Fast manipulation of multi-dimensional arrays in Matlab

Kevin P. Murphy
murphyk@ai.mit.edu

11 September 2002

1 Introduction

Probabilistic inference in graphical models with discrete random variables requires performing various operations on multi-dimensional arrays (discrete potentials). This is true whether we use an exact algorithm like junction tree [CDLS99, HD96] or an approximate algorithm like loopy belief propagation [AM00, KFL01]. These operations consist of element-wise multiplication/division of two arrays of potentially different sizes, and summation (marginalization) over a subset of the dimensions. This report discusses efficient ways to implement these operations in Matlab, with an emphasis on the implementation used in the Bayes Net Toolbox (BNT).

2 Running example

We will introduce an example to illustrate the problems we want to solve. Consider two arrays (tables), Tbig and Tsmall, where Tbig represents a function over the variables X_1, X_2, X_3, X_4 and Tsmall represents a function over X_1, X_3. We will say Tbig's domain is $(1, 2, 3, 4)$, Tsmall's domain is $(1, 3)$, and the difference in their domains is $(2, 4)$. Let the size of variable X_i (i.e., the number of possible values it can have) be denoted by S_i. Here is a straighforward implementation of elementwise multiplication:

```
for i1=1:S1
  for i2=1:S2
    for i3=1:S3
      for i4=1:S4
        Tbig[i1,i2,i3,i4] = Tbig[i1,i2,i3,i4] * Tsmall[i1,i3];
      end
    end
  end
end
```

Similarly, here is a straightforward implementation of marginalizing the big array onto the small domain:

```
for i1=1:S1
  for i3=1:S3
    sum = 0;
    for i2=1:S2
      for i4=1:S4
        sum = sum + Tbig[i1,i2,i3,i4];
      end
    end
    Tsmall[i1,i3] = sum;
  end
end
```

Of course, these are not general solutions, because we have hard-coded the fact that Tbig is 4-dimensional, Tsmall is 2-dimensional, and that we are marginalizing over dimensions 2 and 4. The general solution requires mapping vectors of indices (or subscripts) to 1 dimensional indices (offsets into a 1D array), and vice versa. We discuss these auxiliary functions next, before discussing a variety of different solutions based on these functions.

3 Auxiliary functions

3.1 Converting from multi-dimensional indices to 1D indices

If a d-dimensional array is stored in memory such that the left-most indices toggle fastest (as in Matlab and Fortran — C follows the opposite convention, toggling the right-most indices fastest), then we can compute the 1D index from a vector of indices, subs, as follows:

```
ndx = 1 + (i1-1) + (i2-1)*S1 + (i3-1)*S1*S2 + ... + (id-1)*S1*S2*...*S(d-1)
```

where the vector of indices is $subs = (i_1, \ldots, i_d)$, and the size of the dimensions are $sz = (S_1, \ldots, S_d)$. (We only need to add and subtract 1 if we use 1-based indexing, as in Matlab; in C, we would omit this step.)

This can be written more concisely as

```
ndx = 1 + sum((subs-1) .* [1 cumprod(sz(1:end-1))])
```

If we pass the subscripts separately, we can use the built-in Matlab function

```
ndx = sub2ind(sz, i1, i2, ...)
```

BNT offers a similar function, except the subscripts are passed as a vector:

```
ndx = subv2ind(sz, subs)
```

The BNT function is vectorized, i.e., if each row of subs contains a set of subscripts. then ndx will be a vector of 1D indices. This can be computed by simple matrix multiplication. For example, if

```
subs = [1 1 1;
        2 1 1;
         ...
        2 2 2];
```

and `sz = [2 2 2]`, then `subv2ind(sz, subs)` returns [1 2 ... 8]', which can be computed using

```
cp = [1 cumprod(siz(1:end-1))]';
ndx = (subs-1)*cp + 1;
```

If all dimensions have size 2, this is equivalent to converting a binary number to decimal.

3.2 Converting from 1D indices to multi-dimensional indices

We convert from 1D to multi-D by dividing (which is slower). For instance, if $subs = (1, 3, 2)$ and $sz = (4, 3, 3)$, then ndx will be

$$ndx = 1 + (1-1) * 1 + (3-1) * 4 + (2-1) * 4 * 3 = 21$$

To convert back we get

$$i1 = 1 + \lfloor \frac{(21-1) \bmod 4}{1} \rfloor = 1 + \lfloor \frac{0}{1} \rfloor = 1$$

$$i2 = 1 + \lfloor \frac{(21-1) \bmod 12}{4} \rfloor = 1 + \lfloor \frac{8}{4} \rfloor = 3$$

$$i3 = 1 + \lfloor \frac{(21-1) \bmod 36}{12} \rfloor = 1 + \lfloor \frac{20}{12} \rfloor = 2$$

The matlab and BNT functions sub2ind and subv2ind do this. Note: If all dimensions have size 2, this is equivalent to converting a decimal to a binary number.

3.3 Comparing different domains

In our example, Tbig's domain was (X_1, X_2, X_3, X_4) and Tsmall's domain was (X_1, X_3). The identity of the variables does not matter; what matters is that for each dimension in Tsmall, we can find the corresponding/ equivalent dimension in Tbig. This is called a mapping, and can be computed in BNT using

```
map = find_equiv_posns(small_domain, big_domain)
```

(In R, this function is called match. In Phrog [Phr], this is called an RvMapping.) If $smalldom = (1, 3)$ and $bigdom = (1, 2, 3, 4)$, then $map = (1, 3)$; if $smalldom = (8, 2)$ and $bigdom = (2, 7, 4, 8)$, then $map = (4, 1)$; etc.

4 Naive method

Given the above auxiliary functions, we can implement multiplication as follows:

```
map = find_equiv_posns(Tsmall.domain, Tbig.domain);
for ibig = 1:prod(Tbig.sizes)
  subs_big = ind2subv(Tbig.sizes, ibig);
  subs_small = subs_big(map);
  ismall = subv2ind(Tsmall.sizes, subs_small);
  Tbig.T(ibig) = Tbig.T(ibig) * Tsmall.T(ismall);
end
```

(Note that this involves a single write to Tbig in sequential order, but multiple reads from Tsmall in a random order; this can affect cache performance.)

Marginalization can be implemented as follows (other methods are possible, as we will see below).

```
Tsmall.T = zeros(Tsmall.sizes);
map = find_equiv_posns(Tsmall.domain, Tbig.domain);
for ibig = 1:prod(Tbig.sizes)
  subs_big = ind2subv(Tbig.sizes, ibig);
  subs_small = subs_big(map);
  ismall = subv2ind(Tsmall.sizes, subs_small);
  Tsmall.T(ismall) = Tsmall.T(ismall) + Tbig.T(ibig);
end
```

(Note that this involves multiple writes to Tsmall in a random order, but a single read of each element of Tbig in sequential order; this can affect cache performance.)

The problem is that calling ind2subv and subv2ind inside the inner loop is very slow, so we now seek faster methods.

Row	X_1	X_2	X_3	X_4	ndx
1	1	1	1	1	1
2	2	1	1	1	2
3	1	2	1	1	1
4	2	2	1	1	2
5	1	1	2	1	3
6	2	1	2	1	4
7	1	2	2	1	3
8	2	2	2	1	4
9	1	1	1	2	1
10	2	1	1	2	2
11	1	2	1	2	1
12	2	2	1	2	2
13	1	1	2	2	3
14	2	1	2	2	4
15	1	2	2	2	3
16	2	2	2	2	4

Table 1: An example of computing ndx, where Tbig.domain = [1 2 3 4], Tsmall.domain = [1 3], and Tbig.sizes = [2 2 2 2], ndx is the 1D index corresponding to columns X_1 and X_3.

5 Pre-computing the indices: ndxB

Let `ndx(ibig)` specify the location in Tsmall of the entry that corresponds to entry ibig in Tbig. Then multiplication can be rewritten without a for-loop:

```
Tbig.T(:) = Tbig.T(:) .* Tsmall.T(ndx);
```

Marginalization becomes

```
Tsmall.T = zeros(Tsmall.sizes);
for ibig=1:prod(Tbig.sizes)
  ismall = ndx(ibig);
  Tsmall.T(ismall) = Tsmall.T(ismall) + Tbig.T(ibig);
end
```

which is slow in matlab, but fast in C. ndx can be precomputed as follows:

```
map = find_equiv_posns(Tsmall.domain, Tbig.domain);
subs_big = ind2subv(Tbig.sizes, 1:prod(Tbig.sizes));
ndx = subv2ind(Tsmall.sizes, subs_big(:,map));
```

See Table 1 for an example.

This method of multiplication is about 45 times faster than the previous method on small problems. Unfortunately, storing all the indices for all the pairs of potentials that must be multiplied takes a lot of space (equal to the size of the big array, i.e., B=prod(Tbig.sizes)). We discuss two solutions to this below.

6 Re-using the indices: the index cache

Notice that we can re-use indices for multiplying any pair of tables, or for marginalizing any big table onto a small domain, for which map and Tbig.sizes are the same. (For example, multiplying a domain (B, D) by (B, A, C, D, E) is the same as multiplying (A, B) by (A, E, F, G, B, H), provided the sizes of the big arrays are the same, since in both cases we are multiplying onto dimensions 1 and 4.) Hence we can store the indices in a global cache, indexed by map and Tbig.sizes, and re-use them for many different problems. To

look up the indices, we need to convert map and big-sizes, both of which are lists of positive (small) integers, into integers. This could be done with a hash function. This makes it possible to hide the presence of the cache inside an object (call it a TableEngine) which can multiply/ marginalize (pairs of) tables, e.g.,

```
function Tbig = multiply(TableEngine, Tsmall, Tbig)
  map = find_equiv_posns(Tsmall.domain, Tbig.domain);
  cache_entry_num  = hash_fn(map, Tbig.sizes);
  ndx = TableEngine.ndx_cache{cache_entry_num};
  Tbig.T(:) = Tbig.T(:) .* Tsmall.T(ndx);
```

One big problem with Matlab is that Tbig will be passed by value, since it is modified within the function, and then copied back to the callee. This could be avoided if functions could be inlined, or if pass by reference were supported, but this is not possible with the current version of Matlab.

Another problem is that computing the hash function is slow in Matlab, so what I currently do in BNT is explicitely store the cache-entry-num for every pair of potentials that will be multiplied (i.e., for every pair of neighboring cliques in the jtree). Also, for every potential, I store the cache-entry-num for every domain onto which the potential may be marginalized (this corresponds to all families and singletons in the bnet). This allows me to quickly retrieve the relevant cache entry, but unfortunately requires the inference engine to be aware of the presence of the cache. That is, the current code (inside jtree-ndx-inf-engine/collect-evidence) looks more like the following:

```
ndx = B_NDX_CACHE{engine.mult_cl_by_sep_ndx_id(p,n)};
ndx = double(ndx) + 1;
Tsmall = Tsmall(ndx);
Tbig(:) = Tbig(:) .* Tsmall(:);
```

where mult-cl-by-sep-ndx-id(p,n) is the cache entry number for multiplying clique p by separator n. By implementing a TableEngine with a hash function, we could hide the presence of the cache, and simplify the code. Furthermore, instead of having jtree-inf and jtree-ndx-inf, we would have a single class, jtree-inf, which could use different implementations of the TableEngine object, e.g., with or without cacheing. Indeed, any code that manipulates tables (e.g., loopy) would be able to use different implementations of TableEngine. We will now discuss other possible implementations of the TableEngine, which use different kinds of indices, or even no indices at all.

7 Reducing the size of the indices: ndxSD

We can reduce the space requirements for storing the indices from B to S+D, where S=prod(sz(Tsmall.domain)) and D=prod(sz(diff-domain)). To explain how this works, let us first rewrite the marginalization code, so that we write to each element of Tsmall once in sequential order, but do multiple random reads from Tbig (the opposite of before).

```
small_map = find_equiv_posns(Tsmall.domain, Tbig.domain);
diff = setdiff(Tbig.domain, Tsmall.domain);
diff_map = find_equiv_posns(diff, Tbig.domain);
diff_sz = Tbig.sizes(diff_map);
for ismall = 1:prod(Tsmall.sizes)
  subs_small = ind2subv(Tsmall.sizes, ismall);
  subs_big = zeros(1, length(Tbig.domain));
  sum = 0;
  for jdiff = 1:prod(diff_sz)
    subs_diff = ind2subv(diff_sz, idff);
    subs_big(small_map) = subs_small;
    subs_big(subs_diff) = subs_diff;
    ibig = subv2ind(Tbig.sizes, subs_big);
```

```
    sum = sum + Tbig.T(ibig);
  end
  Tsmall.T(ismall) = sum;
end
```

Now suppose we have a function that computes ibig given ismall and jdiff, call it index(ismall, jdiff). Then we can rewrite the above as follows:

```
for ismall = 1:prod(Tsmall.sizes)
  sum = 0;
  for jdiff = 1:prod(diff_sz)
    sum = sum + Tbig.T(index(ismall, jdiff));
  end
  Tsmall.T(ismall) = sum;
end
```

Similarly, we can implement multiplication by doing a single write to Tbig in a random order, and multiple sequential reads from Tsmall (the opposite of before).

```
for ismall = 1:prod(Tsmall.sizes)
  for jdiff = 1:prod(diff_sz)
    ibig = index(ismall, jdiff);
    Tbig.T(ibig) = Tbig.T(ibig) * Tsmall.T(ismall);
  end
end
```

7.1 Computing the indices

We now explain how to compute the index (what [Phr] calls a FactorMapping) using our running example.[1] Referring to Table 1, we see that entries 1,3,9,11 of Tbig map to Tsmall(1), entries 2,4,10,12 map to 2, entries 5,7,13,15 map to 3, and entries 6,8,14,16 map to 4. Instead of keeping the whole ndx, it is sufficient to keep two tables: one that keeps the first value of each group, call it start (in this case [1,2,5,6]) and one that keeps the offset from the start within each group (in this case [0, 2, 8, 10]). (In BNT, start is called small-ndx and offset is called diff-ndx.) Then we have

```
index(ismall, jdiff) = start(ismall) + offset(jdiff)
```

We can compute the start positions by noticing that, in this example, we just increment X_1 and X_3, keeping the remaining dimensions (the diff domain) constantly clamped to 1; this can be implemented by setting the effective size of the difference dimensions to 1 (so they only have 1 possible value).

```
diff_domain = mysetdiff(Tbig.domain,Tsmall.domain);
diff_sizes = Tbig.sizes(map);
map = find_equiv_posns(diff_domain, Tbig.domain);
sz = Tbig.sizes;
sz(map) = 1; % sz(diff) is 1, sz(small) is normal
subs = ind2subv(sz, 1:prod(Tsmall.sizes));
start = subv2ind(Tbig.sizes, subs);
```

Similarly, we can compute the offsets by incrementing X_2 and X_4, keeping X_1 and X_3 fixed.

```
map = find_equiv_posns(Tsmall.domain, Tbig.domain);
sz = Tbig.sizes;
sz(map) = 1; % sz(small) is 1, sz(diff) is normal
subs = ind2subv(sz, 1:prod(diff_sz));
offset = subv2ind(Tbig.sizes, subs) - 1;
```

[1] The web page at `http://robotics.stanford.edu/~frog/mappings.html` has a similar example, but uses C-based indexing, i.e., starts counting from 0 and toggles the right-most digits fastest.

7.2 Avoiding for-loops

Given start and offset, we can implement multiplication and marginalization using two for-loops, as shown above. This is fast in C, but slow in Matlab. For small problems, it is possible to vectorize both operations, as follows.

First we create a matrix of indices, ndx2, from start and offset, as follows:

```
ndx2 = repmat(start(:), 1, prod(diff_sizes)) + repmat(offset(:)', prod(Tsmall.sizes), 1);
```

In our example, this produces

$$\begin{pmatrix} 1 & 1 & 1 & 1 \\ 2 & 2 & 2 & 2 \\ 5 & 5 & 5 & 5 \\ 6 & 6 & 6 & 6 \end{pmatrix} + \begin{pmatrix} 0 & 2 & 8 & 10 \\ 0 & 2 & 8 & 10 \\ 0 & 2 & 8 & 10 \\ 0 & 2 & 8 & 10 \end{pmatrix} = \begin{pmatrix} 1 & 3 & 9 & 11 \\ 2 & 4 & 10 & 12 \\ 5 & 7 & 13 & 15 \\ 6 & 8 & 14 & 16 \end{pmatrix}$$

Row 1 contains the locations in Tbig which should be summed together and stored in Tsmall(1), etc. Hence we can write

```
Tsmall.T = sum(Tbig.T(ndx2), 2);
```

For multiplication, each element of Tsmall gets mapped to many elements of Tbig, hence

```
Tbig.T(ndx2(:)) = Tbig.T(ndx2(:)) .* repmat(Tsmall.T(:), prod(diff_sz), 1);
```

8 Eliminating for-loops and indices

We can eliminate the need for for-loops and indices as follows. We make Tsmall the same size as Tbig by replicating entries where necessary, and then just doing an element-wise multiplication:

```
temp = extend_domain(Tsmall.T, Tsmall.domain, Tsmall.sizes, Tbig.domain, Tbig.sizes);
Tbig.T = Tbig.T .* temp;
```

Let us explain `extend-domain` using our standard example. First we reshape Tsmall to make it have size [2 1 2 1], so it has the same number of dimensions as Tbig. (This involves no real work.)

```
map = find_equiv_posns(smalldom, bigdom);
sz = ones(1, length(bigdom));
sz(map) = small_sizes;
Tsmall = reshape(Tsmall, sz);
```

Now we replicate Tsmall along dimensions 2 and 4, to make it the same size as big-sizes (we assume the variables have a standardized order, so there is no need to permute dimensions).

```
sz = big_sizes;
sz(map) = 1;
Tsmall = repmat(Tsmall, sz(:)');
```

Now we are ready to multiply: `Tbig .* Tsmall`.

For small problems, this is faster, at least in Matlab, but in C, it would be faster to avoid copying memory. Doug Schwarz wrote a genops class using C that can avoid the repmat above. See `http://myweb.servtech.com/~schwarz/genops.html` (unfortunately no longer available online).

For marginalization, we can simply call Matlab's built-in sum command on each dimension, and then squeeze the result, to get rid of dimensions that have been reduced to size 1 (we must remember to put back any dimensions that were originally size 1, by reshaping).

Method	Section	ndx	Mult			Marg		
			Tsmall	Tbig	vec	Tsmall	Tbig	vec
ndxB	5	B	multi rnd rd	sgl seq wr	yes	multi rnd wr	sgl seq rd	no
ndxSD	7	S+D	multi seq rd	sgl rnd wr	no	sgl seq wr	sgl rnd rd	no
ndx2	7.2	S+D (B)	repmat	sgl rnd wr	yes	sgl seq wr	sgl rnd rd	yes
repmat	8	none	repmat	sgl seq wr	yes	multi seq rd/wr	copy	partly

Table 2: Summary of the methods

```
diff_dom = mysetdiff(Tbig.domain, Tsmall.domain);
map = find_equiv_posns(diff_dom, Tbig.domain);
Tsmall.T = Tbig.T;
for d=1:length(map)
  Tsmall.T = sum(Tsmall.T, map(i));
end
Tsmall.T = squeeze(Tsmall.T);
```

9 Summary

We have now seen several different ways to marginalize/multiply multi-dimensional arrays, which we summarized in Table 2. The columns have the following meaning. ndx is the size of the indices which are used: B = prod(big-sizes), S=prod(small-sizes), D = prod(diff-sizes). (The ndx2 methods takes as input start, of size S, and offset, of size D, but creates a temporary variable, ndx2, of size B.) Mult refers to multiplication, and Marg to marginalization; for each, we specify what kind of operations we perform on Tbig and TSmall (single or multiple reads or writes, random or sequential), and also whether the operation can be vectorized in Matlab. The repmat operation entails multiple sequential reads; copying entails a single sequential read; being indexed by a loop-variable entails a sequential read/write, whereas being indexed indirectly entails a random read/write. (For the last line, marginalization needs to loop over each dimension of the diff-domain, but not over values; hence it is a short loop, which is what we mean by partly vectorized.)

In Matlab, vectorized code will be faster unless it uses too much memory. In C, vectorization is irrelevant, and what matters is cache performance, which depends on the way elements are accessed, and potentially on the size of any indices, which also may be cached. (Note that the indices should be declared as integers, to avoid converting them from reals at every iteration; this is tricky in Matlab.)

To compare the different methods, we can create different implementations (sub-classes) of an abstract MultiDimArrayEngine class, which implements the following methods: engine-¿multiply(smallTable, bigTable) and engine-¿marginalize(bigTable, small-domain). We can imagine at least 4 different classes, corresponding to the implementations in Table 2. (The non-vectorized ones should be implemented in C, of course.) It is an open question which method is best; it will depend on the size of the tables, but possibly other factors.

The really important question is: is there some clever way of mapping these operations onto some hardware primitives, or some fast algorithms like FFT?

References

[AM00] S. M. Aji and R. J. McEliece. The generalized distributive law. *IEEE Trans. Info. Theory*, 46(2):325–343, March 2000.

[CDLS99] R. G. Cowell, A. P. Dawid, S. L. Lauritzen, and D. J. Spiegelhalter. *Probabilistic Networks and Expert Systems*. Springer, 1999.

[HD96] C. Huang and A. Darwiche. Inference in belief networks: A procedural guide. *Intl. J. Approx. Reasoning*, 15(3):225–263, 1996.

[KFL01] F. Kschischang, B. Frey, and H-A. Loeliger. Factor graphs and the sum-product algorithm. *IEEE Trans Info. Theory*, February 2001.

[Phr] Phrog: Stanford's C++ Bayes Net package. http://robotics.stanford.edu/ frog/mappings.html.