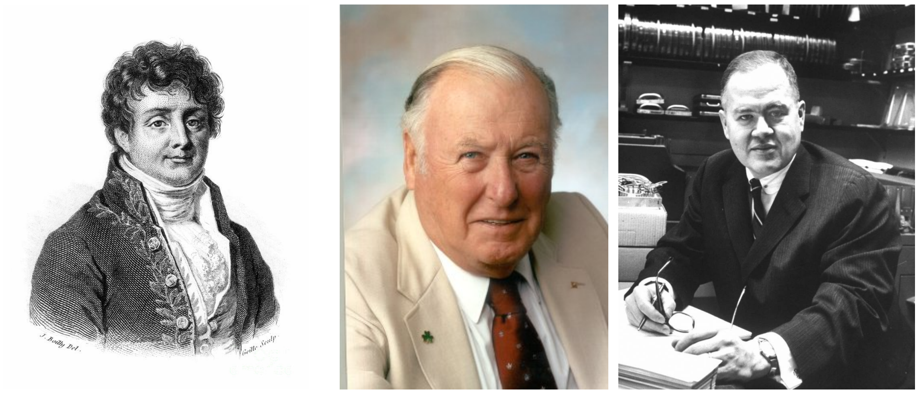
## The Fast Fourier Transform

The Fourier transform was originally developed by Joseph Fourier [3] for the study of heat transfer and vibrations. Fourier transforms are currently used in the study of differential equations, approximation theory, quantum mechanics, time-series analysis, implementation of high precision arithmetic, digital signal processing, GPS, sound and video compression, digital telephony and encryption. The fast Fourier transform is a divide and conquer algorithm developed by Cooley and Tukey [1] to efficiently compute a discrete Fourier transform on a digital computer. In 2000 Dongarra and Sullivan listed the fast Fourier transform among the top 10 algorithms of the 20th century [2].



Joseph Fourier of École Polytechnique, James Cooley of IBM Watson Laboratories and John Tukey of Princeton University and Bell Laboratories.

### The Discrete Fourier Transform

The discrete Fourier transform is given by the matrix-vector multiplication $Ax$ where $A$ is an $N \times N$ matrix with general term given by $a_{kl} = e^{-i2\pi kl/N}$ with $k = 0, 1, \ldots, N-1$ and $l = 0, 1, \ldots, N-1$. While standard mathematical notation for matrices and vectors use index variables which range from 1 to $N$, we have shifted the indices by one so that the first column and first row of $A$ are given by $k = 0$ and $l = 0$. Shifting indices in this way is both the natural for the C programming language and the mathematics. This shifted notation for indices will be used throughout our computational study of linear algebra.

Define $\overline{A}$ to be the matrix whose entries are exactly the complex conjugates of the entries of $A$. Our first result is

**The Fourier Inversion Theorem.** *Let $A$ be the $N \times N$ Fourier transform matrix defined above. Then*

$$A^{-1} = \frac{1}{N}\overline{A}.$$

To see why this formula is true we first prove

**The Orthogonality Lemma.** *Suppose $l, p \in \{0, 1, \ldots, N-1\}$, then*

$$\sum_{q=0}^{N-1} e^{i2\pi(l-p)q/N} = \begin{cases} N & \textit{for } l = p \\ 0 & \textit{otherwise.} \end{cases}$$

**Proof of The Orthogonality Lemma.** Since

$$0 \le l \le N-1 \qquad \text{and} \qquad -(N-1) \le -p \le 0,$$

then $-(N-1) \le l-p \le N-1$ and consequently

$$-2\pi\Big(1-\frac{1}{N}\Big) \le 2\pi(l-p)/N \le 2\pi\Big(1-\frac{1}{N}\Big).$$

Define $\omega = e^{i2\pi(l-p)/N}$. Since the only time $e^{i\theta} = 1$ is when $\theta$ is a multiple of $2\pi$, we conclude that

$$\omega = 1 \qquad \text{if and only if} \qquad l = p.$$

Clearly, if $l = p$ then

$$\sum_{q=0}^{N-1} e^{i2\pi(l-p)q/N} = \sum_{q=0}^{N-1} w^q = \sum_{q=0}^{N-1} 1 = N.$$

On the other hand, if $l \ne p$ then $\omega \ne 1$. In this case,

$$\omega^N = e^{i2\pi(l-p)} = 1$$

and the geometric sum formula yields that

$$\sum_{q=0}^{N-1} e^{i2\pi(l-p)q/N} = \sum_{q=0}^{N-1} \omega^q = \frac{1-\omega^N}{1-\omega} = \frac{1-1}{1-\omega} = 0.$$

This finishes the proof of the lemma. ////

We are now ready to explain the Fourier inversion theorem.

**Proof of The Fourier Inversion Theorem.** Let $b = Ax$ and $c = \frac{1}{N}\overline{A}b$. Claim that $c = x$. By definition

$$b_k = \sum_{l=0}^{N-1} e^{-i2\pi kl/N} x_l \qquad \text{and} \qquad c_p = \frac{1}{N}\sum_{q=0}^{N-1} e^{i2\pi pq/N} b_q.$$

Substituting yields

$$\begin{aligned} c_p &= \sum_{q=0}^{N-1} e^{-i2\pi pq/N}\Big(\frac{1}{N}\sum_{l=0}^{N-1} e^{i2\pi ql/N} x_l\Big) = \frac{1}{N}\sum_{l=0}^{N-1}\Big\{\sum_{q=0}^{N-1} e^{i2\pi(l-p)q/N}\Big\} x_l \\ &= \frac{1}{N}\sum_{l=0}^{N-1}\left\{\begin{array}{ll} N & \text{for } l = p \\ 0 & \text{otherwise} \end{array}\right\} x_l = \frac{N}{N} x_p = x_p. \end{aligned}$$

This finishes the proof of the theorem. ////

Let's pause for a moment to implement a computer program that computes the Fourier transform and the inverse Fourier transform directly from the definitions using matrix-vector multiplication. The resulting FORTRAN code looks like

```
program main
    implicit none
    integer,parameter :: FTSIZE=8
    real*8,parameter :: M_PI=3.14159265358979323846D0
    integer :: i
    complex*16,dimension(0:FTSIZE-1):: X,B,C
    do i=0,FTSIZE-1
        X(i)=cmplx(1D0/(i+1),1D0/(FTSIZE-i),kind(X(i)));
    end do
    print '("N=",I0)',FTSIZE
    print '("X=")'
    call cvecprint(X);
    call dft(X,B)
    print '("B=")'
    call cvecprint(B)
    call invdft(B,C)
    print '("C=")'
    call cvecprint(C)
    return

contains

subroutine dft(x,b)
    complex*16,dimension(0:) :: x,b
    integer :: N,l,k
    N=size(x)
    do l=0,N-1
        b(l)=0
    end do
    do k=0,N-1
        do l=0,N-1
            b(k)=b(k)+exp(cmplx(0D0,-2*M_PI*k*l/N,kind(b(k))))*x(l)
        end do
    end do
end subroutine

subroutine invdft(x,b)
    complex*16,dimension(0:) :: x,b
    integer :: N,l,k
```

```
    N=size(x)
    do l=0,N-1
        b(l)=0
    end do
    do k=0,N-1
        do l=0,N-1
            b(k)=b(k)+exp(cmplx(0,2*M_PI*k*l/N,kind(b(k))))*x(l)/N;
        end do
    end do
end subroutine

subroutine cvecprint(x)
    complex*16,dimension(0:) :: x
    integer :: N,i
    N=size(x)
    do i=0,N-1
        print '("(",G0," ",G0,")")',x(i)
    end do
end subroutine

end program
```

and produces the output

```
N=8
X=
(1.0000000000000000 0.12500000000000000)
(0.50000000000000000 0.14285714285714285)
(0.33333333333333331 0.16666666666666666)
(0.25000000000000000 0.20000000000000001)
(0.20000000000000001 0.25000000000000000)
(0.16666666666666666 0.33333333333333331)
(0.14285714285714285 0.50000000000000000)
(0.12500000000000000 1.0000000000000000)
B=
(2.7178571428571425 2.7178571428571425)
(-0.86391851475668857E-001 -0.20856837951108187)
(0.22204460492503131E-015 -0.58333333333333370)
(0.28564698969660285 -0.68961283657658679)
(0.63452380952381005 -0.63452380952380949)
(1.0197251848090030 -0.42238400144129962)
(1.4476190476190478 -0.12767564783189300E-014)
(1.9810196769700616 0.82056521752897127)
C=
(1.0000000000000000 0.12500000000000014)
```

```
(0.49999999999999978 0.14285714285714307)
(0.33333333333333370 0.16666666666666718)
(0.25000000000000050 0.19999999999999968)
(0.20000000000000015 0.24999999999999944)
(0.16666666666666596 0.33333333333333359)
(0.14285714285714185 0.49999999999999989)
(0.12500000000000078 0.99999999999999989)
```

Note that the value for $c$ is the same as $x$. This is consistent with the Fourier Inversion Theorem and leads us to believe that the above code is producing correct results. Making sure the code is producing the correct answer is an important first step before any sort of optimization is attempted.

We now analyze the performance of the above simple Fourier transform code. Observe that the `dft` and `invdft` routines each consist of two loops of length $N$. As the loops are nested, the resulting number of operations is $N^2$. We will obtain a significant performance increase by changing the code to use the fast Fourier transform algorithm which only takes about $N \log_2 N$ number of operations. Before doing this, we instrument the above slow algorithm with timing routines and also create a parallel version for an example of parallel programming. The modified code looks like

```
program main
    implicit none
    integer,parameter :: FTSIZE=8192
    integer :: tic_start,tic_finish,tic_rate
    real*8 :: t
    real*8,parameter :: M_PI=3.14159265358979323846D0
    integer :: i
    complex*16,dimension(0:FTSIZE-1):: X,B,C
    do i=0,FTSIZE-1
        X(i)=cmplx(1D0/(i+1),1D0/(FTSIZE-i),kind(X(i)));
    end do
    print '("N=",I0)',FTSIZE
    call tic
    call dft(X,B)
    t=toc()
    print '("B(0)=(",G0," ",G0,")")',B(0)
    print '("dft took ",G0," seconds.")',t
!$omp parallel
!$omp single
    call tic
    call pdft(X,B)
    t=toc()
!$omp end single
!$omp end parallel
    print '("B(0)=(",G0," ",G0,")")',B(0)
```

```
    print '("parallel dft took ",G0," seconds.")',t
    return

contains

subroutine tic
    call system_clock(count_rate=tic_rate)
    call system_clock(tic_start)
end subroutine

real*8 function toc()
    call system_clock(tic_finish)
    toc=float(tic_finish-tic_start)/tic_rate
end function

subroutine dft(x,b)
    complex*16,dimension(0:) :: x,b
    integer :: N,l,k
    N=size(x)
    do l=0,N-1
        b(l)=0
    end do
    do k=0,N-1
        do l=0,N-1
            b(k)=b(k)+exp(cmplx(0D0,-2*M_PI*k*l/N,kind(b(k))))*x(l)
        end do
    end do
end subroutine

subroutine pdft(x,b)
    complex*16,dimension(0:) :: x,b
    integer :: N,l,k
    N=size(x)
    do l=0,N-1
        b(l)=0
    end do
!$omp taskloop default(shared)
    do k=0,N-1
        do l=0,N-1
            b(k)=b(k)+exp(cmplx(0D0,-2*M_PI*k*l/N,kind(b(k))))*x(l)
        end do
    end do
!$omp end taskloop
end subroutine
```

```
70
71 end program
```

To use multiple processor cores the only change needed is to add a parallel loop for the matrix multiplication denoted by `omp taskloop` on line 62. On a 3.0Ghz dual-core AMD A6-9225 based notebook computer the above code produces the output

```
N=8192
B(0)=(9.5881900460952654 9.5881900460953098)
dft took 4.6409997940063477 seconds.
B(0)=(9.5881900460952654 9.5881900460953098)
parallel dft took 2.4769999980926514 seconds.
```

For this system, a factor of 1.87 performance increase is obtained when switching from one to two CPUs. The same program when run on a 2.4Ghz twelve-core Intel Xeon E5-2620 based system with hyperthreading enabled produces the output

```
N=8192
B(0)=(9.5881900460952654 9.5881900460953098)
dft took 5.9670000076293945 seconds.
B(0)=(9.5881900460952654 9.5881900460953098)
parallel dft took 0.49399998784065247 seconds.
```

In this case the performance increase was about 12 times faster which is essentially linear with the number of cores. Sometimes performance increases sub-linearly as we will shall see later.

### The Fast Fourier Transform

While a factor 18 speedup was easy to obtain by parallelizing the slow algorithm, in the case of the Fourier transform much more significant gains can be achieved by using a conquer and divide approach. This is possible because the matrix $A$ corresponding to the Fourier transform has a significant number of symmetries in it based on the factors of the length $N$ of the transform. For simplicity we will assume that $N = 2^n$ for some positive integer $n$. Thus, $N$ is divisible by 2 and we can write $2K = N$. It follows that

$$\begin{aligned}\sum_{l=0}^{N-1} e^{-i2\pi kl/N} x_l &= \sum_{l \text{ even}} e^{-i2\pi kl/N} x_l + \sum_{l \text{ odd}} e^{-i2\pi kl/N} x_l \\ &= \sum_{p=0}^{K-1} e^{-i2\pi kp/K} x_{2p} + e^{-i2\pi k/N} \sum_{p=0}^{K-1} e^{-i2\pi kp/K} x_{2p+1}\end{aligned} \tag{1}$$

Note that the original Fourier transform of size $N$ has been rewritten as two smaller Fourier transforms of size $K$ which then need to be combined. The combining is done by multiplying the second transform by the factor $e^{-i2\pi k/N}$ for $k = 0, 1, \ldots, N-1$ which results in $N$ additional multiplications. Therefore, the total number of operations has been reduced to

$$K^2 + N + K^2 = 2\Big(\frac{N}{2}\Big)^2 + N = \frac{1}{2}N^2 + N$$

which is a reduction of almost half the original $N^2$.

We are now ready to prove

**The Fast Fourier Transform Theorem.** *Suppose $N = 2^n$, then the Fourier transform can be computed in $N \log_2 N$ number of operations.*

**Proof of The Fast Fourier Transform Transform Theorem.** Consider the minimal number of operations $T_n$ needed to perform a discrete Fourier transform of size $2^n$. By the conquer and divide step described above, we know that

$$T_n \le 2T_{n-1} + 2^n \qquad \text{and similarly} \qquad T_{n-1} \le 2T_{n-2} + 2^{n-1}.$$

Substituting the latter in to the former yields $T_n \le 2^2 T_{n-2} + 2(2^n)$ and by induction it follows that

$$T_n \le 2^n T_0 + n2^n.$$

Since the transform of length one is the identity then $T_0 = 0$. Consequently, $T_n \le N \log_2 N$. This shows the Fourier transform can be computed in $N \log_2 N$ operations. ////

We remark that $N \log_2 N$ number of operations can be much smaller than $N^2$ when $N$ is large. When $N = 8192$, as used for our previous numerical test, it follows that

$$N \log_2 N = 106496 \qquad \text{and} \qquad N^2 = 67108864.$$

Since $67108864/106496 \approx 630$, using the fast Fourier transform has the performance advantage of about 630 additional processor cores when $N = 8192$. For larger values of $N$ the advantages are even more pronounced. When $N = 65536$ the slow algorithm takes an impractically long time; for values of $N$ corresponding to vectors that are sized to the limits of available memory, the fast algorithm is the only way to complete the computation.

We finish by presenting a recursive routine written in FORTRAN to compute the fast Fourier transform. The code

```
program main
    implicit none
    integer,parameter :: FTSIZE=8
    real*8,parameter :: M_PI=3.14159265358979323846OD0
    integer :: i
    complex*16,dimension(0:FTSIZE-1):: X,B,C
    do i=0,FTSIZE-1
        X(i)=cmplx(1D0/(i+1),1D0/(FTSIZE-i),kind(X(i)));
    end do
    print '("N=",I0)',FTSIZE
    call fft(X,B)
    print '("fft_B=")'
    call cvecprint(B)
    return

```

```
contains

recursive subroutine fftwork(N,s,o,x,p,b)
    integer :: k,N2,N,s,o,p
    complex*16,dimension(0:) :: x,b
    complex*16 :: even,odd,w
    if(N.eq.1) then
        b(p)=x(o)
        return
    end if
    if(mod(N,2).ne.0) then
        print '("N not divisible by 2!")'
        stop
    end if
    N2=N/2
    call fftwork(N2,2*s,o,x,p,b)
    call fftwork(N2,2*s,o+s,x,p+N2,b)
    do k=0,N2-1
        even=b(p+k)
        odd=b(p+k+N2)
        w=exp(cmplx(0.0D0,-2*M_PI*k/N,kind(w)))
        b(p+k)=even+w*odd
        b(p+k+N2)=even-w*odd
    end do
end subroutine

subroutine fft(x,b)
    complex*16,dimension(0:) :: x,b
    integer :: N
    N=size(x)
    call fftwork(N,1,0,x,0,b)
end subroutine

subroutine cvecprint(x)
    complex*16,dimension(0:) :: x
    integer :: N,i
    N=size(x)
    do i=0,N-1
        print '("(",G0," ",G0,")")',x(i)
    end do
end subroutine

end program
```

produces the output

```
N=8
fft_B=
(2.7178571428571425 2.7178571428571425)
(-0.86391851475668746E-001 -0.20856837951108154)
(0.0000000000000000 -0.58333333333333326)
(0.28564698969660318 -0.68961283657658712)
(0.63452380952380949 -0.63452380952380949)
(1.0197251848090021 -0.42238400144129939)
(1.4476190476190476 0.55511151231257827E-016)
(1.9810196769700634 0.82056521752896805)
```

Compare the output for the slow routine to the fast routine. When optimizing an computer program it is important to compared results produced by the optimized code to known correct results. The fact that the output is the same in this case, suggests that the optimized code performs the same calculation as the original program. Although one test case—or even a hundred—would not be sufficient guarantee two different algorithms always produce the same results, such testing is useful and can catch many errors.

For now, we assume the code is correct and proceed to check performance by instrumenting the code with timing routines and also creating a parallel version as was done for the slow Fourier transform. The modified code looks like

```
program main
    implicit none
    integer,parameter :: FTSIZE=1048576
    integer :: tic_start,tic_finish,tic_rate
    real*8 :: t
    real*8,parameter :: M_PI=3.14159265358979323846D0
    integer :: i
    complex*16,dimension(:),allocatable:: X,B
    allocate(X(0:FTSIZE-1))
    allocate(B(0:FTSIZE-1))
    do i=0,FTSIZE-1
        X(i)=cmplx(1D0/(i+1),1D0/(FTSIZE-i),kind(X(i)));
    end do
    print '("N=",I0)',FTSIZE
    call tic
    call fft(X,B)
    t=toc()
    print '("B(0)=(",G0," ",G0,")")',B(0)
    print '("fft took ",G0," seconds.")',t
!$omp parallel
!$omp single
    call tic
    call pfft(X,B)
    t=toc()
```

```
!$omp end single
!$omp end parallel
    print '("B(0)=(",G0," ",G0,")")',B(0)
    print '("parallel fft took ",G0," seconds.")',t
    return

contains

subroutine tic
    call system_clock(count_rate=tic_rate)
    call system_clock(tic_start)
end subroutine

real*8 function toc()
    call system_clock(tic_finish)
    toc=float(tic_finish-tic_start)/tic_rate
end function

recursive subroutine fftwork(N,s,o,x,p,b)
    integer :: k,N2,N,s,o,p
    complex*16,dimension(0:) :: x,b
    complex*16 :: even,odd,w
    if(N.eq.1) then
        b(p)=x(o)
        return
    end if
    if(mod(N,2).ne.0) then
        print '("N not divisible by 2!")'
        stop
    end if
    N2=N/2
    call fftwork(N2,2*s,o,x,p,b)
    call fftwork(N2,2*s,o+s,x,p+N2,b)
    do k=0,N2-1
        even=b(p+k)
        odd=b(p+k+N2)
        w=exp(cmplx(0D0,-2*M_PI*k/N,kind(w)))
        b(p+k)=even+w*odd
        b(p+k+N2)=even-w*odd
    end do
end subroutine

subroutine fft(x,b)
    complex*16,dimension(0:) :: x,b
```

```
    integer :: N
    N=size(x)
    call fftwork(N,1,0,x,0,b)
end subroutine

recursive subroutine pfftwork(N,s,o,x,p,b)
    integer :: k,N2,N,s,o,p
    complex*16,dimension(0:) :: x,b
    complex*16 :: even,odd,w
    if(N.eq.1) then
        b(p)=x(o)
        return
    end if
    if(mod(N,2).ne.0) then
        print '("N not divisible by 2!")'
        stop
    end if
    N2=N/2
!$omp task default(shared) final(s<=32)
    call pfftwork(N2,2*s,o,x,p,b)
!$omp end task
    call pfftwork(N2,2*s,o+s,x,p+N2,b)
!$omp taskwait
    do k=0,N2-1
        even=b(p+k)
        odd=b(p+k+N2)
        w=exp(cmplx(0D0,-2*M_PI*k/N,kind(w)))
        b(p+k)=even+w*odd
        b(p+k+N2)=even-w*odd
    end do
end subroutine

subroutine pfft(x,b)
    complex*16,dimension(0:) :: x,b
    integer :: N
    N=size(x)
    call pfftwork(N,1,0,x,0,b)
end subroutine

subroutine cvecprint(x)
    complex*16,dimension(0:) :: x
    integer :: N,i
    N=size(x)
    do i=0,N-1
```

```
113         print '("(",G0," ",G0,")")',x(i)
114     end do
115 end subroutine
116
117 end program
```

For the parallel code `omp task` directive in line 87 schedules one of the recursive calls to compute a smaller Fourier transform in a separate worker thread while the current thread recursively computes the other Fourier transform. The `omp taskwait` on line 91 makes sure both of the recursive calls have completed before the results are combined with a parallel loop in line 46. After `pfft` recurses 5 times, the stride given by `s` is equal 32 and 32 parallel tasks have been created to perform the computation. At this point the `final` clause causes the rest of the recursion to be made in serial. On a 3.0Ghz dual-core AMD A6-9225 based notebook computer the above code produces the output

```
N=1048576
B(0)=(14.440159752937522 14.440159752937522)
fft took 0.72000002861022949 seconds.
B(0)=(14.440159752937522 14.440159752937522)
parallel fft took 0.48100000619888306 seconds.
```

We note that the fast Fourier transform performed a transform of size $N = 1048576$ faster than the slow algorithm could handle a transform of size $N = 8192$. This time only a factor of 1.5 parallel speedup occurs when working computing two cores. The same program when run on a 2.4Ghz twelve-core Intel Xeon E5-2620 based system produces the output

```
N=1048576
B(0)=(14.440159752937522 14.440159752937522)
fft took 0.82099997997283936 seconds.
B(0)=(14.440159752937522 14.440159752937522)
parallel fft took 0.20800000429153442 seconds.
```

In this case the performance increase was about 4 times faster, which is only 33 percent of the optimal 12 times speedup. Parallel efficiency can be limited by the read-write bandwidth of main memory as well as overhead related to scheduling the parallel loop on the multiple cores. We observe that the memory access patterns of the fast Fourier transform involve recursively skipping by odd and even indexes. In particular the fast Fourier transform does not access memory sequentially. It is likely that this creates additional pressure on the memory subsystem compared to the simple discrete transform discussed earlier.

**References**


1. James Cooley and John Tukey, An algorithm for the machine calculation of complex Fourier series, *Math. Comput.*, Vol. 19, 1965.
2. Jack Dongarra and Francis Sullivan, Top Ten Algorithms of the Century, *Computing in Science and Engineering*, 2000.
3. Joseph Fourier, Théorie analytique de la chaleur, *Firmin Didot Père at Fils*, 1822.

**Homework Problems**

1. The conquer and divide step described in equation (1) splits the terms of the sum for the discrete Fourier transform into odd and even terms. Construct a similar equation for use when $N = 3^n$ that divides the sum into three parts such that $l$ divided by 3 has remainder 0, 1 or 2.

2. Let $z = a + bi$ and $w = u + iv$ be complex numbers. It takes four real-valued multiplications when using the foil method to find the product $zw$. Look up fast complex multiplication, describe it and explain how many real-valued multiplications the fast algorithm uses to find $zw$.

3. Compute the number of real-valued double-precision floating point multiplications and additions per second achieved for test runs of the the fast Fourier transform detailed above. Explain your reasoning and how you counted the total number of operations. How many evaluations of the exponential function are performed?

4. Download the code `fasttime.f90` for determining the speed of the fast Fourier transform from our website. Compile and run it on your computer. Compare the speed of this code to the one developed in class.

5. [Extra Credit] Constuct an improved Fourier transform code that runs faster than either the ones presented in this handout or the one developed in class. Consider evaluating the exponential function ahead of time. Another optimization would be to avoid letting the routine recurse all the way down to the identity transform of length one and instead end with an optimized transform of length 4 or 8.