O meu código está lento: e agora?

Maria Madrugo
Duarte Maia

Introdução

Apesar de poder ser estudado sem contexto, o grande objectivo deste pdf é ser um sitio onde ir quando algum código estiver a demorar muito a correr. Assim sendo, é uma colectânea de coisas miscelâneas que a experiência mostrou por vezes serem responsáveis por esse tipo de comportamento. Antes de começar, algumas considerações gerais:

- O Mathematica tem comandos que permitem medir tempo que código demora, nomeadamente Timing, EchoTiming, RepeatedTiming e AbsoluteTiming. Estes códigos fazem mais ou menos a mesma coisa, mas com as seguintes distinções:
  - O AbsoluteTiming tem a vantagem de ter uma precisão melhor que os restantes, mas varia de computador em computador, uma vez que mede quanto comandos demoraram no ambiente no qual foram corridos. Além disso, é o único que tem em conta todos os factores, como por exemplo o tempo que demora a ir buscar informação à internet, se for caso disso.
  - O Timing retorna uma lista com o tempo que demorou e com o resultado do código.
  - O EchoTiming funciona de forma parecida ao Timing, mas dá Echo (uma variação de Print) do tempo que demora, retornando apenas o resultado do código, o que é útil se depois quisermos utilizar este resultado mais tarde.
  - O RepeatedTiming tem a vantagem de correr muitas vezes o mesmo código e retornar a média dos tempos demorados (usa o mesmo formato de output do Timing, mas substitui o tempo pelo tempo médio). Como consequência, não só tem uma precisão melhor como apresenta resultados melhores em operações que nem sempre demoram o mesmo tempo[1]. Um cuidado a ter com este comando é que ele de facto efectua as alterações associadas. Por exemplo, o código a={}; RepeatedTiming[AppendTo[a,0]] retorna o tempo médio, juntamente com a nova lista a, que tem agora todos os zeros que foi adicionando.

  Destes, o Timing é o mais usado, mas dependendo dos casos é possível que um dos outros seja mais apropriado.
- Se o código que demora for parte da resolução de um projecto para avaliação, vale a pena perguntar ao professor que tempos são aceitáveis, bem como deixar um comentário ao pé do código a dizer algo como (* atenção, este código demora cerca de 5 minutos *). O mesmo aplica-se em qualquer outra situação na qual mais alguém correrá o código.
- Pôr o computador a carregar tende a ajudar a correr mais depressa (faz uma diferença significativa), bem como evitar processos paralelos.

[1] Um exemplo deste comportamento pode ser observado correndo algo como Table[Timing[AppendTo[a, 0]][[1]], 10000]. A maioria das entradas são 0. (o que não dá muita informação), e de vez em quando podem aparecer valores como 0.015625, devido a efeitos secundários de funções, como por exemplo ser preciso alocar mais memória para guardar a lista.

Append

	Código	AbsoluteTiming[f[10^5]][[1]]

Código lento	`f[n_] := Module[{collatz = {}, k = 1},`
	`  While[k <= n,`
	`   If[EvenQ[k],`
	`    AppendTo[collatz, k/2],`
	`    AppendTo[collatz, 3 k + 1]];`
	`   k = k + 1];`
	`  collatz]`
		14.0631

Código rápido	`f[n_] := Module[`
	`  {collatz = ConstantArray[0, n],`
	`   k = 1},`
	` While[k <= n,`
	`  If[EvenQ[k],`
	`   collatz[[k]] = k/2,`
	`   collatz[[k]] = 3 k + 1];`
	`  k = k + 1];`
	` collatz]`
		0.212185

Código rápido e idiomático	`f[n_] :=`
	`  Table[If[EvenQ[k], k/2, 3 k + 1],`
	`   {k, 1, n}]`
		0.0584041

É comum usar o comando Append ou AppendTo para construir listas, começando com uma lista vazia e adicionando elementos à lista incrementalmente. No entanto, este processo é computacionalmente lento.

Para perceber porquê, é preciso recordar que listas são guardadas de forma contígua na memória do computador. Para adicionar um elemento a uma lista de comprimento n, o computador primeiro verifica que há espaço na memória imediatamente à frente da lista. Se não houver, o computador tem que reservar espaço para uma lista de comprimento $n+1$, tendo no processo que procurar na sua memória por espaço livre, e depois tem de *mover a lista toda para a sua nova localização* de modo a poder adicionar outro elemento. Este processo é imprevisível mas geralmente muito lento.

O código rápido funciona fazendo este passo de reservar apenas uma vez, inicializando um bloco de memória tão grande quanto vai ser necessário logo no começo. Como não estamos a modificar o tamanho da lista, esta não está a ser movida em memória.

O código idiomático é equivalente ao código rápido, mas é muito mais curto e $4\times$ mais rápido. De facto, fazer uma lista com dados elementos é tão comum que os programadores do *Mathematica* implementaram o comando Table especificamente para este propósito, que para além de ser conveniente está bastante otimizado. Por exemplo, em vez de iterar sobre $k = 1, \ldots, n$ como no primeiro código, é possível que o computador paralelize o código, fazendo várias coisas ao mesmo tempo.

Listas com número imprevisível de elementos

O método anterior nem sempre é diretamente aplicável, nomeadamente quando o tamanho da lista não é conhecido à partida, por exemplo se quisermos a lista dos primos até n. No entanto, o comando Nothing é útil para este propósito.

	Código	AbsoluteTiming[f[10^6]][[1]]

Código lento	`f[n_] := Module[{primos = {}, k = 2},`
	`  While[k <= n,`
	`  If[PrimeQ[k],`
	`  AppendTo[primos, k]];`
	`  k = k + 1];`
	`  primos]`
		9.96088

Código rápido e idiomático	`f[n_] :=`
	`  Table[If[PrimeQ[k], k, Nothing],`
	`  {k, 2, n}]`
		0.519021

Impedir Cálculo Simbólico; Comando N

	Código	AbsoluteTiming[f[2 * 10^5]][[1]]

Código lento	`f[n_] := Module[{conv = n, k = n},`
	`  While[k >= 1,`
	`  conv =  k + 1/conv;`
	`  k = k - 1`
	`  ];`
	`  conv`
	`  ]`
		6.10166

Código rápido	`f[n_] := Module[{conv = N[n], k = n},`
	`  While[k >= 1,`
	`  conv =  k + 1/conv;`
	`  k = k - 1`
	`  ];`
	`  conv`
	`  ]`
		1.43313

Quando não lhe é dito para fazer arredondamentos, o Mathematica usa *cálculo simbólico* de modo a manter exatidão. Por exemplo, números racionais são representados não com 'ponto flutuante' como em maior parte das linguagens, mas sim como pares de inteiros. Quando estes inteiros são muito grandes, as contas ficam mais complexas, pelo que o Mathematica demora mais tempo a executá-las. Isto é particularmente notável com código que calcula frações contínuas.

Para fazer o código executar mais rápido, basta dizer ao Mathematica para fazer arredondamentos, sendo que *é preciso que ele faça os arredondamentos ao longo que faz as contas; arredondar apenas no final não traz nenhum benefício*. Note-se que arredondamentos propagam-se: um número arredondado mais um número exato dá um número arredondado. Consequentemente, em código como o da tabela acima, é suficiente arredondar apenas a primeira iterada.

Para arredondar um número pode-se usar o comando N ou, se a iterada inicial for um número específico, basta escrever e.g. `0.' (zero arredondado) em vez de `0' (zero exacto).

Este efeito é ainda mais pronunciado em código que usa números irracionais e frações; a tabela abaixo mostra um exemplo no qual a diferença em tempo de execução é particularmente exagerada.

	Código	AbsoluteTiming[f[10^3]][[1]]

Código lento	`f[n_] := Module[{conv = Pi, k = 1},`
	`  While[k <= n,`
	`  conv =  conv  + 1/conv;`
	`  k = k + 1`
	`  ];`
	`  conv`
	`  ]`
		5.80015

Código rápido	`f[n_] := Module[{conv = N[Pi], k = 1},`
	`  While[k <= n,`
	`  conv =  conv  + 1/conv;`
	`  k = k + 1`
	`  ];`
	`  conv`
	`  ]`
		0.0014725

Impedir Cálculo Simbólico; Integrais e Equações

	Código	AbsoluteTiming[f[3]][[1]]

Código lento	`f[x_] := N[Integrate[`
	`  1/(Sqrt[Sin[t]] + Exp[-t^2]),`
	`  {t, 0, x}]]`
		21.245

Código rápido	`f[x_] := NIntegrate[`
	`  1/(Sqrt[Sin[t]] + Exp[-t^2]),`
	`  {t, 0, x}]`
		0.0026742

Apesar de superficialmente parecerem semelhantes, N[Integrate[...]] e NIntegrate[...] são fundamentalmente diferentes. O primeiro calcula um integral simbolicamente e exactamente e depois fornece uma aproximação numérica do resultado; o segundo calcula uma aproximação numérica desde o início, usando métodos que aprenderás na cadeira de Matemática Computacional, que são extremamente rápidos e eficazes.

Este conselho aplica-se também a diversos outros comandos, como Solve vs. NSolve, DSolve vs. NDSolve, e Maximize vs. NMaximize.

Cuidado com SetDelayed :=

	Código	AbsoluteTiming[f[10^4]][[1]]

Código lento	`h[n_] := Sum[1/k, {k, 1, n}]`
	`f[n_] := Module[{valor},`
	`  valor := N[h[n]];`
	`  Sum[Sin[valor + k], {k, 1, n}]`
	`  ]`
		199.78

Código rápido	`h[n_] := Sum[1/k, {k, 1, n}]`
	`f[n_] := Module[{valor},`
	`  valor = N[h[n]];`
	`  Sum[Sin[valor + k], {k, 1, n}]`
	`  ]`
		0.023845

Código muito rápido
(Ver secção sobre comando N acima)
	`h[n_] := Sum[N[1/k], {k, 1, n}]`
	`f[n_] := Module[{valor},`
	`  valor = N[h[n]];`
	`  Sum[Sin[valor + k], {k, 1, n}]`
	`  ]`
		0.0008715

Poderás ter memorizado a diferença entre = e := como `o primeiro é para variáveis, o segundo é para funções'. No entanto, a diferença é um pouco mais profunda, e poderá levar o teu código a ser muito mais lento.

Enquanto que usar = para definir uma função provavelmente dará erro, usar := para definir uma variável não. No entanto, semanticamente os dois são diferentes: escrever x = expr calcula o valor de expr e faz com que x tome este valor. Pelo outro lado, escrever x := expr faz com que, cada vez que o *Mathematica* precise do valor de expr, o calcule novamente. Na maior parte dos casos isto não faz diferença, mas para ver a distinção em acção, tenta executar o seguinte código: (x := RandomReal[]; {x,x,x}).

Neste último exemplo podemos ver o que está a acontecer no código lento. Em vez de RandomReal[] temos um pedaço de código que demora bastante tempo a executar, e o valor de x está a ser usado várias vezes. Com :=, cada vez que x é usado este código é executado, levando a que este seja executado muitas vezes, enquanto que com = ele corre apenas uma vez.

Compile

	Código	AbsoluteTiming[f[1]][[1]]
Código médio	`g[x_] := Sum[Mod[n x, 1], {n, 1, 10^5}]` `f[a_] := Plot[g[x], {x, 0, a}]`	11.4424

Código $2\times$ mais rápido	`g = Compile[x,`
	`  Sum[Mod[n x, 1], {n, 1, 10^5}]`
	`]`
	`f[a_] := Plot[g[x], {x, 0, a}]`
		6.58374

O comando Compile poderá ocasionalmente acelarar código, mas deve ser usado com cuidado.

Este comando serve como um substituto do comando Function que, em vez de manter os comandos em formato *Mathematica*, os traduz para uma linguagem subjacente, mais rápida mas menos flexivel. Assim sendo, Compile só funciona com funções numéricas, sendo incapaz de manipulações simbólicas ou de fazer gráficos.

Recomenda-se que o utilizador não use Compile no seu código a não ser que ache necessário; uso incorreto poderá levar a código significativamente mais *lento*. Para mais, o efeito de Compile no tempo de execução poderá ser difícil de prever; recomenda-se que o seu uso seja sempre acompanhado de medições de tempo de execução para ter a certeza que o seu efeito é de facto benéfico.