Recursos e alternativas para programação eficiente em R

Prof. Wagner H. Bonat

Laboratório de Estatística e Geoinformação
Departamento de Estatística
Universidade Federal do Paraná

Motivação

Motivação

- R não é uma linguagem rápida em termos de tempo computacional.
- R foi explicitamente desenhado para análise iterativa de dados.
- Fácil para humanos, não para o computador.
- Para a maioria das tarefas do dia-a-dia R é rápido o suficiente.
- Comunidade ampla (em geral programadores não profissionais).
- Diversidade de pacotes extras.
- Tem de tudo em todos os sentidos!

- C++ não é uma linguagem rápida em termos de tempo para programação e curva de aprendizado.
- C++ foi desenhada para ser rápida computacionalmente.
- C++ comunidade ampla e ativa.
- Fácil de obter suporte.
- Linguagem de propósito geral.
- Linguagem copilada → maior tempo de programação.
- Mais dificil de fazer pequenos protótipos de códigos.

Motivação

- **Computação científica** - Estamos interessados em desenvolver algoritmos computacionais com algum tipo de aplicação científica.
- Métodos numéricos
  - Sistemas lineares e não-lineares.
  - Derivação e integração numérica.
  - Otimização (linear, quadrática e não-linear).
  - Equações diferenciais.
- Métodos estatísticos
  - Manipulação de bases de dados.
  - Visualização de dados.
  - Distribuições de probabilidade.
  - Modelagem estatística de forma geral.
- Performance computacional pode se tornar crítica.

Objetivos

- Na programação de métodos científicos temos que lidar com duas técnicas de programação geral:
  1. Encontrar e arrumar erros de codificação (*bugs*).
  2. Encontrar e arrumar gargalos da performance computacional.
- Ferramentas para encontrar erros e medir a performance computacional são essenciais.
- **Objetivos:**
  1. Visitar algumas técnicas de *debug* em `R`.
  2. Visitar algumas ferramentas de *profilling* em `R` e `RStudio`.
  3. Discutir algumas estratégias para melhorar a performance computacional do `R`.

Encontrando erros

Ferramentas para *debug*

- O que fazer quando o R retorna uma mensagem de erro inesperada?
- Quais são as ferramentas para encontrar e arrumar o erro?
- Principal documentação RStudio debugging documentation.
  - `traceback()` função que ajuda a encontrar onde um erro ocorreu.
  - `rlang::with_abort()` e `rlang::last_trace()`.
  - `debug()` e `browser()`.
  - Breakpoints `Shift + F9`.
  - `options(error = recover)`.
- Código R (Script1.R).

Estratégia geral

Finding your bug is a process of confirming the many things that you believe are true — until you find one which is not true. - Norm Matloff

1. Google! Dê uma olhada nos pacotes errorist e searcher.
2. Faça o erro reproduzível.
   - Faça o exemplo ser o menor possível.
   - Desenvolver testes automatizados.
3. Descubra onde o erro está.
   - Adote o método científico.
   - Crie hipóteses, desenhe experimentos e teste.
4. Arrume e teste.
   - Cuidado para não incluir novos erros.
   - Importância de automatizar os testes.

Medindo a performance

Medindo a performance

Programmers waste enormous amounts of time thinking about, or worrying about, the speed of noncritical parts of their programs, and these attempts at efficiency actually have a strong negative impact when debugging and maintenance are considered. - Donald Knuth.

- Para ter um código rápido, primeiro precisamos saber onde ele é lento.
- Identificar os chamados *bottlenecks*.
- *Profilling* - Medir o tempo computacional de cada linha de código.
- Encontrado o ponto crítico, testamos diferentes estratégias.
- Dois pacotes são populares profvis bench.
- Código R (Script1.R).

Melhorando a performance

Melhorando a performance

We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil. Yet we should not pass up our opportunities in that critical 3%. A good programmer will not be lulled into complacency by such reasoning, he will be wise to look carefully at the critical code; but only after that code has been identified. — Donald Knuth

Be pragmatic: don't spend hours of your time to save seconds of computer time. - Hadley Wickham.

- Organize o código para otimizar a performance e evitar *bugs*.
- Estratégia do preguiçoso: A função mais rápida é aquela que trabalha menos.
- Vetorize e evite cópias.
- Troque a linguagem.

Organização do código

- Armadilha da tentativa de melhorar o código:
  1. Código rápido porém incorreto.
  2. Código que você acha que é rápido, mas na verdade tem a mesma performance.
- Como evitar essas armadilhas?
  - Identifique o ponto crítico (*bottleneck*).
  - Esboce um conjunto de possibilidades para melhorar a performance.
  - Escreva cada uma em uma função separada.
  - Gere um exemplo representativo da situação.
  - Use *benchmark* para comparar as estratégias.

Como buscar estratégias de melhorias?

- Verifique opções existentes.
  - Consulte o *CRAN task view*.
  - Consulte as dependencias reversas do pacote Rcpp.
- Procure na literatura termos para descrever o *bottleneck*.
- Pergunte para colegas.
- Use os termos encontrados para procurar no Google e StackOverflow.
- Restrinja sua busca a página relacionadas com o R (https://rseek.org/).
- Selecione as opções que parecem promissoras.
- Benchmark cada uma e tente combiná-las para criar uma melhor.
- Pare assim que o código for rápido o suficiente.

Vetorização

- Vetorizar não é apenas evitar `loops`.
- Pensar no código como uma todo. Vetores ao invés de escalares.
- Em `R` vetorizar é simplesmente encontrar uma função em `C` que faz o que você quer :)
- Alguns exemplos: `rowSums()`, `colSums()`, `rowMeans()`, and `colMeans()`.
- Funções para tarefas específicas tendem a ser mais rápidas do que funções genéricas.
- Álgebra matricial é uma forma de `vetorização`.
- Bibliotecas de álgebra linear como a BLAS são altamente eficientes.

Evite cópias

- Regra número 1: Nunca cresça um objeto!
- Uso de funções como `c()`, `append()`, `cbind()`, `rbind()`, ou `paste()` deve ser feiro com cuidado.
- Estudo de caso: Teste-t.
- Objetivo: Executar 1000 experimentos, cada um coletando amostras de 50 individuos. Os primeiros 25 individuos são designados ao grupo 1 e o resto ao grupo 2. Efetuar um teste t para comparar as médias dos grupos 1 e 2.
- Código R(Script1.R).

Dicas do Hadley para melhorar a sua programação

- Read R blogs to see what performance problems other people have struggled with, and how they have made their code faster.
- Read other R programming books, like The Art of R Programming or Patrick Burns' R Inferno to learn about common traps.
- Take an algorithms and data structure course to learn some well known ways of tackling certain classes of problems. I have heard good things about Princeton's Algorithms course offered on Coursera.
- Learn how to parallelise your code. Two places to start are Parallel R and Parallel Computing for Data Science.
- Read general books about optimisation like Mature optimisation or the Pragmatic Programmer.
- Reescreva suas funções em C++.

Melhorando a performance usando C++

Típica situação

- Código está funcionando perfeitamente.
- Já fez o *profiling* e melhorou os principais *bottleneck*.
- Mas seu código ainda não está rápido o suficiente.
- Reescrever partes importantes do seu código em C++ pode ajudar.
- Típicos *bottleneck*s que valem a pena escrever código em C++:
  - *Loop*'s que não podem ser vetorizados, porque são usados em sequência.
  - Funções recursivas, ou problemas que envolvem chamar funções milhões de vezes.
  - Problemas avançados que requerem estruturas ou algoritmos que o R não tem.
- Página do Rcpp.
- Tutorial de **Rcpp** (Script2.R).

Estudo de caso

- Amostrador de Gibbs (Gibbs sampler) extraído do blog do Dirk.

```r
gibbs_r <- function(N, thin) {
  mat <- matrix(nrow = N, ncol = 2)
  x <- y <- 0
  for (i in 1:N) {
    for (j in 1:thin) {
      x <- rgamma(1, 3, y * y + 4)
      y <- rnorm(1, 1 / (x + 1),
                 1 / sqrt(2 * (x + 1)))
    }
    mat[i, ] <- c(x, y)
  }
  mat
}
```

```cpp
#include <Rcpp.h>
using namespace Rcpp;
// [[Rcpp::export]]
NumericMatrix gibbs_cpp(int N, int thin) {
  NumericMatrix mat(N, 2);
  double x = 0, y = 0;
  for(int i = 0; i < N; i++) {
    for(int j = 0; j < thin; j++) {
      x = rgamma(1, 3, 1 / (y * y + 4))[0];
      y = rnorm(1, 1 / (x + 1),
                1 / sqrt(2 * (x + 1)))[0];
    }
    mat(i, 0) = x;
    mat(i, 1) = y;
  }
  return(mat);
}
```

Mais recursos

- O tutorial apresentou apenas os aspectos básicos do `Rcpp`.
- `vignette("Rcpp-quickref")` é uma excelente referência.
- Página do Rcpp.
- Effective C++ e Effective STL.
- C++ Annotations.
- Algorithm Libraries.
- Pacotes adicionais do ecosistema `Rcpp`: `RcppArmadillo`, `RcppEigen` e `RcppGSL`.

Paralelização

Paralelização em R

- Existem diversos esquemas de paralelização.
- Não vamos entrar em detalhes em nenhum :(
- Loops em R tendem a ser lentos, alternativas
  - Escreva código em C++ através do `Rcpp`.
  - Paralelize o seu for (se ele não for sequencial).
- Pacotes úteis: `foreach`, `parallel` e `doParallel`.
- Exemplos: Script3.R.

Projeto

- Modelos hierárquicos são ferramentas populares em modelagem estatística.
- Considere o seguinte modelo:

$$Y_i|b_i \sim P(\lambda_i),$$
$$b_i \sim N(0, \sigma^2)$$

onde $\lambda_i = \exp\{\beta_0 + b_i\}$, para $i = 1, \ldots, n$.

- Função de verossimilhança é dada por

$$L(\beta_0, \sigma^2) = \prod_{i=1}^{n} \int_{-\infty}^{\infty} \frac{\lambda_i^{y_i} \exp^{-\lambda_i}}{y_i!} \frac{1}{\sqrt{2\pi\sigma^2}\exp\{\frac{1}{2\sigma^2}b_i^2\}} db_i.$$

- A estimativa de máxima verossimilhança é $\underset{\beta_0\sigma^2}{\text{argmax}} \quad L(\beta_0, \sigma^2)$.

Objetivos

- Implementar uma função para simulação deste modelo.
- Implementar a estimação por máxima verossimilhança pontual e intervalar.
- Requisitos
  - Usar qualquer uma das estratégias que eu apresentei aqui.
  - Usar qualquer outra estratégia que você possa encontrar.
  - Deve ser pelo menos mais rápido que a minha função.
  - Apresentação de até 10 minutos no último dia de aula 25/06/2021.
  - Solução reproduzível!
  - Benchmark contra o meu código é obrigatório!!