プログラム・ノート

利用者から見た計算機利用における効率
＝FORTRAN＝

研究開発部 磯 本 征 雄

1．はじめに

現在，大型計算機の利用状況は非常に活発であり，利用者数も年を追って増加している。また，これにともなって熟練者数が多くなっている事は確かでしょうが，反面に未熟練者数も増加している。そしてこれら両者の間の熟練度の差はますます大きくなる一方である。当然ここで問題にする「効率」についての理解度も利用者間で大きな差異を生ずるであろう。一方，利用者個々人が計算機をより効率よく利用する事は，単に個人的に有利であるだけでなく，現在のような共同利用の計算機においては全処理量の増加にむすびつき，ひいてはターン・アラウンド・タイムの短縮を可能にする場合さえあり，大いに望まれるところである。したがって計算機を効率よく利用する方法を熟練者だけでなく広く利用者一般が知り得るものにする必要がある。

熟練者にとっては，過去につくられたプログラム又は現在つくられているプログラムに対して「現用システム利用において効率の良いプログラム」といえるか否かの経験的かつ直観的判断がかなり可能です。しかしこのことはあくまでも豊富な経験をもつ熟練者においてのみ可能であり，未熟練者にとっては容易には知り得ないのが現状のようである。これは「効率」が現実には必ずしも一義的に意味づける事の容易でない事に原因していると思われる。国語辞典には[1]「効率＝機械によってなされた有用な仕事の量と機械に供給された全エネルギーとの比」であると説明されている。また計算機用語辞典[2]には「効率(efficiency)＝サービス能力 service-ability)」と説明されている。同辞典のサービス能力の所を見ると，計算機の信頼性・サービス率・使用効率・稼動率等の如き計算機を操作・運用する立場に立って「効率」が考えられている。したがって計算機利用者の立場で「計算機利用効率」を考え，意味づけることは未だなされていないと言っても過言ではないだろう。

敢えて言えば，当大型計算機センターのセンター・ニュースに掲載された「フォートラン700における算術計算の速度(No. 10)」，「オーバーレイ構造のプログラム(No. 11)」等の説明は，計算機利用における効率向上のためのものである。部分的にではあるが，これらの利用により計算機利用効力を高めるための一助となるだろう。しかしやはり広い意味での計算機利用効率を考えたとはいえない。

1）広辞苑，新村出編，岩波書店。
2）A チャングー他著，坂井利之訳，コンピューター用語辞典，講談社。

一般に計算機利用者は，自分自身の作っているプログラムに対して意識的か，無意識かにかかわらず，より効率の良いプログラムである事を目標に作っている。そしてプログラムを作る過程においてすら，より効率よく処理してゆく事を望んでいるにちがいない。ところが「効率」は，個々の計算機利用の立場，プログラムの使用目的・方法，プログラムの需要度及びプログラム自体のアルゴリズムや規模等と密接に関連している。これらすべての吟味なしにプログラム作成の過程あるいはプログラムの実行について，「効率」の良し悪しを局所的に判断・批判する事は必ずしも正しくない。また利用者間の立場の違いによる計算機利用方法の差異などのために，「利用効率」の問題は，単純に衆議一致し得るものでもない。このように「効率」の意味する内容は，かなり流動的といえる。しかし効率向上という問題が個々人により達成されている現状を見ていると，計算機利用における「効率」というものが幾分のあいまいさをもつにしても，具体的意味のある，初心者にも容易に理解（学習）し得る部分をもっている事がわかる。

このような状況のもとでは，「利用者側からみた計算機利用効率」とは何かを考えなおしてみる事は充分に意義があり，必要な事でもある。先にも述べた様に，効率は利用者の間にも判断に大きな違いがある。しかし異る立場の人達の異質の意見を相互に交換し，これらをまとめてゆくならば，そこに「客観的かつ科学的根拠のある，より広い立場での『効率』の意味づけ」を可能にする事もできることと考えている。

このような主旨のもとに今号より数回にわたり「利用者からみた計算機利用効率」について，本センター・ニュースにおいて誌上討論の場を設ける事にしました。今号第1回目は，筆者が問題提起者として本文を書く事になった次第です。以下，私的経験にもとずく「計算機利用効率」についての私の考え・意見をまとめたものです。読者諸兄の御批判・御意見・御感想の投稿をおまち致します。なおこの本文中で示したプログラム実行時間はいずれもNEAC2200シリーズ・・Model 700による結果です。なお“仮想記憶”とか“フォートラン・オプティマイゼーション”等と称せられたある種の効率向上のための手続きが計算機自体の機能として現実に働いているシステムもある。これらは利用者側から見て少し距離があるのでここでは考えない。

2．処理過程と効率

効率とは，一般に経済性を問題にしたものである。したがってより少い労力と少い経費でより多くの仕事をなし得たならば「効率は良い」と言える。ところが計算機利用者にとって，労力・経費・仕事の各々の意味する内容と意義は，利用する人自身のプログラミングの能力・経済力及び利用している計算機の性能や，さらに計算機利用の方法・目的等により非常に違う。反面，プログラム作成からプログラム使用に致るまでの過程はほとんどの場合，計算機利用者間で異る事はない。

「公式」を「計算機用プログラム」化し，これを「利用」するまでの過程を模式図に示したの

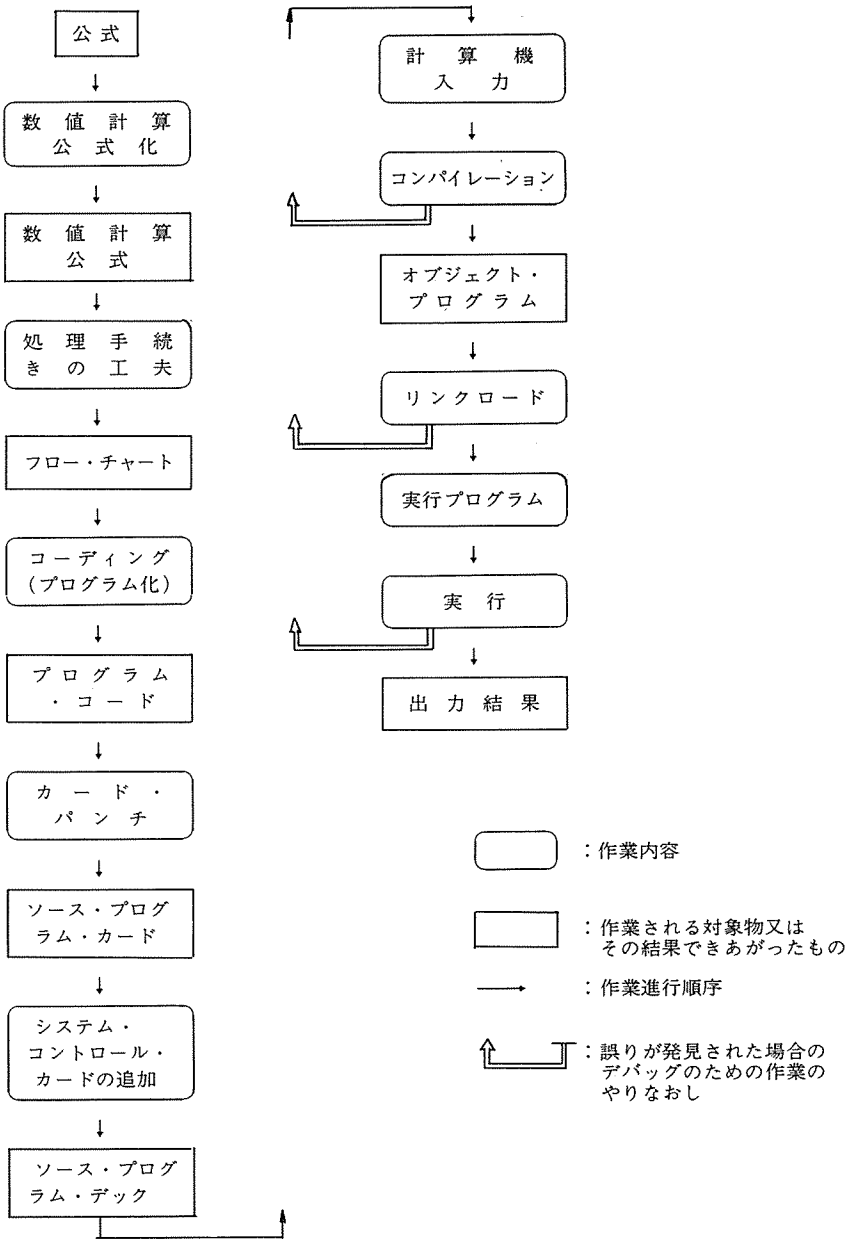

図１　計算機利用における作業手順

のが図１である。図中において，［　］には作業内容（労力）が示されている。［　］には作業される対象又は作業の結果できあがた「物」を示してある。→は作業の進行順序を示してある。また図中 ⌐ は，この部分でプログラムに誤りが発見され，デバッグのために作業手順として前の段階へ返る事を示す。以下で図１に即して処理過程について考える。

最初に計算（処理）されるべき［公式］が与えられる。通常数学的に与えられた公式は，

計算機処理による数値計算に適しているとは限らない。例えば微分方方程式の場合には，微分を差分で近似する事により数値計算用の近似公式をつくらなければならない。このように公式を (数値計算公式化) する事により [数値計算公式] を得る。数値計算公式にはまだ具体的計算手順が考慮されていない。すなわち数値計算公式に対する具体的な (処理手続きの工夫) をする必要がある。この結果，作業手順が [フロー・チャート] として表わされる。次に変数・配列名などの記号を決め，四則演算の適切な機能（表現）を用いてフォートラン・プログラムを作る。この結果を (コーディング) して [プログラム・コード] を作成する。以上で机上でのプログラム作成手順は終る。この段階までの構想が以下でのデバッグ及び実行時の効率に大きく影響する。コーディングシート上のプログラム・コードに従って (カード・パンチ) をおこない，[ソース・プログラム・カード] をつくり，さらに (システム・コントロール・カードの追加) をおこなって，[ソース・プログラム・デック] を準備する。この時のコントロール・カードとは，コンパイレーション，リンクロード及び実行用のものである。

文字通り計算機が利用されるのは，次の (計算機入力) の段階からである。ソース・プログラム・デックが計算機に入力されると第１段階で (コンパイレーション) がなされる。この時文法違反のチェックがなされる。文法違反があった場合には，エラー・メッセージが出力されてジョブは中断される。同時にソース・リストが出力される。この状態に対して利用者は [プログラムコード] にまでたちかえり，あるいはソース・プログラム・リストを見てソース・プログラムを修正しなければならない。但し多くの場合修正は各文・各行ごとに簡単におこなえる。ダイアゴノスティック・エラーもなく，コンパイレーションが無事完了すれば機械語への翻訳がおこなわれ [オブジェクト・プログラム] がつくられる。次の段階でオブジェクト・プログラムをもとにして (リンクロード) が試みられる。この時，複数個のサブプログラム間の連結がうまくいっていない場合はリンクロードは完了せず，エラー・メッセージの出力と共にジョブ処理は中断される。多くの場合，ジョブ全体についての流れ図をもう一度しらべながら誤りの個所を発見し修正しなければならない。したがってフロー・チャートの段階にたち返りプログラム全体にわたって見わたす必要がある。このため修正は困難となる。リンクロードが誤りなく完了すれば，[実行プログラム] がつくられ，主記憶装置上へのロードがおこなわれる。

主記憶装置上にロードされたプロ々ラムは (実行) に移る。そしてソース・データの入力がおこなわれ，計算処理が実行され [計算処理結果] が出力される。計算実行中に異常が発生しジョブ処理が中断される場合がある。この原因には，入力データの誤りの場合とプログラムの誤りの場合がある。いずれの場合もフロー・チャートに立返ってプログラムをしらべ，プログラム又はソース・データの誤りを修正しなければならない。一方，計算が無事終了したようにみえた場合にも，その出力結果が論理的に理解できない，あるいは明らかにまちがった結果を出力する場合もある。この時にはフロー・チャートのチェックはもちろんの事，数値計算公式自体にも誤りがないかをしらべなければならない。このようにプログラム作成手順全搬に

わたって再検討しなければならないため，実行結果出力の段階で発見される誤りの追跡・修正は一般にむつかしい。このような確認がなされた後にプログラムの出力結果は正しいものである事が保証される。以上の手続きの内でプログラムの誤りの修正がいわゆるデバッグである。

これらデバッグの作業自体も計算機利用の１段階である。したがってこの手順をいかに手ぎわよく処理するかという事も計算機利用の効率を考える上で重要である。デバッグの完了したプログラムは実行プログラムとして保存し，以後はプログラムの実行のみをくりかえしおこなう事ができる。

従来，計算機利用における効率について問題にされてきたのは，実行時における計算機の処理効率である。すなわち実行時にプログラムは，はたして最も望ましい手続きで計算を実行しているか否かが問題となる。理解を容易にするために次の例を考える。「４を 20000 回加えよ。」この命令を文字通りに実行すれば19999回の加算の後に答として 80000を得る。ところが文字通りに命令を実行しなくても掛算を知っている者には，４に 20000 を掛けたほうがはるかに速く，しかも全く同じ結果に達する事ができよう。もう一つの例を考える。「$\sum_{n=0}^{100} n!$ を計算せよ。」この計算を文字通り忠実におこなうために，まず n! の値を n＝１から n＝100 まで個々に計算し，後にこれらの和をとって計算結果を得ることができる。ところが，元の式を書きかえて

$$\sum_{n=1}^{100} n! = ((\cdots(((100+1)\times 99+1)\times 98+1)\times\cdots)\times 3+1)\times 2+1$$

とするならば積は98回，和は99回ですみ，前者のものより演算回数は和積共に少く，しかも数表をつくる必要も省けるため費す紙面も少くてすむ。計算機による計算においても事情は全く同じで，このように同じ公式から出発して同じ結果に至るまでに，その過程において「効率」の点で大きく違ういくつかの方法が存在する場合がある。したがって**実行時**の計算処理効率は大いに工夫する事が必要である。

このようにデバッグ及び実行時の効率が主として中央処理装置及び主記憶装置と利用者の労力を中心に考えられるのに対して，周辺機器をも含めた利用効率も考える事ができる。但し一般に周辺機器は何らかの形で常に使われているものなので，ここでは特に利用者側で制御可能な部分を中心に考える。

CPU-Time が中央処理装置の稼動時間であるのに対して Elapsed-Time は計算機及び周辺機器をジョブが占有している時間である。CPU-Time と Elapsed-Time の差の主要な原因は入出力機能に起因する。すなわち，中央処理装置で超高速度に計算処理が実行されてもソース・データの入力及び計算結果の出力が低速であれば計算機システム占有時間は後者によって決まってしまう。(但し，現実にはジョブ処理は同時に１-ジョブのみとは限らないのでこの点が全面的に効率低下につながるわけではない。) 特にライン・プリンターへの過多な出力等は，紙の消費の面だけでなく，ジョブの計算機システム占有時間を著しく長くする。したがって効率を

考える場合には入出力の量を注意深く吟味する事が望まれる。さらに周辺機器の利用方法に補助記憶装置（磁気テープ，ディスク・パック，磁気ドラム）を実行中の作業領域用に使うものと，プログラムの預け入れ用に使うものがある。このように高価な主記憶装置の使用をより安価な周辺機器で代用する事により，例え時間的損失が少しあっても全体としてさらに“効率のよい”ジョブ処理が可能となる。

以上，これまでに述べてきた計算機利用における処理過程での人的・機械的労力と効率について，大きく分けて次の３点にまとめられる。

(1) デバッグの効率的処置。

(2) 実行時の効率的処理。

(3) 周辺機器の効率的利用。

(1), (2), 及び(3)の各々は相互に密接に関連している。また場合によっては相互に矛盾し合うこともある。したがって計算機利用における効率の問題を考える場合には，これら(1), (2)及び(3)の内のいずれをより優先的に考慮するかを状況に応じて考えなければならない。またこれら優先度の順位のつけかたの違いにより，自ら具体的処置も違ってくる。ここでは(1), (2)及び(3)の相互の関係には多少目をつむって，それぞれの場合の効率向上について個々に独立に考える。

３．デバッグ時の効率

デバッグは元来誤りさえ犯さなければ必要のない事である。しかし現実にはプログラム作成の過程においてほとんど避けて通ることはできない。そして多くの場合時間と手間のかかる作業でもある。プログラム作成の過程とは図１において上から下へ向って矢印に従って順次下ってゆくことである。これに対して，デバッグは途中から上へ引返して誤りを修正した後に元の段階にまでもどってくる過程である。この引返してからもとの段階にまでもどってくるまでの“デバッグ”の作業をどれほどはやくおえるかは，作成されるプログラムの構造に依存し，また作成者のプログラミングの力にもよる。したがってプログラム作成の工程においてこのようなデバッグの煩わしい，しかし重要な過程がある事について充分に配慮しておく事が必要である。プログラミングの力は一般にはデバッグの時点においてはどうする事もできない。しかしプログラムの構成を工夫する事によりデバッグの手間をある程度簡略化できる。この工夫は，フロー・チャート作成の段階までに充分に考察しておかなければならない。

デバッグを簡略化させるためには次の事項に留意する事が必要であろう。

(1) プログラム実行の流れは単純なものにする。

（１－１）DO-ループが複雑な入れ子になっているもの，IF文やGO TO文により流れが複雑に交わっているもの等のように分岐点が多かったり，流れの複雑なものは誤りを犯しやすい。誤りの存在が発見された場合にも，修正すべき個所を具体的に追跡してゆく手続きが複雑となり，デバッギングを著しく困難にする。したがってこのような事は避ける。

（1－2）同一又は類似のアルゴリズムが何度も重複して使われている場合には，その部分をサブプログラムにして流れを単純化する。

(2) プログラムは論理の切れ目ごとに適当な大きさに分割し，ブロック化しておくとよい。これによりプログラムの正しさの確認を各々の部分ごとに行なうことができて，デバッグの際の手間が簡単になる。

(3) 変数名，配列名及び配列要素名等について，予め命名法を決めておく。この決められた命名法にもとずく変数，配列，配列要素名を用いるならば，DIMENSION文，COMMON文，仮引数及び実引数やプログラム本文中において混乱や誤解による間違いを犯す危険性を少なくできる。

(4) 周辺機器等による入出力文はなるべく単純な手法を用いる。もし入出力文のFORMATや使用方法が複雑な場合，プログラム中で異常が発見されてもその異常が入出力文の誤りによるものか，プログラムのアルゴリズムの誤りによるものであるのか判断を困難にし，結局デバッグに手間取ることがある。

(5) 数値計算用公式とプログラムの各行の間に容易に1対1の対応関係がつけられるようにしておく。注釈行の利用もその1つである。この事は，プログラム中のアルゴリズムの誤りを発見する上で非常に重要である。もちろん，公式とプログラム間の1対1の対応を明確にするには，公式自体がプログラム化されやすいように工夫されている事が必要である。

以上，これらの点はプログラム作成中における具体的手法というよりは，むしろ“心がまえ”のようなものである。これらが各人にとってどの程度具体的に実行され，うまく効果を発揮するかは，プログラミングの対象にもよるしさらにはプログラマーの能力にも依存する事と思う。

4．実行時の効率

実行時の効率向上について，“一般的に…が最良である。”と表現することは多くの場合不可能である。効率は使用システムの特性にもよるし，アルゴリズムの特性にもよる事である。さらにデバッグ時の効率的処置をも同時に配慮するなら，これらと矛盾する面をもつ事もある。従ってここでは，“どのような工夫が最良であるか”というよりはむしろ“どのような工夫の向づけがあり得のか”を考える事にする。そして具体的例題を通して“効率とは，例えば…の如くに向上させ得るものである”という相対的な方向づけが示せれば幸いである。

さて，プログラムの実行をいかに　率よくおこなうかという問題は，いかに速かに実行できるかという時間的側面と，いかに少ない（主）記憶の領域内で実行できるかという空間的側面から考える事ができる。両側面共に，プログラム実行の流れやアルゴリズムに関係した大局的側面と，各文又は各行における個々の言語機能に依存した局所的側面から効率を考える事ができる。これら個々のものは相互に必ずしも相補的役割をはたすとは限らない。特に空間的側面と時間的側面については相互に矛盾し合う事もあり得るので全体的見透しをつけた上で効率を考えなければならない。表1に，これら効率向上のために工夫し得る点をまとめた。以下各節

表１　実行後の効率向上の工夫

	a．大局的側面	b．局所的側面
時間的側面	1）処理結果が同じであれば処理ステップ数の少いアルゴリズムを用いる。	1）処理結果が同じであれば処理速度の速い機能を用いる。
	2）同じ計算処理（又はアルゴリズム）がプログラム実行中に重複して多数回実行される事を避ける。	2）同じ計算処理が重複して実行される事を避ける。
空間的側面	1）処理結果が同じであれば使用語数の少いアルゴリズム又は手法を用いる。	1）処理結果が同じであれば，使用語数の少なくなる行又は文の表現方法を用いる。
	2）同じアルゴリズムが重複してプログラム中にあらわれることを避け、不必要に多くのコアー・エリアを占有する事を避ける。	2）プログラム中で便宜的かつ１時的に使われる変数又は配列は他の部分にも重複して用いる。

ごとにこれらを具体的例題を通して考える。

4.1　時間的側面

表１に示したものは，いずれもごく当然の事にすぎない。しかし重要な事は，具体的問題に対してこれらをどのように適用するかという事である。表１に即して以下で具体的例題について工夫の例を示す。完全なものが示せた事になるか否かは読者の御批判をまつ次第です。

次の例題を考える。

〔例題〕次の式を計算するプログラムを作れ

$$\sin x = \sum_{n=1}^{\infty} \frac{(-)^{n-1} x^{2n-1}}{(2n-1)!}, \qquad |x| < 1 \qquad (1)$$

この公式は，$\frac{(-1)^{n-1}x^{2n-1}}{(2n-1)!}$ の値を $n=1, 2, \cdots, \infty$ まで加えよという事である。但しここでは $|x|<1$ と制限をつけておいたので，もし有効桁数10桁の計算機で計算する場合には14項以上では 10^{-11} 以下の値をもつのでこれらは切り捨てる。このような考えでつくったのが図２に示したプログラムである。但し図２では上に書いた公式に対して，必要以上に忠実(？)に

図２　公式に $\sin x = \sum_{n=1}^{13} \frac{(-)^{n-1}x^{2n-1}}{(2n-1)!}$ 忠実につくられたプログラム。

１回当りの実行時間は約3.23msec.

```
      SINT(NX,1)=X
      DO 1020 N=2,13
      FCTRL=1.0
      DO 1010 M=1,2*N-1
      FM=M
1010  FCTRL=FCTRL*FM
1020  SINT(NX,1)=SINT(NX,1)-(-1.0)**N*X**(2*N-1)/FCTRL
```

プログラムをつくった。このプログラムの実行時間は1回当り 3.23 msec. である。

図2のプログラムに対して，今大局的側面から見て処理ステップ数の少いアルゴリズムを用いる工夫をする。式(1)において和の中の第 n 番目と第 $(n+1)$ 番目の項の比をみると $\frac{(-)x^2}{(2n+1)(2n)}$ である。したがって x の因子については前項の x^2 倍，符号は前項とは逆である。n の因子については前項の $1/(2n+1)(2n)$ 倍である。このような理由から，和の実行中に前項までの各因子の値がのこされておれば，各々の因子について上記の値を掛ければよい。そして x^{2n-1} 及び $(2n-1)!$ を各項ごとに計算する必要はなくなる。この工夫を加えたのが図3におけるプログラムである。上記の工夫の結果，実行時間は0.365msec となり飛躍的に短縮される。

図3　計算のアルゴリズムを工夫してつくられたプログラム
1回当りの実行時間は約 0.365msec.

```
      SINT(NX,2)=X
*     X2=X**2
      XN=X
*     FCTRL=1.0
*     SGN=-1.0
*     DO 2020 N=2,13
*     FN=N
*     SGN=-SGN
      FCTRL=FCTRL*(FN*2.0-1.0)*(FN*2.0-2.0)
      XN=XN*X2
2020  SINT(NX,2)=SINT(NX,2)-SGN*XN/FCTRL
```

さらに大局的側面から，プログラム中で重複して同じ計算処理をおこなう事を避ける事により実行時間の短縮が可能となる。これは，全く同一の計算が重複して行なわれる場合だけでなく，類似した計算処理が2個所以上にあらわれ，それら相互の間に同じ部分が含まれる場合にも可能となる。例えば，1つのプログラム中で $\sin x$ と $\cos x$ が同時に計算される場合がそれである。$\cos x$ に対して $\sin x$ の場合と同じ考えでプログラムをつくると図4のようになる。これは，実行時間が0.365msec で図3と同じである。図3と図4のプログラムは，いくつかの文

図4　$\cos x = \sum_{n=0}^{13} \frac{(-)^n x^{2n}}{(2n)!}$ ，$|x|<1$ に対するプログラム

1回当りの実行時間は約0.365msec.

```
      COST(NX,2)=1.0
*     X2=X**2
      XN=1.0
*     FCTRL=1.0
*     SGN=-1.0
*     DO 3030 N=2,13
*     FN=N
*     SGN=-SGN
      FCTRL=FCTRL*(FN*2.0)*(FN*2.0-1.0)
      XN=XN*X2
3030  COST(NX,2)=COST(NX,2)-SGN*XN/FCTRL
```

図5　$\sin x = \sum_{n=1}^{13} \frac{(-)^{n-1} x^{2n-1}}{(2n-1)!}$,　$|x|<1$

$\cos x = \sum_{n=0}^{13} \frac{(-)^{n} x^{2n}}{2n!}$,　$|x|<1$　　の計算プログラム

実行時間は1回当り0.486msec である。

```
      SINT(NX, 3)=X
      COST(NX, 3)=1.0
      XN=X
      FCTRL=1.0
      SGN=-1.0
      DO 4020 N=2, 13
      FN=N
      SGN=-SGN
      FCTRL=FCTRL*(FN*2.0-2.0)
      XN=XN*X
      COST(NX, 3)=COST(NX, 3)-SGN*XN/FCTRL
      FCTRL=FCTRL*(FN*2.0-1.0)
      XN=XN*X
4020  SINT(NX, 3)=SINT(NX, 3)-SGN*XN/FCTRL
```

が重複している。(各々の図の中で左側に＊印をつけた行がそれである。) DO-ループの中でのFCTRL も両プログラムで互に類似している。これら類似あるいは同じ部分を整理し，図3と図4のプログラムを1つにまとめたものが図5である。実行時間は図3が0.365msec，図4が0.365msec であり両者合せて0.73msec であったものが図5のようにまとめる事により，0.486msec で実行できる。このプログラムの整理により処理時間0.244msec の節約が可能となった。SGN=-1.0とか FCTRLN=1.0等の単なる代入文であっても1回当り約1μsec 必要な点も注意しなければならない。

計算の重複を避けるのに，もう一つの工夫がある。$\sin x$及び$\cos x$の計算においてしばしばあらわれる $1/n!$ の計算をまとめて最初に計算し，以後はこれらを引用する。この手法により四則演算の回数は非常に少くなり，処理時間は短くなる。特に変数xのいろいろな値について計算する場合に有効である。但し，この場合にはそれだけの記憶領域を別に獲保しなければならないという不利もあるので状況を見定めた上でおこなうべきである。

実行時の処理効率を上げる際に，さらに局所的にプログラムをしらべる必要がある。例えば大阪大学大型計算機センター・ニュース No. 10 (1973年7月) に示された四則演算その他の処理速度の表でもわかるようにわずかとも思える表現の違いが大きな時間的差異をともなう。図6は配列 F(I)の値の計算を逐次 I＝1 から I=10000まで実行する場合の2つの例の比較を示したものである。左は IF 文によるもの，右は DO文によるものである。各々の処理結果は全く同じである。ところが IF 文を使った場合には144msec であったものが DO 文を使った場合には96msec になる。図7には積及びベキ乗算についていくつかの場合の処理時間の比較を示した。左側にはy^2の計算についての3つの異る機能での計算速度の比較が示されている。右側

図6　くりかえし機能と実行速度

```
   I=1
   X=1.0
10 F(I)=X*(X+1.0)
   I=I+1
   X=X+0.01
   IF(I.LE.10000) GO TO 10
```

実行時間 144msec

```
   X=1.0
   DO 15 I=1,10000
   F(I)=X*(X+1.0)
15 X=X+0.01
```

実行時間96msec

```
   DO 5 I=1,10000
 5 CONTINUE
```

実行時間32msec

図7　四則演算と実行速度

X=Y**2.0	111.2μSEC	X=Y*Y*Y*Y*Y*Y*Y*Y*Y*Y	13.7μSEC
X=Y**2	2.3μSEC	X=(Y*Y*Y*Y*Y)**2	8.2μSEC
X=Y*Y	2.4μSEC	X=((Y*Y)**2*Y)**2	7.1μSEC
		X=((Y*Y)**2)**2*Y**2	8.9μSEC
		X=(Y*Y*Y)**3*Y	8.5μSEC
		X=(Y*Y)**4*Y**2	8.7μSEC
		X=(Y*Y)**5	7.2μSEC
		X=Y**10	6.8μSEC

は y^{10} を積及びベキ乗で計算を実行した様々の場合の計算所要時間である。この図を見てわかるように，“より速い機能をより適切に用いる”ことは重要である。このような観点から図3に対して，その9行目における FN*2.0 を FN+FN に書き改める事により時間短縮が可能である。この工夫により図8のプログラムとなり，実行時間も0.343msec となっ0.022 m sec 時間の節約となる。

図8　図3のプログラムに対し，下から3行目 FCTRL=……において
FN*2.0→FN+FN に改めた。
このプログラムの1回当りの実行時間は0.343msec である。

```
      SINT(NX,4)=X
      X2=X**2
      XN=X
      FCTRL=1.0
      SGN=-1.0
      DO 5020 N=2,13
      FN=N
      SGN=-SGN
      FCTRL=FCTRL*(FN+FN-1.0)*(FN+FN-2.0)
      XN=XN*X2
5020  SINT(NX,4)=SINT(NX,4)-SGN*XN/FCTRL
```

最後に，一つの文の中で同じ計算処理が重複して実行される事を避ける事により効率を上げ得る事を指摘しておく。この事は，現実には良く見落されがちである。例えば図3又は図8に

おいて9行目の所にFN＊2.0又はFN＋FNの計算が2度あらわれている。これに対して7行目をFN＝N→FN＝N＋N－1と改め，同時に9行目をFCTRL＝FCTRL＊(FN＋FN－1.0)＊(FN＋FN－2.0)→FCTRL＝FCTRL＊FN＊(FN－1.0)と変える事により計算(FN＋FN)の重複を避ける。この改善の結果が図9のプログラムである。この工夫の結果実行時間は0.337 msecとなり，ここに示した他のプログラムのいずれよりも効率の良いものとなる。このように最適の演算機能を選び，重複を避ける事は具体的状況の中でプログラマー自身が判断しなければならないし，また常に留意する価値はある。

図9　図8のプログラムに対して7行目FN＝N→FN＝N＋N－1とかえ，9行目をFCTRLN＝FCTRL＊FN＊(FN－1.0)と変えた。
このプログラムの1回当りの実行時間は0.337msecである。

```
      SINT(NX,5)=X
      X2=X**2
      XN=X
      FCTRL=1.0
      SGN=-1.0
      DO 6020 N=2,13
      FN=N+N-1
      SGN=-SGN
      FCTRL=FCTRL*FN*(FN-1.0)
      XN=XN*X2
6020  SINT(NX,5)=SINT(NX,5)-SGN*XN/FCTRL
```

4.2　空間的側面

通常の計算機システムにおいて，主記憶装置は最も高価な部分に属する。したがって大きなプログラムを実行する場合，補助記憶でおきかえたほうが安価である。この事から，プログラム実行の際に主記憶装置の占有領域をできるかぎり少くする事をもって効率が良いと言う事ができる。この空間的側面から見た場合の効率も，その向上のための工夫の方向づけが，時間的側面から見た場合と同様に考えられる。表1の下段にこれらをまとめた。要するに主記憶装置上で使用される語数をできる限り少くする事に努める。大局的見地で効率向上の工夫をおこなうにはアルゴリズム全体にわたって工夫が必要であり，特に補助記憶（磁気テープ，ディスク・パック）を利用できるようにプログラムを作る必要がある。

このためには，プログラム化する前の数値計算用の公式をつくる段階から工夫が必要である。すなわち処理結果が同じであるならば使用語数の少いアルゴリズム又は手法を用いるように工夫する。さらにNEAC2200-Model700においては，プログラムを多重フェーズに分割し，オーバーレイ構造にする手法[3]がある。この手法により，必要とする主記憶装置の大きさを大幅に少くする事もできる。但しこの手法は計算機システムが異なる場合にも共通して使えると

3）NEAC2200シリーズModel700フォートラン700-プログラミング仕様説明書，日本電気。大阪大学大型計算機センター・ニュースNo.11('73, 7)

は限らない。プログラム中のデータ領域については，これらを主記憶装置上に常駐させず，ディスク・パックや磁気テープ上に書き出しておいて，必要な時に読み出して使う手法によりやはり計算機システムの有効利用を可能にする。

さらに効率をあげるためには，同じアルゴリズムがプログラム中に重複してあらわれるために不必要に多くのコアー・エリアを使う事を避ける必要がある。これに対する具体的工夫としては，何度もプログラム中にあらわれるアルゴリズムをブロック化し，サブルーチン又は関数とし，さらにENTRY文等を用いるのが常套手段である。このようにサブルーチンや関数にまとめるほどでもない程度のものであれば，文関数を用いたりIF文やGO TO文　を用いて流れを制御し，適宜同じアルゴリズムを何度も繰返し用いる工夫をするのもこの種の改善である。

空間的側面から見た効率向上（コアー・エリアの節約）においては，大筋は大局的な面でほとんど決められる。しかし各々の行及び文について局所的に工夫を凝らすことによっても効率向上への寄与はある。この場合には，言語の機能や，行や文の相互の関係に注目しなければならない。当然の事ながら，使用語数の少い文又は行の表現方法を用いるとよい。局所的にこれを見る場合，フォートラン言語から機械語への翻訳の問題として考えることができる。すなわ

図10　フォートランにおける表現形式と使用語数

(1) 代入文とDATA文

```
SUBROUTINE SUB12                          SUBROUTINE SUB11
A=2.0                                     DATA A, X, Y, Z/2.0, 1.0, 2.0, 3.0/
X=1.0                                     RETURN
Y=2.0                                     END
Z=3.0                          ⇨                                  34語
RETURN
END
        45語
```

(2) 変数のみの算術代入文

```
U=(((X+Y+Z)**2/A+Y+Z)*(Y+Z)/X)**3+((X+Y+Z)*4.0-A/(Y+Z))*(X+Y+Z)
                                  使用語数30語
                    ⇩
T1=X+Y+Z
T2=Y+Z
U=((T1**2/A+T2)*T2/X)**3+(T1*4.0-A/T2)*T1
                                  使用語数29語
```

(3) 組込み関数を用いた算術代入文

```
U=(EXP(X)*SIN(Y)+EXP(-X)*COS(Z))*(EXP(X)+EXP(-X))*(SIN(Y)+COS(Z))
                                  使用語数228語
                    ⇩
SN=SIN(Y)
CN=COS(Z)
EP=EXP(X)
U=(EP*SN+CN/EP)*(EP+1.0/EP)*(SN+CN)
                                  使用語数208語
```

ちフォートランとして少い行数でまとまった形式にプログラムを記述しても，機械語に翻訳された場合に逆に長い文となってしまっては効率を落す事になる。例えば図10におけるプログラムがその良い例である。図10には機能としては等価であるが，その表現の異なる具体例及び各々のオブジェクト・プログラムにおける語数を示してある。矢印はより効率のよい方向を示している。３例とも時間的側面から見た効率もよくなっている。例(1)は多数の代入文を並列にならべるよりは１つのDATA文にまとめる方がより有利である事を示す例である。例(2)及び例(3)は，文中に何度かくりかえしあらわれる式についてはこれらをまとめて１つの変数として取扱うほうが有効である事を示す例である。このように，複雑な文や行を無理やりに集約した形式で表わすよりは，比較的単純な文に分解して，何行かに分けて実行するほうが効率的な場合が多い。またフォートランとしてすでに集約された有効な機能があるならば，上記の事実にかかわらず，これら言語として集約された機能を用いるほうが効率的である。

局所的側面から工夫できる事として，さらに次の事がある。便宜的かつ一時的に作業用として使われた変数又は配列をできるかぎり他の同様の部分にも重複して用いる。

図11　作業用変数の有効利用

```
      SUBROUTINE SUB1
      COMMON U(150), V(150), K, L, X, Y, B, C
      DO 10 I=I, 100
      X=X+1.0
      T1=X*(X+1.0)
      F=FLOAT(I+K)
   10 U(I)=T1*F
C
      DO 20 J=1, 100
      Y=Y*2.0
      T2=Y*B+C
      G=FLOAT(J-L)
   20 V(J)=T2*G
      RETURN
      END
```

使用語数100語

⇨

```
      SUBROUTINE SUB2
      COMMON U(150), V(150), K, L, X, Y, B, C
      DO 10 I=1, 100
      X=X+1.0
      TEMP1=X*(X+1.0)
      TEMP2=FLOAT(I+K)
   10 U(I)=TEMP1*TEMP2
C
      DO 20 I=1, 100
      Y=Y*2.0
      TEMP1=Y*B*C +
      TEMP2=FLOAT(I-L)
   20 V(I)=TEMP1*TEMP2
      RETURN
      END
```

使用語数97語

但し，

```
      SUBROUTINE SUBO
      COMMON U(150), V(150), K, L, X, Y, B, C
      RETURN
      END
```

使用語数34語

図11は，この工夫のなされなかった場合となされた場合の例の比較である。図の左側のプログラムにおいてT1, F, T2及びGは作業用に用いられた変数である。またＩ及びＪはDOループの制御変数である。これらは，プログラムの前半と後半で独立しているので作業用に用いられた変数は全く同じ領域を使う事ができる。したがって

T1, T2　　→ TEMP1,
F, G　　→ TEMP2,
I, J　　→ I,

として各変数を重複させ1つにまとめる。このようにして変数の使用語数を少くする工夫をしたのが図11の右側である。この工夫により3語節約できる。同様の事が配列等に適用された場合には数千語の節約が可能となる場合さえある。図10の場合が表現方法の工夫によるものであったのに対し，図11の例はフォートラ・プログラムの表面にあらわれる使用語数を少くする工夫である。

5．周辺機器その他

前節までは主としてフォートラン・プログラムの範囲の問題であった。ところが計算機システムは主記憶装置及び中央処理装置のほかに様々の周辺機器からなっている。またこれらハードウェアだけでなく，「デバッグ文」とか「科学計算用ライブラリィ・サブプログラム」のようなソフトウェアの利用による効率向上をはかる事も可能である。その他様々のハードウェア，ソフトウェアが計算機システムに付随しているが，これらの利用方法は，利用者個々人の工夫にゆだねなければならない面が多い。これらのうまい利用方法については，むしろ今後利用者からこの欄へ投稿されるであろう原稿にゆだねたい。ここでは大阪大学大型計算機センターの現状に即した内容及び筆者の感じている事について書く。

補助記憶装置である磁気テープ及びディスクパックは，現在大阪大学大型計算機センターにおいては，ソース・データ・ファイルとして開放されソース・プログラムの保存用として利用者各位に使われている。また実行プログラムについては，ユーザーズ・マスター・ジョブ・ファイルとしてやはり利用者に開放さ れている。これらの利用によりプログラムのシステムへの入力の手間を簡略化し，またソース・カードの運搬等の労力を省力化する。また，マスター・ジョブ・ファイルに致っては，コンパイレーションやリンクロードの手続きを省略できる。但し，常時プログラムをシステム内に常駐させている点やその他の問題点がある事は否めない。したがって利用頻度の高い，しかも需要度の大きいプログラムでなければ逆にシステム占有時間が長いので効率を下げる。

磁気テープやディスク・パックは，すでに前節でものべたようにプログラム実行中のデータの中間書込みや，多重フェーズによるオーバーレイ構造プログラムの手法に用いられている。中間書込みにおいて，シーケンシャル・アクセス・ファイルとランダムアクセス・ファイルとでは仕様が異るのでこれらの違いをうまく使いわける事によりプログラム実行の効率を高める事が可能である。

デバッグ時における「デバッグ文」の使用，科学計算用ライブラリィ・サブプログラムの使用等によりプログラム作成の際の効率を高める事が可能となる。これらは常に留めておく価値はあると思う。

一般にプログラミングの際に問題にされる事の少いものにプログラム実行時のソース・データの入力や，実行結果の出力の書式がある。これらは計算機自体にとっては，極度に多くの価値の少い入出力がない限り，余り問題にはならない。しかし計算機利用者にとって入力ソース・データが簡単に作成できる事は，誤りを犯す心配も少く，また無駄な人的労力の消耗を避けることができるので仕事の効率を上げる。処理結果の出力についても同様であ。もし出力結果を再び整理しなおさなければならないとか，出力結果の配置の悪さから読みとりにくい等という事は，せっかくの計算機利用もその価値を減少される事になる。したがって出力結果がそのまま直観的判断にむすびつき，また最終的な整理の結果となっている事が望ましい。このような工夫は，特にプログラム作成者と利用者が違う場合に，効率的な計算機利用のために見落す事のできない重要な点である。

最後に計算機システムの運用面での効率の向上の問題になるが，高速印字機やカード出力の利用方法は利用者側で配慮しなければならない。これらへの過多な出力は，せっかく高速度の中央処理装置で処理された計算結果も，出力がこれに追つかないためにシステム内にジョブが留まり，他のジョブ処理への障害となり計算機の処理効率を落す。これは、利用者個人としては余り問題にはならないが，全処理量の増加やターン・アラウンド・タイムの短縮には必要な配慮である。

6．むすび

ここではフォートラン・プログラムを中心に，計算機利用における効率についてできる限り多面的に問題点を取りあげるように努めた。“計算機利用における効率”に，利用される計算機の側と利用者の側の２つの側面がある。また効率を理解してゆくうえで時間的側面からと空間的側面から見る事ができる。さらに効率向上のための工夫の方法に大局的側面と局所的側面が考えられる。そのうえ，計算機の利用は非常に多方面，多分野にわたっている。したがって計算機利用の効率について，限られた紙面で一般的に議論する事は非常に困難である。ここに試みたささやかな解説は，はなはだ偏狭な意見になっているかもしれない。また中には具体的内容のともなわない部分もある。これらは具体的経験をもつ利用者の意見を聞くのが最も望ましい事であろう。

このような理由から多くの利用者がこの誌面を利用されまして，「効率」の問題についての具体的アイディア・意見を出し合って討論し，より効率よい計算機の利用方法を明らかにする事を望んでいます。