Lists[1]

"Essential Mathematica for Students of Science",James J. Kelly , 2006

http://www.physics.umd.edu/courses/CourseWare

"*Mathematica 4.1* Notebooks **-Complimentary software to accompany our textbook"**,John H. Mathews, and Russell W. Howell, 2002

Initialization

```
ClearAll["Global`*"];
Off[General::spell, General::spell1]
```

Types of lists

The basic data structure in Mathematica is the list. A **List** is an ordered set of objects separated by commas and enclosed in curly brackets. There are no restrictions on the contents or structure of lists.The elements of a list can be any combination of numbers, variables, strings, functions, and other lists; enclosed within curly brackets { } and separated by commas.Mathematica lists often have natural interpretations as vectors, matrices, or tensors, depending on how deeply they are nested. A one-dimensional list acts as a vector, a two-dimensional list as a matrix, and higher-dimensional lists as tensors.One way to enter an arbitrary list is to type a sequence of elements inside curly brackets and separated by commas. The following one-dimensional list vec can be considered a vector with the three components a, b, and c.

```
vec = {a, b, c};
```

An easier way to create a vector of evenly spaced numbers is to use Range. **Range[max]** creates a list of integers from 1 through max; **Range[min, max]** creates a list of numbers from min through max in increments of 1; and **Range[min, max, inc]** returns a list of numbers ranging from min through max with increment inc.

```
vec2 = Range[2, 10, 1 / 2]
```

$$\left\{2, \frac{5}{2}, 3, \frac{7}{2}, 4, \frac{9}{2}, 5, \frac{11}{2}, 6, \frac{13}{2}, 7, \frac{15}{2}, 8, \frac{17}{2}, 9, \frac{19}{2}, 10\right\}$$

A more general function for creating lists is **Table**, which generates lists (nested to any depth) of elements that may depend on the given iterators. For example, to generate a list of values of k^2 as k takes on values from 1 to 10, we evaluate the following.

```
Table[i^2, {i, 1, 10}]
```

```
{1, 4, 9, 16, 25, 36, 49, 64, 81, 100}
```

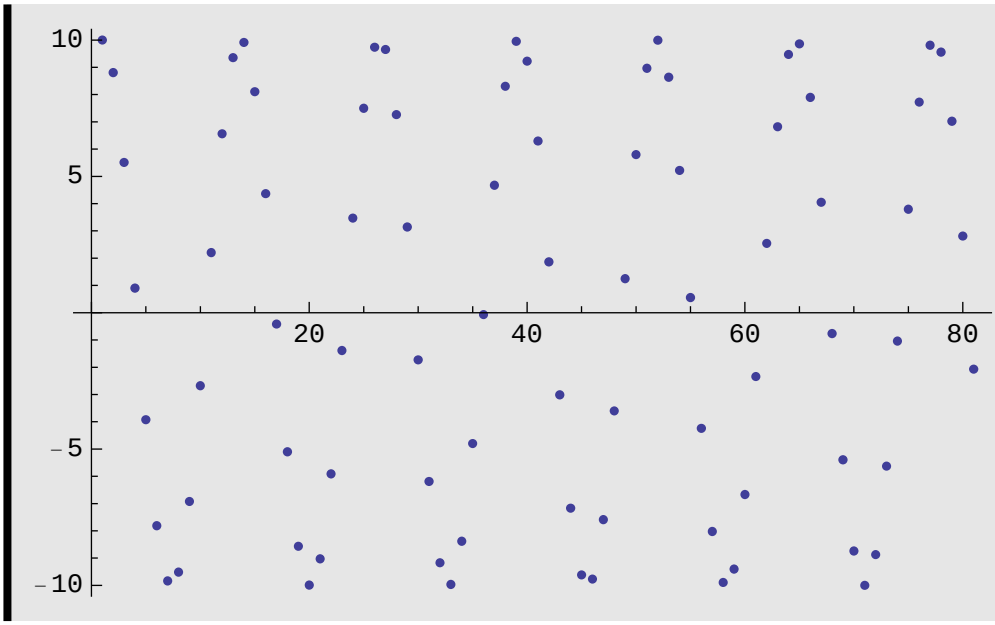
We create tables of more complicated functions in a similar way.

```
vec3 = Table[N[Exp[2 * π * i * t]], {t, 0, 2 π, π / 40}]
```

```
{1., 0.88069 + 0.473694 i, 0.551228 + 0.834354 i,
 0.0902328 + 0.995921 i, -0.392294 + 0.91984 i, -0.781212 + 0.624266 i,
 -0.983716 + 0.179729 i, -0.951485 - 0.307694 i, -0.69221 - 0.721696 i,
 -0.26776 - 0.963486 i, 0.220584 - 0.975368 i, 0.656292 - 0.754507 i,
 0.935395 - 0.353605 i, 0.991293 + 0.131674 i, 0.810648 + 0.585533 i,
 0.436566 + 0.899672 i, -0.0416898 + 0.999131 i, -0.509998 + 0.860176 i,
 -0.85661 + 0.515965 i, -0.998817 + 0.0486346 i, -0.902685 - 0.430301 i,
 -0.591155 - 0.806558 i, -0.138562 - 0.990354 i, 0.347094 - 0.93783 i,
 0.749927 - 0.661521 i, 0.973811 - 0.227359 i, 0.965324 + 0.261055 i,
 0.726491 + 0.687176 i, 0.314302 + 0.949323 i, -0.172886 + 0.984942 i,
 -0.61882 + 0.785533 i, -0.91709 + 0.398679 i, -0.996524 - 0.0833071 i,
 -0.838166 - 0.545415 i, -0.479805 - 0.877375 i, -0.00695184 - 0.999976 i,
 0.46756 - 0.883961 i, 0.830502 - 0.557015 i, 0.995269 - 0.0971541 i,
 0.922545 + 0.38589 i, 0.629682 + 0.776853 i, 0.186564 + 0.982443 i,
 -0.301072 + 0.953601 i, -0.716866 + 0.697211 i, -0.961601 + 0.274451 i,
 -0.976878 - 0.213798 i, -0.759051 - 0.651031 i, -0.3601 - 0.932914 i,
 0.12478 - 0.992184 i, 0.579884 - 0.814699 i, 0.896615 - 0.44281 i,
 0.999396 + 0.034743 i, 0.8637 + 0.504006 i, 0.521908 + 0.853002 i,
 0.055577 + 0.998454 i, -0.424015 + 0.905655 i, -0.802429 + 0.596747 i,
 -0.989367 + 0.145444 i, -0.940221 - 0.340566 i, -0.666719 - 0.74531 i,
 -0.234124 - 0.972207 i, 0.254338 - 0.967115 i, 0.682109 - 0.73125 i,
 0.947115 - 0.320893 i, 0.98612 + 0.166035 i, 0.789816 + 0.613344 i,
 0.405045 + 0.914297 i, -0.0763774 + 0.997079 i, -0.539575 + 0.841938 i,
 -0.874019 + 0.485892 i, -0.999903 + 0.0139033 i, -0.88719 - 0.461403 i,
 -0.562776 - 0.82661 i, -0.104071 - 0.99457 i, 0.379468 - 0.925205 i,
 0.772457 - 0.635067 i, 0.981122 - 0.193389 i, 0.955671 + 0.294436 i,
 0.702177 + 0.712002 i, 0.281129 + 0.95967 i, -0.207002 + 0.978341 i}
```

Many mathematical and plotting functions are designed to operate on lists. Here is a plot of the real part of each element of vec3.

```
ListPlot[10 * Re[vec3], PlotStyle → PointSize[0.01`]]
```

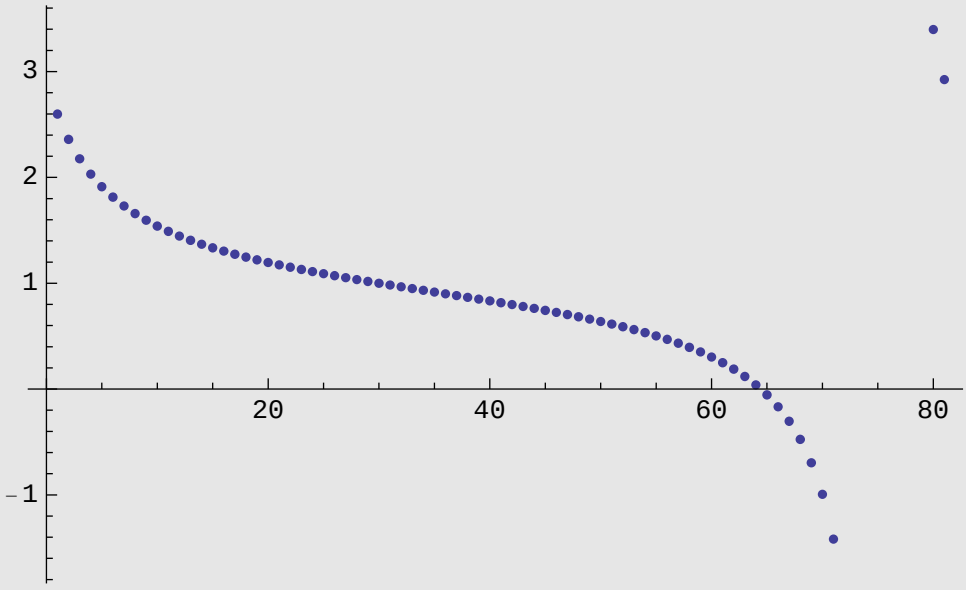


Fourier[list](**finds the discrete Fourier transform of a list of complex numbers)** expects a list of values.

```
four = Fourier[vec3]
```

```
{0.259848 + 0.320581 i, 0.235939 + 0.269047 i, 0.217612 + 0.229547 i,
 0.203079 + 0.198222 i, 0.191239 + 0.172703 i, 0.18138 + 0.151454 i,
 0.17302 + 0.133436 i, 0.165822 + 0.117921 i, 0.159541 + 0.104383 i,
 0.153996 + 0.0924329 i, 0.149052 + 0.0817768 i, 0.144603 + 0.0721877 i,
 0.140567 + 0.0634883 i, 0.136878 + 0.0555373 i, 0.133484 + 0.048221 i,
 0.130341 + 0.0414468 i, 0.127414 + 0.0351382 i, 0.124674 + 0.0292318 i,
 0.122095 + 0.0236739 i, 0.119657 + 0.0184194 i, 0.117342 + 0.0134294 i,
 0.115134 + 0.00867048 i, 0.11302 + 0.00411356 i, 0.110987 - 0.000266846 i,
 0.109026 - 0.00449327 i, 0.107128 - 0.00858579 i, 0.105283 - 0.0125624 i,
 0.103484 - 0.0164396 i, 0.101724 - 0.0202321 i, 0.0999974 - 0.0239539 i,
 0.0982975 - 0.0276178 i, 0.0966188 - 0.031236 i, 0.094956 - 0.0348199 i,
 0.0933039 - 0.0383809 i, 0.0916574 - 0.0419296 i, 0.0900116 - 0.0454769 i,
 0.0883615 - 0.0490333 i, 0.0867022 - 0.0526098 i, 0.0850284 - 0.0562174 i,
 0.0833349 - 0.0598675 i, 0.0816162 - 0.063572 i, 0.0798663 - 0.0673436 i,
 0.078079 - 0.0711959 i, 0.0762474 - 0.0751435 i, 0.0743643 - 0.0792022 i,
 0.0724215 - 0.0833897 i, 0.0704099 - 0.0877254 i, 0.0683195 - 0.0922311 i,
 0.0661387 - 0.0969314 i, 0.0638546 - 0.101854 i, 0.0614524 - 0.107032 i,
 0.0589146 - 0.112502 i, 0.0562213 - 0.118307 i, 0.0533487 - 0.124498 i,
 0.0502687 - 0.131137 i, 0.0469477 - 0.138295 i, 0.043345 - 0.14606 i,
 0.0394108 - 0.154539 i, 0.0350834 - 0.163866 i, 0.0302857 - 0.174207 i,
 0.0249195 - 0.185773 i, 0.018858 - 0.198838 i, 0.0119345 - 0.21376 i,
 0.00392507 - 0.231024 i, -0.00547822 - 0.251291 i, -0.0167109 - 0.275501 i,
 -0.0304098 - 0.305027 i, -0.0475455 - 0.341961 i, -0.0696727 - 0.389653 i,
 -0.0994489 - 0.453831 i, -0.141824 - 0.545163 i, -0.207186 - 0.686043 i,
 -0.32168 - 0.932817 i, -0.575275 - 1.4794 i, -1.62177 - 3.73497 i,
 3.11485 + 6.47411 i, 0.893868 + 1.68712 i, 0.553235 + 0.952933 i,
 0.414928 + 0.654833 i, 0.339787 + 0.492876 i, 0.292467 + 0.390886 i}
```

```
ListPlot[10 * Re[four], PlotStyle → PointSize[0.01`]]
```



We can generate two-dimensional tables by including two iterators.

```
twot = Table[i/j + i^2, {i, 1, 5}, {j, 1, 2}]
```

$$\left\{\left\{2, \frac{3}{2}\right\}, \{6, 5\}, \left\{12, \frac{21}{2}\right\}, \{20, 18\}, \left\{30, \frac{55}{2}\right\}\right\}$$

It may be useful to use **TableForm** to display multidimensional lists.

```
TableForm[twot]
```

2	$\frac{3}{2}$
6	5
12	$\frac{21}{2}$
20	18
30	$\frac{55}{2}$

The options TableAlignments, TableSpacing, and TableHeadings allow us to customize the appearance of a table.

TableAlignments –> {a_1, a_2, ...} specifies alignments for successive dimensions.

TableSpacing –> {s_1, s_2, ...} specifies that s_i spaces should be left in dimension i.

TableHeadings –> {{lbl_{11}, lbl_{12}, ...}, ...} gives explicit labels for each entry.

```
TableForm[twot, TableSpacing → {6, 4, 2, 0}]
```

```
2     3/2

6     5

12    21/2

20    18

30    55/2
```

```
TableForm[twot, TableSpacing → {6, 4, 2, 0},
 TableHeadings → {{"r1", "r2", "r3", "r4", "r5"}, {"c1", "c2"}}]
```

	c1	c2
r1	2	$\frac{3}{2}$
r2	6	5
r3	12	$\frac{21}{2}$
r4	20	18
r5	30	$\frac{55}{2}$

```
twot
```

$$\left\{\left\{2, \frac{3}{2}\right\}, \{6, 5\}, \left\{12, \frac{21}{2}\right\}, \{20, 18\}, \left\{30, \frac{55}{2}\right\}\right\}$$

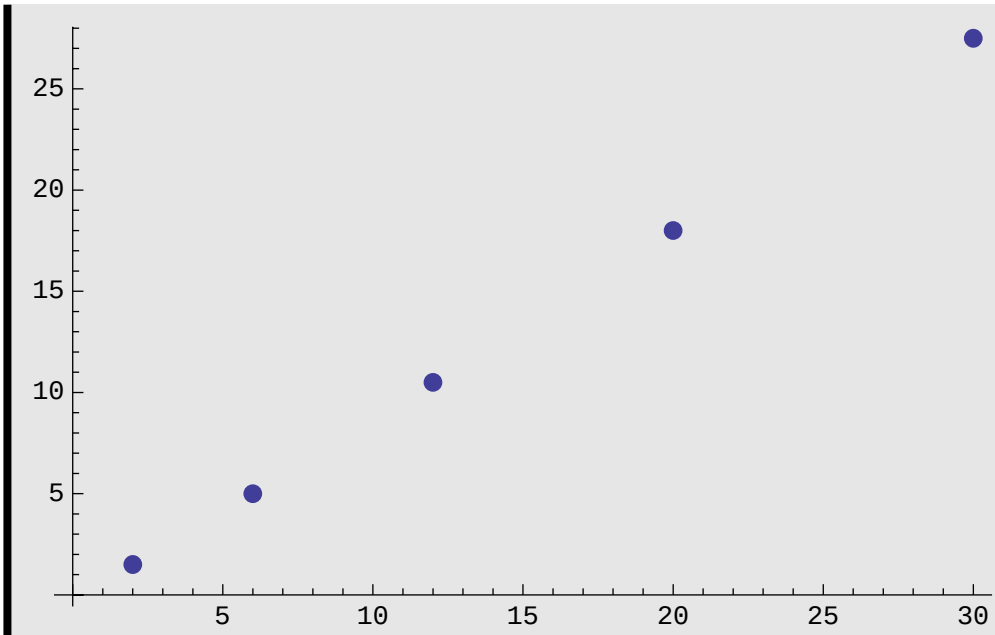
How we interpret an array such as **twot** depends on the context in which it arises: in some cases it may be considered a 5 × 2 matrix, in others a list of five ordered pairs. Here we interpret **twot** as a matrix, and matrix-multiply it by its transpose using **Dot (.)**.

```
twot.Transpose[twot] // MatrixForm
```

$$\begin{pmatrix} \frac{25}{4} & \frac{39}{2} & \frac{159}{4} & 67 & \frac{405}{4} \\ \frac{39}{2} & 61 & \frac{249}{2} & 210 & \frac{635}{2} \\ \frac{159}{4} & \frac{249}{2} & \frac{1017}{4} & 429 & \frac{2595}{4} \\ 67 & 210 & 429 & 724 & 1095 \\ \frac{405}{4} & \frac{635}{2} & \frac{2595}{4} & 1095 & \frac{6625}{4} \end{pmatrix}$$

Here we interpret twot as a list of ordered pairs to be plotted.

```
ListPlot[twot, PlotStyle → PointSize[0.02`]]
```



The first argument to Table can be a conditional statement such as If or Which. An element of the following matrix is 1 if its indices i and j satisfy $|i - j| \leq 2$, and is 0 otherwise.

```
MatrixForm[Table[If[Abs[i - j] ≤ 2, 1, 0], {i, 7}, {j, 7}]]
```

$$\begin{pmatrix} 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 & 1 & 1 \end{pmatrix}$$

There is no need for Mathematica arrays to be rectangular. Here the upper limit of the inner loop j depends on the value of the outer loop i.

```
Table[i - j, {i, 1, 4}, {j, 1, i}]
```

```
{{0}, {1, 0}, {2, 1, 0}, {3, 2, 1, 0}}
```

The result is a triangular array.

```
TableForm[%]
```

```
0
1   0
2   1   0
3   2   1   0
```

We can generate a rank-4 (four-dimensional) tensor by including four iterators in the Table

```
tensor = Table[i + j + k - l, {i, 2}, {j, 3}, {k, 4}, {l, 5}];
TableForm[tensor]
```

```
2 1 0 -1 -2 3 2 1 0 -1 4 3 2 1 0
3 2 1 0  -1 4 3 2 1 0  5 4 3 2 1
4 3 2 1  0  5 4 3 2 1  6 5 4 3 2
5 4 3 2  1  6 5 4 3 2  7 6 5 4 3
3 2 1 0 -1  4 3 2 1 0  5 4 3 2 1
4 3 2 1 0   5 4 3 2 1  6 5 4 3 2
5 4 3 2 1   6 5 4 3 2  7 6 5 4 3
6 5 4 3 2   7 6 5 4 3  8 7 6 5 4
```

It is important to make a distinction between list operations and matrix operations, particularly with respect to matrix multiplication. The function Dot (which we enter as a period .) is used for matrix and vector multiplication. Here is the product of a matrix and a vector.

```
{{a, b}, {c, d}}.{x, y}
```

```
{a x + b y, c x + d y}
```

There is no difference in Mathematica between a row vector and a column vector, as matrix and vector functions automat ically interpret vectors in the appropriate way. Here we multi· ply a matrix by a vector on both the left and the right; in both cases, we type the vector the same way.

```
{x, y}.{{a, b}, {c, d}}.{x, y} // Simplify
```

$$a\,x^2 + y\,(b\,x + c\,x + d\,y)$$

The matrix product of two matrices is.

```
{{1, 2}, {3, 4}}.{{a1, b1}, {a2, b2}} // MatrixForm
```

$$\begin{pmatrix} a1 + 2\,a2 & b1 + 2\,b2 \\ 3\,a1 + 4\,a2 & 3\,b1 + 4\,b2 \end{pmatrix}$$

If we use ordinary multiplication *, Mathematica performs termwise multiplication.

```
{{1, 2}, {3, 4}} * {{a1, b1}, {a2, b2}} // MatrixForm
```

$$\begin{pmatrix} a1 & 2\,b1 \\ 3\,a2 & 4\,b2 \end{pmatrix}$$

A general rule:

if two lists have the same shape, Mathematica will perform termwise arithmetic, and return a list with the same shape.

For example, the difference of two vectors of equal length is a vector of the differences of the elements.

```
{1, 2, 3} - {a1, a2, a3}
```

$$\{1 - a1,\ 2 - a2,\ 3 - a3\}$$

Mathematica will combine a list with a single (scalar) expression.

```
{a1, a2, a3} + c
```

```
{a1 + c, a2 + c, a3 + c}
```

Similarly, multiplying or adding a scalar to a matrix results in the expression being multiplied or added to each element of the matrix.

```
{{1, 2}, {3, 4}} c // MatrixForm
```

$$\begin{pmatrix} c & 2\,c \\ 3\,c & 4\,c \end{pmatrix}$$

```
{{1, 2}, {3, 4}} + k // MatrixForm
```

$$\begin{pmatrix} 1+k & 2+k \\ 3+k & 4+k \end{pmatrix}$$

If two lists cannot be combined because they have different lengths Mathematica produces a warning message

```
{1, 2, 3} + {a1, a2, a3, a4}
```

Thread::tdlen :
Objects of unequal length in {1, 2, 3} + {a1, a2, a3, a4} cannot be combined. ≫

```
{1, 2, 3} + {a1, a2, a3, a4}
```

Use Table to generate a list of the n^{th} prime numbers as k goes from 1 to 10. When we omit a starting value for the iterator n, it is assumed to be 1.(Prime[n]=- gives the n^{th} prime number.)

```
Table[Prime[n], {n, 10}]
```

```
{2, 3, 5, 7, 11, 13, 17, 19, 23, 29}
```

```
Prime[Range[10]]
```

```
{2, 3, 5, 7, 11, 13, 17, 19, 23, 29}
```

If we want to create a vector containing n zeros, where n is an argument of a function called func0

```
func0[n_Integer?Positive] := Table[0, {n}]
func0[10]
```

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

```
zeroMatrix[m_, n_] := Table[0, {m}, {n}]
zeroMatrix[3, 5] // MatrixForm
```

$$\begin{pmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

If we want to generate a 5 × 5 matrix, in which each element is the row number plus the column number, or the number 5, whichever is greater,we use Max (Max[x1,x2,..]=yields the numerically largest of the x_i, or Max[{x1,x2,..},{y1,y2,..},..]=yields the largest element of any of the lists) to determine if 5 or $i + j$ (the row number plus the column number) is greater.

```
Table[Max[{5, i + j}], {i, 5}, {j, 5}] // MatrixForm
```

$$\begin{pmatrix} 5 & 5 & 5 & 5 & 6 \\ 5 & 5 & 5 & 6 & 7 \\ 5 & 5 & 6 & 7 & 8 \\ 5 & 6 & 7 & 8 & 9 \\ 6 & 7 & 8 & 9 & 10 \end{pmatrix}$$

If we want to create the vector `{a x + b y, c x + d y}` using the matrix `{{a, b}, {c, d}}` and the vector `{x, y}` and to compare with the matrix `{{a x, b x}, {c y, d y}}` created with the same matrix and vector, we can illustrate the difference between list and matrix operations. Here are definitions of the matrix and vector.

```
mate = {{a, b}, {c, d}};
vecte = {x, y};
```

For matrix-vector multiplication we use Dot.

```
mate.vecte
```

```
{a x + b y, c x + d y}
```

To perform termwise multiplication we use Times.

```
mate * vecte // MatrixForm
```

$$\begin{pmatrix} a\,x & b\,x \\ c\,y & d\,y \end{pmatrix}$$

We can compute the eigenvectors of a numeric approximation to the matrix and see if they are normalized (that is, see if each eigenvector has unit length).

```
eigvect = Eigenvectors[N[{{1, 2, 3}, {3, 2, 2}, {7, 1, 2}}]]; TableForm[eigvect]
```

```
-0.483239  -0.517864  -0.705901
-0.59605    0.0356482  0.802156
0.0106728  -0.832608   0.55376
```

The function that can determines the length of a vector is

```
veclength[z_?VectorQ] := √(z.z)
```

We can use Table to test if each eigenvector has unit length or the high-level programming function Map (Map[f,expr]=applies f to each element on the first level in *expr* or f/@ expr) to map veclength over each vector in eigvect.

```
Table[veclength[eigvect[[i]]], {i, 3}]
```

```
{1., 1., 1.}
```

```
Map[veclength, eigvect]
```

```
{1., 1., 1.}
```

```
veclength /@ eigvect
```

```
{1., 1., 1.}
```

There are several functions which report the structure of lists.

Length[list] returns the number of elements in the first level of a list, where any sublists are counted just once like

any other element would be counted.

Dimensions[list] reports the structure of a list. If the structure is uniform, a list of lengths is returned for each sublist, but if the structure is irregular the report stops at the deepest regular level

TensorRank[expr] returns the tensor rank of an expression, which is interpreted as the depth to which an expression is represented by sublists of the same length..

The simplest types of lists are vectors, matrices, and tensors. A vector is a simple list of one or more objects. A matrix is a list of vectors of the same length. A tensor is a list of matrices with the same dimensionality. Note that a matrix is a special case of a tensor of rank 2, a vector is a tensor of rank 1, and a scalar is tensor of rank 0.

```
Vect1 = {x, y, z};
Matrix1 = {{a1, a2, a3, a4}, {a5, a6, a7, a8} };
Tensor1 = {{{a1, a2, a3}, {a4, a5, a6}},
   {{a7, a8, a9}, {a10, a11, a12}} };
```

```
Length /@ {Vect1, Matrix1, Tensor1}
```

```
{3, 2, 2}
```

```
Dimensions /@ {Vect1, Matrix1, Tensor1}
```

```
{{3}, {2, 4}, {2, 2, 3}}
```

```
TensorRank /@ {a1, Vect1, Matrix1, Tensor1}
```

```
{0, 1, 2, 3}
```

VectorQ[expr] returns **True** if **expr** has the structure of a vector, and **False** otherwise. Similarly, **MatrixQ** and **TensorQ** test whether an expression has the structure of either a matrix or a tensor.

```
VectorQ /@ {Vect1, Matrix1, Tensor1}
```

```
{True, False, False}
```

```
MatrixQ /@ {Vect1, Matrix1, Tensor1}
```

```
{False, True, False}
```

```
TensorQ /@ {Vect1, Matrix1, Tensor1}
```

```
{True, True, True}
```

A list with nonuniform structure is not a tensor.

```
TensorQ[{{{a1, a2, a3}, {a4, a5, a6}}, {{a7, a8, a9}, {a10, a11, a12}}, a14}]
```

```
False
```

A vector can be printed in **ColumnForm** or in **MatrixForm**, but you will need to write your own display function for more general tensors.

```
ColumnForm[Vect1]
```

```
x
y
z
```

```
MatrixForm[Vect1]
```

$$\begin{pmatrix} x \\ y \\ z \end{pmatrix}$$

```
MatrixForm[Matrix1]
```

$$\begin{pmatrix} a1 & a2 & a3 & a4 \\ a5 & a6 & a7 & a8 \end{pmatrix}$$

Generic arrays can be created using **Array[name,dimensions]** where **name** is the label applied to each element and **dimensions** is a list of dimensions for the array.

```
Array[p, {4}]
```

```
{p[1], p[2], p[3], p[4]}
```

```
Array[r, {7, 5}] // MatrixForm
```

$$\begin{pmatrix} r[1,1] & r[1,2] & r[1,3] & r[1,4] & r[1,5] \\ r[2,1] & r[2,2] & r[2,3] & r[2,4] & r[2,5] \\ r[3,1] & r[3,2] & r[3,3] & r[3,4] & r[3,5] \\ r[4,1] & r[4,2] & r[4,3] & r[4,4] & r[4,5] \\ r[5,1] & r[5,2] & r[5,3] & r[5,4] & r[5,5] \\ r[6,1] & r[6,2] & r[6,3] & r[6,4] & r[6,5] \\ r[7,1] & r[7,2] & r[7,3] & r[7,4] & r[7,5] \end{pmatrix}$$

Extracting parts of Lists

We can extract an element according to its position by using **Part**. In Mathematica, the first element of a list is at position 1, not 0.

```
Part[{a1, a2, a3}, 1]
Part[{a1, a2, a3}, 2]
Part[{a1, a2, a3}, 3]
```

```
a1
```

```
a2
```

```
a3
```

A compact, equivalent way to express the same command is to enclose the desired position between [[and]].

```
{a1, a2, a3}[[1]]
{a1, a2, a3}[[2]]
{a1, a2, a3}[[3]]
```

```
a1
```

```
a2
```

```
a3
```

In a multidimensional array, an element of a list can be another list. For example, here is a table of a class withe names, and schoolcodeclass, with each triple collected in a sublist.

```
class =
    {{"Allison",   15,  987},
     {"Barry",     16,  995},
     {"Chantal",   17,  992},
     {"Dean",      14,  999},
     {"Erin",      15,  975},
     {"Felix",     16,  950},
     {"Gabrielle", 17,  998},
     {"Humberto",  14,  970},
     {"Iris",      18,  970},
     {"Jerry",     20, 1003}};
     TableForm[class, TableSpacing → {0, 3}]
```

```
Allison      15    987
Barry        16    995
Chantal      17    992
Dean         14    999
Erin         15    975
Felix        16    950
Gabrielle    17    998
Humberto     14    970
Iris         18    970
Jerry        20    1003
```

To extract an entire sublist, or record, we specify the position of the record using Part (or the part-specification brackets [[ ]]).

```
Part[class, 2]
```

```
{Barry, 16, 995}
```

To extract a single element of a two-dimensional list (we refer to its coordinate, its row and column) Part[list,coord,column)

```
Part[class, 5, 2]
```

```
15
```

A compact or equivalent ways are

```
A are compact equivalent or ways
```

```
class[[5, 2]]
class[[5]][[2]]
{class[[5, 2]], class[[5]][[2]]}
```

```
15
```

```
15
```

```
{15, 15}
```

Similarly, we extract the schoolcodeclass of Gabrielle by extracting the sublist at position 7, then extracting the third part of Gabrielle's record.

```
class[[7, 3]]
```

```
998
```

To extract elements from deeply nested lists, it may be necessary to specify several coordinates.

It is important to be aware that there is a difference between the part specification class[[1,2]] and class[[{1,2}]] (the latter has curly brackets inside the double square brackets). The former returns the element at position (1, 2) of the list hurr.

```
class[[1,2]]
```

```
15
```

The latter is equivalent to {class[[1]], class[[2]]}.

```
class[[{1, 2}]]
```

```
{{Allison, 15, 987}, {Barry, 16, 995}}
```

Lists need not have a uniform structure — the sublists may vary in length and the type of contents.

```
references = {{{"Allison", "girl"}, {"Age", 15}, {"Code", 987}},
   {{"Barry", "girl"}, {"Age", 16}, {"Code", 995}},
   {{"Chantal", "girl"}, {"Age", 17}, {"Code", 992}},
   {{"Dean", "boy"}, {"Age", 14}, {"Code", 999}},
   {{"Erin", "boy"}, {"Age", 15}, {"Code", 975}},
   {{"Felix", "girl"}, {"Age", 16}, {"Code", 950}}};
```

```
Length[references]
```

```
6
```

```
Dimensions[references]
```

```
{6, 3, 2}
```

```
TensorRank[references]
```

```
3
```

```
TableForm[references[[3]], TableAlignments → {Left, Top}]
```

```
Chantal girl
Age     17
Code    992
```

```
TableForm[references[[5]], TableAlignments → {Left, Top}]
```

```
Erin boy
Age  15
Code 975
```

[[{ }]] notation provides a quick way to manipulate the elements of a list. Here is an example of using this notation to rearrange the elements of a list.

```
list1 = {a, b, c, d, e, f};
```

```
list1[[{1, 4, 2, 5, 3, 6}]]
```

```
{a, d, b, e, c, f}
```

We can define a function called testWithReplacement that takes a sample of a given size from a list of data using Table and Part.

```
testWithReplacement[L_List, n_Integer?Positive] :=
 Module[{ℓ = Length[L]}, Table[L[[RandomInteger[{1, ℓ}]]], {n}]]
```

```
testWithReplacement[{a1, a2, a3, a4, a5}, 10]
```

```
{a3, a5, a4, a2, a4, a3, a5, a2, a1, a4}
```

Selecting elements by position

First[list] returns the first element, **Rest[list]** returns everything but the first element, and **Last[list]** returns the last element of a list.

```
First[{a, b, c, d}]
```

```
a
```

```
First[{{a, b}, {c, d}, {e, f}}]
```

```
{a, b}
```

```
Rest[{a, b, c, d}]
```

```
{b, c, d}
```

```
Last[{a, b, c, d}]
```

```
d
```

We can extract a range of elements using **Take**.

Take[list,n]=gives the first *n* elements of *list*

Take[list,-n]=gives the last *n* elements of *list*.

Take[list,{m,n}]=gives elements *m* through *n* of *list*

Here we extract the first five elements of a list.

```
Take[{a1, a2, a3, a4, a5, a6}, 3]
```

```
{a1, a2, a3}
```

```
Take[{a1, a2, a3, a4, a5, a6}, -3]
```

```
{a4, a5, a6}
```

```
Take[{a1, a2, a3, a4, a5, a6}, {3, 6}]
```

```
{a3, a4, a5, a6}
```

Drop works similarly, removing elements from a list. Here we drop the first three elements from a list.

Drop[list,n]=gives *list* with its first *n* elements dropped.

Drop[list,-n]=gives *list* with its last *n* elements dropped.

Drop[list,{n}]=gives *list* with its n^{th} element dropped.

Drop[list,{m,n}]=gives *list* with elements *m* through *n* in steps of *s* dropped.

```
Drop[{a1, a2, a3, a4, a5, a6, a7}, 4]
```

```
{a5, a6, a7}
```

```
Drop[{a1, a2, a3, a4, a5, a6, a7}, -4]
```

```
{a1, a2, a3}
```

```
Drop[{a1, a2, a3, a4, a5, a6, a7}, {4}]
```

```
{a1, a2, a3, a5, a6, a7}
```

Note that Take and Drop do not alter the value of a list.

```
list2 = {a1, a2, a3, a4, a5, a6, a7}
```

```
{a1, a2, a3, a4, a5, a6, a7}
```

```
Drop[list2, {3, 5}]
```

```
{a1, a2, a6, a7}
```

```
list2
```

```
{a1, a2, a3, a4, a5, a6, a7}
```

To alter the value of list2, we must set the value of list2 to the desired subset.

```
list2 = Drop[list2, {3, 5}];
```

```
list2
```

```
{a1, a2, a6, a7}
```

Here is a function called **testWithoutReplacement** that takes a given number of samples from a list, without choosing the same element more than once.

```
testWithoutReplacement[L_List, n_Integer?Positive] :=
 Module[{ℒ = L, ℓ = Length[L], pos},
  Table[pos = RandomInteger[{1, ℓ}]; ℒ = Drop[ℒ, {pos}]; ℓ = ℓ - 1;
   L[[pos]], {n}]]
```

Here is a sample of fifteen data points from a list of integers from 1 to 50.

```
testWithoutReplacement[Range[50], 15]
```

```
{9, 17, 28, 43, 38, 13, 5, 37, 31, 29, 9, 21, 25, 3, 14}
```

Selecting elements by value or pattern

Elements satisfying some criterion can be selected using **Select[list,criterion]** where **criterion** is a logical function acting on the elements of the list one at a time. The result is a list of elements for which the criterion function tests **True**, with ambiguous cases omitted. Note that **criterion** must be formulated as a pure function with one slot.

```
Select[{1, a, 2, 3, -3, "Grammy"}, OddQ]
```

```
{1, 3, -3}
```

```
Select[{1, a, 2, 3, -3, "Grammy"}, # > 2 &]
```

```
{3}
```

```
Select[{1, a, 2, 3, -3, "Grammy"}, StringQ]
```

```
{Grammy}
```

```
Select[{1, a, 2, 3, -3, "Grammy", "abc"},
 StringQ[#] && StringLength[#] > 4 &]
```

```
{Grammy}
```

Similarly, a list of elements matching a pattern is produced using **Cases[list,pattern]**.

```
Cases[{1, a, 2, 3, -3, "Grammy"}, _?OddQ]
```

```
{1, 3, -3}
```

```
Cases[{1, a^5, 2, 3, -3, "Grammy"}, x_^n_]
```

```
{a^5}
```

Note that because we do not use the parts of the part, it is not necessary to name them. Thus, the preceding example can be be written as

```
Cases[{1, q^5, 2, 3, -3, "Grammy"}, _^_]
```

```
{q^5}
```

without naming either the base or the exponent in the pattern for something to some power.

Naturally, `DeleteCases[list,pattern]` removes elements matching a pattern

```
DeleteCases[{1, q^5, 2, 3, -3, "Grammy"}, _?StringQ]
```

```
{1, q^5, 2, 3, -3}
```

whereas `Count[list,pattern]` just returns the number of elements that match a pattern.

```
Count[{1, q^5, 2, 3, -3, "Grammy"}, _?StringQ]
```

```
1
```

Sorting

The elements of a list can be sorted according to an ordering function that acts pairwise in the list until satisfied by all successive pairs using the function **Sort[list,criterion]**. If the ordering function is omitted, the internal "canonical" ordering function is applied.

```
Sort[{5, -3, -4, 79, 27, 0, -28, 2}, #2 > #1 &]
```

```
{-28, -4, -3, 0, 2, 5, 27, 79}
```

However, sorting stops if ambiguous results are obtained, resulting in partial but incomplete ordering.

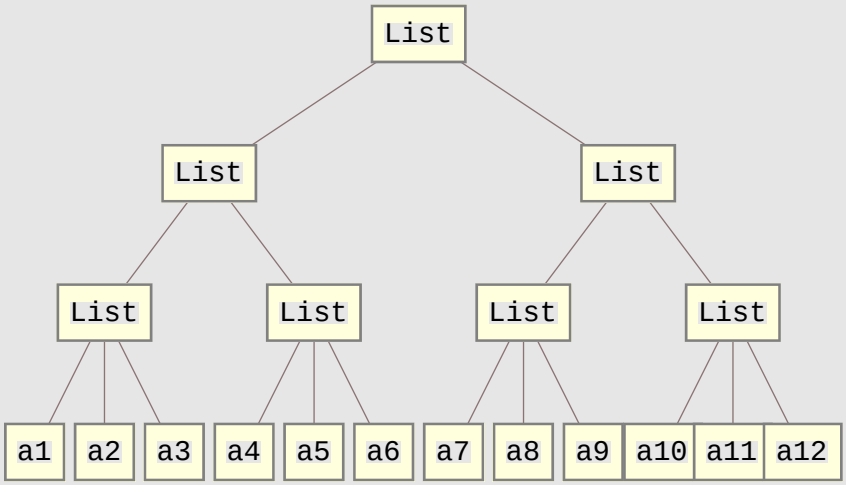
```
Sort[{5, -3, -4, 79, 27, 0, -28, 2}, #2 > #1 &]
```

```
{-28, -4, -3, 0, 2, 5, 27, 79}
```

Restructuring

Sometimes it is necessary to regroup the elements of a list into another list with a different structure. A list containing sublists can be visualized as an inverted tree using **TreeForm**.

```
TreeForm[Tensor1]
```

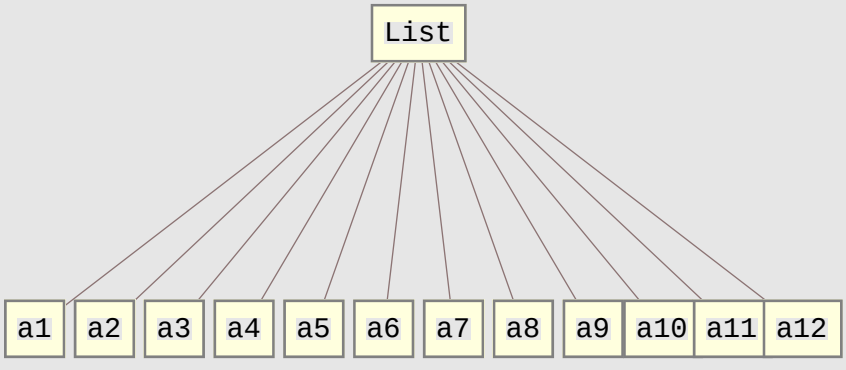


The structure of a list can be "flattened" to a single level using **Flaten**.

```
Flatten[Tensor1]
```

```
{a1, a2, a3, a4, a5, a6, a7, a8, a9, a10, a11, a12}
```

```
TreeForm[%]
```

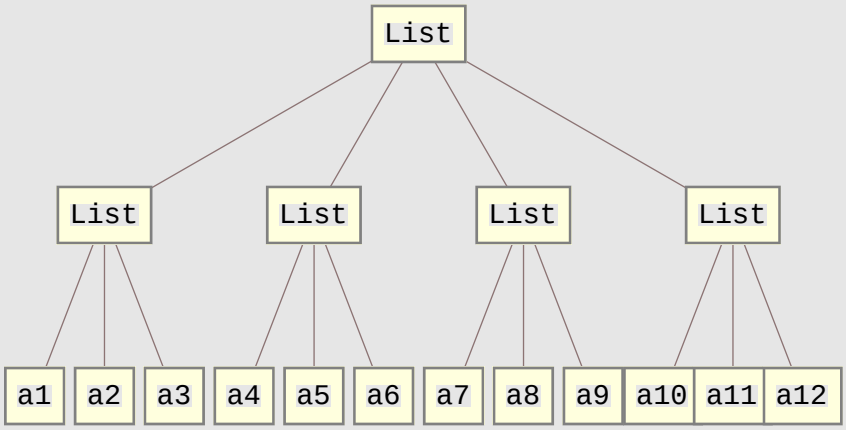


Partial flattening to level `n` is obtained using `Flatten[list,n]`.

```
Flatten[Tensor1, 1]
```

```
{{a1, a2, a3}, {a4, a5, a6}, {a7, a8, a9}, {a10, a11, a12}}
```

```
TreeForm[%]
```



FlatenAt[list,position] flattens a sublist at a specified **position** of **list**. The position can be a sequence list or a list of sequence lists.

```
FlattenAt[{a, {b, c, d}, e, f, {g, h}}, 2]
```

```
{a, b, c, d, e, f, {g, h}}
```

The elements of a flattened list can be regrouped into sublists with **n** elements each using **Partition[list,n]**. Note that any elements left over which do not comprise a complete sublist are dropped.

```
Flatten[Tensor1] // Partition[#, 5] &
```

```
{{a1, a2, a3, a4, a5}, {a6, a7, a8, a9, a10}}
```

```
Flatten[Tensor1] // Partition[#, 2] &
```

```
{{a1, a2}, {a3, a4}, {a5, a6}, {a7, a8}, {a9, a10}, {a11, a12}}
```

```
Nest[Partition[#, 2] &, Flatten[Tensor1], 2]
```

```
{{{a1, a2}, {a3, a4}}, {{a5, a6}, {a7, a8}}, {{a9, a10}, {a11, a12}}}
```

If **Partition[list,n,d]** is supplied another argument, an offset *d* is used each time a sublist is created. The offset gives the difference in position between the first elements of successive sublists. Thus, if the offset is *d*, the first sublist begins at position 1, the second at position $1 + d$, the third at position $1 + 2\,d$, etc.

```
Partition[{a, b, c, d, e, f, g, h, i, j}, 3, 1]
```

```
{{a, b, c}, {b, c, d}, {c, d, e},
 {d, e, f}, {e, f, g}, {f, g, h}, {g, h, i}, {h, i, j}}
```

```
Partition[{a, b, c, d, e, f, g, h, i, j}, 3, 2]
```

```
{{a, b, c}, {c, d, e}, {e, f, g}, {g, h, i}}
```

Characteristics of elements

Often it is necessary to select elements from a list according to certain criteria, rather than according to position. The **Select** function returns the elements of a given list that meet a specified criterion.or example, one way to extract all of the prime numbers less than 100 is first to generate a list of integers from 1 to 100, then select from that list all the elements that satisfy the predicate function **PrimeQ**.

PrimeQ[1] gives False.

PrimeQ[−*n*], where *n* is prime, gives True.

PrimeQ[*n*, GaussianIntegers –> True] determines whether *n* is a Gaussian prime.

PrimeQ[*m* + I *n*] automatically works over the Gaussian integers.

Simplify[*expr* ∈ Primes] can be used to try to determine whether a symbolic expression is mathematically a prime.

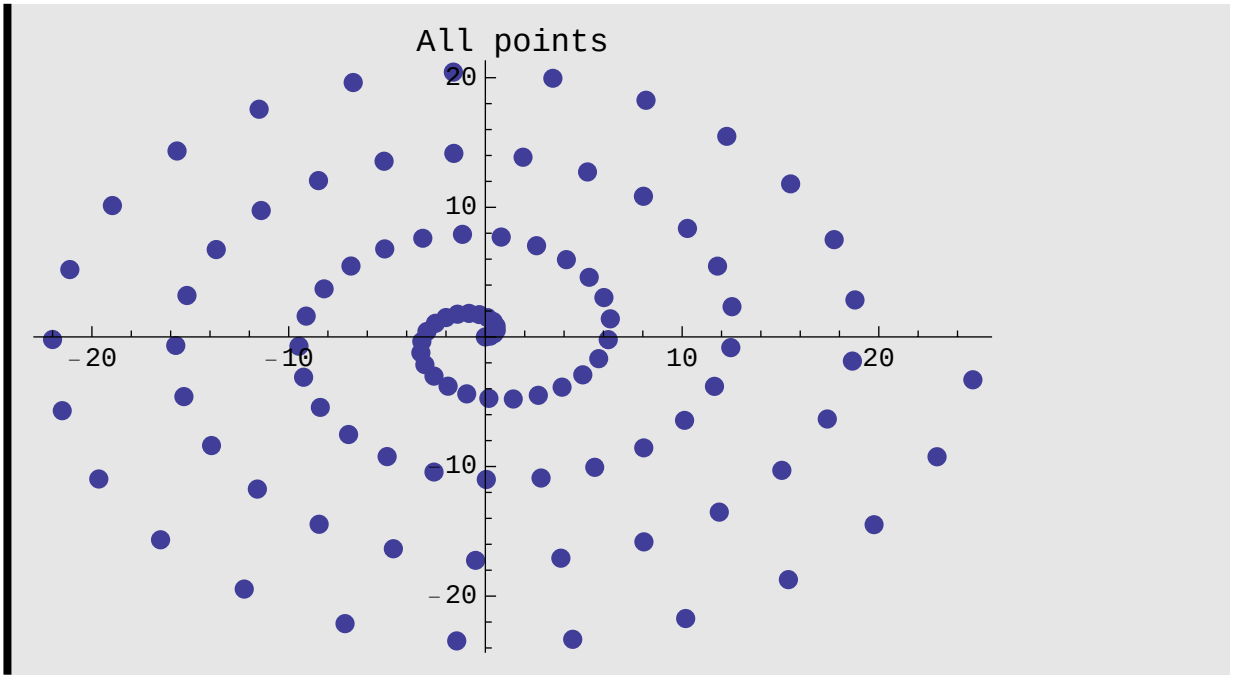
```
Select[Range[100], PrimeQ]
```

```
{2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37,
 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97}
```

Similarly we can choose from a list of ordered pairs all pairs that satisfy some specified criteria. Here is a list of points along a spiral.

```
spiralp = Table[{r Cos[r], r Sin[r]}, {r, 0, 25, 0.25}];
```

```
spiplot = ListPlot[spiralp, PlotStyle → PointSize[0.02`],
  PlotLabel → "All points"]
```



Consider a predicate function that acts on pairs of numbers that returns **True** if its first element times its second element is greater than zero.

```
pred1[{x_, y_}] := (x y > 0)
```

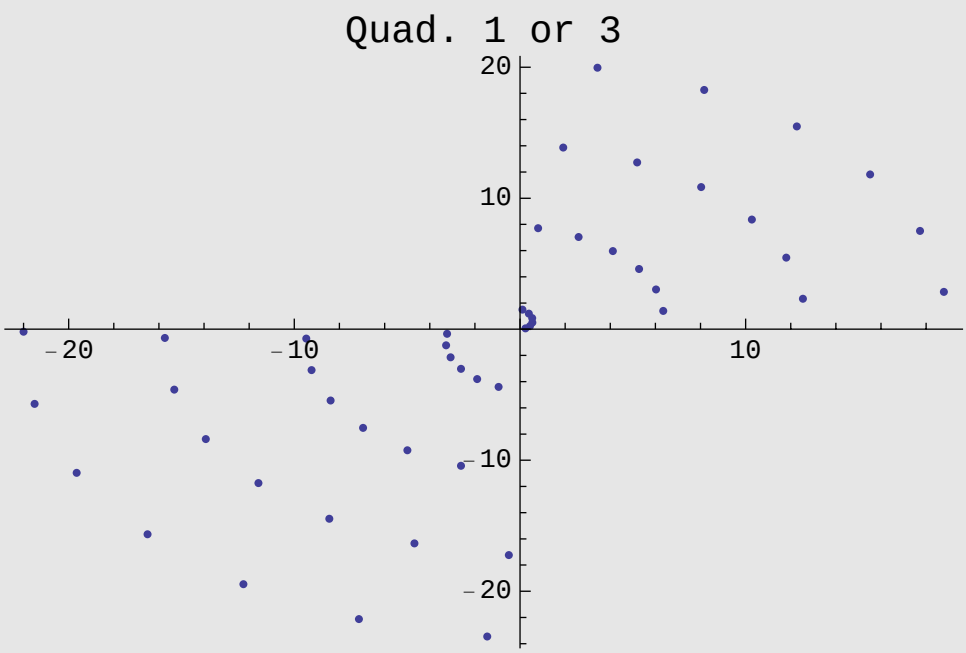
Consider a second predicate that returns **True** if its second element is greater than or equal to its first element.

```
pred2[{x_, y_}] := (y >= x)
```

(We want each of the following graphs to use the same option settings, and **SetOptions** allows us to change the default settings. We will change the settings back to their original values after this example.)

```
PQ1 = Select[spiralp, pred1];
```

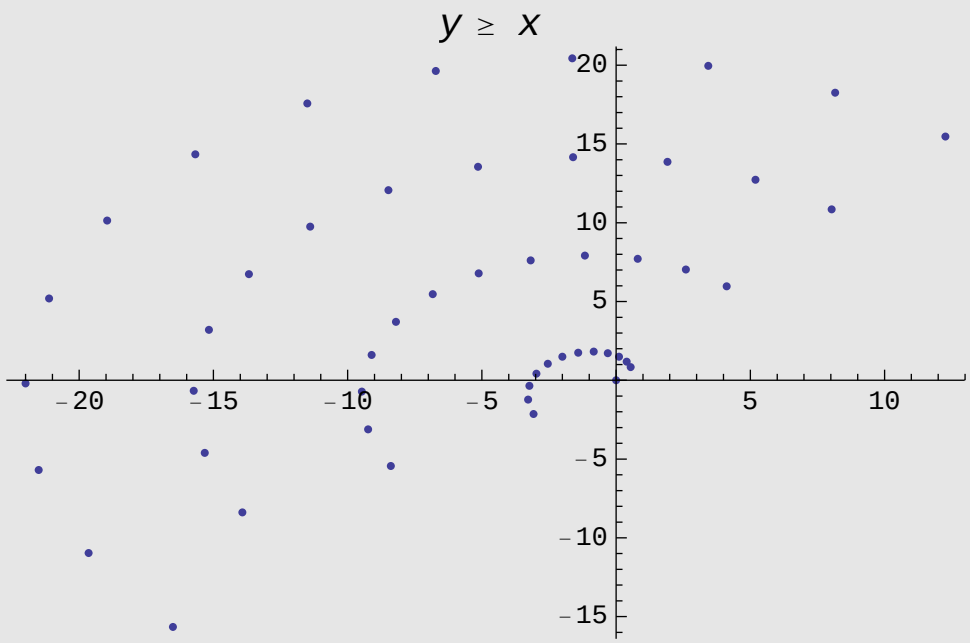
```
plPQ1 = ListPlot[PQ1, PlotLabel → "Quad. 1 or 3"]
```



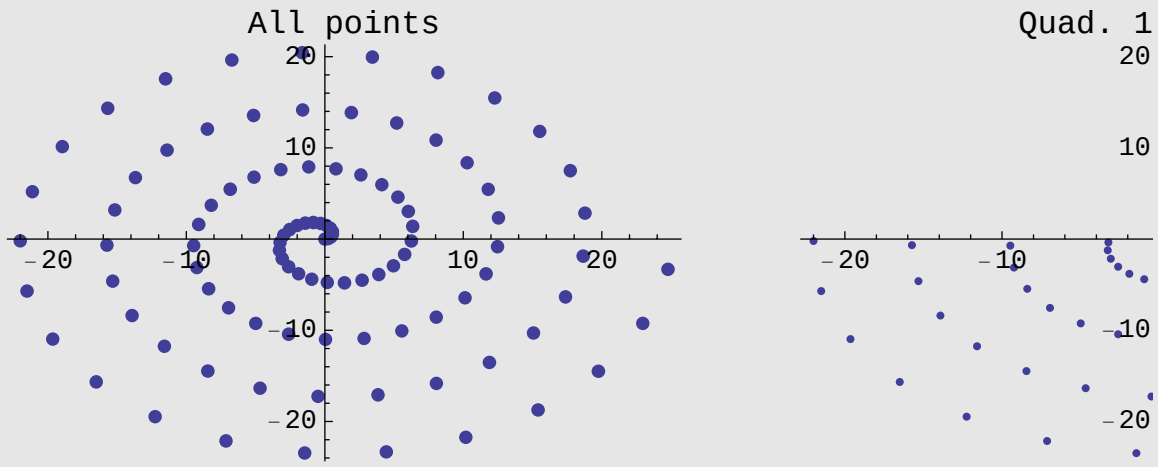
The last plot is of the points for which the y value is greater than or equal to the x value.

```
PQ2 = Select[spiralp, pred2];
```

```
plPQ2 = ListPlot[PQ2, PlotLabel → "" y ≥ x ""]
```

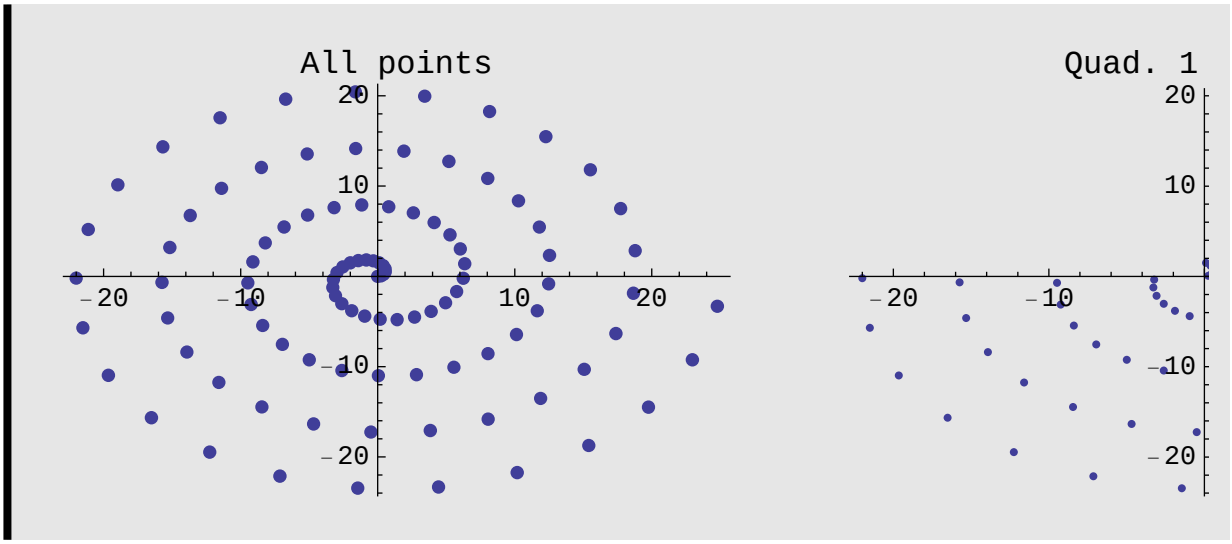



```
Show[GraphicsRow[{spiplot, plPQ1, plPQ2}]]
```




```
SetOptions[ListPlot, AspectRatio → GoldenRatio^-1, PlotRange → Automatic,
  PlotStyle → Automatic, DisplayFunction :> $DisplayFunction,
  Ticks → Automatic];
```

```
Show[GraphicsRow[{spiplot, plPQ1, plPQ2}]]
```



We can test whether an element is present in a list by using **MemberQ** and **FreeQ.**