SINUS-IT : an evolutionary approach to harmonic analysis

Ulviya Abdulkarimova

UNIVERSITÉ DE STRASBOURG

ÉCOLE DOCTORALE 269

Mathématiques, Sciences de l'Information et de l'Ingénieur

Laboratoire ICUBE (UMR CNRS 7357)

THÈSE présentée par:

Ulviya ABDULKARIMOVA

Soutenue le **2 septembre 2021**

pour obtenir le grade de : **Docteur de l'Université de Strasbourg**

Discipline/Spécialité: Informatique

SINUS-IT : une approche évolutionnaire de l'analyse harmonique

THÈSE dirigée par:	
Pr COLLET Pierre	PR, Université de Strasbourg
Dr ROLANDO Christian	DR, Université de Lille
RAPPORTEURS :	
Dr LAVANANT Helène	MCF HDR, Université de Rouen
Dr LEGRAND Pierrick	MCF HDR, Université de Bordeaux
AUTRES MEMBRES DU JURY :	
Prof. ABARCA-DEL-RIO Rodrigo	PR, University of Concepcion, Chile
Pr IDOUMGHAR Lhassane	PR, Université de Mulhouse
INVITÉS :	
Dr AMHAZ Rabih	ECAM, co-encadrant
Dr MONJID Younes	co-encadrant

Contents

Acknowledgements

This PhD project would not have been possible without the support and guidance of my PhD directors Prof. Pierre Collet and Dr. Christian Rolando. I would like to express my sincere gratitude to them for their supervision throughout my research, continual support, encouragement, and invaluable advice.

Next, I am grateful to Mark Haegelin and Anna Leonteva for their support and collaboration throughout my PhD.

I would also like to thank Dr. Heléne Lavanant and Dr. Pierrick Legrand for accepting to be the reviewers of my thesis, Prof. Rodrigo Abarca-del-rio and Dr. Lhassane Idoumghar for accepting to be part of my jury, and finally my tutors Dr. Rabih Amhaz and Dr. Younes Monjid for all the support they have given me.

Finally, I would like to thank my family for their support and encouragement that allowed me to finish this journey, with a special gratitude to my late father-in-law who used to be my main motivator and who is the reason why I started this journey.

Chapter 1

Introduction

In the last few years, there has been a massive increase in the size and complexity of data sets in many different scientific disciplines as a result of advances in technology. Large and complex data sets are usually found in fields such as medical imaging, physics, material science, remote sensing, *etc.* Working with such data sets has become very challenging since traditional/standard methods are no longer efficient. As a result, new techniques must be used to analyse large and complex data. In this PhD, we propose a new harmonic analysis approach that uses artificial evolution, and experiment it on large noisy simulated and real data coming from Fourier Transform Ion Cyclotron Resonance (FT-ICR) mass spectrometry. Then, we propose many ideas for improvement that tests showed to offer interesting perspectives.

Chemists use mass spectrometry to study molecules and compounds in order to determine their isotopic structures and relative abundance. Compounds may have several coarse isotopic structures and each of these peaks may have smaller peaks around them known as fine isotopic structures (detailed in 9.1). The interest to determine both coarse and fine isotopic structures of the compound is to better be able to determine the chemical composition of the compound. Coarse peaks show how many molecules are present with different additional neutron numbers and for each neutron count, fine peaks show to which atom extra neutrons are attached. There is a lot of interest in determining the fine structure of compounds [1, 2]. In fine structure, peaks are very close to each other in frequency, or with very low amplitude, that makes them very difficult to detect.

FT-ICR MS has two main uses:

Determination: For an unknown compound, FT-ICR is used to find out the atomic composition of the analyzed compound. For this purpose, a s/n (signal to noise) ratio between 10 and 1 is necessary.

Quantification: For a compound of known composition, the interest is to determine how many of each molecule kinds are present. For quantification, a lower s/n ratio (around 0.1) is acceptable to detect sub-peaks, since we already know where the peaks are.

In the first case, the interest is quality (where the peaks are) and in the second case, the interest is quantity (how high are the peaks).

In FT-ICR machines, the signal is analyzed by a computer attached to the machine that performs a Fast Fourier Transform (FFT) which returns the frequency-domain spectrum which then is converted to a mass spectrum. However, the problem with the FFT method that is commonly used inside FT-ICR machines is that it cannot yield the phase parameter correctly due to the fact that the machine does not output imaginary values necessary for FFT to compute the phase. Also, when noise is present in the data, FFT performs poorly because it is an exact mathematical method.

The evolutionary method and algorithm we propose (sinus-it) offers both a better tolerance to noise and the possibility to directly find the phase of the different sines, which (in the case of FT-ICR) can be used to help find the finer peaks of the compound fine structure analysis.

1.1 Current approach of dealing with FT-ICR data

Spectrum analysis is one of the main aspects of signal processing. Many experimental signals in fields such as Nuclear Magnetic Resonance (NMR), Magnetic Resonance Imaging (MRI), Ion Cyclotron Resonance Mass Spectrometer (ICR MS), seismology are made of a sum of (damped) sines, blurred with noise. The size of experiments is continuously growing and for example, a spectrum produced by a current generation ICR MS contains 16 megapoints and may contain more than 10,000 sines. Several methods have been implemented to analyze the spectrum of a discrete signal, the standard one being the FFT algorithm to decompose the signal into a sum of sines to find its harmonic contents [3].

Harmonic analysis based on Fourier Transform has been extensively used, though it presents several limitations:

- The signal needs to be apodized for dealing with the starting and ending wiggles due to its finite size.
- The imperfections in the machines that sample the signal add noise that is preventing perfect mathematical algorithms such as an FFT from performing well. Indeed, a perfect modelling of a noisy signal would lead to a decomposition in an infinite number of sines in order to exactly match the noise. In order to tackle this problem, many mathematically-based approaches try to first filter out the noise in order to improve the signal/noise ratio.
- Another limitation of the FFT method is that where in NMR, quadrature detection yields 2 signals, one real and the other dephased by $\pi/2$ representing the imaginary part, this is not the case in ICR meaning that FFT cannot find the phase on ICR data. For this reason, spectra in magnitude mode are commonly used. Knowing the phase parameter would allow to display the spectra in absorption mode which would improve mass accuracy, mass resolving power and signal to noise ratio.
- Finally, FFT needs long transients to resolve the peaks. The interest is to have samples of smaller size because as noise gets larger, the noise becomes more important than the signal and the signal is diminishing as a result. Therefore, smaller sample size is needed to have a good signal to noise ratio. Most of the times, there are complex

mixtures. In these cases, it it desirable to use shorter transients so that the signal to noise ratio is high enough to detect the peaks.

1.2 Approach proposed in this PhD for dealing with FT-ICR data

We propose to use evolutionary algorithms (EAs) to overcome the limitations of the FFT method. The main objective of this PhD project is to study their performance in comparison to FFT and use FT-ICR as an application to check their validity in a real world case, by determining the coarse and fine isotopic structures of given compound, using smaller samples. The algorithm is tested on simulated data of Substance P (a neuropeptide composed of 11 amino acids, *cf.* `https://en.wikipedia.org/wiki/Substance_P` of chemical formula $C_{63}H_{98}N_{18}O_{13}S$) and on real data coming from the FT-ICR machine.

Evolutionary algorithms are stochastic algorithms based on Darwin's theory of evolution, which states that the fittest individuals have higher chance of surviving and producing offspring of the next generation. They are known to produce human-competitive results to many difficult problems. In this PhD, we propose to use a massively parallel real-coded Genetic Algorithm that runs on GPGPU cards to directly estimate the parameters for sine functions (including phase) that compose the signal of an FT-ICR MS, therefore replacing the need for FFT and avoiding a phase determination step. The algorithm produces a sequence of generations over which a population of potential solutions to the problem evolve. Each generation consists of a population with different potential solutions. In each generation, the solutions are evaluated based on their "fitness" to solve the problem. The solutions with the smallest fitness value (fitness computes the error), have a higher chance of being selected as a parent to share their genes with other members of the population to produce a child solution.

GAs are well suited for the harmonic analysis problem because it is a well defined problem whose characteristics are well known. This means that finely tuned crossover and mutation operators (designed to exploit interesting areas of the search space) can be created, based the on deep understanding of the problem. Results are found by the exploration / exploitation capacity of a very large population, using different levels of parallelism.

In order to study the performance of our algorithm, we first test it on a simulated data since its parameters are known. The obtained results are compared with the FFT method.

1.3 Outline

Part I describes the state of the art, where we will discuss:

- the mathematical background of FFT, FT-ICR mass spectrometry,
- different FFT and non-FFT based techniques for analysing harmonic signals,
- Previous work using GA to analyse harmonic signals is also described, as well as
- a description of evolutionary algorithms.

Part II contains the description and results of our algorithms on simulated and experimental data.

Part III presents the conclusion and persepectives to develop both artificial evolution harmonic analysis and Ion Cyclotron Resonance data analysis.

Part I

State of the art

Chapter 2

State of the art of FFT harmonic analysis

Almost everything in the world can be described using waveforms and these waveforms are just the sum of sinuosoids. Fourier Transform is a method to decompose these waveforms into a sum of simple sinusoids. Fourier analysis is a key tool in areas such as signal and image processing, seismology, medical imaging and in many other areas of science and engineering. In this chapter we discuss the properties and types of Fourier Transform.

2.1 Fourier series

Lets consider a periodic function $f(t)$ with a period T. Then:

$$f(t+T) = f(t) \tag{2.1}$$

which means that a function returns to its original value after a period T. Fourier series is a way to represent a periodic function $f(t)$ as sum of simpler functions, known as harmonic components. If we assume a T_0 periodic function, its Fourier series is given as an infinite sum of sine and cosine functions:

$$f(t) = \frac{a_0}{2} + \sum_{n=1}^{\infty}[a_n \cos(2\pi n f_0 t) + b_n \sin(2\pi n f_0 t)] \tag{2.2}$$

where a_n and b_n are called Fourier coefficients and f_0 is the frequency that is related to the period by $f_0 = \frac{1}{T_0}$. The unknown terms of the equation 2.2 can be found as:

$$a_0 = \frac{1}{T_0}\int_{-\frac{T_0}{2}}^{\frac{T_0}{2}} f(t)dt \tag{2.3}$$

$$a_n = \frac{2}{T_0}\int_{-\frac{T_0}{2}}^{\frac{T_0}{2}} f(t)\cos(2\pi n f_0 t)dt \tag{2.4}$$

$$b_n = \frac{2}{T_0} \int_{-\frac{T_0}{2}}^{\frac{T_0}{2}} f(t) \sin(2\pi n f_0 t) dt \tag{2.5}$$

with $n = 1, 2, 3, ...$

Using the identities:

$$\cos t = \frac{e^{it} + e^{-it}}{2} \tag{2.6}$$

and:

$$\sin t = \frac{e^{it} - e^{-it}}{2i} \tag{2.7}$$

we obtain the complex form of Fourier Series:

$$f(t) = \sum_{n=-\infty}^{\infty} c_n e^{i2\pi n f_0 t} \tag{2.8}$$

where the complex Fourier coefficient c_n can be found by:

$$c_n = \frac{1}{T_0} \int_{-\frac{T_0}{2}}^{\frac{T_0}{2}} f(t) e^{-i2\pi n f_0 t} dt \tag{2.9}$$

The complex form of Fourier Series is more convenient mathematically as there is only one coefficient to be found [3].

2.2 Fourier Transform

The Fourier Transform (FT) is the extension of Fourier Series to non-periodic functions. It is a tool that is used to transform signals from time domain into frequency domain. The FT of a function $x(t)$ is defined as:

$$X(f) = \int_{-\infty}^{\infty} x(t) e^{-i2\pi f t} dt \tag{2.10}$$

The result is a function of frequency f. This frequency domain representation is often called the frequency spectrum of x.

$x(t)$ can be obtained from $X(f)$ using Inverse Fourier transform:

$$x(t) = \int_{-\infty}^{\infty} X(f) e^{i2\pi f t} df \tag{2.11}$$

The FT for the sine $\sin(at)$ and cosine $\cos(at)$ functions with variable f is given by:

$$\mathscr{F}(\sin(at)) = \frac{\delta(f - \frac{a}{2\pi}) - \delta(f + \frac{a}{2\pi})}{2i} \tag{2.12}$$

and:

$$\mathscr{F}(\cos(at)) = \frac{\delta(f - \frac{a}{2\pi}) + \delta(f + \frac{a}{2\pi})}{2} \tag{2.13}$$

where $\mathscr{F}$ is the notation for FT and δ is the Dirac Delta function.

When a phase shift ϕ is added to the sine signal, this is equivalent to:

$$\sin(at + \phi) = \sin(at)\cos(\phi) + \cos(at)\sin(\phi) \tag{2.14}$$

Real and imaginary parts are obtained in the FT centered at $\delta(f - \frac{a}{2\pi})$ and $\delta(f + \frac{a}{2\pi})$ with the right angle from the real and imaginary part. It can be noticed that the signal is symmetric about 0 and that there is the same information on the positive $(0, \infty)$ and negative part $(0, -\infty)$.

The FT of a rectangle rect(ax) function centered around zero is given by:

$$\mathscr{F}(\text{rect}(x)) = \Big(\frac{1}{|a|}\Big)\text{sinc}\Big(\frac{f}{a}\Big) \tag{2.15}$$

where sinc(x) = $\sin(x)/x$ gives the oscillations. The smaller the rectangle, the more important are the oscillations as they are proportional to $1/|a|$.

The FT of a two-sided decaying exponential function is a Lorentzian function given by:

$$\mathscr{F}(e^{-a|t|}) = \frac{2a}{a^2 + 4\pi^2 f^2} \tag{2.16}$$

The convolution of two functions $f(x)$ and $g(x)$ is given by:

$$h(z) = \int_{-\infty}^{\infty} f(x)g(x - z)dx \tag{2.17}$$

Then, the FT of the convolution of the functions $f(x)$ and $g(x)$ is the product of their Fourier transforms, given by:

$$\mathscr{F}(h) = \mathscr{F}(f) \times \mathscr{F}(g) \tag{2.18}$$

A real signal (the negative part can be created by symmetry) is the convolution of a sine function, a rectangular window and a damping function which gives the characteristics of a signal after FT [3].

2.2.1 Discrete Fourier Transform (DFT)

Given a continuous-time signal, a finite number of samples can be taken from the original signal to determine the frequency content of the time-domain signal. This procedure is called Discrete Fourier Transform (DFT), which is a type of FT with sampling at regular intervals of N temporal signals. DFT is used to find a set of sinusoids that can be summed up to produce a discrete-time signal.

The equation of DFT is given by:

$$F_k = \sum_{n=0}^{N-1} y_n e^{-i\frac{2\pi}{N}kn} \tag{2.19}$$

where F_k are the Fourier coefficients with $k = 0, ..., N-1$.

The inverse DFT is given by:

$$y_k = \frac{1}{N} \sum_{n=0}^{N-1} F_n e^{i\frac{2\pi}{N}kn} \tag{2.20}$$

where y_k are the data points with $k = 0, ..., N-1$.

In Figure 2.1 a signal in time domain is shown. Figure 2.2 illustrates the result of DFT, given in the frequency domain. The plot in Figure 2.2 is called a spectrum plot which includes the frequencies on the x-axis and amplitude of the signal in the y-axis.

Let $\mathbf{y} = (y_0, y_1, ..., y_{N-1})$ be a data vector and let $\mathbf{F} = (F_0, F_1, ..., F_{N-1})$ be the Fourier coefficients obtained as a result of DFT. Using equation 2.19, we can obtain the values of $F_0, F_1, ..., F_{N-1}$. If we let $\omega_N = e^{-i\frac{2\pi}{N}}$, then we can write the DFT matrix as:

$$\begin{pmatrix} F_0 \\ F_1 \\ F_2 \\ \vdots \\ F_{N-1} \end{pmatrix} = \begin{pmatrix} 1 & 1 & 1 & \dots & 1 \\ 1 & \omega_N^1 & \omega_N^2 & \dots & \omega_N^{N-1} \\ 1 & \omega_N^2 & \omega_N^4 & \dots & \omega_N^{2(N-1)} \\ \vdots & \vdots & \vdots & & \vdots \\ 1 & \omega_N^{N-1} & \omega_N^{2(N-1)} & \dots & \omega_N^{(N-1)^2} \end{pmatrix} \begin{pmatrix} y_0 \\ y_1 \\ y_2 \\ \vdots \\ y_{N-1} \end{pmatrix} \tag{2.21}$$

To calculate each Fourier coefficient, we add up N different values and thus we perform N^2 operations in total. For a large N, this is very time consuming. Fundamental to DFT is the Fast Fourier Transform (FFT) or Cooley-Tukey algorithm which divides the signals into smaller signals and computes the DFT in a time proportional to $N \log_2(N)$ where N is the number of points. These smaller DFTs are then added together to get the DFT of the big signal.

The resulting frequencies can be plotted in a spectrum plot with frequencies on the x-axis and amplitude on the y-axis.

We take N sample points over a time period T, with the distance between two neighbouring points equal to and denoted by Δt. Then, $\frac{1}{\Delta t} = f_s$, where f_s is the sampling frequency, which is the number of samples obtained per second. We also define the time period T as $T = N\Delta t$ which implies that $T = N/f_s$.

There is also Nyquist frequency which is given by $f_{\text{Nyquist}} = f_s/2$. It is the maximum frequency that can be recovered at a given sampling rate. If the frequencies are smaller than the Nyquist frequency, then they can be accurately recovered. However, if the frequencies are larger than the Nyquist frequency, then they will just be the mirror images of the lower half of the spectrum. In this case, aliasing will occur which can be avoided by making sure that all the frequencies are less than the Nyquist frequency.

In the spectrum plot, the frequencies are equally spaced with $\Delta f = \frac{1}{T} = \frac{f_s}{N} = \frac{2f_{\text{Nyquist}}}{N}$, where Δf is called the frequency resolution or width of the frequency bin for a fixed sampling

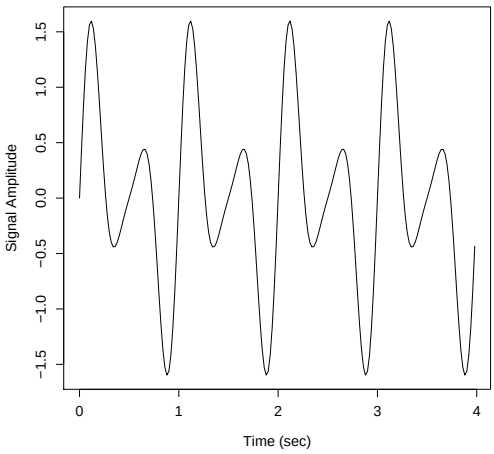

Figure 2.1: Time domain

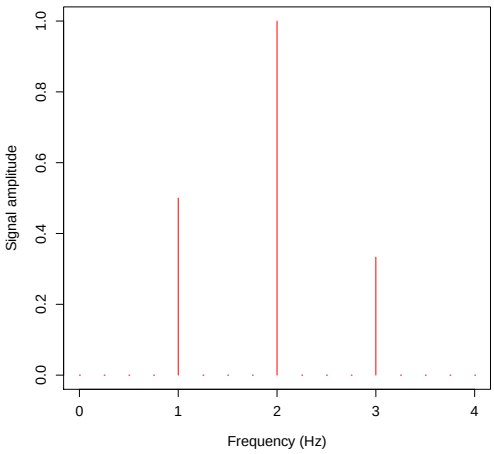

Figure 2.2: Frequency domain

frequency. So, the x-axis will be integer multiples of Δf with the highest frequency being $\frac{N}{2}\Delta f$ [4].

2.3 DFT Errors

Since DFT is just an approximation using discrete values of the signal at equally spaced instants of time, the question is: how many samples are needed to determine the signal? There are two main errors associated with DFT, that are aliasing and leakage.

2.3.1 Aliasing

The aliasing effect arises when the sampling rate is too low and thus the original signal cannot be captured correctly as the high frequencies appear as lower frequencies. In order to avoid aliasing, the sampling must be done at or above the Nyquist sampling rate. According to the Nyquist sampling theorem, the sampling frequency needs to be at least twice the highest frequency component of the signal:

$$f_s \geq 2f_c \tag{2.22}$$

where f_s is the sampling frequency and f_c is the highest frequency component of the signal. $2f_c$ is known as the Nyquist sampling rate.

If Δt is too large, f_s will be too small and thus high frequency components of the signal may be missed. Also, if the period is too short and N is too small, the low frequency components of the signal may be missed.

Aliasing is illustrated in Figure 2.3. The signal is given as the black curve with frequency 0.5 Hz. According to the Nyquist sampling theorem, the sampling frequency has to be at least 1 Hz in this case in order to be able to determine the signal. When the sampling frequency is lower than 1 Hz, for example 0.5 Hz, there are not enough samples to determine the peaks of the signal as seen in Figure 2.3. The red line shows that the signal may be thought as a linear signal when there are too small number of samples. If the sampling frequency is 1 Hz and higher, we can see that there are enough samples to determine the peaks of the signal. Therefore, when sampling frequency is lower than twice of the highest frequency component, this would give us wrong information about the original signal [5].

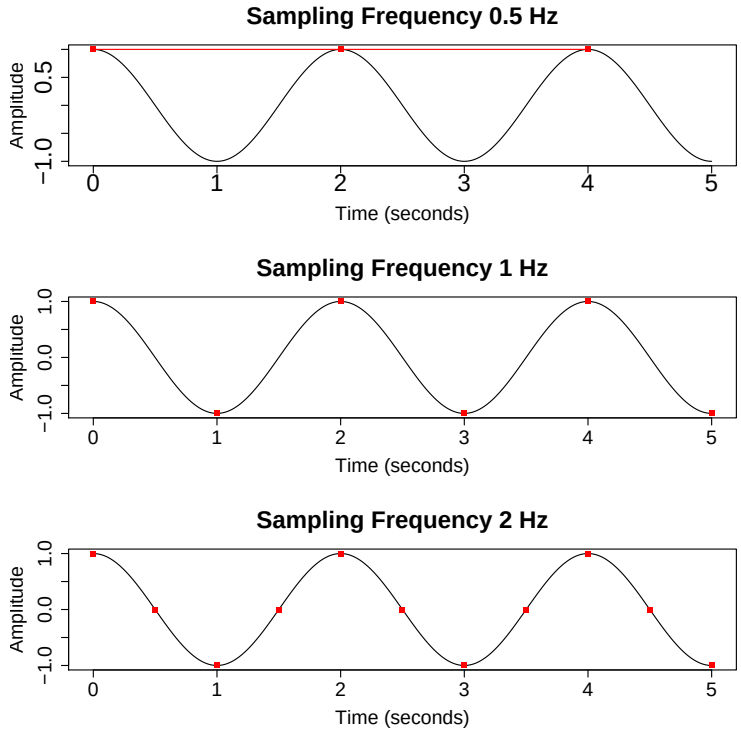

Figure 2.3: Aliasing effect

2.3.2 Spectral Leakage

Spectral leakage happens when the frequency of the input signal is not an exact integer multiple of frequency resolution, in other words, when the number of periods in a signal is not an integer. In this case, there are discontinuities in the signal due to the not-smooth connection of one waveform to the previous waveform. As a result, there will be leakage of

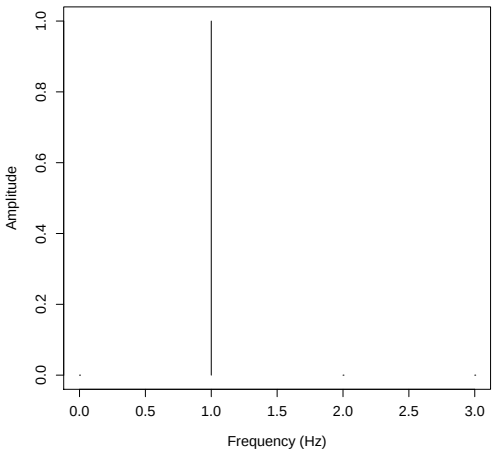

Figure 2.4: No spectral leakage

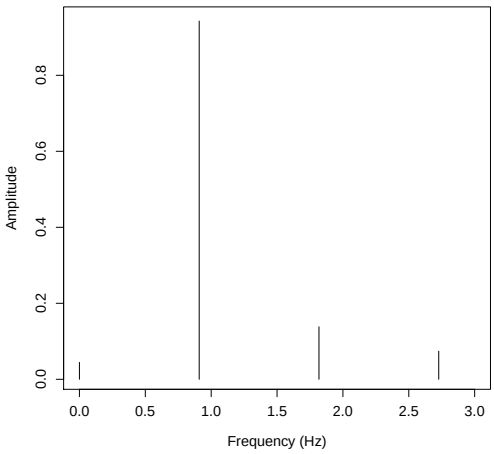

Figure 2.5: Spectral Leakage

energy from one frequency to other frequencies around it. Spectral leakage is problematic because it may not be distinguished from noise.

Lets consider the following simple example. Let the input signal be $\sin(2\pi x)$, where the frequency is 1Hz. Let the sampling frequency be 50Hz and time period 1.1 seconds. The spectrum plot of this is shown in Figure 2.5. The plot in Figure 2.4 shows the spectrum plot if the time period was an integer, 1 second. We see that in this case, we get exactly the frequency we expected. However, in Figure 2.5, the highest frequency is close to 1 and we see some smaller peaks at the frequency bins around. This is called spectral leakage, as the energy measured for the frequency 1Hz is spread onto the frequency bins in the neighbourhood.

One of the approaches to minimize spectral leakage is windowing. Windowing function is used to force the signal to 0 at the end of the waveform to avoid discontinuities. There are several window functions such as Uniform, Hann, Flat Top, etc. Correct selection the window function is very important [5].

Chapter 3

Fourier Transform Ion Cyclotron Resonance Mass Spectrometer (FT-ICR MS)

FT-ICR is a type of mass analyzer that is used to measure the mass to charge ratio (m/z) of ionized molecules based on their cyclotron frequency inside electro-magnetic fields, in order to determine the chemical composition of molecules (Figure 3.1). FT-ICR was introduced by Comisarow & Marshall in 1974 [6]. It offers the highest resolving power (defined as the peak position divided by its width) nowadays up to 10 million and mass accuracy compared to all other types of mass spectrometers.

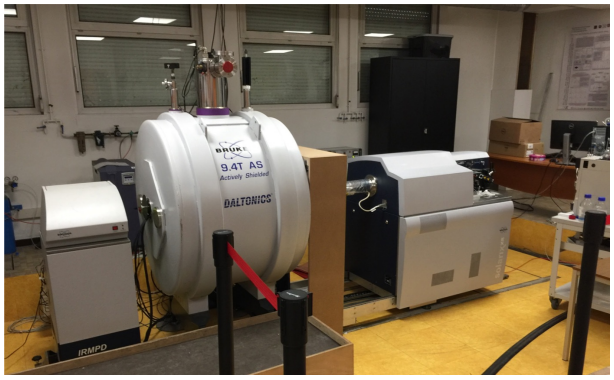

Figure 3.1: FT-ICR device (photo taken at the MSAP laboratory, University of Lille)

This device is a central tool for proteomics, petroleomics, metabolomics analysis. FT-ICR is comprised of an ion trap, a magnet for creating a magnetic field within the trap, a system for exciting the ion motion and detecting the signal, as well as data system to record

the signal, to store and to transform it into mass spectra. The main part of an FT-ICR mass spectrometer is the ICR (Ion Cyclotron Resonance) cell, in which ions are trapped by a magnetic field produced on a circular trajectory. In order to obtain a measurable signal from ions, their synchronous cyclotron motion is excited by applying a radio frequency (RF) voltage [7]. The schematic of the FT-ICR device is given in Figure 3.2.

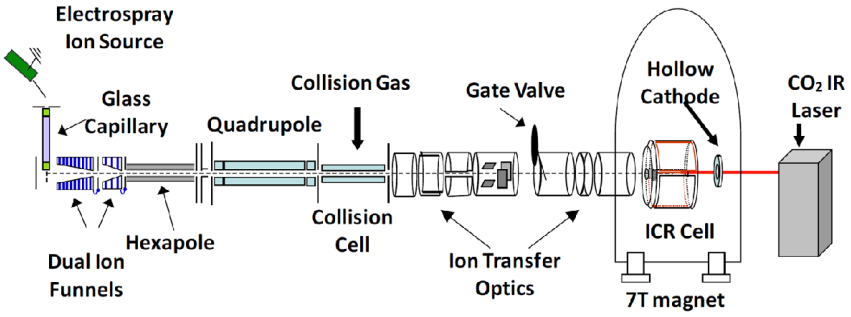

Figure 3.2: FT-ICR schematic Bruker Apex (Adapted from [8])

Ion cyclotron motion in the electro-magnetic field

The force acting on a moving ion with charge q and velocity v is given by the following equation:

$$\mathbf{F} = q\mathbf{v} \times \mathbf{B} \tag{3.1}$$

where $\mathbf{B}$ is the strength of the magnetic field.

This force is called the Lorentz force and is perpendicular to the velocity of the ion and the magnetic field. Figure 3.3 shows how the Lorentz force acts on positive and negative ions in a uniform magnetic field. The ions move in circular motion as a result of the Lorentz force.

Since $F = ma$, $a = v^2/r$ and $v = \omega r$, we obtain the following equality:

$$F = m\frac{(\omega r)^2}{r} = q\omega r B \tag{3.2}$$

Therefore, the cyclotron motion of an ion is given by the following equation:

$$\omega = \frac{qB}{m} \tag{3.3}$$

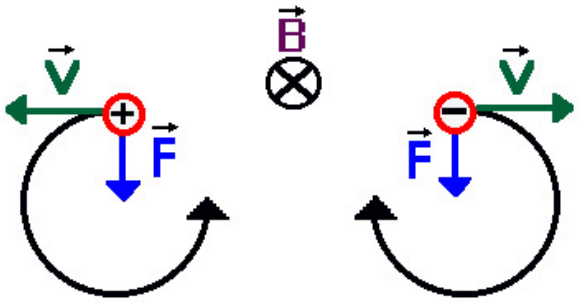

Figure 3.3: Lorentz force action on positive and negative ions (Adapted from [9])

where ω is the cyclotron (or angular) frequency. The cyclotron frequency of the ions depend on the ions' mass to charge ratio and magnetic strength. Therefore, ions with different mass to charge ratios oscillate at different frequencies. By detecting those frequencies, the mass of the ions can be estimated. The cyclotron frequency of an ion hence does not depend on its velocity and kinetic energy. Due to this characteristic, FT-ICR has highest resolving power compared to the other types of mass spectrometers [10].

Excitation of ions and detection of signal

Before excitation, the ions have a low kinetic energy and a small cyclotron radius (about $1mm$) (Figure 3.4 left). Thus, at this stage, signal cannot be detected. During the excitation, the ions are excited to larger cyclotron radii and thus are closer to the detection plates. Prior to the detection of the signal, ions are excited but not simultaneously because the excitation happens when the cyclotron and the excitation waveform frequencies match. Therefore, ions with different frequencies are excited at different times [10]. Ions with the same mass to charge ratio are excited together and thus form an "ion packet" and undergo cyclotron motion (Figure 3.4 middle). Ions that are off resonance with the frequency applied, do not get energy and thus remain near the center of the cell. After excitation, when RF voltage is turned off, the ions remain at their larger radius state that can be detected (Figure 3.4 right).

FT-ICR signal

The recorded signal in FT-ICR is called a transient or free induction decay (FID). The signal consists of a sum of sinusoidal waves with the intensities damped with time, given by the following equation:

$$\mathrm{S}[t] = \sum_{k=1}^{K} A_k exp(-t/\tau_k) \cdot \cos(\omega_k \cdot t + \phi_k) \tag{3.4}$$

where n is the sample number, A is the amplitude, ω is the angular frequency, ϕ is the phase, K is the number of sines and τ is the damping factor that is caused by the collision

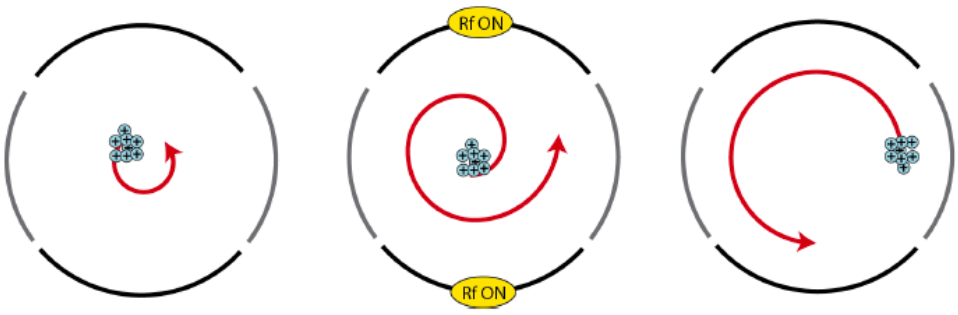

Figure 3.4: Excitation of ions. Left: before excitation; Middle: During excitation; Right: After excitation. (Adapted from [11])

of ions [10].

A finite number of points are taken from the transient and the sample size is determined before the acquisition. Sampling frequency is determined by the software according to the Nyquist theorem. Once the sample size and sampling frequency are determined, the acquisition period can be determined by:

$$T = \frac{N}{f_s} \tag{3.5}$$

where N is the sample size.

The time domain signal is then transformed in the frequency domain by a Fast Fourier Transform algorithm. The process is shown in Figure 3.5.

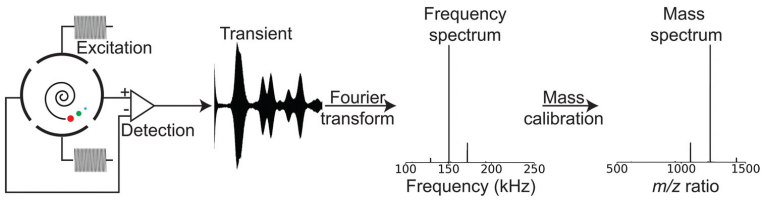

Figure 3.5: Mass spectrum acquisition and processing using an FT-ICR mass spectrometer (adapted from [12])

A single spectrum comprises usually from 128k points to 16M points. Peaks in frequency domain are then transformed in the desired m/z domain which is the inverse of the frequency.

Once excitation is done, time delay (in milliseconds) is observed before detection. This causes a phase shift which can be as large as tens of thousands in radians [13]. As a result, there is a quadratic dependency of the phase with the frequency. This phase shift must be corrected in order to take the advantage of the absorption mode.

Phase correction

The output of FT are complex valued numbers:

$$F(\omega) = Re(\omega) + iIm(\omega) \tag{3.6}$$

where ω is the angular frequency. The magnitude and phase components of the transient can be obtained from the FT output using the equations below:

$$M(\omega) = |F(\omega)| = \sqrt{(Re(\omega))^2 + (Im(\omega))^2} \tag{3.7}$$

$$\phi(\omega) = \arctan\left(\frac{Re(\omega)}{Im(\omega)}\right) \tag{3.8}$$

where $M(\omega)$ is the magnitude mode spectrum and $\phi(\omega)$ is the phase mode spectrum. Magnitude mode spectrum is most commonly used as the phase component is difficult to determine due to varying phase shift. However, it has been known for years that if the spectrum is displayed in absorption mode, this would result in improved mass resolving power (twice as much as in magnitude-mode), mass accuracy and signal to noise ratio [14]. Magnitude mode and absorption mode are two different ways of viewing the FT-ICR data. Processing the data in absorption mode will help to detect more peaks in the spectrum [15].

The phase shift function as given in [10] is as follows:

$$\phi(\omega) = -\frac{\omega_t^2}{2R} + \Big(\frac{\omega_{final}}{R} + t_{delay}\Big)\omega_t - \frac{\omega_0^2}{2R} + \phi_0 \tag{3.9}$$

Figure 3.6 shows the comparison of the magnitude mode and absorption mode. As we can see, in absorption mode, the peak width is narrower which means that the mass resolution is higher (by a factor of $1.4 - 2$) compared to the magnitude mode. More peaks can be resolved in absorption mode that cannot be resolved in the magnitude mode. We can see some extra peaks at the right end of the figure. The more peaks obtain the better information is obtained about the component.

Several methods have been developed to correct the phase and to produce absorption mode spectra. These methods mainly try to find a quadratic phase correction function to correct the phase in spectral data. The method of Qi *et al.* [17] is manually intensive which limits its use. The method of Beu *et al.* [18] requires a special device to simultaneously excite the ions. The method of Xian *et al.* [15] is based on a simplified version of the phase correction function and thus does not take into account the initial phase and other important related effects.

In [13], Kilgour used genetic algorithm techniques to find a quadratic phase correction function and obtained highly improved processing performance compared to previous methods.

3.1 Data Processing in FT-ICR

FT-ICR MS includes Nyquist sampling, Fast Fourier Transform, zero-filling, apodization, oversampling, as well as several non-FT methods [14].

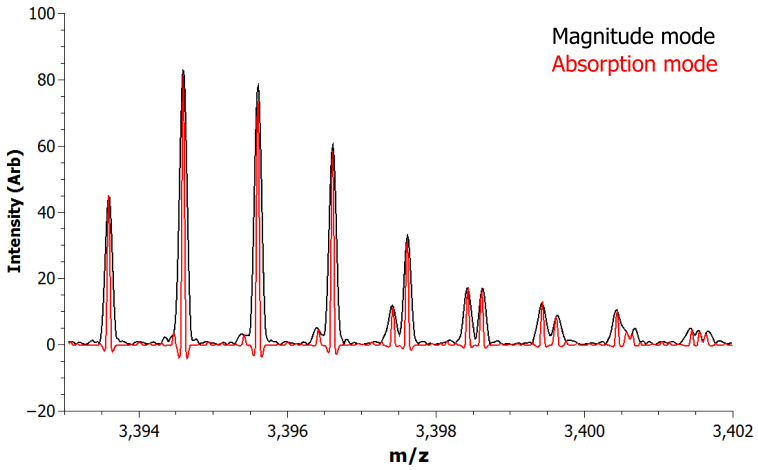

Figure 3.6: Magnitude mode vs absorption mode. (Adapted from [16])

In some applications of FT-ICR transient, several operations such as apodization and zero-filling are needed to be applied for improving the quality of the data before applying FT. Since discrete FT is just an approximation, it is prone to errors due discrete sampling and finite length of data.

Convolution

Convolution is a mathematical operation that is done on two functions and is given by following equation:

$$f \bigotimes g = \int f(u)g(t-u)du \tag{3.10}$$

According to the convolution theorem, convolution of two time domain functions $f(t)$ and $g(t)$ is the product of their Fourier transforms, given by:

$$\mathscr{F}(f \bigotimes g) = \mathscr{F}(f) \times \mathscr{F}(g) \tag{3.11}$$

Convolution is very useful as it simplifies the calculation by converting integration in time domain into multiplication in frequency domain.

Zero-filling

When the number of samples are too few, this would lead to poor resolution of the frequency-domain spectrum. Zero-filling is a technique in which zeros are added to the end of the transient to increase the number of data points. This has only a cosmetic effect on the spectrum since adding zeros does not introduce any new information. Therefore, this technique

is performed before applying FT in order to get better display of the spectrum [19]. However, this method causes discontinuity of the data and hence introduces unwanted signals called artifacts. This problem can be solved by acquiring longer transients or multiplying the transient with an "apodization" function.

Apodization

One of the applications of convolution theorem is apodization. Apodization means "removing the foot". An apodization function is a mathematical function that is used to remove the discontinuities at the beginning and end of the signal. Table 3.1 shows commonly used apodization functions.

Name	Magnitude mode	Absorption mode
Rectangular	1	1
Cosine-bell	$\cos(\pi n/N - \pi/2)$	$\cos(\pi$ n/2N)
Hamming	0.54-0.46cos(2 π n/N)	0.54+0.46cos(2π n/N)
Hann	0.5-0.5cos(2π n/N)	0.5+0.5cos(π n/N)
Gaussian	exp(-1/2((n-N/2)/kN/2)2)	exp(-1/2((n/2)/kN/2)2)
Blackman	0.42-0.5cos(2π n /N)+0.08cos(4π n/N)	0.42+0.5cos(π n/N)+0.08cos(2π n/N)
Triangle	$1 - \lvert 1 - 2n/N \rvert$	1-n/N

Table 3.1: List of commonly used apodization functions (Table adapted from [10])

Figure 3.7 shows the apodized *vs* unapodized spectrum. As we can see, in the unapodized spectrum, there are artifact peaks at the beginning and end of the signal which carry no useful information. Therefore, the signal needs to be multiplied by an apodization function to remove those peaks. This method is applied to improve the resolution of the frequency domain spectrum.

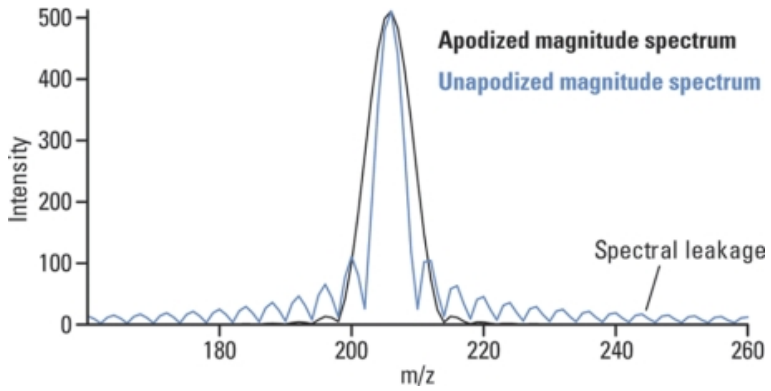

Figure 3.7: Apodized vs non-apodized simulated spectral peak (Adapted from [19])

3.2 Isotopic structures

As discussed earlier, mass spectrometers are used to determine the chemical composition of molecules based on their mass. It produces a mass spectrum which is usually plotted as mass to charge ratio *vs* intensity (or relative abundance %). The largest peak in mass spectrum is called the most abundant peak. This is the base peak with a value 100%. All other peaks are given as the percentage of this base peak. The molecular weight peak is called the molecular ion peak and and it is denoted by $M+$. There are also $M+1$, $M+2$, and so on peaks because of other isotopes. These peaks are referred to as coarse isotopic distributions. Each distribution except the monoisotopic peak has several smaller peaks around them which are known as fine isotopic structures.

For example, the following elements have significant role in nature and thus in organic chemistry: H, C, N, O, and S. Each of these elements has at least two isotopes. The most abundant ones are ^{12}C, ^{16}O, ^{14}N, ^{32}S, and ^{1}H as shown in Table 3.2. The peaks with minor abundances become visible in the spectrum with an increase in the number of atoms. For a known compound, major and minor abundances can be calculated using the rules of probability theory and relative abundance of the involved molecules [20].

Element	Isotopes	Exact Mass	Natural abundance
Hydrogen	^{1}H	1.00783	99.985
	^{2}H	2.01410	0.015
Carbon	^{12}C	12.00000	98.892
	^{13}C	13.00336	1.108
Nitrogen	^{14}N	14.00307	99.634
	^{15}N	15.00011	0.366
Oxygen	^{16}O	15.9949	99.763
	^{17}O	16.9991	0.037
	^{18}O	17.9992	0.20
Sulfur	^{32}S	31.9721	94.78
	^{33}S	32.9715	0.79
	^{34}S	33.9679	4.43

Table 3.2: Exact mass of natural isotopes

3.3 Mass accuracy

Mass accuracy is very important in mass spectrometry in order to correctly identify the chemical composition of ions. A small shift in mass can make a huge difference in determining the compound. Mass accuracy is calculated using the equation below:

$$\text{Mass accuracy} = \frac{m_{\text{experimental}} - m_{\text{theoretical}}}{m_{\text{theoretical}}} 10^6 \text{ ppm (parts per million)} \tag{3.12}$$

where $m_{expected}$ is the peak value of the mass to charge ratio in the mass spectrum. Compared to other mass spectrometers, FT-ICR can achieve mass accuracy up to ppb (parts per

billion).

Mass accuracy is dependent on the resolving power and signal to noise ratio. The resolution is given by the following equation:

$$R = \frac{m}{\Delta m} = \frac{\omega}{\Delta \omega} \tag{3.13}$$

where m is the mass and ω is the cyclotron frequency, Δm and $\Delta \omega$ are the width of the peak at half maximum, which is also called full width at half maximum (FWHM) [10]. The narrower the width of the peak in frequency and mass spectrum, the higher the mass resolution. High resolution mass spectrometers provide higher mass accuracy.

A recent study [21] on a modified Orbitrap Exploris mass spectrometer showed that resolution $> 2,000,000$ can be achieved for m/z < 200 for long transients (4s), allowing to resolve fine isotopic structures. According to the authors, for comparison, same experiment was repeated in a 21 T FT-ICR machine in the absorption mode and they achieved higher resolving power with the new Orbitrap instrument.

Mass calibration

In mass spectrometry, calibration laws must be applied when converting frequency spectra to mass spectra. Mass calibration is important in mass spectrometry for accurate measurements of mass to charge ratios. There are several proposed methods for mass calibration. The chosen method depends on the spectrometer and application. The most common mass calibration methods are internal and external calibration. External calibration is the simplest method. However, internal calibration is more advantageous as it provides higher accuracy. Mass calibration methods are discussed in more detail in [22, 23, 24]. [22] discusses the important considerations in mass calibration in order to obtain the best accuracy and precision.

Chapter 4

Other Fourier-based and non-Fourier methods for harmonic analysis of signals

The recorded signal in FT-ICR is measured at regular time intervals. The accuracy of the spectrum depends on the sampling methods used to obtain the signal. Uniform sampling is done in one-dimensional experiments. However, in multidimensional experiments, non-uniform sampling methods are preferable due to problems caused by uniform sampling [25]. Several Non-Fourier methods have been implemented for this purpose that do not require data to be sampled uniformly. In this chapter, we discuss sampling methods and some commonly used non-Fourier harmonic analysis methods as well as an efficient denoising method.

4.1 Sampling methods

Below, we define some terminology for sampling methods.

Uniform Sampling (US)

Uniform sampling is when the samples are taken contiguously.

Regularly sampled

Sampling spaced by the same time interval for which DFT applies.

Irregularly sampled

Irregular sampling is when the sample values are irregularly spaced and the sampling location is not known. DFT does not apply in this case. In real applications, the sampling location

is usually unknown. [26] presented a method to reconstruct irregularly sampled signals with sampling location not known.

Global Random Sampling (GRS — our contribution)

In iterative algorithms where many evaluations are done, the idea is to iteratively use different sub-samplings of the same signal so that by the end of the run, all the data points of the signal have been used, even though they were never all used on one single evaluation. For instance, on a 16M points signal, rather than evaluating a sum of sines on the 16M points which would take a lot of time) we could use 4k points for generations 0 to 9, then a different subsample of 4k points for generations 10 to 19. On 4000 generations, the 16M points of the signal would have been used (in probability).

In this thesis, we tried 2 versions of GRS sampling: GRS by replacement and GRS by extension.

In GRS by replacement, we start with a sample of size n of signal s and every k generations, we do another sampling n, so that by the end of the run, the algorithm has learned using all the available points of the signal (as described above).

In GRS by extension, we start with a small uniform sample of size n and after k generations, as we change the location where the uniform sample is taken, we add new points so as to use a larger sample.

In the end, we end up using all data points in both versions of GRS.

This can only be done in non-FFT algorithms such as evolutionary algorithms, that can work iteratively on different subsamples of the global signal.

Oversampling

Oversampling means sampling significantly faster than the Nyquist rate. Oversampling has several advantages. It can improve the dynamic range which is the ratio of the largest and smallest signals and the signal-to-noise ratio and thus the resolution. Oversampling can also prevent aliasing and phase-distortion [25].

Non-uniform Sampling (NUS)

Non-uniform sampling: in this case the sampling is regular in time but there may or may not be a value [27].

Figure 4.1: Non-uniform random sampling on grid

In real applications, such as in medical imaging, astronomy, geophysics, etc. non-uniform sampling is more appropriate to use as it is not always possible to obtain uniformly spaced samples. Non-uniform sampling is one of the methods used to improve the Nyquist-Shannon sampling theorem which is for uniform samples.

Non-Fourier methods have been implemented to deal with non-uniformly sampled data since uniform sampling raises problems with limits of resolution and long measurement time is required in order to obtain high resolution frequency domain spectra. Non-uniform sampling speeds up the acquisition time and it is mainly useful in multidimensional FT-ICR data. In some applications, it is even impossible to obtain uniformly sampled data. The FT method can not be performed on NUS data. Also, the frequency-domain spectrum obtained by FT is not accurate when data is discrete and noisy as it can not distinguish between noise and signal. Therefore, non-Fourier methods have been implemented as they can provide more accurate estimates.

4.2 Description of other methods

The FT-ICR mass spectrometer includes non-Fourier methods such as Hartley Transform, Bayesian Maximum Entropy, and Linear prediction.

In this section, we describe these methods and other methods proposed in the literature.

Hartley Transform

Hartley transform (HT) is very similar to Fourier transform except it is real-valued. The Hartley frequency-domain spectrum is given by:

$$H(f) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} x(t)\mathrm{cas}(\omega t)dt \tag{4.1}$$

where $\mathrm{cas}(t) = \cos t + \sin t$.

It produces the same spectra as FT. The HT is especially useful for signals that are represented by real data, such as linearly polarized time-domain signals. Using FFT for such signals increases the complexity as the computational steps are the same as if the data were complex even though the imaginary part is taken as zero. Another useful property of the Hartley transform is that it is its own inverse [28].

Since HT deals with real data, it is twice faster than FFT. Williams and Marshall [29] compared the computational speed of real FFT and fast HT (FHT) methods in ICR mass spectrometry. 1k-16k data points were acquired in ICR mass spectrometry. The comparisons were made on Atmi 1040ST (8-MHz M68000 cpu, l-megabyte RAM) using GFA Basic. It was shown that the computational speed of FHT is equivalent to the speed of real FFT. Even though they are equivalent in computational speed, HT is much simpler compared to FFT, as it does not use complex variables.

Bayesian Maximum Entropy

This method, as described in [30], is based on Baye's theorem and is used to estimate the spectrum using any prior information about the signal:

$$P(H|D,I) = \frac{P(H|I)P(D|H,I)}{P(D|I)} \tag{4.2}$$

where $P(H|D,I)$ is called the posterior probability, H is the hypothesis, which is the spectrum to be tested, D is the data, and I is any prior information such as initial or range of values of the parameters of the signal. $P(H|I)$ is the prior probability of hypothesis conditional on the prior information, which is the initial estimate of the spectrum. $P(D|H,I)$ is called the direct probability distribution for each data point. It is conditional on the hypothesis and prior information. $P(D|I)$ is the prior probability distribution of data, which is conditional on prior information. This method makes it possible to use the data to make inferences about the signal.

[30] compared the Bayesian method with the FFT method using different signal models. The first experimental data consisted of 3 damped and noisy sinusoids with frequencies 250, 300, and 600 Hz. It was obtained using a Nicolet FTMS-2000 FT-ICR instrument operating at 3.0 T. For the analysis of FFT, 16k data was zero filled to 32k. The data was also multiplied by a Gaussian weight function. In the analysis of damped noisy data, 1k points were used in Bayesian analysis and it was shown that Bayesian method returns more accurate results compared to FFT. The error in frequencies were in the range $0.0023 - 0.0203\%$, whereas the error in frequencies using FFT were in the range $0.0217 - 0.1284\%$.

The authors also tested the methods by increasing the complexity of the signals until Bayesian method starts to fail. It was found that increasing the complexity of the signal increases the computation time for Bayesian method. For example, Bayesian analysis was performed using 2k points on an electron-ionized Xe sample, with 7 isotopes. The Bayesian method was able to find the 6 peaks, with computation time of 6 hours. Which is very long compared to FFT, which was able to find even the 7th peak using 32k zero-filled data (Figure 4.2). With more data values, Bayesian method would have been able to find the 7th peak, but in much longer time. Frequencies obtained by Bayesian method was found to be more accurate than FFT, whereas the accuracy of relative abundances were about the same in both methods.

The Bayesian spectral analysis is used to obtain absorption mode spectrum. Using the Bayesian analysis on noisy signals, mass accuracy can be improved by a factor of 10 or more compared to the FFT [30]. However, there are disadvantages of this method. It is very costly in time compared to FFT (hours vs seconds). Therefore, it is recommended to be used in applications of FT-ICR in the following cases:

- high signal frequency accuracy is required
- small number of data points are used (about $\leq 2k$)
- truncated time-domain signal
- 2 or more signal frequencies are close to each other

The Frame method

Frames are known as a redundant and stable way of representing signals. They are particularly useful in nonuniform sampling when the sampling rate is much higher, resulting in oversampling. Which means that there are more than enough samples to reconstruct the signal. If sampling is done at exactly right frequency, then reconstructing functions form a basis.

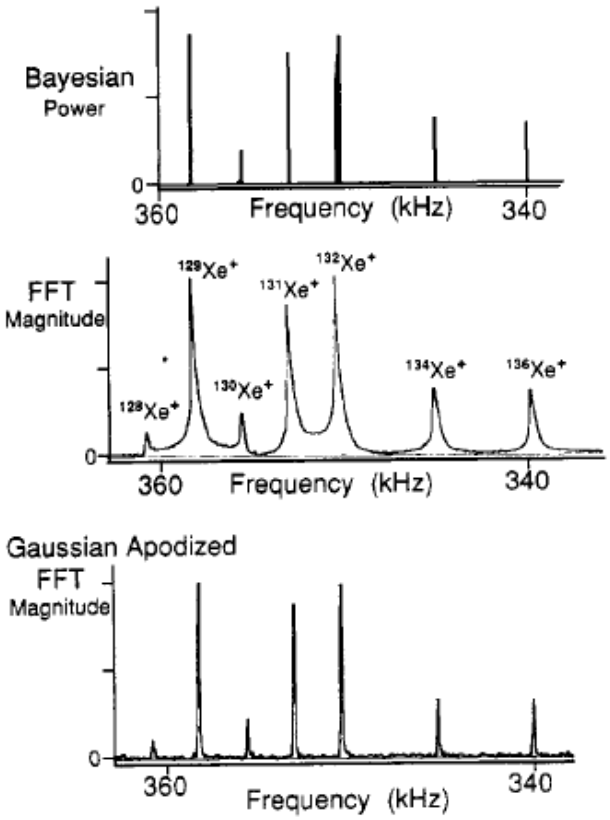

Figure 4.2: Bayesian vs FFT (adapted from [30])

In order to reconstruct the band-limited signal f from its N non-uniform samples $f(t_i)$, we can represent it as a linear combination of the functions f_i:

$$f(t) = \sum_{i}^{N} f(t_i) f_i \tag{4.3}$$

If f_i are linearly independent, then they form a basis. Otherwise, they form a frame.

A family $\{f_i\}_{i \in Z}$ is called a frame in Hilbert Space $\mathbf{H}$, if $\exists$ frame bounds $A, B > 0$ such that:

$$A\|f\|^2 \leq \sum_{i} |\langle f, f_i \rangle|^2 \leq B\|f\|^2 \tag{4.4}$$

$\forall f \in \mathbf{H}$. Then, the frame operator, denoted by S is defined by:

$$Sf = \sum_{i} \langle f, f_i \rangle f_i \tag{4.5}$$

with $AI \leq S \leq BI$, where I is the identity operator. Therefore, S is invertible and:

$$f = \sum_i \langle f, f_i \rangle S^{-1} f_i = \langle f, S^{-1} f_i \rangle f_i \tag{4.6}$$

If a set $\{f_i\}_{i \in Z}$ forms a frame for $\mathbf{H}$, then the signal can be reconstructed from $\langle f, f_i \rangle$.

The sampling theory can be linked with the frame theory using the sinc function. If $\mathbf{B}$ is the space for band-limited signals and $\{\text{sinc}\}$ is a frame for $\mathbf{B}$, then the signal f can be reconstructed from its samples using the following formulas:

$$f(t) = \sum_i f(t_i) S^{-1} e^{2\pi k t_i \omega} \tag{4.7}$$

or:

$$f(t) = \sum_i c_i \text{sinc}(t - t_i) \tag{4.8}$$

where c_i are the solution to the Grammian matrix $\mathbf{Rc{=}b}$ with entries $\text{sinc}(t_i - t_l)$ and b is a vector with entries $\{f(t_i)\}$

Strohmer [27] described two different approaches based on these equations. One is the truncated frames method using large unstructured linear system of equations, which is the traditional approach and leads to an ill-posed problem. This is because this method does not have a specific structure and numerically, it is very expensive. Second approach proposed by Strohmer is based on trigonometric polynomials which leads to well-posed problem. Compared to the truncated frames method, this method is rich in mathematical structure and thus the stability of this method only depends on the sampling rate. The performance of the proposed method was tested on a spectroscopy problem and noisy data to approximate the Earth's magnetic field and desired results were obtained.

Linear Prediction Cholesky Decomposition

The Linear prediction (LP) method assumes that the signal can be expressed as a linear combination of the past or next values. If we let the vector $\mathbf{x} = (x_1, x_2, \ldots, x_m)$ be the vector of data points, $\mathbf{b} = (b_1, b_2, \ldots, b_m)$ be the vector of backward prediction coefficients, then we can write:

$$\mathbf{x} = \mathbf{X}\mathbf{b} \tag{4.9}$$

which can be rewritten in the form:

$$\begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_m \end{pmatrix} = \begin{pmatrix} x_2 & x_3 & \ldots & x_{m+1} \\ x_3 & x_4 & \ldots & x_{m+2} \\ \vdots & \vdots & \vdots & \\ \vdots & & & \\ x_{N-m+1} & x_{N-m+2} & \ldots & x_N \end{pmatrix} \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_m \end{pmatrix} \tag{4.10}$$

If $\mathbf{X}$ is an invertible square matrix, then the solution can directly be obtained:

$$\mathbf{b} = \mathbf{X}^{-1}\mathbf{x} \tag{4.11}$$

However, if $\mathbf{X}$ is not a square matrix or if it is not an invertible matrix, the direct method does not work. In such cases, linear least squares method is used. One of the most commonly used methods is called Singular Value Decomposition (SVD) method. However, this method is not efficient computationally and also it can not deal with large data sets.

Cholesky decomposition method is an advanced linear prediction method that is robust and can deal with larger data sets compared to other linear prediction methods [31]. Given $\mathbf{x} = \mathbf{Xb}$, if we multiply both sides by transpose of $\mathbf{X}$ then:

$$\mathbf{X}^T\mathbf{x} = \mathbf{X}^T\mathbf{Xb} \tag{4.12}$$

The backward prediction coefficient can then be found using:

$$\mathbf{b} = \mathbf{A}^{-1}\mathbf{X}^T\mathbf{Xx} \tag{4.13}$$

where $\mathbf{A} = \mathbf{X}^T\mathbf{X}$ is a square matrix and $\mathbf{A}^{-1}$ is the inverse of $\mathbf{A}$.

In Cholesky decomposition, $\mathbf{A}$ is decomposed into a product of a lower triangular matrix and its transpose. $\mathbf{A}^{-1}$ is obtained using inverse of the triangular matrix and Rao's definition of a general inverse.

After determining the backward prediction coefficients, signal frequencies and damping constants can be determined by finding the roots of the following polynomial:

$$a^p - b_1a^{p-1} - b_2a^{p-2} - \ldots - b_{p-1}a - b_p = 0 \tag{4.14}$$

with roots C_i being of the form:

$$C_i = e^{-\lambda_i j\Delta t - j\omega_i\Delta t} \tag{4.15}$$

where λ_i is damping constant and ω_i is frequency.

The signal amplitudes and phases can be determined using the linear equation below:

$$x_j = \sum_{i=0}^{q}(e^{-\lambda_i j\Delta t}\cos(\omega_i j\Delta t)\mathbf{I}_i\cos(\phi_i) - e^{-\lambda_i j\Delta t}\sin(\omega_i j\Delta t)\mathbf{I}_i\sin(\phi_i)) \tag{4.16}$$

The authors in [31] compared the FFT and LP methods for crude oil sample with thousands of components. In FFT, 8k points with 8k zeros were used, in LP, 8k points and 2400 LP Coefficients were used in the analysis. Peaks were detected from $m/z = 110$ to $m/z = 250$, most of the peaks being around $m/z = 130$. FFT failed to identify peaks with m/z ratio larger than 250 due to low signal to noise ratio, while LP method was able to identify 20 peaks in that region that were possibly not artifacts. With $8k$ data points, the computation time of LP method on a Pentium Pro 200 MHz PC was 2.5 hours.

The limitations of the LP method are: computation time is long and it assumes that the signal is exponentially damped, which is referred to as Prony modeling. This model may not work with ions that have high mass since the decay may not be exponential. Also, the LP method works well with data points up to $8k$.

Multiharmonic Least-Squares Fitting

Another non-Fourier method to analyse periodic signals is known as Multiharmonic Least-Squares Fitting Algorithm as described in [32]. This algorithm is an extension of three [33] and four-parameter [34] sine-fitting algorithm. In the three-parameter algorithm, the signal frequency is known. The four-parameter algorithm assumes that the signal frequency is not known. The accuracy of the estimations based on these methods is low because these algorithms fit the sine wave into nonsinusoidal samples. This yields errors in frequency, amplitude and phase estimates. Multiharmonic Least-Squares Fitting Algorithm was proposed to overcome this problem.

Multiharmonic signal is described using the following model:

$$y(t_m) = C + \sum_{h=1}^{H} D_h \cos\left(2h\pi f t_m + \phi_h\right) = C + \sum_{h=1}^{H} [A_h \cos\left(2h\pi f t_m\right) + B_h \cos\left(2h\pi f t_m\right)] \tag{4.17}$$

where H is the number of harmonics, $y(t)$ are samples at time instants t_m for $m = 1, ..., M$. M is the number of samples. C is the DC component amplitude, D_h is the signal amplitude, ϕ_h is the phase of each sine, f is the fundamental frequency. The samples are taken with sampling frequency f_S and $t_m = \frac{m-1}{f_S}$:

$$D_h = \sqrt{A_h^2 + B_h^2}, \phi_h = -\text{arctan2}(B_h, A_h) \tag{4.18}$$

with A_h is in-phase amplitude and B_h is in-quadrature amplitude for each harmonic h.

First, initial frequency is estimated using Interpolated FFT. This estimation also produces the amplitudes and phases that minimize the root mean square error for the initial frequency. When the signal frequency is known, the solution vector is given by:

$$\mathbf{x} = [(\mathbf{D}^T\mathbf{D})^{-1}\mathbf{D}^T]\mathbf{y} \tag{4.19}$$

where the matrix $\mathbf{D}$ is full Least Squares matrix given below with M rows and $2H + 1$ columns and $\omega = 2\pi f$.

$$\mathbf{D}= \begin{pmatrix} \cos(\omega t_1) & \sin(\omega t_1) & \dots & \sin(H\omega t_1) \\ \cos(\omega t_2) & \sin(\omega t_2) & \dots & \sin(H\omega t_2) \\ \vdots & \vdots & \vdots & \\ \vdots & & & \\ \cos(\omega t_M) & \sin(\omega t_M) & \dots & \sin(H\omega t_M) \end{pmatrix}$$

Then, the best frequency can be determined iteratively which adjust all harmonic parameters in order to minimize the total residual error.

This method was tested using a 16-bit DAQ board National Instruments (NI) 6013 and function generators Wavetek M39, Standford Research DS360 and HP33120A. Triangular waves with Gaussian noise were generated. The method was tested using two signals, triangular and a three harmonic signals. The results based on the proposed method showed that

up to 43 harmonics of triangular signal can be fit in the algorithm. The limitations on the number of harmonics is due to the computer memory limits. Also, Interpolated FFT is used to estimate the initial frequency. Therefore, this method can work with uniform samples. For non-uniform samples, it can yield incorrect estimates.

Uncoiled random QR Denoising algorithm

Denoising algorithms are used to remove noise from the signal before processing it further. Several denoising algorithms have been implemented to improve harmonic signals. One approach is to use singular value decomposition (SVD) [35] of a matrix. However, when the dataset is large, the use of SVD becomes less attractive. SVD decomposition is also very slow, with time complexity $O(ln^2)$. Another limitation of SVD based algorithms are that if the assumed number of frequencies is not correct, then the denoised signal includes artifacts. Even though there were several rapid SVD methods that were implemented, artifacts is still a problem. An SVD based method was successfully attempted in 2D NMR [36]. However, because the size of 2D FT-ICR is at least 1000 times larger than 2D NMR, SVD approach is not suitable in 2D FT-ICR.

Chiron *et al.* [37] presented a harmonic signal denoising procedure called urQRd (uncoiled random QR denoising) based on random sampling. This reduces the dimension of the matrix and makes it possible to be used with large data sets. This method is based on QR decomposition of a randomly chosen matrix from the data. QR decomposition is decomposing a matrix into a product of an upper triangular matrix and an orthogonal matrix.

In the QR denoising algorithm, matrix $Y_{L\times K}$ contains the most important information about the trajectory matrix $H_{L\times N}$. H is called Hankel matrix, which is formed from the data vector X of size S, so that $S = L + N - 1$. The matrix Y is obtained multiplying the matrix H by a matrix $\Omega_{N\times K}$, which is a matrix storing random vectors:

$$Y = H \times \Omega \tag{4.20}$$

For $K \leq L$, $L \leq N$, the size of the matrix Y is smaller than the size of the matrix H. QR factorization is applied on Y so that $Y = QR$ where Q is the reduced rank orthonormal basis of H. QR factorization is performed to form a projection of H onto Q:

$$\tilde{H} = QQ^T H \tag{4.21}$$

Thus, $\tilde{H}$ is projection of H onto Q, with rank K.

Then, denoised signal $\tilde{X}$ can be rebuilt from $\tilde{H}$ using diagonal averaging. The approximation error is given by:

$$\|H - \tilde{H}\| \leq [1 + 9\sqrt{K}\sqrt{L}]\sigma_{p+1} \tag{4.22}$$

with probability at least $1 - 3p^{-p}$ and $p = K - P$, where P is the number of components contained in signal. σ_{p+1} is the greatest $p + 1$ singular value of H. The method described above is referred to random QR denoising, rQRd method. An improved version of this algorithm for larger very long signals is also presented by the authors, and it is called

uncoiled random QR denoising (urQRd). This algorithm is based on FFT, which allows for fast matrix-vector multiplication to improve the calculations of the Hankel matrix H.

The rQRd and urQRd algorithms perform much faster and with smaller computer memory foot-print compared to classical SVD approach. The urQRd is very efficient and can be applied to large datasets found in FT-ICR mass spectrometry experiments and in any large datasets in high-resolution spectroscopes but the algorithm still remains very costly in computing time (several weeks on expensive Xeon-Phi servers).

urQRd algorithm can also be used to reconstruct missing values. Bray *et al.* [38] implemented non-uniform sampling (NUS) acquisition in 2D FT-ICR mass spectrometry using the urQRd algorithm. Uniform sampling in two-dimensional FT-ICR experiments requires large amount of time. Therefore, NUS is used in multidimensional cases. In NUS acquisition, data has to be treated to reconstruct the missing points. urQRd algorithm is used for this purpose, considering the missing values are noise-corrupted values. Then, FFT is applied on the reconstructed signal. It was shown that NUS acquisition increases resolving power and decreases duration of the experiment.

Compressed Sensing Method

Compressed sensing is another signal processing method for reconstructing a signal by solving underdetermined system of linear equations [39]. It is also known as sparse sampling as this method is based on the principle that when the signals are sparse, meaning, there are many zero elements and few non-zero elements, then signals can be reconstructed using fewer samples than what is required by the Shannon-Nyquist sampling theorem. Compressed sensing is known as a powerful technique for audio and image compression and has gained great interest from researchers.

Let $f(t)$ be an unknown function that we wish to recover using as few samples as possible. Let $g_i(t)$ be basis functions such that:

$$f(t) = \sum_i \lambda_i g_i(t) \tag{4.23}$$

where the coefficients λ_i are mostly 0. It is not known which values are 0. Let t_j be sample points, then we obtain the following set of linear equations:

$$f(t_j) = \sum_i \lambda_i g_i(t_j) \tag{4.24}$$

The idea is to solve the linear system for the coefficients λ_i. Since few samples are taken randomly, this system is underdetermined. Therefore, an additional constraint such as minimization of $\ell 1$ norm is imposed.

The solution to the underdetermined linear system can be found by solving the $\ell 1$ minimization problem:

$$argmin_{\lambda_i} \|\lambda_i\|_1 \text{ subject to } \|f(t_j) - \sum_i \lambda_i g_i(t_j)\|_2 < \epsilon \tag{4.25}$$

where $\|\lambda_i\|_1$ is the $\ell 1$ norm and is the sum of the absolute value of λ_i, $\|\cdot\|_2$ is the $\ell 2$ norm, and ϵ is error. $\ell 1$ norm promotes the solutions of the system to have as many 0's as possible.

A similar $\ell 1$ minimization technique is super-resolution [39]. It differs from compressed sensing in sampling technique. In compressed sensing samples are taken randomly over the entire time domain while in super-resolution, regular sampling is used over the entire time domain.

Filter Diagonalization Method

Filter Diagonalization Method (FDM) is a super resolution method that is used to overcome the drawbacks of FT. The basic idea behind this method is to convert nonlinear fitting problem into linear algebra problem where small data matrices are diagonalized to extract the parameters of the sinusoids. This method was first introduced by Neuhauser in the area of quantum mechanics. It was successfully applied to NMR by Mandelstham *et al.* [40].

In FDM, the signal:

$$y_n = \sum_k A_k e^{-i\omega_k n\Delta t} \tag{4.26}$$

can be represented as autocorrelation function of a fictitious dynamical system with Hamiltonian $\hat{\Omega}$. Then:

$$y_n = \langle \Phi(0)|e^{-k\hat{\Omega}n\Delta t})\rangle = \langle \Phi(0)|\hat{U}^n\Phi(0)\rangle \tag{4.27}$$

where $\Phi(0)$ is some initial state. Nonlinear fitting problem is now reduced to diagonalization problem. The idea is to diagonalize the matrix $\hat{U} = e^{-i\Delta t\hat{\Omega}}$ which has the eigenvalues $u_k = e^{i\omega_k\Delta t}$ and associated eigenvectors B_k.

By solving the generalized eigenvalue problem:

$$\mathbf{UB_k} = \mathbf{u_k SB_k}, \tag{4.28}$$

where $\mathbf{S}$ is an overlap matrix, we can obtain the eigenvalues and eigenvectors, and thus the amplitudes, frequencies and phases.

Kozhinov and Tsybin presented this method for finding the sines which provides higher resolution for FT-ICR mass spectrometry data acquisition. The method was tested on cisplatin, substance P, and equine myoglobin. The analysis was done on a standard desktop computer with a quad-core processor. In substance P, FDM was able to determine the peaks even for shorter acquisition time of $5ms$, compared to FT, which required larger acquisition time to detect the peaks. The FDM method was compared to FT and it was shown that with shorter transient, high resolution for macro-molecules and complex mixtures can be obtained with FDM method.

Limitations of the FDM method are that FDM method requires a higher computation time compared to FFT when the number of basis functions is large and it yields a poor resolution when the signal-to-noise ratio is low [41].

Markovich *et al.* [39] compared the super-resolution, compress sensing and FDM methods using different signals. The FDM and super-resolution methods use regular sampling, while in compress sensing, random sampling is used. It was shown that the compressed sensing and super-resolution method work better with arbitrary signals with priori information and they require the data to be on grid. FDM outperforms the super-resolution and compressed sensing when the signal is Lorentzian. The FDM method can recover frequencies off-grid.

The disadvantage of the FDM method was that it is very sensitive to slight deviations in the parameters.

4.2.1 Previous work using Evolutionary Algorithms

In this section, we describe previous works based on evolutionary algorithms.

GA applied to NMR signal

Few attempts to perform harmonic analysis using evolutionary algorithms have been published. In a pioneering work, Choy Sanctuary [42], used a Genetic Algorithm (GA) for parameter estimation in NMR signal analysis in 1998. It should be pointed out that the size of an NMR spectrum is at least 1/1000th of the size of an FT-ICR spectrum. This method was developed in order to overcome the limitations of conventional methods when the signal-to-noise ratio is low.

The probability of crossover and mutation was set to 0.9 and 0.1 respectively. Reproduction was done using the top best 20% chromosomes. Population size and the number of generations were both set to 100. The fitness was evaluated using the following equation:

$$f(x) = \exp\Big(-\frac{(x-x')^2}{NKA}\Big) \tag{4.29}$$

where N is the number of data points, K is the number of sinusoids and A is the average absolute magnitude of the data points.

Four different crossover methods were tested: single-point crossover, two-point crossover, discrete recombination, and intermediate recombination. It was found that the intermediate recombination outperforms all other crossover methods. In intermediate recombination, gene values of children are generated using values around parent genes. Mutation was done using breeder mutation.

The parameter values were used as priori knowledge in the analysis. Since in NMR spectroscopy, frequencies and damping factors are usually known, their values can be used as constraints. Using priori information about some parameters helps GA to obtain the results factor and more accurately. The results with and without constraints were also compared. It was shown that GA achieves best result in fewer generations.

One-dimensional NMR signal consisting of 6 sines was generated in the analysis discussed in the paper. The analysis was done using 128 points on a Pentium 200MHz PC. The computation time was 50 seconds which is longer than other conventional methods. Due to the longer computation time of the GA method caused by the low performance of their computer, it was suggested that this method be used only when the signal-to-noise ratio is low.

GA for analysing Power Systems

In 2006, Zamanan *et al.* [43] used a real coded GA in place of the FFT to describe a simple signal composed of 14 sines and cosines coming from power systems. Considering a voltage waveform of the form:

$$V(t) = A_0 \cos(\omega_0 t + \phi_0) + A_1 \cos(\omega f_1 t + \phi_{f1}) \cos(\omega_0 t + \phi_0) \tag{4.30}$$

GA was used to find the harmonic components. The fitness function used in this analysis is given by the following formula:

$$F = \sqrt{\frac{\sum_{i=1}^{m} Error_i^2}{m}} \tag{4.31}$$

where $Error = V_{actual} - V_{predicted}$.

Comparing the real coded GA with FFT, only 20 points were needed to achieve a proper estimation of the signal with GA method, whereas the FFT required 200,000 points to get the same result. Using 10^4 fewer points seems a remarkable achievement but unfortunately, the paper is quite short (4 pages) and apart from the fitness function, the algorithm is not described, meaning that the results are not reproducible. Then, the tackled problem is not harmonic analysis in general, but dedicated to flicker frequency detection in electrical systems.

Differential evolution algorithm applied to frequency-domain signal

The EA method is also described in papers [44, 45, 46] by the same authors. The authors used Differential Evolution (DE) method in their analysis. This method depends on the assumed number of sinusoids, denoted by K' which must be carefully chosen. It was shown that when the number of samples N is greater than $3K'$ and $K' = K$ where K is the actual number of sines, DE algorithm works very well and returns precise results. If $N > 3K'$ and $K' > K$, DE produces unwanted small frequency components which can be ignored. When $K' < K$ or N is too small, DE returns unsatisfactory results. In [44], the authors simulated a signal with 2 sines, with frequencies 0.1 and 0.11, using $N = 6$, $K' = K = 2$. Good result was obtained using very small number of samples. However, the computation times was too long according to the authors. This method was compared to the Prony method and higher precision was obtained by DE method. The analysis was done using real-valued variables.

In [45], the authors proposed complex valued approach. This method has several advantages compared to the real-valued approach as it returns higher precision, analysis time is shorter and smaller samples are used. As a result of the experimental runs, it was concluded that analysis can be performed using very small number of samples. With small number of harmonic components, high precision can be obtained even with small samples which is not possible when using DFT method.

Multiharmonic Waveform Fitting of Periodic Signals using GA

[47] used GA method to estimate the parameters in multiharmonic waveform fitting instead of the traditional Least-Squares (LS) method. GA is used to determine the fundamental frequency which is used in a multiharmonic least-squares (LS) waveform fitting algorithm.

In GA, since the objective is to determine only the fundamental frequency, there is only one gene in the chromosome. The fitness was evaluated using least-squares error. The algorithm is as follows: initial population is randomly chosen, fitness is evaluated for each individual, if the error is sufficiently small, the algorithm stops and fundamental frequency is obtained. Then the fundamental frequency is used to find the amplitude and phases using the multiharmonic least-squares (LS) waveform fitting method. If the error is not small

enough, crossover and mutation operators are used. The steps are repeated until a good solution is obtained or number of generations is reached.

In the LS method convergence is a problem as it is difficult for this method to find the absolute minimum. The GA method overcomes this problem as it can recover from local optima. This method is also used in impedance measurement and it was found that the GA method is extremely useful in measuring impedance frequency response.

FT-ICR phase correction using GA

Kilgour *et al.* [13] used genetic algorithm for phase correction of FT-ICR signal. The algorithm was used in comparison with autophaser method, which is an iterative method [48]. The autophaser algorithm can produce absorption mode, however, it is computationally intensive. Therefore, GA was used to improve the processing speed.

There is a quadratic relationship between the frequency and phase correction for FT-ICR mass spectrum. The GA is used to optimize the following phase correction function:

$$\phi = af^2 + bf + c \tag{4.32}$$

where f is the frequency. The coefficients a, b, c are determined using least-squares method to the phase correction.

The initial population consists of possible phase correction functions. Population size 100 was used. The fitness was evaluated using the following function:

$$fitness = \frac{\sum_{i=1}^{n} \cos(\theta_i - \phi_i)}{n} \tag{4.33}$$

where n is the number of peaks detected. θ_i is the phase of $i'th$ peak obtained from FT and ϕ_i is the corrected phase.

The algorithm was tested experimentally using crude oil data coming from FT-ICR, with $8M$ points and 3748 peaks. It was found that using GA instead of autophaser algorithm improves the processing speed by 11 hours for a mass spectrometric image of 20000 pixel.

Conclusion on previous work using evolutionary algorithms

The bibliographic study showed that good results have been obtained by artificial evolution on specific signal processing related problems. It is a good indication that they could represent an interesting new class of fundamental algorithms for generic harmonic analysis.

The aim of this PhD thesis is therefore to study the true potential of artificial evolution for harmonic analysis, by empirically checking out its performance compared to mathematically-based Fourier Transform.

Chapter 5

Machine learning and optimisation

The epistemological view of machine learning and optimisation presented in this chapter is the one developed in the CSTB team of ICUBE Laboratory of Strasbourg University.

5.1 Laws and Ontologies: a philosophical background

Laws are among the first texts ever written in the history of humanity. The definition of Law (found in Wikipedia[1]) states:

> *Law is a system of rules created and enforced through social or governmental institutions to regulate behavior,[2] with its precise definition a matter of long-standing debate.*

What is nice with this definition is that *regulating behavior* applies to any kind of entities, be they human beings (*cf.* code of Ur-Nammu[2], that was written about 4000 years ago using cuneiform) or to particles (*cf.* Newton's Law of Universal Gravitation[3]) that is written as the following equation:

$$F = G\frac{m_1 m_2}{d^2}$$

This *Law* explains the behaviour of planets rotating around the Sun. But what are planets, and what is the Sun? These are *Objects*, whose concept are the result of the work of Parmenides of Elea[4] who, 2500 years ago, studied existence, being, becoming and reality (*cf.* `https://en.wikipedia.org/wiki/Ontology`) and therefore created the science of objects.

The Wikipedia definition of an ontology such as defined in computer science is the following:

[1] `https://en.wikipedia.org/wiki/Law`
[2] `https://en.wikipedia.org/wiki/Code_of_Ur-Nammu`
[3] `https://en.wikipedia.org/wiki/Newton's_law_of_universal_gravitation`
[4] `https://en.wikipedia.org/wiki/Parmenides`

> *In computer science and information science, an ontology encompasses a representation, formal naming and definition of the categories, properties and relations between the concepts, data and entities that substantiate one, many, or all domains of discourse. More simply, an ontology is a way of showing the properties of a subject area and how they are related, by defining a set of concepts and categories that represent the subject.*

Indeed, in computer science, ontologies are represented by indented trees and graphs that determine the characteristics of objects and how they are related.

In Newton's Law of Universal Gravitation, what characterizes the objects whose behaviour this law describes is their mass (m_1 and m_2) and the distance d that separates them.

This distinction is very apparent in artificial evolution where two kinds of algorithms exist: algorithms that can be used to create equations that model some data (Genetic Programming) and algorithms that find the value of parameters that allow a pre-existing equation to fit some observed data (Genetic Algorithms, Evolution Strategies or Evolutionary Programming).

To summarize with the example of Newton's Law of Universal Gravitation, out of the trajectories of two objects orbiting one another:

- Genetic Programming (GP) will be able to find the $F = G\frac{m_1 m_2}{d^2}$ law (or equation), while
- Genetic Algorithms (GA) or Evolution Strategies (ES) or Evolutionary Programming (EP) will, given Newton's law $F = G\frac{m_1 m_2}{d^2}$ find the values of m_1, m_2 that fit the given the trajectories.

So *GP will be able to find the law* (this is "'machine learning"), while GA, ES and EP will *find the characteristics of objects* that follow a law (therefore filling ontologies, which could be seen as "optimisation").

Indeed, optimization is the science of finding parameters (that minimize (or maximize) the value of a given equation (a law) that can be found using GP (machine learning).

In this PhD, these two techniques will be used:

- GP for finding a damping law (an equation) for the observed damped signal coming out of ICR and
- ES for finding the parameters (a characterization) of a sum of sines (objects) that compose the damped signal coming out of the ICR.

Chapter 6

Stochastic algorithms

Contrarily to a random search, that is based on trial and error, stochastic algorithm are algorithms that use a random component, so that they can avoid looking exhaustively for all the potential solutions to the problem. The complexity of the Travelling Salesman Problem (TSP) is $(n-1)!/2$ where n is the number of places where the travelling salesman should stop before returning back home. This means that for 25 stops, the number of potential routes is 24!/2, *i.e.* 3,102242009e23 routes. Supposing that a computer can compute 1 billion routes / second (not possible on a 4GHz PC because a 4GHz PC with a scalar processor could only do 4 billion additions per second, and evaluating one route requires 25 additions), evaluating 3,102242009e23 routes would require a deterministic algorithm 3,102242009e14 seconds. At 3600 seconds per hour, 24 hours per day, 365 days per year, this is 9 837 144,877810113 years.

Knowing that a courier driver must deliver around 100 packages per day, there is no way a computer could give him an optimal route for delivering packages to his customers for the day to come.

Stochastic algorithms make stochastic choices (that could be seen as probabilistic choices) to avoid exploring ALL solutions so as to reduce the explored search space. The risk is to miss the best solution (global optimum) to the problem, but the drastic gain in time will make it possible for the deliverer to have a good route to follow, if not the optimal route to follow. This poses the problem of reproducibility of the results, because different runs on the same problem will always yield different results.

So if an algorithm gives a good result, how is it possible to know that it did not obtain the good result by pure luck, and (for a concrete application), how long should an algorithm run to yield a result with a known goodness?

If for a given problem the best solution is known for one instance and if repeating the runs significant number of times returns a solution that is 99% as good as the known solution, then we can be confident that a solution to a new instance of the problem will be 98% as good as the result based on several runs but Evaluating the performance of stochastic algorithms is not easy. John Koza [49] proposed a performance measure called *computational effort*.

The *computational effort I(n,z)* is defined as the minimum number of evaluations must be processed in order to achieve a valid solution with probability of success being greater

than z. z is usually taken as 99%. The formula for the computational effort is:

$$n * \left\lceil \frac{\ln(1-z)}{\ln(1-P(n))} \right\rceil \tag{6.1}$$

where n is the population size, $P(n)$ is the cumulative probability of success defined as number of successful runs divided by number of total runs.

6.1 PSO

The Particle Swarm Optimization (PSO) is a population based evolutionary algorithm that was introduced by Kennedy and Eberhart in 1995. It is inspired from the social behavior of flocking birds or fish schooling in search of food. In PSO, the population consists of particles that represent potential solutions and is initialized with their random positions and velocities. The particles move around the search space with a certain velocity towards the best position found by the particle itself and the best position globally found by other particles. The position and velocity are updated continuously based on the particle's personal best (pbest) and global best (gbest) experience of other particles [50, 51].

This algorithm is described because it will serve as inspiration for the Quantum based Evolution Strategy (QAES) and Quantum Particle Swarm Optimization (QPSO) algorithms that have been tested on the harmonic analysis problem (cf. sections 12.1 and 12.1.1).

Mathematics of PSO

Let $\vec{x_i}$ and $\vec{v_i}$ be position and velocity vectors for particle i in an N dimensinal space.

$$\begin{aligned} \vec{x_i}(t) &= (x_{i1}(t), x_{i2}(t), ..., x_{iN}(t)) \\ \vec{v_i}(t) &= (v_{i1}(t), v_{i2}(t), ..., v_{iN}(t)) \end{aligned} \tag{6.2}$$

The velocities and positions are updated using the following equations:

$$\vec{v_i}(t+1) = \vec{v_i}(t) + c_1 U_1(\vec{p_i} - x_i(t)) + c_2 U_2(\vec{p_g} - x_i(t)) \tag{6.3}$$

$$\vec{x_i}(t+1) = \vec{x_i}(t) + \vec{v_i}(t+1) \tag{6.4}$$

where c_1 and c_2 are positive weight factors of the local information and global information respectively. U_1 and U_2 are random variables with uniform distribution, t is the iteration number, p_i is the best position of the particle i which is called *pbest* and p_g is the best global position among all particles which is called *gbest*. $v_i(t)$ is called inertia, which makes the particle remain in the same direction and velocity. The second term in the equation (8.2) is known as the "self-knowledge" which attracts the individual particle towards its best position and third term is called "social knowledge" which attracts the individual to the best position among all particles.

It is due to the U_1 and U_2 variables that contain random numbers that this algorithm is a stochastic algorithm.

The algorithm starts with initializing a population of particles [50].

- Each particle's position and velocity are randomly initialized. Fitness values of all particles are evaluated using the fitness function.
- The particles current position is set to p_i and objective position to *pbest* and the current position of the particle with the best fitness value among all particles is set to p_g and its objective is *gbest*.
- The position and velocity of particles are updated using the equations 8.2 and 8.3.
- If p_i is better than *pbest*, then update p_i and *pbest*
- Update p_g and *gbest*. If p_g is is better than *gbest* then replace them with the current best fitness value
- The steps are repeated until stopping criterion is reached.

Though PSO can be easily implemented, it may get stuck easily in local optimum. Several methods have been proposed to improve the PSO algorithm. Sun *et al.* proposed quantum based PSO (QPSO) in 2004, where the particles follow quantum behaviour [52], discussed later in the thesis.

6.2 Evolutionary Computing

Evolutionary algorithms are old optimization techniques inspired by Darwin's theory of biological evolution, described in his famous book *On the Origin of Species*, 1859.

The first adaptation of this work into computers were made in the 1950s as shown in David Fogel's Fossil Record [53], with Fraser, Friedberg and Friedman, who presented how binary strings could be evolved through crossovers [54], how computers could self-program using mutations [55, 56], and how evolution could be digitally simulated [57].

But evolution also worked into selecting the algorithms that are now the most used, that are Evolutionary Strategies (Rechenberg and Schwefel [58, 59]), Genetic Algorithms (Holland, Goldberg [60, 61]), and Genetic Programming, (Cramer [62], Koza [63]) to quote the most famous ones.

What needs to be understood is that these different algorithms of the Artificial Evolultion family address different problems.

Evolution Strategies (ES) Schwefel and Rechenberg's *Evolution Strategies* are oriented towards optimizing engineering problems. Therefore, in ES, solutions are represented as vectors of reals, that serve as parameters of functions to be minimised or maximised. As in all artificial evolution paradigms, potential solutions are called individuals. They are grouped into a population that evolves through generations.

Where Darwin described how animals and species evolved through unguided *variations*, that were later on shown by Gregor Mendel to be the result of crossovers and mutations the original ES algorithms did not rely so much on crossovers but more on mutations, that were implemented as added gaussian noise, to the real values that constituted the potential solutions to the problem (the individuals).

Genetic Algorithms (GA) Holland and Goldberg's *Genetic Algorithms* were developed on a more philosophical point of view, in order to address the fundamental question: can evolution have produced complex animals and human beings? Therefore, AG are focussed on developing a mathematical theory of evolution, which has some implications on the coding of these algorithms. Typically, solutions are represented as bit strings (not dissimilar to DNA, which is a strand made of 4 different nitrogen-containing nucleobases cytosine [C], guanine [G], adenine [A] or thymine [T])).

If a bit representation means that mutation is very simple (a 0 will be mutated into a 1 and conversely) the bitstring representation of individuals makes it difficult to represent integers and real values. Indeed, there is no simple way of mutating value 7 (0111 in binary) into value 8 (1000 in binary) which are neighbouring values, as changing 7 into 8 requires 4 mutations. Other types of representations are then used, such as a Gray encoding of binary values, or a Dedekin representation, but these representations also pose their problems.

Then, representing real values with bitstrings is even more challenging. If IEEE 754 Standard Floating-Point Arithmetic representation is used, then, mutating one bit into the exponent coding will considerably change the value of the real number, or even create a "Not A Number" (`nan`), or even worse, a sequence of bits that cannot be interpreted as an IEEE 754 number.

Crossovers between individuals are here again a lot inspired by genetics, where single point crossovers of individuals coded over b bits are implemented as taking the n first bits of parent 1 and glue them to the $b - n$ bits of the second parent to create a child. But here again, if the locus (the crossover point) is located in the middle of the representation of an IEEE 754 real value, the resulting value in the child will be very different from either values in parent 1 or in parent 2.

Finally, in 1985 and 1992, Cramer and Koza developed Genetic Programming [62, 63] whose aim was not to find optimal values to minimize fitness functions (optimization) but to create functions that would solve some observed data (laws), transferring artificial evolution into the Machine Learning domain. GP uses tree based representation.

The major preparatory steps for GP are the following:

- Specifying the set of terminals including independent variables of the problem, random constants
- Specifying the set of primitive functions
- Choosing the fitness measure
- Setting paramters to control the run such as population size, probabilities of performing the crossover and mutation
- Specifying the termination criterion

The evolutionary steps are shown in Figure 6.1.

Genetic Programming In the next section, we discuss the steps involved in Evolutionary Algorithms.

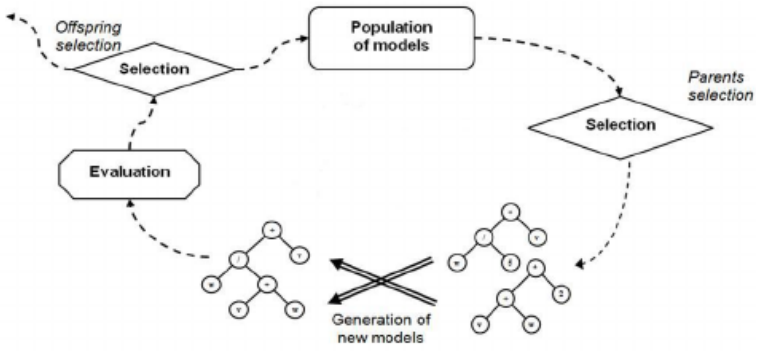

Figure 6.1: Flowchart of GP (adapted from [64])

Global view on evolutionary computing

Therefore, artificial evolution provides algorithms both to populate ontologies (by finding parameters, with Genetic Algorithms or Evolution Strategies) and modelling data (creating laws), with Genetic Programming.

6.2.1 Individual representation

The terminology used in evolutionary computation is borrowed from natural evolution since the idea of evolutionary computation comes from Darwin's theory of *natural selection* [65]. It states that *individuals* that are not well adapted to their environment have less chance to reproduce and survive to the next generation. Evolutionary algorithms take this idea to optimize solutions for difficult problems.

In artificial evolution, *individual* refers to potential solution to a given problem, *fitness* refers to adaptation. Individual representation is an important step. Genetic algorithms bit strings to code individuals, including real values. Evolutionary strategies however are designed for continuous problems and thus use real variables. Genetic programming uses binary trees. The choice of representation depends on the problem. Therefore, it is important to choose the good representation as the search space depends on it.

6.2.2 Description of the evolutionary engine

After individuals representation, the following steps are followed in evolutionary algorithms:

- First, the algorithm starts with initialising individuals, creating the initial population consisting of potential solutions to the given problem. In order to prevent premature convergence towards already known solutions, individuals are initialized with random values.
- The adaptation of each individual (potential solution) is evaluated using what is known as *fitness* or *evaluation* function. A fitness value is assigned to each individual. As

a result of evaluation, individuals become possible parents. The fitness function is dependent on the given problem.

- Create the children using the following steps:
  - Find good parents with an appropriate and possibly stochastic *selection operator*.
  - Call variation operators (crossover and mutation) to create children.
- Determine the fitness of the children.
- Reduce the size of the parents + children population to the initial size of the population of parents, by selecting (from both populations) among the best individuals.
- Check the termination condition. If the termination condition is satisfied, the algorithm is stopped and the best solution is returned. Otherwise, the steps starting from creating a new population of children are repeated until a good solution is obtained or until a time limit (or a number of generations) is reached.

The evolutionary algorithm follows an evolutionary loop as shown in Figure 6.2.

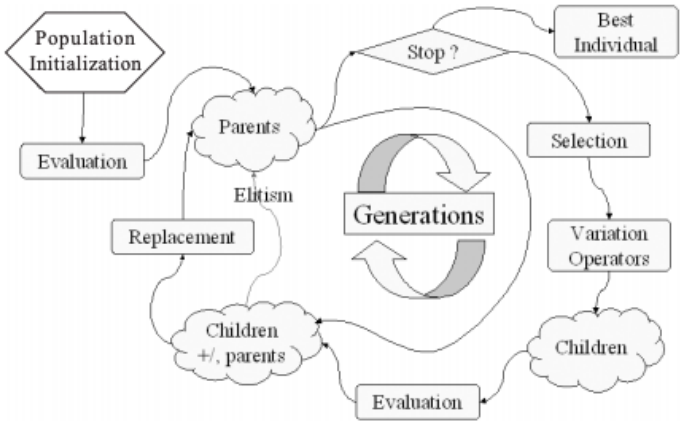

Figure 6.2: Flowchart of evolutionary algorithm (adapted from [66])

The main operators used in evolutionary algorithms are selection, crossover and mutation.

Selection and Replacement

The selection operator is used to select individuals to undergo reproduction based on their fitness values. The Selection operator is very important as selection pressure may cause slow or premature convergence.

There are two stages of selection: one is for parent selection to undergo reproduction and the other one is selecting individuals for next generation.

For parents selection, a selection with replacement is used, that allows the algorithm to choose several times the same parent.

Population reduction is obtained through selection *without* replacement, so as to avoid cloning good individuals in the next generation.

If elitism is used, the best individuals of the population are selected to be part of the next generation, in order to avoid losing the best solutions, because selection of individuals in the next generation is usally stochastic.

There are two kinds of elitism: strong elitism and weak elitism. The original GA uses generational replacement, which replaces the population of parents with population of children. In this case, if elitism is enabled, the best parents will appear in the new generation and children will be used to complete the population of the new generation.

In evolution strategies, population reduction allows both parents or children to be part of the new generation. In this case, weak elitism is used, to select the new generation from the best individuals of both parents and children population.

Using elitism may cause premature convergence, but prevents from losing the best individual once it is found.

Selection algortihms

There are several selection methods proposed in the literature, which are: Ranking [67], Stochastic Universal Sampling [68], Selection in Genitor [69], Truncation selection [70], Deterministic selection and Tournament selection [71, 72] etc.

Tournament selection is one of the best selection methods among all methods mentioned because the selection intensity can be easily tuned. In binary tournaments, 2 individuals are randomly selected and their fitness values are compared. The individual with the highest fitness is the winner of the tournament and thus is selected. This can be done with more individuals (n-ary tournament) which results in increased selection pressure. In the case of premature convergence, selection pressure can be decreased with *stochastic tournament* which is a binary tournament that returns the best of the two individuals, based on probability p, which is a parameter of the stochastic tournament function. If two individuals are selected with a probability $p = 0.5$ this is the same as random selection and if $p = 1$ then it becomes a binary selection tournament.

Crossover

The crossover operator is used to create one or more children by "mixing" the genotypes of parents. The way parents's genes are recombined depends highly on the problem. GA uses bitstring representation, ES uses vector of real values, GP uses tree representation. There are many different crossovers such as single point crossover, multi-point crossover, uniform crossover, barycentric crossover, etc. The choice of crossover depends on the individual representation. For example, if the genotype consists of real values, barycentric crossover may be more beneficial for some problems. In barycentric crossover, mean of the parents' genes is define the gene of the child.

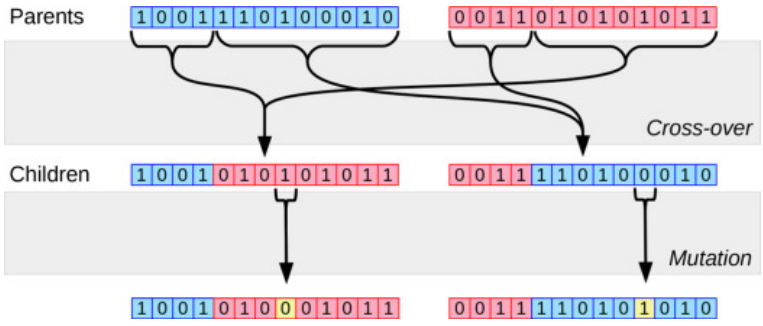

Figure 6.3: Using crossover and mutation to create/change offspring (Adapted from [73]).

The most common crossover operators are single or multi-point crossover. In these crossovers, two parents are chosen and one or more crossover points are selected. Then the parents' genes are swapped to create two children. Figure 6.4 illustrates how multi-point crossover is done.

Figure 6.4: Multi-point crossover (Adapted from [73]).

Then there is a uniform crossover. This is usually used with GA. Figure 6.5 illustrates uniform crossover with one child being created using three parents but it is very disruptive and does not yield good results if genes are semantically grouped.

It is also possible to chose only one parent and have the child to be clone of the parent.

Because crossover is problem dependent, it is not easy to establish the guidelines for crossover operator. In the first stages of the evolution, crossover is a great operator for finding good areas of the search space as it can create children out of diffrent parents. Then, as the generations accumulate, it becomes an exploitation operator that implements the search in the limited region in which the population is converging and therefore, it is doing a local search. When a population has converged (*i.e.* when all individuals of the populations are clones of each other), crossover loses its exploratory power and its efficiency is reduced.

Mutation

Mutation is also problem dependent. It starts as being an exploration operator that allows the algorithm explore out of the gene pool of the population. It therefore makes it possible to search in larger region of search space and by promoting diversification, it helps the population to escape local optima. Mutation is done by changing a value in the genome.

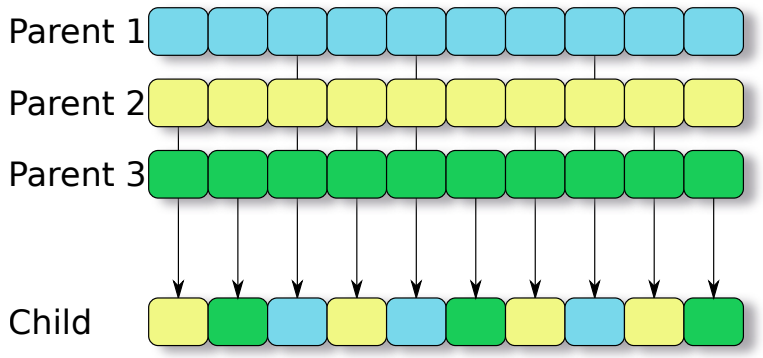

Figure 6.5: Uniform crossover (Adapted from [74])

There are several ways to do this. If genome is given by bit-strings, then a chosen bit can be simply flipped. However, if genome is real, then a Gaussian noise can be added to the selected value.

After variation operators, validation operators can be used to check the validity of the newly created children. The invalid children can be deleted and new children can be created in their place, or they may be repaired.

Figure 6.4 illustrates how children are created using a single point crossover and how mutation operator changes the gene of the offspring.

Stopping criteria

The most commonly used stopping criteria is to stop after several generations so some researchers evaluate used computing resources in terms of numbers of generations. This method works for generational replacement algorithms for which the number of children at each generation is the same as the size of population. However, in the case of Evolutionary Strategies, this method is not suitable because the number of children is not proportional to the size of the population. Therefore, for ES, the number of evaluations is the best choice rather than the number of generations to evaluate the consumed resources and determine when the algorithm should stop.

Run time and fitness value can also be used as stopping criteria. In the case of using run time as stopping criterion, it should be ensured that the algorithm converges at the end not before or after the run, by tuning the selection pressure or population size. In the case of using fitness value as stopping criterion, the algorithm will stop once the indicated fitness value is met.

Parameters

The main parameters used in evolutionary algorithms are the following:

Population size and number of generations

The product of these two parameters define the number of evaluations. One can either use

small population size and large number of generations or vice versa. Using large population size helps to keep diversity and avoid premature convergence.

Crossover and mutation probabilities

The crossover and mutation probabilities depend on the problem. The crossover is called with probability of 80-90 %. Mutation is followed after crossover. Mutation operator is usually called 100% of the time. However, this does not mean mutation of all children. All children have a chance to undergo mutation, but only few of them actually undergo mutation. High mutation probability may cause non-convergence.

Number of children

Number of children per generation is usually the same as the population size except in ES. The choice of number of children depends on the desired speed of the convergence of the algorithm.

Selection pressure

One of the most important parameters in artificial evolution is selection pressure, either for parents selection or for population reduction.

When using a tournament selection, pressure is controlled by the arity of the tournament. As parents selection is done with replacement and population reduction without, the same selection operator has a difference selection pressure in one case or the other.

Selection pressure will increase nearly linearly for parents selection, but not linearly for population reduction. Maximum selection pressure for population reduction is reached for a tournament of arity 10.

Selection pressure controlled by arity does not change with the population size (a tournament-5 will have the same selection pressure whether the population size is 100 or 100000).

Chapter 7

Parallelization of evolutionary algorithms

Evolutionary algorithms are inherently parallel. Indeed, this is what happens in Nature where breeding between individuals of a same species is even asynchronous.

In a computer, however, doing asynchronous parallelization is more difficult than doing synchronous parallelization because accessing shared memory must be done in a controlled fashion (2 threads cannot write at the same memory address simultaneously, otherwise some data will be lost).

Even though all parts of an evolutionary algorithm can be parallelized, the hotspot in evolutionary computation is the evaluation function. Indeed, selecting individuals, recombining and mutating their genes is very simple, compared to evaluating them on thousands of data points, and even more so if the evaluation involves computing several sines per data point.

So, in compute-extensive evaluations such as sums of sines over thousands of points for just one individual, evaluation will use 99% of the total computing time of the evolutionary run.

7.1 GPGPU parallelization

General Purpose Graphic Processing Units (GPGPUs) are processors that are initially designed to process images that are made of millions of pixels, or 3D scenes that are made of millions of 3D triangles. What is interesting is that whether GPGPUs must process pixel images or vectorial images, they have to repeat the exact same algorithm on the millions of entities that they are made of. What is more, the processing that needs to be done on one pixel or one triangle is independent of the same processing that needs to be done on another pixel / triangle of the image.

So the designers of Graphic Processing Units designed specific processors where most of the transistors are used to implement Arithmetic and Logic Units (ALUs), resulting in a strange architecture, with the aim of implementing as much computing power as possible

on the limited space of a silicon chip.

Where on a multi-core CPU, all cores are independent (meaning that they can run their own programs independently), GPGPU designers chose to maximise computing power at the cost of reducing versatility, which was even more possible that when processing a million pixel / vertices image, all the algorithms run on all pixels or vertices are identical to the instruction.

So one very radical simplification that saves a lot of space on the chip is the following : rather than surrounding each ALU with everything it needs to be independent the designers decided to group a number of cores into what they called "multi-processors" and having them share the functional units that tend the processors. Typically, an ALU needs a fetch and dispatch unit, to *fetch* from memory the next operator and operands to be loaded in the ALU and *dispatch* them in the correct registers.

This means that in a 32 core multi-processor, all cores will share the same fetch and dispatch functional unit, that will fetch the next instruction to be executed and will dispatch it in all the 32 cores of the same multi-processor, *meaning that in a multi-processor, all cores must execute the same instruction at the same time.*

This very restrictive form of parallelism is called *Single Instruction Multiple Data* (SIMD) in the Flynn taxonomy. It is perfectly suitable for graphic algorithms that can execute the very same instruction on all the different pixels of an image at the same time.

Now, there is not only one multiprocessor in a GPGPU chip, but many, all of them having their own fetch and dispatch unit meaning that if all the cores within one multiprocessor must do the exact same instruction at the same time (SIMD), several multiprocessors can run different parts of the same program at the same time (they all have their own Program Counter). So the very restrictive SIMD parallelism is relaxed into what is called *Single Program Multiple Data* (SPMD), where different multiprocessors can simultaneously run different functions of the same program.

7.1.1 Parallelizing ES on a GPGPU

The objective of an Evolution Strategy is to find the best parameters that will minimize an error function. If, in the present context of harmonic analysis, we know we are looking for n sines, and if each sine can be determined by 3 parameters (amplitude, frequency and phase), then, it is necessary to find the $3n$ parameters that will create an n sine function that will correspond to the data to be modelled.

So if a generation is made of 1000 new parameters sets (called children), evaluating the children to make parents out of them means that each child *will need to be evaluated using the same evaluation function, i.e. a sum of n sines* on maybe 1000 different points.

This is perfectly compatible with SIMD parallelism that states that if the evaluation of a child is assigned to a 32 core multi-processor, all the 32 cores can be given the same instruction to execute at the same instruction, but on different values. This is the meaning of Single Instruction Multiple Data : all the cores run a single instruction, but using different data = different parameters for the sine function.

This means that GPGPUs are perfectly suited to run Evolutionary Strategies, because all individuals are evaluated using the exact same function.

7.1.2 Parallelizing Genetic Programming on a GPGPU

Things are different for Genetic Programming because in this case, we are looking for a function, so by essence, all individuals will implement different functions, to find out which function best fits the data to be modelled. So assigning different individuals to different cores of a multi-processor will not work, because the evaluation function may need one individual to execute a $\times$ operator while another evaluation function may require a $+$ operator.

But fortunately, there is a workaround, which consists in not parallelizing the evaluation of 32 individuals on a 32 cores multiprocessor, but parallelizing the evaluation of a single individual on the 32 cores.

How is this possible? Well, in order to determine if a function matches well a set of n points, one needs to execute the function n times, once per data point. So supposing that GP is used to find a function that matches 32 points. The 32 function evaluations can be done in parallel on the 32 cores of the multiprocessor.

And because GPGPUs are SPMD (Single Program Multiple Data), the different multi-processors all have their own program counters, meaning that they can run on different parts of the same program.

So supposing there are 128 32-core multiprocessors on a GPU chip, each of the 128 multiprocessors can run a different individual, where all cores of a multi-processor will evaluate one point of the 32 data points to be modelled.

So here again, the GPGPU architecture is perfect for Genetic Programming, even though it means that parallelization is done in a completely different way as for Evolution Strategies.

7.2 Island parallelization

7.2.1 Parallelizing on independent islands

All the previous section describes how an artificial evolution program can be parallelized on one (or several) GPGPU cards of the same computer. However, it may be interesting to use several computers linked together *via* a network.

This could be done in several ways. The standard way would be to run a single algorithm on n machines, meaning that for a 10 000 individuals algorithm running on 10 machines, the population could be divided by 10 and at every generation, each of the 10 machine could evaluate 1000 individuals.

But this would request sending out the 10 000 individuals on the 10 machines at each generation, and get back the results once the evaluation is done. This would periodically (at each generation) overload the network and data transmission and synchronization time would slow down the computation, meaning that it would not be possible to go $10\times$ faster using 10 computers.

So another way of parallelizing the algorithms is used: island parallelization.

During his tour of the Galapagos on the Beagle ship, Darwin brought back to England samples of fauna and flora of the different islands of the Galapagos archipelago, that consists of 18 main islands, 3 smaller islands, and 107 rocks and islets, for 7,880 km^2 of land, spread over 45,000 km^2 of ocean.

Finches can be found on several of the Galapagos islands :

- on the 4640 km^2 Isabela island (the largest of Galapagos),
- on the 585 km^2 Santiago island,
- on the 60 km^2 Española island,
- on the 14 km^2 Genovesa island,
- on the 4.95 km^2 Rábida island,
- on the 4.9 km^2 Daphne island,
- on the 1.3 km^2 Wolf island and
- on the 1 km^2 nearby Darwin island.

The fauna and flora are obviously quite different on the 4640 km^2 Isabela island and on the 1.4 km^2 Wolf island or 1 km^2 Darwin island, which are basically barren rocks, reaching around 200m above the sea meaning that species endemic to each island had to adapt to their very different environment and resources. In fact, on Wolf island, the resources are so scarce that the local finches have evolved into vampires, that adapted to suck blood out of other larger birds to survive.

What this means for evolution is that having different islands to evolve on promotes diversity, which is a feature that all optimization algorithms need, to prevent premature convergence on local optima.

Therefore, running n different evolutionary algorithms (with the same fitness function) on n different machines may be as efficient (or even more efficient) than deterministically parallelizing a single algorithm on n machines.

7.2.2 Interconnecting the islands

When parallelizing on n different isolated islands may yield n different results, interconnecting the islands once in a while is very interesting. Indeed, it may happen that an island gets stuck into a local optimum. When this happens, receiving an individual from another island may allow the population to diversify again in order to find a better result.

The case study is the following. If, after an island is stuck in a local optimum, it receives an individual from another island, then two cases are possible:

1. The incoming individual has a lower fitness value than the local individuals. In this case, the individual will not be used for recombination and it will very probably be removed from the population during the "reduction" phase, when creating the new population through Darwinian selection and the island will remain stuck in its local optimum.

2. The incoming individual has a better fitness value than the local individuals. In this case:

   (a) it will be very often selected among parents to create new children

(b) it will be selected to survive to the next generation and hang around until the average genotype of the island's population has migrated towards the genotype of the good immigrant.

Migrating towards the genotype of the good immigrant means that the island that was stuck into a local minimum will increase its diversity, therefore increasing the exploration potential of the algorithm.

So an island algorithm optimises the Exploration *vs* Exploitation ratio: islands exploit local optima and get stuck until they receive a potentially good individual from some other island, that will get them out of their local optimum, giving them the opportunity to find a better solution while "going towards" the good immigrant.

The island model reduces variability of results, even if several islands are used on the same machine.

If n machines are used, supralinear acceleration is often observed [75] in that using n islands on n machines will allow the algorithm to find good fitness values m times faster, where $m > n$ (which is normally impossible in deterministic parallelization.

Part II

Contribution: algorithms for non-FFT harmonic analysis

Chapter 8

Proposed evolutionary approach to tackle harmonic analysis

8.1 Description of the approach

Following the principle of Occam's razor, the first idea to see how artificial evolution could tackle harmonic analysis was to start with using the simplest algorithm we could think of while nevertheless integrating our understanding of the semantics of the problem. This first algorithm served as a basis for experimentation, that subsequently led to the optimisation of its parameters.

The first version yielded interesting results that encouraged us to finely tune the different parameters of the algorithm, to understand its potential for generic harmonic analysis, with FT-ICR as a real-world application.

Then, this approach led us to propose and test some improvements that will be described at the end of this part.

8.1.1 Short description of the EASEA platform

For this work, we used the EASEA evolutionary computing platform[1], that was created back in year 2000 [76] available on SourceForge and GitHub.

EASEA (EAsy Specification of Evolutionary Algorithms) is a software platform dedicated to evolutionary algorithms that since 2008, automatically parallelizes EAs on parallel architectures, that range from a single GPGPU equipped machine to multi-GPGPU machines, to a cluster or even several clusters of GPGPU machines.

The EASEA platform was initially designed to assist users in the creation of state of the art evolutionary algorithms [76]. It is designed to produce an evolutionary algorithm from a problem description or specification. This specification is written in a C-like language, that contains code for the genetic operators (crossover, mutation, initialization and evaluation) and the genome structure. From these functions, written into a *.ez* file, EASEA generates a

[1] `https://easea.unistra.fr`

complete evolutionary algorithm with potential parallelization of evaluation over GPGPUs, or over a cluster of heterogeneous machines, thanks to the embedded island model discussed in section 7.2.

The generated source file for the evolutionary algorithm is user-readable. It can be used as-is, or as a starting point, to be manually extended by an expert programmer.

The EASEA platform implements not only all different kinds of genetic algorithms, evolution strategies, but also genetic programming and other stochastic algorithms such as CMA-ES, multi-objective optimisation algorithms (NSGA-II, NSGA-III, ASREA, Fast-EMO) but also quantum-inspired evolutionary and particle swarm algorithms such as QAES, QPSO and others.

For multi-objective evolutionary algorithms (MOEAs), a specific stochastic ranking method has been developed, which can be parallelized without impacting quality.

An EASEA evolutionary algorithm is defined by problem-specific pieces of code provided by the user. The genome structure is, of course, the first piece that is needed, followed by genetic operators such as the initialization operator (which constructs a new individual), the crossover operator (which creates a child out of two parents), the mutation operator and finally, the evaluation function that returns a value proportional to how well an individual does on the problem to be solved (fitness). An EASEA source code (`.ez` file) consists in several sections, many of which are dedicated to these problem specific operators. Different papers explain how this is done, including [76, 77, 78, 79, 80].

8.1.2 EASEA parallelization of standard EAs

For single objective algorithms, the initial choice was to parallelize the evaluation step only, because this phase is often the most time-consuming in the whole algorithm. It means that the parallel and sequential versions of an algorithm can be completely identical, including the evaluation step that is then executed on multiple cores.

The result is that a fully sequential code can be run very efficiently on a massively parallel GPGPU card containing thousands of cores.

For multi-objective evolutionary algorithms (MOEAs), specific stochastic ranking methods have been developed, which can be parallelized without impacting quality.

8.1.3 Designing an artificial evolution algorithm to find the parameters of the sines

As seen in the state of the art part, artificial evolution has already been tried on the harmonic analysis problem, but unfortunately, the articles are not complete enough to permit a reproducibility of the results (this will be discussed in section 9.6) so we had to start again from scratch, but this could have been necessary due to the specific hardware available for this PhD, that was described into a journal paper [81].

Algorithm Design

Because the algorithm will run on NVIDIA RTX 2080Ti 4352-core GPU cards that must run many threads per core to efficiently use its specific SPMD spatio-temporal parallel graphic

processor (cf. chapters 1, 2, 3 of [75]) for details on how this specific architecture that can be perfectly exploited by artificial evolution), we have the opportunity to use a very large population size, that can minimize the problem of premature convergence, that is shared by all optimization algorithms.

This in turn will have an influence on the operators design and on the parameters of the algorithm described below.

The evolutionary engine is a real-coded Genetic Algorithm, where individuals are represented as vectors of reals but using a crossover taken from Genetic Algorithms.

It is an extremely simple algorithm, nearly taken from the book. Its performance therefore shows the power of the evolutionary approach, even though obtaining the presented results necessitated a very large number of runs to finely tune its different parameters.

We implemented two versions of the algorithm. One for determining the coarse isotopic distribution (finding isotopes with different numbers of neutrons) and the other one for determining the fine isotopic distributions (finding isotopes with identical number neutrons, but with neutrons attached to different atoms with different energy levels, leading to different mass due to Einstein's $E = mc^2$ equation). The objective of each version is different. In the coarse mode, we are interested in determining the parameters for the main peaks. In fine structure we try to determing peaks very close in frequencies. Therefore, some parameters in the algorithm differ depending on the version. The difference is mainly in the mutation and evaluation function as well as the initialialization of the parameters.

Individual representation

Because the objective is to find the parameters of k sines composing the signal, the GA encodes those parameters into a vector of IEEE754 double precision floating point values semantically grouped by 4: $\{d_1, a_1, f_1, p_1, ..., d_k, a_k, f_k, p_k\}$, where d, a, f, p are respectively the damping coefficient, amplitude, frequency and phase of each of the k sines.

We do not use a global damping coefficient for the signal because different molecules will be damped differently. Therefore, if having a specific damping parameter adds a 4th dimension per sine, (meaning that for 6 sines, the problem will be a 24 dimensions problem compared to a 19 dimensions problem if we had used a common damping coefficient + amplitude, frequency and phase for each of the 6 sines), it will allow the algorithm to better model the data.

The genome of an individual is therefore defined as:

```
GenomeClass {
  float Sin[NB_SIN][4];//[0]=damping coefficient, [1]=amplitude, [2]= frequency, [3]=
      phase
}
```

Using real values may have suggested to use an Evolution Strategy evolutionary engine, but the strong semantics created by the different sines rather pointed towards Genetic Algorithm genetic operators.

Crossover

Evolution Strategies [82] are quite good at solving continuous real-valued problems that are not well known, but the price to pay is that they mostly rely on auto-adaptive or covariance matrix adaptation mutation operators [83], meaning that children are not created

through the interaction of two (or more) parents *via* recombination. This makes Evolution Strategies (that typically do not use crossovers) behave more like a parallel search of single individuals, where the only "interaction" that occurs between individuals is the selection operator that determines which of the $(\mu+\lambda)$ individuals will make it to the next generation.

But in the harmonic analysis problem, we know from the structure of the genome that the values represent damping, amplitude, frequency and phase parameters of the different sines, so we can use 1-point crossover with a non-disruptive locus, carefully chosen in between sines, so as to create an interaction between 2 parents, with a potential for added emergence in the evolution of the individuals.

In EASEA, the crossover produces only 1 child (as is now standard in modern evolutionary algorithms, to avoid particular cases with odd number populations) out of 2 parents, chosen with replacement.

Mutation

The mutation operation is applied to all created children. It locally explore around a local minimum. The sines are selected for mutation with a probability linked to the number of sines: `fpMut` (probability to mutate each sine) `=3/(number of sines)`.

Mutation in coarse mode

For coarse isotopic determination (finding the peaks for isotopes of different neutron numbers), the mutation rates are different than for fine isotopic determination (finding the peaks for isotopes of identical neutron numbers):

```
fmutator_amp_rate = 1.0;
fmutator_freq_rate = 1.0;
fmutator_ph_rate = 0.5;
fmutator_dec_rate = 0.233;
```

All parameters are mutated in a range defined by the following parameters:

```
fintensity_delta_amp = (fMAX_AMP-fMIN_AMP)/(2.0*128);
fintensity_delta_freq = (fMAX_FREQ-fMIN_FREQ)/(2.0*4);
fintensity_delta_phase = (fMAX_PH-fMIN_PH)/(2.0*8);
fintensity_delta_decay = (fMAX_EXP-fMIN_EXP)/(2.0*16);
```

where:

- fMIN_AMP and fMAX_AMP are the lower and upper values of the [fMIN_AMP, fMAX_AMP] interval in which sine amplitudes are sought,
- fMIN_FREQ and fMAX_FREQ are the lower and upper values of the [fMIN_FREQ, fMAX_FREQ] interval in which sine frequencies are sought,
- fMIN_PH and fMAX_PH are the lower and upper values of the [fMIN_PH, fMAX_PH] interval in which sine phases are sought, and
- fMIN_EXP and fMAX_EXP are the lower and upper values of the [fMIN_EXP, fMAX_EXP] interval in which sine phases are sought.

Mutation in fine mode

For the fine isotopic case, the mutation rates for each parameter are given by:

```
fmutator_amp_rate = 0.8;
fmutator_freq_rate = 0.6;
fmutator_ph_rate = 0.4;
fmutator_dec_rate = 0.233;
```

All parameters are mutated in a range defined by following parameters:

```
fintensity_delta_amp = (fMAX_AMP-fMIN_AMP)/(2.0*128);
fintensity_delta_freq = (fMAX_FREQ-fMIN_FREQ)/(2.0*16);
fintensity_delta_phase = (fMAX_PH-fMIN_PH)/(2.0*8);
fintensity_delta_decay = (fMAX_EXP-fMIN_EXP)/(2.0*16);
```

Crossover probability and Mutation probability are set to 1, meaning that children are all created by going through the crossover and mutation functions. However, as described above, the mutation is probabilistic on a per-gene probability, meaning that it is possible that a child does not undergo mutation, even if the mutation function was called.

The very large population size (128k individuals) is made possible by the massive parallelism offered by the GPGPU card. Using such a large population provides for a very good exploration of the search space meaning that it is possible to create a convergent algorithm, that does not need to elaborately fight against premature convergence.

GPU parallelization

The EASEA language automatically parallelizes artificial evolution algorithms on NVIDIA GPGPU cards when the source file (here `sinusit.ez`) is compiled with the `-cuda` option. For this experiment, the hardware is an NVIDIA GEFORCE RTX2080 TI GPGPU card. The way the automatic parallelized is performed is described in many EASEA papers and a book [75, 84, 85].

Merging the sines

When the sines are too close to each other, we merge them into one sine.

The sines are merged using the following formula:

$$\begin{aligned} A\sin(\omega t+\alpha)+B\sin(\omega t+\beta) = \sqrt{(A\cos\alpha+B\cos\beta)^2+(A\sin\alpha+B\sin\beta)^2} \\ *\sin\Big(\omega t+\arctan\big(\frac{A\sin\alpha+B\sin\beta}{A\cos\alpha+B\cos\beta}\big)\Big) \end{aligned} \tag{8.1}$$

For fine isotopic determination, maximum similarity between frequencies of the sines before merging is $epsilon\_freq = 0.00000015$. Whereas for coarse isotopic determination, maximum similarity between freq. and phase of the sines before merging is $epsilon\_freq = 0.00015$. A precision of around 7×10^{-4} is needed to distinguish between the main peaks. But a precision of 2×10^{-6} is needed for the closest fine isotopic structure. We make sure that $epsilon\_freq$ is smaller than the smallest difference between close peaks. This fixes the maximal resolution of the algorithm.

If the sine obtained after merging is out of boundaries for some parameters, a random sine is created instead of merging.

Dependence of phase on frequencies

It is known that there is a quadratic dependence relation between phases and frequencies as discussed in Chapter 3 and in [15]. This relationship is linear in a short interval. Because it is easier to find major isotopes (with a different number of neutrons) than fine isotopes (same number of neutrons, but attached to different atoms), we found a linear dependence relationship of phases on frequencies using the phases of the six main peaks (equation 8.2). The coefficient of determination between the phases and frequencies was 0.996. This dependence was taken into account in the initializer of phase parameter for fine isotopic peaks.

$$phase = 554.7650323280 - 2058.3263117462 * frequency \tag{8.2}$$

This is a major breakthrough of the evolutionary approach. Indeed, with standard Fourier Transform, phase is not available, or rather, it is possible to find it but the process to find it is very long and difficult.

A great advantage of the evolutionary approach is that it finds the phase with a good enough precision on the coarse isotopic determination to be able to use it to help finding the fine isotopic determination.

Individual initialization

The individual initialization is performed by using values for each of the sines within `[MIN_AMP, MAX_AMP]`, `[MIN_FREQ, MAX_FREQ]`, `[MIN_PHASE, MAX_PHASE]`, `[MIN_EXP, MAX_EXP]` intervals that contain known ranges for the simulated data.

```
//Boundaries for amplitude, frequency, phase and exponential decay
double fMIN_AMP = 200.0;
double fMAX_AMP = 120000.0;
double fMIN_FREQ = 0.26;
double fMAX_FREQ = 0.27;
double fMIN_PH = 0.0;
double fMAX_PH = 6.283185308;
double fMIN_EXP = 7.0;
double fMAX_EXP = 11.5;
```

The boundaries mentioned above were used with coarse version of the algorithm. For fine version, the amplitude and frequency ranges were shortened depending on which isotopic peak was in interest.

Initializer in coarse mode

```
  for(int i=0; i<nNB_SIN; i++){
    Genome.Sin[i*4+0]=random((double)fMIN_AMP, (double)fMAX_AMP);
    Genome.Sin[i*4+1]=random((double)fMIN_FREQ, (double)fMAX_FREQ);
    Genome.Sin[i*4+2]=random((double)fMIN_PH, (double)fMAX_PH);
    if(bNO_DECAY==1){
      Genome.Sin[i*4+3]=100.0;
    }else{
      Genome.Sin[i*4+3]=random((double)fMIN_EXP,(double)fMAX_EXP);
    }
  }
```

Initializer in fine mode

```
for(int i=0; i<nNB_SIN; i++){
  Genome.Sin[i*4+0]=random((double)fMIN_AMP, (double)fMAX_AMP);
  Genome.Sin[i*4+1]=random((double)fMIN_FREQ, (double)fMAX_FREQ);
  Genome.Sin[i*4+2]=554.7650323280 -2058.3263117462*Genome.Sin[i*4+1]; //random((double
      )fMIN_PH, (double)fMAX_PH);
  if(bNO_DECAY==1){
    Genome.Sin[i*4+3]=100.0;
  }else{
    Genome.Sin[i*4+3]=random((double)fMIN_EXP,(double)fMAX_EXP);
  }
}
```

In the case of fine isotopes, the phase parameters were initialized using the linear equation obtained from the result of 6 main peaks 8.2.

Evaluation function

The evaluation function is given by L^2 norm:

$$score = \sum_{i=1}^{n} \left(\frac{|f(x_i) - s_i|^2}{n} \right)^{1/2} \tag{8.3}$$

where $f(x_i)$ is the predicted output, x_i is the time and s_i is the expected output.

Study of the effect of parameters

When $fpMut$ is too large, then good mutations on some sines are not noticed since there are bad mutations on other sines. Also, when fpMut is too small, huge number of generations are needed to find the solution. This was tested using 2 sines, 32k points, and 100 generations and the following values were obtained:
$fpMut = 0.1, fitness = 336.44$
$fpMut = 0.5, fitness = 119.95$
$fpMut = 1.0, fitness = 138.69$
One sees that the fitness (error) is better when $fpMut = 0.5$.

Effects of $fmutator\_(amp/freq/ph/dec)\_rate$: when these values are too large, the algorithm is more exploratory on these parameters, but it is then difficult for the algorithm to focus on precise values. When it is too small, huge numbers of generations are needed. This was also tested using 2 sines, 32k points, and 100 generations and the following values were obtained:
$fmutator\_freq\_rate = 0.05, fitness = 245.36$
$fmutator\_freq\_rate = 0.233, fitness = 138.69$
$fmutator\_freq\_rate = 1.0, fitness = 21188.04$
One sees that the fitness (error) is better when $fmutator\_freq\_rate = 0.233$.

Effects of $fintensity\_delta\_(amp/freq/ph/dec)$: when these values are too large, mutations are rarerly useful. When they are too small, huge numbers of generations are required. This was also tested using 2 sines, 32k points, and 100 generations and the following values were obtained:
$fintensity\_delta\_freq = range/2, fitness = 431.35$
$fintensity\_delta\_freq = range/8, , fitness = 138.69$

$fintensity\_delta\_freq = range/1M, fitness = 367.2$
One sees that the fitness (error) is better when $fintensity\_delta\_freq = range/8$.

Run parameters

In our algorithm we use population size 128k and 2k generations for the coarse structure, 512 generations for fine structure. The mutation and crossover probabilities are chosen to be 1. This means that all the children that are created will go under crossover and they all have a chance of being mutated. The mutation function is always called but this does not mean that the child will get mutated. The mutation rates used for each parameter are described previously in the Mutation section.

The run parameters used in our algorithm are shown below:

```
\Default run parameters :          // Please let the parameters appear in this order
  Number of generations : 2048   // NB_GEN
  Time limit: 0                    // In seconds, 0 to deactivate
  Population size : 131072
  Offspring size : 100%
  Mutation probability : 1         // MUT_PROB
  Crossover probability : 1        // XOVER_PROB
  Evaluator goal : minimise        // Maximise
  Selection operator: Tournament 20
  Surviving parents: 100%          // percentage or absolute
  Surviving offspring: 100%
  Reduce parents operator: Tournament 2
  Reduce offspring operator: Tournament 2
  Final reduce operator: Tournament 20

  Elitism: weak        // Weak (best of parents+offspring) or Strong (best of parents)
  Elite: 1
```

These parameters are very important. The execution time of the algorithm depends on the population size, number of generations, as well as sample size and number of sines we choose to find.

For the coarse structure, we used 2k generations whereas for fine structure we used 512 generations. The run time for coarse structure was about 6 hours using 32k points, whereas with fine structure, the run time varied from 2 days to 14 days depending on how many sines we were trying to detect. The sample size also was much larger for fine structure, 2M, compared to 32k with coarse structure.

GRS sampling The sampling modes used in the algorithm are: full sample (uniform), GRS by replacement and GRS by extension (cf. section 4.1).

GRS settings

Here, we define the parameters used in the GRS settings:

nNUS_GRS_GENS: Number of generations elapsed before expanding the GRS points.

nNUS_GRS: Ratio of acquired transient over sampled transient (power of 2).

nNB_GRS_GENS: Defines the number of points to introduce in GRS every nNB_GRS_GENS generations (power of 2)

nNUS_GRS_SAMP: Defines the number of points to introduce in GRS every nNB_GRS_GENS generations (power of 2). For example, with nNUS_GRS_SAMP=2 we introduce nNB_SAMPLES/2 new points at every GRS step.

The number of current generations to pass before changing the GRS points (counter initialization) is given by: int nGENS_BEFORE_GRS_CHANGE=nNB_GRS_GENS;

8.2 Difference between coarse and fine isotopic analysis

The difference between coarse and fine isotopic analysis has been evoked above and the different versions sets of parameters have been described.

The reason why there are two versions is that the objective of coarse and fine isotopic analysis is different.

In the coarse analysis, we are interested in finding damping, amplitude, frequency and phase (which cannot be found easily with Fourier Transform methods due to the fact that ICR machines do not provide the imaginary part of the signal).

Knowing the phase precisely enough is very important, because it makes it possible to find the fine mass differences between isotopes having the same number of neutrons, but whose neutrons are not attached to the same atoms. Indeed, the energy bond is different when an additional neutron is attached to an Hydrogen atom or to an Oxygen atom, and Einstein's $E = mc^2$ function tells uss that Energy equals mass times the square of the speed of light.

So an isotope where a supernumerary neutron is part of a Hydrogn atom nucleus (^{2}H deuterium) will have a different mass than another isotope with the same number of neutrons, but where the extra neutron is part of an ^{17}O Oxygen atom nucleus, but of course, the mass difference will be much smaller than between two isotopes with a different number of neutrons.

This has an influence on the number of data points we currently need to use for both analyses: for determining the fine isotopic structure, we need between 1 and 2 Mega points instead of 32k for determining coarse isotopic structure.

But thanks to the relationship between phase and frequency, knowing the phase precisely enough (thanks to the evolutionary algorithm) makes it possible to narrow down the frequency range of the sines we are looking for in fine isotopic analysis.

8.3 Speedup obtained by parallelizing the sinus-it algorithm

8.3.1 Speedup by GPU parallelization

The EASEA language automatically parallelizes artificial evolution algorithms on NVIDIA GPGPU cards when the source file (here sinusit.ez) is compiled with the -cuda option. For our experiment, the hardware is a PC with an Intel Core i7-9700K overclocked to 4.6GHz CPU and an NVIDIA GEFORCE RTX2080 TI GPGPU card. By parallelizing the evaluation on the GPGPU card, execution time for 10 generations drops from 1199.17 seconds to 3.04s seconds, meaning that an acceleration of $\times 394,46$ is obtained when comparing performance of the sequential and the parallel version of the same algorithm. The way the automatic parallelized is performed is described in many EASEA papers and a book [75, 84, 85].

8.4 Speedup by Island parallelization

In this section, we present how supra-linear acceleration can be achieved to solve any kind of continuous, discrete and combinatorial problems as described in our journal paper [81]. Harmonic analysis problem is shown as an example on a simple system made of 4 computers that exhibit supra-linear acceleration.

8.4.1 Discussion on the computation time between island and isolated runs

In this section, two main sets of experiments are presented, in which harmonic analysis is performed by using:

a) an isolated artificial evolution algorithm with 262 144 individuals.

b) 4 islands with 65 536 individuals each, loosely coupled over an Ethernet network, exchanging individuals every second.

Individuals are composed of the parameters for 10 sines, *i.e.* 30 single precision real values (120 bytes), so the total load on the network between the 4 machines is to transfer 480 bytes per second, without any synchronization between the machines (incoming individuals are integrated at the next new generation, where they replace a bad individual in the accepting island).

Fitness evolution of the island model *vs* isolated runs for similar end results

What can be seen in Fig. 8.1 is that the best fitness of the islands (that represents the error between the evolved sum of sines and the obtained data that must be minimized) improves much faster than the fitness of the isolated runs.

This is expected as 4 machines have 4 times the computing power of one machine. The big question is whether loosely coupled machines can cooperate well enough to be able to exhibit linear acceleration.

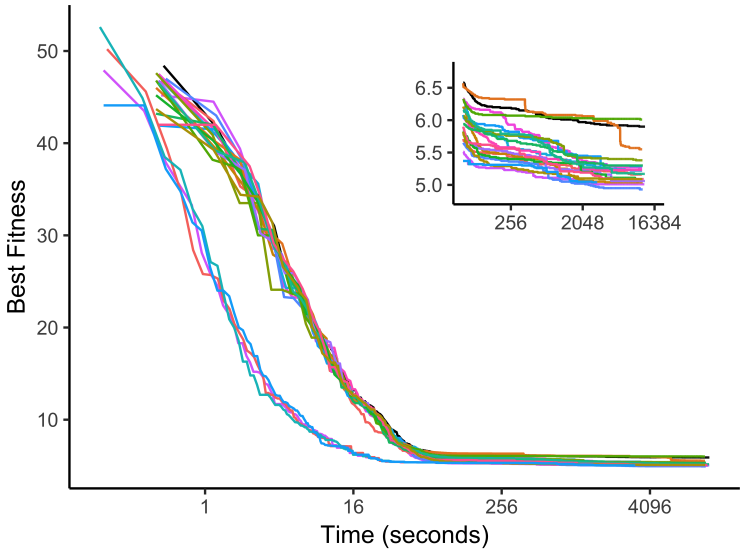

Figure 8.1: Evolution of the fitness of the island model (left curves) *vs* isolated runs (right curves). Lower values means better data fitting. *Nb*: the x-axis is given in $\log_2$ scale.

For further comparison of the island runs with single runs, qualitative acceleration plot is produced as in Figure 8.2. The acceleration is the ratio of the mean time that takes to obtain the given fitness values for the single runs to island runs. The plot shows the acceleration value of approximately 4 until value 10. It should be noted that the acceleration value for fitness value 5.5 is due to the fact that island runs obtained that value much faster compared to the single runs and thus the ratio of the mean time to obtain that value is much larger compared to the other values. The overall result of this plot shows that island model obtains the same quality as the single runs with 4 times faster speed.

8.4.2 Defining qualitative acceleration

Usually, the maximum acceleration that can be obtained by parallelizing an algorithm is described in terms of Amdahl's law [86]:

$$A = \frac{s + p}{s + p/N} \tag{8.4}$$

with A the maximum expected acceleration for N the number of processors, s the sequential time on one processor and p the sequential time of a perfectly parallelizable piece of code.

Gustafson's law [87] shows how this equation is too restrictive, as what must be taken into account is proportional acceleration, and not absolute acceleration:

$$A_p = \frac{s' + p'N}{s' + p'} \tag{8.5}$$

with A_p being Gustafson's proportional acceleration, N the number of processors and s' and p' the sequential and parallel time taken on the parallel system.

However, both acceleration metrics refer to the *number* of instructions executed per second in sequential or parallel systems. We will say that these metrics define quantitative acceleration.

However, on real-world problems, what is interesting is not the number of instructions performed per second, but the *quality* of the results obtained for a given run time.

Because the island model to interconnect different machines is a complex system with emergent properties[2], we are interested in measuring acceleration obtained by the island model *vs* the isolated model not in terms of number of instructions per second but in term of necessary time to obtain a similar quality, as measured by the error to be minimized between the acquired signal and the sum of sines that is evolved to model the signal.

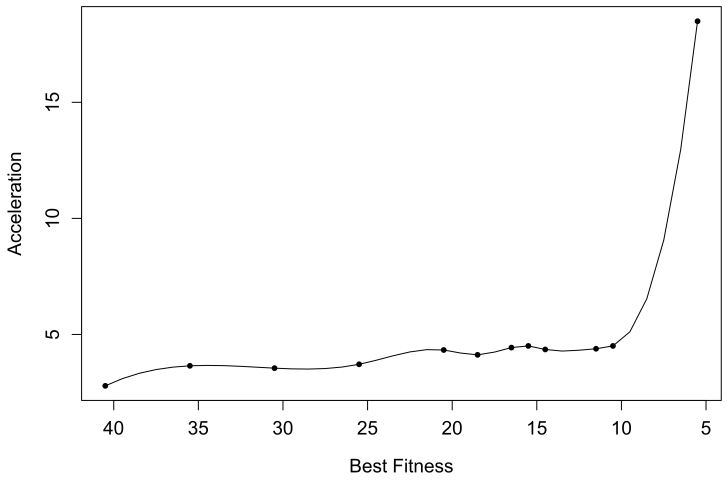

Figure 8.2: Acceleration plot of island runs with respect to single runs.

We therefore propose a new acceleration metrics that we call *qualitative acceleration*, defined by the ratio between the time needed for the island model to obtain an error value over the time needed for an isolated algorithm to obtain the same error value:

$$A_q = \frac{t_{im}(\epsilon)}{t_{is}(\epsilon)} \tag{8.6}$$

With A_q the quantitative acceleration, $t_{im}(\epsilon)$, the time for the island model to reach error value ϵ and $t_{is}\epsilon$ the time for the isolated algorithm to reach the same ϵ error value.

[2] An emergent system can be defined as a number of autonomous entities in interaction, that create several levels of collective organization leading to emergent (or immergent) behaviour. In short complex systems can be summarized by Aristotle's famous statement: the whole is more than the sum of the parts.

Qualitative acceleration can be plotted by taking slices from the Best Fitness over Time curve (Fig. 8.1) and having on the x axis the values attained by both experiments and in y axis the time ratio between isolated model (reference) and island model.

The result is Fig. 8.2, where we can see the time ratio between both models. The time to find value 40.5 is roughly similar for both models. however, it is about 2.5 faster to find value 35.5 with the island model than with the isolated model. This represents an infra-linear acceleration as 4 machines are only 2.5 times faster to find the same result.

However, things get interesting as it becomes more and more difficult to find low error values. From error value 20 to 10, linear acceleration (≈ 4) is achieved.

Then, something really nice happens: *4 machines obtain value 5.5 nearly 20 times faster than one single machine, with an identical population size.*

Values 5.5 was chosen as the lowest value to compare both models because it was the only value found by at least 5 isolated algorithms (the 13 other could not find this error value).

8.4.3 Defining supralinear acceleration

Linear acceleration is defined as obtaining and $\times N$ speedup with N machines. On Fig. 8.2, this would appear as a horizontal line with y value 4.

Super-linear acceleration would be represented by a horizontal line with a y value greater than 4. However, this is not what is observed on Fig. 8.2.

Beyond value 10 (when it becomes really difficult to find better results), the island model becomes much faster as its individuals still benefit from efficient crossover operators, because the rare individual migrations between the islands prevented their population from converging to a local optimum, therefore maintaining genetic diversity between the individuals. Indeed, after a population algorithm has converged (*i.e.* when all the individuals are clones, stuck in a local optimum, sharing identical genes), crossover is ineffective as children will be identical to the parents. This is what is happening in the panmictic isolated 262144 individuals islands, that can only rely on mutation to find better results below error value 10.

We therefore define supralinear acceleration as a non-constant positive acceleration evolution, which keeps improving well beyond superlinearity. Indeed, observed acceleration on the shown example stops at $\approx 20\times$ with only 4 machines, but this is only due to the fact that we stopped qualitative comparison at value 5.5.

Supralinear acceleration will still increase until the island model finds values that cannot be obtained by isolated panmictic runs, in which case acceleration will *de facto* increase to infinite values.

8.4.4 Discussion on PARSEC machines *vs* standard supercomputers

Standard supercomputers are nowadays often made of many independent computers hosting one or several GPGPU cards. Whereas parallelization on the GPGPU cards is difficult to do efficiently for standard algorithms (because they require to be able to identify tens of thousands of independent threads in the algorithm that must run in SIMD mode), the very

high clock frequency speed shows achieved nowadays by CPUs show that it is impossible to perfectly synchronize different machines due to Einstein's principle of locality exposed in his 1905 and 1935 papers.

Attempting to obtain linear accelerations with the number of computers for continuous operation is therefore a hopeless quest that is bound to fail.

Evolutionary algorithms are generic solvers that can tackle all kinds of difficult problems, be they continuous, discrete or combinatorial. They are embarrassingly parallel, meaning that they parallelize perfectly on SIMD GPGPU cards, which more or less impose that all cores execute the same instruction at the same nanosecond, over tens of thousands of threads. This requires absolute time that is achievable inside a single GPGPU chip, but unachievable in between CPUs.

However, because evolutionary algorithms can use Transfer Learning, exchanges between CPU is not limited to meaningless data, that requires millions of bytes to describe a simple cat (photo). Exchanging individuals between islands run on different CPU makes it possible to achieve not only linear or super-linear acceleration, but supra-linear acceleration on several machines, by exchanging but a few bytes per second (480 in the presented example).

Chapter 9

Sinus-it on simulated data

9.1 Coarse and fine isotopic structure of molecules

For the analysis of synthetic data, our objective is to determine the coarse and fine isotopic structures of Substance P (*cf.* `https://en.wikipedia.org/wiki/Substance_P`): of chemical formula $C_{63}H_{98}N_{18}O_{13}S$.

Coarse isotopic structure of Substance P: As any substance, Substance P is a molecule that can be created out of different isotopes of the atoms composing the molecule.

When a sample of Substance P is analysed, thousands of molecules are inserted in the ICR machine, so among these, some molecules are made of the natural isotopes of each atom and some others of different isotopes for different atoms, *i.e.* atoms with extra neutrons. The result is that there are different peaks for the different isotopes of the same molecule (*cf.* Fig. 9.1 left), because a molecule containing one or more extra neutrons (a different isotope) will weigh more than a standard molecule. So there are different observed peaks in the mass spectrometry of Substance P:

1. The first peak corresponds to the mass of the standard molecule multiplied by the number of molecules of this kind.

2. The second peak corresponds to the mass of an isotope of Substance P with *one* extra neutron multiplied by the number of isotopes of this kind.

3. The third peak corresponds to the mass of an isotope of Substance P with *two* extra neutrons multiplied by the number of isotopes of this kind,

4. ...

So the coarse analysis of the spectrum will consist of detecting how many isotopes are present and how many molecules of each isotope are present in the analysed sample of Substance P.

Fine isotopic structure of Substance P: The first isotope of Substance P is a molecule in which one of its atoms has an extra neutron. But because the molecule is made of atoms of different kinds, the extra neutron could be part of a carbon atom (meaning that rather than contaning 63 atoms of standard ^{12}C, the molecule would contain $62\times^{12}\text{C} + 1\times^{13}\text{C}$) or a hydrogen atom, or a nitrogen atom, or an oxygen atom or a sulphur atom.

Due to the composition of the nucleus of each atom, the energy bonding the neutrons to the nucleus will be different depending of the kind of atom in which the neutrons are bonded. Einstein's famous $E = mc^2$ equation tells us that energy equals to mass times the speed of light squared. This means that if neutrons are bonded to different atom nuclei with different energy levels, the bond will show a different mass depending on which atom the extra neutron atom is attached to.

If we zoom on a particular peak, some extra-peaks will show next to it, coming from the slightly different masses of isotopes where the extra neutron is attached to a carbon atom or to an oxygen atom, or to a hydrogen atom.

For the second peak, there are only 5 possibilities because Substance P is made of 5 different atoms ($C_{63}H_{98}N_{18}O_{13}S$), *cf.* Fig. 9.1 right. But for the third peak, the two extra neutrons could be attached to 2 different carbon atoms, or to one carbon atom and one oxygen atom, or even... both neutrons could appear in a single carbon 14 atom isotope. So the number of different possibilities grows combinatorially with the number of additional neutrons.

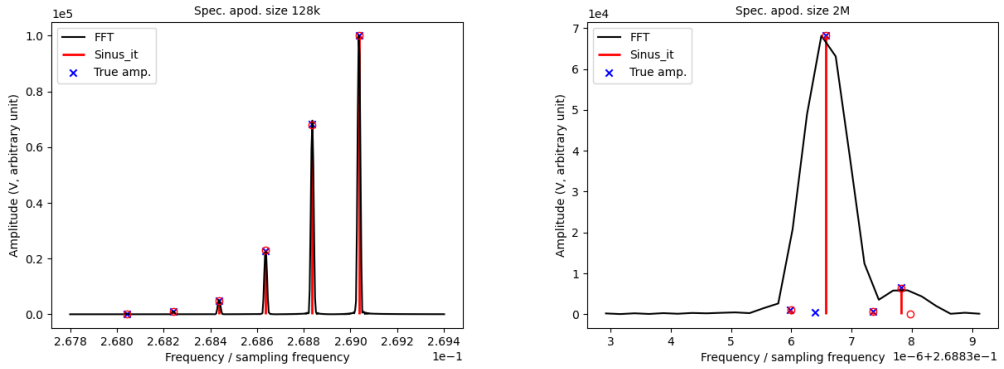

Figure 9.1: Coarse (left) and fine (right) isotopic structure of Substance P. The right figure is a zoom on the second peak of substance P.

Each sub-peak shows how many isotopes are made of which kind of atomic isotopes.

On the importance of noise: the problem with experimental data is that mass spectrometers are not perfect machines (the vacuum in the chamber is not perfect, the sensors are not perfect) meaning that the real data coming out of the machine is noisy.

Exact mathematical approaches such as Fast Fourier Transforms are unfortunately not good at dealing with noise. However, genetic algorithms are stochastic by nature, so they

are less sensitive to noise, but the influence of noise must be tested on the new approach proposed in this PhD.

Artificial / synthetic data generation: A good way to evaluate the quality of a new approach is to test it on some problems whose solutions are already known. What is nice with harmonic analysis is that it is very easy to create data using known sums of sines, to see in a second step if the created harmonic analysis algorithms can find back the sines that compose the data.

It is possible to test the resolution of the approach, to see the capacity of the algorithm to distinguish between very close sines, or to see how well it resists to adding white noise, to determine how many sines can be detected, etc...

When the algorithm is well characterized using artificial data, it is then possible to see how it performs on real data.

This part will first deal with studying the performance of the proposed algorithms on specifically prepared test sets using synthetic data, before testing the approach on real data coming from a real FT-ICR machine.

9.2 Results on simulated data for coarse isotopic distribution

In this section, we present the results using full sample for coarse and fine isotopic peaks. We also study the influence of sample size and noise level. It is worth to mention that double precision was used in the declaration of the parameters of the sinusoid in our algorithm.

The signal is generated in the following form:

$$\text{Signal}[n] = \sum_{k=1}^{K} A_k e^{-\lambda_k n} \cdot \sin(\omega_k \cdot n + \phi_k) + \epsilon \tag{9.1}$$

with n being the sample number, A the amplitude, ω the angular frequency, ϕ the phase, λ is the damping coefficient, K the number of sines and ϵ some added noise.

9.2.1 Performance with noiseless data

We first study the performance of our algorithm on a noiseless data. The data is generated using 6 sines shown in Table 9.1 in coarse mode. We use a portion of the data, with 32k contiguous points. The interest here is to see how our algorithm, sinus-it, would perform compared to FFT when there is no noise present in the data. We repeat the runs 30 times to check the stability of our result. The resulting peaks are presented in the frequency spectrum plot in Figure 9.2. The spectrum plot shows the frequency/sampling frequency on the x-axis and intensities on the y-axis. The blue crosses are the true values, red bars indicate the result of sinus-it and black peaks show the result of apodized FFT. The resulting plot shows that sinus-it can detect all the 6 peaks as well as FFT, however with FFT the precision is not good on the 6th peak.

Parameters	Estimated value	True value	Relative error
Result for sine 1			
Amplitude	99949.6659	100000	5.04E-04
Frequency	0.26903652	0.269036561	1.53E-07
Phase	1.005566156	1	5.57E-03
Damping	10.4698	7.5	3.96E-01
Result for sine 2			
Amplitude	68189.0004	68139.09	7.32E-04
Frequency	0.268836634	0.268836569	2.42E-07
Phase	1.402348015	1.4116509	6.59E-03
Damping	10.6022	11.2	5.34E-02
Result for sine 3			
Amplitude	22833.209	22846.19	5.68E-04
Frequency	0.268636778	0.268636873	3.54E-07
Phase	1.835529646	1.8226883	7.05E-03
Damping	10.7018	8.2	3.05E-01
Result for sine 4			
Amplitude	5024.7617	5024.33	8.59E-05
Frequency	0.268437435	0.268437474	1.45E-07
Phase	2.245845568	2.2331165	5.70E-03
Damping	10.081	10.3	2.13E-02
Result for sine 5			
Amplitude	1021.6414	1022.17	5.17E-04
Frequency	0.268240617	0.268240535	3.05E-07
Phase	2.620985279	2.6384813	6.63E-03
Damping	9.3101	9.8	5.00E-02
Result for sine 6			
Amplitude	221.6469	224.8	1.40E-02
Frequency	0.268042132	0.268041724	1.52E-06
Phase	3.034534143	3.0476992	4.32E-03
Damping	10.4897	8.5	2.34E-01

Table 9.1: True and sinus-it estimated values (with 32k points, noise level 100) for the parameters of the 6 main sines

The relative errors of the 6 sines are presented in the violin plots in Figure 9.3. Our main focus is on the amplitude and frequency errors. As it is seen from the violin plots, as the peak intensity is decreasing as we go from sine 1 to sine 6, the relative errors are increasing which is a normal behaviour as it is becoming harder to get the smaller peaks. However, as shown in violin plots, the resulting relative errors are in the required range to detect the peaks. Since phase is generally not known in FT-ICR MS as it can't be calculated by FFT, the required range is unknown. However, the result that we obtained by sinus-it returns phase parameters with relative errors 10^{-1}.

In reality, we never get perfect data. Every measurement and experiment is corrupted

by an unwanted noise. The real data coming from the FT-ICR machines is always noisy. For this reason, we don't study the algorithm on noiseless data further and we concentrate more on a noisy data in the following sections.

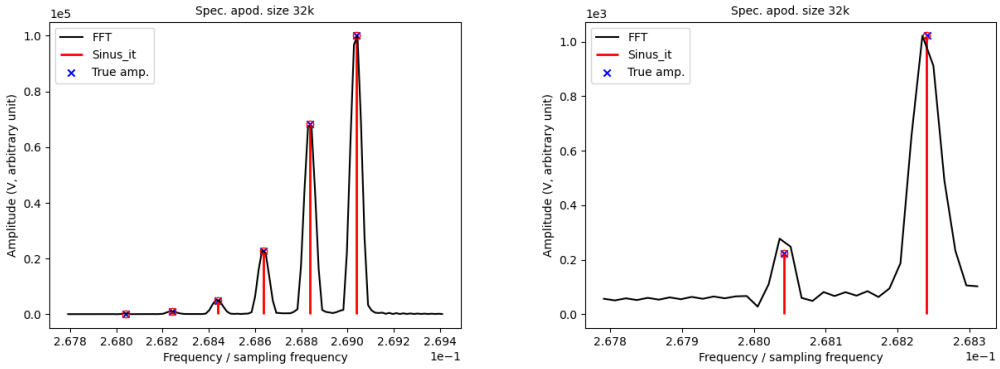

Figure 9.2: Coarse isotopic structure for noiseless data with 32k points (left), zoom on the 5th and 6th peaks (right)

9.2.2 Performance with noise level 100

The signal generator generates transients using the Bruker format (4 bit integer) and damping coefficient which is defined as the percentage of the initial amplitude of the sine at the end of the transient (exponential decay). White noise with a value of 100 is added to the transient.

Because the smallest sine used to simulate Substance P has an amplitude value of 224.8, adding a white noise level of 100 means that the signal/noise ratio (here defined as peak amplitude divided by noise level) is ≈2/1.

Coarse isotopic structure

In these experimental runs, the objective is to determine the 6 main sines that compose the signal using sinus-it. We compare the results obtained by sinus-it with the apodized FFT method.

For coarse isotopic structure, the sines we are interested in are presented in Table 9.1.

Influence of sample size: In order to determine the minimum number of contiguous data points needed to obtain a correct estimation of the signal, we run the genetic algorithm on 2k, 4k, 8k, 16k, 32k, 64k, 128k and 256k contiguously taken data points (Figure 9.4) for 2k generations.

The results of sinus-it showed that 2k and 4k points were not enough to estimate the signal. With 8k points, sinus-it was able to determine all 6 sines whereas FFT could only determine 1 sine. We note that increasing the number of points, increases the precision.

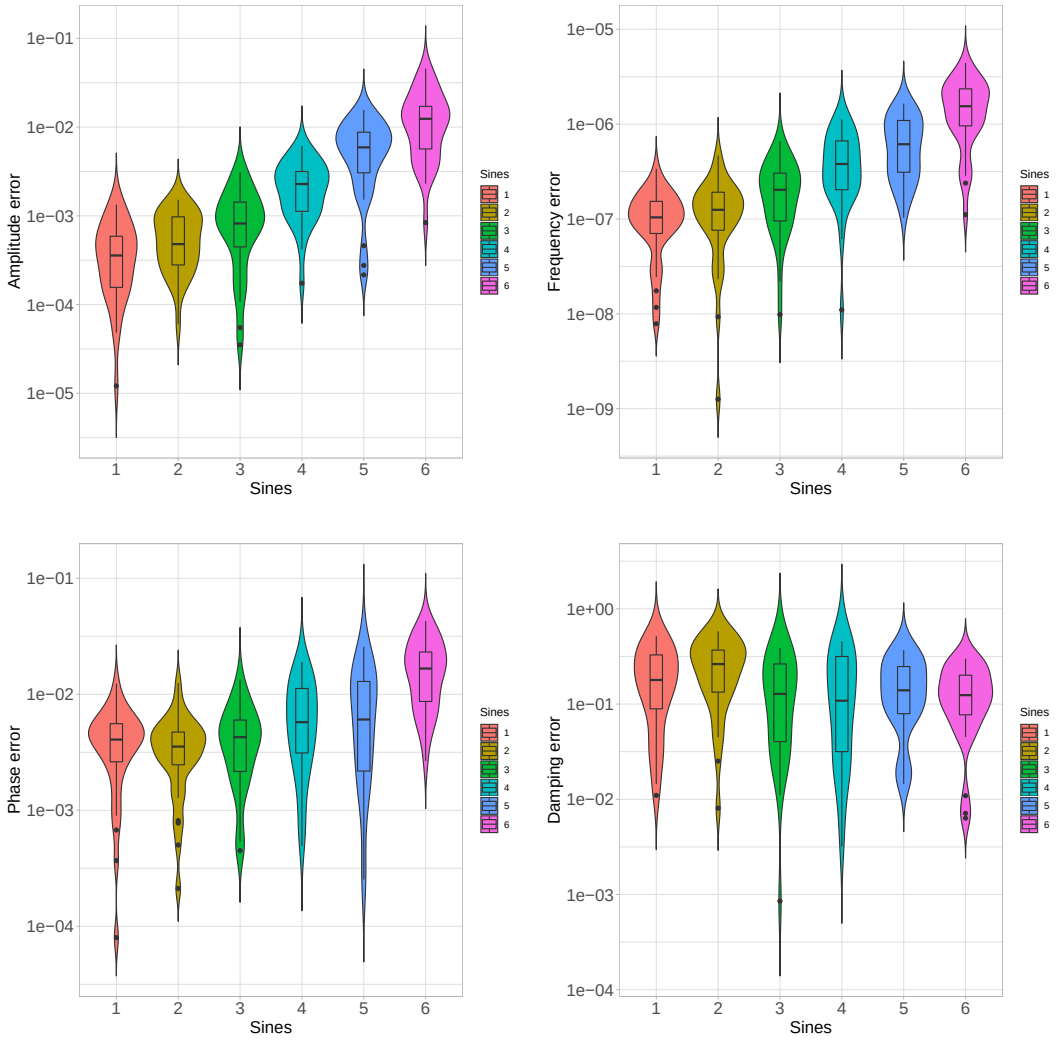

Figure 9.3: Relative errors for noiseless data with 32k points ($\log_{10}$ scale)

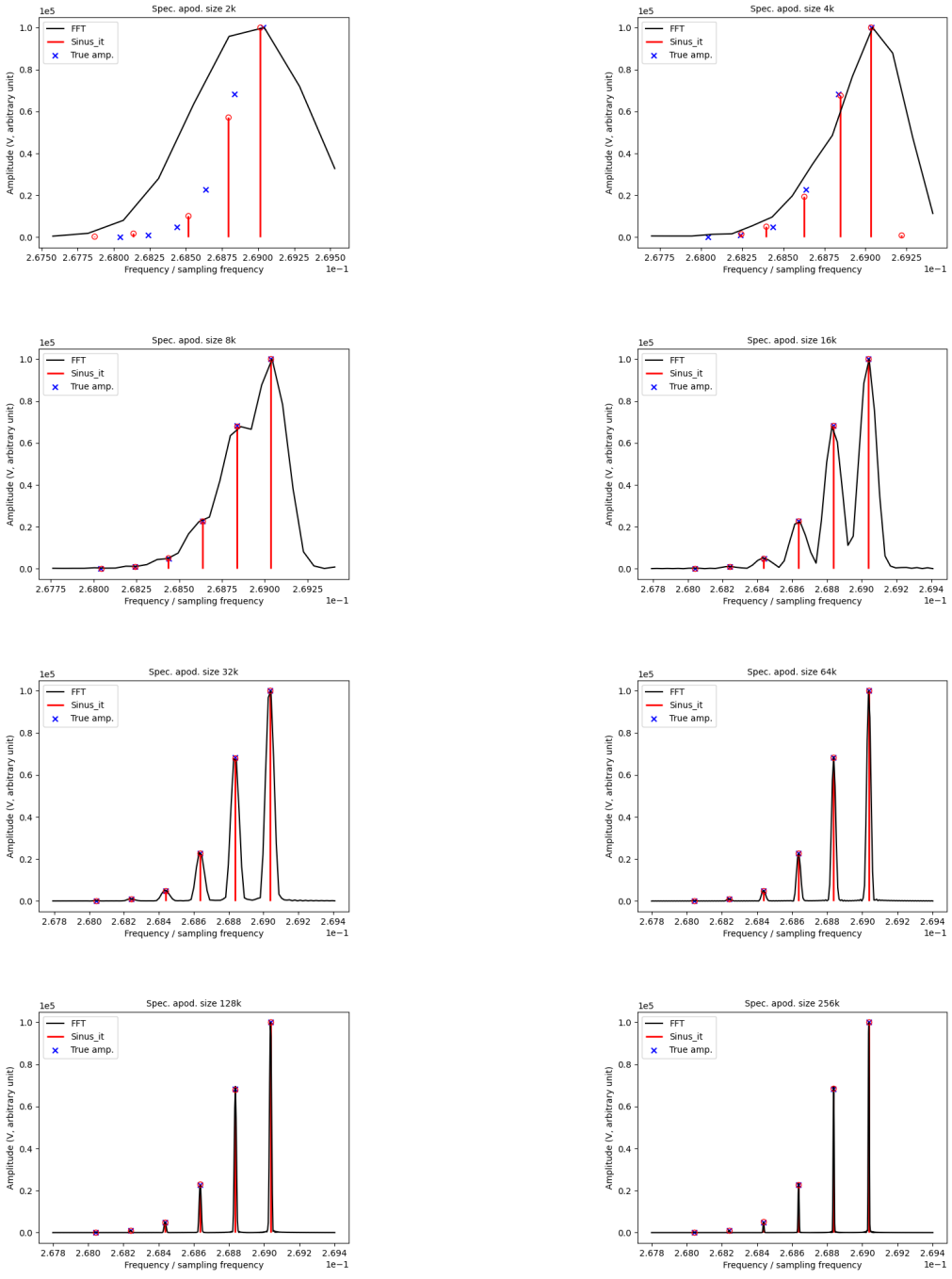

Figure 9.4: Coarse isotopic structure using sample sizes 2k-256k, noise level 100

With 32k points, the sinus-it and FFT both can find all the 6 main peaks of Substance P. Precision gets better as the sample size increases.

For further analysis of coarse isotopic structure, we use 32k points. Even though 8k points also returns good results, with 32k points the precision is better. Table 9.1 presents the estimated and true values for each parameter of the 6 sines obtained with 32k points using sinus-it. The run time over 2k generations was about 6 hours.

Because artificial evolution is stochastic, the results are never the same, so for statistical significance, we repeated the run 30 times with 32k points. The relative errors in amplitude, frequency and phase are displayed in violin plots (Figure 9.5). The wider sections of the violin plots show that most of the values are around that given value whereas the skinnier sections show that there are very few results around those values. The violin plots show larger variability in the amplitude, frequency and phase errors for the 6th sine and larger errors compared to other sines. This is normal, since finding small sines is not very easy and as discussed above. However, all errors are in an acceptable range. There is one outlier in the 6th sine, which is apparent in the plot for frequency errors. For better display, the relative errors are displayed in $\log_{10}$ scale.

9.3 Influence of noise

We test the influence of noise in the performance of our algorithm and compare our results with FFT. As mentioned above, the results presented in the previous sections were based on a noise level of 10^2. We increased the noise level to 10^3, 10^4, 10^5, and 10^6 to see if sinus-it can distinguish signal from noise, where signal/noise ratio is < 1.

With noise level 10^3 (s/n ratio of 0.2), Figure 9.6 shows that we obtain all 6 peaks. FFT can also detect 6 peaks at this noise level, however, the precision is not good for the 6th peak.

With noise level 10^4 (s/n ratio of 0.02), we obtain 5 peaks with our algorithm. The 6th peak is not found (*cf.* Figure 9.7). FFT can also find 5 peaks at this noise level.

With noise level 10^5 (s/n ratio of 0.002), sinus-it can detect 4 peaks even though the noise level is equal to the amplitude of the largest peak (Figure 9.8). Similarly, sinus-it can also detect 4 peaks.

With noise level 10^6 (s/n ratio of 0.0002), as we can see from Figure 9.9, even though the noise is 4 orders of magnitude larger than the intensity of the largest peak, sinus-it still was able to identify the peak positions and intensities correctly for the three peaks. At this noise level, FFT also detects the 3 peaks.

With noise level 10^7 (s/n ratio of 0.00002), both methods can't detect the peaks correctly Figure 9.10.

We conclude that both methods perform similarly when we fix the sample size at 32k and increase the noise level. However, when the noise level was fixed at 100 and sample size varied, sinus-it definitely returned results with high accuracy using 4 times less points than FFT.

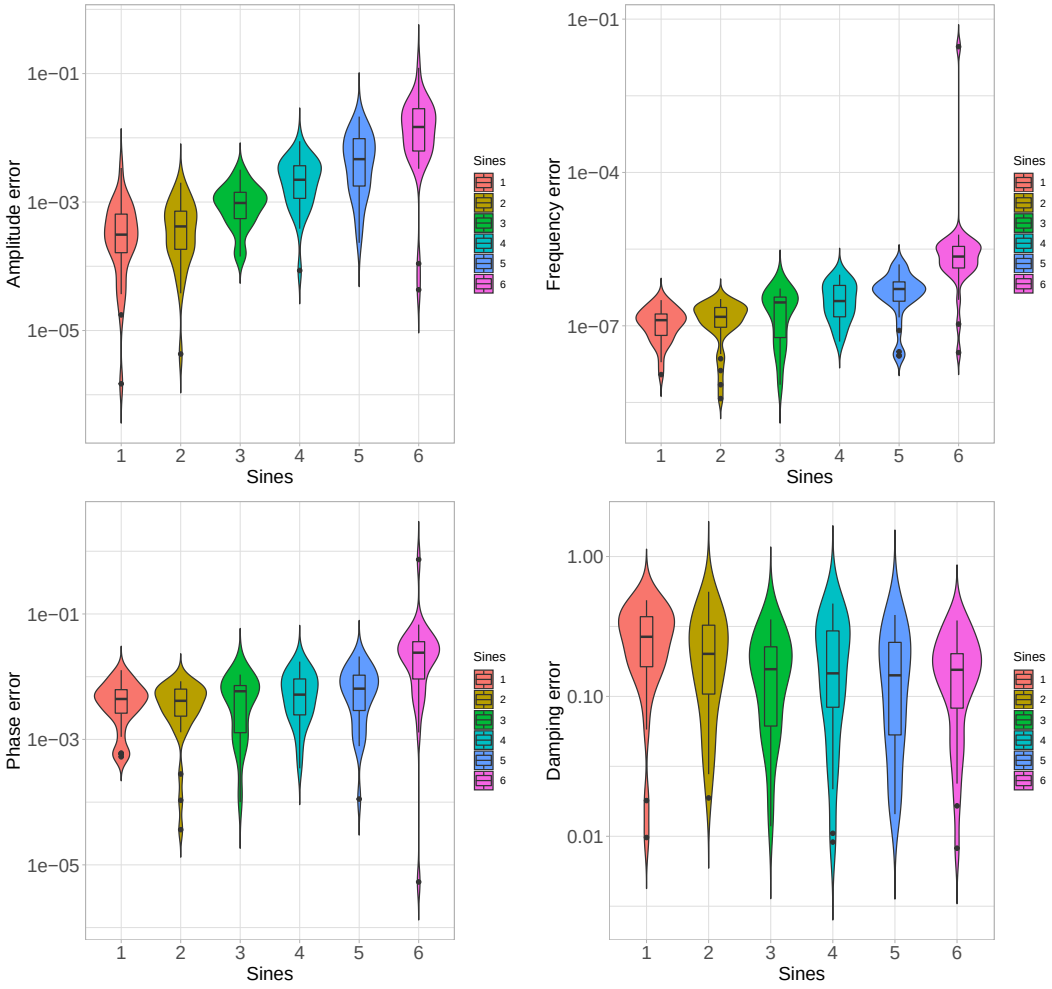

Figure 9.5: Relative errors with 32k points and noise level 100, ($\log_{10}$ scale)

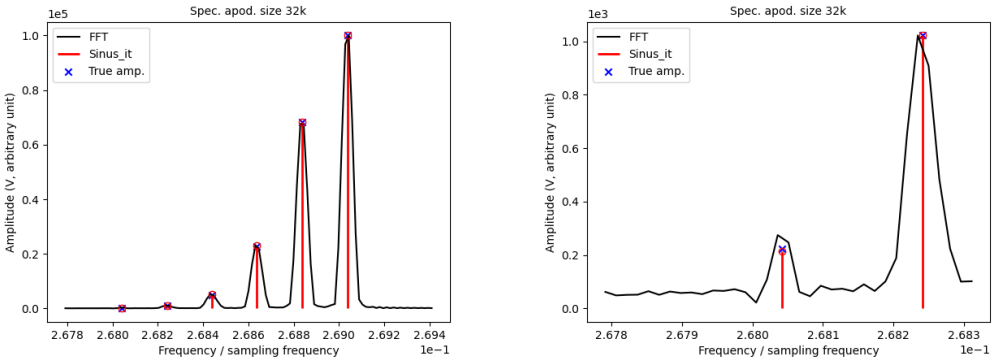

Figure 9.6: Coarse structure with Noise level zoom 10^3, zoom on left peaks (right)

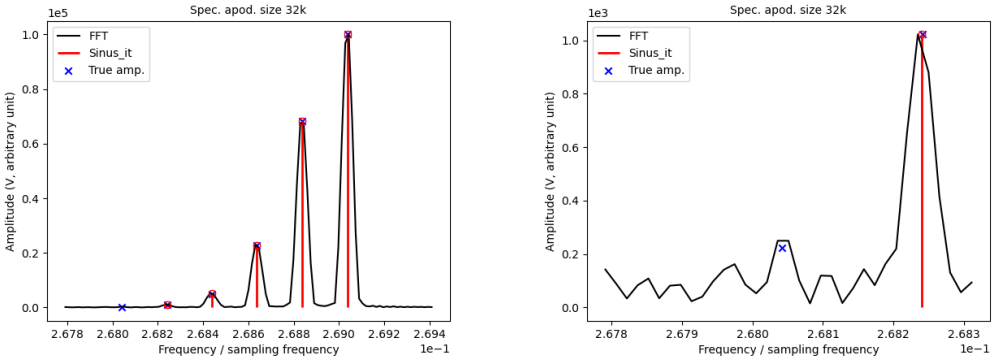

Figure 9.7: Coarse structure with Noise level 10^4, room on left peaks (right)

9.4 True simulated coarse isotopic distribution

In order to determine all major peaks of Substance P, we generated a signal using all 69 sines that are present in our Substance P sample. The objective is to determine the 7 main peaks. Experimental runs were made with 1k, 2k, 4k, 8k, 16k, and 32k uniformly spaced points (Figure 9.11). It was found that at least 8k points are needed to identify the 6 main peaks and 32k points required to find all 7 peaks using sinus-it. FFT however can only detect the first peak with 8k points. With 32k points, it can determine 6 sines only. More points are required for FFT to detect all 7 peaks.

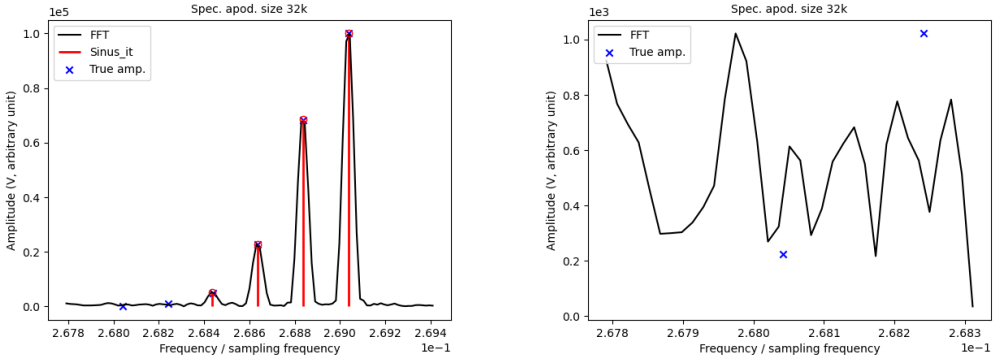

Figure 9.8: Coarse structure with Noise level 10^5, zoom on left peaks (right)

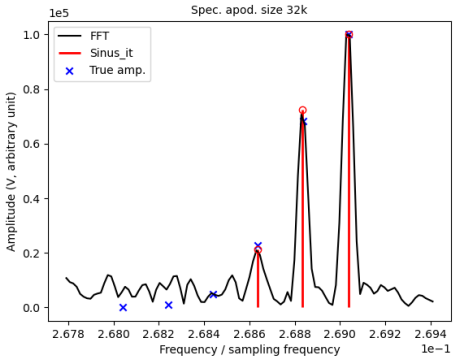

Figure 9.9: Coarse structure with Noise level 10^6

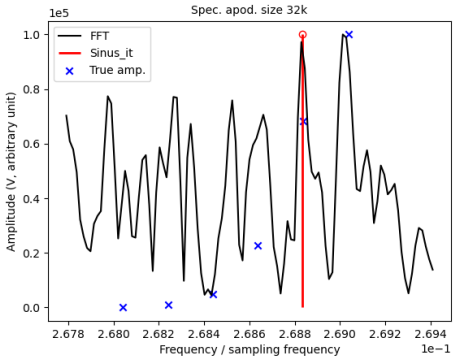

Figure 9.10: Coarse structure with Noise level 10^7

9.5 Fine isotopic structure

In fine isotopic structure, we are interested in determining the sines that are very close to each other in frequency for the first, second and third isotopes. The different sines will show the different in masses corresponding to the difference of energy that bond neutrons to different atoms. It is important to note that in the coarse isotopic structure, the first peak with the highest intensity is called the mono-isotopic peak. We refer to the second peak as first isotope, third peak as second isotope and fourth peak as third isotope.

9.5.1 First isotope fine structure

Because Substance P is made of 5 atoms ($C_{63}H_{98}N_{18}O_{13}S$), the extra neutron of the first isotope can be attached to any of the 5 different atoms, with however a different bond energy,

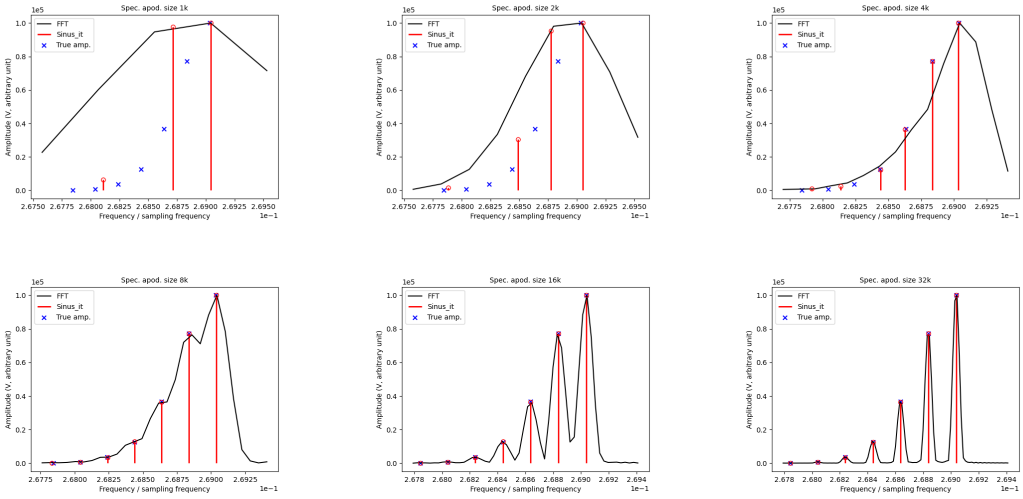

Figure 9.11: True simulated coarse isotopic distribution with 1k-32k points

leading to potentially 5 different sines that are very close to each other.

For the first coarse isotopic peak, the goal is to first detect 3 fine isotopic peaks. Then extend the objective to 5 peaks. In order to detect the 3 peaks, we first try to determine the sample size required to distinguish these 3 small peaks. The runs were repeated for 512K, 1M and 2M points in full sample mode. As it is seen from Figure 9.12, 512k points are not enough to detect the 3 fine isotopic peaks. At least 1M points are needed to recover the sines. Genetic algorithm found all three peaks correctly with 1M and 2M points, whereas FFT only found the first peak. With 2M points, FFT is still having trouble determining the smaller peaks. As in the previous case, FFT requires more points in order to detect all three peaks. With 2M points, we obtained better precision compared to 1M points. Therefore, we use 2M points in our runs for determining the isotopic fine structures. The run time using 512 generations was about 2 days.

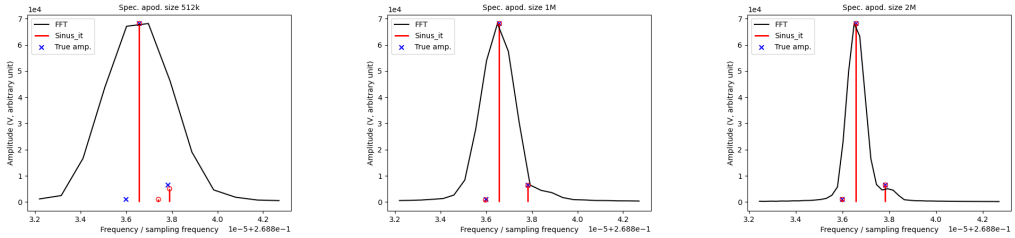

Figure 9.12: Fine structure for first isotope with 3 sines using 512k points (left), 1M points (middle), 2M points (right)

Next, we extend the number of sines to be found to 5, which is the number of fine isotopic peaks in the first isotope. The spectrum plot (Figure 9.13 left) shows that sinus-it can detect 4 of the 5 peaks accurately, whereas FFT can only detect the first peak with 2M points. For comparison, the number of points for FFT was increased to 16M and it is shown in Figure 9.13 (right) that FFT can detect only 3 peaks. The execution time using 5 sines was about 3 days over 512 generations.

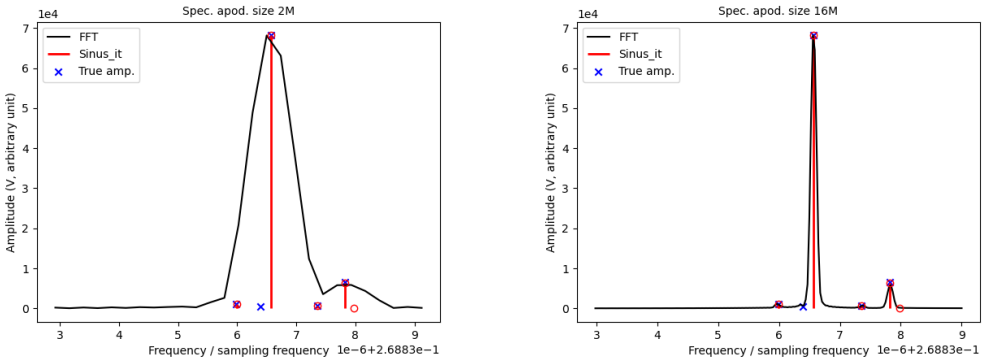

Figure 9.13: Fine structure for first isotope with 5 sines. FFT and sinus-it on 2M points (left), FFT on 16M, sinus-it on 2M points (right)

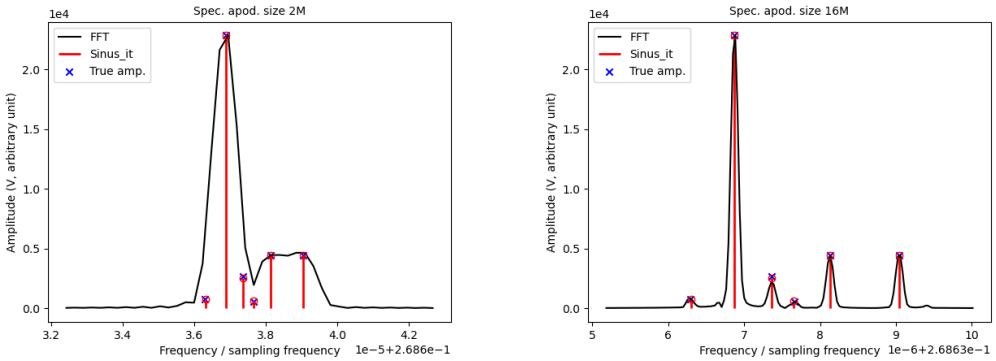

Figure 9.14: Fine structure for second isotope with 6 sines, FFT and sinus-it on 2M points (left), FFT on 16M and sinus-it on 2M points (right)

9.5.2 Second isotope fine structure

For the second isotope, the goal was to first determine 6 fine isotopic peaks. All 6 peaks were successfully found by sinus-it (Figure 9.14 left) using 2M points. FFT however found

only the first peak correctly and was close to determine the second peak. Increasing the number of points to 16M for FFT, we notice that FFT detected all the 6 peaks as shown in Figure 9.14 (right) but with 8 times more points compared to sinus-it. The run time was about 4 days over 512 generations.

The second isotope consists of 11 fine isotopic peaks. We generated a signal with all sines of second isotope and ran sinus-it. The result is shown in Figure 9.15. In Figure 9.15 (left), we plotted sinus-it and FFT results for 2M points and in Figure 9.15 (right), we plot the results for sinus-it using 2M and FFT using 16M points. We can see that with sinus-it, using 2M points, we obtain 7 of the 11 peaks, even 2 very small peaks, whereas FFT can only identify 1 peak and close to detecting the second peak. With 16M however, FFT can identify 4 peaks. Increasing the number of sines to 11 increased the run time to about 7 days over 512 generations.

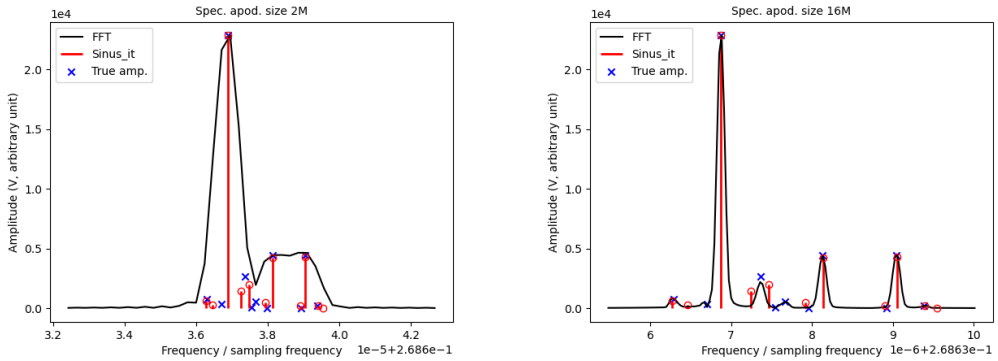

Figure 9.15: Fine structure for second isotope with 11 sines, FFT and sinus-it on 2M points (left), FFT on 16M and sinus-it on 2M points (right)

9.5.3 Third isotope fine structure

As in the case of second isotope, our goal for the third isotope is also to first detect the 6 fine peaks. The result is presented in Figure 9.16 (left). Using 2M points, we can detect all the 6 peaks using sinus-it. FFT however can only detect the first peak, and close to getting the second peak. Again, the number of points was increased to 16M for FFT and the result is shown in Figure 9.16 (right). The result shows that FFT can also detect all the 6 peaks but again with 8 times more points compared to sinus-it. The run time was about 4 days over 512 generations.

The third isotope consists of 18 fine isotopic peaks. In order to try to find these peaks, we generated a signal using the 18 sines. The result based on 2M points is shown in Figure 9.17 (left). We note that sinus-it was able to detect 6 of the fine peaks using 2M points whereas FFT again found the first peak correctly and was close to detect the second peak. We also notice that sinus-it found several peaks with correct peak position, however the precision in amplitudes were not correct. We increased the number of points to 16M for

FFT and we note that FFT can detect 5 peaks and 3 additional peaks with correct peak position but poor precision in amplitudes (Figure 9.17 (right)). With 18 sines, the run time jumped up to about 14 days over 512 generations.

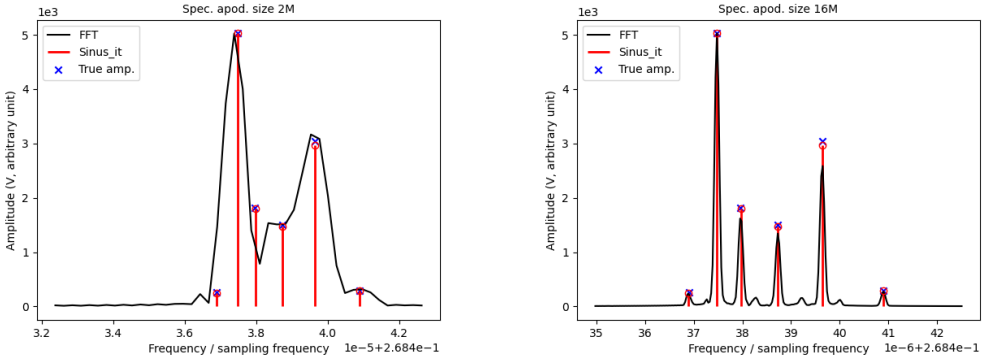

Figure 9.16: Fine structure for third isotope with 6 sines, FFT and sinus-it on 2M points (left), FFT on 16M and sinus-it on 2M points (right)

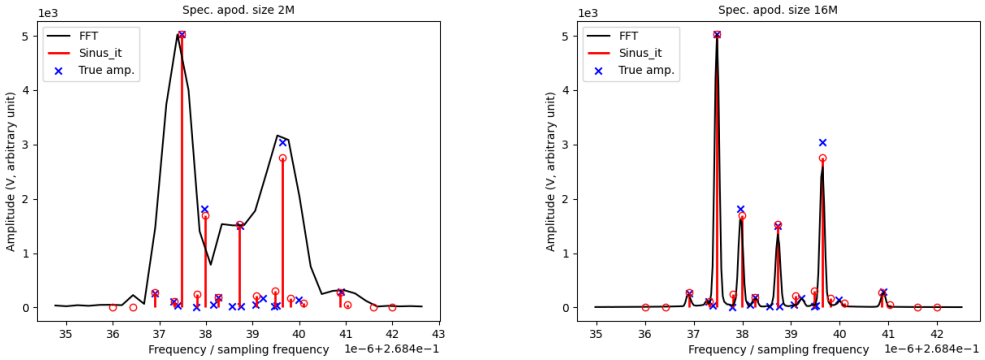

Figure 9.17: Fine structure for third isotope with 18 sines, FFT and sinus-it on 2M points (left), FFT on 16M and sinus-it on 2M points (right)

9.6 Reproducibility

Reproducibility refers to being able to reproduce the same results that was obtained by someone else by repeating the same procedures. Reproducibility is a major problem in computational science. More often, published results are not reproducible because of missing

details, datasets or information on the implementations of the algorithms. Therefore, there is an increased concern about reproducibility of experimental studies in computational science.

Krafczyk *et al.* [88] investigated reproducibility in over 300 papers and attempted to reproduce the results presented in the papers. However, they were not fully able to reproduce the results in any of these papers. The authors suggested three principles for researchers to make their computational results reproducible. These principles are:

- transparency and full reporting on how the results are produced,
- aiming for re-executability when writing research code,
- making the code deterministic.

The methods described in computational studies should be linked to parts of the code that was written to implement the method. Describing the method theoretically is not sufficient for readers to understand the implementation of the method. Then it is important that the code can be executed again and again. The code should be clear and should contain sufficient information about the important parameters to redo the computational experiments. When an experiment is run again, it must produce exactly the same output as the previous run.

It is also important indicate the versions of the software used and software dependencies as different versions of software may produce different results.

In many applications, stochastic algorithms are used. In such algorithms the results can be different when the runs are repeated several times even the same inputs are used. For such algorithms, reproducibility is difficult. In most programs, a random number generator using a seed is used to create the random numbers used in the stochastic algorithm. Randomness between runs can be removed by setting a seed value as a parameter. If the seed is not set by the user, clock time is used making reproducibility really problematic if the seed value has not been saved in the output for later use. A robust algorithm should return exactly the same result even if it is a stochastic algorithm, which is possible if the same seed is used again to reproduce the result. The principle of setting the seed is discussed in ten rules of making research software more robust in [89].

In order to increase the reproducibility it is advisable to output a log file and results file. All the important parameters should be captured and saved in a log file. Even a small change in the code can result in a change in the output.

The advantage of using the EASEA platform is that an `.ez` sourcefile can be run on any computer running a Linux operating system. If the source code is made available, anyone will be able to recompile it using easena on their own computers and reproduce the results using the seed values saved in a log file that is automatically produced as an output along with the results.

As this PhD has been done using the EASEA platform, all its results are reproducible for anyone to check out the quality of the obtained results. The only difference that will appear is in run time, because different machines will have different computing power.

For the sake of reproducibility, the source code of `sinus-it.ez` is provided in Annex A of this thesis, as it is not too long. Therefore, anyone downloading this thesis in the future will be provided with an EASEA source code that can be recompiled and run to reproduce the results presented in Chapter 9.

Chapter 10

Sinus-it on experimental data

Since the proposed method performed well on simulated Substance P data, we now test it on an experimental data coming from the FT-ICR MS (magnetic field strength 9.4 Tesla). We use experimental Substance P and Glutathione data to test the performance of our algorithm with respect to FFT method.

10.1 Experimental substance P

For substance P, we obtained signal of size 16M with an acquisition rate 2.5 MHz. As in the simulated data, the interest is to first identify the coarse isotopic structure. We showed that after obtaining the coarse isotopic structure correctly, we can use the obtained frequency-phase relationship to obtain the fine isotopic structure.

10.1.1 Coarse isotopic structure

First, the coarse isotopic structure is resolved. As mentioned earlier, there are 7 coarse peaks of Substance P. In our algorithm, we set the number of sines to be found to 7 and run the algorithm on coarse mode with 32k contiguous points. The results are displayed in Figure 10.1. With 32k points, sinus-it identified 6 of the 7 peaks, whereas FFT determined only 2 peaks correctly and was close to determine 2 other peaks, but the accuracy is not good. By increasing the number of points to 16M for FFT only, we note that with 16M points FFT identified 5 peak positions correctly, but the precision in amplitude was only correct for the first peak as seen in Figure 10.1 (right). This figure shows the problem with long transients. As we can see the precision in amplitudes is not correct with the FFT method except for the first peak and this is because when long transient is used, small sines start disappearing due to damping of the signal. The noise becomes more important than the signal. Therefore, it is desirable to use smaller sample size in order to work on the part of the data where the signal to noise ratio is not very low.

The results on fine isotopic structure were not satisfying and thus are not presented in this thesis.

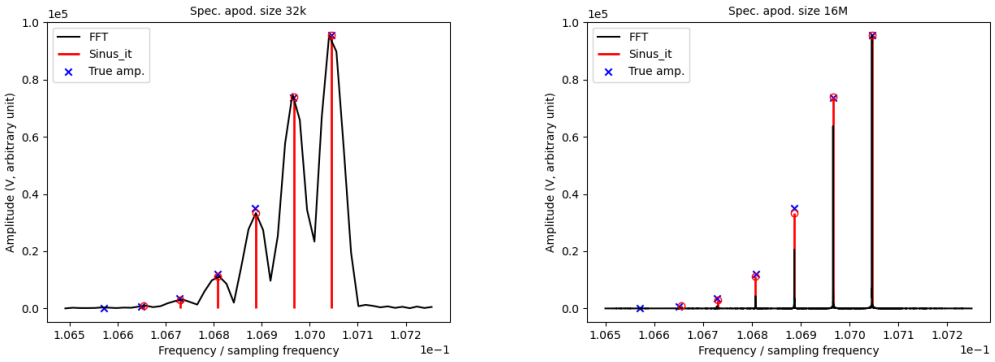

Figure 10.1: Experimental Substance P coarse isotopic distribution with 32k points (left), 16M points for FFT (right)

10.2 Experimental Glutathione

Next, we test our algorithm on an experimental Glutathione data. Glutathione is an antioxidant produced in cells (*cf.* `https://en.wikipedia.org/wiki/Glutathione`): of chemical formula $C_{10}H_{17}N_3O_6S$. Signal of size 16M was generated with an acquisition rate 1 MHz.

10.2.1 Coarse isotopic structure

First, we determine the coarse isotopic structure for Glutathione. We use 32k points and run the algorithm for 2k generations. The result is compared with the FFT method. In our algorithm we looked for 8 peaks. As we see from the Figure 10.2, FFT detects all four peaks, whereas sinus-it detects the first 3 peaks. Zoom on the left peaks is shown in Figure 10.2 (right). As we can see FFT detects the 4th peak even the precision is not good, whereas sinus-it couldn't detect this peak. We notice that sinus-it detects several small peaks around the large peaks. This indicates that we should look for larger number of sines and eliminate the ones that are not important. Since execution time increases with the increase in the number of sines, introduction of dynamic number of sine will be done in the near future.

After obtaining the coarse isotopic structure, we looked into the frequency-phase relationship and we found the linear relationship on a small interval. This indicates that we found the phase parameter correctly for the coarse isotopic structure. The relationship is shown in Figure **??** and the obtained linear equation is:

$$phase = -2748.4555 * frequency + 648.50103 \tag{10.1}$$

We can now use this relation to initialize the phase parameters when finding the fine isotopic peaks.

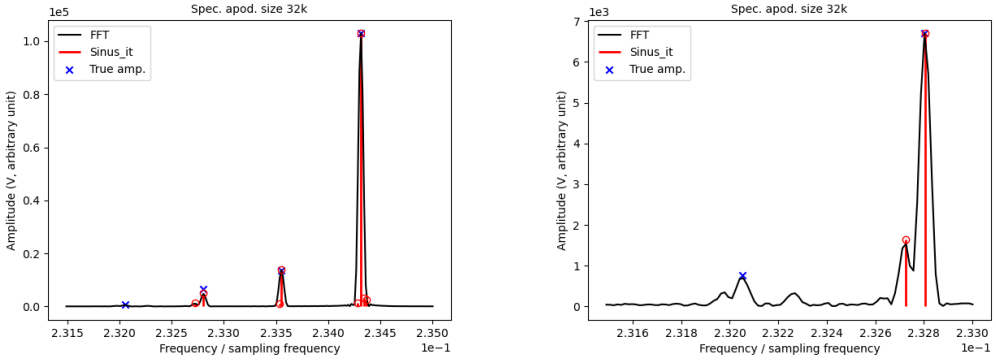

Figure 10.2: Experimental Glutathione coarse isotopic distribution with 32k points (left), zoom on left peaks (right)

10.2.2 Fine isotopic structure

In order to determine the fine isotopic structure for Glutathione, we look for 5 peaks in the first isotope. We run the algorithm on 1M points for 512 generations. The result is shown in Figure 10.3. As we can see from the figure, sinus-it can detect at least 3 peaks correctly, and close to getting the 4th peak whereas FFT can only detect 1 peak and requires more points to be able to detect the other peaks.

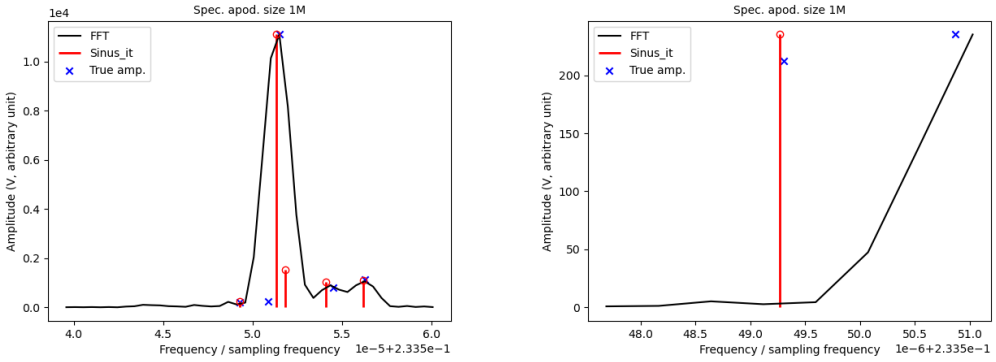

Figure 10.3: Experimental Glutathione fine isotopic distribution for first isotope with 1M points (left), zoom on left peaks (right)

10.2.3 Different starting times

For all the results discussed previously, we chose the samples uniformly from the beginning of the signal. We tested what happens if we choose the samples from different parts of

the signal. For this reason, we picked 32k points starting from 0k, 128k, 256k, 512k and 1024k. We plotted the starting times vs frequencies as shown in Figure 10.4. We noticed a negative linear relationship between the starting times and frequencies, with $R^2 = 0.9831$ and equation:

$$y = 1.006 \times 10^{-10}x + 0.2343 \tag{10.2}$$

.

This relationship describes well the non-ideal behavior of the ions in the FTICR cell.

Next, we plotted the frequency vs phase in Figure 10.5 and we can see the quadratic relationship between the frequency and phase as expected. The quadratic equation is:

$$y = 4.52891 - 0.17223x + 0.0556x^2 \tag{10.3}$$

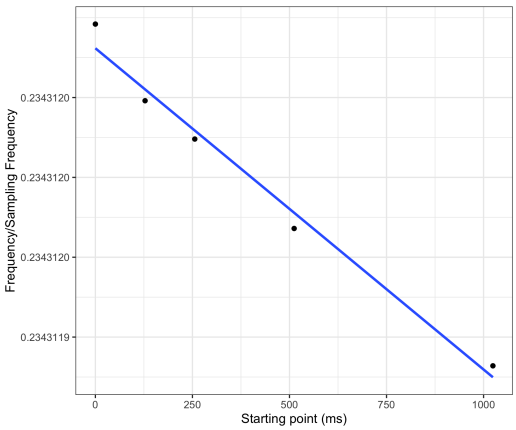

Figure 10.4: Frequency at different starting times

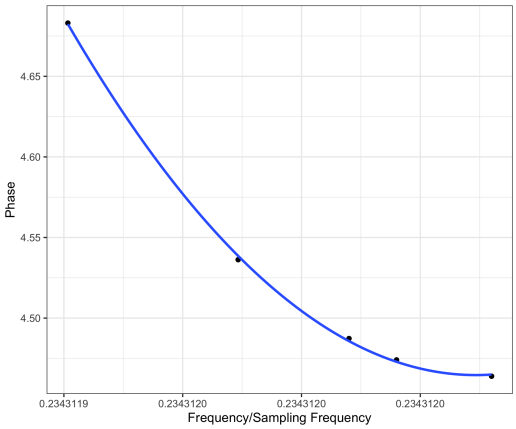

Figure 10.5: Dependence of Phase on Frequency

Chapter 11

Sinus-it speed optimization

In the previous section, we discussed the results obtained with full sample and compared them with results obtained by FFT. Now, our objective is to try different methods in order to obtain the results faster. For speed optimization, we try GRS by replacement (described in section 4.1), GRS by replacement in island model, GRS by extension, full sample in single precision, full sample using BRAD (introduced in section 11.4) in single precision and quantum based approach in double precision.

11.1 GRS by replacement

We explore the different parameters we can use in GRS by replacement. It is important to set the the number of generations elapsed before replacing the GRS points properly so that all the experimental points are used. We try sample sizes ranging from 2k to 16k (powers of 2) in a 32k interval. The number of replaced points is determined by dividing the sample size by a value k where k is an integer ranging from 1 to 16 in powers of 2.

We want each point to be used 8 times. Therefore, we determine the number of generations elapsed before replacing the GRS points `nNB_GRS_GENS` as shown below:

$$\texttt{nNB\_GRS\_GENS} = \frac{\texttt{NB\_GEN}}{(\texttt{nNUS\_GRS} - 1) * \texttt{nNUS\_GRS\_SAMP} * 8} \tag{11.1}$$

where `NB_GEN` = 2048.

We run the algorithm in coarse mode, with 6 sines and as usual conduct 30 runs for each case in order to obtain statistically significant results. Our results show that with GRS by replacement we can easily obtain 5 sines. Obtaining the 6th sine however is not always easy and the results are not stable. Due to randomness, most of the times the 6th sine is detected in 30 runs, but not always. In Table 11.1, even though sample size 2k looks better than 4k, repeating the runs does not prove this. Therefore, this needs to be investigated further. However, we can conclude that with only 2k points, we can always obtain 5 sines whereas with FFT, as discussed in the previous section, at least 128k uniformly sampled points are required to detect 5 sines.

nNUS_GRS_SAMP \Sample size	16k	8k	4k	2k
1	5	4	9	3
2	2	2	9	3
4	5	3	4	6
8	6	5	15	5
16	4	7	10	7

Table 11.1: Number of times 6th sine is not found in 30 runs with GRS by replacement

The execution time for sample sizes 16k, 8k, 4k, 2k is shown in Table 11.2. We note that the execution time decreases as sample size increases. Time does not decrease linearly due to algorithmic overhead (evaluation time is not 100% of the algorithm).

Sample size	Mean execution time (seconds)	SD
16k	11088	65
8k	5908	45
4k	3378	21
2k	2086	18

Table 11.2: Mean execution time for GRS by replacement

11.1.1 Using EASEA's island model for GRS by replacement

In Table 11.1, we showed that obtaining 6th peak using GRS by replacement is not always possible. For example, focusing on the case where sample size is 4k and nNUS_GRS_SAMP = 1, we note that 9 out of 30 runs could not find the 6th peak.

Therefore, we use the island model with the same parameters as a comparison since island models returns less variability thanks to good communication between islands. The island model runs were done on the same machine using 4 islands and the runs were repeated 30 times. The results in all 30 runs were similar, all 6 sines were found whereas with isolated runs, the results were not stable as discussed above. This shows that we can obtain better results with island model with much less variability.

For comparison, we also used 4 different machines each machine being one island. The results were again great. The difference between using one machine with islands and using 4 different machines as islands is in the execution time. When the runs were made on the same machine, the execution time was 4 times larger compared to a single run on one machine. Whereas using 4 machines each being one island returns the result in about the same time as it would for a single run on one machine.

However, it is important to note that in island model the best result is found much faster compared to a single run on one machine. Therefore, we ran the algorithm in island model for 512k generations compared to 2k generations that was used for the single runs. The mean run time over 512k generations using 4 islands on one machine was 2970.112 seconds with a standard deviation of 3. If 4 different machines were used, then the run time over

512k generations would have been 4 times faster compared to one machine with 4 island runs.

When using 4 islands in the same machine with the same number of generations, the execution time becomes about 4 times larger than one run on one machine without island model because 4 islands mean that 4 times more evaluations need to be done. However, when 4 different machines were used for islands, the execution time is similar to the isolated run's execution time. However, when using the island model, best fitness is achieved faster compared to isolated runs and with less variability.

Tests were made with 512 generations only instead of 512 generations, that showed more regular results than one machine wish 2K generations, showing the ability of the island model to perform asynchronous parallelization such as described in our paper [81]. The result of the island model is presented in Figure 11.1.

In Figure 11.2, we compare the best fitness values for the island model and isolated runs. As we can see, with island model, the best fitness values are lower and less variable compared to the isolated runs case.

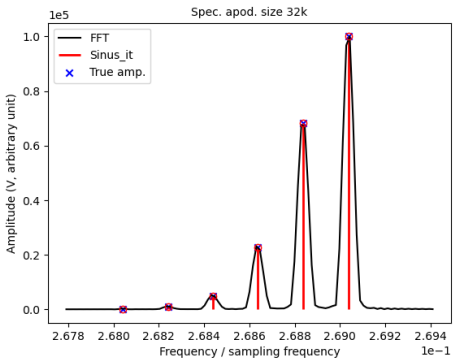

Figure 11.1: Result of GRS by replacement by using the island model, sinus-it 4k points, FFT 32k points

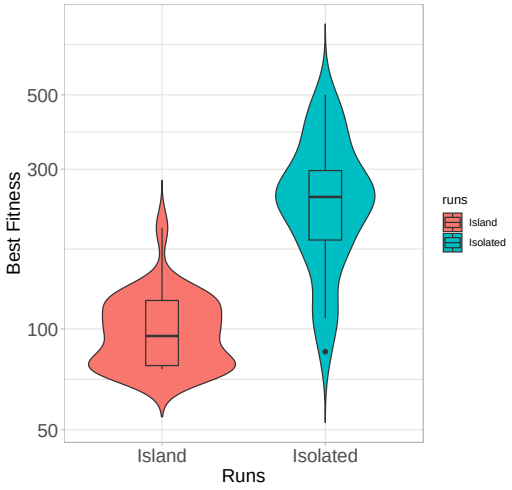

Figure 11.2: Best fitness of island model runs vs isolated runs

11.2 GRS by extension

We next performed GRS by extension using 4k points selected from a 32k interval. Every 128 generations, we increase the number of points by 1k and run it for 2k generations. By the end of 2k generations, we use 19k points. Our results show that we obtain 5 sines with this method. However, the variability is large in the results that sometimes even 5th sine can't be found. The relative errors are presented in the violin plots (Figures 11.3) based on 30 runs. Focusing on the frequency errors, we can see that the error for the 6th sine is very

high, for the 5th sine also there is a large variability with several outliers. There is also one outlier on the upper side for the 3rd and 4th sines.

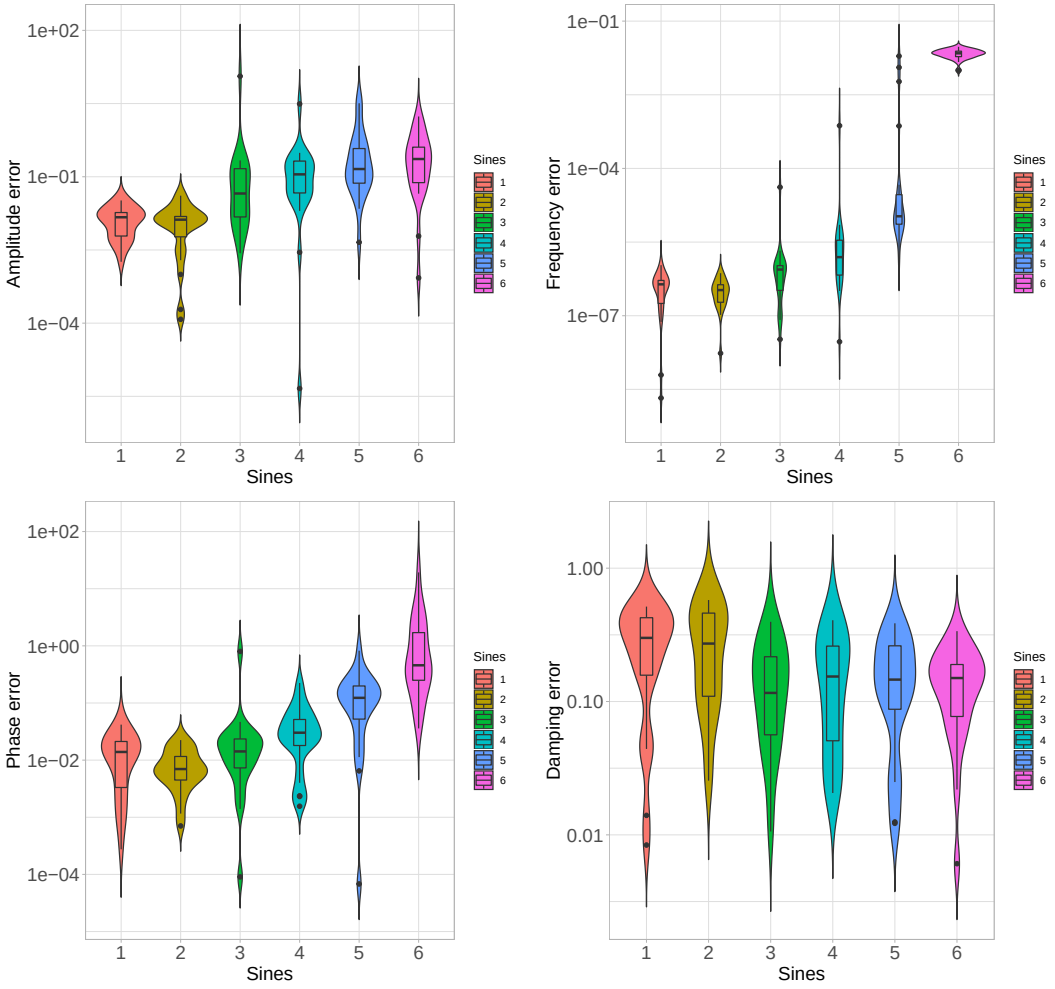

Figure 11.3: Relative errors for GRS by extension ($\log_{10} scale$)

The mean execution time for GRS by extension is about 8196 (sd=67) seconds whereas with full sample, we obtained the result in 21289 (sd=177) seconds, which is about 2-3 times slower than GRS by extension. However, the result on GRS by extension is not promising and it needs to be investigated more.

11.3 Single and double precision

Some tests were made to assess the importance of precision on the quality of the results. Double precision computing takes more time than single precision computing, asking whether the benefits of double precision are worth the longer runtime.

For comparison, we conducted 30 runs using single precision to determine the 6 main peaks in coarse mode using 32k points as was done with double precision in the previous chapter. In the violin plots shown in Figure 11.4, we compare the relative errors for amplitudes and frequencies using double and single precision. On the left we plot the results in double precision and on the right plots we show the results of single precision. We note that there is no significant difference in the relative errors for amplitudes. For frequencies, we note a large variability in the frequency error of the first sine. However, overall the precision is good for both single and double precision cases.

The mean run time with floats was 10038.8 seconds, whereas with doubles, the mean run time was 21283.03 seconds. The standard deviation however was twice higher in floats ($sd = 294.2845$) compared to doubles ($sd = 176.3302$).

Next we try single precision for the fine mode. We look for the first isotope with 3 sines. The result in double and single precision is shown in Figure 11.5. In double precision, all 3 peaks are found, whereas in single precision, only 2 peaks are found. The precision is not enough to determine the smallest peak.

In order to obtain better precision with single precision, we defined a new range reduction technique using binary radians described in the next section.

11.4 Introducing BRAD

When working with elementary functions, some applications require the highest possible precision. It is usual to compute the sine or cosine of large values in areas such as signal processing, where sines are calculated over prolonged periods of time. However, there is a problem with trigonometric computations of large angles using radians, as taking modulo 2π cannot be done in applied mathematics, because π has an infinite number of decimals.

If only a couple of "cycles" must be removed, then, the precision error is not important, and $\sin(0.1)$ and $\sin(0.1 + 2 * \pi)$ are nearly identical.

But when millions of cycles must be removed to compute a sine, the cumulated precision error becomes really problematic. As an example, please find in Table 11.3 a small C program that prints the results of $\sin(0.1)$ and $\sin(0.1 + 10^6 * 2\pi)$ using floats, doubles and long doubles.

The result of this very simple C program is troubling:

```
% cc sinusPC.c ; ./a.out
In float, double and long double, 0.1 is:
0.1L=0.1000000000000000000013552527156068805425093160010874271392822265625000
0.1D=0.1000000000000000055511151231257827021181583404541015625000000000000000
0.1f=0.1000000014901161193847656250000000000000000000000000000000000000000000

In floats, doubles and long doubles, sin(0.1) and sin(0.1+2*PI*1e6) don't show the same values:
sinl(0.1L)           LD=0.0998334166468281523107268504713118772997404448688030242919921875000
sinl(0.1L+2*PI*1e6L) LD=0.0998334164030319239747186463307393466948269633576273918151855468750
```

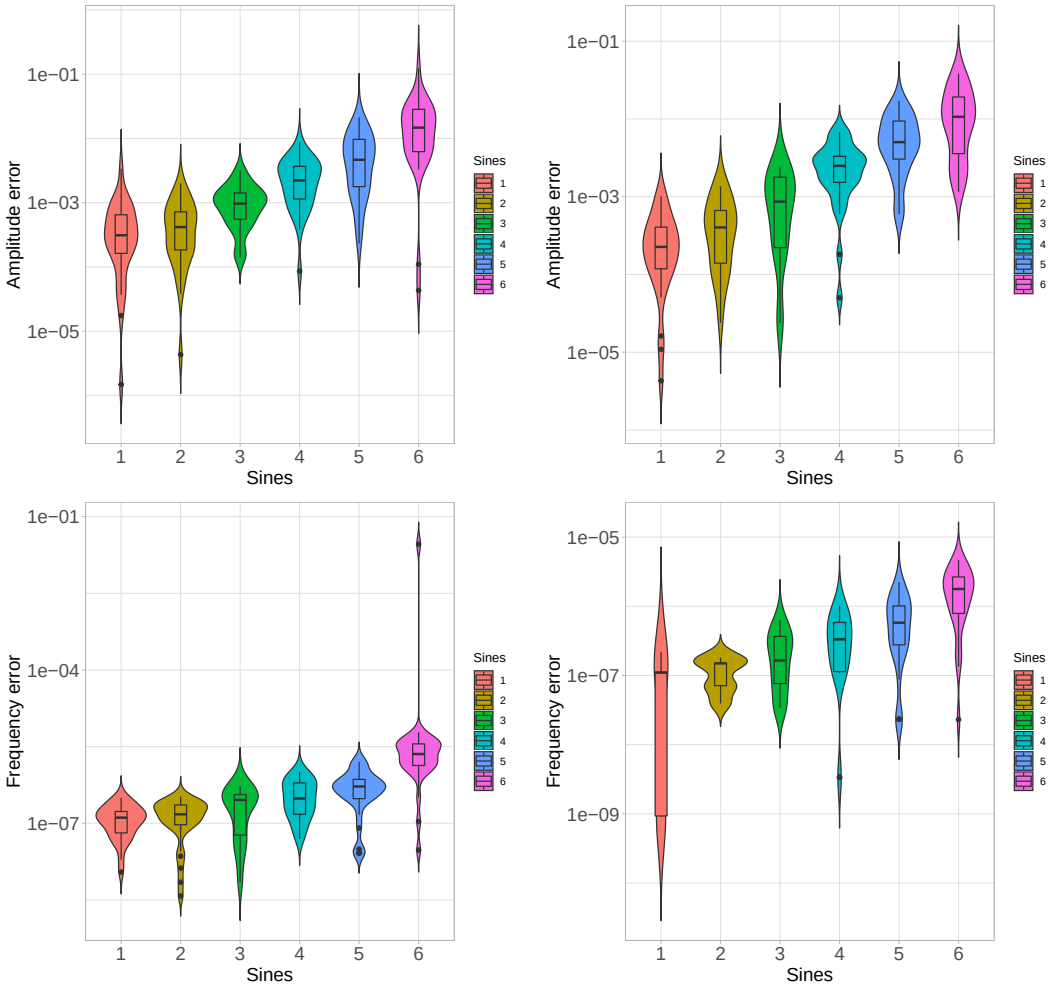

Figure 11.4: Relative errors in double precision (left) and single precision (right) using 32k points ($\log_{10}$ *scale*)

```
sin(0.1D)              D=0.0998334166468281547501817385636968538165092468261718750000000000000
sin(0.1D+2*PI*1e6D)    D=0.0998334158320078324244661871489370241761207580566406250000000000000
sinf(0.1f)             F=0.0998334214091300964355468750000000000000000000000000000000000000000
sinf(0.1f+2*PI*1e6f)   F=0.1916278004646301269531250000000000000000000000000000000000000000000
```

What is troubling is that when in mathematics, the value of $\sin(0.1)$ should be mathematically identical to the value of $\sin(0.1 + 2\pi * 10^6)$, we clearly see above that this is not the case. Let us analyse the above lines:

- If we take the `long double` value as a reference, we see the limit of `long double`

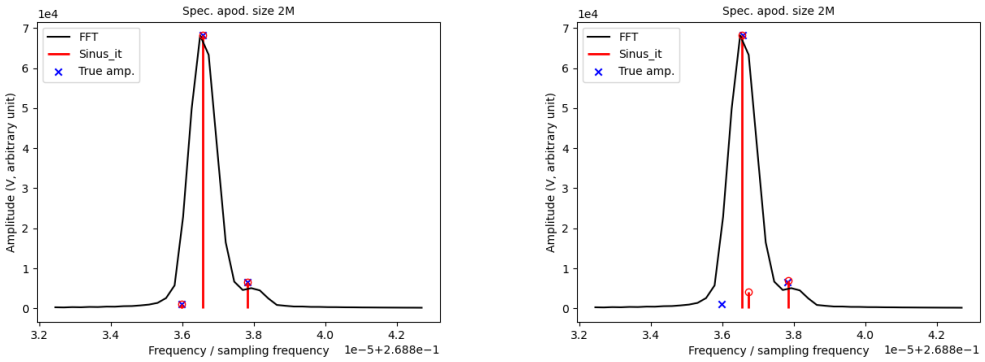

Figure 11.5: Fine structure of first isotope using 3 sines and 2M points (left: double precision, right: single precision)

```
#include<math.h>
#include<stdio.h>

// M_PI is in math.h

int main(){
  printf("In float, double and long double, 0.1 is:\n");
  printf("      0.1L=%.70Lf\n      0.1D=%.70f\n      0.1f=%.70f\n\n",0.1l,0.1,0.1f);

  printf("In floats, doubles and long doubles, sin(0.1) and sin(0.1+2*PI*1e6) do not show
       the same values:\n");
  printf("sinl(0.1L)            LD=%.70Lf\n",sinl(0.1L));
  printf("sinl(0.1L+2*PI*1e6L) LD=%.70Lf\n",sinl(0.1L+2.0L*M_PI*1e6L));
  printf("sin(0.1D)             D=%.70f\n",sin(0.1));
  printf("sin(0.1D+2*PI*1e6D)    D=%.70f\n",sin(0.1+2.0*M_PI*1e6));
  printf("sinf(0.1f)            F=%.70f\n",sinf(0.1f));
  printf("sinf(0.1f+2*PI*1e6f)  F=%.70f\n",sinf(0.1f+2.0*M_PI*1e6f));

  return 0;
}
```

Table 11.3: A small C program to show the problem with computing sines of large values

precision in that after 64 decimals, what follows is a series of 0s.

- When we look at the value of $\sin(0.1 + 2\pi * 10^6)$ in `long double`, we see that it differs from the value of $\sin(0.1)$ at the 9th decimal. This can be problematic knowing that for fine isotopic determination, for 3 sines, the relative error in frequency can be 10^{-10} and a 16 million points acquisition will represent many million cycles that will be impossible to unwind perfectly using 2π on the circumference of the circle.

- When we look at the value of $\sin(0.1)$ when the computation is done in `double`, the precision is lower than in `long double`. This can be seen on the third line because the 17th decimal is wrong compared to $\sin(0.1)$ computed in `long double`. But when

$\sin(0.1 + 2\pi * 10^6)$ is computed in `double`, it is not the 9th decimal that is wrong (as for `long double` but the 8th decimal.

- Then, the 7th decimal of $\sin(0.1)$ in float is wrong.
- But worst of all, the very first decimal of $\sin(0.1 + 2\pi * 10^6)$ in float is wrong, because the million cycles that needs to be unwinded before sin can be computed are all wrong, because each cycle is 2π and π has an infinite number of decimals, and its value in `float` is therefore very imprecise, and the imprecisions add up 1 million times.

11.4.1 Related work on range reduction

In the past, software writers did not try to accurately obtain reduced arguments and therefore, when computing trigonometric functions of large arguments, the computing systems either returned an error, 0.0 or something nonsense. Thus, many people thought that it is not possible to compute trigonometric functions of large arguments. ATT System V Release 4 used to require an error message to show up when the arguments is very large and the result to show as 0.0 [90]. Several argument reduction methods have been implemented. One of the first implemented methods is Payne-Hanek algorithm. In 1982, Mary H. Payne and Robert N. Hanek implemented a table based algorithm to precisely perform argument reduction for trigonometric functions. Assume that y is the reduced argument from x such that

$$y = x - kC \tag{11.2}$$

where k is an integer with $k = \lfloor x/C \rfloor$. Let C be $\pi/4$ and $y \in [-\pi/8, \pi/8]$. Then $k = \lfloor x \cdot 4/\pi \rfloor$.

In order to do trigonometric computation of x, the following steps are performed:

- determine y and k,
- use a given table to deduce the value of the trigonometric function

Rewriting equation 11.2 for $C = \pi/4$, we get:

$$y = \frac{\pi}{4}\Big(x\frac{4}{\pi} - k\Big) \tag{11.3}$$

When x is a large number or very close to $k\pi/4$, then there will be great loss of significant digits when computing the subtraction $x - k\pi/4$. In order to do the computation precisely, enough bits of $4/\pi$ are needed to guarantee full 53 significant bit of mantissa in double precision.

Even though this algorithm works very well with large values of x, it is expensive in terms of operations. It is commonly used in some libraries, especially in the *fdlibm* library.

Ng *et al.* in [90] modified the idea of Payne and Hanek and implemented this algorithm in the *SUN fdlibm* library. The authors showed that it is not necessary to represent the constant k in full precision. By avoiding unnecessary computation, only 100s of bits can be used rather than 1000s of bits. However, this method is still slow and is not easy to implement.

Another proposed method is Cody and Waite's [91] method which is very efficient for small arguments. This method is used in the *binary64 CRLIBMlibrary* described in [92]. This library uses Cody and Waite's method for small arguments and Payne-Hanek's method for large arguments.

Our proposed method is however very easy to implement. In the next section, we introduce our method for range reduction in trigonometric functions.

11.4.2 Proposed exact solution to the impossible computing of modulo 2π in applied mathematics

So this exposes here the problem of using periodic functions with very large arguments: *modulo* 2π *cannot be computed exactly in any computer, because* π *contains an infinite number of decimals.*

In this section, we introduce binary radians, BRADs, for range reduction which has no problem with sines of very large number of BRADs as it can be reduced using a 256 modulo.

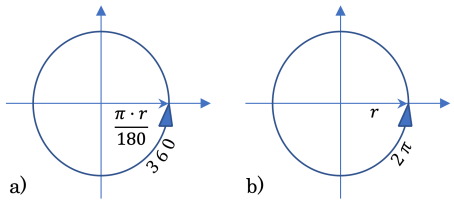

Figure 11.6: a) π on the perimeter b) π on the axis

Measuring an angle has been a long standing problem in the history of mathematics, that has been solved in different ways, depending on the context.

In the 3rd millennium BC, Sumerians were counting using a sexagesimal system that is still in use nowadays for measuring angles, geographic coordinates and time [93].

Among the many ways to do this, two are more common than others:

1. The first one consists in relating the measure of the angle to the radius, as suggested by Al Kashi in 1400, later on developed by Roger Cotes in 1714 [94] finally named Radian by James Thomson (brother of Lord Kelvin) and Thomas Muir in 1874.

2. The second way consists of laying π (the problematic number) on the x axis. By doing this, it is possible to measure an angle using an exact number:

   - 360, for the Babylonians in around 1000 BCE, elaborated on the 3000 BCE Sumerian spacetime notion), bearing the name of "degrees" or even
   - 400 as suggested by the scientists of French revolution, who installed the metric system, suggesting that a right angle would be measured as an angle of 100 grad.

However, the "naturalness" of measuring an angle in Radian described by Cotes is only valid from the point of view of a *mathematician* who can easily describe PI with the greek letter π, but clearly not to applied scientists for the reasons developed hereafter.

Programming languages adopted radians in computing the angles because mathematicians prefer it, but it creates the unsolvable problem of argument reduction (described above): because there are 2π radians in a circle and for any radian $x > 2\pi$, it is necessary to reduce the argument x to value y in the interval $[-\pi, \pi]$ using the equation below:

$$y = x - 2k\pi \tag{11.4}$$

where k is an integer which is the nearest to $x/(2\pi)$.

This is problematic in computer science because π is an irrational and transcendental number, meaning it has an infinite number of decimals when in computers, variables only hold a finite number of decimals. So in computers, it is not $2k\pi$ that is subtracted from x but $2k(\approx\pi)$.

When x is not too large, the error due to the limited number of decimals that are used to approximate π is not too important but, when x is a large number representing millions of cycles, the value of k is also in millions and $2k(\approx\pi)$ becomes very wrong explaining that in float (lowest precision), $\sin(0.1+2\pi*10^6) = 0.19162780046463012695312 5$ is totally different from $\sin(0.1) = 0.09983342140913009643554 6875$ to the first decimal.

If degrees were used, equation 11.4 becomes:

$$y = x - 360k \tag{11.5}$$

and this partly solves the problem because 360 is an integer value.

However, the problem is only "partly" solved because in order to use degrees in a sine function, you need to convert degrees to radians while doing a modulo and the modulo requires a division by 360 and 360 is not a power of 2. So here again, the division by 360 will create precision errors that we want to remove.

So the solution is to introduce "BRADs", for Binary Radians, that postulate that the circumference of a circle is 256 binary radians (and 256 is a power of 2). In BRAD, the *radius* of a circle bears the irrational π value (the radius is $r\pi/180$) while the circumference is exactly 256 (*cf.* fig. 11.7).

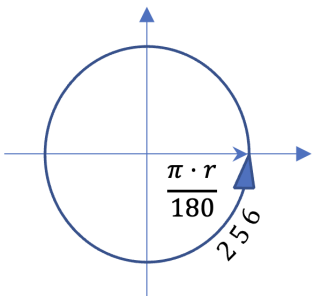

Figure 11.7: A circle is 256 BRADs

```
float brad2rad(float fBrad){
  return (fBrad-(int)(fBrad/256.0)*256.0)*2*M_PI/256.0;
}
```

Table 11.4: Brad2rad C function

```
float rad2brad(float fRad){
  return (fRad*256.0)/2*M_PI;
}
```

Table 11.5: Rad2brad C function

Implementation in C

So the idea is that the whole program is written using BRADs, but when it comes to computing a sine, the BRAD angle is converted into radians using the function in table 11.4.

It is important to note that in a binary representation, dividing by 256 means shifting the binary digits of the mantissa 8 times to the right (`>>8` operation in C), and multiplying by 256 means shifting the mantissa 8 times to the left (`<<8` operation in C).

So because 256 is a power of 2, dividing and multiplying by 256 is simply a bit shift (not a costly division) and is *exact*. This is the real reason for using 256 rather than 360 or 400 for the circumference of a BRAD circle.

The converse function (`rad2brad`) can be provided even though it would never be used in a program that is designed for using BRAD angles. Here again, the multiplication by 256.0 can be implemented as an 8 times bitshift of the mantissa to the left (`<<8` operation in C).

Testing BRAD in single precision

We tested this method in single precision on our problem where large arguments in trigonometric functions are computed, *i.e.* in fine isotopic mode that requires 2M points.

Unfortunately, we did not find a difference in quality by testing sinus-it with or without BRAD even though it should make a difference (the 3 sames peaks were found similarly, *cf.* fig. 11.8. This should therefore be further investigated.

However, we noticed that BRAD in float is faster than float only, with an execution time of 56648 seconds for the BRAD version *vs* 10039 seconds for the standard float version. This probably comes from the removal of divisions thanks to the fact that one turn is 256 BRADs.

But then, the real way to improve precision is to get rid of dyadic values by using an integer representation of floating point values. This needs to be done to improve the quality of the results.

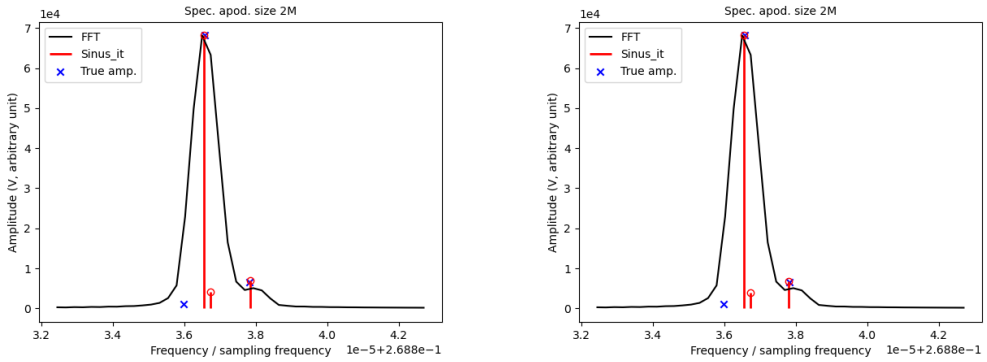

Figure 11.8: Fine structure of first isotope using 3 sines and 32k points (left: single precision, right: BRAD in single precision)

11.5 Comparing execution times

In this section, we summarize the run times for obtaining 6 sines in coarse mode using different cases mentioned in the previous subsections (Table 11.6). We note that with full sample in double precision, mean run time is the highest. In single precision, the run time is twice faster than in double precision, however, the variability of run time is much higher in single precision. Next we note that run time of GRS by extension is slightly less than single precision and variability in execution time is also much smaller. For GRS by replacement, we choose the smallest sample size (2k) for comparison which has the smallest mean run time.

Sampling methods	Precision	Generations	Mean execution time (s)	SD
Full sample (32k)	Double	2k	21289.052	176
Full sample (32k)	Single	2k	10038.795	294
GRS extension (4k)	Double	2k	8196.087	66
GRS replacement (2k)	Double	2k	2085.661	18

Table 11.6: Comparison of execution time for different sampling methods

Chapter 12

Tested improvements

12.1 Quantum based Evolutionary Strategy (QAES)

In this section, we present the mathematics of QPSO as described in [95]. In quantum physics, the state of a particle can be described using its wave-function $\Psi(x,t)$. In QPSO, each particle behaves as a quantum particle. The square of the wave function gives the probability of the particle's appearance in a certain position. A wave-function $\Psi(x,t)$ is used to describe the status of particles with momentum and energy. $\Psi(x,t)$ is governed by Schrodinger equation:

$$i\hbar\frac{\partial\Psi}{\partial t} = \hat{H}\Psi \tag{12.1}$$

where i is the complex number $i^2 = 1$, $\hbar$ is Planck's constant, and $\hat{H}$ is Hamiltonian operator:

$$\hat{H} = -\frac{\hbar^2}{2m}\nabla^2 + V(x) \tag{12.2}$$

where m is the mass of the particle, ∇ is the Laplace operator, V is the potential well. $-\frac{\hbar^2}{2m}\nabla^2\Psi$ is known as the kinetic energy and $V\Psi$ is the potential energy. Thus, the equation can be rewriten as:

$$\hat{H}\Psi = E\Psi \tag{12.3}$$

By solving the Schrodinger equation, one can obtain the probability density function Q and the distribution function F.

$$Q(X_{i,j}(t)) = \frac{1}{L_{i,j}(t)}e^{-2|p_{i,j}(t)-X_{i,j}(t+1)|/L_{i,j}(t)} \tag{12.4}$$

$$F(X_{i,j}(t+1)) = e^{-2|p_{i,j}(t)-X_{i,j}(t+1)|/L_{i,j}(t)}, (j = 1, 2, ..., D) \tag{12.5}$$

where $L_{i,j}(t)$ is the standard deviation of the distribution. For each particle, the search space is determined by $L_{i,j}(t)$. Using the Monte Carlo method, the position of each particle can be found by:

$$X_{i,j}(t+1) = p_{i,j}(t) \pm \frac{L_{i,j}(t)}{2}ln(1/u) \tag{12.6}$$

where u is a random number in $(0, 1)$. $p_{i,j}(t)$ is the local attractor.

$L_{i,j}(t)$ can be computed using *mbest* which is the global attractor and it can be found as the mean of the personal best position (*pbest*) of all particles. Thus,

$$L_{i,j}(t) = 2\beta \cdot |m_j(t) - X_{i,j}(t)| \tag{12.7}$$

Then, the position of each particle can be found by:

$$X_{i,j}(t+1) = p_{i,j}(t) \pm \beta \cdot |m_j(t) - X_{i,j}(t)| ln(1/u) \tag{12.8}$$

where β is contraction-expansion coefficient which can be tuned to control the convergence rate of the algorithm. β is usually set to 1.0 initially and then reduced linearly to 0.5. The pseudo-code of QPSO is given below 1 as presented in [95].

Algorithm 1: QPSO algorithm

1 Initialize the current positions and the pbest positions of all particles;
2 Do
3 Compute the mean best position
4 Select a suitable value for β;
5 For i=1 to population size M
6 evaluate the objective function value f($X_{i,j}$);
7 Update $p_{i,j}$ and $p_{g,j}$
8 if $f(X_{i,j}) < f(p_{i,j})$ then $p_{i,j} = X_{i,j}$;
9 $p_{g,j} = (p_{i,j})$, $1 \leq i \leq M$; for $j = 1$ to dimension D
10 $\phi_{i,j} = rand(0, 1)$, $u = rand(0, 1)$
11 $p_{i,j}(t) = \phi_{i,j} \cdot p_{i,j}(t) + (1 - \phi_{i,j}) \cdot p_{g,j}(t)$
12 if rand(0,1) > 0.5
13 $X_{i,j}(t+1) = p_{i,j}(t) + \beta \cdot abs(m_j(t) - X_{i,j}(t)) \cdot ln(1/u)$
14 else
15 $X_{i,j}(t+1) = p_{i,j}(t) - \beta \cdot abs(m_j(t) - X_{i,j}(t)) \cdot ln(1/u)$
16 endif
17 Endfor
18 Endfor
19 Implement local search strategy in algorithm1
20 Until termination criterion is met

Quantum inspired algorithms have been gaining lots of interest lately in optimization problems. A recent study presented Quantum-Inspired Algorithm with Evolution Strategy (QAES) based on the mixture of Quantum Diffusion Monte Carlo (DMC) method with an Evolutionary Strategy to solve global search optimization problems. This algorithm was successfully applied to black-box problems and promising results were obtained [96, 97]. This algorithm was modified for the problem presented in this thesis and QPSO was used for crossover. The modified algorithm is still under development. Therefore, we present a preliminary result obtained with it in the next section to compare with the sinus-it and FFT method.

12.1.1 Results using the Quantum-Inspired Algorithm with Evolution Strategy (QAES)

The QAES method [96, 97] developed by Anna Ouskova-Leonteva (PhD student in CSTB team) has been tried on this problem. Since this method still needs development and is in the list of future works, we show a preliminary result for comparison. QAES fits very well to our problem. The advantages of this method are that good results can be obtained in much shorter time and with smaller sample sizes compared to the sinus-it method.

The QAES method is highly dependent on its parameters. It generally works well with small populations and sample sizes, however depending on the complexity of the problem, larger interval and population size may be required.

To test the algorithm, we use 3 sines in fine isotopic mode and compare our result with sinus-it.

In Figure 12.1, we can see the violin plots based on 30 runs. The amplitude and frequency errors are in acceptable range. We can see that for the 3rd sine, the errors are larger but they are acceptable under the requirements. The results in amplitude and frequencies are similar to the ones obtained by sinus-it, however for the phase error, we can see that sinus-it had better results and this was because the linear dependence of phases and frequencies was taken into account. However, in QAES, no information was given about the dependence. The error in damping is also larger in QAES. With the development of the algorithm and/or tuning of parameters, it can be possible to obtain smaller errors in phase and damping constant.

What is interesting in the QAES algorithm is the sampling method and execution time. Non-uniform sampling was used in these runs with only 1024 points taken from an interval of size 8M, whereas in sinus-it, we used 2M consecutive points to obtain the results discussed in the previous section.

The execution time is shown in the violin plot in Figure 12.2. We can see that it took less than 80 seconds to get the results described above, whereas with sinus-it, it took about 2 days. This is a tremendous speed up.

The limitation of this method is currently the stability of results which mainly depends on the parameters of the algorithm. Determining the best parameters to have a stable result can be time consuming. However, the work will continue on this in the near future.

12.2 Using Genetic Programming to find the damping function

Because Genetic Programming needs to find both the equation and its parameters, and because we know what the signal is made of (a sum of sines) it was supposed that it would be easier for artificial evolution to do optimization (*i.e.* find the parameters of a sum of sines using ES or GA) rather than machine learning (finding the whole complete equation starting from raw data using GP).

But there was a problem: due to collisions between the studied compound and the never perfect void in the ICR chamber, the signal is not an exact periodic signal that can be modelled through a sum of sines: the amplitude of the signal decreases and is effectively a transient, and not stationary.

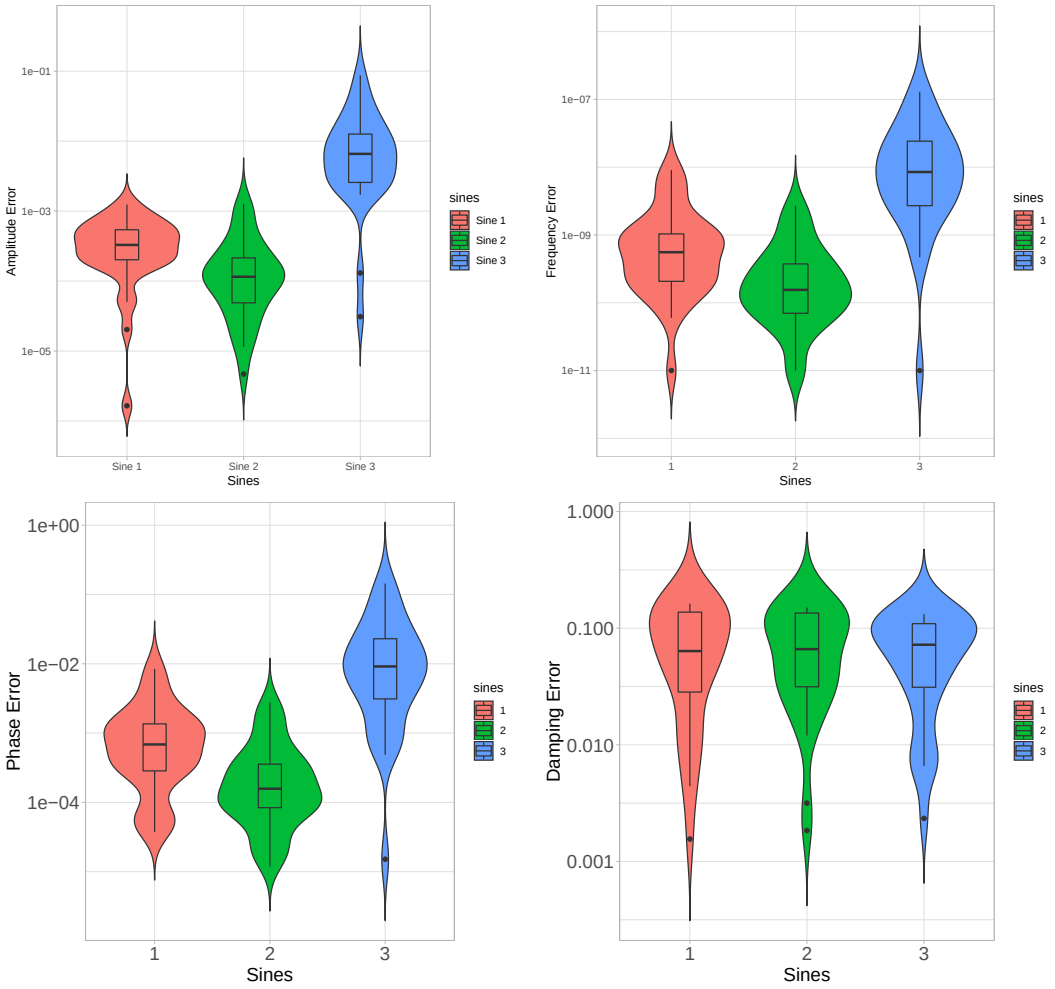

Figure 12.1: Relative errors for QAES ($\log_{10}$ *scale*)

Because the damping is supposed to be exponential, the first idea was to apply the inverse of the damping exponential to the signal in order to "straighten" it (*i.e.* turn the transient into a stationary signal) in what could be seen as a data preparation (or curation) step and then, try to find the parameters of a sum of n sines that would match the "straightened" signal.

However, when looking at a plot of the signal (*cf.* figure 12.3), the curve of the envelope is clearly not a decreasing exponential on the beginning of the signal.

So we decided to start out this work with finding out the equation of the curve of the envelope by using Genetic Programming.

As usual in data modelling, data curation is essential if good results are to be obtained.

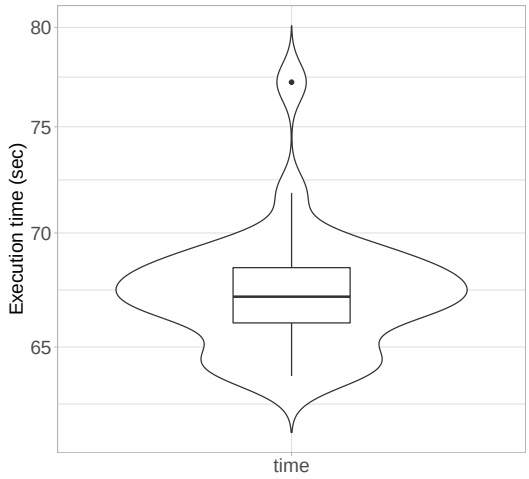

Figure 12.2: QAES Execution time (in seconds)

As said above, the real data produced by the FT-ICR mass spectrometer is damped as ions are lost by different physical phenomena over the acquisition time (collision with background gas and loss of coherence of ions) *cf.* Fig. 12.3.

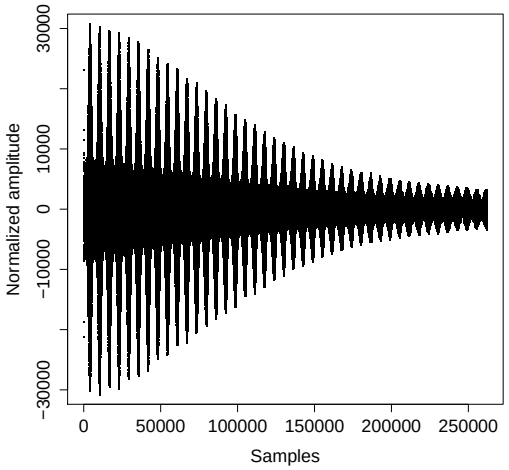

Figure 12.3: Raw data (256k points) produced by the ICR mass spectrometer. The envelope is clearly not a decreasing exponential at the beginning of the signal.

```
\Default run parameters :
  Number of generations : 500
  Population size : 50000
  Offspring size : 50000
  Mutation probability : 0.3
  Crossover probability : 1
  Evaluator goal : minimise
  Selection operator: Tournament 7
  Final reduce operator: Tournament 5
  Elitisim: Weak
  Elite: 1
  max init tree depth : 3
  min init tree depth : 2
  max tree depth : 5
\end
```

Table 12.1: The parameters of the standard Koza-GP algorithm

The most common form of a damped sinusoid is exponential damping. However, the problem with this real-world data is that in the beginning and end of the acquisition, signal magnitude damping does not follow the expected theoretical exponential law, meaning that the amplitude of the signal cannot be normalized using a simple exponential function. Therefore, it is necessary that the signal to be modelled be stationary (constant in magnitude) and periodic.

We propose to use Genetic Programming (GP) in order to find the damping equation to straighten the data. We use a damped signal of size 256k coming from the FT-ICR machine from which the peaks of the signal are extracted. Then, we used a standard regression template of the EASEA platform to check if Genetic Programming could find a non-linear function matching the curve.

The parameters of the standard Koza-GP algorithm (also parallelized on the GPGPU card) are described in Table 12.1.

The reason behind these parameters comes from Koza's experience. In his 2003 book [98], Koza routinely obtains human-competitive results on his 1000-PC super-computer by using huge populations and few generations, because of the tendency of Genetic Programming to converge fast.

Back in 2003, PCs used single-core CPUs and not GPGPUs, meaning that his 1000-PC super-computer was a 1000-CPU machine interconnected over a Beowulf network.

Nowadays, in 2021, GPGPU cards are available that host thousands of cores. This PhD uses the UFAZ PARSEC machine [81] that is made of 26 machines hosting two NVIDIA RTX 2080Ti 4352-core GPU cards, meaning that on top of its 8 CPU cores, each machine has access to 8704 GPU cores.

Then, the particular hardware architecture of GPU cards means that each core must run several threads for an optimal use of the GPGPU processor (due to register-based thread-switching specificities).

Because the problem is simple for Genetic Programming, only one of the 26 machines of the PARSEC supercomputer was used to find the equation of the envelope of the signal. Evaluating 50000 individuals over 8704 cores meant that in average, each core had to run around 5.74 threads, which is optimal for the loading of the multi-processors of the GPGPU

cards.

After many runs, it was found that on this particular problem, 0.3 was the best value for mutation probability, all children were created from crossovers, but most importantly, the most efficient selection operator for parents was a 7-Tournament with replacement and the most efficient population reduction scheme was a 3-Tournament without replacement operator [99].

Because the value to minimise was a mean square error of the difference between the envelope and the created function, the goal for the algorithm was to minimize this error, hence the `Evaluator goal` parameter setting to `minimise`.

Finally, weak elitism [100] is used (because the replacement is non-generational) with an elite number of 1 in order to both:

- not lose the best individual from generation to generation and
- minimise premature convergence.

The following equation was found by GP for modelling the envelope:

$$\begin{aligned}(((sin((3.24077e-05)*(x)))*((sin(0.209725))/(9.50403e-05)))\\ +((exp((3.24077e-05)*(x)))+((((0.982347)/(9.50403e-05))\\ -((0.127674)*(x)))+((exp(0.599937))/(9.50403e-05)))))\end{aligned} \tag{12.9}$$

Which simplfies in:

$$-0.127674x + 0.00999\sin(3.80932x) + e^{3.80932x} + 0.13460 \tag{12.10}$$

Starting from the raw data (*cf.* Figure 12.4 left), the extracted envelope was the blue circles in Figure 12.4 middle. Function 12.10 found by Genetic Programming was the black line in the middle figure.

Figure 12.4 right shows the signal multiplied by the inverse of function 12.10, that we will call "normalized" data.

One sees on Figure 12.4 (right) that the signal/noise ratio is going lower as the amplitude of the damped signal is magnified on the right of the curve to simulate a stationary periodic signal.

12.3 Using derivatives

In this section we present the idea of working with first and second derivatives. The idea was inspired when we faced difficulties in finely determining the sines composing the signal of an experimental data in our previous works as shown in Figure 12.5.

The black points on the figure are the values of the signal and red values are the values found by sinus-it using non-uniform sampling (NUS). This plot shows by green lines that the phase is good, it is nearly perfect. However, there is a lot variations as shown between the green lines and inside the blue circles. The error is more apparent on the peaks of the signal. So, the idea is to improve the variation rather than the signal. For improvement in the variation, we propose to work with derivatives. Between the green lines, the slopes of

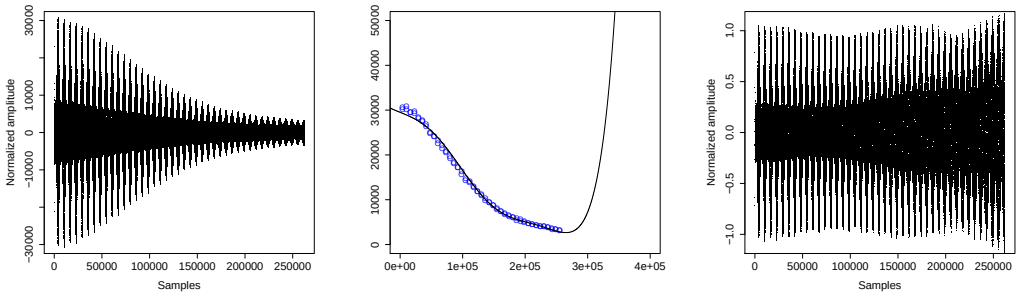

Figure 12.4: Raw data (256k points) produced by the FT-ICR mass spectrometer (left), evolved function to match the amplitude decrease (middle) and normalized signal (right)

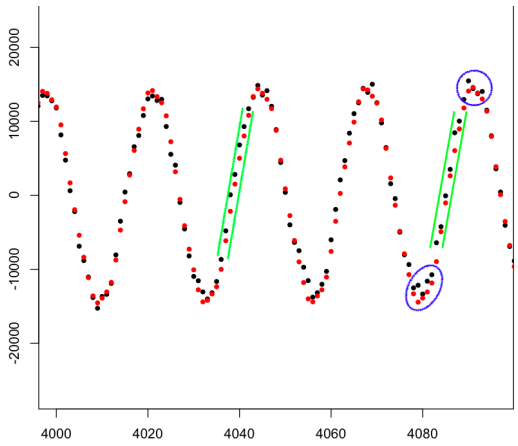

Figure 12.5: Experimental signal

the red and blue dots are nearly the same. On the contrary, focusing on the blue areas, the difference of the derivative between the black and the red curve is very different because it is in this area that the curve "turns" a lot, so if you want to get the details right, the derivative will show us exactly where to work. Then this can be extended to working with second derivatives. The idea would be to minimise the derivatives rather than the values. This method can be done using uniform or non-uniform sampling. In the case of uniform sampling, 2 points can be selected to obtain the first derivatives and 3 points can be selected to obtain the second derivatives (acceleration). Then the value of the derivatives is used in the evaluation function.

One of the preliminary result we obtained using 32k points non-uniformly selected from a 64k interval is shown in Figures 12.6 and 12.7. The plot on the left is done without derivatives and the plot on the right is obtained based on derivatives only. It is impressive that the result is nearly the same, because absolute values are not used to find the right red curve.

We note slight improvement using derivatives, however it needs to be investigated more. There still needs to be improvements which we believe is possible with more investigations in this method, by basing the fitness function both on absolute values and first and second derivatives.

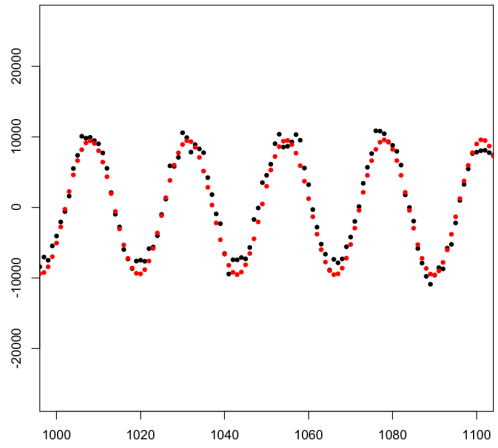

Figure 12.6: Without derivatives

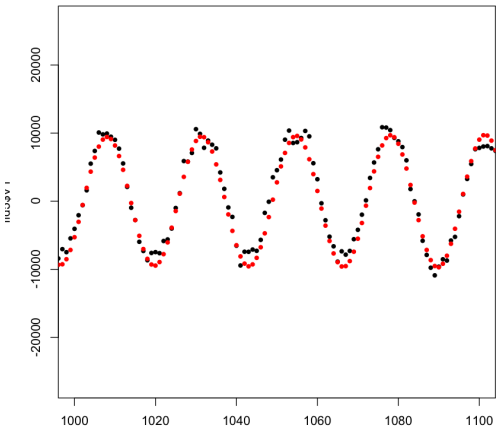

Figure 12.7: Using derivatives

Part III

Conclusion and perspectives

Chapter 13

Conclusion

In this chapter we reflect on our results and discuss the improvements that need to be done in the future.

In this PhD project, we proposed an evolutionary approach using genetic algorithm to overcome the limitations of the FFT method that is traditionally used in FT-ICR machines to obtain the frequency spectrum. Our results show that the approach presented in this PhD is superior to the famous FFT method.

In order to test harmonic analysis using artificial evolution, we used as a real world test data coming from FT-ICR machines, that do not yield the imaginary component that FFT needs to compute the phase, meaning that FT-ICR cannot provide the phase of the signal easily.

Then, genetic algorithms being of a stochastic nature, it was surmised that they would be better equipped to deal with noisy data than FFT that is a pure mathematical method, where noise is really problematic.

Our tests showed that harmonic analysis using artificial evolution could both:

1. provide the phase information quite accurately, which in turn helped the algorithm to find the finer isotopic structures around each peak and

2. deal with lower s/n ratio than FFT.

Our first goal was then to obtain great quality results on coarse isotopic structures to be able to determine the fine isotopic structures using smaller samples compared to FFT.

In order to test the performance of our algorithm, we first used a simulated Substance P data for which the parameters were already known. Since signals obtained in the real world are always noisy, we focused mainly on noisy and damped signals. The first objective was to determine coarse isotopic structures. Our results showed that by using much smaller sample size, we can easily determine the 6 main peaks of Substance P compared to the FFT method which requires much larger sample size to detect all the main peaks.

Once coarse peaks are determined, it is important to detect the smaller peaks around them. This was our next goal. The result of coarse isotopic structure allowed us to realize an important relationship between phases and frequencies. The phases are predicted to have a

quadratic dependence from the frequencies and this is a linear dependence on a short range with a very strong correlation. Knowing this dependence relationship was very helpful in determining the fine isotopic structures as we initialized the phases using the obtained linear equation.

To determine the fine isotopic structures, we zoomed into the first, second and third coarse peaks. We showed that for the first coarse peak, we can detect 4 of the 5 peaks of fine structure, for the second peak 6 out of the 11 peaks of fine structure and similarly, we detect 6 of the 18 peaks of fine structure for the third coarse peak. The peaks that were not detected had very small signal to noise ratios.

Our next goal was to optimize the speed of the algorithm. For this reason, we tried several methods that include: runs in single precision, single precision using BRAD, GRS by replacement, GRS by extension, island model, and QAES.

In the end, our algorithm "only" needs around 6 hours to determine the coarse structure (main peaks of the compound). Then, it took much more time to find fine isotopic peaks:

- 2-3 days for fine isotopic determination of the first isotope
- 7 days for fine isotopic determination of the second isotope
- 13 days for fine isotopic determination of the third isotope

The huge computing time for the third isotope comes from the fact that we were looking for 18 peaks but still, this huge computation time was smaller than for urQrd + FFT (current state of the art) which obtained a result of lower quality on the same data.

In this PhD project, we developed a sampling method called GRS which is related to the algorithm itself. The goal of this method was to use a random sampling that would allow the algorithm to use the entire sample using smaller portion of the data. We showed that using GRS by replacement mode, we can obtain 5 peaks using only 2k points in with an execution time 10 times faster compared to full samples using 32k points. Detecting 5 peaks using only 2k points in less than an hour is a great achievement. Our result is also superior to the result of FDM method mentioned in [41], where only 3 coarse peaks were found. With our algorithm we showed that we may even distinguish the 7th peak if we generate the signal with all sines.

In GRS by replacement, there was a large variability in the 6th peak that it was not found sometimes. To reduce this variability, we used island model in one machine. We showed that with the island model we obtained all the peaks with better reliability.

Our results were mainly based on double precision. We used single precision as a comparison and noticed that for larger samples, the precision decreases compared to the double precision. To improve the precision, we invented binary radians (BRADs). We believed that using binary radians would return better precision since it does not involve division by π in range reduction. However, our results showed that BRAD in float performed similarly compared to single precision but about 1.5-1.6 times faster.

We also tried a quantum based method, QAES, on our problem, in collaboration with Anna Ouskova-Leonteva. We presented only preliminary results using this method. We find this method very interesting and believe that this fits very well to harmonic analysis of signals as it is based on Schrodinger's equation which is a wave function. We obtained

interesting results using this method. Using only 1k points taken non-uniformly from an 8M interval, we obtained similar result as in sinus-it that needed 2M points for the same signal. Also, while sinus-it method took about 2 days for computation, the QAES method obtained the result in less than 80 seconds.

The data that comes from FT-ICR machine is damped. Even though in theory sinusoids are said to be damped exponentially, unfortunately in reality damping is not exponential and we cannot rely on exponential damping in the first part of the transient which has he highest signal/noise ratio. In this PhD, we showed that we can use GP to obtain the damping function which can be used to straighten the signal before analysing it.

To summarize, the following main results and key contributions were made in this PhD project are:

- Achieved super-resolution close to the theoretical limit ($\times 4$) by obtaining good precision using only 8k points with sinus-it compared to 32k points with FFT
- Obtained all the sine parameters including phase and damping factor contrary to FFT
- Obtained absolute amplitude of sines contrary to FFT. Amplitude obtained by FFT depends on apodization function and damping factor
- Achieved the required resolution for high resolution (better than 0.1 ppm in frequencies)
- Obtained accurate isotopic ratios as super-resolution allow to work on the less damped transient part
- Worked on coarse and fine isotopic distribution contrary to other super-resolution algorithms
- Run on fine isotopic distribution are speed-up by using dependence relation of frequencies and phases
- Explored the random sampling method (GRS) to speed-up sinus-it
- Introduced a new range reduction method for periodic functions (BRAD)
- Developed a new method of determining damping functions using Genetic Programming
- Parallelized on GPGPU cards using an island model improve precision and robustness.

Finally, the study on simulated data allowed us to resolve the coarse isotopic structure of an experimental signal coming from the FT-ICR MS. We showed that we can obtain the 6 main peaks of the experimental Substance P using much smaller sample size compared to FFT. We also tested our algorithm on experimental Glutathione and determined the coarse isotopic structure and first fine isotopic structure. Our results showed that in coarse mode, our algorithm required larger number of sines. In fine mode, we showed that our algorithm performed better than FFT with only 1M points.

Chapter 14

Perspectives

A whole chapter dedicated to improvements has been written at the end of Part II. Indeed, even though the presented results are significantly better than Fourier Transform, they have been obtained with a very simple algorithm (whose parameters have been finely tuned using months of experimentation on a very powerful super computer, the PARSEC machine [81]).

But this PhD is the first that we know of to focus on the application of an evolutionary approach to harmonic analysis. Therefore many improvements could not be completed, but only briefly experimented, showing room for improvement.

GRS

The idea of using global random sampling (GRS) was derived during this PhD work. It has not been extensively studied in different problems. We have studied this method on our problem using the two different versions. For GRS by replacement, we have studied different cases of sample sizes and number of points to replace after certain number of generations. However, we did not obtain an obvious pattern in our results to suggest its performance for different sample sizes. Table 11.1 showed the results based on different cases studied and we noted that the results were better with smaller sample sizes compared to larger sample sizes. However, this did not sound promising and thus, the run with sample size 2k was repeated again and this time it did not return any better result compared to larger sample sizes. This instability needs to be investigated more in the future. With all sample sizes tested, we always obtained 5 peaks which is very satisfying.

For GRS by extension, the results we got were very weak compared to GRS by replacement. However, GRS by extension was not investigated extensively. For future work, it is interesting to understand why this method returned weaker results and had trouble finding even the 4th peak sometimes.

Speed optimization is important in computational science. It is desirable to obtain the similar result within less time and using less points. The objective of GRS is to reduce the execution time and obtain results using different subsamples of the global signal. We believe that with GRS method we could have obtained even better results.

Since GRS method could not always find the 6th peak in coarse isotopic mode, it was not

tried in fine isotopic mode. This is another opportunity to try in the future after obtaining stable results with coarse isotopic mode.

BRAD

The idea of using binary radians came when improvement in precision was needed. The idea was tested with float version of sinus-it. The results showed that there was no significant difference in precision compared to float version without BRAD. However, we believe that BRAD should perform better as it does not involve division by 2π in range reduction and even perfect if integers were used instead of floating point. This method will be investigated further in the future and BRAD with integers will be tested. If we are interested in very large precision, BRAD with integers is what should be used.

QAES

Recently, there is a great interest in using quantum based methods. Quantum based method was also tried on our problem for one case only, finding 3 fine isotopic peaks of the first isotope. As discussed in the previous chapters, this method returned very interesting result. The result was obtained using very small sample size and in a slightly greater than one minute compared to 2 days of computing with sinus-it. QAES method is highly dependent on cases. It requires tuning of several parameters. Since QAES is also a stochastic algorithm, currently its limitation is stability of the results. Once good parameters are found, QAES returns stable result as in the case presented in this thesis. Therefore, this method will be looked into in the near future. As a preliminary result, only one case was tested for this PhD. To study the performance of this method, we also need to test it on coarse isotopic peaks and all other fine isotopic peaks. Also, it is interesting to see how this method will perform with GRS and BRAD, as well as on an experimental data.

Derivatives

Finally, working with derivatives instead of values is another area in improvement that needs to be investigated for harmonic analysis problems in future works.

Now that the basis has been established, there are numerous other refinements can be tested, opening the path to different future research projects, some of which we will personally explore in the future, thanks to my appointment as Associate Professor at UFAZ University.

List of publications and presentations

Journal articles

U. Abdulkarimova, A. Leonteva, C. Rolando, A. Jeannin-Girardon, P. Collet
The PARSEC Machine: a Non-Newtonian Supra-linear Supercomputer, Azerbaijan Journal of High Performance Computing, Vol. 2, Issue 2, 2019, pp. 122-140

Conference Papers

M. A. Haegelin, U. Abdulkarimova, P. Collet, C. Rolando. Super-resolution in FT-ICR MS by non-Fourier Transform genetic evolution signal processing. *ASMS Conference on Mass Spectrometry and Allied Topics*, October 31 – November 4, 2021, Pennsylvania Convention Center, Philadelphia, PA

U. Abdulkarimova, I. Peretta, A. Leonteva, Y. Monjid, R. Amhaz, M. Haegelin, P. Collet, C. Rolando. A GA for non-uniform sampling harmonic analysis. *Proceedings of the 2020 Genetic and Evolutionary Computation Conference Companion*, pages 253-254, Cancun, Mexico, 2020.

A. Leonteva, U. Abdulkarimova, T. Wintermantel, A. Jeannin-Girardon, P. Parrend, P. Collet. A quantum simulation algorithm for continuous optimization. *Proceedings of the 2020 Genetic and Evolutionary Computation Conference Companion*, pages 199-200, Cancun, Mexico, 2020.

A. Leonteva, U. Abdulkarimova, A. Jeannin-Girardon, M. Risser, P. Parrend, P. Collet. Quantum-Inspired Algorithm with Evolution Strategy. *Theory and Practice of Natural Computing*, pages 95-106, Taoyuan, Taiwan, 2020.

Presentations

Evolutionary Computation and its Applications. French Azerbaijani University, April 18, 2019, Baku, Azerbajan.

List of Figures

List of Tables

Bibliography

[1] L. Xu, Y. Zhang, Q. Zhang, X. Wang, X. Chu, X. Li, W. Sui, and F. Han. A simplified strategy for molecular formula determination of chemical constituents in traditional chinese medicines based on accurate mass, A+1 and A+2 isotopic peaks using Fourier transform ion cyclotron resonance mass spectrometry. *Rapid Communications in Mass Spectrometry*, 34(24):e8933. doi: https://doi.org/10.1002/rcm.8933.

[2] L. Z. Samarah, R. Khattar, T. H. Tran, S. A. Stopka, C. A. Brantner, P. Parlanti, D. Veličković, J. B. Shaw, B. J. Agtuca, G. Stacey, L. Paša-Tolić, N. Tolić, C. R. Anderton, and A. Vertes. Single-cell metabolic profiling: Metabolite formulas from isotopic fine structures in heterogeneous plant cell populations. *Analytical Chemistry*, 92(10):7289–7298, 2020. doi: 10.1021/acs.analchem.0c00936. PMID: 32314907.

[3] E. O. Brigham. *The Fast Fourier Transform and Its Applications*. Prentice-Hall, Inc., USA, 1988. ISBN 0133075052.

[4] S. L. Brunton and J. N. Kutz. *Data-Driven Science and Engineering: Machine Learning, Dynamical Systems, and Control*. Cambridge University Press, USA, 1st edition, 2019. ISBN 1108422098.

[5] M. Cerna and A. Harvey. The fundamentals of FFT-Based Signal Analysis and Measurement. *National Instruments, Application Note 041*, 2000.

[6] M. B. Comisarow and A. G. Marshall. Frequency-sweep Fourier Transform Ion Cyclotron resonance spectroscopy. *Chemical Physics Letters*, 26(4):489–490, 1974. ISSN 0009-2614. doi: https://doi.org/10.1016/0009-2614(74)80397-0.

[7] E. N. Nikolaev, Y. I. Kostyukevich, and G. N. Vladimirov. Fourier transform ion cyclotron resonance (FT ICR) mass spectrometry: Theory and simulations. *Mass Spectrometry Reviews*, 35(2):219–258, 2016. doi: https://doi.org/10.1002/mas.21422.

[8] N. Wang. Negative ion electron capture dissociation (niECD): A Novel Tandem Mass Spectrometric Technique. Master's thesis, University of Michigan, 2014.

[9] FTICR Theory: A Brief Review, 2019. URL `https://panomics.pnnl.gov/training/tutorials/ft_icr_tutorial.stm`.

[10] Y. Qi and P. B. O'Connor. Data processing in Fourier transform ion cyclotron resonance mass spectrometry. *Mass Spectrometry Reviews*, 33(5):333–352, 2014. doi: https://doi.org/10.1002/mas.21414.

[11] B. L. Heath. Fourier Transform Ion Cyclotron Resonance Mass Spectrometry (FTICR-MS) for the Study of Noncovalent Complexes. Master's thesis, University of Toronto, 2012.

[12] M. A. van Agthoven, Y. P. Y. Lam, P. B. O'Connor, C. Rolando, and M. A. Delsuc. Two-dimensional mass spectrometry: new perspectives for tandem mass spectrometry. *European Biophysics Journal*, 48, 03 2019. doi: 10.1007/s00249-019-01348-5.

[13] D. P. A. Kilgour, M. J. Neal, A. J. Soulby, and P. B. O'Connor. Improved optimization of the Fourier transform ion cyclotron resonance mass spectrometry phase correction function using a genetic algorithm. *Rapid Communications in Mass Spectrometry*, 27 (17):1977–1982, 2013. doi: https://doi.org/10.1002/rcm.6658.

[14] A. G. Marshall and G. S. Hendrickson, C. L. Jackson. Fourier transform ion cyclotron resonance mass spectrometry: a primer. *Mass spectrometry reviews*, 17(1):1–35, 1998. doi: https://doi.org/10.1002/(SICI)1098-2787(1998)17:1⟨1::AID-MAS1⟩3.0.CO;2-K.

[15] F. Xian, C. L. Hendrickson, G. T. Blakney, and A. G. Beu, S. C. Marshall. Automated broadband phase correction of Fourier transform ion cyclotron resonance mass spectra. *Analytical chemistry*, 82(21):8807–8812, 2010. doi: https://doi.org/10.1021/ac101091w.

[16] D. Kilgour. Absorption mode for FT-MS, 2020. URL `http://www.kilgourlab.com/absorption-mode-for-ft-ms/`.

[17] Yulin Qi, Christopher Thompson, Steve Orden, and Peter O'Connor. Phase correction of fourier transform ion cyclotron resonance mass spectra using matlab. *Journal of the American Society for Mass Spectrometry*, 22:138–47, 01 2011. doi: 10.1007/s13361-010-0006-7.

[18] Steven C. Beu, Greg T. Blakney, John P. Quinn, Christopher L. Hendrickson, and Alan G. Marshall. Broadband phase correction of ft-icr mass spectra via simultaneous excitation and detection. *Analytical Chemistry*, 76(19):5756–5761, 2004. doi: 10.1021/ac049733i.

[19] M. Scigelova, M. Hornshaw, and A. Giannakopulos, A. Makarov. Fourier transform mass spectrometry. *Molecular cellular proteomics : MCP.*, 10(7).

[20] P. Beauchamp. Basics of spectroscopy, url = `https://www.cpp.edu/~psbeauchamp/pdf_book/MS_chapter.pdf`, urldate = 2021-07-16.

[21] E. Denisov, E. Damoc, and A. Makarov. Exploring frontiers of orbitrap performance for long transients. *International Journal of Mass Spectrometry*, 466:116607, 2021. ISSN 1387-3806. doi: https://doi.org/10.1016/j.ijms.2021.116607.

[22] J. Romson and A. Emmer. Mass calibration options for accurate electrospray ionization mass spectrometry. *International Journal of Mass Spectrometry*, 467:116619, 2021. ISSN 1387-3806. doi: https://doi.org/10.1016/j.ijms.2021.116619.

[23] D. C. Muddiman and A. L. Oberg. Statistical Evaluation of Internal and External Mass Calibration Laws Utilized in Fourier Transform Ion Cyclotron Resonance Mass Spectrometry. *Analytical Chemistry*, 77(8):2406–2414, 2005. doi: 10.1021/ac048258l.

[24] L. K. Zhang, D. Rempel, B. N. Pramanik, and M. L. Gross. Accurate mass measurements by Fourier transform mass spectrometry. *Mass Spectrometry Reviews*, 24(2): 286–309. doi: https://doi.org/10.1002/mas.20013.

[25] J. C. Mobli, M. Hoch. Nonuniform sampling and non-Fourier signal processing methods in multidimensional NMR. *Rapid Communications in Mass Spectrometry*, 83: 21–41, 2014. doi: https://doi.org/10.1016/j.pnmrs.2014.09.002.

[26] P. Marziliano and M. Vetterli. Reconstruction of irregularly sampled discrete-time bandlimited signals with unknown sampling locations. *IEEE Transactions on Signal Processing*, 48(12):3462–3471, 2000. doi: 10.1109/78.887038.

[27] T. Strohmer. Numerical analysis of the non-uniform sampling problem. *Journal of Computational and Applied Mathematics*, 122(1):297–316, 2000. ISSN 0377-0427. doi: https://doi.org/10.1016/S0377-0427(00)00361-7. Numerical Analysis in the 20th Century Vol. II: Interpolation and Extrapolation.

[28] M. G. Williams, C. P. Marshall. Hartley/Hilbert Transform Spectroscopy: Absorption-Mode Resolution with Magnitude- Mode Precision. *Analytical Chemistry*, 64:916–923, 1992.

[29] C. P. Williams and A. G. Marshall. Hartley transform ion cyclotron resonance mass spectrometry. *Analytical Chemistry*, 61(5):428–431, 1989. doi: 10.1021/ac00180a010.

[30] M. G. Meier, J. E. Marshall. Bayesian versus Fourier Spectral Analysis of Ion Cyclotron Resonance Time-Domain Signals. *Analytical Chemistry*, 62:201–208, 1990.

[31] M. G. Guan, S. Marshall. Linear Prediction Cholesky Decomposition vs Fourier Transform Spectral Analysis for Ion Cyclotron Resonance Mass Spectrometry. *Analytical Chemistry*, 69:1156–1162, 1997.

[32] P. M. Ramos, M. Fonseca da Silva, R.C. Martins, and A.M.C. Serra. Simulation and experimental results of multiharmonic least-squares fitting algorithms applied to periodic signals. *IEEE Transactions on Instrumentation and Measurement*, 55(2): 646–651, 2006. doi: 10.1109/TIM.2006.864260.

[33] IEEE Standard for Digitizing Waveform Recorders. *IEEE Std 1057-1994*, 1994. doi: 10.1109/IEEESTD.1994.122649.

[34] IEEE Standard for Terminology and Test Methods for Analog-to-Digital Converters. *IEEE Std 1241-2010 (Revision of IEEE Std 1241-2000)*, pages 1–139, 2011. doi: 10.1109/IEEESTD.2011.5692956.

[35] G. H. Golub and C. F. V. Loan. *Matrix Computations*. Johns Hopkins Univ Press, Baltimore, 4th edition, 1996.

[36] J. A. Cadzow and M. M. Wu. Analysis of transient data in noise. *IEE Proceedings F: Communications Radar and Signal Processing*, 134(1):69–78, February 1987.

[37] L. Chiron, M. Agthoven, B. Kieffer, C. Rolando, and M. A. Delsuc. Efficient denoising algorithms for large experimental datasets and their applications in Fourier transform ion cyclotron resonance mass spectrometry. *Proceedings of the National Academy of Sciences of the United States of America*, 111:1385–1390, 01 2014. doi: 10.1073/pnas.1306700111.

[38] F. Bray, J. Bouclon, L. Chiron, M. Witt, M. A. Delsuc, and C. Rolando. Nonuniform Sampling Acquisition of Two-Dimensional Fourier Transform Ion Cyclotron Resonance Mass Spectrometry for Increased Mass Resolution of Tandem Mass Spectrometry Precursor Ions. *Analytic Chemistry*, pages 8589–8593, 08 2017.

[39] T. Markovich, S. Blau, J. Sanders, and A. Aspuru-Guzik. Benchmarking Compressed Sensing, Super-Resolution, and Filter Diagonalization. *International Journal of Quantum Chemistry*, 116:1097–1106, 02 2015. doi: 10.1002/qua.25144.

[40] A. M. Vladimir, S. T. Howard, and A. J. Shaka. Application of the filter diagonalization method to one- and two-dimensional nmr spectra. *Journal of Magnetic Resonance*, 133(2):304–312, 1998. doi: https://doi.org/10.1006/jmre.1998.1476.

[41] Y. O. Kozhinov, A. N. Tsybin. Filter Diagonalization Method-Based Mass Spectrometry for Molecular and Macromolecular Structure Analysis. *Analytical Chemistry*, 84 (6):2850–2856, 2012. doi: https://doi.org/10.1021/ac203391z.

[42] W. Y. Choy and B. C. Sanctuary. Using Genetic Algorithms with a Priori Knowledge for Quantitative NMR Signal Analysis. *Journal of Chemical Information and Computer Sciences*, 38(4):685–690, 1998. doi: 10.1021/ci980017p.

[43] N. Zamanan, J. K. Sykulski, and A. K. Al-Othman. Real coded genetic algorithm compared to the classical method of fast fourier transform in harmonics analysis. In *Proceedings of the 41st International Universities Power Engineering Conference*, volume 3, pages 1021–1025, 2006. doi: 10.1109/UPEC.2006.367633.

[44] J. Majewski and R. Wojtyna. Application of evolutionary algorithm to signal analysis in frequency domain. In *2015 Signal Processing: Algorithms, Architectures, Arrangements, and Applications (SPA)*, pages 12–15, 2015. doi: 10.1109/SPA.2015.7365105.

[45] J. Majewski and R. Wojtyna. Evolutionary algorithm for transformation of short-time signal into frequency-domain description. In *2016 Signal Processing: Algorithms, Architectures, Arrangements, and Applications (SPA)*, pages 30–33, 2016. doi: 10.1109/SPA.2016.7763582.

[46] J. Majewski and R. Wojtyna. Results of applying evolutionary algorithms to frequency-domain signal analysis. In *2017 Signal Processing: Algorithms, Architectures, Arrangements, and Applications (SPA)*, pages 121–124, 2017. doi: 10.23919/SPA.2017.8166850.

[47] F. M. Janeiro and P. M. Ramos. Multiharmonic Waveform Fitting of Periodic Signals using Genetic Algorithms. In *2007 IEEE Instrumentation Measurement Technology Conference IMTC 2007*, pages 1–6, 2007. doi: 10.1109/IMTC.2007.379361.

[48] D. P. Kilgour, R. Wills, and P. B. Qi, Y. O'Connor. Autophaser: an algorithm for automated generation of absorption mode spectra for FT-ICR MS. *Analytical chemistry*, 85(8):3903–3911, 2013. doi: https://doi.org/10.1021/ac303289c.

[49] J. R. Koza. *Genetic Programming II: Automatic Discovery of Reusable Programs.* MIT Press, Massachussetts, 1994.

[50] J. A. Rezaee and J. Jasni. Parameter selection in particle swarm optimisation: A survey. *Journal of Experimental Theoretical Artificial Intelligence*, 25, 12 2013. doi: 10.1080/0952813X.2013.782348.

[51] S. Almufti, A. Zebari, and H. Omer. A comparative study of particle swarm optimization and genetic algorithm. 8:40–45, 10 2019. doi: 10.14419/jacst.v8i2.29401.

[52] Jun S., Bin F., and Wenbo X. Particle swarm optimization with particles having quantum behavior. In *Proceedings of the 2004 Congress on Evolutionary Computation (IEEE Cat. No.04TH8753)*, volume 1, pages 325–331 Vol.1, 2004. doi: 10.1109/CEC.2004.1330875.

[53] D. Fogel. *Evolutionary Computation: The Fossil Record.* Wiley-IEEE Press, 1998. ISBN 0-7803-3481-7.

[54] A. S. Fraser. Simulation of genetic systems by automatic digital computers. In *Australian Journal of Biological Sciences*, volume 10, pages 484–491, 1957.

[55] R. M. Friedberg. A learning machine: Part I. *IBM Research Journal*, 2(1), 1958.

[56] R. M. Friedberg, B. Dunham, and J.H. North. A learning machine: Part II. *IBM Research Journal*, 3(3), 1958.

[57] G. Friedman. *Digital Simulation of an Evolutionary Process*, volume 4, pages 171–184. 1959.

[58] I. Rechenberg. *Evolutionstrategie: Optimierung Technisher Systeme nach Prinzipien des Biologischen Evolution.* Fromman-Hozlboog Verlag, Stuttgart, 1973.

[59] H. P. Schwefel. *Numerical Optimization of Computer Models.* John Wiley & Sons, New-York, 1981. 1995 – 2^{nd} edition.

[60] J. H. Holland. *Adaptation in natural and artificial systems.* University of Michigan Press, Ann Arbor, 1975.

[61] D. E. Goldberg. *Genetic algorithms in search, optimization and machine learning.* Addison Wesley, 1989.

[62] N. L. Cramer. A representation for the adaptive generation of simple sequential programs. In *Proceedings Of an International Conference on Genetic Algorithms and their Applications*, pages 183–187, 1985.

[63] J. R. Koza. *Genetic Programming: On the Programming of Computers by means of Natural Evolution.* MIT Press, Massachusetts, 1992.

[64] M. Kommenda, G. Kronberger, St. Winkler, M. Affenzeller, S. Wagner, L. Schickmair, and B. Lindner. Application of genetic programming on temper mill datasets. pages 58–62, 09 2009. doi: 10.1109/LINDI.2009.5258766.

[65] C. Darwin. *On the Origin of Species by Means of Natural Selection or the Preservation of Favoured Races in the Struggle for Life.* John Murray, London, 1859.

[66] P. Collet and J. P. Rennard. Stochastic optimization algorithms. *Handbook of Research on Nature Inspired Computing for Economics and Management*, 04 2007.

[67] J. E. Baker. Adaptive selection methods for genetic algorithms. *Proceedings of an International Conference on Genetic Algorithms and their Applications*, pages 100–111, 1985.

[68] J. E. Baker. Reducing bias and inefficiency in the selection algorithm. pages 14–21, 1987.

[69] D. Whitley. The GENITOR algorithm and selection pressure: Why rank-based allocation of reproductive trials is best. pages 116–121. Morgan Kaufmann, 1989.

[70] H. Mühlenbein and D. Schlierkamp-Voosen. The science of breeding and its application to the breeder genetic algorithm (bga). *Evolutionary Computation*, 1(4):335–360, 1993.

[71] A. Brindle. Genetic algorithms for function optimisation. Technical Report TR81-2, Dept. of Computer Science, University of Alberta, Edmonton, 1981.

[72] T. Blickle and L. Thiele. A mathematical analysis of tournament selection. pages 9–16, 1995.

[73] A. Urso, A. Fiannaca, M. La Rosa, V. Ravì, and R. Rizzo. *Data Mining: Prediction Methods.* 01 2018. ISBN 9780128096338. doi: 10.1016/B978-0-12-809633-8.20462-7.

[74] O. Maitre. *GPGPU for Evolutionary Algorithms.* PhD thesis, University of Strasbourg, 9 2011.

[75] S. Tsutsui and P. Collet. *Massively Parallel Evolutionary Computation on GPGPUs.* 12 2013. ISBN 978-3-642-37959-8. doi: 10.1007/978-3-642-37959-8.

[76] P. Collet, E. Lutton, M. Schoenauer, and J. Louchet. In *Parallel Problem Solving from Nature PPSN VI.*

[77] F. Kurger, O. Maitre, S. Jiménez, L. Baumes, and P. Collet. Speedups between× 70 and× 120 for a Generic Local Search (Memetic) Algorithm on a Single GPGPU Chip. *Applications of Evolutionary Computation*, pages 501–511, 2010.

[78] O. Maitre, N. Lachiche, P. Clauss, L. Baumes, A. Corma, and P. Collet. Efficient Parallel Implementation of Evolutionary Algorithms on GPGPU Cards. *Euro-Par 2009 Parallel Processing*, pages 974–985, 2009.

[79] O. Maitre, S. Querry, N. Lachiche, and P. Collet. EASEA parallelization of tree-based genetic programming. *IEEE CEC 2010*, 2010.

[80] O. Maitre, L. A. Baumes, N. Lachiche, A. Corma, and P. Collet. Coarse grain parallelization of evolutionary algorithms on gpgpu cards with easea. In *GECCO '09: Proceedings of the 11th Annual conference on Genetic and evolutionary computation*, pages 1403–1410, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-325-9. doi: http://doi.acm.org/10.1145/1569901.1570089.

[81] U. Abdulkarimova, A. Leonteva, A. Jeannin-Girardon, and P. Collet. The parsec machine: A non-newtonian supra-linear supercomputer. *Azerbaijan Journal of High Performance Computing*, 2:122–140, 12 2019. doi: 10.32010/26166127.2019.2.2.122.140.

[82] H. G. Beyer and H. P. Schwefel. Evolution strategies - a comprehensive introduction. *Natural Computing*, 1:3–52, 03 2002. doi: 10.1023/A:1015059928466.

[83] N. Hansen and A. Ostermeier. Adapting arbitrary normal mutation distributions in evolution strategies: the covariance matrix adaptation. In *Proceedings of IEEE International Conference on Evolutionary Computation*, pages 312–317, 1996. doi: 10.1109/ICEC.1996.542381.

[84] O. Maitre, N. Lachiche, P. Clauss, L. Baumes, A. Corma, and P. Collet. Efficient parallel implementation of evolutionary algorithms on gpgpu cards. In *Euro-Par*, 2009.

[85] P. Collet, F. Kruger, and O. Maitre. *Automatic parallelization of EC on GPGPUs and clusters of GPGPU machines with EASEA and EASEA-CLOUD*, pages 35–59. 07 2013. ISBN 978-3-642-37958-1. doi: 10.1007/978-3-642-37959-8_3.

[86] G. M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*, AFIPS '67 (Spring), page 483–485, New York, NY, USA, 1967. Association for Computing Machinery. ISBN 9781450378956. doi: 10.1145/1465482.1465560. URL `https://doi.org/10.1145/1465482.1465560`.

[87] J. Gustafson. Reevaluating amdahl's law. *Commun. ACM*, 31:532–533, 05 1988. doi: 10.1145/42411.42415.

[88] M. Krafczyk, A. Shi, A. Bhaskar, D. Marinov, and V. Stodden. Learning from reproducing computational results: introducing three principles and the reproduction package. *Philosophical Transactions of the Royal Society A*, 379, 2021.

[89] M. Taschuk and G. Wilson. Ten simple rules for making research software more robust. *PLOS Computational Biology*, 13(4):1–10, 04 2017. doi: 10.1371/journal.pcbi.1005412. URL `https://doi.org/10.1371/journal.pcbi.1005412`.

[90] K. C. Ng and D. O'Connor. Argument reduction forhuge arguments: Good to the last bit. 2006.

[91] W. Cody and W. Waite. *Software Manual for the Elementary Functions*. Prentice-Hall, Englewood Cliffs, N.J., 1980.

[92] J. Muller. *Elementary Functions: Algorithms and Implementation*. Birkhäuser, 2 edition.

[93] J. M. Matos. The historical development of the concept of angle (2). *The Mathematics Educator*, 1, 1990.

[94] R. Smith. *Harmonia mensurarum*. Cambridge, England, 1722.

[95] X. Maolong, W. Xiaojun, S. Xinyi, S. Jun, and X. Wenbo. Improved quantum-behaved particle swarm optimization with local search strategy. *Journal of Algorithms & Computational Technology*, 11(1):3–12, 2017. doi: 10.1177/1748301816654020.

[96] A. Leonteva, U. Abdulkarimova, A. Jeannin-Girardon, M. Risser, P. Parrend, and P. Collet. Quantum-Inspired Algorithm with Evolution Strategy. In *Theory and Practice of Natural Computing*, pages 95–106. Springer International Publishing, 2020.

[97] A. Leonteva, U. Abdulkarimova, T. M. Wintermantel, A. Jeannin-Girardon, P. Parrend, and P. Collet. A quantum simulation algorithm for continuous optimization. In *Proceedings of the 2020 Genetic and Evolutionary Computation Conference Companion*, GECCO '20, page 199–200, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450371278. doi: 10.1145/3377929.3389990. URL `https://doi.org/10.1145/3377929.3389990`.

[98] J. R. Koza and al. *Genetic Programming IV: Routine Human-Competitive Machine Intelligence*. Kluwer Academic Publishers, 2003.

[99] O. Maitre, D. Sharma, N. Lachiche, and P. Collet. In *EvoApplications 2011*, Heidelberg.

[100] Pierre Collet and Marc Schoenauer. GUIDE: Unifying evolutionary engines through a graphical user interface. In *P. Liardet et al., eds, EA'03*, volume 2936 of *LNCS*, pages 203–215, Marseilles, 2003. Springer. ISBN 3-540-21523-9. doi: doi: 10.1007/b96080. URL `http://springerlink.metapress.com/openurl.asp?genre=article&issn=0302-9743&volume=2936&spage=229`.

Appendix A

Résumé en français de la thèse

Introduction

L'amélioration de la technologie a vu la taille et la complexité des jeux de données augmenter considérablement dans de nombreuses disciplines scientifiques ces dernières années. Le résultat est qu'il est devenu très difficile de continuer à travailler avec des méthodes traditionnelles conçues pour des tailles de données plus restreintes. Les domaines concernés par cette inflation de données sont l'imagerie médicale, la physique, la science des matériaux, la télédétection...

L'analyse spectrale est l'une des branches principales du traitement du signal. De nombreux appareils expérimentaux produisent des signaux qui sont des sommes de sinus amorties (Résonance Magnétique Nucléaire, Imagerie par Réonance Magnétique, Spectrométrie de Masse à Résonance Cyclonique Ionique (ICR-MS)), sismomètres, etc. Mais avec les perfectionnements de ces appareils, le volume de données qu'ils produisent ne cesse de grandir. Le spectre d'un ICR-MS actuel contient typiquement 16 méga points de données et 10 000 sinus. Parmi les différentes méthodes utilisées pour analyser le spectre d'un signal discret, la plus standard est une transformée rapide de Fourier (FFT), qui permet de décomposer le signal en une somme de sinus, pour en trouver les harmoniques.

Les chimistes utilisent la spectrométrie de masse pour étudier les molécules et les composés pour déterminer leur structure isotopique e leur abondance relative. Les composés peuvent comporter plusieurs structures isotopiques différentes, identifiées par des pics larges dans le spectre, et chacune de ces structures peut comporter des variations fines, qui servent à déterminer la composition chimique du composé. Les principales structures isotopiques diffèrent par leur nombre de neutrons, mais pour chaque configuration différente, la masse des neutrons est différente car leur énergie de liaison au noyau atomique est différente. C'est cette différence subtile de masse dûe à l'énergie de liaison qui est détectée dans les pics fins du spectre. Non seulement ces pics fins sont de très faible amplitude, mais certains sont très proches les uns des autres en fréquence, ce qui les rend très difficiles à détecter.

La spectrométrie de masse par cyclotron à ions a deux principaux usages :

1. La détermination, de la composition atomique pour un composé inconnu. A cet effet,

le rapport signal/bruit ne peut pas descendre en dessous de 1.

2. La quantification, pour un composé connu, du nombre de molécules de compositions isotopiques différentes. Un rapport signal-bruit plus faible (de l'ordre de 0.1) est alors acceptable pour détecter les pics secondaires, car on sait déjà où les chercher.

Dans le premier cas, on s'intéresse à la qualité (où se trouvent les pics) et dans le second cas, à la quantité (quelle amplitude pour chaque pic).

Dans cette thèse, nous nous concentrerons sur des données issues d'un spectromètre de masse à résonance cyclonique ionique (ICR-MS) sur la substance P ainsi que sur des données simulées. Dans les ICR-MS, l'analyse effectuée consiste à mesurer le rapport masse/charge de molécules ionisées, en se basant sur leur fréquence cyclotron obtenue en les faisant tourner dans un champ électromagnétique variable de plusieurs Teslas, pour trouver la composition du produit analysé. Comme dit précédemment, la technique habituellement utilisée pour l'analyse est la transformée de Fourier, bien qu'elle présente plusieurs limitations, car le bruit pose problème et le signal doit être apodisé pour gérer les oscillations induites par le démarrage et l'arrêt du signal, du fait de sa taille finie. De plus, l'appareil ne fournissant qu'une composante réelle au nombre complexe que nécessite une transformée de fourier, il est impossible de déterminer la phase des sinus trouvés.

Le principal problème est celui du bruit, dû aux imperfections des machines et qui diminue la performance d'une analyse mathématique parfaite comme une analyse de Fourier. De ce fait, les données brutes doivent être débruitées autant que possible avant de pouvoir être utilisées. En effet, une analyse mathématique (et donc déterministe et parfaite) d'un signal bruité amènerait à une décomposition en un nombre de sinus infini, pour modéliser exactement le bruit. Pour gérer ce problème, de nombreuses approches existent, mais la taille des données produites est très grande (plusieurs millions de points) ce qui rend le débruitage coûteux en temps. En 2014, Chiron *et. al.* ont proposé un algorithme de débruitage de signaux harmoniques appelé urQRd (*uncoiled random QR denoising*) qui fonctionne bien plus rapidement qu'une approche classique SVD (*Single Value Decomposition*) et qui peut être appliquée aux très gros volumes de données produites par des ICR-MS ou autres spectroscopes à haute résolution, mais malgré la performance de l'algorithme, le temps de débruitage sur un signal comportant des millions de points se compte en semaines, sur des serveurs Xeon-Phi comportant un grand nombre de coeurs.

Le signal obtenu par ICR-MS est mesuré à intervalles temporels réguliers. La précision du spectre obtenu dépend de la méthode d'échantillonnage utilisée pour obtenir le signal. Un échantillonnage uniforme est utilisé dans des expériences uni-dimensionnelles. Cependant, dans le cas d'échantillonnages multi-dimensionnels, des méthodes non-uniformes sont préférables. Plusieurs méthodes Fourier et non-Fourier ont été imagniées pour gérer ces échantillonnages non-uniformes, mais la plupart de ces méthodes sont limitées par une importante complexité temporelle, des limitation sur la taille des échantillons, et une mauvaise résolution lorsque le rapport signal/bruit est faible.

Notre apport consistera à explorer l'apport de méthodes évolutionnaires (stratégies d'évolution, programmation génétique) pour dépasser les limitations des autres méthodes existantes dans l'état de l'art.

L'évolution artificielle se base sur la théorie darwinienne de l'évolution naturelle, qui postule que les individus les plus adaptés ont plus de chance de survivre dans leur envi-

ronnement et de se reproduire pour créer une nouvelle génération d'individus. L'évolution artificielle et la programmation génétique produisent régulièrement des résultats compétitifs avec l'intelligence humaine sur de nombreux problèmes depuis le début des années 2000.

SINUS-IT (l'algorithme principal développé dans le cadre de cette thèse) optimise des valeurs réelles pour coder l'amplitude, la fréquence et la phase des sinus qui composent le signal, mais utilise une structure fonctionnelle d'algorithme génétique, car il met en œuvre principalement un croisement monopoint et un opérateur de mutation.

La principale raison de l'utilisation d'un tel moteur évolutionnaire comparé à une stratégie d'évolution comme CMA-ES (Covariance Matrix Adaptation Evolutionary Strategy) dont les opérateurs sont conçus pour travailler sur des problèmes généraux, est que dans le cas de l'analyse harmonique, le problème est bien défini et connu, ce qui signifie qu'on peut concevoir des opérateurs de croisement et de mutation spécifiques, utilisant une compréhension profonde du problème, permettant un très bon compromis exploration / exploitation.

De plus, notre approche utilise un algorithme massivement parallèle qui tourne sur des cartes GPGPU, ce qui lui permet d'estimer directement les paramètres des sinus qui composent le signal sans utiliser de transformée de Fourier, ce qui apporte plusieurs avantages:

1. on peut directement trouver la phase (ce que ne peut trouver la transformée de Fourier dans le cas où (et c'est le cas pour l'ICR-MS) l'appareillage ne permet pas d'obtenir la partie imaginaire du nombre complexe nécessaire à la transformée de Fourier pour trouver la phase),

2. cette méthode n'étant pas déterministe, elle n'a pas besoin de de phase de débruitage et peut travailler directement sur un signal brut,

3. aucune apodisation n'est nécessaire.

Approche actuelle du traitement de données de la spectroscopie de masse

Dans les spectromètres de masse à résonance cyclonique ionique (ICR-MS), le signal es habituellement analysé par un ordinateur faisant partie intégrante de la machine, qui effectue une transformée rapide de Fourier, qui extrait le spectre en fréquence avant qu'il soit converti en spectre de masse.

L'analyse harmonique basée sur la transformée de fourier est utilisée de manière extensive bien qu'elle présente plusieurs limitations :

- Le signal doit être apodisé pour gérer le début et la fin du signal, du fait qu'il est de taille finie.

- Les imperfections de la machine qui échantillonne le signal ajoute un bruit qui empêche des algorithmes mathématiquement parfaits (tel qu'une transformée de Fourier) de donner de bons résultats. En effet, une modélisation parfaite d'un signal bruité amènerait à une décomposition en un nombre de sinus infinis, pour modéliser le bruit. Pour répondre à ce problème, de nombreuses approches mathématiques utilisent une première phase de débruitage pour améliorer le rapport signal / bruit.

- Une autre limite de la méthode par transformée de Fourier est que là où, pour la résonance magnétique nucléaire, la détection quadratique donne 2 signaux, un réel et un autre imaginaire déphasé de $\pi/2$ représentant la partie imaginaire, les ICR-MS ne renvoient que le premier signal, empêchant la transformée de Fourier de trouver la phase. Pour cette raison, les spectres en amplitudes sont les plus communément utilisés. Connaître le paramètre de la phase permettrait d'afficher le spectre en mode d'absorption, ce qui améliorerait la précision sur la masse, le pouvoir de résolution du système et améliorerait le rapport signal/bruit.

- Enfin, les transformées de Fourier ont besoin de longues transitoires pour résoudre les pics. L'intérêt dans notre méthode est de pouvoir utiliser des échantillons de taille plus réduite car plus le bruit augmente, plus il devient prépondérant par rapport au signal, surtout dans la queue de la transitoire. On voit donc qu'il est désirable d'utiliser des transitoires plus courtes pour garder un rapport signal/bruit raisonnable.

Approche proposée dans cette thèse pour analyser les données d'un ICR-MS

Pour dépasser les limites des FFT présentées ci-dessus, nous proposons d'utiliser des algorithmes évolutionnaires (EAs).

Le principal objectif de cette thèse est d'étudier leur performance en comparaison avec les FFT et d'utiliser les données provenant d'une machine ICR-MS pour valider leur usage dans un cas réel, en déterminant la structure principale et la structure isotopic fine de composés donnés, en utilisant de plus petites transitoires. L'algorithme SINUS-IT sera testé sur des données simulées de la substance P (un neuropeptide composé de 11 acides aminés, *cf.* `https://en.wikipedia.org/wiki/Substance_P`, de formule chimique $C_{63}H_{98}N_{18}O_{13}$ ainsi que sur des données réelles provenant de la machine ICR-MS.

Les algorithmes évolutionnaires sont des algorithmes stochastiques basés sur la théorie Darwinienne de l'évolution, qui propose que les invididuts les plus adaptés ont plus de chance de survivre et de se reproduire, et de voir leur lignée persévérer au cours des générations.

Les algorithmes évolutionnaires sont des algorithmes stochastiques basés sur la théorie de l'évolution de Darwin, selon laquelle les individus les plus aptes ont plus de chances de survivre et de produire la descendance de la génération suivante. Ils sont connus pour produire des résultats compétitifs pour l'homme à de nombreux problèmes difficiles. Dans cette thèse, nous proposons d'utiliser un algorithme génétique massivement parallèle codé en réel qui fonctionne sur des cartes GPGPU pour estimer directement les paramètres des fonctions sinusoïdales (y compris la phase) qui composent le signal d'un MS FT-ICR, remplaçant ainsi le besoin de FFT et évitant une étape de détermination de la phase. L'algorithme produit une séquence de générations sur lesquelles évolue une population de solutions potentielles au problème. Chaque génération consiste en une population avec différentes solutions potentielles. À chaque génération, les solutions sont évaluées en fonction de leur "aptitude" à résoudre le problème. Les solutions ayant la plus petite valeur de fitness (le fitness calcule l'erreur), ont plus de chance d'être sélectionnées comme parent pour partager leurs gènes avec d'autres membres de la population afin de produire une solution enfant.

Les GA sont bien adaptés au problème de l'analyse harmonique car il s'agit d'un problème bien défini dont les caractéristiques sont bien connues. Cela signifie que des opérateurs de croisement et de mutation finement ajustés (conçus pour exploiter les zones intéressantes de l'espace de recherche) peuvent être créés, sur la base d'une compréhension approfondie du problème. Les résultats sont trouvés par la capacité d'exploration / exploitation d'une très grande population, en utilisant différents niveaux de parallélisme.

L'approche que nous proposons utilise un GA massivement parallèle codé en réel qui s'exécute sur des cartes GPGPU pour estimer directement les paramètres des fonctions sinusoïdales qui composent le signal d'un MS FT-ICR, remplaçant ainsi la nécessité d'une FFT et évitant une étape de détermination de la phase. Comme la FFT ne peut pas trouver la phase pour les données FT-ICR, dans certaines applications, le paramètre de phase est ignoré. Il existe également plusieurs méthodes de correction de phase proposées dans la littérature, mais elles sont gourmandes en ressources informatiques.

Afin d'étudier les performances de notre algorithme, nous le testons d'abord sur des données simulées puisque ses paramètres sont connus. Les résultats obtenus sont comparés avec la méthode FFT.

Informatique évolutive

Les algorithmes évolutionnaires sont d'anciennes techniques d'optimisation inspirées de la théorie de l'évolution biologique de Darwin, décrite dans son célèbre ouvrage intitulé "De l'origine des espèces" (1859).

Les premières adaptations de ces travaux aux ordinateurs ont été réalisées dans les années 1950, comme le montre Fossil Record de David Fogel, avec Fraser, Friedberg et Friedman, qui ont présenté comment des chaînes binaires pouvaient évoluer grâce à des croisements, comment les ordinateurs pouvaient s'autoprogrammer grâce à des mutations et comment l'évolution pouvait être simulée numériquement.

Mais l'évolution a également contribué à sélectionner les algorithmes qui sont aujourd'hui les plus utilisés, à savoir les stratégies évolutives, les algorithmes génétiques et la programmation génétique, pour ne citer que les plus connus.

Ce qu'il faut comprendre, c'est que ces différents algorithmes de la famille de l'évolution artificielle répondent à des problèmes différents.

Stratégies d'évolution (ES) Les *stratégies d'évolution* de Schwefel et Rechenberg sont orientées vers l'optimisation des problèmes d'ingénierie. Par conséquent, dans les ES, les solutions sont représentées comme des vecteurs de réels, qui servent de paramètres aux fonctions à minimiser ou à maximiser. Comme dans tous les paradigmes d'évolution artificielle, les solutions potentielles sont appelées individus. Elles sont regroupées dans une population qui évolue au fil des générations.

Alors que Darwin décrivait l'évolution des animaux et des espèces par des variations non guidées, dont Gregor Mendel a montré plus tard qu'elles étaient le résultat de croisements et de mutations, les algorithmes ES originaux ne reposaient pas tant sur les croisements que sur les mutations. sur les croisements mais plutôt sur les mutations, qui étaient implémentées

comme un bruit gaussien ajouté aux valeurs réelles qui constituaient les solutions potentielles au problème (les individus).

Les *Genetic Algorithms* de Holland et Goldberg ont été développés d'un point de vue plus philosophique, afin de répondre à la question fondamentale : l'évolution peut-elle avoir produit des animaux et des êtres humains complexes ? Par conséquent, AG se concentre sur le développement d'une théorie mathématique de l'évolution, ce qui a des implications sur le codage de ces algorithmes. En général, les solutions sont représentées sous forme de chaînes de bits (un peu comme l'ADN, qui est un brin composé de quatre bases nucléiques azotées différentes : cytosine [C], guanine [G], adénine [A] ou thymine [T]).

Si une représentation en bits signifie que la mutation est très simple (un 0 sera muté en 1 et inversement), la représentation en chaînes de bits des individus rend difficile la représentation des entiers et des valeurs réelles. En effet, il n'existe pas de moyen simple de muter la valeur 7 (0111 en binaire) en valeur 8 (1000 en binaire) qui sont des valeurs voisines, car changer 7 en 8 nécessite 4 mutations. D'autres types de représentations sont alors utilisés, comme un codage de Gray des valeurs binaires, ou une représentation de Dedekin, mais ces représentations posent également leurs problèmes.

Ensuite, représenter des valeurs réelles avec des chaînes de bits est encore plus difficile. Si l'on utilise la représentation arithmétique à virgule flottante standard IEEE 754, la mutation d'un bit dans le codage de l'exposant modifiera considérablement la valeur du nombre réel, voire créera un "Pas un nombre" (`nan`), ou pire encore, une séquence de bits qui ne peut être interprétée comme un nombre IEEE 754.

Les croisements entre individus sont ici encore très inspirés de la génétique, où les croisements à point unique d'individus codés sur b bits sont implémentés en prenant les n premiers bits du parent 1 et en les collant aux $b-n$ bits du second parent pour créer un enfant. Mais là encore, si le locus (le point de croisement) est situé au milieu de la représentation d'une valeur réelle IEEE 754, la valeur résultante chez l'enfant sera très différente des valeurs du parent 1 ou du parent 2.

Programmation génétique (GP) Enfin, en 1985 et 1992, Cramer et Koza ont développé la programmation génétique dont le but n'était pas de trouver des valeurs optimales pour minimiser les fonctions de fitness (optimisation) mais de créer des fonctions qui résoudraient certaines données observées (lois), transférant l'évolution artificielle dans le domaine de l'apprentissage automatique. La GP utilise une représentation basée sur les arbres.

Les principales étapes préparatoires à la GP sont les suivantes :

- Spécification de l'ensemble des bornes, y compris les variables indépendantes du problème, les constantes aléatoires.
- Spécification de l'ensemble des fonctions primitives
- Choisir la mesure de fitness
- Définir les paramètres pour contrôler l'exécution tels que la taille de la population, les probabilités d'exécution du croisement et de la mutation
- Spécifier le critère de fin

Dans la section suivante, nous abordons les étapes des algorithmes évolutionnaires.

Vue d'ensemble du calcul évolutif L'évolution artificielle fournit donc des algorithmes permettant à la fois de peupler les ontologies (en trouvant des paramètres, avec les algorithmes génétiques ou les stratégies d'évolution) et de modéliser les données (en créant des lois), avec la programmation génétique.

Parallélisation des algorithmes évolutionnaires

Les algorithmes évolutifs sont intrinsèquement parallèles. En effet, c'est ce qui se passe dans la nature où la reproduction entre individus d'une même espèce est même asynchrone.

Dans un ordinateur, cependant, la parallélisation asynchrone est plus difficile que la parallélisation synchrone car l'accès à la mémoire partagée doit se faire de manière contrôlée (2 threads ne peuvent pas écrire à la même adresse mémoire simultanément, sinon certaines données seront perdues).

Même si toutes les parties d'un algorithme évolutionnaire peuvent être parallélisées, le point chaud du calcul évolutionnaire est la fonction d'évaluation. En effet, sélectionner des individus, recombiner et faire muter leurs gènes est très simple, comparé à leur évaluation sur des milliers de points de données, et encore plus si l'évaluation implique le calcul de plusieurs sinus par point de données.

Ainsi, dans le cas d'évaluations nécessitant beaucoup de calcul, comme les sommes de sinus sur des milliers de points pour un seul individu, l'évaluation utilisera 99% du temps de calcul total du cycle évolutif.

Parallélisation GPGPU

Les GPGPU (General Purpose Graphic Processing Units) sont des processeurs initialement conçus pour traiter des images composées de millions de pixels, ou des scènes 3D composées de millions de triangles 3D. Ce qui est intéressant, c'est que, que les GPGPU doivent traiter des images en pixels ou des images vectorielles, ils doivent répéter exactement le même algorithme sur les millions d'entités qui les composent. De plus, le traitement qui doit être effectué sur un pixel ou un triangle est indépendant du même traitement qui doit être effectué sur un autre pixel / triangle de l'image.

Les concepteurs d'unités de traitement graphique ont donc conçu des processeurs spécifiques dans lesquels la plupart des transistors sont utilisés pour mettre en œuvre des Unités Arithmétiques et Logiques (ALU), ce qui donne lieu à une architecture étrange, dans le but de mettre en œuvre autant de puissance de calcul que possible dans l'espace limité d'une puce de silicium.

Alors que sur un CPU multi-cœurs, tous les cœurs sont indépendants (ce qui signifie qu'ils peuvent exécuter leurs propres programmes indépendamment), les concepteurs de GPGPU ont choisi de maximiser la puissance de calcul au prix d'une réduction de la polyvalence, ce qui est d'autant plus possible que lors du traitement d'une image d'un million de pixels / sommets, tous les algorithmes exécutés sur tous les pixels ou sommets sont identiques à l'instruction.

Ainsi, une simplification très radicale qui permet de gagner beaucoup d'espace sur la puce est la suivante : plutôt que d'entourer chaque ALU de tout ce dont elle a besoin pour être

indépendante, les concepteurs ont décidé de regrouper un certain nombre de cœurs dans ce qu'ils ont appelé des "multiprocesseurs" et de leur faire partager les unités fonctionnelles qui alimentent les processeurs. En règle générale, une ALU a besoin d'une unité d'extraction et de distribution, pour extraire de la mémoire le prochain opérateur et les opérandes à charger dans l'ALU et les distribuer dans les registres appropriés.

Cela signifie que dans un multiprocesseur à 32 cœurs, tous les cœurs partageront la même unité fonctionnelle de récupération et de distribution, qui récupérera la prochaine instruction à exécuter et la distribuera dans les 32 cœurs du même multiprocesseur, *¿ signifiant que dans un multiprocesseur, tous les cœurs doivent exécuter la même instruction en même temps.*

Cette forme très restrictive de parallélisme est appelée SIMD (Single Instruction Multiple Data) dans la taxonomie Flynn. Elle convient parfaitement aux algorithmes graphiques qui peuvent exécuter la même instruction sur tous les différents pixels d'une image en même temps.

Or, il n'y a pas qu'un seul multiprocesseur dans une puce GPGPU, mais plusieurs, chacun d'entre eux ayant sa propre unité d'extraction et de distribution, ce qui signifie que si tous les cœurs d'un multiprocesseur doivent exécuter exactement la même instruction en même temps (SIMD), plusieurs multiprocesseurs peuvent exécuter différentes parties du même programme en même temps (ils ont tous leur propre compteur de programme). Ainsi, le parallélisme SIMD très restrictif est assoupli dans ce qu'on appelle le SPMD (Single Program Multiple Data), où différents multiprocesseurs peuvent exécuter simultanément différentes fonctions du même programme.

Parallélisation des îlots

L'idée derrière la parallélisation en îlot est d'utiliser plusieurs ordinateurs reliés entre eux par un réseau. Cela peut se faire de plusieurs manières. La méthode standard consisterait à exécuter un seul algorithme sur n machines, ce qui signifie que pour un algorithme de 10 000 individus s'exécutant sur 10 machines, la population pourrait être divisée par 10 et à chaque génération, chacune des 10 machines pourrait évaluer 1000 individus. Mais cela demanderait d'envoyer les 10 000 individus sur les 10 machines à chaque génération, et de récupérer les résultats une fois l'évaluation effectuée. Cela surchargerait périodiquement (à chaque génération) surchargerait le réseau et le temps de transmission et de synchronisation des données ralentirait le calcul, ce qui signifie qu'il ne serait pas possible d'aller 10 fois plus vite en utilisant 10 ordinateurs. On utilise donc une autre façon de paralléliser les algorithmes : la parallélisation en îlot.

Lorsque la parallélisation sur n îlots isolés différents peut donner n résultats différents, interconnecter les îlots de temps en temps est très utile. les îles de temps en temps est très intéressante. En effet, il peut arriver qu'un îlot reste bloqué dans un optimum local. Lorsque cela se produit, recevoir un individu d'une autre île peut permettre à la population de se diversifier à nouveau afin de trouver un meilleur résultat. L'étude de cas est la suivante. Si, après qu'une île se soit retrouvée coincée dans un optimum local, elle reçoit un individu d'une autre île, alors deux cas sont possibles :

1. L'individu entrant a une valeur de fitness inférieure à celle des individus locaux. Dans ce cas, l'individu ne sera pas utilisé. Dans ce cas, l'individu ne sera pas utilisé pour la recombinaison et il sera très probablement retiré de la population au cours de la période

d'ajustement. probablement retiré de la population pendant la phase de "réduction", lors de la création de la nouvelle population par sélection darwinienne et l'îlot restera bloqué dans son optimum local.
2. L'individu entrant a une meilleure valeur de fitness que les individus locaux. Dans ce cas :
(a) il sera très souvent sélectionné parmi les parents pour créer de nouveaux enfants
(b) il sera sélectionné pour survivre jusqu'à la génération suivante et traîner jusqu'à ce que le jusqu'à ce que le génotype moyen de la population de l'île ait migré vers le génotype du bon immigrant. Migrer vers le génotype du bon immigrant signifie que l'île qui était est coincée dans un minimum local.

Brève description de la plateforme EASEA

Pour ce travail, nous avons utilisé la plateforme de calcul évolutionnaire EASEA, créée en 2000 et disponible sur SourceForge et GitHub.

EASEA (EAsy Specification of Evolutionary Algorithms) est une plateforme logicielle dédiée aux algorithmes évolutionnaires. logiciel dédié aux algorithmes évolutionnaires qui, depuis 2008, parallélise automatiquement les EA sur des architectures parallèles, qui vont d'une seule machine équipée d'une GPGPU à des machines multi-GPGPU, à un cluster ou même à plusieurs clusters de machines GPGPU. ou même plusieurs clusters de machines GPGPU.

La plateforme EASEA a été initialement conçue pour aider les utilisateurs à créer des algorithmes évolutifs de pointe. d'algorithmes évolutionnaires de pointe. Elle est conçue pour produire un algorithme évolutionnaire à partir d'une description ou d'une spécification du problème. Cette spécification est écrite dans un langage de type C qui contient le code des opérateurs génétiques (croisement, mutation (croisement, mutation, initialisation et évaluation) et la structure du génome. et la structure du génome. À partir de ces fonctions, écrites dans un fichier *.ez*, EASEA génère un algorithme évolutionnaire complet avec possibilité de parallélisation potentielle de l'évaluation sur GPGPUs, ou sur un cluster de machines machines hétérogènes, grâce au modèle d'îlot intégré.

Le fichier source généré pour l'algorithme évolutionnaire est lisible par l'utilisateur. Il peut être utilisé tel quel, ou comme point de départ, pour être étendu manuellement par un programmeur expert.

La plateforme EASEA implémente non seulement tous les différents types d'algorithmes génétiques, les stratégies d'évolution, mais aussi la programmation génétique et d'autres algorithmes stochastiques tels que CMA-ES, les algorithmes d'optimisation multi-objectifs (NSGA-II, NSGA-III, ASREA, Fast-EMO) mais aussi les algorithmes évolutionnaires et d'essaimage de particules inspirés par les quantums tels que QAES, QPSO et autres.

Pour les algorithmes évolutionnaires multi-objectifs (MOEA), une méthode de classement stochastique spécifique a été développée. spécifique de classement stochastique, qui peut être parallélisée sans impact sur la qualité. impact sur la qualité.

Un algorithme évolutionnaire EASEA est défini par des morceaux de code spécifiques au problème fournis par l'utilisateur. de code spécifiques au problème fournis par l'utilisateur. La structure du génome est, bien sûr, le premier élément nécessaire, suivi de la structure du génome. La structure du génome est, bien sûr, la première pièce nécessaire, suivie des

opérateurs génétiques tels que l'opérateur d'initialisation (qui construit un nouvel individu), l'opérateur de croisement (qui crée un enfant à partir de deux parents), l'opérateur de mutation et enfin, la fonction d'évaluation qui renvoie une valeur proportionnelle à la performance d'un individu sur le problème à résoudre (fitness). résolu (fitness). Le code source d'EASEA (fichier `.ez`) se compose de plusieurs sections, dont beaucoup sont dédiées à la résolution de problèmes. plusieurs sections, dont un grand nombre est dédié à ces opérateurs spécifiques. Différents articles expliquent comment cela comment cela est fait.

Parallélisation EASEA des EAs standards

Pour les algorithmes à objectif unique, le choix initial était de ne paralléliser que l'étape d'évaluation, car cette phase est souvent la plus importante. l'étape d'évaluation seulement, car cette phase est souvent la plus la plus longue de tout l'algorithme. Cela signifie que les versions parallèles et séquentielles d'un algorithme peuvent être complètement identiques, y compris l'étape d'évaluation qui est ensuite exécutée sur plusieurs cœurs.

Il en résulte qu'un code entièrement séquentiel peut être exécuté très efficacement sur une carte GPGPU massivement parallèle contenant des milliers de cœurs.

Pour les algorithmes évolutionnaires multi-objectifs (MOEA), des méthodes de classement stochastiques spécifiques ont été développées, qui peuvent être parallélisées sans impact sur la qualité.

Conception d'algorithmes

Étant donné que l'algorithme fonctionnera sur des cartes GPU NVIDIA RTX 2080Ti 4352-core qui doivent faire tourner de nombreux threads par cœur pour utiliser efficacement son processeur graphique parallèle spatio-temporel SPMD (pour plus de détails sur la façon dont cette architecture spécifique peut être parfaitement exploitée par l'évolution artificielle), nous avons la possibilité d'utiliser une très grande taille de population, qui peut minimiser le problème de convergence prématurée, qui est partagé par tous les algorithmes d'optimisation.

Ceci aura à son tour une influence sur la conception des opérateurs et sur les paramètres de l'algorithme décrit ci-dessous.

Le moteur évolutif est un algorithme génétique codé en réel, où les individus sont représentés comme des vecteurs de réels, mais qui utilise un croisement tiré des algorithmes génétiques.

Il s'agit d'un algorithme extrêmement simple, presque tiré du livre. Ses performances montrent donc la puissance de l'approche évolutionnaire, même si l'obtention des résultats présentés a nécessité un très grand nombre d'exécutions pour ajuster finement ses différents paramètres.

Nous avons implémenté deux versions de l'algorithme. L'une pour déterminer la distribution isotopique grossière (trouver des isotopes avec des nombres différents de neutrons) et l'autre pour déterminer les distributions isotopiques fines (trouver des isotopes avec un nombre identique de neutrons, mais avec des neutrons attachés à différents atomes avec des niveaux d'énergie différents, conduisant à une masse différente en raison de l'équation $E = mc^2$ d'Einstein). L'objectif de chaque version est différent. En mode grossier, on s'intéresse à la détermination des paramètres des pics principaux. En mode fin, nous essayons

de déterminer des pics très proches en fréquences. Par conséquent, certains paramètres de l'algorithme diffèrent en fonction de la version. La différence se situe principalement dans la fonction de mutation et d'évaluation ainsi que dans l'initialisation des paramètres.

Résultats

Nous avons effectué une étude comparative des méthodes par transformée de Fourier et par évolution artificielle. L'analyse se base sur des données de différentes tailles, artificielles ou réelles, avec un niveau de bruit de 100, et avec ou sans amortissement exponentiel.

Nous avons aussi utilisé différentes méthodes d'échantillonage, uniformes, non-uniformes et nous avons proposé une nouvelle méthode que nous avons appelée *Global Random Sampling* (GRS), qui consiste à utiliser de manière répétitive un petit échantillon jamais pris au même endroit, ce qui permet d'utiliser l'ensemble des points de données, mais jamais en une fois, pour diminuer le temps de calcul sans diminuer la qualité. Deux versions de GRS sont étudiées : un GRS par remplacement et un par extension.

Les résultats obtenus avec SINUS-IT sont meilleurs qu'avec une transformée de Fourier, sans pour autant nécessiter de débruitage, d'apodisation, avec en plus l'avantage d'obtenir la phase avec une bonne précision, le tout sur un échantillonnage non-uniforme, ce qui permet de diminuer le temps d'acquisition sur la machine ICR-MS dans le cas multi-dimensionnel, tout en utilisant seulement une petite portion du signal.

Les tests ont été effectués sur des pics isotopiques fins (3 sinus rapprochés d'1 part par million) ainsi que pour trouver les 6 ou 7 pics principaux de la substance P.

À titre d'exemple, nous trouvons avec notre méthode les 6 sinus principaux de la substance P avec seulement 8192 points, là où la méthode FT détecte 6 sinus avec 32678 points.

D'autres pistes ont aussi été explorées comme un algorithme évolutionnaire hybridé avec un algorithme à essaim particulaire inspiré de la physique quantique qui semble très prometteur.

Ce travail montre l'intérêt des approches évolutionnaires massivement parallèles pour l'analyse harmonique car elles ouvrent de nombreuses pistes de recherche, à la fois fondamentales et appliquées pour mieux exploiter les machines actuelles qui fabriquent un tel volume de données que leur limitation pratique ne sera bientôt plus physique, mais liée au temps de traitement informatique des données produites.

Conclusion

Dans ce projet de thèse, nous avons proposé une approche évolutive utilisant un algorithme génétique pour surmonter les limitations de la méthode FFT qui est traditionnellement utilisée dans les machines FT-ICR pour obtenir le spectre de fréquence. Nos résultats montrent que l'approche présentée dans cette thèse est supérieure à la célèbre méthode FFT.

Afin de tester l'analyse harmonique à l'aide de l'évolution artificielle, nous avons utilisé comme test réel des données provenant de machines FT-ICR, qui ne produisent pas la composante imaginaire dont la FFT a besoin pour calculer la phase, ce qui signifie que FT-ICR ne peut pas fournir la phase du signal facilement.

Ensuite, les algorithmes génétiques étant de nature stochastique, on a supposé qu'ils seraient mieux équipés pour traiter des données bruyantes que la FFT qui est une méthode mathématique pure, où le bruit est vraiment problématique.

Nos tests ont montré que l'analyse harmonique utilisant l'évolution artificielle pouvait à la fois :

1. fournir les informations de phase de manière assez précise, ce qui a aidé l'algorithme à trouver les structures isotopiques plus fines autour de chaque pic et
2. traite un rapport s/n plus faible que la FFT.

Notre premier objectif était donc d'obtenir des résultats de grande qualité sur les structures isotopiques grossières pour pouvoir déterminer les structures isotopiques fines en utilisant des échantillons plus petits par rapport à la FFT.

Afin de tester les performances de notre algorithme, nous avons d'abord utilisé des données simulées de Substance P dont les paramètres étaient déjà connus. Comme les signaux obtenus dans le monde réel sont toujours bruyants, nous nous sommes principalement concentrés sur les signaux bruyants et amortis. Le premier objectif était de déterminer les structures isotopiques grossières. Nos résultats ont montré qu'en utilisant une taille d'échantillon beaucoup plus petite, nous pouvons facilement déterminer les 6 pics principaux de la substance P par rapport à la méthode FFT qui nécessite une taille d'échantillon beaucoup plus grande pour détecter tous les pics principaux.

Une fois les pics grossiers déterminés, il est important de détecter les pics plus petits qui les entourent. C'était notre prochain objectif. Le résultat de la structure isotopique grossière nous a permis de réaliser une relation importante entre les phases et les fréquences. Les phases sont prédites pour avoir une dépendance quadratique des fréquences et c'est une dépendance linéaire sur une courte gamme avec une très forte corrélation. La connaissance de cette relation de dépendance a été très utile pour déterminer les structures isotopiques fines car nous avons initialisé les phases en utilisant l'équation linéaire obtenue.

Au final, notre algorithme ne nécessite "que" 6 heures environ pour déterminer la structure grossière (pics principaux du composé). Ensuite, il a fallu beaucoup plus de temps pour trouver les pics isotopiques fins :

- 2-3 jours pour la détermination isotopique fine du premier isotope
- 7 jours pour la détermination isotopique fine du second isotope
- 13 jours pour la détermination isotopique fine du troisième isotope

L'énorme temps de calcul pour le troisième isotope vient du fait que nous cherchions 18 pics mais malgré tout, ce temps de calcul énorme était plus petit que pour urQrd + FFT (état actuel de l'art) qui obtenait un résultat de moindre qualité sur les mêmes données.

Nos résultats étaient principalement basés sur la double précision. Nous avons utilisé la simple précision comme comparaison et avons remarqué que pour des échantillons plus grands, la précision diminue par rapport à la double précision. Pour améliorer la précision, nous avons inventé les radians binaires (BRADs). Nous pensions que l'utilisation de radians binaires donnerait une meilleure précision puisqu'elle n'implique pas de division par π dans la réduction de l'intervalle. Cependant, nos résultats ont montré que les BRAD en virgule

flottante avaient des performances similaires à celles de la précision simple, mais qu'ils étaient environ 1,5 à 1,6 fois plus rapides.

Pour résumer, les contributions clés suivantes ont été apportées dans ce projet de thèse :

- Une nouvelle méthode d'échantillonnage (GRS).
- Une nouvelle méthode de réduction de plage pour les fonctions périodiques (BRAD).
- Tester une nouvelle méthode basée sur le PSO quantique et le DMC (QAES).
- Utilisation d'une relation de dépendance linéaire entre les phases et les fréquences pour déterminer les structures isotopiques fines.
- Une nouvelle méthode de détermination des fonctions d'amortissement utilisant la programmation génétique
- Parallélisation sur les cartes GPGPU et utilisation d'un modèle d'îlot.

Enfin, l'étude sur des données simulées nous a permis de résoudre la structure isotopique grossière d'un signal expérimental provenant du MS FT-ICR. Nous avons montré que nous pouvons obtenir les 6 pics principaux de la substance expérimentale P en utilisant une taille d'échantillon beaucoup plus petite par rapport à la FFT.

En effet, même si les résultats présentés sont significativement meilleurs que la Transformée de Fourier, ils ont été obtenus avec un algorithme très simple (dont les paramètres ont été finement ajustés en utilisant des mois d'expérimentation sur un super ordinateur très puissant, la machine PARSEC.

Mais ce doctorat est le premier, à notre connaissance, à se concentrer sur l'application d'une approche évolutive à l'analyse harmonique. Par conséquent, de nombreuses améliorations n'ont pas pu être menées à bien, mais seulement brièvement expérimentées, montrant ainsi la possibilité d'amélioration.

Maintenant que la base a été établie, de nombreuses autres améliorations peuvent être testées, ouvrant la voie à différents projets de recherche futurs, dont certains que nous explorerons personnellement à l'avenir, grâce à ma nomination en tant que professeur associé à l'université UFAZ.

Appendix B

SINUS-IT code

The code below is provided for reproducibility of the results presented in section 9. It is the coarse version of the algorithm (the changes to create the fine version are described in section 9.2.2.

The code should be compiled with `EASENA version: 2.20 (RE)` to be found on `https://github.com/EASEA/easea`.

It should be compiled with the `--cuda` option in order to be run on any NVIDIA GPGPU card since the 8800GTX card (but of course, computing speed will differ depending on the card).

What is important in choosing a card is the number of GFlops it can produce in Floating Point precision (this information can be found on `https://en.wikipedia.org/wiki/List_of_Nvidia_graphics_processing_units`).

```
/*-----------------------------------------------------------
Ulviya Abdulkarimova and Marc HAEGELIN 26/04/2021
This file is to be compiled with EASENA version: 2.20 (RE)
Note that EASEA filenames cannot contain dashes "-" but they
can contain underscores "_" as is the case with sinus_it.ez
-----------------------------------------------------------*/

\User declarations :
#include "bradint.h"
#include "very_uniform.h"
#include "quicksort.h"
#include <sys/time.h>

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>

//These defines are mandatory (can't be variables) because used in easena cuda device
    code
//PI and 2*PI with accuracy
#define PI
    3.14159265358979323846264338327950288419716939937510582097494459230781640628620 9
#define PI2
    6.28318530717958647692528676655900576839433879875021164194988918461563281257241 8
//Conversion from brad to rad and vice versa
```

```
#define RAD2BRADMULT
     40.7436654315252059568342434233636766808216693095568508794028400790775801943619 93
#define BRAD2RADMULT
     0.0245436926061702596754894014318711162827903859326180142263667546274048156741110 08
//Define maximal number of sines
#define MAX_SIN 6

//Time at the beginning and the end of the GRS step
struct timeval tv_time_start;
struct timeval tv_time_stop;

//Number of sines to search for in the signal (up to MAX_SIN)
int nNB_SIN=6;
//Compute with brad_int function
bool bBRAD_INT=0;
//Flag to indicate if we search for fine isotopic distribution
bool bFINE_ISOTOPIC=0;
//Boolean flag to indicate if there is a decay or not
bool bNO_DECAY=0;

//Boundaries for amplitude, frequency, phase and exponential decay
double fMIN_AMP = 200.0;
double fMAX_AMP = 120000.0;
double fMIN_FREQ = 0.26;
double fMAX_FREQ = 0.27;
double fMIN_PH = 0.0;
double fMAX_PH = 6.283185308;
double fMIN_EXP = 7.0;
double fMAX_EXP = 11.5;

//Display value of the best individual every nDISPLAY_EVERY generations
int nDISPLAY_EVERY = 10;
//Size of the whole transient
int nTRANSIENT_SIZE = 16777216;
//NUS multiplier (if NUS acquisition : power of 2)
int nNUS_ACQ=1;

//Sampling mode (FULL, NUS, GRS, NUS FILE)
bool bSAMP_FULL = 1;
bool bSAMP_NUS = 0;
bool bSAMP_GRS_extension = 0;
bool bSAMP_GRS_replacement = 0;
bool bNUS_FILE = 0;

//GRS Settings
// e.g. with nNUS_GRS_SAMP==16 we pick up 128 points out of 2048(==128*16) sampling
     points from 32768(==2048*16) full transient with nNUS_GRS==16
//Number of generations elapsed before expanding the GRS points
int nNB_GRS_GENS=128;
//Ratio of acquired transient over sampled transient (power of 2)
int nNUS_GRS=1;
//Defines the number of points to introduce in GRS every nNB_GRS_GENS generations (power
     of 2)
//e.g. with nNUS_GRS_SAMP==1 we introduce nNB_SAMPLES new points at every GRS step
//e.g. with nNUS_GRS_SAMP==2 we introduce nNB_SAMPLES / 2 new points at every GRS step
//e.g. with nNUS_GRS_SAMP==4 we introduce nNB_SAMPLES / 4 new points at every GRS step
int nNUS_GRS_SAMP=4;//8;//4;
//Number of current generations to pass before changing the GRS points (counter
     initialization)
int nGENS_BEFORE_GRS_CHANGE=nNB_GRS_GENS;

//Window within we want to sample
//Start point of the window
int nX_MIN=0; //32768;
//Size of the sampling window (power of 2)
int nWIN_SIZE=2048; //32768;
//End point of the window
```

```
int nX_MAX=nX_MIN+nWIN_SIZE;

//Input transient file name
char cinput_signal_file[150]= "output_signal_Mon_May__3_14:56:58_2021.bin";
char cinput_ranks_file[150]="ranks_Mon_Feb_22_15-10-06_2021.bin";

//Counter initialization for display
int nUSERDISPLAY_COUNTER=nDISPLAY_EVERY;

//Computed settings
int nTRANSIENT_ACQUIRED=nTRANSIENT_SIZE/nNUS_ACQ;
int nNB_SAMPLES = nTRANSIENT_ACQUIRED/nNUS_GRS;

//Backup for the initial number of points
int nINITIAL_nNB_SAMPLES = nNB_SAMPLES;

//New points to introduce for each GRS step
int nNEW_POINTS_PER_GRS_STEP = nNB_SAMPLES / nNUS_GRS_SAMP;

//Step number in GRS
int nSTEP=1;
//Steps left in GRS
int nGRS_STEPS_LEFT;

double fintensity_delta_amp;
double fintensity_delta_freq;
double fintensity_delta_phase;
double fintensity_delta_decay;

double fmutator_amp_rate;
double fmutator_freq_rate;
double fmutator_ph_rate;
double fmutator_dec_rate;

//Maximum similarity between freq. of the sines before merging
double fepsilon_freq; //0.00000015;

//Backup for initial best fitness
double fbestinitialfitness;

//Current generation counter
int nCURRENT_GEN=0;

//Whole transient values
double* fTRANSIENT = (double*) malloc(nTRANSIENT_SIZE*sizeof(double));
//Acquired transient in Bruker format (int32 values)
int* nTRANSIENT;
//Coordinates of acquired points
int* nCOORDS_ACQUIRED = (int*) malloc(nTRANSIENT_ACQUIRED*sizeof(int));
//Consecutive X coordinates in sampled transient points
int *nCOORDS_ACQUIRED_NUS_GRS;

//Missing points in full transient coordinates (GRS sampling with FULL file)
vector<int> nMISSING_GRS_FULL_POINTS;
//Missing points in acquired transient coordinates (GRS sampling with NUS file)
vector<int> nMISSING_GRS_ACQUIRED_POINTS;

//Sampled points
double* fSAMPLE = (double*) malloc(sizeof(double) * 2 * nNB_SAMPLES);

//String to store useful informmations added at each GRS step (for final log file)
char* cGRS_STEPS = NULL;

//Random coordinates in GRS acquired transient for bNUS_FILE when subsampling
int* nRAND_GRS_COORDINATES;

#ifdef __CUDACC__ // if we compile with nvcc
```

```
//Define d_fgpuSample to be the device version for fSAMPLE
__device__ double* d_fgpuSample;

//Device variable for nNB_SAMPLES
__device__ int d_nNB_SAMPLES;

//Device variable for nNB_SIN
__device__ int d_nNB_SIN;

//Device variable for bBRAD_INT
__device__ bool d_bBRAD_INT;

//Device variable for nTRANSIENT_SIZE
__device__ int d_nTRANSIENT_SIZE;

#endif

\end

\User functions:

#ifdef __CUDACC__

//Computes the size of the grid for Npts threads to launch
__device__ __host__ dim3 compute_dimensions(cudaDeviceProp* properties,  long unsigned
    Npts){
       long long int d1=1;
       long long int d2=1;
       long long int d3=ceil(Npts*1.0/(properties->maxThreadsPerBlock));

       while(d3>properties->maxGridSize[2]){
                d2*=2;
                d3=ceil(d3*1.0/2);
       }

       while(d2>properties->maxGridSize[1]){
                d1*=2;
                d2=ceil(d2*1.0/2);
       }

       return dim3(d1, d2, d3);
}

//Kernel to initialize global device variable d_fgpuSample
__global__ void initd_fgpuSample(double* d_fgpuSampleData)
{
    d_fgpuSample = d_fgpuSampleData;
}

//Kernel to update global device variable d_fgpuSample
__global__ void updated_fgpuSample(double* d_fgpuSampleData, int size)
{
  int tid = blockIdx.x * blockDim.x * blockDim.y * blockDim.z+ threadIdx.z * blockDim.y *
       blockDim.x + threadIdx.y * blockDim.x + threadIdx.x;

  if(tid<size){
    d_fgpuSample[tid]=d_fgpuSampleData[tid];
  }
}

#endif


//Conversion function from rad to brad
__device__ __host__ inline double rad2brad(double fRad){
  return fRad*RAD2BRADMULT;
```

```
}

//Conversion function from brad to rad
__device__ __host__ inline double brad2rad(double fBrad){
  return fBrad*BRAD2RADMULT;
}


//Evaluation function
__device__ __host__ inline double fScoreOnGPU_L2(double genome[4*MAX_SIN]){

double y,fScore=0.0;

#ifndef __CUDA_ARCH__ // __host__ compiler part

for (int i=0;i<nNB_SAMPLES;i++){
  y=0; // writing the intialization of y here for clarity
  for(int j=0;j<nNB_SIN;j++)
    y+=exp((log(genome[4*j+3]/100.0)/nTRANSIENT_SIZE)*fSAMPLE[2*i])*genome[4*j+0]*sin(2*
        PI*genome[4*j+1]*fSAMPLE[2*i]+genome[4*j+2]);

fScore+=powf(fSAMPLE[2*i+1]-y,2);//powf(fSAMPLE[2*i+1]-((double)y),2); // square of the
    difference to focus on the large values
}

fScore/=(double)nNB_SAMPLES;

#else // __device__ compiler part

for(int i=0;i<d_nNB_SAMPLES;i++){
  y=0; // writing the intialization of y here for clarity
  for(int j=0;j<d_nNB_SIN;j++)
    y+=exp((log(genome[4*j+3]/100.0)/d_nTRANSIENT_SIZE)*d_fgpuSample[2*i])*genome[4*j+0]*
        sin(2*PI*genome[4*j+1]*d_fgpuSample[2*i]+genome[4*j+2]);

fScore+=powf(d_fgpuSample[2*i+1]-((double)y),2); // square of the difference to focus on
    the large values
}

fScore/=(double)d_nNB_SAMPLES;

#endif

fScore=powf(fScore, .5);

return fScore;
}


//Evaluation function with brad int
__device__ __host__ inline double fScoreOnGPUbradint(double genome[4*MAX_SIN]){

double y,fScore=0.0;

#ifndef __CUDA_ARCH__ // __host__ compiler part

for(int i=0;i<nNB_SAMPLES;i++){
  y=0.0; // writing the intialization of y here for clarity
  for(int j=0;j<nNB_SIN;j++){
    y+=exp((log(genome[4*j+3]/100.0)/nTRANSIENT_SIZE)*fSAMPLE[2*i])*genome[4*j+0]*sin(
        brad2rad(hbrad_modlli64d(2*PI*rad2brad(genome[4*j+1]),(unsigned long long int)
        fSAMPLE[2*i],rad2brad(genome[4*j+2]))));
  }
  fScore+=pow(fSAMPLE[2*i+1]-y,2); // square of the difference to focus on the large
      values
}
```

```
fScore/=(double)nNB_SAMPLES;

#else // __device__ compiler part

for(int i=0;i<d_nNB_SAMPLES;i++){
  y=0.0; // writing the intialization of y here for clarity
  for(int j=0;j<d_nNB_SIN;j++)
    y+=exp((log(genome[4*j+3]/100.0)/d_nTRANSIENT_SIZE)*d_fgpuSample[2*i])*genome[4*j+0]*
        sin(brad2rad(dbrad_modlli64d(2*PI*rad2brad(genome[4*j+1]),(unsigned long long
        int)d_fgpuSample[2*i],rad2brad(genome[4*j+2]))));
  fScore+=pow(d_fgpuSample[2*i+1]-y,2); // square of the difference to focus on the large
       values
}

fScore/=(double)d_nNB_SAMPLES;

#endif

fScore=pow(fScore, .5);
return fScore;

}


\end

\User CUDA:
\end

\Before everything else function:

if(!bNUS_FILE){
  //Update number of samples
  nNB_SAMPLES = nWIN_SIZE / nNUS_GRS;

  //Backup for the initial number of points
  nINITIAL_nNB_SAMPLES = nNB_SAMPLES;

  //New points to introduce for each GRS step
  nNEW_POINTS_PER_GRS_STEP = nNB_SAMPLES / nNUS_GRS_SAMP;

  free(fSAMPLE);

  fSAMPLE = (double*) malloc(sizeof(double) * 2 * nNB_SAMPLES);

}

printf("nX_MIN=%d\n", nX_MIN);
printf("nX_MAX=%d\n", nX_MAX);
printf("nNB_SAMPLES=%d\n", nNB_SAMPLES);
printf("nTRANSIENT_ACQUIRED=%d\n", nTRANSIENT_ACQUIRED);


if(nNB_SIN>MAX_SIN){
  printf("Error : Trying to search for nNB_SIN>%d sines (max. limit)\n", MAX_SIN);
  exit(1);
}

if(bSAMP_FULL==0 && bSAMP_NUS==0 && bSAMP_GRS_extension==0 && bSAMP_GRS_replacement==0){
    printf("Error : At least one of the four (bSAMP_FULL, bSAMP_NUS, bSAMP_GRS_extension,
         bSAMP_GRS_replacement) must be non-zero\n");
    exit(1);
}

if((bSAMP_FULL&&bSAMP_NUS) || (bSAMP_FULL&&bSAMP_GRS_extension) || (bSAMP_NUS&&
    bSAMP_GRS_extension)
    || (bSAMP_FULL&&bSAMP_GRS_replacement) || (bSAMP_NUS&&bSAMP_GRS_replacement)){
```

```
    printf("Error : Only one of the four (bSAMP_FULL, bSAMP_NUS, bSAMP_GRS_extension,
        bSAMP_GRS_replacement) must be non-zero\n");
    exit(1);
}

//Make sure that nNB_SAMPLES<nTRANSIENT_ACQUIRED
if(nNB_SAMPLES>nTRANSIENT_ACQUIRED){
    printf("Error : trying to subsample too many points\n");
    printf("Please make sure that nNB_SAMPLES<=nTRANSIENT_ACQUIRED\n");
    exit(1);
}

//Make sure there are enough points within [nX_MIN, nX_MAX]
if(nNB_SAMPLES>(nX_MAX-nX_MIN)){
    printf("Error : The [nX_MIN, nX_MAX] interval has not enough points for sampling\n");
    printf("Please make sure that nNB_SAMPLES<=(nX_MAX-nX_MIN)\n");
    exit(1);
}

//Make sure that nX_MIN and nX_MAX are in [0, nTRANSIENT_ACQUIRED]
if(nX_MAX>nTRANSIENT_SIZE || nX_MIN>nTRANSIENT_SIZE || nX_MAX<0 || nX_MIN<0){
    printf("Error : trying to window outside of the acquired transient\n");
    printf("Please make sure that nX_MIN and nX_MAX are in [0, nTRANSIENT_ACQUIRED]");
    exit(1);
}

//Make sure that nX_MAX > nX_MIN
if(nX_MAX<=nX_MIN){
    printf("Error : nX_MAX <= nX_MIN\n");
    printf("Please make sure to window in a correct manner\n");
    exit(1);
}

if(log2((double)(nX_MAX-nX_MIN))!=(int)log2((double)(nX_MAX-nX_MIN))){
    printf("Error : The number of points to be considered in the signal is not a power of
         2\n");
    exit(1);
}

if(log2((double)nNB_SAMPLES)!=(int)log2((double)nNB_SAMPLES)){
    printf("Error : The number of points to sample from the signal is not a power of 2\n"
        );
    exit(1);
}

if(log2((double)nNUS_GRS_SAMP)!=(int)log2((double)nNUS_GRS_SAMP)){
    printf("Error : nNUS_GRS_SAMP is not a power of 2\n");
    exit(1);
}

if(nNUS_GRS_SAMP>nNB_SAMPLES){
    printf("Error : nNUS_GRS_SAMP must be lower than nNB_SAMPLES (too big)\n");
    exit(1);
}

if(log2((double)nTRANSIENT_ACQUIRED)!=(int)log2((double)nTRANSIENT_ACQUIRED)){
    printf("Error : nTRANSIENT_ACQUIRED is not a power of 2\n");
    exit(1);
}

if(log2((double)nWIN_SIZE)!=(int)log2((double)nWIN_SIZE)){
    printf("Error : nWIN_SIZE is not a power of 2\n");
    exit(1);
}

if(bFINE_ISOTOPIC){
```

```
  fintensity_delta_amp = (fMAX_AMP-fMIN_AMP)/(2.0*128);
  fintensity_delta_freq = (fMAX_FREQ-fMIN_FREQ)/(2.0*16);
  fintensity_delta_phase = (fMAX_PH-fMIN_PH)/(2.0*8);
  fintensity_delta_decay = (fMAX_EXP-fMIN_EXP)/(2.0*16);

  fmutator_amp_rate = 0.8;
  fmutator_freq_rate = 0.6;
  fmutator_ph_rate = 0.4;
  fmutator_dec_rate = 0.233;

  //Maximum similarity between freq. of the sines before merging
  fepsilon_freq = 0.00000015;


}else{

  fintensity_delta_amp = (fMAX_AMP-fMIN_AMP)/(2.0*128);
  fintensity_delta_freq = (fMAX_FREQ-fMIN_FREQ)/(2.0*4);
  fintensity_delta_phase = (fMAX_PH-fMIN_PH)/(2.0*8);
  fintensity_delta_decay = (fMAX_EXP-fMIN_EXP)/(2.0*16);

  fmutator_amp_rate = 1.0;
  fmutator_freq_rate = 1.0;
  fmutator_ph_rate = 0.5;
  fmutator_dec_rate = 0.233;

  //Maximum similarity between freq. and phase of the sines before merging
  fepsilon_freq = 0.00015;

}


FILE* file = fopen(cinput_signal_file, "r");

if(file==NULL){
  printf("Error : Could not read signal data file\n");
  exit(1);
}


if(bNUS_FILE && bSAMP_GRS_extension){

    //Count the number of times left we can extend the points number in GRS Sampling (
        initialization)
    nGRS_STEPS_LEFT=(nNUS_GRS-1)*nNUS_GRS_SAMP;
    //Consecutive X coordinates in nGRS_STEPS_LEFT(=(nNUS_GRS-1)*nNUS_GRS_SAMP) GRS draws
    nCOORDS_ACQUIRED_NUS_GRS = (int*) malloc(nNB_SAMPLES*sizeof(double));

    //Malloc array to store the input Bruker (int32) signal
    nTRANSIENT = (int*) malloc(nTRANSIENT_ACQUIRED*sizeof(int));
    //Read transient from file un Bruker format (int32)
    fread(nTRANSIENT,sizeof(int),nTRANSIENT_ACQUIRED,file);

    FILE* rank_file = fopen(cinput_ranks_file, "r");

    if(rank_file!=NULL){
        // Now, read the coordinates from rank file in NUS acquisition
        fread(nCOORDS_ACQUIRED, sizeof(int), nTRANSIENT_ACQUIRED, rank_file);
    }else{
        printf("Error : NUS input file selected and ranks.bin was not found\n");
        exit(1);
    }

    //Choose random points from acquired transient
    nRAND_GRS_COORDINATES = (int*) malloc(nNB_SAMPLES*sizeof(int));
    very_uniform_grs(nNB_SAMPLES, nNUS_GRS, nRAND_GRS_COORDINATES);
```

```
    for(int i=0; i<nNB_SAMPLES; i++){
        nCOORDS_ACQUIRED_NUS_GRS[i]=nCOORDS_ACQUIRED[nRAND_GRS_COORDINATES[i]-1]-1;
    }

    //Enumerate and store points coordinates that are missing from previous draw in
         acquired transient
    int k=0; // initialize counter

    for(int i=0; i<nNB_SAMPLES; i++){ // For each existing point in acquired transient
      if(nRAND_GRS_COORDINATES[k]-1!=i){ // If i-th acquired point is not a point
           selected in the current sorted GRS coordinates
        nMISSING_GRS_ACQUIRED_POINTS.push_back(i); // Add i-th acquired point to the
             missing coordinates
      }else{ // else if i-th is already in the GRS acquired points coordinates
        k++; // increment counter
      }
    }

    for(int i=0; i<nNB_SAMPLES; i++){
      fSAMPLE[2*i]=(double)nCOORDS_ACQUIRED_NUS_GRS[i];
      fSAMPLE[2*i+1]=nTRANSIENT[nRAND_GRS_COORDINATES[i]-1];
    }

}else if(bNUS_FILE && bSAMP_GRS_replacement){

    //Count the number of distinct draws we still can use in GRS sampling (initialization
         )
    nGRS_STEPS_LEFT=nNUS_GRS*nNUS_GRS;
    //Consecutive draws of X coordinates in nGRS_STEPS_LEFT(=nNUS_GRS*nNUS) + 1_GRS draws
    nCOORDS_ACQUIRED_NUS_GRS = (int*) malloc((nGRS_STEPS_LEFT+1)*nNB_SAMPLES*sizeof(
         double));

    //Malloc array to store the input Bruker (int32) signal
    nTRANSIENT = (int*) malloc(nTRANSIENT_ACQUIRED*sizeof(int));
    //Read transient from file un Bruker format (int32)
    fread(nTRANSIENT,sizeof(int),nTRANSIENT_ACQUIRED,file);

    FILE* rank_file = fopen(cinput_ranks_file, "r");

    if(rank_file!=NULL){
        // Now, read the coordinates from rank file in NUS acquisition
        fread(nCOORDS_ACQUIRED, sizeof(int), nTRANSIENT_ACQUIRED, rank_file);
    }else{
        printf("Error : NUS input file selected and ranks.bin was not found\n");
        exit(1);
    }

    //Choose random points from acquired transient
    nRAND_GRS_COORDINATES = (int*) malloc(nNB_SAMPLES*sizeof(int));
    very_uniform_grs(nNB_SAMPLES, nNUS_GRS, nRAND_GRS_COORDINATES);

    for(int i=0; i<nNB_SAMPLES; i++){
        nCOORDS_ACQUIRED_NUS_GRS[i]=nCOORDS_ACQUIRED[nRAND_GRS_COORDINATES[i]-1]-1;
    }

    for(int i=0; i<nNB_SAMPLES; i++){
      fSAMPLE[2*i]=(double)nCOORDS_ACQUIRED_NUS_GRS[i];
      fSAMPLE[2*i+1]=nTRANSIENT[nRAND_GRS_COORDINATES[i]-1];
    }

}else if(bNUS_FILE && bSAMP_FULL){

    if(nTRANSIENT_ACQUIRED!=nNB_SAMPLES){
      printf("Error : Make sure that nNB_SAMPLES==nTRANSIENT_ACQUIRED in bNUS_FILE &&
           bSAMP_FULL mode\n");
      exit(1);
    }
```

```
    nTRANSIENT = (int*) malloc(nTRANSIENT_ACQUIRED*sizeof(int));
    fread(nTRANSIENT,sizeof(int),nTRANSIENT_ACQUIRED,file);

    FILE* rank_file = fopen(cinput_ranks_file, "r");

    if(rank_file!=NULL){
        // Now, we create nTRANSIENT_ACQUIRED points within [X_MIN, X_MAX] in NUS
            acquisition
        fread(nCOORDS_ACQUIRED, sizeof(int), nTRANSIENT_ACQUIRED, rank_file);
    }else{
        printf("Error : NUS sampling selected and ranks.bin was not found\n");
        exit(1);
    }

    for(int i=0; i<nNB_SAMPLES; i++){
        fSAMPLE[2*i]=(double)nCOORDS_ACQUIRED[i]-1;
        fSAMPLE[2*i+1]=nTRANSIENT[i];
    }

}else if(bNUS_FILE && bSAMP_NUS){

    nRAND_GRS_COORDINATES = (int*) malloc(nNB_SAMPLES*sizeof(double));

    nTRANSIENT = (int*) malloc(nTRANSIENT_ACQUIRED*sizeof(int));
    fread(nTRANSIENT,sizeof(int),nTRANSIENT_ACQUIRED,file);

    FILE* rank_file = fopen(cinput_ranks_file, "r");

    if(rank_file!=NULL){
        // Now, we create nTRANSIENT_ACQUIRED points within [X_MIN, X_MAX] in NUS
            acquisition
        fread(nCOORDS_ACQUIRED, sizeof(int), nTRANSIENT_ACQUIRED, rank_file);
    }else{
        printf("Error : NUS sampling selected and ranks.bin was not found\n");
        exit(1);
    }

    //Pick up randomly the NUS coordinate points
    very_uniform_nus(nNB_SAMPLES, nNUS_GRS, nRAND_GRS_COORDINATES);

    for(int i=0; i<nNB_SAMPLES; i++){
        fSAMPLE[2*i]=(double)nCOORDS_ACQUIRED[nRAND_GRS_COORDINATES[i]-1]-1;
        fSAMPLE[2*i+1]=nTRANSIENT[nRAND_GRS_COORDINATES[i]-1];
    }

}else if(!bNUS_FILE && bSAMP_NUS){ // FULL file and NUS sampling mode

    //Malloc array to store the input bruker signal
    nTRANSIENT = (int*) malloc(nTRANSIENT_SIZE*sizeof(int));
    //Read transient from file un Bruker format (int32)
    fread(nTRANSIENT,sizeof(int),nTRANSIENT_SIZE,file);

    //Pick up randomly the NUS coordinate points
    very_uniform_nus(nNB_SAMPLES, nNUS_GRS, nCOORDS_ACQUIRED);

    for(int i=0; i<nNB_SAMPLES; i++){
        nCOORDS_ACQUIRED[i]+=(nX_MIN-1); // shift from nX_MIN to be within [nX_MIN,nX_MAX
            ]
    }

    //Initialize fSAMPLE coordinates and points
    for(int i=0; i<nNB_SAMPLES; i++){
        fSAMPLE[2*i]=(double)nCOORDS_ACQUIRED[i];
        fSAMPLE[2*i+1]=nTRANSIENT[nCOORDS_ACQUIRED[i]];
    }
```

```
}else if(!bNUS_FILE && bSAMP_GRS_extension){

    //Count the number of times left we can extend the points number in GRS Sampling (
        initialization)
    nGRS_STEPS_LEFT=(nNUS_GRS-1)*nNUS_GRS_SAMP;
    //Consecutive X coordinates in nGRS_STEPS_LEFT(=(nNUS_GRS-1)*nNUS_GRS_SAMP) GRS draws
    nCOORDS_ACQUIRED_NUS_GRS = (int*) malloc(nNB_SAMPLES*sizeof(double));

    //Malloc array to store the input bruker signal
    nTRANSIENT = (int*) malloc(nTRANSIENT_SIZE*sizeof(int));
    //Read transient from file un Bruker format (int32)
    fread(nTRANSIENT,sizeof(int),nTRANSIENT_SIZE,file);

    //Pick up randomly the GRS coordinate points
    very_uniform_grs(nNB_SAMPLES, nNUS_GRS, nCOORDS_ACQUIRED_NUS_GRS);

    //Enumerate and store points coordinates that are missing from previous draw
    int k=0; // initialize counter

    for(int i=0; i<nTRANSIENT_ACQUIRED; i++){ // For each existing point in acquired
        transient
      if(nCOORDS_ACQUIRED_NUS_GRS[k]!=i){ // If i is not a point selected in the current
          sorted GRS coordinates
        nMISSING_GRS_FULL_POINTS.push_back(i); // Add i to the missing coordinates
      }else{ // else if i is already in the GRS coordinates
        k++; // increment counter
      }
    }

    for(int i=0; i<nNB_SAMPLES; i++){
        nCOORDS_ACQUIRED_NUS_GRS[i]+=(nX_MIN-1); // shift from nX_MIN to be within [
            nX_MIN,nX_MAX]
    }

    //Initialize fSAMPLE coordinates and points
    for(int i=0; i<nNB_SAMPLES; i++){
        fSAMPLE[2*i]=(double)nCOORDS_ACQUIRED_NUS_GRS[i];
        fSAMPLE[2*i+1]=nTRANSIENT[nCOORDS_ACQUIRED_NUS_GRS[i]];
    }

}else if(!bNUS_FILE && bSAMP_GRS_replacement){

    //Count the number of distinct draws we still can use in GRS sampling (initialization
        )
    nGRS_STEPS_LEFT=nNUS_GRS*nNUS_GRS;
    //Consecutive draws of X coordinates in nGRS_STEPS_LEFT(=nNUS_GRS*nNUS) + 1_GRS draws
    nCOORDS_ACQUIRED_NUS_GRS = (int*) malloc((nGRS_STEPS_LEFT+1)*nNB_SAMPLES*sizeof(
        double));

    nTRANSIENT = (int*) malloc(nTRANSIENT_SIZE*sizeof(int));
    fread(nTRANSIENT,sizeof(int),nTRANSIENT_SIZE,file);

    for(int i=0; i<=nGRS_STEPS_LEFT; i++){
        very_uniform_grs(nNB_SAMPLES, nNUS_GRS, nCOORDS_ACQUIRED_NUS_GRS+i*nNB_SAMPLES);
    }

    for(int i=0; i<nNB_SAMPLES*nGRS_STEPS_LEFT; i++){
        nCOORDS_ACQUIRED_NUS_GRS[i]+=(nX_MIN-1); // shift from nX_MIN to be within [
            nX_MIN,nX_MAX]
    }

    for(int i=0; i<nNB_SAMPLES; i++){
        fSAMPLE[2*i]=(double)nCOORDS_ACQUIRED_NUS_GRS[i];
        fSAMPLE[2*i+1]=nTRANSIENT[nCOORDS_ACQUIRED_NUS_GRS[i]];
    }

}else if(!bNUS_FILE && bSAMP_FULL){ // FULL file and FULL sampling mode
```

```
    if(nNB_SAMPLES!=(nX_MAX-nX_MIN)){
      printf("Error : Window size must equal the number of sampling points\n");
      exit(1);
    }

    //Malloc array to store the input bruker signal
    nTRANSIENT = (int*) malloc(nTRANSIENT_SIZE*sizeof(int));
    //Read transient from file un Bruker format (int32)
    fread(nTRANSIENT,sizeof(int),nTRANSIENT_SIZE,file);

    if(nX_MIN+nNB_SAMPLES>nTRANSIENT_ACQUIRED){
        printf("Error : nX_MIN+nNB_SAMPLES>nTRANSIENT_ACQUIRED\n");
        exit(1);
    }

    for(int i=0; i<nTRANSIENT_ACQUIRED; i++){
        nCOORDS_ACQUIRED[i]=nX_MIN+i; // points are consecutive
    }

    //Initialize fSAMPLE coordinates and points
    for(int i=0; i<nNB_SAMPLES; i++){
        fSAMPLE[2*i]=(double)nCOORDS_ACQUIRED[i];
        fSAMPLE[2*i+1]=nTRANSIENT[nCOORDS_ACQUIRED[i]];
    }

}else{ // Flags are not correctly set
    printf("Error : Incorrect sampling flags settings\n");
    printf("Exiting\n");
    exit(1);
}

#ifdef __CUDACC__ // if compiled with nvcc

//Transfer fSAMPLE to d_fgpuSample on the device
double* d_fSAMPLE;

CUDA_SAFE_CALL(cudaMalloc((void**)&d_fSAMPLE, sizeof(double) * 2 * nNB_SAMPLES));

CUDA_SAFE_CALL(cudaMemcpy(d_fSAMPLE, fSAMPLE, nNB_SAMPLES * 2 * sizeof(double),
     cudaMemcpyHostToDevice));

initd_fgpuSample<<<1,1>>>(d_fSAMPLE);

CUDA_SAFE_CALL(cudaDeviceSynchronize());

//Transfer value of nNB_SAMPLES to d_nNB_SAMPLES
CUDA_SAFE_CALL(cudaMemcpyToSymbol(d_nNB_SAMPLES, &nNB_SAMPLES, sizeof(int)));

//Transfer value of nNB_SIN to d_nNB_SIN
CUDA_SAFE_CALL(cudaMemcpyToSymbol(d_nNB_SIN, &nNB_SIN, sizeof(int)));

//Transfer value of bBRAD_INT to d_bBRAD_INT
CUDA_SAFE_CALL(cudaMemcpyToSymbol(d_bBRAD_INT, &bBRAD_INT, sizeof(bool)));

//Transfer value of nTRANSIENT_SIZE to d_nTRANSIENT_SIZE
CUDA_SAFE_CALL(cudaMemcpyToSymbol(d_nTRANSIENT_SIZE, &nTRANSIENT_SIZE, sizeof(int)));

#endif

gettimeofday(&tv_time_start, NULL);

\end

\After everything else function:

//Experiment informations
```

```
printf("-----------------------------------------------------------\n");
printf("nNB_SAMPLES:%d,nX_MIN:%d,nX_MAX:%d\n",nNB_SAMPLES,nX_MIN,nX_MAX);

//Sort the values by frequency
quicksort_freq(bBest->Sin, 0, nNB_SIN-1);

//Display the result to inform the user
printf("\n-----------------------------------------------------------\n");
printf("Obtained function sorted by frequency: \ny=exp(-%.15f*x)*%.15f*sin(%.15f*x+%.15f)
     ",-log(bBest->Sin[3]/100.0)/nTRANSIENT_SIZE,bBest->Sin[0],bBest->Sin[1],bBest->Sin
     [2]);
for (int i=1;i<nNB_SIN;i++)
  printf("+exp(-%.15f*x)*%.15f*sin(%.15f*x+%.15f)",-log(bBest->Sin[i*4+3]/100.0)/
       nTRANSIENT_SIZE,bBest->Sin[i*4+0],bBest->Sin[i*4+1],bBest->Sin[i*4+2]);
printf("\n-----------------------------------------------------------\n");

char input_file_names[500]="cinput_signal_file;";
strcat(input_file_names, cinput_signal_file);

if(bNUS_FILE){;
strcat(input_file_names, "\nInput ranks file;");
strcat(input_file_names, cinput_ranks_file);
}

if(bSAMP_GRS_extension){
logg("___GRS STEPS (by Extension)___");
logg((cGRS_STEPS==NULL ? "(None)" : cGRS_STEPS));
}else if(bSAMP_GRS_replacement){
logg("___GRS STEPS (by Replacement)___");
logg((cGRS_STEPS==NULL ? "(None)" : cGRS_STEPS));
}

logg("__RUN SETTINGS__");
logg("nCURRENT_GEN;", nCURRENT_GEN);

logg("___TRANSIENT SETTINGS___");
logg(input_file_names);
logg("nTRANSIENT_SIZE;", nTRANSIENT_SIZE);
logg("bNUS_FILE;", bNUS_FILE);
logg("nNUS_ACQ;", nNUS_ACQ);
logg("nTRANSIENT_ACQUIRED (=nTRANSIENT_SIZE/nNUS_ACQ);", nTRANSIENT_ACQUIRED);

logg("__PART OF TRANSIENT USED__");
logg("nX_MIN;", nX_MIN);
logg("nWIN_SIZE;", nWIN_SIZE);
logg("nX_MAX (=nX_MIN+nWIN_SIZE);", nX_MAX);

logg("__TREATMENT SETTINGS__");
logg("bNO_DECAY;", bNO_DECAY);
logg("bBRAD_INT;", bBRAD_INT);
logg("bSAMP_FULL;", bSAMP_FULL);
logg("bSAMP_NUS;", bSAMP_NUS);
logg("bSAMP_GRS_extension;", bSAMP_GRS_extension);
logg("bSAMP_GRS_replacement;", bSAMP_GRS_replacement);

logg("__NUS/GRS SETTINGS__");
logg("nNUS_GRS;", nNUS_GRS);
logg("nNB_SAMPLES (=nTRANSIENT_ACQUIRED/nNUS_GRS);", nINITIAL_nNB_SAMPLES);
logg("nNUS_GRS_SAMP;", nNUS_GRS_SAMP);
logg("nNB_GRS_GENS;", nNB_GRS_GENS);

logg("__MUTATOR__");
logg("epsilon_freq;", fepsilon_freq);
logg("fintensity_delta_amp;", fintensity_delta_amp);
logg("fintensity_delta_freq;", fintensity_delta_freq);
logg("fintensity_delta_ph;", fintensity_delta_phase);
logg("fintensity_delta_dec;", fintensity_delta_decay);
```

```
logg("fmutator_amp_rate;", fmutator_amp_rate);
logg("fmutator_freq_rate;", fmutator_freq_rate);
logg("fmutator_ph_rate;", fmutator_ph_rate);
logg("fmutator_dec_rate;", fmutator_dec_rate);

logg("__SINE SETTINGS__");
logg("nNB_SIN;", nNB_SIN);
logg("fMIN_AMP;", fMIN_AMP);
logg("fMAX_AMP;", fMAX_AMP);
logg("fMIN_FREQ;", fMIN_FREQ);
logg("fMAX_FREQ;", fMAX_FREQ);
logg("fMIN_PH;", fMIN_PH);
logg("fMAX_PH;", fMAX_PH);
logg("fMIN_EXP;", fMIN_EXP);
logg("fMAX_EXP;", fMAX_EXP);
logg("\n");


logg("__RESULTS__");
char columns_set[500]="Sine number;Amplitudes;Frequencies;Phases;Decays";

logg(columns_set);

for(int i=0; i<nNB_SIN; i++){

  char sine_set[125];
  snprintf(sine_set, 125, "Sine %d", i+1);

  char amp[25];
  char freq[25];
  char phase[25];
  char decay[25];

  snprintf(amp, 25, ";%.16f", bBest->Sin[4*i+0]);
  snprintf(freq, 25, ";%.16f", bBest->Sin[4*i+1]);
  snprintf(phase, 25, ";%.16f", bBest->Sin[4*i+2]);
  snprintf(decay, 25, ";%.16f", -log(bBest->Sin[4*i+3]/100.0)/nTRANSIENT_SIZE);

  strcat(sine_set, amp);
  strcat(sine_set, freq);
  strcat(sine_set, phase);
  strcat(sine_set, decay);

  logg(sine_set);

}

//free data
free(fSAMPLE);
free(nCOORDS_ACQUIRED);
free(nCOORDS_ACQUIRED_NUS_GRS);
free(nTRANSIENT);
free(fTRANSIENT);
free(cGRS_STEPS);
free(nRAND_GRS_COORDINATES);

\end


\At the beginning of each generation function:
// Global Random Sampling (GRS) section
// if GRS, if we have elapsed all generations before change, if we still have points to
    add
if(bSAMP_GRS_extension && !nGENS_BEFORE_GRS_CHANGE && nGRS_STEPS_LEFT){

    if(bNUS_FILE){
```

```
        //Backup for the number of sampling points
        int nOld_NB_SAMPLES=nNB_SAMPLES;

        //Update the number of sampling points by nNEW_POINTS_PER_GRS_STEP
        if(nNB_SAMPLES<nTRANSIENT_ACQUIRED){
          nNB_SAMPLES+=nNEW_POINTS_PER_GRS_STEP;
        }

        //Extend the coordinates and fSAMPLE by nNEW_POINTS_PER_GRS_STEP
        nCOORDS_ACQUIRED_NUS_GRS = (int*) realloc(nCOORDS_ACQUIRED_NUS_GRS, nNB_SAMPLES*
            sizeof(int));
        nRAND_GRS_COORDINATES = (int*) realloc(nRAND_GRS_COORDINATES, nNB_SAMPLES*sizeof(
            int));
        fSAMPLE = (double*) realloc(fSAMPLE, 2*nNB_SAMPLES*sizeof(double));

        //Initialize the new points coordinates picked up from missing points
        for(int i=0; i<nNEW_POINTS_PER_GRS_STEP; i++){
          int index = random((int)0, (int)nMISSING_GRS_ACQUIRED_POINTS.size());
          nCOORDS_ACQUIRED_NUS_GRS[i+nOld_NB_SAMPLES]=nCOORDS_ACQUIRED[
              nMISSING_GRS_ACQUIRED_POINTS.at(index)]-1;
          nRAND_GRS_COORDINATES[i+nOld_NB_SAMPLES]=nMISSING_GRS_ACQUIRED_POINTS.at(index)
              ;
          nMISSING_GRS_ACQUIRED_POINTS.erase(nMISSING_GRS_ACQUIRED_POINTS.begin()+index);
        }

        //Initialize new range points
        for(int i=0; i<nNEW_POINTS_PER_GRS_STEP; i++){
            fSAMPLE[2*(i+nOld_NB_SAMPLES)]=(double)nCOORDS_ACQUIRED_NUS_GRS[i+
                nOld_NB_SAMPLES];
            fSAMPLE[2*(i+nOld_NB_SAMPLES)+1]=nTRANSIENT[nRAND_GRS_COORDINATES[i+
                nOld_NB_SAMPLES]]; /// Use coordinates in acquired transient not in full
                 transient
        }

    }else{

        //Backup for the number of sampling points
        int nOld_NB_SAMPLES=nNB_SAMPLES;

        //Update the number of sampling points by nNEW_POINTS_PER_GRS_STEP
        if(nNB_SAMPLES<nTRANSIENT_ACQUIRED){
          nNB_SAMPLES+=nNEW_POINTS_PER_GRS_STEP;
        }

        //Extend the coordinates and fSAMPLE by nNEW_POINTS_PER_GRS_STEP
        nCOORDS_ACQUIRED_NUS_GRS = (int*) realloc(nCOORDS_ACQUIRED_NUS_GRS, nNB_SAMPLES*
            sizeof(int));
        fSAMPLE = (double*) realloc(fSAMPLE, 2*nNB_SAMPLES*sizeof(double));

        //Initialize the new points coordinates picked up from missing points
        for(int i=0; i<nNEW_POINTS_PER_GRS_STEP; i++){
          int index = random((int)0, (int)nMISSING_GRS_FULL_POINTS.size());
          nCOORDS_ACQUIRED_NUS_GRS[i+nOld_NB_SAMPLES]=nMISSING_GRS_FULL_POINTS.at(index);
          nMISSING_GRS_FULL_POINTS.erase(nMISSING_GRS_FULL_POINTS.begin()+index);
        }

        //Shift from nX_MIN to be within [nX_MIN,nX_MAX]
        for(int i=0; i<nNEW_POINTS_PER_GRS_STEP; i++){
          nCOORDS_ACQUIRED_NUS_GRS[i+nOld_NB_SAMPLES]+=nX_MIN;
        }

        //Initialize fSAMPLE new points
        for(int i=0; i<nNEW_POINTS_PER_GRS_STEP; i++){
            fSAMPLE[2*(i+nOld_NB_SAMPLES)]=(double)nCOORDS_ACQUIRED_NUS_GRS[i+
                nOld_NB_SAMPLES];
            fSAMPLE[2*(i+nOld_NB_SAMPLES)+1]=nTRANSIENT[nCOORDS_ACQUIRED_NUS_GRS[i+
                nOld_NB_SAMPLES]];
```

```
        }
    }

    gettimeofday(&tv_time_stop, NULL);

    //Decrement counter to enumerate GRS steps number
    nGRS_STEPS_LEFT--;

    //Building an information string for the output log
    char cCHAR_CONTAINER[500]="";
    char cCHAR_CONTAINER_STEP[125]="";

    cGRS_STEPS = (char*) realloc(cGRS_STEPS, sizeof(char)*nSTEP*500);

    snprintf(cCHAR_CONTAINER_STEP, 125, "GRS STEP / TOTAL STEPS = %d / %d\n", nSTEP, (
        nNUS_GRS-1)*nNUS_GRS_SAMP);
    strcat(cCHAR_CONTAINER, cCHAR_CONTAINER_STEP);
    snprintf(cCHAR_CONTAINER_STEP, 125, "STEP DURATION = %.2f sec.\n", (tv_time_stop.
        tv_sec - tv_time_start.tv_sec) + 1e-6*(tv_time_stop.tv_usec - tv_time_start.
        tv_usec));
    strcat(cCHAR_CONTAINER, cCHAR_CONTAINER_STEP);
    snprintf(cCHAR_CONTAINER_STEP, 125, "GENERATION NUMBER = %d\n", nCURRENT_GEN);
    strcat(cCHAR_CONTAINER, cCHAR_CONTAINER_STEP);
    snprintf(cCHAR_CONTAINER_STEP, 125, "CURRENT FITNESS = %f\n", bBest->fitness);
    strcat(cCHAR_CONTAINER, cCHAR_CONTAINER_STEP);
    snprintf(cCHAR_CONTAINER_STEP, 125, "NUMBER OF POINTS = %d\n\n", nNB_SAMPLES);
    strcat(cCHAR_CONTAINER, cCHAR_CONTAINER_STEP);

    strcat(cGRS_STEPS, cCHAR_CONTAINER);

    //Initialize again the counter to the number of generations before GRS
    nGENS_BEFORE_GRS_CHANGE=nNB_GRS_GENS;

    //Put stop timeval into start timeval
    tv_time_start = tv_time_stop;

    //Current GRS step
    nSTEP++;

    #ifdef __CUDACC__ // GPU operations

    //Declare, malloc and initialize d_fSAMPLE to contain the new fSAMPLE on the GPU side
    double* d_fSAMPLE;
    CUDA_SAFE_CALL(cudaMalloc((void**)&d_fSAMPLE, sizeof(double)*2*nNB_SAMPLES));

    CUDA_SAFE_CALL(cudaMemcpy(d_fSAMPLE, fSAMPLE, sizeof(double)*2*nNB_SAMPLES,
        cudaMemcpyHostToDevice));

    //Transfer new value of nNB_SAMPLES to d_nNB_SAMPLES to update
    CUDA_SAFE_CALL(cudaMemcpyToSymbol(d_nNB_SAMPLES, &nNB_SAMPLES, sizeof(int)));

    //Free old d_fgpuSample on the card
    double* GPU_SAMP;

    cudaGetSymbolAddress((void**)&GPU_SAMP, &d_fgpuSample);

    cudaFree(GPU_SAMP);

    cudaDeviceSynchronize();

    //Initialize the new d_fgpuSample
    initd_fgpuSample<<<1,1>>>(d_fSAMPLE);

    #endif
}else if(bSAMP_GRS_replacement && !nGENS_BEFORE_GRS_CHANGE){

      if(bNUS_FILE){
```

```
    for(int i=0; i<nNEW_POINTS_PER_GRS_STEP; i++){
        int randval = random((int)0, (int)nTRANSIENT_ACQUIRED);
        int xcoord = random((int)0, (int)nNB_SAMPLES);
        fSAMPLE[2*xcoord]=(double)nCOORDS_ACQUIRED[randval]-1;
        fSAMPLE[2*xcoord+1]=nTRANSIENT[randval];
    }
}else{

    for(int i=0; i<nNEW_POINTS_PER_GRS_STEP; i++){
       int randval = random((int)0, (int)nNB_SAMPLES);
       nCOORDS_ACQUIRED_NUS_GRS[randval]=nCOORDS_ACQUIRED_NUS_GRS[randval+nSTEP*
           nNB_SAMPLES];
    }

    for(int i=0; i<nNB_SAMPLES; i++){
        fSAMPLE[2*i]=(double)nCOORDS_ACQUIRED_NUS_GRS[i];
        fSAMPLE[2*i+1]=nTRANSIENT[nCOORDS_ACQUIRED_NUS_GRS[i]];
    }
}

gettimeofday(&tv_time_stop, NULL);


//Decrement counter to enumerate GRS steps number
nGRS_STEPS_LEFT--;
//Skip first draw and make a modulo to round-robin
nGRS_STEPS_LEFT = (nGRS_STEPS_LEFT==0 ? (nNUS_GRS*nNUS_GRS) : nGRS_STEPS_LEFT);

//Initialize again the counter to the number of generations before GRS
nGENS_BEFORE_GRS_CHANGE=nNB_GRS_GENS;


//Building an information string for the output log
char cCHAR_CONTAINER[625]="";
char cCHAR_CONTAINER_STEP[125]="";

cGRS_STEPS = (char*) realloc(cGRS_STEPS, sizeof(char)*nSTEP*500);

snprintf(cCHAR_CONTAINER_STEP, 125, "GRS STEP / GRS DRAWS = %d / %d\n", nSTEP,
    nNUS_GRS*nNUS_GRS);
strcat(cCHAR_CONTAINER, cCHAR_CONTAINER_STEP);
snprintf(cCHAR_CONTAINER_STEP, 125, "STEP DURATION = %.2f sec.\n", (tv_time_stop.
    tv_sec - tv_time_start.tv_sec) + 1e-6*(tv_time_stop.tv_usec - tv_time_start.
    tv_usec));
strcat(cCHAR_CONTAINER, cCHAR_CONTAINER_STEP);
snprintf(cCHAR_CONTAINER_STEP, 125, "GENERATION NUMBER = %d\n", nCURRENT_GEN);
strcat(cCHAR_CONTAINER, cCHAR_CONTAINER_STEP);
snprintf(cCHAR_CONTAINER_STEP, 125, "CURRENT FITNESS = %f\n", bBest->fitness);
strcat(cCHAR_CONTAINER, cCHAR_CONTAINER_STEP);
snprintf(cCHAR_CONTAINER_STEP, 125, "TOTAL NUMBER OF POINTS CHANGED = %d\n\n",
    nSTEP * nNEW_POINTS_PER_GRS_STEP);
strcat(cCHAR_CONTAINER, cCHAR_CONTAINER_STEP);

strcat(cGRS_STEPS, cCHAR_CONTAINER);

//Copy stop timeval to start timeval
tv_time_start = tv_time_stop;

//Current GRS step
nSTEP++;

#ifdef __CUDACC__

cudaDeviceProp prop;
cudaGetDeviceProperties(&prop, 0);

double* d_fSAMPLE;
```

```
      CUDA_SAFE_CALL(cudaMalloc((void**)&d_fSAMPLE, sizeof(double)*2*nNB_SAMPLES));

      CUDA_SAFE_CALL(cudaMemcpy(d_fSAMPLE, fSAMPLE, sizeof(double)*2*nNB_SAMPLES,
          cudaMemcpyHostToDevice));

      dim3 grid = compute_dimensions(&prop, 2*nNB_SAMPLES);

      updated_fgpuSample<<<grid, prop.maxThreadsPerBlock>>>(d_fSAMPLE, 2*nNB_SAMPLES);

      cudaDeviceSynchronize();

      cudaFree(d_fSAMPLE);

      #endif
}

//Count down the number of generations left before next GRS step
nGENS_BEFORE_GRS_CHANGE--;

\end

\At the end of each generation function:

//Counter for the number of elapsed generations
nCURRENT_GEN++;
//Adjust ranges to mutate the parameters

// if(nCURRENT_GEN==1){
//   fbestinitialfitness=bBest->fitness;
// }else{
//   fintensity_delta_amp*=bBest->fitness/(100.0*fbestinitialfitness);
//   fintensity_delta_freq*=bBest->fitness/(100.0*fbestinitialfitness);
//   fintensity_delta_phase*=bBest->fitness/(100.0*fbestinitialfitness);
//   fintensity_delta_decay*=bBest->fitness/(100.0*fbestinitialfitness);
// }

//Decrement counter for user regular display
nUSERDISPLAY_COUNTER--;
if(nUSERDISPLAY_COUNTER==0){ // When the counter reaches zero
  //Sort the values of the best individual by frequency
  quicksort_freq(bBest->Sin, 0, nNB_SIN-1);
  //Display the best individual
  printf("\n-----------------------------------------------------------\n");
  printf("Obtained function sorted by frequency: \ny=exp(-%.15f*x)*%.15f*sin(%.15f*x+%.15
      f)",-log(bBest->Sin[3]/100.0)/nTRANSIENT_SIZE,bBest->Sin[0],bBest->Sin[1],bBest->
      Sin[2]);
  for (int i=1;i<nNB_SIN;i++)
      printf("+exp(-%.15f*x)*%.15f*sin(%.15f*x+%.15f)",-log(bBest->Sin[i*4+3]/100.0)/
          nTRANSIENT_SIZE,bBest->Sin[i*4+0],bBest->Sin[i*4+1],bBest->Sin[i*4+2]);
  printf("\n-----------------------------------------------------------\n");

  nUSERDISPLAY_COUNTER=nDISPLAY_EVERY; // Initialize counter back to nDISPLAY_EVERY again
}

\end

\At each generation before reduce function:
  //cout << "At each generation before replacement function called" << endl;
\end

\User classes :
GenomeClass {           // EASEA only implements single dimension arrays, so
  double Sin[4*MAX_SIN];// sin[i*4+0] = amplitude of sine i
                        // sin[i*4+1] = frequency of sine i
                        // sin[i*4+2] = phase of sine i
                        // sin[i*4+3] = decay of sine i
}
```

```
\end

\GenomeClass::display:
\end

\GenomeClass::initialiser : // "initializer" is also accepted

if(bFINE_ISOTOPIC){

  for(int i=0; i<nNB_SIN; i++){
    Genome.Sin[i*4+0]=random((double)fMIN_AMP, (double)fMAX_AMP);
    Genome.Sin[i*4+1]=random((double)fMIN_FREQ, (double)fMAX_FREQ);
    Genome.Sin[i*4+2]=554.7650323280 -2058.3263117462*Genome.Sin[i*4+1]; //random((double
        )fMIN_PH, (double)fMAX_PH);
    if(bNO_DECAY==1){
      Genome.Sin[i*4+3]=100.0;
    }else{
      Genome.Sin[i*4+3]=random((double)fMIN_EXP,(double)fMAX_EXP);
    }
  }

}else{

  for(int i=0; i<nNB_SIN; i++){
    Genome.Sin[i*4+0]=random((double)fMIN_AMP, (double)fMAX_AMP);
    Genome.Sin[i*4+1]=random((double)fMIN_FREQ, (double)fMAX_FREQ);
    Genome.Sin[i*4+2]=random((double)fMIN_PH, (double)fMAX_PH);
    if(bNO_DECAY==1){
      Genome.Sin[i*4+3]=100.0;
    }else{
      Genome.Sin[i*4+3]=random((double)fMIN_EXP,(double)fMAX_EXP);
    }
  }

}

\end

\GenomeClass::crossover : // create child (initialized to parent1) out of parent1 and
    parent2
int nLocus=random(0,nNB_SIN);
for(int i=0; i<nLocus; i++){
  child.Sin[i*4+0]=parent2.Sin[i*4+0];
  child.Sin[i*4+1]=parent2.Sin[i*4+1];
  child.Sin[i*4+2]=parent2.Sin[i*4+2];
  child.Sin[i*4+3]=parent2.Sin[i*4+3];
}
\end

\GenomeClass::mutator:

if(bFINE_ISOTOPIC){

  float fpMut=3/((float)nNB_SIN);  // Probability of mutating a sine

  for (int i=0;i<nNB_SIN;i++){
    if (tossCoin(fpMut)){
          if (tossCoin(fmutator_amp_rate)){ // Mutate amplitude within [fMIN_AMP,fMAX_AMP
              ]
             double delta_amp = fintensity_delta_amp-random(0.0, 2*fintensity_delta_amp);
             if(Genome.Sin[i*4+0]+delta_amp<fMIN_AMP || Genome.Sin[i*4+0]+delta_amp>
                 fMAX_AMP){
               delta_amp*=-1;
             }
             Genome.Sin[i*4+0]+=delta_amp;
          }
```

```
        if (tossCoin(fmutator_freq_rate)){ // Mutate frequency within [fMIN_FREQ,
             fMAX_FREQ]
           double delta_freq = fintensity_delta_freq-random(0.0, 2*
                fintensity_delta_freq);
           if(Genome.Sin[i*4+1]+delta_freq<fMIN_FREQ || Genome.Sin[i*4+1]+delta_freq>
                fMAX_FREQ){
             delta_freq*=-1;
           }
           Genome.Sin[i*4+1]+=delta_freq;

           Genome.Sin[i*4+2]=554.7650323280 -2058.3263117462*Genome.Sin[i*4+1];
        }

        if(bNO_DECAY==0 && tossCoin(fmutator_dec_rate)){ // Mutate decay within [
             fMIN_EXP, fMAX_EXP]
           double delta_exp = fintensity_delta_decay-random(0.0, 2*
                fintensity_delta_decay);
           if(Genome.Sin[i*4+3]+delta_exp<fMIN_EXP || Genome.Sin[i*4+3]+delta_exp>
                fMAX_EXP){
             delta_exp*=-1;
           }
           Genome.Sin[i*4+3]+=delta_exp;
        }
     }
  }

  //Sort the values by frequency
  quicksort_freq(Genome.Sin, 0, nNB_SIN-1);

  for (int i=0;i<nNB_SIN-1;i++){
    if (Genome.Sin[i*4+0])
      if (fabs(Genome.Sin[i*4+1]-Genome.Sin[(i+1)*4+1])<fepsilon_freq){ // Then, we merge
           the sines and we "remove" the first one
        double a0, a1, a2, f, p0, p1, p2, e0, e1, e2;  // we will create a2, e2 and p2
             out of a0, a1, f, p0, p1
        // We take for a common frequency the weighted mean of the frequency of the two
             sines
        f=Genome.Sin[(i+1)*4+1]=(Genome.Sin[(i)*4+1]*Genome.Sin[(i)*4+0]+Genome.Sin[(i+1)
             *4+1]*Genome.Sin[(i+1)*4+0])/(Genome.Sin[(i+0)*4+0]+Genome.Sin[(i+1)*4+0]);
        a0=Genome.Sin[(i+0)*4+0]; a1=Genome.Sin[(i+1)*4+0]; p0=Genome.Sin[(i+0)*4+2]; p1=
             Genome.Sin[(i+1)*4+2];
        e0=Genome.Sin[(i+0)*4+3]; e1=Genome.Sin[(i+1)*4+3];
        a2=sqrt(powf(a0*cos(p0)+a1*cos(p1),2)+powf(a0*sin(p0)+a1*sin(p1),2));
        //p2=atan((a0*sin(p0)+a1*sin(p1))/(a0*cos(p0)+a1*cos(p1)));
        p2=554.7650323280 -2058.3263117462*f;
        e2=(e0*a0+e1*a1)/(a0+a1);
        //If the new amplitude, phase and decay are in the boundaries
        if(a2<fMAX_AMP && a2>fMIN_AMP && p2<fMAX_PH && p2>fMIN_PH && e2<fMAX_EXP && e2>
             fMIN_EXP){
            Genome.Sin[(i+1)*4+0]=a2;
            Genome.Sin[(i+1)*4+1]=f;
            //Genome.Sin[(i+1)*4+2]=p2;
            Genome.Sin[(i+1)*4+2]=554.7650323280 -2058.3263117462*f;
            Genome.Sin[(i+1)*4+3]=e2;
        }else{
          //Create a new random sine instead of merging
          Genome.Sin[(i+1)*4+0]=random((double)fMIN_AMP, (double)fMAX_AMP);
          Genome.Sin[(i+1)*4+1]=random((double)fMIN_FREQ, (double)fMAX_FREQ);
          Genome.Sin[(i+1)*4+2]=554.7650323280 -2058.3263117462*Genome.Sin[(i+1)*4+1]; //
               random((double)fMIN_PH, (double)fMAX_PH);
          if(bNO_DECAY==1){
            Genome.Sin[(i+1)*4+3]=100.0;
          }else{
            Genome.Sin[(i+1)*4+3]=random((double)fMIN_EXP,(double)fMAX_EXP);
          }
        }
        //Create a new random sine
```

```
        Genome.Sin[i*4+0]=random((double)fMIN_AMP, (double)fMAX_AMP);
        Genome.Sin[i*4+1]=random((double)fMIN_FREQ, (double)fMAX_FREQ);
        Genome.Sin[i*4+2]=554.7650323280 -2058.3263117462*Genome.Sin[i*4+1]; //random((
            double)fMIN_PH, (double)fMAX_PH);
        if(bNO_DECAY==1){
          Genome.Sin[i*4+3]=100.0;
        }else{
          Genome.Sin[i*4+3]=random((double)fMIN_EXP,(double)fMAX_EXP);
        }
     }
  }

}else{

  float fpMut=3/((float)nNB_SIN);  // Probability of mutating a sine
  for (int i=0;i<nNB_SIN;i++){
    if (tossCoin(fpMut)){
          if (tossCoin(fmutator_amp_rate)){ // Mutate amplitude within [fMIN_AMP,fMAX_AMP
               ]
             double delta_amp = fintensity_delta_amp-random(0.0, 2*fintensity_delta_amp);
             if(Genome.Sin[i*4+0]+delta_amp<fMIN_AMP || Genome.Sin[i*4+0]+delta_amp>
                  fMAX_AMP){
               delta_amp*=-1;
             }
             Genome.Sin[i*4+0]+=delta_amp;
          }

          if (tossCoin(fmutator_freq_rate)){ // Mutate frequency within [fMIN_FREQ,
               fMAX_FREQ]
             double delta_freq = fintensity_delta_freq-random(0.0, 2*
                  fintensity_delta_freq);
             if(Genome.Sin[i*4+1]+delta_freq<fMIN_FREQ || Genome.Sin[i*4+1]+delta_freq>
                  fMAX_FREQ){
               delta_freq*=-1;
             }
             Genome.Sin[i*4+1]+=delta_freq;
          }

          if (tossCoin(fmutator_ph_rate)){ // Mutate phase within [fMIN_PH, fMAX_PH]
            double delta_phase = fintensity_delta_phase-random(0.0, 2*
                fintensity_delta_phase);
            double new_phase = Genome.Sin[i*4+2]+delta_phase;
            if(new_phase<0){
              new_phase+=PI2;
            }else{
              new_phase=fmod(Genome.Sin[i*4+2]+delta_phase, PI2);
            }
            Genome.Sin[i*4+2]=new_phase;
          }

          if(bNO_DECAY==0 && tossCoin(fmutator_dec_rate)){ // Mutate decay within [
               fMIN_EXP, fMAX_EXP]
             double delta_exp = fintensity_delta_decay-random(0.0, 2*
                  fintensity_delta_decay);
             if(Genome.Sin[i*4+3]+delta_exp<fMIN_EXP || Genome.Sin[i*4+3]+delta_exp>
                  fMAX_EXP){
               delta_exp*=-1;
             }
             Genome.Sin[i*4+3]+=delta_exp;
          }
      }
  }

  // This is a subtlety to improve the efficiency of the crossover
  // for (int i=0;i<nNB_SIN-1;i++){  // an evo-bub sort on the frequency :-)
  //
  // if (Genome.Sin[i*4+1]>Genome.Sin[(i+1)*4+1]){ // only one bubble goes up
```

```
  //     double ampTemp, freqTemp, phaseTemp, decTemp; // the generations do the global
      sorting
  //     ampTemp=Genome.Sin[i*4+0];
  //     freqTemp=Genome.Sin[i*4+1];
  //     phaseTemp=Genome.Sin[i*4+2];
  //     decTemp=Genome.Sin[i*4+3];
  //     Genome.Sin[i*4+0]=Genome.Sin[(i+1)*4+0];
  //     Genome.Sin[i*4+1]=Genome.Sin[(i+1)*4+1];
  //     Genome.Sin[i*4+2]=Genome.Sin[(i+1)*4+2];
  //     Genome.Sin[i*4+3]=Genome.Sin[(i+1)*4+3];
  //     Genome.Sin[(i+1)*4+0]=ampTemp;
  //     Genome.Sin[(i+1)*4+1]=freqTemp;
  //     Genome.Sin[(i+1)*4+2]=phaseTemp;
  //     Genome.Sin[(i+1)*4+3]=decTemp;
  //
  // } }

  //Sort the values by frequency
  quicksort_freq(Genome.Sin, 0, nNB_SIN-1);


  for (int i=0;i<nNB_SIN-1;i++){
    if (Genome.Sin[i*4+0])
      if (fabs(Genome.Sin[i*4+1]-Genome.Sin[(i+1)*4+1])<fepsilon_freq){ // Then, we merge
          the sines and we "remove" the first one
        double a0, a1, a2, f, p0, p1, p2, e0, e1, e2;  // we will create a2, e2 and p2
            out of a0, a1, f, p0, p1
        // We take for a common frequency the weighted mean of the frequency of the two
            sines
        f=Genome.Sin[(i+1)*4+1]=(Genome.Sin[(i)*4+1]*Genome.Sin[(i)*4+0]+Genome.Sin[(i+1)
            *4+1]*Genome.Sin[(i+1)*4+0])/(Genome.Sin[(i+0)*4+0]+Genome.Sin[(i+1)*4+0]);
        a0=Genome.Sin[(i+0)*4+0]; a1=Genome.Sin[(i+1)*4+0]; p0=Genome.Sin[(i+0)*4+2]; p1=
            Genome.Sin[(i+1)*4+2];
        e0=Genome.Sin[(i+0)*4+3]; e1=Genome.Sin[(i+1)*4+3];
        a2=sqrt(powf(a0*cos(p0)+a1*cos(p1),2)+powf(a0*sin(p0)+a1*sin(p1),2));
        p2=atan((a0*sin(p0)+a1*sin(p1))/(a0*cos(p0)+a1*cos(p1)));
        e2=(e0*a0+e1*a1)/(a0+a1);
        //If the new amplitude, phase and decay are in the boundaries
        if(a2<fMAX_AMP && a2>fMIN_AMP && p2<fMAX_PH && p2>fMIN_PH && e2<fMAX_EXP && e2>
            fMIN_EXP){
            Genome.Sin[(i+1)*4+0]=a2; Genome.Sin[(i+1)*4+1]=f; Genome.Sin[(i+1)*4+2]=p2;
                Genome.Sin[(i+1)*4+3]=e2;
        }else{
          //Create a new random sine instead of merging
          Genome.Sin[(i+1)*4+0]=random((double)fMIN_AMP, (double)fMAX_AMP);
          Genome.Sin[(i+1)*4+1]=random((double)fMIN_FREQ, (double)fMAX_FREQ);
          Genome.Sin[(i+1)*4+2]=random((double)fMIN_PH, (double)fMAX_PH);
          if(bNO_DECAY==1){
            Genome.Sin[(i+1)*4+3]=100.0;
          }else{
            Genome.Sin[(i+1)*4+3]=random((double)fMIN_EXP,(double)fMAX_EXP);
          }
        }
        //Create a new random sine
        Genome.Sin[i*4+0]=random((double)fMIN_AMP, (double)fMAX_AMP);
        Genome.Sin[i*4+1]=random((double)fMIN_FREQ, (double)fMAX_FREQ);
        Genome.Sin[i*4+2]=random((double)fMIN_PH, (double)fMAX_PH);
        if(bNO_DECAY==1){
          Genome.Sin[i*4+3]=100.0;
        }else{
          Genome.Sin[i*4+3]=random((double)fMIN_EXP,(double)fMAX_EXP);
        }
      }
  }
}
\end
```

```
\GenomeClass::evaluator: // Returns the score
#ifndef __CUDA_ARCH__ //if __host__ code part
if(bBRAD_INT){
    return (double)fScoreOnGPUbradint(Genome.Sin);
}else{
    return (double)fScoreOnGPU_L2(Genome.Sin);
}
#else //if __device__ code part
if(d_bBRAD_INT){
    return (double)fScoreOnGPUbradint(Genome.Sin);
}else{
    return (double)fScoreOnGPU_L2(Genome.Sin);
}
#endif
\end

\User Makefile options:
\end

\Default run parameters :         // Please let the parameters appear in this order
  Number of generations : 2048   // NB_GEN
  Time limit: 0                   // In seconds, 0 to deactivate
  Population size : 131072 // 16384 // 32768 // 4096
  Offspring size : 100%
  Mutation probability : 1        // MUT_PROB
  Crossover probability : 1       // XOVER_PROB
  Evaluator goal : minimise       // Maximise
  Selection operator: Tournament 20
  Surviving parents: 100%         // percentage or absolute
  Surviving offspring: 100%
  Reduce parents operator: Tournament 2
  Reduce offspring operator: Tournament 2
  Final reduce operator: Tournament 20  // 12

  Elitism: weak        // Weak (best of parents+offspring) or Strong (best of parents)
  Elite: 1
  Print stats: true
  Generate csv stats file:false
  Generate gnuplot script:false
  Generate R script:false
  Plot stats:false

  Remote island model: false
  IP file: ip.txt                  // File containing all the remote island's IP
  Server port : 2929
  Migration probability: 0.333

  Save population: false
  Start from file:false
\end
```

Université de Strasbourg

MSAP

Ulviya ABDULKARIMOVA

SINUS-IT: an evolutionary approach to harmonic analysis

Résumé en français

L'analyse spectrale est l'une des branches principales du traitement du signal. De nombreux appareils expérimentaux produisent des signaux qui sont des sommes de sinus amorties. Avec perfectionnements de ces appareils, le volume de données qu'ils produisent ne cesse de grandir. Dans cette thèse, nous concentrerons sur des données issues d'un spectromètre de masse à résonance cyclonique ionique transformée de Fourier (FT-ICR) et des données simulées. Notre contribution consiste à explorer l'apport des méthodes évolutives pour surmonter les limitations de la méthode de la transformer de Fourier (FT). Nous avons effectué une étude comparative des méthodes par FT et évolution artificielle. Les résultats obtenus avec SINUS-IT sont de meilleure qualité que ceux de FT, sans nécessité de débruitage ou apodisation. SINUS-IT a pu déterminer le paramètre de phase avec une bonne précision et les résultats ont été obtenus en utilisant moins d'échantillons, ce qui réduirait le temps d'acquisition.

Mots clés: *Évolution artificielle, stratégies d'évolution, algorithmes génétiques, transformée de fourier analyse harmonique, FT-ICR, structure isotopique, parallélisation GPGPU, parallélisation en îlots.*

Résumé en anglais

One of the main branches of signal processing is spectral analysis. Many experimental devices produce signals which are sums of damped sines. But with advancements in these devices, the volume of data they generate continues to grow. In this thesis, we focus on data from Fourier Transform Ion Cyclotron Resonance Mass Spectrometer (FT-ICR) and also on a simulated data. Our contribution consists in exploring the contribution of evolutionary methods to overcome the limitations of the Fourier Transform (FT) method. We carried out a comparative study of the methods by FT and by artificial evolution. The results obtained with SINUS-IT are of better quality than those of FT, without requiring denoising or apodization. SINUS-IT was able to determine phase parameter with good precision and results were obtained using less number of samples which would decrease the acquisition time.

Keywords: *Artificial Evolution, Evolution Strategies, Genetic Algorithms, Fourier Transform, Harmonic Analysis, FT-ICR, isotopic structure, GPGPU parallelization, island-based parallelization*