Mô hình hóa các hiện tượng vật lý bằng Scilab

Bản dịch 2012 Nguyễn Quang Chiến

Green Tea Press
9 Washburn Ave
Needham MA 02492

The original form of this book is LaTeX source code. Compiling this code has the effect of generating a device-independent representation of a textbook, which can be converted to other formats and printed.

This book was typeset by the author using latex, dvips and ps2pdf, among other free, open-source programs. The LaTeX source for this book is available from `http://greenteapress.com/matlab`.

Lời giới thiệu dành cho bản gốc cuốn sách MATLAB

Hầu hết các cuốn sách đề cập đến MATLAB đều hướng tới người đọc đã biết lập trình. Cuốn sách này dành cho những người chưa từng lập trình từ trước.

Do đó, trình tự nội dung ở đây sẽ khác thường. Cuốn sách mở đầu với các giá trị vô hướng rồi dần tiếp tục với véc-tơ và ma trận. Cách tiếp cận này rất tốt với người mới bắt đầu lập trình, vì thật khó hiểu được những kiểu đối tượng dữ liệu phức tạp trước khi bạn hiểu ý nghĩa căn bản của lập trình. Nhưng có những vấn đề sau đây nảy sinh:

- Tài liệu MATLAB được viết dưới dạng các ma trận, và các thông báo lỗi cũng vậy. Để hạn chế nhược điểm này, cuốn sách giải thích các thuật ngữ cần thiết từ sớm và giải mã một số thông báo lỗi có thể làm người bắt đầu thấy khó hiểu.

- Nhiều ví dụ trong nửa đầu cuốn sách không phải là viết theo phong cách MATLAB thực sự. Tôi nhận định lại vấn đề này trong phần nửa sau cuốn sách bằng việc chuyển chúng về phong cách chính thống hơn.

Cuốn sách này nhấn mạnh về các hàm số, một phần là vì chúng là cơ chế quan trọng chi phối độ phức tạp của chương trình, và cũng vì chúng rất có ích khi làm việc với các công cụ của MATLAB như `fzero` và `ode45`.

Tôi định rằng bạn đọc đã biết môn toán giải tích, phương trình vi phân, và vật lý, nhưng không cần đại số tuyến tính. Tôi sẽ giải thích về toán trong suốt nội dung sách, nhưng bạn cũng cần biết toán để nắm được những đoạn giải thích đó.

Có những bài tập nhỏ trong từng chương, và một số bài tập lớn hơn ở cuối những chương nhất định.

Nếu bạn muốn góp ý và sửa chữa nội dung cuốn sách, hãy gửi ý kiến của bạn đến `downey@allendowney.com`.

Allen B. Downey
Needham, MA

Lời giới thiệu

Mặc dù đã được phát triển nhiều năm, song phần mềm Scilab vẫn chưa được ứng dụng rộng rãi. Một phần là do tài liệu về Scilab còn quá thưa thớt. Hi vọng cuốn sách này sẽ giúp ích cho bạn làm quen với ngôn ngữ lập trình này. Hơn nữa, cuốn sách có thể còn lý thú đối với học sinh cuối cấp Trung học phổ thông, giúp các em tìm hiểu cách diễn đạt những hiện tượng vật lý đơn giản bằng những phép toán có thể giải được trực tiếp trên máy tính.

Scilab là một bộ phần mềm đồ sộ. Nó có nhiều đặc điểm giống như MATLAB. Một số hàm còn thiếu so với MATLAB, bạn đọc có thể tự viết để bổ sung theo những gợi ý hướng dẫn trong sách.

Quang Chiến
Tháng 12-2012

Mục lục

Chương 1

Các biến và giá trị

1.1 Chiếc máy tính tay

Phần cốt lõi của Scilab là một chiếc máy tính tay. Khi khởi động Scilab bạn sẽ thấy một cửa sổ có tiêu đề Scilab Console. Đây là cửa sổ lệnh dùng để chạy **trình thông dịch** Scilab; nó cho phép bạn gõ vào các **câu lệnh** Scilab, rồi thực hiện chúng và in ra kết quả.

Ban đầu, cửa sổ lệnh hiện một dòng chữ chào người dùng và thông tin về phiên bản Scilab đang chạy, tiếp theo là dấu

```
-->
```

vốn là **dấu nhắc** của Scilab; tức là ký hiệu dùng để nhắc bạn nhập vào một câu lệnh.

Dạng câu lệnh đơn giản nhất là một **biểu thức** toán học, vốn được hợp thành từ các **toán hạng** (ví dụ như các số) và các **toán tử** (như dấu cộng, +).

Nếu bạn gõ vào một biểu thức và ấn Enter (hoặc Return), Scilab sẽ **lượng giá** biểu thức và in ra kết quả.

```
--> 2 + 1
ans =
   3.
```

Để nói rõ hơn: trong ví dụ trên, Scilab tự in ra `-->`; tôi gõ vào `2 + 1` và ấn Enter, rồi Scilab lại in ra `ans = 3.`. Khi tôi nói là "in", điều đó có nghĩa là "hiện lên màn hình". Điều này có thể làm bạn thoạt đầu thấy dễ lẫn, nhưng đó chính là cách mọi người vẫn nói.

Ở phép tính trên, sau số 3 có một dấu chấm; đó là "dấu phẩy" ngăn cách giữa phần nguyên và phần thập phân. Scilab luôn coi những con số là số thực, dù cho đôi khi kết quả tính toán là số nguyên. Khi đó, dấu chấm ở cuối cùng báo hiệu rằng: không còn chữ số nào trong phần thập phân nữa.

Một biểu thức có thể bao gồm bao nhiêu toán tử và toán hạng cũng được. Bạn không cần phải gõ các dấu cách; có người gõ và có người không.

```
--> 1+2+3+4+5+6+7+8+9
ans = 45.
```

Ở kết quả phép tính trên, sau dấu bằng Scilab còn in ra một dòng trống trước khi in kết quả, nhưng để cho gọn, từ nay trở về sau tôi sẽ viết liền kết quả vào đấu bằng để tiết kiệm chỗ.

Các toán tử số học khác cũng giống như bạn đã biết. Phép trừ kí hiệu bởi dấu `-`; phép nhân bởi một dấu sao, `*`; phép chia bởi dấu gạch chéo xuôi `/`.

```
--> 2*3 - 4/5
ans = 5.2
```

Thứ tự thực hiện phép toán cũng giống như trong môn đại số: các phép nhân và chia được thực hiện trước các phép cộng và trừ. Bạn có thể dùng cặp ngoặc đơn để thay đổi thứ tự tính.

```
--> 2 * (3-4) / 5
ans = - 0.4
```

Khi thêm vào cặp ngoặc đơn, tôi cũng đồng thời thay đổi độ dãn cách để ta dễ đọc hơn. Đây là một trong những gợi ý về cách trình bày trong cuốn sách, để chương trình được dễ đọc. Bản thân phong cách không làm ảnh hưởng đến tính năng của chương trình; trình thông dịch Scilab không kiểm tra phong cách. Nhưng người đọc thì có, và quan trọng nhất bạn chính là người đọc thường xuyên nhất các mã lệnh bạn viết ra.

Từ đó dẫn đến định lý thứ nhất về gỡ lỗi chương trình:

> Mã lệnh dễ đọc cũng dễ gỡ lỗi.

Thời gian bạn bỏ ra để làm đẹp mã lệnh hoàn toàn xác đáng; điều này sẽ giúp bạn tiết kiệm thời gian gỡ lỗi!

Toán tử thông dụng tiếp đến là lũy thừa, với ký hiệu `^`, đôi khi còn được gọi là “dấu mũ”. Số 2 nâng lên lũy thừa 16 là

```
--> 2^16
ans = 65536.
```

Cũng như trong đại số cơ bản, phép lũy thừa được thực hiện trước các phép nhân và chia, nhưng một lần nữa, bạn có thể dùng cặp ngoặc tròn để thay đổi thứ tự thực hiện phép tính.

1.2 Các hàm toán học

Scilab biết cách tính gần như mọi hàm toán học bạn biết đến. Nó biết tất cả các hàm lượng giác; sau đây là cách dùng:

```
--> sin(1)
ans = 0.8414710
```

Lệnh này là một ví dụ của một **lời gọi hàm**. Tên của hàm này là `sin`, vốn là ký hiệu thông dụng của hàm lượng giác sin. Giá trị trong cặp ngoặc tròn được gọi là **đối số**. Tất cả các hàm lượng giác trong Scilab đều tính theo ra-đian.

Một số hàm nhận nhiều đối số, khi đó chúng được phân cách bởi cách dấu phẩy. Chẳng hạn, `atan` dùng để tính nghịch đảo của hàm tan, vốn là góc tính theo ra-đian giữa chiều dương trục x và điểm có các tọa độ y và x cho trước.

```
--> atan(1,1)
ans = 0.7853982
```

Nếu bạn không rành chút kiến thức lượng giác trên thì cũng không nên lo lắng. Đó chỉ là một ví dụ cho thấy hàm có nhiều đối số.

Scilab cũng có các hàm lũy thừa, như `exp`, dùng để tính số e nâng lên một số mũ cho trước. Vì vậy `exp(1)` chính là e.

```
--> exp(1)
ans = 2.7182818
```

Nghịch đảo của `exp` là `log`, nhằm tính loga cơ số e:

```
--> log(exp(3))
ans = 3.
```

Ví dụ này cũng cho thấy các lời gọi hàm có thể **lồng ghép** được; nghĩa là bạn có thể dùng kết quả tính được từ hàm này để làm đối số cho một hàm khác.

Một cách tổng quát hơn, bạn có thể dùng lời gọi một hàm như là toán hạng cho một biểu thức.

```
--> sqrt(sin(0.5)^2 + cos(0.5)^2)
ans = 1.
```

Như bạn có thể đoán được, `sqrt` có tác dụng tính căn bậc hai.

Còn có rất nhiều hàm toán học khác, nhưng cuốn sách này không nhằm mục đích là một cuốn sổ tra cứu. Để biết cách dùng các hàm khác, bạn cần phải đọc các đoạn thông tin giải thích từng hàm.

1.3 Thông tin về hàm

Scilab đi kèm theo hai dạng thông tin trực tuyến giải thích cách dùng hàm, đó là `help` và `doc`.

Lệnh `help` hoạt động trong cửa sổ lệnh; bạn chỉ cần gõ `help` theo sau là tên của lệnh.

```
--> help sin
 SIN    Sine of argument in radians.
    SIN(X) is the sine of the elements of X.

    See also asin, sind.

    Overloaded functions or methods (ones with the same name in oth
       directories) help sym/sin.m

    Reference page in Help browser
       doc sin
```

Khi đó, trình duyệt tra cứu (Help Browser) khởi động và đưa bạn đến mục lệnh "sin". Trong đó có giải thích:

- Calling Sequence: cách gọi hàm
- Arguments: đối số của hàm

Description: mô tả tác dụng của hàm

Examples ví dụ sử dụng. Khi bạn ấn vào nút ▷ thì Scilab sẽ tự gõ hộ lệnh vào dấu nhắc và thực hiện. Còn nếu ấn vào nút hình cuốn sổ ghi thì lệnh hoặc đoạn lệnh này sẽ được chèn vào một file mã lệnh (sẽ được giới thiệu trong Chương 2).

See Also Các lệnh hoặc hàm có liên quan. Ở đây có `sinm` là hàm sin tính cho ma trận.

Bạn luôn phải cẩn thận: chữ in và chữ thường là khác nhau. Chẳng hạn, nếu gõ hàm sin nói trên bằng chữ in:

```
--> SIN(1)
       !--error 4
Undefined variable: SIN
```

Ở đây Scilab cho rằng SIN là một biến chưa được định nghĩa; khái niệm "biến" sẽ được đề cập ngay ở mục sau đây.

1.4 Biến

Một trong những đặc điểm giúp Scilab mạnh hơn một máy tính tay là khả năng đặt tên cho một giá trị. Một giá trị sau khi đã đặt tên được gọi là một **biến**.

Scilab đi kèm theo một số giá trị định sẵn. Chẳng hạn, cái tên `%pi` dùng để chỉ đại lượng π trong toán học, vốn gần bằng

```
--> %pi
ans = 3.1415927
```

Và nếu bạn làm bất cứ phép tính nào với số phức, có thể bạn sẽ thấy tiện khi dùng `%i` vốn được định nghĩa sẵn là căn bậc hai của -1. Dấu `%` không phải là phép tính phần trăm mà là để chỉ một tên có giá trị định trước.

Bạn có thể dùng một tên biến ở bất cứ chỗ nào viết được một số; chẳng hạn, như toán hạng trong một biểu thức:

```
--> %pi * 3^2
ans = 28.274334
```

hoặc như đối số cho một hàm:

```
--> sin(pi/2)
ans = 1.

--> exp(i * pi)
ans = - 1. + 1.225D-16i
```

Ở đây chữ D là kí hiệu lũy thừa với cơ số mười. `1.225D-16` cũng như 1,225 $\times 10^{-16}$. Các phần sau của cuốn sách sẽ đề cập về cách viết trên. Ví dụ này cho thấy rằng nhiều hàm Scilab cũng tính được với số phức. Cụ thể ở đây là biểu diễn đẳng thức Euler: $e^{i\pi} = -1$.

Mỗi khi lượng giá một biểu thức, Scilab gán giá trị cho một biến có tên là `ans`. Bạn có thể dùng `ans` trong phép tính tiếp theo như một cách viết tắt cho "giá trị của biểu thức liền trước".

```
--> 3^2 + 4^2
ans = 25.

--> sqrt(ans)
ans = 5.
```

Nhưng nhớ rằng giá trị của `ans` lại thay đổi mỗi khi bạn lượng giá một biểu thức.

1.5 Lệnh gán

Bạn có thể tự tạo các biến và cho chúng các giá trị, bằng cách dùng **lệnh gán**. Toán tử gán là dấu bằng, =.

```
--> x = 6 * 7
x = 42.
```

Ví dụ này tạo ra một biến mới có tên là `x` và gán cho nó giá trị của biểu thức `6 * 7`. Scilab trả lời lại với tên biến và giá trị tính được.

Trong mỗi câu lệnh gán, vế trái phải là một tên biến hợp lệ. Vế phải có thể là một biểu thức bất kì, bao gồm cả lời gọi hàm.

Hầu như bất kỳ dãy chữ cái viết thường và viết in nào cũng tạo nên một tên biến hợp lệ. Một số dấu nhất định cũng hợp lệ, nhưng chỉ có dấu gạch thấp, _, là kí hiệu thông dung nhất. Các chữ số cũng được, nhưng không được đặt chúng ở đầu tên biến. Không được dùng các dáu cách. Các tên biến đều có sự phân biệt giữa chữ in và chữ thường, vì vậy `x` và `X` là các biến khác nhau.

```
--> fibonacci0 = 1;

--> LENGTH = 10;

--> first_name = 'allen'
first_name = allen
```

Hai ví dụ đầu cho thấy cách dùng của dấu chấm phẩy, dùng để ngăn không cho in ra kết quả của câu lệnh. Trong trường hợp này Scilab tạo ra biến và gán nó với giá trị, nhưng không hiển thị gì.

Ví dụ thứ ba cho thấy rằng không phải mọi thứ trong Scilab đều là con số. Một dãy các ký tự trong cặp dấu nháy được gọi là một **chuỗi**.

Các giá trị như `%i` và `%pi` đều được định nghĩa trước và bạn không thể thể gán lại chúng.

1.6 Tại sao phải dùng biến?

Các lý do chung lý giải cho việc dùng biến là

- Để tránh việc tính lại một giá trị được dùng lặp lại nhiều lần. Chẳng hạn, nếu bạn thực hiện tính toán liên quan đến e, có thể bạn sẽ muốn tính nó một lần và lưu lại kết quả.

  ```
  --> e = exp(1)
  e = 2.7182818
  ```

- Để làm cho sự gắn kết giữa mã lệnh và cơ sở toán học trở nên rõ ràng hơn. Nếu bạn tính diện tích của một hình tròn, có thể bạn muốn dùng một biến có tên là `r`:

  ```
  --> r = 3
  r = 3.

  --> area = %pi * r^2
  area = 28.274334
  ```

  Bằng cách này mã lệnh viết ra sẽ giống với công thức quen thuộc πr^2.

- Để chẻ nhỏ một phép tính dài thành một loạt các bước. Chẳng hạn bạn phải lượng giá một biểu thức đồ sộ, gai góc như sau:

```
ans = ((x - theta) * sqrt(2 * %pi) * sigma) ^ -1 * ...
exp(-1/2 * (log(x - theta) - zeta)^2 / sigma^2)
```

  Bạn có thể dùng dấu ba chấm để tách một biểu thức thành nhiều dòng. Chỉ việc gõ vào `...` ở cuối dòng trên rồi tiếp tục gõ xuống dòng dưới.

  Nhưng cách tốt hơn thường là chia phép tính thành một loạt các bước kế tiếp và gán những kết quả trung gian cho các biến.

```
shiftx = x - theta
denom = shiftx * sqrt(2 * %pi) * sigma
temp = (log(shiftx) - zeta) / sigma
exponent = -1/2 * temp^2
ans = exp(exponent) / denom
```

  Tên của các biến trung gian giải thích vai trò của chúng trong phép tính. `shiftx` là giá trị của `x` bị dịch chuyển đi `theta` đơn vị. Cũng dễ hiểu khi đặt `exponent` là đối số cho hàm `exp`, và `denom` thay thế cho mẫu số. Việc chọn những cái tên có nghĩa làm cho mã lệnh dễ đọc và dễ hiểu (xem Định lý thứ nhất về gỡ lỗi).

1.7 Lỗi

Tuy giờ còn sớm nhưng rất có thể bạn đã bắt đầu mắc lỗi khi lập trình. Mỗi khi học thêm được một đặc điểm mới của Scilab, bạn cần phải thử để gây ra lỗi, càng nhiều càng tốt, càng sớm càng tốt.

Khi bạn cố ý gây ra lỗi, bạn sẽ thấy các thông báo lỗi trông như thế nào. Sau này, khi bạn vô tình mắc lỗi, bạn sẽ hiểu được những thông báo lỗi khi đó có ý gì.

Một lỗi hay gặp ở người mới lập trình là việc bỏ qua dấu `*` trong phép nhân.

```
-->area = %pi r^2
              !--error 276
Missing operator, comma, or semicolon.
```

Thông báo lỗi rất rõ ràng: ở vị trí mà dấu chấm than chỉ lên bị thiếu một toán tử, một dấu phẩy, hoặc chấm phẩy. Dấu chấm phẩy thì rõ rồi, nó để kết thúc lệnh hoặc ngăn giữa nhiều lệnh khác nhau trên cùng một dòng. Dấu phẩy cũng có tác dụng tương tự nhưng khác ở chỗ không thể ngăn cho giá trị bị in ra màn hình. Ta không muốn dùng dấu phẩy hoặc chấm phẩy ở đây vì chỉ có một câu lệnh. Vậy ta thiếu một toán tử, cụ thể là dấu nhân (`*`).

Một lỗi hay gặp khác là việc bỏ quên cặp ngoặc tròn bao quanh các đối số của một hàm. Chẳng hạn, trong cách viết toán học, ta thường ghi $\sin \pi$, nhưng trong Scilab thì không được như vậy.

```
--> sin %pi
   !--error 246
Function not defined for given argument type(s),

  check arguments or define function %c_sin for overloading.
```

Vấn đề ở đây là khi bạn bỏ mất cặp ngoặc, Scilab sẽ coi như đối số là một chuỗi kí tự (thay vì một biểu thức).* Trong trường hợp này, hàm `sin` sẽ gây ra một thông báo lỗi hợp lý. Như vậy, bất kể khi bạn áp dụng hàm cho đối số là một biểu thức dài hoặc một con số, thì đều phải đặt chúng trong cặp ngoặc tròn.

Ví dụ này minh họa cho Định lý thứ hai của việc gỡ lỗi.

> Điều duy nhất còn tồi tệ hơn cả việc nhận được thông báo lỗi là việc *không* nhận được thông báo lỗi nào.

Những người mới bắt đầu lập trình ghét các thông báo lỗi và tìm mọi cách xua đuổi chúng đi. Những người lập trình có kinh nghiệm hiểu rằng thông báo lỗi là người bạn của họ. Chúng có thể khó hiểu, và thậm chí có thể đánh lạc hướng, nhưng công sức bỏ ra để hiểu được chúng sẽ được đền bù xứng đáng.

Sau đây là một lỗi thông thường khác của người mới bắt đầu. Nếu bạn chuyển biểu thức toán sau đây sang Scilab:

$$\frac{1}{2\sqrt{\pi}}$$

Bạn có thể muốn viết như sau:

```
1 / 2 * sqrt(%pi)
```

Nhưng cách này sai. Rất sai.

*Bạn có thể tự hỏi, liệu có hàm nào nhận đối số là chuỗi kí tự không? Có đấy, ví dụ hàm length để tính chiều dài của chuỗi; tuy nhiên ở cuốn sách này ta không chú ý nhiều đến kiểu chuỗi kí tự.

1.8 Phép toán số học với những số có phần thập phân

Trong toán học, có một vài loại số: số nguyên, số thực, hữu tỉ, vô tỉ, số ảo, số phức, v.v. Scilab chỉ có một kiểu số, đó là số có phần thập phân, chính xác hơn là **dấu phẩy động**.

Bạn có thể đã nhận thấy rằng Scilab biểu thị các giá trị theo cách viết có phần thập phân. Vì vậy, số hữu tỉ $1/3$ chẳng hạn được biểu diễn bởi giá trị phẩy động như sau

```
--> 1/3
ans = 0.3333333
```

vốn chỉ gần đúng. Thực ra nó không đến nỗi dở như ta có thể hình dung; Scilab dùng nhiều chữ số hơn là nó biểu diễn (hiển thị) trên màn hình. Bạn có thể thay đổi bề rộng hiển thị của phần thập phân bằng lệnh `format`. Chẳng hạn, để thấy 10 chữ số phần thập phân,

```
--> format(13)
--> 1/3
ans = 0.33333333333333
```

Đó là vì một chữ số dành cho dấu phẩy, một chữ số nữa (trong trường hợp này) dành cho hàng đơn vị, và trước đó để dành một chỗ cho dấu âm phòng khi cần đến.

Về bản chất, Scilab dùng dạng dấu phẩy động IEEE với độ chính xác kép, vốn có khoảng 15 chữ số có nghĩa (theo hệ thập phân). Các chữ số không đứng ở đầu hoặc cuối không được tính là chữ số "có nghĩa", vì vậy Scilab có thể biểu diễn cả những số lớn lẫn nhỏ với cùng lượng chữ số có nghĩa như vậy.

Những giá trị rất lớn và rất nhỏ được biểu diễn theo cách viết khoa học.

```
--> factorial(100)
ans = 9.33262D+157
```

Kí hiệu `D` trong cách viết này, như ta đã gặp, là để chỉ lũy thừa của 10. Vì vậy ở đây, $100!$ xấp xỉ với 9.33×10^{157}. Đáp số chính là một số nguyên gồm 158 chữ số, nhưng ở đây ta chỉ biết được 6 chữ số đầu tiên.

Bạn có thể tự nhập vào các số theo cách viết tương tự.

```
--> speed_of_light = 3.0E8
speed_of_light = 300000000.
```

(Dĩ nhiên ta có thể viết chữ D thay cho E như Scilab thường dùng.)

Dù Scilab có thể xử lý được những số lớn, nhưng vẫn có một giới hạn. Để biết được các giới hạn này ta dùng hàm `number_properties` với các đối số `"huge"` hoặc `"tiny"` Các biến được định nghĩa trước, `realmax` và `realmin`, chứa các giá trị số lớn nhất và nhỏ nhất mà Scilab có thể xử lý[†].

```
--> number_properties("huge")
ans = 1.79769D+308

--> number_properties("tiny")
ans = 2.22507D-308
```

Nếu kết quả tính toán quá lớn, Scilab sẽ "làm tròn lên" thành vô cùng.

```
--> factorial(170)
ans = 7.25741D+306

--> factorial(171)
ans = Inf
```

Phép chia cho số không sẽ báo lỗi ngay mà không trả lại kết quả `Inf`.

```
-->1/0
    !--error 27
Division by zero...
```

1.9 Lời chú thích

Ngoài những câu lệnh cấu thành chương trình, sẽ rất có ích nếu ta kèm thêm những lời chú thích để đưa thêm thông tin về chương trình. Hai dấu sổ chéo `//` ngăn cách lời chú thích với mã lệnh.

```
--> speed_of_light = 3.0e8     // mét trên giây
speed_of_light = 300000000.
```

[†]Các tên biến này dễ gây nhầm lẫn; số có dấu phẩy động đôi khi được gọi nhầm là "real" (số thực).

Lời chú thích chạy từ dấu phần trăm về cuối dòng. Ở trường hợp trên nó giải thích về đơn vị của giá trị. Bạn có thể tưởng tượng rằng Scilab sẽ giúp việc theo dõi các đơn vị và thao tác với chúng qua từng phép tính, nhưng thật ra gánh nặng đó được đặt lên vai người lập trình.

Lời chú thích không ảnh hưởng đến việc thực thi chương trình. Chúng chỉ dành cho người đọc. Những lời chú thích hợp lý sẽ làm chương trình dễ đọc hơn, nhưng chú thích dở thì vô dụng hoặc (còn tệ hơn nữa) có thể gây nhầm lẫn.

Hãy tránh việc đặt những lời chú thích thừa:

```
--> x = 5        // gán giá trị 5 cho x
```

Những lời chú thích hay phải bổ sung thông tin vốn không có sẵn trong câu lệnh, như ở ví dụ trên, phải nói về ý nghĩa của biến:

```
--> p = 0         // vị trí từ gốc tọa độ tính theo mét
--> v = 100       // vận tốc tính theo mét / giây
--> a = -9.8      // gia tốc trọng trường tính theo mét/giây^2
```

Nếu bạn dùng các tên biến dài thì bạn có thể cũng không cần những lời chú thích như vậy, nhưng điều này lại bất tiện ở chỗ: câu lệnh dài sẽ khó đọc hơn. Ngoài ra, nếu bạn chuyển từ biểu thức toán vốn dùng tên biến ngắn thì chương trình bạn nên thống nhất với công thức toán học.

1.10 Thuật ngữ

trình thông dịch: Chương trình làm nhiệm vụ đọc và thực thi mã lệnh Scilab.

mã lệnh: Dòng lệnh Scilab được thực thi bởi trình thông dịch.

dấu nhắc: Ký hiệu mà trình thông dịch in ra để chỉ rằng nó đang đợi bạn gõ vào một câu lệnh.

toán tử: Một trong các kí hiệu, như `*` và +, để biểu thị cho các phép toán.

toán hạng: Một số hoặc một biến xuất hiện trong biểu thức bên cạnh các toán tử.

biểu thức: Dãy các toán hạng và toán tử để biểu thị một phép toán và trả lại một giá trị.

giá trị: Kết quả số của một phép tính.

lượng giá: Tính giá trị của một biểu thức.

thứ tự tính toán: Quy tắc chỉ định những phép toán nào trong một biểu thức sẽ được thực hiện trước.

hàm: Một phép tính được đặt tên; chẳng hạn `log10` là tên hàm dùng để tính loga cơ số 10.

gọi: Để hàm được thực thi và tính một kết quả.

lời gọi hàm: Dạng câu lệnh để thực thi một hàm.

đối số: Biểu thức xuất hiện trong một lời gọi hàm để chỉ định các giá trị mà hàm thao tác với.

lời gọi hàm lồng ghép: Biểu thức trong đó dùng kết quả của một lời gọi hàm làm đối số cho một lời gọi hàm khác.

biến: Một giá trị được đặt tên.

lệnh gán: Lệnh tạo ra một biến mới (nếu cần) và cho nó một giá trị.

chuỗi: Giá trị bao gồm một dãy các kí tự (đối ngược với một con số).

dấu phẩy động: Kiểu số mà Scilab sử dụng. Tất cả các số có dấu phẩy động đều biểu diễn được với khoảng 16 chữ số trong phần thập phân (khác với các số nguyên và số thực trong toán học).

cách viết khoa học: Một dạng viết và biểu thị các số lớn và nhỏ; chẳng hạn `3.0e8` để biểu thị cho 3.0×10^8 hay 300,000,000.

lời chú thích: Phần của chương trình nhằm cung cấp thêm thông tin về chương trình, nhưng không ảnh hưởng đến việc thực thi nó.

1.11 Bài tập

Exercise 1.1 *Hãy viết một biểu thức Scilab để lượng giá biểu thức toán sau đây. Bạn có thể coi rằng các biến* `mu`*,* `sigma` *và* `x` *đều đã tồn tại.*

$$\frac{e^{-\left(\frac{x-\mu}{\sigma\sqrt{2}}\right)^2}}{\sigma\sqrt{2\pi}} \tag{1.1}$$

Lưu ý: bạn không thể dùng những chữ cái Hi Lạp trong Scilab; khi chuyển từ biểu thức toán có chứa chữ cái Hi Lạp, ta thường viết hẳn tên nó (coi như bạn đã biết tên các chữ cái này).

Chương 2

Mã lệnh chương trình

2.1 Tập tin lệnh

Đến giờ ta đã gõ tất cả chương trình "vào dấu nhắc lệnh". Điều này cũng ổn nếu như bạn chỉ phải viết một vài dòng lệnh. Vượt quá mức đó, bạn sẽ cần lưu chương trình vào một **tập tin lệnh** rồi thực hiện tập tin lệnh này.

Một tập tin lệnh là một file (tập tin) chứa mã lệnh Scilab, và có hai dạng thông dụng:

.sce là tập tin sẽ được thực thi sau khi Scilab tải chúng. Tác dụng của tập tin này cũng tương tự như ta gõ một loạt câu lệnh vào dấu nhắc.

.sci là tập tin mà Scilab chỉ nạp vào; bao giờ cần đến sẽ thực hiện. Tập tin loại này thường gồm nhiều hàm nhỏ.

Bạn có thể tạo và sửa các tập tin lệnh với bất kì phần mềm biên tập file chữ (text editor) hay trình soạn thảo văn bản nào, nhưng cách làm dễ nhất là chọn trình đơn Application→SciNotes (hoặc kích chuột vào biểu tượng Launch SciNote). Một cửa sổ sẽ xuất hiện trong đó chạy một trình biên tập file chữ dành riêng cho Scilab.

Hãy gõ dòng lệnh sau vào trong trình biên tập

```
x = 5
```

và ấn vào biểu tượng đĩa mềm (giờ đã lỗi thời), hoặc chọn Save từ trình đơn File. Dù bằng cách nào đi nữa, một hộp thoại sẽ xuất hiện tại đó bạn có thể chọn tên

tập tin và thư mục cần lưu vào. Hãy đổi tên thành `myscript.sce` và giữ nguyên thư mục.

Bạn có thể chọn trình đơn Execute→...file with echo (hoặc ấn Ctrl+L) và máy sẽ thực hiện chạy chương trình trong cửa sổ Console.

```
-->exec('C:\Users\Admin\Documents\test1.sce', -1)
 -->x = 5
 x  =

    5.
```

Khi bạn chạy một tập tin lệnh, Scilab thực hiện các lệnh trong tập tin M, lần lượt từng lệnh một, hệt như khi bạn gõ chúng từ dấu nhắc.

Exercise 2.1 *Dãy Fibonacci, kí hiệu F, được mô tả bởi các phương trình $F_1 = 1$, $F_2 = 1$, và với $i \geq 3$, $F_i = F_{i-1} + F_{i-2}$. Các số trong dãy này thường xuất hiện trong tự nhiên ở nhiều loại cây, đặc biệt là ở những cánh hoa hay vẩy được sắp xếp theo hình thù xoáy ốc.*

Biểu thức sau được dùng để tính số Fibonacci thứ n:

$$F_n = \frac{1}{\sqrt{5}} \left[\left(\frac{1+\sqrt{5}}{2} \right)^n - \left(\frac{1-\sqrt{5}}{2} \right)^n \right] \tag{2.1}$$

Hãy chuyển biểu thức này sang Scilab và lưu mã lệnh vào một tập tin có tên là `fibonacci1`. *Tại dấu nhắc lệnh, hãy đặt* `n` *bằng 10 và chạy đoạn mã. Dòng cuối cùng của đoạn mã cần phải gán giá trị của F_n cho* `ans`. *(Giá trị đúng của F_{10} là 55).*

2.2 Tại sao cần dùng tập tin lệnh?

Những lý do thông thường nhất cho việc dùng mã lệnh là:

- Khi bạn đang viết nhiều câu lệnh (nhiều hơn một vài dòng), có thể bạn cần thử vài lần trước khi mã lệnh chạy đúng. Việc đặt mã vào trong một tập tin lệnh sẽ giúp bạn dễ chỉnh sửa hơn là gõ lệnh từ dấu nhắc.

  Mặt khác, bạn có thể thấy khó khăn khi phải chuyển đổi qua lại giữa Console và SciNotes (trình biên tập). Hãy thử sắp xếp các cửa sổ sao cho bạn có thể đồng thời thấy được cả SciNotes và Console, và dùng phím Alt+Tab hoặc chuột để chuyển giữa chúng.

- Nếu bạn chọn tên hợp lý cho tập tin lệnh, bạn sẽ nhớ được là tập tin nào làm nhiệm vụ gì, và bạn có thể sẽ sử dụng lại được một tập tin lệnh của dự án này cho dự án sau.
- Nếu bạn chạy tập tin lệnh nhiều lần, việc gõ tên tập tin lệnh sẽ nhanh hơn là gõ lại toàn bộ mã lệnh!

Không may là sức mạnh của các tập tin lệnh cũng đi kèm với trách nhiệm của người dùng; bạn phải chắc rằng mình chạy đúng tập tin lệnh mà mình cần.

Thứ nhất, mỗi khi chỉnh sửa tập tin lệnh, bạn phải lưu nó lại trước khi chạy. Nếu quên không lưu, bạn sẽ chạy phiên bản cũ của tập tin.

Thứ hai, mỗi khi bạn tạo một tập tin lệnh mới, hãy bắt đầu viết đơn giản, kiểu như `x=5`, để có được kết quả hiện ra rõ ràng. Sau đó chạy tập tin lệnh để chắc rằng bạn nhận được kết quả như mong đợi. Scilab có rất nhiều hàm định nghĩa sẵn. Rất dễ viết một tập tin lệnh có tên giống như tên hàm của Scilab, và nếu không cẩn thận, bạn có thể thấy rằng mình đã chạy hàm của Scilab thay vì tập tin lệnh vừa viết.

Dù trong trường hợp nòa, nếu mã lệnh mà bạn chạy không phải là mã lệnh bạn vừa sửa đổi thì bạn sẽ thấy việc gỡ rỗi thật phát bực! Và điều này dẫn ta đến Định lý thứ ba về gỡ lỗi:

> Bạn phải chắc chắn 100% rằng mã lệnh bạn đang chạy đúng là mã lệnh bạn muốn chạy.

2.3 Không gian làm việc

Các biến bạn vừa tạo ra được lưu vào trong một **không gian làm việc**, hay "workspace", vốn là một tập hợp các biến cùng giá trị của chúng. Lệnh `who` in ra các tên biến có trong không gian này.

```
--> x=5;
--> y=7;
--> z=9;
--> who

Your variables are:

z  y  x
```

```
...
...
```

Lệnh `clear` các biến.

```
--> clear y
--> who

Your variables are:

z  x
...
...
```

Để hiển thị giá trị một biến, bạn có thể dùng hàm `disp`.

```
--> disp(z)
     9.
```

Nhưng sẽ dễ hơn nếu ta chỉ gõ tên biến.

```
--> z
z = 9.
```

(Chặt chẽ mà nói thì tên biến cũng chính là một biểu thức, vì vậy việc lượng giá nó sẽ gán giá trị cho `ans`, nhưng dường như Scilab hiểu điều này như một trường hợp đặc biệt.)

2.4 Các lỗi khác

Một lần nữa, khi thử điều gì mới, bạn nên cố ý tạo ra một số lỗi để sau này còn nhận ra chúng.

Lỗi thông thường nhất với các tập tin lệnh là chạy một tập tin mà không tạo trước các biến cần thiết. Chẳng hạn, `fibonacci1` yêu cầu bạn gán một giá trị cho `n`. Nếu bạn không gán:

```
--> fibonacci1
??? Undefined function or variable "n".

Error in ==> fibonacci1 at 4
diff = t1^(n+1) - t2^(n+1);
```

Chi tiết của thông báo lệnh này có thể sẽ khác trong trường hợp của bạn, tùy theo nội dung mã lệnh bạn gõ vào tập tin. Nhưng ý tưởng chung là `n` chưa được định nghĩa. Lưu ý rằng Scilab báo với bạn dòng lệnh trong chương trình có lỗi xảy ra, và hiển thị dòng đó.

Thông tin này có thể hữu ích, nhưng hãy cẩn thận! Scilab báo với bạn chỗ phát hiện ra trục trặc, chứ không phải là vị trí của lỗi. Ở trường hợp này, lỗi không hề nằm ở tập tin lệnh; mà đúng ra là ở không gian làm việc.

Từ đó dẫn đến Định lý thứ tư về gỡ lỗi:

> Các thông báo lỗi báo cho ta biết trục trặc được phát hiện ở đâu, chứ không phải là nơi khởi nguồn của nó.

Mục đích của ta là tìm ra nguyên nhân và sửa nó—chứ không phải chỉ là làm cho thông báo lỗi biến đi.

2.5 Các điều kiện trước và sau

Mỗi tập tin lệnh đều nên chứa một lời chú thích nhằm trình bày tác dụng của nó, và những yêu cầu của nó đối với không gian làm việc. Chẳng hạn, tôi có thể gõ những dòng sau vào đầu tập tin `fibonacci1`:

```
// Tính số Fibonacci thứ n.
// Điều kiện trước: bạn phải gán một giá trị trước khi chạy
// đoạn mã lệnh này.  Điều kiện sau: kết quả được lưu vào ans.
```

Một **điều kiện trước** (“precondition”) là điều buộc phải đúng lúc chương trình bắt đầu được thực hiện, để chương trình có thể chạy đúng. Một **điều kiện sau** (“postcondition”) là điều sẽ đúng sau khi chương trình kết thúc.

2.6 Phép gán và đẳng thức

Trong toán học, dấu bằng dùng để chỉ hai vế của phương trình có cùng giá trị. Trong Scilab một phép gán *trông* giống như một đẳng thức toán học, nhưng thực ra thì không phải.

Một điểm khác biệt là hai vế của một phép gán thì không thể đổi chỗ cho nhau được. Vế phải có thể được thay bởi một biểu thức hợp lệ bất kì, nhưng vế trái thì nhất thiết là một biến, được gọi là **đích** của phép gán. Vì vậy các lệnh gán sau đều hợp lệ:

```
--> y = 1;
--> x = y+1
x = 2.
```

Nhưng lệnh gán sau thì không:

```
--> y + 1 = x
   !--error 4
Undefined variable: y
```

Máy thông báo lỗi: biến y chưa xác định. Nói chung, máy sẽ báo lỗi và cảnh báo theo nhiều cách khác nhau tùy từng trường hợp. Bạn đừng bao giờ viết lệnh mà vế trái không phải là một biến.

Một điểm khác biệt nữa là ở chỗ phép gán chỉ là tạm thời, theo nghĩa sau đây. Khi bạn gán `x = y+1`, bạn nhận được giá trị *hiện thời* của `y`. Nếu sau này `y` thay đổi, `x` sẽ không thay đổi theo.

Điểm khác biệt thứ ba là một đẳng thức toán là một phát biểu có thể đúng hoặc không đúng. Chẳng hạn, $y = y + 1$ là một phát biểu sai với mọi giá trị của y. Trong Scilab, `y = y+1` là câu lệnh gán hợp lệ và có ích. Nó đọc vào giá trị hiện thời của `y`, tăng thêm một, và thay thế giá trị cũ với giá trị mới này.

```
--> y = 1;
--> y = y+1
y = 2.
```

Khi đọc mã lệnh Scilab, bạn có thể thấy sẽ lợi hơn khi đọc dấu bằng là "nhận giá trị" thay vì "bằng." Do vậy `x = y+1` được đọc là "`x` nhận giá trị của `y` cộng với 1."

Để kiểm tra mức độ hiểu các lệnh gán của bạn, hãy thử làm bài tập sau:

Exercise 2.2 *Hãy viết một số dòng lệnh nhằm tráo đổi giá trị của hai biến* `x` *và* `y`*. Đặt mã lệnh bạn viết vào trong tập tin có tên là* `swap` *và chạy thử nó.*

2.7 Xây dựng dần

Khi bạn bắt đầu viết mã lệnh dài hơn một vài dòng, lúc đó bạn có thể thấy mình dành càng nhiều thời gian để gỡ lỗi. Nếu như bạn viết càng nhiều mã lệnh trước khi bắt tay vào việc gỡ lỗi thì bạn sẽ càng khó tìm ra trục trặc tiềm ẩn trong chương trình.

Xây dựng dần là một cách lập trình nhằm giảm thiểu công sức dành cho gỡ lỗi. Các bước cơ bản của nó gồm có:

1. Luôn bắt đầu với một chương trình chạy được. Nếu bạn có một ví dụ trong sách hoặc một chương trình mà bạn đã viết tương đồng với chương trình đang làm, thì hãy lấy nó để bắt đầu. Còn nếu không, hãy bắt đầu với điều mà bạn *biết* rằng luôn đúng, như `x=5`. Chạy chương trình và khẳng định chắc rằng bạn đang chạy chương trình mà bạn muốn chạy.

   Bước này rất quan trọng, vì ở đa số các môi trường [xây dựng chương trình], có rất nhiều điều nhỏ nhặt làm bạn rối lên mỗi khi bắt đầu một dự án mới. Hãy dẹp chúng qua một bên để có thể tập trung vào lập trình.

2. Mỗi lúc chỉ sửa một chỗ, và có thể kiểm tra được chỗ sửa này. "Kiểm tra được" có nghĩa ảnh hưởng của việc sửa đổi có thể hiện trên màn hình và bạn kiểm tra được. Tốt nhất là bạn cần biết được rằng kết quả đúng là gì, hoặc có khả năng kiểm tra nó bằng một phép tính toán khác.

3. Chạy chương trình xem sự thay đổi có hiệu quả không. Nếu có, hãy quay trở lại Bước 2. Nếu không, bạn cần phải gỡ lỗi, nhưng nếu sự thay đổi nói trên rất nhỏ thì thường bạn sẽ nhanh chóng tìm ra lỗi.

Khi quá trình trên hoạt động tốt, bạn sẽ thấy rằng thường những thay đổi có tác dụng ngay lần đầu, hoặc sai lầm (nếu có) sẽ dễ thấy. Đó là một điều tốt, và dẫn đến Định lý thứ năm về gỡ lỗi:

> Cách gỡ lỗi tốt nhất là cách mà ở đó bạn không phải làm.

Trên thực tế, có hai vấn đề gắn với Xây dựng dần:

- Đôi khi bạn phải viết thêm mã lệnh để có thể tạo ra kết quả dưới dạng nhìn thấy được, giúp cho việc kiểm tra. Mã lệnh thêm vào này được gọi là **dàn giáo** vì bạn dùng nó để xây dựng chương trình nhưng sau này sẽ bỏ nó đi khi chương trình hoàn tất. Nhưng thời gian tiết kiệm được từ việc gỡ lỗi thường luôn xứng đáng với thời gian bỏ ra để dựng dàn giáo.

- Khi bạn mới bắt đầu, thông thường sẽ không rõ bằng cách nào bạn có thể chọn các bước kế tiếp từ `x=5` đến chương trình mà bạn muốn viết. Có một ví dụ về cách làm này ở Mục 5.7 .

Nếu bạn tự thấy mình viết nhiều dòng lệnh trước khi bắt tay vào kiểm tra, và phải dành nhiều thời gian để gỡ lỗi thì bạn nên thử cách Xây dựng dần.

2.8 Kiểm tra thành phần

Trong những dự án phần mềm lớn, **kiểm tra thành phần** là quá trình kiểm tra những bộ phận riêng biệt cấu thành phần mềm, trước khi sắp xếp chúng lại.

Những chương trình ta viết đến giờ đều chưa đủ lớn đến mức phải kiểm tra thành phần, nhưng chính nguyên tắc này cũng có ích khi lần đầu bạn thao tác với một hàm mới hoặc một đặc điểm mới của ngôn ngữ. Bạn cần kiểm tra nó riêng biệt trước khi đưa vào chương trình.

Chẳng hạn, giả sử rằng bạn biết là `x` là sin của một góc nào đó và bạn muốn tính góc này. Bạn tìm thấy hàm Scilab có tên `asin`, và tương đối chắc rằng nó được dùng để tính nghịch đảo của sin. "Tương đối chắc chắn" vẫn là chưa đủ; bạn phải tuyệt đối chắc chắn.

Vì ta đã biết $\sin 0 = 0$, ta có thể thử

```
--> asin(0)
ans = 0.
```

vốn là kết quả đúng. Hơn nữa, ta đã biết sin của góc 90 độ bằng 1, vì vậy nếu ta thử `asin(1)`, ta muốn kết quả bằng 90, phải không?

```
--> asin(1)
ans = 1.5707963
```

Ối! Chúng ta quên mất rằng các hàm lượng giác trong Scilab đều tính theo ra-đian, chứ không phải độ. Vì vậy đáp số đúng là $\pi/2$, và ta có thể khẳng định bằng cách chia kết quả cho `pi`:

```
--> asin(1) / %pi
ans = 0.5
```

Với cách kiểm tra thành phần như thế này, bạn không thực sự kiểm tra lỗi trong Scilab, mà kiểm tra cách hiểu của bạn. Nếu bạn mắc lỗi chỉ vì đã hiểu sai cách hoạt động của Scilab thì sẽ mất rất nhiều thời gian để tìm ra lỗi đó; vì khi nhìn vào mã lệnh bạn tưởng như nó đúng.

Từ đó dẫn đến Định lý thứ sáu về gỡ lỗi:

> Những lỗi tệ nhất không nằm ở mã lệnh mà ở trong đầu bạn.

2.9 Thuật ngữ

tập tin M: Tập tin có chứa một chương trình Scilab.

tập tin lệnh: Tập tin M có chứa một loạt các lệnh Scilab.

không gian làm việc: Tập hợp các biến cùng giá trị của chúng.

điều kiện trước: Điều mà buộc phải đúng khi chương trình bắt đầu chạy, để đảm bảo cho chương trình hoạt động đúng đắn.

điều kiện sau: Điều sẽ đúng khi chương trình hoàn tất.

đích: Biến ở vế trái của lệnh gán.

xây dựng dần: Cách lập trình thông qua việc tạo ra một loạt những thay đổi nhỏ có thể kiểm tra được.

dàn giáo: Mã lệnh được viết để phục vụ cho việc lập trình hoặc gỡ lỗi, nhưng không phải là một phần của sản phẩm chương trình.

kiểm tra thành phần: Quá trình kiểm tra phần mềm bằng việc kiểm tra mỗi thành phần một cách riêng biệt.

2.10 Bài tập

Exercise 2.3 *Hãy tưởng tượng rằng bạn là chủ sở hữu một công ty cho thuê xe hơi với hai địa điểm, Albany and Boston. Một số khách hàng của bạn thuê "một chiều", nghĩa là thuê xe lái từ Albany đến Boston, hoặc ngược lại. Sau một thời gian quan sát, bạn nhận thấy rằng mỗi tuần có 5% số xe đi từ Albany được trả ở Boston, và 3% số xe đi từ Boston được trả ở Albany. Vào đầu mỗi năm, có 150 xe ở mỗi trạm.*

Hãy viết một tập tin lệnh có tên `car_update` *để cập nhật số xe ở mỗi trạm theo từng tuần. Điều kiện đầu là các biến* `a` *và* `b` *chứa số xe ở mỗi địa điểm vào đầu hàng tuần. Điều kiện cuối là* `a` *và* `b` *sau khi thay đổi, phản ánh số xe đã di chuyển.*

Để kiểm tra chương trình, hãy đặt các giá trị đầu cho `a` *và* `b` *tại dấu nhắc lệnh và chạy tập tin lệnh. Chương trình cần hiển thị các giá trị được cập nhật của* `a` *và* `b`*, nhưng không phải các biến trung gian khác.*

Lưu ý rằng các xe là lượng đếm được, vì vậy `a` *và* `b` *phải luôn là những giá trị nguyên. Bạn có thể sẽ cần dùng hàm* `round` *để tính số xe di chuyển trong mỗi tuần.*

Nếu thực hiện tập tin lệnh lặp đi lặp lại, bạn có thể mô phỏng sự di chuyển của xe từ tuần này qua tuần khác. Bạn nghĩ điều gì sẽ xảy ra với số xe? Liệu rằng tất cả các xe sẽ tụ về một trạm không? Liệu số xe sẽ đạt tới trạng thái cân bằng, hay dao động từ tuần này qua tuần khác?

Ở chương tiếp theo ta sẽ đề cập đến cách tự động thực hiện tập tin lệnh, này và cách vẽ đồ thị các giá trị của `a` *và* `b` *theo thời gian.*

Chương 3

Vòng lặp

3.1 Cập nhật các biến

Ở Bài tập 2.3 , bạn có thể đã định viết

```
a = a - 0.05*a + 0.03*b
b = b + 0.05*a - 0.03*b
```

Nhưng điều đó sai, rất sai. Tại sao? Vấn đề là ở chỗ dòng lệnh thứ nhất thay đổi giá trị của `a`, nên khi dòng lệnh thứ hai được thực hiện, nó sẽ lấy giá trị cũ của `b` và giá trị mới của `a`. Kết quả là, sự thay đổi ở `a` không cùng lúc với thay đổi ở `b`; tức là đã vi phạm định luật bảo toàn số lượng xe!

Một cách làm là dùng các biến tạm thời, `anew` và `bnew`:

```
anew = a - 0.05*a + 0.03*b
bnew = b + 0.05*a - 0.03*b
a = anew
b = bnew
```

Cách này có tác dụng cập nhật các biến "đồng thời"; nghĩa là nó đọc cả hai biến cũ trước khi ghi ra hai giá trị mới.

Dưới đây là một cách làm khác có lợi là làm việc tính toán được đơn giản:

```
atob = 0.05*a - 0.03*b
a = a - atob
b = b + atob
```

Xem xét đoạn mã này ta có thể thấy được nó tuân theo định luật bảo toàn số xe. Ngay cả khi giá trị của `atob` là sai thì ít nhất tổng số xe vẫn còn đúng. Và từ đó dẫn đến Định luật thứ bẩy về gỡ lỗi:

> Cách tốt nhất để tránh một lỗi là khiến nó không thể xảy ra.

Trong trường hợp này, việc bỏ những chi tiết thừa cũng loại trừ khả năng gây lỗi.

3.2 Các loại lỗi

Có bốn loại lỗi sau:

Lỗi cú pháp: Bạn đã viết một câu lệnh Scilab mà không thể thực thi được vì nó vi phạm các quy tắc về cú pháp. Chẳng hạn, bạn không thể có hai toán hạng đi liền nhau mà không có toán tử, vì vậy `%pi r^2` là một lỗi cú pháp. Khi Scilab phát hiện ra lỗi cú pháp, nó sẽ hiển thị một thông báo lỗi và dừng chương trình.

Lỗi thực thi: Chương trình của bạn đã bắt đầu chạy, nhưng rồi có điều gì trục trặc diễn ra. Chẳng hạn, nếu bạn cố gắng truy cập một biến chưa tồn tại thì đó là một lỗi thực thi. Khi Scilab phát hiện được vấn đề, nó sẽ in ra thông báo lỗi và dừng lại.

Lỗi logic: Chương trình chạy mà không phát sinh bất cứ thông báo lỗi nào, nhưng nó không thực hiện điều mong muốn. Vấn đề ta gặp ở mục trước, khi thay đổi giá trị của `a` trước lúc đọc giá trị cũ, là một lỗi logic.

Lỗi số trị: Hầu hết những phép tính được thực hiện bởi Scilab đều chỉ gần đúng. Trong đa số trường hợp, sai số là nhỏ và ta không quan tâm đến, nhưng đôi khi các sai số do làm tròn lại là một vấn đề.

Các lỗi cú pháp luôn dễ xử lý nhất. Đôi khi dòng thông báo lỗi có thể gây nhầm lẫn, nhưng thường Scilab sẽ báo cho bạn biết lỗi ở đâu, ít ra là một vị trí gần đúng.

Các lỗi thực thi nói chung là khó hơn vì như tôi đã đề cập ở trên, Scilab nói được vị trí của nó nhưng không nói nguyên nhân gây ra nó.

Các lỗi logic đều khó vì Scilab chẳng giúp gì được. Chỉ có bạn mới biết được chương trình cần phải làm gì, vì vậy chỉ có bạn mới sửa được lỗi. Theo quan điểm của Scilab, nếu chương trình không có gì sai thì lỗi nằm trong đầu bạn!

Các lỗi số trị có thể sẽ rất mẹo mực vì thật không rõ là cái sai có thuộc về bạn hay không. Với những tính toán đơn giản nhất, Scilab cho ta các giá trị số có dấu phẩy động gần sát với giá trị đúng, có nghĩa là 15 chữ số ban đầu là đúng. Nhưng trong một số bài toán với đặc thù "tình trạng xấu", có nghĩa là ngay cả khi chương trình của bạn đã đúng, các sai số do làm tròn vẫn tích tụ lại và số chữ số đúng sẽ ít đi. Đôi khi Scilab có thể cảnh báo cho bạn biết điều này đang xảy ra, nhưng không phải luôn luôn như vậy! Độ chuẩn xác (số chữ số trong kết quả) không bao hàm độ chính xác (số chữ số đúng).

3.3 Sai số tuyệt đối và tương đối

Có hai cách nghĩ về các sai số về số trị, đó là **tuyệt đối** và **tương đối**.

Sai số tuyệt đối chính là độ chênh lệch giữa giá trị đúng và giá trị xấp xỉ. Ta thường biểu thị độ lớn của sai số này, mà bỏ qua dấu của nó, vì dù giá trị xấp xỉ có cao hay thấp thì cũng chẳng ảnh hưởng gì.

Chẳng hạn, ta muốn tính $9!$ bằng công thức $\sqrt{18\pi}(9/e)^9$. Đáp số đúng là $9 \cdot 8 \cdot 7 \cdot 6 \cdot 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1 = 362.880$. Giá trị xấp xỉ tính được là $359.536,87$. Sai số tuyệt đối là 3.343,13.

Thoạt nhìn, ta tưởng chừng như đây là sai số lớn—ta đã tính sai đến hơn 3000—nhưng cũng cần phải xét đến độ lớn của đại lượng mà ta cần tính. Chẳng hạn, $3000 là con số lớn khi nói về tiền lương năm của tôi, nhưng chẳng là gì cả nếu ta nói về khoản nợ của quốc gia.

Một cách tự nhiên để xử lý ván đề này là dùng sai số tương đối, vốn là tỉ lệ (hay số phần trăm) của sai số tuyệt đối so với giá trị đúng. Trong trường hợp này, ta cần chia sai số cho 362.880, thu được $0,00921$, tức là gần 1%. Với phần lớn các mục đích tính toán khác nhau, thì sai lệch 1% là đạt yêu cầu.

3.4 Vòng lặp for

Vòng lặp là một phần chương trình được thực hiện lặp đi lặp lại. **Vòng lặp for** là dạng vòng lặp có dùng câu lệnh `for`.

Cách dùng vòng lặp `for` đơn giản nhất là thực hiện lặp lại một hay nhiều câu lệnh với số lần định trước. Chẳng hạn, ở chương trước ta đã viết một đoạn mã

có tên `car_update` để mô phỏng diễn biến chạy trong một tuần của những xe thuộc một công ty. Để mô phỏng diễn biến trong một năm, ta cần phải chạy nó 52 lần:

```
for i=1:52
    car_update
end
```

Dòng thứ nhất giống như một lệnh gán, và nó *đúng là* một kiểu lệnh gán, nhưng được thực hiện nhiều lần. Lần đầu tiên được chạy, nó tạo ra biến `i` và gán cho nó giá trị bằng 1. Lần thứ hai, `i` nhận giá trị 2, và cứ như vậy cho đến 52.

Toán tử hai chấm, `:`, biểu diễn một **khoảng** số nguyên. Theo tinh thần của cách kiểm tra từng phần, bạn có thể tạo ra một dãy từ dấu nhắc lệnh:

```
--> 1:5
ans =  1.     2.     3.     4.     5.
```

Biến được dùng trong lệnh for này được gọi là **biến vòng lặp**. Theo thông lệ, ta thường lấy các tên `i`, `j` và `k` để đặt cho các biến lặp.

Những câu lệnh bên trong vòng lặp được gọi là **phần thân**. Theo thông lệ, chúng được viết thụt đầu dòng để cho thấy rằng chúng nằm trong vòng lặp, tuy nhiên hình thức viết này không ảnh hưởng đến việc thực hiện chương trình. Điểm kết thúc của vòng lặp được chính thức đánh dấu bởi lệnh `end`.

Để xem vòng lặp hoạt động thế nào, bạn có thể chạy vòng lặp trong đó chỉ in ra biến lặp:

```
--> for i=1:5
    i
end

i = 1.
i = 2.
i = 3.
i = 4.
i = 5.
```

Như ví dụ trên cho thấy, bạn *có thể* chạy một vòng lặp từ dấu nhắc lệnh, nhưng ta thường đặt nó vào một tập tin lệnh hơn.

Exercise 3.1 *Hãy tạo ra một tập tin lệnh có tên* `car_loop` *trong đó dùng một vong lặp* `for` *để chạy* `car_update` *52 lần. Hãy nhớ rằng trước khi chạy* `car_update`*, bạn phải gán các giá trị cho* `a` *và* `b`*. Với bài tập này, hãy bắt đầu bằng các giá trị* `a = 150` *và* `b = 150`.

Nếu mọi việc trôi chảy, chương trình của bạn sẽ hiển thị một đoạn dài các con số trên màn hình. Nhưng thường có quá nhiều số để màn hình hiện ra hết; và ngay cả có hiện hết đi nữa cũng rất khó diễn giải được. Có một đồ thị sẽ tốt hơn!

3.5 Đồ thị

`plot` là một hàm vẽ đồ thị rất đa năng, giúp ta vẽ các điểm, các đường trên hệ tọa độ hai chiều. Thật không may, vì quá đa năng nên nó có thể trở nên khó dùng. (và khó tra cứu thông tin về hàm này!) Ta sẽ bắt đầu một cách đơn giản và dần làm khó hơn.

Để chấm một điểm, ta gõ vào

```
--> plot(1, 2)
```

Một Graphic window (cửa sổ đồ họa) sẽ xuất hiện với một đồ thị trên đó có chấm một điểm màu xanh lam tại tọa độ x bằng 1 và y bằng 2. Để khiến cho điểm này dễ nhìn hơn, bạn có thể chọn một hình khác:

```
--> plot(1, 2, 'o')
```

Chữ cái ở trong cặp dấu nháy đơn là một chuỗi chỉ định hình thức của điểm cần chấm. Bạn cũng có thể chỉ định màu sắc:

```
--> plot(1, 2, 'ro')
```

`r` viết tắt cho red (đỏ); các màu khác gồm có **g**reen (lục), **b**lue (lam), **c**yan (da trời), **m**agenta (tím hồng), **y**ellow (vàng) và blac**k** (đen). Các hình khác gồm có `+`, `*`, `x`, `s` (square / hình vuông), `d` (diamond / hình thoi), và `^` (hình tam giác).

Bạn có thể dùng nhiều hàm plot để chấm các điểm. Hãy thử các lệnh này:

```
--> plot(1, 1, 'o')
--> plot(2, 2, 'o')
```

Bạn sẽ nhìn thấy một hình có hai điểm. Scilab co dãn tỉ lệ đồ thị một cách tự động sao cho các trục chạy từ giá trị nhỏ nhất trên đồ thị đến giá trị lớn nhất. Vì vậy ở ví dụ này, các điểm chấm xuất hiện ở hai góc.

Bạn có thể xóa hình vẽ bằng `clf`.

Exercise 3.2 *Hãy sửa lại* `car\_loop` *sao cho qua mỗi vòng lặp, chương trình sẽ chấm lên đồ thị giá trị của* `a` *theo* `i`.

Một khi chương trình của bạn chạy được, hãy sửa lại để nó chấm các giá trị của `a` *bằng vòng tròn đỏ và của* `b` *bằng hình thoi xanh lam.*

3.6 Dãy

Trong toán học, một **dãy** là một tập hợp các số tương ứng với các số nguyên dương. Các số trong dãy được gọi là **phần tử**. Theo kí hiệu toán học, các phần tử được kèm theo các chỉ số dưới, vì vậy phần tử đầu tiên của dãy A là A_1, tiếp theo là A_2, và cứ như vậy.

Vòng lặp `for` là một cách tự nhiên để tính các phần tử trong một dãy. Chẳng hạn, trong dãy hình học, mỗi phần tử là một bội số (với hệ số không đổi) của số liền trước. Cụ thể, hãy xét dãy số với $A_1 = 1$ và tỉ lệ $A_{i+1}/A_i = 2$, với mọi i. Nói cách khác, mỗi phần tử chỉ lớn bằng nửa phần tử liền trước nó.

Vòng lặp sau đây tính ra 10 phần tử đầu của A:

```
a = 1
for i=2:10
    a = a/2
end
```

Mỗi lượt lặp, ta tìm được phần giá trị tiếp theo của `a` bằng cách chia giá trị trước cho 2. Lưu ý rằng dãy chỉ số bắt đầu từ 2 vì giá trị đầu của `a` tương ứng với A_1, vì vậy lượt lặp đầu tiên ta đi tính A_2.

Mỗi lần qua vòng lặp, ta thay thế phần tử trước bởi phần tử kế tiếp, vì vậy về cuối, `a` chứa phần tử thứ 10. Các phần tử khác được hiển thị trên màn hình, nhưng chúng không được lưu lại trong một biến nào. Sau này, ta sẽ xem cách lưu toàn bộ các phần tử của dãy vào một véc-tơ.

Vòng lặp này tính dãy theo cách **truy hồi**, nghĩa là mỗi phần tử đều phụ thuộc vào phần tử liền trước nó. Với dạng dãy này ta cũng có thể tính **trực tiếp** phần tử thứ i, theo một hàm của i, mà không cần dùng đến phần tử đứng trước. Theo cách viết toán học, $A_i = A_1 r^{i-1}$.

Exercise 3.3 *Hãy viết một tập tin lệnh có tên* `sequence` *trong đó dùng vòng lặp để tính các phần tử của A một cách trực tiếp.*

3.7 Chuỗi

Trong toán học, **chuỗi** là tổng các phần tử của một dãy. Cách đặt tên này không hay trong tiếng Anh ("sequence" và "series" gần như có chung nghĩa); nhưng trong toán thì dãy là một tập hợp số, còn chuỗi lại là một biểu thức (một tổng) với một giá trị duy nhất. Theo kí hiệu toán học, một chuỗi thường được viết với dấu tổng $\sum$.

Chẳng hạn, tổng của 10 phần tử đầu tiên của A là

$$\sum_{i=1}^{10} A_i$$

Một vòng lặp `for` là cách tự nhiên để tính giá trị của chuỗi này:

```
A1 = 1;
total = 0;
for i=1:10
    a = A1 * 0.5^(i-1);
    total = total + a;
end
ans = total
```

`A1` là phần tử đầu tiên của dãy, vì vậy ở mỗi lượt lặp, `a` là phần tử thứ i.

Biến `total`, theo cách dùng ở đây, đôi khi được gọi là **biến tích lũy**; tức là một biến mà dồn lại lần lượt từng kết quả của các phép tính. Trước vòng lặp ta đặt biến này bằng 0. Mỗi lượt lặp ta cộng nó với phần tử thứ i. Ở cuối vòng lặp, `total` sẽ chứa tổng của các phần tử. Vì đó là giá trị mà ta cần tìm, ta gán nó cho `ans`.

Exercise 3.4 *Ví dụ trên đã tính các phần tử của chuỗi một cách trực tiếp; bạn hãy thử dùng SciNotes viết một tập tin lệnh tên là* `series` *để tính cùng tổng đó nhưng với từng phần tử được tính theo cách truy hồi. Bạn sẽ phải cẩn thận về các vị trí bắt đầu và kết thúc vòng lặp.*

3.8 Khái quát hóa

Như đã nói, ví dụ trên luôn luôn lấy tổng 10 phần tử đầu tiên của dãy, nhưng ta có thể tò mò muốn biết giá trị `total` sẽ như thế nào khi ta tăng số lượng các số hạng có trong chuỗi. Nếu bạn đã biết về chuỗi hình học, bạn thấy rằng chuỗi này hội tụ về 2; nghĩa là khi số các số hạng tiến đến vô cùng, thì tổng sẽ tiệm cận về 2.

Để thấy được liệu điều đó có đúng không, trong chương trình ta sẽ thay thế hằng số, 10, với một biến có tên `n`:

```
A1 = 1;
total = 0;
for i=1:n
    a = A1 * 0.5^(i-1);
    total = total + a;
end
ans = total
```

Bây giờ đoạn chương trình có thể tính với bao nhiêu số hạng cũng được, với điều kiện trước là bạn phải đặt `n` trước khi chạy chương trình này. Sau đây là cách chạy chương trình với các giá trị `n` khác nhau:

```
--> n=10;
--> (gõ Ctrl+L từ SciNotes)

total = 1.9980469

--> n=20
--> (gõ Ctrl+L từ SciNotes)

total = 1.9999981

--> n=30
--> (gõ Ctrl+L từ SciNotes)

total = 2.
```

Rõ ràng là nó hội tụ về 2.

Việc thay thế một hằng số bởi một biến được gọi là **khái quát hóa**. Thay vì tính với một số lượng cụ thể và cố định các số hạng, chương trình mới này tổng quát hơn; nó có thể tính với số các số hạng bất kì.

Đây là một ý tưởng quan trọng mà ta sẽ quay về khi thảo luận đến các hàm.

3.9 Thuật ngữ

sai số tuyệt đối: Độ chênh lệch giữa giá trị xấp xỉ và kết quả đúng.

sai số tương đối: Độ chênh lệch giữa giá trị xấp xỉ và kết quả đúng, biểu diễn dưới dạng một phần hoặc số phần trăm của giá trị đúng.

vòng lặp: Phần của chương trình được chạy đi chạy lại nhiều lần.

biến lặp: Biến được định nghĩa trong một câu lệnh `for`; nó được gán các giá trị khác nhau qua mỗi lượt lặp.

khoảng: Tập hợp các giá trị được gán cho một biến lặp, thường được biểu thị bởi toán tử hai chấm, chẳng hạn `1:5`.

phần thân: Những câu lệnh bên trong vòng lặp được thực hiện lặp lại nhiều lần.

dãy: Trong toán học, một tật hợp các số tương ứng với những số nguyên.

phần tử: Một thành viên của tập hợp các số trong dãy.

truy hồi: Cách tính phần tử kế tiếp trong dãy dựa trên những phần tử liền trước.

trực tiếp: Cách tính phần tử trong dãy mà không cần dùng đến những phần tử trước.

chuỗi: Tổng của các phần tử trong một dãy.

biến tích lũy: Biến được dùng để tích tụ kết quả từng ít một.

khái quát hóa: Cách làm chương trình linh hoạt hơn, chẳng hạn bằng việc thay thế một giá trị cụ thể bởi một biến có thể nhận giá trị bất kì.

3.10 Bài tập

Exercise 3.5 *Ta đã thấy các dãy Fibonacci, F, vốn được định nghĩa theo cách truy hồi như sau*

$$F_i = F_{i-1} + F_{i-2}$$

Để bắt đầu, bạn phải chỉ định hai phần tử đầu tiên, nhưng một khi có hai phần tử này rồi, bạn có thể tính toàn bộ các phần tử còn lại. Dãy Fibonacci thông dụng nhất khởi đầu với $F_1 = 1$ và $F_2 = 1$.

Hãy viết một đoạn mã lệnh có tên là `fibonacci2` *trong đó dùng một vòng lặp để tính 10 phần tử đầu tiên của dãy Fibonacci. Điều kiện cuối của chương trình là gán phần tử thứ 10 cho* `ans`.

Bây giờ hãy khái quát hóa chương trình để nó tính phần tử thứ n với `n` *bất kì, kèm theo điều kiện trước là bạn phải đặt giá trị cho* `n` *trước khi chạy chương trình. Để đơn giản, tạm thời ta giả sử rằng* `n` *lớn hơn 2.*

Gợi ý: bạn sẽ phải dùng hai biến để theo dõi hai phần tử liền trước của dãy. Có thể đặt tên chúng là `prev1` *và* `prev2`. *Ban đầu,* `prev1` $= F_1$ *còn* `prev2` $= F_2$. *Ở cuối mỗi lượt lặp, bạn sẽ phải cập nhật* `prev1` *và* `prev2`*; hãy tính cẩn thận thứ tự cập nhật!*

Exercise 3.6 *Hãy viết một tập tin lệnh có tên* `fib_plot` *trong đó lặp* `i` *từ 1 đến 20, dùng* `fibonacci2` *để tính số Fibonacci, rồi chấm F_i với mỗi i dưới dạng một loạt điểm tròn màu đỏ.*

Chương 4

Véc-tơ

4.1 Kiểm tra điều kiện trước

Một số vòng lặp ở chương trước sẽ không chạy đúng nếu giá trị của `n` không được đặt đúng trước khi vòng lặp bắt đầu chạy. Chẳng hạn, vòng lặp sau đây dùng để tính tổng của `n` phần tử đầu tiên của một dãy hình học:

```
A1 = 1;
total = 0;
for i=1:n
    a = A1 * 0.5^(i-1);
    total = total + a;
end
ans = total
```

Nó chạy đúng với bất kì giá trị dương nào của `n`, nhưng điều gì sẽ xảy ra nếu `n` âm? Lúc đó, bạn sẽ nhận được:

```
total = 0.
```

Vì sao? Bởi biểu thức `1:-1` nghĩa là "tất cả các số từ 1 đến -1, đếm theo chiều xuôi." Điều này không thật hiển nhiên, nhưng Scilab hiểu rằng chẳng có con số nào trong một khoảng như vậy, vì thế kết quả là

```
--> 1:-1

ans = []
```

Một cặp ngoặc vuông không chứa gì như vậy được gọi là ma trận rỗng. Trong mọi trường hợp, nếu bạn lặp qua một khoảng rỗng, thì vòng lặp không chạy chút nào; đó là lý do tại sao ở ví dụ này giá trị của `total` bằng không với mọi giá trị âm của `n`.

Nếu bạn chắc rằng mình chẳng bao giờ mắc lỗi, và các điều kiện trước cho các hàm viết ra luôn được thỏa mãn, thì bạn chẳng phải kiểm tra. Nhưng với hầu hết chúng ta, thật nguy hiểm khi viết một chương trình như thế này, một chương trình có thể cho kết quả sai (hoặc ít nhất là vô nghĩa) nếu giá trị đầu vào là số âm. Một cách làm tốt hơn là dùng lệnh `if`.

4.2 `if`

Lệnh `if` cho phép kiểm tra những điều kiện nhất định và thực hiện câu lệnh nếu điều kiện được thỏa mãn. Ở ví dụ trước, ta đã có thể viết:

```
if n<0
    ans = %nan
end
```

Cú pháp ở đây cũng tương tự một vòng lặp `for`. Dòng đầu tiên chỉ định điều kiện mà ta quan tâm đến; trong trường hợp này ta đang hỏi liệu `n` có âm không. Nếu có, Scilab thực hiện phần thân của câu lệnh, tức là các lệnh nằm giữa `if` và `end`.

Scilab không yêu cầu bạn viết thụt đầu dòng những lệnh trong phần thân của lệnh `if`, nhưng nó giúp cho mã lệnh viết ra dễ đọc hơn; và bạn nên làm điều này, có lẽ tôi cũng không phải nhắc nữa.

Ở ví dụ này, điều "đúng đắn" cần làm nếu `n` âm là đặt `ans = %nan`, vốn là một cách tiêu chuẩn để cho thấy rằng kết quả không được xác định (không phải một con số, Not a number).

Nếu điều kiện không được thỏa mãn thì những câu lệnh trong phần thân không được thực hiện. Đôi khi có những câu lệnh khác cần được thực hiện khi điều kiện sai. Trong trường hợp đó bạn có thể mở rộng lệnh `if` với một vế `else`.

Dạng hoàn chỉnh của chương trình ví dụ trên có thể trông như sau:

```
if n<0
    ans = %nan
else
    A1 = 1;
    total = 0;
```

```
    for i=1:n
        a = A1 * 0.5^(i-1);
        total = total + a;
    end
    ans = total
end
```

Các lệnh như `if` và `for` trong đó chứa những câu lệnh khác được gọi là lệnh **phức hợp**. Tất cả các lệnh phức hợp đều kết thúc với `end`.

Ở ví dụ này, một trong những câu lệnh trong vế `else` là một vòng lặp `for`. Việc đặt một câu lệnh phức hợp bên trong một lệnh phức hợp khác là hợp lệ và cũng thường gặp; cách này đôi khi được gọi là **lồng ghép**.

4.3 Toán tử quan hệ

Các toán tử để so sánh hai giá trị, như toán tử < và >, được gọi là **toán tử quan hệ** ví chúng kiểm tra mối tương quan giữa hai biến. Kết quả của một toán tử quan hệ là một trong hai **giá trị logic**: T (biểu diễn cho "đúng", True), hoặc F (biểu diễn cho "sai", False).

Các toán tử quan hệ thường xuất hiện trong lệnh `if`, nhưng bạn cũng có thể lượng giá chúng tại dấu nhắc lệnh:

```
--> x = 5;
--> x < 10

ans = T
```

Bạn cũng có thể gán một giá trị logic vào một biến:

```
--> flag = x > 10

flag = F
```

Một biến có chứa một giá trị logic thường được gọi là **cờ** vì chúng đánh dấu trạng thái của một điều kiện nào đó.

Các toán tử quan hệ khác gồm có <= và >=, vốn rất dễ hiểu, ==, nghĩa là "bằng," và ~=, nghĩa là "khác". (Trong kí hiệu logic, dấu "ngã" còn biểu thị cho "không.")

Đừng quên rằng == là toán tử để kiểm tra độ bằng nhau, còn = là toán tử gán. Nếu bạn thử dùng = trong câu lệnh `if`, bạn sẽ bị cảnh báo:

```
-->if x=5
Warning: obsolete use of '=' instead of '=='.
      !
```

Cách dùng một dấu bằng để so sánh đã lỗi thời. Bạn nên tập thói quen dùng hẳn hai dấu bằng trong phiên bản mới của Scilab.

4.4 Toán tử logic

Để kiểm tra rằng một số nào đó có rơi vào khoảng cho trước không, bạn có thể muốn viết `0 < x < 10`, nhưng đó là cách làm sai. Máy sẽ báo lỗi: không dùng được toán tử này; thực ra không phải vì ta viết dấu nhỏ hơn mà là do biểu thức không phù hợp.

```
-->0 < x < 10
          !--error 144
Undefined operation for the given operands.

check or define function %b_1_s for overloading.
```

Để thực hiện điều này ta dùng một cặp lệnh `if` lồng ghép để kiểm tra riêng hai vế điều kiện:

```
ans = 0
if 0<x
    if x<10
        ans = 1
    end
end
```

Nhưng sẽ gọn hơn nếu ta dùng toán tử AND (và), `&`, để kết hợp các điều kiện lại.

```
--> x = 5;
--> 0<x & x<10

ans = 1

--> x = 17;
```

```
--> 0<x & x<10

ans = 0
```

Kết quả của AND là đúng nếu *cả hai* toán hạng đều đúng. Toán tử OR (hoặc), `|`, sẽ đúng nếu *một trong hai, hoặc cả hai* toán hạng đúng.

4.5 Véc-tơ

Các giá trị mà ta đã thấy đến giờ đều là những con số đơn lẻ, và được gọi là **vô hướng** để phân biệt với **véc-tơ** và **ma trận**, vốn là các tập hợp số.

Một véc-tơ trong Scilab cũng giống như một dãy số trong toán học, đó là một tập hợp các số tương ứng với những số nguyên dương. Cái mà ta gọi là "khoảng" ở chương trước thực ra là một véc-tơ.

Nói chung, bất kể phép tính gì thực hiện được với số vô hướng đều cũng có thể thực hiện được với véc-tơ. Bạn có thể gán giá trị véc-tơ cho một biến:

```
--> X = 1:5

X = 1.    2.    3.    4.    5.
```

Các biến chứa véc-tơ thường được kí hiệu bằng chữ in. Đó chỉ là quy ước; Scilab không yêu cầu điều này, với nhưng người mới lập trình, đây là một cách hữu ích để nhớ ra đâu là số vô hướng và đâu là véc-tơ.

Cũng như với dãy, các số hợp thành một véc-tơ thì được gọi là các **phần tử**.

4.6 Phép toán số học với véc-tơ

Bạn có thể thực hiện phép tính số học với véc-tơ. Nếu bạn cộng một số vô hướng với một véc-tơ, Scilab sẽ cộng số vô hướng đo cho từng phần tử của véc-tơ:

```
--> Y = X+5

Y = 6.    7.    8.    9.    10.
```

Kết quả là một véc-tơ mới; giá trị ban đầu của `X` vẫn không đổi.

Nếu bạn cộng hai véc-tơ, Scilab sẽ cộng các phần tử tương ứng của mỗi véc-tơ và tạo ra một véc-tơ mới chứa các tổng:

```
--> Z = X+Y

Z = 7.     9.    11.    13.    15.
```

Nhưng phép cộng véc-tơ chỉ được khi hai véc-tơ số hạng có cùng kích cỡ. Nếu không thì:

```
--> W = 1:3

W = 1.     2.     3.

-->X+W
    !--error 8
Inconsistent addition.
```

Thông báo lỗi trong trường hợp này hơi lạc hướng, vì chúng ta đang nghĩ về các véc-tơ, chứ không phải ma trận. Vấn đề chỉ là một sự không khớp giữa từ vựng toán học và từ vựng trong Scilab.

4.7 Mọi thứ đều là ma trận

Trong toán học (đặc biệt là đại số tuyến tính), một véc-tơ là một dãy một chiều các giá trị còn ma trận có hai chiều (và nếu bạn muốn nghĩ theo cách đó thì số vô hướng là đại lượng không chiều). Trong Scilab, mọi thứ đều là ma trận.

Bạn có thể thấy điều này khi dùng lệnh `whos` để hiển thị các biến trong không gian làm việc. `whos` cũng giống như `who`, chỉ khác ở chỗ nó còn hiển thị cả kích cỡ và kiểu của từng biến.

Trước hết tôi sẽ tạo ra các giá trị có kiểu khác nhau:

```
--> scalar = 5

scalar = 5.

--> vector = 1:5

vector = 1.     2.     3.     4.     5.

--> matrix = ones(2,3)
```

```
matrix =

     1.     1.     1.
     1.     1.     1.
```

`ones` là một hàm để lập một ma trận mới với số hàng và số cột đã cho, rồi đặt tất cả các phần tử bằng 1. Bây giờ hãy xem ta có gì.

```
--> whos
Name                        Type            Size            Bytes

$                           polynomial      1 by 1          56
%driverName                 string*         1 by 1          40
%e                          constant        1 by 1          24
...
```

Kết quả có quá nhiều biến lập sẵn trong Scilab nhưng ta vẫn có thể nhặt ra được:

```
matrix                      constant        2 by 3          64
scalar                      constant        1 by 1          24
vector                      constant        1 by 5          56
```

Theo Scilab, mọi thứ đều là mảng. Khác biệt duy nhất giữa chúng là kích cỡ, vốn được chỉ định bởi số hàng và số cột. Biến số vô hướng, `scalar`, theo Scilab, là một ma trận có 1 hàng và 1 cột. Còn `vector` là một ma trận thực sự với 1 hàng và 5 cột. Và tất nhiên, `matrix` là một ma trận.

Mục đích của tất cả những điều trên là bạn có thể coi các giá trị trong chương trình như là số vô hướng, véc-tơ và ma trận, và tôi nghĩ rằng bạn nên như vậy, chỉ cần nhớ rằng Scilab coi mọi thứ như ma trận.

Sau đây là một ví dụ khác trong đó thông báo lỗi chỉ có nghĩa nếu bạn hiểu được điều gì đang xảy ra bên trong:

```
--> X = 1:5

X = 1     2     3     4     5

--> Y = 1:5
```

```
Y = 1     2     3     4     5

-->Z = X*Y
         !--error 10
Inconsistent multiplication.
```

Phép nhân không thống nhất. Lí do: để nhân được thì các kích thước cạnh nhau của hai ma trận thừa số thì phải tương đồng. Theo nguyên tắc định nghĩa phép nhân ma trận trong đại số tuyến tính, cột của `X` phải bằng số hàng của `Y`.

Nếu bạn không biết đại số tuyến tính thì cũng không sao. Khi nhìn thấy `X*Y` có lẽ bạn mong đợi rằng từng phần tử của `X` phải được nhân với từng phần tử của `Y` và kết quả được đưa vào một véc-tơ mới. Phép toán đó được gọi là phép nhân **phần tử**, và toán tử thực hiện công việc này là `.*`:

```
--> X .* Y

ans = 1.     4.     9.    16.    25.
```

Ta sẽ trở lại các toán tử thao tác trên phần tử vào một dịp khác; bạn có thể tạm quên chúng.

4.8 Chỉ số

Bạn có thể chọn phần tử trong một véc-tơ bằng cách dùng cặp ngoặc tròn:

```
--> Y = 6:10

Y = 6.     7.     8.     9.    10.

--> Y(1)

ans = 6.

--> Y(5)

ans = 10.
```

Điều này nghĩa là phần tử thứ nhất của `Y` bằng 6 và phần tử thứ năm bằng 10. Con số trong ngoặc tròn được gọi là **chỉ số** vì nó biểu thị phần tử mà bạn muốn của véc-tơ.

Chỉ số có thể là một biểu thức bất kì.

```
--> i = 1;

--> Y(i+1)

ans = 7.
```

Vòng lặp đi cùng véc-tơ cũng tựa như hổ mọc thêm cánh. Chẳng hạn, vòng lặp này hiển thị các phần tử của `Y`.

```
for i=1:5
     Y(i)
end
```

Mỗi lượt lặp ta dùng một giá trị khác của `i` làm chỉ số cho `Y`.

Một hạn chế trong ví dụ này là nếu ta phải biết trước số phần tử của `Y`. Ta có thể khái quát hóa bằng cách dùng hàm `length`, vốn để trả lại số phần tử trong một véc-tơ.

```
for i=1:length(Y)
     Y(i)
end
```

Vậy đấy. Bây giờ nó sẽ chạy được với một véc-tơ có độ dài tùy ý.

4.9 Lỗi chỉ số

Một chỉ số có thể là biểu thức bất kì, nhưng giá trị của biểu thức này phải là số nguyên dương, và nó phải nhỏ hơn hoặc bằng chiều dài của véc-tơ. Nếu nó bằng không hoặc âm, bạn sẽ nhận được thông báo sau:

```
-->Y(0)
     !--error 21
Invalid index.
```

Chỉ số không hợp lệ. Kể cả trường hợp chỉ số quá lớn cũng phát sinh lỗi tương tự.

Điều thú vị xảy ra khi bạn

```
-->Y(1.5)
 ans  =    1.
```

Xem ra Scilab đã làm tròn chỉ số xuống thành 1. Để khẳng định là nó không làm tròn lên số nguyên gần nhất, hãy thử với 1.9 thay vì 1.5.

4.10 Véc-tơ và dãy số

Véc-tơ và dãy số thường gắn bó mật thiết với nhau. Chẳng hạn, một cách khác để tính dãy Fibonacci là bằng cách lưu các giá trị liên tiếp vào một véc-tơ. Một lần nữa, định nghĩa của dãy Fibonacci là $F_1 = 1$, $F_2 = 1$, và $F_i = F_{i-1} + F_{i-2}$ với $i \geq 3$. Trong Scilab, câu lệnh sẽ có dạng

```
F(1) = 1
F(2) = 1
for i=3:n
    F(i) = F(i-1) + F(i-2)
end
ans = F(n)
```

Lưu ý rằng tôi dùng chữ cái viết in cho véc-tơ `F` và chữ cái thường cho các số vô hướng `i` và `n`. Cuối cùng, đoạn chương trình lấy phần tử chót của `F` và lưu nó vào `ans`, vì kết quả của chương trình phải là số Fibonacci thứ n, chứ không phải cả dãy số.

Nếu từng gặp vướng mắc ở Bài tập 3.5 , bạn phải trân trọng sự giản đơn của phiên bản chương trình trên. Cú pháp của Scilab cũng tương đồng với kí hiệu toán học; nó làm ta dễ kiểm tra tính đúng đắn của chương trình. Song có những nhược điểm cần lưu ý gồm

- Bạn phải cẩn thận với khoảng số của vòng lặp. Ở đoạn chương trình trên, vòng lặp chạy từ `3` đến `n`, và mỗi lần ta gán một giá trị vào phần tử thứ `i`. Nó cũng chạy khi ta "đẩy" chỉ số đi hai đơn vị, cho vòng lặp chạy từ 1 đến `n-2`:

```
F(1) = 1
F(2) = 1
for i=1:n-2
    F(i+2) = F(i+1) + F(i)
end
ans = F(n)
```

Bạn dùng phiên bản nào cũng được, nhưng cần chọn một cách làm và phải thống nhất với nó. Nếu bạn kết hợp các phần tử của cả hai cách, bạn sẽ bị lẫn. Tôi ưa dùng dạng có `F(i)` bên vế trái lệnh gán, vì vậy mỗi lượt lặp nó sẽ gán cho phần tử thứ `i`.

- Nếu bạn thực sự chỉ muốn số Fibonacci thứ n, việc lưu trữ cả dãy số sẽ làm tốn dung lượng bộ nhớ của máy. Nhưng nếu mất dung lượng mà đoạn mã trở nên dễ viết và gỡ lỗi hơn thì có lẽ vẫn tốt.

Exercise 4.1 *Hãy viết một vòng lặp để tính* `n` *phần tử đầu tiên của dãy hình học* $A_{i+1} = A_i/2$ *với* $A_1 = 1$. *Lưu ý rằng kí hiệu toán học đặt* A_{i+1} *ở vế trái của đẳng thức. Khi chuyển sang mã lệnh Scilab, có thể bạn sẽ phải đẩy dịch chỉ số.*

4.11 Vẽ đồ thị các véc-tơ

Tính năng vẽ đồ thị rất hữu ích đối với biểu diễn véc-tơ. Nếu bạn gọi `plot` với đối số là một véc-tơ, Scilab sẽ lấy các chỉ số tọa độ theo phương x và giá trị các phần tử làm tọa độ theo phương y. Để biểu diễn giá trị các số Fibonacci đã tính được ở mục trước:

```
plot(F)
```

Hình biểu diễn này thường có ích cho việc gỡ lỗi, đặc biệt là khi véc-tơ của bạn lớn đến nỗi việc in các giá trị số lên màn hình trở nên vô hiệu.

Nếu bạn gọi `plot` với đối số là hai véc-tơ, Scilab sẽ vẽ véc-tơ thứ hai như một hàm số của véc-tơ đầu; nghĩa là giá trị của véc-tơ đầu làm các tọa độ x còn các giá trị tương ứng của véc-tơ sau làm tọa độ y rồi vẽ các cặp điểm (x, y).

```
X = 1:5
Y = 6:10
plot(X, Y)
```

Scilab mặc định màu nét vẽ là xanh lam, nhưng bạn có thể thay đổi bằng một chuỗi kí tự có dạng giống như ta đã thấy ở Mục 3.5 . Chẳng hạn, chuỗi `'ro-'` ra lệnh cho Scilab vẽ những vòng tròn nhỏ ở mỗi điểm dữ liệu; dấu gạch ngang ngụ ý rằng các điểm cần được nối bởi đường thẳng.

Trong ví dụ này, tôi gắn chặt với quy ước đặt tên đối số thứ nhất là `X` (vì nó ứng với tọa độ x) và đối số thứ hai `Y`. Không có gì đặc biệt với những cái tên này; bạn hoàn toàn có thể vẽ `X` như là hàm theo `Y`. Scilab luôn coi véc-tơ thứ nhất là biến độc lập, và véc-tơ thứ hai là biến phụ thuộc.

4.12 Phép rút gọn

Một cách dùng hay thấy ở vòng lặp là chạy qua các phần tử của một mảng về cộng chúng lại, hoặc nhân với nhau, hoặc tính tổng các bình phương, v.v. Kiểu tính toán này được gọi là **phép rút gọn**, vì nó rút gọn một véc-tơ với nhiều phần tử về một con số vô hướng.

Chẳng hạn, vòng lặp này cộng lại những phần tử của một véc-tơ có tên `X` (mà ta coi rằng nó đã được định nghĩa trước).

```
total = 0
for i=1:length(X)
    total = total + X(i)
end
ans = total
```

Các dùng `total` như một biến thu gom cũng giống như điều ta đã thấy ở Mục **??**. Một lần nữa, ta dùng hàm `length` để xác định giới hạn trên của khoảng, vì vậy vòng lặp này sẽ chạy được bất kể độ dài của `X` bằng bao nhiêu. Mỗi lượt lặp, ta cộng `total` với phần tử thứ `i` của `X`, vì vậy khi hết vòng lặp `total` sẽ mang giá trị tổng của tất cả phần tử.

Exercise 4.2 *Hãy viết một vòng lặp để nhân tất cả các phần tử của một véc-tơ với nhau. Có thể bạn sẽ cần đặt một biến thu gom* `product`*, và phải nghĩ xem đặt giá trị ban đầu bằng bao nhiêu trước khi vòng lặp được thực thi.*

4.13 Áp dụng

Một công dụng khác của hàm là chạy qua các phần tử của một véc-tơ, thực hiện một phép toán nào đó lên từng phần tử, rồi tạo ra véc-tơ mới chứa các kết quả.

Dạng tính toán này được gọi là **áp dụng**, bởi vì khi đó bạn áp dụng phép toán với từng phần tử của véc-tơ.

Chẳng hạn, vòng lặp sau đây tính một véc-tơ `Y` trong đó chứa các bình phương của từng phần tử thuộc `X` (một lần nữa, ta giả sử rằng `X` đã được định nghĩa).

```
for i=1:length(X)
    Y(i) = X(i)^2;
end
```

Exercise 4.3 *Hãy viết một vòng lặp để tính một véc-tơ* `Y` *trong đó chứa các giá trị sin của từng phần tử thuộc* `X`*. Để kiểm tra vòng lặp của bạn, hãy viết một chương trình*

1. *sử dụng* `linspace` *(xem thông tin cách dùng) để gán* `X` *là véc-tơ gồm 100 phần tử từ 0 đến* 2π.
2. *dùng vòng lặp vừa viết để lưu các giá trị sin vào* `Y`.
3. *Vẽ đồ thị các phần tử của* `Y` *như là hàm của từng phần tử thuộc* `X`.

4.14 Tìm kiếm

Một cách dùng khác của vòng lặp là tìm kiếm những phần tử trong một véc-tơ rồi trả lại chỉ số của giá trị mà bạn cần tìm (hoặc giá trị đầu tiên thỏa mãn đặc tính nào đó). Chẳng hạn, nếu một véc-tơ bao gồm các giá trị độ cao tính được của một vật thể rơi, bạn có lẽ cũng muốn biết chỉ số nào của véc-tơ đó ứng với lúc vật thể chạm đất (coi rằng mặt đất ở cao độ bằng 0).

Để tạo ra dữ liệu giả định, ta sẽ dùng một dạng đầy đủ của toán tử hai chấm:

```
X = 10:-1:-10
```

Các giá trị trong khoảng này chạy từ 10 đến -10, với **độ dài bước** là -1. Độ dài bước là khoảng cách giữa các phần tử cạnh nhau trong khoảng.

Vòng lặp sau tìm ra chỉ số của phần tử bằng 0 trong `X`:

```
for i=1:length(X)
    if X(i) == 0
        ans = i
    end
end
```

Một điều buồn cười ở vòng lặp này là nó tiếp tục chạy ngay cả khi đã tìm được giá trị mong muốn. Đây có lẽ không phải là điều bạn cần. Song cũng có thể bạn muốn làm cách này, nếu giá trị cần tìm xuất hiện nhiều lần; khi đó vòng lặp sẽ cho chỉ số của phần tử *cuối cùng* thỏa mãn điều kiện đề ra.

Nhưng nếu bạn muốn chỉ số của phần tử đầu tiên (hay biết rằng chỉ có một phần tử như vậy), bạn có thể tiết kiệm một số vòng lặp không cần thiết bằng cách dùng câu lệnh `break`.

```
for i=1:length(X)
    if X(i) == 0
        ans = i
        break
    end
end
```

`break` có tác dụng giống như tên gọi của nó: thoát khỏi vòng lặp. Nó dừng vòng lặp và thực hiện câu lệnh ngay sau vòng lặp (trong trường hợp này, không còn câu lệnh nào nữa, và chương trình kết thúc).

Ví dụ trên minh hoạ cho ý tưởng cơ bản của công việc tìm kiếm, nhưng cũng cho thấy một cách dùng nguy hiểm của lệnh `if`. Hãy nhớ rằng các giá trị dấu phẩy động thường chỉ gần đúng. Điều đó có nghĩa là nếu bạn tìm kiếm dựa trên sự bằng nhau tuyệt đối, có thể bạn sẽ không tìm ra. Chẳng hạn, hãy thử đoạn lệnh sau:

```
X = linspace(1,2);
for i=1:length(X)
    Y(i) = sin(X(i));
end
plot(X, Y)
```

Bạn có thể thấy rằng trên đồ thị, giá trị của $\sin x$ đi qua 0.9 trong khoảng này, nhưng nếu tìm một chỉ số sao cho `Y(i) == 0.9`, bạn sẽ trắng tay.

```
for i=1:length(Y)
    if Y(i) == 0.9
        ans = i
        break
    end
end
```

Điều kiện này không bao giờ đúng, vì thế phần thân của lệnh `if` không bao giờ được thực hiện.

Ngay cả khi đồ thị biểu diễn một đường liên tục, bạn đừng quên rằng cả `X` và `Y` đều là các chuỗi rời rạc, và thường chỉ là giá trị xấp xỉ. Như một quy tắc, bạn nên tránh việc dùng toán tử == để so sánh hai giá trị số có dấu phẩy động. Có một số cách khắc phục điều này mà ta sẽ trở lại sau.

Exercise 4.4 *Hãy viết một vòng lặp để tìm chỉ số của số nguyên đầu tiên xuất hiện trong một véc-tơ và lưu nó vào trong* `ans`*. Nếu không có giá trị số âm nào, bạn nên luôn đặt* `ans` *bằng* -1 *(vốn không phải là một chỉ số hợp lệ) vì đó là cách biểu thị một trường hợp bất thường.*

4.15 Sự thật có thể gây mất hứng

Những người lập trình Scilab có kinh nghiệm sẽ không bao giờ viết những vòng lặp như trong chương này, vì Scilab cung cấp những cách làm đơn giản và nhanh hơn cho việc rút gọn, áp dụng và tìm kiếm.

Chẳng hạn, hàm `sum` có thể dùng để tính tổng của các phần tử trong véc-tơ và `prod` để tính tích.

Nhiều thao tác áp dụng có thể được thực hiện bằng các toán tử đối với từng phần tử. Câu lệnh sau đây gọn hơn là vòng lặp ở Mục 4.13

```
Y = X .^ 2
```

Ngoài ra, đa số các hàm Scilab lập sẵn đều tính được với véc-tơ:

```
X = linspace(0, 2*pi);
Y = sin(X);
plot(X, Y)
```

Sau cùng, hàm `find` có thể đảm nhiệm thao tác tìm kiếm, nhưng để hiểu được nó ta cần biết một số khái niệm khác nữa, vì vậy tạm thời bạn cần bằng lòng với cách làm của mình.

Tôi bắt đầu bằng việc đề cập các vòng lặp đơn giản vì muốn cho thấy những khái niệm cơ bản và tạo điều kiện cho bạn thực hành. Đến lúc nào đó bạn có thể phải viết một vòng lặp mà Scilab không có sẵn một cách làm tắt, mà bạn phải tự lập nên.

Nếu bạn hiểu được các vòng lặp và thấy thoải mái với cách làm tắt thì hãy dùng chúng! Còn nếu không bạn luôn có thể viết hẳn vòng lặp ra.

Exercise 4.5 *Hãy viết một biểu thức để tính tổng của các bình phương từng phần tử trong một véc-tơ.*

4.16 Thuật ngữ

lệnh phức hợp: Một lệnh như `if` và `for`, trong đó chứa các câu lệnh khác ở phần thân được viết thụt đầu dòng.

lồng ghép: Đặt một lệnh phức hợp vào trong phần thân của một lệnh phức hợp khác.

toán tử quan hệ: Toán tử để so sánh hia giá trị và trả lại kết quả là một giá trị logic.

giá trị logic: Giá trị biểu thị cho "đúng" hoặc "sai". Scilab dùng các giá trị tương ứng là `%t` và `%f`.

cờ: Biến bao gồm một giá trị logic, thường được dùng để lưu trữ trạng thái của một điều kiện nào đó.

vô hướng: Một giá trị đơn lẻ.

véc-tơ: Dãy các giá trị.

ma trận: Tập hợp các giá trị xếp theo hai chiều (cũng gọi là "array" (mảng) trong một số tài liệu về Scilab).

chỉ số: Số nguyên dùng để chỉ thị một trong các giá trị trong một véc-tơ hay ma trận.

phần tử: Một trong các giá trị của véc-tơ hay ma trận.

theo phần tử: Phép tính thực hiện trên từng phần tử của một véc-tơ hay ma trận (khác với các phép toán trong môn đại số tuyến tính).

rút gọn: Cách xử lý các phần tử của một véc-tơ để trả về một giá trị đơn lẻ, như tổng các phần tử.

áp dụng: Cách xử lý một véc-tơ bằng việc thực hiện phép toán đối với mỗi phần tử, để cho ra một véc-tơ chứa các kết quả.

tìm kiếm: Cách xử lý một véc-tơ bằng việc kiểm tra từng phần tử một theo thứ tự đến khi tìm thấy một phần tử thỏa mãn điều kiện mong muốn.

4.17 Bài tập

Exercise 4.6 *Tỉ số giữa hai số Fibonacci liên tiếp, F_{n+1}/F_n, sẽ hội tụ về một hằng số khi n tăng lên. Hãy viết một chương trình tính ra một véc-tơ gồm n phần tử đầu tiên của dãy Fibonacci (giả thiết rằng biến* `n` *đã được định nghĩa), rồi tính một véc-tơ mới chứa các tỉ số của cac số Fibonacci liên tiếp. Hãy vẽ đồ thị véc-tơ này xem nó có xu hướng hội tụ không. Nếu có thì nó hội tụ về giá trị nào?*

Exercise 4.7 *Có một hệ phương trình vi phân nổi tiếng được xấp xỉ bởi hệ phương trình như sau:*

$$x_{i+1} = x_i + \sigma\,(y_i - x_i)\,dt \tag{4.1}$$
$$y_{i+1} = y_i + [x_i(r - z_i) - y_i]\,dt \tag{4.2}$$
$$z_{i+1} = z_i + (x_i y_i - b z_i)\,dt \tag{4.3}$$

- *Hãy viết một chương trình tính 10 phần tử đầu tiên của các dãy X, Y và Z lưu chúng vào các véc-tơ có tên* `X`, `Y` *và* `Z`.

  Hãy dùng các giá trị khởi đầu $X_1 = 1$, $Y_1 = 2$ và $Z_1 = 3$, cùng $\sigma = 10$, $b = 8/3$ và $r = 28$, bước thời gian $dt = 0.01$.

- *Hãy đọc các giải thích cách dùng* `plot3` *và* `comet3` *rồi vẽ kết quả trong không gian 3 chiều.*

- *Một khi mã lệnh đã chạy được, hãy dùng các dấu chấm phẩy để ngăn các kết quả in ra rồi sau đó chạy chương trình với các dãy có độ dài lần lượt là 100, 1000 và 10000.*

- *Chạy lại chương trình với các điều kiện khởi đầu khác nhau. Điều này có ảnh hưởng gì đến kết quả?*

- *Hãy chạy chương trình với các giá trị khác nhau của σ, b và r rồi xem liệu bạn có thể hiểu được từng biến có ảnh hưởng gì đến hệ.*

Exercise 4.8 *Phép khớp logistic thường được đem ra làm ví dụ cho việc một biểu hiện phức tạp, hỗn loạn có thể nảy sinh từ các phương trình động lực đơn giản [một số nội dung trong bài này được lấy từ trang Wikipedia]. Nó trở nên phổ biến từ khi xuất hiện bài báo năm 1976 của nhà sinh vật học Robert May.*

Phép khớp logistic được dùng để mô phỏng sinh khối của một loài khi có mặt các yếu tố hạn chế như nguồn thức ăn và dịch bệnh. Trong trường hợp này, có hai yếu tố cần xét đến: (1) Quá trình sinh sản làm tăng sinh khối của các loài tỉ lệ với

số cá thể hiện tại. (2) Quá trình chết đói khiến cho sinh khối giảm với tốc độ tỉ lệ với hiệu số giữa sức mang của môi trường với số cá thể hiện tại.

Điều này có thể viết bằng biểu thức sau

$$X_{i+1} = rX_i(1 - X_i)$$

trong đó X_i là một số giữa 0 và 1 để biểu thị sinh khối vào năm thứ i, còn r là một số dương biểu thị tốc độ tổng hợp của sinh sản và chết đói.

- *Hãy viết một tập tin lệnh có tên* `logmap` *để tính 50 phần tử đầu tiên của X với* `r=3.9` *và* `X1=0.5`*, trong đó* `r` *là tham số của phép khớp logistic còn* `X1` *là số cá thể ban đầu.*

- *Hãy vẽ đồ thị kết quả với một khoảng các giá trị của r từ 2.4 đến 4.0. Biểu hiện của hệ thống sẽ thay đổi ra sao khi bạn thay đổi ra sao khi bạn thay đổi r?*

- *Một cách đặc trưng cho ảnh hưởng của r là vẽ một đồ thị với r là tọa độ x và sinh khối là tọa độ y, để cho thấy ứng với mỗi giá trị của r, giá trị sinh khối ở trạng thái ổn định là bao nhiêu. Thử xem bạn có hình dung được cách vẽ biểu đồ này không?*

Chương 5

Hàm

5.1 Sự xung đột về tên

Hãy nhớ rằng tất cả các tập tin lệnh bạn viết đều chạy trong cùng một không gian làm việc, vì vậy nếu một chương trình làm thay đổi giá trị một biến thì tất cả các chương trình khác đều thấy được sự thay đổi đó. Với một ít các chương trình đơn giản, điều này không đáng kể, nhưng rồi sau này những tương tác giữa các chương trình trở nên không thể quản lý được.

Chẳng hạn, chương trình sau tính tổng của `n` số đầu tiên trong một dãy hình học, nhưng cũng có **hiệu ứng phụ** là gán các giá trị cho `A1`, `total`, `i` và `a`.

```
A1 = 1;
total = 0;
for i=1:10
    a = A1 * 0.5^(i-1);
    total = total + a;
end
ans = total
```

Nếu bạn dùng bất cứ tến biến nào nêu trên trước khi gọi mã lệnh này thì bạn có thể sẽ ngạc nhiên khi thấy rằng sau khi chạy đoạn chương trình, các giá trị đó đã thay đổi. Nếu bạn có 2 đoạn chương trình dùng cùng tên biến, bạn có thể thấy rằng chúng hoạt động riêng biệt nhưng đổ vỡ khi bạn cố gắng kết hợp các chương trình lại. Kiểu tương tác này được gọi là **xung đột về tên**.

Khi số tập tin lệnh của bạn viết tăng lên, đồng thời cũng dài hơn và phức tạp hơn thì sự xung đột về tên càng nghiêm trọng. Để tránh được vấn đề này, cần tạo ra các hàm.

5.2 Hàm

Một **hàm** cũng giống như một tập tin lệnh, chỉ khác ở chỗ

- Mỗi hàm có không gian làm việc riêng của nó, vì vậy bất kì biến nào được định nghĩa trong hàm đều chỉ tồn tại khi hàm đang chạy, và không ảnh hưởng đến các biến trong không gian làm việc khác, ngay cả khi các biến đó có cùng tên.
- Các dữ liệu đầu vào và kết quả đầu ra của hàm đều được định nghĩa một cách cẩn thận để tránh sự tương tác không mong muốn.

Để định nghĩa một hàm, bạn cần tạo ra một tập tin M với tên gọi mong muốn, và đặt một lời định nghĩa hàm vào trong đó. Chẳng hạn, để tạo ra một hàm có tên `myfunc`, hãy tạo ra tập tin M là `myfunc.sce` và đặt vào đó định nghĩa hàm sau.

```
function res = myfunc(x)
    s = sin(x)
    c = cos(x)
    res = abs(s) + abs(c)
endfunction
```

Từ đầu tiên của tập tin phải là `function`, vì đó là dấu hiệu để Scilab nhận biết một tập tin hàm thay vì tập tin lệnh.

Một định nghĩa hàm chính là một câu lệnh phức hợp. Dòng đầu tiên là **dấu** của hàm; nó chỉ định các số liệu đầu vào và đầu ra của hàm. Trong trường hợp này **biến đầu vào** có tên là `x`. Khi hàm này được gọi, đối số do người dùng cung cấp sẽ được gán cho `x`.

Biến đầu ra được gọi là `res`, chữ viết tắt của "result" (kết quả). Bạn có thể gọi biến đầu ra bằng tên gì cũng được, nhưng theo thông lệ, tôi thường đặt nó là `res`. Thường thì việc cuối cùng mà một hàm thực hiện là gán một giá trị cho biến đầu ra.

Một khi bạn đã định nghĩa một hàm mới, bạn có thể gọi nó theo cách giống như gọi các hàm có sẵn trong Scilab. Nếu bạn gọi hàm như một câu lệnh, Scilab sẽ đặt kết quả vào `ans`:

```
--> myfunc(1)

ans = 1.3817733
```

Nhưng thường thấy hơn (đồng thời là cách tốt hơn) là nên gán kết quả cho một biến:

```
--> y = myfunc(1)

y = 1.3817733
```

Khi bạn gỡ lỗi một hàm mới, có thể bạn sẽ phải hiển thị các giá trị trung gian như thế này, nhưng một khi hàm đã chạy tốt, bạn sẽ phải thêm vào các dấu chấm phẩy để khiến nó trở thành một **hàm lặng**. Đa số các hàm có sẵn trong Scilab đều là hàm lặng, chúng tính ra kết quả, nhưng không hiển thị gì (ngoại trừ những thông báo trong một số trường hợp).

Mỗi hàm có một không gian làm việc riêng của nó, vốn được tạo ra khi hàm bắt đầu chạy và bị xóa đi khi hàm kết thúc. Nếu bạn thử truy cập (đọc hay ghi) biến được định nghĩa bên trong hàm, bạn sẽ thấy nó không tồn tại.

```
--> clear
--> y = myfunc(1);
--> who

Your variables are: y
  ...
-->s
  !--error 4
Undefined variable: s
```

Giá trị duy nhất của hàm mà bạn có thể truy cập được là kết quả của nó, ở đây là giá trị gán cho biến `y`.

Nếu bạn có các biến tên là `s` hoặc `c` trong không gian làm việc trước khi gọi `myfunc`, chúng sẽ vẫn ở đó ngay cả khi hàm kết thúc.

```
--> s = 1;
--> c = 1;
--> y = myfunc(1);
--> s, c

s = 1.
c = 1.
```

Như vậy ở trong hàm bạn có thể dùng bất kì tên biến nào bạn muốn mà không sợ xung đột.

5.3 Thông tin về hàm

Tại chỗ bắt đầu của mỗi tập tin hàm, bạn nên ghi một lời chú thích nhằm giải thích tác dụng của hàm được viết ra.

```
// res = myfunc (x)
// Tính khoảng cách Manhattan từ gốc tọa độ
// đến điểm trên đường tròn đơn vị với góc (x) đo bằng radian.

function res = myfunc (x)
    s = sin(x);
    c = cos(x);
    res = abs(s) + abs(c);
endfunction
```

- Dấu của hàm, bao gồm tên hàm, (các) biến đầu vào, và (các) biến đầu ra.
- Một lời mô tả rõ ràng, ngắn gọn về công dụng của hàm. Lời mô tả phải **khái quát**: nên bỏ qua những chi tiết về *cách* hoạt động của hàm, mà chỉ bao gồm những thông tin mà người dùng hàm muốn biết. Bạn có thể thêm vào các chú thích bên trong hàm để lý giải những chi tiết.
- Một lời giải thích ý nghĩa các biến đầu vào; chẳng hạn, trong trường hợp này cần lưu ý rằng `x` được coi là số đo góc tính theo ra-đian.
- Các điều kiện trước và điều kiện sau.

5.4 Tên hàm

Cần lưu ý là tên của tập tin không được phép chứa dấu cách. Chẳng hạn, nếu bạn viết một hàm và đặt tên tập tin là `my func.m`, thì mặc dù trình soạn thảo Scilab vẫn đồng ý, nhưng khi thử chạy, bạn sẽ nhận được:

```
-->y = my func(1);
      !--error 4
Undefined variable: my
```

Ngoài ra, cũng cần lưu ý, tên hàm được viết có thể xung đột với các hàm lập sẵn của Scilab. Chẳng hạn, nếu bạn tạo ra tập tin mã lệnh có tên `sum.sce`, rồi gọi `sum`, Scilab có thể sẽ gọi hàm mới do *bạn* viết, chứ không phải hàm lập sẵn! Hàm nào được gọi sẽ tùy thuộc vào thứ tự của các thư mục xếp trong đường dẫn tìm kiếm, và (trong một số trường hợp) phụ thuộc vào đối số. Chẳng hạn, hãy đặt các lệnh sau vào tập tin có tên `sum.sce`:

```
function res = sum(x)
   res = 7;
endfunction
```

Sau khi lưu lại và chạy chương trình, Scilab sẽ cảnh báo rằng bạn đã định nghĩa đè lên một hàm lập sẵn:

```
-->exec('C:\Users\Chien\Documents\sum.sce', -1)
Warning : redefining function: sum                    . Use funcprot(0)
```

Và từ đó, kết quả tính toán có thể sẽ sai hẳn:

```
--> sum(1:3)

ans = 7.

--> sum

ans = 7.
```

Trong trường hợp đầu Scilab đã dùng hàm lập sẵn; ở trường hợp thứ hai nó thực hiện hàm bạn viết! Chế độ tương tác này có thể rất gây lẫn. Trước khi tạo ra hàm mới, bạn cần kiểm tra xem có sẵn hàm nào của Scilab có cùng tên không. Nếu có, hãy đặt một tên khác!

5.5 Nhiều biến đầu vào

Các hàm có thể, và thường, nhận nhiều biến đầu vào. Chẳng hạn, hàm sau đây nhận hai biến đầu vào, `a` và `b`:

```
function res = hypotenuse(a, b)
    res = sqrt(a^2 + b^2);
endfunction
```

Nếu bạn còn nhớ Định lý Py-ta-go, bạn có thể đã hình dung ra là hàm này để tính chiều dài cạnh huyền của tam giác vuông nếu các cạnh góc vuông là `a` và `b`. (Có một hàm Scilab tên là `hypot` thực hiện điều tương tự.)

Nếu ta gọi nó từ Console với các đối số 3 và 4, ta có thể chắc rằng chiều dài cạnh thứ ba sẽ bằng 5.

```
--> c = hypotenuse(3, 4)

c = 5.
```

Các đối số mà bạn cấp vào đây được gán cho các biến đầu vào theo đúng thứ tự, vì vậy ở trường hợp này 3 được gán cho `a` còn 4 được gán cho `b`. Scilab kiểm tra để đảm bảo rằng bạn cung cấp đủ đối số; nếu cung cấp thiếu, bạn sẽ nhận được:

```
--> c = hypotenuse(3)
 !--error 4
Undefined variable: b

at line       2 of function hypotenuse called by :
 c = hypotenuse(3)
```

Nếu cung cấp thừa đối số, bạn sẽ nhận được:

```
--> c = hypotenuse(3,4,5)
                         !--error 58
Wrong number of input arguments:

Arguments are :

 a         b
```

5.6 Các hàm logic

Ở Mục 4.4 ta đã dùng các toán tử logic để so sánh các giá trị. Scilab cũng cung cấp các **hàm logic** để kiểm tra những điều kiện nhất định và trả lại những giá trị logic: T với nghĩa là "đúng" và F với nghĩa là "sai".

Chẳng hạn, `isprime` dùng để kiểm tra xem số đã cho có phải là số nguyên tố hay không. Bạn phải tải hàm này từ `http://fileexchange.scilab.org/toolboxes/235000/1.0/files/fn_isprim` rồi mở bằng SciNotes rồi chạy mã lệnh. Sau đó:

```
--> isprime(17)

ans = T

--> isprime(21)

ans = F
```

Hàm `isvector` dùng để kiểm tra xem một giá trị có phải là véc-tơ hay không; nếu sai, thì giá trị đó có thể là một số hoặc một ma trận.

Để kiểm tra xem một giá trị bạn đã tính có phải là số nguyên không, ta có thể viết một hàm logic có tên là `isintegral` như sau:

```
function res = isintegral(x)
    if round(x) == x
        res = %t;
    else
        res = %f;
    end
endfunction
```

Ở đây các giá trị lập sẵn `%t` và `%f` cho kết quả lần lượt là T và F. Bạn hoàn toàn có thể viết lại hàm trên với phần thân ngắn gọn bằng một câu lệnh: `res = round(x) == x`

Sau đó, hãy áp dụng hàm này cho trường hợp tính cạnh huyền tam giác:

```
--> c = hypotenuse(3, 4)

c = 5.

--> isintegral(c)

ans = T
```

Hàm này dùng được trong phần lớn các trường hợp, nhưng nhớ rằng các giá trị dấu phẩy động chỉ xấp xỉ đúng, trong một số trường hợp giá trị xấp xỉ là số nguyên nhưng giá trị thực lại không phải.

5.7 Một ví dụ xây dựng dần

Giả dụ rẳng ta muốn viết một chương trình tìm các "bộ ba số Py-ta-go": những tập hợp số nguyên như 3, 4, và 5, là độ dài các cạnh của một tam giác vuông. Nói cách khác, ta muốn tìm các giá trị nguyên a, b và c sao cho $a^2 + b^2 = c^2$.

Sau đây là các bước mà ta sẽ làm theo để phát triển chương trình một cách tăng dần.

- Viết một tập tin lệnh có tên `find_triples` và bắt đầu với một câu lệnh đơn giản như `x=5`.
- Viết một vòng lặp để liệt kê các giá trị a từ 1 đến 3, và hiển thị chúng.
- Viết một vòng lặp lồng ghép để liệt kê các giá trị b từ 1 đến 4, và hiển thị chúng.
- Bên trong vòng lặp, gọi `hypotenuse` để tính c và hiển thị nó.
- Dùng `isintegral` để kiểm tra xem c có bằng một số nguyên hay không.
- Dùng một lệnh if để chỉ in ra những cặp giá trị a, b và c nào thỏa mãn điều kiện.
- Chuyển nội dung tập tin lệnh vào trong một hàm.
- Khái quát hóa hàm này để nhận vào các biến chỉ định khoảng các giá trị cần được tìm kiếm.

Như vậy bản nháp đầu tiên của chương trình này là `x=5`, trông có vẻ ngốc nghếch, nhưng nếu bạn bắt đầu một cách đơn giản và mỗi lúc chỉ thêm vào ít một, bạn sẽ tránh được gỡ lỗi rất nhiều.

Sau đây là bản nháp thứ hai:

```
for a=1:3
    a
end
```

Ở mỗi bước, chương trình đều có thể chạy thử được: nó có in ra kết quả (hay một hiệu ứng thấy được) mà ta có thể kiểm tra.

5.8 Vòng lặp lồng ghép

Bản nháp thứ ba chứa một vòng lặp lồng ghép:

```
for a=1:3
    a
    for b=1:4
        b
    end
end
```

Vòng lặp trong được thực hiện 3 lần, mỗi lần với một giá trị khác nhau của `a`, vì vậy sau đây là kết quả (tôi đã chỉnh lại độ dãn cách để làm nổi bật cấu trúc):

```
--> find_triples

a = 1   b = 1
        b = 2
        b = 3
        b = 4

a = 2   b = 1
        b = 2
        b = 3
        b = 4

a = 3   b = 1
        b = 2
        b = 3
        b = 4
```

Bước tiếp theo là tính c với mỗi cặp giá trị của a và b.

```
for a=1:3
    for b=1:4
        c = hypotenuse(a, b);
        [a, b, c]
    end
end
```

Để hiển thị các giá trị của `a`, `b` và `c`, tôi dùng đến một đặc điểm của Scilab mà ta chưa gặp. Toán tử ngoặc vuông tạo ra một ma trận mới mà, khi được hiển thị, sẽ cho thấy các giá trị trên cùng một dòng:

```
--> find_triples

ans = 1.0000    1.0000    1.4142136
ans = 1.0000    2.0000    2.236068
ans = 1.0000    3.0000    3.162277
ans = 1.0000    4.0000    4.1231056
ans = 2.0000    1.0000    2.236068
ans = 2.0000    2.0000    2.8284271
ans = 2.0000    3.0000    3.6055513
ans = 2.0000    4.0000    4.472136
ans = 3.0000    1.0000    3.1622777
ans = 3.0000    2.0000    3.6055513
ans = 3.0000    3.0000    4.2426407
ans = 3.         4.         5.
```

Bạn đọc tinh mắt sẽ phát hiện được rằng chúng ta đang lãng phí một ít công sức lập trình. Sau khi kiểm tra $a = 1$ và $b = 2$, sẽ chẳng cần kiểm tra $a = 2$ và $b = 1$. Ta có thể loại bỏ công việc thừa này bằng cách chỉnh lại khoảng lặp của vòng thứ hai:

```
for a=1:3
    for b=a:4
        c = hypotenuse(a, b);
        [a, b, c]
    end
end
```

Nếu bạn vẫn theo kịp nội dung, hãy chạy phiên bản mã lệnh này và chắc chắn rằng nó cho kết quả như mong muốn.

5.9 Điều kiện và cờ

Bước tiếp theo là kiểm tra xem giá trị c có nguyên không. Vòng lặp này gọi `isintegral` và in ra giá trị logic thu được.

```
for a=1:3
    for b=a:4
        c = hypotenuse(a, b);
        flag = isintegral(c);
        [c, flag]
    end
end
```

Bằng cách không hiển thị `a` và `b` tôi làm cho việc soát kết quả dễ dàng hơn để đảm bảo rằng các giá trị của `c` và `flag` trông đúng đắn.

```
--> find_triples

ans = 1.4142136          0
ans = 2.236068          0
ans = 3.162277          0
ans = 4.1231056          0
ans = 2.8284271          0
ans = 3.6055513          0
ans = 4.472136          0
ans = 4.2426407          0
ans = 5              1
```

Tôi chọn các khoảng `a` và `b` đều nhỏ (vì vậy số dòng kết quả đầu ra có thể kiểm soát được), nhưng phải bao gồm ít nhất là một bộ ba số Py-ta-go. Một khó khăn thường gặp khi gỡ lỗi là phải phát sinh đủ kết quả để cho thấy rằng mã lệnh có (hoặc không) hoạt động mà không quá thừa thãi.

Bước tiếp theo là dùng `flag` để chỉ hiển thị những bộ ba thỏa mãn yêu cầu:

```
for a=1:3
    for b=a:4
        c = hypotenuse(a, b);
        flag = isintegral(c);
        if flag
            [a, b, c]
        end
    end
end
```

Bây giờ kết quả đẹp và đơn giản hơn:

```
--> find_triples

ans = 3.     4.     5.
```

5.10 Bao bọc và khái quát hóa

Dưới dạng tập tin lệnh, chương trình này có tác dụng phụ là đã gán các giá trị cho `a`, `b`, `c` và `flag`, vốn sẽ làm chương trình khó dùng hơn khi các tên biến trên đang được sử dụng. Bằng cách bọc mã lệnh này vào trong một hàm, ta có thể tránh được sự xung đột về tên; quá trình này được gọi là **bao bọc** vì nó cô lập chương trình khỏi không gian làm việc.

Để đưa mã lệnh đã viết vào trong một hàm, ta phải viết thụt đầu dòng toàn bộ. Trình soạn thảo Scilab cung cấp một cách làm tắt, lệnh Increase Indent dưới trình đơn Text.

Bản nháp đầu tiên của hàm trong đó không có biến đầu vào như sau:

```
function find_triples ()
    for a=1:3
        for b=a:4
            c = hypotenuse(a, b);
            flag = isintegral(c);
            if flag
                disp([a, b, c])
            end
        end
    end
endfunction
```

Cặp ngoặc tròn trong đó không có gì ở dấu của hàm là không cần thiết, nhưng chúng làm rõ là không có biến đầu vào nào. Tương tự, khi tôi gọi một hàm mới, tôi cũng ưa dùng cặp ngoặc để tự nhủ rằng đó là một hàm chứ không phải nội dung tập tin lệnh:

```
--> find_triples()
```

Biến đầu ra cũng không nhất thiết phải có. Như trong trường hợp này, tôi biết rằng cần phải in nhiều giá trị nên không tiện để một kết quả chung `res` cho toàn bộ hàm. Thay vào đó, lệnh `disp` được dùng để in từng giá trị từng giá trị khi duyệt qua vòng lặp.

Thực ra vẫn có phong cách viết tên hàm theo dạng mẫu, như `function res = find_triples ()`. Trong trường hợp như vậy, ngay trên dòng `endfunction` cần một lệnh gán như `res = 0` để báo rằng hàm thực hiện thành công.

Bước tiếp theo là khái quát hóa hàm này bằng cách thêm các biến đầu vào. Cách khái quát khá tự nhiên là thay thế các hằng số 3 và 4 bằng một biến để ta có thể tìm kiếm trên một khoảng lớn tùy ý.

```
function find_triples (n)
    for a=1:n
        for b=a:n
            c = hypotenuse(a, b);
            flag = isintegral(c);
            if flag
                disp([a, b, c])
            end
        end
    end
endfunction
```

Sau đây là các kết quả thu được trong khoảng từ 1 đến 15:

```
--> find_triples(15)

3.     4.     5.
5.    12.    13.
6.     8.    10.
8.    15.    17.
9.    12.    15.
```

Trong các kết quả này có cái hay, có cái không. Các cặp $5, 12, 13$ và $8, 15, 17$ thật sự "mới," còn các cặp khác chỉ là các bội số của cặp $3, 4, 5$ mà ta đã biết.

5.11 Một sai sót

Khi bạn thay đổi dấu ấn của hàm, bạn cũng phải thay đổi tất cả mọi chỗ gọi đến hàm đó. Chẳng hạn, nếu tôi định thêm một biến thứ ba vào `hypotenuse`:

```
function res = hypotenuse(a, b, d)
    res = (a^d + b^d) ^ (1/d);
endfunction
```

Khi `d` bằng 2, hàm này có tác dụng giống như cũ. Hiện không có lý do nào phù hợp để khái quát hàm này theo cách trên; đây chỉ là ví dụ. Bây giờ khi bạn chạy `find_triples`, bạn sẽ nhận được:

```
-->find_triples(20)
 !--error 4
Undefined variable: d

at line       2 of function hypotenuse called by :
at line       4 of function find_triples called by :
find_triples(20)
```

Như vậy thật khó tìm ra lỗi. Đây là một ví dụ cho kĩ thuật phát triển mà đôi khi có ích: thay vì tìm kiếm mọi chỗ trong chương trình có dùng đến `hypotenuse`, bạn có thể chạy chương trình và theo các dòng thông báo để tìm ra lỗi.

Nhưng kĩ thuật này rất mạo hiểm, đặc biệt khi các thông báo lỗi đưa ra gợi ý phải sửa đổi những gì. Nếu bạn làm theo chúng, bạn có thể làm biến mất thông báo lỗi, nhưng điều đó không có nghĩa là chương trình đã làm đúng điều ta muốn. Scilab không biết rằng chương trình *cần phải* làm gì, mà bạn phải có trách nhiệm về điều này.

Và từ đó dẫn đến Định lý thứ tám về gỡ lỗi:

> Các lời thông báo lỗi đôi khi bảo cho bạn biết điều gì trục trặc, nhưng hiếm khi chúng bảo cho bạn cách làm (và nếu chúng cố gắng giúp đi nữa thì cũng thường nói sai).

5.12 `continue`

Ở khâu cải tiến cuối cùng, ta hãy sửa hàm này sao cho nó chỉ hiểu thị những bộ ba số Py-ta-go "thấp nhất", chứ không kể tất cả các bội số của chúng.

Cách làm đơn giản nhất để loại bỏ các bội số là kiểm tra xem a và b có thừa số chung hay không. Nếu có, thì việc chia các số này cho thừa số chung sẽ cho ra một bộ ba số nhỏ hơn mà ta đã kiểm tra.

Scilab có một hàm `gcd` để phân tích ước chung cao nhất của nhiều thành phần; một trường hợp riêng trong đó là ước số chung lớn nhất. Trong ví dụ này giá trị ước số chung lớn nhất đó được gọi là `gcdval`; nếu nó lớn hơn 1, thì a và b có một ước số chung và ta có thể dùng lệnh `continue` để nhảy sang cặp tiếp theo:

```
function find_triples (n)
    for a=1:n
        for b=a:n
            pair = int32([a,b]);
            [gcdval, mat] = gcd(pair);
            if gcdval > 1
                continue
            end
            c = hypotenuse(a, b);
            if isintegral(c)
                disp([a, b, c])
            end
        end
    end
endfunction
```

`continue` làm cho chương trình dừng vòng lặp hiện tại (tức là không thực hiện phần còn lại của thân lệnh nữa), nhảy đến đầu vòng lặp, và "tiếp tục" với lượt lặp liền sau.

Trong trường hợp này, vì ta có hai vòng lặp nên sẽ không hiển nhiên là vòng lặp nào được nhảy đến, nhưng quy tắc là nhảy đến vòng lặp sâu bên trong nhất (đúng theo ý định của ta ở đây).

Tôi cũng làm đơn giản chương trình một chút bằng cách loại bỏ `flag` và dùng `isintegral` làm điều kiện cho lệnh `if`.

Sau đây là các kết quả với `n=40`:

```
--> find_triples(40)

ans =  3.     4.     5.
ans =  5.    12.    13.
```

```
ans =  7.    24.    25.
ans =  8.    15.    17.
ans =  9.    40.    41.
ans = 12.    35.    37.
ans = 20.    21.    29.
```

Có một sự liên hệ thú vị giữa các số Fibonacci và cặp số Py-ta-go. Nếu F là một dãy số Fibonacci, thì

$$(F_n F_{n+3}, 2F_{n+1}F_{n+2}, F_{n+1}^2 + F_{n+2}^2)$$

là một cặp số Py-ta-go với mọi $n \geq 1$.

Exercise 5.1 *Hãy viết một hàm có tên* `fib_triple` *để nhận vào một biến* `n`*, dùng* `fibonacci2` *để tính* `n` *số Fibonacci đầu tiên, rồi kiểm tra xem công thức nói trên có tạo thành bộ ba số Py-ta-go với mỗi số trong dãy không.*

5.13 Khoa học và niềm tin

Ta hãy xem lại một loạt các bước xảy ra khi bạn gọi một hàm:

1. Trước khi hàm bắt đầu chạy, Scilab tạo ra một không gian làm việc mới cho nó.

2. Scilab lượng giá từng đối số và gán các giá trị tìm được lần lượt cho từng biến đầu vào (vốn tồn tại trong không gian làm việc *mới*).

3. Mã lệnh ở phần thân của hàm được thực thi. Đâu đó trong phần thân (thường là ở dòng cuối cùng) một giá trị sẽ được gán cho biến đầu ra.

4. Không gian làm việc của hàm bị xóa bỏ; thứ duy nhất còn lại là giá trị của biến đầu ra và mọi hiệu ứng phụ của hàm (như hiển thị các giá trị hoặc vẽ một hình).

5. Chương trình tiếp tục tại điểm mà nó tạm dừng để thực hiện hàm. Giá trị của lời gọi hàm là giá trị của biến đầu ra.

Khi bạn đang đọc chương trình và gặp một lời gọi hàm, có hai cách diễn giải nó:

- Bạn có thể nghĩ cách khoa học như tôi đã trình bày, và theo các bước thực hiện của chương trình, tiến vào trong hàm rồi sau đó trở lại, hoặc
- Bạn có dựa vào "niềm tin": giả sử rằng hàm hoạt động đúng, và đọc tiếp lệnh sau lời gọi hàm đó.

Khi bạn dùng những hàm lập sẵn, cách tự nhiên là dựa vào niềm tin, một phần là do bạn trông đợi rằng đại đa số các hàm Scilab đều hoạt động đúng, và một phần là do bạn không xem được mã lệnh bên trong phần thân hàm.

Khi bắt đầu tự viết các hàm, bạn có thể sẽ tự thấy mình trong đi theo "luồng thực hiện" của chương trình. Điều này có thể giúp ích khi bạn đang học, nhưng khi đã có kinh nghiệm, bạn nên quen với ý tưởng viết một hàm, kiểm tra để đảm bảo nó chạy đúng, và sau đó quên đi những chi tiết về cách hoạt động của nó.

Việc quên đi các chi tiết được gọi là **trừu tượng hóa**; trong ngữ cảnh này, trừu tượng hóa có nghĩa là quên đi *cách* hoạt động của một hàm, và chỉ giả sử (sau khi kiểm tra hợp lý) rằng hàm chạy được.

5.14 Thuật ngữ

hiệu ứng phụ: Hiệu ứng như thay đổi không gian làm việc, mà không phải là mục đích của chương trình.

xung đột về tên: Hoàn cảnh trong đó hai tập tin chương trình dùng cùng một tên biến đã can thiệp nhau.

biến đầu vào: Biến trong một hàm, được nhận giá trị từ một trong các đối số, khi ta gọi hàm.

biến đầu ra: Biến trong một hàm, dùng để trả một giá trị từ hàm về chương trình gọi.

dấu của hàm: Dòng đầu tiên của một lời định nghĩa hàm, trong đó có chỉ định tên hàm, các biến đầu vào và biến đầu ra.

hàm lặng: Hàm không làm hiển thị giá trị nào, cũng không vẽ hình hay có bất kì hiệu ứng phụ gì.

hàm logic: Hàm trả lại một giá trị logic (1 đóng vai trò "đúng" và 0 đóng vai trò "sai").

bao bọc: Quá trình gói một phần của chương trình vào trong một hàm nhằm hạn chế những tương tác (trong đó có xung đột về tên) giữa hàm và phần còn lại của chương trình.

khái quát hóa: Làm cho hàm trở nên linh hoạt hơn bằng cách thay những giá trị cụ thể bằng các biến đầu vào.

trừu tượng hóa: Sự bỏ qua những chi tiết hoạt động của một hàm nhằm tập trung vào một mô hình đơn giản hơn: hàm làm việc gì?

5.15 Bài tập

Exercise 5.2 *Chọn bất kì tập tin lệnh nào bạn đã viết, bao bọc nó vào trong một hàm có tên thích hợp, rồi khái quát hóa hàm này bằng cách bổ sung một hoặc nhiều biến đầu vào.*

Làm cho hàm trở nên lặng bằng cách gọi nó từ Command Window và đảm bảo rằng bạn có thể hiển thị giá trị đầu ra.

Chương 6

Tìm nghiệm

6.1 Tại sao lại cần dùng hàm?

Chương vừa rồi đã giải thích một số ưu điểm của hàm, bao gồm

- Mỗi hàm có không gian làm việc riêng của nó, vì vậy dùng hàm sẽ tránh được xung đột về tên.
- Các hàm rất hợp với cách Xây dựng dần: bạn có thể gỡ lỗi phần thân của hàm trước (dưới dạng tập tin lệnh), rồi gói nó vào trong một hàm, sau đó khái quát hóa bằng cách thêm các biến đầu vào.
- Hàm cho phép ta chia một vấn đề lớn thành những phần nhỏ để xử lý từng phần một, rồi lắp ghép trở lại thành lời giải hoàn chỉnh.
- Một khi đã có hàm chạy được, bạn có thể quên đi những chi tiết về cách hoạt động của nó, mà chỉ cần biết nó làm gì. Quá trình trừu tượng hóa này là một cách thức quan trọng để ta quản lý được sự phức tạp của những chương trình lớn.

Một lý do khác khiến bạn phải cân nhắc việc dùng hàm là nhiều công cụ quan trọng của Scilab yêu cầu bạn phải viết hàm. Chẳng hạn, ở chương này ta sẽ dùng `fzero` để tìm nghiệm của phương trình phi tuyến. Sau đó ta sẽ dùng `ode` để tìm nghiệm xấp xỉ của các phương trình vi phân.

6.2 Ánh xạ

Trong toán học, **ánh xạ** là sự tương ứng giữa một tập hợp gọi là **tập nguồn** và một tập hợp khác được gọi là **tập đích**. Với mỗi phần tử của tập nguồn, phép ánh xạ sẽ chỉ ra phần tử tương ứng của tập đích.

Bạn có thể tưởng tượng một dãy như là ánh xạ từ tập các số nguyên dương đến tập các phần tử của dãy đó. Bạn có thể tưởng tượng véc-tơ như một ánh xạ từ tập các chỉ số đến các phần tử. Trong các trường hợp này, những ánh xạ là **rời rạc** vì các phần tử trong tập nguồn là đếm được.

Bạn cũng có thể tưởng tượng một hàm như một ánh xạ từ số liệu đầu vào đến số liệu đầu ra, nhưng trong trường hợp này tập nguồn là **liên tục** vì số liệu đầu vào có thể nhận bất kì giá trị nào chứ không riêng gì các số nguyên. (Chặt chẽ mà nói, tập hợp của các số có dấu phẩy động là rời rạc, nhưng vì các số dấu phẩy động nhằm biểu diễn cho các số thực, nên ta hiểu rằng chúng liên tục.)

6.3 Nói thêm về cách kí hiệu

Trong chương này, tôi bắt đầu nói về các hàm toán học, và tôi sẽ dùng dạng kí hiệu mà có thể chưa gặp bao giờ.

Nếu bạn đã học đến hàm qua môn toán, bạn có thể thấy kí hiệu như sau

$$f(x) = x^2 - 2x - 3$$

với ý nghĩa rằng f là một hàm chiếu từ x đến $x^2 - 2x - 3$. Vấn đề là $f(x)$ cũng được dùng để chỉ giá trị của f tương ứng với một giá trị của x. Vì vậy, tôi không thích các kí hiệu này. Tôi ưa dùng kí hiệu sau hơn:

$$f : x \to x^2 - 2x - 3$$

với ý nghĩa rằng "f là hàm chiếu từ x đến $x^2 - 2x - 3$." Trong Scilab, điều này được diễn đạt bởi:

```
function res = error_func(x)
    res = x^2 - 2*x -3;
endfunction
```

Tôi sẽ sớm giải thích lý do tại sao hàm này được gọi là `error_func`. Bây giờ, ta hãy quay trở lại việc lập trình.

6.4 Phương trình phi tuyến

Việc "giải" phương trình có nghĩa là gì? Điều này dường như quá rõ ràng, nhưng tôi muốn chúng ta dành một phút để nghĩ về nó, bắt đầu với một câu hỏi đơn giản: giả sử ta muốn biết giá trị của một biến, x, nhưng tất cả những gì ta biết về nó chỉ là một hệ thức $x^2 = a$.

Nếu đã học môn đại số, chắc bạn biết cách "giải" phương trình này: chỉ cần lấy căn bậc hai của hai vế và ta có $x = \sqrt{a}$. Sau đó, khi thoải mái vì đã giải xong, bạn chuyển sang bài toán tiếp theo.

Nhưng thực sự bạn đã làm gì? Hệ thức mà bạn rút ra tương đương với hệ thức ở đề bài—chúng có cùng thông tin về x—nhưng tại sao hệ thức thứ hai lại được ưa chuộng hơn thứ nhất?

Có hai lý do sau. Đầu tiên là hệ thứ thứ hai đã "tường minh theo x;" bởi vì chỉ có x ở bên vế trái, ta có thể coi vế phải như là một phương thức dễ dàng để tính x, với giả sử là ta đã biết a.

Lý do còn lại là phương thức tính toán này được viết dưới dạng các phép toán mà ta biết cách thực hiện. Giả sử rằng ta biết cách tính căn bậc hai, ta có thể tính được giá trị của x với giá trị a bất kì.

Khi ta nói về giải phương trình, ý nghĩa thông thường của nó là kiểu như "đi tìm một hệ thức tương đương trong đó một ẩn được viết dưới dạng tường minh." Trong phạm vi cuốn sách này, một hệ thức như vậy được tôi gọi là **nghiệm giải tích**, để phân biệt với **nghiệm số trị**, là thứ mà ta cần tìm trong phần tiếp theo đây.

Để ví dụ cho việc tìm nghiệm số trị, ta hãy xét phương trình $x^2 - 2x = 3$. Bạn có thể giải phương trình này theo cách giải tích, bằng phép phân tích thừa số hay dùng công thức giải phương trình bậc hai, để tìm được hai nghiệm của nó, $x = 3$ và $x = -1$. Một cách khác là bạn có thể giải phương trình bằng cách viết $x = \sqrt{2x + 3}$.

Phương trình này không tường minh, vì x xuất hiện ở cả 2 vế, vì vậy chưa rõ là bước biến đổi này có ích gì. Nhưng giả như là vì lý do nào đó ta biết có một nghiệm gần với 4, ta có thể lấy $x = 4$ làm "ước đoán ban đầu," rồi dùng phương trình $x = \sqrt{2x + 3}$ lặp lại nhiều lần để tính những liên tiếp những giá trị xấp xỉ của nghiệm.

Sau đây là điều có thể xảy ra:

```
--> x = 4;

--> x = sqrt(2*x+3)
```

```
x = 3.3166248

--> x = sqrt(2*x+3)

x = 3.1037477

--> x = sqrt(2*x+3)

x = 3.0343855

--> x = sqrt(2*x+3)

x = 3.01144

--> x = sqrt(2*x+3)

x = 3.0038109
```

Sau mỗi lượt lặp, x đã gần hơn đáp số đúng, và sau 5 lần lặp, sai số tương đối chỉ còn khoảng 0.1%, vốn đã đạt yêu cầu cho mọi mục đích tính toán.

Các kĩ thuật giúp tính ra nghiệm số trị được gọi là **phương pháp số**. Điều hay ở phương pháp mà tôi vừa trình bày là nó đơn giản, nhưng không phải lúc nào cũng hoạt động được như ở ví dụ trên, và thực tế nó thường không được dùng nhiều. Ta sẽ xem một phương pháp thông dụng hơn ngay sau đây.

6.5 Tìm nghiệm

Một phương trình phi tuyến như $x^2 - 2x = 3$ là một khẳng định đẳng thức chỉ đúng với một số ít các giá trị của x và sai với tất cả các giá trị khác. Một giá trị khiến cho đẳng thức đúng được gọi là nghiệm; các giá trị khác không phải nghiệm. Nhưng với bất kì một giá trị không phải nghiệm cho trước, cũng chẳng có dấu hiệu gì cho thấy nó gần hay xa một nghiệm, hay ta có thể tìm nghiệm trong khoảng nào.

Để giải quyết hạn chế này, ta cần viết lại phương trình phi tuyến dưới dạng bài toán tìm nghiệm:

- Bước đầu tiên là định nghĩa một "hàm sai số" để tính xem một giá trị cho trước của x cách xa nghiệm là bao nhiêu.

Ở ví dụ này, hàm sai số là

$$f : x \to x^2 - 2x - 3$$

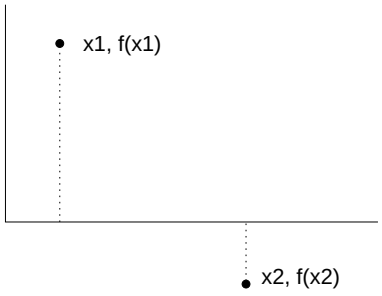
Bất kì giá trị nào của x làm cho $f(x) = 0$ chính là một nghiệm của phương trình ban đầu.

- Bước tiếp theo là tìm các giá trị của x làm cho $f(x) = 0$. Các giá trị này được gọi là **nghiệm của phương trình**.

Việc tìm nghiệm rất hợp với cách giải số trị vì ta có thể dùng các giá trị của f, được tính từ những giá trị khác nhau của x, để suy luận hợp lý về vị trí cần tìm nghiệm.

Chẳng hạn, nếu ta có thể tìm hai giá trị x_1 và x_2 sao cho $f(x_1) > 0$ và $f(x_2) < 0$, thì ta có thể chắc rằng có ít nhất một nghiệm nằm giữa x_1 và x_2 (miễn là f liên tục). Trong trường hợp này, ta sẽ nói rằng x_1 và x_2 bao một nghiệm.

Sau đây là một hình minh họa cho tình huống nói trên:



Nếu như đó là tất cả những gì bạn biết về f, thì bạn sẽ tìm nghiệm ở đâu? Nếu bạn nói "điểm chính giữa x_1 và x_2," thì xin chúc mừng bạn! Bạn đã tìm ra phương pháp số có tên là phân đôi!

Nếu bạn nói, "tôi sẽ nối hai điểm chấm bằng một đường thẳng rồi tính nghiệm của hàm đường thẳng này," thì xin chúc mừng bạn! Bạn đã tìm ra phương pháp cát tuyến!

Còn nếu bạn nói, "tôi sẽ tính f tại một điểm thứ ba, và vẽ một đường parabol đi qua ba điểm này, rồi tìm các nghiệm của hàm parabol," thì... ồ, có lẽ bạn sẽ không nói vậy đâu.

Cuối cùng, nếu bạn nói, "tôi sẽ dùng hàm lập sẵn của Scilab trong đó kết hợp những đặc điểm hay nhất của một số thuật toán mạnh và hiệu quả," thì bạn đã sẵn sàng chuyển sang mục tiếp theo.

6.6 fzero

`fzero` là một hàm lập sẵn của Scilab trong đó kết hợp các đặc điểm tốt nhất của một vài phương pháp số mạnh và hiệu quả.

Để dùng được `fzero`, bạn phải định nghĩa một hàm Scilab để tính hàm sai số suy từ phương trình phi tuyến ban đầu, và bạn phải cung cấp một giá trị ước đoán ban đầu về vị trí nghiệm.

Ta đã thấy một ví dụ của hàm sai số:

```
function res = error_func(x)
    res = x^2 - 2*x -3;
endfuction
```

Bạn có thể gọi `error_func` từ Console, và đảm bảo rằng có các nghiệm ở 3 và -1.

```
--> error_func(3)
ans = 0.

--> error_func(-1)
ans = 0.
```

Nhưng hãy giả vờ rằng ta không biết chắc vị trí của các nghiệm; ta chỉ biết rằng một nghiệm ở gần 4. Sau đó ta có thể gọi `fzero` như sau:

```
--> fsolve(4, error_func)
ans = 3.
```

Được rồi! Ta đã tìm thấy một trong số các nghiệm.

Đối số thứ nhất là giá trị ước đoán ban đầu, còn đối số thứ hai là tên hàm. Nếu ta cung cấp một ước đoán khác, thì (đôi khi) ta sẽ nhận được nghiệm khác.

```
--> fsolve(-2, error_func)
ans = - 1.
```

Bạn có thể tìm được nhiều nghiệm ở gần nhiều điểm giá trị ước đoán:

```
--> fsolve([-2, 4], error_func)

ans = - 1.   3.
```

Đối số thứ hai thực ra là một véc-tơ chứa hai phần tử. Toán tử ngoặc vuông là một trong số vài cách làm tiện lợi để tạo ra một véc-tơ mới.

Bạn có thể tò mò muốn biết `fzero` gọi đến hàm sai số bao nhiêu lần, và gọi ở những vị trí nào. Nếu bạn sửa `error_func` sao cho nó hiển thị giá trị của `x` mỗi lần được gọi, rồi chạy lại `fzero` thì bạn sẽ thu được:

```
--> fzero(4, error_func)
x = 4.
x = 4.0000001
x = 3.1666667
x = 3.0322581
x = 3.0012804
x = 3.0000102
x = 3.
x = 3.
x = 3.
ans = 3
```

Không ngạc nhiên là, nó bắt đầu bằng việc tính $f(4)$. Sau mỗi lần lặp, bước tìm kiếm đã co ngắn lại; `fsolve` dừng khi khoảng bao quá nhỏ và nghiệm được ước tính chính xác đến 10 chữ số. Nếu không cần đạt độ chính xác đến thế, bạn có thể bảo `fsolve` cho một đáp số thô hơn một cách nhanh chóng (hãy xem lời giải thích cách dùng hàm để biết thêm chi tiết).

Còn một điều trục trặc nữa có thể xảy ra khi bạn dùng `fsolve`, nhưng lỗi này có vẻ như ít do bạn gây ra. Có khả năng là `fsolve` không thể tìm được nghiệm.

Nói chung `fsolve` khá mạnh, vì vậy bạn có thể không bao giờ gặp vấn đề khi dùng nó, nhưng nhớ rằng không có bảo đảm gì là `fsolve` sẽ chạy, đặc biệt là nếu bạn chỉ cung cấp được một giá trị làm ước đoán ban đầu. Ngay cả khi bạn cung cấp một khoảng bao nghiệm, mọi thứ vẫn có thể trục trặc nếu như hàm sai số bị gián đoạn.

6.7 Tìm giá trị ước đoán ban đầu

Giá trị ước đoán ban đầu của bạn gần đúng bao nhiêu, thì khả năng hoạt động của `fsolve` sẽ cao bấy nhiêu, và càng cần ít lần lặp hơn.

Khi bạn giải bài toán trong thực tế, thường bạn sẽ có sự nhận định về đáp số. Nhận định này thường đảm bảo có được một ước đoán ban đầu gần đúng để tìm nghiệm.

Một cách làm khác là vẽ đồ thị hàm số và xem nếu bạn có thể tìm nghiệm gần đúng bằng mắt thường hay không. Nếu bạn có một hàm, như `error_func` nhận một biến vô hướng và trả lại một biến đầu ra vô hướng, thì bạn có thể vẽ nó bằng `ezplot`:

```
--> x = [-2 : 0.1 : 5];
--> ezplot(x, error_func(x))
```

Đối số thứ nhất là các hoành độ, đối số thứ hai là các tung độ của những điểm số liệu mà đồ thị hàm số đó đi qua. Ở đây ta vẽ hàm trong phạm vi hoành độ từ -2 đến 5 với các điểm cách đều nhau 0,1. Như vậy kể cả hai điểm đầu cuối thì sẽ có 71 điểm được phát sinh để vẽ hàm này.

Có thể bạn muốn xóa lệnh in giá trị của `x` ra khỏi hàm `error_func` trước khi gọi hàm vẽ.

6.8 Nói thêm về xung đột tên

Các hàm và biến chiếm cùng một "không gian tên," nghĩa là mỗi khi một tên xuất hiện trong biểu thức, Scilab bắt đầu đi tìm một biến có tên như vậy, và nếu biến đó không tồn tại, thì tìm một hàm.

Kết quả là, nếu bạn có một biến có cùng tên với một hàm thì biến sẽ **lấn át** hàm. Chẳng hạn, nếu bạn gán một giá trị cho `sin`, rồi thử dùng hàm `sin`, bạn *có thể* sẽ nhận được lỗi:

```
--> sin = 3;
Warning : redefining function: sin
--> x = 5;
--> sin(x)
         !--error 21
Invalid index.
```

Trong ví dụ này, vấn đề đã rõ ràng. Vì giá trị của `sin` là một số vô hướng, mà số vô hướng thực ra là ma trận 1x1, Scilab sẽ cố gắng truy cập phần tử thứ 5 của ma trận và thấy rằng không có phần tử nào như vậy. Dĩ nhiên là nếu "lời gọi hàm" này đặt cách xa lệnh gán thì thông báo lỗi này còn có thể làm ta bối rối hơn nữa.

Nhưng điều duy nhất còn tệ hơn cả nhận thông báo lỗi là *không* nhận được thông báo lỗi. Nếu giá trị của `sin` là một véc-tơ, hoặc nếu giá trị của `x` đã nhỏ hơn thì bạn gặp rắc rối thực sự.

```
--> sin = 3;
--> sin(1)

ans = 3.
```

Hãy xem, sin của 1 đâu có bằng 3 !

Lỗi ngược lại cũng có thể xuất hiện nếu bạn thử truy cập một biến không xác định mà nó tình cờ trùng tên với một hàm. Chẳng hạn, nếu bạn đã có một hàm tên là `f`, và bây giờ thử tăng một biến cùng tên `f` (trước đó biến này quên không được khởi tạo), bạn sẽ thấy:

```
-->f = f+1
         !--error 144
Undefined operation for the given operands.
check or define function %mc_a_s for overloading.
```

Không có một cách làm chung để tránh tất cả những lỗi xung đột kiểu này, nhưng bạn có thể giảm khả năng xảy ra lỗi bằng cách chọn các tên biến không trùng với các hàm có sẵn, và bằng cách chọn tên các hàm mà chúng ít có khả năng bị lấy làm tên biến. Đó là lý do tại sao trong Mục 6.3 tôi gọi hàm sai số là `error_func` thay vì `f`. Tôi thường đặt tên các hàm kết thúc với `func`, và điều này cũng giúp ích.

6.9 Gỡ lỗi bằng bốn hành động

Khi gỡ lỗi một chương trình, đặc biệt nếu bạn phải đương đầu với một lỗi khó, có bốn việc mà bạn cần thử làm:

đọc: Kiểm tra mã lệnh, tự đọc nhẩm, và kiểm tra xem có đúng là chương trình này có ý định thực hiện đúng điều bạn muốn không.

chạy: Thử nghiệm bằng cách sửa đổi và chạy các phiên bản mã lệnh khác nhau. Thường nếu bạn hiển thị đúng thứ ở đúng chỗ trong chương trình, vấn đề sẽ trở nên hiểu nhiên, nhưng đôi khi bạn phải dành thời gian để dựng dàn giáo.

nghiền ngẫm: Hãy dành thời gian suy nghĩ! Đó là loại lỗi gì: cú pháp, thực thi, logic? Bạn có thể tìm được thông tin gì từ dòng thông báo lỗi, hay từ kết quả đầu ra của chương trình? Loại lỗi gì có thể gây ra vấn đề mà bạn đang thấy? Lần gần nhất bạn đã thay đổi gì ở mã lệnh, trước khi xảy ra lỗi?

rút lui: Đến một lúc nào đó, cách tốt nhất là rút lui, hoàn lại những thay đổi mới nhất, đến khi bạn trở về trạng thái của chương trình hoạt động, mà bạn hiểu được. Sau đó bạn có thể bắt đầu lập lại chương trình.

Những người mới lập trình đội khi lún sâu vào một trong những hoạt động này mà quên những hoạt động khác. Mỗi hoạt động có những nhược điểm riêng của nó.

Chẳng hạn, việc đọc mã lệnh có thể giúp ích nếu vấn đề nằm ở lỗi đánh máy, nhưng vô ích nếu vấn đề ở chỗ hiểu sai về khái niệm. Nếu bạn không hiểu chương trình làm gì, thì dù có đọc lại 100 lần bạn cũng chẳng tìm thấy lỗi, vì lỗi nằm ngay trong đầu bạn.

Chạy thử nghiệm có thể giúp ích, đặc biệt với các kiểm tra nhỏ, đơn giản. Nhưng nếu chạy thử mà không nghĩ hoặc đọc mã lệnh thì bạn có thể rơi vào tình trạng mà tôi gọi là "lập trình bước ngẫu nhiên"—một quá trình thực hiện những thay đổi ngẫu nhiên đến khi chương trình hoạt động đúng. Khỏi phải nói, lập trình bước ngẫu nhiên có thể tốn rất nhiều thời gian.

Lối thoát là dành thêm thời gian suy nghĩ. Gỡ lỗi cũng giống như môn khoa học thực nghiệm. Bạn ít nhất phải có được giả thiết về vấn đề. Nếu có nhiều khả năng, hãy cố gắng thực hiện phép thử để loại trừ một trong số các khả năng đó.

Nghỉ ngơi đôi khi cũng giúp ích cho suy nghĩ. Nói chuyện cũng như vậy. Nếu bạn giải thich vấn đề cho ai đó (và ngay cả cho bản thân), bạn đôi khi có thể tìm thấy lời giải ngay trước khi đặt xong câu hỏi.

Nhưng ngay cả những kĩ thuật gỡ lỗi tốt nhất cũng thất bại nếu có quá nhiều lỗi, hay nếu lỗi bạn cố sửa đang quá lớn và phức tạp. Đôi khi lựa chọn hay nhất là rút lui, làm đơn giản chương trình đến khi bạn có được một chương trình hoạt động, rồi sau đó phát triển lại.

Những người mới lập trình thường miễn cưỡng không muốn rút lui, vì họ không thể chịu được nếu phải xóa một dòng lệnh (dù dòng lệnh đó có sai đi nữa). Nếu bạn cảm thấy được, thì hãy sao lưu chương trình vào một tập tin khác trước khi lược bỏ mã lệnh. Sau đó bạn dán lại những mảnh chương trình vào, mỗi lúc một ít.

Tóm lại, sau đây là Định luật thứ chín về gỡ lỗi:

> Để tìm ra một lỗi khó, cần phải đọc, chạy thử, suy nghĩ, và đôi khi rút lui. Nếu bạn lún sâu vào một trong những hoạt động này mà không có kết quả, hãy thử hoạt động khác.

6.10 Thuật ngữ

nghiệm giải tích: Cách giải phương trình bằng việc thực hiện biến đổi đại số để rút ra một biểu thức tường minh để tính giá trị của một ẩn.

nghiệm số trị: Cách giải phương trình bằng việc tìm một giá trị số thỏa mãn phương trình đó, thường chỉ là xấp xỉ.

phương pháp số: Phương pháp (hoặc thuật toán) để tính ra nghiệm số trị.

ánh xạ: Sự tương ứng giữa các phần tử thuộc một tập hợp (tập nguồn) và các phần tử thuộc tập hợp khác (tập đích). Bạn có thể hình dung các dãy, véc-tơ và hàm như các loại ánh xạ khác nhau.

tập nguồn: Tập hợp các giá trị điểm đầu của ánh xạ.

tập đích: Tập hợp các giá trị điểm cuối của ánh xạ.

tập rời rạc: Một tập hợp, như tập các số nguyên, trong đó các phần tử là đếm được.

tập liên tục: Một tập hợp, như tập các số thực, trong đó các phần tử không đếm được. Bạn có thể hình dung tập các số dấu phẩy động như một tập liện tục.

nghiệm (của hàm): Giá trị trong tập xác định (tập nguồn) của hàm mà ánh xạ của nó chiếu đến 0.

chuôi của hàm: Trong Scilab, chuôi của hàm là một cách tham chiếu đến hàm bằng một cái tên (và truyền nó dưới dạng một đối số) mà không gọi hàm đó.

lấn át: Trường hợp xung đột về tên trong đó một định nghĩa mới khiến cho định nghĩa cũ trở nên không truy cập được. Trong Scilab, các tên biến có thể lấn át được các hàm lập sẵn (có thể dẫn đến những kết quả rất hài hước).

6.11 Bài tập

Exercise 6.1 *1. Hãy viết một hàm có tên* `cheby6` *để lượng giá đa thức Chebyshev bậc 6. Hàm cần nhận một biến đầu vào,* x*, và trả lại*

$$32x^6 - 48x^4 + 18x^2 - 1 \tag{6.1}$$

2. *Hãy dùng* `plot` *để hiển thị một đồ thị của hàm này trong khoảng từ 0 đến 1. Hãy tìm các nghiệm trong khoảng này.*

3. *Hãy dùng* `fzero` *để tìm càng nhiều nghiệm càng tốt. Liệu* `fzero` *có luôn tìm được nghiệm gần sát giá trị ước đoán ban đầu nhất hay không?*

Exercise 6.2 *Khối lượng riêng của một con vịt,* ρ*, là* $0.3g/\mathrm{cm}^3$ *(bằng 0,3 lần khối lượng riêng của nước).*

Thể tích của một khối cầu có bán kính* r *là* $\frac{4}{3}\pi r^3$*.*

Nếu khối cầu có bán kính r *được nhúng trong nước ngập đến độ sâu* d*, thì thể tích của khối cầu phần bị ngập là*

$$volume = \frac{\pi}{3}(3rd^2 - d^3) \quad khi \quad d < 2r$$

Một vật thể luôn nổi lơ lửng ở độ cao sao cho trọng lượng phần bị chìm trong nước đúng bằng trọng lượng của vật ban đầu.

Giả sử rằng con vịt có hình dáng tương đương một quả cầu bán kính 10 cm, phần nhúng nước của con vịt sẽ sâu bao nhiêu?

Sau đây là một số gợi ý để bạn giải bài này:

- *Hãy viết một phương trình liên hệ giữa* ρ*,* d *và* r*.*
- *Sắp xếp lại phương trình sao cho vế phải là số không. Mục tiêu của chúng ta là tìm những giá trị của* d *là nghiệm phương trình này.*
- *Hãy viết một hàm Scilab để lượng giá hàm đã thành lập. Hãy kiểm tra nó, rồi sửa nó thành một hàm lặng.*
- *Hãy dự đoán giá trị của* d_0 *để làm ước tính ban đầu.*
- *Dùng* `fzero` *để tìm ra nghiệm gần* d_0*.*
- *Kiểm tra để bảo đảm rằng kết quả có ý nghĩa. Đặc biệt, kiểm tra điều kiện* $d < 2r$*, bởi nếu không thì công thức thể tích sẽ sai!*
- *Thử các giá trị khác nhau của* ρ *và* r *và xem hiện tượng có diễn ra như bạn mong đợi hay không. Điều gì sẽ xảy ra khi* ρ *tăng? Lên đến vô cùng? Hạ xuống bằng 0 ? Điều gì sẽ xảy ra khi* r *tăng? Lên đến vô cùng? Hạ xuống bằng không?*

*Ví dụ này được chính sửa từ Gerald and Wheatley, *Applied Numerical Analysis*, Fourth Edition, Addison-Wesley, 1989.

Chương 7

Hàm số của véc-tơ

7.1 Hàm số và tập tin

Đến giờ ta mới chỉ đưa một hàm vào trong mỗi tập tin. Cũng có thể đặt nhiều hàm vào trong một tập tin, nhưng chỉ có hàm đầu tiên, **hàm cấp cao nhất** mới gọi được từ Console. Các **hàm phụ trợ** khác có thể được gọi từ bất kì đâu trong tập tin, nhưng không thể gọi từ tập tin khác.

Những chương trình lớn thường cần đến nhiều hàm; việc giữ tất cả các hàm trong cùng một tập tin tuy tiện lợi, nhưng làm cho việc gỡ lỗi trở nên khó khăn vì bạn không thể gọi các hàm phụ trợ từ Console.

Để giúp giải quyết vấn đề này, tôi thường dùng hàm cấp cao nhất để phat triển và thử nghiệm các hàm phụ trợ. Chẳng hạn, tôi có thể tạo ra một tập tin tên là `duck.sce` và khởi đầu với một hàm cấp cao nhất có tên là `duck` mà không nhận vào bất kì biến đầu vào cũng như trả lại bất kì biến đầu ra nào.

Sau đó tôi sẽ viết một hàm có tên là `error_func` để lượng giá hàm sai số cho `fsolve`. Để thử nghiệm `error_func` tôi sẽ gọi nó từ `duck` rồi gọi `duck` từ Console.

Bản nháp chương trình đầu tiên của tôi có thể trông như sau:

```
function res = duck()
    err = error_func(10);
    res = 0;
endfunction

function res = error_func(h)
    rho = 0.3;      // mật độ tính theo g / cm^3
```

```
    r = 10;          // bán kính tính theo cm
    res = h;
endfunction
```

Dòng lệnh `res = h` chưa xong, nhưng để phục vụ mục đích thử nghiệm thì từng ấy mã lệnh cũng đủ. Một khi đã hoàn thành và thử nghiệm `error_func`, tôi sẽ sửa `duck` sao cho nó dùng `fsolve`.

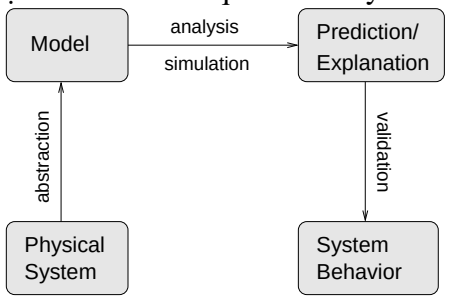
Ở bài toán này tôi có thể chỉ cần đến hai hàm, nhưng nếu có nhiều hàm hơn, tôi có thể lần lượt viết và thử nghiệm từng hàm một, rồi sau đó kết hợp chúng lại thành một chương trình chạy được.

7.2 Mô hình hóa hiện tượng vật lý

Các ví dụ mà chúng ta đã đề cập đến giờ đều liên quan đến toán học; Bài tập 6.2, "bài toán con vịt," là ví dụ đầu tiên mà chúng ta tìm hiểu một hiện tượng vật lý. Nếu bạn chưa làm bài này, hãy quay lại và ít nhất là đọc kỹ bài toán.

Cuốn sách này được viết về chủ đề **mô hình hóa vật lý**, vì vậy tôi nên giải thích ý nghĩa của cụm từ trên. Mô hình hóa vật lý là một công đoạn thiết lập những phỏng đoán về các hệ thống vật lý và giải thích biểu hiện của chúng. Một **hệ vật lý** là thứ tồn tại khách quan mà ta cần quan tâm, chẳng hạn như con vịt.

Hình vẽ sau minh họa các bước của quá trình này:



Một **mô hình** là một dạng biểu diễn được đơn giản hóa cho một hệ vật lý. Quá trình lập một mô hình được gọi là **trừu tượng hóa**. Trong phạm vi bàn đến ở đây, "trừu tượng" ngược nghĩa với "hiện thực;" một mô hình trừu tượng ít có sự tương đồng trực tiếp đến hệ vật lý mà nó mô phỏng, cũng như hội họa trừu tượng không trực tiếp biểu diễn hình ảnh của vạn vật ngoài đời. Một mô hình hiện thực là mô hình bao gồm nhiều chi tiết hơn và tương đồng hơn với thế giới thực.

Sự trừu tượng hóa sẽ dẫn đến việc phải quyết định đúng đắn về việc đưa các yếu tố nào vào mô hình và đơn giản bớt hoặc lược bỏ những yếu tố nào. Chẳng hạn, trong bài toán con vịt, ta xét đến trọng lượng riêng của vịt và lực đẩy nổi của

nước, nhưng bỏ qua lực đẩy nổi do không khí và tác động đạp nước của chân vịt. Ta cũng đơn giản hóa hình dạng con vịt bằng cách giả thiết rằng phần con vịt ngập trong nước cũng gần giống một chỏm cầu. Và chúng ta đã sử dụng những ước tính sơ lược về kích thước và khối lượng của con vịt.

Trong số các quyết định giản hóa trên, có cái chấp nhận được. Khối lượng riêng của con vịt cao hơn của không khí rất nhiều nên ảnh hưởng của lực đẩy nổi của không khí có lẽ sẽ rất nhỏ. Những quyết định khác, như hình dạng khối cầu, thì khó chấp nhận hơn, nhưng có ích. Hình dạng thực tế của con vịt rất phức tạp; mô hình khối cầu giúp ta có thể tính được một kết quả gần đúng mà không phải thực hiện đo đạc chi tiết hình dạng của những con vịt thật.

Một mô hình giống thật hơn chưa chắc đã tốt hơn. Mô hình rất có ích vì chúng có thể được phân tích về mặt toán học và mô phỏng về mặt số trị. Các mô hình quá giống thật sẽ khó có thể mô phỏng được và hoàn toàn không thể phân tích được.

Một mô hình được gọi là thành công nếu nó đạt yêu cầu đề ra. Nếu ta chỉ cần một ước đoán rất thô sơ về phần của con vịt ngập trong nước thì mô hình khối cầu cũng là đủ. Nếu ta cần một lời giải chính xác hơn (vì lý do nào đó) thì ta có thể cần đến một mô hình giống thật hơn.

Việc kiểm tra xem một mô hình có đủ tính đúng đắn hay không được gọi là **thẩm định**. Hình thức chặt chẽ nhất của thẩm định là tiến hành đo trên hệ vật lý và so sánh với kết quả dự đoán của một mô hình.

Nếu điều này không thực hiện được, ta vẫn còn những hình thức thẩm định khác tuy không chặt chẽ bằng. Một hình thức là so sánh nhiều mô hình của cùng một hệ. Nếu chúng không thống nhất thì có biểu hiện là (ít nhất) một trong số các mô hình đó đã sai, và độ lớn của sự khác biệt chính là dấu hiệu về độ tin cậy của những ước tính đó.

Đến giờ chúng ta chỉ thấy được một mô hình vật lý, vì vậy mà một phần của đoạn thảo luận trên vẫn chưa thể làm rõ. Ta sẽ quay trở lại chủ đề này sau, nhưng trước hết cần tìm hiểu thêm về véc-tơ.

7.3 Véc-tơ với vai trò là biến đầu vào

Vì nhiều hàm lập sẵn chấp nhận véc-tơ làm đối số, nên sẽ không có gì lạ khi bạn có thể viết các hàm nhận vào véc-tơ. Sau đây là một ví dụ (ngốc nghếch):

```
function display_vector(X)
    disp(X)
endfunction
```

Chẳng có gì đặc biệt về hàm này cả. Điểm khác biệt duy nhất so với hàm vô hướng mà ta gặp từ trước đó là tôi đã dùng một chữ cái viết in để gợi nhớ rằng `X` là một véc-tơ.

Sau đây là một ví dụ khác về một hàm không trả về giá trị nào; nó chỉ hiển thị giá trị của biến đầu vào:

```
--> display_vector(1:3)

 1.     2.     3.
```

Tiếp theo là một ví dụ lý thú hơn, trong đó gói đoạn mã lệnh ở Mục 4.12 để tính tổng các phần tử của một véc-tơ:

```
function res = mysum(X)
    total = 0;
    for i=1:length(X)
        total = total + X(i);
    end
    res = total;
endfunction
```

Tôi gọi nó là `mysum` để tránh xung đột tên với hàm lập sẵn `sum`, nhưng về công dụng thì hai hàm giống nhau.

Sau đây là cách gọi nó từ Console:

```
--> total = mysum(1:3)

total = 6.
```

Vì hàm này có một giá trị trả về nên tôi đã cố ý gán nó cho một biến.

7.4 Véc-tơ đóng vai trò là biến đầu ra

Cũng không có gì sai khi gán véc-tơ cho một biến đầu ra. Sau đây là một ví dụ trong đó gói đoạn mã lệnh ở Mục 4.13:

```
function res = myapply(X)
    for i=1:length(X)
        Y(i) = X(i)^2
    end
    res = Y
endfunction
```

Lý tưởng nhất lẽ ra phải thay đổi tên của biến đầu ra thành `Res`, để nhắc rằng nó nhận một giá trị véc-tơ, nhưng tôi đã không làm vậy.

Sau đây là cách dùng `myapply`:

```
--> V = myapply(1:3)

V =
   1.
   4.
   9.
```

Exercise 7.1 *Hãy viết một hàm có tên là* `find_target` *để gói đoạn lệnh từ Mục 4.14, để tìm vị trí của giá trị mong muốn trong một véc-tơ.*

7.5 Véc-tơ hóa hàm của bạn

Những hàm tính trên véc-tơ thường cũng dùng được với các số vô hướng, vì Scilab coi rằng số vô hướng là véc-tơ có chiều dài bằng 1.

```
--> mysum(17)

ans = 17.

--> myapply(9)

ans = 81.
```

Không may là, điều còn lại không phải luôn đúng. Nếu bạn viết một hàm để tính cho đầu vào là số vô hướng, thì nó có thể không dùng được cho véc-tơ.

Nhưng có trường hợp vẫn được! Nếu các toán tử và hàm bạn dùng trong thân hàm có thể tính được với véc-tơ thì hàm của bạn sẽ tính được cho véc-tơ.

Chẳng hạn, sau đây chính là hàm đầu tiên mà ta viết:

```
function res = myfunc (x)
    s = sin(x)
    c = cos(x)
    res = abs(s) + abs(c)
endfunction
```

Này! Nó làm việc được với véc-tơ đấy:

```
--> Y = myfunc(1:3)

Y = 1.3817733    1.3254443    1.1311125
```

Đến đây, tôi muốn dừng lại một chút để thừa nhận rằng tôi đã khắt khe một chút khi trình bày về Scilab, vì có một số đặc điểm mà tôi nghĩ đã làm khó người mới học một cách không cần thiết. Nhưng rồi cuối cùng, ở đây ta thấy được những đặc điểm chứng tỏ sức mạnh của Scilab.

Một số hàm khác ta đã viết lại không dùng được với véc-tơ, nhưng chúng có thể sửa được dễ dàng. Chẳng hạn, có hàm `hypotenuse` từ Mục 5.5:

```
function res = hypotenuse(a, b)
    res = sqrt(a^2 + b^2);
endfunction
```

```
--> A = [3,5,8];
--> B = [4,12,15];
--> C = hypotenuse(A, B)

C = 5.    13.    17.
```

Ở trường hợp này, nó ghép cặp các phần tử tương ứng từ hai véc-tơ đầu vào, nên các phần tử của `C` là độ dài cạnh huyền lần lượt của các cặp $(3, 4)$, $(5, 12)$ and $(8, 15)$.

Nói chung, nếu bạn muốn viết một hàm trong đó chỉ dùng các toán tử theo phần tử và các hàm làm việc trên véc-tơ, thì hàm mới viết cũng sẽ làm việc được với véc-tơ.

7.6 Tổng và hiệu

Một phép toán với véc-tơ quan trọng khác là **tổng lũy tích**, vốn nhận một véc-tơ đầu vào và tính một véc-tơ mới gồm tất cả các tổng thành phần của véc-tơ ban đầu. Theo kí hiệu toán, nếu V là véc-tơ ban đầu, thì các phần tử của tổng lũy tích, C, là:

$$C_i = \sum_{j=1}^{i} V_j$$

Nói cách khác, phần tử thứ i của C là tổng của i phần tử đầu tiên trong V. Scilab cung cấp một hàm có tên là `cumsum` để tính tổng lũy tích:

```
--> V = 1:5

V = 1.    2.    3.    4.    5.

--> C = cumsum(V)

C = 1.    3.    6.    10.   15.
```

Exercise 7.2 *Hãy viết một hàm có tên là* `cumulative_sum` *trong đó dùng một vòng lặp để tính tổng lũy tích của véc-tơ đầu vào.*

Phép tính ngược với `cumsum` là `diff`, vốn tính hiệu giữa các phần tử kế tiếp trong véc-tơ đầu vào.

```
--> D = diff(C)

D = 2.    3.    4.    5.
```

Lưu ý rằng véc-tơ đầu ra ngắn hơn 1 phần tử so với véc-tơ đầu vào. Vì vậy, phiên bản `diff` trong Scilab không thật sự là hàm ngược của `cumsum`. Nếu là hàm ngược thì ta đã có thể trông đợi `cumsum(diff(X))` bằng `X`:

```
--> cumsum(diff(V))

ans = 1.    2.    3.    4.
```

Nhưng nó đã không thỏa mãn điều kiện trên.

Exercise 7.3 *Hãy viết một hàm* `mydiff` *để tính nghịch đảo của* `cumsum`*, sao cho cả* `cumsum(mydiff(X))` *và* `mydiff(cumsum(X))` *đều trả lại* `X`*.*

7.7 Tích và thương

Dạng phép nhân của `cumsum` là `cumprod`, dùng để tính **tích số lũy tích**. Theo kí hiệu toán học thì:

$$P_i = \prod_{j=1}^{i} V_j$$

Trong Scilab, hàm đó như sau:

```
--> V = 1:5

V = 1.    2.    3.    4.    5.

--> P = cumprod(V)

P = 1.    2.    6.    24.   120.
```

Exercise 7.4 *Hãy viết một hàm tên là* `cumulative_prod` *trong đó dùng vòng lặp để tính tích số lũy tích từ véc-tơ đầu vào.*

Scilab không cung cấp một phiên bản phép nhân tương ứng với `diff`, mà lẽ ra đã có tên `ratio`, để tính tỉ số giữa các phần tử kế tiếp nhau trong véc-tơ đầu vào.

Exercise 7.5 *Hãy viết một hàm tên là* `myratio` *để tính nghịch đảo của* `cumprod`*, sao cho cả* `cumprod(myratio(X))` *và* `myratio(cumprod(X))` *đều trả lại* `X`*.*

Bạn có thể dùng một vòng lặp, hoặc nếu khéo hơn, có thể lợi dụng hệ thức $e^{\ln a + \ln b} = ab$*.*

Nếu áp dụng `myratio` *cho một véc-tơ có chứa các số Fibonacci, bạn có thể khẳng định được rằng tỉ số giữa các phần tử kế tiếp nhau hội tụ về tỉ số vàng,* $(1+\sqrt{5})/2$ *(xem Bài tập 4.6).*

7.8 Kiểm tra sự tồn tại

Đôi khi ta phải kiểm tra các phần tử của một véc-tơ xem nếu có bất kì phần tử nào thỏa mãn một điều kiện cho trước hay không. Chẳng hạn, bạn muốn biết rằng có phần tử số dương trong véc-tơ hay không. Theo logic học, điều kiện này được gọi là **kiểm tra sự tồn tại**, và nó được kí hiệu bởi dấu $\exists$, đọc là "tồn tại." Chẳng hạn, biểu thức sau

$$\exists x \text{ in } S : x > 0$$

nghĩa là "tồn tại phần tử x nào đó trong tập hợp S sao cho $x > 0$." Trong Scilab, ý này có thể được thể hiện tự nhiên bằng một hàm logic như `exists`, vốn trả lại 1 nếu có phần tử như vậy và 0 nếu không có.

```
function res = exists(X)
    for i=1:length(X)
        if X(i) > 0
            res = %t;
            return
        end
    end
    res = %f;
endfunction
```

Trước đây chưa bắt gặp lệnh `return`; nó tương tự như `break`, chỉ khác là nó thoát khỏi cả hàm chứ không chỉ vòng lặp. Điều đó rất cần đến trong trường hợp này vì ngay khi tìm được một phần tử dương, ta biết được câu trả lời (có tồn tại!) và có thể kết thúc hàm lập tức mà không cần xét đến các phần tử còn lại.

Nếu ta thoát ở cuối vòng lặp, điều đó nghĩa là ta không thể tìm thấy giá trị mong muốn (vì nếu có, ta đã gặp lệnh `return`).

7.9 Kiểm tra sự toàn vẹn

Một phép toán véc-tơ thường gặp khác là kiểm tra xem *tất cả* các phần tử có cùng thỏa mãn một điều kiện không; nó được gọi là **kiểm tra sự toàn vẹn** và được kí hiệu bằng dấu $\forall$ và được đọc là "với mọi." Vì vậy biểu thức này

$$\forall x \text{ in } S : x > 0$$

có nghĩa là "với mọi phần tử x trong tập hợp S, $x > 0$."

Một cách khá ngốc nghếch để lượng giá biểu thức này trong Scilab là đếm số phần tử thỏa mãn điều kiện trên. Một cách tốt hơn là rút gọn bài toán về kiểm tra sự tồn tại; nghĩa là viết lại

$$\forall x \text{ in } S : x > 0$$

theo dạng

$$\sim \exists x \text{ in } S : x \leq 0$$

Trong đó $\sim \exists$ nghĩa là "không tồn tại." Nói cách khác, kiểm tra để đảm bảo tất cả các phần tử phải dương thì cũng như kiểm tra điều ngược lại, có tồn tại phần tử không dương.

Exercise 7.6 *Hãy viết một hàm có tên* `forall` *nhận vào một véc-tơ và trả lại giá trị 1 nếu tất cả các phần tử đều dương và 0 nếu có bất kì phần tử nào không dương.*

7.10 Véc-tơ logic

Khi áp dụng một toán tử logic cho một véc-tơ, kết quả là một **véc-tơ logic**; nghĩa là một véc-tơ trong đó các phần tử đều là những giá trị logic 1 và 0.

```
--> V = -3:3

V = - 3.    - 2.    - 1.     0.     1.     2.     3.

--> L = V>0

L =  F     F     F     F     T     T     T
```

Ở ví dụ này, `L` là véc-tơ logic có các phần tử tương ứng với các phần tử của `V`. Với mỗi phần tử dương của `V`, phần tử tương ứng của `L` bằng 1.

Véc-tơ logic có thể được dùng như *cờ* để lưu giữ trạng thái của một điều kiện. Chúng thường được dùng với hàm `find`, vốn nhận vào một véc-tơ logic và trả lạ một véc-tơ bao gồm các chỉ số của những phần tử "đúng."

Áp dụng `find` vào `L` ta được

```
--> find(L)

ans = 5.     6.     7.
```

nghĩa là các phần tử thứ 5, 6 và 7 có giá trị bằng 1.

Nếu không có phần tử "đúng" nào, kết quả sẽ là véc-tơ rỗng.

```
--> find(V>10)

ans = []
```

Ví dụ này tính toán véc-tơ logic và truyền nó làm đối số cho `find` mà không gán nó vào một biến trung gian. Bạn có thể đọc đoạn mã lệnh này theo cách trừu tượng là "tìm tất cả những chỉ số của các phần tử trong `V` có giá trị lớn hơn 10."

Bạn cũng có thể dùng `find` để viết lại `exists` cho gọn hơn:

```
function res = exists(X)
    L = find(X>0)
    res = length(L) > 0
endfunction
```

Exercise 7.7 *Hãy viết một phiên bản của* `forall` *có dùng* `find`.

7.11 Thuật ngữ

hàm cấp cao nhất: Hàm đầu tiên trong một tập tin M; là hàm duy nhất có thể gọi đến từ Console hoặc từ một tập tin khác.

hàm phụ trợ: Hàm trong tập tin M nhưng không phải hàm cấp cao nhất; nó chỉ có thể được gọi từ một hàm khác trong cùng tập tin.

mô hình hóa vật lý: Quá trình đưa ra phỏng đoán về các hệ vật lý cũng như giải thích biểu hiện của chúng.

hệ vật lý: Thứ tồn tại trong thế giới thực mà chúng ta quan tâm nghiên cứu.

mô hình : Sự mô tả được đơn giản hóa của hệ vật lý, rất thích hợp cho việc phân tích và mô phỏng.

trừu tượng hóa: Quá trình xây dựng một mô hình bằng cách quyết định những yếu tố nào cần được giản hóa hoặc lược bỏ.

thẩm định: Kiểm tra xem mô hình có đạt yêu cầu sử dụng hay không.

kiểm tra sự tồn tại: Một điều kiện logic nhằm diễn đạt ý "có tồn tại" một phần tử thỏa mãn một thuộc tính nhất định trong tập hợp cho trước.

kiểm tra sự toàn vẹn: Một điều kiện logic nhằm diễn đạt ý tất cả các phần tử trong tập hợp đều có chung một thuộc tính nhất định.

véc-tơ logic: Một véc-tơ gồm các giá trị logic 1 hoặc 0, thường là kết quả của phép áp dụng một toán tử logic vào một véc-tơ ban đầu.

Chương 8

Phương trình vi phân thường

8.1 Phương trình vi phân

Phương trình vi phân là phương trình mô tả các đạo hàm của một hàm số chưa biết. "Giải phương trình vi phân" nghĩa là tìm một hàm số có các đạo hàm thỏa mãn phương trình đã cho.

Chẳng hạn, khi vi phuẩn sống trong môi trường đặc biệt thuận lợi thì tốc độ sinh trưởng tại bất kì thời điểm nào cũng tỉ lệ thuận với số vi khuẩn lúc đó. Có lẽ điều ta quan tâm là số vi khuẩn được biểu diễn dưới dạng hàm theo thời gian. Ta hãy định nghĩa f là hàm chiếu từ thời gian, t, đến số vi khuẩn, y. Dù không biết y bằng bao nhiêu, nhưng ta vẫn có thể viết một phương trình để mô tả nó:

$$\frac{df}{dt} = af$$

trong đó a là hằng số đặc trưng cho mức độ vi khuẩn tăng nhanh bao nhiêu.

Lưu ý rằng cả hai vế của phương trình đều là hàm số. Khi ta nói hai hàm số bằng nhau nghĩa là các giá trị của chúng luôn luôn bằng nhau. Nói cách khác:

$$\forall t : \frac{df}{dt}(t) = af(t)$$

Đây là một phương trình vi phân **thường** (PVT) vì tất cả các đạo hàm đều được lấy theo cùng một biến. Nếu như phương trình liên hệ các đạo hàm theo nhiều biến khác nhau (đạo hàm từng phần), thì ta sẽ có phương trình đạo hàm **riêng**.

Phương trình này là **bậc nhất** vì nó chỉ có chứa những đạo hàm cấp một. Nếu có mặt đạo hàm bậc hai, phương trình sẽ là bậc hai, và cứ như vậy.

Phương trình này là **tuyến tính** vì mỗi số hạng chứa t, f hoặc df/dt đều chỉ với bậc lũy thừa bằng một; nếu bất kì số hạng nào có chứa tích các đại lượng trên hoặc các lũy thừa của t, f và df/dt thì phương trình sẽ là phi tuyến.

Các PVT bậc nhất, tuyến tính đều có thể giải được theo cách giải tích; nghĩa là ta có thể biểu diễn nghiệm dưới dạng một hàm số của t. Riêng PVT này có vô số nghiệm, nhưng tất cả nghiệm đó đều có chung dạng sau:

$$f(t) = be^{at}$$

Với giá trị b bất kì, hàm số này đều thỏa mãn PVT. Nếu bạn không tin điều này, hãy tự lấy đạo hàm và kiểm tra.

Nếu ta biết số vi khuẩn tại một thời điểm nhất định thì ta có thể dùng thông tin nói trên để xác định trong số vô hạn các nghiệm, đâu là nghiệm duy nhất thỏa mãn điều kiện trên để mô tả diễn biến của số vi khuẩn thực tế theo thời gian.

Chẳng hạn, nếu đã biết rằng $f(0) = 5$ tỷ tế bào thì ta có thể viết:

$$f(0) = 5 = be^{a0}$$

và giải tìm ra b bằng 5. Từ đó ta tìm được hàm mong muốn:

$$f(t) = 5e^{at}$$

Thông tin thêm nói trên, vốn quyết định giá trị của b, được gọi là **điều kiện ban đầu** (cho dù không phải lúc nào nó cũng được chỉ định tại $t = 0$).

Không may là phần lớn các hệ vật lý thú vị đều được mô tả bởi các phương trình vi phân phi tuyến, mà đa số đều không thể giải được theo cách giải tích. Một cách khác là giải bằng phương pháp số.

8.2 Phương pháp Euler

Phương pháp số đơn giản nhất để giải PVT là phương pháp Euler. Sau đây là một bài kiểm tra nhỏ để xem bạn có thông minh được như Euler không. Giả sử như ở thời điểm t bạn đo được số vi khuẩn, y, và tốc độ tăng, r. Theo bạn thì số vi khuẩn sẽ là bao nhiêu sau một thời gian Δt trôi qua?

Nếu bạn nói rằng $y + r\Delta t$ thì xin chúc mừng! Bạn vừa phát minh ra phương pháp Euler (nhưng bạn vẫn chưa giỏi bằng Euler).

Cách ước tính này dựa trên giả sử rằng r là hằng số, nhưng nhình chung thì không phải vậy, do đó ta chỉ trông đợi là ước lượng sẽ tốt nếu r thay đổi từ từ và Δt nhỏ.

Nhưng hãy (tạm) giả sử rằng PVT đang xét có thể được viết sao cho

$$\frac{df}{dt}(t) = g(t, y)$$

trong đó g là một hàm nào đó chiếu từ (t, y) đến r; nghĩa là, cho trước thời điểm và số vi khuẩn, hàm này sẽ tính tốc độ thay đổi. Sau đó ta có thể tiến từ một thời điểm đến thời điểm tiếp theo bằng các phương trình sau đây:

$$T_{n+1} = T_n + \Delta t \tag{8.1}$$
$$F_{n+1} = F_n + g(t, y)\ \Delta t \tag{8.2}$$

Ở đây $\{T_i\}$ là một dãy các thời điểm mà tại đó ta cần tính các giá trị của f, còn $\{F_i\}$ là một dãy các giá trị ước tính được. Với mỗi chỉ số i, F_i là một ước lượng của $f(T_i)$. Khoảng Δt được gọi là **bước thời gian**.

Giả sử rằng ta khởi đầu tại $t = 0$ và có một điều kiện ban đầu $f(0) = y_0$ (trong đó y_0 là một giá trị cụ thể đã biết); đặt $T_1 = 0$ và $F_1 = y_0$, sau đó dùng các PT 8.1 và 8.2 để tính các giá trị của T_i và F_i đến khi T_i nhận giá trị của t mà ta quan tâm.

Nếu tốc độ tăng thay đổi không quá nhanh và bước thời gian không quá dài thì phương pháp Euler cũng đủ chính xác với nhiều mục đích tính toán. Một cách kiểm tra là chạy nó lần đầu với bước thời gian Δt và lần sau với bước thời gian $\Delta t/2$. Nếu các kết quả như nhau, thì có lẽ kết quả là đúng; nếu không, hãy thử rút ngắn bước thời gian một lần nữa.

Phương pháp Euler là **bậc nhất**, nghĩa là mỗi lần bạn chia đôi bước tính, bạn trông đợi sai số của giá trị ước tính sẽ giảm bớt một nửa. Với một phương pháp bậc hai, bạn trông đợi sai số giảm đi 4 lần; với phương pháp bậc ba giảm 8 lần, v.v. Cái giá phải trả cho phương pháp bậc cao là chúng phải lượng giá g nhiều lần hơn trong cùng một bước thời gian.

8.3 Lưu ý thêm về cách viết

Có rất nhiều kí hiệu toán học trong chương này, vì vậy tôi sẽ tạm dừng ở đây để ôn lại những gì chúng ta đã học được đến giờ. Dưới đây là các biến số, ý nghĩa, và kiểu của chúng:

Tên	Ý nghĩa	Kiểu
t	thời gian	biến vô hướng
Δt	bước thời gian	hằng vô hướng
y	số cá thể	biến vô hướng
r	tốc độ thay đổi	biến vô hướng
f	Hàm chưa biết được chỉ định dưới hình thức ẩn trong PVT.	hàm $t \to y$
df/dt	Đạo hàm bậc nhất theo thời gian của f	hàm $t \to r$
g	"Hàm tốc độ," được suy ra từ PVT, để tính tốc độ thay đổi với giá trị bất kì của t, y.	hàm $t, y \to r$
T	một loạt các thời điểm, t, tại đó ta ước tính $f(t)$	dãy
F	một loạt các giá trị ước tính của $f(t)$	dãy

Như vậy f là một hàm để tính số cá thể (vi trùng), có dạng một hàm theo thời gian, $f(t)$ là giá trị của hàm được tính ở một thời điểm nhất định, và nếu ta gán $f(t)$ cho một biến, ta thường gọi biến đó là y.

Tương tự, g là một "hàm tốc độ" để tính tốc độ thay đổi dưới hình thức một hàm phụ thuộc vào thời gian và số cá thể. Nếu ta gán $g(t, y)$ cho một biến, ta thường gọi nó là r.

df/dt là đạo hàm bậc nhất của f, nó chiếu từ t đến một giá trị tốc độ. Nếu ta gán $df/dt(t)$ cho một biến, ta gọi nó là r.

Thường thì df/dt rất dễ bị lẫn với g, nhưng lưu ý rằng cả về kiểu chúng đã khác nhau rồi. g có tính khái quát hơn: nó có thể tính được tốc độ thay đổi của bất kì số cá thể (giả định) nào tại thời điểm bất kì; df/dt thì cụ thể hơn: đó là tốc độ thay đổi thực sự tại thời điểm t, khi biết số cá thể là $f(t)$.

8.4 `ode`

Một hạn chế của phương pháp Euler là bước thời gian không đổi qua các lượt lặp khác nhau. Nhưng có những phần của lời giải lại khó ước tính hơn phần khác; nếu bước thời gian đủ ngắn để giải được phần khó thì cũng bước thời gian này sẽ khiến việc tính toán trở nên quá nhiều đối với những phần dễ. Cách làm lý tưởng nhất là điều chỉnh bước thời gian trong khi giải. Các phương pháp như vậy được gọi là **thích ứng**, và một trong những phương pháp thích ứng tốt nhất là phương pháp Fehlberg dùng cặp công thức Runge-Kutta. Bạn chưa cần biết chi tiết về công thức

này, vì những người phát triển Scilab đã lấy nó từ thư viện ODEPACK và đưa vào một hàm có tên là `ode`. Chữ `ode` có nghĩa là "ordinary differential equation [solver];" ([chương trình giải] PVT).

Để dùng `ode`, bạn phải viết một hàm Scilab để ước tính g như một hàm của t và y.

Sau đây là một ví dụ minh họa. Giả sử rằng tốc độ tăng trưởng của chuột phụ thuộc vào số con chuột hiện thời và mức độ sẵn có của thức ăn, vốn thay đổi theo thời gian trong năm. Phương trình cơ bản sẽ có dạng

$$\frac{df}{dt}(t) = af(t)\left[1 + \sin(\omega t)\right]$$

trong đó t là thời gian tính theo ngày và $f(t)$ là số con chuột tại thời điểm t.

a và ω là các **tham số**. Tham số là một giá trị đặc trưng cho một khía cạnh vật lý của hệ được mô phỏng. Chẳng hạn, ở Bài tập 6.2 ta đã dùng các tham số `rho` và `r` để định lượng khối lượng riêng và bán kính của một con vịt. Các tham số thường không đổi, nhưng cũng có thể thay đổi theo thời gian trong một số mô hình.

Ở ví dụ này, a đặc trưng cho tốc độ sinh sản, còn ω là tần số của một hàm tuần hoàn để mô tả ảnh hưởng của mức độ cung cấp thức ăn đến sự sinh sản.

Phương trình này đặc trưng cho quan hệ giữa một hàm với đạo hàm của nó. Để ước tính được các giá trị của f bằng cách số trị, ta phải chuyển nó về một hàm tốc độ.

Bước đầu tiên là giới thiệu một biến, y, là một tên gọi khác của $f(t)$

$$\frac{df}{dt}(t) = ay\left[1 + \sin(\omega t)\right]$$

Phương trình này có nghĩa là nếu cho trước các giá trị t và y, ta có thể tính $df/dt(t)$, vốn là tốc độ thay đổi của f. Bước tiếp theo là biểu diễn phép tính đó đưới dạng một hàm gọi là g:

$$g(t, y) = ay\left[1 + \sin(\omega t)\right]$$

Cách viết này rất có ích vì ta có thể dùng hàm với phương pháp Euler hoặc `ode` để ước tính các giá trị của f. Tất cả những việc ta cần làm chỉ là viết một hàm Scilab để lượng giá g. Sau đây là mã lệnh ứng với các giá trị $a = 0.01$ và $\omega = 2\pi/365$ (một chu kì mỗi năm):

```
function res = rats(t, y)
    a = 0.01;
```

```
    omega = 2 * %pi / 365;
    res = a * y * (1 + sin(omega * t));
endfunction
```

Bạn có thể kiểm tra hàm này từ Console bằng cách gọi nó với các giá trị khác nhau của `t` và `y`; kết quả sẽ là tốc độ thay đổi (đơn vị là số chuột mỗi ngày):

```
--> r = rats(0, 2)

r = 0.02
```

Như vậy nếu có 2 con chuột vào ngày 1/1, ta sẽ trông đợi chúng đẻ thêm với một tốc độ sao cho cứ 2 con chuột được sinh ra trong vòng 100 ngày. Nhưng đến tháng 4, tốc độ này đã tăng gấp đôi:

```
--> r = rats(120, 2)

r = 0.0376002
```

Vì tốc độ tăng thì luôn thay đổi, nên không dễ dự đoán số con chuột trong tương lai; song đó chính là việc mà `ode` đảm nhiệm. Sau đây là cách dùng nó:

```
--> y = ode(2, 0, 365, rats)
y = 76.949344
```

Đối số đầu tiên là giá trị đầu của hàm, $f(0) = 2$. Đối số thứ hai là thời giá trị đầu của t. Đối số thứ ba là giá trị t mà tại đó cần tính giá trị hàm, ở đây là 1 năm, hay 365 ngày. Đối số thứ tư là hàm g.

Nếu thay đối số thứ ba từ một giá trị vô hướng (365) bằng một véc-tơ thì `ode` sẽ trả lại kết quả là một véc-tơ tất cả các giá trị của hàm f tại những thời điểm tương ứng các phần tử trong véc-tơ đó.

```
--> T = 1:365;
--> Y = ode(2, 0, T, rats);
```

Giá trị được lưu lại vào véc-tơ `Y`. Như vậy `Y(1)` sẽ ứng với thời điểm là 1 (ngày).

Hãy vẽ đồ thị của `Y` theo `T`:

```
--> plot(T, Y, 'bo-')
```

Để biết được số chuột lúc cuối năm, bạn có thể hiển thị phần tử cuối của mỗi véc-tơ:

```
--> [T($), Y($)]

ans = 365.   76.949311
```

`$` là một từ đặc biệt trong Scilab; khi xuất hiện ở vị trí một chỉ số, nó có nghĩa là "chỉ số của phần tử cuối cùng." Bạn có thể dùng nó trong một biểu thức, vì vậy `Y($-1)` là phần tử áp chót của `Y`.

Số con chuột lúc cuối năm sẽ thay đổi bao nhiêu nếu bạn tăng gấp đôi số chuột ban đầu? Nó sẽ thay đổi bao nhiêu nếu bạn tính đến thời gian sau hai năm? Và nếu bạn tăng a gấp đôi?

8.5 Giải tích hay số trị?

Khi bạn giải PVT theo cách giải tích, kết quả là một hàm, f, cho phép ta tính số cá thể, $f(t)$, với bất kì giá trị nào của t. Khi bạn giải PVT theo cách số trị, bạn nhận được hai véc-tơ. Bạn có thể hình dung hai véc-tơ này như xấp xỉ của hàm f có tính liên tục: "rời rạc" là bởi vì nó chỉ xác định với những giá trị cụ thể của t, và "xấp xỉ" là vì mỗi giá trị F_i chỉ là một ước đoán cho giá trị thật $f(t)$.

Trên đây là những hạn chế của nghiệm số trị. Còn ưu điểm chính là bạn có thể tính nghiệm số trị cho cả những PVT mà không có nghiệm giải tích, vốn chiếm phần lớn các PVT phi tuyến.

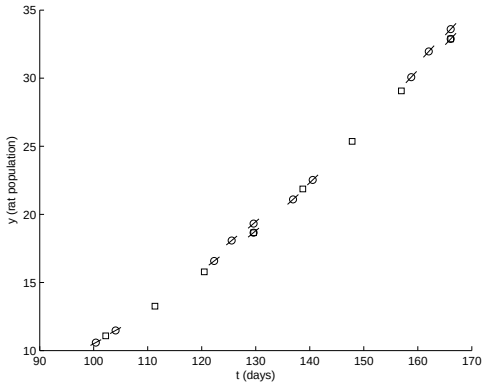
Nếu bạn còn tò mò về cơ chế hoạt động của `ode`, bạn có thể sửa `rats` để hiển thị các điểm, (t, y), tại đó `ode` được dùng để tính g. Sau đây là một phiên bản đơn giản:

```
function res = rats(t, y)
    plot(t, y, 'bo')
    a = 0.01;
    omega = 2 * %pi / 365;
    res = a * y * (1 + sin(omega * t));
endfunction
```

Mỗi lần `rats` được gọi, nó chấm một điểm số liệu.

```
--> clf
--> Y = ode(2, 0, T, rats);
```

Hình vẽ này cho thấy một phần của kết quả đầu ra, được phóng to trong khoảng những ngày từ 100 đến 170:



Các vòng tròn cho thấy những điểm tại đó `ode` gọi đến `rats`. Các đường thẳng đi qua vòng tròn cho thấy độ dốc (tốc độ thay đổi) tính được ở mỗi điểm. Hình chữ nhật cho thấy các vị trí được ước tính (T_i, F_i). Lưu ý rằng `ode` thường lượng giá g vài lần trong mỗi lần ước tính. Bằng cách này, không những giá trị ước tính được cải thiện mà còn phát hiện được những chỗ mà sai số đang tăng, tại đó cần rút ngắn bước thời gian (hoặc ngược lại).

8.6 Điều trục trặc gì có thể xảy ra?

Cần nhớ rằng hàm bạn viết ra sẽ được gọi bởi `ode`, nghĩa là nó phải có dấu của hàm mà `ode` trông đợi: cụ thể là nhận vào hai biến, `t` và `y`, theo đúng thứ tự đó, và trả lại một biến đầu ra, `r`.

Nếu bạn làm việc với một hàm tốc độ kiểu như:

$$g(t, y) = ay$$

Thì bạn có thể đã muốn viết như sau:

```
function res = rate_func(y)          // SAI
    a = 0.1
    res = a * y
endfunction
```

Nhưng cách làm này sai. Tại sao? Vì khi `ode` gọi đến `rate_func`, nó cấp cho hai đối số. Nếu bạn chỉ lấy có một biến đầu vào, bạn sẽ nhận được lỗi. Vì vậy bạn phải viết một hàm nhận cả `t` làm biến đầu vào, ngay cả khi bạn không dùng đến nó.

```
function res = rate_func(t, y)      // ĐÚNG
    a = 0.1
    res = a * y
endfunction
```

Một lỗi thường gặp khác là việc viết một hàm mà không gán kết quả cho một biến đầu ra. Nếu bạn viết lệnh kiểu như:

```
function res = rats(t, y)
    a = 0.01;
    omega = 2 * %pi / 365;
    r = a * y * (1 + sin(omega * t))    // SAI
endfunction
```

rồi gọi nó từ `ode`, bạn sẽ nhận được lỗi:

```
-->Y = ode(2, 0, T, rats);
 !--error 4
Undefined variable: res
at line       6 of function rats called by :
Y = ode(2, 0, T, rats);
```

Lỗi thuộc về hàm `rats()`, ở đó `res` chưa được xác định. Trước khi kết thúc hàm, nhất thiết phải gán giá trị cho `res`.

8.7 Độ cứng

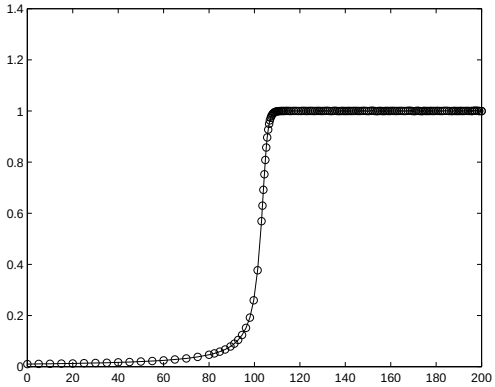
Còn một vấn đề khác mà bạn có thể đối mặt, nhưng nếu nó làm bạn cảm thấy khỏe hơn, thì có lẽ không phải lỗi của bạn: bài toán bạn đang giải có thể thuộc loại **cứng**[*].

Tôi sẽ không trình bày về khía cạnh kĩ thuật liên quan đến độ cứng ở đây, ngoại trừ phải nói rằng với một số bài toán (ở những khoảng thời gian nào đó và điều kiện ban đầu nào đó) thì bước thời gian cần thiết để kiểm soát sai số là rất ngắn; điều đó nghĩa là thời gian tính toán sẽ rất lâu. Sau đây là một ví dụ:

$$\frac{df}{dt} = f^2 - f^3$$

[*]Nội dung trình bày sau đây một phần được dựa trên bài báo đăng bởi Mathworks tại `http://www.mathworks.com/company/newsletters/news_notes/clevescorner/may03_cleve.html`

Nếu bạn giải PVT với điều kiện ban đầu $f(0) = \delta$ trên khoảng từ 0 đến $2/\delta$, với $\delta = 0.01$, bạn sẽ thấy kết quả sau:



Sau bước chuyển dần từ 0 tới 1, bước thời gian trở nên rất ngắn và việc tính toán rất chậm. Với những giá trị nhỏ hơn của δ, tình hình còn tệ hơn.

Scilab có thể dự đoán thông minh tình huống này và hàm `ode` tự chuyển sang chế độ giải phương trình vi phân "cứng". Ngoài ra, bạn có thể chỉ định dùng phương pháp giải chuyên áp dụng với hệ cứng bằng cách đưa `"stifff"` lên vị trí đầu trong danh sách tham biến. Từ dấu nhắc lệnh, hãy gõ vào:`help ode` để biết thêm thông tin chi tiết.

Exercise 8.1 *Bài tập này để thấy một trường hợp mà chế độ giải* `"stiff"` *phát huy tác dụng. Hãy viết một hàm tốc độ cho PVT này rồi dùng* `ode` *với chế độ* `"rkf"` *(sơ đồ Runge-Kutta Fehlberg bậc 4 và 5) để giải với điều kiện ban đầu và khoảng tính cho trước. Bắt đầu với $\delta = 0.1$ và giảm dần nó theo 10 lần. Nếu quá mệt mỏi chờ đợi phép tính thực hiện chưa xong, bạn có thể ấn nút Stop ở cửa sổ Figure hoặc ấn Control-C từ trong Console.*

Bây giờ hãy thay thế `ode("rkf", ...` *bằng* `ode("stiff", ...` *và thử lại!*

8.8 Thuật ngữ

phương trình vi phân: Phương trình liên hệ các đạo hàm của một hàm số chưa biết.

phương trình vi phân thường: Phương trình vi phân thường mà tất cả đạo hàm được lấy theo cùng một biến.

phương trình vi phân riêng: Phương trình vi phân có bao gồm các đạo hàm lấy theo nhiều biến.

bậc nhất (PVT): Phương trình vi phân chỉ chứa các đạo hàm bậc nhất.

tuyến tính: Phương trình vi phân không có chứa tích hoặc lũy thừa của hàm cùng các đạo hàm của nó.

bước thời gian: Khoảng thời gian giữa hai lần ước tính kế tiếp trong lời giải số trị của một phương trình vi phân.

bậc nhất (phương pháp số): Phương pháp theo đó sai số xấp xỉ được trông đợi sẽ giảm đi một nửa khi bước tính được rút ngắn còn một nửa.

thích ứng: Phương thức có thể điều chỉnh được bước thời gian để kiểm soát sai số.

độ cứng: Đặc trưng của PVT khiến cho một số hàm giải PVT chạy rất chậm (hoặc tính ra kết quả không chính xác). Một số hàm giải PVT như `ode` với chế độ `"stiff"` trong Scilab được thiết kế để giải các bài toán có độ cứng.

tham số: Giá trị xuất hiện trong mô hình nhằm định lượng một khía cạnh vật lý nào đó của hệ được mô phỏng.

8.9 Bài tập

Exercise 8.2 *Giả sử rằng bạn có tách chứa 8 ounce cà-phê nóng 90 °C và 1 ounce kem ở nhiệt độ phòng, 20 °C. Bạn biết qua một kinh nghiệm cay đắng là chỉ uống được cà-phê nóng nhất ở mức 60 °C.*

Lưu ý rằng bạn cho kem vào tách cà-phê, và muốn uống càng sớm càng tốt, thì liệu có nên cho kem vào ngay không, hay đợi một lúc? Và nếu cần đợi, thì nên đợi bao lâu?

Để trả lời câu hỏi này, bạn phải mô phỏng quá trình nguội lạnh của nước nóng trong không khí. Cà-phê nóng truyền nhiệt ra ngoài môi trường bằng quá trình đối lưu, bức xạ, và bốc hơi. Việc định lượng từng hiệu ứng nói trên sẽ rất khó và cũng không cần thiết cho việc giải bài toán này.

Để đơn giản hóa, ta có thể dùng Định luật Newton về làm lạnh [†]:

[†] `http://en.wikipedia.org/wiki/Heat_conduction`

$$\frac{df}{dt} = -r(f - e)$$

trong đó f là nhiệt độ cà-phê, được viết theo hàm của thời gian còn df/dt là đạo hàm theo thời gian của nó; e là nhiệt độ môi trường, bằng hằng số trong trường hợp này, và r là một thông số (cũng không đổi) để đặc trưng cho tốc độ truyền nhiệt.

Sẽ dễ ước tính r cho một tách cà phê cụ thể bằng cách đo nhiệt độ vài lần theo thời gian. Ta hãy giả sử rằng điều đó đã được thực hiện và r đã tính được bằng 0.001 với đơn vị là nghịch đảo của giây, $1/s$.

- *Hãy dùng kí hiệu toán học để biểu diễn hàm tốc độ, g, dưới dạng hàm của y, trong đó y là nhiệt độ của cà-phê tại thời điểm bất kì.*

- *Tạo ra tập tin M có tên* `coffee` *và viết một hàm có tên* `coffee` *nhận vào hai biến và không trả lại giá trị đầu ra. Hãy đặt lệnh đơn giản như* `x=5` *vào trong phần thân hàm rồi gọi* `coffee()` *từ Console.*

- *Bổ sung một hàm có tên* `rate_func` *nhận vào* `t` *và* `y` *rồi tính $g(t, y)$. Lưu ý rằng trong trường hợp này g thực ra không phụ thuộc vào t; song dù sao hàm cần viết vẫn phải nhận vào t làm đối số đầu tiên để có thể dùng được* `ode`*.*

  Hãy thử hàm vừa viết được bằng cách thêm một dòng lệnh như `rate_func(0,90)` *vào trong* `coffee`*, rồi gọi* `coffee` *từ Console.*

- *Một khi* `rate_func(0,90)` *chạy được, hãy sửa* `coffee` *để dùng* `ode` *tính nhiệt độ của cà-phê không (chưa tính đến kem) trong vòng 60 phút. Hãy chắc rằng cà-phê ban đầu nguội nhanh, rồi chậm dần, cuối cùng thì tiến gần về và đạt (xấp xỉ) nhiệt độ phòng sau khoảng 1 giờ.*

- *Hãy viết một hàm có tên* `mix_func` *để tính nhiệt độ nhiệt độ cuối cùng của hỗn hợp hai chất lỏng. Hàm phải nhận vào các tham số là thể tích và nhiệt độ của từng loại chất lỏng.*

  Nói chung nhiệt độ của một hỗn hợp thì phụ thuộc vào nhiệt dung riêng của hai chất[‡]*. Nhưng nếu làm đơn giản bằng cách giả sử rằng cà-phê và kem có cùng khối lượng riêng và nhiệt dung riêng thì nhiệt độ cuối cùng là*

[‡] `http://en.wikipedia.org/wiki/Heat_capacity`

$(v_1y_1 + v_2y_2)/(v_1 + v_2)$, *trong đó* v_1 *và* v_2 *là các thể tích của hai loại chất lỏng, và* y_1 *và* y_2 *là nhiệt độ của chúng.*

Hãy thêm mã lệnh vào trong `coffee` *để thử nghiệm hàm* `mix_func` *vừa viết.*

- *Hãy dùng* `mix_func` *và* `ode` *để tính thời điểm bắt đầu lúc uống được cà-phê, trong trường hợp rót kem vào ngay.*
- *Hãy thay đổi* `coffee` *để nó nhận vào biến* t *(có vai trò quyết định sau bao nhiêu giây cà-phê được để nguội trước khi rót kem vào), và trả lại nhiệt độ của tách cà-phê sau khi trộn.*
- *Hãy dùng* `fzero` *tính thời gian* t *cần để nhiệt độ của tách cà-phê sau khi trộn hạ xuống còn 60 °C.*
- *Những kết quả trên sẽ cho bạn biết gì về đáp số của bài toán ban đầu? Có phải đó là đáp số như mong muốn không? Những giả thiết nhằm giản hóa nào đã chi phối lời giải này? Theo bạn thì giả thiết nào có ảnh hưởng đáng kể nhất? Bạn có nghĩ rằng ảnh hưởng này đủ mạnh để tác độ đến kết quả không? Xét toàn diện thì bạn tin tưởng đến mức nào vào việc mô hình này sẽ cho ra đáp số cuối cùng? Bạn có thể làm gì để cải thiện mô hình?*

Chương 9

Hệ các PVT

9.1 Ma trận

Ma trận là dạng hai chiều của một véc-tơ. Cũng như véc-tơ, ma trận gồm có các phần tử được phân biệt bởi chỉ số. Sự khác biệt với véc-tơ là ở chỗ các phần tử ma trận được xếp theo hàng và cột, vì vậy cần có *hai* chỉ số để xác định được một phần tử.

Một trong những cách tạo ra ma trận là dùng hàm `testmatrix` với chế độ `'magic'`, vốn trả lại một ma trận kì ảo với kích cỡ cho trước:

```
--> M = testmatrix('magic',3)

M =  8.     1.     6.
     3.     5.     7.
     4.     9.     2.
```

Nếu không biết kích cỡ của ma trận, bạn có thể nhìn vào cửa sổ Variable Browser, hoặc gõ `whos` để hiển thị thông tin:

```
--> whos
Name                    Type          Size          Bytes

M                       constant      3 by 3        88
```

Hoặc dùng hàm `size`, lần này sẽ trả về một véc-tơ:

```
--> V = size(M)

V = 3.     3.
```

Phần tử đầu tiên của véc-tơ là số hàng, và phần tử thứ hai là số cột.

Để đọc một phần tử của ma trận, bạn cần chỉ ra số thứ tự hàng và cột:

```
--> M(1,2)

ans = 1.

--> M(2,1)

ans = 3.
```

Khi bắt đầu làm quen với ma trận, bạn cần nhớ được chỉ số nào đi trước, hàng hay cột. Tôi thì thường nói nhẩm "hàng, cột", như một câu thần chú. Bạn cũng có thể nhớ kiểu "xuống dưới, sang ngang," hoặc chữ tắt HC.

Một cách khách để tạo ra ma trận là viết các phần tử giữa cặp ngoặc vuông, cùng dấu chấm phẩy giữa các hàng:

```
--> D = [1,2,3 ; 4,5,6]

D =  1.     2.     3.
     4.     5.     6.

--> size(D)

ans = 2.     3.
```

9.2 Véc-tơ hàng và cột

Mặc dù có thể dễ nhớ hơn theo các loại số vô hướng, véc-tơ và ma trận, song theo quan điểm của Scilab, mọi thứ đều là ma trận. Một số vô hướng chính là một ma trận có một hàng và một cột:

```
--> x = 5;
--> size(x)

ans = 1.     1.
```

Và véc-tơ là ma trận có một hàng:

```
--> R = 1:5;
--> size(R)

ans = 1.     5.
```

Thực ra có hai loại véc-tơ. Loại véc-tơ ta đã gặp cho đến giờ là **véc-tơ hàng**, vì các phần tử được sắp trên một hàng; còn loại kia là **véc-tơ cột**, vì các phần tử xếp theo một cột.

Một cách tạo ra véc-tơ cột là lập một ma trận chỉ với một phần tử trên mỗi hàng:

```
--> C = [1;2;3]

C =

     1.
     2.
     3.

--> size(C)

ans = 3.     1.
```

Sự khác biệt giữa véc-tơ hàng và cột là quan trọng trong môn đại số tuyến tính, nhưng với đa số các phép tính với véc-tơ thì điều đó chẳng quan trọng. Khi bạn đánh chỉ số cho véc-tơ, bạn không cần biết đó là loại véc-tơ gì:

```
--> R(2)

ans = 2.

--> C(2)

ans = 2.
```

9.3 Toán tử chuyển vị

Toán tử chuyển vị, vốn trông giống như dấu nháy đơn, được dùng để tính **chuyển vị** của ma trận, vốn là một ma trận mới có tất cả những phần tử của ma trận ban đầu, nhưng với mỗi hàng được dựng thành một cột (hoặc bạn có thể nghĩ ngược lại).

Ở ví dụ sau:

```
--> D = [1,2,3 ; 4,5,6]

D =  1     2     3
     4     5     6
```

`D` có hai hàng, vì vậy ma trận chuyển vị của nó có hai cột:

```
--> Dt = D'

Dt = 1.    4.
     2.    5.
     3.    6.
```

Exercise 9.1 *Toán tử chuyển vị có tác động gì đến véc-tơ hàng, véc-tơ cột, và số vô hướng?*

9.4 Lotka-Voltera

Mô hình Lotka-Voltera mô tả sự tương tác giữa hai loài, vật săn và con mồi, trong cùng một hệ sinh thái. Một ví dụ là hai loài thỏ và cáo.

Mô hình được diễn tả bởi hệ phương trình vi phân sau:

$$R_t = aR - bRF$$
$$F_t = ebRF - cF$$

trong đó

- R là số con thỏ,
- F là số con cáo,

- a là tốc độ tăng trưởng của thỏ khi không có loài săn đuổi,
- c là tốc độ chết của cáo khi không có mồi,
- b là tốc độ chết của thỏ ứng với mỗi tương tác với cáo,
- e là hiệu suất biến đổi số thỏ bị ăn thịt thành cáo.

Thoạt nhìn, bạn có thể nghĩ rằng có thể giải ngay các phương trình bằng cách gọi `ode` một lần để giải ra R như một hàm theo thời gian và một lần khác để giải F. Vấn đề là mỗi phương trình đều chứa đến cả hai biến, cũng vì lý do này mà ta gọi là **hệ phương trình** chứ không phải là một loạt các phương trình riêng lẻ. Để giải một hệ, bạn cần phải giải tất cả phương trình cùng lúc.

Thật may là `ode` có thể giải được hệ phương trình. Sự khác biệt ở đây là điều kiện ban đầu có dạng một véc-tơ chứa những giá trị ban đầu là $R(0)$ và $F(0)$, và kết quả đầu ra là ma trận có chứa một cột dành cho R và một cột cho F.

Sau đây là các hàm tốc độ được viết với các tham số $a = 0.1$, $b = 0.01$, $c = 0.1$ and $e = 0.2$:

```
function res = lotka(t, V)
    // tháo gỡ các phần tử của V
    r = V(1);
    f = V(2);

    // đặt các tham số
    a = 0.1;
    b = 0.01;
    c = 0.1;
    e = 0.2;

    // tính các đạo hàm
    drdt = a*r - b*r*f;
    dfdt = e*b*r*f - c*f;

    // xếp các giá trị đạo hàm vào véc-tơ
    res = [drdt; dfdt];
endfunction
```

Như thường lệ, biến đầu vào thứ nhất là thời gian. Biến đầu vào thứ hai là một véc-tơ `V` gồm hai phần tử, $R(t)$ và $F(t)$. Tôi viết nó bằng chữ in để dễ gợi nhớ

rằng đó là véc-tơ. Phần thân hàm gồm bốn **đoạn**, mỗi đoạn đều được chú thích kèm theo.

Đoạn thứ nhất nhằm **tháo gỡ** véc-tơ bằng cách sao chép các phần tử ra các biến vô hướng. Điều này không nhất thiết phải làm, nhưng việc đặt tên cho từng giá trị giúp tôi nhớ được ý nghĩa của chúng. Nó cũng giúp ích ở đoạn thứ ba, khi ta tính đạo hàm, thể hiện ở cách viết giống như biểu thức toán học ban đầu, góp phần tránh được lỗi.

Đoạn thứ hai đặt các tham số miêu tả các tốc độ sinh sản của thỏ và cáo, cùng với các đặc trưng tương tác giữa chúng. Nếu ta đang nghiên cứu một hệ thực sự, thì những giá trị trên sẽ phải được rút ra từ quá trình quan sát đời sống của loài vật; nhưng ở ví dụ này tôi chỉ chọn các giá trị sao cho thu được kết quả thú vị.

Đoạn cuối cùng **gói** các đạo hàm tính được trở lại vào véc-tơ. Khi `ode` gọi hàm này, nó cung cấp đầu vào là một véc-tơ và trông đợi đầu ra cũng là một véc-tơ.

Bạn đọc tinh mắt có thể nhận ra có điểm khác biệt ở dòng:

```
  res = [drdt; dfdt];
```

Dấu chấm phẩy ngăn cách giữa hai phần tử của véc-tơ này không có gì sai. Nó rất cần trong trường hợp này vì `ode` yêu cầu kết quả của hàm này phải là một véc-tơ cột.

Bây giờ ta có thể chạy `ode` như sau:

```
M = ode([100;10], 0, 1:365, lotka);
```

Như thường lệ, thông số thứ nhất là một chuôi hàm, thứ hai là khoảng thời gian, và thứ ba là điều kiện ban đầu. Ở đây, điều kiện ban đầu là một véc-tơ: phần tử thứ nhất là số con thỏ lúc $t = 0$, phần tử thứ hai là số con cáo.

Thứ tự của hai phần tử này (số thỏ và số cáo) là tùy vào bạn, nhưng phải thống nhất. Nghĩa là điều kiện ban đầu được cung cấp khi gọi đến `ode` phải có cùng thứ tự như trong `lotka`, nơi bạn gỡ véc-tơ đầu vào và gói lại véc-tơ đầu ra. Scilab không hiểu ý nghĩa các giá trị này; tất cả sự theo dõi đều là trách nhiệm của người lập trình.

9.5 Điều gì có thể gây trục trặc?

Một lỗi khác có thể xảy ra khác là việc đảo lộn thứ tự của các phần tử trong số các điều kiện ban đầu, hay là các véc-tơ bên trong `lotka`. Một lần nữa, Scilab không biết rằng các phần tử có ý nghĩa gì, cho nên nó không thể phát hiện ra lỗi kiểu này, mà chỉ cho kết quả sai.

9.6 Ma trận kết quả

Ở trên ta đã gán kết quả tính `ode` vào biến `M`. Để cho tiện, ta hãy chuyển vị `M`:

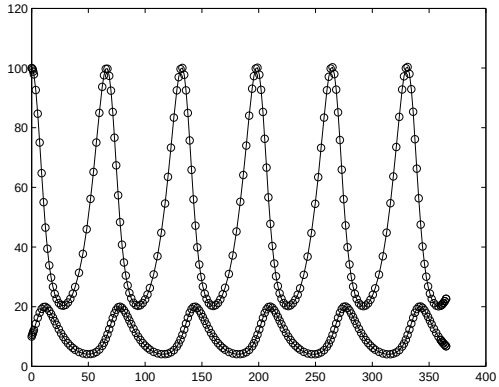
```
--> M = M';
--> size(M)

ans = 365.    2.
```

Bây giờ ta có ma trận mà một cột chứa một biến (trong trường hợp này là R và F) và từng hàng cho mỗi giá trị thời gian.

Cấu trúc này—một cột cho mỗi biến—là cách thông thường để dùng ma trận. `plot` hiểu được cấu trúc này, nên khi bạn viết:

```
--> T = 1:365;
--> plot(T, M)
```

Scilab sẽ hiểu rằng nó cần vẽ đồ thị cho mỗi cột của `M` theo `T`. Kết quả như sau:



Trục x là thời gian tính theo ngày; trục y là số cá thể. Đường cong phía trên chỉ số thỏ; đường phía dưới chỉ số cáo. Kết quả này là một trong số những dạng mẫu mà hệ này có thể rơi vào, tùy theo các điều kiện ban đầu và các tham số. Bạn hãy tự thử nghiệm với các giá trị khác nhau, như một bài tập cho riêng mình.

Bạn có thể sao chép các hàng của `M` ra các biến khác như sau:

```
--> R = M(:, 1);
--> F = M(:, 2);
```

Ở đây, dấu hai chấm biểu thị cho khoảng từ 1 đến `end`, vì vậy `M(:, 1)` nghĩa là "tất cả các hàng thuộc cột 1" và `M(:, 2)` nghĩa là "tất cả các hàng thuộc cột 2."

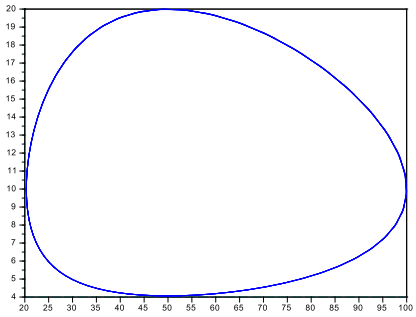
```
--> size(R)

ans = 365.      1.

--> size(F)

ans = 365.      1.
```

Như vậy `R` và `F` là các véc-tơ cột.

Nếu vẽ đồ thị các véc-tơ này theo nhau, kiểu như:

```
--> plot(R, F)
```

thì bạn sẽ thu được một **đồ thị pha** giống hình dưới đây:



Mỗi điểm trên đồ thị này biểu diễn một số thỏ nhất định (theo trục x) và một số cáo nhất định (theo trục y).

Vì chỉ có hai biến trong hệ này, mỗi điểm trên mặt phẳng biểu thị cho **trạng thái đầy đủ** của hệ.

Theo thời gian, trạng thái sẽ di chuyển quanh mặt phẳng: hình vẽ này cho thấy đường đi vạch nên bởi trạng thái trong khoảng thời gian tính toán. Đường đi này được gọi là **quỹ đạo**.

Vì biểu hiện của hệ thống này có tính chu kì nên quỹ đạo thu được là một đường cong khép kín.

Nếu có 3 biến trong hệ, ta sẽ cần 3 chiều không gian để biểu thị trạng thái của hệ, vì vậy quỹ đạo là một đường cong 3 chiều. Bạn có thể dùng `plot3` để vạch nên đường quỹ đạo trong không gian 3 chiều, nhưng khi có từ 4 biến trở lên thì bạn chẳng còn cách biểu thị trực quan nào nữa.

9.7 Thuật ngữ

véc-tơ hàng: Những ma trận chỉ có một hàng trong Scilab.

véc-tơ cột: Những ma trận chỉ có một cột.

chuyển vị: Phép tính để chuyển các hàng của ma trận thành cột (hoặc ngược lại).

hệ phương trình: Tập hợp các phương trình chứa một tập hợp các biến sao cho các phương trình liên hệ qua lại lẫn nhau.

đoạn: Một loạt các lệnh liên tiếp hình thành nên một phần của hàm, thường được kèm theo bởi một lời giải thích rõ ràng.

tháo gỡ: Sao chép các phần tử trong một véc-tơ vào một tập hợp các biến.

đóng gói: Sao chép các giá trị từ một tập hợp các biến vào một véc-tơ.

trạng thái: Nếu một hệ có thể được miêu tả bởi một tập hợp các biến thì giá trị của các biến đó được gọi là trạng thái của hệ.

đồ thị pha: Đồ thị cho thấy trạng thái của hệ như một điểm trên không gian các trạng thái hiện có.

quỹ đạo: Đường đi của đồ thị pha cho thấy trạng thái của một hệ thay đổi như thế nào theo thời gian.

9.8 Bài tập

Exercise 9.2 *Dựa vào các ví dụ ta đã thấy, bạn có thể nghĩ rằng tất cả PVT mô tả số cá thể đều có dạng hàm theo thời gian, nhưng thực ra không phải vậy.*

Theo Wikipedia[*]*, "điểm hấp dẫn Lorenz, được Edward Lorenz đề xuất vào năm 1963, là một hệ động lực tất định ba chiều phi tuyến được suy từ các phương trình giản hóa đặc trưng cho các cuộn đối lưu từ phương trình động lực khí quyển. Với một tập hợp nhất định các tham số, hệ thống sẽ có biểu hiện hỗn loạn và cho thấy cái mà ngày nay ta gọi là điểm hấp dẫn dị thường..."*

[*] `http://en.wikipedia.org/wiki/Lorenz_attractor`

Hệ được mô tả bởi hệ phương trình vi phân sau:

$$x_t = \sigma(y - x) \tag{9.1}$$
$$y_t = x(r - z) - y \tag{9.2}$$
$$z_t = xy - bz \tag{9.3}$$

Các giá trị thông dụng của tham số gồm $\sigma = 10$, $b = 8/3$ *và* $r = 28$.

Hãy dùng `ode` *để ước tính một nghiệm của hệ phương trình này.*

1. *Bước đầu tiên là viết một hàm có tên là* `lorenz` *nhận* `t` *và* `V` *làm các biến đầu vào, trong đó các thành phần của* `V` *được hiểu là các giá trị hiện thời của* `x`, `y` *và* `z`. *Nó cần phải tính được các đạo hàm tương ứng và trả chúng về dưới dạng một véc-tơ cột.*

2. *Bước tiếp theo là thử nghiệm hàm của bạn bằng cách gọi nó từ dòng lệnh với các giá trị như* $t = 0$, $x = 1$, $y = 2$ *và* $z = 3$. *Một khi hàm chạy được, bạn cần biến nó thành một hàm lặng trước khi gọi* `ode`.

3. *Giả định rằng Bước 2 đã xong, bạn có thể dùng* `ode` *để ước tính nghiệm cho khoảng thời gian từ* $t_0 = 0$ *đến* $t_e = 30$ *với điều kiện ban đầu* $x = 1$, $y = 2$ *và* $z = 3$.

4. *Hãy dùng* `plot3` *để vẽ đồ thị quỹ đạo theo* x, y *và* z.

Chương 10

Các hệ bậc hai

10.1 Hàm lồng ghép

Trong Mục 7.1, ta đã thấy một ví dụ của tập tin M với hơn một hàm:

```
function res = duck()
    error = error_func(10)
end

function res = error_func(h)
    rho = 0.3;      // mật độ, tính bằng g / cm^3
    r = 10;         // bán kính, tính bằng cm
    res = ...
endfunction
```

Vì hàm thứ nhất kết thúc trước hàm thứ hai, chúng có cùng cấp thụt đầu dòng. Những hàm như vậy được gọi là **song song**, trái với **lồng ghép**. Một hàm lồng ghép được định nghĩa ở bên trong hàm khác, kiểu như:

```
function res = duck()
    error = error_func(10)

    function res = error_func(h)
        rho = 0.3;      // mật độ, tính bằng g / cm^3
        r = 10;         // bán kính, tính bằng cm
        res = ...
```

```
    end
endfunction
```

Hàm cấp cao nhất, `duck`, là **hàm ngoài** và `error_func` là **hàm trong**.

Các hàm lồng ghép rất có ích vì các biến của hàm ngoài có thể truy cập đến từ hàm trong. Điều này không thực hiện được với các hàm song song.

Ở ví dụ này, việc dùng hàm lồng ghép khiến cho ta có thể chuyển các tham số `rho` và `r` ra ngoài `error_func`.

```
function res = duck(rho)
    r = 10;
    error = error_func(10)

    function res = error_func(h)
        res = ...
    end
endfunction
```

Cả `rho` và `r` đều có thể truy cập đến từ `error_func`. Bằng cách biến `rho` trở thành đối số đầu vào, ta đã khiến việc thử nghiệm `duck` với các giá trị tham số khác nhau được dễ dàng hơn.

10.2 Chuyển động Newton

Định luật II Newton về chuyển động thường được viết như sau

$$F = ma$$

trong đó F là hợp lực tác dụng lên một vật, m là khối lượng của vật, còn a là gia tốc mà vật thu được. Chỉ trong trường hợp đơn giản khi vật chuyển động theo đường thẳng thì cả F và a đều là số vô hướng, nhưng nhìn chung chúng là véc-tơ.

Thậm chí khái quát hơn nữa, nếu F và a thay đổi theo thời gian thì chúng là một hàm và kết quả của việc lượng giá $F(t)$ là một véc-tơ mô tả hợp lực tại thời điểm t. Như vậy một cách viết tường minh hơn định luật Newton là

$$\forall t : \vec{F}(t) = m\vec{a}(t)$$

Thứ tự sắp xếp của phương trình này cho thấy nếu ta đã biết m và a thì có thể tính được lực. Điều này là đúng, nhưng ở phần lớn các mô phỏng vật lý thì ngược lại. Dựa trên mô hình vật lý, bạn biết F và m, rồi đi tính a.

Vậy nếu đã biết gia tốc, a, dưới dạng hàm theo thời gian, bằng cách nào có thể tìm được vị trí p của vật? À, ta biết rằng gia tốc là đạo hàm bậc hai của vị trí, vậy nên ta có thể viết một phương trình vi phân

$$p_{tt} = a$$

trong đó a và p là các hàm số theo thời gian và trả lại các véc-tơ, còn p_{tt} là đạo hàm bậc hai theo thời gian của p.

Vì phương trình này bao gồm một đạo hàm bậc hai nên nó được gọi là PVT bậc hai. `ode` không thể giải phương trình dưới dạng này, nhưng bằng cách đưa vào một biến mới, v, đại diện cho vận tốc thì ta có thể viết lại nó như một hệ các PVT bậc nhất.

$$\begin{aligned} p_t &= v \\ v_t &= a \end{aligned}$$

Phương trình đầu phát biểu rằng đạo hàm bậc nhất của p là v; phương trình thứ hai cho thấy đạo hàm của v là a.

10.3 Hiện tượng rơi tự do

Ta hãy bắt đầu bằng một ví dụ đơn giản, một vật rơi tự do trong chân không (ở đó không có sức cản của không khí). Gần mặt đất, gia tốc trọng trường là $g = -9.8$ m/s^2, trong đó dấu trừ biểu thị rằng trọng lực hướng xuống dưới.

Nếu vật rơi thẳng xuống (cùng theo hướng trọng lực) thì ta có thể mô tả vị trí của nó bằng một giá trị vô hướng là độ cao. Vì vậy, tạm thời bây giờ bài toán là một chiều.

Sau đây là hàm tốc độ mà ta có thể dùng với `ode` để giải bài toán này:

```
function res = freefall(t, X)
    p = X(1);       // thành phần thứ nhất là vị trí
    v = X(2);       // thành phần thứ hai là vận tốc

    dpdt = v;
    dvdt = acceleration(t, p, v);

    res = [dpdt; dvdt];    // đóng gói kết quả vào vector cột
```

```
endfunction

function res = acceleration(t, p, v)
    g = -9.8;       // gia tốc trọng lực tính bằng m/s^2
    res = g;
endfunction
```

Hàm thứ nhất là hàm tốc độ. Nó nhận vào các biến `t` và `X`, trong đó các phần tử của `X` được hiểu là vị trí và vận tốc. Giá trị trả về từ `freefall` là một véc-tơ (cột) chứa những đạo hàm của vị trí và vận tốc, tức lần lượt là vận tốc và gia tốc.

Việc tính p_t thật dễ dàng vì ta đã có vận tốc như một phần tử của `X`. Thứ duy nhất mà ta cần tính là gia tốc, và đó là việc mà hàm thứ hai đảm nhiệm.

`acceleration` làm nhiệm vụ tính gia tốc như một hàm của thời gian, vị trí và vận tốc. Ở ví dụ này, gia tốc tổng hợp là một hằng số, vì vậy ta không cần phải kể ra tất cả những thông tin trên, nhưng rồi sắp tới ta sẽ phải nhắc đến.

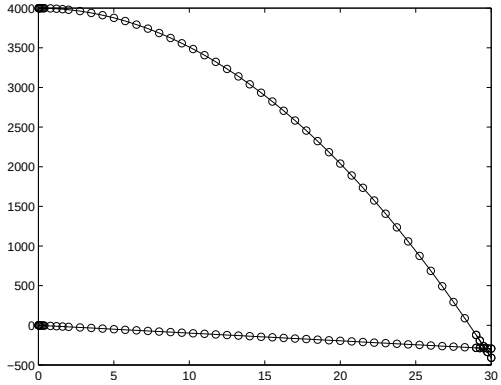
Sau đây là cách chạy `ode` với hàm tốc độ này. (Rút kinh nghiệm từ lần trước, sau khi tính `ode` ta lấy chuyển vị (`'`) của `RES` luôn:

```
--> T = 1:30;
--> RES = ode([4000; 0], 0, T, freefall)';
```

Như mọi khi, đối số thứ nhất là véc-tơ các giá trị ban đầu (trong trường hợp này, độ cao ban đầu là 4000 mét và vận tốc ban đầu là 0), thứ hai là thời điểm đầu, thứ ba là khoảng thời gian (30 giây). Vì vậy bạn có thể hình dung "vật" là một người nhảy dù vừa rời khỏi máy bay ở độ cao 4000 mét.

```
--> plot(T, RES)
```

Và biểu đồ kết quả sẽ như sau:



Trên hình vẽ, đường dưới cho thấy vận tốc bắt đầu tại 0 và hạ xuống một cách tuyến tính. Đường trên cho thấy vị trí bắt đầu tại 4000 m và hạ xuống theo hàm parabol (nhớ rằng đây là hàm parabol theo thời gian chứ không phải đường bay hình parabol).

Lưu ý rằng `ode` không biết vị trí mặt đất ở đâu, vì vậy người nhảy dù vẫn hạ độ cao từ không xuống các giá trị độ cao âm. Ta sẽ giải quyết vấn đề này sau.

10.4 Lực cản không khí

Để giúp cho việc mô phỏng được xác thực hơn, ta có thể thêm vào lực cản không khí. Với những vật lớn di chuyển nhanh trong không trung, lực cản không khí tác dụng lên vật sẽ tỷ lệ thuận với v^2:

$$F_{drag} = cv^2$$

Trong đó c là hằng số cản, phụ thuộc vào các yếu tố mật độ không khí, diện tích mặt cắt ngang của vật thể và đặc tính bề mặt của vật thể. Với mục đích minh họa của bài tập này, có thể lấy $c = 0.2$.

Để chuyển đổi từ lực sang gia tốc, ta cần phải biết khối lượng, vì vậy hãy coi rằng người nhảy dù (cùng với thiết bị đi theo người) nặng 75 kg.

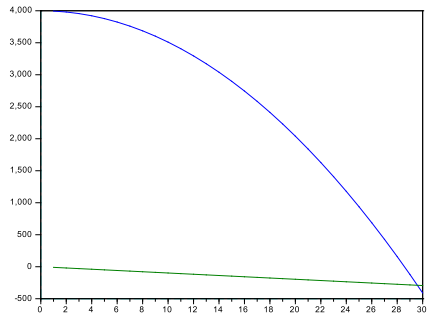
Sau đây là một phiên bản của `acceleration` có tính đến lực cản không khí (bạn không cần phải thay đổi gì trong `freefall`):

```
function res = acceleration(t, p, v)
    a_grav = -9.8;              // gia tốc trọng lực tính bằng m/s^2
    c = 0.2;                    // hằng số cản
    m = 75;                     // khối lượng tính bằng kg
    f_drag = c * v^2;           // lực cản tính bằng N
    a_drag = f_drag / m;        // gia tốc cản tính bằng m/s^2
    res = a_grav + a_drag;      // gia tốc tổng hợp
endfunction
```

Dấu của lực cản (và gia tốc) là dương nếu khi vật đang rơi, hướng của lực cản hướng lên trên. Gia tốc tổng hợp là tổng của gia tốc trọng lực và gia tốc cản. Hãy cẩn thận khi bạn làm việc với các lực và gia tốc; hãy chắc rằng bạn chỉ cộng lực với lực và gia tốc với gia tốc. Trong mã lệnh vừa viết, tôi dùng lời chú thích để tự nhắc nhở mình rằng các đơn vị ứng với từng giá trị. Điều đó sẽ giúp tôi tránh khỏi những điều vô nghĩa kiểu như cộng lực và gia tốc với nhau.

Sau đây là dạng của kết quả khi có lực cản không khí:

```
--> M = ode([4000; 0], 0, T, freefall)';
--> clf; plot(T, M)
```



Có sự khác biệt lớn. Với lực cản không khí, vận tốc tăng lên đến khi gia tốc cản bằng với g; sau đó, vận tốc là không đổi, được biết đến với tên gọi "vận tốc cuối," và vị trí sẽ hạ thấp tuyến tính theo thời gian (và sẽ chậm hơn nhiều so với trường hợp rơi trong chân không). Để kiểm tra lại kết quả một cách cẩn thận hơn, ta có thể đọc các vị trí và vận tốc cuối:

```
--> M($,1)

ans = 2441.2538           // độ cao tính bằng mét

--> M(end,2)

ans = - 60.614344              // vận tốc tính bằng m/s
```

Exercise 10.1 *Hãy tăng khối lượng của người nhảy dù, và kiểm tra lại để chắc rằng vận tốc cuối tăng lên. Quan hệ này chính là nguồn gốc dẫn đến cảm nhận trực giác rằng những vật nặng thì rơi nhanh hơn; và đúng là trong không khí, chúng rơi nhanh hơn thật!*

10.5 Nhảy dù!

Ở mục trước, ta đã thấy rằng vận tốc cuối của một người nhảy dù nặng $75kg$ là khoảng 60 m/s,vốn là khoảng 130 mph. Nếu quả thật phải tiếp đất với vận tốc như vậy, chắc là bạn sẽ chết. Đó là tại sao phải có dù.

Exercise 10.2 *Hãy sửa lại* `acceleration` *sao cho sau 30 giây tự do, người nhảy mở dù, và (gần như) lập tức tăng hằng số cản lên 2,7.*

Vận tốc cuối bây giờ bằng bao nhiêu? Bao lâu (sau khi mở dù), người nhảy sẽ tiếp đất?

10.6 Bài toán hai chiều

Đến đây ta đã dùng `ode` giải một hệ phương trình bậc nhất và một phương trình bậc hai. Bước làm hợp lý tiếp theo là một hệ phương trình bậc hai, và ví dụ hợp lý tiếp theo là về một đường bay. "Đường bay" được vạch ra bởi một vật thể bắn đi trong không khí, thường là hướng về nhằm phá hủy mục tiêu.

Nếu đường bay nằm trong một mặt phẳng, ta có thể coi hệ là hai chiều, với x biểu thị khoảng cách theo phương ngang bay được và y biểu thị độ cao. Như vậy bây giờ thay vì một người nhảy dù, bạn có thể hình dung về người trong gánh xiếc bị bắn ra từ nòng pháo.

Theo Wikipedia*, khoảng cách kỉ lục của người bị bắn từ nòng pháo là xa 56,5 mét.

Sau đây là một khung chương trình có dùng `ode` để tính đường bay của "đạn" trong không gian hai chiều:

```
function res = projectile(t, W)
    P = W(1:2);
    V = W(3:4);

    dPdt = V;
    dVdt = acceleration(t, P, V);

    res = [dPdt; dVdt];
endfunction

function res = acceleration(t, P, V)
    g = -9.8;              // gia tốc trọng lực tính bằng m/s^2
    res = [0; g];
endfunction
```

*http://en.wikipedia.org/wiki/Human_cannonball

Đối số thứ hai của hàm tốc độ là một véc-tơ, `W`, với 4 phần tử. Hai phần tử đầu được gán cho `P`, để biểu diễn vị trí; hai phần sau gán cho `V`, để biểu diễn vận tốc. `P` và `V` là các véc-tơ với các phần tử là những thành phần theo phương x và y.

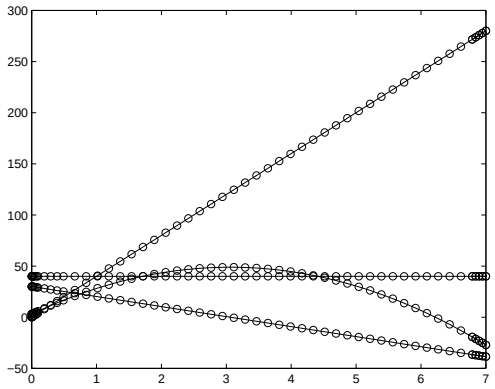
Kết quả thu được từ `acceleration` cũng là một véc-tơ; (tạm thời) bỏ qua lực cản không khí, gia tốc theo phương x bằng không, và theo phương y thì bằng g. Ngoài ra thì mã lệnh giống hệt như ta đã thấy ở Mục 10.3 .

Nếu ta phóng người bay từ độ cao ban đầu là 3 mét, với các vận tốc thành phần là 40 m/s và 30 m/s theo các phương x và y thì lời gọi hàm `ode` sẽ như sau:

```
-->T = 1:10;
-->M = ode([0,3;40,30],0,1:10,projectile)';
```

Và kết quả sẽ như sau:

```
-->plot(T, M);
```



Bạn có thể sẽ phải nghĩ một lát để hình dung ra đường nào ứng với đại lượng gì. Dường như là thời gian bay vào khoảng 6 giây.

Exercise 10.3 *Hãy lọc ra các giá trị thành phần theo các phương x và y của vị trí, vạch ra đường bay của người, rồi ước tính khoảng cách bay được.*

Exercise 10.4 *Hãy thêm sức cản không khí vào bài toán mô phỏng này. Ở trường hợp người nhảy dù, ta giả sử rằng hằng số cản bằng 0,2; nhưng nó được dựa trên giả thiết là người nhảy rơi có mặt tiếp xúc phẳng. Nhưng người bị bắn pháo, bay hướng đầu về phía trước, có lẽ sẽ ứng với hằng số cản chừng 0,1. Vận tốc ban đầu cần để đạt được khoảng cách bay kỷ lục 65,6 sẽ phải bằng bao nhiêu? Gợi ý: góc bắn tối ưu bằng bao nhiêu?*

10.7 Điều gì trục trặc có thể xảy ra?

Điều gì trục trặc có thể xảy ra? À, chẳng hạn như `vertcat`. Để giải thích từ này nghĩa là gì, tôi sẽ bắt đầu bằng **phép ghép nối**, phép toán để ghép hai ma trận thành một ma trận lớn hơn. "Vertical catenation" (ghép theo phương thẳng đứng) sẽ ghép hai ma trận bằng cách đặt chồng lên nhau (cột nối cột); "horizontal catenation" (ghép theo phương ngang) đặt hai ma trận cạnh nhau (hàng nối hàng).

Sau đây là một ví dụ ghép ngang với các véc-tơ hàng:

```
--> x = 1:3

x = 1.    2.    3.

--> y = 4:5

y = 4.    5.

--> z = [x, y]

z = 1.    2.    3.    4.    5.
```

Bên trong cặp ngoặc vuông, dấu phẩy làm nhiệm vụ ghép ngang. Còn toán tử ghép dọc là dấu chấm phẩy. Sau đây là một ví dụ với các ma trận:

```
--> X = zeros(2,3)

X =  0.    0.    0.
     0.    0.    0.

--> Y = ones(2,3)

Y =  1.    1.    1.
     1.    1.    1.

--> Z = [X; Y]

Z =  0.    0.    0.
     0.    0.    0.
```

```
     1.     1.     1.
     1.     1.     1.
```

Các toán tử này chỉ có tác dụng khi các ma trận có cùng kích thước ở chiều được ghép với nhau. Nếu không, bạn sẽ nhận được:

```
--> a = 1:3

a = 1.     2.     3.

--> b = a'

b =  1.
     2.
     3.

-->c = [a,b]
          !--error 5
Inconsistent column/row dimensions.

-->c = [a;b]
          !--error 6
Inconsistent row/column dimensions.
```

Ở ví dụ này, `a` là véc-tơ hàng còn `b` là véc-tơ cột, vì vậy chúng chẳng thể ghép được theo bất cứ chiều nào.

Các phép toán này có liên quan đến `projectile` vì dòng cuối cùng, vốn để gói `dPdt` và `dVdt` vào biến đầu ra:

```
function res = projectile(t, W)
    P = W(1:2);
    V = W(3:4);

    dPdt = V;
    dVdt = acceleration(t, P, V);

    res = [dPdt; dVdt];
endfunction
```

Miễn là cả `dPdt` và `dVdt` đều là véc-tơ cột thì dấu chấm phẩy sẽ thực hiện ghép nối theo chiều dọc, va kết quả là véc-tơ cột có 4 phần tử. Nhưng nếu một trong hai véc-tơ đó là véc-tơ hàng thì sẽ trục trặc.

`ode` trông đợi kết quả của `projectile` như một véc-tơ cột, vì vậy nếu bạn dùng `ode`, có thể sẽ tốt hơn nếu để *mọi đại lượng* dưới dạng véc-tơ cột.

Nhìn chung, nếu bạn gặp trục trặc với lỗi kiểu `Inconsistent column/row dimensions` như trên, hãy dùng `size` để hiển thị kích cỡ của các toán hạng, và đảm bảo chắc chắn véc-tơ của bạn được sắp xếp hàng-cột cho đúng.

10.8 Thuật ngữ

các hàm song song: Những hàm được định nghĩa cạnh nhau, và một hàm kết thúc trước khi hàm khác bắt đầu.

hàm lồng ghép: Hàm được định nghĩa bên trong hàm khác.

hàm ngoài: Hàm có chứa một định nghĩa hàm khác.

hàm trong: Hàm được định nghĩa bên trong một lời định nghĩa hàm khác. Hàm trong có thể truy cập các biến của hàm ngoài.

ghép nối: Phép toán nối tiếp hai ma trận để tạo nên một ma trận mới.

10.9 Bài tập

Exercise 10.5 *Đường bay của quả bóng chày bị chi phối bởi ba lực: trọng lực, lực cản không khí, và lực Magnus do chuyển động quay. Nếu ta bỏ qua gió và lực cản Magnus thì đường bay của quả bóng chày luôn nằm trong một mặt phẳng, và ta có thể mô phỏng nó như vật được bắn đi theo không gian hai chiều.*

Một mô hình đơn giản cho lực cản không khí tác dụng lên quả bóng là:

$$F_d = -\frac{1}{2}\,\rho\,v^2\,A\,C_d\,\hat{V}$$

trong đó F_d là một véc-tơ biểu diễn lực tác dụng lên quả bóng chày gây bởi lực cản, C_d là hệ số cả (0,3 là một giá trị phù hợp), ρ là mật độ không khí (1,3 kg/m^3 ở mực nước biển), A là diện tích mặt cắt ngang của quả bóng (0.0042 m^2), v là độ

lớn của véc-tơ vận tốc, còn $\hat{V}$ là một véc-tơ đơn vị theo hướng của véc-tơ vận tốc. Khối lượng của quả bóng là 0,145 kg.

Bạn có thể đọc thêm về lực cản ở trang `http://en.wikipedia.org/wiki/Drag_(physics)`.

- *Hãy viết một hàm nhận vào các biến gồm vận tốc ban đầu và góc ném của quả bóng chày, dùng* `ode` *để tính đường bay, và trả lại tầm xa (khoảng cách phương ngang bay được) là các biến đầu ra.*

- *Hãy viết một hàm nhận vào biến vận tốc ban đầu của quả bóng, tính góc ném sao cho khiến cho tầm xa là lớn nhất, và trả lại góc ném tối ưu nêu trên cùng với tầm xa như những biến đầu ra. Góc ném tối ưu thay đổi theo vận tốc ban đầu như thế nào?*

- *Khi đội Red Sox đoạt giải World Series năm 2007, họ gặp đội Colorado Rockies tại sân nhà ở Denver, Colorado. Hãy ước tính mật độ không khí ở Mile High City. Nó có ảnh hưởng gì đến sức cản không khí? Hãy dự đoán về ảnh hưởng của yếu tố này [lực cản] đến góc ném tối ưu, và dùng mô phỏng của bạn để kiểm tra lại dự đoán đó.*

- *Sân Green Monster ở Fenway Park ở độ cao khoảng 12 m và chiều dài khoảng 97 m. Vận tốc tối thiểu của quả bóng ngay sau khi vụt phải bằng bao nhiêu để bay hết chiều dài sân (giả thiết rằng bóng bay theo góc tối ưu)? Bạn có nghĩ rằng một người có thể ném bóng hết chiều dài sân được không?*

- *Lực cản của không khí lên quả bóng chày thực tế còn phức tạp hơn rất nhiều so với tính toán bằng mô hình đơn giản nêu trên. Đặc biệt, hệ số cản còn thay đổi theo vận tốc. Bạn có thể đọc thêm từ cuốn* The Physics of Baseball[†]*; và cũng tìm thông tin trên web. Sau đó, hãy thiết lập một mô hình thực tế hơn về lực cản và sửa đổi chương trình tính theo lực cản mới này. Bằng mô hình mới thì tầm xa tính được sẽ thay đổi bao nhiêu? Góc ném tối ưu sẽ thay đổi bao nhiêu?*

[†]Robert K. Adair, Harper Paperbacks, 3rd Edition, 2002.

Chương 11

Tối ưu hóa và nội suy

11.1 Tối ưu hóa

Trong Bài tập 10.5 , bạn được yêu cầu tính góc ban đầu tối ưu sau khi bóng bị vụt. "Tối ưu" là cách nói hoa mỹ cho "tốt nhất;" nghĩa của nó lại tùy thuộc vào bài toán cụ thể. Với bài toán Green Monster—tìm ra góc đánh tối ưu cho một cú hết sân thì nghĩa của "tối ưu" không phải là hiển nhiên.

Rất dễ nghĩ rằng chọn góc mà cho tầm xa là lớn nhất (khoảng cách từ chỗ vụt đến chỗ chạm đất). Nhưng ở đây ta cần cố vượt qua một tường chắn cao 12 m, vì vậy có lẽ ta sẽ cần góc vụt sao cho tầm xa là lớn nhất khi quả bóng bay qua ngưỡng 12 m.

Mặc dù định nghĩa nào trong số trên cũng dùng được cho những mục đích nhất định, nhưng không có định nghĩa nào hoàn toàn đúng. Ở trường hợp này góc "tối ưu" là góc mà cho chiều cao lớn nhất tại điểm mà quả bóng chạm tường, vốn cách chỗ vụt 97 m.

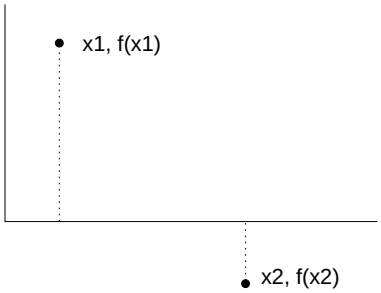
Vì vậy bước đầu tiên của mọi bài toán tối ưu là định nghĩa xem "tối ưu" là gì. Bước thứ hai là định nghĩa khoảng giá trị mà bạn muốn dò tìm. Trong trường hợp này, khoảng giá trị cho phép là giữa 0 độ (song song với mặt đất) đến 90 độ (thẳng lên trời). Ta trông đợi góc tối ưu gần với 45 độ, nhưng không chắc rằng nó chệch với 45 độ là bao nhiêu. Để an toàn, ta có thể bắt đầu tìm trong khoảng rộng nhất có thể.

Cách đơn giản nhất để tìm giá trị tối ưu là chạy chương trình mô phỏng với một khoảng rộng các giá trị rồi lựa ra giá trị cho kết quả tốt nhất. Cách này không hiệu quả lắm, đặc biệt trong bài này khi mà việc tính tầm bay là rất tốn công.

Một thuật toán khá hơn là cách Tìm kiếm theo lát cắt vàng.

11.2 Tìm kiếm theo lát cắt vàng

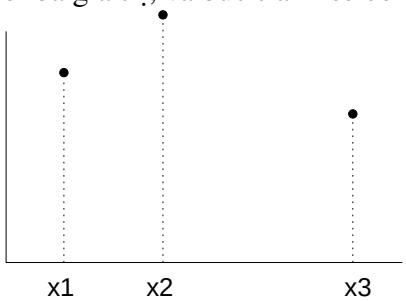
Để trình bày cách tìm kiếm theo lát cắt vàng, tôi sẽ bắt đầu với một dạng đơn giản hơn mà tôi đặt tên là tìm kiếm theo lát cắt bạc. Ý tưởng cơ bản cũng giống như các phương pháp tìm nghiệm mà ta đã thấy ở Mục 6.5 . Trong bài toán tìm nghiệm, ta có một bức tranh như sau:



Cho trước một hàm f có thể lượng giá được, và ta cần tìm một nghiệm của f; nghĩa là một giá trị của x sao cho $f(x) = 0$. Nếu ta có thể tìm được một giá trị, x_1, sao cho $f(x_1)$ dương và một giá trị khác, x_2, sao cho $f(x_2)$ âm, thì phải có một nghiệm trong khoảng giữa chúng (miễn là f liên tục). Lúc này ta nói x_1 và x_2 bao lấy nghiệm.

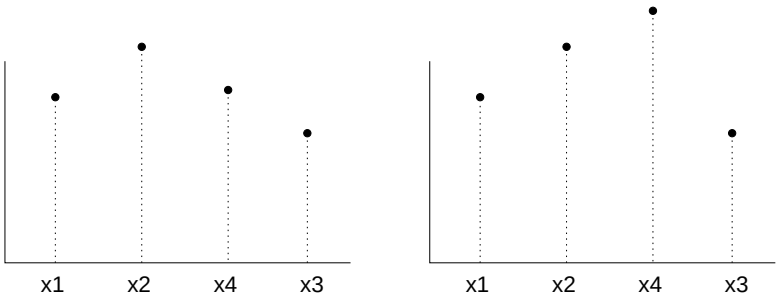
Thuật toán tiếp diễn với việc chọn một giá trị thứ ba, x_3, ở giữa x_1 và x_2 rồi tính $y = f(x_3)$. Nếu y dương thì ta có thể lập ra một cặp mới, (x_3, x_2), bao lấy nghiệm. Nếu y âm thì cặp (x_1, x_3) bao lấy nghiệm. Bằng cách nào đi nữa thì khoảng bao cũng hẹp lại và ước đoán của ta về vị trí nghiệm trở nên chính xác hơn.

Đó là việc tìm nghiệm. Còn cách tìm kiếm theo lát cắt vàng cũng tương tự, nhưng ta phải khởi đầu với ba giá trị, và bức tranh sẽ có dạng sau:



Biểu đồ này cho thấy rằng ta đã tính f tại ba vị trí, x_1, x_2 và x_3, rồi biết được x_2 cho giá trị lớn nhất. Nếu f liên tục, thì có ít nhất là một cực trị địa phương ở giữa x_1 và x_3, vì vậy ta sẽ nói rằng bộ ba (x_1, x_2, x_3) bao lấy một cực đại.

Bước tiếp theo là chọn điểm thứ tư, x_4, rồi tính $f(x_4)$. Có hai kết quả có thể xảy ra, tùy theo là $f(x_4)$ có lớn hơn $f(x_2)$' hay không:



Nếu $f(x_4)$ nhỏ hơn $f(x_2)$ (hình bên trái), thì bộ ba mới (x_1, x_2, x_4) bao lấy cực đại. Nếu $f(x_4)$ lớn hơn $f(x_2)$ (hình bên phải), thì (x_2, x_4, x_3) bao lấy cực đại. Dù với cách nào đi nữa thì khoảng bao cũng thu hẹp lại và ước tính giá trị cực đại của x càng tốt hơn.

Phương pháp này áp dụng được cho hầu hết các giá trị của x_4, nhưng có cách chọn nhất định sẽ có hiệu quả hơn. Ở đây, tôi chọn cách chia đôi khoảng lớn hơn trong số hai khoảng (x_1, x_2) và (x_2, x_3).

Sau đây là đoạn chương trình trong Scilab:

```
function res = optimize(V)
    x1 = V(1);
    x2 = V(2);
    x3 = V(3);

    fx1 = height_func(x1);
    fx2 = height_func(x2);
    fx3 = height_func(x3);

    for i=1:50
        if x3-x2 > x2-x1
            x4 = (x2+x3) / 2;
            fx4 = height_func(x4);
            if fx4 > fx2
                x1 = x2;  fx1 = fx2;
                x2 = x4;  fx2 = fx4;
            else
                x3 = x4;  fx3 = fx4;
            end
        else
            x4 = (x1+x2) / 2;
```

```
            fx4 = height_func(x4);
            if fx4 > fx2
                x3 = x2;  fx3 = fx2;
                x2 = x4;  fx2 = fx4;
            else
                x1 = x4;  fx1 = fx4;
            end
        end

        if abs(x3-x1) < 1e-2
            break
        end
    end
    res = [x1 x2 x3];
endfunction
```

Biến đầu vào là một véc-tơ có chứa ba giá trị và bao lấy một cực đại; trong trường hợp này là các góc tính theo độ. `optimize` bắt đầu bằng việc ước lượng `height_func` cho ba giá trị này. Ta giả sử rằng `height_func` trả lại kết quả mà ta cần tối ưu hóa; trong bài toán Green Monster đó chính là chiều cao của quả bóng khi chạm tường.

Mỗi lượt lặp qua vòng `for` hàm sẽ chọn một giá trị của `x4`, tính `height_func`, rồi cập nhật bộ ba `x1`, `x2` và `x3` tùy theo kết quả thu được.

Sau khi cập nhật, nó tính độ dài khoảng bao nghiệm, `x3-x1`, rồi kiểm tra xem nó đã đủ ngắn chưa. Nếu được rồi, nó sẽ thoát khỏi vòng lặp và trả lại kết quả là bộ ba hiện thời. Trong trường hợp dở nhất, vòng lặp sẽ được thực hiện 50 lần.

Exercise 11.1 *Tôi gọi thuật toán này là Tìm kiếm theo lát cắt bạc vì nó tốt gần bằng cách Tìm kiếm theo lát cắt vàng. Hãy đọc Wikipedia về cách Tìm kiếm theo lát cắt vàng (`http://en.wikipedia.org/wiki/Golden_section_search`) rồi sửa lại mã lệnh trên để tính theo cách mới này.*

Exercise 11.2 *Bạn có thể viết các hàm nhận vào các hàm khác, như ta đã thấy ở* `fzero` *và* `ode`. *Chẳng hạn,* `handle\_func` *nhận vào một chuôi hàm tên là* `func` *rồi gọi nó, truyền vào đối số là* `pi`.

```
function res = handle_func(func)
    res = func(%pi)
```

```
endfunction
```

Bạn có thể gọi `handle\_func` *từ Console và truyền vào các hàm khác nhau làm đối số:*

```
--> handle_func(sin)

ans = 1.225D-16

--> handle_func(cos)

ans = -1.
```

Hãy sửa lại `optimize` *để cho nó nhận vào một chuôi hàm rồi lấy hàm này làm mục tiêu để tối ưu hóa.*

Exercise 11.3 *Hàm* `fminsearch` *của Scilab nhận vào một hàm và tìm cực tiểu địa phương của hàm này. Hãy đọc lời hướng dẫn cách dùng* `fminsearch` *rồi dùng nó để tìm góc đánh tối ưu của quả bóng chày ứng với một vận tốc cho trước.*

11.3 Ánh xạ rời rạc và liên tục

Khi bạn giải PVT theo cách giải tích, kết qủa thu được là một hàm, và bạn có thể coi rằng đó là một phép ánh xạ liên tục. Khi bạn dùng một hàm giải PVT, bạn thu được hai véc-tơ (hoặc một véc-tơ và một ma trận), mà bạn có thể coi là một phép ánh xạ rời rạc.

Chẳng hạn, ở Mục 8.4 , ta đã dùng hàm tốc độ sau để ước tính số con chuột như một hàm theo thời gian:

```
function res = rats(t, y)
    a = 0.01;
    omega = 2 * %pi / 365;
    res = a * y * (1 + sin(omega * t));
endfunction
```

Kết quả thu được từ các lệnh sau là hai véc-tơ:

```
--> T = 1:2:365;
--> Y = ode(2, 0, T, rats);
```

`T` chứa các giá trị thời gian tại đó cần ước tính số chuột bằng `ode` (cụ thể ở đây là số thứ tự ngày, đếm cách nhật); `Y` chứa các giá trị ước tính.

Bây giờ ta hình dung như cần biết số chuột vào ngày thứ 180 của năm. Ta có thể tìm giá trị 180 trong `T`:

```
--> find(T==180)

ans = []
```

Nhưng không có bảo đảm gì rằng sẽ tồn tại một giá trị như vậy trong `T`. Ta có thể tìm ra chỉ số mà tại đó giá trị của `T` cắt qua 180:

```
--> I = find(T>180); I(1)

ans = 91.
```

`I` sẽ nhận được tất cả những chỉ số ứng với các phần tử của `T` mà lớn hơn 180, vì vậy `I(1)` chính là chỉ số *đầu tiên*.

Sau đó ta tìm giá trị tương ứng của `Y`:

```
--> [T(23), Y(23)]

ans = 181.    39.047708
```

Cách này cho ta một ước tính thô sơ về số chuột vào ngày 180. Nếu cần tính kĩ hơn, ta có thể tìm thêm giá trị ngay trước ngày 180:

```
--> [T(22), Y(22)]

ans = 179.    38.241594
```

Như vậy số chuột vào ngày 180 sẽ nằm giữa 38.241594 và 39.047708.

Nhưng cụ thể con số nào trong khoảng này sẽ là ước lượng chính xác nhất? Một cách làm đơn giản là chọn ngay giá trị nào tương ứng thời gian gần với 180 hơn. Trong bài này, cách làm như vậy không hay vì giá trị thời gian mà ta cần xác định lại nằm ngay giữa.

11.4 Nội suy

Một cách làm hay hơn là vạch một đường thẳng nối hai điểm bao ngày 180 và dùng đường thẳng đó để ước tính giá trị giữa chúng. Quá trình này được gọi là **nội suy tuyến tính**, và Scilab cung cấp một hàm có tên `interp1` để đảm nhiệm việc này:

```
--> pop = interp1(T, Y, 180)

pop = 38.644651
```

Hai đối số đầu dùng để chỉ một phép ánh xạ rời rạc từ các giá trị có trong `T` đến các giá trị trong `Y`. Đối số thứ ba là giá trị thời gian mà ta cần nội suy. Kết quả thu được đúng như ta trông đợi, nghĩa là ở chính giữa hai giá trị đầu khoảng bao. Nói cách khác, số chuột dự tính theo cách này bằng trung bình cộng số chuột trước đó một ngày và số chuột sau đó một ngày.

`interp1` cũng có thể nhận một đối số thứ tư để chỉ dạng nội suy mà bạn muốn. Mặc định là `'linear'`, vốn thực hiện nội suy tuyến tính. Một cách lựa chọn khác là `'spline'` vốn dùng một đường cong spline để lượn qua bốn điểm,
.

```
--> pop = interp1(T, Y, 180, 'spline')

pop = 38.645872
```

Ở trường hợp này ta trông đợi các kết quả thu được từ nội suy spline tốt hơn so với tuyến tính, vì chúng dùng nhiều điểm số liệu hơn, và ta biết rằng hàm không phải tuyến tính.

Ta cũng có thể dùng `interp1` để kéo dài đồ thị của số chuột ra ngoài khoảng các giá trị có trong `T`:

```
--> [T($), Y($)]

ans = 365.    76.949311

--> pop = interp1(T, Y, 370, 'spline')

pop = 81.067345
```

Quá trình này được gọi là **ngoại suy**. Với những giá trị thời gian còn gần 365, ngoại suy vẫn có vẻ hợp lý, nhưng càng xa về phía "tương lai," hẳn sẽ càng kém chính xác.

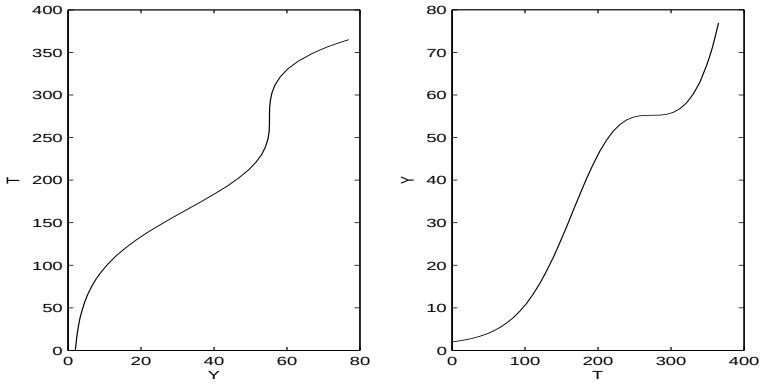
11.5 Nội suy hàm ngược

Ta đã dùng `interp1` để tìm số chuột như một hàm của thời gian; và bằng cách đảo ngược vai trò của `T` và `Y`, ta cũng có thể nội suy thời gian như một hàm của số chuột. Chẳng hạn, ta cần biết sau bao lâu thì số chuột sẽ đạt đến 20.

```
--> interp1(Y, T, 20)

ans = 133.5213
```

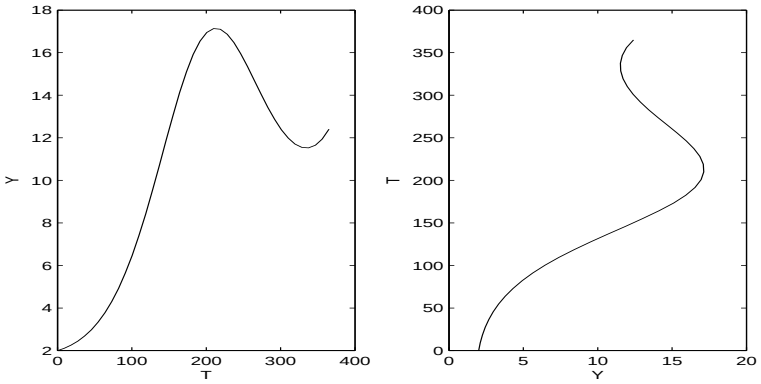
Cách dùng `interp1` thế này có thể dễ nhầm lẫn nếu bạn nghĩ rằng các đối số như là x và y. Có thể hay hơn là bạn hình dung chúng như tập nguồn và tập đích trong một phép ánh xạ (còn đối số thứ ba là phần tử trong tập nguồn).

Hình vẽ dưới đây cho thấy f (`Y` vẽ theo `T`) và nghịch đảo của f (`T` vẽ theo `Y`).



Ở trường hợp này ta có thể dùng `interp1` theo cách nào cũng được vì f có **ánh xạ đơn trị**, tức là mỗi giá trị của tập đích chỉ có một giá trị từ tập nguồn có ánh xạ đến nó.

Nếu ta giảm lượng cung cấp thức ăn sao cho số chuột giảm trong thời kì mùa đông thì có thể sẽ thấy kết quả sau:



Ta vẫn dùng được `interp1` để ánh xạ từ `T` đến `Y`:

```
--> interp1(T, Y, 260)

ans = 15.0309
```

Như vậy là vào ngày 260, số chuột có khoảng 15 con, nhưng nếu ta cần biết vào ngày nào số chuột có 15 con thì sẽ tồn tại hai câu trả lời: 172,44 và 260,44. Nếu thử dùng `interp1`, ta sẽ nhận được kết quả sai:

```
--> interp1(Y, T, 15)

ans = 196.3833                  % SAI
```

Vào ngày 196, số chuột thực tế là 16,8; vì vậy `interp1` còn không đạt gần con số đó! Vấn đề là ở chỗ `T` với vai trò hàm số của `Y` là một **ánh xạ đa trị**; với một giá trị nào đó ở tập nguồn, có hơn một giá trị ở tập đích. Điều này làm cho `interp1` không tính đúng. Tôi không thể tìm thấy tài liệu nào viết về sự hạn chế nói trên, thật hơi tệ.

11.6 Chuột đồng

Như ta đã thấy, một công dụng của nội suy là để diễn giải kết quả của một bài toán số trị; song còn một công dụng khác là để lấp đầy những khoảng trống giữa các số liệu đo rời rạc.

Chẳng hạn*, giả sử rằng số chuột đồng bị chi phối bởi phương trình tốc độ:

*Ví dụ này được trích có sửa đổi từ Gerald & Wheatley, *Applied Numerical Analysis*, Fourth Edition, Addison-Wesley, 1989.

$$g(t, y) = ay - b(t)y^{1.7}$$

trong đó t là thời gian tính theo tháng, y là số chuột, a là một tham số đặc trưng cho tốc độ tăng số chuột trong trường hợp không hạn chế, còn b là hàm số theo thời gian, đặc trưng cho ảnh hưởng của lương thực được cấp đến tốc độ chết.

Mặc dù b xuất hiện trong phương trình như một hàm liên tục, ta có thể sẽ không biết được $b(t)$ với mọi t. Thay vào đó, ta có thể chỉ có các số liệu đo rời rạc sau đây:

```
t           b(t)
-           ----
0           0.0070
1           0.0036
2           0.0011
3           0.0001
4           0.0004
5           0.0013
6           0.0028
7           0.0043
8           0.0056
```

Nếu dùng `ode` để giải phương trình vi phân, thì ta sẽ không thể tiến gần đến những giá trị của t khi hàm tốc độ (và do đó cả b) được lượng giá. Ta cần cung cấp một hàm cho phép xác định b bất kì lúc nào:

```
function res = interpolate_b(t)
    T = 0:8;
    B = [70 36 11 1 4 13 28 43 56] * 1e-4;
    res = interp1(T, B, t);
endfunction
```

Nhìn bao quát, hàm này dùng một phép ánh xạ rời rạc để lập ra một ánh xạ liên tục.

Exercise 11.4 *Hãy viết một hàm tốc độ trong đó dùng* `interpolate_b` *để lượng giá* g *rồi dùng* `ode` *để tính số chuột đồng từ* $t = 0$ *đến* $t = 8$ *với số chuột ban đầu bằng 100 và* $a = 0.9$.

Sau đó sửa lại `interpolate_b` *để dùng nội suy spline và chạy lại* `ode` *để xem phép nội suy ảnh hưởng nhiều đến các kết quả hay không.*

11.7 Thuật ngữ

nội suy: Ước tính giá trị của một hàm dựa vào các giá trị đã biết ở hai phía.

ngoại suy: Ước tính giá trị của một hàm dựa vào các giá trị đã biết nhưng không bao lấy giá trì cần tìm.

ánh xạ đơn trị: Ánh xạ trong đó mỗi giá trị của tập nguồn chiếu đến một giá trị ở tập đích.

ánh xạ đa trị: Ánh xạ trong đó có ít nhất một giá trị ở tập nguồn chiếu đến nhiều giá trị ở tập đích.

11.8 Bài tập

Exercise 11.5 *Một quả bóng golf*[†] *được đánh theo cú xoáy ngược sẽ phát sinh lực nâng, vốn có thể làm tăng tầm xa, nhưng năng lượng để tạo ra độ xoáy có thể sẽ làm giảm vận tốc ban đầu. Hãy viết một chương trình mô phỏng đường bay của một quả bóng golf rồi dùng nó để tính góc đánh và sự phân chia năng lượng để tạo độ xoáy và vận tốc ban đầu (từ một nguồn năng lượng nhất định) sao cho tầm xa theo phương ngang của quả bóng đạt cực đại.*

Hiện tượng quả bóng xoáy bị nâng lên là do lực Magnus[‡]*; lực này vuông góc với trục quay và đường bay. Hệ số nâng tỉ lệ với tốc độ quay, và bằng khoảng 0,1 đối với quả bóng quay được 3000 vòng/phút. Hệ số cản của quả bóng vào khoảng 0,2 khi quả bóng bay nhanh hơn 20 m/s.*

[†]Xem `http://en.wikipedia.org/wiki/Golf_ball`.

[‡]Xem `http://en.wikipedia.org/wiki/Magnus_effect`.

Chương 12

Bây giờ véc-tơ mới thật là véc-tơ

12.1 Véc-tơ là gì?

Từ "véc-tơ" có thể mang những nghĩa khác nhau đối với từng người. Trong Scilab, véc-tơ là một ma trận chỉ có một hàng, hoặc một cột. Cho đến giờ, ta đã dùng các véc-tơ của Scilab để biểu diễn:

dãy: Dãy là một tập hợp các giá trị được nhận diện bởi các chỉ số nguyên; theo cách làm tự nhiên ta có thể lưu các phần tử của dãy như những phần tử của một véc-tơ trong Scilab.

véc-tơ trạng thái: Véc-tơ trạng thái là một tập hợp các giá trị để mô tả trạng thái của một hệ vật lý. Khi gọi `ode`, bạn cho nó các điều kiện ban đầu dưới dạng một véc-tơ trạng thái. Sau đó, khi `ode` gọi hàm tốc độ mà bạn lập nên, nó sẽ trả kết quả là một véc-tơ trạng thái khác.

ánh xạ rời rạc: Nếu có trong tay hai véc-tơ cùng độ dài, bạn có thể hình dung chúng như một phép ánh xạ từ những phần tử của một véc-tơ này sang các phần tử thuộc véc-tơ kia. Chẳng hạn, ở Mục **??**, kết quả thu được từ `ode` là các véc-tơ, `T` và `Y`; chúng biểu diễn một phép ánh xạ từ các giá trị thời gian của `T` sang các giá trị số lượng chuột có trong `Y`.

Trong chương này ta sẽ xét đến một công dụng khác của véc-tơ trong Scilab: để biểu diễn các véc-tơ không gian. Một véc-tơ không gian là một giá trị nhằm biểu diễn một đại lượng vật lý nhiều chiều, như vị trí, vận tốc, gia tốc, hoặc lực*.

*Xem `http://en.wikipedia.org/wiki/Vector_(spatial)`.

Các đại lượng nói trên không thể diễn tả bởi một con số vì chúng có chứa nhiều thành phần. Chẳng hạn, trong không gian tọa độ Đề-các 3 chiều, ta cần phải có 3 con số mới xác định được một vị trí trong không gian, mà thường gọi là các tọa độ x, y và z. Một ví dụ khác, trong không gian tọa độ cực 2 chiều, bạn có thể xác định một vận tốc bằng hai con số, một là độ lớn và thứ hai là góc, thường được gọi là r và θ.

Biểu diễn các véc-tơ không gian bằng véc-tơ trong Scilab rất tiện vì Scilab biết cách thực hiện hầu hết các phép tính với véc-tơ phục vụ cho việc mô phỏng vật lý. Chẳng hạn, giả sử bạn có vận tốc của quả bóng chày theo dạng véc-tơ Scilab với hai phần tử, v_x và v_y, chính là hai thành phần của vận tốc theo các phương x và y.

```
--> V = [30, 40];          // vận tốc tính theo m/s
```

Và giả sử rằng bạn cần phải tính gia tốc tổng hợp của quả bóng dưới tác dụng của lực cản không khí và trọng lực. Theo ký hiệu toán học, lực cản được tính bởi

$$F_d = -\frac{1}{2}\,\rho\,v^2\,A\,C_d\,\hat{V}$$

trong đó V là véc-tơ không gian biểu diễn vận tốc, v là độ lớn của vận tốc (còn được gọi là "tốc độ"), và $\hat{V}$ là một véc-tơ đơn vị theo hướng của véc-tơ vận tốc. Các đại lượng khác, ρ, A và C_d đều là số vô hướng.

Độ lớn của véc-tơ thì bằng căn bậc hai của tổng các bình phương của từng phần tử. Bạn có thể tính với `hypotenuse` ở Mục 5.5 , hoặc dùng hàm Scilab có tên `norm` (**norm**, hay chuẩn, là tên gọi khác [†] cho độ lớn của véc-tơ):

```
--> v = norm(V)

v = 50.
```

$\hat{V}$ là một **véc-tơ đơn vị**, nghĩa là nó phải có chuẩn bằng 1, và chỉ theo cùng hướng với V. Cách dễ nhất để tính véc-tơ đơn vị này là chia V cho chuẩn của chính nó.

```
--> Vhat = V / v

Vhat =  0.6  0.8
```

Bây giờ ta có thể đảm bảo rằng chuẩn của $\hat{V}$ bằng 1:

[†]Độ lớn đôi khi cũng được gọi là độ dài ("length") nhưng tôi tránh dùng từ này vì sẽ lẫn với hàm `length` để trả lại số phần tử có trong một véc-tơ của Scilab.

```
--> norm(Vhat)

ans = 1.
```

Để tính F_d ta chỉ cần nhân các đại lượng vô hướng với $\hat{V}$.

```
Fd = - 1/2 * C * rho * A * v^2 * Vhat
```

Tương tự như vậy, ta có thể tính gia tốc bằng cách chia véc-tơ F_d cho số vô hướng m.

```
Ad = Fd / m
```

Để biểu diễn gia tốc trọng trường, ta tạo nên một véc-tơ gồm hai phần tử:

```
Ag = [0; -9.8]
```

Phần tử x của gia tốc trọng trường bằng 0; phần tử y bằng $-9.8m/s^2$.

Cuối cùng ta tính gia tốc tổng cộng bằng cách cộng véc-tơ lại:

```
A = Ag + Ad;
```

Một điều hay trong cách tính này là ta không cần phải nghĩ nhều về từng thành phần của véc-tơ. Bằng cách coi véc-tơ không gian như những đại lượng cơ bản, ta có thể diễn tả các phép tính phức tạp một cách ngắn gọn.

12.2 Tích vô hướng và tích hữu hướng

Việc nhân một véc-tơ với số vô hướng là quá rõ ràng; cộng hai véc-tơ cũng vậy. Nhưng nhân hai véc-tơ thì tinh vi hơn. Hóa ra rằng có hai phép toán véc-tơ trông tựa như phép nhân, đó là **tích vô hướng** và **tích hữu hướng**.

Tích vô hướng của hai véc-tơ A và B cho kết quả là số vô hướng:

$$d = ab\cos\theta$$

trong đó a là độ lớn của A, b là độ lớn của B, còn θ là góc giữa hai véc-tơ. Ta đã biết cách tính các độ lớn, và có thể hình dung ra cách tính θ, nhưng không cần thiết phải làm việc đó. Scilab có một hàm, `dot`, để tính tích vô hướng.

```
d = A * B'
```

cách này áp dụng được với số chiều bất kì, miễn là `A` và `B` phải là véc-tơ hàng có cùng số phần tử.

Nếu một trong hai toán hạng là véc-tơ đơn vị thì bạn có thể dùng tích vô hướng để tính ra thành phần của véc-tơ A theo hướng véc-tơ đơn vị $\hat{i}$ đó:

```
s = A * ihat'
```

Ở ví dụ này, s là **hình chiếu vô hướng** của A lên phương $\hat{i}$. Còn **hình chiếu véc-tơ** là một véc-tơ có độ lớn s theo phương $\hat{i}$:

```
V = (A * ihat') * ihat
```

Tích hữu hướng của hai véc-tơ A và B là một véc-tơ hướng vuông góc với A và B và có độ lớn:

$$c = ab \sin \theta$$

trong đó (một lần nữa) a là độ lớn của A, b là độ lớn của B, còn θ là góc giữa hai véc-tơ. Scilab không có sẵn hàm để tính tích hữu hướng.

```
function val=crossproduct(A, B)
val = [A(2) * B(3) - A(3) * B(2) ;
A(3) * B(1) - A(1) * B(3)
A(1) * B(2) - A(2) * B(1)];
endfunction
```

`crossproduct` chỉ tính được với véc-tơ 3 chiều; và kết quả cũng là một véc-tơ 3 chiều khác.

Một cách hay dùng của `crossproduct` là để tính các mô-men lực. Nếu bạn biểu diễn tay đòn mô-men R và lực F dưới dạng các véc-tơ 3 chiều, thì mô-men lực sẽ được tính một cách đơn giản:

```
Tau = crossproduct(R, F)
```

Nếu các thành phần của `R` được tính bằng mét và các thành phần của `F` tính bằng newton, thì các thành phần trong mô-men lực `Tau` được tính bằng newton-mét.

12.3 Cơ học thiên thể

Việc mô hình hóa cơ học thiên thể là một dịp tốt để ta thực hiện tính toán với véc-tơ không gian. Hãy tưởng tượng một ngôi sao với khối lượng m_1 tại một điểm trong không gian được mô tả bởi véc-tơ P_1, và một hành tinh khối lượng m_2 tại điểm P_2. Độ lớn của lực hấp dẫn[‡] giữa chúng là

$$f_g = G\frac{m_1 m_2}{r^2}$$

trong đó r là khoảng cách giữa chúng, còn G là hằng số hấp dẫn, bằng khoảng $6.67 \times 10^{-11} Nm^2/kg^2$. Nhớ rằng đây là giá trị của G chỉ trong trường hợp ta tính khối lượng theo ki-lô-gam, khoảng cách theo mét, và lực theo newton.

Hướng của lực lên ngôi sao tại P_1 là chỉ về phía P_2. Ta có thể tính hướng tương đối giữa chúng bằng cách trừ véc-tơ; nếu ta tính `R = P2 - P1`, thì hướng của `R` bây giờ sẽ chỉ từ `P1` đến `P2`.

Khoảng cách giữa hành tinh và ngôi sao là chiều dài của R:

```
r = norm(R)
```

Hướng của lực tác dụng lên ngôi sao là $\hat{R}$:

```
rhat = R / r
```

Exercise 12.1 *Hãy viết một dãy lệnh Scilab để tính* `F12` *là véc-tơ biểu diễn lực mà hành tinh tác dụng lên ngôi sao, và* `F21`*, là lực do ngôi sao tác dụng lên hành tinh.*

Exercise 12.2 *Hãy gói các câu lệnh sau vào một hàm có tên* `gravity_force_func` *nhận vào các biến* `P1`*,* `m1`*,* `P2`*, và* `m2` *rồi trả lại* `F12`*.*

Exercise 12.3 *Hãy viết một chương trình mô phỏng quỹ đạo của Sao Mộc quanh Mặt Trời. Khối lượng của Mặt Trời vào khoảng* 2.0×10^{30} *kg. Bạn có thể lấy số liệu khối lượng của Sao Mộc cũng như khoảng cách đến Mặt Trời và vận tốc của Sao Mộc trên quỹ đạo từ trang* `http://en.wikipedia.org/wiki/Jupiter`*. Chạy chương trình để đảm bảo rằng Sao Mộc quay trọn 1 vòng quỹ đạo quanh Mặt Trời hết khoảng 4332 ngày.*

[‡]See `http://en.wikipedia.org/wiki/Gravity`

12.4 Tạo hình chuyển động

Hình động là một công cụ hữu ích để kiểm tra kết quả tính toán từ một mô hình vật lý. Nếu có gì trục trặc xảy ra, mọi sự sẽ rõ ràng trên hình động. Có hai cách làm hình động trong Scilab. Một cách là dùng `getframe` để chụp một loạt các ảnh và `movie` để phát lại chúng. Một cách làm khác, không chính thức, là vẽ một loạt các hình đồ thị. Sau đây là một ví dụ tôi viết cho Bài tập 12.3 :

```
function animate_func(T,M)
    // tạo hình chuyển động của các vệ tính,
    // giả sử rằng các cột của M là x1, y1, x2, y2.
    X1 = M(:,1);
    Y1 = M(:,2);
    X2 = M(:,3);
    Y2 = M(:,4);

    minmax = [min([X1;X2]), max([X1;X2]), min([Y1;Y2]), max([Y1;Y2]

    for i=1:length(T)
        clf;
        a = gca(); a.data_bounds = minmax;
        draw_func(X1(i), Y1(i), X2(i), Y2(i));
        drawnow;
    end
endfunction
```

Các biến đầu vào là kết quả đầu ra từ `ode`, một véc-tơ `T` và một ma trận `M`. Các cột của `M` chứa các vị trí và vận tốc của Mặt Trời và Sao Mộc, vì vậy `X1` và `Y1` nhận các tọa độ của Mặt Trời; còn `X2` và `Y2` nhận các tọa độ của Sao Mộc.

`minmax` là một véc-tơ gồm 4 phần tử được dùng bên trong vòng lặp để thiết lập các trục trên biểu đồ. Thông tin này cần thiết bởi vì nếu không, Scilab sẽ tự co dãn hình mỗi lần qua một lượt lặp, và các trục sẽ luôn thay đổi, dẫn đến khó theo dõi hình động.

Qua mỗi lượt lặp, `animate_func` dùng `clf` để xóa hình và `axis` để đặt lại các trục. `hold on` giúp ta có thể vẽ được nhiều đường đồ thị lên cùng một hệ trúc (nếu không Scilab sẽ tự xóa hình mỗi lần bạn gọi câu lệnh `plot` mới).

Cũng qua mỗi lượt lặp, ta phải gọi `drawnow` để Scilab thực sự hiển thị từng hình vẽ. Còn nếu không, nó sẽ đợi đến tận khi bạn vẽ xong hết các hình *rồi mới* cập nhật quá trình hiển thị.

`draw_func` là hàm số đảm nhiệm việc vẽ đồ thị:

```
function draw_func(x1, y1, x2, y2)
    plot(x1, y1, 'r.', 'MarkerSize', 50);
    plot(x2, y2, 'b.', 'MarkerSize', 20);
endfunction
```

Các biến đầu vào là vị trí của Mặt Trời và Sao Mộc. `draw_func` dùng `plot` để vẽ Mặt Trời như một dấu đỏ lớn và Sao Mộc như một dấu xanh lam nhỏ hơn.

Exercise 12.4 *Để chắc rằng bạn hiểu được cách hoạt động của* `animate_func`*, hãy thử chú thích để loại ra một số dòng lệnh xem điều gì xảy ra.*

Một hạn chế của kiểu hình động này là tốc độ của hình động phụ thuộc vào máy của bạn thực hiện việc vẽ hình lúc nào. Vì kết quả từ `ode` thường không xuất ra đều theo thời gian nên hình động có thể sẽ chậm lại khi `ode` nhận bước thời gian ngắn và nhanh lên khi bước thời gian dài hơn.

Có hai cách sửa vấn đề này:

1. Khi gọi `ode` bạn có thể chỉ định một véc-tơ chứa các thời điểm cần phải tính kết quả. Sau đây là một ví dụ:

   ```
   end_time = 1000;
   step = end_time/200;
   [T, M] = ode(W, 0, [0:step:end_time], rate_func);
   ```

   Đối số thứ ba là một véc-tơ khoảng số chạy từ 0 đến 1000 với bước chạy quy định bởi `step`. Vì `step` bằng `end_time/200`, nên sẽ có khoảng 200 hàng trong `T` và `M` (chính xác là 201).

   Tùy chọn này sẽ không ảnh hưởng đến độ chính xác của kết quả; `ode` vẫn dùng biến thời gian để thực hiện ước tính, nhưng rồi nó sẽ nội suy các giá trị trước khi trả lại kết quả.

2. Bạn có thể dùng `sleep` để chạy hình động theo đúng thời gian thật. Sau khi vẽ từng hình và gọi nó `drawnow`, bạn có thể tính thời gian từ đó đến hình kế tiếp và dùng `sleep` để đợi (trong một khoảng thời gian tính bằng mili-giây):

```
dt = T(i+1) - T(i);
sleep(dt);
```

Một hạn chế của phương pháp này là nó bỏ qua thời gian cần để vẽ hình, vì vậy nó có xu hướng chạy chậm, đặc biệt nếu hình vẽ phức tạp hoặc bước thời gian quá ngắn.

Exercise 12.5 *Hãy dùng* `animate_func` *và* `draw_func` *để hiển thị kết quả mô phỏng Sao Mộc. Sửa các hàm này để sao cho thời gian một ngày mô phỏng tương đương với 0,001 giây đồng hồ—mỗi vòng quay sẽ mất khoảng 4,3 giây.*

12.5 Bảo toàn năng lượng

Một cách làm tiện dụng để kiểm tra độ chính xác của một chương trình giải PVT là xem nó có bảo toàn được năng lượng của hệ hay không. Với chuyển động của hành tinh, hóa ra là `ode` không thể bảo toàn được.

Động năng của một vật chuyển động bằng $mv^2/2$; động năng của hệ Mặt Trời thì bằng tổng các động năng của từng hành tinh và của Mặt Trời. Thế năng của Mặt Trời với khối lượng m_1 và một hành tinh có khối lượng m_2 cách nhau một khoảng r thì bằng:

$$U = -G\frac{m_1 m_2}{r}$$

Exercise 12.6 *Hãy viết một hàm có tên* `energy_func` *để nhận vào các kết quả* `T` *và* `M` *từ chương trình mô phỏng Sao Mộc, rồi tính tổng năng lượng (động năng và thế năng) của hệ với mỗi vị trí và vận tốc ước tính được. Vẽ đồ thị của kết quả này như một hàm theo thời gian và đảm bảo rằng nó giảm dần trong quá trình mô phỏng. Hàm cần viết phải tính được sự thay đổi tương đối về năng lượng, độ chênh lệch giữa năng lượng ban đầu và lúc kết thúc, và biểu thị độ chênh này theo phần trăm so với năng lượng ban đầu.*

Bạn có thể giảm tốc độ thất thoát năng lượng bằng cách giảm tùy chọn dung sai của `ode` qua việc dùng tham số `rtol`:

```
[T, M] = ode45(W, 0, [0:step:end_time], rtol=1e-5, rate_func);
```

Tên của tùy chọn cần chỉnh là `RelTol`, viết tắt của "relative tolerance" (dung sai tương đối). Giá trị mặc định là `1e-3` hay 0,001. Các giá trị nhỏ hơn sẽ khiến cho `ode` thêm "khắt khe", vì vậy cần hàm sẽ cần tính nhiều hơn để giảm nhỏ sai số.

Exercise 12.7 *Hãy chạy* `ode` *với một loạt các giá trị của* `rtol` *và đảm bảo rằng khi dung sai càng nhỏ thì tốc độ thất thoát năng lượng cũng chậm lại.*

Exercise 12.8 *Hãy chạy chương trình mô phỏng bạn viết với một trong các hàm giải ODE khác có trong Scilab và xem chúng có bảo toàn năng lượng không.*

12.6 Mô hình dùng để làm gì?

Trong mục 7.2 , tôi đã định nghĩa một "mô hình" như một sự mô tả được giản hóa về một hệ vật lý, và đề cập rằng một mô hình tốt rất thích hợp cho việc phân tích và mô phỏng, và khiến cho các ước tính đủ độ tin cậy để dùng cho mục đích định trước.

Từ đó, ta đã thấy một số ví dụ; bây giờ ta có thể nói thêm những công dụng của mô hình. Các mục đích mà mô hình cần đạt đến thường có xu hướng rơi vào ba dạng sau.

dự đoán: Một số mô hình có thể dự đoán về các hệ vật lý. Lấy một ví dụ đơn giản, mô hình con vịt ở Bài tập 6.2 dự đoán về độ cao mà con vịt sẽ nổi. Ở phía đầu kia xét về độ phức tạp, mô hình khí tượng toàn cầu dự đoán tình hình thời tiết hàng mười năm hoặc trăm năm trong tương lai.

thiết kế: Mô hình rất có ích trong thiết kế kĩ thuật, đặc biệt là để kiểm tra tính khả thi của một phương án thiết kế, và để tối ưu hóa. Chẳng hạn, trong Bài tập 11.5 bạn được yêu cầu phải thiết kế cú đánh golf với sự kết hợp hoàn hảo giữa góc vụt, vận tốc và độ xoáy.

giải thích: Mô hình có thể giải thích các câu hỏi khoa học. Chẳng hạn, mô hình Lotka-Volterra trong Mục 9.4 đề xuất một lời giải thích cho cơ chế biến động của hệ quần thể động vật dưới hình thức các mối tương tác giữa loài ăn thịt và vật mồi.

Các ví dụ ở cuối chương này sẽ gồm mỗi mô hình theo từng loại kể trên.

12.7 Thuật ngữ

véc-tơ không gian: Một giá trị dùng để biểu thị một đại lượng vật lý gồm nhiều chiều; chẳng hạn vị trí, vận tốc, gia tốc, và lực.

chuẩn: Độ lớn của một véc-tơ. Đôi khi còn được gọi là "độ dài," nhưng không được nhầm với số phần tử thuộc một véc-tơ trong Scilab.

véc-tơ đơn vị: Véc-tơ có chuẩn bằng 1, dùng để biểu thị hướng.

tích vô hướng: Số vô hướng, là kết quả phép nhân hai véc-tơ, nó tỉ lệ với độ lớn của từng véc-tơ và cô-sin của góc giữa chúng.

tích hữu hướng: Véc-tơ là kết quả phép nhân hai véc-tơ khác, với chuẩn tỉ lệ với chuẩn của từng véc-tơ và với sin của góc giữa chúng, và có hướng vuông góc với cả hai véc-tơ.

hình chiếu: Thành phần của một véc-tơ xét theo hướng của véc-tơ kia (có thể được dùng với nghĩa "hình chiếu vô hướng" hay "hình chiếu véc-tơ").

12.8 Bài tập

Exercise 12.9 *Nếu bạn đặt hai bát nước giống hệt nhau vào ngăn đá tủ lạnh, nước ở trong một bát thì ở nhiệt độ bình thường còn ở bát kia thì đang sôi, liệu bát nước nào sẽ đông đá trước?*

Gợi ý: bạn có thể sẽ cần tìm hiểu thêm về hiệu ứng Mpemba.

Exercise 12.10 *Bạn được yêu cầu phải thiết kế một dường dốc để trượt ván; khác với những dốc trượt thông thường, cái này có thể xoay (kiểu bập bênh) quanh một chốt cố định. Người trượt ván tiến đến đường dốc, trước đó còn đi trên đất bằng, rồi lập tức leo dốc; họ không được phép đặt chân xuống. Nếu họ trượt đủ nhanh thì đường dốc sẽ xoay và họ sẽ chuyển tư thế thành trượt xuống một cách nhẹ nhàng. Sẽ có ban giám khảo chấm điểm kĩ thuật và nghệ thuật cho động tác trượt.*

Việc của bạn là thiết kế một dốc trượt để cho phép người trượt hoàn thành bài biểu diễn này, và tạo ra một mô hình vật lý của hệ, một chương trình mô phỏng để tính ra chuyển động của người trượt trên dốc, và hình động biểu diễn kết quả tính được.

Exercise 12.11 *Một hệ hai sao bao gồm hai ngôi sao quay xung quanh lẫn nhau và đôi khi có cả những hành tinh quay quanh từng ngôi sao [§]. Trong hệ hai sao, một số quỹ đạo là "bền vững" theo nghĩa một hành tinh có thể bay trong quỹ đạo đó mà không bị đâm vào một trong hai sao, và không bị bay tuốt vào không trung.*

[§]Xem `http://en.wikipedia.org/wiki/Binary_star`.

Mô phỏng là công cụ có ích phục vụ cho việc nghiên cứu bản chất của các quỹ đạo này, như đề cập đến trong Holman, M.J. and P.A. Wiegert, 1999, "Long-Term Stability of Planets in Binary Systems," Astronomical Journal *117, có thể tải về từ* `http://citeseer.ist.psu.edu/358720.html`.

Hãy đọc bài báo này rồi sửa lại chương trình mô phỏng hành tinh đã viết để lặp lại hoặc mở rộng kết quả.