

Parallel and Cross-Language Computing

A Hands-On Workshop for Empirical Researchers

Nelson Areal 

nareal@eeg.uminho.pt

NIPE/University of Minho

Miguel Portela 

miguel.portela@eeg.uminho.pt

NIPE/University of Minho/Banco de Portugal

December 14, 2025

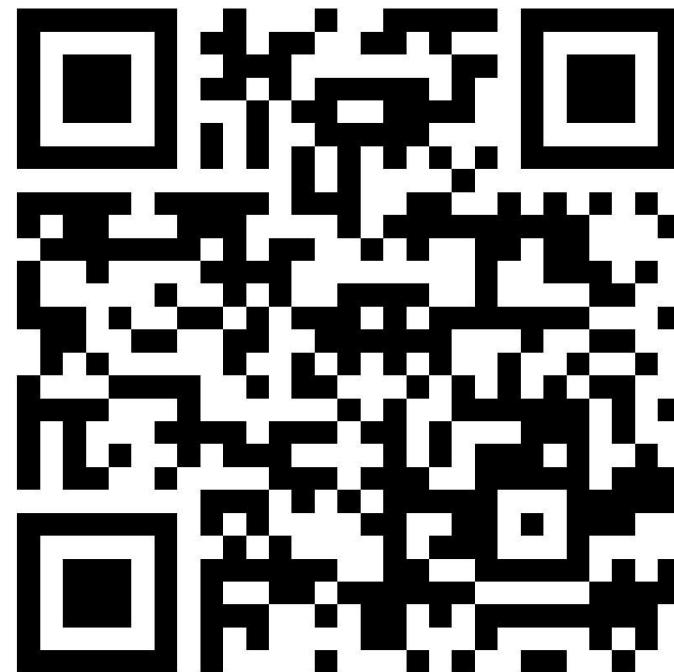


University of Minho

School of Economics, Management and Political Science

Link for the workshop materials

Worksop materials are available at:



https://nareal.github.io/bplim_workshop_2025/



University of Minho

School of Economics, Management and Political Science

Plan for the session

Discuss some strategies to optimize code performance:

- Optimize the algorithm
- Use parallel computing
- Use a lower-level language (C++, Julia, etc.)
- Use the GPU

We are going to illustrate these strategies using two examples.



Some initial thoughts on optimization

*"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. 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%.**"*

Knuth, D. E. (1974). Structured programming with go to statements. ACM Computing Surveys (CSUR), 6(4), 261-301.



Ready made tools

For most data manipulation tasks, there are already great alternatives:

- Apache Arrow
- Polars
- DuckDB
- Apache Spark™

These tools are optimized for performance and can handle large datasets efficiently.

There is no need to reinvent the wheel.



How to identify bottlenecks?

Use **profiling tools** to identify bottlenecks in your code, in R consider using the **profvis** package.

And then focus on optimizing the parts of the code that are most time-consuming.

The strategies we are going to discuss can be applied if the bottleneck is not yet programmed in an optimized way or in parallel.



Two illustrations

We are going to illustrate some of the strategies using two examples:

- American option valuation
- High-dimensional fixed effect models



Valuing an american option



Option valuation with binomial trees

We need the following parameters to value an option using a binomial tree:

- Current underlying asset price: S_0
- Strike price: K
- Time to maturity: T in years
- Risk-free interest rate: r continuously compounded
- Dividend yield: q continuously compounded
- Volatility of the underlying asset: σ
- Number of time steps in the binomial tree: N
- Type of option: Call or Put



Option valuation with binomial trees

Once we have the parameters, we can proceed to build the binomial tree and value the option.

We will be using the CRR (Cox-Ross-Rubinstein) parameters:

- $u = e^{\sigma\sqrt{\Delta t}}$
- $d = e^{-\sigma\sqrt{\Delta t}}$
- $\Delta t = \frac{T}{N}$
- $p = \frac{e^{(r-q)\Delta t} - d}{u - d}$

Option valuation with binomial trees

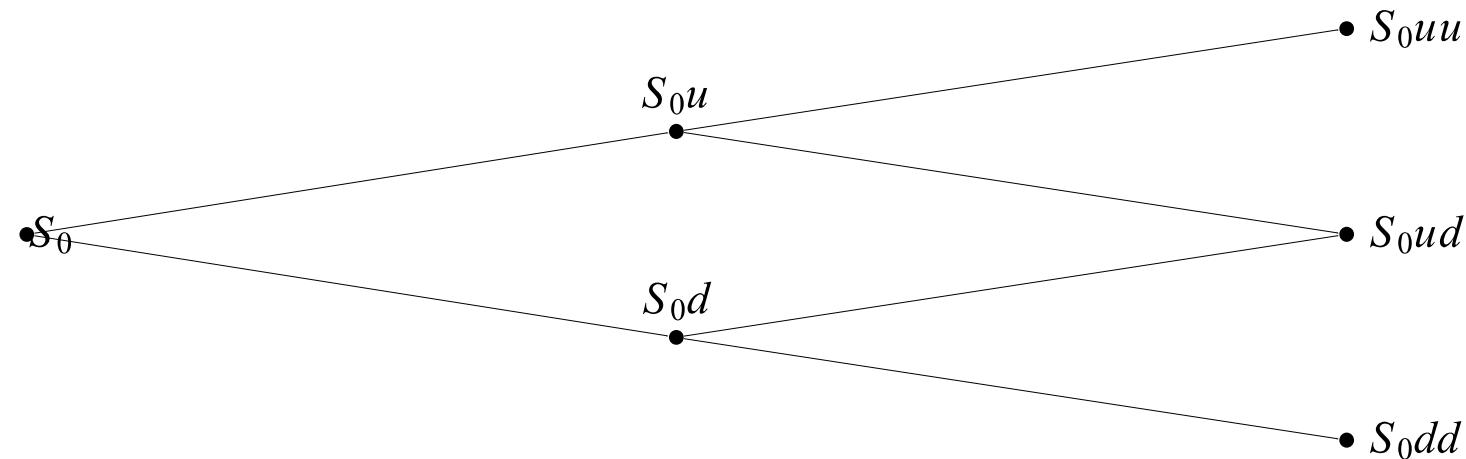
Now we need to follow these steps:

1. Build the underlying asset price tree
2. Calculate the option values at each node, starting from maturity and working backwards to the present.
3. Determine the option price at the initial node.



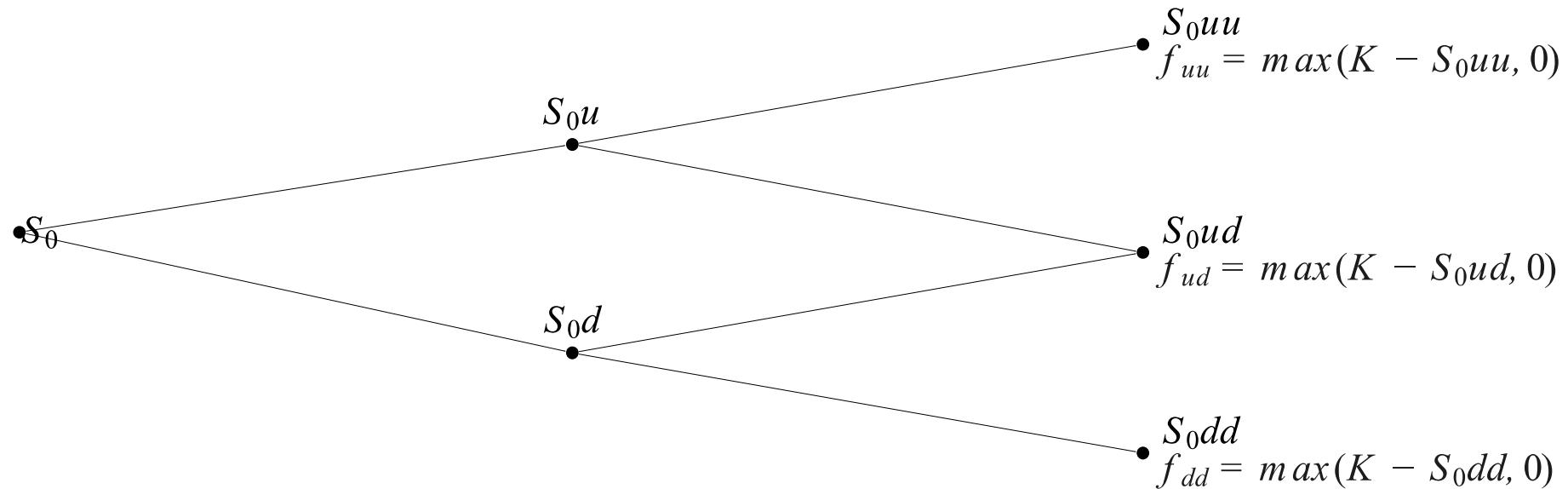
Option valuation with binomial trees

Build the underlying asset price tree



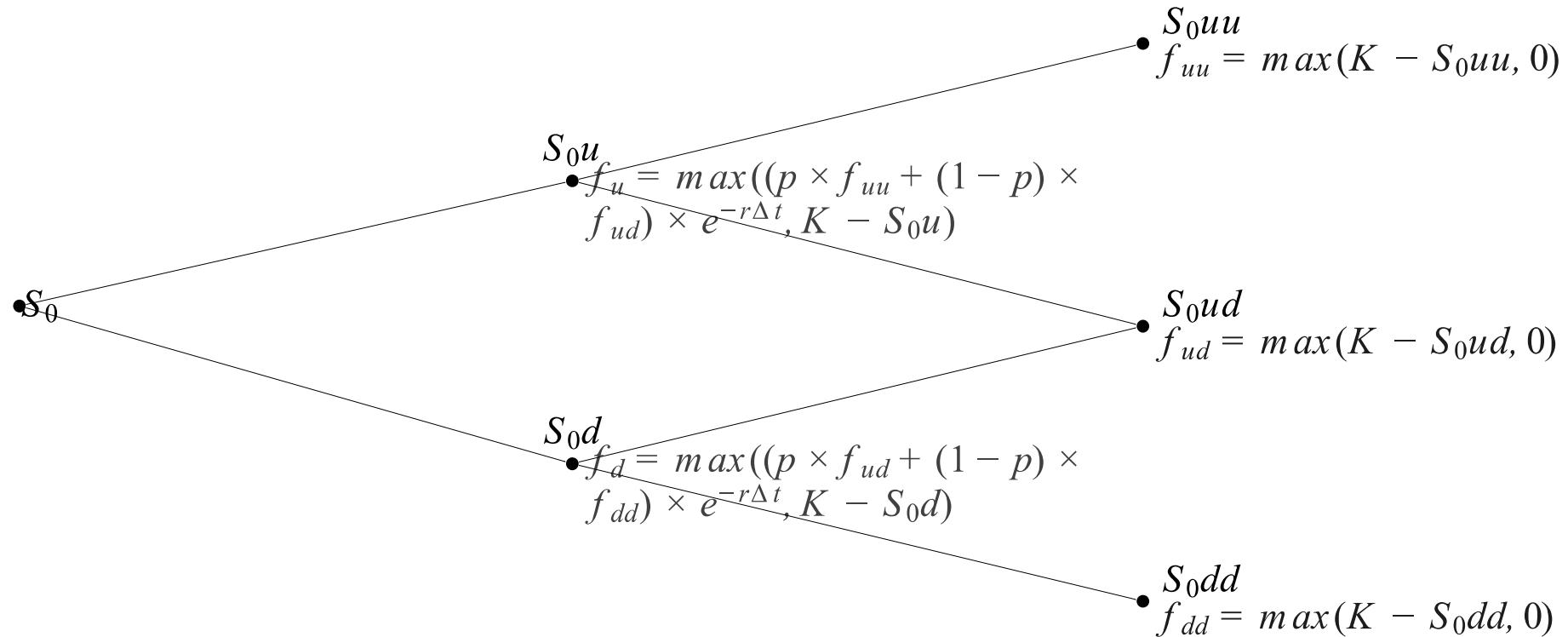
Option valuation with binomial trees

Calculating option values at maturity



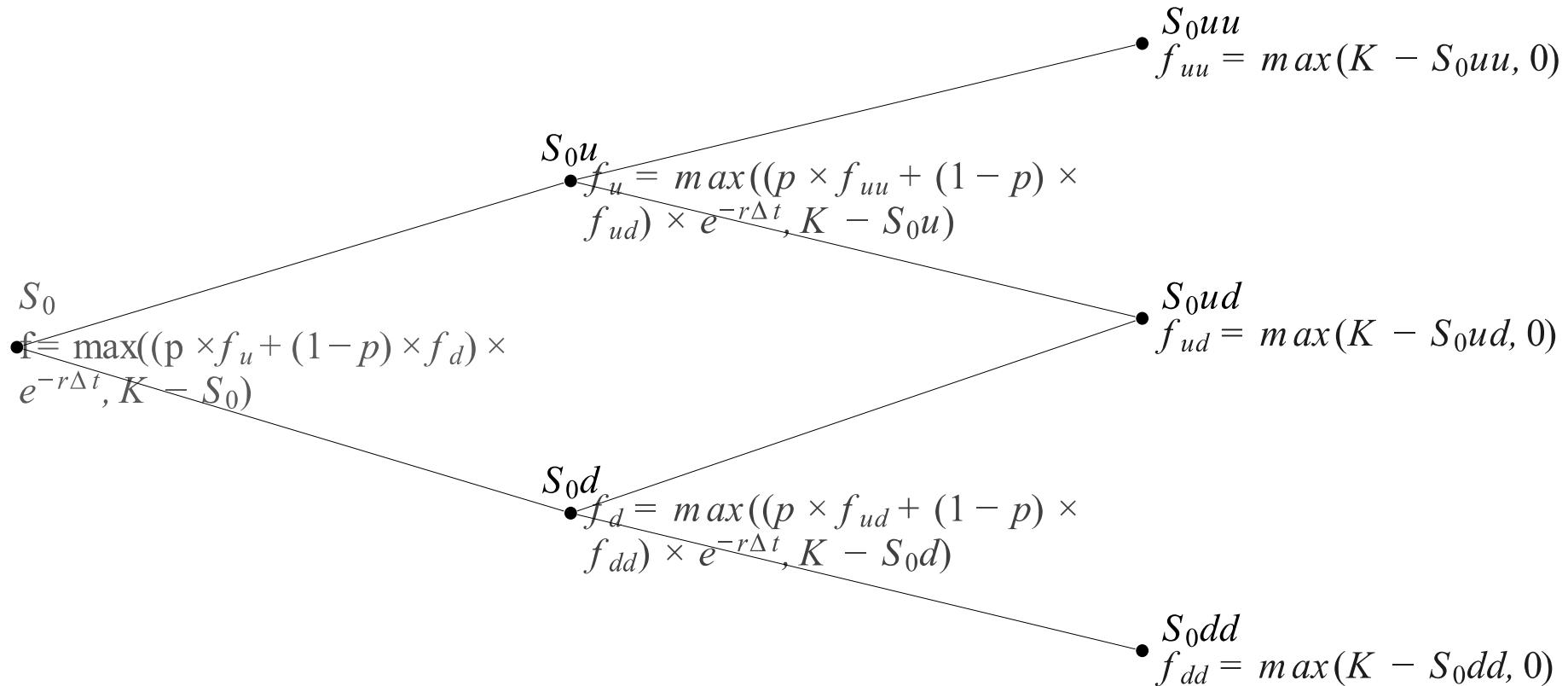
Option valuation with binomial trees

Calculating option values at all previous nodes



Option valuation with binomial trees

Calculating option values at all previous nodes



Binomial trees – Derivatives

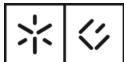
R implementation



R implementation

Naive v1 - Vectors v2 - Vectors v3 - Vectors

```
1 # This function implements a binomial tree algorithm for
2 # valuing American put options using Cox-Ross-Rubinstein parameters.
3
4 ap_naive <- function(S0, K, r, q, tt, sigma, steps) {
5   # S0: Initial stock price
6   # K: Strike price
7   # r: Risk-free interest rate compounded continuously
8   # q: Dividend yield compounded continuously
9   # tt: Time to maturity in years
10  # sigma: Volatility of the underlying stock
11  # steps: Number of time steps in the binomial tree
12
13  dt <- tt / steps # time between steps
14  u <- exp(sigma * sqrt(dt)) # Up factor
15  d <- exp(-sigma * sqrt(dt)) # Down factor
16  p <- (exp((r - q) * dt) - d) / (u - d) # Risk-neutral probability
17
18  # Initialize matrices for asset prices and option values
19  # Rows represent nodes (i), columns represent time steps (j)
20  asset_prices <- matrix(NA, nrow = steps + 1, ncol = steps + 1)
21  option_values <- matrix(NA, nrow = steps + 1, ncol = steps + 1)
22
23  # Fill in asset prices for all nodes in the tree (lower triangular)
24  for (j in 1:(steps + 1)) {
25    for (i in (steps + 2 - j):(steps + 1)) {
26      asset_prices[i, j] <- S0 * u^(steps + 1 - i) * d^(i + j - steps - 2)
27    }
28  }
29
30  # Initialize option values at maturity (last column)
31  for (i in 1:(steps + 1)) {
32    option_values[i, steps + 1] <- max(K - asset_prices[i, steps + 1], 0)
33  }
```



A naive implementation in R

There are other ways that the convergence can be improved:

- Alternative parameters (e.g., Jarrow-Rudd, Tian, Leisen-Reimer)
- Control variates
- Extrapolation techniques (e.g., Richardson extrapolation)

We just want to focus on computational performance improvements.



Benchmarking the R implementations

With only one option:

Code	Results
------	---------

```
1 s <- 40
2 k <- 40
3 v <- 0.30
4 r <- 0.08
5 tt <- 0.25
6 d <- 0
7 nstep <- 1000
8
9 bench_result_one_option_r <- bench::mark(
10   ap_naive(S0 = s, K = k, r = r, q = d, tt = tt, sigma = v, steps = nstep),
11   ap_v1(S0 = s, K = k, r = r, q = d, tt = tt, sigma = v, steps = nstep),
12   ap_v2(S0 = s, K = k, r = r, q = d, tt = tt, sigma = v, steps = nstep),
13   ap_v3(S0 = s, K = k, r = r, q = d, tt = tt, sigma = v, steps = nstep),
14   min_iterations = 10L
15 )
```



Benchmarking the R implementations

With the sample set of 2500 options.

option_set	volatility	time_to_maturity	asset_price	dividend_rate	riskless_rate	exercise_price
1	0.5574030	0.7583443	101.70338	0.035659105	0.09927446	100
2	0.5685377	1.0953070	108.78327	0.020800169	0.00000000	100
3	0.2430698	0.5483695	120.04294	0.010062093	0.09908713	100
4	0.5152238	0.8569535	90.74575	0.013811304	0.08435938	100
5	0.4208728	2.5816131	107.30398	0.002372436	0.09318627	100
6	0.3595480	0.2790196	109.79172	0.038926066	0.04160950	100
7	0.4682942	0.6898661	96.67050	0.010791985	0.08241817	100
8	0.1673333	0.9885690	114.41327	0.088989330	0.07847599	100
9	0.4284961	0.7498327	92.32567	0.029429095	0.00000000	100
10	0.4525324	4.8019876	72.48192	0.099098847	0.05604909	100
11	0.3288709	0.5439403	97.16279	0.041322451	0.09064361	100
12	0.4595561	2.9412671	121.75869	0.041567527	0.09984472	100



Benchmarking the R implementations

With the sample set of 2500 options - Naive

Code

Results

```
1 bench::system_time({  
2   option_values_naive <- option_parameters |>  
3   mutate(  
4     option_value = pmap_dbl(  
5       list(  
6         S0 = asset_price,  
7         K = exercise_price,  
8         r = riskless_rate,  
9         q = dividend_rate,  
10        tt = time_to_maturity,  
11        sigma = volatility,  
12        steps = nstep  
13      ),  
14      ap_naive,  
15      .progress = TRUE  
16    )  
17  )  
18})
```

Benchmarking the R implementations

With the sample set of 2500 options - R v3

Code

Results

```
1 bench::system_time({  
2     option_values_v3 <- option_parameters |>  
3     mutate(  
4         option_value = pmap_dbl(  
5             list(  
6                 S0 = asset_price,  
7                 K = exercise_price,  
8                 r = riskless_rate,  
9                 q = dividend_rate,  
10                tt = time_to_maturity,  
11                sigma = volatility,  
12                steps = nstep  
13            ),  
14            ap_v3,  
15            .progress = TRUE  
16        )  
17    )  
18 })
```

Parallel computing in R

The easiest way to reduce computational time is to use parallel computing.

This can be done in several ways, but one of the most straightforward ways to parallelize code in R is to use the **furrr** package.

The **furrr** package is a parallel implementation of the **purrr** package, which uses the **future** package to provide a simple and consistent API for parallel programming in R.

This allows us to parallelize code on the current machine or on a **cluster of machines**.

Parallel computing in R

Before doing so, it is always a good idea to create **safe versions** of the functions to capture potential errors:

```
1 ap_naive_safe <- safely(ap_naive)
2 ap_v1_safe <- safely(ap_v1)
3 ap_v2_safe <- safely(ap_v2)
4 ap_v3_safe <- safely(ap_v3)
```



Parallel computing in R

Code

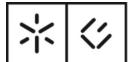
Results - Naive

Results - v1

Results - v2

Results - v3

```
1 plan(multisession, workers = 10)
2 bench::system_time({
3   option_values_v3_par <- option_parameters |>
4     mutate(
5       option_value = future_pmap(
6         list(
7           S0 = asset_price,
8           K = exercise_price,
9           r = riskless_rate,
10          q = dividend_rate,
11          tt = time_to_maturity,
12          sigma = volatility,
13          steps = nstep
14        ),
15        ap_v3_safe,
16        .progress = TRUE
17      )
18    ) |>
19  unnest_wider(option_value, names_sep = "_")
```



Check all implementations return the same results

The results are identical across all implementations:

```
1 identical(  
2   option_values_naive_par,  
3   option_values_v1_par  
4 )  
5  
6 identical(  
7   option_values_v1_par,  
8   option_values_v2_par  
9 )  
10  
11 identical(  
12   option_values_v2_par,  
13   option_values_v3_par  
14 )
```



Improving performance with Rcpp

Another approach to improve performance is to use a lower-level language like C++.

Again, there are several ways to do this, but one of the most straightforward ways is to use the **Rcpp package**.

This package allows us to write C++ code, link and compile and call it as if it were an R function. And it makes it easy to use matrix algebra libraries like **Armadillo** and **Eigen**.

For more details see: **Rcpp**



Improving performance with Rcpp

Using AI to convert R to C++

This example has all the necessary components to use AI tools to convert R code to C++ code:

- an algorithm already implemented in R;
- a large dataset that can serve as a benchmark;
- the new code can be inspected to ensure it is correct (the syntax will not be too different from R).

So we can use an AI coding agent (**claude-code**, **gemini-cli**, **codex**, **Aider**, **Cline**, etc.) to help us convert the code to C++.



Improving performance with Rcpp

Caution when using AI coding agents

- Make sure that you have a comprehensive test suite
- **Make sure that the agent does not change the test suite**
- Use agents in YOLO mode (e.g.: `claude --dangerously-skip-permissions`) at **your own peril!** Best to use a **sandboxed microVM**.



Improving performance with Rcpp

Using claude-code to convert R to C++

```
> claudie
Claude Code v2.0.64 —
Welcome back Nelson!

Sonnet 4.5 · Claude Pro
~/.../aulas/2025-2026/bp/bplim_workshop_2025/R

Tips for getting started
Run /init to create a CLAUDE.md file with instructions for Claude
Recent activity
No recent activity

> Please convert convert the function in @ap_v3.R to Rcpp, the focus is to maintain accuracy and improve performance.
```

Improving performance with Rcpp

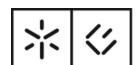
Using claude-code to convert R to C++

The prompt used:

Please convert the function in @ap_v3.R to Rcpp, the focus is to maintain accuracy and improve performance.

was **purposefully kept simple** to see how well the model performs with a minimal prompt.

And yet...



Improving performance with Rcpp

Using claude-code to convert R to C++

Welcome back Nelson!

Sonnet 4.5 · Claude Pro
~/.../aulas/2025-2026/bp/bplim_workshop_2025/R

Tips for getting started
Run /init to create a CLAUDE.md file with instructions for Claude

Recent activity
No recent activity

> Please convert convert the function in @ap_v3.R to Rcpp, the focus is to maintain accuracy and improve performance.
| Read ap_v3.R (136 lines)

- I'll help you convert this American put option pricing function to Rcpp for better performance. Let me first explore the codebase to understand the project structure and then plan the conversion approach.

Enter plan mode?

Claude wants to enter plan mode to explore and design an implementation approach.

In plan mode, Claude will:

- Explore the codebase thoroughly
- Identify existing patterns
- Design an implementation strategy
- Present a plan for your approval

No code changes will be made until you approve the plan.

- > 1. Yes, enter plan mode
2. No, start implementing now



Improving performance with Rcpp

Using claude-code to convert R to C++

Here is Claude's plan:

Implementation Plan: Convert ap_v3 to Rcpp

Overview

Convert the American put option pricing function (ap_v3) to Rcpp for improved performance while maintaining accuracy.
Use split approach: R wrapper for validation + C++ core for computation.

Approach

- Validation: Keep in R wrapper (readable, good error messages, negligible cost)
- Computation: Move to C++ (10-30x speedup on nested loops)
- Integration: Update benchmark notebook to compare performance
- Optimization level: Basic Rcpp translation (no advanced optimizations)

Files to Create

1. /Users/nareal/Documents/work/uminho/aulas/2025-2026/bp/bplim_workshop_2025/R/ap_v3_rcpp.cpp

C++ implementation of the core algorithm:

- Function signature: double ap_v3_cpp(double S0, double K, double r, double q, double tt, double sigma, int steps)
- No input validation (handled by R wrapper)
- Direct translation of algorithm from ap_v3.R lines 92-134
- Key adjustments:
 - Use 0-based indexing for C++ loops
 - Use std::max() instead of R's max()
 - Use pow() for exponentiation
 - NumericVector for option_values storage

2. /Users/nareal/Documents/work/uminho/aulas/2025-2026/bp/bplim_workshop_2025/R/ap_v3_rcpp.R

R wrapper with validation:



Improving performance with Rcpp

Using claude-code to convert R to C++

□ Test compilation and verify correctness

Bash command

```
R --vanilla --quiet -e "
library(Rcpp)
library(here)

# Source the C++ file
cat('Compiling C++ code...\n')
sourceCpp('ap_v3_rcpp.cpp')

# Source the R wrapper and original function
source('ap_v3_rcpp.R')
source('ap_v3.R')

# Test parameters
s <- 40
k <- 40
v <- 0.30
r <- 0.08
tt <- 0.25
d <- 0
nstep <- 1000

cat('\nTesting with single set of parameters...\n')
cat('Parameters: S0=', s, ', K=', k, ', r=', r, ', q=', d, ', tt=', tt, ', sigma=', v, ', steps=', nstep, '\n',
sep='')

# Calculate using both functions
result_v3 <- ap_v3(S0 = s, K = k, r = r, q = d, tt = tt, sigma = v, steps = nstep)
result_rcpp <- ap_v3_rcpp(S0 = s, K = k, r = r, q = d, tt = tt, sigma = v, steps = nstep)
```



Improving performance with Rcpp

Using claude-code to convert R to C++

□ Test compilation and verify correctness

Bash command

```
R --vanilla --quiet -e "
library(Rcpp)
library(bench)

# Source the files
sourceCpp('ap_v3_rcpp.cpp')
source('ap_v3_rcpp.R')
source('ap_v3.R')

# Test parameters
s <- 40
k <- 40
v <- 0.30
r <- 0.08
tt <- 0.25
d <- 0
nstep <- 1000

cat('Benchmarking ap_v3 vs ap_v3_rcpp...\\n\\n')

# Benchmark
bench_result <- bench::mark(
  R = ap_v3($0 = s, K = k, r = r, q = d, tt = tt, sigma = v, steps = nstep),
  Rcpp = ap_v3_rcpp($0 = s, K = k, r = r, q = d, tt = tt, sigma = v, steps = nstep),
  min_iterations = 20L,
  check = FALSE
)
```

Improving performance with Rcpp

Using claude-code to convert R to C++

- Now let me test the compilation and verify correctness:

```
• Bash(R --vanilla --quiet -e " timeout: 1m 0s
    library(Rcpp)...)
[>
> library(Rcpp)
> library(here)
... +93 lines (ctrl+o to expand)
```

- Perfect! Now let me run a quick benchmark to demonstrate the performance improvement:

```
• Bash(R --vanilla --quiet -e "
    library(Rcpp)...
[>
> library(Rcpp)
> library(bench)
... +49 lines (ctrl+o to expand)
```

∴ Thinking...

· Beboppin'... (esc to interrupt • 7m 49s • ↓ 7.1k tokens)

> █

↳ accept edits on (shift+tab to cycle)

Improving performance with Rcpp

R wrapper

C++ code

Source

Benchmark with one option

```
1 # This function wraps the C++ implementation (ap_v3_cpp) with R-based input
2 # validation. It provides the same interface as ap_v3 with improved performance.
3 # The validation code is identical to ap_v3 to ensure the same error checking.
4
5 ap_v3_rcpp <- function(S0, K, r, q, tt, sigma, steps) {
6   # S0: Initial stock price
7   # K: Strike price
8   # r: Risk-free interest rate compounded continuously
9   # q: Dividend yield compounded continuously
10  # tt: Time to maturity in years
11  # sigma: Volatility of the underlying stock
12  # steps: Number of time steps in the binomial tree
13
14  # Input validation
15  # Check for missing arguments
16  if (
17    missing(S0) ||
18    missing(K) ||
19    missing(r) ||
```



Improving performance with Rcpp

Benchmark with all options

Code

Benchmark results

```
1 rcpp_v3_time_all_options <- bench::system_time({  
2   option_values_rcpp <- option_parameters |>  
3   mutate(  
4     option_value = pmap_dbl(  
5       list(  
6         S0 = asset_price,  
7         K = exercise_price,  
8         r = riskless_rate,  
9         q = dividend_rate,  
10        tt = time_to_maturity,  
11        sigma = volatility,  
12        steps = nstep  
13      ),  
14      ap_v3_rcpp,  
15      .progress = TRUE  
16    )  
17  )  
18})
```

Improving performance with Rcpp

Parallelization

Since the Rcpp function creates an external pointer to another object in C++, we can only use it in R session where it was created.

So we need to modify the R wrapper function to create the external pointer inside the function. This is what the **ap_v3_rcpp_par** is doing.



Improving performance with Rcpp

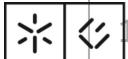
Parallelization results - all options

R wrapper

Results

Results with package

```
1 plan(multisession, workers = 10)
2 rcpp_v3_par_time_all_options <- bench::system_time({
3   option_values_rcpp_par <- option_parameters |>
4     mutate(
5       option_value = future_pmap(
6         list(
7           S0 = asset_price,
8           K = exercise_price,
9           r = riskless_rate,
10          q = dividend_rate,
11          tt = time_to_maturity,
12          sigma = volatility,
13          steps = nstep
14        ),
15        ap_v3_rcpp_par_safe,
16        .progress = TRUE,
17        .options = furrr_options(seed = TRUE)
18      )
19    }
```



Improving performance with Julia

An alternative to C++ is to use **Julia**, a high-level, high-performance programming language for technical computing.

We will be using **JuliaCall** that allows for a seamless integration between R and Julia.



Improving performance with Julia

Using Claude-code to convert R to Julia

```
> I want to convert the @ap_v3.R function to Julia and call it in R using the JuliaCall R package. The focus is to  
maintain accuracy and improve performance. █
```

‣ accept edits on (shift+tab to cycle)

ctrl-g to edit prompt in Zed

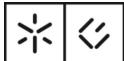
Improving performance with Julia

R wrapper

Julia code

Benchmark with one option

```
1 # R wrapper for Julia implementation of American put option pricing
2 # Uses JuliaCall package to interface with Julia
3
4 # Load required package
5 if (!require("JuliaCall", quietly = TRUE)) {
6   stop(
7     "JuliaCall package is required. Install it with: install.packages('JuliaCall')"
8   )
9 }
10
11 # Initialize Julia (only needs to be done once per session)
12 # This function will set up the Julia connection
13 init_julia_ap <- function(num_threads = "auto") {
14   # Set the number of Julia threads before initialization
15   # Options: "auto" for all cores, or a specific number like "8"
16   Sys.setenv(JULIA_NUM_THREADS = as.character(num_threads))
17
18   # Initialize Julia
19   julia_setup()
```



Improving performance with Julia

Benchmark with all options

Code

Benchmark results

```
1 julia_v3_time_all_options <- bench::system_time({  
2   option_values_julia <- option_parameters |>  
3   mutate(  
4     option_value = pmap_dbl(  
5       list(  
6         S0 = asset_price,  
7         K = exercise_price,  
8         r = riskless_rate,  
9         q = dividend_rate,  
10        tt = time_to_maturity,  
11        sigma = volatility,  
12        steps = nstep  
13      ),  
14      ap_v3_julia,  
15      .progress = TRUE  
16    )  
17  )  
18})
```



Improving performance with Julia

Parallelization

The Julia runtime needs to be started in each worker so the R wrapper function needs to include that step.

Or we can do the parallelization directly in Julia.



Improving performance with Rcpp

Parallelization results - all options

R code

R wrapper

Benchmark results

Parallelization in Julia

```
1 plan(multisession, workers = 8)
2 julia_v3_par_time_all_options <- bench::system_time({
3   option_values_julia_par <- option_parameters |>
4     mutate(
5       option_value = future_pmap(
6         list(
7           S0 = asset_price,
8           K = exercise_price,
9           r = riskless_rate,
10          q = dividend_rate,
11          tt = time_to_maturity,
12          sigma = volatility,
13          steps = nstep
14        ),
15        ap_v3_julia_auto_safe,
16        .progress = TRUE,
17        .options = furrr_options(seed = TRUE)
18      )
19    }
```

Using torch for GPU computing

Torch is a scientific computing framework with wide support for machine learning algorithms and GPU computing.

PyTorch is the Python implementation and **torch for R** that mirrors PyTorch in R.

The advantage of using torch is that it provides a simple way to move data and computations to the GPU, it has the advantage of running on different hardware.



Using torch for GPU computing

There are many problems that can benefit from GPU computing, but not all.

And the transition is not always straightforward. The american option pricing problem is a good example of this. See: **Chapter 45. Options Pricing on the GPU | NVIDIA Developer**

Also consider numerical accuracy when using GPUs, as they may use lower precision arithmetic, the memory that is available, and the overhead of moving data between the CPU and GPU. See: **Numerical accuracy – PyTorch 2.9 documentation**



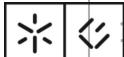
Using torch for GPU computing

Parallelization results - all options

R code

Benchmark results

```
1 # Load required package
2 if (!require("torch", quietly = TRUE)) {
3   stop(
4     "torch package is required. Install it with: install.packages('torch')"
5   )
6 }
7
8 #' Price multiple American put options using a batched binomial tree on a GPU.
9 #
10 #' This function extends the scalar CRR binomial tree implementation so that many
11 #' put option contracts can be priced simultaneously. The binomial time recursion
12 #' remains sequential, but each level operates on the entire batch, allowing the
13 #' GPU to process wide workloads efficiently. All contracts share the same
14 #' number of time steps \code{steps}; other parameters may vary by contract.
15 #
16 #' @param S0 Numeric vector of initial stock prices.
17 #' @param K Numeric vector of strike prices.
18 #' @param r Numeric vector of continuously compounded risk-free rates.
19 #' @param q Numeric vector of dividend yields compounded continuously.
```



Summary

Results for the calculation of 2500 american put options in the test sample:

approach	speedup	total time
Naive R	1.000000	10.09m
V3 R	1.523794	6.62m
V3 R parallel	12.357288	48.97s
V3 Rcpp parallel	86.267275	7.01s
V3 Rcpp parallel (package)	208.490073	2.9s
V3 Julia (parallelization in R)	51.543424	11.74s
V3 Julia (parallelization in Julia)	170.829701	3.54s

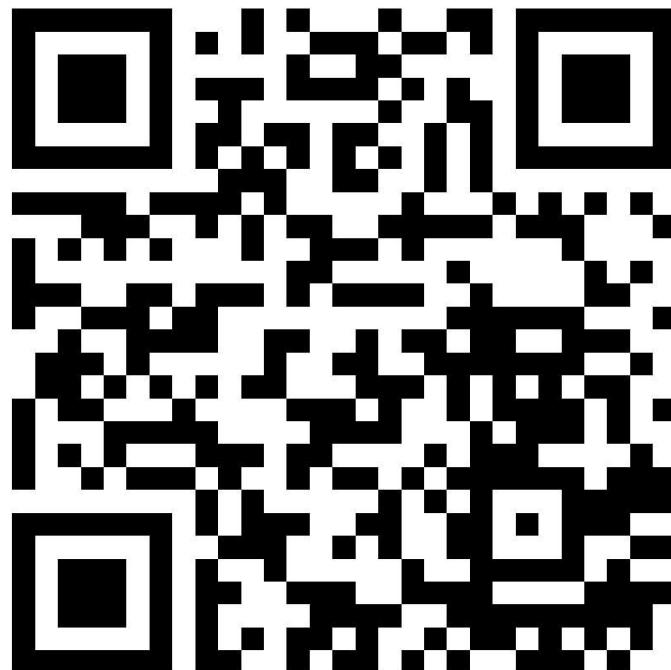


HDFE regression



HDFE regression

Worksop materials for this part are available at:



<https://github.com/reisportela/cp2hdfe>



University of Minho

School of Economics, Management and Political Science