[3.16] Use arrays with rank greater than 2

(This documentation is taken from the TI89/92+ tip list, maintained by Doug Burkett. For the complete tip list, see http://www.geocities.com/TI_TipList/)

The 89/92+ can manipulate 1-dimensional data structures (lists and vectors), and 2-dimensional data structures (matrices and data variables). The number of dimensions is called the *rank*. Lists and vectors have a rank of 1. Matrices and data variables have a rank of 2. Matrices with rank greater than two are useful in many cases. For example, arrays of rank 3 can represent fields and other physical models. Arrays of rank 3 and 4 can be used to represent tensors.

This tip shows how to create and store arrays of any rank on the 89/92+, and how to store and recall elements of those arrays. It also shows how many built-in list functions can be used to manipulate high-rank arrays, and gives a program that finds the transpose of a rank-3 array.

I use the built-in list data structure to store the arrays, because a list can represent an array of any desired rank. First, consider a 3-dimensional array with dimensions $\{d_1, d_2, d_3\}$. An element's location in this array is defined as $\{a_1, a_2, a_3\}$, where

$$1 \le a_1 \le d_1 \qquad 1 \le a_2 \le d_2 \qquad 1 \le a_3 \le d_3$$

For example, a 2x3x4 matrix would have d = {2,3,4}, and an element in the matrix might be located at a = {2,1,3}.

To create an array *mat1* with all zero elements, use

```
newlist(d1*d2*d3)→mat1
```

For example, to create a 2 x 3 x 4 array, use

```
newlist(24)→mat1
```

The array elements are saved in the list such that the last index changes most rapidly as we move towards the end of the list. If we have a 2x2x2 array, and we label the elements as $e_{a1,a2,a3}$, then the list would look like this:

$$\{e_{1,1,1}\ e_{1,1,2}\ e_{1,2,1}\ e_{1,2,2}\ e_{2,1,1}\ e_{2,1,2}\ e_{2,2,1}\ e_{2,2,2}\}$$

For a particular element location $\{a_1, a_2, a_3\}$, we can find the corresponding list element index *k* from

$$k = a_3 + (a_2 - 1)d_3 + (a_1 - 1)d_2d_3 \qquad [1]$$

This can be expanded and factored to eliminate one multiply, and access each variable only once, like this:

$$k = d_3(d_2(a_1 - 1) + a_2 - 1) + a_3 \qquad [2]$$

The routines to save and recall elements to a 3-dimensional list are called *m3sto()* and *m3rcl(). m3sto()* looks like this:

```
m3sto(v,l,d,m)
Func
©(value,location{},dim{},matrix{})
```

```
©Store value in 3D matrix
©12oct00 dburkett@infinet.com

v→m[d[3]*(d[2]*(l[1]-1)+l[2]-1)+l[3]]
m

EndFunc
```

where

v	is the value to store in the array
l	is a 3-element list that specifies the element location as {a_1, a_2, a_3}
d	is a 3-element list that specifies the list dimensions as {d_1, d_2, d_3}
m	is the array list or the name of the array list

m3sto() returns the original array *m* with value *v* stored at location *l*.

For example, if we want to put 7 at location {1,2,3} in array *mat1*, which has dimensions 2 x 3 x 4, then use this:

```
m3sto(7,{1,2,3},{2,3,4},mat1)→mat1
```

To recall an array element, use *m3rcl():*

```
m3rcl(l,d,m)
Func
©(location{},dim{},matrix{})
©Recall element from 3D matrix
©12oct00 dburkett@infinet.com

m[d[3]*(d[2]*(l[1]-1)+l[2]-1)+l[3]]

EndFunc
```

where

l	is a 3-element list that specifies the element location as {a_1, a_2, a_3}
d	is a 3-element list that specifies the list dimensions as {d_1, d_2, d_3}
m	is the array list or the name of the array list

m3rcl() returns a scalar result. To get the element at location {1,2,3} in a 2 x 3 x 4 array *mat1*, and save it in the variable *var1*, use

```
m3rcl({1,2,3},{2,3,4},mat1)→var1
```

The index formula can be extended as needed for arrays of higher dimensions. For example, for a 4-dimensional array, the index formula is

$$k = a_4 + (a_3 - 1)d_4 + (a_2 - 1)d_3d_4 + (a_1 - 1)d_2d_3d_4 \qquad [3]$$

This is expanded and factored to

$$k = d_4(d_3(d_2(a_1 - 1) + a_2 - 1) + a_3 - 1) + a_4 \qquad [4]$$

This index formula is coded in routines *m4sto()* and *m4rcl(),* which are not shown here but are included in the tip list code file *tlcode.zip*. The calling arguments are identical to *m3sto()* and *m3rcl(),* but note

that the the location and dimension lists have dimension 4, since the routines work on a 4-dimensional array.

In general, for an n-dimensional array, you will sum *n* terms to find *k*. The terms are

$$
\begin{aligned}
k = {} & a_n + && \text{(1st term)} \\
& (a_{n-1} - 1)d_n + && \text{(2nd term)} \\
& (a_{n-2} - 1)d_n d_{n-1} + && \text{(3rd term)} \\
& (a_{n-3} - 1)d_n d_{n-1} d_{n-2} + && \text{(4th term)} \\
& (a_{n-4} - 1)d_n d_{n-1} d_{n-2} d_{n-3} + \ldots && \text{(5th term)}
\end{aligned}
\qquad [5]
$$

To find the simplified form of the sum for arrays with rank greater than 4, use equation [4] as a pattern. Begin by writing a nested product of the 'd' terms, beginning with d_n, and stop when you reach d_2. I'll use a 5-dimension array as an example, and the simplified form looks like this, so far:

$$k = d_5(d_4(d_3(d_2(\ldots$$

Next, start adding a_k - 1 terms, and closing the parentheses as you go. Begin with a_1:

$$k = d_5(d_4(d_3(d_2(a_1 - 1) + \ldots$$

and continue until you reach a_{n-1}:

$$k = d_5(d_4(d_3(d_2(a_1 - 1) + a_2 - 1) + a_3 - 1) + a_4 - 1)\ldots$$

and finally add the a_n term to the nested product:

$$k = d_5(d_4(d_3(d_2(a_1 - 1) + a_2 - 1) + a_3 - 1) + a_4 - 1) + a_5$$

This is the simplified form.

This function shows the general expression to find the list index *k* for an element location $\{a_1,a_2,\ldots,a_n\}$ for an array of dimension $\{d_1,d_2,\ldots d_n\}$:

```
mrankni(a,d)
Func
©({loc},{dim}) return index
©Return list index for array element
©30nov00/dburkett@infinet.com; based on expression by Bhuvanesh Bhatt

Σ(when(i≠dim(a),(a[i]-1)*Π(d[j],j,i+1,dim(d)),a[i]),i,1,dim(a))

EndFunc
```

For example, to find the list index for the element at {1,2,3} in a 3x3x3 array, use

```
mrankni({1,2,3},{3,3,3})
```

which returns 6, meaning that the element at {1,2,3} is the 6th element in the list.

This function can be used for arrays of any rank, but it is much slower than the direct implementations given above. For a 4x4x4x4 array, the direct expression evaluates in about 0.037 seconds to find the list index *k*, while the general expression takes about 0.208 seconds.

Even though the general expression is slow, it is useful to find a symbolic expression for the list index. For example, to find the index expression for a rank-5 array, set

```
{a[1],a[2],a[3],a[4],a[5]}→a
{d[1],d[2],d[3],d[4],d[5]}→d
```

Then, executing *mrankni()* returns

```
a5+(a[4]+a[3]*d[4]+a[2]*d[3]*d[4]+a[1]*d[2]*d[3]*d[4]-
((d[2]+1)*d[3]+1)*d[4]-1)*d[5]
```

The expression is not optimized in this form but it can be optimized by factoring on *d* with factor():

```
factor(a5+(a[4]+a[3]*d[4]+a[2]*d[3]*d[4]+a[1]*d[2]*d[3]*d[4]-
((d[2]+1)*d[3]+1)*d[4]-1)*d[5],d)
```

which returns the optimized form

```
(((d[2]*(a[1]-1)+a[2]-1)*d[3]+a[3]-1)*d[4]+a[4]-1)*d[5]+a[5]
```

For some array operations, you may need to find the array element address {a}, given the list index. The three routines below implement this function. *m3aind()* finds the element address for a rank-3 array, and *m4aind()* finds the address for a rank-4 array. *mnaind()* finds the address for an array of any rank. It is slower than *m3aind()* and *m4aind(),* but obviously more general.

Find element address for a rank-3 array:

```
m3aind(i,d)
Func
©(index,{d1,d2,d3}) return {a1,a2,a3}
©Rank 3 matrix
©25oct00 dburkett@infinet.com

local a1,a2

intdiv(i-1,d[2]*d[3])+1→a1
intdiv(i-1-(a1-1)*d[2]*d[3],d[3])+1→a2
{a1,a2,i-d[3]*(d[2]*(a1-1)+a2-1)}

EndFunc
```

Find element address for a rank-4 array:

```
m4aind(i,d)
Func
©(index,{d1,d2,d3,d4}) return {a1,a2,a3,a4}
©Rank 4 matrix
©25oct00 dburkett@infinet.com

local a1,a2,a3
```

```
intdiv(i-1,d[2]*d[3]*d[4])+1→a1
intdiv(i-1-(a1-1)*d[2]*d[3]*d[4],d[3]*d[4])+1→a2
intdiv(i-1-((a1-1)*d[2]*d[3]*d[4]+(a2-1)*d[3]*d[4]),d[4])+1→a3
{a1,a2,a3,i-d[4]*(d[3]*(d[2]*(a1-1)+a2-1)+a3-1)}

EndFunc
```

Find element address for array of any rank:

```
mnaind(i,d)
Func
©mnaind(i,dims) returns the array location of ith list element
©Bhuvanesh Bhatt, Nov 2000

Local j,k,l,jj,a,dd

dim(d)→dd
1→d[dd+1]
1+dd→dd
true→jj

For l,1,dd
 d[l]→k
 If getType(k)≠"NUM" then
  false→jj
  Exit
 EndIf
EndFor

If getType(i)="NUM" and jj and i>product(d):Return {}

For l,1,dd-1
when(l≠1,intDiv(i-1-Σ((a[j]-1)*Π(d[jj],jj,j+1,dd),j,1,l-1),Π(d[k],k,l+1,dd)),int
Div(i-1,Π(d[jj],jj,2,dd)))+1→a[l]
EndFor

a

EndFunc
```

As an example, use *m3aind()* to find the element address of the 7th element of a 3x3x3 array:

```
m3aind(7,{3,3,3})
```
returns {1,3,1}

This example for *m4aind()* finds the 27th element of a 4x4x4x4 array:

```
m4aind(27,{4,4,4,4})
```
returns {1,2,3,3}

This example for *mnaind()* finds the address for the 40th element of a 5x4x3x2x1 array:

```
mnaind(40,{5,4,3,2,1})
```
returns {2,3,2,2,1}

Note that *mnaind()* returns an empty list if an error condition occurs; your calling program can test for this condition to verify proper operation.

mnaind() is especially useful for finding the general expression for the addresses of elements for arrays of any rank. For example, use this call to *mnaind()* for a rank-5 array:

```
mnaind(k,{d[1],d[2],d[3],d[4],d[5]})
```

This will only work properly if *k* and the list *d* are not defined variables. The expression returned is quite lengthy (and not shown here), but it *is* correct.

The 89/92+ do not have built-in functions for higher-dimension arrays, but the simple list storage method means that simple array operations are trivial. These examples show operations on two arrays *m1* and *m2*, with the result in array *m3*. *k* is a constant or expression. The comment in parentheses shows the equivalent built-in 89/92+ array function. In general, *m1* and *m2* must have the same dimensions.

Add two arrays (equivalent to .+)	`m1+m2→m3`
Add an expression to each element (equivalent to .+)	`k+m1→m3`
Subtract arrays (equivalent to .-)	`m1-m2→m3`
Subtract an expression from each element (equivalent to .-)	`m1-k→m3`
Multiply array elements (equivalent to .*)	`m1*m2→m3`
Divide array elements (equivalent to ./)	`m1/m2→m3`
Multiply array elements by an expression (equivalent to .*)	`k*m1→m3`
Divide expression by array elements (equivalent to ./)	`m1/k→m3`
Negate array elements	`-m1→m3`
Raise each array element to a power	`m1^k→m3`
Raise each *m1* element to *m2* power (.^)	`m1^m2 →m3`
Raise an expression to each element *m1* power (.^)	`k^m1→m3`
Take the reciprocal of each element	`1/m1→m3`
Find the factorial of each integer element	`m1!→m3`
Find the sine of each element (also works for cos(), ln(), etc)	`sin(m1)→m3`
Differentiate each array element with respect to *x*	`d(m1,x)→m3`

In general, more complex operations may be handled by nested for-loops that manipulate each array element. Or, in some cases, it is more efficient to process the list elements in sequence. The function below, *transpos(),* shows this approach.

```
transpos(arr,dims,i1,i2)
Func
©Example that transposes a rank-3 array on indices i1 and i2
©Bhuvanesh Bhatt (bbhatt1@towson.edu), Nov2000

Local i,tmp,arr2,dims2

If max(i1,i2)>dim(dims) or min(i1,i2)≤0:Return "Error: invalid indices"

newList(dim(arr))→arr2
```

```
listswap(dims,i1,i2)→dims2

For i,1,dim(arr)
m3aind(i,dims)→tmp
m3sto(m3rcl(tmp,dims,arr),listswap(tmp,i1,i2),dims2,arr2)→arr2
EndFor

arr2

EndFunc
```

Note that *transpos()* calls *m3aind(), m3rcl()* and *m3sto(),* as well as the *listswap()* function described in tip 3.18. These examples show typical results for *transpos().*

```
transpos({a,b,c,d,e,f,g,h},{2,2,2},2,3)      returns      {a,c,b,d,e,g,f,h}
transpos({a,b,c,d,e,f,g,h},{2,2,2},1,2)      returns      {a,b,e,f,c,d,g,h}
```

transpos() is limited to rank-3 arrays, but it can be extended by changing the function references *m3aind()* and so on, as needed. Note that no error checking is done on the *dims* list or on the *i1* and *i2* input arguments, so make sure they are integers. As with a rank-2 array, the transpose function changes the dimensions of the array: if a 1x2x3 array is transposed on the second and third indices, the result is a 1x3x2 array. The *dims2* variable in *transpos()* above gives the dimensions of the resulting array.

If you are using Mathematica, note that regular cubic arrays in Mathematica can be converted to the storage format used in this tip, with Flatten[array].

(Credit to Bhuvanesh Bhatt for the general index and address expressions and programs, the transpos() function, and lots of help with this tip!)