cálculo discreto e programação funcional

@o-santi

na aula de cálculo numérico que puxei esse período, tivemos que entregar uma biblioteca de funções gerais do tópico, incluindo derivadas e integrais discretas. como achei o tópico muito interessante (e com gráficos muito bonitos), decidi trazer algumas coisinhas que aprendemos para cá.

qualquer um que minimamente prestou atenção nas aulas de cálculo 1 e 2 já viram as respectivas definições de integral e derivada contínuas, mas como exatamente transformamo-as em funções calculáveis pelo nosso computador? essa pergunta não só não é 100% clara, como muitas vezes toca em tópicos ainda sombrios na área de computação e matemática, como por exemplo representação de números reais no computador, ou representação de limites como o valor de uma soma infinita.

uma solução para isso é discretizarmos os intervalos. ao invés de assumirmos que o eixo x das abscissas é o conjunto dos reais (ou seja, infinitos números), escolhemos um conjunto finito de números que será o intervalo pelo qual nós iremos integrar (ou derivar). mas o que isso de fato significa?

1 derivadas e programação funcional

consideraremos todos os intervalos de agora em diante como conjuntos (listas, arrays) de pontos (valores inteiros ou floats). isso se dá pois muitas vezes em cálculo numérico (quase sempre) estamos procurando boas soluções aproximadas utilizando as informações que colhemos na vida real, e muitas vezes essas informações vem em forma de pontos (x, y).

assim, podemos começar a estipular como imaginamos que a nossa derivada é. se olharmos para a definição original de derivada, podemos ver algo super interessante

$$f'(x) = \lim_{h\to 0} \frac{f(x+h) - f(x)}{h}$$

se nós imaginarmos que o nosso intervalo de derivação é discretizado, podemos estipular uma função g que é definida sobre o conjunto finito $X = [x_1, x_2, \cdots, x_n]$ tal que

$$\forall x_i \in X : \; g(x_i) = \frac{f(x_{i+1}) - f(x_i)}{x_{i+1} - x_i}$$

isto é, transformamos o intervalo h que teoricamente tende a zero na diferença entre dois elementos consecutivos, e assim eliminamos o problema do limite (já que não sabemos exatamente o que ele significa matematicamente *anyways*). de quebra, também eliminamos outro problema escondido: quando temos dados colhidos na vida real, nem sempre temos a certeza de que o intervalo foi uniformemente dividido (ou seja, o valor de h pode mudar para um mesmo conjunto).

utilizando essa definição, temos uma função que para cada ponto x_i, nos devolve o valor da derivada discreta de uma função f contínua, de modo que podemos aproximar (ou estimar) a derivada da função original sem ter que mexer com limites!

colocamos então como objetivo nosso implementar uma função $g(f, xs)$, que recebe uma função f e o intervalo de derivação xs (um conjunto de valor para os quais f está definida). entretanto, temos um pequeno empasse: como nós iremos iterar por xs? pois precisamos pegar todos os pares consecutivos dessa lista, e essas funções nem sempre são óbvias de se implementar.

se observarmos bem de perto, vemos que a estrutura dessa iteração consecutiva é bem parecida com algumas funções *higher order* comuns em linguagens funcionais, em especial com a família de funções `fold`. em haskell, definimos `foldl` (`fold` com associatividade à esquerda) como:

```
foldl :: (b -> a -> b) -> b -> [a] -> b
foldl f z []     = z
foldl f z (x:xs) = foldl f (f z x) xs
```

isto é, é uma função que recebe outra função f e consecutivamente aplica ela numa lista, carregando consigo o resultado da aplicação anterior de f (chamado de `z`). entretanto, não queremos guardar a última aplicação de função, mas sim o último elemento que fora iterado da lista. assim, o que queremos implementar de fato é a seguinte função:

```
consec :: (x -> x -> y) -> x -> [x] -> [y]
consec f ant []     = []
consec f ant (x:xs) = (f ant x) : (consec f x xs)

consecutivos f (x:xs) = consec f x xs
```

ok, talvez seja um pouco demais para absorver mas vamos com calma. implementamos o nosso iterador de elementos consecutivos `consec` de maneira bem parecida com o `fold`. a ideia é mantermos sempre o elemento anterior `ant` atualizado, de modo que quando calculamos o elemento atual, utilizamos `(f ant x)` e dizemos que `x` é o novo elemento anterior. por fim, concatenamos o resultado recursivamente, chamando `consec` na lista restante `xs`.

podemos ver isso em ação:

```
*Main>  consecutivos (+) [1, 2, 3, 4, 5]
[3,5,7,9]
```

o array `[3, 5, 7, 9]` é exatamente o resultado de aplicar a função soma `(+)` na lista de elementos consecutivos `[(1, 2), (2, 3), (3, 4), (4, 5)]`, que se torna `[(1 + 2), (2 + 3), (3 + 4), (4 + 5)]`, que é exatamente o que queríamos.

utilizando `consecutivos`, é muito fácil definir a derivada discreta em haskell:

```
derivada func xs =
  let g f = \a b -> ((f b) - (f a)) / (b - a) in
    consecutivos (g func) xs
```

entendeu?? essas três linhas acima são tudo que precisamos. a função `g` é a que emula nossa derivada (que fora citada anteriormente), recebendo uma função `f` e retornando a função que calcula aproximadamente o valor da derivada nesse ponto. depois, simplesmente retornamos `consecutivos`, aplicando função que fora fornecida para `derivada` em `g`.

```
import Graphics.Matplotlib

-- função que recebe um inicio, um fim e um tamanho `len`,
-- e retorna uma lista de [inicio, fim (aproximado)]
-- com exatamente `len` elementos
range :: Double -> Double -> Int -> [Double]
range start end len =
  let step = (end - start)/ (fromIntegral len) in
    take len [start, start+step..]

plot_1 = let
    start = -10
    end = 10
    n   = 200
    intervalo = range start end
    -- retorna uma lista com k intervalos
    f x = sin x
  in
    onscreen $
    plotMapLinear f start end (fromIntegral n) @@ [o2
 ↪  "color" "springgreen", o2 "label" "sin(x)"]
    % plotMapLinear cos start end (fromIntegral n) @@ [o2
 ↪  "color" "firebrick", o2 "label" "derivada real =
 ↪  cos(x)"]
    % scatter (intervalo (n-1)) (derivada f (intervalo n))
 ↪  @@ [o2 "label" "derivada aproximada", o2 "color"
 ↪  "tomato"]
    % legend @@ [o2 "loc" "upper right"]
    % tightLayout
plot_1
```

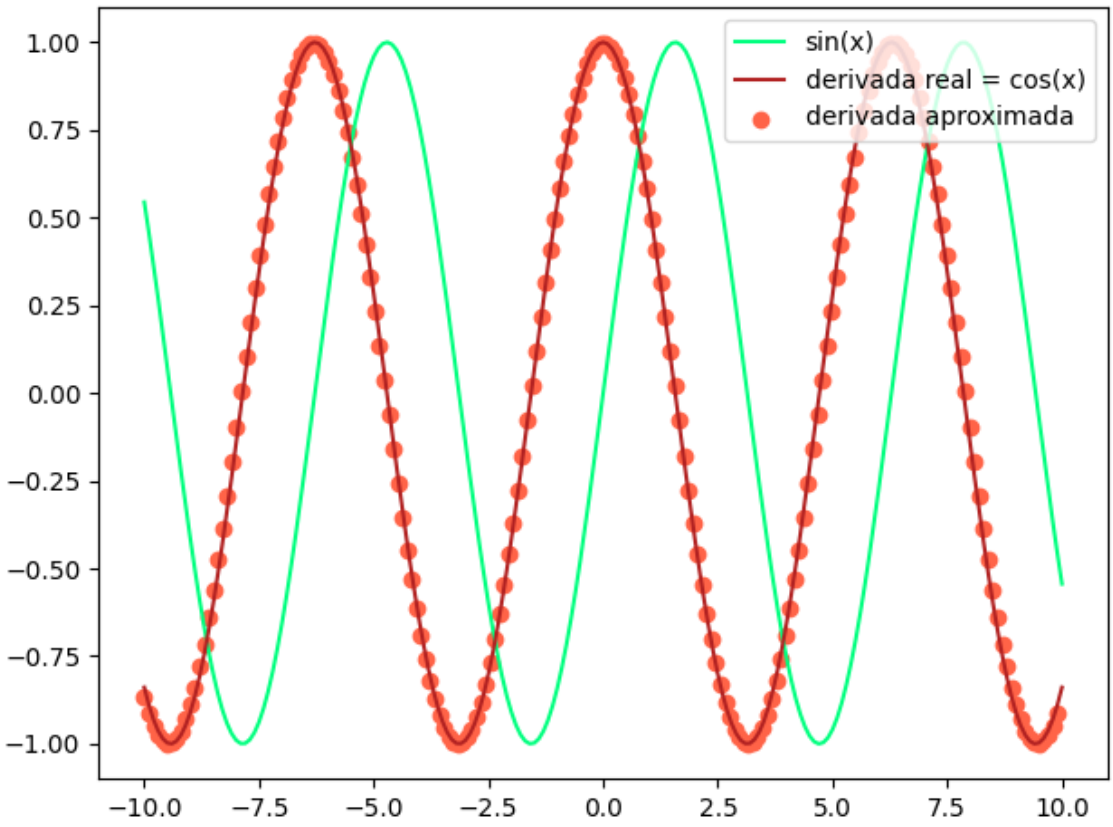

Figure 1: derivada discreta de $sin(x)$

sim, eu sou maluco de usar matplolib com haskell. mas é melhor do que ter que usar python ou julia pra definir funções recursivas sem sequer ter acesso à *tail call optimization* ou tipos algébricos. não tem problema se o código do *plot* parecer um pouco confuso, a única parte realmente importante é que estamos utilizando `scatter xs ys` para mostrar os pontos, onde `xs` é o nosso intervalo e `ys` é a lista de pontos retornada pela derivada.

o que é muito interessante é que a função `derivada` sequer sabe alguma coisa sobre a função seno. ela não sabe que $\sin^2 + \cos^2 = 1$, ou sobre definições de *power series* infinitas, ou qualquer outra relação que possa ser útil para derivada, e ainda sim ela é capaz de estimar (só utilizando o comportamento local da função) como a derivada se parece. de fato, podemos estimar a derivada de *qualquer* função `f`, contanto que ela esteja definida para todos os pontos de `xs`.

2 integral

enquanto definir a derivada foi uma caminhada no parque, a integral é um pouco mais *envolved*. contudo, já temos meio caminho andado: é comum tocar no tópico de somas de rieman quando estudamos cálculo, e é exatamente isso que queremos emular na integral discreta.

em especial, aproximaremos a área embaixo da curva através da regra trapezoidal (ao invés de retângulos, estimamos a área de trapézios embaixo da curva). isso significa que, se f é uma função definida no conjunto $X = [x_1, \cdots, x_n]$:

$$\int_{x_1}^{x_n} f(x)\,dx \approx \sum_{i=1}^{n} (f(x_i) + f(x_{i+1}))(x_{i+1} - x_i)\frac{1}{2}$$

ou seja, para calcular o valor da integral entre x_1 e x_n, somamos as áreas de todos os trapézios consecutivos. te lembra de algo? exatamente, `consecutivos` novamente.

```
integral_intervalo func xs =
  let trapezio f = \a b -> ((f b) + (f a)) * (b - a) * 1/2
  in
    sum $ consecutivos (trapezio func) xs
```

é uma implementação bem parecida com a derivada, só mudamos `g` para a função `trapezio` (que nos dá a área do trapézio entre `a` e `b`) e no final somamos todas essas áreas usando `sum`.

usando a integral podemos, por exemplo, calcular o valor de `e`:

$$\int_0^1 e^x\,dx = e^1 - e^0 = e - 1$$

```
*Main> e = (integral_intervalo exp (range 0 1 1000000)) + 1
*Main> e - (exp 1)
-2.7182803195024974e-6
*Main> e
2.7182791101787256
```

ignorando o fato de que estamos usando `exp` para calcular e depois comparando o resultado com `exp` (o que pode dar errado??), vemos que conseguimos aproximar nosso próprio valor do número de euler bem facilmente.

entretanto, o leitor ávido deve estar percebendo uma leve diferença agora. a função `derivada` recebe `f` e `xs` e retorna os pontos estimados na derivada, enquanto a função `integral_intervalo` recebe `f` e `xs` e retorna um único valor. como faríamos para estimar pontos que estariam na integral de f?

de fato, precisamos utilizar um pouco de engenhosidade. se olhar bem, verá que nomeei a função `integral_intervalo`, e não foi à toa: essa função calcula o valor específico da integral em um intervalo específico, e podemos usá-la para calcular os pontos que estariam na função que representa a integral.

brincando um pouco com a definição de integral, eventualmente você também chegará na seguinte relação:

$$\sigma(\alpha) = \int_c^\alpha f(x)\,dx = F(\alpha) - F(c)$$

onde F é a função que representa a integral de f, e c é uma constante. entendeu agora? a função que definimos σ depende de um parâmetro apenas e retorna o valor da integral nesse ponto menos uma constante (a integral de c). se escolhermos c tal que $F(c) = 0$, temos uma função que nos dá o valor exato da integral de f para cada ponto do domínio (claro, se estiver definido e tudo mais).

implementá-la não é difícil:

```
integral f xs c = map h xs
  where h x
          | c <= x    =   integral_intervalo f (range c x
↪  0.01)
          | otherwise = - integral_intervalo f (range x c
↪  0.01)
```

note que construimos uma função `h` da seguinte forma:

$$h(\alpha) = \begin{cases} \displaystyle\int_c^\alpha f(x)\,dx = F(\alpha) - F(c) \\ \text{se } \alpha \geq c \\ - \displaystyle\int_\alpha^c f(x)\,dx = -(F(c) - F(\alpha)) \\ \text{se } \alpha < c \end{cases} \tag{2.1}$$

ou seja, se o limite inferior de integração for maior que o superior, simplesmente invertemos os dois limites e negamos o resultado da derivada. fazemos isso pois como estaremos fixando c e estamos `map`'eando a função `h` num intervalo, é bem possíveli ocorrerem (e inclusive acontece na maior parte das vezes) casos onde $x < c$. para evitar problemas de sinal, essa é uma solução bem simples.

da mesma forma, podemos vizualizar a integral de $sin(x)$ usando `integral`:

```
plot_2 = let
    start = -10
    end = 10
    n   = 400
    intervalo = range start end -- point free notation
    -- retorna uma lista com k intervalos
    f x = sin x
  in
    file "img2.png"$
    plotMapLinear f start end (fromIntegral n) @@ [o2
↪   "color" "springgreen", o2 "label" "sin(x)"]
    % plotMapLinear (\x -> -cos x) start end (fromIntegral
↪   n) @@ [o2 "color" "firebrick", o2 "label" "integral
↪   real = -cos(x)"]
    % scatter (intervalo (n-1)) (integral f (intervalo n)
↪   (pi/2)) @@ [o2 "label" "integral aproximada de
↪   sin(x)", o2 "color" "tomato"]
    % legend @@ [o2 "loc" "upper right"]
    % tightLayout
plot_2
```

outro exemplo ainda mais divertido é:

```
plot_3 = let
    start = 0.1
    end = 10
    n   = 200
    intervalo k = range start end ((end - start)/ k)
    -- retorna uma lista com k intervalos
    f = log
  in
    onscreen $
    plotMapLinear f start end n @@ [o2 "color"
↪   "springgreen", o2 "label" "log(x)"]
```

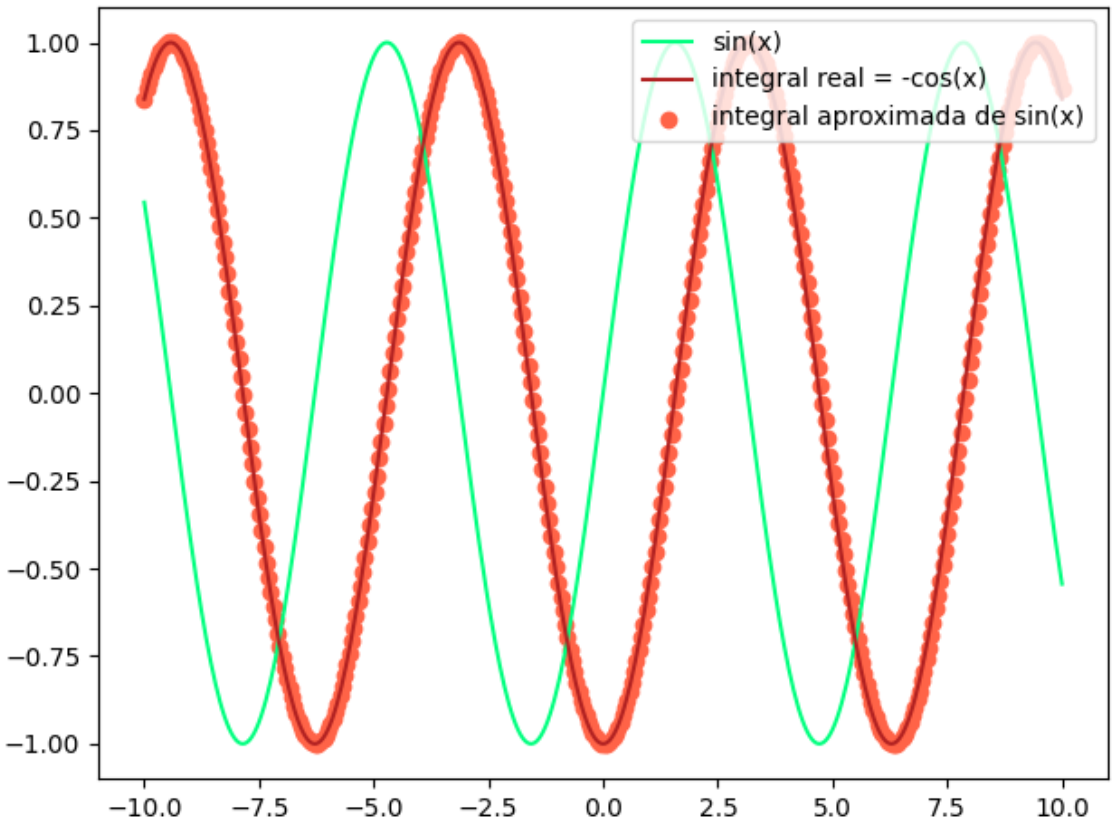

Figure 2: integral discreta de $sin(x)$

```
    % plotMapLinear (\x -> x * ((log x) -1)) start end n
↪   @@ [o2 "color" "firebrick", o2 "label" "integral real
↪   = x (log(x) -1)"]
    % scatter (intervalo (n-1)) (integral f (intervalo n)
↪   0.00000001) @@ [o2 "label" "integral aproximada de
↪   log(x)", o2 "color" "tomato"]
    % legend @@ [o2 "loc" "upper right"]
    % tightLayout
```

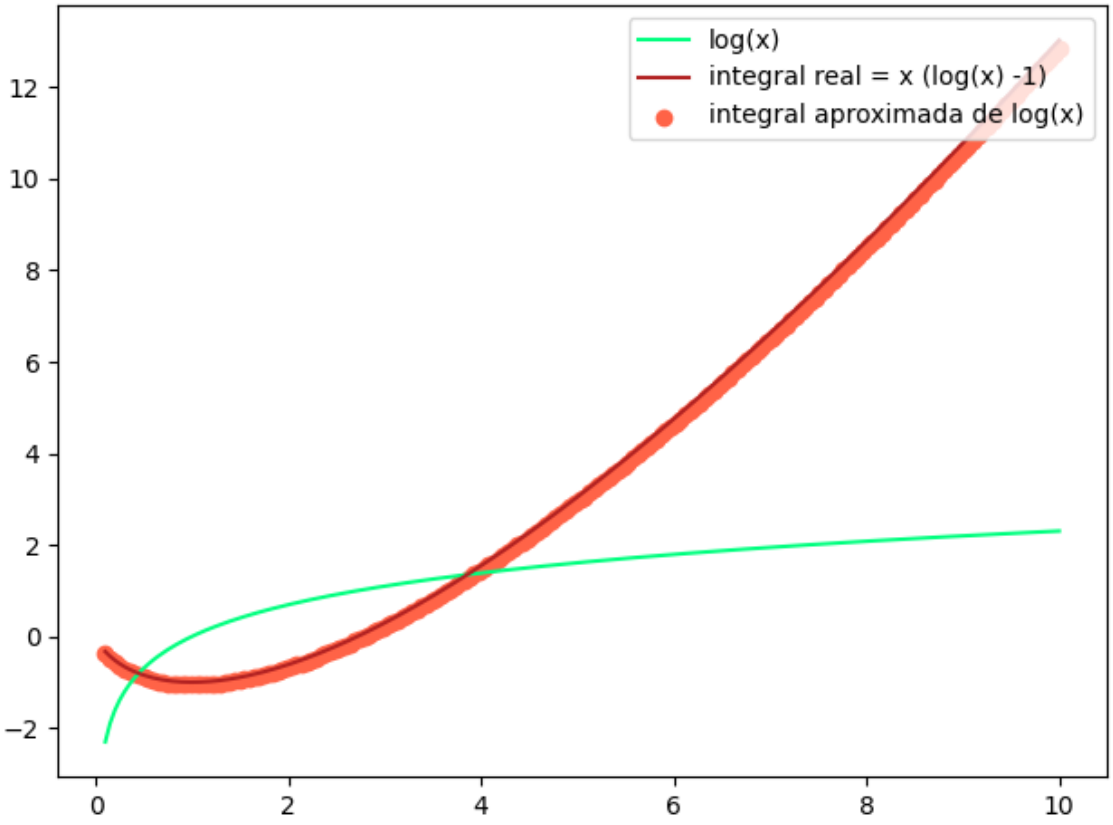

Figure 3: integral discreta de $log(x)$

esse exemplo é divertido porque se me pedirem para integrar $log(x)$ eu não saberei responder (aposto que utiliza alguma substituição trigonométrica), e ainda sim `integral` consegue achar facilmente a curva. note que tivemos que escolher c para um valor muito pequeno, mas não 0, já que a única (possível?) raiz dessa função é quando `x` é zero (claro, estou fugindo do problema que é definir o valor de $log(0)$).

outro exemplo extremamente bonito é o seguinte:

```
plot_4 = let
    start = -10
```

```
    end = 10
    n = 10000 :: Int
    intervalo = range start end -- point free notation
    -- retorna uma lista com k intervalos
    f x = sin(x^2)
  in
    onscreen $
    plotMapLinear f start end (fromIntegral n) @@ [o2
↪   "color" "springgreen", o2 "label" "$sin(x^2)$"]
    % plot (intervalo n) (integral f (intervalo n) 0) @@
↪   [o2 "label" "integral aproximada de $sin(x^2)$", o2
↪   "color" "tomato"]
    % legend @@ [o2 "loc" "upper right"]
    % tightLayout
plot_4
```

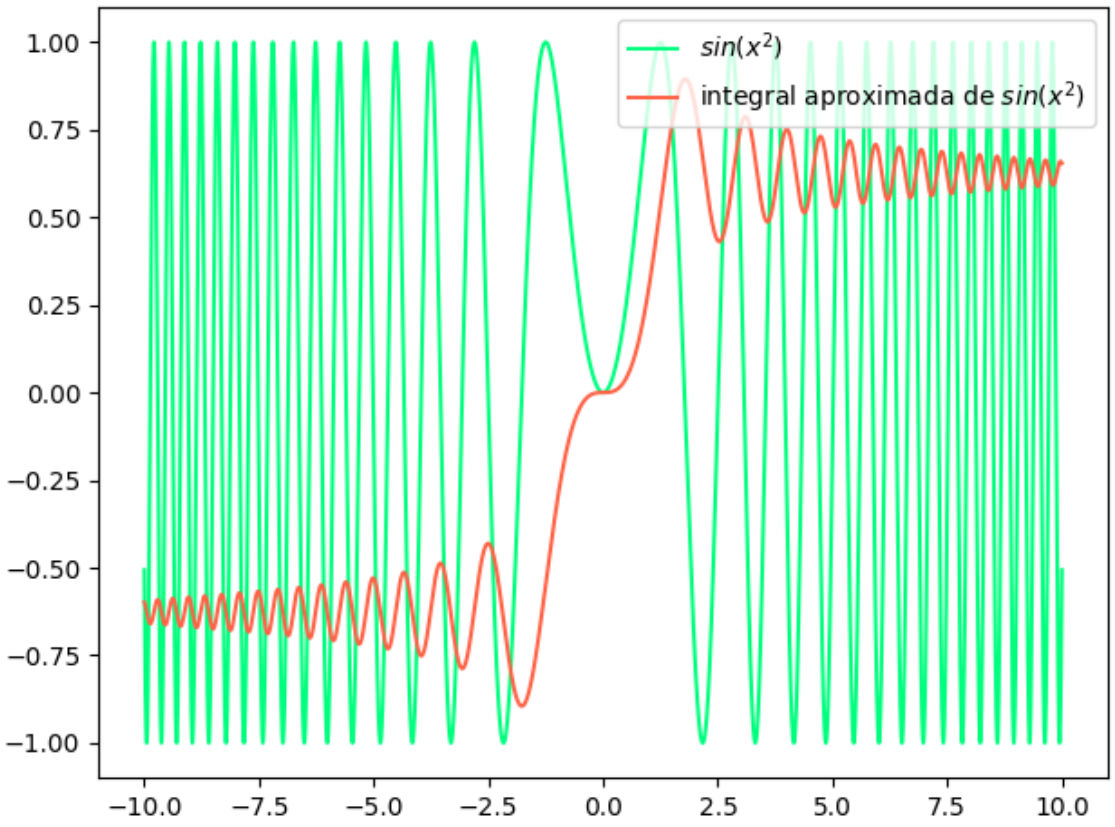

Figure 4: integral discreta de $\frac{\sin(x)}{x}$

a beleza deste exemplo advém do fato de que a integral de $sin(x^2)$ (chamada de *Fresnel Integral*) não possui uma forma elemental (isto é, não pode ser expressa como nenhuma sequência finita das operações comuns que conhecemos), e o máximo que conseguimos fazer é aproximar como uma série de taylor infinita. ainda sim, conseguimos claramente ver um esboço dessa integral.

enquanto parece ser uma definição contínua, o matplotlib está por trás dos panos fazendo uma interpolação de todos os pontos que fornecemos (quando utilizamos `plot`).

podemos usar essas duas integrais famosas para vizualizar um construto ainda mais bonito (e útil): a espiral de euler. essa espiral é utilizada em diversas áreas de cálculo numérico, e pode ser caracterizada pela seguinte equação paramétrica:

$$C = \forall t \in T : (x(t), y(t)) = (S(t), C(t)) = \left(\int_0^t sin(x^2)\,dx, \int_0^t cos(x^2)\,dx \right)$$

para um domínio T que definirmos. observando o lado direito da equação, vemos que as integrais calculadas são exatamente o que calculamos na função `integral`, portanto podemos calcular os pontos `xs` e `ys` usando a aproximação dessa integral e vizualizá-la:

```
euler = let
    start = -10
    end = 10
    n = 500 :: Int
    intervalo = range start end -- point free notation
    -- retorna uma lista com k intervalos

    z = intervalo n
    x = integral (\k -> sin(k**2)) z 0
    y = integral (\k -> cos(k**2)) z 0
  in
    onscreen $
    readData(x, y, z)
    % (mp # "fig = plot.figure()\n"
    # "ax = fig.add_subplot(projection='3d')\n")
    # "ax.plot(data[0], data[1], data[2])\n"
    # "fig.tight_layout()"
```

eu não sei vocês, mas eu acho isso muito foda. apesar de termos que utilizar algumas coisinhas em python diretamente (o que é um pouco feio honestamente), o cálculo principal é feito nas 3 linhas onde definimos `x`, `y` e `z`.

o que é ainda mais bonito é o fato de que as funções que usamos para calcular foram definidas de maneira simples, intuitiva e concisa (todas juntas não passam de 30 linhas de código), não possuem nada de arbitrário ou de incoerência e (portanto) são extremamente gerais no que fazem. enquanto é fato que haskell não é a linguagem mais rápida do mundo, eu encontrei muito mais facilidade de definir essas funções nela do que em julia, a linguagem utilizada na cadeira de cálculo numérico.

apesar disso, tenho que admitir que fazer plots em julia era muito, mas muito mais fácil do que fazer em haskell, e acho que se alguém algum dia sonha em fazer uma linguagem funcional mainstream, faz-se necessário (de caráter urgente) um bom suporte para gráficos; o melhor que eu consegui achar foi esse pequeno *hack* de invocar matplotlib de dentro de haskell.

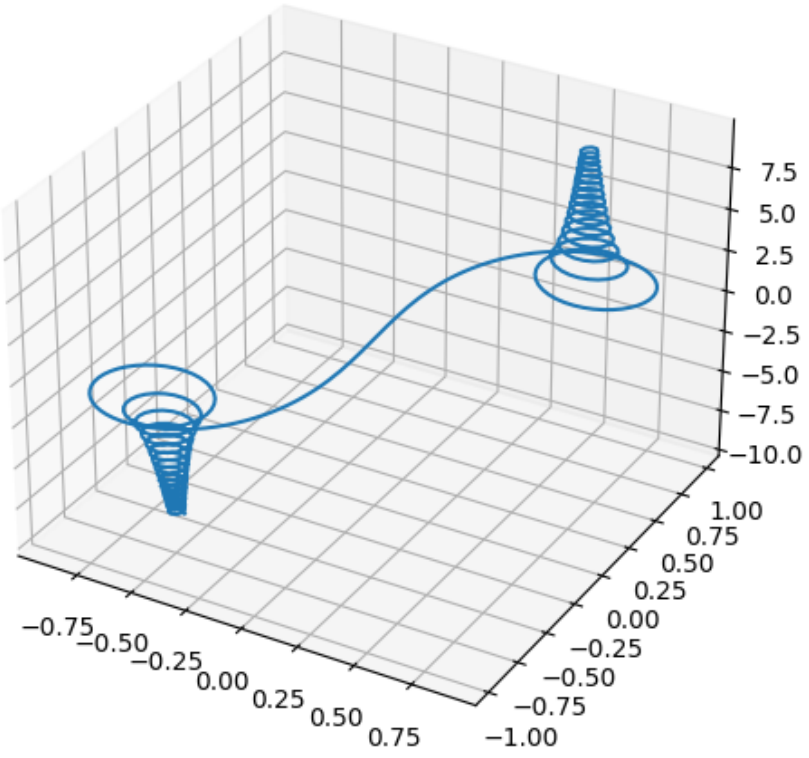

Figure 5: Espiral de Euler

dentro do tópico de cálculo numérico, acho que ainda restam alguns assuntos interessantes que merecem atenção, em especial integrais duplas e derivadas parciais. entretanto não pretendo me estender muito mais neste post; e quem sabe falarei novamente delas no futuro.