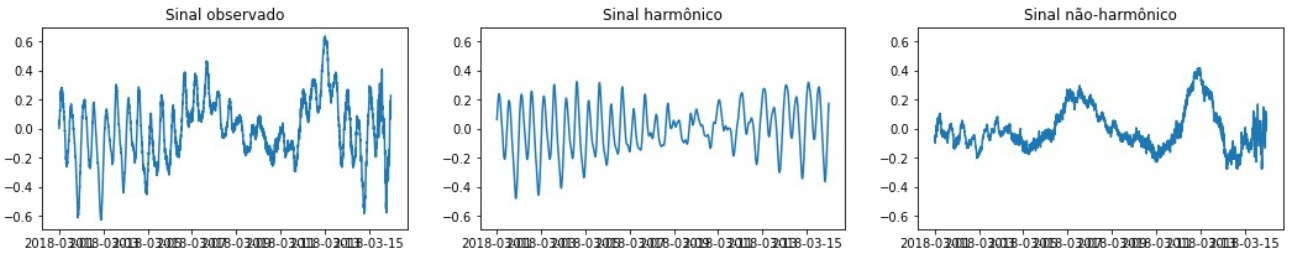
Analisando dados com Python

Investigando o sinal do nível do mar.

Parte 3

V1. 2021.08.23



Carlos A.F. Schettini

Laboratório de Oceanografia Costeira e Estuarina

LOCostE / IO / FURG

Objetivos:

1 Transferir dados entre notebooks

2 Reamostragem temporal & sincronização

3 Decomposição do sinal do nível do mar

3.a Análise Harmônica

3.b Filtragem com filtro Butterworth

Parte 3 – Investigando o sinal do nível do mar

Na primeira parte do tutorial de marés vimos como obter dados de nível da água da Rede Maregráfica Permanente para Geodésia (RMPG) do Instituto Brasileiro de Geografia e Estatística (IBGE), carregar no Juyter, manipular o conteúdo e gerar um gráfico (Figura 1).

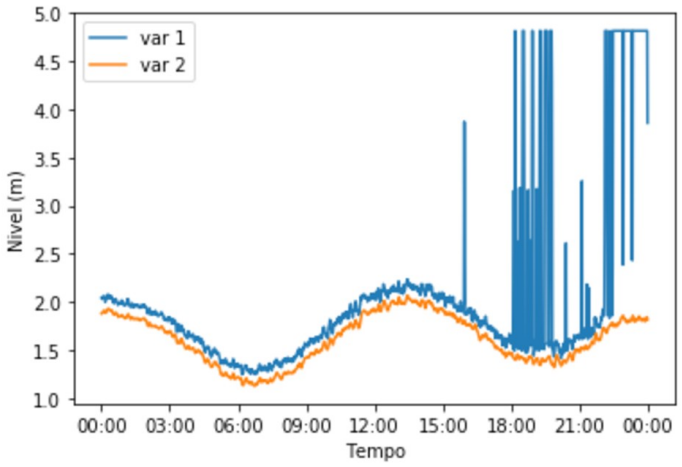


Figura1: Dados de maré do IBGE/RMPG a partir de um único arquivo.

Na segunda parte vimos como carregar múltiplos de diferentes estações, organizando os dados e gerando uma figura com as várias séries temporais (Figura 2). A geração desta figura compreende o primeiro passo da 'análise de dados', que é, basicamente, ver os dados! Ou, em um termo mais elegante, a 'inspeção visual' dos dados. Tipo, ver se os dados estão com cara de dados. A interpretação requer experiência, em especial de já ter visto outras séries de dados com padrões 'suspeitos'. Por exemplo, a Figura 3 apresenta uma série temporal de pressão registrada por um perfilador acústico de correntes por efeito Doppler (ADCP, do acrônimo em inglês) fundeado no estuário do Rio Capibaribe (PE). Notem que no dia 20/setembro há uma mudança do sinal, para menor pressão. Analisando outros parâmetros registrados (heading, pitch e roll), sabemos que houve uma perturbação no instrumento, e o que vemos é que ele foi empurrado para um local um pouco mais raso.

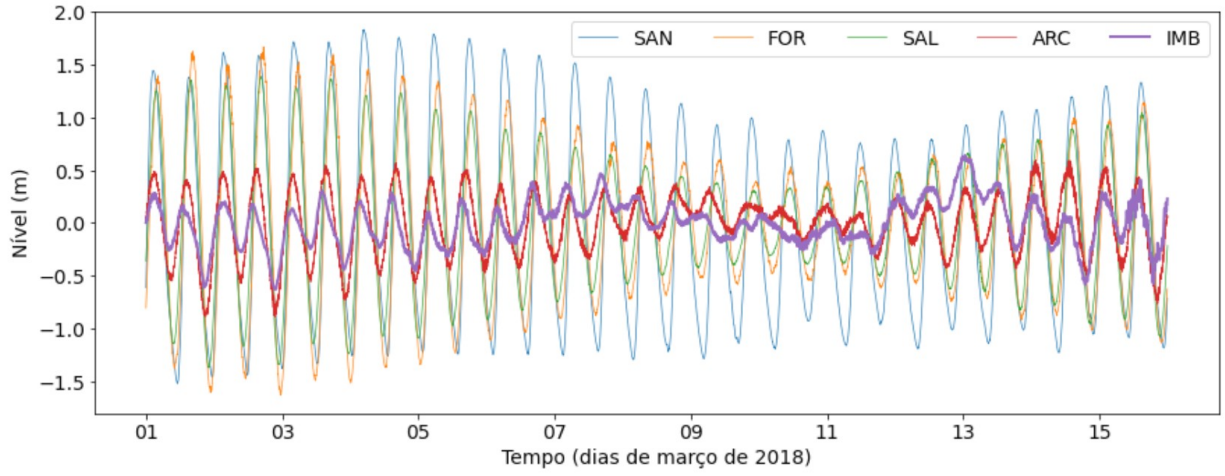


Figura 2: Séries temporais de nível da água de diferentes estações maregráficas do IBGE.

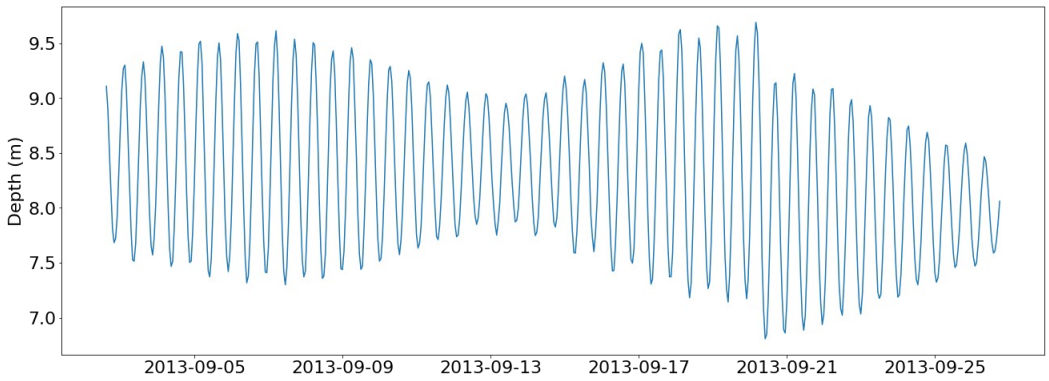


Figura 3: Série temporal de dados de pressão a partir de um ADCP fundeado no estuário do Rio Capibaribe.

Agora vamos analisar os sinais do nível do mar mais profundamente, utilizando duas ferramentas importantes: análise harmônica de marés e filtragem numérica usando o filtro Butterworth. Com isso poderemos explorar as características dos sinais, e quantificar onde há mais ou menos energia (variância). Isso é matemática, mas, toda o efeito deve ter uma causa. O nível do mar não oscila se não houver alguma força associada. E aí entra a física, onde buscamos em quais os agentes físicos possíveis (atração gravitacional, vento, seiches, tsunamis, ondas de vento) podem explicar os resultados.

O que vamos fazer aqui poderia ser continuado no notebook anterior. Mas, para evitar confusão, começaremos um novo notebook. Ter diversos notebooks durante o processamento de dados inevitavelmente gera confusão. Mas um notebook muito longo e com muitas variáveis também gera confusão. Aí vai da capacidade de organizar de cada um, se fazer um ou vários. No

caso de fazer vários, importantíssimo é fazer um cabeçalho explicando o que o notebook faz e se há alguma dependência de outros notebooks.

No notebook anterior foi feito o carregamento dos dados de diversos arquivos de diversas estações e organizados em um dicionário que demos o nome de 'base_dados'. Quando o kernel do notebook está ativo e rodamos o notebook, o dicionário está na memória RAM do computador alocada para este notebook. Para que o conteúdo deste dicionário seja acessível para outro notebook, ou ele (o notebook) compartilha o kernel (https://stackoverflow.com/questions/31621414/share-data-between-ipython-notebooks/60863662), ou devemos fazer uma cópia do conteúdo para a memória sólida do computador (e.g., seu disco rígido), e carregar no segundo notebook. Vamos usar o segundo método, e para isso usaremos o pacote 'pickle'. Ao final do notebook, adicione uma nova célula com o Código 1. Estamos abrindo (criando!) um arquivo 'IBGE_base_dados.pik' e despejando (dump) o conteúdo do dicionário 'base_dados'. Como não especificamos um endereço, o arquivo aparecerá na pasta de trabalho onde foi aberto o notebook.

```
[15]: import pickle

      with open('IBGE_base_dados.pik', 'wb') as handle:
          pickle.dump(base_dados, handle, protocol=pickle.HIGHEST_PROTOCOL)
```

Código 1: Salvando o dicionário 'base_dados' com pickle.

Agora inicializamos outro notebook (Código 2). Colocamos um cabeçalho, importamos o pacote 'pickle' e já vamos deixar essa célula para importar os pacotes que usaremos

posteriormente. Na sequência já abrimos o arquivo ‘Ibge_base_dados.pik’ e transferimos o seu conteúdo para a variável ‘base_dados’. Duas observações: (1) para salvar e carregar arquivos, letras maiúsculas ou minúsculas não fazem diferença! , e (2) a variável ‘base_dados’ agora é uma cópia da variável ‘base_dados’ do outro notebook. Poderia chamar de qualquer outro nome… e o que eu fizer com esta não afetará a outra! Ou seja, se fizer bobagem, podemos carregar de novo. Para apresentar as chaves do dicionário, ou, podemos imaginas as variáveis internas do dicionários que contém os dados, podemos usar o método ‘keys()’.

Maré IBGE - parte 3

Análise do nível do mar

[1]:
```
import pickle
```

Carregando dados pré-processados no notebook 'Mare_IBGE_2.ipynb'

[2]:
```
with open('Ibge_base_dados.pik', 'rb') as handle:

    base_dados = pickle.load(handle)
```

[3]:
```
base_dados.keys()
```

[3]:
```
dict_keys(['IMB', 'ARC', 'SAL', 'FOR', 'SAN'])
```

Código 2: Iniciando outro notebook e carregando o dicionário salvo.

Cada chave do dicionário contém uma lista, e cada elemento da lista é outra lista que contém ano, mês, dia, hora, minuto, segundo, var 1 e var 2… sabemos porque fomos nós que criamos! Se fosse o caso, poderíamos até incluir uma nova chave no dicionário que contivesse uma descrição de como os dados estão organizados. Isso seria muito útil no caso de passar esse material para outros usuários.

Sabemos também que o embora todas as estações compreendam um período de dados igual (15 dias), o intervalo amostral é diferente, o que resulta em listas de tamanhos diferentes. Como vamos analisar comparativamente todas as estações, devemos padronizar todos os dados para que qualquer coisa que seja feita com um, possa ser feita do mesmo jeito com outro, tendo todos o mesmo tamanho e intervalo amostral. Assim, o que faremos para começar é sincronizar todos os dados para um único padrão de referência temporal. Criamos um arranjo de valores de tempo e interpolamos.

O procedimento será feito para cada estação, e como temos 5 estações, a maneira mais elegante de fazer é através de um loop ao invés de repetir tudo 5 vezes. Na verdade, vamos repetir, mas usando um loop. Mas antes de ir direto ao loop, melhor ir por partes (Jack!). Vamos pegar uma das estações e trabalhar com ela. Depois que estivermos satisfeitos com o procedimento, rodamos para as outras.

E depois? Depois que todos os dados estiverem uniformizados e sincronizados com o mesmo intervalo de tempo, podemos partir para análises mais sofisticadas para descrever cada estação e para comparar as estações. Um bom exercício é fazer a decomposição do sinal do nível da água para identificar a quantidade de energia contida nas bandas mareais (entre 10 e 25 horas) e sub-mareais (maior que 25 horas, e o 'sub' é porque falamos em termos de frequência!). E, mesmo na manda mareal podemos separar a componente harmônica e não harmônica… e também aquilo que sobre em oscilações de alta frequência. Um exemplo está em Schettini et al. (2019). [https://www.scielo.br/j/bjoce/a/Zqchf4bprXG5x4rFBMqZB3s/?lang=en].

Já antecipando, impossível trabalhar com maré sem o tempo, e tempo no Python significa objetos datetime! Já poderíamos ter incluido os objetos datetime que criamos no outro notebook no dicionário e carregar neste. Mas como não fizemos, fazemos de novo, e para isto basta copiar a função 'monta_datetime' (marca o conteúdo da célula, copia e cola em uma célula neste notebook, o jeito mais fácil de fazer isso!) (Cógido 3).

[12]:
```
def monta_datetime(d):

    tempo = []
    for i in range(len(d)):
        ano = int(d[i,0])
        mes = int(d[i,1])
        dia = int(d[i,2])
        hora = int(d[i,3])
        minuto = int(d[i,4])
        segundo = int(d[i,5])

        c_tempo = datetime.datetime(ano, mes, dia, hora, minuto, segundo)

        tempo.append(c_tempo)

    return tempo
```

Código 3: Função para criar objeto datetime.

Na sequência, criamos uma lista com as strings que nomeiam as chaves do dicionário e pegamos uma vítima (Código 4)! A variável 'estacoes' é uma lista dos nomes das chaves do dicionário, e a variável 'dados' contem agora uma lista com 4320 elementos da primeira chave (índice 0, lembrem-se!). É a estação Imbituba, mas isso de fato não importa! E, cada elemento da lista 'dados' é uma lista com oito elementos que sabemos ser ano, mês, dia, hora, minuto,

segundo, nível e nível. Dois níveis, porque algumas estações do IBGE registram dois! Se não, fica com um ‘not a number’ = nan.

```
[85]: estacoes = list(base_dados.keys())

      dados = base_dados[estacoes[0]]

      print(estacoes)
      print(len(dados), type(dados))
      print(dados[0])

      ['IMB', 'ARC', 'SAL', 'FOR', 'SAN']
      4320 <class 'list'>
      [2018, 3, 1, 0, 0, 0, 1.697, nan]
```

Código 4: Acessando conteúdo do dicionário.

A função ‘monta_datetime()’ foi elaborada considerando a estrutura que criamos para armazenar os dados no dicionário. Se usarmos a função diretamente com a lista ‘dados’ vai resultar em erro (Codigo 5). Apenas recapitulando, se não entender o erro, copie a última linha que começa com ‘TypeError’ e cola no Google e investiga! Especialmente no StackOverflow! As mensagens de erro não são muito elucidativas à primeira vista. Mas temos que o erro aconteceu na linha 5 da célula da função (ok! Eu esqueci de ativar o ‘show line numbers’, vai para os próximos códigos!), onde estamos atribuindo a ‘ano’ o valor inteiro que está em ‘d[i, 0]’, ou seja, a primeira coisa que está sendo feita é um fatiamento (slicing) de algo maior (‘d’), e o erro fala que o índice tem que ser números inteiros ou fatias, mas não tuplas! Confuso… mas nem tanto. A variável de entrada ‘d’ é ‘dados’, e ‘dados’ é uma lista, e cada elemento da lista é endereçado com um único índice, começando de 0. E alí está sendo informado dois índices (d[i,

0]). Você pode ir ao notebook anterior e lá, quando usamos esta função, verificar o ‘type()’ da variável de entrada que funcionou lá. Concluirá que para a função funcionar :-) devemos usar um arranjo no numpy e não uma lista.

```
[76]: tempo = monta_datetime(dados)

---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-76-c8549cbc2bdf> in <module>
----> 1 tempo = monta_datetime(dados)

<ipython-input-74-0702301b1cdf> in monta_datetime(d)
      3     tempo = []
      4     for i in range(len(d)):
----> 5         ano = int(d[i,0])
      6         mes = int(d[i,1])
      7         dia = int(d[i,2])

TypeError: list indices must be integers or slices, not tuple
```

Código 5: Usando a função ‘monta_datetime()’ e erro.

Então, transformamos ‘dados’ que é uma lista em um objeto arranjo (array) no numpy… e erro (Cógigo 6). Mas esse é fácil… se vamos usar o pacote numpy, temos que importar! Para manter o notebook organizado, fazemos isso lá na célula que importamos o pacote ‘pickle’, usando ‘import numpy as np’. Inserimos isso, executamos para o numpy ser carregado no kernel, e daí podemos executar todo o notebook ou só a célula que estamos trabalhando para fazer efeito. Executando novamente a célula… erro (Código 7). Tá difícil.. mas normal… eu, por experiência, já sei que cedo ou tarde vou usar os pacotes numpy, matplotlib e datetime… quase sempre. Então já começo importando estes pacotes. Aqui, para fins didáticos, por partes! Então, lá em cima, ‘import datetime’! Aproveita é já importa o matplotlib...

```
[7]:
1 dados = np.array(dados)
2
3 tempo = monta_datetime(dados)
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-7-f9165ea5e0db> in <module>
----> 1 dados = np.array(dados)
      2
      3 tempo = monta_datetime(dados)

NameError: name 'np' is not defined
```

Código 6: Convertento lista em arranjo, e erro.

```
[9]:
1 dados = np.array(dados)
2
3 tempo = monta_datetime(dados)
```

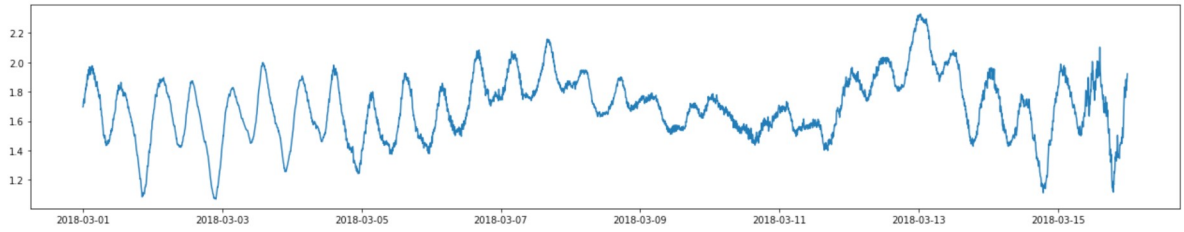
```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-9-f9165ea5e0db> in <module>
      1 dados = np.array(dados)
      2
----> 3 tempo = monta_datetime(dados)

<ipython-input-5-0702301b1cdf> in monta_datetime(d)
     10         segundo = int(d[i,5])
     11
---> 12         c_tempo = datetime.datetime(ano, mes, dia, hora, minuto, segundo)
     13
     14         tempo.append(c_tempo)

NameError: name 'datetime' is not defined
```

Código 7: Criando objeto datetime e erro!

Agora, pacotes carregados, tocamos o barco. Convertemos ‘dados’ para arranjo, criamos o objeto datetime para ‘dados’, muito originalmente como ‘tempo’, separamos os dados de nível da água contido em ‘dados’ e colocamos em ‘nivel’, e pra ver se tá tudo em ordem, fazemos um gráfico do nível em função do tempo (Código 8).

[14]:

```
dados = np.array(dados)

tempo = monta_datetime(dados)

nivel = dados[:,6]

plt.figure(figsize=(22,4))
plt.plot(tempo, nivel)
plt.show()
```



Código 8: Inspecionado os dados (sem erro!)

Podemos fazer uma rápida prospecção sobre as características desta série de dados. O período compreende 15 dias, pois fomos nós que pegamos inicialmente os arquivos do IBGE! Mas se não soubéssemos e quiséssemos saber, podemos simplesmente subtrair o último valor de 'tempo' pelo primeiro e obteremos um objeto 'timedelta'! E, dividindo o comprimento de 'tempo' pelo número de horas deste período, teremos quantos dados (em média, caso não seja constante) são registrados por hora, ou a taxa amostral. No caso desta estação o intervalo amostral é de 5 minutos (Código 9). Essa inspeção (e outras que achar pertinente) deve ser feita com todas as estações antes de continuarmos. Pra isso, basta mudar o índice de 'estacoes' quando fazemos a extração do conteúdo do dicionário. Responda: qual a taxa amostral das outras estações?

```
[16]:  1 tempo[-1] - tempo[0]

[16]: datetime.timedelta(days=14, seconds=86100)

[18]:  1 len(tempo)/(24*15)

[18]: 12.0
```

Código 9: Investigando características dos dados.

Após a inspeção, sabemos que a taxa amostral não é a mesma para todas as estações. E, a menor taxa amostral é de 12 amostras por hora (dt = 5 minutos). Para maioria dos casos, dados de nível do mar em intervalos de 1 hora são suficientes para analisar as componentes astronômicas e meteorológicas de baixa frequência. Porém, quanto maior for a taxa amostral, maior a resolução de fenômenos que podemos investigar. Por exemplo, seiches e tsunamis praticamente desaparecem com um intervalo de 1 hora, e para podermos identificar estes fenômenos, temos que ter maior resolução temporal possível. Um intervalo de 5 minutos já é ótimo para isso!

Ainda que os dados de Imbituba e outras já estejam com intervalo de 5 minutos, para ter certeza que as séries estarão completas e sem falhas, faremos a reamostragem interpolando os dados sobre um novo domínio temporal que temos certeza está regular e completo. Nós já inspecionamos todos os dados e não há nenhuma lacuna percebível… se houvesse, dependendo do tamanho, decidiríamos qual o melhor caminho a tomar. Mas isso não é um problema agora. Um problema (não é, de fato…) e verificar os inícios e fins das séries. Na inspeção, eu também avaliei a data/hora do primeiro e último registro. Todos começam no mesmo (2018-03-01 00:00:00), mas terminam em momentos diferentes, que é devido às diferentes taxas amostrais.

Os que tem taxa amostral terminam em (2018-03-15 23:55:00), e vamos usar este momento como limite superior. Responda, qual a data/hora de início e término de cada série?

Se você investigou e respondeu as perguntas, parabéns! É possível fazer isso substituindo o índice de ‘estacoes’ no Codigo 4, e executnado as células seguintes, uma vez para cada estação. Porém, o jeito mais elegante de fazer isso e sintetizar o código é usando um loop e imprimindo sequencialmente as informações desejadas. Uma outra coisa que sempre devemos verificar também em séries temporais é se não há repetições e se os dados estão arrumados cronologicamente (como o caso dos dados de vazão que usamos no tutorial sobre o Hidroweb, que precisaram ser arrumados quanto a isso!). Fazemos o gráfico do tempo, e neste caso, dentro do loop (Código 10). Todos os dados parecem que estão ok! Imbituba e Fortaleza não aparecem porque estão em baixo da linha de Santana. Salvador muda a taxa amostral, o que já tínhamos observado no notebook anterior. E Arraial do cabo tem muito mais dados do que as outras estações pois tem a taxa amostral maior.

Sobre o Código 10, ele tem a vantagem de fazer a inspeção de todas as estações rapidamente, e permite qualquer processamento adicional seja feito igualmente para todas as estações. Isso é bom mas também é ruim! Automatizar o processamento é bom, mas na análise de dados temos que ser muito cautelosos e nem sempre automatizar é o melhor caminho. O passo a passo, verificando para cada estação tratada, é recomendado. Depois que estivermos seguros de todos os passos e de que tudo está correto, aí sim podemos automatizar. Vamos continuar usando o Código 10, mas vamos fazer o loop rodar para um estação de cada vez primeiro. Para isso basta

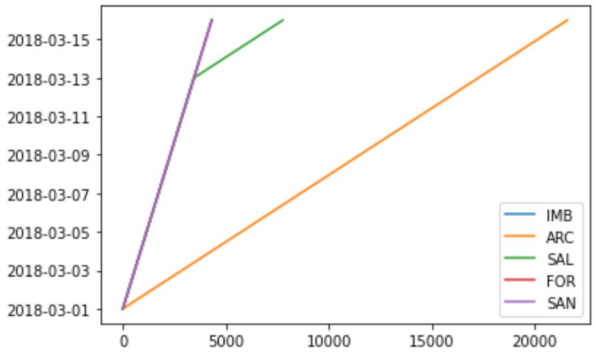
substituir a definição do loop por `for i in range(1):`, e faremos só para a 1.a estação. Se quisermos fazer só para a segundo a definição interna do 'range()' deverá ser 'range(1,2)', e assim por diante. Não entendeu? Veja o 'help' para o método 'range()'.

Já tínhamos um conhecimento prévio dos dados a partir do gráfico que fizemos no notebook anterior. Agora conhecemos mais: sabemos exatamente que começam no mesmo instante, mas não terminam no mesmo! Sabemos que as taxas amostrais são diferentes. Sabemos que os dados estão cronologicamente corretos. Ok… agora vamos a sincronização, que é reamostrar todos sobre um mesmo padrão de referência amostral.

[91]:

```
estacoes = list(base_dados.keys())

for i in range(len(estacoes)):

    dados = base_dados[estacoes[i]]

    dados = np.array(dados)

    tempo = monta_datetime(dados)

    nivel = dados[:,6]

    print(estacoes[i], 'início', tempo[0], 'fim', tempo[-1])
    plt.plot(tempo, label=estacoes[i])

plt.legend()
plt.show()
```

```
IMB início 2018-03-01 00:00:00 fim 2018-03-15 23:55:00
ARC início 2018-03-01 00:00:00 fim 2018-03-15 23:59:00
SAL início 2018-03-01 00:00:00 fim 2018-03-15 23:59:00
FOR início 2018-03-01 00:00:00 fim 2018-03-15 23:55:00
SAN início 2018-03-01 00:00:00 fim 2018-03-15 23:55:00
```



Código 10: Sintetizando a inspeção das séries temporais e gráfico do tempo de cada uma.

Podemos criar um arranjo de objetos datetime regularmente espaçados usando o ‘np.arange()’ tal qual usamos para números (experimente o np.arange()!). O ‘np.arange()’ precisa do valor inicial, final e do intervalo. O intervalo será de 5 minutos, e para criar um intervalo temporal usamos o módulo ‘timedelta()’ do pacote ‘datetime’. E, lembrando que o ‘np.arange()’ não inclui o último valor, temos que adicionar o intervalo no último (Código 11). Nesse código

eu pedi para imprimir os tipos dos diferentes objetos. Em tutoriais anteriores eu já enfatizei a problemática de trabalhar com datas (calendário Gregoriano) com computação em geral. O Python ‘nativo’ tem o pacote ‘datetime’ para trabalhar com datas e horas, mas o pacote ‘numpy’ tem um método alternativo que é o ‘datetime64’. Usando o np.datetime64(), o Código 11 ficaria como Cógido 12.

```
[115]:
1 tempo_inicio = datetime.datetime(2018, 3, 1, 0, 0, 0)
2
3 tempo_fim = datetime.datetime(2018, 3, 15, 23, 55, 0)
4
5 intervalo = datetime.timedelta(minutes = 5)
6
7 tempo_ref = np.arange(tempo_inicio, tempo_fim + intervalo, intervalo)
8
9 print(type(tempo_ref), type(tempo_ref[0]), type(tempo_inicio))
```

```
<class 'numpy.ndarray'> <class 'numpy.datetime64'> <class 'datetime.datetime'>
```

Código 11: Criando arranjo de tempo usando datetime.

```
[138]:
1 tempo_inicio = np.datetime64('2018-03-01T00:00:00')
2
3 tempo_fim = np.datetime64('2018-03-15T23:55:00')
4
5 intervalo = np.timedelta64(5, 'm')
6
7 tempo_ref = np.arange(tempo_inicio, tempo_fim + intervalo, intervalo)
8
9 print(type(tempo_ref), type(tempo_ref[0]), type(tempo_inicio))
```

```
<class 'numpy.ndarray'> <class 'numpy.datetime64'> <class 'numpy.datetime64'>
```

Código 12: Criando arranjo de tempo usando numpy.datetime64.

Eu uso preferencialmente o ‘datetime’, mas é importante saber que existe o ‘datetime64’, e que os objetos são diferentes, e isso pode gerar erros. Por exemplo, se subtrair um objeto

datetime de outro, a diferença será um objeto timedelta (do datetime). Se subtrair um objeto datetime64 de outro, a diferença será um objeto timedelta64 (do numpy). Se subtrair um objeto datetime de um objeto datetime64, dará um erro! Experimente!

Para interpolar, até onde eu experimentei não dá para fazer nem com objetos datetime nem datetime64. A solução mais simples é converter os objeto datetime em float. Para isso, carregamos o pacote `import matplotlib.dates as mdates`, e usamos o módulo 'date2num' (= data para número!), e também, caso necessário, há o que faz o inverso, 'num2date' (= número para data!) (Código 13). Para a primeira estação (Imbituba), não fará diferença e desnecessário a primeira vista! Mas como faremos o mesmo com todas as estações, o resultado é que ao final todas estarão com o mesmo tamanho e mesma taxa de amostragem!

[140]:
```
1 tempo_n = mdates.date2num(tempo)
2 tempo_ref_n = mdates.date2num(tempo_ref)
3
4 nivel_i = np.interp(tempo_ref_n, tempo_n, nivel)
5
6 print(len(nivel), len(nivel_i))
```

```
4320 4320
```

Código 13: Interpolando o nível para o tempo de referência.

Agora é um momento para uma arrumação do código. Lembrando que durante o processo, vamos tentando e testando coisas diferentes e tomando decisões sobre os próximos passos… até aqui o objetivo foi avaliar a consistência dos dados e sincronizar todos para a mesma taxa amostral. Incorporando isto ao que temos e ordenando logicamente, a criação do

tempo de referência (datetime e numérico) vem antes do processamento no loop, e, após interpolar, eu preciso guardar o resultado. No caso, vou montar uma matriz que terá 5 colunas, cada qual para uma estação, na ordem de processamento (Código 14). Preste especial atenção na estrutura lógica que usei para isso, pois é muito útil: **1.o – crio a variável para armazenar resultados ('g_nivel_i' → g de guardo, i de interpolado); 2.o – converto o resultado da interpolação de arranjo do numpy para lista, usando o método 'tolist()'; 3.o – concateno a lista em 'g_nivel_i'; 4.o – reformo a lista para 5 linhas e o número complementar de colunas (-1) e transponho, sendo o resultado do np.reshape() um arranjo!**

Agora é um bom momento para dar uma nova olhadela nos dados (Código 15). So far, so good… olhando o gráfico notamos que faltou arrumar a referência vertical, ou, tornar todos os dados de nível relacionados com uma referência vertical comum. Como não temos (nem precisamos) de um datum vertical, basta subtrair cada série do seu valor médio, depois de interpolar e antes de concatenar para guardar. Tente fazer isso. O resultado será a mesma figura porém com todas as séries oscilando em torno do nível zero (Código 16). Chegamos no mesmo ponto que no tutorial anterior, porém com a diferença (muito importante) que os dados agora estão consistidos, padronizados e sincronizados e prontos para análises posteriores.

[143]:

```
tempo_inicio = datetime.datetime(2018, 3, 1, 0, 0, 0)

tempo_fim = datetime.datetime(2018, 3, 15, 23, 55, 0)

intervalo = datetime.timedelta(minutes = 5)

tempo_ref = np.arange(tempo_inicio, tempo_fim + intervalo, intervalo)

tempo_ref_n = mdates.date2num(tempo_ref)

```

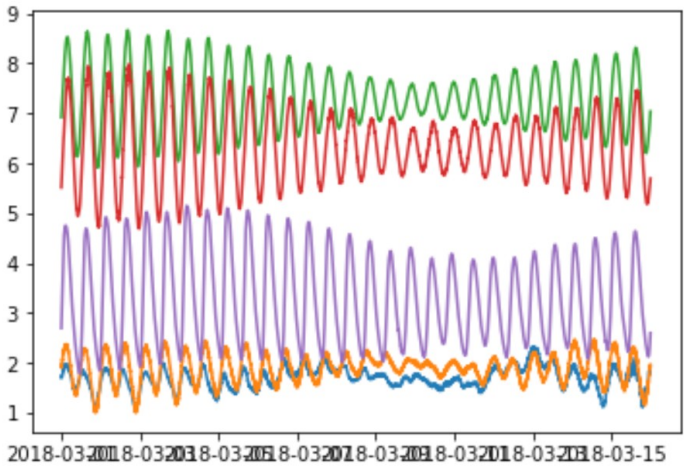
[145]:

```
estacoes = list(base_dados.keys())

g_nivel_i = []

for i in range(5):

    dados = base_dados[estacoes[i]]

    dados = np.array(dados)

    tempo = monta_datetime(dados)

    nivel = dados[:,6]

    tempo_n = mdates.date2num(tempo)

    nivel_i = np.interp(tempo_ref_n, tempo_n, nivel).tolist()

    g_nivel_i = g_nivel_i + nivel_i

nivel_i = np.reshape(g_nivel_i, (5,-1)).T
```

Código 14: Código reformado!

[152]:

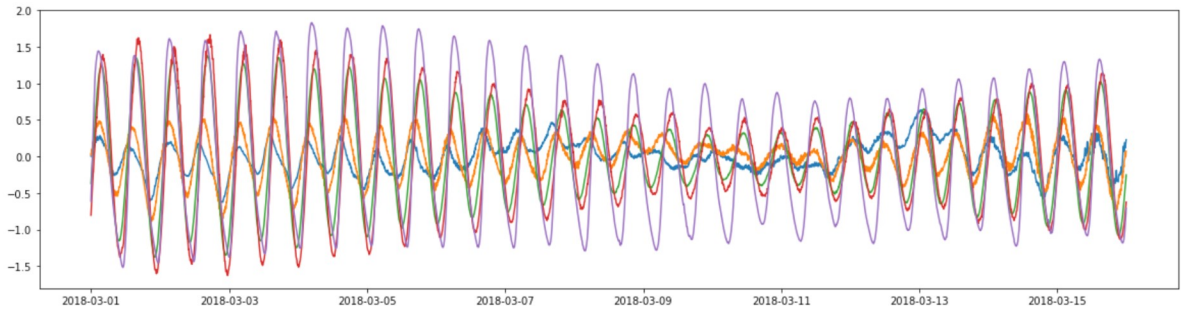
```
1 plt.plot(tempo_ref, nivel_i)
2 plt.show()
```



Código 15: Verificando o andamento do processamento.

[156]:

```
1 plt.figure(figsize=(20,5))
2 plt.plot(tempo_ref, nivel_i)
3 plt.show()
```



Código 16: Mesmo que o Código 15, porém com os dados uniformizados para variar em torno do nível zero.

O sinal do nível da água tem uma componente determinística associada às marés astronômicas e uma componente probabilística causada por outros fenômenos (vento, seiches, tsunamis, etc.). A análise harmônica de marés é a ferramenta que utilizamos para identificar os constituintes harmônicos do sinal. Existe a análise harmônica e a análise harmônica de marés. A primeira identifica se há sinais periódicos em uma série temporal, e nos fornece o período, amplitude e fase destes sinais. No caso da análise harmônica de maré, nós já sabemos em que períodos procurar a energia (amplitude) do sinal. Como a maré astronômica, como o nome diz, advém da resposta aos ciclos astronômicos da Terra, Lua e Sol, nós conhecemos muito bem os ciclos. Por exemplo, a principal forçante das marés é a Lua com metade do dia lunar, cujo período é de 12,4206012 horas (repare a precisão! E veja Theory of tides na Wikipedia!). Então, avaliamos na série temporal quanta energia está contida em oscilações com este período e qual a fase em relação a um sistema referencial (Greenwich), e assim por diante.

Há vários pacotes que fazem a análise harmônica de maré para Python. Usaremos aqui o U-Tide de Daniel Codiga (http://www.po.gso.uri.edu/~codiga/utide/utide.htm, https://www.researchgate.net/publication/280722790_Unified_tidal_analysis_and_prediction_using_the_Utide_Matlab_functions), feito originalmente para Matlab, e traduzido para Python por Wesley Bowman (https://github.com/wesleybowman/UTide). No Github há ampla documentação de como instalar e usar o pacote, e Codiga (2011) apresenta uma extensiva explanação teórica sobre a análise. Carregando o pacote: `from utide import solve, reconstruct`

Para usar (leia o manua!), precisamos de um arranjo de tempo no formato numérico (tempo_ref_n), o nível (cada coluna de 'nivel_i'), e a latitude de cada estação. Voltamos ao site do IBGE/RMPG e pegamos as latitudes e criamos uma lista com elas, na mesma ordem que estão as estações nas colunas de 'nivel_i'. As coordenadas geográficas no site estão decompostas em graus, minutos e segundos, então temos que transformar em graus e décimos de graus, lembrando que estamos no hemisfério sul (Código 17)!

[183]:
```
# latitudes das estações a partir do IBGE/RMPG

lat_imb = -(28 + 13/60 + 52.3/3600)
lat_arr = -(22 + 58/60 + 21/3600)
lat_sal = -(12 + 58/60 + 26.29/3600)
lat_for = -(3 + 42/60 + 52.55/3600)
lat_san = -(0 + 3/60 + 41/3600)

latitudes = [lat_imb, lat_arr, lat_sal, lat_for, lat_san]
```

Código 17: Criando lista com as latitudes das estações.

Agora estamos prontos para fazer a análise harmônica (Cógdigo 18)! Responda: porque estamos somando 3/24 ao tempo? Seguindo a lógica usual, criamos listas vazias que acumularão os resultados das análises. Lembrando que qualquer objeto por ser inserido em uma lista… verifique de que tipo são os objetos 'coef' e 'sinal_h'! Estas são estruturas tipo dicionários, e pode seu conteúdo pode ser explorado usando o método 'keys()'.

[194]:

```
coefs = []
g_sinal_h = []

for i in range(5):

    nivel = nivel_i[:,i]

    coef = solve(tempo_ref_n - 3/24, nivel,
                 lat = latitudes[0],
                 method = 'ols',
                 conf_int = 'MC')

    coefs.append(coef)

    sinal_h = reconstruct(tempo_ref_n - 3/24, coef)

    g_sinal_h.append(sinal_h)
```

```
solve: matrix prep ... solution ... done.
prep/calcs ... done.
solve: matrix prep ... solution ... done.
prep/calcs ... done.
solve: matrix prep ... solution ... done.
prep/calcs ... done.
solve: matrix prep ... solution ... done.
prep/calcs ... done.
solve: matrix prep ... solution ... done.
prep/calcs ... done.
```

Código 18: Fazendo a análise harmônica (solve!) e a reconstrução do nível harmônico (reconstruct!).

Uma vez que temos o sinal harmônico, podemos subtraí-lo do sinal original, e teremos então o sinal não-harmônico. Podemos aproveitar e fazer uma primeira inspeção gerando os gráficos de cada componente (Código 19, Figura 4). O nosso objetivo não é uma interpretação mais profunda, mas tentem identificar as semelhanças e diferenças entre as colunas dentro do contexto do que fizemos até agora. A coluna do meio é a reconstrução do sinal observado quando suas componentes harmônicas são quantificadas. Notem que o sinal não-harmônico (3.a

coluna) ainda contém oscilações periódicas não explicadas pela análise. Isto pode ser parcialmente entendido pelo fato que para realizar uma análise harmônica mais eficiente, precisamos pelo menos 29 dias de dados (1 mês lunar, mais ou menos!). Um período de 15 dias resolve bem as diferentes entre componentes diurnas (~ 25 horas) e semi-diurnas (~ 12.5 horas), mas não resolve tão bem os diferentes constituintes semi-diurnos (M_2, S_2, etc.). Como todo método, por melhor que seja, não faz milagres! A interpretação do resultado deve considerar isso.

[208]:

```
g_nivel_nao_h = []

for i in range(5):

    nivel_h = g_sinal_h[i]['h']

    nivel_nao_h = nivel_i[:,i] - nivel_h

    g_nivel_nao_h.append(nivel_nao_h)

    fig, (ax1, ax2, ax3) = plt.subplots(1, 3, figsize=(18, 3))
    ax1.plot(tempo_ref, nivel_i[:,i])

    ylims = ax1.get_ylim()

    ax2.plot(tempo_ref, nivel_h)
    ax2.set_ylim(ylims)

    ax3.plot(tempo_ref, nivel_nao_h)
    ax3.set_ylim(ylims)

    if i == 0:
        ax1.set_title('Sinal observado')
        ax2.set_title('Sinal harmônico')
        ax3.set_title('Sinal não-harmônico')


```

Código 19: Inspeção dos resultados da análise harmônica.

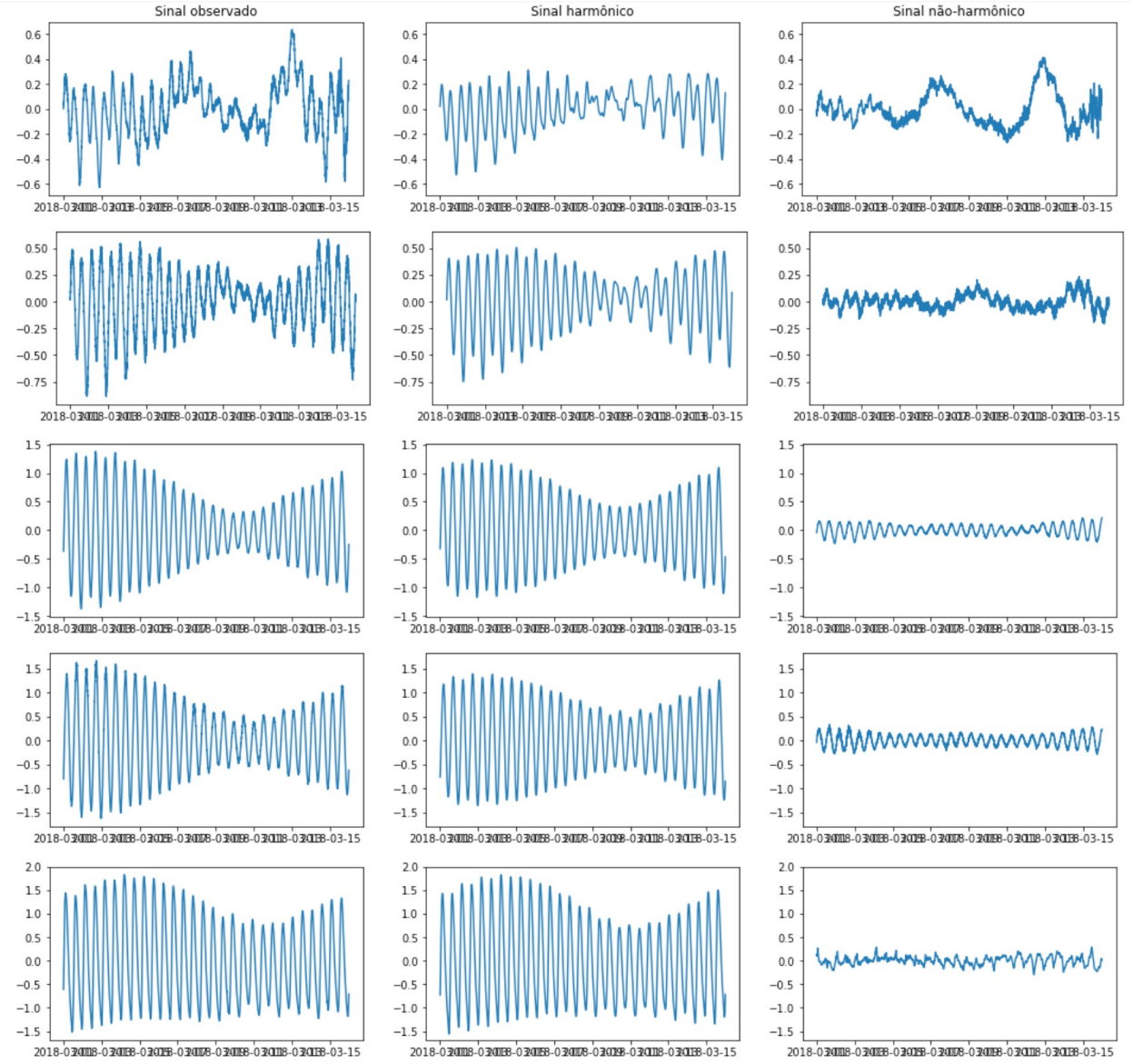


Figura 4: Resultado do Código 19.

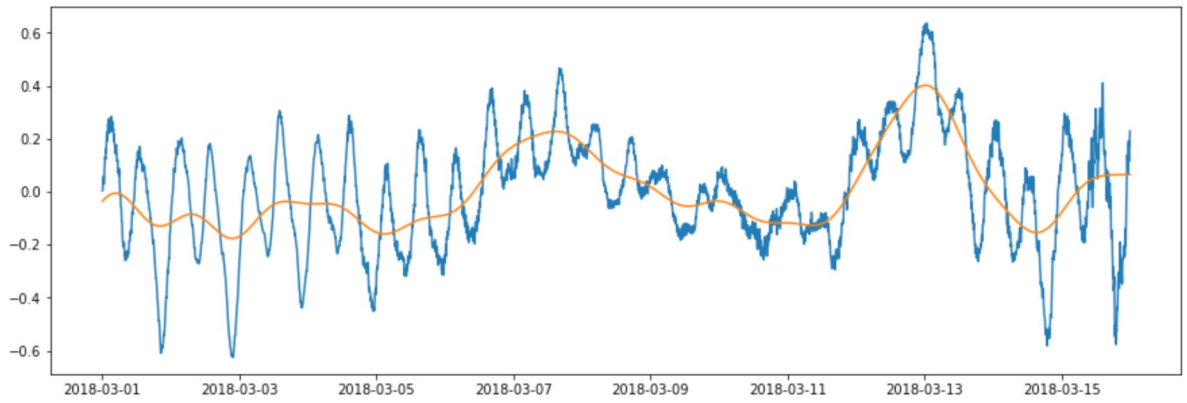
Uma opção agora e investigar como a energia (variância) no sinal não harmônico está distribuída nas bandas de baixa-frequência (períodos > 30 horas), alta-frequência (períodos < 2 horas) e intermediária (30 horas > períodos > 2 horas). Esses limites podem variar um pouco mas

estabelecem bem o que consideremos oscilações de períodos supra-mareias e infra-mareais. Para separar as bandas nós empregamos filtros numéricos. Filtros numéricos são ferramentas essenciais na análise de séries temporais de qualquer natureza, e são muito usados em tele-comunicações e eletrônica. Há uma extensa literatura sobre filtros numéricos. Para análise de nível da água os mais comumente utilizados são só filtros Butterworth e Lanczos. São filtros diferentes, mas o resultado final é praticamente o mesmo. O Butterworth, até onde sei, é o mais amplamente utilizado, e tem como vantagem já estar disponível para Python pelo pacote Scipy.

Inicialmente, carregamos o pacote `import scipy.signal as signal`. Para usar o filtro, primeiro temos que delinear o filtro para depois correr o filtro pela série. Filtragem, de maneira geral, é um processo de convolução (palavra nova!), onde duas funções interagem para produzir uma terceira. Antes de rodar para todas as amostras, vamos testar sobre os dados de Imbituba, e vamos fazer uma filtragem para analisar as oscilações de baixa frequência, ou, aquelas com período maior do que 30 horas. Estabelecemos a frequencia amostral (12 por hora), a ordem do filtro (Wikipedia?) e a frequência de corte (inverso do período = 1/30 horas!). Delineamos o filtro com o método 'butter' do pacote 'signal' (tente signal.butter?), e filtramos, armazenando o resultado em 'nivel_f_30'… e já aproveitamos para ver o que deu (Código 20). A linha laranja é o sinal filtrado de baixa frequência, e podemos ver que o sinal semi-diurno das marés astronômicas foi completamente removido. Um aviso é que em um processo de filtragem, as extremidades da série não são bem resolvidas, em uma escala de metade do período de corte. Ou seja, as primeiras 15 horas e as últimas 15 horas da série filtradas são lixo! Não devem ser interpretadas como sinal!

```
nivel = nivel_i[:, 0]

frequencia_amostral = 12 # 12 por hora, ou a cada 5 minutos!

ordem_filtro = 5

frequencia_corte = 1/30 # período de 30 horas

# delineamento do filtro
B_30, A_30 = signal.butter(ordem_filtro,
                           frequencia_corte,
                           fs = frequencia_amostral,
                           btype  = 'lowpass')

# filtragem
nivel_f_30 = signal.filtfilt(B_30, A_30, nivel)

plt.figure(figsize=(15,5))
plt.plot(tempo_ref, nivel)
plt.plot(tempo_ref, nivel_f_30)
```

```
[<matplotlib.lines.Line2D at 0x1ae3c8b7c40>]
```



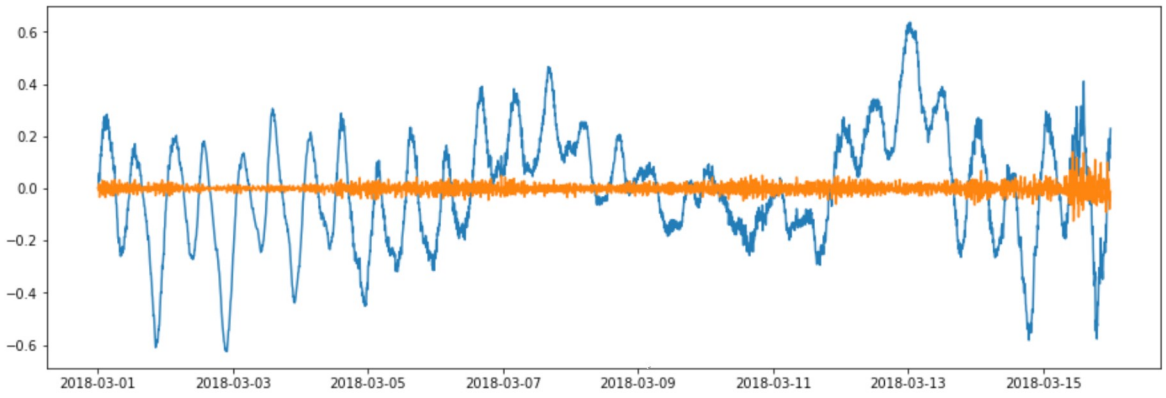
Código 20: Filtragem da série de Imbituba para obter oscilações com período maior do que 30 horas.

Similarmente com o que fizemos para a baixa frequência, podemos fazer para a alta frequência (Código 21)! E, ainda, podemos usar o filtro como passa-banda, escolhendo a faixa intermediária (Código 22).

[250]:

```
nivel = nivel_i[:, 0]

frequencia_amostral = 12 # 12 por hora, ou a cada 5 minutos!

ordem_filtro = 5

frequencia_corte = 1/2# período de 30 horas

# delineamento do filtro
B_2, A_2 = signal.butter(ordem_filtro,
                         frequencia_corte,
                         fs = frequencia_amostral,
                         btype  = 'highpass')

# filtragem
nivel_f_2 = signal.filtfilt(B_2, A_2, nivel)

plt.figure(figsize=(15,5))
plt.plot(tempo_ref, nivel)
plt.plot(tempo_ref, nivel_f_2)
```

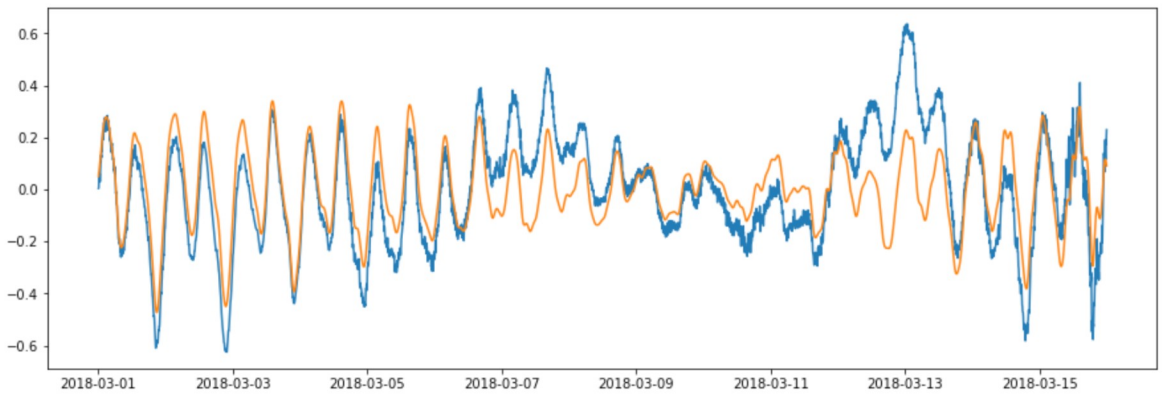
[250]: [<matplotlib.lines.Line2D at 0x1ae3d561a90>]



Código 21: Filtragem da série de Imbituba para obter oscilações de período menor do que 2 horas.

[260]:
```
nivel = nivel_i[:, 0]

frequencia_amostral = 12 # 12 por hora, ou a cada 5 minutos!

ordem_filtro = 5

frequencia_corte_baixa_f = 1/30
frequencia_corte_alta_f = 1/2

# delineamento do filtro
B_30_2, A_30_2 = signal.butter(ordem_filtro,
                           [frequencia_corte_baixa_f, frequencia_corte_alta_f],
                           fs = frequencia_amostral,
                           btype  = 'band')

# filtragem
nivel_f_30_2 = signal.filtfilt(B_30_2, A_30_2, nivel)

plt.figure(figsize=(15,5))
plt.plot(tempo_ref, nivel)
plt.plot(tempo_ref, nivel_f_2)
```

[260]: `[<matplotlib.lines.Line2D at 0x1ae3c7d4250>]`



Código 22: Filtragem da série de Imbituba para obter oscilações de período entre 2 e 30 horas.

Agora que já entendemos as partes e como funciona o filtro, podemos reformar o código para deixar mais sucinto e sem os plots, que faremos todos juntos depois (Código 23). Geralmente é esse tipo de código que obtemos dos outros… o código funcional mais sintético. Mas para chegar nele houve o desenvolvimento das partes. E isso que é importante quando tentamos entender o código que alguém fez! Isso pode ser chamada de engenharia reversa, que é pegar o código e ir fragmentando ele para entendermos as partes menores. Uma vez que

entendemos, fica mais fácil saber onde podemos mexer para adaptar para uma outra situação ou de uma forma que fique mais claro para nós mesmos. Por exemplo, é a partir da linha 26 que é executada a filtragem e o re-arranho das séries em uma lista e depois acumulada em outra lista.

```
1  frequencia_amostral = 12 # 12 por hora, ou a cada 5 minutos!
2  ordem_filtro = 5
3  frequencia_corte_baixa = 1/30    # = periodo de 30 horas
4  frequencia_corte_alta = 1/2      # = período de 2 horas
5
6  # delineamento filtro passa baixa
7  B_30, A_30 = signal.butter(ordem_filtro, frequencia_corte_baixa,
8                             fs = frequencia_amostral,vbtype  = 'lowpass')
9
10 # delineamento do filtro passa alta
11 B_2, A_2 = signal.butter(ordem_filtro, frequencia_corte_alta,
12                            fs = frequencia_amostral, btype  = 'highpass')
13
14 # delineamento do filtro passa banda
15 B_30_2, A_30_2 = signal.butter(ordem_filtro,
16                            [frequencia_corte_baixa, frequencia_corte_alta],
17                            fs = frequencia_amostral, btype  = 'band')
18
19 def filtra_series(serie):
20     serie_f_30 = signal.filtfilt(B_30, A_30, serie)
21     serie_f_30_2 = signal.filtfilt(B_30_2, A_30_2, serie)
22     serie_f_2 = signal.filtfilt(B_2, A_2, serie)
23     serie_f = [serie_f_30, serie_f_30_2, serie_f_2]
24     return serie_f
25
26 g_niveis = []
27 for i in range(5):
28     org_niveis = []
29     nivel = nivel_i[:,i]                                  # nivel interpolado, sinal original
30     nivel_f = filtra_series(nivel)                        # nivel filtrado nas bandas (3 séries)
31     nivel_h = g_sinal_h[i]['h']                           # recupera nível harmônico
32     nivel_nh = np.array(nivel_f[1]) - np.array(nivel_h)   # calcula o nível não harmônico (30/2)
33
34     org_niveis.append(nivel)      # organiza o ordem das séries na lista de saída!
35     org_niveis.append(nivel_h)
36     org_niveis.append(nivel_f[0])
37     org_niveis.append(nivel_nh)
38     org_niveis.append(nivel_f[2])
39
40     g_niveis.append(org_niveis)
```

Código 23: Reformando o código.

E, sempre, inspecionamos visualmente o resultado (Código 24). Imagino que a esta altura já estamos ficando acostumados com a 'engenharia reversa', certo? Então não vou me ater às minuncias do código. Mas informo que não foi elaborado de 'sopetão'… muitos erros, indas e vindas, consultas no Stackoverflow, até chegar nesse aí. Por exemplo, em algum lugar lá em cima foi criado o objeto 'tempo_ref', usando o np.arange(), e é do tipo 'datetime64'. Mas eu prefiro trabalhar com objetos 'datetime', então tive que converter de 'datetime64' para 'datetime' na linha 1. Mas se deixar ativo, em uma segunda vez que eu rodar do código vai dar erro, pois estarei tentando transformar uma lista em lista… então eu comentei para não dar mais erro. E, lógicamente, essa transformação ficaria melhor lá em cima próximo de onde 'tempo_ref' foi criado. É o que eu faria normalmente, mas daí ficaria complicado para mostrar para vocês.

O resto é definição de axes (os painéis onde vão os gráficos = 25), parametrização dos limites x e y em listas, bem como dos títulos. Depois vem a montagem da matriz de gráficos, em linhas (primeiro 'for') e colunas (segundo 'for')… uma verdadeira massaroca! É isso ou repetir o código de fazer um gráfico 25 vezes que definitivamente é o que alguém faria no EXCEL, mas daí não precisaria estar aprendendo Python.

Caso alguém se pergunte sobre o porque da linha 7… porque cortar da figura (os dados continuam lá!) as primeiras e últimas 15 horas? É por causa das caudas do filtro de passa-baixa de 30 horas. Os extremos (½ do período de filtro!) é lixo… veja por si mesmo. O resto é estética! A figura resultante do Código 23 e apresentada na Figura 5.

[400]:

```
# tempo_ref = tempo_ref.tolist()
plt.rcParams.update({'font.size': 16})
fig, axs = plt.subplots(5,5, figsize=(22,12))
fig.subplots_adjust(hspace = .08, wspace=.1)

estacoes = ['Imbituba', 'Arraial do Cabo', 'Salvador', 'Fortaleza', 'Santana']
xlims = [tempo_ref[0] + datetime.timedelta(hours=15), tempo_ref[-1] - datetime.timedelta(hours=15)]
ylims = [[-.7, .7], [-.8, .8], [-1.4, 1.4], [-1.6, 1.6], [-1.9, 1.9]]
titles = ['Sinal original', 'Sinal harmônico', 'Baixa frequência', 'Sinal não harmônico(30-2)', 'Alta frequência']

axs = axs.ravel()

props = dict(boxstyle='round', facecolor='wheat', alpha=0.3)
ax = 0
for li in range(5):

    for co in range(5):

        p_niveis = g_niveis[li]

        p_niveis2 = p_niveis[co]

        axs[ax].plot(tempo_ref, p_niveis2)
        axs[ax].set_xlim(xlims)
        axs[ax].set_ylim(ylims[li])

        if li < 4:
            axs[ax].set_xticklabels('')
        if co > 0:
            axs[ax].set_yticklabels('')
        if co == 0:
            axs[ax].set_ylabel('Nível (m)')
            axs[ax].text(xlims[0] + datetime.timedelta(hours = 12), ylims[li][1] - ylims[li][1]*.28, estacoes[li], bbox=props)
        if li == 4:
            axs[ax].xaxis.set_major_formatter(mdates.DateFormatter('%d'))
            axs[ax].set_xlabel('Março, 2018')
        if li == 0:
            axs[ax].set_title(titles[co])

        ax += 1

```

Código 24: Visualizando o resultado do Código 23.

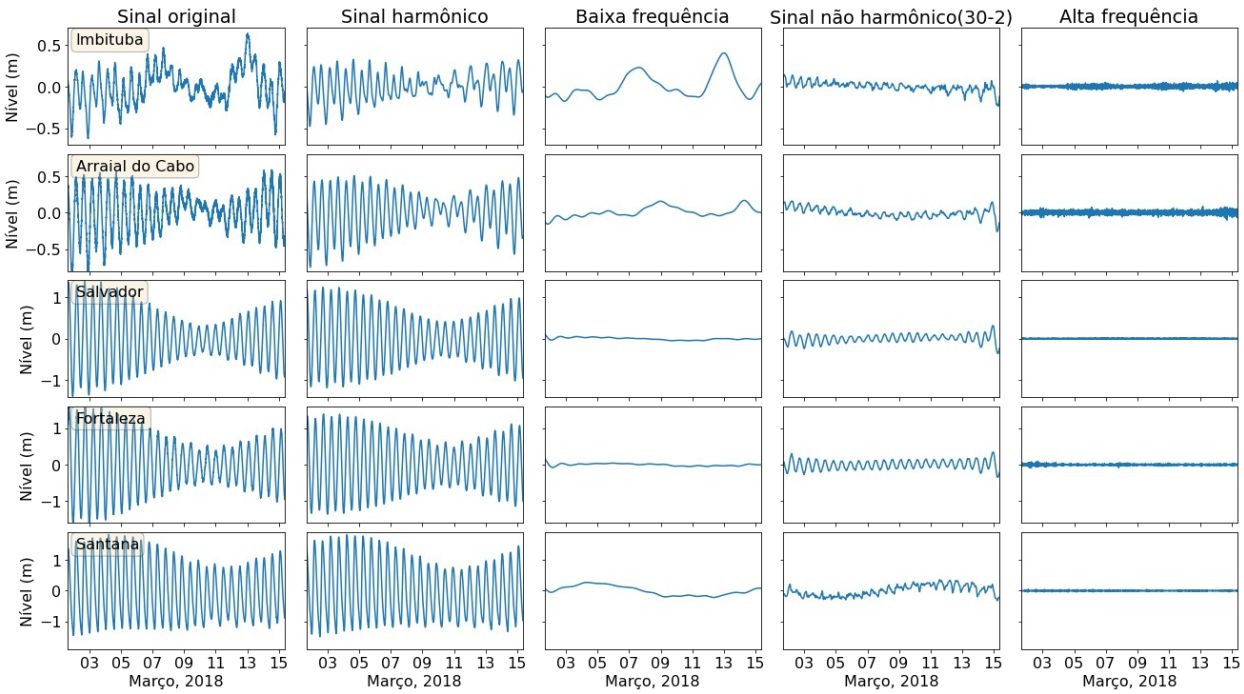


Figura 5: Resultado da decomposição do sinal do nível da água.

A Figura 5, tal qual está, já permite a identificação de várias coisas interessantes nestes dados! Mas sua interpretação está além do nosso objetivo agora. Ela serve para a visualização das diferentes componentes do nível da água. Em termos objetivos, o que precisamos é tabular a quantidade de energia de cada componente, ou, como a variância está distribuída. Pra isto, calculamos a variância de cada componente e o seu percentual relativo em relação as diferentes bandas. Pra começar, temos que eliminar as primeiras e últimas 15 horas dos dados, e depois obter as variâncias. No Código 23 nós apensas omitimos da visualização reduzindo os limites do eixo X, mas para qualquer cálculo nós devemos remover esses dados. Mas, remover em si (lógica humana!) é mais complicado do que eu fazer uma cópia excluindo o que eu não quero (lógica de computador!), ou seja, fazendo um fatiamento.

Para fatiar usando um ‘timedelta’ precisamos trabalhar com arranjo, então, convertemos de volta o ‘tempo_ref’ para arranjo. Aqui não há problema de não comentar o código, pois o numpy é mais flexível do que o Python nativo… se o objeto já é arranjo, não resultará em erro. Para fatiar o tempo criamos outro objeto ‘tempo_ref_2’. Para fatiar os dados, como são 5 estações x 5 séries temporais, temos que fazer dois loops. Guardar as séries fatiadas é facultativo, mas quem sabe faremos algo com elas depois. O objetivo aqui é calcular a variância de cada série e guardar essa informação, e vou criar uma matriz de zeros de 5 x 5 e enchê-la de nan, para então ser preenchida com os resultados da variância dentro do loop (Código 25).

[490]:

```
tempo_ref = np.array(tempo_ref)
td_15_hs = datetime.timedelta(hours = 15)

tempo_ref_2 = tempo_ref[(tempo_ref > tempo_ref[0] + td_15_hs) & (tempo_ref < tempo_ref[-1] - td_15_hs)]

g_var = np.zeros((5, 5))
g_var[:] = np.nan

g_niveis_2 = []
for li in range(5):

    lista_temporaria = []
    var = []

    for co in range(5):
        pega_lista = g_niveis[li][co]

        pega_lista = pega_lista[(tempo_ref > tempo_ref[0] + td_15_hs) & (tempo_ref < tempo_ref[-1] - td_15_hs)]

        calc_var = np.var(pega_lista)

        g_var[li, co] = calc_var

        lista_temporaria.append(pega_lista)


    g_niveis_2.append(lista_temporaria)

```

Código 25: Fatiando os dados e calculando a variância das séries.

Agora, como esperado, visualizamos o resultado. No caso, vou comparar a variância do sinal observado com o somatório da variância do sinal decomposto, e para este tipo de comparação é melhor usar um gráfico de barras. Obviamente, nada é tão simples (Dunning-Kruger), e após algumas visitas ao Stakoverflow, temos o Código 26. A mensagem na caixa rosa é um aviso (warning), não um erro. Interpretando rapidamente o gráfico, parece tudo em ordem! Aumenta a amplitude de maré (½ da altura, só para lembrar!), aumenta a variância que é uma maneira de quantificar da energia do sinal. O que não está parecendo em ordem é que o somatório da variância do sinal decomposto está maior do que a variância do sinal original. Fisicamente falando, energia não surge do nada (1.a lei da termodinâmica!). No momento, sem uma análise mais profunda, acho que isso é porque a janela do filtro não é 'idealmente' quadrada, e há uma pequena sobreposição dos sinais filtrados, o que é chamado de 'vazamento espectral'. Aumentando a ordem do filtro talvez minimize isso, mas há um limite. → Dunning-Krugger → Leitura! Wikipedia pra começar!

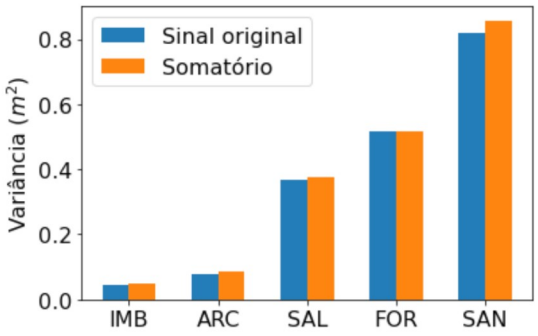
O valor da variância em si não é tão interessante de analisarmos. No caso, vamos avaliar as componentes relativas em termos de fração (ou percentual) do total. Ou seja, a razão entre a variância de cada componente pelo somatório da variância de todos os componentes. Não dá para usar a variância do sinal original uma vez que vimos que não bate exatamente! Calculando … Código 27, erro! Ambos objetos 'g_var' quanto 'sinal_d' são arranjos, e uma ótima coisa de trabalhar com arranjos é poder fazer operações *broadcast*, tipo, operações matemática de elemento a elemento para arranjos do mesmo tamanho. Tamanhos diferentes = erro! O problema aqui são as dimensões… toda vez que eu faço um fatiamento ou uma operação que resulta em

um arranjo uni-dimensional (uma linha ou uma coluna), ele passa a ser, literalmente, no Python, uni-dimensional. Então, se tinha um arranjo que tinha 5 linhas e 5 colunas, o *shape* deste objeto é (5, 5). Se eu fatiar uma coluna ou fazer a média de todas as linhas, isso resultará em um objeto arrano com *shape* (5,). Diz que tem 5 elementos, sem explicitar se é linha ou coluna! Para eu poder fazer uma operação do tipo *broadcast* com este objeto, como no nosso caso, temos que forçar a bidimensionalidade. Neste caso, dizer que isto é uma coluna (Código 28)! E o '.T'?

[515]:

```
sinal_o = g_var[:,0]
sinal_d = np.sum(g_var[:,1:], axis=1)

labs = list(base_dados.keys())
labs.insert(0, '')
xlabs = np.arange(0, 5)

fig, ax = plt.subplots()
ax = plt.gca()

ax.bar(xlabs - .15, sinal_o, width=.3, label='Sinal original')
ax.bar(xlabs + .15, sinal_d, width=.3, label='Somatório ')
ax.set_xticklabels(labs)
ax.set_ylabel('Variância ($m^2$)')

plt.legend()
plt.show()
```

```
<ipython-input-515-66eebe275904>:13: UserWarning: FixedFormatter should only be used together with FixedLocator
  ax.set_xticklabels(labs)
```



Código 26: Inspecionando os resultados do cálculo das variâncias.

```
[513]:  1  var_percentual = g_var[:,1:]/sinal_d*100
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-513-0ce9fadf7b01> in <module>
----> 1 var_percentual = g_var[:,1:]/sinal_d*100

ValueError: operands could not be broadcast together with shapes (5,4) (5,)
```

Código 27: Calculando o percentual das componentes, com erro.

```
[525]:  1  sinal_d = np.atleast_2d(sinal_d).T
        2
        3  var_percentual = g_var[:,1:]/sinal_d*100
```

```
[528]:  1  print(var_percentual.round(2))
```

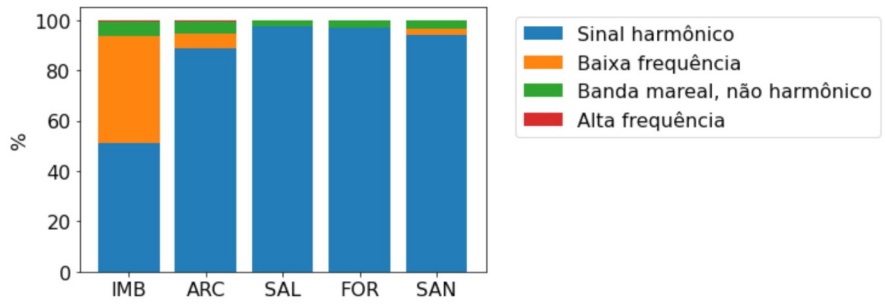
```
[[51.18 42.33  6.08  0.41]
 [88.95  5.79  4.78  0.48]
 [97.23  0.23  2.53  0.01]
 [97.01  0.19  2.76  0.04]
 [93.94  2.69  3.36  0.01]]
```

Código 28: Calculando o percentual das components, sem erro. E visualizando resultado.

Atenção ao Código 28, pois ele faz uma referência circular. Eu atribuo a ‘sinal_d’ o retorno de uma alteração feita sobre ‘sinal_d’, que será utilizada em um contexto específico na linha de código seguinte. Se rodar duas vezes esse código dará erro! Como evitar esse tipo de erro?

Gerar tabelas de resultados não é o forte do Python… para um relatório ou artigo, tem que usar um processador de texto para arrumar a saída numérica. Porém, o mesmo resultado podemos apresentar graficamente (Código 29).

```
[557]:
fig, ax = plt.subplots()

x= np.arange(0,5)

ax.bar(x, var_percentual[:,0], label = 'Sinal harmônico')
ax.bar(x, var_percentual[:,1], bottom=var_percentual[:,0], label = 'Baixa frequência')
ax.bar(x, var_percentual[:,2], bottom=var_percentual[:,0] + var_percentual[:,1], label = 'Banda mareal, não harmônico')
ax.bar(x, var_percentual[:,3], bottom=var_percentual[:,0] + var_percentual[:,1] + var_percentual[:,2], label = 'Alta frequência')

ax.legend(bbox_to_anchor=(1.05, 1))
ax.set_xticklabels(labs)
ax.set_ylabel('%')

<ipython-input-557-77ff44433c64>:11: UserWarning: FixedFormatter should only be used together with FixedLocator
  ax.set_xticklabels(labs)
[557]: Text(0, 0.5, '%')
```



Código 29: Visualizando graficamente a distribuição percentual da variância.

A continuação da análise dos dados será fazer a análise espectral das séries temporais, o que será particularmente didático para ver os sinais originais e o que acontece com os sinais decompostos. Faremos em um outro notebook (e tutorial), uma vez que este já está bem extenso. Então, para encerrarmos, vamos preservar (pickle) o que já fizemos que é o tempo (tempo_ref_2) e os todos os níveis (g_niveis_2). Eu dei uma investigada superficial, mas creio que o jeito mais

direto de fazer isso é criar uma nova lista para colocar estes dois objetos para depois usar o 'pickle' (Código 30).

```
dados_ibge_processados = [tempo_ref_2, g_niveis_2]

with open('Ibge_base_dados_nivel2.pik', 'wb') as file:

    pickle.dump(dados_ibge_processados, file)
```

Código 30: Preservando (salvando!) dados.

Bônus:

Eu já tinha considerado 'encerrado' este notebook, mas decidi que a geração de tabelas numéricas para apresentação dos resultados é importante. Acima eu mencionei que isso é melhor feito em um processador de texto, o que de fato é, mas dá para gerar tabelas razoavelmente decentes para um relatório ou publicação usando o pacote pandas. Ainda preciso estudar melhor como formatar e melhorar a coisa, mas segue abaixo as tabelas das constantes harmônicas (Tabela 1) e da variância decomposta das estações (Tabela 2). O código para fazer está no notebook completo anexo.

Tabela 1: Principais constituintes harmônicos de maré para as estações. (A) amplitude, em m; (g) fase em graus.

	IMB(A)	IMB(g)	ARC(A)	ARC(g)	SAL(A)	SAL(g)	FOR(A)	FOR(g)	SAN(A)	SAN(g)
MSF	0.05	69.0	0.04	37.0	0.03	276.0	0.03	311.0	0.26	306.0
O1	0.08	230.0	0.08	244.0	0.04	277.0	0.05	338.0	0.05	144.0
K1	0.04	196.0	0.04	212.0	0.03	288.0	0.04	285.0	0.04	59.0
M2	0.14	264.0	0.33	281.0	0.80	317.0	0.96	337.0	1.22	322.0
S2	0.13	168.0	0.20	190.0	0.37	231.0	0.38	256.0	0.29	253.0
M4	0.04	56.0	0.03	73.0	0.02	315.0	0.01	44.0	0.27	213.0

Tabela 2: Percentual de variância para componente do nível da água.

	Sinal harmônico	Baixa frequência	Sinal não harmônico(30-2)	Alta frequência
IMB	51.18	42.33	6.08	0.41
ARC	88.95	5.79	4.78	0.48
SAL	97.23	0.23	2.53	0.01
FOR	97.01	0.19	2.76	0.04
SAN	93.94	2.69	3.36	0.01

Notebook completo funcional:

arquivo pdf: Notebook_IBGE_Mare_3.pdf