

TRABALHANDO COM NÚMEROS DE PONTO FLUTUANTE NO PHP

Matéria da Alura no link <https://www.alura.com.br/artigos/php-operacoes-com-numeros-decimais>

Operações com **ponto flutuante** podem ser uma grande dor de cabeça em diversas linguagens, e não é diferente no PHP. Um simples $0.1 + 0.2$ pode dar uma resposta inesperada. **Como ter uma precisão que queremos?** Recentemente estive trabalhando em um programa para que eu possa gerenciar o dinheiro que entra e sai da minha conta bancária, guardando informações sobre cada transação de débito e crédito.

Para armazenarmos estas informações, vamos iniciar criando um arquivo **PHP** com um array que armazenará o débito e o crédito:

```
<?php
```

```
$debitos = array();
```

```
$creditos = array();
```

```
?>
```

Agora, como realizei um pagamento de R\$0,10, adicionamos a seguinte linha no programa, para adicionar o valor no array de débitos:

```
array_push($debitos, 0.1);
```

E mais tarde, realizei outro pagamento de R\$0,20, então adicionamos também mais uma linha:

```
array_push($debitos, 0.2);
```

No final do dia eu recebi um pagamento de R\$0,30, então adicionei um valor no **array** de créditos do programa:

```
array_push($creditos, 0.3);
```

Por enquanto, o nosso código ficou assim:

```
<?php
```

```
$debitos = array();
```

```
$creditos = array();
```

```
array_push($debitos, 0.1);
```

```
array_push($debitos, 0.2);
```

```
array_push($creditos, 0.3);
```

```
?>
```

Agora, preciso consultar o meu saldo e exibí-lo na tela, vamos criar uma função para calcular isso, adicionamos no final do programa:

```
function saldo(array $debitos, array $creditos) {
```

```
    return array_sum($creditos) - array_sum($debitos);
```

```
}
```

Para consultarmos o saldo, vamos imprimir na tela o resultado da função. Chamamos a função e passamos os arrays de débito e crédito como argumento:

```
echo saldo($debitos, $creditos);
```

O código completo do programa ficou:

```
<?php
```

```
$debitos = array();
```

```
$creditos = array();
```

```
array_push($debitos, 0.1);
```

```
array_push($debitos, 0.2);
```

```
array_push($creditos, 0.3);
```

```
function saldo(array $creditos, array $debitos) {
```

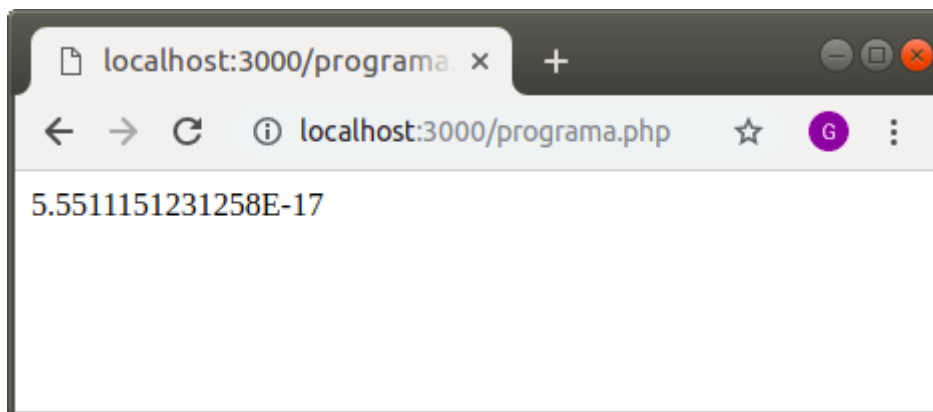
```
    return array_sum($creditos) - array_sum($debitos);
```

```
}
```

```
echo saldo($creditos, $debitos);
```

```
?>
```

E se executarmos o programa, tanto pelo terminal quanto pela web, vamos ver que a saída do programa ficou:



O que aconteceu? Esperávamos que o resultado fosse zero, mas não foi isso que tivemos como resultado.

Realizando operações com Float

Quando tratamos com dinheiro em nossos programas, precisamos do máximo de precisão que pudermos obter, pois erros em operações financeiras podem causar prejuízos. Além disso, temos que tomar cuidado com valores do tipo Float, que são números que possuem um ponto flutuante.

O problema que enfrentamos é que a aritmética que usamos (base 10) é diferente da que o computador utiliza (base 2), e a conversão entre elas pode causar erros de precisão.

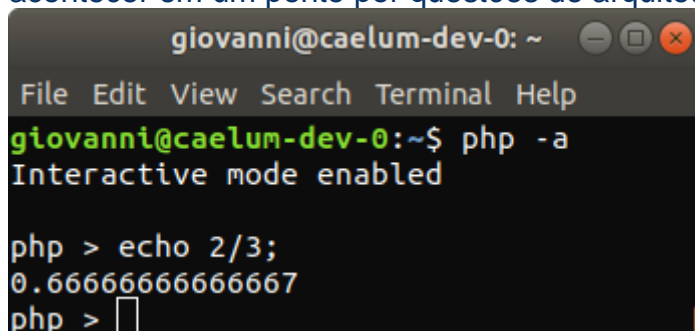
Em números de **base 2**, com um dígito, podemos representar dois números (0 e 1). Com dois dígitos 2^2 podemos representar 4 números (0 a 3), com três dígitos, $2^3 = 8$ (0 a 7) e por assim vai.

Em números de **base 10**, com um dígito, podemos representar 10 números (0 a 9), com dois dígitos podemos representar $10^2 = 100$ números (0 a 99). Com a base 10, representamos mais números com menos dígitos, tendo uma precisão melhor.

Como o computador só entende números binários (base 2), tudo que digitamos nele vira uma sequência de 0 e 1 para que o processador possa interpretar.

Para exemplificar a diferença entre um número em binário e decimal (base 10), vamos pegar por exemplo o número 1025 que em binário, fica 10000000001, precisando de onze dígitos, mas em decimal, seriam utilizados apenas quatro dígitos (1, 0, 2 e 5).

Normalmente não percebemos este erro por acabarmos não utilizando toda esta precisão para representar números mais simples. Mas quando obtemos algo como uma dízima periódica, ou usamos operadores de comparação, vemos que o arredondamento acontece e causa problemas pois ela é infinita, e a **memória do computador é finita**, então o arredondamento tem que acontecer em um ponto por questões de arquitetura do processador.

A screenshot of a terminal window titled 'giovanni@caelum-dev-0: ~'. The window has a menu bar with 'File', 'Edit', 'View', 'Search', 'Terminal', and 'Help'. The prompt is 'giovanni@caelum-dev-0:~\$'. The user enters 'php -a', and the prompt changes to 'Interactive mode enabled'. Then, the user enters 'php > echo 2/3;', and the output is '0.666666666666667'. The prompt returns to 'php >' with a cursor.

```
giovanni@caelum-dev-0: ~  
File Edit View Search Terminal Help  
giovanni@caelum-dev-0:~$ php -a  
Interactive mode enabled  
  
php > echo 2/3;  
0.666666666666667  
php > 
```

E, no caso de dinheiro, precisamos desta precisão que números base 10 oferecem, então float não é o tipo de dado ideal para esta situação por conta de sua precisão. Uma das maneiras que podemos contornar isso é usar ao invés do float, um tipo de dados que não tem arredondamento como o tipo Int.

Utilizando valores inteiros

Podemos usar o menor valor de uma moeda (no caso do real, utilizamos o centavo, por ele não poder ser dividido) e armazenamos ele no programa como **tipo Int**, e fazemos todas as operações nessa unidade, pois o **tipo nt** não tem arredondamento que nem o **Float**, portanto não estará sujeito a perda de precisão devido ao arredondamento.

Utilizando o programa que construímos como exemplo, o código dele ficaria assim:

```
<?php
```

```
$debitos = array();
```

```
$creditos = array();
```

```
array_push($debitos, 10); #Trocamos o valor 0.1 para 10,
```

```
representando os centavos.
```

```
array_push($debitos, 20); #Fazemos a mesma coisa, trocando 0.2
```

```
para 20.
```

```
array_push($creditos, 30); #Trocamos o valor 0.3 para 30.
```

```
function saldo(array $creditos, array $debitos) {
```

```
    return array_sum($creditos) - array_sum($debitos);
```

```
}
```

```
$saldo = saldo($creditos, $debitos);
```

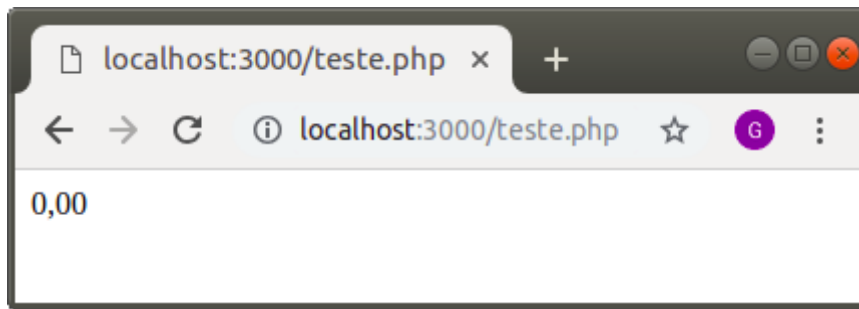
```
echo number_format($saldo, 2, ',', '.');
```

```
?>
```

Mas como estamos tratando como centavos, assim que quisermos representar ele como real, tratamos somente na saída.

```
number_format($saldo, 2, ',', '.');
```

Podemos utilizar a função **number_format** do PHP para exibir os centavos armazenados internamente de uma maneira correta na tela.



Perfeito! Assim não temos mais problemas com a precisão, podemos utilizar números inteiros para tudo então?

Um problema que ocorre com esta abordagem é que o `Int` possui um valor máximo, no caso do PHP, o número máximo dos valores inteiros vai depender da arquitetura para qual o PHP foi compilado.

Se for **32-bit**, o valor máximo é de R\$ 21.474.836,48 (mais ou menos 21 milhões de reais) e se for **64-bit**, o valor máximo é de R\$ 92.233.720.368.547.758,07 (92 quatrilhões de reais). Vale lembrar que o PHP não suporta tipos inteiros unsigned (onde só existem números positivos), ou seja, esse máximo vale tanto para um valor positivo quanto negativo. Você pode descobrir qual o limite na sua instalação do PHP imprimindo a constante **PHP_INT_MAX**.

```
giovanni@caelum-dev-0: ~/Code
File Edit View Search Terminal Help
php > echo PHP_INT_MAX;
9223372036854775807
php > 
```

Esta solução é boa se você sabe que não vai esbarrar no limite máximo do inteiro, pois assim que este limite for excedido, no PHP, o valor começa a ser tratado como `Float`.

No caso de dinheiro, não podemos correr o risco deste valor virar um `Float` e perdermos precisão. Então podemos explorar mais uma solução, usar a extensão `Decimal`.

Fazendo operações com a extensão `Decimal`

Em uma empresa que possui um programa que gerencia o movimento de fluxos de caixa com valores enormes, pode acontecer de precisarmos armazenar um valor maior do que **tipos inteiros** oferecem na nossa arquitetura. A extensão **Decimal** é uma extensão para o PHP 7 que provê um objeto do tipo decimal para a linguagem, que é a maneira recomendada de lidarmos com este tipo de situação.

Para instalá-la, precisaremos utilizar o **PECL**, que é o gerenciador de pacotes que vem com o PHP, diferente do **Composer**, que é responsável por baixar **bibliotecas PHP**, o **PECL** é responsável por extensões C feitas para a linguagem em si.

```
pecl install decimal
```

Isso vai compilar o PHP com esta extensão, se tudo der certo, você pode adicionar no php.ini a seguinte linha para ativá-la.

```
extension=decimal
```

Para que esta extensão funcione, ela precisa do [pacote libmpdecimal](#) instalado, ou se estiver usando uma distribuição Linux, execute o seguinte comando:

```
sudo apt-get install libmpdec-dev
```

É interessante lembrar que esta extensão tem uma dependência na extensão de **JSON** do PHP, então ela deve ser adicionada depois desta extensão. Se você estiver em uma **distribuição Linux como o Debian ou Ubuntu**, edite o seguinte arquivo (se ele não existir, crie):

```
/etc/php/7.2/mods-available/decimal.ini
```

Agora, para que o PHP consiga visualizar a extensão Decimal, precisamos colocar nesse arquivo que criamos, o nome da extensão e a prioridade:

```
; priority=30
```

```
extension=decimal
```

Agora podemos utilizar o **tipo Decimal**! Ele está dentro do namespace com o nome de Decimal. Então para utilizá-lo, precisaremos adicionar a seguinte linha no começo do programa:

```
use Decimal\Decimal;
```

Então, podemos instanciar a classe Decimal com uma string contendo o valor que queremos representar, por exemplo:

```
$decimal = new Decimal("0.1");
```

Utilizando o nosso primeiro exemplo e adaptando ele para utilizar o tipo Decimal, ficaria assim:

```
<?php
```

```
use Decimal\Decimal;
```

```
$debitos = array();
```

```
$creditos = array();
```

```
array_push($debitos, new Decimal("0.1"));
```

```
array_push($debitos, new Decimal("0.2"));
```

```
array_push($creditos, new Decimal("0.3"));
```

```
function saldo(array $creditos, array $debitos) {
```

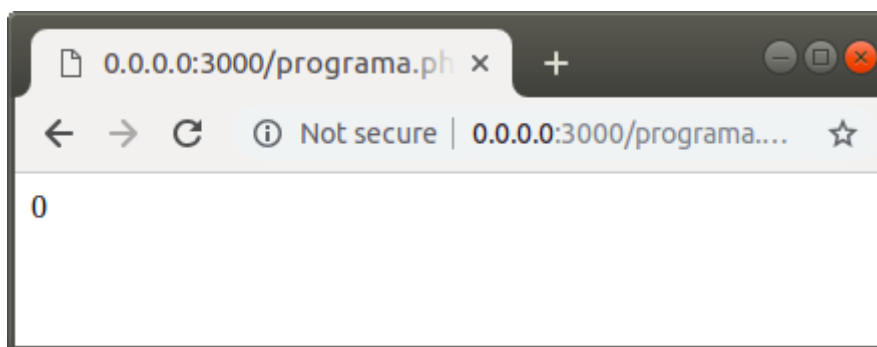
```
    return array_sum($creditos) - array_sum($debitos);
```

```
}
```

```
echo saldo($creditos, $debitos);
```

```
?>
```

E se executarmos, podemos ver que a saída será exibida corretamente:



Se desejar saber mais sobre números de precisão, recomendo o website [Floating Point Guide](#) (em inglês), que mantém um guia sobre tudo que um programador precisa saber sobre precisão numérica. Outro site interessante para quem quiser saber mais sobre como essas coisas funcionam no PHP especificamente é a documentação oficial do PHP na parte de [floats](#).

Vimos que, para obtermos uma maior precisão numérica, o Float não é um tipo de dado recomendado por ele arredondar valores.

Vimos também que o tipo **Int** não possui arredondamento, e que ele é uma solução preferível ao **Float**, mas ele possui limitações em seu tamanho máximo.

Além disso, sabemos agora que o tipo Decimal, instalável através de uma extensão, é o tipo recomendado para tratar com valores de uma maneira mais precisa.

Você gostou do que leu? Aqui na **Alura** temos uma [formação PHP](#), você vai aprender desde os fundamentos da linguagem junto com as boas práticas e o essencial conhecimento de uma boa modelagem orientada a objetos.!