# In-Place Transposition of Rectangular Matrices

Fred G. Gustavson[1] and Tadeusz Swirszcz[2]

[1] T. J. Watson Research Center, Yorktown Heights, NY 10598, USA,
fg2@us.ibm.com
[2] Faculty of Mathematics and Information Science, Warsaw University of Technology, Warsaw, Poland, Europe,
swirszcz@mini.pw.edu.pl

**Abstract.** We present a new Algorithm for In-Place Rectangular Transposition of an $m$ by $n$ matrix $A$ that is efficient. In worst case it is $O(N \log N)$ where $N = mn$. It uses a bit-vector of size `IWORK` words to further increase its efficiency. When `IWORK=0` no extra storage is used. We also review some of the other existing algorithms for this problem. These contributions were made by Gower, Windley, Knuth, Macleod, Laffin and Brebner (ACM Alg. 380), Brenner (ACM Alg. 467), and Cate and Twigg (ACM Alg. 513). Performance results are given and they are compared to an Out-of-Place Transposition algorithm as well as ACM Algorithm 467.

## 1 Introduction

We present a new algorithm that requires little or no extra storage to transpose a $m$ by $n$ rectangular (non-square) matrix $A$ in-place. We assume that $A$ is stored in the standard storage format of the Fortran and C programming languages. We remark that many other programming languages use this same standard format for laying out matrices. One can prove that it requires $O(N \log N)$ operations in worst case where $N = mn$. It uses a bit-vector of size `IWORK` words to further increase its efficiency. When `IWORK=0` no extra storage is used. When `IWORK = m*n/ws` where `ws` is the word size the algorithm has $O(N)$ complexity.

Matrix $A^T$ is an $n$ by $m$ matrix. Now both $A$ and $A^T$ are simultaneously represented by either $A$ or $A^T$. Also, in Fortran, $A$ and $A^T$ are stored stride one by column. An application determines which format is best and frequently, for performance reasons, both formats are used. Currently, Dense Linear Algebra libraries do *not* contain in-place transpose algorithms when $m \neq n$.

Our algorithm is based on following the cycles of a permutation $P$ of length $q = mn - 1$. This permutation $P$ is defined by the mapping of $A$ onto $A^T$ that is induced by the standard storage layouts of Fortran and C. Thus, if one follows a cycle of $P$ then one must eventually return to the beginning point of this cycle of $P$. By using a bit vector one can tag which cycles of $P$ have been visited and then a starting point for each new cycle is easily determined. The cost of this algorithm is easily seen to be $O(q)$ which is minimal. Now, we go further and remove the bit vector. Thus, we need a method to distinguish

between a new cycle and a previous cycle (the original reason for the bit vector). A key observation is that every new cycle has a minimum starting value. If we traverse a proposed new cycle and we find an iterate whose value is less than the current starting value we know that the cycle we are generating has already been generated. We can therefore abort and go on to the next starting value. On the other hand, if we return to the original starting value, thereby completing a cycle, where every iterate is larger than this starting value we are assured that a new cycle has been found and we can therefore record it. Our algorithm is based on this simple idea which is originally due to Gower [2, 7]. In [7] Knuth does an in depth complexity analysis of Gower's Algorithm as it applied to in-situ permutation in general. Knuth showed that in-situ permutation had an average case complexity of $O(n \log n)$ but worst case of $O(n^2)$. However, in the special case when both $P$ and $P^{-1}$ are known (this includes matrix transposition), the worst case is reduced to $O(n \log n)$. Now, using $P$ and $P^{-1}$ is equivalent to searching a cycle in both directions which we did by the BABE (Burn At Both Ends) programming technique.

References [6, 4, 8–10, 13] all emphasize the importance of the fundamental mapping $P(k) = mk \bmod q$. Here $0 < k < q$, $i = \lfloor k/n \rfloor$, $j = k - ni$ and $P(k) = i + mj$ is the location of $A_{ij}$ assuming Fortran storage order and 0-origin subscripting of $A_{ij}$. $A_{ij}$ moves to $(A^T)_{ij}$ under this mapping.

Our algorithm Modified In-Place Transpose (`MIPT`) is closely related to ACM Alg. 467 [8]. However, algorithm `MIPT` has the following four new features. First, we use the BABE approach which makes our code provably $O(N \log N)$ in worst case. Second, `MIPT` removes a bug in ACM Alg. 467. Third, `MIPT` stores a bit vector in an integer array as opposed to using an integer array to just store 0 or 1. Fourth, the BABE inner loop of `MIPT` has been made more efficient than the inner loop of ACM Alg. 467.

To remove the bug, we did not use the mapping $P(k)$. We used instead the Fortran statement `KBAR=M*K-Q*(K/N)`. The map $P(k)$ can cause destructive integer overflow whereas the Fortran statement does *not*.

Our algorithm `MIPT` stores a bit vector in integer array `MOVE` of size `IWORK` instead of having each element of `MOVE` hold 0 or 1. Thus, this gives a factor of 32 storage gain over ACM Algs. 380, 467, and 513. Our experiments indicate that a fixed size for `IWORK`, say 100 words, is always a good idea. Finally, we have made several other changes to ACM Alg. 467 which we describe in Section 3.

In Section 2, we describe our basic key idea first discovered by Gower and later used by most of our references. In Section 3, we fully describe Alg. `MIPT`. It uses our discovery of a duality result which is a key feature of Alg. `MIPT`. We call attention to Theorem 7 of [9] which proves that if a self dual cycle exists then the dual cycle mechanism used in our Alg. `MIPT` (also in ACM Algs. 467 and 513) meets "in the middle" and so the recording of dual and self cycles can be merged into single piece of code. In Section 4, we prove that $\bar{k} =$ `KBAR`. Section 5 gives performance studies. Section 6 traces the history of research on the subject of in-place transpose.

## 2 The Basic In-place Transposition Algorithm IPT

```
ALGORITHM IPT (m,n,A)
DO cnt = 1, mn-2
  k = P(cnt)
  DO WHILE (k > cnt)
    k = P(k)
  END DO
  IF (k = cnt) then
    Transpose that part
    of A which is in the
    new cycle just found
  ENDIF
END DO
```

We have just described Gower's algorithm; see pages 1 and 2 of [2] and page 2 of [7]. The algorithm `IPT` does extra computation in its inner while loop. Here is an observation. Sometimes the algorithm `IPT` completes while `cnt` is still on the first column of $A$; i.e. before `cnt` reaches $m$. However, when $P$ has small cycles this causes `cnt` to become much larger before `IPT` completes. Also, one can compute the number of one-cycles in $P$ for any matrix $A$. This is a greatest common divisor (gcd) computation and there are $1 + gcd(m-1, n-1)$ one cycles. Since non-trivial one cycles always occur in the interior of $A$; ie, for large values of `cnt`, knowing their number can sometimes drastically reduce the cost of running `IPT`. To see this, note that the outer loop of `IPT` runs from 1 to $mn-2$. If one records the total cycle count so far `tcc` then one can leave the outer loop when `tcc` reaches $mn$. We now rename the modification of `IPT` that uses the gcd logic and the `tcc` count our basic algorithm `IPT`.

## 3 Descriptive Aspects of Algorithm MIPT

A programming technique called BABE can be used to speed up the inner while loop of `IPT`. BABE traverses the while loop from both ends and thus requires use of both $P$ and $P^{-1}$. This additionally guarantees that `MIPT` will have a worst case complexity of $O(N \log N)$. Use of BABE allows one to use functional parallelism; [11]. More importantly, matrix elements are *not* accessed during the inner while loop and so no cache misses occur. In fact the inner while loop consists entirely of register based fixed point instructions and hence the inner loop will perform at the peak rate of many processors. The actual transposition part of `IPT` runs very very slowly in comparison to the inner loop processing: Any cycle of $P$ usually accesses the elements of $A$ in a completely random fashion. Hence, a cache miss almost always occurs for each element of the cycle; thus, the whole line of the element is brought into cache and the remaining elements of the line usually never get used. On the other hand, an out-of-place transpose algorithm allows one to bring the elements of $A$ into and out of cache in a fairly

structured way. These facts illustrate the principle of "trading storage to increase performance".

Let $\bar{k} = P(k)$ and $l = q - k$. One can show that $P(l) = q - \bar{k}$. Thus, let `cnt` generate a cycle and suppose that iterate $l = q -$ `cnt` does not belong to this cycle. Then $q -$ `cnt` also generates a cycle. This result shows that a duality principle exists for $P$. The criterion for `cnt` is $j \geq$ `cnt` for every $j$ in the cycle. For $q -$ `cnt` the criterion is $j \leq q -$ `cnt` for every $j$ in the companion cycle.

The value $q$ is the key to understanding $P$ and hence our algorithm. Let $k$ be the storage location of $A_{ji}, 0 < k < q$. One can show $P(k) = mk \bmod q$. $P(k)$ is the storage location of $A_{ij}$. If $d$ is a divisor of $q$, then every iterate of $d$ also divides $q$. Hence, when `cnt` starts at $d$ one can alternatively look at $\bar{k} = mod(nk, q/d)$ where `cnt` begins at 1. So, we can partition $0 < k < q$ into a disjoint union over the divisors of $d$ of $q$. For each $d$, we can apply a suitable variant of algorithm `IPT`, called algorithm `MIPT`, as a routine that is called for each $d$ of $q$. This master algorithm drastically reduces the operation count of the inner while loop of algorithm `IPT`. These remarks also describe several of the features of ACM Alg. 467. Not mentioned is our use of integer array `MOVE` to hold a bit vector (Alg. 467 uses the elements of its integer `MOVE` array to hold either a 0 or 1). We mention our clever use of combining the search for a possible new cycle with its dual cycle when such a dual cycle exists. When a dual cycle does not exist, ACM Algs. 467 and 513 use self duality and meet in the middle. However, when a dual cycle exists, the two new cycles are found using just $P$ ; see also Theorem 7 on page 106 of [9]. Thus, to use our BABE approach we needed a fast way to determine, apriori, when a cycle had a dual cycle.

### 3.1 Apriori Determination of Dual / Self Dual Cycles

The results of Knuth [7] and Fich et. al. [12] allowed us to prove that our BABE approach was sufficient to guarantee an $O(N \log N)$ running time of Algorithm `MIPT`. However, we need a way to modify the current Algorithm `MIPT` to do this: Every divisor $d$ of $q$ is a cycle minima. Hence, at no additional cost, one finds `CL` the cycle length of minima $d$. Knowing `CL` one can use Theorem 7 of [9] which states that a cycle is self dual if and only if $n^{\texttt{CL}/2} = q - 1$. This computation is very cheap if one uses powers-of-two doubling.

### 3.2 Some Details on Algorithm `MIPT`

We now describe the overall features of Algorithm `MIPT` by further contrasting it to the three earlier algorithms [6, 8, 9].

Both Brenner [8] and later Cate and Twigg [9] improved Laflin and Brebner's algorithm [6] by combining the dual and self dual cases into a single processing case. Brenner went further when he applied some elementary Abelian group theory: the natural numbers $0 < i < mn$ partition into $n_d$ Abelian groups where $n_d$ is the number of divisors $d$ of $\phi(q)$; $\phi$ is Euler's phi function. We have $q = \sum_{d|q} ord(G_d)$ and $ord(G_d) = \phi(q/d)$. We and he both recognized that this partition can sometimes greatly reduce the time spent in the search for cycle

minima. However, the inner loop now becomes a double loop where its inner loop must run over *only* the numbers relatively prime to $\phi(q/d)$. This complication forces the overhead of the inner inner loop to increase. However, sometimes the cycle minima search is drastically reduced; in those cases the overall processing time of the new double inner loop is greatly reduced. All three algorithms [6, 8, 9] combine a variant of our bit vector algorithm with Gower's algorithm. By variant we mean they all use an integer sized array of size `IWORK` to just hold a bit. They all say that setting the length of `IWORK` equal to $(m+n)/2$ is a good choice. Now by using a bit vector we gain a factor of 32 over their use of integer array `MOVE`. However, locating and accessing / testing a bit in a 32 bit word has a higher processing cost than a simple integer compare against 0 / 1. Quite simply, the bit vector approach is more costly per unit invocation. However, since one gains a factor of 32 in the size of `IWORK` at no additional storage cost the number of times the costly inner loop will be entered will be lessened and overall, in some cases, there will be a big gain in the processing time of Algorithm `MIPT`.

## 4 Integer Overflow and the Fortran mod Function

Previously, we have seen that $P(k) = mk \bmod q$ where $0 < k < q$, $i = \lfloor k/n \rfloor$, $j = k - ni$ and $P(k) = i + mj$ is the location of $A_{ij}$ assuming Fortran storage order and 0-origin subscripting of $A_{ij}$. $A_{ij}$ moves to $A_{ij}^T$ under this mapping. Now, one can safely use this $ij$ direct definition of $P(k)$ in a Fortran program as integer overflow cannot occur. But, by using the Fortran mod function to compute $P(k)$ instead, one can increase the speed of computing $P(k)$ and hence increase the speed of the inner loop processing of Algorithm `MIPT`. But integer overflow can occur when using the Fortran mod formula for $P(k)$. And this Fortran mod formula then computes an *unwanted* result. More bluntly, the use of the Fortran mod formula produces a bug in Brenner's ACM Alg. 467! In ACM Algs. 380 and 513 the authors compute `m*k-q*(k/n)` as the value for $P(k)$. Now, integer overflow also does occur. However, despite the overflow occuring, the Fortran formula `m*k-q*(k/n)` computes $P(k)$ correctly. Both [6, 9] fail to note this possibility of overflow although [9] states that the Fortran formula `m*k-q*(k/n)`computed faster than the CDC 6600 system modulo function. Further thought would have shown why this is so as these two formulas must compute different results in the case of integer overflow. In any case, we now prove in the Lemma below that `m*k-q*(k/n)` computes $P(k)$ correctly.

Lemma: Given are positive integers $m$ and $n$. Let $q = mn - 1$ such that $q \leq 2^{31} - 1$. Let $0 < k < q$ with $k$ relatively prime to $q$. Set $\bar{k} = mk \bmod q$ and $i = \lfloor k/n \rfloor$. Then $\bar{k} = mk - iq$. Let `KBAR=M*K-(K/N)*Q` be a Fortran statement where `I, K, M, N, Q` and `KBAR` are `INTEGER*4` Fortran variables. We shall prove `KBAR` $= \bar{k}$.

Proof: Put $j = k - ni$. Then $\bar{k} = i + jm = mq - iq$ as simple calculations show. Let $mk = q_1 \times 2^{32} + r_1$ and $iq = q_2 \times 2^{32} + r_2$ with $0 \leq r_l < 2^{31}$, $l = 1, 2$. Let $r_l = b_l \times 2^{31} + s_l$, $l = 1, 2$. Here $0 \leq b_l \leq 1$ and $0 \leq s_l \leq 2^{31}$. Again simple calulations show that $\bar{k} = s_1 - s_2$ when $b_1 = b_2$ and $\bar{k} = 2^{31} + s_1 - s_2$ when

$b_1 \neq b_2$. Also, $0 < \bar{k} < q$ is representable by `INTEGER*4` Fortran variable `KBAR`. Suppose $b_1 = b_2 = 0$. Then `M*K` equals $s_1$, `-I*Q` equals $-s_2$ and the Fortran sum `KBAR` equals $s_1 - s_2 = \bar{k}$. Also, if $b_1 = b_2 = 1$, then `M*K` equals $-(2^{31} - s_1)$ and `-I*Q` equals $2^{31} - s_2$. Their algebraic sum equals $s_1 - s_2 = \bar{k}$ which also equals the Fortran sum `KBAR`. Note that when $b_1 = b_2$, `KBAR` is the sum of terms of mixed sign and overflow will not occur. Now suppose $b_1 = 0$ and $b_2 = 1$. Then `M*K` equals $s_1$ and `-I*Q` equals $2^{31} - s_2$. Both operands are positive and their Fortran sum `KBAR` = their actual sum $\bar{k}$. Since $\bar{k}$ is a 32 bit positive integer no overflow will occur. Finally, let $b_1 = 1$ and $b_2 = 0$. Then `M*K` equals $-(2^{31} - s_1)$ and `-I*Q` equals $-s_2$. After setting $-2^{31} = -2^{32} + 2^{31}$ we find the actual sum equals $-2^{32} + \bar{k}$. However, in this case, for 32 bit arithmetic, overflow always occurs and thus `KBAR` = $\bar{k}$. This completes the proof.

## 5 Performance Studies of `MIPT`

In the first experiment, run only on a Power 5, ACM Alg. 467 was compared to `MIPT`. 190 matrices of row and column sizes from 50 to 1000 in steps of 50 were generated and one experiment was conducted. We made two runs where `IWORK` = 0 and 100. When `IWORK` = 0 both codes do *not* use their arrays `MOVE` as their sizes are zero. When `IWORK` = 100, `MIPT` uses 3200 entries of $k$ whereas ACM Alg. 467 uses 100 entries of $k$ to reduce their respective inner loop searches. We ruled out the cases when $m = n$ as both codes are then the same. There are 19*20/2 = 190 cases in all. We label these cases from 1 to 190 starting at `m=1000, n=950`. Thus `m=1000, n=950:50:-50` are cases 1 to 19, `m= 950, n=900:50:-50` are cases 20 to 37 and so on down to `m=100, n=50` which is case 190. Our performance results as functions of `m` and `n` are essentially random. Results in both Tables are separated into good and bad cases. The good cases mean that `MIPT` is faster than ACM Alg. 467 and the bad cases mean the opposite. The good and bad cases are ordered into buckets of percent. A good case in bucket 10 means that `MIPT` is between 10 to 11 % faster than Alg. 467. In general a good case in bucket `j` means that `MIPT` is between `j % to (j+1) %` faster than Alg. 467. A result in bucket `j` (bad or good) translates to a factor of `1 + j/100` to `1 + (j+1)/100` times faster. If `j >= 100`, then one of the algorithms is more than twice as fast as the other.

```
--------------------TABLE 1 ( IWORK = 0 )------------------------------
   5 BAD CASES  as  2 Triples of %j :  # of Problems :  Problem Numbers
 0:1: 142
 1:4: 18 81 139 147
-----------------------------------------------------------------------
  185 GOOD CASES as 5 Triples of %j :  # of Problems :  Problem Numbers
 0:12: 38 51 60 86 87 107 113 125 127 128 142 155
 1:36: 8 16 21 23 24 27 30 36 55 62 65 71 74 78 93 95 101 102 104 115 129
 130 135 137 146 156 160 167 170 171 172 173 177 180 183 184
 2:62: 5 9 11 13 14 15 20 26 28 29 31 32 33 34 41 45 46 49 50 54 58 61 63
 66 72 75 77 79 80 85 88 89 92 94 103 105 108 109 110 117 118 119 120 122
```

```
 126 131 132 134 136 138 143 144 148 152 162 163 168 169 176 178 185 188
 3:72: 1 2 3 4 6 7 10 12 17 19 22 25 37 39 40 43 44 47 48 52 53 56 57 59
 64 67 68 69 70 73 76 82 83 84 90 91 96 97 98 99 100 106 111 112 114 116
 123 124 133 140 141 145 149 150 151 153 154 157 158 159 161 164 165 174
 175 179 181 182 186 187 189 190
 4:3: 35 121 166
-------------------------------------------------------------------------
```

The results of Table 1 demonstrate that Algorithm `MIPT` is always marginally, about 3 %, faster than ACM Alg. 467.

```
--------------------TABLE 2 ( IWORK = 100 )----------------------------
   76 BAD CASES as 12 Triples  of %j :  # of Problems :  Problem Numbers
 0:20:21 29 32 33 38 42 46 49 60 74 75 76 80 87 113 125 128 139 147 164
 1:20:9 16 24 27 31 45 61 65 78 81 88 105 118 121 136 137 143 146 150 156
 2:9:62 79 84 93 95 107 117 127 166 | 3 4:102 115 119 129 | 4 2:34 160
 5:5:108 134 148 167 177 | 6:2:69 171 | 7:2:98 152 | 8:4:70 111 172 176
 11:2:112 174 | 15:2:123 135 | 16:4:37 54 161 178
   13 more BAD CASES as 13 Pairs of     ( %j  Problem Number )
 (10 173), (12 185), (13 145), (14 179), (19 162), (21 182), (25 99)
 (29 180), (30 188), (38 184), (40 187), (54 189), (90 190)
-------------------------------------------------------------------------
  60 GOOD CASES as  9 Triples of %j :  # of Problems :  Problem Numbers
 0:27:5 7 8 28 30 41 47 51 55 58 64 71 77 83 86 94 96 100 101 104 120
        130 131 132 142 149 155
 1:14:4 17 23 26 36 44 57 67 110 114 122 140 141 170
 2:4:20 25 56 90 |  3:4:52 72 157 163 |  4:3:63 68 109 | 22:2:22 175
 93:2:1 169 |  102:2:2 6 |  114:2:15 138
   41 more GOOD CASES as 41 Pairs of    ( %j  Problem Number )
 ( 5 18), ( 7 116), (11 35), (18 91), (27 126), (34 10), (39 11), (40 12),
 (46 92), (47 133), (48 59), (53 159), (56 124), (58 50), (62 14),
 (66 66), (70 48), (71 158), (72 73), (76 43), (77 151), (82 183),
 (84 186), (87 82), (89 144), (92 39), (94 89), (95 19), (97 85), (104 13),
 (108 181), (110 53), (111 153), (112 3), (113 154), (116 168), (132 40),
 (139 97), (144 103), (156 165), (186 106)
-------------------------------------------------------------------------
```

The results of Table 2 demonstrate that Algorithm `MIPT` benefits from using a bit-vector approach as opposed to using an integer array whose elements represent just single bits. The unit overhead cost of bit processing is greater than a straight integer-compare of zero or one. The 89 bad-case results are due to the effect of the higher unit cost being greater than the savings afforded by inner loop testing. For the 101 good cases, there are more savings afforded by inner loop testing in `MIPT` and these savings more than offset the losses of higher unit cost bit processing. Note that there are 55 bad cases and 56 good cases whose percentage is less than 5. So, in these 101 cases performance is about equal. In the remaining 89 cases of which 45 are good and 34 are bad there are 16 good cases and no bad cases that are more than twice as fast.

In the second experiment, done on an IBM Power 5 and a PowerPC 604, Algorithm MIPT was compared to ESSL subroutine DGETMO which is an out-of-place transpose routine. The major cost of Alg. MIPT over DGETMO has to do with accessing a whole line of $A$ with only one element being used. Now the PowerPC 604 has line size 4 whereas the Power 5 has line size 16. And qualitatively, the two runs show this as the ratios are better on the Power 5.

```
----------TABLE 3 on the Power 5 ( IWORK = 100 )---------------------
  190 CASES   13 Triples  of 100j% :  # of Problems :  Problem Numbers
4:22: 35 36 67 68 69 95 96 97 120 121 122 139 140 141 142 143 155 156
      157 158 170 171
5:28: 20 30 32 33 34 51 52 64 65 66 91 93 94 98 109 110 116 117 118
      119 136 137 138 149 150 159 160 172
6:45: 1 17 21 22 23 24 25 26 27 28 29 31 50 55 56 57 58 59 60 61 62 63
      83 86 87 88 89 90 92 107 108 111 113 114 115 123 132 146 147 148
      151 152 163 176 177
7:31: 16 38 39 41 42 43 44 47 48 49 53 70 81 82 84 100 101 102 104 105
      106 129 130 131 134 153 164 166 167 173 181
8:19: 2 3 4 8 19 37 40 45 46 54 80 103 127 128 133 161 165 174 178
9:26: 5 6 7 9 10 11 12 13 14 71 72 73 74 75 76 77 78 79 112 125 126 135
      145 162 179 185
10:3: 99 144 182 | 11:5: 85 168 180 184 188 | 12:1: 124
13:4: 15 154 186 189 | 14:1: 175 | 15:1: 169 |16:3: 183 187 190 | 46:1: 18
-------------------------------------------------------------------------
----------TABLE 4 on the PowerPC 604 ( IWORK = 100 )--------------------
  190 CASES   4 Triples  of 100j% :  # of Problems :  Problem Numbers
0:39: 19 37 53 54 68 70 84 98 112 113 123 134 135 142 143 145 152 153
      161 162 165 166 167 168 171 172 173 174 176 177 178 179 181 182
      185 186 188 189 190
1:128: 8 10 11 12 13 14 15 16 17 27 28 29 30 31 32 33 34 35 36 41 43 44
       45 46 47 48 49 50 51 52 55 57 58 59 60 61 62 63 64 65 66 67 69 71
       72 74 75 76 77 78 79 80 81 82 83 85 86 87 88 89 90 91 92 93 94 95
       96 97 99 100 101 102 103 104 105 106 107 108 109 110 111 114 115
       116 117 118 119 120 121 122 124 125 126 127 128 129 130 131 132
       133 136 137 138 139 140 141 144 146 147 148 149 150 151 154 155
       156 157 158 159 160 163 164 169 170 175 183 184 187
2:22: 1 2 3 4 5 6 7 9 18 20 21 22 23 24 25 26 38 39 40 42 56 73 | 4:1: 180
 ------------------------------------------------------------------------
```

In Tables 3 and 4 we have made each bucket 100 times larger. So, a result in bucket `j` translates to a factor of `1 + j` to `1 + j+1` times faster. If `j >= 1`, then DGETMO is more than twice as fast as Alg. MIPT. One can see, on the Power 5 in Table 3, that DGETMO is more than five times faster than MIPT for all 190 cases. It is more than ten times faster for 45 cases and for one case it is 48 times faster. On the PowerPC604 DGETMO is again always faster. This time there are 39 cases where it is not twice as fast, 128 cases where it is between 2 and 3 times faster, 22 cases where it is between 3 to 4 times faster and one case where it is 5.9 times faster. So, Tables 3 and 4 corroborate the remarks made in paragraph one of Section 3.

In Section 1 we described our bit vector algorithm that was based on our Algorithm `IPT` (see Section 2). On the Power 5 we compared its performance on our set of 190 matrices against our Algorithm `MIPT` with `IWORK=100`. Surprisingly, `MIPT` was faster in 189 cases and in 15 cases it was more than twice as fast.

## 6 Discussion of Prior In-Place Transpostion Algorithms

We thought our algorithms were new; the second printing of Knuth, Vol. 1, Fundamental Algorithms gave us this impression, see [4]. In [4], we became aware of problem 12 of section 1.3.3 which posed the subject of our paper as an exercise. The solution of problem 12 gave outlines of bit-vector algorithms but gave no specific details. Additionally, Berman's algorithm [1], the algorithm of Pall and Seiden and ACM Algorithm 302 by Boothroyd [3] were cited. However, one of our key discoveries was *not* mentioned: the use of the bit vector could be removed; however, at the cost of many additional computer operations. Later, in [13], Knuth additionally cites Brenner and ACM Algorithm 467, [8], Windley, [2], himself, [7], Cate and Twigg, [9] and Fich, Munro, Poblete, [12].

Windley's [2] paper gave three solutions to the in-place transposition problem which was an exercise to students taking the Cambridge University Diploma in Numerical Analysis and Automatic Computing. Berman's algorithm was noted but not considered. The first solution by M. Fieldhouse was an $O(m^2n^2)$ algorithm. J. C. Gower produced our basic algorithm IPT by *first* discovering one of our key discoveries. Gower also used the count of the elements transposed to speed up his algorithm. The third algorithm, due to Windley, was a variant of Gower's Algorithm which for each $k$, $0 \leq k < mn$ placed one element of the transpose in its correct position. Macleod, in [5], discusses in-situ permutation that governed the matrix transposition. He presents a modification of Gower's Algorithm which he believed to be "the most efficient yet devised". He notes that its performance varied from reasonable to poor depending on the permutation governing matrix transposition.

ACM Alg. 380, in Laflin and Brebner [6], also uses Gower's and our later key discovery. It also uses another discovery of ours: a duality principal. In [6], duality was called symmetric. Laflin and Brebner [6] were first to describe the duality result. Finally, ACM Alg. 380 uses an integer array `MOVE` of length `IWORK`. The purpose of array `MOVE` was to reduce the cost of the additional computer operations imposed by using the key discovery. ACM Alg. 380 gives empirical evidence that `IWORK` should be set to $(m + n)/2$. We note the use of `IWORK` produces a hybrid algorithm that combines the bit vector approach with the use of the key idea. Brenner's ACM Alg. 467 is an improvement of ACM Alg. 380. Cate and Twigg [9] discuss some theoretical results related to in-situ transposition and use these to accelerate ACM Algs. 302 and 380. ACM Alg. 302, see [3] was an improvement of Windley's algorithm. Finally, in [10] Leathers presents experimental evidence that ACM Alg. 513 is inferior to ACM Alg. 467 and laments the publication of ACM Alg. 513. In [12], Fich, Munro and Poblete give new computational complexity results on permuting in-place. When both

$P$ and $P^{-1}$ are known they give in their Figure 5 on page 269 a modified version of Gower's Algorithm. We now briefly indicate why ACM Alg. 467 is faster than ACM Algs. 380 and 513. All three algorithms use array `MOVE` to reduce the use of their costly inner loops. If `IWORK` $= mn$ then one is using our bit vector algorithm and one completely avoids using this inner loop. Unfortunately, these three algorithms are then no longer in-situ. ACM Algs. 467 and 513 use the duality principle in an optimal way whereas ACM Alg. 380 does not. Finally, ACM Alg. 467 additionally recognizes when $q$ has divisors $d$. When $q$ has several divisors $d$ the costly inner loop search can be greatly reduced. We also discovered these facts. In [9], Cate and Twigg note that the Fortran statement for `KBAR` is faster than using the system function for $P(k)$. For this reason, ACM Alg. 380 and 513 avoided a latent bug whereas ACM Alg. 467 did not. We find it a curious fact, which we proved, that the formula for `KBAR` works in the case of integer overflow whereas the Fortran definition of mod forces extra operations that, when integer overflow occurs, compute an unwanted result.

## References


1. M. F. Berman. A Method for Transposing a Matrix. *J. Assoc. Comp. Mach.* Vol. 5, 1958, pp. 383-384.
2. P. F. Windley. Tranpsoing matrices in a digital computer. *Comput. J.*, Vol. 2, April 1959, pp. 47-48.
3. J. Boothroyd. Alg. 302: Transpose vector stored array. *Comm. ACM* Vol. 10, No. 5, 1967, pp. 292-293.
4. D. Knuth. Problem 12, page 180 and Solution to Problem 12. page 517. *The Art of Computer Programming, Vol. 1, 1st edition, 2nd printing* book, Addison-Wesley, Reading Mass., 1969
5. I. D. G. Macleod. An Algorithm For In-Situ Permutation. *The Austrialian Computer Journal* Vol. 2, No. 1 Feb. 1970, pp. 16-19.
6. S. Laflin and M. A. Brebner. Alg. 380: In-situ transposition of a rectangular matrix. *Comm. ACM* Vol. 13, May 1970, pp. 324-326.
7. D. Knuth. Matematical Analysis of Algorithms *Information Processing 71, Invited Papers-Foundations* North-Holland Publishing Company 1972
8. N. Brenner. Matrix Transposition in Place. *Comm. ACM* Vol. 16, No. 11, Nov. 1973, pp. 692-694.
9. E. G. Cate and D. W. Twigg. Algorithm 513: Analysis of In-Situ Transposition. *ACM TOMS* Vol. 3, No. 1, March 1977, pp. 104-110.
10. B. L. Leathers. Remark on Algorithm 513: Analysis of In-Situ Transposition. *ACM TOMS* Vol. 5, No. 4, Dec. 1979, pp. 520.
11. R. C. Agarwal, F. G. Gustavson, M. Zubair. Exploiting functional parallelism of POWER2 to design high-performance numerical algorithms. *IBM Journal of Research and Development*, Vol. 38, No. 5, Sep. 1994, pp. 563,576.
12. F. E. Fich, J. I. Munro, P. V. Poblete. Permuting In Place. *SIAM Journal of Computing*, Vol. 24, No. 2, Apr. 1995, pp. 266,278.
13. D. Knuth. Problem 12, page 182 and Solution to Problem 12. page 523. *The Art of Computer Programming, Vol. 1, 3rd edition, 4th printing* book, Addison-Wesley, Reading Mass., 1997