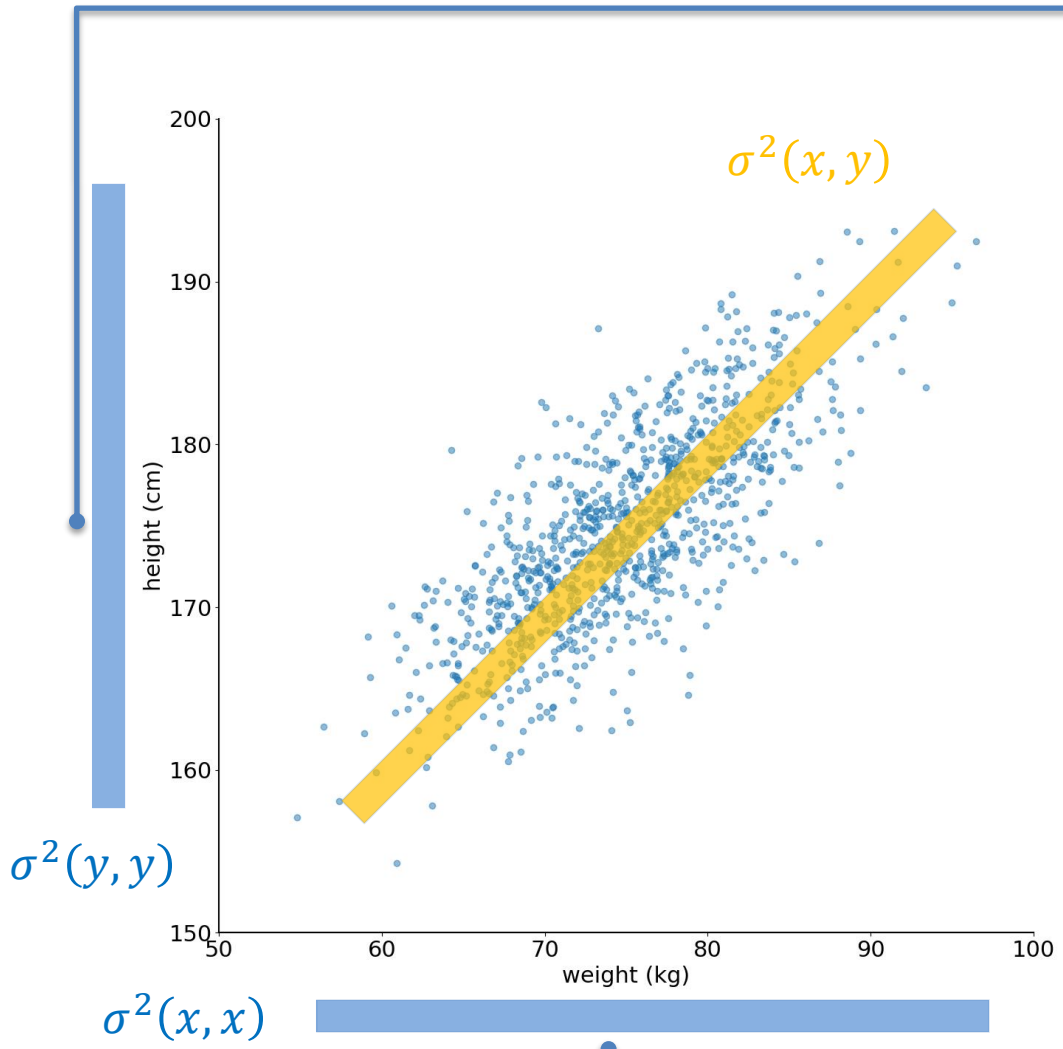
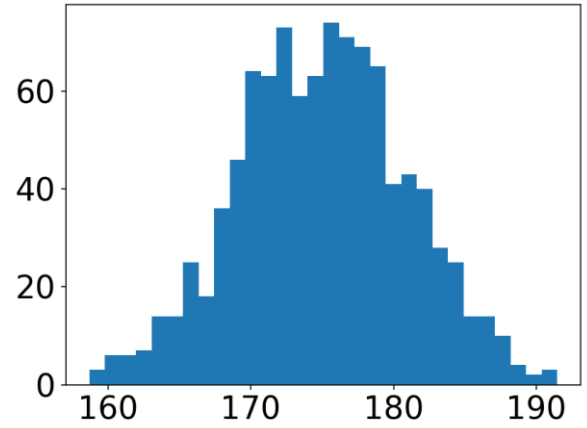
PFE 6

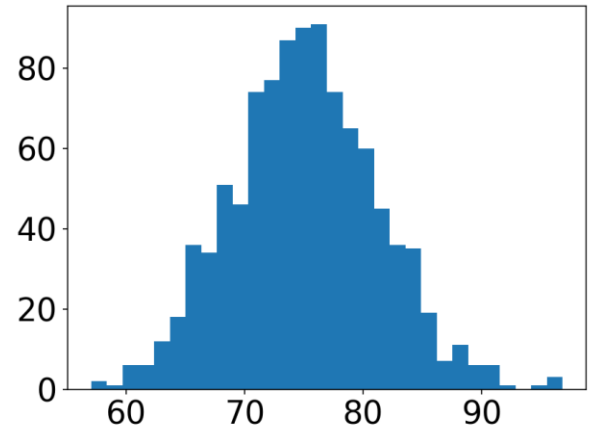
Dr W Palubicki

Covariance



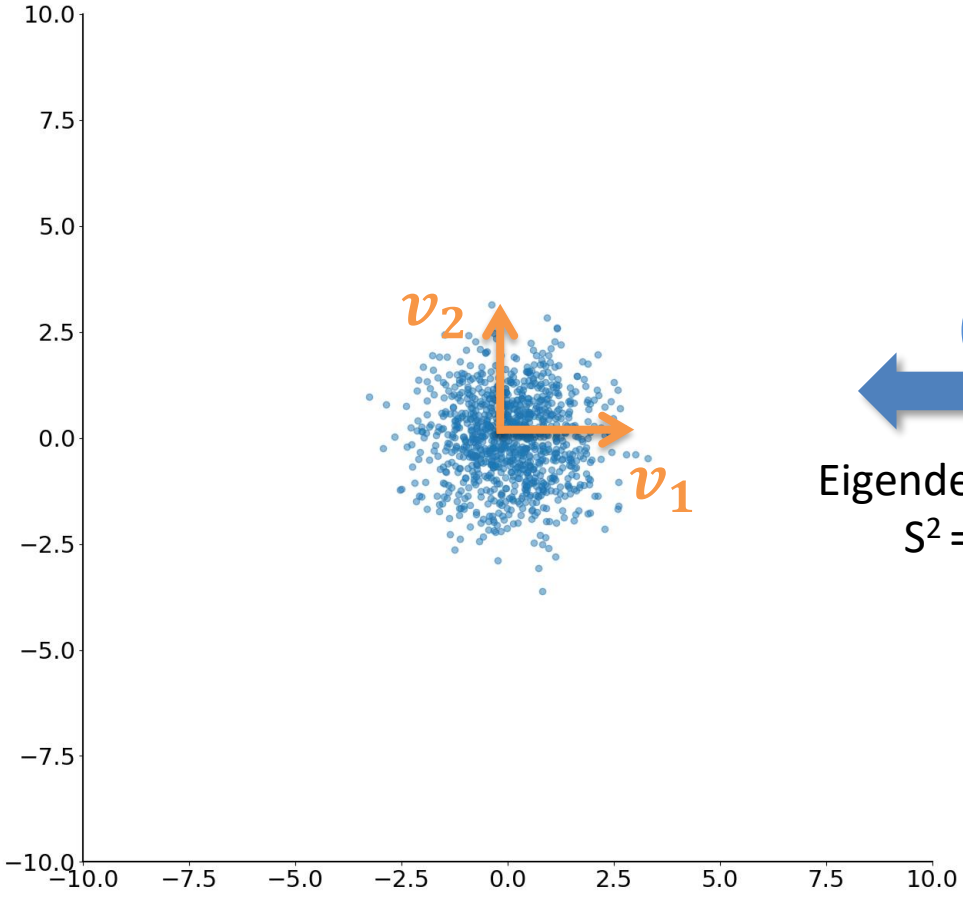


Variance in height = 36.9



Variance in weight = 35.0

Covariance as Linear Transformations

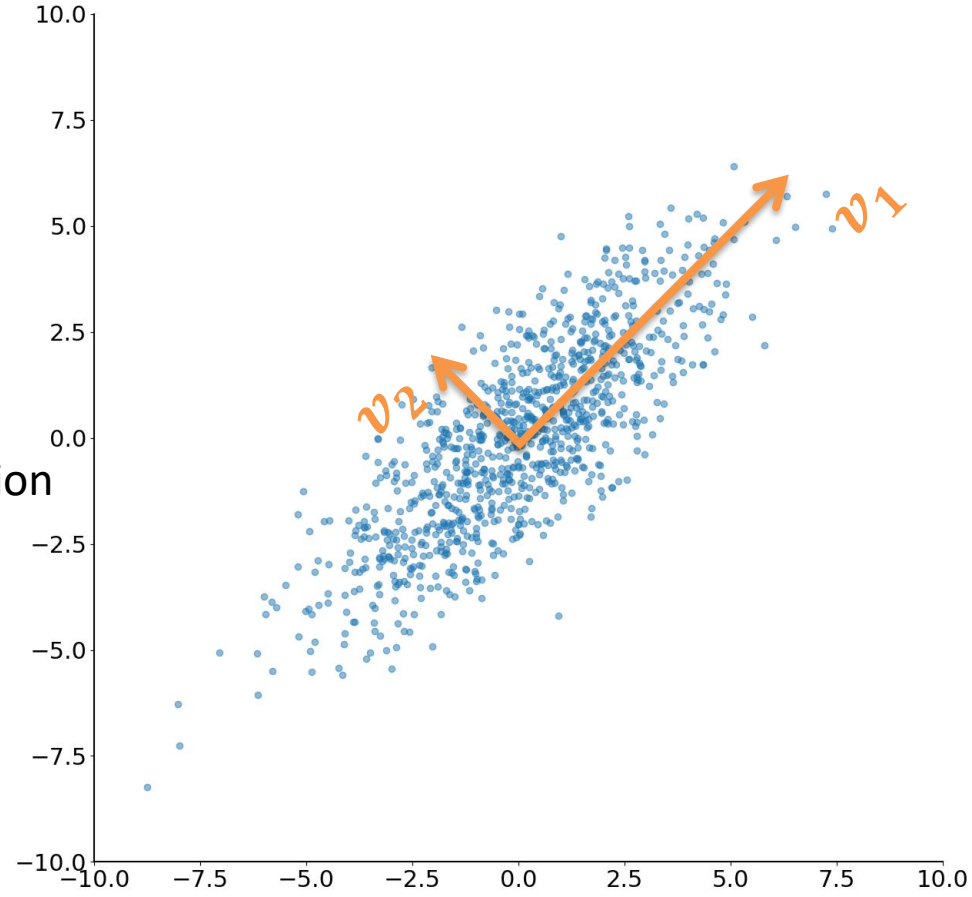


Σ

$(SR)^{-1}$

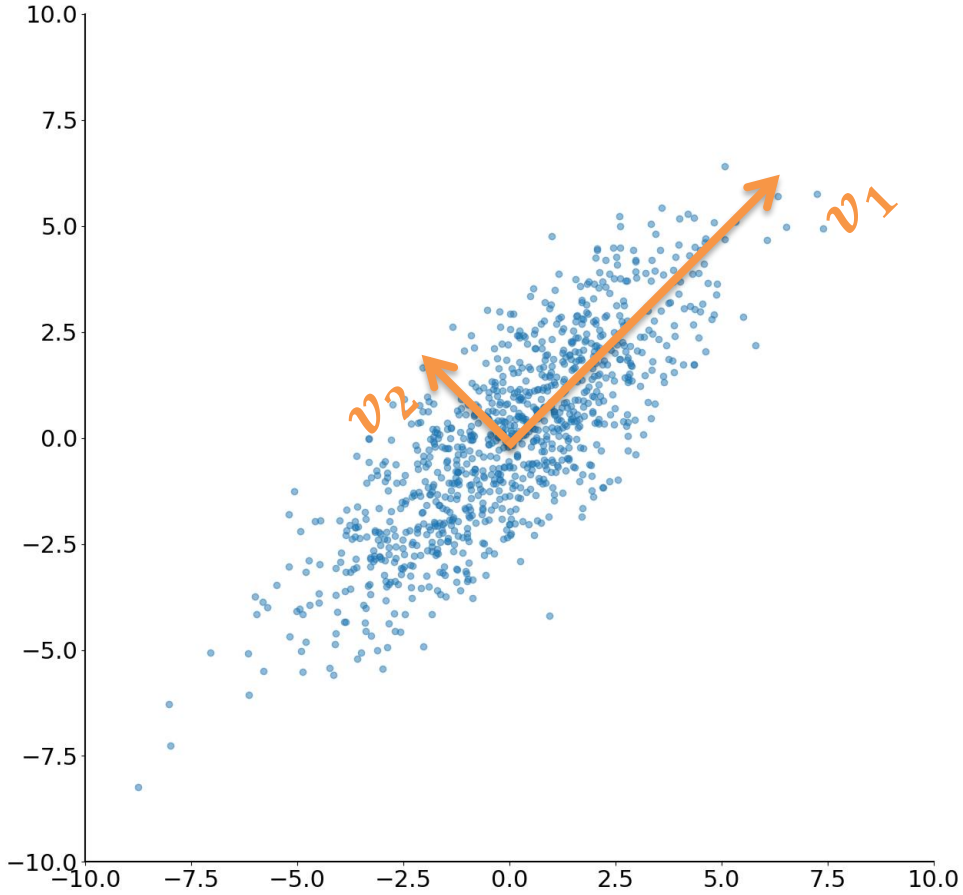
Eigendecomposition

$S^2 = L$, $R = V$

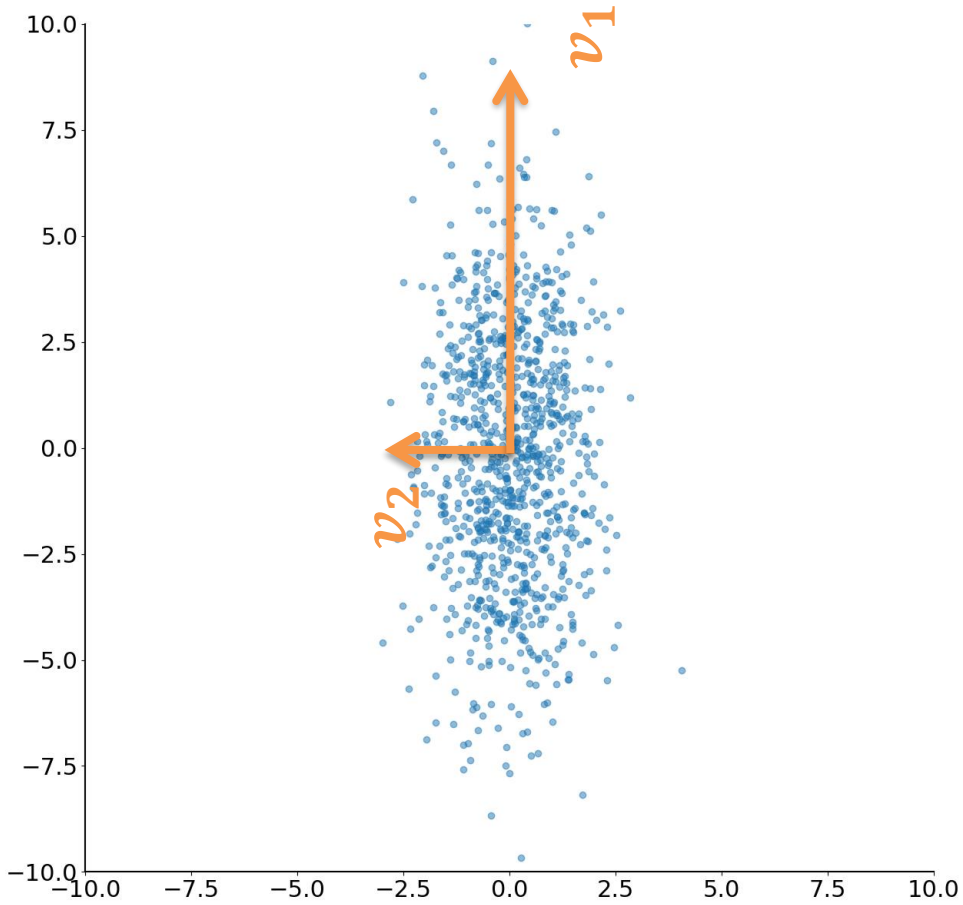


$\Sigma' = RSSR^{-1}$

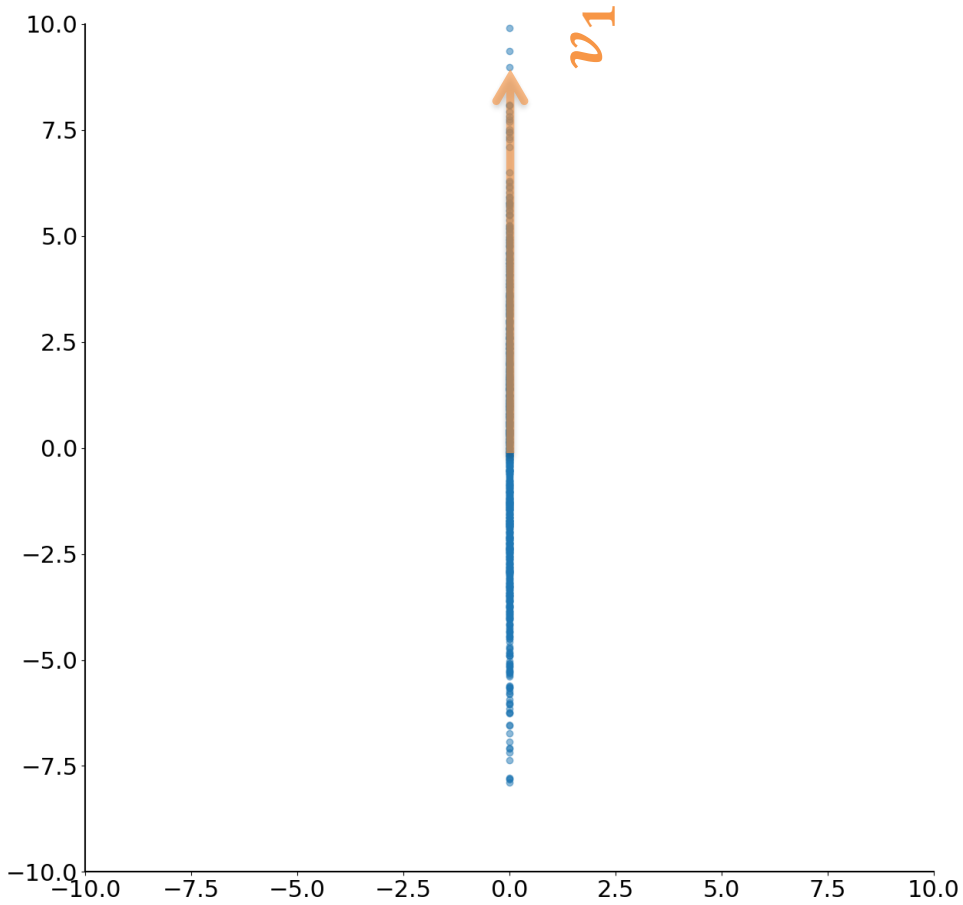
Dimensionality Reduction



Dimensionality Reduction

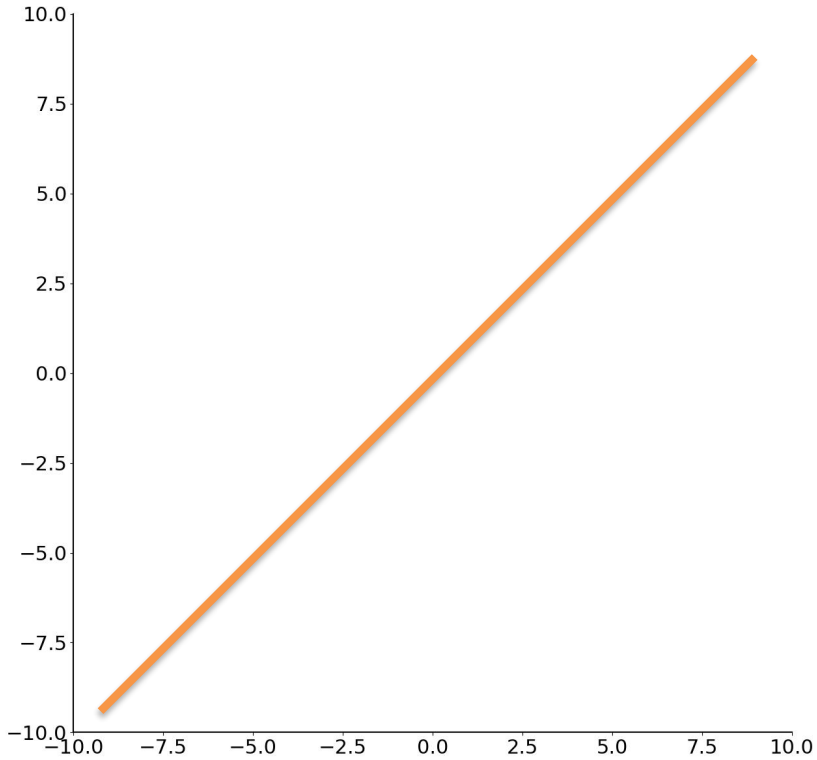


Dimensionality Reduction



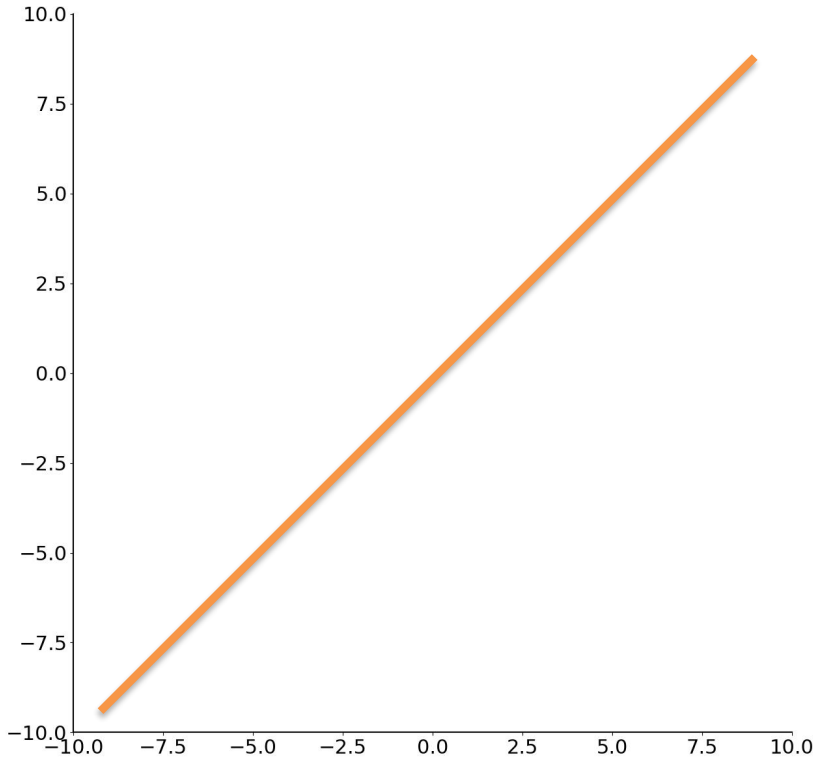
Projection of data on largest eigenvector

Data Model

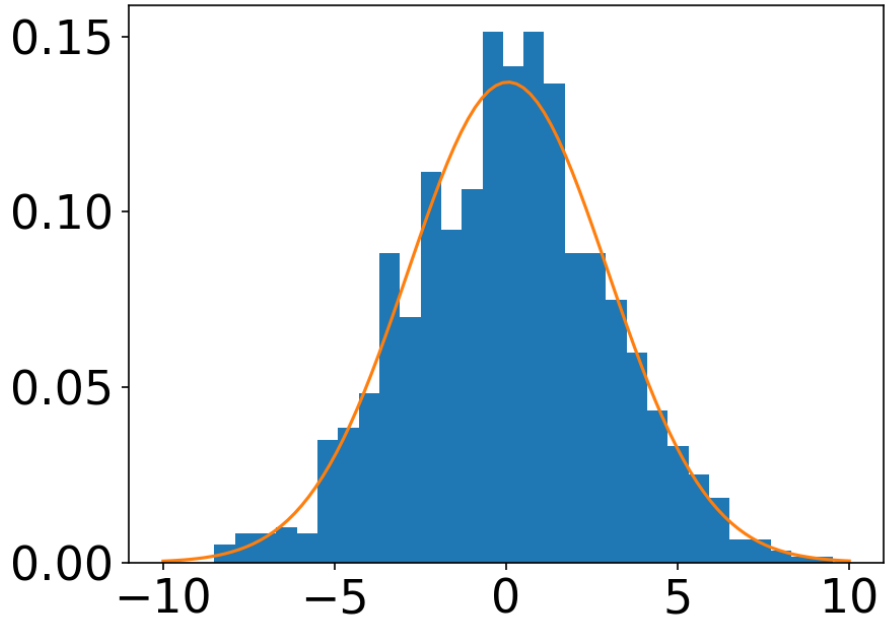


gradient = 1, offset = 0

Data Model

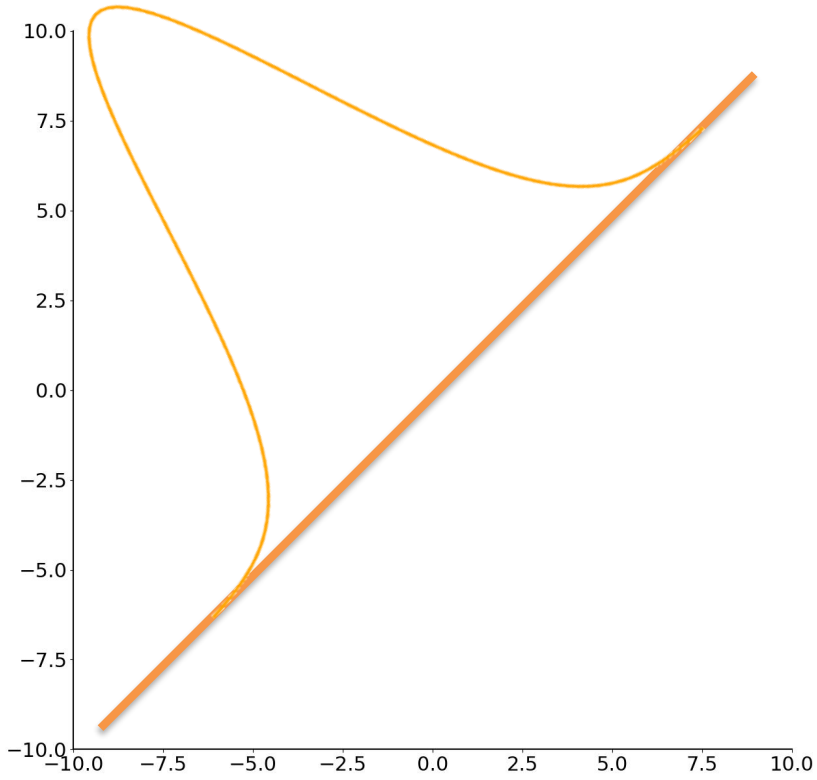


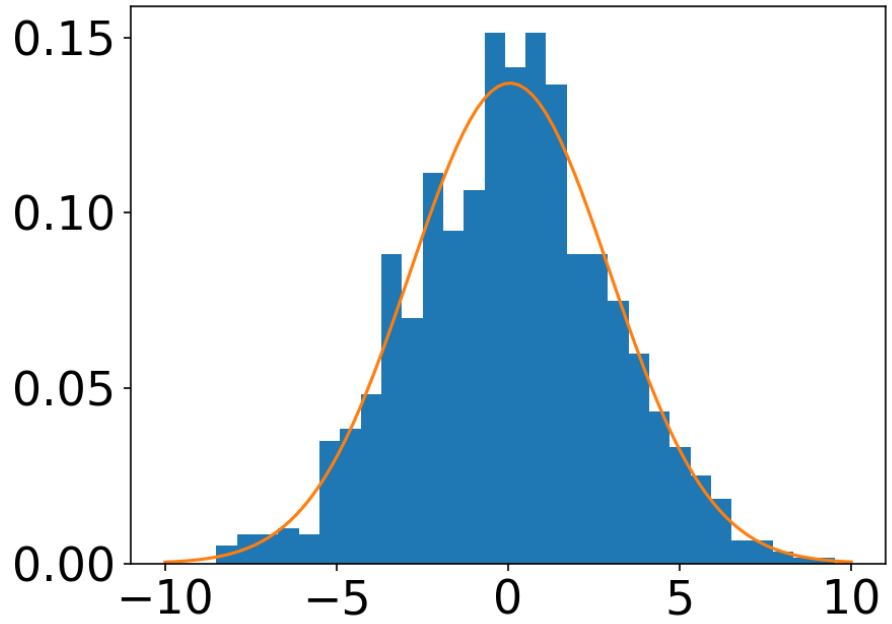
gradient = 1, offset = 0



mu = 0.05, sigma = 2.91

Data Model





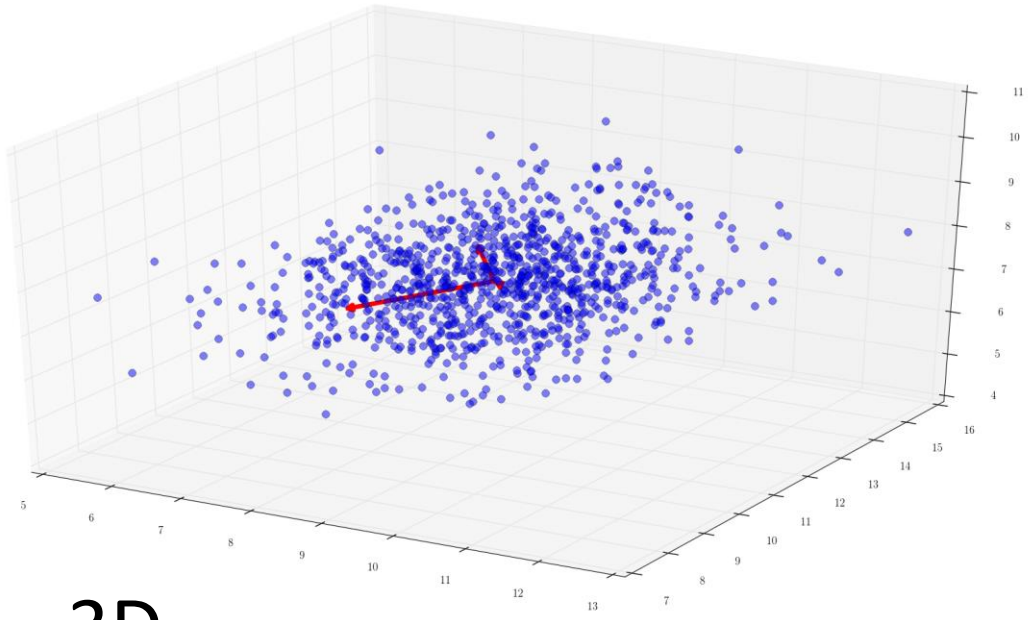
gradient = 1, offset = 0

mu = 0.05, sigma = 2.91

Describe data set of 2000 points with just 4 numbers!

Principal component analysis (PCA)

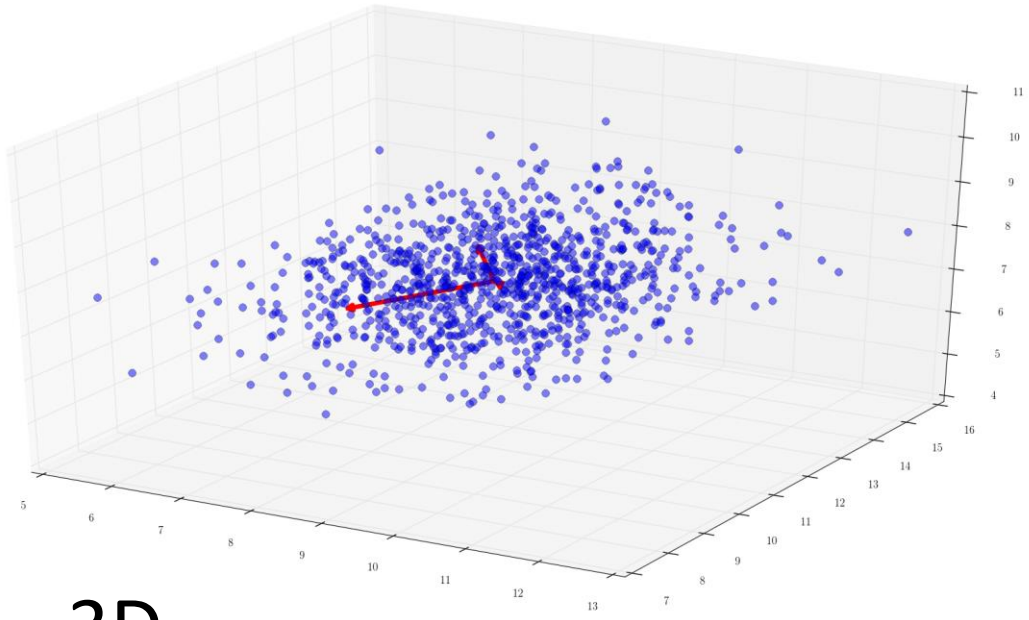
- Reduce dimensions of data sample by projecting onto a lower dimension (while preserving the high variance information)



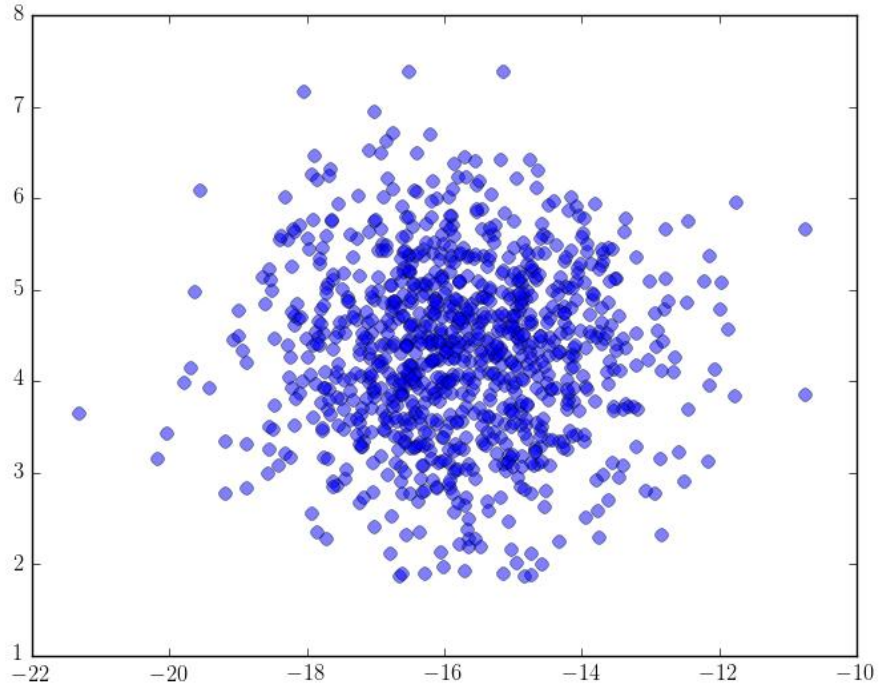
3D

Principal component analysis (PCA)

- Reduce dimensions of data sample by projecting onto a lower dimension (while preserving the high variance information)

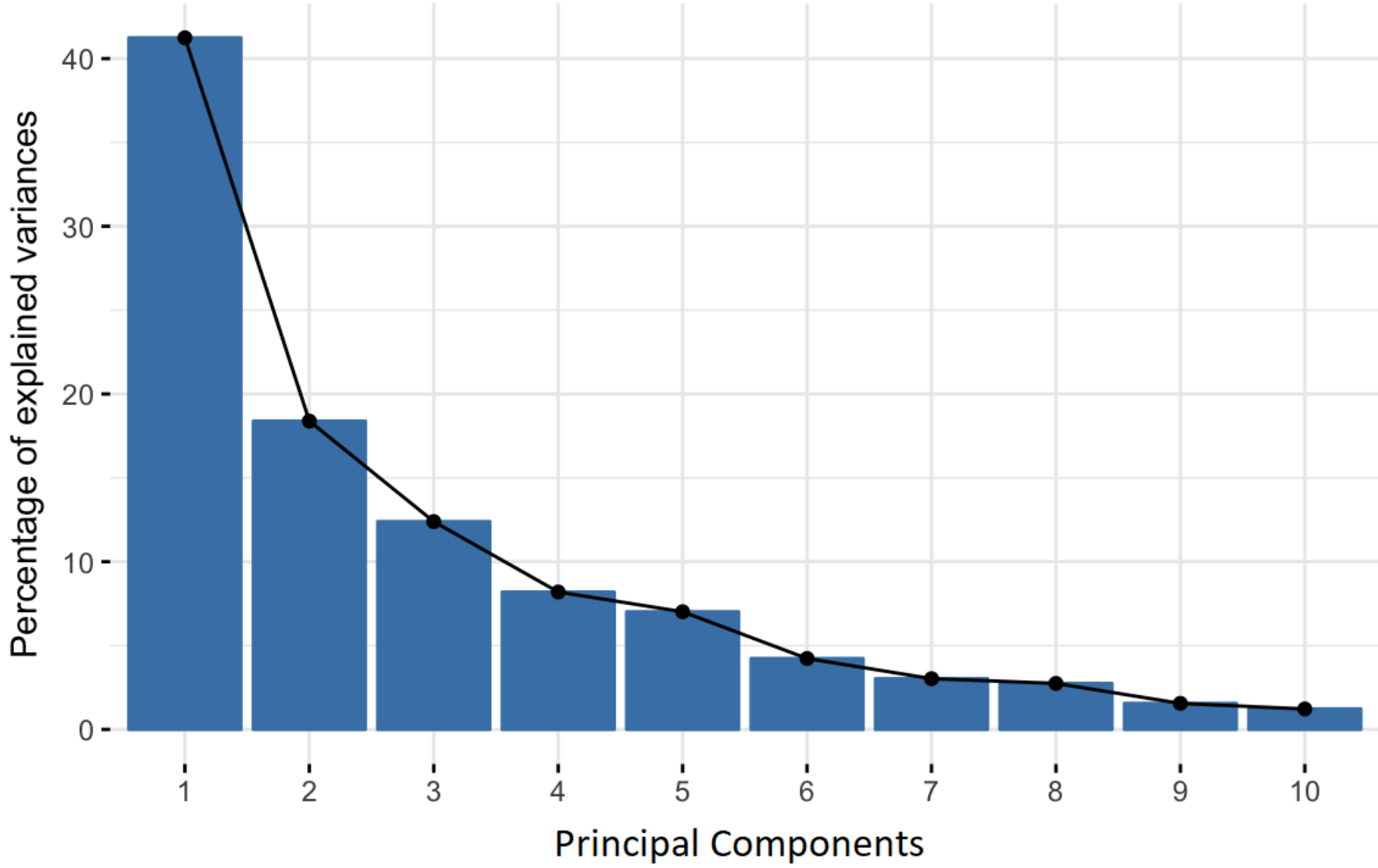


3D



2D

10-Dimensional case



PCA Algorithm

- **Input**: Data X of sample size N
- **Output**: k principal components
- **Centering**: Subtract mean from data
- **Scaling**: Scale each dimension by its variance
- Compute **covariance matrix** C
- Compute k largest **eigenvectors** of C (alternatively calculate **SVD**)

PCA Algorithm

- **Input**: Data X of sample size N
- **Output**: k principal components
- **Centering**: Subtract mean from data
- **Scaling**: Scale each dimension by its variance
- Compute **covariance matrix** C
- Compute k largest **eigenvectors** of C (alternatively calculate **SVD**)

Describes only linear relations in data set!

sklearn.decomposition.PCA

class `sklearn.decomposition.` **`PCA`** (*n_components=None*, *copy=True*, *whiten=False*, *svd_solver='auto'*, *tol=0.0*, *iterated_power='auto'*, *random_state=None*) [source]

Principal component analysis (PCA)

Linear dimensionality reduction using Singular Value Decomposition of the data to project it to a lower dimensional space.

It uses the LAPACK implementation of the full SVD or a randomized truncated SVD by the method of Halko et al. 2009, depending on the shape of the input data and the number of components to extract.

It can also use the scipy.sparse.linalg ARPACK implementation of the truncated SVD.

Notice that this class does not support sparse input. See `TruncatedSVD` for an alternative with sparse data.

Read more in the User Guide.

Parameters:

n_components : *int, float, None or string*

Number of components to keep. if n_components is not set all components are kept:

```
n_components == min(n_samples, n_features)
```

If `n_components == 'mle'` and `svd_solver == 'full'`, Minka's MLE is used to guess the dimension. Use of `n_components == 'mle'` will interpret `svd_solver == 'auto'` as `svd_solver == 'full'`.

If `0 < n_components < 1` and `svd_solver == 'full'`, select the number of components such that the amount of variance that needs to be explained is greater than the percentage specified by n_components.

If `svd_solver == 'arpack'`, the number of components must be strictly less than the minimum

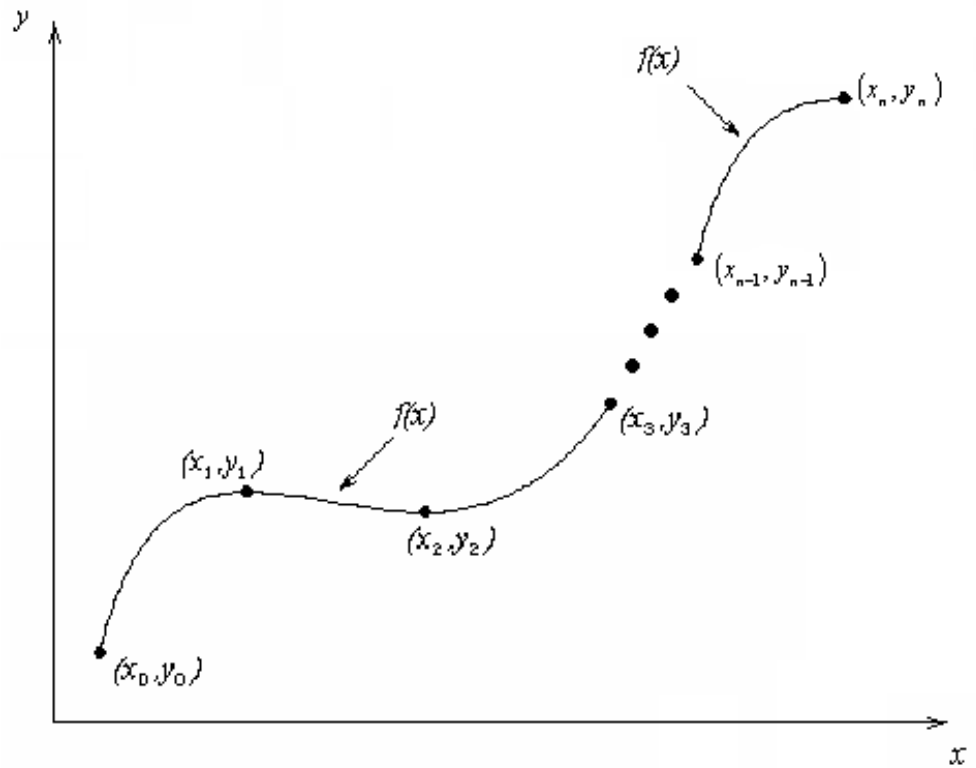
Overview – Next Topics

- **Interpolation**
- **Statistics**
- **Differential equations**
- **Modeling of real world patterns (in 3D)**

What is interpolation?

What is interpolation?

- For given points (x_0, y_0), (x_1, y_1), … (x_n, y_n) find new points within the range of the given points, e.g. via a function f

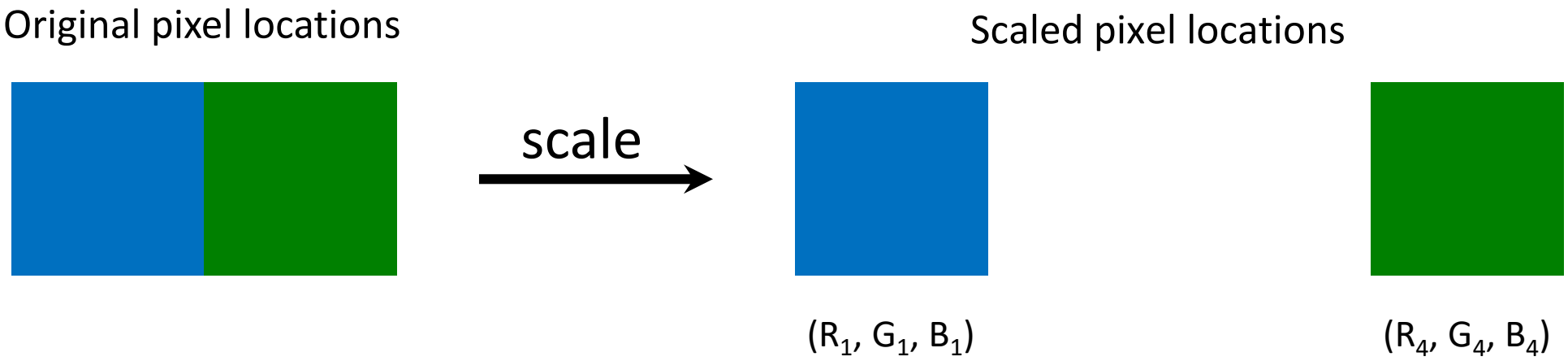


Interpolation of images

low-resolution
image (100×100)

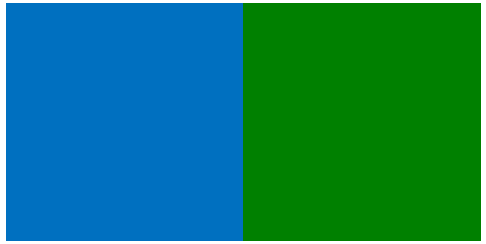
high-resolution
image (400×400)

Color interpolation



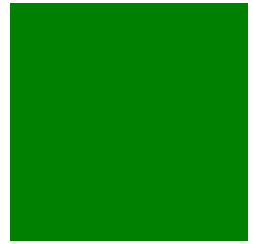
Color interpolation

Original pixel locations



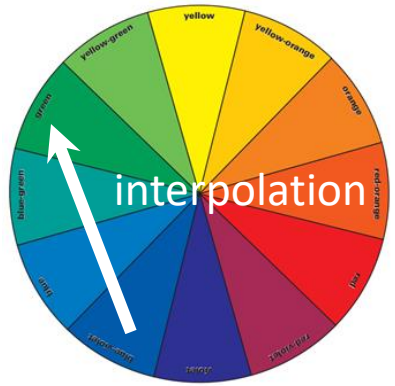
scale

Scaled pixel locations



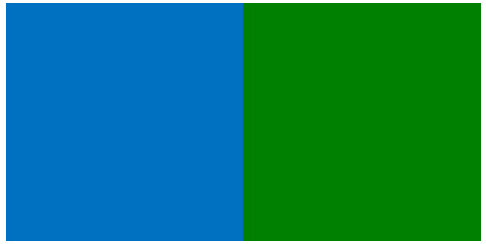
(R_1, G_1, B_1)

(R_4, G_4, B_4)



Color interpolation

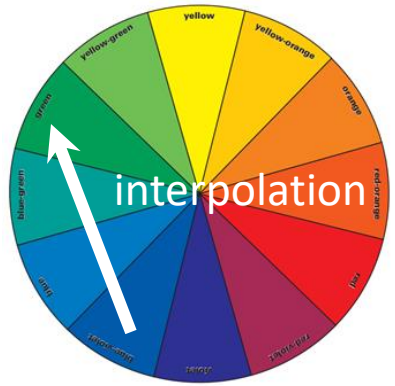
Original pixel locations



scale

Scaled pixel locations

(R_1, G_1, B_1) (R_2, G_2, B_2) (R_3, G_3, B_3) (R_4, G_4, B_4)



Edge-directed interpolation (Li&Orchard’2000)

low-resolution
image (100×100)

high-resolution
image (400×400)

Coloring of images

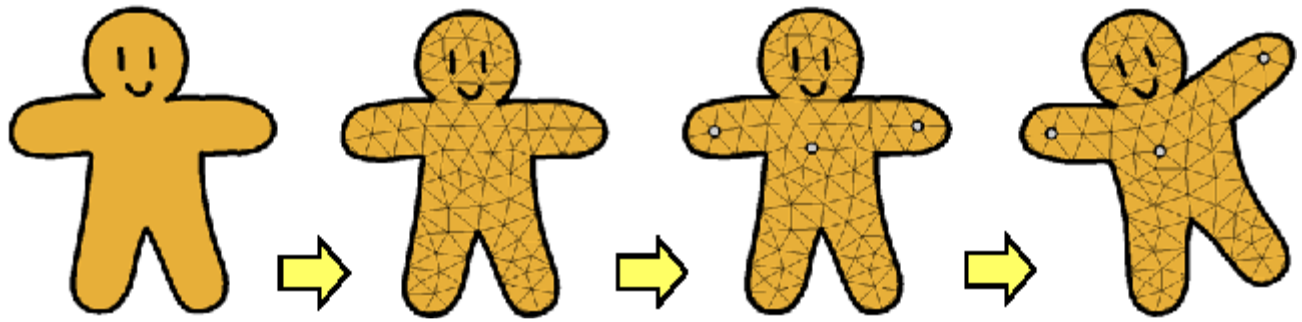
http://www.cs.huji.ac.il/~yweiss/Colorization/

Shape interpolation

http://www.cs.tau.ac.il/~dcor/online_papers/papers/arap.pdf

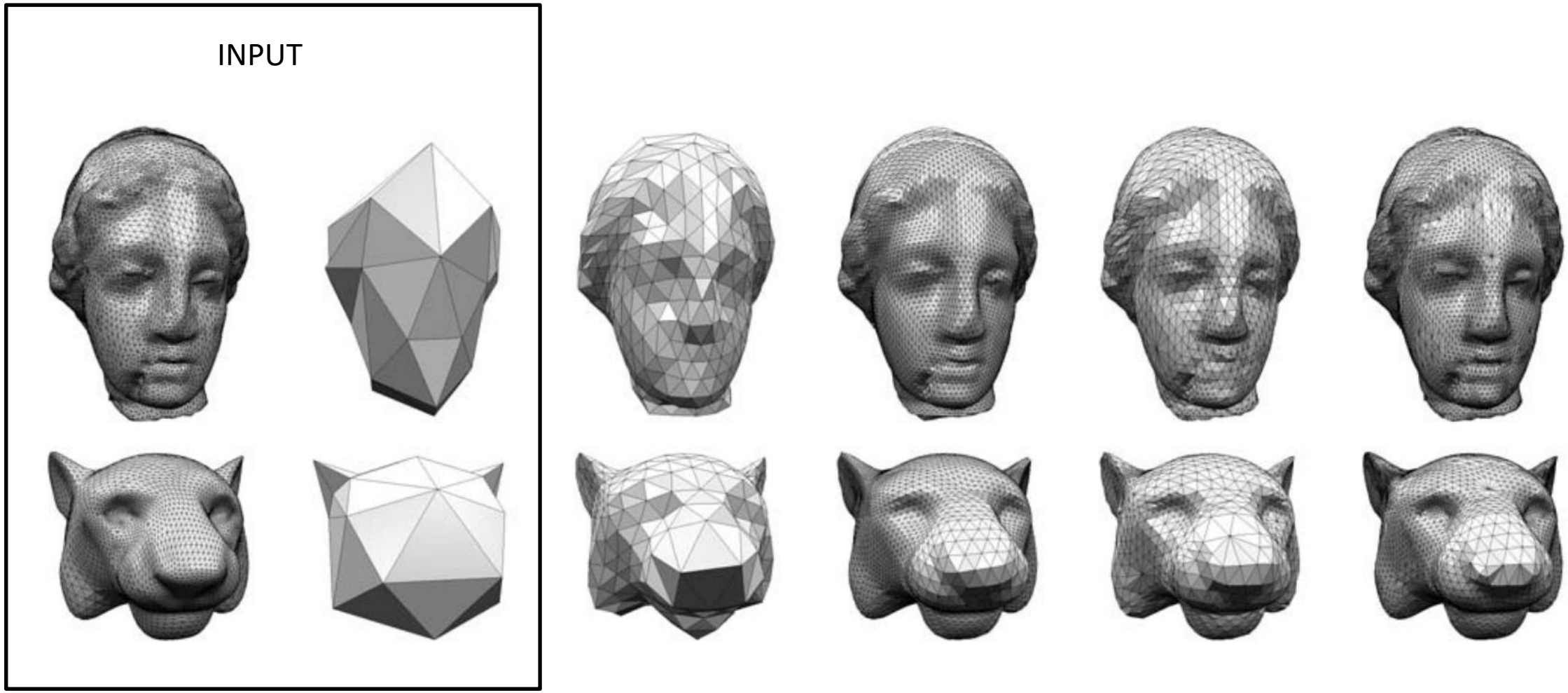
https://en.wikipedia.org/wiki/Morphing

Shape interpolation



Find outline Create vertex mesh Interpolate on meshes

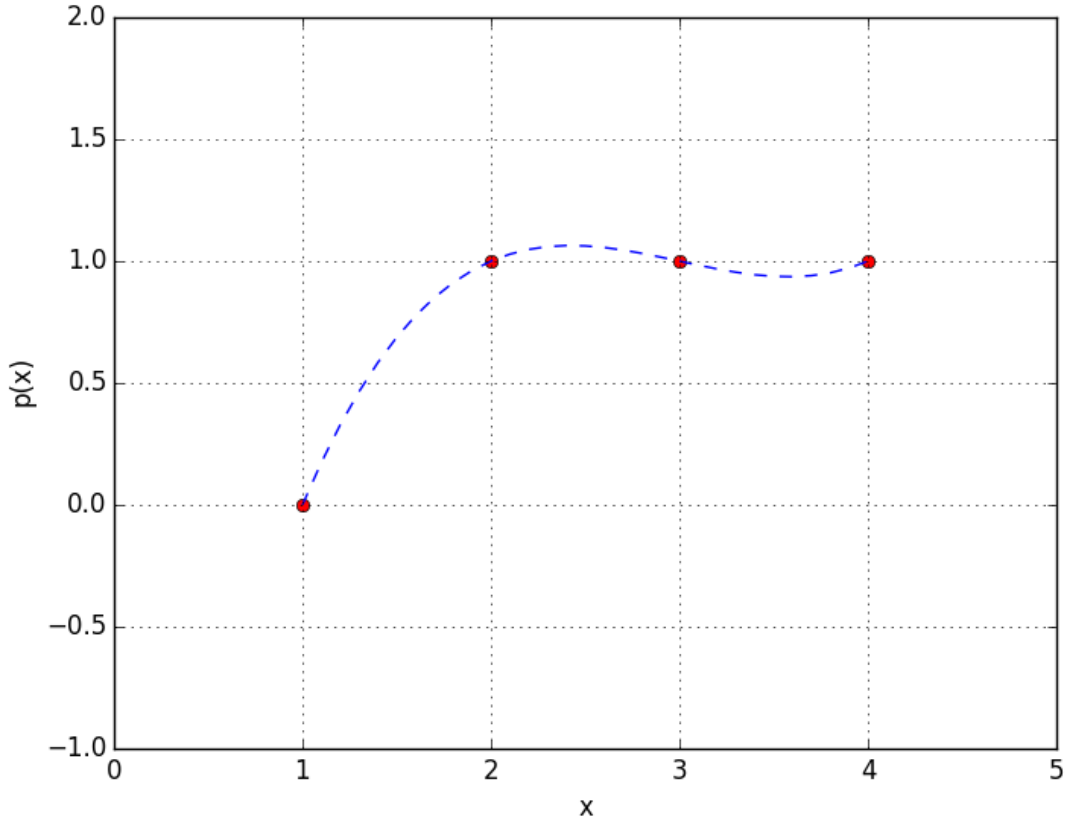
Multi-scale resolution



Lagrange Interpolation

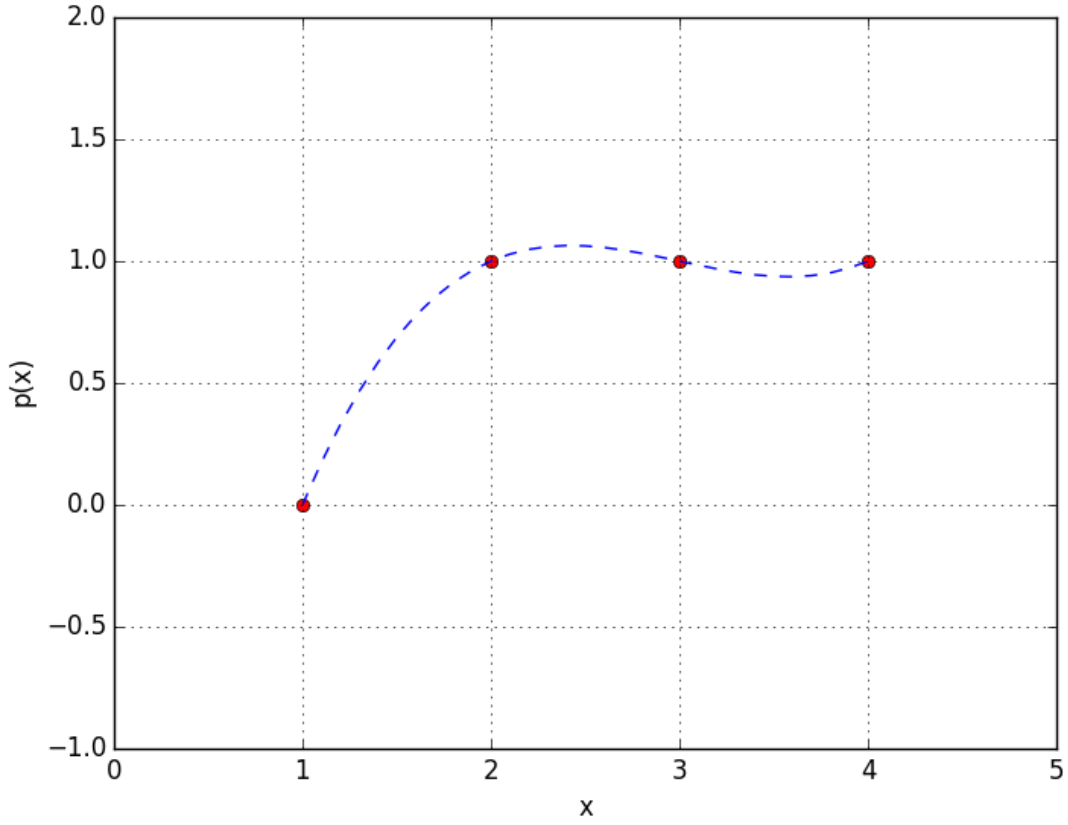
Idea: Create a polynomial basis function for each point that is 1 at that position along the x-axis and 0 at the x-positions of all other points. Then add and scale them together to obtain a polynomial that exactly goes through the specified points.

Example points: (1,0), (2,1), (3,1), (4,1)



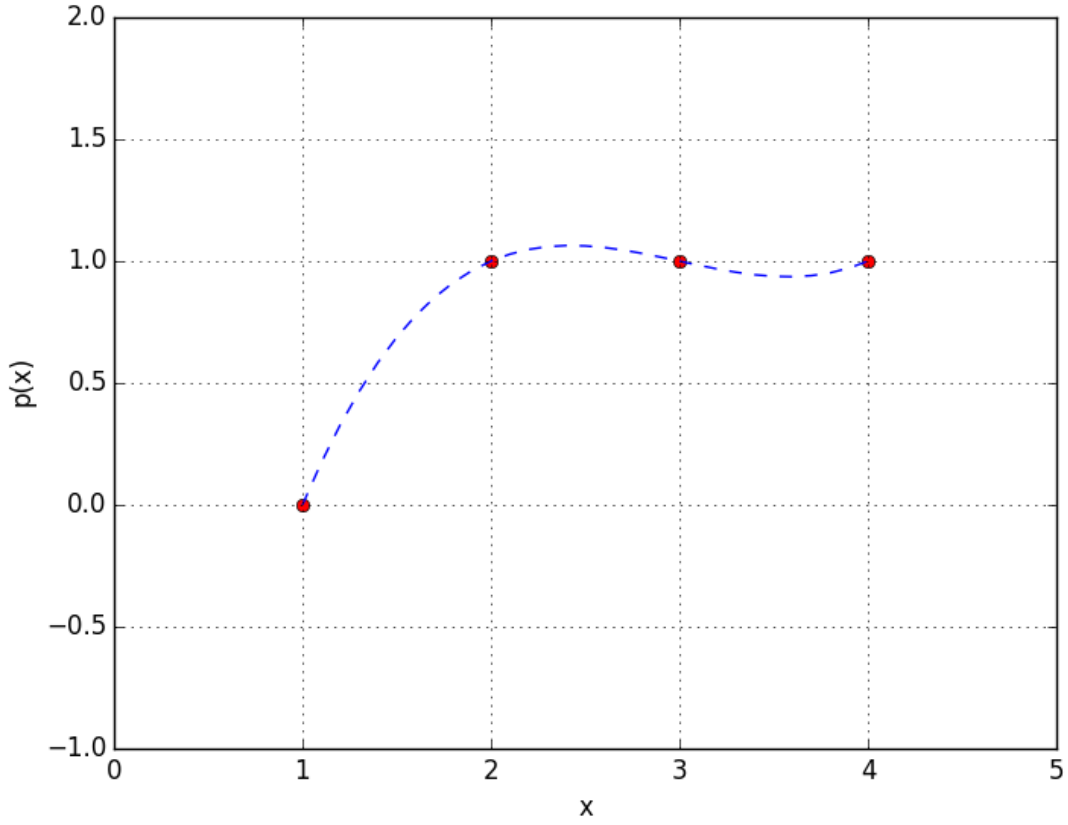
Lagrange Interpolation

$$(x-2)(x-3)(x-4)$$



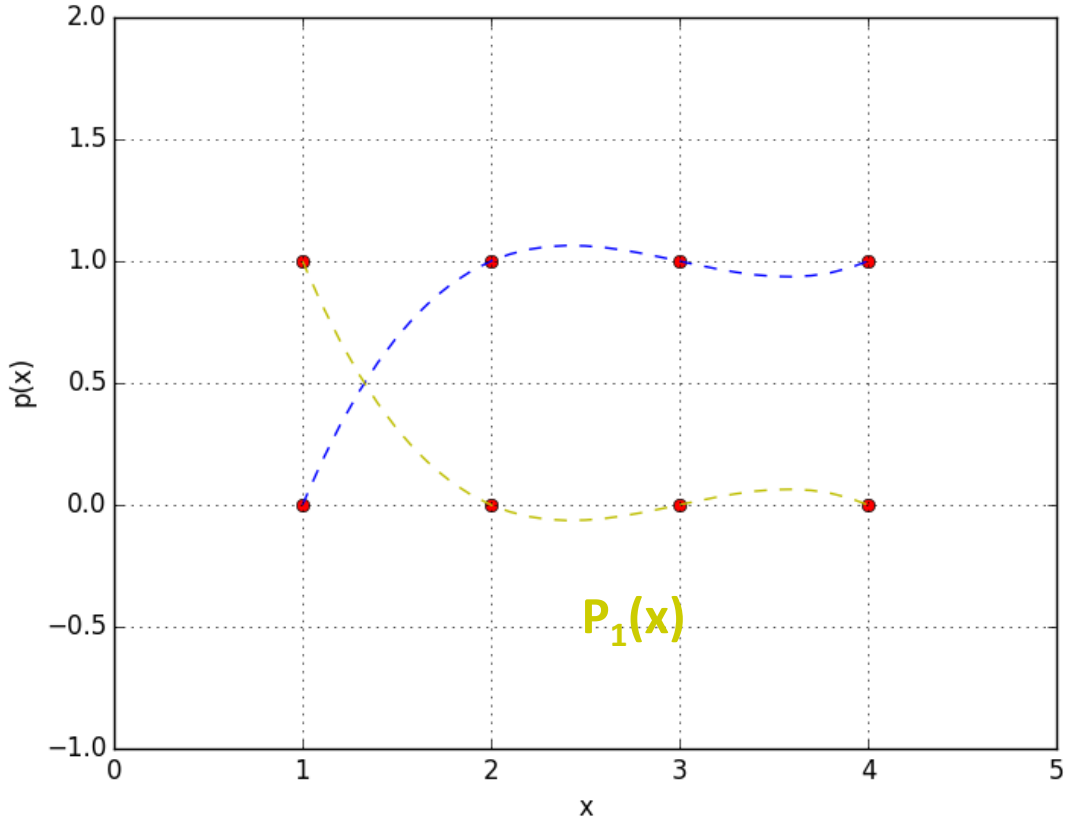
Lagrange Interpolation

$$\frac{(x-2)(x-3)(x-4)}{(1-2)(1-3)(1-4)}$$



Lagrange Interpolation

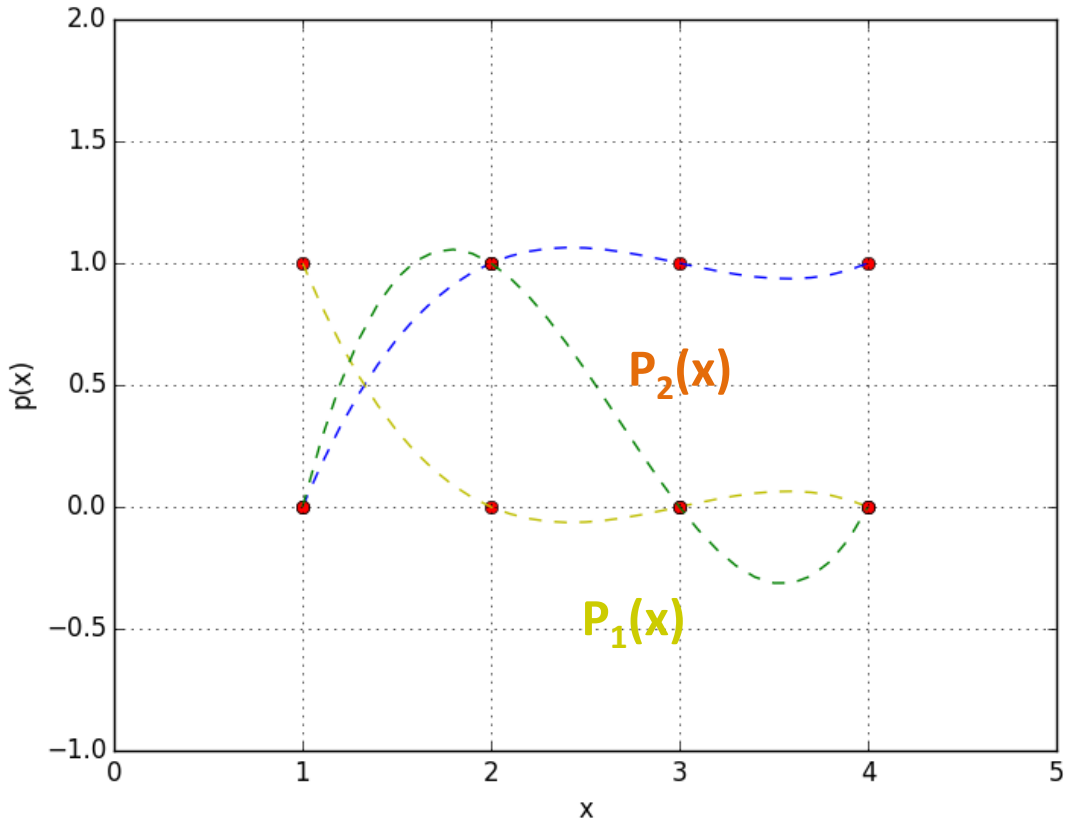
$$P_1(x) = \frac{(x-2)(x-3)(x-4)}{(1-2)(1-3)(1-4)}$$



Lagrange Interpolation

$$P_1(x) = \frac{(x-2)(x-3)(x-4)}{(1-2)(1-3)(1-4)}$$

$$P_2(x) = \frac{(x-1)(x-3)(x-4)}{(2-1)(2-3)(2-4)}$$

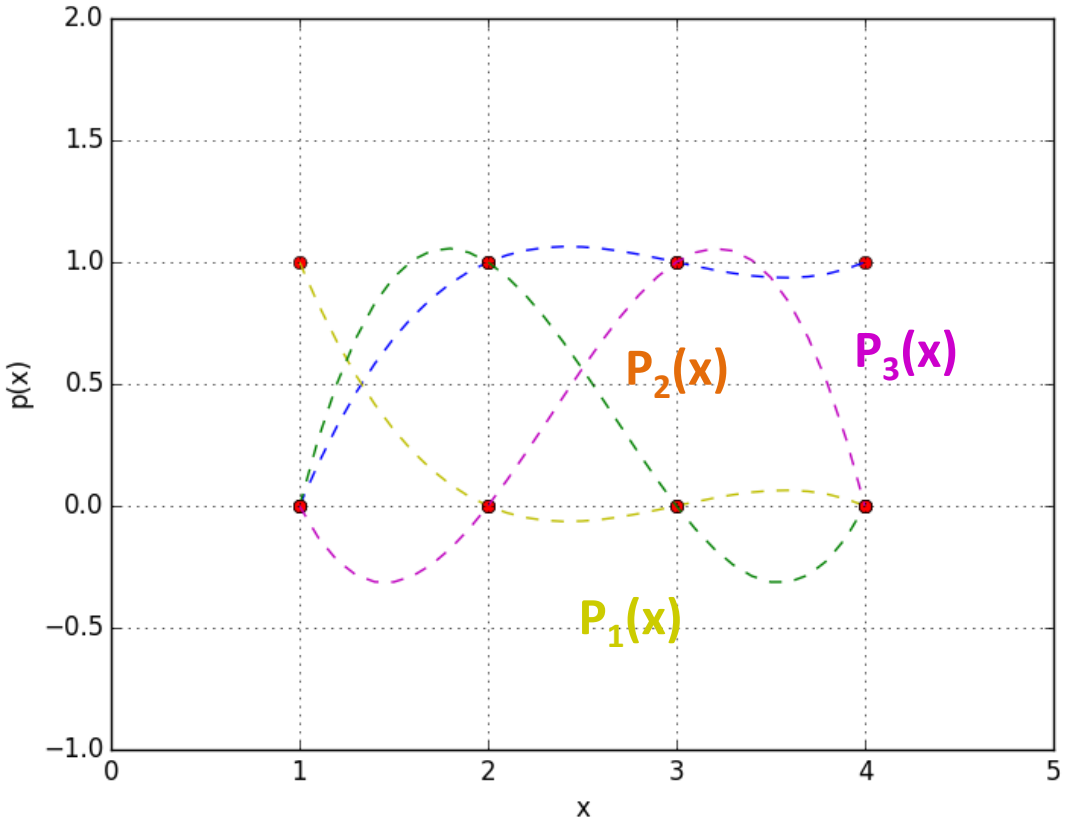


Lagrange Interpolation

$$P_1(x) = \frac{(x-2)(x-3)(x-4)}{(1-2)(1-3)(1-4)}$$

$$P_2(x) = \frac{(x-1)(x-3)(x-4)}{(2-1)(2-3)(2-4)}$$

$$P_3(x) = \frac{(x-1)(x-2)(x-4)}{(3-1)(3-2)(3-4)}$$



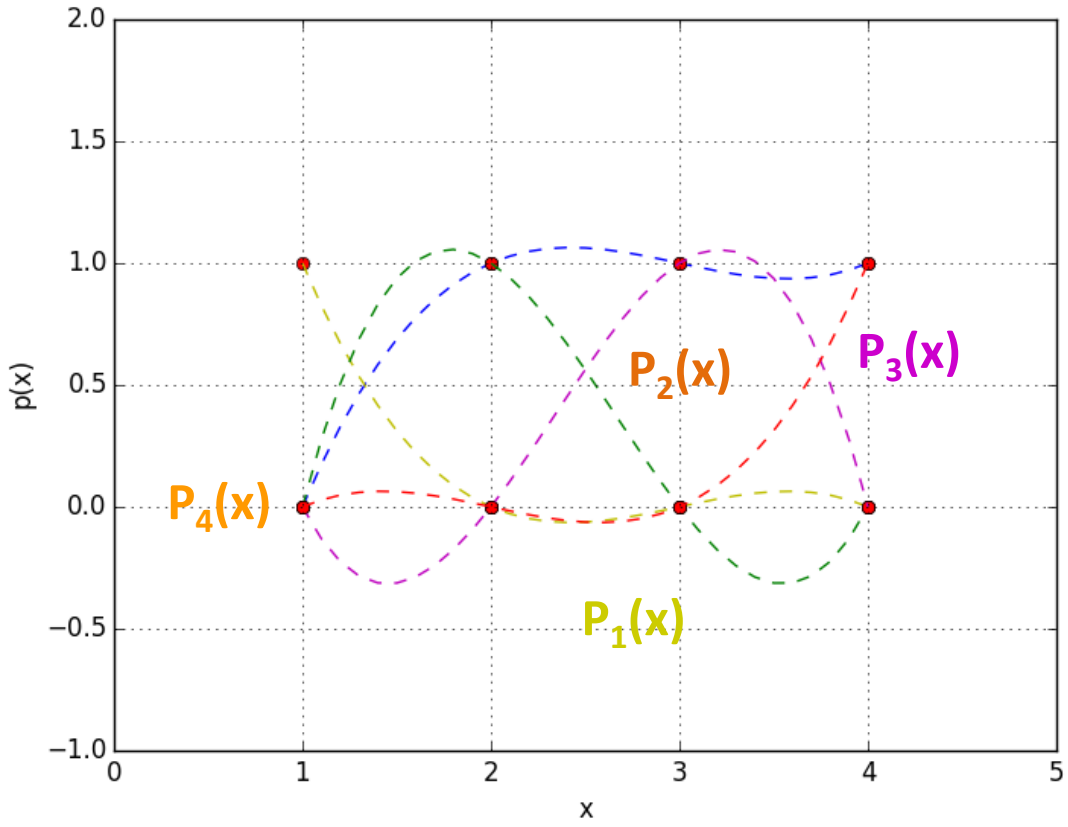
Lagrange Interpolation

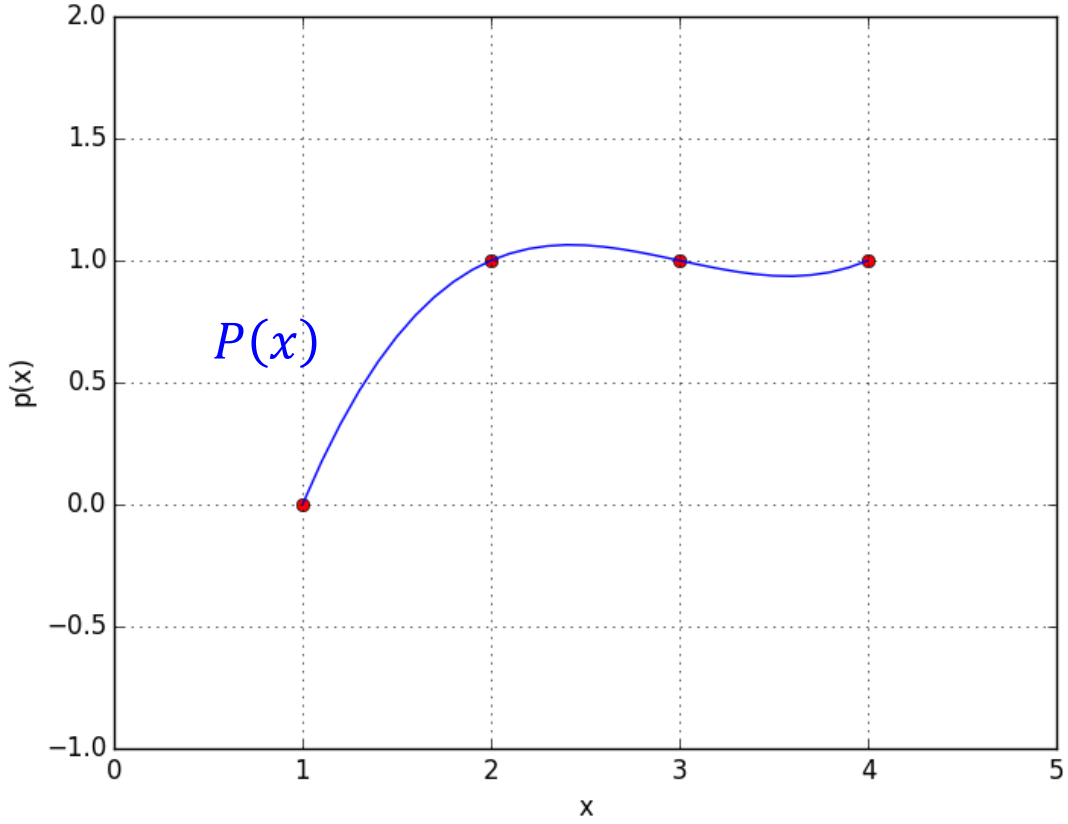
$$P_1(x) = \frac{(x-2)(x-3)(x-4)}{(1-2)(1-3)(1-4)}$$

$$P_2(x) = \frac{(x-1)(x-3)(x-4)}{(2-1)(2-3)(2-4)}$$

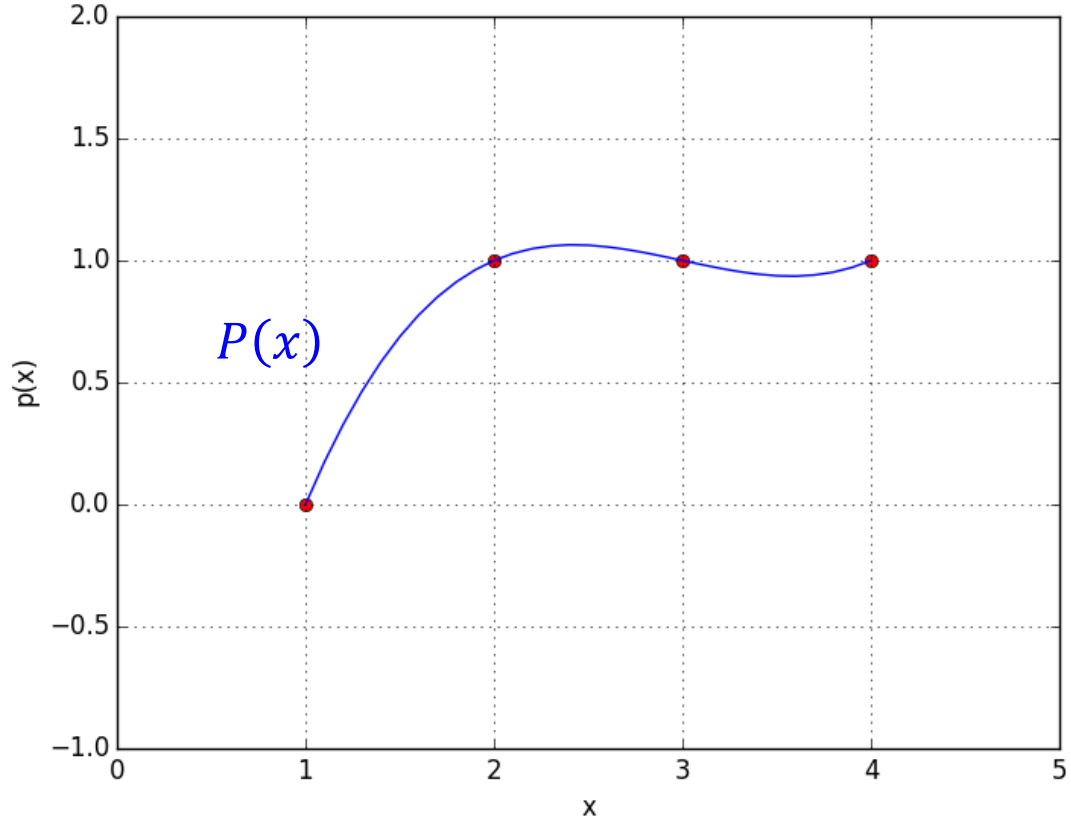
$$P_3(x) = \frac{(x-1)(x-2)(x-4)}{(3-1)(3-2)(3-4)}$$

$$P_4(x) = \frac{(x-1)(x-2)(x-3)}{(4-1)(4-2)(4-3)}$$

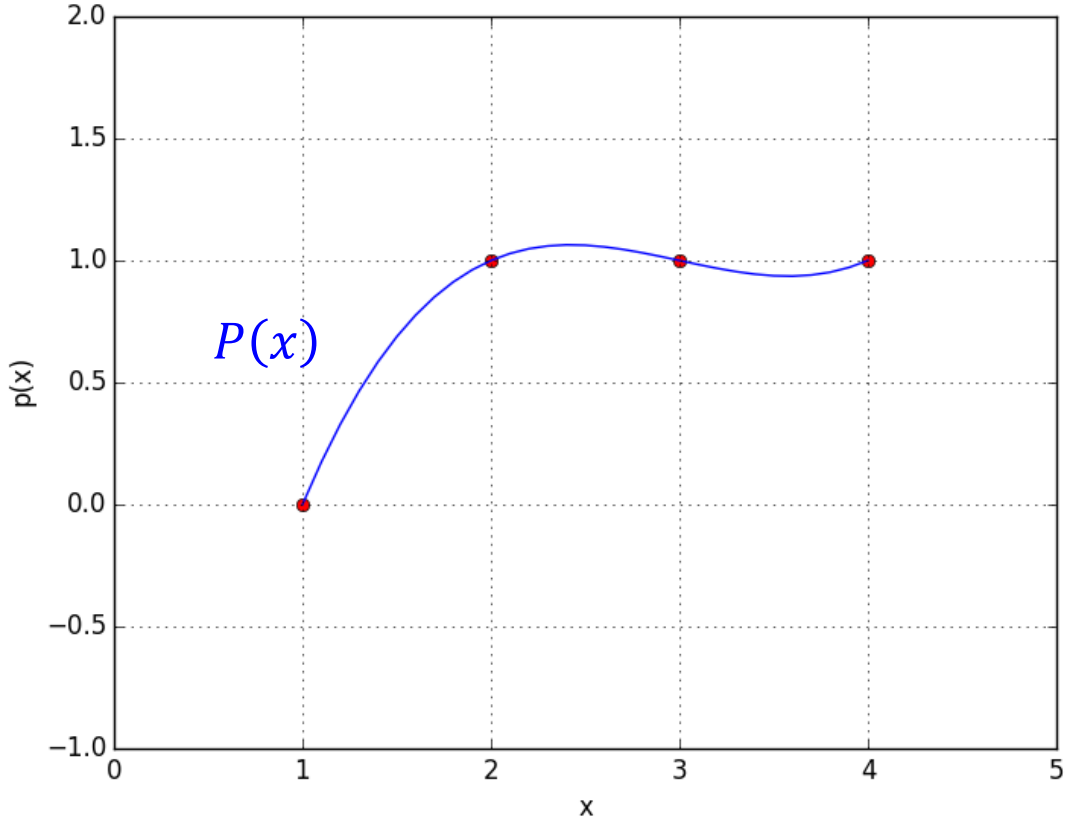




$$P(x) = 0 \cdot P_1(x) + 1 \cdot P_2(x) + 1 \cdot P_3(x) + 1 \cdot P_4(x)$$



$$P(x) = 0 \cdot P_1(x) + 1 \cdot P_2(x) + 1 \cdot P_3(x) + 1 \cdot P_4(x) =$$
$$-3 + \frac{13}{3}\,x - \frac{3}{2}x^2 + \frac{1}{6}x^3$$



$$P(x) = -3 + \frac{13}{3}\,x - \frac{3}{2}x^2 + \frac{1}{6}x^3$$

Polynomial Interpolation

$$y = a_0 + a_1 x^1 + a_2 x^2 + \cdots + a_{n-1} x^{n-1}$$

Polynomial Interpolation

$$
\begin{aligned}
a_o + a_1 x_0 + a_2 x_0^2 + \cdots + a_{n-1} x_0^{n-1} + a_n x_0^n &= y_0 \\
a_o + a_1 x_1 + a_2 x_1^2 + \cdots + a_{n-1} x_1^{n-1} + a_n x_1^n &= y_1 \\
&\vdots \\
a_o + a_1 x_n + a_2 x_n^2 + \cdots + a_{n-1} x_n^{n-1} + a_n x_n^n &= y_n
\end{aligned}
$$

Vandermonde Matrix

$$\begin{bmatrix} x_0^n & x_0^{n-1} & x_0^{n-2} & \dots & x_0 & 1 \\ x_1^n & x_1^{n-1} & x_1^{n-2} & \dots & x_1 & 1 \\ \vdots & \vdots & \vdots & & \vdots & \vdots \\ x_n^n & x_n^{n-1} & x_n^{n-2} & \dots & x_n & 1 \end{bmatrix} \begin{bmatrix} a_n \\ a_{n-1} \\ \vdots \\ a_0 \end{bmatrix} = \begin{bmatrix} y_0 \\ y_1 \\ \vdots \\ y_n \end{bmatrix}$$

Polynomial interpolation

numpy.polynomial.polynomial.polyvander

`numpy.polynomial.polynomial.` **`polyvander`**(*x*, *deg*) [source]

Vandermonde matrix of given degree.

Returns the Vandermonde matrix of degree *deg* and sample points *x*. The Vandermonde matrix is defined by

$$V[..., i] = x^i,$$

where *0 <= i <= deg*. The leading indices of *V* index the elements of *x* and the last index is the power of *x*.

If *c* is a 1-D array of coefficients of length *n + 1* and *V* is the matrix `V = polyvander(x, n)`, then `np.dot(V, c)` and `polyval(x, c)` are the same up to roundoff. This equivalence is useful both for least squares fitting and for the evaluation of a large number of polynomials of the same degree and sample points.

Parameters:

x : *array_like*
: Array of points. The dtype is converted to float64 or complex128 depending on whether any of the elements are complex. If *x* is scalar it is converted to a 1-D array.

deg : *int*
: Degree of the resulting matrix.

Returns:

vander : *ndarray.*
: The Vandermonde matrix. The shape of the returned matrix is `x.shape + (deg + 1,)`, where the last index is the power of *x*. The dtype will be the same as the converted *x*.

See also:

`polyvander2d`, `polyvander3d`

numpy.poly1d¶

class numpy.poly1d(*c_or_r*, *r=0*, *variable=None*) [source]

A one-dimensional polynomial class.

A convenience class, used to encapsulate "natural" operations on polynomials so that said operations may take on their customary form in code (see Examples).

```
In [5]: p = np.poly1d([1,2,3])

In [6]: print np.poly1d(p)
   2
1 x + 2 x + 3
```

- Arithmetic operations (+,-,/,*) work on polynomials

numpy.poly1d¶

class numpy.poly1d(*c_or_r*, *r=0*, *variable=None*) [source]

A one-dimensional polynomial class.

A convenience class, used to encapsulate "natural" operations on polynomials so that said operations may take on their customary form in code (see Examples).

```
In [5]: p = np.poly1d([1,2,3])

In [6]: print np.poly1d(p)
   2
1 x + 2 x + 3

In [7]: p(1)
Out[7]: 6
```

- Arithmetic operations (+,-,/,*) work on polynomials

numpy.poly1d¶

class numpy.poly1d(*c_or_r*, *r=0*, *variable=None*) [source]

A one-dimensional polynomial class.

A convenience class, used to encapsulate "natural" operations on polynomials so that said operations may take on their customary form in code (see Examples).

- Arithmetic operations (+,-,/,*) work on polynomials

```
In [5]: p = np.poly1d([1,2,3])

In [6]: print np.poly1d(p)
   2
1 x + 2 x + 3

In [7]: p(1)
Out[7]: 6

In [8]: p(2)
Out[8]: 11
```

numpy.poly1d¶

class numpy.poly1d(*c_or_r*, *r=0*, *variable=None*) [source]

A one-dimensional polynomial class.

A convenience class, used to encapsulate "natural" operations on polynomials so that said operations may take on their customary form in code (see Examples).

- Arithmetic operations (+,-,/,*) work on polynomials

```
In [5]: p = np.poly1d([1,2,3])

In [6]: print np.poly1d(p)
   2
1 x + 2 x + 3

In [7]: p(1)
Out[7]: 6

In [8]: p(2)
Out[8]: 11

In [9]: p(1.5)
Out[9]: 8.25
```

numpy.poly1d¶

class numpy.poly1d(*c_or_r*, *r=0*, *variable=None*) [source]

A one-dimensional polynomial class.

A convenience class, used to encapsulate "natural" operations on polynomials so that said operations may take on their customary form in code (see Examples).

- Arithmetic operations (+,-,/,*) work on polynomials

```
In [5]: p = np.poly1d([1,2,3])

In [6]: print np.poly1d(p)
   2
1 x + 2 x + 3

In [7]: p(1)
Out[7]: 6

In [8]: p(2)
Out[8]: 11

In [9]: p(1.5)
Out[9]: 8.25

In [10]: print p.deriv(m=1)

2 x + 2
```

numpy.poly1d¶

class numpy.poly1d(*c_or_r*, *r=0*, *variable=None*) [source]

A one-dimensional polynomial class.

A convenience class, used to encapsulate "natural" operations on polynomials so that said operations may take on their customary form in code (see Examples).

- Arithmetic operations (+,-,/,*) work on polynomials

```
In [5]: p = np.poly1d([1,2,3])

In [6]: print np.poly1d(p)
   2
1 x + 2 x + 3

In [7]: p(1)
Out[7]: 6

In [8]: p(2)
Out[8]: 11

In [9]: p(1.5)
Out[9]: 8.25

In [10]: print p.deriv(m=1)

2 x + 2

In [11]: print p.integ(m=2)
          4          3       2
0.08333 x + 0.3333 x + 1.5 x
```

numpy.poly1d¶

class numpy.poly1d(*c_or_r*, *r=0*, *variable=None*) [source]

A one-dimensional polynomial class.

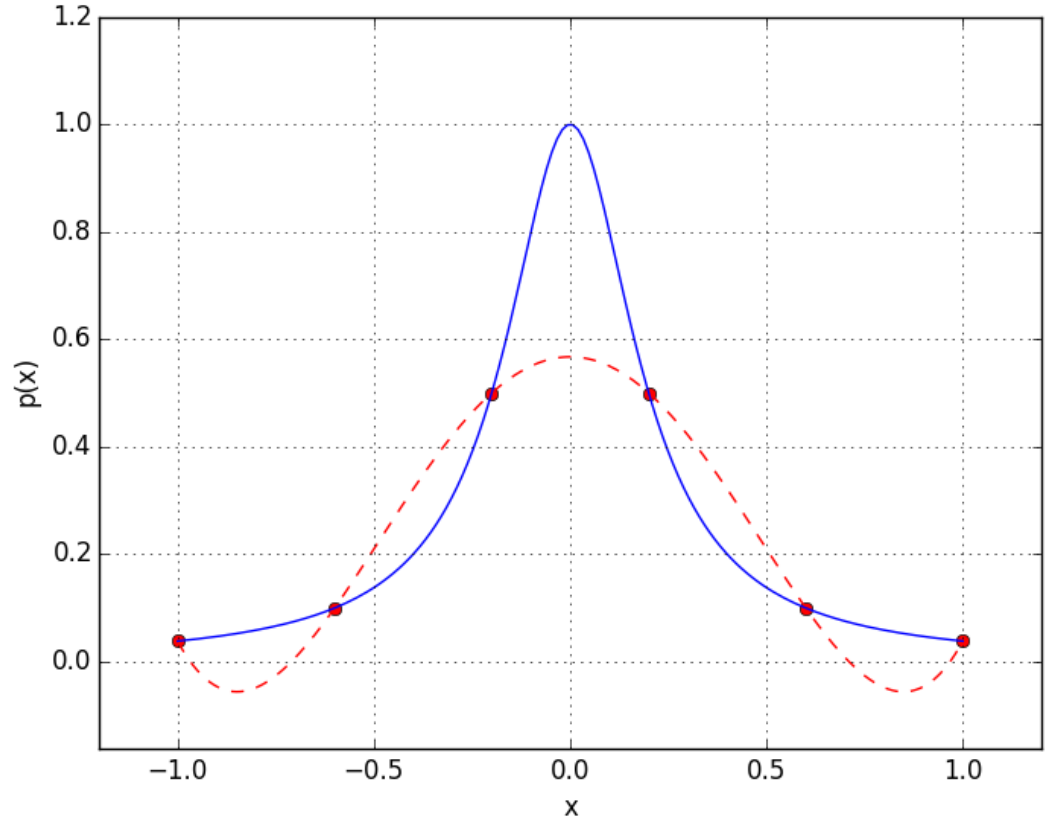
A convenience class, used to encapsulate "natural" operations on polynomials so that said operations may take on their customary form in code (see Examples).

- Arithmetic operations (+,-,/,*) work on polynomials

```
In [5]: p = np.poly1d([1,2,3])

In [6]: print np.poly1d(p)
   2
1 x + 2 x + 3

In [7]: p(1)
Out[7]: 6

In [8]: p(2)
Out[8]: 11

In [9]: p(1.5)
Out[9]: 8.25

In [10]: print p.deriv(m=1)

2 x + 2

In [11]: print p.integ(m=2)
          4          3       2
0.08333 x + 0.3333 x + 1.5 x

In [12]: print p.roots
[-1.+1.41421356j -1.-1.41421356j]
```

Example Lagrange

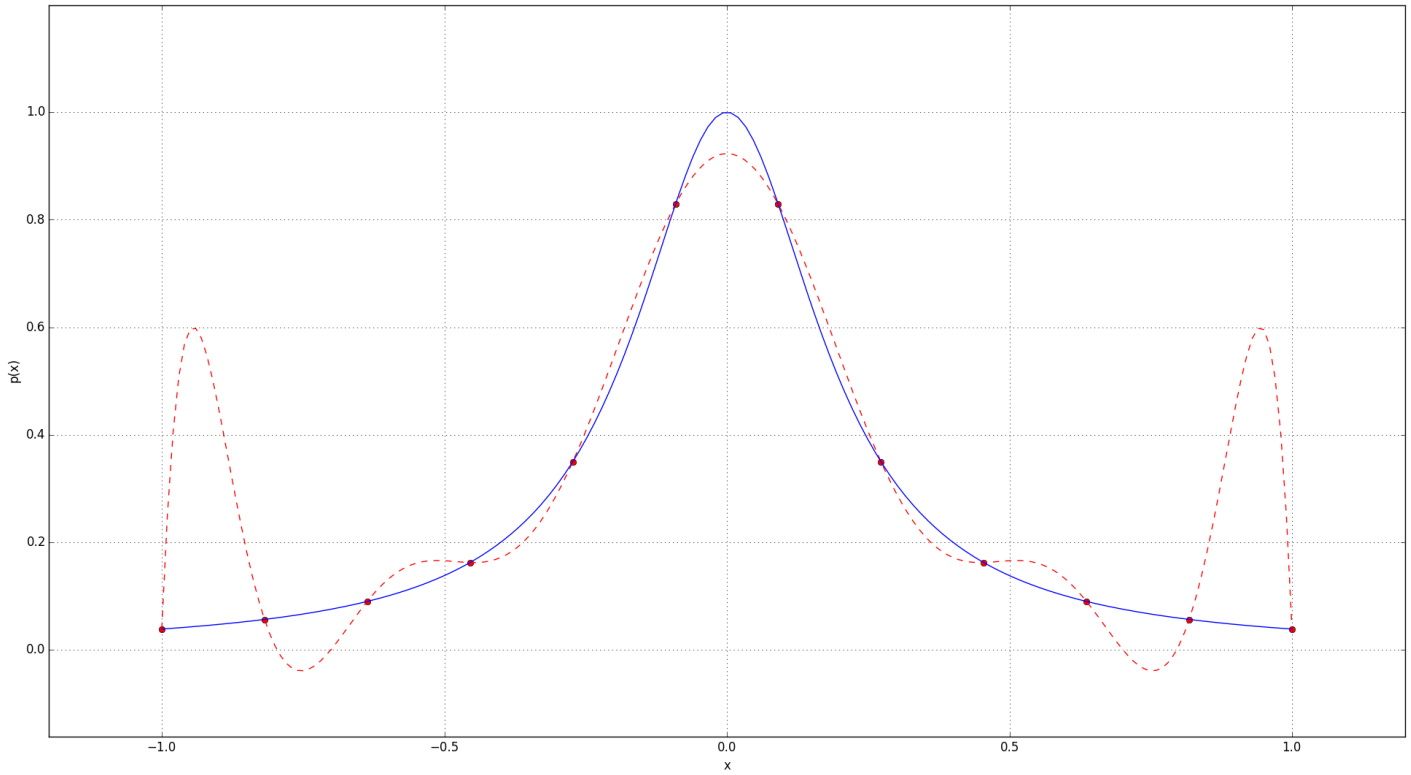
$$f(x) = \frac{1}{1 + 25x^2}$$



6 points

Overfitting

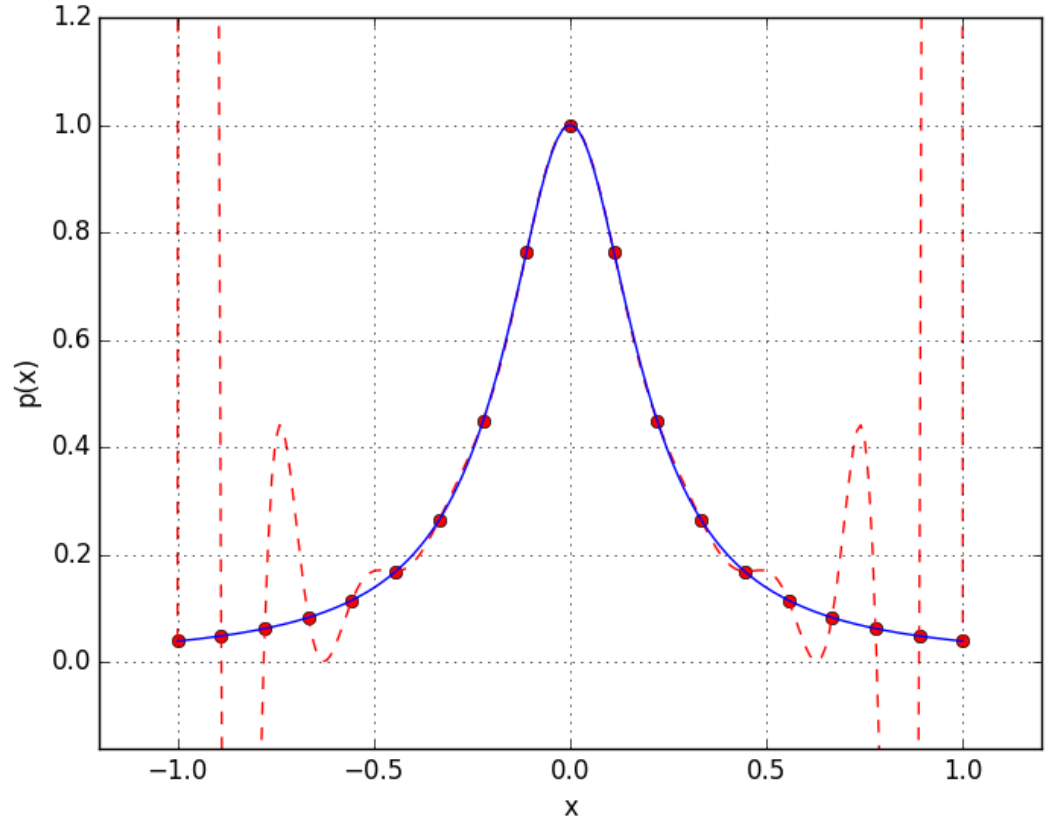
$$f(x) = \frac{1}{1 + 25x^2}$$



12 points

Overfitting

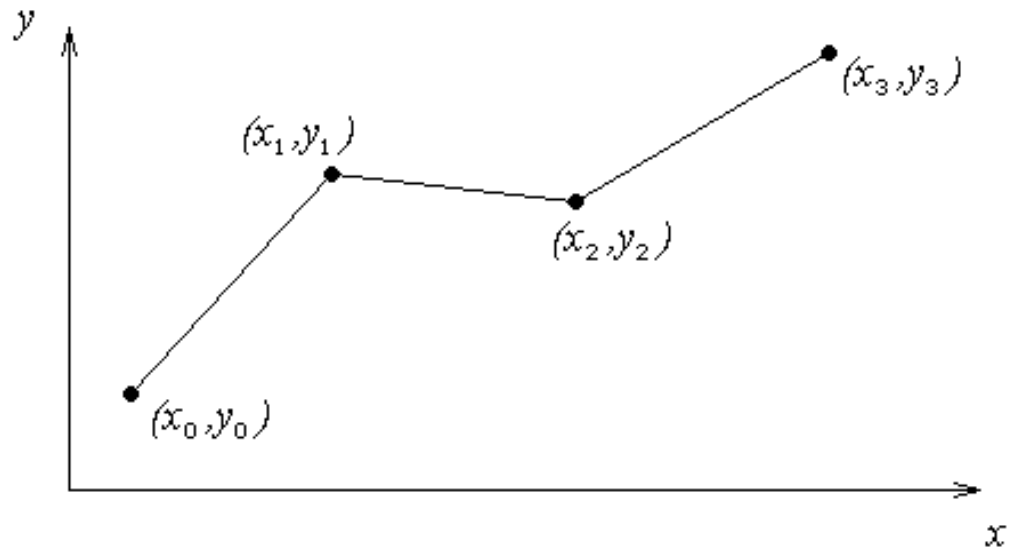
$$f(x) = \frac{1}{1 + 25x^2}$$



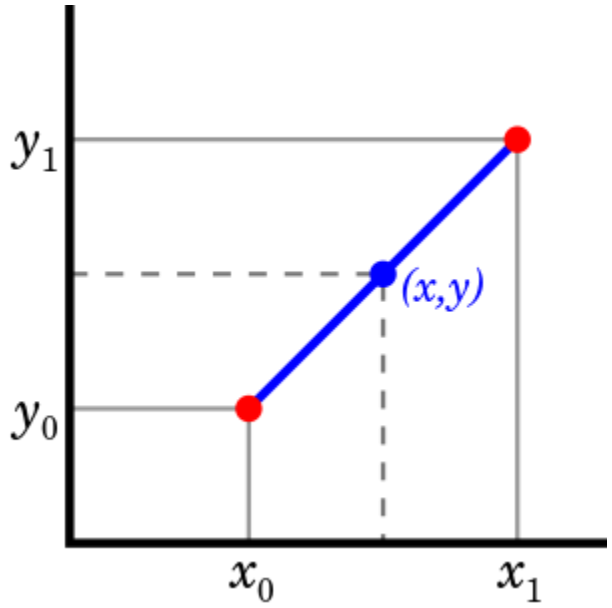
20 points

Linear Spline Interpolation

- For given points (x_0, y_0), (x_1, y_1), … (x_n, y_n) fit spline functions. We create lines that connect consecutive points. Linear splines are defined by $y_i = f(x_i)$



Linear Interpolation

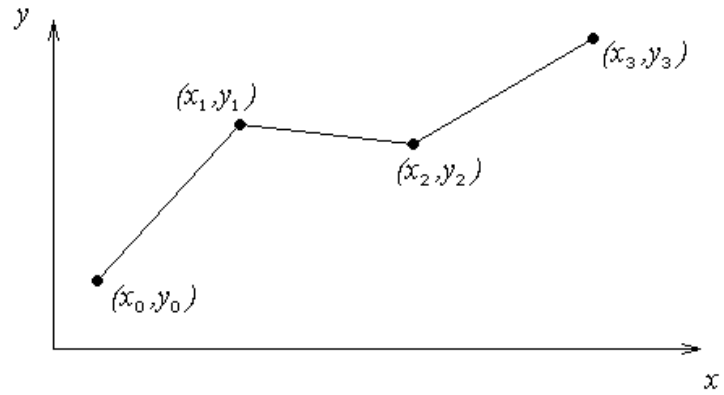


$$\frac{y - y_0}{x - x_0} = \frac{y_1 - y_0}{x_1 - x_0}$$

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

Linear Interpolation

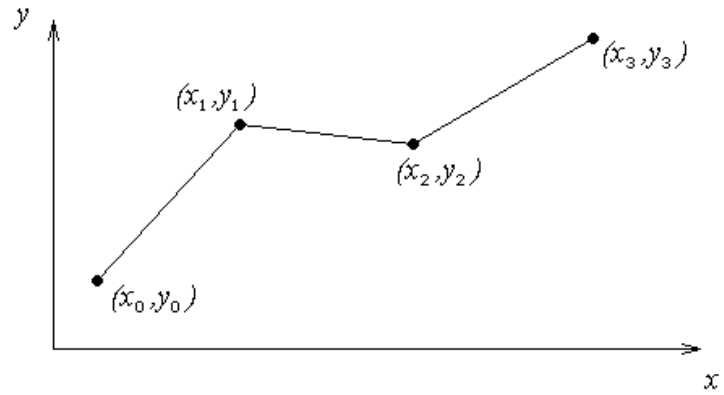
$$f(x) = f(x_0) + \frac{f(x_1) - f(x_0)}{x_1 - x_0}(x - x_0), \qquad x_0 \le x \le x_1$$



Linear Interpolation

$$f(x) = f(x_0) + \frac{f(x_1) - f(x_0)}{x_1 - x_0}(x - x_0), \qquad x_0 \le x \le x_1$$

$$= f(x_1) + \frac{f(x_2) - f(x_1)}{x_2 - x_1}(x - x_1), \qquad x_1 \le x \le x_2$$



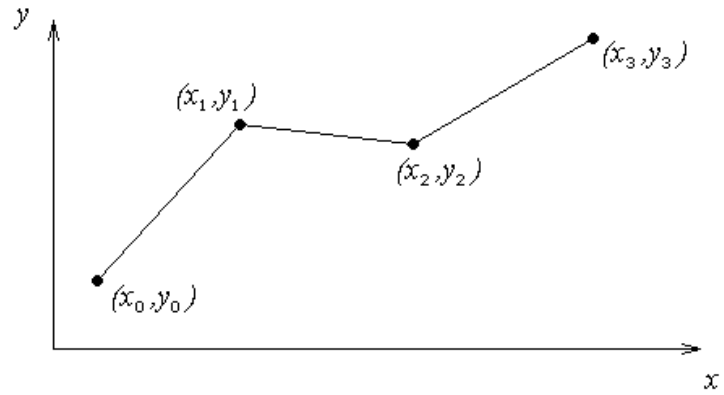
Linear Interpolation – Spline Interpolation

$$f(x) = f(x_0) + \frac{f(x_1) - f(x_0)}{x_1 - x_0}(x - x_0), \qquad x_0 \le x \le x_1$$

$$= f(x_1) + \frac{f(x_2) - f(x_1)}{x_2 - x_1}(x - x_1), \qquad x_1 \le x \le x_2$$

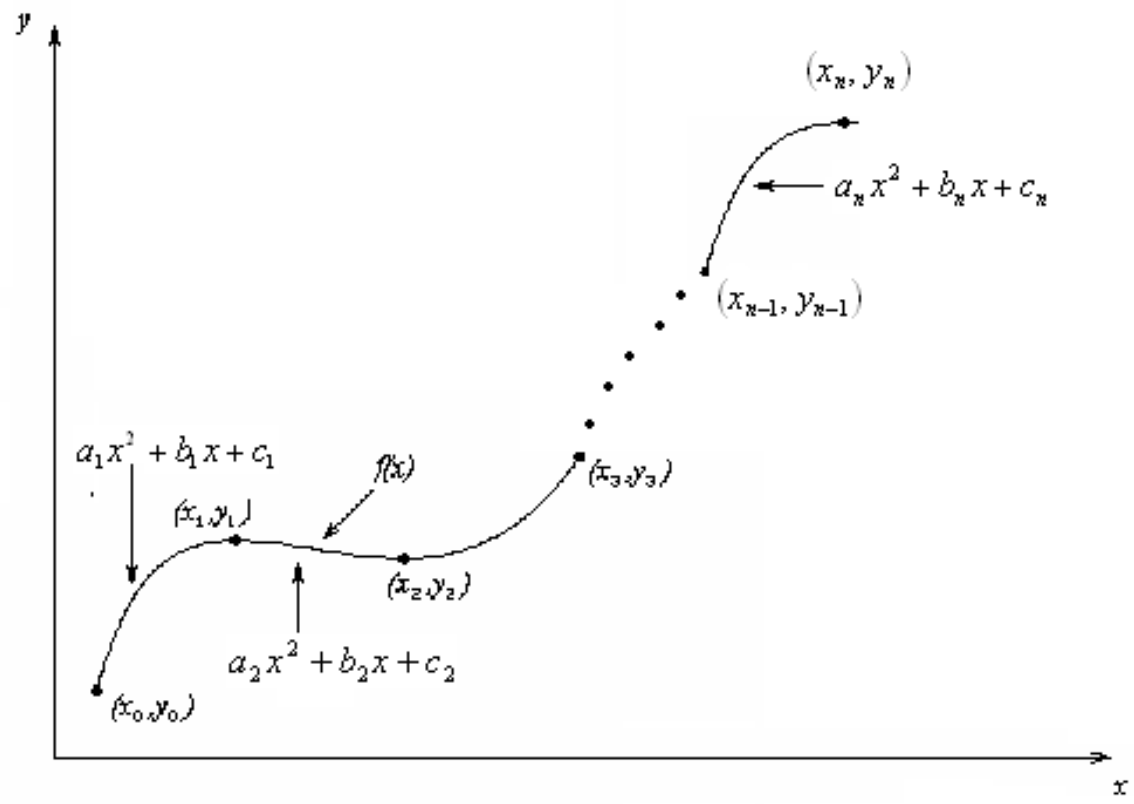
$$\vdots$$

$$= f(x_{n-1}) + \frac{f(x_n) - f(x_{n-1})}{x_n - x_{n-1}}(x - x_{n-1}), \quad x_{n-1} \le x \le x_n$$



Quadratic spline interpolation

$$
\begin{aligned}
f(x) &= a_1 x^2 + b_1 x + c_1, && x_0 \le x \le x_1 \\
&= a_2 x^2 + b_2 x + c_2, && x_1 \le x \le x_2 \\
&\;\;\vdots \\
&= a_n x^2 + b_n x + c_n, && x_{n-1} \le x \le x_n
\end{aligned}
$$



Python Interpolation

scipy.interpolate

- **1D and 2D interpolation**
  - Creates an interpolating **function** from a set of of points
  - The created function returns a scalar value for an array of data.

scipy.interpolate.interp1d

class **scipy.interpolate.interp1d**(*x*, *y*, *kind='linear'*, *axis=-1*, *copy=True*, *bounds_error=True*, *fill_value=np.nan*, *assume_sorted=False*) [source]

Interpolate a 1-D function.

x and *y* are arrays of values used to approximate some function f: `y = f(x)`. This class returns a function whose call method uses interpolation to find the value of new points.

Parameters:

x : *(N,) array_like*
A 1-D array of real values.

y : *(...,N,...) array_like*
A N-D array of real values. The length of *y* along the interpolation axis must be equal to the length of *x*.

kind : *str or int, optional*
Specifies the kind of interpolation as a string ('linear', 'nearest', 'zero', 'slinear', 'quadratic, 'cubic' where 'slinear', 'quadratic' and 'cubic' refer to a spline interpolation of first, second or third order) or as an integer specifying the order of the spline interpolator to use. Default is 'linear'.

axis : *int, optional*
Specifies the axis of *y* along which to interpolate. Interpolation defaults to the last axis of *y*.

copy : *bool, optional*
If True, the class makes internal copies of x and y. If False, references to *x* and *y* are used. The default is to copy.

bounds_error : *bool, optional*
If True, a ValueError is raised any time interpolation is attempted on a value outside of the range of x (where extrapolation is necessary). If False, out of bounds values are assigned *fill_value*. By default, an error is raised.

fill_value : *float, optional*
If provided, then this value will be used to fill in for requested points outside of the data range. If not provided, then the default is NaN.

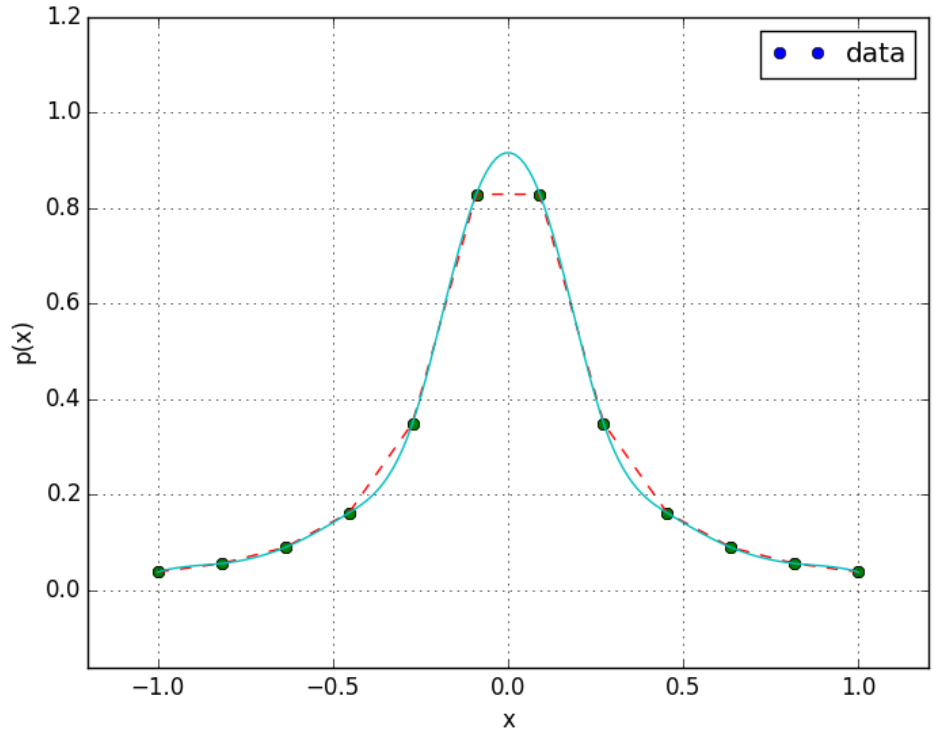
assume_sorted : *bool, optional*
If False, values of *x* can be in any order and they are sorted first. If True, *x* has to be an array of monotonically increasing values.

Examples

```
>>> from scipy import interpolate
>>> x = np.arange(0, 10)
>>> y = np.exp(-x/3.0)
>>> f = interpolate.interp1d(x, y)
```

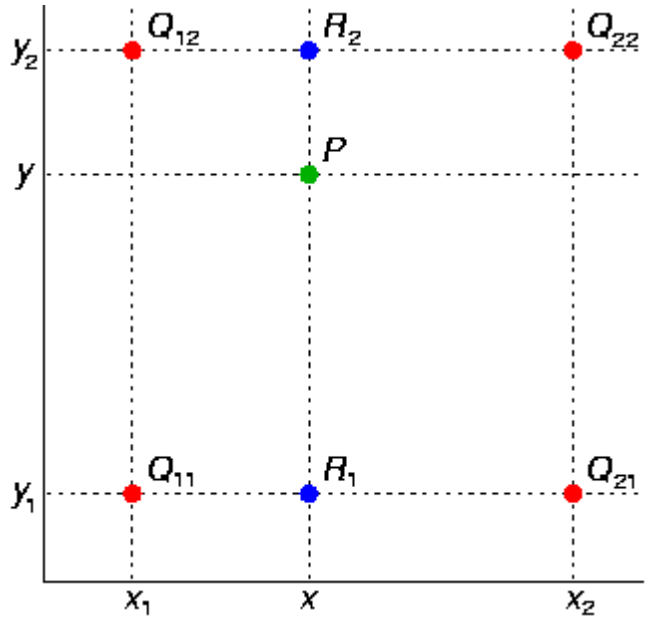
```
>>> xnew = np.arange(0,9, 0.1)
>>> ynew = f(xnew)   # use interpolation function returned by `interp1d`
>>> plt.plot(x, y, 'o', xnew, ynew, '-')
>>> plt.show()
```

Example linear/cubic spline interpolation



Bilinear interpolation

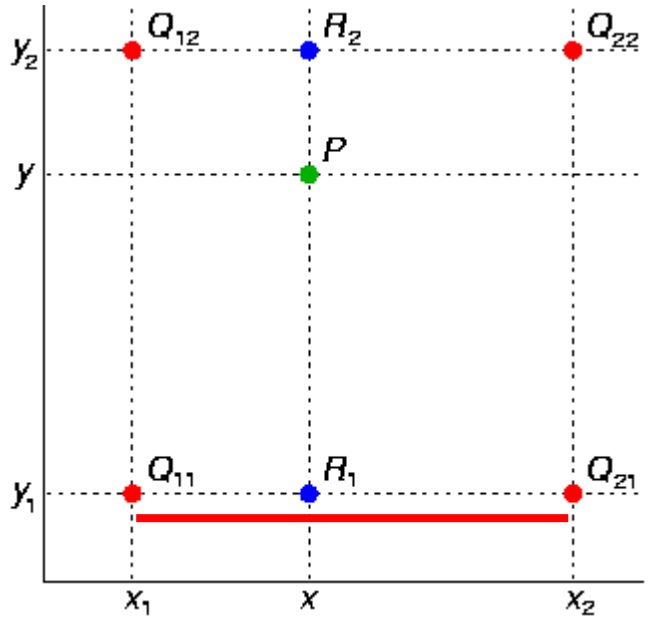
- Interpolating in 2D?
  - First interpolate in respect to x, then in respect to y



http://en.wikipedia.org/wiki/Bilinear_interpolation

Bilinear interpolation

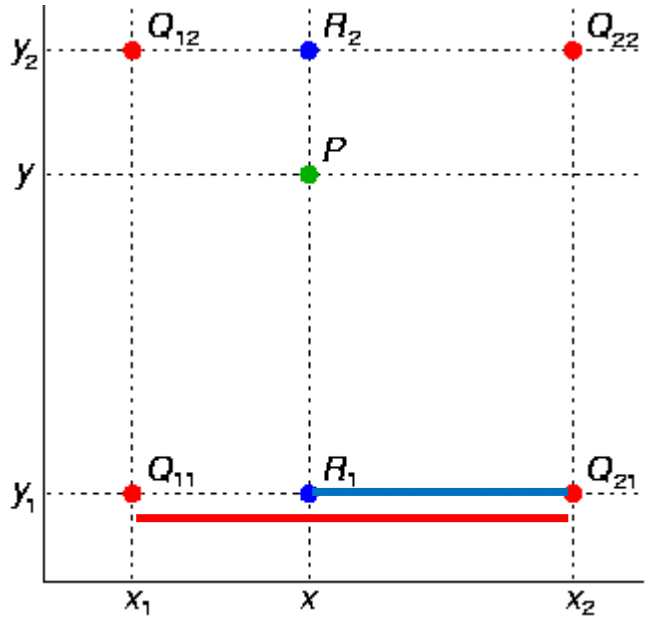
$$f(x,y) \approx \frac{f(Q_{11})}{(x_2-x_1)(y_2-y_1)}(x_2-x)(y_2-y)$$



http://en.wikipedia.org/wiki/Bilinear_interpolation

Bilinear interpolation

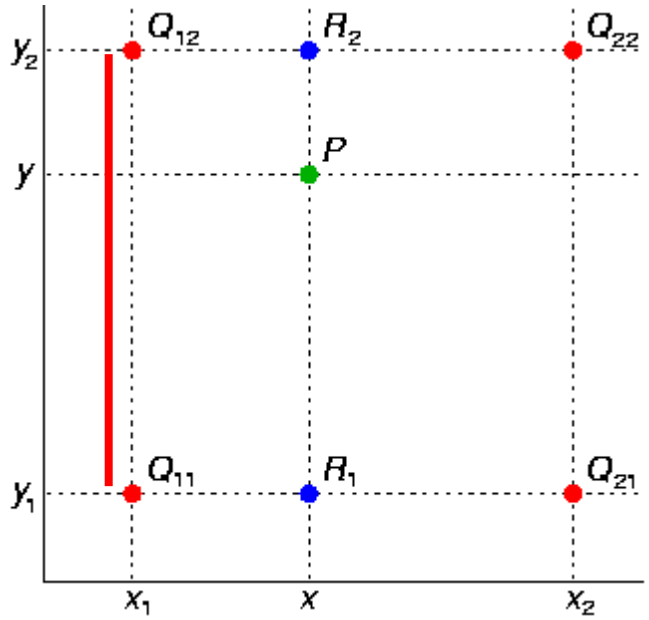
$$f(x, y) \approx \frac{f(Q_{11})}{(x_2 - x_1)(y_2 - y_1)}(x_2 - x)(y_2 - y)$$



http://en.wikipedia.org/wiki/Bilinear_interpolation

Bilinear interpolation

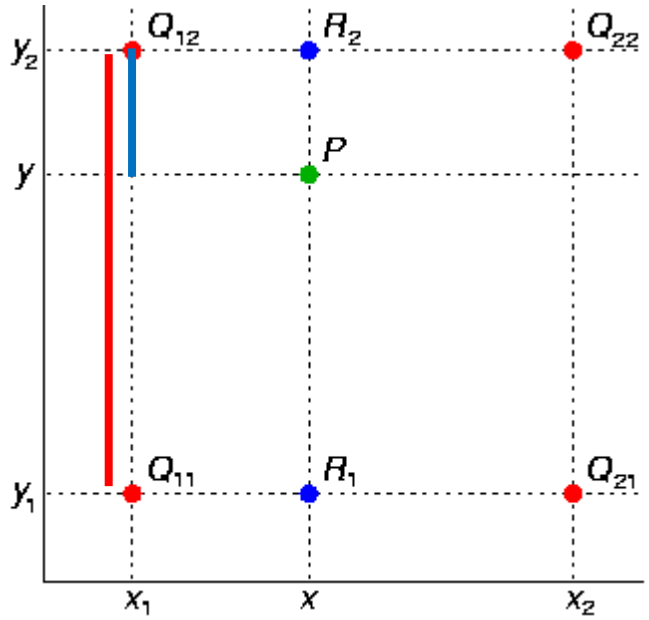
$$f(x,y) \approx \frac{f(Q_{11})}{(x_2-x_1)(y_2-y_1)}(x_2-x)(y_2-y)$$



http://en.wikipedia.org/wiki/Bilinear_interpolation

Bilinear interpolation

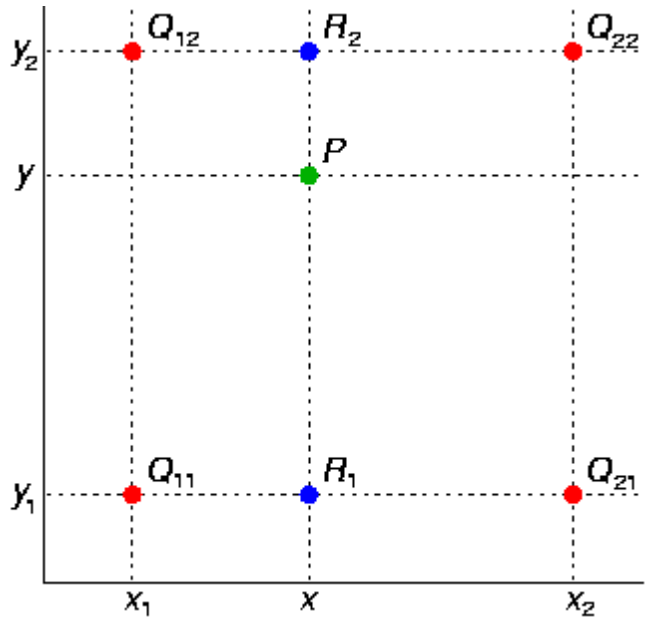
$$f(x,y) \approx \frac{f(Q_{11})}{(x_2-x_1)(y_2-y_1)}(x_2-x)(y_2-y)$$



http://en.wikipedia.org/wiki/Bilinear_interpolation

Bilinear interpolation

$$
\begin{aligned}
f(x, y) \approx & \frac{f(Q_{11})}{(x_2 - x_1)(y_2 - y_1)}(x_2 - x)(y_2 - y) \\
& + \frac{f(Q_{21})}{(x_2 - x_1)(y_2 - y_1)}(x - x_1)(y_2 - y) \\
& + \frac{f(Q_{12})}{(x_2 - x_1)(y_2 - y_1)}(x_2 - x)(y - y_1) \\
& + \frac{f(Q_{22})}{(x_2 - x_1)(y_2 - y_1)}(x - x_1)(y - y_1).
\end{aligned}
$$



http://en.wikipedia.org/wiki/Bilinear_interpolation

scipy.interpolate.interp2d

class **scipy.interpolate.interp2d**(*x*, *y*, *z*, *kind='linear'*, *copy=True*, *bounds_error=False*, *fill_value=nan*)

Interpolate over a 2-D grid. [source]

x, *y* and *z* are arrays of values used to approximate some function f: `z = f(x, y)`. This class returns a function whose call method uses spline interpolation to find the value of new points.

If *x* and *y* represent a regular grid, consider using RectBivariateSpline.

Parameters:

x, y *: array_like*

Arrays defining the data point coordinates.

If the points lie on a regular grid, *x* can specify the column coordinates and *y* the row coordinates, for example:

```
>>> x = [0,1,2];  y = [0,3]; z = [[1,2,3], [4,5,6]]
```

Otherwise, *x* and *y* must specify the full coordinates for each point, for example:

```
>>> x = [0,1,2,0,1,2];  y = [0,0,0,3,3,3]; z = [1,2,3,4,5,6]
```

If *x* and *y* are multi-dimensional, they are flattened before use.

z *: array_like*

The values of the function to interpolate at the data points. If *z* is a multi-dimensional array, it is flattened before use. The length of a flattened *z* array is either len(*x*)*len (*y*) if *x* and *y* specify the column and row coordinates or `len(z) == len(x) == len(y)` if *x* and *y* specify coordinates for each point.

kind *: {'linear', 'cubic', 'quintic'}, optional*

The kind of spline interpolation to use. Default is 'linear'.

copy *: bool, optional*

If True, the class makes internal copies of x, y and z. If False, references may be used. The default is to copy.

bounds_error *: bool, optional*

If True, when interpolated values are requested outside of the domain of the input data (x,y), a ValueError is raised. If False, then *fill_value* is used.

fill_value *: number, optional*

If provided, the value to use for points outside of the interpolation domain. If omitted (None), values outside the domain are extrapolated.

Returns:

values_x *: ndarray, shape xi.shape[:-1] + values.shape[ndim:]*

Interpolated values at input coordinates.

Examples

Construct a 2-D grid and interpolate on it:

```
>>> from scipy import interpolate
>>> x = np.arange(-5.01, 5.01, 0.25)
>>> y = np.arange(-5.01, 5.01, 0.25)
>>> xx, yy = np.meshgrid(x, y)
>>> z = np.sin(xx**2+yy**2)
>>> f = interpolate.interp2d(x, y, z, kind='cubic')
```

Now use the obtained interpolation function and plot the result:

```
>>> xnew = np.arange(-5.01, 5.01, 1e-2)
>>> ynew = np.arange(-5.01, 5.01, 1e-2)
>>> znew = f(xnew, ynew)
>>> plt.plot(x, z[0, :], 'ro-', xnew, znew[0, :], 'b-')
>>> plt.show()
```

numpy.meshgrid

numpy.meshgrid(**xi*, ***kwargs*) [source]

Return coordinate matrices from coordinate vectors.

Make N-D coordinate arrays for vectorized evaluations of N-D scalar/vector fields over N-D grids, given one-dimensional coordinate arrays x1, x2,..., xn.

Changed in version 1.9: 1-D and 0-D cases are allowed.

Parameters:

x1, x2,..., xn : *array_like*
1-D arrays representing the coordinates of a grid.

indexing : *{'xy', 'ij'}, optional*
Cartesian ('xy', default) or matrix ('ij') indexing of output. See Notes for more details.
New in version 1.7.0.

sparse : *bool, optional*
If True a sparse grid is returned in order to conserve memory. Default is False.
New in version 1.7.0.

copy : *bool, optional*
If False, a view into the original arrays are returned in order to conserve memory. Default is True. Please note that `sparse=False, copy=False` will likely return non-contiguous arrays. Furthermore, more than one element of a broadcast array may refer to a single memory location. If you need to write to the arrays, make copies first.
New in version 1.7.0.

Returns:

X1, X2,..., XN : *ndarray*
For vectors *x1*, *x2*,..., 'xn' with lengths `Ni=len(xi)` , return `(N1, N2, N3,...Nn)` shaped arrays if indexing='ij' or `(N2, N1, N3,...Nn)` shaped arrays if indexing='xy' with the elements of *xi* repeated to fill the matrix along the first dimension for *x1*, the second for *x2* and so on.

```
In [53]: x = np.arange(3)

In [54]: y = np.arange(3)

In [55]: xx, yy = np.meshgrid(x, y)

In [56]: xx
Out[56]:
array([[0, 1, 2],
       [0, 1, 2],
       [0, 1, 2]])

In [57]: yy
Out[57]:
array([[0, 0, 0],
       [1, 1, 1],
       [2, 2, 2]])
```

numpy.meshgrid

numpy.meshgrid(**xi*, ***kwargs*) [source]

Return coordinate matrices from coordinate vectors.

Make N-D coordinate arrays for vectorized evaluations of N-D scalar/vector fields over N-D grids, given one-dimensional coordinate arrays x1, x2,..., xn.

Changed in version 1.9: 1-D and 0-D cases are allowed.

Parameters:

x1, x2,..., xn : *array_like*
1-D arrays representing the coordinates of a grid.

indexing : *{'xy', 'ij'}, optional*
Cartesian ('xy', default) or matrix ('ij') indexing of output. See Notes for more details.
New in version 1.7.0.

sparse : *bool, optional*
If True a sparse grid is returned in order to conserve memory. Default is False.
New in version 1.7.0.

copy : *bool, optional*
If False, a view into the original arrays are returned in order to conserve memory. Default is True. Please note that `sparse=False, copy=False` will likely return non-contiguous arrays. Furthermore, more than one element of a broadcast array may refer to a single memory location. If you need to write to the arrays, make copies first.
New in version 1.7.0.

Returns:

X1, X2,..., XN : *ndarray*
For vectors *x1*, *x2*,..., 'xn' with lengths `Ni=len(xi)` , return `(N1, N2, N3,...Nn)` shaped arrays if indexing='ij' or `(N2, N1, N3,...Nn)` shaped arrays if indexing='xy' with the elements of *xi* repeated to fill the matrix along the first dimension for *x1*, the second for *x2* and so on.

```
In [60]: z = sqrt(xx**2 + yy**2)

In [61]: z
Out[61]:
array([[ 0.        ,  1.        ,  2.        ],
       [ 1.        ,  1.41421356,  2.23606798],
       [ 2.        ,  2.23606798,  2.82842712]])
```

numpy.meshgrid

numpy.meshgrid(**xi*, ***kwargs*) [source]

Return coordinate matrices from coordinate vectors.

Make N-D coordinate arrays for vectorized evaluations of N-D scalar/vector fields over N-D grids, given one-dimensional coordinate arrays x1, x2,..., xn.

Changed in version 1.9: 1-D and 0-D cases are allowed.

Parameters:

x1, x2,..., xn : *array_like*
1-D arrays representing the coordinates of a grid.

indexing : *{'xy', 'ij'}, optional*
Cartesian ('xy', default) or matrix ('ij') indexing of output. See Notes for more details.
New in version 1.7.0.

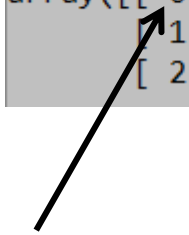
sparse : *bool, optional*
If True a sparse grid is returned in order to conserve memory. Default is False.
New in version 1.7.0.

copy : *bool, optional*
If False, a view into the original arrays are returned in order to conserve memory. Default is True. Please note that `sparse=False, copy=False` will likely return non-contiguous arrays. Furthermore, more than one element of a broadcast array may refer to a single memory location. If you need to write to the arrays, make copies first.
New in version 1.7.0.

Returns:

X1, X2,..., XN : *ndarray*
For vectors *x1*, *x2*,..., 'xn' with lengths `Ni=len(xi)` , return `(N1, N2, N3,...Nn)` shaped arrays if indexing='ij' or `(N2, N1, N3,...Nn)` shaped arrays if indexing='xy' with the elements of *xi* repeated to fill the matrix along the first dimension for *x1*, the second for *x2* and so on.

```
In [60]: z = sqrt(xx**2 + yy**2)

In [61]: z
Out[61]:
array([[ 0.        ,  1.        ,  2.        ],
       [ 1.        ,  1.41421356,  2.23606798],
       [ 2.        ,  2.23606798,  2.82842712]])
```



sqrt(0**2 + 0**2)

numpy.meshgrid

numpy.meshgrid(**xi*, ***kwargs*) [source]

Return coordinate matrices from coordinate vectors.

Make N-D coordinate arrays for vectorized evaluations of N-D scalar/vector fields over N-D grids, given one-dimensional coordinate arrays x1, x2,..., xn.

Changed in version 1.9: 1-D and 0-D cases are allowed.

Parameters:

x1, x2,..., xn : *array_like*
1-D arrays representing the coordinates of a grid.

indexing : *{'xy', 'ij'}, optional*
Cartesian ('xy', default) or matrix ('ij') indexing of output. See Notes for more details.
New in version 1.7.0.

sparse : *bool, optional*
If True a sparse grid is returned in order to conserve memory. Default is False.
New in version 1.7.0.

copy : *bool, optional*
If False, a view into the original arrays are returned in order to conserve memory. Default is True. Please note that `sparse=False, copy=False` will likely return non-contiguous arrays. Furthermore, more than one element of a broadcast array may refer to a single memory location. If you need to write to the arrays, make copies first.
New in version 1.7.0.

Returns:

X1, X2,..., XN : *ndarray*
For vectors *x1*, *x2*,..., 'xn' with lengths `Ni=len(xi)` , return `(N1, N2, N3,...Nn)` shaped arrays if indexing='ij' or `(N2, N1, N3,...Nn)` shaped arrays if indexing='xy' with the elements of *xi* repeated to fill the matrix along the first dimension for *x1*, the second for *x2* and so on.

```
In [60]: z = sqrt(xx**2 + yy**2)

In [61]: z
Out[61]:
array([[ 0.        ,  1.        ,  2.        ],
       [ 1.        ,  1.41421356,  2.23606798],
       [ 2.        ,  2.23606798,  2.82842712]])
```

sqrt(0**2 + 0**2)

sqrt(1**2 + 0**2)

numpy.meshgrid

numpy.meshgrid(**xi, **kwargs*) [source]

Return coordinate matrices from coordinate vectors.

Make N-D coordinate arrays for vectorized evaluations of N-D scalar/vector fields over N-D grids, given one-dimensional coordinate arrays x1, x2,..., xn.

Changed in version 1.9: 1-D and 0-D cases are allowed.

Parameters:

x1, x2,..., xn : *array_like*
1-D arrays representing the coordinates of a grid.

indexing : *{'xy', 'ij'}, optional*
Cartesian ('xy', default) or matrix ('ij') indexing of output. See Notes for more details.
New in version 1.7.0.

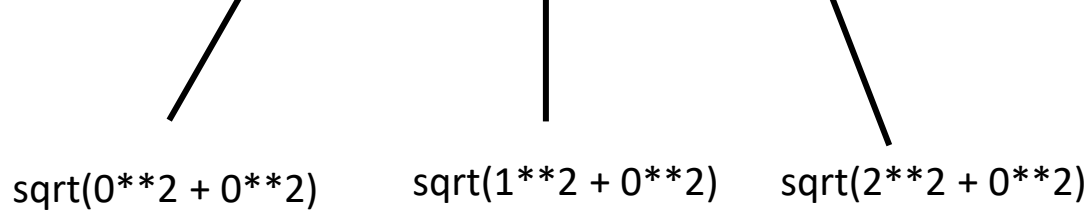
sparse : *bool, optional*
If True a sparse grid is returned in order to conserve memory. Default is False.
New in version 1.7.0.

copy : *bool, optional*
If False, a view into the original arrays are returned in order to conserve memory. Default is True. Please note that `sparse=False, copy=False` will likely return non-contiguous arrays. Furthermore, more than one element of a broadcast array may refer to a single memory location. If you need to write to the arrays, make copies first.
New in version 1.7.0.

Returns:

X1, X2,..., XN : *ndarray*
For vectors *x1*, *x2*,..., 'xn' with lengths `Ni=len(xi)` , return `(N1, N2, N3,...Nn)` shaped arrays if indexing='ij' or `(N2, N1, N3,...Nn)` shaped arrays if indexing='xy' with the elements of *xi* repeated to fill the matrix along the first dimension for *x1*, the second for *x2* and so on.

```
In [60]: z = sqrt(xx**2 + yy**2)

In [61]: z
Out[61]:
array([[ 0.        ,  1.        ,  2.        ],
       [ 1.        ,  1.41421356,  2.23606798],
       [ 2.        ,  2.23606798,  2.82842712]])
```



numpy.meshgrid

numpy.meshgrid(**xi*, ***kwargs*) [source]

Return coordinate matrices from coordinate vectors.

Make N-D coordinate arrays for vectorized evaluations of N-D scalar/vector fields over N-D grids, given one-dimensional coordinate arrays x1, x2,..., xn.

Changed in version 1.9: 1-D and 0-D cases are allowed.

Parameters:

x1, x2,..., xn : *array_like*
1-D arrays representing the coordinates of a grid.

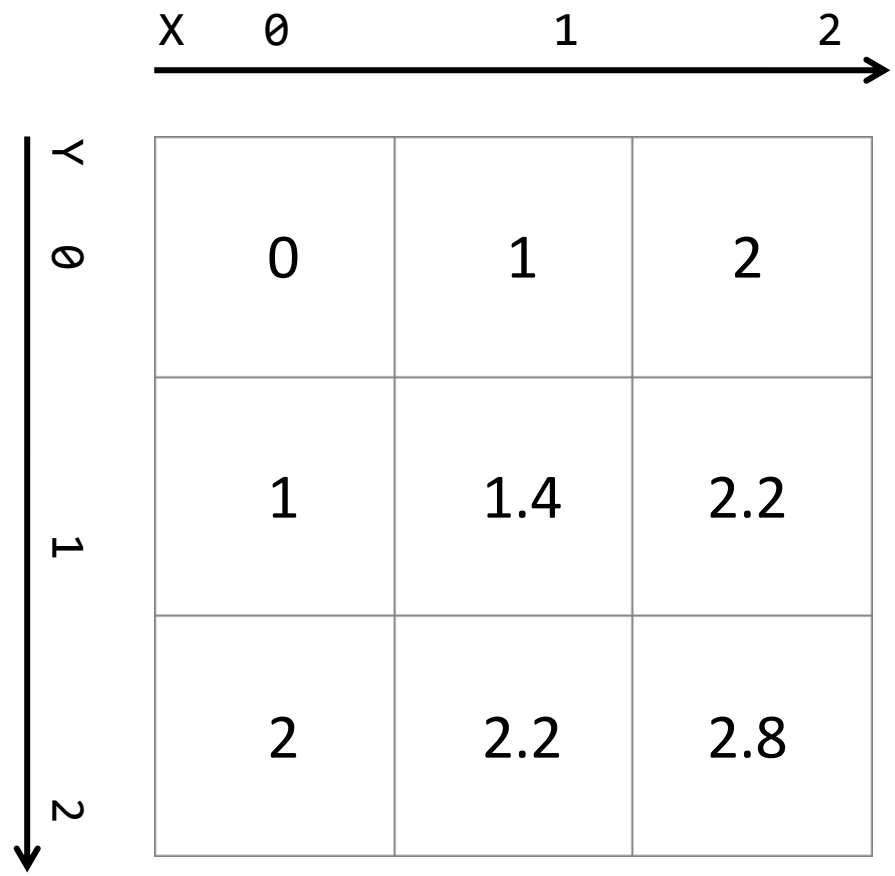
indexing : *{'xy', 'ij'}, optional*
Cartesian ('xy', default) or matrix ('ij') indexing of output. See Notes for more details.
New in version 1.7.0.

sparse : *bool, optional*
If True a sparse grid is returned in order to conserve memory. Default is False.
New in version 1.7.0.

copy : *bool, optional*
If False, a view into the original arrays are returned in order to conserve memory. Default is True. Please note that `sparse=False, copy=False` will likely return non-contiguous arrays. Furthermore, more than one element of a broadcast array may refer to a single memory location. If you need to write to the arrays, make copies first.
New in version 1.7.0.

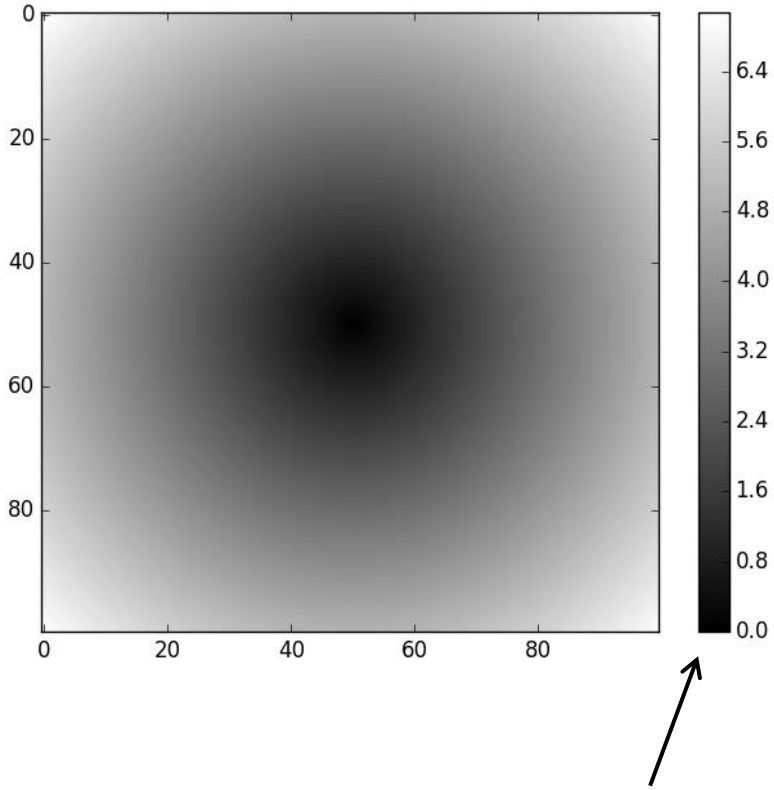
Returns:

X1, X2,..., XN : *ndarray*
For vectors *x1*, *x2*,..., 'xn' with lengths `Ni=len(xi)` , return `(N1, N2, N3,...Nn)` shaped arrays if indexing='ij' or `(N2, N1, N3,...Nn)` shaped arrays if indexing='xy' with the elements of *xi* repeated to fill the matrix along the first dimension for *x1*, the second for *x2* and so on.



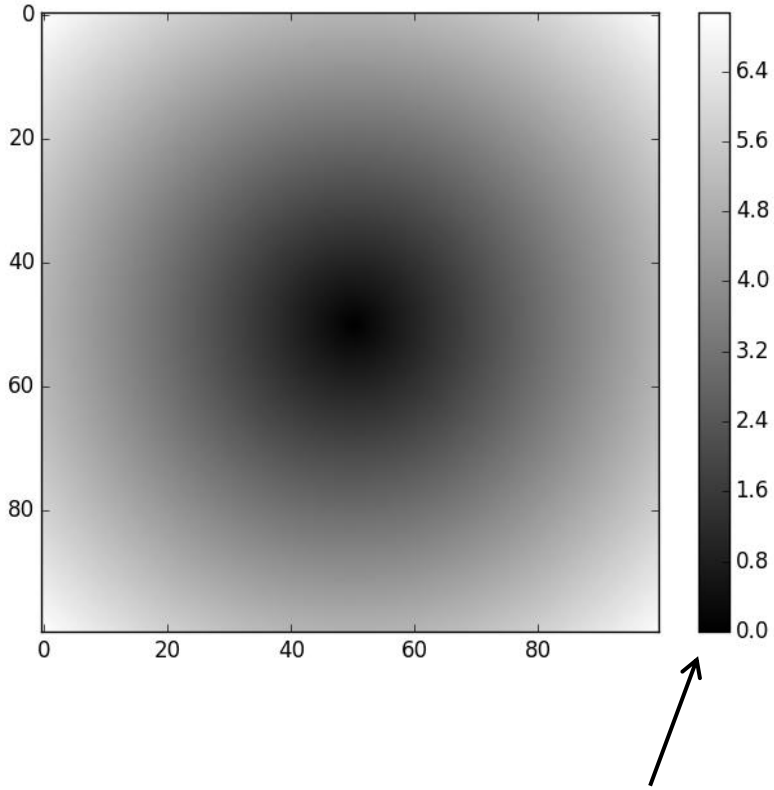
Visualizing 2D functions as color images

$$f(x, y) = \sqrt{x^2 + y^2}$$



*plt.colorbar() creates a color bar

$$f(x,y) = \sqrt{x^2 + y^2}$$



*plt.colorbar() creates a color bar

`matplotlib.pyplot.imshow(X, cmap=None, norm=None, aspect=None, interpolation=None, alpha=None, vmin=None, vmax=None, origin=None, extent=None, shape=None, filternorm=1, filterrad=4.0, imlim=None, resample=None, url=None, hold=None, data=None, **kwargs)`

Display an image on the axes.

Parameters:

X : array_like, shape (n, m) or (n, m, 3) or (n, m, 4)

Display the image in x to current axes. x may be a float array, a uint8 array or a PIL image. If x is an array, it can have the following shapes:

- MxN – luminance (grayscale, float array only)
- MxNx3 – RGB (float or uint8 array)
- MxNx4 – RGBA (float or uint8 array)

The value for each component of MxNx3 and MxNx4 float arrays should be in the range 0.0 to 1.0; MxN float arrays may be normalised.

cmap : `Colormap`, optional, default: None

If None, default to rc `image.cmap` value. `cmap` is ignored when x has RGB(A) information

aspect : ['auto' | 'equal' | scalar], optional, default: None

If 'auto', changes the image aspect ratio to match that of the axes.

If 'equal', and `extent` is None, changes the axes aspect ratio to match that of the image. If `extent` is not `None`, the axes aspect ratio is changed to match that of the extent.

If None, default to rc `image.aspect` value.

interpolation : string, optional, default: None

Acceptable values are 'none', 'nearest', 'bilinear', 'bicubic', 'spline16', 'spline36', 'hanning', 'hamming', 'hermite', 'kaiser', 'quadric', 'catrom', 'gaussian', 'bessel', 'mitchell', 'sinc', 'lanczos'

If `interpolation` is None, default to rc `image.interpolation`. See also the `filternorm` and `filterrad` parameters. If `interpolation` is 'none', then no interpolation is performed on the Agg, ps and pdf backends. Other backends will fall back to 'nearest'.

norm : `Normalize`, optional, default: None

A `Normalize` instance is used to scale luminance data to 0, 1. If `None`, use the default func:`normalize`. `norm` is only used if x is an array of floats.

```
z = sqrt(xx**2 + yy**2)
plt.imshow(z)
```

matplotlib.pyplot.**imshow**(*X, cmap=None, norm=None, aspect=None, interpolation=None, alpha=None, vmin=None, vmax=None, origin=None, extent=None, shape=None, filternorm=1, filterrad=4.0, imlim=None, resample=None, url=None, hold=None, data=None, **kwargs*)

Display an image on the axes.

Parameters:

X : array_like, shape (n, m) or (n, m, 3) or (n, m, 4)

Display the image in x to current axes. x may be a float array, a uint8 array or a PIL image. If x is an array, it can have the following shapes:

- MxN – luminance (grayscale, float array only)
- MxNx3 – RGB (float or uint8 array)
- MxNx4 – RGBA (float or uint8 array)

The value for each component of MxNx3 and MxNx4 float arrays should be in the range 0.0 to 1.0; MxN float arrays may be normalised.

cmap : `Colormap`, optional, default: None

If None, default to rc `image.cmap` value. `cmap` is ignored when x has RGB(A) information

aspect : ['auto' | 'equal' | scalar], optional, default: None

If 'auto', changes the image aspect ratio to match that of the axes.

If 'equal', and `extent` is None, changes the axes aspect ratio to match that of the image. If `extent` is not `None`, the axes aspect ratio is changed to match that of the extent.

If None, default to rc `image.aspect` value.

interpolation : string, optional, default: None

Acceptable values are 'none', 'nearest', 'bilinear', 'bicubic', 'spline16', 'spline36', 'hanning', 'hamming', 'hermite', 'kaiser', 'quadric', 'catrom', 'gaussian', 'bessel', 'mitchell', 'sinc', 'lanczos'

If `interpolation` is None, default to rc `image.interpolation`. See also the `filternorm` and `filterrad` parameters. If `interpolation` is 'none', then no interpolation is performed on the Agg, ps and pdf backends. Other backends will fall back to 'nearest'.

norm : `Normalize`, optional, default: None

A `Normalize` instance is used to scale luminance data to 0, 1. If `None`, use the default func:`normalize`. `norm` is only used if x is an array of floats.

plt.pcolormesh(x, y, z)

matplotlib.pyplot.**pcolormesh**(**args*, ***kwargs*)

Plot a quadrilateral mesh.

Call signatures:

```
pcolormesh(C)
pcolormesh(X, Y, C)
pcolormesh(C, **kwargs)
```

Create a pseudocolor plot of a 2-D array.

pcolormesh is similar to `pcolor()`, but uses a different mechanism and returns a different object; pcolor returns a `PolyCollection` but **pcolormesh** returns a `QuadMesh`. It is much faster, so it is almost always preferred for large arrays.

C may be a masked array, but *X* and *Y* may not. Masked array support is implemented via *cmap* and *norm*; in contrast, `pcolor()` simply does not draw quadrilaterals with masked colors or vertices.

Keyword arguments:

cmap: [*None* | Colormap]

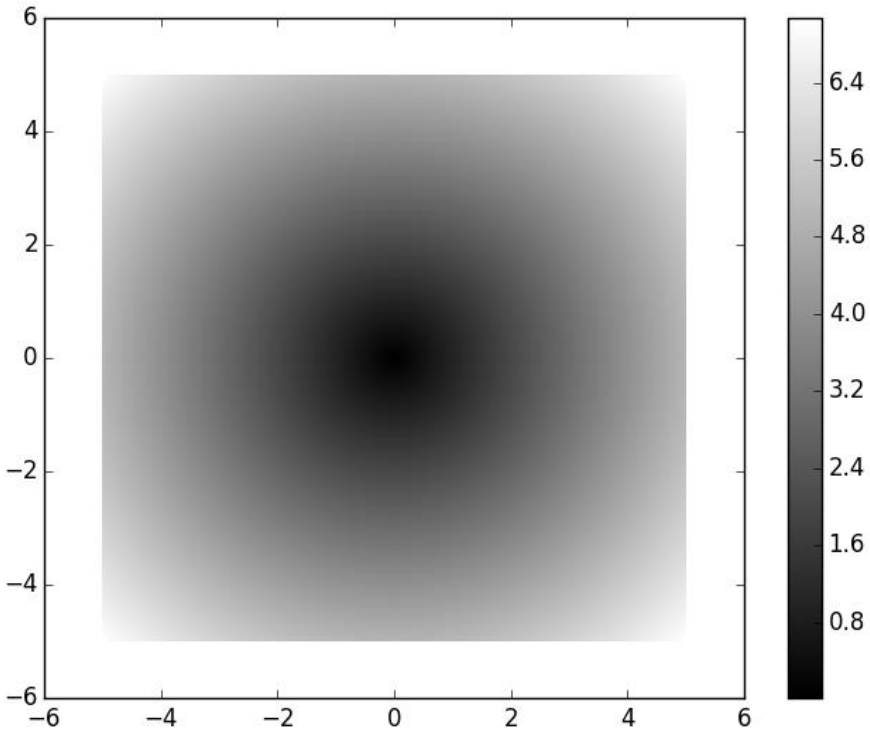
A `matplotlib.colors.Colormap` instance. If *None*, use rc settings.

norm: [*None* | Normalize]

A `matplotlib.colors.Normalize` instance is used to scale luminance data to 0,1. If *None*, defaults to `normalize()`.

vmin/*vmax*: [*None* | scalar]

vmin and *vmax* are used in conjunction with *norm* to normalize luminance data. If either is *None*, it is autoscaled to the respective min or max of the color array *C*. If not *None*, *vmin* or *vmax* passed in here override any pre-existing values supplied in the *norm* instance.



plt.pcolormesh(x, y, z)

matplotlib.pyplot.**pcolormesh**(**args*, ***kwargs*)

Plot a quadrilateral mesh.

Call signatures:

```
pcolormesh(C)
pcolormesh(X, Y, C)
pcolormesh(C, **kwargs)
```

Create a pseudocolor plot of a 2-D array.

pcolormesh is similar to `pcolor()`, but uses a different mechanism and returns a different object; pcolor returns a `PolyCollection` but **pcolormesh** returns a `QuadMesh`. It is much faster, so it is almost always preferred for large arrays.

C may be a masked array, but *X* and *Y* may not. Masked array support is implemented via *cmap* and *norm*; in contrast, `pcolor()` simply does not draw quadrilaterals with masked colors or vertices.

Keyword arguments:

cmap: [*None* | Colormap]

A `matplotlib.colors.Colormap` instance. If *None*, use rc settings.

norm: [*None* | Normalize]

A `matplotlib.colors.Normalize` instance is used to scale luminance data to 0,1. If *None*, defaults to `normalize()`.

vmin/*vmax*: [*None* | scalar]

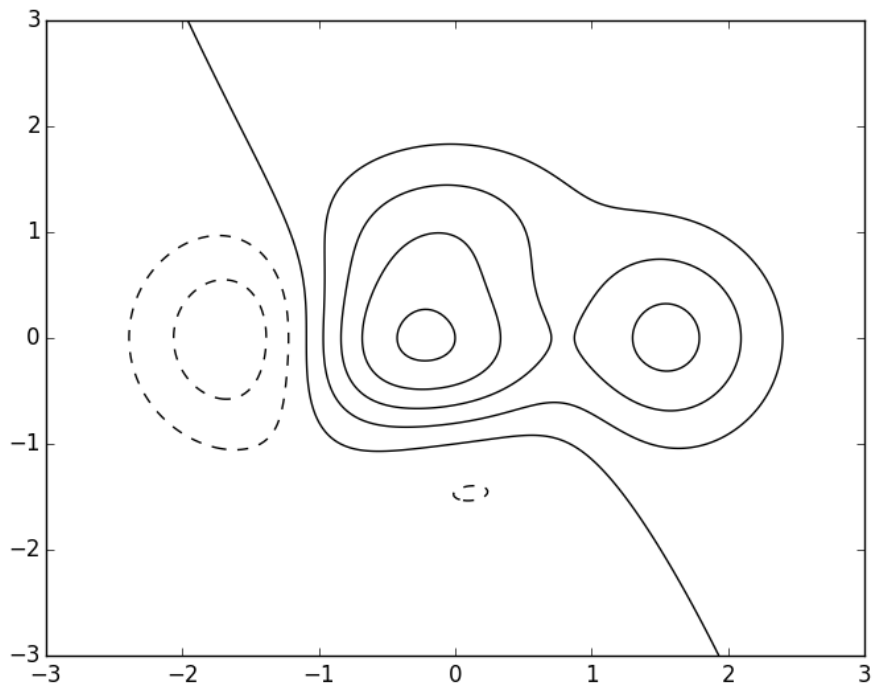
vmin and *vmax* are used in conjunction with *norm* to normalize luminance data. If either is *None*, it is autoscaled to the respective min or max of the color array *C*. If not *None*, *vmin* or *vmax* passed in here override any pre-existing values supplied in the *norm* instance.

matplotlib.pyplot.contour(*args, **kwargs)

Plot contours.

contour() and contourf() draw contour lines and filled contours, respectively. Except as noted, function signatures and return values are the same for both versions.

contourf() differs from the MATLAB version in that it does not draw the polygon edges. To draw edges, add line contours with calls to contour().



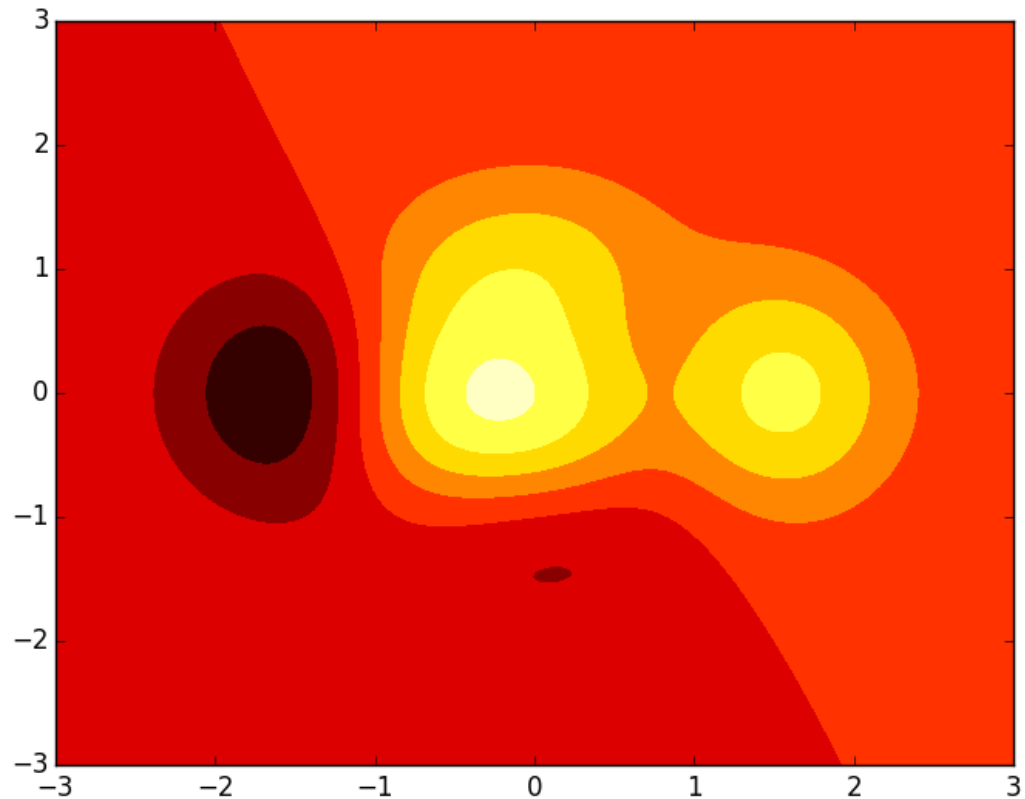
---

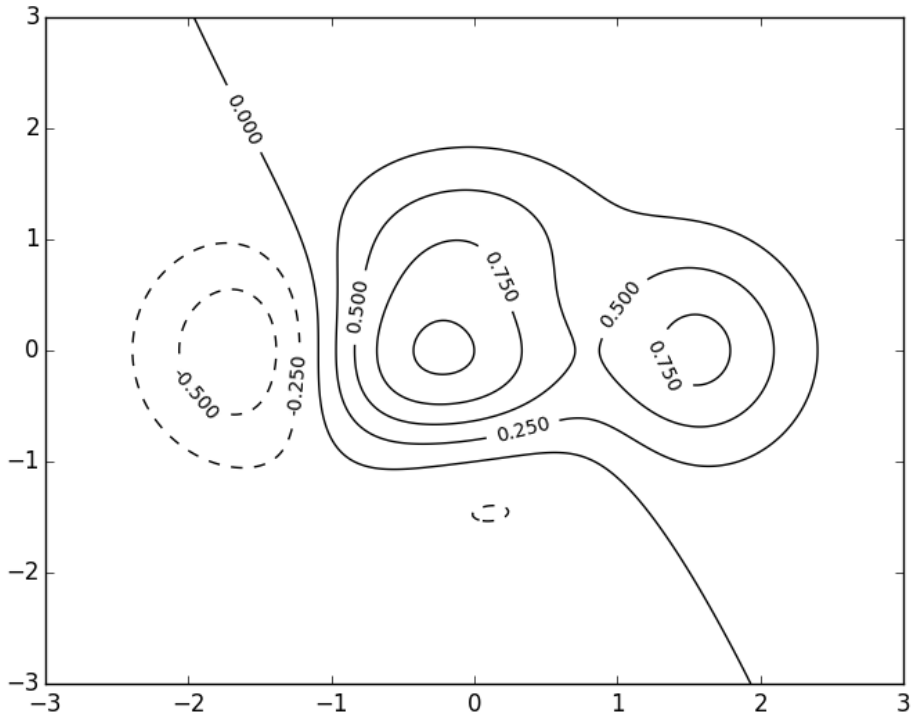
matplotlib.pyplot.**contourf**(**args*, ***kwargs*)

Plot contours.

`contour()` and `contourf()` draw contour lines and filled contours, respectively. Except as noted, function signatures and return values are the same for both versions.

`contourf()` differs from the MATLAB version in that it does not draw the polygon edges. To draw edges, add line contours with calls to `contour()`.





---

matplotlib.pyplot.**clabel**(*CS*, **args*, ***kwargs*)

Label a contour plot.

Call signature:

```
clabel(cs, **kwargs)
```

Adds labels to line contours in *cs*, where *cs* is a `ContourSet` object returned by contour.

```
clabel(cs, v, **kwargs)
```

only labels contours listed in *v*.

Optional keyword arguments:

fontsize:

size in points or relative size e.g., 'smaller', 'x-large'

colors:

- if *None*, the color of each label matches the color of the corresponding contour
- if one string color, e.g., *colors* = 'r' or *colors* = 'red', all labels will be plotted in this color
- if a tuple of matplotlib color args (string, float, rgb, etc), different labels will be plotted in different colors in the order specified

inline:

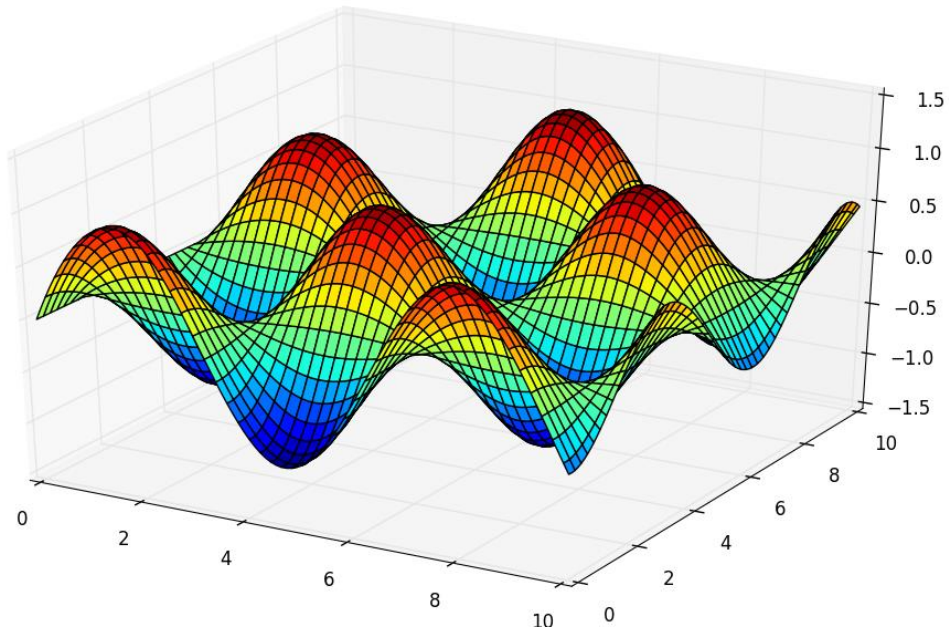
controls whether the underlying contour is removed or not. Default is *True*.

inline_spacing:

space in pixels to leave on each side of label when placing inline. Defaults to 5. This spacing will be exact for labels at locations where the contour is straight, less so for labels on curved contours.

```
from mpl_toolkits.mplot3d import Axes3D
```

- **Axes3D(plt.gcf())** creates a 3D graph.



`plot_surface(X, Y, Z, *args, **kwargs)`

Create a surface plot.

By default it will be colored in shades of a solid color, but it also supports color mapping by supplying the *cmap* argument.

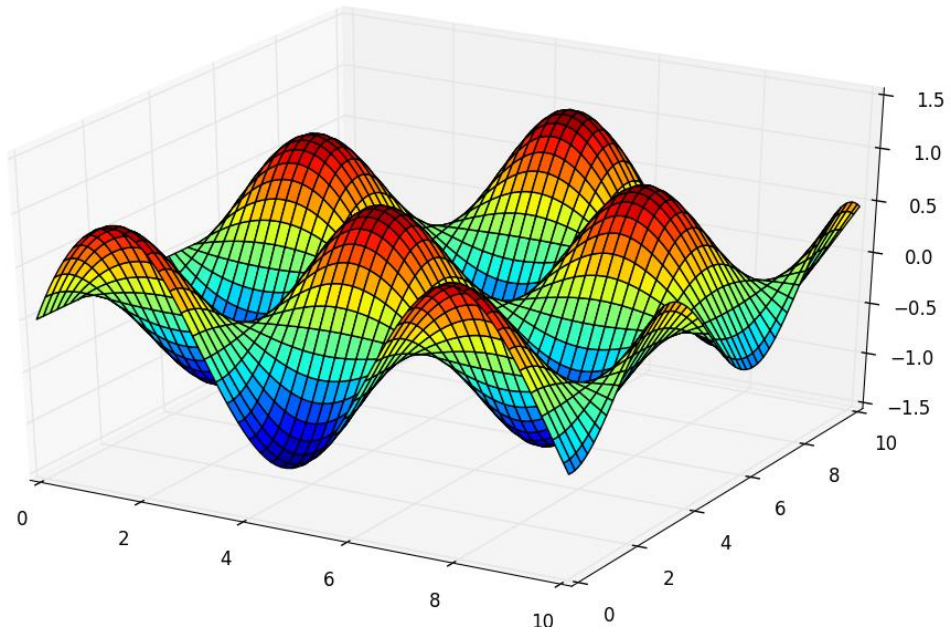
The `rstride` and `cstride` kwargs set the stride used to sample the input data to generate the graph. If 1k by 1k arrays are passed in the default values for the strides will result in a 100x100 grid being plotted.

Argument	Description
X, Y, Z	Data values as 2D arrays
rstride	Array row stride (step size), defaults to 10
cstride	Array column stride (step size), defaults to 10
color	Color of the surface patches
cmap	A colormap for the surface patches.
facecolors	Face colors for the individual patches
norm	An instance of Normalize to map values to colors
vmin	Minimum value to map
vmax	Maximum value to map
shade	Whether to shade the facecolors

Other arguments are passed on to `Poly3DCollection`

3D Plots

- ax = plt.figure().add_subplot(projection='3d')



`plot_surface(X, Y, Z, *args, **kwargs)`

Create a surface plot.

By default it will be colored in shades of a solid color, but it also supports color mapping by supplying the *cmap* argument.

The `rstride` and `cstride` kwargs set the stride used to sample the input data to generate the graph. If 1k by 1k arrays are passed in the default values for the strides will result in a 100x100 grid being plotted.

Argument	Description
X, Y, Z	Data values as 2D arrays
rstride	Array row stride (step size), defaults to 10
cstride	Array column stride (step size), defaults to 10
color	Color of the surface patches
cmap	A colormap for the surface patches.
facecolors	Face colors for the individual patches
norm	An instance of Normalize to map values to colors
vmin	Minimum value to map
vmax	Maximum value to map
shade	Whether to shade the facecolors

Other arguments are passed on to `Poly3DCollection`

3D Plots

- Create mesh grids: meshgrid(x, y)
- Calculate z
- ax.plot_surface(x, y, z)

`plot_surface(X, Y, Z, *args, **kwargs)`

Create a surface plot.

By default it will be colored in shades of a solid color, but it also supports color mapping by supplying the *cmap* argument.

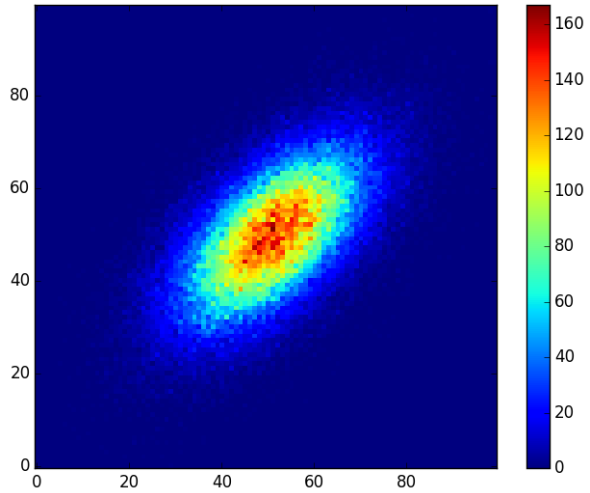
The `rstride` and `cstride` kwargs set the stride used to sample the input data to generate the graph. If 1k by 1k arrays are passed in the default values for the strides will result in a 100x100 grid being plotted.

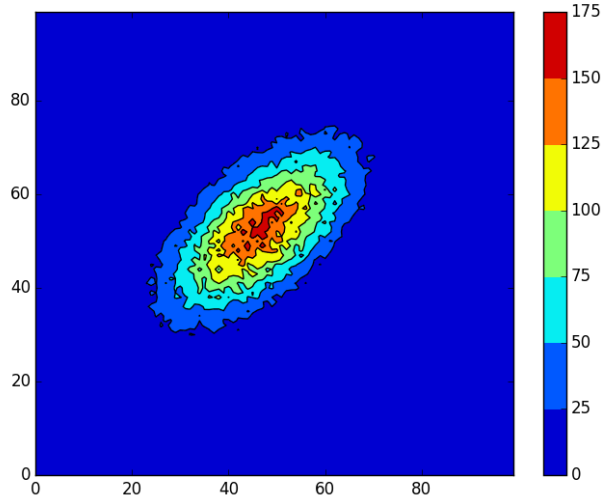
Argument	Description
X, Y, Z	Data values as 2D arrays
rstride	Array row stride (step size), defaults to 10
cstride	Array column stride (step size), defaults to 10
color	Color of the surface patches
cmap	A colormap for the surface patches.
facecolors	Face colors for the individual patches
norm	An instance of Normalize to map values to colors
vmin	Minimum value to map
vmax	Maximum value to map
shade	Whether to shade the facecolors

Other arguments are passed on to `Poly3DCollection`

Exercise

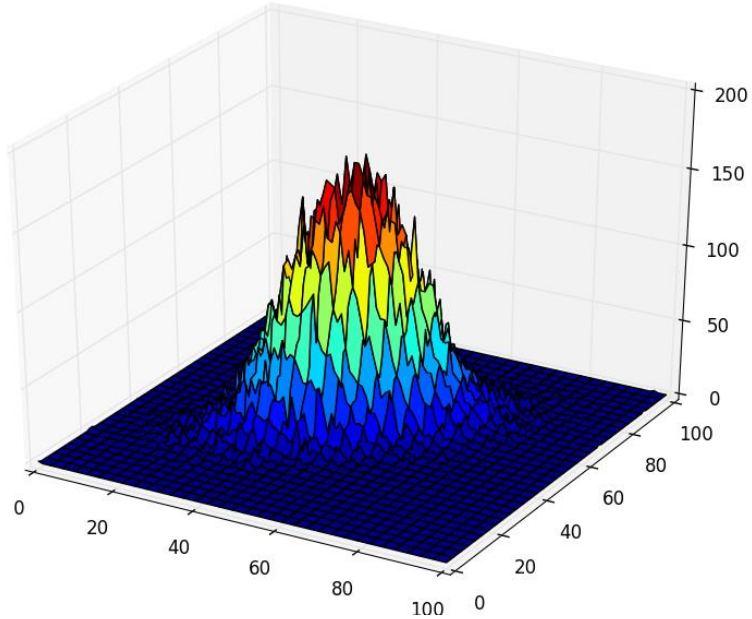
- Download the files X and Y containing two parameters representing some real world data. Create a 2D histogram of these two data samples – visualize them as an image and as a contour plot. Use the function histogram2D to create a 2D histogram. Draw a color bar.





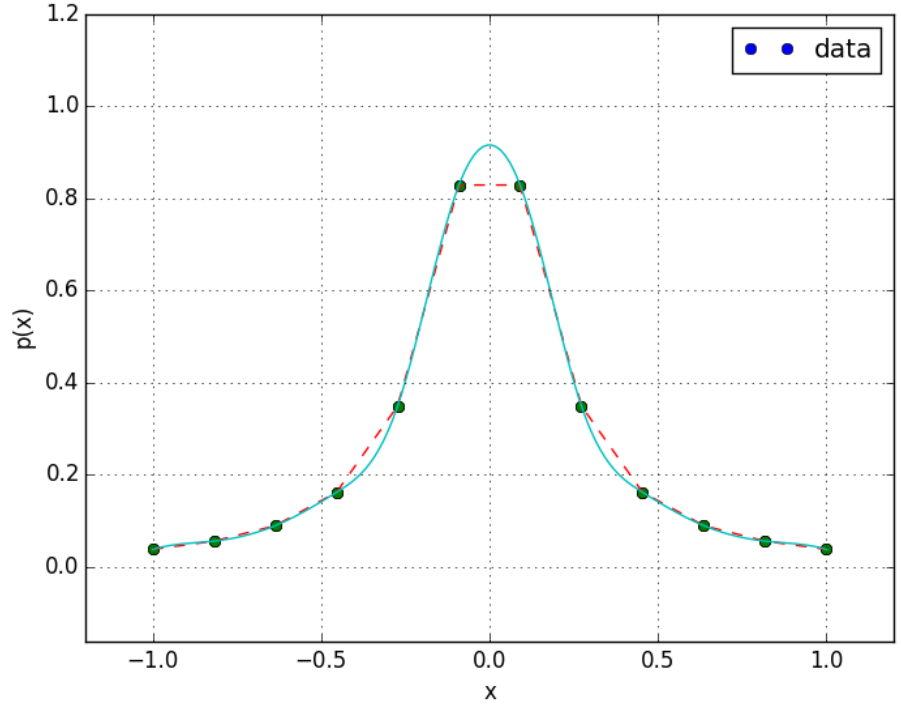
Exercise

- Create a 3D plot of the 2D histogram from the previous exercise. Generate a mesh grid for x and y from 0 to 100 with the linspace and meshgrid functions. Use the plot_surface function with the color map “jet” to visualize the 3D plot.



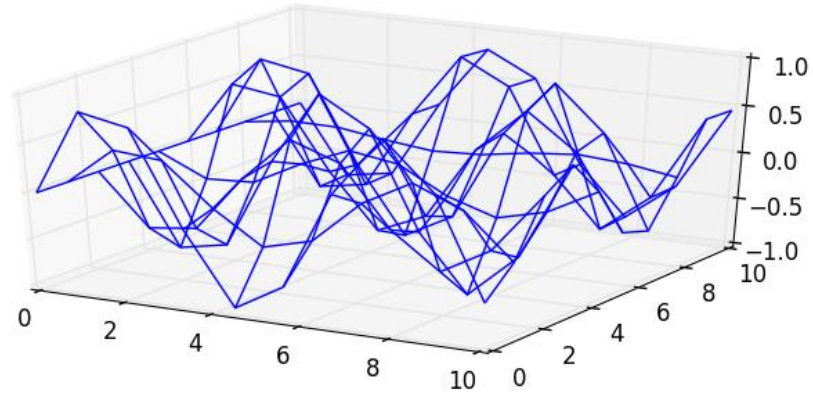
Exercise

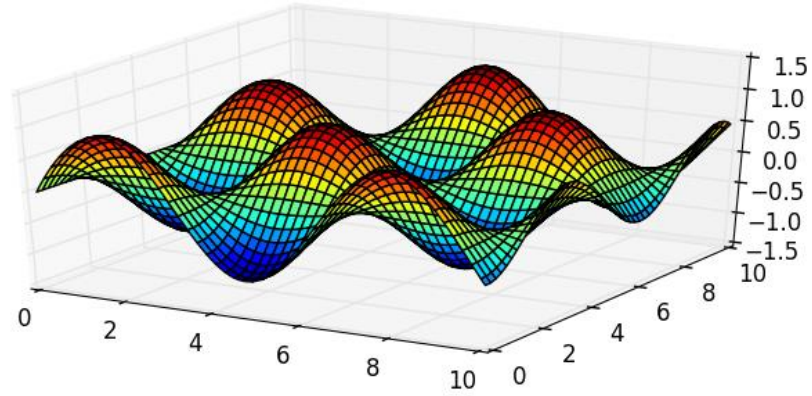
- Find a cubic spline interpolation of the points (x and y coordinates) contained in the file inter1D.npz in the range $x \in [-1.0, 1.0]$. Draw the points and the interpolating function.



Exercise

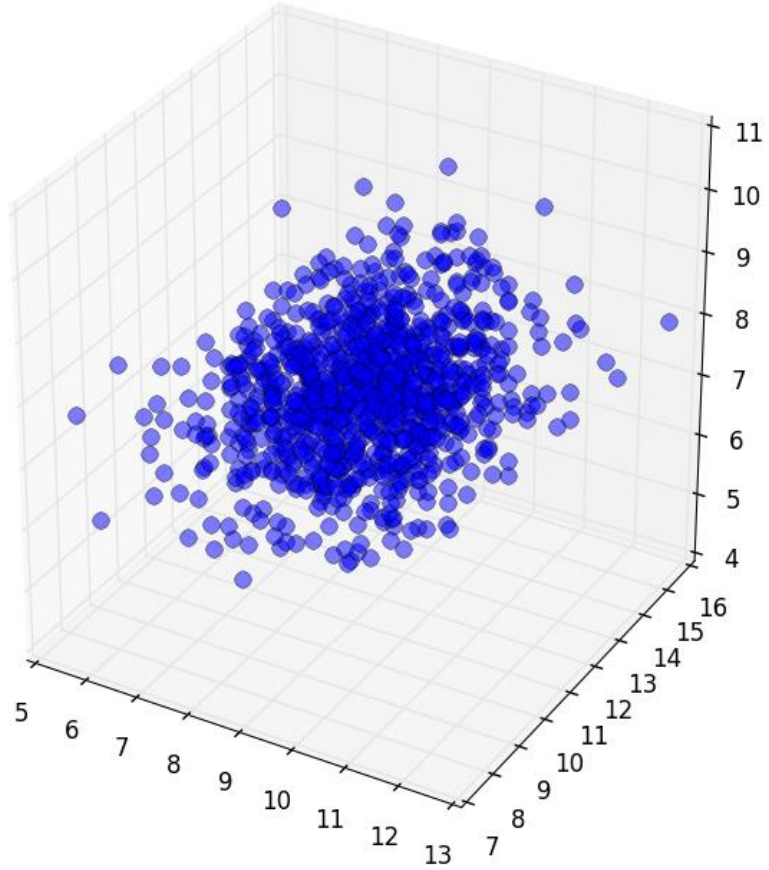
- Open the file inter2D.npz and visualize the data as a 3D wireframe plot (plot_wireframe)
- Find a smoother (increase stride) cubic 2D spline interpolation for these data points (*x, y* and *z* coordinates*)* in the range $x, y \in [0, 10]$ and visualize the result as a 3D plot.





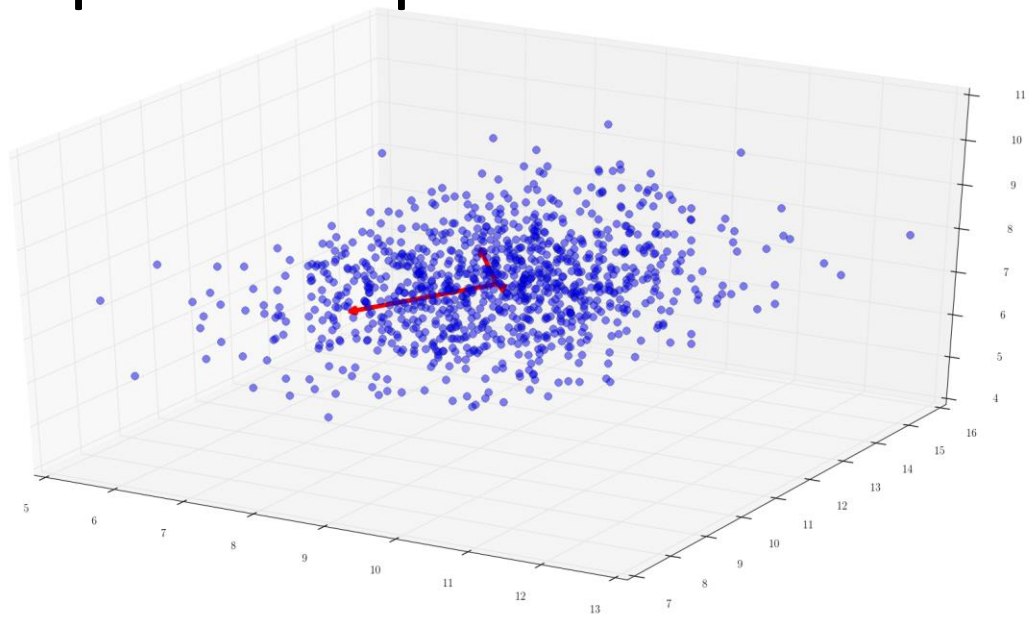
Exercise

- Visualize measurements of the flowers of *ludwiga octovalvis* on a 3D plot. Compute the means and the covariance matrix as well as the correlation matrix – what can you say about the relations of the 3 parameters: sepal length, petal length and sepal width?



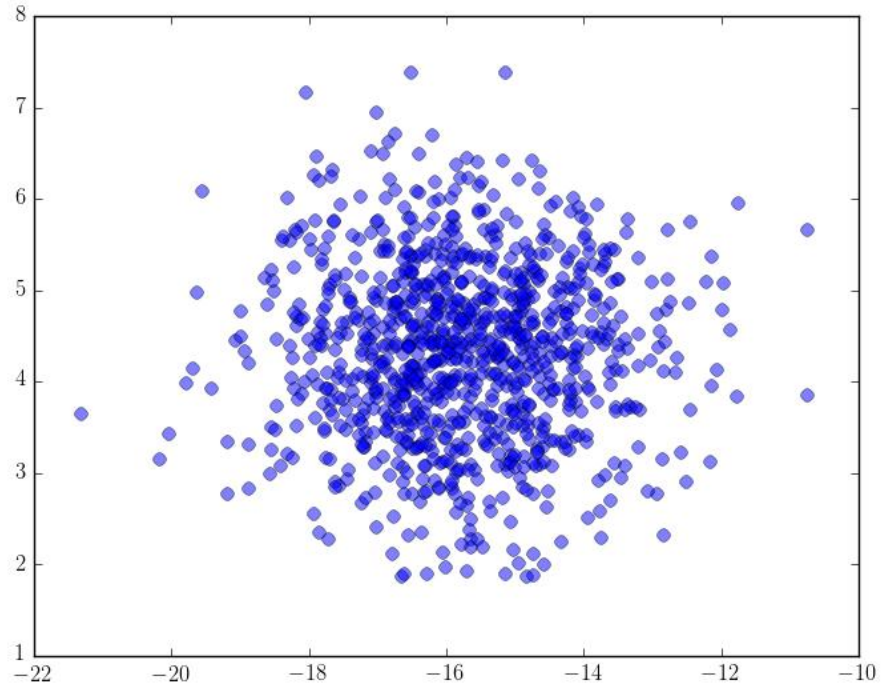
Exercise

- Calculate eigenvectors and eigenvalues of the covariance matrix. Visualize the vectors originating at the data mean (code). To reduce the dimensionality of the data from three to two dimensions what subspace will preserve most information?



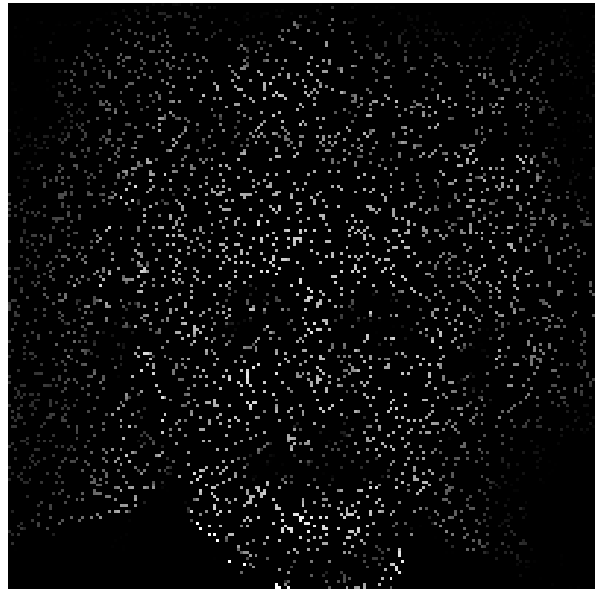
Exercise

- Project the data from 3D to 2D and visualize the result on another plot (multiply the data with a 3x2 matrix constructed from the two largest eigenvectors – alternatively use SVD). Compare your results to the PCA method from the module matplotlib.mlab or sklearn.



Exercise

- Reconstruct the image using an interpolation method of your choice. Which person is depicted in the image?



scipy.interpolate.**griddata(***points, values, xi, method='linear', fill_value=nan, rescale=False***)**