Prelucrarea semnalului vocal folosind python™

noțiuni fundamentale

Adriana STAN
Mircea GIURGIU

UTPRESS
Cluj-Napoca, 2021
ISBN 978-606-737-502-2

Adriana STAN Mircea GIURGIU

Prelucrarea semnalului vocal folosind

Noțiuni fundamentale

The Legrand Orange Book, LaTeX Template, Version 2.0 (9/2/15)
Sursa: http://www.LaTeXTemplates.com.

Prima ediție, 2021

Cuvânt înainte

Primul volum din *Prelucrarea semnalului vocal folosind Python* își propune să introducă o serie de noțiuni fundamentale de prelucrare a semnalului vocal folosind limbajul de programare Python și mediul interactiv Jupyter. Cartea este organizată în 8 tutoriale ce abordează teme individuale și nu necesită cunoștințe anterioare de programare sau prelucrare de semnal vocal. Temele mai avansate din ultimele tutoriale necesită, însă, o anumită familiaritate cu noțiunile de prelucrare a semnalelor discrete.

Volumul are asociată o pagină web în cadrul căreia se regăsesc tutorialele în format electronic alături de resursele media necesare rulării acestora:

www.speech.utcluj.ro/python-speech-book/

Volumul nu își propune să fie o resursă exhaustivă de noțiuni teoretice și programatice ale acestui domeniu, ci mai degrabă un punct de pornire. Totodată, codul asociat noțiunilor de prelucrare a semnalului vocal este menținut la un nivel de abstractizare minim, astfel încât utilizatorul să poată înțelege în detaliu procesele implementate. În cadrul fiecărui tutorial se regăsesc și o serie de exerciții a căror rezolvare vă invităm să o abordați înainte de a utiliza soluțiile disponibile online. Cei ce doresc să aprofundeze conceptele abordate, au la dispoziție o serie de referințe bibliografice indexate la finalul fiecărui tutorial.

Autorii doresc să mulțumească tuturor celor ce au făcut posibilă elaborarea și publicarea acestui volum.

Autorii
Cluj-Napoca, 2021

Cuprins

T1 Introducere în Jupyter și Python

În acest prim tutorial ne vom familiariza cu principalele caracteristici ale limbajului de programare Python și a mediului Jupyter.

Tutorialul acoperă doar funcționalitățile de bază ale limbajului de programare Python și a mediului Jupyter. Documentația completă a acestora o puteți accesa în paginile web oficiale:

- `https://www.python.org/doc/`
- `https://jupyter-notebook.readthedocs.io/en/5.7.4/`

T1.1. Introducere în Jupyter

Proiectul Jupyter este un mediu de programare ce permite utilizatorilor să editeze și să execute secvențe de cod scrise în limbajul Python în mod *interactiv*. Jupyter rulează un server local și poate fi accesat prin intermediul browserului.

Datorită flexibilității și ușurinței de utilizare, acesta a devenit principalul mediu de codare pentru Python, în special în aplicații de dezvoltare a algoritmilor de inteligență artificială și învățare automată. Drept urmare, Google pune la dispoziția utilizatorilor săi o versiune proprie a proiectului Jupyter combinat cu acces la resurse computaționale, sub denumirea de **Google Colab**: https://colab.research.google.com și care permite totodată conectarea la Google Drive pentru stocarea datelor.

Toate tutorialele din cadrul acestei cărți pot fi rulate atât în serverul local cât și în Colab. Pentru a instala o suită completă de unelte software necesare utilizării Python și Jupyter pe mașina locală, se recomandă utilizarea framework-ului Anaconda: https://www.anaconda.com/. Este de menționat faptul că și Google Colab permite conectarea la resursele mașinii locale. Totodată, pentru o testare rapidă a codului se poate utiliza și site-ul https://try.jupyter.org, însă datele nu pot fi salvate persistent pe server.

Organizarea codului în cadrul mediului Jupyter se face prin intermediul așa-numitelor *notebooks* (ro. *caiete*). Fiecare notebook reprezintă un mediu de sine stătător. În cadrul notebook-ului, codul este organizat în *celule* (en. *cell*). Fiecare celulă poate fi rulată individual și poate conține secvențe de cod de lungimi diferite, funcții, clase, etc. Celulele Jupyter sunt incluse însă în același domeniu de vizibilitate sau *namespace*. Acest lucru înseamnă că definirea unei variabile sau a unei funcții într-o celulă va face ca aceasta să fie disponibilă și în celelalte celule din cadrul aceluiași notebook. Trebuie să subliniem faptul că execuția codului trebuie să fie realizată secvențial. În sensul că, dacă dorim ca o variabilă sau funcție să fie definită, va trebui să executăm mai întâi celula ce conține instrucțiunile necesare și doar mai apoi să executăm celule ce modifică sau utilizează aceste variabile sau funcții.

Pe lângă celulele ce conțin cod, o altă funcționalitate a Jupyter este cea de celule text în cadrul cărora pot fi inserate comentarii sau explicații

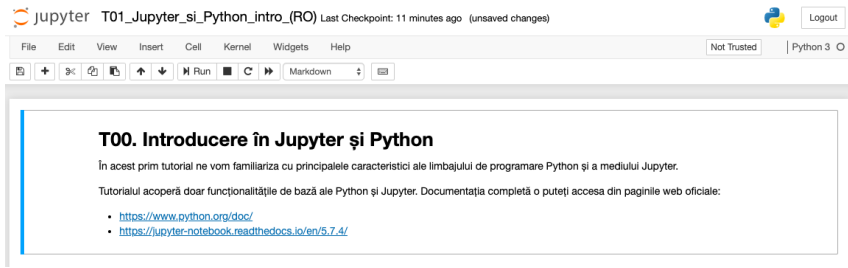


Fig.T1.1. Mediul Jupyter local

suplimentare referitoare la codul Python. Editarea textului este flexibilă și folosește notația Markdown: `https://www.markdownguide.org/`.

T1.1.1 Mediul Jupyter local

Mediul Jupyter local poate fi lansat din Anaconda și se va deschide automat într-o fereastră nouă a navigatorului web implicit. În partea de sus a ferestrei există un rând de opțiuni de meniu (`File`, `Edit`, `View`, `Insert`, ...) și un rând de iconițe cu unelte (*dischetă, semnul plus, foarfece, fișiere,* etc.). Acestea pot fi utilizate pentru manipularea celulelor Jupyter și a codului sau a comentariilor din cadrul acestora. Cele mai importante funcții sunt:

Inserare și ștergere celule

- Iconița cu semnul plus [+] - inserează o nouă celulă sub cea curentă. Tipul celulei (cod sau text) poate fi modificat din meniul `Cell` -> `Cell Type`.
- `Insert` -> `Insert Cell Above` - pentru a insera o celulă deasupra celei curente.
- `Edit` -> `Delete cells` - șterge celula curentă. Dacă sunt selectate mai multe celule, le șterge pe toate.

Ștergere output celule

- `Kernel` -> `Restart` din meniu pentru a restarta mediul de lucru. Atenție! Se vor șterge toate variabilele, definițiile de funcții și clase. Toate celulele vor trebui executate din nou.
- `Kernel` -> `Restart and clear output` pentru a restarta notebook-ul și a șterge datele de execuție ale celulelor (output-ul).

Salvare notebook

- Toate notebook-urile sunt salvate automat în directorul unde este a fost instalat Anaconda sau conform configurațiilor explicite ulterioare.

- Dacă doriți salvarea într-un alt director, navigați la `File` -> `Download as` -> `IPython Notebook (.ipynb)`

Comenzi rapide (shortcuts)

- În al doilea rând de comenzi din bara de meniu, butonul sub formă de tastatură va afișa lista de comenzi rapide ce poate fi utilizată în notebook.

T1.2. Introducere în Python

Python este un limbaj de programare de nivel înalt, cu scop general, de tip interpretor, ce pune accent pe ușurința de citire a codului. Lizibilitatea este dată de faptul că identarea codului este obligatorie și de faptul că nu există un simbol specific obligatoriu pentru marcarea sfârșitului instrucțiunilor simple sau compuse.

Un alt aspect important al limbajului Python este faptul că tipurile de date sunt determinate dinamic. Acest lucru înseamnă ca nu este necesară declararea tipului unei variabile înainte a fi utilizată. Cu toate acestea, Python nu permite utilizarea operațiilor ce nu sunt corect definite asupra datelor (de exemplu adunarea unui număr la un o variabilă de tip string). O altă particularitate este dată de faptul că toate variabilele sunt obiecte și nu există tipuri de date primitive. Din punct de vedere al managementului memoriei, Python include un mecanism automat și dinamic. Totodată, permite utilizarea a mai multe paradigme programatice, precum cea orientată pe obiecte, imperativă, funcțională și procedurală. Din punct de vedere al suportului și reutilizării codului, limbajul Python dispune de o bibliotecă standard extinsă.

Datorită simplității și a facilităților oferite, Python a devenit în ultimii ani unul dintre cele mai utilizate limbaje de programare în domeniul inteligenței artificiale și a prelucrărilor numerice.

Secțiunile următoare vor indexa cele mai importante operații și instrucțiuni Python alături de exemple specifice. Pentru o listă completă a capabilităților limbajului de programare, accesați documentația oficială: `https://www.python.org/doc/`

T1.2.1 Obiecte, tipuri de date și variabile

În limbajul Python toate datele sunt **obiecte** și fiecare obiect are un **tip**. Printre tipurile de bază se numără:

- `int` (integer; număr întreg fără cifre zecimale)
  - ex. `10`, `-3`

- `float` (float; număr real cu zecimale)
  - ex. `7.41`, `-0.006`
- `str` (string; șir de caractere ce poate fi încadrat de apostrof, ghilimele sau ghilimele triple)
  - `'acesta este un string incadrat de apostrof'`
  - `"acesta este un string incadrat de ghilimele"`
  - `'''acesta este un string incadrat de trei semne de apostrof'''`
  - `"""acesta este un string incadrat de trei semne de ghilimele"""`
- `bool` (boolean; valoare binară ce poate lua valorile True sau False)
  - `True`, `False`
- `NoneType` (tip special de date ce marchează lipsa unei valori)
  - `None`

În Python, o **variabilă** este numele dat în cod unui **obiect** specific, unei **instanțe** de obiect sau unei valori. Prin definirea variabilelor, datele pot fi referite într-un limbaj mai apropiat de înțelegerea utilizatorului. **Numele variabilelor** poate conține doar litere, simbolul underscore (_) sau numere (fără spații, cratime, sau alte caractere). Numele variabilelor trebuie să înceapă cu o literă sau simbolul *underscore* și nu se recomandă utilizarea literelor mari.

- ex. `adriana`, `adriana_stan`, `adriana123`, `a_stan_123`, `_adriana`

Convenția de notare (`https://www.python.org/dev/peps/pep-0008/`) a variabilelor este de a separa cuvintele prin *underscore*. Spre deosebire de alte limbaje unde se folosește preponderent notația de tip `camelcase`, de exemplu Java: `adrianaStan`.

T1.2.2 Operatori

Operatorii sunt simboluri speciale ce operează asupra diferitelor valori din cod. Printre operatorii de bază se numără:

- operatori aritmetici
  - `+` (adunare)
  - `-` (scădere)
  - `*` (înmulțire)

  - / (împărțire)
  - ** (exponent, ridicare la putere)
- operatori de atribuire
  - = (atribuirea unei valori)
  - += (adunare și reatribuire)
  - -= (scădere și reatribuire)
  - *= (înmulțire și reatribuire)
- operatori relaționali (returnează `True` sau `False`)
  - == (egalitate)
  - != (inegalitate)
  - < (mai mic)
  - <= (mai mic sau egal)
  - > (mai mare)
  - >= (mai mare sau egal)

În cadrul expresiilor compuse, **precedența operatorilor** determină ordinea de evaluare a operațiilor. Operatorii cu prioritatea mai mare sunt evaluați prima dată. Operatorii cu prioritate egală sunt evaluați de la stânga la dreapta. Precedența operatorilor poate fi modificată prin utilizarea parantezelor `()`. Ordinea completă a precedenței operatorilor poate fi regăsită la adresa: `https://docs.python.org/3/reference/expressions.html#operator-precedence`.

> **OBS T1.1** În Jupyter, dacă ultima instrucțiune din secvența de cod dintr-o celulă returnează o valoare ce nu este atribuită unei variabile, aceasta va fi afișată automat. În caz contrar, rezultatul poate fi afișat cu ajutorul funcției `print()`. Afișarea automată poate fi dezactivată folosind simbolul punct și virgulă ; la finalul ultimei instrucțiuni.

Exemple de utilizare a operatorilor:

[1]:
```
# Atribuire
num1 = 10
num2 = -3
num3 = 7.41
num4 = -.6
num5 = 7
num6 = 3
num7 = 11.11
```

[2]:
```
# Adunare
num1 + num2
```

[2]:
```
7
```

[3]:
```
# Scădere
num2 - num3
```

[3]:
```
-10.41
```

[4]:
```
# Înmulțire
num3 * num4
```

[4]:
```
-4.446
```

[5]:
```
# Împărțire
num4 / num5
```

[5]:
```
-0.08571428571428572
```

[6]:
```
# Exponent
num5 ** num6
```

[6]:
```
343
```

[7]:
```
# Incrementare variabilă existentă
num7 += 4
print (num7)
```

[7]:
```
15.11
```

[8]:
```
# Decrementare variabilă existentă
num6 -= 2
print (num6)
```

[8]:
```
1
```

[9]:
```
# Înmulțire și reatribuire
num3 *= 5
print (num3)
```

[9]:
```
37.05
```

[10]:
```
# Atribuirea rezultatului unei expresii unei variabile
num8 = num1 + num2 * num3
print (num8)
```

[10]:
```
-101.15
```

[11]:
```
# Verificare egalitate
num1 + num2 == num5
```

[11]:
```
True
```

[12]:
```
# Verificare inegalitate
num3 != num4
```

[12]:
```
True
```

[13]:
```
# Mai mic
num5 < num6
```

[13]:
```
False
```

[14]:
```
# Expresie relațională compusă
5 > 3 > 1
```

[14]:
```
True
```

[15]:
```
# Expresie relațională compusă
5 > 3 < 4 == 3 + 1
```

[15]:
```
True
```

[16]:
```
# Atribuire sir de caractere
string1 = 'un exemplu'
string2 = "mere si pere "
print (string1)
print (string2)
```

[16]:
```
un exemplu
mere si pere
```

[17]:
```
# Adunare siruri de caractere
string1 + ' de utilizare a operatorului +'
```

[17]:
```
'un exemplu de utilizarea a operatorului +'
```

[18]:
```
# Sirul de caractere inițial nu a fost modificat
string1
```

[18]:
```
'un exemplu'
```

[19]:
```
# Înmulțire sir de caractere cu o valoare numerica
string2 * 3
```

[19]:
```
'mere si pere mere si pere mere si pere'
```

[20]:
```
# Nici acest string nu a fost modificat
string2
```

[20]:
```
'mere si pere '
```

[21]:
```
# Verificare egalitate siruri de caractere
string1 == string2
```

[21]:
```
False
```

[22]:
```
# Sunt egale sirurile de caractere?
string1 == 'un examplu'
```

[22]:
```
True
```

[23]:
```
# Adunare și reatribuire
string1 += ' ce modifica sirul de caractere initial'
print (string1)
```

[23]:
```
un exemplu ce modifica sirul de caractere initial
```

[24]:
```
# Înmulțire și reatribuire
string2 *= 3
print (string2)
```

[24]:
```
mere si pere mere si pere mere si pere
```

OBS T1.2 Scăderea, împărțirea și decrementarea nu se aplică șirurilor de caractere.

OBS T1.3 În Python nu există operator ternar, dar poate fi înlocuit cu o instrucțiune `if` scrisă într-o singură linie: `a if (a>=b) else b`

T1.2.3 Containere de bază

Containerele sunt obiecte ce pot fi utilizate pentru a grupa alte obiecte ce nu trebuie să aibă același tip de bază. Containerele de bază în Python includ:

- `str` (string/șir de caractere: invariabil; indexat prin întregi; elementele sunt stocate în ordinea în care au fost adăugate în container)
- `list` (listă: variabil; indexat prin întregi; elementele sunt stocate în ordinea în care au fost adăugate în container)
  ex. `[3, 5, 6, 3, 'caine', 'pisica', False]`
- `tuple` (tuplu: invariabil; indexat prin întregi; elementele sunt stocate în ordinea în care au fost adăugate în container)
  ex. `(3, 5, 6, 3, 'caine', 'pisica', False)`
- `set` (set: variabil; nu este indexat; elementele **NU** sunt stocate în ordinea în care au fost adăugate; poate conține doar obiecte immutable;

nu poate conține obiecte duplicat)
ex. `{3, 5, 6, 3, 'caine', 'pisica', False}`

- `dict` (dictionary: variabil; perechi cheie-valoare indexate de obiecte immutabile; elementele **NU** sunt stocate în ordinea în care au fost adăugate; cheile trebuie să fie unice)
ex. `{'nume': 'Ioana', 'varsta': 23, 'mancare_preferata': ['pizza', 'fructe', 'peste']}`

> **OBS T1.4** Obiectele **variabile (en. *mutable*)** pot fi modificate după ce au fost create, iar cele **invariabile (en. *immutable*)** nu pot fi modificate.

Definirea elementelor în liste, tupluri sau seturi se face cu ajutorul simbolului virgulă (,). Definirea dicționarelor folosește două puncte (:) pentru a separa cheia dicționarului de valoare, iar virgula (,) este folosită pentru a separa perechile din dicționar.

Șirurile de caractere, listele și tuplurile sunt de tip **secvență** și pot utiliza operatorii: +, *, +=, și *= .

[25]:
```
# Atribuirea containerelor
list1 = [3, 5, 6, 3, 'caine', 'pisica', False]
tuple1 = (3, 5, 6, 3, 'caine', 'pisica', False)
set1 = {3, 5, 6, 3, 'caine', 'pisica', False}
dict1 = {'nume': 'Ioana', 'varsta': 23, ' \
        mancare_preferata': ['pizza', 'fructe', 'peste']}
```

[26]:
```
# Elementele listei sunt stocate în ordinea în care
# au fost adăugate în container
print (list1)
```

[26]:
```
[3, 5, 6, 3, 'caine', 'pisica', False]
```

[27]:
```
# Elementele tuplului sunt stocate în ordinea în care
# au fost adăugate în container
print (tuple1)
```

[27]:
```
(3, 5, 6, 3, 'caine', 'pisica', False)
```

[28]:
```
# Elementele setului NU sunt stocate în ordinea în care
# au fost adăugate în container
# Totodată, valoarea 3 apare o singură dată în set
print (set1)
```

[28]:
```
set([False, 3, 5, 6, 'caine', 'pisica'])
```

[29]:
```
# Elementele dicționarului NU sunt neapărat stocate în
# ordinea în care au  fost adăugate în container
print (dict1)
```

[29]:
```
{'varsta': 23, 'mancare_preferata': ['pizza', 'fructe',␣
↪'peste'], 'nume': 'Ioana'}
```

[30]:
```
# Adăugare și reatribuire
list1 += [5, 'prune']
print (list1)
```

[30]:
```
[3, 5, 6, 3, 'caine', 'pisica', False, 5, 'prune']
```

[31]:
```
# Adăugare și reatribuire
tuple1 += (5, 'prune')
print (tuple1)
```

[31]:
```
(3, 5, 6, 3, 'caine', 'pisica', False, 5, 'prune')
```

[32]:
```
# Înmulțirea unei liste cu un scalar
[1, 2, 3, 4] * 2
```

[32]:
```
[1, 2, 3, 4, 1, 2, 3, 4]
```

[33]:
```
# Înmulțirea unui tuplu cu un scalar
(1, 2, 3, 4) * 3
```

[33]:
```
(1, 2, 3, 4, 1, 2, 3, 4, 1, 2, 3, 4)
```

Accesarea datelor din containere

Pentru containerele de tip secvență: șiruri de caractere, liste și dicționare, se poate utiliza notația de indexare `subscript` (cu paranteze drepte) pentru a accesa datele de la o anumită poziție:

- stringurile, listele, și tuplurile sunt indexate de întregi, **începând cu 0** pentru primul element
- aceste secvențe permit și accesarea unui domeniu de indecși, denumit **slicing**
- se poate utiliza **indexarea negativă** pentru a începe de la finalul secvenței
- dicționarele sunt indexate prin cheile lor și nu prin valori întregi.

OBS T1.5 Containerele de tip `set` nu sunt indexate, astfel că nu putem accesa elementele prin notația subscript (paranteze drepte)

[34]:
```
# Primul element dintr-o secvență
print (list1[0])
```

[34]:
```
3
```

[35]:
```
# Ultimul element dintr-o secvență
print (tuple1[-1])
```

[35]:
```
prune
```

[36]:
```
# Domeniu de elemente dintr-o secvență
print (string1[3:8])
```

[36]:
```
examp
```

[37]:
```
# Domeniu de elemente dintr-o secvență
print (tuple1[:-3])
```

[37]:
```
(3, 5, 6, 3, 'caine', 'pisica')
```

[38]:
```
# Domeniu de elemente dintr-o secvență
print (list1[4:])
```

[38]:
```
['caine', 'pisica', False, 5, 'prune']
```

[39]:
```
# Un element dintr-un dicționar indexat de cheie
print (dict1['name'])
```

[39]:
```
Ioana
```

[40]:
```
# Un element dintr-o secvență conținută ca valoare
# într-un dicționar
print (dict1['fav_foods'][2])
```

[40]:
```
peste
```

T1.2.4 Definirea funcțiilor

Funcțiile în Python, asemenea altor limbaje de programare trebuie să includă un set de *argumente de intrare*, o *secvență de instrucțiuni* și un set de *valori returnate*. Oricare dintre aceste elemente poate să lipsească. Definirea unei funcții se face cu ajutorul cuvântului cheie `def` urmat de numele funcției, lista parametrilor de intrare încadrați de paranteze rotunde și simbolul : ce marchează începutul unei secvențe de instrucțiuni compuse:

[41]:
```
# Definirea unei funcții
def sumation(a,b):
    c = a+b
```

```
    return c
```

[42]:
```
# Funcție fără parametri de intrare și fără valoare returnată
def print_func():
    print ("Functia mea")
```

Dacă o funcție returnează mai multe valori, acestea vor reprezenta un tuplu.

[43]:
```
# Funcție ce returnează mai multe valori
def sum_and_pow(a,b):
    return (a+b, a**b)
```

Apelul unei funcții se face prin numele acesteia urmat de variabilele de intrare încadrate de paranteze rotunde:

[44]:
```
# Apelul funcției
print (sumation(3,4))
print (print_func())
print (sum_and_pow(3,4))
```

[44]:
```
7
Functia mea
None
(7, 81)
```

În cazul în care o funcție returnează mai multe valori, acestea pot fi extrase în cadrul variabilelor în mod selectiv:

[45]:
```
# extragerea valorilor returnate în variabile individuale
a,b = sum_and_pow(2,3)
print (a,b)

# extragerea primei valori returnate
c,_ = sum_and_pow(3,2)
print(c)

# extragerea celei de-a doua valori returnate
_,d = sum_and_pow(3,2)
print(d)

# extragerea ambelor valori returnate într-un tuplu
t = sum_and_pow(3,2)
print (t)
```

```
[45]: (5, 8)
      5
      9
      (5, 9)
```

T1.2.4.1 Argumente poziționale și argumente cu cheie (`keyword`)

Argumentele unei funcții pot fi identificate prin poziția lor în apelul funcției (**poziționale**) sau prin utilizarea unui cuvânt cheie înainte de acestea (**keyword**).

În funcție de antetul și tipul argumentelor unei funcții, aceasta poate fi apelată în diverse moduri:

- `func()` - apel fără argumente
- `func(arg)` - apel cu un argument pozițional
- `func(arg1, arg2)` - apel cu două argumente poziționale
- `func(arg1, arg2, ..., argn)` - apel cu multiple argumente poziționale
- `func(kwarg=value)` - apel `func` cu un argument de tip keyword și valoare implicită
- `func(kwarg1=value1, kwarg2=value2)` - apel cu două argumente de tip keyword și valori implicite
- `func(kwarg1=value1, kwarg2=value2, ..., kwargn=valuen)` - apel cu multiple argumente de tip keyword
- `func(arg1, arg2, kwarg1=value1, kwarg2=value2)`- apel cu argumente poziționale și de tip keyword

OBS T1.6 Când se utilizeaază argumente **poziționale**, acestea trebuie transmise în ordinea în care au fost definite în funcție (**semnătura funcției**)

OBS T1.7 Când se utilizează argumente de tip **keyword**, acestea pot fi transmise în orice ordine, atâta timp cât se specifică numele argumentului.

OBS T1.8 Dacă se utilizează și argumente **poziționale** și argumente de tip **keyword**, argumentele poziționale trebuie să fie primele.

OBS T1.9 Argumentele de tip **keyword** pot să aibă valori implicite specificate în antetul funcției.

[46]:
```
# Definire funcție cu argumente pozitionale și keyword
# fără valoare implicită
def func1(a, b, inc=''):
    return a+b+inc
```

[47]:
```
# Apel funcție cu argumente poziționale și de tip keyword
func1(2, 3, inc=3)
```

[47]: 8

[48]:
```
# Definire funcție cu argumente pozitionale și keyword
# cu valoare implicită
def func2(a, b, inc=5):
    return a+b+inc
```

[49]:
```
# Apel funcție cu argumente poziționale și de tip keyword
# cu valoare implicită
func2(2, 3)
```

[49]: 10

[50]:
```
# Apel funcție cu argumente poziționale și de tip keyword
# cu valoare implicită
func2(2, 3, inc=6)
```

[50]: 11

T1.2.5 Clase: crearea obiectelor proprii

Prin paradigma obiectuală, Python permite definirea claselor proprii. Antetul acestora este dat de cuvântul cheie `class` urmat de numele clasei, iar între paranteze rotunde sunt enumerate clasele moștenite de clasa curentă.

OBS T1.10 Clasa de bază `object` nu trebuie specificată în clar

[51]:
```
# Definirea unei clase derivată explicit din clasa de bază
# Python - `object`
class Clasa_mea(object):
    my_property = 'Aceasta este clasa mea'
```

```
# Definirea unei clase derivată implicit din clasa de bază
# Python - `object`
class Clasa_mea():
    my_property = 'Aceasta este clasa mea'

# Definirea unei noi clase `Clasa_mea_derivata` derivată
# din tipul `Clasa_mea`
class Clasa_mea_derivata(Clasa_mea):
    my_property = 'Aceasta este clasa mea derivata'
```

Pentru a crea un obiect de tipul clasei definite se utilizează atribuirea simplă:

[52]:
```
# Crearea instanțelor de clasă
t = Clasa_mea()
d = Clasa_mea_derivata()
print (t)
print (d)
```

[52]:
```
<__main__.My_Class instance at 0x7f35c4643518>
<__main__.My_Dict_Class instance at 0x7f35c4643560>
```

T1.2.5.1 Atributele obiectelor (metode și proprietăți)

Fiecare tip de obiecte din Python are **atribute** diferite ce pot fi referite prin nume (similar cu variabilele). Pentru a accesa atributele unui obiect, se utilizează punctul (`.`) după numele obiectului și apoi atributul (ex. `obj.atribut`)

Când atributul unui obiect este apelabil, acel atribut este denumit **metodă**. Este similar cu o funcție, însă această funcție este limitată la obiectul ce o apelează.

Când atributul unui obiect nu este apelabil, acel atribut este denumit **proprietate**. Este doar o anumită informație a obiectului respectiv și este la rândul său un obiect.

Funcția predefinită `dir()` poate fi utilizată pentru a returna o listă a atributelor unui obiect.

[53]:
```
# Afisarea atributului obiectelor din clasa My_Class
print (t.my_property)
```

[53]:
```
Aceasta este clasa mea
```

[54]:
```
# Crearea unei clase cu diferite atribute și metode
class Clasa_mea2():
    varsta = 12
```

```
    inaltime = 175
    def get_varsta(self):
        return self.varsta
    def get_inaltime(self):
        return self.inaltime
```

[55]:
```
# Apel metode ale obiectelor My_Class2
o = Clasa_mea2()
print(o.get_varsta())
print (o.get_inaltime())
```

[55]:
```
12
175
```

[56]:
```
# Afișarea atributelor și metodelor unui obiect:
dir(Clasa_mea2)
```

[56]:
```
['__doc__', '__module__', 'varsta', 'get_varsta',␣
 ↪'get_inaltime',
'inaltime']
```

Se poate observa că există două atribute implicite: `__doc__` - reprezintă documentația clasei și `__module__` ce reprezintă modulul din care face parte (vom discuta ulterior despre acest concept).

[57]:
```
# Utilizarea atributului de documentație pentru un obiect
# al unei clase
class Clasa_mea2():
    '''Clasa_mea2 nu face nimic deocamdata'''
    varsta = 12
    inaltime = 175

# Nu este necesară instanțierea unui obiect pentru a apela
# atributele unei clase:
Clasa_mea2().__doc__
```

[57]:
```
'Clasa_mea2 nu face nimic deocamdata'
```

T1.2.5.2 Crearea metodelor de inițializare (constructor)

O metodă de inițializare este similară cu metoda constructor din Java și este utilizată pentru a inițializa atributele de clasă sau pentru a apela metode sau funcții specifice atunci când un nou obiect este creat.

Iar destructorul este definit astfel: - `__del__(self)`:

OBS T1.11 În metodele de clasă din Python, referința `self` asupra obiectului curent este obligatorie!!

[58]:
```
# Exemplu de clasă cu metodă de inițializare (constructor),
# metode proprii și destructor
class Student(object):
    def __init__(self, nume = "Maria", varsta ="27"):
        self.nume = nume
        self.varsta = varsta

    def print_name(self):
        print ("Numele meu este " + self.name+'!')

    def print_greeting(self, greet):
        print (greet+' ' + self.nume+'!')

    def print_name_age(self):
        print ('Numele meu este %s si am %d de ani.'  \
                %(self.nume, self.varsta))

    def __del__ (self):
        print ('Obiectul a fost distrus! Semnat, ' + self.
 ↪nume)
```

[59]:
```
# objA va folosi parametrii impliciti ai constructorului
objA = Student()
# objB și objC vor folosi valorile proprii pentru atribute
objB = Student(name="Dan")
objC = Student(name="Vlad", age=20)

objA.print_nume()
objB.print_greeting("Salut")
objC.print_name_age()

# Ștergerea obiectului implică apelarea implicită
# a destructorului
del objA
```

[59]:
```
Numele meu este Maria!
Salyt Dan!
Numele meu este Vlad și am 20 de ani.
```

```
Obiectul a fost distrus! Semnat, Maria
```

T1.2.5.3 Conversia de tip (casting)

După cum am menționat anterior, toate datele în Python sunt stocate în obiecte fără a avea la dispoziție tipuri de date primitive.

[60]:
```
# Afișarea atributelor și metodelor unui obiect de tip int()
dir(int)
```

[60]:
```
['__abs__',
 '__add__',
 '__and__',
 '__class__',
 '__cmp__',
 '__coerce__',
 '__delattr__',
 '__div__',
 '__divmod__',
 '__doc__',
 '__float__',
 '__floordiv__',
 '__format__',
 '__getattribute__',
 '__getnewargs__',
 '__hash__',
 '__hex__',
 '__index__',
 '__init__',
 '__int__',
 '__invert__',
 '__long__',
 '__lshift__',
 '__mod__',
 '__mul__',
 '__neg__',
 '__new__',
 '__nonzero__',
 '__oct__',
 '__or__',
 '__pos__',
 '__pow__',
```

```
'__radd__',
'__rand__',
'__rdiv__',
'__rdivmod__',
'__reduce__',
'__reduce_ex__',
'__repr__',
'__rfloordiv__',
'__rlshift__',
'__rmod__',
'__rmul__',
'__ror__',
'__rpow__',
'__rrshift__',
'__rshift__',
'__rsub__',
'__rtruediv__',
'__rxor__',
'__setattr__',
'__sizeof__',
'__str__',
'__sub__',
'__subclasshook__',
'__truediv__',
'__trunc__',
'__xor__',
'bit_length',
'conjugate',
'denominator',
'imag',
'numerator',
'real']
```

Astfel că toate tipurile și containerele de bază pe care le-am folosit până acum au definite și **metode constructor**:

- `int()`
- `float()`
- `str()`
- `list()`
- `tuple()`
- `set()`

- `dict()`

Acestea pot să fie utilizate pentru a face inițializarea unui obiect de tipul celui definit pe baza unor variabile sau parametri de intrare. Aceste metode pot fi considerate ca fiind și metode de cast sau conversie explicită de tip. Trebuie să avem grijă însă să nu pierdem informație în cadrul acestor conversii.

[61]:
```
# Inițializare explicită obiect de tip int
a = int(12)
print (a)

# Nu diferă programatic cu nimic față de:
b = 12
print (b)
```

[61]:
```
12
12
```

[62]:
```
# Conversie string la int
int("23")
```

[62]:
```
23
```

[63]:
```
# Conversie float la int cu pierdere de informație
int(23.4)
```

[63]:
```
23
```

[64]:
```
# Conversie listă la set
my_list = [1,2,3,3,4,4,5,6,7,7]
set(my_list)
```

[64]:
```
{1, 2, 3, 4, 5, 6, 7}
```

T1.2.6 Funcții și metode predefinite

Limbajul Python conține un set predefinit de funcții ce facilitează lucrul cu obiecte sau mediul de programare. O listă scurtă a celor mai importante funcții este redată mai jos:

- `type(obj)` determină tipul unui obiect
- `len(container)` determină numărul de elemente dintr-un container
- `callable(obj)` determină dacă un obiect este apelabil (funcție)
- `sorted(container)` returnează o listă ordonată a elementelor dintr-un container

- `sum(container)` returnează suma elementelor dintr-un container dcu valori numerice
- `min(container)` returnează cel mai mic element dintr-un container
- `max(container)` returnează cel mai mare element dintr-un container
- `abs(number)` returnează modulul unui număr
- `repr(obj)` returnează reprezentarea string a unui obiect

  Lista completă a funcțiilor predefinite: `https://docs.python.org/3/library/functions.html`

Aceste funcții pot fi aplicate asupra oricărui obiect de tipul specificat ca parametru de intrare. Funcții specifice claselor individuale vor fi indexate ulterior.

[65]:
```
# Tipul obiectului
type(string1)
```

[65]:
```
str
```

[66]:
```
# Numărul de elemente dintr-un container de tip dicționar
len(dict1)
```

[66]:
```
3
```

[67]:
```
# Numărul de elemente dintr-un container de tip string
len(string2)
```

[67]:
```
57
```

[68]:
```
# Este un obiect apelabil?
callable(len) # funcție/metodă
```

[68]:
```
True
```

[69]:
```
# Este un obiect apelabil?
callable(dict1) # instanță de clasă
```

[69]:
```
False
```

[70]:
```
# Funcția sorted() pentru a returna o nouă listă ordonată
# cu elementele din container
sorted([10, 1, 3.6, 7, 5, 2, -3])
```

[70]:
```
[-3, 1, 2, 3.6, 5, 7, 10]
```

[71]:
```
# Funcția sorted() pentru a returna o nouă listă ordonată cu
# elementele din container
# - remarcați faptul că literele majuscule sunt
# primele (vezi cod ASCII)
```

```
sorted(['dogs', 'cats', 'zebras', 'Chicago', 'California',\
        'ants', 'mice'])
```

[71]:
```
['California', 'Chicago', 'ants', 'cats', 'dogs', 'mice',
'zebras']
```

[72]:
```
# Suma elementelor conținute în container
sum([10, 1, 3.6, 7, 5, 2, -3])
```

[72]:
```
25.6
```

[73]:
```
# Minimul elementelor conținute în container
min([10, 1, 3.6, 7, 5, 2, -3])
```

[73]:
```
-3
```

[74]:
```
# Minimul elementelor conținute în container
min(['g', 'z', 'a', 'y'])
```

[74]:
```
'a'
```

[75]:
```
# Maximul elementelor conținute în container
max([10, 1, 3.6, 7, 5, 2, -3])
```

[75]:
```
10
```

[76]:
```
# Minimul elementelor conținute în container
min('gibberish')
```

[76]:
```
'b'
```

[77]:
```
# Modulul numărului:
abs(-10)
```

[77]:
```
10
```

[78]:
```
# Conversia în șir de caractere a obiectului
repr(set1)
```

[78]:
```
"set([False, 3, 5, 6, 'dog', 'cat'])"
```

T1.2.6.1 Metode specifice ale obiectelor de tip string

- `.capitalize()` retunează stringul cu prima literă transformată în majusculă
- `.upper()` returnează stringul cu toate literele transformate în majuscule
- `.lower()` returnează stringul cu toate literele transformate în litere minuscule

- `.count(substring)` returnează numărul de apariții ale substringului în string
- `.startswith(substring)` determină dacă stringul începe cu substringul dat ca argument
- `.endswith(substring)` determină dacă stringul se termină cu substringul dat ca argument
- `.replace(old, new)` retunează o copie a stringului original în care aparițiile stringului `old` sunt înlocuite cu `new`

[79]:
```
# Atribuie un string la o variabilă
a_string = 'tHis is a sTriNg'
```

[80]:
```
# Versiunea capitalizată (prima literă majusculă)
# a stringului
a_string.capitalize()
```

[80]: `'This is a string'`

[81]:
```
# Versiunea cu litere majuscule a stringului
a_string.upper()
```

[81]: `'THIS IS A STRING'`

[82]:
```
# Versiunea cu litere minuscule a stringului
a_string.lower()
```

[82]: `'this is a string'`

[83]:
```
# Stringul inițial nu este modificat
a_string
```

[83]: `'tHis is a sTriNg'`

[84]:
```
# Numără aparițiile substringului în string
a_string.count('i')
```

[84]: `3`

[85]:
```
# Numără aparițiile substringului în string începând
# cu o anumită poziție
a_string.count('i', 7)
```

[85]: `1`

[86]:
```
# Numără aparițiile substringului în string
a_string.count('is')
```

[86]: `2`

[87]:
```
# Verifică dacă stringul începe cu this
a_string.startswith('this')
```

[87]:
```
False
```

Metodele pot fi înlănțuite atâta timp cât rezultatul metodei anterioare este de tipul metodei curente:

[88]:
```
# Verifică dacă stringul în format cu litere minuscule
# începe cu this
a_string.lower().startswith('this')
```

[88]:
```
True
```

[89]:
```
# Verifică dacă stringul se termină 'Ng'
a_string.endswith('Ng')
```

[89]:
```
True
```

[90]:
```
# Returnează o copie a stringului în care "is" este înlocuit
# cu "XYZ"
a_string.replace('is', 'XYZ')
```

[90]:
```
'tHXYZ XYZ a sTriNg'
```

[91]:
```
# Returnează o copie a stringului în care "i" este
# înlocuit cu "!"
a_string.replace('i', '!')
```

[91]:
```
'tH!s !s a sTr!Ng'
```

[92]:
```
# Returnează o versiune a stringului în care primele două
# apariții ale "i" sunt înlocuite cu "!"
a_string.replace('i', '!', 2)
```

[92]:
```
'tH!s !s a sTriNg'
```

T1.2.6.2 Metode specifice ale obiectelor de tip listă

- `.append(item)` adaugă un singur element în listă
- `.extend([item1, item2, ...])` adaugă mei multe elemente în listă
- `.remove(item)` șterge un element din listă
- `.pop()` șterge și returnează ultimul element din listă
- `.pop(index)` șterge și returnează un element de pe poziția dată

[93]:
```
# Definire listă
my_list = ['a', 'b', 'c']
```

[94]:
```
# Adaugă un element la listă
my_list.append('d')
print(my_list)
```

[94]:
```
['a', 'b', 'c', 'd']
```

[95]:
```
# Adaugă două elemente la finalul listei
my_list.extend(['e','f'])
print(my_list)
```

[95]:
```
['a', 'b', 'c', 'd', 'e', 'f']
```

[96]:
```
# Șterge elementul "a"
my_list.remove('a')
print(my_list)
```

[96]:
```
['b', 'c', 'd', 'e', 'f']
```

[97]:
```
# Returnează și șterge ultimul element din listă
my_list.pop()
```

[97]:
```
'f'
```

[98]:
```
# Șterge un element din listă de pe poziția dată
my_list.pop(3)
```

[98]:
```
'e'
```

T1.2.6.3 Metode specifice ale obiectelor de tip set

- `.add(item)` adaugă un singur element în set
- `.update([item1, item2, ...])` adaugă mai multe elemente în set
- `.update(set2, set3, ...)` adaugă toate elementele din toate seturile date în setul inițial
- `.remove(item)` elimină un singur element din set
- `.pop()` șterge și returnează un singur element aleator din set
- `.difference(set2)` returnează diferența dintre setul inițial și cel dat ca parametru
- `.intersection(set2)` returnează elementele comune celor două seturi
- `.union(set2)` returnează elementele comune și necomune celor două seturi (reuniunea)
- `.symmetric_difference(set2)` returnează elementele ce sunt doar într-un set (nu în ambele)
- `.issuperset(set2)` setul inițial conține toate elementele setului dat ca parametru?

- `.issubset(set2)` este setul inițial un subset al setului dat?

[99]:
```
# Definire set
my_set = {1,2,3}
print(my_set)
```

[99]:
```
set([1, 2, 3])
```

[100]:
```
# Adaugă un element
my_set.add(4)
print(my_set)
```

[100]:
```
set([1, 2, 3, 4])
```

[101]:
```
#adaugă elemente multiple la set
my_set.update([3,4,5,6])
print(my_set)
```

[101]:
```
set([1, 2, 3, 4, 5, 6])
```

[102]:
```
# Adaugă elementele din alte două seturi
my_second_set = {7,8,9}
my_third_set = {10,11}
my_set.update(my_second_set, my_third_set)
print(my_set)
```

[102]:
```
set([1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11])
```

T1.2.6.4 Metode specifice ale obiectelor de tip dicționar

- `.update([(key1, val1), (key2, val2), ...])` adaugă mai multe perechi cheie-valoare la dict
- `.update(dict2)` adaugă toate elementele unui alt dicționar
- `.pop(key)` șterge cheia și returnează valoarea asociată (eroare dacă nu există cheia)
- `.pop(key, default_val)` șterge cheia și returnează valoarea asociată (sau valoarea default dacă nu există cheia)
- `.get(key)` returnează valoarea asociată unei anumite chei (sau None dacă nu există cheia)
- `.get(key, default_val)` returnează valoarea asociată unei chei (sau default_val dacă nu există cheia)
- `.keys()` returnează lista cheilor din dicționar
- `.values()` returnează lista valorilor din dicționar
- `.items()` returnează perechile cheie-valoare din dicționar

[103]:
```
# Definire dicționar
my_dict = {1:"Ana", 2:"Maria", 3:"Dan", 4:"Vlad"}
```

[104]:
```
# Adaugă chei și valori în mod explicit
my_dict.update([(5,"Elena"), (6, "Alex")])
print(my_dict)
```

[104]:
```
{1: 'Ana', 2: 'Maria', 3: 'Dan', 4: 'Vlad', 5: 'Elena',
6: 'Alex'}
```

[105]:
```
# Definim al doilea dicționar
my_second_dict = {7:"Dan", 8:"Roxana"}
# La update doar cheile trebuie să fie diferite
my_dict.update(my_second_dict)
my_dict
```

[105]:
```
{1: 'Ana',
 2: 'Maria',
 3: 'Dan',
 4: 'Vlad',
 5: 'Elena',
 6: 'Alex',
 7: 'Dan',
 8: 'Roxana'}
```

[106]:
```
# Returnează valoarea asociată cheii 3
my_dict.get(3)
```

[106]:
```
'Dan'
```

[107]:
```
# Returnează lista cheilor
my_dict.keys()
```

[107]:
```
[1, 2, 3, 4, 5, 6, 7, 8]
```

[108]:
```
# returnează lista valorilor
my_dict.values()
```

[108]:
```
['Ana', 'Maria', 'Dan', 'Vlad', 'Elena', 'Alex', 'Dan',
'Roxana']
```

[109]:
```
# Returnează lista perechilor cheie-valoare
my_dict.items()
```

[109]:
```
[(1, 'Ana'),
 (2, 'Maria'),
 (3, 'Dan'),
```

```
 (4, 'Vlad'),
 (5, 'Elena'),
 (6, 'Alex'),
 (7, 'Dan'),
 (8, 'Roxana')]
```

[110]:
```
# Verifică dacă o cheie există în dicționar
3 in my_dict
```

[110]:
```
True
```

T1.2.7 Afișare, formatare stringuri, specificatori de tip

Afișarea în Python se face în mod similar cu cea din C, unde **specificatorii de format** sunt utilizați pentru a afișa variabilele într-un mod predefinit:

- %s - string (sau orice alt obiect cu reprezentare String)
- %d - întregi
- %f - valori reale
- %.<number of digits>f - valori reale cu un număr specific de cifre zecimale
- %x/%X - întregi în reprezentare hexa (lowercase/uppercase)

[111]:
```
nume = "Maria"
varsta = 21
```

[112]:
```
print ("Salut! Numele meu este %s" %nume)
```

[112]:
```
Salut! Numele meu este Maria
```

[113]:
```
print ("Salut! Numele meu este %s si am %d de ani."  \
       %(nume, varsta))
```

[113]:
```
Salut! Numele meu este Maria si am 21 de ani.
```

Alternativ, se poate utiliza concatenarea de stringuri sau funcția `print()` cu argumente multiple:

[114]:
```
print ("Salut! Numele meu este" + nume + " si am "+␣
 ↪str(varsta) +" de ani.")
```

[114]:
```
Salut! Numele meu este Maria si am 21 de ani.
```

[115]:
```
print ("Salut! Numele meu este ", nume, " si am ", varsta , \
       " de ani.")
```

[115]:
```
'Salut! Numele meu este Maria si am 21 de ani.'
```

T1.2.8 Instrucțiuni de control

Până în acest moment, am discutat doar despre datele disponibile în limbajul Python. În continuare vom introduce o serie de instrucțiuni necesare procesării acestor date, denumite `instrucțiuni de control`.

T1.2.8.1 Instrucțiunea ciclică `while`

Instrucțiunea ciclică `while` repetă execuția unui set de instrucțiuni atât timp cât condiția inițială este adevărată.

> Notă: Există posibilitatea iterării infinite în cadrul instrucțiunii `while`, dacă expresia condițională nu devine `False`. Este necesară modificarea variabilei de test în interiorul instrucțiunii.

[116]:
```
# Instrucțiune while ce decrementează o variabilă
i = 5
while i > 0:
    print (i)
    i-=1
```

[116]:
```
5
4
3
2
1
```

[117]:
```
# Instrucțiune while ce parcurge un string
s = 'abcd'
while s:
    print (s)
    s = s[1:]
```

[117]:
```
abcd
bcd
cd
d
```

Instrucțiunea `while` în Python permite utilizarea unei ramuri de `else` ce se execută atunci când se iese din `while` în mod normal, fără utilizarea unor instrucțiuni de salt de tipul `break` (vor fi discutate ulterior):

[118]:
```
# Ieșirea pe ramura else din while
i = 5
while i>0:
    print (i)
```

```
    i-=2
else:
    print ("Normal exit")
```

[118]:
```
5
3
1
Normal exit
```

T1.2.8.2 Instrucțiunea ciclică `for`

Instrucțiunea ciclică `for` permite iterarea unui set de instrucțiuni pentru un număr fix de iterații. Spre deosebire de limbajul standard C/C++ în care se utilizează un contor pentru a controla numărul de iterații, în Python instrucțiunea `for` este de tipul `for_each`. Aceasta înseamnă că necesită un obiect de tip secvență sau obiect iterabil ce va genera un set de date a cărui lungime este egală cu numărul de iterații ale buclei `for`.

Stringurile, listele, tuplurile, seturile și dicționarele sunt obiecte de tip container **iterabile**.

> **OBS T1.12** Deoarece bucla **for** iterează peste elementele unui container atât timp cât mai există elemente în acesta, nu este nevoie de o condiție de ieșire din buclă

[119]:
```
# Iterare listă de întregi
my_list = [1,2,3,4,5,6,7]
for i in my_list:
    print (i)
```

[119]:
```
1
2
3
4
5
6
7
```

[120]:
```
# Iterare listă de caractere
my_list = ['a','b','c','d','e']
for c in my_list:
    print (c)
```

[120]:
```
a
b
c
d
e
```

[121]:
```
# Iterare domeniu specific [0,10) - funcția range retunează
# o listă de întregi
for i in range(5):
    print (i)
```

[121]:
```
0
1
2
3
4
```

[122]:
```
# Iterare domeniu specific [2,8]
for i in range(2,8):
    print (i)
```

[122]:
```
2
3
4
5
6
7
```

[123]:
```
# Iterare domeniu specific cu incrementarea variabilei
# cu o valoare fixă
for i in range(2,10,3):
    print (i)
```

[123]:
```
2
5
8
```

[124]:
```
# Iterare domeniu specific cu decrementarea variabilei
# cu o valoare fixă
for i in range(10,0,-2):
    print (i)
```

[124]:
```
10
8
6
```

```
4
2
```

T1.2.8.3 Instrucțiunea condițională `if`

Instrucțiunea `if` permite testarea unei condiții și executarea unui set de instrucțiuni în cazul în care condiția e evaluată ca fiind `True`. Se pot adăuga ramuri de `elif` și `else` pentru a executa un set de instrucțiuni alternative atunci când condiția este `False`.

[125]:
```
# Instrucțiune if simplă
a = 3
b = 5
if (a>b):
    print ("Max is a")
else:
    print ("Max is b")
```

[125]:
```
Max is b
```

[126]:
```
# Instrucțiune if imbricată cu ramuri elif
a = 2
b = 7
c = 7
if (a>b):
    if (a>c):
        print ("Max is A")
    elif (c>a):
        print ("Max is C")
    else:
        print ("A and C are both max")
elif (b>a):
    if (b>c):
        print ("Max is B")
    elif (c>b):
        print ("Max is C")
    else:
        print ("B and C are both max")
else:
    if (a>c):
        print ("A and B are both max")
    elif (c>a):
        print ("Max is C")
```

```
    else:
        print ("A, B and C are all max")
```

[126]:
```
B and C are both max
```

[127]:
```
# Operator ternar sub formă de instrucțiune if
a = 3
b = 5
a if a>b else b
```

[127]:
```
5
```

T1.2.8.4 Instrucțiuni de salt: `break`, `continue`, `pass`

În cadrul instrucțiunilor ciclice, este nevoie uneori ca acestea să-și termine execuția în mod forțat, independent de iterator sau condiție de test. Pentru aceasta există două instrucțiuni de salt: `break` și `continue` ce au efect doar dacă se află în interiorul unor instrucțiuni ciclice de tipul `while` sau `for`.

Instrucțiunea `break` va forța ieșirea din bucla curentă și nu va mai executa nici o altă instrucțiune din această buclă. Dacă sunt mai mult bucle imbricate, aceasta va ieși doar din cea curentă.

Instrucțiunea `continue` va sări la următoarea iterație din buclă sau la testarea condiției inițiale fără a mai executa restul instrucțiunilor, însă fără a ieși din buclă.

Instrucțiunea `pass` este instrucțiunea vidă și nu are niciun efect programatic. Este folosită ca și placeholder în cod ce trebuie completat ulterior.

[128]:
```
# Exemplu break în while: nu se executa bucla după
# ce i ajunge la valoarea 3
i = 5
while i > 0:
    if i==3:
        break
    print (i)
    i-=1
```

[128]:
```
5
4
```

[129]:
```
# Exemplu continue în while: se sare peste restul
# instrucțiunilor când i ajunge la valoarea 3
# Trebuie să avem grijă să actualizăm variabila
# de test înainte de salt
i = 5
```

```
while i > 0:
    if i==3:
        i-=1
        continue
    print (i)
    i-=1
```

[129]:
```
5
4
2
1
```

[130]:
```
# Exemplu pass în while: nu se întâmplă nimic,
# bucla se execută normal
i = 5
while i > 0:
    if i==3:
        pass
    print (i)
    i-=1
```

[130]:
```
5
4
3
2
1
```

[131]:
```
# Exemplu break în for: nu se executa bucla după
# ce i ajunge la valoarea 3
for i in range(5):
    if i==3:
        break
    print (i)
    i-=1
```

[131]:
```
0
1
2
```

[132]:
```
# Exemplu continue în for:  se sare peste restul
# instrucțiunilor când i ajunge la valoarea 3
for i in range(5):
    if i==3:
        continue
```

```
    print (i)
    i-=1
```

[132]:
```
0
1
2
4
```

T1.2.9 Accesul la cod extern (import)

Reutilizarea codului este unul dintre cele mai importante aspecte ale programării și permite definirea unui set de funcții sau clase în mod independent ce pot fi incluse ulterior în alte coduri sau aplicații conexe. În Python, organizarea codului extern se face prin intermediul **modulelor**.

Modulele sunt fișiere externe ce conțin clase, funcții și definiții de constante. Modulul trebuie să fie accesibil codului curent prin calea de system (*system path*) sau prin calea curentă (*current path*). Pentru ca un modul să fie făcut disponibil în codul curent, se utilizează cuvântul cheie `import` cu următoarele opțiuni de sintaxă:

- `import module_name`
- `import module_name as m` - folosește un alias pentru numele modulului
- `from module_name import submodule` - importă doar un submodul al modulului
- `from module import *` - importă toate clasele, funcțiile și constantele fără a fi necesară utilizarea numelui modulului înainte de acestea

[133]:
```
# Importăm modulul math
import math
math.sqrt(25)
```

[133]:
```
5.0
```

[134]:
```
# Importăm modulul math și îi atribuim un alias
import math as m
m.sqrt(25)
```

[134]:
```
5.0
```

[135]:
```
# Importăm toate funcțiile din modulul math
# fără a mai fi necesară utilizarea numelui său
# în cadrul apelului funcțiilor
from math import *
```

```
sqrt(25)
```

[135]:
```
5.0
```

T1.2.10 Lucrul cu fișiere

Citirea și scrierea datelor din/în fișiere externe este esențială în majoritatea aplicațiilor programatice. Fișierele se pot afla pe un disc local sau la o adresă URL și pot fi stocate în format text sau binar.

Cele mai importante funcții de lucru cu fișierele sunt prezentate mai jos:

- Deschiderea fișierelor:
  `f = open(file_path_and_name, 'read_mode')`
  - **read_mode** poate fi *'w'* pentru scriere/creare, *'r'* pentru citire și *'a'* pentru adăugare la final (append). Când este utilizat modul 'w', dacă fișierul nu există, acesta este creat. Dacă există, conținutul este **șters**. Dacă nu se dorește ștergerea conținutului, se poate utiliza modul 'a'. Pentru fișiere binare se adaugă modul *'b'*, ex. *'wb'*.
  - **file_path_and_name** conține calea absolută sau relativă către fișier.
- Închiderea unui fișier:
  `f.close()`
- Citirea din fișier text:
  - `f.read()` - returnează întreg conținutul fișierului în format string
  - `f.readlines()` - returnează o listă a liniilor individuale conținute în fișier
- Scrierea în fișiere text:
  - `f.write(string)` - scrie stringul în fișier
  - `f.writeline(list_of_strings)` - scrie o listă de stringuri în fișier

[136]:
```
# Crează un fișier denumit test.txt și scrie o linie în el
f = open("test.txt", 'w')
f.write("Hello, this a line\n")
```

```
f.close()
```

[137]:
```
# Adaugă alte linii la fișier
f = open("test.txt", 'a')
f.writelines(["A second line\n", "A third line\n"])
f.close()
```

[138]:
```
# Citește conținutul fișierului și îl afișează
f = open("test.txt", 'r')
for line in f.readlines():
    print (line)
f.close()
```

[138]:
```
Hello, this a line
A second line
A third line
```

T1.2.10.1 Manageri de context și instrucțiunea `with`

O metodă mai bună de deschidere a fișierelor și de a ne asigura că fluxurile de citire/scriere sunt închise la final este prin utilizarea instrucțiunii `with`. Aceasta va crea un manager de conținut și se va asigura că fișierul este închis la ieșirea din blocul de instrucțiuni, indiferent de rezultatul acestora.

[139]:
```
with open("test.txt") as f:
    for line in f.readlines():
        print (line)
```

[139]:
```
Hello, this a line
A second line
A third line
```

T1.3. Concluzii

În cadrul acestui prim tutorial am indexat o serie minimală de noțiuni și instrucțiuni necesare programării în limbajul Python și lucrul cu mediul interactiv Jupyter. În niciun caz acest tutorial nu își propune să introducă totalitatea elementelor din programarea Python, ci doar pe acelea ce sunt necesare în cadrul următoarelor tutoriale și pentru lucru cu secvențe de cod de bază în Python. În bibliografia suplimentară veți regăsi și două titluri axate pe învățarea automată și învățarea folosind rețele neurale adânci, unul dintre cele mai importante domenii de aplicabilitate ale limbajului Python la ora actuală.

Bibliografie suplimentară

- Mark Lutz, "Learning Python", 3rd Edition, O'Reilly Media, 2007
- Aurélien Géron, "Hands-on Machine Learning with Scikit-Learn and Tensorflow", O'Reilly Media, 2017
- Francois Chollet, "Deep Learning with Python", Manning Publications, 2017

Resurse media

- Python 3 Documentation, online: `https://docs.python.org/3/`
- Style Guide for Python - PEP 8, online: `https://legacy.python.org/dev/peps/pep-0008/`
- CodeAcademy Python Course, online: `https://www.codecademy.com/learn/learn-python`
- Jupyter Documentation, online: `https://jupyter-notebook.readthedocs.io/en/5.7.4/`

T2 Citirea, scrierea, redarea și afișarea semnalelor vocale

După scurta introducere în Python, primul tutorial de prelucrare a semnalului vocal se va axa pe principalele module și funcții utilizate pentru procesarea fișierelor audio. Totodată, vor fi introduse și noțiuni de bază referitoare la producerea vorbirii și modelele matematice/programatice de producere a vorbirii, precum și noțiuni referitoare la achiziția și stocarea digitală a semnalului vocal.

T2.1. Vorbirea. Caracteristici fundamentale

Comunicarea reprezintă una dintre caracteristicile cele mai importante ale evoluției speciei umane. Prin comunicare se realizează de fapt transferul de informație de la sursă la destinație cu ajutorul limbii sau a unui limbaj comun. Acest transfer poate fi efectuat într-o formă scrisă sau verbală. Dacă forma scrisă este de cele mai multe ori o codare brută a mesajului, folosind un set finit de simboluri grafice, forma verbală cuprinde un univers complex, potențial infinit, de factori și ele- mente care interacționează pentru a facilita transmiterea mesajului către destinatar. Alături de fonemele ce alcătuiesc mesajul, caracteristicile fiziologice ale vorbitorului, accentul, intonația și ritmul vorbirii, precum și elementele non-verbale (mimică, gestică, poziția corpului, etc.) determină modul în care destinatarul decodează și interpretează mesajul transmis. Astfel că același mesaj codat în mod identic în formă scrisă, nu va fi niciodată reprodus identic în mod verbal, nici chiar de către același vorbitor la intervale consecutive, scurte de timp. Această variabilitate de codare a mesajului în formă verbală reprezintă una dintre cele mai importante provocări din domeniul mod- elării inteligenței și a fiziologiei umane. Această provocare se referă atât la metodele de sinteză a vorbirii, precum și la cele de recunoaștere a acesteia.

Din punct de vedere fiziologic, sistemul vocal uman este compus din ansamblul organelor fona- toare: plămâni, esofag, laringe, glotis (corzi vocale), faringe, cavitatea orală, cavitatea nazală, limbă, văl palatin, maxilar și buze. În Fig. T1.1 este prezentat acest ansamblu cu excepția plămânilor. Aceștia din urmă reprezintă sursa vorbirii prin expirația aerului, iar restul organelor contribuie la modularea acestui flux de aer și generarea sunetului perceput în exteriorul corpului fie de către unul sau mai mulți observatori, fie de un dispozitiv de achiziție a sunetului, cum ar fi un microfon.

Pornind de la sistemul anatomic de producere a vorbirii, modelarea acestui proces complex necesită anumite simplificări și aproximări, multitudinea proceselor implicate și complexitatea lor neputând fi modelată corect în totalitate. Astfel că există mai multe modele acceptate ca fiind valide pentru modelarea vorbirii din punct de vedere matematic, iar în continuare sunt prezentate cele mai importante dintre acestea:

- modelul tuburilor acustice;
- modelul filtrelor trece-bandă;
- modelul sinusoidal;
- modelul liniar-separabil sau sursă-filtru;
- modelul undelor glotale.

Prin modelarea producerii vorbirii, aplicațiile de codare, recunoaștere sau sinteză a semnalului vocal pot fi mult mai ușor implementate. Rezultatele lor apropiindu-se în ultimii ani de calitatea vocii naturale.

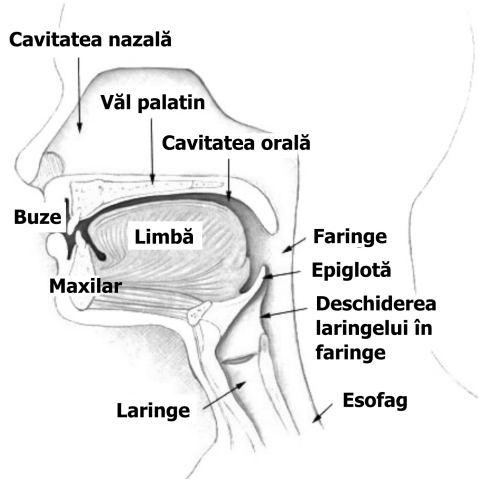


Fig.T2.1. Fiziologia sistemului vocal uman

T2.2. Fundamente ale achiziției și stocării semnalului vocal în format digital

Semnalul vocal poate fi achiziționat folosind un microfon. Cu ajutorul diafragmei, microfonul realizează conversia mișcării particulelor de aer generate de fonație, în curent electric (semnal).

Semnalul astfel rezultat poate fi stocat fie în format analogic, fie digital. Deși în domeniul analogic, informația stocată este în mod teoretic fără pierderi de informație, acest format face mai dificilă analiza și postprocesarea semnalului înregistrat. Astfel că, cel mai comun mod de stocare în ziua de azi, este cel digital.

Formatul de stocare digital sau conversia analog-digitală a semnalului implică un număr de aproximări și pierderi de informație. Dintre acestea, cele mai importante sunt:

- **frecvența de eșantionare (Fs)** (aproximarea în domeniul timp): reprezintă numărul de eșantioane de semnal dintr-o secundă ce vor fi stocate. Unitatea de măsură penrtu aceasta este Hertz-ul [Hz], iar cele mai utilizate valori ale frecvenței de eșantionare ale unui semnal vocal sunt: 8kHz, 16kHz și 48kHz. O regulă esențială pentru selectarea

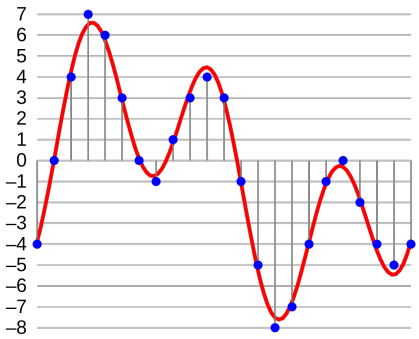


Fig.T2.2. Conversia digitală a semnalului - forma de undă analogică este redată cu roșu, iar reprezentarea digitală cu albastru. Barele verticale reprezintă aproximarea în timp, dată de perioada de eșantionare. Punctele albastre reprezintă aproximarea în amplitudine.

valorii frecvenței de eșantionare este bazată pe teorema lui Nyquist, care spune că **frecvența de eșantionare a unui semnal trebuie să fie egală cu cel puțin dublul frecvenței utile maxime din semnalul de intrare**. În cazul vorbirii, frecvența maximă este de aproximativ 10kHz, astfel că o frecvență de eșantionare de 22kHz ar fi cea corectă și suficientă. Însă, majoritatea informației utile din semnalul vocal este situată în banda de 300-3400Hz. Astfel că, în versiunile inițiale ale sistemelor de achiziție de semnal vocal (sau telefonie), frecvența de eșantionare era setată la 8kHz.

- **rezoluția de bit (sau numărul de biți/eșantion - en. bits per sample - bps)** (aproximare în domeniul amplitudinii): este numărul de biți pe care este stocată informația din fiecare eșantion de semnal și este direct proporțional cu rezoluția de amplitudine a eșantioanelor. Valori comune utilizate în prelucrarea digitală a semnalului vocal sunt: 8bps, 16bps sau 32bps. Rezoluția de bit ne dă acuratețea aproximării amplitudinii semnalului.

În ceea ce privește stocarea datelor audio, cel mai utilizat format digital este **WAV**: `https://en.wikipedia.org/wiki/WAV` . Acesta nu codează semnalul digital în niciun fel, ci stochează eșantioanele ca atare. Alte formate uzuale pentru semnalul audio sunt: MP3, OGG sau FLAC. Acestea realizează o anumită codare/compresie a semnalului și pot mări gradul de distorsiune al semnalului original.

Odată ce semnalul vocal este în format digital, acesta poate fi analizat și procesat folosind diverse unelte și metode. Cea mai simplă analiză este reprezentată de vizualizarea formei de undă în domeniul timp. Prin simpla vizualizare a valorilor eșantioanelor semnalului vocal achiziționat, o serie de caracteristici specifice pot fi extrase. Cele mai evidente sunt: periodicitatea, amplitudinea și numărul de treceri prin zero ale semnalului.

O altă metodă fundamentală de analiză a semnalului vocal este cea de analiză în frecvență sau analiza spectrală. Spectrul este obținut cu ajutorul transformatei Fourier și va fi discutat într-un capitol viitor.

Următoarele secțiuni ale acestui tutorial se vor axa pe accesarea datelor audio din fișier și vizualizarea acestora.

T2.3. Citirea semnalului vocal din fișier

Citirea din fișier a eșantioanelor unui semnal digitizat necesită cunoașterea formatului de stocare al datelor din fișier. Astfel că sunt necesare o serie de funcții specializate ce pot interpreta acest format prin identificarea antetului specific, a eventualei codări și a datelor sau valorilor eșantioanelor în sine. Rezultatul acestor funcții este de obicei o structură specifică ce conține datele într-un format interpretabil de limbajul de programare utilizat. În cele mai multe cazuri, eșantioanele semnalului sunt returnate sub forma unei matrici de valori întregi sau reale. Pe lângă eșantioane, este important să obținem și caracteristicile acestor, precum frecvența de eșantionare la care s-a făcur digitizarea, rezoluția de bit și numărul de canale audio (mono sau stereo).

În Python, unul dintre modulele ce conține funcții specializate pentru prelucrarea fișierelor audio de tip `*.wav` este modulul `wave`.[1] Astfel că, dacă dorim să utilizăm aceste funcții, va trebui să importăm modulul `wave` în codul nostru:

[1]:
```
import wave
```

OBS T2.1 În loc de sintaxa anterioară, putem importa toate funcțiile modulului fără a mai fi necesară prefixarea acestora:

```
from wave import *
```

Însă, dacă utilizăm acest format s-ar putea să suprascriem funcții de bază ale Python, cu funcții definite doar în modulul `wave`. De exemplu, funcția `open` ce poate opera asupra oricărui fișier, va fi înlocuită cu cea disponibilă în modulul `wave`.

```
f = open('aa.txt')
```

Astfel că este recomandat să utilizăm un import simplu al modulului `wave` sau să folosim un alias:

[1] `https://docs.python.org/3/library/wave.html`

```
import wave as w
```

OBS T2.2 În funcție de sistemul de operare pe care rulăm Jupyter, calea către fișiere trebuie specificată în mod diferit. De exemplu, în Windows, separatorul implicit de nivel de stocare este backslash \, însă în Python, precum și în alte limbaje de programare backslash-ul reprezintă începutul unei secvențe escape (ex. `\n` - rând nou). Astfel că, pentru a utiliza acest separator în căile către fișiere, el trebuie dublat: `'cale\\catre\\fisier.wav'`. Pentru simplitate, se poate utiliza, însă, forwardslash chiar și în sistemul de operare Windows: `cale/catre/fisier.wav`

OBS T2.3 Atunci când dorim să accesăm fișiere ce nu sunt în directorul curent, este important să facem diferența dintre calea absolută (en. *absolute path*) și calea relativă (en. *relative path*). Calea absolută pornește din rădăcina arborelui director al sistemului de operare (ex. în Windows `C:\Users\Student\fisier.wav`) și nu este indicat să fie folosită datorită incompatibilităților ce apar la migrarea codului pe o altă mașină. Calea relativă pornește de la directorul curent și parcurge arborele director pas cu pas. De exemplu, dacă avem nevoie să accesăm un fișier ce se află într-un director părinte al celui curent, vom scrie: `..\director_parinte\fisier.wav`

Acum că avem acces la funcțiile de citire a fișierelor `*.wav` și știm cum să accesăm fișierele locale, putem să mergem mai departe și să citim primul nostru fișier audio. Înregistrări cu semnal audio sunt disponibile în directorul `speech_files/`

[2]:
```
input_wav_file = './speech_files/adr_rnd1_001.wav'
wav_struct = wave.open(input_wav_file, 'r')
```

Dacă ne uităm la definiția funcției `wave.open()`[2], putem să observăm faptul că returnează un obiect de tip `Wave_read`, sau un handler al datelor stocate în acel fișier. Prin simpla apelare a funcției `open()`, datele stocate nu sunt efectiv citite. Rezultatul funcției doar indică programului o anumită zonă din memorie de unde acestea pot fi extrase ulterior.

Ne reamintim faptul că atunci când stocăm un semnal audio chiar și în format necodat, precum cel `wav`, este foarte important să stocăm și alte informații legate de conținutul fișierului, cum ar fi **frecvența de eșantionare** și **rezoluția de bit** sau numărul de canale audio. Astfel că, obiectul `Wave_read` are un set de funcții predefinite ce pot returna aceste

[2] `https://docs.python.org/3/library/wave.html`

informații.

> **OBS T2.4** Numărul de canale pe care este stocat un semnal audio nu a fost discutat în partea introductivă, deoarece pentru voce nu se utilizează formatul stereo. Dacă, însă, acest format este utilizat, se poate presupune faptul că ambele canale conțin aceeași informație, iar unul dintre ele poate fi ignorat.

[3]:
```
# Returnează frecvența de eșantionare
sampling_frequency = wav_struct.getframerate()
print ("The sampling frequency of the file is: %d [Hz]" \
         %sampling_frequency)

# Returnează numărul de biți pe care e stocat un eșantion
bit_depth = wav_struct.getsampwidth()
print ("The sample width of the file is: %d \
         [bytes/sample] or %d [bits/sample] \
        %(bit_depth, bit_depth*8))

# Returnează numărul de canale
no_channels = wav_struct.getnchannels()
print ("The number of channels in the file is %d or %s" \
       %(no_channels, 'mono' if no_channels==1  \
        else 'stereo'))

# Retunează numărul de eșantioane
nframes = wav_struct.getnframes()
print ("The number of samples in the file is: %d" %nframes)

# Returnează tipul compresiei. Pentru fișiere wav,
# aceasta este 'None'
compression_type = wav_struct.getcomptype()
print ("The compression type of the file is: %s "  \
        %compression_type)

# Sau putem obține toată informația de mai sus
# într-o singură instrucțiune
(nchannels, sampwidth, framerate, nframes, comptype, \
         compname) = wav_struct.getparams()
```

[3]:
```
The sampling frequency of the file is: 48000 [Hz]
The sample width: 2 [bytes/sample] or 16 [bits/sample]
The number of channels in the file is 1 or mono
The number of samples in the file is: 174628
The compression type of the file is: NONE
```

Însă tot nu am citit valorile eșantioanelor. Pentru aceasta, vom utiliza funcția `wave.readframes(n)`, unde `n` specifică numărul de eșantioane pe care dorim să le citim. Dacă dorim să citim toate eșantioanele, putem seta `n` la valoarea `-1`. Funcția `readframes()` returnează un vector de obiecte de tip byte, ce pot fi interpretate și ca obiecte string.

Pentru a converti aceste obiecte în valori numerice, se poate utiliza funcția `numpy.frombuffer()` în care specificăm formatul `int16` corespunzător numerelor întregi stocate pe 16 biți, echivalent cu rezoluția de bit a fișierului nostru (precum și a majorității fișierelor de tip `wav`).

OBS T2.5 NumPy este modulul fundamental pentru calculule numerice în Python: `http://www.numpy.org/`. Pentru a-l utiliza trebuie să-l importăm în cod. O convenție generală de utilizare a sa implică și folosirea alias-ului `np` pentru el:

[4]:
```
import numpy as np
# Citim biții din fișierul de intrare
wav_bytes = wav_struct.readframes(-1)
# Afișam tipul datelor citite
print ("wav_bytes is of type %s" %type(wav_bytes))
# Convertim datele în format int16
wav_data = np.frombuffer(wav_bytes, dtype='int16')
# Afișăm tipul datelor convertite
print ("wav_data is of type %s" %type(wav_data))
```

[4]:
```
wav_bytes is of type <type 'str'>
wav_data is of type <type 'numpy.ndarray'>
```

OBS T2.6 În cazul în care doriți să rulați codul de mai sus de mai multe ori, după ce s-a realizat citirea, cursorul de fișier este poziționat la sfârșitul acestuia. Astfel că, pentru a reciti datele trebuie utilizată funcția `rewind()`: `wav_struct.rewind()`. Cu excepția cazului în care ați închis deja fluxul de fișier și se va genera o eroare.

> **OBS T2.7** Trebuie să avem grijă să **închidem întotdeauna** fluxul de fișier după ce au fost citite datele. Altfel s-ar puta să apară ulterior erori de citire/scriere pe disc sau pierderi de date.

[5]:
```
# Închidem fluxul de fișier
wav_struct.close()
```

După ce am rulat pașii anteriori, eșantioanele de semnal și caracteristicile lor sunt stocate în variabilele noastre. Putem acum să le vizualizăm sau să le manipulăm.

T2.4. Redarea semnalului vocal

Eșantioanele semnalului citit din fișier sunt acum stocate în variabila `wav_data`. Un prim pas logic ar fi să ascultăm conținutul audio din fișier și să ne asigurăm că într-adevăr avem semnal vocal. În iPython avem la dispoziție o metodă simplă de a include un semnal audio în notebook, iar browserul va avea grijă ca acesta să fie redat.

OBS T2.8 Dacă dorim să redăm un semnal folosind cod Python pur, este nevoie de un alt set de module: `http://guzalexander.com/2012/08/17/playing-a-sound-with-python.html`

[6]:
```
import IPython
IPython.display.Audio(wav_data, rate=sampling_frequency)
```

[6]: 

Pentru a înțelege cât de importantă este frecvența de eșantioanare, putem să modificăm acest parametru în funcția de redare audio:

[7]:
```
# Înjumătățim frecvența de eșantionare
IPython.display.Audio(wav_data, rate=sampling_frequency/2)
```

[7]: 

[8]:
```
# Dublăm frecvența de eșantionare
IPython.display.Audio(wav_data, rate=sampling_frequency*2)
```

[8]: 

Exercițiu T2.4.1 De ce sunt ultimele două semnale audio atât de diferite de cel original? Nu am modificat variabila *wav_data* deloc. ■

T2.5. Vizualizarea semnalului vocal

În aplicații software, vizualizarea datelor este denumită **plotare** (en. *plotting*), astfel că vom folosi acest termen pentru a ne referi la diferitele forme de vizualizare a informației conținute de semnalul vocal.

În Python cel mai utilizat modul pentru plotare grafică este Matplotlib.[1]

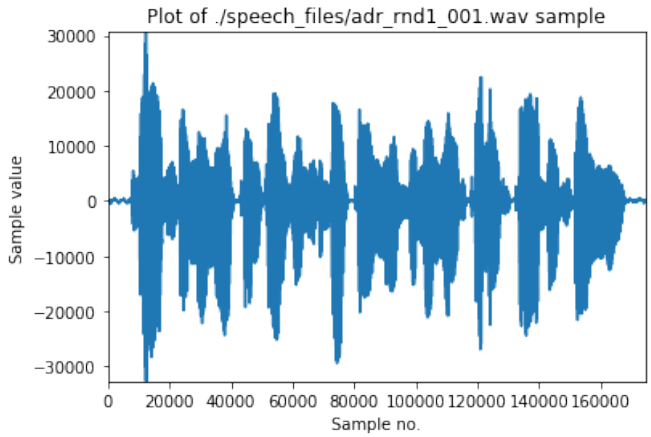
[9]:
```
import matplotlib.pylab as pl
%matplotlib inline
```

> **OBS T2.9** Instrucțiunea `%matplotlib inline` face parte din sintaxa iPython și nu trebuie utilizată în codul Python pur. Rezultatul ei este că ploturile sunt integrate în browser și nu apar într-o fereastră separată.

Să plotăm datele citite anterior:

[10]:
```
# Creăm o nouă fereastră de plot
pl.figure()
# Dăm un titlu ferestrei
pl.title("Plot of %s sample" %input_wav_file)
# Afișăm datele din wav_data
pl.plot(wav_data)
# Setăm limitele axei oX în funcție de lungimea wav_data
pl.xlim([0, len(wav_data)])
# Setăm limitele axei oY în funcție de valorile minime
# și maxime din wav_data
pl.ylim([min(wav_data)-1, max(wav_data)+1])
# Denumim axa oX
pl.xlabel("Sample no.")
# Denumim axa oY
pl.ylabel("Sample value")
```

[1] `https://matplotlib.org/index.html`



Analizând figura, putem observa că sunt aproximativ 160.000 de eșantioane cu valori cuprinse între -30.000 și 30.000.

OBS T2.10 Valorile afișate ale eșantioanelor nu sunt valorile absolute ale curentului electric generat de microfon, ci mai degrabă un rezultat al convențiilor de stocare a datelor și a rezoluției de bit. Pentru a avea o reprezentare uniformă a semnalului vocal, independent de numărul de biți utilizați în digitizare, modul standard de afișare a datelor audio este prin scalare în intervalul [-1, 1] pe axa oY și având timpul pe axa oX:

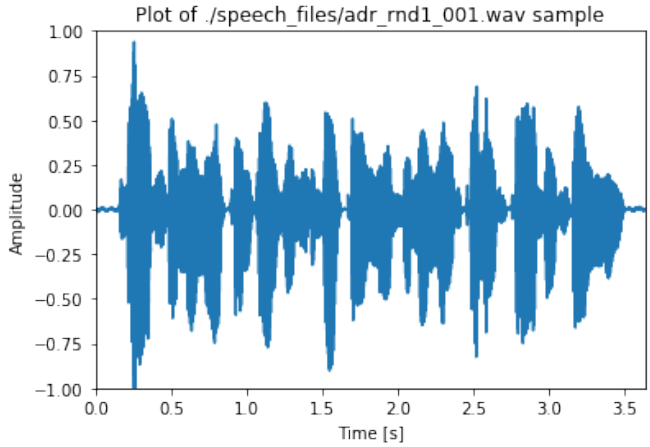
[11]:
```
# Scalăm wav_data la [-1,1]
wav_data = wav_data/float(max(abs(wav_data)))
# Calculăm durata totală a semnalului(în secunde)
duration = len(wav_data)*1.00/sampling_frequency
print ("Signal duration %f second(s)" %duration)
```

[11]:
```
Signal duration 3.638083 second(s)
```

Afișăm datele în noul format:

[12]:
```
# Convertim indexul eșantionului într-un multiplu întreg al
# perioadei de eșantionare (=1/frecvența de eșantionare)
time_axis = np.arange(0, len(wav_data)) \
        *1.00/sampling_frequency
# În locul indexului eșantionului, folosim variabila
# time_axis pentru axa oX
pl.plot(time_axis, wav_data)
# Limităm axele oX și oY
pl.xlim([0, duration])
pl.ylim([-1, 1])
```

```
# Adăugăm etichete pentru axe și titlu pentru fereastră
pl.xlabel('Time [s]')
pl.ylabel('Amplitude')
pl.title("Plot of %s sample" %input_wav_file)
```



Exercițiu T2.5.1 Citiți și alte fișiere audio, ascultați-le și afișați eșantioanele lor. ■

T2.6. Scrierea eșantioanelor audio în fișier

Să presupunem că dorim să facem anumite modificări asupra datelor audio și să le stocăm mai apoi într-un alt fișier (sau poate chiar același).

Pentru exemplificare vom considera că din fișierul audio inițial dorim să păstrăm doar primele 10.000 de eșantioane. Astfel că, vom crea o variabilă ce conține aceste date:

[14]:
```
new_wav_data = wav_data[:10000]
```

Și dorim să stocăm datele într-un fișier denumit `output_wave.wav` care se află în directorul `speech_files/`:

[15]:
```
output_filename = 'output_wave.wav'
output_folder = './speech_files'

# Deschidem un flux de ieșire către locația fișierului
wav_out = wave.open(output_folder+'/'+output_filename, 'w')

# Definim proprietățile fluxului de ieșire
wav_out.setnchannels(no_channels)
wav_out.setsampwidth(sampwidth)
wav_out.setframerate(sampling_frequency)
wav_out.setnframes(len(new_wav_data))

# Scriem eșantioanele in fisier
wav_out.writeframes(new_wav_data)

# Închidem fluxul de fișier de ieșire
wav_out.close()
```

Exercițiu T2.6.1 Citiți fișierul nou generat și ascultați-l. ■

T2.7. Concluzii

În acest prim tutorial de procesare a semnalului vocal folosind Python am introdus o serie de operații de bază cu fișiere folosind modulul `wave`. Am reușit să citim și să scriem fișiere `*.wav` și, de asemenea, să le ascultăm sau să vizualizăm valorile eșantioanelor lor.

În tutorialul următor vom realiza o serie de analize specifice semnalului vocal și vom încerca să determinăm părțile de semnal ce conțin vorbire și cele ce sunt liniște sau zgomot de fundal.

Bibliografie suplimentară

- Benesty et al, "Springer Handbook of Speech Processing", Springer, 2008
- S. V. Vaseghi, "Multimedia Signal Processing: Theory and applications in Speech, Music and Communications", John Wiley Sons, 2007

Resurse media

- The Virtual Linguistics Campus, "Speech Anatomy", online: `https://www.youtube.com/watch?v=-m-gudHhLxc`
- Simon King, "Speech Processing Course", University of Edinburgh, online: `http://www.speech.zone/courses/speech-processing/module-1-introduction/`

T3 Analiza pe termen scurt

În acest tutorial vom introduce noțiunea de **analiză pe termen scurt** în domeniul timp a semnalului vocal. Aceasta mai este denumită și **analiză la nivel de cadru** sau **analiză cadru cu cadru**.

Dar mai întâi să ne reamintim modul în care putem să citim eșantioanele și informațiile aferente unui semnal vocal:

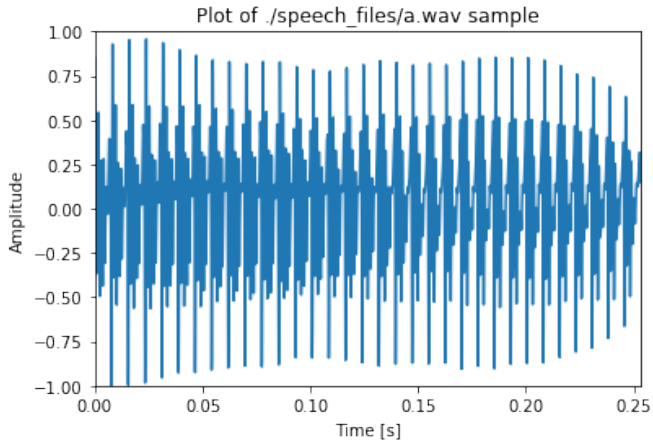
[1]:
```
import wave
import numpy as np

input_wav_file = './speech_files/a.wav'
wav_struct = wave.open(input_wav_file, 'r')
# Determinăm frecvența de eșantionare
sampling_frequency = wav_struct.getframerate()
# Determinăm rezoluția de bit
bit_depth = wav_struct.getsampwidth()
# Determinăm numărul de canale
no_channels = wav_struct.getnchannels()
# Citim eșantioanele
wav_bytes = wav_struct.readframes(-1)
# Convertim datele citite în valori întregi
wav_data = np.frombuffer(wav_bytes, dtype='int16')
# Închidem fluxul de intrare
wav_struct.close()
```

Și cum se ascultă și vizualizează datele audio:

[2]:
```
import matplotlib.pylab as pl
%matplotlib inline
# Normalizăm wav_data la [-1,1]
wav_data = wav_data/float(max(abs(wav_data)))
# Calculăm axa oX ca o secvență de multipli întregi ai
# perioadei de eșantionare
time_axis = np.arange(0, len(wav_data))* \
        *1.00/sampling_frequency
# Calculăm durata semnalului în secunde
duration = len(wav_data)*1.00/sampling_frequency
print ("Duration %f second" %duration)
# Afișăm datele cu axa oX având unitatea de timp
pl.plot(time_axis, wav_data)
pl.xlim([0, duration])
pl.ylim([-1, 1])
pl.xlabel('Time [s]')
pl.ylabel('Amplitude')
```

```
pl.title("Plot of %s sample" %input_wav_file)
```

[2]:
```
Duration 0.253063 second
```



[3]:
```
# Ascultăm semnalul
import IPython
IPython.display.Audio(wav_data, rate=sampling_frequency)
```

[3]:


T3.1. Caracteristici fundamentale ale semnalului vocal

Din punct de vedere fiziologic, sistemul de vorbire uman este compus din următoarele organe: plămâni, esofag, laringe, glottis (corzi vocale), faringe, cavitatea bucală, cavitatea nazală, limbă, uvulă, văl palatin, dinți, mandibulă și buze. Toate aceste organe sunt denumite **articulatori** și interacționează într-o manieră complexă pentru a realiza sunete.

Sursa vorbirii este fluxul de aer expirat din plămâni. Acest flux de aer este ulterior modulat în calea sa prin organele articulatoarea, ca mai apoi să producă vocea la ieșirea din cavitatea bucală sau nazală (sau combinat). Deși fiecare persoană are caracteristici individuale ale vocii, limba vorbită determină un set predefinit de poziții ale articulatorilor ce corespund setului de foneme utilizat în limba respectivă. Tranziția de la o poziția de articulare la alta este denumită **co-articulare**.

Datorită acestor tranziții și a modificării constante a poziției articulatorilor, semnalul vocal nu este staționar. Cu toate acestea, poate fi considerat ca fiind **cvasi-staționar** pe intervale de durată egală cu 20-40ms. Această durată este denumită **constanta de staționaritate** a semnalului vocal. În acest interval de timp, se poate face presupunerea că semnalul nu își schimbă în mod fundamental caracterisiticile, atât cele temporale, cât și cele spectrale. Ca urmare, înainte de a realiza orice tip de analiză sau extragere de caracteristici din semnalul vocal, acesta trebuie segmentat în **ferestre sau cadre de analiză**. Durata cadrelor având un efect direct asupra numărului de eșantioane ce sunt procesate la un moment dat de către algoritm sau metoda de analiză.

De exemplu, pentru un semnal eșantionat la 16kHz vom folosi o fereastră de analiză de 320 - 640 de eșantioane. Într-un tutorial viitor vom discuta despre transformata Fourier și multiplele analize derivate din aceasta. Ca o consecință a utilizării acestei transformate pe scară largă în prelucrarea semnalului vocal și în special a algoritmilor de calcul rapid a coeficienților Fourier, numărul de eșantioane dintr-un cadru este ales să fie egal cu o putere a lui 2 (de ex. 128, 256, 512, etc.). În plus, cadrele de analiză nu sunt disjuncte și de cele mai multe ori au un grad de suprapunere exprimat procentual (de ex. 50%, 25%, etc.)

Din punct de vedere al procesării de semnal, segmentarea semnalului în cadre de analiză este echivalentă cu înmulțirea semnalului cu o fereastră rectangulară de lungime egală cu constanta de staționaritate și care are diferite întârzieri.

OBS T3.1 Noțiunile de *cadru de analiză* și *fereastră de analiză* vor fi utilizate interschimbabil fără însă a avea vreun efect asupra procesării datelor.

T3.2. Analiza cadru cu cadru a semnalului vocal

Să încercăm atunci să împărțim semnalul vocal citit anterior în cadre de analiză. În primul rând trebuie să determinăm lungimea ferestrei de analiză. Dacă semnalul vocal este cvasi-staționar pe o perioadă de 20 până la 40ms, numărul de eșantioane corespunzătoare acestei perioade este dependent de frecvența de eșantionare:

[4]:
```
# Numărul de eșantioane dintr-un cadru de analiză de 20msec
window_length_20 = int(20*1e-3*sampling_frequency)
# Numărul de eșantioane dintr-un cadru de analiză de 40msec
window_length_40 = int(40*1e-3*sampling_frequency)
print ("The analysis window should be between %d and %d" \
        %(window_length_20, window_length_40))
```

[4]:
```
The length of the analysis window should be between 320 and
640 samples
```

Astfel că, dacă alegem orice valoare cuprinsă între cele două valori calculate mai sus, semnalul vocal poate fi considerat staționar.

[5]:
```
# Selectăm durata cadrului de analiză de 20ms
window_length = window_length_20
```

În funcție de lungimea cadrului de analiză, vom calcula numărul întreg de cadre existente în semnalul de intrare.

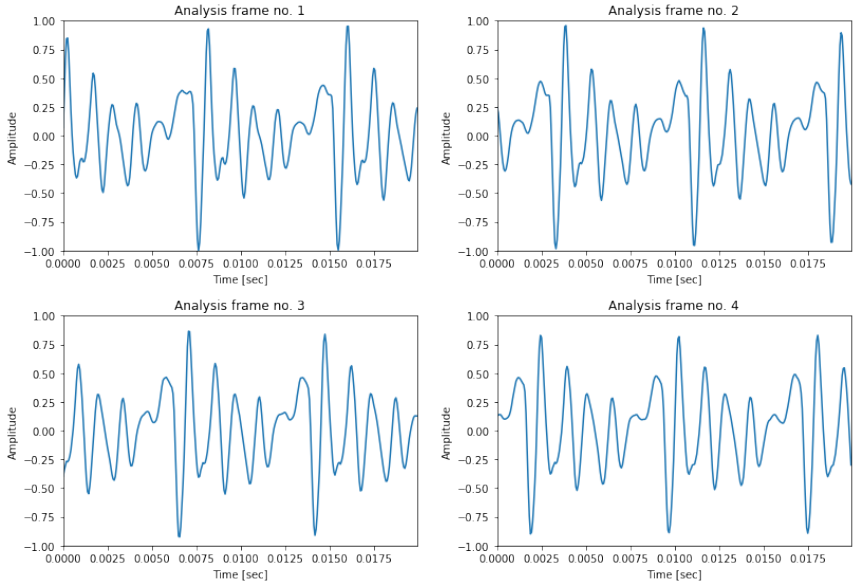
> **OBS T3.2** Dacă lungimea semnalului nu este egală cu un multiplu întreg al lungimii cadrului de analiză, ultimul cadru va fi analizat separat.

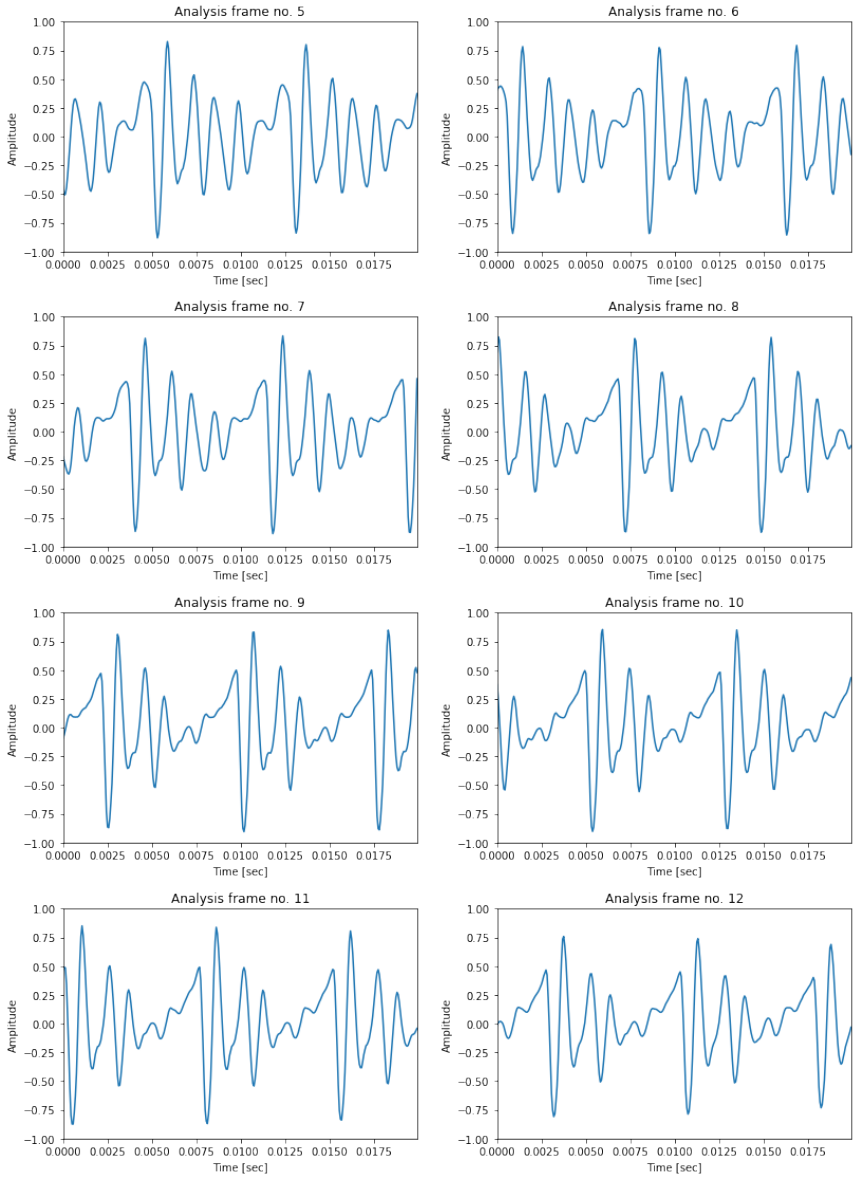
[6]:
```
# Calculăm numărul întreg de cadre din semnalul de intrare
number_of_frames = int(len(wav_data)/window_length)
print ("Number of frames: %d" %number_of_frames)
```

[6]:
```
Number of frames: 12
```

Extragem pe rând fiecare cadru de analiză din semnal și îl plotăm:

[7]:
```
# Creăm variabila de timp pentru axa oX a plotului
time_axis = np.arange(0, window_length ) \
        *1.00/sampling_frequency

for k in range(number_of_frames):
    # Extragem doar un cadru din semnal
    # Echivalent cu înmulțirea semnalului cu
    # o fereastră rectangulară
    # cu o întârziere egală cu i*window_length
    current_frame = wav_data[k*window_length:  \
            (k+1)*window_length]
    # Plot
    pl.figure()
    pl.plot(time_axis, current_frame)
    pl.title("Analysis frame no. %d" %(k+1))
    pl.xlabel('Time [sec]')
    pl.ylabel('Amplitude')
    pl.xlim((0, time_axis[-1]))
    pl.ylim((-1, 1))
```





Exercițiu T3.2.1 Modificați valorile lungimii cadrului de analiză și analizați rezultatele. Folosiți valorile 32, 64, 128 și 256 și afișați rezultatele pentru un singur cadru de analiză. ■

Exercițiu T3.2.2 Schimbați fișierul de intrare cu unul ce conține o consoană și afișați din nou rezultatele. ■

T3.3. Tipuri de ferestre de analiză

Exemplele anterioare au folosit (nu în mod explicit) o fereastră de analiză de tip rectangular. Acest lucru înseamnă că toate eșantioanele din fereastră au ponderi egale în cadrul analizei ulterioare. Însă, în practică, se folosesc mai degrabă ferestre cu ponderi variabile, precum Hamming, Hanning sau Blackman.[1] Ce este comun acestora, este faptul că eșantioanele din centrul ferestrei sunt ponderate unitar, iar cele de la extremități au ponderi ce tind spre 0. În afara ferestrei de analiză, ponderile sunt 0.

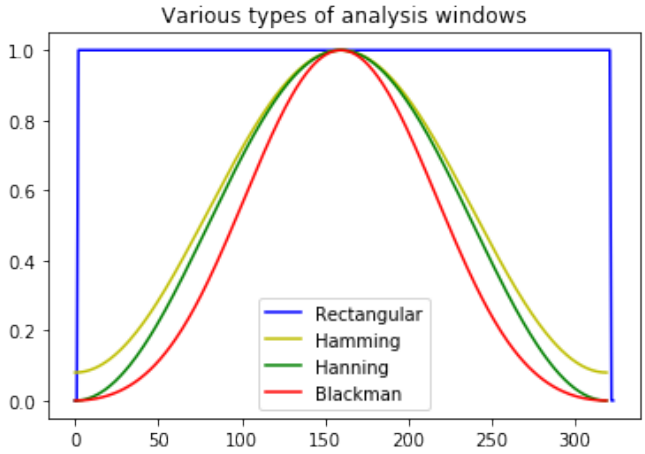
În secvența de cod de mai jos afișăm forma acestor ferestre folosindu-ne de funcțiile ale modulul SciPy: [2]

[10]:
```
# Importăm funcțiile aferente ferestrelor de analiză;
# boxcar este fereastra rectangulară
from scipy.signal import boxcar, hamming, hann, blackman

# Generăm o fereastră rectangulară puțin mai lungă pentru
# a putea vizualiza forma acesteia în mod corect.
boxcar_window = np.zeros(window_length+4)
boxcar_window[2:window_length+2] = boxcar(window_length)

# Generăm ferestrele Hamming, Hanning și Blackman
hamming_window = hamming(window_length)
hanning_window = hann(window_length)
blackman_window = blackman(window_length)

# Plotăm ferestrele
pl.title('Various types of analysis windows')
```

[1] https://docs.scipy.org/doc/scipy-0.19.1/reference/generated/scipy.signal.hamming.html
https://docs.scipy.org/doc/scipy-1.0.0/reference/generated/scipy.signal.hanning.html
https://docs.scipy.org/doc/scipy-0.16.1/reference/generated/scipy.signal.blackman.html

[2] https://www.scipy.org/

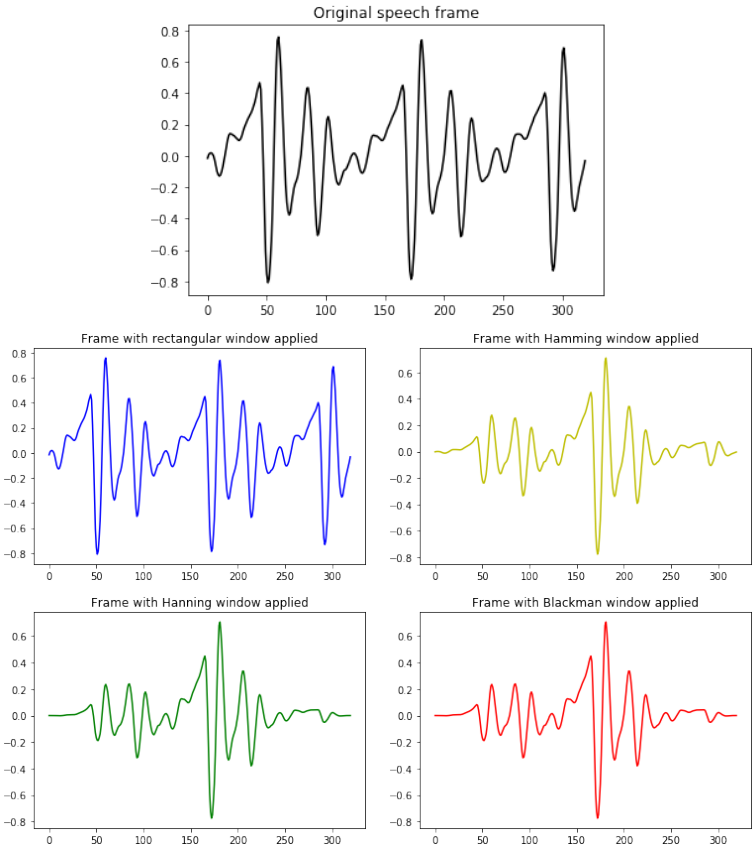
```
pl.plot(boxcar_window, 'b')
pl.plot(hamming_window, 'y')
pl.plot(hanning_window, 'g')
pl.plot(blackman_window, 'r')
pl.legend(['Rectangular', 'Hamming', 'Hanning', 'Blackman']);
```



Plotul de mai sus afișează doar ponderile corespunzătoare fiecărui tip de fereastră. Dacă dorim să observăm efectul ferestrelor asupra semnalului vocal trebuie să aplicăm aceste ferestre pe cadrele de analiză:

[11]:
```
# Selectăm un singur cadru din semnalul vocal, cadrul 2
k = 2
one_frame = wav_data[k*window_length: (k+1)*window_length]

# Înmulțim cadrul cu fiecare fereastră în parte.
# Înmulțirea se realizează element cu element.
boxcar_window = boxcar(window_length)
rectangular_frame = np.multiply(boxcar_window, current_frame)
hamming_frame = np.multiply(hamming_window, current_frame)
hanning_frame = np.multiply(hanning_window, current_frame)
blackman_frame = np.multiply(blackman_window, current_frame)
# Plot
pl.plot(current_frame, 'k')
pl.title('Original speech frame')
pl.figure()
pl.plot(rectangular_frame, 'b')
pl.title('Frame with rectangular window applied')
pl.figure()
```

```
pl.plot(hamming_frame, 'y')
pl.title('Frame with Hamming window applied')
pl.figure()
pl.plot(hanning_frame, 'g')
pl.title('Frame with Hanning window applied')
pl.figure()
pl.plot(hanning_frame, 'r')
pl.title('Frame with Blackman window applied')
```



Se poate observa efectul de ponderare al amplitudinii eșantioanelor din extremitățile ferestrei de analiză pentru ultimele 3 tipuri de fereastră.

T3.4. Răspunsul în frecvență al ferestrelor de analiză

În momentul de față s-ar putea să nu fie clar de ce e nevoie ca fiecare eșantion din fereastra de analiză să fie ponderat diferit. Astfel că introducem aici o noțiune necesară, însă puțin mai avansată, aceea de **răspuns în frecvență**. Această noțiune va fi reluată mai pe larg în cadrul tutorialului de analiză în frecvență și transformată Fourier.

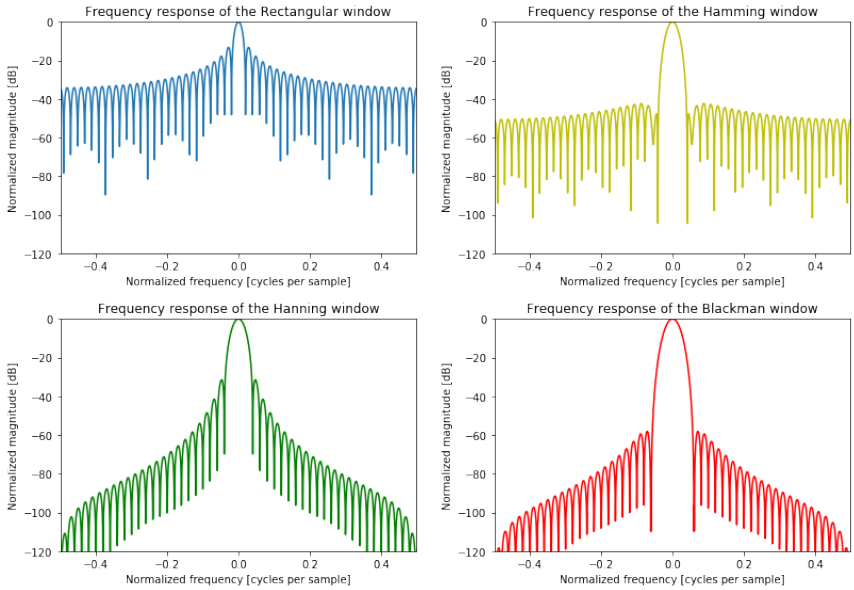
Am menționat anterior faptul că aplicarea unei ferestre de analiză asupra semnalului este echivalentă cu înmulțirea semnalului cu o serie de coeficienți ce sunt 0 în afara ferestrei. Acest lucru nu afectează doar amplitudinea semnalului, ci și spectrul său. În domeniu spectral, acest fenomen este cunoscut sub numele de **spectral leakage** și înseamnă că energia lobului spectral principal se va disipa și în lobii adiacenți. Fiecare tip de fereastră de analiză are un grad diferit de disipare, iar ploturile de mai jos exemplifică acest fapt:

> **OBS T3.3** Mai multe informații: `https://en.wikipedia.org/wiki/Spectral_leakage`

> **OBS T3.4** Nu e necesară înțelegerea codului de mai jos momentan!

[12]:
```
from scipy.fftpack import fft, fftshift
pl.figure()
window =  boxcar(51)
window_fft = fft(window, 2048) / (len(window)/2.0)
frequency_axis = np.linspace(-0.5, 0.5, len(window_fft))
response = 20 * np.log10(np.abs(fftshift(window_fft  \
        / abs(window_fft).max())))
pl.plot(frequency_axis, response)
pl.axis([-0.5, 0.5, -120, 0])
pl.title("Frequency response of the Rectangular window")
pl.ylabel("Normalized magnitude [dB]")
pl.xlabel("Normalized frequency [cycles per sample]")
```

```
pl.figure()
window =  hamming(51)
window_fft = fft(window, 2048) / (len(window)/2.0)
frequency_axis = np.linspace(-0.5, 0.5, len(window_fft))
response = 20 * np.log10(np.abs(fftshift(window_fft  \
        / abs(window_fft).max())))
pl.plot(frequency_axis, response, 'y')
pl.axis([-0.5, 0.5, -120, 0])
pl.title("Frequency response of the Hamming window")
pl.ylabel("Normalized magnitude [dB]")
pl.xlabel("Normalized frequency [cycles per sample]")
pl.figure()
window = hann(51)
window_fft = fft(window, 2048) / (len(window)/2.0)
frequency_axis = np.linspace(-0.5, 0.5, len(window_fft))
response = 20 * np.log10(np.abs(fftshift(window_fft  \
        / abs(window_fft).max())))
pl.plot(frequency_axis, response, 'g')
pl.axis([-0.5, 0.5, -120, 0])
pl.title("Frequency response of the Hanning window")
pl.ylabel("Normalized magnitude [dB]")
pl.xlabel("Normalized frequency [cycles per sample]")
pl.figure()
window = blackman(51)
window_fft = fft(window, 2048) / (len(window)/2.0)
frequency_axis = np.linspace(-0.5, 0.5, len(window_fft))
response = 20 * np.log10(np.abs(fftshift(window_fft  \
        / abs(window_fft).max())))
pl.plot(frequency_axis, response, 'r')
pl.axis([-0.5, 0.5, -120, 0])
pl.title("Frequency response of the Blackman window")
pl.ylabel("Normalized magnitude [dB]")
pl.xlabel("Normalized frequency [cycles per sample]");
```



Ar trebui să remarcăm faptul că fenomenul de spectral leakage este cel mai pronunțat pentru fereastra rectangulară. În aplicații practice, se folosește de cele mai multe ori fereastra Hamming datorită atenuării constante din afara lobului principal.

T3.5. Cadre de analiză suprapuse

În toate exemplele de mai sus am folosit cadre de analiză disjuncte. Acest lucru înseamnă că per global, eșantioanele din marginile ferestrei nu vor avea o influență egală cu cele centrale în analizele ulterioare, fapt ce nu este de dorit. Pentru a evita acest lucru trebuie să utilizăm cadre de analiză suprapuse, astfel încât toate eșantioanele sau cel puțin majoritatea lor să se afle la un moment dat în centrul ferestrei de analiză.

Factorul de suprapunere reprezintă gradul de suprapunere al cadrelor și poate fi ales în domeniul $p \in [0,1)$ (sau exprimat procentual între 0 și 100%), unde 0 înseamnă fără suprapunere, iar 1 este complet suprapus (nu are utilitate practică). Figura de mai jos afișează cazul unei suprapuneri de 0.33 (33%).

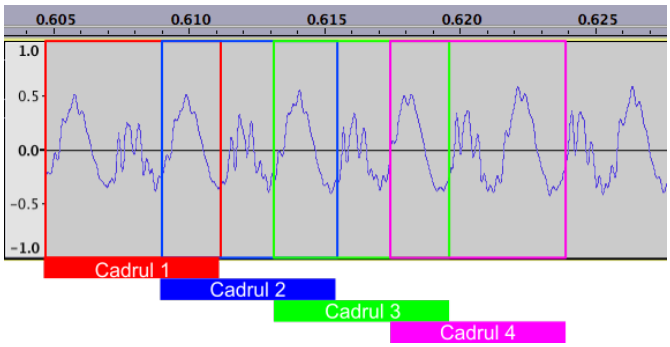


Fig.T3.1. Cadre de analiză cu factor de suprapunere 0.33 (33%)

Astfel că, vom schimba codul anterior pentru a putea utiliza și o variabilă ce reprezintă factorul de suprapunere al cadrelor, p:

[13]:
```
# Definim factorul de suprapunere
p = 0.2
# Stabilim o lungime a ferestrei egală cu 100 de eșantioane
window_length = 100
# Calculăm numărul de cadre
```

```
number_of_frames = int(len(wav_data)/((1-p)*window_length))
# Extragem cadrele suprapuse din semnal
for k in range(number_of_frames):
    current_frame = wav_data[int(k*(1-p)*window_length): \
            int((k*(1-p)+1)*window_length)]
```

Exercițiu T3.5.1 Modificați factorul de suprapunere și afișați rezultatele pentru primele 5 cadre de semnal vocal. ■

Ultimul cadru de analiză (dacă există)

În ambele cazuri, cu sau fără suprapunerea ferestrelor de analiză, dacă lungimea semnalului nu este egală cu un multiplu întreg al lungimii cadrului de analiză, trebuie să ținem cont și de ultimul cadru și să îl tratăm în mod individual:

[14]:
```
# Ultimul cadru de analiză
last_frame = wav_data[number_of_frames*window_length:]
```

T3.6. Concluzii

În acest tutorial am reușit să extragem din semnalul vocal de intrare câte un cadru de analiză, astfel încât să putem realiza analize ulterioare asupra semnalului, analize ce necesită ipoteza de staționaritate a semnalului. Am văzut, totodată, și efectul diferitelor tipuri de ferestre de analiză asupra semnalului vocal și modul în care acestea se comportă în domeniul frecvenței.

BIBLIOGRAFIE SUPLIMENTARĂ

- X. Huang, A. Acero, H.-W. Hon, ”Spoken Language Processing: A Guide to Theory, Algorithm, and System Development”, Prentice Hall, 2001
- M. Giurgiu, L. Peev, “Sinteza din text a semnalului vocal. Vol I.”, Editura Risoprint, Cluj-Napoca, 2006
- Paul Taylor, ”Text to speech synthesis”, Cambridge University Press, 2009

RESURSE MEDIA

- Wikiversity - ”Psycholinguistics/Models of Speech Production”, online: `https://en.wikiversity.org/wiki/Psycholinguistics/Models_of_Speech_Production`
- The Virtual Linguistics Campus, "PHO107 - Basic Segments of Speech (Vowels I)"
  `https://www.youtube.com/watch?v=xa5bG_wrK7s`,
  `https://www.youtube.com/watch?v=kB8PyODhC_8`
- Speech Graphics' Simone Articulation System
  `https://www.youtube.com/watch?v=wYwk07QM4rc`

T4 Detecția liniște-vorbire

În acest tutorial vom încerca să rezolvăm problema aparent simplă de a determina unde începe și unde se termină semnalul vocal. Această problemă este denumită și **voice activity detection** sau **endpoint detection** și este esențială pentru majoritatea aplicațiilor de prelucrare de semnal vocal.

Una dintre cele mai importante aplicații ale detecției liniște-vorbire este în cadrul aplicațiilor de transmisie a vorbirii (precum VoIP sau transmisiuni mobile). În cadrul acestora, pentru a reduce debitul de date, pe durata segmentelor de liniște se generează automat un zgomot aleator în loc de a transmite cadrele de semnal.

Înainte de a începe această analiză, trebuie să vizualizăm și să determinăm anumite caracteristici particulare ale semnalului vocal și a zgomotului. Astfel că, vom citi în primă fază un semnal ce conține atât voce, cât și liniște:

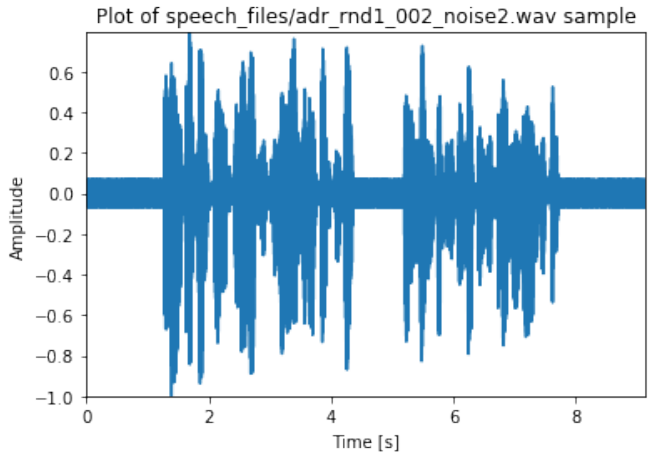
[1]:
```
import wave
import numpy as np
input_wav_file = 'speech_files/adr_rnd1_002_noise2.wav'
wav_struct = wave.open(input_wav_file, 'r')
# Extragem frecvența de eșantionare
sampling_frequency = wav_struct.getframerate()
# Citim eșantioanele
wav_bytes = wav_struct.readframes(-1)
# Facem conversia la valori întregi
wav_data = np.frombuffer(wav_bytes, dtype='int16')
wav_data_int = wav_data
# Închidem fișierul de intrare
wav_struct.close()
```

Să-l vizualizăm și să-l ascultăm:

[2]:
```
import matplotlib.pylab as pl
%matplotlib inline

# Normalizăm datele la [-1,1]
wav_data = wav_data/float(max(abs(wav_data)))
# Calculăm axa oY ca funcție de timp
time_axis = np.arange(0, len(wav_data)) \
        *1.00/sampling_frequency
# Calculăm durata totală a semnalului (în secunde)
duration = len(wav_data)*1.00/sampling_frequency
print ("Duration %f second" %duration)
# Afișăm semnalul
```

```
pl.plot(time_axis, wav_data)
pl.xlim([0, duration])
pl.ylim([min(wav_data), max(wav_data)])
pl.xlabel('Time [s]')
pl.ylabel('Amplitude')
pl.title("Plot of %s sample" %input_wav_file);
```

[2]:
```
Duration 9.140271 second
```



[3]:
```
import IPython
IPython.display.Audio(wav_data, rate=sampling_frequency)
```

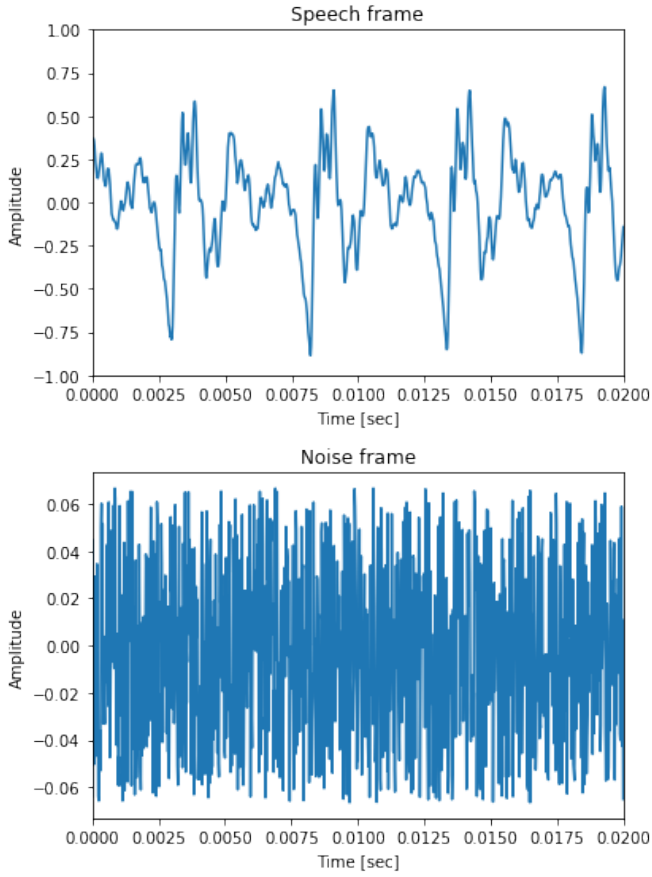
[3]:


Extragem un cadru de vorbire și unul de liniște și le afișăm împreună:

[4]:
```
window_length = int(20*1e-3*sampling_frequency)
# Cadru de vorbire
frame_no = 135
speech_frame = wav_data[window_length*frame_no:  \
        window_length*(frame_no+1)]
# Cadru de zgomot
frame_no = 1
noise_frame = wav_data[window_length*frame_no:  \
        window_length*(frame_no+1)]
```

[5]:
```
time_axis = np.arange(0, window_length) \
        *1.00/sampling_frequency
# Afișăm cadrul de vorbire
```

```
pl.figure()
pl.plot(time_axis, speech_frame)
pl.xlabel('Time [sec]')
pl.ylabel('Amplitude')
pl.xlim((0, time_axis[-1]))
pl.ylim((-1,1))
pl.title("Speech frame")

# Afișăm cadrul de liniște/zgomot
pl.figure()
pl.plot(time_axis, noise_frame)
pl.xlabel('Time [sec]')
pl.ylabel('Amplitude')
pl.xlim((0, time_axis[-1]))
pl.title("Noise frame");
```



OBS T4.1 Atenție la valorile de pe axa oY! Zgomotul are amplitudine mai mică.

Exercițiu T4.0.1 Ce se poate observa? Ce caracteristici ale semnalului ați folosi pentru a discrimina între voce și zgomot? ■

T4.1. Detecția liniște-vorbire folosind NTZ și energia

Vom introduce în cele ce urmează două măsuri de bază ce ne pot ajuta să facem detecția liniște-vorbire: *numărul de treceri prin zero* și *energia*. Aceste măsuri sunt calculate din forma de undă a semnalului în domeniul timp.

Numărul de treceri prin zero (NTZ) (en. *zero crossing rate* - ZCR) este definit ca fiind egal cu numărul de intersectări ale semnalului cu axa oX.

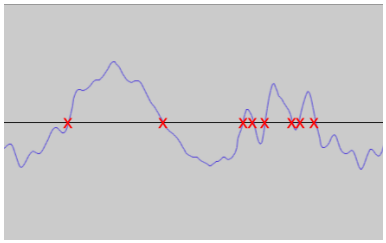


Fig.T4.1. Treceri prin zero ale formei de undă a semnalului

Să implementăm calculul acestei măsuri folosindu-ne de faptul că înmulțirea a două numere are rezultat negativ doar atunci când numerele au semne opuse:

[6]:
```
# Funcția de calcul a numărului de treceri prin zero
def ZCR(input_speech):
    zcr = 0
    for i in range(len(input_speech)-1):
      if input_speech[i]*input_speech[i+1] < 0:
        zcr+=1
    return zcr
```

Energia unui semnal discret este definită ca fiind suma amplitudinilor eșantioanelor semnalului:

$E = \sum_{n=-\infty}^{\infty} \left| x(n)^2 \right|$

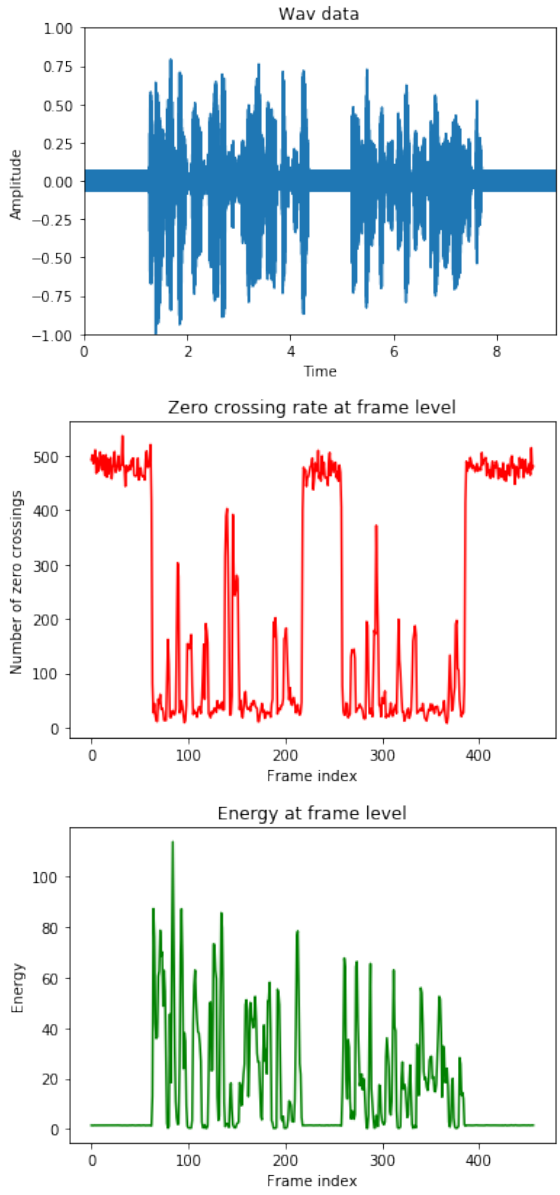
Să implementăm calculul energiei ca funcție:

[7]:
```
# Funcția de calcul a energiei semnalului
def energy (input_speech):
    ener = sum([sample**2 for sample in input_speech])
    return ener
```

Odată ce avem aceste funcții, putem să calculăm cele două caracteristici pentru semnalul citit anterior și să afișăm variația lor de-a lungul întregului semnal:

[8]:
```
# Lungimea ferestrei de analiză ca număr de eșantioane
window_length = int(20*1e-3*sampling_frequency)
# Factorul de suprapunere a cadrelor
p = 0
# Numărul de cadre din semnal
number_of_frames = int(len(wav_data)/((1-p)*window_length))
# Creăm doi vectori cu valori nule în care vom stoca valorile
# energiei și NTZ din fiecare cadru de semnal
frames_zcr = [0]*number_of_frames
frames_energy = [0]*number_of_frames
# Parcurgem cadrele de semnal și calculăm energia și NTZ
for k in range(number_of_frames):
    current_frame = wav_data[k*window_length:  \
            (k+1)*window_length]
    frames_zcr[k] = ZCR(current_frame)
    frames_energy[k] = energy(current_frame)
# Axa de timp oX
time_axis = np.arange(0, len(wav_data)) \
        *1.00/sampling_frequency
# Plot semnal
pl.figure()
pl.plot(time_axis, wav_data)
pl.title('Wav data')
pl.xlabel('Time')
pl.ylabel('Amplitude')
pl.xlim([0, time_axis[-1]])
pl.ylim([-1, 1])
# Plot NTZ
pl.figure()
pl.plot(frames_zcr, 'r')
pl.title('Zero crossing rate at frame level')
pl.xlabel('Frame index')
pl.ylabel('Number of zero crossings')
```

```
# Plot energie
pl.figure()
pl.plot(frames_energy, 'g')
pl.title('Energy at frame level')
pl.xlabel('Frame index')
pl.ylabel('Energy');
```



În ploturile de mai sus putem observa că există o diferență clară între NTZ și energie pentru zonele de liniște/zgomot de fundal și cele de vorbire. Diferențele sunt rezumate în tabelul de mai jos:

	Zgomot/liniște	**Vorbire**
NTZ	mare	mic
Energie	mică	mare

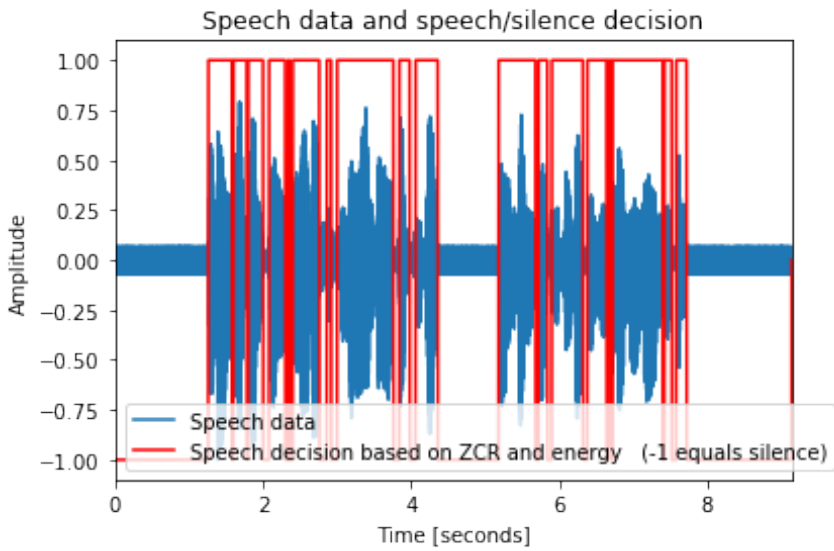
Pornind de la aceste observații, putem crea un prim algoritm de detecție liniște-vorbire. Deoarece NTZ și energia variază de la un semnal la altul, nu putem determina praguri absolute de detecție a liniștii și vorbirii. Astfel că, aceste praguri vor fi determinate direct din semnalul analizat.

Cel mai simplu algoritm de calcul este bazat pe estimarea numărului mediu NTZ și valoarea medie a energiei din primele 100 msec ale semnalului și de a face presupunerea că acest interval conține doar liniște/zgomot de fond. Pe baza acestor valori se stabilește mai apoi un prag proporțional de detecție a vorbirii/zgomotului. Și anume: dacă numărul de treceri prin zero dintr-un cadru de semnal scade sub acest prag, iar energia este peste pragul calculat, cadrul conține zgomot de fond/liniște. Prin inversarea condițiilor, se obține cadrul de vorbire.

Implementăm acest algoritm în cele ce urmează:

[9]:
```
# Calculăm valoarea medie a NTZ și energiei pentru
# primele 100msec de semnal
# NOTA: În această situație un cadru are lungimea de 20msec,
# deci trebuie să folosim primele 5 cadre
zcr_mean = sum(frames_zcr[:5])/5
energy_mean = sum(frames_energy[:5])/5
# Stabilim o valoare a pragului proporțional (60%)
# din valorile calculate anterior
threshold = 0.6
# Calculăm valorile pragurilor proporționale pentru
# NTZ și energie
zcr_threshold = threshold*zcr_mean
energy_threshold = threshold*energy_mean
# Inițializăm un vector de decizie voce/zgomot
speech_decision = np.zeros(number_of_frames)
# Trecem prin toate cadrele de semnal și determinăm
# dacă valorile NTZ  și energie pentru cadrul respectiv
# sunt sub sau peste pragurile determinate anterior
for k in range(number_of_frames):
    if frames_zcr[k] < zcr_threshold  \
            and frames_energy[k] > energy_threshold:
```

```
        speech_decision[k] = 1
    else:
        speech_decision[k] = -1
# Creăm un vector de lungime egală cu semnalul de intrare
# pentru a putea afișa mai bine rezultatele
speech_decision_plot = np.zeros(len(wav_data))
for k in range(number_of_frames):
    speech_decision_plot[k*window_length: \
            (k+1)*window_length] = speech_decision[k]
# Afișăm rezultatele
time_axis = np.arange(0, len(wav_data)) \
        *1.00/sampling_frequency
pl.plot(time_axis, wav_data)
pl.plot(time_axis, speech_decision_plot,'r')
pl.title('Speech data and speech/silence decision')
pl.xlabel('Time [seconds]')
pl.ylabel('Amplitude')
pl.xlim([0, time_axis[-1]])
pl.ylim([-1.1, 1.1])
pl.legend(['Speech data', 'Speech decision based on ZCR  \
        and energy (-1 equals silence)']);
```



Putem observa că acest algoritm simplu are anumite probleme, iar deciziile sale nu sunt în conformitate cu ceea ce ne-am fi așteptat să obținem.

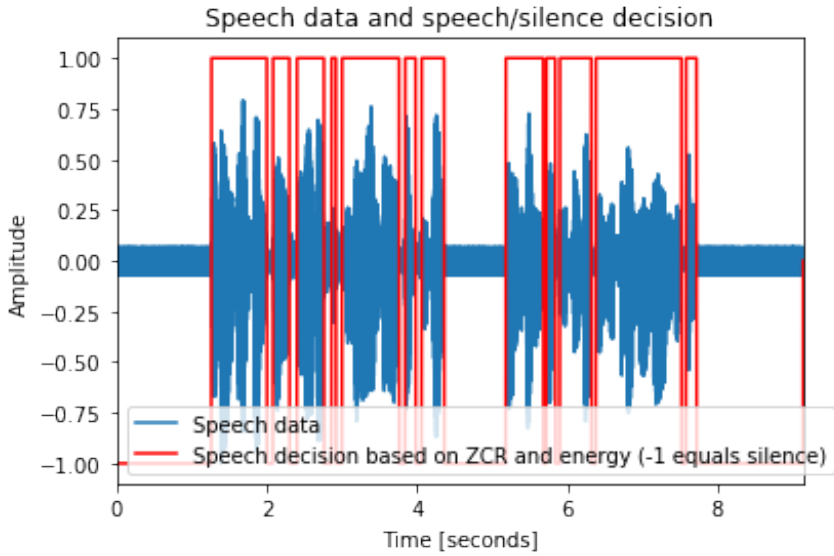
Acest lucru se datorează faptului că anumite segmente de vorbire au caracteristici similare cu cele ale zgomotului. De exemplu, consoanele *s* și *ș* au forma de undă foarte apropiată de cea a zgomotului alb gausian. Vom discuta în tutorialul următor mai multe despre caracteristicile segmentelor sonore individuale.

Exercițiu T4.1.1 Modificați valoarea pragului și analizați rezultatele. ■

Exercițiu T4.1.2 De ce sunt detectate cadre de voce în zonele de liniște? ■

O primă soluție la aceste probleme este de a media decizia de-a lungul mai multor cadre, deoarece e puțin probabil ca un singur cadru de semnal să fie diferit de cele din jur. Astfel că vom verifica dacă decizia pentru fiecare cadru e diferită de cele două adiacente. În caz afirmativ, modificăm decizia inițială:

[10]:
```
# Dacă există un singur cadru de voce înconjurat de liniște
# sau invers, vom modifica decizia pentru acel cadru
for k in range(1, number_of_frames-1):
    if (speech_decision[k] != speech_decision[k-1]) and \
            (speech_decision[k] != speech_decision[k+1]):
        speech_decision[k] = speech_decision[k+1]
# Creăm un vector de lungime egală cu lungimea semnalului
# de intrare pentru afișare
speech_decision_plot = np.zeros(len(wav_data))
for k in range(number_of_frames):
    speech_decision_plot[k*window_length: \
            (k+1)*window_length] = speech_decision[k]

# Afișăm rezultatele
time_axis = np.arange(0, len(wav_data)) \
        *1.00/sampling_frequency
pl.plot(time_axis, wav_data)
pl.plot(time_axis, speech_decision_plot,'r')
pl.title('Speech data and speech/silence decision')
pl.xlabel('Time [seconds]')
pl.ylabel('Amplitude')
pl.xlim([0, time_axis[-1]])
pl.ylim([-1.1, 1.1])
pl.legend(['Speech data', 'Speech decision based on ZCR \
         and energy (-1 equals silence)']);
```



Nici acest lucru nu a generat rezultate mult mai bune... Un algoritm mai complex și eficient în domeniul timp este Rabiner-Sambur. [1]

Notă: Pentru ca algoritmul să funcționeze corect, la fel ca și mai sus, primele 100msec de semnal trebuie să conțină doar liniște.

Exercițiu T4.1.3 Rulați din nou algoritmul de mai sus pentru fișierul *adr_rnd1_001.wav* din directorul *speech_files/*. Ce se întâmplă? ■

Exercițiu T4.1.4 Cum ați rezolva problemele algoritmului nostru simplu? ■

Exercițiu T4.1.5 Implementați algoritmul Rabiner-Sambur algorithm și evaluați rezultatele. ■

[1] `https://ia800309.us.archive.org/15/items/bstj54-2-297/bstj54-2-297.pdf`

T4.2. Detecția liniște vorbire folosind caracteristici spectrale probabilistice

Ca alternativă la implementarea metodei Rabiner-Sambur, putem să utilizăm un algoritm de detecție liniște vorbire deja implementat, cum ar fi cel disponibil la această adresă `https://github.com/wiseman/py-webrtcvad/`. Algoritmul folosește metode probabilistice de estimare a caracteristicilor spectrale ale vocii și zgomotului. Întelegerea modului de funcționare a acestuia la momentul actual nu este necesară, ci dorim doar să exemplificăm modul în care detecția liniște vorbire ar trebui să aibă loc în practică.

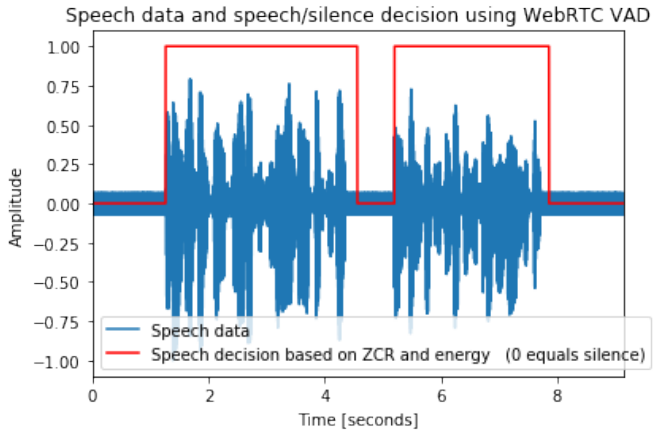
Pentru a instala modulul vom folosi utilitarul `pip`:

[11]:
```
!pip install webrtcvad
```

[11]:
```
Collecting webrtcvad
Installing collected packages: webrtcvad
Successfully installed webrtcvad-2.0.10
```

[12]:
```
# Importăm modulul
import webrtcvad
# Instanțiem un obiect de tip VAD
vad = webrtcvad.Vad()
# Setăm modul de operare (v. documentația)
vad.set_mode(3)
# Parcurgem cadru cu cadru semnalul și reținem decizia VAD
speech_decision = np.zeros(number_of_frames)
for k in range(number_of_frames):
    current_frame = wav_data_int[k*window_length:  \
            (k+1)*window_length]
    speech_decision[k] = int(vad.is_speech(current_frame,  \
            sampling_frequency))
    #print (vad.is_speech(current_frame, sampling_frequency))
# Creăm un vector de lungime egală cu semnalul de intrare
# pentru a putea afișa mai bine rezultatele
speech_decision_plot = np.zeros(len(wav_data))
```

```
for k in range(number_of_frames):
    speech_decision_plot[k*window_length: \
            (k+1)*window_length] = speech_decision[k]
# Afișăm rezultatele
time_axis = np.arange(0, len(wav_data)) \
        *1.00/sampling_frequency
pl.plot(time_axis, wav_data)
pl.plot(time_axis, speech_decision_plot,'r')
pl.title('Speech data and speech/silence decision \
         using WebRTC VAD')
pl.xlabel('Time [seconds]')
pl.ylabel('Amplitude')
pl.xlim([0, time_axis[-1]])
pl.ylim([-1.1, 1.1])
pl.legend(['Speech data', 'Speech decision based on ZCR  \
         and energy (0 equals silence)']);
```



Se poate observa că spre deosebire de algoritmul nostru simplu, acesta din urmă realizează o detecție corectă a segmentelor de liniște și vorbire. Studiul algoritmilor VAD este în continuare de mare interes în comunitarea științifică, în special în cazurile în care zgomotul de fond acoperă în proporție foarte mare semnalul vocal.

T4.3. Concluzii

Tutorialul 4 a introdus noțiunile de număr de treceri prin zero și energie și am văzut modul în care acestea pot fi utilizate pentru discriminarea dintre segmentele de voce și cele de zgomot/liniște de fundal. Cu toate că aceste măsuri în sine pot realiza o oarecare discriminare liniște-vorbire, în aplicații practice, datorită variabilității mari a semnalelor și a tipurilor de zgomot, simpla analiză a formei de undă a unui semnal vocal nu este suficientă pentru dezvoltsarea unui algoritm robust.

BIBLIOGRAFIE SUPLIMENTARĂ

- X. Huang, A. Acero, H.-W. Hon, "Spoken Language Processing: A Guide to Theory, Algorithm, and System Development", Prentice Hall, 2001
- Benesty et al, "Springer Handbook of Speech Processing", Springer, 2008

RESURSE MEDIA

- Tom Backstrom, "Voice Activity Detection", Speech Processing course, 2015, online: `https://mycourses.aalto.fi/pluginfile.php/146209/mod_resource/content/1/slides_07_vad.pdf`
- ETSI Standard - Voice Activity Detection (VAD) for Enhanced Full Rate (EFR) speech traffic channels, online: `http://www.etsi.org/deliver/etsi_i_ets/300700_300799/300730/01_20_103/ets_300730e01c.pdf`
- Md Sahidullah, Student Member, Goutam Saha, "Comparison of Speech Activity Detection Techniques for Speaker Recognition", arXiv 1210.0297, online: `https://arxiv.org/pdf/1210.0297.pdf`

T5 Detecția F0 în domeniul timp

În cadrul acestui tutorial vom introduce metode de estimare a **frecvenței fundamentale** a semnalului vocal în domeniul timp. Aceste metode se bazează strict pe analiza periodicității formei de undă.

T5.1. Frecvența fundamentală a vorbirii

Într-un tutorial anterior am prezentat modelele de producere a semnalului vocal și componentele fiziologice ce intervin în vorbire. Totodată, am determinat și caracteristicile și clasificările segmentelor vocale în funcție de diverși factori. Printre cele mai importante clasificări, se află și cea de clasificare a semnalelor în sonor-nesonor. Aceasta se referă la prezența sau absența periodicității semnalului și este determinată de utilizarea sau nu a corzilor vocale în modularea fluxului de aer în timpul vorbirii.

Corzile vocale reprezintă două pliuri simetrice ale membranei laringelui. În timpul fonației, înălțimea sunetului rezultat este controlată de poziția acestor pliuri și de gradul de închidere/deschidere a orificiului delimitat de ele. Periodicitatea sunetului este însă dată de vibrația liberă a unui segment al corzilor vocale și nu de mișcarea controlată a acestora cu ajutorul mușchilor conecși așa cum este de cele mai multe ori considerat.

Această oscilație a corzilor vocale determină periodicitatea semnalului vocal pe segmentele sonore. Măsura obiectivă a acestei periodicități este definită de **frecvența fundamentală (F0)**. Segmentele sonore din vorbire sunt în principal fonemele vocalice (de ex. pentru limba română, ”a”, ”ă”, ”â”, ”e”, ”i”, ”o” și ”u”). În limba engleză, însă, există o diferență între termenii ”pitch” și ”fundamental frequency”. ”Pitch” se referă la frecvența fundamentală percepută, iar ”fundamental frequency” se referă la frecvența fundamentală obiectiv calculată din semnalul vocal. Această diferență vine, printre altele și din faptul că, dacă într-un semnal audio se înlătură primele câteva armonici, frecvența percepută de ascultător va fi tot cea fundamentală, deși în spectru aceasta nu este prezentă, timbrul sunetului fiind însă ușor diferit.

Astfel că, vom defini **frecvența fundamentală (F0)** ca fiind frecvența de oscilație a corzilor vocale și cea mai joasă frecvență prezentă în spectrul semnalului vocal. Deoarece frecvența fundamentală este rezultatul caracteristicilor fiziologice ale unei persoane, valoarea sa medie variază de la un vorbitor la altul, iar diferențe majore există și între categoriile de vorbitori. La femei, valoarea medie a F0 este de aproximativ 210Hz, iar la bărbați este de 120Hz. În cazul copiilor, valoare medie F0 urcă până la 300Hz. A nu

se face confunzia între F0 din timpul vorbirii cu cea din timpul cântatului. Un cântăreț trebuie să modifice constant frecvența fundamentală pentru a putea reda notele muzicale.

În cadrul aplicațiilor de prelucrare de voce, frecvența fundamentală este extrem de importantă, iar cei mai mulți algoritmi de codare a semnalului vocal sau de procesare tratează în mod independent fluxul de date F0 față de răspunsul tractului vocal. Determinarea cu acuratețe a F0 devine astfel un domeniu de studiu larg, iar metodele de determinare a acestei frecvențe sunt multiple. De exemplu, în cadrul codorului CELP (Code Excited Linear Prediction) utilizat în codarea GSM, posibilitatea estimării F0 folosind un set redus de impulsuri face ca această codare să fie extrem de eficientă.

T5.2. Detecția frecvenței fundamentale în domeniul timp

Datorită caracterului cvasi-periodic al semnalelor sonore, cele mai simple metode de determinare a F0 din semnalul vocal se referă la estimarea acestei periodicități din forma de undă. **Periodicitatea** unui semnal este dată de intervalul de timp după care valorile eșantioanelor semnalului se repetă. Acest interval este denumit perioada semnalului (T_0) și este egală cu $1/F_0$. Cel mai simplu exemplu de semnal periodic este o funcție sinus. Spunem însă că semnalul vocal este cvasi-periodic deoarece chiar și în segmentele sonore, datorită fenomenelor de coarticulare (trecerea de la un fonem la altul și modificarea poziției tractului vocal în acest proces), periodicitatea semnalului nu este pură, existând oarecare diferențe între perioade consecutive ale sale.

Având toate aceste informații la dispoziție, să încercăm sp determinăm periodicitatea semnalului vocal pornind doar de la forma de undă a acestuia. În primul rând, pentru comparație, vom citi două segmente vocale, o consoană și o vocală.

[1]:
```
import wave
import numpy as np
# Citim o vocală
input_wav_vowel = 'speech_files/a.wav'
wav_struct_vowel = wave.open(input_wav_vowel, 'r')
# Frecvența de eșantionare
sampling_frequency_vowel = wav_struct_vowel.getframerate()
# Eșantioanele semnalului
wav_bytes_vowel = wav_struct_vowel.readframes(-1)
# Conversie la valori întregi
wav_data_vowel = np.frombuffer(wav_bytes_vowel, \
        dtype='int16')
# Normalizare a datelor în intervalul [-1,1]
wav_data_vowel = wav_data_vowel \
        /float(max(abs(wav_data_vowel)))
# Închidem fluxul de intrare
```

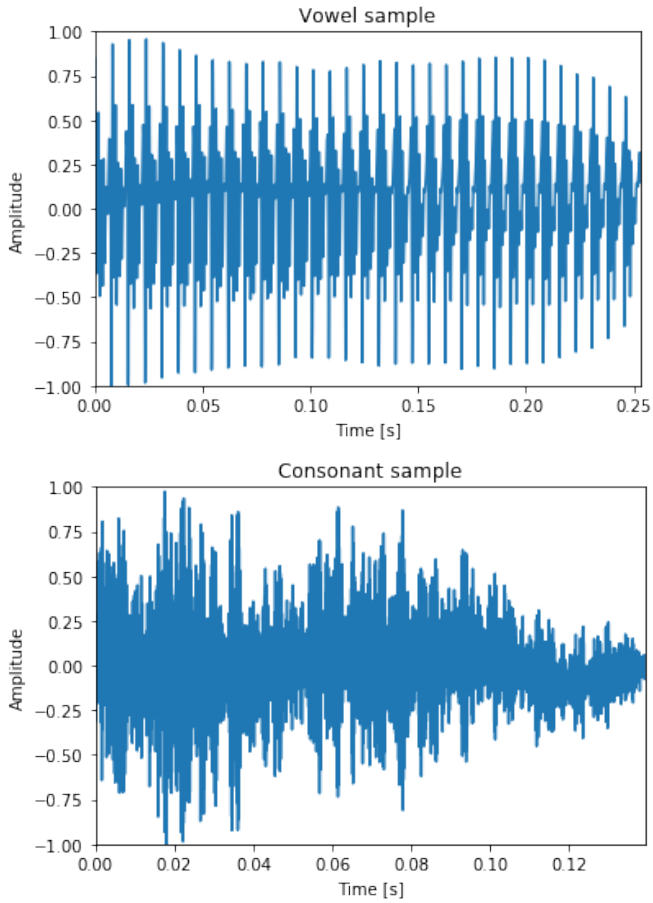
```
wav_struct_vowel.close()

# Citim  o consoană
input_wav_consonant = 'speech_files/s.wav'
wav_struct_consonant = wave.open(input_wav_consonant, 'r')
# Frecvența de eșantionare
sampling_frequency_consonant =  \
        wav_struct_consonant.getframerate()
# Eșantioanele semnalului
wav_bytes_consonant = wav_struct_consonant.readframes(-1)
# Conversie la valori întregi
wav_data_consonant = np.frombuffer(wav_bytes_consonant,  \
        dtype='int16')
# Nomalizare a datelor în intervalul [-1,1]
wav_data_consonant = wav_data_consonant \
        /float(max(abs(wav_data_consonant)))
# Închidem fluxul de intrare
wav_struct_consonant.close()
```

Afișăm și ascultăm datele:

[2]:
```
import matplotlib.pylab as pl
%matplotlib inline

# Plot vocală
# Durata semnalului
duration_vowel = len(wav_data_vowel) \
        *1.00/sampling_frequency_vowel
print ("Duration %f second" %duration_vowel)
time_axis_vowel = np.arange(0, len(wav_data_vowel)) \
        *1.00/sampling_frequency_vowel
pl.plot(time_axis_vowel, wav_data_vowel)
pl.xlim([0, duration_vowel])
pl.ylim([-1, 1])
pl.xlabel('Time [s]')
pl.ylabel('Amplitude')
pl.title("Vowel sample" )

# Plot consoană
# Durata semnalului
duration_consonant = len(wav_data_consonant) \
        *1.00/sampling_frequency_consonant
```

```
print ("Duration %f second" %duration_consonant)
pl.figure()
time_axis_consonant = np.arange(0,\
  len(wav_data_consonant))*1.00/sampling_frequency_consonant
pl.plot(time_axis_consonant, wav_data_consonant)
pl.xlim([0, duration_consonant])
pl.ylim([-1, 1])
pl.xlabel('Time [s]')
pl.ylabel('Amplitude')
pl.title("Consonant sample");
```

[2]:
```
Duration 0.253063 second
Duration 0.139252 second
```



[3]:
```
import IPython
IPython.display.Audio(wav_data_vowel, \
        rate=sampling_frequency_vowel)
```

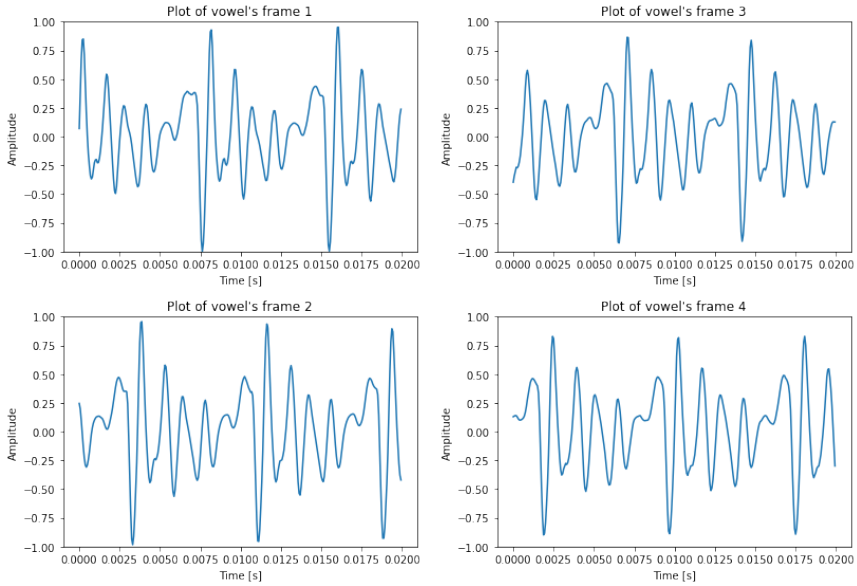
[3]:

[4]:
```
IPython.display.Audio(wav_data_consonant,  \
        rate=sampling_frequency_consonant)
```

[4]:

Să vizualizăm câteva cadre individuale din cele 2 semnale:

[5]:
```
##################################
# Cadrele vocalei

# Fereastra de analiză de 20msec fără suprapunere
window_length_vowel = int(20*1e-3*sampling_frequency_vowel)
p = 0
# Calculăm numărul de cadre
number_of_frames_vowel = int(len(wav_data_vowel) \
        /((1-p)*window_length_vowel))
# Variabila pentru axa timpului
time_axis_vowel = np.arange(0, window_length_vowel) \
        *1.00/sampling_frequency_vowel
# Afișăm doar primele 4 cadre
for k in range(4):
    current_frame = wav_data_vowel[int(k *(1-p))  \
            *window_length_vowel: int((k*(1-p)+1)) \
            *window_length_vowel]
    pl.figure()
    pl.plot(time_axis_vowel, current_frame)
    pl.ylim([-1, 1])
    pl.xlabel('Time [s]')
    pl.ylabel('Amplitude')
    pl.title("Plot of vowel's frame %i" %(k+1))
```



Din ploturile de mai sus putem observa că forma de undă este cvasi-periodică și că perioada sa fundamentală (T0) este în jur de 0.0075 secunde. Aceasta înseamnă că frecvența fundamentală este egală cu:

$$F\_0 = \frac{1}{T_0} = \frac{1}{0.0075} \approx 133[Hz] \tag{T5.2.1}$$

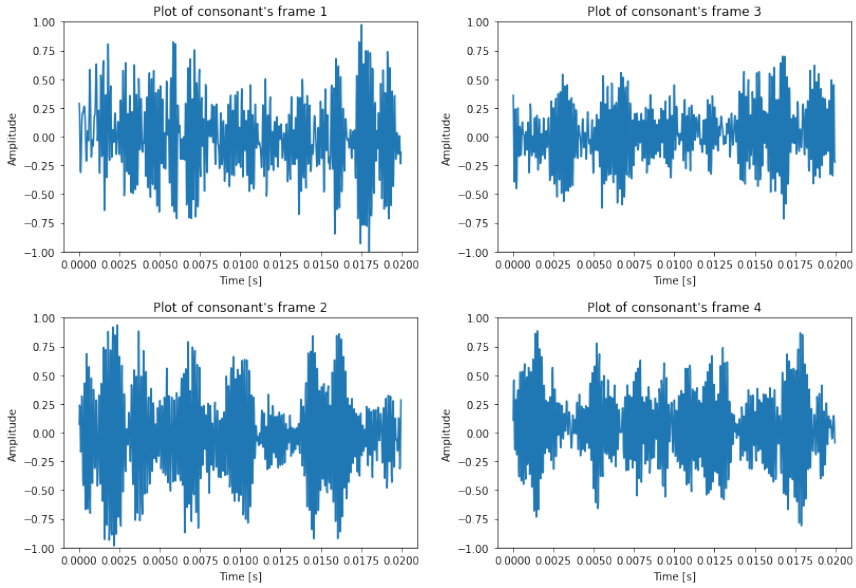
Exercițiu T5.2.1 Schimbați semnalul de intrare la `speech_files/e.wav`. ■

Exercițiu T5.2.2 Se mai poate determina F0 folosind doar forma de undă? ■

Și acum vizualizăm consoana:

[6]:
```
################################
# Cadrele consoanei

# Fereastra de analiză de 20msec fără suprapunere
window_length_consonant = \
         int(20*1e-3*sampling_frequency_consonant)
p = 0
# Calculăm numărul de cadre
number_of_frames_consonant = int(len(wav_data_consonant) \
        /((1-p)*window_length_consonant))
```

```
# Variabila pentru axa timpului
time_axis_consonant = np.arange(0, window_length_consonant) \
        *1.00/sampling_frequency_consonant
# Afișăm doar primele 4 cadre
for k in range(4):
    current_frame = wav_data_consonant[int(k \
            *(1-p)*window_length_consonant):int((k*(1-p)+1) \
            *window_length_consonant)]
    pl.figure()
    pl.plot(time_axis_consonant, current_frame)
    pl.ylim([-1, 1])
    pl.xlabel('Time [s]')
    pl.ylabel('Amplitude')
    pl.title("Plot of consonant's frame %d" %(k+1))
```



Exerciţiu T5.2.3 Se poate determina perioada fundamentală sau F0 în cazul consoanei? ■

Următoarele secțiuni vor prezenta două funcții diferite utilizate pentru a estima periodicitatea (sau lipsa acesteia) unui semnal vocal, precum și metode de îmbunătățire a estimării F_0.

T5.2.1 Funcția de autocorelație

Funcția de autocorelație este definită ca:

$$r(m) = \frac{1}{N} \sum_{n=1}^{N-m} x(n) * x(n+m),\ m = \overline{1,N} \tag{T5.2.2}$$

și poate fi folosită pentru a determina dacă un semnal este periodic prin calcularea similarității semnalului cu versiuni întârziate ale sale. Dacă semnalul este periodic cu perioada m, valoarea similarității la întârzierea $t = m$ va avea un maxim.

Funcția de mai jos implementează această formulă și returnează valoarea funcției de autocorelație pentru un semnal de intrare:

[7]:
```
def compute_autocorrelation(input_frame):
    sample_length =  len(input_frame)
    autocor = np.zeros(sample_length)
    for m in range(sample_length):
        for j in range(sample_length-m):
            autocor[m] += input_frame[j]*input_frame[j+m]
    return 1.0/sample_length * autocor
```

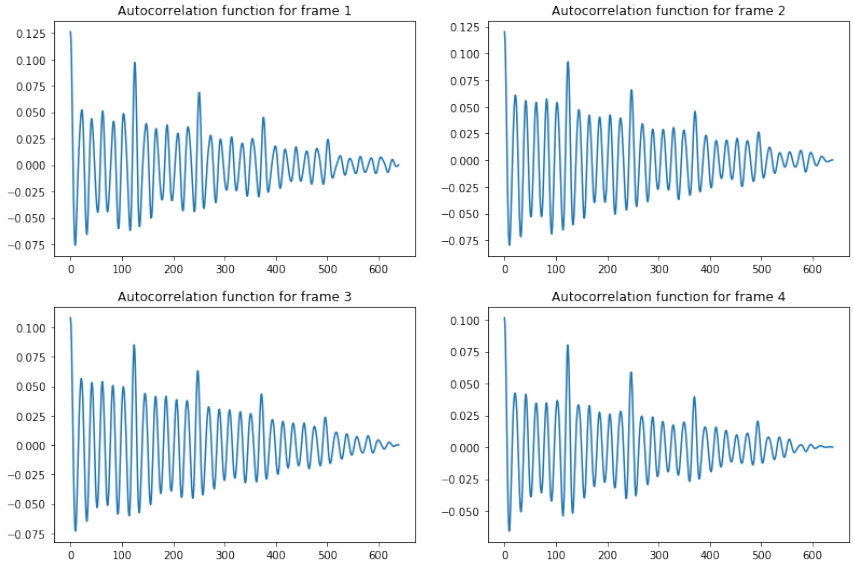
O alternativă este să utilizăm funcția disponibilă în modulul NumPy și de a reține doar ultima jumătate a valorilor acesteia, funcția fiind simetrică:

[8]:
```
def compute_autocorrelation_2(x):
    result = np.correlate(x, x, mode='full')
    return result[int(result.size/2):]
```

Să afișăm valorile funcției de autocorelație pentru semnalele citite. Vom folosi o fereastră de analiză de 40msec pentru a vizualiza mai bine maximele funcției:

[9]:
```
# Fereastra de 40msec
window_length_vowel = int(40*1e-3*sampling_frequency_vowel)
p = 0

# Numărul de cadre de analiză
number_of_frames = int(len(wav_data_vowel) \
        /window_length_vowel)

# Calculăm funcția de autocorelație pentru primele 4 cadre
for k in range(4):
    current_frame = wav_data_vowel[int(k *(1-p))  \
            *window_length_vowel: int((k*(1-p)+1)) \
```

```
            *window_length_vowel]
    frame_autocor = compute_autocorrelation(current_frame)
    pl.figure()
    pl.plot(frame_autocor)
    pl.title('Autocorrelation function for frame %d' %(k+1))
```



Uitându-ne la definiția funcției de autocorelație, precum și la ploturi, ar trebui să fie evident faptul că funcția prezintă maxime la intervale egale cu perioada fundamentală (T_0).

Pentru a determina F_0, trebuie să identificăm aceste maxime și să le convertim în domeniul frecvență (Hz).

Trebuie menționat faptul că funcția de autocorelație este o funcție de timp și că valorile indicate de maxime pentru semnale discrete sunt multipli ai perioadei de eșantionare T_s. Astfel că, ploturile de mai sus ar trebui să aibă timpul ca unitate de măsură pe axa oX.

Exercițiu T5.2.4 Afișați funcția de autocorelație pentru câteva cadre ale vocalei folosind timpul ca unitate de măsură pe axa oX ■

Vom urma un exemplu pentru ca lucrurile să fie mai clare:

Să presupunem că frecvența de eșantionare este de 16kHz, fiecare valoare a funcție de autocorelație reprezintă o întârziere egală cu

$$1/16000 = 62.5 \text{ nanosecunde.}$$

Să presupunem că primul maxim al funcției este observat la indexul 110. Aceasta înseamnă că perioada fundamentală a cadrului analizat este:

$$index * 1/F_s = 110 * 1/16000 = 6.875 \text{ microsecunde}$$

de unde rezultă:

$$F_0 = 1/T_0 = 16000/110 \approx 146\ Hz$$

Exercițiu T5.2.5 Ce valoare are F_0 pentru vocala citită estimat de pe grafic? ■

Exercițiu T5.2.6 Modificați lungimea ferestrei de analiză și examinați rezultatele. Folosiți valorile: 32, 128, 512, 1024, 2048. Se modifică valoarea F_0? ■

Exercițiu T5.2.7 Există vreo restricție asupra lungimii ferestrei de analiză determinată de funcția de autocorelație sau F_0? ■

Exercițiu T5.2.8 Având un set de semnale eșantionate la 16kHz și știind că valorile F_0 pentru majoritatea vorbitorilor sunt curprinse între 50-400Hz, calculați durata minimă a ferestrei de analiză, astfel încât funcția de autocorelație să prezinte cel puțin 2 maxime. ■

Exercițiu T5.2.9 Afișați funcția de autocorelație și pentru consoană. ■

Exercițiu T5.2.10 Se poate determina F_0 acum? Ce valoare are? ■

T5.2.2 AMDF

Funcția **Average Magnitude Difference Function (AMDF)** este definită ca:

$$AMDF(m) = \frac{1}{N}\sum_{n=1}^{N-m} |x(n) - x(n+m)|,\ m = \overline{1,N} \tag{T5.2.3}$$

Se poate observa că în loc de înmulțirea versiunilor întârziate ale semnalului, funcția AMDF calculează diferența în modul ale acestor întârzieri.

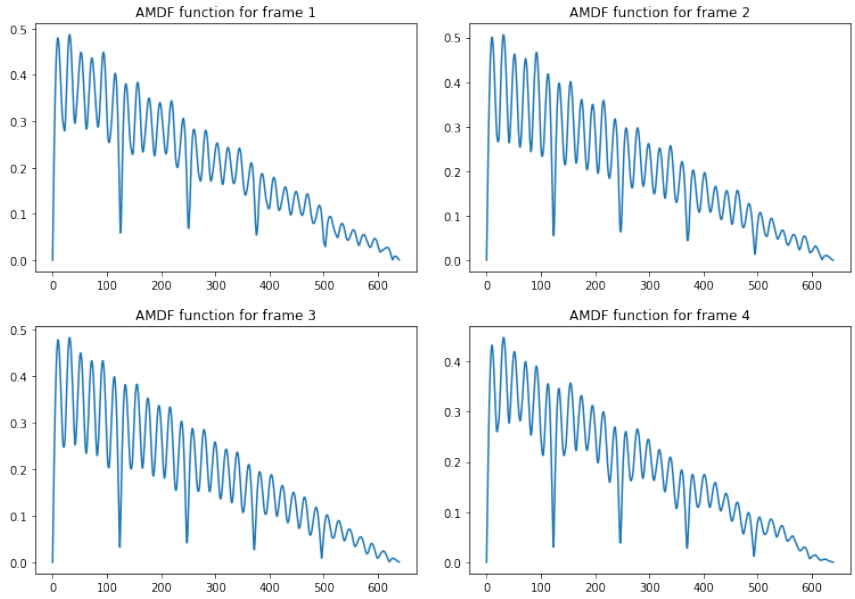
În practică acest lucru este foarte util în calculul F_0 pe dispozitive cu resurse limitate unde se dorește evitarea complexității calculului operațiilor de înmulțire mult mai complexe din punct de vedere computațional.

Implementăm și funcția AMDF:

[10]:
```
def compute_amdf(input_frame):
  sample_length =  len(input_frame)
  amdf = np.zeros(len(input_frame))
  for m in range(sample_length):
    for j in range(sample_length-m):
      amdf[m] += abs(input_frame[j] - input_frame[j+m])
  return 1.00/sample_length*amdf
```

Afișăm funcția AMDF aplicată asupra primelor 4 cadre de semnal:

[11]:
```
for k in range(4):
      current_frame = wav_data_vowel[int(k *(1-p))  \
            *window_length_vowel: int((k*(1-p)+1)) \
            *window_length_vowel]
    frame_amdf = compute_amdf(current_frame)
    pl.figure()
    pl.plot(frame_amdf)
    pl.title('AMDF function for frame %d' %(k+1))
```



Primul lucru ce poate fi observat este că în loc de maxime, funcția AMDF va prezenta **minime** distanțate cu T_0. Putem, din nou, să estimăm

valoarea F_0 în mod similar cu cea estimată din funcția de autocorelație.

Exercițiu T5.2.11 Care este valoarea F_0 calculată din funcția AMDF? ■

Exercițiu T5.2.12 Afișați funcția AMDF pentru primele cadre din consoană. Se pot determina minime periodice? ■

T5.3. Detecția automată a F_0

Folosind funcțiile de autocorelație și AMDF am putut să detectăm valorile F_0 mult mai ușor decât din forma de undă a semnalului. Dar detecția a fost realizată manual. Această metodă nu este eficientă atunci când dorim să calculăm frecvența fundamentală a mai multor semnale vocale. Astfel că, avem nevoie de o metodă automată de calcul a F_0 pornind de la maximele sau minimele funcțiilor prezentate anterior.

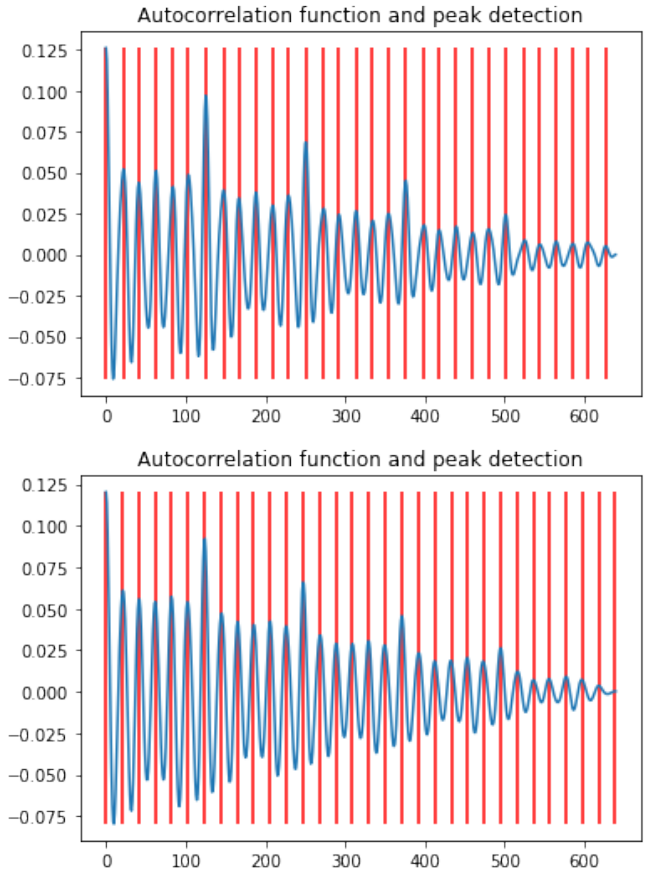
O valoare maximă locală a unei funcții poate fi detectată automat prin compararea valorii ei cu cele din imediata vecinătate. Definim o funcție ce determină indicii maximelor locale dintr-un vector de valori:

[12]:
```
def peak_detection_1(sample):
    # Există întotdeauna un maxim în origine
    # pentru autocorelație
    indices = [0]
    for i in range(1,len(sample)-1):
        # Comparăm valoarea curentă cu vecinii
        if (sample[i] > sample[i-1])  \
                and (sample[i] > sample[i+1]):
            indices.append(i)
    return indices
```

Aplicăm algoritmul asupra câtorva cadre de semnal:

[13]:
```
for k in range(2):
    current_frame = wav_data_vowel[int(k *(1-p))  \
            *window_length_vowel: int((k*(1-p)+1)) \
            *window_length_vowel]
    frame_autocor = compute_autocorrelation(current_frame)
    indices = peak_detection_1(frame_autocor)
    # afișăm rezultatele
    pl.figure()
    pl.plot(frame_autocor)
    pl.title('Autocorrelation function and peak detection')
```

```
    pl.vlines(indices, min(frame_autocor), \
            max(frame_autocor), 'r');
```

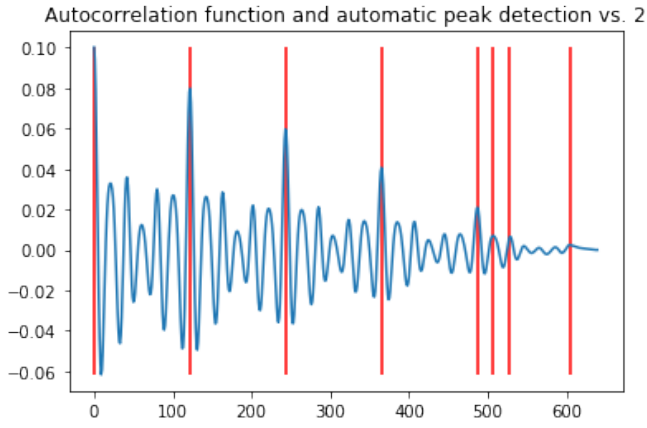


Se pare că am detectat prea multe maxime. Ceea ce nu specificat în funcția de detecție a maximelor este că dorim doar cele mai ”mari” maxime. Rescriem funcția:

[14]:
```
def peak_detection_2(sample):
    local_max = []
    local_max_ind = []
    indices = [0]
    for i in range(1,len(sample)-1):
        if (sample[i] > sample[i-1]) and \
                (sample[i] > sample[i+1]):
            # Stocăm valorile și indecșii maximelor locale
            local_max_ind.append(i)
            local_max.append(sample[i])
```

```
    # Parcurgem restul maximelor și îl determinăm
    # pe următorul cu valoarea cea mai mare după
    # cel din origine
    i = 1
    while i < len(local_max):
        next_max = np.argmax(local_max[i:])
        indices.append(local_max_ind[next_max+i])
        i = i+next_max+1
    return indices
```

[15]:
```
# Vizualizăm un singur cadru de semnal
k = 4
current_frame = wav_data_vowel[int(k *(1-p))  \
            *window_length_vowel: int((k*(1-p)+1)) \
            *window_length_vowel]
    frame_amdf = compute_amdf(current_frame)
indices = peak_detection_2(frame_autocor)
# afișăm rezultatele
pl.figure()
pl.plot(frame_autocor)
pl.title('Autocorrelation function and automatic peak \
        detection vs. 2')
pl.vlines(indices, min(frame_autocor), \
        max(frame_autocor), 'r');
```



Arată mai bine! Dar putem rafina funcție de mai sus ținând cont de valorile minime și maxime pe care F_0 le poate avea. Am menționat anterior că acestea pot fi în intervalul $[55,450]Hz$. Să vedem ce înseamnă aceasta

în eșantioane ale funcției de autocorelație. Ne reamintim faptul că fiecare eșantion de semnal este distanțat cu $T_s = \frac{1}{F_s}$ secunde.

Un sinus de 50Hz va avea nevoie de N eșantioane pentru a realiza un ciclu complet. Valoarea N depinde de frecvența de eșantionare. Să calculăm mai întâi perioada unui sinus de 50Hz:

$$F_0 = 50 \text{ Hz}$$

$$T_0 = \frac{1}{50} = 20 \text{ msec}$$

Știind frecvența de eșantionare >$F_s = 16kHz$,
perioada de eșantionare este:

$$T_s = \frac{1}{Fs} = 62.5 \text{ microseconds.}$$

Astfel că pentru $T_0 = 20$ msec (perioada sinusului de 50Hz) avem nevoie de

$$N = \frac{20*10^{-3}}{62.5*10^{-6}} = 320 \text{ samples.}$$

Să simplificăm ecuația de mai sus pentru a o putea utiliza pe viitor:

$$N = \frac{\frac{1}{F_0}}{\frac{1}{F_s}} = \frac{F_s}{F_0}$$

Dar pentru 450Hz? Avem nevoie de:

$$N = \frac{16000}{450} \approx 35 \text{ samples}$$

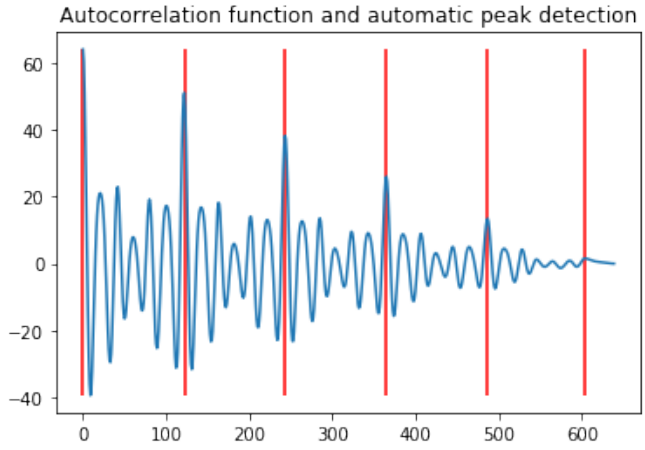
Aceasta înseamnă că dacă dorim să calculăm un F_0 maxim de 450Hz, maximele funcției de autocorelație vor fi la cel puțin 35 de eșantioane distanță, pentru un semnal eșantioant la 16kHz. Iar pentru un semnal cu F_0 de 50Hz, maximele vor fi distanțate cu 320 de eșantioane. Ca urmare, fereastra de analiză aleasă pentru calculul funcției de autocorelație sau AMDF trebuie să țină cont de aceste extreme.

Introducem această condiție în funcția de calcul a maximelor:

[16]:
```
def peak_detection_3(sample, sampling_frequency, max_F0):
    local_max = [sample[0]]
    local_max_ind = [0]
    indices = [0]
    min_peak_distance = sampling_frequency//max_F0
    for i in range(1,len(sample)-1):
        if sample[i] > sample[i-1] and  \
```

```
                sample[i] > sample[i+1]:
            local_max_ind.append(i)
            local_max.append(sample[i])
    i = 1
    while i < len(local_max):
        next_max = np.argmax(local_max[i:])
        if local_max_ind[i+next_max] - local_max_ind[i-1] \
                > min_peak_distance:
            indices.append(local_max_ind[next_max+i])
        i = i+next_max+1
    return indices
```

[17]:
```
# Luăm doar un cadru de analiză
k = 4
current_frame = wav_data_vowel[int(k *(1-p))  \
            *window_length_vowel: int((k*(1-p)+1)) \
            *window_length_vowel]
frame_autocor = compute_autocorrelation_2(current_frame)
indices = peak_detection_3(frame_autocor,  \
        sampling_frequency_vowel, 450)
pl.figure()
pl.plot(frame_autocor)
pl.title('Autocorrelation function and automatic \
         peak detection vs. 3')
pl.vlines(indices, min(frame_autocor), max(frame_autocor), \
         'r');
```



Acum că știm poziția maximelor în funcția de autocorelație, putem să

calculăm automat valoarea lui F_0:

[18]:
```
indices = peak_detection_3(frame_autocor, \
        sampling_frequency_vowel, 450)
# Calculăm distanța dintre indecșii returnați de funcție:
difs = [x-indices[i-1] for i,x in enumerate(indices)][1:]
# Determinăm media diferențelor
average_dist = np.mean(difs[:3])
# Și o convertim în Hz
F0 = sampling_frequency_vowel/average_dist
print ("\nThe F0 value for frame no. %d is: %.2f Hz\n" \
        %(k,F0))
```

[18]:
```
The F0 value for frame no. 4 is: 131.51 Hz
```

Exercițiu T5.3.1 Calculați valoarea F_0 pentru același cadru de vorbire folosind un software extern (de ex. Praat)? Sunt similare valorile? ■

Exercițiu T5.3.2 Implementați funcția ce determină minimele funcției AMDF și recalculați F_0. ■

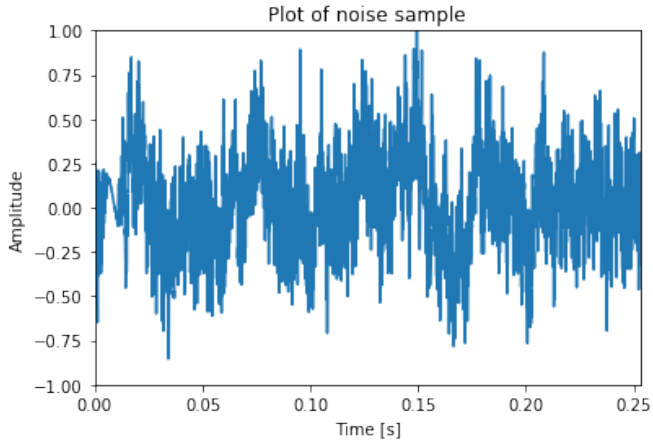
Exercițiu T5.3.3 Înregistrați-vă pronunțând vocalele din limba română și calculați valoarea F0 medie pentru fiecare dintre ele. ■

T5.4. Îmbunătățirea algoritmilor de detecție ai F_0

Ce se întâmplă cu algoritmii anteriori dacă adăugăm zgomot? Codul de mai jos adaugă zgomot alb gaussian peste semnalul de intrare și recalculează funcția de autocorelație:

[19]:
```
# Citim un fișier cu zgomot
noise_file =  'speech_files/noise.wav'
noise_struct = wave.open(noise_file, 'r')
noise_sampling_frequency = noise_struct.getframerate()
noise_bytes = noise_struct.readframes(len(wav_data_vowel))
noise_data = np.frombuffer(noise_bytes, dtype='int16')
noise_struct.close()
```

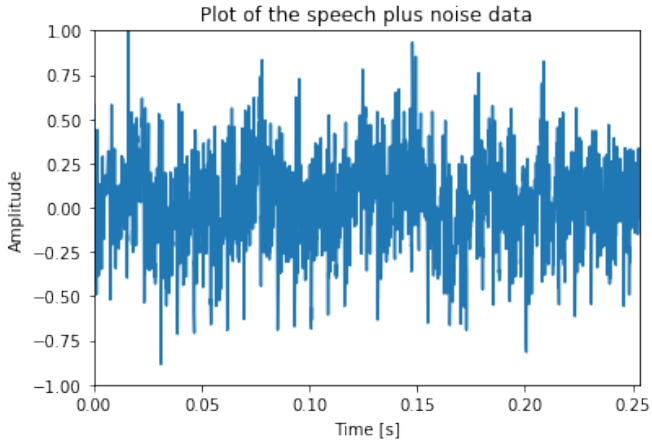
[20]:
```
# Normalizăm noise_data la [-1,1] and și îl ponderăm
# pentru a nu acoperi semnalul de intrare în întregime
noise_data = noise_data/float(max(abs(noise_data)))
# Afișăm semnalul de zgomot
time_axis = np.arange(0, len(noise_data)) \
        *1.00/noise_sampling_frequency
pl.plot(time_axis, noise_data)
pl.xlim([0, time_axis[-1]])
pl.ylim([-1, 1])
pl.xlabel('Time [s]')
pl.ylabel('Amplitude')
pl.title('Plot of noise sample');
```



[21]:
```
import IPython
IPython.display.Audio(noise_data, \
        rate=noise_sampling_frequency)
```

[21]:

[22]:
```
# Adăugăm zgomotul peste semnalul vocal
noisy_speech = wav_data_vowel[:len(wav_data_vowel)] \
        + noise_data
noisy_speech = noisy_speech/float(max(abs(noisy_speech)))
# Axa timpului
time_axis = np.arange(0, len(noisy_speech)) \
        *1.00/noise_sampling_frequency

# Afișăm semnalul rezultat
pl.plot(time_axis, noisy_speech)
pl.xlim([0, time_axis[-1]])
pl.ylim([-1, 1])
pl.xlabel('Time [s]')
pl.ylabel('Amplitude')
pl.title("Plot of the speech plus noise data");
```



Din forma de undă a semnalului cu zgomot nu mai putem acum să determinăm periodicitatea și probabil am fi înclinați să spunem că acest cadru de analiză aparține unei consoane. Să-l ascultăm, însă și să vedem dacă funcția de autocorelație funcționează și în acest caz:

[23]:
```
IPython.display.Audio(noisy_speech, \
        rate=noise_sampling_frequency)
```

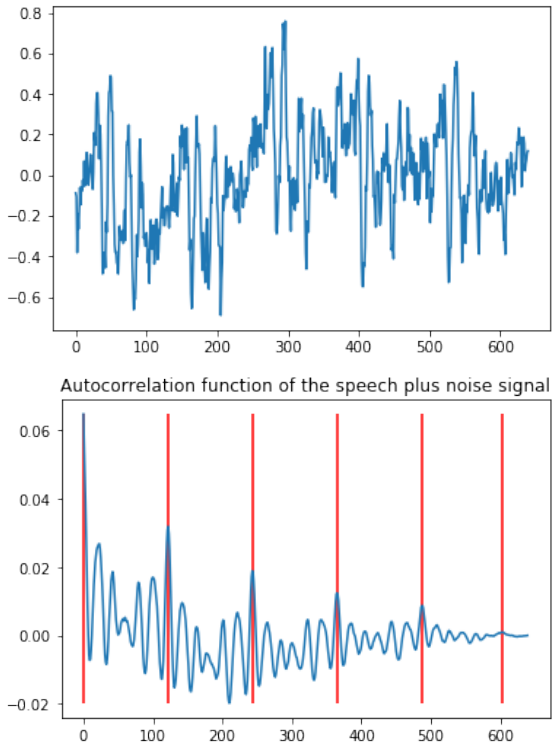
[23]: 0:00 / 0:00

Recalculăm funcția de autocorelație și valoarea lui F_0:

[24]:
```
k = 4
current_frame = \
        noisy_speech[int(k*(1-p)*window_length_vowel): \
        int((k*(1-p)+1)*window_length_vowel)]
frame_autocor = []
frame_autocor = compute_autocorrelation(current_frame)
indices = peak_detection_3(frame_autocor, \
        noise_sampling_frequency, 450)
difs = [x-indices[i-1] for i,x in enumerate(indices)][1:]
average_dist = np.mean(difs[:3])
# Calcul F0 în Hz
F0 = sampling_frequency_vowel/average_dist
print ("\nThe F0 value for the frame no. %d is: %.2f Hz\n" \
        %(k,F0))
# Plot
pl.figure()
pl.plot(current_frame)
```

```
pl.figure()
pl.plot(frame_autocor)
pl.title('Autocorrelation of the speech plus noise signal')
pl.vlines(indices, min(frame_autocor), max(frame_autocor),  \
        'r');
```

[24]: `The F0 value for the frame no. 4 is: 131.51 Hz`



Se poate observa că, deși forma de undă nu mai este periodică în mod evident, funcția de autocorelație reușește să extragă periodicitatea semnalului. Pentru a îmbunăți rezultatele funcției de autocorelație, se pot aplica o serie de pre-procesări ale semnalului, astfel încât maximele parazite din autocorelație să fie reduse, iar valoarea F_0 calculată să fie cât mai apropiată de valoarea reală.

T5.4.1 Filtrarea trece jos

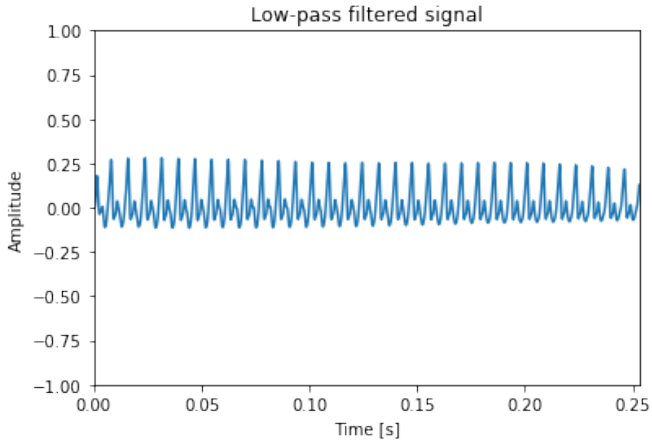
Dacă încercăm să determinăm o valoare a lui F_0 ce se află sub 450Hz, aceasta înseamnă că restul informației din banda de frecvențe superioară poate fi ignorată. Deci putem filtra semnalul trece-jos și să îmbunătățim astfel

performanțele algoritmului de detecție al frecvenței fundamentale. Vom implementa un astfel de filtru și vom recalcula valoarea lui F_0:

[25]:
```
# Proiectăm un filtru trece jos cu frecvența de tăiere
# la 400Hz
from scipy.signal import lfilter, butter
# Frecvența de tăiere
cutoff = 400
# Ordinul filtrului
order = 5
# Frecvența Nyquist
nyq = 0.5 * sampling_frequency_vowel
# Normalizăm frecvența de tăiere
normal_cutoff = cutoff / nyq
# Calculăm coeficienții unui filtru Butterworth de
# tip trece-jos
b, a = butter(order, normal_cutoff, btype='low', \
        analog=False)
# Filtrăm datele
filtered_data = lfilter(b, a, wav_data_vowel)
```

Și afișăm rezultatele:

[26]:
```
time_axis_filtered = np.arange(0, len(filtered_data)) \
        *1.00/sampling_frequency_vowel
pl.plot(time_axis_filtered, filtered_data)
pl.title('Low-pass filtered signal')
pl.xlim([0, time_axis_filtered[-1]])
pl.ylim([-1, 1])
pl.xlabel('Time [s]')
pl.ylabel('Amplitude');
```



Putem să și ascultăm semnalul filtrat:

[27]:
```
IPython.display.Audio(filtered_data, \
        rate=sampling_frequency_vowel)
```

[27]: 0:00 / 0:00

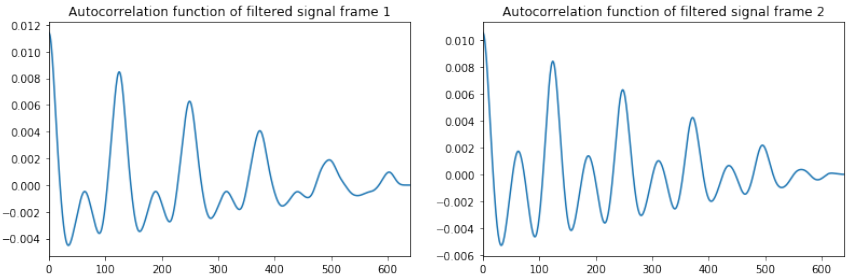
Și să-l comparăm cu cel original:

[28]:
```
IPython.display.Audio(wav_data_vowel, \
        rate=sampling_frequency_vowel)
```

[28]: 0:00 / 0:00

Iar acum putem să calculăm funcția de autocorelație pentru semnalul filtrat

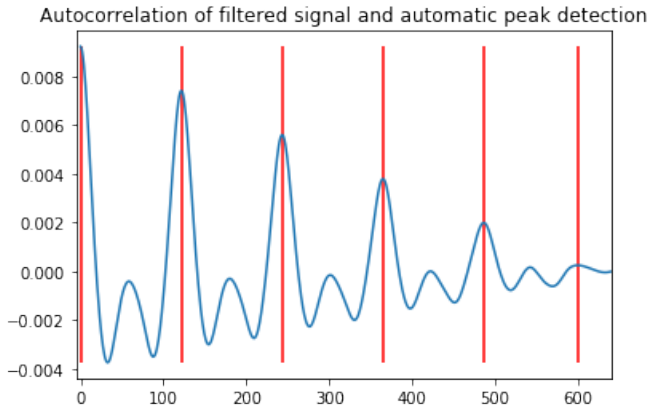
[29]:
```
for k in range(2):
    current_frame = filtered_data[k*window_length_vowel: \
            (k+1)*window_length_vowel]
    frame_autocor = compute_autocorrelation(current_frame)
    pl.figure()
    pl.plot(frame_autocor)
    pl.xlim([0, window_length_vowel])
    pl.title('Autocorrelation of filtered signal frame %d' \
            %(k+1))
```



Maximele parazite din funcția de autocorelație au dispărut parțial și putem să determinăm iar F_0:

[30]:
```
indices = peak_detection_3(frame_autocor, \
        sampling_frequency_vowel, 450)
difs = [x-indices[i-1] for i,x in enumerate(indices)][1:]
average_dist = np.mean(difs[:3])
# Convertim în Hz
F0 = sampling_frequency_vowel/average_dist
pl.figure()
pl.plot(frame_autocor)
pl.xlim([-5, window_length_vowel])
pl.title('Autocorrelation of filtered signal and  \
        automatic peak detection')
pl.vlines(indices, min(frame_autocor), max(frame_autocor), \
         'r')
print ("\nThe F0 value for frame no. %d is: %.2f Hz\n" \
         %(k,F0))
```

[30]:
```
The F0 value for frame no. 4 is: 131.51 Hz
```



T5.4.2 Limitarea semnalului (Metoda Center Clipping)

O altă metodă de reducere a maximelor parazite din funcția de autocorelație se bazează pe **funcția de center clipping**. Aceasta este este definită ca:

$$y(n) = clc[x(n)] = \begin{cases} x(n) - C_L, & x(n) \geq C_L \\ 0, & |x(n))| < C_L \\ x(n) + C_L, & x(n) \leq -C_L \end{cases} \quad \text{(T5.4.1)}$$

Unde C_L este pragul de limitare ce este de obicei setat la 30% din amplitudinea maximă a semnalului.

http://notedetection.weebly.com/center-clipping.html

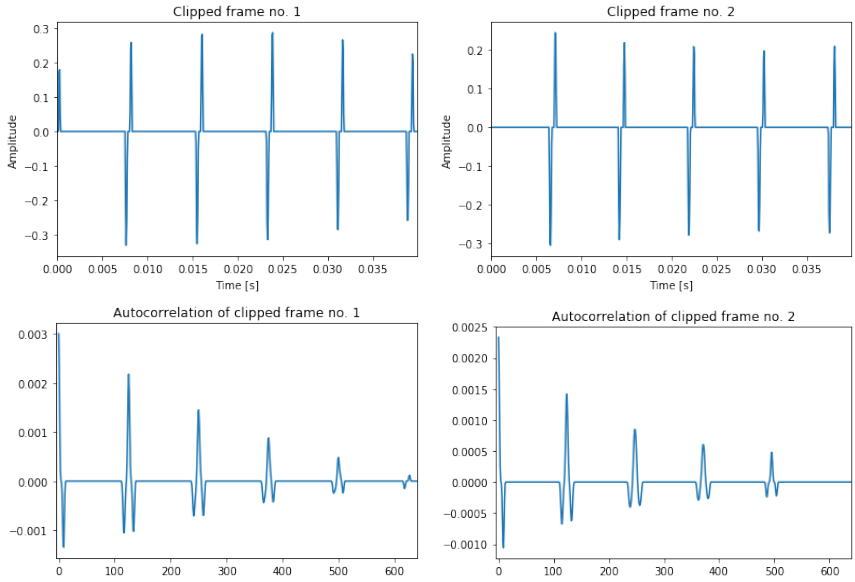
Interpretarea funcției spune că toate eșantioanele ce au valori sub pragul C_L vor fi setate la 0, iar restul vor fi calculate conform definiției de mai sus. Implementăm și noi această funcție:

[31]:
```
def center_clipping(input_frame, cl_percentage):
    cl = cl_percentage * max(abs(input_frame))
    clipped_signal = np.zeros(len(input_frame))
    for i in range(len(input_frame)):
        if input_frame[i] >= cl:
            clipped_signal[i] = input_frame[i] - cl
        elif input_frame[i] <= -cl:
            clipped_signal[i] = input_frame[i] + cl
        elif abs(input_frame[i]) < cl:
            clipped_signal[i] = 0
    return clipped_signal
```

Și o aplicăm asupra semnalului înainte de a calcula autocorelația:

[32]:
```
limiting_threshold = 0.7*max(wav_data_vowel)
time_axis_clipped = np.arange(0, window_length_vowel) \
        *1.00/sampling_frequency_vowel
for k in range(2):
    current_frame = wav_data_vowel[k*window_length_vowel: \
            (k+1)*window_length_vowel]
    clipped_frame = center_clipping(current_frame, \
            limiting_threshold)
    clipped_frame_autocor = \
            compute_autocorrelation(clipped_frame)
    # Afișăm cadrul după center clipping
    pl.figure()
    pl.plot(time_axis_clipped, clipped_frame)
    pl.xlim([0, time_axis_clipped[-1]])
```

```
    pl.xlabel('Time [s]')
    pl.ylabel('Amplitude')
    pl.title('Clipped frame no. %d' %(k+1))

    # Afișăm autocorelația cadrului
    pl.figure()
    pl.plot(clipped_frame_autocor)
    pl.xlim([-5, window_length_vowel])
    pl.title('Autocorrelation of clipped frame no. %d' \
            %(k+1))
```

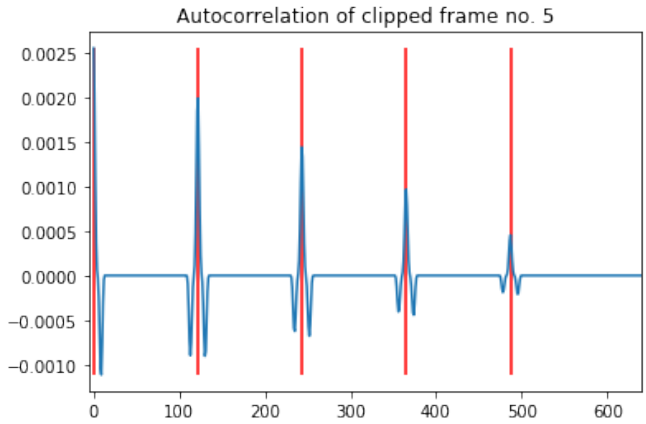


Calculăm și valorile efective ale F_0:

[33]:
```
# calculăm F0 din semnalul limitat
indices = peak_detection_3(clipped_frame_autocor, \
        sampling_frequency_vowel, 450)
difs = [x-indices[i-1] for i,x in enumerate(indices)][1:]
average_dist = np.mean(difs[:3])
F0 = sampling_frequency_vowel/average_dist
pl.figure()
pl.plot(clipped_frame_autocor)
pl.xlim([-5, window_length_vowel])
pl.title('Autocorrelation of clipped frame no. %d' %(k+1))
pl.vlines(indices, min(clipped_frame_autocor), \
        max(clipped_frame_autocor), 'r')
```

```
print ("\nThe F0 value for frame no. %d is: %.2f Hz\n" \
       %(k,F0))
```

[33]:
```
The F0 value for frame no. 4 is: 131.51 Hz
```



Se poate observa faptul că maximele din funcția de autocorelație sunt acum mult mai bine evidențiate.

Exercițiu T5.4.1 Variați pragul de limitare de la 0.1 la 0.9 din valoarea maximă și vizualizați rezultatul. ■

T5.5. Concluzii

În acest tutorial am încercat să determinăm periodicitatea semnalului vocal folosind forma de undă a acestuia și două funcții simple. Pe baza acestora am putut calcula automat valorile frecvenței fundamentale ale fiecărui segment vocal.

Într-un tutorial viitor vom încerca să determină frecvența fundamentală folosind reprezentarea în frecvență a semnalului vocal. Această reprezentare este mult mai robustă la zgomot și ne oferă o serie de alte informații privind periodicitatea semnalului în ansamblul ei.

BIBLIOGRAFIE SUPLIMENTARĂ

- Benesty et al, "Springer Handbook of Speech Processing", Springer, 2008
- Paul Taylor, "Text to speech synthesis", Cambridge University Press, 2009

RESURSE MEDIA

- Hartmut Traunmüller, Anders Eriksson, "The frequency range of the voice fundamental in the speech of male and female adults ", online: `http://www2.ling.su.se/staff/hartmut/f0_m&f.pdf`
- David Talkin, "A Robust Algorithm for Pitch Tracking", online: `https://www.ee.columbia.edu/~dpwe/papers/Talkin95-rapt.pdf`
- Coursera, "Audio Signal Processing for Music Applications", `https://www.coursera.org/lecture/audio-signal-processing/pitch-detection-Vr9du`
- Geoff Boynton, Course on Sensation and Perception - Chapter 11: Sound, The Auditory System, and Pitch Perception, online: `http://courses.washington.edu/psy333/lecture_pdfs/chapter11_SoundPitch.pdf`

T6 Analiza prin transformata Fourier

T6.1. Analiza în domeniul frecvenței

Până în prezent, toate procesările făcute asupra semnalului vocal au fost realizate în domeniul timp. Și, deși în anumite cazuri este suficient, în aplicații de prelucrare și analiză a semnalului vocal, este necesar să ne uităm și la reprezentarea din domeniul frecvență.

În 1822, Jean Baptiste Joseph Fourier (1768-1839) a introdus o metodă de descriere a oricărei funcții matematice folosind funcții trigonometrice simple. Această metodă este denumită **Transformata Fourier** și repezintă baza tuturor analizelor în frecvență.

Cu toate acestea, reprezentarea în frecvență nu este un concept pur matematic, deoarece urechea umană realizează o descompunere similară a sunetului. Cilii aparatului Corti reacționează în mod diferit la frecvențele sunetelor. Frecvențele înalte sunt prelucrate de cilii apropiați de membrana bazilară, iar frecvențele joase de cilii apropiați de apexul cohlear.

În următoarele secțiuni vom încerca să calculăm și să vizualizăm transformata Fourier a semnalelor vocale și vom introduce și una dintre cele mai importante aplicații ale acesteia, **spectrograma**.

Haideți să începem! Dacă dorim să reprezentăm un semnal în frecvență, mai întâi trebuie să-l citim.

În acest tutorial vom folosi 2 vocale rostite de aceeași persoană:

[1]:
```
import wave
import numpy as np

########################
# Citim prima vocală
input_wav_file_1 = 'speech_files/adr_e1.wav'
wav_struct_1 = wave.open(input_wav_file_1, 'r')
wav_bytes_1 = wav_struct_1.readframes(-1)
wav_data_1 = np.frombuffer(wav_bytes_1, dtype='int16')
wav_data_1 = wav_data_1/float(max(abs(wav_data_1)))
```

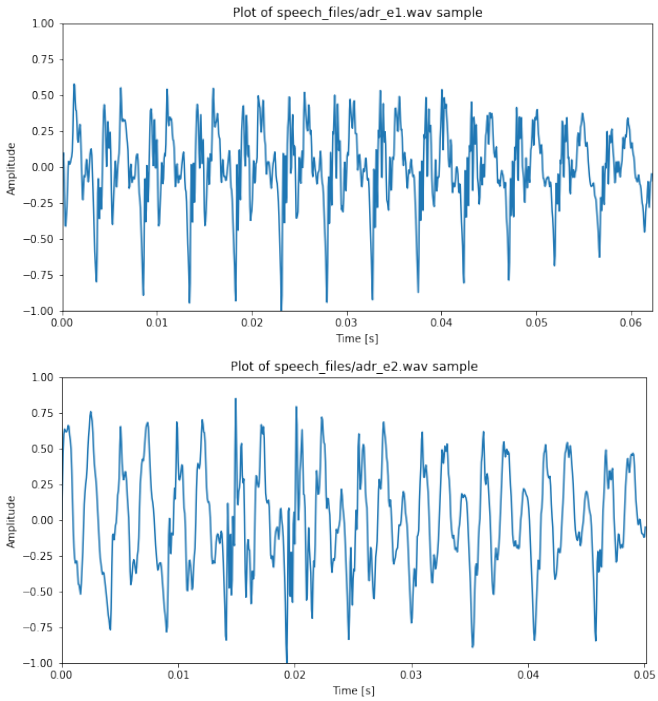
```
########################
# Citim a doua vocală
input_wav_file_2 = 'speech_files/adr_e2.wav'
wav_struct_2 = wave.open(input_wav_file_2, 'r')
wav_bytes_2 = wav_struct_2.readframes(-1)
wav_data_2 = np.frombuffer(wav_bytes_2, dtype='int16')
wav_data_2 = wav_data_2/float(max(abs(wav_data_2)))

# Ambele fișiere au aceeași parametri, astfel că
# îi extragem o singură dată
sampling_frequency = wav_struct_1.getframerate()
bit_depth = wav_struct_1.getsampwidth()
no_channels = wav_struct_1.getnchannels()
# Închidem fluxurile de citire
wav_struct_1.close()
wav_struct_2.close()
```

... afișăm și ascultăm semnalele:

[2]:
```
import matplotlib.pylab as pl
%matplotlib inline

########################
# Afișăm primul semnal
pl.figure(figsize=[10,5])
time_axis = np.arange(0, len(wav_data_1))\
    *1.00/sampling_frequency
duration_1 = len(wav_data_1)*1.00/sampling_frequency
print ("Duration of the first file: %f second(s)" \
        %duration_1)
pl.plot(time_axis, wav_data_1)
pl.xlim([0, duration_1])
pl.ylim([-1, 1])
pl.xlabel('Time [s]')
pl.ylabel('Amplitude')
pl.title("Plot of %s sample" %input_wav_file_1)

########################
# Afișăm al doilea semnal
pl.figure(figsize=[10,5])
time_axis = np.arange(0, len(wav_data_2))\
    *1.00/sampling_frequency
```

```
duration_2 = len(wav_data_2)*1.00/sampling_frequency
print ("Duration of the first file: %f second(s)"\
       %duration_2)
pl.plot(time_axis, wav_data_2)
pl.xlim([0, duration_2])
pl.ylim([-1, 1])
pl.xlabel('Time [s]')
pl.ylabel('Amplitude')
A = pl.title("Plot of %s sample" %input_wav_file_2)
```

[2]:
```
Duration of the first file: 0.062250 second(s)
Duration of the first file: 0.050125 second(s)
```



Exercițiu T6.1.1 Cele două fișiere conțin aceeași vocală? ■

Să le ascultăm...

[3]:
```
import IPython
IPython.display.Audio(wav_data_1, rate=sampling_frequency)
```

[3]:

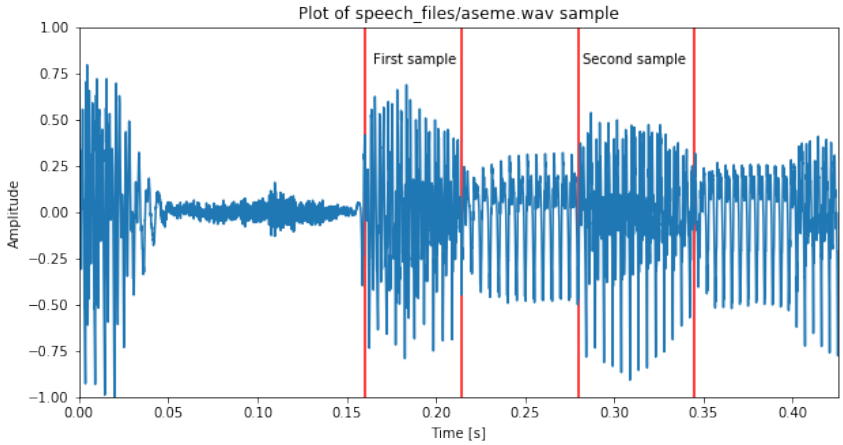
[4]:
```
IPython.display.Audio(wav_data_2, rate=sampling_frequency)
```

[4]:

S-ar putea ca un context al celor două vocale să fie mai de ajutor. Codul de mai jos afișează și redă segmentul de voce din care au fost extrase vocalele:

[5]:
```
# Citim segmentul vocal
input_wav_file_context = 'speech_files/aseme.wav'
wav_struct_context = wave.open(input_wav_file_context, 'r')
wav_bytes_context = wav_struct_context.readframes(-1)
wav_data_context = np.frombuffer(wav_bytes_context,\
                                 dtype='int16')
wav_data_context = wav_data_context \
            /float(max(abs(wav_data_context)))
sampling_frequency_context = \
    wav_struct_context.getframerate()

pl.figure(figsize=[10,5])
time_axis = np.arange(0, len(wav_data_context))\
    *1.00/sampling_frequency
duration_context = len(wav_data_context)\
    *1.00/sampling_frequency
print ("Duration of the context file: %f second(s)" \
       %duration_context)

pl.plot(time_axis, wav_data_context)
pl.xlim([0, duration_context])
pl.ylim([-1, 1])
pl.xlabel('Time [s]')
pl.ylabel('Amplitude')
pl.title("Plot of %s sample" %input_wav_file_context)
pl.vlines([0.16,0.214, 0.28, 0.345], -1, 1, 'r')
pl.text(0.165,0.80,'First sample')
pl.text(0.282,0.80,'Second sample')
```

[5]:
```
Duration of the context file: 0.425438 second(s)
```



Și îl redăm:

[6]:
```
IPython.display.Audio(wav_data_context, \
        rate=sampling_frequency_context)
```

[6]: 0:00 / 0:00

Se pare că, deși cele două fișiere conțin același semnal, formele lor de undă nu sunt similare. Atunci cum am putea să facem de exemplu recunoașterea vorbirii dacă există atât de multă variabilitate chiar și la nivel de fonem pentru același vorbitor în cadrul aceluiași cuvânt?

Răspunsul este să utilizăm o transformată de spațiu ce uniformizează această variabilitate. Această transformată este transformata în domeniul frecvenței, și anume **transformata Fourier (TF)**. TF este o transformată ce permite reprezentarea oricărui semnal periodic ca o sumă de funcții trigonometrice: sinus și cosinus. Aceasta înseamnă că orice semnal periodic poate fi descompus într-un set de componente sinusoidale cu amplitudini, faze și frecvențe fundamentale diferite.

Transformata Fourier de bază operează asupra semnalelor sau funcțiilor continue. În cazul nostru, semnalele fiind digitizate, vom folosi transformata Fourier discretă și algoritmi de calcul rapid ai coeficienților denumiți colectiv **transformata Fourier rapidă**.

T6.1.1 Transformata Fourier Discretă

Transformata Fourier discretă este definită astfel:

$$X_k = \sum_{n=0}^{N-1} x_n \cdot e^{-\frac{2\pi i}{N} kn} = \sum_{n=0}^{N-1} x_n \left[\cos(\frac{2\pi kn}{N}) - i \cdot \sin(\frac{2\pi kn}{N}) \right] \quad \text{(T6.1.1)}$$

Fiecare coeficient X_k reprezintă un număr complex ce înglobează amplitudinea și faza unei componente sinusoidale complexe a semnalului x_n. Frecvența sinusoidei este de k ciclii per N eșantioane. Iar amplitudinea și faza ei sunt:

$$|X_k| / N = \sqrt{R(X_k)^2 + Im(X_k)^2} / N \tag{T6.1.2}$$

$$arg(X_k) = atan2(Im(X_k), Re(X_k)) = -i * ln(\frac{X_k}{|X_k|}) \tag{T6.1.3}$$

unde $atan2()$ este varianta cu două argumente a funcției arctangentă. Dacă secvența de intrare este reală, atunci coeficienții sunt complex conjugați.

T6.1.2 Transformata Fourier Rapidă

Calculul coeficienților Fourier ridică probleme atunci când resursele de calcul sunt limitate. Acest fapt restrânge aplicabilitatea algoritmului pentru procesări în timp real. Fiind esențial atunci să se dezvolte algoritmi eficienți, rapizi și exacți pentru calculul acestor coeficienți. Totalitatea acestor algoritmi este denumită **Transformata Fourier Rapidă - en. Fast Fourier Transforms (FFT)** și include un număr mare de metode. Cele mai cunoscute și utilizate sunt cele cu **decimare în timp** și **decimare în frecvență**. Ambele metode implică descompunerea calculului coeficienților până la un nivel de 2 eșantioane și recombinarea rezultatelor parțiale. Astfel că, în aplicații practice, lungimea semnalelor discrete asupra cărora se aplică FFT trebuie să fie o putere a lui 2. Dacă această condiție nu este îndeplinită, datele de intrare sunt completate cu zero (en. *zero-padded*)

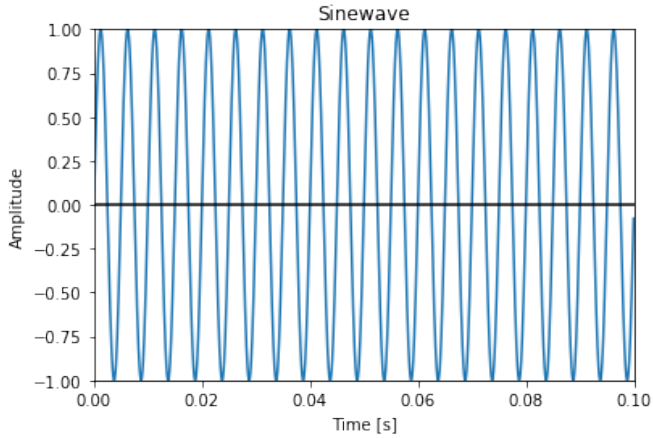
În cele ce urmează vom aplica FFT asupra a două semnale: un sinus și un semnal vocal pentru a înțelege mai bine ce reprezintă coeficienții Fourier și analiza spectrală:

T6.1.2.1 FFT sinus

Să analizăm mai întâi un semnal mai simplu - un sinus, pentru a înțelege FFT mai ușor. Vom utiliza pentru calculul FFT o funcție disponibilă în modulul `scipy`.

[7]:
```
# Generăm un sinus
volume = 0.5      # domeniu [0.0, 1.0]
fs = 16000        # frecvența de eșantionare în Hz
```

```
duration = 0.1    # durata în secunde
f = 200.0         # frecvența sinusului
# Generăm eșantioanele
sinewave = (np.sin(2*np.pi*np.arange(fs*duration)\
                   *f/fs)).astype(np.float32)
# Afișăm sinusul
time_axis = np.arange(0, len(sinewave))*1.00/fs
pl.plot(time_axis, sinewave)
pl.xlim([0, duration])
pl.ylim([-1, 1])
pl.title('Sinewave')
pl.xlabel('Time [s]')
pl.ylabel('Amplitude')
pl.axhline(y=0, color='k');
```



[8]:
```
# Calculăm transformata Fourier rapidă a sinusului
from scipy.fftpack import fft, fftshift
sine_fft = fft(sinewave, len(sinewave)) \
           / (len(sinewave)/2.0)
# Afișăm primii 10 coeficienți FFT
print ('\n'.join(["Coef #"+str(i+1).zfill(2)+":"+str(x) \
                  for i,x in enumerate(sine_fft[:10])]))
```

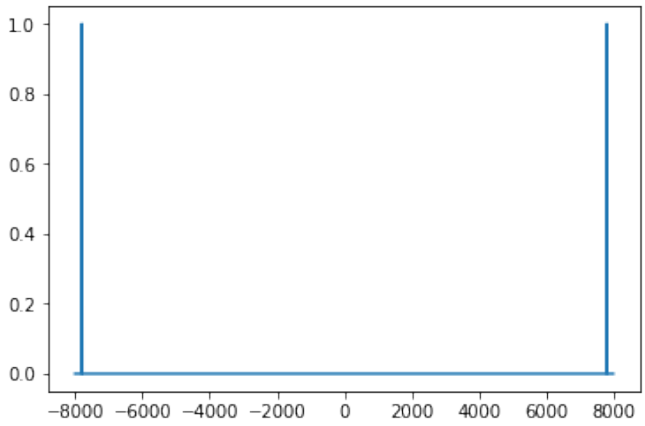
[8]:
```
Coef #01:(1.490116e-09+0j)
Coef #02:(5.1729072e-11+7.11989e-11j)
Coef #03:(2.7359465e-10+8.88962e-11j)
Coef #04:(2.78754e-10-9.057253e-11j)
Coef #05:(5.5597055e-11-7.65227e-11j)
```

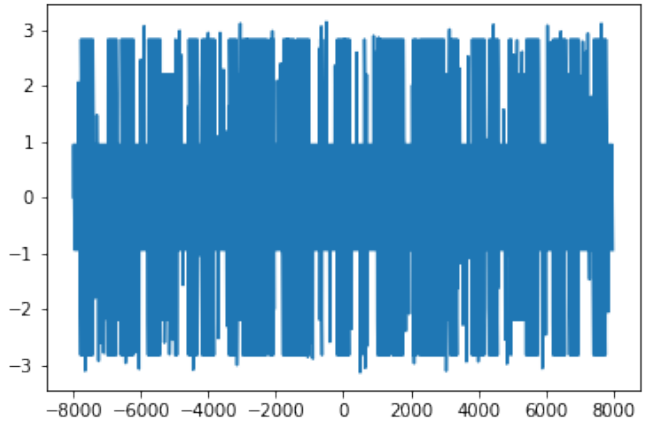
```
Coef #06:(1.1744019e-08+1.03235545e-08j)
Coef #07:(6.128793e-11+8.435567e-11j)
Coef #08:(3.2559241e-10+1.0579141e-10j)
Coef #09:(3.4616426e-10-1.1247557e-10j)
Coef #10:(7.68852e-11-1.0582356e-10j)
```

Cunoscând faptul că rezultatul transformatei Fourier este un set de numere complexe, reprezentarea grafică a acestora poate fi făcută în coordonate carteziene sau polare. În prelucrarea semnalelor și în special în analiza spectrală se utilizează reprezentarea polară și mai mult decât atât, se efectuează și o o separare a amplitudinii valorilor complexe de faza acestora. Rezultă astfel două reprezentări grafice denumite: **spectrul de amplitudini** și **spectrul de faze**.

În ambele reprezentări axa oX este cea a frecvenței, iar informația afișată poate fi astfel ușor interpretată. În exemplul de mai sus, se poate observa în spectrul de amplitudini o valoare unitară în dreptul frecvenței de 300Hz ce corespunde cu ceea ce am generat inițial.

[9]:
```
# Afișăm spectrul de amplitudini
magnitude_spectrum = np.abs(sine_fft)
# Axa de frecvențe oX
frequency_axis = np.arange(-len(sinewave)//2,\
            len(sinewave)//2)*fs/len(sinewave)
pl.plot(frequency_axis, magnitude_spectrum);
# Afișăm spectrul de faze
pl.figure()
phase_spectrum = np.angle(sine_fft)
pl.plot(frequency_axis, phase_spectrum);
```





Se pot observa următoarele:

- spectrul de amplitudini este simetric față de origine. Acest lucru se datorează faptului că secvența de intrare a FFT, sinusul, este o
- secvență reală. În practică se va afișa doar jumătatea superioară (frecvențele pozitive) ale spectrului de amplitudini.
- spectrul de faze nu oferă prea multe informații la momentul actual și mai mult decât atât pare să nu respecte faza sinusul generat. Problema fazei componentelor spectrale se datorează periodicității unghiului în coordonate polare. Și anume -2π este egal cu 2π astfel că se aplică de obiecei așa numitul proces de *phase unwrapping* în care se ia în considerare faza minimă ce poate genera acea valoare.

Exercițiu T6.1.2 Generați două sinusuri cu frecvențe diferite și însumați-le. Afișați FFT pentru acest semnal și verificați frecvențele observate în spectru. ■

T6.1.2.2 FFT semnalului vocal

Pentru a putea calcula FFT pentru semnalele noastre de intrare, trebuie mai întâi să ajustăm lungimea ferestrei de analiză de 20-40 ms la un număr de eșantioane egal cu o putere a lui 2:

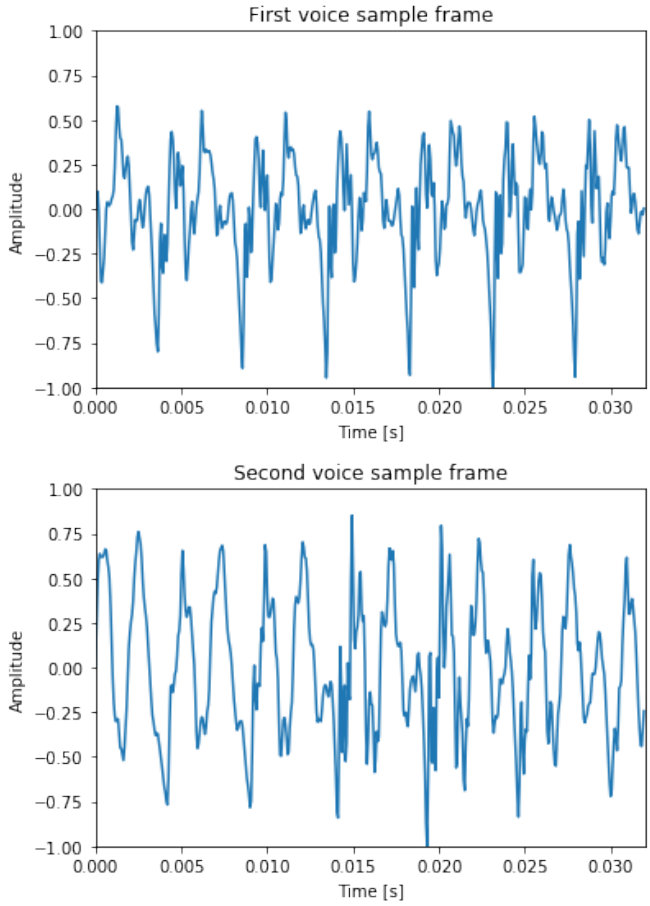
[10]:
```
# Fereastra de 20ms
window_length = int(20*1e-3*sampling_frequency)
print ("20ms equals %d samples" %window_length)
# Ajustăm fereastra la o putere a lui 2
window_fft = int(2**np.ceil(np.log2(window_length)))
```

```
print ("The FFT-adjusted window length is %d samples" \
       %window_fft)
```

[10]:
```
20ms equals 320 samples
The FFT-adjusted window length is 512 samples
```

Extragem un cadru de semnal din fiecare vocală:

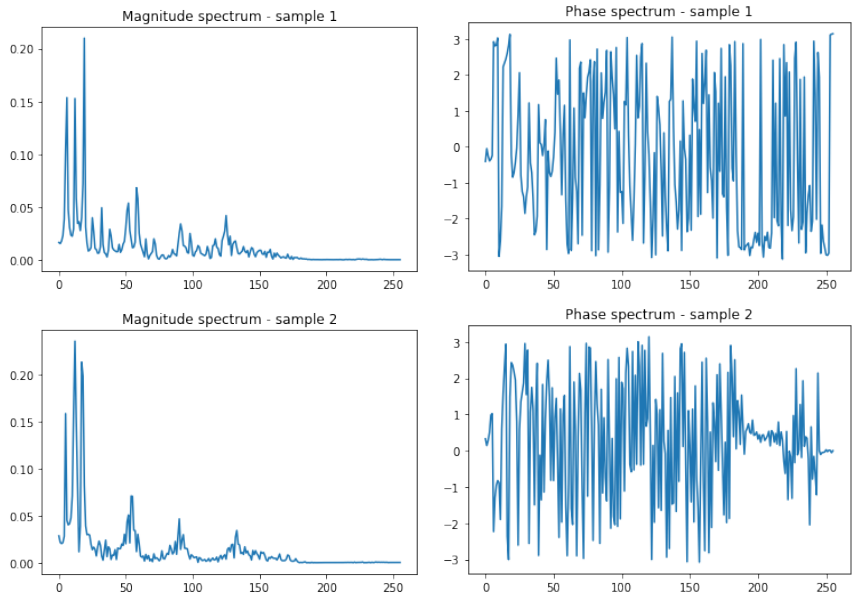
[11]:
```
# Fără suprapunere
p = 0
# Luăm primul cadru din ambele semnale
k = 0
current_frame_1 = wav_data_1[int(k*(1-p)*window_fft)\
                    :int((k*(1-p)+1)*window_fft)]
current_frame_2 = wav_data_2[int(k*(1-p)*window_fft)\
                    :int((k*(1-p)+1)*window_fft)]
time_axis = np.arange(0, window_fft)\
        *1.00/sampling_frequency
duration = window_fft*1.00/sampling_frequency
# Primul semnal
pl.figure()
pl.plot(time_axis,current_frame_1)
pl.xlim([0, duration])
pl.ylim([-1, 1])
pl.title('First voice sample frame')
pl.xlabel('Time [s]')
pl.ylabel('Amplitude')
# Al doilea semnal
pl.figure()
pl.plot(time_axis, current_frame_2)
pl.xlim([0, duration])
pl.ylim([-1, 1])
pl.title('Second voice sample frame')
pl.xlabel('Time [s]')
pl.ylabel('Amplitude');
```



Calculăm FFT pentru cele două cadre:

[12]:
```
frame_fft_1 = fft(current_frame_1, window_fft) \
        / (window_fft/2.0)
frame_fft_2 = fft(current_frame_2, window_fft) \
        / (window_fft/2.0)
# Spectrul de amplitudini și faze pentru primul cadru
# Vom folosi doar coeficienții corespunzători frecvențelor
# pozitive
magnitude_spectrum_1 = \
        np.abs(frame_fft_1[len(frame_fft_1)//2:][::-1])
phase_spectrum_1 = \
        np.angle(frame_fft_1[len(frame_fft_1)//2:][::-1])
# Spectrul de amplitudini și faze pentru al doilea cadru
# Vom folosi doar coeficienții corespunzători
# frecvențelor pozitive
```

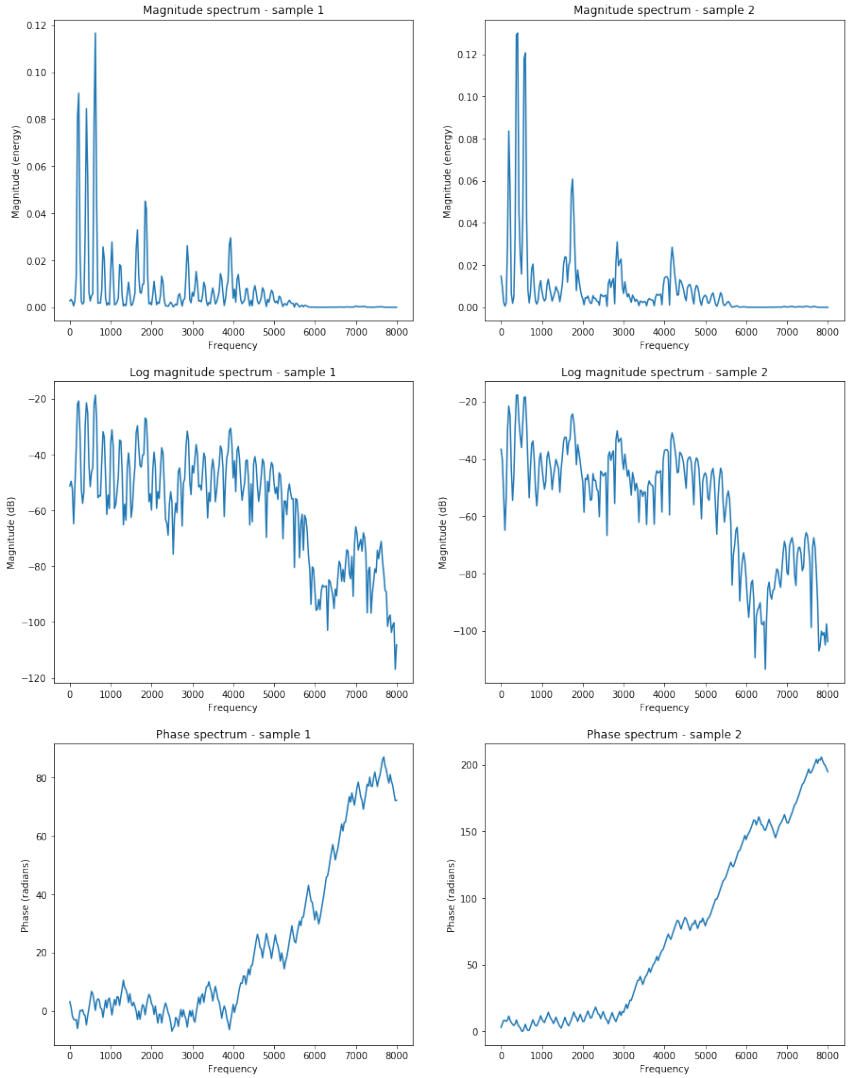
```
magnitude_spectrum_2 = \
        np.abs(frame_fft_2[len(frame_fft_1)//2:][::-1])
phase_spectrum_2 = \
        np.angle(frame_fft_2[len(frame_fft_1)//2:][::-1])
pl.figure()
pl.plot(magnitude_spectrum_1)
pl.title("Magnitude spectrum - sample 1")
pl.figure()
pl.plot(magnitude_spectrum_2)
pl.title("Magnitude spectrum - sample 2")
pl.figure()
pl.plot(phase_spectrum_1)
pl.title("Phase spectrum - sample 1")
pl.figure()
pl.plot(phase_spectrum_2)
pl.title("Phase spectrum - sample 2");
```



Exerciţiu T6.1.3 Ce ne spun plot-urile? ■

Haideți să vizualizăm rezultatele FFT folosind și o funcție specializată din modulul `matplotlib` și să utilizăm și o scară logaritmică pe axa oY pentru a comprima domeniul de valori de pe aceasta:

[13]:
```
pl.figure(figsize=(15,20))
# Primul sample
pl.subplot(3,2,1)
pl.magnitude_spectrum(current_frame_1, \
            Fs = sampling_frequency)
pl.title ("Magnitude spectrum - sample 1")
pl.subplot(3,2,3)
pl.magnitude_spectrum(current_frame_1, \
            Fs = sampling_frequency, scale = 'dB')
pl.title ("Log magnitude spectrum - sample 1")
pl.subplot(3,2,5)
A = pl.phase_spectrum(current_frame_1, \
            Fs = sampling_frequency)
pl.title ("Phase spectrum - sample 1")

# Al doilea cadru
pl.subplot(3,2,2)
pl.magnitude_spectrum(current_frame_2, \
            Fs = sampling_frequency)
pl.title ("Magnitude spectrum - sample 2")
pl.subplot(3,2,4)
pl.magnitude_spectrum(current_frame_2, \
            Fs = sampling_frequency, scale = 'dB')
pl.title ("Log magnitude spectrum - sample 2")
pl.subplot(3,2,6)
pl.phase_spectrum(current_frame_2, \
            Fs = sampling_frequency)
pl.title ("Phase spectrum - sample 2");
```



Se poate observa faptul că funcția dedicată spectrului de faze din cadrul modulului `matplotlib` aplică și procesul de phase unwrapping.

OBS T6.1 Ar trebui să fie clar faptul că în spectru de amplitudini componentele cu amplitudini diferite de 0 se regăsesc la valori multipli ai frecvenței fundamentale = *armonici*.

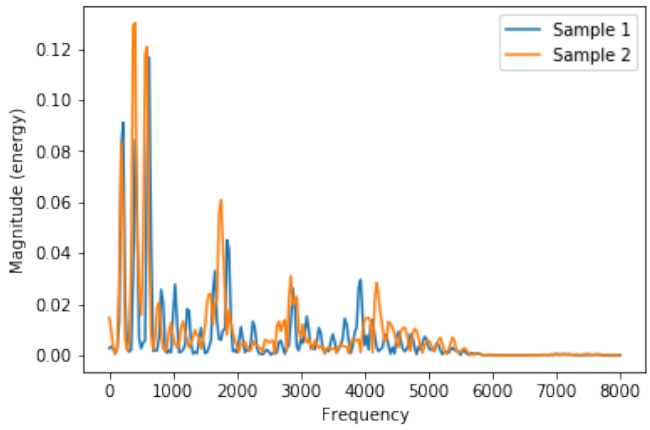
OBS T6.2 Valoarea maximă a frecvenței prezente în spectru este jumătate din frecvența de eșantionare conform teoremei lui Nyquist.

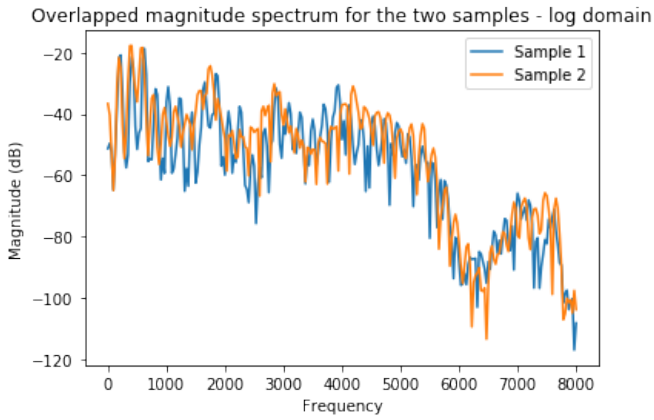
Exercițiu T6.1.4 Spectrele celor două vocale sunt mai apropiate decât forma lor de undă? ■

Exercițiu T6.1.5 Puteți determina frecvența fundamentală din spectru? Ce valoare are aceasta pentru fiecare vocală? Comparați cu rezultatele de la autocorelație sau AMDF. ■

Ca să fie și mai evidentă asemănarea celor două spectre de amplitudini a semnalelor, putem să le afișăm suprapuse:

[14]:
```
pl.magnitude_spectrum(current_frame_1, \
                Fs = sampling_frequency);
pl.magnitude_spectrum(current_frame_2, \
                Fs = sampling_frequency);
pl.legend(["Sample 1", "Sample 2"])
pl.figure()
pl.magnitude_spectrum(current_frame_1, \
        Fs = sampling_frequency, scale = 'dB');
pl.magnitude_spectrum(current_frame_2,
        Fs = sampling_frequency, scale = 'dB');
pl.title("Overlapped magnitude spectrum for \
         the two samples - log domain");
pl.legend(["Sample 1", "Sample 2"]);
```

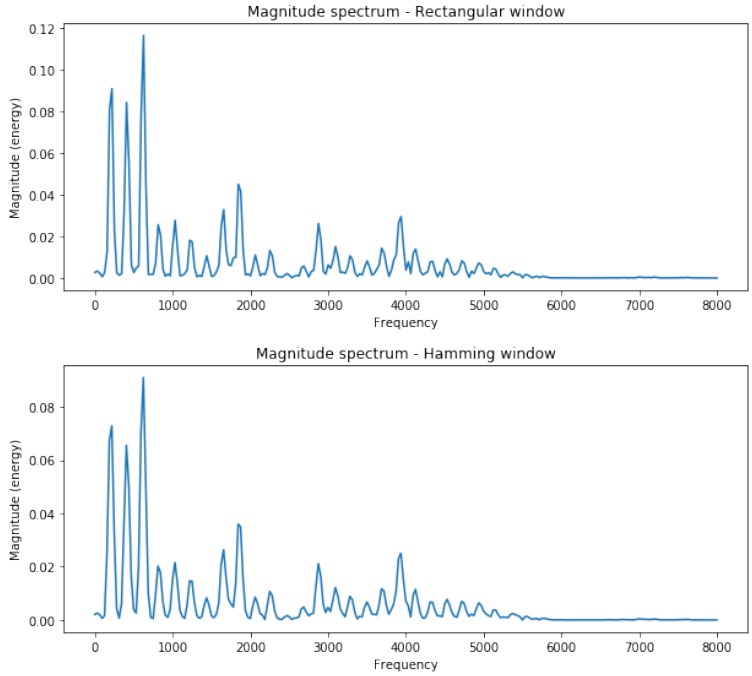




T6.1.3 Efectul tipului ferestrei de analiză în domeniul spectral

Ne reamintim faptul că într-un tutorial anterior am afișat spectrele diferitelor tipuri de ferestre de analiză și am discutat despre fenomenul de spectral leakage. Să vedem acum cum afectează utilizarea unei ferestre de analiză ponderate spectrul semnalului. Să folosim o fereastră Hamming:

[15]:
```
# Aplicăm Hamming asupra cadrului de semnal
hamming_frame = current_frame_1 * \
            np.hamming(window_fft)
# Calculăm FFT
frame_fft = fft(current_frame_1, window_fft) \
            / (window_fft/2.0)
hamming_frame_fft = fft(hamming_frame, window_fft) \
            / (window_fft/2.0)
# Afișăm rezultatele
pl.figure(figsize=(10,4))
pl.magnitude_spectrum(current_frame_1, \
            Fs = sampling_frequency)
pl.title ("Magnitude spectrum - Rectangular window")
pl.figure(figsize=(10,4))
pl.magnitude_spectrum(hamming_frame, \
            Fs = sampling_frequency)
pl.title ("Magnitude spectrum - Hamming window");
```



Exercițiu T6.1.6 S-a modificat spectrul odată cu modificarea tipului ferestrei de analiză? Motivați răspunsul. ■

T6.1.4 Inversa FFT

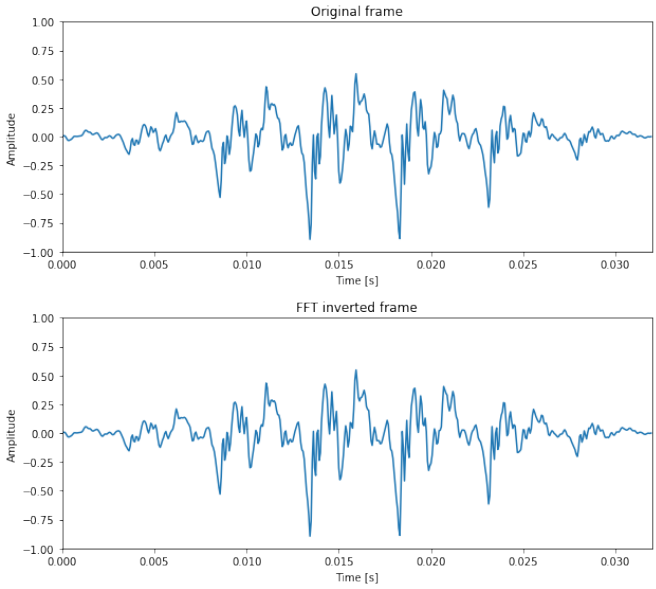
Transformata Fourier inversă va recompune semnalul în domeniul timp pornind de la coeficienții Fourier și este definită astfel :

$$x_n = \frac{1}{N} \sum_{k=0}^{N-1} X_k \cdot e^{i2\pi kn/N} \tag{T6.1.4}$$

Exercițiu T6.1.7 Ce se întâmplă dacă inversăm rezultatele FFT? Ce vom obține? ■

[16]:
```
from scipy import ifft
# Aplicăm IFFT asupra cadrului Hamming
inv_frame =ifft(hamming_frame_fft)
# Afișăm rezultatele
```

```
pl.figure(figsize=(10,4))
time_axis = np.arange(0, window_fft)*1.00/sampling_frequency
duration = window_fft*1.00/sampling_frequency
# Cadrul original
pl.plot(time_axis,hamming_frame)
pl.xlim([0, duration])
pl.ylim([-1, 1])
pl.xlabel('Time [s]')
pl.ylabel('Amplitude')
pl.title('Original frame')
# cadrul inversat
pl.figure(figsize=(10,4))
pl.plot(time_axis,window_fft/2*inv_frame.real)
pl.xlim([0, duration])
pl.ylim([-1, 1])
pl.xlabel('Time [s]')
pl.ylabel('Amplitude')
pl.title('FFT inverted frame');
```



Se poate observa că dacă nu am modificat în vreun fel coeficienții Fourier calculați anterior, rezultatul inversei FFT va fi identic cu semnalul original.

T6.2. Aplicații FFT: spectrograma

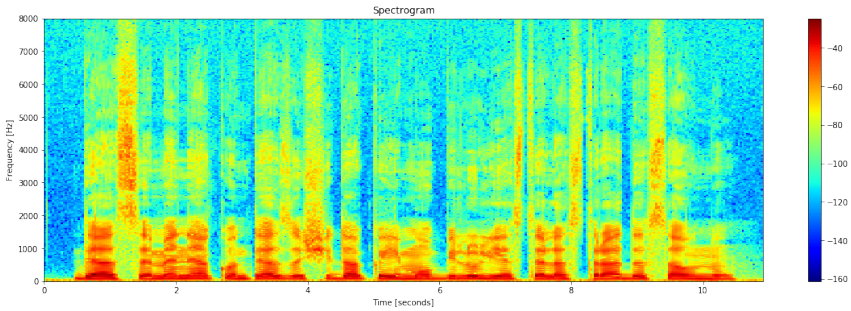
Vizualizarea spectrului de amplitudini al unui semnal vocal oferă informații suplimentare referitoare la conținutul acestuia, precum și informații privind periodicitatea sau aperiodicitatea segmentului vocal analizat. Este util însă în aplicații mai complexe să se analizeze nu doar spectrul instantaneu, ci și evoluția sa în timp. Această vizualizare timp-frecvență este denumită **spectrogramă**. Fiecare punct din această spectrogramă oferă informații referitoare la amplitudinea A_i a unei componente de frecvență f_i la momentul de timp t_j. Axa oX reprezintă **axa timpului**, axa oY **axa frecvențelor**, iar nivelele de gri sau culoarea dau amplitudinea normalizată a componentei de frecvență.

Exercițiu T6.2.1 Ce credeți? Care este frecvența maximă pe care o putem afișa într-o spectrogramă? Dar timpul maxim? ■

Să afișăm o spectrogramă folosind funcția `specgram` a modulului `matplotlib`. Vom folosi un semnal de durată mai mare pentru a putea interpreta informația mai ușor:

[17]:
```
from scipy.io.wavfile import read
from scipy import signal
from matplotlib import rcParams
# Utilizăm un set de culori specific
rcParams['image.cmap'] = 'jet'
input_wav_file = 'speech_files/adr_rnd1_001.wav'
wav_struct = wave.open(input_wav_file, 'r')
wav_bytes = wav_struct.readframes(-1)
wav_data = np.frombuffer(wav_bytes, dtype='int16')
wav_data = wav_data/float(max(abs(wav_data)))
# Plotăm spectrograma
pl.figure(figsize=(20,6))
pl.specgram(wav_data, NFFT=512, \
     Fs=sampling_frequency, noverlap=0, scale = 'dB')
```

```
pl.colorbar()
pl.title("Spectrogram")
pl.ylabel("Frequency [Hz]")
pl.xlabel("Time [seconds]");
```



Exercițiu T6.2.2 Cum se poate interpreta informația din spectrogramă? Puteți identifica anumite foneme pe baza ei? Sau marginile fonemelor? ■

Din simpla vizualizare a spectrogramei unui semnal vocal putem obține o serie de informații legate de conținutul acestuia. Cum ar fi prezența vocalelor dată de periodicitatea pe verticală sau energia mai mare acestora evidențiată de nivelel de roșu din reprezentare.

Pe de altă parte putem aproxima și marginile fonemelor într-o oarecare măsură: modificările bruște sau discontinuitățile din reprezentare pot indica aceste tranziții dintre foneme. Este important de menționat faptul că foneticienii exeperimentați pot să recunoască identitatea fonemelor direct din spectrogramă.

Exercițiu T6.2.3 Modificați parametrul NFFT din funcția specgram la valorile 32 și 2048. Ce se întâmplă? Cum explicați modificările apărute în spectrogramă? ■

Rezoluția în frecvență

În momentul în care alegem numărul de eșantioane pe baza cărora se va calcula FFT, implicit stabilim și rezoluția în frecvență a acesteia. Știind că frecvența maximă din spectrul unui semnal este egală cu jumătate din frecvența de eșantionare, putem calcula rezoluția în frecvență a FFT-ului.

De exemplu, dacă avem un semnal eșantionat la 16kHz și îl analizăm pe cadre de lungime 512 eșantioane, rezoluția în frecvență este egală cu:

$\Delta f = 16000 \div 2 \div 512 \approx 15.6\ Hz$

Ceea ce însemnă că noi vom putea calcula amplitudinile componentelor spectrale doar la multipli ai acestei rezoluții. Dacă dorim o rezoluție mai

bună trebuie să mărim numărul de puncte FFT și implicit lungimea ferestrei. Însă, dacă luăm cadre de analiză mai lungi, pierdem din rezoluția în timp. Astfel că este întodeauna necesar să realizăm un compromis între rezoluția în timp și rezoluția în frecvență a reprezentării spectrale și în special a spectrogramei sau să utilizăm cadre de analiză cu grad de suprapunere mare. Acest lucru s-a întâmplat și în cadrul exercițiului anterior unde am vizualizat spectrograma pentru diferite valori ale lui NFFT.

T6.3. Aplicații FFT: bancuri de filtre

O altă aplicație importantă a FFT este de a proiecta așa numitele **bancuri de filtre**. Acestea sunt seturi de filtre de lungime fixă sau variabilă ce acoperă întreg domeniul de frecvențe al unui semnal. Bancurile de filtre sunt utilizate pentru a comprima sau analiza informația conținută de semnal prin medierea spectrului din benzile respective de frecvență.

Cel mai utilizat banc de filtre în prelucrarea semnalului vocal este **bancul de filtre Mel**.[1] Scala de frecvențe neliniară Mel este bazată pe caracteristicile psiho-acustice ale auzului uman, iar un Mel este definit ca:

$$m = 2595 log_{10}(1 + \frac{f}{700}) = 1127 ln(1 + \frac{f}{700}) \; [Mel] \tag{T6.3.1}$$

Pentru a reveni la domeniul de frecvențe liniar utilizăm formula:

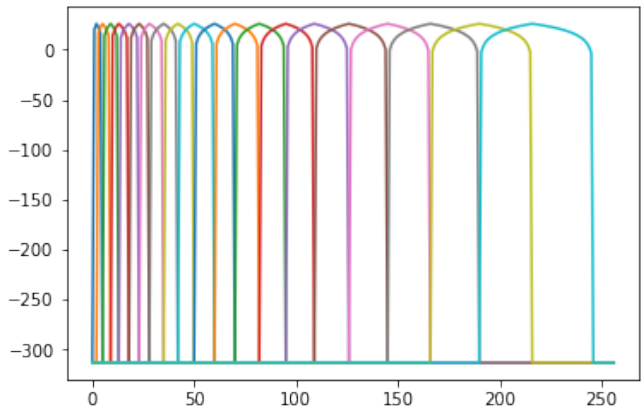
$$f = 700(10^{\frac{m}{2595}} - 1) = 700(e^{\frac{m}{1127}} - 1) \; [Hz] \tag{T6.3.2}$$

Să afișăm un astfel de banc de filtre:

[18]:
```
# 20 filtre
nfilt=20
NFFT = 512
pow_frames = 10
low_freq_mel = 0
# Convertim Hz la Mel
high_freq_mel = (2595 * \
    np.log10(1 + (sampling_frequency / 2) / 700))
# Filtre egal distanțate pe scală Mel
mel_points = np.linspace(low_freq_mel, \
                high_freq_mel, nfilt + 2)
# Convertim Mel în Hz
hz_points = (700 * (10**(mel_points \
```

[1] https://en.wikipedia.org/wiki/Mel_scale

```
                / 2595) - 1))
bin = np.floor((NFFT + 1) * hz_points \
                / sampling_frequency)
# Bancul de filtre
fbank = np.zeros((nfilt, \
            int(np.floor(NFFT / 2 + 1))))
for m in range(1, nfilt + 1):
    f_m_minus = int(bin[m - 1])
    f_m = int(bin[m])
    f_m_plus = int(bin[m + 1])
    for k in range(f_m_minus, f_m):
        fbank[m - 1, k] = (k - bin[m - 1]) \
            / (bin[m] - bin[m - 1])
    for k in range(f_m, f_m_plus):
        fbank[m - 1, k] = (bin[m + 1] - k) \
            / (bin[m + 1] - bin[m])
filter_banks = np.dot(pow_frames, fbank.T)
# Stabilitate numerică
filter_banks = np.where(filter_banks == 0,\
        np.finfo(float).eps, filter_banks)
# Afișăm în dB
filter_banks = 20 * np.log10(filter_banks)
freq_axis = np.arange(0, NFFT//2+1)\
        *sampling_frequency/NFFT
pl.plot(freq_axis, filter_banks)
pl.xlabel("Frequency [Hz]")
pl.title ("20 Mel filterbanks");
```



Se poate observa că pe baza scalei Mel de frecvențe, în domeniul liniar

vom obține benzi de frecvență foarte înguste în zona joasă-medie și benzi de frecvență mai largi în zona frecvențelor înalte. Acest lucru înseamnă că utilizăm o rezoluție de procesare mai bună în domeniul de frecvențe în care urechea umană este mai sensibilă și putem să reducem informația mai mult în domeniul în care nu este atât de sensibilă.

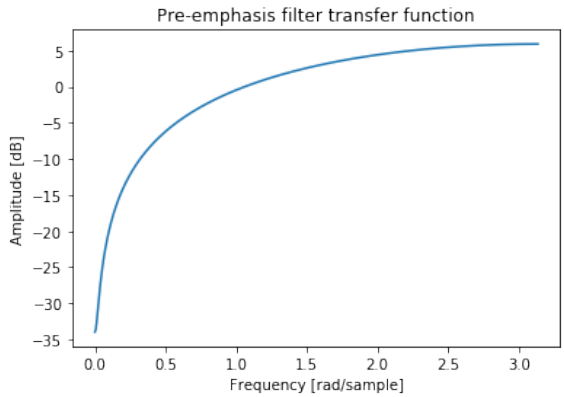
T6.4. Filtrul de pre-accentuare

Deoarece aerul introduce o atenuare de 6dB/octavă asupra sunetelor, atunci când facem înregistrări ale semnalului vocal, această atenuare se va traduce într-o pantă descrescătoare a spectrului (en. *spectral tilt*) și o reprezentare incorectă a frecvențelor medii și înalte inițiale.

Astfel că, un prim pas de pre-procesare în orice aplicație de voce este acela de a aplica un filtru de **pre-accentuare** asupra semnalului. Acest filtru este de tip trece sus și de cele mai multe ori de ordin întâi.

Să proiectăm un astfel de filtru și să-i afișăm caracteristica de transfer:

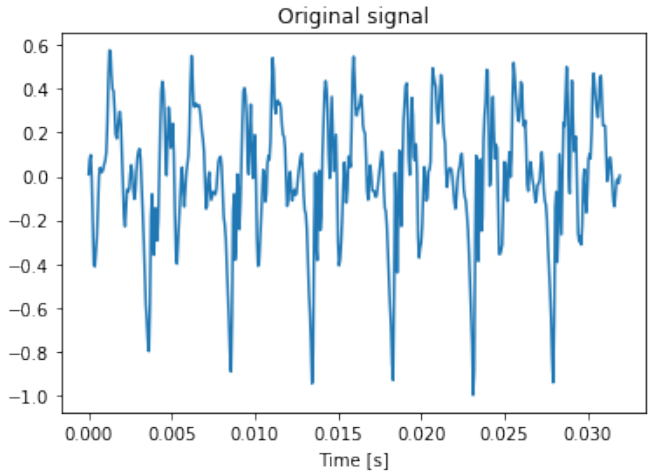
[19]:
```
from scipy.signal import freqz
# Coeficienții filtrului
b = [1, -0.98]
a = [1]
# Calculăm răspunsul filtrului în domeniul Z
w,h = freqz(b,a)
pl.plot(w, 20 * np.log10(abs(h)))
pl.title("Pre-emphasis filter transfer function")
pl.ylabel('Amplitude [dB]')
pl.xlabel('Frequency [rad/sample]');
```

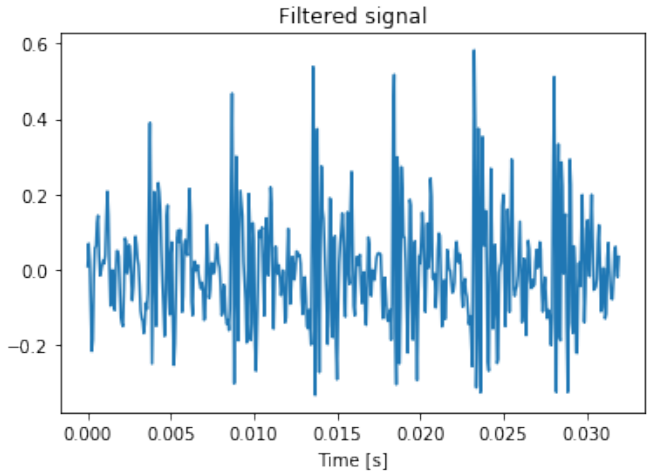


Exercițiu T6.4.1 Modificați valoarea celui de-al doilea coeficient b_k la 0.92 și 0.95 și afișați rezultatele. Cum se modifică funcția de transfer a filtrului? ■

Să vedem ce se întâmplă dacă aplicăm acest filtru asupra unui semnal vocal:

[20]:
```
from scipy.signal import lfilter
# Filtrăm una dintre vocale
filtered_data = lfilter(b, a, wav_data_1)
# Afișăm doar câteva eșantioane pentru a observa
# efectul filtrului
pl.figure()
pl.plot(wav_data_1[:window_length])
pl.title("Semnalul original")
pl.figure()
pl.plot(filtered_data[:window_length])
pl.title ("Semnalul filtrat");
```





Exerciţiu T6.4.2 Ce s-a modificat în semnal? Este același? Ascultați semnalul filtrat. ■

Exerciţiu T6.4.3 Cum se modifică spectrul semnalului? Afișați spectrul pentru semnalul original și cel filtrat și interpretați rezultatele. ■

T6.5. Concluzii

În acest tutorial am introdus analiza în frecvență a semnalului vocal--una dintre cele mai importante transformări utilizate în procesările de voce. Am afișat spectrul și spectrograma, am prezentat scala de frecvențe neliniară Mel, precum și ce înseamnă un banc de filtre sau filtrul de pre-accentuare.

BIBLIOGRAFIE SUPLIMENTARĂ

- S. V. Vaseghi, "Multimedia Signal Processing: Theory and applications in Speech, Music and Communications", John Wiley Sons, 2007
- Paul Taylor, "Text to speech synthesis", Cambridge University Press, 2009
- S. Renals, H. Shimodaira, Automatic Speech Recognition Course, University of Edinburgh, online slides: `http://www.inf.ed.ac.uk/teaching/courses/asr/2016-17/asr02-signal-handout.pdf`

RESURSE MEDIA

- NTi Audio Webinar - Basics of FFT Analysis
  `https://www.youtube.com/watch?v=AyUk-bZHJxI`
- NTi Audio Webinar - Advanced Topics of FFT Analysis
  `https://www.youtube.com/watch?v=SKfSkiufU34`

T7 Analiza cepstrală

În aplicații de codare, recunoaștere și sinteză a semnalului vocal, este foarte importantă reducerea dimensiunii și a complexității parametrilor de intrare. Reamintim faptul că semnalul vocal are un grad înalt de redundanță, atât în domeniul timp, cât și în frecvență. Pentru a reduce această redundanță, se folosesc metode de modelare matematică a semnalului vocal. Printre cele mai cunoscute se numără: modelul liniar-separabil (sursă-filtru), modelul undelor glotale sau modelul tuburilor acustice. Cel mai des utilizat, însă, este **modelul sursă-filtru** sau liniar separabil.

T7.1. Modelul sursă-filtru de producere a vorbirii

Modelul sursă-filtru (MSF) de modelare a semnalului vocal pornește de la o simplificare extremă a tipurilor de semnale vocale și face separarea lor strict pe baza sonorității (sonor-nesonor). Sonoritatea se referă la prezența sau absența periodicității din forma de undă, periodicitatea fiind un rezultat al utilizării corzilor vocale în fonație. Astfel că, transpus în noțiuni lingvistice se referă la discriminarea vocale - consoane. Totodată consideră totalitatea organelor fonatoare (cu excepția plămânilor și a corzilor vocale) ca fiind un simplu fitru ce modulează sursa de aer, fie ea periodică, în cazul vocalelor, sau aperiodică, în cazul consoanelor.

Ca urmare, MSF va modela sursa pentru semnalele sonore ca fiind un tren de impulsuri distanțate cu perioada fundamentală (T_0), iar pentru semnalele nesonore, sursa va fi un zgomot alb gaussian. În funcție de semnalul vocal dorit la ieșirea modelului, filtrul dat de tractul vocal va fi implementat folosind coeficienții estimați pentru acesta.

În figura de mai jos este prezentată diagrama modelului sursă filtru.

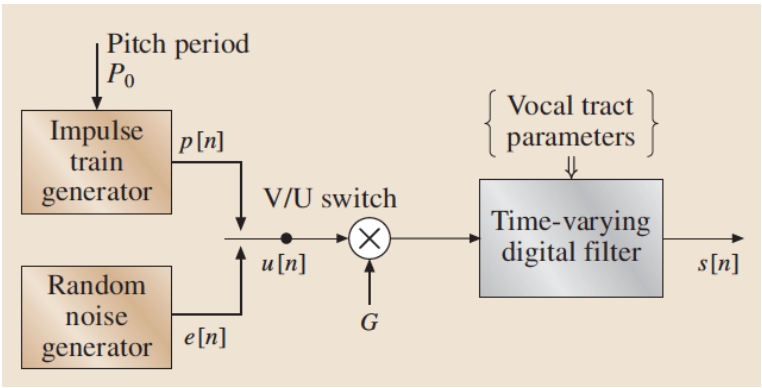


Fig.T7.1. Modelul sursă-filtru de producere a vorbirii

T7.2. Cepstrumul

Pentru a putea utiliza MSF este nevoie ca informațiile legate atât de sursă, cât și de filtru să poată fi extrase din semnalul vocal. Însă, conform teoriei semnalelor, ieșirea unui sistem va fi dată de convoluția dintre intrare $s(n)$ și funcția de transfer a sistemului $h(n)$:

$$y[n] = x[n] \otimes h[n] \tag{T7.2.1}$$

În cazul semnalului vocal, sursa este dată de fluxul de aer expirat și modulat (sau nu) de vibrația corzilor vocale, iar filtrul sau sistemul este reprezentat de tractul vocal.

Pentru a putea separa sursa de filtru, este nevoie de o operație matematică homomorfică ce va transforma operația de convoluție (neliniară) într-o operație liniară, cum ar fi adunarea. O astfel de transformare sau filtrare este și cepstrumul.

Cepstrumul este definit ca fiind transformata Fourier inversă a logaritmului modulului transformatei Fourier directe:

$$c[n] = \mathcal{F}^{-1}\{log|\mathcal{F}(x[n])|\} \tag{T7.2.2}$$

Domeniul de definiție al cepstrumului nu va fi însă timpul, deoarece din transformata Fourier s-a păstrat doar spectrul de amplitudini, informația de fază fiind eliminată prin aplicarea modulului. Acest nou domeniu este denumit **quefrency** pentru a face diferențierea dintre domeniul de frecvență și acesta.

Urmărind operațiile matematice aplicate asupra semnalului de intrare, se poate observa că transformata Fourier va transforma convoluția în înmulțire, iar logaritmarea va transforma operația de înmulțire în adunare. Ca urmare, cepstrumul va fi alcătuit din **suma reprezentării cepstrale a sursei și reprezentarea cepstrală a tractului vocal**.

Pentru a putea aplica și înțelege analiza cepstrală, mai întâi citim două semnale, unul sonor și unul nesonor:

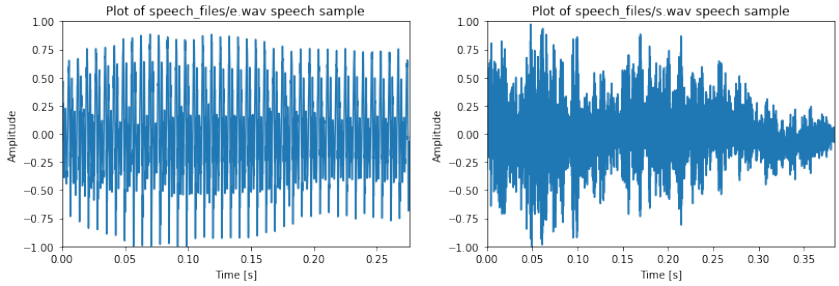
[1]:
```
import wave
import numpy as np
######################
# Citim vocala
input_wav_vowel = 'speech_files/e.wav'
wav_struct_vowel = wave.open(input_wav_vowel, 'r')
sampling_frequency = wav_struct_vowel.getframerate()
wav_bytes_vowel = wav_struct_vowel.readframes(-1)
wav_data_vowel = np.frombuffer(wav_bytes_vowel, \
                               dtype='int16')
wav_data_vowel = wav_data_vowel\
                /float(max(abs(wav_data_vowel)))
wav_struct_vowel.close()
######################
# Citim consoana
input_wav_consonant = 'speech_files/s.wav'
wav_struct_consonant = wave.open(input_wav_consonant, 'r')
wav_bytes_consonant = wav_struct_consonant.readframes(-1)
wav_data_consonant = np.frombuffer(wav_bytes_consonant, \
                                   dtype='int16')
wav_data_consonant = wav_data_consonant\
                /float(max(abs(wav_data_consonant)))
wav_struct_consonant.close()
```

Să le și afișăm și ascultăm:

[2]:
```
import matplotlib.pylab as pl
%matplotlib inline
######################
# Plot vocală
time_axis_vowel = np.arange(0, len(wav_data_vowel))\
            *1.00/sampling_frequency
duration_vowel = len(wav_data_vowel)\
            *1.00/sampling_frequency
print ("Vowel duration %f second" %duration_vowel)
pl.plot(time_axis_vowel, wav_data_vowel)
pl.xlim([0, duration_vowel])
pl.ylim([-1, 1])
pl.xlabel('Time [s]')
pl.ylabel('Amplitude')
pl.title("Plot of %s speech sample" \
            %input_wav_vowel)
```

```
######################
# Plot consoana
time_axis_consonant = \
        np.arange(0, len(wav_data_consonant))\
        *1.00/sampling_frequency
duration_consonant = len(wav_data_consonant)\
        *1.00/sampling_frequency
print ("Consonant duration %f second" %duration_consonant)
pl.figure()
pl.plot(time_axis_consonant, wav_data_consonant)
pl.xlim([0, duration_consonant])
pl.ylim([-1, 1])
pl.xlabel('Time [s]')
pl.ylabel('Amplitude')
pl.title("Plot of %s speech sample" %input_wav_consonant);
```

[2]:
```
Vowel duration 0.275625 second
Consonant duration 0.383813 second
```



[3]:
```
import IPython
IPython.display.Audio(wav_data_vowel, \
        rate=sampling_frequency)
```

[3]:

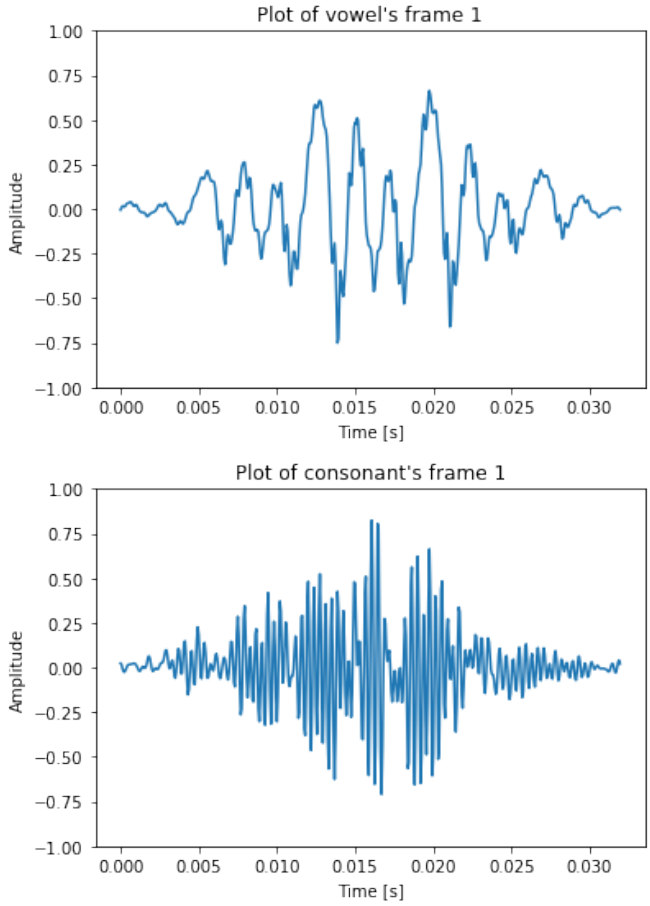

[4]:
```
IPython.display.Audio(wav_data_consonant, \
        rate=sampling_frequency)
```

[4]:


Și tot la fel ca în tutorialele anterioare, să extragem câte un cadru de analiză din fiecare semnal:

[5]:
```
# Fereastra de analiză cu lungime egală cu
# putere a lui 2 fără suprapunere
window_length = int(20*1e-3*sampling_frequency)
window_fft = int(2**np.ceil(np.log2(window_length)))
print ("Window length: %d samples" %window_fft)
p = 0
# Primul cadru
k = 0
# Cadrul de vocală
vowel_frame = wav_data_vowel[int(k*(1-p))\
            *window_fft: int((k*(1-p)+1))*window_fft]
# Fereastră Hamming
vowel_frame = np.hamming(window_fft) * vowel_frame
# Cadru de consoană
consonant_frame = wav_data_consonant[int(k*(1-p))\
            *window_fft: int((k*(1-p)+1))*window_fft]
# Fereastră Hamming
consonant_frame = np.hamming(window_fft) * consonant_frame
# Afișăm cadrele extrase
pl.figure()
time_axis = np.arange(0, window_fft)\
                *1.00/sampling_frequency
pl.plot(time_axis, vowel_frame)
pl.ylim([-1, 1])
pl.xlabel('Time [s]')
pl.ylabel('Amplitude')
pl.title("Plot of vowel's frame %i" %(k+1))
pl.figure()
pl.plot(time_axis, consonant_frame)
pl.ylim([-1, 1])
pl.xlabel('Time [s]')
pl.ylabel('Amplitude')
pl.title("Plot of consonant's frame %i" %(k+1));
```

[5]:
```
Window length: 512 samples
```



Odată ce avem la dispoziție cele două cadre de semnal, putem calcula cepstrumul sau coeficienții cepstrali folosind formula de definiție.

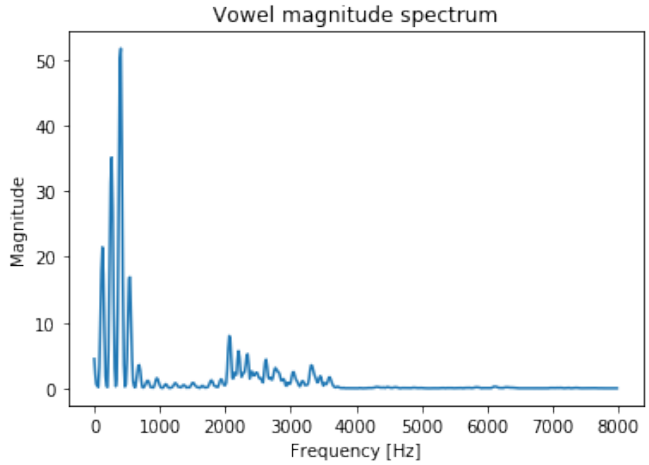
În secvența de cod următoare vor fi afișați și pașii intermediari ai analizei cepstrale și efectul lor asupra semnalului:

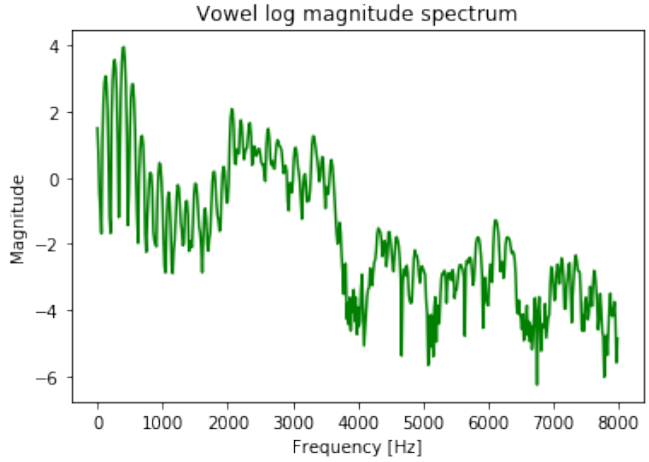
> **OBS T7.1** Instrucțiunile de mai jos calculează cepstrul real și nu cel complex.

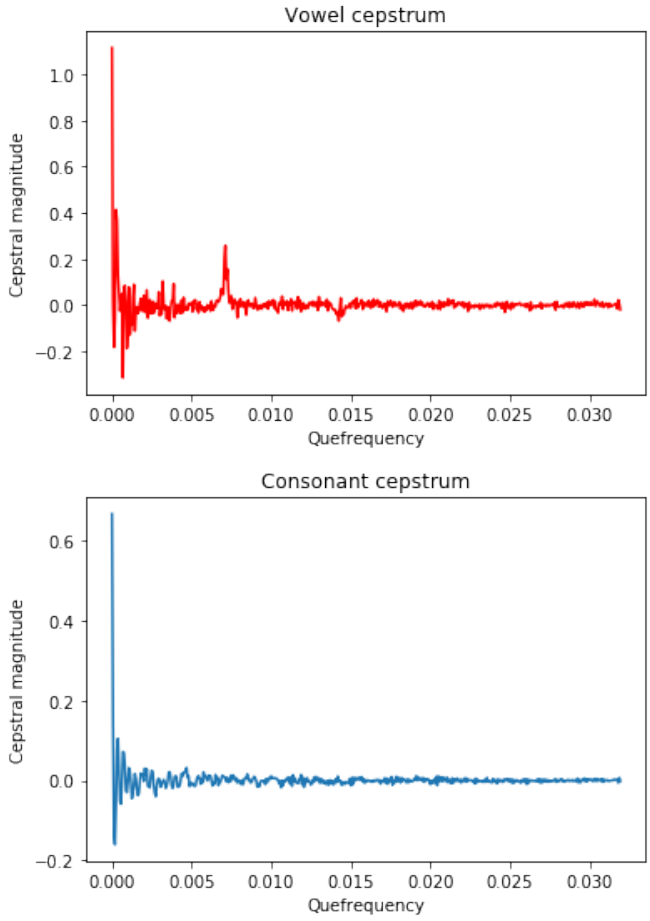
[6]:
```
from scipy.fftpack import fft
from scipy.fftpack import ifft
import numpy as np
# Def cepstrum: inversa Fourier a logaritmului
# modulului transformatei Fourier a semnalului
```

```
# 1) FFT
# OBS: pentru a evita fenomenul de alias,
# numărul de puncte FFT trebuie să fie mai mare
# decât lungimea ferestrei
nfft = 1024
fft_vowel = fft(vowel_frame, nfft)
# 2) Modulul FFT = spectrul de amplitudini
abs_fft_vowel = np.abs(fft_vowel)
pl.figure()
frequency_axis = np.arange(nfft//2)\
            *sampling_frequency/nfft
pl.plot(frequency_axis, \
            abs_fft_vowel[nfft//2:][::-1],)
pl.xlabel("Frequency [Hz]")
pl.ylabel("Magnitude")
pl.title("Vowel magnitude spectrum")
# 3) Logaritm din modul de FFT
log_abs_fft_vowel = np.log(abs_fft_vowel)
pl.figure()
pl.plot(frequency_axis, \
        log_abs_fft_vowel[nfft//2:][::-1], 'g')
pl.xlabel("Frequency [Hz]")
pl.ylabel("Magnitude")
pl.title("Vowel log magnitude spectrum")
# 4) CEPSTRUM = IFFT de logaritm din modul de FFT
vowel_cepstrum = ifft(log_abs_fft_vowel)
pl.figure()
que_axis = np.arange(0, window_fft)\
            *1.00/sampling_frequency
pl.plot(time_axis, \
        vowel_cepstrum[nfft//2:][::-1],'r')
pl.xlabel("Quefrequency")
pl.ylabel("Cepstral magnitude")
pl.title("Vowel cepstrum")
# Pentru consoană vom calcula coeficienții
# într-o singură instrucțiune
consonant_cepstrum = ifft(np.log10(\
        np.abs(fft(consonant_frame, nfft))))
# Afișăm cepstrumul
pl.figure()
pl.plot(time_axis, consonant_cepstrum[nfft//2:][::-1])
```

```
pl.xlabel("Quefrequency")
pl.ylabel("Cepstral magnitude")
pl.title("Consonant cepstrum");
```







Exerciţiu T7.2.1 Ce puteți spune despre cele două cepstrumuri? ■

Exerciţiu T7.2.2 Se poate calcula F0 din cepstrumul vocalei? Ce valoare are pentru cadrul curent? ■

Exerciţiu T7.2.3 Calculați și comparați cepstrumul pentru cele două vocale utilizate în tutorialul anterior (Transformata Fourier). Ce se poate observa? ■

Exerciţiu T7.2.4 Ce aplicații ar putea avea cepstrumul? ■

T7.3. Procesul de liftare

Mergând mai departe și folosind procesul de **liftare**, putem separa din cadrul cepstrumului caracteristicile sursei de caracteristicile filtrului. Ca urmare a operațiilor din formula de calcul a cepstrumului, informațiile legate de filtru vor fi reprezentate de valori mici ale quefrecvenței, iar cele legate de sursă de valorile mari. Astfel că liftarea se referă la separarea coeficienților de ordin mic (de obicei primii 20-30) de coeficienții cepstrali de ordin înalt. Convenția de denumire pentru aceste seturi de coeficienți este: **h** pentru coeficienții filtrului și **u** pentru coeficienții sursei.

Se poate observa în figura de mai jos acest proces de liftare. Totodată se poate observa și faptul că pentru un semnal periodic, cepstrumul va conține o serie maxime în partea superioară. Aceste maxime sunt date de F_0 și constituie o altă modalitate de calcul a frecvenței fundamentale a semnalului vocal.

Un alt aspect important al cepstrumului este evidențiat de calcu-

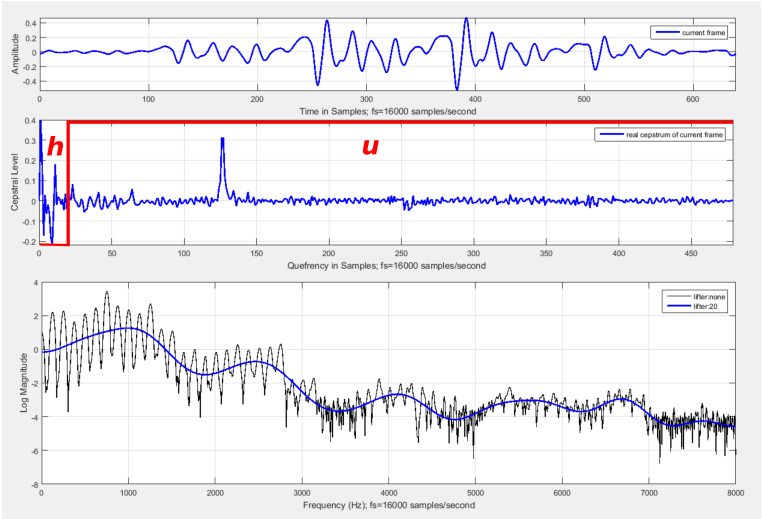


Fig.T7.2. Cepstrum și procesul de liftare - segment sonor

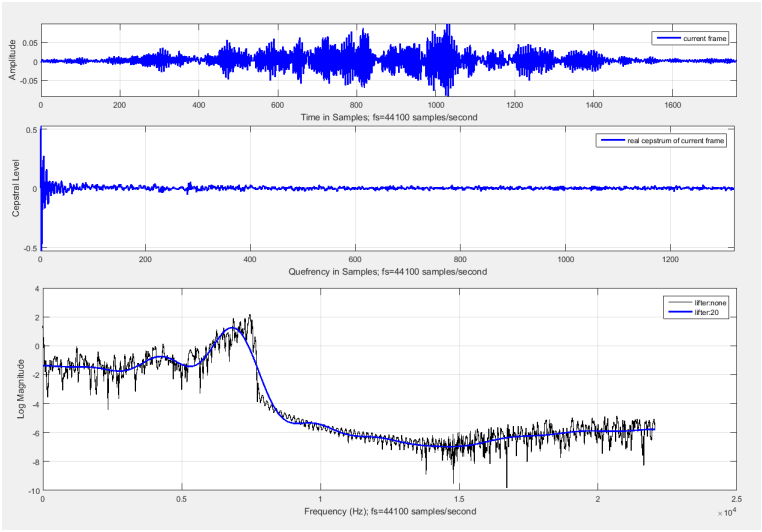


Fig.T7.3. Cepstrum și procesul de liftare - segment nesonor

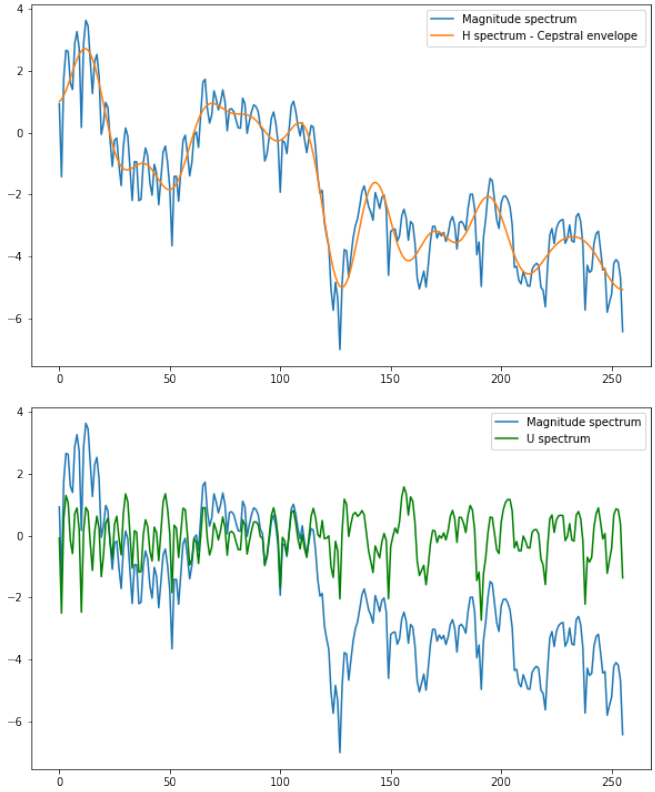
larea transformatei Fourier a coeficienților cepstrali de ordin jos, corespunzători tractului vocal. Eliminând componenta periodică dată de vibrația corzilor vocale prin procesul de liftare, spectrul nou obținut va pune în evidență mult mai bine anvelopa spectrală și odată cu aceasta și **formanții** = frecvențele de rezonanță ale tractului vocal.

Astfel că, dorim să vizualizăm spectrul dat de segmentele individuale ale procesului de liftare, *spectrul lui h* și *spectrul lui u*.

> **OBS T7.2** Vom calcula cepstrul complex de data aceasta. Față de cepstrul real, acesta calculează logaritmul complex și astfel ține cont și de faza componentelor de frecvență ale semnalului. Codul de mai jos conține și o serie de alte operații matematice necesare unui calcul corect a liftării. Înțelegerea completă a codului nu este necesară la acest moment. Rezultatul pașilor intermediari poate fi afișat pentru o mai bună înțelegere a lor.

[7]:
```
# Fereastră Hamming
hamm_frame = vowel_frame*np.hamming(window_fft)
# Rearanjăm cadrul de analiză
bw=np.concatenate((hamm_frame[int(window_fft/2):\
    window_fft], hamm_frame[:int(window_fft/2)]))
# Calculăm FFT
frame_fft=fft(bw,window_fft)
phase_rad=np.angle(frame_fft)
```

```
phase_rad_unwrap=np.unwrap(phase_rad);
# Logaritm complex
complex_logarithm = np.log(np.abs(frame_fft))\
            +phase_rad_unwrap*np.sqrt(-1+0j);
# Spectrul de amplitudini
magnitude_spectrum = np.real(complex_logarithm)
# Cepstrumul
ceps=ifft(complex_logarithm,window_fft)
# Creăm o fereastră rectangulară de ordin 20
# pentru a extrage separat coeficienții pentru h și u
lifter_index = 20
lifter = np.zeros(window_fft)
lifter[0:lifter_index+1] = 1
lifter[window_fft-lifter_index+1:window_fft] = 1
# Extragem H
cepsl=np.real(ceps*lifter)
h=np.real(fft(cepsl,window_fft))
# Extragem U
inv_lifter = (1-lifter)
cepsll = np.real(ceps*inv_lifter)
u = np.real(fft(cepsll,window_fft))
# Plotăm spectrul și anvelopa cepstrală (spectrul lui h)
pl.figure(figsize=(10,6))
pl.plot(magnitude_spectrum[window_fft//2:][::-1])
pl.plot(h[window_fft//2:][::-1])
pl.legend(["Magnitude spectrum",\
           "H spectrum - Cepstral envelope "])
# Plotăm spectrul semnalului și a lui u
pl.figure(figsize=(10,6))
pl.plot(magnitude_spectrum[window_fft//2:][::-1])
pl.plot(u[window_fft//2:][::-1], 'g')
pl.legend(["Magnitude spectrum","U spectrum" ]);
```



Din anvelopa cepstrală putem să observăm acum o mai bună evidențiere a formanților (benzile de frecvență cu energie ridicată). Iar din cele două spectre, ale lui h și u, putem să vedem faptul că însumarea lor ne va da spectrul de amplitudini al semnalului, ceea ce înseamnă că prin intermediul cepstrumului am reușit să facem separarea sursă-filtru.

Exercițiu T7.3.1 Modificați indexul coeficientului la care se realizează liftarea (variabila `lifter_index`) și interpretați rezultatele. ■

T7.4. Coeficienții Mel-cepstrali

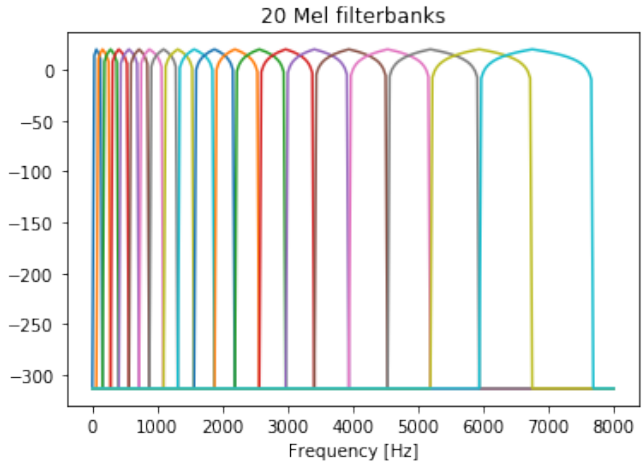
În practică, cea mai des întâlnită aplicație a cepstrumului este în recunuoașterea automată a vorbirii, pentru parametrizarea formei de undă a semnalului vocal și reducerea variabilității parametrilor la nivel de foneme intra și inter-vorbitor. În plus, pentru a caracteriza forma de undă și din punct de vedere perceptual, nu doar fizic, se folosesc scale de frecvență neliniare. Una dintre cele mai des utilizate este scala Mel, discutată și în tutorialul anterior.

Rezultatul acestor două procesări este cunoscut sub numele de **coeficienți Mel-cepstrali** (en. Mel Frequency Cepstral Coefficients - **MFCC**). Aceștia reprezintă energia medie din benzile de frecvență date un banc de filtre de lungime N, egal distanțate pe scala Mel.

Mai întâi reluăm din tutorialul anterior vizualizarea filtrelor egal distanțate pe scală Mel:

[8]:
```
# 20 filtre
nfilt=20
NFFT = 512
pow_frames = 10
low_freq_mel = 0
# Convertim Hz la Mel
high_freq_mel = (2595 * \
    np.log10(1 + (sampling_frequency / 2) / 700))
# Filtre egal distanțate pe scală Mel
mel_points = np.linspace(low_freq_mel, \
                high_freq_mel, nfilt + 2)
# Convertim Mel în Hz
hz_points = (700 * (10**(mel_points \
                / 2595) - 1))
bin = np.floor((NFFT + 1) * hz_points \
                / sampling_frequency)
# Bancul de filtre
fbank = np.zeros((nfilt, \
```

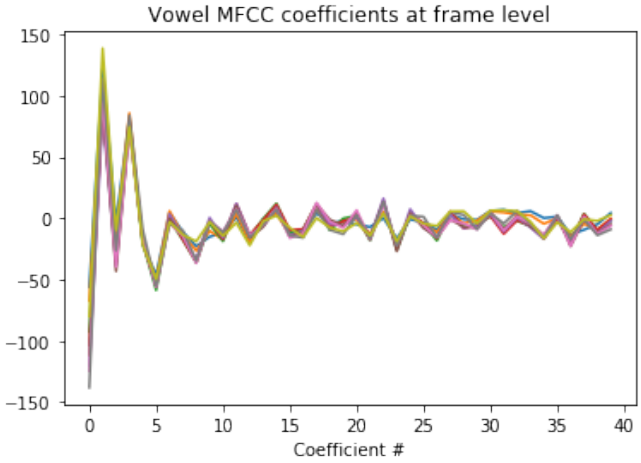
```
            int(np.floor(NFFT / 2 + 1))))
for m in range(1, nfilt + 1):
    f_m_minus = int(bin[m - 1])
    f_m = int(bin[m])
    f_m_plus = int(bin[m + 1])
    for k in range(f_m_minus, f_m):
        fbank[m - 1, k] = (k - bin[m - 1]) \
            / (bin[m] - bin[m - 1])
    for k in range(f_m, f_m_plus):
        fbank[m - 1, k] = (bin[m + 1] - k) \
            / (bin[m + 1] - bin[m])
filter_banks = np.dot(pow_frames, fbank.T)
# Stabilitate numerică
filter_banks = np.where(filter_banks == 0,\
        np.finfo(float).eps, filter_banks)
# Afișăm în dB
filter_banks = 20 * np.log10(filter_banks)
freq_axis = np.arange(0, NFFT//2+1)\
        *sampling_frequency/NFFT
pl.plot(freq_axis, filter_banks)
pl.xlabel("Frequency [Hz]")
pl.title ("20 Mel filterbanks");
```



Pentru a calcula coeficienții MFCC din cadrele semnalului de intrare, putem să utilizăm funcția disponibilă în modulul `librosa`. În cazul în care modulul nu este instalat, putem face acest lucru utilizând utilitarul `pip`

[9]:
```
!pip install librosa
```

[10]:
```
import librosa.feature as lb
# Extragem automat coeficienții MFCC din toate cadrele
# vocalei. Numărul de coeficienți extrași este egal cu 40
vowel_mfcc = lb.mfcc(wav_data_vowel, \
                     sampling_frequency, n_mfcc=40)

# plotăm pe același grafic coeficienții din cadrele
# de semnal ale vocalei
pl.title("Vowel MFCC coefficients at frame level")
pl.xlabel("Coefficient #")
pl.plot(vowel_mfcc);
```



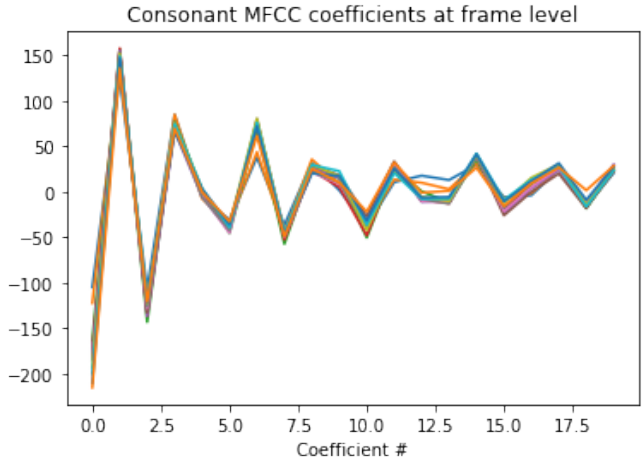
Se poate observa faptul că aceștia sunt foarte similari de-a lungul cadrelor de analiză.

Exercițiu T7.4.1 Care este diferența dintre coeficienții MFCC și cepstrum? ■

Exercițiu T7.4.2 Extrageți coeficienții MFCC din aceeași vocală de la mai mulți vorbitori. Directorul speech_files/ conține înregistrări ale vocalelor din limba română pronunțate de doi vorbitori diferiți. Plotați rezultatele. ■

Să vizualizăm și coeficienții Mel-cepstrali pentru o consoană:

[11]:
```
# Citim consoana
input_wav_file_2 = 'speech_files/s.wav'
wav_struct_2 = wave.open(input_wav_file_2, 'r')
wav_bytes_2 = wav_struct_2.readframes(-1)
wav_data_2 = np.frombuffer(wav_bytes_2, dtype='int16')
wav_data_2 = wav_data_2/float(max(abs(wav_data_2)))
sampling_frequency = wav_struct_2.getframerate()
wav_struct_2.close()
# Extragem coeficienții MFCC
consonant_mfccs = lb.mfcc(wav_data_2, sampling_frequency)
# Afișăm coeficienții
pl.title("Consonant MFCC coefficients at frame level")
pl.xlabel("Coefficient #")
pl.plot(consonant_mfccs);
```



Exercițiu T7.4.3 Extrageți coeficienții MFCC din aceeași consoană de la mai mulți vorbitori. Sunt similari și pentru consoane sau doar pentru vocale? ■

T7.5. Concluzii

În acest tutorial a fost prezentată metoda cepstrală de analiză a semnalului vocal. Aceasta se bazează pe o transformare homomorfică pentru a separa sursa de tractul vocal. Cepstrumul combinat cu scala neliniară de frecvențe Mel este aplicat în multiple metode de analiză, recunoaștere și sinteză a semnalului vocal pentru a modela tractul vocal și pentru a reduce variabilitatea parametrilor extrași din segmentele semnalului vocal.

BIBLIOGRAFIE SUPLIMENTARĂ

- L. Rabiner, B.-H. Juang, "Fundamentals of Speech Recognition", Prentice Hall, 1993
- X. Huang, A. Acero, H.-W. Hon, ”Spoken Language Processing: A Guide to Theory, Algorithm, and System Development”, Prentice Hall, 2001
- S. V. Vaseghi, “Multimedia Signal Processing: Theory and applications in Speech, Music and Communications”, John Wiley Sons, 2007
- Paul Taylor, ”Text to speech synthesis”, Cambridge University Press, 2009

RESURSE MEDIA

- MIT, "Brains, Minds and Machines" - Summer Course, Unit 7: Audition and Speech, online: `https://ocw.mit.edu/resources/res-9-003-brains-minds-and-machines-summer-course-summer-2015/unit-7.-audition-and-speech`
- S. Renals, H. Shimodaira, Automatic Speech Recognition Course, University of Edinburgh, online slides: `http://www.inf.ed.ac.uk/teaching/courses/asr/2016-17/asr02-signal-handout.pdf`

T8 Analiza și sinteza prin predicție liniară

T8.1. Coeficienții de predicție liniară

Analiza prin predicție liniară (en. *Linear Prediction Analysis*) este o altă metodă, alături de cepstrum, de separare a sursei de filtru din modelul sursă-filtru de producere a vorbirii. Principiul fundamental al acestei analize este bazat pe gradul înalt de corelație al eșantioanelor semnalului vocal. Această corelație este dată de inerția organelor fonatoare, astfel încât sunetul emis nu poate fi modificat într-un interval de timp foarte mic. Tot această inerție stă și la baza cvasi-staționarității semnalului vocal.

Datorită acestei corelații eșantioanele de semnal pot fi estimate ca o sumă ponderată a eșantioanele anterioare:

$$\hat{y[n]} = \sum_{k=1}^{p} a_k y[n-k] \tag{T8.1.1}$$

unde p este ordinul de predicție. Eroarea de predicție este dată de:

$$e[n] = y[n] - \hat{y[n]} = \sum_{k=1}^{p} y[n-k] \tag{T8.1.2}$$

Dacă trecem în domeniul z obținem:

$$E(z) = Y(z) - \sum_{k=1}^{p} a_k z^{-k} Y(z) \tag{T8.1.3}$$

Împărțim ambii termeni cu $Y(z)$ și inversăm ecuația:

$$\frac{Y(z)}{E(z)} = \frac{1}{1 - \sum_{k=1}^{p} a_k z^{-k}} \tag{T8.1.4}$$

Această ecuație seamănă cu o funcție de transfer ce conține doar poli. Acest lucru este în conformitate și cu un alt model de producere a vorbirii derivat

din principii de acustică teoretică. Acesta spune că tractul vocal poate fi modelat cu un set finit de tuburi de diferite lungimi și raze. Aceste tuburi introduc fiecare o pereche de poli complex conjugați în funcția de transfer a tractului vocal. Drept urmare, putem scrie $H(z)$ ca un filtru ce are doar poli:

$$H(z) = \frac{1}{1 - \sum_{k=1}^{p} a_k z^{-k}} \tag{T8.1.5}$$

Pornind de la aceste două observații putem conclude faptul că eroare de predicție din ecuațiile inițiale nu este altceva decât sursa semnalului vocal și anume oscilația corzilor vocale sau fluxul de aer nemodulat expirat din plămâni.

Deci, pentru a determina sursa și filtrul din modelul de producere a vorbirii este suficient să determinăm coeficienții a_k ai funcției de transfer anterioare. Acești coeficienți sunt denumiți **coeficienți de predicție liniară** (en. *linear prediction coefficients* (LPC)). Calculul lor implică rezolvarea unui sistem de ecuații de ordin p, iar pentru aceasta există o serie de metode matematice de rezolvare rapidă, dintre care cea mai des folosită este recursivitatea Levinson-Durbin https://en.wikipedia.org/wiki/Levinson_recursion . Detalierea algoritmilor de calcul ai coeficienților LPC nu face parte din scopul acestei cărți și lăsăm la latitudinea cititorului aprofundarea acestora.

Să vedem acum ce informații ne oferă coeficienții LPC despre semnalul vocal. Să citim mai întâi două semnale: sonor și nesonor:

[1]:
```
import wave
import numpy as np
######################
# Citim vocala
input_wav_vowel = 'speech_files/a.wav'
wav_struct_vowel = wave.open(input_wav_vowel, 'r')
sampling_frequency = wav_struct_vowel.getframerate()
wav_bytes_vowel = wav_struct_vowel.readframes(-1)
wav_data_vowel = np.frombuffer(wav_bytes_vowel, \
        dtype='int16')
wav_data_vowel = wav_data_vowel \
        /float(max(abs(wav_data_vowel)))
wav_struct_vowel.close()
######################
# Citim consoana
input_wav_consonant = 'speech_files/s.wav'
wav_struct_consonant = wave.open(input_wav_consonant, 'r')
```

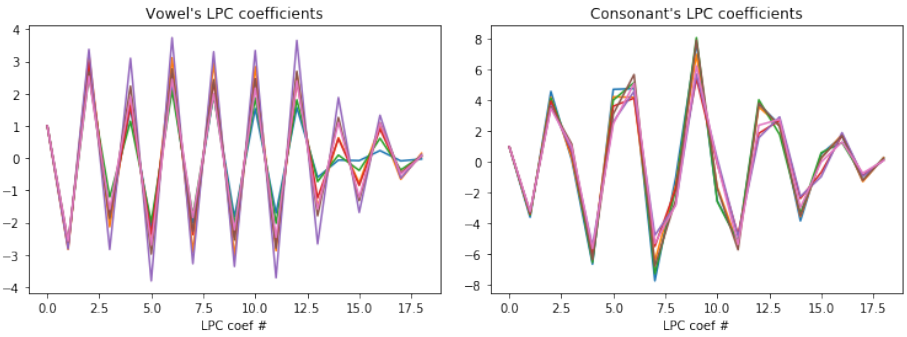
```
wav_bytes_consonant = wav_struct_consonant.readframes(-1)
wav_data_consonant = np.frombuffer(wav_bytes_consonant,\
         dtype='int16')
sampling_frequency_c = wav_struct_consonant.getframerate()
wav_data_consonant = wav_data_consonant  \
        /float(max(abs(wav_data_consonant)))
wav_struct_consonant.close()
```

Pentru a extrage coeficienții LPC din fiecare cadru de semnal, vom folosi din nou modulul `librosa`, submodulul `core` ce conține funcția `lpc()`. Această funcție ia ca intrare un semnal și un ordin al predictorului și returnează coeficienții LPC, inclusiv termenul liber din numitorul funcției de transfer.

OBS T8.1 Ordinul predictorului pentru semnalele vocale a fost stabilit empiric de către Fant ca fiind egal cu frecvența de eșantionare exprimată în kHz plus 2. Această formulă se bazează pe observația că în medie într-un semnal vocal nu poate să existe mai mult un pol la fiecare kilohertz. Astfel că alegerea unui ordin egal cu $[F_s] + 2$ funcționează și în practică.

[2]:
```
import librosa.core as lb

from scipy.signal import hamming
import matplotlib.pylab as pl
%matplotlib inline

# Fereastra de analiză cu lungime egală cu putere
# a lui 2 fără suprapunere
window_length = int(20*1e-3*sampling_frequency)
window_fft = int(2**np.ceil(np.log2(window_length)))
p = 0
# Fereastră Hamming
hamming_window = hamming(window_fft)
# Numărul de cadre
number_of_frames = int(len(wav_data_vowel)/window_fft)
# Stabilim ordinul LPC la Fs + 2
lpc_order = sampling_frequency//1000 + 2
# Inițializăm o matrice nulă în care vom stoca valorile
# coeficienților LPC din fiecare cadru. Numărul
# de coeficienți LPC returnat de funcție este egal
# cu ordinul LPC+1 datorită termenului liber
lpcs = np.zeros ([number_of_frames, lpc_order+1])
```

```
for k in range(number_of_frames):
    # Extragem doar un cadru din semnal
    current_frame = wav_data_vowel[k*window_fft:  \
            (k+1)*window_fft]
    hamming_frame = np.multiply(hamming_window,  \
            current_frame)
    lpcs[k,:] = lb.lpc(hamming_frame, lpc_order)
# Plot
pl.plot(np.transpose(lpcs))
pl.title("Vowel's LPC coefficients")
pl.xlabel('LPC coef #');

# Numarul de cadre din consoana
number_of_frames = int(len(wav_data_consonant)/window_fft)
lpcs = np.zeros ([number_of_frames, lpc_order+1])
for k in range(number_of_frames):
    # Extragem doar un cadru din semnal
    current_frame = wav_data_consonant[k*window_fft:  \
            (k+1)*window_fft]
    hamming_frame = np.multiply(hamming_window,  \
            current_frame)
    lpcs[k,:] = lb.lpc(hamming_frame, lpc_order)
# Plot
pl.figure()
pl.plot(np.transpose(lpcs))
pl.title("Consonant's LPC coefficients")
pl.xlabel('LPC coef #');
```



Se poate observa faptul că valorile acestor coeficienți sunt constante de-a lungul celor două segmente vocale și că valoarea primului coeficient returnat de funcția `lpc()` este întotdeauna 1. Valorile constante ale coeficienților

LPC ne indică faptul că tractul vocal și filtrul determinat de acesta nu se modifică. Ceea ce este adevărat atât timp cât segmentul vocal conține o singură fonemă cu caracteristici statice, cum sunt vocalele sau anumite consoane.

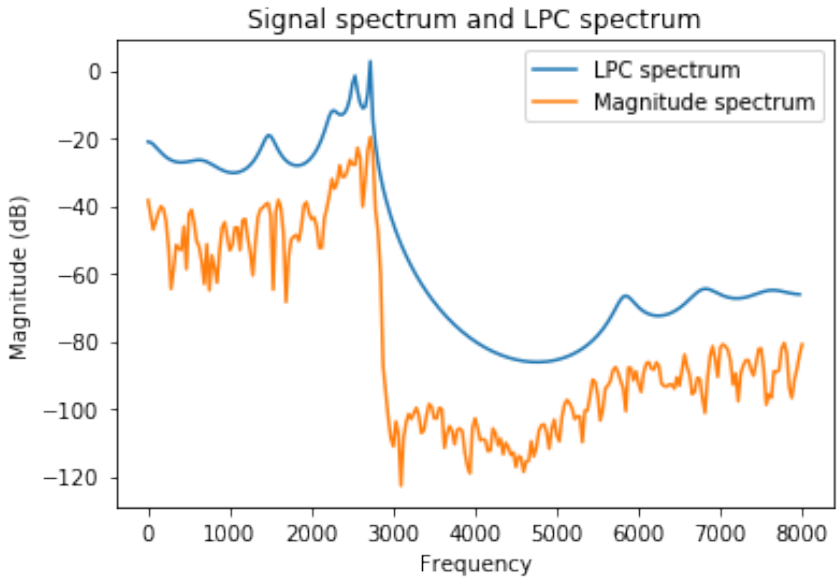
Exercițiu T8.1.1 Afișați valorile coeficienților LPC pentru o altă consoană cu caracteristici mai dinamice, cum ar fi p, c, d, etc. ■

Exercițiu T8.1.2 Afișați valorile coeficienților LPC pentru un segment vocal ce conține mai multe foneme. Sunt valorile coeficienților LPC constante? ■

T8.2. Spectrul LPC

Știind că acești coeficienți LPC sunt de fapt coeficienții unui filtru, putem să vizualizăm răspunsul său în frecvență. Vom afișa acest răspuns alături de spectrul semnalului, pentru a putea identifica eventualele similitudini:

[3]:
```
from scipy.signal import freqz
# Extragem un singur cadru al vocalei
k = 3
vowel_frame = wav_data_vowel[k*window_fft: (k+1)*window_fft]
hamming_frame = np.multiply(hamming_window, current_frame)
a = lb.lpc(hamming_frame, lpc_order)
# Obținem răspunsul în frecvență al filtrului dat de
# coeficienții LPC. Lungimea răspunsului o luăm egală
# cu numărul de puncte FFT al spectrului semnalului
w, h = freqz(1, a , window_fft//2)
# Axa frecvenței
freq_axis = np.arange(window_fft//2)*sampling_frequency \
        /window_fft
# Afișăm pe axă logaritmică spectrul LPC
pl.plot(freq_axis, 20*np.log10(1.0/window_fft*abs(h)))
# Spectrul semnalului
pl.magnitude_spectrum(hamming_frame,  \
        Fs = sampling_frequency, scale='dB')
pl.title("Signal spectrum and LPC spectrum")
pl.legend(["LPC spectrum", "Magnitude spectrum"]);
```



Din figura anterioară putem să observăm faptul că spectrul LPC, asemeni spectrului lui h dat de coeficienții cepstrali este anvelopa spectrală a spectrului semnalului. Astfel că putem să concluzionăm faptul că acești coeficienți sunt o bună aproximare a filtrului dat de tractul vocal.

Exercițiu T8.2.1 Variați ordinul coeficienților LPC și observați modificarea spectrului LPC. ■

Exercițiu T8.2.2 Afișați spectrul LPC pentru consoană. ■

T8.2.1 Calculul formanților pe baza coeficienților LPC

Având mai bine evidențiată anvelopa spectrală a semnalului vocal și zonele de energie maximă locală, putem să determinăm formanții segmentelor sonore.

După cum am menționat anterior, **formanții** sunt frecvențele de rezonanță ale tractului vocal și sunt prezenți doar în cadrul segmentelor sonore. Formanții sunt un element important al analizei semnalului vocal și determină identitatea sunetului emis (vocala). Ca urmare, o primă formă de sinteză de voce, utilizată și în ziua de azi de către foneticieni, este **sinteza formantică**.[1]

[1] `https://ccrma.stanford.edu/~jos/pasp/Formant_Synthesis_Models.html`

Din spectrul LPC afișat anterior putem identifica punctele de energie spectrală maximă locală corespunzătoare formanților prin identificarea punctelor de inflexiune ale funcției matematice. Punctele de inflexiune sunt date de rădăcinile numărătorului funcției.

Secvența de cod următoare calculează aceste rădăcini complexe și le ordonează crescător în funcție de faza lor. Valorile în Hz a formanților fiind date de formula:

$$F = \frac{faza}{2\pi} * F_s \ [Hz]$$

În aplicații practice, se folosesc maxim primii 3-4 formanți. Vom limita și noi calculul lor la 4 valori:

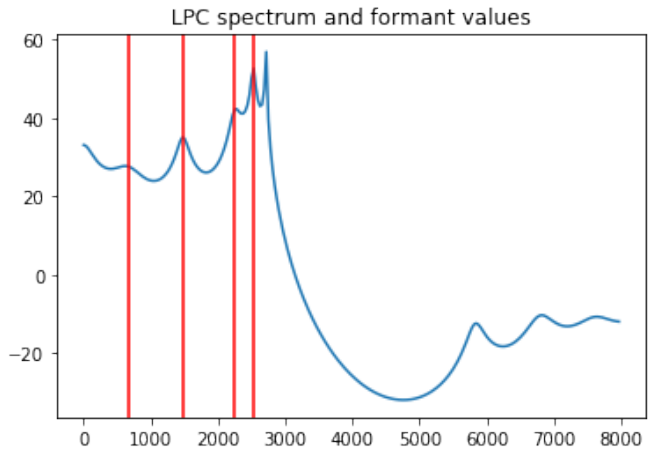
[4]:
```
def extract_formants(input_sample, lpc_order, fs):
    a = lb.lpc(input_sample,lpc_order)

    # Extragem rădăcinile polinomului dat de
    # coeficienții LPC
    roots = np.roots(a)
    # Rădăcinile sunt complex conjugate, reținem doar
    # o valoare din pereche
    roots = roots[np.where(np.imag(roots) > 0)]
    # Calculăm fazele rădăcinilor
    angles = np.arctan2(np.imag(roots), np.real(roots))
    # Calculăm frecvențele
    freqs = angles * (fs / (2 * np.pi))
    # Reordonăm în ordinea crescătoare a fazelor
    frequency_indices = np.argsort(freqs)
    formants = [int(x) for x in freqs[frequency_indices]]
    # Benzile de frecvență ale formanților
    # sunt date de distanța polilor față de
    # cercul unitate
    bw = -1 / 2 * (fs / (2 * np.pi))  \
            * np.log(np.abs(roots[frequency_indices]))
    # Frecvențele formanților trebuie să
    # fie mai mari decât 90Hz cu o bandă
    # de frecvențe mai mică de 400Hz
    formants = [f for i,f in enumerate(formants) \
                if f>90 and bw[i]<400]
    return formants[:4]
```

[5]:
```
# Extragem valorile formanților dintr-un cadru al vocalei
formants = extract_formants(hamming_frame, lpc_order, \
        sampling_frequency)
print ("Formant values: "+' '.join([str(x)+'Hz' \
        for x in formants]) )
```

[5]:
```
Formant values: 661Hz 1476Hz 2247Hz 2524Hz
```

Să afișăm aceste valori peste spectrul LPC:

[6]:
```
# Plot spectrul LPC
pl.plot(freq_axis, 20*np.log10(abs(h)))
# Plot valori formanți
for f in formants:
    pl.axvline(f, color = 'r')
pl.title("LPC spectrum and formant values");
```



Valorile formanților determinate anterior se suprapun perfect cu punctele de inflexiune ale spectrului LPC.

Exercițiu T8.2.3 Încercați să determinați formanții și pentru consoană. Ce obțineți? ■

T8.2.2 Eroarea de predicție

La începutul acestui tutorial am menționat faptul că eroarea de predicție este egală cu sursa de semnal. Să vedem cum arată această eroare de

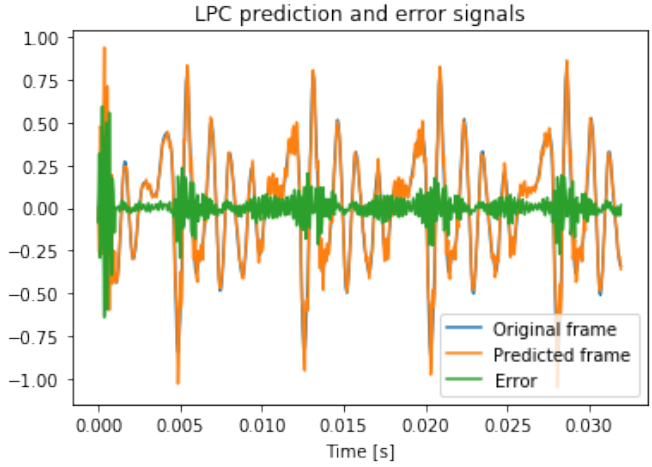
predicție. Vom filtra semnalul vocal cu filtrul invers dat de:

$$H(z) = \sum_{k=1}^{p} a_k z^{-k} \tag{T8.2.1}$$

pentru a obține semnalul prezis, iar mai apoi vom scădea din semnalul original semnalul prezis.

Trebuie să ținem cont de modul în care funcția `lpc()` ne returnează valorile coeficienților. Și anume, acestea includ termenul liber și semnul minus dinaintea sumei de la numitor:

[7]:
```
from scipy.signal import lfilter
# Creăm setul de coeficienți pentru filtrul invers
a_hat =  -1*a
a_hat[0] = 0
# Cadru din vocală
frame = vowel_frame
# Filtrăm cu filtrul invers
y_hat = lfilter(a_hat, 1, frame)
# Calculăm eroarea
err = frame - y_hat
# Plot
time_axis = np.arange(0, window_fft)*1.00/sampling_frequency
pl.plot(time_axis, frame)
pl.plot(time_axis, y_hat)
pl.plot(time_axis, err);
pl.title("LPC prediction and error signals")
pl.legend(["Original frame", "Predicted frame", "Error"]);
```



Observăm că semnalul prezis este foarte apropiat de semnalul original, eroarea fiind aproape zero de-a lungul acestui cadru de analiză.

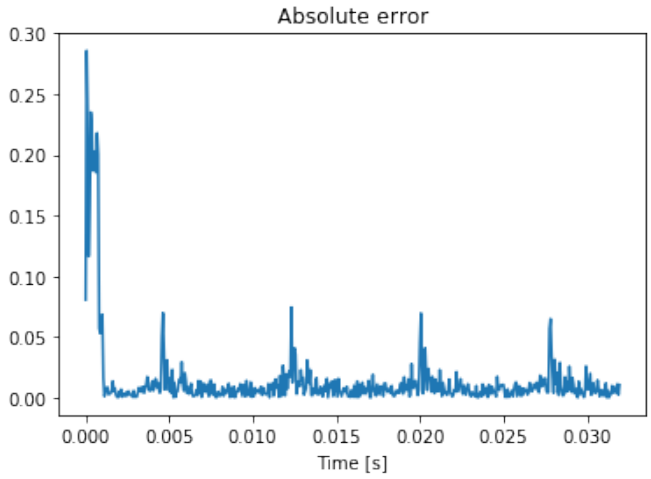
Exercițiu T8.2.4 Calculați eroarea de predicție și pentru un cadru al consoanei? Ce observați? ■

T8.3. Sinteza LPC

Până în momentul de față am reușit să extragem coeficienții LPC din semnal, să vizualizăm spectrul dat de acești coeficienți și să calculăm eroarea de predicție. Însă una dintre cele mai mari aplicații ale analizei LPC este cea de codare. Pe lângă extragerea coeficienților LPC este nevoie să se realizeze și sinteza semnalului vocal folosind cât mai puțini parametri transmiși sau stocați. Astfel că, dacă am reușit să utilizăm doar $Fs + 2$ coeficienți pentru a modela filtrul, trebuie să găsim și o modalitate de a reduce datele sursei.

Să vedem mai întâi ce informații putem regăsi în această eroare. Afișăm eroarea absolută însă pentru un cadru neponderat Hamming, pentru a fi mai evidentă informația:

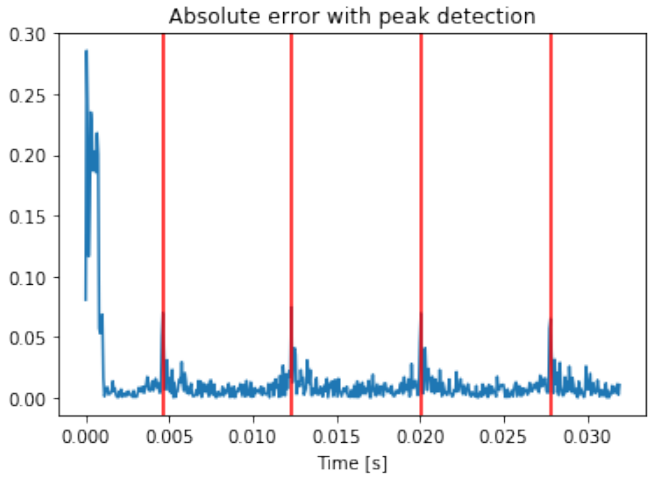
[8]:
```
# Cadru din vocală
frame = vowel_frame
a = lb.lpc(frame, lpc_order)
# Creăm setul de coeficienți pentru filtrul invers
a_hat =  -1*a
a_hat[0] = 0
# Filtrăm cu filtrul invers
y_hat = lfilter(a_hat, 1, frame)
# Calculăm eroarea
err = frame - y_hat
# Eroarea pătratică
err_square = abs(err)
pl.plot(time_axis, err_square);
pl.xlabel("Time [s]")
pl.title("Absolute error");
```



Se poate observa că pentru secvenţe sonore, în eroarea de predicţie apar maxime distanţate cu perioada fundamentală T0 (exceptând eroarea din primele câteva eșantioane). Să încercăm să extragem F_0 din eroarea de predicție:

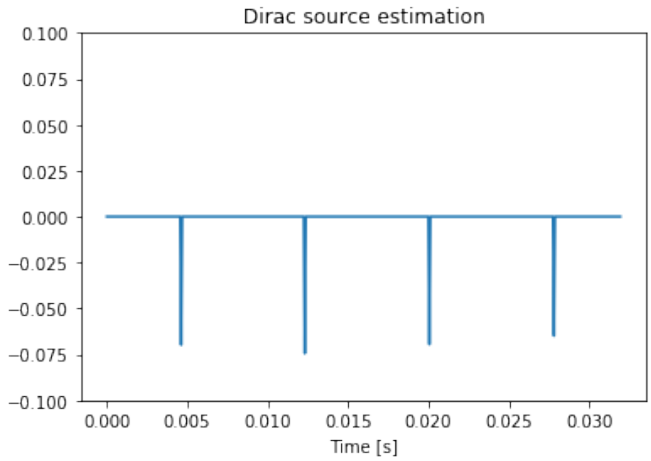
[9]:
```
from scipy.signal import find_peaks
# Reținem doar maximele distanțate cu minim
# 1/480Hz = T0 minim și ignorând eroare
# de început de cadru
peaks,_ = find_peaks(err_square[20:], distance=90)
# Corectăm indecșii returnați
peaks = peaks+20
# Calculăm distanța dintre indecșii returnați de funcție:
difs = [x-peaks[i-1] for i,x in enumerate(peaks)][1:]
# Determinăm media diferențelor
average_dist = np.mean(difs)
# Și o convertim în Hz
F0 = sampling_frequency/average_dist
# Afișăm
print ("F0 estimated from LPC error: %d Hz" %(int(F0)))
pl.plot(time_axis, err_square);
for z in peaks:
    pl.axvline(z*1.0/sampling_frequency, color = 'r')
pl.xlabel("Time [s]")
pl.title("Absolute error");
```

[9]:
```
F0 estimated from LPC error: 129 Hz
```

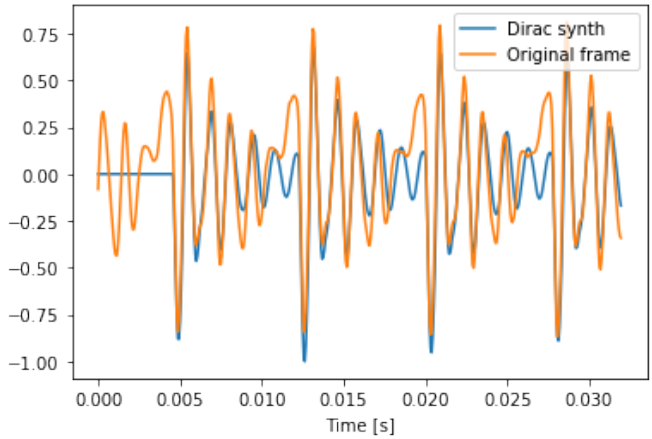


Dacă eroarea de predicție conține aceste maxime distanțate cu T_0, iar această eroare reprezintă sursa ideală de semnal pentru filtrul LPC, am putea să încercăm să modelăm această sursă cu impulsuri Dirac poziționate la indecșii maximelor și de amplitudine egală cu acestea:

[10]:
```
# Vector pentru sursa Dirac
dirac_source = np.zeros(window_fft)
dirac_source[peaks] = err[peaks]
pl.plot(time_axis, dirac_source)
pl.ylim([-0.10,0.10])
pl.xlabel("Time [s]")
pl.title("Dirac source estimation")
```



[11]:
```
# Filtrăm sursa Dirac cu filtrul LPC:
dirac_synth = lfilter([1.],a, dirac_source)
# Normalizăm pentru că nu am calculat
# câștigul filtrului LPC
dirac_synth = dirac_synth/(max(abs(dirac_synth)))
# Plot
time_axis = np.arange(0, window_fft)*1.00/sampling_frequency
pl.plot(time_axis, dirac_synth)
pl.plot(time_axis, vowel_frame)
pl.xlabel("Time [s]")
pl.legend(["Dirac synth", "Original frame"]);
```



[12]:
```
# Repetăm cadrul pentru a putea auzi rezultatul
dirac_synth_long = np.tile(dirac_synth, 3)
# Ascultăm sinteza
import IPython
IPython.display.Audio(dirac_synth_long,  \
        rate=sampling_frequency)
```

[12]:

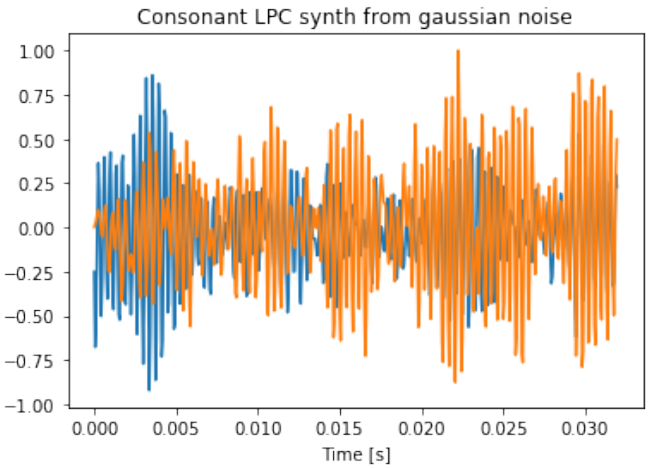

[13]:
```
# Ascultăm și semnalul original
IPython.display.Audio(wav_data_vowel,  \
        rate=sampling_frequency)
```

[13]:


Exercițiu T8.3.1 Reluați pașii de sinteză folosind altă vocală. Poate fi determinată identitatea vocalei din sinteza cu impulsuri Dirac? ■

Pentru consoane se folosește zgomot alb gaussian.

[14]:
```
# Cadrul consoanei
consonant_frame= wav_data_consonant[k*window_fft:  \
        (k+1)*window_fft]
noise = np.random.normal(scale=0.05* \
        np.max(consonant_frame), size=window_fft)
# Filtrare inversă consoană
a = lb.lpc(consonant_frame, lpc_order)
noise_synth = lfilter([1],a, noise)
noise_synth = noise_synth/max(abs(noise_synth))
noise_synth_long = np.tile(noise_synth, 10)
pl.plot(time_axis,consonant_frame)
pl.plot(time_axis, noise_synth)
pl.xlabel("Time [s]")
pl.title("Consonant LPC synth from gaussian noise");
```



[15]:
```
IPython.display.Audio(noise_synth, rate=sampling_frequency_c)
```

[15]:

[16]:
```
IPython.display.Audio(wav_data_consonant, \
        rate=sampling_frequency_c)
```

[16]:


Exerciţiu T8.3.2 Încercați să realizați sinteza LPC pe un întreg semnal vocal făcând distincția automat între consoane și vocale pe baza erorii de predicție. ■

În mod evident, simplificând atât de mult sursa de semnal va rezulta într-o degradare majoră a calității semnalului sintetizat. Pentru a evita acest lucru se folosesc combinații de impulsuri și zgomot cu ponderi variabile atât pentru consoane, cât și pentru vocale.

T8.4. Concluzii

În cadrul acestui tutorial am introdus analiza prin predicție liniară. Această analiză permite separarea sursei de filtru din modelul liniar-separabil de producere a vorbirii. Am văzut totodată și modul în care putem calcul formanții segmentelor sonore pornind de la spectrul dat de coeficienții LPC, precum și modul în care putem realiza sinteza LPC minimizând informația din sursa de semnal. De altfel, una dintre cele mai importante aplicații ale analizei prin predicție liniară este cea de codare. Metoda de codare din GSM - o variantă a Code Excited Linear Prediction - utilizează acest tip de analiză.

BIBLIOGRAFIE SUPLIMENTARĂ

- Wai C. Chu, "Speech Coding Algorithms", Wiley&Sons, 2003
- Paul Taylor, ”Text to speech synthesis”, Cambridge University Press, 2009
- Benesty et al, “Springer Handbook of Speech Processing”, Springer, 2008

RESURSE MEDIA

- ETSI GSM Standard 2g - online: `https://www.etsi.org/technologies/mobile/2g`
- IRCAM, "AudioSculpt 3.0 - user manual" - online `http://support.ircam.fr/docs/AudioSculpt/3.0/co/LPC.html`
- Coursera, "Internet of Things: Communication Technologies Course", online: `https://www.coursera.org/lecture/internet-of-things-communication/linear-predictive-coding-of-speech-LZ1OC`


Speech
processing
GROUP