EPS File kaisetulogo.eps.ps not found

ミニ特集　システム制御理論は如何にして創られたか

制御系 CAD の環境と実現

山本　裕*, 古賀　雅伸†, 藤岡　久也‡

1. はじめに

今日制御系 CAD にはどのような性能が要求されるのであろうか．それはどのような期待の下に，如何なる方向に向かって行くのか．本論をここから始めてみたい．

初期のころの制御系 CAD とは，制御系設計を行うのに必要な様々な数値計算を行うプログラムの域を大幅に出るものではなかった．それらは一つには当時の計算機の水準からの制約によるものであるし，また他方ではユーザインタフェースに対する関心がいまほど高くないためでもあった．

しかし今日では事情は全く異なっている．計算機の能力の向上と資源の拡大はあらゆる局面における計算機の使用を前提とすることを可能にした．つい 10 年前には大学で使用される計算機資源のほとんどが 16 ビットパソコンで，その上での CAD が主流であったのとまさに隔世の感がある．今日 CAD とは単なる設計支援数値計算ルーチンのことではなく，充実した計算機環境の下でのモデリングから設計までを多様にかつ統合的に支援する環境のことでなければならない．言い換えれば，計算機の中にモデルや設計のための多様な支援モジュールが組み込まれた上で，モデル化から設計までの仮想的な世界が実現され，現実のプラント上での実験をかなりの程度計算機上のそれに置き換えることが出来るものが指向され，望まれているのである．

そういうものを CAD の理想像とするならば，それを支えるべき計算機上の環境はどのような条件をみたすものでならないかを本論で考えてみたい．

なお，本解説はロバスト制御研究会 CAD グループ主催になる 1995 年の札幌での SICE 学術講演会でのラウンドテーブルディスカッション「アドバンスト制御系 CAD— 開発者によるパネル討論」での討論を基にしたものである．

*京都大学工学部
†東京工業大学 工学部
‡大阪大学 工学部

JL 0008/95/3408-0200 © 1995 SICE

2. 計算機環境とオブジェクト指向

上で述べた理想像はあくまで一つの理想であって，現在それが実現されているわけではない．いや，むしろ現在の制御系 CAD はまだその理想形から相当に隔たっていると言わねばならないであろう．

ここではそのような理想形を実現するために必要とされる条件について考えてみよう．

現在いくつかの商用ソフトウェアが存在し，それらは制御用 CAD 開発のための環境をある程度提供している．またある程度のユーザインタフェースも提供されており，実際にそれによって開発された様々な設計用ソフトウェアが販売されてもいる．しかしながら，これらのソフトウェアが提供している環境，プログラムはまだまだ理想からは遠く，ユーザにも開発者にもかなりの程度の負担を強いているのが現状と思われる．例えば指数関数 $\exp(t)$ などは，数学では変数が実数，複素数，行列などの区別なく，同じ記号で表わされ，適切に定義が拡張されて用いられるが，通常プログラムではなかなかそうは行かず引数の型をプログラマが識別することが必要になる．それだけならまだしも，型が異なると別の名前をもつ別の関数を呼び出す必要のあることがほとんどである．例えば指数関数の例で言うと

$$\text{実数} \rightarrow \text{複素数} \rightarrow \text{行列}$$

と言うように概念が拡張されているわけであるから，これらに対してデータのクラスも同じように拡張され，上位クラスで持っていた性質が継承され（オブジェクト指向概念におけるクラスの継承），かつこれらに対して一つの関数 exp が定義されていて，引数の種類に関わりなく

$$\exp(A)$$

と共通の呼び出し方が出来るのが望ましい（関数のオーバーローディング）．必要な型のチェックやエラーハンドリングはオブジェクトが持っている情報が関数に引き渡される時に行われればよく，ユーザやプログラマはそれに煩わされずに抽象的な世界でプログラムできることがより快適な開発環境を提供することになる．

これらはオブジェクト指向プログラミングの考え方であるが，制御の世界には意外に多様なオブジェクト（システムのタイプにしても連続系，離散系，サンプル値系，線形系，非線形系，ハイブリッド系などさまざま）があり，多様な演算も行うためこのような考え方は有効なのである．無論，現実にこれらを実現するにはよく設計されたデータタイプとその取扱い方，柔軟な（例えば必要に応じて拡張できるような）データ構造などが必要になり，必要な機能は多様なものとなる．なお，オブジェクト指向に基づいて制御系 CAD を構築した先駆的な研究として，smalltalk を用いた [3] や C++ を用いた [4] があり，また数式処理を用いた CAD の研究には [5] がある．

一方今日のソフトウェアは，かならずしも個人で開発されるものではなく，多くの人々による資産の共有が重要な要素となってきている．また一方では様々な特定要求に合わせて機能を拡張する必要もあろう．このためには安定し，拡張性に富みかつ移植性の高い言語仕様が望まれることになる．

以下では著者らが開発した MaTX, $\mathcal{H}_{\text{SYS}}$ モジュールなどのプログラムでの経験を基にこれらの問題について述べる．古賀担当の MaTX では主にプラットフォームとしての言語の側面を歴史的推移を踏まえながら述べ，藤岡担当の $\mathcal{H}_{\text{SYS}}$ モジュールの節では，プラットフォーム上で開発されたパッケージの開発における経験からこのような CAD 開発における問題点と次世代の言語仕様に要求されるものについて考える．ただ MaTX と $\mathcal{H}_{\text{SYS}}$ モジュールは開発目的も環境も異なり，そこにおける著者らの立場も必ずしも一様ではない．MaTX は言語仕様を提供するものであり，プログラミング言語で言う低レベル[1]処理を受け持つものであり，一方 $\mathcal{H}_{\text{SYS}}$ モジュールはそのようなプラットフォーム上で動作するアプリケーションの色彩を持っている．それらの立場の違いを反映して，ここでの議論もやや方向性の異なるものが出てきている．またユーザとして想定している層にも微妙な違いの生じている部分もある．本論では各々の開発者 (MaTX– 古賀, $\mathcal{H}_{\text{SYS}}$ モジュール –藤岡) の意見の個性の違いを無理にまとめることはせず，むしろ論じた内容をそのまま掲げることにした．開発者の立場からみた CAD に対する見

[1] 低級と言う意味ではなく，より要素技術的であると言うこと，例えば C は MATLAB に比べて低レベル処理ができると言う意味で低レベル言語である．一方構造体のように C で用意されていて，現在の MATLAB などにはないようなデータ型もあったりしてこの区別は一概にはいい難い．

解の例として参考になれば幸いである．

3. MaTX の開発とその仕様について

以下では古賀が開発した MaTX [6] という CAD プログラムの開発経緯における経験に基づいて CAD に要求される基本条件について述べる．

3.1 CAD が使われ続けるための条件

著者 (古賀) が MaTX に関するコードを書き始めたのは 1989 年，今から 7 年前のことである．当時，著者は東京工業大学大学院修士課程の 2 年生で，制御工学を専攻しており，古田勝久教授の研究室に所属していた．研究室では DPACS-F (Design Package for Control Systems developed by Furuta Laboratory) [7] という制御系 CAD プログラムパッケージが使われていた．この CAD パッケージは 1970 年代後半にミニコンピュータ (NOVA, MRDOS, 96KB メモリ，10MB ディスク) 上で開発されたものであり，状態空間表現，伝達関数表現，マルコフパラメータ表現，微分 (差分) 方程式表現，入出力データ表現のいずれかで記述される線形時不変システムの制御系の解析，設計を実行できた．DPACS-F は (コマンドラインインタプリタとデータハンドリング処理を行う)Supervisor と呼ばれるプログラムと各処理プログラム群からなり，ユーザはコマンドラインで入力を繰り返すことにより設計を行えた．このように優れた CAD パッケージを使える環境にいながら，なぜ MaTX を開発することになったのかを振り返ることによって制御系 CAD が使われ続けるための条件について考えてみたい．

3.1.1 拡張性のある CAD

DPACS-F を含め 1970 年代に開発された制御系 CAD パッケージで採用されたコマンドラインインタプリタを用いるユーザインタフェースは，メニューによりコマンドをユーザに選択させるユーザインタフェースと比較して任意にコマンドを組み合わせられる自由度をユーザに与えた点で優れていたが，使いやすさはコマンドセットの設計に依存していた．コマンドの機能がマクロ的になるとコマンドの組合せにより得られる自由度が減り，コマンドの機能が基本的になるとユーザに要求するコマンド入力数が増えるという問題があった．

基本的な機能をもつコマンドを組合せることでマクロ的な機能をもつコマンドを作り出す機能を CAD にもたせることがこの問題を解決する方法の 1 つである．(現在の多くの商用ソフトウェアは，この方法をコマンドスクリプトとして採用している．) 新しいコマンドを定義する術をユーザに与えていることが重要な点である．当時は

$$\text{制御系CADのユーザ(グループ)} = \text{制御系CADの開発者(グループ)}$$

であったため，コマンドを定義する機能をユーザに与える必要が無かったのであろう．

MaTX を開発することになった 1 つ目の理由は簡単に拡張できる CAD を使いたいという要求を満たすためであった．CAD の使われる環境は様々で，状況に応じて機能を拡張する必要があることから考えて，CAD を拡張する方法をユーザに提供することが使われ続ける CAD であるための条件の 1 つになると思われる．

3.1.2 移植性の高い CAD

C 言語が普及する以前，1970 年代および 1980 年代前半，多くの制御系 CAD は FORTRAN を用いて記述された．当時，開発者は計算機の能力の制約からシステム (ハードウェア，オペレーティングシステム，開発言語) に強く依存したプログラミングを強いられた．このため，その頃開発された制御系 CAD を開発環境 (またはシステム) と異なるシステムに移植することは大変な作業であった．FORTRAN の言語規格が数回にわたって変更されたことも FORTRAN で記述された CAD の移植性を悪くした原因と考えられる．

1980 年代末，多くの大学の研究室では 16 ビットパソコンが制御系の解析，設計，シミュレーション，さらに実験に使われていた．当時，DPACS-F は MV4000 という 32 ビット CPU をもつスーパーミニコンで，マルチユーザ環境を提供するオペレーティングシステム上で使用されていた．

1990 年代に入ると，32 ビット CPU を搭載した (性能はミニコンと同程度，価格はパソコンの数倍でミニコンの数分の 1 という) EWS(エンジニアリングワークステーション) が大学の研究室に導入され始めた．ワークステーションには，マルチタ

スク・マルチユーザ環境を提供し，ネットワークに対応したオペレーティングシステム (UNIX) が搭載されていた．UNIX 自身が C 言語で記述されていたこともあり，ワークステーション上でのプログラミングには C 言語が用いられた．また，効率の良いコードを生成する安価なコンパイラがパソコン上でも提供されるようになり，C 言語は広く普及していった．

ワークステーションの登場，そしてパソコンの高速化，低価格化が進む中，現在使用されているいくつかの商用ソフトウェアには FORTRAN から C 言語に書き直されたものもあるとは言え，上で述べた移植性の悪さから 1970 年代に開発された CAD を新しい計算機環境へ移植する作業は進まなかった．

MaTX を開発することになった 2 つめの理由は，安くて速い計算機で CAD を使いたいという要求を満たすためであった．このことから，移植性の高いソフトウェアであることが使われ続ける CAD であるためのもう一つの必須の条件になるであろう．

3.1.3 使われ続けるための条件

以上のことをまとめると，1 つの制御系 CAD が長い間使われ続けるためには，次の条件が浮かび上がる．

- 拡張性があること．
- 移植性が高いこと．

これは一般のソフトウェアについて，使われ続けるための条件として言われていることと一致する．

3.2 今後の制御系 CAD に要求される機能

次に，MaTX の開発の経験を基にして，今後の制御系 CAD に要求されるであろう機能について考えてみたい．

3.2.1 インタプリタとコンパイラ

制御系 CAD 開発のための環境を提供する多くの商用ソフトウェアはプログラムを実行するインタプリタを提供している．これは，ソースコードのコンパイル，リンクといった面倒な作業からユーザを解放するためである．また，インタプリタはソースコードを 1 行 1 行解釈，実行するので，その時点で必要な変数や関数の情報をもっており，変数や関数の宣言を必要としない．このため多くのユーザはインタプリタを歓迎している．

しかし，インタプリタは一旦ソースコードをなんらかの内部コードに変換してから実行するので，コンパイラによって生成される実行プログラムに比べて繰り返し計算が不得手である．繰り返し計算を必要とする処理の例としてシミュレーションや最適化計算などが挙げられる．どちらも本来時間のかかる処理なので，ユーザは高速化を求めている．

このようにユーザはプログラム開発および使用に際しての簡便さと計算処理の高速化を求めている．この 2 つの要求に答えるためには，インタプリタとコンパイラの両処理系をユーザに提供する必要があると考えられる．著者は MaTX の両処理系をユーザに提供し，

- 繰り返しを必要としない解析・設計にはインタプリタ
- シミュレーションのように繰り返しを必要とする処理にはコンパイラ

を使用するようユーザに勧めている．

今後，CAD の使用目的が多様化するにつれ，ユーザは制御系 CAD に対して簡便さと同時に高速性をますます要求するもと思われる．この要求に答える 1 つの方法がインタプリタとコンパイラを提供することであると思われる．

3.2.2 データ型定義機能

MaTX という名前がまだ無かった頃，MaTX には，2 つのデータ型，実数と実行列が存在していた．整数を実数で，複素数を 2 個の実数で代用していた．すぐに複素数と複素行列が追加され，行列を扱える電卓として使われた．次に，MaTX でボード線図やナイキスト線図を描くため，伝達関数を表現するため 1 変数の多項式，有理多項式，多項式行列，有理多項式行列が追加された．それ以前は，係数を代入した行列で多項式を，2 個の多項式で有理多項式を表現し，演算や係数の操作は専用の関数を用いて行った．4 つのデータ型の追加により，伝達関数などのデータの定義や操作

を自然に表現でき，論文や教科書に書かれた数学的表現をほとんどそのままプログラムとして記述できるようになった．そして，`if`，`for`，`while` などの制御構造が導入され，ループカウンタを表現するため整数型が，警告などのメッセージを表示するため文字列型が追加された．

その頃，行列の LU 分解，QR 分解，QZ 分解，SVD 分解など複数個の返り値をもつ関数が追加された．複数の返り値をもつ関数の実現方法として，C 言語のようにデータのポインタ (アドレス) を関数の引数として渡す方法もあるが，ユーザにとってプログラムが分かりにくくなる恐れがあり，ポインタの導入をやめた．

データ型の異なるデータをまとめて扱えるリスト型を追加し，複数の返り値をもつ関数はリスト型の関数として実装した．リスト型の導入により，状態空間表現によるシステムの表現などの抽象的なデータ表現が可能になった．その後，時系列データなど同じ操作が施される同じ型のデータをまとめて扱うため配列型が，行列や配列の区間を表現，保存するため指数型が導入された．配列型の導入により，行列の指数関数も (成分毎に演算を行う) 配列の指数関数も，実数や複素数の指数関数と同様に `exp(A)` で記述できるようになった．

現在，MaTX には，整数，実数，複素数，文字列，多項式，有理多項式，行列，配列，指数，リストのデータ型が用意されている．これらのデータ型が整えられた後 4 年間，新しいデータ型は追加されていない．この期間，必要になったデータ構造はリスト型を用いて実装し，演算や操作のために専用の関数を用意した．この方法の長所は処理系 (MaTX) 内部の変更を伴わないことであり，短所は (多項式を行列で表現する方法と同様に) データを扱う関数内でデータの型チェックが必要である，データの定義や操作が数学的表現と違うものとなるなどである．

以上，MaTX を開発した経験を基に，データ型を区別する重要性について考えてみた．1 つの (例えば行列) データ型を用意し，データ型を区別しないソフトウェアについて「ユーザがデータ型を意識しなくて良いので使いやすい」という評価もあるが，マニュアルに掲載されている似たような名前の関数群から自分が本当に必要とするものを捜し出さなければならない苦労，数学的表現で書かれたアルゴリズムと関数呼び出しの羅列を比べなければならない苦労のことを考えると，データ型を区別しない方が使いやすいとは言えないのではないだろうか．

制御理論が発展するにつれ，多くのデータ構造が必要になり，データ型を区別しないソフトウェアのデメリットの方が大きくなると思われる．そして，前章で議論した拡張性のある CAD であるためには，新しいデータ型を定義する術をユーザに与えることが重要になると思われる．

3.2.3　コントローラの実現作業の自動化

近年，CAD により設計されたコントローラをディジタルコントローラで実現する作業を自動化する試みが行われている．これは，制御系の設計から (ディジタルコントローラによる) 実現までに要する時間の短縮と CAD により設計されたコントローラと実現されたコントローラの一貫性を保つことを目指している．多くの制御系 CAD は便利で使いやすいシミュレーション環境を提供しており，シミュレーション環境で作成したコントローラを簡単にリアルタイム制御環境に移植できることが望まれている．すでにいくつかの方法が発表されている中で，たとえば，MATRIXx の AutoCode や Simulink の C-code 生成ツールボックスのようにソフトウェアパッケージに C 言語のコードを出力するモジュールを付加する方法などが提案されている．これらのツールには，効率の良いコードの生成，デバッグやデータ保存の方法，安価な開発 / 使用環境 (ハードウェアとソフトウェア) などが要求される．

著者は，シミュレーションプログラムとリアルタイム制御プログラムにできるだけ同じ構造をもたせ，最小限の変更でシミュレーションプログラムからリアルタイム制御プログラムを自動生成するソフトウェアの開発を進めている．このソフトウェアを用いると，シミュレーションと実験で同じデータが扱えるのでデバッグ作業が容易になる，2 つのプログラムが同じユーザインターフェースをもつので作業の繰り返しが簡単になる，などの効果がある．また，プログラムは制御系設計に用いるものと同じ言語で記述され，リアルタイム制御プログラムに解析・設計ルーチンを組み込めるので，コントローラの修正と実験の繰り返しをプログラムをコンパイルし直さずに実行できるという特徴がある．

制御理論がますます発達し，複雑で次数の高い

コントローラが使われるにつれ，コントローラを実現する作業を自動化したいという要求がますます高まるものと予想される．

4. $\mathcal{H}$SYS モジュールの環境

$\mathcal{H}$SYS モジュール [1] とは，サンプル値系，連続時間系，離散時間系の解析・設計を行うため，著者 (藤岡) らによって開発された，Xmath [2] 上で動作する関数パッケージである．サンプル値制御系に関する最新の理論を実装しただけでなく，連続時間系，離散時間系に対しても解析・設計を統一した書式で取り扱うことが出来る仕様になっている．一例を挙げると，$\mathcal{H}_\infty/\mathcal{H}_2$ 設計，周波数応答の計算，各種表示ルーチンなどが，システムの種類によらずまったく同じ書式で呼び出せ，計算させることが出来るようになっている．このような高度な仕様を実現するため，パッケージはかなり大規模化し，200 に及ぶ関数から構成されている．

パッケージの大規模化の傾向は $\mathcal{H}$SYS モジュールだけのものではなく，最近リリースされる制御系解析・設計パッケージは，100 以上の関数群から構成されることが珍しくない．この規模になると，パッケージ全体をうまくデザインしないと，開発者にとってはプログラミング，デバッグ，メンテナンスに絶望的な手間が必要となり，またユーザにとってはマニュアルと首っ引きでないと使いこなせない難解なものとなってしまう．そのための効率的なシステム設計を支えるのが基礎となるプログラミング言語であり，その機能は開発されるパッケージの実装に決定的な影響を及ぼす[2]．

本節では，制御系解析・設計アルゴリズムの記述を効率よく行うためにプログラミング言語が持つべき機能は何かについて，$\mathcal{H}$SYS モジュールの実装[3]を例に取り考えたい．これは同時にエンドユーザにとっての使いやすさを左右する問題となるものである．なお，制御系 CAD においてプログラミング言語の果たすべき役割は，解析・設計アルゴリズム記述だけに限らず，例えば GUI 記述などたくさんあるが，本節の著者 (藤岡) の知識・経験の範囲をこえるのでここでは取り上げない．

[2] あまりよい例ではないが，FORTRAN でグラフィックユーザインタフェース (GUI) を含む CAD のすべてを開発するような状況を考えてみると違いが分かるであろう．

[3] 以下で示す実装には，開発当時 ('93∼'94) の Xmath の機能に制約を受けた部分もあるが，そのうちのいくつかは最新のバージョンではより容易に実装できる．

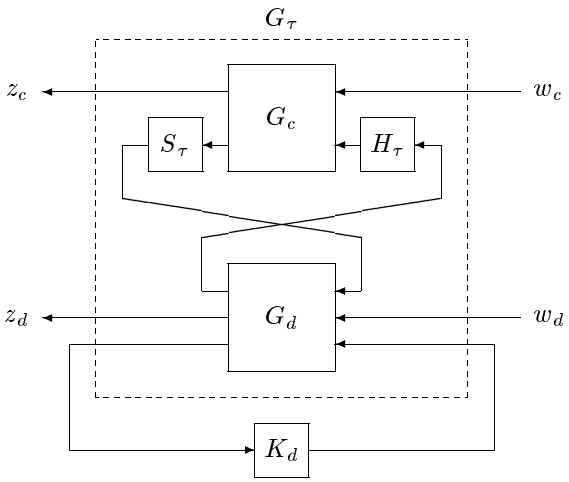

Fig. 1: Sampled-Data Feedback System $G_\tau \star K_d$

4.1 データ構造とデータ型

データ構造はアルゴリズムとともにプログラムを支える重要な基本要素である．適切なデータ構造を選択すれば，アルゴリズムを簡潔に記述できる．プログラミング言語は通常その目的に応じて，重要なデータ構造をあらかじめデータ型として用意している．古くは整数型や実数型などのごく基本的なデータ型だけが提供されているに過ぎなかったが，現在ではデータベースにおける住所レコードのように抽象的なデータを一まとまりのものとして取り扱えるようになっている．これを抽象化したのが抽象データ型であり，またオブジェクト指向におけるオブジェクトの概念にもつながる.

制御系解析・設計において必要なデータ構造は，整数，実数，文字列など一般的なものだけでなく，ざっと考えても行列，多項式，有理式，システムの状態空間表現など非常に多い．したがって，それらをデータ型として用いることができるプログラミング言語が望ましい．

$\mathcal{H}$SYS モジュールの設計方針とそのために選択したデータ構造の例を述べよう．$\mathcal{H}$SYS モジュールは，サンプル値制御系の解析・設計が可能であることを主な特徴としている．この際連続時間，離散時間システムをその特殊な場合として含むことも前提としている．$\mathcal{H}$SYS モジュールで取り

扱うことができるサンプル値制御系設計問題のブロック線図を図 1 に示す．一般化プラント G_τ は，連続時間部分 G_c, 離散時間部分 G_d, サンプラ S_τ およびホールド H_τ から構成される．与えられた一般化プラント G_τ に対して離散時間補償器 K_d を設計するのがサンプル値制御系の設計問題である．このようなサンプル値制御系を設計する関数の実装において，どのようなデータ構造を用いるべきであろうか．$\mathcal{H}$SYS モジュール では G_τ を表現するデータ構造を用意し，1 つの変数で表現すべきだという結論に達した．それにより書式の簡潔化が計れるばかりでなく，システムの型によらない統一的な記述が可能になる (4.2 節参照).

ただ，$\mathcal{H}$SYS モジュール の動作環境である Xmath [2] におけるプログラミング言語である MathScript には動的システム (状態空間・時不変) を表すデータ型 Dynamic System は用意されていたが，G_τ のためのデータ型は (当然ながら) 用意されていないので，G_τ のデータ構造 (仮想的なデータ型として hsys と呼ぶ) を別のデータ型を用いて表現することになった.

具体的な実装は次の 2 つのステップで行われている．まず G_c と G_d の入出力信号が分割されていることに注目して，Dynamic System と入出力信号の分割情報を合わせたデータ構造 (仮想的なデータ型として psys: $\mathrm{partitioned\ system}$ と呼ぶ) を，汎用のデータパッキングのための型である list 型の変数として実装した．次に，G_c と G_d を表す psys (実体は list 型の変数) とサンプル時間などを list 型の変数にパックし，hsys の実装とした (実体はやはり list 型の変数)．これらのデータ構造を作成するのに，Dynamic System 型や list 型を下部構造として使用した．もし，それらが用意されていなければ，psys や hsys の実装には膨大な手間が必要であったと思われる．

ここで用意したデータ構造は単純なデータパッキングでしかないが，それでもプログラムの可読性に大きく貢献するものである．標準的な $\mathcal{H}_\infty$ 補償器設計問題などで用いる一般化プラント G は Fig. 2 に示すような 4 ポートシステムである．4 ポートシステムは，$\mathcal{H}$SYS モジュール では psys として 1 つの変数で表現され，$\mathcal{H}_\infty$ 補償器設計は

```
K = hinf_syn(G)
```

と記述できる．もし G が入出力信号の分割を情報として持たないと，それを引数として指定

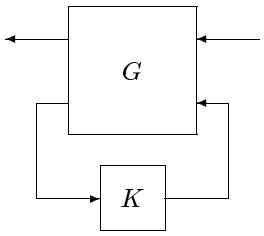

Fig. 2: 4-port System G

しなければならない．実際，そのように実装されたパッケージがいくつかあるが，引数の取り方がパッケージごとにばらばらでユーザの混乱を招きがちなのに対して，4 ポートシステムを 1 変数で表現した上の記述はユーザにとってわかりやすい自然なものであると言えよう．

今後望まれるのは，より豊富なデータ型と，ユーザによるデータ型定義機能であろう．まず，データ型は豊富であるほうが当然良さそうに見える．それは確かに正しいのであるが，単純に用意しただけではあまり意味がなく，少なくとも次の 3 点には注意が必要である．

1. あるデータ型の変数が持つ内部変数 (例えば G_τ における G_c) へのアクセスをきちんと管理しなければ，データの一貫性を保つことが難しい．これは，データ隠蔽とかカプセル化と呼ばれるが，システマティックに行うためには，データ型では概念的にもはや力不足で，オブジェクト指向におけるクラスの概念を導入する必要があるだろう.

2. 複雑なデータ型を用意しても，より単純なデータ型が必要なくなるわけではない．例えば，実数型が用意されていても，整数型が必要な状況はループ変数など少なくない．整数型，実数型のような低いレベルの型に関する問題の解決としては得策ではないかもしれないが，一般にはクラスの概念を導入して継承を用いて解決すべき問題であろう．

3. プログラミング言語の基本的な文法は，例えば C 言語など標準的なものにできるだけ近い方が良いが，それぞれのデータ型の変数の (アクセス可能な) 内部変数にアクセスする

ための文法はそれにふさわしいものを用意しなければならない. 例えば，行列やシステムのブロック構造を考慮したアクセスを表現する方法は重要であると思われる．

たとえこれらの点に注意して, できるだけ多くのデータ型をあらかじめ用意しても，制御理論が発展し続けるかぎり，必要となるデータ構造は増え続けるであろう. データ型のユーザ定義機能が望まれる理由がここにある．より具体的に, ユーザがデータ型を定義することには以下のような利点があると考えられる:

1. データ構造は必要に応じてデータ型として取り扱う方が，データ管理とアルゴリズム記述を分離しやすく，したがってメンテナンスが容易である．
2. 言語仕様を拡張し続けるのは互換性や安定性の面で好ましいことではない．
3. 逆接的に聞こえるかもしれないが，ユーザやプログラマにとって使いやすいデータ型を確立するために必要である．システムの状態空間表現のような，制御工学においてはごくごく基本的であるはずのデータ構造さえ，型としての実装は少ないのが現状である．データ型として標準的なものは，机の上での議論ではなく，プログラマによる実装，ユーザからプログラマへのフィードバックを繰り返す試行錯誤を経て決定されるべきである．

4.2 関数・オペレータのオーバーロード

$\mathcal{H}_{\mathrm{SYS}}$ モジュールは，連続時間系・離散時間系・サンプル値制御系をまったく同等に扱うために，対応する機能の関数には同じ名前を用い, 同じ文法で呼び出せるものとした. 例えば，前節で連続時間 $\mathcal{H}_\infty$ 設計問題の文法を示したが, 実は連続時間系，離散時間系，サンプル値系 (Fig. 1 参照) すべてに対して

```
K = hinf_syn(G)
```

と記述できるのである．これらの違いはデータ `G` が持っており，関数 `hinf_syn()` はその違いを自動識別して処理する. もちろん `K` の型は `G` の型に応じて決定される. データ (引数) の属性は引数自身が持っているのだから，それによって関数名を変えるのは冗長であり，また混乱の元となる．関数の名前の決め方に統一を取り使いやすいインタフェースを提供するためにも，データの属性で関数名を変えるべきではない．

$\mathcal{H}_{\mathrm{SYS}}$ モジュールでの実装は非常に単純で, 引数のデータ構造によって条件分岐を行った. メンテナンスの手間を犠牲にして，ユーザの使いやすさを優先した実装である.

上のように，まったく同じ名前で，引数の型や数が異なるだけの関数を定義する機能は (関数の) オーバーロードと呼ばれる．条件分岐で疑似的に実現するのではなくて，プログラミング言語レベルがオーバーロードをサポートしていれば，まったく独立した関数として定義でき, メンテナンスも容易である．これは，後で述べる型チェック，型変換の機能と同時に実現されるべきであろうと思われる．また, 関数のオーバーロードは名前空間があふれるのを防ぐはたらきもある．

オペレータ (演算子) もオーバーロード可能であれば (実際には，かつうまく定義すればという条件がつくが)，プログラムの可読性を飛躍的に高めることができる．$\mathcal{H}_{\mathrm{SYS}}$ モジュール ではパーザ[4] を自分で用意でもしない限りオペレータのオーバーロードは不可能 (当時) なので用いなかったが，実際にパーサを自前で用意したパッケージもリリースされている．自前でパーサを用意するのは (当面の現実的な解決策になったとしても) 本末転倒であるから，プログラミング言語でサポートすべきであろう．

また，オーバーロードとは直接関係ないかも知れないが，システムのフィードバック結合は制御工学において基本的な演算であり，オペレータを使って表現すべきではないかと思われる．ただし，上でも述べたようにオペレータはうまく定義しないと逆に可読性を悪くすることに注意しなければならない．

4.3 型チェック・型変換

データ型を導入したり，関数をオーバーロードしたりすると，変数の型をチェックしたり，型を変換する必要が生じる．これらは，地味に思えるか

[4] 構文解析を行なう一種の前処理プログラム．

もしれないが，実は引数の型を自動的にチェックしたり，必要なときに自動的に型変換する機能は非常に重要であり，実現が強く望まれる．実際 $\mathcal{H}$SYS モジュール では，独自のデータ構造や疑似的なオーバーロードを導入した代償とも言えるが，変数の型 (構造) チェックや型 (構造) 変換のためにかなりのコードを用意しなければならなかった．しかし，もしそれらをきちんと行わなければ，デバッグは困難となり，開発の効率を下げる原因となったであろう．

データ型はできるだけ厳密にチェックすべきであり，プログラムで記述するのではなく，プログラミング言語レベルで行うべきである．変数の型を意識しないことは, 楽なことのように思えるかもしれないが, 実は気付きにくいバグを生み出し, デバッグの手間を増やしていることに注意してほしい.

以上，$\mathcal{H}$SYS モジュール を開発した経験を通じて，データ型，関数・オペレータの多重定義，型チェック・型変換の重要性について考えてみた．商業ソフトウエアの最近のバージョン (例えば [2]) で，これらの機能が徐々に実現されて来ていることは喜ばしいことである．

本節の最後に $\mathcal{H}$SYS モジュールをプログラムする機会を与えてくださった原教授 (東工大), プログラミングについてゼロから教育してくださった近藤助教授 (茨城大) に感謝します.

5. あとがき

以上 MaTX と $\mathcal{H}$SYS モジュールを例として制御系 CAD の現状とその実現，望まれる機能などについて述べた．なお，残念ながらデータ標準化については当初予定していたにも関わらず触れることが出来なかった．これについては現在 IFAC/IEEE-CSS Working Group on guidelines for CACSD software と言うグループが討論して，その作業内容を “Command-language standard for CACSD software” と言う中間報告にまとめている．標準化の是非については様々な意見があろうが，将来はこのような委員会にも日本からの意見が反映されるようであって欲しいと願っている．

References

[1] S. Hara, Y. Yamamoto, H. Fujioka and M. Takeda: “$\mathcal{H}$SYS Module Manual,” 住商エレクトロニクス, 1995.

[2] “Xmath Basics,” Integrated Systems, Inc., 1996.

[3] 赤堀，原: “オブジェクト指向に基づく制御系の解析設計支援システム,” 計測自動制御学会論文集, vol. 24, no. 5, pp. 506–513, 1988.

[4] R. Kondo: “La++ User's Guide,” オンラインドキュメント, 1992.

[5] 斉藤，阿部，高橋: “計算機援用による制御工学 — 制御工学と数式処理の周辺,” システムと制御，vol. 29, no. 12, pp. 785–794, 1985.

[6] 古賀，古田: “数値処理と数式処理を融合した制御系 CAD 言語 MaTX,” 計測自動制御学会論文集, vol. 29, no. 10, pp. 1192–1198, 1993.

[7] 古田，梶原: “制御系のための CAD,” 計測と制御, vol. 18, no. 9, pp. 55–64, 1979.