Une bibliothèque de programmes optimisés pour le calcul de la Transformée de Fourier Rapide de séquences de taille N=2n (algorithme de "Split Radix")

Gérard Yvon, Pierre Duhamel

CENTRE NATIONAL D'ETUDES DES TELECOMMUNICATIONS

CENTRE NATIONAL DE LA RECHERCHE SCIENTIFIQUE

CENTRE DE RECHERCHES EN PHYSIQUE DE L'ENVIRONNEMENT TERRESTRE ET PLANETAIRE

CRPE

NOTE TECHNIQUE
CRPE/197

UNE BIBLIOTHEQUE DE PROGRAMMES OPTIMISES POUR LE CALCUL DE LA TRANSFORMEE DE FOURIER RAPIDE DE SEQUENCES DE TAILLE $N = 2^n$ (algorithme "Split Radix")

Par

G. YVON

P. DUHAMEL

RPE/ETP
38-40, rue du Général Leclerc
92131 ISSY-LES-MOULINEAUX, FRANCE

CENTRE NATIONAL D'ETUDES
DES TELECOMMUNICATIONS
Centre Paris B

CENTRE NATONAL DE LA
RECHERCHE SCIENTIFIQUE
Département SDU

CENTRE DE RECHERCHE EN PHYSIQUE DE
L'ENVIRONNEMENT TERRESTRE ET PLANETAIRE

NOTE TECHNIQUE CRPE/197
juin 1992

Une bibliothèque de programmes optimisés pour le calcul de la Transformée de Fourier Rapide de séquences de taille $N=2^n$ (algorithme "Split Radix")

par

Gérard YVON et Pierre DUHAMEL

PAB/RPE/ETP
38-40 rue du Général Leclerc
92131 ISSY LES MOULINEAUX

Le Directeur

G. SOMMERIA

Le Chef de département

J.C. BIC

LISTE DE DIFFUSION SYSTEMATIQUE

C N E T

MM.	FENEYROL	Directeur du CNET
	THABARD	Directeur Adjoint du CNET
	FUERXER	Adjoint Militaire au Directeur du CNET
	MEREUR	Directeur des Programmes
	BLOCH	DICET
	THUE	DICET
MME	HENAFF	DICET
MM.	PIGNAL	PAB
	RAMAT	PAB
	ZYLBERSZTEJN	PAB-BAG
	MALOBERTI	PAB-SHM
	HOCQUET	PAB-STC
	THEBAULT	PAB-STS
	SOMMERIA	PAB-RPE
	BERTHELIER	PAB-RPE
MME	PARIS	PAB-RPE
	BAUDIN	PAB-RPE
	BIC	PAB-RPE
	CERISIER	PAB-RPE
	LAVERGNAT	PAB-RPE
	ROBERT	PAB-RPE
	VIDAL-MADJAR	PAB-RPE
	TESTUD	PAB-RPE

C N R S

M.	BERROIR	TOAE
MM.	CHARPENTIER	SPI
	CADET	INSU
MME	COUTURIER	INSU
MME	SAHAL	TOAE
M.	MEGIE	MEN

C N E S

MMES	AMMAR
	DEBOUZY
MM.	BAUDOIN
	FELLOUS
	HERNANDEZ (Toulouse)

E E R M

M.	ANDRE

Bibliothèques

CNET-SDI (2)
CNET-EDB
CNET-RPE (Issy) (5)
CNET-RPE (St Maur) (2)
Observatoire de Meudon
CNRS-SA
CNRS-INIST
CNRS-LPCE

LISTE COMPLEMENTAIRE

DIT/AIN	BITOUN
PAB/RPE	GLOAGUEN
PAB/RPE	LEVY
PAB/RPE	ROBERT
PAB/RPE	REZEAU
PAB/STS	BEHE
PAB/RPE	PIRON
PAB/RPE	MORANE
PAB/RPE	LE LOCH
PAB/RPE	HAUSER
PAB/RPE	DELAHAYE
PAB/RPE	CANU
PAB/RPE	YOUSSEF
PAB/RPE	KABECHE AHMED
PAB/RPE	DELAPORTE
PAB/RPE	RIOUL
PAB/MCT	BON
PAB/STS	ROJAT
PAA/SRE	LEROY
PAA/RGE	DE FROMENT
LAB/SMR	GONZALES
LAB/IFE	TREGUIER
LAB/SMR	LE GOFF
LAB/SMR	RENAN
LAB/IFE	LE CALVEZ
LAA/TSS	CHERBONNEL
LAA/RSH	HOUDOIN
LAA/ELR	ZEDDAM
LAA/SMR	DEBEAU
LAA/ELR	MAGLIULO
LAA/TSS	JULLIEN
LAA/TSS	LE GUYADER
LAA/TSS	PETIT
LAA/TSS	GILLOIRE
LAA/ITP	FRENOT
LAA/TSS	JOUVET
LAA/TSS	MONNE

CNS GRENOBLE

DIR	CHANIAC
TCS/MMA	VAREILLE
CCI/MCC	LARDY
CCI/RCA	LECOURVOISIER
TCI/TME	TEMERSON
CCI/CAT	BALESTRO
CCI/CAT	RENAUDIN

EXTERIEUR

CELAR	MAGUET
ENST BREST	
ENSPS	BECKER

Une bibliothèque de programmes optimisés pour le calcul de la Transformée de Fourier Rapide de séquences de taille $N=2^n$ (algorithme "Split Radix")

Gérard YVON et Pierre DUHAMEL

I INTRODUCTION : pourquoi une nouvelle bibliothèque.

Parmi tous les algorithmes de transformées de Fourier, celui dit à base double ("split radix") fait partie de ceux qui nécessitent le nombre d'opérations le plus faible connu tout en gardant une structure régulière, dans le cas où le nombre de points est une puissance de 2.
L'objet de cette note est tout d'abord de justifier une nouvelle programmation d'algorithmes de Transformées de Fourier, en montrant qu'il est peu probable que l'on améliore sensiblement ces nouveaux algorithmes. Ceux-ci constituant alors un optimum pratique, et étant aussi généraux que les algorithmes précédemment connus, il est logique d'en rechercher une implantation la plus efficace possible, sachant que, du point de vue du nombre d'opérations, on part de la borne la plus "saine" possible. On verra de toute façon que les gains en temps de calcul sont sensibles.

Cette note décrit diverses implantations en FORTRAN et en langage C de cet algorithme. Ces implantations ont été réalisées sur une station VAX/VMS 3100, puis ensuite sur une station SUN et sur un MAC. Les temps de calcul ont étés comparés avec ceux d'une bibliothèque chaque fois que possible.

L'algorithme à base double (ou "Split-Radix")

Considérons un diagramme de l'algorithme de Transformées de Fourier Rapide (TFR, ou FFT pour Fast Fourier Transforms) à base 2. Il peut être directement transformé en diagramme de l'algorithme à base 4 en changeant simplement la valeur des exposants de la racine N-ième de l'unité (appelés *twiddle factors* en anglais). Si l'on effectue cette opération de manière progressive, il est alors apparent qu'à chaque étape, l'algorithme à base 4 est plus efficace pour les termes impairs de la Transformée de Fourier, alors que celui à base 2 reste le meilleur pour les termes pairs. Aussi, on peut penser qu'en restreignant cette opération localement à la partie du diagramme

calculant les termes impairs, on peut améliorer l'algorithme. Il se trouve que c'est en effet le cas.

L'algorithme résultant est dit à base double (*split radix* en anglais) et utilise la décomposition décrite ci-dessous. Soit à calculer la Transformée de Fourier Discrète suivante (DFT en anglais):

$$\text{si (1) } X_k = \sum_{n=0}^{N-1} x_n W_N^{nk} \qquad \left(W_N = \cos\frac{2\pi}{N} - j \sin\frac{2\pi}{N} \right)$$

nous pouvons la décomposer comme suit (décomposition dite "à entrelacement fréquentiel", ou Decimation In Frequency - DIF) :

$$X_{2k} = \sum_{n=0}^{N/2-1} (x_n + x_{n+(N/2)}) W_N^{2nk}$$

$$(2) \quad X_{4k+1} = \sum_{n=0}^{N/4-1} [(x_n - x_{n+(N/2)}) - j\,(x_{n+(N/4)} - x_{n+3(N/2)})]\, W_N^{n} W_N^{4nk}$$

$$X_{4k+3} = \sum_{n=0}^{N/4-1} [(x_n - x_{n+(N/2)}) + j\,(x_{n+(N/4)} - x_{n+3(N/2)})]\, W_N^{3n} W_N^{4nk}$$

La première étape de la décomposition en racine double remplace alors une DFT de longueur N par une DFT de longueur N/2 et deux DFT de longueur N/4 au prix de (N/2 - 4) multiplications complexes (3 multiplications et 3 additions réelles), et 2 multiplications par la racine 8-ème de l'unité (2 multiplications et 2 additions réelles). La DFT de longueur N est obtenue par utilisation successive d'une telle décomposition, jusqu'à la dernière étape où quelques papillons classiques à base 2 (sans *twiddle factors*) sont nécessaires (voir Fig. 1 pour une FFT à racine double de longueur 32).

Puisque l'algorithme à base double est une combinaison des décompositions à base 2 et à base 4 (à chaque étape), on pourrait s'attendre à ce que sa complexité arithmétique soit intermédiaire entre les complexités des algorithmes à base 2 et à base 4. En fait, le nombre de multiplications et d'additions nécessaires est inférieur au nombre correspondant de chacun des autres algorithmes, comme il est montré ci-dessous.

Soit M_n^c (respectivement, A_n^c) le nombre de multiplications réelles (respectivement additions) nécessaires pour calculer une DFT complexe de longueur 2^n avec l'algorithme à racine double. A partir de (2), nous obtenons

$$(3) \qquad M_n^c = M_{n-1}^c + 2\, M_{n-2}^c + 3 \cdot 2^{n-1} - 8$$

et, avec les conditions initiales $M_1 = 0$, $M_2 = 0$, nous obtenons

$$(4) \qquad M_n^c = 2^n(n-3) + 4.$$

En oubliant pour l'instant le nombre d'additions nécessaires pour réaliser les multiplications complexes, le nombre de celles qui restent peut être facilement évalué à $n \cdot 2^{n+1}$, puisque, à chacune des n étapes, un nouveau point est engendré par une addition complexe. Ainsi, puisque le nombre d'additions réelles nécessaires pour calculer un produit complexe est égal au nombre de multiplications, nous avons :

$$(5) \quad A_n^c = n \cdot 2^{n+1} + M_n^c$$

$$A_n^c = 3 \cdot 2^n(n-1) + 4$$

Ces quantités sont calculées dans les Tables I et II, et comparées à celles des algorithmes classiques (base 2, 4, 8, facteurs réels).

L'observation des tables I et II montre que l'algorithme à base double nécessite le nombre le plus faible à la fois de multiplications et d'additions.

Pourquoi mixer deux bases ?

L'idée de base des algorithmes de Transformées de Fourier Rapide consiste à diviser l'ensemble des valeurs de sortie X_k , k=0...N-1, en sous ensembles dont l'union forme l'ensemble complet des valeurs de sortie. Chacun des sous ensemble est alors calculé avec un algorithme adéquat. Par exemple, un étage d'algorithme DIF à base 2 réalise la division suivante :

$$(6) \quad \{X_k\}_{\text{radix-2}} = \{X_{2k1}\} \cup \{X_{2k1+1}\} \qquad k1 = 0...\frac{N}{2} - 1$$

alors que l'algorithme DIF à base 4 (en supposant que m est pair) utilise la division :

$$(7) \quad \{X_k\}_{\text{radix-4}} = \{X_{4k1}\} \cup \{X_{4k1+1}\} \cup \{X_{4k1+2}\} \cup \{X_{4k1+3}\} \quad k1 = 0...\frac{N}{4} - 1$$

Comme il est bien connu, l'évaluation de chacun des sous ensembles, quand il sont choisis correctement, est faite à l'aide d'une DFT de la taille de ces sous ensembles, plus éventuellement quelques opérations auxiliaires sous forme de multiplications par des racines de l'unité.

Maintenant, la division des ensembles de sortie en r ensembles de taille N/r chacun est seulement l'une parmi de nombreuses divisions possibles. Evidemment, tant que l'égalité suivante est vérifiée :

$$(8) \quad \{X_{k1}\} \cup \{X_{k2}\} \cup \ldots \cup \{X_{kl}\} = \{X_k\}$$

c'est à dire si le subdivision en sous ensembles est complet, alors nous avons un algorithme valide pour calculer la DFT de la séquence d'entrée x_n , n = 0, ..., N-1

Dans la suite, nous allons explorer diverses subdivisions et expliquer pourquoi l'algorithme à racine double 2/4 est actuellement un bon choix. Dans le but d'obtenir un algorithme aussi bon que possible pour la DFT de longueur 2^m, nous regarderons d'abord le sous ensemble X_{2k1} dans (2), ce qui donne, à partir de (1),

(9)

$$X_{2k1} = \sum_{n=0}^{N-1} x_n W_N^{2nk1} = \sum_{n=0}^{N/2-1} (x_n + x_{n+(N/2)}) W_{N/2}^{nk1} \qquad k1 = 0...\frac{N}{2} - 1$$

Ainsi, l'ensemble des sorties X_{2k1} est exactement la sortie d'une DFT de longueur N/2, sans aucun coût multiplicatif supplémentaire. Si nous voulons développer le meilleurs algorithme possible pour une DFT de longueur 2^m (en terme de nombre d'opérations), alors nous devons aussi utiliser cet algorithme pour la DFT de taille moitié indiqué en [9]. C'est pourquoi il semble clair que la division qui commencera ainsi

(10)

$$\left\{\cup_k \{X_k\}\right\} = \{X_{2k1}\} \cup \left\{\cup_j \{X_j\}\right\} \qquad k1 = 0...\frac{N}{2} - 1,\ j \neq 2k1,$$

est une bonne stratégie, puisque aucun coût multiplicatif supplémentaire n'est encore apparu; et c'est même la stratégie optimale (d'un point de vue calcul), puisque chaque fois qu'une DFT plus courte peut être trouvée sans coût supplémentaire, alors l'algorithme optimal doit aussi lui être appliqué (argument de type "programmation dynamique"). Effectivement, cette division est exactement ce qui est fait dans les algorithmes optimaux pour les DFT de longueur 2^m [5], [6].

Cependant, il n'y a pas de raison à priori de considérer tous les termes à index impair comme un seul sous-ensemble, comme cela est fait dans une approche à base 2. Une possibilité est d'utiliser une décomposition à base 4 pour les termes d'index impair.

Si les sous ensembles sont choisis convenablement, une telle décomposition demande :

. N/2 DFT de taille 2 où seules les sorties à index impairs sont nécessaires (étage d'entrée)

. approximativement N/2 - 2 multiplications par des racines de l'unité (certaines étant triviales).

. 2 DFT de taille N/4 (étage de sortie).

Ces nombres peuvent être facilement déduits des exemples.

A cette étape de la comparaison entre les algorithme à base 2, 4, et double une question surgit naturellement : est-il possible d'améliorer l'algorithme à base double 2/4 en calculant les termes impairs de la DFT à l'aide d'une racine supérieure (8, par exemple) ?

Il peut être facilement déduit qu'un algorithme à base double 2/8 est plus mauvais qu'un algorithme à base double 2/4 d'un point de vue complexité arithmétique. Plusieurs autres décompositions ont été étudiées, sans pouvoir améliorer l'initiale (2/4). Ainsi, on peut conjecturer qu'en travaillant avec des racines séparées 2 et 4 pour les termes (localement) pairs et impairs de la DFT, nous avons fait descendre le nombre de multiplications à un minimum pour cette classe d'algorithmes. C'est ce que nous vérifions dans le paragraphe suivant.

Comparaisons avec les bornes

Les algorithmes de FFT ont maintenant atteint une grande maturité, au moins dans le cas 1-D, et il est alors possible de faire des remarques sur les éventuelles améliorations qui sont réalisables et celles qui ne le sont pas :

En fait, en utilisant la théorie de la complexité des calculs, il est possible de calculer la borne inférieure du nombre de multiplications nécessaires pour calculer une DFT de longueur donnée. Cette borne est obtenue en décomposant le calcul initial de la DFT en un ensemble de produits de polynômes, opérations pour lesquelles la borne inférieure est connue. Alors, les complexités multiplicatives peuvent être additionnées, pourvu que certaines conditions sur les coefficients des polynômes soient vérifiées.

Complexité multiplicative

Nous nous limitons ici aux FFT dont les longueurs sont des puissances de deux : Winograd [5] a été le premier à obtenir une borne inférieure pour le nombre de multiplications complexes nécessaires pour calculer une DFT de longueur 2^n. Ce travail a ensuite été affiné dans [6], qui procure une borne inférieure réalisable, avec la complexité multiplicative suivante :

$$(11) \quad \mu_c [\, DFT\, 2^n \,] = 2^{n+1} - 2n^2 + 4n - 8$$

Cela signifie qu'il n'existera jamais d'algorithme calculant une DFT de longueur 2^n avec un nombre de multiplications complexes inférieur à celui donné par l'équation (11). De plus, puisque la démonstration est constructive [6], cet algorithme optimal est connu. Malheureusement, il est inutilisable pratiquement pour des longueurs supérieures à 64 (il nécessite alors beaucoup trop d'additions).

La partie inférieure de la figure 2 montre la variation de cette borne inférieure et du nombre de multiplications complexes exigé par quelques algorithmes courants (base 2, base 4, base mixte). Nous voyons clairement que l'algorithme SRFFT suit cette borne inférieure jusqu'à N=64, et la serre de près pour N=128. La divergence est assez rapide après.

Il est aussi possible d'obtenir une borne inférieure réalisable du nombre de multiplications réelles [7].

$$(12) \quad \mu_r [\, DFT\, 2^n \,] = 2^{n+2} - 2n^2 - 2n + 4$$

La variation de cette borne, ainsi que celle du nombre de multiplications réelles exigé par quelques algorithmes courants est donnée dans la partie supérieure de la figure 2. De nouveau, cette borne inférieure réalisable n'est pas d'usage pratique au-dessus d'une certaine limite. Mais, dans ce cas, la limite est beaucoup plus faible : l'algorithme à base double (SRFFT), ainsi que celui à base 4, suit la borne inférieure du nombre de multiplications réelles jusqu'à N = 16 qui est aussi le dernier point où nous pouvons utiliser un algorithme optimal de produits de polynômes (modulo u^2+1) alors que N=32 nécessite un produit optimal modulo u^4+1 qui ne peut s'obtenir qu'au prix d'un nombre irréaliste d'additions.

Il a été montré [6] que les trois algorithmes suivants : algorithme optimal minimisant les multiplications complexes, algorithme optimal minimisant les multiplications réelles, et SRFFT, ont exactement la même structure : elles réalisent la

décomposition en produits de polynômes de la même manière, et diffèrent uniquement dans la manière de calculer les produits de polynômes.

Nous pouvons faire une autre remarque intéressante : nous obtenons le même nombre de multiplications avec SRFFT et avec l'algorithme FFT à facteurs réels et base 2 [8] (qui est, d'un autre point de vue, mal conditionné numériquement, et a besoin d'environ 20% d'additions en plus). Ce résultat est obtenu en utilisant quelques trucs calculatoires pour remplacer les multiplications par des racines de l'unité par des opérations purement réelles ou purement imaginaires. Maintenant se pose la question de savoir s'il est possible de faire la même chose avec la base 4 ? ou même SRFFT ? Un tel résultat fournirait des algorithmes avec encore moins d'opérations. La connaissance de la borne inférieure nous dit que cela est impossible puisque, pour certains points (N=16, par exemple) cela conduirait à un algorithme avec des performances meilleures que la borne inférieure. Le défi pour éventuellement améliorer SRFFT est maintenant le suivant :

La comparaison de SRFFT avec μ_C [DFT 2^n] nous montre qu'aucun algorithme utilisant les multiplications complexes ne pourra améliorer SRFFT de façon significative pour les longueurs < 512. De plus, l'astuce permettant d'obtenir des algorithmes à facteurs réels ne peut pas être appliqué aux racines supérieures à 2 (ou tout au moins pas de la même manière).

La discussion ci-dessus montre ainsi qu'il reste très peu d'approches (encore inconnues) qui pourraient éventuellement améliorer le meilleur algorithme FFT connu de longueur 2^n.

Complexité additive

La situation est légèrement différente si nous nous tournons vers le nombre d'additions :

La plupart des travaux sur la complexité de calcul concerne le nombre de multiplications. En ce qui concerne le nombre d'additions, nous pouvons distinguer entre les additions dues aux multiplications complexes et celles dues aux papillons. Pour le cas $N = 2^n$, il a été montré dans [9], que pour cette dernière catégorie le nombre d'additions utilisées dans les algorithmes actuels est aussi l'optimum. Les différences entre les divers algorithmes sont ainsi dues seulement aux variations du nombre de multiplications complexes. En conclusion, nous pouvons voir que la seule voie pour diminuer le nombre d'additions est de diminuer le nombre de multiplications complexes vraies (qui est proche de la borne inférieure).

Nous pouvons conclure que nous avons maintenant des algorithmes pratiques (principalement WFTA et SRFFT) qui suivent la structure mathématique du problème pour calculer la DFT avec le nombre minimum de multiplications, ainsi qu'une connaissance de leur degré d'optimalité.

II IMPLANTATION

La SRFFT a été implantée en FORTRAN , et comparée sur plusieurs machines avec les librairies disponibles, comme avec des logiciels spécialisés (programmes pour Traitement du Signal Numérique fournis par le Comité DSP de IEEE ASSP). Diverses versions ont été programmées : compactes ou rapides, pour des données complexes, réelles, ou à symétrie hermitienne. Nous avons pris un soin particulier pour utiliser le nombre le plus faible de stockage temporaire à l'intérieur des blocs (les "papillons"), et pour stocker le nombre minimum de sinus et cosinus (un total de N/2 positions mémoire pour les versions rapides). Nous avons même été amenés à consulter le code assembleur généré par le compilateur FORTRAN pour tenter de minimiser les transferts en mémoire, par une utilisation optimale des registres, De plus, dans les versions rapides, nous avons choisi de précalculer les adresses de début de blocs, - cf fig. 1 -, au nombre d'environ [(N+1)/3] pour accélérer le programme qui peut ainsi avoir la structure usuelle en 3 boucles des algorithmes FFT classiques.

Comme dans les algorithmes classiques, nous pouvons distinguer deux types d'algorithmes :

- L'un est de types DIF (entrelacement fréquentiel -fig. 1-) où les données sont fournies en ordre en entrée, et le résultat du calcul obtenu en "bit reverse" (ordre correspondant au renversement binaire des indices de calcul). Il faut donc réarranger ces données dans l'ordre naturel pour une utilisation ultérieure.
- L'autre est de type DIT (entrelacement temporel), et est un algorithme dual du précédent (son graphe est transposé du précédent), et nous devons lui fournir les données après leur avoir fait subir un réarrangement de type "bit reverse", le résultat étant obtenu dans l'ordre naturel.

Avoir ces deux types d'algorithmes à disposition présente l'avantage, dans certaines circonstances, de pouvoir gagner le temps de calcul correspondant à ces réarrangements, temps qui sont non négligeables au total.

Précalculs des sinus et cosinus

Les formules (2) nous montrent que nous aurons besoin des W_N^n et des W_N^{3n} pour n=0,...,N/4-1, c'est à dire des $\cos n\frac{2\pi}{N}$, $\cos 3n\frac{2\pi}{N}$.

Pour une valeur de N donnée, ces différentes valeurs peuvent être précalculées et conservées dans un tableau. Pour les calculs successifs de plusieurs fft de même longueur N, il suffit alors de lire les valeurs voulues dans ce tableau. Même dans cette phase, il est possible de réduire les calculs et la place utilisée.

En effet, ne seront calculés que les cosinus. Pour les sinus nous appliquerons les formules $\sin(n) = \cos(\frac{N}{4} - n)$ et $\sin(3n) = -\cos(3\frac{N}{4} - 3n)$, où $\cos(n) = \cos n\frac{2\pi}{N}$.

Soient w1c et w3c les tableaux contenant les valeurs des cos(n) et cos(3n) respectivement.

Les w1c(n) = cos(n) sont calculés dans le premier quadrant, par pas de $2\pi/N$.
Les w3c s'en déduisent à partir des formules suivantes :

w3c(n) = w1c(3n), n=1,...,N/12
w3c(n) = -w1c(N/2 - 3n), n=N/12+1,...,N/6
w3c(n) = -w1c(3n - N/2), n=N/6+1,...,N/4 - 1

Il suffit donc de calculer les cos(n) pour avoir tous les coefficients utiles pour la suite.

Le précalcul de ces mêmes cos(n) est lui-même simplifié en utilisant les formules de décomposition cos(a+b) et sin(a+b), et en utilisant le fait que cos(N/4 - n) = sin(n).

En calculant seulement $\cos(2\pi/N)$ et $\sin(2\pi/N)$, et sachant que $\cos(N/8) = 1/\sqrt{2}$, par application des formules ci-dessus, nous calculons alors successivement les valeurs w1c(n), w1c(N/4-n), w1c(N/8 + n - 1) et w1c(N/8 - n + 1). pour n=2,...,N/16.

Toutes ces "astuces" nous sont apparues indispensables, car les algorithmes TFR sont devenus tellement rapides que les précalculs éffectués sans précaution prenaient un pourcentage de temps notable.

Une vérification a été faite pour comparer les calculs effectués ainsi avec le calcul direct des cosinus. La différence absolue entre tous les couples de valeurs est toujours inférieure à 10^{-6} pour tous les N de 2^3 à 2^{15}.

Précalculs des adresses de début de blocs

Le premier étage de l'algorithme à base double remplace le calcul d'une TFD de longueur N par celui d'une TFD de longueur N/2, et deux de longueur N/4. On obtient l'algorithme complet par application successive de cette décomposition, jusqu'au dernier étage, où quelques "papillons" usuels à base 2 sont nécessaires.

Cependant, l'algorithme à base double possède un handicap : d'une part, le calcul se propageant dans le graphe avec la forme d'un "L", la progression dans ce graphe est moins régulière, et d'autre part, les blocs de base en forme de L ne doivent pas être appliqués à l'ensemble des données d'un étage. Ces briques de base sont à appliquer à des blocs contigus de données, blocs qui sont en nombre variable dans les étages : un algorithme à base 2 en demanderait 2^i par étage, pour i=0,...,N-1. En ce qui concerne l'algorithme à base double, nous avons les calculs suivants :

Soit Nb(i) le nombre de blocs à l'étage i.
Nous avons donc Nb(i) = Nb(i-1) + 2 Nb(i-2).

Par récurrence, nous pouvons montrer que $Nb(i) = (2^i - (-1)^i)/3$

Pour obtenir une vitesse d'exécution maximale, les adresses de début de blocs, pour chaque étage sont également précalculés, en même temps que les constantes multiplicatives. La taille du tableau jx0 contenant ces adresses est la somme des Nb(i) sur tous les étages où s'applique l'algorithme. On trouve alors :

(2 n - 2) / 3 pour m pair

(2 n - 1) / 3 pour m impair où $n = 2^m$

Dans notre implantation, l'algorithme à base double se déroule sur m-2 étages, le tableau jx0 a donc été dimensionné à nmax/3, nmax étant la taille maximum des suites pouvant être traitées.

Un seul tableau jx0 est nécessaire. il sera lu du début vers la fin pour les algorithmes à décimation fréquentielle, et de la fin vers le début pour les algorithmes à décimation temporelles.

Etage 1 : 1 seul bloc ==> jx0(1) = 1

Etage 2 : 1 seul bloc ==> jx0(2) = 1

Etage 3 : 3 blocs ==> jx0(3) = 1, jx0(4) = N/2+1, jx0(5) =3 N/4 +1

Pour l'étage i, les adresses se calculent de la façon suivante :

- les adresses de l'étage i-1, qui correspondent aux DFT de longueur n/2, qui seront divisées par 2
- les adresses de l'étage i-2, correspondant aux DFT de longueur n/4, qui seront divisées par 4 et décalées de n/4.

L'ensemble des sous-programmes FFT utilisera la même routine d'initialisation, que ces sous-programmes s'appliquent à des données réelles ou complexes, ou qu'ils soient de type DIF ou DIT.

Réarrangement ("bit reversal")

Les temps de calcul des réarrangements ont été optimisés en utilisant des techniques venues de [10] et [17]. Aucune table de valeurs précalculées n'est utilisée.

Temps d'exécution

Des temps absolus sont donnés pour l'ensemble des programmes dans la Table 3 en simple précision et la Table 4 en double précision, sur une station Vax 3100 (analogue à un micro-Vax 3100). Les noms des programmes se décodent de la façon suivante : FFTXYZ

X : c : transformée d'une suite complexe quelconque
c : transformée d'une suite réelle quelconque
h : transformée d'une suite complexe à symétrie hermitienne

Y : f : décimation fréquentielle ("bit reverse" à la fin)
t : décimation temporelle ("bit reverse" avant la FFT)

Z : c : version compacte (sans initialisation)
r : version rapide (appel initial à un programme d'initialisation)

Nous vérifions facilement que les versions pour les données réelles sont deux fois plus rapides que leurs correspondantes complexes (ce qui correspond au rapport de leur complexité arithmétique respective). Elles nécessitent aussi moitié moins de mémoire que les versions complexes (c'est à dire que les calculs se font vraiment "sur place").

Quand nous utilisons ces programmes pour une convolution cyclique de données réelles, les temps sont donnés dans la Table 5, sur la même machine que pour la Table 3. Les programmes c0e6kf et c06fkf proviennent de la bibliothèque NAG pour le calcul de la convolution cyclique de données réelles. La première colonne correspond à un premier appel de nos programmes (initialisation plus FFT du filtre), alors que la seconde correspond aux appels suivants où cette FFT n'est plus utile (les

coefficients sont simplement réutilisés). Les différences dans les temps d'exécution sont notables.

Malheureusement, nous avons eu peu de temps pour réaliser une comparaison avec la librairie IMSL. Des temps de calculs sont donnés dans la Table 6 pour une FFT complexe sur un PC 386 (DOS, 20MHz), et la Table 7 pour une station de travail UNIX SUN3

Conclusion

Nous avons premièrement rappelé quelques résultats récents à propos des algorithmes de FFT dont la longueur est une puissance de 2, en montrant que l'algorithme à base double est efficace d'un point de vue complexité de calcul, et en vérifiant ses performances par rapport aux bornes inférieures théoriques. Puis nous avons fourni quelques temps de calcul d'implantation en FORTRAN de cette FFT à base double sur divers calculateurs, comparé avec des librairies scientifiques. En conclusion sur les temps de calcul, nous pouvons voir qu'ils sont toujours meilleurs que ceux obtenus par des programmes plus compliqués. Des améliorations de plus de 10% peuvent être obtenues par rapport à des programmes hautement optimisés de FFT complexes. Cela ne représente peut-être pas beaucoup, mais puisque nos programmes sont compacts, sur-place, facilement portables et plus courts que ces programmes plus lents, ils sont de bons candidats pour être utilisés. La situation est même meilleure pour des cas spéciaux, tel que les FFT sur des valeurs réelles et la convolution cyclique de données réelles, où les gains typiques sont de l'ordre de 30%, dûs à l'adaptation facile de l'algorithme SRFFT à ces cas. Nous travaillons maintenant à une adaptation de l'algorithme SRFFT sur un calculateur vectoriel. Les premiers résultats sont très prometteurs.

Remerciements :

Nous voulons remercier Y. Meyerfeld, CRPE St Maur, France, qui a fourni certains des temps de calcul utilisés dans cette note.

Figures et tables

Fig. 1. Algorithme à racine double de longueur 32.

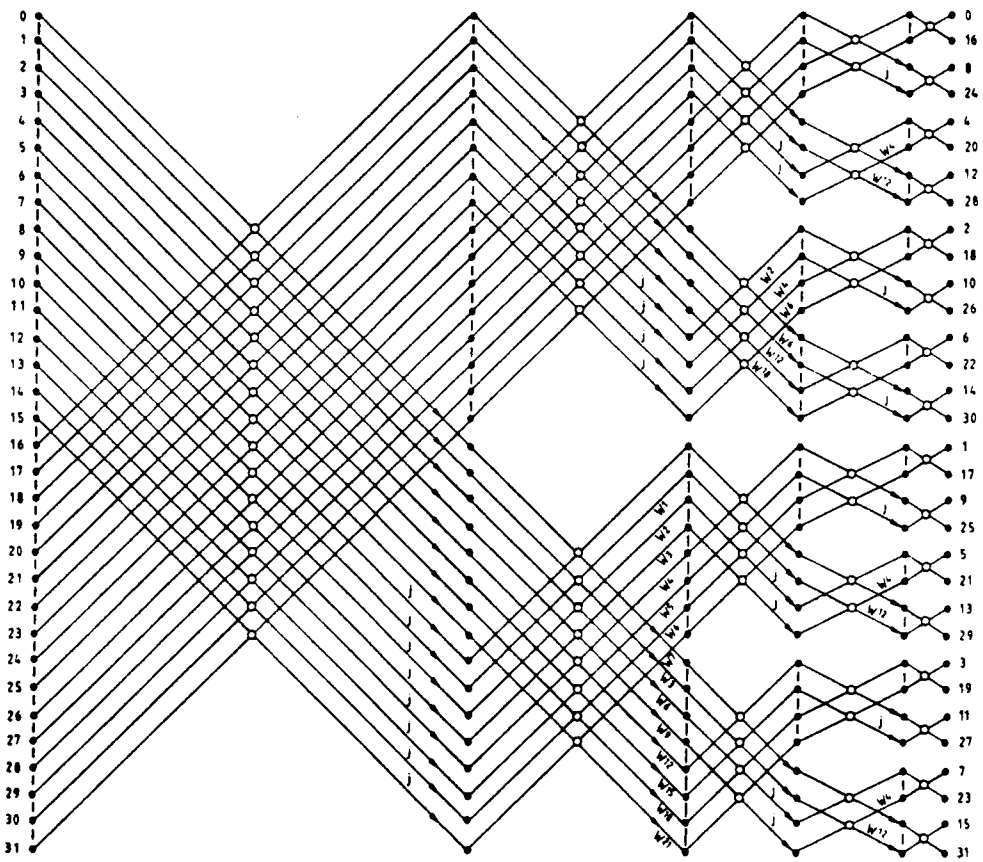


Fig. 2. Nombre de multiplications non-triviales (Mr) ou complexes (Mc) par point de sortie utilisé par plusieurs algorithmes.

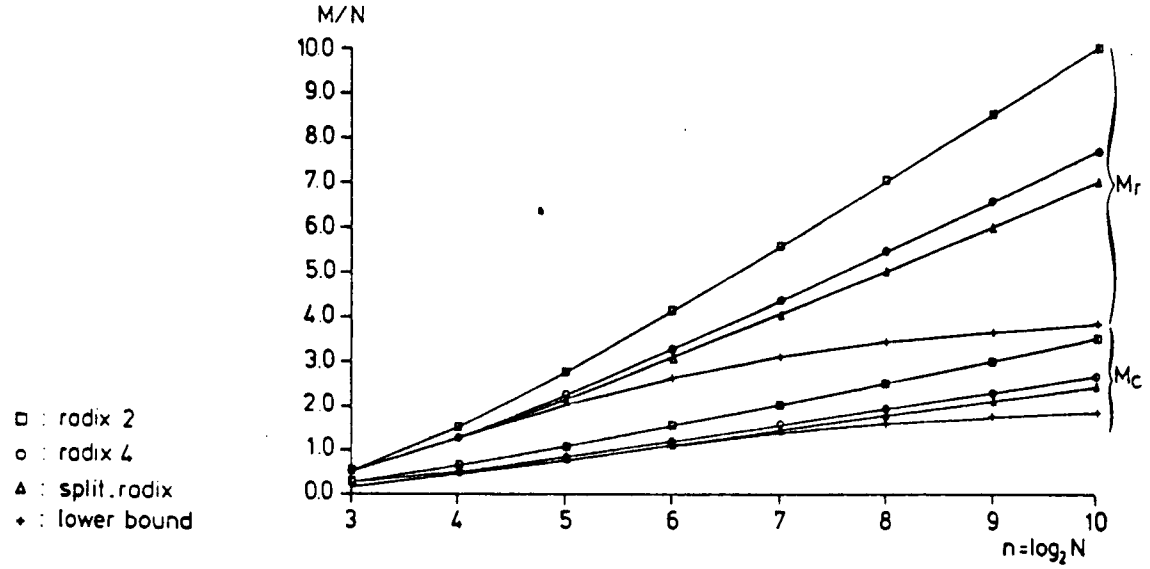


Table 1

Nombre de multiplications réelles non triviales pour calculer une TFD complexe de longueur N.

N	Base 2	Base 4	base 8	Facteurs réels (2,3,4)	Racine double
16	24	20		20	**20**
32	88			68	**68**
64	264	208	**204**	196	**196**
128	712			516	516
256	1800	1392		1284	1284
512	4360		3204	3076	3076
1024	10248	7856		7172	7172
2048				16388	16388

Table 2

Nombre d'additions réelles pour calculer une TFD complexe de longueur N.

N	Base 2	Base 4	base 8	Facteurs réels (2,3,4)	Racine double
16	152	148		148	**148**
32	408			424	**388**
64	1032	976	**972**	1104	**964**
128	2504			2720	2308
256	5896	5488		6464	5380
512	13566		12420	14976	12292
1024	30728	28336		34048	27652
2048				76288	61444

Temps cpu moyen (calculé sur 10 appels, en 1/100 s) des différents programmes de transformées de Fourier à base double, sur une station Vax 3100.

Le temps mesuré pour les versions rapides ne comprend pas l'initialisation, ni le réarrangement

Table 3 : Calculs en simple précision, version FORTRAN.

		complexe				réel		sym. herm.	
m	n	fftcfc	fftctc	fftcfr	fftctr	fftrtc	fftrtr	ffthfc	ffthfr
3	**8**	0.20	0.10	0.10	0.00	0.00	0.10	0.10	0.10
4	16	0.20	0.10	0.10	0.10	0.10	0.00	0.00	0.10
5	32	0.30	0.20	0.20	0.10	0.10	0.30	0.20	0.20
6	64	0.50	0.50	0.30	0.20	0.20	0.20	0.30	0.20
7	128	1.50	1.40	0.50	0.60	0.60	0.40	0.80	0.40
8	256	2.30	2.60	1.00	0.70	1.40	0.60	1.60	0.60
9	512	5.50	5.20	2.50	2.20	2.20	1.30	3.00	1.10
10	1024	11.20	11.10	5.10	4.90	4.90	2.40	5.00	2.50
11	2048	23.80	26.20	12.00	10.80	11.00	5.50	11.00	5.60
12	4096	52.30	51.30	25.40	24.10	23.60	12.00	24.70	12.30
13	8192	108.20	108.90	55.80	52.70	51.60	28.70	52.70	28.30
14	16384	234.40	234.40	129.80	119.00	109.30	61.60	111.30	62.80

Table 3 bis : Calculs en simple précision, version C.

		complexe				réel		sym. herm.	
m	n	fftcfc	fftctc	fftcfr	fftctr	fftrtc	fftrtr	ffthfc	ffthfr
3	**8**	0.10	0.10	0.10	0.10	0.10	0.00	0.10	0.00
4	16	0.20	0.30	0.20	0.10	0.20	0.10	0.20	0.00
5	32	0.50	0.50	0.20	0.10	0.20	0.10	0.10	0.10
6	64	1.10	1.10	0.40	0.10	0.40	0.30	0.60	0.30
7	128	2.30	2.30	0.50	0.40	1.10	0.30	1.30	0.30
8	256	4.70	4.80	1.20	1.10	2.40	0.60	2.70	0.70
9	512	9.90	10.30	2.50	2.60	4.40	1.60	5.70	1.50
10	1024	20.80	21.00	5.90	5.70	9.80	3.30	11.50	3.30
11	2048	44.00	44.00	13.40	12.70	19.60	7.20	24.20	7.30
12	4096	91.20	90.30	28.60	27.60	44.00	15.60	50.90	16.50
13	8192	188.50	187.00	64.20	60.30	90.50	34.90	104.10	36.50
14	16384	399.70	399.30	145.30	140.40	191.20	76.10	216.70	80.70

Table 4 : Calculs en double précision, version FORTRAN.

		complexe				réel		sym. herm.	
m	n	fftcfc	fftctc	fftcfr	fftctr	fftrtc	fftrtr	ffthfc	ffthfr
3	**8**	0.10	0.20	0.00	0.00	0.00	0.10	0.10	0.00
4	**16**	0.30	0.00	0.20	0.10	0.10	0.10	0.20	0.10
5	**32**	0.60	0.60	0.20	0.20	0.30	0.10	0.40	0.20
6	64	1.00	0.90	0.50	0.40	0.40	0.30	0.70	0.20
7	128	2.40	2.20	0.70	1.00	1.00	0.50	1.20	0.00
8	256	4.90	4.70	1.70	2.10	2.00	0.90	2.40	1.00
9	512	10.20	10.00	3.70	4.00	4.40	2.00	4.80	2.00
10	1024	22.00	21.00	8.90	8.90	9.60	4.60	10.60	4.40
11	2048	45.60	45.10	19.80	19.80	20.70	10.00	22.20	9.90
12	4096	96.80	94.80	44.20	44.90	43.60	22.70	47.10	22.10
13	8192	202.80	198.50	98.40	99.70	93.00	50.50	99.30	49.70
14	16384	432.10	418.10	215.90	219.00	197.80	111.80	205.30	109.40

Table 4 bis : Calculs en double précision, version C.

		complexe				réel		sym. herm.	
m	n	fftcfc	fftctc	fftcfr	fftctr	fftrtc	fftrtr	ffthfc	ffthfr
3	**8**	0.20	0.20	0.10	0.20	0.00	0.00	0.00	0.00
4	**16**	0.30	0.40	0.20	0.30	0.10	0.10	0.10	0.10
5	**32**	0.70	0.70	0.30	0.20	0.30	0.20	0.40	0.20
6	**64**	1.40	1.50	0.60	0.70	0.80	0.30	0.90	0.30
7	128	3.30	3.30	1.40	1.50	1.50	0.70	1.60	0.70
8	256	6.60	6.80	3.10	3.00	3.00	1.50	3.50	1.60
9	512	13.80	14.50	6.70	6.50	6.60	3.20	7.30	3.10
10	1024	30.00	30.60	15.40	15.00	14.30	7.10	15.60	7.20
11	2048	64.00	65.80	35.10	34.10	30.30	16.20	33.10	16.60
12	4096	142.40	156.20	87.40	84.70	65.90	35.30	78.60	39.10
13	8192	314.50	305.00	166.70	162.80	137.50	78.00	146.20	79.50
14	16384	591.40	610.00	388.80	361.20	298.20	174.90	307.20	173.20

Table 5 : Temps cpu moyen (en 1/100 s) pour le calcul d'une convolution cyclique en double précision avec SRFFT comparé aux programmes c06ekf et c06fkf de NAG, sur Vax/VMS (convr, c06ekf et c06fkf calculent la convolution cyclique de deux suites réelles, convc la calcule pour deux suites complexes).

m	n	convr (+init)	convr	c06ekf	c06fkf	convc (+init)	convc
3	8	0.20	0.10	0.20	0.30	0.30	0.10
4	16	0.20	0.20	0.60	0.50	0.30	0.20
5	32	0.40	0.40	0.70	0.80	0.60	0.50
6	64	0.70	0.50	1.10	1.30	1.30	1.00
7	128	1.40	1.00	3.00	3.20	2.70	1.70
8	256	3.10	2.00	5.00	5.40	5.90	3.90
9	512	6.80	4.50	10.10	10.30	13.80	9.10
10	1024	15.00	10.10	26.10	27.10	30.80	20.90
11	2048	34.00	22.70	48.40	50.10	67.30	44.60
12	4096	76.40	49.40	100.90	104.00	148.90	98.70
13	8192	167.70	109.20	251.80	257.60	325.00	217.60
14	16384	367.20	239.90	472.90	481.40	763.40	517.00

Table 6 : Comparaison des temps cpu de SRFFT avec IMSL (en secondes) sur un PC 386

n	SRFFT	F2TCF
128	0.05	0.058
256	0.062	0.171
512	0.113	0.390
1024	0.332	0.832
2048	0.769	1.921
4096	1.640	3.949
8192	3.570	9.011

Table 7 : Comparaison des temps cpu de SRFFT avec IMSL (en secondes) sur une station SUN 3 (UNIX)

n	SRFFT	F2TCF
128	0.0064	0.00839
256	0.0144	0.0164
512	0.0308	0.0380
1024	0.07126	0.0824
2048	0.1516	0.1716
4096	0.333196	0.3652
8192	0.701199	0.825198

III MODE D'EMPLOI

Tous les programmes existent en trois versions, les noms de ces programmes et les paramètres d'appel étant les mêmes dans les trois versions :

FORTRAN simple précision : les paramètres x, y et z sont en simple précision, et les calculs s'effectuent en simple précision.

FORTRAN double précision : les paramètres x, y et z sont en double précision, et les calculs s'effectuent en double précision.

C : les paramètres x, y et z sont en simple précision (type float), et les calculs s'effectuent en simple ou double précision, selon l'option de compilation choisie (quand celà est possible. Sur VAX, nous pouvons compiler avec l'option : cc/PRECISION=SINGLE ou DOUBLE).

Précisons de plus que dans cette version C, les fonctions sont prototypées, suivant en cela la norme C/ANSI. Attention donc aux compilateurs qui n'acceptent pas cette norme.

L'ensemble de ces programmes peut être obtenu auprès des auteurs.

Tableau des programmes disponibles

	RAPIDE		COMPACT	
	Décimation en fréquence	Décimation en temps	Décimation en fréquence	Décimation en temps
données complexes	fftcfr(x,y,m)	fftctr(x,y,m)	fftcfc(x,y,m)	fftctc(x,y,m)
données réelles	\/ /\	fftrtr(x,m)	\/ /\	fftrtc(x,m)
données complexes avec symétrie hermitienne i.e. réelle inverse	ffthfr(x,m)	\/ /\	ffthfc(x,m)	\/ /\

Les noms se décriptent de la façon suivante :

les 3 premières lettres, fft, pour "Fast Fourier Transform", bien sûr.

lettre 4 : c --> suite complexe quelconque

r --> suite réelle
h --> suite complexe à symétrie hermitienne

lettre 5 : f --> décimation fréquentielle (réarrangement après calcul)
t -- > décimation temporelle (réarrangement avant calcul)

lettre 6 c --> version compacte
r --> version rapide (nécessite (5n/6 positions mémoire en plus)

Dans le cas de données réelles, le résultat est à symétrie hermitienne, et est fourni sous la forme suivante :

Re X_i, i=0,...,N/2, Im X_{N-i}, i=(N/2)-1,..., N-1

soit Re X0 Re X1 Re X2 Re X3 Re X4 Im X3 Im X2 Im X1 pour N=8

Les données en entrée des versions à symétrie hermitienne doivent être fournies suivant la même convention. Dans ce cas, les sorties sont réelles.

A noter : ce sont des FFT inverses de données à symétrie hermitienne (ce qui est toujours le cas utile : nous pouvons directement faire FFTRTR suivi de FFTHFR, et nous retrouverons la séquence d'entrée multipliée par N)

Autres programmes :

fft(z,n,idir) pour le calcul d'une fft complexe

convr(x,h,n,ifirst) pour la convolution (données réelles)

convc(z,h,n,ifirst) pour la convolution (données complexes)

Calcul d'une FFT (directe ou inverse) d'une suite complexe

L'initialisation et les réarrangements sont gérés par le programme qui utilise, pour le calcul de la fft, le sous-programme fftcfr

Il faut dans ce cas faire l'appel :

```
call fft(z,n,idir)
```

z : tableau contenant la suite complexe de dimension n à transformer, c'est à dire z(1),z(2),...,z(n)
idir = 0 pour fft
différent de 0 pour fft inverse

Les valeurs de la fft sont retournées dans le tableau z, sans normalisation par 1/N, et dans le bon ordre.
Pour le calcul de la transformée inverse, on applique la formule :
IFFT(z) = 1/N j [FFT(jz*)]*, comme indiqué en [16].

Il s'agit là d'une version fournie pour le confort de l'utilisateur disposant déjà de données complexes, mais qui nécessite cependant un temps de calcul légèrement supérieur aux versions "rapides", où les appels doivent être faits directement en fonction des parties réelles et imaginaires.

Calcul d'une FFT :versions compactes

Ces versions n'utilisent pas de précalculs, mais sont moins rapides que les versions rapides (environ 2 fois moins rapides sur Vax)
Ces programmes ne seront utilisés que dans des cas spéciaux, par exemple si la place mémoire est très limitée.

Programmes : fftcfc, fftctc, fftrtc, ffthfc
(sans normalisation par 1/N dans la transformée inverse)
Les résultats sont retournés dans les tableaux x et y dans le bon ordre.

Calcul d'une FFT: versions rapides

Dans ce cas, il faut gérer soi-même les initialisation et les réarrangements.

Les arguments pour les appels des sous-programmes sont les données elles-mêmes (x et y, tableaux de dimension n = 2**m) et m (m <= mmax).

mmax vaut 14 dans les sous-programmes proposés (longueur maximum des transformées = 2**14)

Les différents sous-programmes utilisent des tableaux de travail qui sont définis par des "common". Un seul paramètre permet de les dimensionner : il s'agit de mmax. Cette valeur pourra être modifiée dans chaque sous-programme utilisé, soit pour réduire la place perdue en cas de m petits, soit pour étendre cette place si on veut utiliser m > mmax.

La taille mémoire requise pour ce tableau est de 5*nmax/6 (avec nmax=2**mmax)

Les appels des sous programmes sont décrits ci-après :

Données complexes

FFT sur 2**m points

décimation en fréquence

```
call inifft(m)      ! à faire chaque fois qu'on change de longueur
                    ! seulement une fois par longueur
call fftcfr(x,y,m)
call brxy(x,y,m)    ! "bit reversal"
```

décimation en temps

```
call inifft(m)      ! à faire chaque fois qu'on change de longueur
                    ! seulement une fois par longueur
call brxy(x,y,m)    ! "bit reversal"
call fftctr(x,y,m)
```

FFT inverse sur 2**m points sans normalisation

décimation en fréquence

```
call inifft(m)      ! à faire chaque fois qu'on change de longueur
                    ! seulement une fois par longueur
call fftcfr(y,x,m)  ! il suffit d'échanger les arguments x et y
call brxy(x,y,m)    ! "bit reversal"
```

décimation en temps

```
call inifft(m)      ! à faire chaque fois qu'on change de longueur
                    ! seulement une fois par longueur
call brxy(x,y,m)    ! "bit reversal"
call fftcfr(y,x,m)  ! il suffit d'échanger les arguments x et y
```

Données réelles

FFT directe sur 2**m points

```
call inifft(m)
call brx(x,m)
call fftrtr(x,m)
```

Données complexes à symétrie hermitienne

FFT inverse sur 2**m points

```
call inifft(m)
call ffthfr(x,m)  ! voir les paramètres d'appels plus loin
call brx(x,m)
```

Convolution cyclique (avec normalisation)

Convolution complexe

```
call convc(z,h,n,ifirst)
```

Convolution réelle

```
call convr(x,h,n,ifirst)
```

Paramètres d'appel des sous-programmes

Données complexes

ft(z,n,idir) : Transformée de Fourier directe (idir=0) ou inverse (idir différent de 0) du tableau complexe z de dimension n
(n doit être compris entre 8 et 2**mmax et être une puissance de 2)

fftcfc(x,y,m) : FFT d'une séquence complexe (x + j*y) de taille 2**m
x = tableau des parties réelles des données
y = tableau des parties imaginaires des données

fftctc(x,y,m) : FFT d'une séquence complexe (x + j*y) de taille 2**m
x = tableau des parties réelles des données
y = tableau des parties imaginaires des données

fftcfr(x,y,m) : FFT d'une séquence complexe (x + j*y) de taille 2**m
x = tableau des parties réelles des données
y = tableau des parties imaginaires des données
Version rapide, qui nécessite donc un appel à inifft.
Les réarrangements sont à effectuer après l'appel à fftcfr

fftctr(x,y,m) : FFT d'une séquence complexe (x + j*y) de taille 2**m
x = tableau des parties réelles des données
y = tableau des parties imaginaires des données
Version rapide, qui nécessite donc un appel à inifft.
Les réarrangements sont à effectuer avant l' appel à fftctr

Données réelles

fftrtc(x,m) : Calcule la transformée de Fourier d'un tableau x de réels de dimension n = 2**m
Ordre des données en sortie :
[Re(0),Re(1),...,Re(n/2),Im(n/2-1),...,Im(1)]

fftrtr(x,m) : Calcule la transformée de Fourier d'un tableau x de réels de dimension n = 2**m
Ordre des données en sortie :
[Re(0),Re(1),...,Re(n/2),Im(n/2-1),...,Im(1)]
Version rapide, (nécessite un appel à inifft avant), et décimation en temps , qui doit donc être précédé d'un réarrangement (brx).

Données complexes à symétrie hermitienne

ffthfc(x,m) : FFT inverse portant sur une séquence x à symétrie hermitienne de taille n = 2**m

Ordre des données en entrée :
[Re(0),Re(1),...,Re(n/2),Im(n/2-1),...,Im(1)]
NB : ceci est en fait la FFT inverse de la FFT sur données réelles
Les données en sortie ne sont pas normalisées

ffthfr(x,m) : FFT inverse portant sur une séquence x à symétrie hermitienne de taille n = 2**m

Ordre des données en entrée :
[Re(0),Re(1),...,Re(n/2),Im(n/2-1),...,Im(1)]
Version rapide (nécessite un appel à inifft avant), et décimation en fréquences (réarrangement des données à effectuer après).
NB : ceci est en fait la FFT inverse de la FFT sur données réelles.
Les données en sortie ne sont pas normalisées par 1/n.

Précalculs

inifft(m) : Précalcul des cos et des débuts de blocs.

sous programme à appeler avant utilisation des versions rapides. Il est commun à toutes les versions, et un seul appel à inifft est nécessaire.

Réarrangement ("bit reversal")

brxy(x,y,m) : Réarrangement des données, cas complexe

brx(x,m) : Réarrangement des données, cas réel

x = tableau des parties réelles des données de dimension 2**m
y = tableau des parties imaginaires des données

Convolution

Convolution complexe

convc(z,h,n,ifirst) :
inifft + fftcfr + mul2z + fftctr(inverse) + normalisation par 1/n

Convolution cyclique de la séquence complexe z par la séquence complexe h, toutes deux de longueurs n
(n doit être compris entre 8 et 2**mmax et être une puissance de 2)

z et h étant les suites à convoluer, Z et H les transformées de Fourier, on utilise la propriété z*h = TF-1(ZH)

ifirst permet de ne calculer qu'une fois H quand on doit calculer z*h pour plusieurs z et h fixé.

ifirst = 1 :
- en entrée, z et h contiennent les suites à convoluer
- en sortie, z contient le produit de convolution z*h
et h contient la transformée de Fourier (divisée par n)
de l'entrée h, donc H/n

ifirst différent de 1 :
- en entrée, z contient la suite à convoluer, mais h
contient déjà H/n calculé lors du précédent appel
avec ifirst = 1.
- en sortie, z contient le produit de convolution z*h
h est inchangé (H/n).

Convolution réelle

convr(x,h,n,ifirst) :
inifft + brx + fftrtr + mul2sh + ffthfr + brx + normalisation par 1/n

Convolution cyclique des séquences réelles x et h, toute deux de longueur n
(n doit être compris entre 8 et 2**mmax et être une puissance de 2)

x et h étant les suites à convoluer, X et H les transformées
de Fourier, on utilise la propriété x*h = TF-1(XH)

ifirst permet de ne calculer qu'une fois H quand on doit
calculer x*h pour plusieurs x et h fixé.
ifirst = 1 :
- en entrée, x et h contiennent les suites à convoluer
- en sortie, z contient le produit de convolution x*h
et h contient la transformée de Fourier (divisée par n)
de l'entrée h, donc H/n

ifirst différent de 1 :
- en entrée, x contient la suite à convoluer, mais h
contient déjà H/n calculé lors du précédent appel
avec ifirst = 1.
- en sortie, x contient le produit de convolution x*h
h est inchangé (H/n).

mul2sh(x,h,m) : Calcule le produit de deux suites complexes x et h de longueur n = 2**m, ayant la symétrie hermitienne.
Les données dans x et h doivent rangées comme en sortie d'une fft réelle, c'est à dire
[Re(0),Re(1),...,Re(n/2),Im(n/2-1),...,Im(1)]
Le produit est retourne dans x, h est inchangé

mul2z(x,y,h1,h2) : Calcule le produit de deux suites complexes (x,y) et (h1,h2) de longueur n = 2**m.
Le produit est retourne dans (x,y)
(h1,h2) est inchangé

Appel à partir d'un programme en C

Il est possible d'appeler les sous programmes FORTRAN à partir d'un programme en C.

Il faut mettre les déclarations suivantes (avant le main) :

```
extern void fftcfc();
extern void fftcfr();
extern void fftctc();
extern void fftctr();
extern void fftrtc();
extern void fftrtr();
extern void ffthfc();
extern void ffthfr();
extern void inifft();
extern void brxy();
extern void brx();
```

Les paramètres fortran étant passés par adresse, les appels se feront ainsi :

```
float *x, *y ou float x[], y[];
int m;

inifft(&m);
fftcfr(x,y,&m);
brxy(x,y,&m);
```

Les données sont déclarées en simple ou en double précision selon la version utilisée.

```
float = simple précision Fortran = 32 bits
double = double précision Fortran = 64 bits
```

Evidemment, si m est déclaré comme pointeur, les appels gardent la même forme qu'en fortran.

```
float *x, *y;
int *m;

inifft(m);     ! m doit être initialisé avant l'appel.
fftcfr(x,y,m);
brxy(x,y,m);
```

III REFERENCES

[1] P. DUHAMEL "Implementation of "split-radix", FFT algorithms for complex, real, and real-symetric data" IEEE Trans on ASSP, Vol. 34, N° 2, pp. 285-295, April 1986

[2] Z. WANG "Fast Algorithms for the discrete W transforms and the discrete Fourier transform", IEEE Trans. on ASSP, Vol 32, pp. 803-816, Aug. 1984

[3] M.VETTERLI, H. J. NUSSBAUMER, "Simple FFT and DCT algorithms with reduced number of operations", Signal Processing, Vol 6, pp. 267-278, July 1984

[4] J. B. MARTENS "Recursive cyclotomic factorization : A new algorithm for calculating the DFT", IEEE Trans. on ASSP, Vol 32, pp. 750-761, Apr. 1984

[5] S. WINOGRAD "Arithmetic complexity of computations", SIAM CBMS-NSF series, N° 33, SIAM, Philadelphia, 1980

[6] P. DUHAMEL "Algorithms meeting the lower bounds on the multiplicative complexity of length-2n DFT's and their connection with practical algorithms", IEEE Trans. on ASSP, Vol. 38, pp. 1504-1511, Sept. 1990

[7] M. T. HEIDEMAN, C. S. BURRUS, "On the number of multiplications necessary to compute a length-2n DFT", IEEE Trans ASSP, Vol. 34, pp. 91-95, Feb. 1986

[8] C. RADER, N. BRENNER "A new principle for Fast Fourier Transform", IEEE Trans. on ASSP, Vol. 24, pp. 264-265, 1976

[9] C. H.PAPADIMITRIOU " Optimality of the Fast Fourier Transform", J. Ass. Comput. Mach., Vol. 26, pp. 95-102, Jan. 1979

[11] H. SORENSEN, M. T. HEIDEMAN, C. S. BURRUS : "On computing the split-radix FFT", IEEE Trans on ASSP, vol 34, n° 1, Feb 1986

[12] C. S. BURRUS : "A Duhamel-Hollman split-radix DIF FFT (Ref : Electronics letters, jan. 5, 1984)" Rice Univ. Dec. 1984

[13] P. DUHAMEL : "Un algorithme de transformation de Fourier rapide à double base", Annales des télécommunications, tome 40, n° 9-10, sept-oct 1985

[15] P. DUHAMEL, M. VETTERLI : "Improved Fourier and Hartley Transform Algorithms Application to Cyclic Convolution of Real Data", IEEE Trans on ASSP, vol 35, n° 6, June 1987

[16] P. DUHAMEL, B. PIRON, et J.M. ETCHETO : "On Computing the Inverse DFT", IEEE Trans on ASSP, Vol 36. N° 2, pp 285-286, Feb. 1988

[17] A. A. YONG : "A Better FFT Bit-Reversal Algorithm Without Tables", IEEE Trans on ASSP, Vol 39, N° 10, pp. 2365-2367, Oct. 1991

IV LES PROGRAMMES

Vous trouverez danc ce chapitre l'ensemble du code source FORTRAN de tous les programmes, dans l'ordre ci-dessous.

fft(z,n,idir)
convc(z,h,n,ifirst)
convr(x,h,n,ifirst)
inifft(m)
brx(x,m)
brxy(x,y,m)
fftcfc(x,y,m)
fftctc(x,y,m)
fftcfr(xr,xi,m)
fftctr(x,y,m)
fftrtc(x,m)
fftrtr(x,m)
ffthfc(x,m)
ffthfr(x,m)
mul2sh(x,h,m)
mul2z(x,y,h1,h2,m)

```
C
C-----------------------------------------------------------------------------
C       Transformée de Fourier
C       subroutine fft(z,n,idir)       Pierre Duhamel
C
C       Ce sous-programme calcule la transformée de Fourier
C       directe (idir=0) ou inverse (idir diffErent de 0)
C       du tableau complexe z de dimension n
C       (n doit être compris entre 8 et 2**mmax
C       et être une puissance de 2)
C-----------------------------------------------------------------------------
C
        subroutine fft(z,n,idir)
C
        parameter (mmax=14,nmax=2**mmax)
        complex z(*)
        dimension x(nmax),y(nmax)
        save
        data id/0/
C
C   id contrôle l'initialisation (inifft) quand n change
C
        m=1
        nc=2
        do 10 i=1,mmax
            nc=2*nc
            m=m+1
            if(n.eq.nc) go to 20
10      continue
        print *,'Erreur (subroutine fft): dimension du tableau incorrecte'
        return
20      if(id.ne.m) then
          call inifft(m)
          id=m
        endif
        if(idir.eq.0) then
          do 30 i=1,n
            x(i)=real(z(i))
30          y(i)=aimag(z(i))
          call fftcfr(x,y,m)
          call brxy(x,y,m)
          do 40 i=1,n
40          z(i)=cmplx(x(i),y(i))
        else
          do 50 i=1,n
            x(i)=aimag(z(i))
50          y(i)=real(z(i))
          call fftcfr(x,y,m)
          call brxy(x,y,m)
          do 60 i=1,n
60          z(i)=cmplx(y(i),x(i))/n
        endif
        return
        end
```

```
C
C-------------------------------------------------------------------------------c
C       CONVOLUTION COMPLEXE
C
C       Convolution cyclique de la séquence complexe z par la
C       séquence complexe h, toutes deux de longueurs n
C       (n doit être compris entre 8 et 2**mmax
C       et être une puissance de 2)
C
C       z et h étant les suites à convoluer, Z et H les transformées
C       de Fourier, on utilise la propriété z*h = TF-1(ZH)
C
C       ifirst permet de ne calculer qu'une fois H quand on doit
C       calculer z*h pour plusieurs z et h fixé.
C       ifirst = 1 :
C           - en entrée, z et h contiennent les suites à convoluer
C           - en sortie, z contient le produit de convolution z*h
C           et h contient la transformée de Fourier (divisée par n)
C           de l'entrée h, donc H/n
C       ifirst different de 1 :
C           - en entrée, z contient la suite à convoluer, mais h
C           contient déjà H/n calculé lors du précédent appel
C           avec ifirst = 1.
C           - en sortie, z contient le produit de convolution z*h
C           h est inchangé (H/n).
C-------------------------------------------------------------------------------c
C
        subroutine convc(z,h,n,ifirst)
C
        parameter (mmax=14,nmax=2**mmax)
        complex z(*),h(*)
        dimension x(nmax),y(nmax),h1(nmax),h2(nmax)
        save
        data id/0/
C
C   id controle l'initialisation (inifft) quand n change
C
        m=1
        nc=2
        do 10 i=1,mmax
            nc=2*nc
            m=m+1
            if(n.eq.nc) go to 20
10      continue
        print *,'Erreur (subroutine convc): dimension du tableau incorrecte'
        return
C
20      if(id.ne.m) then
            call inifft(m)
            id=m
        endif
C
        do 30 i=1,n
            x(i)=real(z(i))
            y(i)=aimag(z(i))
            h1(i)=real(h(i))
            h2(i)=aimag(h(i))
30      continue
```

```
C
      if(ifirst.eq.1) then
          call fftcfr(h1,h2,m)
          do 35 i=1,n
              h1(i)=h1(i)/n
              h2(i)=h2(i)/n
              h(i)=cmplx(h1(i),h2(i))
35        continue
      endif
C
      call fftcfr(x,y,m)
      call mul2z(x,y,h1,h2,m)
      call fftctr(y,x,m)
      do 40 i=1,n
40        z(i)=cmplx(x(i),y(i))
C
      return
      end
```

```
C
C-------------------------------------------------------------------------------------------c
C          CONVOLUTION REELLE
C     subroutine convr(x,h,n,ifirst)
C
C          Convolution cyclique des séquences réelles x et h, toute deux
C          de longueur n
C          (n doit être compris entre 8 et 2**mmax
C          et être une puissance de 2)
C
C          x et h étant les suites à convoluer, X et H les transformées
C          de Fourier, on utilise la propriété x*h = TF-1(XH)
C
C          ifirst permet de ne calculer qu'une fois H quand on doit
C          calculer x*h pour plusieurs x et h fixé.
C          ifirst = 1 :
C               - en entrée, x et h contiennent les suites à convoluer
C               - en sortie, z contient le produit de convolution x*h
C               et h contient la transformée de Fourier (divisée par n)
C               de l'entrée h, donc H/n
C          ifirst different de 1 :
C               - en entrée, x contient la suite à convoluer, mais h
C               contient déjà H/n calculé lors du précédent appel
C               avec ifirst = 1.
C               - en sortie, x contient le produit de convolution x*h
C               h est inchangé (H/n).
C-------------------------------------------------------------------------------------------c
C
          subroutine convr(x,h,n,ifirst)
C
          parameter (mmax=14,nmax=2**mmax)
          dimension x(nmax),h(nmax)
          save
          data id/0/
C
C     id controle l'initialisation (inifft) quand n change
C
          m=1
          nc=2
          do 10 i=1,mmax
               nc=2*nc
               m=m+1
               if(n.eq.nc) go to 20
10        continue
          print *,'Erreur (subroutine convr): dimension du tableau incorrecte'
          return
C
20        if(id.ne.m) then
               call inifft(m)
               id=m
          endif
          if(ifirst.eq.1) then
               call brx(h,m)
               call fftrtr(h,m)
               do 30 i=1,n
                    h(i)=h(i)/n
30             continue
          endif
```

```
C
      call brx(x,m)
      call fftrtr(x,m)
      call mul2sh(x,h,m)
      call ffthfr(x,m)
      call brx(x,m)
C
      return
      end
```

```
C
C-------------------------------------------------------------------------------------------c
C       Précalcul des cos et des débuts de blocs.
C
C       sous programme à appeler avant utilisation des versions
C       rapides. Il est commun à toutes les versions, et un seul
C       appel à inifft est nécessaire.
C
C       Taille mémoire requise :
C           w1c    :        nmax/4
C           w3c    :        nmax/4
C           jx0 :   nmax/3
C                   ---------
C           total  :  5*nmax/6
C-------------------------------------------------------------------------------------------c
C
      subroutine inifft(m)
C
            parameter (mmax=14,nmax=2**mmax,ns3=nmax/3,ns4=nmax/4)
            dimension w1c(ns4),w3c(ns4),jx0(ns3)
            double precision ang, c, s
            common /precos/w1c,w3c,jx0
C
            if ((m.lt.3).or.(m.gt.mmax)) then
                print *
                print *,'/// INIFFT  INIFFT  INIFFT  INIFFT  INIFFT ///'
                print *,'///  m n''est pas dans les limites permises ///'
                print *,'///       2         < m <',mmax+1,'       ///'
                print *,'/// INIFFT  INIFFT  INIFFT  INIFFT  INIFFT ///'
                print *
                stop
            end if
            n = 2**m
            xn = n
            dpi= 6.283185307179586
            ang = dpi/xn
            n2=n/2
            n2p1=n2+1
            n4=n/4
            n8=n4/2
            n16=n8/2
            n6=n/6
            n12=n6/2
C
      c = cos(ang)
      s = sin(ang)
      w1c(1) = c
      w1c(n4-1) = s
     w1c(n/8) = 0.707106781186547
      do 40 i=2,n16
            w1c(i) = w1c(i-1)*c-w1c(n4-i+1)*s
            w1c(n4-i) = w1c(n4-i+1)*c+w1c(i-1)*s
            w1c(n8+i-1) = w1c(n8+i-2)*c-w1c(n8-i+2)*s
            w1c(n8-i+1) = w1c(n8-i+2)*c+w1c(n8+i-2)*s
40    continue
            do 32 i=1,n12
                w3c(i) = w1c(3*i)
32    continue
```

```
            do 33 i=n12+1,n6
                w3c(i) = -w1c(n2-3*i)
 33   continue
            do 34 i=n6+1,n4-1
                w3c(i) = -w1c(3*i-n2)
 34   continue
C
      jx0(1) = 1
      jx0(2) = 1
      jx0(3) = 1
      jx0(4) = n2p1
      jx0(5) = 3*n4+1
      ip = 5
      nb = 3
      lnb = 1
      do 30 i=1,m-4
            do 20 j=ip+1,ip+nb
                jx0(j) = jx0(j-nb)/2+1
 20         continue
            ip = ip+nb
            ip1=-(nb+nb+lnb)
            do 10 j=ip+1,ip+lnb
                jx0(j) = jx0(j+ip1)/4+n2p1
                jx0(j+lnb) = jx0(j)+n4
 10         continue
            ip = ip+lnb+lnb
            llnb = lnb
            lnb = nb
            nb = lnb+llnb+llnb
 30   continue
            return
      end
```

```
C
C-------------------------------------------------------------------------c
C       Réarrangement des données
C       Cas réel
C       (bit reversal, ou inversion binaire)
C
C       A connection between bit-reversal and matrix
C       transposition : hardware and software consequences
C       P. DUHAMEL
C       IEEE Trans. on ASSP, Nov 1990
C-------------------------------------------------------------------------c
C
        subroutine brx(x,m)
        dimension x(*)
C
        n = 2**m
        m1 = m/2
        n1 = 2**m1
        ia1=n1/2
        ia2=n/n1
        ia3=ia1+ia2
        nh = n/2
        do 10 ipair=0,(m-m1-m1)*n1,n1
            ibr = 0
            xt = x(ipair+ia1+1)
            x(ipair+ia1+1) = x(ipair+ia2+1)
            x(ipair+ia2+1) = xt
            do 20 i=2+ipair,ia1+ipair
                k = nh
                if(k.gt.ibr) go to 23
24              ibr = ibr-k
                k = k/2
                if(k.le.ibr) go to 24
23              ibr = ibr+k
                xt = x(ibr+i+ia1)
                x(ibr+i+ia1) = x(ibr+i+ia2)
                x(ibr+i+ia2) = xt

                jbr = 0
                if (m.lt.4) goto 20
                do 30 j=ibr+ipair+1,ibr+i-1
                    jbri=jbr+i
                    xt = x(jbri)
                    x(jbri) = x(j)
                    x(j) = xt
                    xt = x(jbri+ia1)
                    x(jbri+ia1) = x(j+ia2)
                    x(j+ia2) = xt
                    xt = x(jbri+ia2)
                    x(jbri+ia2) = x(j+ia1)
                    x(j+ia1) = xt
                    xt = x(jbri+ia3)
                    x(jbri+ia3) = x(j+ia3)
                    x(j+ia3) = xt
                    k = nh
                    if(k.gt.jbr) go to 33
34                  jbr = jbr-k
                    k = k/2
```

```
                if(k.le.jbr) go to 34
33              jbr = jbr+k
C
30            continue
20          continue
10        continue
          return
          end
```

```
C
C-------------------------------------------------------------------------------c
C       Réarrangement des données
C       Cas complexe
C       (bit reversal, ou inversion binaire)
C
C       A connection between bit-reversal and matrix
C       transposition : hardware and software consequences
C       P. DUHAMEL
C       IEEE Trans. on ASSP, Nov 1990
C-------------------------------------------------------------------------------c
C
        subroutine brxy(x,y,m)
        dimension x(*),y(*)
C
        n = 2**m
        m1 = m/2
        n1 = 2**m1
        ia1=n1/2
        ia2=n/n1
        ia3=ia1+ia2
        nh = n/2
        do 10 ipair=0,(m-m1-m1)*n1,n1
            ibr = 0
            xt = x(ipair+ia1+1)
            x(ipair+ia1+1) = x(ipair+ia2+1)
            x(ipair+ia2+1) = xt
            xt = y(ipair+ia1+1)
            y(ipair+ia1+1) = y(ipair+ia2+1)
            y(ipair+ia2+1) = xt
            do 20 i=2+ipair,ia1+ipair
                k = nh
                if(k.gt.ibr) go to 23
24              ibr = ibr-k
                k = k/2
                if(k.le.ibr) go to 24
23              ibr = ibr+k
                xt = x(ibr+i+ia1)
                x(ibr+i+ia1) = x(ibr+i+ia2)
                x(ibr+i+ia2) = xt
                xt = y(ibr+i+ia1)
                y(ibr+i+ia1) = y(ibr+i+ia2)
                y(ibr+i+ia2) = xt
C
                jbr = 0
                if (m.lt.4) goto 20
                do 30 j=ibr+ipair+1,ibr+i-1
                    jbri=jbr+i
                    xt = x(jbri)
                    x(jbri) = x(j)
                    x(j) = xt
                    xt = y(jbri)
                    y(jbri) = y(j)
                    y(j) = xt
C
                    xt = x(jbri+ia1)
                    x(jbri+ia1) = x(j+ia2)
                    x(j+ia2) = xt
```

```
                xt = y(jbri+ia1)
                y(jbri+ia1) = y(j+ia2)
                y(j+ia2) = xt
C
                xt = x(jbri+ia2)
                x(jbri+ia2) = x(j+ia1)
                x(j+ia1) = xt
                xt = y(jbri+ia2)
                y(jbri+ia2) = y(j+ia1)
                y(j+ia1) = xt
C
                xt = x(jbri+ia3)
                x(jbri+ia3) = x(j+ia3)
                x(j+ia3) = xt
                xt = y(jbri+ia3)
                y(jbri+ia3) = y(j+ia3)
                y(j+ia3) = xt
                k = nh
                if(k.gt.jbr) go to 33
34              jbr = jbr-k
                k = k/2
                if(k.le.jbr) go to 34
33              jbr = jbr+k
C
30          continue
20        continue
10      continue
        return
        end
```

```
C
C-----------------------------------------------------------------------c
C       Calcule la SRFFT sur des données complexes (x,y)
C       Version compacte, et "décimation en fréquences".
C
C       On computing the split-radix FFT
C       SORENSEN, HEIDEMAN, BURRUS
C       IEEE Trans on ASSP, vol 34, n! 1, Feb 1986
C
C       A Duhamel-Hollman split-radix DIF FFT
C       (Ref : Electronics letters, jan. 5, 1984)
C       Complex input and output data in arrays x et y
C       Length is n = 2 ** m
C       C.S. Burrus,        Rice Univ.      Dec. 1984
C-----------------------------------------------------------------------c
C
        subroutine fftcfc(x,y,m)
        real  x(*),y(*)
        parameter (mmax=14)
C
        if ((m.lt.3).or.(m.gt.mmax)) then
            print *,'m n"est pas dans les limites permises'
            print *,'  2 < m <',mmax+1
            stop
        endif
        n = 2**m
        n2 = 2*n
        do 10 k = 1, m-1
            n2 = n2/2
            n4 = n2/4
            e = 6.283185307179586/n2
            a = 0
            do 20 j = 1, n4
                a3 = 3*a
                cc1 = cos(a)
                ss1 = sin(a)
                cc3 = cos(a3)
                ss3 = sin(a3)
                a = j*e
                is = j
                id = 2*n2
40              do 30 i0 = is, n-1, id
                    i1 = i0 + n4
                    i2 = i1 + n4
                    i3 = i2 + n4
C                   general 6 mult.
                    r1      = x(i0) - x(i2)
                    x(i0)   = x(i0) + x(i2)
                    r2      = x(i1) - x(i3)
                    x(i1)   = x(i1) + x(i3)
                    s1      = y(i0) - y(i2)
                    y(i0)   = y(i0) + y(i2)
                    s2      = y(i1) - y(i3)
                    y(i1)   = y(i1) + y(i3)
C
                    s3      = r1 - s2
                    r1      = r1 + s2
                    s2      = r2 - s1
```

```
                    r2      = r2 + s1
                    x(i2)   = r1*cc1 - s2*ss1
                    y(i2)   = -s2*cc1 -r1*ss1
                    x(i3)   = s3*cc3 + r2*ss3
                    y(i3)   = r2*cc3 - s3*ss3
30                continue
                  is = 2*id - n2 + j
                  id = 4*id
                  if (is.lt.n) goto 40
20              continue
10          continue
C
C---------------------last stage, length-2 butterfly-------------------
C
            is = 1
            id = 4
50          do 60 i0 = is, n, id
                i1  = i0 + 1
                r1  = x(i0)
                x(i0)   = r1 + x(i1)
                x(i1)   = r1 - x(i1)
                r1  = y(i0)
                y(i0)   = r1 + y(i1)
                y(i1)   = r1 - y(i1)
60          continue
            is = 2*id - 1
            id = 4*id
            if (is.lt.n) goto 50
C
            call  brxy(x,y,m)
            return
            end
```

```
C
C-------------------------------------------------------------------------c
C       Calcule la SRFFT sur des données complexes (x,y)
C       Version compacte, et "décimation en temps"
C
C       On computing the split-radix FFT
C       SORENSEN, HEIDEMAN, BURRUS
C       IEEE Trans on ASSP, vol 34, n! 1, Feb 1986
C
C       A Duhamel-Hollman split-radix DIT FFT
C       (Ref : Electronics letters, jan. 5, 1984)
C       Complex input and output data in arrays x et y
C       Length is n = 2 ** m
C       H.V. Sorensen,      Rice Univ.      Jan. 4 1985
C-------------------------------------------------------------------------c
C
        subroutine fftctc(x,y,m)
        real  x(*),y(*)
        parameter (mmax=14)
C
        if ((m.lt.3).or.(m.gt.mmax)) then
            print *,'m n"est pas dans les limites permises'
            print *,'  2 < m <',mmax+1
            stop
        endif
C
        call  brxy(x,y,m)
C
C-----------------length-2  butterfly-------------------------c
C
        n = 2**m
        is=1
        id=4
70      do 60 i0=is,n,id
            i1=i0+1
            r1=x(i0)
            x(i0)=r1+x(i1)
            x(i1)=r1-x(i1)
            r1=y(i0)
            y(i0)=r1+y(i1)
            y(i1)=r1-y(i1)
60      continue
        is=2*id-1
        id=4*id
        if (is.lt.n) go to 70
C
C---------------L SHAPED BUTTERFLIES ----------
C
        n2=2
        do 10 k=2,m
            n2=n2*2
            n4=n2/4
            e=6.283185307179586/n2
            a=0
            do 20 j=1,n4
                a3=3*a
                cc1=cos(a)
                ss1=sin(a)
```

```
                cc3=cos(a3)
                ss3=sin(a3)
                a=j*e
                is=j
                id=2*n2
40              do 30 i0=is,n-1,id
                    i1=i0+n4
                    i2=i1+n4
                    i3=i2+n4
C
                    r1=x(i2)*cc1+y(i2)*ss1
                    s1=y(i2)*cc1-x(i2)*ss1
                    r2=x(i3)*cc3+y(i3)*ss3
                    s2=y(i3)*cc3-x(i3)*ss3
                    r3=r1+r2
                    r2=r1-r2
                    r1=s1+s2
                    s2=s1-s2
C
                    x(i2)=x(i0)-r3
                    x(i0)=x(i0)+r3
                    x(i3)=x(i1)-s2
                    x(i1)=x(i1)+s2
                    y(i2)=y(i0)-r1
                    y(i0)=y(i0)+r1
                    y(i3)=y(i1)+r2
                    y(i1)=y(i1)-r2
C
30              continue
                is=2*id-n2+j
                id=4*id
                if (is.lt.n) go to 40
20          continue
10      continue
        return
        end
```

```
C
C-------------------------------------------------------------------------------c
C          FFT d'une séquence complexe (xr + j*xi) de taille 2**m
C          Algorithme SRFFT, version rapide, qui nécessite donc
C          un appel à inifft, et décimation en fréquences
C          (les réarrangements sont à effectuer après l'appel à
C          la FFT)
C
C          Un algorithme de transformation de Fourier rapide
C          à double base.
C          Pierre DUHAMEL
C          Annales des télécommunications, tome 40, n! 9-10,
C          sept-oct 1985
C-------------------------------------------------------------------------------c
C
      subroutine fftcfr(xr,xi,m)
C
      parameter (mmax=14,nmax=2**mmax,ns3=nmax/3,ns4=nmax/4)
      dimension xr(*),xi(*)
     dimension w1c(ns4),w3c(ns4),jx0(ns3)
     common /precos/w1c,w3c,jx0
C
      data c21/0.707106781186547/
      data cm/-1.414213562373095/
C
            if ((m.lt.3).or.(m.gt.mmax)) then
                print *,'m n''est pas dans les limites permises'
                print *,'  2 < m <',mmax+1
                stop
            endif
            n = 2**m
            nd4 = n/4
    n1 = n/2
C
     istep = 1
     ib = 0
     nb = 1
     lnb = 0
C
C           Boucle sur les etages
C
     do 30 j=1,m-3
      n1 = n1/2
C
C             Boucle sur les blocs
C
      do 10 i=ib+1,ib+nb
C
C-------------0-mult  butterfly------------
C
            i0 = jx0(i)
            i1 = i0+n1
            i2 = i1+n1
            i3 = i2+n1
            sp = xr(i1)-xr(i3)
            s = xi(i1)-xi(i3)
            xr(i1) = xr(i1)+xr(i3)
            r = xr(i0)-xr(i2)
```

```
            xr(i3) = r-s
            xr(i0) = xr(i0)+xr(i2)
            xr(i2) = r+s
            r = xi(i0)-xi(i2)
            xi(i0) = xi(i0)+xi(i2)
            xi(i1) = xi(i1)+xi(i3)
            xi(i3) = r+sp
            xi(i2) = r-sp
C
C-------------general 6-mult butterfly--------
C
            jstep = 0
            do 5 ij=i0+1,i0+n1-1
                i1 = ij+n1
                i2 = i1+n1
                i3 = i2+n1
                r = xi(ij)-xi(i2)
                xi(ij) = xi(ij)+xi(i2)
                s = xi(i1)-xi(i3)
                xi(i1) = xi(i1)+xi(i3)
                sp = xr(i1)-xr(i3)
                t = r-sp
                r = r+sp
                xr(i1) = xr(i1)+xr(i3)
                rp = xr(ij)-xr(i2)
                sp = rp+s
                jstep = jstep+istep
                xi(i2) = t*w1c(jstep)-sp*w1c(nd4-jstep)
                xr(ij) = xr(ij)+xr(i2)
                xr(i2) = sp*w1c(jstep)+t*w1c(nd4-jstep)
                rp = rp-s
                xi(i3) = r*w3c(jstep)+rp*w3c(nd4-jstep)
                xr(i3) = rp*w3c(jstep)-r*w3c(nd4-jstep)
 5          continue
C
 10           continue
C
      ib = ib+nb
      llnb = lnb
      lnb = nb
      nb = lnb+llnb+llnb
            istep = istep+istep
 30         continue
C
C
     do 50 i=ib+1,ib+nb
C
C-------------8-point  butterfly--------------
C
              i0 = jx0(i)
              i1 = i0+1
              i3 = i0+3
              i5 = i0+5
              i7 = i0+7
              r = xi(i1)-xi(i5)
              xi(i1) = xi(i1)+xi(i5)
              s = xi(i3)-xi(i7)
             xi(i3) = xi(i3)+xi(i7)
```

```
            sp = xr(i3)-xr(i7)
            xr(i3) = xr(i3)+xr(i7)
             rp = xr(i1)-xr(i5)
            t = s+rp
            y2 = (t-sp+r)*c21
            x2 = t*cm+y2
            xr(i1) = xr(i1)+xr(i5)
            t = r+sp
            sp = (t-rp+s)*c21
            rp = t*cm+sp
C
             i2 = i0+2
             i4 = i0+4
             i6 = i0+6
            r = xi(i0)-xi(i4)
            xi(i0) = xi(i0)+xi(i4)
            s = xi(i2)-xi(i6)
             xi(i2) = xi(i2)+xi(i6)
             t = xr(i2)-xr(i6)
            t1 = r+t
            xi(i6) = t1+rp
            xi(i7) = t1-rp
            t = r-t
            xi(i4) = t+x2
            xi(i5) = t-x2
            xr(i2) = xr(i2)+xr(i6)
            r = xr(i0)-xr(i4)
            t = r-s
            xr(i6) = t+sp
            xr(i7) = t-sp
            xr(i0) = xr(i0)+xr(i4)
            s = r+s
            xr(i4) = s+y2
            xr(i5) = s-y2
 50   continue
C
C--------------4-point  dft----------------
C
     ib = ib+nb
     nb = nb+lnb+lnb
     do 60 i=ib+1,ib+nb
             i0 = jx0(i)
             i1 = i0+1
             i2 = i1+1
             i3 = i2+1
            r = xi(i0)+xi(i2)
            s = xi(i0)-xi(i2)
            t = xi(i1)+xi(i3)
            xi(i0) = r+t
            u = xi(i1)-xi(i3)
            xi(i1) = r-t
            r = xr(i1)+xr(i3)
            t = xr(i1)-xr(i3)
            xi(i3) = s+t
            xi(i2) = s-t
            t = xr(i0)-xr(i2)
            xr(i3) = t-u
            s = xr(i0)+xr(i2)
```

```
            xr(i0) = s+r
            xr(i1) = s-r
            xr(i2) = t+u
 60   continue
C
      return
      end
```

```
C
C-------------------------------------------------------------------------------c
C       FFT d'une séquence complexe (xr + j*xi) de taille 2**m
C       Algorithme SRFFT, version rapide (qui nécessite donc
C       un appel à inifft) et décimation en temps
C       (les réarrangements sont à effectuer avant l'appel à
C       la FFT)
C
C       On computing the split-radix FFT
C       SORENSEN, HEIDEMAN, BURRUS
C       IEEE Trans on ASSP, vol 34, n! 1, Feb 1986
C
C       A Duhamel-Hollman split-radix DIT FFT
C       (Ref : Electronics letters, jan. 5, 1984)
C       H.V. Sorensen,      Rice Univ.      Jan. 4 1985
C-------------------------------------------------------------------------------c
C
        subroutine fftctr(x,y,m)
C
        parameter (mmax=14,nmax=2**mmax,ns3=nmax/3,ns4=nmax/4)
        dimension x(*),y(*)
        dimension w1c(ns4),w3c(ns4),jx0(ns3)
        common /precos/w1c,w3c,jx0
        data rac2s2/0.707106781186547/
C
        if ((m.lt.3).or.(m.gt.mmax)) then
            print *,'m n"est pas dans les limites permises'
            print *,'  2 < m <',mmax+1
            stop
        endif
C
C----------------length-4  butterfly
C
        n = 2**m
        nd4 = n/4
        sgn=(-1)**m
        nb=(n/2+sgn)/3
        lnb=(n-sgn)/3
        ib=n/6
        do 60 i=ib+1,ib+nb
            i0=jx0(i)
            i1 = i0+1
            i2 = i1+1
            i3 = i2+1
            r3=x(i2)+x(i3)
            r2=x(i2)-x(i3)
            r1=y(i2)+y(i3)
            r4=y(i2)-y(i3)
            t0=x(i0)+x(i1)
            t1=x(i0)-x(i1)
            x(i2)=t0-r3
            x(i0)=t0+r3
            x(i3)=t1-r4
            x(i1)=t1+r4
            t0=y(i0)+y(i1)
            t1=y(i0)-y(i1)
            y(i2)=t0-r1
            y(i0)=t0+r1
```

```
                y(i3)=t1+r2
                y(i1)=t1-r2
60          continue
C
            llnb=lnb
            lnb=nb
            nb=(llnb-lnb)/2
            ib=ib-nb
C
C-------------8-point  butterfly--------------
C
            do 40 i=ib+1,ib+nb
C---------------without  mult-------------
                i0=jx0(i)
                i4 = i0+4
                i5 = i4+1
                i6 = i5+1
                i7 = i6+1
                r1=x(i4)+x(i5)
                r2=x(i6)+x(i7)
                r4=r1-r2
                t1=x(i4)-x(i5)
                t2=x(i6)-x(i7)
                r3=r1+r2
                x(i4)=x(i0)-r3
                x(i0)=x(i0)+r3
C
                r1=y(i4)+y(i5)
                r2=y(i6)+y(i7)
                t3=y(i4)-y(i5)
                t4=y(i6)-y(i7)
                i2 = i0+2
                y(i6)=y(i2)+r4
                y(i2)=y(i2)-r4
                r4=r1-r2
                x(i6)=x(i2)-r4
                x(i2)=x(i2)+r4
                r3=r1+r2
                y(i4)=y(i0)-r3
                y(i0)=y(i0)+r3
C---------------with  2  mult
                r1=t1+t3
                r2=t2-t4
                r3=(r1-r2)*rac2s2
                i1 = i0+1
                x(i5)=x(i1)-r3
                x(i1)=x(i1)+r3
                r3=(r1+r2)*rac2s2
                i3 = i1+2
                y(i7)=y(i3)+r3
                y(i3)=y(i3)-r3
                r1=t3-t1
                r2=t2+t4
                r3=(r1+r2)*rac2s2
                x(i7)=x(i3)-r3
                x(i3)=x(i3)+r3
                r3=(r1-r2)*rac2s2
                y(i5)=y(i1)-r3
```

```
                y(i1)=y(i1)+r3
 40          continue
C
C----------------L SHAPED BUTTERFLIES ----------
C
            n2=4
            istep=n/16
C
C boucle sur les etages
C
            do 30 j=1,m-3
              llnb=lnb
              lnb=nb
              nb=(llnb-lnb)/2
              ib=ib-nb
              n1=n2
              n2=n2+n2
              n3=n1+n2
C
C boucle sur les blocs de l etage j
              do 10 i=ib+1,ib+nb
C  0_mult
                 i0=jx0(i)
                 i2=i0+n1+n1
                 i3=i2+n1
C
                 r3=x(i2)+x(i3)
                 r2=x(i2)-x(i3)
                 x(i2)=x(i0)-r3
                 x(i0)=x(i0)+r3
                 s2=y(i2)-y(i3)
                 i1=i0+n1
                 x(i3)=x(i1)-s2
                 x(i1)=x(i1)+s2
                 r1=y(i2)+y(i3)
                 y(i2)=y(i0)-r1
                 y(i0)=y(i0)+r1
                 y(i3)=y(i1)+r2
                 y(i1)=y(i1)-r2
C
C general butterfly
C
                 jstep = 0
                 do 5 ij=jx0(i)+1,jx0(i)+n1-1
                    jstep=jstep+istep
                    s1=y(ij+n2)*w1c(jstep)-x(ij+n2)*w1c(nd4-jstep)
                    s2=y(ij+n3)*w3c(jstep)+x(ij+n3)*w3c(nd4-jstep)
                    s3=s1+s2
                    r1=x(ij+n2)*w1c(jstep)+y(ij+n2)*w1c(nd4-jstep)
                    r2=x(ij+n3)*w3c(jstep)-y(ij+n3)*w3c(nd4-jstep)
                    r3=r1+r2
                    x(ij+n2)=x(ij)-r3
                    x(ij)=x(ij)+r3
                    r2=r1-r2
                    y(ij+n3)=y(ij+n1)+r2
                    y(ij+n1)=y(ij+n1)-r2
                    s2=s1-s2
                    x(ij+n3)=x(ij+n1)-s2
```

```
                x(ij+n1)=x(ij+n1)+s2
                y(ij+n2)=y(ij)-s3
                y(ij)=y(ij)+s3
C
5             continue
10          continue
            istep=istep/2
30        continue
          return
          end
```

```
C
C-------------------------------------------------------------------------------c
C          Calcule la transformée de Fourier d'un tableau x de
C          réels de dimension 2**m
C          Ordre des données en sortie :
C              [ Re(0),Re(1),...,Re(N/2),Im(N/2-1),...,Im(1) ]
C          Algorithme SRFFT, version compacte, et décimation en
C          temps
C
C          Improved Fourier and Hartley Transform Algorithms:
C          Application to CycliC Convolution of Real Data
C          P. DUHAMEL, M. VETTERLI
C          IEEE Trans on ASSP, vol ASSP-35, n! 6, June 1987
C-------------------------------------------------------------------------------c
C
          subroutine fftrtc(x,m)
C
          parameter (mmax=14)
          dimension x(*)
          data rac2s2/0.707106781186547/
C
          if ((m.lt.3).or.(m.gt.mmax)) then
              print *,'m n''est pas dans les limites permises'
              print *,'  2 < m <',mmax+1
              stop
          endif
C
          call brx(x,m)
C
C----------------length two transforms-----------
C
          n = 2**m
          is = 1
          id = 4
 10     do 20 i0=is,n,id
          i1 = i0+1
          r1 = x(i0)
          x(i0) = r1+x(i1)
          x(i1) = r1-x(i1)
 20     continue
      is = 2*id-1
      id = 4*id
      if(is.lt.n) go to 10
C
C-----------------other butterflies-------------
C
     n2=2
     do 100 k=2,m
          n2 = n2*2
          n4 = n2/4
C--------------without mult------------
          is = 1
          id = 2*n2
 30        do 40 i0=is,n-1,id
              i1 = i0+n4
              i2 = i1+n4
              i3 = i2+n4
              t0 = x(i2)+x(i3)
```

```
                x(i3) =-x(i2)+x(i3)
                x(i2) = x(i0)-t0
                x(i0) = x(i0)+t0
40          continue
            is = 2*id-n2+1
            id = 4*id
            if(is.lt.n) go to 30
            if(n4.lt.2) go to 100
C-----------with 2 real mult
            is = n4/2+1
            id = 2*n2
50          do 60 i0=is,n-1,id
                i1 = i0+n4
                i2 = i1+n4
                i3 = i2+n4
                t1 = (x(i2)-x(i3))*rac2s2
                t2 = (x(i2)+x(i3))*rac2s2
                x(i2) = -t2-x(i1)
                x(i3) = -t2+x(i1)
                x(i1) = x(i0)-t1
                x(i0) = x(i0)+t1
60          continue
            is = 2*id-n2+n4/2+1
            id = 4*id
            if(is.lt.n) go to 50
            e = 6.283185307179586/n2
            a = e
            if(n4.lt.4) go to 100
            do 90 j=2,n4/2
                a3 = 3.*a
                cc1 = cos(a)
                ss1 = sin(a)
                cc3 = cos(a3)
                ss3 = sin(a3)
                a = j*e
                is = j
                id = 2*n2
70              do 80 ia0=is,n-1,id
C---------------with 6 real mult----------
                    ib1 = ia0+n4
                    ia1 = ib1-j-j+2
                    ib0 = ia1+n4
                    ia2 = ib1+n4
                    ia3 = ia2+n4
                    ib2 = ib0+n4
                    ib3 = ib2+n4
                    c2 = x(ia2)*cc1+x(ib2)*ss1
                    d2 = -x(ia2)*ss1+x(ib2)*cc1
                    c3 = x(ia3)*cc3+x(ib3)*ss3
                    d3 = -x(ia3)*ss3+x(ib3)*cc3
                    t1 = c2+c3
                    c3 = c2-c3
                    t2 = d2-d3
                    d3 = d2+d3
                    x(ia2) = -x(ib0)+d3
                    x(ib2) = -x(ib1)-c3
                    x(ia3) = x(ib1)-c3
                    x(ib3) = x(ib0)+d3
```

```
                x(ib1) = x(ia1)+t2
                x(ib0) = x(ia0)-t1
                x(ia0) = x(ia0)+t1
                x(ia1) = x(ia1)-t2
80          continue
            is = 2*id-n2+j
            id = 4*id
            if(is.lt.n) go to 70
90        continue
100   continue
        return
        end
```

```
C
C-------------------------------------------------------------------------------c
C           Calcule la transformee de Fourier d'un tableau x de
C           reels de dimension 2**m
C           Ordre des donnees en sortie :
C               [ Re(0),Re(1),...,Re(N/2),Im(N/2-1),...,Im(1) ]
C           Algorithme SRFFT, version rapide, (nécessite un appel à
C           inifft avant), et décimation en temps , qui doit donc
C           être précédé d'un réarrangemnt (brx).
C
C           Improved Fourier and Hartley Transform Algorithms:
C           Application to CycliC Convolution of Real Data
C           P. DUHAMEL, M. VETTERLI
C           IEEE Trans on ASSP, vol ASSP-35, n! 6, June 1987
C-------------------------------------------------------------------------------c
C
      subroutine fftrtr(x,m)
C
            parameter (mmax=14,nmax=2**mmax,ns3=nmax/3,ns4=nmax/4)
            dimension x(*)
            dimension w1c(ns4),w3c(ns4),jx0(ns3)
            common /precos/w1c,w3c,jx0
            data rac2s2/0.707106781186547/
C
            if ((m.lt.3).or.(m.gt.mmax)) then
                print *,'m n''est pas dans les limites permises'
                print *,'  2 < m <',mmax+1
                return
            endif
C
C-----------------length-4  transforms-----------
C
            n = 2**m
            nd4 = n/4
            sgn=(-1)**m
            nb=(n/2+sgn)/3
            lnb=(n-sgn)/3
            ib=n/6
C
      do 20 i=ib+1,ib+nb
                i0=jx0(i)
                i1 = i0+1
                i2 = i1+1
                i3 = i2+1
                r1 = x(i0)+x(i1)
                t0 = x(i2)+x(i3)
                x(i3) = x(i3)-x(i2)
                x(i1) = x(i0)-x(i1)
                x(i2) = r1-t0
                x(i0) = r1+t0
 20   continue
C
            llnb=lnb
            lnb=nb
            nb=(llnb-lnb)/2
            ib=ib-nb
C
C-------------8-point  butterfly--------------
```

```
C
              do 30 i=ib+1,ib+nb
C--------------without  mult------------
                 i0=jx0(i)
                 i4 = i0+4
                 i5 = i0+5
                 i6 = i0+6
                 i7 = i0+7
                 r1 = x(i4)-x(i5)
                 r3 = x(i4)+x(i5)
                 r2 = x(i7)-x(i6)
                 r4 = x(i6)+x(i7)
                 t0 = r3+r4
                 x(i6) = r4-r3
                 x(i4) = x(i0)-t0
                 x(i0) = x(i0)+t0
C---------------with 2 real mult
                 t1 = (r1+r2)*rac2s2
                 t2 = (r2-r1)*rac2s2
                 i3 = i0+3
                 x(i5) = t2-x(i3)
                 x(i7) = t2+x(i3)
                 i1 = i0+1
                 x(i3) = x(i1)-t1
                 x(i1) = x(i1)+t1
 30          continue
C
C-----------------other  butterflies-------------
C
             istep=n/16
             n8 = 1
             n4 = 2
              n2 = 4
C
C -  boucle sur les etages
C
     do 100 k=4,m
               llnb=lnb
               lnb=nb
               nb=(llnb-lnb)/2
               ib=ib-nb
               n8 = n4
               n4 = n2
               n2 = n2+n2
C
C -- boucles sur les blocs
               do 60 i=ib+1,ib+nb
C---------------without  mult------------
                 i0=jx0(i)
                 i1 = i0+n4
                 i2 = i1+n4
                 i3 = i2+n4
                 t0 = x(i2)+x(i3)
                 x(i3) = -x(i2)+x(i3)
                 x(i2) = x(i0)-t0
                 x(i0) = x(i0)+t0
C---------------with 2 real mult
                 i0=i0+n8
```

```
                i1 = i0+n4
                i2 = i1+n4
                i3 = i2+n4
                t1 = (x(i3)-x(i2))*rac2s2
                t2 = -(x(i2)+x(i3))*rac2s2
                x(i2) = t2-x(i1)
                x(i3) = t2+x(i1)
                x(i1) = x(i0)+t1
                x(i0) = x(i0)-t1
 60         continue
C
C – boucle sur les autres points
C
              do 80 i = ib+1, ib+nb
                 jstep=0
              do 90 j=1,n8-1
                      jstep=jstep+istep
                      ia0=jx0(i)+j
C-----------------with 6 real mult--------------
                 ia2 = ia0+n2
                 ib2 = ia2+n4-j-j
                 c2 = x(ia2)*w1c(jstep)+x(ib2)*w1c(nd4-jstep)
                 d2 = -x(ia2)*w1c(nd4-jstep)+x(ib2)*w1c(jstep)
                 ia3 = ia2+n4
                 ib3 = ib2+n4
                 c3 = x(ia3)*w3c(jstep)-x(ib3)*w3c(nd4-jstep)
                 d3 = x(ia3)*w3c(nd4-jstep)+x(ib3)*w3c(jstep)
                 ib1 = ia0+n4
                 t1 = c2+c3
                 c3 = c2-c3
                 x(ib2) = -x(ib1)-c3
                 x(ia3) = x(ib1)-c3
                 t2 = d2-d3
                 ia1 = ib1-j-j
                 x(ib1) = x(ia1)+t2
                 x(ia1) = x(ia1)-t2
                 d3 = d2+d3
                 ib0 = ia1+n4
                 x(ia2) = -x(ib0)+d3
                 x(ib3) = x(ib0)+d3
                 x(ib0) = x(ia0)-t1
                 x(ia0) = x(ia0)+t1
C
C
 90         continue
 80     continue
            istep=istep/2
 100   continue
C
      return
      end
```

```
C
C-------------------------------------------------------------------------c
C    FFT inverse portant sur une séquence à symétrie
C         hermitienne x de taille 2**m
C         Ordre des données en entrée :
C             [ Re(0),Re(1),...,Re(N/2),Im(N/2-1),...,Im(1) ]
C         Version compacte, décimation en fréquences
C         Sortie non normalisée.
C
C         Real-Valued Fast Fourier Transform Algorithms
C         SORENSEN, JONES, HEIDEMAN, BURRUS
C         IEEE Trans on ASSP, Vol 35, n! 6, June 1987
C-------------------------------------------------------------------------c
C
          subroutine ffthfc(x,m)
          parameter (mmax=14)
          real  x(*)
C
          if ((m.lt.3).or.(m.gt.mmax)) then
              print *,'m n"est pas dans les limites permises'
              print *,'  2 < m <',mmax+1
              stop
          endif
C
C -- L shaped butterflies --
C
          n = 2**m
          n2=2*n
          do 10 k=1,m-1
              is=0
              id=n2
              n2=n2/2
              n4=n2/4
              n8=n4/2
              e=6.283185307179586/n2
17            do 15 i=is,n-1,id
                  i1=i+1
                  i2=i1+n4
                  i3=i2+n4
                  i4=i3+n4
                  t1=x(i1)-x(i3)
                  x(i1)=x(i1)+x(i3)
                  x(i2)=x(i2)+x(i2)
                  x(i3)=t1-2*x(i4)
                  x(i4)=t1+2*x(i4)
                  if (n4.eq.1) goto 15
                  i1=i1+n8
                  i2=i2+n8
                  i3=i3+n8
                  i4=i4+n8
                  t1=(x(i2)-x(i1))/sqrt(2.0)
                  t2=(x(i4)+x(i3))/sqrt(2.0)
                  x(i1)=x(i1)+x(i2)
                  x(i2)=x(i4)-x(i3)
                  x(i3)=2*(-t2-t1)
                  x(i4)=2*(-t2+t1)
15            continue
              is=2*id-n2
```

```
                id=4*id
                if (is.lt.n-1) goto 17
                a=e
                do 20 j=2,n8
                    a3=3*a
                    cc1=cos(a)
                    ss1=sin(a)
                    cc3=cos(a3)
                    ss3=sin(a3)
                    a=j*e
                    is=0
                    id=2*n2
40                  do 30 i=is,n-1,id
                        i1=i+j
                        i2=i1+n4
                        i3=i2+n4
                        i4=i3+n4
                        i5=i+n4-j+2
                        i6=i5+n4
                        i7=i6+n4
                        i8=i7+n4
                        t1=x(i1)-x(i6)
                        x(i1)=x(i1)+x(i6)
                        t2=x(i5)-x(i2)
                        x(i5)=x(i2)+x(i5)
                        t3=x(i8)+x(i3)
                        x(i6)=x(i8)-x(i3)
                        t4=x(i4)+x(i7)
                        x(i2)=x(i4)-x(i7)
                        t5=t1-t4
                        t1=t1+t4
                        t4=t2-t3
                        t2=t2+t3
                        x(i3)=t5*cc1+t4*ss1
                        x(i7)=-t4*cc1+t5*ss1
                        x(i4)=t1*cc3-t2*ss3
                        x(i8)=t2*cc3+t1*ss3
30                  continue
                    is=2*id-n2
                    id=4*id
                    if (is.lt.n-1) goto 40
20              continue
10          continue
C
C -- Length two butterflies --
C
            is=1
            id=4
70          do 60 i0=is,n,id
                i1=i0+1
                r1=x(i0)
                x(i0)=r1+x(i1)
                x(i1)=r1-x(i1)
60          continue
            is=2*id-1
            id=4*id
            if (is.lt.n) goto 70
C
```

```
call brx(x,m)
return
end
```

```
C
C-------------------------------------------------------------------------------------c
C    FFT inverse portant sur une séquence à symétrie
C          hermitienne x  de taille 2**m
C          Ordre des données en entrée :
C              [ Re(0),Re(1),...,Re(N/2),Im(N/2-1),...,Im(1) ]
C          Version rapide (nécessite un appel à inifft avant), et décimation
C          en  fréquences (réarrangement des données à effectuer après)
C          Sortie non normalisée.
C
C          Real-Valued Fast Fourier Transform Algorithms
C          SORENSEN, JONES, HEIDEMAN, BURRUS
C          IEEE Trans on ASSP, Vol 35, n! 6, June 1987
C-------------------------------------------------------------------------------------c
C
          subroutine ffthfr(x,m)
C
          parameter (mmax=14,nmax=2**mmax,ns3=nmax/3,ns4=nmax/4)
          real  x(*)
          dimension w1c(ns4),w3c(ns4),jx0(ns3)
          common /precos/w1c,w3c,jx0
          data rac2s2/0.707106781186547/
          data rac2/1.414213562373094/
C
          if ((m.lt.3).or.(m.gt.mmax)) then
              print *,'m n''est pas dans les limites permises'
              print *,'  2 < m <',mmax+1
              stop
          endif
          n = 2**m
          nd4 = n/4
          n4=n/2
          n8=n4/2
          istep=1
          ib=0
          nb=1
          lnb=0
C
C -- L shaped butterflies --
          do 10 k=1,m-3
              n4=n8
              n8=n4/2
              do 15 ijk=ib+1,ib+nb
                  jstep=0
                  i1=jx0(ijk)
                  i2=i1+n4
                  i3=i2+n4
                  i4=i3+n4
                  t1=x(i1)-x(i3)
                  x(i1)=x(i1)+x(i3)
                  x(i2)=x(i2)+x(i2)
                  x(i3)=t1-2*x(i4)
                  x(i4)=t1+2*x(i4)
                  i1=i1+n8
                  i2=i2+n8
                  i3=i3+n8
                  i4=i4+n8
                  t1=(x(i2)-x(i1))*rac2
```

```
                t2=-(x(i4)+x(i3))*rac2
                x(i1)=x(i1)+x(i2)
                x(i2)=x(i4)-x(i3)
                x(i3)=t2-t1
                x(i4)=t2+t1
                do 20 j=1,n8-1
                    jstep=jstep+istep
                    i=jx0(ijk)
                    i1=i+j
                    i2=i1+n4
                    i3=i2+n4
                    i4=i3+n4
                    i5=i+n4-j
                    i6=i5+n4
                    i7=i6+n4
                    i8=i7+n4
                    t1=x(i1)-x(i6)
                    x(i1)=x(i1)+x(i6)
                    t2=x(i5)-x(i2)
                    x(i5)=x(i2)+x(i5)
                    t3=x(i8)+x(i3)
                    x(i6)=x(i8)-x(i3)
                    t4=x(i4)+x(i7)
                    x(i2)=x(i4)-x(i7)
                    t5=t1-t4
                    t1=t1+t4
                    t4=t2-t3
                    t2=t2+t3
                    x(i3)=t5*w1c(jstep)+t4*w1c(nd4-jstep)
                    x(i7)=-t4*w1c(jstep)+t5*w1c(nd4-jstep)
                    x(i4)=t1*w3c(jstep)+t2*w3c(nd4-jstep)
                    x(i8)=t2*w3c(jstep)-t1*w3c(nd4-jstep)
20              continue
15          continue
            istep=istep+istep
            ib=ib+nb
            llnb=lnb
            lnb=nb
            nb=lnb+llnb+llnb
10      continue
C
C-------------8-point  butterfly--------------
C
        do 30 i=ib+1,ib+nb
            i0=jx0(i)
            i1 = i0+1
            i4 = i0+4
            t0=x(i0)-x(i4)
            x(i0)=x(i0)+x(i4)
            i2 = i1+1
            x(i2)=x(i2)+x(i2)
            i5 = i0+5
            i6 = i0+6
            t2=x(i6)+x(i6)
            t1=t0+t2
            t2=t0-t2
            i3 = i0+3
            t0=(x(i3)-x(i1))*rac2
```

```
                x(i1)=x(i1)+x(i3)
                i7 = i6+1
                t4=-(x(i7)+x(i5))*rac2
                x(i3)=x(i7)-x(i5)
                t3=t0+t4
                x(i6)=t1+t3
                    x(i7)=t1-t3
                t4=t4-t0
                x(i4)=t2+t4
                x(i5)=t2-t4
 30         continue
C
            ib=ib+nb
            nb=nb+lnb+lnb
C
C -- Length 4 butterflies --
C
            do 60 i=ib+1,ib+nb
                i0=jx0(i)
                i1=i0+1
                i2=i1+1
                i3=i2+1
                t1=x(i0)+x(i2)
                t2=x(i0)-x(i2)
                t3=x(i1)+x(i1)
                t4=x(i3)+x(i3)
                x(i0)=t1+t3
                x(i1)=t1-t3
                x(i2)=t2-t4
                x(i3)=t2+t4
60          continue
C
            return
            end
```

```
C
C-------------------------------------------------------------------------------c
C       Calcule le produit de deux suites complexes x et h
C       de longueur n = 2**m, ayant la symétrie hermitienne.
C
C       Les données dans x et h doivent rangées comme en
C       sortie d'une fft réelle, c'est à dire
C       [ Re(0),Re(1),...,Re(n/2),Im(n/2-1),...,Im(1) ]
C
C       Le produit est retourné dans x, h est inchangé
C-------------------------------------------------------------------------------c
C
        subroutine mul2sh(x,h,m)
        dimension x(*),h(*)
C
        n=2**m
        x(1)=x(1)*h(1)
        x(n/2+1)=x(n/2+1)*h(n/2+1)
        do 10 i=2,n/2
            xt=x(i)
            x(i)=x(i)*h(i)-x(n+2-i)*h(n+2-i)
10          x(n+2-i)=xt*h(n+2-i)+x(n+2-i)*h(i)
        return
        end
C
C-----------------------------------------------------------------------c
C       Calcule le produit de deux suites complexes (x,y)
C       et (h1,h2) de longueur n = 2**m
C
C       Le produit est retourné dans (x,y)
C       (h1,h2) est inchangé
C-----------------------------------------------------------------------c
C       subroutine mul2z(x,y,h1,h2,m)
        dimension x(*),y(*),h1(*),h2(*)
C
        n=2**m
        do 10 i=1,n
            xt=x(i)
            x(i)=x(i)*h1(i)-y(i)*h2(i)
10          y(i)=xt*h2(i)+y(i)*h1(i)
        return
        end
```

V COPIE DES ARTICLES

[1] P. DUHAMEL "Implementation of "split-radix", FFT algorithms for complex, real, and real-symetric data" IEEE Trans on ASSP, Vol. 34, N° 2, pp. 285-295, April 1986

[10] P. DUHAMEL : "A connection between bit-reversal and matrix transposition : hardware and software consequences", IEEE Trans. on ASSP, Vol 38, N° 11, pp. 1893-1896, Nov 1990

[15] P. DUHAMEL, M. VETTERLI : "Improved Fourier and Hartley Transform Algorithms Application to Cyclic Convolution of Real Data", IEEE Trans on ASSP, vol 35, n° 6, June 1987

[16] P. DUHAMEL, B. PIRON, et J.M. ETCHETO : "On Computing the Inverse DFT", IEEE Trans on ASSP, Vol 36. N° 2, pp 285-286, Feb. 1988

[17] A. A. YONG : "A Better FFT Bit-Reversal Algorithm Without Tables", IEEE Trans on ASSP, Vol 39, N° 10, pp. 2365-2367, Oct. 1991

Implementation of "Split-Radix" FFT Algorithms for Complex, Real, and Real-Symmetric Data

PIERRE DUHAMEL

Abstract—**A new algorithm is presented for the fast computation of the discrete Fourier transform. This algorithm belongs to that class of recently proposed 2^n-FFT's which present the same arithmetic complexity (the lowest among any previously published one). Moreover, this algorithm has the advantage of being performed "in-place," by repetitive use of a "butterfly"-type structure, without any data reordering inside the algorithm. Furthermore, it can easily be applied to real and real-symmetric data with reduced arithmetic complexity by removing all redundancy in the algorithm.**

I. Introduction

SINCE the early paper by Cooley and Tukey [1], a lot of work has been done on the FFT algorithm, and this has resulted in classes of algorithms such as radix-2^m algorithms, Winograd algorithm (WFTA) [10], and prime factor algorithms (PFA) [8].

Among these, the radix-2 and radix-4 algorithms are the ones that have been mostly used for practical applications. This is due to their simple structure, with a constant geometry (butterfly type), and the possibility of performing them "in place," even if they are more costly in terms of number of multiplications than WFTA and PFA.

Recently, some real factors radix-2 algorithms have been proposed [2]–[4] which require fewer multiplications, while preserving more or less the advantages mentioned above. Unfortunately, these methods need 20–30 percent more additions, and seem to be numerically ill conditioned.

Even more recently [11], [13]–[15], some algorithms were proposed that need the same number of multiplications as the real factor algorithms, but reduce the number of additions, and should not be too sensitive to quantization noise.

Let us also point out that an early paper [12] already proposed an algorithm with the same computational complexity as the recent ones, but remained unreferenced until very recently.

This paper is concerned with the evaluation and implementation of one of these recent algorithms: the "split-radix" FFT algorithm [11], which seems to include the advantages of the recent methods, namely, the following:

- the lowest number of multiplications, together with real-factor algorithms;
- the lowest number of additions among the 2^m algorithms;
- the same regularity as radix-4 algorithms;
- the same flexibility as radix-2 algorithms (useful for all $N = 2^n$);
- no reordering of the data inside the algorithm;
- possibility of in-place implementation for real and real-symmetric data with reduced arithmetic complexity; and
- numerically as well conditioned as radix-4 algorithms.

After briefly recalling the split-radix algorithm, and evaluating the number of multiplications and additions involved in a complex DFT, we describe how to increase the regularity of the initial algorithm by means of a permutation of the outputs of the computational cell, which keeps the "in-place" property of the algorithm. Afterwards, we show how the split-radix algorithm can be adapted to real and real-symmetric data. Finally, we enlighten some consequences of the characteristics of the presented algorithms: possible improvements of other transforms (DCT, 2-D DFT through polynomial transforms), and compatibility of the different arithmetic complexities involved.

II. The Split-Radix Algorithm

This algorithm comes from a very simple observation:

> A radix-2 algorithm diagram can be transformed quite straightforwardly into a radix-4 algorithm diagram simply by changing the exponents of the twiddle factors. It is quite clear then that at each stage of the algorithm a radix-4 is better for the odd terms of the DFT and a radix-2 for the even terms of the DFT. So, one might guess that restricting this transformation locally to the lower part of the diagram might improve the algorithm. It turns out that this is indeed the case.

The "split-radix" algorithm is then based on the following decomposition:

$$X_k = \sum_{n=0}^{N-1} x_n W_N^{nk}$$

if

$$\left(W_N \triangleq \cos\frac{2\pi}{N} - j\sin\frac{2\pi}{N}\right)$$

Manuscript received February 28, 1985; revised September 11, 1985.
The author is with the CNET/PAB/RPE/ETP, 38-40, Rue du Général Leclerc, 92131 Issy-les-Moulineaux, France.
IEEE Log Number 8407032.

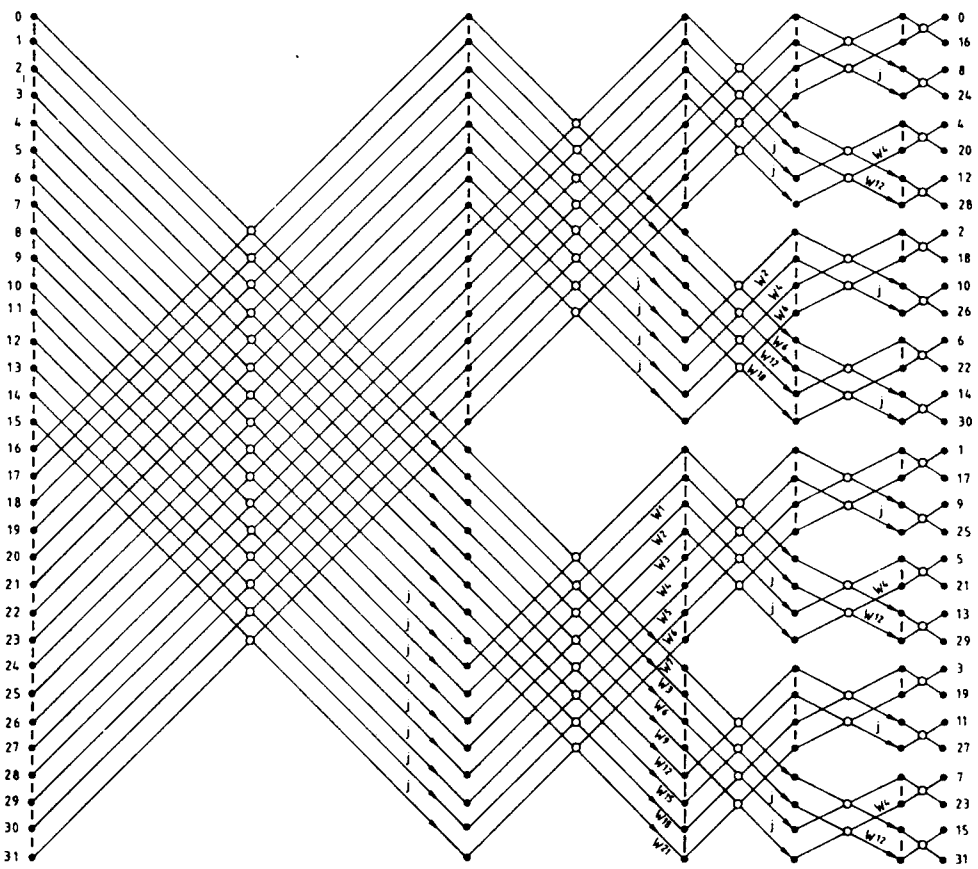


Fig. 1. Length 32 split-radix algorithm.

is the DFT to be computed, it is decomposed into

$$X_{2k} = \sum_{n=0}^{N/2-1} (x_n + x_{n+(N/2)})\, W_N^{2nk}$$

$$X_{4k+1} = \sum_{n=0}^{N/4-1} [(x_n - x_{n+(N/2)}) - j(x_{n+(N/4)} - x_{n+3(N/4)})]\, W_N^n W_N^{4nk} \quad (1)$$

$$X_{4k+3} = \sum_{n=0}^{N/4-1} [(x_n - x_{n+(N/2)}) + j(x_{n+(N/4)} - x_{n+(3N/4)})]\, W_N^{3n} W_N^{4nk}.$$

The first stage of a split-radix decimation-in-frequency decomposition then replaces a DFT of length N by one DFT of length $N/2$ and two DFT's of length $N/4$ at the cost of $(N/2 - 4)$ general complex multiplications (3 real mult + 3 adds), and 2 multiplications by the eighth root of unity (2 real mult + 2 adds).

The length-N DFT is then obtained by successive use of such decompositions, up to the last stage where some usual radix-2 butterflies (without twiddle factors) are needed (see Fig. 1 for a length 32-split-radix FFT).

III. "Split-Radix" on Complex Data

As the "split-radix" is a combination of radix-2 and radix-4 decomposition (at each step), one could expect that its arithmetic complexity will be somewhere in between the complexities of radix-2 and radix-4. In fact, the number of multiplications and additions involved is lower than the corresponding ones for both algorithms, as is shown in the next section.

A. Arithmetic Complexity

Let M_n^c (respectively, A_n^c) be the number of real multiplications (respectively, additions) needed to perform a 2^n-complex DFT with the split-radix algorithm. By (1), we get

$$M_n^c = M_{n-1}^c + 2M_{n-2}^c + 3 \cdot 2^{n-1} - 8$$

and, with the initial conditions $M_1 = 0$, $M_2 = 0$, we obtain

$$M_n^c = 2^n(n - 3) + 4. \quad (2)$$

Disregarding for a while the number of additions needed to perform the complex multiplications, the remaining ones can easily be evaluated as $n \cdot 2^{n+1}$, since, at each of the n stages, a new point is generated by a complex addition. Then, since the number of real additions needed to compute a complex product is equal to the number of multiplications, we have:

$$A_n^c = n \cdot 2^{n+1} + M_n^c$$

$$A_n^c = 3 \cdot 2^n(n - 1) + 4. \quad (3)$$

TABLE I
NUMBER OF NONTRIVIAL REAL MULTIPLICATIONS TO COMPUTE A LENGTH-N COMPLEX DFT

N	Radix 2	Radix 4	Radix 8	Real Factor (2,3,4)	WANG [14]	FFCT [13]	RCFA [15]	Split-Radix
16	**24**	**20**		**20**	**20**	**20**	**20**	**20**
32	**88**			**68**	**68**	**68**	**68**	**68**
64	264	208	204	196	204	196	196	196
128	712			516	564	516	516	516
256	1800	1392		1284	1468	1284	1284	1284
512	4360		3204	3076	3652	3076	3076	3076
1024	10 248	7856		7172	8780	7172	7172	7172
2048				16 388	20 564	16 388	16 388	16 388

TABLE II
NUMBER OF REAL ADDITIONS TO COMPUTE A LENGTH-N COMPLEX DFT

N	Radix 2	Radix 4	Radix 8	Rader-Brenner [2]	WANG [14]	FFCT [13]	RCFA [15]	Split-Radix
16	**152**	**148**		**148**	**148**	**148**	**148**	**148**
32	**408**			**424**	**388**	**388**	**388**	**388**
64	1032	976	972	1104	972	964	964	964
128	2504			2720	2356	2308	2308	2308
256	5896	5488		6464	5564	5380	5380	5380
512	13 566		12 420	14 976	12 868	12 292	12 292	12 292
1024	30 728	28 336		34 048	29 260	27 652	27 652	27 652
2048				76 288	65 620	61 444	61 444	61 444

These quantities are computed in Tables I and II, and compared to classical algorithms (radix-2,4,8, real factors), and more recent ones, by Wang [14], Vetterli and Nussbaumer [13], and Martens [15].

Observing Tables I and II shows that the split-radix algorithm has the lowest number of both multiplications and additions, together with FFCT [13] and RCFA [15]. (The number of operations given in [15] has been corrected in Tables I and II to take into account the fact a multiplication by $\sqrt[8]{1}$ needs only 2 real multiplications and 2 real additions.)

B. Connection with Other Algorithms

It is interesting to note that the "split-radix" FFT algorithm has strong connections with many of the published algorithms.

- *Relation with Classical Algorithms:* As far as "classical" algorithms (radix-2,4,8, and even variable radix, as proposed in [6]) are concerned, it is easy to see that all these algorithms, together with split-radix, have exactly the same number of complex multiplications (counting j, $\sqrt[8]{1}$, W^k altogether), and can be compared by using the same diagrams, based on radix-2 butterflies. Hence, the split-radix may be considered as belonging to the same class of algorithms. Among the advantages of the split-radix is that it has the smallest number of nontrivial multiplications (it maximizes the number of complex multiplications equal to j).

At this stage of the comparison between radix-2,4,8 and the split-radix algorithm, a question naturally arises: is it possible to improve the proposed algorithm by computing the odd terms of the DFT through a higher radix (8, for example)?

It can easily be checked out that a $\frac{2}{8}$-split-radix algorithm is worse than a $\frac{2}{4}$-split-radix algorithm from an arithmetic complexity point of view.

Several other decompositions were studied, but no one could improve the initial one. So, one can conjecture that working with separate radixes 2 and 4 for (locally) even and odd terms of the DFT has brought down the number of multiplications to a minimum for that class of algorithms.

It is also interesting to notice that real factor and split-radix algorithms need exactly the same number of multiplications, although being obtained by completely different approaches. The superiority of the split-radix algorithm comes from its lower number of additions, higher regularity, and better numerical accuracy.

Since Wang's algorithm [14] is seen from Tables I and II to give intermediate computational complexity between classical and other recent algorithms, we shall not consider it here.

In the rest of this section, we shall see that three of the recent algorithms, FFCT [13], RCFA [15], and the split-radix, are closely related. In fact, Martens' algorithm [15] is exactly a decimation-in-frequency split-radix, while some of the characteristics of the FFCT can be derived from a decimation-in-time split-radix designed for real data.

- *Connection with RCFA:* In fact, as explained in [13], RCFA is based on the equivalence between the DFT

$$X_k = \sum_{n=0}^{N-1} x_n W_N^{nk} \tag{6}$$

and the alternative polynomial expression

$$X_k = [X(z) \bmod (z^N - 1)] \bmod (z - W_N^k) \quad (7)$$

where

$$X(z) = \sum_{n=0}^{N-1} x_n z^n.$$

The DFT is then obtained by means of a factorization of $z^N - 1$ in cyclotomic polynomials in $Q[j]$, ($=Q$ extended by $j = -1$), which is not explicitly stated in [15]

$$z^{2^n} - 1 = (z^{2^{n-1}} - 1) \cdot (z^{2^{n-2}} + j) \cdot (z^{2^{n-2}} - j) \quad (8)$$

and the whole DFT becomes, as W_N^{2i} (respectively, W_N^{4i+1}, W_N^{4i+3}) are roots of $z^{2^{n-1}} - 1$ (respectively, $z^{2^{n-2}} - j$, $z^{2^{n-2}} + j$)

$$\begin{aligned} X_{2k} &= [X(z) \bmod (z^{N/2} - 1)] \bmod (z - W_N^{2k}) \\ X_{4k+1} &= [X(z) \bmod (z^{N/4} - j)] \bmod (z - W_N^{4k+1}) \\ X_{4k+3} &= [X(z) \bmod (z^{N/4} + j)] \bmod (z - W_N^{4k+3}) \end{aligned} \quad (9)$$

which is exactly the same expression as a split-radix decimation-in-frequency decomposition, as given in (1), when X_{4k+1} and X_{4k+3} are in turn evaluated through a length $N/4$ DFT at the cost of some multiplications by W_N^k and W_N^{3k} [11], [15].

Although these two algorithms are now recognized to be identical, the presentation given here has the advantage that it provides a better understanding of what should be the elementary computational cell, and will lead to improvements of the algorithms both in terms of regularity of the resulting program, and of the number of memory calls. This point will be treated in Section III-C. It also allows us to study DIT and DIF versions of the algorithm more easily (especially in the case of real data). This point will be treated in a subsequent paper.

• *Connection with FFCT:* As the FFCT is primarily a discrete cosine transform algorithm (DCT), we shall first give the expression of a DCT as part of a Fourier transform. Then by applying a decimation-in-time split-radix algorithm, this will naturally introduce the mapping used in FFCT, as first proposed in [20].

Let

$$\text{DCT}(k, N, x) = \sum_{n=0}^{N-1} x_n \cos \frac{2\pi(2n+1)k}{4N} \qquad k = 0, \cdots N-1 \quad (10)$$

be the DCT to be computed (notations are taken from [13]).

It is fairly easy to see that DCT (k, N, x) can be obtained from a DFT of length $4N$ on real data

$$X'_k = \sum_{n=0}^{4N-1} x'_n W_{4N}^{nk} \quad (11)$$

by taking

$$\begin{cases} x'_{2n+1} = x_n & n = 0, \cdots N-1 \\ x'_i = 0 & \text{otherwise.} \end{cases} \quad (12)$$

In fact, in that case,

$$X'_k = \sum_{n=0}^{N-1} x_n \cos \frac{2\pi(2n+1)k}{4N} + j \sum_{n=0}^{N-1} x_n \sin \frac{2\pi(2n+1)k}{4N} \quad (13)$$

and

$$\text{DCT}(k, N, x) = \text{Re}\{X'_k\} \qquad k = 0, \cdots N-1.$$

Let us now apply a decimation-in-time split-radix decomposition to the DFT

$$X'_k = \sum_{n=0}^{2N-1} x'_{2n} W_{4N}^{2nk} + \sum_{n=0}^{N-1} x'_{4n+1} W_{4N}^{(4n+1)k} + \sum_{n=0}^{N-1} x'_{4n+3} W_{4N}^{(4n+3)k} \quad (14)$$

and, if we replace x'_n by x_n, as in (10) above, we get

$$X'_k = \sum_{n=0}^{N/2-1} x_{2n} W_{4N}^{(4n+1)k} + \sum_{n=0}^{N/2-1} x_{2n+1} W_{4N}^{(4n+3)k}. \quad (15)$$

Both terms in (15) are each one-half of the two expressions which are usually computed in a "split-radix" algorithm (14). Furthermore, since these two expressions were shown in [19] to be of the same type in the case of a decimation-in-frequency algorithm, it is natural to try to combine them into a single term: this can be done by using a change of variable in the second term of (15)

$$n = N - i - 1 \quad (16)$$

which yields

$$X'_k = \sum_{n=0}^{N/2-1} x_{2n} W_{4N}^{(4n+1)k} + \sum_{i=N/2}^{N-1} x_{2N-2i-1} W_{4N}^{-(4i+1)k} \quad (17)$$

and, since x_n is real,

$$\begin{aligned} \text{DCT}(k, N, x) &= \text{Re}\{X'_k\} \\ &= \text{Re} \sum_{n=0}^{N/2-1} x_{2k} W_{4N}^{(4n+1)k} + \sum_{i=N/2}^{N-1} x_{2N-2i-1} W_{4N}^{(4i+1)k} \\ &= \text{Re}\{U_k\} \end{aligned} \quad (18)$$

with

$$U_k = \sum_{n=0}^{N-1} u_n W_{4N}^{(4n+1)k}$$

and

$$\begin{aligned} u_n &= x_{2n} & n &= 0, \cdots N/2-1 \\ u_n &= x_{2N-2n-1} & n &= N/2, \cdots N-1 \end{aligned} \quad (19)$$

which is the mapping defined by Vetterli and Nussbaumer.

Let us now consider more precisely the relation between U_k and DCT (k, N, x):

$$\begin{aligned} U_k &= \sum_{n=0}^{N-1} u_n \cos \frac{2\pi(4n+1)k}{4N} \\ &\quad + j \sum_{n=0}^{N-1} u_n \sin \frac{2\pi(4n+1)k}{4N} \\ &= \sum_{n=0}^{N-1} u_n \cos \frac{2\pi(4n+1)k}{4N} \\ &\quad + j \sum_{n=0}^{N-1} u_n \cos \frac{(2\pi(4n+1)N}{4N} - \frac{2\pi(4n+1)k}{4N} \\ &= \mathrm{DCT}\,(k, N, x) + j\,\mathrm{DCT}\,(N-k, N, x) \end{aligned} \tag{21}$$

and, since an alternative expression for U_k is:

$$U_k = W_{4N}^k \sum_{n=0}^{N-1} u_n W_{4N}^{4nk}. \tag{22}$$

We have shown that a length-N DCT can be obtained through a real DFT of length N, plus $N/2 - 1$ complex multiplications, as first established in [13] for the FFCT.

The original FFCT algorithm is based on further decomposition of the real DFT into smaller DCT's, and so on, with the aid of the mapping (20) at each step.

One should notice that a more regular DCT algorithm could be obtained by one step of FFCT, followed by a real split-radix algorithm, which gives exactly the same operations count, while avoiding other data reordering.

C. Implementation in the Case of Complex Data

From (1), it is quite obvious that the split-radix algorithm can be performed, in place, by repetitive use of the "butterfly"-type structure given in Fig. 2.

Furthermore, the minimum number of arithmetic operations is obtained with only 4 types of butterflies: the general one (6 real mult), as given in Fig. 2; plus two special cases: $k = 0$ (without multiplications), and $k = N/8$ (4 real multiplications). The 4th one is a usual radix-2 butterfly (without multiplication) to compute length-2 DFT's.

The implementation of the algorithm described in Fig. 1 is based on the fact that, at each stage $n^o i$ of the algorithm, the butterflies are always applied in a repetitive manner to blocks of length $N/2^i$. In a Fortran implementation that can be obtained upon request, we have chosen to precompute the first address of each block to which butterflies are applied. Furthermore, the number of memory calls (loads and stores) is minimized by including the radix-2 butterflies into the last two stages. The penultimate stage transforms the 8-points DFT's into a 4-point DFT to be computed in the last stage, plus an odd DFT in which the other points are computed. The last stage is composed of full 4-point DFT's.

Without counting initializations in both cases (computation of the multiplicative constants and of the address of the blocks), the run times were 92.2 ms for a 1024-point FFT, versus 100.3 ms for the same FFT computed with the compact radix-4 program by Morris [21] (time averages over 200 runs on a Honeywell-Bull DPS 8), using an optimizing compiler under MULTICS system).

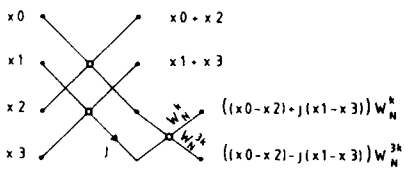


Fig. 2. The "butterfly" used in the split-radix algorithm.

TABLE III
NUMBER OF FLOATING-POINT OPERATIONS IN A 1024-POINT FFT. (ESTIMATIONS BASED ON THE ASSEMBLER GENERATED BY THE FORTRAN COMPILER ON A HONEYWELL-BULL dps 8 UNDER MULTICS OPERATING SYSTEM)

	FFT 4	Split-Radix
fld	22 884	21 280
fstr	28 506	27 652
fad	19 120	18 749
fsab	9 386	9 073
fmp	8 026	7 342

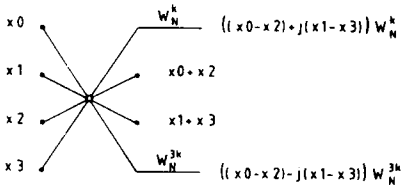


Fig. 3. Butterfly with permuted outputs for a more regular "split-radix" algorithm.

Furthermore, careful inspection of the program shows that the number of each floating-point operation (load, store, addition, substration, multiplication) is lower in split-radix than in radix-4 (Table III). This fact tends to indicate that this program should run faster than radix-4 on any machine (for comparable implementations, since an autogen technique [9] should improve both algorithms). Tests were made on several machines, always giving a 10–20 percent improvement over a comparable radix-4 implementation.

Comparison of the overheads also turns out in favor of split-radix, since for a 1024-point FFT, radix-4 needs the computation of 2295 multiplicative constants (through sine and cosine evaluations) and split-radix 1536 multiplicative constants, plus 341 integer addresses.

For some specific applications (hardware implementation, necessity of reducing the overhead, or the memory occupation) it may be useful to increase the regularity of the algorithm: this can be done by the use of a butterfly with permuted outputs, as shown in Fig. 3.

In that case, the length-32 split-radix diagram becomes as shown in Fig. 4. The price that has to be paid for this

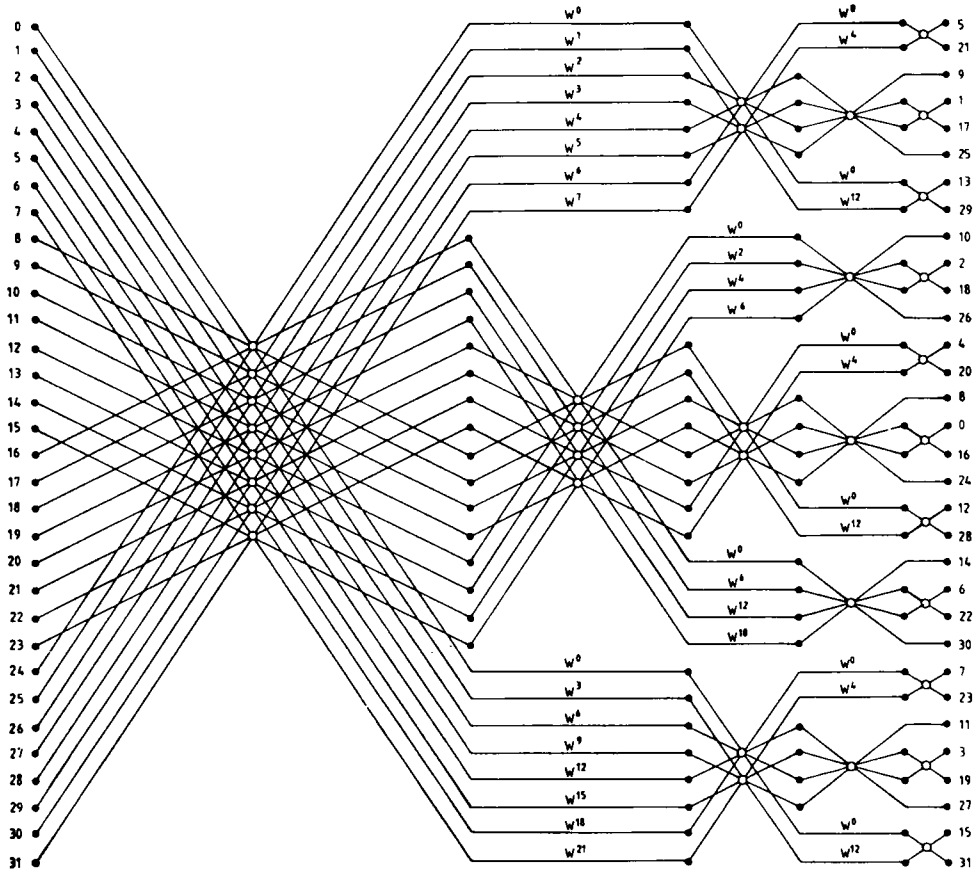


Fig. 4. Symmetric "split-radix" diagram.

increase in the regularity of the program is twofold. First, the outputs are no longer in a bit-reversed order, and in-place reordering is not possible. Second, this permutation induces more floating-point loads and stores.

In fact, it turns out that, for a software implementation, the resulting program is notably slower (96.7 ms for a 1K DFT). Nevertheless, the regularity of the resulting diagram may be attractive for hardware implementation.

IV. "Split-Radix" Algorithm on Real Data

Redundancy reduction is very easy when a split-radix FFT algorithm is applied to real signals. In fact, when the sequence $\{x_n\}$ is real, X_k and X_{N-k} are complex conjugates. When reported in (1), this means that it is useless to compute both $\{X_{4k+1}\}$ and $\{X_{4k+3}\}$ since

$$X_{4k+3} = X_{N-(4k'+1)} = X^*_{4k'+1}.$$

This simple remark allows us to compute the arithmetic complexity of split-radix algorithm applied to real signals.

A. Arithmetic Complexity

In the case of real signals, (1) now changes a real DFT of length 2^n into a real DFT of length 2^{n-1} (to get X_{2k}), plus a complex DFT of length 2^{n-2} (to get X_{4k+1}), at the cost of $3(2^{n-2} - 2) + 2$ real multiplications and $3(2^{n-2} - 2) + 2$ real additions due to the twiddle factors, plus 2^n additions.

Thus, we can get the multiplicative complexity M^r_n and additive complexity A^r_n of a 2^n real DFT:

$$M^r_n = M^r_{n-1} + M^c_{n-2} + 3(2^{n-2} - 2) + 2$$

and, with the initial condition $M^c_1 = 0$, $M^r_2 = 0$, we obtain

$$M^r_n = 2^{n-1}(n - 3) + 2 = M^c_n/2. \tag{23}$$

The number of additions can be obtained as

$$A^r_n = A^r_{n-1} + A^c_{n-2} + 2^n + 3.2^{n-2} - 4$$
$$= A^r_{n-1} + 2^{n-2}(3n - 2)$$

and with

$$A^r_1 = 2$$

this yields

$$A^r_n = (3n - 5)2^{n-1} + 4 = A^c_n/2 - 2^n + 2. \tag{24}$$

A remark that will be useful later is that the multiplicative complexity is exactly one-half that of the complex case, while the number of additions is less than one-half of the corresponding one in the complex case.

Comparison to other published algorithms is made in Table IV, where the operation count of [17] has been changed to take into account the 3 mults 3 adds algorithms for complex multiplication. Here again, the split-radix algorithm is seen to have the same arithmetic complexity as

TABLE IV
ARITHMETIC COMPLEXITY FOR THE COMPUTATION OF DFT'S ON REAL DATA

	Multiplications			Additions		
N	Bergland [17]	FFCT [13]	Split-Radix	Bergland [17]	FFCT [13]	Split-Radix
16	12	10	10	60	60	60
32	44	34	34	172	164	164
64	132	98	98	572	420	420
128	356	258	258	1124	1028	1028
256	900	642	642	2692	2436	2436
512	2180	1538	1538	6276	5636	5636
1024	5124	3586	3586	14 340	12 804	12 804
2048	11 780	8194	8194	32 260	28 676	28 676

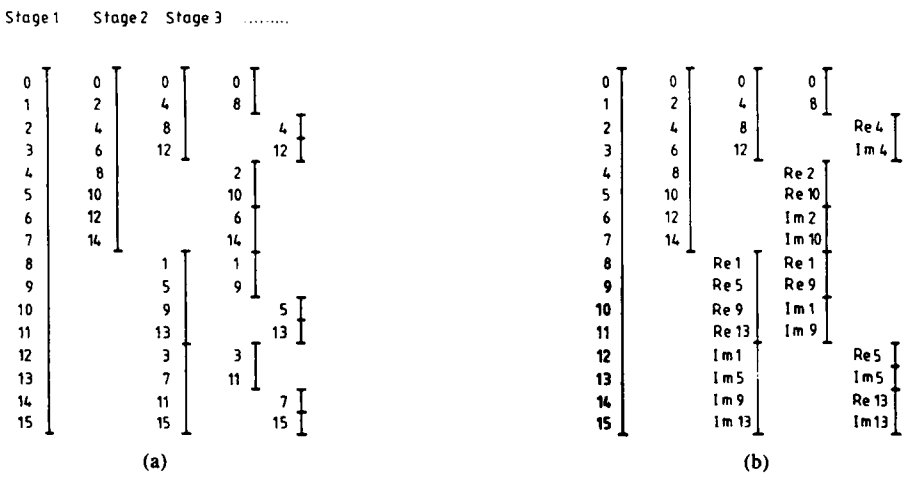


Fig. 5. Schematic representation of the progression of a 16-point split-radix algorithm: (a) on complex data and (b) on real data.

FFCT, and significantly less operations than the algorithm by Bergland [17].

B. Implementation

Since it is intended to perform the split-radix algorithm "in place" on real data, we need a schematic representation of the way the algorithm works in the different stages and of the place where the intermediate results are stored in.

To make this representation clearer, let us begin with the split-radix on complex data, as shown in Fig. 5(a) for a 16-point FFT. This diagram can be explained as follows.

On stage 0, the butterflies are applied to one block of length N, and this block is useful in the computation of every transformed sample X_k, $k = 0, \cdots, 15$. After this first step, we have now, as given in (1), one block of length $N/2$ belonging to stage 1 from which the even points X_{2k}, $k = 0, \cdots, 7$ are computed, and two blocks of length $N/4$, hence belonging to stage 2, used in the computation of X_{4k+1}, $k = 0, \cdots, 3$, respectively. This process is then applied recursively to all blocks of length $N/2^i$, appearing at stage i.

When the split-radix algorithm is applied to real data, the block of state 0 is real, inducing a block of length $N/2$ at stage 1, which is also real. The other two blocks, of length $N/4$, at stage 2, are complex but, as noticed in Section IV-A, $\{X_{4k+3}\}$ needs not be computed, so that the corresponding locations in the diagram can be used to store the imaginary part of the block used to compute $\{X_{4k+1}\}$. This process is repeated until the whole transform is obtained.

One should also notice that, since it is necessary to distinguish between butterflies on real data (the first block of each stage) and on complex data, the number of different butterflies needed to fulfill the minimum number of multiplications is now 8 (but many of them are seldom used, which makes suboptimal implementations with reasonably good performances easy to find).

Increasing the regularity of the split-radix algorithm on real data, through a permutation of the outputs of the butterflies on complex data, is also of interest, and is given on Fig. 6.

V. "SPLIT-RADIX" ALGORITHM ON REAL-SYMMETRIC DATA

FFT's of real-symmetric signals are also of interest, for example, when computing the energy spectrum through autocorrelation, or even when computing autocorrelations [22].

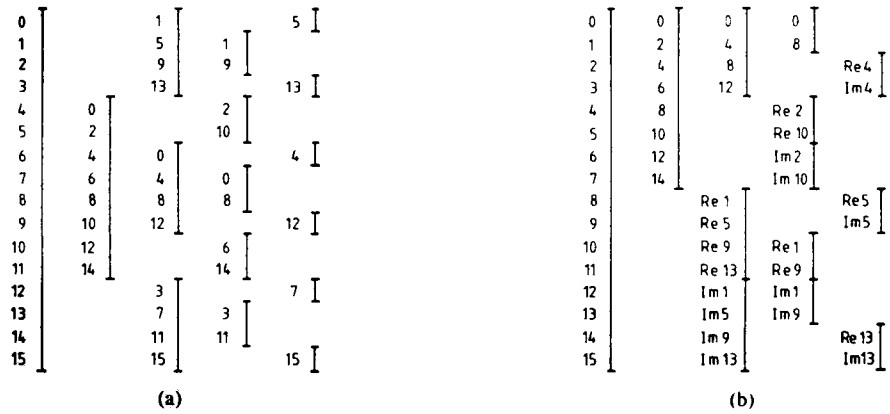


Fig. 6. Schematic representation of a 16-point split-radix algorithm with increased regularity: (a) on complex data and (b) on real data.

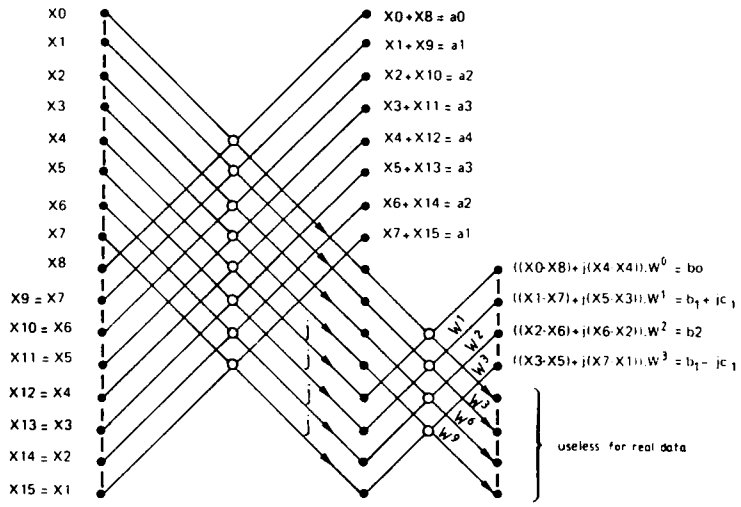


Fig. 7. Symmetries involved in one step of split-radix decomposition of a real-symmetric signal (note that if $W_N^{N/4-1} = c + js$, $W_N^1 = s + jc$).

Scrutinizing the split-radix algorithm on real data in the case of real-symmetric entries, as given in Fig. 7, shows that the first step of the algorithm transforms a real-symmetric sequence of length N into one real-symmetric sequence of length $N/2$, ($a_i = a_{N/2-i}$) plus one complex sequence of length $N/4$, with complex conjugate symmetry ($b_i = b^*_{N/4-i}$) which, in turn, have to be transformed. We shall see in Section V-B that this transformation keeps the same number of independent variables, allowing an "in-place" implementation.

A first remark that will allow us to compute the arithmetic complexity of this algorithm is that the DFT of a signal with complex conjugate symmetry can be obtained exactly by reversing the graph of the split-radix algorithm on real data, and conjugating every twiddle factor. Hence, the arithmetic complexity of this part of the algorithm is exactly the same as an FFT on real data.

We shall only use this remark to obtain the arithmetic complexity of the algorithm, but the final algorithm described in Section V-B will be obtained only by redundancy reduction.

A. Arithmetic Complexity

This decomposition, described above, of a length-N real-symmetric sequence into a length-$N/2$ real-symmetric sequence plus a length-$N/4$ Hermitian one is seen from (1) to be obtained at the cost of $3(2^{n-3} - 1) + 1$ multiplications, and $2^{n+1} + 3(2^{n-3} - 1)$ additions. This yields

$$M_n^{rs} = M_{n-1}^{rs} + M_{n-2}^{r} + 3(2^{n-3} - 1) + 1$$

$$M_n^{rs} = 2^{n-2}(n - 3) + 1 = M_n^r/2 \tag{25}$$

and

$$A_n^{rs} = A_{n-1}^{rs} + A_{n-2}^{r} + (3n - 4)\,2^{n-3} + 1$$

$$A_n^{rs} = 2^{n-2}(3n - 7) + n + 3.$$

It should be noticed, in this operation count, that multiplications by 2 were counted neither as a multiplication nor as an addition. This can be justified in the case of fixed-point implementation, since in that case, overflow prevention involves scaling by 2 or 4 on the outputs of

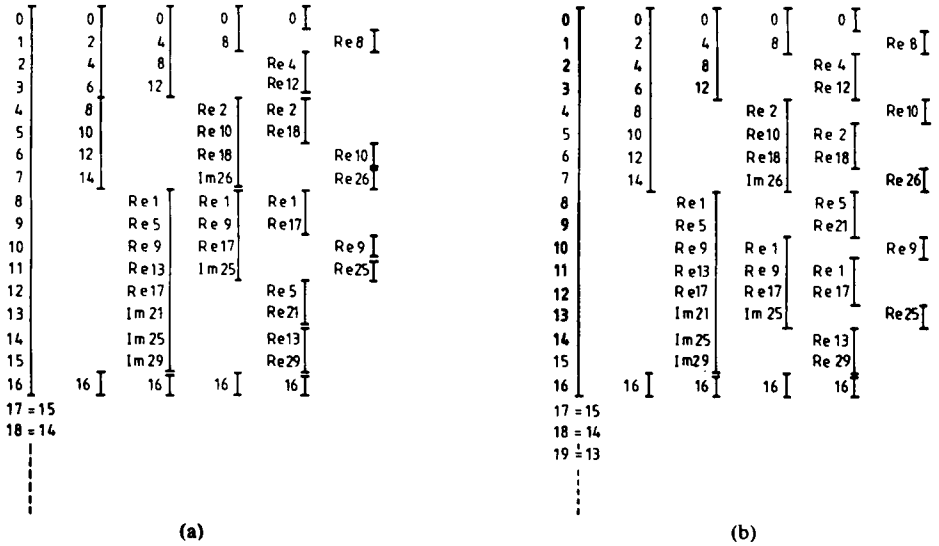


Fig. 8. Schematic representation of the progression of a 32-point split-radix algorithm on real-symmetric data: (a) with outputs in bit reversed order and (b) with permuted outputs for a more regular implementation.

the butterflies, which can be combined with the multiplications by 2 due to the algorithm.

Precise comparison was difficult to make with Ziegler's algorithm [18], since he used the 4-mults, 2-adds complex multiplication algorithm. Nevertheless, split-radix should need less multiplications and less additions, since both redundancy reduction methods should roughly cut by half the corresponding numbers in Bergland's algorithm and real split-radix. Here again, the arithmetic complexity given above can be seen to be equal to the one of an algorithm based on FFCT for real-symmetric FFT computation [22].

B. Implementation

Due to the symmetries of the sequences involved in a stage of split-radix algorithm on real-symmetric sequence (Fig. 7), the $N/2 + 1$ memory locations needed to store the nonredundant initial data can also be used to store the transformed sequences: $N/4 + 1$ locations for the real-symmetric sequence of length $N/2$, plus $N/4$ for the complex one with conjugate symmetry. This is shown in the first stage of Fig. 8(a), where the $(N/2 + 1)$th location is always used for storage of the point of symmetry of the real sequence. This set of operations can be applied with a minimum number of arithmetic operations by means of 3 different "butterflies" characterized by their twiddle factors (w^o, $w_N^{N/8}$, others).

Furthermore, a straightforward application of (1) to a length-$N/4$ complex sequence with conjugate symmetry shows that the initial data are transformed into three sequences of respective length $N/8$, $N/16$, and $N/16$, each of them also having a complex-conjugate symmetry. This has two consequences: the number of twiddle factors can be cut by half, due to the symmetry, and in-place implementation is possible [see the lower part of stage 2 in Fig. 8(a)]. Since this part of the computation needs 4 different butterflies for a minimum number of operations, the total is now 7 for a full transform on real-symmetric data.

Fig. 8(b) gives the progression of the same algorithm with permuted outputs for a more regular implementation.

This algorithm of Fig. 8(b) has been implemented on a TMS 320, giving a run time of 1.6 ms for a 256-point transform (memory occupation: 744 words) and 245 μs for a 64-point transform (memory occupation: 406 words). The algorithm of Fig. 8(a) has been estimated to be about 10 percent faster for the 256-point transform.

VI. Consequences

A number of different algorithms can be revisited and improved by the use of split-radix decompositions.

As already noted in Section III-B, a new DCT algorithm with increased regularity can be obtained by one mapping of FFCT plus a real split-radix, which gives exactly the same number of multiplications and additions as FFCT while avoiding some data reordering.

Furthermore, odd DFT's can be obtained very easily with a split-radix algorithm by removing from its diagram the length-$N/2$ DFT which is used to compute the even points of the transform. The resulting operation counts are:

$$M_{n-1}^o = M_n^c - M_{n-1}^c = 2^n(n-1)$$

$$A_{n-1}^o = A_n^c - A_{n-1}^c - 2^n = 2^{n-1}(3n-2). \quad (26)$$

From Table V, it is seen that this algorithm requires less additions than the Rader–Brenner odd DFT algorithms [it should be pointed out that [23] gives already the same operation count as (26)]. This fact allows 2-D DFT's by polynomial transforms to be derived with reduced number of additions.

More important conseqences can be established from the consideration of the arithmetic complexities of the dif-

TABLE V
NUMBER OF NONTRIVIAL REAL OPERATIONS TO COMPUTE A LENGTH-N COMPLEX ODD DFT

	Multiplications		Additions	
N	Rader-Brenner [2]	Split-Radix	Rader-Brenner [2]	Split-Radix
8	16	16	64	
16	48	48	212	208
32	128	128	552	512
64	320	320	1360	1216
128	768	768	3232	2816
256	1792	1792	7488	6400
512	4096	4096	17 024	14 336
1024	9216	9216	38 144	31 744

ferent algorithms involved. In fact, it appears that all the considered algorithms (DFT's on complex, real, real-symmetric data, odd DFT's, DCT) are now of compatible complexity, i.e., an improvement on any of these algorithms allows us to derive better ones for the others, since they can be expressed mutually as follows.

a) Complex DFT 2^n — 2 real DFT 2^n + $2^{n+1} - 4$ additions
but also
real DFT 2^n — real DFT 2^{n-1} + complex DFT 2^{n-2} + $(3 \cdot 2^{n-2} - 4)$ multiplications + $(2^n + 3 \cdot 2^{n-2} - n)$ additions.

b) Real DFT 2^n — 1 real symmetric DFT 2^n + 1 real antisymmetric DFT 2^n + $(2^n - 2)$ additions
but also
real symmetric DFT 2^n — 1 real symmetric DFT 2^{n-1} + 1 inverse real DFT + $3(2^{n-3} - 1) + 1$ multiplications + $(3n - 4)\,2^{n-3} + 1$ additions.

c) Real DFT 2^n — 1 real DFT 2^{n-1} + 2 DCT 2^{n-2} + $(3 \cdot 2^{n-1} - 2)$ additions
but also
DCT 2^n — 1 real DFT 2^n + $(3 \cdot 2^{n-1} - 2)$ multiplications + $(3 \cdot 2^{n-1} - 3)$ additions.

d) Complex DFT 2^n — 1 complex DFT 2^{n-1} + 1 odd DFT 2^{n-1} + 2^n additions
but also
odd DFT 2^{n-1} — 2 complex DFT's 2^{n-2} + $2(3 \cdot 2^{n-2} - n)$ multiplications + $(2^n + 3 \cdot 2^{n-1} - 8)$ additions.

Furthermore, a comparison has been made in [19] between the number of true complex multiplications (i.e., $\neq +j = -W_N^{N/4}$) in a complex split-radix algorithm, and in a method based on a decomposition of the DFT into a number of polynomial products being themselves computed with optimum algorithms (cf. Table VI).

TABLE VI
NUMBER OF NONTRIVIAL COMPLEX MULTIPLICATIONS INVOLVED IN THE 2^n-FFT ALGORITHMS

N	Split-Radix	Polynomial Product [19]
8	2	2
16	8	8
32	26	26
64	72	72
128	186	178
256	456	408
512	1082	890
1024	2504	1880

These results tend to indicate that all these algorithms described above could be optimum (as far as the number of true complex multiplications is concerned) up to length 64, and optimum in a property defined subclass for longer transforms, since the divergence between both multiplicative complexities, as given in Table VI begins to appear exactly when the degrees of the polynomial products involved are too high for the optimum polynomial products algorithms to be of practical interest.

It should also be pointed out that, when considering nontrivial real multiplications, the split-radix algorithm is optimum up to and including $N = 16$ [24]. Nevertheless, the optimality on complex multiplications seems to give a better compromise to the number of addition $\leftrightarrow$ number of real multiplications for $16 < N \leq 64$.

VII. CONCLUSION

New algorithms were proposed for FFT computation on complex, real, and real-symmetric data, with possible application to DCT, odd DFT's, and polynomial transforms.

Although obtained only by redundancy reduction in the complex split-radix algorithms, they were shown to have the same arithmetic complexity as the best algorithms recently proposed, while having the advantage of a more regular implementation.

Furthermore, there are strong indications that they could be optimum (as far as the number of true complex multiplications is concerned) in a properly defined subclass of algorithms (e.g., algorithms using roots of unity as multiplying constants).

Acknowledgment

The author would like to thank H. Hollmann and M. Vetterli for stimulating discussion, C. S. Burrus for bringing [12] to his attention, M. Bellanger for stating the problem of real-symmetric FFT's, S. Hethuin for implementing it on TMS 320, and L. R. Morris for useful advice on program optimization.

References

[1] J. W. Cooley and J. W. Tukey, "An algorithm for machine computation of complex Fourier series," *Math Comput.*, vol. 19, pp. 297–301, 1965.

[2] C. M. Rader and N. M. Brenner, "A new principle for fast Fourier transformation," *IEEE Trans. Acoust., Speech, Signal Processing*, vol. ASSP-24, pp. 264–265, 1976.

[3] K. M. Cho and G. C. Temes, "Real-factor FFT algorithms," in *Proc. IEEE ICASSP*, 1978, pp. 634–637.

[4] R. D. Preuss, "Very fast computation of the radix-2, discrete Fourier transform," *IEEE Trans. Acoust., Speech, Signal Processing*, vol. ASSP-30, pp. 595–607, 1982.

[5] H. J. Nussbaumer, *Fast Fourier Transform and Convolution Algorithm*. Berlin, Germany: Springer-Verlag, 1981.

[6] H. Johnson and C. S. Burrus, "Twiddle factors in the radix-2 FFT," in *Proc. 1982 Asilomar Conf. Circuits Syst., Comput.*, 1982, pp. 413–416.

[7] S. Winograd, "On the multiplicative complexity of the discrete Fourier transform," *Adv. Math.*, vol. 32, pp. 83–117, 1979.

[8] C. S. Burrus and P. W. Eschenbacher, "An in-place, in-order prime factor FFT algorithm," *IEEE Trans. Acoust., Speech, Signal Processing*, vol. ASSP-29, pp. 806–817, 1981.

[9] L. R. Morris, "Automatic generation of time efficient digital signal processing software," *IEEE Trans. Acoust., Speech, Signal Processing*, vol. ASSP-25, pp. 74–79, Feb. 1977.

[10] S. Winograd, "On computing the discrete Fourier transform," *Math. Comput.*, vol. 32, pp. 175–199, Jan. 1978.

[11] P. Duhamel and H. Hollmann, "Split-radix FFT algorithm," *Electron. Lett.*, vol. 20, pp. 14–16, Jan. 1984.

[12] R. Yavne, "An economical method for calculating the discrete Fourier transform," in *AFIPS Proc.*, vol. 33, *Fall Joint Comput. Conf.*, Washington, DC, 1968, pp. 115–125.

[13] M. Vetterli and H. J. Nussbaumer, "Simple FFT and DCT algorithms with reduced number of operations," *Signal Processing*, vol. 6, pp. 267–278, July 1984.

[14] Z. Wang, "Fast algorithms for the discrete *W* transforms and the discrete Fourier transform," *IEEE Trans. Acoust., Speech, Signal Processing*, vol. ASSP-32, pp. 803–816, Aug. 1984.

[15] J. B. Martens, "Recursive cyclotomic factorization—A new algorithm for calculating the discrete Fourier transform," *IEEE Trans. Acoust., Speech, Signal Processing*, vol. ASSP-32, pp. 750–761, Apr. 1984.

[16] ——, "Discrete Fourier transform algorithms for real valued sequences," *IEEE Trans. Acoust., Speech, Signal Processing*, vol. ASSP-32, pp. 390–396, Apr. 1984.

[17] G. D. Bergland, "A fast Fourier transform algorithm for real-valued series," *Commun. ACM*, vol. 11, pp. 703–710, Oct. 1968.

[18] H. Ziegler, "A fast Fourier transform algorithm for symmetric real-valued series," *IEEE Trans. Audio Electroacoust.*, vol. AE-20, pp. 353–356, Dec. 1972.

[19] P. Duhamel and H. Hollmann, "Existence of a 2^n-FFT algorithm with a number of multiplications lower than 2^{n+1}," *Electron. Lett.*, vol. 20, pp. 690–692, Aug. 1984.

[20] M. J. Narasimha and A. M. Peterson, "On the computation of the discrete cosine transform," *IEEE Trans. Commun.*, vol. COM-26, pp. 934–936, June 1978.

[21] L. R. Morris, "Digital signal processing software," Ottawa, Canada, DSPSW, Inc., 1982, 1983.

[22] M. Vetterli, "FFT's of signals with symmetries and application to autocorrelation computation," submitted to MELECON 85, Madrid, Spain.

[23] ——, "Fast 2-D discrete cosine transform," accepted for publication, ICASSP'85, Tampa, FL, Mar. 1985.

[24] M. T. Heideman and C. S. Burrus, "Multiply/add tradeoff in length-2^n FFT algorithms," *ICASSP'85*, pp. 780–783, 1985.

Pierre Duhamel was born in Saint Maulvis, France, in 1953. He received the Ingenieur degree in electrical engineering from the National Institute for Applied Sciences, Rennes, France, in 1975, and the Dr. Ing. degree from Orsay University, Orsay, France, in 1978.

From 1975 to 1980 he was with Thomson-CSF, Paris, France, where his research interests were in circuit theory and signal processing, including digital filtering and automatic analog fault diagnosis. In 1980 he joined the National Research Center in Telecommunications (CNET), Issy-les-Moulineaux, France, where his activities were first concerned with the design of recursive CCD filters. He is now working on fast convolution algorithms, including number theoretic transforms and fast Fourier transforms.

A Connection Between Bit Reversal and Matrix Transposition: Hardware and Software Consequences

PIERRE DUHAMEL, SENIOR MEMBER, IEEE

Abstract—**First, a simple connection between bit reversal and matrix transposition is explained in the context of 2-D fast Fourier transform.**

This connection is then shown to be useful for hardware and software implementations of 2-D FFT's based on the row-column algorithm: the bit-reverse operations involved in the row and column 1-D transforms of length M can merge with the matrix transposition in between, to form a single bit reversal of length $N = M^2$, which is more efficient than considering these operations separately.

Furthermore, this connection is also the base of new bit-reverse algorithms that are explained in the second part of the paper. A first one can be implemented in two different versions, depending on the availability of an auxiliary memory of size $\sqrt{N}$. One of these versions turns out to be as fast as a recently proposed algorithm by Evans, while the second one is twice as fast as the usual one. The second algorithm is a divide and conquer approach to the problem that allows easy parallelization.

I. Introduction

In recent years, a lot of interest has emerged in the search for more efficient fast Fourier transform algorithms possessing both low arithmetic complexity and regular structure [1]. In some sense, this search was successful, since these new efficient algorithms, such as the split-radix algorithm, achieved low computing time, while remaining fairly compact. Nevertheless, the improvement remains limited (10–20% on most computers, compared to the autogen-radix-4 program by Morris [2] that was thought to be the fastest implementation at that time). Furthermore, there are some indications that we have now arrived at a practical optimum, at least on the number of operations [3] (multiplications and additions).

This work was performed mainly on the algorithm itself, and very little attention was paid to the permutation that is needed by most FFT programs: the bit reversal.

In fact, the more the algorithms are improved, the more significant is the percentage of time spent in bit reversing the data: in a length 1024 split-radix FFT, the bit reversal requires about 20% of the time needed to perform the FFT itself.

In this paper, we first point out a connection between bit reversal and matrix transposition, in the context of 2-D row-column FFT's. This connection is explained through very simple considerations. The main result here is to show that the two bit reversals involved in the row and column 1-D FFT of length M can merge with the matrix transpose in between to form a single bit reversal of length M^2, which is more efficient than considering these operations separately.

Then, the use of this consideration is shown to reduce by a factor of two the number of computations required for performing a bit reversal. This is obtained by direct programming of the above principle.

In the special case where the data are stored in mass memory, we point out that this provides a very simple algorithm similar to the fast matrix transposition of Eklundh [6], performing both bit reversals (rows plus columns) together with the matrix transposition.

Manuscript received October 8, 1988; revised July 20, 1989.

The author is with CNET/PAB/RPE, 92131 Issy-les-Moulineaux, France.

IEEE Log Number 9038431.

II. A Connection Between Bit Reversal and Matrix Transposition

Let us first consider the well-known implementation of 2-D FFT of length $2''' \cdot 2''' = M \cdot M$, when implemented by the row-column algorithm.

Since the 2-D FFT is separable in both dimensions, the DFT of a 2-D sequence can be obtained by computing the 1-D DFT of each column of the sequence, followed by the 1-D DFT of all the rows of the result.

Usually, since in most computers matrices are stored columnwise, one prefers to perform a matrix transposition after the DFT's on the rows. Row DFT's are then performed once again.

The advantage of this method is that one needs only to access the data by blocks that are stored in the same neighborhood. And this is advantageous not only when data are stored on disk, but also on any machine possessing virtual memory since it causes less "page faults."

The only disadvantage is that the result of the 2-D row-column FFT is the transpose of the desired 2-D DFT.

Let us now consider this row-column algorithm, and assume that the row 1-D transforms are FFT's with bit-reversed outputs, followed by a 1-D bit reversal of length M (i.e., the 1-D FFT is a straightforward decimation-in-frequency algorithm, or a reorganization of a decimation-in-time one), and that, after the matrix transposition, we

0096-3518/90/1100-1893$01.00 © 1990 IEEE

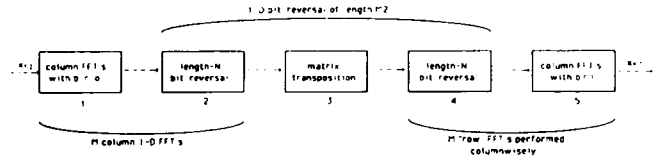


Fig. 1. A 2-D row column FFT algorithm.

choose first to bit reverse the data and then to apply FFT's with bit-reversed inputs. This overall organization is summarized in Fig. 1.

When decomposed so, the 2-D row-column algorithm is made up of five different parts, three of which are essentially permutations, and are performed consecutively. In fact, these three permutations can easily be shown to form a single 1-D bit reversal of length $N = M^2$ as follows.

Let b_{ri} (respectively, b_{rj}) be the bit-reversed value of index i (respectively, j), and $X'_{i,j}$ be the output of the column 1-D FFT's with bit-reversed outputs. The first block of Fig. 1 provides $X'_{bri,j}$, the matrix transposition provides $X'_{j,bri}$, while the next bit reversal provides $X'_{brj,bri}$. And, if the DFT has a length $2^m \cdot 2^m$, the index (b_{rj}, b_{ri}), is seen to be the bit-reversed value of the index (i, j):

$$(b_{rj} \cdot 2^m + b_{ri}) = b_r(i \cdot 2^m + j). \qquad (1)$$

Although almost trivial, (1) may be especially useful for digital signal processor (DSP) implementations of the 2-D FFT since most of the available digital signal processors now have bit-reverse addressing capabilities. This very simple consideration completely avoids the matrix transposition, and the three permutations turn out to be free: they are simplified to a special indexing available in the processor.

For hardware implementations, the application is straightforward, and for software implementations on general purpose computers, performing a single bit reversal of length $N = M^2$ turns out to be much more efficient than performing the three original operations separately (which were $2M$ bit reversals of length M plus the matrix transposition).

A remark should be made here: the two boxes of Fig. 1 labeled 1 and 5, performing respectively, an FFT with bit-reversed output and an FFT with bit-reversed input, can be of exactly the same type when implemented on processors with bit-reversed addressing capabilities: box 5 may be the same program as box 1, with data accessed in the bit-reverse mode. This performs the desired changes.

III. Bit-Reversal Algorithms Based on this Approach

This simple connection between bit reversal and matrix transposition can provide an efficient way of programming a bit-reversal algorithm for a set of data of length N, in cases where the data are stored either in core memory or in mass memory.

A. When Data are Stored in Core Memory

The bit-reverse algorithm which is mostly used [7] is a bit-reverse counter, providing i and b_{ri} in sequence. About N bit-reverse counts are required, a test $(b_{ri} < i)$ being required to decide whether the exchange between $x(i)$ and $x(b_{ri})$ has already been done.

Nevertheless, the considerations above are easily seen to provide a simple way to bit reverse the whole set of data by computing about $N^2/2$ bit-reverse counts, without any test, since a matrix transposition does not require any.

In fact, let us consider a bit reverse of length $N = 2^m \cdot 2^m$ and cut the indexes in two halves. In terms of these two halves, i and j, (1) provides a correspondance between an index and its bit-reversed value. Furthermore, it is useful to exchange the roles of i and b_{ri}:

$$(b_{ri} \cdot 2^m + i) = b_r(b_{ri} \cdot 2^m + j). \qquad (2)$$

Hence, (2) provides a way (by applying it for $i > j$, or $j > i$, for example), to obtain a couple of indexes to swap, for which the bit reversal has not yet been performed (demonstration is very easy). Therefore, no test is required, and this brings to a minimum the number of bit-reverse counts to be performed. Extension to odd powers of 2 is also very easy, by considering separately the two cases where the middle digit is 0 or 1.

A Fortran program implementing this approach is given in Fig. 2. This program is exactly a matrix transposition algorithm with one bit-reverse counter inside each loop. The outermost loop takes into account lengths that are odd powers of 2.

It turns out that algorithms possessing about the same characteristics were proposed in a recent paper by Evans [4], and in a much earlier paper by Polge *et al.* [5], each time with different explanation and implementation. The implementation we give in Fig. 2 differs essentially in that it does not require any auxiliary storage, which makes it suitable when only a few FFT's of the same length are to be performed. (The other ones, given in [4] and [5], require the precomputation of an auxiliary storage of length $\sqrt{N}$ containing bit-reversed values, and can hardly be considered as performing the bit reversal "in place.")

The program of Fig. 2, although being fairly compact, is about twice as fast as the usual one, but much slower than the ones described in [4], [5]. Nevertheless, replacing the bit-reverse counters inside the loops by straightforward reading of precomputed bit-reversed indexes from 1 to $\sqrt{N}$ makes this algorithm as fast as the ones in [4], [5]. This shows that most of the gain obtained in [4] was due to the use of an auxiliary storage of length $\sqrt{N}$.

B. When Data are Stored in Mass Memory

This approach also gives possibilities for bit reversing a set of data of length $N = 2^{2m}$ too huge to remain in core memory, with the advantage of a reduced number of transfers between mass and core memory. Once again, this may happen either when reordering a huge set of data coming from 1-D FFT, or when implementing row-column 2-D FFT's using the scheme in Fig. 1. In fact, data that are stored in mass memory are only available by blocks: reading one value means reading the whole block in which the value is stored. It is therefore important to apply an algorithm in which most operations are per-

```
      subroutine newbr2(x,n,m)

c
c     performs bit-reverse reordering of vector x
c     x has length n = 2**m
c
      dimension x(n)
c
      m1 = m/2
      n1 = 2**m1
      nn = n/2
      do 10 ipair=0,(m-m1-m1)*n1,n1
      ibr = 0
      do 20 i=2*ipair,n1+ipair
 21   k = nn
 22   if(k.gt.ibr) go to 23
 24   ibr = ibr-k
      k = k/2
      if(k.le.ibr) go to 24
 23   ibr = ibr+k
c
c
      jbr = 0
      do 30 j=ibr+ipair+1,ibr+i-1
      xt = x(jbr+i)
      x(jbr+i) = x(j)
      x(j) = xt
 31   k = nn
 32   if(k.gt.jbr) go to 33
 34   jbr = jbr-k
      k = k/2
      if(k.le.jbr) go to 34
 33   jbr = jbr+k
c
 30   continue
 20   continue
 10   continue
      return
      end
```

Fig. 2. An improved bit-reversal algorithm.

formed inside the blocks. Application of (1) makes this possible.

The procedure is as follows:

1) Bit-reverse the data by blocks of length 2^m.

2) "Transpose" the data, by considering them as a matrix of length $2^m \cdot 2^m$. This can be done with a fast algorithm by Eklundh [6], that requires m read/write passes over the matrix if the core memory can only handle two columns (2^{m+1} values).

3) Bit-reverse data by blocks of length 2^m before storing them.

Careful examination of the fast matrix transposition shows that the two bit-reverse operations can be included in the first and last step of Eklundh's algorithm: The algorithm requires m read/write cycles of the data to be performed. The three steps above can be performed by applying a length-2^m bit reversal at the first pass of the algorithm, after the first read of each column, and at the last pass of the fast transposition, before the last write of each column. Since Eklundh's algorithm has a very regular structure, this can be performed very easily.

This procedure can also be improved by including the two bit reversals into the fast transposition algorithm, which reduces heavily the computational load.

The trick here is to decompose the bit reversal of i by using a kind of "decimation in frequency" algorithm. The bit reversal of j is decomposed accordingly into a "decimation-in-time" algorithm. Each of these algorithms requires m steps, the same number as is required in Eklundh's algorithm. The fast bit-reversal algorithm is then obtained by merging the kth step of each of the algorithms for the computation of b_{ri} (point 1 below), matrix transpose (point 2 below), and b_{rj} (point 3 below). Furthermore, points 1, 2, and 3 can be seen to merge into a very simple permutation, which is as simple as one step of Eklundh's algorithm. The overall algorithm is then as follows:

at each step n^0k, $k = 0, \cdots, m-1$

read columns i and $j = i + 2^{m-k-1}$, $i = 0, \cdots, 2^m - 1$, by step of 2^{m-k}

1-2-3-replace col.i by the even terms of col.i, followed by the odd terms of col.j, and replace col.j by the odd terms of col.i, followed by the odd terms of col.j.

The bit-reverse algorithm of length $N = 2^m \cdot 2^m$ has now exactly the same complexity as a fast matrix transposition, which means that, in Fig. 1, the two boxes labeled "1-D bit-reversal" are given for free.

With the algorithm above, we have obtained a bit-reversal algorithm that needs m read/write cycles for a DFT of length $2^m \cdot 2^m$, the data being read by blocks of length 2^m. Note that, as was the case for Eklundh's algorithm, this procedure can be performed as soon as two rows at a time can be stored in core memory.

An example is given in Fig. 3, corresponding either to a bit reversal of length 64, or to the three central permutations of Fig. 1, for an 8.8 FFT.

A sample program is given in Fig. 4. This program intends only to be easily understandable. Read and write instructions should be modified according to the type of storage device that is used.

Both algorithms could have been explained in another (more fundamental) way: The basic equation (1) on which these algorithms are based can be explained as a divide and conquer approach applied to the bit reversal, seen as a multidimensional problem. If some index has to be bit reversed, it is possible to separate its binary representation into several parts ("digits"). The bit-reversed value of the index is given by the digit reverse of the bit-reversed digits. This is a tensor product of permutations. This more theoretical presentation allows an easy generation of higher radix divide and conquer approaches. These approaches would reduce the number of read/write passes on the data if more than two rows can be stored in core memory, for example. The drawback is that special cases are to be taken care of, depending on the parities of m.

There is also a possibility that the bit reversal, seen as an m-step procedure, can be merged with vector-radix algorithms [8], step by step, resulting in efficient in-order 2-DFT algorithms, similar to the ones in [9].

Let us also remark here that both types of algorithms described in Sections III-A and -B are fundamentally different: when data are stored in core memory, it is more efficient to compute the index and its bit-reversed value first, and then to swap the corresponding data. But, when the data are stored in mass memory, data are swapped m times, allowing the bit reversal to be performed progressively on the whole matrix.

IV. Conclusion

In this paper, we have shown that a very simple connection between bit reversal and matrix transposition (the bit reversal can be seen as a kind of multidimensional matrix transposition through recursive use of (1) had many

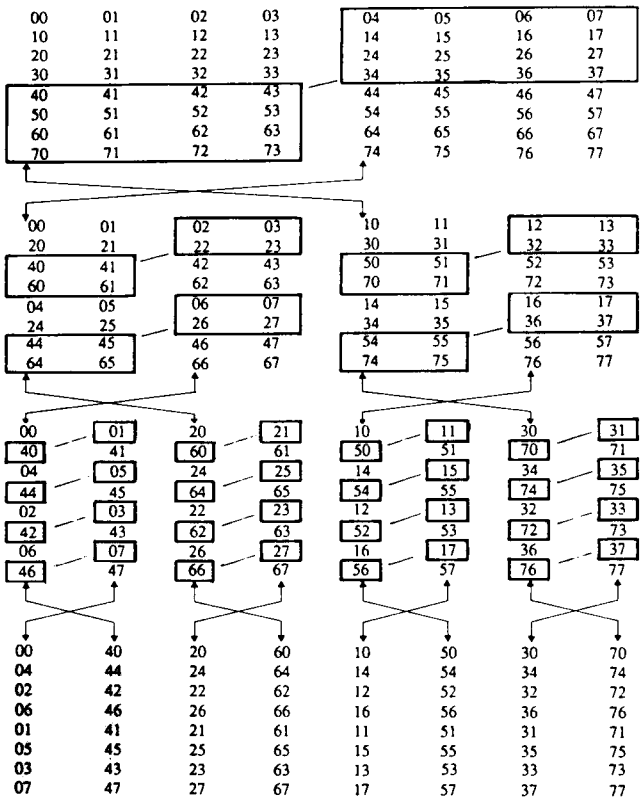


Fig. 3. Progression of the bit-reversal procedure.

```
      subroutine newbr3(x,n,m)
c
c     performs row and column bit-reversals
c     plus matrix transposition of x
c     x has dimension (n,n) = (2**m,2**m)
c
      dimension x(128,128)
      dimension t0(256)
      n2 = n
      n1 = n2+n2
      do 30 k=1,m
      n1 = n2
      n2 = n2/2
      do 20 ii=1,n,n1
      do 20 i=ii,ii+n2-1
      ipn2 = i+n2
      do 10 j=1,n
c
c     read col i and ipn2
c
      t0(j) = x(j,i)
      t0(j+n) = x(j,ipn2)
10    continue
      do 15 j=1,n
c
c     write col i and ipn2
c
      x(j,i) = t0(j+j-1)
      x(j,ipn2) = t0(j+j)
15    continue
20     continue
30    continue
      return
      end
```

Fig. 4. Sample program for the second algorithm.

consequences, such as:

1) no need for matrix transposition in 2-D row-column FFT's, neither when implemented on DSP's with bit-reversed addressing capabilities, nor when implemented in hardware;
2) improvement of the 2-D row-column FFT on general-purpose computers (a single bit reversal of length M^2 replaces $2.M$ bit-reversals of length M, plus a matrix transposition);
3) a new description of a recently proposed bit-reversal algorithm, which is mainly a matrix transposition with bit-reversed indices;
4) a new algorithm for bit reversing a set of data of length M^2, requiring only $\log_2 M$ read/write cycles when data are stored in mass memory (useful for row-column implementations of 2-D FFT's, as seen in points 1 and 2).

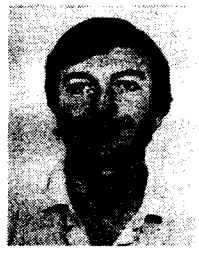
This shows that, in nearly all implementations of 2-D row-column FFT's, either the bit reversals or the matrix transposition can be given for free.

Acknowledgment

Most of these results are due to a problem stated by P. Gole who wanted to accelerate the computation of 2-D Hadamard transforms. Thanks are also due to L. Auslander for pointing out the tensor product interpretation.

References

[1] P. Duhamel, "Implementation of split-radix FFT algorithms for complex, real, and real-symmetric data," *IEEE Trans. Acoust., Speech, Signal Processing*, vol. 24, no. 2, pp. 285–295.
[2] L. R. Morris, "Digital signal processing software," DSPSW Inc., Ottawa, Canada, 1983.
[3] M. Vetterli and P. Duhamel, "Split-radix algorithms for length-p^m DFT's," *IEEE Trans. Acoust., Speech, Signal, Processing*, vol. 37, no. 1, pp. 57–64, Jan. 1989.
[4] D. M. Evans, "An improved digit-reversal permutation algorithm for the fast Fourier transform and Hartley transform," *IEEE Trans. Acoust., Speech, Signal Processing*, vol. ASSP-35, no. 8, pp. 1120–1125, 1987.
[5] R. J. Polge, B. K. Bhaganan, and J. M. Carswell, "Fast computational algorithms for bit reversal," *IEEE Trans. Comput.*, vol. C-23, pp. 1–9, 1974.
[6] J. O. Eklundh, "A fast computer method for matrix transposing," *IEEE Trans. Comput.*, vol. C-21, pp. 801–803, July 1972.
[7] C. S. Burrus, and T. W. Parks, *DFT/FFT and Convolution Algorithms*. New York: Wiley, 1985.
[8] Z. J. Mou and P. Duhamel, "In-place butterfly style FFT of 2-D real sequences," *IEEE Trans. Acoust., Speech, Signal Processing*, vol. 36, no. 10, pp. 1642–1650, Oct. 1988.
[9] J. Johnson, R. Johnson, D. Rodriquez, and R. Tolimieri, "A methodology for designing, modifying, and implementing Fourier transform algorithms on various architectures," private communication.



Pierre Duhamel (M'87-SM'87) was born in France in 1953. He received the Ingenieur degree in electrical engineering from the National Institute for Applied Sciences (INSA), Rennes, France, in 1975, the Dr. Ing. degree in 1978, and the Doctorat es Sciences in 1986, both from Orsay University, France.

From 1975 to 1980 he was with Thomson-CSF, Paris, France, where his research interests were in circuit theory and signal processing, including digital filtering and analog fault diagnosis. In 1980, he joined the National Research Center in Telecommunications (CNET), Issy-les-Moulineaux, France, where his activities were first concerned with the design of recursive CCD filters. He is now working on fast Fourier transforms and convolution algorithms, and on the application of the same techniques to adaptive filtering, spectral analysis, and wavelet transforms. He is a member of the DSP Committee and is now serving as an Associate Editor of the IEEE TRANSACTIONS ON ACOUSTICS, SPEECH, AND SIGNAL PROCESSING.

T-ASSP/35/6/13866

Improved Fourier and Hartley Transform Algorithms: Application to Cyclic Convolution of Real Data

P. Duhamel
M. Vetterli

Reprinted from: IEEE TRANSACTIONS ON ACOUSTICS, SPEECH, AND SIGNAL PROCESSING,
Vol. ASSP-35, No. 6, JUNE 1987

Improved Fourier and Hartley Transform Algorithms: Application to Cyclic Convolution of Real Data

PIERRE DUHAMEL, MEMBER, IEEE, AND MARTIN VETTERLI, MEMBER, IEEE

Abstract—**This paper highlights the possible tradeoffs between arithmetic and structural complexity when computing cyclic convolution of real data in the transform domain.**

Both Fourier and Hartley-based schemes are first explained in their usual form and then improved, either from the structural point of view or in the number of operations involved.

Namely, we first present an algorithm for the in-place computation of the discrete Fourier transform on real data: a decimation-in-time split-radix algorithm, more compact than the previously published one. Second, we present a new fast Hartley transform algorithm with a reduced number of operations.

A more regular convolution scheme based on FFT's is also proposed.

Finally, we show that Hartley transforms belong to a larger class of algorithms characterized by their "generalized" convolution property.

I. Introduction

THE radix-2 fast Fourier transform algorithm was first explained by Cooley and Tukey in 1965 in a version on complex data [1]. But, as most of the data to be treated are real instead of complex, the need for a version of the FFT on real data removing extra calculations from the complex case became soon apparent, and such a program was quickly published by Bergland [3].

In fact, this program does not seem to have been widely used, and many people preferred either to compute a length 2^n real DFT either by the use of a length 2^{n-1} complex FFT plus some additional operations, or to compute two real FFT's at a time by using one length 2^n complex FFT.

Let us also notice that a version of the radix-2 FFT algorithm was also published in [4] for symmetric real-values series, and that the algorithm by Preuss [2] can be straightforwardly applied to real and real symmetric data.

More recently, the 2^n FFT algorithms with the minimum known number of both multiplications and additions were also proposed in a real-data version [6], [7], [10], [11].

At about the same time, Bracewell [8] proposed the use of fast discrete Hartley transforms (FDHT), which are real in nature, as a substitute to the FFT for computing cyclic convolutions of real data.

Manuscript received March 14, 1986; revised December 10, 1986.

P. Duhamel is with CNET/PAB/RPE, 92131 Issy-les-Moulineaux, France.

M. Vetterli was with the Ecole Polytechnique Fédérale de Lausanne, CH-1007 Lausanne, Switzerland. He is now with the Center for Telecommunications Research, Columbia University, New York, NY 10027.

IEEE Log Number 8613866.

But it was not clear at all what approach was best suited for cyclic convolutions of real sequences when taking into account the structure of the algorithm as well as its computational complexity.

Nevertheless, the well-known approach of computing real convolutions through FFT algorithms on complex data will not be considered here, and we shall only consider programs specially designed for real data since they obtain the lowest known number of arithmetic operations without an increase in program length.

In Section II, we shall first briefly present the two basic approaches considered here: convolution using FFT's and convolution using FDHT's. This comparison shows that the former has some advantages when considering arithmetic complexity, while the latter has a simpler structure since the DHT is self-inverse.

Arithmetic complexities for these cyclic convolution schemes are given when the FFT and FDHT are computed by the algorithms with the lowest known number of both multiplications and additions.

Sections III and IV describe improved FFT and FDHT programs, together with a more regular FFT-based convolution scheme.

First, a decimation-in-time split-radix FFT algorithm on real data is presented, with a more regular structure and a more compact code than the previously published DIF one (it has exactly the same arithmetic complexity).

Section IV presents an improved FDHT algorithm, allowing us to obtain any length -2^n FDHT using only two additions more than the best FFT algorithms known for real data (four additions more for the whole convolution). In this section, we also show how to derive improved FDHT algorithms from their FFT counterparts.

Finally, Section IV discusses a generalized convolution property, which includes the various approaches presented in this paper.

II. The Initial Schemes

A. Convolution Using FFT's

Let $\{X_k^f\}$ be the DFT of $\{x_n\}$:

$$X_k^f = \sum_{n=0}^{N-1} x_n \left(\cos \frac{2\pi nk}{N} - j \sin \frac{2\pi nk}{N} \right) = \sum_{n=0}^{N-1} x_n W_N^{nk}. \quad (1)$$

0096-3518/87/0600-0818$01.00 © 1987 IEEE

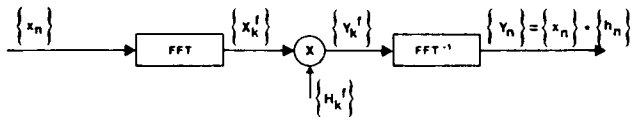


Fig. 1. Usual convolution scheme using FFT's.

TABLE I
NUMBER OF ARITHMETIC OPERATIONS FOR A CYCLIC CONVOLUTION ON REAL DATA

		Number of Additions			
N	Number of Mult.	Initial FFT-Based	Initial FDHT-Based	FFT-Based with 2 Forward FFT's	Improved FDHT-Based
8	15	49	53	55	53
16	43	141	153	155	145
32	115	373	393	403	377
64	291	933	977	995	937
128	707	2245	2329	2371	2249
256	1667	5253	5425	5507	5257
512	3843	12 037	12 377	12 547	12 041
1024	8707	27 141	27 825	28 163	27 145
2048	19 459	60 421	61 785	62 467	60 425

Since the DFT has the convolution property, the convolution scheme has the structure given in Fig. 1.

When the initial sequences $\{x_n\}$ and $\{h_n\}$ are real, $\{X_k\}$, $\{H_k\}$, and $\{Y_k\}$ have Hermitian symmetry: $Y_k = Y^*_{-k}$, and the forward DFT has to be performed on real data, while the inverse FFT has to be performed on data with Hermitian symmetry. Both kind of FFT's have been published [7], [11]. They have the same arithmetic complexity, and can be performed in place. (In fact, an FFT on data with Hermitian symmetry can be derived by reversing the flow graph of an FFT on real data.)

If the FFT's are performed with the lowest arithmetic complexity (lowest number of both additions and multiplications) using FFCT [6] or split-radix [7] algorithms, it leads to the following number of operations:

$$M_n^{\text{conv. FFT}} = 2M_n^{fr} + 2 + 3 \cdot (2^{n-1} - 1) = 2^{n-1}(2n - 3) + 3 \quad (2)$$

$$A_n^{\text{conv. FFT}} = 2A_n^{fr} + 3(2^{n-1} - 1) = 2^{n-1}(6n - 7) + 5 \quad (3)$$

where M_n^{fr} (respectively, A_n^{fr}) is the number of multiplications (respectively, additions) needed to compute a length 2^n DFT on real data.

In these operation counts, it has been taken into account that, due to the symmetry of both X_k^f and H_k^f, Y_k^f can be obtained with two real multiplications for $k = 0$ and $k = N/2$, and $N/2 - 1$ complex multiplications (three real mults + three real adds each) for $k = 1, \cdots, N/2 - 1$.

This arithmetic complexity is rather low, as shown in Table I, since it is based on a powerful FFT algorithm for real data.

Let us recall here that both split-radix and FFCT algorithms meet the minimum possible number of nontrivial real multiplications up to and including length $N = 16$, and the minimum possible number of nontrivial complex multiplications (i.e., $\neq j, \pm 1$) up to and including length $N = 64$, while possessing the lowest known number of additions needed to compute a length $N = 2^n$ DFT.

Nevertheless, a disadvantage of this scheme is that it needs both forward transform on real data and inverse transform on complex data with Hermitian symmetry. This results in an increase in program length for the cyclic convolution.

B. Convolution Using FDHT's

The discrete Hartley transform is defined as the sum of the real and imaginary part of the Fourier transform:

$$X_k^h = \sum_{n=0}^{N-1} x_n \left(\cos \frac{2\pi nk}{N} + \sin \frac{2\pi nk}{N} \right). \quad (4)$$

As can be seen from the definition, the DHT is its own inverse, and it is a real transform. Furthermore, the convolution in the time domain corresponds to the following operation in the Hartley domain:

$$Y_k^h = X_k^h \cdot H_e^h(k) + X_{-k}^h \cdot H_0^h(k) \quad (5)$$

where $H_e^h(k)$ [respectively, $H_0^h(k)$] is the even (respectively, odd) part of the Hartley transform of $\{h_n\}$.

So, the convolution scheme using Hartley transforms becomes as shown in Fig. 2.

When considering (5), the "multiplications" in the Hartley domain seem to need two multiplications per point, but it it easy to see that by grouping the computation of Y_k^h and Y_{-k}^h, this number can be reduced to 3/2 multiplications per point:

$$\begin{bmatrix} Y_k^h \\ Y_{-k}^h \end{bmatrix} = \begin{bmatrix} H_e(k) & H_0(k) \\ -H_0(k) & H_e(k) \end{bmatrix} \begin{bmatrix} X_k^h \\ X_{-k}^h \end{bmatrix} \quad (6)$$

and (6) is easily recognized as a complex multiplication, needing only three real multiplications and three additions.

If we consider also that for $k = 0$ and $N/2$, (6) reduces

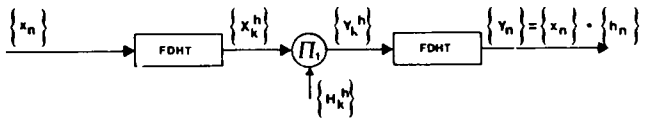


Fig. 2. Convolution scheme based on FDHT's.

to

$$Y_0^h = X_0^h \quad H_e(0)$$
$$Y_{N/2}^h = X_{N/2}^h \quad H_e(N/2) \tag{7}$$

one can see that the set of "multiplications" in the Hartley domain needs exactly the same number of operations as in the Fourier domain.

The remaining part of the scheme given in Fig. 2 to be evaluated consists of the FDHT.

FDHT was first introduced by Bracewell [8] in a radix-2, decimation-in-time version, and Sorensen *et al.* [9] recently proposed different types of FDHT algorithms. (See also [15].)

Inspection of these algorithms allows the following conclusions.

A radix-2 FDHT algorithm needs $N - 2$ additions more than the corresponding FFT algorithm on real data and the same number of multiplications.

The split-radix algorithm, which seems already to be one of the best compromises for real-data FFT's (lowest known number of operations, in-place computation, compact program), also gives the lowest known number of operations for the FDHT, but requires $2(2^{n-1} - (-1)^{n-1}/3)$ additions more than the corresponding split-radix FFT algorithm on real data.

If the FDHT in Fig. 2 is computed through a split-radix algorithm, this gives the following arithmetic complexity for the cyclic convolution:

$$M_n^{\text{conv. DHT}} = 2^{n-1}(2n - 3) + 3 \tag{8}$$

$$A_n^{\text{conv. DHT}} = 2^{n-1}(6n - 7) + 5 + 4\frac{2^{n-1} - (-1)^{n-1}}{3}$$
$$= 2^{n-1}\left(6n - \frac{17}{3}\right) + 5 - \frac{4}{3}(-1)^{n-1}. \tag{9}$$

One can see that convolution using FDHT's will have a simpler structure, even if the FDHT programs have the same code length as FFT programs, due to the self-inverse property of the Hartley transform. This simplicity will be obtained at the cost of a small increase of the number of additions.

III. Improved FFT-Based Convolution Schemes

These initial schemes described in Section II can be improved in two ways: either by an improvement of the basic algorithm (FFT on real data, FDHT) or by an improvement of the structure of the convolution scheme. The FFT-based convolution scheme is relevant to both improvements.

A. A DIT Split-Radix FFT Algorithm on Real Data

The decimation-in-frequency (DIF) split-radix decomposition applies a radix-2 decomposition to the even-indexed samples, and a radix-4 decomposition to the odd-indexed samples of the transform $\{X_k\}$. The decimation-in-time (DIT) split-radix decomposition of the DFT, being the dual algorithm of the DIF SRFFT, considers separately the even-indexed samples $\{x_{2n}\}$, the samples $\equiv 1 \pmod 4$, $\{x_{4n+1}\}$, and the samples $\equiv 3 \pmod 4$: $\{x_{4n+3}\}$. This DIT decomposition is given in (10).

$$X_k^f = \sum_{k=0}^{N/2-1} x_{2n} W_N^{2nk} + W_N^k \sum_{n=0}^{N/4-1} x_{4n+1} W_N^{4nk} + W_N^{3k} \sum_{n=0}^{N/4-1} x_{4n+3} W_N^{4nk}. \tag{10}$$

When the initial sequence $\{x_n\}$ is real, (10) involves DFT's of real data only. Furthermore, since these DFT's have Hermitian symmetry, it is possible to work "in place" by using, for a DFT of length L, the $L/2 + 1$ first locations for the real part of the corresponding samples of the transform, and the $L/2 - 1$ remaining ones for the imaginary part.

When two butterflies are done at the same time, this algorithm requires exactly the same number of operations as the DIF split-radix FFT on real data [7] and the FFCT algorithm [6], [11], but requires only four different types of butterflies to meet the minimum number of operations, thus resulting in a more compact program. This has to be compared to the eight different butterflies needed by the DIF SRFFT [7] and the absence of compact looped programs for implementing FFCT [6].

The Fortran code corresponding to this algorithm is given in Appendix I. This program uses the indexing scheme given by Sorensen *et al.* in [12], which gives the usual three-loop structures of radix-2 or radix-4 programs.

It should be remarked that this program was written to be as compact as possible, and that a much faster one could be obtained by following the approach that was used in the complex case [7].

Since run-time comparisons on general-purpose machines are very machine-dependent (and even compiler-dependent), timings will not be given here. Nevertheless, precise timings are more important on DSP's: we could obtain run times as low as 235 μs for a length-64 real data FFT on a TMS 32010 (TI's benchmarks are 400.2 μs) and 572 μs for a length-128 real-data FFT (versus 955 ms for TI's benchmarks).

B. A More Regular Convolution Scheme Using FFT's

Even if the FFT program is compact and effective, the convolution scheme of Fig. 1 still requires two different FFT programs: a forward FFT on real data, and an inverse FFT on data with Hermitian symmetry.

We shall now propose a new scheme using two forward

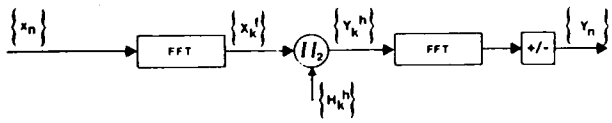


Fig. 3. Modified FFT-based convolution scheme using only forward FFT's on real data.

FFT's on real data, at the cost of a few more additions: this improvement is based on some of the remarks made when studying FDHT's.

The convolution property in the Hartley domain, as given in (5), can be rewritten as follows, by reversing the roles of X_k and H_k:

$$Y_k^h = X_e^h(k) \cdot H_0^h(k) H_{-k}^h. \tag{11}$$

But $X_e^h(k)$, being the even part of the Hartley transform of $\{x_n\}$, is also the real part of the DFT of $\{x_n\}$, and $X_0^h(k)$ is the corresponding imaginary part.

We can then rewrite (6) as follows:

$$\begin{bmatrix} Y_k^h \\ Y_{-k}^h \end{bmatrix} = \begin{bmatrix} H_k^h & H_{-k}^h \\ H_{-k}^h & -H_k^h \end{bmatrix} \begin{bmatrix} \operatorname{Re} X_k^f \\ \operatorname{Im} X_k^f \end{bmatrix} \tag{12}$$

which is also equivalent to a complex product.

Furthermore, since a DHT is the sum of the real and imaginary part of a DFT, the last FDHT in Fig. 2 can be replaced by an FFT followed by the sum of the real and imaginary part of each DFT component, thus resulting in the diagram of Fig. 3.

This scheme requires exactly the same number of operations as the usual convolution scheme based on FFT's, plus $N - 2$ additions due to the $+/-$ box:

$$A_n^{\text{conv. fft}} = 2^{n-1}(6n - 5) + 3. \tag{13}$$

This structure is now as regular as the one using FDHT's, to the price of about $2^n/3$ extra additions compared to the FDHT scheme. It has the advantage of using only usual FFT algorithms on real data, which should become widely used due to their simplicity and compactness.

IV. Improved FDHT Algorithm

As can be seen from Section II-B, DHT's and DFT's are very simply related. In fact, FDHT algorithms, as found in [8] and [9], can be improved by making full use of the close relationship between Fourier and Hartley transforms. In fact, nearly all the extra additions needed to compute FDHT's instead of FFT's can be nested inside the twiddle factors of the FFT, thus resulting in hybrid FDHT/FFT algorithms.

By multiplying the DFT of $\{x_n\}$ by $(1 + j)$, we obtain (14):

$$\begin{aligned}(1 + j)X_k^f &= (\operatorname{Re} X_k^f - \operatorname{Im} X_k^f) + j(\operatorname{Re} X_k^f + \operatorname{Im} X_k^f) \\ &= X_k^h + jX_{-k}^h \end{aligned} \tag{14}$$

which means that multiplication of X_k^f by $(1 + j)$ gives us the Hartley transform of $\{x_n\}$.

Let us apply this remark to the basic split-radix decimation-in-time decomposition of the DFT [10] as given in (10):

$$\begin{aligned}(1 + j) \cdot X_k^f = X_k^h + jX_{-k}^h = (1 + j) &\sum_{n=0}^{N/2-1} x_{2n} W_N^{2nk} \\ &+ (1 + j) W_N^k \sum_{n=0}^{N/4-1} x_{4n+1} W_N^{4nk} \\ &+ (1 + j) W_N^{3k} \sum_{n=0}^{N/4-1} x_{4n+3} W_N^{4nk} \end{aligned} \tag{15}$$

and with the following notations

$$\begin{aligned} \{X_k^{2h}\} &\triangleq \text{DHT of } \{x_{2n}\} \\ \{X_k^{1f}\} &\triangleq \text{DFT of } \{x_{4n+1}\} \\ \{X_k^{3f}\} &\triangleq \text{DFT of } \{x_{4n+3}\} \end{aligned}$$

(15) can now be rewritten as follows;

$$\begin{aligned} X_k^h + jX_{-k}^h = X_k^{2h} &+ jX_{-k}^{2h} \\ &+ [(1 + j) \cdot W_N^k] X_k^{1f} \\ &+ [(1 + j) W_N^{3k}] X_k^{3f}. \end{aligned} \tag{16}$$

Equation (16) is now the basic recursion of the improved FDHT algorithm: it transforms the length N DHT into a length $N/2$ DHT plus two length $N/4$ DFT's, and some multiplications by twiddle factors of the form $(1 + j) W_N^k$ or $(1 + j) W_N^{3k}$.

As in the case of FFT's on real data, the butterflies with twiddle factors W_N^k have to be computed together with the ones with twiddle factors W_N^{-k}. This needs $(N/8 - 1)$ general 6 mult–18 adds butterflies, plus one 0 mult–6 adds butterfly (corresponding to $k = 0$) and one 2 mult–4 adds butterfly (corresponding to $k = N/8$).

Iteration of these addition counts gives exactly the same number as that obtained for FFT's on real data, except for the FDHT of length 4 which needs two extra additions.

This leads to

$$A_n^{\text{hart}} = 2^{n-1}(3 \cdot n - 5) + 6 \tag{17}$$

and the whole convolution with this algorithm will need

$$A_n^{\text{conv hart.}} = 2^{n-1}(6n - 7) + 9. \tag{18}$$

The number of multiplications is the same as for the other schemes.

The convolution scheme using this new fast Hartley transform algorithm now has the same number of multiplications and four additions more than the best known scheme, with the advantage of using only a single DHT program to be used twice.

This result is interesting theoretically since it is another example, besides [6], that uses a decomposition of a transform into smaller transforms of another type to achieve a reduction in arithmetic complexity.

Furthermore, the process used for deriving this new algorithm has interesting characteristics: it shows how the

additional operations used to convert transforms of one type (FFT's) into transforms of another type (FDHT's) can be nested inside existing algorithm, and hence can almost disappear. This may be useful for other transforms as well.

Note also that a reverse procedure is also of interest since the same procedure can be used to convert FDHT algorithms into FFT algorithms; this shows that it is hopeless to find FDHT algorithms requiring fewer arithmetic operations than the corresponding FFT algorithms. In fact, if there was a possibility of finding improved FDHT algorithms, they could be used as a starting point for deriving improved FFT algorithms using the technique described above. We have thus demonstrated that the only improvement that can be expected from FDHT's is a more regular structure (self-inverse property), a fact which was becoming more and more apparent from the recent literature.

From a practical point of view, it should be emphasized that this new algorithm brings only a slight improvement ($N/3$ additions) over the split-radix FDHT algorithm proposed in [9]. Nevertheless, a Fortran implementation is given in Appendix II for the purpose of providing the precise computation of the "butterflies" converting the FFT into an FDHT. It should be clear from what has been stated above that the improvement in the number of additions would produce an increase in the computational speed only when in-line code is used. This happens when programming DSP's or when using autogen techniques [16]. Otherwise, the program given in Appendix II will certainly be slower than the one in [9].

V. The "Generalized" Convolution Property

In fact, the existence of FDHT, which is a real-valued transform possessing some kind of convolution property, states a theoretical problem since it has been shown in [13] that real transforms having a convolution property could not exist.

The usual convolution property is defined as follows:

$$\text{let } y_n = \{x_n\} * \{h_n\} = \sum_{i=0}^{N-1} x_i \otimes h_{n-i}. \qquad (19)$$

The considered transform T will have the so-called convolution property if

$$T\{y_n\} = T\{x_n\} \otimes T\{h_n\} \qquad (20)$$

where $\otimes$ is the usual term-by-term multiplication. In other words, the convolution product in the time domain becomes a usual multiplication in the transform domain.

But it was shown in [13] that the existence of "square" transforms having the convolution property depends only on the existence of an α that is a root of unity of order N and on the existence of N^{-1}. In the case of the DFT, $\alpha = W_N$, as defined in (1), but the Hartley transform is not such a transform, although it has some kind of convolution property. (In fact, it was also shown that a real transform having the convolution property could not exist for $N > 2$.)

Nevertheless, this contradiction is only apparent since the "multiplication" in the Hartley domain is not the same one as is defined in (19).

The FDHT belongs then to the larger class of transforms having the "generalized" convolution property

$$\text{let } y_n = x_n * h_n = \sum_{i=0}^{N-1} x_i \otimes h_{n-i}; \qquad (21)$$

then

$$T'\{y_n\} = T'\{x_n\} \circledast T'\{h_n\} \qquad (22)$$

where $\circledast$ is allowed to be different from the usual term-by-term multiplication.

In fact, Ansari [14] has already shown that the discrete Hartley transform is not the only combination of the DFT coefficients having the generalized cyclic convolution property.

Further research of other members of this class of transforms should be of interest.

VI. Conclusion

In this paper, we have first presented two usual schemes for cyclic convolution of real signals via FFT or FDHT. Operation counts are given for both of them using the fastest known algorithms, showing that the FFT-based convolution has a lower arithmetic complexity, but a more complex structure.

Improvements are then made to both schemes since we propose

- an improved FFT algorithm on real data with increased regularity and compactness
- an improved FDHT algorithm which uses less operations than the previously proposed algorithms
- an FFT-based scheme using two forward transforms on real data
- an FDHT-based algorithm with reduced number of additions.

All operation counts are summarized in Table I, allowing one to choose the best tradeoff between low arithmetic complexity and structural complexity.

As can be seen from Table I, the difference between the arithmetic complexities involved are so small that they will not have a strong influence on the timings of the different algorithms. Furthermore, the FDHT-based schemes require the use of a special-purpose program for computing the FDHT of the input signal that will be used only for the computation of cyclic convolutions since the Hartley transform has no physical significance in itself.

These are the reasons why we believe that, in many circumstances, the scheme using two forward FFT's should be used since it provides a good compromise between structural complexity, arithmetic complexity, and is based on an FFT program that should become widely used, due to its low arithmetic complexity and regular structure.

APPENDIX I
SPLIT-RADIX, DECIMATION-IN-TIME FAST FOURIER TRANSFORM

$n = 2^m$: length of the transform.
x must be in natural order on input.

On output, x_k will contain the real part of X_k, $k = 1, \cdots, n/2 + 1$ and the imaginary part of X_k, $k = n/2, \cdots, n$.

This program uses the usual 4 mults-2 adds complex multiplication algorithm.

```
      subroutine splitdit(x,n,m)
      dimension x(1)
      data rac2s2/0.707106778/
c
c-----------------digit reverse counter----------
c
 1    j = 1
      n1 = n-1
      do 5 i=1,n1
      if(i.ge.j) go to 2
      xt = x(j)
      x(j) = x(i)
      x(i) = xt
 2    k = n/2
 3    if(k.ge.j) go to 4
      j = j-k
      k = k/2
      go to 3
 4    j = j+k
 5    continue
c
c-----------------length two transforms----------
c
      is = 1
      id = 4
 10   do 20 i0=is,n,id
      i1 = i0+1
      r1 = x(i0)
      x(i0) = r1+x(i1)
      x(i1) = r1-x(i1)
 20   continue
      is = 2*id-1
      id = 4*id
      if(is.lt.n) go to 10
c
c-----------------other butterflies------------ -
c
      n2 = 2
      do 100 k=2,m
      n2 = n2*2
      n4 = n2/4
c---------------without mult-------------
      is = 1
      id = 2*n2
 30      do 40 i0=is,n-1,id
         i1 = i0+n4
         i2 = i1+n4
         i3 = i2+n4
         t0 = x(i2)+x(i3)
         x(i3) = x(i2)-x(i3)
         x(i2) = x(i0)-t0
         x(i0) = x(i0)+t0
 40      continue
      is = 2*id-n2+1
      id = 4*id
      if(is.lt.n) go to 30
      if(n4.lt.2) go to 100
c----------------with 2 real mult
      is = n4/2+1
      id = 2*n2
 50      do 60 i0=is,n-1,id
         i1 = i0+n4
         i2 = i1+n4
         i3 = i2+n4
         t1 = (x(i2)-x(i3))*rac2s2
         t2 = (x(i2)+x(i3))*rac2s2
         x(i2) = t2-x(i1)
         x(i3) = t2+x(i1)
         x(i1) = x(i0)-t1
         x(i0) = x(i0)+t1
 60      continue
      is = 2*id-n2+n4/2+1
      id = 4*id
      if(is.lt.n) go to 50
      e = 6.283185307179586/n2
      a = e
      if(n4.lt.4) go to 100
         do 90 j=2,n4/2
         a3 = 3.*a
         cc1 = cos(a)
         ss1 = sin(a)
         cc3 = cos(a3)
         ss3 = sin(a3)
         a = j*e
         is = j
         id = 2*n2
 70         do 80 ia0=is,n-1,id
c-----------------------with 6 real mult--------------
            ib1 = ia0+n4
            ia1 = ib1-j+2
            ib0 = ia1+n4
            ia2 = ib1+n4
            ia3 = ia2+n4
            ib2 = ib0+n4
            ib3 = ib2+n4
            c2 = x(ia2)*cc1-x(ib2)*ss1
            d2 = -(x(ia2)*ss1+x(ib2)*cc1)
            c3 = x(ia3)*cc3-x(ib3)*ss3
            d3 = -(x(ia3)*ss3+x(ib3)*cc3)
            t1 = c2+c3
            c3 = c2-c3
            t2 = d2-d3
            d3 = d2+d3
            x(ia2) = -x(ib0)-d3
            x(ib2) = -x(ib1)+c3
            x(ia3) = x(ib1)+c3
            x(ib3) = x(ib0)-d3
            x(ib1) = x(ia1)+t2
            x(ib0) = x(ia0)-t1
            x(ia0) = x(ia0)+t1
            x(ia1) = x(ia1)-t2
c
c
 80         continue
         is = 2*id-n2+j
         id = 4*id
         if(is.lt.n) go to 70
 90      continue
 100   continue
c
       return
       end
```

APPENDIX II
PARTS OF PROGRAM TO BE INSERTED IN THE DIT SPLIT-RADIX FFT PROGRAM TO OBTAIN AN FDHT PROGRAM

1) To be inserted after line 48.

```
      if(i0.ne.1) go to 35
      t0 = x(i2)+x(i3)
      t1 = x(i2)-x(i3)
      x(i2) = x(i0)-t0
      x(i0) = x(i0)+t0
      x(i3) = x(i1)-t1
      x(i1) = x(i1)+t1
      go to 40
35    continue
```

2) To be inserted after line 64.

```
      if(i0.ne.(n4/2+1)) go to 55
      t1 = x(i2)*rac2
      t2 = x(i3)*rac2
      x(i3) = x(i1)-t2
      x(i1) = x(i1)+t2
      x(i2) = x(i0)-t1
      x(i0) = x(i0)+t1
      go to 60
55    continue
```

3) To be inserted after line 83.

```
      cps1 = cc1+ss1
      cps3 = cc3+ss3
      cms1 = cc1-ss1
      cms3 = cc3-ss3
```

4) To be inserted after line 95.

```
      if(ia0.ne.j) go to 75
      c2 = x(ia2)*cps1+x(ib2)*cms1
      d2 = x(ia2)*cms1-x(ib2)*cps1
      c3 = x(ia3)*cps3+x(ib3)*cms3
      d3 = x(ia3)*cms3-x(ib3)*cps3
      t1 = c2+c3
      c3 = c2-c3
      t2 = d2-d3
      d3 = d2+d3
      x(ib2) = x(ia1)-c3
      x(ia1) = x(ia1)+c3
      x(ia3) = x(ib1)-t2
      x(ib1) = x(ib1)+t2
      x(ib3) = x(ib0)+d3
      x(ib0) = x(ib0)-d3
      x(ia2) = x(ia0)-t1
      x(ia0) = x(ia0)+t1
      go to 80
75    continue
```

ACKNOWLEDGMENT

The authors would like to thank the reviewers for their constructive criticisms.

REFERENCES

[1] J. W. Cooley and J. W. Tukey, "An algorithm for the machine calculation of complex Fourier series," *Math. Comput.*, vol. 19, pp. 297-301, 1965.

[2] R. D. Preuss, "Very fast computation of the radix 2 discrete Fourier transform," *IEEE Trans. Acoust., Speech, Signal Processing*, vol. ASSP-30, pp. 595-607, 1982.

[3] G. D. Bergland, "A fast Fourier transform algorithm for real-valued series," *Commun. ACM*, vol. 11, pp. 703-710, Oct. 1968.

[4] H. Ziegler, "A fast Fourier transform algorithm for symmetric real-valued series," *IEEE Trans. Audio Electroacoust.*, vol. AU-20, pp. 353-356, Dec. 1972.

[5] J. B. Martens, "Discrete Fourier transform algorithms for real-valued sequences," *IEEE Trans. Acoust., Speech, Signal Processing*, vol. ASSP-32, pp. 390-396, Apr. 1984.

[6] M. Vetterli and H. J. Nussbaumer, "Simple FFT and DCT algorithms with reduced number of operations," *Signal Processing*, vol. 6, pp. 267-278, July 1984.

[7] P. Duhamel, "Implementation of split-radix FFT algorithms for complex, real, and real-symmetric data," *IEEE Trans. Acoust., Speech, Signal Processing*, vol. ASSP-34, pp. 285-295, Apr. 1986.

[8] R. N. Bracewell, "The fast Hartley transform," *Proc. IEEE*, vol. 22, pp. 1010-1018, Aug. 1984.

[9] H. V. Sorensen, D. L. Jones, C. S. Burrus, and M. T. Heideman, "On computing the discrete Hartley transform," *IEEE Trans. Acoust., Speech, Signal Processing*, vol. ASSP-33, pp. 1231-1238, Oct. 1985.

[10] P. Duhamel, "Un algorithme de transformation de Fourier rapide à double base," *Ann. Télécommun.*, vol. 40, pp. 481-494, Sept.-Oct. 1985.

[11] M. Vetterli and H. J. Nussbaumer, "Algorithmes de transformation de Fourier et en cosinus mono et bi-dimensionnels," *Ann. Télécommun.*, vol. 40, pp. 466-476, Sept.-Oct. 1985.

[12] H. V. Sorensen, M. T. Heideman, and C. S. Burrus, "On computing the split-radix FFT," *IEEE Trans. Acoust., Speech, Signal Processing*, vol. ASSP-34, pp. 152-156, Feb. 1986.

[13] R. C. Agarwal and C. S. Burrus, "Fast convolution using Fermat number transform with application to digital filtering," *IEEE Trans. Acoust., Speech, Signal Processing*, vol. ASSP-22, pp. 87-97, Apr. 1974.

[14] R. Ansari, "An extension of the discrete Fourier transform," *IEEE Trans. Circuits Syst.*, vol. CAS-32, pp. 618-619, June 1985.

[15] S.-C. Pei and J.-L. Wu, "Split-radix fast Hartley transform," *Electron. Lett.*, vol. 22, pp. 26-27, Jan. 1986.

[16] L. R. Morris, "Automatic generation of time efficient digital signal processing software," *IEEE Trans. Acoust., Speech, Signal Processing*, vol. ASSP-25, pp. 74-79, Feb. 1977.

Pierre Duhamel (M'87) was born in France in 1953. He received the Ingenieur degree in electrical engineering from the National Institute for Applied Sciences, Rennes, France, in 1975, the Dr. Ing. degree in 1978, and the Doctorat es Sciences in 1986 from Orsay University, France.

From 1975 to 1980 he was with Thomson-CSF, Paris, France, where his research interests were in circuit theory and signal processing, including digital filtering and automatic analog fault diagnosis. In 1980 he joined the National Research Center in Telecommunications (CNET), Issy-les-Moulineaux, France, where his activities were first concerned with the design of recursive CCD filters. He is now working on fast convolution algorithms, including number theoretic transforms and fast Fourier transforms.

Martin Vetterli (M'87) for a photograph and biography, see p. 372 of the March 1987 issue of this TRANSACTIONS.

On Computing the Inverse DFT

P. Duhamel
B. Piron
J. M. Etcheto

Reprinted from
IEEE TRANSACTIONS ON ACOUSTICS, SPEECH, AND SIGNAL PROCESSING
Vol. 36, No. 2, February 1988

Correspondence

On Computing the Inverse DFT

P. DUHAMEL, B. PIRON, AND J. M. ETCHETO

Abstract—**This correspondence indicates a (possibly) new method for computing an inverse discrete Fourier transform (IDFT) through the use of a forward DFT program.**

We point out that, in many cases, this is obtained without any additional cost, either in terms of program length or in terms of computational time.

I. Forward and Inverse DFT [1]-[3]

The forward DFT is usually defined as follows:

$$X_k = \sum_{n=0}^{N-1} x_n W_N^{nk} = \mathrm{DFT}_k \{x_n\} \tag{1}$$

with

$$W_N \triangleq \exp\left(-j\frac{2\pi}{N}\right)$$

and the inverse DFT has nearly the same expression, except for a normalizing factor $1/N$ and a conjugation of the Nth root of unity:

$$x_n = \frac{1}{N}\sum_{k=0}^{N-1} X_k W_N^{-nk}. \tag{2}$$

As we are very often interested only in relative values of the sequence, the scaling factor is generally omitted:

$$x'_n = \sum_{k=0}^{N-1} X_k W_N^{-nk} = \mathrm{IDFT}_n \{X_k\} \tag{3}$$

or sometimes distributed between the two expressions to increase their symmetry:

$$\begin{cases} X'_k = \dfrac{1}{\sqrt{N}}\displaystyle\sum_{n=0}^{N-1} x_n W_N^{nk} \\ x_n = \dfrac{1}{\sqrt{N}}\displaystyle\sum_{k=0}^{N-1} X'_k W_N^{-nk}. \end{cases} \tag{4}$$

This correspondence first recalls the usual algorithms used to compute the IDFT, and then proposes a new one, which turns out to have some advantages compared to the classical ones.

II. Usual Algorithms for Computing the IDFT [1]-[3]

Since both expressions (1) and (3) for the forward and inverse DFT are nearly symmetrical, a trivial solution for computing the IDFT is to conjugate every multiplying constant (including multiplications by j in radix-4 or split-radix programs [3], [4]) inside the forward FFT algorithm. This takes into account the conjugation of W_N in (3). A multiplication by $1/N$ may be needed to obtain (2) if necessary.

But this solution requires two different subprograms for computing the forward and inverse DFT. Therefore, when the scaling factor $1/N$ is omitted or when (4) is used, it is often preferred to use the forward DFT as an intermediate step.

Manuscript received November 1, 1986; revised August 24, 1987.

The authors are with CNET/PAB/RPE, 92131 Issy-les-Moulineaux, France.

IEEE Log Number 8718023.

In fact, it is easily seen that

$$\mathrm{IDFT}_n \{X_k\} = \sum_{k=0}^{N-1} X_k W_N^{-nk} = \mathrm{DFT}_{-n} \{X_k\} \tag{5}$$

which means that the unscaled inverse DFT of $\{X_k\}$ is equivalent to a forward DFT followed by a permutation:

$$x'_n = x''_{(-n)_N}. \tag{6}$$

This permutation has to be performed after FFT computation, which is time consuming or has to be included in the subsequent calculation, which gives rather intricate programs.

III. The New Algorithm

The new method is based on the remark that conjugating x'_n in (3) gives an expression which is already in the form of a DFT:

$$x'^*_n = \sum_{k=0}^{N-1} X_k^* W_N^{nk}. \tag{7}$$

A second remark is that

$$x_n = a_n + j \cdot b_n \Rightarrow j \cdot x_n^* = b_n + j \cdot a_n. \tag{8}$$

Hence, if we multiply (7) by j, we get

$$j \cdot x'^*_n = \sum_{k=0}^{N-1} j \cdot X_k^* W_N^{nk} \tag{9}$$

and

$$x'_n = j\left[\sum_{k=0}^{N-1} (jX_k^*)\, W_N^{nk}\right]^*. \tag{10}$$

In other words, (10) means that the unscaled IDFT of a sequence can be obtained by the following procedure:

- exchange the real and imaginary parts of the initial sequence,
- perform a forward DFT,
- exchange the real and imaginary parts of the result.

At first glance, the above procedure seems to be more complicated than the usual one, but we shall see that in many circumstances, this procedure is absolutely costless when compared to a forward FFT.

In fact, FFT programs are generally written in real rather than complex arithmetic (to increase execution speed). Hence, if written in Fortran, FFT subroutines are generally of the type

SUBROUTINE FFT (*XR*, *XI*, *N*)

where *XR* (respectively, *XI*) is the real (respectively, imaginary) part of the sequence to be transformed on input, and of the result on output.

In such cases, an unscaled IDFT can be obtained by exchanging the real and imaginary parts in the call of the subroutine.

If the IDFT of the sequence $YR(I) + j \cdot YI(I)$ is to be computed, (10) tells us that it will be obtained in the following manner:

CALL FFT (*YI*, *YR*, *N*)

which will perform the three steps described above.

It is easily seen that the same kind of procedure will provide the desired IDFT in any of the programming languages for which subprogram's arguments are transmitted by means of their address.

Let us also point out that, obviously, this method also holds for the more symmetrical definition of the DFT and IDFT given in (4) as well as for multidimensional DFT's and IDFT's.

REFERENCES

[1] L. R. Rabiner and B. Gold, *Theory and Application of Digital Signal Processing*. Englewood Cliffs, NJ: Prentice-Hall, 1975.
[2] E. O. Brigham, *The Fast Fourier Transform*. Englewood Cliffs, NJ: Prentice-Hall, 1974.
[3] C. S. Burrus and T. W. Parks, *DFT/FFT and Convolution Algorithms*. New York: Wiley, 1985.
[4] P. Duhamel, "Implementation of split-radix FFT algorithms for complex, real, and real-symmetric data," *IEEE Trans. Acoust., Speech, Signal Processing*, vol. ASSP-34, pp. 285–295, Apr. 1986.

0096-3518/88/0200-0286$01.00 © 1988 IEEE

A Better FFT Bit-Reversal Algorithm Without Tables

Angelo A. Yong

Abstract—**The bit-reversal counteralgorithm of Gold and Rader bit reverses a continuous sequence of N numbers by running a loop $N-1$ times. The heuristic approach presented here repeats a similar loop only $N/4$ times.**

I. Introduction

The need for an efficient bit-reversal algorithm for the fast Fourier and Hartley transforms, using a radix-2, 4-2, 8-4-2, or the split-radix $2/4$, is well known.

In a recent paper [4] on this subject, its author obtains a small improvement of the bit-reversal counteralgorithm (BRCA), Fig. 1, by reducing the upper limit of the counter from $N-2$ to $N-1-$ square root (rN), where $N = 2^E$ and "r" equals 1 for an even E or 2 if E is odd; the relative savings reported for $N = 128$ is 8.5%. In part II, a faster BRCA is proposed. The working storage of the BRCA does not depend on N and it may become insignificant (see [4, introduction]) when compared to the requirements of the algorithms that make use of a seed table [2], [3], [5] on the order of the square root of N.

II. An Efficient Bit-Reversal Procedure

The initial step, to improve the algorithm, comes from the observation that given a couple of numbers (I, J), where I is an even

Manuscript received August 11, 1990; revised April 17, 1991.

The author is with the Department of Electronics, Universidad Simon Bolivar, Caracas, Venezuela.

IEEE Log Number 9101869.

```
N2= N / 2
J = 0
for I = 0 to N-2
   if (I<J) swap(I,J)
   K = N2
    while (K <= J)
       J = J - K
       K = K / 2
   endwhile
   J = J + K
endfor
```

Fig. 1. Bit-reversal counteralgorithm.

integer and J its bit reversal, then $J + N/2$ is the bit reversal of $I + 1$. Constructing a table for a particular case, $N = 16$ for example, is shown below:

I	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
J	0	8	4	12	2	10	6	14	1	9	5	13	3	11	7	15

Reordering the table by taking two groups $N/2$ apart, is as follows:

I	0	1	8	9	2	3	10	11	4	5	12	13	6	7	14	15
J	0	8	1	9	4	12	5	13	2	10	3	11	6	14	7	15
		S			S	S		S		S				S		

where S in the last row indicates that the elements of an array indexed by I and J are to be swapped considering that $I < J$. As it can be observed, the second couple is always swapped, whereas the first and fourth ones depend on $I < J$, and the third one is left over. Given the first couple (I, J) of a group, the next three are: $(I + 1, J + N/2)$, $(I + N/2, J + 1)$, and $(I + 1 + N/2, J + 1 + N/2)$.

The reasoning behind this choice is that given any even $I < N/2$ and J as its bit reversal, then J is even since the most significant bit of I is zero; therefore, the bit reversal of $J + 1$ will be $I + N/2$. Here, $I + N/2$ is again an even integer and consequently the bit reversal of $I + N/2 + 1$ must be $J + 1 + N/2$. Since the couple $(I + 1 + N/2, J + 1 + N/2)$ is formed by adding a constant to (I, J), it is enough to check if I is less than J to do their swaps; $I + 1$ is definitely less than $J + N/2$ since the integer I was chosen as even and less than $N/2$. The binary representation of $I + N/2$ has a one at its leftmost position, due to the $N/2$ term, and a zero in its least significant bit since $I + N/2$ is even ; it is clear then that $I + N/2$ will always be greater than $J + 1$ and so this couple can be omitted from the swapping procedure.

An implementation of the algorithm is shown in Fig. 2. Since the bit reversals of only the first $N/2$ even integers are required, the main loop repeats only $N/4$ times. A further saving, in computational terms, is due to the fact that K, before entering the while statement, starts at $N/4$ instead of $N/2$ to bypass the bit reversal of odd integers.

A graph of the execution times normalized by N against E for the three BRCA discussed in this correspondence and for Walker's

```
N2 = N / 2
N4 = N / 4
N21 = N2 + 1
J = 0
for I = 0 to N2-2 step 2
   if (I<J)
      swap(I,J)
      swap(I+N21,J+N21)
   endif
   swap(I+1,J+N2)
   K = N4
   while (K <= J)
      J = J - K
      K = K / 2
   endwhile
   J = J + K
endfor
```

Fig. 2. Improved bit-reversal algorithm.

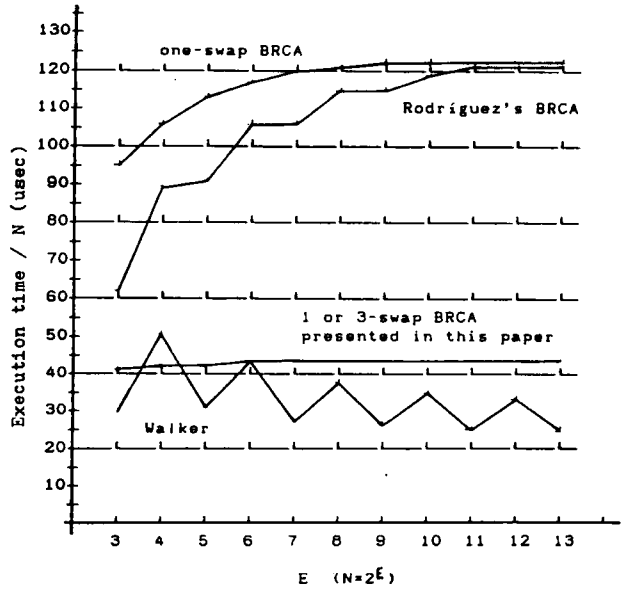


Fig. 3. Execution times/N of three algorithms of the bit-reversal counter types and for Walker's algorithm which uses a seed table.

algorithm [5] is presented in Fig. 3. Although the time values are machine and compiler dependent, the relative positions of the curves show that the speed of the algorithm proposed here comes closer to that of Walker's.

The programs were written in Pascal and tested on a 4.77-MHz personal computer without the swapping operations to actually compare the times the algorithms take to determine the array indexes for swapping.

ACKNOWLEDGMENT

The paper by Polge *et al.* [2] was introduced to the author by one of the reviewers.

REFERENCES

[1] B. Gold and C. M. Rader, *Digital Processing of Signals*. New York: McGraw-Hill, 1969 (reprinted 1983).

[2] R.J. Polge, B. K. Bhagavan, and J. M. Carswell, "Fast computational algorithms for bit reversal," *IEEE Trans. Comput.*, vol. C-23, no. 1, pp. 1-9, Jan. 1974.

[3] D. M. W. Evans, "An improved digit-reversal permutation algorithm for the fast Fourier and Hartley transforms," *IEEE Trans. Acoust., Speech, Signal Processing*, vol. ASSP-35, pp. 1120-1125, Aug. 1987.

[4] J. J. Rodriguez, "An improved FFT digit-reversal algorithm," *IEEE Trans. Acoust., Speech, Signal Processing*, vol. 37, pp. 1298-1300, Aug. 1989.

[5] J. S. Walker, "A new bit reversal algorithm," *IEEE Trans. Acoust., Speech, Signal Processing*, vol. 38, pp. 1472-1473, Aug. 1990.