付録 I

RTSS マニュアル・序論

A

はじめに

やや散文的な記述を許していただこうと思う。

1992 年 11 月に曽根研ワークステーションがディスクトラブルに見舞われたときのことが忘れられない。筆者は当時システム管理者だったから，貴重な 1 週間をシステムのリストアに費やさざるを得なかった。当時筆者はこのプログラム RTSS（当時はまだこの名前で呼ばれてはいなかったが）を用いたシミュレーションの仕事をしていたが，当然その仕事も滞る。その仕事の関係でお世話になっていた鉄道総研の田中　裕さんにそのいきさつをお話ししたところ，「我々が master で，ワークステーションが slave なはずなのだけれど，実際には slave に支配されてしまうことが多いですね」といったことをおっしゃった。それに対して，「シミュレーションだけでなく，文書の処理も TEX ですからワークステーション任せなのです」といったら，「それじゃあ，高木さんの全存在じゃないですか」。

ワークステーションに依存しなければならなかった理由は明確であった。当時曽根研にあったパソコンは V30 CPU という貧弱なものが主流で，メモリ空間が 640kBytes しかなかった。このプログラムが行っているような饋電等価回路演算は，大きな行列の演算を必要としたが，それには能力不足が明確だった。これらの制約から当初メインフレームで行われていたプログラム開発が，ワークステーション導入に伴いこちらに移行したのはきわめて自然な成行きだったと思う。そして，ワークステーションで計算を行うのなら，結果の処理や文書の管理などもそこで行うのがやりやすいということになった。

結局，RTSS が走りやすい環境に，他のすべてを合わせていった，というのが実感だ。そういう意味で，もしワークステーションが筆者の研究室における全存在なのであれば，その本当の意味は「RTSS が全存在」ということなのである。

じっさい，RTSS は曽根研究室における筆者の全存在と呼ぶにふさわしい。卒業論文[37]・修士論文[38]はいずれもこのプログラムによって出された結果を用いて書かれている。筆者のこの何年間だかの研究のすべての成果やノウハウがここに凝縮しているというのは驚くべきことだ。逆にいうと，このプログラム程度のものが全存在ということは，筆者の力なんてちっぽけなものだ，ということをも痛感させられる。あるいは，人間一人がなしうる貢献というのは，この程度のものなのかもしれない。

いま，せめて願うことは，このコードがいろいろな場所で生き，活用され続けることだ。そのために，このマニュアルがすこしでも役に立てばよいと考えている。

〈A.1〉 沿革

このプログラムは，標準的な直流饋電システムのシミュレーションを汎用的に行えるシミュレーションプログラムとして，東京大学工学部電気工学科曽根研究室において，**曽根　悟教授**のご指導のもと筆者が開発・維持しているものである。筆者が卒論生として曽根研にお世話になるようになった 1989 年から開

発を始め，現在に至るまで継続的に改良が続けられている。

⟨A.1.1⟩ ver.1.0 — FORTRAN プログラム

NEC の PC-9801 が日本で手に入る唯一の「マトモなパソコン」だった時代から一転して，IBM PC 系の殴り込みなどから最近のパソコンの性能向上は目を見張るものがあるが，このプログラムを開発した当時はパソコンで鑑電特性の計算をするためには（主に MS-DOS の制約から）相当な工夫をしなければならないことが明白だった。このため，開発当初の 1989 年 10 月より 1990 年 12 月にかけては，このプログラムは東京大学工学部オンライン計算機センターの M880 計算機上で FORTRAN 77 言語によって記述されたプログラムとして動いていた。なお，行列演算のサブルーチンとしては数値演算ライブラリ MSL2 を用いていた。この FORTRAN プログラムは，当研究室における筆者の卒業研究[37, 52]のためのツールとして開発され，卒業研究に引続き筆者の修士課程での研究の初期の段階においても活用された[53][45]。当時はその名はなかったものの，このプログラムが RTSS ver.1.0 と呼ぶべきものだと考えている。

このプログラム（当時は `rt` = Ryo Takagi が作った `kiden` プログラムということで `kidenrt` などと呼んでいた）の基本部分には，過去に曽根研究室で開発されたさまざまなシミュレーションプログラムの経験が反映されている。しかし，**橋本 樹明博士**（現・文部省宇宙科学研究所助手，当時曽根研究室の博士課程の学生だった）をはじめとする研究室の諸先輩から，松富[1]などが残しているプログラムにはバグがあり，「そのコードを見てバグを発見して直すよりは自分で書いた方が早いだろう」と勧められたため，従来のプログラムのコードは全く参照せずに自分ですべての部分を書き下ろした。

この ver.1.0 の段階で導入された新しい機能の最たるものが，駅間走行時分一定化シミュレーション技術のインプリメンテーションである。このシミュレーションプログラムが他に例を見ないものとして評価されるゆえんでもある。この駅間走行時分一定化のソースコードは，現在のプログラムにもかなりの部分がほぼそのままの形で反映されており，RTSS の「いちばん古い部分」でもある。この他の部分は，例えば等価回路演算のアルゴリズムには松富論文[1]とか電気学会技術報告[19]などで紹介されるコンベンショナルなものが用いられている。ただし，等価回路計算に時間がかかることを見越し，回路の演算頻度を列車運動シミュレーションの 4 分の 1 に減らす手法を採り入れ，演算時間の短縮化をはかっている。

⟨A.1.2⟩ ver.2.0 の開発

1990 年度はじめに当研究室にもワークステーションが導入され，本プログラムも計算に課金のかかる M880 計算機から移植することとなった。当初は C 言語[33]での書換えを考えていたが，**入内島 健博士**（現・(株) 東芝）および**八杉 昌宏博士**（現・東大理学部情報科学科）という二人の先輩から強力に C++ を勧められたため，最終的には C++言語[32, 34, 35, 36]へと移植することになった。

お二人のお勧めの中には，自動的に FORTRAN 言語を C 言語に「翻訳」する `f2c` なるコマンドの利用もあった。しかし，こちらは「人間に可読なコード」を生成するコマンドではあり得ず，むしろ C コンパイラしかない環境のもとで FORTRAN を利用可能にするためのツールと考えるべきだったようだ。従って，移植はすべて人間の手による翻訳で行った。当然ながら C++ の特徴である「クラス」機能を利用するためにデータ構造などは大幅な変更を行っている。

1991 年 5 月には C++ 言語によるプログラムとして全面的な書換えが完了し，ワークステーション上で動くユーティリティとして動き始めた。その後，新しいデータ・関数・機能の付加，修士論文[38]のための研究の進捗，常磐新線でのシミュレーション・プロジェクトへの本シミュレータの適用[71, 72]などに伴い，さまざまな改造を実施した。

ver.2.0 は，処理系の変更と同時に相当大がかりな書換えを行ったため，ver.1.0 のコードが受け継がれている部分の方がはるかに少ないのだが，例外的に駅間走行時分一定化の部分だけは ver.1.0 のコードがほぼそのままのかたちで受け継がれている。ただし，ver.1.0 では別に用意した近似計算ルーチンで計算していたため誤差が大きかった。これを，通常の列車運動の計算に用いている関数による計算ルーチンを加え，精度をあげた。また，データ形式の大幅な変更により，さまざまな形態の路線のシミュレーション

に応用できる汎用性を備えることができた。速度制限を取り扱うこともできるようになった。また，等価回路演算の高速化のため，変電所および列車の電気的特性を媒介変数表示するという tricky な方法を用いて，Newton-Raphson 法を応用した計算スピードアップをはかった。このかわりに回路計算の頻度を下げるのはやめてしまった。

この他，列車の主回路電力制御によるピークカット[38][39]のシミュレーションのため，フルノッチ比なる概念を導入した。この概念の導入は，研究上のメリットもさることながら，さまざまなコードをわかりやすく記述できるというプログラム上のメリットも同時にもたらした。

(A.1.3) ver.2.1 および ver.2.2 の開発

その後，筆者はそのまま博士課程に進学したので，RTSS もさらに開発が続けられている。

まず，1993 年から (社) 日本鉄道電気技術協会に委員会が設けられ，そこで RTSS を再びシミュレーション[73][74] に用いることになった。

そのためにかなり大規模な改良が行われた。RTSS のコードのかなりの部分は実は構造体型データの配列を管理する部分になっているが，構造体の種類がいろいろであるためその都度同じようなコードが書かれていた。ver.2.1 ではこれらの配列コードの整理が大規模に行われ，ほとんどの配列が `template` なる機能[36]を用いて書き直された。この `template` 機能は 1990 年になって ANSI 標準に加えられた新しい機能であり，gcc 2.3.X あたりでは安定して動いていなかった。最近の gcc（2.6.0 を現段階では利用中）ではほぼバグもとれて安定しており，安心して使えるようになった。このテンプレート機能によって減ったコードは全体で 200〜300 行くらいに及ぶものと思うが，機能追加のためにコードはそれ以上に増殖してしまった。最近，RTSS のソースファイルの総行数は 12000 行を越えているようだ。

このプロジェクトにおいては，実測定とシミュレーションとの整合性が問題になっており，実測定データとシミュレーション結果との比較を行っている。この比較には RTSS の他にメーカのシミュレーションプログラムも参加した。幸いにしてどの比較でも悪くない一致をみたといえる。

このプロジェクトに参加したため，RTSS には実際のシステムに合わせるための「さまざまな，細かい改良」が目だっている。シミュレーションの命といえる等価回路演算ルーチンは，このバージョンで大きなバグがとれ，安定性も飛躍的に向上している。

シミュレーションプログラムの結果と実測値を，あるいはシミュレーションプログラムの結果同士を突き合わせるというのは，シミュレーションをする立場からすれば大変な仕事だ。このプロジェクトでお会いしたメーカ側の方々も大変苦労されたようで，メーカ側の研究員（女性）が「どんどん歳をとってしまう感じがしますね」といっていたのが思い出される。同業者としてそういう神経のすり減るような気持ちはよくわかるが，これがシミュレーションという研究手法の一つの宿命なのかも知れない。

1994 年 8 月まで続いたこのプロジェクトでは，さらに進んで変電所の送出電圧をリアルタイム制御する場合の検討も行った。それにも RTSS を利用することになったのだが，これは運輸省・交通安全公害研究所の主任研究官，**水間　毅博士**がおっしゃったひとことによって筆者の仕事となったものである。十分なアルゴリズム上の検討を行うことはできなかったが，いちおうデモンストレーションのためには十分と思われる結果が出た。そこで，報告書にリアルタイム制御のための 1 章を設けて記述を行うことができた。

(A.1.4) それ以降の開発，および総合シミュレータ化

筆者の博士論文のプロジェクトでは最適化シミュレーションルーチンを組み込んだ RTSS ver.3 の開発にも着手した。ところが，これがなかなかに険しい道であった。まず，最適化手法のためのプログラミングに手間取った。最適化手法としては Sequential Gradient-Restoration Algorithm の一種に分類される SCGRA[29]を用いているが，すでに完成しているものから SCGRA に関係するコードだけを拾い出してみると，すでに 2000 行を越えている。しかも，問題の特殊性（パラメータなし，終端条件固定など）を最大限利用した結果としてのコードなので，先行きが思いやられる。

そのような大規模なプログラムであることに加え，関係する変数が非線形なだけでなく，微係数まで含めて不連続であるようだと解けないらしく,「丸め」る操作が必要らしい。特に，フルノッチ比と電流（評価関数……エネルギーに密接に関係する）との関係は特に考慮しておかなければならない点の一つだということもわかっている。

一方，数値的最適化を行わないプログラムについては，順次いろいろな機能の追加が進み，現在 RTSS の最新バージョン番号は 2.4.0 となっているはずである。

なお，1994 年度から 2 年間,「鉄道総合シミュレータ」とかいう題目で，シミュレーションプログラムそのものを作る研究に科学研究費による助成が出た。この関係で，将来的に総合シミュレータとして必要なダイヤ評価などのコードも取り込んでゆく可能性がある。

(A.1.5) RTSS というプログラム名の由来

RTSS とは

Railway Total System Simulator

の略であるが，

Ryo TAKAGI and Satoru SONE

の略，すなわちコードを書いた高木　亮と指導教官の曽根　悟教授の頭文字の略，という 2 つの意味を持たせている。この名前が初めて世に出たのは文献 [60] でだった。その後かなりいろいろな場所で，この名前でプログラムを紹介している。

ところで，英語名のなかにある Total という言葉には，このプログラムの目指す方向が示されている。つまり鉄道システム全体のシミュレーションに使える汎用プログラムを目指そう，という意思表示でもある。

(A.2) C++言語

本プログラムは，いわゆる「オブジェクト指向言語」の一つである C++を用いて記述されている。

C++言語は C 言語にオブジェクト指向プログラミングのための仕様を追加した，C 言語のスーパーセットである。オブジェクト指向プログラミングの狙いは，大規模なプログラムを「楽で間違いを少なく」開発できる，というのが基本的な狙いだろう。しかし,「楽で間違いが少ない」という説は少々疑いたくなることも多い。オブジェクトをきちんと設計できなければ，かえってオブジェクトという存在が障害になることも多い。オブジェクトの設計いかんにかかってくるから，新しいルーチンを書こうとするとオブジェクトの設計で時間の大半が過ぎてしまうこともしばしばある。

C++言語自体にも，若い言語ゆえの問題点が少なくはない。例えば，C++の言語仕様は急速に標準化が進んでいるとはいえまだ十分ではない。なにより，現在の問題点は C++ の言語仕様が年を追うごとに拡張されていることだ。そのため，C++ の本は分厚くて読みにくいものになってしまっている。幸い，コンパイルに利用している GNU C++[32]が，文献 [36] で紹介されている標準化案にある言語仕様のほとんどをサポートするようになっているようなので，UNIX システムで利用する限りにおいては心配は少ない。

UNIX では世界的に標準言語の座を占めている C 言語のスーパーセットとして考えられている C++言語は，オブジェクト指向言語の中ではもっとも明るい将来を持っている言語でもある。オブジェクト指向言語としては，C 言語との互換性を考えたため不完全な部分が多いともいわれるが，そのような背景から比較的長い将来にわたって比較的ポピュラーな言語であってくれる可能性が大きい。

ただ，C や C++を FORTRAN に比べたとき，科学技術計算に使用するに当たっての弱点の一つが「ライブラリ資源の蓄積がまだ小さいこと」であろう。C 言語に比べると，C++言語では複素数クラスがすで

にライブラリの形で与えられているし，高水準入出力制御法としての「ストリーマ」なども同様に与えられている。しかし，このプログラムではこの「ストリーマ」機能は使わなかった。もったいないとは思ったが，C の標準入出力関数で書くのがどうもやりやすい感じに思えたのだ。じっさいデータファイルの出力などの局面では，いちいちストリーマクラスを使う必然性は感じられない。それに，例えば行列演算のようなものにしても，FORTRAN では既存のものが使用できたのに，C++では自作しなければならなかった。

⟨A.3⟩ 仕様

主な仕様を示す。

処理系 C++ 言語にて記述。GNU C++ Compiler (g++), GNU C++ Class Library (libg++) を使用してコンパイル。現在は GNU C++ 2.6.0 を利用している。SunOS 4.1.3 の走る Sun SPARC Station 10 および SPARC Station 5, 2, ELC, SLC で計算を行なう。なお，g++ の走るマシンであれば，ソースコードを移植すれば走るはずであるし，SPARC チップを CPU に用いたワークステーションならばバイナリの実行形式ファイルを移植するだけで動作するだろう。

列車数，特性 メモリの許容範囲内でいくつでも可能。特性はインバータ制御電気車および 4 象限チョッパ制御電気車のみに対応している。現在のところ，全列車の特性は同一である必要がある。

駅数，列車ダイヤ 駅数はメモリの許容範囲内でいくつでも可能である。次駅までの線路条件，走行時分，停車時分などに関するデータはすべて「次駅データ」にまとめられる。これをダイヤ 1 周期分だけ集めたものをダイヤパターンデータと称し，メモリの許容範囲内でいくつでもこのダイヤパターンを持つことが可能。それぞれのダイヤパターンにそって列車は等間隔配置される。

変電所数，特性 メモリの許容範囲内でいくつでも可能。折れ線 V-I 特性を持つ変電所ならばどんな特性のものもシミュレートできる。変電所毎に特性を与えることが可能。変電所母線から饋電線接続点までのインピーダンスを与えることができる。

饋電線数，形態，饋電定数 メモリの許容範囲内でいくつでも可能。形態は，「線分（端点あり）」「円（端点なし）」の 2 種類が選べる。これらを組み合わせることによって，あらゆる形態の路線がシミュレートできる。饋電定数は全線一定。原則として，全線並列饋電，変電所 Busbar で上下タイを行なうことを前提としているが，データの与え方で相当いろいろなことができるはずである。

距離・位置の表現法 饋電線ごとに距離原点をセットできる。例えば山手線と京浜東北線を同時にシミュレートするような場合は，山手線の 2 本の饋電線は「円形」で東京が原点，京浜東北線の 2 本は「線分」で大宮原点，などと表現が可能。変電所位置は饋電線接続点の位置を饋電線ごとに与える。従って，上の例に関していえば東京にある同じ変電所でも山手線と京浜東北線では位置が違う数字になる。列車は「どの饋電線の下を走るか」を決めて，その饋電線の座標で位置を表現。

付録 II

RTSS マニュアル・プログラムの概要

B

シミュレーションの方法の概要

直流饋電システムのシミュレーションプログラム（饋電特性シミュレータ）は，従来の曽根研における研究でも作成されている[1, 2, 5, 14]。曽根研以外でも，新たにシミュレータを開発したという報告がある[15, 18]。本研究において開発したシミュレーションプログラムもこれら既存のシミュレータも，ごく基本的でおおまかな考え方および構造については共通である。

〈B.1〉 プログラムのおおまかな構造

直流饋電回路に流れる電流を求めるためには[19]，

1. 所定のダイヤに従って列車群の配置や，速度・状態を求める。
2. 饋電用変電所・電車線路からなる饋電系統の対応する位置に，上記の電気車を配置して等価回路を作成する。
3. この等価回路を解いて，電圧・電流分布および各種電力を求める。

という手順をとればよい。これは，ある瞬間における饋電回路の状態を解析したものである。

しかし，一般には饋電特性とは「饋電系統内に点存する多数の電気車が連続走行するときの饋電系統の電圧・電流特性，電力特性，エネルギー特性，電気車の走行特性などの諸特性を総称したもの」である。特にエネルギーは電力を時間積分しないと求まらない。従って，饋電特性シミュレーションプログラムは，上記の手順1～3を連続的に，Δt 時間間隔ごとに，繰り返し行う必要がある。

そこで，饋電特性シミュレーションプログラムは必然的に，図5.1（21ページ）のような構造を持つことになる。つまり，プログラムは大きく分けて「列車を動かす部分」（列車運動シミュレーション部）と「饋電等価回路演算」部分とにわかれるのである。

〈B.2〉 列車運動シミュレーション部と饋電等価回路演算部との関係

これらの二つの部分は，以上に述べたようにかなり独立した機能を持っている。しかし一つのシミュレーションプログラムの中で動作している場合，完全に独立ではなく，次のように相互にデータをやりとりすることになる。

まず，饋電等価回路演算部は当然ながら列車の位置・速度・状態を列車運動シミュレーション部から受け取らなければ計算ができないはずである。このうち，列車の状態については後に詳しく説明するが，ここでは「力行（加速のこと）中」「惰行中」「ブレーキ中」などといったことがらを列車の状態と呼んでいると理解していただければ十分であろう。

一方，列車の性能は電車線電圧に強く依存することが知られている。このことから，列車運動シミュレーション部も饋電等価回路演算部から列車のパンタ点電圧のフィードバックを受けないと正しい計算ができないことがわかる。このことは，饋電回路の条件を変えると列車の速度が変化することがあることを意味する。

C

クラス・オブジェクトの大まかな構成

この章ではクラス・オブジェクトの概要を述べる。

⟨C.1⟩ 配列クラステンプレート table その他

C++ のテンプレート機能（パラメトライズド・クラス）を使った配列クラスである。なにぶん新しい言語のため，このテンプレート機能のきちんとした説明が載っている参考書がなかなかなく，苦労した（もっとも，コンパイレーションに使用している GNU C++ コンパイラも，バージョンが 2.3.X だったころはテンプレート機能がうまく働いていなかったようだから，よかったのかもしれない）。

パラメトライズド・クラス（クラス・テンプレート）は，型宣言の際に「引数（パラメータ）」としてクラスの型名とか整数のデータなど，いくつかのパラメータを渡すことのできる形式である。実際にコードを書いてみると，int 型の配列であっても double 型の配列であってもほとんど同じ操作になることが多いのだが，肝心の「型」の名前が違うというだけで異なるコードを書かなければならない。このことを奇麗に解決する手法として，1988 年に初めて提案され，1990 年 7 月に ANSI C++ 標準化委員会に採択された新しい機能である[36]。

RTSS で利用するテンプレートクラスとしては，table，reftable，zerotable，zeroreftable の 4 つが定義されている。細かな違いがあるものの基本はどれも同一であって，例えば，構造体 foo の配列を作りたい場合

```
table<foo> x;
```

のように宣言すればそれでよい。例えば，この配列の中身を参照する場合，通常の C 言語でポインタに対してするように x[3] のようにブラケット演算子が使える。ここで，配列として実際に確保した範囲を越える領域をアクセスしようとすると，通常の C なら Segmentation Fault を起こすところだが，それも回避され，エラーとして表示され，プログラムの実行が止まるから安全である。このテンプレートは，以下に述べるさまざまな配列を含むクラスの基底クラスとして用いられている。

⟨C.2⟩ 行列演算クラス matrix

行列演算を行うクラスである。自作であるが、逆行列、行列の 4 分割など必要な機能を一応備えている。

⟨C.3⟩ データリードクラス readdata

データリードを行うためのクラスである。# から行末までをコメント行として扱うルーチン、実数や複素数を読み出すルーチンなど、基本的な関数がいくつか含まれている。

RTSS は C++ のせっかくの「ストリーム・クラス」を使っていないため、入出力部分はいささかエレガントでない。そのうえ、コマンド名の検索に異常な行数を使ってしまっている。このあたりの改善は，今後の課題だろう。

⟨C.4⟩ 変電所クラス elecchar

変電所の特性を表すためのオブジェクトが elecchar である。電気的な特性に関する種々のデータを取り扱う。変電所の電圧—電流特性を模擬するのがこのオブジェクト主要な機能である。この機能については後に詳しく述べることにする。

なお、この elecchar という名称は，このクラスが列車・変電所共通の基本クラスとして，饋電等価回路から見た電気的な特性を模擬するためのオブジェクトとなることから名付けられている。

⟨C.5⟩ 列車クラス train

列車の特性を表すためのオブジェクトが train である。上記の elecchar と同様，電気的な特性に関する種々のデータを取り扱う。それに加え，列車の運動に関するデータを取り扱っている。

変電所は複数の饋電線に同時に接続（母線を通じて饋電線同士のタイを行うことに対応している）できることおよび位置が不変であることを除けば，等価回路上列車との区別がない。このことを利用するため，列車クラスは変電所クラスから派生させることとし，電気的な特性を示す関数について列車クラスと変電所クラスでまったく同様のインタフェースを持たせた。そして，変電所クラスからの拡張部分に上記の列車運動シミュレーション部分を入れることにした。この様子を図 C.1に示そう。

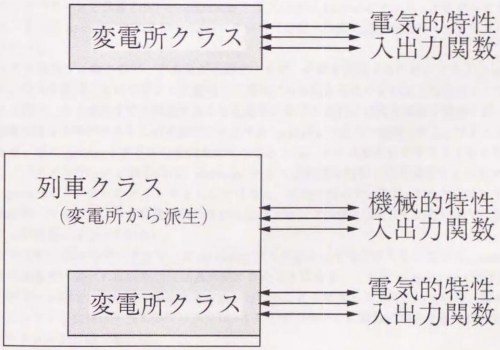
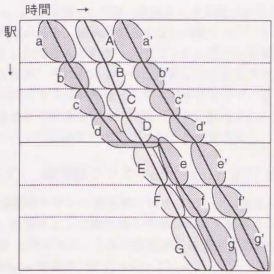


図 C.1: 変電所クラスおよび列車クラス



- 次駅データ:
  A, B, C, ..., a, b, c, ...
- 急行列車のダイヤパターン:
  { A, B, C, D, E, F, G }
- 待避を行う緩行列車のダイヤパターン:
  { a, b, c, d, e, f, g }
- 待避を行わない緩行列車のダイヤパターン:
  { a', b', c', d', e', f', g' }

図 C.2: 次駅データ配列の概念

いっぽう，現在のところ信号保安設備関係はまったくインプリメントしていない。このことから，列車の走行は各列車が互いに完全に独立にシミュレート可能である。このため，列車の走行に関するルーチンは列車クラスの中にすべて封じ込めることが可能である。この列車走行に関するデータは後に述べる nextsta クラスとして保存し，これへのポインタを列車データに持たせることで対処している。

これも，詳しいことは後述する。

⟨C.6⟩ 列車ダイヤパターンクラス diapattern，および次駅データクラス nextsta

列車ダイヤのモデルは，列車の路線全体にわたる走らせ方を規定するものであり，diapattern クラスが一つのダイヤパターンを管理する。diapattern クラスは実は nextsta クラス（次駅データクラス）の配列である。さらに nextsta クラスは gradcrv クラス（勾配・曲線・速度制限データクラス）の派生クラスとなっている。

列車ダイヤを表現する最小のデータ単位が次駅データで，それを表現するためのクラスが nextsta クラスである。全体を駅間ごとに切りとって管理し，列車がある駅からその次の駅（場合によっては駅間に特定の地点を設け，その地点までと限定することもある）までの走行に必要な情報を格納する。

勾配・曲線情報などのデータもこの次駅データクラス nextsta によって管理される。ファイルリード時には独立させ，別に gradcrv クラスの配列をつくることとし，データを記述するファイルも別にしてある。しかし，ファイルリードが終われば，nextsta クラスは必要な勾配・曲線情報をも自分で保持する（該当する gradcrv クラスへのポインタとしてではなく，自分で持っている）。この方法はメモリ容量をいささか無駄使いするが，メモリ容量制限の多かったパソコンでの使用は考えていないので，現在までのところ目だった問題にはなっていない。

次駅までの走行に関するデータがすべて nextsta クラスによって管理されるのだから，nextsta オブジェクトを必要数だけ並べれば列車の行路表が描かれたことになる。これが diapattern クラスである。このダイヤパターンの上に等時隔で複数列車を張り付けることもできるようになっている。高木が今までにシミュレーションしたほぼすべてのケースでは，1 ダイヤパターンに複数の列車を張り付けるこの方法でシミュレーションが可能であった。

このことをもう少し詳しく見てみよう。図 C.2 は次駅データ配列，すなわち diapattern クラスの概念的な図を示す。A, B, C, ... は急行列車のダイヤパターンであり，緩行列車のそれは退避を行うものが a, b, c, ...，行わないものが a', b', c', ... となっている。当然ながら同一の駅間を扱う次駅データである A

と a, a' には同一の勾配・曲線データが書かれる（多くの場合速度制限データも同じだろう）。さらに，緩行列車の場合，退避駅を含まない次駅データ，すなわち a ～ c, e ～ g と a' ～ c', e' ～ g' と（d と d' とを除く）はほとんどのケースでまったく同一になる。しかし，A ～ G の 1 組，a ～ g の 1 組，そして a' ～ g' の 1 組が，それぞれ 1 つの行路表に対応するので，同一でもデータは持たなければならない。

列車クラスは初期値としてどれかのダイヤパターン内の次駅データへのポインタを持ち，これを用いて列車走行の計算を行う。次駅データに記述された走行区間が終了したら，列車クラスは現在の次駅データの「次」の次駅データにポインタを切替える。

⟨C.7⟩ 饋電線クラス feedline および Y 行列作成クラス feed_y

饋電線クラスは饋電線に関するデータ（例えば饋電定数……長さ当たりの饋電線の抵抗），および列車・変電所とこの饋電線との接続関係を記述する。この接続関係を記述するためのデータは feedpos 構造体 と呼ばれる。1 本の饋電線には通常複数の変電所並びに列車が接続するから，feedline クラスは feedpos 構造体の配列を持っている。ただし，変電所は位置不変だが列車は位置も数も変わるので，これらを分ける必要がある。従って，他と同様な単純な配列というわけにゆかなかったため，このクラスは table クラステンプレートを使っていない。

直流電気鉄道の饋電システムを詳細に見ると，図 3.1・3.2（10ページ）のようになっている。これからわかるように 1 本の饋電線というのは変電所と変電所を結ぶものである。また饋電線と電車線（架線）とは別になっており，ざっと 200～300 メートル間隔で「饋電分岐線」によって結ばれている。プログラム，および等価回路上ではこれらのことは無視し，饋電線と電車線は一体であるとして計算を行うほか，饋電線に 3 つ以上の変電所が取り付くことも可能にする。また，饋電線と饋電線はすべて変電所の母線で接続されることを利用してモデルを作る。帰線としてレールを使っているが，レールの抵抗分も饋電線の抵抗に合算し，変電所・列車とも地上側はすべて接地するように考える。さらに，列車の長さは考慮しない。電圧・電流は定常解だけを求めるようにし，過渡解析および高調波解析は行わない。こうして得られる等価回路は図 5.2（22ページ）に示すようになる。列車・変電所以外はすべて抵抗だけのネットワークである。この簡略化によって饋電等価回路の演算はだいぶ容易になる。

さて，この饋電等価回路から Y 行列を作る必要がある。列車・変電所をのぞいた回路網のブランチについて，電流および電圧ベクトルをそれぞれ $\boldsymbol{I}_B$, $\boldsymbol{V}_B$ としよう。図 5.2の場合は 5 つのブランチ（図中 R_1 ～R_5）が存在する。ここで，これら 5 つのブランチのアドミタンスを対角成分に持つ行列 Y_B を定義しよう。図 5.2の場合

$$Y_B = \begin{pmatrix} 1/R_1 & & & & 0 \\ & 1/R_2 & & & \\ & & 1/R_3 & & \\ & & & 1/R_4 & \\ 0 & & & & 1/R_5 \end{pmatrix} \tag{C.1}$$

となる。このとき、

$$\boldsymbol{I}_B = Y_B \cdot \boldsymbol{V}_B \tag{C.2}$$

となる。

ここで，節点–枝接続行列 H なるものを導入しよう。節点–枝接続行列とは，ノードとブランチの接続関係を表す行列であり，ブランチ k はノード i が始点でノード j が終点であるとき，その (i, k) 要素は 1,

その (j,k) 要素は -1 となる。1，-1 以外の要素はすべて 0 である。H は，例えば図5.2の場合には

$$H = \begin{pmatrix} 1 & 0 & 0 & 1 & 0 \\ 0 & -1 & 1 & 0 & -1 \\ 0 & 0 & 0 & -1 & 1 \\ -1 & 1 & 0 & 0 & 0 \\ 0 & 0 & -1 & 0 & 0 \end{pmatrix} \tag{C.3}$$

のように書ける。

これを利用すると，すべての列車・変電所の集合はこの回路網のカットセットになるから，列車・変電所の電流・電圧ベクトルをそれぞれ $\boldsymbol{I}$，$\boldsymbol{V}$ とすると

$$\boldsymbol{I} = H \cdot \boldsymbol{I}_B \tag{C.4}$$

$$\boldsymbol{V}_B = H^T \cdot \boldsymbol{V} \tag{C.5}$$

となる。

式(C.2)・(C.4)・(C.5) より $\boldsymbol{V}_B, \boldsymbol{I}_B$ を消去すれば，

$$\boldsymbol{I} = H \cdot Y_B \cdot H^T \cdot \boldsymbol{V} \tag{C.6}$$

$H \cdot Y_B \cdot H^T \equiv Y$ とすれば、

$$\boldsymbol{I} - Y \cdot \boldsymbol{V} = 0 \tag{C.7}$$

という式が導かれる。この式，および Y 行列が回路計算の基本となる。

ここで注意したいのは，Y 行列を使っている限りブランチを意識する必要がないことだ。ブランチは列車の位置に依存してその本数や位置関係がかわるため，番号をつけるのが難しい。それを意識しないで済むことで，簡単になる要素が非常に多い。

さて，饋電システム内には複数の饋電線があり，それぞれが feedpos 構造体の配列を持っている。その構造体を距離でソートすれば，feedpos 構造体の間の距離が求まるだろう。さらに，こうしてソートした feedpos 配列を利用して，ブランチの数，および饋電定数と距離とからブランチのアドミタンスを求めるのは容易である。こうして取り出されるブランチに関する情報（始点ノード番号，終点ノード番号，アドミタンスなど）は，ブランチ1本あたり1つの bran_node 構造体に格納する。この情報を bran_node 構造体の配列の形で1つのオブジェクトに集め，この中で上記の Y 行列を求める操作を行えば楽であろう。これが feed_y オブジェクトである。従って，feed_y オブジェクトは bran_node 構造体の配列である。table クラステンプレートを使っている。

各 feedline オブジェクトは feedpos 構造体のソートを行い，bran_node 情報を引き出してそれを小さな feed_y オブジェクトに配列として格納する。饋電システム内には全体を統括する feed_y オブジェクトが1つあり，各 feedline オブジェクトが作成する小さな feed_y オブジェクトを統合し，全体の回路について Y 行列作成の作業を行うようになっている（図C.3）。従って，特に必要がなければ接続行列やブランチアドミタンス行列はこの feed_y オブジェクトの中でのみ取り扱われることになる。

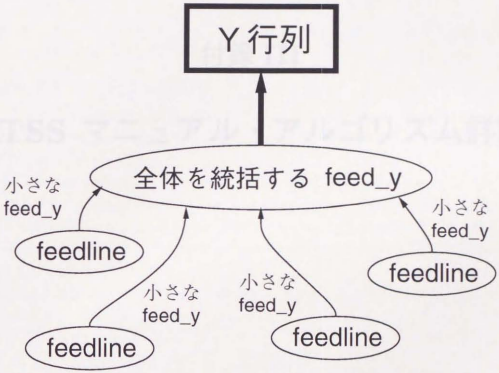


図 C.3: Y 行列の作成にかかわる主なオブジェクト

付録 III

RTSS マニュアル・アルゴリズム詳説

D

配列管理オブジェクト

この章では，いろいろなアルゴリズムを述べる前に，配列管理オブジェクト `table`，`reftable`，`zerotable`，`zeroreftable` について述べる。

⟨D.1⟩ 配列管理オブジェクトの目的

あるオブジェクトの配列を手軽に定義したいというのが，このクラスの基本的な目的である。ほとんどのクラスライブラリに似たような機能のものがあるのだろうが，このプログラムでは自分で書いたライブラリを使っている。

配列管理オブジェクトにおいてやりたいことは，以下のようにまとめられる:

(1) 配列を管理する。通常の C のポインタ機能だと「配列の外に飛び出す」エラーが頻繁に起こるが，それを Segmentation Fault エラーで検出しているのは危険（大量のデータを扱っていると，Segmentation Fault が出ないでプログラムが動き続けることすらある！）。これを正確に検出し，エラーを出して止まらせたい。

(2) 配列に手軽に要素を付け加えたい。可変長の配列。配列を一気に消去したい場合にも対応できる。要素の数を知ることも容易にできる。

(3) 配列を手軽に代入したい。

(4) 配列の中身を見るには，従来と同じく `[]` 演算子が使いたい。

(5) いろいろな種類のオブジェクトに，同じコードで対応させたい。

C++ にとっては，これらの機能の実現はいわば「お手のもの」であるといってよい。

⟨D.2⟩ 考え方

まず，⟨D.1⟩の (5)から，配列は当然のことながらテンプレート機能を使って実現することになる。また，(3)の実現は代入演算子のオーバーロードで，また (4)の実現は `[]` 演算子のオーバーロードでそれぞれ実現できるだろう。`[]` 演算子において，配列範囲外の参照をチェックする機構をつけておけば完璧である。

(2)にあるような要素を付け加えたり削除したりを自在にできるようにする機能の実現に，この種のクラスライブラリのミソがあろう。しかし，RTSS で使っている自作のクラスはこの部分が十分といえないうらみがある。それでも，「1 つ加える」のは `+=` 演算子をオーバーロードすることによって簡単に実現できる。また，配列同士を「くっつける」ことも，`+=` 演算子により簡単に実現できるようになっている。削除の方がいま一つだな，と反省しているところだが，一斉に全部削除なら簡単にできるようにはなっている。

⟨D.3⟩ 関数リファレンス

この table クラス群は，いくつかのクラスの基底クラスとなっている。そのリストを示そう:

- diapattern クラス（nextsta クラスオブジェクトの配列）
- sschar クラス（subchar 構造体オブジェクトの配列）
- sscon クラス（ssfeedconnect 構造体オブジェクトの配列）
- feed_y クラス（bran_node 構造体オブジェクトの配列）
- gradcrv クラス（gcvel 構造体オブジェクトの配列）
- nextsta クラス（gcvel 構造体オブジェクトの配列……gradcrv からの派生クラス）
- gradcrv クラス（gcvel 構造体オブジェクトの配列）
- station_obj クラス（to_station 構造体オブジェクトの配列：現在設計中）

これらが table，reftable，zerotable，zeroreftable のどれかを基底クラスとして持つクラスとなっている（いうまでもないことだが，テンプレートクラスであるから型はすべて異なっており，関数名が同一という以上のことはないのだが）。つまり，以下で述べる関数はすべてこれらのクラスも共通に持っていることに注意してほしい。

⟨D.3.1⟩ 変数

基本的な変数は 3 つ，num，data，mmax である。

なお，以下で TP 型というのはテンプレート機能で置換される前の型をいう。例えば，宣言で

```
template<class TP> class table {
    TP* data;
}
```

などと書いておいて，さらに

```
table<foo> a;
```

と書かれたならば，a においては TP は foo と置き換わっていることになる。ここには基本的に int でも double でも何でも書くことができる。

data … TP * 型，すなわち TP オブジェクトへのポインタである。これが配列そのものを表す。

mmax … 現在 data ポインタの指している配列が，TP オブジェクトいくつ分のメモリを持っているかを示す。当然のことながら，mmax 個以上のオブジェクトをここに持つことは許されない。

num … 現在配列にいくつのオブジェクト TP があるかを示す。これは mmax の値よりも小さければよい。すなわち，配列の中身を「すべて消す」操作をしたいなら，num の値を単に 0 にすればよいことになる。

⟨D.3.2⟩ コンストラクタ

どのクラスも，引数なしのデフォルトコンストラクタと，いわゆる X(X&)（参照による初期化用コンストラクタ）しか持っていない。

デフォルトコンストラクタは，単に data にヌルポインタを，num と mmax にゼロを代入するだけである。参照による初期化用コンストラクタは，mmax，num および data の中身を（ポインタをコピーするだけではなく）きちんと 1 要素ずつコピーする。

⟨D.3.3⟩ 要素を参照する

基本的には，operator[]() 関数を使うことによって可能なのだが，table，zerotable と reftable，zeroreftable とでは機能に大きな違いがある。すなわち，前 2 者では返り値が TP オブジェクトそのも

の（のコピー）であるのに対し，後2者では返り値がTPへの参照となっている。つまり，ref がついていると配列の中身をあとで変更可能ということになる。従って，この両者は時と場合によって使いわけをしなければならないことになる。特に前2者の場合，この関数は const メンバ関数である（この辺は C++ の解説書を参照のこと）。なお，後2者において，operator[]() が const でないために困るケースが出てくることがあるため，operator[]() と同一の仕組みながら TP 型（参照型ではなく）を返す関数として element() を用意している。

この点を除けば，関数 operator[]() の機能は基本的に同一である。table<foo> 型のオブジェクト a の持つ配列の3番めのデータにアクセスしたい場合，

```
foo b = a[3];
```

のように，通常のポインタに対するのと同じように書くことで，メモリ上確保された範囲を越えることなくデータにアクセス可能となる。もちろん，

```
foo b = a.operator[](3);
```

でもよい。メンバ関数ならば後者の方法がよいだろう。

(D.3.4) データを加える

基本的には，operator+=() 関数を使うことによって可能である。　operator+=() 関数は，引数として TP 型ををとる場合と自分自身の型（table<foo> なら table<foo> など）をとる場合とがある。

前者では，引数に与えたオブジェクトを配列の一番末尾にコピーする。もし，配列がすでに確保したメモリいっぱいまで使われてしまっているならば，配列を大きくする操作を行う（一時的記憶領域に配列の中身をコピーし，data を delete し，より大きい領域を new し，中身を戻す）。ここで，table および reftable クラスは「より大きい値」として，直前のより GETMEMORYNUMBER だけ大きな配列をnew 演算子で得るようにしている。+= 演算子は通常1つのデータのみを与えるが，1つだけならば単に1つだけ大きな配列を new してくれば足りる。しかし，そうせずに GETMEMORYNUMBER（現在40に#define されている）だけ大きな領域を確保することにより，高価な操作である delete や new の回数を削減することができる。ただ，場合によっては記憶領域が無駄になることもあるため，zerotable および zeroreftable で「1つだけ」のタイプのものも用意はしてある。

後者の場合は，ふたつの配列をつないで一つにする。もちろん，+= 演算子の左辺が前，右辺が後となる。table・reftable とzerotable・zeroreftable の差異は前者と同じである。

(D.3.5) 要素の数を知る

number() 関数を使うことによって可能である。　number() 関数は引数をとらない。int 型であり，単純にnum を返すだけである。

(D.3.6) データを消去する

任意のデータを1つだけ消去することは，現時点では不可能である。しかし，データをいっきにすべて消去することなら，renew() 関数を使うことによって可能である。　renew() 関数は void 型である。引数として int 型を1つとる場合と，引数なしの場合がある。引数なしの場合は単純に num を 0 にリセットするだけである。引数をとる場合，将来的にはその数だけのオブジェクトを格納する可能性があることをオブジェクトに知らせる。すなわち，

```
table<foo> a;
a.renew(45);
int b = a.number();     // ここでは b = 0 となる
```

としたばあい，a は少なくとも 45 個の foo オブジェクトを格納することができるだけの記憶領域を動的に確保する。しかし，その後でnumber() 関数を参照してみると，0 となる，すなわちオブジェクトは記憶領域だけは確保したもののオブジェクトは一つも「もっていない」状態になっていることがわかる。

これの応用で，setnum() という関数も用意されている。　これはvoid 型の関数で，引数として int 型を 1 つとる。renew() したあとで num 変数を希望の値にセットする。例えば，

```
table<foo> a;
a.setnum(45);
int b = a.number();
```

のように操作すると，b は 45 となる。ただし，以前number() で返る値が 20 だったものをこの setnum() 関数で 45 に増やした場合，新しい配列の 0〜19 番めに古い値は保存されないことに注意する必要がある。

(D.3.7)　その他

このクラスの派生クラスでは，error26_name() なる関数をプライベート関数として用意しておくことが推奨される。　error26_name() 関数は void 型である。引数は何もとらず，operator[]() 関数で添字変域範囲の逸脱を起こした場合に表示すべきエラーメッセージの表示ルーチンを書き込む。こうすることによって，エラーメッセージからのディバッグがより容易になることだろう。

E

等価回路演算の方法

まず、この章では電気的評価の基本となる等価回路演算の方法の概要を述べる。

⟨E.1⟩ Newton-Raphson 法を用いた饋電等価回路演算

⟨E.1.1⟩ 考え方

図 5.2（22ページ）の直流饋電等価回路は，変電所・列車をのぞけば抵抗だけを含む回路である。変電所は電圧源＋直列内部抵抗で，列車は電流源でそれぞれ近似することが多いが，このような近似が成立していればこの回路は行列演算 1 回で解くことができる。すなわち，回路の Y 行列を用いて，変電所・列車の電流・電圧ベクトルを $\boldsymbol{I}$・V とするならば、すでに 129ページで述べた式

$$\boldsymbol{I} - Y \cdot \boldsymbol{V} = 0 \tag{C.7}$$

を解けば容易である。

しかし，列車・変電所は非線形性を持っている。例えば列車は速度が低いか電圧が高いときは電力一定負荷に近くなり，逆に速度が高いか電圧が低いときは純抵抗に近くなる。最近はブレーキ時に電動機で発電して電力を架線に返す電力回生ブレーキ付き電気車が一般的になっているが，この回生中電圧が上がると「絞り込み」という制御をかけて電流を抑制するようにしている。一方，変電所は，負荷である饋電システム側から電源側への逆流はできないのがふつうである。

これらの非線形性の存在により，従来のプログラムでは

(1). ある $\boldsymbol{V}_0$ を仮定し，$i = 0$ として (2) へ
(2). $\boldsymbol{V}_i$ と式 (C.7) から $\boldsymbol{I}_i$ を求める
(3). $\boldsymbol{I}_i$ と列車・変電所の特性から $\boldsymbol{V}_{i+1}$ を求める
(4). $\boldsymbol{V}_i$ と $\boldsymbol{V}_{i+1}$ との差が収束判定限界内なら終了，そうでなければ i を一つ増やして (2) へ

という手順を踏んでいた。しかし，この方法では収束がきわめて悪く，解けないケースも多かった。

RTSS では，この部分には多変数の Newton-Raphson 法を用いている。まず，それぞれの列車・変電所の特性は V–I ないし I–V 平面上で 1 本の曲線として表示されると仮定しよう。このとき，曲線は 1 つの媒介変数を用いて記述できるはずである。1 列車または 1 変電所あたり 1 つの媒介変数が必要である。列車または変電所 No. i の特性を表す媒介変数を θ_i，それを集めてベクトルとしたものを $\boldsymbol{\theta}$ とすると，

$$\boldsymbol{V} = \boldsymbol{V}(\boldsymbol{\theta}) \tag{E.1}$$

$$\boldsymbol{I} = \boldsymbol{I}(\boldsymbol{\theta}) \tag{E.2}$$

と書き表せる。こうすると，式(C.7)を満たす$\boldsymbol{I}, \boldsymbol{V}$を求める問題は

$$\boldsymbol{I}(\boldsymbol{\theta}) - Y\,\boldsymbol{V}(\boldsymbol{\theta}) \equiv \boldsymbol{F}(\boldsymbol{\theta}) = 0 \tag{E.3}$$

を解いて$\boldsymbol{\theta}$を求める問題に帰着できる。これをNewton-Raphson法により解けば解が求まる。

この方法でも計算不能となる場合がまだ残るが，従来のプログラムよりは格段に少なく，また繰り返し計算の回数も減少し，計算時間の短縮が図られた。

⟨E.1.2⟩ 多変数 Newton-Raphson 法

多変数のNewton-Raphson法は，連立方程式を数値的に解く方法として大変ポピュラーな手法であり，収束も速いことが知られている。その方法の概要は以下の通りである。

連立方程式

$$\boldsymbol{F}(\boldsymbol{\theta}) = \boldsymbol{0} \tag{E.4}$$

は，第0近似解$\boldsymbol{\theta}_0$が与えられたとき、次の漸化式を用いて解くことができる。

$$\boldsymbol{\theta}_{k+1} = \boldsymbol{\theta}_k - J^{-1}\,\boldsymbol{F}(\boldsymbol{\theta}_k) \tag{E.5}$$

ただし，Jはヤコビ行列:

$$J = \begin{pmatrix} \dfrac{\partial f_1}{\partial \theta_1} & \cdots & \dfrac{\partial f_1}{\partial \theta_n} \\ \cdot & & \cdot \\ \cdot & & \cdot \\ \dfrac{\partial f_n}{\partial \theta_1} & \cdots & \dfrac{\partial f_n}{\partial \theta_n} \end{pmatrix} \tag{E.6}$$

である。ここに

$$\boldsymbol{F}(\boldsymbol{\theta}) = (f_1, f_2, \ldots, f_n)^T,$$
$$\boldsymbol{\theta} = (\theta_1, \theta_2, \ldots, \theta_n)^T,$$

また$\boldsymbol{\theta}_k$は解くべき方程式の第k近似解である。

⟨E.1.3⟩ プログラム

`feeder::ycal()` 関数で等価回路のY行列を作成し，次いで`feeder::ccal()` 関数で等価回路を数値的に解く操作を行う。`ccal()` 関数の方に，Newton-Raphson法の基本的な部分が含まれている。また，これらの関数を順次呼び出す関数として，`feeder::simccsub()` 関数が用意されている。

⟨E.1.3.1⟩ Y行列の作成 `simccsub()` 関数で，計算の最初にまず`ycal()` 関数を呼び，Y行列を求める。この`ycal()` 関数内では，次のようなことを行う。

(1) まず列車オブジェクト`train`から饋電線接続点データとなる`feedpos`構造体を引き出し，対応する饋電線オブジェクト`feedline`に順次記憶させてゆく。こうすると，`feedline`がもともと持っている変電所と饋電線の接続点情報を合わせ，列車・変電所と饋電線のすべての接続点，すなわちノードに関する情報が，複数の`feedline`オブジェクトに分散して集められたことになる。

(2) このあと，それぞれの`feedline`オブジェクトの中で配列になっている`feedpos`構造体を距離でソートする。そうすると，隣接する`feedpos`構造体の間がブランチということになる。`feedline`オブジェクトがデータとして持っている饋電定数値に`feedpos`同士の距離をかければ、ブランチのインピーダンスが求まる。`feedpos`はその点のノード番号も知っているから，ブランチはどのノードからどのノードまでを結ぶかがわかる。こうすると，すべてのブランチについて，ブランチの起点・終点ノード，インピーダンスが把握できることになる。こうして，1つのブランチに対して1つの`bran_node`構造体が作られる。

なお，それぞれの feedline オブジェクトには feeder_shape 列挙型定数データが含まれている。これが Long であれば通常の直線上の路線となり，Circle であれば饋電線の末端同士がつながれたモデルになる。Circle は山手線のような環状路線のシミュレーションにおいて用いられる。また，ブランチインピーダンス決定に際して，変電所母線と饋電線接続点が離れているようなケースを扱えるよう，饋電線の長さに比例する項だけでなく饋電線接続点とそのノードとの間に一定のインピーダンスを挿入できるような機能も備えている。さらに，隣あうノードの距離が 1m を下回る場合は 1m として計算する例外処理機能もある。

(3) たくさんの bran_node 構造体がそれぞれの feedline オブジェクト内で生成された段階では，ブランチはまだナンバリングされていない。これは，ブランチの全体数がまだはっきりしないからである。そこで，これらを一箇所に集めることを考える。まずは，それぞれの feedline オブジェクトから bran_node 構造体の配列を feed_y クラスのかたちで返す。すでに (C.7) ないしは図 C.3にて述べた通り，feed_y は bran_node の配列として定義されている。

なお，(2)・(3) の一連の操作は，それぞれの feedline オブジェクトに対して feedline::sortx() 関数 を呼び出すことで行われる。この sortx() 関数は feed_y オブジェクトを返す。

(4) 返された複数の feed_y オブジェクトをまとめ，一つの feed_y オブジェクトとする。この操作は，sortx() 関数が返した feed_y オブジェクトを，feeder クラスのメンバオブジェクトである feedy に += 演算子で加えることによって実現される。

(5) こうすると，すべてのブランチに関する情報，すなわち bran_node 構造体が一つの feed_y オブジェクト feedy の中に集められたことになる。こうなれば，ブランチの番号は feedy における配列の順番として定義すれば容易であり，節点—枝接続行列，ブランチアドミタンス行列，従って求める Y 行列を求めることは，実に容易であることがわかろう。

ところで，Y 行列をこのようにして求めると，変電所と列車が隣接している場合にその両者の間にあるアドミタンスが非常に大きくなる。このときにはごくわずかの列車・変電所の電圧のずれが大きな電流になることとなり，結果として収束が悪くなる。そこで，129ページで述べた式

$$\boldsymbol{I} - Y \cdot \boldsymbol{V} = 0 \tag{C.7}$$

において，列車と変電所とで電圧・電流を逆に見ることにしてある。そのために Y 行列を 4 分割し，書き換えるルーチンも作ってある。例えば，変電所が電圧源として計算され，そのきわめて近傍にある列車が電流源であるなら，その間のアドミタンスが非常に大きくても極端に大きな電流がながれるようなことはなくなり，収束がわずかながら改善される。

(E.1.3.2) Newton-Raphson 法による演算 基本的な Newton-Raphson 法の考え方は上記に述べたが，収束「しない」場合の対策としてリミットサイクル発見ルーチンを設けてある。これは，式 (E.3) の左辺の $\boldsymbol{F}(\boldsymbol{\theta})$ のうち最大のものの値が周期的に現れるようになったらそこでリセットをかけるというものである。そのために，ccal() 関数の冒頭にその値を格納する配列 ccbftt が static に定義されている。

リミットサイクルを検出したばあい，まず 1 サイクル分の媒介変数の平均をとり，それから計算をやり直す。それでも失敗する場合は，いったん ccal() 関数を抜け，計算失敗しないように列車の特性を変更してやり直す（この場合は誤差が出る）。このリトライもまた失敗したならば，計算そのものが不能であるといって諦めるプログラムとしてある。

⟨E.2⟩ 変電所における電圧–電流特性の実現

ここでは、変電所における V–I 特性の実現方法を簡単に論じる。列車についても媒介変数を使う部分については同じだが、速度などの要素が複雑に絡むため同一には扱えない。そこで、ここでは論じず、付録 (III) F章にて改めて論じる。

(E.2.1) 考え方

変電所の特性を表すのに必要なデータは，電圧–電流特性である。変電所の位置と饋電線との接続関係は饋電線データファイルにおいて与えられる。この電圧–電流特性の与え方は次のように至ってシンプルである。

まず，変電所の電圧–電流特性は V–I 平面上の折れ線で与える。折れ線は電流については正側（電力供給側）および負側（電力回生側）のどこを通ってもよいが，あとで饋電等価回路演算の際困らないようなものを与えないと回路演算失敗の原因を作ることになるので注意が必要だ。また，当然のことながら電圧の方は正側のみを通るように特性を与えなければならない（与えようと思えば負電圧も与えられないことはないが）。

特性の折れ曲がる点は，通常 3 つの成分（媒介変数，電圧，電流）によって表示する。その点は変電所毎に任意の数を設定できる。例えば，折れ曲がる点のうち i 番目のものの 3 成分がそれぞれ θ_i, V_i, I_i，点 $i+1$ の 3 成分がそれぞれ $\theta_{i+1}, V_{i+1}, I_{i+1}$ と与えられているとすれば，点 i から点 $i+1$ までの間の特性は媒介変数を用いて

$$V(\theta) = \frac{\theta-\theta_i}{\theta_{i+1}-\theta_i}(V_{i+1}-V_i)+V_i \tag{E.7}$$

$$I(\theta) = \frac{\theta-\theta_i}{\theta_{i+1}-\theta_i}(I_{i+1}-I_i)+I_i \tag{E.8}$$

と表せる。要するに線形に補間しているだけである。

(E.2.2) プログラム

特性の折れ曲がる点を与えるのが subchar 構造体である。この構造体を配列にしたものが sschar クラスである。sschar クラスは配列クラステンプレート zeroreftable を用い，これからの派生として定義されている。

変電所は elecchar クラスとして表現される。このクラスのオブジェクトが変電所の数だけ存在することになる。この elecchar オブジェクトは sschar クラスのオブジェクトを一つずつ持ち，subchar オブジェクトを必要な数だけ sschar クラスにデータとして持たせるようにしている。

これらの構造体およびクラスの中身は 'elecchar.hh' で定義されている。

subchar 構造体は teta, volt, ampere, controlnext, voltlow の 5 成分で構成されている。これらのうち controlnext だけが bool 型変数で，他は double 型である。これらのうち teta, volt, ampere はそれぞれ式 (E.7)・(E.8) における θ_i, V_i, I_i に対応する。これ以外の 2 つは「変電所電圧リアルタイム制御・タイプ A」に対応するもので，後述する。なお，teta は綴りが違う (正しくは theta) が，θ のつもりだ。

各 elecchar オブジェクトは現在の電圧・電流に対応するθ の値を変数 teta として保存している。こうして，回路計算がさきに述べたような極めてシンプルなアルゴリズムでできるようになる。

sschar データから変電所の電圧電流特性を実現する関数は，このアイディアによって極めてシンプルに書けている。elecchar::teta_vi() 関数がこれを実現する。この関数では，現在 elecchar オブジェクトが持っている teta の値に対応する電圧・電流値を計算する機能を持つ。これとあわせ，Newton-Raphson 法の計算に必要な，その位置での関数の微係数の計算も行っている。

なお，teta は subchar 構造体の指し示す各点に対応する媒介変数を表すだけであるから，自由に選ぶことができる。ただし，elecchar のメンバである subc 配列（sschar 型）の最初の subchar 構造体に記述された値より小さい teta，および最後の subchar に記述されたのより大きい teta は許されないようになっており，この関数で teta の変域も変域内に抑え込まれる。また，subc 配列の前の方ほどデータとして与えられる teta の値も小さいことを仮定してプログラムが書かれている。データファイルのこの点のチェックはしていないので，ユーザはデータファイルの記述においてはきちんと考慮を払う必要がある。なお，電圧・電流の方は，大きい方から書いても小さい方から書いても，順にみてゆくため，特性が正しければ問題はない。

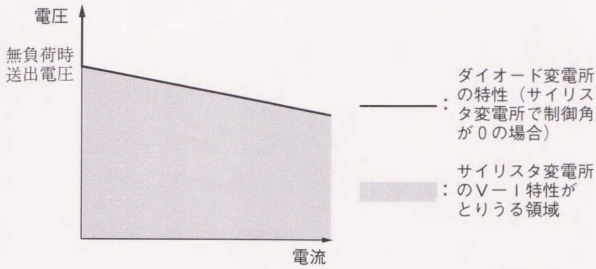


図 E.1: ダイオードおよびサイリスタ変電所の V–I 特性

(E.2.3)　ノウハウ

ところで，データをきちんと与えたとしても，計算に失敗することはよくある。折れ線でデータを与えている限りこれは仕方のないことかも知れない（実物は折れ線でも何でもないからだ！）。現在までに高木が経験したこととしては次のようなことがあげられる。

- ふつうのいわゆるシリコン変電所は無負荷時送り出し電圧以上では電流ゼロで，無負荷時送り出し電圧以下の場合電流の増加にともなって電圧が低下していくようにモデル化される（図 E.1）。しかし，このモデルでは特に負荷が小さい場合に計算失敗を起こしやすい。対策として次のようなことが有効であった。
  - この部分の特性を負側にわずかに傾ける。
  - この部分は媒介変数の変域を多めにとる。
- 複雑な形態の特性も計算失敗の原因になる。特に，インバータつきの変電所をモデル化する場合は要注意だ。このようなばあい，インバータとコンバータを一つの特性にモデル化するのではなく，分けてモデル化することが有効だった。

計算失敗の例を示そう。表 E.1(1) のようなデータファイルを与え，計算を行った。ところが，この特性を与えたところ，計算失敗がかなりの頻度で（等価回路演算 6 回に 1 回程度）起こった。こんなに失敗するのでは計算結果は使い物にならないだろう。ところが，これを訂正し，同一特性ながら表 E.1(2) のように変更したところ，計算失敗はまったく起きなくなったのである。

このファイルは S ファイルであるから，データファイルの意味は (J.4) を参照されたい。また，特に注意していただきたいのは，変更したのは特性上の折れ曲がり点の電圧・電流値ではなく，媒介変数の値のみであることだ。主としていわゆる V–I 特性の「垂直部分」（電流がゼロの領域）に対して，媒介変数の「幅」を大きめに与えることで計算失敗が防止されていることがわかる。

すなわち，問題は V–I 特性がタテに垂直に折れ曲がっている部分に集中しておきているようだ。この部分については何らかの特例的なコードを書いてやる必要があるのかも知れない。その場合，後述する、電車のモデルで回生絞り込み電圧以上になった場合のケースが参考になるかも知れないと思っている。

表 E.1: 同一変電所特性でもデータの与え方により計算失敗が起こる

```
substations 8

# (0) A SS
subchar 5
-3610.0 1800.0 -3333.333334
-3360.0 1550.0 -3333.333334
-30.0 1550.0 -1.0
0.0 1520.0 0.0
20000.0 1070.0 10000.0
ratedcurrent 4000.0 2.0 2.5 3.0

# (1) B SS
subchar 3
-280.0 1800.0 -1.0
0.0 1520.0 0.0
20000.0 1070.0 10000.0
ratedcurrent 2000.0 2.0 2.5 3.0

# ……以下略
```

(1) 大量のエラーが起きたケース

```
substations 8

# (0) A SS
subchar 5
-43300.0 1800.0 -3333.333334
-23000.0 1550.0 -3333.333334
-20000.0 1550.0 -1.0
0.0 1520.0 0.0
20000.0 1070.0 10000.0
ratedcurrent 4000.0 2.0 2.5 3.0

# (1) B SS
subchar 3
-20000.0 1800.0 -1.0
0.0 1520.0 0.0
20000.0 1070.0 10000.0
ratedcurrent 2000.0 2.0 2.5 3.0

# ……以下略
```

(2) エラーが起きなかったケース

F

列車の電気的モデル

列車モデルは，列車の運動方程式，電流—電圧特性，応荷重特性などを模擬するものである。電気的な，つまり列車の運動そのものに関係ない変数や関数は変電所のものと同じであるので，train クラスは elecchar（変電所モデルを格納するクラス）からの派生クラスとして定義されている。この章では，主としてこの列車モデルのうち電気的部分のアルゴリズムを詳細に論じる。

RTSS で利用する等価回路表現においては，変電所も列車も同一のノードであり区別がないこと（ただし変電所に限り複数の饋電線と関係を持つことができることだけが違う）から，変電所と列車を同じ変電所クラスへのポインタからアクセスすることにより，オブジェクト指向プログラミングのまねごとができている。いうまでもなく，列車と変電所ではその中身はまるで違うが，電圧と電流が関係を持って変化するという点では同一なのである。その点をある程度（巧みに，と書きたいところだけれど自分で書くと照れくさいので書かない）利用したのが現在の RTSS のクラス定義の形態である。

なお，列車のモデルとしてはインバータ制御のものだけをプログラムしてある。これでインバータ制御および 4 象限チョッパ制御の電気車はシミュレートできると思うが，他のタイプのものは不可能だ。もっとも，インバータ制御であっても若干の制限やモデル誤差は存在せざるを得ない。この辺の詳細も述べることにしよう。

〈F.1〉 基本

列車の電気的特性を，最終的には I–V 平面上の 1 本の曲線として表し，それを媒介変数表示してやろうというのが基本的な考え方である。ところが，実際には速度により電流が大幅に変わるわけだから，実車と同様に速度に応じて列車の I–V 特性を変更するための機構を用意しておかなければならない。

〈F.1.1〉 考え方

まず，電圧が一定とした場合の議論から入る。インバータ制御電車のトルク（引張力）–速度曲線を考えよう。これは例えば図 F.1のように与えられる。このうち「定トルク領域」とは速度によらずトルクが一定の領域である。また,「定パワー領域」は速度によらずパワーが一定の領域で，トルクは速度に反比例する。最後の「フリーラン領域」は電動機の特性に従って加速して行く領域である。従来の直流電動機とのアナロジーから，この領域ではトルクは速度の 2 乗に反比例すると考えられる。

パンタ点電流–速度曲線について見ると，図 F.2のようになる。低速域での損失があるものの，定トルク領域ではほぼ速度に比例したパンタ点電流が流れる。定パワー領域はその名の通りパワー一定なので電流は速度によらず一定となる。またフリーラン領域では電流は速度に反比例ということになる。

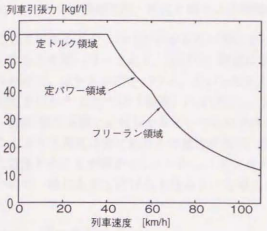


図 F.1: 列車の引張力–速度曲線の例

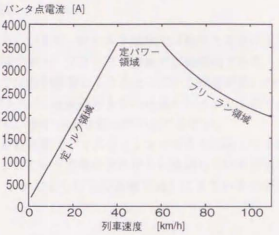


図 F.2: 列車のパンタ点電流–速度曲線の例

まず，これらを式にして示そう。列車速度を v とする。定トルク領域では，速度 0 での電流を I_0，定トルク領域から定パワー領域に移行する速度を v_l，v_l での電流を $I_{\max}$ とするならば

$$I = I_0 + (I_{\max} - I_0) \cdot \frac{v}{v_l} \tag{F.1}$$

となる。次いで，定パワー領域では

$$I = I_{\max} \tag{F.2}$$

となる。さらに，フリーラン領域では，定パワー領域からフリーラン領域に移行する速度を v_h とするならば

$$I = I_{\max} \cdot \frac{v_h}{v} \tag{F.3}$$

で表される。

さて，実際に回路計算をする際には，列車の速度 v は定数である，すなわち回路演算の最中に変化することはないと仮定して計算を行う（**付録 (II) B**参照）。しかし，電圧は変化すると考えなければならない。このとき電流–電圧特性はどのように変化させたらよいのだろうか？ ここで次のような仮定を置く:

- **v_l および v_h はパンタ点電圧に比例する。**

すなわち，列車のパンタ点電圧を V，v_l および v_h のパンタ点ノミナル電圧値 V_N における値をそれぞれ v_{low}，v_{high} とすると

$$v_l = v_{\mathrm{low}} \cdot \frac{V}{V_N} \tag{F.4}$$

$$v_h = v_{\mathrm{high}} \cdot \frac{V}{V_N} \tag{F.5}$$

となる。

この仮定は必ずしも正確とはいえないものの，それほど不自然なものでもない。少なくとも，従来のシミュレーションの経験からすると，大きな誤差がこれを原因として出たことはなかった。

このことを踏まえ，式 (F.1)〜(F.3) の 3 つの式をじっくり見ると，つぎのようなことがいえる。

(1) 定トルク領域では，電圧と電流が反比例関係にある。
(2) 定パワー領域では，電流は電圧によらず一定である。

(3) フリーラン領域では，電流と電圧は比例関係にある。

ここまで来ると少々わけがわからない話になってしまう。つまり，定トルク領域では電圧と電流が反比例，すなわち定パワーであり，定パワー領域は定電流領域であり，フリーラン領域では純抵抗である，というわけだ。定トルクが定パワー，といったあたりですでに頭が爆発しそうになっている読者が多いのではあるまいか？ ここでの「領域」の呼び名は，あくまでトルク–速度曲線上での性質からとったものであり，電流–電圧曲線上の性質からとったものではないことに，特に注意を払っていただきたい。

このような混乱は常に起きる可能性があり，現に高木が研究室でディスカッションする上にあっても説明に苦慮することが多かった。しかし,「定トルク領域」などという言葉はそれなりに流通している言葉らしいので，他にあえて呼び名を作ることも考えられず，しかたなしにこの言葉で通してきているのが現状だ。

⟨F.1.2⟩ プログラム

`train::teta_vi()` 関数が，基本的にこのあたりのことを取り扱っている。いわゆる仮想関数というものになっており，`train` オブジェクトに `elecchar` へのポインタからアクセスしても `elecchar::teta_vi()` 関数ではなくこちらが呼ばれるようになっている。ここではまだノッチ・荷重の考慮をしていない（⟨F.2⟩ 参照）が，やっていることの基本は `elecchar::teta_vi()` 関数と同一である。この関数の中で，各変電所・列車オブジェクト毎に与えられた媒介変数（各オブジェクト内では `tt` という変数に保存）から電圧・電流およびそれらの媒介変数による微係数（$dV/d\theta$，$dI/d\theta$）を求める。

まず，列車では定電圧特性は考えられないことを利用し，`tt`（各列車・変電所オブジェクトが持っている媒介変数 θ）からまずパンタ点電圧を決めてしまうようにしている。この場合，`tt` $= 0$ がノミナル電圧値，すなわち 1,500V になるように

$$V_P = \theta + 1500 \tag{F.6}$$

としている。このノミナル電圧値を変更できるようにはしていないが，近い将来したいと思っている。

電圧がまず決まってしまうので，あとは電流関係を決めればよいことになるが，それには ⟨F.1.1⟩ で述べたいくつかのパラメータを決めてやらなければならない。

力行時については，必要なデータを `powerdata` なる構造体に格納している。それぞれの `train` オブジェクトは，この `powerdata` 構造体を `pdata` というメンバ変数として 1 つずつ保持している。

列車速度 v … これは現在の状態として与えられ，変数 `vel` として記憶されている。
パンタ点電圧 V … これは媒介変数からすでに式 (F.6) によって求まっている。
起動時電流 I_0 … データ `pdata.zerocur` にて与える。
定トルク・定パワー領域境界速度 v_{low} … データ `pdata` によって計算した値を `vlow_p` として保存し，これを用いる。
定パワー・フリーラン領域境界速度 v_{high} …データ `pdata` によって計算した値を`vhigh_p` として保存し，これを用いる。
力行時最大電流 I_{max} … データ `pdata` によって計算した値を`ampmax_p` として保存し，これを用いる。

回生時については基本的に力行時の裏返しなのだが，定トルク領域についてはわずかに考え方が異なっている。力行時は速度 0 でもトルクが出るかわりに速度 0 での電流が与えられるが，回生時は速度がある値 v_{off} を下回るとトルクが出ないモデルとしている[1]。v_{off} を用いると，ブレーキ時には式 (F.1) が次のように置き換えられる:

$$I = \begin{cases} I_{\mathrm{max}} \cdot \dfrac{v - v_{\mathrm{off}}}{v_l - v_{\mathrm{off}}} & \text{for } v \geq v_{\mathrm{off}}, \\ 0 & \text{otherwise.} \end{cases} \tag{F.7}$$

[1] 逆相ブレーキを使うモデルも考えられないことはない。これも，必要ならば将来インプリメントすることは考えられよう

必要なデータは力行時と同じように brakedata なる構造体に格納している。それぞれの train オブジェクトは，この brakedata 構造体を bdata というメンバ変数として 1 つずつ保持している。当然ながら，力行時と回生時とでは境界速度 v_{low}，v_{high} や最大電流値 $I_{\max}$ も異なるため，ブレーキ用に別なデータを保持することになる。

列車速度 v … これは現在の状態として与えられ，変数 vel として記憶されている。
パンタ点電圧 V_P … これは媒介変数からすでに式 (F.6) によって求まっている。
回生オフ速度 v_{off} … データ bdata.regenoff にて与える。
定トルク・定パワー領域境界速度 v_{low} … データ bdata によって計算した値を vlow_b として保存し，これを用いる。
定パワー・フリーラン領域境界速度 v_{high} …データ bdata によって計算した値をvhigh_b として保存し，これを用いる。
回生時最大電流 $I_{\max}$ … データ bdata によって計算した値をampmax_b として保存し，これを用いる。

なお，現在のプログラムでは，せっかく列車毎に性能データ powerdata，brakedata を持っているのに，入力ルーチンの都合ですべての列車が同一性能となってしまっている。

⟨F.2⟩ ノッチ，荷重の考慮

実際には，列車の I–V（電流–電圧）特性曲線は速度以外にもノッチ（加速の強さを表すものと考えればよかろう），そして荷重によって変わってくる。本章ではそれらを考慮するためのモデルを与える。

通常，このような特性を記述するために必要なデータがすべて与えられることはまずない。RTSS が開発・維持されている大学のような環境はもちろん，鉄道総研のような研究所，ないし場合によってはメーカですらこのようなデータの詳細が得られないことがあるようだ。そのため，合理的と思われるある種の仮定をおいて，速度・ノッチ・荷重変化時の特性を近似することになる。RTSS の仮定は，そういう観点からすると決してよい近似になっているとは思えない。

⟨F.2.1⟩ 応荷重装置のモデル化

⟨F.2.1.1⟩ 考え方 o 応荷重装置のモデリングは，RTSS ver.2.0 までは 2 点指示方式を用いていた。これは，Inv_pmode1（定トルク領域），Inv_pmode2（定パワー領域），Inv_pmode3（フリーラン領域）の境界の速度が，混雑率 pbempty.conges のときそれぞれ pbempty.lowvel, pbempty.highvel であり，混雑率が pbfull.conges のときはそれぞれ pbfull.lowvel, pbfull.highvel であったとするとき，lowvel および highvel はこれらの中間の混雑率にあっては混雑率に比例して完全に線形に配分されるとするものだ。詳しくいうと，データとして混雑率 C_l のときの値 K_l, C_h のときの値 K_h が与えられているとき，混雑率 C のときの値は

$$K = \frac{K_h - K_l}{C_h - C_l} \times C + K_l \tag{F.8}$$

で与えられるとするものだ。

このように 2 つの混雑率において値を指定する場合でも，Inv_pmode2 が十分広い速度域にわたっていればほとんど問題はない（少しある）が，一部の電車ではこの領域が極端に狭いものがあり，場合によってはこの補間法は適用できないと考えられる。

そこで，線形な補間法の他に次のような補間法を用意し，スイッチで切替えられるようにした。

線形補間が適用できないケースとは，図 F.3のように定パワー領域がないものである。定トルク領域の引張力に関しては従来と同じ線形補間で計算可能であると考えるべきだ。従来の方法で計算された定トルク領域の引張力の値を T としよう。

フリーラン領域にあっては，引張力は速度の自乗に反比例することが知られている。そこで，混雑率ゼロのときの引張力および定トルク／フリーランモードの境界速度を T_e, v_e，混雑率 C_m のときのそれらを

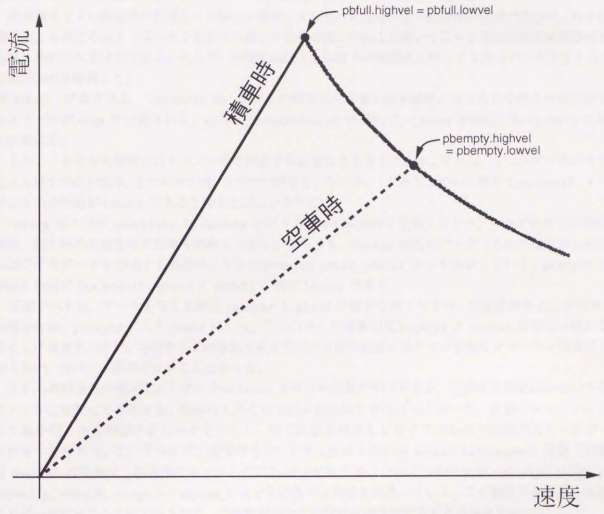


図 F.3: 線形補間が適用できないケースとして想定されるもの

T_m, v_m としよう。このとき，データが正確に

$$T_e v_e^2 = T_m v_m^2 \tag{F.9}$$

と与えられることはないと考えられるので，Tv^2 が混雑率によって線形に変化すると仮定する。このとき，求められた Tv^2 の値を K_{Tvv} としよう。こうすれば，モード境界速度は

$$v = \sqrt{\frac{K_{Tvv}}{T}} \tag{F.10}$$

として容易に求められる。

一方，電流についてはフリーラン領域の電流が速度に反比例することを利用する。混雑率ゼロおよび C_m のときのモード境界速度（それぞれ v_e, v_m）における電流（最大電流）をそれぞれ I_e, I_m とすれば

$$I_e v_e = I_m v_m \tag{F.11}$$

とは正確にならないことから，Iv が混雑率によって線形に変化するものと仮定する。こうして求めた Iv を K_{Iv} とすれば，最大電流は

$$I = \frac{K_{Iv}}{v} \tag{F.12}$$

と求められる。最大電流は電圧によるモード境界速度の変化にかかわらず一定である。

線形補間でよい混雑率の領域とこの新しい補間によらなければならない混雑率の領域の区分は，ある混雑率 C_m を境にそれより低いところがこの新しい補間方式，それより高いところが従来の線形補間領域と簡単に決められるはずである。そこで，RTSS ver.2.1.0 以降では混雑率に関して 3 点のデータを与える 3 点指示方式を採用した。

(F.2.1.2) プログラム 'enumerat.hh' にはこの補間法と従来の線形補間とのどちらを使うかを指定するスイッチが enum で定義される。pbchar_interpolation がそれで，Linear が線形，Inv_mode13 が新しい形式だ。

しかし，どうやら補間方式をスイッチで指定する必要はなさそうである。これは，3 つのデータのうちたとえば 1 つめが空車，2 つめが中間，3 つめが満車としたとき，1 つめと 2 つめの間が Inv_mode13，2 つめと 3 つめの間が Linear であると考えればよいからだ。

'train.hh' では powerdata, brakedata というふたつの構造体を定義しており，それぞれが力行時の性能，回生時の性能を示す指標を格納してある。どちらも，pbchar 形式のデータ（これが混雑率 1 点分に対応するデータを格納する構造体）を 3 つ pbempty, pbmid, pbfull として格納している。pbempty と pbmid の間が Inv_mode13，pbmid と pbfull の間が Linear である。

注意すべきは，データを与える際に pbempty と pbmid の間が有効（つまり，境界混雑率 C_m がゼロ）の場合には，pbempty にしろ pbmid にしろ，二つのモード境界速度 highvel と lowvel は等しい値になることに注意すべきだ。なぜならこの補間方式を利用する際の前提が定トルク領域とフリーラン領域がとなりあい，定パワー領域がないことだからだ。

これらの構造体に格納されたデータは train クラスの関数が利用するが，利用する関数はいろいろなファイルに分散して存在する。利用のしかたは力行／回生問わずほとんど同一だ。共通のルーチンに入れて集中化した方が保守がしやすそうだし，同じ計算を何度もしなくて済むので計算のスピードアップにもつながろう。というわけで，混雑率をセットするために用いる train::setconges() 関数（引数は double）のなかで，混雑率をセットしたのち (F.1.2) にて述べたいくつかの変数，vlow_p, vhigh_p, ampmax_p, vlow_b, vhigh_b, ampmax_b などを計算する関数を用意している。この数字は混雑率を仮定すれば一意に定まる定数であるため，この関数の中で計算すれば他で計算する必要はない。

このほか，応荷重補間に関係する記述がある電気的特性関数は以下の通り：

- train::tracf() 関数。引数は double, double, double で，順に列車速度，パンタ点電圧，フルノッチ比（(F.2.2) に後述）。牽引力を定める関数。
- train::nr_constv() 関数。フルノッチ比を決める関数の定速走行モード用。
- train::ttvi_power() 関数。媒介変数 θ から電圧・電流とその媒介変数による微分値を求めるルーチン。モード境界速度の利用は前の二つと同一だ。
- train::cgdef() 関数。引数は double, double。現在の混雑率から cg という定数を決める。この定数は次の関数で使われる。従来の線形補間のための関数であり，最終的には setconges() 関数において利用されるサブルーチンだ。
- train::cgcal() 関数。引数は double, double, double。前述の cgdef 関数で決めた cg なる定数を利用し，さまざまな線形補間計算を行なう。これも setconges() において利用されるサブルーチンだ。
- train::velcal_mid() 関数。引数は double × 6。cgdef 関数で得た定数 cg および，境界混雑率 C_m 時および空車時の引張力・モード境界速度，現在の引張力を引数に与える。非線形補間方式においてモード境界速度を返す関数だが，このモード境界速度は与えるものも得られるものも 2 つの速度 highvel と lowvel が同一の値であることに注意しよう。プログラムもそのように組み立てら

れる。これもサブルーチン。

- `train::ampcal_mid()` 関数。引数は double × 6。非線形補間方式において最大電流を返すサブルーチン。なお，定トルク領域におけるトルクを返す関数は用意しなかったが，これはトルクのみは混雑率からの線形補間で求めうるものだからだ。

(F.2.2) フルノッチ比

実際の電車では，ある速度において列車がとり得る牽引力を比較的少数の不連続な値のみとすることが多い（最近のインバータ電車などでもそうなっているものがある）。しかし，インバータ電車であれば望み通りの牽引力を常に発揮することができるし，そうであるべきだと思われる。このため，以下で定義するような「フルノッチ比」を考え，プログラムの各所で利用することにしている。

(F.2.2.1) 定義[39] まず，インバータ制御またはチョッパ制御（界磁チョッパを除く）の電気車であれば，出しうる最大の牽引力より小さければどんな値でも自由に好きな牽引力を出せるものと考えよう。また，速度一定と仮定した場合，パンタ点から見た効率が牽引力の値によらず一定と仮定する。このとき，列車速度 v のときに，列車がその速度における最大牽引力 $T_{max}(v)$ を発揮している（これを「フルノッチで加速している」と呼ぶ）ときの電流を $I_{max}(v)$ とすれば，牽引力 T を出しているときの電流 I は

$$I = \left(\frac{T}{T_{max}}\right) \cdot I_{max} \tag{F.13}$$

で与えられる。ここで，$T/T_{max} = r$ とすれば，

$$T = r \cdot T_{max} \tag{F.14}$$

$$I = r \cdot I_{max} \tag{F.15}$$

と与えられる。r は牽引力および電流のフルノッチ時に対する比率を表すことになるので，これをフルノッチ比と定義する。なお，フルノッチ比は力行時には 0 ～ 1，回生時は 0 ～ −1 の範囲の値をとるものとし，回生側も上と同様に定義する。

従来の走行パターンにおいては，フルノッチ比は力行時には 1，惰行時には 0，最大減速度でのブレーキ時には −1 となる。

このような「フルノッチ比」を用いて本章の以下の説明が行われる。

(F.2.3) フルノッチ比の概念の拡張？

特に，ブレーキ時の振舞いを記述するには，フルノッチ比というのを電気的な部分と機械的な部分とに分ける必要を感じている。これは，電気的なブレーキ力（これはある効率で架線に返るから無駄とはいえない。仮に逆相ブレーキのような利用法をしたとしても，ブレーキシュー摩耗はないし，制御性も機械的ブレーキよりははるかによい）と，機械的なブレーキ力（これは効率ゼロである上ブレーキシューの摩耗まで招くから損は 2 重になる）をそれぞれ最適に制御しようなどという目的のためには，ブレーキ力の総和だけを 1 つの「フルノッチ比」で表現するのでは得策ではないと考えられるからだ。

「フルノッチ比」の考え方自体は，関数の与え方を工夫することでいろいろな車種に適用可能だろう。ただ，単純なシミュレーションをやっている分にはよいものの，「フルノッチ比」を用いて何かいいことをしようといった場合，特に抵抗制御電気車の場合はどの範囲まで制御可能かをいつもチェックしていなければならず，困難が伴うことが予想される。抵抗制御車の場合はノッチ戻し[2]が不可だとかいった問題が

[2] ノッチ戻しとは，列車が加速中にいったんあげた「ノッチ」を低い値に戻すことをいう。こうするとというまでもなく加速力が低下する。これができない電車では，通常連続走行可能なノッチを設けておき，加速を弱めたい場合にはそのノッチで「止める」ことをする。自動車のアクセルのように自在に力を加減することは，運転手の操作ではできないことになっているものが多い。直流の抵抗制御車でこれができない理由は，主としてカム軸で制御しているスイッチが接触器であり，電流を入れることはできても遮断できないことによる。最近のインバータ電車でも，ノッチ戻しができる領域を何らかの形で限っているものが多い。

あるため，いずれにしても精密なモデリングは困難だろうとは思う。もっとも，いまさら抵抗制御電車を仮定したシミュレーションはないだろうという希望的な観測ができる時代になったのは，シミュレーションが楽だというようなつまらぬ観点からだけでなく，大変喜ばしいことだといってよいだろう。

⟨F.3⟩ 回生絞り込みモデルと系の最高電圧

回生絞り込み特性は，シミュレーション結果としての回生率や回生電力量に大きく影響するパラメータとなる。これをプログラムがどうモデル化しているかについて，やや詳細に述べる。

(F.3.1) 2種の特性モデルと絞り込み対象の電流

回生絞り込み特性は通常図 7.22（46ページ）のように与えられる。同図の (1) および (2) のいずれのケースも実際に存在しているそうだ。

(1) のケースでは，回生電流の絞り込みは次のように行われる。当該混雑率における最大回生電流を I_max（ただし $I_{empty} \leq I_{max} \leq I_{full}$，パンタ点電圧を V_P，絞り込みを行う前の回生電流を I_R とするとき

$$I_R \geq I_S \equiv I_{max} - I_{full} \times \frac{V_P - V_s}{V_e - V_s} \tag{F.16}$$

であるとき，I_R を I_S まで（$I_S < 0$ ならばゼロまで）絞り込む，というものだ。

一方 (2) のケースでは，

$$I_R \geq I_S \equiv I_{full} \times \frac{V_e - V_P}{V_e - V_s} \tag{F.17}$$

であるときに，同様に I_R を I_S まで（$I_S < 0$ ならばゼロまで）絞り込む。

回生絞り込みを行う理由というのは架線電圧の異常な上昇を防ぐためである。従って，同一の電圧で満車時より空車時の電流が小さくなければならない合理的な理由はない。ゆえに図 7.22(2) の回生絞り込み特性が実現できないが，同図 (1) が実現可能ということはないだろう。そうなれば，(1) より (2) としたほうが空車時の回生能力をより有効に使うことができるといえる。(1) を採用しているケースでは，車両側の特性を変えることによる回生特性改善の余地が少なからず残っている，といういい方もできる。

RTSS は当初 (2) のみをサポートしていた。これは曽根教授の考え方が反映されているもので，こちらがいいという過去の検討における結論を踏まえた割り切りである。いかにも曽根研のシミュレーションプログラムらしい特徴だったと思う。しかし，文献 [73, 74] での検討に用いるさい実際の条件に合わせるべく，ver.2.1.3 より (1) もサポートするように変更した。

ところで，この絞り込み特性の縦軸の電流はどの電流のことをいうのだろう。これにも考え方が2つあった。図 7.23（47ページ）を見ていただきたい。考えられるのは「モータ電流」と「主回路回生電流」とである。

モータ電流はモータの消費している電力とはあまり関係がなく，むしろ電車の引張力（すなわちモータのトルク）との関係が強い。大きなトルクを出している低速では，モータ電流は大きいが，モータの消費する電力は小さいため，インバータの直流側に返る主回路回生電流は小さくなる。したがって，式 (F.16) または (F.17) で I_R ほかが主回路回生電流であるとするならば，V_P が絞り込み開始電圧 V_s を越えていても I_R が小さいため絞り込みが行なわれない可能性が高くなる。このように，主回路回生電流が小さくなる中低速域では，主回路回生電流を見て絞り込む方式では絞り込みなしとなる場合でも，モータ電流を見て絞り込む方式では絞り込みを行うようなケースが出てくる。

混雑率の場合と同様に，同一の電圧で高速時より低速時の電流が小さくなければならない合理的な理由はない。したがって，回生絞り込みはモータ電流ではなく主回路回生電流に対して行った方が電気車の電力回生能力がより生かされる。実際の電車は制御の容易さからモータ電流を見る方式が主流のようだが，図 7.22の特性 (1)(2) の比較と同様，この方式では回生特性改善の余地を残していることになる。

RTSS は当初主回路回生電流を見る方式のみをサポートしていた。このあたりは曽根教授の考え方が反映されているが，絞り込み特性 (2) だけをサポートして (1) は切り捨てるくらいだからこちらをこのようにするのはむしろ当然といえる。しかし，文献 [74] での検討に用いるさい実際の条件に合わせるべく，ver.2.1.4 よりモータ電流を見る方式もサポートするように変更した。

(F.3.2) 回生絞り込みと失効との区別

パンタ点電圧が高くなると絞り込みを始め，ある電圧より上では回生電流はまったく流れない，というような仕組みになっている。実際の電車では，あまり回生電流が小さくなった場合には主回路をオフにして，回生を切ってしまう（これが一般にいう回生失効の状態だ）。

ちなみに，ブレーキ時は回生ブレーキ（電気的ブレーキ）力と空気ブレーキ（機械的ブレーキ）力の和は等しくなるように制御されるため，列車としての減速度がマクロに見て変化するようなことはないが，実際には空気ブレーキ系の立ち上がりの遅さなどがあるため，とくに回生失効の状態になった際の切り変わり時にはショックがあったり，ATO 運転線区では定点停止に支障を来たしたりすることがあるようだ。

そして，いったんこうして主回路を切ってしまうと，その後空気ブレーキのみで電車は止まることになり，止まるまでの間にパンタ点電圧が下がっても回生ブレーキが再びかかることはないのが普通だ。もっとも，なかには，回生失効後 20 秒経つと再び回生にトライするという「飛びつき機能」を持っている電車がある。それに，インバータ制御電気車ならば，主回路を完全に開くのではなく，再び回生可能な状態になるまで待機するようなことも可能であるという。将来はそのような制御方法が一般化することになるだろう。

このシミュレータでは当初この回生ブレーキが「再びかからない」という条件のシミュレーションはできなかった。RTSS ver.2.1.3 以降，この条件のシミュレーションも可能となっている。この条件のシミュレーションにおいては，ある列車の主回路回生電流がある電流値より小さくなった場合に，それ以後の回生を禁止するフラグを立てる。禁止フラグが解除されるのは，フルノッチ比が非負となったばあい，もしくは回生オフ速度を下回った場合のいずれかである。

⟨F.4⟩ 補機負荷のモデル

列車補機負荷のモデリングは，大きく分けてパワー一定モデリングと電流一定モデリングとがある。従来はこの補機モデルが入っていなかったが，これは明らかに単にサボっていただけであった。媒介変数表示だから関数を追加するのはそう難しいことではないのである。

RTSS ver.2.1.1 から電流一定モデリングでサポートした。しかし，これでは評価量にかなりまずい影響があるため（補機パワーが電圧増加に伴って増大してしまう），パワー一定モデルを ver.2.1.3 からサポートした。両方をデータファイルで選択できるようにしてある。

プログラミング自体は簡単である。`train.hh` というファイルで定義される `train` クラスには，`teta_vi()` 関数があり，`train.cc` にてコードが記述されている。ここで `a`（電流）および `di`（電流の媒介変数による微分値）を計算しているが，これを計算し終ったところに `ttvi_auxcurr()` なる関数を挿入しただけだ。この関数のコードは `trteta.cc` にあるが，補機電流の値，およびその媒介変数による微分値を，`a` および `di` に加える操作をしている程度の簡単な関数だ。

特に注意すべきは，列車が消費する電流はマイナスの値をとる約束になっていることだ。また，回生失効率の計算に影響がないように，主回路分と補機込みとで電力計算を分ける必要が生じたため，この関数が主回路の電流の演算の後におかれていることを利用し，補機分を加える前の `a` の値を `shibo` という変数にコピーして保存している。

G

列車の運動モデル

この章では，列車の運動モデルを取り扱う。当然のことながら電気的モデルと運動モデルとは密接な関係があるので，なかなかこのように「きれい」に分割することは難しい。本章の読者は折りにふれて付録 (III) F を参照することになるだろう。

⟨G.1⟩ 運動方程式，単位系

列車運動シミュレーション部は，列車を質点と見なし，列車の引張力・ブレーキ力・走行抵抗・勾配抵抗などをもとに列車の運動方程式をたて，それを数値的に解くものである。列車の運動方程式は次式で表される[19]。

$$W_k \cdot (1+\gamma) \cdot \frac{dV_k}{dt} = F_k - B_k - R_k - G_k \tag{G.1}$$

ただし，W_k : k番目の列車の総質量 [t]
γ : 回転部分による補正係数
V_k : k番目の列車の速度 [m/s]
F_k : k番目の列車の引張力 [kN]
B_k : k番目の列車のブレーキ力 [kN]
R_k : k番目の列車の走行抵抗 [kN]
G_k : k番目の列車の勾配抵抗 [kN]

現在の列車の位置・速度が与えられると，Δt 時間後の列車の位置・速度は式 (G.1) を解くことによって簡単に求められる。そこで，現在の列車群の配置およびすべての列車の速度が与えられているとき，この方程式を列車ごとに解いて行けば，Δt 時間後の列車群の配置および各列車の速度は簡単に求められる。

⟨G.2⟩ 列車の状態と状態遷移則

列車は,「力行」「定速走行」「惰行」「ブレーキ」などの状態を状況に応じて切替えながら走行する。また，同じ「力行状態」でも弱い力行，強い力行などの区別（ノッチの選択）が運転手の動作を完全に模擬することは難しいため，適当な仮定を置いてプログラムが組まれるのが普通だ。この部分は結果に少なからぬ影響を与える部分であり，またこのプログラムのもっとも特徴的な部分でもあるので，ここでやや詳細に説明を加えようと思う。

(G.2.1)　駅間走行時分一定化技術

この種のプログラムとしては，従来からいろいろな種類のものが書かれてきているし，現にいろいろな場所でいろいろなプログラムが用いられている。だが，従来のプログラムは，列車が加速をやめる位置や速度をデータとして与えて列車の駅間走行パターンを指定している。この方法でも列車の性能が一定であれば問題はないが，現実の列車はパンタ点の電圧によって性能が変化する。これをモデルに取り込めば必然的に列車の駅間走行時分が条件通りにならない現象が起きる。

饋電特性の中でももっとも重要な各種エネルギーの評価にとって，駅間走行時分は重要なパラメータである。駅間走行時分の狂いはシミュレーション結果の信頼度を下げてしまうことにつながる。これに対処するため，RTSS では駅間走行時分を高精度に一定化する列車運動シミュレーション技術を開発しインプリメントしてある。この点が，RTSS が他の多くの直流饋電システムシミュレータと異なる著しい特徴となっている。(A.1.1)に述べたように，このインプリメンテーション部分が RTSS の「いちばん古い部分」でもある。

パンタ点電圧が変動し，それに伴って列車性能も変化する条件のもとで，駅間走行時分を高精度に一定化するためには，加速をやめる位置・速度をそのときの条件に応じて求めればよい。パンタ点電圧は前もって予測することが難しいが，列車が加速していないとき（惰行・ブレーキ）には列車の運動はパンタ点電圧と無関係である。

惰行状態における運動がパンタ点電圧と無関係というのは理解しやすいと思われるが，ブレーキ時の列車運動がパンタ点電圧と無関係というのはなぜいえるのだろうか。これは，普通の電気車は電空併用演算をしており，電気ブレーキ力の不足時には空気ブレーキによる補足を自動的に行い，電気ブレーキと空気ブレーキのブレーキ力の和が一定となるように制御しているからである。この演算および制御が正確に行われている限りにおいて，マクロにみた場合の列車の減速度は一定に保たれるのである。

このことを利用すれば，加速中のある瞬間に列車が加速をやめ，あとは目標地点まで加速せずにゆくと仮定した場合の，目標地点までの到着時刻は計算できる。平坦な路線で速度制限などがなく，列車の駅間走行パターンが《力行→惰行→ブレーキ→停止》となる場合，列車は力行中この時間を毎回推定し，推定値が所定の時間より短くなったところで力行をやめればよい。このような，比較的簡単なアルゴリズムで，駅間走行時分一定化シミュレーションが実現できる。

もっとも，このように書くと簡単そうではあるが，実際には列車の走行時間を推定するのには列車の運動方程式を解く必要がある。これをするのにいちいち数値積分をしていたのでは時間がかかりすぎるので，近似計算ルーチンを作っておき，ある精度まではこの近似で追い込むようにしている。

駅間走行時分合わせに失敗した場合，列車の遅れ分は次の駅間にそのまま持ち越されるようにプログラムしている。一方，列車の進み分は停車時分に加算される。シミュレーション条件によっては，(G.2.2)に述べるように駅間走行時分合わせに失敗することもある。最小時間刻み幅程度の誤差はつねに起きる可能性がある。

(G.2.2)　一般的な列車状態と状態遷移則

(G.2.1)で述べた駅間走行時分一定化技術は，列車の駅間走行パターンとして「駅停車」「力行」「惰行」「ブレーキ」の 4 つのみを考え，しかも《力行→惰行→ブレーキ→停止》のような順に現れる単純なパターンのみを仮定している。しかし，局所的な速度制限や勾配が存在するなどの実際の路線の条件を考慮すると，このような単純なパターンだけで議論ができるわけではない。そこで，より現実的なモデルへの適用のためには，走行パターンになんらかの仮定をおく必要が出てくる。

このプログラムでは通常の列車の状態として前記の 4 つのほか「定速走行」の 5 つを考えた。「定速走行」状態は弱い力行または弱いブレーキのどちらにもなり得る状態である。ここでは列車の速度を一定に保とうとするモードであると簡単に理解していただこう。また，駅間の前半は力行領域，後半は惰行領域とし，力行状態，または弱い力行で速度を維持しなければならない定速走行状態は，力行領域以外では現

表 G.1: 一般的なシミュレーションモデルにおける5列車状態とその遷移条件

状態		条件
遷移前	遷移後	
停車	力行	停車時間終了
力行	惰行	ノッチオフ後次駅到着までの推定時間が定時以内 (力行領域終了)
	定速走行	制限速度を超過
	ブレーキ	オフブレーキの条件（ブレーキオフ地点およびその地点での列車指定速度（駅停車の場合は停止目標位置およびゼロ速度）が与えられたとき，そこから所定の列車減速度を用いて逆算してちょうどよい地点に到達）
定速走行	惰行	力行領域にあり，かつノッチオフ後次駅到着までの推定時間が定時以内 (力行領域終了)
	惰行	惰行領域にあり，かつ勾配を含んだ走行抵抗が正
	力行	力行領域にあり，かつ制限速度を下回った
	ブレーキ	オフブレーキの条件
ブレーキ	定速走行	制限速度を超過し，勾配を含んだ走行抵抗が負
	惰行	ブレーキオフ速度に到達
	停車	速度が0に到達
惰行	ブレーキ	オフブレーキの条件
	力行	ノッチアップ（速度制限解除）地点に到達

れないようにした。力行領域は，発駅側から，上記の方法によって判定したノッチオフ地点までとしている。ノッチオフ位置決定の計算の際，列車は必要な場所では定速走行を必ず維持できるものとしている。この5状態についての状態遷移則は表 G.1に示した。

なお，状態遷移は遅延なしに行われるほか，ジャークコントロールは考慮していない。実際の車両では数秒オーダの遅延が存在するほか，ジャークコントロールを行うことになっているものが多い（特にインバータ制御のような新しい電車はなおさらだ）。これらの条件を無視することは駅間走行時分で2〜3秒早く走ったことと等価になる可能性が強いため，列車消費エネルギーの低下となって現れるものと考えられる。

このルールは《力行→惰行→ブレーキ→停止》のパターンを基本とし，それを拡張したものになっている。状態遷移のようすを，図 G.1・G.2のふたつのケースについてこれに当てはめて考えてみよう。

1. 一定の比較的低い速度制限がかかっている場合（図 G.1）。駅を出た (A) 列車は，B 点まで力行し，速度制限にかかって定速走行状態に移行する。駅間走行時分一定化の条件によってC で惰行に移り，速度がだんだんおちる。D 点で「オフブレーキの条件」を満たすのでブレーキ状態に移り E 点で停車。
2. 発駅および着駅に近いほど低くなる形状の速度制限がかかっている場合（図 G.2）。駅を出た (A) 列

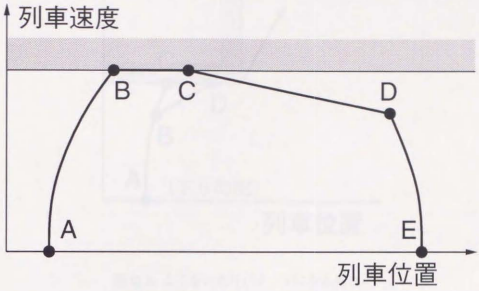


図 G.1: 走行パターン例 (1)

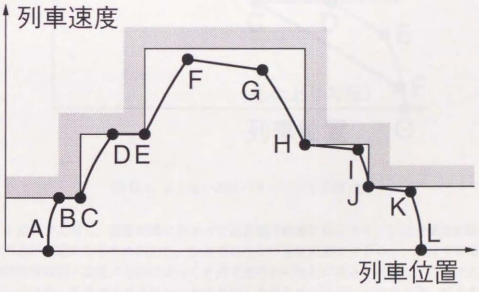


図 G.2: 走行パターン例 (2)

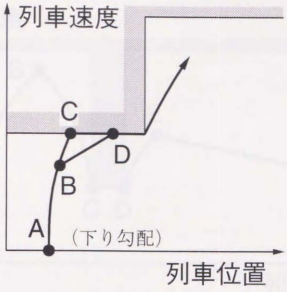


図 G.3: よくない走行パターンになる例 (1)

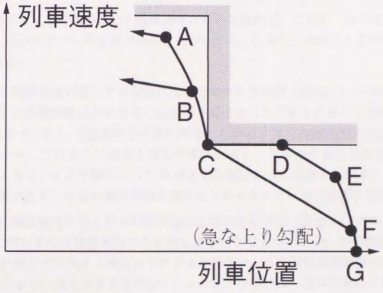


図 G.4: よくない走行パターンになる例 (2)

車は，B 点まで力行し，速度制限にかかって定速走行状態に移行する。C 点で速度制限が解除になり，まだ力行領域にあるため再力行。D 点でふたたび速度制限にかかるが，E 点で同様に再力行する。駅間走行時分一定化の条件によって F 点で力行から惰行に移ると同時に，ここで力行領域が終了する。その後，G 点で「オフブレーキの条件」を満たすためブレーキに入り，H 点までに速度制限より低い速度まで減速する。力行領域は F 点で終了しているので，H 点から惰行に入る（力行領域ならば定速走行となる）。I 点，J 点はそれぞれ G 点，H 点と同様。J 点から同様に K 点まで惰行で走って再びブレーキをかけ，L 点で停車する。

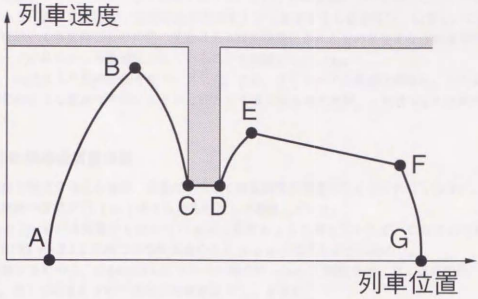


図 G.5: よくない走行パターンになる例 (3)

これらのケースのように「行儀のよい」線路条件のもとであれば，このルールで定性的に見てノーマルで，比較的最適にも近いパターンが生成できることがわかる。しかし，次のようなケースには問題が出てくることが考えられる。

- 駅間起点付近に制限速度が低く下り勾配の急な区間がある場合（図 G.3）。このルールでは，区間前半（力行領域）は速度制限にかかるまでは力行を続けることになるため，この例では A 点を発車した列車は C 点まで力行して定速走行状態に移る。しかし，下り勾配のため定速走行状態は弱いブレーキであるから，このように走ると力行の直後にブレーキをかけることになり，好ましくない。このパターンよりは，B 点で惰行に入り D 点まで勾配によって加速するパターンとし，速度制限が解除された地点の先でこの分の遅れを取り戻すほうがエネルギー的にも得になるはずである。
- 駅間終点付近に制限速度が低く登り勾配の急な区間がある場合（図 G.4）。このルールでは区間後半（惰行領域）に力行または速度維持のための定速走行ができないため，速度制限のため C 点まで減速した列車は F 点までに大きく速度を下げざるを得ない。この結果速度制限の前の区間でのブレーキ初速を A 点のように高くせざるを得ない。このような場合は，前の区間のブレーキ初速を B 点のように下げる代わりに C 点から D 点まで定速走行モードを入れて速度を維持したほうがエネルギー的に得になるケースがある。さらに，このような場合，低い速度での近似計算がうまくゆかないため駅間走行時分が精密に一定化できない場合も出てくる。
- 駅中間に制限速度が低い場所がある場合（図 G.5）。基本的に力行領域は区間前半に，惰行領域は区間後半になるわけだから，図のように前半ではオフブレーキ運転となる。E 点でノッチオフとなり力行領域が終了するまでは速度制限より低ければとにかく力行するルールのためにこうなってしまう。極端な場合，速度制限箇所の先ではまったく力行しないこともある。

これらは，主に列車力行エネルギーの増大として結果に現れるはずである。また，力行を行う場所が変化することから，回生ブレーキの効き具合にも影響がある。

これらの「悪い」条件に対応するには，駅間をいくつかの小区間にさらに細分すればよい。例えば図 G.5は A 点（発駅）から C 点までおよび C 点から G 点（着駅）までの 2 区間に分割し，それぞれの区間

を走行する時間をランカーブより与えるようにする。それぞれの区間についてはこのルールを適用することが可能であり，駅間走行時分一定化の基本的な考え方も変更する必要がない。ただし，この議論では，ランカーブにおいて各区間ごとの時間の配分はある程度最適化されたものを与えなければならない。すなわちランカーブがあらかじめ最適化されていることを前提にしている。

RTSS は，ver.2.2 より駅間分割をサポートした。また，ランカーブの最適化機能も，どのような形にするかおよびどのような最適化手法によるかは別として導入する考えだが，これは ver.3 以降でということになるだろう。

⟨G.3⟩ 回転部等価質量係数

列車の運動方程式を考える場合，質量だけでなく回転部等価質量も与えなければならない。通常回転部等価質量は車両の質量が $(1+w_r)$ 倍されたものとして考慮している。

RTSS ver.2.1.0 までは荷重分も含めて $(1+w_r)$ 倍するように考えていたが，これでは正確ではない。このため，RTSS ver.2.1.1 以降では空車質量のみを $(1+w_r)$ 倍するように改めた。

これを実現するために，`trainmotion` クラスに新たに `eqrot()` 関数を設けた。この関数の中では空車質量 W_{empty} および荷重を含めた現在の列車質量 W_{curr} を求め，

$$w_R = \frac{w_r \times W_{empty}}{W_{curr}} \tag{G.2}$$

で与えられる w_R を返すようにしてある。従来直接 w_r（変数名は `wrotate`）を参照していた場所はすべてこの `eqrot()` 関数の返り値を利用するように改めてある。

H

列車ダイヤのモデル

列車ダイヤのモデルは，列車の路線全体にわたる走らせ方を規定するものであり，diapattern クラスが一つのダイヤパターンを管理する。diapattern クラスは実は nextsta クラスの配列である。さらに nextsta クラスは gradcrv クラスの派生クラスとなっている。

列車ダイヤ，ないしは列車の「走り方」を表現する最小のデータ単位が次駅データで，それを表現するためのクラスが nextsta クラスである。全体を駅間ごとに切りとって管理し，列車がある駅からその次の駅（場合によっては駅間に特定の地点を設け，その地点までと限定することもある）までの走行に必要な情報を格納する。

勾配・曲線情報などのデータもこの次駅データクラス nextsta によって管理される。ファイルリード時には独立させ，別に gradcrv クラスの配列をつくることとし，データを記述するファイルも別にしてある。しかし，ファイルリードが終われば，nextsta クラスは必要な勾配・曲線情報をも自分で保持する。

次駅までの走行に関するデータがすべて nextsta クラスによって管理されるのだから，nextsta オブジェクトを必要数だけ並べれば列車の行路表が描かれたことになる。これが diapattern クラスである。場合によっては（というか高木が今までにシミュレーションしたほぼすべてのケース）このダイヤパターンの上に等時隔で複数列車を張り付けることもできるようになっている。

なお，ダイヤパターンデータとは別に，信号系のシミュレーションのためのモデルを構築中である。

〈H.1〉 勾配・曲線・速度制限データクラス gradcrv

勾配・曲線・制限速度は，路線を適当な区間に分けたとき，その区間内では一定である。となりの区間と連続である必要はない。すなわち，緩和曲線や縦曲線は考慮しない。そこで，路線をいくつもの区間に分けて，「区間始点」「区間終点」「勾配」「曲線」「制限速度」を一組としたデータを与えてやればよい。この一組のデータを gcvel データと呼ぶ。このことは (J.2)にて述べた通りだが，このデータ 1 つを格納するのが gcvel 構造体である。その定義は ‘nextsta.hh’ にあるが，変数 5 つだけの簡単なものだ。

次駅データとの関連から，すなわち次駅データはある駅から次の駅までの走行に関するデータをまとめて保持するものだから，この gcvel 構造体データも駅間ごとに与える。こうすると，gcvel データは駅間ごとにいくつかが 1 組となる。この 1 組を 1 つの gradcrv データと称することも (J.2) にて述べたが，この gradcrv データを格納するのが gradcrv クラスである。この定義も ‘nextsta.hh’ にあるが，zeroreftable テンプレートクラスの機能を使ったため定義自体は簡単である。

⟨H.2⟩ 次駅データクラス nextsta

列車の次駅までの走行パターンを決めるためのデータをセットにしたのが次駅データクラス nextsta である。このクラスは gradcrv からの派生として定義されている。

次駅データとは，次のデータの集まりである:

- 現駅（または，データの対象となる区間の開始点）の位置
- 次駅（または，データの対象となる区間の終了点）の位置
- 次駅（または，データの対象となる区間の終了点）での列車速度
- 次駅（または，データの対象となる区間の終了点）までの所要時分
- 次駅での停車時分
- ノッチオフ速度（現在のプログラムでは意味をなさないが，0.0 を書き込む。将来必要が出てきた場合は有効にする計画である）
- 列車走行方向（通常キロ程が増加する方向が (+)1，反対は −1）
- 饋電線番号
- 勾配・曲線・制限速度データ（gradcrv データ）
- 次駅データの種類を表す記号
- 次駅データ切替地点（駅以外の地点で切替を行う場合に必要）
- 次の「次駅データ」へのポインタ

特に強調すべきこととして，次駅データは勾配・曲線・速度制限データを持っていることだ。つまり，ファイルリードが終われば，nextsta クラスは必要な勾配・曲線情報をも自分で保持する（該当する gradcrv クラスへのポインタとしてではなく，自分で持っている）。この方法はメモリ容量をいささか無駄使いするが，メモリ容量制限の多かったパソコンでの使用は考えていないので，現在までのところ目だった問題にはなっていない。

⟨H.3⟩ ダイヤパターンデータクラス diapattern

⟨H.3.1⟩ 基本

次駅データは，ある駅から次の駅までの走行に要するデータを集めたものである。次駅までの走行に関するデータがすべて nextsta クラスによって管理されるのだから，すでに ⟨C.6⟩（127ページ）にて述べたように，nextsta オブジェクトを必要数だけ並べれば列車の行路表が描かれたことになる。これがダイヤパターンデータと呼ばれるものの基本であり，diapattern クラスが管理する。次駅データは，ダイヤパターンデータの中で配列として与えられている。なお，RTSS がシミュレート可能な列車ダイヤは，完全に周期的なダイヤであり，ダイヤパターンデータクラスの設計にもそのことが前提として考えられている。

ダイヤパターンデータとは，詳しくは

- 次駅データの配列
- このダイヤパターン 1 周期の所要時間
- このダイヤパターンの中で走る列車の数
- パターン倍数
- 列車群の位相（最初の列車が最初の駅を出発する時刻）
- 列車の初期化用変数群

などからなるデータである。

列車が次駅データの有効な範囲を越えた場合は，次の次駅データに移る。この移行のときに，列車位置・列車が電力の供給を受ける饋電線の番号・列車の走行方向などのデータを，ある程度自由に変えるこ

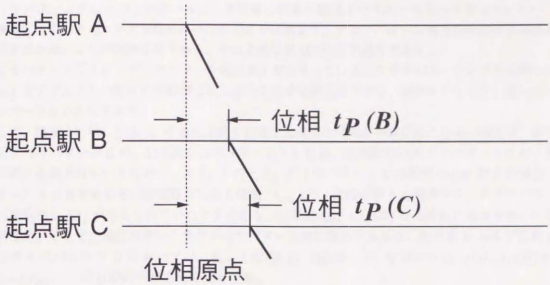


図 H.1: 列車位相の定義

とができる。また，次駅データ配列の最後のデータの次は，周期性の条件から，最初に戻る約束になっている。

ダイヤパターンデータにおいて与えられるダイヤは，必ずしも運用上「実現可能」である必要はない。例えば直線的な路線の起点から終点まで走る列車が5分に1本あるが，その他の列車は存在しないようなもの（この場合，起点および終点に無限大容量の車庫がないと実現しない）も，シミュレート可能である。

(H.3.2) 複数のダイヤパターンデータが存在する場合

ダイヤパターンデータは通常複数存在する。次駅データの配列 $\boldsymbol{A}$, $\boldsymbol{B}$ があり，それぞれ $A_1, A_2, \cdots, A_n$ および $B_1, B_2, \cdots, B_m$ という要素を持たせる。このとき，次駅データ $A_1, A_2, \cdots, A_n$ の順に走る列車群と，$B_1, B_2, \cdots, B_m$ の順に走る列車群とがそれぞれ存在しうる。例えば，JR 山手線のような完全な環状線にあっては，このような2つの次駅データが存在してはじめて，内・外回りの列車運動をデータとして与えることができる。また，昼間時の営団銀座線のように，普通の複線で普通列車1種類のみの平行ダイヤ，運転区間も1種類のみである場合は，ダイヤパターンデータは1つであるが，急行・普通の2種の列車があれば2つのダイヤパターンを用意しなければならないし，区間運転などが存在すればダイヤパターンはより多く必要になる。

このような場合，それぞれのダイヤパターンデータにおいて周期 T_{circ} を求めることを考える。このダイヤパターン1周期の所要時間（t_{whole} とする）およびこのダイヤパターンの中で走る列車の数（N_{cars} とする）のふたつのデータから，

$$T_{\mathrm{circ}} = \frac{t_{\mathrm{whole}}}{N_{\mathrm{cars}}} \tag{H.1}$$

となるはずだが，これがすべてのダイヤパターンに対して一定であるとは限らない。そこで，パターン倍数（N_{dptm} とする）という変数（整数）を用意しておき，すべてのパターンについて

$$T_{\mathrm{gcs}} \equiv \frac{t_{\mathrm{whole}}}{N_{\mathrm{cars}}} \cdot N_{\mathrm{dptm}} \tag{H.2}$$

で定義されるシミュレーションサイクル T_{gcs} が一定となるようにしてある。

(H.3.3) 列車位相

ダイヤパターンがひとつしかない場合，等間隔に列車の配置を行うルールだけを持っていれば十分だが，複数ダイヤパターンがある場合にはこれだけでは済まず，ダイヤパターン相互の時間的な関係を規定する必要がある。これが列車位相である。その定義は図 **H.1**に示す通りである。

ダイヤパターンごとに,「ダイヤパターン起点駅」が決まっている。ダイヤパターンクラスの持っている `nextsta` オブジェクトの配列の先頭がこれに当たるのが自然だろうから，RTSS でもそれに従ったインプリメンテーションとしてある。

そして，図 **H.1**に示した通り，その起点駅を列車が出発する時刻が列車位相となる。例えば，図で起点駅 A はダイヤパターン A の，起点駅 B はダイヤパターン B の，起点駅 C はダイヤパターン C の，それぞれ起点駅と定義されているものとしよう。このとき，ダイヤパターン A の位相 $t_P(A)$ が 0 の場合，ダイヤパターン A に属する列車は位相原点になる時刻（$t_{\mathrm{sim}} = 0$）に起点駅 A を発車する。ダイヤパターン B には列車位相 $t_P(B)$ が与えられていたとするなら，列車は $t_{\mathrm{sim}} = t_P(B)$ なる時刻に発車する。C についても同様だ。そして，複数列車が 1 つのダイヤパターン内に存在するなら，起点駅 D および位相 $t_P(D)$ を持つダイヤパターン D においては，T_{circ}（式 (**H.1**)，160ページ）を用いれば $t_P(D)$, $t_P(D) + T_{\mathrm{circ}}$, $t_P(D) + 2T_{\mathrm{circ}}, \ldots$ が D 駅の列車出発時刻となる。

⟨H.3.4⟩ `trainvar` コマンドでデータを「だます」

`trainvar` コマンドを用いて，データをいわば「だます」ような操作を行うことが可能だ。これは，周期的でないダイヤでのシミュレーションに使われる。

列車位相の定義は図 **H.1**で示した通りだ。ここで，周期的ダイヤを前提にして考えよう。同図で C 駅を位相 $t_P(C)$ にて出発する列車があるとしよう。ダイヤは周期的だというのが前提になっているから，C 駅での停車時分 t_{stop} が $t_P(C)$ より短い時間で設定されていたならば，列車は $t_P(C) - t_{\mathrm{stop}}$ なる時刻に C 駅に到着するように走行する初期設定となるはずである。

通常はこれで何ら困らないわけだが，周期的でないダイヤを取り扱おうとする場合に問題が生じる。周期的でないダイヤの場合，1 ダイヤパターンあたり 1 列車を張り付けるようにしてデータを記述するが，C 駅での停車時分 t_{stop} が $t_P(C)$ より短い時間で設定されていると困ることが起きる。すなわち，列車は時刻 0 から $t_P(C)$ までは C 駅で動かずにいて欲しいのに，このままでは時刻 0 から $t_P(C) - t_{\mathrm{stop}}$ までの間は C 駅に到着すべく動いてしまうからである。

本来は，ダイヤパターンデータ全体をきちんと書き直し，C 駅での停車時分を長くとるのが本当だろうが，いわばデータを「だます」やり方として，`initialdeparturetime` フラグを利用する方法がある。このフラグを利用すると，`trainvar` コマンドに駅出発時刻を記入することができる。こうすることにより，データを「だまし」，列車が時刻 0 から $t_P(C) - t_{\mathrm{stop}}$ までの間に動くことだけを防止することができる。

なお，`trainvar` コマンドで駅出発時刻を指定する場合，`phase` コマンドで指定する位相のほうはゼロと与えるべきである。

付録 IV

RTSS マニュアル・使い方

I

コマンド行の引数

rtss プログラムの立ち上げは，次のようにすればよい。

```
rtss [options]
```

コマンドオプションは次のいずれか（複数指定可）である。

- -g勾配・曲線・制限速度データファイル名
- -n次駅データファイル名
- -s変電所特性データファイル名
- -p列車性能データファイル名
- -f饋電線データファイル名
- -t列車初期位置データファイル名
- -r出力データファイル名
- -dディレクトリ名

-d オプションは，それまでに指定したファイル名すべてに，同じディレクトリ名を付加するオプションである。これらのオプションは一つ一つ空白で区切る。しかし，一つのオプションは空白を空けずに続けて書く必要がある。つまり，-g, -n, ... などのあとには空白を入れてはいけない。

なお，これらのデータファイル名は，デフォルトではプレゼントワーキングディレクトリの次のような名称のファイルである:

勾配・曲線・制限速度データファイル ‘gradcrv.dat’（G ファイルと略称）

次駅データファイル ‘nextsta.dat’（N ファイルと略称）

変電所特性データファイル ‘ssdata.dat’（S ファイルと略称）

列車性能データファイル ‘pbdata.dat’（P ファイルと略称）

饋電線データファイル ‘feeder.dat’（F ファイルと略称）

列車初期位置データファイル ‘trdist.dat’（T ファイルと略称）

出力データファイル ‘result.dat’（R ファイルと略称）

なお，-t オプションで指定する列車初期位置データファイルは，現在無効となっている。

これらのオプションはかなり複雑で，与えるのも面倒だろう。そこで，適切なシェルスクリプトのようなものを書いてやることをお勧めしたい。

J

データファイルの仕様と書き方

鐵電システムのシミュレーションに必要なすべてのデータは，データファイルによって与えられなければならない。このファイルは5つにわかれている。それぞれについて説明しよう。

〈J.1〉 基本

データファイルのデータ行は，すべてこまんどデータ（コマンドの種類によっては省略可能だったり個数可変だったりする）次のような構成となっている:

```
commandname [data...]
```

例えば，勾配・曲線・制限速度データファイルに出てくる gcvel コマンドは次のような形をしている。

```
gcvel  9
 16.382  16.458  -5.00    0.0  80.0
 16.458  16.480  -5.00  622.0  80.0
 16.480  16.578 -35.00  622.0  80.0
 16.578  16.728 -35.00    0.0  80.0
 16.728  17.100 -33.00  450.0  80.0
 17.100  17.160  -2.00  450.0  80.0
 17.160  17.220  -2.00  355.0  80.0
 17.220  17.316  -2.00  355.0  40.0
 17.316  17.382   5.00  355.0  40.0
```

このうち，最初のデータとなる 9 は整数で，その後のデータ（5つの倍精度実数で1組）の組数を表す。データは先頭から読まれ，最初の整数データによって全体のデータの個数を可変としている。この場合は，最初のデータが9であったから，これに45個の実数データが続くことになる。

データの読み込みは，C 言語の標準的な入出力関数のひとつ printf() を利用して記述してある。このため，ひとつながりの数字の間でなければ，どこに改行コードやスペース文字，タブ文字などを挿入しても大丈夫だ。従って，上のように適当な場所でデータを改行し，読みやすくすることができる。

また，ファイルの先頭に「ふらぐが書き込まれることがある。フラグはコマンドと同一の書式を持つが，データファイルにおいて最初のコマンドが出現するまでの間に書かれなければならない。もちろん，コマンドだからといってどこに書いてもよいわけではなく，順序などがある程度決まっている。また，多くの場合フラグは「書かなければならない」ものではなく，コマンドは「書かなければならない」という

違いがある。しかし，コマンドの中にも省略可能なものがあるし，書かれたフラグの種類によってコマンドが省略可能になったり，データの個数や種類が変更になったりすることもある。

また，コメント行はデータファイルの任意の場所に挿入することが可能である。コメントは UNIX の C シェルスクリプトなどと同じく，# 文字から行末までをコメントとみなす約束になっている。例えば，上の gcvel コマンドの場合，

```
gcvel  9
 16.382  16.458  -5.00    0.0  80.0
 16.458  16.480  -5.00  622.0  80.0
 16.480  16.578 -35.00  622.0  80.0 #This is a comment
 16.578  16.728 -35.00    0.0  80.0
 16.728  17.100 -33.00  450.0  80.0
 17.100  17.160  -2.00  450.0  80.0
 17.160  17.220  -2.00  355.0  80.0
 17.220  17.316  -2.00  355.0  40.0
 17.316  17.382   5.00  355.0  40.0
```

のように挿入することも可能だ。なお，漢字コードへの対応についてはきちんと行っていない。EUC 漢字コードならば大丈夫であると思われるが，確認はしていないので，at your own risk にて行うこと。

なお，例えば G ファイル（勾配・曲線・制限速度データファイル）にあるコマンドであれば，G コマンドと略称することがある。N コマンド, P コマンド, S コマンド, F コマンド なども同様。G フラグ, N フラグ, P フラグ, S フラグ, F フラグ なども同様である。

⟨J.2⟩ 勾配・曲線・制限速度データファイル（G ファイル）

勾配・曲線・制限速度は，路線を適当な区間に分けたとき，その区間内では一定である。となりの区間と連続である必要はない。すなわち，緩和曲線や縦曲線は考慮しない。そこで，路線をいくつもの区間に分けて，「区間始点」「区間終点」「勾配」「曲線」「制限速度」を一組としたデータを与えてやればよい。この一組のデータを gcvel データ と呼ぶことにする。

次駅データとの関連から，gcvel データは駅間ごとに与える。こうすると，gcvel データは駅間ごとにいくつかが 1 組となる。この 1 組を 1 つの GRADCRV で一た称する。場合によって，上り下りで別な gradcrv データを与えることも，同一の gradcrv データを共用することも，どちらも可能である。

⟨J.2.1⟩ フラグ

このデータファイルに書き込めるフラグは，現在のところ次の 1 種類だけが認められる。

- vlimitmikomi フラグ: 速度制限に見込む余裕。データは倍精度実数の 1 つだけで，単位は km/h とする。このフラグを与えると，速度制限がこの分だけ低くなったのと同じ振舞いをする。データファイルそのものを修正しても同じ効果は得られる。

⟨J.2.2⟩ コマンド

このデータファイルには次のようなコマンドを，ここで出てきた順に書き込む必要がある。

1. number コマンド: gradcrv データの数。

   ```
   number 29
   ```

   のように書く。データファイルにはこれ以降 gcvel コマンドがここで指定した回数だけ現れなければならない。

2. gcvel コマンド: 可変長データを持つコマンドであり，このコマンド 1 つで gradcrv データ 1 つを表現する。

最初のデータは整数で，この gradcrv データの中に gcvel データがいくつ含まれるかを示す。次いで，(この整数×5) 個の倍精度実数データが与えられ，必要な数の gcvel データを表現する。

1 つの gradcrv データにつきこのコマンドが 1 つずつ対応するはずであるから，最初の number コマンドで指定された数だけの gcvel コマンドが現われなければならない。

gcvel データを表す 5 つの倍精度実数は順に「区間始点 (km)」「区間終点 (km)」「勾配 (‰)」「曲線 (m, ゼロは直線)」「制限速度 (km/h, ゼロは制限なし)」を意味する。「区間始点」は必ず「区間終点」より小さな数字でなければならない。また，勾配は「区間始点」から「区間終点」に向かう列車に対する上り勾配で表示する。また，ひとつの gcvel コマンド中において，となりあうふたつの gcvel データのうち，データファイルの先頭側にあるものの「区間終点」と末尾側にあるものの「区間始点」は同一の数字でなければならない。

```
gcvel 2
0.000 0.400 0.00 0.0 0.0 0.400 0.800 -3.00 0.0 0.0
```

のように書く。

データファイルでは通常複数の gcvel コマンドを書くことになるが，これは RTSS に対して複数の gradcrv データを入力したことと同じである。このとき，gradcrv データには内部で自動的に番号が付けられる。この番号は，最初の gcvel コマンドに対応するものがゼロ，次が 1，次いで 2，……のようになる。最初がゼロである理由は C 言語を利用している人ならばわかると思う。なお，この番号はあとで次駅データを作成する際に必要となるものなのだが，データファイルを見ていちいち数えるのではわかりにくいので，対応する番号をデータファイル上にコメント行として記入しておくとわかりやすいだろう。

⟨J.2.3⟩ サンプル

データファイルの例を次に示す。本来 gcvel コマンドが 29 個なければならないが，長いので省略している。

```
#########
# Flags #
#########

vlimitmikomi    5.0

##########################
# Beginning of commands #
##########################

number 29

# (0) Tokyo - Yurakucho
gcvel 3
    0.000  0.400  0.00  0.0  0.0
    0.400  0.580  0.00  400.0  0.0
    0.580  0.780  0.00  0.0  0.0

# (1) Yurakucho - Shimbashi
gcvel  8
    0.780  1.000  0.00  0.0  0.0
    1.000  1.190  0.00  600.0  0.0
```

```
    1.190  1.490  0.00  0.0  0.0
    1.490  1.550  5.00  0.0  0.0
    1.550  1.660  5.00  400.0  0.0
    1.660  1.750  0.00  400.0  0.0
    1.750  1.870  0.00  520.0  0.0
    1.870  1.890  0.00  0.0  0.0

# (2) Shimbashi - Hamamatsu-cho
gcvel  12
    1.890  1.960  0.00  0.0  0.0
    1.960  2.000  0.00  0.0  0.0
    2.000  2.060  5.00  460.0  0.0
    2.060  2.070  0.00  460.0  0.0
    2.070  2.120  -5.00  0.0  0.0
    2.120  2.160  0.00  0.0  0.0
    2.160  2.210  -11.70  0.0  0.0
    2.210  2.310  -10.00  0.0  0.0
    2.310  2.610  -9.10  0.0  0.0
    2.610  2.680  0.00  0.0  0.0
    2.680  2.760  -5.00  0.0  0.0
    2.760  3.050  0.30  0.0  0.0

#                          ……以下略
```

⟨J.3⟩ 次駅データファイル（N ファイル）

次駅データファイルは，列車の次駅までの走行に関するデータ，およびダイヤパターンデータを書き込むファイルである。

⟨H.2⟩（159ページ）にて述べたことの繰り返しとなるが，次駅データとは，次のデータの集まりである:

- 現駅（または，データの対象となる区間の開始点）の位置
- 次駅（または，データの対象となる区間の終了点）の位置
- 次駅（または，データの対象となる区間の終了点）での列車速度
- 次駅（または，データの対象となる区間の終了点）までの所要時分
- 次駅での停車時分
- ノッチオフ速度（現在のプログラムでは意味をなさないが，0.0 を書き込む。将来必要が出てきた場合は有効にする計画である）
- 列車走行方向（通常キロ程が増加する方向が (+)1，反対は −1）
- 饋電線番号
- 勾配・曲線・制限速度データ（gradcrv データ）
- 次駅データの種類を表す記号
- 次駅データ切替地点（駅以外の地点で切替を行う場合に必要）
- 次の「次駅データ」へのポインタ（データとして与える必要はない）

ダイヤパターンは常に周期をなしており，次駅データはダイヤパターンデータの中で配列として与えられている。ダイヤパターンデータとは，

- 次駅データの配列
- このダイヤパターン 1 周期の所要時間（twhole）
- このダイヤパターンの中で走る列車の数（cars）
- パターン倍数（dptm）

- 列車群の位相（最初の列車が最初の駅を出発する時刻）
- 列車の初期化用変数群

などからなるデータである。

列車が次駅データの有効な範囲を越えた場合は，次の次駅データに移る。この移行のときに，列車位置・列車が電力の供給を受ける饋電線の番号・列車の走行方向などのデータを，ある程度自由に変えることができる。また，次駅データ配列の最後のデータの次は，最初に戻る約束になっている。

シミュレート可能な列車ダイヤは，完全に周期的なダイヤでなければならない。

ダイヤパターンデータにおいて与えられるダイヤは，必ずしも運用上「実現可能」である必要はない。例えば直線的な路線の起点から終点まで走る列車が5分に1本あるが，その他の列車は存在しないようなもの（この場合，起点および終点に無限大容量の車庫がないと実現しない）も，シミュレート可能である。

また，ダイヤパターンデータは通常複数存在する。次駅データの配列 $\boldsymbol{A}, \boldsymbol{B}$ があり，それぞれ $A_1, A_2, \cdots, A_n$ および $B_1, B_2, \cdots, B_m$ という要素を持たせる。このとき，次駅データ $A_1, A_2, \cdots, A_n$ の順に走る列車群と，$B_1, B_2, \cdots, B_m$ の順に走る列車群とがそれぞれ存在しうる。例えば，JR 山手線のような完全な環状線にあっては，このような2つの次駅データが存在してはじめて，内・外回りの列車運動をデータとして与えることができる。また，昼間時の営団銀座線のように，普通の複線で普通列車1種類のみの平行ダイヤ，運転区間も1種類のみである場合は，ダイヤパターンデータは1つであるが，急行・普通の2種の列車があれば2つのダイヤパターンを用意しなければならないし，区間運転などが存在すればダイヤパターンはより多く必要になる。

このような場合，それぞれのダイヤパターンデータにおいて周期を求めると，このダイヤパターン1周期の所要時間（twhole）およびこのダイヤパターンの中で走る列車の数（cars）のふたつのデータから，

```
twhole / cars
```

となるはずだが，これがすべてのダイヤパターンに対して一定であるとは限らない。そこで，パターン倍数（dptm）という変数を用意しておき，

```
twhole / cars * dptm
```

が一定となるようにしてある。dptm は整数だ。

(J.3.1) フラグ

このデータファイルに書き込めるフラグは，現在のところ次のものが認められる。

- conges_station フラグ: 混雑率を駅間毎に変更できるようにするフラグで，データはない。このフラグを与えると，congestion コマンドを各次駅データ毎に指定でき，混雑率が駅間毎に変更できるようになる。
- ontprec_delay_set フラグ: 定時ノッチオフ位置決定ルーチンを使用しているにもかかわらず列車があまりに進んだり遅れたりするばあい，フリーランの回数を増やした上で定時ノッチオフ位置決定ルーチンのうち近似計算のルーチンの使用をその次駅データについて抑制する。このフラグで，この操作を行う列車の進み遅れの閾値を設定することができる。データはこの閾値が倍精度実数型でひとつだけ指定可能（単位・秒）である。このフラグがない場合，デフォルト値の 0.6 が採用される。
- ontprec フラグ: 定時ノッチオフ位置決定ルーチンのうち近似計算のルーチンの使用をすべてのケースで抑制するフラグである。このフラグを指定すると，計算時間がかかるようになるが，フリーランの回数がゼロの場合に有効だろう。データはない。

- initialdeparturetime フラグ: trainvar コマンドで tdept，すなわち列車の出発時間を指定できるようにするフラグである。データはない。この機能は，いわば RTSS を「だます」ために利用することができる（(**H**.3.4), 161ページ）。
- station_object_valid フラグ: station_obj オブジェクトを有効にするフラグである。このオブジェクトは現在設計中である。

⟨J.3.2⟩ コマンド

データファイルには次のようなコマンドを，ここで出てきた順に書き込む必要がある。ただし，文中に明示してあるコマンドについては，順序が問われないこともある。

1. patterns コマンド: ダイヤパターンデータの数。データの先頭に 1 つ書かれる nextsta コマンド，およびそれに続く cycletime, phase, cars, patterncirc, trainvar の各コマンド群，および nextsta コマンドで指定した数の次駅データ指定コマンド群を合わせたものが，1 つのダイヤパターンデータとなる。データファイルには，ダイヤパターンデータがここで指定した回数だけ現れなければならない。

   ```
   patterns 2
   ```

   のように書く。

2. ダイヤパターンデータ指定コマンド群: nextsta, cycletime, phase, cars, patterncirc, trainvar の各コマンド群，および nextsta コマンドで指定した数の次駅データ指定コマンド群を合わせたものだ。これらのうち cycletime, phase, cars, patterncirc, trainvar の各コマンドは，ダイヤパターンデータの中の最初の次駅データが現われる前に，どの順で現われてもよい。1 つのコマンドが複数回現われてはいけない。また，どのコマンドも省略できない。

   2A. nextsta コマンド: 1 組のダイヤパターンデータの先頭には，必ずこのコマンドを置かなければならない。一つのダイヤパターンデータの中に，次駅データがいくつあるかを示す。

   ```
   nextsta 29
   ```

   のように書く。

   2B. cycletime コマンド: ダイヤパターンデータの 1 周当たりの時間を，秒単位で与える。

   ```
   cycletime 3600.0
   ```

   のように書く。

   2C. phase コマンド: パターンの位相ずれを，秒単位で与える。

   ```
   phase 0.0
   ```

   のように書く。

   例えば，山手線のダイヤを内・外回りとも東京駅発の次駅データ配列として与え，内回りのパターンに対して phase 0.0，外回りのパターンに対して phase 60.0 と与えると，内回り電車が東京駅を時刻 0.0 秒に出発すると外回り電車は東京駅を 60.0 秒に出発するパターンとなる。

   2D. cars コマンド: ダイヤパターンデータの中を走っている列車数を与える。

   ```
   cars 25
   ```

   のように書く。

   車両は等間隔にダイヤパターン内に配置される。従って，

   ```
   cycletime 3600.0
   cars 25
   ```

のように指定すると，列車の間隔は

$$\frac{3600[\mathrm{sec}]}{25\text{本}} = 144[\mathrm{sec}]$$

となる。これが1パターン当たりの最低周期である。

2E. `patterncirc` コマンド: パターン倍数。各ダイヤパターンが完全に周期的であるとしても，1パターンあたりの最低周期が一致するとはかぎらない。そこで，`cycletime`, `cars`, `patterncirc` の各コマンドにて指定した値をそれぞれ c, n, p とするとき

$$\frac{c}{n} \cdot p = \mathrm{const.} \tag{J.1}$$

となるような整数 p を与える。

```
patterncirc 1
```

のように書く。

2F. `trainvar` コマンド: 列車データの初期値を与えるコマンドである。

```
trainvar 0.0 0.0 0 0.0
```

のように書く。

`trainvar` の後にはかならず4つのデータを書き込む。順に

- 列車初期位置 [km]
- 列車初期速度 [km/h]
- 列車初期状態，整数で与える。この整数は，プログラム中の列挙データ型 `car_stat` において，該当する整数または文字列を与える。
  - `Power` または 0 (力行)
  - `Brake` または 1 (ブレーキ)
  - `Coast` または 2 (惰行)
  - `Stn_stop` または 3 (駅停車)
  - `Const_vel` または 4 (定速走行)
  - `Regenerate` または 5 (回生のみ)
- 列車駅出発時初期遅れ [s]

プログラムは列車の初期配置を自動的に決定する関数を持っている。これらのデータは，この初期配置を決定する際に列車に初期の状態変数として与えられるデータである。

なお，`initialdeparturetime` フラグがある場合，`trainvar` コマンドに与えるデータは5つとし，順に

- 列車初期位置 (上記と同じ)
- 列車初期速度 (上記と同じ)
- 列車初期状態 (上記と同じ)
- 列車出発時刻 [s]
- 列車駅出発時初期遅れ (上記と同じ)

を書き込む。当然ながら，列車出発時刻の指定は列車初期状態が「駅停車」でなければ意味を持たない。この時刻は当該駅での所定の停車時分と無関係に設定できるので，結果的にデータをだます機能を持つ（〈H.3.4), 161ページ）。

2G. 次駅データ群: `startpoint`, `endpoint`, `endvelocity`, `startstoptime`, `stationstop`, `notchoff`, `direc`, `nfline`, `nextsta_pattern`, `gradcrvnumber`, `gradcrv`, `congestion` および `nextnxx` のコマンド群（ただし `nextnxx` は省略可能，`congestion` は `conges_station` フラグを書かな

かった場合は記入禁止，また gradcrv は 1 つとは限らない）の 1 まとまりをもって，1 つの次駅データを表現する。この次駅データを，nextsta コマンドで指定された数だけ配置する。

なお，これらのコマンドのうち，nextnxx だけが省略可能である。これらのコマンドはどの順で出てきてもよい。ただし，gradcrvnumber コマンドの直後には，コマンドで指示された数だけの gradcrv コマンドが出てくる必要があるほか，nextnxx コマンドの配置方法は注意を要する。

(i) startpoint コマンド: 次駅データの開始地点。通常は，発駅のある地点を指す。

```
startpoint 0.0
```

のように書く。

(ii) endpoint コマンド: 次駅データの終了地点。通常は，着駅のある地点を指す。

```
endpoint 0.78
```

のように書く。

(iii) endvelocity コマンド: 次駅データの終了地点での速度。通常は，終了地点は着駅のある地点であるから，0.0 を与えればよい。

```
endvelocity 0.0
```

のように書く。

(iv) startstoptime コマンド: 次駅データの開始地点から終了地点までの所要時分（単位秒）。

```
startstoptime 70.0
```

のように書く。

(v) stationstop コマンド: 次駅での停車時分。

```
startstoptime 20.0
```

のように書く。

(vi) notchoff コマンド: ノッチオフ速度を入れるが，現状のプログラムでは完全なダミーパラメータとなっている。

```
notchoff 0.0
```

と書く。

(vii) direc コマンド: 列車走行方向。キロ程の数字が増加する方向に走るとき (+)1，その逆は −1 と与える。

```
direc -1
```

のように書く。

(viii) nfline コマンド: 饋電線番号。列車に電力を供給する饋電線の番号を整数として与える。

```
nfline 0
```

のように書く。饋電線の数を n とすると，ここに書きうる数字は $0, 1, \cdots, n-1$ である。範囲外の数字を書き込んではいけない。なお，饋電線数は饋電線データファイル 'feeder.dat' のなかで，feeders コマンドによって指定する。

(ix) nextsta_pattern コマンド: 次駅データの種類を与える。いまのところ Station タイプ，つまり次駅データの終点が停車駅である場合にしかプログラムが対応していない。

```
nextsta_pattern Station
```

のように書く。

(x) gradcrvnumber コマンド: gradcrv コマンドが現われる回数を整数で記入する。このコマンドに続けて gradcrv コマンドをその回数だけ書き込まなければならない。

```
gradcrvnumber 3
```

のように書く。いうまでもなく，gradcrvnumber 0 などという指定はできない。

(xi) congestion コマンド: この次駅データが有効な範囲内での混雑率を指定する。混雑率は % ではなく，比で記入することに注意されたい。すなわち，いわゆる 200% 乗車の場合なら

```
congestion 2.0
```

のように書く。

なお，conges_station フラグを記入しなかった場合，このコマンドを記入するとエラーとなる。

(xii) gradcrv コマンド: 勾配・曲線・制限速度データ（gradcrv データ）を選択し，場合によっては適当に変換して，次駅データの中に取り込む。仮に複数の gradcrv コマンドが 1 つの次駅データに与えられているならば，選択された gradcrv データは連結されて 1 つの配列とされる。

```
gradcrv 3 0
```

または

```
gradcrv 3 1 2.5
```

のように書く。

gradcrv コマンドの後には 2 ないし 3 個の数字を記入する。まず，最初の数字は整数であって，gradcrv データの番号を記入する。この番号は，〈J.2〉で述べた勾配・曲線・制限速度データファイルにおいて，最初に現われる gradcrv データを 0 番として，後に出てくるものを 1, 2, ⋯, として定める。

次の数字も整数である。これが 0 の場合は 3 つめの数字は記入する必要はなく，gradcrv データはデータファイルのまま代入される。それ以外の場合は 3 つめの数字が必要である。

まず，2 つめの整数が 0 より大である場合は，新たな距離原点を 3 つめの数字の位置にとり，距離を書き改める。例えば，ある gradcrv データに含まれる gcvel データにおいて，始点が 5.0, 終点が 6.5 であるとしよう。この gcvel データを含む gradcrv データを

```
gradcrv 3 1 2.5
```

なるコマンドで呼び出したとすれば，新しい距離原点を 2.5 にとるので，始点の 5.0 は 2.5 に，終点の 6.5 は 4.0 に，それぞれ修正される。

```
gradcrv 3 1 6.5
```

ならば，始点は −1.5 に，終点は 0.0 になる。このようなルールですべての gcvel データが修正され，新しい gradcrv データが生成される。

2 つめの整数が 0 より小である場合は，gcvel 配列の順序を逆にする。同時に始点と終点の位置も入れ換えられる。さらに，距離原点は 3 つめの数字にとり，距離の符号も反転させる。例えば，上と同じように gcvel データにおいて始点が 5.0, 終点が 6.5 であるときに，この gcvel データを含む gradcrv データを

```
gradcrv 3 -1 2.5
```

なるコマンドで呼び出したとすれば，始点・終点を入れ換えて，新しい距離原点を 2.5 にとったうえ符号を反転させるので，始点は −4.0 に，終点は−2.5 に，それぞれ修正される。

```
gradcrv 3 -1 6.5
```

ならば，同様にして始点は 0.0 に，終点は 1.5 になる。このようなルールですべての gcvel データが修正され，入れ換えられて，新しい gradcrv データが生成される。

複数の gradcrv コマンドがあるときは，これらの変換の結果出てくる gcvel データの配列が連続していなければならない。つまり，ある gradcrv コマンドで生成された gradcrv データの配列先頭にある gcvel データの区間始点は，直前の gradcrv コマンドで生成された gradcrv データの中の配列末尾にある gcvel データの区間終点と一致していなければならない。

(xiii) nextnxx コマンド: このコマンドのみ省略可能である。このコマンドは，次駅データに与えられている区間の中間で，饋電線番号を変更するときにのみ用いる。

```
nextnxx 53.32
```

のように書く。この位置を列車が過ぎたならば，次駅データを次のものに移行させる。次の次駅データは，nfline コマンド以外の必須コマンドとしてすべて同一の値を書き込まなければならない。

なお，startpoint, endpoint, endvelocity, startstoptime, stationstop, notchoff, direc, nfline, nextsta_pattern, congestion, gradcrvnumber の各コマンドはすべて省略不可能である（ただし，conges_station フラグがないならば congestion コマンドは記入禁止）。これらのコマンドはどの順で出てきてもよい（ただし，gradcrvnumber コマンドの直後には，コマンドで指示された数だけの gradcrv コマンドが出てくる必要がある）。これらが 1 つでも欠けていたり，複数あったりするとエラーとなる。gradcrvnumber 0 が許されていないから，実質的には gradcrv コマンドも必須である。

しかし，nextnxx コマンドは省略可能であるため，次のような問題を生ずる。すなわち，プログラムは必須コマンドがすべて出つくしてしまうと，次駅データの記述が終わったと考えて次に進んでしまうのである。そこで

```
startpoint 55.45
endpoint 48.44
nextnxx 53.32
endvelocity 0.00
startstoptime 265.0
stationstop 20.0
notchoff 0.0
direc -1
nfline 9
gradcrvnumber 1
gradcrv 19 0
nextsta_pattern Station
```

のように，必須コマンドの中間にいれないと正しく認識されない。

```
startpoint 55.45
endpoint 48.44
endvelocity 0.00
startstoptime 265.0
stationstop 20.0
notchoff 0.0
direc -1
nfline 9
```

```
gradcrvnumber 1
gradcrv 19 0
nextsta_pattern Station
nextnxx 53.32
```

とすれば，この nextnxx コマンドは，これらの一連の startpoint ～ nextsta_pattern コマンド群で示される次駅データに対するコマンドではなく，この次に記述される次駅データに対するコマンドとして理解されてしまうので，注意を要する。

データファイルの例を次に示す。本来次駅データがダイヤパターンごとに 29 個なければならないが，長いので省略している。

```
patterns 2

nextsta 29
cycletime 3600.0
phase 0.0
cars 25
patterncirc 1
trainvar 0.0  0.0  0  0.0

startpoint 0.000
endpoint 0.780
endvelocity 0.00
startstoptime 70.0
stationstop 20.0
notchoff 0.0
direc 1
nfline 0
nextsta_pattern Station
gradcrvnumber 1
gradcrv 0 0

startpoint 0.780
endpoint 1.890
endvelocity 0.00
startstoptime 90.0
stationstop 30.0
notchoff 0.0
direc 1
nfline 0
nextsta_pattern Station
gradcrvnumber 1
gradcrv 1 0

#                                   ……以下，次駅データ省略

nextsta 29

cycletime 3600.0
phase 60.0
cars 25
patterncirc 1
trainvar 34.475  0.0  0  0.0

startpoint 34.475
```

```
endpoint 33.225
endvelocity 0.00
startstoptime 100.0
stationstop 20.0
notchoff 0.0
direc -1
nfline 1
nextsta_pattern Station
gradcrvnumber 1
gradcrv 28 0

#                                   ……以下略
```

⟨J.4⟩ 変電所特性データファイル（S ファイル）

変電所特性データファイルは，各変電所の特性を与えるファイルである。変電所の特性は，変電所ごとに設定することができる。変電所の特性は V–I 平面上で折れ線で与えられるものに限る。

⟨J.4.1⟩ フラグ

このデータファイルには，現在のところフラグの書き込みは認められていない。

⟨J.4.2⟩ コマンド

データファイルには次の項目を，この順に記入する。順番を変えてはいけない。

1. substations コマンド: 変電所数を整数で示す。

   ```
   substations 11
   ```

   のように書く。データファイルには，subchar コマンドがここで指定した回数だけ現われなければならない。

2. subchar コマンド: 可変長データを持つコマンドであり，このコマンド 1 つで 1 変電所分の特性データを表現する。

   最初のデータは整数である。変電所の特性は折れ線で表示されるが，その折れ線の端点の数を整数で示す。このあとに，端点データがここで指定した数だけ続いて現われなければならない。当然ながら，ここで 1 以下の数を指定しても意味がない。

   端点データは 1 点あたり 3 つの倍精度実数値で表現され，折れ線の端点の電圧・電流と，媒介変数の値を示す。媒介変数 0 のときは例えば無負荷時送出電圧となるように考えるべきであろう。また，媒介変数の値のとり方を変にすると計算失敗の原因となることもある。データは，媒介変数，電圧，電流の順に，実数を 3 つずつ書き込む。例えば

   ```
   subchar 3
   -200.0 1800.0 0.0
   0.0 1600.0 0.0
   20000.0 1100.0 20000.0
   ```

   のように書く。

   なお，複数の端点データがあるとき，各行の媒介変数は，行が下に行くにしたがって単調に増加，ないしは減少しなければならない。また，媒介変数の変域は，最初の端点データにある値と最後の端点データにある値との間に制限される。したがって，はじめと終わりの端点データは実質的に最高電圧・最低電圧などを決定する点になる。この点は，シミュレーションの目的にかなうように，または計算がきちんと終了するように，適当に与えればよい。

3. ratedcurrent コマンド: 変電所の定格電流値（単位 A），I_{Sl}, I_{Sc} および I_{Sm} の値（単位 p.u.）を，実数で与える。

```
ratedcurrent 4800.0 2.0 2.5 3.0
```

のように書く。通常のシミュレーションではあまり深い意味は持たないが，指定はしなければならない。

⟨J.4.3⟩ サンプル

データファイルの例を次に示す。

```
substations 3

# (0) A SS
subchar 5
        -43300.0        1800.0          -3333.333334
        -23000.0        1551.0          -3333.333334
        -20000.0        1551.0          -1.0
        0.0             1521.0          0.0
        20000.0         1091.0          10000.0
ratedcurrent 2000.0 2.0 2.5 3.0

# (1) B SS
subchar 3
        -20000.0        1800.0          -1.0
        0.0             1521.0          0.0
        20000.0         1091.0          10000.0
ratedcurrent 2000.0 2.0 2.5 3.0

# (2) Dummy SS (con. 0)
subchar 2
        -200.0          1800.0          0.0
        1200.0          400.0           0.0
ratedcurrent 0.0 2.0 2.5 3.0
```

⟨J.5⟩ 列車性能データファイル（P ファイル）

列車の性能は，入出力インタフェースの問題から現状では全列車同一のものしか与えられない。また，インバータ制御車両が前提となっている。

⟨J.5.1⟩ フラグ

このデータファイルに書き込めるフラグは，現在のところ次の 6 種類が認められる。すべてデータのないフラグである。これらのフラグは 2 度以上書き込まないようにすること。2 度書き込むと，トグルになっていて元に戻ることになり，フラグの指定を無効としてしまう。

1. regendantei フラグ: 回生失効をシミュレートする。このフラグを与えると，回生絞り込みと回生失効を区別してシミュレーションを行う。
2. differentregend フラグ: 回生絞り込み終了電圧と系の最高電圧が異なるデータを取り扱う。このフラグを与えると，回生絞り込み終了電圧を最高電圧とは別にデータとして与えることができるようになる。
3. shiboriparallel フラグ: 回生絞り込み特性を (1)（(**F.3**)，図 7.22参照）とする。このフラグを与えると回生率は悪くなることだろう。
4. auxpower フラグ: 補機負荷をパワー (kW) で与えるようにする。デフォルトは電流値 (A) である。

5. torque_shibori フラグ: 主回路回生電流（(F.3)，図 7.23参照）のかわりにモータ電流を回生絞り込みの基準に用いる。このフラグを与えると回生絞り込みが必要でない場合にも行われることになる。
6. anteimotor フラグ: 回生失効の判定基準にもモータ電流を用いる。このフラグを与えると，新たにモータ電流をデータとして与える必要が出てくる。

⟨J.5.2⟩ コマンド

データファイルには，この順に次のコマンドをすべて書き込まなければならない。

1. cs_command コマンド: 列車で行う制御の種類を示す。None（従来方式・制御なし），Ec_auto（フルノッチ比制御あり），Ea_cvauto0（定数調整制御あり）のなかから選んで，文字列で記入する。
2. powerdata コマンド: 列車の力行性能を記述するコマンド群の始まりを指定するコマンドである。このコマンド群は以下のようなものだ。
   - 2A. ec コマンド: 性能を表す数字は，電車線電圧がこのコマンドで指定される電圧値であるときのものを記入する。
   - 2B. zerocur コマンド: 0km/h フルノッチ力行時の電流値を記入する。
   - 2C. minimumvoltage コマンド: 最低電圧を記入する。400V など，かなり低くてよい。
   - 2D. bminvoltage コマンド: ダミーパラメータである。通常 1200V 程度を記入する。
   - 2E. maximumvoltage コマンド: 最高電圧を記入する。
   - 2F. empty コマンド，mid コマンド，full コマンド: 応荷重装置のシミュレーションに用いる特性指標を記入する。empty のあとのものが空車，full のあとのものが満車で，もうひとつ mid はその中間のものを与える。mid は「3 点指示方式」の採用によって必要となったもので，詳しいことは (F.2.1) を参照されたい。これら 3 点とも指定することが求められる。
     それぞれのコマンドの下には以下のコマンド群をすべて記入する。
     - (i) congestion コマンド: 混雑率。empty の下は 0.0，full の下は最大乗車時を記入。なお，full の混雑率以上では応荷重装置は働かない（最大性能で固定される）ように考える。
     - (ii) maximumcurrent コマンド: 編成あたりの最大パンタ点電流 (A) を記入。
     - (iii) lowvelocity コマンド: 定トルク領域終端速度 (km/h) を記入。
     - (iv) highvelocity コマンド: 定パワー領域終端速度 (km/h) を記入。
     - (v) tractionforce コマンド: 定トルク領域引張力 (tf) を記入する。
     - (vi) full_lowvel コマンド: ここでは力行満車時の定トルク領域終端速度を
3. brakedata コマンド: 列車のブレーキ時性能を記述するコマンド群の始まりを指定するコマンドである。このコマンド群は以下のようなものだ。
   - 3A. ec コマンド: ブレーキ性能を表す数字は，電車線電圧がこのコマンドで指定される電圧値であるときのものを記入する。
   - 3B. regenoff コマンド: 回生ブレーキ終端速度 (km/h) を記入する。
   - 3C. minimumvoltage, bminvoltage, maximumvoltage コマンド: 力行時データと同じ。
   - 3D. regenlimitstartvoltage コマンド: 回生絞り込み開始電圧 (V) を記入する。
   - 3E. regenlimitendvoltage コマンド: 回生絞り込み終了電圧 (V) を記入する。differentregend フラグがない場合はこのコマンドを記入するとエラーとなる。
   - 3F. regenoffanteika コマンド: パンタ点電流がこれを下回ったら回生失効とする電流値 (A) を記入する。regendantei フラグがない場合はこのコマンドを記入するとエラーとなる。また，anteimotor フラグが立っているとこれはモータ電流のこととなる。
   - 3G. empty コマンド，mid コマンド，full コマンド: 力行時と同じく，応荷重装置のシミュレーションに用いる特性指標を記入する。empty のあとのものが空車，full のあとのものが満車で，も

うひとつ mid はその中間のものを与える「3 点指示方式」である。

それぞれのコマンドの下には力行時と同じコマンド群をすべて記入する。もちろん回生ブレーキ時の特性として記入するべきである。なお，anteimotor フラグが立っている場合に限り次のコマンドをも記入する。

- motorcurrent コマンド: そのときの最大モータ電流 (A) を記入する。

4. autodata コマンド: cs_command コマンドに Ec_auto を指定した場合に意味があるコマンド群である。意味がない場合も省略できないので，以下のコマンドすべてに適当な値を入れておく。

   4A. minimumvelocity コマンド: このコマンドは，Ec_auto および Ea_cvauto0 のモードにおいて，制御をかけはじめる最低の速度 (km/h) を示す。この速度より下では，列車はどのモードであっても None とまったく同じふるまいをする。

   4B. fullvelocity コマンド: このコマンドは，制御をフルにかける最低の速度を示す。minimumvelocity コマンドで指定した速度とこの速度の間では，制御をフルにはかけない状態になる。この速度より上では制御はフルにかかる。

   4C. powerlimitvoltage コマンド: 力行時，列車の電流を絞り込みはじめる電圧。これより電車線電圧が低いと「フルノッチ比」が下がる。惰行時には，列車がピーク救済のための回生ブレーキをかけはじめる電圧である。

   4D. poweroffvoltage コマンド: 力行時，列車の電流を完全に絞り込み終わる電圧。これより電車線電圧が低いとフルノッチ比は一定となり，速度が fullvelocity コマンドで指定された速度より高ければフルノッチ比 0，すなわち力行電流 0 となる。惰行時には，列車がピーク救済のための回生ブレーキをもっとも強くかける領域の上限電圧となり，これより電車線電圧が低いとフルノッチ比は最低となり，速度が fullvelocity より高ければフルノッチ比は−1 である。

   4E. regenerationstartvoltage コマンド: 惰行時，回生失効防止のため惰行車が加速しはじめる電圧。

   4F. regenerationfullvoltage コマンド: 惰行時，回生失効防止のため惰行車が最大加速する領域の下限電圧。

5. congestion コマンド: 列車の混雑率を与える。%で与えてはならない。

```
congestion 1.0
```

   のように書く。

6. auxcurrent コマンド: 列車の補機電流 (A) を与える。auxpower フラグが立っていれば補機パワー (kW) を与える。

```
auxcurrent 30.0
```

   のように書く。

7. setweight コマンド: 順に「電動車自重」「電動車定員乗車時荷重」「付随車自重」「付随車定員乗車時荷重」を，4 つの実数によって与える。

```
setweight 190.0 50.0 150.0 50.0
```

   のように書く。

8. setrest コマンド: 走行抵抗式を 6 つの定数で与える。6 定数を順に a, b, c, d, e, f，列車速度 v[km/h]，電動車質量 M_d[t]，付随車質量 M_t[t]，編成両数 n[両] とするとき，走行抵抗 R[kgf] は

$$R = (a + bv)M_d + (c + dv)M_t + \{e + (n-1)f\}v^2 \tag{J.2}$$

   と与える。通常は

```
setrest 1.65 .0247 .78 .028 .0358 .0078
```

などと書けばよい。

9. setmres コマンド: 曲線抵抗の係数，回転部分補正係数，平坦線減速度，編成両数を入れる。

```
setmres 800.0 0.0825 3.0 10
```

のように書く。

⟨J.5.3⟩ サンプル

データファイルの例を下に示す。

```
# train characteristics file for RTSS ver. 2.1.1

regendantei     # This flag enables regeneration stop simulation.
differentregend # This flag makes regendvol and maxvol different.
shiboriparallel # This flag makes overvoltage limit narrower.
auxpower        # This flag makes auxcurr point auxiliary power.
torque_shibori  # This flag makes motor current for regeneration limit.
anteimotor      # With this flag, RTSS will use motor current for
                # regeneration down decision.

cs_command None

powerdata
        ec 1350.0
        zerocur 100.0
        minimumvoltage 400.00
        bminvoltage 1200.0
        maximumvoltage 1800.0
        empty
                congestion 0.0
                maximumcurrent 1750.0
                lowvelocity 64.0
                highvelocity 64.0
                tractionforce 11.9995
        mid
                congestion 0.5
                maximumcurrent 1850.0
                lowvelocity 60.54
                highvelocity 60.54
                tractionforce 13.6405
        full
                congestion 2.5
                maximumcurrent 1850.0
                lowvelocity 42
                highvelocity 60.54
                tractionforce 20.2459

brakedata
        ec 1650.0
        regenoff 5.0
        minimumvoltage 400.00
        bminvoltage 1200.0
        maximumvoltage 1800.0
        regenlimitstartvoltage 1650.0
        regenlimitendvoltage 1700.0
```

```
        regenoffanteika 70.0
        empty
                congestion 0.0
                maximumcurrent 1340.0
                lowvelocity 87.0
                highvelocity 87.0
                tractionforce 10.2432
                motorcurrent 376.0
        mid
                congestion 2.5
                maximumcurrent 1670.0
                lowvelocity 69.81
                highvelocity 69.81
                tractionforce 17.2975
                motorcurrent 616.0
        full
                congestion 2.5
                maximumcurrent 1670.0
                lowvelocity 69.81
                highvelocity 69.81
                tractionforce 17.2975
                motorcurrent 616.0

autodata
        minimumvelocity 20.0
        fullvelocity 35.0
        powerlimitvoltage 1450.0
        poweroffvoltage 1200.0
        regenerationstartvoltage 1600.0
        regenerationfullvoltage 1750.0

congestion 0.20
auxcurrent 30   # [kW]
setweight 65.0  16.665  51.4  15.4
setrest 2  .11  1  .0132  .063  .0078
setmres 600.0  0.0825  3.0  4
```

⟨J.6⟩ 饋電線データファイル（F ファイル）

饋電線とその接続関係を示すデータファイルである。また，饋電システム全体に関するデータもここに書き込まれる。

⟨J.6.1⟩ フラグ

このデータファイルに書き込めるフラグは，現在のところ次の 1 種類のみが認められる。

- options フラグ: このフラグでいろいろなオプションを設定することができる。このフラグを複数書くことができるが，同じオプションを 2 度書くとトグルスイッチになっているため指定が無効化されて元に戻ってしまうから注意が必要だ。データはオプションを指示する次の文字列のいずれかである：
  1. ccalresult: これを与えると，ログに計算結果を等価回路演算毎に吐き出す。ログファイルが大量になるのであまりお勧めできない。
  2. ssdirection: これを与えると，変電所での方面別電流を計算する。まだバグが残っているようだ。

3. kilowatt_subout: 出力ファイルにおいて，変電所別出力電力量を kW 単位で表示する。
4. per_car_out: 列車の力行・回生電力量を列車あたりで出力。列車がすべて同じような走り方をするのでないと意味がない。

⟨J.6.2⟩ コマンド

このデータファイルには次のようなコマンドを，ここで出てきた順に書き込む必要がある。

1. feeders コマンド: 饋電線の数，およびダイヤパターンに対する指示のふたつのデータを書き込む。

   饋電線の数は整数で記入する。ダイヤパターンに対する指示は，余裕時分の再配分を行なうときは Buffer_yes，早着防止を行なうときは Delay_yes，何もしないときは No_comm を，それぞれ文字列で記入する。

   ```
   feeders 2 No_comm
   ```

   のように書く。

2. freeruns コマンド: シミュレーション開始時までの空回しの回数を整数で記入する。多くの場合，大きいほど正確なシミュレーションができるが，時間がかかる。ただし，場合によっては空回しが行えないモデルもあろう。

   ```
   freeruns 3
   ```

   のように書く。

3. simulatings コマンド: シミュレーションを行なう周期を記入する。周期的に列車が動いてくれる分には，大きい必要はない。通常 1 ないし 2 で十分である。

   ```
   simulatings 2
   ```

   のように書く。

4. setfeedline コマンド: すべての饋電線について，その長さ，饋電定数，形状を記入する。このコマンドは，feeders コマンドで指定した饋電線数と同じ数だけ現われなければならない。

   ```
   setfeedline 34.475 0.0327 Long
   setfeedline 34.475 0.0327 Long
   ```

   のように書く。データは，それぞれ饋電線長 [km]，饋電定数 [Ω/km]，饋電線形状である。形状は，線分状のときは 0 または Long，環状のときは 1 または Circle を，それぞれ記入すればよい。また，最初の setfeedline コマンドが饋電線番号 0，次が 1，などのように対応する。この饋電線番号が次駅データファイル 'nextsta.dat' にて nfline コマンドで指定される饋電線番号に対応するので注意されたい。

5. ssfeederconnections コマンド: 可変長データを持つコマンドであり，このコマンド 1 つで変電所と饋電線との接続関係をすべて表現する。

   最初のデータは整数で，変電所と饋電線との接続点に関するデータがいくつあるかを記入する。次いで，この整数の数だけの饋電線・変電所接続点データが続く。

   ひとつの饋電線・変電所接続点データは 2 つの整数と 3 つの倍精度実数の都合 5 つの数字を 1 組としたデータである。このうち，最初の整数が饋電線番号，次の整数が変電所番号，3 つ目（倍精度実数）が接続点位置 (km)，4 つ目・5 つ目（どちらも倍精度実数値）はそれぞれ Busbar から起点側接続点までのインピーダンス，および Busbar から終点側接続点までのインピーダンス（どちらも単位 Ω）を示す。なお，変電所が饋電線から極端に離れていない多くのシミュレーションケースでは，最後の 2 つのデータは 0 となるが，変電所が饋電線から離れており，Busbar と饋電線との間のインピーダンスが無視できない場合には 0 以外の値を入れる。

```
ssfeederconnections 6
0 1 0.0 0.0 0.0
0 0 5.0 0.0 0.0
0 2 10.0 0.0 0.0
1 1 0.0 0.0 0.0
1 0 5.0 0.0 0.0
1 2 10.0 0.0 0.0
```

のように書く。

データファイルの例を下に示す。

```
# "feeder" data file format for RTSS 2.1.1

# options command: you can write any of them before feeders command
#       options ccalresult      # toggle g_sw_ccalresult, default: FALSE
        options ssdirection     # toggle g_sw_ss_direction, default: FALSE
        options kilowatt_subout # toggle g_sw_kilowatt_subout
        options per_car_out     # toggle g_sw_per_car_out

feeders 2 No_comm
freeruns 3              # free run 3 cycles
simulatings 2           # simulation 2 cycles

setfeedline     7.0     0.033   Long    # No. 0 feedline, Long/Circle
setfeedline     7.0     0.033   Long    # No. 1 feedline, Long/Circle

ssfeederconnections 6
#       nfline  csno    pos     sttimp  endimp
        0       1       1.0     0       0
        0       0       6.0     0       0
        0       2       11.0    0       0
        1       1       1.0     0       0
        1       0       6.0     0       0
        1       2       11.0    0       0
```

⟨J.7⟩ 列車初期位置データファイル

このデータファイルは，なければプログラムが勝手に計算して作るので，ユーザは気にする必要はないものだった。列車初期位置の計算には非常に時間がかかるので，このようなファイルをつくって少しでも計算時間を短縮しようとしたものである。しかし，RTSS ver.2.1 開発の過程でバグが発見されたため，現在はデータのセーブはするものの利用していない。将来はプログラムを整備し，再び機能を復活させる予定であるが，いつになることやら……。

この機能が有効になった暁には次のことに注意されたい。すなわち，条件を変えれば当然初期位置データファイルの中身も変わるので，条件を変えた場合に別な条件における列車初期位置データファイルを誤って読み込むことのないよう，コマンドオプションの指定には注意すべきであろう。ただし，列車初期位置は電車線電圧などには依存しない（電車線電圧としてノミナル値を仮定するため）ので，ある程度の共用はできる。

RTSS マニュアル・一般概念索引

き

こ

さ

し

た

て

と

の

は

ひ

ふ

へ

め

も

り

れ

RTSS マニュアル・関数・変数索引

A

C

D

E

F

G

M

N

O

R

S

T

V

Y

Z

RTSS マニュアル・クラス・構造体名索引

RTSS マニュアル・コマンド名索引

N

O

P

R

S

T

V

Z

RTSS マニュアル・記号索引

C

I

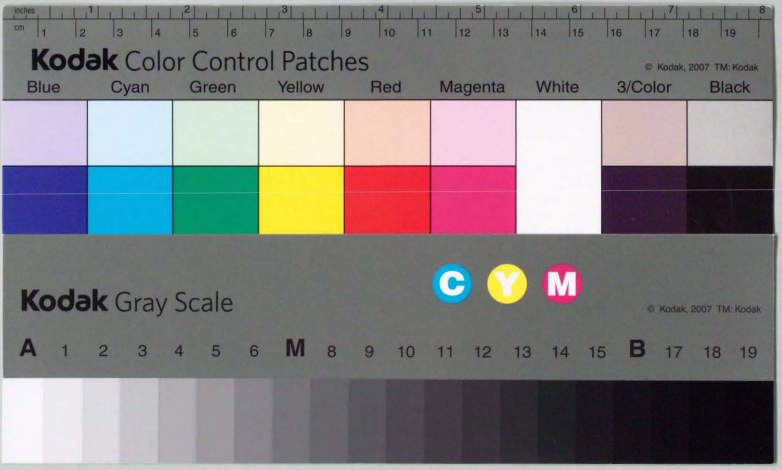
K

N

R

T

V


inches
cm
Kodak Color Control Patches
© Kodak, 2007 TM: Kodak
Blue
Cyan
Green
Yellow
Red
Magenta
White
3/Color
Black
Kodak Gray Scale
C
Y
M
© Kodak, 2007 TM: Kodak
A 1 2 3 4 5 6 M 8 9 10 11 12 13 14 15 B 17 18 19