[7.43] Faster function calls with list and matrix arguments

Function call execution time increases substantially with list and matrix arguments. The increase is independent of the code executed by the function, so I call it *overhead*. This tip quantifies the overhead and shows a method to reduce it, at least for global lists and matrices.

I use various timing results in this tip. All execution time data is for a HW2 TI-92 Plus, AMS 2.05. HW1 calculators run about 17% slower. I tested lists and matrices with random floating point integer elements, so execution times will be different if the elements are integers or symbolic elements.

Matrix arguments

I used this test program to measure the overhead for a HW2 TI-92 Plus, AMS 2.05:

```
t()
Prgm
Local k,s,res

Define s(mat)=Func     © Define a function with a single matrix argument
 Return Ø              © Just return a constant
EndFunc

For k,1,5Ø             © Loop fifty times for better timing resolution, ...
 s(m)→res              © ... calling the subroutine with the matrix argument
EndFor

EndPrgm
```

m is a global matrix, initialized to random integer values with *randmat()*. Tests with 34 matrices, with up to 200 rows and 20 columns, resulted in this model equation to estimate the execution time:

$$T = a + b \cdot n_r + c \cdot n_c + d \cdot n_c \cdot n_r \qquad [1]$$

where *T* is the execution time for each call, and

n_r = number of matrix rows
n_c = number of matrix columns

a = 2.5404 E-2 b = 8.2538 E-4 c = 7.615 E-5 d = 8.2447 E-4

This model equation is accurate to about 3%. This table shows some typical overhead times:

Number of rows nr	Number of columns nc	Execution time per call
10	1	42.0 mS
10	4	66.9 mS
100	1	191 mS
100	4	438 mS
200	1	355 mS
200	3	685 mS
3	200	538 mS
5	5	50.5 mS
0	10	117 mS
20	20	373 mS

The overhead is about 0.7 seconds for the largest tested matrix. This may not be critical for single function calls, but it is significant if the function is called repeatedly, as is the case when using the numeric solver, performing numeric integration, or plotting the function. For example, if the 200 x 3 matrix is passed to a function which is plotted at each of the 240 display pixel columns of the TI-92 Plus, the overhead is about 2.7 minutes.

If more than one matrix is used as a function argument, the overhead predictably increases to about the sum of the individual overheads of each matrix. For example, if the arguments are two matrices with dimensions of 50 rows x 2 columns, and 50 rows x 3 columns, the total overhead is about 149 mS + 191 mS = 340 mS.

The overhead may be reduced by passing the matrix by reference, instead of by value. For the timing results and model equation above, the matrix was passed by value, that is, the matrix itself was passed as the function argument. A typical function call is

`f(matrix)` *(call by value)*

To pass the matrix by reference instead of value, the function argument is the matrix name as a string:

`f("mat_name")` *(call by reference)*

With the value method, the element [2,3] is accessed with

`mat[2,3]` *(access by value)*

With the reference method, the same element is accessed using indirection:

`#mat_name[2,3]` *(access by reference)*

where *mat_name* is the name of the matrix. Indirection, a feature built into TI Basic, is accomplished when the number character '#' precedes a variable name, for example, #*var*. This expression does not return the value of *var*, instead, it returns the value of the variable whose name is stored as a string in *var*. For example, if *var* contains the string "var2", and the value of variable *var2* is 7, then #var returns 7.

The reference method works only when the matrix is a global variable. A variable which is local in a program or function cannot be accessed by any other function which is called.

These two test routines demonstrate the overhead reduction when using the reference method.

```
tval()
Prgm
© Pass matrix by value

Local f1,k,r

Define f1(mat)=Func
 Return mat[1,1]
EndFunc

For k,1,5Ø
 f1(m)→r
EndFor

EndPrgm
```

```
tref()
Prgm
© Pass matrix by reference (name)

Local f2,k,r

Define f2(mat)=Func
 Return #mat[1,1]
EndFunc

For k,1,5Ø
 f2("m")→r
EndFor

EndPrgm
```

The *tval()* routine passes the matrix *m* by value, and *tref()* passes the matrix name as a string. *tval()* takes about 199 mS for each function call, while *tref()* takes about 31 mS. This is an improvement of about 84%.

Even though it is faster to call a function with the reference method, it takes longer to access the matrix elements with indirection. For some combination of function calls and matrix element accesses the total execution time is the same for either method. This is called the break-even point. In general, the total execution time is

$$T = N_c T_c + N_a T_a$$

where *Nc* and *Na* are respectively the number of function calls and element accesses, and *Tc* and *Ta* are the execution times for each function call and element access. More specifically, the total execution times for each method are

$$T_v = N_{cv} T_{cv} + N_{av} T_{av} \qquad \textit{Value method}$$
$$T_r = N_{cr} T_{cr} + N_{ar} T_{ar} \qquad \textit{Reference method}$$

The break-even point is at Tv = Tr. We want to compare the two methods with the same conditions, so we equate the number of function calls and element accesses for each method:

$$N_{cv} = N_{cr} = N_c \qquad\qquad N_{av} = N_{ar} = N_a$$

Equate the expressions for Tv and Tr: $N_c T_{cv} + N_a T_{av} = N_c T_{cr} + N_a T_{av}$

and solve for Nc: $N_c = N_a \dfrac{T_{ar} - T_{av}}{T_{cv} - T_{cr}}$ or $\dfrac{N_c}{N_a} = \dfrac{T_{ar} - T_{av}}{T_{cv} - T_{cr}}$ [2]

With equation [2], we can find the break-even point at which the two methods have the same execution time, for some number of function calls and matrix accesses. Tcv is found with equation [1] above, and Tcr is a constant:

$$T_{cr} = 16.88 \text{ mS/access}$$

Timing experiments show that the time required to access a matrix element depends on the number of matrix rows and columns, as well as the element's location in the matrix. The access time also changes slightly if the indices are constants or variables. Timing data shows that Tar - Tav is relatively constant at about 5.7 mS, so equation [2] simplifies to

$$\frac{Nc}{Na} = \frac{5.7 \text{ mS}}{Tcv - 16.88 \text{ mS}} \qquad [3]$$

For example, suppose we have a matrix with 50 rows and 3 columns. We use equation [1] to find Tcv = 191 mS, and Nc/Na = 0.033. If the function accesses only three elements (Na=3), then Nc = 0.1. Since Nc < 1, the reference method is always faster, regardless of the number of function calls. However, if the function accesses every matrix element (Na = 150), then Nc = 4.95. In this case, the reference method is faster only if we call the function more than 5 times.

As another example, consider a matrix with 10 rows and 10 columns. Equation [1] gives Tcv = 117 mS, and Nc/Na = .057. If the function accesses 10 elements, then Nc = .57, so the reference method is always faster. However, if the function accesses all 100 matrix elements, then Nc = 5.7, so the execution time is less for the reference method only if we call the function at least 6 times.

The general result is that the reference method may be faster if the matrix is large and few elements are accessed.

List arguments

When lists are passed as function arguments, the call overhead can also be considerable, as this table shows:

Number of list elements	Overhead / function call
0	22 mS
100	84.6 mS
400	269.1 mS
700	459.8 mS

Timing data for lists of 11 sizes results in this model equation, which is accurate to 1 or 2%:

$$T = 623.2967E\text{-}6\ (N) + 0.021577 \qquad [4]$$

where *T* is the overhead per function call and *N* is the number of list elements. As with the case for matrices above, we can find a function to calculate the break-even point in terms of the number of function calls, and the number of list element accesses made by the function. It is, in fact, the same function with different constants, that is

$$\frac{N_c}{N_a} = \frac{T_{ar} - T_{av}}{T_{cv} - T_{cr}}$$

where

Nc = the number of function calls
Na = the number of list element accesses in the function

Tar = time required to access an element by reference: *expr(list_name&"[]")*
Tav = time required to access an element by value: *list[]*
Tcv = time required to call the function with the list as a value argument; from [4] above
Tcr = time required to call the function with the list argument by name

Tar and Tav are functions of the size of the list, as well as which element of the list is accessed. These functions estimate the mean execution time for a single list element:

$$T_{ar}(N) = 23.0918E - 6 \cdot N + 16.948E - 3$$

$$Tav(N) = 23.1404E - 6 \cdot N + 10.368E - 3$$

where *N* is the number of list elements, so

$$T_{ar}(N) - T_{av}(N) = 48.6E - 9 \cdot N + 6.58E - 3$$

The small *N* coefficient can be ignored. More timing experiments give

Tcr = 16.9 mS

So the equation simplifies to $$\frac{N_c}{N_a} = \frac{6.58\ \text{mS}}{T_{cv} - 16.9\ \text{mS}} \qquad [5]$$

For example, for a list of 100 elements, equation [4] gives Tcv = 83.91 mS. If three elements are accessed each function call (Na = 3), then Nc is 0.29, which means that the reference method is always faster. If all 100 elements are accessed each call (Na = 100), then Nc = 9.8, which means that the reference method will be faster when the function is called at least 10 times.

I mentioned above that the element access time also depends on the size of the list. For a list consisting of 250 elements, it takes about 16 mS to access element [1], 22 mS for element [125], and 28 mS for element [250]. This is substantial variation, which means that the mean timing results above apply only if the probabilities of accessing any particular elements are equal. If you know that your application tends to access elements at a particular list index, you need to account for that in the break-even analysis.

Execution time data for matrix element accesses

This section shows the test data for matrix element access times. All times are for a TI-92 Plus, HW2, AMS 2.05. The test matrices are created with randmat(), so the matrix elements are floating point integers. The mode settings for the tests are RAD and APPROX.

Table 1 shows the time required to access a single matrix element when the matrix indices are constants, for example, *matrix[1,1]*. The test data shows that the overhead to access a matrix element with the reference method (by indirection) is about 5.3 mS. The data also shows the access time variation with respect to the size of the matrix, and the element's position in the matrix.

Table 1
Matrix element access time, constant indices

Matrix dimensions	**Element accessed**	**Access time Tav, value method (mS)**	**Access time Tar, reference method (mS)**	**Reference method overhead Tar - Tav**
10 rows, 10 columns	[1,1]	14.3	19.2	4.9
	[5,5]	16.7	21.8	5.1
	[10,10]	19.3	24.5	5.2
20 rows, 20 columns	[1,1]	14.2	19.4	5.2
	[5,5]	17.7	23.4	5.7
	[10,10]	22.8	28.2	5.4
	[15,15]	27.8	33.3	5.5
	[20,20]	32.7	38.2	5.5

Table 2 shows the time required to access a matrix element when the indices are local variables, for example, *matrix[m1,m2]*. The data shows that it takes slightly longer to access an element with variable indices, but the overhead for indirect access is about the same, at a mean of 5.8 mS.

Table 2
Matrix element access time, variable indices

Matrix dimensions	**Element accessed**	**Access time Tav, value method (mS)**	**Access time Tar, reference method (mS)**	**Reference method overhead Tar - Tav**
10 rows, 10 columns	[1,1]	15.5	21.2	5.7
	[5,5]	17.9	23.7	5.8
	[10,10]	20.6	26.2	5.6
20 rows, 20 columns	[1,1]	15.5	21.3	5.8
	[5,5]	19.5	25.2	5.7
	[10,10]	24.2	30.2	6.0
	[15,15]	29.5	35.3	5.8
	[20,20]	34.2	40.2	6.0

Table 3 shows the mean time required to access each element of matrices of various sizes. Element indices are local variables, and the elements were accessed by value, for example, *matrix[m1,m2]*.

Table 3
Mean matrix element access time, direct access

Matrix rows	**Matrix columns**	**Mean access time, single element (mS)**
5	25	18.9
5	35	19.7
5	45	20.9
5	55	21.4
5	65	22.5
5	75	23.6
5	85	25.0
5	95	25.9
5	105	27.0
5	115	27.9
5	125	28.9
5	135	30.1
5	145	31.3
5	155	32.4
10	10	18.2
10	20	20.3
10	30	22.6
10	40	24.9
10	50	26.8
10	60	30.9
10	70	31.0
10	80	33.3
15	15	20.5
15	25	23.7
15	35	26.9
15	45	30.3
20	10	21.5
20	20	25.5
20	30	29.3
25	5	19.1
30	10	23.8
30	20	30.0
35	5	21.2
40	10	25.2
40	20	34.0
45	5	21.9
50	10	27.7
55	5	23.6
60	10	30.3
65	5	25.0
70	10	32.9
75	5	26.0
80	10	35.4
85	5	27.4
95	5	29.1
105	5	30.5
115	5	31.7
125	5	33.2
135	5	34.9
145	5	36.4
155	5	37.9

The access time in Table 3 can be estimated with this model equation:

$$T_{av} = a + bN_r + cN_c + dN_cN_r$$

where

Nr = number of matrix rows	Nc = number of matrix columns
a = 15.7766 E-3	c = -754.840 E-9
b = 33.9161 E-8	d = 21.2832 E-6

This model equation has a maximum relative error of about 6.7%, but the RMS error is about 1.5%, so it may be useful to estimate access times for matrices not shown in the table. The graph below shows the points used to generate the model equation, which are the points from table 3.

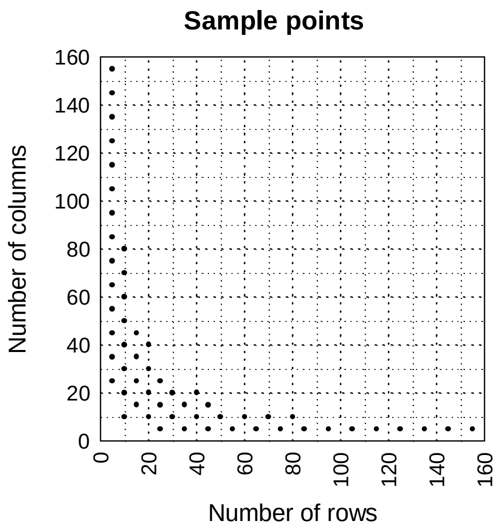

Table 4 shows the mean time to access an element in a few of the matrices from table 3, but using indirection instead of accessing the elements directly. The indices are local variables. The mean overhead, compared to direct access, is about 5.7 mS.

Table 4
Mean matrix element access time, indirect access

Matrix Row	Matrix Columns	Mean acces time, single element (mS)	Indirect access overhead (mS)
5	25	23.8	5.0
5	105	32.6	5.6
5	145	36.9	5.6
10	40	30.4	5.5
10	70	36.4	5.3
15	15	27.2	6.6
25	5	24.9	5.8
25	25	35.2	5.8
40	10	31.3	6.2
70	10	38.7	5.8
105	5	36.3	5.7
145	5	42.1	5.7