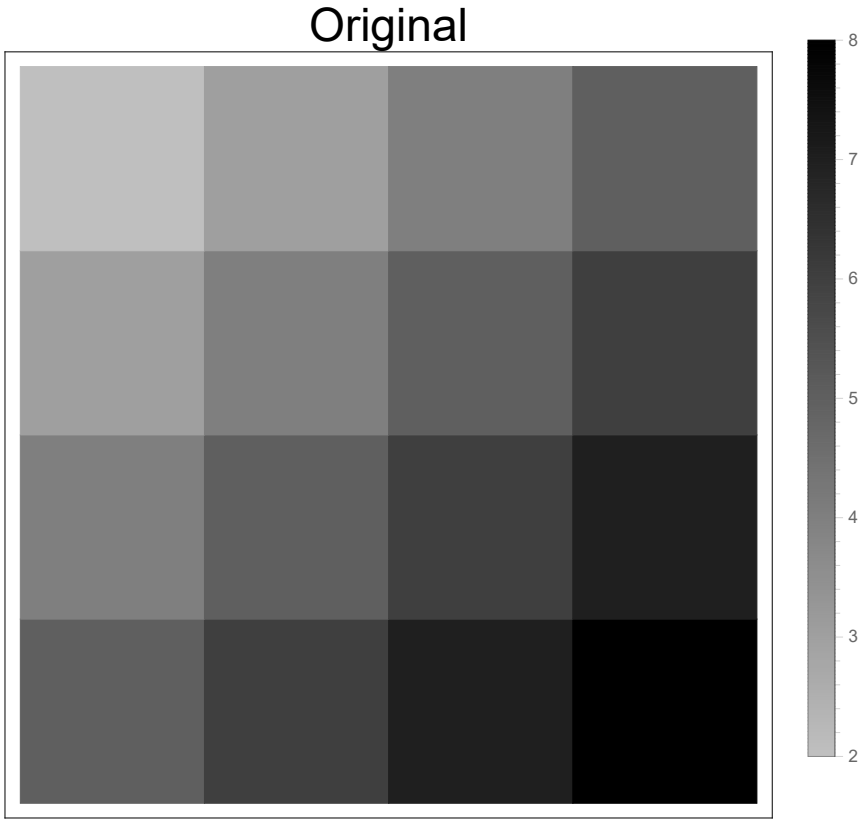
```
Abs@LinearSolve[{{1, 0, 0}, {0, 1, 0}, {0, 0, 1}}, {-3, -5, -7}]
```

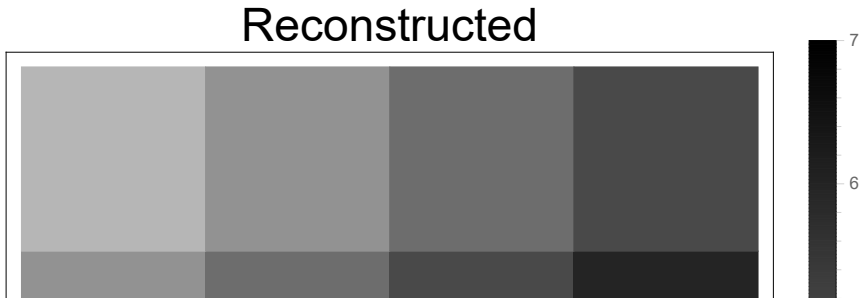
```
{3, 5, 7}
```

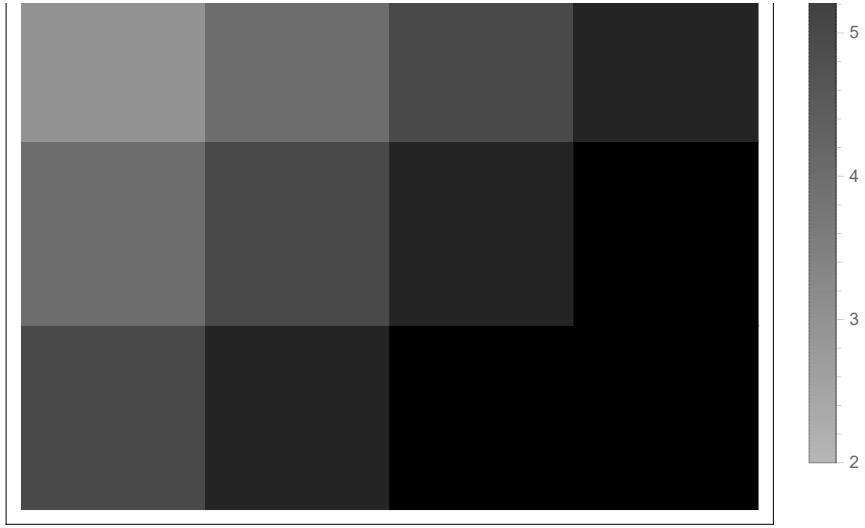
```
MatrixForm[orig]
```

$$\begin{pmatrix} 2 & 3 & 4 & 5 \\ 3 & 4 & 5 & 6 \\ 4 & 5 & 6 & 7 \\ 5 & 6 & 7 & 8 \end{pmatrix}$$

Example





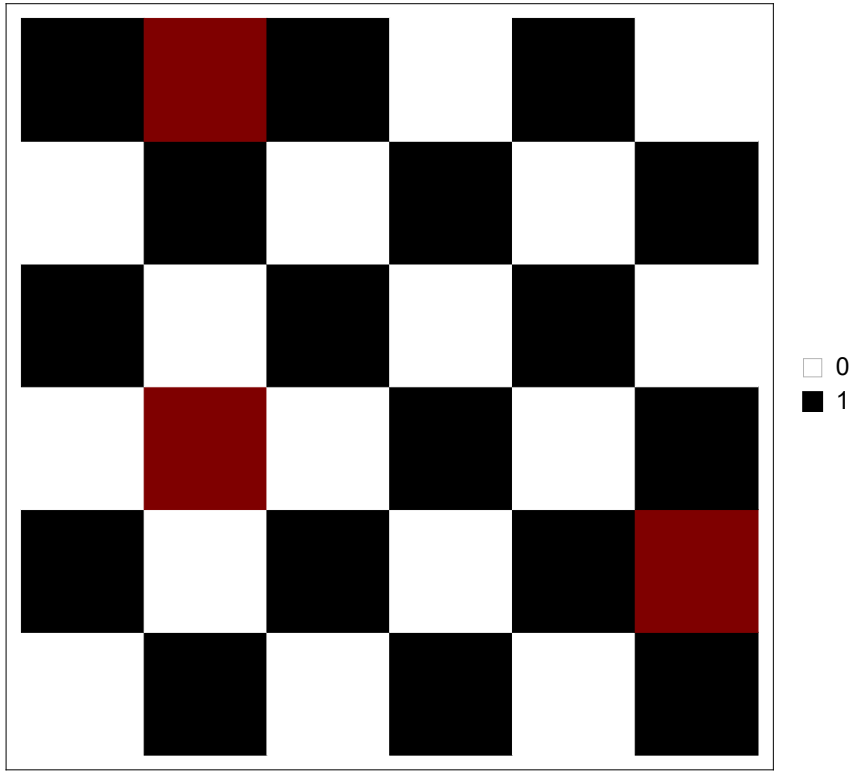

5
4
3
2

Example

This interpolation assumes that there is a relation between neighboring pixel. This is equivalent to saying that your pixel resolution is good enough to resolve structure in your array(image), i.e. your sampling needs to be sufficient small.

In the following example we violate this sampling condition. With an extreme case, a checkerboard pattern:

$$\begin{pmatrix} 1 & 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & \text{Missing}[] & 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 & 1 & 0 \\ 0 & \text{Missing}[] & 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 & 1 & \text{Missing}[] \\ 0 & 1 & 0 & 1 & 0 & 1 \end{pmatrix}$$



Example

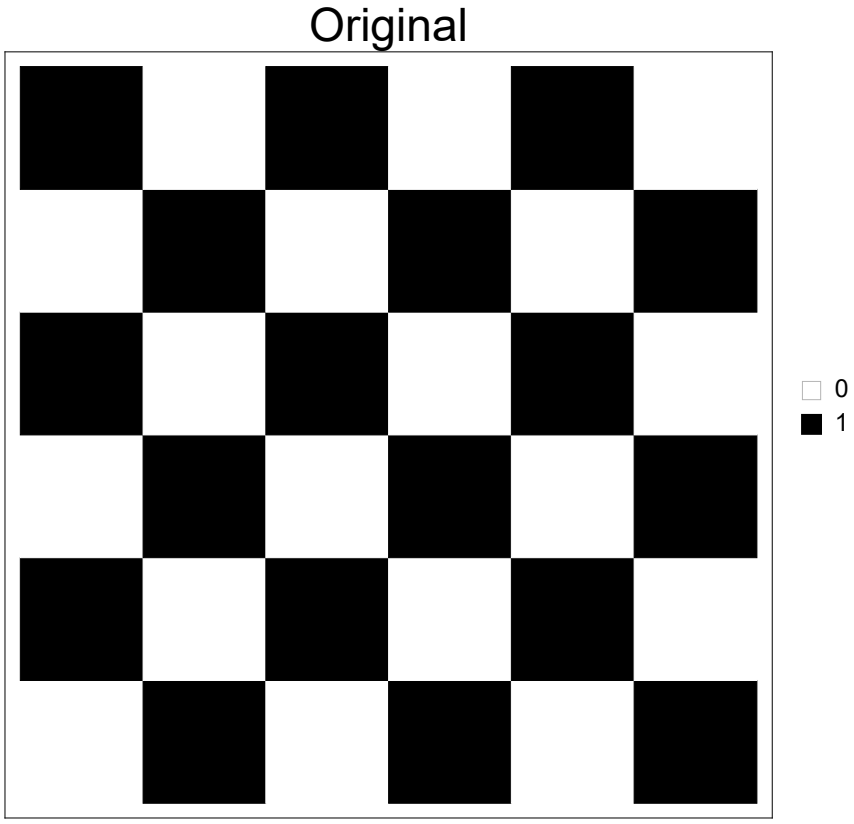
```
equations = {
    y12 - 1/2 × 1 - 1/2 × 1 == 0, y42 - 0/4 - 0/4 - 0/4 - 0/4 == 0, y56 - 1/2 - 1/2 == 0};
MatrixForm /@ Normal[CoefficientArrays[equations, {y12, y42, y56}]]
```

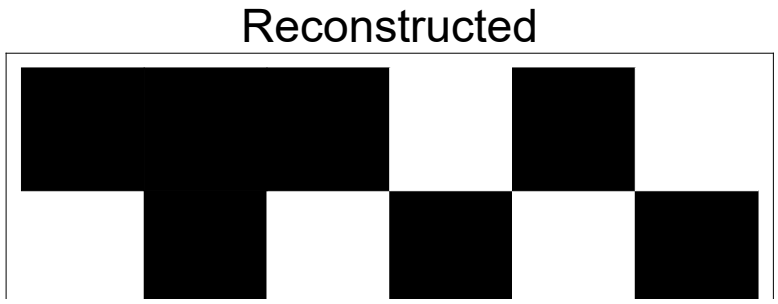
$$\left\{\begin{pmatrix} -1 \\ 0 \\ -1 \end{pmatrix}, \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}\right\}$$

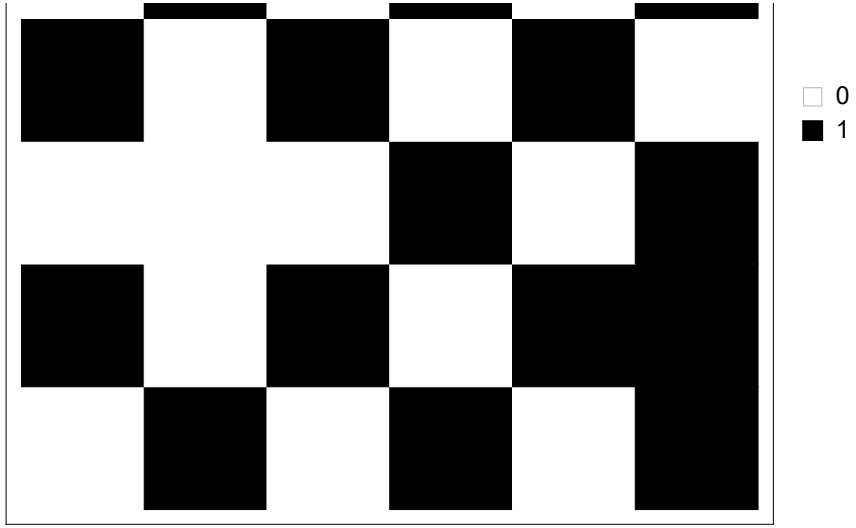
```
Abs@LinearSolve[{{1, 0, 0}, {0, 1, 0}, {0, 0, 1}}, {{-1}, {0}, {-1}}]
```

```
{1, 0, 1}
```

Example







Since neighbor fields are effectively anti-correlated the Laplace algorithm fails to recover the lost data.

Example

Out[ ]=

Original

Damaged (10%)

Repaired

Out[ ]=

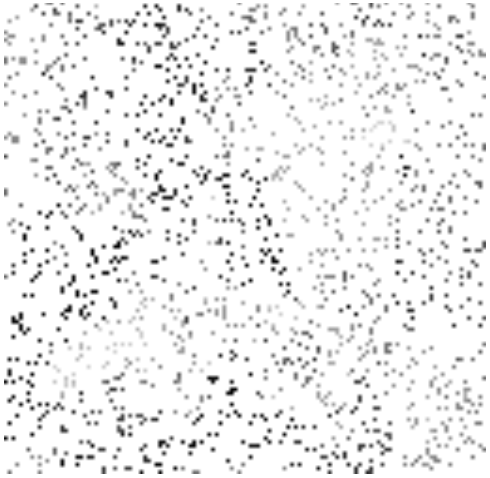
Original

Damaged (50%)

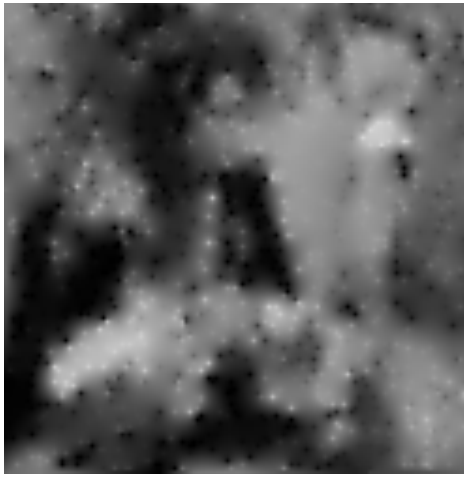
Repaired

Out[ ]=

Original



Damaged (90%)



Repaired

RBF Image reconstruction

The Laplace algorithm is surprisingly powerful in reconstructing a damaged image.

Assume you have a damaged image, i.e. some or many of the image pixel are lost. RBF interpolation can be used to recover the image.

In[ ]:=
```
imgOrig = ImageResize[ExampleData[{"TestImage", "Lena"}], {128}]
```

Out[ ]=

Be careful, code execution takes a lot of time (growing with image dimensions)!

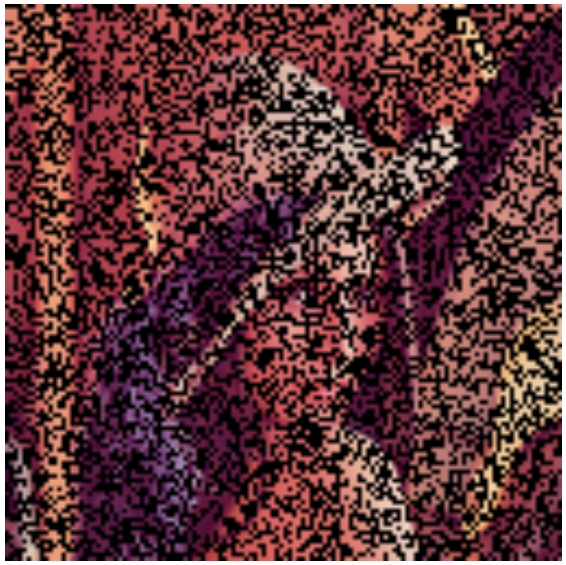
Example: RBF Image reconstruction

5% damaged pixel



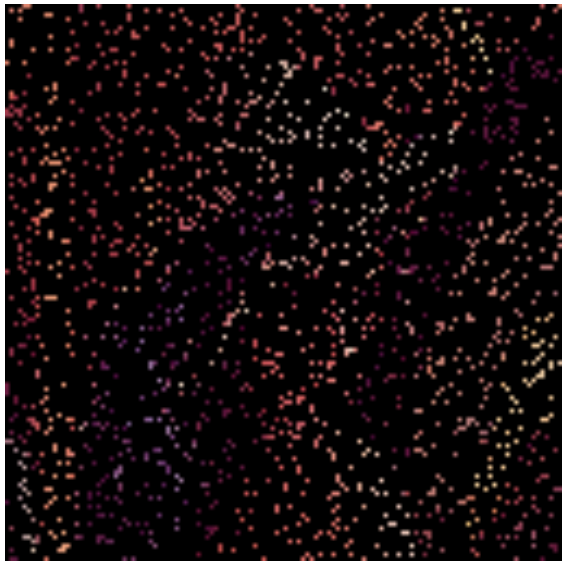
Example: RBF Image reconstruction

50% damaged pixel


Example: RBF Image reconstruction

90% damaged pixel


RBF Image reconstruction

RBF interpolation is extremely expensive in terms of computational operations.

Consider a NxM image with D damaged positions:

For each known pixel we get one equation with NxM terms

In total we the get a linear system of NxM-D equations

Solving a system with n equations takes $O(n^3)$ operations. To compute the RBF coefficient matrix We need to perform $O((\text{NxM} - D)^3)$ computer operations.
To interpolate a single positions requires another $O(\text{NxM} - D)$ steps

Example

512x512 RGB image with 10% damaged pixel:

per color channel:

$(512 * 512 * 0.9)^3 + 0.1 \times 512*512*512*0.9 = 1.3132496525491964*^16$

This is many computational operations. There are of course optimized linear solvers but the costs remain high.

RBF Image reconstruction

In[•]:=
```
im = ImageResize[ExampleData[{"TestImage", "Elaine"}], {256}];
dam = Image@damageImage[im, Round[0.7 × 256 * 256]]
```

Out[•]=

RBF Image reconstruction

```
Image[Inpaint[dam, Binarize[dam, 10^98], Method → #]] & /@ {"FastMarching", "NavierStokes"}
```

Smoothing

To smooth a data set is to create an approximating function that attempts to capture important patterns in the data, while leaving out noise or other fine-scale structures/rapid phenomena.

In smoothing, the data points of a signal are modified so individual points (presumably because of noise) are reduced, and points that are lower than the adjacent points are increased leading to a smoother signal.

Smoothing

Smoothing differs from curve fitting in the following ways:

- curve fitting often involves the use of an explicit function form for the result
- the aim of smoothing is to give a general idea of relatively slow changes of value with little attention paid to the close matching of data values
- smoothing methods often have an associated tuning parameter which is used to control the extent of smoothing

In terms of the frequency components of a signal, a smoothing operation acts as a low-pass filter, reducing the high-frequency components and passing the low-frequency components with little change.

Smoothing your data always means to loose some information!

Linear smoothing

If your smoothed values can be written as a linear transformation of the observed values, the smoothing operation is known as a linear smoother; the matrix representing the transformation is known as a smoother matrix.

Applying such a matrix transformation is called convolution (“Faltung”, “shift and multiply”). Thus the matrix is also called convolution matrix or a convolution kernel. In the case of simple series of data points (rather than a multi-dimensional image), the convolution kernel is a one-dimensional vector.

Smoothing Algorithms

Moving Average

The simplest smoothing algorithm is the ***rectangular boxcar*** or ***unweighted sliding-average*** smooth; it simply replaces each point in the signal with the average of m adjacent points, where m is a positive integer called the smooth width. For example, for a 3-point smooth ($m = 3$):

Out[•]=

$$S_j = \frac{1}{3}\,(Y_{j-1} + Y_j + Y_{j+1}) \qquad \texttt{for j=2 to n-1}$$

S_j is the j-th point in the smoothed signal, Y_j the j-th point in the original signal, n is the total number of points in the signal.

Similar smooth operations can be constructed for any desired smooth width, m. Usually m is an odd number.

Moving Average

The general expression for the centered smoothing (symmetrically around S_j is

Out[•]= $S_j = \frac{1}{m} \sum_{i=-\frac{m-1}{2}}^{\frac{m-1}{2}} Y_{j+i}$ for j = $\frac{m-1}{2}$ + 1 to n - $\frac{m-1}{2}$

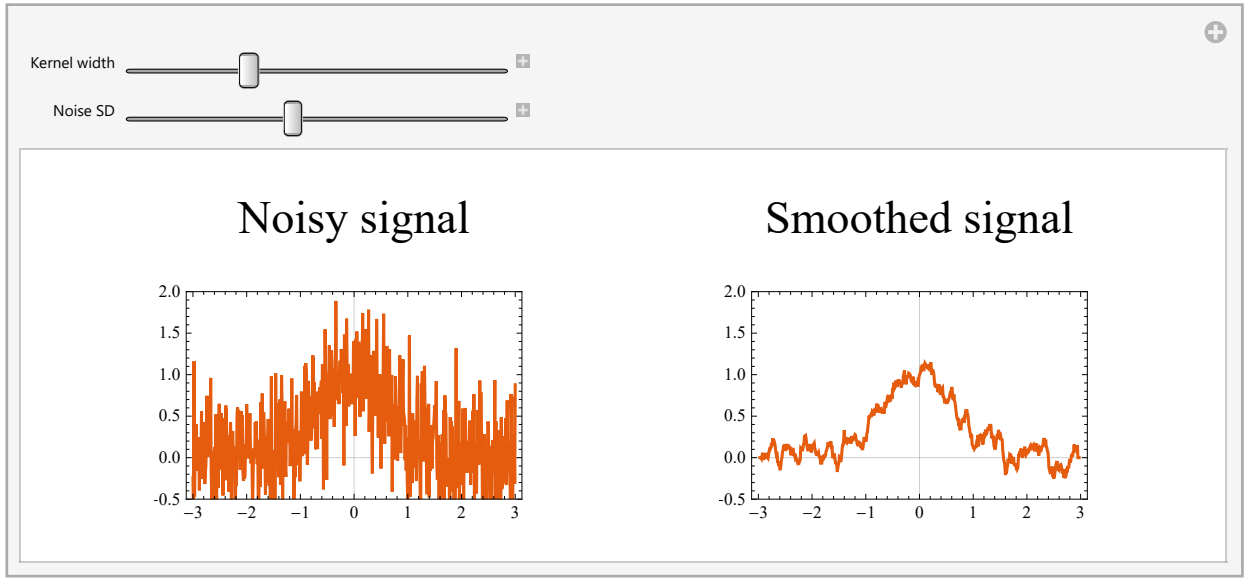
If the smoothing is not symmetrical around the j-th position, i.e.

Out[•]= $S_j = \frac{1}{m} \sum_{i=0}^{m-1} Y_{j+i}$ for j = 1 to n - (m - 1)

but this produces a relative shift between the input and output signals.

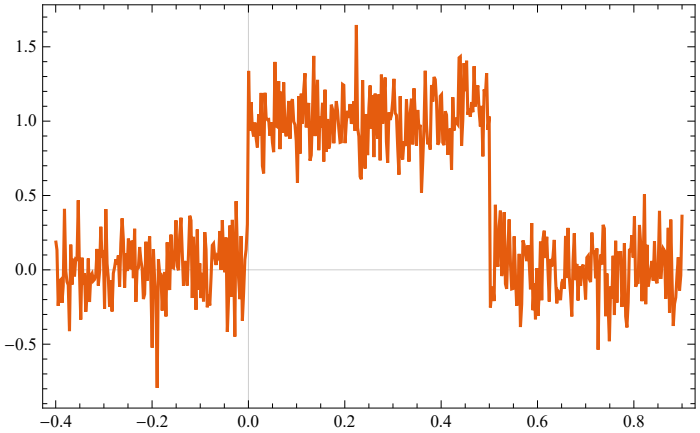
Moving Average

Out[ ]=

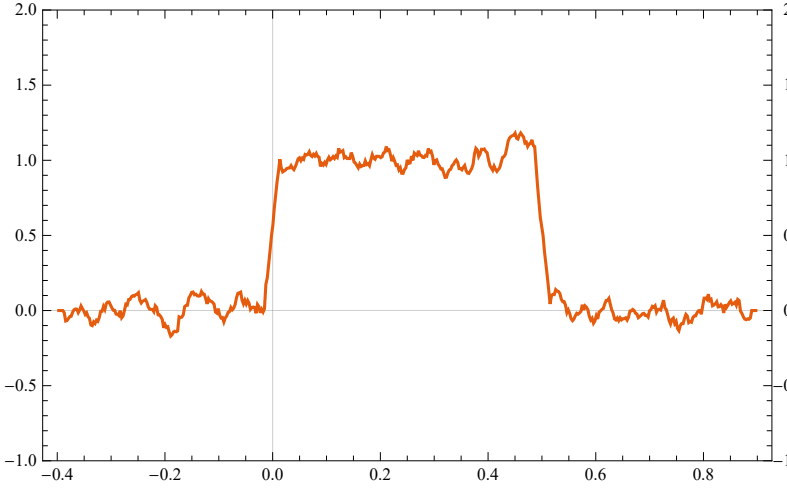


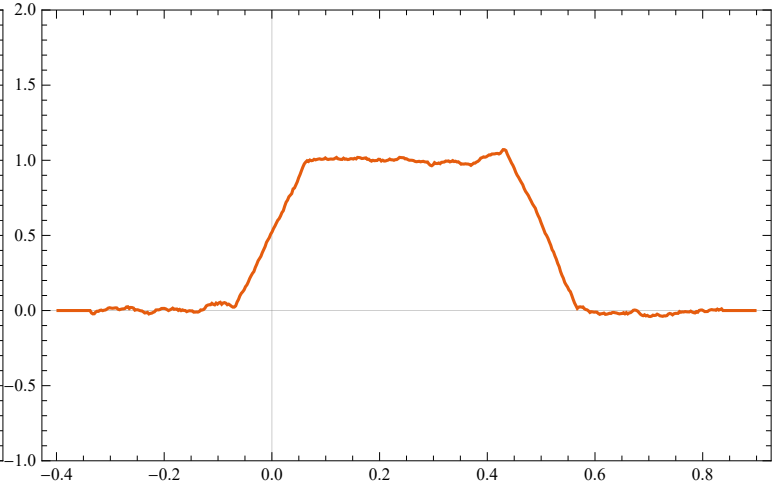
Noise Reduction vs. Step Response

Consider a pulse signal buried in noise:



Filtering this signal with 11 and 51 point moving average filters reduces the noise but the step edges become less sharp!





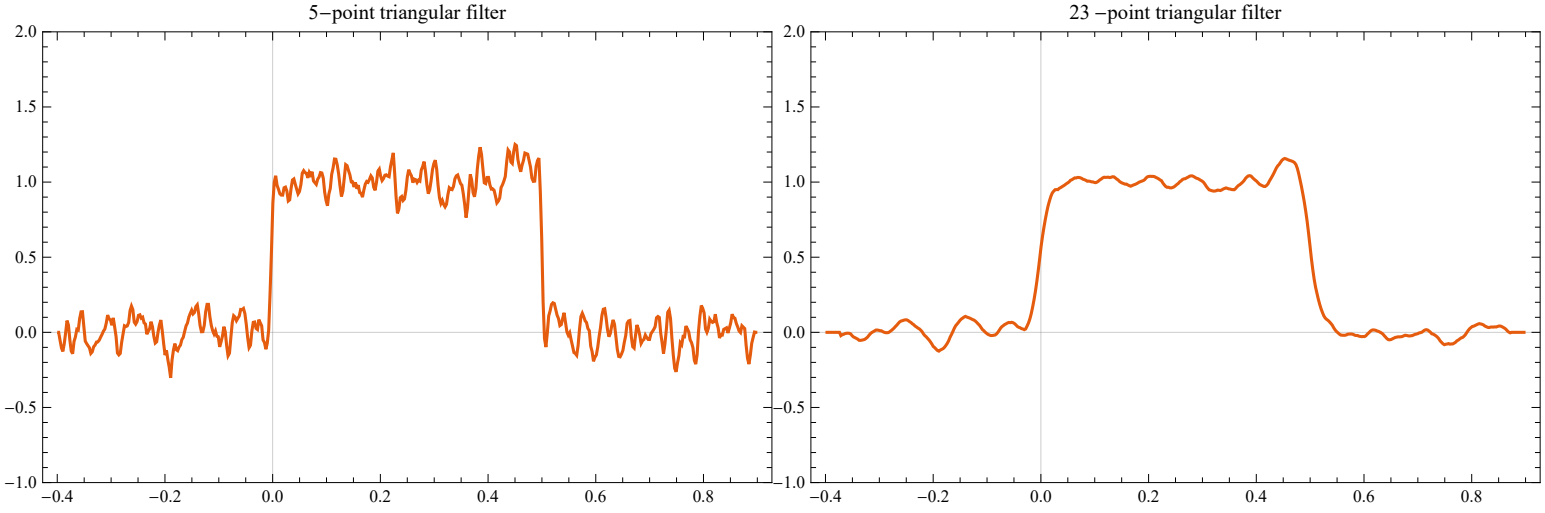
Triangular Smooth

The triangular smooth is like the rectangular smooth, above, except that it implements a weighted smoothing function. For a 5-point smooth (m = 5):

Out[•]=

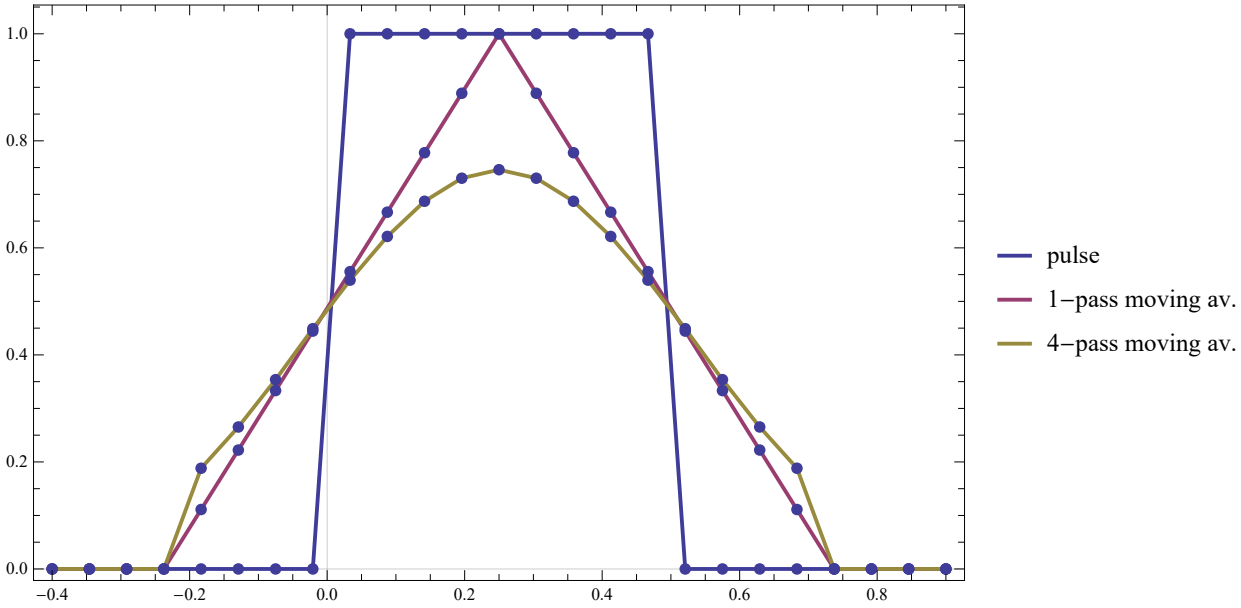
$$S_j = \frac{1}{9}\left(Y_{j-2} + 2\,Y_{j-1} + 3\,Y_j + 2\,Y_{j+1} + Y_{j+2}\right) \quad \texttt{for j=3 to n-2}$$

The the normalizing denominator for a m-point triangular smooth is $\frac{1}{4}(1+m)^2$.



Multi-Pass moving average filters

Imagine that we reapply the filter to the smoothed signal:



Passing a signal through a moving average filter twice is equivalent to using a triangular filter kernel. After four or more passes, the equivalent kernel looks like a Gaussian (central limit theorem).

Noise reduction

If the noise in the data is "white noise" (that is, evenly distributed over all frequencies) and its standard deviation is s, then the standard deviation of the noise remaining in the signal after the first pass of an unweighted sliding-average smooth will be

Out[•]//TraditionalForm=

$$\mathrm{SD}_S \approx \frac{\mathrm{SD}_Y}{\sqrt{m}}$$

Despite its simplicity, this smooth is actually optimum for the common problem of reducing white noise while keeping the sharpest step response. The response to a step change is in fact linear, so this filter has the advantage of responding completely with no residual effect withing its response time, which is equal to the smooth width divided by the sampling rate.

If a triangular smooth is used instead, the noise will be slightly less, about

Out[•]//TraditionalForm=

$$\mathrm{SD}_S \approx \frac{0.8\ \mathrm{SD}_Y}{\sqrt{m}}$$

End Effects and Lost Points

Out[ ]=

$$S_j = \frac{1}{3}\,(Y_{j-1} + Y_j + Y_{j+1}) \quad \text{for j=2 to n-1}$$

1	2	3	4	5	6
1	2	3	4	5	6
	2	3	4	5	

Note in the equations above that the 3-point rectangular smooth is defined only for j = 2 to n-1. There is not enough data in the signal to define a complete 3-point smooth for the first point in the signal (j = 1) or for the last point (j = n) , because there are no data points before the first point or after the last point.

End Effects and Lost Points

In general, for an m-width smooth, there will be (m-1)/2 points at the beginning of the signal and (m-1)/2 points at the end of the signal for which a complete m-width smooth can not be calculated.

- accept the loss of points and trim off those points or replace them with zeros in the smooth signal
- use progressively smaller smooths at the ends of the signal, for example to use 2, 3, 5, 7... point smooths for signal points 1, 2, 3,and 4..., and for points n, n-1, n-2, n-3..., respectively. The later approach may be preferable if the edges of the signal contain critical information, but it increases execution time.

Savitzky-Golay Filter

A Savitzky–Golay filter is a digital filter that can be applied to a set of digital data points for the purpose of smoothing the data. This is achieved by fitting successive sub-sets of adjacent data points with a low-degree polynomial by the method of linear least squares. When the data points are equally spaced, an analytical solution to the least-squares equations can be found, in the form of a single set of “convolution coefficients” that can be applied to all data sub-sets.

Savitzky, A.; Golay, M.J.E. (1964). "Smoothing and Differentiation of Data by Simplified Least Squares Procedures". Analytical Chemistry 36 (8): 1627–39. doi:10.1021/ac60214a047

Savitzky-Golay Filter

The data consists of a set of n $\{x_j, y_j\}$ points ($j = 1, \ldots, n$), where x is an independent variable and y_j is an observed value. They are treated with a set of m convolution coefficients, C_i, according to the expression

Out[•]=

$$Y_j = \sum_{i=-\frac{1}{2}(m-1)}^{\frac{m-1}{2}} C_i\, y_{j+i} \qquad \frac{m+1}{2} \le j \le n - \frac{m-1}{2}$$

For example, for smoothing by a 5-point quadratic polynomial, $m = 5$, $i = -2, -1, 0, 1, 2$ and the j –th smoothed data point, Y_j, is given by

Out[•]//TraditionalForm=

$$Y_j = \frac{1}{35}(-3\,y_{j-2} + 12\,y_{j-1} + 17\,y_j + 12\,y_{j+1} - 3\,y_{j+2})$$

where $C_{-2} = -3/35$, $C_{-1} = 12/35$, ...

Derivation of the Convolution Coefficients

When the data points are equally spaced, an analytical solution to the least-squares equations can be found. This solution forms the basis of the convolution method of numerical smoothing and differentiation. Suppose that the data consists of a set of n points (x_j, y_j) ($j = 1, \ldots, n$), where x is an independent variable and y_j is a datum value. A polynomial will be fitted by linear least squares to a set of m (an odd number) adjacent data points, each separated by an interval h. Firstly, a change of variable is made

Out[•]//TraditionalForm=

$$z = \frac{x - \overline{x}}{h}$$

where $\overline{x}$ is the value at the central point. z takes the values $\frac{1-m}{2}, \ldots, 0, \ldots, \frac{m-1}{2}$ (e.g. $m = 5 \Rightarrow z = -2, -1, 0, 1, 2$). The polynomial of degree k is defined as

Out[•]//TraditionalForm=

$$Y = a_0 + a_1 z + a_2 z^2 + \ldots + a_k z^k$$

Derivation of the Convolution Coefficients

The coefficients a_0, a_1 etc. are obtained by solving the normal equations

Out[ ]//TraditionalForm=

$$a = \left(J^{\mathsf{T}} J\right)^{-1} J^{\mathsf{T}} y$$

The i-th row of J has the values 1, z_i, z_i^2, ...

Derivation of the Convolution Coefficients

For example, for a cubic polynomial fitted to 5 points, $z = -2, -1, 0, 1, 2$ the normal equations are solved as follows.

Out[•]//TraditionalForm=

$$J = \begin{pmatrix} 1 & -2 & 4 & -8 \\ 1 & -1 & 1 & -1 \\ 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 \\ 1 & 2 & 4 & 8 \end{pmatrix}$$

Out[•]//TraditionalForm=

$$J^{\mathsf{T}} J = \begin{pmatrix} 1 & 1 & 1 & 1 & 1 \\ -2 & -1 & 0 & 1 & 2 \\ 4 & 1 & 0 & 1 & 4 \\ -8 & -1 & 0 & 1 & 8 \end{pmatrix} \begin{pmatrix} 1 & -2 & 4 & -8 \\ 1 & -1 & 1 & -1 \\ 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 \\ 1 & 2 & 4 & 8 \end{pmatrix} = \begin{pmatrix} 1+1+1+1+1 & -2-1+0+1+2 & 4+1+0+1+4 & -8-1+0+1+8 \\ -2-1+0+1+2 & 4+1+0+1+4 & -8-1+0+1+8 & 16+1+0+1+16 \\ 4+1+0+1+4 & -8-1+0+1+8 & 16+1+0+1+16 & -32-1+0+1+32 \\ -8-1+0+1+8 & 16+1+0+1+16 & -32-1+0+1+32 & 64+1+0+1+64 \end{pmatrix}$$

Out[•]=

$$J^{\mathsf{T}} J = \begin{pmatrix} m & \sum z & \sum z^2 & \sum z^3 \\ \sum z & \sum z^2 & \sum z^3 & \sum z^4 \\ \sum z^2 & \sum z^3 & \sum z^4 & \sum z^5 \\ \sum z^3 & \sum z^4 & \sum z^5 & \sum z^6 \end{pmatrix} = \begin{pmatrix} 5 & 0 & 10 & 0 \\ 0 & 10 & 0 & 34 \\ 10 & 0 & 34 & 0 \\ 0 & 34 & 0 & 130 \end{pmatrix}$$

$$\left(J^{\mathsf{T}} J\right)^{-1} = \begin{pmatrix} \frac{17}{35} & 0 & -\frac{1}{7} & 0 \\ 0 & \frac{65}{72} & 0 & -\frac{17}{72} \\ -\frac{1}{7} & 0 & \frac{1}{14} & 0 \\ 0 & -\frac{17}{72} & 0 & \frac{5}{72} \end{pmatrix}$$

Derivation of the Convolution Coefficients

$$(J^T J)^{-1} J^T Y = \begin{pmatrix} \frac{17}{35} & 0 & -\frac{1}{7} & 0 \\ 0 & \frac{65}{72} & 0 & -\frac{17}{72} \\ -\frac{1}{7} & 0 & \frac{1}{14} & 0 \\ 0 & -\frac{17}{72} & 0 & \frac{5}{72} \end{pmatrix} \begin{pmatrix} 1 & 1 & 1 & 1 & 1 \\ -2 & -1 & 0 & 1 & 2 \\ 4 & 1 & 0 & 1 & 4 \\ -8 & -1 & 0 & 1 & 8 \end{pmatrix} \begin{pmatrix} y_{j-2} \\ y_{j-1} \\ y_j \\ y_{j+1} \\ y_{j+2} \end{pmatrix} =$$

$$\begin{pmatrix} \frac{17}{35} - \frac{4}{7} & \frac{17}{35} - \frac{1}{7} & \frac{17}{35} & \frac{17}{35} - \frac{1}{7} & \frac{17}{35} - \frac{4}{7} \\ -\frac{2\times 65}{72} + \frac{8\times 17}{72} & -\frac{65}{72} + \frac{17}{72} & 0 & \frac{65}{72} - \frac{17}{72} & \frac{65\times 2}{72} - \frac{8\times 17}{72} \\ \cdots & \cdots & \cdots & \cdots & \cdots \\ \cdots & \cdots & \cdots & \cdots & \cdots \end{pmatrix} = \begin{pmatrix} -\frac{3}{35} & \frac{12}{35} & \frac{17}{35} & \frac{12}{35} & -\frac{3}{35} \\ \frac{1}{12} & -\frac{2}{3} & 0 & \frac{2}{3} & -\frac{1}{12} \\ \frac{1}{7} & -\frac{1}{14} & -\frac{1}{7} & -\frac{1}{14} & \frac{1}{7} \\ -\frac{1}{12} & \frac{1}{6} & 0 & -\frac{1}{6} & \frac{1}{12} \end{pmatrix}$$

The rows of the

Out[•]//TraditionalForm=

$$a_{0,j} = \frac{1}{35}(-3\,y_{j-2} + 12\,y_{j-1} + 17\,y_j + 12\,y_{j+1} - 3\,y_{j-2})$$

Out[•]//TraditionalForm= $$a_{1,j} = \frac{1}{12}(y_{j-2} - 8\,y_{j-1} + 8\,y_{j+1} - y_{j-2})$$

Out[•]//TraditionalForm= $$a_{2,j} = \frac{1}{14}(2\,y_{j-2} - y_{j-1} - 2\,y_j - y_{j+1} + 2\,y_{j-2})$$

Out[•]//TraditionalForm= $$a_{3,j} = \frac{1}{12}(-y_{j-2} + 2\,y_{j-1} - 2\,y_{j+1} + y_{j-2})$$

Derivation of the Convolution Coefficients

The coefficients of y in these expressions are known as convolution coefficients. They are elements of the matrix

Out[ ]//TraditionalForm=

$$C = \left(J^{\mathsf{T}} J\right)^{-1} J^{\mathsf{T}}$$

Derivation of the Convolution Coefficients

In general,

Out[•]= $$\left(C \otimes y\right)_j = Y_j = \sum_{i=-\frac{1}{2}(m-1)}^{\frac{m-1}{2}} C_i \, y_{i+j} \qquad \frac{m+1}{2} \le j \le n - \frac{m-1}{2}$$

In matrix notation this example is written as

Out[•]//TraditionalForm=

$$\begin{pmatrix} Y_3 \\ Y_4 \\ Y_5 \\ \dots \end{pmatrix} = \frac{1}{35} \begin{pmatrix} -3 & 12 & 17 & 12 & -3 & 0 & 0 & \dots \\ 0 & -3 & 12 & 17 & 12 & -3 & 0 & \dots \\ 0 & 0 & -3 & 12 & 17 & 12 & -3 & \dots \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \ddots \end{pmatrix} \begin{pmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \\ y_6 \\ y_7 \\ \vdots \end{pmatrix}$$

S-G kernel as smooth derivator

Consider linear data: $y = \{1, 2, 3, 4, 5\}$ applying the SG kernel $\{-\frac{2}{10}, \frac{-1}{10}, 0, \frac{1}{10}, \frac{2}{10}\}$:

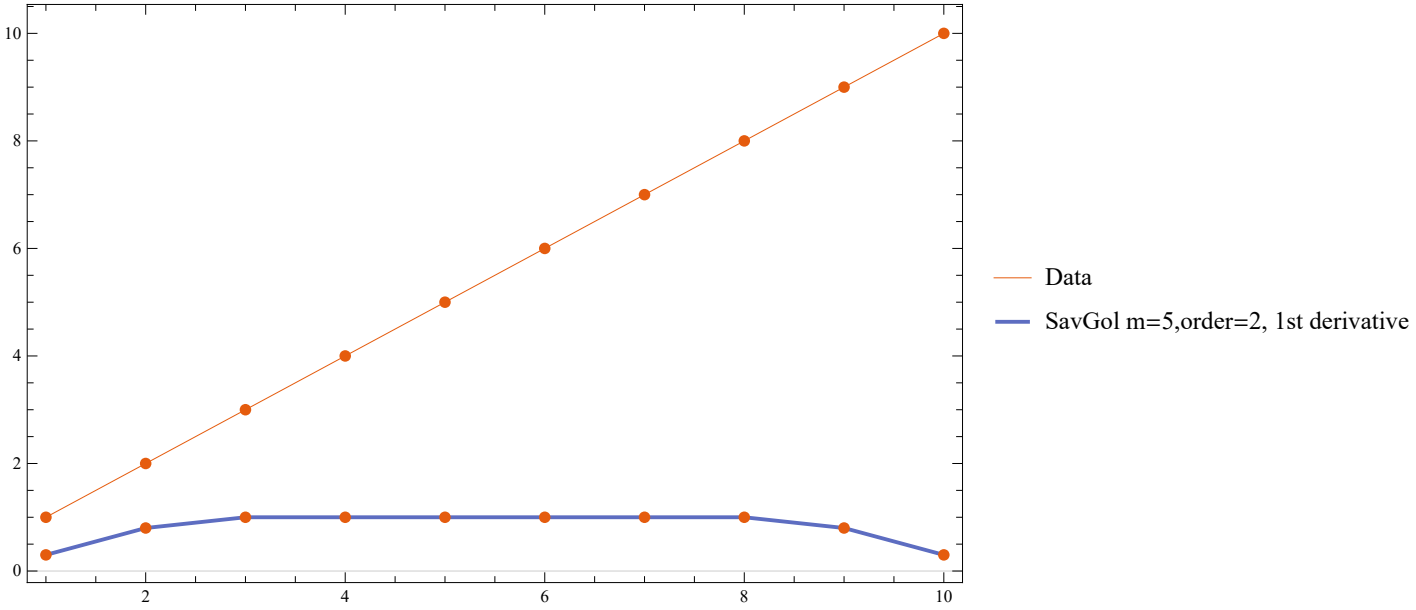
Out[ ]//TraditionalForm=

$$\begin{pmatrix} -\frac{2}{10} \\ -\frac{1}{10} \\ 0 \\ \frac{1}{10} \\ \frac{2}{10} \end{pmatrix} . \begin{pmatrix} 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{pmatrix} = -\frac{1}{5} - \frac{2}{10} + 0 + \frac{4}{10} + \frac{5}{5} = -0.2 - 0.2 + 0 + 0.4 + 1 = 1$$

This is exactly the slope of the linear data! Looking at the kernel we note, that points on the right side of the center points are positively weighted while the left half of the kernel provides negative weights. Accordingly, if the data has a positive slope, i.e. y-values are increasing, then the convolution gives a positive value - the first derivative of the (smoothed) data.

S-G kernel as smooth derivator

Looking at the rows of the S-G coefficient matrix we not that the a_{1_j} is such a kernel. The first row of the coefficient matrix gives the 0-th derivative, i.e. a smooth. The second row gives the 1st derivative (kernel), the third row gives the 2nd derivative, and so on.



Note the edge effects. In the following realization I chose to extend the dataset to the left and right: $y_3, y_2, y_1, \ldots, y_n, y_{n-1}, y_{n-2}$.

Alternatively one could use non-symmetric S-G kernel.

Computing the kernel coefficients

```
In[•]:= Clear[SGKernel];
SGKernel[nl_, nr_, deriv_, order_] := Module[{a, m = order, sum, mm, b, np = nr + nl + 1, fac, c, kk, e, ata},
  a = Quiet@Table[i^j /. Indeterminate :> 1, {i, -nl, nr, 1}, {j, 0, m}];
  ata = Inverse[Transpose[a].a];
  deriv ! N[Total /@ Quiet @Table[ata[[deriv + 1]] × Table[n^j /. Indeterminate :> 1, {j, 0, m}], {n, -nl, nr}]]
 ]
Clear[SGFilter]
SGFilter[ydata_, {nl_, nr_}, order_, deriv_Integer : 0] := Module[{dat, left, right},
  left = Reverse[ydata[[1 ;; nl]]];
  right = Reverse[ydata[[-nr ;; -1]]];
  dat = Join[left, ydata, right];
  ListCorrelate[SGKernel[nl, nr, deriv, order], dat]]
```

Compare with the native Mathematica function:

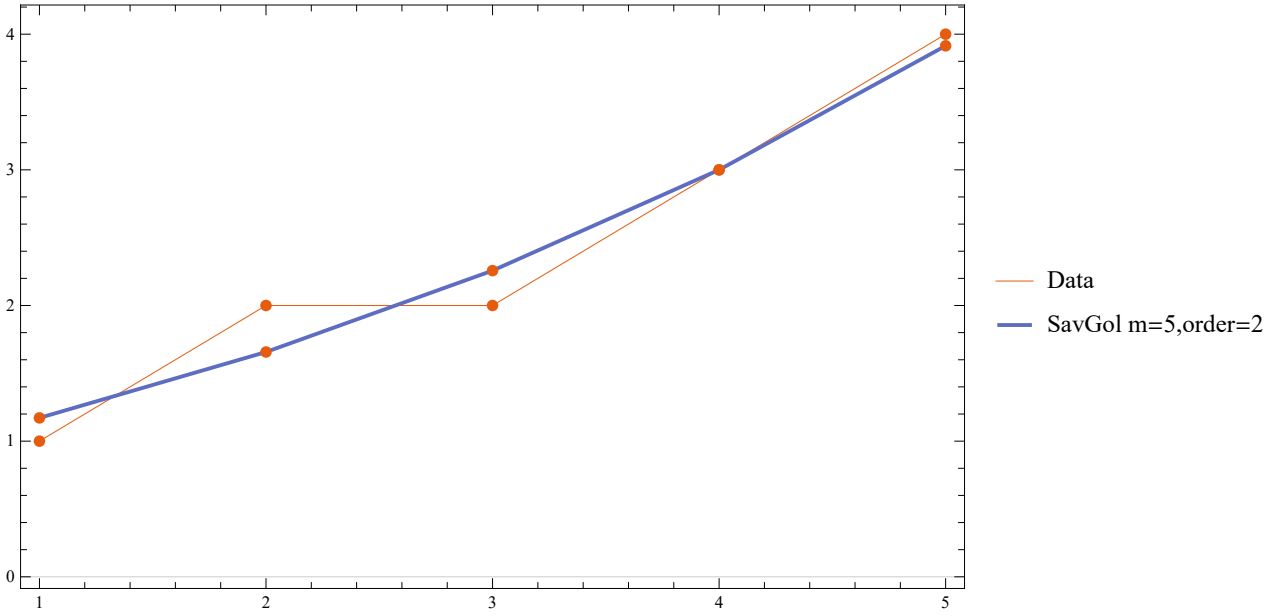
```
In[•]:= SGKernel[2, 2, 0, 3]
SavitzkyGolayMatrix[{2}, 3, 0]
```

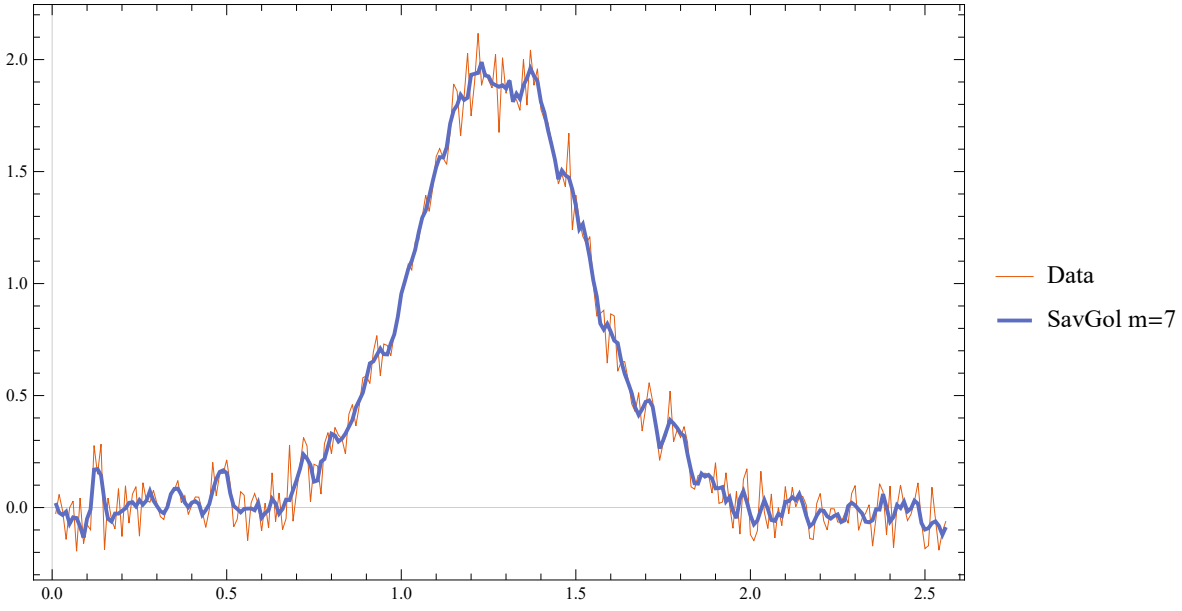
```
Out[•]= {-0.0857143, 0.342857, 0.485714, 0.342857, -0.0857143}
```

```
Out[•]= {-0.0857143, 0.342857, 0.485714, 0.342857, -0.0857143}
```

S-G kernel as smooth derivator

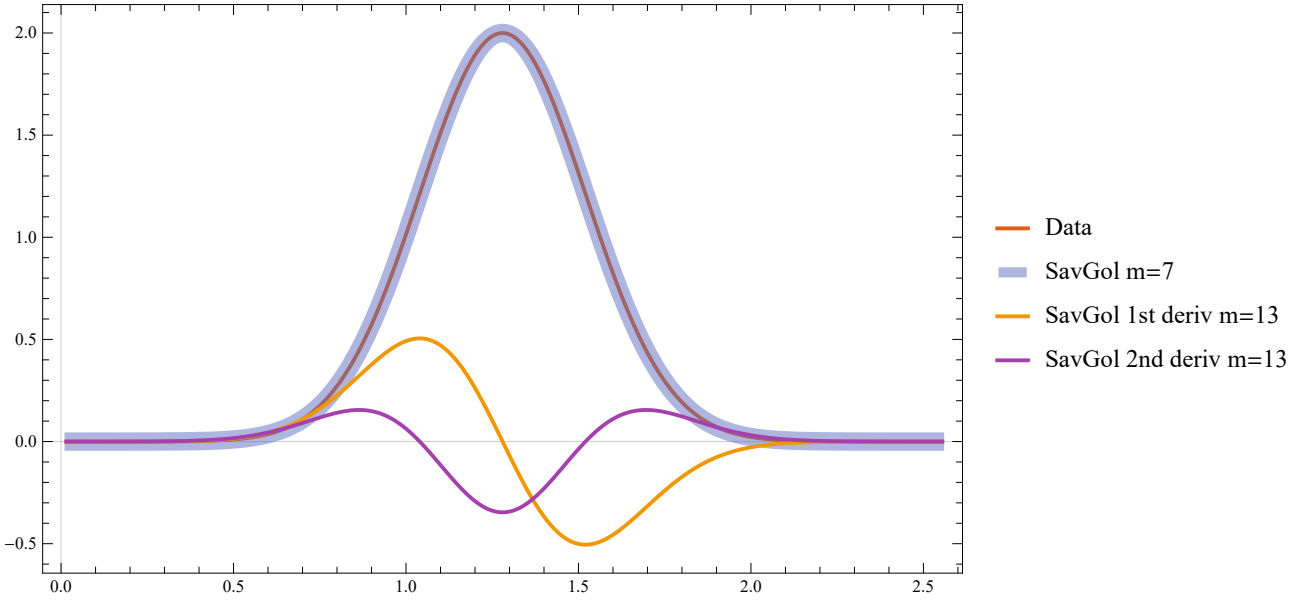
The following shows a 5-point S-G smooth of order 2. Each point is approximated with a quadratic function of 5 points width.




2.0
1.5
1.0
0.5
0.0
0.0
0.5
1.0
1.5
2.0
2.5
Data
SavGol m=7

S-G kernel as smooth derivator

The S-G kernel reproduces the un-noisy data perfectly. The S-G derivatives



Computing the kernel coefficients

5-point (2,2) kernel

		z				
		−2	−1	0	1	2
order	0	0.2	0.2	0.2	0.2	0.2
	1	−0.0857143	0.342857	0.485714	0.342857	−0.0857143
	2	−0.0857143	0.342857	0.485714	0.342857	−0.0857143
	3	0.	0.	1.	0.	0.

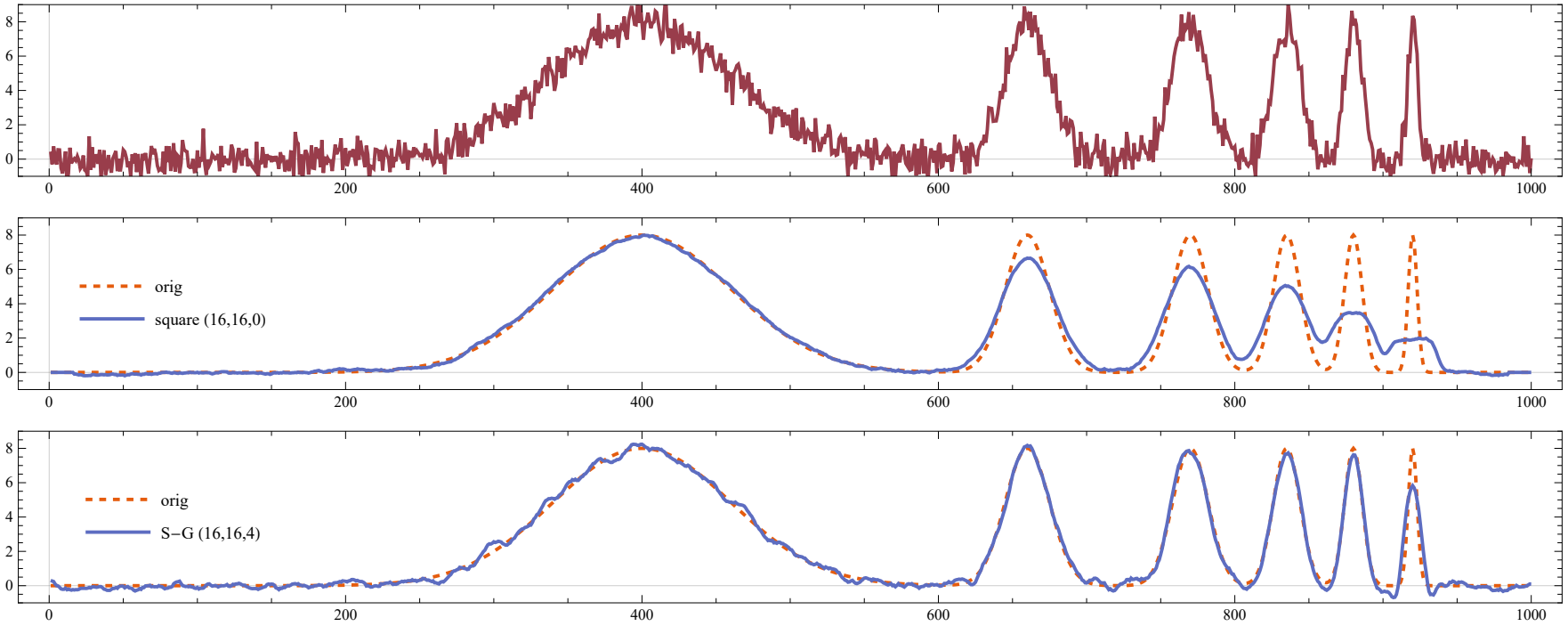
7-point (3,3) kernel

		z						
		−3	−2	−1	0	1	2	3
order	1/2	−0.0952381	0.142857	0.285714	0.333333	0.285714	0.142857	−0.0952381
	3/4	0.021645	−0.12987	0.324675	0.5671	0.324675	−0.12987	0.021645

9-point (4,4) kernel

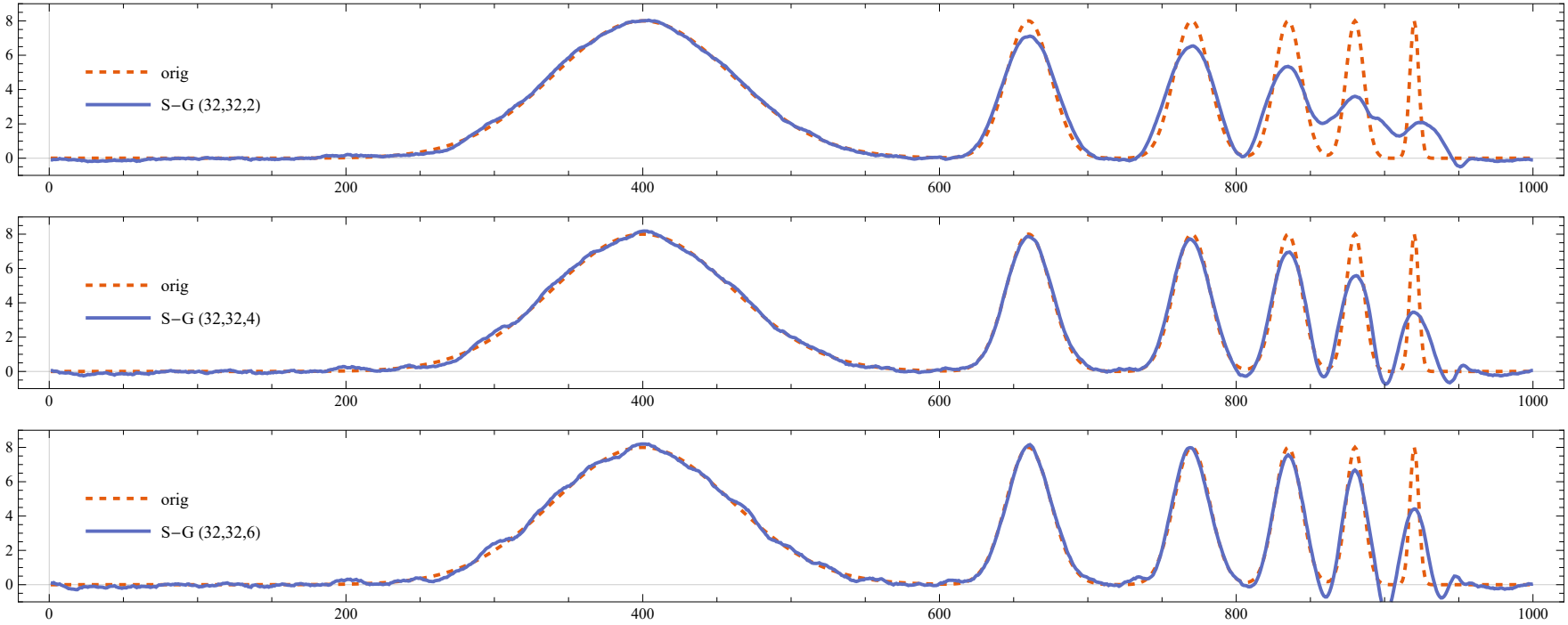
		z						
		−4	−3	−2	−1	0	1	2
order	1/2	−0.0909091	0.0606061	0.168831	0.233766	0.255411	0.233766	0.168831
	3/4	0.034965	−0.128205	0.0699301	0.314685	0.417249	0.314685	0.0699301
	5/6	−0.00543901	0.043512	−0.152292	0.304584	0.61927	0.304584	−0.152292

Peak preservation



Top: Synthetic noisy data consisting of a sequence of progressively narrower bumps, and additive Gaussian white noise. **Center:** Result of smoothing the data by a simple moving window average. The window extends 16 points leftward and rightward, for a total of 33 points. Note that narrow features are broadened and suffer corresponding loss of amplitude. The dotted curve is the underlying function used to generate the synthetic data. **Bottom:** Result of smoothing the data by a Savitzky-Golay smoothing filter (of degree 4) using the same 33 points. While there is less smoothing of the broadest feature, narrower features have their heights and widths preserved.

Influence of polynomial degree



Result of a 65 point S-G filter applied to the same data as before with degree 2 (top), 4 (middle), and 6 (bottom). All of these filters are in-optimally broad for the resolution of the narrow features. Higher-order filters do best at preserving feature heights and widths, but do less smoothing on broader features.

Some properties of convolution

- The sum of convolution coefficients for smoothing is equal to one. The sum of coefficients for odd derivatives is zero.
- The sum of squared convolution coefficients for smoothing is equal to the value of the central coefficient.
- Smoothing of a function leaves the area under the function unchanged.
- Convolution of a symmetric function with even-derivative coefficients conserves the center of symmetry.

Signal distortion and noise reduction

It is inevitable that the signal will be distorted in the convolution process. From property 3 above, when data which has a peak is smoothed the peak height will be reduced and the half-width will be increased. Both the extent of the distortion and S/N (signal-to-noise ratio) improvement:

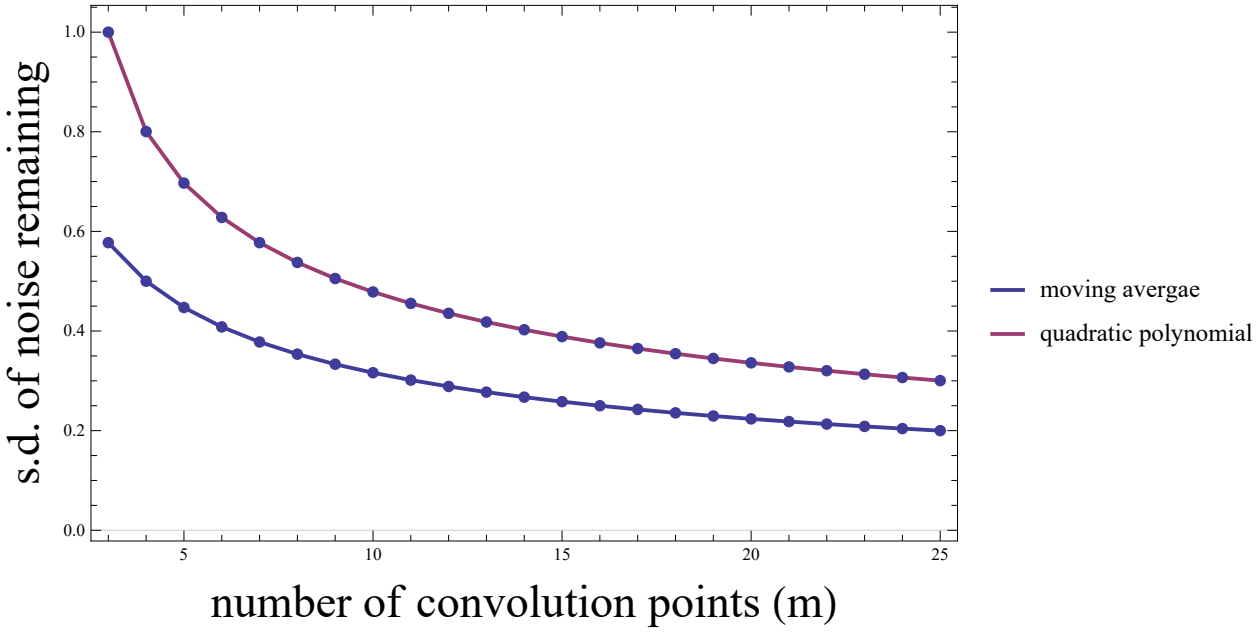
- decrease as the degree of the polynomial increases
- increase as the width, *m* of the convolution function increases

For example, If the noise in all data points is uncorrelated and has a constant standard deviation, σ, the standard deviation on the noise will be decreased by convolution with an *m*-point smoothing function to

`polynomial degree 0 or 1:` $\sqrt{\frac{1}{m}}\ \sigma$

`polynomial degree 2 or 3:` $\sqrt{\frac{3\,(3\,m^2-7)}{4\,m\,(m^2-4)}}\ \sigma$

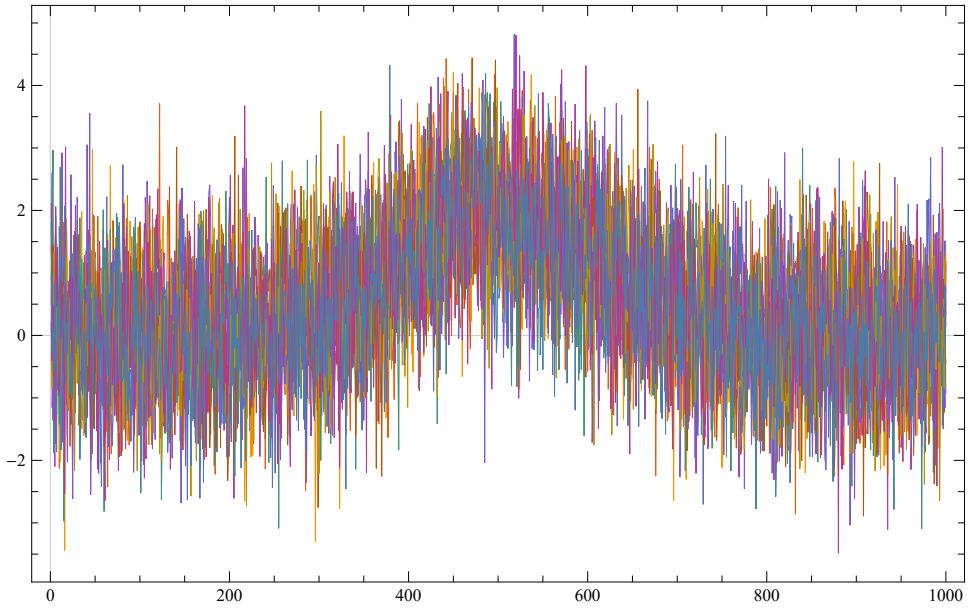
Signal distortion and noise reduction

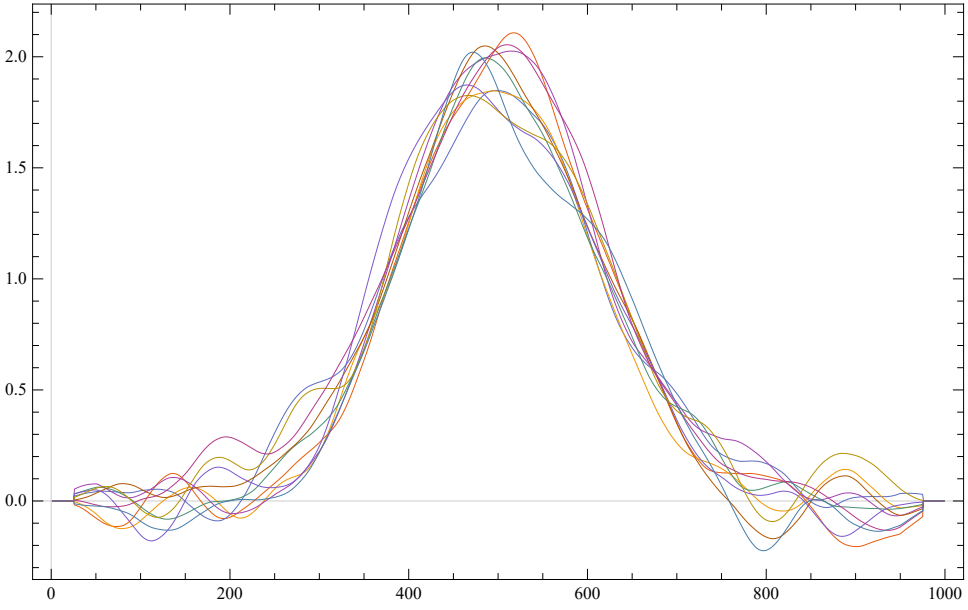


For example, with a 9-point linear function (moving average) two thirds of the noise is removed and with a 9-point quadratic/cubic smoothing function only about half the noise is removed. Most of the noise remaining is low-frequency noise

Example

Peak characterization after smoothing?

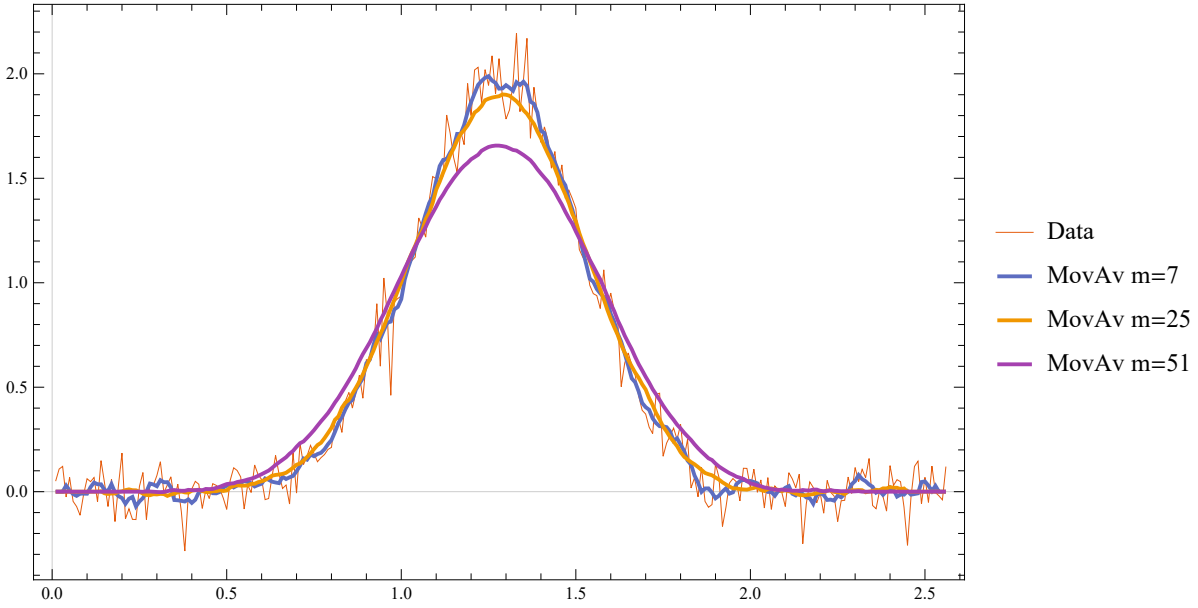

4
2
0
−2
0
200
400
600
800
1000


2.0
1.5
1.0
0.5
0.0
0
200
400
600
800
1000

These figures show ten superimposed plots with the same peak but with independent white noise, each plotted with a different line color, unsmoothed on the left and smoothed on the right (3 passes of a 51-point moving average). Inspection of the smoothed signals on the right clearly shows the variation in peak position, height, and width between the 10 samples caused by the low frequency noise remaining in the smoothed signals. Just because a signal looks smooth does not mean there is no noise. Low-frequency noise remaining in the signals after smoothing will still interfere with precise measurement of peak position, height, and width.

Smooth Ratio

The figures above show examples of the effect of three different smooth widths on noisy Gaussian-shaped peaks. In the figure on the left, the peak has a (true) height of 2.0 and there are 80 points in the half-width of the peak. The orange line is the original unsmoothed peak. The three superimposed lines are the results of smoothing this peak with a triangular smooth of width (from top to bottom) 7, 25, and 51 points. Because the peak width is 80 points, the smooth ratios of these three smooths are 7/80 = 0.09, 25/80 = 0.31, and 51/80 = 0.64, respectively.



Smooth Ratio

The figure, qualitatively demonstrates two points:

- Measuring the height of noisy peaks is much better done by curve fitting the unsmoothed data rather than by taking the maximum of the smoothed data. (smooth ratios below 0.2 should be used)
- If the objective of the measurement is to measure the peak position (x-axis value of the peak), much larger smooth ratios can be employed if desired, because smoothing has little effect on the peak position (unless peak is asymmetrical or the increase in peak width is so much that it causes adjacent peaks to overlap).

Local Regression - LOESS

From the NIST Engineering Statistics Handbook

LOESS is one of many “modern” modeling methods that build on “classical” methods, such as linear and nonlinear least squares regression. Modern regression methods are designed to address situations in which the classical procedures do not perform well or cannot be effectively applied without undue labor. LOESS combines much of the simplicity of linear least squares regression with the flexibility of nonlinear regression.

It does this by fitting simple models to localized subsets of the data to build up a function that describes the deterministic part of the variation in the data, point by point. In fact, one of the chief attractions of this method is that the data analyst is not required to specify a global function of any form to fit a model to the data, only to fit segments of the data.

Local Regression - LOESS

The trade-off for these features is increased computation. Because it is so computationally intensive, LOESS would have been practically impossible to use in the era when least squares regression was being developed. Most other modern methods for process modeling are similar to LOESS in this respect. These methods have been consciously designed to use our current computational ability to the fullest possible advantage to achieve goals not easily achieved by traditional approaches.

Local Regression - LOESS

LOESS, originally proposed by Cleveland (1979) and further developed by Cleveland and Devlin (1988), specifically denotes a method that is (somewhat) more descriptively known as locally weighted polynomial regression.

At each point in the data set a low-degree polynomial is fit to a subset of the data, with explanatory variable values near the point whose response is being estimated. The polynomial is fit using weighted least squares, giving more weight to points near the point whose response is being estimated and less weight to points further away. The value of the regression function for the point is then obtained by evaluating the local polynomial using the explanatory variable values for that data point.

The LOESS fit is complete after regression function values have been computed for each of the n data points. Many of the details of this method, such as the degree of the polynomial model and the weights, are flexible.

Localized Subsets of Data

The subsets of data used for each weighted least squares fit in LOESS are determined by a nearest neighbors algorithm.

A user-specified input to the procedure called the "bandwidth" or "smoothing parameter" determines how much of the data is used to fit each local polynomial.

The smoothing parameter, q, is a number between $(d+1)/n$ and 1, with d denoting the degree of the local polynomial. The value of q is the proportion of data used in each fit. The subset of data used in each weighted least squares fit is comprised of the $n\,q$ (rounded to the next largest integer) points whose explanatory variables values are closest to the point at which the response is being estimated.

Localized Subsets of Data

q is called the smoothing parameter because it controls the flexibility of the LOESS regression function. Large values of q produce the smoothest functions that wiggle the least in response to fluctuations in the data. The smaller *q* is, the closer the regression function will conform to the data. Using too small a value of the smoothing parameter is not desirable, however, since the regression function will eventually start to capture the random error in the data. Useful values of the smoothing parameter typically lie in the range 0.25 to 0.5 for most LOESS applications.

Degree of Local Polynomials

The local polynomials fit to each subset of the data are almost always of first or second degree; that is, either locally linear (in the straight line sense) or locally quadratic. Using a zero degree polynomial turns LOESS into a weighted moving average. Such a simple local model might work well for some situations, but may not always approximate the underlying function well enough. Higher-degree polynomials would work in theory, but yield models that are not really in the spirit of LOESS. LOESS is based on the ideas that any function can be well approximated in a small neighborhood by a low-order polynomial and that simple models can be fit to data easily. High-degree polynomials would tend to overfit the data in each subset and are numerically unstable, making accurate computations difficult.

Weight Function

As mentioned above, the weight function gives the most weight to the data points nearest the point of estimation and the least weight to the data points that are furthest away. The use of the weights is based on the idea that points near each other in the explanatory variable space are more likely to be related to each other in a simple way than points that are further apart. Following this logic, points that are likely to follow the local model best influence the local model parameter estimates the most. Points that are less likely to actually conform to the local model have less influence on the local model parameter estimates.

The traditional weight function used for LOESS is the tri-cube weight function,

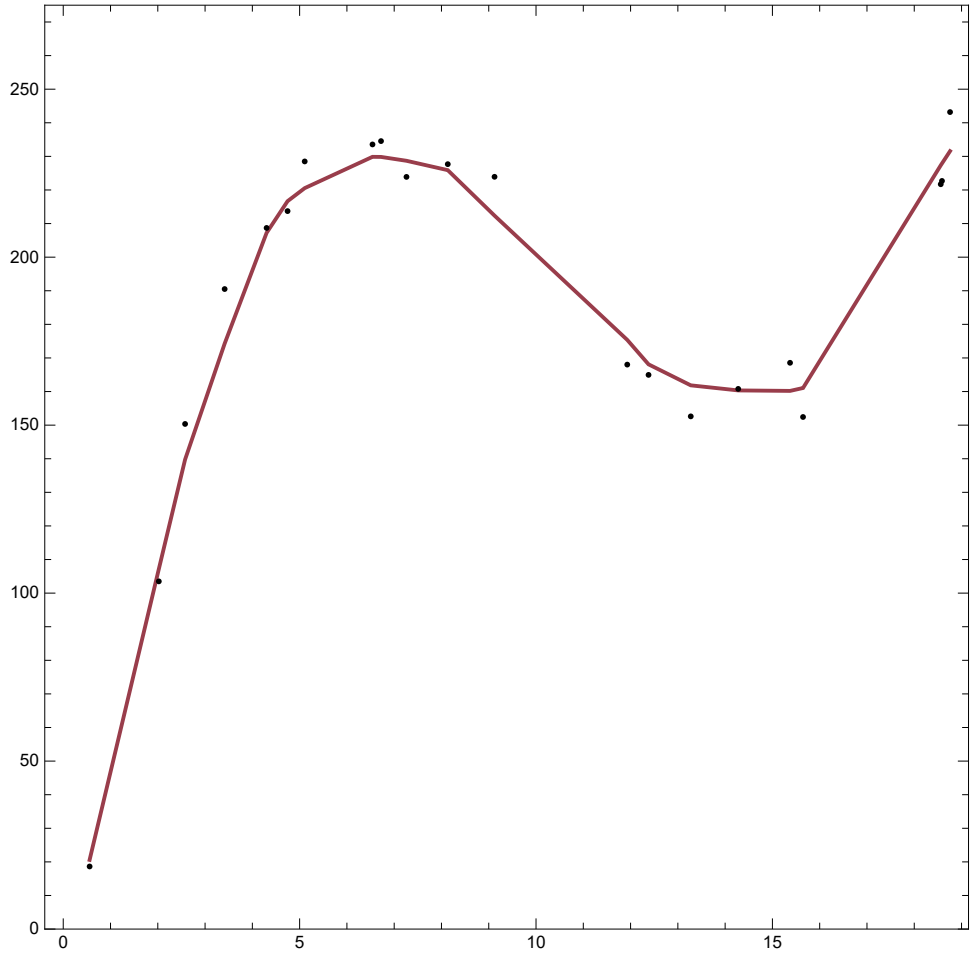
Out[ ]//TraditionalForm=

$$w(x) = \begin{cases} \left(1 - |x|^3\right)^3 & |x| < 1 \\ 0 & |x| \geq 1 \end{cases}$$

Localized Subsets of Data

However, any other weight function that satisfies the properties listed in Cleveland (1979) could also be used. The weight for a specific point in any localized subset of data is obtained by evaluating the weight function at the distance between that point and the point of estimation, after scaling the distance so that the maximum absolute distance over all of the points in the subset of data is exactly one.

Example




```
Grid[loess = Table[tabvalues[data, i], {i, 1, Length[data]}], Frame → All]
```

0.55782

x	y	Distance	Scaled Distance	Weight
0.55782	18.6365	0.	0.	1.
2.02173	103.496	1.46391	0.321769	0.903349
2.57733	150.354	2.01951	0.44389	0.75989
3.41403	190.51	2.85621	0.627799	0.426217
4.30141	208.701	3.74359	0.822847	0.0868617
4.74484	213.711	4.18702	0.920313	0.0107231
5.10738	228.494	4.54956	1.	0

$-12.3368 + 59.0331\,x$

20.593

2.02173

x	y	Distance	Scaled Distance	Weight
0.55782	18.6365	1.46391	0.474424	0.712642
2.02173	103.496	0.	0.	1.
2.57733	150.354	0.555598	0.180059	0.982589
3.41403	190.51	1.3923	0.451218	0.748942
4.30141	208.701	2.27968	0.738801	0.212501
4.74484	213.711	2.72311	0.882508	0.0305716
5.10738	228.494	3.08565	1.	0

$-7.17613 + 56.5538\,x$

107.16

2.57733

x	y	Distance	Scaled Distance	Weight
0.55782	18.6365	2.01951	0.798207	0.118686
2.02173	103.496	0.555598	0.219599	0.968565
2.57733	150.354	0.	0.	1.
3.41403	190.51	0.836704	0.330706	0.895373
4.30141	208.701	1.72408	0.681442	0.319402
4.74484	213.711	2.16751	0.856707	0.0511567
5.10738	228.494	2.53005	1.	0

$8.17927 + 51.0561\,x$

139.767

3.41403

x	y	Distance	Scaled Distance	Weight
0.55782	18.6365	2.85621	1.	0
2.02173	103.496	1.3923	0.487465	0.6912
2.57733	150.354	0.836704	0.292942	0.926464
3.41403	190.51	0.	0.	1.
4.30141	208.701	0.88738	0.310684	0.912705
4.74484	213.711	1.33081	0.465936	0.726202
5.10738	228.494	1.69335	0.592866	0.496066

$49.3158 + 36.5982\,x$

174.263

4.30141

x	y	Distance	Scaled Distance	Weight
2.02173	103.496	2.27968	1.	0
2.57733	150.354	1.72408	0.756283	0.182703
3.41403	190.51	0.88738	0.389256	0.83329
4.30141	208.701	0.	0.	1.
4.74484	213.711	0.443431	0.194514	0.978083
5.10738	228.494	0.80597	0.353545	0.873199
6.54117	233.554	2.23976	0.982487	0.000137576

$101.727 + 24.5283\,x$

207.233

4.74484

x	y	Distance	Scaled Distance	Weight
2.57733	150.354	2.16751	1.	0
3.41403	190.51	1.33081	0.61398	0.453953
4.30141	208.701	0.443431	0.20458	0.974532
4.74484	213.711	0.	0.	1.
5.10738	228.494	0.362539	0.16726	0.986028
6.54117	233.554	1.79633	0.82875	0.0799477
6.72162	234.551	1.97678	0.912002	0.0140749

129.589 + 18.351 x

216.662

5.10738

x	y	Distance	Scaled Distance	Weight
3.41403	190.51	1.69335	0.786624	0.135207
4.30141	208.701	0.80597	0.374403	0.85067
4.74484	213.711	0.362539	0.168413	0.985738
5.10738	228.494	0.	0.	1.
6.54117	233.554	1.43379	0.666048	0.349699
6.72162	234.551	1.61424	0.749874	0.193439
7.26006	223.892	2.15268	1.	0

157.427 + 12.3581 x

220.544

6.54117

x	y	Distance	Scaled Distance	Weight
4.30141	208.701	2.23976	1.	0
4.74484	213.711	1.79633	0.802018	0.113461
5.10738	228.494	1.43379	0.640153	0.401405
6.54117	233.554	0.	0.	1.
6.72162	234.551	0.180451	0.0805674	0.998432
7.26006	223.892	0.718892	0.320969	0.904045
8.13359	227.683	1.59242	0.710979	0.26289

229.492 + 0.0563287 x

229.861

6.72162

x	y	Distance	Scaled Distance	Weight
4.74484	213.711	1.97678	0.823376	0.0862299
5.10738	228.494	1.61424	0.67237	0.337203
6.54117	233.554	0.180451	0.0751624	0.998727
6.72162	234.551	0.	0.	1.
7.26006	223.892	0.538441	0.224274	0.966538
8.13359	227.683	1.41197	0.58812	0.505458
9.12244	223.92	2.40082	1.	0

234.658 – 0.717589 x

229.835

7.26006

x	y	Distance	Scaled Distance	Weight
5.10738	228.494	2.15268	0.460998	0.733942
6.54117	233.554	0.718892	0.153951	0.989093
6.72162	234.551	0.538441	0.115307	0.995408
7.26006	223.892	0.	0.	1.
8.13359	227.683	0.873529	0.187067	0.98049
9.12244	223.92	1.86238	0.39883	0.821499
11.9297	168.02	4.66961	1.	0

242.14 – 1.85307 x

228.686

8.13359	x y Distance Scaled Distance Weight 6.54117 233.554 1.59242 0.375024 0.849965 6.72162 234.551 1.41197 0.332527 0.893699 7.26006 223.892 0.873529 0.205721 0.974108 8.13359 227.683 0. 0. 1. 9.12244 223.92 0.98885 0.23288 0.962587 11.9297 168.02 3.79608 0.893999 0.0232679 12.3798 164.958 4.24618 1. 0	262.176 – 4.46024 x	225.898
9.12244	x y Distance Scaled Distance Weight 6.72162 234.551 2.40082 0.578452 0.524477 7.26006 223.892 1.86238 0.44872 0.752702 8.13359 227.683 0.98885 0.238253 0.959973 9.12244 223.92 0. 0. 1. 11.9297 168.02 2.80723 0.676371 0.32933 12.3798 164.958 3.25733 0.784818 0.137867 13.2729 152.611 4.15042 1. 0	316.136 – 11.3757 x	212.362
11.9297	x y Distance Scaled Distance Weight 7.26006 223.892 4.66961 1. 0 8.13359 227.683 3.79608 0.812933 0.0991017 9.12244 223.92 2.80723 0.60117 0.479559 11.9297 168.02 0. 0. 1. 12.3798 164.958 0.450101 0.0963895 0.997316 13.2729 152.611 1.3432 0.287646 0.930286 14.2767 160.787 2.34708 0.502629 0.66538	331.722 – 13.1059 x	175.372
12.3798	x y Distance Scaled Distance Weight 8.13359 227.683 4.24618 1. 0 9.12244 223.92 3.25733 0.76712 0.165081 11.9297 168.02 0.450101 0.106001 0.996431 12.3798 164.958 0. 0. 1. 13.2729 152.611 0.893095 0.210329 0.972345 14.2767 160.787 1.89698 0.446749 0.755649 15.3731 168.556 2.99334 0.704948 0.274214	251.944 – 6.77263 x	168.1
13.2729	x y Distance Scaled Distance Weight 9.12244 223.92 4.15042 1. 0 11.9297 168.02 1.3432 0.323629 0.901722 12.3798 164.958 0.893095 0.215182 0.970406 13.2729 152.611 0. 0. 1. 14.2767 160.787 1.00388 0.241875 0.958146 15.3731 168.556 2.10024 0.50603 0.659463 15.6477 152.427 2.3748 0.572183 0.536716	181.696 – 1.49535 x	161.849

14.2767

x	y	Distance	Scaled Distance	Weight
11.9297	168.02	2.34708	0.547898	0.583283
12.3798	164.958	1.89698	0.442827	0.761457
13.2729	152.611	1.00388	0.234345	0.961886
14.2767	160.787	0.	0.	1.
15.3731	168.556	1.09636	0.255932	0.950547
15.6477	152.427	1.37092	0.320025	0.904861
18.5605	221.707	4.28379	1.	0

176.391 – 1.12459 x

160.335

15.3731

x	y	Distance	Scaled Distance	Weight
12.3798	164.958	2.99334	0.931478	0.007056
13.2729	152.611	2.10024	0.653561	0.37455
14.2767	160.787	1.09636	0.341169	0.885536
15.3731	168.556	0.	0.	1.
15.6477	152.427	0.274561	0.085439	0.99813
18.5605	221.707	3.18743	0.991878	0.0000141168
18.5866	222.69	3.21353	1.	0

143.181 + 1.10657 x

160.192

15.6477

x	y	Distance	Scaled Distance	Weight
13.2729	152.611	2.3748	0.763696	0.170574
14.2767	160.787	1.37092	0.440864	0.764337
15.3731	168.556	0.274561	0.0882942	0.997936
15.6477	152.427	0.	0.	1.
18.5605	221.707	2.91287	0.93673	0.0056449
18.5866	222.69	2.93897	0.945123	0.00377901
18.7573	243.188	3.10962	1.	0

140.665 + 1.30314 x

161.056

18.5605

x	y	Distance	Scaled Distance	Weight
13.2729	152.611	5.28767	1.	0
14.2767	160.787	4.28379	0.810146	0.102681
15.3731	168.556	3.18743	0.602804	0.476301
15.6477	152.427	2.91287	0.55088	0.577646
18.5605	221.707	0.	0.	1.
18.5866	222.69	0.0260999	0.00493599	1.
18.7573	243.188	0.196746	0.0372084	0.999845

−166.603 + 21.2248 x

227.34

18.5866

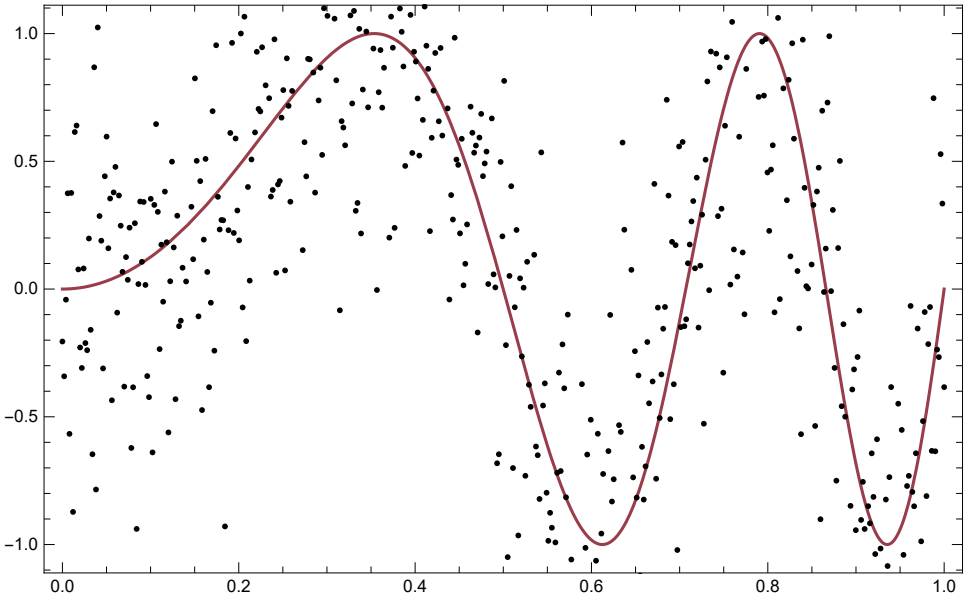
x	y	Distance	Scaled Distance	Weight
13.2729	152.611	5.31377	1.	0
14.2767	160.787	4.30989	0.811079	0.101477
15.3731	168.556	3.21353	0.604755	0.472408
15.6477	152.427	2.93897	0.553086	0.573461
18.5605	221.707	0.0260999	0.00491174	1.
18.5866	222.69	0.	0.	1.
18.7573	243.188	0.170646	0.0321139	0.999901

−166.745 + 21.2327 x

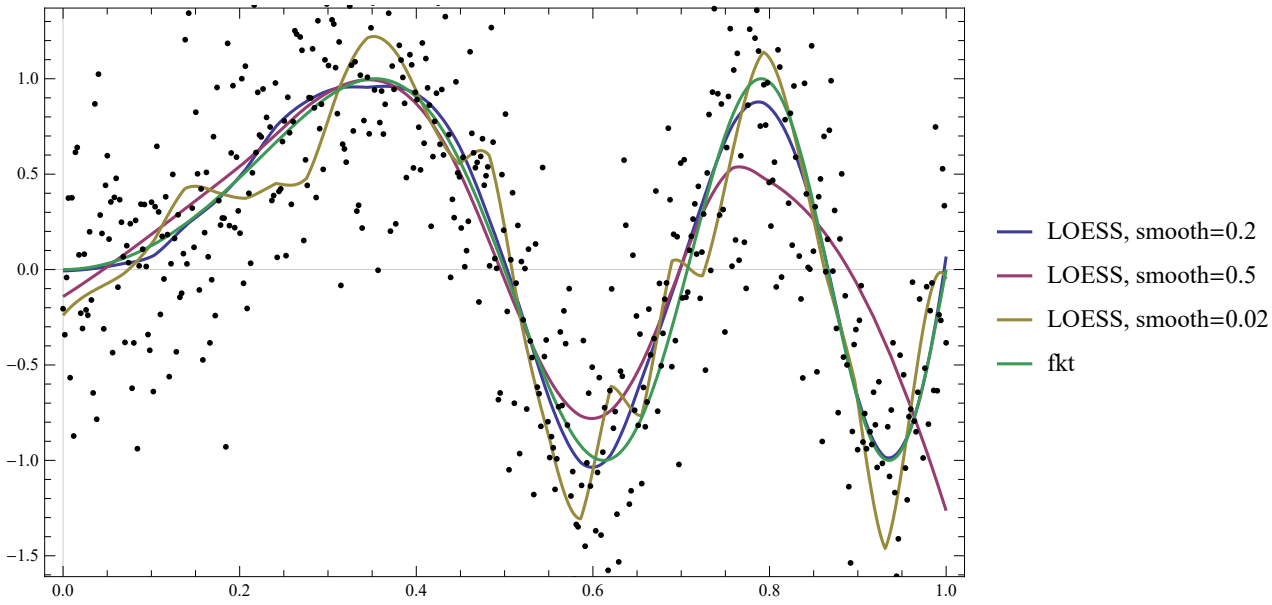
227.899

18.7573	x y Distance Scaled Distance Weight 13.2729 152.611 5.48442 1. 0 14.2767 160.787 4.48054 0.816957 0.0940394 15.3731 168.556 3.38418 0.617053 0.447792 15.6477 152.427 3.10962 0.566991 0.54679 18.5605 221.707 0.196746 0.0358736 0.999862 18.5866 222.69 0.170646 0.0311147 0.99991 18.7573 243.188 0. 0. 1.	−167.643 + 21.2825 x	231.559

Example



The interpolation strongly depends on the smoothing range:


1.0
0.5
0.0
−0.5
−1.0
−1.5
0.0
0.2
0.4
0.6
0.8
1.0
LOESS, smooth=0.2
LOESS, smooth=0.5
LOESS, smooth=0.02
fkt

- Init