配布先限定

NAL TM-369

UDC 681.31

# 航空宇宙技術研究所資料

TECHNICAL MEMORANDUM OF NATIONAL AEROSPACE LABORATORY

TM-369

## ＦＡＣＯＭ230-75アレイプロセッサーについて　II
## ―処理能力の実測とプログラム技術―

末松俊二・中村　孝・石塚只夫
福田正大・吉田正広・三好　甫

1978 年 10 月

航空宇宙技術研究所
NATIONAL AEROSPACE LABORATORY

This document is provided by JAXA.

# FACOM230−75アレイプロセッサーについてⅡ*

## —処理能力の実測とプログラム技術—

末松俊二　石塚只夫　吉田正広 **
中村　孝　福田正大　三好　甫

### 1. 緒　言

我が国初の大型アレイプロセッサであるFACOM 230−75アレイプロセッサ（以下単にAPと略記する）は昭和52年8月航技研センターに導入され仮運用を開始したが，53年4月より本運用のはこびとなった。仮運用期間中に航技研，富士通双方により種々のプログラムがAPで実行され，FACOM230−75との処理能力の比較，単体命令での理論的予測処理能力と実際処理能力の比較，運用方式，オペレーティングシステムの問題点とその解決方策の検討がなされてきた。

本資料はAPの命令単体における実際の処理速度および種々の問題をAPで実行させた場合のFACOM 230−75（以下単にCPと略記する）との処理能力の比較，およびこの比較を通じて大きな問題となったAPの能力を十分に発揮させるために必要であると考えられるプログラム技術に関するものである。

### 2. APの機械命令の実行速度に関する理論値と実測値

#### 2.1 AP機械命令の実行速度に関する理論値

以前[1)]にAPのM型，R型命令，V型命令の固有の実行速度に関する一覧表を示した。V型命令に関してはデータの供給能力，他装置との主記憶競合およびパイプラインの立ちあがり時間，後処理時間を考慮した実際の実行速度の算出式とその算出式に現われる種々の定数が示され，それに基ずいてベクトル一要素あたりの各種V型命令の実行時間を縦軸に，ベクトル長を横軸にとり，実行時間の単位をAPのクロック時間（$T_u$=90ns）[注1)]を単位として図示した。M型，R型命令に関してはバッファメモリにおけるnot found probabilityとレジスタ競合を考慮して平均命令実行時間の算出式とそれに基ずく平均命令実行時間を表示した。

52年7月AP製作終了後これらの数値，実行時間の算出式について幾度か富士通側より訂正があり，その訂正は全てAPの処理能力の低下につながるものであったため航技研計算センタは改善要求を行ない，その結果本年1月にAPのハードウェアに関する最終性能が確定した。そこでその最終形に基ずきAP機械命令の実行速度理論値の算出を行なった。先ずAP機械命令の固有の実行速度 $T_i$ の一覧表を表2.1.1，2.1.2に示す。表中アンダーラインのある部分は以前の資料[1)]と異なる部分であり，その後の括弧の部分は以前の資料[1)]における数字である。実行速度の単位はクロック時間 $T_u$ である。またV型命令の固有の実行時間は $T_i=\lambda N+\mu$ の形をしているが，ここで$N$はベクトル長を表わしている。

V型命令として表2.1.2にあるものの他にGAT，SCAT，FMK命令が追加になっている。

次にV型命令の実際の実行速度の算出式と算出式に現われる定数を示す。ここでも以前の資料[1)]から変更になった部分はアンダーラインで示し，そのあとの括弧の中は以前の資料[1)]のものである。V型命令の実際の実行速度を$T_e$ とすると

$$T_e=(T_I+T_M+\underline{T_P'}(T_P))\cdot T_u \quad ;T_u \text{ はクロック時間}$$

$$T_M\begin{cases} =(1+\beta)\left[\left\{(m-n)d+\dfrac{P}{11}\right\}/\left(\dfrac{W}{11}\right)\right]\lambda N \\ \qquad\qquad -\lambda N \; ; [\;]>1 \\ =\beta\lambda N \qquad ; [\;]\leq 1 \end{cases} \qquad (2.1.1)$$

$$T_P'=T_P+\underline{T_{Pd}+T_{PW}+T_{PB}+T_{PO}+T_{PS}+T_{PM}}(0) \qquad (2.1.2.)$$

（2.1.1），（2.1.2）式において$N$はベクトル長，$\lambda$は $T_i$ の表における$\lambda$，$n$はベクトルレジスタ（以下VRと略記する）にあるオペランドの数，例えば $A_i=B_i+C_i$ の場合，オペランドベクトル$A_i,B_i,C_i$のうち2つがVR上にあれば$n=2$である。$m$はベクトル演算に必要なベクトルオペランドの数，$A_i=B_i+C_i$の場合は $m=3$，$S=\Sigma A_i\times B_i$ の場合は $m=2$，

＊　昭和53年9月6日受付
＊＊計算センター
注1）ns；ナノセカンドの略，$10^{-9}$ 秒

This document is provided by JAXA.

表2.1.1

1クロック＝1 $T_{\mu}$＝90 ns

| M型命令 | | R型命令 | |
|---|---|---|---|
| 命令名称 | 実行速度クロック数（$T_I$） | 命令名称 | 実行速度クロック数（$T_I$） |
| Load | 2 | Add Register Fixed Point | 1 |
| Load Immediate | 1 | Add Register Floating Point Single | 8 |
| Store | 2 | Add Register Floating Point Double | 8 |
| Add | 2 | Add Register Floating Point Quadruple | 12(13) |
| Add Immediate | 1 | Subtract Register Fixed Point | 1 |
| Subtract | 2 | Subtract Register Floating Point Single | 8 |
| Subtract Immediate | 1 | Subtract Register Floating Point Double | 8 |
| Multiply | 8(7) | Subtract Register Floating Point Quadruple | 12(13) |
| Shift Right Logical | 1 | Multiply Register Fixed Point | 7 |
| Shift Left Logical | 1 | Multiply Register Floating Point Single | 9 |
| Shift Right Arithmetic | 1 | Multiply Register Floating Point Double | 12 |
| Shift Left Arithmetic | 1 | Multiply Register Floating Point Quadruple | 19(20) |
| Shift Right Circular | 1 | Divide Register Fixed Point | 44 |
| Shift Left Circular | 1 | Divide Register Floating Point Single | 21(23) |
| Branch on Condition | 3(4)成立<br>4(5)不成立 | Divide Register Floating Point Double | 30 |
| Branch and Link | 3(4) | Divide Register Floating Point Quadruple | 80(79) |
| Branch on Count | 3(4)成立<br>4(5)不成立 | Transfer Register | 1 |
| Set Bit | 1 | Masked Transfer Register | 2 |
| Reset Bit | 1 | Load Multiple Registers | 2＋2$m$ |
| Set Bit of Memory | 4 | Store Multiple Registers | 2＋2$m$ |
| Reset Bit of Memory | 4 | Supervisor Call | 238(250) |
| Test Bit of Memory | 4 | Supervisor Release | 85(70) |
| Load Condition Code and Set | 16 | Call CPU | 10(8) |
| | | Load Data Base Register | 5 |
| | | Restor Vector Register | 920(900) |
| | | Save Vector Register | 920(900) |
| | | Load Program Base Register | 9(8) |
| | | Buffer Memory Clear | 1038(130) |
| | | APU No Operation | 1 |
| | | Halt APU | 5(30) |
| | | Clear Watch dog Timer | 5 |

This document is provided by JAXA.

表2.1.2

| V 型 命 令 | | | | 単位 $1T_*=90$ns<br>$N$はオペランド長 |
|---|---|---|---|---|
| 命 令 名 称 | 省略型 | 単精度（$T_I$） | 倍精度（$T_I$） | 4倍精度（$T_I$） |
| Vector Move | VMV* | $\frac{1}{2}N+8$ | $N+8$ | $4N+9$ (8) |
| Vector Add | VAD | $\frac{1}{2}N+8$ | $N+8$ | $4N+9$ (8) |
| Vector Multiply | VML | $N+9$ | $2N+11$ | $8N+12$ (10) |
| Vector Divide | VDV | $9N+10$* | $16N+12$* | $63N+18$ (20) |
| Average | AVG | $\frac{1}{2}N+8$ | $N+8$ | $4N+9$ (8) |
| Vector Sum | VSM* | $\frac{1}{2}N+38$ (31) | $N+28$ (24) | $4N+25$ (20) |
| Vector Norm | VNM* | $N+31$ | $2N+27$ | $8N+20$ (19) |
| Inner Product | IPD* | $N+31$ | $2N+27$ (26) | $8N+20$ (19) |
| Adjacent Mean | AJM | $\frac{1}{2}N+8$ | $N+8$ | $4N+9$ (8) |
| Find Address MAX | FAX | $2N+39$ (48) | $2N+24$ (30) | $8N+10$ (25) |
| Find Address MIN | FAN | $2N+39$ (48) | $2N+24$ (30) | $8N+10$ (25) |
| Find Address MIN Positive | FAP | $2N+39$ (48) | $2N+24$ (30) | $8N+10$ (25) |
| Find Relation Equal | FRE | $2N+13$ (30) | $2N+13$ (30) | $4N+14$ (30) |
| Find Relation Not Equal | FRNE | $2N+13$ (30) | $2N+13$ (30) | $4N+14$ (30) |
| Find Relation Low | FRL | $2N+13$ (30) | $2N+13$ (30) | $4N+14$ (30) |
| Find Relation High or Equal | FRHE | $2N+13$ (30) | $2N+13$ (30) | $4N+14$ (30) |
| Compare Equal | CME* | $1.5N+20+2W$ | $2N+20+2W$ | $4N+20+2W$ |
| Compare Low | CML* | $1.5N+20+2W$ | $2N+20+2W$ | $4N+20+2W$ |
| Convolving Add | CVA | $(m+27)N+6$ (8) | $(2m+20)N+6$ (8) | $(8m+10)N+8$ |
| Convolving Multiply | CVM | $(m+20)N+13$ (12) | $(2m+12)N+13$ (15) | $8mN+22$ (20) |
| Partial Matrix Multiply | PMM | $(m+20)N+13$ (12) | $(2m+13)N+13$ (15) | $8mN+22$ (20) |
| Polynomial | PLY | $(14m+6)\left(\left[\frac{N-1}{4}\right]+1\right)+5$ | $(17m-1)\left(\left[\frac{N-1}{2}\right]+1\right)+5$ | $(33m-19)N+6$ |
| Scalar Multiply and Vector Add | SMVA | $3.5N+18$ (14) | $5N+19$ (17) | $10$ (8) $N+32$ (21) |
| Vector Move with Distance | VMD | $N+8$ | $N+8$ | $4N+8$ |
| Vector Masked Move | VMM | $N+\alpha$* | $N+\alpha$* | $4N+\beta$* |
| Logical Sum | LSM | $2(N/2^5+1)+14$ (5) | | |
| Logical Product | LPR | $2(N/2^5+1)+14$ (5) | | |
| Logical Difference | LDF | $2(N/2^5+1)+14$ (5) | | |
| One's Count | ONC | $4W+2P+10$ $W$=Bitsの語数<br>$P$="1" Bitの数 | | |

*表2.1.2においてVMVを固定小数点⇄浮動小数点変換に用いる場合の$T_i$は浮動小数点単精度を$S$，倍精度を$D$，4倍精度を$Q$で，固定小数点を$F$と表わせば，$F\to S$；$N+8$，$F\to D$；$N+8$，$F\to Q$；$2N+8$，$S\to F$；$N+11$，$D\to F$；$2N+11$，$Q\to F$；$4N+17$となる。FDV命令は実際はもう少し複雑な式である。CME，CML命令のWは出力語数である。すなわち，32bit毎に1語である。IPD，VNM，VSM命令の$T_i$はベクトル要素数が10以下の程度では一定値，ほぼ40程度である。VMM命令の実行速度中の$\alpha$，$\beta$は複雑な形になっており，GAT命令とともにその実行速度については後に述べる。

This document is provided by JAXA.

$d$は1つのベクトルに対する1$T_u$当りのデータ要求量，例えば $A_i=B_i+C_i$ は単精度の場合 $\lambda=0.5T_u$ であり，ベクトルは単精度であるので0.5DW（ダブルワード）となり，$d=0.5\mathrm{DW}/0.5T_u=1\cdot\mathrm{DW}/T_u$ だから $d=1$ である。$P=1.5$，$W$はインターリーブ数，$\beta$は待ち時間係数で，〔　〕＞1の場合0.25，〔　〕≦1の場合$N$＝5，10，15，20，30で0.2，0.15，0.1，0.05，0.0程度に見込めば良い。$T_P$はパイプラインの立ちあがり時間，$T_{Pd}$はディスクリプタ競合による待ち時間，$T_{PW}$はディスクリプタ語数による遅れ時間，$T_{PB}$はデータバッファ制御による遅れ時間，$T_{PO}$はオペランドベクトルが奇数番地から始まる場合に生ずる遅れ時間，$T_{PS}$はV型命令の前にSTORE命令がある場合，V型命令のオペランドが主記憶上にあれば（VR上に全部ある場合は除く）生ずる遅れ時間，$T_{PM}$はオペランド競合（2つのオペランドのメモリ競合）による遅れ時間である。

（2.1.1）式において $T_M=\gamma\lambda N$ とおくと

$$\gamma\begin{cases}=(1+\beta)\left[\left\{(m-n)d+\dfrac{P}{11}\right\}/\left(\dfrac{W}{11}\right)\right]-1 & ;[\quad]>1\\ =\beta & ;[\quad]\le 1\end{cases}\qquad(2.1.3)$$

すると $T_i=\lambda N+\mu$ と $T_e$ の式より

$$T_e=\{(1+\gamma)\lambda N+\mu+T_P'\}\cdot T_u\qquad(2.1.4)$$

従ってベクトル一要素当りの演算時間は

$$\widetilde{T}_e=T_e/N=\{(1+\gamma)\lambda+(\mu+T_P')/N\}\cdot T_u\qquad(2.1.5)$$

となる。ここで（2.1.2）式の$T_P$の値も以前の資料[1]に対して修正があり，それを表2.1.3に示す。

表2.1.3

| 命令の続き方 | ベクトルレジスタ | $T_P$(単位$T_u$) |
|---|---|---|
| M，R－V<br>V－V | 全オペランドがベクトルレジスタにある | 27（23） |
| | オペランドのうちベクトルレジスタにないものがある | 35（34） |
| V－M，R | 全オペランドがベクトルレジスタにある | 21（16） |
| | オペランドのうちベクトルレジスタにないものがある | 23（16） |

（2.1.2）式右辺の $T_P$を除いた部分，すなわち$T_{Pd}$，$T_{PW}$，$T_{PB}$，$T_{PO}$，$T_{PS}$，$T_{PM}$は$T_{PW}$を除いて全てV型命令を実行する際に必要なデータのメモリ，およびデータレジスタにおけるハードウェア上の位置に関係しており，ほぼランダムであると考えなければならない。その期待値を理論的に求めることはできないが，もし，求めることができてもいくつかの仮定をおかなければならず，余り意味がないと考えられるので値の最大値または範囲を示すことにする。

$$T_{Pd}\le 2\cdot T_u\qquad(2.1.6)$$

$$T_{PB}=-1T_u,\ 0T_u,\ +1T_u\qquad(2.1.7)$$

$$T_{PO}\le 1\cdot T_u\qquad(2.1.8)$$

$T_{PS}\le 12\cdot T_u$　（STORE先がVRであれば遅れはない）　(2.1.9)

$T_{PM}\le 11\cdot T_u$　（全てのオペランドがVR上にあれば最大1・$T_u$）　(2.1.10)

$P_{PW}=2\cdot T_u$　（リストベクトルの場合）
　$=2\cdot T_u$　（ビットストリングの場合）
　$=10\cdot T_u$　（FMK，GAT，SCAT命令の場合）
　$=2\cdot T_u$　（SMVA命令の場合）
　$=-2\cdot T_u$　（スカラーの場合）　(2.1.11)

$T_{PS}$，$T_{PM}$，$T_{PW}$はFMK，GAT，SCAT命令の場合に遅れは非常に大きくなる可能性があるが，$T_{PS}$はV型命令の前のSTORE命令の書き込み先がVRであれば無視して良いし，書き込み生が主記憶にある場合，STORE命令とV型命令の間に他のM型，R型命令があり，その実行時間がX・$T_u$ であればその遅れは$T_{PS}$＝（12－X）・$T_u$となる。$T_{PM}$はV型命令のオペランドの先頭番地の主記憶バンク間の距離をXバンクとするとき$T_{PM}$＝（11－X）・$T_u$ となるので，その期待値は11よりはるかに小さく5$T_u$以下であろう。FMK，GAT，SCAT命令の場合の遅れは命令固有のものであって，利用者は制御できない。（2.1.4）式の$T_P'$を$T_P$で置きかえて計算した理論値と種々の実測データとの比較から$T_P'$の$T_P$以外の期待値は，FMK，GAT，SCAT命令を除き5～7・$T_u$（V型命令の全オペランドがVR上にあれば1～2・$T_u$）程度であろう。$T_P$の修正と上に述べた5～7・$T_u$（1～2・$T_u$）の遅れの和をとると全体で11～13・$T_u$（10～11・$T_u$）程度の遅れとなり小さなベクトルをあつかう時の性能の低下は可成りなものとなる。修正された定数と算出式を使い，$T_P'$の$T_P$以外の遅れを6・$T_u$（1・$T_u$）として計算した理論的実行速度を，VAD，VML命令について以前の資料[1]のデータと比較したものが図2.1.1および図2.1.2である。他の命令に関しても傾向は同様であってベクトル長の小さいもの程影響が大きく遅くなっている。新理論値の旧理論値からのベクトル一要素当りの実行速度の遅れはベクトル長を$N$とした時，$T_i$ が変わらない場合は全てのオペランドが主記憶にある場合に大

This document is provided by JAXA.

VAD命令実行速度／VECTOR 1 要素

V-V SEQ. Single　32 way

| | | |
|---|---|---|
| A | all data in memory | 新理論値 |
| B | 〃 | 旧理論値 |
| C | all data in VR | 新理論値 |
| D | 〃 | 旧理論値 |

$T_u$

ベクトル長

図2.1.1

VML命令実行速度／VECTOR 1 要素

V-V SEQ. Single

| | | |
|---|---|---|
| A | all data in memory | 新理論値 |
| B | 〃 | 旧理論値 |
| C | all data in VR | 新理論値 |
| D | 〃 | 旧理論値 |

$T_u$

ベクトル長

図2.1.2

体12／$N \cdot T_u$，全てのオペランドがVRにある場合には10／$N \cdot T_u$ 程度である。

M型，R型命令の平均命令実行時間の算出方式は旧資料[1)]の場合と同じであるがコンパイラの作りだすオブジェクトプログラムの検討と実測データの比較から全ての場合にレジスタのバンク競合，前後の命令におけるレジスタ競合（フローティングスカラ演算を除く）があると仮定すべきであること，およびインデックス修飾を行う

This document is provided by JAXA.

表 2.1.4

| 精度 | 230−75−CPおよびAP M&R型命令平均命令実行時間(単位ns)NFP 5% | | | | |
|---|---|---|---|---|---|
| | 75−CP | AP(M&R) | AP(M&R)<br>フローティング演算50%減 | AP(M&R)<br>フローティング演算70%減 | AP(M&R)<br>フローティング演算90%減 |
| 単精度 | 272 | 463(400) | 416(354) | 397(339) | 377(319) |
| 倍精度 | 294 | 491(421) | 430(365) | 406(343) | 380(321) |
| 4倍精度 | 566 | 624(553) | 496(431) | 445(382) | 393(334) |

場合にインデックスレジスタ読み出しを先行制御により吸収できなかったことを考慮して，計算しなおした結果を表 2.1.4 に示す。表において括弧中は以前の資料[1]の値である。表から明らかな様に全ての場合について約60 ns 程度，以前の資料[1]より遅くなっている。またV型命令を全く使用せずにプログラムを書き，AP−FORTRANでコンパイル，実行させればそのプログラムのAP実行時間は平均的にCPでの実行時間の約1.7倍かかることになることを意味する。従って中には2倍程度かかるものもあるであろう。

## 2.2 AP機械命令および配列特殊関数の実行速度の実測値

2.1 節で述べた様にAPのV型命令の実行速度は，オペランドの主記憶上の位置とか他装置とのメモリ競合，ディスクリプタ競合といったランダムな要因に左右され

表 2.2.1

単位 $T_\mu$

| 命令 | 要素数 | 単精度 all data in memory 新理論値 | 単精度 all data in memory 実測値 | 単精度 all data in VR 新理論値 | 単精度 all data in VR 実測値 | 倍精度 all data in memory 新理論値 | 倍精度 all data in memory 実測値 |
|---|---|---|---|---|---|---|---|
| VAD | 5 | 14.87 | 16.4 | 12.0 | 11.6 | 15.58 | 16.22 |
| | 20 | 4.22 | 4.38 | 3.38 | 3.27 | 4.90 | 5.06 |
| | 50 | 2.09 | 2.14 | 1.64 | 1.56 | 2.77 | 2.89 |
| | 500 | 0.81 | 0.66 | 0.61 | 0.57 | 1.49 | 1.22 |
| VML | 5 | 15.6 | 16.4 | 12.8 | 12.0 | 17.2 | 17.1 |
| | 20 | 4.65 | 4.87 | 3.95 | 3.84 | 5.80 | 5.81 |
| | 50 | 2.44 | 2.54 | 2.16 | 2.10 | 3.48 | 3.49 |
| | 500 | 1.14 | 1.15 | 1.12 | 1.10 | 2.15 | 2.15 |
| VDV | 5 | 25.4 | 24.4 | 22.60 | 20.4 | 34.2 | 31.22 |
| | 20 | 13.1 | 12.25 | 12.40 | 11.22 | 20.55 | 19.04 |
| | 50 | 10.46 | 9.96 | 10.18 | 9.56 | 17.5 | 16.9 |
| | 500 | 9.15 | 8.61 | 9.12 | 8.57 | 16.15 | 15.53 |
| VMV | 5 | 14.8 | 14.0 | 12.0 | 10.8 | 15.4 | 13.4 |
| | 20 | 4.08 | 3.8 | 3.38 | 3.09 | 4.6 | 4.1 |
| | 50 | 1.92 | 1.88 | 1.64 | 1.56 | 2.42 | 2.40 |
| | 500 | 0.64 | 0.61 | 0.61 | 0.57 | 1.14 | 1.15 |
| VSM | 5 | 19.4 | 22.84 | 16.6 | 20.18 | 18.6 | 19.8 |
| | 20 | 5.23 | 5.78 | 4.53 | 5.15 | 5.4 | 5.6 |
| | 50 | 2.38 | 2.61 | 2.10 | 2.35 | 2.74 | 2.84 |
| | 500 | 0.69 | 0.68 | 0.66 | 0.68 | 1.17 | 1.18 |
| IPD | 5 | 20.0 | 24.8 | 17.2 | 20.62 | 20.2 | 23.2 |
| | 20 | 5.75 | 6.71 | 5.05 | 5.65 | 6.55 | 7.22 |
| | 50 | 2.88 | 3.28 | 2.60 | 2.88 | 3.78 | 4.11 |
| | 500 | 1.19 | 1.21 | 1.16 | 1.17 | 2.18 | 2.23 |
| CML | 5 | 16.4 | 18.84 | 13.6 | 14.8 | 17.0 | 19.09 |
| | 20 | 5.23 | 5.85 | 4.50 | 4.78 | 5.75 | 6.24 |
| | 50 | 2.96 | 3.65 | 2.68 | 2.81 | 3.46 | 4.07 |
| | 500 | 1.65 | 1.73 | 1.62 | 1.73 | 2.12 | 2.22 |

This document is provided by JAXA.

表2.2.2

単位：$T_*$

| 命令 | 要素数 | 単精度 | V-Reg all | 倍精度 |
|---|---|---|---|---|
| BITS (all on) | 5 | 27.38 | 22.58 | 26.91 |
| | 20 | 12.01 | 10.72 | 12.36 |
| | 50 | 9.28 | 8.79 | 9.75 |
| | 500 | 7.19 | 7.14 | 7.24 |
| BITS (Half on) | 5 | 40.60 | 35.84 | 38.90 |
| | 20 | 14.74 | 13.36 | 14.12 |
| | 50 | 9.88 | 9.53 | 9.96 |
| | 500 | 6.61 | 6.56 | 6.64 |
| BITS (all off) | 5 | 24.98 | 21.38 | 24.53 |
| | 20 | 9.12 | 7.92 | 9.07 |
| | 50 | 6.57 | 6.08 | 6.62 |
| | 500 | 4.32 | 4.27 | 4.31 |
| IDXL (all on) | 5 | 22.40 | 20.04 | 24.49 |
| | 20 | 10.82 | 10.20 | 11.32 |
| | 50 | 8.66 | 8.44 | 9.13 |
| | 500 | 7.20 | 7.24 | 7.24 |
| IDXL (Half on) | 5 | 21.24 | 18.80 | 21.09 |
| | 20 | 9.26 | 8.69 | 9.68 |
| | 50 | 7.59 | 7.28 | 7.82 |
| | 500 | 5.76 | 5.72 | 6.03 |
| IDXL (all off) | 5 | 18.00 | 15.60 | 17.60 |
| | 20 | 7.53 | 6.91 | 7.43 |
| | 50 | 5.59 | 5.33 | 5.71 |
| | 500 | 4.20 | 4.20 | 4.23 |
| GAT (Integer all on) | 5 | 25.04 | 19.00 | 21.72 |
| | 20 | 7.76 | 8.11 | 6.92 |
| | 50 | 4.09 | 5.60 | 4.22 |
| | 500 | 3.09 | 4.47 | 2.48 |
| GAT (Integer Half on) | 5 | 22.44 | 17.20 | 21.00 |
| | 20 | 6.30 | 6.08 | 5.95 |
| | 50 | 2.92 | 3.54 | 3.08 |
| | 500 | 1.43 | 2.37 | 1.45 |
| GAT (Integer all off) | 5 | 21.08 | 16.40 | 19.89 |
| | 20 | 5.32 | 4.07 | 4.97 |
| | 50 | 1.88 | 1.35 | 2.05 |
| | 500 | 0.18 | 0.18 | 0.20 |
| GAT (Logical all on) | 5 | 50.20 | 44.20 | 43.72 |
| | 20 | 19.24 | 19.24 | 18.02 |
| | 50 | 13.16 | 14.67 | 12.89 |
| | 500 | 10.60 | 11.66 | 10.54 |
| GAT (Logical Half on) | 5 | 43.60 | 38.80 | 40.20 |
| | 20 | 15.61 | 14.90 | 15.03 |
| | 50 | 10.47 | 10.92 | 10.61 |
| | 500 | 7.08 | 8.10 | 7.69 |
| GAT (Logical all off) | 5 | 40.84 | 36.14 | 37.91 |
| | 20 | 13.30 | 12.10 | 12.49 |
| | 50 | 7.80 | 7.32 | 7.46 |
| | 500 | 4.46 | 4.37 | 4.43 |
| SCAT (Integer all on) | 5 | 33.04 | 27.00 | 30.90 |
| | 20 | 11.18 | 10.47 | 11.88 |
| | 50 | 7.11 | 7.16 | 8.51 |
| | 500 | 4.60 | 4.78 | 5.25 |
| SCAT (Integer Half on) | 5 | 31.28 | 25.24 | 29.46 |
| | 20 | 9.50 | 8.52 | 9.75 |
| | 50 | 5.90 | 5.23 | 6.42 |
| | 500 | 3.35 | 2.81 | 3.36 |
| SCAT (Integer all off) | 5 | 30.44 | 23.60 | 28.62 |
| | 20 | 8.12 | 6.70 | 8.21 |
| | 50 | 4.37 | 3.84 | 4.54 |
| | 500 | 1.78 | 1.69 | 2.29 |
| SCAT (Logical all on) | 5 | 58.24 | 52.68 | 55.55 |
| | 20 | 22.70 | 21.55 | 22.53 |
| | 50 | 15.69 | 15.60 | 16.62 |
| | 500 | 11.22 | 11.93 | 12.58 |
| SCAT (Logical Half on) | 5 | 52.22 | 46.80 | 51.35 |
| | 20 | 18.72 | 17.16 | 18.89 |
| | 50 | 12.78 | 12.16 | 13.32 |
| | 500 | 8.63 | 8.50 | 9.38 |
| SCAT (Logical all off) | 5 | 50.04 | 44.00 | 49.35 |
| | 20 | 16.23 | 14.28 | 16.40 |
| | 50 | 9.08 | 9.09 | 10.41 |
| | 500 | 6.01 | 5.92 | 6.52 |
| MASK (Integer all on) | 5 | 23.34 | 21.24 | 21.91 |
| | 20 | 8.44 | 7.82 | 8.58 |
| | 50 | 5.14 | 4.96 | 5.37 |
| | 500 | 3.30 | 3.21 | 4.13 |
| MASK (Integer Half on) | 5 | 34.24 | 31.94 | 32.32 |
| | 20 | 10.75 | 9.73 | 9.99 |
| | 50 | 5.44 | 5.48 | 5.74 |
| | 500 | 2.74 | 2.52 | 3.08 |
| MASK (Integer all off) | 5 | 20.00 | 17.77 | 19.33 |
| | 20 | 5.98 | 5.31 | 6.79 |
| | 50 | 2.98 | 2.72 | 3.76 |
| | 500 | 1.19 | 1.19 | 2.17 |
| MASK (Logical all on) | 5 | 25.24 | 23.24 | 23.73 |
| | 20 | 9.10 | 8.43 | 8.39 |
| | 50 | 5.62 | 5.31 | 5.43 |
| | 500 | 3.31 | 3.26 | 3.30 |
| MASK (Logical Half on) | 5 | 21.64 | 19.46 | 21.35 |
| | 20 | 7.40 | 6.78 | 7.05 |
| | 50 | 4.27 | 3.96 | 4.24 |
| | 500 | 2.25 | 2.21 | 2.28 |
| MASK (Logical all off) | 5 | 20.80 | 18.88 | 20.64 |
| | 20 | 6.24 | 5.62 | 6.68 |
| | 50 | 3.20 | 2.85 | 4.40 |
| | 500 | 1.32 | 1.14 | 2.26 |
| IONC (all on) | 5 | 18.40 | 16.00 | 17.31 |
| | 20 | 6.06 | 5.49 | 5.79 |
| | 50 | 3.70 | 3.44 | 3.60 |
| | 500 | 2.25 | 2.30 | 2.26 |
| IONC (Half on) | 5 | 17.68 | 15.24 | 16.49 |
| | 20 | 5.09 | 4.46 | 4.76 |
| | 50 | 2.72 | 2.50 | 2.60 |
| | 500 | 1.27 | 1.23 | 1.26 |
| IONC (all off) | 5 | 16.54 | 14.00 | 15.38 |
| | 20 | 4.06 | 3.48 | 3.78 |
| | 50 | 1.75 | 1.44 | 1.63 |
| | 500 | 0.25 | 0.25 | 0.26 |
| IFBN | 5 | 2.00 | 2.00 | 2.04 |
| | 20 | 0.54 | 0.50 | 0.50 |
| | 50 | 0.16 | 0.16 | 0.16 |
| | 500 | 0.01 | 0.01 | 0.01 |

This document is provided by JAXA.

るため実測値は必ずしも一定せず，ベクトル一要素当りの実行速度はベクトル長が短かい場合，特に変動しやすい。そこで代表的な命令に関して何度か実測した結果の代表的な数値を表2.2.1にあげ，合せて新理論値との比較を行う。（命令のつづきは，M&R－V－SEQ[1]である）

表より明らかな様に要素数の小さい所で理論値と実測値の差は比較的大きいが要素数の大きい所では大体において良い一致を示している。これはまたランダムネスの効果は要素数の小さい所では大きく現われるということとVSM，IPD命令等の和をとる命令は，ベクトルの長さが小さい所ではハードウェアの作り方により $T_i$ が $\lambda \cdot N+\mu$ の形をしていないことから当然といえる。

以上の実測値と理論値の比較から命令単体の実行速度に関しては理論的算出式が使用に耐えるものであることが示された。そこで以前の資料[1]の各命令実行速度図は“all operand in memory”の場合は $12/N \cdot T_u$，“all operand in VR” の場合は $10/N \cdot T_u$ 程度実行速度を遅らせて使えば使用できることになる。また以前の資料[1]のV型命令の平均命令実行時間も同様にすれば使用できる。V型命令のオペランドの一部がVRにある場合には以前の資料[1]の訂正は $12/N \sim 10/N \cdot T_u$ の遅れでおさえられることも明らかである。

次にいくつかの配列特殊関数の実行速度の実側値とリストベクトルの実行速度の理論的実行速度算出式を示し，合せてこれらの関数の使用法の一端について述べることにする。

APの機械命令を便宜的に，a）単純命令，b）複合命令Ⅰ，c）複合命令Ⅱ，d）比較，サーチ命令，e）ビット命令，に分ける。a）群に属するものとしてVMV，VAD，VML，VDV，VSM命令があり，b）群に属するものとしてIPD，VNM，AJM，AVG，SMVA命令があり，c）群にはCVA，CVM，PMM，POLY命令があり，d）群にはFIND関係，比較関係の命令がある。a）群のVSM命令とb），c），d）群のFIND関係の命令は全て配列特殊関数としてFORTRANから直接使用できる。そしてその実行速度は2.1節における方法で理論的に算出できる。またVSMとIPDに関してはその実測値が2.2節に表示してあるので，ここでは機械命令語と直接一対一に対応しない配列特殊関数[3]（IONCは機械命令と一対一に対応するがその速度算出は特殊なのでこれも含める）についいの実行速度の実測値を表2.2.2に示す。

この表から実行速度はデータ精度には関係せず，データがVRにある場合にパイプラインの立ち上り時間の関係で，データ長が短かい程一要素あたりの実行時間が速くなっていることがわかる。またGAT，SCAT関数[3]はマスク情報がInteger の場合の方がBit のものより2倍程度速いことは注意すべきである。これは配列特殊関数がどの様な機械命令語を使っているかということを調べれば判明することであって，それを表2.2.3に示す。

表2.2.3

| 配列特殊関数名 | マスク情報 | 機械命令（V型） |
|---|---|---|
| MASK | L，I | VMM |
| GAT | I | VMM |
| GAT | L | GAT，VMM |
| SCAT | I | VMM，VMM |
| SCAT | L | GAT，VMM，VMM |
| BITS | | GAT |
| IDXL | | GAT |

マスク情報LはBit列，Iは整数列を示す。

V型命令のあつかうオペランドのうち1つでもリストベクトルがあれば，そのV型命令固有の実行時間 $T_i$ は，

$$T_i=\tilde{\lambda}N+\mu+25 \quad \begin{array}{l}\tilde{\lambda}=2\,;\,\lambda\leq 2\\ \tilde{\lambda}=\lambda\,;\,\lambda>2\end{array} \qquad (2.2.1)$$

となる。（2.2.1）式において $\lambda$ は表2.1.2.における $\lambda$ である。従ってその実行速度は（2.1.4），（2.1.5）式の $\lambda$ に $\tilde{\lambda}$ を代入すれば，ほぼ正確に求まる。（2.2.1）式より $\lambda$ が $\frac{1}{2}$，1のV型命令はリストベクトルではそれぞれ4倍，2倍の時間がかかることと，ベクトル長が短かい場合に（2.2.1）式の25という値はベクトル一要素当りの実行速度を低下させることに注意する必要がある。

V型命令の上手な使用法として，複合命令が使える場合には複合命令を使うのが望ましく，複合命令もⅠ型よりⅡ型の方が実行速度は速い。さらにベクトル長が短かい時にはパイプラインの立ち上り時間をかせぐ意味でも複合命令を使うことが望ましい。例えば

A（＊）＝0.5＊（B（＊）＋C（＊）） （2.2.2）

A（＊）＝AVRG（B（＊），C（＊）） （2.2.3）

を比較すると単精度，倍精度で（2.2.2）は $T_i$ の値がそれぞれ $\frac{1}{2}N+8+N+9=\frac{3}{2}N+17$，$N+8+2N+11=3N+19$ であるのに対し（2.2.3）は，それぞれ $\frac{1}{2}N+8$，$N+8$ で約3倍の速度である上に，パイプラインの立ち上り時間 $T_P'$ も半分になるという利点がある。同様な関係が複合Ⅰと複合Ⅱの命令の間でも成り

This document is provided by JAXA.

立つ。10×10 の行列と要素数10のベクトルの積 $A_i=B_{ij}\cdot C_i$ もIPDを10回繰り返えせば，（その実行時間はデータ供給能力を無視すれば単精度で $T_P'$ を56とすると）$10(N+31)+10T_P'=1050\cdot T_u$ かかるが，PMMを使えば，$(10+20)\cdot N+13+T_P'=377\cdot T_u$ ですむ。但しSMVAはベクトルが短かいとき（all operand in memoryのとき $N\leq 32$, all operand in VRのとき $N\leq 24$）を除いては不利になる。またAJMはVADとVMLを使うよりは $T_i$ の値からも，$T_P'$ の値からも得であるが $T_i$ の値に関しAVRGを使う方がもっと速いことに注意する必要がある。

リストベクトル，GAT，SCAT，MASK，BITS，IDXL，IONC等配列特殊関数[3]には種々の利用法があり，非常に凝った使用法もある。この様な関数や論理配列，論理式の組合せは実行速度迄考慮して考えると詰将棋の様な面白さがあるが，ここではごく平凡でありきたりの使用法の一例を示すにとどめる。例題はIF文をもつDOループのベクトル演算化に関するものである。

```
   DIMENSION  A(100),B(100),D(100),
   E(100),F(100),L(100),LN(100),
   IA(100),IB(100)
   LOGICAL L, LN
   INDEX ID/1, IM/, IC/1, IN/
   DO 1 I=1, 100
   D(I)=E(I)
   IF(A(I).LT.C) D(I)=C*E(I)
   B(I)=A(I)+D(I)
1  CONTINUE
```

(1) 上のFORTRAN文は配列特殊関数GAT，SCATを使えば，

```
   L(*)=A(*).LT.C
   IM=IONC(L(*))
   D(ID)=GAT(L(*), E(*))
   F(ID)=GAT(L(*), A(*))
   F(ID)=F(ID)+D(ID)*C
   IN=100-IM
   LN(*)=.NOT.L(*)
   D(ID)=GAT(LN(*), E(*))
   E(IC)=GAT(LN(*), A(*))
   E(IC)=E(IC)+D(IC)
   B(*)=SCAT(L(*), B(*), F(*))
   B(*)=SCAT(LN(*), B(*), E(*))
```

と表現できる。

(2) 配列特殊関数GAT，SCAT，IDXLを使えば，

```
   L(*)=A(*).LT.C
   IM=IONC(L(*))
   IA(ID)=IDXL(L(*))
   IN=100-IM
   IB(IC)=IDXL(.NOT.L(*))
   D(ID)=GAT(IA(ID), E(*))
   F(ID)=GAT(IA(ID), A(*))
   F(ID)=F(ID)+C*D(ID)
   D(IC)=GAT(IB(IC), E(*))
   E(IC)=GAT(IB(IC), A(*))
   E(IC)=E(IC)+D(IC)
   B(*)=SCAT(IA(ID), B(*), F(ID))

   B(*)=SCAT(IB(IC), B(*), E(IC))
```

と表現できる。

(3) 配列特殊関数MASKを使うと，

```
   L(*)=A(*).LT.C
   B(*)=A(*)+E(*)
   D(*)=A(*)+C*E(*)
   B(*)=AMASK(L(*), D(*), B(*)
   (B(*)=SCAT(L(*), B(*), D(*)
```

としても良いが，L(*)が論理配列の場合はMASKの方がSCATより速い）

と表現できる。

(4) 配列特殊関数IDXLからリストを作り，それによりリストベクトルを用いれば

```
   L(*)=A(*).LT.C
   IM=IONC(L(*))
   IN=100-IM
   IA(ID)=IDXL)(L(*))
   IB(IC)=IDXL(.NOT.L(*))
   B(IA(ID))=A(IA(ID))+C*E(IA(ID))
   B(IB(IC))=A(IB(IC))+E(IB(IC))
```

と表現できる。

(1)〜(4)の記述法は計算結果から見れば全く等価であるが，実行速度に関しては場合によりかなり差がでてくる。実行速度の面からどの記述を取るかを選択する場合，次に述べる点を考慮にして決めるべきである。

(i) 表2.2.3に示した様に配列特殊関数BITS，IDXL，GAT，SCAT，MASKは機械命令語GAT，VMMの組合せである。これらの配列特殊関数の実行速度の算出は複雑であるが目安として以下の式を用いても良い。

This document is provided by JAXA.

4N+100≦IDXL，BITSの実行速度≦7N+100
（2.2.4）注1）

N+110≦［GAT（MASK情報整数列），MASK（MASK情報整数列，論理型）］≦3N+110
（2.2.5）注1）

1.5N+140≦SCAT（MASK情報整数型）≦4.5N+140
（2.2.6）注1）

ＧＡＴ，ＳＣＡＴ関数でＭＡＳＫ情報が論理型のものの実行速度は，それぞれ（2.2.5），（2.2.6）式に（2.2.4）式の辺々を加えたもので上下から押えられる。これはＭＡＳＫ情報が論理型の場合いったん論理型から整数型への変換を行うことによるものである。（2.2.4）～（2.2.6）式において実行速度はＭＡＳＫ情報の数と全体の要素数の比が０から１に近ずくに従い左辺で表わされる速度から右辺で表わされる速度に近ずく。

(ii)　リストベクトルの実行速度は（2.2.1）式で表わされるがこの速度は加減算に関して通常のベクトル加減算の４倍かかり，乗算に関して２倍かかる。また立ち上り時間がかかる。

(iii)　表現の単純さ。

記述式の単純さを選択基準にすれば(3)，(4)が優れているが，実行速度からいえば必ずしもそうではない。判断分岐に従う計算が単純であり，かつ判断分岐が複雑でなければ(3)が最も良いであろう。また計算が単純で判断分岐が複雑であれば(4)が良いであろう。しかし判断分岐のもとでの計算が複雑，かつ長大であれば(1)，(2)にも長所がある。(1)と(2)の使い分けは比較的簡単であって１つのＭＡＳＫ情報で２度以上ＧＡＴ，ＳＣＡＴ関数を使う場合は速度の面から(2)が優る。

以上の他にＩＦ文の表記法にはＦＩＮＤ関係，ＩＦ－ＢＯＮ，ＩＥＢＯＦＦ等の配列特殊関数も使える。これは１つのベクトル計算において，絶対数が極く少数の要素（１～数要素）のみが例外的な計算を必要とする様な場合である。またあらかじめ例外的な計算を必要とするベクトルの要素位置がわかっている場合には，これらの表記法は用いてはならない。ＩＮＤＥＸ変数の制御を用いるか，全部計算を行なった後で別に計算して，はめ込むかすべきである。判断分岐を含むベクトル計算の記述法に関しては後の章で詳しく述べることにする。

## 2.3　基本的演算の実行速度の実測値

注1）（2.2.4）～（2.2.6）式は，演算の立ち上り時間$T_P'$を含んでいる。

本節では通常のベクトル演算において生ずる基本的演算のパターンについて，ＡＰとＣＰの実行速度の実測値と実測値に基ずく実行速度比を示すことにする。

先ず第一に基本外部関数のうち代表的なものの実行速度の実測値を示す。ＡＰ－ＦＯＲＴＲＡＮではＳＩＮ，ＣＯＳ，ＬＯＧといった基本外部関数の変数部に，ベクトルを代入すれば一度にベクトルの各要素を変数とする基本外部関数の値を計算する様になっている。すなわちベクトル$X=(x_1,\cdots x_N)$としたとき通常のＦＯＲＴＲＡＮでは，

```
    DO  1  I＝1，N
 1  A（I）＝SIN（X（I））
```
（Ａ）

としか記述できないが，ＡＰ－ＦＯＲＴＲＡＮでは上の記述の代りに

A（＊）＝SIN（X（＊））注1）　　（Ｂ）

と記述しても良いことになっている。また，(A)，(B)の記述法に対応して実行速度と精度が異なっている注2)ので必要に応じて使いわけると良い。実行速度の実測値とその比率を表2.3.1～表2.3.4に示す。

表2.3.1　ＳＩＮ

| ベクトル長 | 単精度 AP | 単精度 CP | CP/AP | 倍精度 AP | 倍精度 CP | CP/AP |
|---|---|---|---|---|---|---|
| 5 | 56μs | 21μs | 0.375 | 82μs | 29μs | 0.354 |
| 10 | 30μs | | 0.700 | 45μs | | 0.644 |
| 20 | 16μs | | 1.31 | 26μs | | 1.12 |
| 50 | 7.6μs | | 2.76 | 14μs | | 2.07 |
| 70 | 6.1μs | | 3.44 | 12μs | | 2.42 |
| 100 | 4.9μs | | 4.29 | 10μs | | 2.9 |

表2.3.2　EXP

| ベクトル長 | 単精度 AP | 単精度 CP | CP/AP | 倍精度 AP | 倍精度 CP | CP/AP |
|---|---|---|---|---|---|---|
| 5 | 42μs | 17μs | 0.404 | 74μs | 26μs | 0.351 |
| 10 | 22μs | | 0.773 | 40μs | | 0.650 |
| 20 | 12μs | | 1.42 | 23μs | | 1.13 |
| 50 | 5.7μs | | 2.98 | 13μs | | 2.00 |
| 70 | 4.6μs | | 3.70 | 11μs | | 2.36 |
| 100 | 3.7μs | | 4.59 | 9.5μs | | 2.74 |

注１）“＊”の代りにＩＮＤＥＸ変数でも良い。

注２）基本外部関数の精度については，センターに掲示するか，もしくはセンタニュースに載せる予定である。

This document is provided by JAXA.

表2.3.3　SQRT

| ベクトル長 | 単精度 AP | 単精度 CP | CP/AP | 倍精度 AP | 倍精度 CP | CP/AP |
|---|---|---|---|---|---|---|
| 5 | 45μs | 19μs | 0.422 | 51μs | 23μs | 0.451 |
| 10 | 24μs | | 0.792 | 29μs | | 0.793 |
| 20 | 13μs | | 1.46 | 18μs | | 1.28 |
| 50 | 6.5μs | | 2.92 | 11μs | | 2.09 |
| 70 | 5.3μs | | 3.58 | 9.8μs | | 2.35 |
| 100 | 4.3μs | | 4.42 | 8.9μs | | 2.58 |

表2.3.4　LOGe

| ベクトル長 | 単精度 AP | 単精度 CP | CP/AP | 倍精度 AP | 倍精度 CP | CP/AP |
|---|---|---|---|---|---|---|
| 5 | 49μs | 21μs | 0.429 | 64μs | 31μs | 0.484 |
| 10 | 26μs | | 0.808 | 36μs | | 0.861 |
| 20 | 13μs | | 1.62 | 22μs | | 1.41 |
| 50 | 7.6μs | | 2.76 | 13μs | | 2.38 |
| 70 | 6.3μs | | 3.33 | 12μs | | 2.58 |
| 100 | 5.4μs | | 3.89 | 11μs | | 2.82 |

なおCPの実行速度はベクトル長に関して不変であるが変数の値により多少異なるので平均値をとった。APとCPの実行速度はいずれの場合でもベクトル長10と20の間で交差している。また表には示さなかったがAPの実行速度はベクトル長300位迄は減少を続けるが，ベクトル長100の場合と300の場合の実行速度の差はベクトル長100の場合と50の場合の実行速度の差は若干下回る程度である。

次にいくつかのベクトル演算の基本的なパターンに関するAPとCPの実行速度の実測値とその速度比を示す。パターン(1)〜(5)迄は普通のDOループによく現われる型であり，(6)はリストベクトルの単純な型，(7)は1次元圧縮性粘性非定常流の計算に用いる最も単純な差分法によるパターンであって通常のFORTRANではDOループの中にIF文のある形になっているのでとりあげたが，データはIF文の正否が一定周期で交互にでてくる様に人為的に作ってある。(8)はフレドホルム第2種積分方程式の反復解法のパターンである。以下にパターン(1)〜(8)の数式表現と通常のFORTRAN記述(A)と，ベクトルを用いたFORTRAN記述(B)を示し，(A)をCP(OPT2)注1)で実行し，(B)をAP(OPT2)注1)で実行した場合の実行速度の実測値とその実行速度比を，表2.3.5〜表2.3.12に示す。((A)はAPでそのまま実行可能であるが速度はCPの半分程度になる。)

(1) $a_i = b_i + c_i * d_i$

(A)
```
      DO 1 I=1, M
    1 A(I)=B(I)+C(I)*D(I)
```
(B)
```
      INDEX IX/1, M/ ; IX は Index 変数, 以下(3)迄同様
      A(IX)=B(IX)+C(IX)*D(
            IX)
```

(2) $a_i = b_i - (X * b_i + Y * d_i)$

(A)
```
      DO 1 I=1, M
    1 A(I)=B(I)-(X*B(1)+Y*D(1))
```
(B)
```
      A(IX)=B(IX)-(X*B(IX)+Y*D(
            IX))
```

(3) $a_i = X + b_i * (Y * c_i + Z * d_i)$

(A)
```
      DO 1 I=1, M
    1 A(I)=X+B(I)*(Y*C(I)+Z*D(I))
```
(B)
```
      A(IX)=X+B(IX)*(Y*C(IX)
            +Z*D(IX))
```

(4) $a_i = a_i + X * (\frac{1}{2} * (b_{i+1} + c_{i+1} + b_{i-1} + c_{i-1}) - 2 * (b_i + c_i)) * d_i$

(A)
```
      DO 1 I=2, M-1
    1 A(I)=A(I)+X*(0.5*(B(I+1)
          +B(I-1)+C(I+1)+C(1-1)
          -2.0*(B(I)+C(I)))*D(I)
```
(B)
```
      INDEX IX/1, M/, IZ/2, M-1/,
      IY/1, M-2/, IW/3, M/
      B(IX)=AVRG(B(IX), C(IX))
      C(IZ)=B(IY)+B(IW)
      B(IZ)=4.0*B(IZ)
      C(IZ)=C(IZ)-B(IZ)
      B(IZ)=C(IZ)*D(IZ)
      A(IZ)=A(IZ)+B(IZ)
```

(5) $a_i = X + (b_{i+1} - b_{i-1}) / (0.5 * (c_{i+1} + c_{i-1}))$

(A)
```
      DO 1 I=2, M-1
    1 A(I)=X+(B(I+1)-B(I-1))/
           (0.5*(C(I+1)+C(I-1)))
```
(B)
```
      D(IZ)=AVRG(C(IY), C(IW))
      C(IZ)=B(IW)-B(IY)
      D(IZ)=C(IZ)/D(IZ)
      A(IZ)=X+D(IZ)
```

(6) $a_{IA(i)} = b_{IB(i)} + c_{IB(i)} * d_{IA(i)}$

ここで$f(i)$, $g(i)$は$i$の整数型関数とする。

(A)
```
      DO 1 I=1, M
```

---

注1) フォートランコンパイラの最適化のレベルを表わす。OPT2は最高レベルの最適化を行っている。

This document is provided by JAXA.

```
      J=IA(I)
      K=IB(I)
    1 A(J)=B(K)+C(K)*D(J)
(B)   A(IA(I))=B(IB(I))
     +C(IB(I))*D(IA(I))
```

(7) $a_i = c_i - ZB*(c_i - c_{i+1})*bi$
$\quad - ZA(b_{i+1} - b_{i-1}) \quad b_i > 0$

$a_i = c_i - ZB*(c_{i+1} - c_i)*bi$
$\quad - ZA(b_{i+1} - b_{i-1}) \quad b_i \leq 0$

$\tilde{a}_i = b_i - ZB*(bi - b_{i-1})*b_i$
$\quad - \{ZC*(b_{i+1}c_{i+1} - d_{i-1}c_{i-1})$
$\quad + ZD*(b_{i+1} - 2b_i + b_{i-1})\}/c_i$
$\quad b_i > 0$

$\tilde{a}_i = b_i - ZB*(b_{i+1} - b_i)*b_i$
$\quad - \{ZC*(d_{i+1}c_{i+1} - d_{i-1}c_{i-1})$
$\quad + ZD*(b_{i+1} - 2b_i + b_{i+1})\}/c_i$
$\quad b_i \leq 0$

$\hat{a}_i = d_i - ZB*(d_i - d_{i-1})*b_i - ZE*d_i*$
$\quad (b_{i+1} - b_{i-1}) - \{ZF*(b_{i+1} - b_{i-1})^2$
$\quad + ZG*(d_{i+1} - 2d_i + d_{i-1})\}/c_i$
$\quad b_i > 0$

$\hat{a}_i = d_i - ZB*(d_{i+1} - d_i)*b_i - ZE*d_i*$
$\quad (b_{i+1} - b_{i-1}) - \{ZF*(b_{i+1} - b_{i-1})^2$
$\quad + ZG*(d_{i+1} - 2d_i + d_{i-1})\}/c_i$
$\quad b_i \leq 0$

(A)，(B)のFORTRAN文は省略するが通常のFORTRANは$b_i$の正負にIF判定があり，AP－FORTRANはMASKを使っている。

(8) $a_i = b_i - \Sigma e_{ij} d_j + c_i$

(A)，(B)ともに省略する。(B)ではPMMを使用している。

表2.3.4～表2.3.10において，ベクトル長が３００～５００程度になるとAPの実行速度が相対的に低下しているのはVRの関係であって，テストプログラムでVRの使用をコンパイラにまかせたためと思われる。(7)～(8)はVRを意識して使うため筆者がVRーロードを行なっているのでその様なことは起っていない。これで見るとコンパイラはVRの利用に関して，それ程高度な割り付け法を行なっていない様なので，注1)自信のある利用者は自分でVRを使うべきであるといえる。(6)はリストベクトルに関するものであるが，一般的でないリストを利用しているため現実的なリスト計算ではAPの実行速度はもう少し遅れると思われる（メモリバンク競合のファクタがもっと入る）。(5)が遅いのはVDV命令のためである。この結果からもVDV命令（ベクトルの割り算）は必要最小限に押さえる様努力すべきであろう。

注1）コンパイラにVRの利用をまかせた場合，ベクトルテンポラリとしてのみ使用する様になっている。ベクトルテンポラリとはA＝B＋C＋Dと表わしたとき，APはB′＝B＋C，A＝B′＋Dと実行するが，B′をベクトルテンポラリという。

表２３.５ (1)

| ベクトル長 | 実行時間 (単位μs) | | 実行速度比 |
|---|---|---|---|
| | AP | CP | CP/AP |
| 5 | 9.2 | 9.6 | 1.0 |
| 10 | 9.9 | 17.9 | 1.8 |
| 20 | 11.2 | 34.5 | 3.1 |
| 30 | 12.6 | 51.4 | 4.1 |
| 40 | 13.9 | 68.0 | 4.9 |
| 50 | 15.3 | 85.2 | 5.6 |
| 100 | 22.0 | 170.1 | 7.7 |
| 200 | 35.5 | 338.3 | 9.5 |
| 300 | 49.0 | 512.9 | 10.5 |
| 500 | 96.5 | 858.7 | 8.9 |

表2.3.6 (2)

| ベクトル長 | 実行時間 (単位μs) | | 実行速度比 |
|---|---|---|---|
| | AP | CP | CP/AP |
| 5 | 17.1 | 14.9 | 0.9 |
| 10 | 18.6 | 28.7 | 1.5 |
| 20 | 21.3 | 56.3 | 2.6 |
| 30 | 24.0 | 83.9 | 3.5 |
| 40 | 26.7 | 111.7 | 4.2 |
| 50 | 29.4 | 139.5 | 4.7 |
| 100 | 42.9 | 277.8 | 6.5 |
| 200 | 69.9 | 558.8 | 8.0 |
| 300 | 108.9 | 836.2 | 7.7 |
| 500 | 192.7 | 1406.8 | 7.3 |

表2.3.7 (3)

| ベクトル長 | 実行時間 (単位μs) | | 実行速度比 |
|---|---|---|---|
| | AP | CP | CP/AP |
| 5 | 21.1 | 16.8 | 0.8 |
| 10 | 23.3 | 32.9 | 1.4 |
| 20 | 26.9 | 64.7 | 2.4 |
| 30 | 30.4 | 96.9 | 3.2 |
| 40 | 34.0 | 129.5 | 3.8 |
| 50 | 37.7 | 161.7 | 4.3 |
| 100 | 55.7 | 323.3 | 5.8 |
| 200 | 91.7 | 648.7 | 7.1 |
| 300 | 139.8 | 977.7 | 7.0 |
| 500 | 240.4 | 1660.3 | 6.9 |

This document is provided by JAXA.

表2.3.8　(4)

| ベクトル長 | 実行時間 AP | (単位μs) CP | 実行速度比 CP/AP |
|---|---|---|---|
| 5 | 26.5 | 17.3 | 0.7 |
| 10 | 29.4 | 43.9 | 1.5 |
| 20 | 34.6 | 96.5 | 2.8 |
| 30 | 38.4 | 149.5 | 3.9 |
| 40 | 43.2 | 202.2 | 4.7 |
| 50 | 47.6 | 254.5 | 5.3 |
| 100 | 70.1 | 520.4 | 7.4 |
| 200 | 115.3 | 1058.7 | 9.2 |
| 300 | 184.0 | 1600.2 | 8.7 |
| 500 | 333.1 | 2685.2 | 8.1 |

表 2.3.9　(5)

| ベクトル長 | 実行時間 AP | (単位μs) CP | 実行速度比 CP/AP |
|---|---|---|---|
| 5 | 18.1 | 13.1 | 0.7 |
| 10 | 22.9 | 32.9 | 1.4 |
| 20 | 32.1 | 72.3 | 2.3 |
| 30 | 40.7 | 112.5 | 2.8 |
| 40 | 49.9 | 152.3 | 3.1 |
| 50 | 58.9 | 192.6 | 3.3 |
| 100 | 103.9 | 394.5 | 3.8 |
| 200 | 194.0 | 796.7 | 4.1 |
| 300 | 307.1 | 1209.5 | 3.9 |
| 500 | 521.5 | 2048.2 | 3.9 |

表2.3.10　(6)

| ベクトル長 | 実行時間 AP | (単位μs) CP | 実行速度比 CP/AP |
|---|---|---|---|
| 5 | 15.7 | 17.9 | 1.1 |
| 10 | 20.7 | 34.4 | 1.7 |
| 20 | 30.8 | 67.6 | 2.2 |
| 30 | 38.5 | 100.4 | 2.6 |
| 40 | 46.3 | 133.7 | 2.9 |
| 50 | 54.2 | 166.4 | 3.1 |
| 100 | 98.3 | 331.9 | 3.4 |
| 200 | 176.0 | 664.2 | 3.8 |
| 300 | 263.2 | 1003.2 | 3.8 |
| 500 | 417.8 | 1699.2 | 4.1 |

表2.3.11　(7)

| ベクトル長 | 実行時間 AP | (単位μs) CP | 実行速度比 CP/AP |
|---|---|---|---|
| 50 | 399 | 1273 | 3.2 |
| 100 | 631 | 2625 | 4.2 |
| 200 | 1098 | 5380 | 4.9 |
| 400 | 2025 | 10948 | 5.4 |

表2.3.12　(8)

| ベクトル長 | 実行時間 AP | (単位μs) CP | 実行速度比 CP/AP |
|---|---|---|---|
| 50 | 350 | 4175 | 11.9 |
| 100 | 1127 | 17274 | 15.3 |
| 200 | 4039 | 68600 | 19.0 |
| 400 | 15262 | 332326 | 21.8 |

# 3　プログラムレベルでのＡＰの実行速度

## 3.1　プログラムレベルでのＡＰの実行速度の算出

一つのプログラムに対するＡＰの処理能力はコンパイラの作るオブジェクトプログラムの実行における総走行命令数を実行時間で除することにより定まる。これを$E_{AP}$とすると

$$E_{AP}=\frac{\sum_{i=1}^{\widetilde{M}} M_i+\sum_{i=1}^{\widetilde{R}} R_i+\sum_{i=1}^{\widetilde{V}} V_i N_i}{\sum_{i=1}^{\widetilde{M}} TM_i+\sum_{i=1}^{\widetilde{R}} TR_i+\sum_{i=1}^{\widetilde{V}} TV_i N_i} \quad (3.1.1)$$

(3.1.1)式において，$M_i$，$\widetilde{M}$，$TM_i$ はそれぞれM型命令，M型命令の総走行ステップ数，$M_i$ 命令の実行時間であり，$R_i$，$\widetilde{R}$，$TR_i$ はそれぞれR型命令，R型命令の総走行ステップ数，$R_i$ 命令の実行時間であり，$V_i$，$\widetilde{V}$，$TV_i$はそれぞれV型命令，V型命令の総走行ステップ数，オペランドの長さが$N_i$ の場合の$V_i$ 命令のベクトル一要素当りの実行時間，$N_i$は$V_i$命令のベクトルオペランド長である。

(3.1.1)式は正確であるが$E_{AP}$ を見積もるのに不適当であるので，不正確ではあるが平均化を行う。M型，R型命令は種々の場合の平均命令実行時間を2.1で算出してあるのでこれを用いることにする。そのため，$M=M+\widetilde{R}$ と改めて書き，M型，R型命令の平均命令実行時間をＴＭと書く。V型命令についてはプログラム走行中に表われる複合命令を全て単純V型命令，すなわち2.2節における(a)群に分解してその総走行命令数を数えなおし改めて$\widetilde{V}$とする。そして$N$を全てV型命令の単純V型命令に分解して考えた場合のプログラム走行中におけるV型命令のオペランドの出現頻度の重みを$N_i$ にかけて計算したものであり，プログラムにおけるV型命令のオペランドの平均の長さとする。つまり，$A_i=B_{ij}*C_j$　$i,j=1,10$ の場合A(＊)=PMM(C(＊),B(＊,＊))はVSMが10回，VMLが10回と数え，$N_i=10$のベクトルが20回出現したと数える。従ってそのプログラムに上記PMMが10回出現し，$A_i=B_i\times C_i$　$i=1,2,\cdots\cdots,100$が5回，$A_i=B_i$　$i=1,2,\cdots\cdots,50$が1回出現すれば$N=200\times10+100\times5+50/200+5+1\cong12$

This document is provided by JAXA.

となる。TVをV型命令のオペランド長$N$に対するV型命令の1ベクトルオペレーションの平均命令実行時間算出法において今回確定した$T_i$, $T'_P$ を代入して求め，これをTVとする。従って以前の資料[1]と同じくTVは$N$とインターリーブ数と精度の関数となっているが，1つのプログラムでオペランド長の平均を上記の様にとり，そのプログラムで扱うV型命令のオペランドの精度を固定し，インターリーブ数を32とすればTVはプログラム単位で定まる。従って，(3.1.1)式は

$$E_{AP}=\frac{\widetilde{M}+\widetilde{V}\cdot N}{TM\cdot\widetilde{M}+TV\cdot\widetilde{V}\cdot N} \qquad (3.1.2)$$

と簡単化される。TMは表2.1.4の様に種々の値をとり得るが精度を固定すれば(通常のプログラムでは単精度と倍精度の浮動小数点演算の出現比率は4：1といわれるのでこれを採用すれば精度を固定する必要がなくなる。しかし航技研ではプログラム単位で倍精度化できるので4：1の比率を採用するのが適当かどうか不明であるので，あえて精度別にした。)あとはV型命令の頻度と$N$の大きさからプログラム走行中の浮動小数点演算のうちV型命令で実行されるものの百分率の大凡の見当をつけて表2.1.4からTMを定めることができる。この様にすれば(3.1.2)式の$E_{AP}$はプログラムから大凡の値が求まるが(その代り正確さは失なわれた。しかしこの様な評価に正確さは不用であって統計的な確からしさがある程度保証されていれば十分である。)CPとの処理能力を比較して或るプログラムをAPで実行するのが有利か，CPで実行するのが有利かの見当をつけるためにはまだ不便である。

そこでV型命令のCP命令に対する変換率を導入する。これはV型命令をCPで実行するとCP命令いくつに対応するかという比率であってこれをVCPと表わす。(3.1.2)式を定める際に，AP－プログラム中のV型命令を全て単純命令に分解したのでVAD, VML, VDV, VMV, VSM命令について考えれば良い。VADはCP－プログラムでは

```
      DO  1  I=1, N
   1  A(I)=B(I)+C(I)
```

と表わされるがこれはCP4命令に対応し，同じ様にVML, VDVもCP4命令に，VMVは3命令に，VSMは2命令に対応する。従ってこれらの命令の出現確率を定めればVCPは定まる様に思えるが，事情はもっと複雑である。通常1つのDOループをV型命令で書きなおすためには複数のV型命令が必要である。従ってCP命令の数対V型命令の数の比は確率変数となり，その期待値は1と4の間にあることは理論的に明らかであるが，その分布密度関数は全く不明である。そこで航技研のプログラムを対象としてVCPの値を調査したが，その結果は $2.0\leq VCP\leq 3.6$ であった。この調査はプログラム中の最も内側(走行回数の多い)のDOループのアセンブラーリストからV型命令で記述できる部分を抜き出し，命令数を数え，一方スカラー演算の部分をはずしてV型命令でDOループの書きなおしを行うことによりなされた。この調査の結果にもとずき，VCPの標準値として VCP＝0.2, 2.75, 3.5 を取ることを決める。VCPが定まったのでAPとCPの実行速度の比較のために(3.1.2)式を変形して，

$$\widetilde{E}_{AP}=\frac{\widetilde{M}+VCP\cdot\widetilde{V}\cdot N}{TM\cdot\widetilde{M}+TV\cdot\widetilde{V}\cdot N} \qquad (3.1.3)$$

とする。(3.1.3)式は時間の単位が$T_*$ であるので，$E_{AP}$をMIPS(Mega Instruction per Second)を単位として表わせば

$$E_{AP}=\frac{\frac{\widetilde{M}}{\widetilde{V}}+VCP\cdot N}{TM\cdot\frac{\widetilde{M}}{\widetilde{V}}+TV\cdot N}\times\frac{1}{T_*}=\frac{\frac{\widetilde{M}}{\widetilde{V}}+VCP\cdot N}{\frac{\widetilde{M}}{\widetilde{V}}\cdot TM+TV\cdot N}\times 11.1_{MIPS} \qquad (3.1.4)$$

となる。ここでV型命令300回に一度VRをプログラムでロードするとすると，これを考慮して(3.1.4)式は

$$E_{AP}=\frac{\frac{\widetilde{M}}{\widetilde{V}}+VCP\cdot N}{\frac{\widetilde{M}}{\widetilde{V}}\cdot TM+TV\cdot N+3.3}\times 11.1_{MIPS} \qquad (3.1.5)$$

となる。CPの実行速度はMIPSを単位として単精度で3.75MIPS, 倍精度で3.31MIPS, 4倍精度で1.74MIPSであるので，これを$E_{CP}$とし，$E_{CP}$と$\widetilde{E}_{AP}$の比を縦軸に，プログラム中のV型命令の総走行数とM&R型命令の総走行数の比 $\widetilde{M}/\widetilde{V}$ を横軸にとってAPとCPの実行速度比を，VCPの値と精度の全ての組合せについて図3.1.1～図3.1.9に示した。図中VLとあるのは(3.1.5)式中の$N$を表わす。読者は自分のプログラムでの$N$の大きさについては大体知り得るであろうし$\widetilde{M}/\widetilde{V}$の値も大体見当がつくであろう。注1)

注1) $\widetilde{M}/\widetilde{V}$ の値を計算する場合のV型命令の数え方を例で示すと,

```
      DIMENSION  B(20)
      DO  1  I=1, 100
      DO  1  J=1, 100
   1  A(*,I,J)=PMM(B(*),C(*,*,I,J))
```

のV型命令の数はPMMをVADとVSMに分解して数えるから，VSMの数20, VMLの数20で$(20+20)\times100\times100=4\times10^5$となる。M型，R型命令の数はアセンブリーリストを調べればわかる。

This document is provided by JAXA.

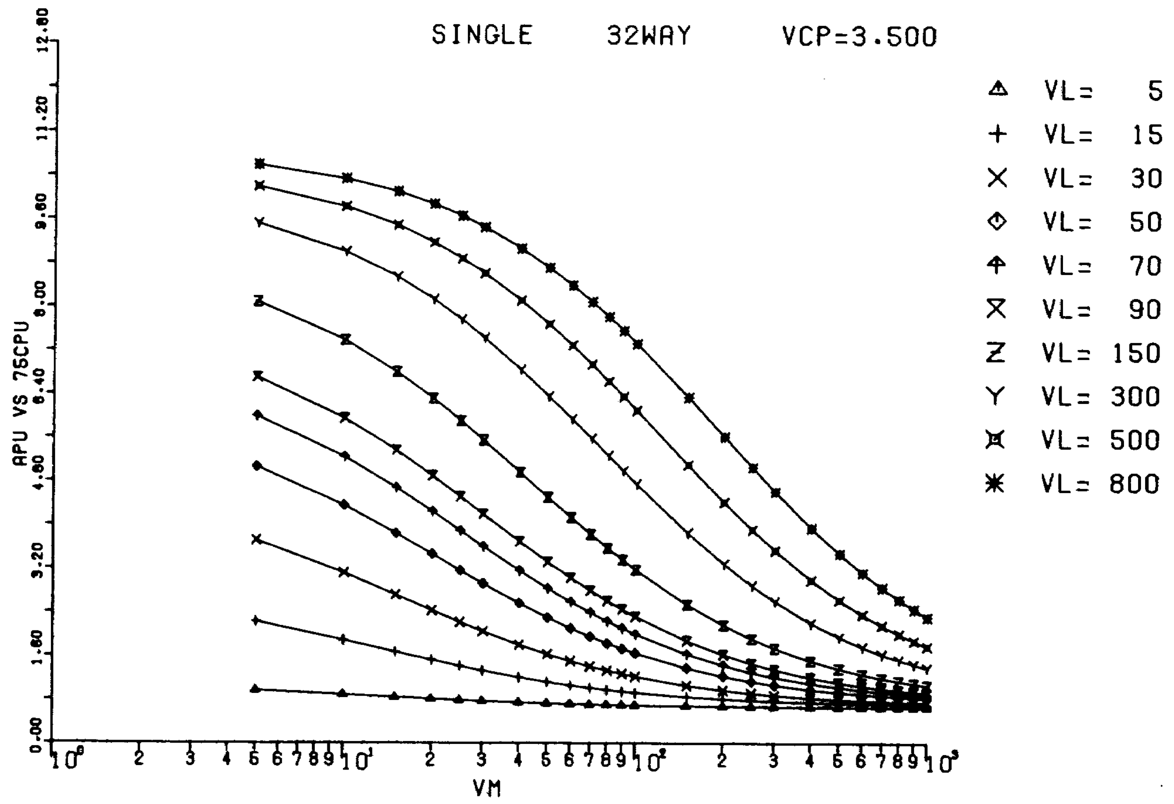



図３－１－１

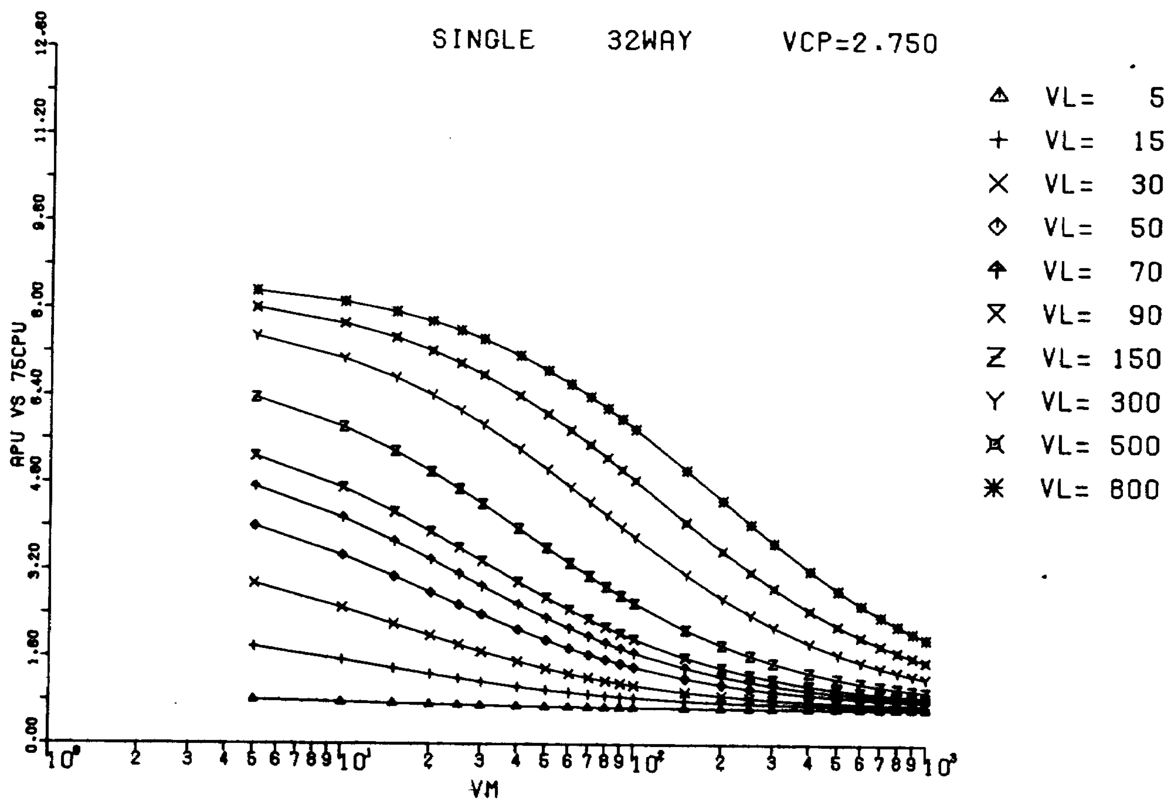



図３－１－２

This document is provided by JAXA.

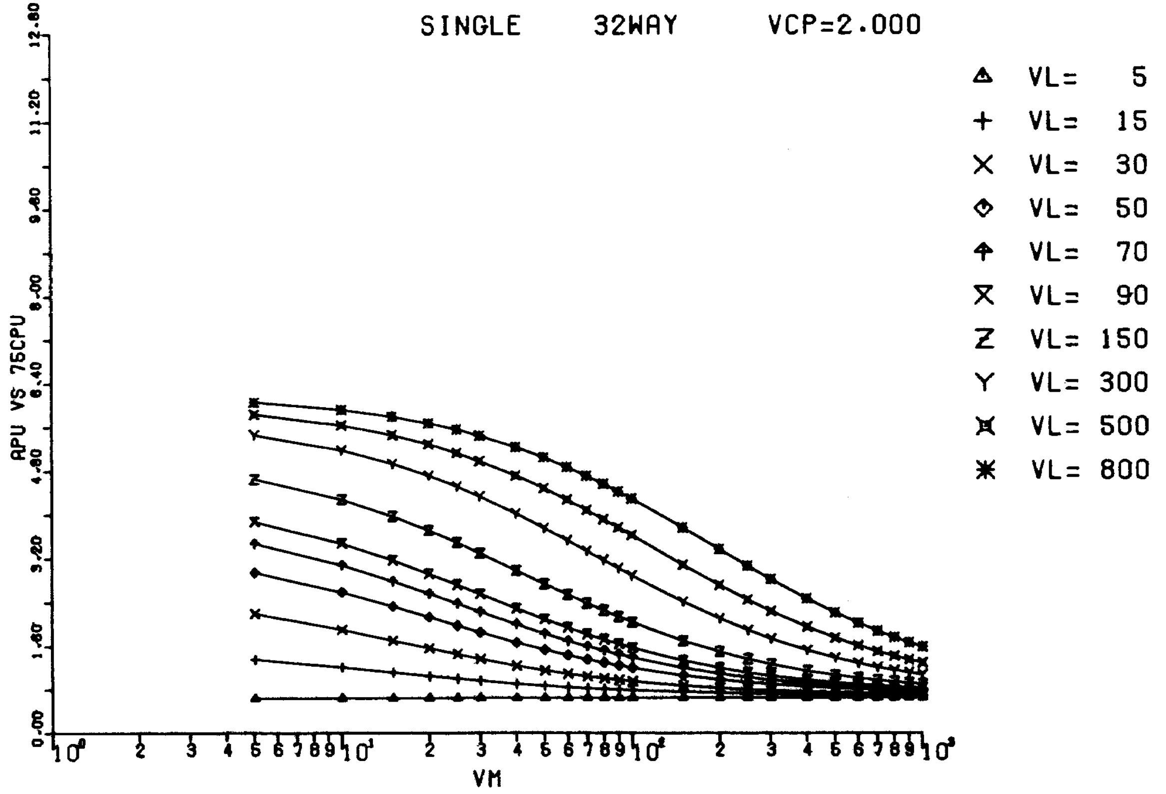



図3－1－3

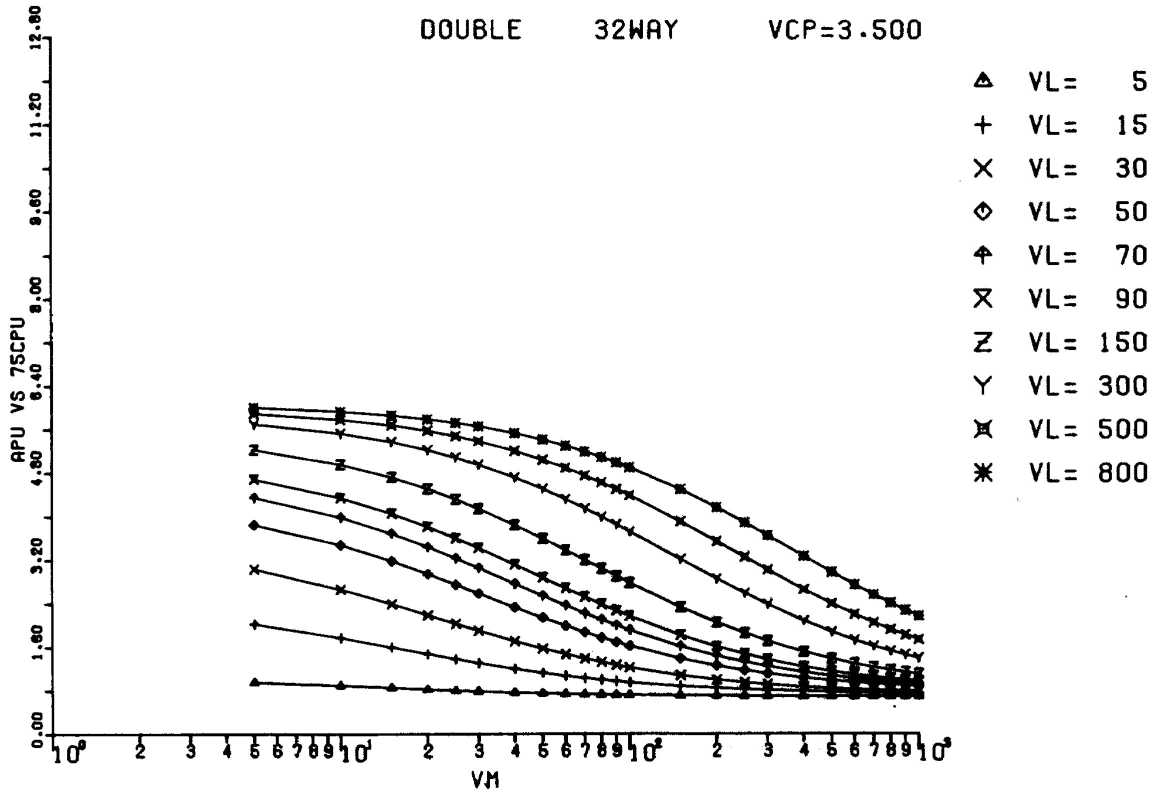



図3－1－4

This document is provided by JAXA.

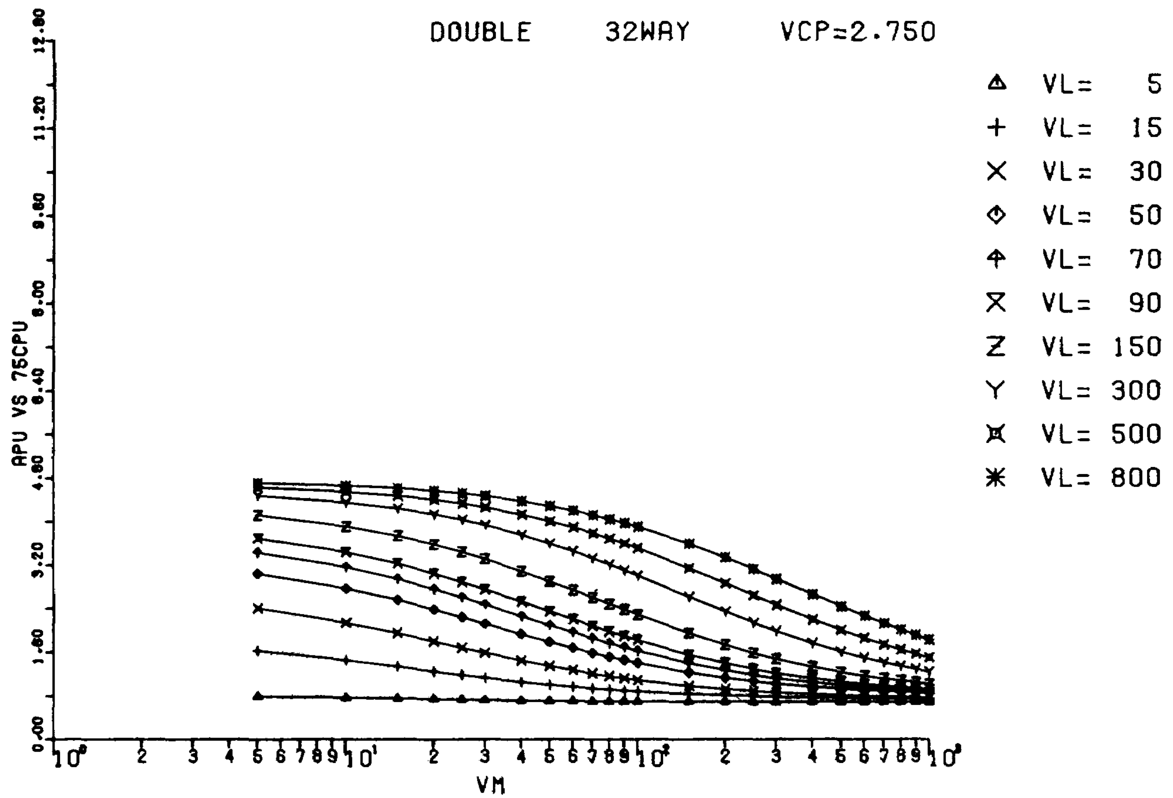



図3－1－5

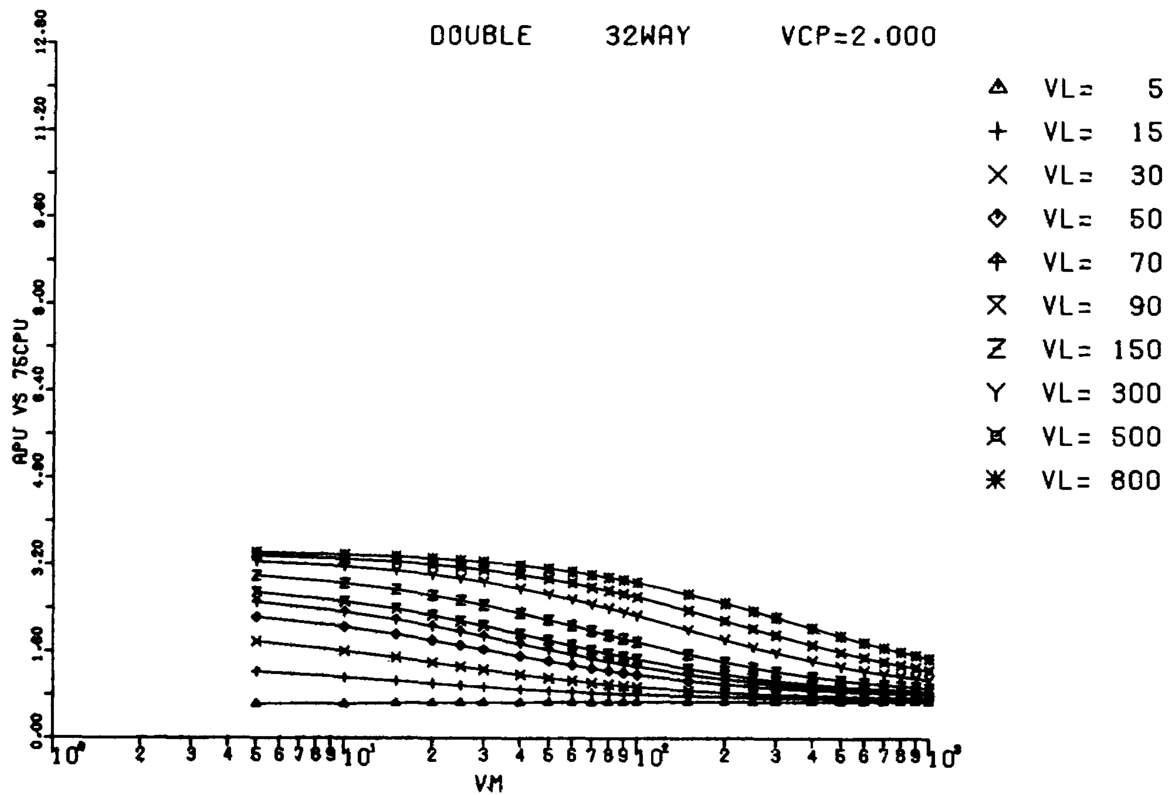



図3－1－6

This document is provided by JAXA.

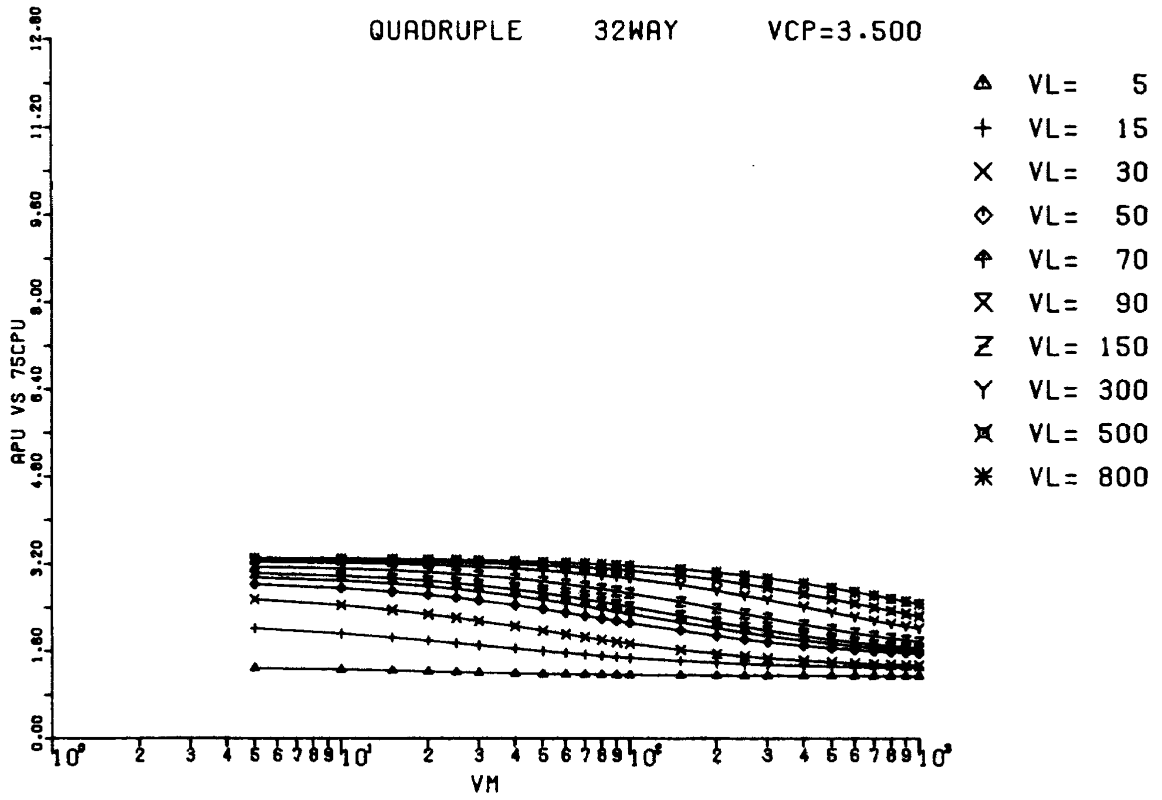



図3－1－7

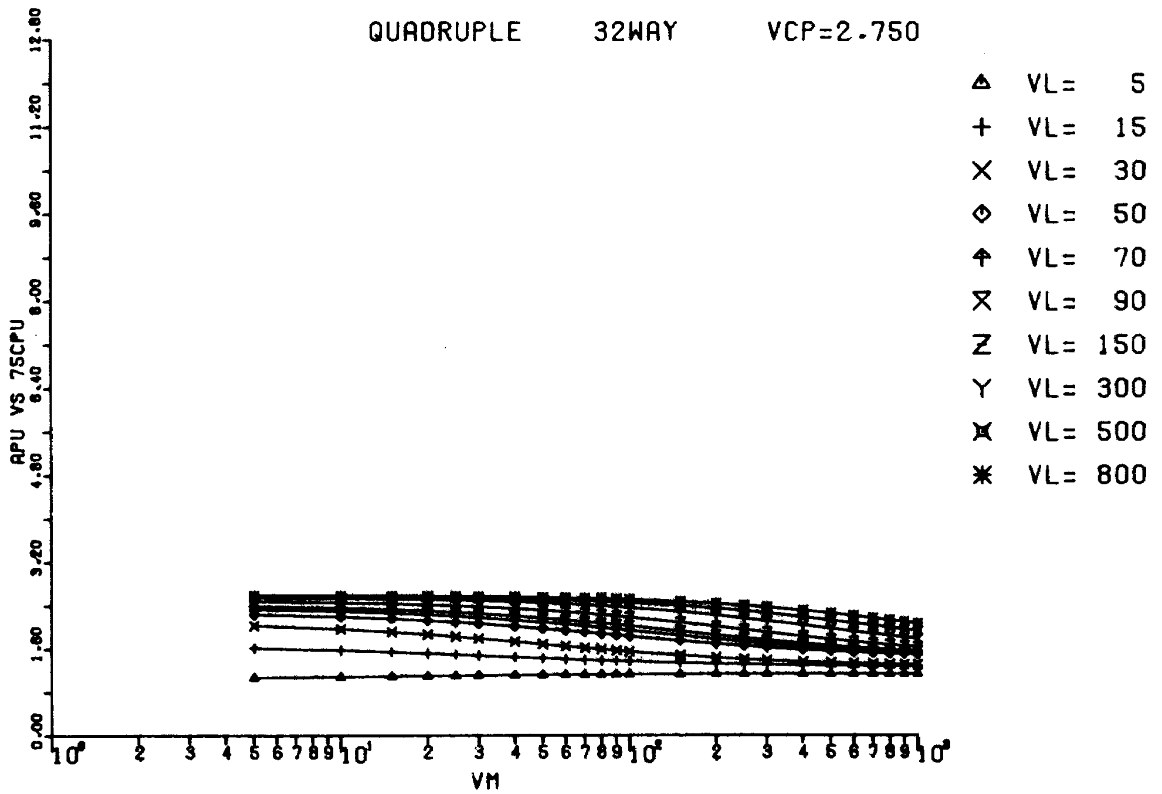



図3－1－8

This document is provided by JAXA.

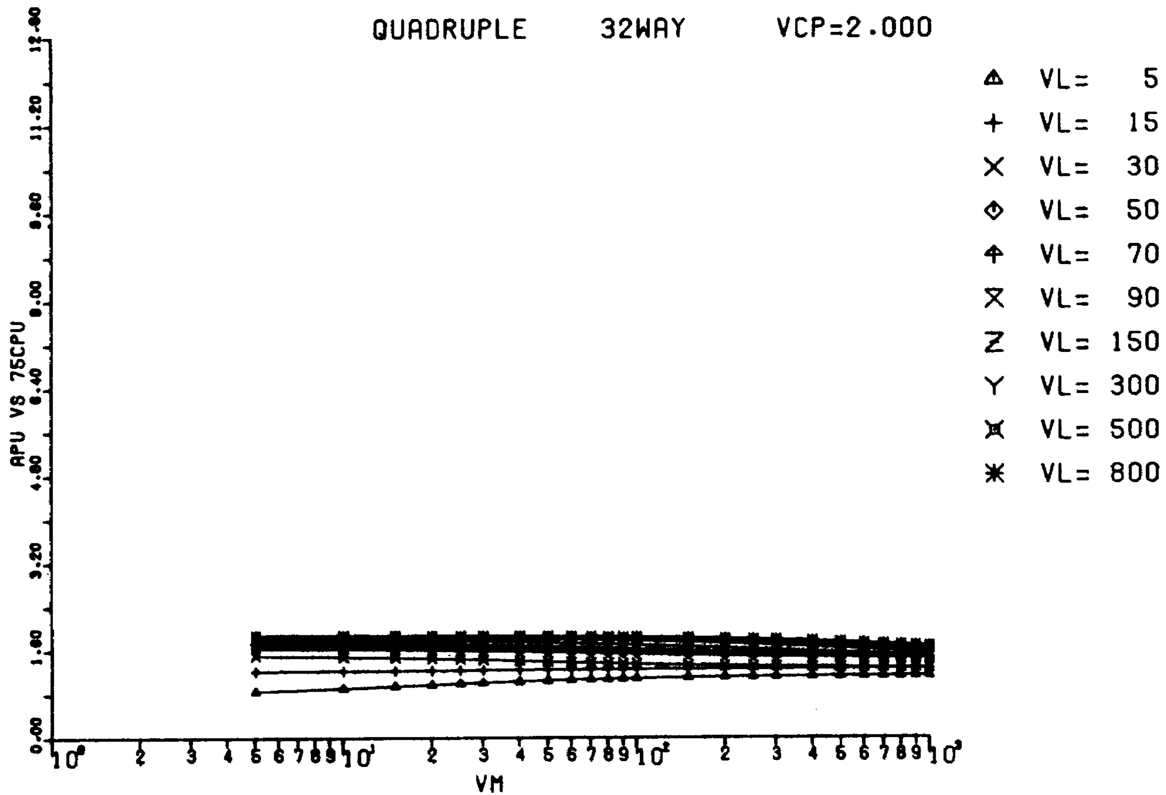



図3－1－9

VCPの値は出現頻度の大きなV型命令のグループをCPのDOループで書きなおし，そのアセンブリーリストを調べればそのプログラムで値は大体わかる。すなわち$N$，$\widetilde{M}/\widetilde{V}$ ，VCPの値からCPとAPの処理速度比は図3.1.1～図3.1.9から読み取ることができる。注2)

図3.1.1～図3.1.9の全てについて$N$はできるだけ大きい方が望ましい。以前の資料[1)]の各命令の実行速度曲線からも明らかな様に，ベクトル長 $N=5\sim50$ 位迄は実行速度が急上昇する所であり，$N$の値を大きくする様な努力が必要である。また$\widetilde{M}/\widetilde{V}$ は小さい程良いが，これは当然のことであるが，プログラムをできる限りベクトル化する必要があることを示している。以上の事項を達成するためのプログラム技術について述べるのが，本資料後半の主題である。

---

注2) この$\widetilde{E}_{AP}/E_{CP}$はあくまで大体の見当をつけるためのものであるから$N$，$\widetilde{M}/\widetilde{V}$，VCPの値にそれ程神経を使う必要はない。$N$は本文でも述べた様に大体わかるし，$\widetilde{M}/\widetilde{V}$ も或る程度馴れてくれば，それ程見積りに失敗することはないであろう。VCPはDOループの中にロード，ストア，トランスファ命令が多いと大きくなる。

## 3.2 プログラムレベルでのAPとCPの実行速度についての実測結果

仮運用期間中に航技研，富士通双方によりいくつかの実際使用されているプログラムをAP用に作りなおし，APとCPの実行速度の比較をフルスケールのプログラムに関して行なったが，その結果を表に示す。

表3.2.1の問題群は最後のラプラス方程式の場合を除けば平均的にCPの5倍程度の処理能力を示している。それに対して表3.2.2の問題群でのAPの処理能力はCPの半分から12倍強迄と大きなばらつきを示している。その理由は以下に述べる点にある。

(1) 表3.2.1の問題群はフルスケールといってもサブルーチンレベルのものであり，データの構造も整っている。さらにその計算法は周知のものであるためベクトル表現への変換も簡単であった。ラプラス方程式も計算法とコーディング技術を考えればもっとAPの実行速度を速めることができるが，CP向きに書かれたプログラムをそのままAPベクトル記述に変換したため実行速度はそれ程速くならなかった。

(2) 表3.2.2の問題群は一般の計算機利用者がCPで計算しているものを提供してもらってプログラムの変換を行なったものであるが，それらは3つのグループに分

This document is provided by JAXA.

表3.2.1

| 問題 | 精度 | 要素数 | 実行速度実測値(単位秒) | | 実行速度比 CP/AP |
|---|---|---|---|---|---|
| | | | AP | CP | |
| 固有値QL法 | 単精度 | 200 | 11.7 | 59.1 | 5.1 |
| | | 500 | 117.8 | 998.1 | 8.5 |
| | 倍精度 | 200 | 17.2 | 76.9 | 4.5 |
| | | 500 | 190.9 | 1255.8 | 6.6 |
| 固有値QR法 | 単精度 | 256 | 138.5 | 630.2 | 4.6 |
| | | 1296 | 516.0 | 3000.2 | 5.8 |
| ガウスサイラル反復法 | 単精度 | 21 | 40.4 | 134.7 | 3.3 |
| | | 36 | 47.2 | 210.5 | 4.5 |
| 固有値反復法 | 単精度 | 900 | 1.0 | 21.3 | 21.3 |
| 逆行列 | 単精度 | 400 | 15.6 | 108.7 | 7.0 |
| 連立一次(ガウス消去) | 単精度 | 400 | 21.0 | 84.4 | 4.0 |
| 熱伝導方程式(一次元) | 単精度 | 500 | 13.1 | 99.3 | 7.6 |
| ラプラス方程式(二次元) | 単精度 | 100 | 30.9 | 78.2 | 2.5 |

表3.2.2

| 問題 | 要素数 | 実行速度実測値(単位秒) | | 実行速度比 CP/AP |
|---|---|---|---|---|
| | | AP | CP | |
| 1 | 700 | 132.9 | 670.6 | 5.1 |
| 2 | 72 | 821.3 | 3271.7 | 4.0 |
| 3 | 144 | 1132.9 | 6527.1 | 5.8 |
| 4 | | 1048.1 | 3093.7 | 3.0 |
| 5 | | 71.7 | 913.5 | 12.7 |
| 6 | 49~2401 | 4.7 | 39.8 | 8.5 |
| 7 | 199 | 2.8 | 15.9 | 5.7 |
| 8 | | 8.1 | 31.3 | 3.9 |
| 9 | 121~161 | 492.4 | 673.1 | 1.4 |
| 10 | 200 | 8.1 | 8.3 | 1.0 |
| 11 | | 1691.2 | 2403.8 | 1.4 |
| 12 | | 7.4 | 7.5 | 1.0 |
| 13 | | 196.7 | 179.0 | 0.9 |
| 14 | 35 | 637.7 | 399.0 | 0.6 |
| 15 | | 1105.7 | 697.0 | 0.6 |
| 16 | | 551.9 | 272.4 | 0.4 |
| 17 | | 27.9 | 151.1 | 5.4 |

けられる。第一のグループは問題1，2，3，17であってプログラム変換者は研究者に問題と計算法を解説してもらってプログラム構造に迄手を入れて変換を行なったものである。特に問題17はデータの構造にも手を入れて変換したもので全く新しいプログラムであるといえる。データの構造に手を入れなければ問題17の実行速度はCPと同程度にしかならなかった。第2のグループは問題4，5，6，7，8であって，プログラム変換者は問題の内容も計算法も知らず，プログラムの構造もデータの構造も変えずにCP用に書かれたプログラムの中で，ベクトル演算で表現できるところだけをベクトル演算に変換したが，問題の解法とプログラムがもともとAPにも適していたため，その様な初歩的な変換でも実行速度はかなり上っているものである。これ等の問題は計算法を知りプログラムの構造とデータの構造をAP向きにさらに洗練させれば，その実行速度はまだ向上すると思われる。第3のグループは問題9～16であってコーディングレベルでは全くAP向きでないものからなっている。従って問題の内容も計算法も知らない変換者はプログラム構造にもデータ構造にも手がつけられなかったものである。このグループは実際にはさらに3つのグループに分けて考えるべきである。第一のグループは本質的にAP向きでないもの，例えば常微分方程式の初期値問題の様なものからなる。第2のグループとして，問題はAP向きであるがコーディングレベルでの計算法がAP向きでないものからなる。第3のグループは問題もコーディングレベルでの計算法もAP向きであるが，プログラムの構造とデータの構造がAP向きでないものよりなる。問題17は，初めはこの第3グループに属していた。そしてこの第3グループに属しているものが予想外に多いということが沢山の利用者プログラムから明らかとなった。研究者の多くはこれ迄のCPになれていて，少しずつデータを取ってはサブルーチン形式で計算を行ない，それをループで繰り返す習慣がある様であってハードウェアの構造を認識してプログラムを組む習慣をも

This document is provided by JAXA.

っていない。このことは計算機がその方向を目指してこれ迄進んできた以上やむを得ないことであるが，実際には現在一般化しつつあるキャッシュメモリ，バーチュアルメモリの技術を考慮するならばこれは得策ではない。ハードウェアの構造を考慮したプログラム，計算技術を習得することは利用者に大きな利得をもたらすのであって，研究者にとっては半ば義務でさえあると考える。ＡＰがＣＰの５倍程度の処理能力を発揮しうるということは，昭和５７～５８年頃に我々に提供される最高の国産計算機の処理能力がＣＰの高々５倍程度であることを考えるならば，多少従来の計算機と違って勝手は悪くともこれを使いこなすため，いささかの努力をはらうことは進められても良いものと考える。ＡＰの処理能力を最大限に発揮させるために必要なことは，データの構造とその処理手順がＡＰに向くものであれば，それに合わせてプログラム構造を考えることが根本的に重要であって，本来ＣＰを意識して書かれたプログラムの部分を取りだしてＡＰ用に変換することは，さしてＡＰの能力を発揮させることにはなり得ない。このことはそれ程困難なことではないのであって従来のデータとその処理手続きを局所的に見ていたのを大域的に見るということであり，それに合わせてサブルーチン等の構造も考えなければならないということである。

## 4　ＡＰに適したプログラム構造

### 4.1　科学技術計算プログラムの一般的構造とＡＰプログラム

科学技術計算のプログラムは基本的には繰り返し構造をもつ処理装置リミットのプログラムとして特徴ずけることができる。これは２３０－７５処理装置を１秒使う小さな計算であっても平均的に約３５０万個の機械命令を実行するのであって，いま仮にＦＯＲＴＲＡＮ１ステートメントが機械命令１０に対応すると多少，多めに見積もってもこれはＦＯＲＴＲＡＮ３５万ステートメントに対応する。航技研のプログラムの平均的な大きさが，ステートメント数で４００～５００であることを考えれば３５万ステートメントの実行は１ステートメント当り，７００回の繰り返しに対応することになる。実際には繰り返しは全プログラム・ステートメントの一部に集中する傾向にあることと，計算時間が１秒というのは標準からかけはなれて小さいことを考え合わせると通常の計算プログラムにおける同一ステートメントの繰り返し数は非常に大きな数になる筈であり，これらのステートメントにより処理されるオペランドの数もまた非常に大きくなる。いまＦＯＲＴＲＡＮステートメントのうち主要部分を占める演算ステートメントを取り出すと，それは一般的に

$$y=f(x_1, x_2, \cdots\cdots, x_n) \qquad (4.1.1)$$

の形をしている。$x_1, x_2, \cdots\cdots, x_n$ は演算$f$のソースオペランド，$y$はシンクオペランドである。(4.1.1)式は実行において

$$y=f(x) \qquad (4.1.2)$$

または

$$y=f(x_1, x_2) \qquad (4.1.2)'$$

の形の単位演算の形に分解して実行されると考えて良い。例で示すと

$$A=B+C*D**2 \qquad (4.1.3)$$

という演算ステートメントは

$$D'=D**2$$
$$C'=C*D' \qquad (4.1.3)'$$
$$A=B+C'$$

と(4.1.2)，(4.1.2)′式の形に分解して処理される。いま全ての演算を単位演算に分解し，サブルーチン等も実行順序に従って組込んだ形で科学技術計算のプログラムの構造を(4.1.2)式の記号(簡単のためであって(4.1.2)′式の方が一般的である)を使って示したものが図4.1.1である。演算は$f_a$……$f_m$で表わし，演算$f_k$のオペランドは $\{x_k^i\}_{i=1}^{n7}$，$\{y_k^i\}_{i=1}^{n7}$ の様に表わしてある。Ｕに判断分岐を表わし，繰り返しの数は$n_i$で表わしてある。繰り返しはＦＯＲＴＲＡＮではＤＯ文で表わされることが多いが，ＩＦ文等の判断分岐を使って表わされることもある。演算フローの分岐はＩＦ文を使って表わされることが多いが種々のGO TO文を使って表わすこともある。演算$f$の静的な(実行回数ではない)数はＦＯＲＴＲＡＮ１ステートメント当り５つと数えると航技研の場合１プログラム当り平均的に2000～3000程度であろう。図4.1.1の様に2000～3000程度の演算が繰り返し実行されることにより，実際にプログラム走行の際に実行される演算の数は$10^7$～$10^{10}$のオーダに達し，演算の対象となるオペランドの数も$10^7$～$10^{10}$のオーダに達する。いま図4.1.1における繰り返し演算ループ$f_n$についてこれがＤＯ文で以下の様なものであるとする。

```
      DO 1 I=1, 10
      A(I)=B(I)+C(I)*D(I)**2        (4.1.4)
    1 E(I)=A(I)*F(I)
```

これを(4.1.3)′の形でかくと

```
      DO 1 I=1, 10
      D'(I)=D(1)**2      ; f_k       (4.1.4)'
      C'(I)=C(I)*D'(I)   ; f_k+1
```

This document is provided by JAXA.

$$A(I) = B(I) + C'(I) \quad ; f_{k+2}$$

$$1\ E(I) = A(I) * F(I) \quad ; f_{k+3}$$

となる。この様にかかれた演算を計算機は$f_k$(D(1)), $f_{k+1}$(C(1), D′(1)), $f_{k+2}$(B(1), C′(1)), $f_{k+3}$(A(1), F(1)), $f_k$(D(2)), …… $f_{k+3}$(A(10), F(10), と実行するが$f_k$(D(1), …… $f_k$(D(10)), …… $f_{k+1}$(C(1), D′(1)), $f_{k+2}$(C(2), D′(2)), $f_{k+1}$(C(10), D′(10)), $f_{k+3}$(A(10), F(10)) と実行しても演算結果は変らない。この場合，記号的に

$$\left.\begin{aligned} & f_k(X^i_{k,1}\, X^i_{k,2}) = (f^1_k, f^2_k \cdots\cdots f^1_k) \\ & f^i_k = f_k(x^i_{k,1}, x^i_{k,2}) \end{aligned}\right\} \quad (4.1.5)$$

とかけば(4.1.4)′は$f_k$, $f_{k+1}$, $f_{k+2}$, $f_{k+3}$と実行しても良いことになりベクトル演算化できることになる。APハードウェアの実行速度についての議論で述べた様にAPのベクトル演算実行速度は一般的にベクトル演算の対象となるベクトルの要素数が大きい程速いのでAPの演算能力を最高度に発揮させるためにはできる限り多くの演算をベクトル化し，そのベクトルオペランドの長さをできるだけ長くすれば良いということになる。図4.1.1からこれは比較的容易であって，演算のベクトル化において問題となるのは許容主記憶量のみである様に見える。例えばループ$f_k = \cdots f_{i-1}$ とベクトルオペ

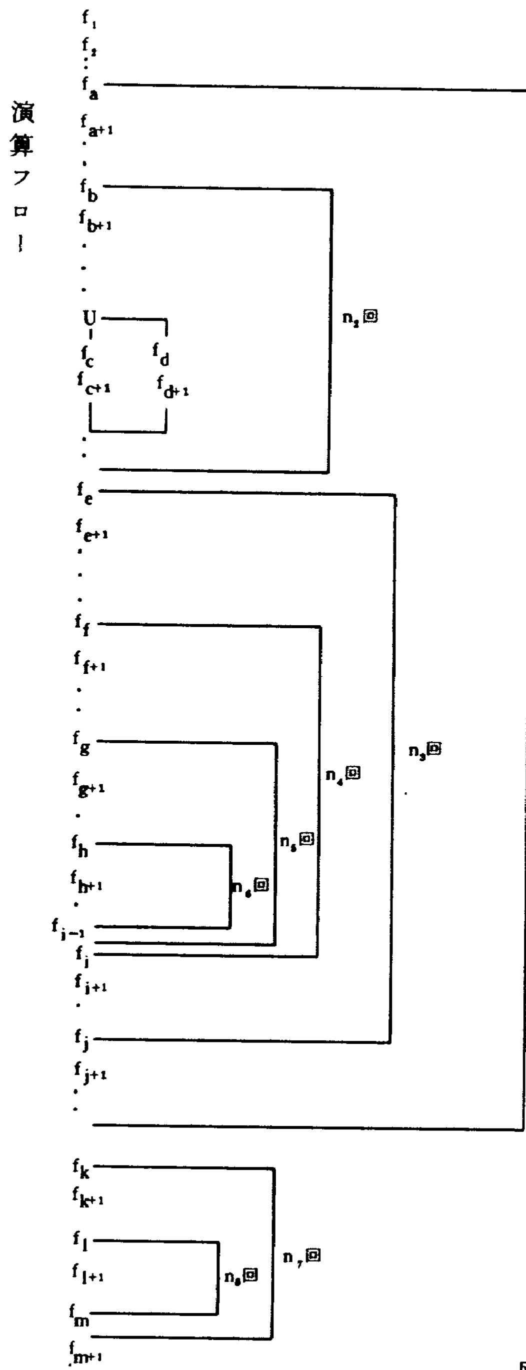



図 4.1.1

This document is provided by JAXA.

ランド長$n_1$・$n_2$・$n_3$・$n_5$・$n_6$をもつベクトル演算に帰着できる様であるが一般的にはこれは不可能である。その理由はプログラムの繰返し構造に2つの型があることによる。第1の型は同時並行処理可能な繰返し構造（以下P型と略記する）であり，他の1つは同時並行処理不可能な繰返し構造（以下NP型と略記する）である。P型の最も基本的な型の例として（4.1.4）式があり，NP型の例として

```
     F(1)=C
     DO1 I=1, M                     (4.1.6)
  1  F(I+1)=A*F(I)+B
```

がある。（4.1.6）がNP型であることはF（I+1）の計算とF(I)の計算には順序があり，F（I+1）とF(I)を同時に並行処理することができないことから明らかであり，（4.1.6）の演算のベクトル化は困難である。（しかし，いわゆるツリーの高さを低くする方法によってベクトル化できるが，それについては後の章で述べる。）これで繰返し構造に2つの型があり，1つはベクトル化可能であり，他はベクトル化が困難であることがわかったが，プログラムの繰り返し構造がP型であるかNP型であるかということはプログラムの局所的な性格であるだけでなく大域的な性格でもあるということが重要である。（4.1.6）は局所的にはNP型であるが大域的には必ずしもそうではない。

いま，図4.1.4の$f_k$−ループと$f_l$−ループが

```
     DO 1 I=1,N                                  ┐
     A(I)=B(I)+C           f_k                   │
     D(I)-G*B(I)           f_k+1                 │
     F(I,1)=E(I)           f_k+2                 │ ループk
     DO 2 J=I,M                      ┐           │
     F(I,J)=A(I)+F(I,J-1)+D(I)  f_l  ┘ ループl   │
  2  CONTINUE                                    │
  1  CONTINUE                                    ┘
                                          (4.1.7)
```

の様に表現されたとするとループ$l$それだけではNP型でループ$k$はP型である。このとき $f_k$ $f_{k+1}$ $f_{k+2}$ $f_l$ をAP−FORTRANのベクトル記号を用いて以下の様に表現できる。

```
     A(*)=B(*)+C                 ; f'_k
     D(*)=G*B(*)                 ; f'_k+1
     F(*,1)=E(*)                 ; f'_k+2
     DO 1 J=2,M
  1  F(*,J)=F(*,J-1)+D(*)        ; F'_l
                                          (4.1.7)'
```

すなわち（4.1.7）は全体としてベクトル長NをもつP型の繰り返し構造となる。

```
     DO 1 I=1,M                              ┐
     A(1)=CA                                 │
     A(2)=CB                                 │
     DO 2 J=1,N      ┐                       │
  2  B(J)=A(J)       ┘ ループK1              │ ループl
     DO 3 J=2,N-1                            │ (4.1.8)
  3  A(J)=B(J)+X*(B(J-1)                     │
          -2.0*B(J)+B(J+1))                  │
                         ;ループk2           │
  1  CONTINUE                                ┘
```

はループ$k_1$，$k_2$ともにP型であるがループ$l$で考えるとベクトル長 N×M のP型構造にはできない。従って（4.1.8）はベクトル長Nに対してはP型構造であるがベクトル長 N×M に対してはNP型構造である。

この節でのべてきたことを念頭におけばAPに適したプログラムを作成するためには

1. プログラムの大域的，局所的な繰り返し構造を考慮してできうる限り多くの演算をベクトル演算化すること。

2. サブルーチン等は通常局所的な見地から作成されているがこれも大域的な見地からベクトル演算化すること（引数にはベクトルを用いる。）

3. 繰り返し構造の最も内側（図4.1.1でいえばループ$f_h$）のベクトル演算の長さを最大にすること。

4. 繰り返し構造の中の演算分岐は演算のベクトル化にとって問題となるがこれ等の処理に各種論理型，比較型の配列特殊関数，リストベクトル演算等を最大限に利用すること。

等に留意することが必要である。

表3.2.2の問題9〜16には上記の様にプログラム全体の繰り返し構造を把握した上でプログラムをAP用に作りなおせば実行速度を速めることができるプログラムがまだ可成りある。

APプログラムの作成において附随的ではあるが重要な問題として以下のものがある。

i）プログラム作成の効率をあげるためにプログラムのサブルーチン化は進められるべきであるが実行ステートメントが極く少数のサブルーチンを数多く作るのは奨められない。コンパイラはサブルーチンの数だけ呼ばれるためコンパイルジョブステップが入出力リミットのジョブステップであることと相まってそのための主記憶時間は無視し得なくなりジョブのターンアラウンドタイムを引き延す結果となる。

ii）入出力を含めてAPプログラム中にCPを要求す

This document is provided by JAXA.

るサブルーチン等が頻繁に呼ばれるとＡＰ－ＣＰ通信のためのモニタオーバーヘッドが激増し，この場合，そのジョブ自身の実行も殆んど止まってしまうだけでなく，他のジョブの実行も殆んど停止させてしまうことになるのでプログラム上必要なＣＰで実行する部分はできるだけまとめて実行させることが必要である。

最后にＡＰプログラムの構造とコーディング技術とは異種の問題であり，本資料の範囲をこえる問題であるためこれ迄議論の対象としなかった数値計算法について簡単に述べておく。

或る問題を解く場合，如何なる数値計算法を選択するかということは或る程度問題により規定されてしまうことは事実であるが複数の解法の可能性があることは通常である。その際，研究者の計算法，もっと狭い所では計算スキームに対する選択の基準は以下の様なものであろう。

(1) 実践的に確実に解が得られる見込みのたつもの。

(2) できるだけ経済的に高い精度の解が得られるもの。

(3) 手慣れた解法

(1)は必要条件であって動かせないが可能性としては復数の解法がこの条件を満足するのが通常である。(2)，(3)についてはＡＰの様なハードウェアの出現はこれ迄の習慣を再検討することを要請している。計算機の型式の変動によって解法の再検討の必要性が生ずるという様なことは本末転倒である様だが，現在の計算機の出現と発展がディスクリート法の様にそれ迄あまり実践的にはかえりみられなかった解法をクローズアップさせ，それに伴なう数値計算法，数値解析の今日の隆盛を現存せしめたことを考えるならばこれはそれ程驚くにはあたらないであろう。ＣＰでは非経済的な解法がＡＰでは経済的な解法であることもしばしば生ずるのである。

### 4.2 前節の一般論についての具体的例による補足

前節においてＡＰに適したプログラムを作成するにあたってのいくつかの注意事項について述べた。これらの注意は一言でいえばプログラムの全体の構造をＡＰに適したものとすることに留意すべきであってプログラムを局所的にベクトル化することに努力することより大域的にＡＰに適したものとすることに努めるということになるが，本節では２，３の具体例をあげてプログラム構造についてのほんの僅かの注意がどれくらい効果があるかを示すことにする。

例1. 繰り返しの最も内側のベクトル演算のオペランド長を大きくすることの効果

有限要素法では Ｍ×Ｎ 行例；Ａ（Ｍ，Ｎ）とＮ×Ｌ行列Ｂ（Ｎ，Ｌ）の積が頻発する。ここで，Ｎは３～６でＭ，Ｌは数十の計算が含まれている。いま，一般性を失なうことなく Ｍ＞Ｌ とする。（Ｌ＞Ｍの場合は転置して考えればコーディングは異なるが考え方は同じである。）またコーディング，速度の点では Ｌ＝１ の場合で考えれば同じであるので Ｌ＝１ とする。通常は行列積の公式により次の様にコーディングするのが自然である。

```
   DO 1 I=1,M
   C(I)=0
   DO 1 J=1,N                        ( 4.2.1 )
 1 C(I)=C(I)+A(I,J)*B(J)
```

ベクトル記法で（4.2.1）をそのままかきなおすと

```
   DO 1 I=1,M
 1 C(I)=IPD(A(I*),B(*))              ( 4.2.1.)'
```

もっと速いのは，

$$C(*)=PMM(B(*),\overline{A(I,*)}) \qquad (4.2.1)''$$

ここで $\overline{A(I,*)}$ は転置である。（4.2.1）～（4.2.1）″は自然であるがＡＰはベクトルが長い程速いこととＮが３～５，Ｍが２０～５０であることおよび（4.2.1）はＭとＮを入れかえても結果は同じにできることを考慮すると次の様にもかける

```
   DO 1 I=1,M
 1 C(I)=A(I,1)*B(1)+…+A(I,N)*B(N)
                                     ( 4.2.2 )
```

これをベクトル記法でかけば

```
 C(*)=A(*,1)B(1)+…+A(*,N)*B(N)
                                     ( 4.2.2 )'
```

ベクトルレジスタを徹底的に使えばC(M)，A(M,N)，B(N)とW(M) をVRLOADして

```
  C(*)=A(*,1)*B(1)
  W(*)=A(*,2)*B(2)
  C(*)=C(*)+W(*)
  W(*)=A(*,2)*B(3)
  C(*)=C(*)+W(*)                     ( 4.2.2 )''
  ..................................
  W(*)=A(*,N)*B(N)
  C(*)=C(*)+W(*)
```

とかける。（4.2.2）′，（4.2.2）″のベクトル長はＭである。表 4.2.1 に（4.2.1），（4.2.1）″，（4.2.1）′（4.2.2）″による実行時間と（4.2.1）″と（4.2.2）″の実行速度比を Ｎ＝３ の場合について示している。（4.2.1）はＣＰで実行させているが，この場合実行時間はＭの大きさにほぼ比例して増大してゆくので Ｍ＝２０

This document is provided by JAXA.

表4.2.1　行列積と実行速度と速度比

| 計算方法 ＼ ベクトル長M | (4.2.1)CPで実行 | | (4.2.1)′ APで実行 | (4.2.2)′ APで実行 | (4.2.2)″ APで実行 | (4.2.1)′/(4.2.2)″ |
|---|---|---|---|---|---|---|
| | opt 0 | opt 2 | | | | |
| 10 | | | 35.0μs | 29.2μs | 21.6μs | 1.6 |
| 20 | 384μs | 145μs | 62.5μs | 32.8μs | 25.2μs | 2.5 |
| 50 | | | 144.9μs | 43.6μs | 36.3μs | 4.0 |
| 100 | | | 282.2μs | 62.0μs | 54.4μs | 5.2 |
| 200 | | | 556.9μs | 100.8μs | 90.6μs | 6.1 |
| 300 | | | 831.2μs | 238.2μs | 126.6μs | 6.6 |

の場合についてのみ示してある。Ｍ＝５０の場合は2.5倍，Ｍ＝２００で表の値の１０倍の時間がかかるとみて良いであろう。表から，行列積に関してＡＰはＣＰに比較していずれの場合も速いがＡＰにより実行させた場合でもベクトル長を大きくとることがいかに大事であるか，そしてベクトル長を大きくするためにほんの少しの細工も結果に大きな利得をもたらすということが明かである。同様な事例はいくらでも考えられるわけであって，上の例ではＭの長さに目をつけたわけであるが，Ｍが小さくても同時に行いうる行列積の個数が大きければその個数に着目して計算を行なえば同様な結果が得られる。すなわち４×４の行列の積を５０個行う場合，行列をたとえばＡ(Ｉ，Ｊ，Ｋ)，Ｂ(Ｉ，Ｊ，Ｋ)；Ｉ＝Ｊ＝４，Ｋ＝５０としたときＡ×Ｂ＝Ｃにおける演算をＫ方向に行なえばベクトル長５０のベクトル演算に帰着させることができる。これは学識的に

```
DO 1 K=1,50
DO 2 J=1,4
DO 3 I=1,4
………………………
```

とかくところを

```
DO 1 J=1,4
DO 2 I=1,4
DO 3 K=1,50
………………………
```

とかくことに相当し，もっとも内側の繰り返しの長さを最大にすることと同じである。

ここまで簡単な行列積の問題で述べてきた手法をフルスケールの問題に適用して成功したのが表3.2.2の問題１７であり，この問題は３次非圧縮ポテンシャル流のパネル法による計算プログラムであって，高速化はパネル毎に行なっていた計算を演算毎にまとめて行いうる様にデータの配置を考えなおすことにより実現した。

例２　ＮＰ型の繰り返しをＰ型の繰り返しに変換することによる効果

ＮＰ型の繰り返しの代表的なものとして

$$f_i=a_i f_{i-1}+b_i \qquad (4.2.3)$$

$$f_i=a_i f_{i-1}+b_i f_{i-2} \qquad (4.2.4)$$

がある。前に述べた様にこれはこのままではベクトル演算化できないし，(4.2.3)，(4.2.4)が常微分方程式の初期値問題，あるいはそれと類似の問題から生じたものであればベクトル演算しても効果は上らない。しかしながら，これが(4.1.7)のようにより大きなループの中にある場合には全体としてベクトル演算化することが可能である場合もある。いま，(4.2.4)をモデルにとりこれがより大きなループの中に以下の形で含まれている場合について考える。

```
   DO 1 I=1,M
   X=A(I)
   Y=B(I)
   F(1)=C(I)
   F(2)=D(I)                          (4.2.4)
   DO 2 J=3,100
 2 F(J)=X*F(J-1)+Y*F(J-2)
………………………………………………
```

この場合，Ａ(Ｉ)，Ｂ(Ｉ)，Ｃ(Ｉ)，Ｄ(Ｉ)がＦの値と無関係であれば次の様にベクトル化できる。

```
   F(*,1)=C(*)
   F(*,2)=D(*)
   DO 1 J=3,100                      (4.2.5)
 3 F(*,J)=A(*)・F(*,J-1)
          +B(*)*F(*,J-2)
```

この場合のベクトル長はＭである。表4.2.2はＭの長さを変化させた場合のＣＰとＡＰの実行時間と実行速度比について示したものである。表に示されている実行時間は繰り返し $\{f_i=af_{i-1}+bf_{i-2}\}_{i=3}^{100}$ １回当りのものである。表からも明かな様にＭの数が大きくなるに従いＡＰのＣＰに対する実行速度比は急激に上昇する。

This document is provided by JAXA.

表4.2.2　$f_i=af_{i-1}+bf_{i-2}$ の実行速度

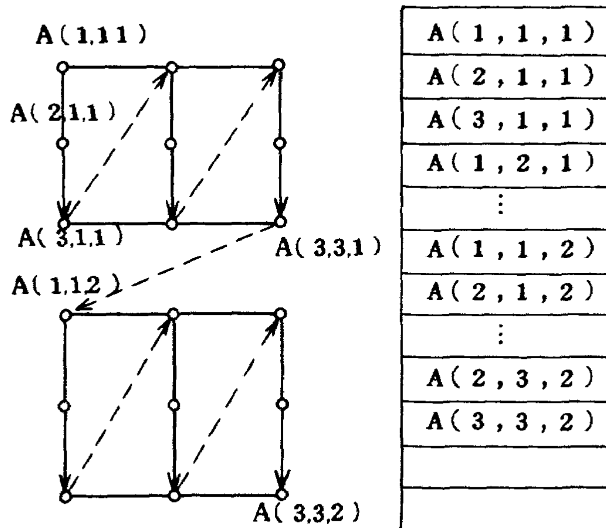
| ベクトル長M | (4.2.4) CP | (4.2.5) AP | 実行速度比 CP/AP |
|---|---|---|---|
| 5 | | 316.6μs | 1.5 |
| 10 | | 165.4μs | 2.8 |
| 20 | | 113.0μs | 4.2 |
| 30 | | 84.9μs | 5.5 |
| 50 | 467μs | 61.9μs | 7.6 |
| 70 | | 51.2μs | 9.2 |
| 100 | | 43.8μs | 10.7 |
| 200 | | 34.3μs | 13.7 |
| 400 | | 29.4μs | 16.0 |

## 5　APープログラム作成に知っておきたい基礎的事項

### 5.1　ハードウェアとコンパイラに関する事項

よりよいAPプログラムを作るためには，ハードウェア[2)]やソフトウェア[3),4)]についてある程度理解しているのが好ましい。以下にそのようなものの幾つかについて述べる。

APがこれまでのCPと本質的に異なっているのは，いわゆるベクトル命令というものを持っていることにある。それは必然的にDIMENSION宣言されたアレイ[注1)]を取り扱うので，FORTRANでDIMENSION宣言されたアレイが主記憶上でどう配置されるかを知っておく必要がある。AP−FORTRANでは１次元から７次元までのDIMENSIONが許されているが，結局そのどれもが主記憶上では１次元として取り扱われる。その配置のされ方を３次元アレイA（３，３，２）によって例示する。




APが取り扱うベクトルには基本ベクトルとリストベクトルの２種類がある。基本ベクトルとは，主記憶上で１次的な規則性をもった要素集合のことである。それは，全要素が隣接しているか（図5.1.1），或は等しい間隔で並んでいる（図5.1.2）ものである。図5.1.3のようなものは高次の規則性といい，これらの要素集合を基本ベクトルとして１回のベクトル命令で処理することはできない。次にリストベクトルとは処理の対象となっている要素の位置をリストと称するベクトルに作成することにより，間接的に要素集合を指定したものである。基本的にはいかなる要素集合でもリストを作成することによってリストベクトルとして扱える。特殊なものとして，同じ要素が何度も現われるようなベクトルも取り扱える。しかし，リストそのものの作成に時間がかかったり，或は相当量の主記憶を消費したりするのでは好ましくない。さらには，演算速度の速いベクトル命令では，リストベクトルの方が基本ベクトルよりも２〜４倍遅くなるという事実に留意する必要がある。高次の規則性をもった要素集合にベクトル演算を施すには，ＤＯ文を用いて基本ベクトルとして何回かの演算を施す方法と，リストベクトルとして施す方法の２通りがあり，前者の場合には演算の行なわれるベクトル長が大きくなるように注意する。このことを理解すれば，１つのアレイにインデックス変数を２つ以上使えないとか，“＊”との混在が許されな

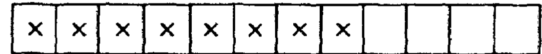



図 5. 1. 1

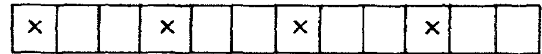



図 5. 1. 2

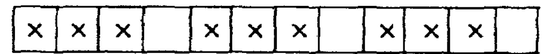



図 5. 1. 3

注1）以下，アレイと記した場合はFORTRAN（含AP−FORTRAN）でDIMENSION宣言されたものを指す。

This document is provided by JAXA.

いとか，2つ以上の“＊”は続いていれば許されるとかという制約の基が了解されるわけである。さらに不必要な大きさの多次元アレイをとらないということの意味もわかるであろう。又，プログラムテクニックの1つとして，2次元以上のアレイを（それと同じ大きさの）1次元アレイとEQUIVALENCEで結ぶことによってベクトル表現が可能となるものも生じるわけである。例えば，

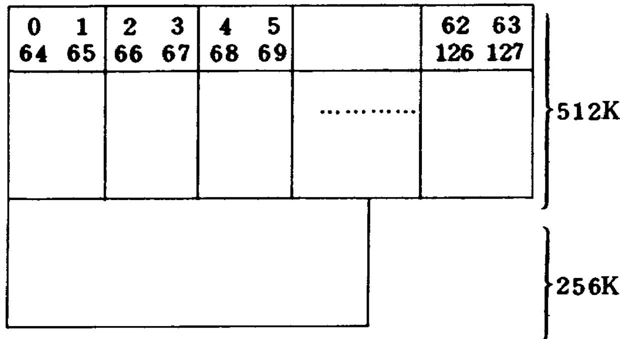
DIMENSION　A($n$, $n$), B($n^2$)
EQUIVALENCE　(A(1, 1), B(1))
INDEX　IX/1, $n^2$, $n+1$/

としておけば，Aの対角要素だけを処理したいときには，B(IX)によって実行することができるし，A($i$, $i+1$) $1 \le i \le n-1$ を処理したいときにはインデックスを

IX/$n+1$, $n^2-1$, $n+1$/

と変えればよい。

このアレイの配置と共に知っておくべき事は，主記憶のバンク構成である。主記憶へのアクセスのためのバスは32路あって（32ウェイという），1部16ウェイの所もある，絶対番地との関係は次図のようになっている。




これは，例えば1番地と3番地は同時にアクセス可能であるが，1番地と64又は65番地とは同時にアクセスできず，32ウェイの能力を発揮しないことになる。従って跳びのある基本ベクトルの場合，その跳びが$2^n$の形で単精度なら $n \ge 6$，倍精度なら $n \ge 5$ となるのは甚だ不都合である。このようなことはAP−FORTRANで

```
DIMENSION A(1024)
INDEX IX/1, 1024, 64/
A(IX)=…………
```

と陽に表現した場合だけでなく

```
DIMENSION A(64, 50)
A(I, *)=…………
```

の場合等にも生じることに留意しなければならない。

APの高速性を十分に発揮させるには大きいデータ供給能力を必要とし，従来の主記憶アクセス法では不十分な所もあるので，それを補うために1792語のベクトルレジスタなるものがある。このベクトルレジスタはFORTRANプログラムで使用可能であり，これを最大限に活用するようにコーディングすることが処理速度をさらに上げることにつながる。使用方法には2通りあって，1つはユーザが陽に使用する方法，もう1つはFORTRANコンパイラにその使用を任せてしまう方法である。後者がどのような使用方法をとっているかその形態について述べる。それを一言でいってしまえば，ベクトル演算を含む数式処理時のベクトルテンポラリ（以下略してVTと記す）領域としてである。このVTにも，コンパイル時に大きさのわかっているベクトルを扱うテンポラリ（以下略してSVTと記す）と，実行時になるまで大きさのわからないベクトルを扱うテンポラリ（以下略してDVTと記す）の2種類がある。具体的に話をすれば以下のようになる。例えば，

```
DIMENSION A(50), B(50), C(50),
          D(50), E(50)
A(*)=B(*)+X*C(*)+D(*)*E(*)
```

というプログラムの場合の数式処理手順は，

```
SVT1(*)=X*C(*)
SVT1(*)=B(*)+SVT1(*)
SVT2(*)=D(*)*E(*)
A(*)=SVT1(*)+SVT2(*)
```

と分解され，ベクトルSVT1とSVT2がベクトルレジスタ上にとられる。DVTが現われるのは，上例だと“＊”の代わりに

INDEX　IX/1, N/

としたインデックスを用いたり，DIMENSION宣言が定数でなく変数になったりしている場合等である。この場合にはさらに，SVT1とSVT2の間にM&R型命令が入ってV型命令が連続しなくなる。各領域間の関係は図5.1.4のようになる。実際にベクトルレジスタに

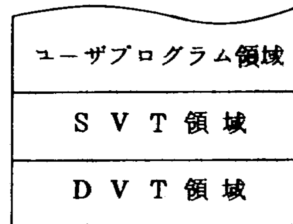
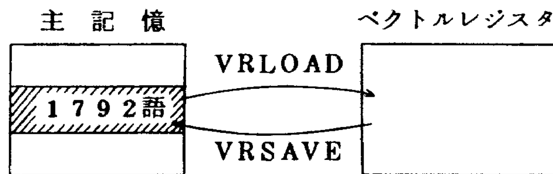



図5.1.4

ロードされるのはSVT領域の先頭から1792語であり，FORTRANコンパイラは無条件にユーザプログラ

This document is provided by JAXA.

ム領域の後に2K語の領域を確保してしまう。これからもわかるように，FORTRANコンパイラにベクトルレジスタの使用をまかせるのであれば，数式表現はなるべく長く書く方がよく，自分で定義したVTへのストアは必要最小限にとどめる。なお，コンパイラによるこのような使用方法は限られたものであってベクトルレジスタの能力を十分いかしきれていない。一方自分でベクトルレジスタを使用する場合には，数式表現はできるだけ細かく分けて（最小単位にまで），VTがベクトルレジスタ上にあるようにして，そこで演算が続けられるようにするのが好ましい。又，テンポラリ領域としてのみ使用するのではなく，使用頻度の高いデータベクトルをもベクトルレジスタ上にあるようにすると一層効果が上がる。主記憶とベクトルレジスタの関係は次のようである。

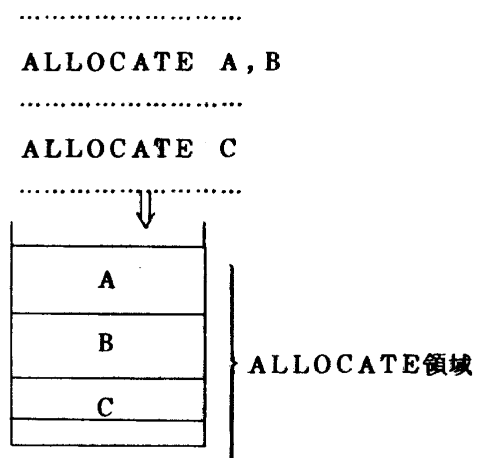



AP−FORTRANのVRLOADというサブルーチンにより，主記憶内の指定された1792語の内容がベクトルレジスタにロードされ，以後この領域に対するアクセスはモニタモードでのM&R型の書き込みが主記憶とベクトルレジスタの両方に書き込む以外すべてベクトルレジスタに対して行なわれる。そのために，ベクトルレジスタ上で変更された値を主記憶へ戻すには，AP−FORTRANのVRSAVEというサブルーチンによる。ベクトルレジスタと対応のついているこの領域をWRITEしたり，この領域にREADしたりするとCPへタスクスイッチが行なわれ，そのときに自動的にシステムによってベクトルレジスタの内容と当該領域の内容との両立性がとられる。なお，このVRLOAD, VRSAVEのためにはそれぞれ大略100μsecの時間がかかる。

もう1つの大事な点は，APとCPとの間で不必要な通信を行なわないということである。AP−CP通信を頻繁に行なうとシステム全体のオーバヘッドを増し，自身のジョブの進行のみならず他のジョブの進行にも悪影響を及ぼす。APにはI/O処理ルーチンと文字処理ルーチンとがないので，この2つに該当するFORTRANステートメントはCPによって処理され，AP−CP間の通信が行なわれる。それ故，I/Oはできる限りまとめてだすようにして，DO文の中で単純変数などをREADしたりWRITEしたりせず，VTを経由して実行するのが好ましい。なお，FORTRANステートメントで連続したI/Oは1回のAP−CP通信しか起こさないようにコンパイラの方で処理してくれる。その他，どうしてもCPで処理したい部分（I/Oを除く）がAP−プログラムの中で幾ステートメントにもわたって現われるときには，その部分だけをまとめてCP−サブプログラムにしてしまい，一括してCPに処理を任せるというのも1つの方法である。

AP−FORTRANには領域をダイナミックに確保したり解放したりする機能が用意されている。領域を確保するにはALLOCATE文を用いる。ALLOCATE文の中に現われるアレイは，その文が実行されない限り実領域の割り当てが行なわれない。このことに注意しないと領域侵害等を起こす。このALLOCATE配列のための記憶域は実行時のキィパラメータ"AREA=($n_1$, $n_2$)"で1次量$n_1$と2次量$n_2$を指定する。1次量の分の大きさだけはそのプログラムが実行される最初の段階で確保される。このことはDVT領域のためにも必要である。プログラムの実行中に1次量だけで足りなくなった場合，EXTEND DOMAINによってそのプログラムのすぐ後に2次量で指定された領域を確保する。このとき他のプログラム領域を侵害するようになったら，いったん自分のプログラムをロールアウトして，必要な大きさの空きが主記憶にあるかどうかを調べてロールインする。不要になった領域を解放するにはEREE文を用いる。このようにして解放した領域は他のユーザプログラムは使用できず，自分のプログラムで再利用できるだけである。

次は実際にALLOCATEとFREEの行なわれる様子を図に示す。以下のような場合には




というように，ALLOCATE文の実行順に又1つのALLOCATE文の中では始めに現われたアレイから順に実領域が確保される。注意しなければならないのは，

This document is provided by JAXA.

FREE文によって領域の再利用を図る場合のFREEの順序である。例えば

……………………

ALLOCATE　A，B

……………………

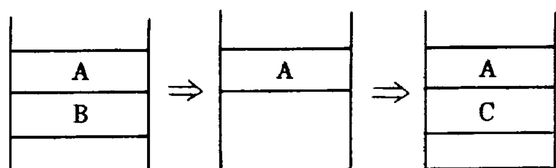
FREE　B

……………………

ALLOCATE　C

……………………

は，

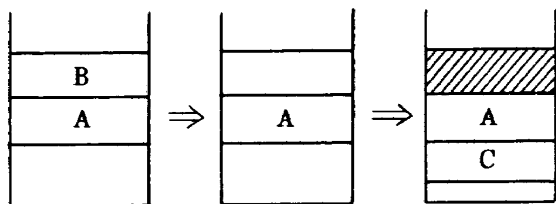



ALLOCATE　A，B　FREE　B　ALLOCATE　C

となるが，最初のALLOCATE文が

ALLOCATE　B，A

となっていると




ALLOCATE　B，A　FREE　B　ALLOCATE　C

となって，Bが占めていた記憶域（斜線部）は再利用不可能となる。いったんこのようになってしまったら，

FREE　A，C

としても斜線部の再利用不可の領域は利用可能とならない。なおFREE文に複数の配列名を書いたときにはシステムの方で自動的に番地の高い方から順に解放してくれる。例えば

ALLOCATE　A，B

とした後，

FREE　A，B

としても

FREE　B，A

としても効果は同じである。しかし，連続した文であっても

FREE　A

FREE　B

としたら，Aが占めていた領域は再利用不可能となる。従ってこの場合には警告エラーメッセイジを出力して処理を続けるが，この様な状態は甚だ好ましくない。故にFREEする場合にはALLOCATEした順序に注意して，番地の高い方から解放することは絶対に必要である。またALLOCATEされた配列にはEQUIVALENCEで結合することはできないし，COMMON 宣言も許されないので注意すること。ALLOCATEにより領域が拡張される場合，前述の様に1次量で足りなくなった場合ロールアウト，ロールインが生じるので，ALLOCATE，FREEは効果的に使用しなければならないことも重要である。

### 5.2　高速入出力

APは前節で述べた様に入出力処理ルーチンを備えていない，そのために処理をCPへ依頼することになる。それではCPで一般に行なわれている入出力処理がどの様に行なわれているか簡単に述べると，データをブロック長に分割しバッファを使用して入出力を行なっている。データ量が大きくなるとファイルへの入出力回数が多くなり時間がかかるし，ブロック長を大きくするとバッファも大きくなり，バッファそのものが主記憶領域内にあるため限られている。従ってAPの様に大量のデータを入出力する場合，一般の入出力処理方法ではAPの機能を損うことになる。高速入出力はこの様なことを少しでも解消させようとするものである。つまり大量のデータをより速く入出力処理し，APを効率的に使用しようとするものである。この高速入出力には2つの機能がありまずそれについて説明する。

1）バッファレス機能

前述した様に一般の入出力はバッファを使用している。そして大量のデータを効率的に処理しようとすると大きなバッファが必要となるがそれを大きくすることは困難である。そこでこの機能は入出力文に表われた入出力並びのデータ領域を仮のバッファに見たて，直接入出力する方法である。従ってこの場合の入出力は書式なし入出力文で入出力並びの配列名全体が対象となる。この様にすることによりバッファの節約とバッファの転送時間を無くそうとしている。

2）パラレルファイル機能

一般の入出力は1つの入出力論理機番に対して，処理対象となるファイルは1つである。従ってAPの様に大量のデータの入出力を必要とする場合は問題となる。例えば大量のデータを一度に1つのファイルに入出力する場合又は一度に1つのファイルに入出力できない場合は複数回の入出力処理を行うことになり，そのために費す

This document is provided by JAXA.

時間も大きい。この様な入出力処理はＡＰには不向である。だがパラレルファイル機能は複数個のファイル（図5.2.1参照；この図では２個のファイルで例示してある）に，データを複数ブロックに分割して同時に入出力しようとするものである。この様に処理方式を変えることにより入出力時間の減少を行なっている。

以上高速入出力の２つの機能について述べたが，この高速入出力を使用するには，AP－FORTRANの翻訳時のパラメータに"HFILE"と指定すれば良い。またパラレルファイルで使用するためにはジョブ制御文で使

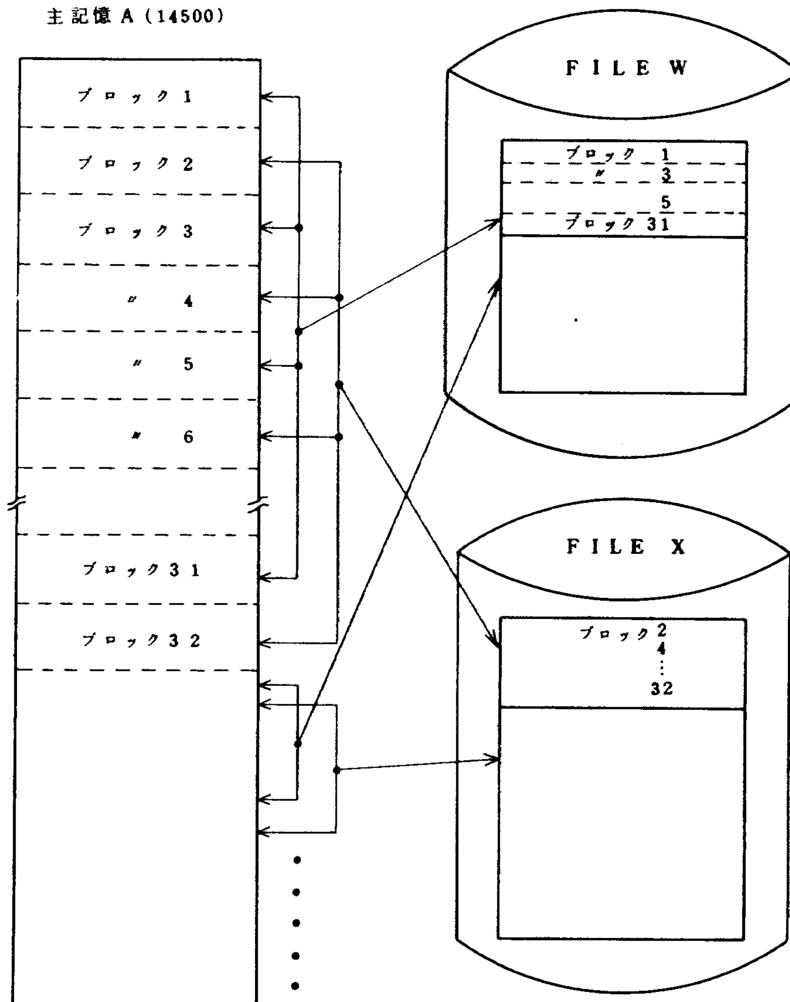



図 5. 2. 1

```
DIMENSION A(14500), B(14500)
      •
READ (1) A
B(*)=SIN(C(*))
      •
      •
S=VSUM(B(*))
      •
A(*)=A(*)+B(*)*C
      •
```

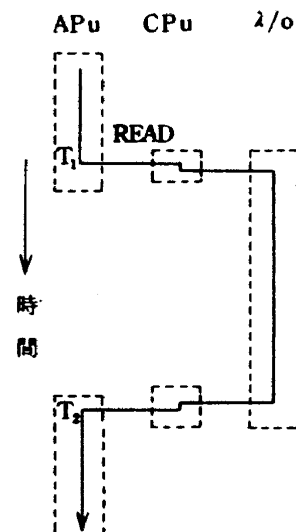



図5.2.2　非同期でない入出力

This document is provided by JAXA.

```
DIMENSION A(14500), B(14500)
    ・
    ・
READ(1, ID=n) A
B(*)=SIN(C(*))
    ・
    ・
S=VSUM(B(*))
    ・
WAIT(1, ID=n) A
A(*)=A(*)+B(*)+C
```

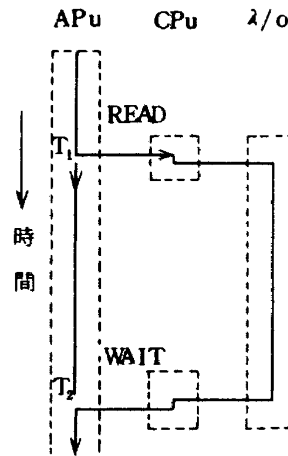



図5.2.3　非同期の入出力

用するファイルを定義しなければならない。注1)

次にこの高速入出力を利用してAPを効率良く使用する方法がある。それは非同期入出力による処理方法である。この処理方法を説明すると，まず非同期でない場合の図5.2.2と非同期の場合の図5.2.3を比べる。非同期でない場合は入出力処理が完了するまでは，次のステートメントを実行しない。従って入出力に要する時間中APは遊んでいることになる。しかし，非同期の場合は入出力タスクを発生させ，次々のステートメントを実行し，WAIT文のところまで行ったら同期をとる様にしている。この様に処理することにより入出力に要する時間中もAPはプログラムを中断することなく実行するわけである。ただ注意しなければならないのは入出力文からWAIT文までのステートメント中に入出力された配列名を使用してはいけない。もし使用した場合の結果は保証されない。

3）高速入出力の実測

高速入出力を使用した実際の測定値を表5.2.1に示し

表5.2.1　高速入出力

単位：秒

| ファイル構成 | | Hファイル | 一般ファイル |
|---|---|---|---|
| パラレルファイル | DR×2+DP | 2.281 | * |
| | DR×2 | 2.322 | * |
| | DR+DP | 2.425 | * |
| 単一ファイル | DR | 2.789 | 2.753 |
| | DP | 4.679 | 4.622 |

WORD：260400語，DR：BLKSIZE=23430バイト，DP：BLKSIZE=1300バイト

注1）使用方法については文献3）を参照のこと。

てある。表欄にHファイルと記してあるのが高速入出力を使用した場合，一般ファイルと記してあるのは通常行なわれている入出力方式である。ファイル構成の欄でDRは磁気ドラム，DPは磁気ディスクのことであり，DR×2+DP　は磁気ドラム2本と磁気ディスクとをパラレルファイルで使用した場合を示している。DR×2，DR+DP　も同様である。そしてこのパラレルファイルに関して，一般ファイルの実測値の欄は“＊”になっているが，これは前述した様に一般の入出力は処理対象となるファイルは1つしかなく，パラレルファイルの意味はなさない。従って実測してみてもDR，DPの単ファイルの場合と等しいことは明らかであるので省略した。このことは後述する非同期入出力での表5.2.2についても同様である。また　DP×2　の構成を示していないのもDPのチャネルが1つしかなく，（ハードウェア上の問題で）パラレルファイルとしての意味はなさないからである。DRはチャネルを2つ有している。故に表5.2.1の様なファイル構成で実測した。この実測結果自明なことであるがDPよりDRの方が速い。また単ファイルの場合，Hファイルと一般ファイルとではほとんど変わらない。むしろ一般ファイルへの入出力の方が速い結果が出ている。これはDRやDPのアクセスタイムやポジショニングタイムの影響があるからであり，それは数十msのオーダである。しかしパラレルファイルにすると確かに速くなっている。

次に非同期入出力に関する表5.2.2を参照してもらうと，ここでもやはり同様のことが言える。そして，非同期入出力の場合と非同期入出力でない場合の差が顕著に表われている。

この表5.2.2の測定の際にWRITE文からWAIT文までの間に約1.3秒くらいの（入出力には関係の無い）

This document is provided by JAXA.

表5.2.2 非同期入出力

単位：秒

| ファイル構成 | 非同期の場合 | | 非同期でない場合 | |
|---|---|---|---|---|
| | H ファイル | 一般 ファイル | H ファイル | 一般 ファイル |
| DR×2+DP | 2.446 | * | 3.591 | * |
| DR×2 | 2.425 | * | 3.694 | * |
| DR+DP | 2.532 | * | 3.734 | * |
| DR | 2.893 | 2.960 | 4.097 | 4.062 |
| DP | 4.722 | 4.641 | 6.007 | 5.906 |

WORD；260400語，DR；BLKSIZE=23430バイト，DP；BLKSIZE=1300バイト

プログラムを挿入して計測した。従って，この表の非同期でない入出力でのHファイルと一般ファイルの実測値は入出力処理時間＋1.3秒ということになる。くどい様であるが非同期入出力は入出力処理の間に，WAIT文の所まで処理しているからその値はほとんど入出力処理に要した時間だけが値として表われている。またパラレルファイルとしての機能も十分表われている。これでほぼ高速入出力についての説明が終ったが，最後にもう1つ重要なことがある。それはファイルを使用する際ブロック長を指定せねばならない。データはブロック長に合わせて分割されるが，そのブロック長が大きすぎたり，小さすぎてもファイルへの入出力が頻繁に行なわれ時間が余計かかることになる。

そこで表5.2.3はDRへの入出力に関して，ブロック長を変えた際の実測値を表示してある。

表 5.2.3

| | | ブロック長（バイト） | 処理時間（秒） |
|---|---|---|---|
| 一般入出力 | BLK<BAND | 1378 | 20.911 |
| | BLK=BAND | 23436 | 2.436 |
| | BLK>BAND | 31248 | 3.659 |
| 高速入出力 | BLK<BAND | 1378 | 4.902 |
| | BLK=BAND | 23436 | 2.454 |
| | BLK>BAND | 31248 | 4.031 |

BLK：ブロック長　BAND：バンド長

DRは最大容量が6MB（メガバイト）であり，平均アクセスタイムは10ms である。そして1トラック5859バイトであり，実際の入出力は1バンド＝4トラックで処理している。従ってDRの場合のブロック長は23436バイトにすると，最も好ましい入出力のブロック長になる。DPの場合は1トラック＝13030バイトであるからブロック長をその様にとると良い。表5.2.3に於てもブロック長とバンド長が等しい時が最も速くなっている。

以上高速入出力について述べたが，この高速入出力を使用する最大の利点はAPをより効率的に使用することである。それには今まで述べたことはもちろんのことであるがプログラムの処理手順を考慮しながら，ハードウェアというものを意識して使用することが望まれる。

# 6 APープログラムのコーディング例

## 6.1 1つのプログラム単位としてのコーディング例

### 6.1.1 数値積分計算

関数 $f(x)$ の数値積分公式のなかに，ガウス型積分公式と呼ばれているものがある。その一般的な形は，重み関数 $w(x)$ を導入して，関数 $f(x)$ そのものではなく，$f(x)w(x)$ の積分値を求めるものである。ここで，$w$ は適当な族の $f$ に対して $fw$ が区間 $[a,b]$ で可積分となるものである。積分 $\int_a^b f(x)w(x)dx$ を

$$\int_a^b f(x)w(x)dx \fallingdotseq \sum_{i=1}^{n} a_i f(x_i)$$

という形で求める。この公式が，高々 $2n-1$ 次である多項式に対して正確であるためには，

『$x_i$ が，$w(x)$ を重み関数とする区間 $[a,b]$ の直交多項式の根であって，$a_i$ が

$$a_i=\int_a^b \frac{w(x)w(x)}{(x-x_i)w'(x_i)}dx$$

$$w(x)=\prod_{i=1}^{n}(x-x_i)$$

である』

ことである。この公式による理論的な誤差は

$$f^{(2n)}(\xi)/(2n)!\int_a^b w^2(x)w(x)dx$$

$$a\leq\xi\leq b$$

で与えられる。

この方法の特徴は，関数値を計算すべき分点の数とその値があらかじめ与えられているので，計算が高速であること，しかも滑らかな関数に対しては少ない分点の数で十分よい近似値が得られることである。一方，分点の値があらかじめ決っているということは，関数入力の積分しかできないことであり，又分点の数を自由に変えることが困難である。

この公式で，区間を $[-1,1]$，重み関数を1とすると直交多項式としてルジャンドルの多項式が得られ，こ

This document is provided by JAXA.

の場合が普通にガウスの積分公式と呼ばれているものである。区間が〔0，∞〕で重み関数が $e^{-x}$ のときにはラゲールの多項式が得られ，区間が〔$-\infty$，∞〕で重み関数が $e^{-x^2}$ のときにはエルミートの多項式が得られる。

多数の積分値を計算する必要のあるとき，試計算等によりあらかじめ分点数を決めることができれば，簡単で有効な方法である。

各場合に必要なデータを次に掲げる。Nは分点数で4つの場合の数値に限定した。

I）$\int_{-1}^{1} f(x)\,dx \fallingdotseq \sum_{i=1}^{N} a_i f(x_i)$

ルジャンドルの多項式の零点は，正負が対になって現われ，その対に応じる $a_i$ の値は同じなので，実際のデータ数は半分で済む。

N＝32

| | | |
|---|---|---|
| $x_i$: | 0.99726386184948156 | 0.98561151154526834 |
| | 0.96476225558750643 | 0.93490607593773969 |
| | 0.89632115576605212 | 0.84936761373256997 |
| | 0.79448379596794241 | 0.73218211874028968 |
| | 0.66304426693021520 | 0.58771575724076233 |
| | 0.50689990893222939 | 0.42135127613063535 |
| | 0.33186860228212765 | 0.23928736225213707 |
| | 0.14447196158279649 | $0.48307665687738316_{-1}$ |
| $a_i$: | $0.70186100094700966_{-2}$ | $0.16274394730905674_{-1}$ |
| | $0.25392065309262062_{-1}$ | $0.34273862913021431_{-1}$ |
| | $0.42835898022226678_{-1}$ | $0.50998059262376177_{-1}$ |
| | $0.58684093478535549_{-1}$ | $0.65822222776361846_{-1}$ |
| | $0.72345794108848506_{-1}$ | $0.78193895787070307_{-1}$ |
| | $0.83311924226946755_{-1}$ | $0.87652093004403812_{-1}$ |
| | $0.91173878695763886_{-1}$ | $0.93844399080804566_{-1}$ |
| | $0.95638720079274860_{-1}$ | $0.96540088514727800_{-1}$ |

N＝24

| | | |
|---|---|---|
| $x_i$: | 0.99518721999702136 | 0.97472855597130950 |
| | 0.93827455200273276 | 0.88641552700440103 |
| | 0.82000198597390292 | 0.74012419157855436 |
| | 0.64809365193697557 | 0.54542147138883954 |
| | 0.43379350762604514 | 0.31504267969616337 |
| | 0.19111886747361631 | $0.64056892862605626_{-1}$ |
| $a_i$: | $0.12341229799987200_{-1}$ | $0.28531388628933663_{-1}$ |
| | $0.44277438817419806_{-1}$ | $0.59298584915436781_{-1}$ |
| | $0.73346481411080306_{-1}$ | $0.86190161531953276_{-1}$ |
| | $0.97618652104113888_{-1}$ | 0.10744427011596563 |
| | 0.11550566805372560 | 0.12167047292780339 |
| | 0.12583745634682830 | 0.12793819534675216 |

N＝16

| | | |
|---|---|---|
| $x_i$: | 0.98940093499164993 | 0.94457502307323258 |
| | 0.86563120238783174 | 0.75540440835500303 |
| | 0.61787624440264375 | 0.45801677765722739 |
| | 0.28160355077925891 | 0.95012509837637440 |
| $a_i$: | $0.27152459411754095_{-1}$ | $0.62253523938647893_{-1}$ |
| | $0.95158511682492785_{-1}$ | 0.12462897125553387 |
| | 0.14959598881657673 | 0.16915651939500254 |
| | 0.18260341504492359 | 0.18945061045506850 |

N＝12

| | | |
|---|---|---|
| $x_i$: | 0.98156063424671925 | 0.90411725637047486 |
| | 0.76990267419430469 | 0.58731795428661745 |
| | 0.36783149899818019 | 0.12523340851146892 |
| $a_i$: | $0.47175336386511827_{-1}$ | 0.10693932599531843 |
| | 0.16007832854334623 | 0.20316742672306592 |
| | 0.23349253653835481 | 0.24914704581340279 |

II）$\int_{0}^{\infty} e^{-x} f(x)\,dx \fallingdotseq \sum_{i=1}^{N} a_i f(x_i)$

N＝26

| | | |
|---|---|---|
| $x_i$: | $5.45644827179085360_{-2}$ | $2.87707979106123607_{-1}$ |
| | $7.08016019478644922_{-1}$ | 1.31708136604384414 |
| | 2.11712094462846926 | 3.11109387887793412 |
| | 4.30277087604037028 | 5.69681882616946593 |
| | 7.29890996307018320 | 9.11586289729421475 |
| | $1.11558249549375105_{+1}$ | $1.34285090324592041_{+1}$ |
| | $1.59455039481433189_{+1}$ | $1.87206861454602847_{+1}$ |
| | $2.17707746085414357_{+1}$ | $2.51160936623167377_{+1}$ |
| | $2.87816468734501689_{+1}$ | $3.27986732184099763_{+1}$ |
| | $3.72069826296128271_{+1}$ | $4.20586159244345338_{+1}$ |
| | $4.74238993562815293_{+1}$ | $5.34021851972532916_{+1}$ |
| | $6.01427756900768968_{+1}$ | $6.78914797054512926_{+1}$ |
| | $7.71179990693328597_{+1}$ | $8.90284027504109733_{+1}$ |
| $a_i$: | $1.32616884147601980_{-1}$ | $2.44867368016292435_{-1}$ |
| | $2.53392022127125576_{-1}$ | $1.88645983641759894_{-1}$ |
| | $1.07912341642588073_{-1}$ | $4.86565654489217972_{-2}$ |
| | $1.74821379784894276_{-2}$ | $5.02498383683131229_{-3}$ |
| | $1.15526916074960977_{-3}$ | $2.11791158300360477_{-4}$ |
| | $3.07925260417151313_{-5}$ | $3.52344009102437004_{-6}$ |
| | $3.14155632535924590_{-7}$ | $2.15545681440371042_{-8}$ |
| | $1.12047811767047841_{-9}$ | $4.32900240513688527_{-11}$ |
| | $1.21368915983720525_{-12}$ | $2.39607422326129944_{-14}$ |
| | $3.20521855983162644_{-16}$ | $2.76249415060793682_{-18}$ |

This document is provided by JAXA.

| | | |
|---|---|---|
| | 1.43306436407272879$_{-20}$ | 4.06484469411303358$_{-23}$ |
| | 5.45951332279888852$_{-26}$ | 2.74322098713532642$_{-29}$ |
| | 3.30023047815207488$_{-33}$ | 3.08375226958557708$_{-38}$ |

N＝２２

| | | |
|---|---|---|
| $x_i$: | 6.42676287448089085$_{-2}$ | 3.38967254814911020$_{-1}$ |
| | 8.34589985449178405$_{-1}$ | 1.55371338675002609 |
| | 2.50000623673617244 | 3.67842034446363703 |
| | 5.09534906896873815 | 6.75883551581354881 |
| | 8.67885331367842746 | 1.08676868935405716$_{+1}$ |
| | 1.33404510514952406$_{+1}$ | 1.61158113387663116$_{+1}$ |
| | 1.92170036849013543$_{+1}$ | 2.26733166014345901$_{+1}$ |
| | 2.65223195876371379$_{+1}$ | 3.08133579445065581$_{+1}$ |
| | 3.56133352031961700$_{+1}$ | 4.10169685984716853$_{+1}$ |
| | 4.71667578491535205$_{+1}$ | 5.42973852565788842$_{+1}$ |
| | 6.28570962462990292$_{+1}$ | 7.39955070085994982$_{+1}$ |
| $a_i$: | 1.54701987639893375$_{-1}$ | 2.74175082328564683$_{-1}$ |
| | 2.63409701535633718$_{-1}$ | 1.75934609263072543$_{-1}$ |
| | 8.71256221722098802$_{-2}$ | 3.27526878965386203$_{-2}$ |
| | 9.42490025699826495$_{-3}$ | 2.07733575265461581$_{-3}$ |
| | 3.49167371492300859$_{-4}$ | 4.43821931406614274$_{-5}$ |
| | 4.21422990110462038$_{-6}$ | 2.94066824613015617$_{-7}$ |
| | 1.47619382638974377$_{-8}$ | 5.18642302572486127$_{-10}$ |
| | 1.23052625155441281$_{-11}$ | 1.88046550963147326$_{-13}$ |
| | 1.73556413912683628$_{-15}$ | 8.83140230832711454$_{-18}$ |
| | 2.16071864681819978$_{-20}$ | 2.03111996758332949$_{-23}$ |
| | 4.79790352099079971$_{-27}$ | 9.78024650668382284$_{-32}$ |

N＝１６

| | | |
|---|---|---|
| $x_i$: | 8.76494104789278404$_{-2}$ | 4.62696328915080832$_{-1}$ |
| | 1.14105777483122686 | 2.12928364509838062 |
| | 3.43708663389320665 | 5.07801861454976791 |
| | 7.07033853504823413 | 9.43831433639193878 |
| | 1.22142233688661587$_{+1}$ | 1.54415273687816171$_{+1}$ |
| | 1.91801568567531349$_{+1}$ | 2.35159056939919085$_{+1}$ |
| | 2.85787297428821404$_{+1}$ | 3.45833987022866258$_{+1}$ |
| | 4.19404526476883326$_{+1}$ | 5.17011603395433184$_{+1}$ |
| $a_i$: | 2.06151714957800994$_{-1}$ | 3.31057854950884166$_{-1}$ |
| | 2.65795777644214153$_{-1}$ | 1.36296934296377540$_{-1}$ |
| | 4.73289286941252190$_{-2}$ | 1.12999000803394532$_{-2}$ |
| | 1.84907094352631086$_{-3}$ | 2.04271915308278460$_{-4}$ |
| | 1.48445868739812988$_{-5}$ | 6.82831933087119956$_{-7}$ |
| | 1.88102484107967321$_{-8}$ | 2.86235024297388162$_{-10}$ |
| | 2.12707903322410297$_{-12}$ | 6.29796700251786779$_{-15}$ |
| | 5.05047370003551282$_{-18}$ | 4.16146237037285519$_{-22}$ |

N＝１０

| | | |
|---|---|---|
| $x_i$: | 1.37793470540492431$_{-1}$ | 7.29454549503170498$_{-1}$ |
| | 1.80834290174031605 | 3.40143369785489951 |
| | 5.55249614006380363 | 8.33015274676449670 |
| | 1.18437858379000656$_{+1}$ | 1.62792578313781021$_{+1}$ |
| | 2.19965858119807620$_{+1}$ | 2.99206970122738916$_{+1}$ |
| $a_i$: | 3.08441115765020142$_{-1}$ | 4.01119929155273552$_{-1}$ |
| | 2.18068287611809422$_{-1}$ | 6.20874560986777474$_{-2}$ |
| | 9.50151697518110055$_{-3}$ | 7.53008388587538775$_{-4}$ |
| | 2.82592334959956557$_{-5}$ | 4.24931398496268637$_{-7}$ |
| | 1.83956482397963078$_{-9}$ | 9.91182721960900856$_{-13}$ |

III) $\int_{-\infty}^{\infty} e^{-x^2} f(x)\,dx \fallingdotseq \sum_{i=1}^{N} a_i f(x_i)$

エルミートの多項式の零点は，正負が対になって現われ，その対に応じる $a_i$ の値は同じなので，実際のデータ数は半分で済む。

N＝３０

| | | |
|---|---|---|
| $x_i$: | 6.86334529352989158 | 6.13827922012393462 |
| | 5.53314715156749573 | 4.98891896858994394 |
| | 4.48305535709251834 | 4.00390860386122882 |
| | 3.54444387315534989 | 3.09997052958644175 |
| | 2.66713212453561720 | 2.24339146776150407 |
| | 1.82674114360368804 | 1.41552780019818851 |
| | 1.00833827104672346 | 6.03921058625552308$_{-1}$ |
| | 2.01128576548871486$_{-1}$ | |
| $a_i$: | 2.90825470013122623$_{-21}$ | 2.81033360275090371$_{-17}$ |
| | 2.87860708054870606$_{-14}$ | 8.10618629746304420$_{-12}$ |
| | 9.17858042437852821$_{-10}$ | 5.10852245077594628$_{-8}$ |
| | 1.57909488732471029$_{-6}$ | 2.93872522892298764$_{-5}$ |
| | 3.48310124318685523$_{-4}$ | 2.73792247306765846$_{-3}$ |
| | 1.47038297048266835$_{-2}$ | 5.51441768702342512$_{-2}$ |
| | 1.46735847540890100$_{-1}$ | 2.80130930839212667 |
| | 3.86394889541813863$_{-1}$ | |

N＝２４

| | | |
|---|---|---|
| $x_i$: | 6.01592556142573972 | 5.25938292766804437 |
| | 4.62566275642378727 | 4.05366440244814950 |
| | 3.52000681303452471 | 3.01254613756556483 |
| | 2.52388101701142697 | 2.04900357366169891 |
| | 1.58425001096169415 | 1.12676081761124507 |
| | 6.74171107037212236$_{-1}$ | 2.24414547472515585$_{-1}$ |
| $a_i$: | 1.66436849648910887$_{-16}$ | 6.58462024307817006$_{-13}$ |
| | 3.04625426998756390$_{-10}$ | 4.01897117494142968$_{-8}$ |
| | 2.15824570490233363$_{-6}$ | 5.68869163640437977$_{-5}$ |

This document is provided by JAXA.

8.23692482688417458$_{-4}$ 7.04835581007267097$_{-3}$
3.74454705032307460$_{-2}$ 1.27739621784559161$_{-1}$
2.86179535346443018$_{-1}$ 4.26931163868699250$_{-1}$

N＝16

$x_i$: 4.68873893930581836 3.86944790486012270
3.17699916197995603 2.54620215784748136
1.95178799091625398 1.38025853919888080
8.22951449144655893$_{-1}$ 2.73481046138152452$_{-1}$

$a_i$: 2.65480747401118224$_{-10}$ 2.32098084486521065$_{-7}$
2.71186009253788151$_{-5}$ 9.32284008624180530$_{-4}$
1.28803115355099737$_{-2}$ 8.38100413989858294$_{-2}$
2.80647458528533675$_{-1}$ 5.07929479016613742$_{-1}$

N＝12

$x_i$: 3.88972489786978192 3.02063702512088977
2.27950708050105990 1.59768263515260480
9.47788391240163744$_{-1}$ 3.14240376254359111$_{-1}$

$a_i$: 2.65855168435630161$_{-7}$ 8.57368704358785865$_{-5}$
3.90539058462906186$_{-3}$ 5.16079856158839300$_{-2}$
2.60492310264161129$_{-1}$ 5.70135236262479578$_{-1}$

分点は等間隔であるが，分点数を自由に変えられる積分公式にニュートンーコーツの公式がある。この一般の形は

$$\int_a^b f(x)dx \fallingdotseq h\sum_{i=0}^{n} B_i^n f(x_i)$$

$$h=(b-a)/n,\ x_i=a+ih \quad 0\leq i\leq n$$

$$B_i^n=B_{n-i}^n=(-1)^{n-i}/(ni!(n-i)!)$$

$$\times\int_0^n\left(\prod_{j=0}^{n}(t-j)\right)/(t-i)\,dt$$

で与えられる。この式で，$n=1$ としたときが台形公式，$n=2$ としたときがシンプソンの公式と呼ばれているものである。長い区間にわたる積分を行なうときには，高次一大きい$n$一の公式を使うよりも，積分区間を分割してそれに低次の公式を繰り返し適用する方が優れている。このようにすると，通常の台形則，シンプソン則

$$\int_a^b f(x)dx \fallingdotseq \frac{h}{2}(f_0+2f_1+2f_2+\cdots+2f_{n-1}+f_n)$$

$$\int_a^b f(x)dx \fallingdotseq \frac{h}{3}(f_0+4f_1+2f_2+4f_3+\cdots+4f_{n-1}+f_n)$$

$$h=(b-a)/n,\ f_i=f(a+ih) \quad 0\leq i\leq n$$

が得られる。ここでは，区間〔$a,b$〕が$n$等分されている。なおシンプソン則では$n$は偶数である。

これらの公式による理論的誤差は，台形公式では

$$(b-a)^3/12n^2\ f''(\xi) \qquad a\leq\xi\leq b$$

シンプソン則では

$$(b-a)^5/180n^4\ f^{(4)}(\xi) \quad a\leq\xi\leq b$$

である。なお端点で$f'$又は$f''$が存在しないときには，一般には誤査は上式より大きくなるので計算に工夫をこらす必要がある。

この方法をプログラム化するとき，分点の数を固定すればプログラムそのものは非常に簡単である。以下ではシンプソン則だけを考える。或は，分点数を倍々にしふやして区間を細分化してゆき，そのつど積分値を計算して，適当な収束判定によって計算を止めるという手法も比較的容易である。このとき，分点数を倍々にすると，新しく計算する必要のある関数値は新しい分点数の半分で済むという利点がある。それは，$S=\int_a^b f(x)dx$，$n$を偶数として区間を$2n$等分した場合の式

$$\frac{3}{b-a}S=(f_0+f_{2n})/2n+2/2n\sum_{i=1}^{n-1}f_{2i}$$

$$+4/2n\sum_{i=1}^{n}f_{2i-1}$$

$$=(f_0+f_{2n})/2n$$

$$+1/n\left(\sum_{i=1}^{n/2-1}f_{4i}+\sum_{i=1}^{n/2}f_{4i-2}\right)$$

$$+4/2n\sum_{i=1}^{n}f_{2i-1}$$

において，$f_{4i}$，$f_{4i-2}$が1つ前の段階で計算されているからである。これより

$$P_n=(f_0+f_n)/n+2/n\sum_{i=1}^{n/2-1}f_{2i},$$

$$R_n=1/n\sum_{i=1}^{n/2}f_{2i-1}$$

$$S=\frac{b-a}{3}(P_n+4R_n),\quad P_{2n}=1/2\,P_n+R_n$$

という計算式が得られる。APープログラムということを考慮し，プログラムの簡単さを求めるなら最初の$n$は小さくとも4，6，8程度が好ましい。

上述したのは全区間を一度に細分するものであるが，そうではなく或る1つの部分区間だけを細分していき，その部分区間で積分が収束したら次の部分区間に進むという手法もある。これの目的とするところは，関数が充分滑らかで細分する必要のない部分区間までをも細分するのは止める，即ち関数値計算の回数を減らしかつ同程度の精度をだそうとすることにある。

ここに掲げるコーディング例は，積分計算だけの単体プログラムであるから，実際の使用時にはそのプログラ

This document is provided by JAXA.

ム内での特殊性を考慮して，適当に修正することが好ましい。なお，函数値の計算に当っては，被積分関数がよほど複雑でしかもプログラム内の随所で使用されるというものでない限り，インライン化してサブプログラム化しないようにする。

下記コーディング例はいずれも $\int_a^b 1/(1+x^2)dx$ を計算するものである。P，Qには定積分の下限値 $a$，上限値 $b$ をそれぞれ代入する。Sには積分値が格納される。

1）有限区間のガウス公式（32分点）

```
DIMENSION A(16),B(16),C(32),F(32)
INDEX IX/1,16/,JX/17,32/
```

| A，Bにデータ文により32分点の場合の$a_i$ と$x_i$の値を与える。 |
|---|

```
    Y=0.5*(Q+P)
    Z=0.5*(Q-P)
    C(IX)=Z*B(*)+Y
    C(JX)=Z*(-B(*))+Y
    F(*)=1.0/(1.0+C(*)*C(*))
    S=Z*AIPD(A(*), F(IX)+F(JX))
```

2）分点を倍々にするシンプソン法

最初の $n$ として4を取り，最大5/2等分しか行なわない場合のコーディング例。この方法のコーディングで種々の変形が考えられる。

```
    DIMENSION A(513),F(256),
              FL(512), LA(512)
    LOGICAL LA
    INDEX IX/1,513,128/, JX/1,5/,
          KX/2,512/, LX/NA,512,
          NB/, NX/1, NC/
    LA(*)=.TRUE.
    FL(*)=IDXL(LA(*))
    A(1)=P
    A(513)=Q
    R=(Q-P)/FL(512)
    T=(Q-P)/3.0
    A(KX)=P+R*FL(KX-1)
    F(JX)=1.0/(1.0+A(IX)*A(IX)
    X=0.25*(F(1)+F(5))+0.5*F(3)
    Y=0.25*(F(2)+F(4))
    R=T*(X+4.0*Y) ：4等分した場合の積
    NA=65
    NB=128
    NC=4
    DO 10 I=1,7
    X=0.5*X+Y
    F(NX)=1.0/(1.0+A(LX)*A(LX))
    Y=VSUM(F(NX))/FLT(2*NC)
    S=T*(X+4.0*Y)
```

| RとS等を用いて適当な収束判定をする，収束ならば GO TO 20 |
|---|

```
    R=S
    NA=(NA+1)/2
    NB=NB/2
    NC=2*NC
10  CONTINUE
```

| 収束しなかった場合の処理 |
|---|

```
20  CONTINUE
```

| 収束後の計算 |
|---|

（注意）今の場合アルゴリズム上インデックスの跳びが $2^n$ の形になったが，主記憶のバンク構成を考えれば，単精度で64以上，倍精度で32以上の $2^n$ の形の跳びが現われないように配慮しなければならない。

### 6.1.2 逐次代入処理計算

$m$ 次逐次代入処理といわれるものの一般形は

$$x_i=a_i x_{i-1}+b_i x_{i-2}+\cdots\cdots+f_i x_{i-m}$$

又は

$$x_i=a_i x_{i-1}+b_i x_{i-2}+\cdots\cdots+f_i x_{i-m}+g_i$$

で，前者が同次形，後者が非同次形である。これはさらに2つに分類されて，特殊関数の計算の場合等のように最終の $x_n$ の値だけが必要とされる場合と，途中の $x_i$ すべての履歴を必要とする場合とである。また，実用的には同次形では $m=2$，非同次形では $m=1$ のことが多い。後者についてはNP型の例として以前に詳しく説明した。そこでは，単一の逐次代入処理を考えるのではなく，それがもう1つのループによって複数個になっている場合には自然な方法でベクトル化でき，数個で十分スピードアップが図られることを示した。ここでは単一の逐次代入処理を異った角度から考えてみる。

$m$ 次の同次形逐次代入処理は行列とベクトルとによって

This document is provided by JAXA.

$$\begin{pmatrix} x_i \\ x_{i-1} \\ \vdots \\ x_{i-m+1} \end{pmatrix} = \begin{pmatrix} a_i & b_i & \cdots & f_i \\ 1 & 0 & & \\ 0 & 1 & \ddots & 0 \\ \vdots & 0 & \ddots & \\ 0 & \cdots 0 & 1 & 0 \end{pmatrix} \begin{pmatrix} x_{i-1} \\ x_{i-2} \\ \vdots \\ x_{i-m} \end{pmatrix}$$

$$= M_i \begin{pmatrix} x_{i-1} \\ x_{i-2} \\ \vdots \\ x_{i-m} \end{pmatrix}$$

$$= M_i M_{i-1} \cdots M_1 \begin{pmatrix} x_0 \\ x_{-1} \\ \vdots \\ x_{1-m} \end{pmatrix}$$

と表わされ，又$m$次非同次逐次代入処理は

$$\begin{pmatrix} x_i \\ x_{i-1} \\ \vdots \\ x_{i-m+1} \\ 1 \end{pmatrix} = \begin{pmatrix} a_i & b_i & \cdots & f_i & g_i \\ 1 & 0 & & & \\ 0 & 1 & \ddots & 0 & \\ \vdots & & & 1 & 0 \\ 0 & & & 0 & 1 \end{pmatrix} \begin{pmatrix} x_{i-1} \\ x_{i-2} \\ \vdots \\ x_{i-m} \\ 1 \end{pmatrix}$$

$$= N_i \begin{pmatrix} x_{i-1} \\ x_{i-2} \\ \vdots \\ x_{i-m} \\ 1 \end{pmatrix}$$

$$= N_i N_{i-1} \cdots N_1 \begin{pmatrix} x_0 \\ x_{-1} \\ \vdots \\ x_{1-m} \\ 1 \end{pmatrix}$$

と表わされる。ここに現われる行列積の計算は，いわゆるトーナメント方式によって遂行できるが，その名の由来のとおり $n=2^k$ の形のときに最も単純になる。その方法を$n=8$のときに図示すると図6.1.1のようになる。最終値だけを求める場合には前進過程だけを行なえばよく，途中の履歴も必要な場合には後退過程までを行なえばよい。

もう少し具体的に話をするために1次非同次形の逐次代入処理

$$x_i = a_i x_{i-1} + b_i \qquad 2 \leq i \leq n$$
$$x_1 = b_1$$

について考えてみる。この一般項は

$$x_n = \sum_{i=1}^{n} \left( \prod_{j=i+1}^{n} a_j \right) b_i$$

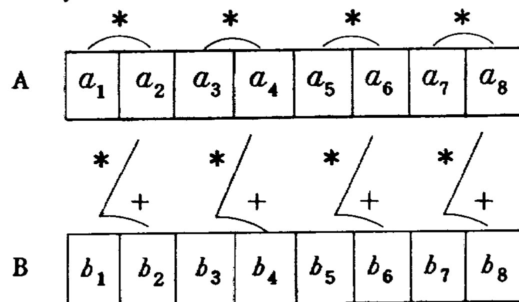
と書ける。又，行列表示は

$$N_i = \begin{pmatrix} a_i & b_i \\ 0 & 1 \end{pmatrix} \quad i \geqq 2 \quad N_1 = \begin{pmatrix} 1 & b_1 \\ 0 & 1 \end{pmatrix}$$

として

$$\begin{pmatrix} x_i \\ 1 \end{pmatrix} = N_i \cdots\cdots N_1 \begin{pmatrix} 0 \\ 1 \end{pmatrix}$$

と書ける。$n=8$ の場合の前進過程を図示すると図6.1.2となる。ここで$a_1=1$とする。プログラムするためにもう少しわかりやすく書くと次のようになる。まず$a_i$と$b_i$とをアレイA，Bにストアする。




後は同じパターンを繰り返すことになる。これより前進過程のみを行なう場合（$x_n$ の値だけを求める）わかり易いプログラムの一例は以下のようである。プログラムを簡単にするために今後すべて $n=2^k$ とする。

```
      DIMENSION A(N),B(N)
      INDEX IX/1,NA,2/,IY/1,NB/
      NA=N
      NB=N/2
      DO 10 I=1,K
      B(IY)=A(IX+1)*B(IX)+B(IX+1)
      A(IY)=A(IX)*A(IX+1)
      NA=NB
      NB=NB/2
   10 CONTINUE
```

このプログラムがどう進むかを図示すれば（$n=8$）の場合は図6.1.3となって結局(1)に$x_n$の値がストアされることになる。このプログラム方法では後退過程を行なわせることはできないのでそのためには次のようにするとよい。

```
      DIMENSION A(N),B(N)
      INDEX IX/IS,IE,IS/
      IS=2
      IE=N
      ID=1
      DO 10 I=1,K
      B(IX)=A(IX)*B(IX-ID)+B(IX)
```

This document is provided by JAXA.

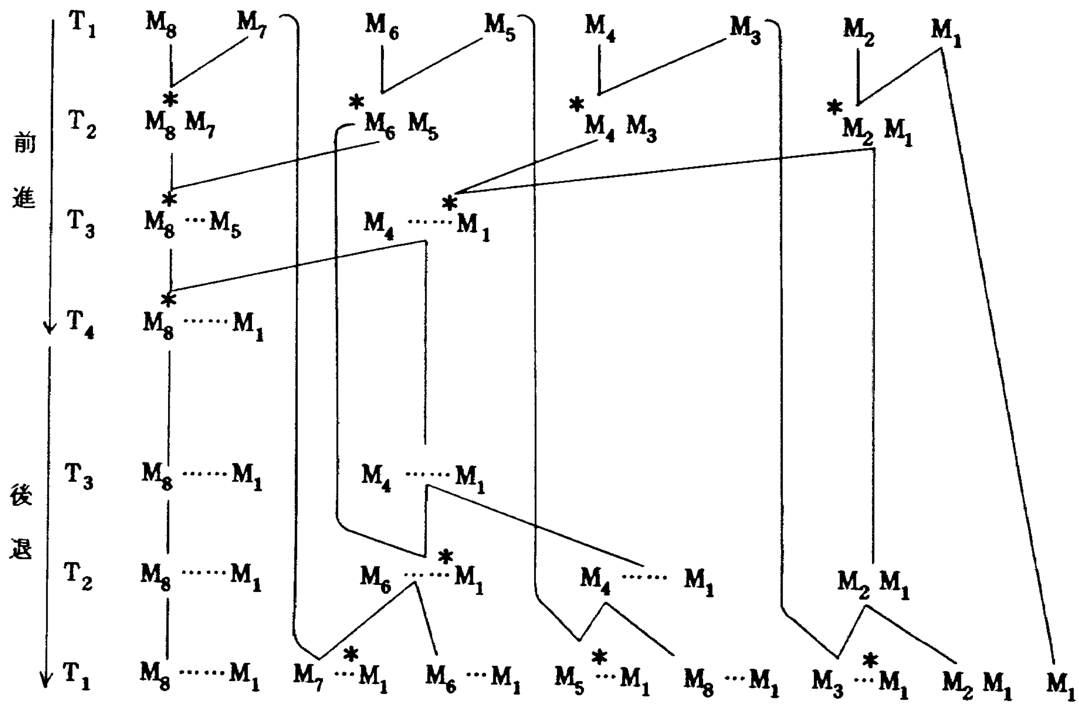



図 6.1.1

$$\begin{pmatrix} a_8 & b_8 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} a_7 & b_7 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} a_6 & b_6 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} a_5 & b_5 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} a_4 & b_4 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} a_3 & b_3 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} a_2 & b_2 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} a_1 & b_1 \\ 0 & 1 \end{pmatrix}$$

$$\begin{pmatrix} a_8a_7 & a_8b_7+b_8 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} a_6a_5 & a_6b_5+b_6 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} a_4a_3 & a_4b_3+b_4 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} a_2a_1 & \overbrace{a_2b_1+b_2}^{x_2} \\ 0 & 1 \end{pmatrix}$$

$$\begin{pmatrix} \prod_{i=5}^{8} a_i & \sum_{i=5}^{8} ( \prod_{j=i+1}^{8} a_j ) b_i \\ 0 & 1 \end{pmatrix} \qquad \begin{pmatrix} \prod_{i=1}^{4} a_i & \overbrace{\sum_{i=1}^{4} ( \prod_{0=i+1}^{4} a_j ) b_i}^{x_4} \\ 0 & 1 \end{pmatrix}$$

$$\begin{pmatrix} \prod_{i=1}^{8} a_i & \overbrace{\sum_{i=1}^{8} ( \prod_{j=i+1}^{8} a_j ) b_i}^{x_8} \\ 0 & 1 \end{pmatrix}$$

図 6.1.2

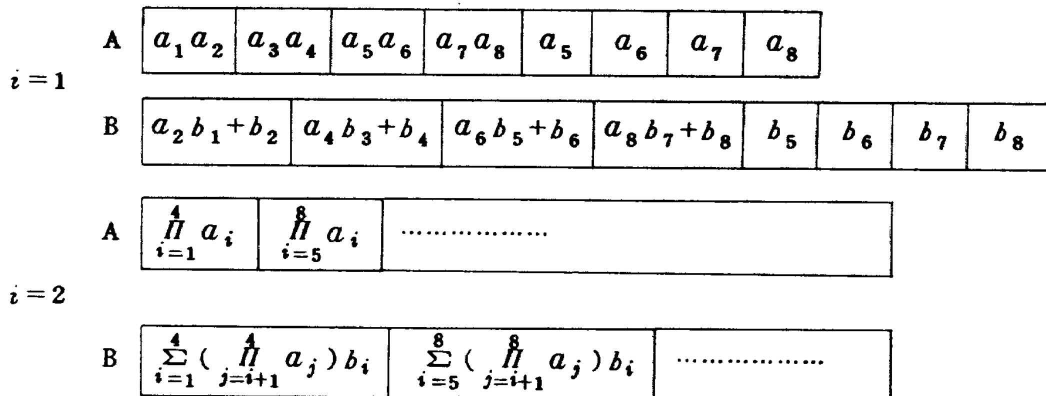



図 6.1.3

This document is provided by JAXA.

```
      A(IX)=A(IX)*A(IX-ID)
      ID=IS
      IS=2*IS
   10 CONTINUE
      ID=ID/4
      IS=2*ID
      IE=IE-1
      DO 20 I=1,K-1
      B(IX+ID)=A(IX+ID)*B(IX)
                     +B(IX+ID)
      A(IX+ID)=A(IX+ID)*A(ID)
      IS=ID
      ID=ID/2
   20 CONTINUE
```

このプログラムの進み方も容易に理解されるが，2度目のループが終った後のアレイの状態は，Aの第$i$要素が $\prod_{j=1}^{i} a_j$ でBの第$i$要素が$x_i$ となっている。又このプログラムではインデックスの跳びが$2^m$ の形になって好ましくないがアルゴリズム上不可避である。

2次の同次形逐次代入処理についても同様に考えればよい。ここでこのような問題をとり上げたのは，一見してベクトル化できないとみえるものでも，ミクロ的に考えてみてもそれが可能となる場合もあることを了解して頂きたいからである。但し，全体の効率を考えた場合には，データ構造も考慮した上でマクロ的にベクトル化した方が好ましい。前記プログラムの場合 $n=128$ でもCPのDO文による方が所要時間は少ない。又，逐次代入処理そのものに内在しているが，ここに述べた手法で現われる危険性としてオーバーフロー，アンダーフロー，桁落ちがあり，これらについて十分注意する必要がある。

### 6.1.3 平均値，標準偏差，ヒストグラム

$c_j=\sum_{i=1}^{N} a_{ij} b_i \quad 1\leq j\leq N$ で定まる$c_j$ の平均値，標準偏差を求める。$c_j$ を計算する前に$a_{ij}$のうち或る値$p$より小さいものを$q$で置き換える。さらに，$c_j$ の値域をM等分してヒストグラムを作成する。以上のことを行なうAP－プログラムは次のようにコーディングできる。

```
      DIMENSION A(N,N),B(N),
                C(N),H(M),MH(M)
      LOGICAL LG(M)
      INDEX IX/I,N/
      ………………………
      A(*,*)=AMASK((A(*,*).LT.P),
                   Q,A(*,*))
      C(*)=PMM(B(*),A(*,*))
      AV=VSUM(C(*))/FLOAT(N)
      SD=SQRT(VNRM(C(*))/FLOAT(N)-
                           AV**2)
      AMN=FMNV(C(*))
      BAND=(FMXV(C(*))-AMN)/FLOAT(M)
      LG(*)=.TRUE.
      H(*)=BAND*IDXL(LG(*))+AMN
      MH(1)=N
      DO 10 I=2,M
      MH(I)=IONC(C(*).GE.H(I-1))
   10 MH(I-1)=MH(I-1)-MH(I)
      ………………………
```

もし$c_j$ を大きさの順に並べる必要があれば，そのように並べ換えてからヒストグラムの作成を行なえばよい。そのときにはDO 10 の中を変える必要がある。例えば小さい順に並んでいるとして

```
      I=1
      DO 10 K=1,M-1
      J=IFGE(C(IX),H(K))-1
      I=I+J
   10 MH(K)=J
      MH(M)=N-IVSUM(MH(IY))
```

とするとよい。（インデックスIYは1，M－1）

### 6.1.4 巾乗法による固有値計算

$n$次実行列Aの固有値を絶対値の大きい順に並べて $\lambda_1, \lambda_2, \cdots\cdots, \lambda_n$ としたとき

$$|\lambda_1|>|\lambda_2|$$

となっていて$\lambda_1$ が実数の場合には巾乗法によって求めるのが甚だ簡単である。それには例えば次のようにコーディングすればよい。

```
      DIMENSION A(N,N),V(N)
      ……………
```

| Vに初期近似をセットする |
|---|

```
      V(*)=PMM(V(*),A(*,*))
      IM=IFMX(ABS(V(*)))
      XO=V(IM)
      V(*)=V(*)/XO
      DO 10 I=1,NMAX
      V(*)=PMM(V(*),A(*,*))
```

This document is provided by JAXA.

```
      XN=V(IM)
```

XOとXNを用いて判定
収束なら GO TO 20

```
      IM=IFMX(ABS(V(*)))
      XO=V(IM)
      V(*)=V(*)/XO
   10 CONTINUE
```

収束しなかった場合の処理

```
   20 CONTINUE
```

このプログラムにもいくつか修正することができる。例えば始めに

```
      V(*)=PMM(PMM(PMM(V(*),A(*,*)),
                   A(*,*)),A(*,*))
```

と何度か行列積を行なってしまう。上記の場合では$A^3V$が計算されるので，これがオーバーフローしたりする危険があってはならない。別な修正として，数度繰り返した後では最大値をとる位置が変わらなくなるので毎回毎回IFMX をする必要はない。又このプログラムでは絶対値最大要素が±1になるように正規化しているが必ずしもその必要がない。

数個の固有値を計算するにはサブスペース法に依ったり，或は前記の1つずつ求めるアルゴリズムをそれまでに求めた固有ベクトルに関する成分だけを除去するように修正したりすればよい。どの場合にも，固有値が実でなければならないし，絶対値で分離されているのが簡単である。

### 6.2 ミクロ的なコーディング例

#### 6.2.1 型変換－FLOAT－について

CPには固定小数点数を浮動小数点数に変換したり，浮動小数点数を固定小数点数に変換する機械命令があるが，APにはこれに対応する命令がない。APでこのような型変換を行なうには2通りの方法があって，スカラー変数を対象とする場合にはM＆R型命令を組み合わせて行ない，ベクトル変数の場合には機械命令のVMVがこれを行なう。M＆R型による型変換は，変換個数が5個程度でもV型による場合の数倍の時間がかかる。従って，実数を整数に変換するにはいくつかをまとめて行なう方がよい。又，DOの制御変数などを実数化して用いる場合等のように，整数を実数として計算に使用する際には，それが頻繁に現われるなら必要となる大きさにまで整数を実数化したものを用意しておくのがよい。さらに，いくつかのサブプログラムでも使用する場合にはCOMMON で共通化しておくのもよい。以下にそのプログラム例を示す。まず，1からNまでの整数を実数化したものを作るには次のようにする。

```
      DIMENSION FLT(N),LG(N)
      LOGICAL LG
      LG(*)=.TRUE.
      FLT(*)=IDXL(LG(*))
```

これがよく使われる場合として，区間$[a,b]$を$n$等分した点

$$x_i=a+ih \quad 0\le i\le n,\ h=(b-a)/n$$

の計算がある。例えば，点$x_i$における函数値の計算等である。この場合CP－FORTRANだと

```
      ………………
      X=A
      H=(B-A)/FLOAT(N)
      DO 10 I=1,N+1
      F(I)=…………
   10 X=X+H
```

のような形になる。これがAP－FORTRANだと，最初の例のように1～N＋1の浮動小数点数をFLTに作っておいて

```
      X(*)=A+((B-A)/FLT(N))*(FLT(*)-1.0)
      F(*)=……………
```

とすればよいことになる。以後のプログラム例ではすべてFLTというアレイにはその大きさに相当するだけの浮動小数点数が作られているものとする。

これによると，例えばCP－FORTRANで書かれた

```
      DIMENSION A(N),B(N),C(N)
      DO 10 I=1,N
      P=X*FLOAT(I)-Y
      A(I)=P+S
      B(I)=P**2+T*P+U
   10 C(I)=(A(I)+B(I))*0.5
```

というプログラムは，AP－FORTRANで

```
      DIMENSION A(N),B(N),C(N),
                FLT(N)
      A(*)=X*FLT(*)-Y
      B(*)=(FLT(*)+P)*FLT(*)+U
      A(*)=A(*)+S
      C(*)=AVRG(A(*),B(*))
```

と書き換えることができる。

This document is provided by JAXA.

### 6.2.2　EQUIVALENCEの用法とVRLOADの仕方

EQUIVALENCEの用法としては幾つか考えられるが，まず第1に5章で述べたように多次元アレイを1次元アレイと同一視することによってベクトル化を可能にする用法がある。これは，多次元アレイというものがFORTRANが提供してくれる1つの便利さであって，計算機内ではすべて1次元的アレイであることを考えれば自然な事柄である。次に考えられることは，小さいアレイやスカラー変数を大きくアレイとEQUIVALENCEで結ぶことによって，有効長のあるベクトルとすることである。これは，プログラムを作るに当り，プログラムを作りやすくしたり或は見やすくしたりするために，物理的又は工学的変数の各々に変数名を付け，それがために一見したところベクトル化できないようなものを作り出すことになっているからである。もう1つはこれと同工異曲であるが，ベクトルレジスタを自分で使おうとする場合に，やはりプログラムを見やすくしたり，コーディングしやすくしたりするのに有効に利用できる。さらにリストベクトルを使用するときにも必要となることがある。それは，リストベクトルは1次元アレイにのみ使用可能なので，2次元以上の多次元アレイに対してリストベクトルを用いようとすると，その多次元アレイを1次元アレイとEQUIVALENCEで結ぶ必要がある。その他にも，配列断面子添字“＊”を使えるようにするために用いる等種々の利用法が考えられる。以上のいずれの場合にも注意しなければならない最も重要なことは，主記憶上での配置に矛盾をきたさないことである。また第5章で述べた様にEQUIVALENCEはALLOCATEされた配列と結合することは許されていない。従って，EQUIVALENCEでアレイを結合して，効果的なベクトル演算（“＊”やインデックス変数を用いて）を行おうとするには，CPと違った意味でEQUIVALENCEの用法を考慮する必要がある。

$n$次単位行列を作る。

```
DIMENSION A(N,N),A1(n²)
EQUIVALENCE (A(1,1),A1(1))
INDEX IX/1,n²,n+1/
A(*,*)=0.0
A1(IX)=1.0
```

スカラー変数をつなげてベクトルとする例としては次のようなものが考えられる。

```
DIMENSION A(3),B(3)
S=A(1)*X+A(2)*Y+A(3)*Z+B(1)*VX
        +B(2)*VY+B(3)*VZ
```

というCPープログラムは

```
DIMENSION A(3),B(3),AB(6),XY(6)
EQUIVALENCE (AB(1),A(1)),
   (AB(4),B(1)),(XY(1),X),
   (XY(2),Y),………,(XY(6),VZ)
S=AIPD(AB(*),XY(*))
```

ベクトルレジスタを自分で使う場合の方法としては

```
DIMENSION VR(1792),A(100),B(100),
          C(100)
…………………………
EQUIVALENCE (VR(1),A(1)),
   (VR(101),B(1)),(VR(201),C(1)),
……………………
CALL VRLOAD(VR)
A(*)=X*S(*)
B(*)=A(*)+T(*)
```

というような方法，又は

```
DIMENSION A(100),B(100),AB(2),
          C(100)
………………
EQUIVALENCE (A(100),AB(1)),
            (B(1),AB(2))
………………
CALL VRLOAD(A)
A(*)=X*S(*)
B(*)=A(*)+T(*)
```

という方法がある。もう1つインデックスを用いて

```
DIMENSION VR(1792),………
INDEX IX/1,N/
CALL VRLOAD(VR)
VR(IX)=X*S(IX)
VR(IX+N)=VR(IX)+T(IX)
```

とする方法がある。

又，

```
XU=C1*P
XC=C2*P
ZU=C3*P
  ⋮
QD=C10*P
```

というCPープログラムは

```
DIMENSION C(10),X(10)
EQUIVALENCE (C(1),C1),
   (C(2),C2),………,(C(10),C10),
   (X(1),XU),………,(X(10),QD)
X(*)=P*C(*)
```

This document is provided by JAXA.

と簡単に書ける。或いは，すでにベクトル化されている

```
DIMENSION F(3),AV(6),AW(6)
F(*)=0.0
AV(*)=0.0
AW(*)=0.0
```

というプログラムでも

```
DIMENSION F(3),AV(6),AW(6),
          AX(15)
EQUIVALENCE (AX(1),F(1)),
   (AX(4),AV(1)),(AX(10),AW(1))
AX(*)=0.0
```

とすれば，ベクトル長も長くなりさらにベクトル命令の立ち上がり後処理も1回ですみ，ディスクリプタセットも1個ですむようになる。

多次元アレイにリストベクトルを適用する簡単な例として

```
      DIMENSION A(N,N),LST(N,N)
      DO 10 I=1,N
      DO 10 J=1,N
 10   A(LST(J,I)I)=…………
```

という式を考える。AP－FORTRANでは

```
      A(LST(*,I),I)=…………
```

という表現が許されていないので，Aをそれと同じ大きさの1次元アレイ$B(n^2)$とEQUIVALENCEで結び，(LST(J,I),I)に対応するリストをアレイLSU(N)に作る。以上をまとめると

```
      DIMENSION A(N,N),B(n²),
                LST(N,N),LSU(N)
      EQUIVALENCE (A(1,1),B(1))
      DO 10 I=1,N
      LSU(*)=LST(*,I)+N*(I-1)
 10   B(LSU(*))=…………
```

と変換することができる。

6.2.3 1次元アレイのワーキングエリアの活用とファンクション・サブプログラム

CPープログラムではワーキングエリアとしてスカラー変数で十分であったところでも，APープログラムでベクトル化を行なうためには，1次元又はそれ以上の次元のワーキングエリアを導入する必要のある場合がある。この際に注意すべきこととして，効率のよいワーキングエリアのとり方をして，不必要に多くのワーキングエリアをとらないようにすることである。それでないとAPープログラムはCPープログラムに比べて全体に大きくなる傾向にあるのをさらに助長させることになる。そのために，サブプログラムでもそのようなワーキングエリアを使用する場合にはCOMMONで共通化しておくのもよい。或はALLOCATE・FREE機能により，ワーキングエリアが必要となったときにALLOCATEし，不必要になったらFREEするという方法もある。この後者の方法では，ベクトルレジスタを自分で使おうとする場合に不便である。

次に幾つかのコーディング例を挙げるが，6.2.1で述べたFLOATに関する事項もこの1種と見なされる。その際にもコーディング例として挙げたが，DOの中で関数値計算を行なうことがよくある。このとき，ユーザーによってはファンクション・サブプログラムを作って関数値を計算させていることがあるが，それはベクトル化を行なうには最もまずい手法の1つといってよい。いろいろな理由によりどうしてもサブプログラム化したいときには，変数値をベクトルで渡し関数値をベクトルで受けるサブルーチン・サブプログラムにするのがよい。例えば簡単な例として

```
      DIMENSION A(N)
      S=0.0
      DO 10 I=1,N
      X=FLOAT(I)*Y
 10   S=S+A(I)*FUNC(X)
```

というようなものは

```
      DIMENSION A(N),FLT(N),X(N),
                F(N)
      X(*)=Y*FLT(*)
      F(*)=FUNC({X})……関数値計算をインライン化
又は  CALL FUNC(X,F)……関数値計算をサブルーチン・サブプログラム化
      S=AIPD(A(*),F(*))
```

とすることができる。1次元ワーキングエリアを使用する単純な例として，行列の行或は列の交換がある。これはCPープログラムで書くと

```
      DO 10 I=1,N
      X=A(I,K)
      A(I,K)=A(I,L)
 10   A(I,L)=X
```

となるが，APープログラムではワーキングエリアWORK(N)を用いて

```
      WORK(*)=A(*,K)
      A(*,K)=A(*,L)
      A(*,L)=WORK(*)
```

This document is provided by JAXA.

と変換できる。もう少し複雑な例として

```
      DIMENSI N A(N,N)
      AMAX=0.0
      DO 10 I=1,N
      S=0.0
      DO 20 J=1,N
   20 S=S+A(J,I)**2
      IF(AMAX.GE.S) GO TO 10
      AMAX=S
      K=I
   10 CONTINUE
```

というCP－プログラムは

```
      DIMENSION A(N,N),WORK(N)
      DO 10 I=1,N
   10 WORK(I)=VNRM(A(*,I))
      K=IFMX(WORK(*))
      AMAX=WORK(K)
```

と変換できる。

6.2.4　配列特殊関数に関する注意

1）配列特殊関数とベクトル添字

基本ベクトルにV型演算を施す場合，最小限度必要な情報はベクトルの先頭番地，跳び，ベクトル長，データの型である。ここで関係のあるのは前三者である。主記憶上ではアレイはすべて1次元的であって，多次元アレイは単にFORTRANが関与しているだけである。従って多次元アレイであってもその結果が主記憶上で1次元ベクトル的になればV型演算を施すことができる。（5.1.参照）例えば，FORTRAN文法書[3]にはIPD VNRMの説明の項に

```
    AIPD(A(*),B(*))
    VNRM(A(*))
```

のようにFORTRANでも1次元のアレイのことしか書いていない。しかし，

$$d=\sum_{i,j=1}^{n} a_{ij} b_{ij}$$

$$s=\sum_{i,j=1}^{n} a_{ij}^{2}$$

を計算しようとする場合には，A，BのDIMENSIONを丁度の大きさ(N,N)にとってあれば

```
    D=AIPD(A(*,*),B(*,*))
    S=VNRM(A(*,*))
```

のように表現することができる。この場合に注意する必要のあるのは，2次元以上の場合には複数個のベクトル添字は相続く“＊”だけが許されるという事柄である。

同様にPMMの場合には

```
    C(*)=PMM(B(*),A(*,*))
```

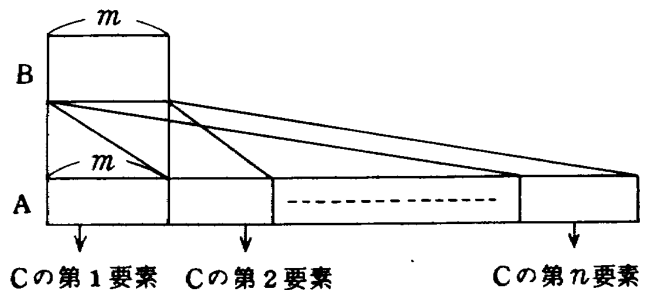
となっているが，これも

```
    C(*)=PMM(B(*),A(*))
```

と書くことも許される。この場合特に注意すべきこととして，ベクトルA，B，Cの長さをそれぞれ$l$，$m$，$n$とすれば

$$l=m\cdot n$$

という関係式を満たしていなければならないことである。概念的には下図のように書ける。（アレイプロセッサハードウェア解説書のV型命令の説明の項を参照）




2）最大・最小要素位置及び比較要素位置（IFMX，IFMN，IFMP，IFEQ，IFNE，IFGE）

データ種類としては基本ベクトル，リストベクトルの2通りが許されているが，それぞれ別個に説明する。まず基本ベクトルの場合であるが，この場合には対象となっているベクトル要素だけを取り出して新しく1つの基本ベクトルを作り，そのベクトルに各配列特殊関数の操作を施した場合の値が出力として出される。そのために，元のアレイでは何番目になるか修正する必要のある場合がある。以下にIFMXを例にとって詳しく述べる。

```
    A=(1,2,3,4,1,2,3,4,1,2)
```

としたとき，

```
    DIMENSION A(10)
    INDEX IX/1,8/
    I=IFMX(A(*))
    J=IFMX(A(IX))
```

のような配列断面子添字“＊”や，初期値が1で跳びが1のインデックスIXによる場合には問題がない。即ち，この場合には I＝4，J＝4 が出力されてくる。注意しなければならないのは，初期値が1より大きかったり，跳びが1より大きいインデックスを用いた場合である。上例を使えば，

```
    INDEX IX/3,10/,IY/1,10,2/,
```

This document is provided by JAXA.

```
      IZ/2,10,2/
   I=IFMX(A(IX))
   J=IFMX(A(IY))
   K=IFMX(A(IZ))
```

とした場合には，それぞれ I=2，J=2，K=2 となるが，実際のアレイでの要素位置はすべて同じく4である。

これからもわかるように，インデックスを

```
   INDEX IX/m1,m2,m3/
   I=IFMX(A(IX))
```

とした場合の実際に最大値をとる要素位置ITは

```
   IT=m3*(I-1)+m1
```

で与えられる。リストベクトルの場合も同様に考えればよく，

```
   INDEX IX/m1,m2,m3/
   I=IFMX(A(LST(IX)))
```

というプログラムの場合，実際の最大要素位置は

```
   LST(m3*(I-1)+m1)
```

で与えられる。例えば前の例で

```
   LST=(1,3,6,4,8,10,5,7,8,2)
   INDEX IX/2,10,3/
   I=IFMX(A(LST(IX)))
```

とすれば

```
   (1,3,2,4,4,2,1,3,4,2)
```

とリストで並べ換えられた要素のうち，インデックスで指定された

```
   (3,4,3)
```

の中から最大値が探され，その最初に見つかったものの位置がIにストアされる。今の場合だと I=2 になる。これがAの何番目の要素に対応するものかといえば，K=LST(3*(2-1)+2)=LST(5)=8 番目の要素である。

3) その他

論理ベクトルを作るときの関係演算子としては，"EQ" "LT"，"GT" のどれかになるようにするのが好ましい。上記のものだと機械命令は1つしか作られないが，"NE"，"LE"，"GE" の場合には2つの機械命令が作られる。もちろん他の部分との相関で決められる必要がある。

IFBON，IFBOFF はベクトル命令は使用しておらずスカラー命令によって実行されている。例えば

```
   L=IFBON(LG(*))
```

という内容をベクトル命令によって行なうとすれば，

```
   IS=IONC(LG(*))
   IF(IS.EQ.O) GO TO 10
   INT(*)=IDXL(LG(*))
   L=INT(1)
   GO TO 20
10 L=O
20 CONTINUE
```

となろうが，平均的にはスカラー命令による方が速く処理できる。但しIFBON，IFBOFF はその引き数が必ず配列断面子添字 "*" による論理型アレイでなければならない。インデックス添字を用いたり，論理式を書いたりするとコンパイルエラーになる。もしそのようなことをする必要が生じた場合には，前記のように "IONC" と "IDXL" を用いるか，必要となるアレイの大きさを計算してその大きさ分の論理型アレイをALLOCATE 配列として確保するかである。後者の例として，

```
   LOGICAL LG(LDM)
   INDEX IX/1,M/
   ……………………
   LDM=M
   ALLOCATE LG
   LG(*)=A(*).LT.S.AND.B(*).GT.B
   K=IFBON(LG(*))
   FREE LG
```

が考えられる。この場合論理式が

```
   LG(*)=A(*).LT.S
```

と単純な場合には比較要素位置を求める配列特殊関数IFLTが使える。

#### 6.2.5 DO文中のIFの処理例

DO文中のIF文の型として2つのものを考えることができる。1つはDOの制御変数のみがIF文の分岐を支配している場合，もう1つはそれ以外の論理式によって分岐を行なう場合である。前者については，インデックスを適宜に作ること等によって概ね処理できる。後者については2.2に基本的な変換方式とその処理速度に関して詳述してあるが，ここでは幾つかの具体的なコーディング例を示す。

1) DOの制御変数による分岐が現われる主な場合は，境界部分の計算だけが内部の計算と異なっているような処理が必要となるときである。1次元の場合の例として

```
   DIMENSION A(N)
   DO 10 I=1,N
   IF(I.NE.1) GO TO 11
   A(I)=……………
   GO TO 10
11 IF(I.NE.N) GO TO 12
   A(I)=……………
```

This document is provided by JAXA.

```
      GO TO 10
   12 A(I)=……………
   10 CONTINUE
```

というCP−プログラムは

```
      DIMENSION A(N)
      INDEX IX/2,N-1/
      A(IX)=……………
      A(1)=……………
      A(N)=……………
```

と変換できる。(CP−プログラムでもこのベクトル表現をDO文化したものの方がずっと処理速度が速い)，細かい注意として，A(1)，A(N)の計算式が内部部分の計算式と殆んど同じような場合，例えば最も簡単なものとして

```
      A(IX)=f
      A(1)=f+1
      A(N)=f-1
```

のような場合には

```
      A(*)=f
      A(1)=A(1)+1
      A(N)=A(N)-1
```

とする方が勝っている。2次元の場合にはもう少し複雑になるが基本的には同じである。$f$，$g$，$h$は処理内容であるとして

```
      DIMENSION A(N,N)
      DO 10 J=1,N
      DO 10 I=1,N
      IF(J.NE.1) GO TO 11
      A(I,J)=f
      GO TO 10
   11 IF(J.NE.N) GO TO 12
      A(I,J)=g
      GO TO 10
   12 A(I,J)=h
   10 CONTINUE
```

というCP−プログラムを考える。これはAの第1列と第N列とが処理内容が異っている例である。このとき$h$という処理内容が配列断面子添字“＊”の使用を許すようなものであれば以下のように変換することができる。

```
      DIMENSION A(N,N),B(N,N-2)
      EQUIVALENCE (A(1,2),B(1,1))
      A(*,1)=f
      B(*,*)=h
      A(*,N)=g
```

或いは$h$の処理がさほど複雑でなく，その処理内容を第1列と第N列にまで拡張してもプログラム例外を起こす心配のないときには

```
      A(*,*)=h
      A(*,1)=f
      A(*,N)=g
```

というコーディングをすることもできる。このような配列断面子添字“＊”を用いることができないような処理内容である場合には1回のDO−ループがどうしても必要になる。このことは，Aの第1行と第N行とだけがその処理内容で異っている場合にも同じである。しかしここで考える必要のあることは，どちらを行方向とし，どちらを列方向とするかは殆んど習慣的なものであり数式表現をそのままプログラムに移した場合が多いことである。プログラムの他の部分との競り合いがなければ，始めから転置した形の行列を扱っているものとしてデータ構造を考えれば解消できる事柄である。

これらとは違ったタイプのものとして

```
      DIMESION A(N,N),B(N)
      ………………………
      DO 10 I=1,N
      S=0.0
      DO 20 J=1,N
      IF(I.NE.J) S=S+ABS(A(I,J))
   20 CONTINUE
   10 B(I)=S
```

というプログラムを考える。これは $b_i=\sum_{j\neq i}|a_{ij}|$ という計算だから

```
      DIMENSION A(N,N),C(n²),B(N)
      EQUIVALENCE (A(1,1),C(1))
      INDEX IX/1,n²,n+1/
      ……………………
      DO 10 I=1,N
   10 B(I)=VSUM(ABS(A(I,*)))
      B(*)=B(*)-ABS(C(IX))
```

と変換できる。

2）DOの中に論理式があってその真偽によって処理が異なる場合の例は2.2にも書いた通り，一般的には

```
      DO 10 I=1,N
      ……………………
      IF(e) GO TO 20
      ……………………
      GO TO 10
   20 CONTINUE
      ……………………
   10 CONTINUE
```

This document is provided by JAXA.

という形で表わすことができ，応用的にはこれが複雑な構造になっていることもある。単純な場合の例として

```
      DO 10 I=1,N
      A(I)=S*B(I)
      IF(A(I).LE.0.0) GO TO 20
      C(I)=A(I)/P
      GO TO 10
   20 C(I)=(F(I)/(D(I)+E(I)))*T
   10 CONTINUE
```

というＣＰ－プログラムは，配列特殊関数ＭＡＳＫを使えば，

```
      A(*)=S*B(*)
      C(*)=AMASK(A(*).GT.0.0,
               A(*)/P,
               (F(*)/(D(*)+E(*)))*T)
```

と書くことができる。この表現では長さＮのＶ型除算が２回実行される。これを

```
      LG(*)=A(*).GT.0.0
      IC=IONC(LG(*))
      ID=N-IC
      IA(IX)=IDXL(LG(*))
      IB(IY)=IDXL(.NOT.LG(*))
      B(IA(IX))=A(IA(IX))/P
      B(IA(IY))=(F(IA(IX))/(D(IA(IX))
                +E(IA(IX))))*T
```

とリストベクトルによる表現にすれば，除算は全部でＮ回となる。又，配列の要素が或る値より大きいときその値を別の値で置き換えるＣＰ－プログラム

```
      DO 10 I=1,N
      IF(M(I).GT.NA) M(I)=NB
   10 CONTINUE
```

は，ＭＡＳＫを用いて

```
      M(*)=MASK(M(*).GT.NA,M(*),NB)
```

と記述できる。この例と同じようであるが

```
      DO 10 I=1,N
      IF(ABS(A(I)).LE.EPS) GO TO 20
      X(I)=P/A(I)
      GO TO 10
   20 X(I)=0.0
   10 CONTINUE
```

を

```
      X(*)=AMASK(ABS(A(*)).GT.EPS,
                 P/A(*),0.0)
```

と変換することはできない。というのは，右辺に現われるベクトル"P/A(*)"を計算してしまうのでＡに零要素があったときには演算例外を起こしてしまう。そのため例えばリストベクトルを用いれば

```
      INDEX IX/1,K/
      LOGICAL LG(N)
      X(*)=0.0
      LG(*)=ABS(A(*)).GT.EPS
      K=IONC(LG(*))
      IF(K.EQ.0) GO TO 10
      LST(*)=IDXL(LG(*))
      X(LST(IX))=P/A(LST(IX))
   10 CONTINUE
```

と表わせる。

### 6.2.6 その他のＣＰ－プログラムのＡＰ－プログラムへの書き換え例

以下に挙げるのは主としてユーザのプログラムの中から例をとり上げたもので，始めにＣＰ－プログラムをその次にＡＰ－プログラムへの変換例を示す。

1)
```
      DIMENSION A(100),B(100),C(100)
      ……………………
      S=0.0
      P=1.0-A(1)*B(1)
      DO 10 I=2,N
      Q=1.0-A(I)*B(I)
      S=S+(P+Q)*0.5*C(I)
   10 P=Q
```

これはプログラムを見てもわかるが，元の計算式

$$S=\sum_{i=1}^{N-1}(1-(a_i b_i+a_{i+1} b_{i+1})/2)\,c_{i+1}$$

から，

```
      INDEX IX/1,N/,IY/2,N/
      S=AIPD(AJM(1.0-A(IX)*B(IX)),
             C(IY))
```

と変換できる。

2)
```
      ……………………
      DO 10 I=2,M
      S=0.0
      T=0.0
      DO 20 J=1,I
      K=1+I-J
      S=S+A(K)
   20 T=T+B(K)
      X(I)=S+U
      IF(IS.EQ.1) Y(I)=T+V
      IF(IS.EQ.2) Y(I)=T
```

This document is provided by JAXA.

```
  10 CONTINUE
```

このプログラムの DO 20のループはA，Bの和を

$s=a_i+a_{i-1}\cdots\cdots+a_1$

と求めているが，通常の順に和をとっていいから

```
     ……………………
     INDEX IX/1,I/,IY/2,M/
     DO 10 I=2,M
     X(I)=VSUM(A(IX))+U
  10 Y(I)=VSUM(B(IX))
     IF(IS.EQ.2) GO TO 20
     Y(IY)=Y(IY)+V
  20 CONTINUE
```

この変換においてISは必ず1か2のどちらかの値であると仮定した。

```
  3) ……………………
     DO 10 I=1,N
     M=I
     IF(E(I).NE.0.0) GO TO 15
  10 CONTINUE
     GO TO 30
  15 R=E(M)
     M=M-1
     DO 20 I=1,M
  20 D(I)=R
  30 CONTINUE
```

この変換例としては

```
     INDEX IX/1,N/,IY/1,M/
     M=IFNE(E(IX),0.0)
     IF(M.EQ.0) GO TO 10
     M=M-1
     D(IY)=E(M+1)
     GO TO 20
  10 M=N
  20 CONTINUE
```

非零要素が1つもないときにMにNを代入する必要がなければ，最後から3つ目と2つ目のスラートメントは必要がなく，始めから3つ目のスラートメントの“GO TO 10”は“GO TO 20”になる。

```
  4) DO 10 I=1,N
     K=NQ(I)
     IF(K.EQ.0) GO TO 30
     X=P(I)
     Y=Q(K)+D*(R(K)+S(K))
     IF(XM.GE.X.AND.XM.LT.Y)
                              GO TO 20
  10 CONTINUE
     GO TO 30
  20 CONTINUE
     ……………………
  30 CONTINUE
```

の変換例としては

```
     LOGICAL LG(LDM)
     INDEX IX/1,K/
     K=IIFEQ(NQ(*),0)
     IF(K.EQ.1) GO TO 30
     IF(K.EQ.0) K=N+1
     K=K-1
     WORK(IX)=Q(NQ(IX)+D*(R(NQ(IX))
                          +S(NQ(IX)))
     LDM=K
     ALLOCATE LG
     LG(*)=XM.GE.P(IX).AND.XM.LT.
                           WORK(IX)
     L=IFBON(LG(*))
     FREE LG
     IF(L.EQ.0) GO TO 10
     X=P(L)
     Y=WORK(L)
     GO TO 20
  10 X=P(K)
     Y=WORK(K)
     GO TO 30
  20 CONTINUE
     ……………………
  30 CONTINUE
```

となる。

以上いろいろと部分的にみた変換例を挙げてきたが，このDO－ループの中でなにをどのように計算しようとしているかを考えてAP－プログラムをコーディングすればさほど難しくはない。却ってCP－プログラムを変換するという立場の方が困難が多い。とりわけこのことはDO－ループ中のIF文を処理しようとするときにはっきりとしてくる。

## 7　結　　語

航技研に導入されたAPは科学技術計算において特徴的である多発するベクトル演算を通常の計算機の能力よりはるかに上廻る能力で処理できる様に設計された特殊な構造をもつ計算機である。従ってAPの性能を最高度に発揮させるためには，プログラムとデータの構造をこ

This document is provided by JAXA.

の特殊性に適合する様にしなければならない。その様なプログラムを作成するためには利用者はAPの構造とともにAPがどの様な場合にどの程度の処理能力を発揮するかを心得ておく必要がある。プログラム作成にあたってこうした知識を要求することは計算機利用の一般的趨勢に反する様であるが，APを使いこなすことによって我々が手にする処理能力が次の世代の最高の汎用機の能力にほぼ等しいか或いはそれを超えていることを考慮し，APの様な特殊計算機が現在および将来の科学技術計算の需要に答えて数多く開発されようとしていることを考慮するならば，この様な要求に満足させる様努めることは研究者にとって必要なことである。

本資料は上記の観点から

(1) 機械命令の実行速度の算出式を確定し，それに基ずいて算出された値と実測値を照合することにより算出式の信頼性を保証し，機械命令の実行速度を示すこと。

(2) プログラムレベルでのAPの実行速度の算出式を求めプログラムがAP向きであるか，CP向きであるかを定量的に判断することを可能にすること。

(3) 配列特殊関数，基本外部関数，基本的DOループパターンに関するAPの実行速度の実測値を示し，それらに対するFACOM-230-75の実行速度との比較を行ない，計算機利用者にAPの実行速度に対する目安を与えること。

(4) 上記の結果に基ずきAPの性能を十分に発揮させるためのプログラム技術のうち重要と思われるものを示すこと。

を行なった。

稿を終えるにあたり，CP-プログラムをAP-プログラムの変換において多大の労をお願いした富士通㈱LP部鈴木滋氏，棚倉由行氏他FA課の諸氏，ならびに電算機第一技術部三輪修氏，内田啓一氏に感謝の意を表したい。表(3.2.1)および表(3.2.2)のプログラム変換ならびに実行速度の計測は富士通㈱FA課諸氏によるものである。

## 文献

1) 三好　甫他；FACOM230-75アレイプロセッサーについてI
　　ーハードウェアの性能ー
　　(TM-325)

2) 富士通㈱；アレイプロセッサ・ハードウェア解説書

3) 富士通㈱；FACOM M-VII AP-FORTRAN 文法書

4) 富士通㈱；FACOM M-VII AP-FORTRAN 使用手引書

This document is provided by JAXA.

**航空宇宙技術研究所資料 369 号**

昭和53年10月発行

発 行 所　航 空 宇 宙 技 術 研 究 所
東 京 都 調 布 市 深 大 寺 町 1880
電話武蔵野三鷹(0422)47-5911(大代表)〒182

印 刷 所　株 式 会 社　実 業 公 報 社
東 京 都 千 代 田 区 九 段 南 4 — 2 —12


This document is provided by JAXA.

Printed in Japan

This document is provided by JAXA.