Teoria dos Números Básica com o *sagemath*

Vinicius Martins Teodosio Rocha

Sumário

Prefácio

O *sagemath* é um poderoso software para a manipulação de objetos matemáticos. É um software gratuito e de código aberto, sob a licensa GPL, construído sobre diversos outros pacotes, também de código aberto, como o NumPy, SciPy, matplotlib, Maxima, GAP, R, etc. Baseado na linguagem de programação Python é uma ótima ferramenta para explorar conceitos matemáticos.

Nesse texto usamos o *sagemath* para explorar a área da matemática conhecida como teoria dos números, ou aritmética. O *sagemath* possui uma enorme gama de recursos relacionados à teoria dos números básica, além de dispor de métodos que auxiliam a construção de ferramentas para nossas próprias investigações.

Sendo baseado em Python, o *sagemath* consegue executar qualquer comando ou script Python da forma esperada (a menos de um conjunto de medida nula). Com o risco de exagerarmos no tom informal, tentamos evitar ao máximo o uso conhecimentos prévios sobre a linguagem Python e sobre algoritmos de uma forma geral. Evidentemente, alguns conceitos e técnicas apresentados fazem uso de conhecimentos específicos de algoritmos, mas **não é necessário saber programar** para seguir essas notas, basta estar disposto a aprender.

Aproveitamos para destacar que existe uma enorme quantidade de recursos do *sagemath* que não abordaremos em detalhes, como o cálculo simbólico, a construção de gráficos integrado ao matplotlib, objetos da álgebra abstrata, como grupos e aneis, combinatória, computação numérica... Dessa forma, instigamos ao leitor que faça um passeio pela documentação do *sagemath* ([1]) e descubra outras ferramentas disponíveis que podem auxiliá-lo no seu dia a dia de estudante ou professor, além da documentação dos recursos, também há diversos tutoriais ou materiais sobre temas específicos e uma lista é mantida com os livros e publicações sobre o *sagemath*. Recomendamos também o ótimo livro [2], única referência em português sobre o *sagemath* de conhecimento do autor. Em inglês há mais recursos, destacamos [3] (Versão em inglês, original francês) e [4] (Versão em inglês, também disponível em espanhol), ambos com versões gratuitas disponíveis nos sites dos autores e contendo ao menos um capítulo sobre teoria dos números..

SEPARADOR

Ao escrever essas notas fizemos algumas escolhas sobre o estilo e o conteúdo que necessitam justificativa. Com o intuito de não tornar esse texto mais longo que deveria, optamos por omitir algumas demonstrações e discussões mais teóricas, indicando referências mais adequadas (e decerto mais bem escritas) para o que for omitido. Dessa forma, essas notas não tem como objetivo substituir um texto mais formal de teoria dos números, com todas as propriedades, teoremas e demonstrações. A exceção ocorre quando as demonstrações são, de alguma forma, construtivas. Nesse caso procuramos fornecer um esboço da demonstração que auxilie na construção de um código *sagemath*.

Apesar do nosso objetivo principal ser a apresentação do *sagemath* como uma ferramenta pronta para o uso, também gostaríamos que o leitor desenvolvesse um raciocínio lógico que o permita transpor uma lista simples de procedimentos em linguagem matemática para código *sagemath*. Para tal iremos apresentar a construção de algumas ferramentas básicas que já estão implementadas no sage, então, sim, em alguns momentos estaremos *reinventando a roda*, mas cremos que esse é um passo importante para o entendimento dos algoritmos mais avançados que apresentaremos ao final do texto.

Para os que já são familiarizados com a linguagem Python, inserimos algumas observações onde o *sagemath* supostamente funciona de forma diferente do Python, ou onde objetos ligeiramente diferentes são usados. Para os que não são familiarizados com o *sagemath* ou Python, optamos por não inserir uma longa seção inicial tratando de sintaxe, lógica de programação, etc. Vamos direto ao ponto e espalhamos essas explicações aqui e ali, na medida que se fazem necessárias. Para os leitores mais sistemáticos, os três primeiros capítulos do já citado [2] fornecem uma introdução completa ao *sagemath*, sugerimos sua leitura antes de explorar esse texto.

Descrição capítulo a capítulo

Inserimos em cada capítulo uma seção chamada **Explore!**, onde descrevemos alguns tópicos adicionais que acreditamos serem aplicações interessantes. Dentro de cada tópico há alguns exercícios mas, para alguns deles, o instrutor pode desenvolver miniprojetos estendendo o escopo abordado. No final de cada capítulo há uma seção **Exercícios**, com problemas adicionais, generalizações de resultados ou implementações alternativas às mostradas no corpo do texto. Tais exercícios não são, em geral, fundamentais para a compreensão do assunto. Por outro lado, no corpo do texto há determinados exercícios que julgamos essenciais para o desenvolvimento da aprendizagem, alguns, inclusive, podem não fazer muito sentido se ignorados e deixados para depois. Tentamos indicar os exercícios mais trabalhosos ou que exigem um aprofundamento das técnicas com um sinal X, no entanto nenhum requer assuntos não abordados, exceto menção explícita a alguma referência.

Introdução

O *sagemath*

Apresentamos brevemente como utilizar o *sagemath* e alguns conceitos básicos.

Há diversas formas de se trabalhar com o *sagemath*, e uma de suas vantagens é que você nem precisa ter o *sagemath* instalado em seu computador para usá-lo. Na prática, não há muitas razões para a maior parte dos usuários instalar o *sagemath*.

De toda forma, se optar pela instalação, siga as instruções em `https://www.sagemath.org/download.html`. Após a instalação a forma mais direta de utilização é através da linha de comando: abra uma linha de comando, digite `sage` e, voilá, você já está dentro do interpretador interativo *sagemath*. Tente digitar algum código

```
1 2+2
```

```
4
```

Usaremos o padrão acima nessas notas, o texto no quadro azul é o código *sagemath* e o texto no quadro verde é o resultado do código acima. No interpretador *sagemath*, qualquer código sage válido inserido será executado e você verá o resultado na hora. Para códigos mais elaborados o mais usual é salvá-lo em um arquivo com a extensão `.sage` e executar o comando `sage` na linha de comando com o arquivo como argumento. No entanto, a interface mais amigável para o *sagemath* é fornecida pelo Jupyter, um projeto para o desenvolvimento de padrões para computação interativa. Ao utilizar o *sagemath* em um notebook Jupyter você vai trabalhar em um página web local, através do seu navegador, como na Figura 1.

Par a maior parte das pessoas, no entanto, recomendamos a utilização de alguma opção *online* do *sagemath*. Há duas opções principais. A mais simples e direta de todas é utilizar a célula de cálculo no site *SageMathCell*, acessado em `https://sagecell.sagemath.org`. Nesse site você verá um campo de texto onde pode inserir código *sagemath*. Em seguida basta clicar em *Evaluate* para ver o resultado, como na Figura 2. Todos os códigos utilizados nesse texto podem ser executados no *SageMathCell*. Vale destacar, no entanto, que ao fechar o site todo o seu código será perdido, bem como o resultado. Dessa forma se quiser salvar o seu trabalho, você pode copiar o código que digitou no *SageMathCell* e salvar em um arquivo de texto, podendo voltar a trabalhar nele em outro momento.

Outra opção, que evita o problema de perder os códigos, é usar o *CoCalc*, uma plataforma online permitindo o uso de diversas ferramentas, incluindo o *sagemath*. Apesar de existirem diversos planos pagos, fornecendo mais capacidade de memória e processamento, há uma versão gratuita que é o suficiente para grande parte das aplicações. Ao criar uma conta no site do *CoCalc*, você poderá criar arquivos e pro-

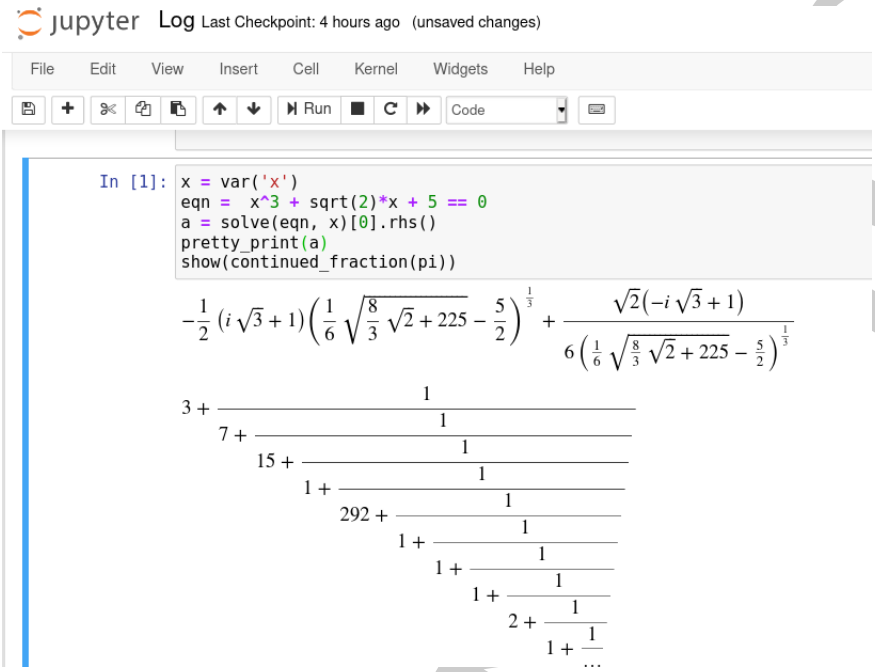


Figura 1: Tela do *Jupyter* {fig:jupyte

jetos de diversos tipos. Estaremos interessado em particular em *worksheets* sagemath, arquivos no formato `.sagews`, que funcionam de forma parecida com notebooks do Jupyter[1]. A grande vantagem é que os seus arquivos ficarão salvos na sua conta, mesmo após o fechamento do navegador, além disso você pode trabalhar colaborativamente com outros colegas no mesmo arquivo.

O *CoCalc* e o *Jupyter* fornecem diversas outras ferramentas, recomendamos que o leitor acesse os sites dessas plataformas e descubra quais dessas ferramentas pode achar útil. Como comentamos, para essas notas, o uso do *SageMathCell* será suficiente.

Vemos a seguir alguns conceitos básicos sobre o *sagemath*.

Conjuntos Numéricos Os objetos mais básicos que utilizaremos serão os números. Sabemos que o número 12, por exemplo, é um número inteiro, racional, real e complexo. No *sagemath*, no entanto, faz certa diferença onde os números estão sendo considerados. Não fará sentido, por exemplo, perguntar ao *sagemath* a fatoração em primos de `12.0`, pois o número 12, inserido dessa forma, com o `.0`, será considerado como um número real. No entanto, pedir a fatoração em primos de `12` retorna o resultado esperado. Os recursos disponíveis e as operações irão portanto depender do *tipo* do objeto considerado. Vejamos como definir alguns tipos números e verificamos o tipo dos objetos gerados usando a função `parent`. No *sagemath*, a unidade imaginária $i = \sqrt{-1}$ pode ser inserida usando a letra `I`.

Código	Resultado
`parent(5)`	Integer Ring
`parent(5/3)`	Rational Field
`parent(5.12)`	Real Field with 53 bits of precision
`parent(5+3*I)`	Symbolic Ring

[1] também é possível usar o Jupyter com o *sagemath* dentro do Cocalc

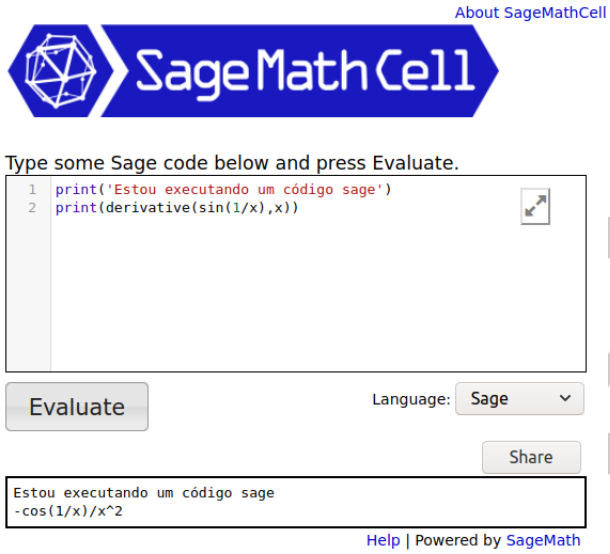


Figura 2: Tela do SageMathCell {fig:sagema

O último resultado não foi bem o esperado, ele ficará mais claro no Parágrafo **Sequências Recorrentes e Expressões Simbólicas** na Seção 1.7.5 (mas, se ainda estiver desconfiado, calcule `I^2`). Podemos tentar converter um número de um tipo para outro usando a *coerção*. Para isso usaremos `ZZ` para os inteiros, `QQ` para os racionais, `RR` para os reais e `CC` para os complexos. Vamos tentar converter alguns números de um conjunto para o outro . Assumiremos que você executará o código abaixo, e todos os demais nessas notas, em uma célula como no *SageMathCell*. Nesse tipo de célula você pode escrever várias linhas de código ao mesmo tempo. O *sagemath* irá executar todas, no entanto só exibirá o resultado da última linha. Para forçar a exibição usamos a função `print`.

```
print(QQ(5.12))
print(parent(QQ(5.12)))
print(RR(1/7))
print(QQ(pi))
```

```
128/25
Rational Field
0.142857142857143
---------------------------------------------------------------
TypeError                 Traceback (most recent call last)
# ...  GRANDE MENSAGEM DE ERRO... #
TypeError: unable to convert pi to a rational
```

Destacamos que a aritmética nos inteiros e nos racionais acontece de forma exata[2]. Por outro lado, como números reais, e complexos, em geral tem representação decimal infinita, não é possível armazená-los completamente na memória do seu computador. Dessa forma, o *sagemath* (e qualquer outro sistema) usa uma aproximação

[2]Para os conhecedores de Python: no *sagemath*, o operador `/` entre inteiros retorna um objeto do tipo `Rational`, um número racional. Para obter o resultado como ponto flutuante ou número real do *sagemath* use `float(a/b)` ou `RR(a/b)`, respectivamente.

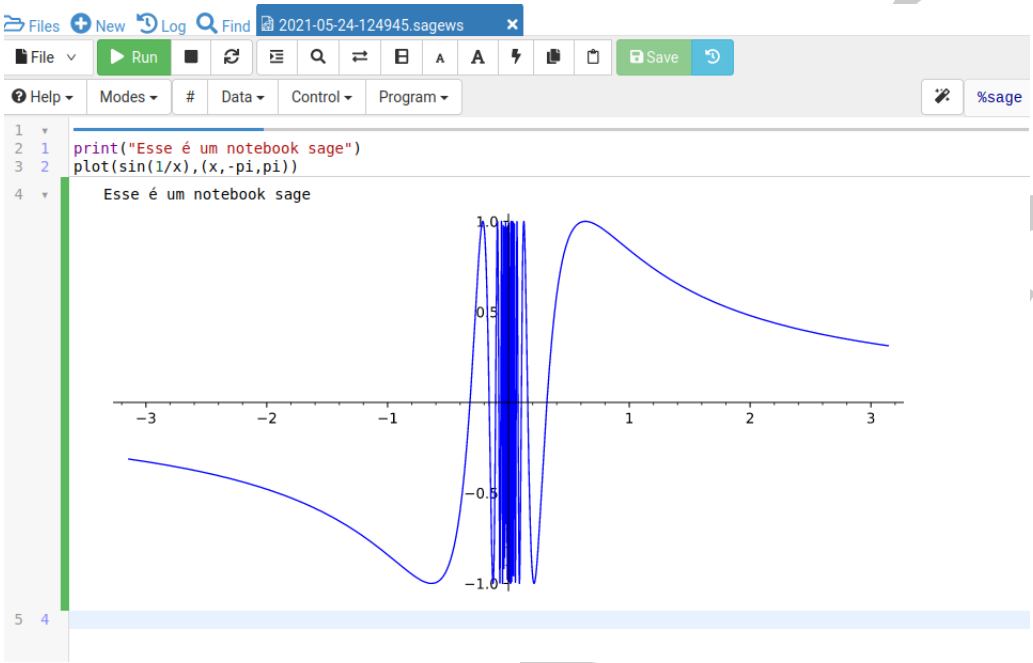


Figura 3: Tela do *CoCalc*

{img:cocalc}

do número real com uma precisão pré-determinada, chamada de *ponto flutuante*. É possível aumentar arbitrariamente a precisão considerada.

```
print(RR(e))
RR100 = RealField(prec=100)
print(RR100)
print(RR100(e))
```

```
2.71828182845905
Real Field with 100 bits of precision
2.7182818284590452353602874714
```

É preciso um pouco de cautela ao trabalhar com números reais. Considere os números $1 + 10^{-20} \neq 1 + 2 \times 10^{-20}$. Veja o que ocorre quando o comparamos como elementos do `RR` com a precisão padrão do *sagemath* e com o `RR100` que definimos acima[3].

```
RR(1+10^(-20)) == RR(1+2*10^(-20))
RR100(1+10^(-20)) == RR100(1+2*10^(-20))
```

```
True
False
```

Operações básicas Como é de se esperar, podemos fazer as operações básicas com os números, e, em alguns casos, até com outros objetos. Usamos o `*` para o produto, o `/` para divisão — embora isso possa não retornar o resultado esperado

[3] O `prec=100` não significa que tais números reais terão precisão de 100 dígitos decimais, e sim 100 bits.

— e, ao contrário do Python, usamos `^` para exponenciação. A ordem das operações segue a convenção usual, no entanto recomendamos o uso de parenteses mesmo quando não for necessário. Assim $1 + \frac{2\times3+1}{1+\frac{1}{2^3}}$, por exemplo, se escreveria como `1+(2*3+1)/(1+(1/(2^3)))`

Listas Um dos tipos de objetos mais importantes que usaremos no *sagemath* é a lista. Uma lista pode ser pensada como uma sequência finita de elementos que podem ser acessados pela posição, chamada de *índice*, sendo o primeiro elemento o elemento de índice 0, o segundo de índice 1, etc. No *sagemath* usamos colchetes para definir listas e para recuperar seus elementos. Os comandos a seguir mostram algumas propriedades da lista.

```
L = ["Euler", "Gauss", "Germain"]
"Gauss" in L
```

```
True
```

```
L[0]; L[2]
```

```
'Euler'
'Germain'
```

É importante destacar que listas no *sagemath* **não são conjuntos**, já que eles podem ter elementos repetidos e a ordem dos elementos importa. No entanto, na prática usaremos listas como conjuntos nas nossas aplicações, principalmente pela facilidade de se recuperar elementos [4].

Uma das técnicas mais úteis para se definir uma lista em *sagemath* é através da compreensão de lista. A compreensão de listas é muito natural para nós matemáticos pois tem a estrutura parecida com a definição de um conjunto a partir de outro. Por exemplo, poderíamos definir o conjunto dos 5 primeiros quadrados perfeitos como $\{n^2 \mid n \in \{1,2,3,4,5\}\}$. No *sagemath* isso pode ser feito assim:

```
[n^2 for n in [1..5]]
```

```
[1, 4, 9, 16, 25]
```

Se quiséssemos agora calcular o fatorial dos 5 primeiros quadrados perfeitos, bastaria usar a lista já criada:

```
Q = [n^2 for n in [1..5]]
[factorial(x) for x in Q]
```

```
[1, 24, 362880, 20922789888000, 155112100433309859840000000]
```

A notação `[a..b]` cria uma lista que funciona como o conjunto $\{a, a+1, \ldots, b\}$. Uma outra forma de criar listas de números é utilizar a função `srange(x)`, que

[4] Existe uma forma de se trabalhar com conjuntos no *sagemath* mas, por enquanto, isso não nos será muito útil

cria uma lista que funciona como o conjunto $\{0, \ldots, x-1\}$ quando $x \geq 0$ é inteiro. Podemos também adicionar um parâmetro a mais para começar de outro inteiro diferente do zero. Veja:

```
srange(-5,5)
```

```
[-5, -4, -3, -2, -1, 0, 1, 2, 3, 4]
```

Note que, em ambos os casos, a lista gerada pelo `srange` não inclui o último elemento, de forma que `[a..b]` e `srange(a,b+1)` geram a mesma lista, assim como `[0..x]` e `srange(x+1)`. A função `srange` também permite alguns parâmetros adicionais que nos serão úteis mais adiante [5].

Listamos alguns métodos e funções de listas que usaremos com frequência. Usamos a lista `L = [1,2,3]` nos exemplos, os métodos não retornam uma lista nova, e sim alteram a lista `L` (efeitos não acumulativos, i.e. redefinimos a lista `L` em cada linha)

	Definição	Exemplo	Resultado
`min`	Menor elemento	`min(L)`	`1`
`max`	Maior elemento	`max(L)`	`3`
`sum`	Soma	`sum([1..6])`	`21`
`prod`	Produto	`prod([1..5])`	`120`
`sorted`	Ordena	`sorted([1,-1,2,-12])`	`[-12, -1, 1, 2]`
`.append`	Insere no final	`L.append(5)`	`L` → `[1,2,3,5]`
`.remove`	Remove 1ª ocorrência	`L.remove(2)`	`L` → `[1,3]`
	`Lr = [1,2,3,2,4]`	`Lr.remove(2)`	`Lr` → `[1,3,2,4]`
`.pop`	Remove por índice	`L.pop(0)`	`L` → `[2,3]`
`.reverse`	Inverte ordem	`L.reverse()`	`L` → `[3,2,1]`

If (else, elif), for, while Apresentamos, brevemente, alguns conceitos que utilizaremos sobre a estrutura dos nossos códigos. Ao se deparar com um código, o *sagemath* irá executar os comandos sequencialmente, da primeira até a última linha. Há duas formas principais de alterarmos essa ordem, utilizando condicionais ou laços de repetição. Um *condicional* permitirá que você possa determinar comportamentos diferentes dependendo se uma condição for válida ou não. Em (quase?) toda linguagem de programação o condicional usa a palavra em inglês `if`, que significa *se*. A sintaxe é a seguinte:

[5] Para os conhecedores de Python: existem algumas diferenças entre o `srange` e o `range` usual. A mais importante é que os números de uma lista gerada com `srange` são inteiros da classe `Integer`, e não da classe `int` do Python. Isso significa que os métodos específicos da classe `Integer` não irão funcionar em inteiros gerados usando o `range` usual.

```
1 if (condicao):
2   Comando #1
3   Comando #2
4   Comando #3
5 Comando #4
```

Note que alguns comandos estão com um espaçamento diferente. O nome disso é *indentação*, esses comandos estão mais afastados do início da linha, e no mesmo nível, justamente para indicar ao *sagemath* o que deve ser executado caso a condição seja verdadeira. No exemplo acima, seriam executados os comandos #1, #2 e #3 caso a condição seja verdadeira. O comando #4 não está com a mesma indentação dos anteriores, dizemos que ele está *fora do* `if`, portanto ele será executado sempre, mesmo que a condição seja falsa. A `condicao` é um pedaço de código que o *sagemath* consiga testar se é verdadeiro ou falso. Formalmente, existe um tipo de objeto, chamado *booleano*, que assume dois valores diferentes, `True` para verdadeiro e `False` para falso. Ao se deparar com um comando `if` como acima, o *sagemath* tenta transformar a condição em um valor booleano. Na tabela abaixo mostramos os testes mais comuns usados em condições.

sagemath	Condição	Exemplo	Resultado
`a == b`	a igual à b	`1 == 3`	`False`
`a > b`	a maior que b	`2 > pi`	`False`
`a >= b`	a maior ou igual à b	`2 >= 2`	`True`
`a < b`	a menor que b	`2 < 2`	`False`
`a <= b`	a menor ou igual à b	`2 <= 7`	`True`
`a=!b`	a diferente de b	`1!=2`	`True`
`a in L`	a está na lista L	`3 in [2,4,6]`	`False`
`not P`	negação de P, não P	`not (1 == 2)`	`True`

Você pode testar alguns desses valores substituindo o valor de `a` e `b` ou de algumas constantes. Mas antes leia as seguintes observações:

- Note que utilizamos `==` para comparar dois objetos. De fato, o `==` tem um significado diferente do sinal de igual simples `=` que utilizamos anteriormente. O sinal de igual funciona como atribuição: Ao escrever `a = 2` estamos atribuindo ao `a` o valor 2 (matematicamente, seria como escrever '*Seja* $a = 2$' ou '*Tome* $a = 2$') . Enquanto que o `==` serve apenas para testar se dois objetos são iguais, de forma que `a == 2` é um objeto do tipo `bool` que tem apenas dois valores: `True` se `a` for 2 ou `False` caso contrário [6].

- O comportamento de algum desses operadores pode parecer estranho quando envolvem expressões simbólicas (descobriremos mais adiante o que isso significa) ou outros objetos. No entanto, com números você não deve ter problemas e obter o resultado esperado.

[6] Se `x` já existe, `x=x+1` faz sentido pois estamos atribuindo à `x` o valor do seu sucessor (teste!), enquanto que `x == x+1` é um teste que deveria retornar sempre `Falso` já que nenhum *número* é igual ao seu sucessor... ou é? (Dica: no *sagemath* existe o `Infinity`. Quanto é $\infty + 1$?)

- Na última linha da tabela utilizamos o `not` para negar logicamente o valor de uma outra condição `P`. Também é possível fazer outras operações entre condições usando o `and` (e) e `or` (ou).

Também é possível passar ao *sagemath* um outro bloco de comandos a ser executado caso a condição seja falsa, com o `else`, ou testar várias condições simultaneamente com o `elif`. Nos reservamos a comentar esses comandos na medida que forem necessários.

Outra forma de mudar a ordem usual de uma lista de comandos é utilizar um laço de repetição. O primeiro laço de repetição que apresentaremos é o `for`, usado quando gostaríamos de repetir um bloco de comandos enquanto uma variável varia em uma lista de elementos. A sintaxe é dada por

```
for obj in L:
  Comando #1
  Comando #2
  ...
```

Onde `L` é uma lista (ou outro objeto iterável). Nesse exemplo, inicialmente `obj` assume o valor do primeiro elemento da lista, em seguida os comandos do bloco são executados, ao chegar no final do nível de indentação, atribui-se ao `obj` o segundo elemento da lista e voltamos ao Comando #1, isso ocorre tantas vezes quanto há elementos na lista `L`, em cada repetição, chamada de *iteração*, o valor de `obj` é um valor diferente da lista. Quando a lista acaba o programa *sai do* `for` e os comandos da indentação anterior continuam a ser executados.

```
matematicos = ["Euler","Gauss","Dirichlet"]
for m in matematicos:
  print("{} foi um matematico".format(m))
```

```
Euler foi um matematico
Gauss foi um matematico
Dirichlet foi um matematico
```

O `for` é particularmente útil quando queremos que um bloco de comandos se repita uma quantidade predeterminada de vezes. Por exemplo, se quisermos exibir uma mesma mensagem 3 vezes, usamos

```
for i in [1..3]:
  print("OI")
```

```
OI
OI
OI
```

O outro laço de repetição que nos será particularmente útil é chamado de `while`. O *while*, enquanto em português, funciona como o `if`: fornecemos uma condição a ser testada e, caso a condição seja verdadeira, um bloco de comandos será executado. No entanto, ao final do bloco de comandos, a condição será testada novamente, em seguida, se ainda for verdadeira, o bloco será novamente executado. Esse processo se repetirá enquanto a condição for verdadeira. Quando a condição for falsa, o bloco não

é executado e programa segue a execução normal das linhas posteriores. A sintaxe é a seguinte:

```
while (condicao):
  Comand #1
  Comand #2
  ...
```

Funções e métodos Funções no *sagemath* funcionam de uma forma muito parecida com o que estamos habituados na matemática, se f é uma função e n é um elemento no domínio dessa função $f(n)$ será o valor de f calculada ou computada em n. Diferente das funções matemáticas, não existe uma rigidez muito grande com relação ao domínio e contradomínio nas funções do *sagemath*— é possível, inclusive, que uma função não tenha nenhum argumento ou não retorne nada. Já vimos algumas funções como `print` e o `parent`, mas vejamos algumas funções matemáticas que o leitor deve conhecer

Função	Exemplo	Resultado
$\sin x$	`sin(pi/3)`	`1/2*sqrt(3)`
e^x	`e^(3.0)`	`20.0855369231877`
$\sqrt{x}$	`sqrt(9)`	`3`
$\sqrt{x}$	`sqrt(3)`	`sqrt(3)`
$\log_e x = \ln x$	`log(2.0)`	`0.693147180559945`
$\log_b x$	`log(100,10)`	`2`
$n!$	`factorial(6)`	`720`

Note que algumas funções retornam resultados exatos em certos momentos e aproximados em outros. Para forçar um resultado aproximado use a função `N`, por exemplo, `N(sqrt(3))` retorna `1.73205...`[7]

As funções no *sagemath* podem receber mais de um valor de entrada, por exemplo número binomial $\binom{n}{m}$ está implementado como a função `binomial`: `binomial(5,3)` retorna `10`, como o esperado. O que chamamos matemática de *valor de entrada* ou *argumento* de uma função é chamado computacionalmente de *parâmetro*. Algumas funções tem parâmetros opcionais, veja na tabela acima como ln e $\log_b$ usam a mesma função `log`. Ao calcular a função `log` com apenas um parâmetro é calculado o logaritmo neperiano, na base e, como aconteceu com o `log(2.0)`, ao passar um argumento extra, como fizemos com o `log(100,10)`, estamos avisando ao *sagemath* que a base do logaritmo agora é 10. Para alguns parâmetros opcionais é necessário

[7] A função `N` ou `n` retorna a aproximação **n**umérica do argumento e é equivalente a transformar o resultado em número real usando `RR`, no entanto você pode especificar a quantidade de dígitos (total) usando o parâmetro extra `digits`, por exemplo `N(pi,digits=4)` retorna `3.142`. Outra possibilidade é colocar o elemento do domínio já como um número real, como fizemos com o `log(2.0)`. Como é comum usarmos n ou N como nome de outras variáveis, essa função também é chamada de `numerical_approx`.

mencionar explicitamente o nome do parâmetro, como o parâmetro `prec` para a coerção com o `RR`.

Existe um outro tipo objeto, de certa forma parecido com uma função, que também pode nos ser útil, o chamamos de *método*. Já nos deparamos com eles ao falar, por exemplo, do `.append` para listas. A distinção teórica entre métodos e funções não é importante para nossas aplicações, mas devemos distinguir a sintaxe. Uma função existe por si só, usamos a função inserindo o nome dela e, em seguida, entre parentes, seus argumentos ou parâmetros, como vimos nos exemplos acima. Um método, no entanto, está relacionado a um objeto e é usado da seguinte forma: `obj.metodo(parametros)`. veja o código a seguir, onde usamos o método `.nth_root` para calcular a raiz quinta de 2.

```
a = 2.0
a.nth_root(5)
```

```
1.14869835499704
```

Se você tentar usar o `.nth_root` como uma função deve obter um erro. Usaremos o ponto antes do nome de um método para indicar que ele é um método, e não uma função. Muitas das funções no *sagemath* também existem como métodos (pode parecer estranho mas calcule, por exemplo, `pi.sin()`). Assim como funções, métodos também podem receber argumento, inclusive opcionais.

Uma parte fundamental do *sagemath*, e qualquer outra linguagem de programação, é que podemos criar a nossas próprias funções. Para isso devemos informar ao *sagemath* o nome da função e os parâmetros que ela recebe através da linha `def nome(parametros):`. Para avisar ao sage o resultado da função, usamos a palavra chave `return`. Vejamos um exemplo simples, vamos definir a função $f(x) = x^2 - x + 41$

```
def f(x):
  return x^2 - x + 41
```

Se você executar o código acima o *sagemath* não exibirá resultado nenhum, afinal, você apenas definiu a função. Para utilizar, basta calculá-la para algum argumento

```
f(12)
```

```
173
```

Note que, como aconteceu com o condicional e os laços de repetição, o código após o `:` está em um nível maior de indentação. É possível colocar linhas de código *dentro* de uma função, usando a indentação, para indicar que os códigos envolvidos devem ser utilizados. Na função a seguir passamos os parâmetros `a,b,c` que são coeficientes de um polinômio quadrático $ax^2 + bx + c$ e retornamos as raízes.

```
def raizes(a,b,c):
  delta = b^2 - 4*a*c
  r1 = (-b + sqrt(delta))/(2*a)
  r2 = (-b - sqrt(delta))/(2*a)
```

```
5     return (r1,r2)
6
7   # Calculamos as raizes de x^2 - 7x + 10
8   raizes(1,-7,10)
```

```
(5,2)
```

Note que não estamos nos preocupando com a validade dos parâmetros passados. Podemos melhorar o código acima da seguinte forma:

```
1   def raizes(a,b,c):
2     if (a == 0):
3       return "Erro: polinomio nao quadratico."
4     delta = b^2 - 4*a*c
5     r1 = (-b + sqrt(delta))/(2*a)
6     r2 = (-b - sqrt(delta))/(2*a)
7     return (r1,r2)
8   print(raizes(0,1,1))
9   print(raizes(1,0,2))
```

```
Erro: polinomio nao quadratico.
(sqrt(-2), -sqrt(-2))
```

Duas coisas a se observar aqui, usamos o `if` dentro de um bloco que já tem um nível de indentação, portanto o código dentro do `if` está com dois níveis de indentação. Outro ponto importante é que, num código normal, ao sair do `if`, as demais linhas seriam executadas. No entanto, dentro de uma função o *sagemath* interrompe a execução ao encontrar o `return`. Isso significa que, se $a = 0$, a condição `a==0` é verdadeira, portanto a linha 3 é executada e, como é um `return`, a função acaba nesse ponto. Caso $a \neq 0$, o *sagemath* irá ignorar o `return` dentro do `if` e seguirá com as linhas posteriores [8].

Finalizamos listando mais algumas funções que usaremos frequentemente, exploraremos mais a fundo algumas dessas funções no texto.

Erros Em um dos parágrafos anteriores, ao tentarmos considerar o número π como um elemento do conjunto dos racionais Q, nos deparamos com um erro. Na verdade, para economizar espaço não exibimos o erro completo, apenas o início e a última linha — ao total são exibidas mais de 60 linhas de códigos aparentemente indecifráveis. Alguns erros são um pouco mais concisos.

```
1   1/0
```

```
-------------------------------------------------------
```

[8] Idealmente, o teste que fizemos para verificar se $a = 0$ e como procedemos em caso afirmativo deveria ser feito usando um processo chamado de tratamento de exceções, que formaliza a resposta à ocorrência de erros. Não é recomendado que uma função retorne o resultado esperado, ou uma mensagem de erro caso exista algum problema, como fizemos acima. No entanto essas técnicas envolvem conhecimentos mais específicos de programação e não serão abordadas nesse texto — me perdoem, programadores.

Função	Definição	Exemplo	Resultado
`min`	Menor elemento	`min(e^pi,pi^e)`	`pi^e`
`max`	Maior elemento	`max(1,2)`	`2`
`floor`	Função piso, maior inteiro	`floor(pi)`	`3`
		`floor(-sqrt(2))`	`-2`
`ceil`	Função teto, menor inteiro	`ceil(e)`	`3`
`round`	Arredondamento	`round(pi)`	`3`
		`round(e)`	`3`
		`round(2.5)`	`3`
		`round(1.23456,4)`	`1.2346`
`frac`	Parte fracionária	`frac(pi)`	`pi-3`
		`N(frac(pi), digits=4)`	`0.1416`
`sgn`	Sinal	`sgn((-1)^13)`	`-1`
		`sgn(0)`	`0`
`abs`	Valor absoluto	`abs((-2)^2)`	`4`

```
ZeroDivisionError      Traceback (most recent call last)
<ipython-input-92-72ac74c5f414> in <module>
----> 1 Integer(1)/Integer(0)

## 'Apenas' 13 linhas omitidas ##

ZeroDivisionError: rational division by zero
```

De toda forma, ao se deparar com um erro, duas partes serão importantes: o início, onde você verá o seu próprio código e uma seta indicando a linha onde o erro ocorreu, e a última linha, onde o *sagemath* te informará qual foi o erro. No nosso caso, só há uma linha, o `1/0` (aparecendo como `Integer(1)/Integer(0)`), e a última linha: `ZeroDivisionError: rational division by zero`, há um nome para o erro e uma descrição que, nesse caso, deve ser bem clara (estamos dividindo um número por zero). Sendo baseado no Python, muitos dos erros que você verá são erros do Python, e você pode verificar na documentação os principais tipos de erro. Alguns erros não são tão compreensíveis, mas, a medida que for ganhando experiência você ficará mais familiar com eles [9].

Ajuda Além da documentação que mencionamos, você pode obter ajuda sobre as funções e objetos dentro do próprio *sagemath*. Para ver a documentação de, digamos, `objeto`, basta usar `objeto?`. Se você digitar `objeto??` verá, além da documentação o código que os criadores usaram para implementar o objeto em questão. Para os

[9] Para os conhecedores de Python: Alguns erros que você esperaria no Python podem aparecer de forma diferente no *sagemath*, ou até mesmo não fornecer um erro. Por exemplo, **no Python**, importe a função `log` com `from math import log` e tente calcular `log(0)`. O que ocorre ao tentar isso usando o `log` do *sagemath* (sem importar da biblioteca `math`)?

objetos principais a documentação obtida com um `?` terá explicações aprofundadas e exemplos. Esse tipo de ajuda é particularmente útil para relembrar os parâmetros opcionais de uma função que envolve muitos argumentos, por exemplo, teste `plot?`.

Coerção A coerção que fizemos acima ao tentar converter números entre conjuntos numéricos funciona em casos mais gerais e é uma importante ferramenta do *sagemath*. Veja o exemplo a seguir

```
1 f(x) = x^2 + 1
2 print(f)
3 print("Tipo de f:\t", parent(f))
4 P.<x> = PolynomialRing(QQ)
5 f = P(f)
6 print(f)
7 print("Novo tipo de f:\t", parent(f))
```

```
x |--> x^2 + 1
Tipo de f:   Callable function ring with argument x
x^2 + 1
Novo tipo de f:  Univariate Polynomial Ring in x over Rational
   Field
```

Não importa muito se você não sabe que objetos são esses, o que queremos destacar é que definimos $f(x) = x^2 + 1$, que é considerado como uma função pelo *sagemath*, e depois o convertemos para um polinômio com coeficientes em $\mathbb{Q}$. Apesar de parecidos esses objetos são diferentes e tem métodos diferentes — tente, por exemplo, calcular o grau de f usando o método `f.degree()` antes e depois da linha 5, onde o convertemos para um polinômio. Como o exemplo do `QQ(pi)` mostrou acima, nem sempre é possível fazer a coerção de um tipo de objeto em outro.

Capítulo 1

Divisibilidade

{chap:div}

Nesse capítulo apresentamos os principais conceitos relacionados à divisão.

1.1 Divisão Euclidiana

Sejam $a, b \in \mathbb{Z}$ tais que $b \neq 0$. Pela divisão euclidiana sabemos que é possível dividir a por b e, como resultado, obtemos q, r inteiros tais que $0 \leq r < b$ tais que

$$a = bq + r$$

Além disso os inteiros q e r são únicos com essa propriedade. Vamos relembrar a demonstração desse fato.

Demonstração. Considere o conjunto $R = \{a - bk \mid k \in \mathbb{Z}\}$. Note que, como $b \neq 0$, o conjunto R é infinito. Na verdade, podemos considerar o conjunto R como uma progressão aritmética *nas duas direções* com termo inicial a e razão b. Dessa forma, temos que $S \cap \mathbb{Z}_{\geq 0} \neq \emptyset$. Pelo princípio da boa ordenação existe portanto um elemento minimal em R, digamos $r_0 = a - bk_0$ para algum $k_0 \in \mathbb{Z}$. Temos então que

$$a = r_0 + bk_0$$

Por definição temos que $r_0 \geq 0$. Vale também que $r_0 < |b|$, caso contrário $0 < r_0 - |b| = a + b(k_0 \pm 1)$ seria um elemento de $S \cap \mathbb{Z}_{\geq 0}$ menor que r_0, contradizendo a minimalidade de r_0. Portanto $r = r_0$ e $q = k_0$ satisfazem a hipótese desejada. A unicidade do par q e r fica como exercício para o leitor. □

Obviamente o *sagemath* já possui um comando que calcula, dados inteiros a e $b \neq 0$, o quociente q e o resto r da divisão de a por b. Contudo, vamos aproveitar esse simples problema para nos familiarizar com o *sagemath*. Usaremos os conceitos básicos de listas.

Como vimos, o conjunto R na demonstração é infinito, então não podemos representá-lo no *sagemath* (pelo menos não como uma lista). Fixemos $a = 10$ e $b = 3$ e listemos os elementos de R com k variando de -5 a 5, para isso usaremos a construção por compreensão de listas, apresentada na Introdução.

```
1 a = 10
2 b = 3
3 [a-k*b for k in srange(-5,6)]
```

```
[25, 22, 19, 16, 13, 10, 7, 4, 1, -2, -5]
```

Note que o menor elemento não negativo na lista é o 1, que é, de fato, o resto da divisão de $a = 10$ por $b = 3$, já que $10 = 3 \times 3 + 1$ e $0 \leq r = 1 < 3$. No entanto, ainda que seja possível capturar através do *sagemath* o menor elemento positivo de uma lista, essa tecnica pode não funcionar. Tente criar o subconjunto análogo de S para $a = 1932$ e $b = 7$.

Podemos resolver esse problema usando o *laço de repetição* `while`. Ao invés de criar um subconjunto de R e procurarmos ali o menor não negativo, começamos do a; isto é, $k = 0$, e caminhamos na direção certa para encontrar o resto. Note que a "direção certa" depende dos sinais de a e b. Por exemplo, se tomarmos $a = 10$ e $b = -3$:

```
1 a = 10
2 b = -3
3 [a-k*b for k in srange(-5,6)]
```

```
[-5, -2, 1, 4, 7, 10, 13, 16, 19, 22, 25]
```

A lista está no sentido oposto à anterior! Assim, a partir do 10 devemos caminhar para trás tomando $k = -1, -2, \ldots$, para encontrar o menor não negativo. Tome $a = 39$ e $b = 4$. A célula a seguir toma $k = 0, 1, 2, \ldots$ testando o valor de $a - kb$. Se chegarmos à um valor negativo para determinado k então o menor não negativo foi obtido no passo anterior.

```
1 a = 39
2 b = 4
3 k = 0
4 while a-(k+1)*b >= 0:
5   k += 1
6 print("Quociente:", k)
7 print("Resto:", a-k*b)
```

```
Quociente: 9
Resto:  3
```

Teste esse código com outros valores para a e b. Mas cuidado! Se você trocar o sinal de b o seu algoritmo não vai terminar nunca (por que?). Vejamos com calma o que está acontecendo. Nas primeiras três linhas apenas atribuímos os valores de a, b e o valor inicial de k. Na linha 4 criamos um laço de repetição que testa se o número $a - (k+1)b$ é não negativo. Caso seja, passamos para o próximo valor de k na linha seguinte e voltamos para a linha com o `while`, testando novamente. Caso o valor obtido seja negativo, isso implica que o menor natural não negativo é o $a - kb$, que é o resto, e o quociente é o k. As demais linhas apenas exibem o resultado. Nos exercícios você deverá criar um algoritmo parecido para os demais possibilidades de sinal de a e b.

Obviamente o *sagemath* já possui uma ferramenta para o cálculo do quociente e do resto, o método `.quo_rem` da classe `Integer`.

```
a = 39
b = 4
a.quo_rem(b)
```

```
(9, 3)
```

O retorno é uma tupla onde o primeiro elemento é o quociente e o segundo é o resto... Bem, quase isso: esse método tem um comportamento que requer certo cuidado. Pela nossa definição, o resto **deve ser não negativo**. No entanto, o `.quo_rem` retorna um resto negativo quando $b < 0$.

```
39.quo_rem(-4)
```

```
(-10, -1)
```

De fato, $39 = (-10)(-4) - 1$ mas isso não é a nossa divisão euclidiana de 39 por -4, e sim $39 = (-9)(-4) + 3$ pois $0 \leq r = 3 < |-4| = 4$. Uma forma mais direta de se encontrar o resto é usar o operador `%`.

```
39 % 4
```

```
3
```

No entanto ainda temos um resto negativo quando $b < 0$. Uma maneira de se obter o resto no intervalo esperado é usar o método `.mod`.

```
mod(39,-4)
```

```
3
```

Discutiremos esse método nos próximos capítulos, mas destacamos que o resultado retornado pelo `.mod` não é um inteiro (você pode descobrir o tipo de um objeto com a função `parent`, tente!).

Quando o resto da divisão euclidiana de a por b é zero dizemos que b *divide* a e denotamos essa relação por $b \mid a$. O *sagemath* sabe nos dizer quando isso acontece:

```
2.divides(37); 5.divides(25)
```

```
False
True
```

Se b divide a dizemos que b é um *divisor* ou *fator* de a, ou que a é um múltiplo de b. O *sagemath* consegue encontrar todos os divisores de um número dado com a função `divisors`.

```
divisors(12) ; divisors(7)
```

```
[1, 2, 3, 4, 6, 12]
[1, 7]
```

Note que a lista contém apenas os divisores positivos, mas, pela nossa definição, segue que se $b \mid a$ então $-b \mid a$, de forma que os negativos também são considerados divisores.

1.2 Primos

Esse é um bom momento para relembrar a definição de número primo. Um natural $p > 1$ é dito *primo* se seus únicos divisores positivos são 1 ou p. Você poderia testar se um número n é primo verificando se sua lista de divisores é `[1,p]`, mas o método `.is_prime` já faz isso:

```
12.is_prime() ; 19.is_prime()
```

```
False
True
```

Números primos são extremamente importantes na teoria dos números. Apresentamos algumas ferramentas do *sagemath* envolvendo primos. Além de verificar se um dado natural é primo o *sagemath* consegue gerar e listar os primos. A função `primes_first_n(n)` retorna os n primeiros primos, a função `primes(a,b)` retorna os primos entre a e $b-1$ e `random_prime(n)` retorna um primo aleatório menor ou igual a n. Vejamos alguns exemplos

```
print("Os 10 primeiros primos sao", primes_first_n(10))
print("Os primos entre 50 e 80 sao", list(primes(50, 81)))
print("Um primo menor que 10^10:", random_prime(10^10))
```

```
Os 10 primeiros primos sao [2, 3, 5, 7, 11, 13, 17, 19, 23, 29]
Os primos entre 50 e 80 sao [53, 59, 61, 67, 71, 73, 79]
Um primo menor que 10^10: 3943466743
```

O comando `primes(a,b)` na verdade não retorna uma lista, e sim um objeto iterável, não importa muito agora o que isso significa, mas você deve saber que podemos transformá-lo em uma lista, como vimos no exemplo. A função `random_prime` será bastante útil nas nossas investigações, com ela podemos também passar um limite inferior para o primo desejado, usando o parâmetro `lbound`, de forma que `random_prime(a,lbound=b)` irá retornar um primo entre a e b.

```
random_prime(10^20, lbound=10^19)
```

```
27132302867607456901
```

Note que esse comando é equivalente a escolher um elemento aleatório da lista dos primos entre 10^{19} e 10^{20}. Isso pode ser feito com a função `choice`, que escolhe um elemento aleatório de uma lista, nosso comando então seria da forma `choice(list(primes(a,b+1)))`. Contudo esse método é bem mais lento pois gera todos os primos entre a e b para depois escolher um deles aleatoriamente. Obviamente, métodos envolvendo uma escolha aleatória irão retornar resultados possivel-

mente diferentes cada vez que forem executados. A possibilidade de gerar primos, sobretudo primos com muitos dígitos, é essencial para a criptografia, como discutiremos na seção **??**

Os métodos envolvendo primos, como o `.is_prime()`, e diversos outros métodos de aritmética, podem se tornar mais eficientes se permitimos que eles utilizem algoritmos probabilísticos. Isso significa que existe uma probabilidade muito muito muito baixa de tais métodos retornarem resultados errados, por exemplo, um número composto para `.random_prime()`. Existe uma forma de avisar ao *sagemath* se você prefere ou não utilizar algoritmos probabilísticos ou resultados não provados em seus cálculos através da função `proof.arithmetic()`, veja [1, Global proof preferences]. Discutiremos brevemente a ideia de pseudoprimo e do funcionamento de algoritmos probabilísticos no capítulo **??**.

1.3 MDC

{sec:mdc}

Um dos conceitos mais importantes da aritmética básica é o de *máximo divisor comum*: Dados dois inteiros a e b não ambos nulos, o máximo divisor comum entre a e b, denotado por $\text{mdc}(a,b), \gcd(a,b)$ ou simplesmente (a,b), é o maior inteiro possível dividindo ambos a e b, é portanto o maior elemento possível da interseção entre o conjunto dos divisores de a e de b. Como esses conjuntos são finitos e 1 pertence a ambos os conjuntos, o mdc é sempre um inteiro positivo. Se $\text{mdc}(a,b) = 1$ dizemos que a e b são *coprimos, relativamente primos ou primos entre si*.

Podemos encontrar o mdc entre a e b no *sagemath* encontrando o maior elemento presente em ambas as listas `divisors(a)` e `divisors(b)`. Vejamos uma forma de fazer isso:

```
a = 324
b = 212
max([d for d in divisors(a) if d in divisors(b)])
```

```
4
```

A compreensão de listas cria uma lista nova com os elementos de `divisors(a)` que estão também em `divisors(b)`, ou seja, a interseção de tais listas, em seguida, usando a função `max`, encontramos o maior elemento dessa lista, obtendo portanto o $\text{mdc}(a,b)$. Como não poderia deixar de ser, o *sagemath* já possui uma função que calcula o mdc, o `gcd`:

```
gcd(324,212)
```

```
4
```

Há uma questão que evitamos até agora mas esse é um bom momento para discutir. Será que o que estamos fazendo é eficiente? Por exemplo, para descobrir o mdc acima fizemos a interseção entre os divisores de a e de b. Porém, encontrar a lista dos divisores de um número é equivalente a encontrar sua fatoração em primos e, como veremos mais adiante, fatorar um número pode ser muito lento se o número for muito grande. Como um teste, tente calcular o $\text{mdc}(10^{50}+1, 10^{49}-1)$ usando a in-

terseção de listas e usando o comando `gcd`. Você deverá notar que o comando nativo `gcd` é muito mais rápido. Isso acontece pois o *sagemath* está usando internamente o *algoritmo de Euclides* para o cálculo do mdc. Antes de apresentar tal algoritmo, um lema.

{lemma:mdc}

Lema 1.1. *Sejam $a, b \in \mathbb{Z}$ não ambos nulos. Então* $\text{mdc}(a,b) = \text{mdc}(b, a-kb)$ *para todo* $k \in \mathbb{Z}$. *Em particular, se r é o resto da divisão euclidiana de a por b, então* $\text{mdc}(a,b) = \text{mdc}(b,r)$.

Demonstração. Exercício. □

Como o resto da divisão de a por b é necessariamente menor que $|b|$, na prática estamos trocando o cálculo do mdc desejado pelo cálculo do mdc de dois números menores, com a vantagem de não termos usado a fatoração ou a lista de divisores dos números envolvidos. Essa vantagem se mostrará essencial em diversos momentos.

O algoritmo de Euclides pode então ser descrito da seguinte forma, dados inteiros a, b não nulos tais que $a = bq_1 + r_1$ é a divisão euclidiana de a por b, o lema afirma que $\text{mdc}(a,b) = \text{mdc}(b, r_1)$. Repetimos a ideia com b e r_1, se $b = r_1 q_2 + r_2$ é a divisão euclidiana de b por r_1, novamente pelo lema temos $\text{mdc}(b, r_1) = \text{mdc}(r_1, r_2)$. Assim obtemos uma sequência de igualdades

$$\text{mdc}(a,b) = \text{mdc}(b,r_1) = \text{mdc}(r_1,r_2) = \text{mdc}(r_2,r_3) = \cdots$$

e essa sequência certamente acabará uma vez que $r_1 > r_2 > \cdots \geq 0$, daí usamos que $\text{mdc}(r_k, 0) = r_k$.

ex:mdceucl}

Exemplo 1.1. Calculemos o $\text{mdc}(342, 101)$ usando o algoritmo de Euclides.

$$\begin{aligned}
342 &= 3 \times \boxed{101} + \textcircled{39} \\
\boxed{101} &= 2 \times \boxed{39} + \textcircled{23} \\
\boxed{39} &= 1 \times \boxed{23} + \textcircled{16} \\
\boxed{23} &= 1 \times \boxed{16} + \boxed{7} \\
\boxed{16} &= 2 \times \boxed{7} + \boxed{2} \\
7 &= 3 \times \boxed{2} + \boxed{1} \\
2 &= 2 \times 1 + 0.
\end{aligned}$$

Pelo algoritmo de Euclides temos portanto

$$\begin{aligned}
\text{mdc}(342,101) &= \text{mdc}(101,39) = \text{mdc}(39,23) \\
&= \text{mdc}(23,16) = \text{mdc}(16,7) \\
&= \text{mdc}(7,2) = \text{mdc}(2,1) \\
&= \text{mdc}(1,0) = 1.
\end{aligned}$$

Observe que, usando a fatoração em primos, poderíamos calcular facilmente o mdc acima. No entanto, como discutimos, para números grandes, com potencialmente centenas de dígitos, o algoritmo de Euclides é muito mais eficiente que o cálculo do mdc usando a fatoração. Para implementar o algoritmo de Euclides usamos novamente o laço `while`.

```
1 a = 342
2 b = 101
3 (q,r) = (b,a.quo_rem(b)[1])
4 while r != 0:
5   (q,r) = (r,q.quo_rem(r)[1])
6
7 print("mdc({},{})={}".format(a,b,q))
```

```
mdc(342,101)= 1
```

Vamos entender o que está acontecendo. A linha 3 associa a `q` o valor de `b` e associa a `r` o segundo valor da lista `a.quo_rem(b)` (lembre que listas começam a ser indexadas do 0, logo o elemento de índice 1 é o segundo elemento), portanto, associa a `r` o valor do resto da divisão de a por b. O *sagemath* faz essas atribuições *simultaneamente*, observamos a seguir porque isso é importante.

- Se $b \mid a$ temos que o resto da divisão é 0 e $\text{mdc}(a,b) = b$, e esse é o resultado obtido pois, na ultima linha `q` terá o valor de `b`.
- Se $b \nmid a$, o resto da divisão será não nula e o algoritmo entrara dentro do laço `while`. Dentro dele atribuímos a q o valor antigo de r e a r o resto da divisão do valor anterior de q pelo r, isto é, em cada passo o q faz o trabalho do r_i e o r do r_{i+1}. Em seguida testamos novamente se $r = 0$ e repetimos o processo.

Como vimos, eventualmente o resto será nulo, assim o mdc será o valor do último resto, que estará guardado na variável `q`, portanto a última linha sempre exibe o mdc entre a e b. Como estamos usando o método `.quo_rem`. que não retorna o resto esperando $0 \leq r \leq |b|$, nosso algoritmo em tese funcionaria apenas para valores positivos de a e b. Verifique o que acontece se mudarmos o sinal de a ou b. Você consegue explicar esse resultado? Tente "consertar" o algoritmo usando a função `abs(n)` que retorna o valor absoluto de n. Outro pequeno problema com nosso algoritmo é que se $b = 0$, um erro será retornado pois estaríamos fazendo uma divisão por 0 em `a.quo_rem(b)`. Para evitar isso basta testar anteriormente se $b = 0$, nesse caso $\text{mdc}(a,0) = a$ (lembre que apenas um dos valores de a ou b pode ser 0, então se $b = 0$ devemos ter $a \neq 0$.) Nos exercícios você aprenderá replicar o algoritmo para o cálculo do mdc usando o conceito de *recursão*.

Uma observação importante: ao tomar `(q,r) = (r,q.quo_rem(r)[1])` destacamos que a atribuição acontece de forma simultânea, também chamada de atribuição paralela. Isso significa que utiliza-se o mesmo valor de q nas primeira e segunda coordenadas, ou seja, apesar de atualizarmos o valor de q na primeira coordenada, na segunda, ao atualizar o valor de r, usamos o valor de q antigo. Se as atribuições fossem feitas uma a uma, em linhas diferentes, o resultado não seria o mesmo. Vejamos um exemplo mais transparente, vamos atribuir o valor 2 à variável `n`, em seguida atribuímos ao próprio `n` o seu dobro e a uma variável nova `m` o triplo de `n`.

```
1 n = 2
2 (n,m) = (2*n,3*n)
3 print(n,m)
```

```
4 6
```

Como era esperado, agora se fizermos essas atribuições uma de cada vez...

```
1 n = 2
2 n = 2*n
3 m = 3*n
4 print(n,m)
```

```
4 12
```

... o que faz sentido já que, na linha 3 ao atribuir a `m` o valor de `3*n`, o `n` já foi alterado pra o seu dobro, de forma que $m = 2n = 3 \times (2 \times 2) = 12$. Algumas linguagens de programação não permitem atribuições paralelas, sendo necessário a criação de uma variável temporária.

Exercício 1.1. Escreva um código *sagemath* que troque o valor de duas variáveis a e b. Com atribuição paralela isso pode ser feito em uma linha, sem atribuição paralela será necessário criar uma variável extra, mas, se as variáveis forem números inteiros, é possível trocar os valores sem uma variável extra (Dica: você pode operar esses números).

Um teorema de grande importância teórica é o Teorema de Bézout-Bachet, que afirma que o mdc(a, b) pode ser escrito como uma combinação linear inteira de a e b.

thm:bezout}

Teorema 1.2 (Bézout-Bachet)**.** *Sejam $a, b \in \mathbb{Z}$ não ambos nulos. Existem $x, y \in \mathbb{Z}$ satisfazendo*

$$ax + by = \text{mdc}(a, b)$$

Uma importante consequência do teorema é que se $c \mid a$ e $c \mid b$, então $c \mid ax + by = \text{mdc}(a, b)$. Essa propriedade aparece as vezes na definição de mdc pois é a propriedade utilizada para generalizar o conceito de mdc para estruturas mais gerais, sem uma ordem padrão como os inteiros.

A demonstração do teorema de Bézout-Bachet segue uma linha parecida com o da divisão euclidiana. Definimos um conjunto S formado por todas as combinações lineares inteiras de a e b e mostramos que o menor elemento não negativo desse conjunto é exatamente o mdc(a, b). No entanto, pela bidimensionalidade do problema, tentar encontrar o mdc como menor elemento do conjunto S não parece muito tentador. De fato, se testarmos todas as possibilidades de x e y em um determinado intervalo, digamos, em $[-K, K], K > 0$, e seguirmos aumentando o K eventualmente encontraremos um dos pares x e o y do teorema. No entanto existe um outro método mais eficiente para encontrar esses coeficientes. Esse algoritmo é conhecido como *algoritmo de Euclides estendido*. O algoritmo de Euclides que calcula o mdc, com o auxílio do Lema **??**, pode ser usado para encontrar os coeficientes x e y do Teorema 1.2. Vejamos primeiramente um exemplo com números.

Exemplo 1.2. Usaremos o cálculo do mdc feito no Exemplo 1.1. A ideia é partir da penúltima divisão, isolar o resto, que é o mdc e, de baixo para cima, substituir o divisor pela sua expressão como o resto da divisão anterior. É importante não efetuar as multiplicações divisor $\times$ quociente ao substituir pois iremos colocá-lo em

evidência no passo seguinte, já que o quociente é o resto da divisão anterior. Observe o exemplo a seguir

$$\begin{aligned}
1 &= 7 - 3 \times \boxed{2} \\
&= 7 - 3 \times (16 - 2 \times 7) \\
&= 7 \times \boxed{7} - 3 \times 16 \\
&= 7 \times (23 - 1 \times 16) - 3 \times 16 \\
&= 7 \times 23 - 10 \times \boxed{16} \\
&= 7 \times 23 - 10 \times (39 - 1 \times 23) \\
&= 17 \times \boxed{23} - 10 \times 39 \\
&= 17 \times (101 - 2 \times 39) - 10 \times 39 \\
&= 17 \times 101 - 44 \times \boxed{39} \\
&= 17 \times 101 - 44 \times (342 - 3 \times 101) \\
&= (-44) \times 342 + (149) \times 191
\end{aligned}$$

Para implementar o algoritmo de Euclides estendido mudamos um pouco a estratégia e vamos tentar sistematizar as contas efetuadas. Note que, se tivéssemos começado as contas acima de outra divisão anterior, e não da última, que fornece o $\text{mdc}(a, b)$, todos os restos das divisões intermediárias $r_i = q_{i+2}r_{i+1} + r_{i+2}$ poderiam ser escritos como combinação inteira de a e b. Escreva

$$\begin{array}{rcccl}
x_i a + y_i b & = & r_i & = & q_{i+2}r_{i+1} + r_{i+2} \\
x_{i+1} a + y_{i+1} b & = & r_{i+1} & = & q_{i+3}r_{i+2} + r_{i+3}
\end{array}$$

Assim

$$\begin{aligned}
r_{i+2} &= r_i - q_{i+2}r_{i+1} \\
&= x_i a + y_i b - q_{i+2}(x_{i+1}a + y_{i+1}b) \\
&= (x_i - q_{i+2}x_{i+1})a + (y_i - q_{i+2}y_{i+1})b
\end{aligned}$$

A última identidade mostra como podemos escrever r_{i+2} como combinação inteira de a e b com coeficientes dependendo dos coeficientes em que os dois restos anteriores são escritos como combinação inteira de a e b e do quociente q_{i+2}. Os dois primeiros valores para os x_i, y_i podem ser obtidos pelas divisões iniciais:

$$\begin{aligned}
a &= bq_0 + r_0 \\
b &= q_1 r_0 + r_1
\end{aligned}$$

Assim

$$\begin{aligned}
r_0 &= (1)a + (-q_0)b \\
r_1 &= b - qr_0 = b - q_1(a - bq_0) = (-q_0)a + (1 + q_0 q_1)b
\end{aligned}$$

No código *sagemath* abaixo usamos as relações obtidas acima para encontrar os coeficientes de a e b na identidade $ax + by = 1$. Note que,

```
1 a = 999
2 b = 212
3 (q0,r0) = a.quo_rem(b)
4 (q1,r1) = b.quo_rem(r0)
5 (x0,y0,x1,y1) = (1,-q0,-q1,1+q0*q1)
6 while r1 > 0:
7     (q0,r0,q1,r1) = (q1,r1,r0.quo_rem(r1)[0],r0.quo_rem(r1)[1])
8     (x0,y0,x1,y1) = (x1,y1,x0-q1*x1,y0-q1*y1)
```

```
9
10 print(r0,x0,y0)
```

```
1 -73 344
```

E, de fato, $-73 \times 999 + 344 \times 212 = 1$. A ideia é que, a cada passo, os `x0` e `y0` e os `x1` e `y1` fazem os trabalhos dos x_i e y_i e dos x_{i+1} e y_{i+1}, respectivamente. Enquanto que, como no algoritmo de Euclides que calcula o mdc, visto em REF, `q1` `r1` fazem os papeis de q_i e r_i.

O algoritmo Euclides estendido será útil em outros dois momentos futuros: ao procurar pelo inverso modular, no Capítulo **??**, e ao resolvermos a equação diofantina linear $ax + by = c$ no caso geral, no Capítulo **??**. Naturalmente, o sage possui um comando que executa o algoritmo de Euclides estendido, o `xgcd`.

```
1 xgcd(999,212)
```

```
(1, -73, 344)
```

A função `xgcd(a,b)` retorna uma 3-upla cujo primeiro elemento é o $\text{mdc}(a,b)$, e o segundo e terceiro elementos são, respectivamente o x e y satisfazendo $ax + by = \text{mdc}(a,b)$. Vale observar que o par (x,y) obtido pelo algoritmo descrito acima não é o único satisfazendo $ax + by = \text{mdc}(a,b)$. A solução geral será discutida na Seção **??**. Nos exercícios ao final desse capítulo você verá outras duas formas de se implementar o algoritmo de Euclides estendido.

1.4 Teorema Fundamental da Aritmética

A importância dos números primos fica evidente no Teorema Fundamental da Aritmética

{TFA}

Teorema 1.3 (Teorema Fundamental da Arimética)**.** *Seja $n \in \mathbb{N}$, $n > 1$. Então n pode ser fatorado como*

$$n = p_1^{e_1} p_2^{e_2} \cdots p_r^{e_r},$$

onde os p_i são primos distintos e $e_i \in \mathbb{N}$. Além disso a fatoração em primos é única a menos de um reordenamento dos primos. Em outras palavras, a fatoração é única se supomos $p_1 < p_2 < \cdots < p_r$.

A demonstração do teorema fundamental da aritmética é relativamente simples e pode ser encontrada em qualquer livro de teoria dos números, veja, por exemplo, [5, Sec 1.3]. Uma resultado fundamental utilizado em sua demonstração é o fato de que se p é primo e $p \mid ab$, com $a, b \in \mathbb{Z}$, então $p \mid a$ ou $p \mid b$.

A técnica mais ingênua para se encontrar a fatoração de um dado $n > 1$ é encontrar um primo $p \mid n$. Como $n = pq$ para algum $q \in \mathbb{N}$, reduzimos o problema para encontrar a fatoração de q. Repetindo esse procedimento com $n \leftarrow q$, como $q < n$, eventualmente chegaremos no caso em que o q é primo.

Aproveitamos a ocasião para falar um pouco sobre *recursividade*. A ideia de um processo recursivo é que, após definidos alguns casos iniciais, os demais casos são resolvidos através de regras que os reduzem para algum caso inicial. Um exemplo

familiar de definição por recursão é a definição do *fatorial* de um inteiro $n \geq 1$. Definimos o fatorial de 0 como 1, ou seja, $0! = 1$, e, para $n \geq 1$, definimos $n! = n \times (n-1)!$. Apesar da definição de mdc não ter sido recursiva, o leitor mais atento deve ter observado que, no algoritmo de Euclides para o cálculo do mdc, definimos o caso inicial $\text{mdc}(n, 0) = n$ e calculamos o mdc recursivamente utilizando o Lema 1.1 sucessivas vezes até nos depararmos com o caso inicial.

A técnica que descrevemos para se fatorar um número $n > 1$ foi uma técnica recursiva já que, após encontrar um primo p dividindo n, passamos ao problema de fatorar o número $q = n/p$, que é menor que n. Observe que esse processo eventualmente termina, isso ocorre quando o último primo dividindo n for encontrado, caso em que o quociente será 1.

```
1  def fat(n, fatores = []):
2      if n == 1:
3          return fatores
4      p = 2
5      while  not p.divides(n):
6          p = p.next_prime()
7
8      return fat(n/p,fatores+[p])
9
10 print(fat(1729))
```

```
[7, 13, 19]
```

Vamos analisar o código acima. Estamos criando uma função para a fatoração de n. Usamos uma lista auxiliar `fatores` para guardar os fatores primos de n. Inicialmente, não sabemos nenhum fator de n, por isso, na última linha, não passamos o parâmetro l para a função `fat` [1]. Iniciamos verificando se o $n = 1$, nesse caso não há nenhum primo dividindo n e retornamos a lista vazia. Em seguida tomamos o primeiro primo $p = 2$ e criamos um laço `while` que testa se $p \mid n$:

- Se $p \nmid n$, então o atribuímos a p o valor do próximo primo usando o método `.next_prime`, essa atribuição é repetida até que encontremos o primeiro primo p que divide n.

- Se $p \mid n$, retornamos a própria função `fat`, agora sendo calculada em n/p, e passando a lista de fatores adicionada do primo p encontrado.

Esse processo é repetido até que o último fator primo de n seja encontrado, nesse caso a condição do `if n == 1` será satisfeita e o programa irá retornar a lista com todos os fatores, voltando todos os níveis anteriores. Vejamos um diagrama que mostra o fluxo do programa para $n = 21$.

[1] Note que, na primeira linha do código, atribuímos `fatores = []`. Portanto, quando a função for chamada sem o segundo argumento ele será considerado como a lista fazia `[]`, e isso só ocorrerá na primeira vez.

$$\text{fat}(21) \to \begin{cases} p = 2 \nmid 21 \\ p = 3 \mid 21 \end{cases} \to \text{fat}\left(\frac{21}{3}, [3]\right) \to \begin{cases} p = 2 \nmid 7 \\ p = 3 \nmid 7 \\ p = 5 \nmid 7 \\ p = 7 \mid 7 \end{cases} \to \text{fat}\left(\frac{7}{7}, [3,7]\right) \to n = 1 : [3,7]$$

Observe que nossa função `fat` retorna uma lista com a fatoração em primos de n *por extenso*, ou seja, os primos que dividem n com potência maior que 1 aparecerão mais de uma vez na lista. Podemos verificar se a fatoração está correta considerando o produto de todos os elementos da lista usando a função `prod`

{teste}

```
1 fat99 = fat(99)
2 print(fat99)
3 print(prod(fat(99)) == 99)
```

```
[3, 3, 11]
True
```

Apesar de funcionar nosso algoritmo não é muito eficiente. Note que em cada vez que a função `fat` é executada iniciamos a nossa procura por fatores primos a partir do 2, mas isso não é necessário, uma vez que, se as execuções sucessivas de são feitas com fatores do n inicial, se $2 \nmid n$, então 2 não vai dividir nenhum fator de n. Isso poderia ser resolvido utilizando uma variável global para guardar o valor do primo da vez, ou tomando p como o maior primo da lista `fatores` caso essa lista seja não vazia. Também testamos primos maiores que $\sqrt{n}$, o que é desnecessário. Por simplicidade não optamos por nenhuma dessas otimizações. Nos exercícios você irá criar um algoritmo para fatoração sem usar a recursividade.

Além do discutido, soluções que fazem o uso da recursividade, ainda que elegantes, geralmente não são recomendáveis. Uma das desvantagens da recursividade é o grande uso de memória. Algumas linguagens de programação possuem uma profundidade máxima de recursão; isto é, uma função pode ser executada dentro de si mesma uma quantidade máxima de vezes. Tente usar a função `fat` que criamos para encontrar a fatoração de um número com muitos fatores primos. A partir de certa quantidade de primos (mesmo repetidos) você deve se deparar com um erro como `RecursionError: maximum recursion depth exceeded` (profundidade máxima de recursão excedida).

Naturalmente, o *sagemath* possui uma ferramenta para encontrar a fatoração de um número n, a função `factor`.

```
1 factor(420)
```

```
2^2 * 3 * 5 * 7
```

Observe que, ao contrário da nossa função `fat`, a função `factor` agrupa os primos iguais em uma só fator, com expoente igual a quantidade de vezes que esse primo aparece na fatoração. Na verdade a fatoração no *sagemath* retorna um objeto de uma classe chamada `Factorization` (nesse caso, da subclasse `IntegerFactorization`). Se isso não fez muito sentido para você, não se preocupe. O importante é que podemos transformar a fatoração de um número em uma lista:

```
n = 2288
F = factor(n)
print(F)
print(list(F))
```

```
2^4 * 11 * 13
[(2, 4), (11, 1), (13, 1)]
```

A lista obtida é composta de pares (p,e), onde p é um número primo dividindo n enquanto e representa a potência máxima de p dividindo e; isto é, é o expoente de p na fatoração única de n. Na Subseção 1.7.1 você verá como manipular a fatoração como um dicionário.

O enunciado do Teorema 1.3 trata apenas de naturais $n > 1$, mas é natural que se $n < -1$ também podemos fatorar o n de forma única, basta adicionar um -1 na fatoração de n. É exatamente isso que o *sagemath* faz

```
factor(-10)
```

```
-1 * 2 * 5
```

Se estivermos interessados apenas nos fatores primos de um número, podemos usar a função `prime_divisors`.

```
prime_divisors(2^2*3^3*5^5)
```

```
[2,3,5]
```

1.5 Frações Contínuas

Voltemos ao Exemplo 1.1. Divivindo ambos os lados da primeira identidade por 101, obteremos

$$\frac{342}{101} = 3 + \frac{39}{101} = 3 + \frac{1}{\frac{101}{39}}$$

Agora note que se dividirmos a segunda identidade por 39, obteremos do lado esquerdo uma expressão para $\frac{101}{39}$, dada por $\frac{101}{39} = 2 + \frac{23}{39}$. Assim, nossa identidade inicial pode ser escrita como

$$\frac{342}{101} = 3 + \frac{1}{2 + \frac{23}{39}} = 3 + \frac{1}{2 + \frac{1}{\frac{39}{23}}}$$

Observe que podemos repetir o processo anterior com a fração $\frac{39}{23}$, obtendo a expressão

$$\frac{342}{101} = 3 + \frac{1}{2 + \frac{1}{1 + \frac{1}{\frac{23}{16}}}}$$

Após seguir o procedimento com todas as divisões, obtemos

$$\frac{342}{101} = 3 + \cfrac{1}{2 + \cfrac{1}{1 + \cfrac{1}{1 + \cfrac{1}{2 + \cfrac{1}{3 + \frac{1}{2}}}}}} \tag{1.1}$$ {eq:fraccon

O que fizemos acima foi encontrar a representação de $\frac{342}{101}$ por *frações contínuas* . Denotamos a sequência acima por $[3; 2, 1, 1, 2, 3, 2]$. Podemos definir um procedimento análogo no caso mais geral em que x é um número real: dado $x \in \mathbb{R}$, definimos de forma recursiva $\alpha_0 = x$, $a_n = \lfloor \alpha_n \rfloor$ e, se $\alpha_n \notin \mathbb{Z}$, $\alpha_{n+1} = \frac{1}{\alpha_n - a_n}$.

Se acontecer $\alpha_n = a_n$, temos

$$x = \alpha_0 = [a_0; a_1, \ldots, a_n] = a_0 + \cfrac{1}{a_1 + \cfrac{1}{a_2 + \ddots + \frac{1}{a_n}}}.$$

Caso $\alpha_n \neq a_n$ para todo n, a expansão é infinita, denotada por

$$x = \alpha_0 = [a_0; a_1, a_2 \ldots] = a_0 + \cfrac{1}{a_1 + \cfrac{1}{a_2 + \ddots}}.$$

Exercício 1.2. Verifique que os cálculos feitos no início da seção para $\frac{342}{101}$ correspondem à definição acima.

Note que os termos da fração contínua de a/b são exatamente os quocientes na sequência de divisões euclidianas que usamos para obter $\mathrm{mdc}(a, b)$. Vejamos o funcionamento da expressão em frações contínuas quando x não é racional. Tome $x = \pi$, então $\alpha_0 = \pi \notin \mathbb{Z}$ e $a_0 = \lfloor \pi \rfloor = 3$, portanto

$$\alpha_1 = \frac{1}{\alpha_0 - a_0} = \frac{1}{3 - \pi} = \frac{1}{0.1415926535\ldots} = 7.0625133\ldots$$

Repetindo o procedimento temos $a_1 = \lfloor \alpha_1 \rfloor = 7$, logo

$$\alpha_2 = \frac{1}{\alpha_1 - a_1} = \frac{1}{7.0625133\cdots - 7} = 15.99659440\ldots$$

Assim

$$\pi = 3 + \cfrac{1}{7 + \cfrac{1}{15 + \ddots}}$$

Exercício 1.3. Crie um código que calcule os 10 primeiros termos da representação por frações contínuas de π. Note que, se $x > 0$, então $x - \lfloor x \rfloor$ é a parte fracionária de x, que pode ser encontrada no *sagemath* através da função `frac`.

No *sagemath* podemos utilizar a função `continued_fraction` para encontrar a representação como fração contínua de um número real:

```
continued_fraction(e)
```

```
[2; 1, 2, 1, 1, 4, 1, 1, 6, 1, 1, 8, 1, 1, 10, 1, 1, 12, 1, 1,
   ...]
```

A função `show(continued_fraction` exibirá a fração contínua como em (1.1).

```
show(continued_fraction(e))
```

$$2+\cfrac{1}{1+\cfrac{1}{2+\cfrac{1}{1+\cfrac{1}{1+\cfrac{1}{4+\cfrac{1}{1+\cfrac{1}{1+\cfrac{1}{6+\cfrac{1}{1+\frac{1}{\ddots}}}}}}}}}}$$

Uma das aplicações das frações contínuas é que elas fornecem boas aproximações para números irracionais. No exemplo acima, se truncarmos a expansão em $3+\frac{1}{7}=\frac{22}{7}$ obtemos

$$\left|\pi-\frac{22}{7}\right|\approx 0.00126448,\ \text{enquanto}\ \left|\pi-\frac{314}{100}\right|\approx 0.00159265,$$

se truncarmos em $3+\cfrac{1}{7+\frac{1}{15}}=\frac{333}{106}$ obtemos

$$\left|\pi-\frac{333}{106}\right|\approx 0.00008321962,\ \text{enquanto}\ \left|\pi-\frac{31415}{10000}\right|\approx 0.0000926535.$$

As frações obtidas pelo truncamento de frações contínuas são chamadas de *reduzidas* ou *convergentes*. Em certo sentido, as aproximações de irracionais obtidas através de convergentes de suas frações contínuas são as melhores aproximações possíveis. Podemos encontrar as reduzidas usando o método `.convergent` dos objetos definidos a partir da função `continued_fraction`:

```
cp = continued_fraction(pi)
print(cp.convergent(1))
print(cp.convergent(2))
print(cp.convergent(10))
```

```
22/7
333/106
4272943/1360120
```

Um importante resultado é que todo número real tem uma representação por frações contínuas. Os racionais obviamente tem uma representação finita e essencialmente única, enquanto os irracionais tem uma representação única e infinita. Um resultado interessante afirma que os reais que tem expansão em frações contínuas periódicas são exatamente as raízes das equações quadráticas com coeficientes inteiros: a razão áurea, por exemplo, que será abordada na subseção Sequências Recorrentes e Expressões Simbólicas da Seção 1.7, é dada por $\frac{1+\sqrt{5}}{2} = [1;1,1,1,1,1,...]$. É possível obter o período da recorrência, no entanto é necessário lidar com o número como um elemento de um corpo quadrático. Na célula a seguir exibimos a fração periódica de $\frac{7+2\sqrt{5}}{2}$ primeiramente como número real e, em seguida, como um elemento de $\mathbb{Q}[\sqrt{5}] = \{a+b\sqrt{5} \mid a,b \in \mathbb{Q}\}$. O asterisco indica o período na própria representação.

```
print("Como numero real:", continued_fraction((7+2*sqrt(5))/2))
K.<sqrt5> = QuadraticField(5)
cf = continued_fraction((7+2*sqrt5)/2)
print("Como raiz quadratica:", cf)
print("Periodo:", cf.period())
```

```
Como numero real: [5; 1, 2, 1, 3, 1, 2, 1, 3, 1, 2, 1, 3, 1, 2,
   1, 3, 1, 2, 1, ...]
Como raiz quadratica: [5; (1, 2, 1, 3)*]
Periodo: (1, 2, 1, 3)
```

Uma exposição completa sobre frações contínuas, com destaque para aproximações de números irracionais, pode ser encontrada em [5, Cap. 3], indicamos também [6] e suas referências.

1.6 Aritmética em Outros Aneis

- einsenstein
- gaussianos

1.7 Explore!

c1:explore}

1.7.1 Quantos zeros no final de n!?

ar:zerofat}

No exercício 1.21 você vai aprender a calcular o fatorial de um inteiro $n \geq 0$. Por enquanto vamos permitir que você use a função `factorial` do *sagemath*.

```
factorial(33)
```

```
8683317618811886495518194401280000000
```

Tente trocar o argumento da função fatorial por outros inteiros e você deverá notar que para $n \geq 5$ o fatorial sempre termina com 0. É um exercício interessante de teoria dos números descobrir com quantos zeros termina $n!$. Após refletir um pouco você deverá observar que a quantidade de zeros no final de um número é o k máximo tal

que $10^k \mid n$. Como $10 = 2 \times 5$, bastaria contar a potência máxima simultanea de 2 e 5 na fatoração de $n!$.

```
1 factor(factorial(33))
```

```
2^31 * 3^15 * 5^7 * 7^4 * 11^3 * 13^2 * 17 * 19 * 23 * 29 * 31
```

A potência máxima de 2 e 5 que divide 33! é portanto 7, que é de fato o número de zeros no final de 33!. Na verdade, como $n!$ é o produto dos naturais menores que n, e existem mais multiplos de 2 do que múltiplos de 5, basta conhecer a potência de 5 na fatoração de $n!$. Isso pode ser feito transformando a fatoração de $n!$ em um dicionário e tomando o valor da chave 5:

```
1 m = factorial(33)
2 print(dict(factor(m)))
3 dict(factor(m))[5]
```

```
{2: 31, 3: 15, 5: 7, 7: 4, 11: 3, 13: 2, 17: 1, 19: 1, 23: 1,
   29: 1, 31: 1}
7
```

Um dicionário no *sagemath* funciona como uma lista mas, ao invés de usar inteiros $0, 1, \ldots$ como índices, pode se usar quase qualquer objeto, chamado aqui de chave. O elemento correspondente à chave é chamado de valor. No caso do `dict(factor(m))` as chaves são os primos na fatoração de m e o valor correspondente da chave p, recuperada como numa lista `dict(factor(m))[p]`, será a potência do primo p na fatoração de m.

Note, no entanto, que usando o método descrito acima estamos calculando a fatoração de $n!$. Já discutimos como isso pode ser lento ou praticamente impossível se n for muito grande. No entanto, não precisamos realmente saber toda a fatoração de $n!$ para encontrar quantos fatores 5 aparecem em sua fatoração. Basta somar quantas vezes o fator 5 aparece em todos os naturais menores ou iguais a n. Podemos contar os múltiplos de 5 menores que n apenas encontrando o quociente da divisão de n por 5, mas observe que isso não dá realmente a quantidade de vezes que o 5 aparece como fator dos naturais menores que n. Por exemplo até o 37 existe 7 mútiplos de 5, a saber $5, 10, 15, 20, 25$, 30 e 37 e, de fato, o quociente da divisão de 37 por 5 é 7 pois $37 = 7 \times 5 + 2$. No entanto, no 37!, o 5 aparece duas vezes no fator 25, assim ele deve ser contado mais uma vez. Se tomarmos o fatorial de $n = 130$ o mesmo vai acontecer com o fator $50 = 2 \times 5^2$, com o $75 = 3 \times 5^2$, com o $100 = 4 \times 5^2$ e com o $125 = 5^3$. Assim, para obter o valor exato iremos somar

- os mútiplos de 5 menores que n, i.e. o quociente de n por 5, em seguida
- os mútiplos de 25 menores que n, i.e. o quociente de n por 25 (note que isso é suficiente já que, apesar de 25 possuir dois fatores 5, um deles já foi contado no primeiro caso), em seguida
- os múltiplos de 125 menores que n, i.e. o quociente de n por 125 (novamente, dois dos fatores 5 já foram contados),
- etc

Uma forma compacta de se escrever essa soma é usar a função piso, $\lfloor x \rfloor$, definida como o maior inteiro menor que x para x real. Deixamos como exercício para o leitor verificar que se a e b são positivos então $\lfloor a/b \rfloor$ é exatamente o quociente na divisão de a por b. Assim, a quantidade de zeros no final de $n!$ é dada por

$$\sum_{k=1}^{\infty} \left\lfloor \frac{n}{5^k} \right\rfloor = \left\lfloor \frac{n}{5} \right\rfloor + \left\lfloor \frac{n}{5^2} \right\rfloor + \left\lfloor \frac{n}{5^3} \right\rfloor + \cdots$$

Note que o somatório é finito uma vez que, eventualmente $5^k > n$ e portanto $\lfloor n/5^k \rfloor = 0$ a partir de certo k. Por exemplo, para $n = 1000$, temos que $5^5 = 3125 > n$. Assim 1000! termina com $\sum_{k=1}^{4} \left\lfloor \frac{1000}{5^k} \right\rfloor$ zeros.

```
n = 1000
floor(n/5) + floor(n/5^2) + folor(n/5^3) + floor(n/5^4)
```

```
249
```

Não vamos colocar a expansão de 1000! aqui mas você pode verificar calcular no sage e confirmar o resultado.

Exercício 1.4. Escreva um código que calcule a quantidade de zeros no final de $n!$ para um n qualquer (Dica: use o laço de repetição `while` para saber quando parar)

Exercício 1.5. Faça uma comparação de tempo entre os dois métodos. A partir de que número fatorar o $n!$ passa a ser impraticável, enquanto que o método alternativo é rápido?

Exercício 1.6 (Desafio)**.** O que discutimos funciona para a representação decimal de um número. Você consegue generalizar nosso argumento e calcular a quantidade de zeros no final de $n!$ quando expressamos esse número em uma base diferente de 10? Escreva um código no *sagemath* que encontre esse número para uma base b arbitrária.

1.7.2 Números Perfeitos

Sabemos que todo natural divide a si mesmo. Um *divisor próprio* de um natural n é um divisor positivo de n diferente do n. A função `divisors` do *sagemath* retorna todos os divisores de um número, incluindo ele mesmo, para obter a lista dos divisores próprios basta remover o próprio número da lista, isso pode ser feito de algumas formas:

```
Dp10 = divisors(10)[:-1]
print(Dp10)
```

```
[1, 2, 5]
```

Como a lista dos divisores é ordenada de forma crescente, o último elemento de `divisors(n)` será sempre o próprio n. O comando `divisors(n)[:-1]` retorna a a lista `divisors(n)` sem o último elemento, ou seja apenas os divisores próprios. Há três comportamentos possíveis para a soma dos divisores de um número. Podemos calcular a soma dos elementos em uma lista com a função `sum`, veja:

```
print("Soma dos div. proprios de 10 =", sum(divisors(10)[:-1]))
print("Soma dos div. proprios de 6 =", sum(divisors(6)[:-1]))
print("Soma dos div. proprios de 12 =", sum(divisors(12)[:-1]))
```

```
Soma dos div. proprios de 10 = 8
Soma dos div. proprios de 6 = 6
Soma dos div. proprios de 12 = 16
```

Um número n cuja soma dos divisores próprios é

- menor que n, como o 10, é chamado de *número deficiente*,
- maior que n, como o 12, é chamado de *número abundante*, e
- igual a n, como o 6, é chamado de *número perfeito*.

Números perfeitos tem sido estudados desde os gregos. No sétimo volume de seus Elementos, Euclides provou que que todos os números de uma determinada forma eram perfeitos. Muitos séculos depois, Euler provou que todos os números perfeitos pares são da forma descrita por Euclides. Antes de apresentarmos essa fórmula, use a soma dos conjuntos dos divisores próprios para resolver o exercício a seguir, em seguida tente encontrar algum padrão. Pode ajudar se você fatorar os números perfeitos que encontrar.

Exercício 1.7. Construa um código que encontre os 10 primeiros números perfeitos.

O teorema a seguir classifica todos os números perfeitos pares.

Teorema 1.4 (Euclides/Euler)**.** *Seja n um número perfeito par, então tem n é da forma $n = 2^e(2^{e+1} - 1)$, onde $e \in \mathbb{N}$ com $2^{e+1} - 1$ primo. Reciprocamente, todo número perfeito par é da forma descrita.*

De posse desse teorema, você pode encontrar números perfeitos pares encontrando primos da forma $2^k - 1$. Primos dessa forma são chamados de *primos de Mersenne*, comentaremos mais sobre ele no Capítulo **??**.

Exercício 1.8. Construa um código que encontre os primeiros 100 números perfeitos pares.

Exercício 1.9. Verifique numericamente que não existe nenhum numero perfeito ímpar menor que 10 000.

Exercício 1.10. *Números amigos* são dois naturais a e b tais que a soma dos divisores próprios de a é igual a b e a soma dos divisores próprios de b é igual a a. Por exemplo $(a, b) = (220, 284)$ é um par de números amigos pois os divisores próprios de 220 somam 284 e os de 284 somam 220.

a) Encontre outro par de números amigos.

b) Pesquise sobre a regra de Thābit ibn Qurra e liste outros pares de números amigos.

1.7.3 Conjectura de Collatz

Defina a função $f : \mathbb{N} \to \mathbb{N}$ da seguinte forma

$$f(n) = \begin{cases} n/2 & \text{se } n \text{ é par} \\ 3n+1 & \text{se } n \text{ é ímpar} \end{cases}$$

Por exemplo $f(10) = 5$, $f(5) = 16$, $f(16) = 8$, $f(8) = 4$, $f(4) = 2$, $f(2) = 1$. Podemos definir a função f no *sagemath* da seguinte forma

```
def f(n):
  if n % 2 == 0:
    return n/2
  else:
    return 3*n+1
print(f(10))
```

```
5
```

Note que o exemplo que fornecemos, ao iniciarmos com $n = 10$ e aplicarmos f sucessivamente chegamos no número 1. A *conjectura de Collatz* afirma que, independente do $k \in \mathbb{N}$ inicial, a sequência $f^i(k)$, $i = 1, 2, 3, \ldots$ eventualmente assume o valor 1. Como o próprio nome diz, a conjectura de Collatz ainda não foi demonstrada, no entanto ela já foi verificada para valores iniciais k até 2^{28}. Nos exercícios a seguir você construirá ferramentas no *sagemath* que contribuem para a crença que a conjectura é verdadeira.

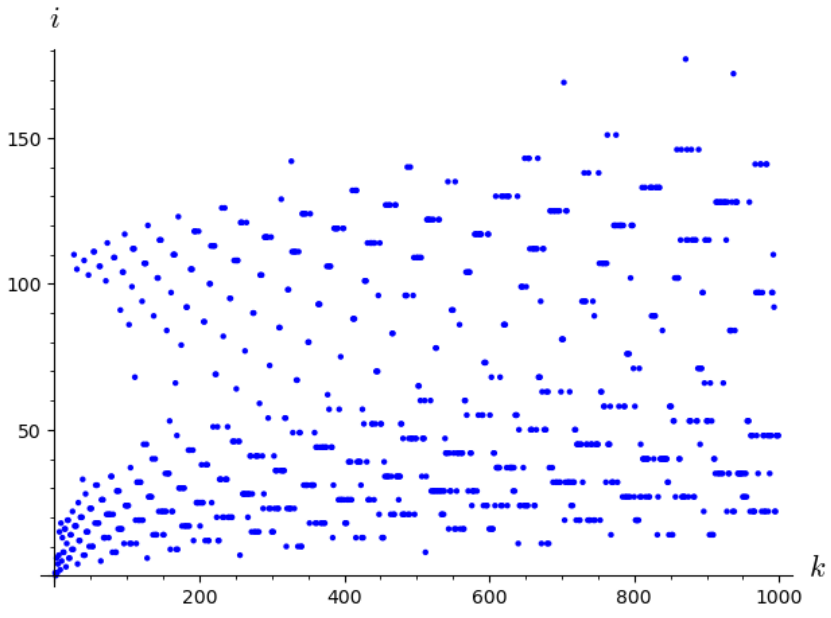
Exercício 1.11. Escreva um código em *sagemath* que, para dado $k \in \mathbb{N}$, encontre o menor inteiro i tal que $f^i(k) = 1$. Você pode usar a função f que definimos acima. Crie uma função chamada `collatz` que retorna, para cada k, o i acima, isto é, `collatz(k)` deve retornar $\min\{i \in \mathbb{N} \mid f^i(k) = 1\}$

Exercício 1.12. Crie uma lista com os 1000 primeiros valores do tempo de parada de k. Se você criou a função `collatz` no exercício acima você pode ter uma bela visualização da distribuição dos tempos de parada, como na Figura 1.1, usando o comando a seguir

```
list_plot([(k,collatz(k)) for k in [1..1000]])
```

Mesmo que você tenha feito tudo certo, não há garantia que a sua função `collatz` vai eventualmente chegar em um resultado i. Se a conjectura de Collatz for falsa e existir um contra exemplo, isto é, algum $k_0 \in \mathbb{N}$ tal que $f^i(k_0) \neq 1$ para todo $i \in \mathbb{N}$, então a sua função será executada por toda a eternidade. Então se você tentou calcular `collatz(k)` e o seu programa não retornou resultado alguma das seguintes opções é verdadeira: 1) você cometeu algum erro no código (esperamos que não), 2) o seu k é muito grande e o seu computador ainda está no caminho (o mais provável), ou 3) parabéns, você encontrou um contra exemplo para a conjectura de Collatz.

Mesmo que a conjectura de Collatz já tenha sido verificada para números muito muito grandes, não há garantia que ela seja válida. Um exemplo clássico disso é uma conjectura de Polya, feita em 1919, sobre a paridade da quantidade de divisores



mg:collatz}

Figura 1.1: Tempo de parada $f^i(k) = 1$

até certo número. Embora a afirmação pareça ser válida ao análisar os primeiros naturais, um contra exemplo foi encontrado na década de 60. Hoje sabemos que o menor contra exemplo para conjectura de Polya é o número 906 150 257. Assim, ainda que programas de computador forneçam indícios de que uma afirmação seja verdade, uma demonstração é essencial para realmente confirmar sua validade. Por outro lado, se suspeitarmos que uma afirmação é falsa, um contra exemplo encontrado computacionalmente já garante que afirmação não é verdadeira.

Exercício 1.13. Pesquise sobre a conjectura de Polya e verifique que ela é válida para os primeiros 1000 naturais.

1.7.4 Primos Gêmeos

Vejamos uma lista com os 100 primeiros primos

```
1 print(primes_first_n(100))
```

```
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59,
   61, 67, 71, 73, 79, 83, 89, 97, 101, 103, 107, 109, 113, 127,
   131, 137, 139, 149, 151, 157, 163, 167, 173, 179, 181, 191,
   193, 197, 199, 211, 223, 227, 229, 233, 239, 241, 251, 257,
   263, 269, 271, 277, 281, 283, 293, 307, 311, 313, 317, 331,
   337, 347, 349, 353, 359, 367, 373, 379, 383, 389, 397, 401,
   409, 419, 421, 431, 433, 439, 443, 449, 457, 461, 463, 467,
   479, 487, 491, 499, 503, 509, 521, 523, 541]
```

Analisando atentamente notamos que os primos parecem se distanciar uns dos outros a medida que crescem, mas, vez ou outra, existem pares de primos muito próximos, por exemplo: 107 e 109, 239 e 241, 419 e 421, etc. Primos $p < q$ são ditos

gêmeos se $q = p + 2$. Faça os exercícios a seguir e crie sua própria conjectura sobre primos gêmeos.

Exercício 1.14. Crie um código que conte quantos pares de primos gêmeos existem até 1000. E áté 10 000? E 100 000?

Discutiremos mais sobre primos especiais e a distribuição dos números primos no Capítulo **??**.

1.7.5 Sequências Recorrentes e Expressões Simbólicas

{par:fib}

A sequência de Fibonacci (u_n) é definida da seguinte forma: $u_0 = u_1 = 1$ e $u_n = u_{n-1} + u_{n-2}$ para $n \geq 2$. Usando a recursividade podemos calcular o valor de u_n de forma simples

```
1 def fib(n):
2   if n == 0 or n == 1:
3     return 1
4   return fib(n-1) + fib(n-2)
5 [fib(n) for n in [0..10]]
```

```
[1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
```

De forma mais geral, uma sequência recorrente linear de ordem k é uma sequência (x_n) tais que existem constantes $c_1, \cdots, c_k$ tais que

$$x_{n+k} = \sum_{j=1}^{k} c_j x_{n+k-j}$$

para $n \in \mathbb{N}$. Tais sequências dependem dos seus k primeiros termos.

Exercício 1.15. Crie um código sage que calcule o valor de uma sequência recorrente linear, como a (x_n) acima, dados os seus k primeiros termos e as constantes c_j, $1 \leq j \leq k$.

Apesar de apresentarmos a sequência de Fibonacci de forma recursiva, existe uma forma geral para os termos u_n:

$$u_n = \frac{\varphi^n - \psi^n}{\varphi - \psi}, \quad \text{onde } \varphi = \frac{1+\sqrt{5}}{2} \text{ e } \psi = \frac{1-\sqrt{5}}{2}$$

A existência de uma fórmula desse tipo vale para sequências recorrentes lineares mais gerais, recomendamos a leitura de [5, Apêndice B], especialmente a Seção B.4 (no caso da fórmula acima para sequência de Fibonacci, você pode demonstrar a fórmula por indução). O código a seguir calcula o n-ésimo elemento na sequência de Fibonacci usando essa fórmula geral.

```
1 def fib_g(n):
2     phi = (1+sqrt(5))/2
3     psi = (1-sqrt(5))/2
4     return (phi^n - psi^n)/(phi-psi)
```

```
5 print(fib_g(5))
```

O resultado deveria ser `8`, certo? Mas...

```
1/320*sqrt(5)*((sqrt(5) + 1)^6 - (sqrt(5) - 1)^6)
```

Se você estiver usando o *sagemath* em algum notebook (no cocalc, SageMathCell ou jupyter), pode pedir para o *sagemath* mostrar o resultado usando a função `show` ao inves do `print` [2], será exibida a expressão

$$\frac{1}{320}\sqrt{5}\left(\left(\sqrt{5}+1\right)^6-\left(\sqrt{5}-1\right)^6\right)$$

O leitor mais corajoso pode verificar que essa expressão realmente é igual a 8. No *sagemath*, podemos obter esse resultando usando a função `expand`:

```
1 expand(fib_g(6))
```

```
8
```

O resultado *inesperado* é na verdade uma das vantagens do *sagemath*. Em uma linguagem de programação usual, como no próprio Python, ao definir $\varphi = \frac{1+\sqrt{5}}{2}$, o valor seria convertido em um número real — na verdade, no ponto flutuante que aproxima φ, induzindo um erro de aproximação — depois disso as contas são feitas com essas aproximações. **O código abaixo está em linguagem Python, e não *sagemath*** (por isso foi necessário importar a função `sqrt`, no *sagemath* isso não é necessário).

```
1 >>> from math import sqrt
2 >>> phi = (1+sqrt(5))/2
3 >>> print(phi)
4 >>> print(type(phi))
```

```
1.618033988749895
<class 'float'>
```

No *sagemath*, por outro lado, o cálculo é feito internamente de forma simbólica. Vejamos o tipo de dado que é φ.

```
1 phi = (1+sqrt(5))/2
2 parent(phi)
```

```
Symbolic Ring
```

Isso significa que, da forma como foi definido, `phi` é uma *expressão simbólica*. Você pode visualizar o valor real (aproximado) de φ com `RR(phi)`, mas há muitas vantagens em se usar expressões simbólicas, mesmo quando estamos trabalhando apenas com números. Não usaremos a computação simbólica de forma mais aprofundada nesse texto, mas a ideia é que os cálculos são feitos de forma mais parecida com o que nós (humanos) fazemos (afinal, ao fazer uma conta com o φ você substituiria

[2] No interpretador *sagemath* essa função irá retornar o código LaTeX da visualização exibida.

seu valor por $\approx 1.61803398874989...$ e usaria esse número?). Os exercícios a seguir o guiarão por algumas das possibilidades do cálculo simbólico [3].

Exercício 1.16. Não apenas números podem ser tratados simbolicamente pelo *sagemath* mas, principalmente, variáveis. Por padrão a única variável (no sentido computacional) que o *sagemath* assume ser uma variável (no sentido matemático) é o `x`, mas você pode definir outras variáveis.

a) Crie uma variável y com a função `var('y')` e verifique o seu tipo usando a função `parent`, como fizemos acima.

b) Defina $f = x^3 - y^3$ e use a função `factor` para encontrar a fatoração de f (sim, é o mesmo `factor` que fatora um número em primos!)

c) É possível substituir valores em uma expressão simbólica. Tente `f.subs(x==2,y==1)`.

d) `derivative(f,y)` deve ser autoexplicativo. Mude a expressão para f e calcule sua derivada. Usando o método `.subs()` é possível calcular a derivada em um ponto específico. O que ocorre se tentarmos calcular a derivada em um ponto onde a função não é diferenciável? (Dica: no sage $|x|$ pode ser escrito como `abs(x)`)

d Desafio: Construa um código que calcule (simbolicamente) a derivada de uma função sem usar a função `derivative`. (Dica: Crie uma nova variável `h`, se lembre da definição de derivada e use a função `limit`).

Exercício 1.17. A função `bool()` verifica se o argumento passado é verdadeiro ou falso. Tente explicar o que está acontecendo no código a seguir:

```
1 print(bool(sqrt(x^2) == x))
2 assume(x>0)
3 print(bool(sqrt(x^2) == x))
```

```
False
True
```

Exercício 1.18. Uma das mais importantes aplicações do cálculo simbólico é a resolução de equações.

a) Interprete os resultados de `solve(x^5-1,x)` e `solve(x^7+x^2-2*x+2,x)`.

Vamos encontrar a interseção entre o círculo unitário $x^2 + y^2 = 1$ e a reta $x + 2y = 1$.

```
1 eqs = [x^2 + y^2 == 1, x+2*y == 1]
2 solve(eqs,x,y)
```

```
[[x == 1, y == 0], [x == (-3/5), y == (4/5)]]
```

[3] Algumas das manipulações que faremos com expressões simbólicas, quando lidam com expressões polinomiais, são feitas de forma mais eficiente no *sagemath* usando outro tipo de objeto, elementos de anéis de polinômios.

b) Encontre os pontos na interseção entre as curvas $y^2 = x^3 + x$ e $y = 3x$.

c) Você pode verificar se o resultado está correto o substituindo nas equações...

d) ... ou pode *verificar* visualmente exibindo as curvas e os pontos envolvidos. Por exemplo, o ponto $(0,0)$ é um ponto na interseção.

```
var('y')
c1 = implicit_plot(y^2 == x^3 +
   x,(-1,1),(-1,1),color='green')
c2 = implicit_plot(y == 3*x,(-1,1),(-1,1),color='red')
p = point((0,0),size=40,color='black')
show(c1+c2+p)
```

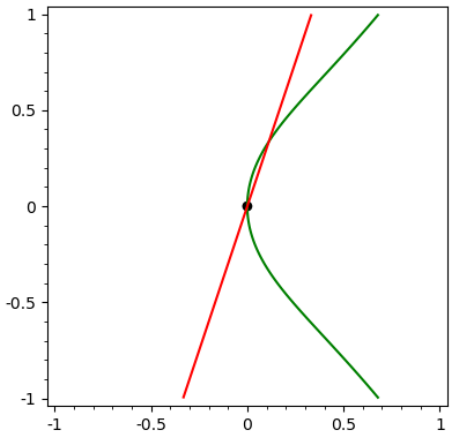


Figura 1.2: Interseção entre $y^2 = x^3 + x$ e $y = 3x$ {img:cepoin

O resultado está na Figura 1.2. Os `(-1,1)` representam os intervalos de x e y no gráfico. Assim o código `implicit_plot(eq,(a,b),(c,d))` vai exibir o gráfico da equação implícita nas variáveis x e y dada em `eq` no retângulo $[a,b] \times [c,d]$, isto é, $a \leq x \leq b$ e $c \leq y \leq d$. Para uma das soluções do item d) você vai precisar alterar o intervalo para conseguir visualizar o ponto.

Exercício 1.19. Use a técnica apresentada em [5, B.4] para encontrar a fórmula geral da recorrência $x_n = 4x_{n-1} + 3x_{n-2} - 14x_{n-3} - 6x_{n-4}$ com termos iniciais $x_0 = x_1 = 4, x_2 = 22$ e $x_3 = 58$.

Apesar de não entrarmos em muitos detalhes, o cálculo simbólico é uma das ferramentas mais úteis do *sagemath*. A documentação oficial possui uma extensa lista de aplicações, que incluem resolução de equações, cálculos envolvendo derivadas, integrais, séries, etc. Veja [1, Symbolic Calculus].

1.8 Exercícios

Exercício 1.20. Crie um código *sagemath* que funcione como a função `divisors`. Tente otimizar o seu código (Veja o item *b)* do Exercício 1.23).

{ex:fatoria

Exercício 1.21. Use a recursividade para criar uma função que calcule o fatorial de um número. Siga as instruções:

a) Defina uma função meu_fatorial com a com assinatura `def meu_fatorial(n):`

b) Teste se $n = 0$, nesse caso a função deve retornar o valor 1.

c) Se $n > 1$, a função deve retornar $n \times (n-1)!$, isto é, `n*meu_fatorial(n-1)`.

Exercício 1.22. Use a recursividade para calcular o mdc entre dois inteiros não ambos nulos. Use o Lema 1.1.

acaoprimos}

Exercício 1.23. Reescreva um algoritmo que encontre a fatoração de um natural $n > 1$ com as otimizações discutidas. Atente para os itens a seguir.

a) Remova o uso da recursividade, dessa forma você pode criar uma lista `fatores` dentro da própria função e armazenar os fatores primos encontrados.

b) Prove que se p é o menor primo dividindo n, então $p \leq \sqrt{n}$. Use esse fato para limitar superiormente a busca pelos fatores primos de n.

c) Prove que se p é o menor primo dividindo n, então nenhum primo menor que p divide n/p. Use esse fato para limitar inferiormente a busca pelos fatores primos de n.

Exercício 1.24. XGCD de forma mais direta

Exercício 1.25. O algoritmo de Euclides estendido pode ser descrito na forma matricial. Iniciamos com a matriz

$$M = \begin{pmatrix} x_0 & x_1 \\ y_0 & y_1 \\ r_0 & r_1 \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \\ a & b \end{pmatrix}.$$

Agora, enquanto $r_1 \neq 0$, tome $q = \lfloor r_0/r_1 \rfloor$, e alteramos a matriz M da seguinte forma:

$$M \longleftarrow M \begin{pmatrix} 0 & 1 \\ 1 & -q \end{pmatrix}.$$

Verifique que, ao final do processo teremos que $r_0 = \text{mdc}(a, b)$ e $x_0 a + y_0 b = \text{mdc}(a, b)$. Pesquise sobre a manipulação de matrizes no *sagemath* e implemente o algoritmo de Euclides estendido da forma descrita acima. (Note que não é preciso definir variáveis auxiliares $x_0, x_1, y_0, \ldots$, todas as informações estão contidas na matriz M e em q.)

Exercício 1.26. Dada uma fração contínua $[a_0; a_1, a_2, a_3, \ldots]$, sua n-ésima convergente p_n/q_n pode ser calculada através da relação de recorrência

$$\begin{cases} p_n = a_n p_{n-1} + p_{n-2} \\ q_n = a_n q_{n-1} + q_{n-2} \end{cases}$$

a) As relações acima são conhecidas como fórmulas de Wallis. Dê uma demonstração usando indução.

b) Crie um código que calcule a n-ésima convergente usando as fórmulas de Wallis. Compare o resultado do seu código com o obtido pelo método `.convergent`. (Dica: você pode tratar uma fração contínua como uma lista para obter seus termos, por exemplo, `continued_fraction(pi)[4]` retorna `292`.)

Exercício 1.27. Um bom aluno de álgebra abstrata deve ter notado que existem muitas semelhanças entre o anel dos inteiros $\mathbb{Z}$ e o anel dos polinômios em uma indeterminada com coeficientes em um corpo, por exemplo, $\mathbb{Q}[t]$. Por exemplo, é possível efetuar a divisão euclidiana de um polinômio por outro (não nulo), com condições análogas de unicidade, você também pode calcular o mdc entre polinômios, fatorá-los, etc. O *sagemath* consegue manipular polinômios tão bem quanto manipula os inteiros. Para trabalhar com os polinômios precisamos primeiramente definir o ambiente em que eles estão (já que, como os números, um polinômio com a mesma expressão pode ser considerado como elemento de vários aneis distintos), sobre os racionais, por exemplo, podemos definir $\mathbb{Q}[t]$ com o comando `P.<t> = QQ[]`. Ao mesmo tempo que estamos definindo `P` $= \mathbb{Q}[t]$, também estamos indicando ao *sagemath* que, a partir de agora, `t` deve ser considerado como a indeterminada do anel $\mathbb{Q}[t]$. Na célula a seguir definimos o anel $\mathbb{Q}[t]$, e exploramos o polinômio $f(t) = t^6 - 1 \in \mathbb{Q}[t]$. Não exibimos o resultado.

```
P.<t> = QQ[t]
f = P(t^6 - 1)
print(f)
print(f(5))
print(f.quo_rem(P(t^2 - 1)))
print(f.quo_rem(P(t^4+2*t)))
print(f.is_irreducible())
print(factor(f))
f.roots()
f.roots(QQbar)
```

a) Execute as funções acima e interprete o resultado.

b) Pesquise por mais funções disponíveis para polinômios.

c) Teste a aritmética para polinômios sobre outros anéis.

Exercício 1.28. Pesquise sobre a função `timeit` ou o comando mágico `%timeit`. COMPARAR OS TEMPOS DE EXECUÇÃO

Índice Remissivo

Funções e Métodos *sagemath*

Referências Bibliográficas

[1] The Sage Developers, *SageMath, the Sage Mathematics Software System (Version 9.0)*, 2021. `https://www.sagemath.org`.

[2] L. Silva, M. Santos, and R. Machado, *Elementos de Computação Matemática com SageMath*. Rio de Janeiro: Textos Universitários SBM, 2019.

[3] P. Zimmermann, A. Casamayou, N. Cohen, G. Connan, T. Dumont, L. Fousse, F. Maltey, M. Meulien, M. Mezzarobba, C. Pernet, *et al.*, *Computational Mathematics with SageMath*. Other Titles in Applied Mathematics, Society for Industrial and Applied Mathematics, 2018.

[4] G. Bard, *Sage for Undergraduates*. American Mathematical Society, 2015.

[5] F. E. B. Martinez, C. G. T. d. A. Moreira, N. C. Saldanha, and E. Tengan, *Teoria dos Números: um passeio com primos e outros números*. Rio de Janeiro: Impa, 2010.

[6] E. de Andrade, C. Bracciali, and S. B. de Matemática Aplicada e Computacional, *Frações contínuas: propriedades e aplicações*. Notas em Matemática Aplicada, SBMAC, 2005.