ChatGPT

Documentação Técnica da Linguagem Quadcode Script (QCS)

[1] **Introdução à Quadcode Script:** A Quadcode Script (QCS) é uma linguagem de scripting baseada em **Lua 5.3**, desenvolvida para criar indicadores técnicos personalizados e estratégias automatizadas em plataformas de trading como **IQ Option** e **Avalon** [1] [2]. Por ser embutida no ambiente de negociação, a QCS permite analisar dados de mercado em tempo real, gerar sinais de compra/venda e até executar ações automatizadas (por exemplo, abrir ou fechar ordens) diretamente no terminal de trading [2]. A sintaxe e semântica da linguagem aproveitam muitos recursos do Lua original, garantindo **execução rápida** e baixo impacto de performance na plataforma [3]. Entretanto, a QCS estende Lua com funções específicas de análise técnica e impõe certas **particularidades importantes**, as quais documentaremos detalhadamente (como tratamento especial de valores `0` em condições lógicas e disponibilidade de funções matemáticas globais sem prefixo `Math`).

Visão Geral e Estrutura de um Script QCS: Um script QCS é essencialmente um *indicador técnico ou estratégia* definido pelo usuário. Cada script tipicamente começa declarando um **instrumento** (ou indicador) e suas configurações básicas, seguido pela definição de quaisquer parâmetros de entrada customizados, e então a lógica de cálculo principal que produz os resultados (valores de indicadores ou sinais). Por fim, o script especifica como apresentar esses resultados no gráfico (por exemplo, traçando linhas, candles, formas ou áreas coloridas). Importante: a função especial `instrument{ ... }` deve ser chamada **uma única vez no início** do script, definindo o nome e propriedades do indicador [4] [5]. Em seguida, podem ser declarados parâmetros de entrada com `input` e agrupados opcionalmente em seções com `input_group` para facilitar ajustes pelo usuário no terminal [6] [7]. Após as entradas, o corpo do script consiste em expressões de cálculo que utilizam **séries de dados de preço** (preço atual e histórico) e funções para computar o indicador ou estratégia. O script pode definir variáveis locais com resultados intermediários e usar estruturas de controle de fluxo do Lua (condicionais `if-elseif-else`, loops etc.) conforme necessário para implementar a lógica desejada. Finalmente, para **exibir** a saída, utilizam-se funções de plotagem – por exemplo, `plot()` para linhas, `plot_candle()` para candles customizados, `plot_shape()` para ícones de sinal, `hline()` para linhas de referência horizontais, entre outras – definindo cores, estilos e outras características visuais [8] [9].

1. Tipos de Dados, Variáveis e Séries Temporais

A QCS suporta os mesmos tipos primitivos do Lua, adaptados ao contexto de análise de mercado. Os principais tipos incluem:

- **Inteiros (`integer`):** valores numéricos inteiros de 64 bits com sinal. Utilizados para períodos, contagens de barras, índices, etc. [10]. Exemplo: `periodo = 14` (um período default típico) [11].
- **Numéricos (`numeric`):** valores de ponto flutuante (64 bits) para preços e cálculos com decimais [12]. Qualquer literal com ponto (ex: `3.14`) é numérico.
- **Booleanos (`boolean`):** valores lógicos `true` ou `false`. Em Lua (e QCS), apenas `false` e `nil` são considerados falsos em condições; todos os outros valores são verdadeiros por padrão [13]. **Atenção:** em algumas funções especiais da QCS (explicitamente documentadas), o valor numérico `0` **é tratado como falso** também [14], diferentemente do comportamento

padrão do Lua. Isso é relevante, por exemplo, na função ternária `iff` (descrita adiante), onde `0` é interpretado como condição falsa [14].

- **Strings (`string`):** cadeias de texto, usadas para nomes, rótulos, identificação de ativos, etc. [15]. Strings em QCS seguem a sintaxe Lua: delimitadas por aspas duplas ou simples. Exemplo: `instrument { name = "SMA", overlay = true }` – aqui `"SMA"` é uma string definida como nome do indicador [16].

Além dos tipos básicos, a QCS introduz o tipo fundamental `series`, que representa uma sequência histórica de valores (uma série temporal). Uma variável do tipo série carrega valores barra a barra (candle a candle), permitindo acesso tanto ao **valor corrente** quanto a históricos via indexação. Por exemplo, as palavras reservadas `open, high, low, close` são séries fornecidas pelo ambiente, correspondendo aos preços de abertura, máxima, mínima e fechamento da barra atual, respectivamente [17] [18]. Se indexadas com um deslocamento entre colchetes, retornam valores passados: `close[1]` retorna o fechamento da barra anterior, `close[2]` de duas barras atrás, e assim por diante [19]. Se o índice ultrapassar o histórico disponível, o resultado é `nil` (ou `nan`, "not a number", dependendo do caso) [19]. Por exemplo:

```
valor_atual   = close          -- preço de fechamento da barra atual
valor_ontem  = close[1]       -- preço de fechamento da barra anterior
valor_atras2 = close[2]       -- preço de fechamento de duas barras atrás
print(valor_atual, valor_ontem, valor_atras2)
```

Na saída acima, `valor_atras2` será `nil` se não houver duas barras de histórico (por exemplo, no início da série) [19].

Operações com séries: Uma característica poderosa é que podemos aplicar **operações aritméticas diretamente em variáveis do tipo `series`**, e o resultado será outra série calculada elemento a elemento. A QCS suporta operações como soma, subtração, multiplicação, divisão e exponenciação entre séries e/ou números escalar (constantes) [20] [21]. Por exemplo, `mediaHL = (high + low) / 2` produzirá uma série `mediaHL` onde cada barra é a média de máxima e mínima da respectiva barra (conhecida como **HL2**, ou *midpoint*) [22] [23]. Também são permitidas comparações entre séries (>, <, ==, etc.), resultando em valores booleanos (ou séries booleanas) que podem ser usados para definir condições de trade ou mudança de cor em gráficos [24] [25]. *Exemplo:* a expressão `close > open` produz uma condição booleana verdadeira em barras onde o preço fechou acima do preço de abertura (candle de alta), e falsa caso contrário.

No entanto, devido a limitações da engine (herdadas do Lua), **não se pode usar diretamente uma série booleana em uma estrutura de controle como `if`** sem tratamento [26] [27]. Por exemplo, `if (close > open) then ... end` é sintaticamente válido, mas não se comporta como esperado porque `close > open` é do tipo `series` (contendo potencialmente múltiplos valores ao longo do histórico, não um booleano simples). Para contornar isso e avaliar condições com base no valor **atual** de uma série, a QCS fornece a função `get_value()` – que retorna o valor da série na barra corrente [26] [28] – ou alternativamente podemos converter a condição em série para um único valor booleano usando a função `conditional()` (veja seção de Funções de Condição). O padrão recomendado é:

```
condicao = conditional(close > open)    -- converte condição em série
booleana
if get_value(condicao) then
```

```
    print("Barra atual fechou em alta")
end
```

No exemplo acima, `conditional(close > open)` cria uma série booleana chamada `condicao` cujo valor **na barra atual** reflete se houve alta. O `get_value(condicao)` extrai esse valor booleano atual para uso no `if` [29] [30]. Em resumo: para usar resultados de séries em controles de fluxo, **sempre extrair o valor atual com** `get_value()` ou usar as funções condicionais adequadas, pois comparações diretas `series` vs `numeric` não são permitidas [26].

Variáveis locais vs globais: Assim como em Lua, podemos (e devemos, por boas práticas) declarar variáveis locais usando a palavra-chave `local`. Variáveis locais dentro do script são reavaliadas a cada cálculo de barra sem persistir estado global, evitando interferência entre execuções e melhorando performance. Exemplo: `local diff = close - open` calcula a diferença na barra atual e armazena em `diff` localmente. Se omitirmos `local`, a variável torna-se global no contexto do script, o que é desencorajado para evitar conflitos e lentidão.

Notação e convenções importantes: - A QCS adota a sintaxe Lua para operadores lógicos: usa `and` para E lógico, `or` para OU, e `not` para negação. Operadores de comparação são `==` (igual) e `~=` (diferente). Desenvolvedores acostumados com JavaScript devem notar estas diferenças (em JS seria `&&`, `||`, `!`, `===`, `!==`, etc.). - **Funções matemáticas globais:** Diferentemente do JavaScript, **não se deve usar prefixo** `Math.` nas chamadas de funções matemáticas. Todas as funções matemáticas comuns (valor absoluto, raiz, seno, etc.) são disponibilizadas como funções globais em QCS – ou seja, usa-se `abs(x)` em vez de `Math.abs(x)`. Por exemplo, `valor = abs(-5)` retornará 5 corretamente, ao passo que `Math.abs(-5)` geraria erro por `Math` não existir no ambiente QCS. A documentação oficial exemplifica o uso correto: `value1 = abs(-1) -- equals 1` [31]. Da mesma forma, outras funções como `sqrt`, `sin`, `cos` etc. são invocadas diretamente (sem prefixo). *Observação:* em alguns exemplos, a documentação utiliza a notação `math.min`, `math.max` [32] – isso reflete a herança do Lua (onde existe biblioteca `math`). Entretanto, em QCS atual, pode-se empregar simplesmente `min()` e `max()` conforme listadas nas funções built-in [33] [34]. O importante é **não usar** `Math.` **com 'M' maiúsculo**, pois isso corresponde ao objeto Math do JavaScript (inexistente em Lua/QCS) – fonte de erros comuns para iniciantes.

2. Componentes-Chave de um Script QCS

Nesta seção detalharemos os principais blocos e funções que estruturam um script Quadcode Script, conforme definidos na documentação oficial, incluindo declarações de indicador, entradas customizadas e tipos de séries embutidas.

2.1 Declaração do Instrumento (Indicador)

Cada script começa declarando o **instrumento** que está sendo criado – isto é, definindo um novo indicador ou estratégia. A QCS provê a função especial `instrument{ }` para esse fim. A sintaxe típica é:

```
instrument { name = "Nome do Meu Indicador", overlay = true }
```

Onde se especifica pelo menos dois parâmetros principais dentro das chaves:

- **name (string):** Nome completo do indicador, exibido na interface da plataforma (por exemplo, lista de indicadores e título do painel). Deve descrever claramente a função do indicador [35].
- **overlay (boolean):** Define se o indicador será **sobreposto** ao gráfico de preços (`true`, como médias móveis, bandas de Bollinger etc.) ou exibido em um **painel separado** abaixo (`false`, como osciladores tipo RSI, MACD) [5].

A função `instrument` **deve ser chamada uma vez no início do script** [4] antes de qualquer cálculo. Opcionalmente, podem-se fornecer parâmetros adicionais na declaração: - **short_name (string):** um nome curto ou abreviado do indicador, se desejado (não documentado oficialmente na página de parâmetros, mas suportado – por exemplo, `short_name = "RSI"`). - **icon (string):** um identificador ou URL de ícone para o indicador, caso a plataforma permita mostrar um ícone personalizado na lista de indicadores (ex.: `icon = "indicators:MA"` para usar um ícone padrão de média móvel) [36] [37]. - **description (string):** descrição do indicador que pode aparecer em tooltips ou na área abaixo do gráfico quando overlay for false [38].

Caso `overlay = false`, o indicador aparecerá em um subgráfico separado (por exemplo, um oscilador terá seu próprio eixo y). Se `overlay = true`, os valores plotados são desenhados sobre o gráfico de preços do ativo atual [5]. *Exemplo:* `instrument { name = "Volume Oscillator", overlay = false }` declararia um indicador de Volume que aparece abaixo do gráfico de preços.

2.2 Parâmetros de Entrada do Usuário

Para tornar indicadores e estratégias flexíveis, a QCS permite definir **inputs** personalizáveis – valores que o usuário pode ajustar na interface (como períodos, cores, opções de fonte de dados, etc.). Existem duas construções principais para isso:

- `input`: define um único parâmetro de entrada.
- `input_group`: organiza múltiplos inputs relacionados em um grupo com um rótulo comum.

A função `input` possui duas formas de uso: uma forma simplificada (passando apenas valor padrão e nome) e uma forma completa usando tabela com campos nomeados. A forma mais legível utiliza a sintaxe de tabela, por exemplo:

```
período = input { default = 14, type = input.integer, name = "Period", min = 1 }
```

Isso cria um input inteiro com valor padrão 14, nome "Period" e valor mínimo 1. Na interface da plataforma, ao anexar o script, este parâmetro aparecerá permitindo ajuste [11]. Os campos comuns em `input { }` incluem: - `default`: valor padrão inicial. - `type`: tipo do input (deve ser um dos tipos suportados, e.g. `input.integer`, `input.double` para decimal, `input.color` para escolha de cor, `input.string_selection` para seleção de string de um conjunto, etc.). - `name`: rótulo legível do parâmetro. - Parâmetros opcionais específicos do tipo, como limites mínimo/máximo para números (`min`, `max`) e passo (`step`), ou lista de opções no caso de seleção.

Já a função **`input_group { }** permite agrupar vários inputs sob um mesmo título, o que melhora a organização visual nas configurações do indicador. Por exemplo, podemos agrupar inputs de personalização de cor e espessura de linha:

```
input_group {
    "Média Móvel",  -- Título do grupo
    cor_linha = input { default = "#FF0000", type = input.color, name = "Cor
da Linha" },
    largura = input { default = 2, type = input.line_width, name =
"Espessura" }
}
```

Nesse exemplo, definimos dois inputs (cor e largura) dentro de um grupo nomeado "Média Móvel". Na interface, esses dois inputs aparecerão juntos sob o cabeçalho "Média Móvel" [39]. Grupos são úteis para separar categorias de parâmetros como *"Configurações de Período"*, *"Opções de Visualização"*, etc., tornando a interface mais amigável.

Seleção de fonte de dados: A QCS também provê inputs especiais para permitir que o usuário escolha qual série de preço usar em cálculos (por exemplo, permitir optar entre usar **close, open, high, low** ou até outra série derivada como volume). Isso é feito combinando `input.string_selection` com listas predefinidas. Por exemplo, a biblioteca embutida define `inputs.titles` ou `inputs.titles_overlay` representando opções "Close", "Open", "High", "Low" etc. [40] [41]. Um uso típico:

```
fonte = input(1, "Source", input.string_selection, inputs.titles_overlay)
...
local seriesEscolhida = inputs[fonte]
```

No trecho acima, `input(1, ...)` define um dropdown onde 1 corresponde à primeira opção (geralmente "Close"). A variável `seriesEscolhida` então obtém a série real correspondente através de `inputs[...]` (um array provido pelo ambiente com as séries padrão) [41]. Esse mecanismo permite escrever indicadores genéricos aplicáveis a diferentes preços (fechamento, abertura, etc.) conforme seleção do usuário. De forma similar, existe uma tabela `averages.titles` e funções `averages.<nome>` para seleção de métodos de média (SMA, EMA, SSMA, etc.) em indicadores que permitam escolher o tipo de média [42] [43].

Exemplo completo de inputs: Abaixo um exemplo resumido inspirado no indicador RSI oficial, demonstrando vários tipos de input:

```
instrument { name = "RSI Custom", overlay = false }

input_group {
    "RSI",
    periodo = input(14, "Period", input.integer, 1, 250),               --
inteiro entre 1 e 250
    fonte = input(1, "Source", input.string_selection, inputs.titles),   --
escolha de série de preço
    media = input(averages.rma, "Moving Average", input.string_selection,
averages.titles) -- escolha do tipo de média (RMA=Wilder por padrão)
}

input_group {
```

```
    "Linhas de Referência",
    sobrecompra = input(70, "Overbought", input.double, 1, 100, 1),
    sobrevenda = input(30, "Oversold", input.double, 1, 100, 1),
    cor_zona = input { default = rgba(255,255,255,0.05), type = input.color,
name = "Cor Zona OB/OS" }
}
```

No código hipotético acima, agrupamos os parâmetros do RSI (período, fonte de preços, tipo de média) e em outro grupo definimos níveis de sobrecompra/survivenda e uma cor para preencher a zona entre eles. A plataforma exibiria isso de forma organizada, facilitando o ajuste pelo trader.

2.3 Séries Embutidas (Preços, Tempo e Volume)

O ambiente QCS fornece acesso direto a várias **séries de dados de mercado**, sem necessidade de declaração explícita. Estas séries embutidas representam preços do ativo atual, informações de tempo e volume, e podem ser usadas como entradas para cálculos de indicadores. As principais séries disponíveis incluem:

- **Preço – Séries de OHLC:**
- `open` – preço de **abertura** da barra atual [17] (tipo: series float; *overlayable*, ou seja, faz sentido graficá-la sobre o chart principal).
- `high` – preço **máximo** da barra atual [44].
- `low` – preço **mínimo** da barra atual [45].
- `close` – preço de **fechamento** da barra atual [18].
- **Séries derivadas comuns:**
  - `hml` – valor de *High minus Low* (máxima menos mínima) na barra atual [46]. Essencialmente `hml = high - low`.
  - `hl2` – média simples de High e Low [22] [23], equivalente a `(high + low) / 2`.
  - `hlc3` – média de High, Low e Close [47] [48], i.e. `(high + low + close) / 3`.
  - `ohlc4` – média de Open, High, Low e Close [49] [50], i.e. `(open + high + low + close) / 4`. (Essa geralmente é a *média aritmética da barra*).
  - `hlcc4` – média ponderada onde o Close tem peso duplo: `(high + low + close + close) / 4` [51] [52].
  - `tr` – *True Range*, ou intervalo verdadeiro, definido como o máximo entre `(high - low)`, `(high - close[1])` e `(close[1] - low)` [53] [54]. Essa série é útil em indicadores de volatilidade (e.g. ATR).

Exemplo: Para obter o preço de fechamento da barra anterior, usa-se `close[1]`. Para a máxima de 10 barras atrás: `high[10]`. Para calcular, digamos, a média de máximas e mínimas da barra anterior: `média_anterior = (high[1] + low[1]) / 2`. As séries de preço podem ser combinadas livremente com operações e funções matemáticas, gerando novas séries. Por exemplo, `range_percent = (high - low) / close * 100` resultaria numa série percentual do *range* em relação ao preço de fechamento.

- **Tempo – Séries temporais:** QCS também fornece séries relativas a timestamp das barras:
- `open_time` – timestamp de **abertura** da barra (epoch/unix time em milissegundos) [55] [56].
- `close_time` – timestamp de **fechamento** da barra [57] [58].
- `day` – dia da abertura da barra (número do dia no ano) [59] [60].
- `week_day` – dia da semana da abertura (0=Domingo, 1=Segunda, ... possivelmente) [61] [62].
- `month` – mês da abertura (1-12) [63] [64].

- `year` – ano da abertura (ex: 2025) [65] [66].
- `hour`, `minute`, `second` – hora, minuto e segundo da abertura da barra (dependendo da resolução do gráfico, estas podem ser relevantes) [67] [68].

Essas séries de tempo são do tipo `series int` (inteiros) e *não são "overlayable"*, ou seja, por padrão seu plot abre em painel separado, já que são valores de tempo, não de preço [69] [70]. São úteis para construir indicadores baseados em horário (por exemplo, colorir barras de pregão regular vs pós-mercado) ou para usar em condições, como `if hour == 9 then ... end` (embora a plataforma já segmente gráficos por período).

- **Volume – Série de volume de negociação:**
- `volume` – volume negociado na barra atual [71] [72]. Tipicamente número de contratos, ações ou lotes dependendo do ativo. Tipo: `series float` (*non-overlayable*, costuma ser plotado em subjanela). Um exemplo de uso: `plot(volume, title="Volume", color=color.green, style=plot.style_histogram)` para desenhar um histograma de volumes.

Todas essas séries são atualizadas a cada nova barra e podem ser indexadas para histórico como os preços. Vale ressaltar: comparar diretamente uma série com um valor fixo não é permitido (ex: `if close > 100 then` dá erro), devendo-se usar `get_value()` ou converter para série booleana com `conditional`. Entretanto, a QCS oferece a função auxiliar `get_value(series)` para extrair o valor atual de qualquer série (preço, tempo ou volume) [26] [27]. Se a série não tiver valor na barra corrente, um default pode ser fornecido como segundo argumento (veja seção 3).

Outro ponto: por design do Lua, não se pode fazer comparação direta entre tipos diferentes. Assim, **não é possível comparar uma `series` com um `number` diretamente** (ex: `if close > 100 then` *não funcionaria*, pois `close` é série). O correto seria: `if get_value(close) > 100 then ... end` ou usar `close > 100` dentro de `conditional()` para obter uma série booleana (porém, de novo precisaria extrair o valor para agir no if).

Funções auxiliares para séries: A QCS inclui várias funções para manipular séries de forma segura e obter informações delas. Algumas das principais são: - `is_series(x)` – retorna `true` se o valor dado é do tipo série, caso contrário `false` [73] [74]. - `is_value(x)` – retorna `true` se o valor dado é um número (não série) e não `nil` [75] [76]. - `isnan(x)` – retorna `true` se *x* não é numérico ou é `nan` (not a number) [77] [78]. Útil para checar resultados de divisões por zero ou inícios de série sem dados suficientes. - `get_name(series)` – retorna o **nome** (identificador) da série dada, como string [79] [80]. Por exemplo, `get_name(close)` retornaria `"close"`. - `make_series(name)` – cria explicitamente uma nova série vazia, opcionalmente com um nome definido [81] [82]. Essa função é raramente necessária, mas permite inicializar séries manualmente. Após criar, pode-se usar o método `:set(valor)` para atribuir valor na barra atual daquela série (como mostrado no exemplo: `cg = make_series(); cg:set(1)` [82]). - `index_range(start_index)` – gera uma série de inteiros começando em `start_index` e aumentando sequencialmente por barra [83] [84]. Útil para obter, por exemplo, uma contagem de barras ou um eixo X customizado.

3. Operadores Lógicos e Controle de Fluxo

Sendo baseada em Lua, a QCS suporta todas as estruturas de controle padrão da linguagem, as quais podem ser empregadas livremente no script para implementar lógica condicional ou iterativa. Entretanto, devido à natureza orientada a séries temporais, certos cuidados são necessários.

3.1 Condicionais (if/else) com Séries e Funções de Condição

A estrutura condicional básica em QCS segue a sintaxe Lua:

```
if condição_então then
    -- bloco se condição for verdadeira
elseif outra_condição then
    -- bloco alternativo
else
    -- bloco se nenhuma condição anterior for verdadeira
end
```

A diferença crucial na QCS é que **as condições geralmente envolvem séries de dados**, não apenas variáveis escalares. Conforme mencionado, não podemos colocar diretamente uma série como condição do `if` sem processá-la. Portanto, ao escrever condições envolvendo dados de mercado, costuma-se utilizar: - A função `conditional(condição_boolean)` – que **converte** uma expressão booleana em série para uma série booleana cujo valor atual reflete a condição [85] [86]. - Em seguida, usar `get_value()` para extrair esse valor atual para uso na estrutura de controle.

Exemplo prático: queremos executar uma lógica quando o fechamento atual for maior que o anterior **e** o volume atual for maior que o volume anterior:

```
cond = conditional(close > close[1] and volume > volume[1])
if get_value(cond) then
    -- por exemplo, imprimir ou sinalizar "mercado em alta com aumento de
volume"
    print("Alta com volume crescente")
end
```

No código acima, `close > close[1] and volume > volume[1]` produz uma série booleana (porque tanto close quanto volume são séries). `conditional(...)` registra essa condição em formato de série (apenas a barra atual terá valor definido imediatamente após a chamada). `get_value(cond)` então retorna `true` ou `false` conforme a condição na barra atual [28] [30], permitindo o `if` decidir.

Uma alternativa é usar diretamente `if (close > close[1]) and (volume > volume[1]) then ... end`, mas isso **não funcionaria**, pois a condição dentro do `if` ainda é do tipo série. Portanto, **é obrigatório** converter e extrair valor atual, conforme acima.

Condição com 0 como falso: Em QCS, ao contrário do Lua tradicional, algumas funções consideram 0 como falso na avaliação de condições, para conveniência dos casos de indicadores (ex: um oscilador que retorna 0 pode ser tratado como sinal desativado). Um caso notório é a função ternária `iff` (descrita mais adiante), em que a documentação especifica: *"A condição é considerada falsa se for nil, nan, false ou 0; verdadeira caso contrário"* [87]. Portanto, é importante estar ciente de que `0` **pode atuar como falso** em certos contextos QCS, mesmo que normalmente no Lua `0` seja true. Fora dessas funções especiais, a regra geral do Lua se mantém (0 é avaliado como true em condicionais normais), mas ao usar funções QCS de utilidade lógica, convém verificar seus detalhes.

3.2 Operador Ternário e Função `iff`

Como Lua não possui um operador ternário (`cond ? valor1 : valor2` como em C/JS), a QCS introduz a função `iff(condição, valor_se_verdadeiro, valor_se_falso)` para preencher essa lacuna de maneira segura e eficiente [14] [88]. Essa função avalia a condição e retorna uma série cujos valores são escolhidos entre os dois fornecidos conforme a condição para cada barra.

- Se **qualquer** dos argumentos `valor_se_verdadeiro` ou `valor_se_falso` for do tipo `series`, o retorno de `iff` será do tipo série; se ambos forem valores simples, retorna um valor simples (numeric ou string, etc.) [89] [88].
- A condição em `iff` é interpretada de forma ampliada: 0 conta como false (conforme mencionado) além de false, nil e nan [87].

Exemplo de uso típico: colorir candles de verde se fechamento maior que abertura, vermelho caso contrário. Podemos definir:

```
cor = iff(close > open, color.green, color.red)
plot_candle(open, high, low, close, "MyCandles", cor)
```

Acima, `iff(close > open, color.green, color.red)` produzirá uma série de cores (tipo série de string de cor) onde cada barra recebe `"green"` ou `"red"` conforme a condição daquele candle [90] [91]. Essa série é passada para o parâmetro de cor do `plot_candle` que então desenhará o candle em verde ou vermelho automaticamente.

Outro exemplo: gerar sinais de flecha somente na barra em que determinada condição acontece:

```
buySignal = iff(condition, 1, 0)
plot_shape(buySignal, shape_style.arrowup, shape_location.belowbar,
color.green, "Buy")
```

No snippet, definimos `buySignal` como 1 se `condition` for verdadeira, ou 0 caso contrário. Ao passar isso para `plot_shape`, como o estilo não é absoluto, a QCS vai interpretar 1 como true (desenha a seta) e 0 como false (não desenha nada) na barra [92] [93]. Assim obtemos uma seta "Buy" apenas onde a condição ocorreu.

Importante: Ao usar `iff` para cálculos numéricos, lembre-se que se a condição for false em alguma barra, `iff` retornará o valor do ramo "falso" naquela barra. Se esse valor falso for, por exemplo, 0 ou nil, isso pode afetar séries calculadas. Caso queira evitar interrupções na série resultante, pode-se combinar `iff` com funções como `nz` ou `fixnan` (ver seção de Depuração) para substituir NaNs.

3.3 Loops e Iterações

Embora indicadores técnicos raramente requeiram loops explícitos (pois as operações vetorizadas em séries já computam valores barra a barra eficientemente), a QCS permite uso de **loops** do Lua para cálculos personalizados se necessário. Podem ser utilizados: - `for` **numérico:** ex: `for i = 1, 10 do ... end` para iterar de 1 a 10. - `while` **loop:** ex: `while cond do ... end`. - `repeat ... until`: ex:

```
local count=0
repeat
  count += 1
until someSeries[count] == nil
```

(embora esse último caso seja pouco comum).

Ao usar loops, cuidado para não iterar indefinidamente por engano – lembre que um script de indicador roda continuamente a cada nova barra, então loops muito extensos em cada cálculo podem prejudicar a performance do terminal. Em geral, é preferível utilizar as funções prontas de séries (como `highest`, `lowest`, `sum`, etc. descritas adiante) em vez de iterar manualmente sobre valores passados, pois as funções nativas são otimizadas e mais concisas.

Um exemplo de loop possível: cálcular a soma móvel manualmente (não recomendado, apenas ilustrativo):

```
local period = 5
local sumSeries = make_series()
for offset = 0, period-1 do
    sumSeries:set(get_value(sumSeries) + (close[offset] or 0))
end
```

No pseudo-código acima, utilizamos `close[offset]` para somar os últimos 5 fechamentos. Entretanto, existe a função `sum(close, 5)` pronta que faz exatamente isso, de forma mais eficiente e retornando uma série de somas móveis (ver Funções Matemáticas). Portanto, só recorra a loops se não houver função nativa que atenda à necessidade.

3.4 Funções de Comparação de Extremos e Contagem de Barras

Relacionado a controle de fluxo, frequentemente queremos detectar **máximos/mínimos recentes** ou **quantas barras se passaram** desde um evento. Para isso, a QCS disponibiliza: - `highest(series, n)` – retorna uma série cujo valor em cada barra é o maior valor da série de entrada nos últimos `n` períodos (incluindo o atual) [94] [95]. Também existe a forma `highest(n)` que é atalho para `highest(high, n)`, ou seja, o maior preço máximo nos últimos n períodos [96]. - `lowest(series, n)` – análogo ao acima, retorna o menor valor da série nos últimos `n` períodos. Há também `lowest(n)` equivalente a `lowest(low, n)` (menor mínima recente) [97] [98]. - `bars_since_highest(series, n)` – retorna quantas barras desde que ocorreu o valor mais alto da série nos últimos `n` períodos (inclusive atual). Se na barra atual a série é o ponto mais alto, retorna 0; se não houve valor >= ao da barra atual nos `n` anteriores, retorna n (ou possivelmente `nan` se não aplicável) [99] [100]. Da mesma forma `bars_since_lowest(series, n)` para mínimos [101]. Existem variantes `bars_since_highest(n)` e `bars_since_lowest(n)` que presumivelmente usam `high` e `low` por padrão [102] [103].

Essas funções facilitam a criação de condições como "se o fechamento atual for a máxima dos últimos 20 candles" – podemos checar: `if get_value(close == highest(close, 20)) then ... end` ou usando contagem: `if bars_since_highest(close, 20) == 0 then ... end` (esta última diretamente retornaria 0 na barra que for novo topo de 20 períodos). Lembrando de extrair o valor se usar diretamente no if.

4. Funções de Plotagem e Aparência de Gráficos

Uma vez calculados os valores do indicador ou definidos os sinais de estratégia, precisamos **exibi-los graficamente**. A QCS fornece um conjunto robusto de funções para plotagem de linhas, candles customizados, formas (setas, símbolos), áreas preenchidas e outros elementos visuais. Essas funções geralmente produzem *plots* (traçados) que aparecem no gráfico do ativo ou em painéis separados, conforme especificado.

Abaixo listamos as principais funções de visualização e suas utilidades:

- `plot(series, title, color, width, style)` – Plota uma **linha** correspondente à série fornecida. Retorna o identificador do plot (nomeado). Parâmetros:
- `series`: a série de valores a desenhar (normalmente float).
- `title` (string): nome do plot (exibido na legenda do gráfico).
- `color`: cor da linha (pode ser constante – e.g. `color.red` – ou uma série de cores que varia por barra).
- `width` (int): espessura da linha em pixels.
- `style`: estilo da linha. Há vários estilos disponíveis através de constantes de enum, como `plot.style_solid` (linha sólida padrão), `plot.style_dash_line` (tracejada), `plot.style_histogram` (barras verticais estilo histograma), `plot.style_area` (área preenchida sob a curva), etc. *(Essas constantes residem provavelmente em um namespace como* `plot.style_*`*)*. Se não especificado, default é linha sólida.

Exemplo: `plot(sma(close, 20), "SMA 20", color.blue, 2)` desenharia uma média móvel de 20 períodos em azul, linha de espessura 2px.

- `plot_candle(open, high, low, close, title, color)` – Desenha **velas (candles)** customizadas baseadas nas séries de preços fornecidas. Útil para plotar candles de um cálculo próprio (como Heikin Ashi, ou reconstruir candles de timeframes diferentes). Parâmetros:
- `open, high, low, close`: quatro séries numéricas correspondentes aos valores OHLC das "velas" a desenhar. É comum calculá-las antes. Exemplo: no indicador de Heikin Ashi da documentação, após calcular `ha_open, ha_high, ha_low, ha_close`, chamam `plot_candle(ha_open, ha_high, ha_low, ha_close, "Heikin Ashi Candles", cor_candle)` [8] [91].
- `title`: nome do plot (ex: "Heikin Ashi Candles").
- `color`: pode ser usado para colorir candles bullish/bearish diferentemente. Geralmente se passa uma expressão condicional: por exemplo, `ha_open < ha_close and bullish_color or bearish_color` para escolher verde ou vermelho [104] [91]. Em QCS, a sintaxe `cond and X or Y` funciona como ternário simples quando X não é false/nil – mas é mais seguro usar `iff` ou calcular antes a cor. No exemplo da documentação, eles fizeram `ha_open < ha_close and bullish_color or bearish_color` dentro do plot (aproveitando a lógica Lua where `exp1 and exp2 or exp3`).

Dica: Se quiser ocultar certas velas condicionalmente, pode-se passar `na` (not available) como valores para open/high/low/close nessas barras, ou usar `drop_plot_value` (abaixo) para removê-las após plotadas.

- `plot_shape(series_bool, style, location, color, text, size, offset, text_color)` – Desenha **formas/símbolos** em posições de interesse no gráfico, geralmente para sinalização. Esta função é bastante versátil. Parâmetros principais:

- `series_bool`: Uma série cujos valores serão interpretados como booleanos (true/false) indicando onde desenhar a forma. Normalmente resulta de uma condição convertida: por exemplo, podemos passar `close > open` diretamente; internamente qualquer valor não-nulo/não-zero é considerado true para plotagem se o `location` não for absoluto [92].
- `style`: qual símbolo desenhar. A QCS define diversos **enum** em `shape_style`, como `shape_style.arrowup`, `arrowdown`, `circle`, `cross`, `diamond`, `flag`, `labelup`, `labeldown`, `square`, `triangleup`, `triangledown`, `xcross` etc. [105] [106]. Padrão é `shape_style.xcross` (um X).
- `location`: onde posicionar a forma verticalmente. Valores de enum em `shape_location`: `abovebar` (acima da barra), `belowbar` (abaixo da barra), `top` (no topo do gráfico), `bottom` (na base do gráfico) ou `absolute` (posicionar usando o valor numérico da série fornecida) [107] [108]. Por exemplo, para desenhar setas de compra abaixo das barras: usar `shape_location.belowbar`. Se quisermos desenhar símbolos no valor de uma série (ex: marcar pontos em cima de um indicador), pode-se usar `shape_location.absolute` e passar a série de valores a marcar em vez de um boolean.
- `color`: cor do símbolo.
- `text`: texto a exibir junto à forma (por exemplo, "BUY" ou "S" para sell). Se não quiser texto, pode usar string vazia ou omitir.
- `size`: tamanho do símbolo, definido em `shape_size` enum: `tiny, small, normal, large, huge` ou `auto` (automático proporcional ao zoom) [109]. Padrão é `auto`.
- `offset`: deslocamento em número de barras para esquerda ou direita. Útil se quiser que o símbolo apareça algumas barras antes ou depois da condição. Padrão 0 (na barra da condição).
- `text_color`: cor do texto se `text` for fornecido (por padrão, branco).

Exemplo de uso:

```
condCompra = close > sma(close, 50)
-- condição: fechamento cruzou acima da média de 50
plot_shape(condCompra, shape_style.triangleup, shape_location.belowbar,
color.lime, "Buy", shape_size.normal)
```

Isso desenharia um triângulo verde **abaixo** da barra em cada ponto que o fechamento estiver acima da média 50 (provavelmente queremos na barra do cruzamento exato; aqui o condCompra seria true para todas barras acima, então desenharia em todas – num caso real usaríamos cruzamento ex: `cross = close > sma50 and close[1] <= sma50[1]` para sinal apenas quando cruza). Em todo caso, `plot_shape` facilita destacar visualmente eventos discretos.

Limitações: A documentação ressalta que **algumas formas não são suportadas em plataformas móveis**, sendo substituídas automaticamente por outras formas semelhantes [110]. Por exemplo, `circle` vira quadrado no mobile, `flag` também vira quadrado, e setas (`arrowup/down`) e triângulos para baixo viram triângulo pra cima em alguns casos [111]. Ou seja, ao desenvolver scripts, tenha em mente essa limitação caso o público use aplicativo móvel (os símbolos podem aparecer diferentes, mas a lógica permanece).

- `hline { value = X, color = Y, width = Z, style = ..., show_label = bool }` – Desenha uma **linha horizontal** no valor especificado do eixo vertical. Serve para marcar níveis de sobrecompra/sobrevenda, zero, etc. Parâmetros:
- `value`: valor numérico onde traçar a linha (ex: 0, 30, 70).
- `color`: cor da linha.

- `width`: espessura (similar ao plot).
- `style`: estilo da linha (sólida, tracejada, pontilhada, etc. - similares aos de plot, possivelmente via `hline.style_dashed` etc.).
- `show_label`: boolean definindo se deve mostrar um rótulo textual com o valor à esquerda da linha no gráfico. Se `false`, a linha aparece sem o marcador de valor na escala [112].

Exemplo de uso no RSI:

```
hline { value = sobrecompra, color = overbought_color, width = 1, style =
style.dash_line }
hline { value = sobrevenda, color = oversold_color, width = 1, style =
style.dash_line }
```

Isso traça linhas pontilhadas nos níveis de sobrecompra/sobrevenda configurados, com cores definidas pelo usuário [113] [114]. Observação: há formas abreviadas, como `hline(0, "", cor, largura, offset, estilo)` – em alguns exemplos usam `hline(0, "", color.gray, 1, 0, style.dash_line)` para traçar uma linha cinza pontilhada em 0, ignorando nome (passam `""` no lugar de título) [115]. A sintaxe com chaves permite nomear parâmetros, evitando confusão de ordem.

- `fill(a, b, color)` / `fill_area(a, b, title, color)` – Preenche o espaço entre duas séries ou duas linhas horizontais (ou uma série e uma linha) com uma cor ou gradiente. Essa função é útil para destacar regiões (ex: área entre duas médias móveis, ou fundo do RSI entre linhas de sobrecompra/sobrevenda). Uso:
- `fill(series1, series2, color)` colorirá a região entre os plots das séries `series1` e `series2` com a cor especificada (geralmente semi-transparente).
- Alternativamente, `fill_area` pode aceitar valores fixos: por exemplo, `fill_area(sobrecompra, sobrevenda, "", bg_color)` para preencher entre duas constantes horizontais (veja exemplo do RSI) [113] [112].

No código RSI oficial, vemos:

```
fill_area(overbought, oversold, "", bg_color)
```

Isso preenche a área entre as linhas de sobrecompra e sobrevenda com a cor semitransparente definida em `bg_color` [113]. O título foi deixado vazio, e as duas primeiras entradas podem ser valores (70 e 30, ou variáveis correspondentes). A transparência da cor é importante para não esconder completamente o gráfico abaixo – por isso geralmente define-se `bg_color` com um alpha baixo (no exemplo, `rgba(255,255,255,0.05)` = branco 5% opaco) [116].

- `rect(x1, y1, x2, y2, color)` – Desenha um **retângulo** dado dois pontos opostos (canto superior esquerdo e canto inferior direito, ou vice-versa). Os parâmetros provavelmente são coordenadas de barra (x) e valor (y). Essa função não está muito detalhada na doc resumida, mas provavelmente:
- `x1, y1`: coordenada inicial (x em unidades de índice de barra, y em valor de preço/eixo).
- `x2, y2`: coordenada final.
- `color`: cor de preenchimento do retângulo (possivelmente sem contorno definido, ou com cor).

`rect` pode ser útil para marcar regiões de tempo no gráfico (por exemplo, destacar intervalo entre duas datas ou eventos). Não vimos um uso explícito na doc apresentada, mas sabe-se que existe (listado no menu Charts and Appearance).

- `drop_plot_value(plotName, bar_index)` – Remove (descarta) um valor plotado em um determinado plot numa barra específica. Útil para "apagar" um desenho em certa barra após já plotado, sem afetar a continuidade da série. Por exemplo, se queremos que um símbolo apareça apenas na primeira ocorrência de uma sequência e não repetidamente, poderíamos após o primeiro plot "dropar" os subsequentes até a condição mudar.

A sintaxe requer o nome do plot (definido em `plot` ou implícito como título) e o índice absoluto da barra (geralmente obtido via uma variável do tipo `current_bar_id` ou calculado). A doc indica que `current_bar_id` é uma variável especial representando o índice da barra atual [117]. Assim, `drop_plot_value("NomePlot", current_bar_id)` removeria o valor plotado do plot especificado na barra atual.

Exemplo de caso de uso (conforme dúvida no StackOverflow): gerar um sinal apenas na primeira barra de uma sequência de tendência, e não em todas. Uma estratégia:

```
plotId = plot_shape(condContinuo, style.cross, location.top, color.red,
"Seq")
if condContinuo and condContinuo[1] then   -- condição true nesta barra e
também na anterior, logo não é início
    drop_plot_value("Seq", current_bar_id)  -- remove o símbolo desta barra
por não ser início de sequência
end
```

No pseudo-código acima, assumindo `condContinuo` é uma série booleana, plotamos um símbolo sempre que condContinuo é true, mas se também era true na barra anterior, significa que não é o início, então removemos o símbolo dessa barra [117] [118]. Assim, apenas o primeiro true (quando condContinuo era false na barra anterior) permanecerá com símbolo.

Observação: `drop_plot_value` atua no plot referenciado pelo nome dado. Importante nomear o plot de forma única (via parâmetro title ou retornando o id de plot se disponível). Na doc 1.1.0, mostram `drop_plot_value('i don't want', current_bar_id)` como exemplo, possivelmente removendo valores de plots chamados *'i don't want'* em certa barra [119]. É uma função de nicho, mas útil para limpar gráficos de ruído visual indesejado, especialmente em estratégias que marcariam muitas barras consecutivas.

5. Funções Matemáticas e Estatísticas (Operações Numéricas Avançadas)

Além das operações básicas (+, -, , /, ^) *aplicáveis a números e séries, a QCS traz um conjunto extenso de funções matemáticas e de estatística para uso direto nos scripts. Estas funções englobam tanto aquelas equivalentes à biblioteca padrão* `math` *do Lua, quanto funções específicas para análise técnica. Todas as funções listadas a seguir aceitam tanto valores escalares quanto séries (quando aplicável) e retornam o mesmo tipo de entrada** (por exemplo, se você passar uma série, a saída será uma série com operação aplicada elemento a elemento) [120]. Vamos agrupá-las por categoria:

Funções matemáticas básicas (álgebra e trigonometria): - `abs(x)` – Retorna o valor absoluto de *x*. Converte números negativos em positivos, mantendo zero e positivos inalterados [121] [31] . - `ceil(x)` – Arredonda *x* para cima (próximo inteiro maior ou igual). - `floor(x)` – Arredonda *x* para baixo (inteiro menor ou igual). - `round(x)` – Arredonda *x* ao inteiro mais próximo (meios possivelmente para cima). - `modf(x)` – Decompõe *x* em parte inteira e fracionária. Retorna provavelmente uma tupla (inteiro, fracção) similar ao Lua `math.modf` . - `sqrt(x)` – Raiz quadrada de *x*. Se *x* for série, computa raiz de cada valor [34] [122] . - Funções trigonométricas: `sin(x)`, `cos(x)`, `tan(x)`, e as inversas `asin(x)`, `acos(x)`, `atan(x)`. Recebem/retornam valores em radianos. - Conversão angular: - `deg(x)` – converte *x* de radianos para graus [123] . - `rad(x)` – converte *x* de graus para radianos [124] [125] . - Exponencial e logarítmica: - `exp(x)` – função exponencial *e^x*. - `log(x)` – logaritmo natural (base e) de *x*. Provavelmente também `log10` não listado explicitamente mas possivelmente disponível (não confirmado nas linhas fornecidas). - `max(a, b)` / `min(a, b)` – Máximo ou mínimo entre dois valores (ou séries) dado elemento a elemento. Também suportam mais de dois argumentos ou no contexto de séries (ex.: `max(high, ha_close, ha_open)` foi usado no exemplo Heikin Ashi para pegar o maior de três valores [32]).

Funções estatísticas e de séries: - `sum(series, n)` – Soma dos últimos *n* valores da série. Retorna uma série onde cada ponto é a soma móvel dos *n* valores até aquele ponto [126] (na verdade aparece `sum(series, period)` na lista, indicando função de math). - `median(series, n)` – Mediana dos últimos *n* valores da série. - `stdev(series, n)` – Desvio-padrão dos últimos *n* valores (provavelmente população, não deixa claro se n-1 ou n base). - `mad(series, n)` – **Mean Absolute Deviation**, ou Desvio Absoluto Médio, sobre *n* períodos (aparece listado como math) [127] [128] . - `correlation(series1, series2, n)` – Correlação (provavelmente de Pearson) entre duas séries nos últimos *n* pontos [129] . Retorna uma série (ou valor) de coeficiente de correlação que varia de -1 a 1. - **Máximos e mínimos móveis:** - `highest(series, n)` – maior valor da série nos últimos *n* períodos [94] [95] . - `lowest(series, n)` – menor valor da série nos últimos *n* períodos [130] [98] . - (A versão de passar apenas `n` já comentada antes assume high/low). - **Percentis:** - `percentile_nearest_rank(series, p)` – calcula o percentil *p* (0-100) dos valores da série, provavelmente usando método "nearest rank". - `percentile_linear_interpolation(series, p)` – percentil *p* com interpolação linear se *p* cair entre dois valores. Essas funções são úteis para estatísticas de distribuição (ex: mediana seria percentile 50%). - **Funções Prime** (indicador específico): - `lower_prime(series, period)` e `upper_prime(series, period)` – parecem referir-se às bandas "Prime". Provavelmente calculam bandas inferiores e superiores de um indicador "Prime Bands" (listado em math como funções) [131] [132] . O indicador Prime Bands possivelmente relaciona-se a um tipo de envelope adaptativo. - `prime_bands(series, period)` – Talvez retorne simultaneamente as bandas superior/inferior? Ou uma estrutura. Difícil inferir sem documentação detalhada, mas estão listadas [133] [134] . Este parece ser um caso especial de indicador incorporado nas math funcs. - **Diferenças e mudanças:** - `change(series, n)` – variação do valor da série em relação a *n* barras atrás. Essencialmente `series - series[n]`. Se n for 1, é o *delta* simples entre barras consecutivas [135] . Útil para calcular momentum, por exemplo. - Outras funções de momentum integradas: (pela lista não aparece, mas possivelmente) - `roc(series, period)` – Rate of Change em porcentagem (de fato, ROC é documentado como indicador separado, mas também aparece possivelmente como função com default period=1) [136] [137] . E.g. *ROC = ((valor atual - valor n barras atrás) / valor n barras atrás) * 100%*. - **Médias móveis e suavizações:** (esta é uma categoria grande, separaremos em seção 6 a seguir, pois a QCS implementa várias funções para diferentes tipos de médias e indicadores técnicos)

Antes de passar para indicadores técnicos, recapitulemos uma peculiaridade: as funções matemáticas padrão estão disponíveis globalmente (sem `Math.`) e aceitam séries. Caso uma função receba uma

série, ela **processa elemento a elemento**. Por exemplo, `abs(close - sma(close,20))` produzirá a série do desvio absoluto do fechamento em relação à média de 20 períodos, calculado para cada barra.

Exemplos combinando funções: - Para calcular a **Média Móvel Exponencial de 10 do volume** e comparar com volume atual:

```
emaVol = ema(volume, 10)
if get_value(volume > emaVol) then
    advise("Vol↑")  -- (conceitualmente) dar um aviso de volume acima da
média
end
```

- Para calcular o **desvio padrão** de 20 períodos do preço de fechamento e plotar duas linhas de um canal de volatilidade (ex: Bollinger Bands sem média central):

```
sd = stdev(close, 20)
plot(close + 2*sd, "BBand Upper", color.gray)
plot(close - 2*sd, "BBand Lower", color.gray)
```

Aqui usamos a propriedade que operações com séries funcionam ponto a ponto – `2*sd` duplica os valores da série de desvio padrão, então `close + 2*sd` resulta em uma série calculando *fechamento + 2*desvio*, e o `plot` traçaria a banda superior.

6. Indicadores Técnicos Embutidos e Funções de Biblioteca

Para agilizar o desenvolvimento, a Quadcode Script traz implementações prontas de muitos indicadores técnicos populares, acessíveis como funções ou através da biblioteca de indicadores. Esses indicadores englobam **Médias Móveis de vários tipos**, **Ociladores de Momento** (RSI, CCI, CMO, Estocástico etc.), entre outros. Vamos listar os principais e suas assinaturas, lembrando que todos operam em séries (geralmente no *close* ou série escolhida pelo usuário):

6.1 Médias Móveis e Suavizações

A QCS suporta diferentes algoritmos de média móvel, muitos dos quais podem ser invocados por funções dedicadas: - `sma(series, period)` – **Simple Moving Average** (Média Móvel Simples) [138] [139]. Calcula a média aritmética dos últimos *period* valores da série. Se não houver valores suficientes (ex: nas primeiras barras), retorna `nan` até acumular *period* valores [139] [140]. É provavelmente a função de média mais básica; cada valor tem peso igual. - `ema(series, period)` – **Exponential Moving Average** (Média Móvel Exponencial). Aplica ponderação exponencial decrescente aos dados – valores mais recentes têm peso maior. Usada para reagir mais rápido às mudanças que a SMA. Fórmula típica: incorpora a EMA anterior: *EMA = EMA_prev + α*(valor_atual - EMA_prev)*, com α = 2/(period+1)*. Na QCS, `ema(close, 10)` retornaria a EMA de 10 períodos do fechamento (equivalente a função do Pine Script ou outras plataformas). - `wma(series, period)` – **Weighted Moving Average** (Média Móvel Ponderada linear). Dá pesos lineares (period para o mais recente, 1 para o mais antigo). - `rma(series, period)` – **Wilder's Moving Average** (às vezes chamada SMA suavisada, ou *Rolling Moving Average*). Essa é a média móvel usada no RSI, por exemplo, que é similar à EMA porém com α = 1/period (um tipo de EMA com suavização específica). `rma` aparece na lista e provavelmente corresponde à *Smoothed Moving Average* de Wilder (método do RSI). - `ssma(series, period)` –

Possivelmente **Smoothed SMA** ou **Double/Triple Exponential**? No contexto RSI, `averages.ssma` é mencionado [141]. Suspeito que signifique *Symmetrically Weighted SMA* ou algo assim. Entretanto, dado que no script RSI eles permitem escolher `averages.ssma` como uma das opções de suavização além de rma e outras, deve ser *Smoothed Simple Moving Average* (uma técnica). Precisaria confirmar, mas podemos descrevê-la como uma média móvel suavizada (pode ser sinônimo de Wilder's? Ou outra?). - `hma(series, period)` – **Hull Moving Average**, uma média móvel de reação rápida que reduz o lag, calculada combinando WMAs de metade do período etc. - `alma(series, period)` – **Arnaud Legoux Moving Average**, uma média móvel com filtro Gaussiano ajustável para suavidade e responsividade. - `kama(series, period)` – **Kaufman's Adaptive Moving Average**, média adaptativa que ajusta fator de suavização conforme volatilidade (Efficient Ratio). - `lsma(series, period)` – **Least Squares Moving Average** (regressão linear ou Linear Regression Line). - `tma(series, period)` – Possivelmente **Triangular Moving Average** (ou T3? Triangular SMA calculada como média da SMA). - `vwma(series, period, volume_series)` – **Volume-Weighted Moving Average**. Pode ser que exista, dado que listaram VWMA. Se a assinatura padrão, precisaria do volume para ponderar. - `vidya(series, period)` – **VIDYA (Variable Index Dynamic Average)** de Chande, que ajusta peso com volatilidade. - `Rainbow MA` – provavelmente uma família de MAs coloridas (talvez função não diretamente chamada, mas há um indicador com esse nome).

No site, essas aparecem agrupadas em **Moving Averages** e têm até implementações no GitHub [142] [143]. No UI, são tratadas como indicadores completos, mas também servem como funções: Por exemplo, `EMA` tem página e possivelmente uma função `ema()`, assim como `SMA` tem função `sma()` [138]. A *documentação 1.1.0* mostra por exemplo `sma(series, period)` retornando uma série [144].

Exemplo de uso de médias: Se quisermos 3 médias de tipos diferentes:

```
ma1 = sma(close, 50)
ma2 = ema(close, 50)
ma3 = kama(close, 50)
plot(ma1, "SMA50", color.orange)
plot(ma2, "EMA50", color.blue)
plot(ma3, "KAMA50", color.green)
```

Isso desenharia três linhas de médias de 50 períodos com métodos distintos para comparação.

6.2 Osciladores de Momento e Força

A QCS implementa várias funções para osciladores famosos, que geralmente medem momentum ou força relativa:

- `rsi(series, period)` – **Relative Strength Index** [145] [146]. Retorna a série do RSI calculado sobre a série de entrada (geralmente fechamento) e período dado. O RSI resulta em um valor oscilando entre 0 e 100. No exemplo oficial, eles demonstram a implementação manual do RSI usando médias móveis do ganho e perda [43] [113], mas também fornecem a função `rsi()` para conveniência. A função provavelmente usa a média móvel de Wilder (RMA) internamente. Ex: `rsi(close, 14)` retorna a série do RSI de 14 períodos.
- `roc(series, period=1)` – **Rate of Change** em percentual [136] [137]. Retorna ((valor atual - valor de n barras atrás) / valor de n barras atrás) * 100. Se período não especificado, supõe 1

(diferença percentual entre barra atual e anterior). Útil como oscilador sem bound teórico (pode ser >100% em casos extremos).

- `cci(series, period)` – **Commodity Channel Index**. Tradicionalmente calculado sobre a *Typical Price* (HLC/3) e período, devolvendo um oscilador centrado em 0, raramente saindo do intervalo ±200. Esperamos `cci(close, 20)` ou `cci(hlc3, 20)` (não sabemos se a função exige a série típica ou calcula internamente a média typical price).
- `cmo(series, period)` – **Chande Momentum Oscillator**, semelhante ao RSI porém com escala -100 a +100 (basicamente 100 * (Soma dos ganhos - Soma das perdas) / (Soma dos ganhos + Soma das perdas)). Possivelmente implementado e chamável via `cmo(close, 14)`. De fato, aparece `cmo(series, period)` nas buscas [147].
- `stochastic(series, period)` – **Estocástico** %K. A assinatura sugere apenas um parâmetro series e period [148] [126]. Provavelmente retorna %K (talvez normalizado 0-100) calculado como *100 * (close - lowest(low, n)) / (highest(high, n) - lowest(low, n))*. Mas estranhamente só vê um series, talvez se passar `close` ele internamente pega `high` e `low`? Ou a função estocástico pode ter sobrecarga. Ainda, estocástico completo teria %D (média do %K). Talvez essa função retorne uma tabela com .%K e .%D ou somente %K e o %D deve ser calculado aplicando uma média (ex: sma) sobre o output. Sem clareza completa, mas sabe-se que há uma referência a `stochastic(series, period)` no índice [149].
- Outros osciladores possivelmente disponíveis:
- `mom(series, period)` (momentum simples = difference, parecido com roc mas não em percentagem).
- `macd(series, fast_period, slow_period, signal_period)` – se implementaram MACD, seria ideal como função retornando possivelmente um objeto ou tupla (MACD line e Signal line, e histograma). Não vimos referência explícita, mas dado que MACD é popular, ou foi deixado para o usuário compor com ema() ou está embutido.
- `ao(high, low, period_fast, period_slow)` – Awesome Oscillator (média de median price (HL2) 5 e 34 por padrão). Possível mas não certo se função existe ou se script library apenas.

Cada oscilador embutido tem normalmente uma página de documentação (vide Popular Indicators Library). Exemplo, RSI e CCI com `rsi(series, period)` e `cci(series, period)` respectivamente. Assim, um script pode chamar diretamente `rsi(close, 14)` em vez de implementar a fórmula manualmente. Isso é útil para manter o código conciso e confiável.

Exemplo prático combinando indicadores embutidos: Digamos que queremos criar uma condição de compra quando RSI < 30 (sobrevendido) *e* CCI < -100:

```
r = rsi(close, 14)
c = cci(close, 20)
cond_compra = (r < 30) and (c < -100)
plot_shape(cond_compra, shape_style.arrowup, shape_location.belowbar,
color.green, "Buy")
```

Isso utiliza as funções `rsi` e `cci` diretamente. Note que estamos usando `(r < 30)` onde `r` é uma série – essa expressão gera uma série booleana (true quando RSI daquela barra <30). Passamos essa série booleana para `plot_shape`, que aceita e interpreta valores booleanos barra a barra. Assim, desenhará seta verde nas barras onde ambas condições forem verdadeiras (graças à natureza and: produz true somente quando ambas séries booleans são true simultaneamente naquela barra).

6.3 Indicadores de Tendência e Outros

A documentação citou alguns indicadores de tendência, embora menos numerosos: - **Parabolic SAR:** Existe referência a um indicador *Parabolic SAR* na biblioteca [150]. Possivelmente há uma função `psar(step, max_step)` ou similar, porém não temos confirmação da assinatura. Parabolic SAR é mais complexo e talvez implementado como script de exemplo. Dado que só *Parabolic SAR* aparece isolado, é provável que não haja uma função simples, mas sim uma implementação no repositório que calcula recursivamente os SARs. - **Outros**: Bem, o repositório talvez tenha ADX, ATR etc. ATR poderia ser calculado via `tr` + média: e.g. `atr = ema(tr, period)` diretamente, já que definiram `tr` (true range). - **Bollinger Bands:** Podem não ter função direta, mas podem ser compostas (SMA ± N*stdev). Talvez forneçam uma função, mas não listada. - Ichimoku:** provavelmente não implementado como função, seria script separado.

De forma geral, a QCS fornece os **blocos básicos** (médias, desvios, máximos etc.) para compor indicadores. E para alguns indicadores populares, fornece funções prontas. Por isso, verifique na documentação ou repositório se existe função pronta antes de implementar do zero – isso assegura consistência com outros usuários e possivelmente melhor desempenho, além de menos linhas de código.

7. Funções de Timeframe e Multi-Ativo

A Quadcode Script também contempla cenários de análise multi-temporal e multi-ativo. Para isso, existem funções e variáveis que auxiliam a obter dados de outros timeframes ou ativos, bem como identificar o timeframe atual. Vamos explorar:

- **Identificação do Timeframe Atual:**
- `resolution` (variável): contém o timeframe do gráfico atual como string (ex.: `"5m"` para 5 minutos, `"1H"` para 1 hora, `"1D"` diário, etc.) [151] [152]. Pode ser usada para decisões lógicas ou exibição. Ex: `print("Resolução atual:", resolution)` mostraria `"Resolução atual: 5m"` se o gráfico for 5 minutos.
- Flags booleanas de timeframe:
  - `is_intrahour` – true se o timeframe é menor que 1 hora (ou seja, gráfico em minutos) [153] [154].
  - `is_intraday` – true se o timeframe é < 1 dia (ou seja, minutos ou horas) [155] [156].
  - `is_daily` – true se o timeframe é diário (ou possivelmente incluindo superiores? A doc do 1.1 estava confusa, mas tende a indicar *exatamente diário*) [157] [158].
  - `is_weekly` – true se o gráfico é semanal [159] [160].
  - `is_monthly` – true se mensal [161] [162].
  - `is_dwm` – true se é diário, semanal ou mensal (qualquer um dos três) [163] [164]. Essencialmente, `is_dwm` agrupa os timeframes "de posição" (>=1 dia) como verdadeiros e intraday como falsos [165] [166]. Isso pode ser útil para scripts que ajustam parâmetros para gráficos diários vs intradiários de uma vez só.

Essas variáveis podem ser utilizadas, por exemplo, para ajustar automaticamente um parâmetro:

```
length = 14
if is_dwm then
    length = 45  -- se gráfico diário ou maior, usar período 45 em vez de 14
```

```
end
sma_long = sma(close, length)
```

Conforme o exemplo da documentação, eles sugerem algo nessa linha, alterando uma variável de período se `timeframe.is_daily()` (ou is_dwm) for true [167] [168]. *Nota:* O acesso talvez seja via objeto `timeframe` como mostrado (`timeframe.is_daily()`), mas na implementação final parece que expuseram como variáveis (no 1.1 doc, `is_daily` foi chamado de variável mas exemplo usou `timeframe.is_daily()`. É possível que usem um namespace `timeframe.*`). Verifique a versão atual; assumiremos que pode ser acessado diretamente como `is_daily` etc., pois a listagem sugere são variáveis globais booleans.

- **Multi-ativo e Símbolos:**
- `current_ticker_id` (variável): representa o ID numérico do ativo atual no qual o script está rodando [169] [170]. Cada símbolo negociável (par de moeda, ação, etc.) tem um ID único na plataforma. Essa variável permite referenciar o ativo corrente de forma programática (por exemplo, para comparações ou para chamar `security`).
- `get_ticker_id("TICKER")`: retorna o ID do ticker com o código fornecido (string). Se o ticker não for encontrado, retorna nil [171] [172]. Por exemplo, `eurusd_id = get_ticker_id("EURUSD")` buscaria o ID do par EUR/USD. Esse ID é necessário para requisitar dados de outro ativo via `security`.
- `security(ticker_id, resolution_str)`: função para obter dados de outro ativo ou timeframe [173] [174]. Retorna uma tabela contendo séries do ativo/timeframe solicitado [175] [176]. Parâmetros:
  - `ticker_id`: o ID do ativo (pode usar `current_ticker_id` ou um obtido por `get_ticker_id`).
  - `resolution_str`: string representando o timeframe desejado, e.g. `"5m"`, `"1H"`, `"1D"`, etc. Existe também atalho `"Current"` ou uso de constante `current_ticker_id` combinado com timeframe para pegar o ativo atual em outro timeframe.

  A função devolve uma estrutura do tipo:

```
{
  close = <series>,
  open = <series>,
  high = <series>,
  low = <series>,
  volume = <series>,
  open_time = <series>,
  close_time = <series>
}
```

  Ou seja, um *"mini-candle histórico"* daquele ativo/timeframe, que pode ser indexado. Exemplo de uso:

```
other = security(eurusd_id, "5s")
plot(other.close, "EURUSD 5s Close", color.orange)
```

Isso, se suportado, plotaria o fechamento de EURUSD em 5 segundos no gráfico atual. No exemplo da doc: `security_data = security(current_ticker_id, "5s")` `plot(security_data.close)` [177] – pega dados do mesmo ativo no timeframe de 5 segundos e plota o fechamento (provavelmente num gráfico de timeframe maior). Essa técnica permite multi-timeframe: você pode acessar, por exemplo, o fechamento diário e semanal enquanto está num gráfico de 1h.

Atenção: Ao usar `security`, considere que ao solicitar timeframes superiores, a série retornada terá menos pontos (um valor diário para várias barras horárias). Provavelmente a QCS repete o último valor conhecido até atualizar (similar ao TradingView `security` semantics). Isso não foi detalhado, mas por precaução, se for comparar timeframe maior com atual, usar `get_value()` para pegar o valor sincro na barra atual.

- **Avisos (Advice):**
  - `advise(signal)` – função para emitir um aviso de compra/venda/neutro. De acordo com doc:
  - Param `signal`: um valor do *enum* `advice.*` representando estado do conselho [178]. Possíveis valores: `advice.buy`, `advice.sell`, `advice.neutral` [179]. Internamente, isso gera notificações no traderoom (ex.: setinha ou texto "Buy"/"Sell").

O uso típico seria dentro de condições:

```
if ... then
    advise(advice.buy)
elseif ... then
    advise(advice.sell)
else
    advise(advice.neutral)
end
```

No exemplo da documentação, curiosamente usaram strings "Buy"/"Sell"/"Wait" diretamente [180], mas o parâmetro formal indicado é do tipo enum. Pode ser que `advise("Buy")` seja aceito e interpretado como `advice.buy`. De qualquer forma, convém usar os enums se disponíveis (podem ser membro de table advice ou constants).

Observação: `advise` não executa ordens, apenas sinaliza visualmente ao usuário uma recomendação. É uma ferramenta de criação de *indicadores de sinal*.

8. Tratamento de Exceções, Valores Nulos e Depuração

Para garantir robustez dos scripts e auxiliar no desenvolvimento, a QCS oferece algumas funções utilitárias para lidar com valores inválidos, depurar logs e impor requisitos de versão.

- `na(value)` – Verifica se um determinado valor está disponível (não é `nan` nem `nil`). Retorna booleano: `true` se *value* for `nan` ou `nil`, indicando "não disponível" (apesar de literalmente "na" significar *not available*, a doc fraseou de modo confuso, mas pelo exemplo: `if na(div) then print("Value is not a number") end` [181] [182], fica claro que `na(x)` retorna **true quando x é nan/nil**). Use isso para checar se um cálculo resultou em valor numérico válido antes de utilizá-lo.

- `nz(value, default=0)` – Substitui valores "não números" (*nan* ou *nil*) pelo valor default fornecido (ou 0 se não especificado) [183] [184]. Essa função tem comportamento:

- Se `value` for série, devolve uma série onde todos `nan` foram trocados pelo default.
- Se `value` for escalar: retorna `value` se este não for `nil` ou `nan`; caso seja, retorna o `default`.
- O parâmetro `default` pode ser série também – se for série, utiliza o valor atual dessa série quando preciso substituir [185].

Exemplos:

```
x = close[10]        -- nas primeiras 10 barras, close[10] será nil
y = nz(x, close)     -- se x for nil, retorna close atual
z = nz(5/0)
-- 5/0 resulta nan, nz substitui pelo default 0, então z = 0
```

No segundo exemplo, ao acessar 10 barras atrás, nas barras iniciais `x` é nil; `nz(x, close)` então pega `close` atual no lugar (que pode ser ou não boa lógica, mas exemplifica uso de série default).

- `fixnan(series)` – Para séries numéricas, substitui valores `nan` na série pelo último valor válido anterior [186] [187]. Isso é útil para evitar interrupções em cálculos acumulativos. Por exemplo, se você divide por zero e obtém nan em uma barra, `fixnan(resultado)` fará com que naquela barra o resultado passe a ser igual ao último valor não-nan anterior (mantém a continuidade) [187] [188]. Útil em indicadores como RSI ou outros onde divisões por zero podem ocorrer no início mas se deseja propagar ultimo valor válido.

Exemplo:

```
div = close / open   -- se open for 0 em alguma barra (hipotético, preços
geralmente não 0), div seria inf ou nan
div_safe = fixnan(div)
```

Assim, `div_safe` carrega a divisão, mas se houver nan, repete o valor anterior (ou zero se logo no início sem anterior).

- `assert(condition, message)` – Se a condição for false, **interrompe a execução** do script e registra a mensagem de erro fornecida [189] [190]. É útil durante desenvolvimento para garantir pressupostos (e.g., parâmetros válidos). Ex: `assert(period > 0, "Period must be positive")`. No caso de falha, o script pararia e no log do terminal apareceria "Period must be positive", ajudando o desenvolvedor.

- `error(message)` – Sem condição: sempre que chamado, para o script imediatamente e exibe a mensagem de erro no log [191] [192]. Use para casos em que detectou uma situação inválida e não quer continuar. Ex: `if period < 1 or period > 500 then error("Period out of range!") end`.

- `print(...args)` – Escreve no log do script todos os argumentos fornecidos (de forma concatenada ou separada) [193] [194]. Essa função é vital para depuração, já que permite inspecionar valores durante a execução. Por exemplo, `print("Valor RSI=", get_value(r))` imprimirá o texto e o valor numérico atual do RSI no console de scripts. **Obs.:** Use com parcimônia se o script roda em tempo real, pois excesso de prints pode afetar performance ou inundar o log. Uma dica: condicione prints a certas barras (ex: `if index_range(0) % 100 == 0 then print(...) end` para logar a cada 100 barras).

- `version(min_version_number)` – Especifica a versão mínima do QCS necessária para rodar o script [195] [196]. Isso é útil se você usar funções recentes adicionadas numa certa versão. Por exemplo, `version(1.1)` assegura que apenas plataformas com QCS 1.1.0 ou superior executarão o script (provavelmente se rodado em versão inferior, dará erro informando versão incompatível). Também possivelmente retorna a versão atual em uso, que pode ser armazenada se útil. A sintaxe de exemplo: `current_version = version(1.1.0)` [196]. Em geral, coloque `version(x.y)` no topo do script para travar a compatibilidade.

Além dessas, lembre que se um valor não existe em certa barra (ex: acesso a `serie[10]` nas primeiras barras), ele será `nil`. Ao realizar operações com `nil` ou `nan` (ex: somar, comparar, etc.), geralmente resultará em `nan` ou erro. Portanto, funções como `nz` e `fixnan` ajudam a contornar esses buracos de dados.

Para identificar barra atual ou número de barras: - QCS provavelmente tem uma variável `current_bar_id` (mencionada em contexto do `drop_plot_value`) que dá o índice da barra corrente (talvez 0 na primeira barra histórica incrementando até N na atual). - Pode haver também `last_bar_index` etc. Não vimos explicitamente, mas dado que `index_range(1)` retorna 1 na barra inicial e incrementa, você pode usar `get_value(index_range(1))` para obter o índice da barra atual (se index_range(1) gera 1,2,3,...). - Se precisar do total de barras histórico, talvez usar `bars_count = get_value(index_range(1))` no final (difícil dizer se index_range start=1 ou 0; doc sugere start default 1 e series[0] = start_index, então na última barra `series[i] = start_index + i`. Então se start=1 e última barra i = N-1, valor = N = total de barras).

9. Aplicações Práticas: Estratégias Automatizadas em IQ Option e Avalon

Tendo coberto a sintaxe e funções da QCS, vamos ao contexto de uso principal: implementar **estratégias de trading automatizadas e indicadores customizados** nas plataformas da família

Quadcode, notadamente IQ Option (uma corretora conhecida de opções e CFDs) e Avalon (outras plataformas white-label de negociação possivelmente baseadas na mesma tecnologia QCS).

Essas plataformas permitem que usuários avançados insiram scripts QCS para: - Criar **indicadores personalizados** que aparecem sobre o gráfico ou em painéis inferiores (como exemplificado com médias, RSI modificado, etc.). - Desenvolver **ferramentas de sinalização** (indicadores que disparam sinais de compra/venda, usando `plot_shape`, `advise` etc.). - Em certos casos, criar **robôs ou estratégias automatizadas** que **possam executar ordens** ou ao menos alertar o trader para executar (conforme a Glossário: "Scripts podem realizar várias funções, como abrir ou fechar ordens..." [2]). A habilidade de executar ordens pode depender de permissões específicas ou ambientes de teste nas plataformas – portanto, recomenda-se consultar a documentação específica da corretora sobre *"script de auto negociação"*.

9.1 Integração na Plataforma IQ Option

Na IQ Option, há uma seção de *Scripts* ou *Indicadores Personalizados* onde o usuário pode colar o código QCS. O fluxo típico: 1. **Desenvolvimento:** Escrever e testar localmente seu script QCS em um editor de texto (pode usar a sintaxe .lua ou .txt). 2. **Importação:** No painel do traderoom, usar *Import Script* para carregar o arquivo .lua/.txt do script [197] [198] . Ou colar o código via opção *Add Manually* que abre um editor interno [199] . 3. **Execução:** Após salvar, clicar em *Add to chart* para aplicar o indicador/estratégia no gráfico atual [200] . Se for indicador overlay, aparecerá sobre as velas; se separado, em painel abaixo. 4. **Parâmetros:** O script, se bem estruturado com `input`, terá um diálogo de configurações permitindo ao usuário alterar os valores (períodos, cores etc.) sem editar código. Esses inputs aparecem organizados conforme `input_group` definidos [201] [39] . 5. **Visualização:** Os plots definidos (linhas, formas, candles) são renderizados em tempo real. O trader pode observar, por exemplo, uma linha de média ou setas de sinal aparecendo conforme o mercado evolui. 6. **Sinais e Ordens:** Se o script usar `advise(advice.buy/sell)`, possivelmente a plataforma exibirá setas ou alertas na tela (ex.: "Buy" ou "Sell") quando as condições ocorrerem. Esses conselhos também podem ser usados para acionar, por exemplo, notificações push, embora isso seja mais no lado plataforma do que do script. - Quanto a ordens automáticas: IQ Option tradicionalmente não permitia bots totalmente automáticos do lado cliente; no entanto, se a QCS agora suporta abrir/fechar ordens, é provável que seja via `advise` integrado a funcionalidade de negociação (por exemplo, um *script de negociação* rodando na conta prática ou via APIs internas). **É prudente confirmar**: no Glossário, "abrir ou fechar ordens" sugere que *estratégias criadas via script podem de fato transacionar*. Pode ser restrito a certos ambientes ou torneios de bots. - Caso seja suportado, talvez exista funções não cobertas acima, como `buy()` ou `sell()` ou `place_order(...)`. Porém, a documentação pública foca em indicadores. É possível que a execução de ordens seja obtida ligando o `advise` a sistemas de auto-trade do servidor.

Considerações de Performance: Os scripts QCS rodam localmente dentro do ambiente da corretora (no browser ou app do trader). É crucial que sejam eficientes para não travar o gráfico. Algumas dicas: - Use variáveis `local` para cálculos temporários em loops ou funções repetidas para evitar pesquisa global lenta. - Prefira operações vetoriais e funções built-in (escritas em C++ no backend) sobre loops manuais em Lua. - Evite escrever em logs excessivamente (apenas para debug, depois remover ou condicionar). - Atente ao *memory leak*: se criar estruturas grandes (tabelas acumulando dados), libere ou reuse-as. Felizmente, QCS é reativo barra a barra e não precisa armazenar muitos dados manualmente (os históricos de séries são gerenciados pelo motor). - **Timers em scripts:** Se precisar de algum temporizador ou espera (ex: checar condição a cada X segundos independentemente de novas barras), considere que QCS é orientado a barras – ele recalcula quando chega nova barra ou novo tick. Em gráficos time-based, isso significa recalcular quando a barra fecha. Para granularidade abaixo do timeframe do gráfico, a técnica é usar `security` de timeframe menor. Não há função de `sleep()` ou similar, pois isso congelaria o script. Então "timers mal sincronizados" refere-se a cuidado ao usar

resoluções distintas; ex: não use `second` series para tentar algo a cada X segundos a menos que esteja no gráfico de 1s ou similar, senão `second` só atualiza por barra. - **Execução contínua:** Lembre que indicadores recalculam continuamente; se quiser executar certa lógica apenas quando a barra fecha, use `close_time` ou comparativo de tempo, ou funções de timeframe – mas QCS provavelmente só computa em fechamento de barra por padrão (depende se a plataforma atualiza script on every tick or on candle close – pelo safe side, supomos ao menos em cada fechamento e possivelmente intrabar se ticks considerados, mas não documentado claramente).

Gerenciamento de ordens e risco: Em estratégias automáticas, uma vez identificada condição de entrada, deveria haver também lógica de saída (stop loss, take profit ou condição contrária). Como a QCS não mostra explicitamente funções de trade, pode ser que estratégias completas sejam implementadas vinculando `advise` a algum sistema ou lendo valores indicativos. Caso o desenvolvedor esteja criando um *robô cliente*, ele poderia ler as variáveis/sinais do indicador e acionar ordens via API ou manual. Por exemplo, muitos usam indicadores custom que plotam setas e usam programas externos para ler esses sinais e enviar ordens. Alternativamente, se QCS puder "clicar", seria interno.

9.2 Uso na Plataforma Avalon e Outras (White-labels Quadcode)

Avalon possivelmente é outra plataforma de trading alimentada pelo engine Quadcode (pode ser um nome fictício para um white-label). Do ponto de vista do script QCS, o comportamento deve ser idêntico, pois o engine é o mesmo. As diferenças seriam: - Ativos disponíveis (os tickers e IDs podem variar se forem mercados diferentes). - Possíveis restrições ou extensões de usabilidade (ex: se Avalon permite trading automatizado, talvez exponha funções de ordem). - Interface, mas o script em si permanece igual.

Portanto, ao portar scripts entre IQ Option e Avalon, geralmente não é necessário alterar código (exceto talvez nomes de tickers ou detalhes cosméticos). Ainda assim: - **Teste em conta demo**: Sempre teste scripts em ambientes simulados. Especialmente se há qualquer automatização de ordem. - **Adaptação de parâmetros**: O perfil de ativos pode mudar (ex: período ideal para EURUSD pode não ser para um cripto), então torna-se útil expor inputs configuráveis em vez de valores fixos no código. - **Confiabilidade**: Verifique se as funções comportam-se conforme esperado. Por exemplo, `security` para outros timeframes – checar se realmente traz os dados corretos. - **Limites de histórico**: Plataformas podem limitar quantas barras históricas carregam. Um indicador que precise de 1000 barras pode não ter isso disponível imediatamente – muitas vezes QCS scripts calculam historicamente no load, mas se pedir demais pode ser truncado. Considere limitar períodos ou indicar no manual do usuário que certos indicadores exigem bastante histórico (ex: indicadores de ciclo, Fourier, etc.)

9.3 Exemplo de Estratégia Completa

A seguir montamos um exemplo simples de estratégia, integrando vários aspectos discutidos:

Exemplo: Estratégia de Cruzamento de Médias com Confirmação de Volume

- *Ideia:* Quando a média móvel rápida cruza acima da lenta, e houver aumento de volume significativo, gerar um sinal de **Compra**. Quando cruza abaixo, sinal de **Venda**. Só sinalizar no momento do cruzamento inicial (não em barras seguintes da tendência).
- *Parâmetros:* Período da média curta, período da média longa, multiplicador de volume.
- *Visualização:* Plotar as duas MAs, colorir candles verdes/vermelhos quando acima/abaixo da lenta, e marcar setas nas entradas. Opcional: usar `advise` para notificar Buy/Sell.

```
instrument { name="CrossVolume Bot", overlay=true }

-- Inputs
fast_period = input(5, "Fast MA Period", input.integer, 1)
slow_period = input(20, "Slow MA Period", input.integer, 1)
vol_factor = input(1.5, "Volume factor", input.double, 0.1, 10.0, 0.1,
    tooltip="Mínimo fator multiplicativo do volume vs média para confirmar
sinal")

input_group {
    "Colors",
    bull_color = input { default = "#2E77D1", type = input.color, name="Bull
Candle" },
    bear_color = input { default = "#E74C3C", type = input.color, name="Bear
Candle" }
}

-- Cálculos principais
ma_fast = ema(close, fast_period)
ma_slow = ema(close, slow_period)

-- Volume médio para comparação (ex: média simples do volume de slow_period)
vol_avg = sma(volume, slow_period)
vol_cond = volume > vol_factor * vol_avg   -- condição: volume atual > X
vezes volume médio

-- Cruzamentos de médias:
cross_up   = ma_fast > ma_slow and ma_fast[1] <= ma_slow[1]   -- cruzamento
altista nesta barra
cross_down = ma_fast < ma_slow and ma_fast[1] >= ma_slow[1]   -- cruzamento
baixista nesta barra

-- Sinais de compra/venda confirmados por volume
buy_signal  = cross_up   and get_value(vol_cond)
-- bool: cruzou p/ cima E volume ok
sell_signal = cross_down and get_value(vol_cond)    -- bool: cruzou p/ baixo
E volume ok

-- Plotagem das médias
plot(ma_fast, "MA Fast", color.yellow, 1)
plot(ma_slow, "MA Slow", color.blue, 1)

-- Colorir candles conforme posição em relação à MA lenta
candle_color = iff(close > ma_slow, bull_color, bear_color)
plot_candle(open, high, low, close, "Price", candle_color)

-- Plotar setas de sinal apenas na barra de sinal inicial
plot_shape(buy_signal,  shape_style.arrowup,   shape_location.belowbar,
bull_color, "BUY", shape_size.normal)
plot_shape(sell_signal, shape_style.arrowdown, shape_location.abovebar,
```

```
bear_color, "SELL", shape_size.normal)

-- Recomendações (aviso no painel)
if buy_signal then
    advise(advice.buy)
elseif sell_signal then
    advise(advice.sell)
else
    advise(advice.neutral)
end
```

Explicação do Exemplo: Calculamos EMA rápida e lenta. `vol_cond` verifica se volume atual excede 1.5x (padrão do input) a média de volume (SMA) – isso serve como filtro: só consideraremos cruzamentos com volume acima da média em 50%. `cross_up` e `cross_down` detectam o instante do cruzamento (comparando com valores da barra anterior). As variáveis `buy_signal` e `sell_signal` então combinam cruzamento + filtro de volume. Usamos `get_value(vol_cond)` porque `vol_cond` é série booleana e precisamos do valor atual para a expressão booleana final – como estamos já numa expressão booleana do Lua (and/and), talvez não precisasse, mas é seguro garantir que seja bool.

Plotamos as EMAs em amarelo e azul. Colorimos as velas conforme se estão acima ou abaixo da MA lenta (como tendência de fundo). Plotamos setas: para compra abaixo da barra (verde), para venda acima da barra (vermelha). Notem que não evitamos múltiplos sinais seguidos explicitamente aqui – porém, devido ao cruzamento ser momentâneo, não vai ter repetição a cada barra (somente se cruzar de novo, mas isso implica antes cruzou contrário). Poderíamos usar `drop_plot_value` se quiséssemos suprimir sinais redundantes, mas nesse caso não é necessário.

Finalmente, `advise(...)` indica Buy/Sell/Neutral conforme sinal. Assim, no traderoom, o indicador possivelmente mostrará "Buy" ou "Sell" no painel de instrumento quando ativo.

Execução: Quando aplicar esse script, o trader verá duas médias no gráfico, velas coloridas, e no momento que a média rápida cruza a lenta pra cima com volume alto, aparecerá uma seta "BUY" abaixo daquela barra (e possivelmente o terminal exibirá um aviso de compra). O `advise.neutral` nos outros momentos evita que fique mostrando o último sinal (dependendo de como a interface lida, `neutral` talvez limpe ou mostre "Neutral").

Nota de segurança: Estratégias automatizadas devem ser bem testadas. No exemplo acima, colocamos neutral toda vez que não há sinal; se a plataforma interpretar isso como "feche posições", poderia ter implicações (mas creio que `advise.neutral` é só para exibir status neutro). Sempre leia a doc específica da corretora se planeja automação real.

9.4 Melhores Práticas Finais

Para fechar esta documentação, destacamos algumas **boas práticas e dicas** ao trabalhar com Quadcode Script:

- **Estruture e documente seu código:** Use comentários `--` para explicar partes complexas, especialmente se o script for ser mantido ou compartilhado com outros desenvolvedores. Mantenha consistência de nomenclatura (por exemplo, use inglês ou português nas variáveis, mas de forma uniforme).

- **Limite de recursos:** Evite que seu script consuma tempo excessivo de cálculo. Indicadores pesados (FFT, loops aninhados grandes) podem travar o gráfico ou ser interrompidos. Faça uso de funções built-in que são otimizadas em C++.
- **Testes de validação:** Use `assert` para garantir que inputs estão dentro do esperado (por exemplo, evitar período 0 ou negativo, conforme demonstrado).
- **Nil e nan propagation:** Esteja atento aos *nan*. Um nan infecta qualquer cálculo subsequente (qualquer operação com nan resulta nan). Então utilize `nz` para substituí-los por 0 ou outro default quando apropriado, ou `fixnan` para séries. Especialmente no início da série, muitas funções dão nan (por exemplo, antes de ter dados suficientes pro período).
- **Atualizações da linguagem:** Fique atento a atualizações do QCS (versões novas podem adicionar funções ou mudar comportamento). Por isso, mencionar `version(x.y)` garante que se alguém tentar usar seu script em versão muito antiga será alertado. Monitore o repositório oficial [202] e changelogs.
- **Depuração incremental:** Ao desenvolver, teste partes isoladas. Por exemplo, primeiro plote as médias para ver se estão corretas, depois implemente o sinal, etc. Use `print` no modo debug para checar valores de séries em certos pontos (mas remova ou desative prints intensivos na versão final).
- **Uso de `security`:** Se usar dados multi-timeframe ou multi-ativo, saiba que isso pode aumentar a carga e possivelmente requer conexão estável (pois a plataforma puxa mais dados). Use com moderação e, se possível, evite chamadas muito frequentes (por ex, não aninhe security dentro de loop). Em vez disso, faça uma chamada e armazene o resultado em variável, reutilizando-o.
- **Compatibilidade Multi-ativo:** Se seu script usa tickers fixos (via `get_ticker_id("ABC")`), considere torná-los inputs para flexibilidade. Ou ao menos documente que está fixo para certo ativo.

Com essas práticas, você garantirá que seus scripts QCS sejam **claros, performáticos e confiáveis** – características essenciais para um desenvolvedor sênior nessa área.

Conclusão

A linguagem Quadcode Script une a facilidade do Lua a um poderoso conjunto de funções orientadas à análise técnica, permitindo implementar indicadores customizados e estratégias de trading dentro de plataformas como IQ Option e Avalon. Abordamos nesta documentação todos os recursos oficiais conhecidos da QCS: desde sua sintaxe básica e tipos de dados (particularmente o conceito de *series* de preços) até as funções mais avançadas para cálculos técnicos, manipulação de gráficos e integração multi-timeframe/multi-ativo. Destacamos as **peculiaridades sintáticas** (como a omissão do prefixo `Math.` e o tratamento especial de `0` em certas funções) que podem surpreender desenvolvedores acostumados a JavaScript ou mesmo ao comportamento padrão do Lua. Também apresentamos exemplos funcionais e padrões de código para ilustrar o uso correto de cada elemento – incluindo casos de uso reais em lógica de trading.

Um desenvolvedor **QSC Sênior** deve agora estar apto a: - **Consultar rapidamente** esta referência para lembrar a sintaxe exata de uma função (por exemplo, parâmetros de `plot_shape` ou `security`). - **Evitar erros comuns** – como usar série em if sem `get_value`, chamar `Math.abs` em vez de `abs`, ou esquecer de inicializar inputs adequadamente. - **Aplicar QCS em estratégias reais**, integrando sinais com elementos visuais e possivelmente automação de ordens, entendendo as limitações e responsabilidades envolvidas.

Por fim, lembramos que o sucesso de implementar estratégias automatizadas não depende apenas da linguagem, mas de **teste e avaliação robusta** dos indicadores e condições definidas. Utilize

ferramentas de backtesting se disponíveis, ou execute seu script em modo simulado antes de confiar capital real a ele. A QCS fornece os meios – cabe ao desenvolvedor usá-los com diligência, combinando conhecimento técnico com prudência de trading.

Recursos para Aprofundamento: Para complementar esta documentação, consulte o repositório oficial **QuadCodeScript-Library** no GitHub [203], que contém código fonte de vários indicadores embutidos – uma ótima forma de ver exemplos práticos avançados escritos em QCS. A comunidade de usuários em fóruns e issues do GitHub também pode ajudar com dúvidas específicas e compartilhamento de estratégias escritas em QCS. Com a base consolidada aqui e aprendizado contínuo, você poderá aproveitar todo o potencial da Quadcode Script em suas iniciativas de trading automatizado e análise técnica customizada. Boa programação e bons trades!

---

[1] [3] Script Basics | Quadcode Script
https://quadcode-tech.github.io/quadcodescript-docs/Script-Basics/

[2] Glossary | Quadcode Script
https://quadcode-tech.github.io/quadcodescript-docs/Glossary/

[4] [5] [35] [38] instrument | Quadcode Script
https://quadcode-tech.github.io/quadcodescript-docs/Key-Script-Components/instrument/

[6] [7] [39] [201] input, input_group | Quadcode Script
https://quadcode-tech.github.io/quadcodescript-docs/Key-Script-Components/input,-input_group/

[8] [11] [32] [90] [91] [104] [200] Your First Indicator | Quadcode Script
https://quadcode-tech.github.io/quadcodescript-docs/Your-First-Indicator/

[9] [92] [93] [105] [106] [107] [108] [109] [110] [111] plot_shape | Quadcode Script
https://quadcode-tech.github.io/quadcodescript-docs/Key-Script-Components/Charts-and-Appearance/plot_shape/

[10] [12] [13] [15] [16] [19] built in types | Quadcode Script
https://quadcode-tech.github.io/quadcodescript-docs/Key-Script-Components/built-in-types/

[14] [87] [88] [89] [94] [95] [96] [97] [98] [99] [100] [101] [102] [103] [117] [118] [126] [130] [147] [148] [149] Binary Options Hunt | PDF
https://www.scribd.com/document/694573467/Binary-Options-Hunt

[17] [18] [20] [21] [22] [23] [24] [25] [26] [27] [44] [45] [46] [47] [48] [49] [50] [51] [52] [53] [54] [55] [56] [57] [58] [59] [60] [61] [62] [63] [64] [65] [66] [67] [68] [69] [70] [71] [72] built in series | Quadcode Script
https://quadcode-tech.github.io/quadcodescript-docs/Key-Script-Components/built-in-series/

[28] [29] [30] get_value | Quadcode Script
https://quadcode-tech.github.io/quadcodescript-docs/Functions/Built-in-Series-Functions/get_value/

[31] [120] [121] abs | Quadcode Script
https://quadcode-tech.github.io/quadcodescript-docs/Functions/Maths-Functions/abs/

[33] [34] [122] [123] [124] [125] [127] [128] [129] [131] [132] [133] [134] [135] Maths Functions | Quadcode Script
https://quadcode-tech.github.io/quadcodescript-docs/Functions/Maths-Functions/

[36] [119] trading - IQOption Script | QuadCode - Stack Overflow
https://stackoverflow.com/questions/77230197/iqoption-script-quadcode

[37] [40] [41] [138] [139] [140] SMA | Quadcode Script
https://quadcode-tech.github.io/quadcodescript-docs/Popular-Indicators-Library/Moving-Averages/SMA/

42 43 112 113 114 116 141 145 146 RSI | Quadcode Script
https://quadcode-tech.github.io/quadcodescript-docs/Popular-Indicators-Library/Momentum/RSI/

73 74 is_series | Quadcode Script
https://quadcode-tech.github.io/quadcodescript-docs/Functions/Built-in-Series-Functions/is_series/

75 76 is_value | Quadcode Script
https://quadcode-tech.github.io/quadcodescript-docs/Functions/Built-in-Series-Functions/is_value/

77 78 isnan | Quadcode Script
https://quadcode-tech.github.io/quadcodescript-docs/Functions/Built-in-Series-Functions/isnan/

79 80 get_name | Quadcode Script
https://quadcode-tech.github.io/quadcodescript-docs/Functions/Built-in-Series-Functions/get_name/

81 82 make_series | Quadcode Script
https://quadcode-tech.github.io/quadcodescript-docs/Functions/Built-in-Series-Functions/make_series/

83 84 index_range | Quadcode Script
https://quadcode-tech.github.io/quadcodescript-docs/Functions/Built-in-Series-Functions/index_range/

85 86 conditional — Quadcode Script 1.1.0 documentation
https://quadcode-tech.github.io/quadcodescript-docs/api/misc/conditional.html

115 136 137 ROC | Quadcode Script
https://quadcode-tech.github.io/quadcodescript-docs/Popular-Indicators-Library/Momentum/ROC/

142 143 150 Popular Indicators Library | Quadcode Script
https://quadcode-tech.github.io/quadcodescript-docs/Popular-Indicators-Library/

144 sma — Quadcode Script 1.1.0 documentation
https://quadcode-tech.github.io/quadcodescript-docs/api/averages/sma.html

151 152 resolution | Quadcode Script
https://quadcode-tech.github.io/quadcodescript-docs/Functions/Timeframes/resolution/

153 154 is_intrahour | Quadcode Script
https://quadcode-tech.github.io/quadcodescript-docs/Functions/Timeframes/is_intrahour/

155 156 is_intraday | Quadcode Script
https://quadcode-tech.github.io/quadcodescript-docs/Functions/Timeframes/is_intraday/

157 158 167 is_daily | Quadcode Script
https://quadcode-tech.github.io/quadcodescript-docs/Functions/Timeframes/is_daily/

159 160 is_weekly | Quadcode Script
https://quadcode-tech.github.io/quadcodescript-docs/Functions/Timeframes/is_weekly/

161 162 is_monthly | Quadcode Script
https://quadcode-tech.github.io/quadcodescript-docs/Functions/Timeframes/is_monthly/

163 164 165 166 168 is_dwm | Quadcode Script
https://quadcode-tech.github.io/quadcodescript-docs/Functions/Timeframes/is_dwm/

169 170 current_ticker_id | Quadcode Script
https://quadcode-tech.github.io/quadcodescript-docs/Functions/Timeframes/current_ticker_id/

171 172 get_ticker_id | Quadcode Script
https://quadcode-tech.github.io/quadcodescript-docs/Functions/Asset-Functions/get_ticker_id/

173 174 175 176 177 security | Quadcode Script
https://quadcode-tech.github.io/quadcodescript-docs/Functions/Asset-Functions/security/

178 179 180 advise | Quadcode Script
https://quadcode-tech.github.io/quadcodescript-docs/Functions/Asset-Functions/advise/

181 182 na | Quadcode Script
https://quadcode-tech.github.io/quadcodescript-docs/Functions/Troubleshooting-and-Debugging/na/

183 184 185 nz | Quadcode Script
https://quadcode-tech.github.io/quadcodescript-docs/Functions/Troubleshooting-and-Debugging/nz/

186 187 188 fixnan | Quadcode Script
https://quadcode-tech.github.io/quadcodescript-docs/Functions/Troubleshooting-and-Debugging/fixnan/

189 190 assert | Quadcode Script
https://quadcode-tech.github.io/quadcodescript-docs/Functions/Troubleshooting-and-Debugging/assert/

191 192 error | Quadcode Script
https://quadcode-tech.github.io/quadcodescript-docs/Functions/Troubleshooting-and-Debugging/error/

193 194 print | Quadcode Script
https://quadcode-tech.github.io/quadcodescript-docs/Functions/Troubleshooting-and-Debugging/print/

195 196 version | Quadcode Script
https://quadcode-tech.github.io/quadcodescript-docs/Functions/Troubleshooting-and-Debugging/version/

197 198 199 Script Options | Quadcode Script
https://quadcode-tech.github.io/quadcodescript-docs/Script-Options/

202 quadcode-tech/quadcodescript-library - GitHub
https://github.com/quadcode-tech/quadcodescript-library

203 quadcode-tech/quadcodescript-docs - GitHub
https://github.com/quadcode-tech/quadcodescript-docs