ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Modulated Fourier expansions for ODEs with oscillatory solutions

Bachelor Thesis

Ambra Toletti

Spring semester 2012

Advisors: Prof. Dr. R.Hiptmair
Department of Mathematics, ETH Zürich

Abstract

The aim of my theses is to investigate the application of collocation approach, with modulated Fourier expansions as trial set, to ordinary differential equations with oscillatory solutions and to implement the obtained method using MATLAB [6]. This will be done as follows.

First of all (chapter 1) we recall the definitions of *second order differential equation with oscillatory solution* and *modulated Fourier expansion* and we present the *collocation approach* for approximating the solution of a differential equation.

Then (chapter 2) we apply the collocation approach to the equation of a perturbed oscillation using the set of modulated Fourier expansions as trial set for getting a model that can be implemented with MATLAB. Here we distinguish two classes of perturbed oscillations: the oscillations whose perturbation depends on the state and the ones whose doesn't. We also identify two classes of numerical methods: the *matrix form*'s and the *one step form*'s ones.

It follows the implementation of the methods using MATLAB (chapter 3) and two numerical experiments (chapters 4 and 5), where the implementations are used and compared. The third chapter contains the functions corresponding to the methods developed in the previous chapter: an *ad hoc* method for oscillations whose perturbations depend on the time only, a general matrix form's method for oscillations with arbitrary perturbation function and finally an implementation of the one step form's method. Applications of these functions can be found in chapters four and five together with the study of obtained errors.

Finally (chapter 6) the results of the numerical experiments are used for drawing conclusions about the different methods and their implementations (precision, efficiency, ...).

Contents

Chapter 1

Introduction

In this chapter we define the problem and some tools that are useful to solve it (or to find an approximation of its solution).

1.1 ODEs with oscillatory solutions

The subject of my theses are *ordinary differential equations (ODEs) with oscillatory solutions*, i.e. equations of the form

$$\ddot{z} + \omega^2 z = g(t, z) \tag{1.1}$$

with given *initial conditions*

$$z(t_0) = z_0 \text{ and } \dot{z}(t_0) = \dot{z}_0 \tag{1.2}$$

where t corresponds to the *time variable* ($t \in I \subset \mathbb{R}^{\geq 0}$), z represents the *state* at time t ($z : I \to D$ for some open $D \subset \mathbb{R}^d$, $d \in \mathbb{N}$), $\omega \in \mathbb{R}^*$ is some given parameter and $g : I \times D =: \Omega \to \mathbb{R}^d$ is a given function.

Remarks:

1. Equation (1.1) can be seen as a *linear oscillation* (see next subsection) with a *perturbation* $g(t, z)$.
2. By setting $\mathbf{y}(t) := \begin{pmatrix} z(t) \\ \dot{z}(t) \end{pmatrix}$ and consequently $\dot{\mathbf{y}}(t) := \begin{pmatrix} \dot{z}(t) \\ \ddot{z}(t) \end{pmatrix}$, (1.1) can be transformed in the first-order ordinary differential equation

$$\dot{\mathbf{y}} = \begin{pmatrix} 0 & 1 \\ -\omega^2 & 0 \end{pmatrix} \mathbf{y} + \begin{pmatrix} 0 \\ g(t, \mathbf{y}_1) \end{pmatrix} \tag{1.3}$$

with initial condition

$$\mathbf{y}(t_0) = \mathbf{y}_0 := \begin{pmatrix} z_0 \\ \dot{z}_0 \end{pmatrix} \tag{1.4}$$

The right hand side of (1.3) can be written as a function $\mathbf{f} : \Omega \to \mathbb{R}^{2d}$ that describes the relation between $\dot{\mathbf{y}}$ and $(t, \mathbf{y})$, i.e. (1.3) can be rewritten as $\dot{\mathbf{y}} = \mathbf{f}(t, \mathbf{y})$.

3. From now on we'll just consider the one dimensional case, i.e. we set $d = 1$ to be fixed.

1.1.1 The linear oscillation

The linear oscillation is a special case of ODE with oscillatory solution. This corresponds to the case $g \equiv 0$ and is given by the following second-order ODE:

$$\ddot{z} = -\omega^2 z \tag{1.5}$$

The one-dimensional case (i.e. $d = 1$) has exact solution given by

$$z(t) = \alpha \cos(\omega t) + \beta \sin(\omega t) \tag{1.6}$$

with α, β some constant values that can be determined using initial conditions.
In this case, by setting $\mathbf{y}$ as above we can rewrite equation (1.5) as follows

$$\dot{\mathbf{y}} = \begin{pmatrix} 0 & 1 \\ -\omega^2 & 0 \end{pmatrix} \mathbf{y} \tag{1.7}$$

i.e. a first-order ODE in $\mathbb{R}^2$ with solution

$$\mathbf{y}(t) = \begin{pmatrix} \alpha \cos(\omega t) + \beta \sin(\omega t) \\ \omega(-\alpha \sin(\omega t) + \beta \cos(\omega t)) \end{pmatrix} \tag{1.8}$$

1.2 Collocation approach

In this section we present the *collocation approach* (cfr. section 2.2.1 in [4], section 6.3 in [1] and section II.1.2 in [3]), that is a method for finding a numerical approximation of $\mathbf{y}(t_1)$ for some $t_1 \in I$, where $\mathbf{y}$ is the solution of problem (1.3-4).

Recall the first-order differential equation (1.3-4)

$$\dot{\mathbf{y}} = \mathbf{f}(t, \mathbf{y}) \tag{1.9}$$

with initial condition

$$\mathbf{y}(t_0) = \mathbf{y}_0 \tag{1.10}$$

The idea of collocation's approach is to find a function $\mathbf{y}_h : [t_0, t_1] \to \mathbb{R}^2$ that approximates the solution $\mathbf{y}(t)$ of the ODE for $t \in [t_0, t_1]$ in some $s+1$-dimensional trial space V ($s \in \mathbb{N}$ depends on the choice of V). Often V is

set to be $\mathcal{P}_s$ (the set of univariate polynomials of degree at most s), but later (see next section) we will choose a different V.

The function $\mathbf{y}_h \in V$ is then defined by the following initial and *collocation conditions*

$$\mathbf{y}_h(t_0) = \mathbf{y}_0 \tag{1.11}$$
$$\dot{\mathbf{y}}_h(\tau_j) = \mathbf{f}(\tau_j, \mathbf{y}_h(\tau_j)), j = 1, \ldots, s \tag{1.12}$$

where $s = \dim V - 1$ and $t_0 \leq \tau_1 < \ldots < \tau_s \leq t_1$ are called the *collocation's points* and can be chosen in several ways (e.g. equidistant points, Gauss points, etc). We require the collocation's points $\{\tau_j\}$ to be unisolvent for the function set V.

1.3 Truncated modulated Fourier expansions

For fixed $p, m \in \mathbb{N}$ the set of *truncated modulated Fourier expansions* $V^{m,p}$ is given by functions

$$v(t) = \sum_{k=-m}^{m} e^{ik\omega t} \cdot \mu_k(t) \tag{1.13}$$

where $\mu_k \in \mathcal{P}_p$. In this case the dimension of $V^{m,p}$ is given by

$$\dim(V^{m,p}) = (2m+1)(p+1) \tag{1.14}$$

Remark: In (1.13) the exponential term may be replaced by $\cos(k\omega t)$ and $\sin(k\omega t)$:

$$v(t) = \mu_0(t) + \sum_{k=1}^{m} (\sin(k\omega t)\mu_k(t) + \cos(k\omega t)\mu_{-k}(t)) \tag{1.15}$$

Chapter 2

Collocation for ODEs with oscillatory solutions

Consider the ODE defined in section 1.1 and let $t_1 \in I$ be some time for which one wants to compute an approximative solution $z_h(t_1)$ using the collocation method defined in section 1.2, with truncated modulated Fourier expansions (section 1.3) as trial space. This approach is usually ([7], page 453) called the *method of envelopes*: the $\left\{e^{ik\omega t}\right\}$ are called the *carriers* and the $\{\mu_k\}$ the *envelopes*.

2.1 Collocation conditions

Consider the ODE given by (1.1-2) in the form described in (1.3-4). Define $\mathbf{y}_h \in V^{m,p}$ to be as in section 1.2, i.e. $\mathbf{y}_h$ is defined by conditions (1.11-12) that, in this case become

$$\mathbf{y}_h(t_0) = \mathbf{y}_0 = \begin{pmatrix} z_0 \\ \dot{z}_0 \end{pmatrix} \tag{2.1}$$

$$\dot{\mathbf{y}}_h(\tau_j) = \begin{pmatrix} 0 & 1 \\ -\omega^2 & 0 \end{pmatrix} \mathbf{y}_h(\tau_j) + \begin{pmatrix} 0 \\ g\left(\tau_j, \mathbf{y}_{h,1}(\tau_j)\right) \end{pmatrix}, j = 1, \ldots, s \tag{2.2}$$

Fix $m, p \in \mathbb{N}$ and let $V^{m,p}$ be as defined in (1.13). Let $z_h \in V^{m,p}$, then

$$z_h(t) = \sum_{k=-m}^{m} e^{ik\omega t} \cdot \mu_k(t) \tag{2.3}$$

$$\dot{z}_h(t) = \sum_{k=-m}^{m} e^{ik\omega t} \cdot \left(ik\omega\mu_k(t) + \dot{\mu}_k(t)\right) \tag{2.4}$$

$$\ddot{z}_h(t) = \sum_{k=-m}^{m} e^{ik\omega t} \cdot \left(-k^2\omega^2\mu_k(t) + 2ik\omega\dot{\mu}_k(t) + \ddot{\mu}_k(t)\right) \tag{2.5}$$

Set $\mathbf{y}_h(t) = \begin{pmatrix} z_h(t) \\ \dot{z}_h(t) \end{pmatrix}$. We get

$$\mathbf{y}_h(t) = \sum_{-m}^{m} e^{ik\omega t} \cdot \begin{pmatrix} \mu_k(t) \\ ik\omega\mu_k(t) + \dot{\mu}_k(t) \end{pmatrix} \tag{2.6}$$

$$\dot{\mathbf{y}}_h(t) = \sum_{-m}^{m} e^{ik\omega t} \cdot \begin{pmatrix} ik\omega\mu_k(t) + \dot{\mu}_k(t) \\ -k^2\omega^2\mu_k(t) + 2ik\omega\dot{\mu}_k(t) + \ddot{\mu}_k(t) \end{pmatrix} \tag{2.7}$$

Using

$$\mathbf{y}_h(t_0) = \sum_{k=-m}^{m} e^{ik\omega t_0} \cdot \begin{pmatrix} \mu_k(t_0) \\ ik\omega\mu_k(t_0) + \dot{\mu}_k(t_0) \end{pmatrix}$$
$$\mathbf{y}_0 = \begin{pmatrix} z_0 \\ \dot{z}_0 \end{pmatrix}$$

can rewrite the initial condition (2.1) as

$$\sum_{k=-m}^{m} e^{ik\omega t_0} \cdot \begin{pmatrix} \mu_k(t_0) \\ ik\omega\mu_k(t_0) + \dot{\mu}_k(t_0) \end{pmatrix} = \begin{pmatrix} z_0 \\ \dot{z}_0 \end{pmatrix} \tag{2.8}$$

and using (2.6-7) we can rewrite collocation conditions (2.2) as

$$\begin{aligned} &\sum_{-m}^{m} e^{ik\omega\tau_j} \cdot \begin{pmatrix} ik\omega\mu_k(\tau_j) + \dot{\mu}_k(\tau_j) \\ -k^2\omega^2\mu_k(\tau_j) + 2ik\omega\dot{\mu}_k(\tau_j) + \ddot{\mu}_k(\tau_j) \end{pmatrix} = \dot{\mathbf{y}}_h(\tau_j) \\ &= \begin{pmatrix} 0 & 1 \\ -\omega^2 & 0 \end{pmatrix} \mathbf{y}_h(\tau_j) + \begin{pmatrix} 0 \\ g\left(\tau_j, \mathbf{y}_{h,1}(tau_j)\right) \end{pmatrix} \\ &= \sum_{-m}^{m} e^{ik\omega\tau_j} \cdot \begin{pmatrix} ik\omega\mu_k(\tau_j) + \dot{\mu}_k(\tau_j) \\ -\omega^2 \cdot \mu_k(\tau_j) \end{pmatrix} + \begin{pmatrix} 0 \\ g\left(\tau_j, \sum_{k=-m}^{m} e^{ik\omega\tau_j} \cdot \mu_k(\tau_j)\right) \end{pmatrix} \end{aligned} \tag{2.9}$$

The condition for the first component in (2.9) is, naturally, always satisfied, hence we reduce (2.9) to

$$\begin{aligned} \ddot{z}_h(\tau_j) &= \sum_{k=-m}^{m} e^{ik\omega\tau_j} \cdot \left(-k^2\omega^2\mu_k(\tau_j) + 2ik\omega\dot{\mu}_k(\tau_j) + \ddot{\mu}_k(\tau_j)\right) \\ &= -\omega^2 \sum_{k=-m}^{m} e^{ik\omega\tau_j} \cdot \mu_k(\tau_j) + g\left(\tau_j, \sum_{k=-m}^{m} e^{ik\omega\tau_j} \cdot \mu_k(\tau_j)\right) \\ &= -\omega^2 z_h(\tau_j) + g(\tau_j, z_h(\tau_j)) \end{aligned} \tag{2.10}$$

where $t_0 \leq \tau_1 < \ldots < \tau_s \leq t_1$ as in (1.12).

Note that here $s = \dim V^{m,p} - 2$ (for $p > 0$) because, since $\dim \frac{d^2}{dt^2} V^{m,p} = \dim V^{m,p} - 2$, one can obtain at most $\dim V^{m,p} - 2$ linearly independent equations of the form (2.10). Other two equations are required in order to have a

uniquely determined system and they are given by (2.8), i.e.

$$\begin{cases} z_h(t_0) = z_0 \\ \dot{z}_h(t_0) = \dot{z}_0 \end{cases} \tag{2.11}$$

2.2 Special case: the linear oscillation

Consider ODE (1.5) with some initial conditions $z(t_0) = z_0$ and $\dot{z}(t_0) = \dot{z}_0$. In this case the first collocation's condition is given by (2.11) and the second one is obtained by removing the perturbation in (2.10), i.e. we have

$$\begin{aligned} \ddot{z}_h(\tau_j) &= \sum_{k=-m}^{m} e^{ik\omega\tau_j} \cdot \left(-k^2\omega^2\mu_k(\tau_j) + 2ik\omega\dot{\mu}_k(\tau_j) + \ddot{\mu}_k(\tau_j)\right) \\ &= -\omega^2 \sum_{k=-m}^{m} e^{ik\omega\tau_j} \cdot \mu_k(\tau_j) \end{aligned} \tag{2.12}$$

for $t_0 \leq \tau_1 < \ldots < \tau_{(2m+1)(p+1)-2} \leq t_1$.

2.2.1 The matrix form

Last equations can also be written as

$$\sum_{k=-m}^{m} e^{ik\omega\tau_j} \cdot \left(-\omega^2\left(k^2-1\right) \cdot \mu_k(\tau_j) + 2ik\omega\dot{\mu}_k(\tau_j) + \ddot{\mu}_k(\tau_j)\right) = 0 \tag{2.13}$$

These together with (2.11) give a linear system of $(2m+1)(p+1)$ equalities in $(2m+1)(p+1)$ variables (because of (1.14)) that can be written as follows

$$\mathbf{B} \cdot \mathbf{a} = \mathbf{c} \tag{2.14}$$

where

1. $\mathbf{a}$ is a vector containing the coefficients of the polynomials μ_k with the following notation: let $\mu_k(t) = \sum_{d=0}^{p} a_{k,d}t^d$, then put the $\{a_{k,d}\}$ in $\mathbf{a}$ with indices $\{k \cdot (p+1) + d\}$.
   Note that we use indices in $\{-m \cdot (p+1), \ldots, m \cdot (p+1) + p\}$.
2. $\mathbf{c} = \begin{pmatrix} 0 \\ \vdots \\ 0 \\ z_0 \\ \dot{z}_0 \end{pmatrix}$ corresponds to the right hand side of the system

3. $\mathbf{B} = \left\{ b_{j,k\cdot(p+1)+d} \right\}_{1\le j\le(2m+1)(p+1),-m\le k\le m,0\le d\le p}$ is the matrix containing the coefficients.

$$b_{j,k(p+1)+d} = \begin{cases} -e^{ik\omega\tau_j}\omega^2(k^2-1), & \text{if } d=0 \\ e^{ik\omega\tau_j}\left(-\omega^2(k^2-1)\tau_j + 2ik\omega\right), & \text{if } d=1 \\ e^{ik\omega\tau_j}\left(-\omega^2(k^2-1)\tau_j^d + 2ik\omega d\tau_j^{d-1} + d(d-1)\tau_j^{d-2}\right), & \text{if } d\ge 2 \end{cases} \tag{2.15}$$

for $1 \le j \le (2m+1)(p+1)-2$ and

$$b_{(2m+1)(p+1)-1,k(p+1)+d} = e^{ik\omega t_0}t_0^d \tag{2.16}$$

$$b_{(2m+1)(p+1),k(p+1)+d} = \begin{cases} e^{ik\omega t_0}ik\omega, & \text{if } d=0 \\ e^{ik\omega t_0}\left(ik\omega t_0^d + dt_0^{d-1}\right), & \text{if } d\ge 1 \end{cases} \tag{2.17}$$

Hence the linear case is uniquely solvable if and only if the matrix $\mathbf{B}$ is regular, and in the case $\mathbf{a}$ is obtained by computing $\mathbf{B}^{-1}\mathbf{c}$ and it is unique.

Remarks:

1. The approximation $z_h(t_1)$ can be computed as $\Re(\langle \mathbf{a}, \mathbf{x}(t_1)\rangle)$ where $\mathbf{x}(t_1)$ is a vector whose entries correspond to the monomials $t_1^d e^{ik\omega t_1}$ (see function `xvec` in appendix A1 for a MATLAB definition of $\mathbf{x}$), $\langle\cdot,\cdot\rangle$ denotes the scalar product in $\mathbb{C}^{(2m+1)(p+1)}$ and $\Re(\cdot)$ is the real part of a complex number.

2. In section 3.1 is presented a MATLAB function that, for given m, p, t_0, ω and $\tau_1, \ldots, \tau_{(2m+1)(p+1)-2}$, computes matrix $\mathbf{B}$.

2.2.2 Regularity of B

Let us take a look at the condition of $\mathbf{B}$: a condition value near to 1 indicates that $\mathbf{B}$ is well conditioned, i.e. solving a linear system including this matrix won't produce errors. A large condition value, indicates that inverting $\mathbf{B}$ will produces errors and consequently the solution of a linear system including $\mathbf{B}$ by matrix inversion will be inexact, in particular if the condition is equal ∞ the matrix is singular and cannot be inverted. (cfr. documentation of `cond` in [6]).
Since the condition depends on scaling too, before computing the condition it is a good idea to rescale $\mathbf{B}$ s.t. the diagonal contains only 1's. In Figures 2.1 and 2.2 are plotted the conditions of scaled $\mathbf{B}$ corresponding to $m = 1$ and $m = 3$ respectively, with $\omega = 10^2$, $t_0 = 0$ and $t_1 = 1$ for $p = 1, \ldots, 10$ and using both equidistant and Gauss' collocation points in $[t_0, t_1]$.

Figure 2.1: Condition of **B** for $m = 1$ and $\omega = 10^2$

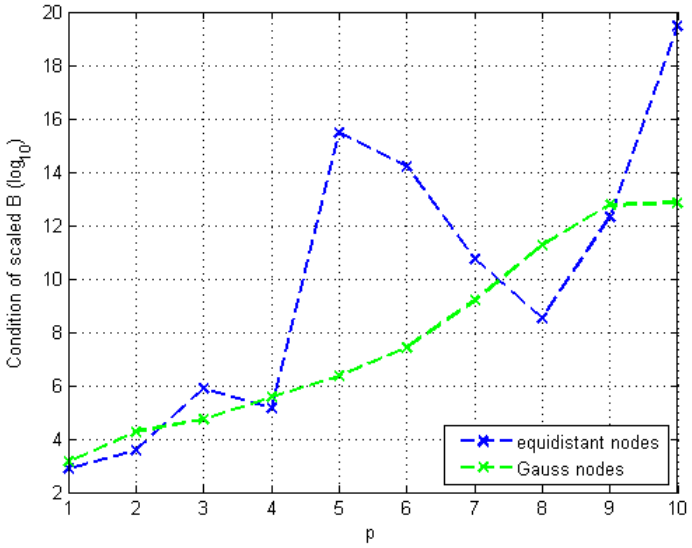
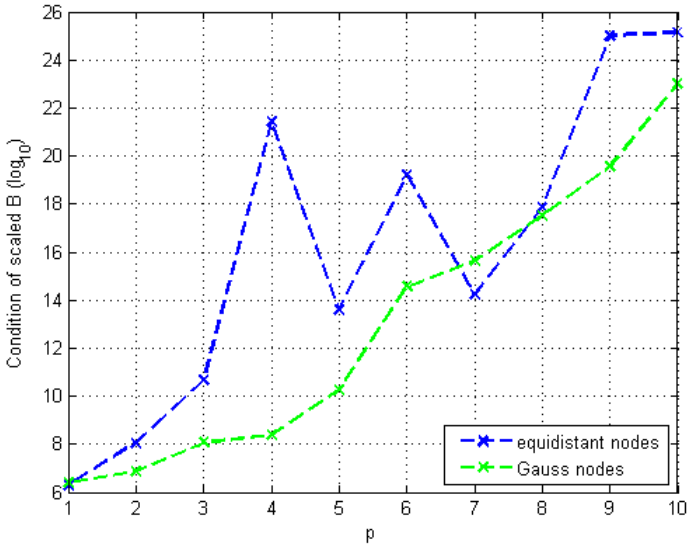


Figure 2.2: Condition of **B** for $m = 3$ and $\omega = 10^2$



Remark: One can see that the condition increases as p increases (and the same happens for different values of m and ω) but it doesn't reach ∞. Therefore, we may conclude that **B** is regular, but for large values of p it becomes very ill-conditioned, and this will produces errors when solving (2.14), and **a** will be inexact.

2.2.3 Exactness of solution

Reminder: The exact solution of the linear oscillation is of the form (1.6). i.e

$$z(t) = \alpha \cos(\omega t) + \beta \sin(\omega t)$$

For fixed initial values $z(t_0) = z_0$ and $\dot{z}(t_0) = \dot{z}_0$ we write

$$\begin{aligned} z(t_0) &= \alpha \cos(\omega t_0) + \beta \sin(\omega t_0) = z_0 \\ \dot{z}(t_0) &= \omega \cdot (-\alpha \sin(\omega t_0) + \beta \cos(\omega t_0)) = \dot{z}_0 \end{aligned}$$

and we get

$$\alpha = z_0 \cos(\omega t_0) - \frac{\dot{z}_0}{\omega} \sin(\omega t_0) \tag{2.18}$$

$$\beta = z_0 \sin(\omega t_0) + \frac{\dot{z}_0}{\omega} \cos(\omega t_0) \tag{2.19}$$

Using the identities

$$\sin(x) = \frac{1}{2i}\left(e^{ix} - e^{-ix}\right) \tag{2.20}$$

$$\cos(x) = \frac{1}{2}\left(e^{ix} + e^{-ix}\right) \tag{2.21}$$

the exact solution can be written in exponential form as follows

$$z(t) = \frac{\alpha + \beta i}{2} e^{-i\omega t} + \frac{\alpha - \beta i}{2} e^{i\omega t} \tag{2.22}$$

Note that $z(t) \in V^{m,p}$, hence we expect that the approximation given by the above method is equal to the exact solution, i.e. $\mathbf{a} = \left(a_{k(p+1)+d}\right)_{-m \le k \le m, 0 \le d \le p}$ satisfies:

$$a_{k(p+1)+d} = \begin{cases} \frac{\alpha+\beta i}{2} & \text{, if } k = -1 \text{ and } d = 0 \\ \frac{\alpha-\beta i}{2} & \text{, if } k = 1 \text{ and } d = 0 \\ 0 & \text{, otherwise} \end{cases} \tag{2.23}$$

where α and β as defined in (2.20) and (2.21) and, consequently,

$$a_{-(p+1)} = \frac{\alpha + \beta i}{2} = \left(\frac{z_0}{2} + \frac{\dot{z}_0 i}{2\omega}\right) e^{i\omega t_0} \tag{2.24}$$

$$a_{p+1} = \frac{\alpha - \beta i}{2} = \left(\frac{z_0}{2} - \frac{\dot{z}_0 i}{2\omega}\right) e^{-i\omega t_0} \tag{2.25}$$

Now, we plug $\mathbf{a}$ in $\mathbf{B} \cdot \tilde{\mathbf{a}} = \mathbf{c}$. For $1 \le j \le s$, we have

$$\mathbf{B}_{j,.} \cdot \mathbf{a} = \underbrace{b_{j,-(p+1)}}_{=0} a_{-(p+1)} + \underbrace{b_{j,p+1}}_{=0} a_{p+1} = 0 \tag{2.26}$$

and for $j = n-1, n$ we have

$$\begin{aligned}\mathbf{B}_{n-1,.} \cdot \mathbf{a} &= b_{n-1,-(p+1)} a_{-(p+1)} + b_{n-1,p+1} a_{p+1} \\ &= e^{-i\omega t_0}\left(\frac{z_0}{2} + \frac{\dot{z}_0 i}{2\omega}\right) e^{i\omega t_0} + e^{i\omega t_0}\left(\frac{z_0}{2} - \frac{\dot{z}_0 i}{2\omega}\right) e^{-i\omega t_0} \\ &= \frac{z_0}{2} + \frac{\dot{z}_0 i}{2\omega} + \frac{z_0}{2} - \frac{\dot{z}_0 i}{2\omega} = z_0\end{aligned} \tag{2.27}$$

$$\begin{aligned}\mathbf{B}_{n,.} \cdot \mathbf{a} &= b_{n,-(p+1)} a_{-(p+1)} + b_{n,p+1} a_{p+1} \\ &= -i\omega e^{-i\omega t_0}\left(\frac{z_0}{2} + \frac{\dot{z}_0 i}{2\omega}\right) e^{i\omega t_0} + i\omega e^{i\omega t_0}\left(\frac{z_0}{2} - \frac{\dot{z}_0 i}{2\omega}\right) e^{-i\omega t_0} \\ &= -i\omega\frac{z_0}{2} + \frac{\dot{z}_0}{2} + i\omega\frac{z_0}{2} + \frac{\dot{z}_0}{2} = \dot{z}_0\end{aligned} \tag{2.28}$$

i.e. $\mathbf{a}$ is a solution of $\mathbf{B} \cdot \tilde{\mathbf{a}} = \mathbf{c}$, from the regularity of $\mathbf{B}$ it follows uniqueness of this solution. Hence we've proven the theoretical exactness, a numerical proof is performed in section 4.3.

2.3 General case with perturbation

Now we go back to the general case described in section 1.1 (for $d = 1$, again), and we try to apply the method of envelopes, i.e. we use collocation conditions (2.10-11) on this problem. It can be helpful to rewrite (2.10) in the form

$$\begin{aligned}&\sum_{k=-m}^{m} e^{ik\omega\tau_j} \cdot \left(-\omega^2\left(k^2-1\right) \cdot \mu_k(\tau_j) + 2ik\omega\dot{\mu}_k(\tau_j) + \ddot{\mu}_k(\tau_j)\right) \\ &\qquad = g\left(\tau_j, \sum_{k=-m}^{m} e^{ik\omega\tau_j} \cdot \mu_k(\tau_j)\right)\end{aligned} \tag{2.29}$$

Note that the left hand side is the same as in (2.13), hence it can be written as $\mathbf{B} \cdot \mathbf{a}$, with $\mathbf{a}$ and $\mathbf{B}$ as defined for (2.14).

At this point there are two possibilities:

1. If g is a function that does not depend on the state, i.e. $g(t,z) = g(t)$, then the right hand side can be written as $g(\tau_j)$ and does not depend on $\mathbf{a}$. Hence

$$\mathbf{a} = \mathbf{B}^{-1}\tilde{\mathbf{c}}, \text{ with } \tilde{\mathbf{c}} = \begin{pmatrix} g(\tau_1) \\ \vdots \\ g(\tau_s) \\ z_0 \\ \dot{z}_0 \end{pmatrix} \tag{2.30}$$

   See chapter 4 for an application.

2. otherwise one cannot derive a "nice" system as for the previous case and we must apply some iterative method like the *Newton's method*, described in the following section or use the one step collocation method presented in section 2.5.
   See chapter 5 for an application.

2.4 The Newton method for perturbed oscillations

In this section we look for an approximation for the second case described in the previous section.

First of all, assume the existence of functions $\{g_j\}$ s.t.

$$g_j(\mathbf{a}) = g\left(\tau_j, \sum_{k=-m}^{m} e^{ik\omega\tau_j} \cdot \mu_k(\tau_j)\right), \text{for all } j = 1, \ldots, s \tag{2.31}$$

then, the system can be written as

$$\mathbf{B} \cdot \mathbf{a} - \begin{pmatrix} g_1(\mathbf{a}) \\ \vdots \\ g_s(\mathbf{a}) \\ z_0 \\ \dot{z}_0 \end{pmatrix} = 0 \tag{2.32}$$

with $s = (2m+1)(p+1) - 2$.
Let us denote the left hand side of (2.32) by $\mathbf{G}(\mathbf{a})$, and using the Newton method, find a *root* of $\mathbf{G}$, i.e. a solution of (2.32).

The Newton method is the following (cfr. [4] at pages 235-236)

1. Choose an arbitrary start-point $\mathbf{a}^{(0)} \in \mathbb{R}^{(2m+1)(p+1)}$.
2. Define recursively

$$\mathbf{a}^{(l+1)} := \mathbf{a}^{(l)} - \left(D\mathbf{G}\left(\mathbf{a}^{(l)}\right)\right)^{-1} \cdot \mathbf{G}\left(\mathbf{a}^{(l)}\right) \tag{2.33}$$

   where

$$D\mathbf{G}\left(\mathbf{a}^{(l)}\right) = \mathbf{B} - \begin{pmatrix} \frac{\partial g_1}{\partial a_1}\left(\mathbf{a}^{(l)}\right) & \ldots & \frac{\partial g_1}{\partial a_{s+2}}\left(\mathbf{a}^{(l)}\right) \\ \vdots & \vdots & \vdots \\ \frac{\partial g_s}{\partial a_1}\left(\mathbf{a}^{(l)}\right) & \ldots & \frac{\partial g_s}{\partial a_{s+2}}\left(\mathbf{a}^{(l)}\right) \\ 0 & \ldots & 0 \\ 0 & \ldots & 0 \end{pmatrix} \tag{2.34}$$

3. Stop when some l' with $\mathbf{a}^{(l')} = \mathbf{a}^{(l'-1)}$ (or some previously fixed l_{max}) is reached, and set $\mathbf{a} = \mathbf{a}^{(l')}$

Remarks:

1. $\mathbf{G}(\cdot)$ and $D\mathbf{G}(\cdot)$ can be computed using the MATLAB function `HelpFunction` (see appendix A.2).

2. In order to make the computation more efficient one can use $D\mathbf{G}\left(\mathbf{a}^{(0)}\right)$ instead of $D\mathbf{G}\left(\mathbf{a}^{(l)}\right)$, this increases the error, but it reduces significantly the computation time. This variation of Newton method is often (see [4] p.235 and [1] p.266) called *simplified Newton method*.

2.5 One step collocation method

In this section we present the *one step form* of collocation method (cfr. pp. 144-147 in [4], section II.2.1 in [3] and section 6.3 in [1]). The first part is dedicated to the presentation of the one step approach and the second one to the explicit computation of the approximation.

2.5.1 The procedure

Let us consider the ODE (1.1-2) in the following form

$$\ddot{z}(t) = -\omega^2 \cdot z(t) + g(t, z(t)) =: f(t, z(t)) \tag{2.35}$$

with initial conditions

$$z(t_0) = z_0 \text{ and } \dot{z}(t_0) = \dot{z}_0 \tag{2.36}$$

and let $V^{m,p}$ be the set of truncated modulated Fourier expansions for some chosen parameters m and p, and let $s = (2m+1)(p+1) - 2$ be the dimension of the space $W^{m,p} := \left\{ \frac{d^2}{dt^2} v(t) | v \in V^{m,p} \right\}$.

The main idea of this method is to interpolate f into the space $W^{m,p}$ and then to integrate twice the resulting interpolation for getting the approximative solution function z_h.

First of all, we rewrite the collocation's nodes in interval $[t_0, t_1]$ ($t_0 \leq \tau_1 < \ldots < \tau_s \leq t_1$) as

$$\tau_i = t_0 + c_i \cdot h \tag{2.37}$$

where $h = t_1 - t_0$ and $0 \leq c_1 < \ldots < c_s \leq 1$ are the *interpolation nodes*.

For $j = 1, \dots, s$, let us define some auxiliary functions $H_j(\cdot)$ s.t.

$$H_j(c) \in \left\{ \frac{d^2}{dt^2} v(c) | v(\cdot) \in V^{m,p} \right\} \tag{2.38}$$

$$H_j(c_i) = \begin{cases} 1, \text{ if } i = j \\ 0, \text{ otherwise} \end{cases} \tag{2.39}$$

Using these functions we can interpolate f in $W^{m,p}$ and write

$$\ddot{z}_h(t_0 + c \cdot h) = \sum_{j=1}^{s} k_j \cdot H_j(c) \tag{2.40}$$

where k_j are the *increments* and are defined by

$$k_j = f(t_0 + c_j \cdot h, z_h(t_0 + c_j \cdot h)) \tag{2.41}$$

By integrating we get

$$\begin{aligned} \dot{z}_h(t_0 + c \cdot h) &= \dot{z}_h(t_0) + \int_{t_0}^{t_0 + c \cdot h} \ddot{z}_h(\tau) d\tau \\ &= \dot{z}_0 + h \cdot \sum_{j=1}^{s} k_j \cdot \int_0^c H_j(\xi) d\xi \\ &= \dot{z}_0 + h \cdot \sum_{j=1}^{s} k_j \cdot (\mathbf{H}_i(c) - \mathbf{H}_i(0)) \end{aligned}$$

where $\mathbf{H}_i$ is the primitive of H_i. By another integration we get

$$\begin{aligned} z_h(t_0 + c \cdot h) &= z(t_0) + \int_{t_0}^{t_0 + c \cdot h} \dot{z}_h(\tau) d\tau \\ &= z_0 + h \cdot \int_0^c (\dot{z}_0 + h \cdot \sum_{j=1}^{s} k_j \cdot (\mathbf{H}_j(\xi) - \mathbf{H}_j(0))) d\xi \\ &= z_0 + (h\dot{z}_0 - h^2 \cdot \sum_{j=1}^{s} k_j \cdot \mathbf{H}_j(0)) \cdot c + h^2 \cdot \sum_{j=1}^{s} k_j \cdot \int_0^c \mathbf{H}_i(\xi) d\xi \end{aligned}$$

At this point by setting

$$a_{ji} = \int_0^{c_i} \mathbf{H}_j(\xi) d\xi \tag{2.42}$$

$$b_j = \int_0^1 \mathbf{H}_j(\xi) d\xi \tag{2.43}$$

we can rewrite the increments as

$$k_i = f(t_0 + c_i \cdot h, z_0 + (h\dot{z}_0 - h^2 \cdot \sum_{j=1}^{s} k_j \cdot \mathbf{H}_j(0)) \cdot c_i + h^2 \cdot \sum_{j=1}^{s} k_j \cdot a_{ji}) \tag{2.44}$$

and define the approximative solution at time t_1 as

$$z_h(t_1) = z_0 + h\dot{z}_0 - h^2 \cdot \sum_{j=1}^{s} k_j \cdot \mathbf{H}_j(0) + h^2 \cdot \sum_{j=1}^{s} k_j \cdot b_j \tag{2.45}$$

2.5.2 Explicit computation

Let us propose a practical way for computing an approximative solution using this method. Let the interpolation nodes $\{c_i\}_{1\leq i\leq s}$ be fixed (e.g. choose them to be either the equidistant or the Gauss' nodes in $[0,1]$.

First of all we need to find the auxiliary functions $H_j(\cdot)$ for $j = 1,\dots,s$ according to (2.38-39). Condition (2.38) is equivalent to say that there exist coefficients $\left\{h^j_{k,d}\right\}_{-m\leq k\leq m, 0\leq d\leq p_k}$ (with $p_k = p-2$ if $k = 0$ and $p_k = p$ otherwise) such that $H_j(\cdot)$ can be written in form

$$H_j(c) = \sum_{d=0}^{p-2} h^j_{0,d} \cdot c^d + \sum_{k=-m,k\neq 0}^{m} \sum_{d=0}^{p} h^j_{k,d} \cdot c^d e^{ik\omega c} = \left\langle \mathbf{h}^j, \tilde{\mathbf{x}}(c) \right\rangle \tag{2.46}$$

where $\mathbf{h}^j$ is a vector containing the coefficients $h^j_{k,d}$ and $\mathbf{x}(c)$ is defined to be the same as vector $\mathbf{x}(c)$ defined in section 2.2.1 but without the entries corresponding to 1 and c.

Using this notation we can rewrite condition (2.39) for every j as

$$\underbrace{\begin{pmatrix} \mathbf{x}(c_1)^T \\ \vdots \\ \mathbf{x}(c_s)^T \end{pmatrix}}_{=:\mathbf{C}} \cdot \mathbf{h}^j = \begin{pmatrix} 0 \\ \vdots \\ 1 \\ \vdots \\ 0 \end{pmatrix} \tag{2.47}$$

where the 1 of the vector in RHS corresponds to the j^{th} row. By defining $\mathbf{h} = \left(\mathbf{h}^1, \dots, \mathbf{h}^s\right)$ we can rewrite (2.39) as

$$\mathbf{C} \cdot \mathbf{h} = I_s \tag{2.48}$$

and consequently we get

$$\mathbf{h} = \mathbf{C}^{-1} \tag{2.49}$$

i.e. the j^{th} column of the inverse of $\mathbf{C}$ contains the coefficients $\left\{h^j_{k,d}\right\}$ of $H_j(\cdot)$.

Remark: See function `Cmat` in appendix A.3 for an application using MATLAB.

At this point we may also find an explicit formula for finding directly the coefficients $\mathbf{H}_j(0)$, a_{ij} and b_j in order to save computation time when we'll go to the machine.

Note that because of the linearity of integrals, the primitive of $H_j(\cdot)$ can be found by computing the scalar vector of $\mathbf{h}^j$ with a vector (say $\tilde{\mathbf{x}}(\cdot)$) whose entries are the primitives of $\mathbf{x}(\cdot)$'s components. i.e. $\tilde{\mathbf{x}}(c) = \{\tilde{x}_{k,d}(c)\}$ where

$$k = 0 \to \tilde{x}_{k,d}(c) = \int c^d dc = \frac{c^{d+1}}{d+1} \tag{2.50}$$

$$\begin{aligned} k \neq 0 \to \tilde{x}_{k,d}(c) &= \int c^d e^{ik\omega c} dc \\ &= \frac{c^d e^{ik\omega c}}{ik\omega} - \frac{d}{ik\omega} \int c^{d-1} e^{ik\omega c} dc \\ &= \dots \\ &= \sum_{l=0}^{d} \frac{(-1)^l d!}{(ik\omega)^{l+1}(d-l)!} \cdot c^{d-l} e^{ik\omega c} \end{aligned} \tag{2.51}$$

Therefore, by setting $\mathbf{H}(0) := \begin{pmatrix} \mathbf{H}_1(0) \\ \vdots \\ \mathbf{H}_s(0) \end{pmatrix}$ we can compute the $\mathbf{H}_j(0)$ as

$$\mathbf{H}(0) = \mathbf{h}^T \cdot \tilde{\mathbf{x}}(0) \tag{2.52}$$

Remark: The entries of $\tilde{\mathbf{x}}(0)$ are

$$\begin{aligned} k = 0 &\to \tilde{x}_{k,d}(0) = 0 \\ k \neq 0 &\to \tilde{x}_{k,d}(0) = \frac{(-1)^d d!}{(ik\omega)^{d+1}} \end{aligned}$$

In order to compute a_{ij} and b_j according to (2.42-43) we need to compute the primitive of $\mathbf{H}_j(\cdot)$ too. As before we use linearity that allows us to write the primitive of $\mathbf{H}_j(\cdot)$ as the scalar product of $\mathbf{h}^j$ with a vector (say $\tilde{\tilde{\mathbf{x}}}(\cdot)$) whose components are the primitives of $\tilde{\mathbf{x}}(\cdot)$'s entries. Using the results of (2.50-51) we can compute the components $\{\tilde{\tilde{x}}_{k,d}(c)\}$ of $\tilde{\tilde{\mathbf{x}}}(c)$ as

$$k = 0 \to \tilde{\tilde{x}}_{k,d}(c) = \int \frac{c^{d+1}}{d+1} dc = \frac{c^{d+2}}{(d+1)(d+2)} \tag{2.53}$$

$$
\begin{aligned}
k \neq 0 \to \tilde{\tilde{x}}_{k,d}(c) &= \int \sum_{l=0}^{d} \frac{(-1)^l d!}{(ik\omega)^{l+1}(d-l)!} \cdot c^{d-l} e^{ik\omega c} dc \\
&= \sum_{l=0}^{d} \frac{(-1)^l d!}{(ik\omega)^{l+1}(d-l)!} \cdot \int c^{d-l} e^{ik\omega c} dc \\
&= \sum_{l_1=0}^{d} \sum_{l_2=0}^{d-l_1} \frac{(-1)^{l_1+l_2} d!}{(ik\omega)^{l_1+l_2+2}(d-l_1-l_2)} \cdot c^{d-l_1-l_2} e^{ik\omega c}
\end{aligned} \tag{2.54}
$$

Remark: See function `yvec` in appendix A.4 for an application using MATLAB.

Using this notation we get

$$
\int_0^c \mathbf{H}_j(\xi) d\xi = \left\langle \mathbf{h}^j, \tilde{\tilde{\mathbf{x}}}(c) - \tilde{\tilde{\mathbf{x}}}(0) \right\rangle \tag{2.55}
$$

Therefore by setting $\mathbf{a} = \left(a_{ij}\right)_{1 \leq i,j \leq s}$ and $\mathbf{b} = \left(b_j\right)_{1 \leq j \leq s}$ we can compute the coefficients a_{ji} and b_j as follows

$$
\mathbf{a} = \mathbf{h}^T \cdot \left(\tilde{\tilde{\mathbf{x}}}(c_1) - \tilde{\tilde{\mathbf{x}}}(0), \ldots, \tilde{\tilde{\mathbf{x}}}(c_s) - \tilde{\tilde{\mathbf{x}}}(0) \right) \tag{2.56}
$$

$$
\mathbf{b} = \mathbf{h}^T \cdot \left(\tilde{\tilde{\mathbf{x}}}(1) - \tilde{\tilde{\mathbf{x}}}(0) \right) \tag{2.57}
$$

Now the approximative solution at time t_1 can be found by solving the non-linear system of equations given by (2.44) with the recently computed coefficients $\mathbf{H}_j(0)$ and a_{ji} and by plugging the resulting increments with coefficients into equation (2.45).

Remark: The system of equations for the increments is not linear, hence one has to solve it using some iterative method as the Newton method presented in the previous section or using the MATLAB function `fsolve`.

Chapter 3

MATLAB implementations

In this chapter we present the implementations in MATLAB ([6]) of the method of envelopes described in the previous chapter for finding a approximation of the solution at time t_1 of a second order ordinary differential equation of the form

$$\ddot{z}(t) + \omega^2 \cdot z(t) = g(t, z(t)) \tag{3.1}$$

with initial conditions

$$z(t_0) = z_0 \tag{3.2}$$
$$\dot{z}(t_0) = \dot{z}_0 \tag{3.3}$$

in the time-interval $[t_0, t_1]$.

The first function corresponds to the method developed *ad hoc* for the first case described in section 2.3, i.e. when the perturbation g depends on the time only ($g(t, z(t)) \equiv g(t)$) , while the other two are the functions corresponding to the methods that use either the Newton iterations (see section 2.4) or the one step form (see section 2.5) and that can be applied to any ODE of the type (3.1-3). In the following we're calling the first two *matrix form* methods and the last one *one step form* method.

3.1 The collocation matrix

First of all we write a function `Bmat` that defines matrix **B** for given t_0, ω, m, p and τ according to (2.15-17)

```
1 function [ B ] = Bmat( t0, omega, m, p, tau )
2 %Bmat compute the matrix B using (2.15-17)
3 %   t0 = initial time
4 %   omega = parameter of the ODE
```

```
5  %   m = truncation parameter
6  %   p = maximal degree of polynomials
7  %   tau = (2m+1)(p+1)-2 vector containing the collocation's
8  %         points
9
10 % Initialization
11 n = (2*m+1)*(p+1);
12 B = zeros(n,n);
13 s = n-2;
14
15 % Fill the matrix according to (2.15-17)
16 l = m*(p+1)+1;
17 for k=-m:m
18     % second collocation condition (2.15)
19     for j= 1:s
20         % d=0
21         B(j,k*(p+1)+l) = -exp(1i*k*omega*tau(j))*omega^2*(k^2-1);
22         % d=1
23         HP = -omega^2*(k^2-1)*tau(j)+2i*k*omega;
24         B(j,k*(p+1)+1+l) = exp(1i*k*omega*tau(j))*HP;
25         % d=>2
26         for d=2:p
27             HP1 = -omega^2*(k^2-1)*(tau(j))^d;
28             HP2 = 2i*d*k*omega*(tau(j))^(d-1)+d*(d-1)*(tau(j))^(d-2);
29             HP = HP1+HP2;
30             B(j,k*(p+1)+d+l) = exp(1i*k*omega*tau(j))*HP;
31         end
32     end
33
34     % first collocation condition
35     for d=0:p
36         B(n-1,k*(p+1)+d+l) = exp(1i*k*omega*t0)*t0^d;    %(2.16)
37     end
38     B(n,k*(p+1)+l) = exp(1i*k*omega*t0)*1i*k*omega;
39     for d=1:p                                            %(2.17)
40         HP = 1i*k*omega*t0^d+d*t0^(d-1);
41         B(n,k*(p+1)+d+l) = exp(1i*k*omega*t0)*HP;
42     end
43 end
44
45 end
```

3.2 Collocation method for state-independent perturbation function

Then, we write a function *ad hoc* for the first case described in section 2.3. We define a function `FindAppr` that, for given time interval ($T = [t_0, t_1]$), ODE parameter (ω), initial values ($Z_0 = (z_0, \dot{z}_0)'$), truncation and polynomial parameters (m and p, respectively) and for a perturbation function g that only depends on the time variable, computes an approximative solution of the

corresponding ODE ($\ddot{z} + \omega^2 z = g(t)$) at time t_1 according to (2.19) and using either equidistant or Gausses points as collocation's points (depending on the last input argument).

```
function [ zAppr ] = FindAppr( T, omega, g, Z0, m, p, t)
%FindAppr gives an approximative solution of in the ODE where the
%perturbation depends on the time only for the parameters:
%   T = chosen time interval for the approximation
%   omega = the chosen parameter
%   g = the perturbation function
%   Z0 = initial conditions (i.e. z(t0) and z'(t0))
%   m = truncation parameter
%   p = maximal polynomial's degree
%   t = 0 if want equidistant points, 1, for Gauss points

s = (2*m+1)*(p+1)-2;

% Define collocation points
if t==0
    %(equidistant)
    h = (T(2)-T(1))/(s-1);
    tau = [T(1):h:T(2)];
else
    %(Gauss)
    tau = GaussQuad(s-1);
    tau = (T(2)-T(1))/2*tau+(T(1)+T(2))/2;
end

% Compute B;
B = Bmat(T(1), omega, m, p, tau);

% Compute c;
c = zeros(s+2,1);
for j=1:s
    c(j)=g(tau(j));
end

c(s+1) = Z0(1);
c(s+2) = Z0(2);

% Find coefficients according to (2.30)
a = B\c;

% Compute the approximation at t1:
x = xvec(m,p,T(2),omega);
zAppr = a.'*x ;
zAppr = real(zAppr);

end
```

Remarks:

1. Function `GaussQuad` called at line `21` is defined in [5] at page 115.
2. Function `xvec` called at line `41` is defined in appendix A.1.

3.3 General matrix form collocation method

Now we want to write a function that computes a numerical approximation (at some chosen time) for any given ODE of the form (3.1-3) using the *matrix approach*.

We define function `FindAppr2` that, for given time interval ($T = [t_0, t_1]$), ODE parameter (ω), initial values ($Z_0 = (z_0, \dot{z}_0)'$), truncation and polynomial parameters (m and p, respectively) and for a perturbation function g that can depend on state variable too, computes an approximative solution of the corresponding ODE ($\ddot{z} + \omega^2 z = g(t, z)$) at time t_1 using either equidistant or Gausses points as collocation's points (depending on the last input argument) through a Newton iteration (see section 2.4 for a description).

```
1  function[ zAppr ] = FindAppr2(T, omega, g, Z0, m, p, t)
2  %FindAppr2 computes a numerical approximation of the ODE
3  %y''+omega^2*y = g(t,y) using the method of envelopes.
4  %   T = time-interval
5  %   omega = ODE parameter
6  %   g = perturbation function
7  %   m = truncation parameter
8  %   p = polynomial's degree parameter
9  %   Y0 = initial conditions (i.e. y(t0) and y'(t0))
10 %   t = nodes parameter (i.e. 0 for equidistant nodes,
11 %                        1 for Gauss nodes)
12
13 % Define initial and final time
14 t0 = T(1);
15 t1 = T(2);
16
17 % Define the initial conditions
18 z0 = Z0(1);
19 z1 = Z0(2);
20
21 % Find s
22 s = (2*m+1)*(p+1)-2;
23
24 % Find matrix B for equidistant (t=0)
25 if t==0
26     h = (t1-t0)/(s-1);
27     tau = t0:h:t1;
28 % or Gauss points (t=1)
29 else
30     tau = GaussQuad(s-1);
31     tau = (t1-t0)/2*tau+(t0+t1)/2;
```

```
32 end
33
34 B = Bmat(t0, omega, m, p, tau);
35
36 % Define Start vector
37 a_old=zeros(s+2,1);
38 a_old((m-1)*(p+1)+1)=(z0/2+1i*z1/(2*omega))*exp(1i*omega*t0);
39 a_old((m+1)*(p+1)+1)=(z0/2-1i*z1/(2*omega))*exp(-1i*omega*t0);
40 [G, DG0] = HelpFunction(a_old, B, g, omega^2, m, p, tau, Z0, 0 );
41
42 % Use (simplified) Newton-Method
43 k=0;
44 kMax = 4000; % Fixed maximal quantity of iterations
45 Shot2 = 0; % Is a second shot required?
46
47 % If the start vector gives a highly ill conditioned matrix,
48 % then the obtained approximation will be very bad, therefore in
49 % order to save computation time we do not compute it.
50 if rcond(DG0) < 1E-16
51    Shot2 = 1;
52 else
53    while k < kMax
54         a_new = a_old-DG0\G;  % according to (2.33)
55         if isnan(a_new)
56             Shot2 = 1;
57             break
58         end
59         if norm(a_new-a_old) < 1E-11
60             a_old = a_new;
61             break
62         end
63         a_old = a_new;
64         k=k+1;
65         [G, DG] = HelpFunction(a_old, B, g, omega, m, p, tau, Z0, 1 );
66     end
67 end
68
69 % a second possibility for divergent iterations
70 if Shot2 == 1
71     % Define a new Start vector
72     a_old=zeros(s+2,1);
73     [G, DG0] = HelpFunction(a_old, B, g, omega^2, m, p, tau, Z0, 0 );
74     % As before
75     if rcond(DG0) < 1E-16
76         zAppr = Inf;
77         return
78     end
79     k=0;
80     while k < kMax % Fixed maximal quantity of iterations
81         a_new = a_old-DG0\G;  % according to (2.33)
82         if isnan(a_new)
83             zAppr = NaN;
84             return
85         end
```

```
86          if norm(a_new-a_old) < 1E-11
87              a_old = a_new;
88              break
89          end
90          a_old = a_new;
91          k=k+1;
92          [G, DG] = HelpFunction(a_old, B, g, omega, m, p, tau, Z0, 1 );
93      end
94  end
95
96  a = a_old;
97
98  % Compute approximation at time t1
99  x = xvec(m, p ,t1, omega);
100 zAppr = x.'*a;
101 zAppr = real(zAppr);
102
103 end
```

Remarks:

1. Function `GaussQuad` called at line `30` is defined in [5] at page 115.
2. Note that the first "Start vector" (lines `37-39`) used for Newton iterations corresponds to the vector containing the coefficients of the linear case's solution.
3. Function `HelpFunction` called at line `40` is defined in appendix A.2.
4. The motivation for variable `Shot2` defined at line `45` is the following: the convergence of Newton iterations is connected with the choice of the start vector. Sometimes changing the start vector one can make convergent a divergent iteration. For this reason we give a "second shot" to divergent iterations.
5. Function `xvec` called at line `99` is defined in appendix A.1.

3.4 One step collocation method

Finally we write a function, called `osm`, that computes a numerical approximation of the ODE ($\ddot{z} + \omega^2 z = g(t,z)$) using the one step method presented in section 2.5. With given time interval ($T = [t_0, t_1]$), ODE parameter (ω), initial values ($Z_0 = (z_0, \dot{z}_0)'$), truncation and polynomial parameters (m and p, respectively) and with an arbitrary perturbation function g, using either equidistant or Gausses points as collocation's points (depending on the last input argument).

```
1  function [ zAppr ] = osm(  T, omega, g, Z0, m, p, t )
2  % osm finds an approximative solution of ODE
```

```
% z''+omega^2 z = g(t,z) at time T(2) using the one
% step form of the collocation method with truncated
% modulated Fourier expansions as trial set.
%   T = time interval
%   omega = ODE parameter
%   g = perturbation function
%   Z0 = initial conditions
%   m = truncation parameter
%   p = polynomial degree parameter
%   t = equidistant (t=0) or Gauss nodes?

% initialization
s = (2*m+1)*(p+1)-2;
s1 = m*(p+1)+1;
s2 = m*(p+1)-1;
h = T(2)-T(1);

% Define collocation points
if t==0
    %(equidistant)
   c = 0:(1/(s-1)):1;
else
    %(Gauss)
    c = GaussQuad(s-1);
    c = 1/2*c+1/2;
end

% Find HelpFunctions according to (2.38-39)
C = Cmat( omega, m, p, c );
H = inv(C);                          %(2.49)

% Find H0
y = zeros(s,1);
for k = 1:m
    for d = 0:p
    y(-k*(p+1)+d+s1) = (-1)^d*factorial(d)/(-1i*k*omega)^(d+1);
    y(k*(p+1)+d+s2) = (-1)^d*factorial(d)/(1i*k*omega)^(d+1);
    end
end
H0 = H.'*y;                          %(2.52)

% Find a_ij and b_i
y0 = yvec( omega, m, p, 0);
Y = zeros(s);
for j=1:s
    yj = yvec( omega, m, p, c(j));
    Y(:,j) = yj-y0;
end
a = H.'*Y;                           %(2.56)
y1 = yvec( omega, m, p, 1);
b = H.'*(y1-y0);                     %(2.57)

% Find increments according to (2.44)
K = fsolve(@(K) HelpIncrements(T, omega, g, Z0, s, c, h, a, H0, K),ones(s,1),optimset('TolF
```

```
57
58 % Compute approximation at time t1 according to (2.45)
59 zAppr = Z0(1)+h*Z0(2)-h^2*K.'*H0+h^2*K.'*b;
60
61 end
```

Remarks:

1. Function `GaussQuad` called at line 26 is defined in [5] at page 115.
2. Function `Cmat` called at line 31 is defined in appendix A.3.
3. Function `yvec` called at line 45 is defined in appendix A.4.
4. Function `HelpIncrements` called at line 72 is defined in appendix A.5.

Chapter 4

Numerical experiments

In this chapter we test the three functions of the previous chapter on a ODE with oscillatory solutions of the first type presented in section 2.3, i.e. when the perturbation is a function of the time only. In particular we are interested in testing function `FindAppr`, which was developed ad hoc for this situation. We also try to use the other two functions and we compare the obtained results.

4.1 The problem

Take the the second order equation used as numerical example in [7] at page 486.

$$\ddot{z} + \frac{z}{\varepsilon^2} = \frac{e^{-t}}{\varepsilon^2} \text{ for } \varepsilon > 0 \tag{4.1}$$

with initial conditions

$$z(0) = 1 + \frac{1}{1+\varepsilon^2} \tag{4.2}$$

$$\dot{z}(0) = -\frac{1}{1+\varepsilon^2} \tag{4.3}$$

The solution of this ODE is given by

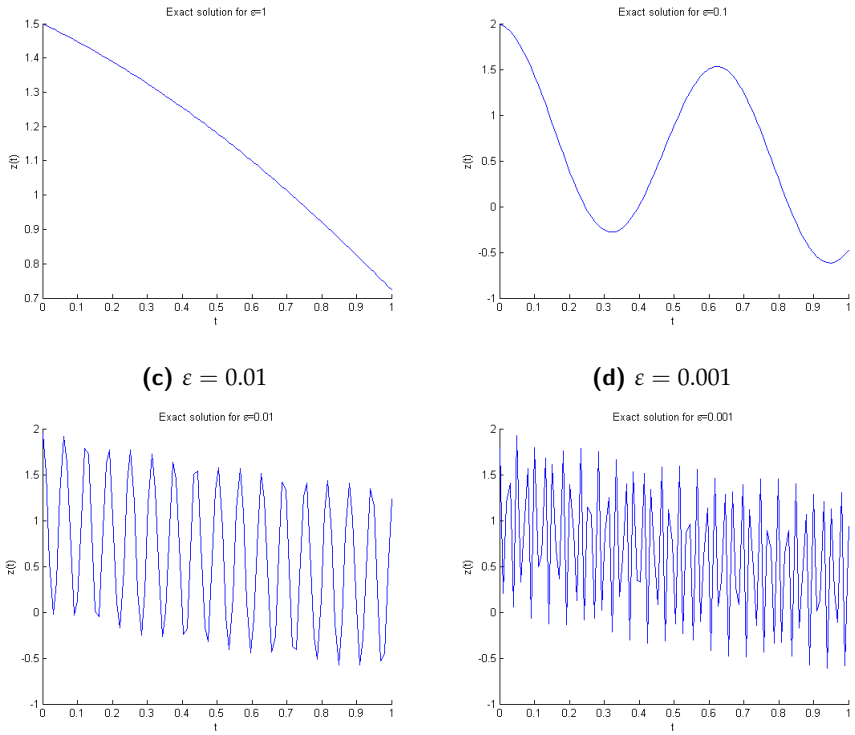
$$z(t) = \cos\left(\frac{t}{\varepsilon}\right) + \frac{e^{-t}}{1+\varepsilon^2} \tag{4.4}$$

In Figure 4.1 is shown the exact solution for some values of ε.

Now we want to find numerically an approximation $z_h(t_1)$ at time $t_1 = 1$, using the MATLAB implementations of the method of envelopes presented in the previous chapter.

Figure 4.1: Exact solution for some εs

(a) $\varepsilon = 1$ **(b)** $\varepsilon = 0.1$



(c) $\varepsilon = 0.01$ **(d)** $\varepsilon = 0.001$

4.2 Projection error

Before doing anything it may be helpful to study how good is the chosen trial space $V^{m,p}$ for this particular problem.

Consider (4.1) in the following form:

$$\ddot{z} = \underbrace{-\frac{1}{\varepsilon^2}z + \frac{e^{-t}}{\varepsilon^2}}_{=:f(t,z(t))} \tag{4.5}$$

using (4.4) f can be written as:

$$f(t) = f(t,z(t)) = -\frac{1}{\varepsilon^2}\cos\left(\frac{t}{\varepsilon}\right) + \frac{e^{-t}}{\varepsilon^2}\left(1 - \frac{1}{1+\varepsilon^2}\right) \tag{4.6}$$

The study of the projection of f in the space of the second derivatives of $V^{m,p}$, i.e. in $W^{m,p} := \left\{\frac{d^2}{dt^2}v : v \in V^{m,p}\right\}$ may give a clue about how good the approximation (at time $t_1 = 1$) given by the method of envelopes for this

problem will be, because the interpolation error of z in $V^{m,p}$ is bounded by the projection error of f in $W^{m,p}$ (see Theorem 2.2.30 in [4]).

The functions $w(t) \in W^{m,p}$ are of the form

$$w(t) = \sum_{d=0}^{p-2} r_{0,d} t^d + \sum_{k=-m,k\neq 0}^{m} e^{\frac{ikt}{\varepsilon}} \left(\sum_{d=0}^{p} r_{k,d} t^d \right) \tag{4.7}$$

and the projection of f must satisfy

$$\begin{cases} w_f(\tau_1) = f(\tau_1) \\ \vdots \\ w_f(\tau_s) = f(\tau_s) \end{cases} \tag{4.8}$$

where $0 \leq \tau_1 < \ldots < \tau_s \leq 1$ are the interpolation points and can be chosen in several ways (we'll use equidistant and Gauss nodes).

From (4.7) it's evident that $\dim W^{m,p} = (2m+1)(p+1) - 2$. Therefore for the projection of f into $W^{m,p}$ we need $s = (2m+1)(p+1) - 2$ interpolation pairs $(\tau_j, f(\tau_j))$. The coefficients of w_f, $\{r_{k,d}\}_{-m\leq k\leq m, 0\leq d\leq p}$, are computed by solving system (4.8), which in matrix form becomes $\mathbf{T} \cdot \mathbf{r} = \mathbf{f}$, where $\mathbf{f}$ is a vector containing the values $f(\tau_j)$, and $\mathbf{T} = \left(T_{j,l(k,d)} \right)$ is a matrix with entries $T_{j,l(k,d)} = \tau_j^d e^{\frac{ik\tau_j}{\varepsilon}}$ (where $l(k,d)$ is some function that for k and d gives the position in the matrix $\mathbf{T}$).

At this point the projection's error is given by $\|f - w\|_\infty$. If we consider the interval $[0,1]$, the values of these errors for $\varepsilon = 0.01$, $m = 1$ and $p = 1, \ldots 9$ are shown in Figure 4.3.

Remarks:

1. One can see that the projection error converges exponentially, in particular for Gauss' nodes.
2. However, if $\mathbf{T}$ is bad conditioned (e.g. for $p = 10$ and using equidistant nodes), the error increases (e.g. $1.5997 \cdot 10^3$). Hence, we may have problems when using the method of envelopes, due to the condition of $\mathbf{B}$.

Figure 4.2: Projections of f in W

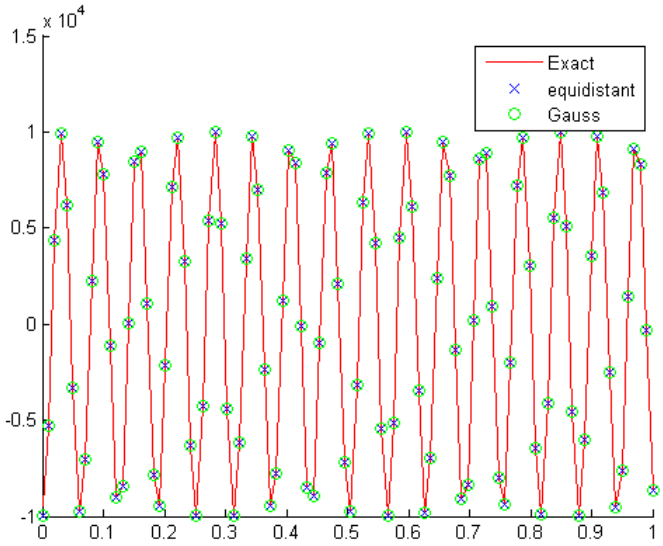
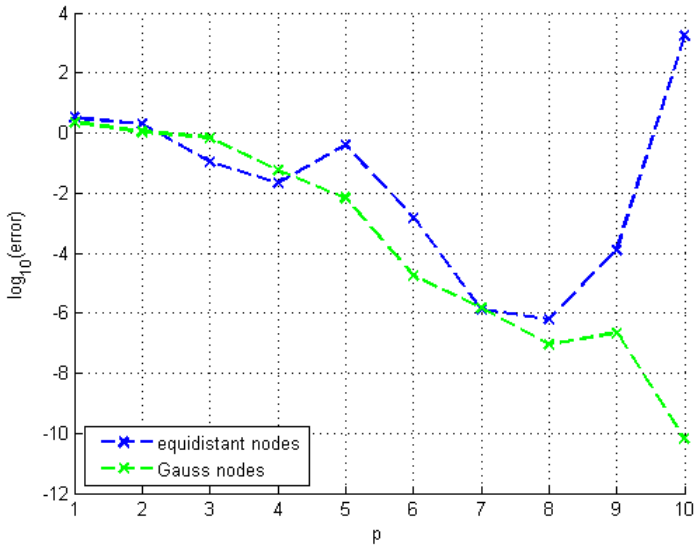


Figure 4.3: Projection errors for $m = 1$ and $\varepsilon = 10^{-2}$



4.3 The linear case

In this section we study how good the implemented methods are when applied to a linear oscillation.

First we try to solve the linear case using matrix inversion on (2.14): we

define matrix **B** using function `Bmat` for some chosen ω, m and p together with the initial values and both equidistant and Gauss' nodes in $[0,1]$ as τ to compute the coefficients.
And we compare the obtained **a** with the theoretical one described in (2.23). Looking at the maximum-norm of the difference between the obtained **a** and the theoretical one, we see that for $m = 1$, $\varepsilon = 10^{-2}$ and $p = 1,\dots,10$, it is equal to $0.1110 \cdot 10^{-15}$, that is the machine precision, when using Gauss' nodes and also when using equidistant nodes but for $p = 5,6$ (for these values the error's value is above 10^{-7}), i.e. we can conclude that in general this method finds the exact solution for the linear case.

Then we apply the functions implemented in the previous chapter using $g \equiv 0$, $t_1 = 1$, $m = 1$ and different values of p and we compare it with the exact solution given by combining (1.6) and (2.8-9).
Using function `FindAppr` with Gauss nodes we get errors equal 0 for all $p = 1,\dots,10$, using equidistant nodes we also have the most part of the errors equal to 0 and below 10^{-7} for $p = 5,6$. Therefore we can say that this function finds the exact solution for the linear oscillation.
Function `FindAppr2` gives all errors equal 0 for both Gauss and equidistant nodes and $p = 1,\dots,9$ (for $p = 10$ we got 0 when using Gauss and ∞ for equidistant ones), this is also due to the choice of the start vector for the (simplified) Newton iteration (see lines `37-39` in the code of `FindAppr2`), which corresponds to the vector containing the exact coefficients of the truncated modulated Fourier expansions representing the solution of the linear oscillation. Hence, also this function solves exactly the linear case.
The errors given by the one step method by solving the linear case are shown in Table 4.1. One can see that for small values of p the approximation is very good, but it degenerates for bigger values. The explanation of this phenomena is that the condition of the matrices (let the first one be called $DIncr$) used by `fsolve` for finding the increments become very high for large values of p.

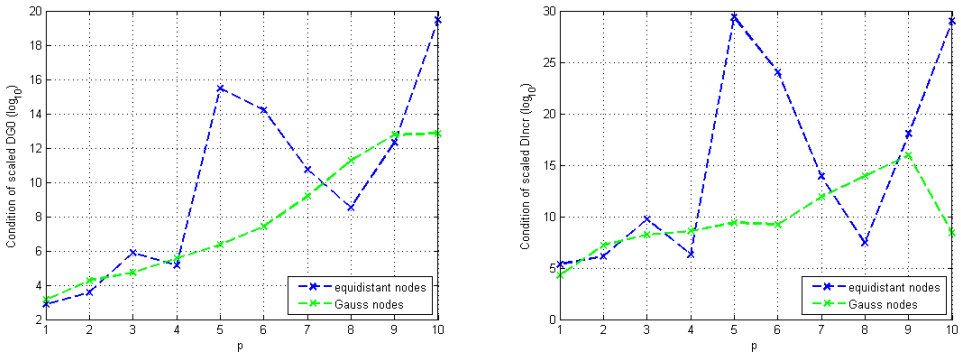
Note that also the jacobian of function **G** for the start vector of the (simplified) Newton iteration (called $DG0$) performed by `FindAppr2` can be very ill conditioned and consequently produce large errors' values. In Figure 4.4 are shown the the evolutions with respect to p of conditions of scaled $DIncr$ and $DG0$ for the linear case. In these plots one can see that the condition increases with p and it tends to be larger when using equidistant nodes. Therefore, it may be a good idea to use only small values of p in functions `FindAppr2` and `osm`.

Table 4.1: Errors using `osm` for $m = 1$ and $\varepsilon = 10^{-2}$

p	equidistant points	Gauss points
1	5.1456e-013	3.4455e-014
2	4.4075e-012	3.9344e-012
3	3.1734e-012	5.1240e-011
4	1.5210e-011	1.0740e-011
5	1.2599e+000	4.9800e-010
6	1.8331e+000	4.0749e-011
7	2.1937e+000	1.7692e+000
8	1.2070e-009	1.5886e+000
9	2.8912e-001	1.9872e+000
10	2.5665e+000	5.0346e-008

Figure 4.4: Condition of the (scaled) matrix ...

(a) $DG0$ used in `FindAppr2` **(b)** $DIncr$ used by `fsolve` in `osm`



4.4 The perturbed case and error's analysis using the ad hoc method

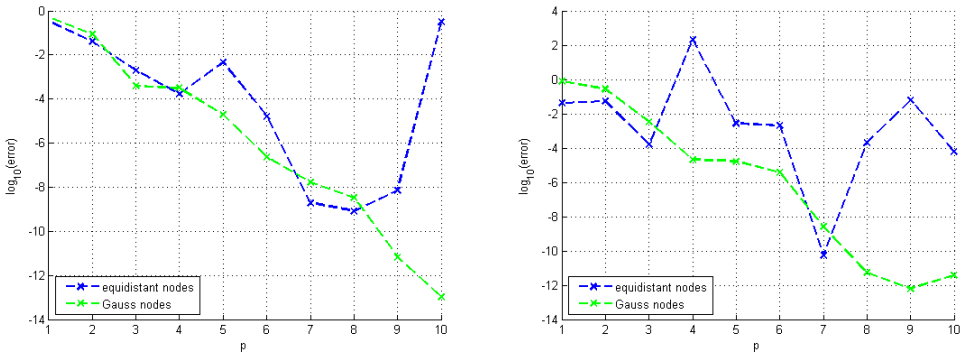
Now we look for an approximation for the perturbed case. Note, again, that we are in the first case described in section 2.3. Therefore we apply to problem (4.1-3) the *ad hoc* function `FindAppr` for different parameters' choices and we analyse the accuracy of the found approximations (with respect to the given exact solution (4.4)).

4.4.1 Error w.r.t m and p

First of all let us fix $\varepsilon = 0.01$ (i.e. $\omega = 10^2$) and $t_1 = 1$. Look at the plots of the approximation's errors for $m = 1$ and $m = 3$ and p going from 1 to 10 in Figure 4.5.

Figure 4.5: log(error) vs. p for $t_1 = 1$, $\varepsilon = 0.01$ and ...

(a) $m = 1$ **(b)** $m = 3$

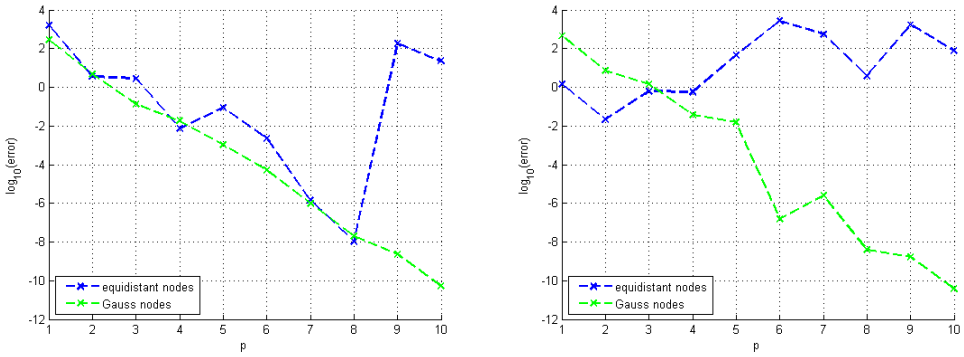


Remarks:

1. In both plots one can note exponential convergence for the errors, in particular when using Gauss' points.
2. It may also be concluded that Gauss' nodes tend to give smaller error's values.
3. In Figure 4.6 one can see that even for smaller ε analogous conclusions can be drawn. Moreover, in that case one can also note that error's values do not converge when using $m = 3$ and equidistant nodes.

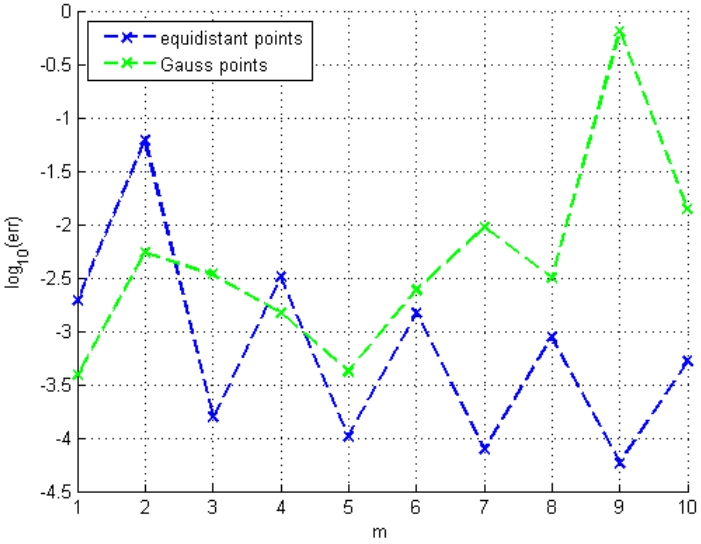
Figure 4.6: log(error) vs. p for $t_1 = 1$, $\varepsilon = 10^{-4}$ and ...

(a) $m = 1$ **(b)** $m = 3$



Now take a look at the situation in which p stays fixed and m increases (Figure 4.7). Note that in this case the errors don't seem to converge.

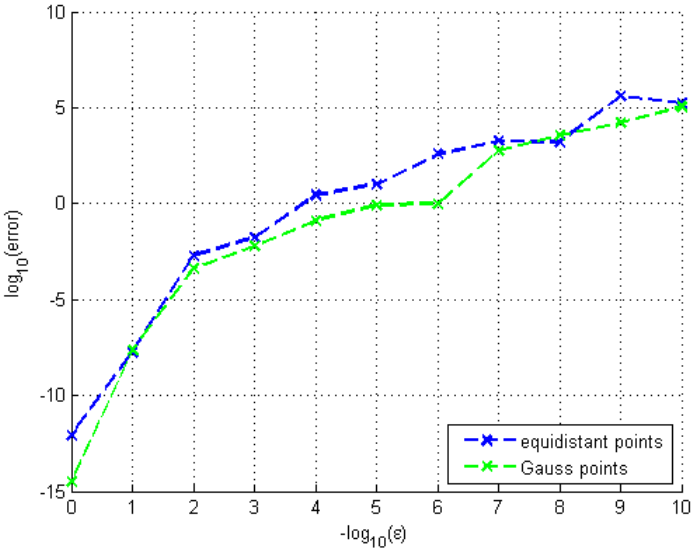
Figure 4.7: log(error) vs. m for $t_1 = 1$ and $p = 3$



4.4.2 Error w.r.t ε

In this section we analyse the errors of the approximation at time $t_1 = 1$ for $\varepsilon = 10^{-k}$ with k going from 1 to 10, keeping m and p fixed. First, take a look at Figure 4.8.

Figure 4.8: log(error) vs. -log(ε), $t_1 = 1$, $m = 1$ and $p = 3$

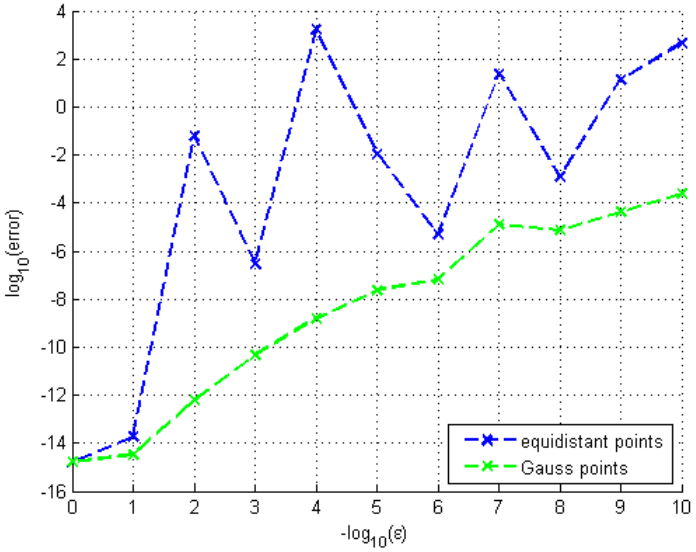


The plot highlights that

1. The error's value increases as ε decreases for both equidistant and Gauss points.
2. Using Gauss nodes gives, in general, smaller errors than using equidistant ones.

Note that the same behaviour is observed when using different values of m and p (Figure 4.9).

Figure 4.9: log(error) vs. -log(ε), $t_1 = 1$, $m = 3$ and $p = 9$



4.4.3 Error w.r.t. t_1

Finally, we analyse the relation between the errors and the parameter t_1. i.e. we compute the approximation's errors at t_1 varying from 0.5 to 10 (with step length 0.5). Consider Figure 4.10.

This plot shows that:

- In general Gauss nodes give smaller errors
- In the cases in which equidistant nodes produce better results, the difference is small.
- The difference between the results can also be considerable, in the cases where Gauss nodes give smaller errors.

The same reasoning can be made when looking at Figure 4.11, where we use different values of m, p and ε

Figure 4.10: log(error) vs. t_1, $\varepsilon = 0.01$, $m = 1$ and $p = 3$

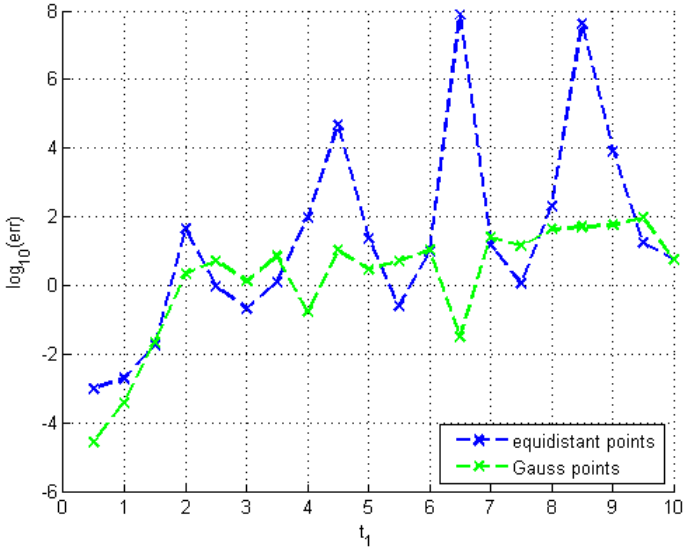
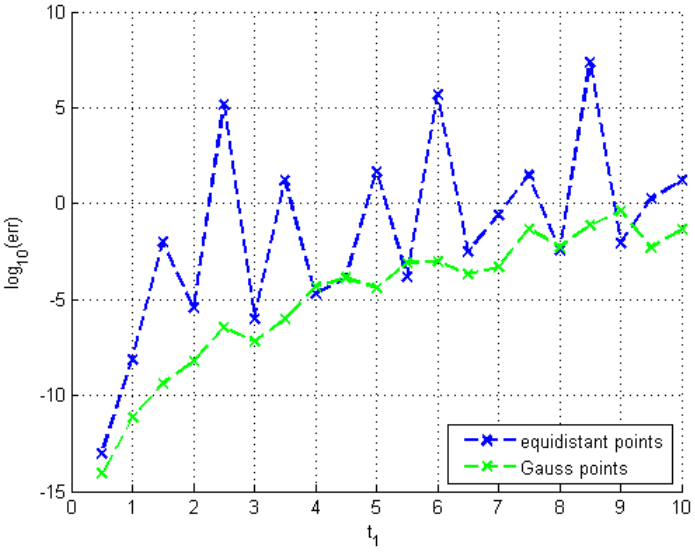


Figure 4.11: log(error) vs. t_1, $\varepsilon = 0.01$, $m = 1$ and $p = 9$



4.5 Errors analysis for the other methods

In this section we apply to problem (4.1-3) the general methods instead of the *ad hoc* one.

We begin by using the general matrix form method, i.e. we apply function

`FindAppr2`. The obtained error plots w.r.t p and for different values of m and ε are shown in Figures 4.12-13.

Figure 4.12: log(error) vs. p using function `FindAppr2` for $t_1 = 1$, $\varepsilon = 0.01$ and ...

(a) $m = 1$ **(b)** $m = 3$

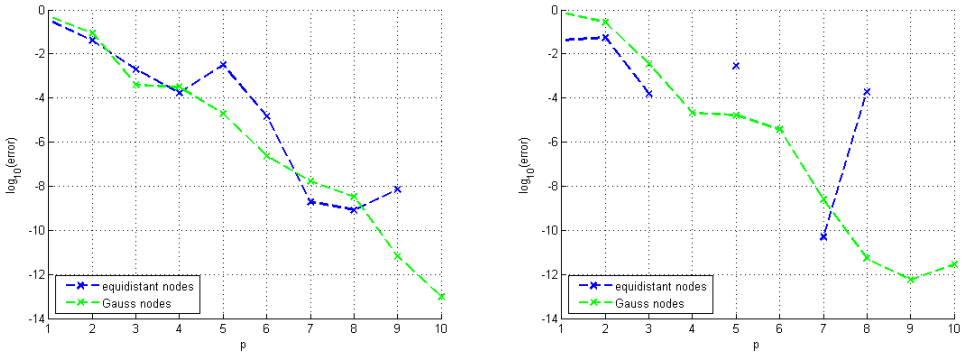
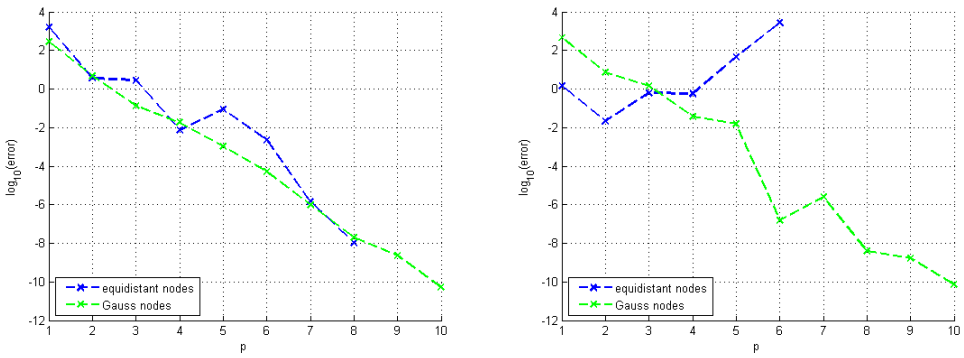


Figure 4.13: log(error) vs. p using function `FindAppr2` for $t_1 = 1$, $\varepsilon = 10^{-4}$ and ...

(a) $m = 1$ **(b)** $m = 3$



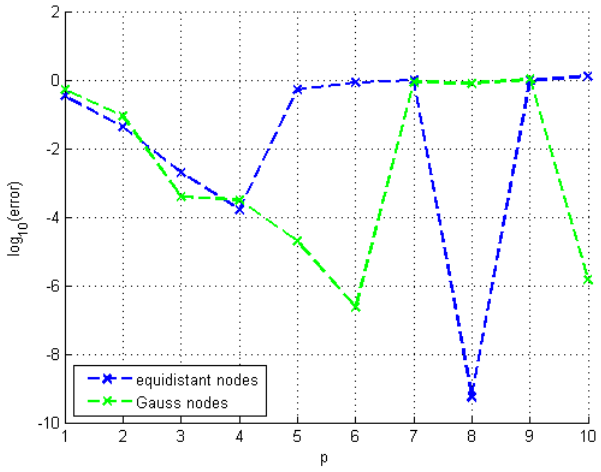
Remarks:

1. Note that for the error corresponding to the Gauss nodes there's no difference with the above plots.
2. When using equidistant nodes some error's values are not displayed. This means that for such p's the matrix $D\mathbf{G}$ is very ill conditioned and therefore this method gives a very inexact solution or no solution at all.

Finally let us apply the one step form's function `osm` to this problem and

take a look of the obtained errors (Figure 4.14).

Figure 4.14: log(error) vs. p using function `osm` for $t_1 = 1$, $m = 1$ and $\varepsilon = 10^{-2}$



We can see that for small p values the plot is very similar to the ones obtained with the previous two methods, but for higher values it degenerates and losses precision (with respect to the other methods) exactly as it did solving the linear case (see Table 4.1 and Figure 4.4) due to the condition of the used matrices.

4.6 Conclusion

From the error analysis of previous sections one can conclude that in general it is better to use collocation with Gauss' nodes than equidistant nodes. This is because the first one yields often to smaller error's values, in particular when using large t_1 and/or small ε. Even if the error produced by collocation with equidistant nodes is smaller, the difference is often a small fry. Conversely, if equidistant points are used and the error value is bigger than the one that would be produced by using Gauss' points, the obtained approximation can be considerably worse.

Moreover, the plots of the errors with respect to the p-value for the matrix form methods show that for Gauss nodes we have exponential convergence (i.e. the error's values go to zero as p increases), while this is not always true if using equidistant points. A sort of convergence for Gauss' nodes can be seen in the plots of the one step method too: in fact for $p = 1, \ldots, 6$ the behaviour of the green lines is the same as in the previous ones.

For the general methods one can also note that collocation with Gauss nodes is safer because a solution is always found. This does not hold when using equidistant nodes (see in particular Figures 4.12-13).

Another important point is the relative efficiency and precision of the methods: the ad hoc one (function `FindAppr`) is extremely faster than the others and the produced approximations are at least as good as the ones given by the others. Therefore, we suggest to use this method, whenever possible.

Chapter 5

Another numerical experiment

In this chapter we investigate the quality of the implemented methods through the application to an oscillation with state-depending perturbation.

5.1 The (new) problem

Let us consider a new ODE with oscillatory solution: the equation for the *mathematical pendulum*

$$\ddot{z} = -\lambda \sin(z) \tag{5.1}$$

where $\lambda \in \mathbb{R}$ is a (large) fixed parameter, with initial conditions

$$z(0) = z_0 \tag{5.2}$$
$$\dot{z}(0) = 0 \tag{5.3}$$

where z_0 a small *initial displacement*'s parameter.

Remark: In this case there's no known general solution, therefore we have to compute the reference solution using some other numerical method: here we use the *Störmer-Verlet* procedure and the MATLAB integrator `ode45`.

As for the previous example, we want to compute a numerical approximation $z_h(t_1)$ at some time t_1 (as before we'll use $t_1 = 1$).

5.2 The reference solution

Since no general solution for this ODE is available, we have to compute the reference solution at time t_1 using numerical methods. It is useful to call f the RHS of (5.1), i.e.

$$f(t, z) = -\lambda \cdot \sin(z) \tag{5.4}$$

A first way to find $z_{ref}(t_1)$ is to use the Störmer-Verlet procedure (cfr. [4], section 1.4.4, pp. 104-116 and [2]) i.e.

1. Divide the interval $[0, t_1]$ into $N \in \mathbb{N}$ timesteps of length $h = \frac{t_1}{N}$ and get a time-grid $\left\{0 = t^{(0)}, t^{(1)}, \ldots, t^{(N-1)}, t^{(N)} = t_1\right\}$
2. Define recursively $z_k \approx z(t^{(k)})$ according to

$$z_{k+1} = -z_{k-1} + 2z_k + h^2 f(z_k) \tag{5.5}$$

3. Set $z_{ref}(t_1) = y_N$.

Remark: Note that for the calculation of z_1 we need z_0 and z_{-1}, therefore we consider z_0 to be the initial displacement and for z_{-1} we consider a virtual point $t^{(-1)} = t^{(0)} - h$ for which it holds

$$z_1 = -z_{-1} + 2z_0 + h^2 f(z_0) \tag{5.6}$$

$$\frac{z_1 - z_{-1}}{2h} = \dot{z}(t^{(0)}) = 0 \tag{5.7}$$

i.e.

$$z_{-1} = z_0 - h \cdot 0 + \frac{h^2}{2} f(z_0) \tag{5.8}$$

Another way for finding $z_{ref}(t_1)$ is to transform (5.1) into a 1^{st} order system of ODEs and find a solution using the MATLAB integrator `ode45`, i.e.

1. Define $\mathbf{y}(t) := \begin{pmatrix} z(t) \\ \dot{z}(t) \end{pmatrix}$ and consequently $\mathbf{y}_0 := \begin{pmatrix} z_0 \\ \dot{z}_0 \end{pmatrix}$.
2. Get

$$\dot{\mathbf{y}}(t) = \begin{pmatrix} \dot{z}(t) \\ \ddot{z}(t) \end{pmatrix} = \begin{pmatrix} \dot{z}(t) \\ f(z(t)) \end{pmatrix} =: \mathbf{F}(\mathbf{y}(t)) \tag{5.9}$$

3. Apply `ode45`:

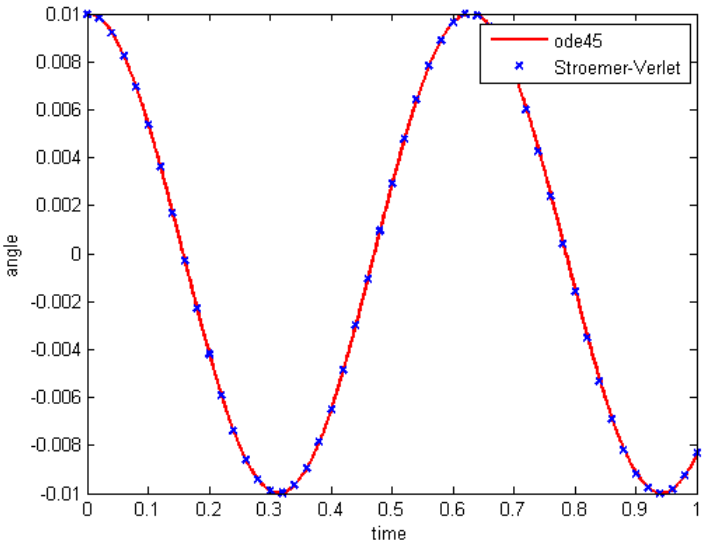
```
1 [t,y] = ode45(F,[0,t1],y0, options)
```

4. Set z_{ref} to be the first component of $\mathbf{y}$.

Remark: See function `yEx` in appendix A.6 for an application.

Note that the z_{ref} given by the two methods are very similar, in particular for large values of N, see Figure 5.1.

Figure 5.1: z_{ref} for $z_0 = 10^{-2}$, $\lambda = 100$ and $t_1 = 1$



5.3 The implementation

In order to apply the method developed in the previous chapters and find an approximative solution using the implemented MATLAB functions, we need to write the problem in form (1.1):

$$\ddot{z} = -\underbrace{\lambda}_{=:\omega^2} z + \underbrace{\lambda \cdot (z - \sin(z))}_{=:g(t,z)} \tag{5.10}$$

Note that we are in the second case described in section 2.3. Therefore we must use the functions for the general case, i.e. function `FindAppr2` or `osm`.

5.4 Error's analysis for the matrix form's method

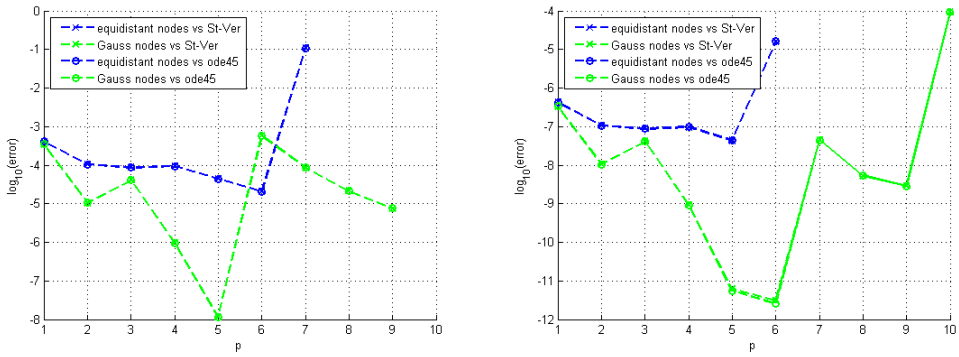
We begin by investigating the precision of the matrix form's method, i.e. we apply function `FindAppr2` to problem (5.1-3) and we study the errors produced for different values of p and m. From now on let $t_1 = 1$.
Please note that for the perturbed oscillation we do not have any "given" exact solution formula. Therefore we'll use both reference solutions found in section 5.2 to get the approximation's error.

Remark: In section 4.3 was proven that function `FindAppr2` solves exactly the linear case. Therefore here we only consider the perturbed case.

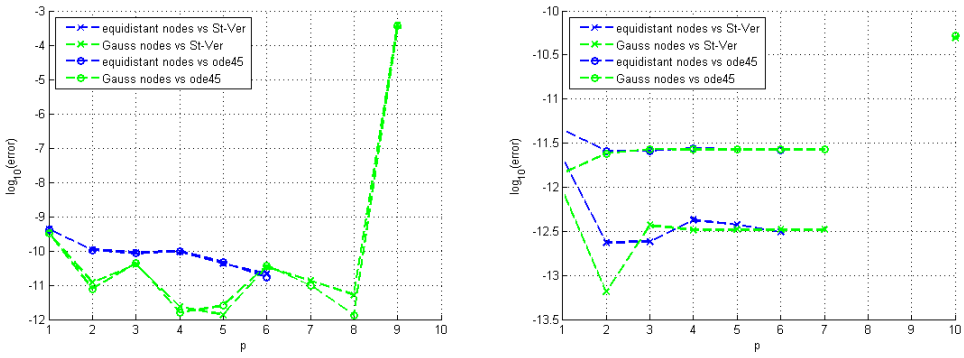
In Figure 5.2 are plotted the errors at t_1 w.r.t. both reference solutions for $m = 1$, $N = 10^6$ (the parameter of Störmer-Verlet iterations), $p = 1, \ldots, 10$, $\lambda = 10^2$ and for different values of z_0.

Figure 5.2: log(error) vs. p using function `FindAppr2` for $t_1 = 1$, $m = 1$, $\lambda = 10^2$ and ...

(a) $z_0 = 10^{-1}$ **(b)** $z_0 = 10^{-2}$



(c) $z_0 = 10^{-3}$ **(d)** $z_0 = 10^{-4}$

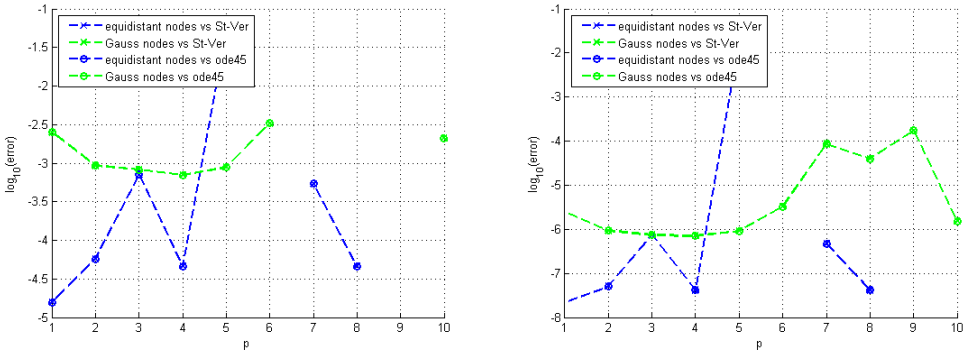


One can see that we got smaller errors when using Gauss nodes instead of equidistant nodes. For larger values of z_0 (i.e. plots **(a-c)**) we got a sort of convergence of errors' values for increasing p. However this convergence degenerates for "too large" p, due to the condition of the used matrix for the Newton iterations (cfr. Figure 4.4 that shows the behaviour of this condition for p increasing in the linear case). For this value of parameter N (i.e. $N = 10^6$), the error with respect to the reference solution corresponding to the Störmer-Verlet method is smaller than the one with respect to the `ode45` reference solution. It may also be interesting to note that smaller initial displacements give smaller average error's values, in particular in **(d)** the error's value is constant because the maximal machine precision has been reached.

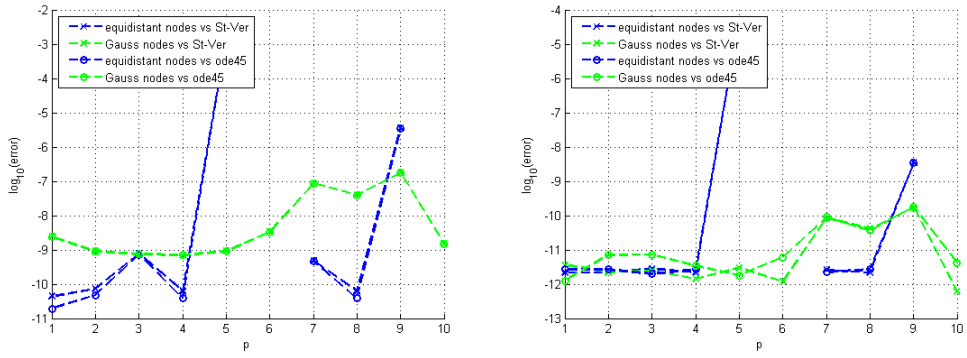
In Figure 5.3 one can see the plots corresponding to the same situations as in Figure 5.2 but with $\lambda = 10^4$.

Figure 5.3: log(error) vs. p using function `FindAppr2` for $t_1 = 1$, $m = 1$, $\lambda = 10^4$ and ...

(a) $z_0 = 10^{-1}$ **(b)** $z_0 = 10^{-2}$



(c) $z_0 = 10^{-3}$ **(d)** $z_0 = 10^{-4}$



In these cases the error's value corresponding to Gauss' nodes is quite constant, but for larger z_0 values (e.g. **(a-c)**) is larger than the most part of the errors produced by using equidistant nodes. Again, one can see that the average error's value decreases together with the initial displacement. Moreover, this value reaches the machine precision when a very small value for the initial displacement is chosen (e.g. **(d)**). Note that the error value corresponding to some values of p is not shown the plots in Figure 5.3. This happens most often when using equidistant nodes and it is due to the condition of $D\mathbf{G}(\cdot)$: in these cases the method does not produce an approximative solution to our problem. Therefore, we may conclude (again) that Gauss' nodes are recommended.

5.5 Errors analysis for the one step method

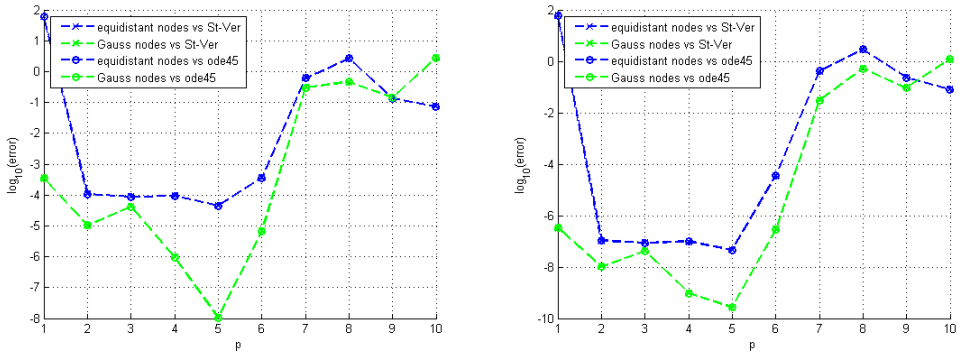
Now we apply function `osm` to ODE (5.1-3) with $m = 1$ fixed and with different values of p and to compare the resulting approximations to the reference solutions computed in section 5.2.

Remark: For this method the application to the linear case has been already studied in section 4.3. Therefore in this section only the perturbed case will be studied.

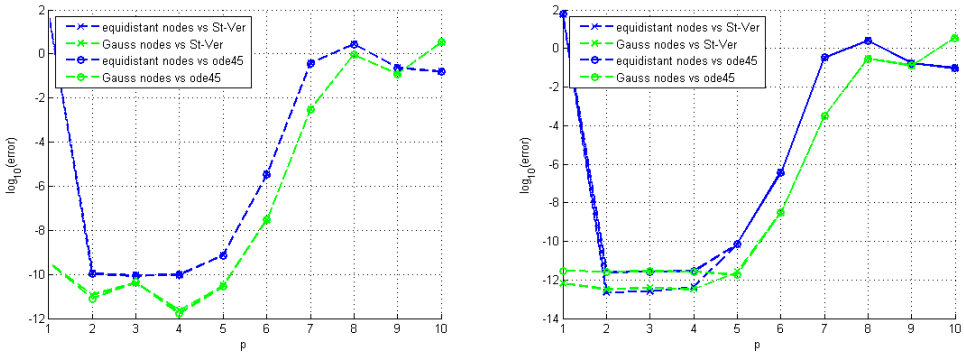
In Figure 5.4 one can see the plots corresponding to the same situations of Figure 5.2, but this time we used function `osm` instead of `FindAppr2` for finding the approximative solution.

Figure 5.4: log(error) vs. p using function `osm` for $t_1 = 1$, $m = 1$, $\lambda = 10^2$ and ...

(a) $z_0 = 10^{-1}$ **(b)** $z_0 = 10^{-2}$



(c) $z_0 = 10^{-3}$ **(d)** $z_0 = 10^{-4}$



These plots highlight that:

1. Again, collocation with Gauss' points gives smaller errors than the one

with equidistant nodes.

2. The values obtained for Gauss' nodes are very similar to the ones produced by using the matrix form method. For equidistant nodes we got bigger error's values for $p = 1$ and $p > 4$, i.e. when neither the linear case is correctly solved.

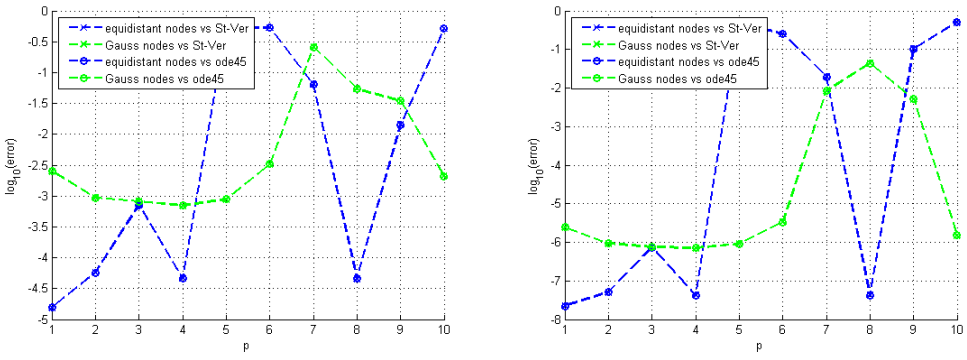
3. Exactly as in Figures 5.2-3, the average value of the errors decreases together with the initial displacement value.

4. It may be important to note that this method gives a solution for all values of p for both equidistant and Gauss' nodes.

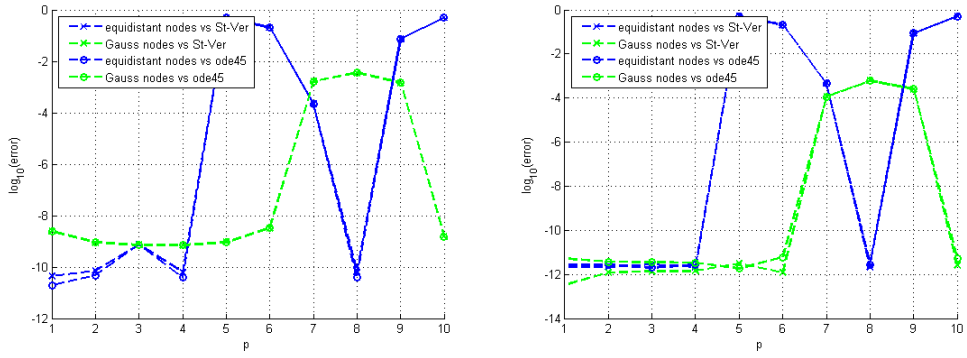
In Figure 5.5, we plotted the errors produced by `osm` and corresponding to the same situations as in Figure 5.3. One can conclude the same as for the previous Figures.

Figure 5.5: log(error) vs. p using function `osm` for $t_1 = 1$, $m = 1$, $\lambda = 10^4$ and ...

(a) $z_0 = 10^{-1}$ **(b)** $z_0 = 10^{-2}$



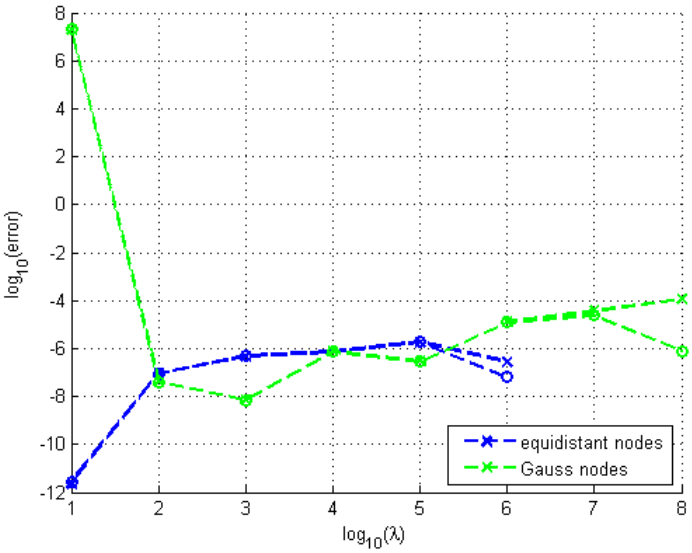
(c) $z_0 = 10^{-3}$ **(d)** $z_0 = 10^{-4}$



5.6 Errors for different frequencies

Finally, it may be interesting to see how the approximative solutions produced by the implemented functions behave with respect to the parameter λ, i.e. with respect to the frequency of the oscillation. This time let $m = 1$, $p = 3$, $t_1 = 1$ and $z_0 = 10^{-2}$ be fixed and observe (in Figures 5.6-7) what happens by applying functions `FindAppr2` and `osm` to our problem with different values of λ i.e. for $\lambda = 10^k$ with $k = 1, \ldots, 8$.

Figure 5.6: errors vs. λ using `FindAppr2` for $m = 1$, $p = 3$, $t_1 = 1$ and z_0



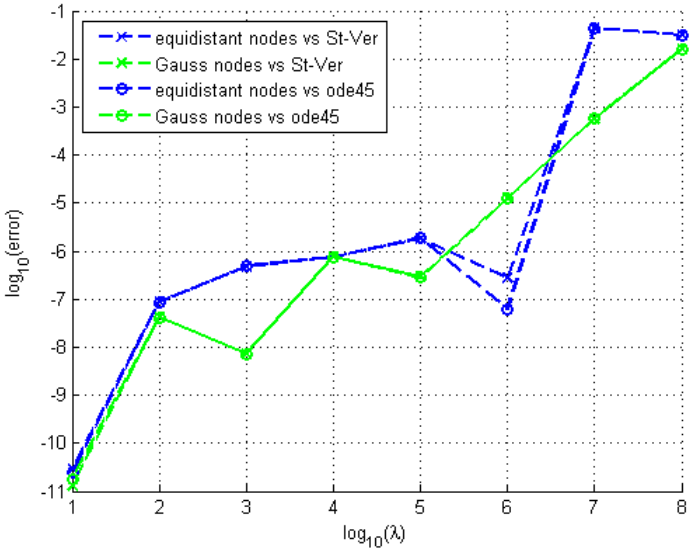
In these figures one can see that in both cases the error increases with λ and that in general it may be a good idea to use Gauss' nodes than equidistant ones. Because no solution is found when using `FindAppr2` with equidistant nodes and large values of p and because the errors produced by the first ones are often smaller than the others when using `osm`.
Moreover, using regression, we get that in the first case (i.e. using function `FindAppr2`) the error's values tend to stay quite constant among all the p and in the second one (i.e. using function `osm`) the following relation holds

$$error \approx \Theta\left(\lambda^{1.1}\right) \tag{5.11}$$

i.e. the error increases as fast as parameter λ.

Therefore we may conclude that errors do not increase faster as λ and consequently we may get good results even if λ is considerably large.

Figure 5.7: errors vs. λ using `osm` for $m = 1$, $p = 3$, $t_1 = 1$ and z_0



5.7 Conclusion

The previous sections suggest that in general the use of Gauss' collocation nodes is recommended because it often produces smaller error's values than collocation with equidistant nodes and even when the approximation obtained by using the others appears to be better, the difference is always a small fry. Moreover, for such a choice of nodes the plots show some convergence, which blows up at some point (often $p = 6$) because of the condition of the matrix used in the (simplified) Newton iterations (in `FindAppr2`) or because of the error produced by `fsolve` trying to solve the system of equations for the increments (in `osm`). Therefore, we may conclude that one needs to reach a compromise between the desire of precision (i.e. the temptation to choose a large p) and the limit imposed by the condition of the matrices used by the functions. For this particular example such a compromise may be to choose $p = 5$.

Another important aspect is the relation between the size of the initial displacement value z_0 and the average size of the errors obtained with the implemented functions: the smaller the initial value, the smaller the produced error. In particular, one can see in Figures 5.2-4 that for very small values of z_0 (e.g. 10^{-4}), the error's value is constant among the p values because machine precision is reached.

Finally, section 5.6 highlights that the error's size increases together with p.

Hence, we may choose quite large values for λ and still keep some accuracy in the approximation (e.g. in Figures 5.6-7 one can see that for such a parameters' choice the error size is about 10^{-6} even if we choose $\lambda = 10^5$).

Chapter 6

Conclusion

From the error's analysis performed for the numerical experiments in previous chapters we can draw the following conclusions about the methods presented in chapter 2 and their implementations using MATLAB described in chapter 3.

First of all, the error's analysis for the linear oscillation performed in section 4.3 shows that the *matrix form* methods and their implementations solves correctly the linear case for all but few values of the polynomial degree's parameter p. Conversely, the *one step form* implementation finds approximative solution's of the linear oscillation than can be quite far away from the exact value. In particular when using equidistant collocation nodes and large values of parameter p. These error's values are due to the fact that the matrices used for the (simplified) Newton method (in `FindAppr2`) and by function `fsolve` (in `osm`) become ill conditioned for increasing p value.

Applied to both linear and perturbed oscillations, the use of Gauss's collocation nodes tends to give better results than collocation with equidistant nodes. Moreover, for *matrix form* methods applied to oscillations with state-independent perturbation this type of collocation shows exponentially convergent error's size (see chapter 4, in particular Figures 4.5-6 and 4.12-13) and a solution is always produced (this is not true when using equidistant nodes).

The first numerical experiments shows even another important point: when the perturbation function is independent from the state, (i.e. $g(t, z) \equiv g(t)$), the use of function `FindAppr` (i.e. to find the coefficients of the modulated Fourier expansion by matrix inversion as described in the first part of section 2.3) is recommended because it is more efficient (i.e. it run much faster)

and gives results as least as precise as the ones that would be produced by the other methods: e.g. error's size in the order of $\approx 10^{-11}$ (about machine precision) for the the first proposed numerical experiment. Moreover, Figures 4.12-14 show that functions `FindAppr2` and `osm` "fail" for some (large) values of p: the first one because it doesn't produce a solution and the second one because it gives quite large error's values.

The second numerical experiment (chapter 5) shows that the error's size is strictly connected with the choice of the oscillation's initial displacement: the smaller the chosen parameter, the smaller the approximative solution's error. For very small initial parameter's choice the errors tend to stay constant because the maximal machine precision's value is reached.
Another point is that both general methods tend to "blow up" for increasing p values. For the one step form's method (i.e. function `osm`) this problem was already highlighted during the first numerical experiment's analysis for both the linear and the perturbed oscillation: in Table 4.1 one can see that for $p > 4$ using equidistant nodes and for $p > 6$ using Gauss nodes this function gives a very inaccurate approximation for the linear oscillation, while in Figure 4.14 one can see that for the same values of p the approximation of the perturbed oscillation's solution blows up too.

Therefore we have to reach a compromise between the theoretically convergence of the error's size for increasing p value and the limits imposed by the condition of the needed matrices, which for too large values of p become very ill conditioned and cause inaccuracy. A "rule of thumb" may be to choose p in the interval $[3, 6]$.

Appendix A

Useful MATLAB functions

A.1 xvec

```
function[x] = xvec(m,p,t,omega)
% xvec gives a vector containing the monomials of the collocation
% method at a given time t.
%   m = truncation parameter
%   p = polynomial's degree parameter
%   t = time
%   omega = ODE parameter

x = zeros((2*m+1)*(p+1),1);
l = m*(p+1)+1;

for k=-m:m
    for d=0:p
       x(k*(p+1)+d+l) = t^d*exp(1i*k*t*omega);
    end
end

end
```

A.2 HelpFunction

```
function [ G , DG ] = HelpFunction( a, B, g, omega, m, p, tau, Y0, u )
%HelpFunction gives the values of G and DG
%   a = coefficients vector
%   B = collocation matrix
%   g = perturbation function
%   omega = ODE parameter
%   m,p = collocation parameters
%   tau = collocation nodes vector
%   Y0 = initial conditions
```

```
%   u = gaussian required if 0

% initialization
y0 = Y0(1);
y1 = Y0(2);
n = (2*m+1)*(p+1);
y = zeros(n,1);

% for DG
if u == 0
    Y = zeros(n,n);
    ah = sym('ah',[n,1]);
    xh = sym('xh',[n,1]);
    th = sym('th');
    Dg = jacobian(g(th,xh.'*ah),ah);
end

% Computation of the g_j and of the Dg_j
for j=1:n-2
    x = xvec(m,p,tau(j),omega);
    y(j) = g(tau(j),x.'*a);
    if u == 0
        Yh = subs(Dg, th, tau(j));
        Yh = subs(Yh, xh, x);
        Y(j,:) = subs(Yh, ah, a);
    end
end

y(n-1) = y0;
y(n) = y1;

% Definition of G and of DG
G = B*a-y;      % according to (2.32)
if u == 0
    DG = B-Y;   % according to (2.34)
else
    DG = 'Not Required';
end

end
```

A.3 Cmat

```
function [ C ] = Cmat( omega, m, p, c )
%Cmat gives the matrix for finding the coefficients of functions H_i
%   omega = ODE parameters
%   m = truncation parameter
%   p = polynomial degree parameter
%   c = 0-1 collocation nodes' vector

s = (2*m+1)*(p+1)-2;
```

```
C = zeros(s);
l = m*(p+1)+1;

for j = 1:s
    x = xvec(m, p, c(j), omega);
    x = x.';
    C(j,:) = x([1:l+p-2,l+p+1:end]);
end

end
```

A.4 yvec

```
function [ y ] = yvec( omega, m, p, t)
%yvec gives the primitives of auxiliary functions at t
%   omega = ODE parameter
%   m = truncation parameter
%   p = polynomial degree parameter

% Initialization
s = (2*m+1)*(p+1)-2;
s1 = m*(p+1)+1;
s2 = m*(p+1)-1;
y = zeros(s,1);

% k=0 according to (2.53)
for d=0:(p-2)
    y(d+s1) = t^(d+2)/((d+1)*(d+2));
end

% k\=0 according to (2.54)
for k=1:m
    for d=0:p
        y1 = 0;
        y2 = 0;
        for l1 = 0:d
           for l2 = 0:(d-l1)
               l = l1+l2;
               y1 = y1+(-1)^(l)*factorial(d)*t^(d-l)*exp(-1i*k*omega*t)/((-1i*k*omega)^(l+2)*
               y2 = y2+(-1)^(l)*factorial(d)*t^(d-l)*exp(1i*k*omega*t)/((1i*k*omega)^(l+2)*
           end
        end
        y(-k*(p+1)+d+s1) = y1;
        y(k*(p+1)+d+s2) = y2;
    end
end

end
```

A.5 HelpIncrements

```
function [ G ] = HelpIncrements(T, omega, g, Z0, s, c, h, a, H0, K)
% HrlpIncrements helps finding the increments

% initialization
f = @(t,z) -omega^2*z+g(t,z);    % f defined as in (2.35)
G = zeros(s,1);

for j=1:s
    th = T(1)+c(j)*h;
    yh = Z0(1)+h*Z0(2)*c(j)+h^2*(a(:,j)-c(j).*H0).'*K;
    G(j) = f(th,yh)-K(j);        % according to (2.44)
end

end
```

A.6 yEx

```
function [ ySt, yod ] = yEx( y0, lambda, t1, N )
%yEx computes a numerical approximation of y at time t_1 using
%Stoermer-Verlet method and ode45 integrator

%   y0 = initial displacement (small)
%   lambda = ODE parameter
%   t1 = final time
%   N = Quantity of timesteps for Stoermer-Verlet (min 10^6)

% RHS Definition as in (5.4)
f = @(y) -lambda*sin(y);

%Stoermer-Verlet Approximation:
% Step-length
h = t1/N;

% Start points
y_old = y0+h^2*f(y0)/2; % according to (5.8)
y_new =y0;

% Iteration according to (5.5)
for k=1:N
    y = -y_old+2*y_new+h^2*f(y_new);
    y_old = y_new;
    y_new = y;
end

ySt = y_new;

% ode45 Approximation
```

```
31 % set tight tolerance
32 options = odeset('abstol',1E-11,'reltol',1E-11,'stats','on');
33
34 % solve the 1st order ODE
35 odefun = @(t,y) [y(2);f(y(1))];      % according to (5.9)
36 [t,y] = ode45(odefun, [0,t1],[y0;0],options);
37
38 yod = y(end,1);
39
40 end
```

Bibliography

[1] P Deuflhard and F. Bornemann, *Numerische mathematik 2, gewöhnliche differentialgleichhungen*, de Gruyter, Berlin, Germany, 2008.

[2] C.Lubich E.Hairer and G.Wanner, *Geometric numerical integration illustrated by the störmer-verlet method*, Acta Numerica **12** (2003), 399–450.

[3] _____, *vol 31 of springer series in computational mathematics*, Geometric numerical integration (2, ed.), Springer, Heidelberg, Germany, 2006.

[4] Prof. Ralf Hiptmair and Dr. Vasile Gradinaru, *Numerische mathematik (numerik der odes)*, Lecture's notes 2011, available at http://www.sam.math.ethz.ch/~hiptmair/tmp/NUMODE11.pdf.

[5] Daniel Kressner, *Numerische methoden*, Lecture's notes 2010, available at http://www.math.ethz.ch/education/bachelor/lectures/fs2010/math/nm/skript.pdf.

[6] MATLAB, *version 7.12.0 (r2011a)*, The MathWorks Inc., Natick, Massachusetts, 2011.

[7] W.L. Miranker and M. van Veldhuizen, *The method of envelopes*, Mathematics of computation **32** (1978), 453–496.