ALMA MATER STUDIORUM - UNIVERSITÀ DI BOLOGNA

SCUOLA DI INGEGNERIA E ARCHITETTURA

DEI-ARCES

INGEGNERIA ELETTRONICA

TESI DI LAUREA

in

Elaborazione statistica dei segnali nei sistemi elettronici

SVILUPPO DI UN'APPLICAZIONE SMARTPHONE PER L'ELABORAZIONE E LA SELEZIONE STEP-BASED DI BRANI MUSICALI

Viviana Liuni

RELATORE:
Chiar.mo Prof. Riccardo Rovatti

CORRELATORE/CORRELATORI
Valerio Cambareri e Carlos Formigli

Anno Accademico 2012/13

Sessione II

INDICE

1. Introduzione

Al giorno d'oggi è inevitabile considerare quanto la maggior parte del pubblico di consumatori e utilizzatori di sistemi elettronici sia in costante contatto con dispositivi smartphone e tablet.

La Quinto Monitor sul Mobile Marketing, agenzia leader nel mondo del mobile marketing e delle digital communication solutions, ha cercato di delineare i trend circa l'utilizzo dei dispositivi mobili da parte degli Italiani. La ricerca che hanno condotto su un campione di oltre 1000 intervistati tra i 15 e i 65 anni, sottolinea il forte incremento dell'uso di smartphone e tablet, utilizzati dal 62% degli Italiani poiché sempre più potenti e "user-friendly".

Compito dell'ingegnere è, quindi, quello di riuscire ad usufruire di queste nuove tecnologie per poter arrivare anche all'utente comune, proponendo soluzioni software che sfruttino le conoscenze tecniche apprese negli anni di studi, in questo caso, quello della statistica dei segnali e dei sistemi elettronici.

Per quanto il mondo degli smartphone stia diventando sempre più ampio e si stia diffondendo fra tutte le aziende produttrici di cellulari o di computer, due sono i sistemi operativi maggiormente implementati in questi ultimi tre anni, iOs e Android. Per quanto possa risultare interessante creare un'applicazione multi-piattaforma, in questa tesi si tratterà solo dello sviluppo su iPhone iOs, in quanto il linguaggio è poco conosciuto e ha dato luogo a una ricerca e sviluppo più stimolante.

1.1 Obiettivo del Progetto

Per seguire la specifica di cui si è trattato si è pensata e voluta costruire un'applicazione che serva a motivare tutti gli utenti che utilizzano tale smartphone durante le sessioni di corsa. Con la caratteristica particolare che attraverso appositi comandi l'utente possa scegliere i brani che desidera inserire nella propria applicazione, da noi chiamata Music Runner, questa fa in modo che essi possano essere elaborati da un algoritmo di beat detection che ne identifichi i battiti per minuto.

Per fare ciò l'utente accede alla propria libreria iPod (ovvero l'elenco di file mp3 che sono stati importati dal computer), e l'applicazione li elabora compilando una lista, che chiameremo libreria di bpm, che associa ad ogni brano il proprio bpm.

Quando la libreria di bpm è stata riempita con le scelte dell'utente, questo può utilizzare Music Runner per le proprie sessioni di corsa.

La schermata principale dell'applicazione contiene un pulsante "play"; quando l'utente lo preme Music Runner inizierà a campionare il segnale dall'accelerometro e a riprodurre un primo brano musicale. Appena il brano finisce oppure quando l'utente decide di cambiarlo, l'applicazione calcolerà la cadenza della corsa o della camminata dell'utente ed andrà ad eseguire dalla libreria precedentemente costituita la canzone con il bpm più indicato al ritmo della corsa.

Possiamo quindi dividere lo sviluppo di tale progetto in due macro-step che identificano i due momenti separati che contraddistinguono Music Runner: l'algoritmo di beat detection e l'algoritmo di step detection. Nel corso della trattazione verranno spiegati tali algoritmi, e citate le fonti da cui sono stati tratti; inoltre avremo premura di informare di come è stato pensato il codice e dei framework importati in modo che potessero funzionare su smartphone iOs. Vale la pena anticipare, che mentre in un software avanzato come Matlab certe operazioni di statistica dei segnali, come una autocorrelazione, siano fattibili con una sola funzione, su iOs la maggior parte delle volte tutto deve essere di nuovo implementato o opportunamente inizializzato.

I capitoli a seguire parleranno quindi prima dei due algoritmi, poi della struttura dell'applicazione e del suo sviluppo, infine dei risultati ottenuti in termini di accuratezza e velocità degli algoritmi.

2. Gli Algoritmi

2.1 Estrazione del ritmo (Beat Detection)

L'estrazione automatica del ritmo da una porzione di un brano musicale è stato un tema alquanto dibattuto nel campo della ricerca. Attraverso una serie di filtri combinati, i creatori dell'algoritmo che andremo a descrivere sono riusciti a ricreare tale estrattore in modo da funzionare come l'orecchio umano(*Tempo and beat analysis of acoustic musical signals [2]*), sebbene sia adatto soprattutto su brani con un accentuato "beat" della batteria, mentre per quelli di musica più strumentale, come quella classica, il risultato non è sempre attendibile. Il nostro scopo però, e quello di motivare l'utente durante la corsa con la musica che selezioniamo in automatico, quindi le limitazioni del codice non pregiudicano il risultato finale.

Per capire come questo algoritmo sia stato elaborato, bisogna un attimo fare delle ipotesi sui segnali che andremo ad elaborare, poiché ne bisogna estrarre il ritmo senza far affidamento a degli impulsi facilmente identificabili, cosa che ad esempio non è vera quando si analizza il segnale della corsa di un soggetto preso da un accelerometro, che consiste principalmente in tante oscillazioni quasi periodiche.

2.1.1 Semplificazioni psicoacustiche

Una delle maggiori difficoltà del modello trascrittivo della percezione del ritmo è la complessità di raggruppare le armoniche parziali con le corrispondenti note, e determinare i tempi di inizio di quest'ultime. Anche se vengono fatte ipotesi semplificative riguardo il tono e il contenuto timbrico, non è un compito facile individuare i tempi di battere e levare.

Tuttavia, sembra che da una dimostrazione psicoacustica sulla percezione del beat certi tipi di manipolazioni del segnale e semplificazioni possano essere eseguiti senza incidere sul contenuto impulsivo di un segnale musicale.

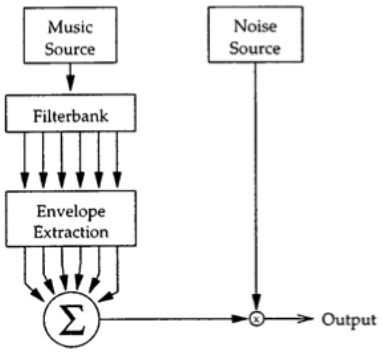


Fig. 2.1.1: Segnale rumoroso che non ha le stesse caratteristiche ritmiche dell'input musicale, indicato come somma degli inviluppi in ampiezza.

Si consideri il diagramma a blocchi in figura 2.1.1.
Si costruisce del “rumore modulato in ampiezza” come un segnale ottenuto col vocoding di uno a rumore bianco con uno a inviluppi di sottobande del segnale musicale (dove il vocoder è un dispositivo elettronico o un software in grado di codificare un qualsiasi segnale audio attraverso i parametri di un modello matematico). Questo viene realizzato eseguendo un’analisi frequenziale del brano (processato attraverso un banco di filtri passabanda) e anche con del rumore bianco uscente da un generatore pseudo-random. L’ampiezza di ciascuna banda del segnale di rumore è modulata con l’inviluppo della corrispondente banda dell’output musicale ottenuto dal banco di filtri e i risultati rumorosi sono sommati insieme.

Per molti tipi di banchi di filtri frequenziali il rumore risultante ha una percezione ritmica che è significativamente la stessa del segnale originario. Anche se ci sono poche bande il battito e le caratteristiche metriche del segnale iniziale sono immediatamente riconoscibili.
Dal momento che l’unica informazione conservata è l’inviluppo, è chiaro che questo è sufficiente per estrarre il ritmo da un segnale musicale, quindi le note non sono una componente necessaria ai fini della percezione ritmica. Questo permette di ridurre la dimensione dei dati d’ingresso.

Altri tipi di semplificazioni non sono possibili. Per esempio, se viene usata una sola banda, o in maniera equivalente le sotto bande sono linearmente combinate prima di modularle con il rumore, un uditore non può più percepire il contenuto ritmico della maggior parte dei segnali.

Perciò si può dedurre che separare il segnale in sottobande e mantenerne gli inviluppi divisi sia necessario per processare accuratamente il ritmo. Per dirla in altro modo, l’algoritmo in figura 2.1.1 è un metodo per generare nuovi segnali la cui rappresentazione attraverso la somma di inviluppi usciti da un banco di filtri è la stessa dell’estratto da un brano musicale. Tuttavia, dal momento che questi nuovi segnali spesso non hanno una equivalenza percettiva con gli originali, questa struttura è inadeguata per rappresentare dati di un segnale musicale che sono importanti per la comprensione ritmica.

Questa considerazione porta ad un’ipotesi psicoacustica riguardo la percezione del ritmo: il sistema uditivo realizza una sorta di integrazione ritmica a bande incrociate, e non una semplice sommatoria di bande in frequenza.

Per i nostri scopi è importante considerare un algoritmo di elaborazione ritmica che tratti separatamente le bande frequenziali e alla fine combini i risultati, anziché tentare di eseguire il beat-tracking sulla somma delle uscite.

2.1.2 Descrizione dell'algoritmo

I ricercatori che hanno ideato l'algoritmo di beat-tracking, ovvero di rilevamento del beat, che verrà qui di seguito presentato, si sono ispirati a lavoro di Large e Kolen, prendendone largamente spunto, in quanto utilizza una rete di risonatori per bloccare la fase con il beat del segnale e determinare la frequenza dell'impulso. Tuttavia, il metodo qui presentato è alquanto diverso. I risonatori sono analiticamente molto più semplici. Vengono usati banchi di risonatori e fatto il pre e post processing del segnale per estrarre accuratamente le informazioni desiderate, poiché questo modello opera su dati acustici anziché su uno stream di eventi.

Un impulso ritmico è descritto da una frequenza e da una componente di fase; la frequenza dell'impulso in un segnale musicale ritmico è il tempo o la velocità del ritmo, e la sua fase indica dove si verifica il "battere" del brano. Cioè, le volte in cui si verifica un impulso si può dire che siano a fase zero, e quindi i punti nel tempo esattamente in mezzo agli impulsi hanno fase a π radianti. E' importante notare che mentre il riconoscimento umano del "battere" è sensibile alla fase solo sotto certe condizioni particolari, la risposta ritmica è specialmente un fenomeno in fase.

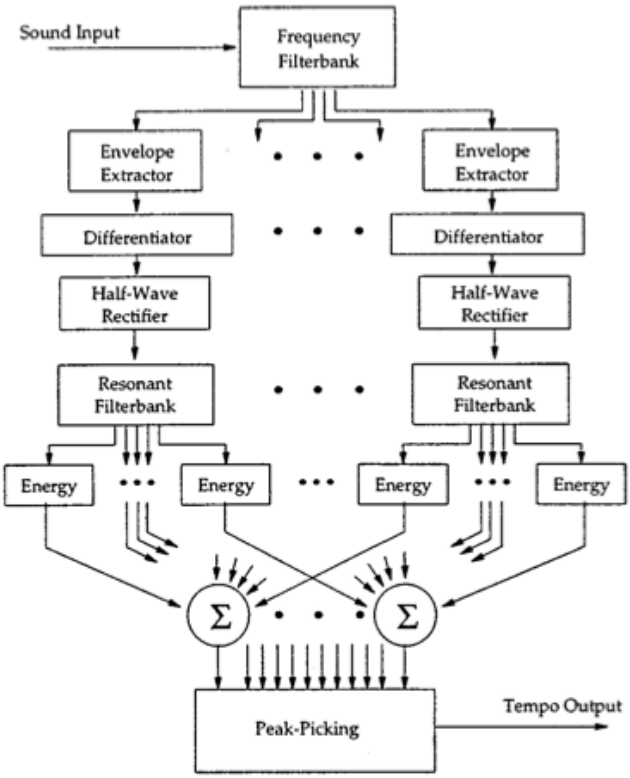


Fig. 2.1.2: Schema a blocchi dell'algoritmo

La figura 2.1.2 mostra una visione d'insieme dell'algoritmo di tempo-analisi come una rete di flusso del segnale.

Quando il segnale entra vengono usati dei banchi di filtri per dividerlo in sei bande. Per ciascuna di queste sottobande ne si calcola l'inviluppo dell'ampiezza e lo si deriva. Ciascuno degli inviluppi derivati passa in un altro banco di filtri, questa volta a pettine, per sfruttarne le caratteristiche di ritardare il segnale. A questo punto viene salvata l'energia contenuta in ciascun segnale di output dentro un vettore; alla fine il valore a maggior energia viene utilizzato per ricavarne il bpm corrispondente.

2.1.3 Analisi in frequenza ed estrazione del segnale

Come già discusso, gli inviluppi estratti da un piccolo numero di canali di ampie frequenze sono informazioni sufficienti per analizzare ritmicamente un segnale musicale, almeno per ascoltatori umani. Inoltre, studi empirici sull'uso di vari banchi di filtri con questo algoritmo hanno dimostrato che non è particolarmente sensibile a bande specifiche. Partendo dal segnale in figura 2.1.3, andremo a illustrare i passi che costituiscono l'algoritmo.

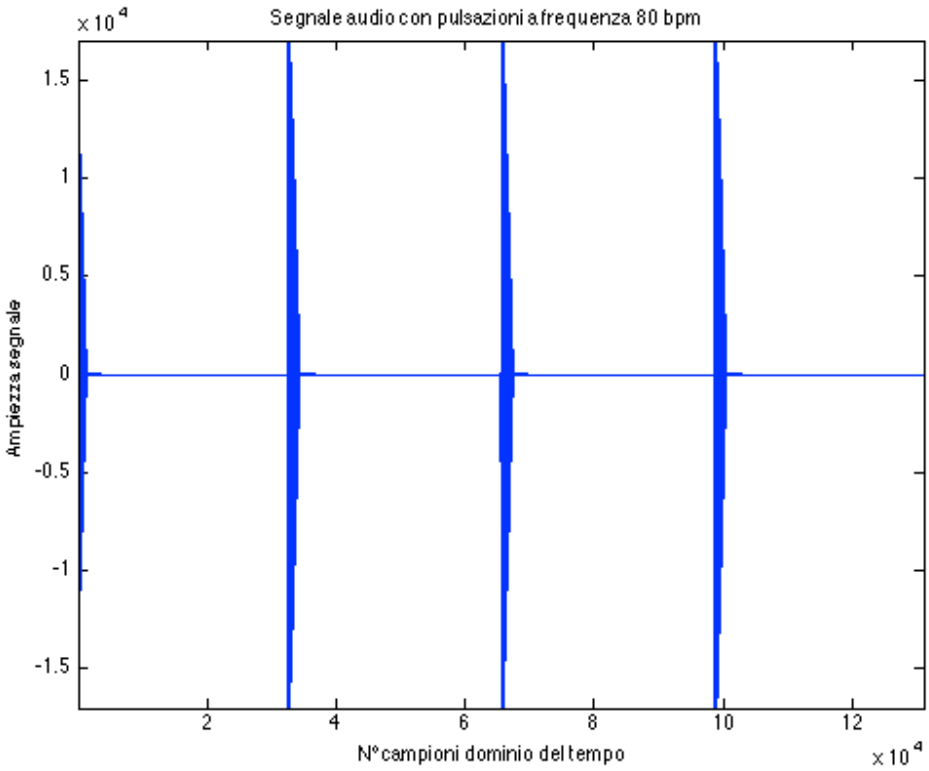


Fig. 2.1.3: Estratto di un brano musicale di soli impulsi, a frequenza 80bpm.

2.1.4 Banco di filtri (Filterbank)

Considerando la frequenza di campionamento, estraiamo dal brano musicale un vettore di 44100 campioni dal centro del segnale originario, poi lo dividiamo in sei segnali separati, in modo che ciascuno abbia il contenuto in frequenza di un certo range. Questo ha l'effetto di separare i suoni da diversi gruppi di strumenti musicali e analizzarli separatamente. Tale scissione avviene attraverso la FFT del segnale che poi viene divisa opportunamente in parti assegnate alle loro bande di frequenza; le diverse bande sono: 0-1000Hz , 1000-2000 Hz, 2000-4000Hz, 4000-8000Hz, 8000-16000Hz, 16000-22050 Hz. Successivamente vengono antitrasformate tutte e sei le bande e mandate alla funzione di smoothing.

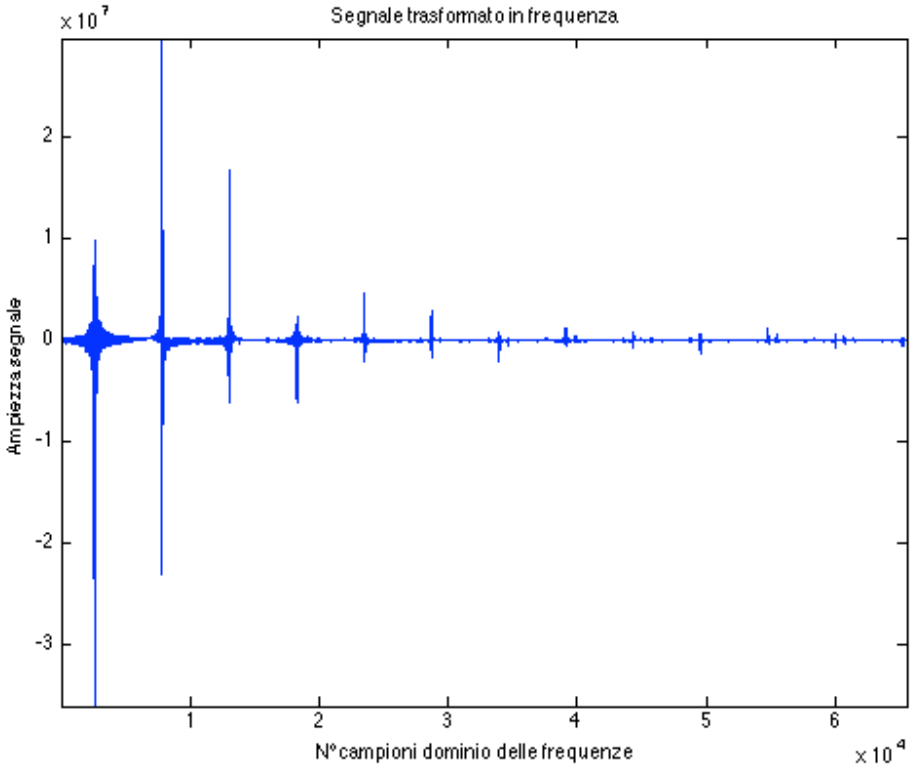


Fig. 2.1.4: Segnale filtrato prima di essere suddiviso in sei bande distinte.

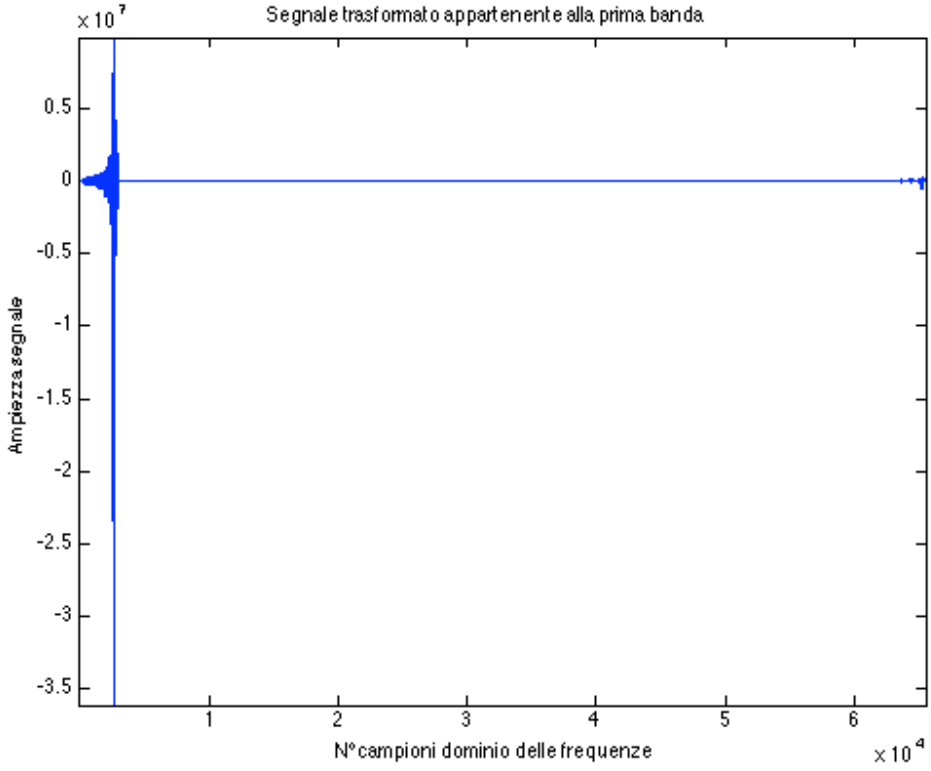


Fig. 2.1.5: Uscita dal banco di filtri appartenente alla prima delle sei bande.

2.1.5 Estrazione di Inviluppo (Smoothing)

Poiché dobbiamo identificare il ritmo del nostro segnale, bisogna ridurlo a una forma in cui è possibile vedere i cambiamenti improvvisi nel suono. Questo è fatto riducendo il segnale al suo inviluppo, che può essere pensato come il trend generale dell'ampiezza del suono. In sostanza, vengono prese ciascuna delle sei bande in frequenza e filtrate con un passa-basso. Per fare questo si è prima fatto il valore assoluto del segnale nel dominio del tempo così da diminuire il contenuto in alta frequenza e sfruttare solo la parte positiva dell'inviluppo . Si è poi fatta la convoluzione di ciascun segnale con la metà destra di una finestra di Hanning della lunghezza di 0.4 secondi. I grafici risultanti dai segnali in frequenza corrispondono agli esatti inviluppi dei segnali originali.

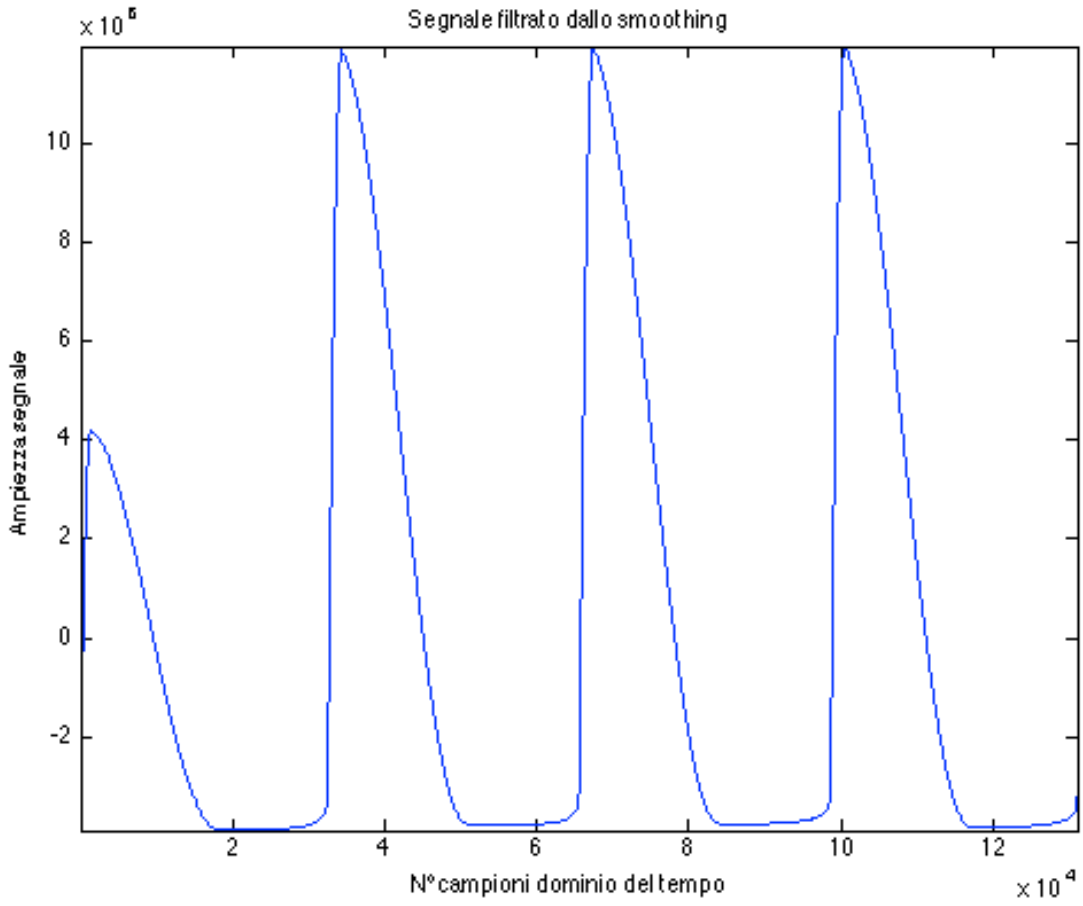


Fig. 2.1.6: Segnale filtrato e rettificato, appartenente alla prima banda.

2.1.6 Differenziazione e Rettificazione

Ora che abbiamo i sei segnali in forma di inviluppo, basta semplicemente differenziarli per accentuare quando ci sono le variazioni di ampiezza del suono. I più grandi cambiamenti dovrebbero corrispondere ai battiti, in quanto il ritmo è solo un accento periodico del suono. I sei segnali vengono differenziati nel tempo e poi rettificati così che se ne possa vedere solo l'aumento del livello sonoro. Questi segnali possono essere così analizzati ritmicamente.

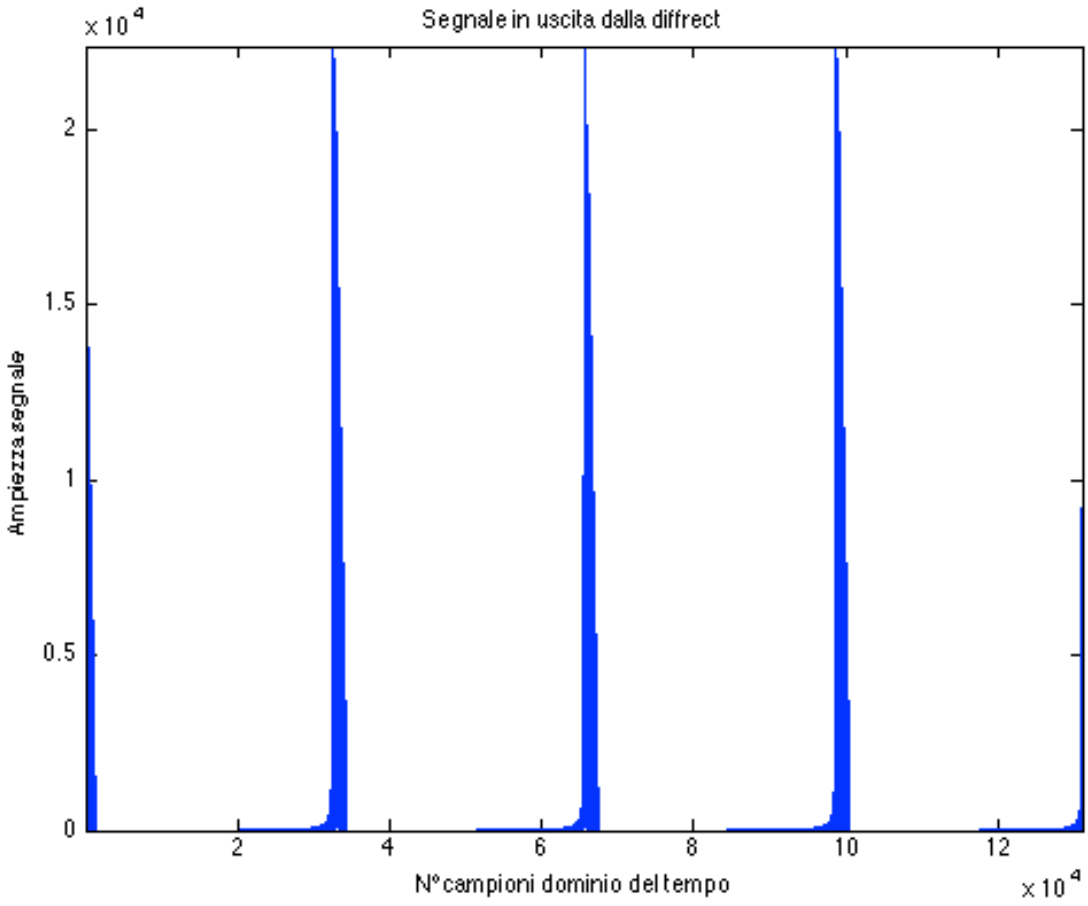


Fig. 2.1.7: Segnale differenziato e rettificato, appartenente alla prima banda.

2.1.7 Filtri a Pettine

Questo è il passaggio più intenso computazionalmente. Bisogna fare la convoluzione dei sei segnali con vari filtri a pettine per determinare chi produce più energia. Un filtro a pettine ha la sua risposta all'impulso formata da una serie di impulsi a intervalli regolari, proprio come se fossero degli echi dell'impulso in input. Abbiamo scelto per questi filtri una risposta temporale all'impulso che consiste in tre impulsi unitari, in modo da catturare almeno due periodi da ogni possibile segnale di ritmo. Convolvendo ogni filtro a pettine con il nostro segnale dovrebbe dare luogo a un output che ha una più alta energia quando la frequenza caratteristica di quel filtro è vicina a un multiplo del ritmo della canzone. Questo perché la convoluzione con il filtro a pettine produce un vettore di output formato da una versione con eco del segnale originale, che avrà dei picchi maggiori e maggior energia se il ritmo del segnale e il filtro a pettine corrispondono.

Ogni filtro a pettine è stato implementato specificando gli intervalli di ritmo che ci interessavano scorrere e la spaziatura tra essi. Questo ha creato una serie di filtri a pettine che vengono trasformati al dominio delle frequenze e moltiplicati per la FFT del segnale in ogni banda, e di questo prodotto ne è stata calcolata l'energia. Infine tutte queste energie sono state inserite additivamente in un vettore, dentro il quale ogni elemento corrisponde a un filtro a pettine ed al suo BPM associato. Così, in questo vettore si spera di avere un picco corrispondente all'energia aggiunta dalle sei bande, al ritmo fondamentale del brano ed altri picchi multipli di quest'ultimo. Comunque, anche se i picchi non fossero così distribuiti, si prendere come bpm risultate quello a massima energia.

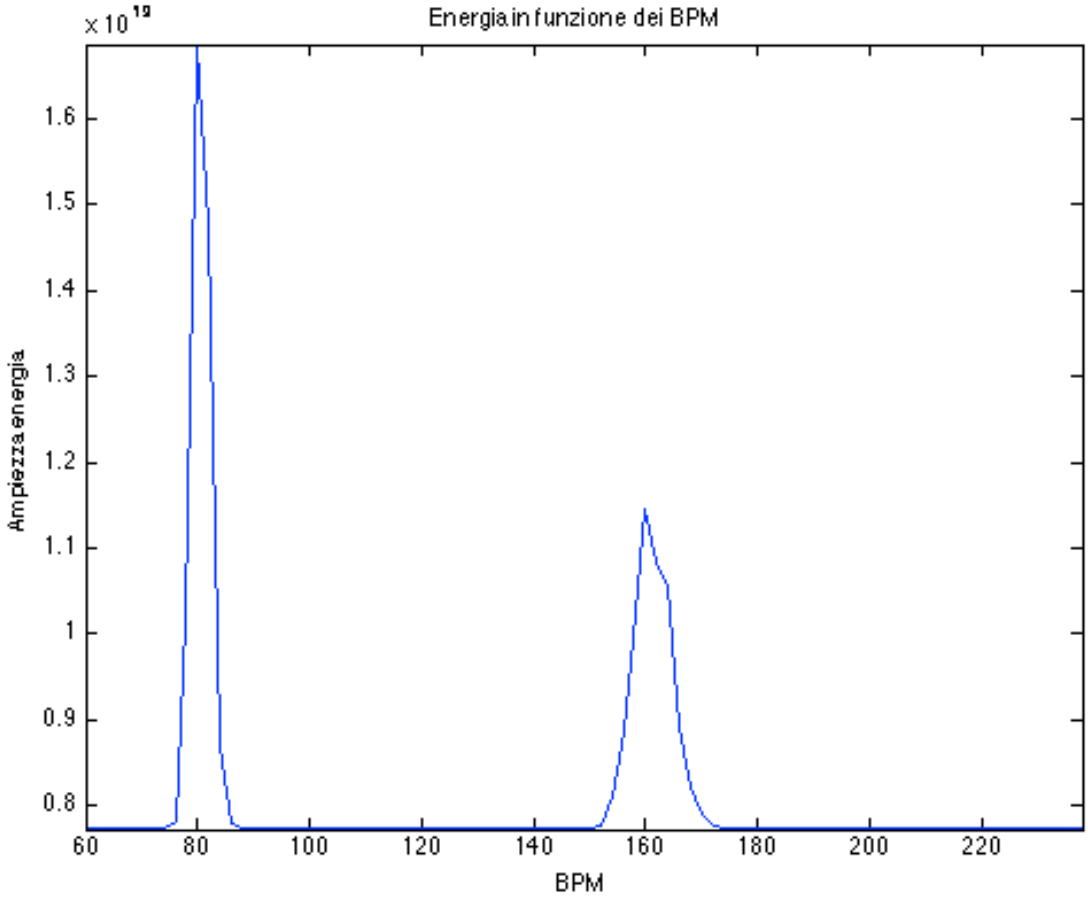


Fig. 2.1.8: Distribuzione dell'energia del segnale in una scala rappresentata dai bpm

Come si può vedere nella figura 2.1.8, il risultato è conforme a ciò che ci aspettavamo, ovvero una grande risposta energetica intorno agli 80bpm, e una meno accentuata intorno a un suo multiplo, 160bpm. Nei capitoli finali di questa trattazione (4.5) approfondiremo meglio l'implementazione di tale algoritmo e le sue prestazioni.

2.2 Estrazione della cadenza (Step Detection)

Per questa parte del progetto si è scelto un nuovo algoritmo principalmente perché il tipo di segnale che si deve analizzare è molto più semplice del caso musicale, e si è potuto evitare di usare un metodo che richieda un carico computazionale molto elevato. Mentre ad esempio il calcolo dei bpm può essere fatto in un momento in cui l'utente può attendere i circa 10s che servono per analizzare il brano, durante una sessione di corsa è impensabile che l'elaborazione duri più di qualche millisecondo, quindi si è preferito scrivere codice con un algoritmo più semplice, ma del tutto efficace allo scopo di analizzare la periodicità del passo, piuttosto che sottoporre il segnale a delle operazioni superflue, come la divisione in bande.

Lo studio fatto su tale algoritmo riguarda nuovi usi dei sistemi portabili per misurare i parametri del gait-cycle. Le misurazioni sono state fatte con un accelerometro triassiale posto sulla parte inferiore del tronco durante la fase di cammino del soggetto preso in esame.

I segnali presi da ciascun trial sono stati trasformati in un sistema di coordinate orizzontali-verticali e analizzate con un'autocorrelazione *unbiased* per ottenere la cadenza, lunghezza, regolarità e simmetria del passo. Ai fini della nostra applicazione abbiamo estratto solo il parametro della cadenza, ma ciò non toglie che per sviluppi futuri si possano sfruttare anche gli altri per aiutare l'utente nel proprio allenamento fornendogli un vasto range di dati.

Negli ultimi anni, gli accelerometri piezoresistivi a bassa inerzia hanno avuto un calo dei costi talmente rapido, che sono stati assunti dai laboratori biomeccanici come strumenti di studio di casi a budget limitato. Con l'avvento degli smartphone, in particolare quelli contenenti accelerometri, giroscopi e magnetometri, l'attenzione si è rivolta anche a questi nuovi strumenti, ad oggi alla portata di un pubblico sempre più numeroso.

Esattamente come in questa tesi ne viene sfruttata la tecnologia integrata per studiare i dati della corsa di un soggetto che si allena, tali algoritmi potrebbero essere impiegati anche in campo medico per lo studio della regolarità del passo, e in particolare nei casi di ricerca, come ad esempio quelli sul morbo di Parkinson.

L'algoritmo che è stato implementato su uno smarphone iOs utilizza come strumento principale l'autocorrelazione del segnale campionato dall'accelerometro. Come suggerisce la trattazione scientifica cui è ispirato (*Estimation of gait cycle characteristics by trunk accelerometry [1]*), inizialmente si è studiato il segnale proveniente dall'accelerometro verticale di un dispositivo iPhone posto sul tronco.
Dall'osservazione dei soggetti durante la corsa che fanno uso di iPod, si è notato che il trend più comune ricade sull'uso di fascia da braccio: è bastato fare il modulo dell'accelerazione per rendere l'algoritmo indipendente dall'orientamento del dispositivo.

L'uso dell'autocorrelazione ha un senso in quanto è la correlazione incrociata di un segnale con se stesso ed è utile per cercare in un segnale pattern che si ripetono, così da determinare la presenza di un segnale periodico che è stato affetto da troppo rumore, o identificare la frequenza fondamentale di un segnale che non contiene originariamente la componente di frequenza del rumore, bensì varie frequenze armoniche.

Il segnale che vorremmo idealmente studiare dall'accelerometro, infatti, ha un aspetto di questo tipo:

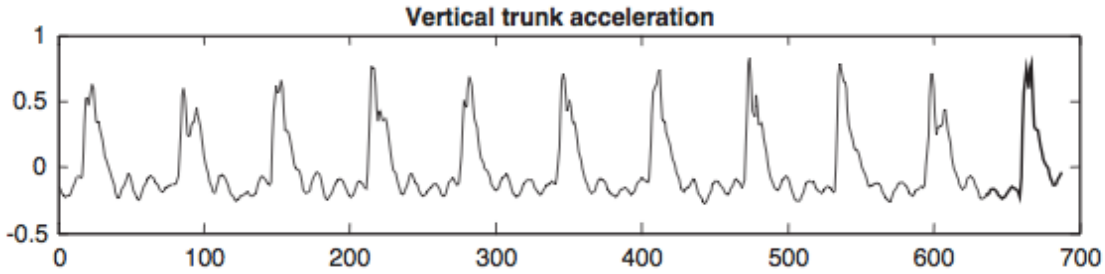


Fig. 2.2.1: Segnale poco rumoroso dell'accelerazione verticale durante il cammino

Del quale sarebbe alquanto semplice identificarne la cadenza, calcolando la distanza fra i picchi. Purtroppo il segnale che proviene dal nostro accelerometro ha caratteristiche come di seguito in figura 2.2.2.

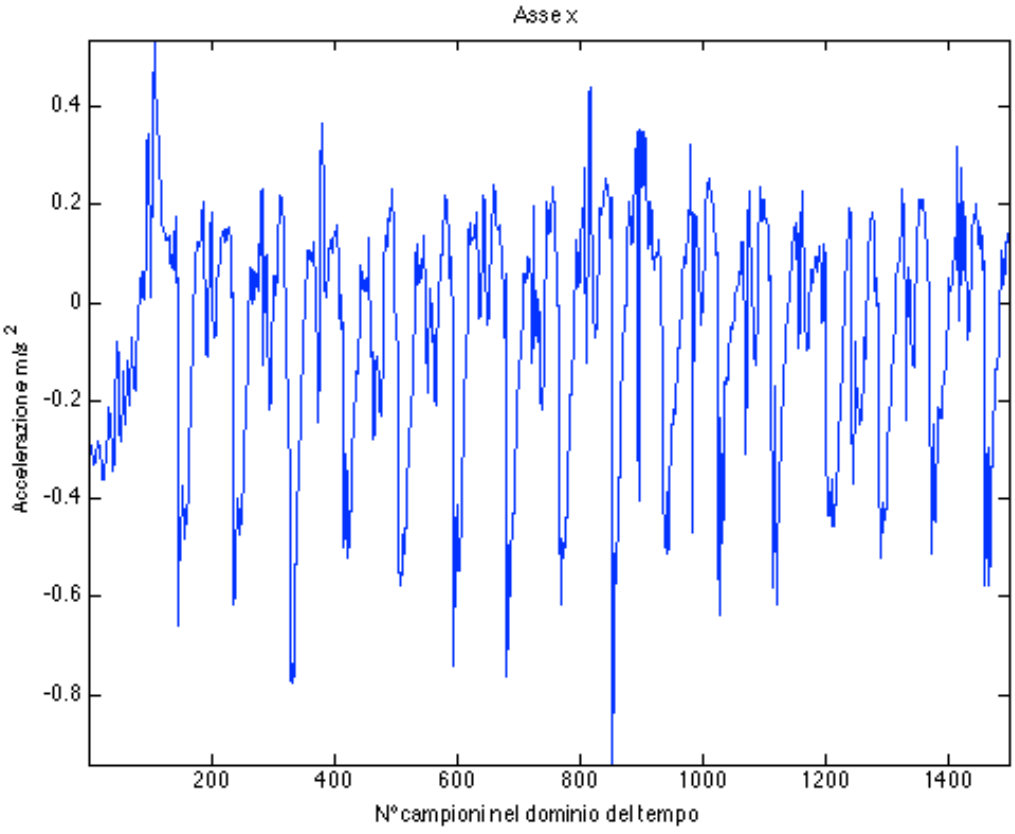


Fig. 2.2.2: Accelerazione sull'asse x durante una sessione di un minuto di corsa

Per il quale è difficile definire in automatico da uno smartphone, durante una sessione di corsa, quale sia la frequenza del passo (anche se ad occhio è abbastanza immediato, ma i picchi sono diversi da soggetto a soggetto e non in tutti i casi sarebbero validi gli stessi parametri per calcolare la distanza tra essi).

Dopo aver filtrato il segnale, facendone un'autocorrelazione *unbiased*, invece, è possibile quantificare i valori di picco del primo e secondo periodo dominante, che indicano lo sfasamento rispettivamente tra lo step e lo stride (cioè tra quando il piede tocca terra e quando viene sollevato).

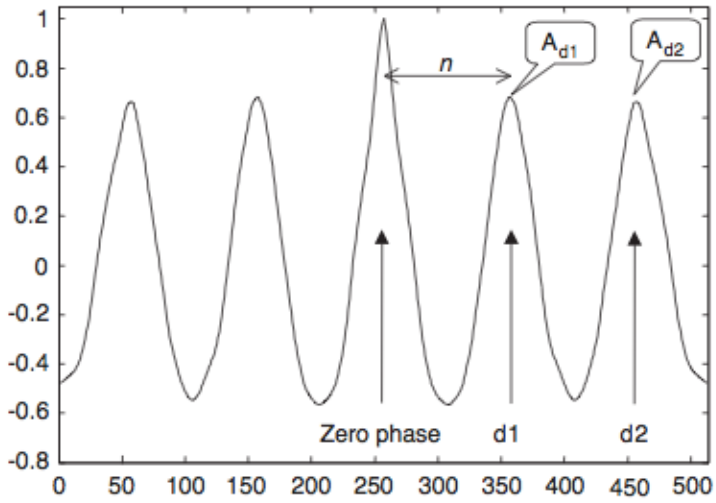


Fig. 2.2.3: Autocorrelazione bilaterale dell'accelerazione di un accelerometro posto sul tronco di un soggetto che cammina. Si indica con n il numero di campioni per periodo dominante, con d1 il primo picco e d2 il secondo picco.

La cadenza, intesa come step per minuto, può essere quindi calcolata se sono noti il tempo della camminata, la frequenza di campionamento e il numero di campioni per periodo dominante.
Se indichiamo con *n* il numero di campioni tra il secondo e il primo periodo dominante, *S* il tempo in secondi, *f* la frequenza di campionamento, *N* il numero di campioni e *Q* il numero di passi, possiamo così trovare la cadenza *c*:

$$N = Sf;$$
$$Q = N/n;$$
$$c = 60Q/S;$$

Infine:

$$c = 60(N/n)/(N/f) = 60\, f/n;$$

Abbiamo così ricavato la cadenza del passo sia nel caso di cammino, che in quello di corsa, dove il segnale è più accentuato e l'algoritmo riesce meglio a calcolare l'autocorrelazione fra i passi.

3. Lo Sviluppo su smartphone iOs

3.1 Architettura smartphone iOs

iOS è il sistema operativo sviluppato da Apple.Inc per iPhone, iPod Touch e iPad.

Come Mac OS X è una derivazione di FreeBSD, usa un kernel Mach e Darwin. iOS ha quattro livelli di astrazione:

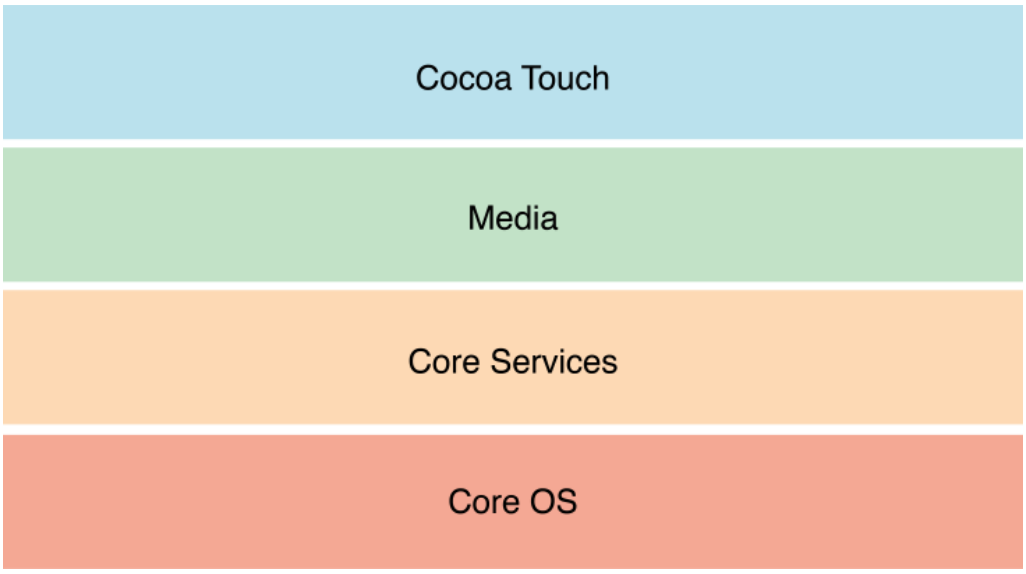


Fig. 3.1.1: Schema a blocchi dell'architettura iOs

Core OS

Questo livello contiene il kernel, il file system, l'infrastruttura network, di sicurezza e gestione della batteria e diversi driver. Contiene inoltre la libreria libSystem composta da elementi per:

-Threading (POSIX threads)
-Networking (BDS Sockets)
-File system access
-Standard I/O
-Bonjour and DNS Services
-Locale Information
-Memory Allocation
-Math Computations (di cui useremo il framework *Accelerate*)

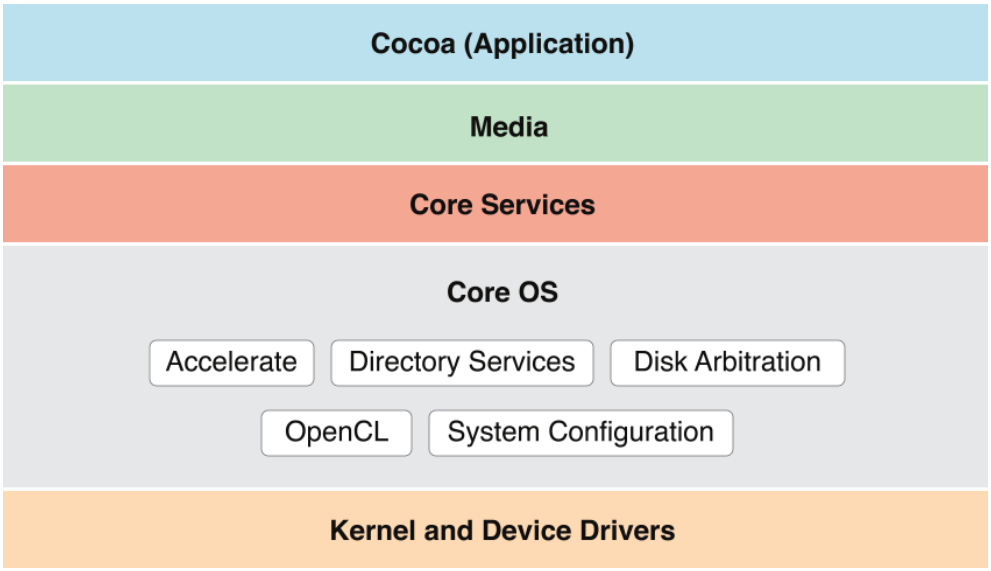


Fig. 3.1.2: Alcuni esempi di quali framework sono contenuti dentro Core OS

Core Services

I frameworks in questo livello servono per i principali servizi di sistema che l'applicazione usa. Questo livello include:

-Core Foundation framework
-CFNetwork framework
-Security framework
-SQLite library (di cui useremo *Core Data*)
-XML libraries

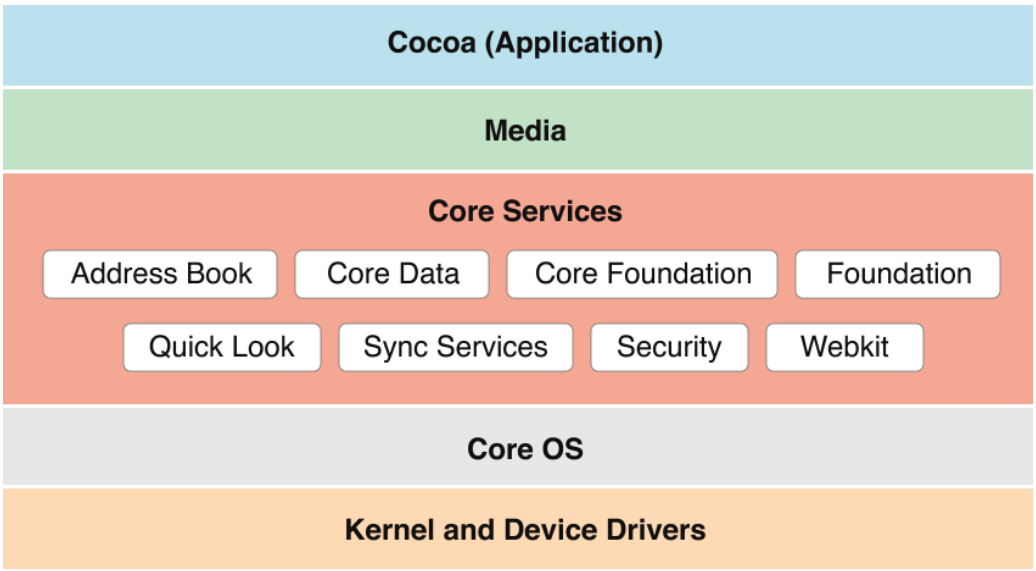


Fig. 3.1.3: Alcuni esempi di quali framework sono contenuti dentro Core Services

Media

Livello con tecnologie e frameworks per la gestione di file grafici e multimediali che dipendono dal livello precedente. Queste sono:

Per file Grafici:
-Quartz
-Core Animation
-OpenGL ES

Per file Audio:
-Core Audio & Audio ToolBox frameworks
-OpenAL

Per file Video :
-MediaPlayer framework

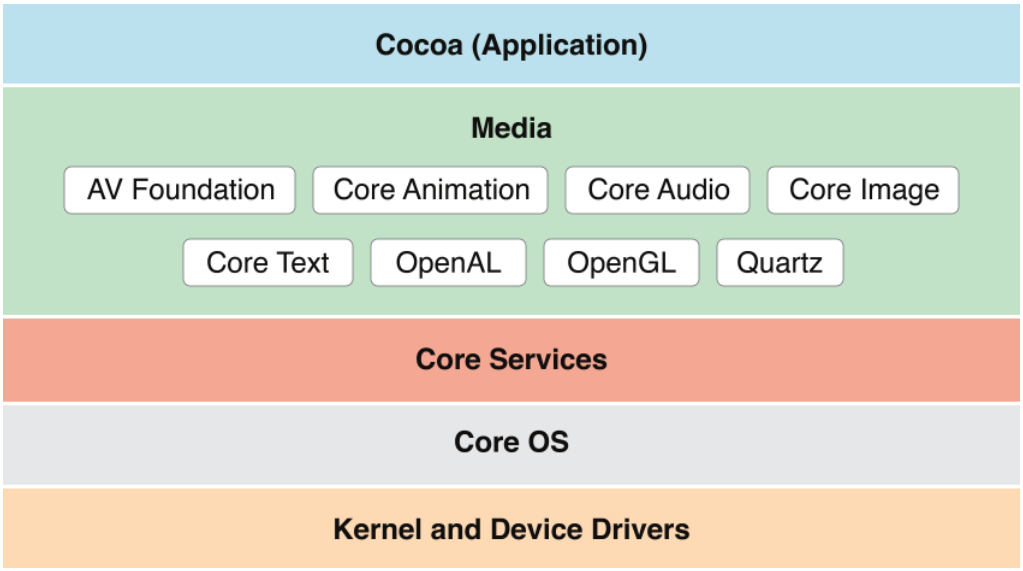


Fig. 3.1.4: Alcuni esempi di quali framework sono contenuti dentro il livello Media

Cocoa Touch

I frameworks di questo livello supportano direttamente le applicazioni basate su iOS.
Include frameworks basati su Objective-C molto importanti nello sviluppo per iPhone OS:

-UIKit framework
-Addressbook framework

-Addressbook UI framework
-Core Location framework

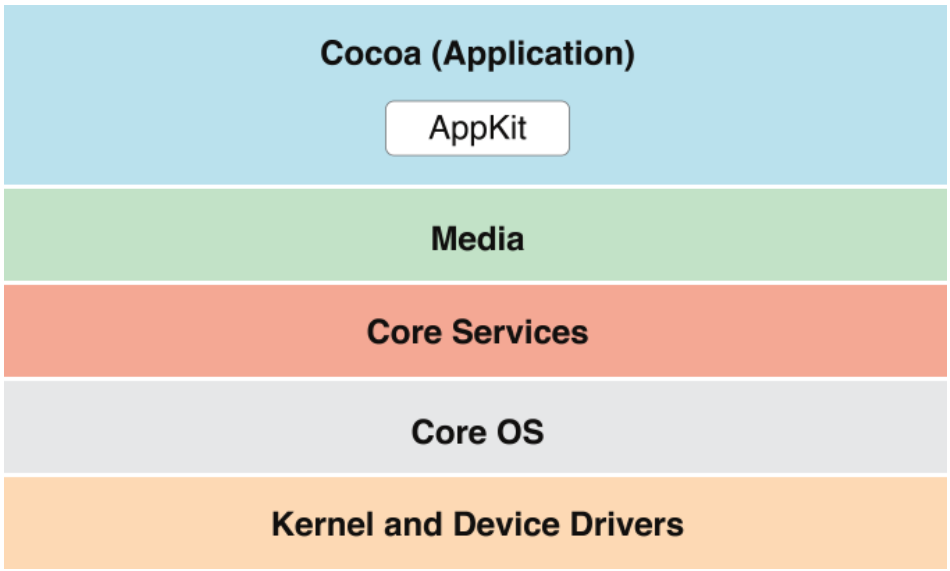


Fig. 3.1.5: Esempio di cosa contenga il livello Cocoa Touch.

I framework più importanti al nostro scopo sono:

•UIKit framework
Contiene gli oggetti per la creazione di interfacce per gli utenti e definisce la struttura per il comportamento di un'applicazione; infatti contiene:

-Application integration
-Servizi di Graphics and windowing
-Event-handling
-Standard views and controls
-Servizi per Web content and text
-Accelerometer data
-Accesso alla fotocamera integrata
-Accesso alla photo library
-Device-specific information

•Foundation framework
Definisce il comportamento base degli oggetti e i meccanismi per la loro gestione. Essenzialmente è una cover orientata ad oggetti del framework Core Foundation. Fornisce ,inoltre, agli oggetti:

-Funzionamento di Core Cocoa framework per non-UI
-Root class
-Strutture e interazioni del Sistema Operativo
-Internationalization
-Collection
-Scripting
-XML processing, Web access

3.2 Ambienti di sviluppo

Per la programmazione di una qualsiasi applicazione iPhone, iPod Touch o iPad l'ambiente di sviluppo messo a disposizione da Apple è una piattaforma SDK che contiene quattro programmi fondamentali:
-Xcode
-Interface Builder
-iPhone Simulator
-Instruments

Gli ultimi due, sono programmi di cui non ci occuperemo approfonditamente: lo scopo del primo è di creare un ambiente di simulazione virtuale del funzionamento dell'applicazione in fase di progettazione e compilazione, la quale è ottima dal punto di vista della simulazione delle APIs, ma non adeguata a quella di prestazioni di sistema, in quanto utilizza le risorse del computer per farla funzionare, e non quelle di un normale iPhone; è buona norma, una volta terminata l'applicazione, richiedere un certificato di autorizzazione alla Apple e testare la propria app direttamente sul dispositivo, per studiarne le prestazioni su di esso. Lo scopo della seconda è invece di dare un'idea dell'utilizzo delle risorse della propria applicazione attraverso grafici durante l'esecuzione.

Adesso,invece, è da rivolgere l'attenzione agli altri due applicativi, poiché fondamentali per lo sviluppo su iOS.

3.2.1 Xcode

Xcode è l'Integrated Development Environment (IDE) della Apple per Mac OS X e iPhone OS. E' anche un'applicazione che si occupa di molti dettagli di progetto. Permette di:

-Creare e gestire progetti
-Scrivere codice sorgente in editors con strumenti per facilitarne il compito, come la syntax coloring e la automatic indentig
-Navigare e cercare attraverso i componenti di un progetto, inclusi gli header file e la documentazione
-Compilare il progetto
-Fare il debug del progetto localmente, nell'iPhone simulator, o in remoto, in un graphical source-level debugger

Xcode compila i progetti dal codice sorgente scritto in C, C++, Objective-C e
Objective-C++. Genera tutti i tipi di eseguibili supportati da Mac OS X,ma per l'iPhone è possibile solo lo sviluppo di applicazioni. Xcode è adatto specialmente allo sviluppo per Cocoa,e in virtù di questo,quando si crea un progetto, Xcode setta l'ambiente iniziale di sviluppo usando il template che corrisponde al tipo scelto.

Attraverso Xcode si procede anche alla richiesta dei certificati e delle autorizzazioni che permettano all'applicazione di funzionare su un dispositivo iPhone, rilasciate direttamente dalla Apple e, una volta finita e testata l'applicazione, la richiesta di rilascio su appStore.

Xcode è un programma strettamente integrato con un altro appartenente a SDK, Interface Builder, del quale daremo una breve panoramica.

3.2.2 Interface Builder

Il secondo maggior software di sviluppo per i progetti Cocoa è Interface Builder. Come suggerisce il suo nome, Interface Builder è uno strumento grafico per creare interfacce utente. Per le prime versioni di iOs questo tool era esterno ad Xcode e modificava in automatico i file contenuti in quest'ultimo. I file in questione sono detti NIB.
Un NIB file contiene gli oggetti che appaiono sull'interfaccia utente in forma archiviata, la quale presenta le informazioni su ciascuno di essi. Quando si crea e salva un'interfaccia utente su Interface Builder, tutte le informazioni necessarie a ricreare tale interfaccia sono racchiuse nel NIB file, il quale viene posto in una directory localizzata nel progetto Cocoa.
Nelle versioni più recenti di Xcode, invece questi due tool sono stati completamente integrati, fino all'arrivo dello strumento *storyboard*, che evita che ogni view utente sia creata in due file separati, racchiudendole tutte in un diagramma facile da interpretare e che permette in maniera più immediata la modifica di più view contemporaneamente, ma soprattutto come collegarle tra loro.

3.3 Introduzione all'Objective C

Come già anticipato precedentemente, il linguaggio principalmente utilizzato per lo sviluppo di un'applicazione iPhone, iPod Touch o iPad, è l'Objective-C, il quale permette di gestire metodi e classi che si rifanno a tutta la struttura gerarchica elaborata per Cocoa. Objective C è un'estensione del linguaggio C e viene usato in tandem con librerie fisse di oggetti standard, denominati framework, come Cocoa e GNUstep. Il programmatore non è obbligato ad ereditare le funzionalità della classe base esistente, NSObject,ma può dichiarare nuove classi base che non ereditino nessuna delle funzionalità preesistenti.

3.3.1 Oggetti

Come suggerisce il nome, questo è un linguaggio di programmazione orientato ad oggetti. Gli oggetti associano dati a particolari operazioni di cui possono usufruire o che possano modificarli. In Objective-C queste operazioni sono chiamate metodi dell'oggetto; i dati su cui esse hanno degli effetti si dicono variabili d'istanza. In pratica, un oggetto raggruppa una struttura di dati (variabili d'istanza) e un gruppo di procedure (metodi) dentro un'unità di programmazione indipendente.

In Objective-C le variabili d'istanza sono interne all'oggetto; generalmente, si ha l'accesso allo stato di un oggetto solamente attraverso i suoi metodi, ed inoltre un oggetto può vedere solo i metodi che sono stati scritti per lui.

In Objective-C gli oggetti identificati sono distinti dal tipo di dato id. Questo tipo è definito con un puntatore a un oggetto, o meglio, alle sue variabili d'istanza.
In pratica, quando un metodo ritorna un valore, *id* sostituisce *int* come tipo di dato di default. La parola chiave nil è definita com un oggetto nullo, un id=0.

Il tipo *id*, però, non contiene informazioni riguardo a un oggetto, eccetto il fatto che esso sia un oggetto. Ma gli oggetti non sono tutti uguali, poiché ognuno ha le proprie istanze e metodi. Il programma ha bisogno di informazioni più specifiche riguardo l'oggetto che contiene. Per supportare ciò, tutti gli oggetti hanno una variabile d'istanza isa che identifica la classe dell'oggetto. Gli oggetti sono dinamicamente tipizzati a runtime: quando necessario il runtime system può trovare l'esatta classe cui un oggetto appartiene soltanto chiedendoglielo.

3.3.2 Sintassi dei messaggi

Il modello di programmazione dell'Objective-C è basato sullo scambio di messaggi tra oggetti. Per dire a un oggetto di fare qualcosa, gli si invia un messaggio dicendogli di applicare un metodo.
L'invio del messaggio *faiQualcosa* all'oggetto *Ogg* è espresso da:

```
[Ogg faiQualcosa];
```

Il nome del metodo in un messaggio serve a selezionare l'implementazione del metodo. Per questo motivo, i nomi dei metodi nei messaggi sono spesso indirizzati ai *selectors*. I metodi sono scritti in maniera diversa dalle funzioni in C, presentandosi come segue:

```
-(int) faiLaRadiceQuadrataDiInt: (int) i {
    return [self radiceQuadrataDiInt:i]; }
```

e può essere evocato nella seguente forma:

```
[unaIstanza faiLaRadiceQuadrataDiInt:4];
```

seguendo la forma:

```
[instance method:Argument];

[instance method:argument method:argument];
```

In questo modo è possibile inviare messaggi ad un oggetto, anche se l'oggetto non è

capace di rispondere.
Il tipo di dato che il metodo ritorna è espresso nella forma comune al C, così come il tipo dell'argomento. Se non viene specificato nulla, si assume che sia di tipo defaut, *id*.

3.3.3 Classi

In Objective-C si definisce un oggetto tramite la sua classe. La definizione di una classe è un prototipo per il tipo di oggetto; dichiara le variabili d'istanza che diventano parte di ogni membro della classe e definisce un gruppo di metodi che tutti gli oggetti della classe possono usare.

Le definizioni di una classe sono additive; ciascuna classe che viene definita è basata su un'altra da cui ha ereditato metodi e variabili d'istanza. La nuova classe implementa semplicemente, o modifica, ciò che eredita. Quindi non è necessario duplicare il codice ereditato. Tutte le classi appartengono a un albero gerarchico che parte dalla singola classe radice (root). Quando si scrive un codice basato sul framework foundation, la classe radice è generalmente NSObject. Tutte le altre classi hanno una superclass, e possono avere un numero di subclass.

Durante il runtime la classe viene trasformata in un "class object", la quale può vedere le variabili d'istanza, ma non utilizzarle perché i metodi creati sono stati fatti per le istanze stesse, e non per la classe. E' per questo, che vengono scritti anche i metodi di classe.

Per permettere a un oggetto di nascondere i propri dati, il compilatore limita lo scopo delle variabili d'istanza,che possono essere di tipo:

@private-> accessibili solo dalla classe che le dichiara
@protected->accessibili solo dalla classe che le dichiara e da quelle che ereditano da essa
@public->accessibili ovunque

Una classe è costituita da due parti:

- *Interfaccia*: dichiara i metodi e le variabili d'istanza della classe, e il nome della superclasse
- *Implementazione*: definisce la classe e i suoi metodi

L'Objective-C richiede che l'interfaccia e l'implementazione di una classe siano dichiarati in blocchi differenti. Per convenzione l'interfaccia è messa in un file con suffisso .h , mentre l'implementazione in un file con suffisso .m.
Sia l'implementazione che l'interfaccia devono essere chiusi da @end.

•Interfaccia

L'interfaccia di una classe è solitamente definita in un file .h
La convenzione usata è quella di assegnare il nome al file basandosi sul nome della classe.

```
#import "NomeSuperclasse.h"

@interface NomeClasse: NomeSuperclasse { //variabili d'istanza }

//metodi di classe
+metodoClasse1
+metodoClasse2

...
//metodi d'istanza
-metodoIstanza1
-metodoIstanza2 ...
@end
```

Da notare che il segno meno indica i metodi d'istanza, e il segno più i metodi di classe. Metodi di classe, metodi d'istanza e variabili d'istanza possono avere lo stesso nome. Se l'interfaccia menziona classi che non appartengono alla propria gerarchia, bisogna importarle esplicitamente o dichiararle con la direttiva @class.

•Implementazione

L'interfaccia dichiara solo i prototipi dei metodi e non i metodi stessi che vengono inseriti nell'implementazione. L'implementazione è solitamente scritta in un file con estensione .m. La convenzione usata è quella di assegnare il nome al file basandosi sul nome della classe, come fatto per l'interfaccia.

```
#import "NomeClasse.h"

@implementation NomeClasse
+metodoClasse1
{ //implementazione }

+metodoClasse2
{ //implementazione } ...
-metodoIstanza1
{ //implementazione }
-metodoIstanza2
{ //implementazione } ...
@end
```

Objective-C offre due termini che possono essere usati nel definire un metodo, per riferirli all'oggetto stesso che deve utilizzare tale metodo: self e super.
Durante l'implementazione capita che due variabili abbiano lo stesso nome, ma che una di esse appartenga ai dati della classe che la sta utilizzando: attraverso *self* (come *this* in Java) non può esistere nessuna ambiguità, perché indica, attraverso la notazione dotted *self.variabile*, che tale variabile utilizzata è quella della classe, mentre l'altra no. In altri contesti invece occorre che la classe invii un messaggio a se stessa, anziché a un altro oggetto, e anche in quel caso viene utilizzata la parola self:

```
[self method:Argument];
```

Discorso analogo può essere fatto con la parola *super*, riferendosi al padre della classe stessa.

3.3.4 Forwarding

Objective-C rende possibile inviare ad un oggetto un messaggio non specificato nella propria interfaccia. Un oggetto può "catturare" questo messaggio è può inviarlo ad un altro oggetto. Questo comportamento è chiamato forwarding o delega del messaggio. In alternativa, è possibile usare un gestore degli errori nel caso il messaggio non possa essere inoltrato. Se l'oggetto non inoltra il messaggio, non gestisce l'errore o non risponde viene generato un errore di run-time.

3.3.5 Proprietà

Mentre negli altri linguaggi le variabili d'istanza richiedono metodi espliciti di lettura e scrittura (getter e setters), Objective-C introduce le proprietà, con la seguente sintassi:

```
@interface Persona: NSObject
{...}
@property(readonly) NSString *nome;
@property(readonly) int età

-(id)initWithName:(NSString)nome età:(int)età; @end
```

Una volta inserite nell'interfaccia, si può accedere alle proprietà usando la notazione: NSString *nome = unaPersona.nome
Oppure con il comando @ synthetize del file di implementazione.

3.3.6 Categorie

Uno dei modi per migliorare la struttura del codice sorgente è quello di suddividerlo in parti più piccole, che in Objective-C si traduce nella creazione di categorie.

Una categoria raccoglie implementazioni di metodi di file separati. Il programmatore può mettere dei gruppi di metodi correlati in una categoria per renderli più leggibili.
Inoltre, i metodi inseriti in una categoria sono aggiunti alla classe al run-time. In questo modo le categorie consentono al programmatore di aggiungere metodi ad una classe esistente senza bisogno di ricompilarla o senza avere accesso al suo sorgente.
I metodi inseriti nelle categorie, diventano indistinguibili da quelli originari della classe quando il programma è in esecuzione. Una categoria ha pieno accesso a tutte le variabili d'istanza della classe, incluse le private.

3.3.7 Allocare e Inizializzare oggetti

Un argomento molto importante su questa breve panoramica sull'Objective-C, tratta il modo in cui si allocano e inizializzano gli oggetti.
Per creare oggetti in Objective-C, infatti, bisogna seguire due step:
-Allocare dinamicamente la memoria per il nuovo oggetto.
-Inizializzare il nuovo spazio di memoria allocato a un appropriato valore.
Un oggetto non è completamente funzionale finché entrambi questi step non sono stati completati. Ciascuno è portato a termine da due metodi differenti, che generalmente si trovano su una singola linea di codice:

```
id anObject = [[classObject alloc] init];
```

Separare allocazione da inizializzazione permette un controllo individuale di ciascuno step così che entrambi possono essere modificati indipendentemente tra loro.
In Objective-C la memoria per nuovi oggetti è allocata usando i metodi di classe definiti nella classe NSObject. Questi metodi sono principalmente due: *alloc* e *allocWithZone:*.

Questi metodi allocano abbastanza memoria per tenere tutte le variabili di istanza per un oggetto che appartiene alla classe ricevente. Non hanno bisogno di essere sovrascritti e modificati in sottoclassi.
I metodi *alloc* ed *allocWithZone* inizializzano una variabile di istanza *isa* di un oggetto appena allocato, in modo che punti alla classe dell'oggetto. Tutte le altre variabili di istanza sono impostate a 0. Di solito un oggetto ha bisogno di essere inizializzato più specificatamente prima di poter essere usato in modo sicuro.

Un metodo *init* normalmente inizializza le variabili di istanza del ricevente, poi le restituisce. E' responsabilità del metodo restituire un oggetto che può essere usato senza errori.
Dato che un metodo *init* potrebbe restituire un oggetto diverso dal ricevente appena allocato, o anche restituire nil, è importante che i programmi usino il valore restituito dal metodo di inizializzazione, non solo quelli ritornati da *alloc* o *allocWithZone*.
Quindi, per inizializzare in modo sicuro, è necessario combinare entrambi i metodi in un'unica riga.

Quando un nuovo oggetto è creato, tutti i bit di memoria (tranne per l'*isa*) - e quindi i valori per tutte le variabili di istanza - sono impostati a 0.
Alla fine dell'utilizzo della variabile, poi, è essenziale per non occupare troppa memoria, rilasciare tutte quelle allocate nel codice con una *release.*

Il controllo della memoria è un argomento molto delicato, soprattutto per dispositivi come degli smartphone. Se non viene accuratamente gestita, l'applicazione va in crash e si chiude senza nemmeno avvisare l'utente. Oltre al metodo *release*, esiste anche il metodo *retain.* Entrambi si occupano di gestire un contatore, il reference count, che agisce su una variabile quando è necessario che il garbage collector non la cancelli prima del tempo. Una *retain* incrementa il contatore, e una *release* lo decrementa. Una volta arrivato a zero, tale variabile viene eliminata. Se questa variabile non viene decrementata correttamente, rischia di andare ad occupare inutilmente la memoria, quindi è molto importante aver presente il numero delle *retain* e *release* fatte.

Nelle ultime versioni di Xcode e di iOs è stato introdotto l'ARC, Automatic Reference Counting, che si occupa da solo di gestire *retain* e *release* e cancellare le variabili quando non servono più, tanto che il compilatore non permette nemmeno più di scrivere tali righe di codice dichiarando errore. Questo strumento può essere molto utile soprattutto quando si gestiscono grandi quantità di dati, come nel caso che andremo ad esaminare, con l'applicazione svolta per questa tesi.

3.3.8 Core Data

Prima di volgere l'attenzione sul progetto di questa tesi è importante soffermarsi un attimo su un'altra struttura resa disponibile dal SDK di Apple, il Core Data.

E' indispensabile, infatti, quando si utilizza un'applicazione sul proprio dispositivo, che essa mantenga memoria dei dati inseriti, come ad esempio un elenco oppure una lista, ovvero che ci sia uno strumento all'interno del device che ne garantisca la permanenza finché non sarà l'utente a decidere di eliminare tali dati.

La funzione del Core Data è proprio questa, cioè quella di mettere a disposizione dell'utente un servizio efficiente che conservi tutte le informazioni che non devono essere temporanee. In realtà dal framework fornito dal Core Service è possibile costruire e utilizzare un database scritto con SQLite. Tale database è relazionale, organizzato in righe e colonne ed esterno al dispositivo. Ma objective-C è un linguaggio ad oggetti e il framework Core Data si propone di creare un compromesso tra la struttura relazionale di un database e la gerarchia a classi del linguaggio utilizzato, salvando i dati direttamente sul device e gestendo la sua memoria. Core Data non è un database, anche se può avere SQLite come backend, ma gestisce dati che sono sottoclassi della *NSManagedObject*, quindi prima devono essere istanziati. Core Data può manipolare direttamente gli oggetti in ram, il che è molto più veloce che per un database.

Le funzioni principali di questo framework sono:

-Tenere aggiornate le connessioni fra gli oggetti automaticamente
-Quando un oggetto viene cancellato è possibile cancellare automaticamente quelli che gli sono collegati
-Agisce su dati residenti in ram
-Istanzia rapidamente nuovi record

Per poter utilizzare tale strumento, alla creazione di un nuovo progetto bisogna spuntare l'opzione *Use Core Data for storage*. In questo modo, vengono scritti automaticamente i metodi di base per poter gestire lo stack del Core Data.

Lo stack è formato da quattro oggetti fondamentali:
-Persistent Object Store
-Persistent Store Coordinator
-Managed Object Model
-Managed Object Context

Il primo si occupa di archiviare gli oggetti persistenti, il secondo ne coordina l'archiviazione, il terzo crea il modello su cui verrà creato il database e l'ultimo è come un blocco per appunti che si segna il nome degli oggetti che dovranno essere salvati rispettando la struttura del modello che è stata creata. Il database può avere un unico modello, quindi ogni volta che il modello originario viene modificato l'applicazione deve essere eliminata e reinstallata per permettere il corretto funzionamento del Core Data, altrimenti va subito in crash e non è possibile utilizzarla.

Il Core Data è organizzato in entità ed attributi. In un analogo con il database SQLite, le entità sono le righe, mentre gli attributi le colonne. Ciascun entità può avere un numero illimitato di attributi, ma ogni attributo appartiene a una sola entità. Si possono, però, creare relazioni tra le varie entità, così che se viene modificato un attributo in una, si modifica anche quello corrispondente alla relazione con l'altra.

A livello di codice, quindi, se vogliamo salvare dei dati, è necessario richiamare l'attributo di una entità e salvare l'oggetto che ci interessa sotto la categoria data da tale attributo. Il Core Data, nel salvataggio, creerà automaticamente una riga di dati contenente tutte le informazioni relative agli attributi salvati. Se delle colonne di attributi non vengono riempite il loro spazio viene impostato come *nil*.

Quando si vuole riempire una *tableView* con gli oggetti memorizzati nel database si ricorre al *fetchedResultsController*, uno strumento del Core Data che facilita la raccolta e la visualizzazione dei dati nelle tabelle. Il *fetchedResultsController* utilizza la richiesta di fetch del managed object context per ottenere i dati. La richiesta di fetch può contenere facoltativamente un descrittore di ordinamento, o un predicato, che determina l'ordinamento dei risultati. Il predicato limita il numero di oggetti restituiti dal fetch.

Il *fetchedResultsController* gestisce la memoria in modo molto aggressivo, assicurandosi che soltanto gli oggetti che servono realmente siano mantenuti in memoria e rilasciando gli altri ogni volta che si verificano problemi di memoria.

Dopo questa breve overview su alcuni strumenti principali su come programmare per smartphone iOs, andiamo ad introdurre l'applicazione svolta per questa tesi:
Music Runner.

4. Music Runner

Music Runner è un'applicazione per smartphone iOs e iPod Touch che può essere utilizzata facilmente come player musicale, e realizzata appositamente per questa tesi.

I dispositivi di questa linea sono diventati famosi inizialmente anche per l'uso che gli utenti ne facevano durante i propri allenamenti di corsa (si ricorda ad esempio l'esperimento *nike plus* con un sensore posto sotto la suola delle scarpe che comunicava con l'iPod), e per tale motivo si è pensato a un modo per rendere tali sessioni più motivanti:

l'idea che ne è scaturita ha dato luogo a Music Runner, un player musicale che seleziona il brano più appropriato in base al ritmo della corsa.

Il metodo di funzionamento è molto semplice e intuitivo; una volta installata l'app, se non sono ancora stati classificati dei brani in base al loro bpm, viene lanciato un piccolo tutorial di introduzione che porta l'utente a scegliere i brani che desidera ascoltare durante le proprie sessioni di corsa.

Questi brani vengono analizzati uno a uno, ne viene ricavato il bpm di ciascuno e salvato attraverso il Core Data in una tabella cui l'utente può accedere in qualsiasi momento attraverso un pulsante di informazione: se non desidera più che alcuni brani vengano riprodotti, può cancellarli dall'elenco, oppure aggiungerne di nuovi per rendere la propria sessione ancora più eterogenea.

Nel momento di inizio allenamento, invece, basta semplicemente lanciare l'applicativo, e il pulsante *play* che si presenta subito ai suoi occhi. Il software in questo modo avvia il player musicale impostandogli uno dei brani a bpm più basso; contemporaneamente l'accelerometro inizia a campionare e un timer a contare i secondi. Quando il brano termina, o l'utente decide di cambiare canzone, viene inviato il segnale complessivo dell'accelerometro e il tempo impiegato all'algoritmo di step detection, che in meno di un secondo ricava la cadenza del passo e la relativa canzone con bpm all'interno dello stesso range. Il pulsante pausa, oltre a mettere in standby in brano, si occupa anche di azzerare i dati dell'accelerometro, interromperne il campionamento e azzerare il tempo relativo, poiché non ha senso campionare il ritmo di una persona che ad esempio può essersi momentaneamente fermata. Se viene ripremuto riprende sia la canzone dal punto di interruzione sia l'accelerometro a campionare dall'inizio.
Il pulsante *stop*, invece, azzera completamente tutto e riporta l'app in situazione di riposo.

Dal punto di vista della scelta dei brani sono state fatte due scelte formali per rendere il funzionamento più coerente con l'uso che viene fatto di questa applicazione:

la prima è stata di dividere in intervalli di 20bpm i range di scelta dentro cui una certa cadenza può cadere, poiché potrebbero esistere brani con ritmo molto vicino, ma non uguali e quindi si è voluto ovviare al problema che nessun brano potesse essere scelto per una differenza davvero irrisoria, di cui l'utente non ne ha nemmeno la percezione; la seconda invece è stata quella di mettere una soglia a 180 bpm: se per un brano viene calcolato un bpm superiore a tale soglia, questo viene dimezzato, perché ritmi di corsa così rapidi, soprattutto per delle sessioni di allenamento di mezz'ora, sono quasi impensabili e quelle canzoni non sarebbero mai state scelte.

Dopo questa breve introduzione andiamo a descrivere com'è tecnicamente stata sviluppata questa applicazione.

4.1 L'interfaccia grafica

L'interfaccia grafica è un argomento molto importante nel campo dello sviluppo software per smartphone, soprattutto per i dispositivi iOs che ne fanno il proprio punto di forza, tanto che una delle prime cose su cui viene posta l'enfasi nelle guide ufficiali è proprio quanto sia importante che la grafica risulti ben studiata e accattivante, ma soprattutto indicata per il tipo di applicativo che si decide di distribuire su appStore.

A questo proposito uno dei framework principali forniti con XCode è l'UIKit, il quale rende possibile programmare in maniera molto veloce e intuitiva qualsiasi interfaccia grafica coerente con dei dispositivi touchscreen, dalle tabelle, ai pulsanti, dalle collection view alla navigation bar.

Nel nostro caso, la struttura dell'app è molto semplice, e non abbiamo fatto ricorso né a *tabBar* né a *navigationBar*, poiché la view principale è quella del player, mentre quella secondaria è quella della tabella con i brani e non ne servono altre.
La struttura grafica che più si addiceva al nostro progetto è quella della utility application: una view principale con un pulsante *info*, il quale una volta premuto porta alla seconda view con un'animazione.

4.1.1 Main View

La main view è quella che si presenta all'utente dopo l'avvio dell'applicazione. E' gestita da un unico file .m e .h e da un solo oggetto dentro la storyboard.

All'interno di questa view sono stati collocati pochi elementi, poiché durante una sessione di corsa non ne servono molti: un pulsante play, uno stop, uno pausa, uno forward e alcune label, di cui una per tenere conto della durata totale della sessione.

Nello stato di riposto dell'app, è visibile solo il pulsante play, tutti gli altri sono come congelati e nascosti perché inutili.
Se viene premuto il tasto play, quest'ultimo diventa invisibile, e gli altri compaiono.

Per far si che queste operazioni possano funzionare bisogna intervenire sia a livello di codice, sia a livello di storyboard. Nel sorgente dobbiamo creare le variabili che distingueranno ciascun pulsante. Siccome sono variabili che si riferiscono ad oggetti in output, dovranno essere dichiarate seguendo questa semantica:

IBOutlet UIButton *nomeVariabile;

In questo modo informiamo la storyboard che il codice agirà su uno dei suoi oggetti, ad esempio rendendolo visibile o invisibile al momento giusto.
I nostri pulsanti, inoltre dovranno compiere delle azioni, una volta premuti, come ad esempio far partire il player musicale: nel file di implementazione devono essere creati dei metodi appositi che abbiano come uscita il tipo *IBAction*:

(IBAction)nomeMetodo;

Anche questa sintassi serve per comunicare con la storyboard, per dire che l'oggetto che verrà collegato a quel metodo lo farà eseguire.

Ultimo passo per rendere operativi i bottoni dell'interfaccia è quello di fare tutti i collegamenti necessari tra nomi, metodi e pulsanti sulla storyboard.
Il file storyboard ha un aspetto di questo tipo:

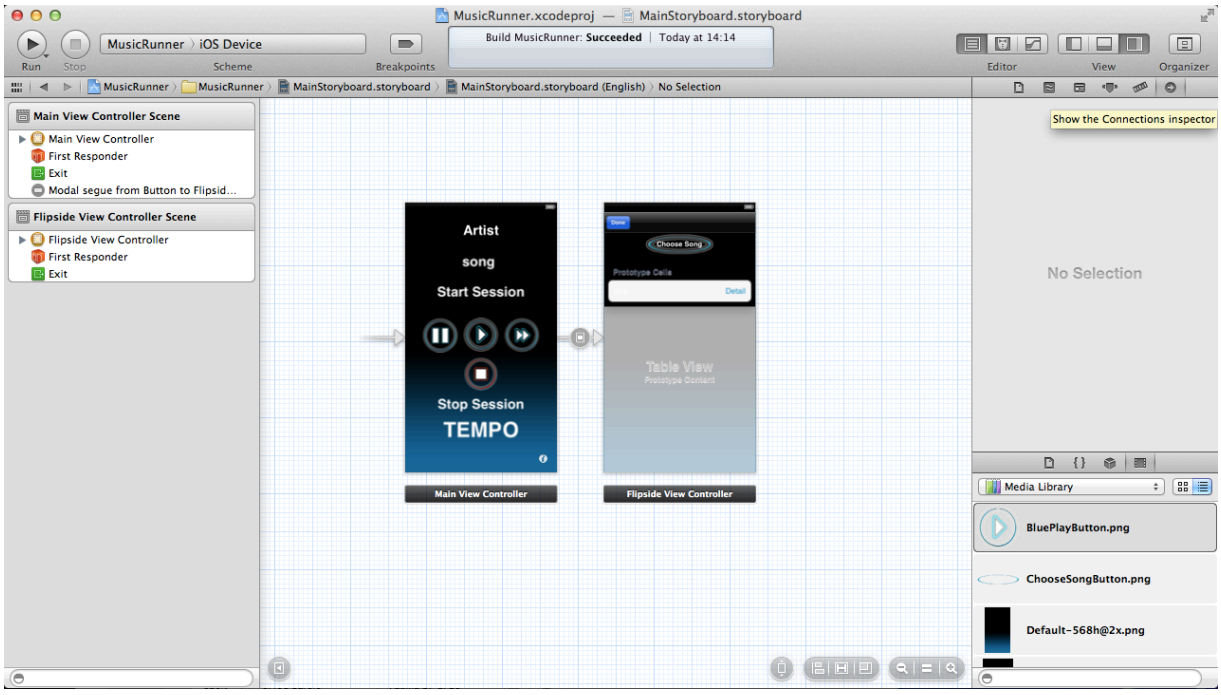


Fig. 4.1.1: come si presenta la storyboard all'interno di Xcode

Per collegare opportunamente un oggetto è necessario selezionarlo e andare nel menù dei link sulla destra, quello con un cerchio e una freccia al centro:

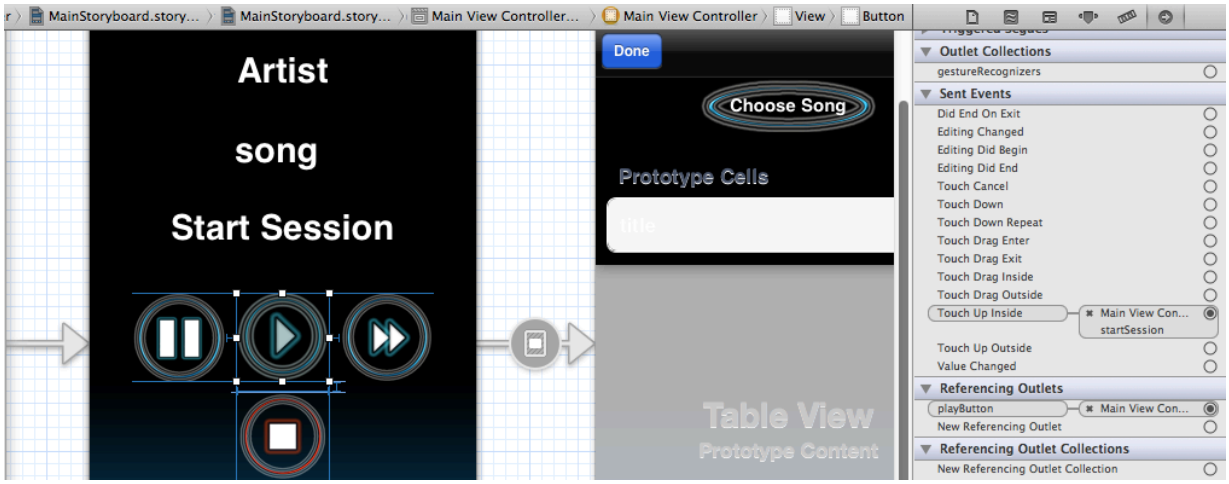


Fig. 4.1.2: Come eseguire i collegamenti con gli elementi di grafica

Come si può notare dall'immagine sono stati fatti due collegamenti; se si seleziona l'intera view infatti, in questo stesso riquadro compaiono i nomi di tutte le variabili *IBOutlet* e di tutti i metodi *IBAction* presenti nel sorgente. Premendo sul pallino di destra della variabile/metodo che ci interessa e trascinandolo fino all'oggetto sull'interfaccia corrispondente, viene creato il collegamento, e l'immagine sopra riportata ne è la conferma: questo pulsante ha come azione il metodo "startSession" del sorgente "MainViewController" e come variabile "playButton" dello stesso file.

Tale operazione deve essere ripetuta per tutti gli oggetti che devono rispondere a degli eventi dell'interfaccia, come ad esempio le UILabel "Artist" e "song" che devono cambiare con il nuovo brano e riportare i campi corretti della canzone in riproduzione.

4.1.2 Flipside View

Una volta premuto il pulsante info in basso a destra della main view, la pagina ruota e viene visualizzata la flipside view. Anche la grafica di questo riquadro è di per sé molto semplice, in quanto deve mostrare all'utente i brani che sono stati catalogati.

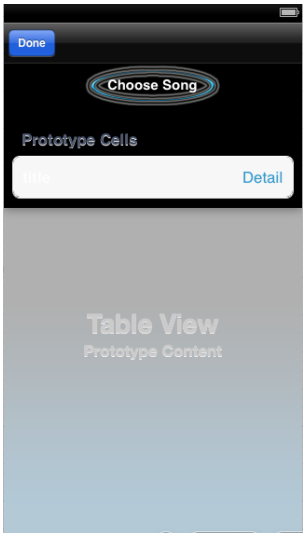


Fig. 4.1.3: Intergaccia grafica della Filpside View

In alto a sinistra il pulsante *done* riporta alla main view, mentre quello "Choose Song" permette di accedere alla libreria iPod e selezionare quanti brani si vuole.

Al centro possiamo vedere una *UITableView*, tecnicamente detta di tipo "gouped" e visualizzazione dei dati "right detail": in questo modo potremo scrivere a sinistra il nome della canzone e a destra quello del corrispondente bpm. Le *UITableView* sono facili da inserire graficamente nelle interfacce, ma richiedono delle implementazioni obbligatorie a livello di codice per poter funzionare.

Nel file interfaccia .h devono essere dichiarati il delegate e il datasource:

```
@interface FlipsideViewController :
UIViewController<MPMediaPickerControllerDelegate,UITableViewDataSource,
UITableViewDelegate,UIAlertViewDelegate>
{NSMutableArray *brani;
}
```

Questa opzione serve per importare tutti i metodi utili alla *UITableView* per funzionare dal framework UIKit. Se non lo facessimo il compilatore non riconoscerebbe i metodi successivi che sono obbligatori per questo tipo di oggetto. Tali metodi sono:

```
- (NSInteger)numberOfSectionsInTableView:(UITableView *)tableView{...}
- (NSInteger)tableView:(UITableView *)tableView numberOfRowsInSection:
  (NSInteger)section{...}
```

```
- (UITableViewCell *)tableView:(UITableView *)tableView
  cellForRowAtIndexPath:(NSIndexPath *)indexPath{...}
```

Il primo imposta il numero di sezioni della table view (nel nostro caso una, ma se fossero di più lo stile *grouped* che abbiamo scelto le farebbe visualizzare separate fra loro); il secondo imposta il numero delle righe: siccome di solito le tabelle vengono riempite con degli array, basta semplicemente impostare il valore di ritorno pari alla dimensione di tale array; il terzo, ancora più importante, imposta i valori di ogni singola cella della tabella, il suo stile, e cosa scrivere nelle label “title” e “detail” di ciascuna di esse.
Senza questi tre metodi la tabella non potrebbe funzionare e non verrebbe nemmeno visualizzata.
Altri metodi interessanti che appartengono al datasource della UITableView sono:

```
-(void)tableView:(UITableView *)tableView commitEditingStyle:
(UITableViewCellEditingStyle)editingStyle forRowAtIndexPath:(NSIndexPath
*)indexPath{...}
-(void)tableView:(UITableView *)tableView didSelectRowAtIndexPath:
(NSIndexPath *)indexPath{...}
```

Il secondo non verrà utilizzato in questa applicazione, ma serve a definire cosa deve succedere se un utente seleziona una riga della tabella; il primo invece fa partire le operazioni di editing della tabella, e cancella gli elementi che l’utente non desidera più.

Questi sono gli elementi più salienti che riguardano l’interfaccia grafica del nostro applicativo smartphone. L’aspetto finale, dopo aver creato degli elementi con photoshop (come l’aspetto dello sfondo e dei bottoni), risulta quello in figura 4.1.4.

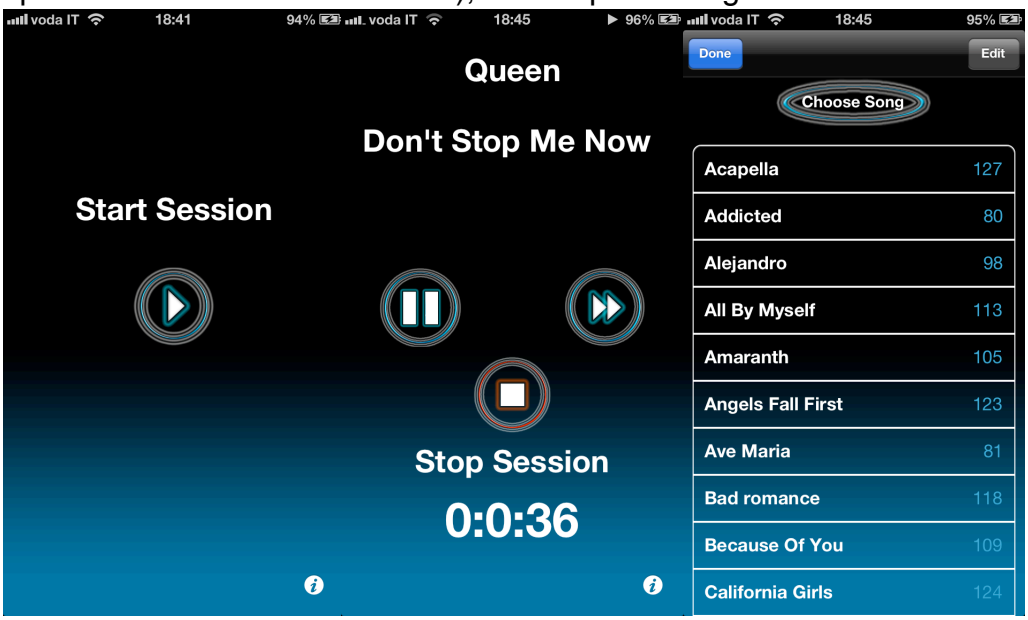


Fig. 4.1.4: Tutte e tre le view della nostra interfaccia grafica

Nel capitolo seguente invece, andremo ad illustrare come è stata svolta l’integrazione con il player musicale e l’iPod.

4.2 Accesso alla libreria iPod e utilizzo del Music Player

Come già anticipato questa applicazione interagisce molto con l'iPod nativo del dispositivo, soprattutto per quanto riguarda l'accesso ai brani della libreria. Tale scelta è stata fatta per rendere l'applicazione più ad personam, in modo che l'utente avesse come scelta brani che lui stesso ha aggiunto in precedenza, anziché fare l'analisi su brani campione e proporre solo quelli all'utente finale. Per fare ciò si è ricorsi allo strumento *mediaPicker*.

MPMediaPickerController è una classe del *framework* MediaPlayer che permette di invocare tutta la libreria iPod installata sullo smartphone e caricare uno o più brani. Anche per questo oggetto è necessario importare il delegato, *MPMediaPickerControllerDelegate*, e funziona attraverso due metodi principali:

```
- (void)mediaPicker: (MPMediaPickerController *)mediaPicker
  didPickMediaItems:(MPMediaItemCollection *)mediaItemCollection {...}
- (void)mediaPickerDidCancel:(MPMediaPickerController *)mediaPicker
  {...}
```

Il primo indica che quanto la finestra del picker è stata chiusa dall'utente, dopo aver selezionato dei brani, viene creata una *MPMediaItemCollection*, ovvero una collezione di oggetti *MPMediaItem*; dentro tale metodo è possibile gestire la collezione: nel nostro caso ciascun oggetto in essa contenuta verrà elaborato dall'algoritmo di beat detection e poi l'id corrispondente a ciascun brano verrà salvato in memoria con il bpm ad esso associato.
Il secondo metodo invece si occupa di dire cosa succede se l'operazione viene annullata: nel nostro caso viene semplicemente chiusa la finestra del picker e si ritorna alla flipside view senza che nulla accada.

Come è possibile invocare il picker? Come abbiamo illustrato nella sezione relativa alla grafica dell'app, nella flipside view è presente un pulsante, *choose song*, il quale è collegato a un metodo IBAction che si occupa di inizializzare e far comparire il picker come segue:

```
MPMediaPickerController *pickerController =
[[MPMediaPickerController alloc] initWithMediaTypes: MPMediaTypeMusic];
pickerController.prompt = @"Choose songs";
pickerController.allowsPickingMultipleItems = YES;
pickerController.delegate = self;
[self presentViewController:pickerController animated:YES completion:nil];
```

Nel primo metodo viene allocato e inizializzato un nuovo oggetto picker, e successivamente ne vengono impostati i valori di default, in particolare se l'utente è autorizzato a selezionare più di un oggetto. Alla fine viene lanciata una nuova view sopra alla flipside che è quella di default del picker. Tale view ha un'aspetto di questo tipo:

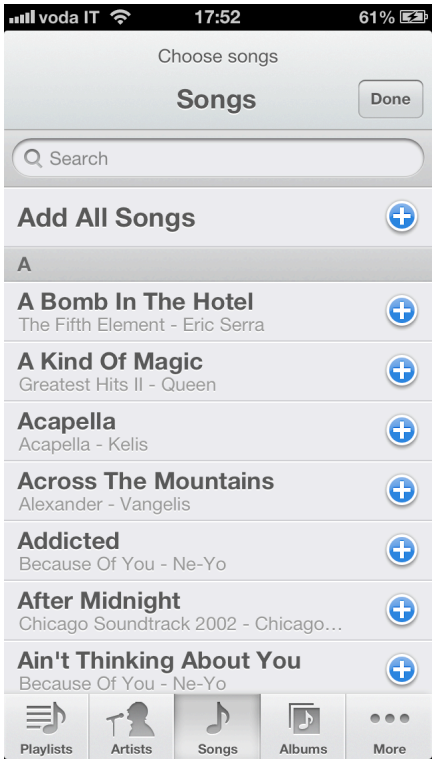


Fig. 4.2.1: View del picker controller quando vengono caricati brani dalla libreria iPod

L'utente può così scorrere facilmente nella sua libreria multimediale, cercando un brano specifico, aggiungendoli tutti, o visualizzarli secondo altri criteri, come playlist o autori del brano. Una volta premuto il tasto *done*, se non è stato selezionato nulla, l'applicazione non svolge alcuna operazione, se invece la *MPMediaItemCollection* contiene almeno un file multimediale questo viene elaborato ed aggiunto alla *UITableView*.

Nella main view invece, occorre sfruttare tutt'altra funzionalità del framework MediaPlayer, ovvero quella del *MPMusicPlayerController*. Questo strumento può essere inizializzato secondo due diverse modalità:

```
[MPMusicPlayerController iPodMusicPlayer];

[MPMusicPlayerController applicationMusicPlayer];
```

Scegliendo quella di tipo "iPod" il brano scelto dalla nostra app sfrutterà l'iPod integrato per essere riprodotta. Il player si comporterà in maniera appropriata in base alle diverse situazioni: ad esempio se l'applicazione viene messa in background la canzone continuerà ad essere riprodotta; se viene schiacciato il pulsante pausa o avviene un'interruzione come una chiamata telefonica, una volta che l'utente decide di riprendere ad ascoltare il brano ripartirà da dove si è fermato. Se l'applicazione viene chiusa senza che il music player sia stato interrotto con l'apposito pulsante e il brano non è terminato, questo continua ad essere riprodotto finché non viene messo in stop dal player iPod.

Scegliendo il tipo “application”, invece, non abbiamo nessuna di queste specifiche: se il brano viene messo in pausa, una volta che riparte ricomincia da capo; se l’applicazione va in background la canzone viene interrotta, idem se viene chiusa del tutto senza aver premuto il pulsante di fine sessione.

Quale delle opzioni scegliere dipende solo dalle specifiche desiderate, che nel nostro caso si sono orientate di più verso la soluzione iPod.

Una volta inizializzato il player, bisogna impostare il brano che deve riprodurre. Grazie al Core Data abbiamo una libreria interna con un parametro memorizzato molto importante: l’id della canzone. Salvare il brano con il proprio id, anziché per nome, evita ambiguità con i casi di omonimia fra più canzoni, che possono avere bpm diverso. L’id, invece, è univoco per ogni brano della libreria.

Come impostazione di default abbiamo deciso che il primo brano che debba essere caricato nel player sia scelto random tra quelli a bpm della fascia più bassa. Attraverso una *query* il brano viene ricercato nella libreria iPod e fatto eseguire appena l’utente schiaccia il pulsante play. La struttura della *query* è di questo tipo:

```
MPMediaQuery* query = [MPMediaQuery songsQuery];

[query addFilterPredicate:[MPMediaPropertyPredicate
predicateWithValue:numberID forProperty:MPMediaItemPropertyPersistentID
comparisonType:MPMediaPredicateComparisonEqualTo]];

//passa la query al player
[myPlayer setQueueWithQuery:query];
```

Viene creata una *query* di tipo *song* e gli viene aggiunto un predicato con cui filtrare i brani della libreria. La *query* infatti può essere usata anche per caricare più brani che debbano susseguirsi nella riproduzione, come ad esempio tutti quelli che appartengono a un determinato artista. Nel nostro caso il predicato è quello dell’id, che riporterà un unico valore in uscita, quindi una sola canzone per volta. Ottenuto il brano di interesse, viene passato al player musicale.

Le *MPMediaItemProperty* sono molto utili in operazioni di questo tipo. Alcuni esempi di quelli esistenti sono i seguenti:

```
NSString *const MPMediaItemPropertyPersistentID;            // filterable
NSString *const MPMediaItemPropertyAlbumPersistentID;       // filterable
NSString *const MPMediaItemPropertyArtistPersistentID;      // filterable
NSString *const MPMediaItemPropertyAlbumArtistPersistentID; // filterable
NSString *const MPMediaItemPropertyPlaybackDuration;
NSString *const MPMediaItemPropertyAlbumTrackNumber;
NSString *const MPMediaItemPropertyAlbumTrackCount;
NSString *const MPMediaItemPropertyBeatsPerMinute;
```

Come possiamo notare, alcune di esse sono commentate come *filterable*: quelli sono i valori che è possibile mettere come predicato della *query*. Un altra proprietà che salta agli occhi è quella identificata come *BeatsPerMinute*: alla luce di ciò può sembrare che il lavoro che è stato fatto per implementare un algoritmo di beat detection sia assolutamente inutile, ma ciò non è vero. Questo attributo infatti è di default impostato a zero, a meno che l'utente dalla sua libreria iTunes sul computer non si sia messo a modificare a mano tutti i valori di ciascuna canzone in essa contenuta. Non è accettabile dare per scontato che questa operazione sia stata fatta, al massimo si potrebbe mettere un controllo dopo che il mediaPicker abbia importato i brani per vedere se quel tag sia stato impostato o no. Sarebbe interessante poter modificare il campo una volta che il nostro algoritmo abbia calcolato il corrispondente bpm, ma anche in questo caso non è possibile poiché la specifica *const* rende queste stringhe immutabili da ciò che è esterno ad iTunes.

L'operazione di caricamento del brano sopra discussa, deve essere fatta ogni volta che la canzone precedente finisce o quando l'utente schiaccia il pulsante forward, altrimenti l'iPod non riprodurrà nulla e l'utente si ritroverà con del silenzio nelle cuffie.

Per chiudere questo capitolo mancano solo i metodi di comando per il music player; una volta che l'utente digita il pulsante specifico, vengono azionati i seguenti messaggi, del tutto intuitivi:

```
[myPlayer play];
[myPlayer pause];
[myPlayer stop];
```

4.3 Uso del Core Data

Dopo aver citato più volte lo strumento Core Data, andiamo ad illustrare come esso debba essere implementato perché funzioni correttamente.

Se si decide che la propria applicazione dovrà usare questo strumento, nell'atto della creazione bisogna spuntare l'opzione *Use Core Data*:

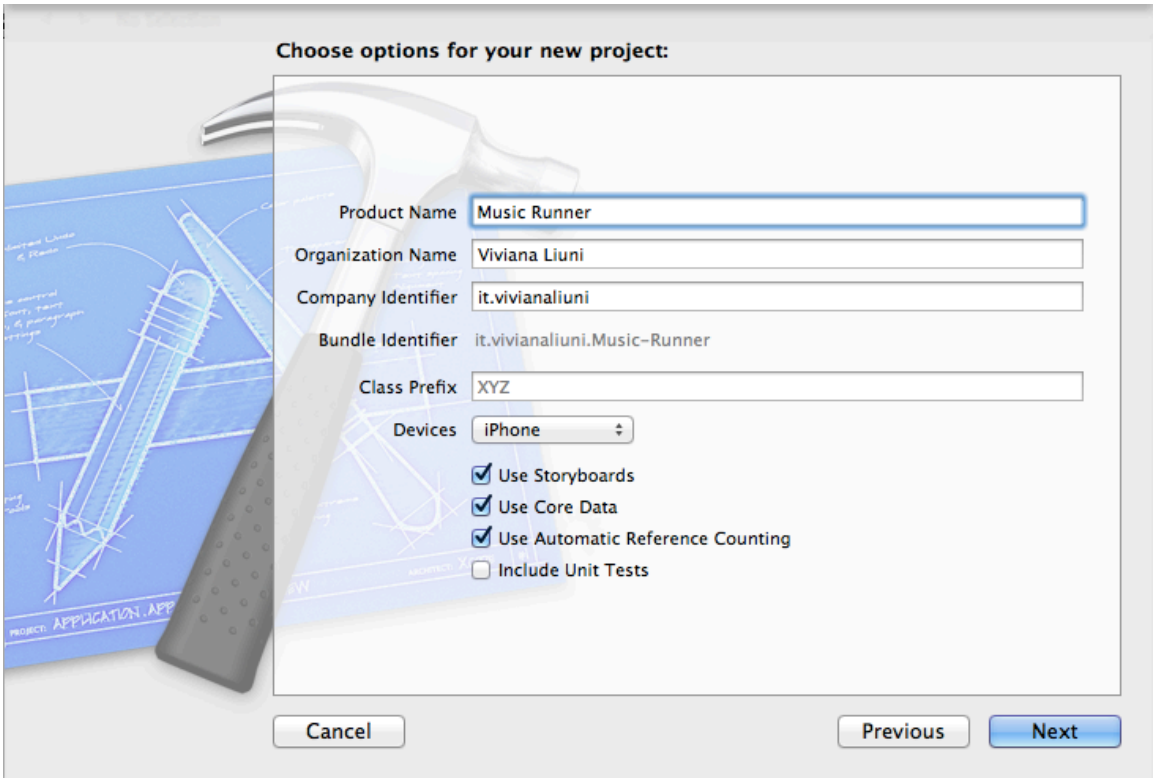


Fig. 4.3.1: Schermata di creazione di una nuova applicazione

In questo modo verranno automaticamente generati i file e le righe di codice obbligatori per il corretto funzionamento di tale opzione, in particolare tutti i metodi di salvataggio dentro l'appDelegate (l'appDelegate è un file sempre presente e creato di default che gestisce l'avvio di ogni app: nel nostro caso all'avvio deve essere caricata anche la struttura del Core Data), e un file *.xcdatamodeld* che lo sviluppatore deve modificare per creare il modello che gli serve.

Nel nostro caso tale file si presenta come in figura 4.3.2.

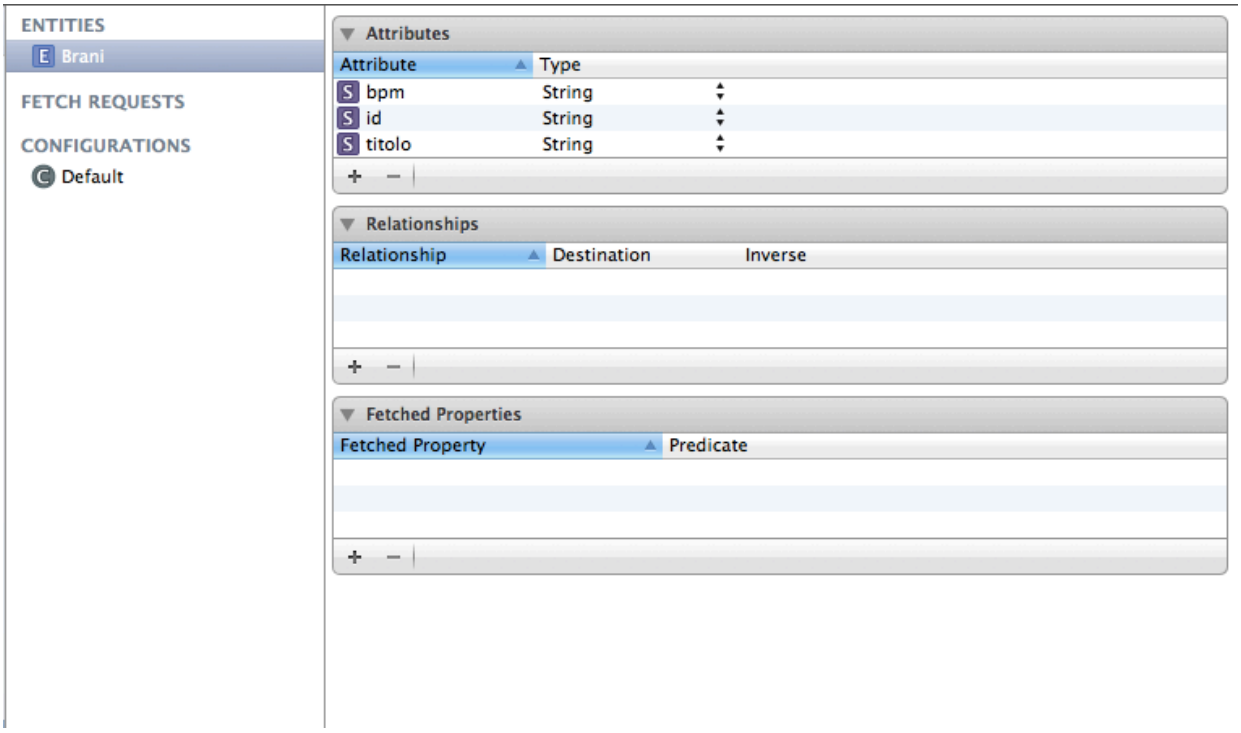


Fig. 4.3.2: Schermata di creazione della struttura dei futuri oggetti del Core Data di questa applicazione

In alto a sinistra troviamo le *entità*, ovvero una collezione che contiene al suo interno tanti oggetti creati secondo lo stesso modello. Nel nostro caso ogni oggetto che viene salvato sotto l'entità *Brani* contiene tre key diverse, qui chiamate *attributi*, che sono *bpm, id, titolo*, e che verranno salvate come delle stringhe. Ogni volta che salveremo un oggetto con il modello *Brani* nel Core Data, dovremo inserire i campi appositi corrispondenti a ciascun attributo. Allo stesso modo se vogliamo caricare le informazioni che ci servono, come l'id, basterà chiedere a tale oggetto di consegnarci il valore che corrisponde a questo attributo.

Nella flipside view dobbiamo invocare il Core Data sia per la funzione di salvataggio sia per quella di caricamento dei dati da visualizzare nella *UITableView*.

Prima di procedere con il codice è obbligatorio incollare i seguenti metodi, altrimenti le nostre operazioni non avranno alcun esito, e anzi daranno luogo ad errori di compilazione:

```
- (void)saveContext{...}

// Returns the managed object context for the application.
// If the context doesn't already exist, it is created and bound to the
persistent store coordinator for the application.
- (NSManagedObjectContext *)managedObjectContext{...}

// Returns the managed object model for the application.
// If the model doesn't already exist, it is created from the
application's model.
- (NSManagedObjectModel *)managedObjectModel{...}
```

```
// Returns the persistent store coordinator for the application.
// If the coordinator doesn't already exist, it is created and the
application's store added to it.
- (NSPersistentStoreCoordinator *)persistentStoreCoordinator{...}

#pragma mark - Application's Documents directory

// Returns the URL to the application's Documents directory.
- (NSURL *)applicationDocumentsDirectory{...}
```

Dopo aver scritto tali metodi nel file .m, andiamo a dichiarare le variabili necessarie nel file .h:

```
@protected NSManagedObjectContext *managedObjectContext;
@protected NSManagedObjectModel *managedObjectModel;
@protected NSPersistentStoreCoordinator *persistentStoreCoordinator;
```

La variabile che più ci interessa e che dovremo usare è il *managedObjectContext*, poiché invocandolo possiamo caricare tutte le entità specificate nel file *xcdatamodeld* della nostra app.

Il salvataggio dei dati è di per sé molto semplice ed immediato.
Dobbiamo prima di tutto creare un nuovo oggetto *NSManageObject* con l'entità *Brani* presa dal nostro context:

```
NSManagedObject *oggetto =[NSEntityDescription
insertNewObjectForEntityForName:@"Brani" inManagedObjectContext:context];
```

Dopodiché assegnamo le nostre variabili agli specifici attributi:

```
[oggetto setValue:ID forKey:@"id"];
[oggetto setValue:titolo forKey:@"titolo"];
[oggetto setValue:[NSString stringWithFormat:@"%d",tag] forKey:@"bpm"];
```

Infine salviamo e controlliamo che non ci siano errori:

```
NSError *error;
if (![context save:&error]) {
NSLog(@"Errore durante il salvataggio: %@", [error
localizedDescription]);}
```

In questo modo, il nostro *NSManagedObject* così costituito viene aggiunto allo stack del Core Data. Come già anticipato in un capitolo precedente, se non viene assegnato nessun valore a uno o più degli attributi, questi vengono salvati come *nil*.

Il caricamento dei dati, invece, richiede anch'esso un metodo standard che esegue una *fetchRequest*. Se la tabella di nostro interesse deve essere riempita all'avvio della view e ogni volta che si aggiungono o rimuovono brani dal Core Data, conviene creare un metodo a parte da invocare ogni volta che se ne ha bisogno. Il metodo nel nostro codice è stato chiamato caricaBrani:

```
-(void) caricaBrani{
    //Otteniamo il puntatore al NSManagedContext
    NSManagedObjectContext *context = [self managedObjectContext];

    //istanziamo la classe NSFetchRequest
    NSFetchRequest *fetchRequest = [[NSFetchRequest alloc] init];

    //istanziamo l'Entità da passare alla Fetch Request
    NSEntityDescription *entity = [NSEntityDescription
entityForName:@"Brani" inManagedObjectContext:context];
    //Settiamo la proprietà Entity della Fetch Request
    [fetchRequest setEntity:entity];

    //Eseguiamo la Fetch Request e salviamo il risultato in un array,
per visualizzarlo nella tabella
    NSError *error=nil;
    NSArray *fo = [context executeFetchRequest:fetchRequest
error:&error];
    [self.brani addObjectsFromArray:fo];
    [self.brani sortUsingComparator:(NSComparator)^(id obj1, id obj2){
        NSString *lastName1 = [obj1 valueForKey:@"titolo"];
        NSString *lastName2 = [obj2 valueForKey:@"titolo"];
        return [lastName1 caseInsensitiveCompare:lastName2]; }];

    [self.tabella reloadData];
}
```

La parte commentata in verde è quella che riguarda le operazioni di default da eseguire per caricare tutti gli NSManagedObject presenti nel nostro context con entità di tipo *Brani*.
Una volta che la richiesta ha generato un array, lo andiamo a collocare dentro un altro array visibile a tutta la classe, nel nostro caso `brani` (da notare la notazione `self` posta prima di questa variabile che indica che tale array appartiene alle variabili di istanza della classe stessa). Dopo aver riempito l'array abbiamo aggiunto una piccola clausola di ordinamento in base ai titoli dei vari oggetti che verranno visualizzati. Alla fine di questo metodo la tabella viene ricaricata per permettere di visualizzare tutti gli oggetti.

Una volta caricati nel nostro array tutti gli oggetti bisogna comunicare alla tabella come visualizzarli. Come anticipato dobbiamo implementare dei metodi obbligatori per la tableView, i quali sono stati così completati:

```
- (NSInteger)numberOfSectionsInTableView:(UITableView *)tableView
{
    // Return the number of sections.
    return 1;}
```

```
- (NSInteger)tableView:(UITableView *)tableView numberOfRowsInSection:
(NSInteger)section
{

    return [brani count];}

- (UITableViewCell *)tableView:(UITableView *)tableView
cellForRowAtIndexPath:(NSIndexPath *)indexPath
{
    static NSString *CellIdentifier= @"cella_canzoni";
    UITableViewCell *cell = [tableView
dequeueReusableCellWithIdentifier:CellIdentifier
forIndexPath:indexPath];

    NSManagedObject *brano=[brani objectAtIndex:indexPath.row];

    cell.textLabel.text=[NSString stringWithFormat:@"%@",[brano
valueForKey:@"titolo"]];
    cell.detailTextLabel.text=[NSString stringWithFormat:@"%@",[brano
valueForKey:@"bpm"]];

    return cell;
}
```

Questi metodi vengono invocati ogni volta che ricarichiamo la tabella. In numero di righe è pari alla lunghezza dell'array `brani`, e le celle vengono riempite con il suo contenuto. Il nostro array infatti viene scorso per indice, in base all'indice di riga di ciascuna cella: per ogni elemento dell'array viene creato un nuovo *NSManagedObject* preso dal corrispondente indice in `brani`, e il valore con attributo *titolo* viene messo nella textLabel della cella, mentre quello con attributo *bpm* viene messo nella detailTextLabel della stessa. L'attributo ID non viene visualizzato perché andrebbe ad appesantire inutilmente la grafica; esso è caricato solo nella main view nel momento in cui deve essere eseguita la *query*.

Quando l'utente decide di cancellare un brano, invece, questo deve essere eliminato anche dal context:

```
-(void)tableView:(UITableView *)tableView commitEditingStyle:
(UITableViewCellEditingStyle)editingStyle forRowAtIndexPath:(NSIndexPath
*)indexPath{

    if (editingStyle == UITableViewCellEditingStyleDelete) {
        // Delete the row from the data source.
        NSManagedObject *eventToDelete = [brani
objectAtIndex:indexPath.row];
            [[self managedObjectContext] deleteObject:eventToDelete];

           NSError *error=nil;
           if (![[self managedObjectContext] save:&error]) {
                NSLog(@"Errore durante il salvataggio: %@", [error
localizedDescription]);
           }}
    brani=[[NSMutableArray alloc] initWithObjects: nil];
    [self caricaBrani];}
```

Nel metodo che si occupa di gestire la modalità editing della tabella andiamo a creare un *NSManagedObject* che corrisponde alla riga selezionata dall'utente, poi lo eliminiamo dal context e salviamo. Alla fine, invocando il metodo caricaBrani, andiamo a ricaricare la tabella senza l'elemento cancellato.

Queste sono le semplici operazioni che basta fare per poter usare al meglio il salvataggio di una certa mole di dati. Il Core Data si presta anche facilmente al salvataggio su iCloud.

iCloud è una piattaforma di salvataggio e condivisione online di dati fra i vari dispositivi iOs e Os X posseduti da quell'utente. Se viene abilitato su un'applicazione questa salva i suoi elementi su tale database remoto, ed è possibile recuperarli sia da altri dispositivi sia nel caso di smarrimento o di cancellazione involontaria dell'app. Il Core Data, infatti, non è immune da quest'ultima eventualità: se l'applicazione viene cancellata dallo smartphone, i suoi dati vengono tutti persi, e se installata nuovamente bisogna compiere da capo tutte le operazioni. Impostando il caricamento dei dati del Core Data su iCloud, invece, se l'utente ne consente l'utilizzo, tutti gli *NSManagedObject* vengono automaticamente trasferiti sul database remoto, e allo stesso modo recuperati nel caso di bisogno. Operazione senz'altro comoda se l'utente ha perso mezza giornata per calcolare i bpm di tutta la sua libreria musicale e questi rischiano di essere persi per un semplice errore.

Prima di andare a descrivere com'è stato implementato l'algoritmo di beat detection, andiamo brevemente a descrivere un framework fondamentale per il suo sviluppo: *accelerate*.

4.4 Framework: accelerate

Uno strumento assolutamente necessario ai fini del signal processing che andremo a eseguire è indubbiamente la FFT. La Fast Fourier Transform è un algoritmo ottimizzato per calcolare la trasformata di Fourier discreta. Fortunatamente il framework accelerate viene in nostro soccorso fornendo l'implementazione di tale funzione.

In particolar modo dentro questo framework a noi interessa la vDSP library, che offre funzioni matematiche sia per l'elaborazione di file audio, ma anche di file immagine o video.

4.4.1 Introduzione a vDSP

La vDSP API lavora con tipi di dato reali e complessi. Le sue funzioni includono trasformate di Fourier, conversioni di tipo di dato, e operazioni vettoriali. Tali funzioni sono state implementate in due modalità: come codice vettorizzato, il quale usa istruzioni vettoriali del processore, o come codice scalare. La vDSP API usa la versione più appropriata in base agli argomenti che le vengono passati e dalle capacità vettoriali del processore.

La maggior parte delle funzioni di vDSP richiedono un input e producono un output e qualunque sia il tipo di dato, se vettore o scalare, viene passato per referenza, quindi attraverso puntatori alla memoria dove i dati di input devono essere letti e quelli di output devono essere scritti. Ogni funzione vDSP, quindi, avrà come tipo di ritorno un *void*, poichè il dato in uscita viene modificato come puntatore.

Altri argomenti che possono essere richiesti dalle funzioni vDSP sono:
- Adress strides, i quali dicono alla funzione di quanto scorrere lungo l'array passato in input o in output.
- Flags, che influenzano il comportamento della funzione in qualche modo, ad esempio dicendo se la FFT debba essere diretta o inversa.
-Element counts, che dicono alla funzione quanti elementi processare.

Tutti e tre sono valori interi e vengono passati alla funzione direttamente, non come puntatori.

Riguardo gli Address Strides possiamo aggiungere che oltre a poter essere positivi, alcune funzioni li prevedono anche negativi e ne possono cambiare l'aritmetica delle operazioni.

Riguardo l'uso di vettori complessi, invece, ci sono due formati diversi per memorizzare ciascun numero complesso dentro un vettore di elementi: split complex e interleaved complex.
Nei vettori di split complex (*DSPSplitComplex* e *DPSDoubleSplitComplex*), la parte reale degli elementi nel vettore è memorizzata in un array e quella immaginaria in un altro. Ogni elemento per essere letto necessita di uno stride pari a 1. Molte funzioni, come la FFT, fanno uso di questo tipo di dato.
Nei vettori interleaved complex (*DSPComplex* e *DSPDoubleComplex*), i numeri complessi sono memorizzati come coppie pari di floating point o valori double. Per scorrere ogni elemento del vettore è necessario uno stride pari a 2.

Piccola notazione formale, è che ogni funzione di questa libreria ha sempre il prefisso *vDSP_* ed eventualmente un suffisso *D* che indica la double-precision; se tale suffisso non è presente, le operazioni vengono eseguite in single-precision.

4.4.2 FFT Weights Arrays

La vDSP API offre trasformate di Fourier per trasformare in una o due dimensioni dati tra il dominio del tempo e quello delle frequenze.

Per migliorare le prestazioni, le funzioni vDSP che elaborano i dati nel dominio delle frequenze si aspettano un array di esponenziali complessi (a volte chiamati fattori di twiddle) che devono esistere prima che la funzione venga chiamata. Una volta creato, questo array di pesi della FFT può essere riutilizzato più volte dalla stessa funzione di Fourier o condivisa anche con altre.

Gli array con i pesi della FFT sono creati chiamando la *vDSP_create-fftsetup* (single-precision) o la *vDSP_create-fftsetupD* (double-precision). E' obbligatorio invocare una di queste due funzioni prima di svolgere qualsiasi operazione nel dominio delle frequenze. Esse richiedono in ingresso la dimensione dell'array per svolgere le operazioni di setup, che sono:
-Creare una struttura dati che contenga l'array
-Costruire l'array
-Restituire un puntatore a una struttura dati (o NULL se non può essere allocata)

Il puntatore alla struttura dati è poi passato come argomento alle funzioni di Fourier quando vengono usate.

Le funzioni di setup prendono in ingresso un argomento chiamato *log2n*: indica specificatamente che bisogna passargli il logaritmo in base 2 di *n*, dove *n* è il numero di divisioni della circonferenza unitaria complessa che l'array rappresenta, e specifica il massimo numero di elementi che possono essere processati dalle successive funzioni di Fourier. Questo argomento log2n deve essere uguale o superiore allo stesso argomento log2n richiesto in ingresso anche dalle funzioni Fourier che utilizzano questo array di pesi.

4.4.3 Packing di array monodimensionali

Le funzioni di trasformate discrete di Fourier nella vDSP API usano un caso unico di data formatting per preservare memoria. Queste funzioni scrivono i propri output in un speciale formato così che l'uscita complessa non richieda più memoria di quella dell'ingresso reale.

Le applicazioni che usano la FFT devono usare due funzioni, una prima della FFT e l'altra dopo.

Un array di reali deve essere trasformato in un array even-odd prima che venga trasformato, ovvero un array con prima tutti gli indici pari, poi tutti quelli dispari. Questo viene fatto attraverso la funzione *vDSP_ctoz*.
Il risultato di una FFT su un *AEvenOdd* di dimensione *n* è un array complesso di dimensione *2n*, con un formato speciale:

A= {A[0],...,A[n]} //Array di reali
A= {A[0],A[2]...A[n-1],A[1],A[3],...A[n]} //Array even-odd
C= {[DC,0],C[1],C[2],..C[n/2],[NY,0],Cc[n/2],...Cc[2],Cc[1]} //Array di complex

Dove:
- DC e NY sono le componenti DC e Nyquist (valori reali)
- L'array C è un complesso nella rappresentazione split
- L'array Cc è il complesso coniugato di C nella rappresentazione split

Per un array reale *A* di dimensione *n*, il risultato complesso richiede una dimensione di spazio *2*n*. Tuttavia, molti di questi dati sono per la maggior parte zeri o ridondanti, e possono essere omessi. Per memorizzare il risultato nello stesso spazio dell'array di input l'algoritmo elimina i valori non necessari. La FFT reale memorizza il risultato come segue:
C= {[DC,NY],C[1],C[2],...,C[n/2]}

Nella documentazione ufficiale [5] vengono trattati i packing formats sia per il caso monodimensionale sia quello bidimensionale. Siccome per i nostri scopi è sufficiente conoscere solo il primo caso, parleremo di quello e ometteremo il secondo.

A causa della sua innata simmetria, la trasformata di Fourier con *n* floating-point inputs produce *n/2+1* outputs di complessi. Poichè i dati dei punti *n/2-i* sono uguali ai complessi coniugati dei punti *n/2+i,* la prima metà dei dati frequenziali sono sufficienti per mantenere l'informazione originale.

Inoltre la FFT data packing sa già che la parte immaginaria del primo e ultimo elemento complesso di output è sempre zero. Ciò rende possibile memorizzare la parte reale dell'ultimo elemento di output dove dovrebbe esserci il primo elemento della parte immaginaria, poichè il primo e ultimo valore nullo di quest'ultima sono sempre impliciti.

Per esempio, consideriamo un vettore reale di otto punti:

Fig. 4.4.1: rappresentazione di un vettore di reali

Trasformando questi otto valori reali nel dominio delle frequenze ne risultano cinque valori complessi:

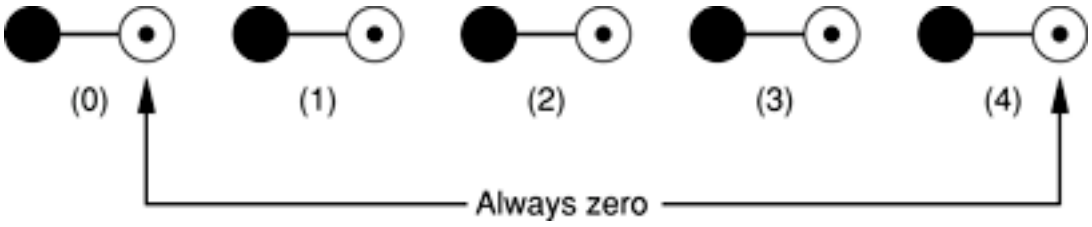


Fig. 4.4.2: rappresentazione di un vettore complesso nel dominio delle frequenze

I cinque valori complessi sono "impachettati" nel vettore di output come segue:

Fig. 4.4.3: rappresentazione di come risulta il packaging finale del vettore complesso nel dominio delle frequenze

4.4.4 Scaling delle trasformate di Fourier

Per offrire una velocità di esecuzione migliore possibile, le funzioni della libreria vDSP non sempre aderiscono strettamente alle formule matematiche della trasformata di Fourier, e devono essere di conseguenza scalate. Questo paragrafo si occupa di specificare lo *scaling* per ciascun tipo di trasformata di Fourier implementata nella libreria vDSP. I fattori di scaling sono anche dichiarati esplicitamente nelle formule che accompagnano le definizioni delle varie funzioni nella guida ufficiale [5].

I fattori di scaling sono i seguenti:

One-Dimensional Transforms

Real forward transforms: $RF_{imp} = RF_{math} * 2$

Real inverse transforms: $RI_{imp} = RI_{math} * n$

Complex forward transforms: $CF_{imp} = CF_{math}$

Complex inverse transforms: $CI_{imp} = CI_{math} * n$

Indicano i valori implementati in termini di quelli matematici.

Trasformata Fourier Reale

La formula della trasformata di Fourier monodimensionale è la seguente:

$$C_m = \sum_{n=0}^{N-1} C_n e^{\left[\frac{-i2\pi}{N}\right]^{nm}}$$

I valori dei coefficienti restituiti dalla trasformazione reale di tipo implementativo sono uguali al doppio di quelli matematici:

$$RF_{imp} = RF_{math} * 2$$

La formula della trasformazione inversa invece è:

$$C_m = \sum_{n=0}^{N-1} C_n e^{\left[\frac{i2\pi}{N}\right]^{nm}}$$

In questo caso i coefficienti restituiti dalla trasformata inversa di tipo implementativo sono N volte quelli matematici:

$$RI_{imp} = RI_{math} * N$$

Trasformata Fourier Complessa

I valori dei coefficienti di Fourier restituiti dalla trasformazione complessa di tipo implementativo sono uguali a quelli matematici:

$$CF_{imp} = CF_{math}$$

I valori dei coefficienti di Fourier restituiti dalla trasformazione complessa inversa di tipo implementativo sono N volte quelli matematici:

$$CI_{imp} = CI_{math} * N$$

Esempi di scaling

Prendendo come esempio la trasformata complessa, se trasformiamo nel dominio delle frequenze, nessun fattore di scala viene introdotto, mentre se antitrasformiamo il risultato è moltiplicato di un fattore N, dove N è la lunghezza del vettore. Quindi alla fine di ogni operazione di trasformazione è opportuno scalare nuovamente del giusto fattore il segnale ottenuto per riportarlo ai corretti valori.

Nel capitolo seguente andremo a vedere come queste nozioni siano state usate per implementare l'algoritmo di beat detection.

4.5 Implementazione dell'algoritmo di beat detection

Per l'implementazione dell'algoritmo di beat detection sono richiesti diversi passaggi. Per rendere il codice più ordinato, anziché inserirlo nel flipside controller, è stata creata una nuova classe invocata da quest'ultimo attraverso un solo metodo; chiameremo tale classe *Convertitore*. Ciò che succede dentro *Convertitore* è ignoto alla flipside, la quale si aspetta in uscita solo l'*int* che corrisponde al bpm del brano musicale che si sta considerando. Di default, se qualcosa non è andato per il verso giusto dentro *Convertitore*, il valore di quel bpm sarà pari a zero.
Il metodo invocato è il seguente:

```
-(int)convertietagga:(MPMediaItem *)song{...}
```

Il termine "converti" si riferisce al fatto che ogni *MPMediaItem* ricevuto dalla flipside deve essere trasformato da mp3 a PCM prima che qualsiasi operazione di signal processing abbia inizio. Perciò appena il brano entra nella classe, questo viene subito passato a un altro metodo che si occupa di svolgere tale operazione.

4.5.1 Conversione

Per convertire dei file audio bisogna far riferimento a un altro framework, chiamato AVFoundation. Dentro tale libreria sono forniti due strumenti, detti *AVAssetReader* e *AVAssetWriter*, che si occupano rispettivamente di leggere e scrivere file audio. Noi useremo solo il primo, in quanto il secondo serve per scrivere su file, cosa che al momento non ci interessa.

Appena il nostro *MPMediaItem* entra nel metodo di conversione, ne viene ricavato l'indirizzo nella libreria iPod attraverso la proprietà *MPMediaItemPropertyAssetURL*, e con questo viene inizializzato *l'AVAssetReader*; contemporaneamente viene creata anche una nuova variabile di tipo *NSMutableData*, dentro la quale verrà salvato alla fine il file PCM.

Svolte queste operazioni preliminari, bisogna inserire le specifiche del nuovo segnale, attraverso questo codice:

```
AudioChannelLayout channelLayout;
memset(&channelLayout, 0, sizeof(AudioChannelLayout));
channelLayout.mChannelLayoutTag = kAudioChannelLayoutTag_Mono;

NSDictionary *settings = [NSDictionary dictionaryWithObjectsAndKeys:
[NSNumber numberWithInt:kAudioFormatLinearPCM],
AVFormatIDKey,
[NSNumber numberWithFloat:44100.0],
AVSampleRateKey,
[NSNumber numberWithInt:1],
AVNumberOfChannelsKey,
[NSData dataWithBytes:&channelLayout length:sizeof(AudioChannelLayout)],
AVChannelLayoutKey,
[NSNumber numberWithInt:16],
AVLinearPCMBitDepthKey,
```

```
[NSNumber numberWithBool:NO],
AVLinearPCMIsNonInterleaved,
[NSNumber numberWithBool:NO],
AVLinearPCMIsFloatKey,
[NSNumber numberWithBool:NO],
AVLinearPCMIsBigEndianKey,nil];
```

Viene creato un nuovo canale audio, che sarà di tipo *mono*, anzichè stereo, perchè non ci interessa avere il segnale sdoppiato. Le righe successive si occupano di creare un *NSDictionary*, una collezione che funziona per key (come per le mappe in Java), il quale conterrà tutte le caratteristiche del nuovo segnale, come ad esempio la frequenza di campionamento di 44100 Hz.
Infine questo dizionario viene assegnato all'*AVAssetReaderTrackOutput*.
Con i due metodi:

```
[reader addOutput:output];
[reader startReading];
```

è possibile far partire la lettura e conversione del file mp3.

Siccome tale file ci serve dentro un *NSData*, facciamo partire un ciclo che inserisca i byte del nostro brano dentro la variabile creata all'inizio del metodo:

```
while ([reader status] != AVAssetReaderStatusCompleted) {

CMSampleBufferRef buffer = [output copyNextSampleBuffer];
if (buffer == NULL) continue;

CMBlockBufferRef blockBuffer = CMSampleBufferGetDataBuffer(buffer);
size_t size = CMBlockBufferGetDataLength(blockBuffer);
uint8_t *outBytes = malloc(size);
CMBlockBufferCopyDataBytes(blockBuffer, 0, size, outBytes);
CMSampleBufferInvalidate(buffer);
CFRelease(buffer);
[fullsongData appendBytes:outBytes length:size];
free(outBytes);}
```

Essenzialmente si tratta di prendere i byte dal buffer del nostro *AVAssetReaderTrackOutput* e di inserirli di volta in volta dentro la variabile fullsongData, che è un *NSMutableData* e quindi ha la possibilità di aggiungere o rimuovere dati in qualsiasi momento; in questo caso aggiunge i byte di output finché la conversione non è terminata.

Quando il ciclo finisce bisogna collocare ciò che contiene fullsongData dentro un array, condiviso dal resto del codice, che ci servirà per eseguire le operazioni di signal processing:

```
if (reader.status == AVAssetReaderStatusFailed || reader.status ==
AVAssetReaderStatusUnknown){

        conversioneRiuscita=NO;
           return;
    }

if (reader.status == AVAssetReaderStatusCompleted){
        songLength = [fullsongData length];
        songWaveform = (SInt16 *)fullsongData.bytes;
        conversioneRiuscita=YES;
}
```

Per prima cosa controlliamo che la lettura sia andata a buon fine. E' stata creata una variabile BOOL che indica al resto della classe se la conversione è avvenuta con successo. Se non è così il controllo che c'è a valle di questo metodo fallisce e viene restituito un bpm=0.

Se tutto è andato a buon fine aggiungiamo al vettore songWaveform i byte del brano e ne calcoliamo la lunghezza, poi rendiamo positivo il booleano di controllo.

4.5.2 Impostazioni preliminari

L'algoritmo che andremo ad implementare richiede una lunghezza minima dei vettori per poter funzionare. Tale vincolo è dato prima di tutto dalla creazione dei filtri a pettine, dei quali quello a minor bpm (massimo tempo caratteristico) dovrà avere il terzo picco della sua risposta temporale alla posizione 88201 (valore determinato dal file pcm), cosa che rende grandi gli array nel dominio del tempo. La lunghezza minima del pettine, infatti è data da:

$$2 \bullet (60/minBPM) \bullet Fc + 1$$

Che nel nostro caso, dato che Fc=44100 e minBPM=60, da proprio 88201.

Il secondo vincolo è dato dalla FFT, che per funzionare al meglio deve lavorare con dimensioni di vettori pari a una potenza di due.

Abbiamo così creato una variabile globale, detta *DIM*, la quale è la potenza di due subito successiva a 44100, ovvero 65536. *DIM* indicherà la dimensione dei vettori nel dominio delle frequenze, quindi i vettori nel dominio dei tempi avranno dimensione 131072. Questo implica che oltre al controllo sulla conversione, dobbiamo fare anche un controllo sulla lunghezza del brano: se tale lunghezza è inferiore a 2*DIM, il brano non può essere analizzato e viene restituito bpm=0.

Come nota aggiuntiva, possiamo dire che si è provato a svolgere l'analisi del segnale con DIM=44100, quindi prendendo log2n=16 anzichè 17. Il segnale infatti, può non essere lungo come una potenza di due, l'importante è che lo sia l'array di pesi costruito all'inizio. Tuttavia questa scelta ha dato luogo ad alcune imperfezioni. I risultati in termini di bpm sono stati leggermente diversi, e alcuni segnali erano corrotti, come ad esempio questo inviluppo:

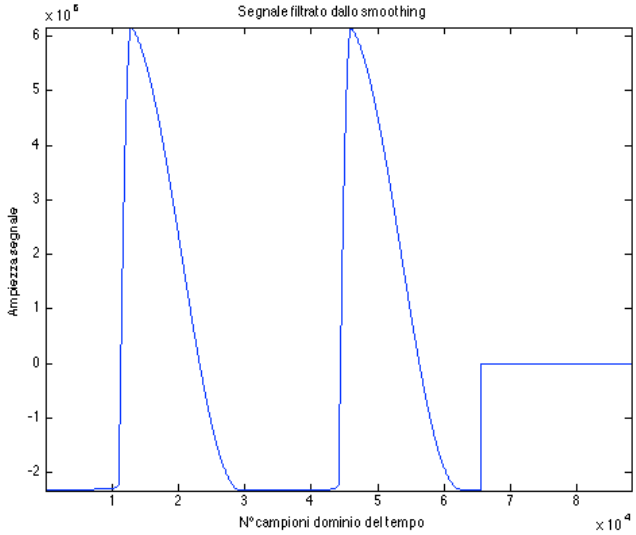


Fig. 4.5.1: Esempio di distorsione del segnale nella fase di smoothing con DIM=44100

Possiamo vedere che una delle tre creste è stata completamente smorzata, e ciò si è ripercosso sul file uscente dalla diffrect, che presenta un picco troppo alto dove non deve esserci, creando poi problemi nel calcolo dell'energia. In questo caso tale picco coincide con il segnale, quindi il calcolo dell'energia non è stato compromesso, ma questo può essere vero nel caso di impulsi regolari e distinti, non in un brano polifonico, dove potrebbe dare luogo a veri e propri errori.

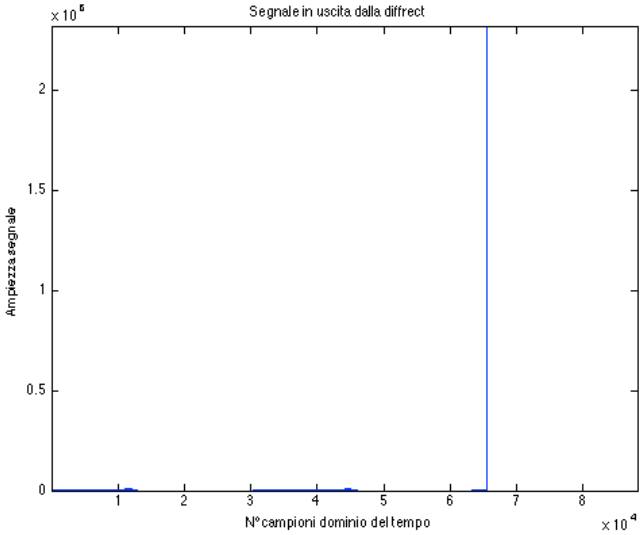


Fig. 4.5.2: Ripercussione della distorsione sul segnale uscente dalla diffrect

D'altra parte, nonostante si sia perso in accuratezza, si è guadagnato in tempi di esecuzione del calcolo, quindi è un'opzione che si potrebbe non scartare del tutto, qualora quest'ultima condizione risultasse più stringente: mantenendo la prima opzione, infatti, i tempi di esecuzione, inclusa la conversione, sono di circa 10s per brano, mentre portando i campioni a 44100 si passa a 6-7s per brano, che per un utente che deve aspettare che l'algoritmo esegua il calcolo su decine di brani può voler dire molto.

Dopo questa piccola osservazione, torniamo a descrivere il codice, il quale necessita a questo punto della giusta allocazione in memoria di alcuni valori.

```
int log2n=log2f((float) 2*DIM);
fft_weights = vDSP_create_fftsetup(log2n, kFFTRadix2);
fftWaveform = calloc(2*DIM, sizeof(float));
fftTransformed = calloc(2*DIM, sizeof(float));
Input.realp = calloc(DIM, sizeof(float));
Input.imagp = calloc(DIM, sizeof(float));
if(fft_weights==NULL|| fftTransformed==NULL ||fftWaveform==NULL||
input.realp==NULL||input.imagp==NULL) return 0;

NSMutableArray *bandlimits=[[NSMutableArray alloc]
initWithObjects:@"0",@"1000",@"2000",@"4000",@"8000",@"16000", nil];
```

In pratica, dobbiamo allocare memoria a tutte le variabili che ci serviranno per la FFT. Assegnato il giusto valore a log2n, creiamo anche l'array di pesi con la funzione *vDSP_create_fftsetup*, e assegnamo ai due vettori nel dominio del tempo lunghezza 2*DIM e quelli nelle frequenze DIM. *fftWaveform* e *fftTransformed* saranno gli array di ingresso/uscita rispettivamente della trasformata diretta ed inversa; *input* invece è un *vDSPSplitComplex*, diviso nella sua parte reale e immaginaria. Dopo aver controllato che siano stati tutti allocati in memoria, creiamo anche un array che contenga i valori delle sei bande.

Dopo queste operazioni preliminari viene invocato il metodo:

```
-(int)calculate:(NSMutableArray*)bandlimits withMaxFreq:
(int)maxfreq{...}
```

che prende in ingresso l'array di bande e la frequenza massima (22050), e restituisce il bpm calcolato. Il valore ottenuto viene poi consegnato al flipside controller (ovvero la classe che gestisce la flipside view).

Si è deciso di dividere il codice in tanti metodi, anziché accorparli in uno unico, così che alla fine di ognuno, tutte le variabili allocate all'interno venissero rilasciate e si potesse liberare memoria più velocemente. Vedremo infatti che non solo lavoreremo con array di 131072 elementi, ma anche con matrici di 131072x6 float, che porteranno il codice ad essere più lento e per evitare anche problemi di memoria, meglio che vengano cancellate il prima possibile anche le matrici che non servono più.

Dentro il metodo *calculate*, essenzialmente, vengono chiamati in ordine i metodi che implementano il *filterbank*, lo *smoothing*, il *diffrect* e il *timecomb*. Quindi per prima cosa andremo a descrivere brevemente come è articolato questo metodo e poi in ordine come sono stati implementati i quattro step.

All'inizio di *calculate,* ponendo *nrange*=2*DIM e *nbands*=6, viene allocata in memoria la prima matrice di *DSPSplitComplex*, che conterrà il risultato del metodo filterbank:

```
DSPSplitComplex *filtered;
filtered=(DSPSplitComplex*)malloc(sizeof(DSPSplitComplex)*DIM);
for (int i=0; i<DIM; i++) {
filtered[i].realp=(float*)malloc(sizeof(float)*nbands);
filtered[i].imagp=(float*)malloc(sizeof(float)*nbands);}
```

Questa matrice e le successive verranno tutte allocate dinamicamente, e non staticamente, perché altrimenti l'algoritmo non viene portato a termine e l'applicazione va in crash solo dopo i primi due metodi.

In sequenza quindi viene allocata la matrice *filtered*, passata al metodo del *filterbank*, che la riempie, poi creata un'altra, *finestred,* che viene riempita dal filtro di *smoothing*. Lo stesso accade per la matrice *diff* passata al metodo *diffrect*, dopo il quale viene trasformata con Fourier e passata al *timecomb*. Per velocizzare i tempi di calcolo la *timecomb* è chiamata più volte nell'intervallo in cui dovrebbe esserci il bpm, così da cercarne l'esatto valore, in questo modo:

```
int d=[self timeComb:dft withAcc:2 withMinBPM:60 withMaxBPM:240
withnrange:nrange withnbands:nbands];

int e=[self timeComb:dft withAcc:0.5 withMinBPM:d-2 withMaxBPM:d+2
withnrange:nrange withnbands:nbands];

int f=[self timeComb:dft withAcc:0.1 withMinBPM:e-0.5 withMaxBPM:e+0.5
withnrange:nrange withnbands:nbands];

int g=[self timeComb:dft withAcc:0.01 withMinBPM:f-0.1 withMaxBPM:f+0.1
withnrange:nrange withnbands:nbands];
```

Dove g è il risultato finale che viene restituito, mentre d-e-f servono per accelerare i tempi di calcolo usando il metodo su degli intervalli sempre più ristretti.
Prima di procedere a descrivere i metodi principali ci soffermiamo un attimo su quello che calcola di volta in volta la FFT, per vedere come sono stati implementati i concetti del capitolo su *accelerate*.

4.5.3 Calcolo FFT

Ogni volta che si avrà la necessità di usare la trasformata diretta o inversa di Fourier, la funzione chiamata è:

```
- (void)computeFFT:(FFTDirection)direction
{
    int nrange=floor(2*DIM);
    int log2n=log2f((float) nrange);

    if (direction==FFT_FORWARD){
    vDSP_ctoz((DSPComplex*)fftWaveform, 2, &input, 1, DIM);
```

Questa funzione prende in ingresso un parametro di tipo *FFTDirection*, che indica se deve essere compiuta una trasformata diretta o inversa. Nel primo caso il segnale inserito dentro *fftWaveform* viene scomposto secondo le regole di packaging dentro la variabile *input*. Su quest'ultimo viene poi calcolata la FFT (diretta o inversa dipende da quanto vale il parametro *direction*). I valori della trasformata sono contenuti in *input* e verranno utilizzati dal codice che ne aveva bisogno senza problemi, in quanto tale variabile è condivisa da tutta la classe. Nel secondo caso invece quando questo metodo verrà chiamato, *input* conterrà già i dati da antitrasformare e vi si metteranno dentro *fftTransformed* i corrispondenti valori nel dominio dei tempi, dopo che l'apposita funzione ne avrà correttamente ristabilito il packaging originario.
Si può notare anche l'intervento della funzione *vDSP_vsmul* la quale moltiplica il vettore *input* per un certo fattore di scala che serve ritornare al giusto valore nell'asse dei tempi, come anticipato nel paragrafo sullo scaling. Tale fattore è 1/2*N: 1/2 per quello introdotto dalla trasformazione diretta e 1/N da quella inversa.

4.5.4 Filterbank

All'inizio di questo metodo viene prima di tutto accorciato il segnale del brano musicale, prendendone solo 2*DIM campioni.
Poi vengono creati due vettori che contengono i sei valori per dividere il segnale frequenziale fra le varie bande. I due array contengono gli indici di sinistra e di destra che i cicli successivi utilizzeranno per la suddivisione. Per ricavare l'esatto indice che corrisponde alla frequenza da cercare viene svolta la seguente proporzione:

$$i = bandlimits[j] \bullet DIM/maxfrequency; \qquad j=0...5$$

Alla fine del metodo viene restituita una matrice contenente in ogni riga solo il segnale appartenente al range della banda corrispondente, e il resto posto a zero.

4.5.5 Smoothing

Il metodo che si occupa dello smoothing per prima cosa crea la finestra che servirà per il filtraggio. Si tratta di una half-window con length=0.4*2*maxfrequency e con funzione:

$$h(t)= cos(i \bullet \pi/lenght/2)\text{^}2$$

Dove *i* è un indice che va da zero a *length*; i valori da *lenght* a DIM sono tutti nulli.

Una volta creata la finestra questa viene trasformata e moltiplicata per ogni banda del segnale, che è stata prima opportunamente rettificata nel dominio dei tempi. Il risultato viene antitrasformato e la funzione ha termine.

4.5.6 Diffrect

Questo metodo si occupa semplicemente di differenziare il segnale; anche qui vengono scorse una per volta le sei bande e viene fatta la differenza tra il termine corrente e quello che lo precede. Se questa differenza è maggiore di zero essa viene salvata in output, altrimenti al suo posto viene salvato un valore nullo.

4.5.7 Timecomb

Questo è l'ultimo metodo utilizzato dall'algoritmo. Prende in ingresso una matrice di complessi, un indice di incremento, e un range di bpm in cui esaminare il segnale.

Ponendo dim=(maxBPM-minBPM)/incremento, parte un ciclo avente come limite massimo tale valore. Poi si crea un array di zeri che sarà il filtro a pettine e verrà ricreato ogni volta con nuovi indici cui porre i picchi così costruiti:

```
int nstep = floor(120/BPM*MAX_FREQUENCY);
for(int a = 0;a<npulses;a++){
    filter[a*nstep+1]=1;
 }
```

Dove BPM è il valore corrente di battiti per minuto, che ad ogni ciclo viene incrementato. Una volta costruito il filtro a pettine viene trasformato in frequenza e moltiplicato per ogni banda del segnale. Di tale numero complesso ne si fa poi il modulo al quadrato calcolandone l'energia con una sommatoria. Se tale energia trovata è superiore al valore attuale (inizialmente zero), vengono aggiornate contemporaneamente le variabili di energia massima e di bpm. Alla fine del ciclo, quest'ultimo valore viene poi restituito dall'intera funzione.

Ecco così riassunto come questo algoritmo sia stato implementato, evitando dettagli in objective-C, che sono praticamente analoghi a quelli di un'applicazione in C o matlab, eccetto la parte sull'uso della FFT di cui abbiamo già parlato. Adesso andiamo a descrivere la seconda parte di questo progetto, che è quello che si occupa del calcolo della cadenza.

4.6 Sensori inerziali

Prima di procedere soffermiamoci un attimo a descrivere in poche parole cosa siano i sensori inerziali.

I sensori inerziali sono dei circuiti integrati opportunamente costruiti per fornire informazioni riguardo alle grandezze fisiche di cui campionano il segnale. Il nostro dispositivo è fornito dei tre seguenti:
-Accelerometro
-Giroscopio
-Magnetometro

L'accelerometro misura l'accelerazione di un oggetto lungo un certo asse. I più comuni ne hanno tre ortogonali. Il giroscopio invece misura la rotazione intorno agli stessi, mentre il magnetometro le linee di campo magnetico terrestre. Quest'ultimo è molto soggetto a interferenze, poiché basta avvicinarvi una calamita che le linee di campo vengono deformate e non riesce più a distinguere i poli terrestri, finché essa non viene rimossa.

L'accelerometro si basa sulla rilevazione dell'inerzia di una massa quando viene sottoposta a un'accelerazione. La massa viene sospesa ad un elemento elastico, mentre il circuito cui è collegata ne campiona lo spostamento rispetto alla struttura fissa del dispositivo.

L'accelerometro di cui è dotato il nostro smartphone ha gli assi orientati in questo modo:

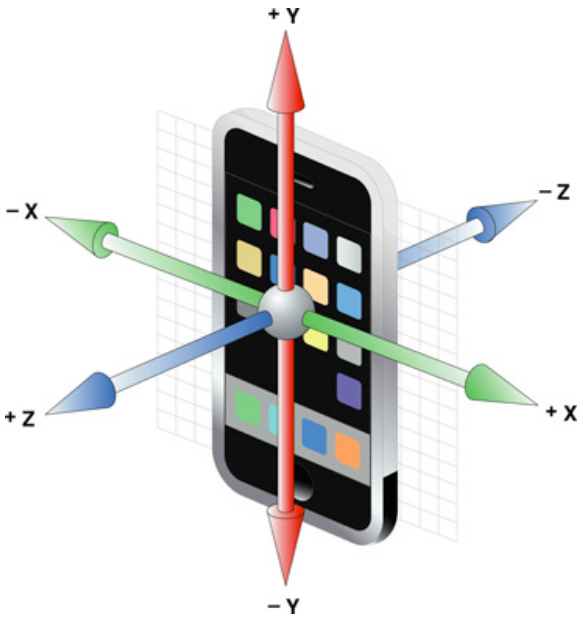


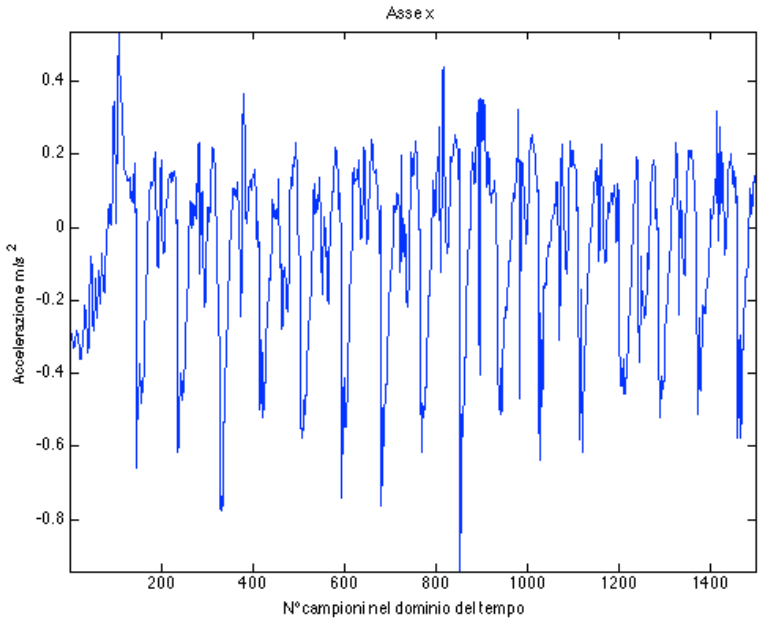
Fig. 4.6.1: Disposizione degli assi dell'accelerometro in un dispositivo iOs

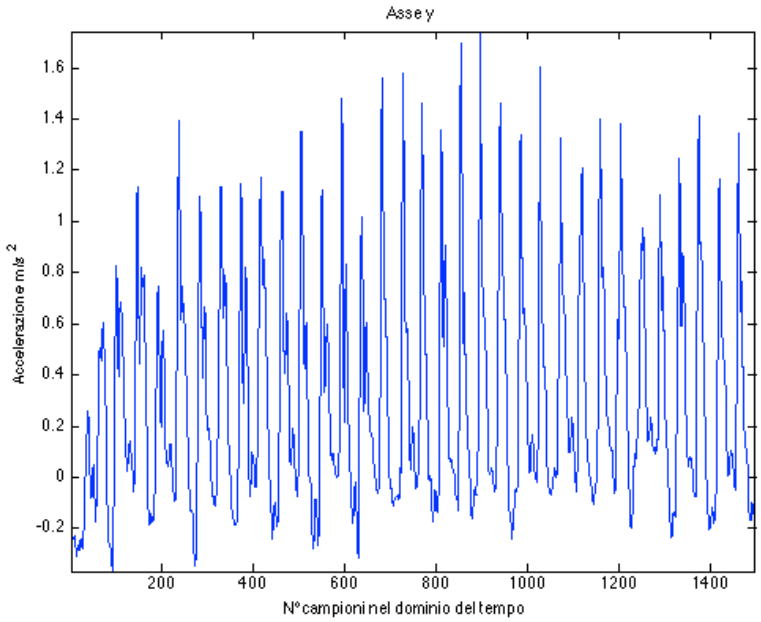
Ipotizzando che il dispositivo rimanga più o meno in posizione verticale durante una sessione di corsa in cui sia stato agganciato a livello della cintura, il segnale che ci interessa è quello proveniente dall'asse y.

4.7 Implementazione dell'algoritmo di step detection

4.7.1 Indipendenza dall'orientamento

Tali considerazioni sono state fatte ipotizzando che il dispositivo iOs sia sempre tenuto in posizione verticale. Nel caso però che un utente decida di usarlo in posizione diversa, dobbiamo rendere l'algoritmo indipendente da essa. Qui di seguito riportiamo i grafici dei segnali provenienti da tutti e tre gli assi, durante una sessione di corsa di 15s, in cui lo smartphone è tenuto solidale al corpo con una fascia da braccio.





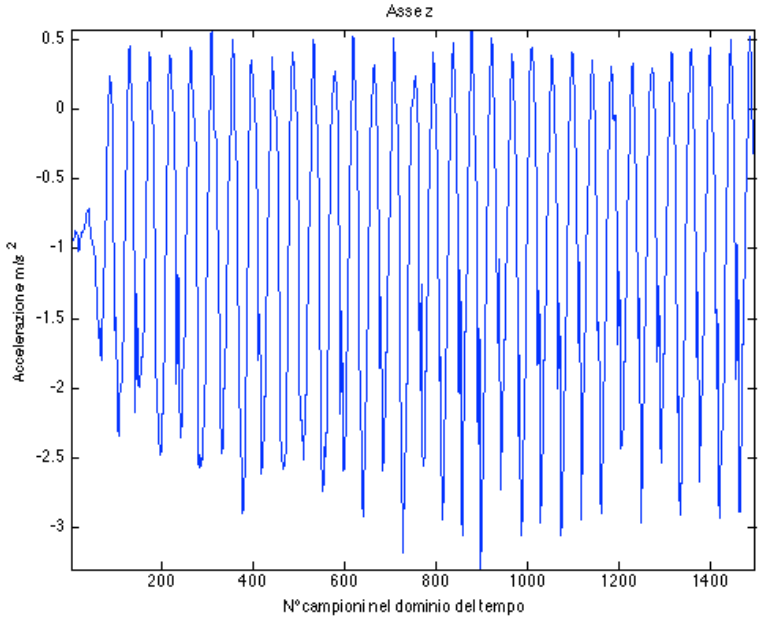


Fig. 4.7.1/2/3: Segnale rispettivamente degli assi x,y,z per un tempo circa di 15s

Possiamo notare come tutti e tre i segnali contengano informazioni utili sulla periodicità del passo, tanto che potremmo calcolare la cadenza da ognuno preso singolarmente. Per rendere il processo indipendente dall'orientamento del dispositivo abbiamo calcolato il modulo dell'accelerazione a partire dalle sue tre componenti, e da questo è stata calcolata la cadenza: in questo modo il risultato non cambia in base all'orientamento del dispositivo. Nell'immagine seguente possiamo vedere il segnale risultante.

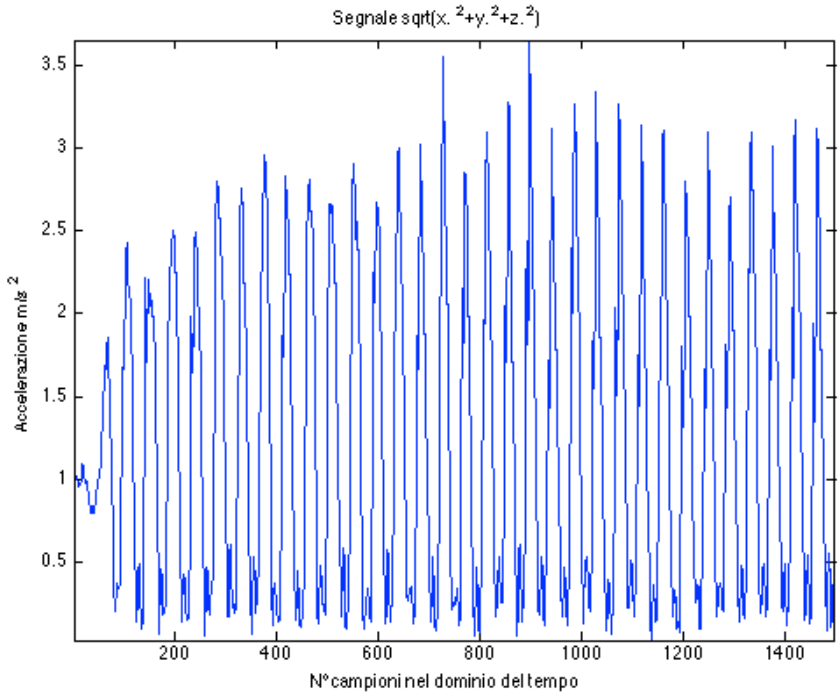


Fig. 4.7.4: Modulo dell'accelerazione

Di questo è stata calcolata la funzione di autocorrelazione, che ha dato luogo a un segnale di questo tipo:

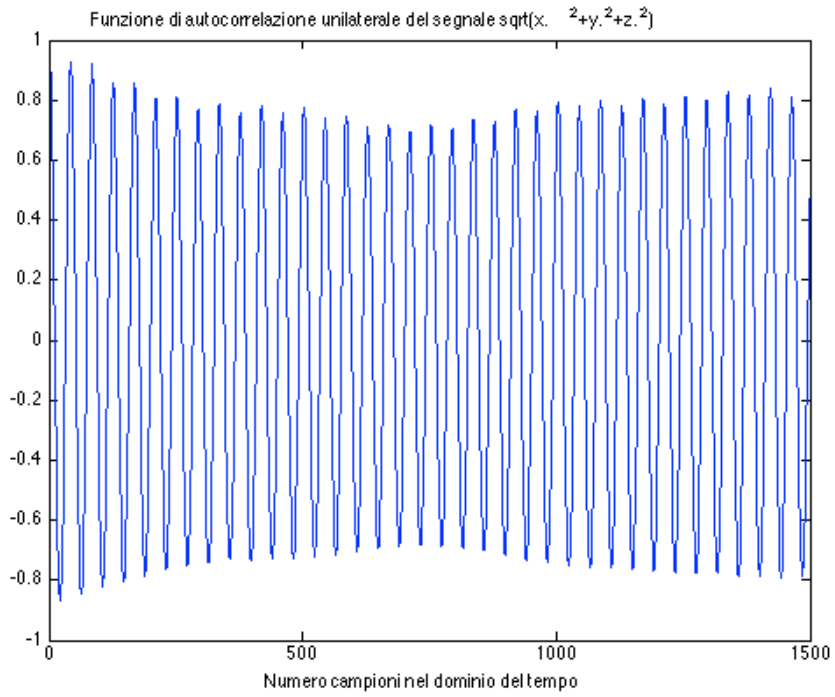


Fig. 4.7.5: Funzione di autocorrelazione del modulo dell'accelerazione in una sessione di 15s

Quello che interessa a noi è il primo picco e sua distanza con l'origine. Facendo diverse prove si è visto che il calcolo del passo da lo stesso risultato in tutte e tre le orientazioni dello smarphone.

4.7.2 Inizializzazione accelerometro ed elaborazione

L'algoritmo di step detection viene invocato dalla main view ogni volta che una canzone finisce o l'utente decide di cambiarla.

Per prima cosa la classe deve inizializzare l'accelerometro. A questo scopo deve importare il delegato, *UIAccelerometerDelegate*, poi lo deve inizializzare, attraverso questi due metodi:

```
[[UIAccelerometer sharedAccelerometer] setUpdateInterval:0.0077];
[[UIAccelerometer sharedAccelerometer] setDelegate:nil];
```

Il primo definisce ogni quanti secondi l'accelerometro debba campionare; in questo modo quindi è settata una frequenza di circa 128 Hz, sopra la quale non è possibile andare. Il secondo metodo serve a fermare l'acquisizione dell'accelerometro, perché ciò che ci interessa è che l'accelerometro acquisisca quando vogliamo noi. Per fare ciò, appena parte la sessione impostiamo il delegato da *nil* a *self*, e ogni 0,0077 secondi la nostra classe eseguirà questo metodo:

Dentro questo metodo vengono prelevati i valori numerici dei tre assi dell'accelerometro e posti dentro gli appositi *NSMutableArray*, che in quanto tali non hanno dimensione fissa, ma aggiungono sempre nuovi dati senza problemi.

```
- (void) accelerometer:(UIAccelerometer *)accelerometer didAccelerate:
(UIAcceleration *)acceleration {

    [yAxis addObject:[NSNumber numberWithFloat:[acceleration y]]];
    [xAxis addObject:[NSNumber numberWithFloat:[acceleration x]]];
    [zAxis addObject:[NSNumber numberWithFloat:[acceleration z]]];

}
```

Quando l'utente decide di cambiare canzone, o questa termina, il segnale campionato dall'accelerometro e la sua durata temporale espressa in secondi vengono elaborati da un metodo di un'altra classe, che è stata chiamata *CalcolaCadenza*.

CalcolaCadenza contiene quattro metodi:

```
- (int)trovaCadenza:(NSMutableArray *)accelerometro withTime:
(float)time{...}
- (NSMutableArray*)peackDet:(NSMutableArray *)xcorr{...}
- (void)computeFFT:(FFTDirection)direction{...}
- (NSMutableArray*)filtra:(NSMutableArray *)y{...}
```

Fra questi possiamo riconoscere *computeFFT*, che è lo stesso metodo presentato nell'algoritmo di beat detection.
trovaCadenza, invece, è il metodo che viene invocato dalla main view e restituisce a quest'ultima i passi per minuto che serviranno per decidere il brano successivo da eseguire. Dentro tale metodo viene per prima cosa invocato il metodo *filtra*, che prende in ingresso il segnale dell'accelerometro e ne restituisce la funzione di autocorrelazione.

Dentro *filtra* viene prima di tutto inizializzata la FFT attraverso l'allocazione in memoria delle variabili necessarie e la creazione dell'array di pesi, poi creata una finestra di Hamming attraverso la funzione *vDSP_hamm_window*.

Successivamente sia il segnale che la finestra vengono trasformati in frequenza e moltiplicati tra di loro, così da ottenere un segnale con meno rumore di quello iniziale.

Per calcolare la funzione di autocorrelazione di quest'ultimo possono essere seguite due strade:

A. Antitrasformare il segnale nel dominio del tempo e calcolare l'autocorrelazione attraverso una serie di cicli *for*.

B. Moltiplicare il segnale, nel dominio delle frequenze, per il suo complesso coniugato, e solo dopo antitrasformare.

Sono state provate entrambe le strade, ma la seconda ha dato luogo a risultati migliori in termini di velocità e di accuratezza.

Una volta ottenuta la funzione di autocorrelazione, per proseguire con i calcoli, ne devono essere ricavati l'indice del primo(coincidente con la posizione zero) e del secondo picco, così da calcolarne la distanza per ottenere il numero di campioni per periodo dominante. Attraverso la funzione *peackDet* viene svolta questa operazione. Il metodo prende in ingresso l'array con l'autocorrelazione e restituisce un array di indici con tutti i picchi, attraverso un semplice algoritmo di peack detection che ne memorizza la posizione lungo il vettore.

Una volta ottenuta la differenza in termini di campioni tra il secondo e il primo picco ricaviamo il numero di passi compiuti dall'utente.

Come illustrato in precedenza la cadenza è data da:

$$c = 60f/n;$$

Siccome la nostra funzione prende in ingresso il tempo e il numero di campioni, possiamo ricavare immediatamente *f*, e di conseguenza la cadenza.

Se ci serve anche sapere il numero di passi, li possiamo facilmente calcolare come:

$$Q = c \bullet S/60.$$

D'altra parte *n* è ricavato dalla funzione di autocorrelazione, ed è sempre un intero. Per questo motivo la risoluzione di *c* è ristretta alla dimensione del passo di quantizzazione, che è la differenza tra due valori successivi di *c*, da cui possiamo ricavare:

$$\Delta(c_n) = (c_n - c_{n+1}).$$

$$\Delta(c_n) = 60f/n - 60f/(n+1) = 60f/(n^2 + n).$$

Sottraendo tale valore a quello di c, abbiamo trovato una stima più accurata della cadenza.

5. Risultati

Nei capitoli precedenti abbiamo illustrato come sia stata realizzata l'applicazione Music Runner, e la teoria che sta dietro agli algoritmi in essa contenuti. Nelle seguenti tabelle sono riportati alcuni dati che mostrano l'accuratezza della nostra applicazione.

N° Traccia	BPM reale	BPM calcolato
1	119	118
2	109	109
3	130	128
4	125	124
5	132	125
6	122	122
7	110	109
8	138	136
9	126	125
10	130	129

Come si può vedere, i brani sopra analizzati hanno bpm praticamente identici. Questo non implica la totale infallibilità dell'algoritmo, in quanto le canzoni analizzate appartengono alla tipologia ottimale su cui fare l'analisi, ovvero brani che presentano pulsazioni date dalla batteria. L'efficienza dell'algoritmo cala con brani troppo strumentali, ma ciò non è stato considerato un grande difetto, perché se l'utente decide di inserirli nell'applicazione, la percentuale di errore cui è affetto il bpm non è rilevante perché dia fastidio alle sessioni di corsa dell'utente.

Per rendere il tutto più completo sono stati presi un centinaio di brani con i bpm noti e sono stati ricavati l'errore relativo e il Mean Squared Error; questi brani sono stati suddivisi in tre categorie:
-Lenti: 0-89
-Medi: 90-119
-Veloci: 120-240

I valori che sono stati ricavati sono:

	Lenti	Medi	Veloci	Tutti
Errore Relativo	5,74%	2,27%	1,57%	2,75%
MSE (bpm)	28,64	13,4	8,86	15

Indicando con MSE il risultato di:

$$MSE = \frac{\sum_{i=1}^{n} (x_i - \hat{x}_i)^2}{n}$$

possiamo vedere dalla tabella che l'errore percentuale è relativamente piccolo per il nostro scopo, e se su un centinaio di brani la deviazione standard è di circa 15 bpm, mentre gli intervalli che consideriamo entro cui debba ricadere la cadenza del passo è di 20 bpm, anche questo valore è accettabile.

Inoltre si può vedere che l'errore relativo più basso è quello dei brani veloci, la qual cosa è molto interessante, poiché i ritmi medi di corsa ricadono all'interno di quel range.

Per quanto riguarda i test sulla corsa si è preso un soggetto che corresse per un minuto e contasse i propri passi mentre usava Music Runner. Questo test è stato ripetuto più volte e i valori della cadenza sono stati salvati per ogni sessione.
Tali esperimenti hanno dato luogo ai seguenti risultati:
Errore relativo= 2,8%
MSE=21,4 passi al minuto
Che sono abbastanza accettabili visto l'uso che deve essere fatto dell'applicazione.

6. Conclusione

Realizzare questa applicazione, nonostante alcune difficoltà, ha dato grandi soddisfazioni perché risulta affidabile, robusta ed adatta allo scopo. Nonostante la nota più dolente sia il tempo di calcolo dei bpm, l'annuncio dei nuovi dispositivi che usciranno sul mercato prevedono un processore molto più potente che renderà il tempo di calcolo di 10s decisamente inferiore. I risultati ottenuti e dichiarati nel capitolo precedente, rendono Music Runner un'applicazione adatta per essere diffusa su appStore, e gli algoritmi in essa presentati sono un ottimo caso di studio su come l'elaborazione dei segnali possa essere ben impiegata su strumenti che fino a qualche anno fa si pensava adatti solo per chiamare o mandare messaggi. Questo risultato è stato raggiunto più o meno in sei mesi di ricerca e sviluppo, in quanto la sua realizzazione non è stata immediata soprattutto per problemi di uso della memoria, che hanno richiesto più tempo, ma che alla fine sono stati risolti con successo.

Tale progetto è ancora in via di sviluppo per poterne ampliare le capacità.

Primo punto da migliorare è la velocità di esecuzione dell'algoritmo di beat detection: già abbiamo notato dei miglioramenti in questo senso prendendo DIM=44100. Si è anche provato con array più piccoli, cambiando il parametro che genera i picchi del filtro a pettine, e si è trovato che alcuni risultati coincidevano con quelli già ottenuti ed altri, invece, i bpm assegnati erano l'esatta metà, quindi urgono nuove soluzioni per trovare un buon compromesso tra velocità e accuratezza.

Altre modifiche che possono essere fatte sono ad esempio di introdurre la modalità "sprint", che se chiamata dall'utente, viene scelta una canzone con bpm maggiore alla cadenza della corsa.

Un'altra modalità che si può inserire, e che sarà l'utente a decidere se impostarla o no, è quella che cambi il brano in riproduzione automaticamente appena l'utente modifica la cadenza della propria corsa, inviando quindi ad ogni intervallo Δt il segnale dell'accelerometro all'algoritmo di step detection, per vedere se il ritmo è cambiato.

Altre modifiche che possono essere apportate sono l'inserimento di un tutorial al primo avvio dell'app, che spieghi all'utente tutte le varie funzioni di cui dispone. La possibilità di inserire le proprie caratteristiche fisiche e l'integrazione con il GPS, per calcolare quanta strada si sia fatta e di conseguenza il numero di calorie bruciate.

Inoltre l'integrazione con iCloud che permetta di sincronizzare con altri dispositivi e di salvare online i propri dati, ma soprattutto i bpm già calcolati.

Queste sono alcune delle idee con cui è possibile espandere quest'applicazione e renderla sempre di più utile a chi ne farà uso.

7. Bibliografia

-[1] ”Estimation of gait cycle characteristics by trunk accelerometry” di Rolf Moe-Nilssen, Jorunn L. Helbostad (The Balance and Gait Unit, Section of Physiotherapy Science, Institute of Public Health and Primary Health Care, University of Bergen, Ulriksdal 8c, N-5009 Bergen, Norway).

-[2] “Tempo and beat analysis of acoustic musical signals” di Eric D. Scheirer (Machine Listening Group, E15-401D MIT Media Laboratory, Cambridge, Massachusetts 02139, 27 Diecembre 1996).

-[3] vDSP_Programming_Guide (documentazione ufficiale iPhone developers)
-CoreAudioOverview (documentazione ufficiale iPhone developers)

7.1 Link Utili

- http://developer.apple.com/devcenter/ios/index.action

- http://developers.facebook.com/

- http://www.devapp.it/

- http://emdief.blogspot.com/2010/04/guida-objective-c-in-italiano-le-basi.html

7.2 Approfondimenti

- Bill Dudney,Chris Adamson, “Sviluppare applicazioni con iPhone SDK”, Apogeo;

- Enrico Amedeo, “Objective-C”, Pocket Apogeo;