Apêndice F:

Código do aplicativo Simplex Gráfico v.1.0.0

Data de Publicação: 20 de março de 2025

```
library(shiny)
library(ggplot2)
library(DT)

ui <- fluidPage(
  tags$head(

    tags$style(HTML("
      @import url('https://fonts.googleapis.com/css2?family=Yusei+Magic&display=swap');
      body {
        background-color: #b3b3ff;
        color: black;
      }
      h2 {
        font-family: 'Yusei Magic', sans-serif;
      }
      .shiny-input-container {
        color: #474747;
      }"))
  ),
  titlePanel("Solução Gráfica"),
   fluidRow(column(12, h6("Desenvolvido por: Luciane Ferreira Alcoforado-AFA"))),  # Autoria do aplicati
  sidebarLayout(
    sidebarPanel(
      selectInput("obj", "Objetivo:", c("max", "min")),
      textInput("cost_vec", "Vetor custo:", "-5,1"),
      textInput("R1", "R1:", "1,1,10"),
      textInput("R2", "R2:", "-5,1,-10"),
      selectInput("dir_1", "Direção 1:", c("<=", ">=", "=")),
      selectInput("dir_2", "Direção 2:", c("<=", ">=", "=")),
      numericInput("x_lim", "Limite do eixo x:", 10),
      numericInput("y_lim", "Limite do eixo y:", 10),
      width = 2  # Defina a largura desejada
    ),

    mainPanel(
      tabsetPanel(
        # Primeira aba
        tabPanel("Gráfico",
          h3("Modelo"),
          uiOutput("output"),
      plotOutput("output2"),
      # tableOutput("table"),
      h4("Vértices"),
      DT::dataTableOutput("table")
        ),
      # Segunda aba
        tabPanel("Teste",
          h3("Observe o modelo, o gráfico e identifique o vértice ótimo."),
          plotOutput("output3" , width = "300px", height = "200px"),
          radioButtons("respostamarcadavertice",
# adiciona botões de rádio com o ID "respostamarcadavertice"
"Radio Buttons input:",
```

```
# adiciona um rótulo para os botões de rádio
c("label 1" = "option1",
# define as opções e os rótulos dos botões de rádio
"label 2" = "option2"),
selected = NA), # define nenhuma opção como selecionada por padrão
h4(verbatimTextOutput("respostavertice")),
# adiciona uma saída de texto verbatim com o ID "resposta" e um título h4
h4(verbatimTextOutput("resultadovertice")),
h4(verbatimTextOutput("resultadovertice1")),

        ) #fecha aba do teste
      )

    )

  )
)

server <- function(input, output, session) {
# Código para processar os dados com base nas entradas
 # R1 <- reactive({as.numeric(strsplit(input$R1, ",")[[1]])})
 # R2 <- reactive({as.numeric(strsplit(input$R2, ",")[[1]])})
# Uso da função debounce para esperar a digitação do usuário
#  R1 <- reactive({ input_R1 <- input$R1; if (grepl(",", input_R1)) { values <- tryCatch(as.numeric(strs
# R2 <- reactive({ input_R2 <- input$R2; if (grepl(",", input_R2)) { values <- tryCatch(as.numeric(strsp

# Função auxiliar para verificar se o valor é válido
  is_valid_value <- function(value) {
    return(!any(is.na(value) | is.infinite(value)))
  }
 # Uso da função debounce para esperar a digitação do usuário
# Variáveis reativas para armazenar valores anteriores
  prev_R1 <- reactiveVal(c(0, 0, 0))
  prev_R2 <- reactiveVal(c(0, 0, 0))
  prev_cost_vec <- reactiveVal(c(0, 0))

  # Uso da função debounce para esperar a digitação do usuário
  R1 <- debounce(reactive({
    input_R1 <- input$R1
    if (grepl(",", input_R1)) {
      values <- tryCatch(as.numeric(strsplit(input_R1, ",")[[1]]), error = function(e) NULL)
      if (!is.null(values) && length(values) == 3 && all(is.finite(values))) {
        prev_R1(values)
        return(values)
      } else {
        return(prev_R1())
      }
    } else {
      return(prev_R1())
    }
  }), millis = 1000)  # Adicione o argumento millis para definir o tempo de espera em milissegundos

  R2 <- debounce(reactive({
    input_R2 <- input$R2
    if (grepl(",", input_R2)) {
      values <- tryCatch(as.numeric(strsplit(input_R2, ",")[[1]]), error = function(e) NULL)
      if (!is.null(values) && length(values) == 3 && all(is.finite(values))) {
        prev_R2(values)
        return(values)
```

```
      } else {
        return(prev_R2())
      }
    } else {
      return(prev_R2())
    }
  }), millis = 1000)  # Adicione o argumento millis para definir o tempo de espera em milissegundos
  dir_1 <- reactive({input$dir_1})
  dir_2 <- reactive({input$dir_2})
  obj <- reactive({input$obj})

#  cost_vec <- reactive({as.numeric(strsplit(input$cost_vec, ",")[[1]])})
cost_vec <- debounce(reactive({
    input_cost_vec <- input$cost_vec
    if (grepl(",", input_cost_vec)) {
      values <- tryCatch(as.numeric(strsplit(input_cost_vec, ",")[[1]]), error = function(e) NULL)
      if (!is.null(values) && length(values) == 2 && all(is.finite(values))) {
        prev_cost_vec(values)
        return(values)
      } else {
        return(prev_cost_vec())
      }
    } else {
      return(prev_cost_vec())
    }
  }), millis = 1000)  # Adicione o argumento millis para definir o tempo de espera em milissegundos

lim_x=reactive({input$x_lim})
lim_y=reactive({input$y_lim})

#Cálculo dos vértices viáveis
# Criação de um objeto reactiveValues para armazenar tabela1
rv <- reactiveValues()

# Criação da matriz reativa
rv$matriz <- reactive({
  rbind(R1(), R2(), R3=c(1,0,0), R4=c(0,1,0))
})


# Criação da matriz tabela1 reativa
rv$tabela1 = reactive({
  matriz <- rv$matriz()

intersecao <- c()
for (i in 1:(nrow(matriz)-1)) {
 for (j in (i+1):nrow(matriz)) {
 det <- det(matrix(c(matriz[i,1], matriz[i,2], matriz[j,1], matriz[j,2]), ncol = 2))
 if (det != 0) {
 x1 <- det(matrix(c(matriz[i,3], matriz[i,2], matriz[j,3], matriz[j,2]), ncol = 2)) / det
 x2 <- det(matrix(c(matriz[i,1], matriz[i,3], matriz[j,1], matriz[j,3]), ncol = 2)) / det
 intersecao <- c(intersecao, c(x1,x2))
 } } }
intersecao <- round(intersecao,3)
tabela <- data.frame(cbind(intersecao[seq(1, length(intersecao), 2)], intersecao[seq(2, length(intersecao
# Filter rows with positive or null values in both columns
tabela <- round(tabela[tabela$X1 >= 0 & tabela$X2 >= 0, ],3)
rownames(tabela) <- NULL
```

```
# Multiplicar os valores de cada coluna por c1 e c2 respectivamente para obter o valor de z.
  tabela$z <- round(tabela$X1 * cost_vec()[1] + tabela$X2 * cost_vec()[2],2)

# Retornar tabela
  tabela

})

# Criação das opções para o botão
observe({
  # Verifique se tabela1 existe e não está vazia
  if (!is.null(rv$tabela1()[,1:2]) && nrow(rv$tabela1()[,1:2]) > 0) {
    # Obtenha as linhas de tabela1 como uma lista de strings
    options <- apply(rv$tabela1()[,1:2], 1, function(row) paste(row, collapse = ", "))
  }
    # Adicione as opções extras
    options <- c(unique(options), "não há solução ótima")

    # Atualize as opções do botão
updateRadioButtons(session, "respostamarcadavertice",
# atualiza os botões de rádio com o ID "respostamarcadavertice"
label = paste0(c_label, ", Marque sua resposta. Pode ter mais de uma resposta correta!"),
# define o rótulo dos botões de rádio usando a variável c_label
choices = options,
# define as opções dos botões de rádio usando a variável options
selected = character(0)
# define nenhuma opção como selecionada por padrão
)
})



# Criação de um identificador de infinitas soluções ótimas.

infinitassolucoesotimas <- reactive({
r1 = R1()[1:2] #  coeficientes da restrição 1
r2 = R2()[1:2] #  coeficientes da restrição 2
vec_custo = cost_vec() # coeficientes da função objetivo

# Calculando as inclinações
inclinação_R1 = -r1[1] / r1[2]
inclinação_R2 = -r2[1] / r2[2]
inclinação_vec_custo = -vec_custo[1] / vec_custo[2]

# Verificando o paralelismo
if (inclinação_R1 == inclinação_vec_custo) {
  return(" A restrição 1 é paralela à função objetivo.")
}

if (inclinação_R2 == inclinação_vec_custo) {
  return(" A restrição 2 é paralela à função objetivo.")
}else {return(" Nenhuma restrição é paralela à função objetivo.")}

})

# Criação do objeto respostacertavertice
respostacertavertice <- reactive({
  # Definindo a matriz A, a tabela T, o vetor de desigualdades D e o vetor b
  A <- rv$matriz()[,1:2]
```

```
T <- rv$tabela1()[,1:2] #fornece as coordenadas dos pontos de intersecção
T<-rbind(T,c(20000,20000)) #acrescentar valor infinito 20000
D <- c(dir_1(), dir_2())
b <- rv$matriz()[,3]
objetivo <- obj()
custo <- cost_vec()
#lpsolve<- lpSolve::lp(direction = objetivo, objective.in = custo,  const.mat = A, const.dir = D, cons

# Inicializando o vetor resultado
resultado <- rep(TRUE, nrow(T))

# Multiplicando a matriz A por cada linha da tabela T e testando se atende às desigualdades em D
for (i in 1:nrow(T)) {
  produto <- A %*% as.numeric(T[i, ])
  # produto é dataframe com 2l e 1c, cada linha representa o lado esquerdo da restrição 1 e 2

  # Testando se o produto atende às desigualdades em D
  if (D[1] == "=") {
    resultado[i] <- resultado[i] && (round(produto[1],2) == b[1])
  } else if (D[1] == "<=") {
    resultado[i] <- resultado[i] && (round(produto[1],2) <= b[1])
  } else if (D[1] == ">=") {
    resultado[i] <- resultado[i] && (round(produto[1],2) >= b[1])
  }

  if (D[2] == "=") {
    resultado[i] <- resultado[i] && (round(produto[2],2) == b[2])
  } else if (D[2] == "<=") {
    resultado[i] <- resultado[i] && (round(produto[2],2) <= b[2])
  } else if (D[2] == ">=") {
    resultado[i] <- resultado[i] && (round(produto[2],2) >= b[2])
  }
}

# Verifique quais linhas de tabela 1 possuem valores viáveis
valid_rows <- which(resultado == TRUE)

# Se valid_rows estiver vazio, então não há solução ótima
if (length(valid_rows) == 0) {
  return("não há solução ótima")
} else {

# Crie a tabela reduzida
reduced_tabela1 <- rv$tabela1()[valid_rows, ]

#parametros para verificar solucao ilimitada
 x_inf=20000; y_inf=20000
 z_inf=custo[1]*x_inf+custo[2]*y_inf

 if (objetivo == "min" & length(valid_rows) == 1 )

if (objetivo == "min") {
  #No caso de solução ilimitada, caso haja apenas um valor viável
  #ele será de max ou min. Necessário testar em qual situação ele está.
  if (length(valid_rows) == 1) {
      if(reduced_tabela1$z>z_inf){
  return("não há solução ótima")}else{resposta<-reduced_tabela1[,1:2]}
 } else {
  # Se o objetivo for min, retorne as linhas que correspondem ao min(reduced_tabela1$z)
```

```
    resposta <- reduced_tabela1[which(reduced_tabela1$z == min(reduced_tabela1$z)),1:2]
    #resposta terá duas posição, tem que verificar se z_inf é menor do que o z
    #caso seja return("não há solução ótima")
    if(all(reduced_tabela1$z[which(reduced_tabela1$z == min(reduced_tabela1$z)),1:2])>z_inf){
      return("não há solução ótima")}
     }
  }
  if (objetivo == "max") {
    # Caso contrário, retorne as linhas que correspondem ao max(reduced_tabela1$z)
    resposta <- reduced_tabela1[which(reduced_tabela1$z == max(reduced_tabela1$z)),1:2]
  }
  if (resposta == "não há solução ótima"){resultado<-resposta}else{
  # Cole as duas colunas da resposta como um texto único
  linhas = sapply(1:nrow(resposta), function(i) paste(resposta[i,], collapse = ", "))
  resultado = paste(linhas, collapse = " e ")}
  return(resultado)
  }
})


calc_intercept <- function(z) { return(z / c2) }
c_label=c("Usuário") #atualizar com input$nome

# Mostra o modelo PPL
  output$output <- renderUI({
    HTML(paste0("<p><strong>", obj(), "</strong> z = ", cost_vec()[1], "x<sub>1</sub> + ", cost_vec()[2]
                "<p><strong>sujeito a</strong></p>",
                "<p>R<sub>1</sub>: ", R1()[1], "x<sub>1</sub> + ", R1()[2], "x<sub>2</sub> ", dir_1(), "
                "<p>R<sub>2</sub>: ", R2()[1], "x<sub>1</sub> + ", R2()[2], "x<sub>2</sub> ", dir_2(), "
                "<p>x<sub>1</sub>, x<sub>2</sub>   0</p>"))
  })
# Mostra a representação gráfica do modelo
output$output2 <- renderPlot({

c1 = cost_vec()[1]
c2 = cost_vec()[2]
x_vals <- seq(0, lim_x(), by = 0.05) # Valores de x
y_vals <- seq(0, lim_y(), by = 0.05) # Valores de y
data <- expand.grid(x = x_vals, y = y_vals)

lim_z = c1*lim_x()[1]+c2*lim_y()[1]
z_data <- data.frame(z = seq(0,lim_z, by = lim_z/10))
#z_data$intercept <- sapply(z_data$z, calc_intercept)
z_data$intercept <- z_data$z/cost_vec()[2]

switch(input$dir_1,
           "="  = {data$ineq1 <- (R1()[1] * data$x + R1()[2] * data$y) == R1()[3]},
           "<=" = {data$ineq1 <- (R1()[1] * data$x + R1()[2] * data$y) <= R1()[3]},
           ">=" = {data$ineq1 <- (R1()[1] * data$x + R1()[2] * data$y) >= R1()[3]}
    )
switch(input$dir_2,
           "="  = {data$ineq2 <- (R2()[1] * data$x + R2()[2] * data$y) == R2()[3]},
           "<=" = {data$ineq2 <- (R2()[1] * data$x + R2()[2] * data$y) <= R2()[3]},
           ">=" = {data$ineq2 <- (R2()[1] * data$x + R2()[2] * data$y) >= R2()[3]}
    )

    # Plot lines and intersection points
    p=ggplot(data, aes(x = x, y = y)) +
  geom_tile(aes(fill = ineq1), alpha = 0.5) +
```

```
  geom_tile(aes(fill = ineq2), alpha = 0.35) +
  scale_fill_manual(values = c("white", "blue"),
                    name = "Região Viável") +
  coord_cartesian(xlim = c(0, lim_x()), ylim = c(0, lim_y())) +
  labs(title = "", x = "x", y = "y") +
    theme_minimal()

p + geom_segment(aes(x = 0, y = 0, xend = c1/sqrt(c1^2+c2^2), yend = c2/sqrt(c1^2+c2^2)),
                 arrow = arrow(length = unit(0.3, "cm")), color = "red")+
  geom_abline(data = z_data, aes(intercept = intercept,
                                 slope = -c1/c2),
              linetype = "dashed", color = "yellow")+
coord_fixed(ylim = c(0, lim_x()))
  })

# Mostra no teste a representação gráfica em tamanho reduzido
output$output3 <- renderPlot({
    # Convert input values to numeric
    r1 <- as.numeric(strsplit(input$R1, ",")[[1]])
    r2 <- as.numeric(strsplit(input$R2, ",")[[1]])
    cost_vec <- as.numeric(strsplit(input$cost_vec, ",")[[1]])

lim_x=input$x_lim
lim_y=input$y_lim

x_vals <- seq(0, lim_x, by = 0.05) # Valores de x
y_vals <- seq(0, lim_y, by = 0.05) # Valores de y
data <- expand.grid(x = x_vals, y = y_vals)

switch(input$dir_1,
           "="  = {data$ineq1 <- (r1[1] * data$x + r1[2] * data$y) == r1[3]},
           "<=" = {data$ineq1 <- (r1[1] * data$x + r1[2] * data$y) <= r1[3]},
           ">=" = {data$ineq1 <- (r1[1] * data$x + r1[2] * data$y) >= r1[3]}
    )
switch(input$dir_2,
           "="  = {data$ineq2 <- (r2[1] * data$x + r2[2] * data$y) == r2[3]},
           "<=" = {data$ineq2 <- (r2[1] * data$x + r2[2] * data$y) <= r2[3]},
           ">=" = {data$ineq2 <- (r2[1] * data$x + r2[2] * data$y) >= r2[3]}
    )

c1 = cost_vec[1]
c2 = cost_vec[2]
lim_z = c1*lim_x+c2*lim_y
z_data <- data.frame(z = seq(0,lim_z, by = lim_z/10))
calc_intercept <- function(z) { return(z / c2) }
z_data$intercept <- sapply(z_data$z, calc_intercept)


    # Plot lines and intersection points
    p=ggplot(data, aes(x = x, y = y)) +
  geom_tile(aes(fill = ineq1), alpha = 0.5) +
  geom_tile(aes(fill = ineq2), alpha = 0.35) +
  scale_fill_manual(values = c("white", "blue"),
                    name = "Região Viável") +
  coord_cartesian(xlim = c(0, lim_x), ylim = c(0, lim_y)) +
  labs(title = "", x = "x", y = "y") +
    theme_minimal()

p + geom_segment(aes(x = 0, y = 0, xend = c1/sqrt(c1^2+c2^2), yend = c2/sqrt(c1^2+c2^2)),
```

```
                arrow = arrow(length = unit(0.3, "cm")), color = "red")+
  geom_abline(data = z_data, aes(intercept = intercept,
                                 slope = -c1/c2),
              linetype = "dashed", color = "yellow")+
coord_fixed(ylim = c(0, lim_x))
})

# Exibe a tabela com os vértices e o valor de z, nem todo vértice é viável para o PPL.
output$table <- DT::renderDataTable({
  DT::datatable(rv$tabela1())
})

#Exibe a resposta marcada pelo usuário no teste
output$respostavertice <- renderText({ paste0(input$nome,
", sua resposta é: ", input$respostamarcadavertice) })

# Exibição condicional de texto com base em input$respostamarcadavertice e respostacertavertice
output$resultadovertice <- renderText({
  partes = strsplit(respostacertavertice(), " e ")[[1]]
  # Verifique se a resposta do usuário é igual à resposta certa
  if (input$respostamarcadavertice %in% partes) {
    # Se a resposta do usuário for igual à resposta certa
    paste0(input$nome, ", sua resposta está correta!")
    # Retorna um texto informando que a resposta está correta
  } else {
    # Caso contrário
    paste0(input$nome, ", infelizmente sua resposta está incorreta.")
    # Retorna um texto informando que a resposta está incorreta
  }
})

# Retorna um texto informando a possibilidade de infinitas soluções ótimas.
output$resultadovertice1 <- renderText({
  partes = strsplit(respostacertavertice(), " e ")[[1]]
  infinitassolucoesotimas1 = infinitassolucoesotimas()
      paste0(input$nome, ", DICA:", infinitassolucoesotimas1)
})
}
shinyApp(ui = ui, server = server)
```

Comentários Técnicos

1. Estrutura e Organização do Código:

- **Separação UI/Server:** O código segue a estrutura padrão do Shiny, com a interface do usuário (UI) definida em ui <- fluidPage(...) e a lógica do servidor em server <- function(input, output, session) {...}.
- **Uso de Pacotes:** O aplicativo utiliza os pacotes shiny, ggplot2 e DT, que são adequados para a criação de interfaces web interativas, visualização de dados e exibição de tabelas, respectivamente.
- **Estilização CSS:** A estilização da interface é feita diretamente no código R, utilizando a tag tags$style(HTML(...)).

2. Reatividade e Gerenciamento de Dados:

- **Objetos Reativos:** O código faz uso extensivo de objetos reativos (reactive({...})) para atualizar dinamicamente os elementos da interface. Isso garante que as alterações nos dados de entrada sejam refletidas imediatamente nos gráficos e tabelas.

- **Função debounce():** A função debounce() é utilizada para evitar cálculos excessivos enquanto o usuário digita os dados. Isso melhora a performance do aplicativo, especialmente em entradas complexas.
- **reactiveValues():** O uso de reactiveValues() para armazenar a matriz e a tabela de vértices é uma boa prática, pois permite que esses dados sejam atualizados de forma reativa e acessados por diferentes partes do aplicativo.
- **Cálculo de Vértices:** O cálculo dos vértices é feito utilizando operações matriciais e lógicas.
- **Validação de Entradas:** O código inclui algumas validações básicas para as entradas do usuário, como verificar se os valores são numéricos e se as dimensões dos vetores estão corretas.

3. Visualização de Dados:

- **ggplot2:** O uso do pacote ggplot2 para criar os gráficos permite gerar visualizações personalizadas e de alta qualidade.
- **Representação da Região Viável:** Na representação da região viável é utilizado o geom_tile() e scale_fill_manual().
- **Curvas de Nível:** A adição das curvas de nível da função objetivo utilizando geom_abline() e geom_segment() auxilia na visualização da direção de otimização.
- **DT::dataTableOutput():** A utilização do pacote DT para renderizar as tabelas garante interatividade e boa formatação.

4. Lógica do Teste:

- **Validação da Resposta:** Teste lógico para validar a resposta do usuário no teste.
- **Identificação de Solução Ótima:** O código identifica a solução ótima, considerando os diferentes casos (maximizar/minimizar, soluções únicas/múltiplas, exceto soluções ilimitadas).
- **Mensagens de Feedback:** As mensagens de feedback para o usuário são informativas e ajudam a entender os resultados.