University of Castilla-La Mancha

Computing Systems Department

Simulation of Renewable Generation Systems on Parallel Architectures

Ph.D. Thesis

submitted to the Computing Systems Department
of the University of Castilla-La Mancha
in fulfillment of the requirements for the degree
of Doctor of Philosophy in
Tecnologías Informáticas Avanzadas (Computer Science)

Alberto Jiménez Ruiz

Advisors: Gerardo Fernández Escribano
Miguel Cañas Carretón

Albacete, November 2023

Universidad de Castilla-La Mancha

Departamento de Sistemas Informáticos

Simulación de Sistemas de Energía Renovable en Arquitecturas Paralelas

Tesis Doctoral

presentada al Departamento de Sistemas Informáticos
de la Universidad de Castilla-La Mancha
para la obtención del título de
Doctor en Tecnologías Informáticas Avanzadas

Alberto Jiménez Ruiz

Directores: Gerardo Fernández Escribano
Miguel Cañas Carretón

Albacete, noviembre de 2023

Dedicado a mis padres:
Francisco Javier Jiménez García y
M.ª del Pilar Ruiz Martínez,
por apoyarme tantísimo durante
estos años tan extraordinarios.

Agradecimientos

Ante todo, quisiera expresar mi más sincero agradecimiento a mis directores Gerardo Fernández Escribano y Miguel Cañas Carretón por darme la oportunidad de realizar esta tesis doctoral con ellos. Sin su guía y conocimientos jamás habría podido escribir estas líneas. Con ellos me he introducido en el mundo de la investigación, he crecido como persona y aguardo con ilusión lo que me espera por esta nueva senda. En especial me gustaría darle las gracias a Gerardo Fernández Escribano, con el que he aprendido muchísimo y que ha sido una constante lección de trabajo y perfección. También me gustaría dar las gracias a José Luis Sánchez García por sus consejos y sugerencias adicionales sobre esta tesis.

También le agradezco todo su apoyo a todos los compañeros y amigos del Grupo de Redes y Arquitecturas de Altas Prestaciones (RAAP). Gracias a ellos pude sobrellevar mejor todo el trabajo relacionado con la tesis. Muchas veces nos preguntábamos alguna duda puntual, y así nos ayudábamos unos a otros. Ellos me han introducido a otras áreas del conocimiento a las que anteriormente no les había prestado tanta atención. Les agradezco toda la ayuda prestada y que hayan expandido mis horizontes. No puedo olvidarme del personal del RAAP, en especial de Raúl Galindo Moreno, que me ayudaron a montar algún equipo que he necesitado a la hora de realizar esta tesis.

De entre todos los compañeros del RAAP me gustaría nombrar a Rubén Miguélez Tercero, con el que he colaborado en artículos y actas de congreso, y con el que también he compartido una estancia doctoral en el extranjero. Sus consejos y sugerencias me han ayudado muchísimo en esta tesis.

Por último, me gustaría agradecerles esta tesis a mis padres Francisco Javier Jiménez García y M.ª del Pilar Ruiz Martínez, que me han estado apoyando durante todos estos años de trabajo. No siempre he sido un hijo muy agradecido, pero sin su compresión y paciencia, sobre todo paciencia, jamás habría presentado esta tesis.

La investigación presente en esta tesis doctoral ha sido financiada por el Fondo Europeo de Desarrollo Regional (FEDER) y la Junta de Comunidades de Castilla-La Mancha a través de los proyectos SBPLY/17/180501/000353, SBPLY/19/180501/000287 y SBPLY/21/180501/000195; por la Universidad de Castilla-La Mancha (UCLM) a través de las ayudas a grupos de investigación I+D+i con referencias 2021-GRIN-31042 y 2023-GRIN-34056; y por los proyectos RTI2018-098156-B-C52, PID2021–123627OB-C52 y PID2021–126082OB-C21 del Ministerio de Ciencia, Innovación y Universidades.

Abstract

The world is facing an unprecedented crisis due to the foreseeable lack of traditional energy sources. Geopolitical turmoil and increasing energy prices have promoted the installation of larger and more renewable energy power plants, chief among them wind farms and photovoltaic power plants. These power plants consist of many small individual generation units, which together can match the output of a single steam powered synchronous generator of a traditional type of power plant, such as combined cycle power plants, or nuclear plants. This considerable number of electrical elements within renewable power plants pose a challenge to power system operators, as they must oversee the proper operation of the electrical grid, since their behavior must be simulated, and this means a large computational cost. The large computational cost means that aggregated models must be used, which are simplified versions of the mathematical models of the renewable generation machines. The precision of these models is lower regarding the non-aggregated models due to the simplifications considered, and therefore their results will show deviations from the real behavior of these systems. Another option is to simulate power systems using architectures with high parallelism. These architectures enable the simulation of many individual generation units without loss of precision and with reasonable computational cost.

Graphics Processing Units (GPUs) are coprocessors mostly dedicated to the creation, modification and processing of data related to the display of images. These processors are designed to efficiently process millions of elements in parallel due to the nature of graphics processing calculations, both in 2D and 3D. Currently, modern vendors allow users to harness the GPU potential for other applications, such as, for example, power system simulation.

This thesis explores the viability of power system simulation on parallel architectures, mainly those featuring renewable energy sources. Renewable energy sources are mostly based on many individual generation units, and the simulation of such systems will benefit tremendously from parallel simulation. Simulations of both large wind farms and photovoltaic power plants will be considered. The algorithms of simulation of these systems will be adapted to the architecture of the graphics cards.

First, the simulation of a wind farm will be presented. The state of the individual wind turbines will be processed in parallel with the GPUs, while also solving the wind farm on the GPU. The topology of the wind farm, as it will be shown, leaves little room for parallelism, but, nevertheless, a considerable speedup is achieved. The equations of the wind turbine are linearized, and their equilibrium point is obtained.

Second, a simulation of a large photovoltaic power plant is presented. The nominal power of these facilities ranges from the 10s to 100s of MW, existing already cases of photovoltaic power plants with more than a gigawatt of installed power, but their output heavily depends on external phenomena, such as cloud occlusion. The breakage of individual solar panels also takes a dent in the output. The simulation using graphics cards allows to account for these conditions, while achieving a great speedup.

Resumen

El mundo se enfrenta a una crisis sin precedentes en el futuro cercano debido a la falta de fuentes de energía tradicionales. La agitación en la política internacional y los precios ascendentes de la energía promueven la instalación de más y mayores plantas de generación basadas en energías renovables, siendo las más importantes entre ellas los parques eólicos y los huertos solares. Estas centrales de generación constan de muchos elementos de generación pequeños que juntos pueden generar tanta energía como un solo generador síncrono accionado por una turbina de vapor, como el que se emplea en las centrales de ciclo combinado o en las centrales nucleares. Este gran sistema de generadores de energía renovable plantea un problema para el operador del sistema eléctrico, encargado de velar por la correcta operación de la red eléctrica, pues el comportamiento de este sistema se ha de simular y eso supone un gran coste computacional. Este gran coste computacional obliga a usar modelos agregados, que son versiones simplificadas de los modelos matemáticos de las unidades de generación renovable. La precisión de estos modelos es inferior respecto a los modelos sin agregar debido a las simplificaciones consideradas por lo que sus resultados presentan desviaciones respecto del comportamiento real de estos sistemas. Otra opción es simular el sistema eléctrico usando arquitecturas con un gran paralelismo. Estas arquitecturas permiten la simulación de múltiples generadores sin pérdida de precisión y con un coste computacional razonable.

Las unidades de procesamiento gráfico son coprocesadores dedicados mayoritariamente a la creación, modificación, y procesamiento de datos relacionados con la representación visual. Estos procesadores se diseñan para procesar eficientemente millones de elementos en paralelo debido a la naturaleza de los cálculos relacionados con el procesado de imágenes, tanto en 2D como en 3D. Actualmente, los fabricantes permiten a los usuarios aprovechar el potencial de las GPU para otras aplicaciones como, por ejemplo, la simulación de sistemas eléctricos.

Esta tesis explora la viabilidad de la simulación de sistemas eléctricos en arquitecturas paralelas, principalmente aquellas con presencia de energías renovables. Los sistemas basados en energías renovables se basan principalmente en muchos generadores individuales, y la simulación de éstos en paralelo se verá muy beneficiada. Se considerará tanto la simulación de parques de aerogeneradores como de huertos solares. Los algoritmos de simulación de estos sistemas se adaptarán a la arquitectura de las tarjetas gráficas.

En primer lugar, se presentará la simulación de un parque eólico. El estado de cada aerogenerador se procesará en paralelo con GPUs, a la vez que se resuelve en su conjunto el parque eólico también en la GPU. La topología típica del parque eólico da pocas oportunidades a la paralelización, pero aun así se ha conseguido lograr un gran aumento del rendimiento. Las ecuaciones del parque eólico se han linealizado y se ha obtenido su punto de equilibrio.

En segundo lugar, se presenta la simulación de un gran parque solar fotovoltaico. La potencia instalada de esta clase de instalaciones se encuentra en el orden de las decenas a cientos de megavatios, existiendo casos incluso que superan el gigavatio, pero su potencia se ve afectada por fenómenos externos como son la oclusión por una nube. La rotura de paneles singulares también merma la salida de todo el parque. La simulación usando tarjetas gráficas permite tener en cuenta estas cuestiones, logrando un gran aumento del rendimiento.

Glossary

API	Application Programming Interface
BESS	Battery Energy Storage Systems
BiCG	Biconjugate Gradient
BiCGStab	Biconjugate Gradient Stabilized
CAD	Computer Aided Design
CC	Compute Capability
CFD	Computational Fluid Dynamics
CG	Conjugate Gradient
CGNR	Conjugate Gradient Normal Residual
COO	Coordinate [Sparse Matrix Storage Format]
CPU	Central Processing Unit
CSC	Compressed Sparse Column
CSR	Compressed Sparse Row
CSR-I	Compressed Sparse Row Improved
CUDA	Compute Unified Device Architecture
DAE	Differential Algebraic Equation
DIA	Diagonal [Sparse Matrix Storage Format]
DFIG	Doubly-Fed Induction Generator
DOK	Dictionary of Keys
ELL	ELLPACK [Proper name of a Software for Solving Elliptic Problems]
GEMM	General Matrix-Multiply Product
GEMV	General Matrix-Vector Product
HVDC	High Voltage Direct Current
HYB	Hybrid [Sparse Matrix Storage Format]
EMT	Electromagnetic Transient
INT32	32-bit Integer
IC	Integrated Circuit
IGBT	Insulated-Gate Bipolar Transistor
ISA	Instruction Set Architecture
FMA	Fused Multiply-Add
FP32	Simple Precision Floating Point
FPGA	Field-Programmable Gate Array
GPGPU	General-Purpose Computing on Graphics Processing Units
GPC	Graphics Processing Cluster
GPU	Graphics Processing Unit
GSC	Grid-Side Converter
ILP	Instruction Level Parallelism
LHS	Left-Hand Side

LTI	Linear Time Invariant
MATE	Multi-Area Thévenin Equivalent
MPPT	Maximum Power Point Tracking
NTSC	National Television System Committee
ODE	Ordinary Differential Equation
OoOE	Out-of-Order Execution
PAL	Phase Alternating Line
PCC	Point of Common Coupling
PF	Power Flow
PI	Proportional Integral
PHES	Pumped Hydro Energy Storage
PMSG	Permanent Magnet Synchronous Generator
PS	Power System
PSO	Power System Operator
PTX	Parallel Thread Execution
PV	Photovoltaics
PVPP	Photovoltaic Power Plant
PWM	Pulse-Width Modulation
RAAP	Grupo de Redes y Arquitecturas de Altas Prestaciones
RAM	Random Access Memory
REE	Spanish Electric Grid, in Spanish “Red Eléctrica de España”
RHS	Right-Hand Side
RPF	Radial Power Flow
RSC	Rotor Side Converter
SECAM	Séquentiel de couleur à mémoire
SCIG	Squirrel Cage Induction Generator
SIMD	Single Instruction Multiple Data
SFU	Special Function Unit
SM	Stream Multiprocessors
STC	Standard Test Conditions
UCLM	Universidad de Castilla-La Mancha / University of Castilla-La Mancha
TFlop	Teraflops
THD	Total Harmonic Distortion
TPC	Texture Processing Cluster
TSR	Tip Speed Ratio
VRAM	Video Random Access Memory
UM	Unified Memory
ZCPMM	Zero-Copy Pinned Mapped Memory

Index

List of Figures

List of Tables

List of Codes

Chapter 1. Introduction

The following chapter will introduce the topic of this Ph.D. thesis, its motivation, and its objectives.

1.1. Introduction

The use of energy is deeply tied to the very existence of human society. The oldest energy source that humankind has ever used was the wind. The great inventions of the sail and the mill enabled both sailing and grinding, which are activities key to agriculture and trading; likewise, watermills take advantage of waterfalls and water streams using the same principle as the traditional mill. Solar power was also used, albeit indirectly, since many civilizations relied on the warmth the sun produces to grow their crops. In fact, the sun was so important that many ancient civilizations regarded it as a deity. Finally, wood can also be burnt and used as a source of energy, mainly for heating and forging. Centuries and millennia come and go, societies flourish and crumble, and primary energy sources remain the same until the industrial revolution. James Watt's steam engine demanded a large amount of heat to boil water, and it required high-density fuels. Fossil fuels were the perfect match. Coal was the energy source back then and, later, petroleum derivatives, such as kerosene, gasoline, and fuel oil were also used. Further discoveries such as electricity, the light bulb, and the electric generator and motor pushed industrial societies into the electrical age. Electricity is generated thanks to large generators that convert the motion of steam vapor into electrical energy and it was then distributed through a nationwide Power System (PS). Fossil fuels generate the heat needed to boil that water and, with the advent of locomotives and cars, also directly powered them.

Society progresses, energy consumption soars, and then, new problems arise. Some of these energy sources are of limited supply. For example, it is predicted that before 2030 we will reach the oil peak, which is the moment when the maximum amount of petroleum will be extracted and then a downward path of production will occur due to the lack of new oil fields and their associated cost [1]. Another issue is the ever-increasing worldwide energy demand. The global electricity demand in 2010 was 18548 TWh, which increased to up to 24700 TWh in 2021, which is equivalent to a 33% increase in just 11 years [2]. The growth of electrical demand does not slow down. In fact, the rise of living standards across the world, the electrification, and the renewal of national vehicle fleets that are powered by batteries and not fossil fuels bring about an even larger electrical energy demand. The foreseeable lack of traditional energy sources coupled with the need for larger amounts of energy rekindles the interest in alternative energy sources such as wind and solar power. Other renewable energy sources such as hydropower and biomass have more limited viability, either because the land and nutrient requirements required for soil renewal entail a low yield [3], or because the available potential for new large hydroelectric projects in developed countries is limited [4].

However, renewable energy sources have their downsides. The main one is their lack of control, which puts a strain on PSs. Wind resource and solar irradiance are fickle. Energy storage systems such as Pumped Hydro Energy Storage (PHES) and Battery Energy Storage Systems

(BESS) seek to add more flexibility to the operation of PSs. PHES pumps water upward to store electrical energy as potential energy, while BESS takes advantage of chemical reactions to store the energy in battery cells. Both will play a crucial role in the foreseeable future and will be the key to a sustainable future despite their challenges [5]. Wind and solar power generation is irregular but also disperse. They require many generation units spanning hectares of land. Handling so many small generators is one of the responsibilities of the Power System Operators (PSO).

Before the widespread adoption of renewable energy sources, the management of electrical PS was straightforward. Large power plants supplied a myriad of small and medium-sized consumers, and their energy needs could be estimated to a reasonable extent. This allows the grid operator to coordinate with the managers of the power plants to ensure the electrical demand is always covered while meeting the technical and regulatory requirements imposed by the grid operator. Traditional power plants are also flexible, so energy production can be increased or reduced as needed. This means that grid operators can set the generation power targets as demand varies. That will change with the advent of new renewable energy sources, chief among them wind and solar power. As mentioned above, the weather is capricious, and the PSO must consider weather forecast data in their models to estimate both demand (extreme cold and hot weather usually predicts large energy use) and generation (still and cloudy weather does not see much wind and solar generation respectively). Therefore, network operators must perform many simulations of the PS to check its state, optimize it, and verify its viability.

PS simulation requires a large amount of computing power. Its state changes greatly on an hourly basis, and faults can also trigger the protections of the PS, which can happen at any moment. The traditional solution to this problem was using faster processors. Moore's law predicted that the number of transistors crammed into a processor doubles every 2 years [6]. However, the pace of technology has slowed down and, therefore, we can no longer rely solely on faster Central Processing Units (CPUs). The alternative followed by most CPU vendors is the enlargement of CPU caches, the inclusion of more powerful instructions in their Instruction Set Architectures (ISA), and the addition of more CPU cores. The rationale behind these improvements is related to the clock cycle. Larger caches mean that the CPU can prefetch more data from RAM and thus read from the L1, L2, and L3 caches, which are orders of magnitude faster. On the other hand, more powerful instructions mostly involve vectorized instruction sets, which under the Single Instruction Multiple Data (SIMD) paradigm, apply the same operation to many different data elements. For example, the *Golden Cove* CPU architecture developed by Intel for the x86-64 ISA features the AVX-512 instruction set, whose registers contain up to 16 single precision values [7]. Finally, improvements in branching predictors optimize the flow in the instruction pipeline, thus executing faster with code that relies on branching instruction.

The code must be programmed to take advantage of these new instruction sets. Compilers can vectorize the standard C code. However, a handwritten assembler is essential to fully exploit the potential of CPUs, and many software projects do so. The FFMPEG project[1] and its associated encoding and decoding libraries feature snippets of assembler code tailored to each computer

[1] The FFMPEG Project. `https://ffmpeg.org/`

architecture in the busiest sections of the code. CPU architectures that do not have specialized assembler code in the FFMPEG codebase execute generic C functions as a fallback.

These advances in parallelism are remarkable. Modern high-end CPUs cram dozens of CPU cores, and vectorization opens a new venue that enables further parallelism. But that is not enough. As mentioned above, traditional CPUs were designed to excel in branched code at the expense of parallel code. Researchers sought a new processor for specialized scientific and engineering calculations in which most of the time is spent on massively parallelizable tasks, such as adding two vectors together. As it happened, this processor already existed, and it was already installed on many computers: it was the Graphics Processing Unit (GPU).

This new processor offers massive parallelism combined with ease of use and availability. This Ph.D. thesis will explore the simulation of PSs in GPUs and will show the speedup what these systems are capable of. The following sections of this work will further explain the motivation, objectives, and structure of this thesis.

1.2. Motivation and Objectives

GPUs have a massive processing power and have an upper hand in parallelizable tasks. However, they have their downsides. Special compilers are needed to use them, and programming with optimization in mind is essential, for a naive implementation of an algorithm in the GPU will be many times slower than in the CPU. PS simulation requires substantial amounts of computing power, and they would benefit greatly from GPUs. Considering that, a question arises:

> HOW CAN WE IMPLEMENT THE SIMULATION OF POWER SYSTEMS IN GPUS IN AN EFFICIENT WAY?

This question is very broad, as PS simulation can range from detailed models that carefully model internal behavior and electronics [8] to a PS where the elements have been abstracted as generators and consumers that inject and absorb a certain amount of power [9]. This work will center around a middle approach: renewable energy generator models whose output depends on external conditions but that have simplified the model to such an extent that fast harmonics related to electronic switching are disregarded. To efficiently simulate this kind of systems, novel algorithms were developed that take advantage of the new architectural features of graphics cards. The following thesis fulfils these main objectives:

- **Objective 1: Parallel simulation of wind turbines as standalone elements.** Wind turbines are large machines whose output depends heavily on the wind speed and the voltage present at their terminals. The output of these machines can be aggregated to create a large wind turbine whose behavior is like the sum of all machines. This approach offers results that resemble the expected one, but they still lack precision [10]. In this thesis, each individual wind turbine will be simulated at the same time on a GPU, while also obtaining a speedup.
- **Objective 2: Efficient simulation of an integrated wind farm efficiently on graphics cards.** These wind turbines are connected in an internal network that has a connection point to the rest of the PS. Most wind turbine layouts are connected in radial or ring topologies due to its saving costs in wiring [11]. PS connections are represented using an

admittance matrix, and this matrix is part of a linear system that must be solved to simulate PS. Traditional parallelism methods that process matrices with these characteristics do not extract much parallelism for the case of PSs. In this thesis, a novel integration technique will be presented.

- **Objective 3: Detailed solar photovoltaic power plant simulation.** Solar Photovoltaic Power Plants (PVPP) are large installations with a vast number of solar Photovoltaic (PV) panels, whose number can reach millions. Modern PS simulations use aggregated models for the PVPPs. This thesis will show that a detailed PS simulation of a PVPP is possible with affordable computational cost.

1.3. Methodology

The work plan that fulfils all objectives presented above is one that can be regarded as usual. It can be roughly split into 4 distinct stages:

- **Stage 1: Introduction and Literary Review.** First, a problem is established, which is simulation of PSs on graphics cards. After that, the state-of-the-art literary is reviewed to check what has been done in previous works. Both sequential and parallel versions of the algorithms presented are reviewed to further polish the objectives of this thesis.
- **Stage 2: Global Kernel Development.** After small pieces of the soon-to-be kernels are benchmarked, the final kernel code is assembled. This code will solve the objects laid out in the previous item.
- **Stage 3: Data Measurements and Result Analysis.** Different implementations are tested. The computing cost of each is worked out, and data is presented.
- **Stage 4: Conclusions and Future Work.** Conclusions are drawn from all data obtained in the previous stages. It is concluded that GPU is indeed more than capable of fully simulating a PS, and that this area of research is worth further exploring. After all the work is presented, new areas of research that might extend the content of this thesis were discovered.

1.4. Structure of this Thesis

This thesis is structured into the following chapters:

- **Chapter 1: Introduction**. This chapter introduces the topic of the thesis, which is the viability of PS simulation on graphics cards with low computational cost. It starts with a brief introduction that introduces the problem surrounding PS simulation while also presenting the distinctive features of graphics cards. The motivation and objectives section clarifies the objectives sought and concludes with the methodology followed while researching this thesis and a section that presents the structure of this thesis.
- **Chapter 2: Background.** In this chapter, a global review of PS simulation is presented with special emphasis on GPU simulation. This thesis will focus on NVIDIA graphics cards which are programmed thanks to the Compute Unified Device Architecture (CUDA) platform [12]. This chapter also briefly summarizes the state-of-the-art features of the new graphics cards that are relevant to the work in this thesis.

- **Chapter 3: Modeling of Wind Turbines.** This chapter introduces the models of the wind turbines that will be simulated, which are Doubly-Fed Induction Generators (DFIGs), by first explaining all relevant systems, subsystems, machines, and elements; and then presenting the mathematical models that govern them all, both the 5^{th} and 3^{rd} order model along with their pros and cons. The mathematical methods and transformations used to integrate these mathematical models are also shown in this chapter.
- **Chapter 4: Modeling of Photovoltaic Power Plants.** This chapter presents models of the solar panels that have been used throughout the thesis. This chapter also addresses solar panel inverters and their control strategies.
- **Chapter 5: Modeling of Power Lines.** Here, the details the different models used for the transmission lines, wires, and any connecting equipment are detailed. The π-section model is compared to the Bergeron model in terms of their adequateness and performance.
- **Chapter 6: CUDA Kernel Development.** After all the mathematical models are presented in the previous chapter, the kernels used to simulate the PSs are shown in this chapter. Design decisions are based on both the architecture of the graphics cards and the particularities of the models to be solved.
- **Chapter 7: Case Studies.** The implementations that are developed in the previous chapter are tested against each other and the computational cost is obtained. The computational cost results, which will be presented in this chapter, show that the CUDA implementation is the most performant, despite its weaknesses.
- **Chapter 8: Conclusions and Future Work.** This chapter summarizes the results of this thesis. All contributions are presented here along with all journal and conference papers produced in conjunction with this thesis. New venues of research have also been opened thanks to the work presented here, and this chapter lists all of them.
- **Chapter 9: Conclusiones y Trabajo Futuro.** This chapter is identical to the previous chapter, but its contents have been translated into Spanish.
- **References.** This section lists the bibliography used throughout the thesis.
- **Annex A: List of Unified Equations.** The final system of the DFIG presented in Chapter 3 after combining all equations and then linearizing them is a large set of equations too complex to be included in Chapter 3. They have therefore been moved to this annex for the sake of clarity and brevity.
- **Annex B: Parameters and Specifications.** The nominal parameters of both the DFIGs, the solar panels, and their connection elements are listed in this section. It has also been included the specifications of the machine where all simulations were run.

Chapter 2. Background

This chapter will present the state of the art of PS simulation on massively parallel systems, and more specifically on graphics card-based systems. Before proceeding, it is necessary to introduce a brief description of current graphics cards, their programming capabilities, and their computational power. This chapter is divided into four sections. First, Section 2.1 introduces the graphics processing units, its architecture and the tools to develop on them, while Section 2.2 does the same for all the types of PS studies that can be conducted. Next, there are two sections dedicated to the parallelization of the problems described in Section 2.2 with the devices described in Section 2.1. Section 2.3 expounds the different ways in which the mathematical operations used to solve PSs are parallelized, while Section 2.4 gathers the main strategies to parallelize the PS simulation as a whole.

2.1. Graphics Processing Units

Video signals are made of pulses that must be sent at a precise clock signal to produce a valid image. In the beginning, the three main standards of analog TV were National Television System Committee (NTSC), Phase Alternating Line (PAL) and Séquentiel de couleur à mémoire (SECAM). These standards define a line period, where data was sent for each video horizontal line, and blanking periods, when no data is sent. If a CPU is tasked with producing this signal with no external aid, then all remaining CPU tasks must be processed during the blanking periods to produce a video signal without distortions. For this reason, graphics processing was soon offloaded to external chips. The first graphics processing devices did simple jobs. They had a primitive version of Video Random Access Memory (VRAM) that could, depending on the video mode selected, hold either colored characters or bitmap-mode graphics with lower resolution. Its job was to compose the image for each scan line, while the CPU sent data to the video chip if any pixel or character changes. For example, if the CPU drew a line, it had to loop from the beginning to the end of the line and modify the VRAM or send appropriate commands to change the color value of each pixel. Technology moves forward, consumers wanted more detailed graphics, and these graphics processing devices became more advanced and were no longer tied to the CPUs, becoming properly Graphics Processing Units (GPUs).

The software that demanded more graphics processing power were either videogames or Computer Aided Design (CAD) related ones. These programs demand fast drawing of surfaces in 3D space and fast line drawing. These functions were the ones that were initially implemented in the hardware to offload more work to the GPU. This implementation of the subroutines that dealt with those operations is known as the fixed graphics pipeline, pictured in Figure 1. For instance, a 3D scene is made of vertices. These vertices are 3D coordinates. First, the GPU projects these coordinates into a 2D space, multiplying each vector coordinates with a projection matrix. The coordinates can be previously modified with, for example, a rotation matrix or a scaling matrix, which takes away even more work from the CPU. The vertices are then structured into primitives, which are the elemental shapes to be drawn. Each vertex contains information about its color and its texture coordinates. The shapes, which in modern hardware are triangles, are divided into fragments, and the color or texture position of each fragment is computed in the rasterization

process. Then, each fragment is processed and the output color of each pixel is obtained. Should any special effects be needed, they are added at the end of the pipeline during the pixel processing stage. The graphics card vendors created unified standards called OpenGL [13] and DirectX [14], which enabled multi-platform and multi-graphic card compatibility across all software. The fixed pipeline remained in use for a long time until programmers sought to create more advanced graphical applications that were not possible before. To make that possible, shaders were created. Shaders were small programs that could replace some stages of the fixed pipeline with custom code. These shaders executed code in parallel seamlessly, for the operations were applied to each vertex and/or fragment.

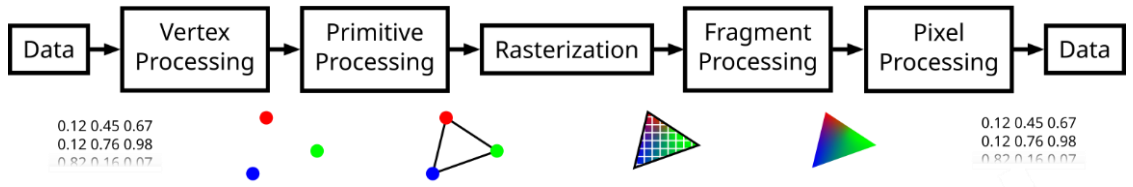


Figure 1. Fixed graphics pipeline.

Soon developers realized that GPUs were able to perform general computations. They adapted shaders by equating vertex data such as position, with other data such as voltage, current, or temperature. Operations were performed as usual while data was not represented on screen, but interpreted as needed. This approach is cumbersome and prone to errors due to the fact that shaders were not designed for this purpose. There was another issue, which was the unequal work performed by each stage of the pipeline. The hardware back then implemented each stage separately, which prevented fully using the GPU for certain loads. To solve this issue, a new architecture was proposed that unified all stages. Now, all types of shaders were executed by the same GPU hardware. That innovation finally paved the way for the use of the GPU as a general computing device.

In 2006 NVIDIA released the CUDA [12] platform for their graphics cards. This platform enables developers to harness GPU computing power to do general programming. This section of the thesis briefly explains how CUDA works, its programming model, its shortcomings, and its window of opportunity.

There are two main platforms to program GPUs: CUDA [12] and OpenCL [15]. The main difference between them is that OpenCL is portable across all GPU vendors and even Field-Programmable Gate Arrays (FPGAs), while CUDA works officially only on NVIDIA graphics cards. There are studies that show that the performance of CUDA on NVIDIA cards is better than OpenCL [16], while there are also papers that claim the opposite, claiming that if compared fairly, both implementations show similar performance [17]. CUDA has been chosen for this thesis for its position of dominance in the General-Purpose Computing on Graphics Processing Units (GPGPU) market.

The remainder of this section will be devoted to the concepts of the CUDA platform that will influence the design decisions of the kernels developed in this thesis. An introduction to the CUDA platform can be found in the official C++ Programming Guide released by NVIDIA [12]. Hardware architecture, code generation, and other internals vary from graphics card to graphics card, and therefore, different whitepapers and technical notes should be consulted.

2.1.1. CUDA Hardware Architecture

The graphics card used in this work is the *NVIDIA GeForce RTX 3090*, whose main specifications are listed in Annex B.3. This graphics card mounts a *GA102* chip which follows the *Ampere* architecture [18]. Figure 2 shows a summary of this architecture. An *Ampere* GPU has a Gigathread engine which handles the communication between the CPU applications and the GPU internal processors and performs context switching between different CPU applications if they use the resources of the machine. The memory controllers of the *Ampere* GPU handle the copying of memory between the host and the device, and they also manage the hardware related to NVLink. NVLink is a proprietary short-range and wired connection interface between NVIDIA graphics cards whose objectives are low latency and high bandwidth [19]. The GPU architecture also features a global L2 cache. As memory access latency is orders of magnitude slower than access to registers (see Section 2.1.3), a cache is installed to fetch previously accessed values faster. There is an inner L1 cache inside the Stream Multiprocessors (SMs) as well, as will be later explained. Finally, the *Ampere* GPU features Graphics Processing Clusters (GPCs) and Texture Processing Clusters (TPCs). The GPC is the dominant high-level hardware block with all the key graphics processing subunits [18]. It comprises the sections of the GPU that perform the computing. A GPC in the *Ampere* architecture features a raster engine and 8 raster operators. When an application renders 3D, the raster engine takes the information about all the triangles and generates its pixel information. This component is responsible for many of the stages of the fixed graphics pipeline. Raster operators are specialized hardware devoted to tasks such as anti-aliasing or Z-culling. More information on these tasks are can be found in any book dedicated to 3D graphics programming, such as [13]. The GPC is made of TPCs, which interact with the same raster engine. Each TPC features a Polymorph engine which, among other jobs, performs tessellation. Tessellation is the subdivision of polygons into smaller ones to add detail [13].

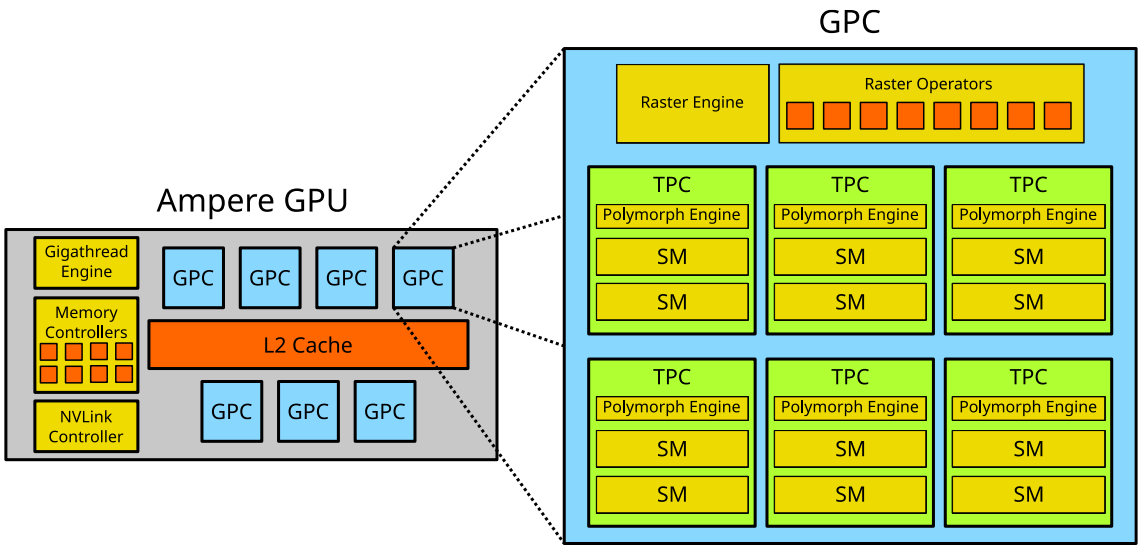


Figure 2. Simplified Ampere GPU architecture.

Finally, each TPC has two SMs. The SM is the actual piece of hardware that runs the CUDA code. Its architecture is shown in Figure 3. The SMs in the *Ampere* architecture are divided into partitions. Each partition has an L0 cache which holds the instruction lists, its own warp scheduler and warp dispatcher. The warp scheduler selects which warp shall be executed next, while the dispatcher issues the instructions to be executed. Warps, on the other hand, are groups of 32 threads to be executed in lockstep (see Section 2.1.2). The *Ampere* architecture features a single

dispatcher per scheduler, although previous CUDA architectures such as *Maxwell* are double issue, that is, a scheduler chooses two warps to be executed [20]. The warp scheduler then picks a warp that is currently not stalled and executes it [18]. The reasons that prevent a warp from being executed range from memory throttling to synchronization waits. If the threads use too many registers, the partitions will not be capable of storing all these threads, and will choose to run fewer warps, thus harming occupancy. Occupancy is defined as the ratio of active warps in an SM to the maximum number of active warps supported by the SM [21].

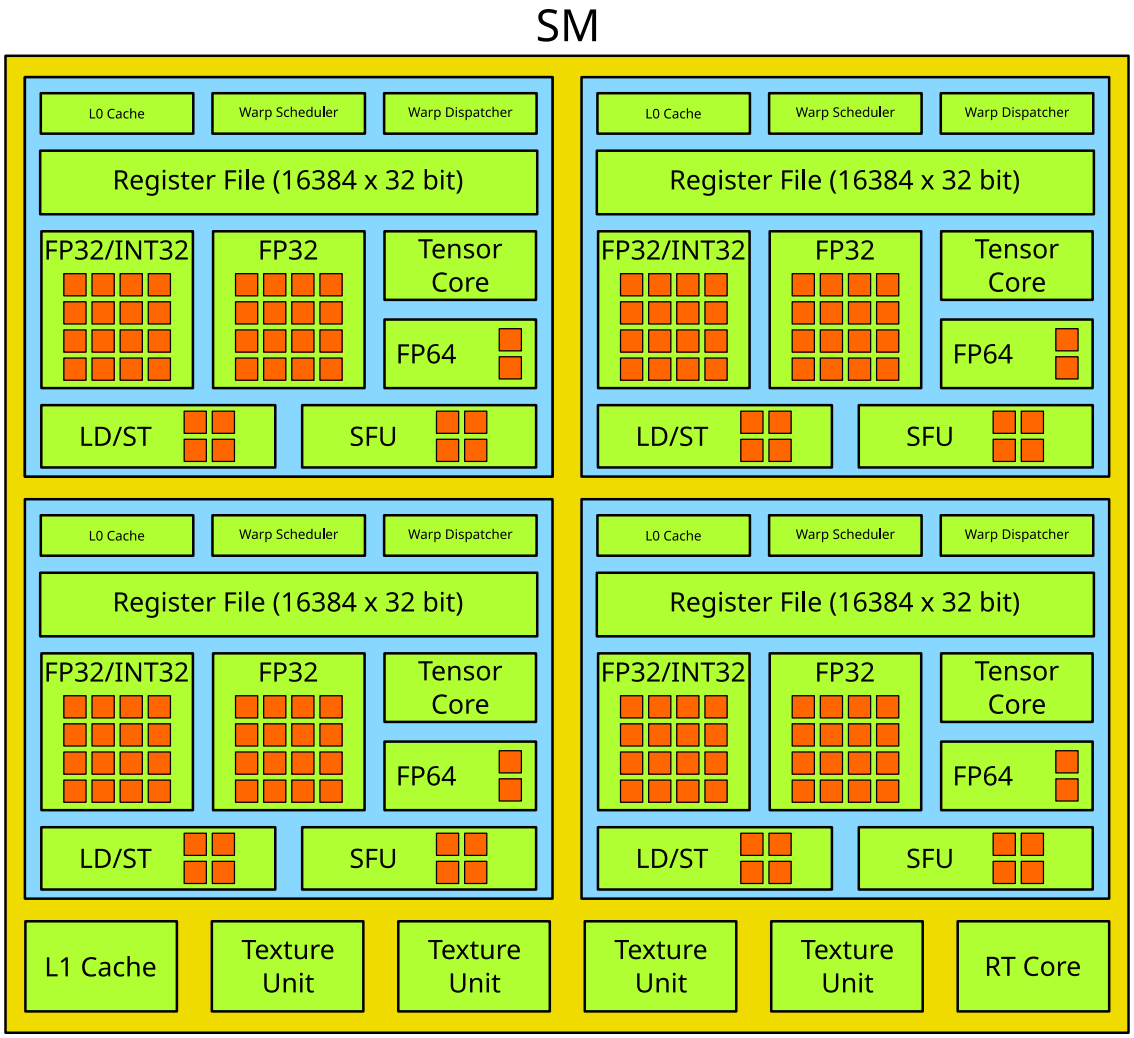


Figure 3. Streaming multiprocessor in the Ampere architecture.

The dispatcher issues instructions that are executed by the CUDA cores. CUDA cores are the minimal hardware abstraction and in each one of them a single thread is executed. Each SM has 16 cores that can perform (Simple Precision Floating Point) FP32 or (32-bit Integer) INT32 operations, i.e., process floating point number or 32-bit integer numbers. The SM of the *Ampere* architecture also features 16 CUDA cores that can only work as floating-point processing units, and 2 FP64 cores per SM that process double precision numbers. The Teraflops per Second (TFLOP/s) of FP64 cores is $1/64^{th}$ of the TFLOP rate of FP32 cores [18]. The architecture prioritizes the FP32 cores while FP64 ones are installed to be able to execute all CUDA programs, despite their subpar performance. The registers of the CUDA cores are shared across the partition. The partitions of the SM of the *Ampere* architecture have 16384 32-bit long registers. Each partition also has load and store units, which manage the fetching of data from VRAM; Special

Function Units (SFUs), that accelerate special functions such as `sinf`, `cosf` and `log2f`; and a tensor core. Tensor cores perform the operation $D = A \cdot B + C$, where A, B and C are 4x4 matrices. The input matrices A and B must be FP16 (half precision), while C and D can either be half or single precision.

The L1 cache, the texture units, and the raytracing core are shared by all partitions of the SM. The L1 cache, just as the L2 cache, is an intermediate memory that seeks to reduce memory access times, and the L1 cache also works as a texture cache in the *Ampere* architecture [18]. The load and store unit can bypass these caches to enhance performance, if necessary, by using the appropriate low-level instruction (more on that in Section 2.1.6)[2]. The texture units perform tasks related to texture data fetching and texture interpolation. The final component of the SM is the raytracing core (labelled as RT core in Figure 3). The RT core is optimized to perform operations related to the problem of raytracing. Raytracing is a set of techniques that calculate the final pixels colors of an image by modelling light rays as a stream of particles that bounce and refract when impacting on surfaces. At the time of drafting this thesis, developers cannot use RT cores directly to process custom functions. Instead, NVIDIA provides the library NVIDIA Optix[3] to take advantage of these cores for only raytracing problems.

2.1.2. CUDA Programming Model

Section 2.1.1 briefly introduced the hardware architecture of the graphics card used. As previously shown, its architecture is radically different to the CPU's. Most well stablished programming languages and paradigms are designed around an abstract computing machine whose programs execute sequentially with a flat memory address space. To harness the power of the GPUs NVIDIA created the CUDA platform. This platform relies upon threads and blocks to expose the graphics card parallel architecture to the developer. Its programming model is represented in Figure 4. In CUDA parlance, the CPU and RAM are known as the host, while the GPU and VRAM are known as the device. The host code that wants to interact with the device launches kernels. These kernels are the functions executed in the device. CUDA comes in several dialects, but the two principal ones are CUDA C++ and CUDA Fortran. Both are extensions of C++ and Fortran, respectively, that add new syntax to those languages. CUDA functions exist alongside C++/Fortran code and are prefixed with the `__global__` or `__device__` keywords. Code 1 shows what a basic kernel looks like in the CUDA C++ dialect. The `__global__` specifier tells the compiler that `kernel_example` is a CUDA function. This kernel adds to the output vector the contents of the input vector. The execution is divided into multiple subunits called threads, which will be explained further below.

[2] The compiler can be instructed to bypass the L1 cache by using the flags "`-dlcm={ca,cg,cv}`" [12].

[3] NVIDIA OptiX Ray Tracing Engine. `https://developer.nvidia.com/rtx/ray-tracing/optix`

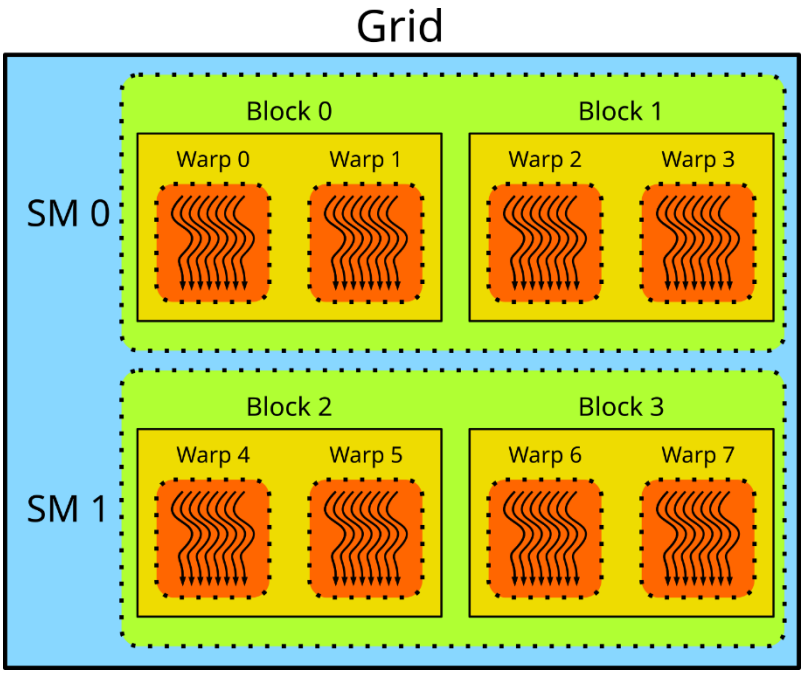


Figure 4. CUDA programming model.

```
1 __global__ void kernel_example(int *output, int *input)
2 {
3   output[threadIdx.x] += input[threadIdx.x];
4 }
```

Code 1. CUDA C++ kernel example.

CUDA kernels consist of a function that is launched once but executed many times in parallel. Each execution is known as a CUDA thread (marked as a curved arrow in Figure 4). Each one of these threads is executed in a CUDA core. A CUDA core is the threads' counterparts in the GPU hardware as previously shown in Section 2.1.1. A group of 32 consecutive threads is a warp and all threads within a warp must work in unison. That is, instructions are issued to warps, and they execute the same function regardless of the processed data. Sometimes threads must share information. This is done by reading a memory area shared by all threads. If a thread reads a value, it must have the certainty that the other thread that has computed that value has done so. This is where synchronization becomes an issue. CUDA provides the `__syncthreads` call to synchronize all threads in a block (see Section 2.1.7). A block is a group of threads residing in a SM that can share information efficiently, and a warp cannot contain threads that belong to different blocks. In Figure 4, the dashed lines represent the Streaming Multiprocessors (SM) and the warps. These abstractions are not part of the programming model but are hardware abstractions as previously shown in Section 2.1.1.

2.1.2.1 CUDA Kernel Execution

CUDA uses by default a queued execution model. Kernels are launched sequentially, and its Application Programming Interface (API) ensures that the next kernel executes after the previous

one has finished. Section 2.1.2 detailed the CUDA programming model and mentioned how blocks can synchronize themselves seamlessly. The only way to globally synchronize kernels that works under all conditions is to use multiple kernel launches, but there are alternatives. The new synchronization possibilities are further detailed in Section 2.1.7.

When kernels are launched in CUDA, it must be specified the number of blocks, the number of threads per block, and optionally the dynamically allocated shared memory available and the stream number. The syntax to call a CUDA kernel is shown in Code 2. Only the first two parameters are required, while the amount of dynamically allocated shared memory and stream number default value is 0.

```
1 kernel_example<<<blocks,threads_per_block, shared_memory,stream>>>(args)
```

Code 2. Kernel invocation syntax.

The maximum number of threads per block is 1024 [12]. This is a hard limit imposed by the architecture. On the other hand, the maximum number of blocks per kernel launch is $2^{31} - 1$. The block and thread size can also be specified with a 3-element tuple. In this way, if the kernel performs a task that processes a surface or a volume, the developer can write the kernel with cleaner code. The limit of 1024 threads per block applies to the multiplication of all dimensions of the block size. As an example, the largest block can either be $(1024,1,1)$ or $(32,32,1)$. The maximum number of y and z blocks is $2^{16} - 1$. Kernel launching has some overhead and it should be avoided to launch many kernels.

Dynamically shared memory is a chunk of shared memory (see Section 2.1.3) whose size is not hardcoded. They are declared as a vector with no size and prefixed with the `extern __shared__` keywords. Streams are a feature of CUDA that enable different kernels to be executed in parallel. Each stream is an independent queue, which means that if multiple kernels are launched in the same queue, the CUDA platform will not launch the latter blocks before all threads of the former blocks have completed. Kernels can also be called from other kernels by taking advantage of dynamic parallelism. Dynamic parallelism suits irregular and hierarchical workloads, such as recursive functions which further refine the results, but the grade of refinement is different for each section.

2.1.3. CUDA Memory Model

Memory transactions are key to craft performant kernels. As was mentioned in the previous section, the fastest memory that is available to CUDA cores are registers. Registers, like their CPU counterpart, are woven into the GPU. The floating-point units and the integer processing units use the values contained in the registers directly. Registers can only be accessed by their own thread on principle, but with warp intrinsics (see Section 2.1.7), the same register across all threads in the warp can exchange information with each other.

The next storage type is shared memory, which is implemented in the L1 cache for all GPUs from the *Ampere* architecture [18]. This type of memory is designed to be read from all threads from a block efficiently and fast, although particular care must be taken to avoid bank conflicts, which happens when two threads in the same warp interact with the same memory bank, and that slows down memory access transactions. Memory is divided in CUDA into 32 banks whose

elements are interleaved. Memory access to each bank is illustrated in Figure 5. Each bank element size corresponds to the size of a FP32 (4 bytes). On the left, each thread accesses an element of the vector. This vector is 32 elements long and each element lies in a different memory bank. The 32nd element would be in bank 0, the 33rd on bank 1, and so forth. In this situation there are no bank conflicts. On the center, each thread accesses an even element. As it can be seen, some banks are accessed twice, and their memory accesses must be serialized, halving the shared memory performance. Finally, on the right, threads are accessing the memory in a seemingly random pattern, but no bank conflicts occur, resulting in the same performance as on the left side.

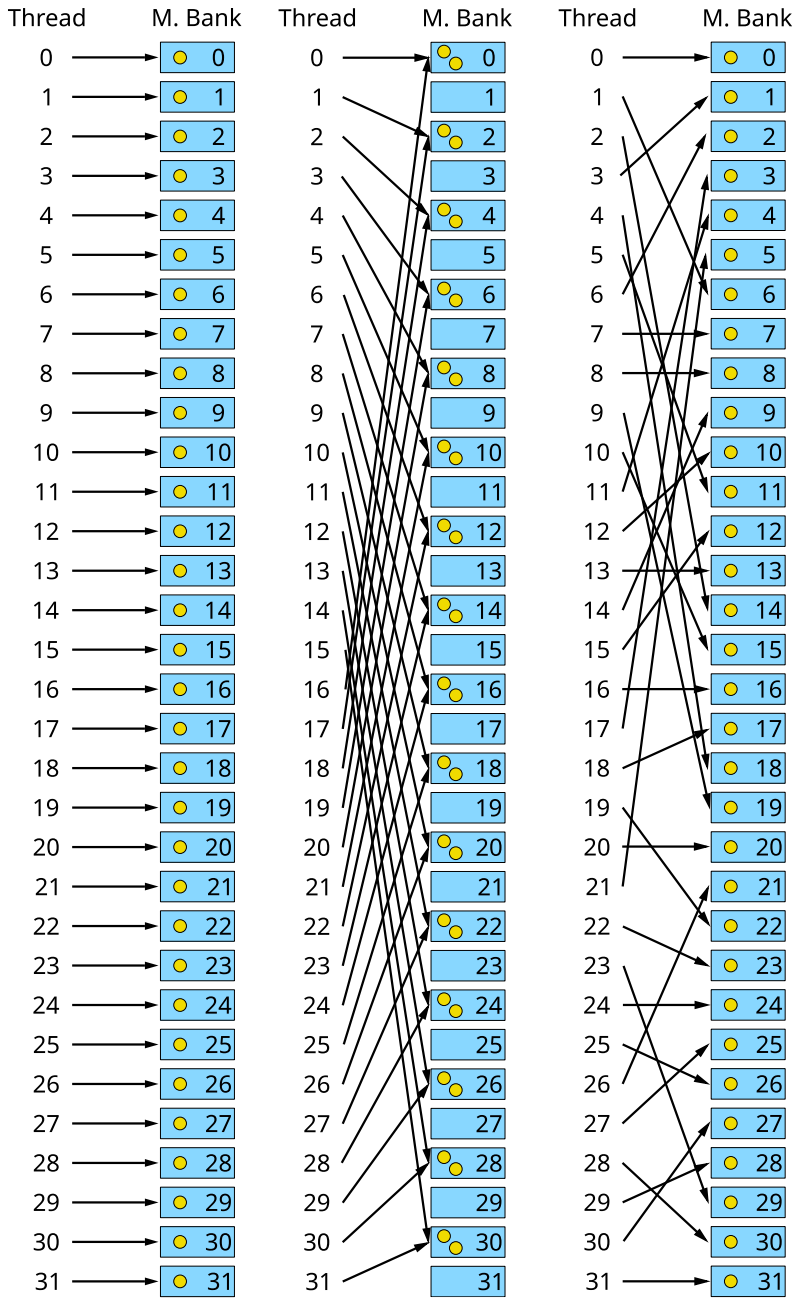


Figure 5. Example of bank conflict (adapted from [12]).

The scope of the previous memories is limited, and their size can influence the number of warps that can be executed on each GPU. The larger the shared memory is, or the larger the number of registers a kernel uses, the lower the number of blocks that will be assigned to be executed concurrently to the SMs [12].

The next type of memory, and largest one, is global memory. Global memory resides in the VRAM and can be used by all threads. Access to it is cached, but even then, its latency is hundreds of times larger than that of registers. To read this type of memory efficiently, all accesses (read or write) must be coalesced. That means, all threads within a warp must access values from the VRAM that are close together so that memory transactions can be reduced. CUDA uses a weak memory model. Load and store operations can be reordered and there is by default no guarantee that other threads will see ordered access unless it is explicitly used instructions with release and acquire modifiers, as it will be explained in Section 2.1.7.

There are other types of memory, such as local memory, that resides alongside global memory but can only be used in a single thread, and constant memory, that is read-only but also optimized for threads that read the same location at the same time; or texture memory, that feature value interpolation built into the memory read. These types of memory were not used while developing the kernels presented in this thesis.

2.1.3.1 Host-Device Memory Transfers

No type of CUDA memory can be accessed from host, and kernels cannot access RAM. The main function to perform memory transfer is `cudaMemcpy`. This function copies a chunk of data from RAM to VRAM, and the reverse. This function has implicit synchronization and locks the stream 0 (see Section 2.1.2). If it is needed to copy data asynchronously or is necessary to copy data synchronously in another stream, the function `cudaMemcpyAsync` is available. This method of data transfer is the fastest one, although its drawbacks are its lack of parallelism with kernel execution, and the overhead incurred when calling a `memcpy`-like function. Another downside is that managing memory that uses this pattern is more complicated for the developer. Its allocation and lifetime must be managed manually with the `cudaMalloc` and `cudaFree` functions. The CUDA API frees any still allocated memory at the end of the program. The lack of parallelism of memory copying can be mitigated using memory streams. The kernel can be split into multiple kernel launches, each launch with a part of the total block count. Before each kernel launch, a call to `cudaMemcpyAsync` is performed. The GPU copying engine is shared across all streams, and these calls are queued. However, when one copying completes, the following kernel can be executed while the next copying is currently happening.

The second memory copying scheme available is known as Zero-Copy Pinned Mapped Memory (ZCPMM). ZCPMM allows the GPU to access the RAM directly. This type of memory must be allocated using the `cudaMallocHost` function, using the `cudaHostAllocMapped` flag. The host can then use the allocated memory as usual. After that, a new pointer is obtained with the `cudaHostGetDevicePointer` function, which returns a direction that can be passed and used by the CUDA kernel. The GPU then requests the appropriate memory pages from RAM as data is needed. Memory transactions must be kept at a minimum, because if memory

access is scattered, performance plummets. This type of memory transfer can be as fast as `cudaMemcpy` if memory is accessed contiguously.

The final type of memory copying scheme is called Unified Memory (UM). This type of memory can be addressed by any processor in the system (CPU or GPU) and is allocated using `cudaMallocManaged` and freed with `cudaFree`. In GPUs with architecture *Pascal* or newer, when they access UM data for the first time, it triggers a page fault, which then compels the memory controller to copy only the memory pages touched. On previous architectures, all pages of memory were migrated to the GPU before executing the kernel should this memory was used. The device tries to keep data locality to prevent too many page faults. This type of memory is the easiest to manage out of all the CUDA memory schemes. Its downsides are the hidden overheads caused by page faults, which can hamper kernel performance.

The three memory copy schemes were tested during the research carried out for this thesis and it was shown that manual allocation was the most performant solution speed-wise. ZCPMM achieves copying speeds like the traditional method if data locality is maintained and, therefore, memory transactions are reduced. However, irregular accesses make this type of memory transfer the lower performant. UM was slightly less performant than both manual copying and ZCPMM at its best, but it was able to perform better than ZCPMM with irregular read patterns.

2.1.4. CUDA Compute Capability

Internally, the compiler `nvcc` converts the special kernel launch syntax into calls to `cuLaunchKernel`, which is the function that performs the kernel call. This function also checks that the kernel can be launched in the GPU by checking GPU's Compute Capability (CC). CC is a versioned specification of features, size limits, and instruction sets by NVIDIA that all its CUDA-capable graphics cards adhere to. Kernels must be compiled against a specific CC and code must be generated separately for each CC. The latest CC version is 9.0, while the latest version of `nvcc` compiles for at least is CC 5.0 [12]. Table I shows the new features that the Compute Capability guarantees a graphics card to have. Features unlisted are assumed to be available on all CCs.

Table I has columns for CC 5.x to CC 9.0. These range of CCs correspond to the available targets of the `nvcc` compiler, `release 12.1`. CC compilations are forward compatible but not backward compatible. A CUDA program built targeting a lower CC with an older CUDA toolkit will run on newer platforms, yet an application compiled with a newer toolkit might not run on old platforms. On some cases, a CUDA kernel designed for an older CC might run on newer platforms but slower. Consider Table II, which lists some of the minimal technical specifications that a CC guarantees. The parameters that more profoundly influence the design of the kernel, such as the memory bank size or the warp size, are kept constant and are not likely to change in the future. On the other hand, the values which depend on the number of units built into the GPU vary. For example, the values listed for maximum number of resident blocks per SM and the maximum amount of shared memory per SM shown in Table II correspond to the highest range graphics card that supported their corresponding CC. It can also be shown that the specifications do not necessarily get higher the newer the CC is. NVIDIA graphics card designers might optimize for a new generation of graphics cards other areas at the expense of others, and Table II shows it.

Table I. Compute capability feature list (adapted and abridged from [12]).

<table>
<tr><th rowspan="2">Feature</th><th colspan="5">Compute Capability</th></tr>
<tr><th>5.x</th><th>6.x</th><th>7.x</th><th>8.x</th><th>9.0</th></tr>
<tr><td>Half-precision floating-point operations: addition, subtraction, multiplication, comparison, warp shuffle functions, conversion</td><td>No</td><td colspan="4">Yes</td></tr>
<tr><td>Atomic addition operating on 64-bit floating point values in global memory and shared memory (<code>atomicAdd()</code>)</td><td>No</td><td colspan="4">Yes</td></tr>
<tr><td>Atomic addition operating on float2 and float4 floating point vectors in global memory (<code>atomicAdd()</code>)</td><td colspan="3">No</td><td colspan="2">Yes</td></tr>
<tr><td>Tensor Cores</td><td rowspan="2">No</td><td colspan="4" rowspan="2">Yes</td></tr>
<tr><td>Mixed precision warp-matrix functions (Warp matrix functions)</td></tr>
<tr><td>Bfloat16-precision floating-point operations: addition, subtraction, multiplication, comparison, warp shuffle functions, conversion</td><td colspan="3" rowspan="4">No</td><td colspan="2" rowspan="4">Yes</td></tr>
<tr><td>Hardware-accelerated <code>memcpy_async</code> (Asynchronous data copies using <code>cuda::pipeline</code>)</td></tr>
<tr><td>Hardware-accelerated Split Arrive/Wait Barrier (Asynchronous barrier)</td></tr>
<tr><td>L2 cache residency management (Device memory L2 access management)</td></tr>
<tr><td>DPX instructions for accelerated dynamic programming</td><td colspan="3" rowspan="4">No</td><td rowspan="4">Yes</td></tr>
<tr><td>Distributed shared memory</td></tr>
<tr><td>Thread block cluster</td></tr>
<tr><td>Tensor memory accelerator unit</td></tr>
</table>

Table II. Technical specifications depending on the compute capability (adapted and abridged from [12]).

<table>
<tr><th rowspan="2">Feature</th><th colspan="14">Compute Capability</th></tr>
<tr><th>5.0</th><th>5.2</th><th>5.3</th><th>6.0</th><th>6.1</th><th>6.2</th><th>7.0</th><th>7.2</th><th>7.5</th><th>8.0</th><th>8.6</th><th>8.7</th><th>8.9</th><th>9.0</th></tr>
<tr><td>Maximum number of residents grids per device</td><td colspan="2">32</td><td>16</td><td>128</td><td>32</td><td>16</td><td>128</td><td>16</td><td colspan="6">128</td></tr>
<tr><td>Maximum amount of shared memory per SM (kB)</td><td>64</td><td>96</td><td colspan="2">64</td><td>96</td><td>64</td><td colspan="2">96</td><td>64</td><td>164</td><td>100</td><td>164</td><td>100</td><td>228</td></tr>
<tr><td>Maximum number of resident blocks per SM</td><td colspan="8">32</td><td>16</td><td>32</td><td colspan="2">16</td><td>24</td><td>32</td></tr>
<tr><td>Maximum number of resident warps per SM</td><td colspan="8">64</td><td>32</td><td>64</td><td colspan="3">48</td><td>64</td></tr>
<tr><td>Maximum number of 32-bit registers per thread</td><td colspan="14">255</td></tr>
<tr><td>Maximum x- or y-dimensionality of a block</td><td colspan="14">1024</td></tr>
<tr><td>Maximum z-dimension of a block</td><td colspan="14">64</td></tr>
<tr><td>Number of shared memory banks</td><td colspan="14">32</td></tr>
<tr><td>Maximum number of threads per block</td><td colspan="14">1024</td></tr>
<tr><td>Warp size</td><td colspan="14">32</td></tr>
</table>

2.1.5. CUDA Compiler and Assembly Languages

The kernels presented in this thesis were developed using the CUDA/C++ dialect and when CUDA is mentioned in this thesis, it refers to its C++ dialect. CUDA code is compiled by the `nvcc` compiler, which only processes both C++ and CUDA code. To better understand how the compiler works, Figure 6 represents the compilation stages of a CUDA program.

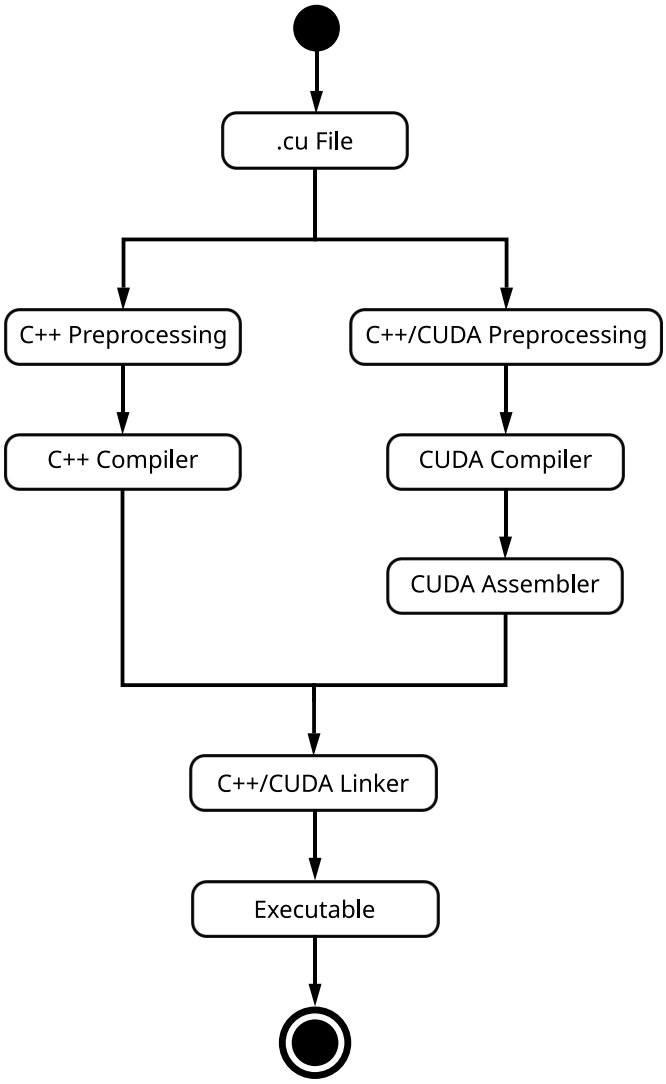


Figure 6. NVCC compilation flowchart.

The `nvcc` compiler is the starting point. To compile a `.cu` file the developer calls this program and passes any relevant compilation flags. The compiler identifies which parts of the code are kernels (see Code 1) and which ones are C++ code. A preliminary C++ pre-processing replaces CUDA syntactic sugar that facilitates kernel launch in C++ with calls to `cuLaunchKernel` (see Code 2). After the pre-processing, the C++ part of the file is built using the `cudafe++` utility, which internally calls an associated compiler, which is `gcc` or `msvc` depending on the platform.

The CUDA kernel parts of the code are pre-processed as well. The pre-processing is performed for each targeted CC. The final executable will have as many copies as CC architectures were targeted. The `nvcc` default CC is 5.2. The kernels are compiled internally by

`cicc`, which outputs Parallel Thread Execution (PTX) virtual GPU code. This is not the final machine code but an optimized bytecode. Then, the PTX code is assembled into Streaming Assembler (SASS) code. This is the code that runs directly on the GPU. Finally, the CUDA linker embeds the compiled CUDA into the C++ to call it at `cuLaunchKernel`.

To illustrate the differences between PTX and SASS code, the kernel shown in Code 3 was compiled using `nvcc` targeting CC 8.6. The kernel adds two vectors elementwise and each thread processes only one value. Their PTX and SASS disassembled code is shown in Code 4 and Code 5 respectively. The PTX code first loads the parameters into registers using the `ld` instruction. It is added after the name of the function the size of the operands and/or their position. It loads a parameter, hence the `param` suffix. This kernel was compiled in a 64-bit machine and so are the pointers. The `cvta.to.global` instruction converts the pointers' addresses into generic addresses [22]. After that, it performs a multiplication and addition to find the input and output data addresses and uses the `ld` and `st` functions to load and store the elements of the vector. The `global` suffix indicates that they interact with the global memory. The CUDA PTX bytecode is well documented and more information about its programming model can be found in [22]. On the other hand, the SASS specification is closed source. NVIDIA only gives a one-line description of each instruction and no description of either the registers or the special registers, as they are called in [23]. The SASS code has been further optimized, as there is no one-to-one relationship between PTX and SASS code. With the brief descriptions of the SASS instructions featured in [23], and relying on the previously built PTX code, the SASS code might be understood. However, the lack of tools to do so shows that PTX and the SASS are both key to fully understand the GPU kernel execution.

```
1 __global__ void addKernel(int *c, const int *a, const int *b)
2 {
3     int i = threadIdx.x;
4     c[i] = a[i] + b[i];
5 }
```

Code 3. Example CUDA kernel.

2.1.6. CUDA Atomics

CUDA exposes many parallel units of computation. These computation units might want to update the same value. For example, adding one to a variable in the global memory. Doing so traditionally requires reading the value to a register, performing the operation in the register, and writing back the updated value. If a second thread reads the value before the first has written the updated value, the update of the former thread will be lost. This condition is known as a data race. The CUDA platform aims to solve this issue with the aid of atomic operations.

Atomic operations were introduced with CC 1.1. CC 1.2 added the ability to perform atomic operations on shared memory, and graphics cards that supported CC 2.0 were able to perform these operations on floating point data. CC 6.0 added efficient atomic functions for double precision floats. The new frontiers in CUDA atomics are the inclusion of routines that manage vector types (`float2`, `float4`) (see Section 2.1.8).

```
1  .visible .entry _Z9addKernelPiPKiS1_(
2   .param .u64 _Z9addKernelPiPKiS1__param_0,
3   .param .u64 _Z9addKernelPiPKiS1__param_1,
4   .param .u64 _Z9addKernelPiPKiS1__param_2
5  )
6  {
7   .reg .b32  %r<5>;
8   .reg .b64  %rd<11>;
9    ld.param.u64 %rd1, [_Z9addKernelPiPKiS1__param_0];
10   ld.param.u64 %rd2, [_Z9addKernelPiPKiS1__param_1];
11   ld.param.u64 %rd3, [_Z9addKernelPiPKiS1__param_2];
12   cvta.to.global.u64  %rd4, %rd1;
13   cvta.to.global.u64  %rd5, %rd3;
14    cvta.to.global.u64  %rd6, %rd2;
15    mov.u32 %r1, %tid.x;
16    mul.wide.s32  %rd7, %r1, 4;
17    add.s64 %rd8, %rd6, %rd7;
18    ld.global.u32 %r2, [%rd8];
19    add.s64 %rd9, %rd5, %rd7;
20   ld.global.u32 %r3, [%rd9];
21   add.s32 %r4, %r3, %r2;
22   add.s64 %rd10, %rd4, %rd7;
23   st.global.u32 [%rd10], %r4;
24   ret;
25 }
```

Code 4. CUDA PTX disassembly.

```
1  Function : _Z9addKernelPiPKiS1_
2  .headerflags @"EF_CUDA_TEXMODE_UNIFIED EF_CUDA_64BIT_ADDRESS EF_CUDA_SM86 EF_
3  MOV R1, c[0x0][0x28] ;
4  S2R R6, SR_TID.X ;
5  MOV R7, 0x4 ;
6  ULDC.64 UR4, c[0x0][0x118] ;
7  IMAD.WIDE R2, R6, R7, c[0x0][0x168] ;
8  IMAD.WIDE R4, R6.reuse, R7.reuse, c[0x0][0x170] ;
9  LDG.E R2, [R2.64] ;
10 LDG.E R5, [R4.64] ;
11 IMAD.WIDE R6, R6, R7, c[0x0][0x160] ;
12 IADD3 R9, R2, R5, RZ ;
13 STG.E [R6.64], R9 ;
14 EXIT ;
15 BRA 0xc0 ;
16 NOP ; [Repeated 11 times]
```

Code 5. CUDA SASS disassembly.

The atomic operation reads, updates, and writes the variable in memory with a single step. The atomic operations performed in global memory bypass the L1 cache and are resolved directly by the L2 cache memory controller. The documentation does not give details about the exact hardware implementation of this, but it probably consists of a queue of memory operations that get resolved. This queueing greatly hampers the execution of the kernels, as the operations get serialized, the load and store units get bottlenecked, and parallelism is lost. They should be used sparingly.

Atomic operations were used twice during the kernels implemented in this thesis: an `atomicAdd` to update the LU matrix decomposition (see Chapter 6), and implicitly by the cooperative groups' technologies, which is key to an efficient grid-wide thread synchronization. More on cooperative groups can be found in Section 2.1.7.

2.1.7. CUDA Synchronization

One of the most prominent issues regarding CUDA programming is synchronization. At the bottom level, the threads within a warp are executed in lockstep and are, therefore, guaranteed to be synchronized. Blocks, which are groups of threads that can share memory efficiently, must be explicitly synchronized via the `__syncthreads` function so that the rest of threads see their memory operations, and to ensure that all threads have reached that point. On the other hand, grid-wide synchronization (i.e., synchronization of all launched threads) traditionally relies on implicit kernel synchronization, which is achieved by splitting the kernel into individual pieces that are called from the host or the device. To keep this data, global memory must be used which, despite its lower performance compared to shared memory and registers, is kept across kernel calls.

To solve this problem, NVIDIA introduced in CUDA 9 the cooperative groups. Cooperative groups allow synchronization of custom groups of threads that can span the whole grid (all threads that execute on the GPU), or even various GPUs. Grid-wide synchronization is available only in NVIDIA GPU architectures that support CC 6 or beyond. CUDA cooperative groups' grid-wide sync uses internally a highly optimized release-acquire mutex available only in GPUs that support CC 6 to count all blocks that arrived at the sync operation. The inner workings of the cooperative groups' technology are shown in Code 6. Code 6 features an adapted version of the internal CUDA API call that synchronizes all threads of a grid. The key parts of the kernel are the PTX instructions `atom.add.release.gpu.u32` and `ld.acquire.gpu.u32`. The former adds 1 to a temporary value that holds the number of blocks that have already arrived and makes sure that other threads see this operation before all other previous writing operations. The latter loads a value and ensures that later memory and writings will not occur before those instructions. The `release.gpu` suffix represents the set of all threads in the current program executing on the same compute device as the current thread. This also includes other kernel grids invoked by the host program on the same compute device [22].

This new operation compares favorably to methods based on manually updating a variable in global memory using atomic operations [24]. The cooperative groups' technology is key to deliver performant kernels in this thesis. This feature allows to launch a single, monolithic kernel that runs until the end of the simulation, slashing the number of kernel invocations from host code, and thus greatly reducing kernel launch overhead.

Therefore, there are three synchronization methods available: block-level synchronization with `__syncthreads`, multiple kernel invocations both host-side and client-side from another CUDA kernel, and CUDA cooperative groups. Works like *Zhang et al. (2020)* [25] seem to suggest that cooperative groups have a lower performance than multiple kernel launches. However, this affirmation is not universal because as it will be shown, the lowest performant strategy for the kernels developed in this thesis is the multiple kernel launches method, and the

fastest synchronization method is using `__syncthreads` but, as it will be shown later, it is needed to synchronize all blocks of the grid, and therefore this method cannot always be relied on.

```
1  __device__ void grid_sync(volatile barrier_t *arrived)
2  {
3      unsigned int expected = gridDim.x * gridDim.y * gridDim.z;
4
5      bool cta_master = (threadIdx.x + threadIdx.y + threadIdx.z == 0);
6      bool gpu_master = (blockIdx.x + blockIdx.y + blockIdx.z == 0);
7
8      __syncthreads();
9
10     if (cta_master) {
11         unsigned int oldArrive;
12         unsigned int nb = 1;
13         if (gpu_master) {
14             nb = 0x80000000 - (expected - 1);
15         }
16
17              asm volatile("atom.add.release.gpu.u32  %0,[%1],%2;" : "=r"(oldArrive) :
_CG_ASM_PTR_CONSTRAINT((unsigned int*)arrived), "r"(nb) : "memory");
18
19         unsigned int current_arrive;
20         do {
21                     asm volatile("ld.acquire.gpu.u32  %0,[%1];" : "=r"(current_arrive) :
_CG_ASM_PTR_CONSTRAINT((unsigned int *)arrived) : "memory");
22         } while (!bar_has_flipped(oldArrive, current_arrive));
23
24     }
25
26     __syncthreads();
27 }
```

Code 6. Cooperative groups grid sync.

The differences between the results obtained in this thesis and in [25] might be caused by the larger number of resources used by the CUDA kernels. The authors of [25] micro-benchmarked these strategies using a vector reduction kernel, which is a kernel that has very few input arguments and little register usage. Start-up kernel code related to argument initialization might be to blame for the high kernel launch penalty in their work.

Cooperative groups' synchronization was chosen for the simulation of multiple wind turbines within a PS. The advantages of one unified kernel outweigh the performance bonus of multiple kernel launches. Furthermore, each kernel launch is a blank slate, and at its beginning it is needed to load data into shared memory to achieve memory read performance.

There are, however, many downsides to the cooperative groups, being the main one the limited grid size. During grid-wide kernel synchronization, all launched warps wait to all other warps to arrive at the memory barrier. This fact, along with the fact that the CUDA execution model is non-preemptive [12], means that the GPU scheduler is the sole responsible of setting which warp will be executed next. Due to this architectural feature, there is no mechanism to yield the currently executed warp to run another warp in another block. It is the job of the GPU schedulers to swap currently running if they are stalled. The GPU scheduler allocates the resources to run a certain number of warps belonging to different blocks in parallel. If a SM cannot run more warps

in parallel, it will not be assigned another block. Therefore, if all SMs are being fully occupied and all warps are waiting for all warps to reach the synchronization barrier, the kernel will enter a deadlock, as the queued blocks will not have the chance to be executed. Prior to the introduction of cooperative groups, works such as *Xiao et al. (2010)* [26] and *Anand et al. (2019)* [27] proposed synchronized barriers that acknowledged this restriction. A one-to-one mapping between CUDA blocks and SM was enforced to prevent a deadlock. This is the reason kernels using cooperative groups must be launched with the `cudaLaunchCooperativeKernel` function. This function checks whether the GPU has enough resources to fit all blocks in the SMs at the same time and launches them.

Despite all these restrictions, cooperative groups enable fine synchronization among blocks, and prevent kernel launch overhead. Furthermore, as all individual kernel launches can be merged into a large and monolithic kernel, the kernel can be further optimized, as shared memory can be used, because the lifetime of this memory is equal to the lifetime of the block.

There is a final type of warp-level synchronization known as warp-level intrinsics [12]. All registers of all threads are in the same register file. Registers are preassigned a location in the register file before execution and warp intrinsics can access the registers of another thread. These type of intrinsics need no synchronization since warps always run in lockstep (see Section 2.1.1).

2.1.8. CUDA Vector Types

Many properties are described as a vector. Some examples are positions in 3D space, which are described with 3 values; and color in the RGBA space, with 4 values. If a parallel algorithm tries to process many of these values at once, they usually read a component from memory in parallel. As mentioned in Section 2.1.3, coalescent accesses are key to achieve high parallelism and kernels should be designed to achieve as much coalescence as possible. This rule of thumb suggests a kernel design where the vector components are laid out with a stride equal to the total number of elements to be processed. This layout requests the minimal global memory sectors, and it is indeed a performant solution. As an example, Figure 7 shows how the individual values of an array made from RGB values can be arranged in memory. Each value is made of 3 floating point values. If their values are grouped by value, the memory will, in theory, perform more memory requests per operation, harming performance. The optimal solution is to pack all components of composite types together as shown in the latter layout of Figure 7.

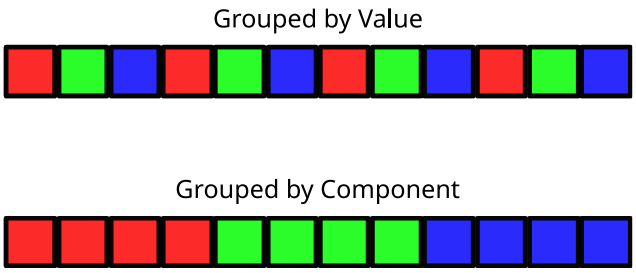


Figure 7. Aggrupation strategies for array of RGB values.

To reduce the number of requests of some composite types, the CUDA programming language provides fixed length vector types. These vector types are denoted by their base type plus its component number. The CUDA platform offers vectors of up to 4 components, and the available

vector types are `char`, `short`, `int`, `long`, `longlong`, `ulonglong`, `float` and `double`. Reading and writing these vector types is optimized, as the GPU can request multiple vectors values at once, thus avoiding gaps among the memory requests, and coalescing them. The kernel shown in Code 7 reads a `float2` value, modifies it, and writes it back.

```
1 __global__ void kernel_float2(float2* res)
2 {
3  float2 temp;
4  temp=res[threadIdx.x];
5  temp.x+=1;
6  temp.y+=2;
7   res[threadIdx.x]= temp;
8 }
```

Code 7. Float2 value processing example.

The PTX disassembly is shown in Code 8. The global memory read and write operations are performed by the `ld.global.v2.f32` and `st.global.v2.f32` instructions, respectively. These instructions perform 64-bit memory reading requests, achieving coalescing access. These PTX instructions have their `LDG.E.64` and `STG.E.64` counterparts, which shows that the bytecode has a one-to-one equivalency to SASS code. Memory operations on 4-elements floating vectors emit the `ld.global.v4.f32` and `st.global.v4.f32` instructions. The CC 8.6 does not support unified vector load and store operations beyond 128 bits. Therefore, while processing a `double2` outputs a single instruction, reading a `double4` requires two `ld.global.v2.f64` instructions, thus losing coalescence.

```
1  ld.param.u64  %rd1, [_Z13kernel_float2P6float2_param_0];
2  cvta.to.global.u64  %rd2, %rd1;
3  mov.u32  %r1, %tid.x;
4  mul.wide.u32  %rd3, %r1, 8;
5  add.s64  %rd4, %rd2, %rd3;
6  ld.global.v2.f32 {%f1, %f2}, [%rd4];
7  add.f32  %f5, %f2, 0f40000000;
8  add.f32  %f6, %f1, 0f3F800000;
9  st.global.v2.f32 [%rd4], {%f6, %f5};
10 ret;
```

Code 8. PTX disassembly of float2 value processing kernel.

The vector types have been used to store complex quantities. CUDA offers the `cuComplex.h` header, which contains functions to perform basic arithmetic operations on complex numbers. The complex CUDA types are `cuFloatComplex` and `cuDoubleComplex`. They are implemented under the hood as `float2` and `double2` vectors, respectively. Complex numbers are a staple of mathematical constructs used in many scientific disciplines, including PS simulation. These complex magnitudes have been implemented as `float2` respectively, as it will be explained in Chapter 6.

2.1.9. Instruction Level Parallelism in CUDA

The CUDA platform delivers parallelism through the execution of many parallel threads. Their parallelism is enabled by the architecture of the GPU, as previously shown in Section 2.1.1. Even

then, a sequential piece of code can achieve mild parallelism through Instruction Level Parallelism (ILP) [28]. Machine code instructions take data as input from some registers, and outputs its results in other registers. If a subsequent instruction uses as input the output of a previous instruction, a data dependency exists among them. Modern CPU processors use advanced Out-of-Order Execution (OoOE) strategies to skirt around data dependencies, process more instructions per cycle, and increase processor performance. GPU's strength rests on its parallelism, and it has been prioritized to cram as many parallel computation units as possible at the expense of branch prediction or OoOE circuitry. Nevertheless, ILP should be considered to craft high-performant kernels.

Data dependencies prevent instructions from being executed in parallel. Compiling the kernels with NVCC emits PTX code that uses many placeholder registers, which could be reused to reduce register usage. The conversion to SASS performs this optimization, while also reordering some bytecode operations of the original PTX code. For this reason, the disassembly of SASS has been used in this section. Code 9 shows two CUDA kernels that highlight the use of ILP. Each kernel sums 8 values. Code 10 and Code 11 show the SASS disassembly of both kernels, with the register renaming and instruction reordering performed. The disassembly was abridged, as some instructions that load data from memory were removed for the sake of clarity. The registers have also been highlighted as well. The first kernel uses the register `R0` as an accumulator and sums the values there. It is obvious that there are data dependencies among all instructions, for they all depend on `R0`. There is no parallelism available. The second kernel performs the same operation, but a carefully chosen parenthesis instructs the compiler to perform these operations first. Machine code lines 1, 2, 4 and 5 can be processed in parallel. ILP has been considered while crafting the wind turbine model integrator, by multiplying separately the matrices A and B, as it will be shown in Section 6.1.4.

```
1  __global__ void kernel_ILP1_Example(float* output,float* input)
2  {
3   int position=threadIdx.x*8;
4   output[threadIdx.x]=
5     input[position+0]+input[position+1]+
6     input[position+2]+input[position+3]+
7     input[position+4]+input[position+5]+
8     input[position+6]+input[position+7];
9  }
10
11 __global__ void kernel_ILP2_Example(float* output,float* input)
12 {
13 int position=threadIdx.x*8;
14 output[threadIdx.x]=
15   ((input[position+0]+input[position+1])+
16   (input[position+2]+input[position+3]))+
17   ((input[position+4]+input[position+5])+
18   (input[position+6]+input[position+7]));
19 }
```

Code 9. CUDA kernels to test ILP.

```
1 FADD.FTZ R0, R0, R5;
2 FADD.FTZ R0, R0, R7;
3 FADD.FTZ R0, R0, R9;
4 FADD.FTZ R0, R0, R11;
5 FADD.FTZ R0, R0, R13;
6 FADD.FTZ R0, R0, R15;
7 FADD.FTZ R17, R0, R17;
```

Code 10. Abridged SASS disassembly of ILP kernel 1.

```
1 FADD.FTZ R0, R0, R5;
2 FADD.FTZ R7, R4, R7;
3 FADD.FTZ R0, R0, R7;
4 FADD.FTZ R6, R6, R9;
5 FADD.FTZ R11, R8, R11;
6 FADD.FTZ R11, R6, R11;
7 FADD.FTZ R11, R0, R11;
```

Code 11. Abridged SASS disassembly of ILP kernel 2.

2.2. Types of Power System Studies

Power systems (PSs) are one of the marvels of engineering. They interconnect billions of producers and consumers and span entire continents. Their goal is to generate, transport and distribute electricity to consumers safely, reliably, and cheaply in real time. Simulating their behavior is key to guarantee their stability. They are made of nodes (also called busbars, or buses), which are connected by power lines. Busbars are the points where generators and loads are connected or, more generally, where the voltage of this type of systems wants to be known. As an example, it is shown in Figure 8 the IEEE Test Feeder 123 [29]. Feeders are the distribution power lines that supply remote nodes and, as shown in Figure 8, there is a redundant ring connection surrounded by many feeder lines.

There are quite different phenomena that ought to be considered, and different study techniques to find out different properties of the PS. This section will enumerate the main PS study techniques. The distinct types of PS studies and simulation techniques can be divided into many distinct categories. Each type of analysis will be briefly introduced in the following subsections.

2.2.1. Power Flow Analysis

Power flow (PF) analysis, also called load flow analysis, is a numerical tool that seeks to understand how power flows through the PS. Its goal is to obtain the voltage of each busbar provided the active power of every connected generator and load is known [9]. The load flow problem defines three types of busbars: PQ, where the active and reactive power generation and consumption is known for all devices connected to it; PV, where a busbar features a power plant with unknown reactive power output, whose control system demands it keeps its busbar at a certain value; and slack, where the voltage of the busbar is set by the designer. It is necessary at least a slack bus to solve the load flow problem.

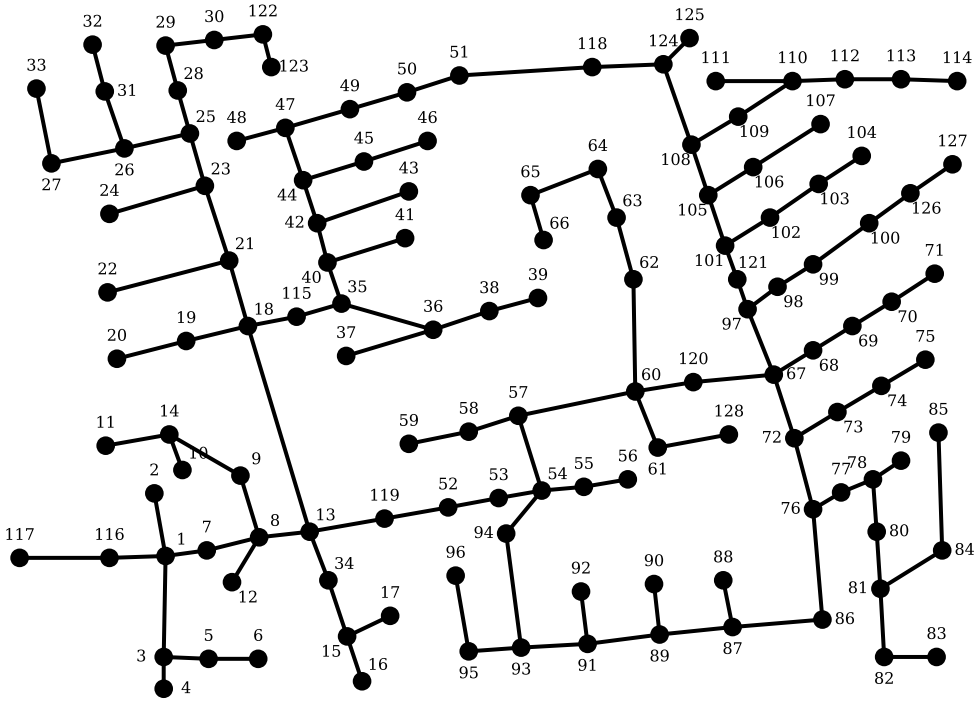


Figure 8. IEEE test feeder 123 (adapted from [29]).

Equations (1) and (2) are respectively the defining equations of the load flow problem, where P_i and Q_i are the injected active and reactive power at node i, $|V_i|$ is the modulus of the complex value of the voltage phasor at node i, and δ_i its angle. The modulus and angle of the admittance between node i and node k are represented by $|Y_{ik}|$ and θ_{ik} respectively. The variables P_{gi}, P_{di}, Q_{gi} and Q_{di} represent the generation and demand of active and reactive power at that node by external elements.

$$P_i = |V_i| \sum_{k}^{n} |V_k||Y_{ik}| \cos(\theta_{ik} + \delta_k - \delta_i) - P_{gi} + P_{di} \tag{1}$$

$$Q_i = -|V_i| \sum_{k}^{n} |V_k||Y_{ik}| \sin(\theta_{ik} + \delta_k - \delta_i) - Q_{gi} + Q_{di} \tag{2}$$

The expressions of Equations (1) and (2) are equal to zero and form a nonlinear system with $2n$ equations and $2n$ unknowns, where n is the number of nodes. In the initial stages of digital computers, the load flow system was solved using Gauss-Seidel iterations, but currently the prevailing method is the Newton-Raphson method. The Newton-Raphson method linearizes the matrices around a voltage approximation. The next approximation is calculated by solving the Equation (3), which corresponds to the multivariate Newton-Raphson method applied to Equations (1) and (2). These expressions are evaluated, and its Jacobian matrix is obtained around the values of the current iterations (see Section 3.8), which is the Jacobian Matrix of the nonlinear equations. The Jacobian and the linear vector form a linear system, which is solved, and its result is subtracted to the vector containing the modulus of the bus voltages and their angle. More on the methods to solve this linear system can be found in Sections 2.3.4 and 2.3.5. The process is repeated until convergence.

$$\begin{bmatrix}\theta_{n+1}\\ v_{n+1}\end{bmatrix} = \begin{bmatrix}\theta_n\\ v_n\end{bmatrix} - \begin{bmatrix}\frac{\partial P}{\partial \Delta\theta} & \frac{\partial P}{\partial \Delta \mathrm{v}}\\ \frac{\partial Q}{\partial \Delta\theta} & \frac{\partial Q}{\partial \Delta \mathrm{v}}\end{bmatrix}^{-1} \cdot \begin{bmatrix}P\left(\begin{bmatrix}\theta_n\\ v_n\end{bmatrix}\right)\\ Q\left(\begin{bmatrix}\theta_n\\ v_n\end{bmatrix}\right)\end{bmatrix} \quad (3)$$

2.2.2. Optimal Power Flow Analysis

The previous analysis calculated the state of the system under a set of operating conditions, yet for a given power consumption profile, some configurations are more efficient than others. Minimizing power loss is of utmost importance, and many PS analyses have appeared to address this issue. These studies are grouped under the umbrella of optimal power flow analysis. The goal of this study is to minimize the cost function shown in Equation (4), where P_i is the power generated by the power plant i, and C_i is the cost per MW of running that power plant. The defining equations of the conventional load flow analysis shown in Equations (1) and (2) are reinterpreted as constraints of the minimization problem. Additional constraints that set the lower and upper bounds of active and reactive power generation of each power plant are set as well. The optimization problem is currently solved using the interior-point method [30], which searches the solution iteratively. Given an initial guess that fulfils the constraints which is not the minimal, it computes the next step by solving a linear system, whose matrix is obtained with a method that is reminiscent from the Newton-Raphson method.

$$Total\ Cost = \sum_{i}^{n} C_i P_i \quad (4)$$

2.2.3. Continuation Power Flow Analysis

The techniques mentioned above can also be repurposed to study the stability of the power system against extraordinary circumstances. For instance, large increases of the load might jeopardize the whole PS. As an example, the simple 4-busbar power system proposed in [31] and pictured in Figure 9 was examined with a varying load. The load at the node 2 was varied with a λ parameter as shown in Equation (5). The term P_{2base} refers to the active power consumed by the load at node 2, while P_2 is its final active power. The larger the active power consumed at node 2, the lower the voltage. However, it reaches a point where the PS becomes unstable. Figure 10 shows that beyond a critical λ, the system becomes unstable. Near the critical λ the determinant of the Jacobian matrix becomes zero, thus making the system unsolvable. The solutions below are unstable, and the algorithm of convergence has difficulty converging to them. The continuation power flow analysis [32] is a tool that enables researchers and network operators to pinpoint the critical generation and avoid the singularity of the Jacobian matrix. This process reformulates Equations (1) and (2) in order to obtain the tangent of the PS curve shown in Figure 10. After that, this tangent value is used to get the next value of the network voltages. These network voltages are then fed to the Newton-Raphson algorithm as a first approximation.

$$P_2 = P_{2base}(1 + \lambda) \quad (5)$$

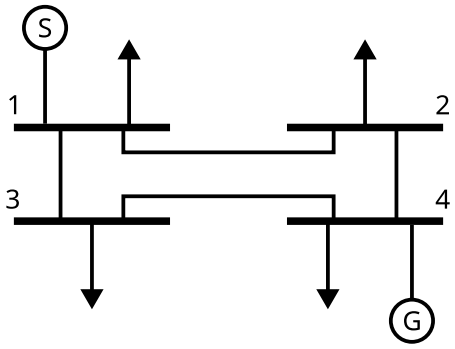


Figure 9. 4-busbar power system (adapted from [31]).

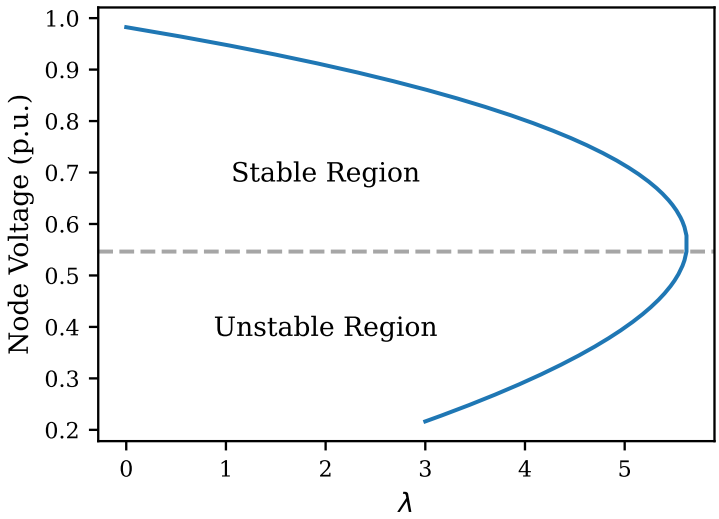


Figure 10. Unstable power system.

2.2.4. Small-Signal Stability Analysis

Small-signal stability analysis studies the ability of PS generators to restore the grid frequency. The backbone of PSs around the globe is the synchronous generator. These machines convert rotational energy into electrical energy at a certain voltage and frequency [33]. They always rotate at exactly the synchronous speed. Control systems in power plants with synchronous generators adjust the input mechanical power to keep the power grid frequency as close to 50 Hz or 60 Hz. Load shedding, a power line fault, or a power plant sudden disconnection could lead the generators into an unstable state. The model of a synchronous generator is nonlinear but can be assumed that with a small signal change around a point, it will behave linearly, and thus linearization is a valid approach to study the system easier thanks to control theory (more information on the process of linearization in Section 3.8). This linearization results in the system of Equations (6), where f and g are nonlinear functions, x is a vector that describes the internal state of the system, and y is a vector of dependent variables. The solution of the system (6) is Equation (7), and that matrix characterizes the behavior of the generator. The stability of the generator will depend on the eigenvalues of the matrix [34].

$$\begin{bmatrix}\Delta\dot{x}\\ 0\end{bmatrix} = \begin{bmatrix}\frac{\partial f}{\partial \Delta \mathrm{x}} & \frac{\partial f}{\partial \Delta \mathrm{u}}\\ \frac{\partial g}{\partial \Delta x} & \frac{\partial g}{\partial \Delta \mathrm{u}}\end{bmatrix} \cdot \begin{bmatrix}\Delta \mathrm{x}\\ \Delta \mathrm{y}\end{bmatrix} \tag{6}$$

$$\Delta\dot{x} = \left(\frac{\partial f}{\partial \Delta \mathrm{x}} - \frac{\partial f}{\partial \Delta \mathrm{u}}\left(\frac{\partial g}{\partial \Delta \mathrm{u}}\right)^{-1}\frac{\partial g}{\partial \Delta x}\right)\Delta \mathrm{x} \tag{7}$$

2.2.5. Electromagnetic Transient Simulation

The previous power analyses studied PS without considering transients of any kind. The internal conditions of generators and consumers change, and so does the state of the power grid, where a short circuit might take place. For that reason, it is necessary to carry out simulations that account for these events. Electromagnetic Transient (EMT) simulations have a long history, and have been studied since computers were within reach to simulate PSs [35]. EMT simulation models power lines as elements where its inductances and capacitances (see Chapter 5) have a state that changes with time. The differential equations that describe these elements are continuous, but can be discretized, as it will be shown in Section 5.5. The discretization step makes the system more stable and makes possible to integrate it with larger timesteps [36]. The discretization converts the equivalent coils and capacitors into resistances (or admittances, which is the inverse of the resistance) and current sources which depend on the previous value of the current and voltage. The values of all admittances of the system are organized into a matrix admittance as shown in Section 5.6, where each non-diagonal (i, j) element of the matrix corresponds to the admittance between the nodes i and j. The resulting system is a linear one solved for each timestep.

As an example, Figure 11 represents the matrix admittance of a system previously defined in Figure 8. The black points represent the nonzero values of the matrix. As nodes are only connected to other nodes in their vicinity (see Figure 8), the admittance between most nodes will be equal to zero. All previous types of studies which deal with matrices will present matrices with many zeros. These matrices are known as sparse matrices, and it will have profound consequences for the ways to parallelize the kernel.

2.2.6. Phasor Simulation

EMT simulation delivers accurate results for any type of input signal. Its largest downside is the small timestep required to solve it. EMT uses numerical integration methods, and the larger the timestep, the larger the approximation will drift. One key insight of transient PS simulation is that in the stationary state all magnitudes are sinusoidal signals. Its amplitude and phase shift can be represented as a complex number known as phasor. Phasor theory is well stablished and can be found in books such as, for instance, [9]. The differential equations of the inductance and capacitance of the power line are converted into algebraic equations with complex numbers. These algebraic equations are linear, and the resulting equations can be arranged as a linear system as shown in Equation (8), where $[Y]$ represents the matrix admittance of size n, V is the vector of n unknown complex voltages, and I represents the sum of the currents injected by current sources. The admittance matrix is in the likeness of the matrix of the EMT simulation.

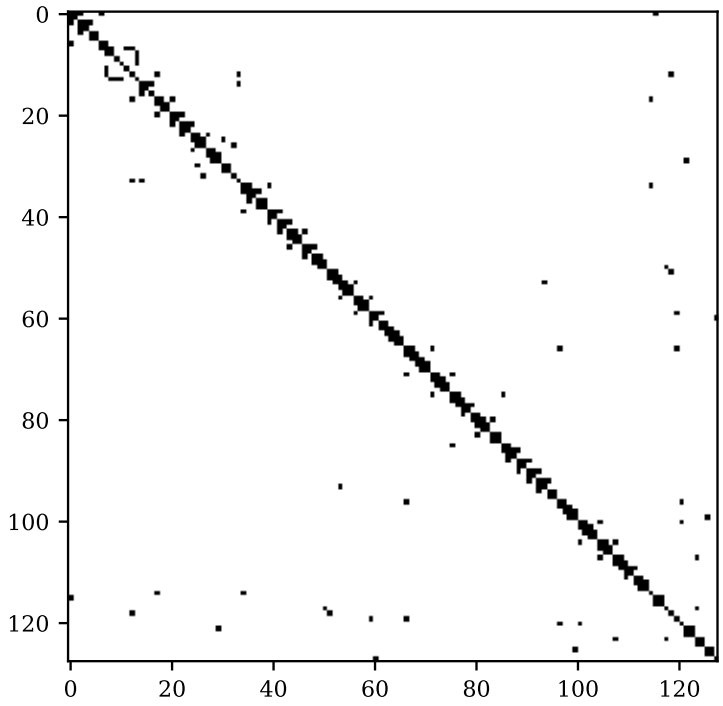


Figure 11. IEEE test feeder 123 admittance matrix.

$$[Y] \cdot V = I \tag{8}$$

The simulation, for each time step, does not consider individual transients, and can be regarded as a quasi-static simulation. Quasi-Static simulation refers to systems whose state only depends on its current conditions [37]. Equation (9) shows the general expression of quasi-static systems. f is a system of n equations and x is a vector of n unknown states. Finally, $u(t)$ represents the inputs of the system, which may or may not change in time.

$$f(x, u(t)) = 0 \tag{9}$$

2.2.7. Dynamic Phasor Simulation

Phasor simulation of PS calculates its stationary state with phasors. By their nature, phasors can only represent sinusoidal signals at a base frequency. With the appearance of High Voltage Direct Current (HVDC) and the advent of nonlinear generator and loads, PS harmonics have become an even more relevant topic. Harmonics are waves with a frequency that is a multiple of the fundamental frequency, i.e., the frequency of the PS. The effects of harmonics range from interferences and excessive power losses to motor failure and excessive wear in insulated cables, and for that reason the maximum harmonic content is regulated by the grid operators [38]. The equations that define power system grid are linear, as shown in previous sections, and a preliminary harmonic study can be performed by separating the input voltage and current signals into their pure sinusoidal components and then solving the PS for each frequency similarly to the phasor simulation mentioned in Section 2.2.6. Nonlinear generators work internally with semiconductor switches, which perform many switching events, whose frequency can reach up to

350 kHz [39]. Phasor simulation does not capture those transients, and EMT simulation requires a timestep too small. A compromise solution is the dynamic phasor simulation.

Dynamic phasor simulation was first applied for the study of inverters [39] and strikes a balance between transient simulation and static phasor simulation. It works by obtaining the frequency components of the electric signals of interest using a slide window. The slide window is the set of all immediate prior values of that signal. Consider an electrical signal $x(\tau)$ with $\tau \in [t-T, t]$, where t is the current timestep and T is the length of the slide window. Using a Fourier series, the signal $x(\tau)$, can be expressed as a function of the coefficients $X_k(t)$. Equation (11) shows how to calculate these coefficients, where k is the coefficient number and f_0 is the fundamental frequency of the PS. The electrical equations can be rewritten using the X_k coefficients, thus forming a PS problem that can be solved in time with higher timesteps.

$$x(\tau) = \sum_{k=-\infty}^{\infty} X_k(t) e^{j2\pi k f_0 \tau} \tag{10}$$

$$X_k(t) = \frac{1}{T} \int_{t-T}^{t} x(\tau) e^{-j2\pi k f_0 \tau} \, d\tau \tag{11}$$

2.2.8. Probabilistic Power System Analysis

This section concludes with a brief introduction to probabilistic power system analysis. The previously summarized analyses assume that the load profiles and the behavior and state of the PS are known in advance. Short circuits, which cause havoc on PS as they provoke power line shutdowns and sudden voltage variations, occur randomly. The first studies on probabilistic PS analysis [40] assigned a distribution function to the generated and consumed power of the PS devices and calculated the network voltages with their probability distribution. These ideas were extended by works such as [41], where the reliability of the PS is calculated while also suggesting where new power plants should be installed. Other works such as [42] address the consequences of short-circuits by using Monte-Carlo simulation, where many short-circuit events are simulated and then, their results are weighted, and so the authors obtained a histogram which shows the maximum voltage in each point of the PS during a short circuit.

2.3. Parallel Power System Math Operations

The previous section briefly introduced the different PS simulation strategies and the mathematical tools that underpin them. The different analyses' goals are disparate, yet their basic mathematical operations are similar. Most of them process matrices which must be solved. Graphics cards can deliver a large computational speedup, but the algorithms must be modified to suit their parallel architecture. The state-of-the-art of the methods to parallelize these simulation basic operations will be presented in the subsequent subsections.

First, the parallelization of the nonlinear models of the wind turbine will be considered, and then their linearized counterparts (see Section 3.10). The linearized models are systems of Ordinary Differential Equations (ODEs) which are expressed in a state-space linear system

formulation. These models are then integrated with numerical integration schemes such as Euler's method or Heun's method (see Section 3.9). These methods applied to state-space ODEs involve three operations: vector addition, vector multiplication by a constant, and matrix-vector multiplication, which are numerical procedures also discussed in this section.

The other mathematical operations used by the kernels proposed in this thesis are related to the solution of linear systems. Linear system resolution can be performed with either dense or sparse matrices. Each type of system is treated very differently, as it will be shown below.

2.3.1. Parallelization of Nonlinear Models

This thesis will study both wind farms and PVPPs. While each PV panel has been considered as a current source with no internal state (see Chapter 4), wind turbines are modelled as current sources whose output depends on its internal state (see Chapter 3). The models of the wind turbine are expressed as an explicit differential equation as shown in Equation (12), where the state x and the input u at a timestep t, affect the derivative of the states $\dot{x}$, and the output of the system y. The original wind turbine model is nonlinear, and it was considered whether to parallelize the model as is, or not. *Song et al. (2018)* [43] represent the equations that govern the models of the device connected to the power system as a dependence graph. In this graph, they extract which mathematical operations of the models can be parallelized using an automatic code tool. As an example, Figure 12 illustrates the partition of the sample mathematical expression $\frac{5y(e^{3z}-20)+77x}{-5+x}$. Operator precedence imposes an order of operations, and that order is the basis to build the execution tree of this operation. Some operations can be performed in parallel, and they can thus be parallelized. This venue of parallelism suffers two crucial issues: excess of needed synchronization and unequal distribution of work. Divergence, which is an issue that might appear when a kernel performs different operations with different threads, can be mitigated by grouping threads that are assigned the same work in the same warp. Even then, threads must share the computation results after each computation is done. Grouping threads by their work type means that threads pertaining to the same operation are far away, and thus a `__syncthreads` operation is required (see Section 2.1.7, Section 2.1.8 and Figure 7). Had quick synchronization been prioritized, data exchange should be performed with warp intrinsics in close threads (see Section 2.1.7).

$$\begin{cases} \dot{x} = F\big(x, u(t)\big) \\ y = G\big(x, u(t)\big) \end{cases} \tag{12}$$

The second complication is the unequal distribution of work. Figure 12 shows the use of a transcendental function, the exponential function. These functions are implemented as polynomial approximations[4]. These approximations take many Fused Multiply-Add (FMA) operations, which

[4] As an example, the algorithm that computes the exponential function in the Glibc library `https://www.gnu.org/software/libc/`, can be consulted at the source file "`sysdeps/ieee75/dbl-64/e_exp.c`"

takes much running time, and makes the rest of the threads idle. The computational cost of these functions can be reduced if the `-use_fast_math` flag is activated. This flag is passed to `nvcc` and instructs the compiler to change all functions and statements on the left column of Table III to their fast counterpart, shown on the right column. These special functions are processed by the SFUs (see Section 2.1.1). These functions have a lower precision, and the recommended procedure is to use the fast functions explicitly when precision is not of much importance. For these reasons, the parallel computation of nonlinear equations has not been considered for this thesis.

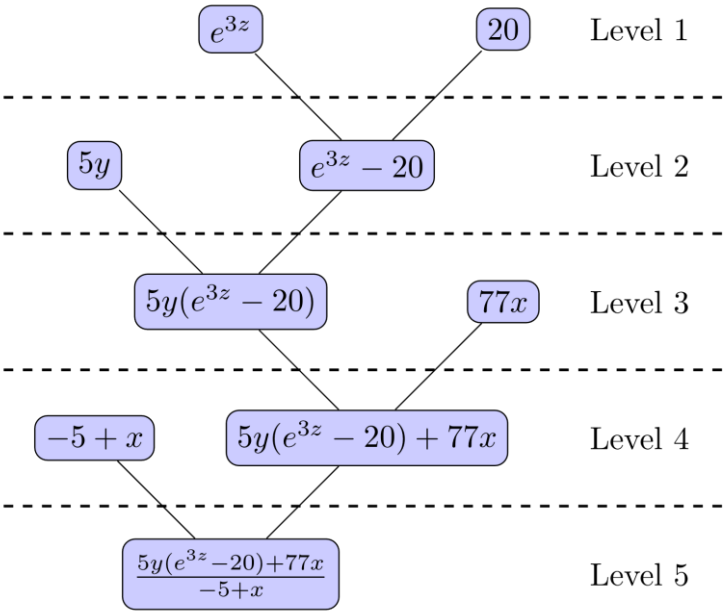


Figure 12. Example of math operation parallelism.

Table III. Fast math CUDA functions.

Original function	Fast function
`x/y`	`__fdividef(x,y)`
`sinf(x)`	`__sinf(x)`
`cosf(x)`	`__cosf(x)`
`tanf(x)`	`__tanf(x)`
`sincosf(x,sptr,cptr)`	`__sincosf(x,sptr,cptr)`
`logf(x)`	`__logf(x)`
`log2f(x)`	`__log2f(x)`
`log10f(x)`	`__log10f(x)`
`expf(x)`	`__expf(x)`
`exp10f(x)`	`__exp10f(x)`
`powf(x,y)`	`__powf(x,y)`

2.3.2. Parallelization of Vector Addition and Multiplication

The two former vector-only operation mentioned in the title of this subsection perform elementwise operations. Assigning a thread to each component contiguously, all threads perform the same simple operation in lockstep, the memory operations are perfectly coalescent, and there is no synchronization needed. Almost no effort is required to divide the workload across the parallel processing units due to the lack of interdependencies. These problems are known as embarrassingly parallel ones [44]. Due to their triviality, there is no scientific research focused on them.

2.3.3. Parallelization of Matrix-Vector Multiplication

The matrix-vector multiplication, also known as General Matrix-Vector Product (GEMV), has conversely received more attention. *Sørensen (2011)* [45] studied the performance of a dense matrix-vector multiplication on a *NVIDIA GeForce 8800 GTX* graphics card. Despite its older architecture and capabilities, the principles behind the optimization of matrix-vector multiplication also work on newer architectures. Different block sizes were tested, and it was concluded that the best strategy for matrices which are the size of the matrices of the linearized state-space model of the wind turbines is to assign one thread per row. This approach removes the need of extra synchronization, while also leveraging the parallel capabilities of the GPU. It is also processed many linearized models of wind turbines in parallel for added efficiency.

Processing many matrix-vector operations in parallel is known as batched operations and *Dong et al. (2016)* [46] studied them. The article is focused on the batched General Matrix-Multiply Product (GEMM) operation, but it dedicates a section to the batched GEMV. The authors assign threads per row, as done in [45], to avoid memory access penalties on global memory.

With larger matrices, this strategy is suboptimal as confirmed by *He et al. (2018)* [47]. They crafted a GEMV kernel that assigned various threads to the same row and then, used warp intrinsics to sum the partial values efficiently. The authors tested the GEMV operation with matrices with a size in the order of 10^8 elements, both with low number of rows and large number of columns, and vice versa. The case study matrices of this study go well beyond the order of magnitude of the matrices of this thesis but, as it was previously shown in [45], it is an indication that different matrix sizes require different multiplication strategies.

Beyond dense matrix-vector multiplication, most of the scientific literature on matrix-vector multiplication is devoted to sparse matrix-vector multiplication. A sparse matrix is a matrix which have so many zeros in it that is computationally advantageous to process it with specialized algorithms [48].

As an example, *Yang et al. (2012)* [49] studied how to improve the multiplication of sparse matrices on the GPU. While dense matrices values are stored contiguously in either row-major or column-major order, sparse matrices are stored differently, as storing all zeros explicitly would be unfeasible due to their generally enormous size. The authors of [49] modified the Compressed Sparse Row (CSR) format, which is one of the most widely used formats for storing sparse

matrices, to perform better on GPUs. They added padding at the end of each compressed row to improve global memory alignment and it showed a 5-30% increase in performance.

Other works such as *Cao et al. (2010)* [50], *Wang et al. (2010)* [51], *Nagasaka et al. (2016)* [52] and *Filippone et al. (2017)* [48] studied the best way to perform this operation on GPUs. All works cite as a major hindrance the scattered memory access that plagues sparse computations, and the difference between these articles lies in the type of matrix processed and their storage format (more on sparse matrix storage formats in Section 2.3.6).

The linear matrices that come from the linearized model of the wind turbine are small, hence the lack of incentive to implement sparsity techniques.

2.3.4. Parallelization of Dense Linear System Solvers

The resolution of linear systems is a problem of utmost importance for the field of engineering. Libraries such as OpenBLAS[5] or Intel MKL[6] include functions that solve linear systems optimized for each CPU architecture. There is a great interest in solving this operation with little computational cost, and hence the interest on its implementation on graphics cards. The NVIDIA corporation already provides the cuBLAS library, which implements many basic mathematic operations, including linear system routines. As noted in Section 6.2, a linear system must be solved sequentially for each timestep when processing the wind farm and, as mentioned in Section 2.1, there is a small overhead at launching CUDA kernels. An efficient kernel design should call cuBLAS from the kernel itself. However, since CUDA 10, the cuBLAS library has removed support for in-kernel cuBLAS function calls [12]. cuBLAS internally uses custom block configurations that might not suit the original calling kernel, and that requires the use of dynamic parallelism. Dynamic parallelism incurs in similar performance penalties, and it should be avoided if possible. Furthermore, cuBLAS is a closed source library, and the compiler might have trouble inlining cuBLAS functions to increase the performance in versions where in-GPU cuBLAS was supported. These issues explain the lack of interest on cuBLAS invocation from CUDA kernels, and hence its eventual removal.

The developers chose to roll their own customized kernels to solve linear systems. These operations are roughly divided into matrix processing, and vector substitution. OpenBLAS, MKL, and cuBLAS offer two paths to solve the system: performing the matrix inverse, and then a matrix vector multiplication or decomposing the matrix with a LU, LDL or QR decomposition [53], and then performing a triangular substitution and/or a matrix vector multiplication.

The bulk of the scientific literature recommends solving linear systems first by decomposing the matrix, and then substituting values back and forth. The reasons to prefer this method against

[5] OpenBLAS. `https://www.openblas.net/`

[6] Intel® Math Kernel Library (Intel® MKL). `https://software.intel.com/en-us/mkl`

precomputing the matrix inverse are a slight precision loss when inverting and the lower time complexity of matrix decomposition [54]. However, that does not mean that using the matrix inverse to solve a linear system cannot be an acceptable approach in some cases [55]. More information on how the matrix inverse is exploited can be found in Section 6.2.

The parallel matrix inversion on graphics cards was studied in works such as *Xuebin et al. (2023)* [56]. The authors tackle the problem of solving many small dense linear systems on the GPU. They design their kernel around the fact that its parallelism stems from the processing of many matrices at once. The threads process at least one row of matrix elements for read and write operations and each block processes many matrices, while the block size is as close to 1024 threads as possible to optimize kernel occupancy. Their algorithm performs up to 80 times better than its CPU counterpart for low size matrices ($n \leq 32$).

Tian et al. (2014) [57] used the Gauss-Jordan algorithm to invert a matrix on graphics cards. This algorithm traverses the rows and uses the values of each row to update the values of the rest of the rows. Looping through the rows is sequential and the algorithm iterates through them sequentially, but also assigns a thread per matrix element to achieve parallelism. The paper does not mention the way each row processing is synchronized. It is assumed that each iteration of the Gauss-Jordan algorithm was performed with a single kernel launched from the CPU, and they were implicitly synchronized by the CUDA API interface, as CUDA by default launches a kernel when all threads of the previous kernel have finished.

Lopez et al. (2018) [58] used a matrix inversion iterative process. This kind of iterative processes are well known and, as an example, the Schulz rule is presented in Equation (13). The goal is to calculate the inverse of A, which has been labeled as x in the expression. Evaluating repeatedly this expression converges to the inverse. This method converges quadratically, although some authors such as [59] have found faster converging expressions. These faster expressions were used by the authors of [58] to develop their kernel. The higher-level iteration expression involves many GEMM operations, and their kernel assigned a thread per element of the output of the matrix multiplication. They achieved a speedup of over 5x against the Gauss-Jordan algorithm.

$$x_{k+1} = x_k(2I - Ax_k) \tag{13}$$

After obtaining the matrix inverse, a matrix-vector product is performed to finally solve the linear system. The ways this operation has been optimized for GPU architectures can be consulted in Section 2.3.3.

The factorization of a matrix using a LU, LDL or QR decomposition has been researched as well. *Baboulin et al. (2017)* [60] studied the use LDL decomposition of a matrix. The LDL decomposition decomposes a symmetrical matrix A into a lower triangular matrix L and a diagonal matrix D as shown in Equation (14). The LU algorithm is like the LDL algorithm, but it is used on non-symmetrical matrices. Its expression is shown in Equation (15), where U is an upper triangular matrix. The authors considered the traditional Bunch-Kaufmann algorithm [61], and the Aasen algorithm [62]. The Aasen algorithm features less communication among the parallel execution units but requires more computationally expensive pivoting. For this reason, this algorithm was rejected by the authors of [60]. The Bunch-Kaufmann algorithm, like the algorithm in the proposal in this thesis, requires numerical pivoting to guarantee the numerical stability of this system. The matrices of the PS processed in this thesis have the same structure,

but different values. Therefore, the matrices were pre-pivoted to avoid the pivoting issue in the proposal of this thesis.

$$A = L \cdot D \cdot L^T \tag{14}$$

$$A = L \cdot U \tag{15}$$

As seen in this section, solving a linear system in CUDA involves a serial operation (iterating over rows), and then applying parallelism over to calculations performed while processing that row. Processing the rows serially is due to the column dependence of the rows. If a row has no zeros in it, it shall modify the rest of the matrix below it. Sparse matrices, which have many zeros, do not suffer this issue. The algorithms to solve linear systems characterized by sparse linear systems can leverage this lack of column dependence and deliver a great speedup. These algorithms will be introduced in the next section.

2.3.5. Parallelization of Sparse Linear System Solvers

Sparse linear systems arise in fields of engineering such as Computational Fluid Dynamics (CFD), material science, bioinformatics, and more importantly, electrical engineering. It was already mentioned in Section 2.1.2 that the equations that govern PS analyses can be arranged in a matrix form. The value of each (i, j) value of the matrix corresponds or is multiplied by the admittance between these two nodes. The admittance between unconnected nodes is zero, and because most nodes are only connected to a few nodes in their vicinity, most values of the matrix will be zero, hence the sparsity of the matrix.

As with dense matrices, NVIDIA has released the cuSPARSE library to process sparse linear systems. NVIDIA has further released the cuSOLVER library, which wraps cuBLAS and cuSPARSE to facilitate the developer the solving of these systems. As with cuBLAS, those libraries are meant to be called by the host code, and kernels where the linear system solver is a just a part of the algorithm should implement their own parallel routines [12].

Hogg et al. (2016) [63] factorizes sparse matrix systems using supernodes. A supernode is a set of contiguous columns with the same sparsity structure below. As an example, Figure 13 shows an example of a supernode. This is important because the LU and LDL factorization updates the lower right matrix by using the leading values below the diagonal. This confined sparsity can be processed by dense linear functions. In Figure 18, the red supernode is processed as a dense matrix below there, as the sparsity pattern is similar for all columns of the supernode. These algorithms work for sparse matrices whose columns can be effectively grouped as a supernode, while giving mediocre results for the rest of matrices. This supernodal approach was used by *Li et al. (2013)* [64] on how to optimize linear solvers on GPU related to PF. They calibrated the kernels to perform the dense calculation as efficiently as possible and achieved a speedup of 0.75x. The largest amount of runtime consists of memory copying with their approach. The structure of the sparse matrices of the PVPP and the wind farm do not leave much parallelism to supernodes, so this approach was discarded.

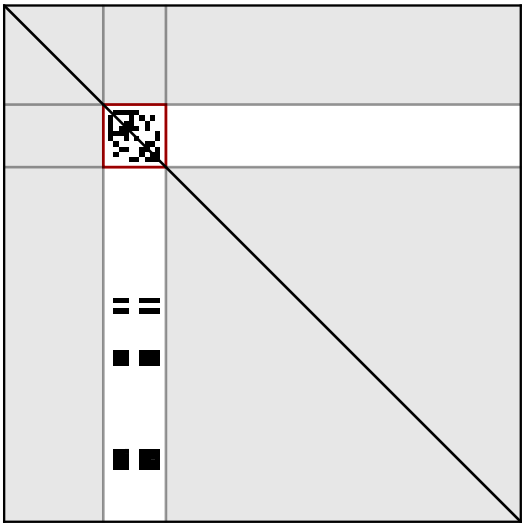


Figure 13. Sparse supernodes.

Lee et al. (2018) [65] used a different approach. They analyzed the matrix for column dependences and organized the columns into levels. The levels are the set of columns which can be processed simultaneously. The authors created three distinct kernels, depending on the level type: cluster, batch, and pipeline. Cluster columns are processed simultaneously, while batch levels consist of two columns that are processed simultaneously. The third level is called pipeline, and it processes one column after another overlapping the final processed values with the first values of the first matrix. To that end, they assign a thread per column and use global memory synchronization to wait until this column can be processed. The proposal for PVPPs simulation follows a strategy like theirs, but it has been implemented in this thesis as chains of columns. Chains of columns is a concept that combines the cluster and pipeline modes of execution into a unified kernel that has the best features of both modes. More information on chains of columns can be found in Chapter 6.

The authors of *Zhou et al. (2017)* [66] avoid all parallelization issues of sparse linear solvers by solving many independent PF problems at once. The matrices associated to the load flow problem share the same structure, and by assigning consecutive threads to the same of columns of different matrices, thread divergence can be avoided. It further parallelizes the matrix processing by means of splitting the columns into dependence levels using a dependency tree with the columns belonging to the same dependence level being processed in parallel. The kernel that processes the matrix that defines the connection of all individual panels in the PVPP in this thesis was processed using a strategy similar to theirs and optimizes column processing to avoid costly synchronization primitives, as described in Section 6.2.

To conclude this section, it should be noted that the previous section addressed the methods to parallelize dense linear systems. Dense linear systems offer two venues to solve them: inversion and matrix decomposition. The matrix inverse of sparse system is usually not sparse. Sparse matrices have usually huge dimensions, and a dense linear matrix simply does not fit in memory and would be computationally expensive. For that reason, matrix inverses are not used for sparse matrices, although they can be used locally, as it was shown with the concept of supernodes. More information on the use of local matrix inverses can be found in Chapter 6.

2.3.6. Sparse Matrix Storage Formats

Before further venturing into the ways PSs are parallelized in CUDA, it is included below a brief list of the ways to store sparse matrices in memory. These storage methods impact the ways matrices can be worked on, and the parallelization opportunities. The main storage formats are listed below:

- **Dictionary of Keys (DOK):** before processing and operating with a sparse matrix, that matrix must be built. These matrices are usually built in software irregularly. As an example, the general matrix that defines the PVPP (see Section 6.3) was built element by element. For each element in the PVPP, it was found out which components they were connected to and filled the rows accordingly. This construction method requires reading and writing matrix values by their location and does not require to iterate through either rows or columns. The structure that is optimized for these use cases is the dictionary of keys (DOK). Dictionaries are implemented internally as binary trees or hashmaps, the former when ordering among the keys is required, the latter when it is not [67]. The DOK storage has been partially used in this thesis while constructing the matrices before the kernel was executed. That implementation is a vector of rows where each row is stored as a dictionary which uses as key the column number. The dictionary is implemented as the `std::map` C++ type, where `std::map` is a generic dictionary implemented as a Red/Black binary tree (see Figure 14), that has many benefits, for example it guarantees fast worst-case access times [67]. This disadvantage of this structure, alongside hashmaps, and all other tree structures, is that they require scattered memory access and their performance in graphics cards is suboptimal.
- **Coordinate Format (COO):** the COO format is the most straightforward way to store a sparse matrix. The matrix is stored as an array of triplets, where each triplet stores the row position, the column position, and the nonzero value of the matrix. The COO representation of the matrix (16) can be found at the linear array (17). The downside of this format is the large size of the array and the slow retrieval of individual values, due to its $O(n)$ search time.

$$M = \begin{bmatrix} 0 & 0 & 0 & 6 \\ 0 & 2 & 0 & 5 \\ 3 & 0 & 1 & 0 \\ 7 & 8 & 9 & 0 \end{bmatrix} \tag{16}$$

$$COO = [0 \quad 3 \quad 6 \quad 1 \quad 1 \quad 2 \quad 1 \quad 3 \quad 5 \quad 2 \quad 0 \quad 3 \quad 2 \quad 2 \quad 1 \quad 3 \quad 0 \quad 7 \quad 3 \quad 1 \quad 8 \quad 3 \quad 2 \quad 9 \quad] \tag{17}$$

- **Compressed Sparse Row (CSR):** CSR is one of the most popular sparse matrix storage formats. The sparse matrix is scanned row-wise and the nonzero values are appended to the array A shown in (18), while the column location of this values is recorded in the array (19) [51]. The row pointer array (20) indicates the beginning of each matrix row. The number of elements per row is worked out by subtracting the value of $row_{pointer}$ between position $n + 1$ and n, where n is the row of interest. This format offers optimal row traversal but difficult column traversal.

$$A = [6 \quad 2 \quad 5 \quad 3 \quad 1 \quad 7 \quad 8 \quad 9] \tag{18}$$

$$column_{index} = [3 \quad 1 \quad 3 \quad 0 \quad 2 \quad 0 \quad 1 \quad 2] \tag{19}$$

$$row_{pointer} = [0 \quad 1 \quad 3 \quad 5 \quad 8] \tag{20}$$

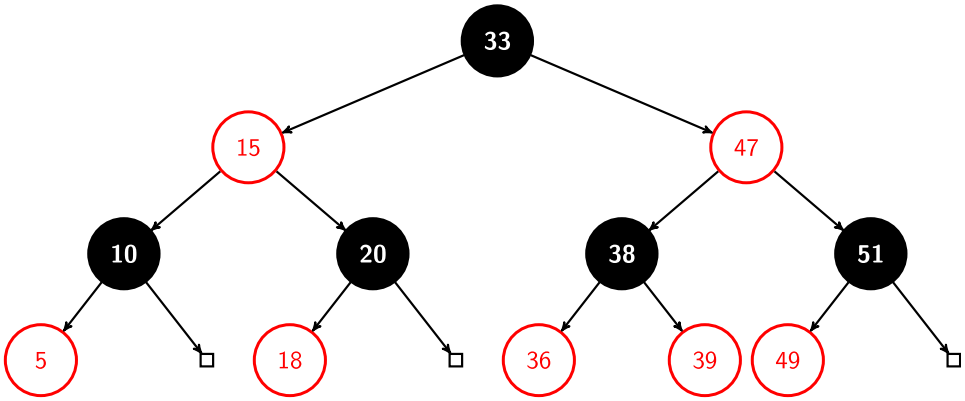


Figure 14. Red/black binary tree.

- **Compressed Sparse Column (CSC):** CSC is like CSR, but columns are traversed. The arrays (21), (22) and (23) show the way matrices are stored in CSC format [51]. Unlike CSR, it can loop through columns efficiently, but lacks performance at traversing rows.

$$A = [3 \quad 7 \quad 2 \quad 8 \quad 1 \quad 9 \quad 6 \quad 5] \tag{21}$$

$$row_{index} = [2 \quad 3 \quad 1 \quad 3 \quad 2 \quad 3 \quad 0 \quad 1] \tag{22}$$

$$column_{pointer} = [0 \quad 2 \quad 4 \quad 6 \quad 8] \tag{23}$$

- **ELLPACK (ELL)**: the ELLPACK format is tailored to sparse matrices with a maximum number of elements per row [51]. In this format, the spare elements are laid out in a matrix of size nxK, where n is the number of rows of the original matrix and K is the maximum number of nonzero elements in the sparse matrix. All nonzero elements are stored sequentially in the matrix A (24), where all rows with nonzero elements are padded with zeros. The column position of the matrix elements is stored in the column order matrix as shown in (25). This format is advantageous when the number of nonzero values is regular across rows. The matrices addressed in this thesis come from electrical systems where many branches converge into one node. The row of that node will have many nonzero values and, therefore, the ELLPACK format is not suited for this application.

$$A = \begin{bmatrix} 6 & 0 & 0 \\ 2 & 5 & 0 \\ 3 & 1 & 0 \\ 7 & 8 & 9 \end{bmatrix} \tag{24}$$

$$column_{index} = \begin{bmatrix} 3 & \cdot & \cdot \\ 1 & 3 & \cdot \\ 0 & 2 & \cdot \\ 0 & 1 & 2 \end{bmatrix} \tag{25}$$

There are more sparse matrix storage formats that are mixtures of the previous basic ones. For example, the Hybrid (HYB) format [68], proposed by NVIDIA, is a combination of ELL and COO. It stores the matrix as ELL matrices, where K is equal to the usual number of zeros. Should

any row have excess nonzero values, they would be stored as an additional COO list. Another variation is the Compressed Sparse Row Improved (CSR-I) format [69], which adds padding to the CSR arrays and the sparse matrix can also be stored using a permutation array [70]. Finally, there are specialized formats for matrices with certain structures, such as the Diagonal (DIA) sparse format, which is optimized for matrices whose values are in the vicinity of the matrix diagonal.

2.4. Parallel Power System Simulation Strategies

There is already research on engineering simulations about renewable energy with graphics cards for non-PS problems where the parallelism of graphics card really shines. For example, *Niu et al. (2011)* [71] studied the flow of wind around a wind turbine, *Robledo et al. (2019)* [72] calculated the best position of household PV panels given the obstacles surrounding the house with raytracing techniques, which are well optimized for GPUs, and *Oberkirsch et al. (2021)* [73] optimized the heliostats of solar tower power plants (not PV). Both works process a massive number of solar-related elements, although they have nothing to do with PS computation. The previous sections briefly mentioned the diverse ways in which the mathematical operations to solve PS problems were parallelized in state-of-the-art GPUs. This section will gather the main global strategies to simulate PS on graphics cards.

2.4.1. Aggregated Models

Aggregating all wind turbines and solar panels into one large device avoids the computation of every individual wind turbine, and therefore reduces the computational cost. It is the main approach to simulating a very large number of wind turbines and PV panels quickly and its main downside is the lack of individual detail of the individual generators [74]. This approach cannot be regarded as parallel, yet it is used to solve the main issue that compels researchers towards developing parallel solutions: computational cost.

Works such as *Kilani et al. (2012)* [75] and *Al-bayati et al. (2016)* [76] explore the use of aggregated models to simulate the global behavior of wind turbines. These aggregated models combine the output of many wind turbines into a single and large wind turbine, and this avoids the extra computational cost and the need for parallelism. Their results approximate the individualized models, but during disturbances there are differences in the global power output due to the aggregation process. The solvers proposed in this thesis calculate the individual state of all wind turbines and solar panels simultaneously and does not use aggregated models.

When using this approach, it is not so important to make use of computational approaches to decrease the running time. For this reason, transport and distribution system operators use simplified models to predict the behavior of the power system, as these deliver the results faster [77]. However, more detailed studies on the behavior of renewable power plants are needed to achieve greater accuracy in reliability testing and the calculation of the power output. To simulate their generated output both accurately and with a computationally affordable cost, it is necessary to use alternative computer architectures that can exploit the intrinsic parallelism of these models, and thus considerably reduce the simulation time. This is the reason this thesis proposes a

simulation with GPUs which does not feature any kind of aggregation. Thus, with such computational power, simulation of renewable power plants with enhanced detail is possible.

2.4.2. Processing of Multiple Power System Configurations

This strategy, also known as batched simulations, entails processing the PS under different conditions or processing many different PSs at once. One of the first works that deal with this topic is *Li et al. (2011)* [78]. This work studied the resolution of the PF problem on NVIDIA GPUs. It implements its solution using the Conjugate Gradient Normal Residual (CGNR) method, which is related to the Conjugate Gradient (CG) family of methods. For illustration purposes, Code 12 details the most basic of these methods, the CG method. The goal is to solve the linear system $Ax = b$ using an initial approximation x_0. The algorithm calculates a series of residuals named r_k, and obtains the next approximation through a series of scalar products and vector additions. The CG algorithm works only for matrices which are symmetric and positive definitive[7] and are numerically unstable. There are variants such as the Biconjugate Gradient (BiCG) or the Biconjugate Gradient Stabilized (BiCGStab) which work for non-symmetric matrices and are numerically stable [53].

```
1  r_0 = b − Ax_0
2  if |r_0| < ε
3      return x_0
4  end if
5  for k = 0 to max_iterations
6      α_k = r_k^T r_k / (p_k^T A p_k)
7      x_{k+1} = x_k + α_k p_k
8      if |r_0| < ε
9          break
10     end if
11     β_k = r_{k+1}^T r_{k+1} / (r_k^T r_k)
12     p_{k+1} = r_{k+1} + β_k p_k
13     k = k + 1
14 end for
15 return x_{k+1}
```

Code 12. Conjugate Gradient (CG) algorithm.

This method approximates the solution of the linear system until the residual vector values go below a selected threshold value. This article implements each suboperation of the CGNR (such as inner products and matrix-vector multiplications) as separate kernels. The kernels are launched consecutively and, therefore they are implicitly synchronized as shown in Sections 2.1.2 and 2.1.7. The authors obtained a large speedup by assigning a thread to each matrix row of the problem. For example, the largest PFs produced a matrix with hundreds of thousands of rows. A parallel matrix-vector computation can assign a thread per matrix row, and no synchronization would be necessary. These kinds of workloads are easily parallelizable. Nevertheless, the authors noticed that the overhead of kernel launching with so many threads should be considered.

[7] Matrix A is positive-definite if, for every nonzero vector z, $z^T Az > 0$.

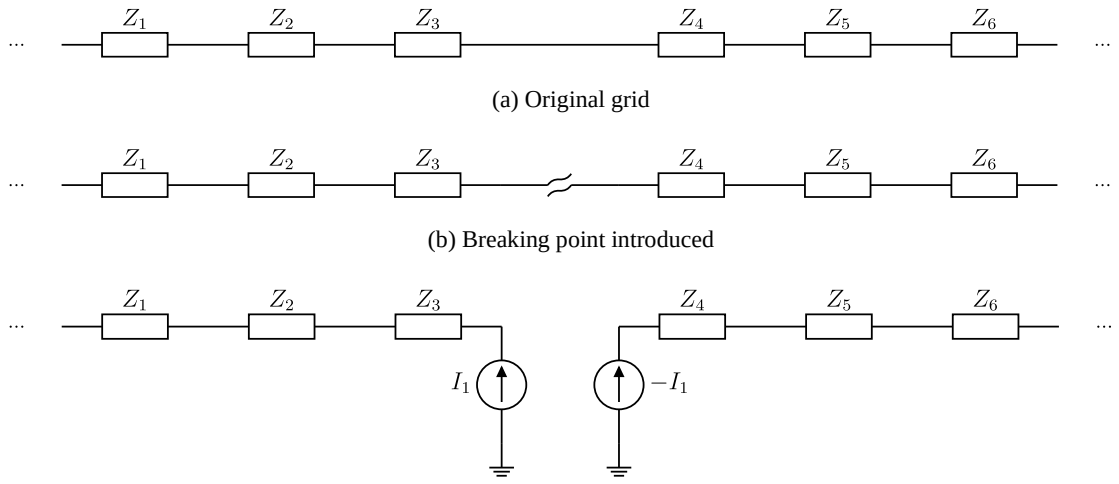
Another early work related to PS simulation is *Papadakis et al. (2011)* [79]. This work simulates the economic dispatch problem of PS using particle swarm optimizations. The goal of this study is to find the most cost-effective operating point of all generation units given the demand. To that end the researchers optimized the global cost function by iteratively improving a solution. There are at first many candidate solutions (or particles), that vary according to the better position that other particles find. This article, like [78], takes advantage of the GPU by updating the position of each particle in parallel. The kernels were also implicitly synchronized.

The kernels proposed in this thesis solve efficiently a large and singular connected power system in parallel. *Song et al. (2020)* [80] parallelized the resolution of many small power systems at once, solving them independently. Another example of this approach was penned by *Wang et al. (2021)* [81], who solved the PF on the GPU by simulating, in parallel, a large number of situations with no relationship between them. After all the simulations are performed, the results are gathered.

2.4.3. Node Tearing

The number of nodes in the grids to be studied, whether in a wind farm or a PVPP, conditions the size of the matrix. The larger the number of nodes, the larger size of the matrix. As it was pointed out in the previous section, simulating many PS at once is a way to achieve parallelism. Node tearing seeks to do so by tearing up the PS. Figure 15 illustrates how the grid is torn apart. The PS is a collection of nodes interconnected by impedances, resistances, or any passive device. It has been decided that the original grid (a) will be partitioned in the node that connects Z_3 and Z_4. Thus, the passive elements that connect to that node are disconnected as shown in (b). Before disconnecting the elements there was current flowing in the line, and a virtual current source must be added (c) for each end of the breaking point. There are only two branches connecting to that node, and hence two current sources must be added. The value of the current source will be a new unknown of the system.

The linear system that defines the problem to solve the network voltages of the PS (see Sections 2.2.5, 2.2.6 and 5.6) can now be represented as the linear system (26), where Y_1, Y_2, … represent the admittance matrices of all n subcircuits. The unknown node voltages of each circuit are the vectors v_1, v_2, … and their injected currents are the vectors i_1, i_2, … respectively. The matrices P_1, P_2, … are the connectivity matrices and are defined as shown in Equation (27). Finally, i_{lnk} represents the linking currents, i.e., the values of the virtual currents added to the circuit as shown in Figure 15. The solution of the linear system in terms of its constituents is found in Equations (28) and (29). First, it is calculated the linking values and once its value is obtained, the node voltages of the subcircuits are calculated. This method is known as node tearing, or Diakoptics [82]. It can extract parallelism by processing in parallel the terms of the summations of the Equations (28) and the $Y_k^{-1} i_k$ terms of Equation (29). For historical reasons, this method was not used and instead sparsity techniques were used [83] (see Section 2.3.5). This method, nevertheless, can deliver parallelization in GPUs if the grids to be studied are especially unfavorable, as it will be shown in Chapter 6.



(a) Original grid

(b) Breaking point introduced

(c) Virtual current sources added

Figure 15. Node tearing.

$$\begin{bmatrix} Y_1 & 0 & \vdots & 0 & P_1 \\ 0 & Y_2 & \vdots & 0 & P_2 \\ \cdots & \cdots & \ddots & 0 & \cdots \\ 0 & 0 & \vdots & Y_n & P_n \\ P_1^T & P_2^T & \vdots & P_n^T & 0 \end{bmatrix} . \begin{bmatrix} v_1 \\ v_2 \\ \cdots \\ v_n \\ i_{lnk} \end{bmatrix} = \begin{bmatrix} i_1 \\ i_2 \\ \cdots \\ i_n \\ 0 \end{bmatrix} \tag{26}$$

$$P_k(i,j) = \begin{cases} -1 & if\ link\ j\ injects\ current\ into\ node\ i \\ 1 & if\ link\ j\ draws\ current\ into\ node\ i \\ 0 & otherwise \end{cases} \tag{27}$$

$$i_{lnk} = \left(\sum_k^n P_k^T Y_k^{-1} P_k \right)^{-1} \left(\sum_k^n P_k^T Y_k^{-1} i_k \right) \tag{28}$$

$$v_k = Y_k^{-1}(i_k - P_k i_{lnk}) = Y_k^{-1} i_k - Y_k^{-1} P_k i_{lnk} \tag{29}$$

The Multi-Area Thévenin Equivalent (MATE) method is an extension of Diakoptics. It reinterprets its equations as Thévenin equivalents. $Y_k^{-1} i_k$ is the Thévenin equivalent source vector of subcircuit k, while $Y_k^{-1} P_k$ is the Thévenin equivalent impedance matrix of subcircuit k. On the other hand, $\sum_k^n P_k^T Y_k^{-1} i_k$ is the equivalent Thévenin equivalent voltage source as seen from the linking current sources, while $\sum_k^n P_k^T Y_k^{-1} P_k$ is its equivalent Thévenin equivalent impedance matrix as seen from the virtual current source [83]. This novel approach opens new optimization venues to reduce the computing cost. For instance, each sub-circuit can be processed with different timesteps depending on the precision required. More on these methods can be found in the next section.

2.4.4. Delayed Line Models and Multi-Rate Methods

Power lines that are hundreds of kilometers long can be modelled using delayed line models. These models assume that the state of an end of the line does not immediately affect the other end of the line. This occurs due to the finite speed of the propagation of electric signals. The articles of *Zhou (2018)* [84] and *Debnath et al. (2021)* [85] introduced a delayed line element to split the power system into individual PSs, so that they can be solved in parallel. Consider the connected

impedances of Figure 15. These impedances represent the lumped elements of the power lines in what is known as the π-section line model (see Section 5.1). The admittance matrix using this model is shown in Figure 16. It can be clearly seen that there is a strong column dependence while processing this matrix (see Section 2.3.5), and that there is little parallelism available to process this matrix. This basic system forms a banded matrix, which is a matrix whose nonzero elements are all near the diagonal. These kind of matrices can be processed in parallel using specialized algorithms [86], but as the matrices that define the interconnected elements of the wind farms and the PVPPs are not banded matrices, these methods cannot be applied[8]. The use of delayed line models, such as the Bergeron line model (see Section 5.2) produces a sparse matrix with the pattern shown in Figure 17. The delayed line models create isolated circuits, which can be regarded as independent, and thus be processed in parallel as if node tearing were applied (see Section 2.4.3). Its only downside is that it has added a new host of nodes, quadrupling the size of the matrix.

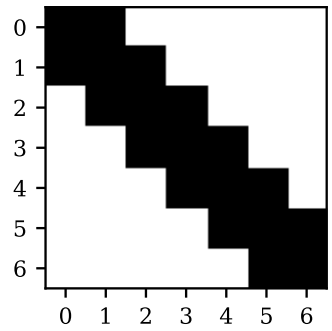


Figure 16. Sparsity pattern of π-section line.

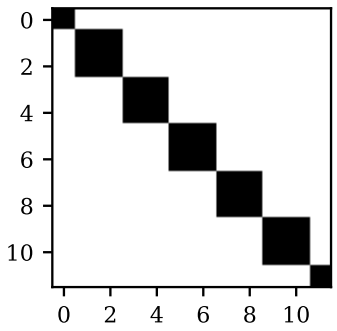


Figure 17. Sparsity pattern of Bergeron line.

This delayed line considers the propagation speed of the electric signals, and it allows the efficient partitioning of the PS, as a state change in one area does not immediately affect all subsystems in the next system. This partitioning works if the PS spans a large area, as is mentioned in the titles of these two articles. An earlier work from the same authors is *Debnath et al. (2012)* [87] where the PS is solved using the nodal analysis method. No further meaningful detail on how the solver is implemented on the GPU is given.

[8] Node ordering allows the nonzero elements to congregate around the diagonal, but as shown in Figure 11, there is a small set of nonzero values far away from the diagonal.

The power plants studied in this thesis connect elements that are close together, and its overhead lines are not long enough to consider delayed line models. It will be nevertheless explored the idea of using these models, as shown in Section 5.2. Delayed line models create isolated circuits which can be processed in parallel. The use of delayed line models is thus recommended for large PS whose power lines span hundreds of kilometers due to the achievable parallelism.

Zhou et al. (2017) [88] uses a multi-level approach. The PS is split with Bergeron power line models and then each individual system is solved by using dense linear algebra. Authors note that although the admittance matrices of the subsystems are sparse, it is feasible to solve them using dense linear algebra techniques.

The PS can also be integrated with multi-rate techniques, where the node voltages are updated at a slower integration time, while the devices connected to it feature detailed models, as shown in *Lin et al. (2022)* [89] and *Cheng et al. (2022)* [90]. The connected elements were finely integrated while the PS used a coarser integration step. The interface between the fine-detailed and coarse detailed models was modeled as a time delayed power line model.

2.4.5. Use of Column-Level Dependency Levels

There are works such as *Song et al. (2014)* [91], which present the implementation of the PS solver. They solve the PS in GPU using dense linear algebra in lieu of sparsity techniques. Use of dense matrices limits the size of the PS, and it is not a problem in this paper because the test case is 9 busbars long. Their solution blocks the computation threads of the GPU with a busy loop until each row of the matrix can be processed. This solution, however, could lead to deadlock if larger matrices are processed. Deadlock occurs when all the threads are waiting on the GPU for other threads to be processed, but these threads cannot be launched because there is no room for them on the GPU. The factorization of the matrix is also performed with column parallelism, as mentioned in Section 2.3.5. The GPU execution model is non-preemptively scheduled, meaning that the developer cannot yield the threads to launch other threads.

Gnanavignesh et al. (2019) [92] used the same approach while *Lee et al. (2018)* [65] solved it with a busy loop until each row of the matrix can be processed. This busy loop prevents each thread from processing a column before it is ready, but suffers from deadlocks, as previously shown. *Feng et al. (2021)* [93] similarly relied on the number of cases to achieve parallelism. The proposed work in the thesis does not require the processing of multiple PS configurations at the same time to extract parallelism. The proposed kernels can simulate a single and unified large wind farm and a PVPP with efficiency.

2.4.6. Use of External Libraries to Solve the Power System

The interconnection part of the power system is a sparse matrix. Many researchers have opted to delegate this computation to specialized libraries. The work of *Song et al. (2018)* [43] can be cited among those. This article focuses on the parallel processing of the controller models and mentions that the network voltage equations can be solved with libraries such as GLU. The GLU library [94] which is tailor-made for circuit simulation, works by using column-level parallelism.

As explained in Section 2.3.5, the performance of libraries that rely on column level parallelism relies on the adequateness of the circuit. For instance, the internal transmission line circuit of the wind farm offers almost no column level parallelism, and its small size negates the benefits of using such a library. It was found that the parallelism achieved in [94] for the PS solver was only up to 50% faster, while the parallelization of the control system of that article is up to 30 and 55 times faster depending on the graphics card used, respectively. However, their code for processing the node voltage equations, which in this thesis has been called the grid solver, is only 35% faster on the GPU. Most of the speedup claimed by the authors lies in the parallelization of the control system, which is more complex.

The wind farm studied in this thesis involves the integration of wind turbine linearized models. These state-space linearized models are simple, and no performance advantage can be found by using an external library when integrating those systems. Performance was enhanced by overlapping the various stages of the power system solver from scratch and without external libraries.

The use of dynamic parallelism suffers similar issues. For instance, in *Lin et al. (2019)* [95] the researchers have avoided the overhead surrounding host-device interactions by using dynamic parallelism. Most calculation of the PS in [95] involve the models of AC to DC inverters, along with its associated modular multilevel converter, transformer, and filter. A main kernel launched sub-kernels to process each submodels of these electronic AC to DC inverters. No sparsity techniques were required to integrate this system and the obtained speed-up was up to 270x. The measured speedup was given for the controller system, which is a fine-grained inverter.

2.4.7. Hybrid CPU-GPU Solvers

Power system parallelization can be difficult. It is therefore sensible to solve this part on the CPU, i.e., split the computation of the power system grid and its connected elements, and solve only the most parallelizable one on the GPU, i.e., the connected elements. For instance, *Araújo et al. (2019)* [96] use a hybrid CPU-GPU approach to power system simulation which solves the system dynamically. Operations that are more suitable to be parallelized are offloaded to the GPU, while the main part of the implementation stays on the CPU. The integration of the connected devices (generator and loads) is performed on the GPU, while the power system itself is executed on the CPU. The solution proposed in *Lin et al. (2021)* [97] suffers from the same issue, although some parts of the code are implemented on both the CPU and GPU and, under different conditions, either the CPU or the GPU is used to increase the total speed-up. This approach either generates many memory transfers between kernels, or needs kernels to wait for the CPU, which wastes resources. The work presented in *Song et al. (2017)* [98] is fully implemented on the GPU, yet the power system solver is implemented as different kernels executed one after another, which is a strategy that adds overhead. The CPU handles the synchronization among stages.

In this thesis the kernel that processes the wind farms are combined into a single kernel that achieves better performance. The PVPP kernels, on the other hand, perform quasi-static simulations (see Section 2.2.6). There are less timesteps to be processed, and the CPU handles the synchronization between each iteration.

2.4.8. Conclusions on Parallel Power System Simulation Techniques

It has been presented many parallelization techniques throughout this chapter. Despite being based on similar mathematical tools, their implementations are vastly different. As a final example of a parallel PS simulation strategy tailor-made to a specific layout, *Marin (2015)* [99] explored the optimization of Radial Power Flow (RPF) using what he called TreeFix operations. The downside of his approach was its limited scope. Radial PSs are currently implemented in distribution systems and due to the arrival of Microgrids [100], it is expected that these radial PSs will become more meshed as time continues. This approach can only be applied to radial grids.

It can be concluded that there is no universal optimization kernel design that can be applied for all PS problems. The kernels developed in this thesis for wind farms and PVPPs will be therefore quite different from each other. Chapter 6 shows the development of vastly different kernels for the simulation of the wind farms and the PVPP.

Chapter 3. Modeling of Wind Turbines

The wind turbines considered in this thesis are equipped with a Doubly-Fed Induction Generator (DFIG). This type of wind turbine has been chosen due to the fact that it is the most representative type of wind turbine, for its market share is over 50% [101]. Section 3.1 gives a brief introduction to the wind turbine models studied and the following sections describe all parts of the model according to the phenomena being studied. Section 3.2 introduces the model of the electrical elements, while Section 3.3 details the equations that govern the mechanical elements of the wind turbine. Section 3.4 briefly explains the control strategy used by the wind turbine. Section 3.5 introduces the per-unit system, while Sections 3.6 and 3.7 are summaries of the DFIG equations. Section 3.8 details how the whole system is linearized and simplified, while Sections 3.9 and 3.10 expound the methods to integrate the linearized models and the way the equilibrium point is obtained. This chapter concludes with Section 3.11, which is a brief reference to the way the equilibrium point is obtained for interconnected wind turbines, and Section 3.12, which is a study on the numerical stability of the 3rd order model when used within an interconnected wind farm.

3.1. Overview of the DFIG

Figure 18 shows a simplified diagram of this type of wind turbine. The wind energy is captured through its blades and transferred to the wind turbine shaft, where it is kept as rotational kinetic energy. The shaft rotates at the same speed of the blades with a large torque. The gearbox converts the rotating speed of the shaft to a speed range more suitable to the generator. The generator is a DFIG as mentioned above, and its stator windings are connected to the external grid, where the electricity generated is fed. On the other hand, its rotor windings are connected to an AC-DC-AC back-to-back converter that regulates the amplitude and phase of the rotor windings to set the DFIG into its optimal setpoint for maximum power generation. For security reasons, the voltages used within the wind turbine are low, such as 690 V [102]. Therefore, a step-up transformer is needed to raise the voltage to the 20 kV range to avoid further electrical losses.

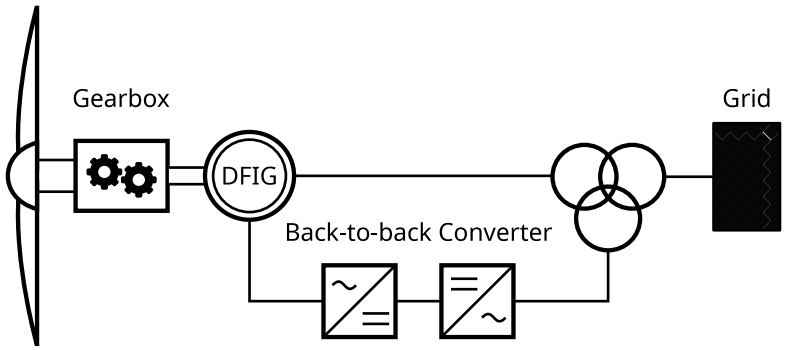


Figure 18. DFIG wind turbine.

To clarify the relationship between all submodels, Figure 19 showcases each subsystem present in the DFIG model, and how its inputs, colored in green, and its outputs, colored in yellow, are interconnected. The blocks that represent physical elements are filled in blue, while

the blocks that represents the controllers are filled in red. The variables shown in the diagram will be explained in the next sections.

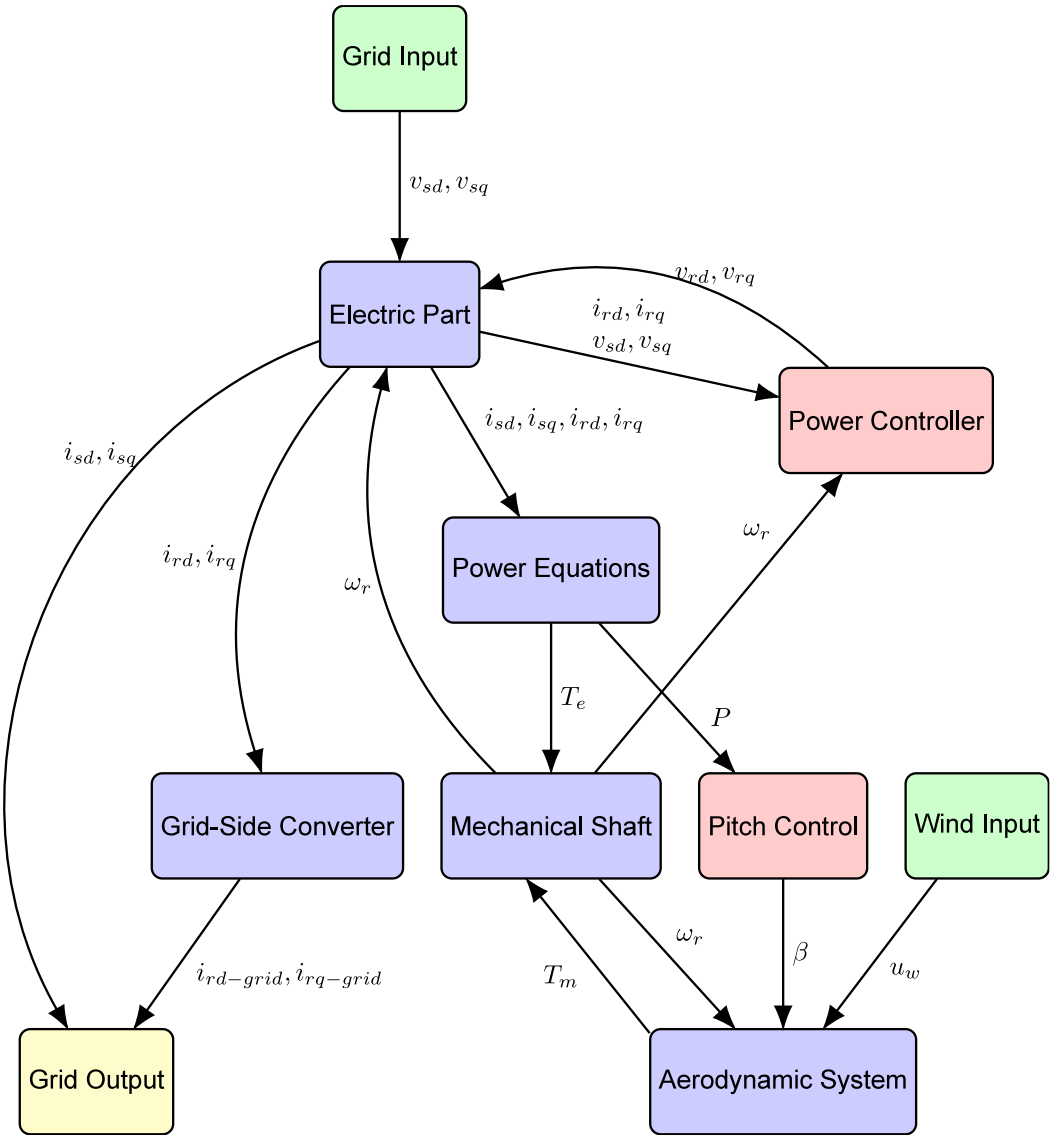


Figure 19. Dynamic model of the DFIG.

3.2. Modelling of Electrical Elements

The DFIG model system can be divided into equations that describe the electrical and mechanical phenomena of the system. The following sections describe the former, while Section 3.3 describes the mechanical elements.

3.2.1. Dq0 Transformation

Dealing with equations that describe a three-phase system is cumbersome. The nature of alternating current entails that for a constant load both voltage and current vary sinusoidally

across time. Furthermore, three values per magnitude must be managed. The most straightforward way to simplify electrical equations is to use electrical phasors, where the amplitude and phase are identified with a complex number. This representation is not adequate for the study of power system transients, not to mention wind turbine transients (see Section 2.2.6).

The *dq0* transformation transforms a three-phase signal with sinusoidal components *abc* into a set of continuous *dq0* signals. This transformation is widely known since calculations in the *dq0* space are simpler, reduce the number of components to be studied, and simplify the design of controllers. The transformation matrix is shown in Equations (30) and (31). The inverse operation $[T_{dq0\rightarrow abc}]$ is carried out with the inverse of the matrix shown in Equation (30). In these equations θ represents the rotational angle of the *dq0* axis. The general expression of the rotational angle is given at Equation (32), which gets simplified to Equation (33) for a constant rotational speed. In this thesis the rotational speed is equal to $2\pi 50 \frac{rad}{s}$ as the global PS the wind farm is connected to is considered to work at 50 Hz. θ_0 is the initial angle of the *dq0* transformation at $t = 0$, and k_{dq0} is a constant whose value is equal to $\frac{2}{3}$.

$$[T_{abc\rightarrow dq0}] = \begin{bmatrix} cos(\theta) & cos\left(\theta - \frac{2\pi}{3}\right) & cos\left(\theta + \frac{2\pi}{3}\right) \\ sen(\theta) & sen\left(\theta - \frac{2\pi}{3}\right) & sen\left(\theta + \frac{2\pi}{3}\right) \\ \frac{1}{2} & \frac{1}{2} & \frac{1}{2} \end{bmatrix} \tag{30}$$

$$\begin{bmatrix} x_d \\ x_q \\ x_0 \end{bmatrix} = k_{dq0} \cdot [T_{abc\rightarrow dq0}] \cdot \begin{bmatrix} x_a \\ x_b \\ x_c \end{bmatrix} \tag{31}$$

$$\theta = \int_0^t \omega(t)dt + \theta_0 \tag{32}$$

$$\theta = \omega t + \theta_0 \tag{33}$$

There are variants of this transformation where the value of k_{dq0} is $\sqrt{\frac{2}{3}}$ and the values at the bottom of the matrix (30) are equal to $\sqrt{\frac{1}{2}}$. With these constants the *dq0* matrix becomes orthonormal, i.e., it verifies the property $A^T = A^{-1}$. Orthonormal matrices keep the scalar product constant, which means that calculating the power in the reference power system *abc* and in the *dq0* space renders the same value.

Let a three-phase balanced system in the *abc* frame of reference shown in Equation (34) be considered. Using $\frac{2}{3}$ for k_{dq0}, $\frac{1}{2}$ in the bottom row, and rotating the *dq0* frame synchronously yields the identity shown in Equation (35). The simplicity of the *dq0* components is clear using this set of constants. All three components have been turned into constant signals. When calculating electric torques and active and reactive power the value must be multiplied by $\frac{3}{2}$. Nevertheless, per-unit values have been used, and thanks to the base values selected, this constant does not appear in the final equations.

Furthermore, when the system is balanced ($x_a + x_b + x_c = 0$), the 0 component is equal to zero, and can be omitted. The zero component will not be added for the rest of equations of this section. During the modelling of the power line, in Chapter 5, the zero component will be omitted as well.

$$\begin{cases} x_a = U_m sen(\omega t) \\ x_b = U_m sen\left(\omega t - \frac{2\pi}{3}\right) \\ x_c = U_m sen\left(\omega t + \frac{2\pi}{3}\right) \end{cases} \tag{34}$$

$$\begin{bmatrix} x_d \\ x_q \\ x_0 \end{bmatrix} = \begin{bmatrix} 0 \\ U_m \\ 0 \end{bmatrix} \tag{35}$$

3.2.2. 5th Order Model of the DFIG

The core of the wind turbine is its DFIG, whose layout is shown in Figure 20. Both the stator and rotor winding are energized, and a current is induced if the shaft is spinning. Both the stator and rotor are composed of three-phase coils (shown both the stator and rotor ones in the star configuration in Figure 20). The shaded parts of the figure correspond to the situation of the coils, existing an air gap between stator and rotor. The inputted mechanical torque is converted to electrical energy as described by the theory of asynchronous generators. The equations link the rotor speed and the voltages of the stator and rotor with the current output of its windings and internal magnetic flux.

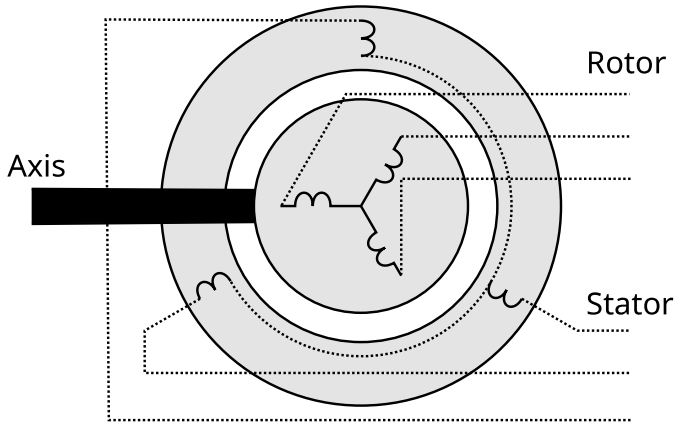


Figure 20. Doubly-Fed induction generator.

The equivalent circuit that represents the DFIG is represented in Figure 21, and the equations that model its behavior are shown in the system of Equations (36), while the relationship between magnetic fluxes and currents is shown in the system of Equations (37). The passive devices represent the losses and inductances of the windings, while the additional voltage sources account for the extra terms that appear after applying the *dq0* transformation of the system (there are similar extra terms when expressing the power line models in the *dq0* frame of reference, as will be shown in Section 5.5). More information on how the equivalent circuit has been obtained can be found in [33]. There are 5 state variables in this model. These variables are the magnetic fluxes

ϕ_{sd}, ϕ_{sq}, ϕ_{rd}, ϕ_{rq} of the stator and rotor, both in the d and q component, and the rotational speed of the rotor ω_r. Hence its name as 5th order model.

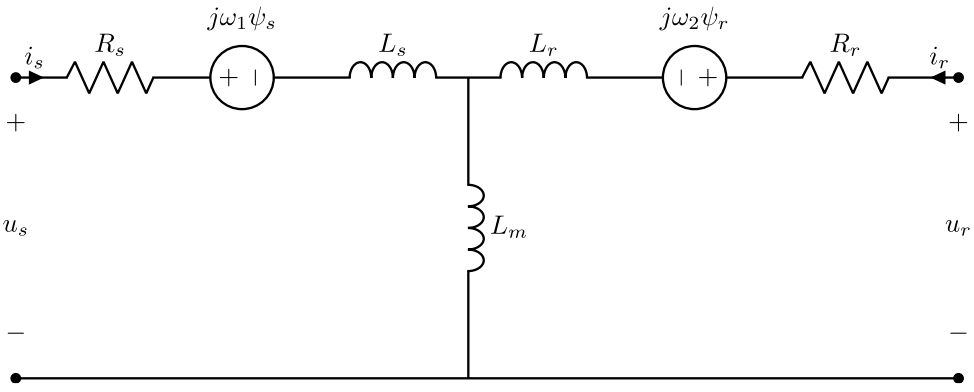


Figure 21. DFIG equivalent circuit.

$$\begin{cases} \frac{1}{\omega_{eb}}\frac{d\phi_{sd}}{dt} = v_{sd} - R_s i_{sd} + \omega_s \phi_{sq} \\ \frac{1}{\omega_{eb}}\frac{d\phi_{sq}}{dt} = v_{sq} - R_s i_{sq} - \omega_s \phi_{sd} \\ \frac{1}{\omega_{eb}}\frac{d\phi_{rd}}{dt} = v_{rd} - R_r i_{rd} + (\omega_s - \omega_r)\phi_{rq} \\ \frac{1}{\omega_{eb}}\frac{d\phi_{rq}}{dt} = v_{rq} - R_r i_{rq} - (\omega_s - \omega_r)\phi_{rd} \end{cases} \quad (36)$$

$$\begin{cases} \phi_{sd} = L_s i_{sd} + L_m i_{rd} \\ \phi_{sq} = L_s i_{sq} + L_m i_{rq} \\ \phi_{rd} = L_m i_{sd} + L_r i_{rd} \\ \phi_{rq} = L_m i_{sq} + L_r i_{rq} \end{cases} \quad (37)$$

The equations can also be expressed using the currents as state variables. Nevertheless, the magnetic flux form is preferred. Stating the states as magnetic fluxes allows to simulate magnetic saturation phenomena. Magnetic saturation occurs when an external current is no longer able to further magnetize a material. The relationship between magnetic flux and current is linear up to a certain limit, where if the electrical current keeps rising, the material does not magnetize further. As shown in the system of Equations (37), no current is affected by a single magnetic flux and, therefore, magnetic fluxes as state variables are needed to simulate magnetic saturation. r_s and r_r are the stator and rotor resistances, respectively. v_{sd}, v_{sq}, v_{rd}, v_{rq} represent the voltages at the terminals of the stator and rotor in the d and q components as well. The voltage of the electric grid will be aligned to the q-axis. Therefore: $v_{sd} = 0.0\ p.u.$ and $v_{sq} = 1.0\ p.u.$

The rotoric voltages are controlled by the electronic converter, and its control strategy will be detailed in Section 3.4. i_{sd}, i_{sq}, i_{rd}, i_{rq} represent the currents with the sign convention shown in Figure 21. ω_s is the synchronous speed of the reference system. This reference speed is equal to $1.0\ p.u.$ as it is considered that network frequency remains constant across simulations. All previous magnitudes were expressed in the per-unit system. The final one, ω_{eb}, is the synchronous speed of the grid. This is the only magnitude whose value is not in the per-unit system. It is equal to $2\pi 50\ \frac{rad}{s}$. In the conversion Equations (37), $L_s = L_{ls} + L_m$ and $L_r = L_{lr} +$

L_m, with L_m being the magnetizing inductance and L_{ls} and L_{lr} the leakage inductances of the stator and rotor. The currents can be expressed in terms of the magnetic fluxes reordering the equations shown in Equation (37). The expressions are presented in the system shown below in Equation (38).

$$\begin{cases} i_{sd} = \dfrac{L_m \phi_{rd}}{L_\sigma} - \dfrac{L_r \phi_{sd}}{L_\sigma} \\ i_{sq} = \dfrac{L_m \phi_{rq}}{L_\sigma} - \dfrac{L_r \phi_{sq}}{L_\sigma} \\ i_{rd} = \dfrac{L_m \phi_{sd}}{L_\sigma} - \dfrac{L_s \phi_{rd}}{L_\sigma} \\ i_{rq} = \dfrac{L_m \phi_{sq}}{L_\sigma} - \dfrac{L_s \phi_{rq}}{L_\sigma} \\ L_\sigma = L_m^2 - L_r L_s \end{cases} \tag{38}$$

3.2.3. 3rd Order Model of the DFIG

The previous sections introduced a precise model of the DFIG. The model abstracts away all internal interactions as a set of inductors and resistances. However, it can be further simplified. The issue is that the integration step required to simulate voltage dips. During those dips, high frequency transients are generated (as it will be shown in Section 7.1) and, therefore, a small integration step is needed to ensure the stability of the simulation.

A 3rd order DFIG wind turbine model is identical to its 5th order counterpart except for the removal of the stator magnetic flux derivative term. This new system cannot simulate accurately the power grid transients. Instead, its output is the average of the oscillations of the magnetic fluxes that would be observed in the 5th order model [103]. When that derivative term is removed, the system of Equations (36) turns into the system of Equations (39). No modifications are applied into the equations that relate magnetic fluxes and currents (37) and (38).

$$\begin{cases} 0 = v_{sd} - R_s i_{sd} + \omega_s \phi_{sq} \\ 0 = v_{sq} - R_s i_{sq} - \omega_s \phi_{sd} \\ \dfrac{1}{\omega_{eb}} \dfrac{d\phi_{rd}}{dt} = v_{rd} - R_r i_{rd} + (\omega_s - \omega_r)\phi_{rq} \\ \dfrac{1}{\omega_{eb}} \dfrac{d\phi_{rq}}{dt} = v_{rq} - R_r i_{rq} - (\omega_s - \omega_r)\phi_{rd} \end{cases} \tag{39}$$

For a 3rd order system, it has been obtained the expressions that calculate the stator magnetic fluxes from the two first Equations of (39). Substituting the Equations (38) into the system of Equations (39) and solving for ϕ_{sd} and ϕ_{sq} yields the expressions that calculate the stator fluxes in the 3rd order DFIG model as shown below in the Equations (40)-(43).

$$\begin{aligned} T_1 = {} & v_{sq}\omega_s L_m^4 - \phi_{rq}\omega_s L_m^3 R_s - 2v_{sq}\omega_s L_m^2 L_r L_s - v_{sd} L_m^2 L_r L_s + \phi_{rq}\omega_s L_m L_r L_s R_s \\ & + \phi_{rd} L_m L_r R_s^2 + v_{sq}\omega_s L_r^2 L_s^2 + v_{sd} L_r^2 L_s R_s \end{aligned} \tag{40}$$

$$\begin{aligned} T_2 = {} & v_{sd}\omega_s L_m^4 - \phi_{rd}\omega_s L_m^3 R_s - 2v_{sd}\omega_s L_m^2 L_r L_s + v_{sq} L_m^2 L_r L_s + \phi_{rd}\omega_s L_m L_r L_s R_s \\ & - \phi_{rq} L_m L_r R_s^2 + v_{sq}\omega_s L_r^2 L_s^2 + v_{sd} L_r^2 L_s R_s \end{aligned} \tag{41}$$

$$\phi_{sd} = \frac{T_1}{L_m^4\omega_s^2 - 2L_m^2 L_r L_s \omega_s^2 + L_r^2 L_s^2 \omega_s^2 + L_r^2 R_s^2} \tag{42}$$

$$\phi_{sq} = -\frac{T_2}{L_m^4\omega_s^2 - 2L_m^2 L_r L_s \omega_s^2 + L_r^2 L_s^2 \omega_s^2 + L_r^2 R_s^2} \tag{43}$$

3.2.4. Power Equations

The following equations calculate auxiliary magnitudes that will be used for both output and control of the wind turbine. One of these magnitudes is the active power, which is the power generated by the wind turbine, and its expression is shown in Equation (44). As noted previously, this expression calculates power using the per-unit system and, according to the sign convention used throughout this thesis, generated power is negative, and consumed power is positive. The expression of active power generated and/or consumed from the rotor is similar as shown in Equation (45). The electronic converter is simulated as a lossless device; therefore, its input and output power are similar. At the point of connection its generated power is equal to Equation (46).

$$P_s = v_{sd} i_{sd} + v_{sq} i_{sq} \tag{44}$$

$$P_r = v_{rd} i_{rd} + v_{rq} i_{rq} \tag{45}$$

$$P_{total} = P_s + P_r = v_{sd} i_{sd} + v_{sq} i_{sq} + v_{rd} i_{rd} + v_{rq} i_{rq} \tag{46}$$

The reactive power, which is the power-like magnitude related to the storage and retrieval of energy during each AC cycle due to an offset in the voltage and current sinewaves. Electronic converters can work with a unitary power factor from the side of the stator, i.e. it does not exchange reactive power [104]. This control strategy has been followed for the simulations performed in this thesis. The expression that shows the whole reactive power generated or absorbed from the grid is therefore equal to the one that calculates the reactive power from the stator side, and its expression is shown in Equation (47).

$$Q_{total} = Q_s = v_{sq} i_{sd} - v_{sd} i_{sq} \tag{47}$$

The final equation shown in this section is the definition of the electrical torque. This work uses the definition from [104], which is shown in Equation (48). A negative electric torque indicates, according to the sign convention used, that the machine is a generator.

$$T_e = L_m\left(i_{rd} i_{sq} - i_{sd} i_{rq}\right) \tag{48}$$

3.2.5. Modelling of the Grid-Side Converter

The back-to-back controller transfers all active power from the rotor side to the grid. This controller has been modelled as if it exchanged no reactive power with the exterior. The expression of the dumped stationary current is shown, for each component, in the Equations (49) and (50). These values calculate the current dumped by the GSC after all transients have occurred. To simulate the dynamics, it is proposed here a low-pass filter, as the transfer function for the GSC proposed in [105] is similar to those of low pass filters. The expressions of the low pass

filters are presented in the Equations (51)-(54), where the parameter α_{GSC} regulates the cutoff frequency of the filter. Finally, it is presented in Equations (55) and (56) the final current that the wind turbine exchanges with the grid.

$$i_{rd-GSC-stationary} = v_{sd}\frac{P_r}{v_{sd}^2 + v_{sq}^2} \tag{49}$$

$$i_{rq-GSC-stationary} = v_{sq}\frac{P_r}{v_{sd}^2 + v_{sq}^2} \tag{50}$$

$$\frac{dGSC_d}{dt} = -\alpha_{GSC} + i_{rd-GSC-stationary} \tag{51}$$

$$i_{rd-GSC} = \alpha_{GSC} GSC_d \tag{52}$$

$$\frac{dGSC_q}{dt} = -\alpha_{GSC} + i_{rq-GSC-stationary} \tag{53}$$

$$i_{rq-GSC} = \alpha_{GSC} GSC_q \tag{54}$$

$$i_{total-d} = i_{sd} + i_{rd-GSC} \tag{55}$$

$$i_{total-q} = i_{sq} + i_{rq-GSC} \tag{56}$$

3.3. Modelling of Mechanical Elements

There are more parts of the wind turbine that shall be modelled which do not involve electrical phenomena, such as the shaft and the wind blades. This section will further explain their modelling.

3.3.1. Mechanical Shaft

The DFIG generates electric power due to the rotation of its shaft coupled to a winding surrounded by a magnetic field. It has been used the two-mass mechanical model, which has been described in many works, such as [106]. The two-mass model conceptualizes the shaft and the gearbox as the system shown in Figure 22. The speed of the blades ω_t and the speed of the DFIG rotor ω_r depend on the electrical torque T_e (see Section 3.2.4), the mechanical torque T_m (see Section 3.3.3), and the angle difference between the two masses θ_{shaft}. The H_r and H_t are the inertia constants of both masses, and K represents the torsion stiffness of the drive train. The differential equations that describe the change of the angular speed of the two masses and the change of the angular difference are shown in the Equations (57)-(59).

$$\frac{d\omega_r}{dt} = \frac{1}{2H_r}\left(T_e - K\theta_{shaft}\right) \tag{57}$$

$$\frac{d\omega_t}{dt} = \frac{1}{2H_t}\left(T_m - K\theta_{shaft}\right) \tag{58}$$

$$\frac{d\theta_{shaft}}{dt} = (\omega_t - \omega_r)\omega_b \tag{59}$$

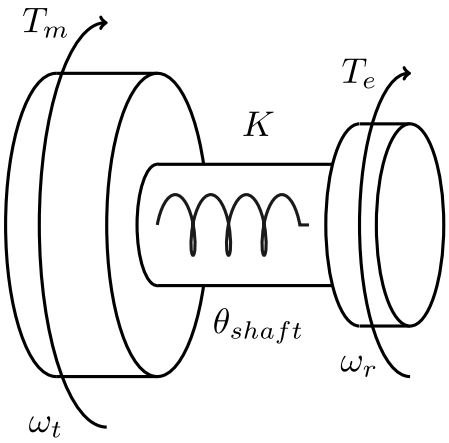


Figure 22. Two-Mass model of the DFIG drive train.

3.3.2. Pitch Control

The pitch control is a device that modifies the attack angle of the blades against the incoming wind to generate as much power as needed. This control is usually activated to reduce the generated power if the wind speed is superior to the nominal power to avoid damages. These simulations always work with values lower or equal to the nominal wind speed. This system has been omitted and its expression is just (60), where β is the attack angle of the blades with respect to the angle of rotation. $\beta = 0$ generates the maximum amount of power.

$$\beta = 0 \tag{60}$$

3.3.3. Aerodynamic System

The simulation of wind alongside wind turbines is a complex topic. Most articles on wind turbine simulation on graphics cards are focused on studying the aerodynamics of the wind turbine. These results belong to the field of CFD and are beyond the scope of the model used in this thesis, as this model is centered around its electrical output. It has been sought a simplified wind turbine where the only input value is a single wind speed value.

The per-unit values have been used to model the previous sets of equations. The equations featured in this section are expressed using real values. If the per-unit values are used, a subindex will be added. It is shown in Equations (61) and (62) the relationship between the real and the per-unit values. ω_{blades} is the angular speed of the wind turbine, while ω_{tpu} is the per-unit mechanical speed of the rotor. In previous sections, it was presented as ω_t. The pu suffix was added to highlight that is a per-unit magnitude. $r_{multiplier}$ is the multiplier ratio of the gearbox and T_{mpu} is the mechanical per-unit applied over the electric generator in the per-unit system. The base units used in these expressions are T_b, which is the base torque; and ω_{mb}; which is the base mechanical torque. Finally, P_{wind} is the wind speed generated by the wind in the per-unit system. According to the sign convention, generated power is positive.

$$\omega_{blades} = \frac{\omega_{tpu}\omega_{mb}}{r_{multiplier}} \tag{61}$$

$$T_{mpu} = -\frac{P_{wind}}{\omega_t \omega_{mb} T_b} \tag{62}$$

The torque calculated thanks to Equation (62) is the applied torque which is then applied on the generator. The gearbox has been modelled as a lossless element. Since the rotational speed and mechanical torque are expressed in the per-unit system, its numerical values are similar across both sides of the gearbox. The generated power according to the wind is shown in Equation (63) where c_p is the power coefficient. This coefficient cannot be superior to Betz's limit $\left(\frac{16}{27}\right)$. r_{blades} is the length of the blades in meters, and u_w is the wind speed in m/s.

$$P_{wind} = \frac{1}{2} c_p \pi r_{blades}^2 u_w^3 \tag{63}$$

The power coefficient depends mostly on the angle of attack of the blades β, and on the coefficient λ, which is the relationship between the linear velocity of the tip of the blades and the wind speed, known as the Tip Speed Ratio (TSR), as shown in Equation (64). It has been proposed different models that represent this behavior, and in this thesis it has been used the equations and coefficients presented in [107]. First, an intermediate value called δ is assigned, whose expression is shown in Equation (65) which, in turn, is part of the final equation that models c_p, shown in Equation (66).

$$\lambda = \frac{\omega_{blades} r_{blades}}{u_w} \tag{64}$$

$$\delta = \left(\frac{1}{\lambda + 0.08\beta} - \frac{0.03}{1 + \beta^3}\right)^{-1} \tag{65}$$

$$c_p(\lambda, \beta) = 0{,}5\left(\frac{116}{\delta} - 0{,}4\delta\beta - 5\right) e^{\frac{-21}{\delta}} \tag{66}$$

3.4. Modelling of the Control System

The electronic converter that is part of the DFIG wind turbine is mounted in a back-to-back configuration. The converter sits between two AC systems with different amplitude and phase to transfer electric power between them. The converter is divided in 3 parts:

- **GSC (Grid-Side Converter):** it is a three-phase inverter that transfers energy to the DC link and vice-versa. The GSC has been modelled in a such a way that it does not exchange reactive power with the external grid. The GSC is regarded as ideal.
- **DC link:** it is the link that transmits DC-power to the GSC. It has a capacitor that stabilizes its voltage, eliminates high-frequency harmonics, and removes sudden variations of current. The rotor-side converter, which will be presented below, injects, or absorbs the energy stored in it to set a certain amplitude and phase in the rotors' terminals. The GSC job is to keep the voltage of the DC-link as constant as possible [104]. It has been assumed that the DC-link bus voltage is constant and ideal.

- **RSC (Rotor-Side Converter**): it is a three-phase inverter like the GSC that adjusts the voltage in the rotor. This rotor has been modelled as a voltage source connected to the rotor.

As previously shown, the variables on which the converter can act to modify the behavior of the wind turbine are the rotor voltages. Wind turbine control must take care of them. The controller used in this thesis is a Proportional Integral (PI) one. Its behavior is schematized in Figure 23. As presented in this figure, rotor currents are compared against a reference value, where the PI takes its difference as an input, outputting the rotor voltages that the RSC shall set. The generic plant block abstracts and simplifies the global behavior of the wind turbine. Its response are the actual rotor currents, which will be again compared with the reference values in a loop.

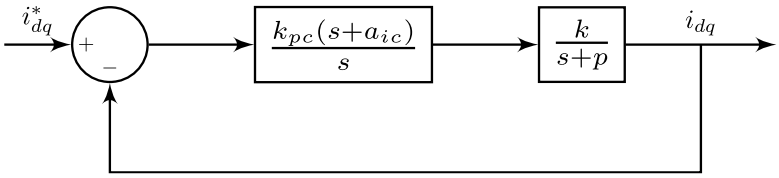


Figure 23. Simplified model of the DFIG with controller.

The PI controller has two parameters: K_p and K_i. Furthermore, there will be two PI controllers, one for each component of the rotor current. The K_p and K_i constants have been assumed to be identical for each PI controller.

3.4.1. Active Power Control

Active power is related with the electric torque and, therefore, plays a key role in its control expression. The expression shown in Equation (67) that calculates the reference electric torque is presented below [104].

$$T_{ref} = -K_{opt}\omega_r^2 \tag{67}$$

Where K_{opt} is a constant that depends on the wind turbine. On the other hand, Equation (68) corresponds to the relationship between the optimal i_{rq} and the reference torque [104].

$$i_{rq-optimal} = -\frac{L_s}{L_m}\frac{T_{ref}}{v_{sq}} \tag{68}$$

3.4.2. Reactive Power Control

The reactive power is linked to the *d* component according to the alignment used in this thesis. The control equation of the reactive power [104] is equal to the expression shown in Equation (69), where Q_{ref} is the reference reactive power in the per-unit system. This reactive power depends on the wind farm requirements and the commands issued by the grid operators, such as

the REE (Spanish Electric Grid, in Spanish "Red Eléctrica de España"), which has the monopoly of electrical energy transport in Spain.

$$i_{rd-optimal} = -\frac{v_{sq}}{\omega_s L_m} - \frac{L_s}{L_m v_{sq}} Q_{ref} \tag{69}$$

3.4.3. Control Constants

The expressions that will be presented in this section regard the voltage controller of each component as independent, while the wind power plant is regarded as a transfer function with a single and stable pole. The controller will be designed using a simplified zero-positioning technique. It can be assumed that a simplified version of the transfer function of a DFIG wind turbine has only a single dominant pole and that this controller will add a zero to offset this pole. The simplified expressions of the rotor voltages [104] are shown in Equations (70)-(72).

$$u_{rd} = R_r i_{rd} + \frac{1}{\omega_b}\sigma_L \frac{d}{dt} i_{rd} - (\omega_s - \omega_r)\sigma_L i_{rq} + \frac{1}{\omega_{eb}} \frac{L_m}{L_{ss}} \frac{d\frac{|u_s|}{\omega_s}}{dt} \tag{70}$$

$$u_{rq} = R_r i_{rq} + \frac{1}{\omega_b}\sigma_L \frac{d}{dt} i_{rq} + (\omega_s - \omega_r)\sigma_L i_{rd} + (\omega_s - \omega_r)\frac{L_m}{L_{ss}} \frac{|u_s|}{\omega_s} \tag{71}$$

$$\sigma_L = L_{rr} - \frac{L_m^2}{L_{ss}} \tag{72}$$

In order to calculate the transfer function it will be used a simplified one that omits certain terms [104]. This simplified version is found in Equation (73).

$$u_{dq} = R_r i_{dq} + \frac{1}{\omega_b}\sigma_L \frac{d}{dt} i_{dq} = i_{dq}\left(R_r + \frac{1}{\omega_b}\sigma_L s\right) \tag{73}$$

The transfer function in open loop of the layout shown in Figure 23 is Equation (74). Cancelling a pole assigning the zero so that $a_{ic} = p$ leads to Equation (75). The closed loop transfer function of Equation (75) is shown in Equation (76). Finally, the value of the k_{pc} controller constant can be calculated using the expression shown in Equation (77) [104], where t_r is the rise time of the controller. The designer picks this value to fit the requirements. On the other hand, the simplified transfer function that can be inferred from Equations (70) and (71) is presented in Equations (78).

$$G(s) = k\frac{k_{pc}(s + a_{ic})}{s(s + p)} \tag{74}$$

$$G(s) = k\frac{k_{pc}}{s} \tag{75}$$

$$C(s) = \frac{G(s)}{1 + G(s)} = \frac{k \cdot k_{pc}}{s + k \cdot k_{pc}} \tag{76}$$

$$k_{pc} = \frac{ln(9)}{k t_r}(1 + m) \tag{77}$$

$$P(s) = \frac{i_{dq}}{u_{dq}} = \frac{1}{R_r + \frac{1}{\omega_b}\sigma_L s} = \frac{\frac{\omega_b}{\sigma_L}}{R_r \frac{\omega_b}{\sigma_L} + s} \tag{78}$$

Comparing the expression of the transfer function of the wind turbine with the expressions of the controller constants, it can be concluded that the control constants K_p and K_i are given by Equations (79) and (80).

$$K_p = \frac{\ln(9)}{\frac{\omega_b}{\sigma_L} t_r}(1+m) \tag{79}$$

$$K_i = R_r \frac{\omega_b \ln(9)}{\sigma_L t_r}(1+m) \tag{80}$$

3.5. Per-Unit System

This thesis uses the per-unit system to describe the models of the wind turbine. Table IV shows the formulae to calculate the per-unit values used in this work. It has been chosen to work with these magnitudes to simplify the mathematical expressions of the model. The per-unit values were obtained by dividing them with their base counterparts. Thanks to this choice, per-unit values are close to 1.0, which aids mathematical convergence [104].

Table IV. Base units.

Parameter	**Symbol**	**Formula**
Base Voltage	v_b	Initial value, nominal wind turbine voltage
Base Power	P_b	Initial value, nominal wind turbine power
Base Electrical Angular Speed	ω_b	$2\pi 50$
Base Current	I_b	$\frac{2}{3}\frac{P_b}{v_b}$
Base Mechanical Angular Speed	ω_{mb}	$\frac{\omega_b}{pair\ of\ poles}$
Base Resistance	R_b	$\frac{v_b}{I_b}$
Base Admittance	Y_b	$\frac{I_b}{v_b}$
Base Inductance	L_b	$\frac{v_b}{I_b \omega_b}$
Base Capacitance	C_b	$\frac{I_b}{v_b \omega_b}$
Base Magnetic Flux	ϕ_b	$\frac{v_b}{\omega_b}$
Base Torque	T_b	$\frac{P_b}{\omega_{mb}}$

3.6. Input and Output Variables of the Wind Turbine Model

This section summarizes what are the state variables of the whole wind turbine model, its inputs, and its outputs. The 5th order DFIG wind turbine model features 11 states, while the 3rd order model has 9 states. They are:

- $\phi_{sd}, \phi_{sq}, \phi_{rd}, \phi_{rq}$: magnetic fluxes in the d and q directions for both the stator and rotor (p.u.). In the 3rd order model, the magnetic fluxes of the stator are not states.
- ω_r, ω_t: rotor speed of the rotor and the blades (p.u.).
- θ_{shaft}: angular difference of the two masses in the mechanical shaft (rad).
- $PIState_{Active}$: this state is associated with the integral part of the PI controller shown in Section 3.4.1. The output of this PI controller regulates the voltage v_{rq}.
- $PIState_{Reactive}$: this state is associated with the integral part of the PI controller shown in Section 3.4.2. The output of this PI controller regulates the voltage v_{rd}.
- GSC_d: this state is associated with the d-component of the GSC controller.
- GSC_q: this state is associated with the q-component of the GSC controller.

Likewise, the inputs of the wind turbine system are:

- v_{sd}, v_{sq}: d and q components of the voltage on the wind turbine terminals (p.u.).
- Q_{ref}: reactive power of reference. This value sets the desired reactive power output of the wind turbine as presented in Section 3.4.2.
- u_w: perpendicular wind speed (m/s).

The outputs of this system are the total currents of the wind turbine ($i_{total-d}$ and $i_{total-q}$). These outputs will be fed to the global PS solver to update the v_{sd} and v_{sq}values, respectively. The rotor speed ω_r, the electrical torque T_e, and the generated power P_e were also included as output variables, raising the total number of output variables to 5. The 3rd order DFIG wind turbine model features 9 states, which are those of the 5th order model except the stator magnetic fluxes ϕ_{sd} and ϕ_{sq}.

3.7. Summary of Wind Turbine Nonlinear Equations

As a summary of all previous subsections, it is included below in the full set of equations (81)-(111) of the DFIG wind turbine 5th model.

$$\frac{1}{\omega_{eb}}\frac{d\phi_{sd}}{dt} = v_{sd} - R_s i_{sd} + \omega_s \phi_{sq} \tag{81}$$

$$\frac{1}{\omega_{eb}}\frac{d\phi_{sq}}{dt} = v_{sq} - R_s i_{sq} - \omega_s \phi_{sd} \tag{82}$$

$$\frac{1}{\omega_{eb}}\frac{d\phi_{rd}}{dt} = v_{rd} - R_r i_{rd} + (\omega_s - \omega_r)\phi_{rq} \tag{83}$$

$$\frac{1}{\omega_{eb}}\frac{d\phi_{rq}}{dt} = v_{rq} - R_r i_{rq} - (\omega_s - \omega_r)\phi_{rd} \tag{84}$$

$$\phi_{sd} = L_s i_{sd} + L_m i_{rd} \tag{85}$$

$$\phi_{sq} = L_s i_{sq} + L_m i_{rq} \tag{86}$$

$$\phi_{rd} = L_m i_{sd} + L_r i_{rd} \tag{87}$$

$$\phi_{rq} = L_m i_{sq} + L_r i_{rq} \tag{88}$$

$$P_e = v_{sd} i_{sd} + v_{sq} i_{sq} + v_{rd} i_{rd} + v_{rd} i_{rq} \tag{89}$$

$$Q_e = v_{sq} i_{sd} - v_{sd} i_{sq} \tag{90}$$

$$T_e = L_m\left(i_{rd} i_{sq} - i_{sd} i_{rq}\right) \tag{91}$$

$$i_{rd-GSC-stationary} = v_{sd} \frac{P_r}{v_{sd}^2 + v_{sq}^2} \tag{92}$$

$$i_{rq-GSC-stationary} = v_{sq} \frac{P_r}{v_{sd}^2 + v_{sq}^2} \tag{93}$$

$$\frac{dGSC_d}{dt} = -\alpha_{GSC} + i_{rd-GSC-stationary} \tag{94}$$

$$i_{rd-GSC} = \alpha_{GSC} GSC_d \tag{95}$$

$$\frac{dGSC_q}{dt} = -\alpha_{GSC} + i_{rq-GSC-stationary} \tag{96}$$

$$i_{rq-GSC} = \alpha_{GSC} GSC_q \tag{97}$$

$$i_{total-d} = i_{sd} + i_{rd-GSC} \tag{98}$$

$$i_{total-q} = i_{sq} + i_{rq-GSC} \tag{99}$$

$$\frac{d\omega_r}{dt} = \frac{1}{2H_r}\left(T_e - K\theta_{shaft}\right) \tag{100}$$

$$\frac{d\omega_t}{dt} = \frac{1}{2H_t}\left(T_m - \mathrm{K}\theta_{\mathrm{shaft}}\right) \tag{101}$$

$$\frac{d\theta_{shaft}}{dt} = (\omega_t - \omega_r)\omega_b \tag{102}$$

$$i_{rq-optimal} = \frac{-L_s}{L_m} \frac{-K_{opt}\omega_r^2}{v_{sq}} \tag{103}$$

$$i_{rd-optimal} = \frac{v_{sq}}{\omega_s L_m} - \frac{L_s}{L_m v_{sq}} Q_{ref} \tag{104}$$

$$\beta = 0 \tag{105}$$

$$P_{wind} = \frac{1}{2} c_p \pi r_{blades}^2 u_w^3 \tag{106}$$

$$\omega_{blades} = \frac{\omega_t \cdot \omega_{\mathrm{base}}}{r_{multiplier}} \tag{107}$$

$$T_{m\_pu} = \frac{-P_{wind}}{(\omega_r \cdot \omega_{\mathrm{mb}}) T_b} \tag{108}$$

$$\lambda = \frac{\omega_{blades} r_{blades}}{u_w} \tag{109}$$

$$\frac{1}{\delta} = \frac{1}{\lambda + 0{,}08\beta} - \frac{0{,}035}{1 + \beta^3} \tag{110}$$

$$c_p(\lambda,\beta) = 0{,}5\left(\frac{116}{\delta} - 0{,}4\delta\beta - 5\right)e^{\frac{-21}{\delta}} \tag{111}$$

In the 3rd order model of the DFIG wind turbine model, the statoric magnetic fluxes are not independent variables. Its expressions are shown in Equations (114) and (115). To facilitates the reading of these long expressions, it has been defined two auxiliary values in Equations (112) and (113).

$$\begin{aligned} T_1 = {} & v_{sq}\omega_s L_m^4 - \phi_{rq}\omega_s L_m^3 R_s - 2v_{sq}\omega_s L_m^2 L_r L_s - v_{sd} L_m^2 L_r L_s + \phi_{rq}\omega_s L_m L_r L_s R_s \\ & + \phi_{rd} L_m L_r R_s^2 + v_{sq}\omega_s L_r^2 L_s^2 + v_{sd} L_r^2 L_s R_s \end{aligned} \tag{112}$$

$$\begin{aligned} T_2 = {} & v_{sd}\omega_s L_m^4 - \phi_{rd}\omega_s L_m^3 R_s - 2v_{sd}\omega_s L_m^2 L_r L_s + v_{sq} L_m^2 L_r L_s + \phi_{rd}\omega_s L_m L_r L_s R_s \\ & - \phi_{rq} L_m L_r R_s^2 + v_{sq}\omega_s L_r^2 L_s^2 + v_{sd} L_r^2 L_s R_s \end{aligned} \tag{113}$$

$$\phi_{sd} = \frac{T_1}{L_m^4\omega_s^2 - 2L_m^2 L_r L_s \omega_s^2 + L_r^2 L_s^2 \omega_s^2 + L_r^2 R_s^2} \tag{114}$$

$$\phi_{sq} = -\frac{T_2}{L_m^4\omega_s^2 - 2L_m^2 L_r L_s \omega_s^2 + L_r^2 L_s^2 \omega_s^2 + L_r^2 R_s^2} \tag{115}$$

3.8. Linearization of System

The previous section showed the equations of the 5th and 3rd order equations of the DFIG wind turbine. These equations involve variable whose value changes according to mathematical expressions that feature derivatives. These expressions form what is known as an Ordinary Differential Equation (ODE) system, and Equation (116) shows the general expression of a n-order implicit ODE, where F and G are arbitrary functions of the time t, the state vector x, and its derivative $\dot{x}$; and y is the output of the ODE. The equations that form the wind turbine system have at most first order derivatives and their states can be explicitly stated. For that reason, the ODE expression can be expressed as first order explicit system (117). In this expression, the time dependence has been explicitly rewritten as an input vector $u(t,x)$.

$$\begin{cases} F(t,x,\dot{x}) = 0 \\ G(t,x,\dot{x}) = y \end{cases} \tag{116}$$

$$\begin{cases} \dot{x} = F\big(x,u(t)\big) \\ y = G\big(x,u(t)\big) \end{cases} \tag{117}$$

Out of all types of ODEs, Linear Time Invariant (LTI) ones are a subset of ODEs that are more convenient to study. Among the main properties of the LTI systems are their linearity and their time invariance. A system is linear when the linearity condition (see Equation (118)) is fulfilled, where u_1 and u_2 are input vectors and α and β are arbitrary constants. The time invariance condition is fulfilled if, after offsetting the input of the system by a certain t_0, the output of the system is also shifted. LTI systems have been widely studied [108], and many mathematical properties such as the stability of the system can be obtained easily. LTI systems can be expressed in the state-space matrix form as shown in Equation (119), where A, B, C and D are time-

invariant matrices. The matrix dimensions of a system with n states, p inputs and q outputs are (n,n), (n,p), (q,n) and (q,p) respectively.

$$F(x,\alpha u_1+\beta u_2)=\alpha F(x,u_1)+\beta F(x,u_2) \tag{118}$$

$$\begin{cases}\dot{x}(t)=Ax(t)+Bu(t)\\ y(t)=Cx(t)+Du(t)\end{cases} \tag{119}$$

As inferred from Equation (119), all state and input variables must be multiplied by a constant, yet the wind turbine model has many nonlinearities. As an example, in one of the equations that models the electrical phenomena (83), both the rotor speed ω_r and the magnetic flux ϕ_{rq} are multiplied among themselves; in the expressions that yield the optimal rotor currents shown in Equations (103) and (104), the input voltage is in the denominator; and the equation that calculates the power coefficient shown in Equation (111) features an exponential whose power is an intermediary value that, ultimately, depends on ω_t. Working with a nonlinear system poses several changes, chief among them is their irregularity and difficulty to be parallelized (more information on Chapter 6). However, nonlinear systems can be approximated thanks to the linearization process. This section will briefly introduce it and expound its mathematical background. Most of the linearization techniques can be consulted in [109].

All analytical functions can, by definition, be represented locally as a sum of powers. A well-known power series is the Taylor series, whose formula is shown in Equation (120), where f is an analytical function and x_0 is the point at which derivatives are considered. If the expression (120) is applied to the nonlinear ODE system, and if the high order derivatives are ignored, we get Equation (121)[9]. The expression has been linearized around the equilibrium point. The equilibrium point (122) x_0 is the state where for a certain input u_0, the derivatives of the system are equal to zero, i.e., the system no longer evolves.

$$f(x)=\sum_{n=0}^{\infty}\frac{f^{(n)}(x_0)}{n!}(x-x_0)^n \tag{120}$$

$$f\big(x(t),u(t)\big)\approx f(x_0,u_0)+\left.\frac{\partial f}{\partial x}\right|_{x_0,u_0}(x-x_0)+\left.\frac{\partial f}{\partial u}\right|_{x_0,u_0}(u-u_0) \tag{121}$$

$$f(x_0,u_0)=0 \tag{122}$$

The nonlinear function is multi-input and multi-output. The counterpart of the derivative for these functions is the Jacobian Matrix, whose elements are the partial first order derivatives of each combination of input and output of the original function, and they correspond to the A and B matrices as portrayed in Equations (123) and (124). Reapplying the same process for the G function presented in Equation (117) produces the C and D matrices (125) and (126). For convenience, the Δx, Δu, and Δy values have also been defined in Equations (127)-(129). It is

[9] It is first assumed in this thesis that the Taylor series of the nonlinear wind turbine system of equations converges. There are smooth and differentiable functions whose Taylor series does not converge, such as $\ln(1+x)$, which does not converge for $x>1$. It will be shown in Section 7.1 that the linearized model is, indeed, a good approximation.

straightforward that the derivative of Δx and x is equal as x_0 is constant. Finally, the linearized system can be written as a state-space system (130) if the expressions (123)-(129) are applied.

The expressions of these matrices are quite long, and they can be consulted in the Annex A, where Section A.4 shows the 3rd order model linearization matrices and Section A.5 shows the 5th order model matrices.

$$A = \frac{\partial F}{\partial x}\Big|_{x_0,u_0} = \begin{bmatrix} \frac{\partial F_1}{\partial x_1}\Big|_{x_0,u_0} & \cdots & \frac{\partial F_1}{\partial x_n}\Big|_{x_0,u_0} \\ \vdots & \ddots & \vdots \\ \frac{\partial F_m}{\partial x_1}\Big|_{x_0,u_0} & \cdots & \frac{\partial F_m}{\partial x_n}\Big|_{x_0,u_0} \end{bmatrix} \tag{123}$$

$$B = \frac{\partial F}{\partial u}\Big|_{x_0,u_0} = \begin{bmatrix} \frac{\partial F_1}{\partial u_1}\Big|_{x_0,u_0} & \cdots & \frac{\partial F_1}{\partial u_n}\Big|_{x_0,u_0} \\ \vdots & \ddots & \vdots \\ \frac{\partial F_m}{\partial u_1}\Big|_{x_0,u_0} & \cdots & \frac{\partial F_m}{\partial u_n}\Big|_{x_0,u_0} \end{bmatrix} \tag{124}$$

$$C = \frac{\partial G}{\partial x}\Big|_{x_0,u_0} = \begin{bmatrix} \frac{\partial G_1}{\partial x_1}\Big|_{x_0,u_0} & \cdots & \frac{\partial G_1}{\partial x_n}\Big|_{x_0,u_0} \\ \vdots & \ddots & \vdots \\ \frac{\partial G_m}{\partial x_1}\Big|_{x_0,u_0} & \cdots & \frac{\partial G_m}{\partial x_n}\Big|_{x_0,u_0} \end{bmatrix} \tag{125}$$

$$D = \frac{\partial G}{\partial u}\Big|_{x_0,u_0} = \begin{bmatrix} \frac{\partial G_1}{\partial u_1}\Big|_{x_0,u_0} & \cdots & \frac{\partial G_1}{\partial u_n}\Big|_{x_0,u_0} \\ \vdots & \ddots & \vdots \\ \frac{\partial G_m}{\partial u_1}\Big|_{x_0,u_0} & \cdots & \frac{\partial G_m}{\partial u_n}\Big|_{x_0,u_0} \end{bmatrix} \tag{126}$$

$$\Delta x = x - x_0 \tag{127}$$

$$\Delta u = u - u_0 \tag{128}$$

$$\Delta y = y - y_0 \tag{129}$$

$$\begin{cases} \dot{\Delta x}(t) = A\Delta x(t) + B\Delta u(t) \\ \Delta y(t) = C\Delta x(t) + D\Delta u(t) \end{cases} \tag{130}$$

3.9. Numerical Methods of Integration

The previous section presented a method to linearize the wind turbine nonlinear system. The differential system of Equations (130) has an analytic solution provided the mathematical expression of $u(t)$ is known. However, this does not usually occur as the input data of a wind turbine is empirical and highly irregular. Furthermore, its analytical solution (131) relies on matrix exponentials, which can introduce numerical instability [110]. This expression also features the approximate solution if the input is considered as constant. These are the reasons why numerical integration has been chosen to integrate the state-space systems.

$$\Delta x(t) = e^{At}\left(\int_0^t e^{-Az}B\Delta u(z)\,dz + \Delta x(0)\right) \approx A^{-1}(e^{At} - I)Bu(t) + e^{At}\Delta x(0) \tag{131}$$

There are many numerical integration methods. All methods originate from the definition of derivative which, for a sufficiently small timestep h verifies the following expression (132). This expression can also be used to calculate the next step, which is known as Euler's method, and it is the simplest numerical method of integration [36]. The lower the timestep is, the lower the error of the results becomes, and a sufficiently large timestep can make a stable system unstable. As an example, the first order linear ODE shown in Equation (133) is stable. Given an initial state, its value will eventually approach zero with no oscillations. However, the discretization using Euler's method shows widely different results depending on the integration step as it can be seen in Figure 24. The analytic solution approaches zero as expected, but the solutions of larger timesteps oscillate, overshoot, and at worst do not converge. Larger timesteps show that the solution diverges. Tweaking Euler's method can mitigate this issue, as more complex methods are more stable, and their values are closer to the analytic solution. The alternatives range from multiderivative methods like the Parker-Sochacki method [111]; which not only uses the function to be integrated, but its derivatives as well; to other alternative methods such as the exponential integrator method [112], which regards the input variable as constant between timesteps and calculates the next timestep using the (131) expression. But due to their simplicity, the Runge-Kutta family of methods [36] are the most widely used. The methods of the Runge-Kutta family calculate the next step of a generic $\frac{dx}{dt} = f(t,x)$ using the expressions (134) and (135), where the number order of the method is s, and the constants a_{ij}, b_i and c_i depend on the Runge-Kutta method used.

$$x(t+h, u+h) \approx x(t,u) + h\frac{dx(t,u)}{dt} \tag{132}$$

$$\frac{dx}{dt} = -10x \tag{133}$$

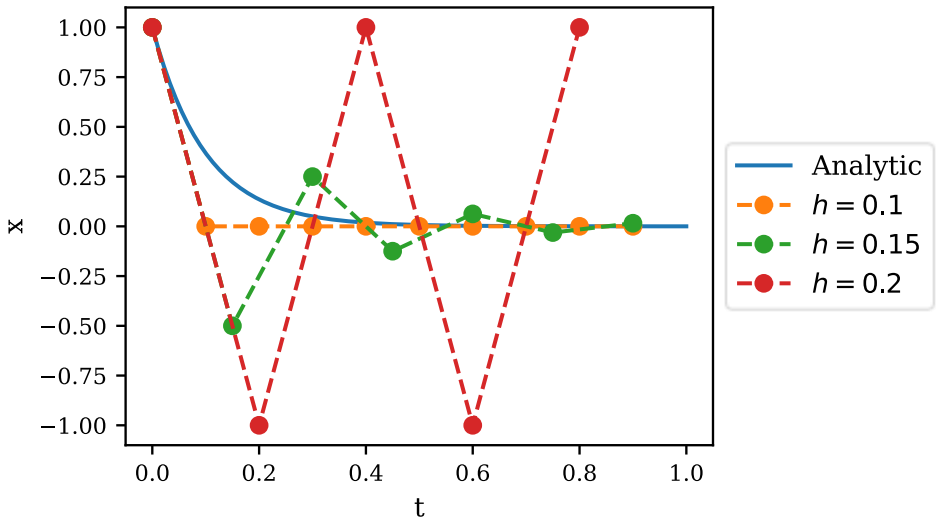


Figure 24. Numerical stability of Euler's method.

$$x(t+h) = x(t) + h\sum_{i=1}^{s} b_i k_i \tag{134}$$

$$k_i = f(t_n + c_i h, x_n + h\sum_{i=1}^{s} a_{ij} k_j) \tag{135}$$

The stability of each integration method can be studied by considering a generic first order ODE as shown in Equation (136), where A is a constant. Control theory states that this continuous system will be stable if its pole, which is A, is negative. A discrete system, such as the ones that result from applying the numeric integration methods with a constant timestep, are stable if only if the condition (137) is fulfilled. Using this relationship, the stability of Euler's method can be calculated. For example, applying Equation (137) to Euler's method when numerically integrating the simplified ODE system yields Equation (138). The term $1 + hA$ is known as the stability condition, and it is key to compare different Runge-Kutta integration methods. Table V shows various integration methods, and the minimum timestep depending on the value of A. Most of these methods are also known by a RKx name, where x is the order of the method. As it is shown, more powerful integration methods allow larger timesteps without losing stability. Their downside is their extra computing cost. Out of all the Runge-Kutta methods presented, the method of Heun, also known as the modified Euler's method, or the trapezoid rule, strikes a balance between accuracy and computing efficiency. This method is used in industry standard software related to PS simulation such as PSS®E [113]. Its expression is presented in Equations (139) and (140). The new value is calculated by averaging the function at the t and at the $t + h$ instant, had the function been advanced using Euler's method. The equations to calculate the new states for a State-Space LTI system are obtained when applying Equations (139) and (140) to Equation (119), and they are shown in Equation (141).

$$\frac{dx(t)}{dt} = Ax(t) \tag{136}$$

$$\left|\frac{x(t+h)}{x(t)}\right| < 1 \tag{137}$$

$$\left|\frac{x(t) + hAx(t)}{x(t)}\right| = |1 + hA| < 1 \tag{138}$$

$$x(t+h) = x(t) + h\frac{f\big(t+h, x_{temp}\big) + f\big(t, x(t)\big)}{2} \tag{139}$$

$$x_{temp} = x(t) + hf\big(t, x(t)\big) \tag{140}$$

$$\begin{cases} \Delta x(t+h) = \Delta x(t) + h\dfrac{A\big(\Delta x(t) + x_{temp}\big) + B\big(\Delta u(t) + \Delta u(t+h)\big)}{2} \\ \mathrm{x_{temp}} = \Delta x(t) + h\big(A\Delta x(t) + B\Delta u(t)\big) \\ \Delta y(t) = C\Delta x(t) + D\Delta u(t) \end{cases} \tag{141}$$

3.10. Equilibrium Point

The previous section addressed the linearization process, which converted the nonlinear DFIG model that was presented throughout the whole Chapter 3 into an LTI system in state-space form.

Linearization requires an equilibrium point to linearize around and this section will explain the steps to obtain it.

Table V. Stability of different Runge-Kutta methods.

Method	Stability Condition	Stable Integration Step
(Explicit) Euler (RK1) [10]	$Ah + 1$	$h < \frac{-2}{A}$
Implicit Euler, or Backward Euler	$\frac{1}{1 - Ah}$	$h > 0$
Heun, or Trapezoid (RK2) [53]	$1 + Ah + \frac{(Ah)^2}{2}$	$h < \frac{-2}{A}$
4th order Runge Kutta (RK4) [53]	$1 + Ah + \frac{(Ah)^2}{2} + \frac{(Ah)^3}{6} + \frac{(Ah)^4}{24}$	$h < \frac{-2.7853}{A}$
Bogacki-Shampine (RK3) [114]	$1 + Ah + \frac{(Ah)^2}{2} + \frac{3(Ah)^3}{16} + \frac{(Ah)^4}{48}$	$h < \frac{-3.1523}{A}$
Dormand-Prince (RK5) [114]	$1 + Ah + \frac{(Ah)^2}{2} + \frac{(Ah)^3}{6} + \frac{(Ah)^4}{24} + \frac{(Ah)^5}{120} + \frac{(Ah)^6}{600}$	$h < \frac{-3.3066}{A}$
Dormand-Prince (RK8) [114]	$1 + Ah + 0.5(Ah)^2 + 0.1667(Ah)^3 + 0.04167(Ah)^4 + 0.008333(Ah)^5 + 0.001389(Ah)^6 + 0.0001984(Ah)^7 + 2.48 \cdot 10^{-5}(Ah)^8 + 2.76 \cdot 10^{-6}(Ah)^9 + 2.42 \cdot 10^{-7}(Ah)^{10} + 2.44 \cdot 10^{-8}(Ah)^{11} - 2{,}04 \cdot 10^{-10}(Ah)^{12}$	$h < \frac{-5.1666}{A}$

The previous section already defined the equilibrium point as the state where the system no longer evolves, i.e., where all its derivatives are equal to zero. Getting the equilibrium point thus entails solving an algebraic system where the derivatives are zeroed for a given input variable, which has been named as u_0 in the previous section. All simulations of the wind turbine elements start in the equilibrium point and, therefore, the initial input will be u_0, where $u_0 = u(0)$. There is sadly no numerical solution of the DFIG equilibrium point. The main reason is the exponential term of the aerodynamic system shown in Equation (66), which indirectly contains the rotor speed variable. A naive approach to obtain the equilibrium point is to simulate the wind farm model with a constant input until it converges. This approach is suboptimal, and it is not guaranteed to converge, as the initial values of that previous simulation must still be picked.

Therefore, numerical methods must be used that find the root of the equilibrium point. The most common method to solve this problem is Newton's method [53]. Equation (142) shows its multivariate form, where the k^{th} approximation of the root vector x of a function f is approximated using the inverse of the Jacobian Matrix (see Section 3.8). The success of this method depends on the election of the initial value x^0. As an example, the function $f(x) = sgn(x)\sqrt{|x|}$ is continuous for all $\mathbb{R}$ and clearly has a root at $x = 0$. Applying Newton's method choosing as the first approximation $x^0 = 1$ does not converge to the solution but oscillates between 1 and -1. There are alternative methods such as the secant's method, Broyden's method

[10] Euler's method is also known as the Explicit Euler method to distinguish it from the implicit method.

and Steffensen's method [53], but these methods also rely on successive refinements using the slope of the function. These methods are locally convergent, but not globally.

$$x^{k+1} = x^k - J_f^{-1}(x^k) f(x^k) \tag{142}$$

To tackle the issue of choosing the initial guess, the nonlinear system of equations has been reduced to an equivalent system with a single unknown that has an obvious initial guess. The system of equations presented in Section 3.7 was analytically solved without the equation that has the differential term of the rotor speed, effectively removing the rotor speed as an independent variable. Then, the system was solved numerically. The mathematical expression of this analytical solution is exceedingly long, and it has been moved to the Annex A, where it can be consulted. The expressions shown there return the initial value of all the states provided the rotor speed is known. Then, the roots of the Equation (58), which has the mechanical torque in the expression, which in turn contains all the problematic terms, are found using the single-variable version of Equation (142), shown in Equation (143). The terms of the Equation (58) which depend on the rest of states are substituted using the expressions found in Section A.3.

$$x^{k+1} = x^k - \frac{f(x^k)}{f'(x^k)} \tag{143}$$

The value 1.0 was used as a starting guess. This value corresponds to the base rotor speed in the per-unit system. The wind turbine is estimated to work with a rotational speed of 1.0 on normal operation, hence the validity of this initial guess. The method proposed in this thesis to find the equilibrium point combines both analytical and iterative approaches and requires fewer starting values, thus avoiding convergence issues.

3.11. Initialization of Voltage Input

This subsection clarifies the way voltage inputs of the wind turbines were initialized. The wind turbines are connected through transmission lines as it will be described in Chapter 5. These transmission lines have losses and, although the voltage at the Point of Common Coupling (PCC) is known, the voltage at the wind turbine terminals will be a bit different. To that end, it was first assumed that the voltage at the wind turbines terminal was the voltage at the PCC. The equilibrium point was calculated and so were their new currents. The wind farm grid made of transmission lines (see Chapter 5) was solved, and new voltages for the wind turbines were obtained. Due to the low impedance of the power lines these voltages are remarkably like the previous ones. The cycle is repeated until convergence is achieved. Figure 25 summarizes this interaction.

The wind farm model will be also modelled in this way. The integration of the wind farm models, and the grid dynamics occur at the same time. Figure 25 shows them being calculated at separate times. The input of each part of the simulation is thus delayed by one timestep, which can be represented as a unit delay. These unit delays cause instabilities in the 3rd order model as noticed in Section 3.12.

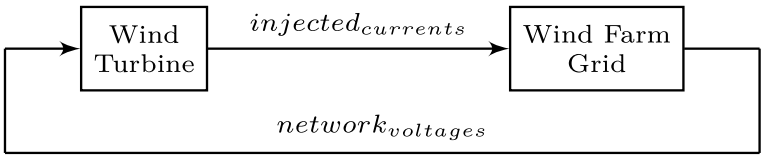


Figure 25. Interaction between wind turbine and wind farm grid.

3.12. 3rd Order Model Stability

The final subsection of the chapter of this thesis that deals with the wind turbines models based on DFIGs concludes with a brief study on the stability of the 3rd order model applied to the simulation in PS. As previously presented, the output of the 3rd order DFIG model shows less oscillations. However, it was found that this model was incompatible with the grid models used in this thesis. The 3rd order model presented instabilities due to the implicit unit step delay introduced during the simulation.

As it will be shown in Section 6.1.1, the wind turbines are integrated and, with the obtained voltages, the voltage on each bus of the grid is obtained. These phenomena nevertheless occur at the same time, and the input voltage required by the wind turbine models are the voltage values at the previous timestep and are therefore required to be solved at the same time with a nonlinear solver that is computationally expensive. To solve this issue, a unit step is introduced that allows one part of the simulator individually, breaking algebraic loops, and allowing faster simulations. Figure 26 shows the interaction with the added unit step.

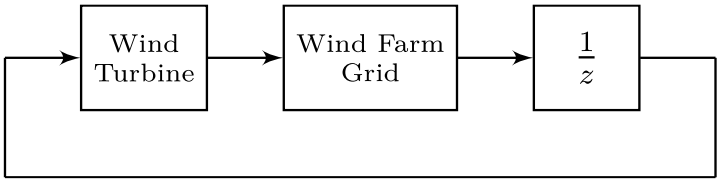


Figure 26. Updated interaction between wind turbine and wind farm grid with unit step delay.

Studying the simulation stability of this global system requires to discretize all its components. The general expression of discrete state-space systems is shown in the pair of Equations (144), where $x[n]$ and $x[n+1]$ represent the vector of states at timesteps n and $n+1$, $u[n]$ is the input vector, $y[n]$ is the output vector, and A_d, B_d, C_d, and D_d are matrices. For performing this study, it has been considered a simplified 3rd DFIG system whose rotor speed remains constant, and their rotor voltages are zero. The equations of this system have been presented in Sections 3.2.2 and 3.2.3.

$$\begin{cases} x[n+1] = A_d x[n] + B_d u[n] \\ y[n] = C_d x[n] + D_d u[n] \end{cases} \tag{144}$$

Applying Heun's method to continuous state-space systems (141) yields the pair of Equations (145). It has been considered that $\Delta u(t+h) \approx \Delta u(t)$ and that it has been discretized with a timestep h.

$$\begin{cases} \Delta x[n+1] = (I + hA + \frac{h^2A^2}{2})\Delta x[n] + (hB + \frac{h^2AB}{2})\Delta u[n] \\ \Delta y[n] = C\Delta x[n] + D\Delta u[n] \end{cases} \quad (145)$$

After that, the grid was discretized. The power line considered for this study is a reduced version of the π-section line (see Section 5.1). Its equivalent circuit is shown in Figure 27. The power line has been reduced to a simple lumped inductance. Many software packages, such as MATLAB-Simulink[11], cannot connect a current source in series to an inductance because then it could not formulate the state-space representation of the grid. It has been added in parallel a large resistance that affects little to the simulation. This way, it is possible to represent the grid equations in the state space formulation. The matrix Equations (146)-(149) show the continuous power system representation of the power line shown in Figure 27. The states of the system are the d and q currents of the coil and the inputs are the current injected by the DFIG and the voltage of the system at the V_{cc} node, both in the *dq0* reference frame. Its outputs are the voltage at the DFIG's terminals.

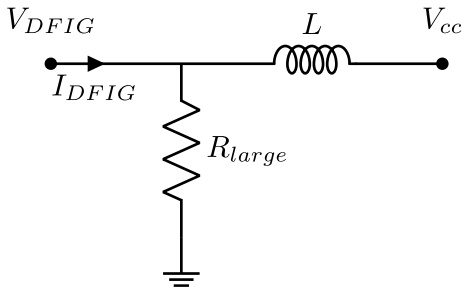


Figure 27. Simplified power line to study voltage instability.

$$A_{line} = \begin{bmatrix} -R_{large} & L \\ -L & -R \end{bmatrix} \cdot \frac{\omega}{L} \quad (146)$$

$$B_{line} = \begin{bmatrix} R_{large} & 0 & -1 & 0 \\ 0 & R_{large} & 0 & -1 \end{bmatrix} \cdot \frac{\omega}{L} \quad (147)$$

$$C_{line} = \begin{bmatrix} -R_{large} & 0 \\ 0 & -R_{large} \end{bmatrix} \quad (148)$$

$$D_{line} = \begin{bmatrix} R_{large} & 0 & 0 & 0 \\ 0 & R_{large} & 0 & 0 \end{bmatrix} \quad (149)$$

Finally, the discrete unit delay state-space representation is given in the Equations (150)-(153). The inputs are, in this case, the voltage outputs of the previous state-space system, and its output corresponds to the state at the previous timestep.

$$A_{delay} = \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix} \quad (150)$$

$$B_{delay} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \quad (151)$$

[11] MATLAB version: 9.13.0 (R2023a). `https://www.mathworks.com/`

$$C_{delay} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \tag{152}$$

$$D_{delay} = \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix} \tag{153}$$

For reference, the state-space that results from the concatenation of two state-space systems with states x_1 and x_2 is given below, in the Equations (154) and (155). These operations can be obtained trivially by operating algebraically the generic state-space expressions. Applying this operation twice, a final unified system is obtained.

$$\begin{bmatrix} x_1[n+1] \\ x_2[n+1] \end{bmatrix} = \begin{bmatrix} A_1 & 0 \\ B_2C_1 & A_2 \end{bmatrix} \begin{bmatrix} x_1[n] \\ x_2[n] \end{bmatrix} + \begin{bmatrix} B_1 \\ B_2D_1 \end{bmatrix} u[n] \tag{154}$$

$$y[n] = \begin{bmatrix} D_2C_1 \\ C_2 \end{bmatrix} \begin{bmatrix} x_1[n] \\ x_2[n] \end{bmatrix} + [D_2D_1]u[n] \tag{155}$$

Finally, this system has feedback to itself, and its output and its input are equal ($y[n] = u[n]$). Expressing the input in terms of the output expression and reorganizing yields the final expression of the discrete system, as shown in Equation (156), which has no external inputs. Control theory states that a discrete system will be stable if and only if, the absolute value of all eigenvalues of the A matrix is lower than 1.0 [53]. The discrete system shown in Equation (156) corresponds to a LTI discrete system whose A matrix corresponds to the $(A + B(I - D)^{-1}C)$ term. It was found that there were eigenvalues whose absolute value was larger than one, thus making the system unstable. If it is studied the system without the unit step, it becomes stable. The 5th order DFIG model does not show this issue.

$$x[n+1] = (A + B(I - D)^{-1}C)x[n] \tag{156}$$

Chapter 4. Modeling of Photovoltaic Power Plants

Photovoltaic (PV) power plants have taken the world of energy generation by storm. Small, flat and with no mechanical parts, PV panels can be installed in many different configurations, taking advantage of the free space in top of buildings, roofs, and even in large Photovoltaic Power Plants (PVPPs). These large PVPPs span dozens of square kilometers due to the PV panels' low power density. As mentioned in previous sections, one of the drawbacks of wind turbines is the need to install many individual generation units, causing a larger simulation cost. The nominal power of wind turbines is in the order of the megawatts, while PV panels' nominal power is in the order of the hundreds of watts. This makes parallelization of PVPPs essential to tackle, as new PVPPs are being installed with progressively larger sizes due to the lower installation costs of PV panels [115]. Knowing the state of the PVPP is key to model the behavior of the PS by the grid operators to ensure its viability and safety.

The following subsections will describe all systems related to PVPPs, starting in Section 4.1 with the smallest and most basic unit, which is the PV cell. Section 4.2 describes how the PV cells are mounted into PV panels, which are then laid out into PV arrays as presented in Section 4.3. The PV panels work with DC power, which must be converted to AC, and therefore inverters are needed. This chapter concludes with the description of those inverters in Section 4.4 and a summary of all equations introduced throughout this chapter in Section 4.5.

4.1. PV Cell

The most basic unit of a PV panel is the PV cell. The PV cell is composed of two layers of semiconductor material where the photoelectric effect takes place. The inner workings of the PV cell are represented in Figure 28. This figure shows the internal structure of the PV cell, which is a p-n junction of semiconductors, usually silicon. Small impurities can be added to a slab of pure semiconductor material, also known as intrinsic semiconductor, and named in Figure 28 as i-type semiconductor, to form both p-type and n-type semiconductors. This process, known as doping, changes the way the material conducts electricity. P-Type doping adds traces of trivalent elements, such as boron or aluminum, to the silicon matrix, while n-type doping adds pentavalent elements like phosphorus, arsenic, antimony, and bismuth. Silicon has four valence electrons, and the impurities in the silicon crystal donate a free electron per atom, for the case of n-type impurities, while there is one hole per atom in the case of p-type impurities. Holes are, in the context of semiconductors, an abstraction of the lack of one electron where it should be one. As atoms get their holes filled by nearby electrons from other atoms, these holes travel to nearby atoms, and they can be considered as particles with positive charge.

The doped regions previously mentioned are electrically neutral if they are isolated. Joining the p and n-type regions diffuses nearby electrons of the n-type region into the holes of the p-type region. This small layer is the depletion layer, and it is the phenomenon behind semiconductor technologies. This depletion layer is not electrical neutral. It is electronegative in the n-type part and positive in the p-type part. It prevents current from flowing through the semiconductor. External voltages affect the thickness of the depletion layer, shrinking or enlarging it, which is the

underlying principle behind diodes. As shown in the figure, a PV cell is like a diode, and indeed its mathematical model resembles the one from a diode. Applying a positive voltage from the n-type to the p-type enlarges the depletion layer, and no current flow should be possible. An energetic photon can take electrons away from the semiconductor and allow for additional current flow. Figure 28 shows the real flow of current, where the current flows from negative to positive.

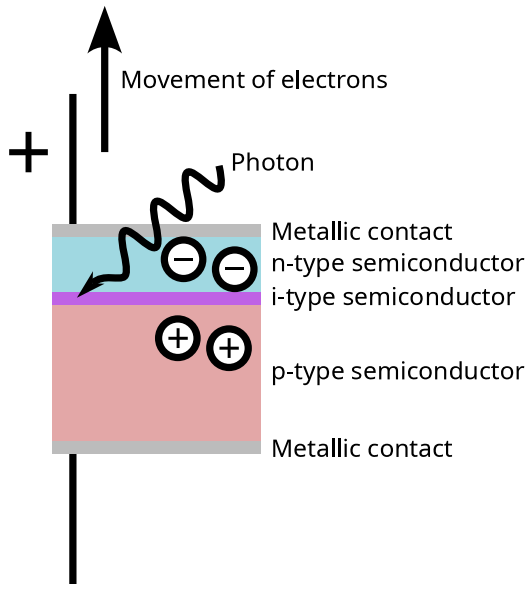


Figure 28. Photovoltaic cell and p-n junction (layers not to scale).

As presented above, there are two behaviors that shall be considered: the p-n junction diode behavior and the current flow caused by photons impacting on the PV cell. The Equations (157) and (158) describe the electrical characteristics of the PV cell, and Figure 29 represents these equations as an electrical circuit. In the PV cell model r_s and r_{sh} are the series and shunt parasitic resistors, while D_1 and D_2 are elements in place to mimic the second and third terms of Equation (157) respectively. They model the junction recombination phenomenon. The recombination of the electron-hole pairs can either happen in the neutral region of the cell, or in the depletion layer. The former exhibits a behavior with an ideality factor of $n = 1$, the latter of $n = 2$ [116]. n_a is related to the behavior of the PV panel at high voltages. The constants q and k correspond to the electrical charge of the electron and the Boltzmann constant, while I_{01} and I_{O2} are constants that depend on the PV cell type. The current source I'_{SC} represents the short circuit current of the PV cell without considering the parasitic resistors. Its output depends on both the current irradiance G, and the PV cell temperature T. I_{SC} is equal to the short circuit current of the PV cell with irradiance G_0 and temperature T_0. Finally, K_0 quantifies the effect of the temperature on the generated current by the PV panel. If the test is conducted under normalized conditions, G is equal to $1000\frac{W}{m^2}$ and T is 25 ºC.

$$I = I'_{SC} - I_{O1}\left(e^{\frac{q(V+I\cdot r_s)}{kTn_a}} - 1\right) - I_{O2}\left(e^{\frac{q(V+I\cdot r_s)}{2kTn_a}} - 1\right) - \frac{V + I\cdot r_s}{r_{sh}} \tag{157}$$

$$I'_{SC} = I_{SC}\frac{G}{G_0}\cdot\left(1 - \mathrm{K_0}(\mathrm{T} - \mathrm{T_0})\right) \tag{158}$$

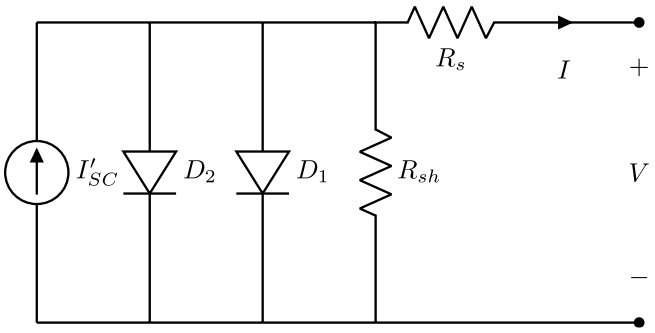


Figure 29. PV cell electric model.

The PV cell model presented has two diodes. However, a cursory review of the literature regarding PV cell and panel simulation shows that most proposals disregard the diode term with ideality factor $n = 2$ due to the small contribution to the PV cell current. This small contribution is smaller the larger the applied voltage, and Figure 30 represents this behavior. It has been represented the current that flows through the diode when a certain voltage is applied. The current axis uses a logarithmic scale for the sake of clarity. As expected, the contribution of the $n = 2$ diode gets smaller the larger the voltage. However, with small voltages, this behavior becomes more noticeable. This thesis proposes the simulation of broken and partially shaded PV panels. The applied voltage is in these occasions low, and the effect becomes noticeable. Furthermore, as it will be presented in Section 5.6, there is no additional running-time penalty if the two-diode model is used.

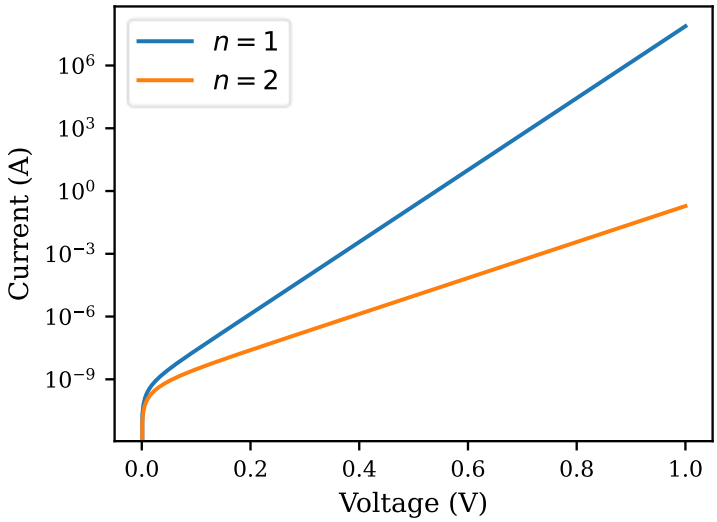


Figure 30. Diode current depending on ideality factor n.

PV cells are fragile, produce little power, and are difficult to handle. For these reasons, these PV cells are mounted into more robust PV panels, whose model will be presented in the following section.

4.2. PV Panel

Typically, the PV cells are arranged in a grid layout, and these cells are wired in series, and then in parallel, as shown in Figure 31. Most common PV panels mount 60(6x10), 72(6x12) and 120(6x20) PV cells. This figure shows a connection in series (also known as string) of 40 PV cells with two bypass diodes. The bypass diode is a device that allows current to flow should the any PV cell be shaded. As an example, Figure 31 shows a PV panel where one PV cell is shaded. No current will flow through the shaded cell as it can be inferred from Equations (157) and (158). This lack of current will force the whole panel to not output current as well. In this situation, almost all current of I'_{SC} will flow back through diodes D_1 and D_2. Just a single cell can drag the entire PV panel out. To prevent this issue, the bypass diode bypasses the problematic PV string and allows the rest of the PV cells to output current. In this figure, only the top PV cell string remains inactive. The red arrows show the flow of current in these circumstances.

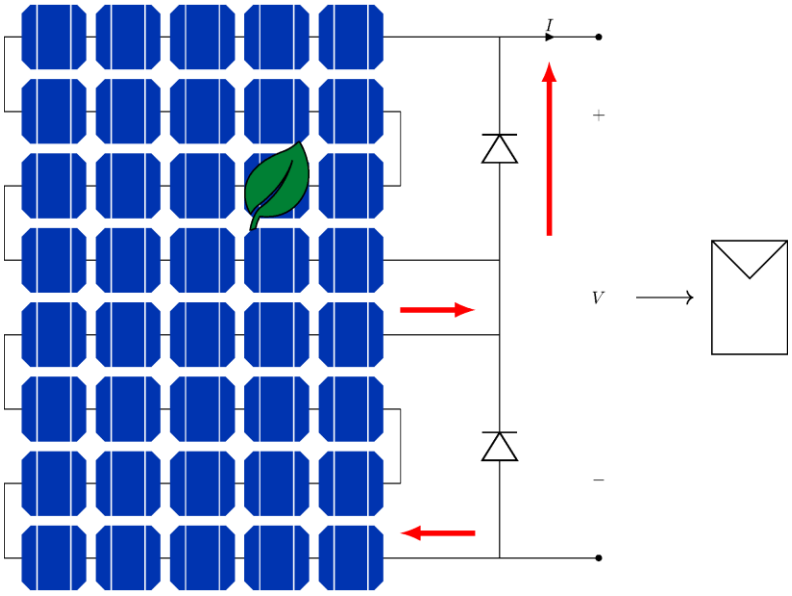


Figure 31. PV panel with two bypass diodes and a shaded PV cell.

The most basic unit in PVPPs is the PV panel, as PV cells are always encased in PV panels alongside the bypass diodes. For that reason, the PV panel has been treated as a standalone device in this thesis. Its electric circuit model is like the one shown in Figure 29, only adding the bypass diode, as Figure 32 shows. The bypass diode was modelled using the Shockley diode formula, shown in Equation (159), which is an approximation of the real behavior of the diode [117]. Its voltage corresponds to the negative of the PV panel voltage, and therefore the diode is reversed biased, not conducting under normal circumstances.

4.2.1. PV Panel Temperature

The output of the PV panel depends on the temperature of the panel. Larger temperatures downgrade the performance of the PV panel, and Figure 33 illustrates it. The temperature coefficient of most PV panels ranges from -0.3%/ºC to -0.5%/ºC, i.e., the maximum power output of the PV panel reduces approximately 0.4% per ºC.

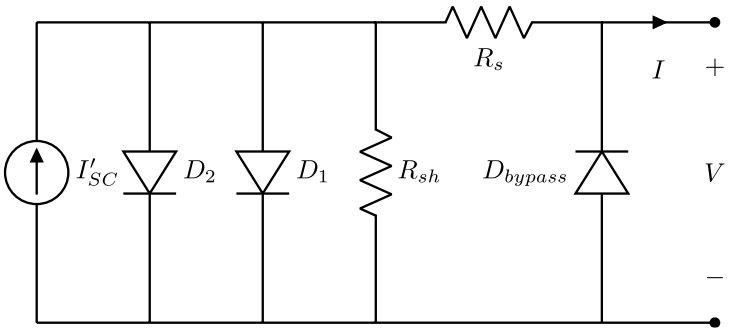


Figure 32. PV panel equivalent circuit.

$$I = I_S\left(e^{\frac{qV}{kT}} - 1\right) \tag{159}$$

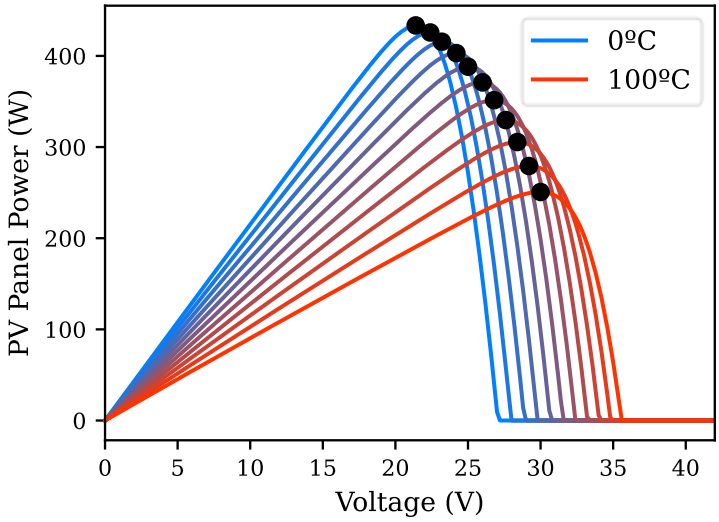


Figure 33. PV panel IV-curve while varying the temperature.

The Standard Test Conditions (STC) are the conditions under which PV panels are assessed. The test is conducted shining on the PV panel with a light of $1000\frac{W}{m^2}$ and a temperature of 25ºC. As mentioned, temperature greatly affects the performance of the PV panel, and this temperature might not be representative of the temperature of PV panels in real, operating conditions. Transfer of thermal energy affects the temperature of the PV panel. Equation (160) describes the change in PV panel temperature T_p considering that the PV panel is a uniform solid taking into account the most significant thermal phenomena. These phenomena are:

$$\frac{dT_p}{dt} = \frac{\alpha G S_p - \sigma\varepsilon S_p T_p^4 - h_p S_p\left(T_p - T_{air}\right) - P}{m_p c_p} \tag{160}$$

- **Emitted radiation:** the PV panel emits thermal radiation as all objects in nature. The Stefan-Boltzmann expression $\sigma\varepsilon S_p T_p^4$ calculates the power radiated, where σ is the Stefan-Boltzmann constant $\sigma = 5.67 \cdot 10^{-8}\frac{W}{m^2K^4}$ and ε is the emissivity of the PV panel, which is approximately $\varepsilon = 0.9$ [118].
- **Absorbed radiation:** the PV panel absorbs solar radiation as well. The absorbed radiation is directly proportional to the area perpendicular to the solar radiation S_p, the

current solar irradiance G, and its absorptivity α, being represented this behavior in Equation (160) with the $\alpha G S_p$ term. On one hand, the absorptivity is, according to Kirchhoff's law of thermal radiation equal to the emissivity $\alpha = \varepsilon$. On the other hand, the perpendicular area depends for fixed PV panels on both the solar inclination and the inclination of the PV panel itself. Large PVPPs aim to generate the largest possible energy, thus PV panels are installed on solar trackers (see Figure 34). The solar trackers orient the PV panels to face the sun perpendicularly, being the perpendicular area equal to the surface area of the PV panel. Finally, some irradiated power is not lost as thermal energy but converted into electrical energy P, and therefore its electrical output was subtracted from the irradiation term.

Figure 34. PV panels mounted on solar trackers (source: Amonix. Inc.).

- **Convection:** PV panels partly cool down thanks to air breezes. The heat transfer through convection is calculated by means of Newton's law of cooling, whose expression is $h_p S_p (T_p - T_{air})$, where h_p is the convection coefficient, and T_{air} is the air temperature. The convection coefficient depends on the wind speed, the shape of the object and other properties such as air density and air viscosity. There are empirical expressions that approximate this value and there are also indirect measurements that calculate the coefficient, such as [119], that state that $h_p \approx 10 \frac{W}{m^2 K}$ for a wind speed of $3 \frac{m}{s}$.
- **Conduction:** thermal conduction is the process by which heat is transferred from a hot object to a cold one by direct contact. This phenomenon is similar to convection which, on the other hand, transfers heat through the movement of matter. Conduction is most significant when two solids are in contact. As shown in Figure 34, the PV panels are mounted on PV solar trackers, which consist of a small frame mounted on a column. Therefore, conduction is not an important process to be considered, as the contact surface is negligible compared to the exposed surface of the PV panel to the ambient.

Equation (160) is an ODE and Figure 35 shows the temperature of a 400W PV panel according to that equation, when it changes considering that the air temperature is 30ºC, the irradiation is 1000 $\frac{W}{m^2}$, and that there is a moderate wind speed of $\frac{3m}{s}$. The starting conditions are the PV panel STC. The PV panel temperature rises as expected, albeit slowly. The system represented on Equation (160) can be approximated as a first-order linear system, and the rise time of this system is $903s$. On the other hand, the average wind speed in the regions with clouds

is in the (10,20) $\frac{m}{s}$ range, and cloud sizes can range from 50 m up to 1 km [120], so clouds can cover and uncover PV panels in less than a minute. The variation of temperatures is, therefore, an order of magnitude slower than the changes in irradiance. The variations of temperature will only slightly affect the variations in generated output, and if necessary, the temperature variation can be computed before solving the electrical circuit.

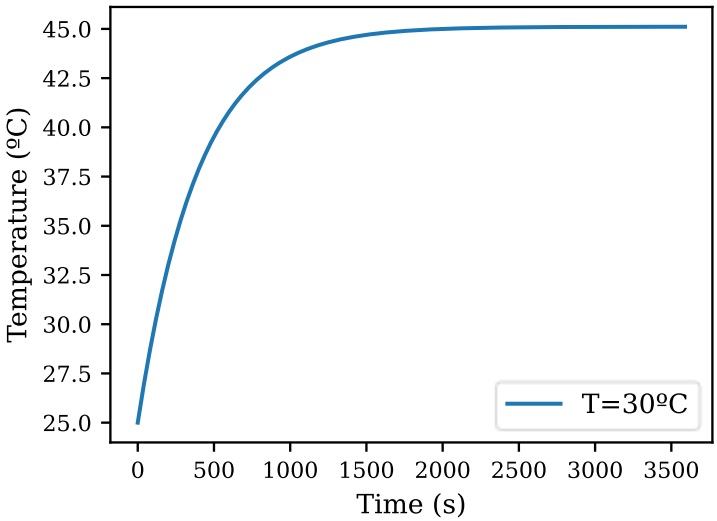


Figure 35. PV panel temperature variation.

4.2.2. PV Linear Equivalent

The PV panel model presented above features nonlinear devices such as diodes. These devices must be linearized to assemble the full system of the PVPP (see Section 5.6). Linearization entails the conversion of a nonlinear device into passive electronic models and constant sources of voltage or current. Let a generic nonlinear $I(v)$ be considered, whose current output depends only on its terminals' voltage. Its linearization around a voltage v_0 is shown in Equation (161). Reordering Equation (161) into Equation (162) hints to an intuitive equivalent circuit made from two parallel elements: a current source, and an admittance. The new linearized equivalent circuit is shown in Figure 36.

$$I(v) \approx I(v_0) + \left.\frac{dI(v)}{dv}\right|_{v=v_0} (v - v_0) \tag{161}$$

$$I(v) \approx \left(I(v_0) - v_0 \left.\frac{dI(v)}{dv}\right|_{v=v_0} \right) + v \left.\frac{dI(v)}{dv}\right|_{v=v_0} \tag{162}$$

These linearized elements will become part of the internal PVPP grid circuit. As they are linearized around a point, the larger the difference between the applied voltage and the linearization point voltage, the larger the error. The solver will approximate the solution iteratively, updating the linearization voltage until $v \approx v_0$.

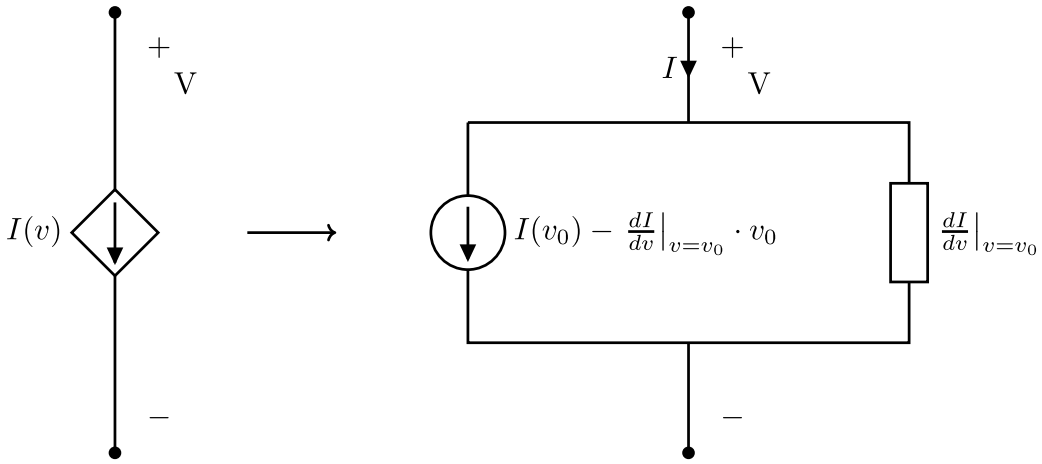
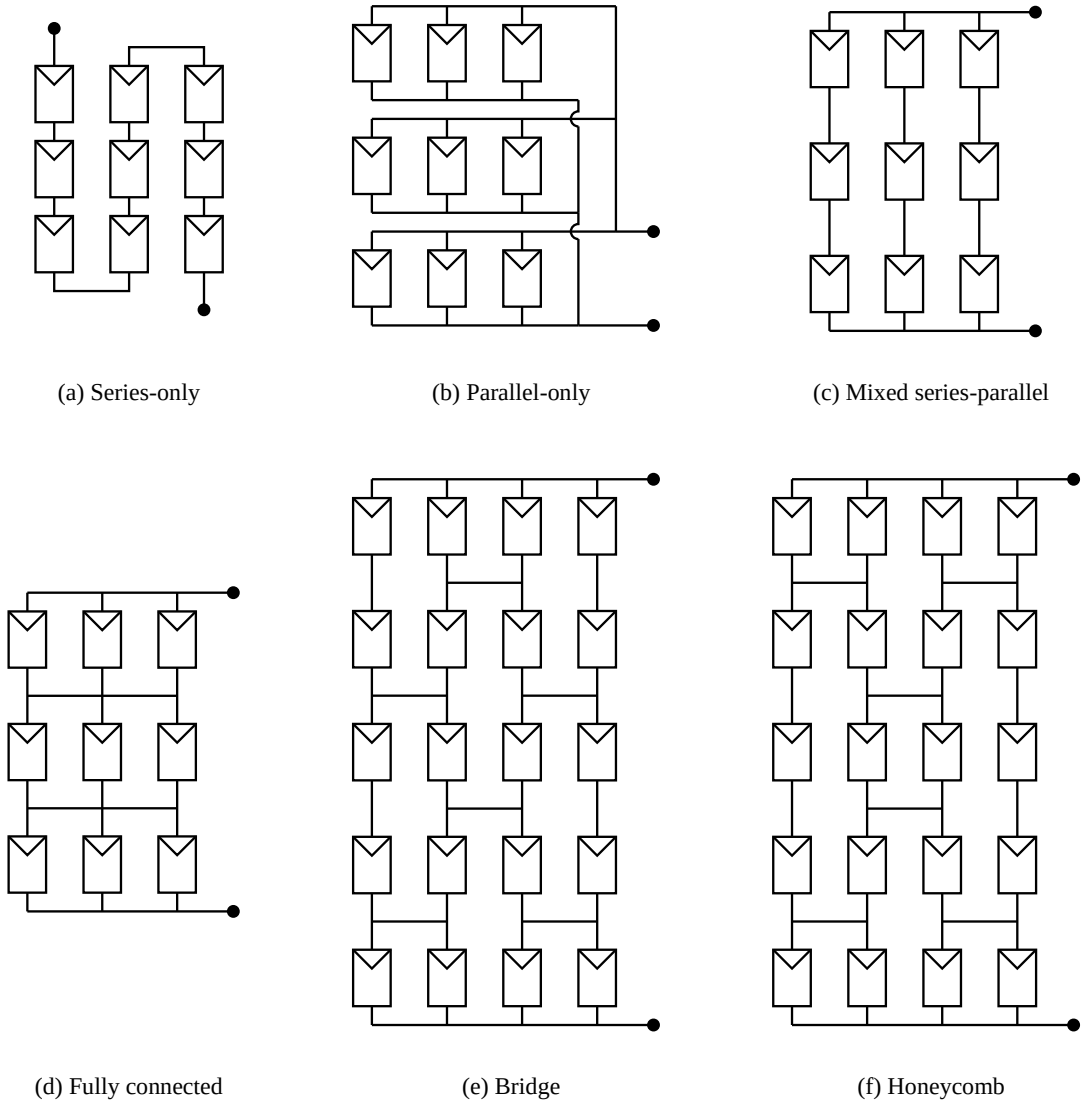


Figure 36. PV panel linearization.

4.3. PV Array

PV panels are devices that connect many PV cells to form a standalone device which can be mounted on rooftops or solar trackers. The maximum power output of the PV panels available in the market ranges from 300 W to 600 W. Most PV installations require to interconnect many PV panels to generate a sizeable amount of power. This group of PV panels is known as a PV array. Figure 37 shows the most widespread electrical connection layouts of PV arrays. One of the easiest connection schemes is the series-only layout. It consists of PV panels connected in series in what is known as a PV string. This design is suitable for small PV household installations due to its ease of maintenance and its low cost in wiring. The downside of this layout is that a fault in the PV string, whether in the wiring or a PV panel, grinds the entire installation to a halt. PV panels can be connected in parallel as well, in what is known as the parallel-only layout. This layout is more resilient than the series-only layout. Its downside is the low voltage of the installation. As shown in Figure 33, the optimal voltage ranges from 20 to 30 V. Many PV panels in parallel add up current, which means more losses in the wiring. A mixture of the previous two schemes is shown in Figure 37(c), in the mixed series-parallel configuration. This configuration strikes a balance between resilience and optimal voltage, and it is the most common PV panel layout [121]. To further protect losses in the PV strings, a blocking diode can be added as shown in Figure 38. The PV voltage across each individual PV panel can vary if any problem arises in any panel connected to it. The PV strings can be interconnected into a fully connected layout. This layout achieves the best performance, while its con is the large amount of wiring required. To alleviate this issue, some lateral connections can be omitted, forming the bridge and honeycomb layouts [121].



(a) Series-only (b) Parallel-only (c) Mixed series-parallel

(d) Fully connected (e) Bridge (f) Honeycomb

Figure 37. PV array configurations.

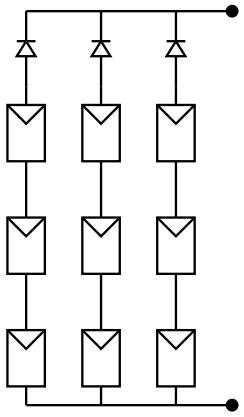


Figure 38. Mixed series-parallel layout with blocking diode.

4.4. Inverters

On one hand, PV panels work on DC power, while on the other hand PSs transport electrical AC power at a high voltage. The inverter is the device that converts DC generation from PV panels into AC voltage. This conversion is performed under appropriate conditions in order to generate the optimal power. The following section details the algorithm to find out the optimal voltage and discusses the electrical model of the inverter.

4.4.1. Maximum Power Point Tracking

Maximum Power Point Tracking (MPPT) is a controller that maximizes the extracted output of the PV panels. The inputs of the controller are the instantaneous DC voltage and current of the PV panels, although more complex controllers might have extra inputs such as the measured irradiation. The most widespread MPPT algorithm is known as perturb and observe [122], which is explained in Figure 39. The MPPT controller measures the voltage and current at an instant k and then, calculates the power and voltage difference with the current and previous timesteps. With these comparisons, the controller increases or decreases the voltage to find the optimal voltage. This operation is repeated in a loop for each measurement.

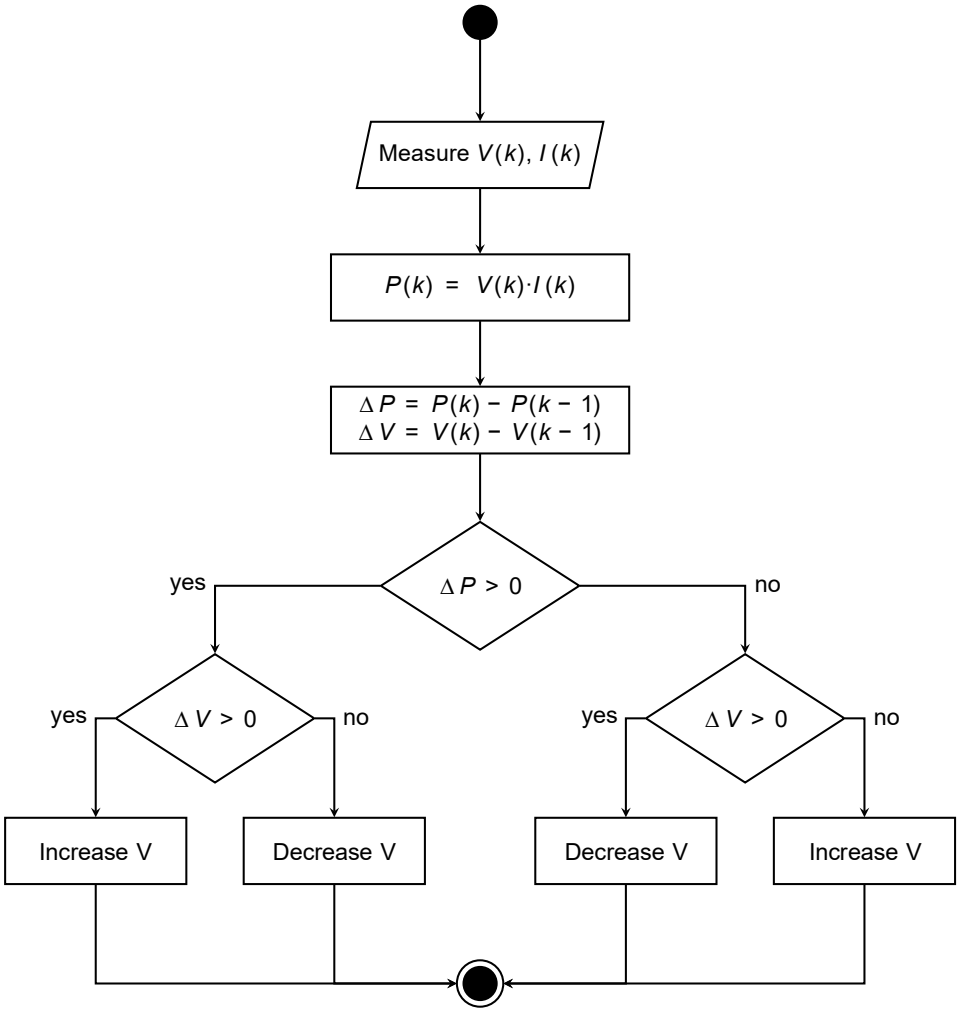


Figure 39. Perturb and observe algorithm.

The main issue of this method is that it does not reach the equilibrium point. The controller oscillates around the maximum power point. Other alternatives such as the incremental conductance method [122] avoid this issue by taking advantage of the derivate of the power function. The algorithm flowchart looks like Figure 39, but the compared quantities are ΔV, ΔI and $\frac{\Delta I}{\Delta V}$. Even simpler approaches calculate the optimal voltage value multiplying the open circuit voltage of the PV panel layout times a constant K, where K is a coefficient whose value ranges from 0.7 to 0.8 [122].

One issue with these approaches is that they converge to the local minima. Unequal shading creates local maxima, and the MPPT might be unable to reach the new optimal maxima. Figure 40 illustrates this issue. 20 PV strings with 20 PV panels each were simulated considering the STC. The irradiation of the first 10 PV panels of the first 10 PV strings was gradually lowered, from 100% to 0% with a step of 10%. The red dots of Figure 40 indicate the maximum generated output of the PV panels and the ideal voltage. As expected, the maximum generated output is lower the less irradiation shines upon the PV panels. This unequal shading produces a secondary maximum, which for the worst case (25% of the total of the PV panels are completely shaded) is the actual optimum value and is at odds with the other cases.

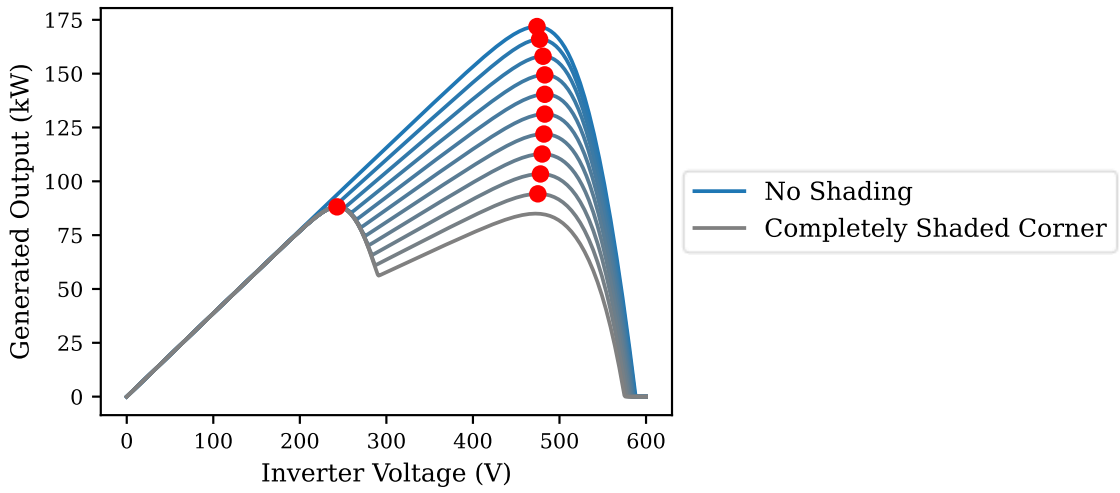


Figure 40. MPPT with shaded corner.

The industry standard control strategies do not find the optimal voltage under all conditions due to their iterative nature as explained above. The large PVPP that will be simulated in this work is considered as stationary, i.e., for a given temperature and irradiation map, the MPPT will have arrived at its optimal value. For the function of the generated power at a certain voltage $P(V)$, which is defined in Equation (163) as the product of the voltage set by the MPPT and the output current, its maximums verify Equation (164). The analytical expression of $I(V)$ is unknown, and the derivatives have been obtained through discretization. Applying the central difference discretization to the first and second derivatives of $P(V)$ renders Equation (163) into Equations (165) and (166).

Applying the Newton-Raphson method converges into the optimal value provided the initial value is chosen close to the actual solution. If the initial value strays away from the local maximum, then $\left|\frac{\partial^2 P(V_n)}{\partial V^2}\right| \approx 0$, and the Newton-Raphson iteration diverges. To fix this issue, it is proposed that each Newton-Raphson iteration is clamped, as shown in Equation (167), where

$n_{stringpanels}$ is the number of PV panels per PV string and α is a constant. It has been picked in this thesis $\alpha = 0.5$, due to its impressive results for a wide range of initial values and its speed of convergence.

$$P(V) = V \cdot I(V) \tag{163}$$

$$\frac{\partial P(V)}{\partial V} = 0 \tag{164}$$

$$\frac{\partial P(V_n)}{\partial V} \approx I(V_n) + \frac{I(V_n + \varepsilon) - I(V_n - \varepsilon)}{2\varepsilon} V_n \tag{165}$$

$$\frac{\partial^2 P(V_n)}{\partial V^2} \approx \frac{V_n\big(I(V_n + \varepsilon) - 2I(V_n) + I(V_n - \varepsilon)\big)}{\varepsilon^2} + \frac{I(V_n + \varepsilon) - I(V_n - \varepsilon)}{\varepsilon} \tag{166}$$

$$V_{n+1} = V_n - clamp\left(\frac{\left(\frac{\partial P(V_n)}{\partial V}\right)}{\left(\frac{\partial^2 P(V_n)}{\partial V^2}\right)}, -\alpha n_{stringpanels}, \alpha n_{stringpanels}\right) \tag{167}$$

The Newton-Raphson iteration is applied for each PVPP subsystem connected to each inverter to find the optimal voltage. Each set of PV panels only needs to be evaluated for three different voltages: $V_n - \varepsilon$, V_n and $V_n + \varepsilon$. Two iterations already converge in the optimal value if the initial value is taken around the theoretical optimal value under optimal conditions.

4.4.2. Electrical Model of the Inverter

The previous section detailed how the MPPT controller is considered in this thesis. As explained above, the state of the PVPP will be regarded as static, and it will be considered that the MPPT has already converged. The same approach will be used for the electrical model of the PVPP inverter. Power inverters are electrical devices that transform the DC input electrical power into AC power. The output AC frequency must match the specifications of either the isolated electrical device to be supplied or the frequency of the global PS. The PVPPs simulated throughout this thesis are considered to be located in Europe, therefore $f = 50Hz$. The inverter can further control the generated power by varying the magnitude and phase of the output current.

There are many inverter topologies and most of them are based on power electronic devices whose commutation create the output signal. Figure 41 shows the full electrical model of the two-level three-phase voltage-source inverter. Its input is connected to a PV panel layout as described in Section 4.3, which is then connected to a capacitor that stabilizes its voltage. The commutated circuits present in the inverter produce periodic variations of the DC voltage and current, called ripple, which should be smoothened out to ensure optimal performance of the PV panels. Then, many inverters are equipped with a DC-to-DC boost converter that rises the input voltage into a level suitable for further stages. The duty cycle, which is the ratio of the time an electronic device is ON compared to the total time, controls the output voltage and current through. The Insulated-Gate Bipolar Transistors (IGBTs) S_{b1} and S_{b2} are switched alternatively using high-frequency Pulse-Width Modulation (PWM) [39].

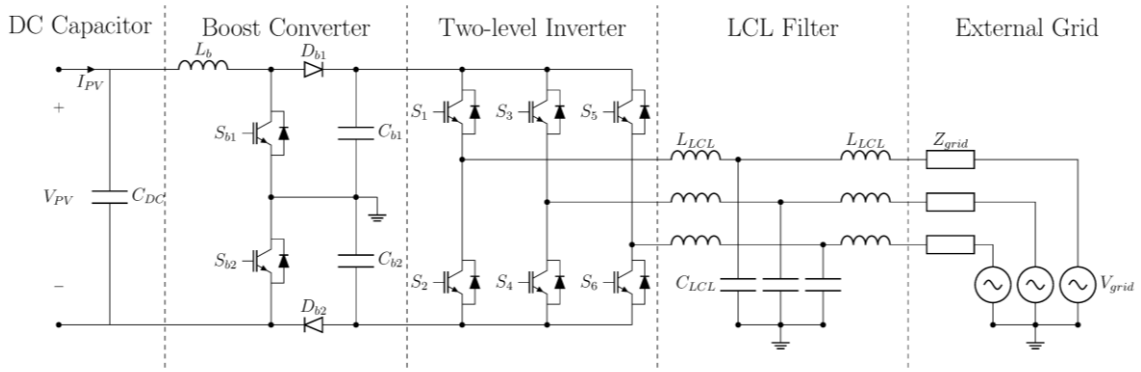


Figure 41. Two-level three-phase voltage-source inverter with boost converter.

The next stage of the inverter is its core, the two-level inverter circuit. Just as the boost converter, the IGBTs S_1 to S_6 are driven with a PWM modulation technique to produce a waveform that resembles a sinusoidal. This modulation can be a high-frequency saw-tooth function, known as carrier signal, which is compared to the input waveform, depending on whether the input is higher or lower than the sawtooth carrier signal; the semiconductor devices are either active or inactive. The pairs S_1,S_2 ; S_3,S_4 and S_5,S_6 are commuted alternatively. The stage shown in Figure 41 shows an inverter with a two-level inverter stage. Its output voltages can be, according to the circuit, $\frac{V_{boost}}{2}$ and $\frac{-V_{boost}}{2}$, where V_{boost} is the voltage output at both ends of the boost converter stage, as Figure 42 helps to illustrate. The magnitude and phase of this waveform can be controlled to match the requirements of active and reactive power of the PVPP. Its magnitude and phase match, notwithstanding losses, the desired wave at the fundamental frequency. The harmonics caused by the PWM are high-frequency, and the larger the frequency of the carrier signal, the greater the frequency of the harmonics [39].

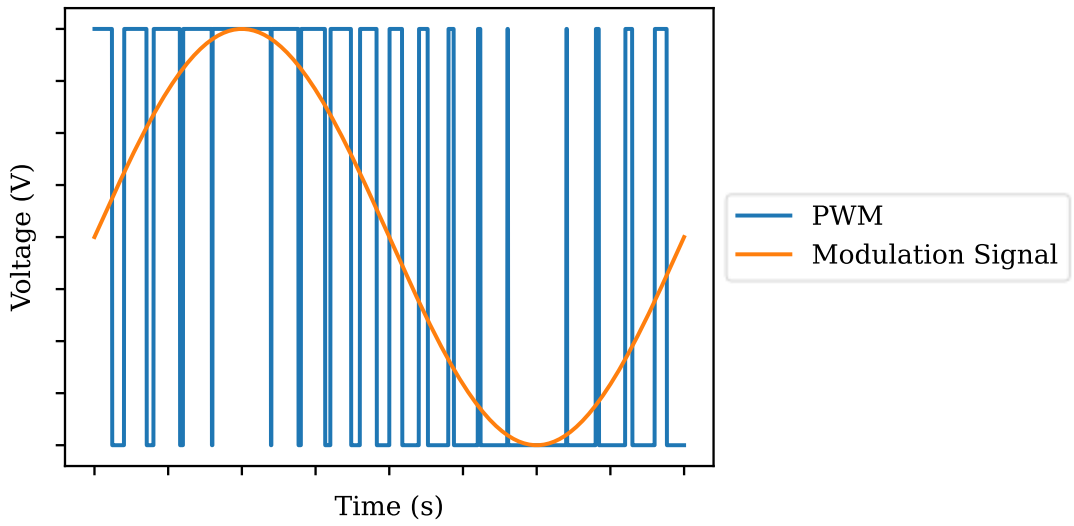


Figure 42. Input and output PWM waveforms.

The output of the inverter cannot be dumped into the PS. Grid operators set quality guidelines that limit the Total Harmonic Distortion (THD) of a given generator. They must be filtered away. A low-pass passive filter that attenuates that sits between the IGBT inverter and the external grid does this job. Common filter types such as RC, RL, and LC, where the resistors, coils and

capacitors and laid out and sized to dampen out the harmonics without affecting the main frequency. Figure 41 shows a LCL filter, which has been overwhelmingly adopted for inverters because comparatively better harmonic attenuation is achievable. Finally, the inverter is connected to an external grid. The external grid is characterized by a grid voltage and impedance. This point of connection is known as the Point of Common Coupling (PCC).

The detailed model of the PVPP solar inverter has been fleshed out up to now. On its most basic level, the inverter sets the PV panels DC voltage, and outputs a certain AC current into the PS. This model is time-dependent, as it has discrete elements with internal states, such as capacitors and coils. Furthermore, the PWM carrier signal, and its associated control strategy must be computed. The timestep required to properly simulate the inverter would be orders of magnitude smaller than the period (inverse of the frequency) of the external grid. A cursory glance of the inverter model shown in Figure 41 reveals that the average energy balance of the inverter is zero, as no large long-term energy storage elements are present. The previous sections have shown that the models of the PV panel layouts down to the PV cell do not have internal state, i.e., its output depends only on the inputs. It has also been shown that the effects of temperature variations change slowly. Furthermore, phenomena such as cloud occlusion can take up to 20 seconds to fully reduce the generated power [123]. On the other hand, the electrical transients take much less time as Figure 43 shows. It has been highlighted in red the speed at which clouds pass by and provoke changes in solar irradiation. Quick electrical transients such as lightning and over voltage operation are well under the range of the frequencies of interest. Short circuits are also fast phenomena, but the time to fully clear them up can be large, hence its broad range.

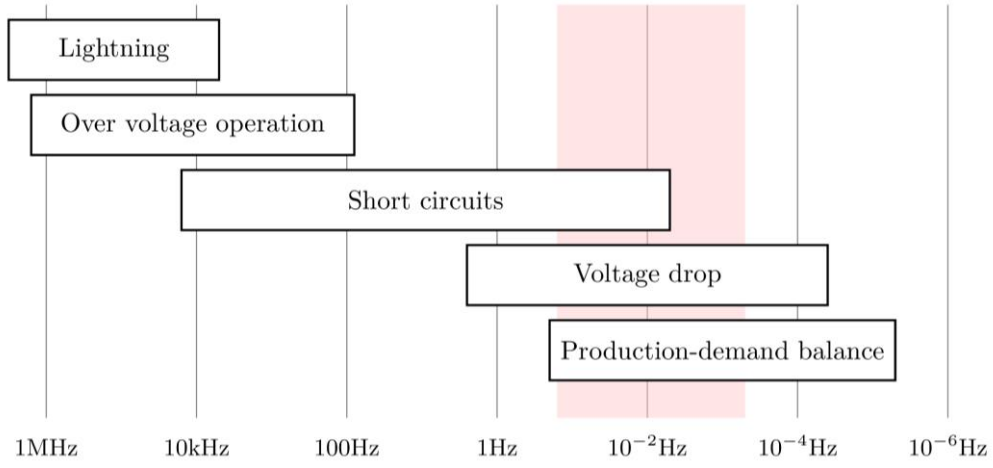


Figure 43. Electric phenomena speed (adapted from [124]).

For that reason, it has been proposed in this thesis a simplified PV inverter model, shown in Figure 44, which transforms the power between DC and AC ideally. The DC size is represented as a controllable voltage source, while the AC size is represented by a current source. The input P_i and output P_o is identical. Therefore, $P_i = P_o$, hence the expression (168).

$$V_{DC} I_{PV} = 3 V_{grid} I_{grid}^{*} \tag{168}$$

The DC input voltage corresponds to the optimal voltage of the PV array which is calculated as shown in Section 4.4.1. On the other hand, the AC current is calculated solving Equation (168). For each iteration, the calculated current is calculated, and used in the next iteration. The next

chapter of this thesis, Chapter 5, will show how the interconnection of various elements treats these current sources.

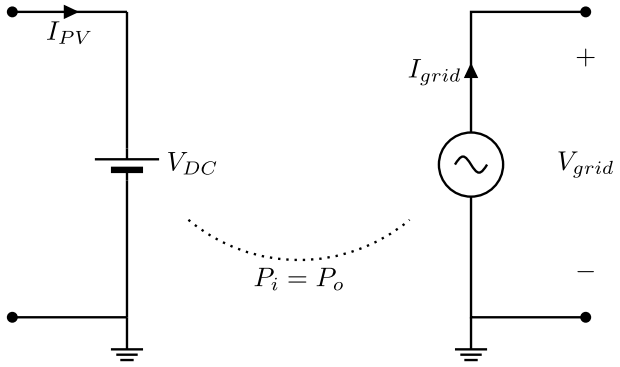


Figure 44. Simplified PVPP inverter model.

4.5. Summary of PVPP Equations

This section concludes the chapter of this thesis by summarizing all equations that describe the PVPP in one section in a single list as shown in the Equations (169)-(178).

$$I = I'_{SC} - I_{O1}\left(e^{\frac{q(V+I\cdot r_s)}{kTn_a}} - 1\right) - I_{O2}\left(e^{\frac{q(V+I\cdot r_s)}{2kTn_a}} - 1\right) - \frac{V + I\cdot r_s}{r_{sh}} \tag{169}$$

$$I'_{SC} = I_{SC}\frac{G}{G_0}\cdot\left(1 - \mathrm{K_0(T - T_0)}\right) \tag{170}$$

$$I = I_S\left(e^{\frac{qV}{kT}} - 1\right) \tag{171}$$

$$\frac{dT_p}{dt} = \frac{\alpha G S_p - \sigma\varepsilon S_p T_p^4 - h_p S_p\left(T_p - T_{air}\right) - P}{m_p c_p} \tag{172}$$

$$I(v) \approx \left(I(v_0) - v_0\left.\frac{dI(v)}{dv}\right|_{v=v_0}\right) + v\left.\frac{dI(v)}{dv}\right|_{v=v_0} \tag{173}$$

$$P(V) = V\cdot I(V) \tag{174}$$

$$\frac{\partial P(V_n)}{\partial V} \approx I(V_n) + \frac{I(V_n+\varepsilon) - I(V_n-\varepsilon)}{2\varepsilon}V_n \tag{175}$$

$$\frac{\partial^2 P(V_n)}{\partial V^2} \approx \frac{V_n\left(I(V_n+\varepsilon) - 2I(V_n) + I(V_n-\varepsilon)\right)}{2\varepsilon} + \frac{I(V_n+\varepsilon)I(V_n-\varepsilon)}{\varepsilon} \tag{176}$$

$$V_{n+1} = V_n - clamp\left(\frac{\left(\frac{\partial P(V_n)}{\partial V}\right)}{\left(\frac{\partial^2 P(V_n)}{\partial V^2}\right)}, -\alpha n_{stringpanels}, \alpha n_{stringpanels}\right) \tag{177}$$

$$V_{DC}I_{PV} = 3V_{grid}{I_{grid}}^* \tag{178}$$

Chapter 5. Modeling of Power Lines

The previous chapters detailed the models of the renewable energy power plants that will be used throughout this thesis, which are the DFIG, the PV panel and the PV inverter. This chapter will be devoted to the model of the power lines that will interconnect each element.

Power lines are pieces of conducting metal that transport electrical power across their ends. They can be classified by their construction, where there are overhead power lines and underground cables; by their nominal voltage: low (up to 1000 V DC, or 1500 V AC), medium (1 to 36 kV), or high (36+ kV)[12]; by the material of the metal core or the isolation layer; by the geometry between the power line and adjacent power lines; and by their intended use, where power lines transmit power across large distances on high voltage and distribution lines do so at low voltages, for low distances, in order to supply consumers. In essence, all power lines, from a power line that spans hundreds of kilometers to the smallest wire are similar, and so are their models. The difference between models stems from the precision and accuracy of the solution. The chosen power line model is the π-section line model, presented below, in Section 5.1 although this section will briefly introduce other types of models, such as the Bergeron power line model in Section 5.2 and briefly the frequency-dependent power line model in Section 5.3. These models will be compared in Section 5.4 and, finally, the π-section line model will be expressed and discretized in Section 5.5. This chapter concludes with a brief description of the sparse matrix assembled from all power line discretized models in Section 5.6.

5.1. π-Section Power Line Model

The flow of power through a power line can be calculated, for instance, using the electromagnetic field theory of Maxwell, or simply considering the power line as an ideal and lossless conductor. There is a power line model that sits between these two levels of abstraction, and it is the π-section line model, also known as the lumped model. Its name originates from the shape of its equivalent circuit as it can be noticed in Figure 45, where it is shown the representation of the full power line, whose inputs are the voltages and currents V_1, V_2, I_1 and I_2. Given a frequency, conductor type, and line configurations; the power line has a characteristic distributed resistance, inductance, capacitance, and conductance[13] named as r, l, c and g, respectively. Calculating the lumped values of the passive elements of this model requires integrating a differential element of the power line, whose equivalent circuit is identical to Figure

[12] The threshold for low voltage is defined on Spanish regulations [125]. These regulations do not define the medium voltage section. It is nevertheless a well stablished convention in the industry to differentiate between medium and high voltage setting the boundary at 36 kV.

[13] Most models omit the conductance, therefore $g = 0$. The power lines and wires considered in this thesis have no conductance.

45. Moving the shunted capacitors and conductances together gives the approximations (179)-(182), where ℓ is the length of the line in km.

$$R = r\ell\ [\Omega] \tag{179}$$

$$L = l\ell\ [H] \tag{180}$$

$$C = c\ell\ [F] \tag{181}$$

$$G = g\ell\ [S] \tag{182}$$

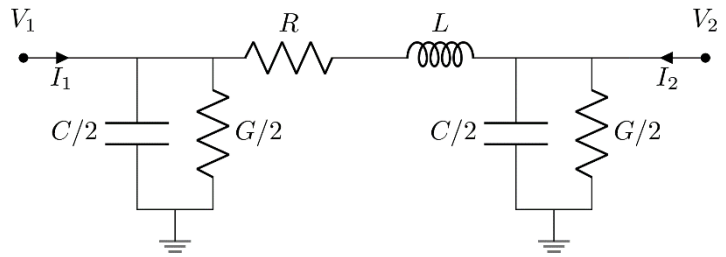


Figure 45. π-section line model.

The previous approximation is acceptable for short to medium power lines. Should long power lines be considered, then the previous expressions must be multiplied with a correction factor. This correction factor can be deduced by considering a π-section line with differential length and integrating the differential power line previously mentioned without reordering the shunting branches. The correction factors depend on the γ constant, whose value is shown in Equation (183). The final expressions with the correction factors are shown in Equations (184)-(187). More information on the mathematical background of these expressions can be consulted in [126].

$$\gamma = \sqrt{(r + j\omega l) \cdot (g + j\omega c)} \tag{183}$$

$$R = r\ell \cdot \frac{\sinh(\gamma\ell)}{\gamma\ell}\ [\Omega] \tag{184}$$

$$L = l\ell\ \cdot \frac{\sinh(\gamma\ell)}{\gamma\ell}[H] \tag{185}$$

$$C = c\ell \cdot \frac{\frac{\gamma\ell}{2}}{\tanh\left(\frac{\gamma\ell}{2}\right)}\ [F] \tag{186}$$

$$G = g\ell \cdot \frac{\frac{\gamma\ell}{2}}{\tanh\left(\frac{\gamma\ell}{2}\right)}\ [S] \tag{187}$$

The corrected π-section line achieves better results for long distances, but it is still not flawless. It does not consider the speed of the voltage and current waveforms. Thus, a sudden variation in the conditions on one end immediately affects the other end if a π-section line model is considered. Alternative models, such as the Bergeron line model partially addresses these issues.

Finally, to conclude this section, it should be noted that transformers can be represented as π-section line model by representing them through their equivalent inductance if the per unit system is used [31] as they can be equated to a lumped line where both the resistance and conductance

are zero. When doing so, electrical quantities do not change when referring to one side of the transformer or the other.

5.2. Bergeron Power Line Model

The power line characterized by a π-section differential element forms a system of partial differential equations differentiated along time and length. These Equations, (188) and (189), are the widely known telegrapher's equations. Differentiating one equation by the length and substituting the other equation to remove the dependence between voltage and current yields the Equations (190) and (191).

$$\frac{\partial V(x,t)}{\partial x} = -l\frac{\partial I(x,t)}{\partial t} - rI(x,t) \tag{188}$$

$$\frac{\partial I(x,t)}{\partial x} = -c\frac{\partial V(x,t)}{\partial t} - gV(x,t) \tag{189}$$

$$\frac{\partial^2 V(x,t)}{\partial x^2} = lc\frac{\partial^2 V(x,t)}{\partial t^2} + (rc+lg)\frac{\partial V(x,t)}{\partial t} + rgV(x,t) \tag{190}$$

$$\frac{\partial^2 I(x,t)}{\partial x^2} = lc\frac{\partial^2 I(x,t)}{\partial t^2} + (rc+lg)\frac{\partial I(x,t)}{\partial t} + rgI(x,t) \tag{191}$$

As it was mentioned, the shunt admittance g is omitted in most power line models, hence $g = 0$. To study these equations the resistive r can be omitted as the inductive and capacitive terms are more influential. As an example, the 33 kV power line with parameters $r = 0.01526\frac{\Omega}{km}$, $l = 0.8838\frac{mH}{km}$, $c = 0.0126\frac{\mu F}{km}$ will be considered [127]. The opposition to the flow of AC current by a coil is known as the reactance, with variable name x, and it is equal to 0.2777 $\frac{\Omega}{km}$ if a frequency of 50Hz is considered, verifying $r \ll x$.

Applying these simplifications, Equations (190) and (191) transform into the wave equation, where the wave speed v is equal to Equation (192). It is then trivial to calculate the propagation time τ of a wave between both ends, as shown in Equation (193). Solving these equations verifies the Equation (195), where Z_c is the characteristic impedance, also known as surge impedance, whose expression is shown in Equation (194). The Equation (195), represents a travelling wave from point 1 to 2 that takes a time τ, while Equation (196) shows its counterpart when a wave travelling the opposite direction is considered. These expressions can be reordered as shown in Equations (197) and (198), and expressed as a dependent current source and resistance as presented in Equations (199) and (200), where I_{Ber1} and I_{Ber2} are the current sources that depend on the state of the end, and Z_{Ber} is equal to Z_c. Z_c is also known as the surge impedance. Figure 46 represents its equivalent electrical circuit.

$$v = \frac{1}{\sqrt{lc}} \tag{192}$$

$$\tau = \frac{\ell}{v} \tag{193}$$

$$Z_c = \sqrt{\frac{l}{c}} \tag{194}$$

$$V_1(t-\tau) + Z_c I_1(t-\tau) = V_2(t) - Z_c I_2(t) \tag{195}$$

$$V_1(t) - Z_c I_1(t) = V_2(t-\tau) + Z_c I_2(t-\tau) \tag{196}$$

$$I_1(t) = \frac{V_1(t)}{Z_{Ber}} + I_{Ber1}(t) \tag{197}$$

$$I_2(t) = \frac{V_2(t)}{Z_{Ber}} + I_{Ber2}(t) \tag{198}$$

$$I_{Ber1}(t) = -\frac{V_2(t-\tau)}{Z_c} - I_2(t-\tau) \tag{199}$$

$$I_{Ber2}(t) = -\frac{V_1(t-\tau)}{Z_c} - I_1(t-\tau) \tag{200}$$

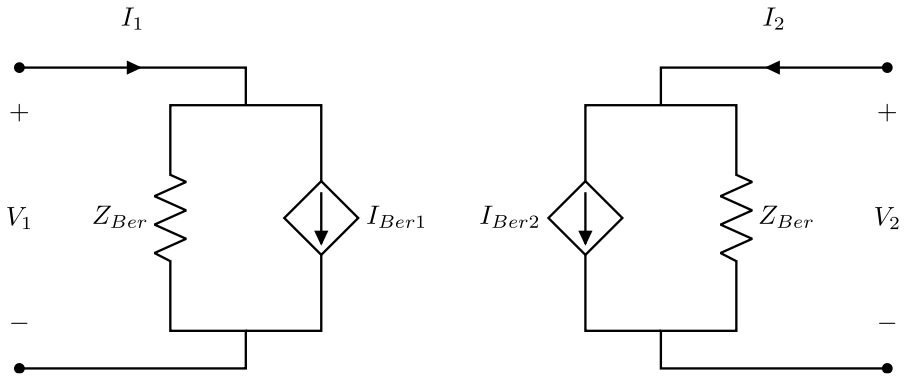


Figure 46. Two-Section Bergeron line model.

The expressions presented above do not consider the power line losses. A second version of the Bergeron model includes the resistance of the model by concatenating two lossless Bergeron line models of the line at half-length and then distributing the resistance. The resistance is represented as four lumped elements, each with a fourth of the total power line resistance and placed as shown in Figure 47. In this figure, the two middle resistance lumped elements are in series and have been expressed as $\frac{R}{2}$. This model can be simplified as a Norton equivalent, obtaining an equivalent circuit similar to Figure 46, where the values of Z_{Ber}, $I_{Ber1}(t)$ and $I_{Ber2}(t)$ are shown in the Equations (201)-(203).

$$Z_{Ber} = Z_c + \frac{R}{4} \tag{201}$$

$$\begin{aligned} I_{Ber1}(t) = \frac{R}{4Z_{Ber}} &\left(-\frac{V_2(t-\tau)}{Z_{Ber}} - \frac{Z_c - \frac{R}{4}}{Z_{Ber}} I_2(t-\tau) \right) \\ &+ \frac{Z_c}{Z_{Ber}} \left(-\frac{V_1(t-\tau)}{Z_{ber}} - \frac{Z_c - \frac{R}{4}}{Z_{Ber}} I_1(t-\tau) \right) \end{aligned} \tag{202}$$

$$I_{Ber2}(t) = \frac{R}{4Z_{Ber}}\left(-\frac{V_1(t-\tau)}{Z_{Ber}} - \frac{Z_c - \frac{R}{4}}{Z_{Ber}} I_1(t-\tau)\right) + \frac{Z_c}{Z_{Ber}}\left(-\frac{V_2(t-\tau)}{Z_{ber}} - \frac{Z_c - \frac{R}{4}}{Z_{Ber}} I_2(t-\tau)\right) \tag{203}$$

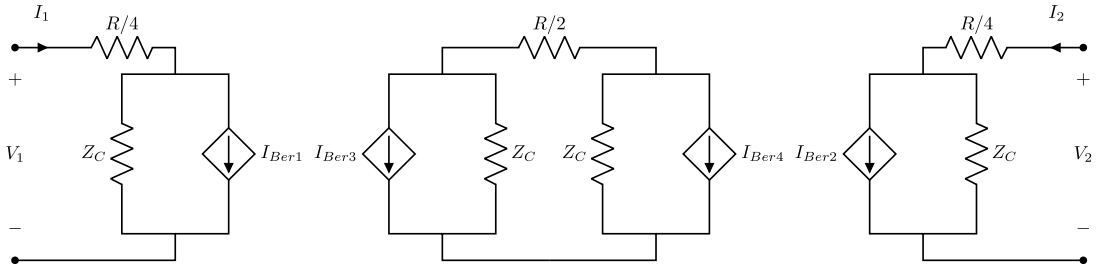


Figure 47. Three-Section Bergeron line model.

5.3. Frequency-Dependent Power Line Models

The previous power lines models presented in the section feature the resistance, inductance, capacitance, and shunt admittance as parameters, whether in lumped or distributed form. In fact, the resistance, inductance, and capacitance are not constant but depend on the frequency of the signal. The π-section line equivalent circuit with correction factors can be solved in the frequency domain by converting the capacitance and inductances into phasors. Then, a Fourier transform is applied to get the time domain solution. The main advantage of these methods is that they produce more accurate results at the expense of much greater computational cost. More information on frequency dependent power line models can be consulted in [128] which, as shown in the reference, look similar to Bergeron power line models.

5.4. Comparison of Power Line Models

The previous subsections introduced two types of power line models, the π-section line model and the Bergeron line model; and mentioned the existence of frequency-dependent power line models. There are two types of installations that will be studied this thesis: the wind farm internal grid and the PVPP internal grid. For the latter, the output of the PVPP will be regarded as stationary. In that situation, the PV panels are outputting DC power, and the PVPP, having reached equilibrium, will not suffer further variations. The Bergeron and frequency-dependent power line models are not suitable for DC-steady operation and have been discarded. Therefore, a simplified version of the π-section power line model was used, where the inductance and capacitance were omitted. Most models omit the shunt admittance due to its small value and was omitted here as well.

The issue about which model suits better the studied systems is arisen for the AC part of the PVPP and the power lines of the wind farm. It was studied the output of each model for the power

lines of the case studies. Figure 48 represents the input and output of a 100 km long power line, whose parameters are listed in Annex B.2. On one end, V_1 is set at a constant 35 kV AC at 50 Hz. The other end of the power line is connected to an impedance which consumes 10 MW of power. The voltage drop at the second end of the power line V_2 is equal to V_1 for an ideal conductor, and indeed, it is close to it due to the small voltage drop in the power line. The output of both the π-section line and the Bergeron similar are similar as the figure suggests.

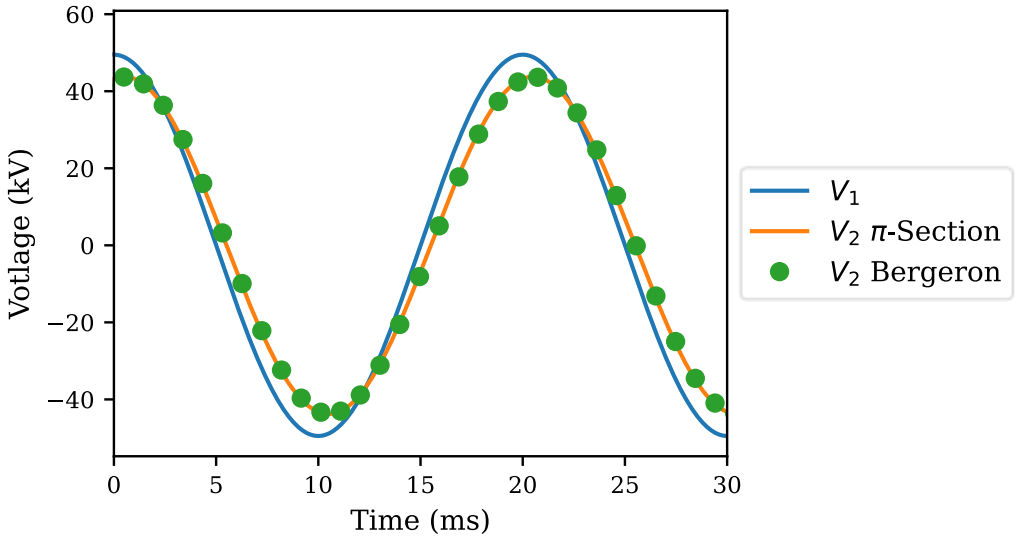


Figure 48. π-section power line model vs Bergeron power line model.

It was considered at first to represent the AC lines as Bergeron models. The resulting power system structure is very easy to parallelize (see Section 2.4.4), and this strategy has already been applied for parallel simulation with good results [85], [88]. However, the Bergeron model was finally ruled out due to its major downside: its maximum integration step. The figure suggests it is required at most a very small integration to properly use this model. Considering the speed of light, and a integration step of 50 μs, the Bergeron model is suitable only for power lines with over 15km in length [126]. The frequency-dependent power models are an extension of the Bergeron model and must be regarded as such. For these reasons, the π-section power line was finally chosen for the power systems studied in this thesis.

5.5. Dq0 Equivalent and Discretization

The π-section representation of power lines is the suitable model to simulate the connections between wind turbines and PV panels. Simulating the stationary state of the PVPP removes the need of passive devices with internal state from the power line model, as all derivatives of the fundamental expression of the inductor (205) and capacitor (206) are zero. These are differential equations and can be numerically integrated as introduced in Section 3.9. The issue with straightforward integration is its instability. Sudden changes in either the voltage and/or current require exceedingly small integration steps to be simulated properly, as it was shown in Figure 24. For that reason, the devices of the π-section line model defined by the Equations (204)-(206) must be discretized. The power line model was also transformed into the *dq0* frame of reference, which has been already described in Section 3.2.1. Transforming it to the *dq0* frame of reference will avoid potentially costly transformations between the *abc* and the *dq0* frames, as it requires the

computation of sines. Another advantage of the *dq0* frame of reference is the gained stability of the system. In the stationary state, all three *dq0* signals are constant, while the abc voltage or current signals oscillate.

$$V_r = RI_r \tag{204}$$

$$V_l = L\frac{dI_l}{dt} \tag{205}$$

$$I_c = C\frac{dV_c}{dt} \tag{206}$$

The π-section line equivalent is made of two branches: capacitor and resistor with an inductor in series (see Figure 45). Assuming that the RLC values of the power line are equal in all phases, and that there is no mutual inductance, the Equations (204)-(206) give rise to the Equations (207) and (208). The previous equations are defined in absolute values and in the abc reference frame. Plugging the transformation expression from *dq0* to abc and from per-unit to absolute values allows to express the equations in per-unit and *dq0* values.

$$v_b\left[T_{dq0\to abc}\right]V_{dq0-pu} = R_{pu}R_b\left[T_{dq0\to abc}\right]I_{dq0-pu} + L_{pu}L_b\frac{d\left(\left[T_{dq0\to abc}\right]I_{dq0-pu}\right)}{dt} \tag{207}$$

$$I_b\left[T_{dq0\to abc}\right]I_{dq0-pu} = C_{pu}C_b\frac{d\left(\left[T_{dq0\to abc}\right]V_{dq0-pu}\right)}{dt} \tag{208}$$

Cancelling out the base values and transformation matrices gives the expressions (209) and (210), where ω is the rotating speed of the $dq0$ frame of reference.

$$V_{dq0-pu} = \begin{bmatrix} R_{pu}I_{d-pu} - L_{pu}I_{q-pu} + \frac{L_{pu}}{\omega}\frac{dI_{d-pu}}{dt} \\ R_{pu}I_{q-pu} + L_{pu}I_{d-pu} + \frac{L_{pu}}{\omega}\frac{dI_{q-pu}}{dt} \\ R_{pu}I_{0-pu} + \frac{L_{pu}}{\omega}\frac{dI_{0-pu}}{dt} \end{bmatrix} \tag{209}$$

$$I_{dq0-pu} = \begin{bmatrix} -C_{pu}V_{q-pu} + \frac{C_{pu}}{\omega}\frac{dV_{d-pu}}{dt} \\ C_{pu}V_{d-pu} + \frac{C_{pu}}{\omega}\frac{dV_{q-pu}}{dt} \\ \frac{C_{pu}}{\omega}\frac{dV_{0-pu}}{dt} \end{bmatrix} \tag{210}$$

If the system is balanced, the *0* component will be always zero, and can be omitted. Omitting the *0* component leaves the *d* and *q* component. These components can be identified with complex numbers, where the real part corresponds to the *d* component, and the imaginary with the *q* component, which gives the Equations (211) and (212).

$$V_{pu} = \left(R_{pu} + L_{pu}j\right)I_{pu} + \frac{L_{pu}}{\omega}\frac{dI_{pu}}{dt} \tag{211}$$

$$I_{pu} = C_{pu}jV_{pu} + \frac{C_{pu}}{\omega}\frac{dV_{pu}}{dt} \tag{212}$$

The next step is to perform the discretization itself. The electrical transients are stiff, and it was shown that discretizing using the backward Euler method produced the most stable results [36]. Integrating for a timestep Δt returns the expressions (213) and (214), which are similar to the widely known discretization techniques presented in [35].

$$V_{pu}(t) = \left(R_{pu} + L_{pu}j\right)I_{pu}(t) + \frac{L_{pu}}{\omega\Delta t}\left(I_{pu}(t) - I_{pu}(t-\Delta t)\right) \tag{213}$$

$$I_{pu}(t) = C_{pu}jV_{pu}(t) + \frac{C_{pu}}{\omega\Delta t}\left(V_{pu}(t) - V_{pu}(t-\Delta t)\right) \tag{214}$$

Reordering the Equations (213) and (214) to express them in terms of the voltage at t and the values of the voltage and current at $t-\Delta t$ gives the Equations (215) and (216).

$$I_{pu}(t) = \left(R_{pu} + \frac{L_{pu}}{\omega\Delta t} + L_{pu}j\right)V_{pu}(t) + \frac{L_{pu}}{\omega\Delta t\left(R_{pu} + L_{pu}j\right) + L_{pu}}I_{pu}(t-\Delta t) \tag{215}$$

$$I_{pu}(t) = \frac{C_{pu}(1+\omega\Delta tj)}{\omega\Delta t}V_{pu}(t) + \frac{-C_{pu}}{\omega\Delta t}V_{pu}(t-\Delta t) \tag{216}$$

These expressions can be read as an admittance in parallel with a current source. The equivalent circuit is shown in Figure 49. It has been divided by 2 the expressions related to the capacitor elements to account for the two branches of the π-section line (see Figure 45).

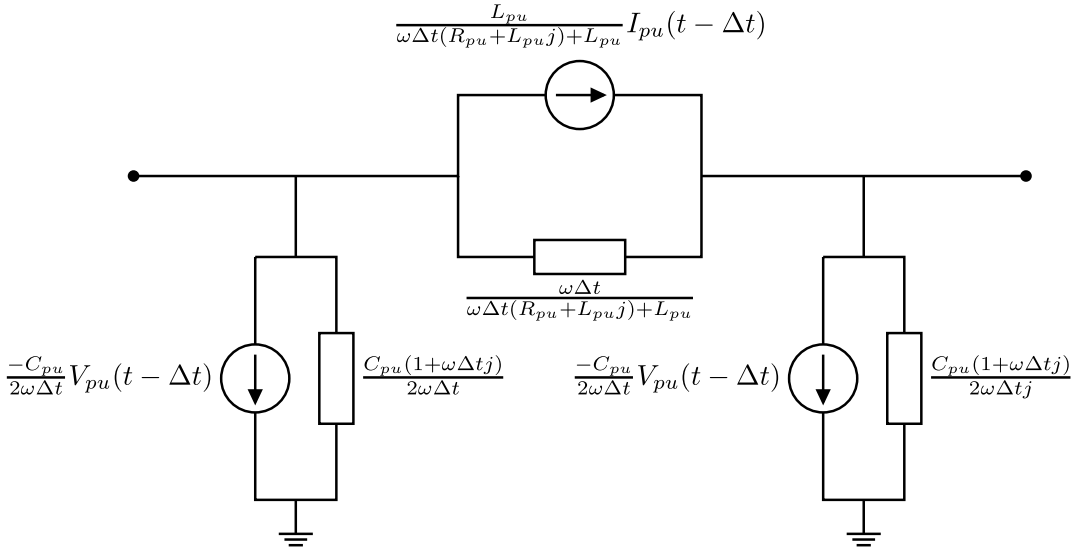


Figure 49. Discretized π-section power line model.

5.6. Sparse Matrix Representation

The discretization carried out in the previous subsection left a power line model with complex admittances and current sources. As for wind turbines, the models already expounded in the thesis have as outputs its stator currents. These stator currents can be represented as current sources that inject current in their respective connection points. Furthermore, the nonlinear elements of the PV panels are linearized as an admittance plus a current source to solve them. The remaining elements in the PV panel model are admittances, current sources, and the shunt and series

resistors, as shown in Chapter 4. The inverter has been represented as a current source and a voltage source.

As it can be inferred, the whole wind farm and PVPP can be represented as a series of current sources and admittances. Applying Kirchhoff's current law for each node and Ohm's law for every branch of the circuit allows the equations to be organized into a linear system of n equations, where n is the number of nodes of the system as shown in Equation (217)[14]. The unknowns are arranged in the vector V_{dq}, where each component represents the voltage at the node n; and I_{dq} is the vector that represents the sum of injected currents at node n. If multiple current sources are connected to the same nodes, their currents are summed, thus not resulting a larger matrix with additional computing cost. $[Y]$ is the matrix admittance and Equation (218) and Equation (219) show how it is filled out. For every element in the diagonal, Y_{ii} is equal to the sum of all admittances connected to the node i, while Y_{ij} is equal to the admittance between node i and node j. As it can be inferred, the matrix is symmetrical.

$$[Y] \cdot V_{dq} = I_{dq} \tag{217}$$

$$\begin{bmatrix} Y_{11} & \cdots & Y_{1N} \\ \vdots & \ddots & \vdots \\ Y_{N1} & \cdots & Y_{NN} \end{bmatrix} \begin{bmatrix} V_{dq_1} \\ \vdots \\ V_{dq_N} \end{bmatrix} = \begin{bmatrix} I_{dq_1} \\ \vdots \\ I_{dq_N} \end{bmatrix} \tag{218}$$

$$Y_{ij} = \begin{cases} i \neq j, admittance\ between\ i\ and\ j \\ i = j, \Sigma\ admittance\ nodes\ connected\ to\ i \end{cases} \tag{219}$$

Not all voltages are unknown, as the voltage at the PCC is known. To account for these issues, the linear system (218) is transformed into the system (220), where the known voltage has been moved to the Right-Hand Side (RHS). If there are no admittances connected to ground, the determinant admittance matrix as shown in the linear system (218) is zero and, therefore, the linear system would be unsolvable. This reduced matrix has now a definite solution and can be solved.

$$\begin{bmatrix} Y_{22} & \cdots & Y_{2N} \\ \vdots & \ddots & \vdots \\ Y_{N2} & \cdots & Y_{NN} \end{bmatrix} \begin{bmatrix} V_{dq_2} \\ \vdots \\ V_{dq_N} \end{bmatrix} = \begin{bmatrix} I_{dq_2} - Y_{21}V_1 \\ \vdots \\ I_{dq_N} - Y_{N1}V_1 \end{bmatrix} \tag{220}$$

A key feature of this matrix is that it is sparse. As mentioned before, a matrix is called sparse if a relatively small number of its elements is zero [53]. This fact will be essential to solve the linear system efficiently, and the CUDA kernel developed to solve it must account for that. This matrix accounts only for the inclusion of admittances and current sources. Additional voltage sources, such as the ones that model the inverters (see Section 4.4) require to extend the matrix into what is known as Modified Nodal Analysis (MNA) [129]. The new expanded matrix is no longer symmetrical.

[14] For the remaining of the equations of this thesis, the suffix *pu* will be dropped.

Chapter 6. CUDA Kernel Development

The previous chapters of this thesis addressed the mathematical models of the electrical devices that will be studied. The simulation of both PVPPs and wind farms requires many computational resources and, as explained in Chapter 2, heterogeneous architectures with specialized hardware can dramatically reduce the running time of this simulation. This chapter will be devoted to the implementation of these models in graphics cards. The parallel code was programmed using the CUDA [12] platform. CUDA exposes parallel execution primitives to organize the work. Threads correspond to each parallel execution of the kernel code. These threads are organized into blocks, which can share data efficiently. Finally, many blocks can be launched at the same time, and that group of blocks in parallel is known as the grid. More information on the CUDA execution model can be found in Section 2.1.

Owing to the difference in requirements of the simulation of wind farms and PVPPs, their parallelization strategies, and therefore the kernels, will be radically different. This chapter has been therefore divided into three sections. Section 6.1 details the parallelization of the wind turbine models, while Section 6.2 expounds how the global wind farm was parallelized. Finally, Section 6.3 details the parallelization strategies followed to reduce the computational cost of PVPPs simulation.

6.1. Parallelization of Wind Turbine Models

This thesis studies the solving of wind farms whose wind turbines are DFIG. Chapter 3 has introduced its mathematical models and the algorithms to be used to solve it. As a remainder, the expression (221) details Heun's method (as explained in Section 3.9) applied to matrices in state-space form. The nonlinear system of the DFIG was linearized around an equilibrium point (more information on Sections 3.8 and 3.10), and the matrices A, B, C, D were obtained.

$$\begin{cases} \Delta x(t+h) = \Delta x(t) + h\dfrac{A\big(\Delta x(t) + x_{temp}\big) + B\big(\Delta u(t) + \Delta u(t+h)\big)}{2} \\ x_{\text{temp}} = \Delta x(t) + h\big(A\Delta x(t) + B\Delta u(t)\big) \\ \Delta y(t) = C\Delta x(t) + D\Delta u(t) \end{cases} \tag{221}$$

That expression calculates the state vector by means of the previous one. At a first glance, it is a serial expression with little room for parallelism, but there are venues to extract parallelism. These venues are the following:

- **Wind turbine level parallelism:** each wind turbine system is processed in parallel. Each state-space system is assigned a thread and Heun's method is executed there. Parallelism stems from the fact that many threads are executed at once. This is the most straightforward parallelism approach, and it is used, in general, when the number of tasks that can be parallelized exceeds the tens of thousands. This method falls short for simulating a wind farm efficiently. As an example, the largest wind farm in the world is comprised of 7000 wind turbines [130]. The graphics card *NVIDIA GeForce RTX 3090*,

which has been used in this work, has 82 SM and 4 schedulers each. As it was explained in Section 2.1.1, each scheduler issues the instructions to be executed by each warp, and a warp contains 32 threads. Therefore, the number of threads that can ideally be executed is $32 \cdot 4 \cdot 82 = 10496$. The graphics card architecture is optimized to quickly swap inactive threads by active threads to execute them and reduce the running time (see Section 2.1.1). Assigning a thread to a wind turbine model is therefore insufficient to extract as much parallelism as possible. For that reason, there must be further venues of parallelism should this parallelism strategy be followed.

- **Step level parallelism:** the use of numerical integration algorithms tailored to parallel architectures has a long history. There were early works aimed to design numerical integration algorithms with steps that could be parallelized. Back in 1967, *Minanker et al.* [131] proposed Runge-Kutta algorithms that could be parallelized. It was previously shown in this thesis, in Section 3.9, that the Runge-Kutta family of algorithms involve calculating temporary values. These temporary values have data dependencies among them. Therefore, if one value is calculated, then the others must be worked out in advance. The design of algorithms where some a_{ij} coefficients are zero (see Equation (135)), can decouple the calculation of these coefficients, thus resulting in parallelism. Careful consideration must be given to these new Runge-Kutta variants, for they might not be stable for certain problems. Another alternative to create a parallel integration method is shown in [132], where the classic 4th level Runge-Kutta algorithm is modified so that intermediary values are calculated not using previous values, but with a process that uses previous approximations of these intermediary values. The calculation of these approximated values is performed in parallel. As a downside of this approach, it is noted in [132] that this parallel algorithm is less precise than its serial counterpart. The other great downside of these approaches is that the maximum parallelism available is limited by the number of steps of the original method. For instance, the classic Runge-Kutta method calculates 4 temporary values. If the computational cost of calculating these values is similar and assuming perfect parallelism, it means that the lowest running time is at best, 4 times lower. Their downsides, coupled with issues such as the added instability have ruled out this method.
- **Problem level parallelism:** algorithms such as waveform relaxation [133], separate the resolution of semi-explicit Differential Algebraic Equation (DAE) systems. The general expression of semi-explicit DAEs is shown in Equation (222). Both $x(t)$ and $y(t)$ are vectors with m and n in length respectively that contain the dependent variables of the system[15]. The former subexpressions expand to a subsystem of *m* equations, and the latter to one of *n* equations. The waveform relaxation method iterates over all variables of the $x(t)$ vector and updates each one of them individually, numerically integrating the system considering the rest of variables as constant. This step is repeated for the $y(t)$ variables. These individual iterations are performed in parallel. After iterating over all variables of the $x(t)$, the process is repeated until convergence is achieved. This method shines when the DAE features many separate states, while its downsides are the

[15] If the system has no dependent variables with the differential state ($n = 0$), the DAE becomes an ODE, which is the type of system that DFIG wind turbines are defined as.

convergence issues, especially when stiff systems with a large simulation time are considered. The wind turbine models do not have more than 11 states; therefore, it has been discarded.

$$\begin{cases} \dfrac{dx(t)}{dt} = F(x(t), y(t), t) \\ 0 = G(x(t), y(t), t) \end{cases} \tag{222}$$

- **Time domain level parallelism:** algorithms such as Parareal [134], PITA [135] and PFASST [136] divide the simulation time into chunks that are simulated in parallel. After simulating each chunk in parallel their starting points are corrected after all chunks are simulated. These methods end when convergence has been achieved. In general, these methods work by using two propagators, a coarse and a fine one. The term propagators refer to the methods that propagate the solution in time. In this thesis, Heun's method is used to solve the ODE forwards, so it "propagates" the solution forwards. Using a larger timestep (the coarse propagator), it can be obtained approximate solutions of the states at the start of each chunk, while the fine propagator, with a smaller timestep, integrates the chunks. These approximate solutions and the final value of the state of the previous chunks are used to further refine the complete system. These methods have convergence issues, and their main differences derive in the way they ameliorate the lack of convergence by approximating the ODEs.
- **Combination of the previous techniques:** the previous techniques can be combined if needed. As an example, the individual iterations of the waveform relaxation might be further parallelized using a time domain level parallelization scheme.

The parallelism strategy of choice is wind turbine parallelism. The previous algorithm strategies suffer issues of convergence, and they are convenient only when there is a vast number of states. Furthermore, the fact that these algorithms take advantage of convergence means that the solution would be inexact unless many iterations are performed[16].

6.1.1. Thread Level Parallelization of Wind Turbine Models

One the parallelization method was chosen, it is necessary to distribute the graphics card resources, starting by assigning the work each thread must perform. As it can be observed in the expression (221), most of the mathematical operations consist of multiplying a matrix times a vector. Therefore, parallelization will be based around this task. It is suggested the following parallelization schemes [45]:

- **One thread per matrix:** this strategy performs the matrix-vector multiplication without parallelism. A loop iterates over all rows of the matrix and performs a scalar product

[16] Decimal numbers are stored as IEEE-754 on both the host and the device, and therefore their precision is not infinite [137]. It might be possible that an iterative process reaches the maximum precision available for IEEE-754 encoded values.

between each row and vector. This algorithm exposes no parallelism to the GPU but relies on the multiplication of many matrices at once. As it was mentioned at the beginning of Section 6.1, this parallelization scheme is not enough to take advantage of the full resources of the graphics card. Therefore, it has been discarded.

- **One thread per matrix element:** each thread multiplies a matrix element by the corresponding vector element. Then, a parallel vector reduction is performed [138]. For a vector of size n, it is launched $\frac{n}{2}$ threads. Each thread adds the value of its corresponding component plus the value of the other half of the vector as shown in Figure 50. It is required to synchronize between various stages. If the target vector is larger than the thread count of the CUDA blocks, the synchronization must be performed with either cooperative groups or with independent kernel launches.

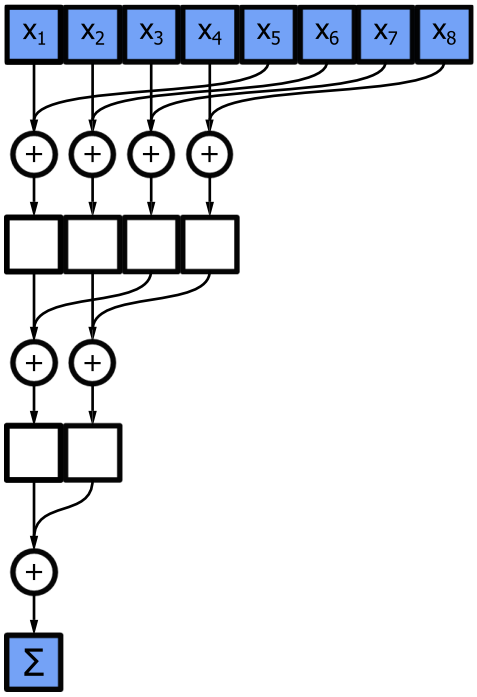


Figure 50. Parallel vector reduction.

- **One thread per matrix row:** each thread performs a scalar product of each matrix row and the vector in parallel.
- **Multiple threads per matrix row:** each thread processes a part of the row in series. Then, a parallel reduction is performed to work out the final result as shown in Figure 50.
- **One thread per multiple matrix rows:** each thread is assigned multiple rows and processes them in series.
- **Multiple threads per multiple matrix rows:** this mode is a combination of the previous ones, where a thread processes multiple threads but only a part of each row.

The optimal matrix multiplication strategy for a matrix of m rows and n columns is pictured in Figure 51. As it is expected, the strategy which requires the least amount of synchronization operations is the optimal solution for matrices of low size. The GPU can launch many threads and perform context switches in the SM to hide latencies, yet it has its limits. Launching a kernel carries a small overhead, which becomes more noticeable the larger the block count, hence the need to process more work per thread. It has been highlighted with a red square the matrix sizes

of the linearized wind turbine models. Therefore, the design of the kernel follows the one thread per matrix strategy.

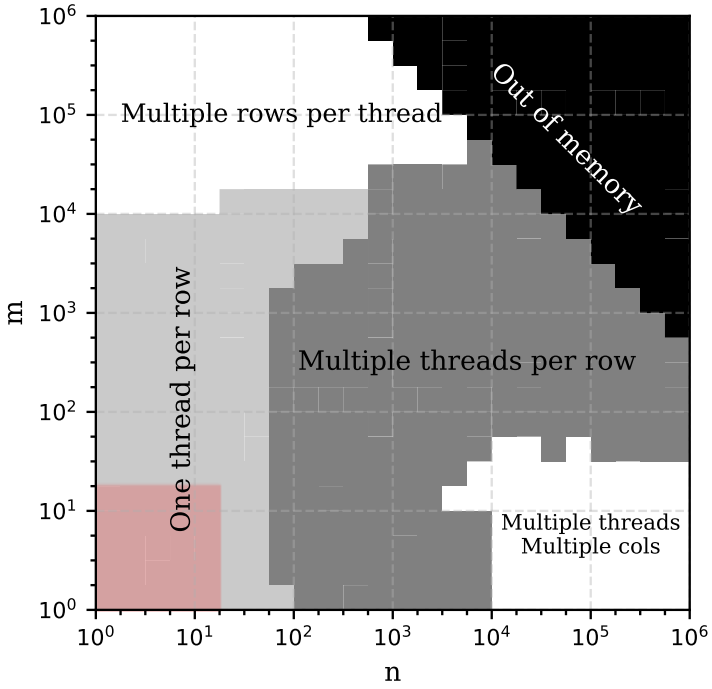


Figure 51. Optimal parallel multiplication strategy (adapted from [45]).

6.1.2. Block Level Parallelization of Wind Turbine Models

Once it has been set which workload each thread processes (processing one matrix-row per thread as shown in the previous section), the block configuration of the kernel ought to be set. The following design choices might have been considered:

- **One block, one thread:** each block has a single thread. The threads must share information among themselves, as it will be later shown. This information sharing can be performed using global memory and atomic functions. It has been ruled out for being inefficient.
- **One block, one ODE:** each block features a thread per state and/or a thread per output variable. The small number of states and outputs make this option inefficient as it was shown in Section 6.1.1.
- **One block, multiple ODEs:** each block will simulate many state-space linearized models. Threads correspond to different wind turbines and do not share information among themselves. This design choice does not have the drawbacks of the former, and hence it was the design of choice. To increase occupancy and reduce thread divergence, 32 wind turbines will be integrated, as explained below.

Once each block has been assigned to integrate multiple models, the threads within them must be distributed. That is, assigning to each thread a row of the matrices A, B, C and D, and a wind turbine. The most straightforward way to organize the threads is shown in Table VI. The state space system features 9 or 11 states, depending on whether the 5th or 3rd order wind turbine model

is being integrated, while the number of outputs is always equal to 5. The total number of threads per linearized model is the maximum of the state and output number of variables. The calculation of the output values is independent of the new state calculation and can be performed in parallel in the same thread thanks to ILP (see Section 2.1.9).

Table VI. Example of wind turbine integration thread organization.

Thread	Wind Turbine	Row
0	0	0
1	0	1
2	0	2
3	0	3
…	…	…
10	0	10
11	1	0
12	1	1
…	…	…

The linearized matrix data was stored in global memory, and then copied into registers. The wind turbines are connected to different areas of the internal grid of the wind farm. The voltage in these areas will be different, and so will be the wind speed. The linearization process will thus produce matrices of different value. It was considered the use of constant memory so as to reduce register usage and increase occupancy (see Section 2.1.1). Constant memory is just 64kB long, and the required memory for all linearized matrices of all wind turbines would overflow the available size. Another downside of constant memory that discouraged its use for this situation is the fact that it is optimized for multiple consecutive threads reading the same value. The data that each thread reads from the matrices is unique to the thread, therefore scattered memory read would be unavoidable, negating the advantages of constant memory.

The part of the CUDA kernel that processes wind turbine models is composed of loops that iterate a vector and row to perform the scalar product. Code 13 shows a simplified loop that performs this job. The vector `X_s` corresponds to the shared state of all wind turbines by the current block. The vector resides in shared memory, and its performance hinges on the lack of memory banks collisions (see Section 2.1.3). Table VII shows the memory accesses to the shared memory vector `X_s`, and it shows that on the first iteration, up to 11 threads read the same memory location, which is suboptimal as all these memory transactions are serialized. Therefore, this thread organization is suboptimal.

```
1 for (int i=0;i<7; i++){
2   Var1+=A_register[i]*X_s[current_wind_turbine*number_of_states+i];
3 }
```

Code 13. Simplified CUDA loop for the first wind turbine thread organization.

Table VIII offers an alternative thread organization, where threads that process the same matrix row but on different wind turbines are laid out consecutively, whereas Code 14 shows the resulting loop. Memory access patterns to `X_s` have varied. The memory access pattern of this new loop is shown in Table IX. This access pattern shows an access without memory bank conflicts. Therefore, threads within a block will be organized in that way for optimal performance.

Table VII.Memory accesses to X_s vector, first try.

Thread	Wind Turbine	Row	Iteration 0	Iteration 1
0	0	0	0	1
1	0	1	0	1
2	0	2	0	1
3	0	3	0	1
…	…	…	…	…
10	0	10	0	1
11	1	0	11	12
12	1	0	11	12
…	…	…	…	…

Table VIII. Proper wind turbine integration thread organization.

Thread	Wind Turbine	Row
0	0	0
1	1	0
2	2	0
3	3	0
…	…	…
31	31	0
32	0	1
33	1	1
…	…	…

```
1 for (int i=0;i<7; i++){
2   Var1+=A_register[i]*X_s[current_wind_turbine +i*WIND_TURBINES_IN_BLOCK];
3 }
```

Code 14. Simplified CUDA loop for the second wind turbine thread organization.

Table IX. Memory accesses to X_s vector, second try.

Thread	Wind Turbine	Row	Iteration 0	Iteration 1
0	0	0	0	32
1	1	0	1	33
2	2	0	2	34
3	3	0	3	35
…	…	…	…	…
31	31	0	31	63
32	0	1	0	32
33	1	1	1	33
…	…	…	…	…

6.1.3. Wind Turbine Integration Kernel Flowchart

The startup section of the code that deals with the integration of the linearized wind farm models is shown in Figure 52. The kernel starts executing some startup code, which sets, depending on the thread and block id, which wind turbine and which row of which matrix shall be processed by it. After that, the kernel copies the row values to registers, calculates the initial value, and reads the initial input values. There are two sources of inputs, the grid voltage solver and the external inputs which were copied to the global memory before the kernel was executed, and these inputs are written into shared memory.

After that, the kernel enters a loop that simulates the wind turbine with Heun's method, as described in Section 6.1. The flowchart of all the operations performed within this iteration is shown in Figure 53. First, the values of the inputs at the next stage are read. The values of the wind speed and the requested reactive power (see Section 3.6) are present in memory and can be read. The network voltages are not, and these values are considered equal between both steps of Heun's method. The input values of the current iteration are read from the shared memory that holds the values of the next iteration in order to avoid excessive memory operations. Each kernel thread processes an input element. It was previously mentioned in Section 6.1.2 that the number of threads would be equal to the number of wind turbines per block plus the number of states. For every model considered, the number of states is larger than the number of inputs, and no input remains unread. Thanks to the thread distribution presented in Section 6.1.2, all idle threads belong to the same warp, thus avoiding divergence and letting the SM avoid the dispatching of those threads while they are waiting on the synchronization barrier.

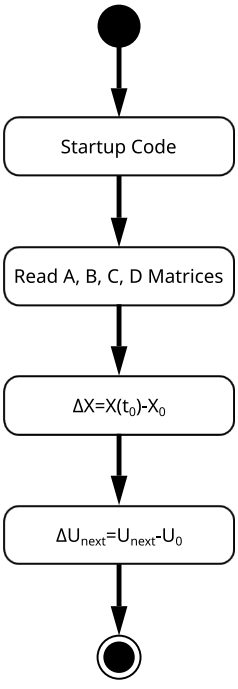


Figure 52. Startup code of wind turbine linearized model integration in CUDA.

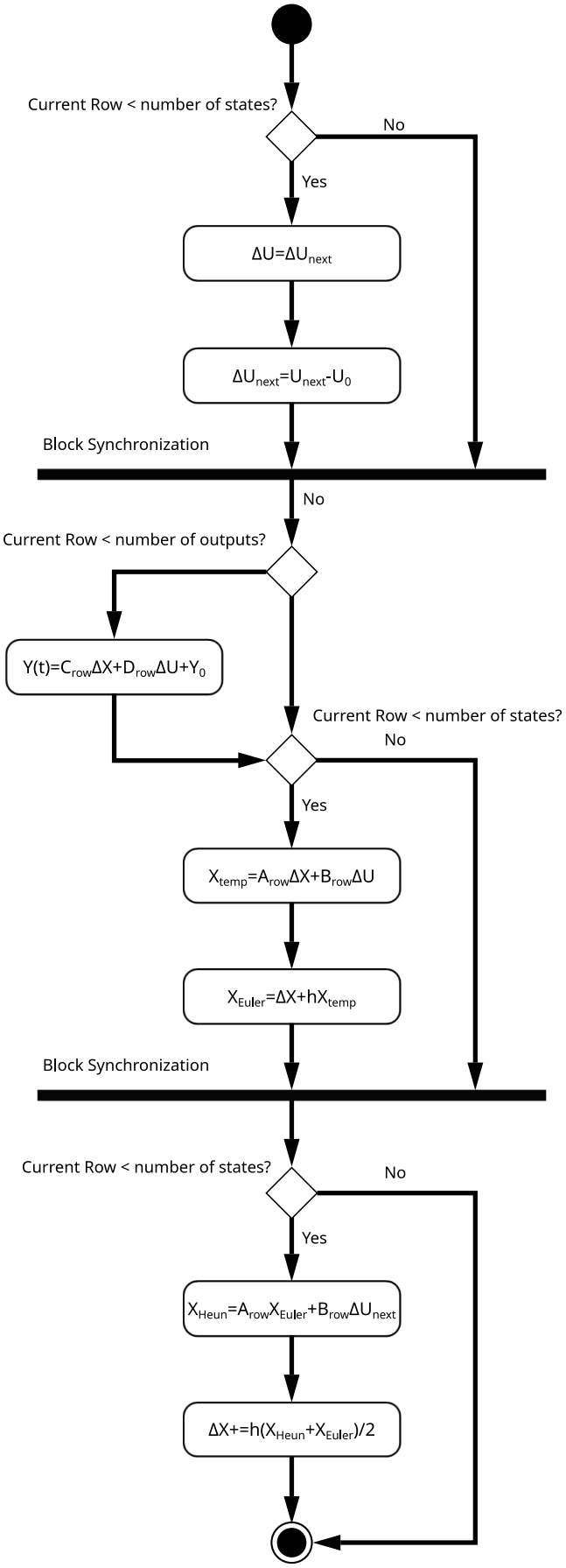


Figure 53. Loop stage of the wind turbine linearized model integration in CUDA.

Heun's method requires a block-level synchronization owing to the need to read X_{Euler} from other threads. Before the first synchronization, each thread performs four scalar products: $C_{row} \cdot \Delta X$, $D_{row} \cdot \Delta U$, $A_{row} \cdot \Delta X$, and $D_{row} \cdot \Delta U$. These scalar products were programmed taking advantage of loop unrolling and interleaved operations, as it will be explained in the next section (see Section 6.1.4). The second stage of the Heun integrator solves the remaining operations, i.e., calculating the derivative value if the states evolved using Euler's method, and finally advancing the states using the average of the derivative at the current timestep and at the next timestep. In this stage, and in the previous one, it is also required to add vectors and multiply them by the timestep value constant, named as h in the diagram. The threads that process each row simply perform a multiplication and addition with the corresponding values. No further parallelism strategies are needed.

After this final operation, the kernel performs another synchronization so that all threads are aware of the new states. However, the network grid solver requires on itself a grid-wide synchronization primitive, which supersedes the need of block-level synchronization.

6.1.4. Loop Unrolling Applied to State Space Integration

The previous section mentioned how loop unrolling and interleaved access were indispensable to kernel execution performance. Loop unrolling is the process that converts a loop statement into a set of instruction with no counter and no condition checking. Loop unrolling is a widely known optimization that compiler such as GCC[17] apply if applicable. Loop unrolling can be either partial, where the number of iterations is reduced by cloning the instructions in the function body a certain number of times and then reducing the number of iterations by that amount, or total, where the loop no longer exists and is substituted by the instructions of the loop body times the number of iterations. The CUDA platform provides the `#pragma unroll` directive to hint the compiler that the loop should be unrolled.

This optimization can only be applied if the number of iterations is known at compile time. To make the kernels generic, the number of states, inputs and outputs were passed as template arguments. Template arguments are a method to generalize C++ functions and classes which can be programmed with generic variable types, and the compiler creates different versions depending on the template arguments when calling them. Templates also work with integer values. At launching the CUDA kernel, some `if` statement switches between the invocations of the kernel with 9 or 11 components, so the CUDA compiler creates different versions of the kernel. The loop unrolling can be best understood with an example. Code 15 shows a loop that makes part of the kernel. It performs the scalar products of the state and input vector with matrices A and B. The scalar products are added into two different variables to take advantage of instruction level parallelism, as it will be shown below.

[17] GCC, the GNU Compiler Collection. `https://gcc.gnu.org/`

```
1 Var1=0;Var2=0;
2 for (int i=0; i<(tamx > tamu ? tamx :tamu); i++){
3         if (i<tamx)
4               Var1+=A_register[i]*X_s[current_wind_turbine +i*WIND_TURBINES_IN_BLOCK];
5         if (i<tamu)
6               Var2+=B_register[i]*U_s[current_wind_turbine +i*WIND_TURBINES_IN_BLOCK];
7 }
```

Code 15. Wind turbine processing loop, phase 1.

This loop does not seem to implement a constant-value loop. However, after substituting the template arguments (see Code 16), the upper limit can be worked out, and the loop can be unrolled. The directive `#pragma unroll` does not force the compiler to unroll the loop, but merely suggests it. The number of iterations is low, and the compiler effectively unrolls it, and the compiler internally converts it to Code 17.

```
1 Var1=0;Var2=0;
2 for (int i=0; i<(11 > 4 ? 11 :4);i++){
3         if (i<11)
4               Var1+=A_register[i]*X_s[current_wind_turbine +i*32];
5         if (i<4)
6               Var2+=B_register[i]*U_s[current_wind_turbine +i*32];
7 }
```

Code 16. Wind turbine processing loop, phase 2.

```
1  Var1=0;Var2=0;
2  if (0<11)
3        Var1+=A_register[0]*X_s[current_wind_turbine +0*32];
4  if (0<4)
5        Var2+=B_register[0]*U_s[current_wind_turbine +0*32];
6  if (1<11)
7        Var1+=A_register[1]*X_s[current_wind_turbine +1*32];
8  if (1<4)
9        Var2+=B_register[1]*U_s[current_wind_turbine +1*32];
10 if (2<11)
11       Var1+=A_register[2]*X_s[current_wind_turbine +2*32];
12 if (2<4)
13       Var2+=B_register[2]*U_s[current_wind_turbine +2*32];
14 if (3<11)
15       Var1+=A_register[3]*X_s[current_wind_turbine +3*32];
16 if (3<4)
17       Var2+=B_register[3]*U_s[current_wind_turbine +3*32];
…
```

Code 17. Unrolled wind turbine processing loop, phase 3.

Finally, the conditions of the `if` statements are known at compile time and can be simplified away. To check that the loop was unrolled, the kernel was decompiled. PTX disassembled code showed that there was indeed no loop, but the instructions were still not reordered to take advantage of the GPU code, as both memory accesses and floating-point instructions were interleaved. The SASS disassembly of the code, which is shown in Code 18, has moved all load value functions to the beginning. The `LDS` function loads a shared value in the direction in brackets to a register. Next, a `FFMA` operation is performed, i.e., calculates $d = a \cdot b + c$. The

result of the operation overwrites the loaded value from shared memory to reduce register count. It has been colored the registers which hold state and inputs, while it has been highlighted with italics the registers which do not change. These registers hold the matrices row values. A final register, `RZ`, is an special register that always reads zero [22][18]. ILP is maintained as shown in Code 18, as the compiler has avoided collocating SASS instructions which are dependent. The output register (the left one for both the `LDS` and `FFMA` instruction) does not appear for most instructions as input registers in the previous ones.

```
1  LDS R25, [R23];
2  LDS R26, [R23+0x80];
3  LDS R29, [R23+0x100];
4  LDS R27, [R23+0x380];
5  LDS R32, [R23+0x180];
6  LDS R28, [R23+0x400];
7  LDS R31, [R23+0x200];
8  LDS R30, [R23+0x480];
9  LDS R33, [R23+0x280];
10 LDS R4 , [R23+0x500];
11 LDS R5 , [R23+0x300];
12 FFMA R6 , R0 , R25, RZ ;
13 FFMA R7 , R13, R26, R6 ;
14 FFMA R8 , R12, R29, R7 ;
15 FFMA R6 , R2 , R27, RZ ;
16 FFMA R9 , R11, R32, R8 ;
17 FFMA R7 , R15, R28, R6 ;
18 FFMA R6 , R10, R31, R9 ;
19 FFMA R30, R16, R30, R7 ;
20 FFMA R33, R3 , R33, R6 ;
21 FFMA R4 , R17, R4 , R30;
22 FFMA R5 , R14, R5 , R33;
```

Code 18. Abridged SASS disassembly of unrolled wind turbine processing loop.

The kernel introduced in the thesis used variables `Var1` and `Var2` as accumulators. These accumulators are the halves of the derivative at an instant `t` for this loop. Floating point IEEE-754 arithmetic is not commutative [137] and the final results of a serial implementation and the proposed parallel implementation will be different. The associativity of the `+` operator in C++ is left-to-right, and a serial operation would perform all operations in series, as shown in the expression (223). The `-ffast-math` flag from the GCC compiler allows the reordering of IEEE-754 operations. These reorderings are not IEEE-754 compliant because the final value slightly changes. This flag was activated for the compilation of the C++ implementation for faster results.

$$\frac{d\Delta x}{dt} = \begin{bmatrix} ((((((a_{11}x_1 + a_{12}x_2) + \cdots) + a_{1m}x_m) + b_{11}u_1 + \cdots) + b_{1n}u_n) \\ ((((((a_{21}x_1 + a_{22}x_2) + \cdots) + a_{2m}x_m) + b_{21}u_1 + \cdots) + b_{2n}u_n) \\ \vdots \\ ((((((a_{m1}x_1 + a_{m2}x_2) + \cdots) + a_{mm}x_m) + b_{m1}u_1 + \cdots) + b_{mn}u_n) \end{bmatrix} \tag{223}$$

[18] The equivalent PTX instruction features a register whose behavior is similar. It has been assumed that the SASS `RZ` register has the same function.

On the other hand, the CUDA version does this operation using the order shown in the expression (224). These differences of the order of summations provoke small variations of the final result at the edge of the IEEE-754 precision limits.

$$\frac{d\Delta x}{dt} = \begin{bmatrix} \Big(\big((a_{11}x_1 + a_{12}x_2) + \cdots\big) + a_{1m}x_m\Big) + \big((b_{11}u_1 + \cdots) + b_{1n}u_n\big) \\ \Big(\big((a_{21}x_1 + a_{22}x_2) + \cdots\big) + a_{2m}x_m\Big) + \big((b_{21}u_1 + \cdots) + b_{2n}u_n\big) \\ \vdots \\ \Big(\big((a_{m1}x_1 + a_{m2}x_2) + \cdots\big) + a_{mm}x_m\Big) + \big((b_{m1}u_1 + \cdots) + b_{mn}u_n\big) \end{bmatrix} \tag{224}$$

6.2. Parallelization of Wind Farms with Unfavorable Grids

The previous sections of this chapter detailed the strategies to integrate individual linearized models in state-space representation. These models do not operate in the vacuum, but they interact through the internal grid of the wind farm with each other. Solving the grid across time requires transient studies, such as phasor simulation, dynamic phasor simulation or EMT. The case study presented in this thesis evaluates the proposed CUDA kernel against voltage dips (see Section 7.1). These voltages dips are sudden variations of the input voltage, which cause large transients. Phasor simulation (see Section 2.2.6) does not consider transients, and dynamic phasor simulation (see Section 2.2.7) is better suited for repetitive phenomena, such as voltage switching. Therefore, an EMT simulation (see Section 2.2.5) must be performed. The fundamental equation to be solved is the linear system already presented in Chapter 5, whose expression is reprinted here, in Equation (225), where $[Y]$ represents the admittances of the system, V its node voltages and I the injected current in each node. The magnitudes of the admittance, voltage and current are complex numbers for the wind farms and the AC part of the PVPP, and real numbers for the DC part of the PVPP.

$$[Y] \cdot V = I \tag{225}$$

It was concluded in Chapter 5 that the appropriate power line model is the π-section line. This model represents the power lines as lumped resistances, inductances, and capacitances, which were later discretized and turned into admittances with associated current sources whose values depend on the state of the capacitors and inductors. In the EMT simulation, the admittances make part of a linear system matrix, while the current sources are part of the RHS vector of the linear system. The π-section line model themselves were transformed using the *dq0* transformation to reduce numerical oscillations, and to reach a stationary state where the signals are continuous.

After all these considerations, the sparse matrix and the RHS vector is built. It was shown in Chapter 2 that parallelization of sparse matrix relied on column level parallelism. This parallelism stems from the general expressions that define the updates of the LU matrix. The expressions of the LU and LDL decomposition, previously presented in Chapter 2 are shown here as a reminder in Equations (226) and (227). The goal of these decompositions is to decompose matrix A into a lower triangular L, a diagonal D, and an upper triangular U matrices. The LU decomposition can be applied to symmetrical matrices, while the LU decomposition cannot. The matrix that defines the system is symmetric, while the matrix of the wind farm is not. It will be used the LU decomposition in the serial linear solver of the wind farm due to its performance speedup and the fact that it does not require the computation of square roots.

$$A = L \cdot D \cdot L^T \tag{226}$$

$$A = L \cdot U \tag{227}$$

The LU decomposition algorithm decomposes a square non-symmetrical matrix A of size n by iterating over all columns and, for each column i, applying Equation (228) and then Equation (229). Equation (228) updates the values of column i, dividing them by the term in the diagonal. This operation does not change the sparsity of the column. However, the update operation shown at Equation (229) modifies the values of other columns as shown in Figure 54. This figure presents the update of a value which was previously zero, while the column highlighted in yellow was being processed. The new value will be equal to the negative of the multiplied numbers shown with the red arrows.

$$A_{i+1:n,i} = \frac{A_{i+1:n,i}}{A_{i,i}} \tag{228}$$

$$A_{i+1:n,i+1:n} = A_{i+1:n,i+1:n} - A_{i+1:n,i}\, A^T_{i,i+1:n} \tag{229}$$

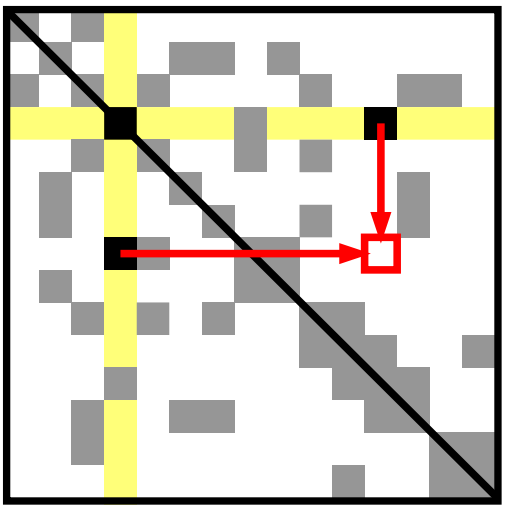


Figure 54. LU decomposition value update.

This update operation forces the serialization of the LU decomposition algorithm. Nevertheless, if the matrix structure is analyzed beforehand, it can be shown that not all columns depend on all the previous columns. Thus, a dependency tree can be built. This dependency tree will be heavily dependent on the layout of the interconnections. Most wind farms internal layout is radial due to low economic cost [139]. Radial layout connects wind farm turbines in series as strings, and these strings are connected directly to the PCC. This layout results in sparse matrix with extremely low parallelism, as Figure 55 shows. It shows the sparsity pattern of a small wind farm, with 20 wind turbines, laid out in 5 strings. The main diagonal corresponds to the admittances of the power lines that connect adjacent wind turbines and also corresponds to the admittances of the step-up transformer that links the wind turbine and the main power lines.

The matrix sparsity (proportion of non-zero values to the total number of values) is in this case equal to 92.97%, although heavily column dependent. As shown, the quasi-band pattern of this matrix entails that processing a column affects the next one. But, as shown in Figure 55, there are at most 3 or 4 values in each column, so there is little parallelism to be exploited. The dependency tree is thus long and thin, making parallelism through traditional methods impossible.

Furthermore, the performance of the LU transformation depends on the amount of fill-in produced. The fill-ins are the non-zero values which are added by the factorization algorithm to compute the matrix. Non-zero values cannot be omitted and, thus, the computing cost increases. Figure 54 shows how not only the LU decomposition updated the values of the matrix but can create a value where there was a zero. Fill-ins are overly sensitive to row permutation, and for this reason a permutation matrix is usually added to the LU expression (227). For radial networks it was found that the optimal node ordering was the one shown in Figure 56 and it was demonstrated by iterating over all possible reorderings of small matrices and extrapolating the results. Figure 57 shows, for the matrices that represent small wind farms of 1, 2, 3 and 4 wind turbines, the amount of available reorderings that produce a certain sparsity. For every possible reordering of a matrix of n rows, there is $n!$ permutations and, as shown, node reordering really improves the sparsity, while the vast majority of reorderings offer subpar performance.

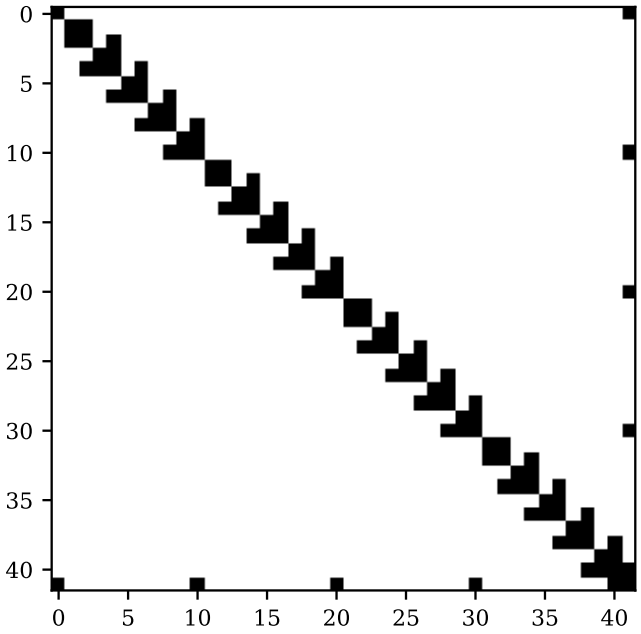


Figure 55. 20-Wind turbine wind farm sparse matrix.

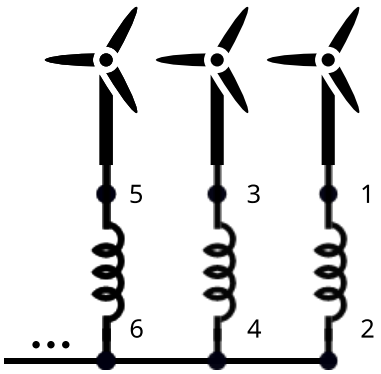


Figure 56. Optimal node ordering for radial layouts.

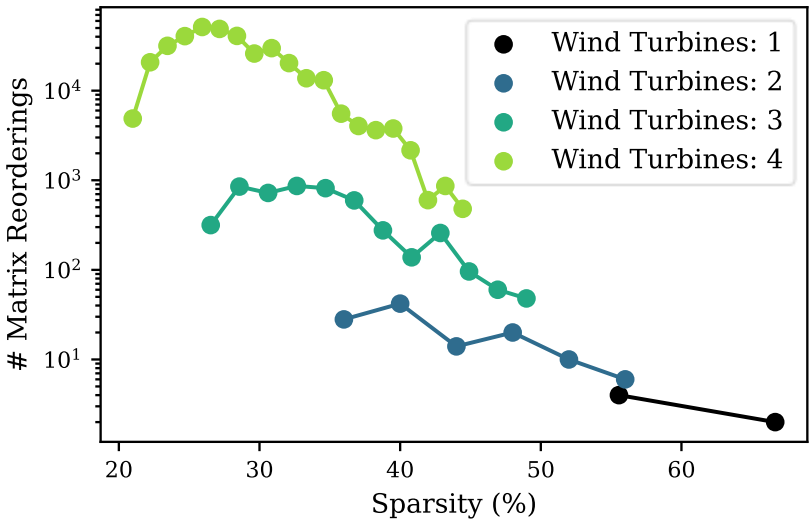


Figure 57. Matrix orderings and sparsity.

The classic LU decomposition offers little parallelism, and the matrix inverse of this sparse matrix turns the sparse matrix into a dense one. Figure 58 plots the number of non-zero items in the LU matrix decomposition against the matrix inverse. The matrix inverse is fully dense, and its size grows quadratically, while the LU matrix size grows linearly.

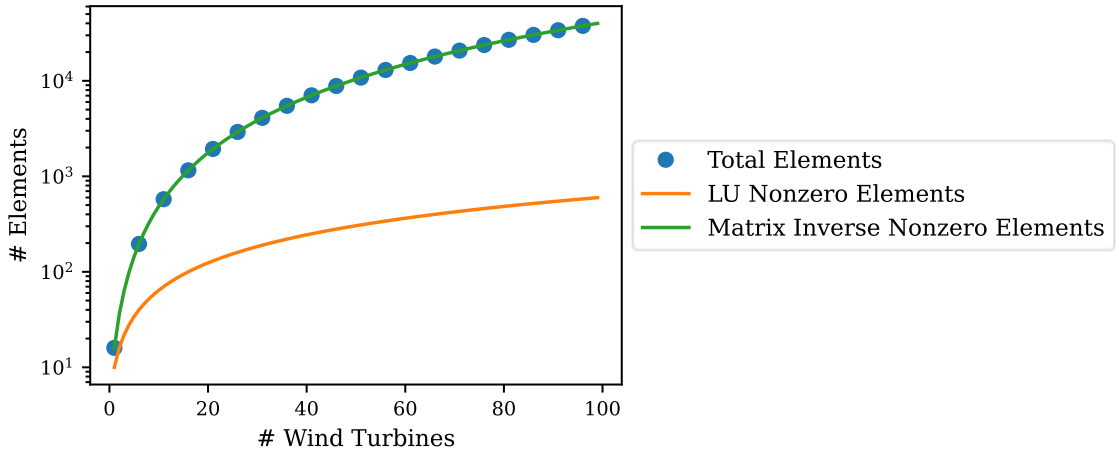


Figure 58. Number of nonzero elements vs matrix inverse nonzero elements.

One of the features of the system implemented in this thesis is that its admittance matrix does not change, provided the timestep is kept constant, as can be inferred from the expressions of the discretized passive components from Section 5.5. Only the additional current sources depend on the previous values of the voltage and current across the power line while the admittance value does not change across iterations. Therefore, the admittance matrix can be preprocessed at the beginning, while solving the system by only performing substitutions in each iteration. All these characteristics play a role when designing the GPU implementation.

To exploit the parallelism, the grid has been torn apart to obtain a system like the matrix system shown in Equation (230). Tearing apart a grid consists in disconnecting two nodes and connecting a current source of unknown value to both. This unknown value will then be a new unknown of the linear system. Tearing the system apart creates many isolated subcircuits plus a new system that correlates all new current sources. In the linear system shown in Equation (230),

all the nodes that are part of the first subcircuit are grouped into the vector X_1, and those of the 2nd subcircuit correspond to X_2. $A_1, A_2, \ldots$ are the admittances of these subcircuits. On the other hand, $D_1, D_2, \ldots$ codify how the inserted current sources are connected. If the current source j is connected to the node i, then $D_{i,j} = 1$ if the virtual current source is pointing out of the circuit, -1 if it is pointing into circuit. X_u represents the new unknowns inserted by tearing the system, which are the currents of these new current sources. This system is similar to the node tearing approach (see Section 2.4.3). More on how this system is solved in parallel is included in the next section.

$$\begin{bmatrix} A_1 & 0 & \cdots & D_1 \\ 0 & A_2 & \cdots & D_2 \\ \vdots & \vdots & \ddots & \vdots \\ D_1^T & D_2^T & \cdots & 0 \end{bmatrix} \begin{bmatrix} X_1 \\ X_2 \\ \vdots \\ X_u \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ 0 \end{bmatrix} \tag{230}$$

The general algorithm to solve the wind farm is roughly a loop where the following steps must be taken for each timestep:

1. Get the state of every connected wind turbine using as an input parameter the grid voltage where these devices are connected.
2. Calculate the injected current for each bus taking into account the current sources shown in Equation (215), Equation (216) and Figure 49.
3. Solve the sparse linear system shown in Equation (220) and get the bus voltages.
4. Calculate the next value of the current sources referred to in step 2, which are related to the internal state of the power lines.
5. Advance the simulation time using the integration timestep.

6.2.1. Obtaining the State of Every Connected Turbine

Given an initial voltage at the wind turbine terminals, their models are integrated for a certain timestep, and the current output is calculated. This consists in solving the linear system presented in Equation (221). A parallel matrix vector operation is performed, and one thread is assigned to every row in the matrix, and each thread calculates a scalar product between the rows of the *A*, *B*, *C*, *D* matrices and ΔX and ΔU. To perform the vector addition in Equation (221), one thread is assigned per component. As noted in Section 6.1, Heun's method is used. This method requires the calculation of a first step using Euler's method, and then the calculation of the derivative if the states are those of Euler's. This requires a block synchronization primitive as shown in Section 6.1.3.

6.2.2. Calculating the Injected Current for Each Bus

The injected current for each bus (i.e., b_i) can be calculated in parallel. It is necessary to add the injected current of each current source, which includes the wind turbines and the current sources in Figure 49. Then, a subtraction is performed on the first matrix column times the slack voltage, as shown in Equation (220), and the contributions of all the power lines connected to it are added. The RHS of the linear system can be calculated as a sum of terms as shown in Equation (231). The value of the RHS b_i is the sum of the variable terms $b_{i,j}$ multiplied by a certain constant $c_{i,j}$. The origin of $b_{i,j}$ is varied. They can either be the current injected by the

generator (i.e., the wind turbine) connected to that node, or it can be the voltage of the slack bus. The current sources that arise from the discretized power line models also take part in the summation. All these contributions can be regarded as a multiplication of a value which is constant throughout the simulation and a variable, as shown in Table X. One thread has been assigned per component to perform the initial multiplication. All threads working on b_i belong to the same block.

$$b_i = \sum_j c_{i,j} \cdot b_{i,j} \tag{231}$$

Table X. Constant and variables for the RHS of the matrix.

	Constant	**Variable**
Generator Current	1 or -1 depending on whether in or out of node	$I_{generator}$
Subtraction of First Row (220)	-1	V_1
Admittance of Resistor and Inductor (215), (216)	$\frac{L_{pu}}{\omega_b h(jL_{pu} + R_{pu}) + L_{pu}}$	$I_{dq_{n-1}}$
Admittance of Capacitor (215), (216)	$\frac{-C_{pu}}{\omega_b h}$	$V_{dq_{n-1}}$

This sum is performed in parallel using a vector reduction with warp intrinsics. There are many small vector reductions, and the size of each individual reduction is also different, and particular care must be taken to perform such operations in parallel. Considering the hard limit imposed by cooperative groups, fewer threads must be used, even at the cost of thread divergence, which in normal circumstances should be avoided to obtain high-performance kernels [12]. Figure 59 shows how a reduction of various vectors is performed. As an example, three vectors of sizes 3, 2 and 2 are reduced. For the reduction to work, the sizes of the vectors to be reduced are rounded up to the next power of two, and the spaces are padded with zeros. Vectors must also start at a position which is a multiple of their rounded-up size. To maximize thread usage and reduce padding, the vectors have been ordered from longer to shorter. These vector reductions have been performed first, in shared memory, adding synchronization barriers between each stage and then, for the last 32 items, warp shuffle intrinsics have been used [12]. Bank conflicts, which is an issue in CUDA that results in slow shared memory accesses, are avoided. Furthermore, warp shuffle intrinsics show better performance than shared memory when applicable [140].

6.2.3. Parallel Grid Resolution

The solution of the system of Equation (230) can be computed first by obtaining X_u, whose solution is Equation (232), whereas the rest of the unknowns are calculated using Equation (233). The parts marked as pre-computed can be calculated once and their values reused for each iteration. The bulk of the scientific literature recommends solving linear systems first by decomposing the matrix, and then substituting values back and forth as it was explained in Chapter 2. However, that does not mean that using the matrix inverse to solve a linear system cannot be an acceptable approach in some cases [55]. It will be shown in Section 7.2 that a matrix inverse constrained to small subblocks offers a precision similar to that of the LU decomposition. For that reason, and for its parallelization properties, partitioning with a matrix inverse has been chosen for the kernel.

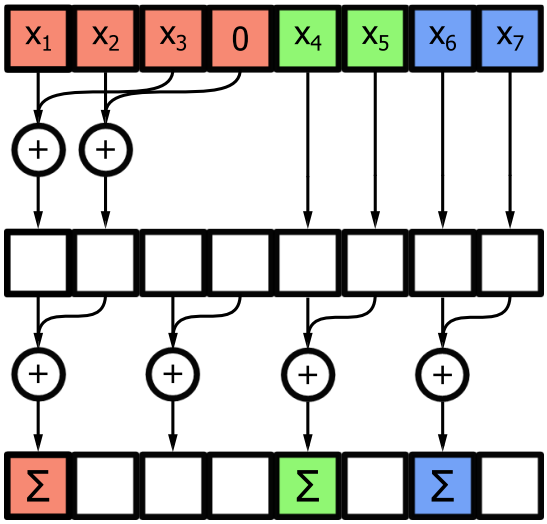


Figure 59. Multiple parallel vector reductions.

$$X_u = \overbrace{\left(\sum_i D_i^T A_I^{-1} D_i\right)^{-1}}^{pre-computed} \sum_i \overbrace{D_i^T A_i^{-1}}^{pre-computed} \cdot b_i \tag{232}$$

$$X_i = A_i^{-1} \cdot b_i - \overbrace{A_i^{-1} D_i}^{pre-computed} \cdot X_u \tag{233}$$

Parallelism is achieved in two venues: parallelizing the summation involving b_i in Equation (232), and parallelizing Equation (233) for each group row of the matrices. Matrix inverses must be performed to precompute these values. Using a strategy similar to the turbine integration step, a thread has been assigned to each row of each matrix $D_i^T A_i^{-1}$. Each thread performs a scalar product between a row of $D_i^T A_i^{-1}$ and its corresponding b_i.

6.2.4. Calculation of the Next Step

Finally, to calculate the new value of each element of the power line, a thread has been assigned to V_{dq_n} and I_{dq_n}, respectively. Each thread calculates the values presented as shown in Equations (215) and (216).

6.2.5. Wind Farm Kernel Flowchart

The way the CUDA kernel works in general is shown in Figure 60. As mentioned above, firstly, the wind turbine LTI systems are solved, then the RHS of the sparse linear system is assembled, as shown in the linear system presented in Equation (231). Since this operation can be expressed as a summation of all individual contributions, a parallel vector reduction is performed to achieve parallelism. To solve the sparse linear system, vector reductions detailed in Equations (232) and (233) are performed in parallel. Kernel threads perform this parallel reduction as previously shown in Figure 59.

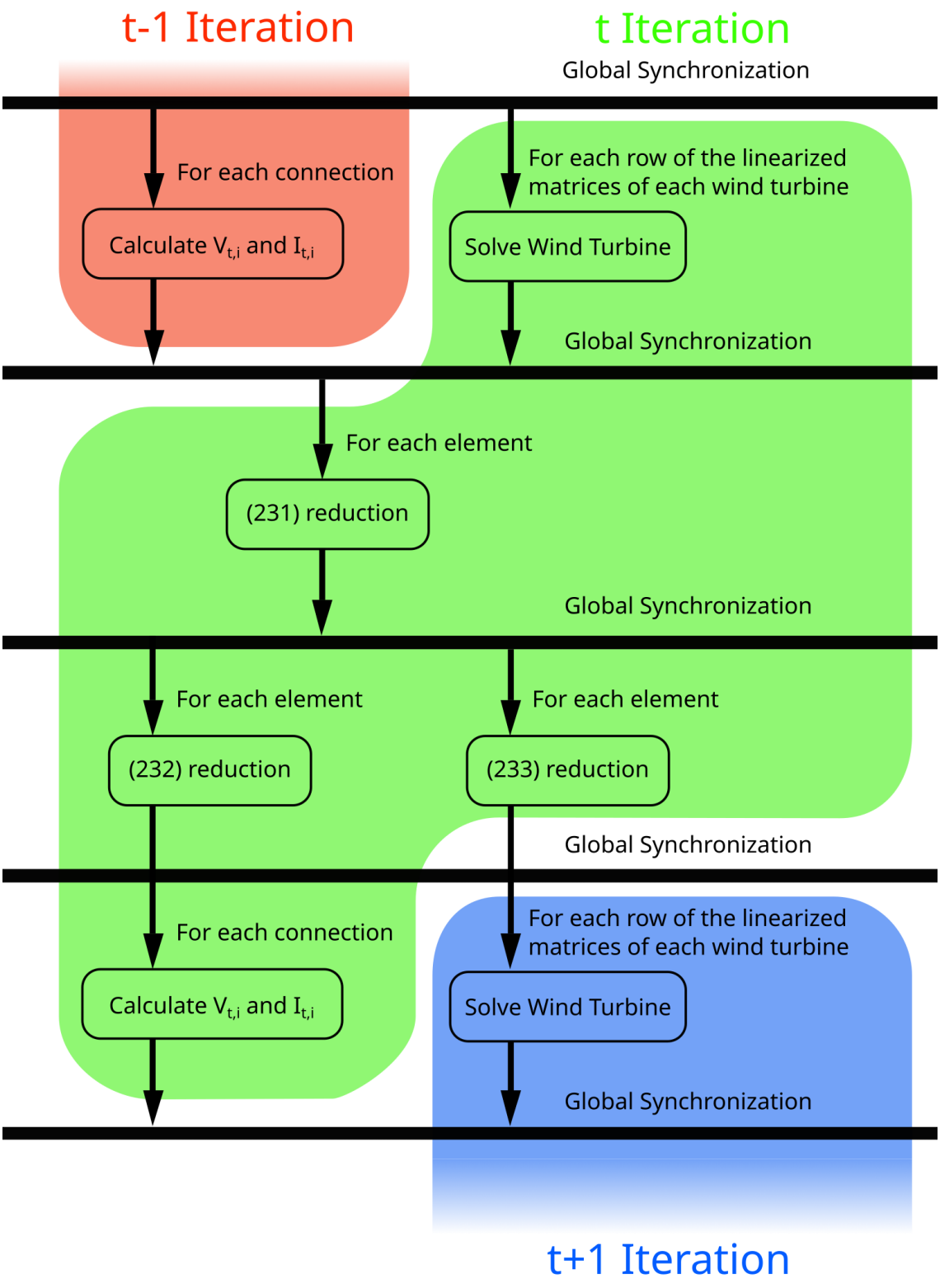


Figure 60. Wind farm implementation iteration flowchart (parallel).

After solving the linear system, both the voltage difference and the current across each line are calculated to update the current contribution of each power line, and the cycle continues until the simulation ends. The horizontal lines of the loop in Figure 60 are global synchronization barriers, and they are executed for the kernel that runs this simulation. After all the threads are

synchronized, the kernel operations continue. All the threads of all the blocks must wait until the rest of the threads have arrived at those barriers. This is necessary so that the PS state is propagated across all the threads. To further speed up the computation, both the integration of the wind turbine and the calculation of the voltage and current across the connections have been executed at the same time, since processing the wind turbine models only needs the node voltages. Each block in Figure 60 has been colored to show which operation each of them belongs to. As indicated in this figure, the current iteration (i.e., iteration at instant t), overlaps with the previous and next iterations. At the beginning of the loop, both $V_{t,i}$ and $I_{t,i}$ are equal to the initial value and, therefore, the top left block of Figure 60 is not executed. In the last loop iteration, the "Solve Wind Turbine" stage, which corresponds to the integration of the wind turbine models, is not executed.

6.2.6. Detailed Wind Farm Kernel Execution

Figure 61 further explains the behavior of the power system, using a simplified iteration flowchart that details the operations in each step. Figure 60 has some overlapping operations, while these operations have been represented as one operation after another in Figure 61 to explain the kernel code more easily.

As shown, the kernel is composed of various stages. In the thesis, wind turbine solving is overlapped with the kernel section of the code that calculates the voltage and currents across each branch of the kernel. The kernel in this figure is split into three distinct parts: solving the wind turbine; the power system solver, which is divided into 2 stages; and finally, the calculation of both the current and voltages across each branch of the system. As shown, the threads of the kernel are grouped into blocks which, in turn, are the elements that make up the grid. Local synchronization in this figure refers to the synchronization of all threads within the block, while global synchronization deals with all threads simultaneously.

The currents injected into the network in the "Solve Wind Turbine" section are calculated given the internal state of the machine, the network voltage of the power system at the connection points and other inputs such as the wind speed. The wind turbine system is expressed in state-space matrix form. To calculate the next step in the iteration, a matrix-vector multiplication must be performed twice. Between each matrix-vector multiplication, sometimes a vector must be added to another as required by Heun's method or multiplied by a constant. Each thread of the kernel performs a scalar product of a matrix row and the vector to compute the matrix-vector multiplication and each thread processes one component when performing the vector additions and the multiplication of a vector with a constant. Each of these operations is implemented internally as a FMA operation. After many of these operations, a local synchronization primitive is inserted to ensure that the next operation reads the correct values. This stage has been already described in Section 6.1.3.

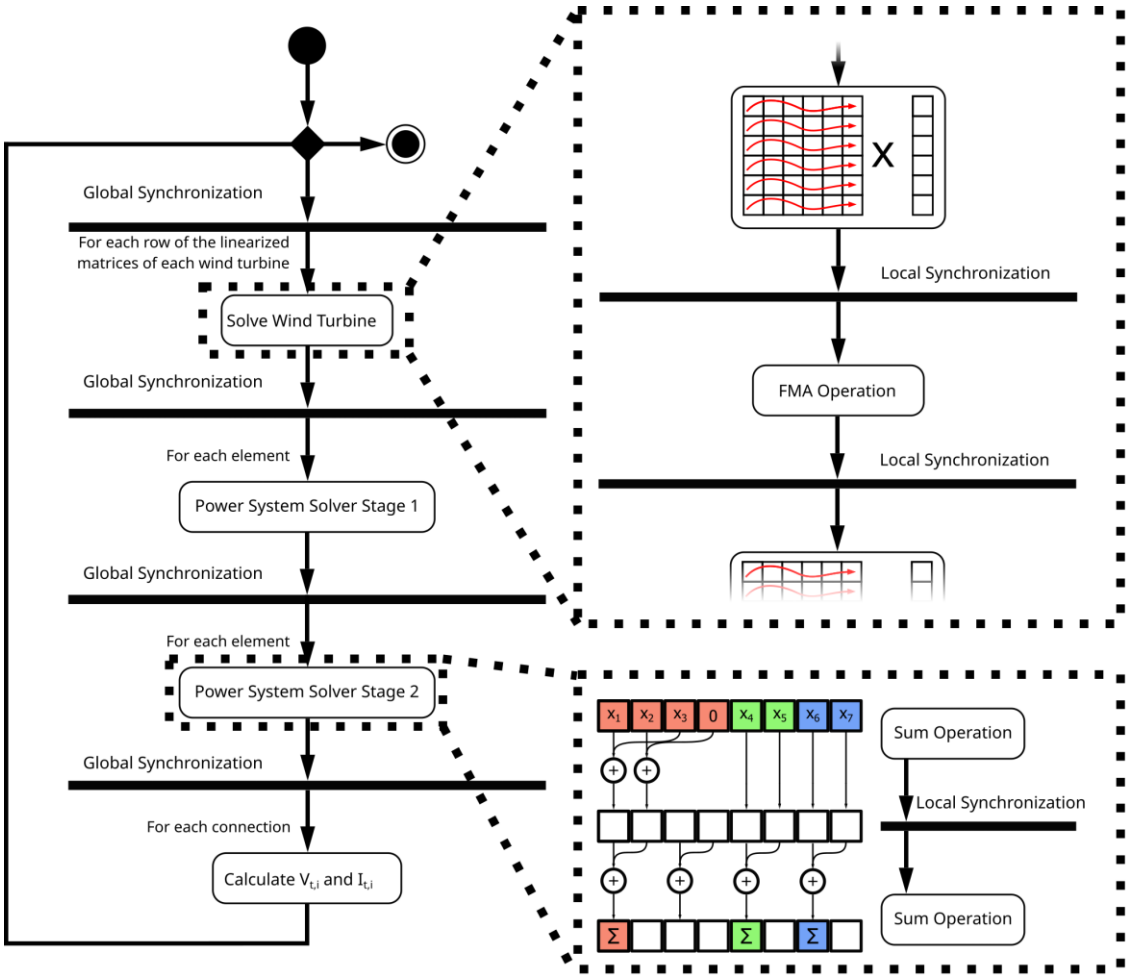


Figure 61. CUDA kernel flowchart (simplified version with no overlapping).

The pseudocode shown in Code 19 outlines how the wind turbine solver part is solved. Every thread checks whether it has assigned a valid wind turbine and vector component. For performance purposes, blocks integrate 32 wind turbines and, therefore, some threads are idle. The regions that deal with the matrix-vector multiplication have been highlighted in blue, and the FMA operations are in red. For the sake of brevity, the second matrix-vector multiplication has been summarized as a function.

The second part of the implementation presented in this section shows how the power system stage is performed. Both stages work similarly, and they have been split in order to insert a global synchronization. They are implemented as a basic vector reduction which, instead of adding all values, performs many smaller vector reductions. This stage is part of the same kernel launches as the wind turbine simulation routine, and therefore each block has 32 threads times the number of states of the wind turbine models. As many blocks as needed are launched to process all the vector reductions. In the simple example previously presented in Figure 59, vectors of sizes 3, 2 and 2 are reduced. The flowchart shows the code as an unwinded loop, while Code 20 shows the code as a loop with an added piece of code that reads the values into shared memory. It must be checked whether the vector size matches the loop iteration. After each iteration, a local thread synchronization is performed, although for the last 32 elements warp-level primitives have been used for increased performance. The beginning of this section, in which data is copied from global memory to shared memory, is highlighted in green, the blue code corresponds to the sum

operation, which successively reduces all vectors, and the red code is the local synchronization primitive.

```
1  if current_thread < number_wind_turbines

2    if component_number < max_number_output:
3      Derivative_Vector=0
4      //for-loop scalar vector multiplication
5      Scalar_Product(
6        Matrix_A[component_number*row_length],
7        State_Vector,
8        Derivative_Vector)

9      Scalar_Product(
10       Matrix_B[component_number*row_length],
11       Input_Vector,
12       Derivative_Vector)

13     block synchronize
14     if component_number < max_number_states:
15       Temp_Next_Step =
16         State_Vector +
17         integration_step * Derivative_Vector

18     Calculate_next_derivative(
19       Temp_Next_Step,
20       Next_input)
21     ...
22     block synchronize
23     State_Vector+=integration_step/2 * (
24       Derivative_Vector +
25       Next_ Derivative_Vector)

26 end if
```

Code 19. Pseudocode of wind turbine simulation.

Finally, for every branch in the power system, the voltage and current across them have been calculated. Each thread computes the voltage and current across one power line.

```
1  if current_thread < number_of_elements
2    if is_element_assigned_to_current_thread():
3      Copy_to_shared_memory(
4        shared_memory[current_thread],
5        global_memory[current_thread])
6      shared_memory[current_thread]*= multiplier
7    else
8      shared_memory[threadIdx.x]=0
9  end if
9  block synchronize
10 for s=BLOCK_SIZE/2 to 1; s/=2:
11   if current_thread < number_of_elements:
12     if elements_vector_processed_by_this_thread < s:
13       shared_memory[current_thread] +=
14         shared_memory [current_thread +s])
15   block synchronize
```

Code 20. Pseudocode of the main part of the power system solver stage.

6.3. Parallelization of Large PVPPs

The previous sections detailed the way the kernel that integrates wind farms works, and the design decisions behind it. It was shown that the sparse linear system resulting from these networks had little parallelism potential. To solve this issue, the grid was partitioned, and each partition was solved as a dense linear system, which opened the window to the performance gains of GPUs. It was mentioned in Section 6.1 the concept of DAEs. The DAE is a model that combines both independent states which vary with time and dependent states that are bound by the set of algebraic equations. The highly detailed model presented for the simulation of PV panels does not depend on its internal state, but only on external inputs, and the inverter and PVPP models do not depend as well, which are regarded as constant voltage and current sources. The behavior of the system changes due to the external magnitudes of irradiance and temperature. For these types of simulations, a power swing caused by a passing cloud can take up to 20 seconds to fully reduce the generated power [123]. However, most electrical transients caused by sudden variations in the system state are cleared in less than a second. Such variations include capacitor energizing, a lightning strike, or an induction generator under a voltage a dip [141]. If the power system features three-phase magnitudes that are balanced and stationary, these can be expressed as two continuous signals using the *dq0* transformation as explained in Section 3.2.1, while DC signals in the PV panels side are continuous. Therefore, the PVPP variables are only modified by gradual changes in the external conditions, where all elements can absorb the changes smoothly, with almost no change to the differentiable states ($\dot{x} \approx 0$), as a quasi-static state simulation. Removing the differentiable states eliminates oscillations within this system, and its value is equal to the average of these oscillations [142]. Hence, (222) is simplified into (234). The system also gains time independence, so all individual timesteps can be simulated in parallel.

$$0 = \mathrm{G}\big(y(t)\big) \tag{234}$$

This is a nonlinear system due to the diode-like currents that make up the PV panel model. These elements can be approximated using a linear equivalent of the nonlinear elements as mentioned in Section 4.2.2. After solving these approximate linear systems, the linearized approximation is updated.

The system can be solved performing an LU decomposition. To solve the system in Equation (225), expressed as $L \cdot U \cdot x = I$, first the linear system $L \cdot y = I$ is solved with a forward substitution scheme, and then the equation $U \cdot x = y$ is solved with a similar approach. Both the LU decomposition, and the backward and forward substitution, can be performed in parallel using a GPU. The way the developed kernel works is shown in Figure 62. There are three different sections that run on the GPU: LU decomposition, and two triangular sparse substitutions.

6.3.1. PVPP Kernel Overview

As mentioned above, the solver integrates a linear system, which then calculates the next step by selecting an appropriate α value. It is proposed in this thesis an adaptive α value for faster convergence that is independent for each variable. After obtaining the current that flows through the nonlinear component thanks to the linear system, for each component a new candidate voltage, called $\tilde{V}_{next}$, is calculated (see Equation (235)). To prevent large fluctuations in value that might lead to non-convergence, each voltage is modified depending on the multiplier α. To

further aid convergence, a small conductance called G_{helper} is added in parallel to each nonlinear element. This helper conductance aids the convergence of the nonlinear elements. After fully solving the nonlinear system, the value of the helper conductance is progressively reduced until it reaches zero, which is like not adding a conductance. In this way, a system with many nonlinear elements can converge to its solution.

$$V_{t+1} = (1-\alpha)V_t + \alpha\tilde{V}_{t+1} \;;\; 0 < \alpha \leq 1 \tag{235}$$

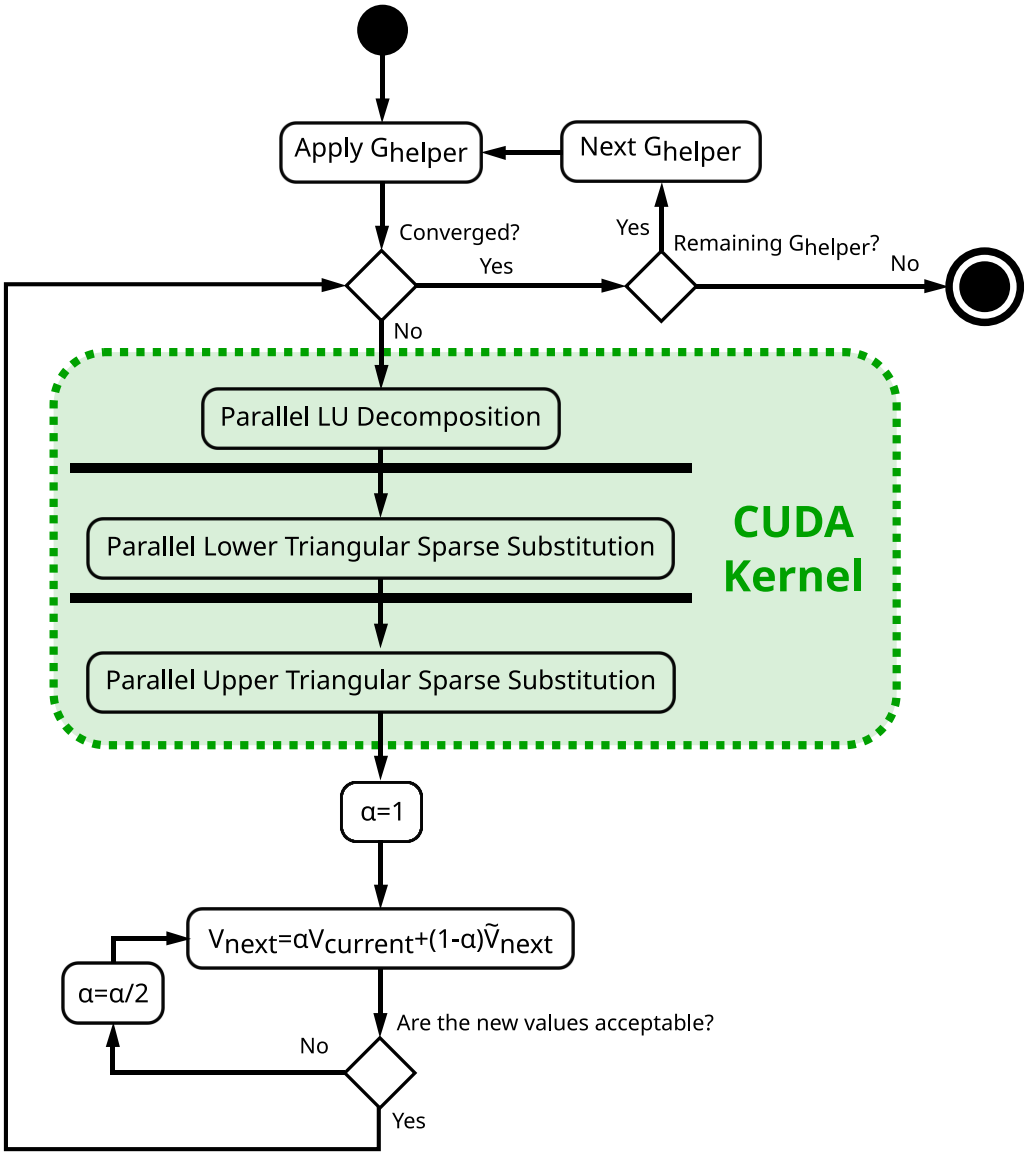


Figure 62. PVPP solver flowchart.

The bulk of the kernel is for the resolution of the linear system, as can be inferred from Figure 62. The LU decomposition of the matrix has been performed on the GPU and then a triangular system resolution is also performed on the GPU as well. To perform an LU decomposition of a square matrix of size n, it is iterated for each column j from 1 to n. First, for every column, the values below the diagonal are updated according to Equation (228). Then, the operation shown in Equation (229) is performed. An outer product is performed between two vectors, and its result is subtracted from the submatrix in the lower right region. In the triangular sparse substitution stage,

the columns are traversed sequentially, and an iterative backward or forward substitution is performed.

After solving the general system, for an iteration V_t, a new voltage $\tilde{V}_{t+1}$ is obtained across the current source. If the system has converged, then $V_t = \tilde{V}_{t+1}$. The voltage is updated using the adjustable constant α as shown in Equation (235), where α is equal to 1 at first, but it can be reduced to aid convergence. After the nonlinear elements have converged, the DC voltage inverter controlled by the MPPT is modified using the Newton-Raphson method as shown in Section 4.4.1. The use of an α avoids catastrophic divergences when updating the voltages that are the inputs of the nonlinear elements (i.e., the diodes).

6.3.2. Column Dependency Tree and Chains of Columns

As mentioned in Section 6.2, a dependency tree must be assembled to find out which columns can be processed in parallel. The dependency tree uses the algorithm presented in Code 21, which was adapted from [94]. Then, the columns are organized into levels. A level is a group of columns which can be processed in parallel, and the number of columns that belong to each level depends on the structure of the matrix. Network partition increases the width of the levels and, thus, increases parallelism. After each level is processed, the computation can proceed with the following level. It is necessary to fully process all the columns of each level, so grid-level synchronization is needed, but this is computationally expensive and should be avoided. Fortunately, merging some levels into one by creating "chains of columns" solves this issue. A chain of columns is a group of columns which can be processed sequentially, and each column only requires the previous column to be processed. Therefore, only a local synchronization is needed.

```
1  for k = 1 to n
2      /* Look up for all nonzeros in column k of U */
3      for i = 1 to k-1 where A[i,k]≠0
4          if column of i of L is not empty
5              add i to k's dependency list
6          end if
7      end for
8      /* Look left for all nonzeros in row k of L */
9      for i = 1 to k-1 where A[k,i]≠0
10         add i to k's dependency list
11     end for
12 end for
```

Code 21. Pseudocode to obtain column dependencies (adapted from [94]).

Figure 63 shows how these chains of columns are processed. First, the dependency tree of the columns of the matrix is obtained, and then these columns are sorted into levels, where all the columns of a level can be simulated in parallel. Therefore, all the threads that process a single column within a block can perform a block-wide synchronous operation and simply continue with the next column. On the right of Figure 63 each thread block is represented with a square. One thread block processes 32 columns, and every column has 32 CUDA threads assigned to work on it in parallel. Figure 64 shows the operation to perform the LU decomposition of the j^{th} column of the matrix according to Equations (228) and (229) (the diagram shows the 4th one, which is marked in yellow). It shows how the work is distributed across threads, with each thread in the

same block processing the elements associated with the column of each element in the top row being processed. A thread has been assigned for all the values in the top row which are right of the diagonal of the matrix (marked in black). Every thread first calculates the sub-column it corresponds to, as is shown in Figure 64, where a single thread processes a column. This column will be called k. After that, the column j is subtracted from the k column times $A_{j,k}$, as shown in Equation (229).

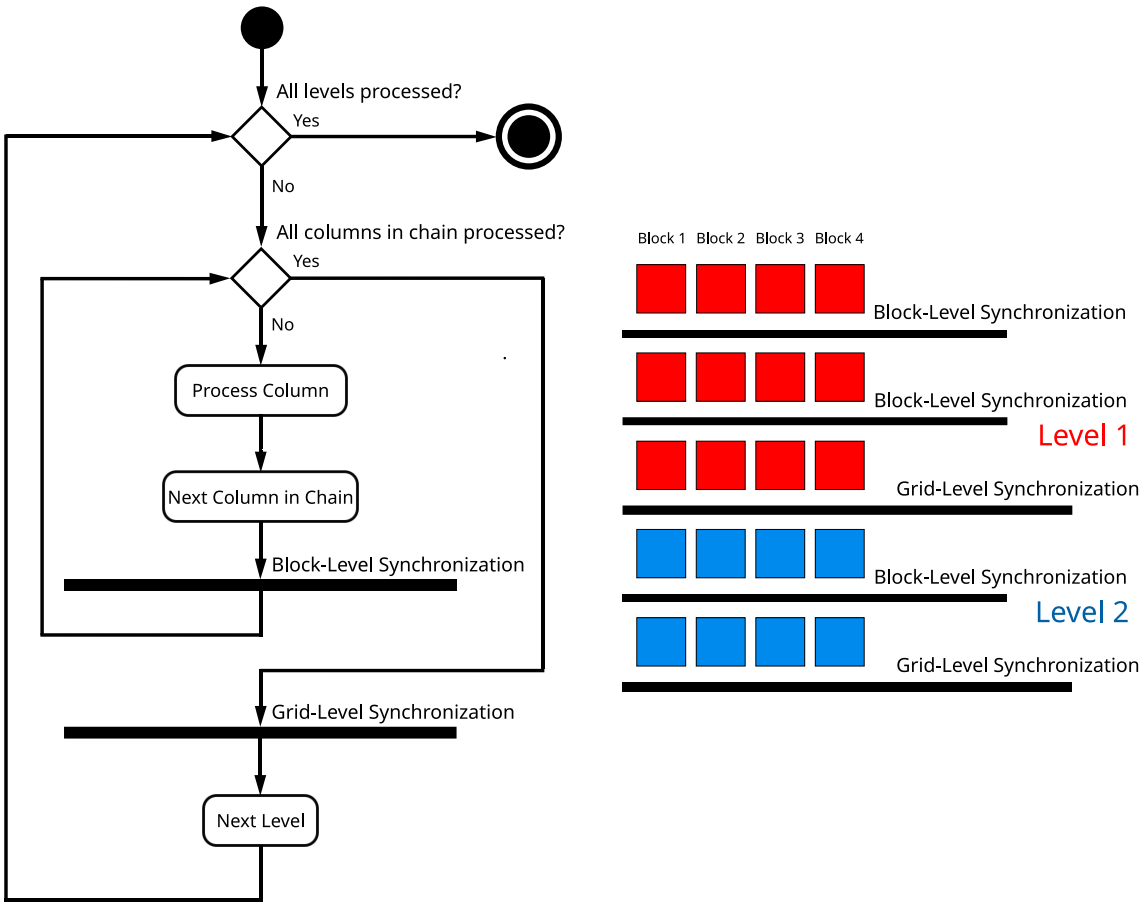


Figure 63. Synchronization of levels of "chain of columns."

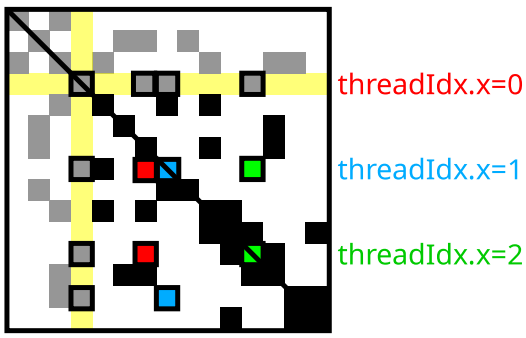


Figure 64. Work distribution across threads.

6.3.3. PVPP Kernel Computation of One Column

The pseudocode in Code 22 further clarifies how the kernel integrates one column, i.e., performing operations in Equations (228) and (229). It has been configured in such a way that each thread block in CUDA will devote 32 threads to process each column (`threadIdx.x`), and that each block will integrate 8 columns (`threadIdx.y`), giving 256 threads per block. Together with the block index (`blockIdx.x`), it will set which column each thread will process. As many blocks as needed to process all the columns will be launched.

```
1  block parallel for blockIdx.x,threadIdx.y = 1 to every available column
2    thread parallel for threadIdx.x = 1 to threads_per_column
3      j =corresponding_column_for_this_level(blockIdx.x,threadIdx.y)
4      global read A[j,j] and write it into shared memory
5      for row_idx = index of elements below of diagonal step threads_per_row
6        i = global read corresponding row for the element in row_idx position
7        global and shared write A[i,j] /= (shared read A[j,j])
8      end for
9      block synchronize
10     for col_idx = index of elements in row j right of diagonal step threads_per_column
11       k = global read corresponding row for the element in col_idx position
12       temp = A[j,k]
13       for row_idx = index of elements in column j below of diagonal
14         i = global read matching row for the element in row_idx position
15         temp2 = shared read A[i,j]
16         for row_idx2 = index of elements in column k
17           global read position of A[i,k] using row_idx2
18           if (corresponding column of row_idx2)==i
19             global atomic addition A[i,k] += (-temp*temp2)
20             break
21           end if
22         end for
23       end for
24     end for
25   end thread parallel for
26 end block parallel for
```

Code 22. Pseudocode of LU decomposition for one column.

The LU decomposition is performed in-place, so both triangular matrices reside in what was the memory of the input matrix *A*. The matrices are also stored in CSC format to reduce iterations. The values of each column of the matrix are stored contiguously. Obtaining the corresponding column for a given element consists in reading the "row index" vector. Data shared across all threads is stored in shared memory to increase performance.

The two vectors that are subtracted are traversed sequentially, which allows the CUDA kernel to prefetch data, thus reducing the overhead due to memory transactions. Finally, when the LU decomposition is ready, a parallel substitution operation is performed to finally solve the linear system, as shown in Code 23. The substitution stage follows the same principles as the LU decomposition regarding vector traversing. These triangular substitution kernels are parallelized by assigning a thread to each vector component. The dependency tree only guarantees that no column needs to read values not yet calculated, not that two columns within the same level shall not update the same value. For this reason, an atomic subtraction is needed to update the values of

the submatrices. CUDA does not have an atomic subtraction for floating values with double precision, hence the addition with a negative sign as shown in Code 22 and Code 23.

```
1  block parallel for blockIdx.x = 1 to every available column
2    thread parallel for threadIdx.x = 1 to threads_per_column
3      j =corresponding_column_for_this_level(blockIdx.x,threadIdx.y)
4      global read A[j,j] and write it into a register
5      global read b[j] and write it into a register
6      block synchronize
7      for row_idx = index of elements below diagonal step threads_per_column
8        i = global read corresponding row for the element in row_idx position
9        if i==j
10         b[i] = global write (global read b[i]/ register read A[j,j])
11       else
12         global atomic addition b[i] += (- (register read b[j])*( register read A[j,j]))
13       end if
14     end for
15   end thread parallel for
16 end block parallel for
```

Code 23. Pseudocode of triangular substitution for one column.

For memory transfers to the GPU, a unified memory model might not be as performant when these memory transactions are not unified. Therefore, memory copies with `cudaMemcpy` have been used as they show better results in applications with a large compute-to-memory transfer ratio [143].

6.3.4. Unified Real Value Matrix

The PVPP has two defined areas: the areas behind PVPP inverters, which run on DC, and the area directly connected to the rest of the PS, which runs on AC. The line models used in the AC section are based on the phasor version of the π-section line, which uses complex quantities. The DC nodes, much more numerous, use DC. Both regions have been unified into a single matrix by converting complex calculations into real ones. The definitions of complex addition and multiplication presented in Equations (236) and (237) respectively are well known. If the first term is identified with the value that is part of the matrix of the linear system, and the second with the variable, these expressions have a matrix counterpart as shown in Equations (238) and (239). Thus, the complex part of the matrix can be expressed as a matrix with the double of rows and columns. This transformation does not alter the chains of columns presented above as Figure 65 shows. Each column which holds complex values is turned into two columns which are dependent on one another, keeping the dependence linearity.

$$(a + bj) + (c + dj) = (a + c) + (b + d)j \tag{236}$$

$$(a + bj)(c + dj) = (ac - bd) + (ad + bc)j \tag{237}$$

$$\begin{bmatrix} a \\ b \end{bmatrix} + \begin{bmatrix} c \\ d \end{bmatrix} \tag{238}$$

$$\begin{bmatrix} a & -b \\ b & a \end{bmatrix} \cdot \begin{bmatrix} c \\ d \end{bmatrix} \tag{239}$$

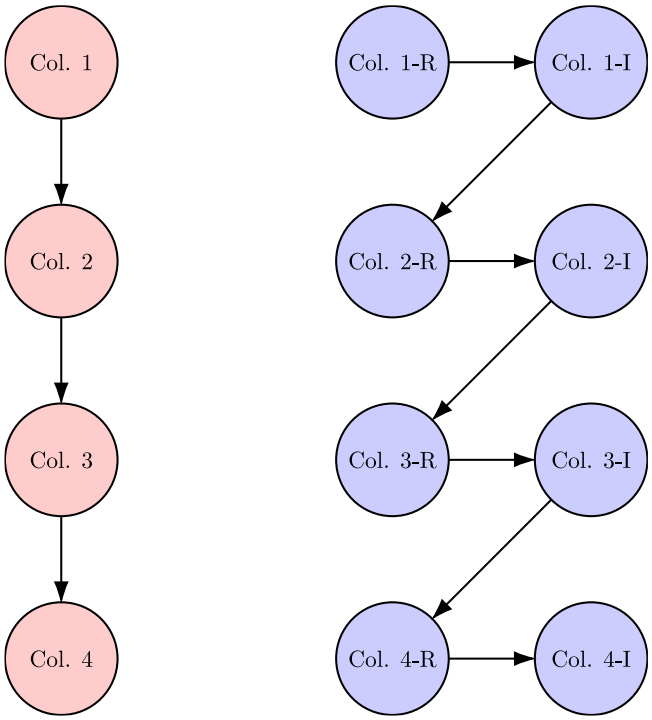


Figure 65. Complex to real column dependence.

Chapter 7. Case Studies

Based on the kernels and techniques proposed in this thesis, this section will introduce the case studies of both the wind farm and the PVPP. All the simulations were carried out with a computer equipped with an *Intel i9-10900K* CPU at 3.70GHz (10 physical cores), 96 GB of RAM and an *NVIDIA RTX 3090* graphics card whose maximum Compute Capability is 8.6. The operating system of the machine is *Ubuntu 22.04 LTS*. The rest of the specifications of the graphics card can be found in Annex B.3. Section 7.1 details the simulation of the standalone wind turbine models while Section 7.2 expounds the results and computational cost of simulating the whole wind park. The results of simulating the PVPP are commented in Section 7.3.

7.1. Standalone Wind Turbine Parallel Simulation

This section will check the validity of the wind turbine kernels developed in Chapter 6. First, the DFIG will be modelled in standalone mode. In this mode all wind turbines are isolated and there is no grid connecting them. To check the validity of the simulation, both a voltage dip and a gust of wind will be simulated. Voltage dips are sudden drops in the grid voltage level caused by short circuits. The voltage dip to be simulated corresponds to the REE proposal [144] and it is pictured in Figure 66, which corresponds to a voltage dip down to 0.2 p.u. with a recovery time of 15s. On the other hand, the gust of wind inputs keeps the voltage at the wind turbine's terminals constant but varies the wind speed as shown in Figure 67. A Simulink implementation of both the nonlinear and linearized models of the wind turbine was also developed to check the correctness of the C++ and CUDA implementations.

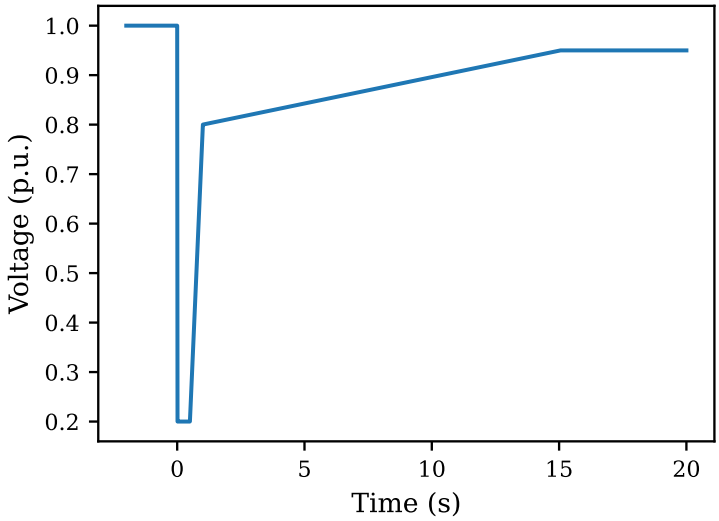


Figure 66. Voltage dip.

Figure 68 shows the stator and rotor currents in *dq0* components (i_{sd}, i_{sq}, i_{rd}, i_{rq}) of the wind turbines equipped with a DFIG expressed using the 5th order model. When a voltage dip occurs, the magnetic fluxes of both the stator and the rotor experiment transients which reach up to 5 times their value pre-voltage dip, to eventually stabilize quickly. On the other hand, Figure 69

represents those same magnitudes when the 3rd order model of the DFIG is considered. As shown, electromagnetic transients have disappeared and have been substituted by their average value. It can be shown that the behavior of the linearized models is a bit different than their nonlinear counterpart during the transient, and that difference increases the further from the equilibrium point the DFIG is. The linearization method (see Section 3.8) approximates nonlinear functions in their vicinity. A voltage dip entails sudden voltage changes and is therefore far away from the equilibrium point, where the differences between nonlinear and linearized elements are larger. Nevertheless, the linearized system captures the general behavior of the system, showing large spikes of current. All linearized models return the same values, and they have been represented with dotted series to avoid line overlapping. Comparable results can be found when plotting the generated electric power and the electric torque of the wind turbines. Figure 70 shows how these magnitudes vary depending on the model of the DFIG used. There are voltage differences at its lowest, but the linearized model captures the general form of the transients of the machine. During a voltage dip, the voltage magnitude is the furthest away from the equilibrium point, which explains the discrepancies shown in Figure 70.

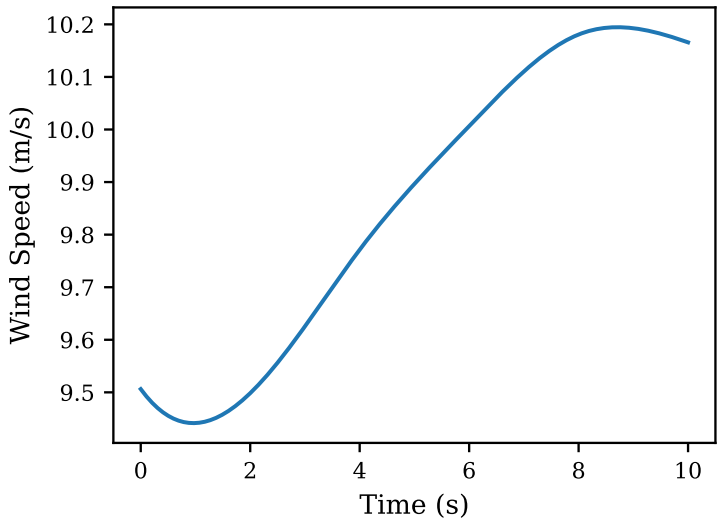


Figure 67. Gust of wind.

The picture is quite different when simulating gusts of wind. Figure 71 shows the current passing through the terminals of the DFIG when the wind turbine faces a gust of wind. The general shape of the obtained values is similar, but it can be appreciated a small offset in the current values. The 3rd order model simplifies away the electromagnetic transients of the DFIG, and, because the electromagnetic transients are not significant in the simulation of gusts of wind, the obtained values will be virtually similar, as Figure 72 confirms. The values of the electrical power and electric torque have also been represented in Figure 73 for the sake of completeness. As presented, there are no differences between 5th and 3rd order models.

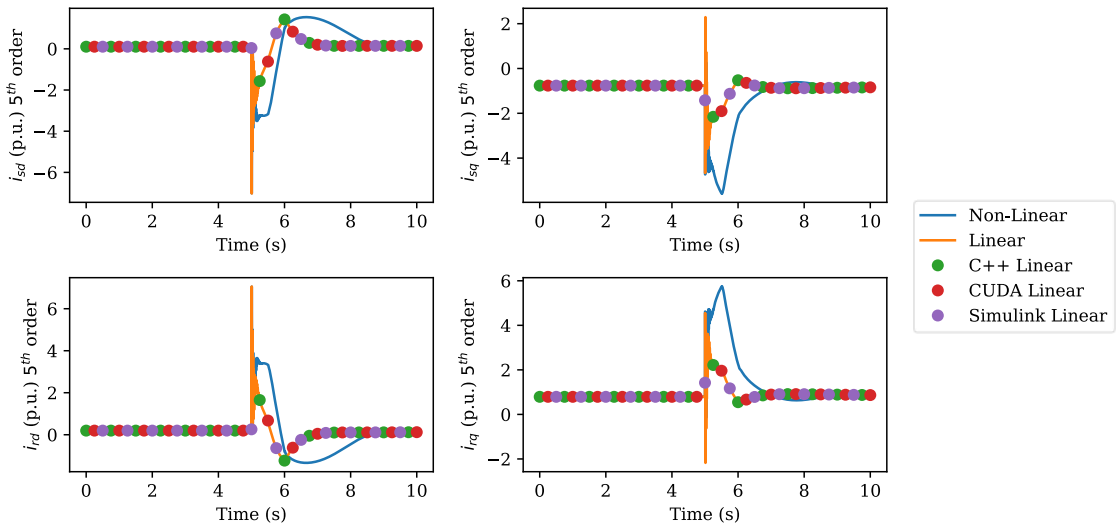


Figure 68. Stator and rotor currents for voltage dip (5th order).

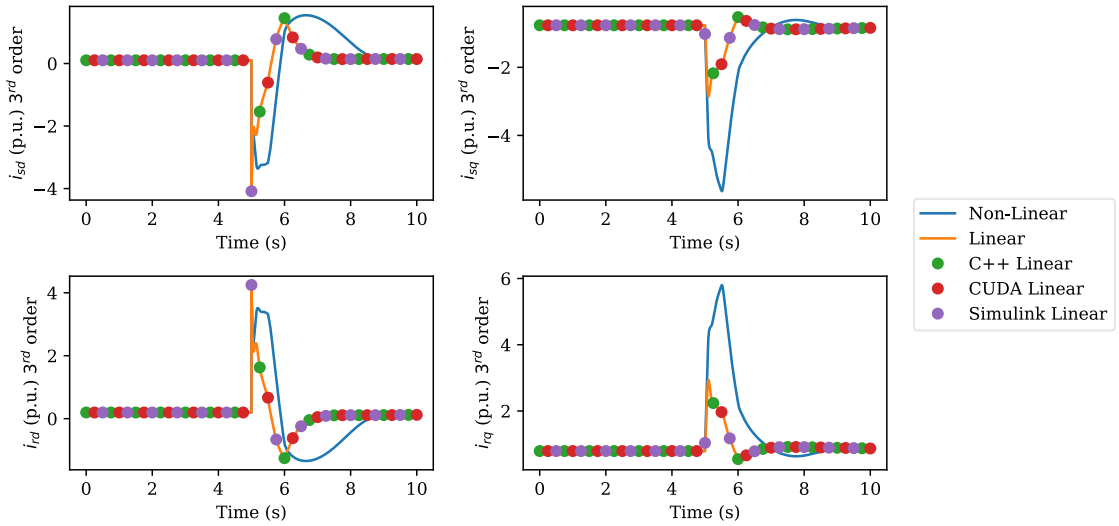


Figure 69. Stator and rotor currents for voltage dip (3rd order).

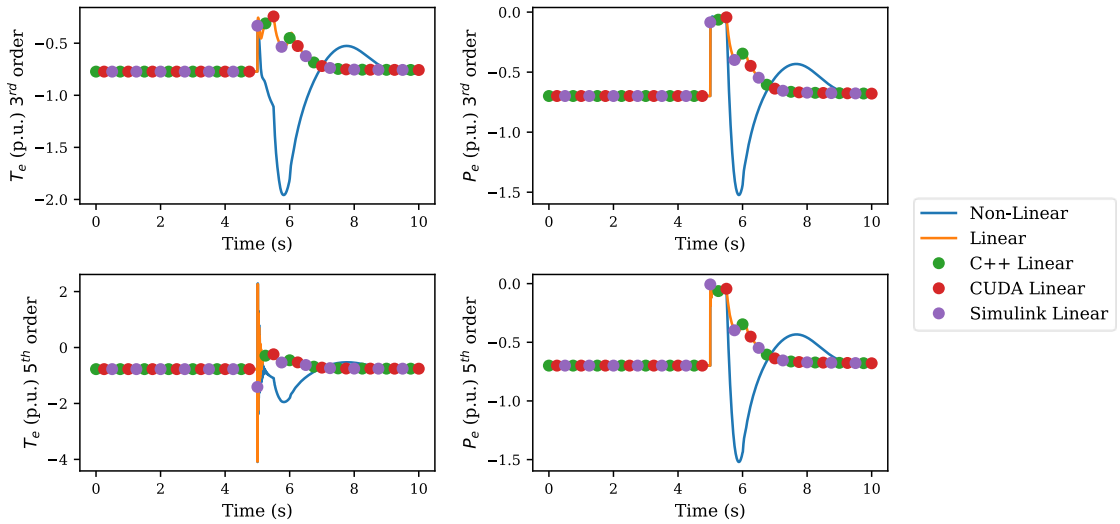


Figure 70. Power and electric torque output for voltage dip (5^{th} and 3^{rd} order).

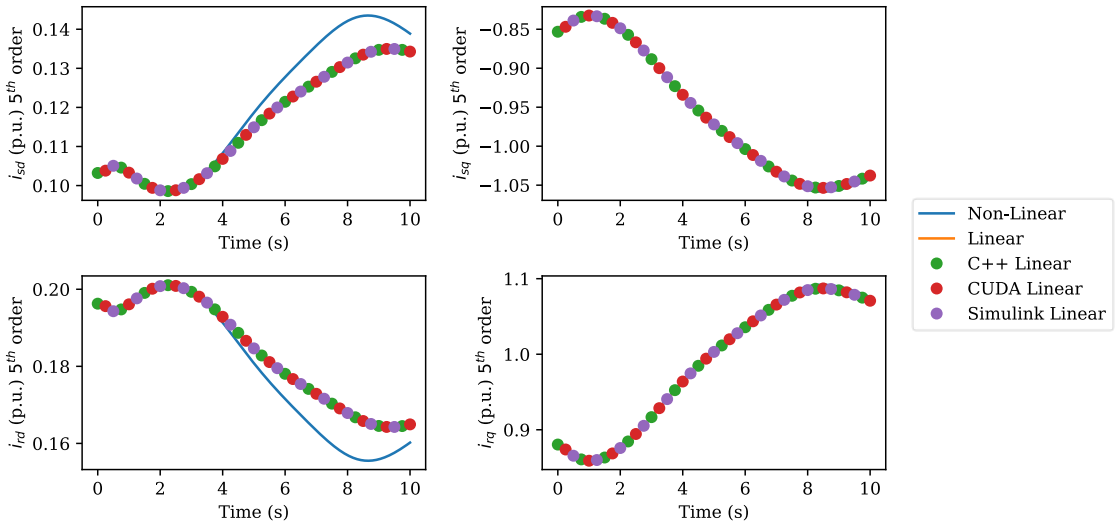


Figure 71. Stator and rotor currents for gust of wind (5^{th} order).

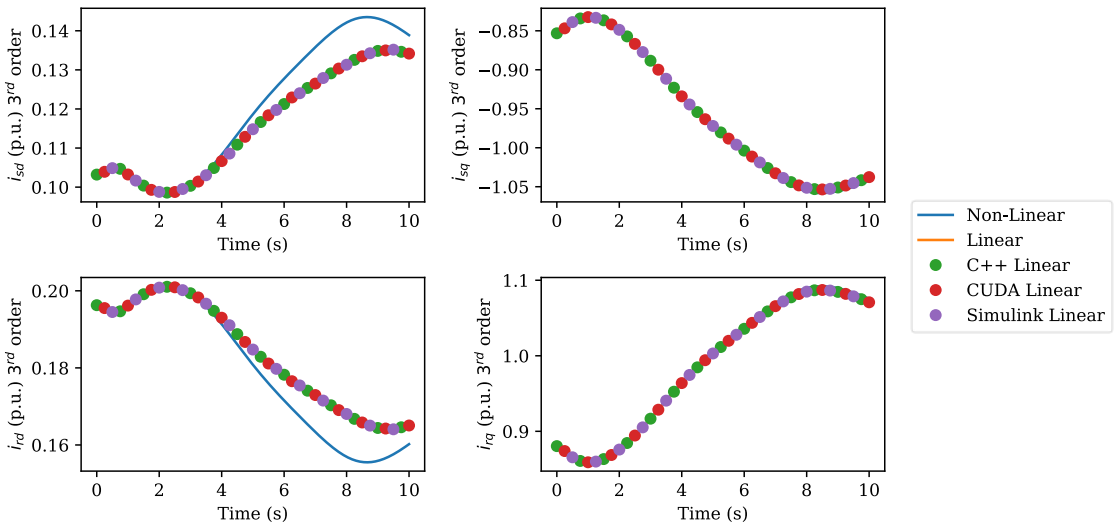


Figure 72. Stator and rotor currents for gust of wind (3rd order).

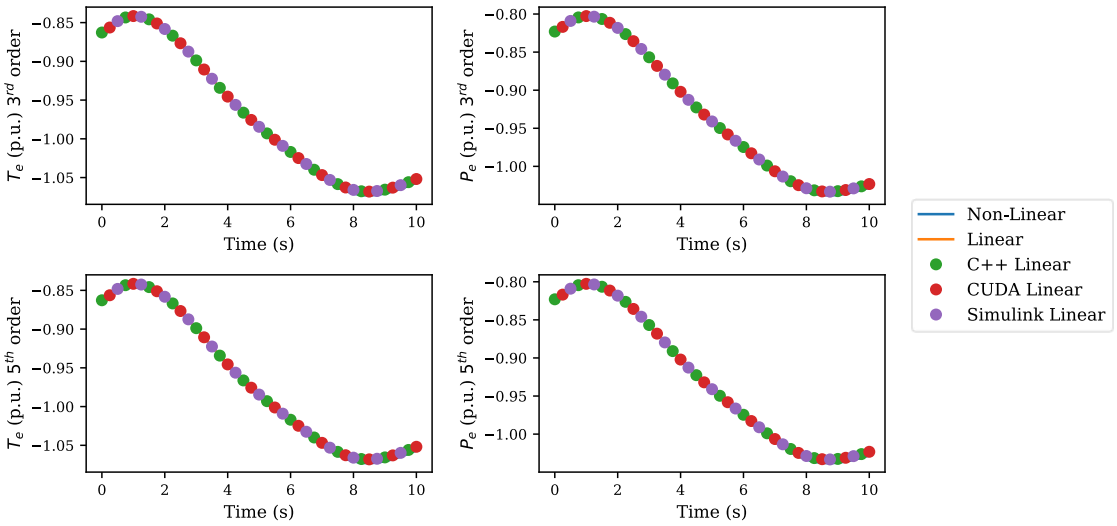


Figure 73. Power and electric torque output for gust of wind (5th and 3rd order).

Finally, it has been plotted in Figure 74 the angular speed obtained for all simulations performed in this section. These results show that the linearized system properly captures the changes of rotor speed for gusts of wind, while there are differences in the behavior on the evolution of the rotor speed depending on whether a nonlinear or linearized model is used. In the case of the voltage dip both models eventually return to the equilibrium point.

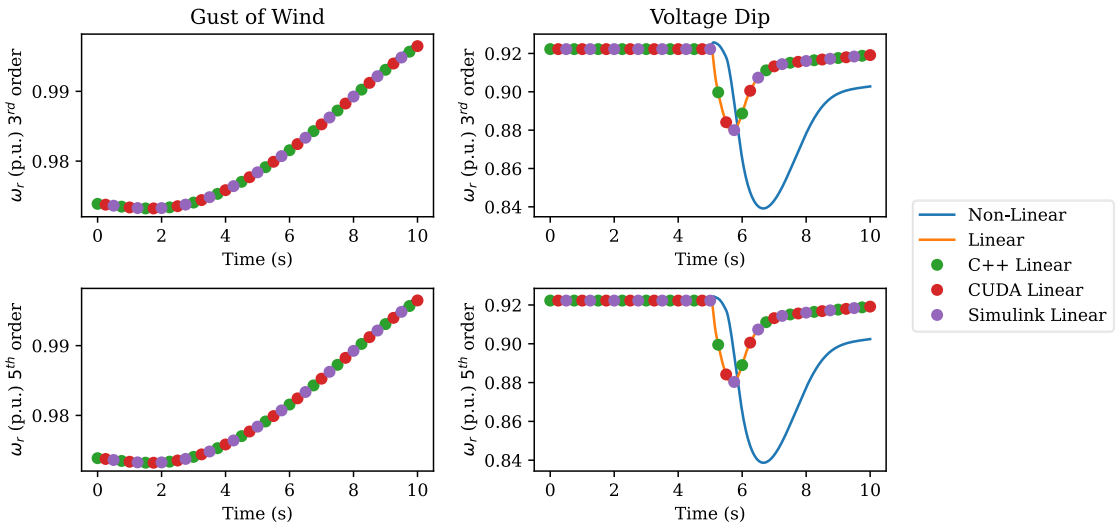


Figure 74. Rotor speed for a gust of wind and a voltage dip (5^{th} and 3^{rd} order).

To conclude this section, it has been checked whether Heun's method is an adequate method of integration. To that end, the linearized items were integrated with a variety of fixed step numerical integration methods, such as the Heun, RK-41, RK-42 and RK-5, where the latter are methods part of the Runge-Kutta family [36]. As shown in Figure 75, the validity of the simulation does not depend on the integration method.

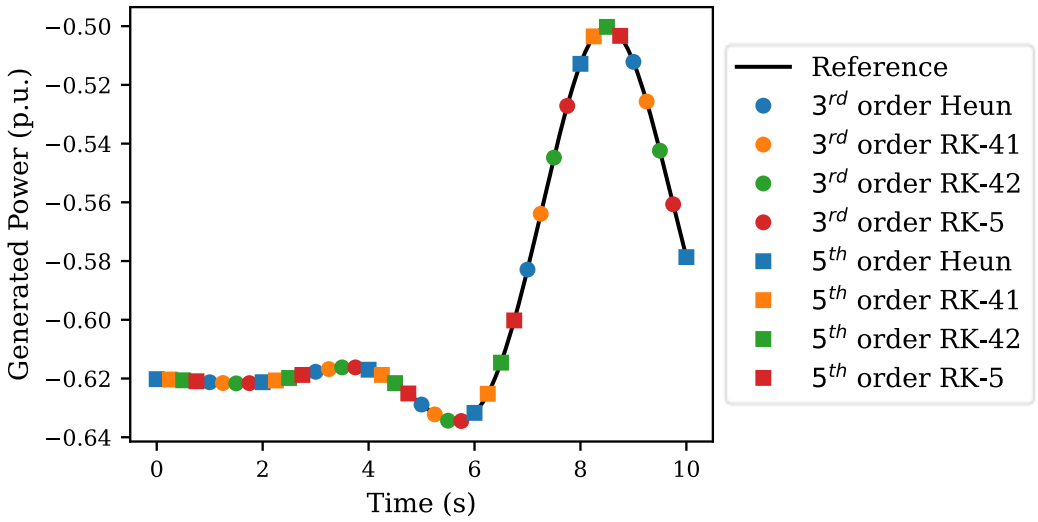


Figure 75. Comparison of integration techniques.

7.2. Wind Farm Parallel Simulation

The system simulated is a wind farm whose wind turbines are equipped with DFIGs and are connected in series at 33 kV. Each string of turbines is composed of 5 wind turbines. Their power rating is 2 MW. The DFIG is represented by a 5^{th} order model whose internal parameters are also shown in Annex B.1. As presented, the 3^{rd} order models presented instabilities (see Section 3.12) and were not adequate for the global wind farm model.

The internal generator voltage of the wind turbines is lower than the grid voltage, so a step-up transformer for each wind turbine is added between the wind turbine and the power line which was represented as an impedance. Figure 76 shows the internal layout of the wind farm simulated in this work, where the checked rectangle represents the external grid, which is considered as a constant voltage source. This consideration rests on the fact that the wind park is connected to a well meshed grid with large generation units in the vicinity. This configuration of connecting the wind turbines in series has been chosen due to its low cost, thanks to the low total length of power line necessary, as noted in Section 6.2. The concentration node is separated from the external grid with a power line, and the subsystems of the wind farm correspond to each array of wind turbines.

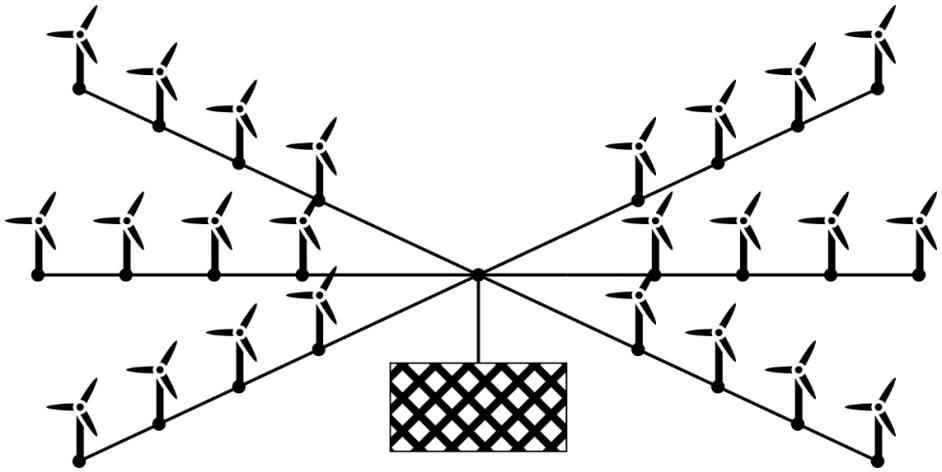


Figure 76. Wind farm layout.

The voltage dip was also simulated using the Simulink tool for different timesteps and the results are shown in Figure 77 for a wind farm of 20 wind turbines. As it can be seen, the results, which is the current of the whole wind farm, obtained in Simulink are like those of the proposed implementation.

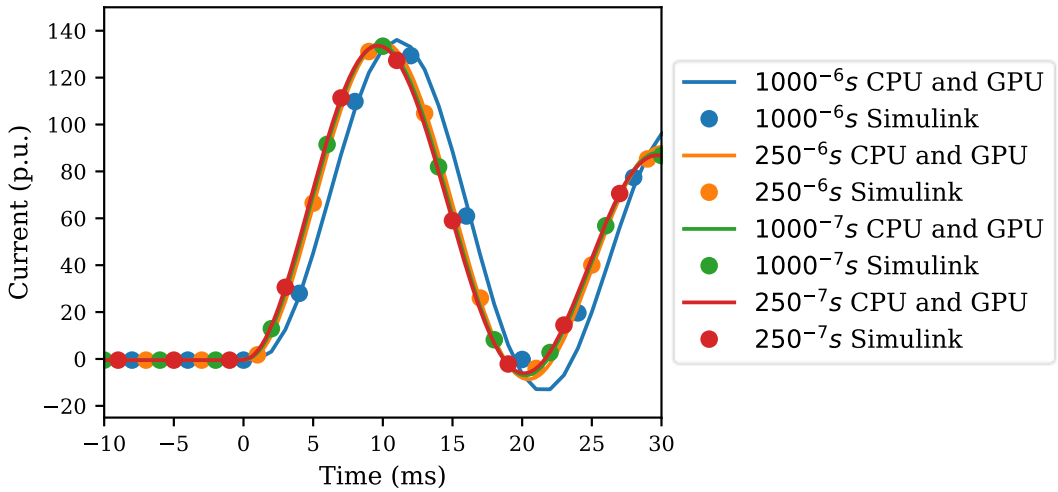


Figure 77. Voltage dip case in Simulink vs CUDA implementation.

As for the synchronization strategy to be used with the wind farm to be simulated, cooperative groups, multiple kernel launches and, as a reference, block level synchronization with `__syncthreads` were tested against each other. If the wind turbines have been integrated

independently, the voltage at each timestep would have been copied to the GPU memory before kernel launches for maximum performance, and no global synchronization would be performed, for the wind turbines do not exchange data. The reason for doing so is the computational cost of global synchronization. To evaluate the effect of the different synchronization primitives, the wind turbines were integrated independently with an additional integration primitive, as shown in Figure 78. In the case of implicit kernel synchronization, the loop is outside the graphics card, launching a kernel to process a single timestep. The results are shown in Figure 79. As mentioned above, the approach that is based on launching a kernel many times in a loop is the slowest one, while a single kernel launch using grid-wide sync is faster. Nevertheless, there is an overhead related to the global synchronization stage that cannot be avoided.

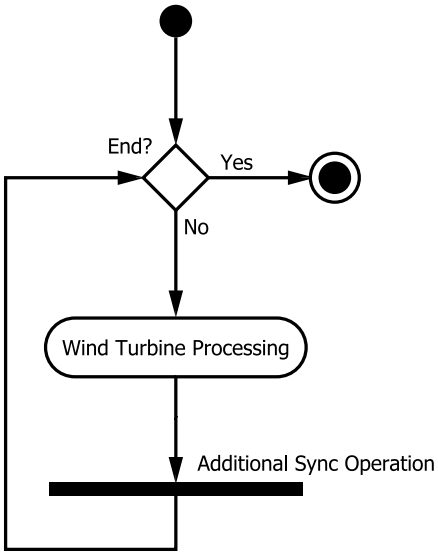


Figure 78. Test program flowchart for synchronization strategies.

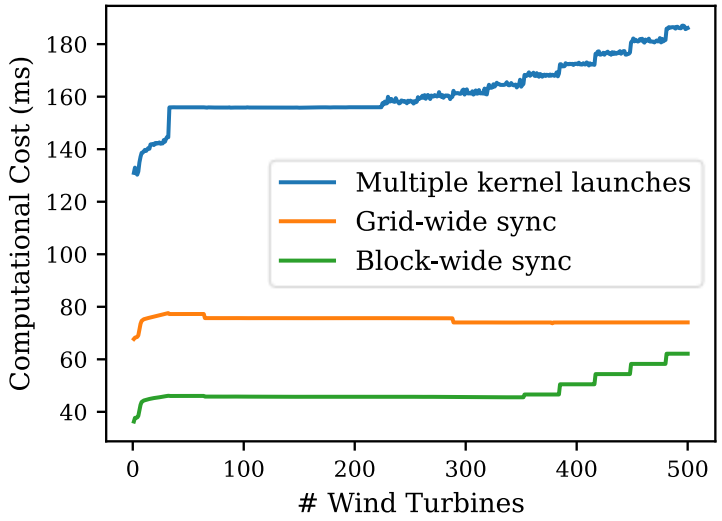


Figure 79. Computational cost of synchronization strategies.

The costliest operation in the power system implementation is the node voltage solver. The need to globally synchronize threads posed a challenge. To avoid the kernel-launching overhead and prevent unnecessary memory read operations, cooperative groups were implemented in the kernel. Due to the way the linearized wind turbines have been integrated in this kernel, the number of threads in each block must be at least $32 \cdot number\ of\ states$, which is equal to 32·11=352 threads. The number of threads per block has been raised to 512, so the parallel vector reductions require fewer blocks.

For grids whose groups are approximately of 10-15 nodes, such as the one in this case study, it has been empirically tested the error obtained while solving the general linear system with a LU decomposition and with partial matrix inverses, i.e., only the matrix of the subcircuits is inverted. For this purpose, a linear system whose matrix is the admittance matrix of the system has been solved, and all the values in the right-hand vector have been set to one. Then, the Mean Absolute Error (MAE) of each solver has been calculated by comparing each solving technique, using different data types, against the LU factorization with long double numerical precision. As is shown in Figure 80, the matrix inverse is the method that has the highest MAE compared with the rest, but such an error pale in comparison with the error introduced by different numerical types. We can also see how the partitioning, where the matrix inverse is constrained to small blocks, gives an error comparable to that of the LU factorization. The matrix that represents the system connections was found to be sparse regardless of the wind turbines simulated, while the node order also impacted the fill-in of the matrix.

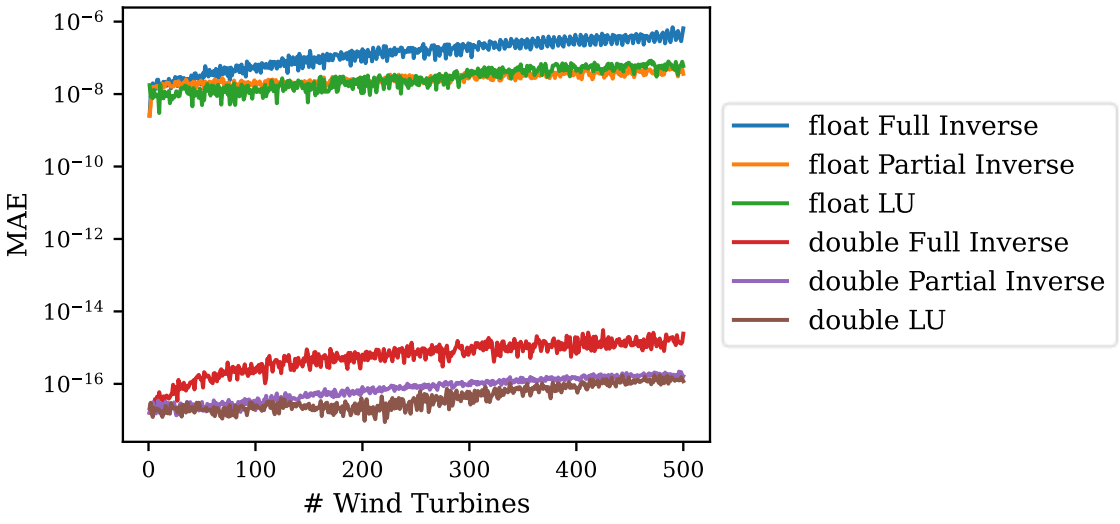


Figure 80. Mean absolute error of linear system resolution.

To avoid kernel-launching overhead and prevent unnecessary memory read operations, cooperative groups were also implemented in the kernel. To further test the limits of the cooperative groups strategy, it was launched an empty kernel in cooperative launched mode in the *NVIDIA RTX 3090* (see Annex B). Figure 81 shows the maximum number of blocks depending on the number of threads per block. According to Figure 81, only 328 blocks at maximum can be launched in this GPU using this configuration. This shows the need to maximize thread usage.

To check the computational speedup of the CUDA implementation, two main implementations have been compared: one is a sequential C++ implementation while the other takes advantage of

the CUDA platform. The wind turbines are solved in the C++ implementation using the linearized system in a sequential way, while the rest of the implementation performs a back-and-forth substitution using the triangular and diagonal matrices outputted by the LU decomposition. The reason to choose this method on the CPU is that it is the one that needs to compute fewer mathematical operations given that parallelism is not a concern. Integrated solutions such as MATLAB-Simulink were ruled out due to their lack of global synchronization primitives and their reliance on external libraries such as cuFFT and cuBLAS [12], which are not adequate due to their overhead due to multiple external calls. They have just been completed for the sake of checking the validity of the simulations (see Figure 77).

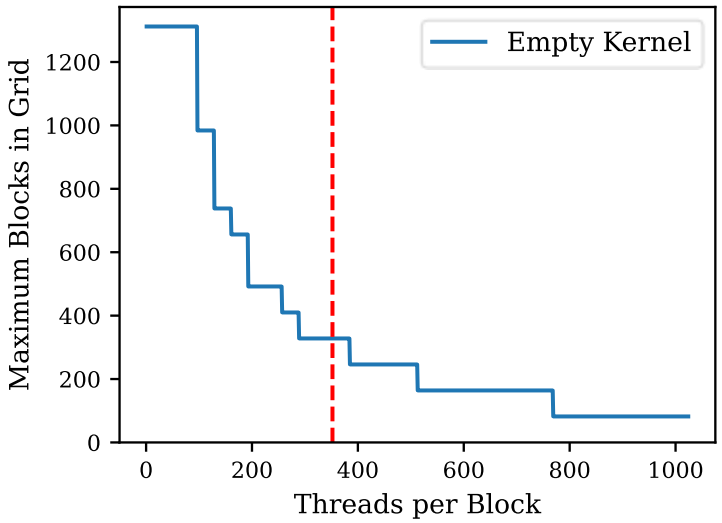
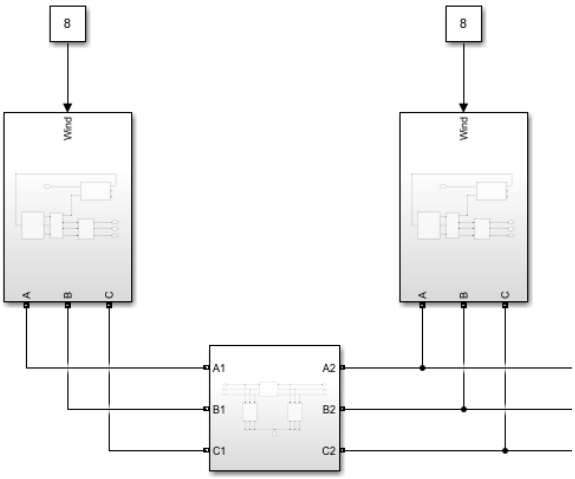


Figure 81. Threads per block vs maximum number of blocks in cooperative mode.

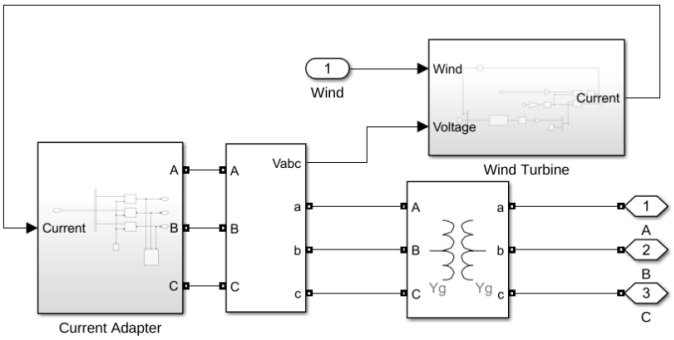
The blocks that compose the Simulink implementation are shown in Figure 82. The power system has been implemented using the SimPowerSystems blockset, which internally discretizes the passive elements (coils and inductors) and solves the global power system, while integrating the rest of the blocks with Heun's method. Each wind turbine is connected by a power line (a), while the state-space system (c) integrates the matrices while converting the outputs to and from the *dq0* reference frame. An adapter between the native Simulink blocks and the SimPowerSystems blocks is needed to communicate these two blocks (b).

The relative speed-up of the three implementations is shown in Figure 83. All three solutions were compared against the C++ implementation. As expected, with a small number of wind turbines and/or electrical nodes, the GPU CUDA kernel performance is worse than its CPU counterpart. This happens due to the overhead introduced by CUDA, and the fact that the parallelization available when integrating a small number of wind turbines is limited. However, when 65 wind turbines are simulated, the GPU CUDA kernel is faster. And it shows as well that when 2464 wind turbines are simulated, a speedup of 26.39x is obtained.

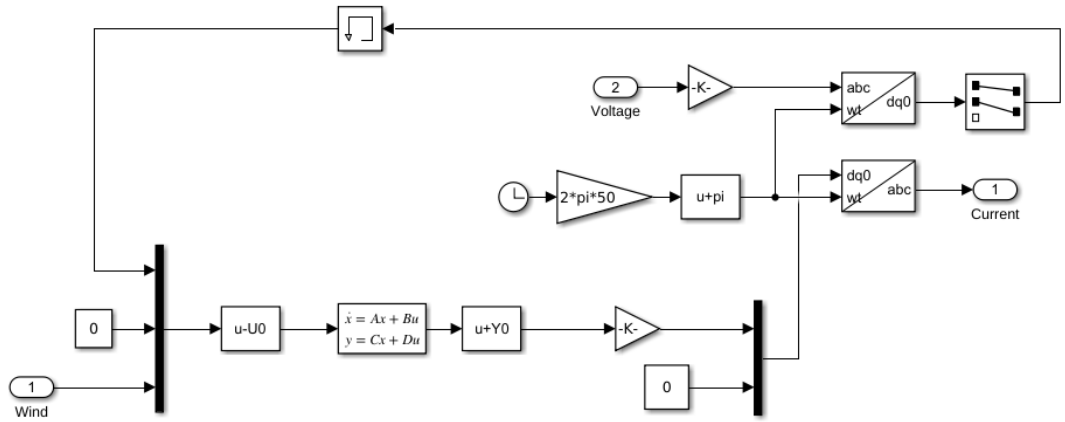
The Simulink implementation is the least performant, and it was found that its computation time grew quadratically. On the other hand, the computational cost of the C++ and CUDA versions grows linearly. For the larger wind farms, the Simulink version is orders of magnitude slower than its counterparts. The Simulink simulation was, in fact, 19000 times slower than the CUDA kernel for the highest number of wind turbines simulated.



(a) Block model of array of wind turbines



(b) Wind turbine adapter



(c) Wind turbine state-space integrator

Figure 82. Wind farm Simulink implementation.

The main bottlenecks of the CUDA implementation are the global synchronization and the amount of inactive CUDA threads. Despite the cooperative groups being an optimized way of globally syncing threads, it is nonetheless a scheme based on atomic operations on values stored in global memory. Works such as [43] deal with power system simulation by working out the solutions using a mixture of atomic functions and block-level synchronization. The computational cost obtained in that work when calculating the node-voltages was at best 26% lower, while in this work the kernel was up to 26.39x faster. The substitution of loops with parallel reduction gives better results, despite the substantial number of threads required. However, the use of

cooperative groups puts a limit on the number of wind turbines that can be simulated, as only up to 2464 wind turbines can be simulated in parallel. The limiting factor is the number of blocks that can fit inside the SM of the graphics card. The main bottleneck that prevented more blocks from fitting was the kernel register count. Using the `maxrregcount` directive, which allows aggressive optimizations to reduce the register count [12], proved to be essential to simulate a large number of wind turbines. The method of PS processing was able to parallelize a PS when the alternatives could not, and without using aggregated models as previously presented in [75] and [76]. For instance, the authors of [43] claimed a speed-up of 30x and 55x depending on the GPU used. However, their code for processing the power system grid is only 35% faster on the GPU. Most of the speed-up claimed by them lies in the parallelization of the control system, which is more complex. The wind turbines were simulated using linearized systems in a state-space system. Therefore, the proposed kernel could parallelize power systems that were previously hard to parallelize. Other works such as [97] do not present global speed-up figures, and the global AC grid computation cost is unknown and cannot be compared.

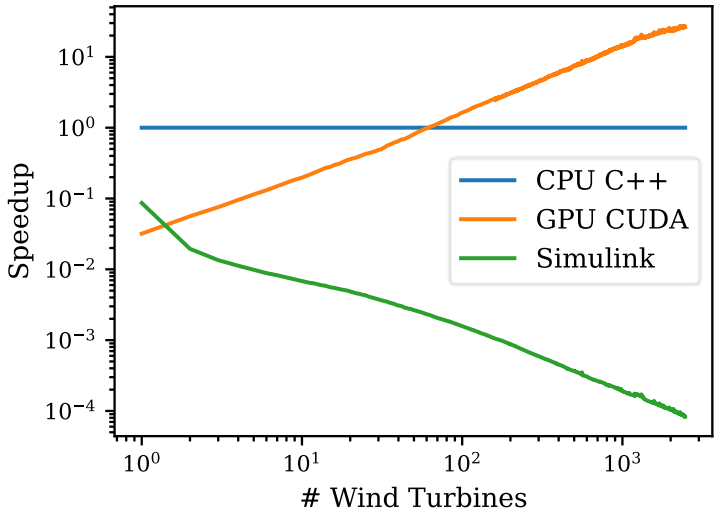


Figure 83. Wind farm simulation speedup.

Works such as [84] and [85] used delayed line models, as mentioned in Section 5.2. The main bottleneck was caused while calculating X_u (see Section 6.2.3). The power simulated in [85] spans the whole USA mainland, while the overhead lines in [84]. reach up to 500km. These power lines offer a convenient point of rupture to create subsystems that can be easily parallelized. The maximum number of nodes studied in [84] was 5, while the approach does without delayed line models and still achieves a notable speed-up. A wind farm cannot be simulated with this strategy because calculating this value is a serial operation and that stalls all the threads. The proposal of this thesis was able to parallelize the power system simulation without breaking the power system, and therefore without compromising the accuracy of the results.

Other works such as [96], [80], and [81] give an insight into the issue posed by the memory and kernel overhead. The process in [96] is divided into multiple small kernels, while some tasks are offloaded to the CPU, resulting in speed-up loss, while the works [80] and [81] need batch sizes in the order of thousands to reach noticeable speed-ups. The proposal in this thesis also exceeds the speedup in [94].

7.2.1. Computational Energy Consumption Study

Finally, an energy consumption study was carried out, and its results are shown in Figure 84. This energy consumption of the machine was measured by obtaining the current drawn by the whole computer with the specifications shown in Annex B.3 with no peripherals attached. It was found that the CPU consumed approximately 109W while working on the Simulink and C++ simulations, while the CUDA implementation required 194W. The C++ implementation is the most performant solution for wind farms with a small number of wind turbines. The total energy consumption of the CUDA version remained stable since its running time was relatively constant. While simulating more than a hundred wind turbines, the C++ implementation starts consuming more energy due to the increased running time. It was found that the C++ version consumed 14.81 times more energy than the CUDA version for the largest wind farms, saving up to 1549 joules of energy. Therefore, CUDA is preferred for the simulation of large wind farms due to its low computing time and energy use. The energy consumption of the Simulink implementations remained high due to the extra running time.

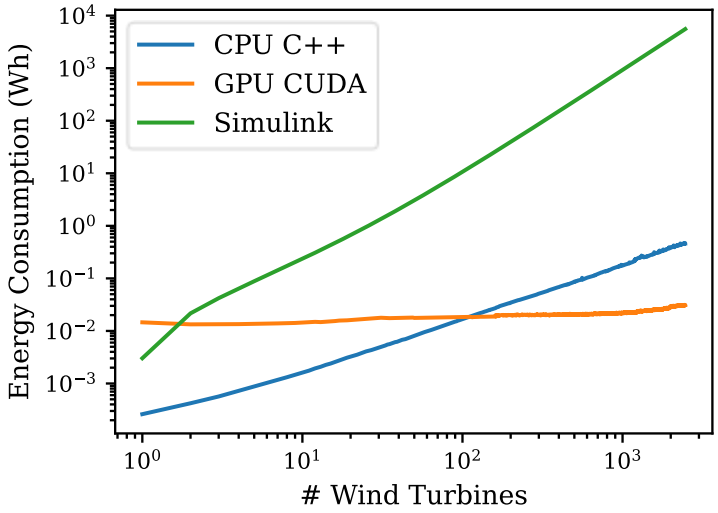


Figure 84. Wind farm simulation energy consumption.

7.3. Photovoltaic Power Plant Parallel Simulation

The second case study proposed in this thesis is related to the simulation of a large PVPP. It is proposed a PVPP that is similar to the "Núñez de Balboa" PVPP. This power plant has 350 MW of installed capacity and spans 10 km^2 [145]. A hierarchical topology was laid out to connect such an enormous number of PV panels. Starting from the top, on the first level of interconnection, a step-up transformer is deployed which connects the PVPP to the rest of the power system. Then, power inverters feed all the generated DC electric power into the AC grid. The nominal power of the installed inverters of the power plant is 1.5 MW, and up to 330 power inverters have been installed in a ring layout. Each inverter, in turn, is connected to 2500 solar panels with 400 W of nominal power and deployed in the layouts shown in Figure 37 in Section 4.3.

The irradiance and temperature conditions are represented as a 2D monochrome bitmap whose values correspond to the irradiance and temperature, respectively. Before each iteration, these

values are considered by each single PV panel given its spatial location. Each pixel of this irradiation map corresponds to a single PV panel. A simplified dataset, where the temperature of each PV panel is constant across the PV farm, was used in the case study for the sake of simplicity, although real weather data can also be used if available. Irradiation maps and other weather data are available via tools such as, for example, PVGIS (Photovoltaic Geographical Information System)[19].

The conditions considered were a sunny day with a single cloud passing over the PVPP. This cloud had a constant linear speed, and its shape did not change, although the number, speed, and shape of clouds can be modified in order to study their impact. The simulated cloud was oval-shaped and reduced the irradiance by up to 70%, and it occupied approximately 5% of the total irradiation map. It is plotted in Figure 85 the power generated by the PVPP when a cloud passes over. It was considered an irradiance of $G = 1000 \frac{W}{m^2}$ and a PV panel temperature of $T = 25ºC$, which are the default conditions under which PV panels are tested [146]. An oval-shaped cloud that covers 100·50 panels was considered as well. This cloud reduced the solar irradiance received by the affected PV panels by 70%, as can be inferred from the irradiance drops in [147]. The dip reaches up to 2 MW (plotted using the per-unit system, where the generated output is divided by the base power, i.e., the nominal power of the PVPP, 350 MW). This dip was caused by just one cloud in a matter of minutes, and it should be considered by the transport and distribution grid operators to tune their predictions.

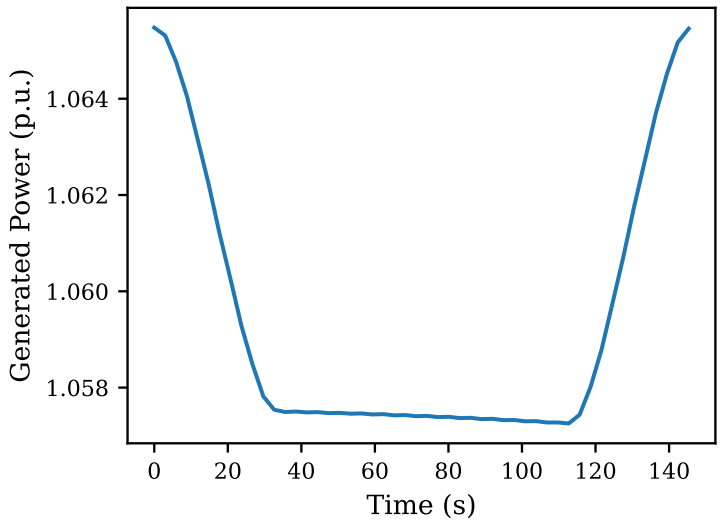


Figure 85. Generated power when a cloud passes over.

Figure 86 shows the power generated by the PV panels under partial shading conditions. Thanks to the parallelism achieved, it is possible to obtain these results on a per-panel basis. The different connection layouts have been tested by considering that a cloud covers approximately the top left corner of the block of PV panels. The number in parentheses represents the average power generated per panel. The PV panels are connected horizontally in series, while extra connections across strings have been added vertically for the fully connected, bridge, and

[19] JRC Photovoltaic Geographical Information System (PVGIS). https://re.jrc.ec.europa.eu/pvg_tools/

honeycomb architectures. As shown, the shaded PV panels do not produce energy, and can also drag other PV panels' generated output down, and that is considering that the inverters have set an optimal voltage for all their strings. An unshaded panel has no voltage drop, with the voltage drop increasing in the following panels to match the global voltage set by the inverter, the rest of the PV panels move away from their optimal voltage. A fully connected layout smoothens the voltage differences out, allowing for more generated power. The bridge and honeycomb strike a balance between higher generated power and reduced installation costs due to a smaller amount of wiring.

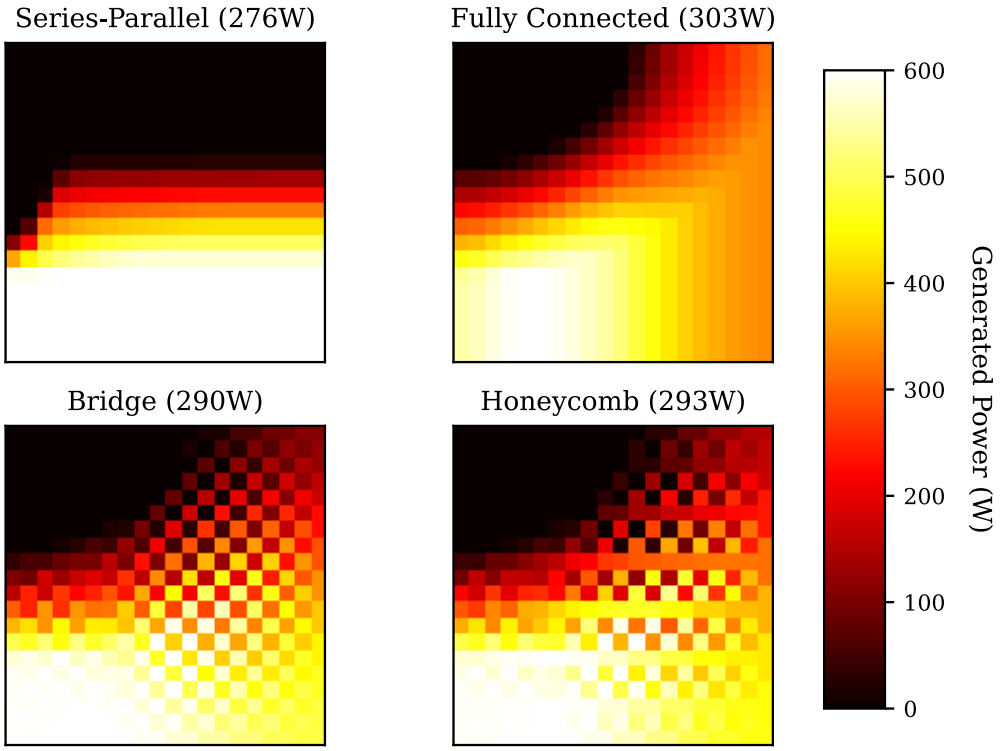


Figure 86. Generated power while partially shaded under different configurations.

The generated power also depends on the health of each individual solar panel. A PV panel that is not producing power due to a malfunction will drag close PV panels down. Figure 87 shows the generated output of the PVPP where a panel is broken. A broken panel acts as a closed circuit, which makes the voltage difference across its terminals equal to zero. A fully connected layout shorts all PV panels in parallel, creating the largest power drop. The bridge and honeycomb schemes distribute the voltage loss across the neighboring panels, while the series-parallel configuration isolates the issue to the string to which it is connected. In this case, the voltage across all PV panels in a series-parallel slightly rises to compensate for the lack of voltage drop across the terminals of the broken PV panel. This information can help the operators of the PVPP to quickly pinpoint issues with individual panels, and thus avoid financial losses in advance thanks to the simulation of the system response under multiple conditions in minutes. Variations in the generated power related to varying the temperature and the irradiance are shown in Figure 88 and Figure 89. The base conditions for testing the PV panels are $T = 25ºC$ and $G = 1000\frac{W}{m^2}$. The generated power is related to the irradiance and negatively related to the temperature increment relative to the base conditions. These results follow the well-known behavior of individual PV panels.

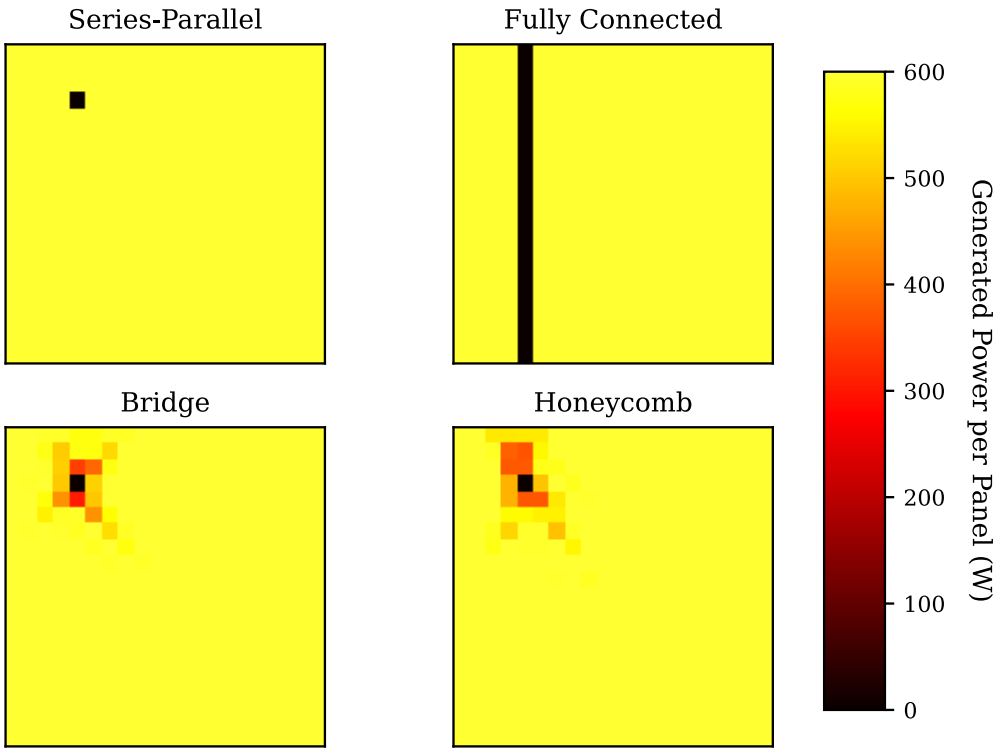


Figure 87. Generated power while a PV panel is broken.

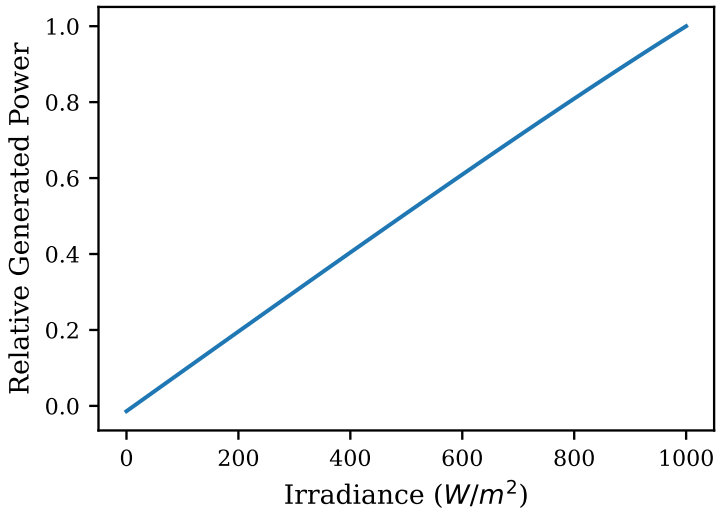


Figure 88. Generated power relative to base conditions when the irradiance changes.

An aggregated model of the PV panels has also been created. This aggregated model has been represented as a large and unique solar panel. This model output is featured in Figure 90. As shown, the detailed model is less optimistic, and gives a more accurate output of the PVPP. This shows that the detailed model that has been used in this thesis is more suitable for profitability studies of PVPPs. As can be seen, the aggregated model returned a larger generated power. Prediction of the generated power is essential for the management of the global power grid and for that reason the whole PVPP was simulated with precision. The transport and distribution grid operators demand results that are as accurate as possible and, as shown in Figure 90, the

aggregated model difference can be as much as 0.03 $p.u.$. This corresponds to a difference of 10.5 MW, which should be taken into account.

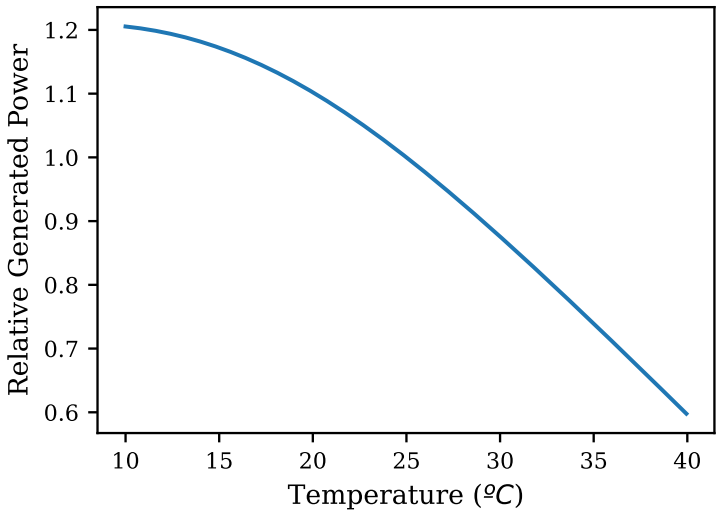


Figure 89. Generated power relative to base conditions when the temperature changes.

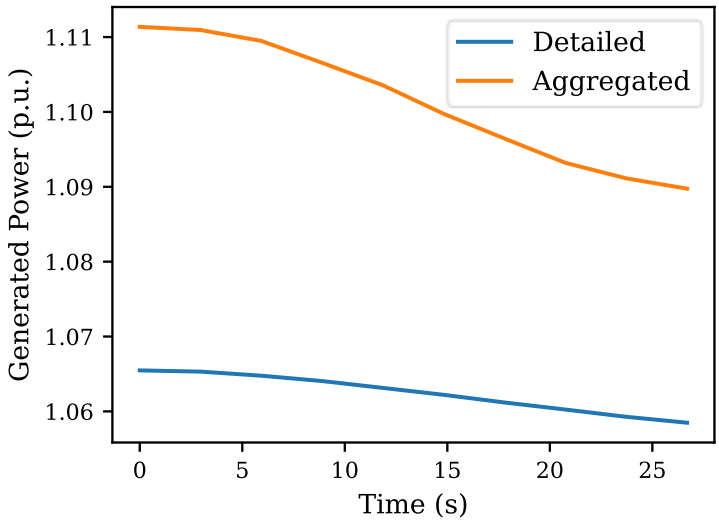


Figure 90. Generated power with and without aggregated models.

The amount of parallelizable potential is shown in Figure 92, which also shows the number of columns that can be run in parallel for each level. The alternating coloring scheme corresponds to the kernels launched. The global synchronization needed is greatly reduced by grouping columns as chains, as was shown in Section 6.3. Only 17 kernel launches are needed to process the matrix to simulate a 350 MW PVPP for the fully connected layout, while only 4 kernel launches for the serial-parallel configuration. The computational cost, on the other hand, is shown in Figure 91. For that figure, multiple PVPP sizes considering its nominal power were studied, as shown on the horizontal axis, while varying the number of inverters and maintaining the number of PV panels per inverter. A pure C++ implementation was developed to compare with the implementation of the C++ CUDA solution, which takes advantage of heterogeneous architectures. It can be inferred that the C++ version only outperforms the CUDA kernel when dealing with a small number of

devices. The CUDA version achieves a speedup of up to 14.3 when simulating the 350 MW PVPP using the serial-parallel configuration, while the rest of the layouts achieve a speedup of between 4.3 and 5.3. The main bottleneck is the lack of memory coalescing.

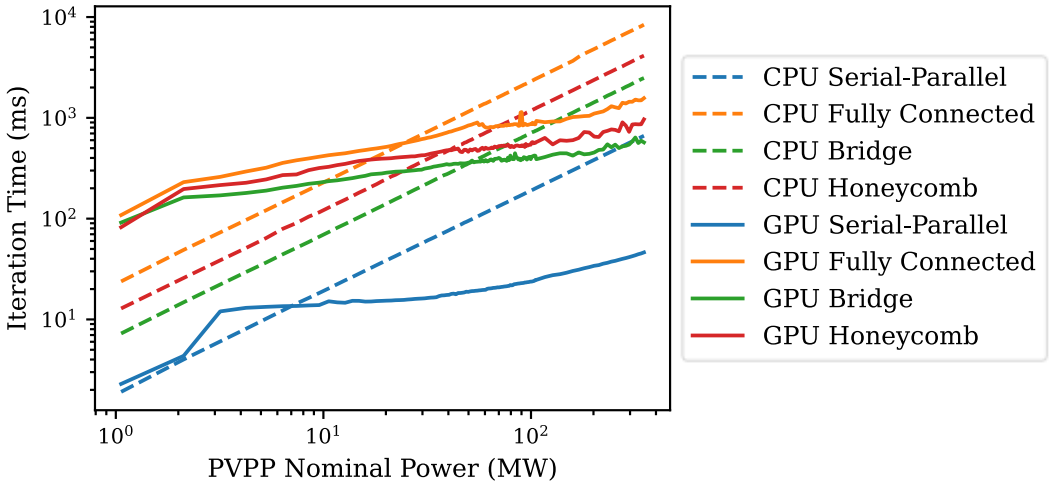
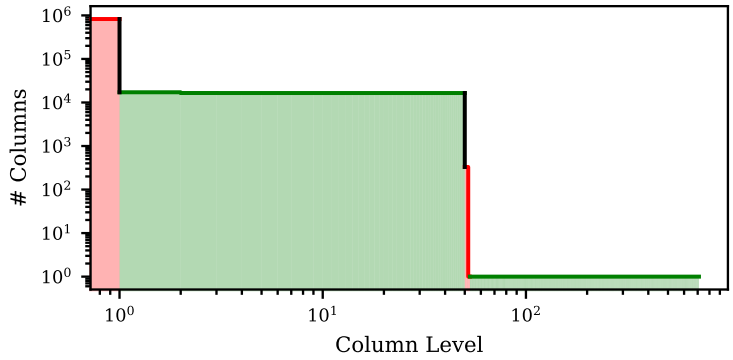
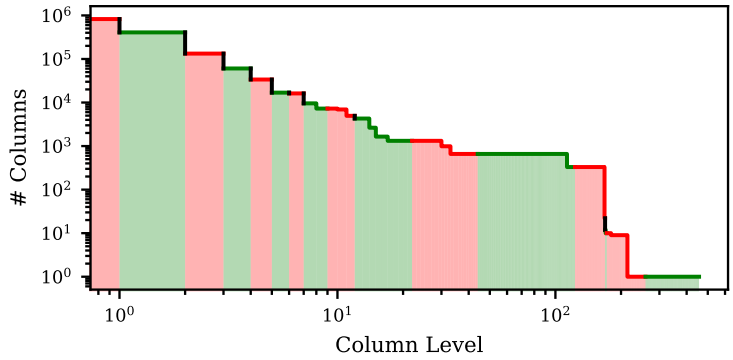


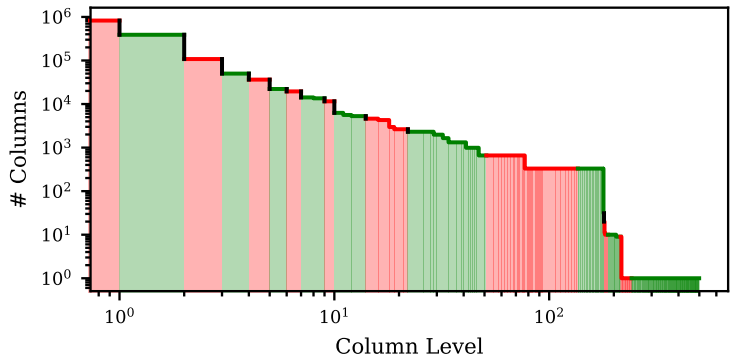
Figure 91. Computation time per PVPP simulation iteration.



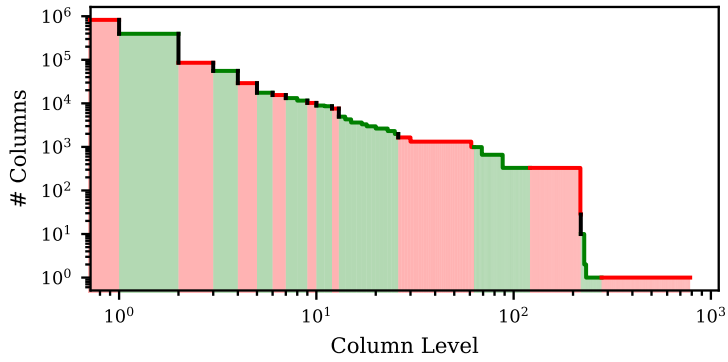
(a) Serial-Parallel Layout



(b) Bridge Layout



(c) Honeycomb Layout



(d) Totally Connected Layout

Figure 92. Column levels of a 350 MW PVPP under different configurations.

Chapter 8. Conclusions and Future Work

This chapter summarizes the findings and contributions of this Ph.D. thesis. Some future lines of work that have been discovered while elaborating this thesis will be presented as well. Finally, this chapter concludes with a list of publications and conferences that are related to the work of this thesis.

8.1. Conclusions

This thesis sought to answer the question: "*How can we implement the simulation of power Systems in GPUs in an efficient way?*" The introduction of renewable energy raises the running time of these simulations due to the large amount of energy generating devices, and that is an issue as mentioned in Chapter 2. To solve this issue, these power simulations can be run on heterogeneous architectures, such as graphics cards. The use of heterogeneous architectures was explored, and new methods of solving these power systems in graphics cards were proposed, implemented, and tested. It was studied the main types of renewable energy power plants that feature many individual generators: wind farms and photovoltaic power plants (PVPPs).

The conclusions of this thesis can be summarized as follows:

1. *Regarding the objective: "Parallel simulation of wind turbine as standalone elements."*

Renewable energy is as clean as difficult to manage. The energy sources are fickle and unpredictable for short timeframes. For that reason, the power system grid operators and power plant managers and owners need to know in advance the state of the power plants in order to ensure its profitability or the stability of the general power system. Running these simulations efficiently required implementing a wind farm power system simulator from scratch. There are several types of models of these kinds of machines, and the more detailed the model, the better the results.

Simulating wind farms requires first to lay out the mathematical models of the wind turbines. This thesis considered wind turbines equipped with DFIGs. The DFIG models were analyzed in both the 5^{th} and 3^{rd} order, where the latter is a simplification of the former that reduces numerical oscillations. The wind turbines were simulated considering a two-mass drive train and disregarding the pitch control. The controller that regulates the RSC was designed around a PI regulator, and the GSC was considered lossless with its output modelled as a low pass filter that follows the exchanged power of the RSC.

It was shown that the wind turbine models were highly nonlinear. Not only were there nonlinear, but their equations offered little parallelism opportunities. For that reason, the wind turbine models were linearized into a state-space system. The linearization of the system enables efficient parallelization thanks to coalescent transfer of memory.

After fully developing the mathematical model of the wind turbine, it was parallelized. One thread processes one row of the matrix for maximum performance. The results show that the output of the linearized models, despite slight differences for values which are far off the equilibrium point, capture the dynamics of the wind turbine, i.e., show the same type of transients and the time to clear off faults were shown to be similar.

2. *Regarding the objective: "Efficient simulation of an integrated wind farm efficiently on graphics cards."*

After the simulation of the standalone wind turbines was tested, it was laid out a full simulation. Parallelism is key to simulate large PSs with affordable computing cost and, as shown in the thesis, traditional parallelism solutions cannot extract enough parallelism. The whole PS was expressed in terms of the *dq0* transformation. This transformation was used as well to simplify the equations that govern the wind turbine model. The values of the transformation can be mapped into the real and imaginary parts of a complex number. The bibliography on complex-number simulation of power systems on graphics cards is scarce and thanks to the work performed in this thesis, it will shed some light on this matter.

Various power line models were considered, mainly the π-section line and the Bergeron line model. The Bergeron line model is traditionally used for long power lines of more than 15km. The equations of both line models were expounded, and it was concluded that the Bergeron line model imposes a timestep requirement which makes transient simulation impossible in both wind farms and PVPPs.

After obtaining the matrix that defines the grid of the wind farm, it was studied the way to parallelize the associated linear system. The traditional methods to solve such systems: the LU, LDL, and QR decomposition, can be parallelized; but they rely on column level parallelism, where some columns must wait other to be processed. Even with that outlook, it was found the optimal node order arrangement to perform the LU decomposition in serial mode.

Solving the grid of the wind farm required the use of network partition techniques. These techniques tear up the PS, solve each part in parallel, and combine the results to fully calculate the PS. Despite these techniques falling out of favor due to their limitations in favor of sparsity techniques, these techniques have been repurposed to simulate the grid of the wind farm efficiently and with parallelism. Each individual circuit was then solved with precomputed matrix inverses. Matrix inversion has been traditionally disregarded for its space requirements and the precision loss it entails. It was shown in this Ph.D. thesis that matrix inverses are a reliable tool when it is confined to small circuits.

The traditional way to execute code on graphics cards is to launch kernels in parallel, and if all parallel code must be synchronized, split the code into two kernels and launch first a kernel, and then the other. CUDA has recently released new synchronization primitives that open new venues to program performant kernels. Cooperative groups can synchronize a full kernel without actually finishing it. There is not much research performed on PS simulation that takes advantage of these synchronization techniques. This synchronization techniques were key to deliver a performant kernel.

3. *Regarding the objective: "Detailed solar photovoltaic power plant simulation."*

Photovoltaic Power Plants (PVPP) are enormous power plants that harness the power of the sun to generate electricity. These power plants span many square kilometers and feature many individual PV panels. The computational cost made the use of detailed models impossible and aggregated models were used instead.

This thesis sets up detailed PVPP models from the bottom-up. First, it defines the most basic unit of generation, the PV cell. And then, they are grouped into PV panels and PV arrays. To convert the DC power of the PV panels to the AC power of the global PS, many PV panel inverters must be used.

It was studied the dynamics of the PVPPs, and it was pointed out that the electromagnetic transients are irrelevant compared to atmospheric phenomena and temperature variance. This key insight enabled quasi-static simulation, which are simulations that disregard the transient state.

PV panels are modelled as current sources with diodes and resistors that represent their losses. The diodes are responsible for the nonlinearity of the model. The circuit was solved using Newton-Raphson iterations with a variable step and helper conductances that aided into the convergence of the PVPP.

The sparse matrix that resulted from the PVPP is exceptionally large and was solved using a parallel LU decomposer. The kernel proposed in this thesis introduces the concept of chains of columns, where each chain is made from columns that must be processed in series, but the chains themselves are executed in parallel. This concept reduces the number of kernel invocations and enabled greater performance gains.

8.2. Future Work

The main topic of this thesis is the applicability of GPUs to the simulation of PSs with high penetration of renewable energy. It has been proposed several kernels that speed-up the computation of both wind farms of PVPPs. The ideas fleshed out in the thesis are nothing but the starting point to further parallelize PSs. As a final thought, some new lines of research are the following:

- **Parallelization of nonlinear models:** the models of wind turbines simulated in this work were linearized in order to better harness the parallel potential of GPUs. The linearization process is just an approximation of the original system, and its output drifts from the correct value as the state goes further from the equilibrium point. Consequently, future research should be focused on parallelizing the nonlinear wind turbine models. Doing so would enhance the accuracy of the models but would pose challenges to the development of kernels capable of handling them.
- **Use of variable timestep integration algorithms:** the wind turbine models were integrated using Heun's method, whose timestep is fixed. New venues of research might explore the use of algorithms with variable timestep that would accelerate the solving of the wind farm. These variable timesteps would undoubtedly affect the discretization process used to discretize the grid.

- **Study of the stability of the 3rd order model with wind farms:** it was found that the 3rd order model was unstable when connected to a wind farm. The instability stems from the use of a unit delay, which were included implicitly in the algorithm used to compute the power system. These 3rd order model wind turbines gave accurate results when simulated in standalone mode, i.e., when no external grid was considered. The use of unit step delay prevents costly algebraic loops that would slow down the simulation. A new venue of research might be the adaptation of 3rd order wind turbine DFIG models that are appropriate to be simulated in wind farms with unit steps.
- **Inclusion of other types of wind turbines:** this thesis focused on wind turbines equipped with DFIGs, but there are many other types of wind turbines, such as the ones equipped with a Squirrel Cage Induction Generator (SCIG), or a Permanent Magnet Synchronous Generator (PMSG), just to name a few. The potential for parallelism could be studied as well.
- **Use of iterative solvers applied to renewable energy generation:** the linear systems presented in this thesis have been solved through matrix decomposition. The alternatives are the iterative solvers. Iterative solvers promise large speedups at the expense of sometimes not converging. The solution of the grid voltages of the wind farm and the PVPP might be carried out with these solvers. A new venue of research would be to implement them applied to renewable energy simulation and finding out the best preconditioners, which are matrices that modify the system so that it converges easier.
- **Parallelized distributed energy simulation:** this thesis has focused on the simulation of large power plants. However, there is a push to install household renewable energy sources, mainly PV. A case study worth exploring is the power system of a large city with plenty of distributed solar power generation. These new systems usually feature batteries to store excess energy for both small and large facilities. Its dynamic behavior can be studied with parallel architectures to find out their optimal configurations under a large number of use cases.

8.3. List of Publications

During the production of this thesis, several publications were published. These publications are listed below.

8.3.1. Published Journal Publications

- A. Jiménez-Ruiz, M. Cañas-Carretón, G. Fernández-Escribano, D. Ruiz-Coll, S. Martín-Martínez, and E. Gómez-Lázaro, *"Wind farm simulations based on a DFIG machine using parallel programming,"* Journal of Supercomputing, vol. 75, no. 3, pp. 1641–1653, Mar. 2019, doi: 10.1007/s11227-018-2723-9. (Q2 Quartile)

Abstract: *"New computational techniques for simulating a large array of wind turbines are highly needed to model modern electrical grid networks. In this paper, an implementation of a doubly fed induction generator wind turbine model solver is proposed. This solver will run on an NVIDIA graphic processing unit, and it will be coded using the compute unified device architecture (CUDA). The implementation will integrate a linear time-invariant system*

represented by state-space matrices. It has been implemented a CUDA kernel capable of simulating many wind turbines in parallel with different wind profiles and using different configurations. Strategies such as optimizing memory access and overlapping data transfers with the kernel were used to obtain the results. The CUDA implementation reaches an occupancy of 95%, while simulating 500 wind turbines where each unit is subject to a different wind profile or using different configuration parameters."

- A. Jiménez-Ruiz, G. Fernández-Escribano, M. Cañas-Carretón, and J. L. Sánchez, *"Using GPUs to simulate photovoltaic power plants: Special cases of performance loss,"* Journal of Computational Science, vol. 71, p. 102042, Jul. 2023, doi: 10.1016/j.jocs.2023.102042. (Q1 Quartile in "Computer Science (miscellaneous)" and Q2 in "Modelling and Simulation, and Theoretical Computer Science")

Abstract: *"The installed capacity of renewable energy is soaring, and Photovoltaic Power is a sizeable part of it. Photovoltaic panels are being installed on both buildings and in large Photovoltaic Power Plants (PVPPs), reaching up to 2 GW of installed capacity. We need to perform simulations of the state of PVPPs to ensure the stability of the power system and their profitability. Traditional simulations that take into consideration each panel can accurately predict the generated power but are too computationally taxing. To lower the computing cost of these simulations, a GPU-based solver that processes the dynamic model of a PVPP connected to a power system has been developed. It simulates the output of the whole photovoltaic power plant down to the photovoltaic panel level. The characteristics of the power system to be simulated were analyzed in order to parallelize the system. The two-diode model is used for the PV panels, and this model was linearized, resulting in a large sparse matrix with millions of columns that must be solved. Our proposal can simulate a model of the PVPP with both accuracy and low computational cost with no PV panel aggregation required. The system was solved using an adaptive step and helper conductances to aid convergence of the matrix. Parallel column factorization and column grouping is key to achieving a high performant simulation, which makes it feasible to simulate events such as cloud occlusion efficiently. We also studied how the wiring of PV panels affects the global power plant output when panels are shaded and/or broken. It was found that the aggregated model returns a higher generated output, which is key to predicting its output and its profitability. PVPP simulations on the GPU solver are up to 14.3 times faster than on the CPU."*

- A. Jiménez-Ruiz, G. Fernández-Escribano, M. Cañas-Carretón, and J. L. Sánchez, *"Using graphics processing units for the efficient dynamic simulation of wind farms,"* Computers and Electrical Engineering, vol. 112, p. 109018, Dec. 2023, doi: 10.1016/j.compeleceng.2023.109018. (Q1 Quartile in "Computer Science (miscellaneous)", Q1 in "Electrical and Electronic Engineering", and Q2 in "Control and Systems Engineering")

Abstract: *"Dynamic simulations are crucial for proper power system operation. However, these can be computationally expensive. This article introduces an efficient CUDA kernel for GPU acceleration, that achieves parallelization without the need to solve many different power systems at once. The proposal unifies generator and grid execution within a single CUDA kernel, ensuring accurate dynamic simulations of wind farms. A custom solution from scratch was developed, that integrates the power system network solver with connected devices. Wind turbines are modeled as state-space linear systems, while the grid is parallelized through subunit*

partitioning and small submatrix inverses to reduce computational costs. The resulting kernel extracts parallelism in grids with limited inherent parallelism, outperforming previous methods such as column-level parallelism. Simulations on an NVIDIA GeForce RTX 3090 show a 26.39x speedup with respect to the C++ version, and the cooperative groups' synchronization primitives provide optimal performance while simulating up to 2464 wind turbines."

8.3.2. Submitted Conference Proceedings

- A. Jiménez-Ruiz, M. Cañas-Carretón, G. Fernández-Escribano, D. Ruiz-Coll, S. Martín-Martínez, and E. Gómez-Lázaro, *"Simulation of the electromechanical model of a doubly-fed induction generator wind turbine using an NVIDIA GPU,"* in Proceedings of the 18th International Conference on Computational and Mathematical Methods in Science and Engineering, CMMSE 2018 9-13 July, 2018, Rota, Cádiz, Spain, Jul. 2020, p. 13.

Abstract: *"Parallel simulation techniques have become a great tool to overcome studies where a large demand of computational effort is needed. One of these cases is the integration of renewable energy in power systems, such as the wind energy. In this paper, it is proposed a highly parallel solver for the simulation of a large number of wind turbines equipped with a doubly-fed induction generator (DFIG), coded using the NVIDIA's CUDA architecture, based on NVIDIA graphic processing units (GPUs). The implementation involves the solution of a linearized time-invariant system in state space form, where both 5^{th} and 3^{rd} order representations of the generator model that is included in a wind turbine are considered. Previously to the CUDA approach, these models have also been implemented in MATLAB-Simulink and C++ to compare the computational cost. Results show that the CUDA implementation is up to 15 times faster than the other implementations."*

- A. Jiménez-Ruiz, M. Cañas-Carretón, G. Fernández-Escribano, J. L. Sánchez, and H. Kalva, *"Estudio y Análisis de la Simulación de Sistemas Eléctricos de Potencia con Generación de Energía Renovable Mediante Tarjetas Gráficas,"* XXXI Jornadas de Paralelismo (JP20/21) – Sarteco, Sep. 2021, Málaga, Spain

Abstract: *"La operación y planificación de los sistemas eléctricos requiere del uso de simulaciones dinámicas para asegurar el suministro en las condiciones requeridas. Sin embargo, estas simulaciones son computacionalmente muy exigentes debido a la complejidad y número de componentes de un sistema eléctrico. Dicha exigencia se ve incrementada notablemente con la inclusión de generación renovable debido a su carácter distribuido, ya que incrementa el nivel de complejidad y número de elementos a considerar en las simulaciones. Tradicionalmente se han empleado modelos agregados que permiten disminuir dicha exigencia a cambio de sacrificar nivel de detalle. Como alternativa al uso de modelos agregados, se ha recurrido al uso de aceleradores gráficos debido a su elevado poder computacional. La arquitectura interna de los aceleradores gráficos obliga a usar algoritmos optimizados para ellos, teniendo en cuenta que algunas operaciones pueden realizarse eficientemente en el acelerador gráfico, y otras no. El objetivo de este artículo es el estudio de los algoritmos usados en la simulación dinámica de sistemas eléctricos de potencia con presencia de energía renovable y extraer conclusiones sobre su idoneidad para ser ejecutados usando aceleradores gráficos."*

Chapter 9. Conclusiones y Trabajo Futuro

Este capítulo resume los hallazgos y contribuciones de esta tesis doctoral. Algunas líneas de trabajo futuras han sido descubiertas a lo largo de la elaboración de esta tesis, y éstas se presentarán a continuación. Por último, este capítulo concluye con una lista de publicaciones en revistas indexadas y publicaciones en congresos relacionadas con la tesis.

9.1. Conclusiones

Esta tesis buscó responder a la siguiente pregunta: *"¿Cómo podemos implementar la simulación de sistemas de potencia en GPU de manera eficiente?"*. La introducción de energías renovables incrementa el tiempo de computación de estas simulaciones a causa de la gran cantidad de elementos generadores de energía renovable, tal y como se menciona en el Capítulo 2. Para resolver este problema, se pueden simular los sistemas eléctricos en arquitecturas heterogéneas como las tarjetas gráficas. Se exploró el uso de éstas, y se propusieron nuevos métodos de resolver estos sistemas de potencia, se implementaron, y se validaron. Se estudiaron las principales plantas de generación de energía renovable con gran cantidad de generadores: parques eólicos y parques fotovoltaicos.

Las conclusiones de esta tesis se pueden resumir del siguiente modo:

1. *Respecto al objetivo: "Simulación en paralela de parque eólico como elementos individuales".*

La energía renovable es tan limpia como difícil de gestionar. Las fuentes de energía son inconsistentes e impredecibles a corto plazo. Por esa razón, los gestores de la red eléctrica y de las centrales de energía necesitan saber por adelantado el estado de las plantas generadores para asegurar tanto su rentabilidad como la estabilidad del sistema de potencia en general. Ejecutar estas simulaciones eficientemente requirió implementar un simulador de plantas eólicas desde cero. Existen varios modelos de estas máquinas y, cuanto más detallado sea el modelo, mejores serán los resultados.

Simular parques eólicos requiere en primer lugar detallar los modelos matemáticos de los aerogeneradores. Esta tesis consideró aerogeneradores equipados con DFIGs. Se analizaron los modelos de 5º y 3º orden de los DFIGs, donde el último es una simplificación del primero que reduce las oscilaciones numéricas. Se simularon los aerogeneradores considerando un sistema de transmisión de dos masas e ignorando el sistema de control de pitch. El controlador que regula el controlador del lado del rotor (RSC) fue diseñado como un controlador PI, mientras que el controlador del lado de la red (GSC) se diseñó como si transmitiera toda la potencia a la red sin pérdidas, cuya función de transferencia se asemeja a un filtro paso bajo.

Se vio que los modelos de los aerogeneradores eran profundamente no lineales. No sólo no eran lineales, sino que sus ecuaciones no podían paralelizarse fácilmente. Por esa razón se

linealizaron los modelos de aerogeneradores en un sistema de espacio de estados. La linealización de este sistema permite la paralelización eficiente gracias a la transferencia coalescente de memoria.

Tras desarrollar completamente el modelo matemático del aerogenerador se paralelizó éste. Cada hilo procesa una fila de la matriz para obtener un máximo rendimiento. Los resultados muestran que la salida de los modelos linealizados, salvo pequeñas diferencias de valores que se encuentran muy alejados del punto de equilibrio, capturan los transitorios del aerogenerador, es decir, muestran el mismo tipo de transitorios y el tiempo para despejar los cortocircuitos es también similar.

2. *Respecto al objetivo: "Simulación eficiente de aerogeneradores en tarjetas gráficas".*

Después de validar la simulación de aerogeneradores en solitario, se planteó una simulación completa de un parque eólico. El paralelismo es clave para simular grandes sistemas de potencia con un coste computacional moderado y, como se mostró en esta tesis, las soluciones tradicionales de paralelismo no pueden extraer suficiente paralelismo. Se planteó el sistema de potencia usando la transformación *dq0*. Esta transformación se usó tanto para simplificar el modelo de los aerogeneradores como para el sistema de potencia. Los valores de esta transformación se pueden representar como números complejos. La bibliografía sobre simulación con números complejos de sistemas de potencia en tarjetas gráficas es escasa, y gracias al trabajo realizado en esta tesis, se podrá arrojar luz sobre este tema.

Se consideraron varios modelos de líneas de transmisión, principalmente el modelo de línea de sección-π y el modelo de Bergeron. El modelo de línea de Bergeron se usa tradicionalmente para líneas de transmisión de más de 15km. Las ecuaciones de ambos modelos fueron desarrolladas, y se concluyó que el modelo de línea de Bergeron impone un tiempo de integración que impide la simulación de transitorios tanto para parques eólicos como para planta fotovoltaicas.

Una vez se obtuvo la matriz que define la red del parque eólico, se estudió la forma de paralelizar el sistema lineal resultante. Los métodos tradicionales para resolver estos sistemas, que son las descomposiciones LU, LDL, y QR, pueden paralelizarse, pero su paralelización descansa en la paralelización a nivel de columna, donde algunas columnas han de esperar a otras para ser procesadas. Incluso con esa limitación, se obtuvo cuál era el orden óptimo de nodos del sistema para realizar la descomposición LU en serie.

Resolver la red del parque eólico requirió el uso de técnicas de particionado de la red eléctrica. Estas técnicas seccionan el sistema eléctrico, procesan cada sección en paralelo, y recombinan los resultados para obtener la solución global del sistema de potencia. A pesar de que estas técnicas cayeron en desuso debido a sus limitaciones en favor de las técnicas que aprovechan el que la matriz del sistema de potencia sea dispersa, estas técnicas han sido rediseñadas para simular un parque eólico con paralelismo y eficiencia. Cada circuito individual se resolvió con matrices inversas precalculadas. La inversión de matrices ha sido tradicionalmente menospreciada por sus requerimientos de espacio y la falta de precisión en los cálculos que conlleva. Se ha visto en esta tesis doctoral que la inversión matricial es una técnica fiable cuando está limitada a la resolución de circuitos pequeños.

El método tradicional de ejecutar código en tarjetas gráficos consiste en lanzar *kernels* en paralelo, y si todo el código en paralelo ha de sincronizarse, separar el código en dos *kernels* y ejecutar primero uno, y luego el otro. CUDA ha estrenado recientemente nuevas primitivas de sincronización que permiten nuevas formas de desarrollar *kernels* de gran rendimiento. Los grupos cooperativos son capaces de sincronizar un *kernel* completo sin concluirlo. No existe mucha literatura científica realizada sobre simulación de sistemas eléctricos que aproveche estas técnicas de sincronización. Estas nuevas técnicas de sincronización fueron claves para alcanzar un gran rendimiento.

3. *Respecto al objetivo: "Simulación detallado de planta fotovoltaica".*

Las plantas fotovoltaicas son instalaciones de generación eléctrica colosales que generan electricidad a través de la energía solar. Estas plantas ocupan muchos kilómetros cuadrados y poseen muchos paneles fotovoltaicos individuales. El coste computacional supuso que el uso de modelos detallados se descartara, y que se usasen modelos agregados.

Por otro lado, en esta tesis se han implementado modelos detallados de plantas fotovoltaicas desde los elementos más básicos. Primero, se define la unidad más básica de generación, la celda solar; para después agruparlas en paneles y arreglos fotovoltaicos. Para convertir la corriente continua de los paneles fotovoltaicos en corriente alterna, muchos inversores han de usarse.

Se estudiaron las dinámicas de paneles fotovoltaicos y se comprobó que los transitorios electromagnéticos son irrelevantes en comparación con los fenómenos atmosféricos y la variación de temperatura. Este hecho es clave, y permite la simulación cuasi-estática, que son simulaciones que ignoran los estados transitorios.

Se han modelado los paneles fotovoltaicos como fuentes de corriente con diodos y resistencias que representan las pérdidas. Los diodos son responsables de la no linealidad del modelo. El circuito fue entonces resuelto usando iteraciones del método de Newton-Raphson con un paso de integración variable y conductancias de apoyo que ayudaron a la convergencia del panel.

La matriz dispersa que identifica a la planta fotovoltaica es muy grande y se resolvió usando una descomposición LU. El kernel propuesto en esta tesis introduce el concepto de las cadenas de columnas, donde cada cadena se compone de columnas que han de procesarse en serie, pero donde las cadenas se pueden ejecutar en paralelo. Este concepto reduce las invocaciones a kernel y permite alcanzar un mayor ahorro en el coste computacional.

9.2. Trabajos Futuros

El principal tema de esta tesis es la aplicabilidad de las GPUs a la simulación de sistemas de potencia con gran penetración de energías renovables. Se han propuesto varios *kernels* que aceleran la computación tanto de parques eólicos como de plantas fotovoltaicas. Las ideas que se han detallado tesis no son más que el punto de partida para continuar la paralelización de sistemas potencia. A modo de sugerencia, se detallan posibles líneas de investigación.

- **Paralelización de modelos no lineales:** los modelos de aerogeneradores simulados en este trabajo fueron linealizados para aprovechar la arquitectura paralela de las GPUs. El proceso de linealización es solo una aproximación del sistema original y su salida se desvía del valor correcto cuando el estado se desvía cada vez más del punto de equilibrio. Por consiguiente, las nuevas investigaciones deben enfocarse en paralelizar modelos no lineales de aerogeneradores. Hacerlo mejoraría la precisión de los modelos, aunque plantearía nuevos retos a la creación de kernels capaces de integrarlos.
- **Uso de algoritmos de paso variable:** los aerogeneradores se integraron con el método de Heun, cuyo paso de integración es fijo. Nuevas líneas de investigación como el uso de algoritmos de paso variable permitirán acelerar la integración de un parque eólico. Estos pasos de integración variables afectarían sin duda al proceso de integración usado para la red eléctrica.
- **Estudio de la estabilidad del modelo de tercer orden para parques eólicos:** se vio que el modelo de 3º orden es inestable cuando se integra un parque eólico en su conjunto. La inestabilidad de éste proviene del uso de un retraso unitario de un paso de integración implícito en el algoritmo usado para computar el sistema de potencia. Estos modelos de 3º orden devuelven resultados precisos al ser simulados de forma individual, es decir, si no se considera que hay una red eléctrica externa. El uso del retraso unitario evita bucles algebraicos computacionalmente costosos que ralentizarían la simulación. Una nueva línea de investigación podría ser la adopción de modelos de aerogeneradores de 3º orden adecuados para ser simulados en parques eólicos con pasos unitarios.
- **Inclusión de otros tipos de aerogeneradores:** esta tesis está enfocada en aerogeneradores equipados con DFIGs, pero hay muchos otros tipos de aerogeneradores, como los equipados con un generador de jaula de ardilla (SCIG), o generador síncrono de imanes permanentes (PMSG), sólo por nombrar unos pocos. Su potencial para el paralelismo también puede ser estudiado.
- **Uso de solucionadores iterativos aplicados a la generación con energías renovables:** los sistemas lineales presentados en la tesis han sido resueltos gracias a descomposiciones matriciales. Las alternativas son los solucionadores iterativos. Los solucionadores iterativos prometen grandes mejoras del rendimiento a costa de su posible no convergencia. La resolución de los voltajes del parque eólico y la planta fotovoltaica podrían realizarse con estos solucionadores. Una nueva línea de investigación podría ser implementarlos aplicados a las energías renovables y encontrar los mejores precondicionadores. Estos precondicionadores son matrices que modifican el sistema para que converja más rápido.
- **Simulación de sistemas de energía distribuida:** esta tesis se ha enfocado en la simulación de grandes plantas fotovoltaicas. No obstante, hay un gran empuje a la instalación de generadores de energía a nivel doméstico, principalmente energía fotovoltaica. Sería fructífero explorar el sistema de potencia de una gran ciudad con gran cantidad de energía solar distribuida. Normalmente forman parte de estos nuevos sistemas las baterías para almacenar la energía sobrante tanto a pequeña como a gran escala. El comportamiento dinámico de estos se puede estudiar con arquitecturas paralelas para descubrir su configuración óptima bajo un gran número de casos de estudio.

9.3. Lista de Publicaciones

Durante la elaboración de esta tesis, se presentaron varios trabajos relacionados con el tema de ésta. Estas publicaciones se listan a continuación.

9.3.1. Artículos de Revista Publicados

- A. Jiménez-Ruiz, M. Cañas-Carretón, G. Fernández-Escribano, D. Ruiz-Coll, S. Martín-Martínez, and E. Gómez-Lázaro, *"Wind farm simulations based on a DFIG machine using parallel programming,"* Journal of Supercomputing, vol. 75, no. 3, pp. 1641–1653, Mar. 2019, doi: 10.1007/s11227-018-2723-9. (Cuartil Q2)

Resumen: *"New computational techniques for simulating a large array of wind turbines are highly needed to model modern electrical grid networks. In this paper, an implementation of a doubly fed induction generator wind turbine model solver is proposed. This solver will run on an NVIDIA graphic processing unit, and it will be coded using the compute unified device architecture (CUDA). The implementation will integrate a linear time-invariant system represented by state-space matrices. It has been implemented a CUDA kernel capable of simulating many wind turbines in parallel with different wind profiles and using different configurations. Strategies such as optimizing memory access and overlapping data transfers with the kernel were used to obtain the results. The CUDA implementation reaches an occupancy of 95%, while simulating 500 wind turbines where each unit is subject to a different wind profile or using different configuration parameters."*

- A. Jiménez-Ruiz, G. Fernández-Escribano, M. Cañas-Carretón, and J. L. Sánchez, *"Using GPUs to simulate photovoltaic power plants: Special cases of performance loss,"* Journal of Computational Science, vol. 71, p. 102042, Jul. 2023, doi: 10.1016/j.jocs.2023.102042. (Cuartil Q1 en "Computer Science (miscellaneous)" y Cuartil Q2 en "Modelling and Simulation, and Theoretical Computer Science")

Resumen: *"The installed capacity of renewable energy is soaring, and Photovoltaic Power is a sizeable part of it. Photovoltaic panels are being installed on both buildings and in large Photovoltaic Power Plants (PVPPs), reaching up to 2 GW of installed capacity. We need to perform simulations of the state of PVPPs to ensure the stability of the power system and their profitability. Traditional simulations that take into consideration each panel can accurately predict the generated power but are too computationally taxing. To lower the computing cost of these simulations, a GPU-based solver that processes the dynamic model of a PVPP connected to a power system has been developed. It simulates the output of the whole photovoltaic power plant down to the photovoltaic panel level. The characteristics of the power system to be simulated were analyzed in order to parallelize the system. The two-diode model is used for the PV panels, and this model was linearized, resulting in a large sparse matrix with millions of columns that must be solved. Our proposal can simulate a model of the PVPP with both accuracy and low computational cost with no PV panel aggregation required. The system was solved using an adaptive step and helper conductances to aid convergence of the matrix. Parallel column factorization and column grouping is key to achieving a high performant simulation, which makes it feasible to simulate events such as cloud occlusion efficiently. We also studied how the wiring of PV panels affects the global power plant output when panels are shaded and/or broken. It was*

found that the aggregated model returns a higher generated output, which is key to predicting its output and its profitability. PVPP simulations on the GPU solver are up to 14.3 times faster than on the CPU."

- A. Jiménez-Ruiz, G. Fernández-Escribano, M. Cañas-Carretón, and J. L. Sánchez, *"Using graphics processing units for the efficient dynamic simulation of wind farms,"* Computers and Electrical Engineering, vol. 112, p. 109018, Dec. 2023, doi: 10.1016/j.compeleceng.2023.109018. (Cuartil Q1 en "Computer Science (miscellaneous)", Cuartil Q1 en "Electrical and Electronic Engineering" y Cuartil Q2 en "Control and Systems Engineering")

Resumen: *"Dynamic simulations are crucial for proper power system operation. However, these can be computationally expensive. This article introduces an efficient CUDA kernel for GPU acceleration, that achieves parallelization without the need to solve many different power systems at once. The proposal unifies generator and grid execution within a single CUDA kernel, ensuring accurate dynamic simulations of wind farms. A custom solution from scratch was developed, that integrates the power system network solver with connected devices. Wind turbines are modeled as state-space linear systems, while the grid is parallelized through subunit partitioning and small submatrix inverses to reduce computational costs. The resulting kernel extracts parallelism in grids with limited inherent parallelism, outperforming previous methods such as column-level parallelism. Simulations on an NVIDIA GeForce RTX 3090 show a 26.39x speedup with respect to the C++ version, and the cooperative groups' synchronization primitives provide optimal performance while simulating up to 2464 wind turbines."*

9.3.2. Actas de Congreso Presentadas

- A. Jiménez-Ruiz, M. Cañas-Carretón, G. Fernández-Escribano, D. Ruiz-Coll, S. Martín-Martínez, and E. Gómez-Lázaro, *"Simulation of the electromechanical model of a doubly-fed induction generator wind turbine using an NVIDIA GPU,"* in Proceedings of the 18th International Conference on Computational and Mathematical Methods in Science and Engineering, CMMSE 2018 9-13 July, 2018, Rota, Cádiz, Spain, Jul. 2020, p. 13.

Resumen: *"Parallel simulation techniques have become a great tool to overcome studies where a large demand of computational effort is needed. One of these cases is the integration of renewable energy in power systems, such as the wind energy. In this paper, it is proposed a highly parallel solver for the simulation of a large number of wind turbines equipped with a doubly-fed induction generator (DFIG), coded using the NVIDIA's CUDA architecture, based on NVIDIA graphic processing units (GPUs). The implementation involves the solution of a linearized time-invariant system in state space form, where both 5^{th} and 3^{rd} order representations of the generator model that is included in a wind turbine are considered. Previously to the CUDA approach, these models have also been implemented in MATLAB-Simulink and C++ to compare the computational cost. Results show that the CUDA implementation is up to 15 times faster than the other implementations."*

- A. Jiménez-Ruiz, M. Cañas-Carretón, G. Fernández-Escribano, J. L. Sánchez, and H. Kalva, *"Estudio y Análisis de la Simulación de Sistemas Eléctricos de Potencia con*

Generación de Energía Renovable Mediante Tarjetas Gráficas," XXXI Jornadas de Paralelismo (JP20/21) – Sarteco, Sep. 2021, Málaga, Spain

Resumen: *"La operación y planificación de los sistemas eléctricos requiere del uso de simulaciones dinámicas para asegurar el suministro en las condiciones requeridas. Sin embargo, estas simulaciones son computacionalmente muy exigentes debido a la complejidad y número de componentes de un sistema eléctrico. Dicha exigencia se ve incrementada notablemente con la inclusión de generación renovable debido a su carácter distribuido, ya que incrementa el nivel de complejidad y número de elementos a considerar en las simulaciones. Tradicionalmente se han empleado modelos agregados que permiten disminuir dicha exigencia a cambio de sacrificar nivel de detalle. Como alternativa al uso de modelos agregados, se ha recurrido al uso de aceleradores gráficos debido a su elevado poder computacional. La arquitectura interna de los aceleradores gráficos obliga a usar algoritmos optimizados para ellos, teniendo en cuenta que algunas operaciones pueden realizarse eficientemente en el acelerador gráfico, y otras no. El objetivo de este artículo es el estudio de los algoritmos usados en la simulación dinámica de sistemas eléctricos de potencia con presencia de energía renovable y extraer conclusiones sobre su idoneidad para ser ejecutados usando aceleradores gráficos."*

References

[1] J. Laherrère, C. A. S. Hall, and R. Bentley, “How much oil remains for the world to produce? Comparing assessment methods, and separating fact from fiction,” *Current Research in Environmental Sustainability*, vol. 4, p. 100174, Jan. 2022, doi: 10.1016/j.crsust.2022.100174.

[2] IEA, “World Energy Outlook 2022,” Paris, 2022. Accessed: Dec. 19, 2022. [Online]. Available: https://www.iea.org/reports/world-energy-outlook-2022

[3] L. Reijnders, “Conditions for the sustainability of biomass based fuel use,” *Energy Policy*, vol. 34, no. 7, pp. 863–876, May 2006, doi: 10.1016/j.enpol.2004.09.001.

[4] C. E. Marín and R. G. Marín, “Agua y energía: producción hidroeléctrica en España,” *Investigaciones Geográficas*, no. 51, Art. no. 51, Apr. 2010, doi: 10.14198/INGEO2010.51.05.

[5] M. R. Chakraborty, S. Dawn, P. K. Saha, J. B. Basu, and T. S. Ustun, “A Comparative Review on Energy Storage Systems and Their Application in Deregulated Systems,” *Batteries*, vol. 8, no. 9, Art. no. 9, Sep. 2022, doi: 10.3390/batteries8090124.

[6] G. E. Moore, “Cramming more components onto integrated circuits,” *Electronics*, vol. 38, no. 8, p. 114, 1965, doi: 10.1109/N-SSC.2006.4785860.

[7] Intel, *Intel® 64 and IA-32 Architectures Software Developer’s Manual*, vol. 1, 2A, 2B, 2C, 2D, 3A, 3B, 3C, 3D and 4. Santa Clara, USA, 2022.

[8] Q.-C. Zhong and G. Weiss, “Synchronverters: Inverters That Mimic Synchronous Generators,” *IEEE Transactions on Industrial Electronics*, vol. 58, no. 4, pp. 1259–1267, Apr. 2011, doi: 10.1109/TIE.2010.2048839.

[9] P. Kundur, *Power System Stability and Control*. New York: MHP, 1994.

[10] P. Wang, Z. Zhang, Q. Huang, N. Wang, X. Zhang, and W.-J. Lee, “Improved Wind Farm Aggregated Modeling Method for Large-Scale Power System Stability Studies,” *IEEE Transactions on Power Systems*, vol. 33, no. 6, pp. 6332–6342, Nov. 2018, doi: 10.1109/TPWRS.2018.2828411.

[11] S. Lumbreras and A. Ramos, “Offshore wind farm electrical design: a review,” *Wind Energy*, vol. 16, no. 3, pp. 459–473, 2013, doi: 10.1002/we.1498.

[12] NVIDIA Corporation, *CUDA C++ Programming Guide*, 12.1. NVIDIA Corporation, 2023. Accessed: May 01, 2021. [Online]. Available: http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html

[13] The Khronos Group, *OpenGL 4.6 Core Profile*, 2.0., vol. 1. Beaverton, Oregon, USA, 2022.

[14] H. Xu and C. P. Wang, “A Review and Development of 3-D Accelerator Technology for Games,” in *2009 Second International Symposium on Intelligent Information Technology and Security Informatics*, Jan. 2009, pp. 59–63. doi: 10.1109/IITSI.2009.20.

[15] “OpenCL - The open standard for parallel programming of heterogeneous systems,” The Khronos Group. Accessed: Jan. 16, 2020. [Online]. Available: https://www.khronos.org/opencl/

[16] K. Karimi, N. G. Dickson, and F. Hamze, “A Performance Comparison of CUDA and OpenCL.” arXiv, May 16, 2011. doi: 10.48550/arXiv.1005.2581.

[17] J. Fang, A. L. Varbanescu, and H. Sips, “A Comprehensive Performance Comparison of CUDA and OpenCL,” in *2011 International Conference on Parallel Processing*, Sep. 2011, pp. 216–225. doi: 10.1109/ICPP.2011.45.

[18] NVIDIA Corporation, “NVIDIA Ampere GA102 GPU Architecture Second-Generation RTX,” Santa Clara, USA, V2.0, 2021. Accessed: Jan. 01, 2023. [Online]. Available: https://www.nvidia.com/content/PDF/nvidia-ampere-ga-102-gpu-architecture-whitepaper-v2.pdf

[19] A. Li *et al.*, “Evaluating Modern GPU Interconnect: PCIe, NVLink, NV-SLI, NVSwitch and GPUDirect,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 31, no. 1, pp. 94–110, Jan. 2020, doi: 10.1109/TPDS.2019.2928289.

[20] NVIDIA, “NVIDIA GeForce GTX 750 Ti WhitePaper,” NVIDIA, Santa Clara, Estados Unidos, WhitePaper NVIDIA GeForce GTX 750 Ti WhitePaper, 2014. Accessed: May 31,

2023. [Online]. Available: https://www.nvidia.com/content/dam/en-zz/Solutions/design-visualization/technologies/turing-architecture/NVIDIA-Turing-Architecture-Whitepaper.pdf
[21] *NVIDIA Nsight Visual Studio Edition Documentation*. 2023. Accessed: Jun. 05, 2023. [Online]. Available: https://docs.nvidia.com/nsight-visual-studio-edition/
[22] NVIDIA Corporation, "Parallel Thread Execution ISA Version 6.4," concept, Aug. 2019. Accessed: May 31, 2023. [Online]. Available: http://docs.nvidia.com/cuda/parallel-thread-execution/index.html
[23] "CUDA Binary Utilities," Apr. 2023. Accessed: May 31, 2023. [Online]. Available: https://docs.nvidia.com/cuda/cuda-binary-utilities/
[24] B. Wang, C. Zhang, F. Wang, and J. Feng, "A Synchronization Mechanism between CUDA Blocks for GPU," in *Proceedings of the 2017 2nd International Conference on Control, Automation and Artificial Intelligence (CAAI 2017)*, Sanya, China: Atlantis Press, 2017. doi: 10.2991/caai-17.2017.56.
[25] L. Zhang, M. Wahib, H. Zhang, and S. Matsuoka, "A Study of Single and Multi-device Synchronization Methods in Nvidia GPUs," in *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, May 2020, pp. 483–493. doi: 10.1109/IPDPS47924.2020.00057.
[26] S. Xiao and W. Feng, "Inter-block GPU communication via fast barrier synchronization," in *2010 IEEE International Symposium on Parallel Distributed Processing (IPDPS)*, Apr. 2010, pp. 1–12. doi: 10.1109/IPDPS.2010.5470477.
[27] A. S. Anand, A. Srivastava, and R. K. Shyamasundar, "A deadlock-free lock-based synchronization for GPUs," *Concurrency and Computation: Practice and Experience*, vol. 31, no. 7, p. e4991, 2019, doi: https://doi.org/10.1002/cpe.4991.
[28] V. Volkov, "Understanding Latency Hiding on GPUs," UC Berkeley, 2016. Accessed: Jun. 01, 2023. [Online]. Available: https://escholarship.org/uc/item/1wb7f3h4
[29] K. P. Schneider *et al.*, "Analytic Considerations and Design Basis for the IEEE Distribution Test Feeders," *IEEE Transactions on Power Systems*, vol. 33, no. 3, pp. 3181–3188, May 2018, doi: 10.1109/TPWRS.2017.2760011.
[30] M. Połomski and B. Baron, "Optimal Power Flow by Interior Point and Non Interior Point Modern Optimization Algorithms," *Acta Energetica Power Engineering Quarterly*, vol. 1/2013, pp. 132–139, Jul. 2013, doi: 10.12736/issn.2300-3022.2013112.
[31] J. Grainger and W. Stevenson, *Power Systems Analysis*, 2nd ed. U.S.: McGraw-Hill Education / Asia, 2016.
[32] V. Ajjarapu and C. Christy, "The continuation power flow: a tool for steady state voltage stability analysis," *IEEE Transactions on Power Systems*, vol. 7, no. 1, pp. 416–423, Feb. 1992, doi: 10.1109/59.141737.
[33] I. Boldea, *Synchronous Generators*, 2nd ed. Boca Raton, FL: CRC Press, 2015.
[34] *Modern Control Theory*. Berlin/Heidelberg: Springer-Verlag, 2005. doi: 10.1007/3-540-28087-1.
[35] H. W. Dommel and W. S. Meyer, "Computation of electromagnetic transients," *Proceedings of the IEEE*, vol. 62, no. 7, pp. 983–993, Jul. 1974, doi: 10.1109/PROC.1974.9550.
[36] J. C. Butcher, *Numerical Methods for Ordinary Differential Equations*. John Wiley & Sons, 2016.
[37] J. Zhao *et al.*, "Power System Dynamic State Estimation: Motivations, Definitions, Methodologies, and Future Work," *IEEE Transactions on Power Systems*, vol. 34, no. 4, pp. 3188–3198, Jul. 2019, doi: 10.1109/TPWRS.2019.2894769.
[38] Ministerio de Energía, Turismo y Agenda Digital, *Resolución de 1 de febrero de 2018, de la Secretaría de Estado de Energía, por la que se aprueba el procedimiento de operación 12.2 "Instalaciones conectadas a la red de transporte y equipo generador: requisitos mínimos de diseño, equipamiento, funcionamiento, puesta en servicio y seguridad" de los sistemas eléctricos no peninsulares*, vol. BOE-A-2018-2198. 2018, pp. 18835–18875. Accessed: Jun. 04, 2023. [Online]. Available: https://www.boe.es/eli/es/res/2018/02/01/(3)
[39] J. Rodriguez, Jih-Sheng Lai, and Fang Zheng Peng, "Multilevel inverters: a survey of topologies, controls, and applications," *IEEE Transactions on Industrial Electronics*, vol.

49, no. 4, pp. 724–738, Aug. 2002, doi: 10.1109/TIE.2002.801052.
[40] R. R. Booth, “Power System Simulation Model Based on Probability Analysis,” *IEEE Transactions on Power Apparatus and Systems*, vol. PAS-91, no. 1, pp. 62–69, Jan. 1972, doi: 10.1109/TPAS.1972.293291.
[41] R. Allan and R. Billinton, “Probabilistic assessment of power systems,” *Proceedings of the IEEE*, vol. 88, no. 2, pp. 140–162, Feb. 2000, doi: 10.1109/5.823995.
[42] K. J. Timko, A. Bose, and P. M. Anderson, “Monte Carlo Simulation of Power System Stability,” *IEEE Transactions on Power Apparatus and Systems*, vol. PAS-102, no. 10, pp. 3453–3459, Oct. 1983, doi: 10.1109/TPAS.1983.317843.
[43] Y. Song, Y. Chen, S. Huang, Y. Xu, Z. Yu, and W. Xue, “Efficient GPU-Based Electromagnetic Transient Simulation for Power Systems With Thread-Oriented Transformation and Automatic Code Generation,” *IEEE Access*, vol. 6, pp. 25724–25736, 2018, doi: 10.1109/ACCESS.2018.2833506.
[44] M. Herlihy and N. Shavit, *The Art of Multiprocessor Programming, Revised Reprint*. Elsevier, 2012.
[45] H. H. B. Sørensen, “High-Performance Matrix-Vector Multiplication on the GPU,” in *Euro-Par 2011: Parallel Processing Workshops*, M. Alexander, P. D’Ambra, A. Belloum, G. Bosilca, M. Cannataro, M. Danelutto, B. Di Martino, M. Gerndt, E. Jeannot, R. Namyst, J. Roman, S. L. Scott, J. L. Traff, G. Vallée, and J. Weidendorfer, Eds., in Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2012, pp. 377–386. doi: 10.1007/978-3-642-29737-3_42.
[46] T. Dong, A. Haidar, P. Luszczek, S. Tomov, A. Abdelfattah, and J. Dongarra, “MAGMA Batched: A Batched BLAS Approach for Small Matrix Factorizations and Applications on GPUs,” 2016. Accessed: Jun. 05, 2023. [Online]. Available: https://www.semanticscholar.org/paper/MAGMA-Batched%3A-A-Batched-BLAS-Approach-for-Small-on-Dong-Haidar/c7e102402afabdc9a72c74009bf7bcc37c2d6358
[47] G. He, J. Gao, and J. Wang, “Efficient dense matrix-vector multiplication on GPU,” *Concurrency and Computation: Practice and Experience*, vol. 30, no. 19, p. e4705, 2018, doi: 10.1002/cpe.4705.
[48] S. Filippone, V. Cardellini, D. Barbieri, and A. Fanfarillo, “Sparse Matrix-Vector Multiplication on GPGPUs,” *ACM Trans. Math. Softw.*, vol. 43, no. 4, p. 30:1-30:49, Jan. 2017, doi: 10.1145/3017994.
[49] M. Yang, C. Sun, Z. Li, and D. Cao, “An improved sparse matrix-vector multiplication kernel for solving modified equation in large scale power flow calculation on CUDA,” in *Proceedings of The 7th International Power Electronics and Motion Control Conference*, Jun. 2012, pp. 2028–2031. doi: 10.1109/IPEMC.2012.6259153.
[50] W. Cao, L. Yao, Z. Li, Y. Wang, and Z. Wang, “Implementing Sparse Matrix-Vector multiplication using CUDA based on a hybrid sparse matrix format,” in *2010 International Conference on Computer Application and System Modeling (ICCASM 2010)*, Oct. 2010, pp. V11-161-V11-165. doi: 10.1109/ICCASM.2010.5623237.
[51] Z. Wang, X. Xu, W. Zhao, Y. Zhang, and S. He, “Optimizing sparse matrix-vector multiplication on CUDA,” in *2010 2nd International Conference on Education Technology and Computer*, Jun. 2010, pp. V4-109-V4-113. doi: 10.1109/ICETC.2010.5529724.
[52] Y. Nagasaka, A. Nukada, and S. Matsuoka, “Adaptive Multi-level Blocking Optimization for Sparse Matrix Vector Multiplication on GPU,” *Procedia Computer Science*, vol. 80, pp. 131–142, Jan. 2016, doi: 10.1016/j.procs.2016.05.304.
[53] W. H. Press, B. P. Flannery, S. A. Teukolsky, and W. T. Vetterling, *Numerical Recipes in C: The Art of Scientific Computing, Second Edition*, 2nd edition. Cambridge ; New York: Cambridge University Press, 1992.
[54] Å. Björck, “Direct Methods for Linear Systems,” in *Numerical Methods in Matrix Computations*, Å. Björck, Ed., in Texts in Applied Mathematics. , Cham: Springer International Publishing, 2015, pp. 1–209. doi: 10.1007/978-3-319-05089-8_1.
[55] A. Druinsky and S. Toledo, “How Accurate is inv(A)*b?,” *arXiv:1201.6035 [cs, math]*, Jan. 2012, Accessed: Apr. 11, 2021. [Online]. Available: http://arxiv.org/abs/1201.6035
[56] J. Xuebin, C. Yewang, F. Wentao, Z. Yong, and D. Jixiang, “Fast algorithm for parallel

solving inversion of large scale small matrices based on GPU," *J Supercomput*, May 2023, doi: 10.1007/s11227-023-05336-7.

[57] N. Tian, L. Guo, M. Ren, and C. Ai, "Implementing the Matrix Inversion by Gauss-Jordan Method with CUDA," in *Wireless Algorithms, Systems, and Applications*, Z. Cai, C. Wang, S. Cheng, H. Wang, and H. Gao, Eds., in Lecture Notes in Computer Science. Cham: Springer International Publishing, 2014, pp. 44–53. doi: 10.1007/978-3-319-07782-6_5.

[58] D. Lopez, M. Varalakshmi, and A. P. Kesarkar, "Embarrassingly Parallel GPU Based Matrix Inversion Algorithm for Big Climate Data Assimilation," *Int. J. Grid High Perform. Comput.*, vol. 10, no. 1, pp. 71–92, Jan. 2018, doi: 10.4018/IJGHPC.2018010105.

[59] F. Soleymani, "A Rapid Numerical Algorithm to Compute Matrix Inversion," *International Journal of Mathematics and Mathematical Sciences*, vol. 2012, p. e134653, Sep. 2012, doi: 10.1155/2012/134653.

[60] M. Baboulin, J. Dongarra, A. Rémy, S. Tomov, and I. Yamazaki, "Solving dense symmetric indefinite systems using GPUs," *Concurrency and Computation: Practice and Experience*, vol. 29, no. 9, p. e4055, 2017, doi: 10.1002/cpe.4055.

[61] J. Bunch and L. Kaufman, "Some Stable Methods for Calculating Inertia and Solving Symmetric Linear Systems," *Mathematics of Computation - Math. Comput.*, vol. 31, pp. 163–163, Jan. 1977, doi: 10.1090/S0025-5718-1977-0428694-0.

[62] J. O. Aasen, "On the reduction of a symmetric matrix to tridiagonal form," *BIT*, vol. 11, no. 3, pp. 233–242, Sep. 1971, doi: 10.1007/BF01931804.

[63] J. D. Hogg, E. Ovtchinnikov, and J. A. Scott, "A Sparse Symmetric Indefinite Direct Solver for GPU Architectures," *ACM Trans. Math. Softw.*, vol. 42, no. 1, p. 1:1-1:25, Jan. 2016, doi: 10.1145/2756548.

[64] X. Li, F. Li, and J. M. Clark, "Exploration of multifrontal method with GPU in power flow computation," in *2013 IEEE Power & Energy Society General Meeting*, Jul. 2013, pp. 1–5. doi: 10.1109/PESMG.2013.6673057.

[65] W.-K. Lee, R. Achar, and M. S. Nakhla, "Dynamic GPU Parallel Sparse LU Factorization for Fast Circuit Simulation," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 26, no. 11, pp. 2518–2529, Nov. 2018, doi: 10.1109/TVLSI.2018.2858014.

[66] G. Zhou *et al.*, "GPU-Based Batch LU-Factorization Solver for Concurrent Analysis of Massive Power Flows," *IEEE Transactions on Power Systems*, vol. 32, no. 6, pp. 4975–4977, Nov. 2017, doi: 10.1109/TPWRS.2017.2662322.

[67] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms, third edition*. Cambridge, Mass, 2009.

[68] N. Bell and M. Garland, "Efficient Sparse Matrix-Vector Multiplication on CUDA," Jan. 2009.

[69] G. Flegar, "Sparse Linear System Solvers on GPUs: Parallel Preconditioning, Workload Balancing and Communication Reduction," 2019.

[70] A. Mehrabi, D. Lee, N. Chatterjee, D. J. Sorin, B. C. Lee, and M. O'Connor, "Learning Sparse Matrix Row Permutations for Efficient SpMM on GPU Architectures," in *2021 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, Mar. 2021, pp. 48–58. doi: 10.1109/ISPASS51385.2021.00016.

[71] Y.-Y. Niu, H.-W. Tang, L.-C. Lee, and T. I. Tseng, "Simulation of flowfields induced by wind blades based on a parallelized low-speed flow solver," *Computers & Fluids*, vol. 45, no. 1, pp. 249–253, Jun. 2011, doi: 10.1016/j.compfluid.2010.11.007.

[72] J. Robledo, J. Leloux, E. Lorenzo, and C. A. Gueymard, "From video games to solar energy: 3D shading simulation for PV using GPU," *Solar Energy*, vol. 193, pp. 962–980, Nov. 2019, doi: 10.1016/j.solener.2019.09.041.

[73] L. Oberkirsch, D. Maldonado Quinto, P. Schwarzbözl, and B. Hoffschmidt, "GPU-based aim point optimization for solar tower power plants," *Solar Energy*, vol. 220, pp. 1089–1098, May 2021, doi: 10.1016/j.solener.2020.11.053.

[74] H. S. Jang, K. Y. Bae, H.-S. Park, and D. K. Sung, "Effect of aggregation for multi-site photovoltaic (PV) farms," in *2015 IEEE International Conference on Smart Grid Communications (SmartGridComm)*, Nov. 2015, pp. 623–628. doi: 10.1109/SmartGridComm.2015.7436370.

[75] K. Ben Kilani and M. Elleuch, “Simplified modelling of wind farms for voltage dip transients,” in *International Multi-Conference on Systems, Signals & Devices*, Mar. 2012, pp. 1–6. doi: 10.1109/SSD.2012.6198082.
[76] A. M. S. Al-bayati, F. Mancilla-David, and J. L. Domínguez-Garcíal, “Aggregated models of wind farms: Current methods and future trends,” in *2016 North American Power Symposium (NAPS)*, Sep. 2016, pp. 1–6. doi: 10.1109/NAPS.2016.7747954.
[77] B. Haut, F.-X. Bouchez, and F. Villella, “Improved Real-Time Computation Engine for a Dispatcher Training Center of the European Transmission Network,” in *2012 SC Companion: High Performance Computing, Networking Storage and Analysis*, Nov. 2012, pp. 333–340. doi: 10.1109/SC.Companion.2012.52.
[78] Z. Li, V. D. Donde, J.-C. Tournier, and F. Yang, “On limitations of traditional multi-core and potential of many-core processing architectures for sparse linear solvers used in large-scale power system applications,” in *2011 IEEE Power and Energy Society General Meeting*, Jul. 2011, pp. 1–8. doi: 10.1109/PES.2011.6039675.
[79] S. E. Papadakis and A. G. Bakrtzis, “A GPU accelerated PSO with application to Economic Dispatch problem,” in *2011 16th International Conference on Intelligent System Applications to Power Systems*, Sep. 2011, pp. 1–6. doi: 10.1109/ISAP.2011.6082162.
[80] Y. Song, Y. Chen, Z. Yu, S. Huang, and C. Shen, “CloudPSS: A high-performance power system simulator based on cloud computing,” *Energy Reports*, vol. 6, pp. 1611–1618, Dec. 2020, doi: 10.1016/j.egyr.2020.12.028.
[81] Z. Wang, S. Wende-von Berg, and M. Braun, “Fast parallel Newton–Raphson power flow solver for large number of system calculations with CPU and GPU,” *Sustainable Energy, Grids and Networks*, vol. 27, p. 100483, Sep. 2021, doi: 10.1016/j.segan.2021.100483.
[82] G. Kron, *Diakoptics: The piecewise solution of large-scale systems*. Macdonald & Co, 1963.
[83] B. D. Bonatto, M. L. Armstrong, J. R. Martí, and H. W. Dommel, “Current and voltage dependent sources modelling in MATE–multi-area Thévenin equivalent concept,” *Electric Power Systems Research*, vol. 138, pp. 138–145, Sep. 2016, doi: 10.1016/j.epsr.2016.03.011.
[84] Z. Zhou, “Massively Parallel Electromagnetic Transient Simulation of Large Power Systems,” University of Alberta, Alberta, Canada, 2018. doi: 10.7939/R3B56DK7X.
[85] J. Sun, S. Debnath, M. Saeedifard, and P. R. V. Marthi, “Real-Time Electromagnetic Transient Simulation of Multi-Terminal HVDC–AC Grids Based on GPU,” *IEEE Transactions on Industrial Electronics*, vol. 68, no. 8, pp. 7002–7011, Aug. 2021, doi: 10.1109/TIE.2020.3005059.
[86] D. Koester, S. Ranka, and G. Fox, “Parallel Direct Methods for Block-Diagonal-Bordered Sparse Matrices,” Nov. 1995.
[87] J. Kumar Debnath, W.-K. Fung, A. M. Gole, and S. Filizadeh, “Electromagnetic transient simulation of large-scale electrical power networks using graphics processing units,” in *2012 25th IEEE Canadian Conference on Electrical and Computer Engineering (CCECE)*, Apr. 2012, pp. 1–4. doi: 10.1109/CCECE.2012.6334886.
[88] Z. Zhou and V. Dinavahi, “Fine-Grained Network Decomposition for Massively Parallel Electromagnetic Transient Simulation of Large Power Systems,” *IEEE Power and Energy Technology Systems Journal*, vol. 4, no. 3, pp. 51–64, Sep. 2017, doi: 10.1109/JPETS.2017.2732360.
[89] N. Lin, S. Cao, and V. Dinavahi, “Massively Parallel Modeling of Battery Energy Storage Systems for AC/DC Grid High-Performance Transient Simulation,” *IEEE Transactions on Power Systems*, pp. 1–12, 2022, doi: 10.1109/TPWRS.2022.3196286.
[90] T. Cheng, N. Lin, and V. Dinavahi, “Hybrid Parallel-in-Time-and-Space Transient Stability Simulation of Large-Scale AC/DC Grids,” *IEEE Transactions on Power Systems*, vol. 37, no. 6, pp. 4709–4719, Nov. 2022, doi: 10.1109/TPWRS.2022.3153450.
[91] Y. Song, Y. Chen, Z. Yu, S. Huang, and C. Laijun, “A Fine-Grained Parallel EMTP Algorithm Compatible to Graphic Processing Units,” presented at the IEEE Power and Energy Society General Meeting, Jul. 2014. doi: 10.1109/PESGM.2014.6939325.
[92] R. Gnanavignesh and U. J. Shenoy, “GPU-Accelerated Sparse LU Factorization for Power System Simulation,” in *2019 IEEE PES Innovative Smart Grid Technologies Europe (ISGT-*

Europe), Sep. 2019, pp. 1–5. doi: 10.1109/ISGTEurope.2019.8905648.

[93] Y. Feng, C. Qin, Y. Zheng, J. Hua, and G. Zhou, "GPU-accelerated Power System Sensitivity Analysis," in *2021 International Conference on Green Energy, Computing and Sustainable Technology (GECOST)*, Jul. 2021, pp. 1–5. doi: 10.1109/GECOST52368.2021.9538670.

[94] S. Peng and S. X.-D. Tan, "GLU3.0: Fast GPU-based Parallel Sparse LU Factorization for Circuit Simulation," *IEEE Design Test*, vol. 37, no. 3, pp. 78–90, Jun. 2020, doi: 10.1109/MDAT.2020.2974910.

[95] N. Lin and V. Dinavahi, "Parallel High-Fidelity Electromagnetic Transient Simulation of Large-Scale Multi-Terminal DC Grids," *IEEE Power and Energy Technology Systems Journal*, vol. 6, no. 1, pp. 59–70, Mar. 2019, doi: 10.1109/JPETS.2018.2881483.

[96] I. Araújo, V. Tadaiesky, D. Cardoso, Y. Fukuyama, and Á. Santana, "Simultaneous parallel power flow calculations using hybrid CPU-GPU approach," *International Journal of Electrical Power & Energy Systems*, vol. 105, pp. 229–236, Feb. 2019, doi: 10.1016/j.ijepes.2018.08.033.

[97] N. Lin, S. Cao, and V. Dinavahi, "Adaptive Heterogeneous Transient Analysis of Wind Farm Integrated Comprehensive AC/DC Grids," *IEEE Transactions on Energy Conversion*, vol. 36, no. 3, pp. 2370–2379, Sep. 2021, doi: 10.1109/TEC.2020.3043307.

[98] Y. Song, Y. Chen, S. Huang, Y. Xu, Z. Yu, and J. R. Marti, "Fully GPU-based electromagnetic transient simulation considering large-scale control systems for system-level studies," *Transmission Distribution IET Generation*, vol. 11, no. 11, pp. 2840–2851, 2017, doi: 10.1049/iet-gtd.2016.2078.

[99] M. Marin, "GPU-enhanced power flow analysis," thesis, Perpignan, 2015. Accessed: Jan. 22, 2020. [Online]. Available: http://www.theses.fr/2015PERP0041

[100] J. Schiffer, D. Zonetti, R. Ortega, A. M. Stanković, T. Sezi, and J. Raisch, "A survey on modeling of microgrids—From fundamental physics to phasors and voltage sources," *Automatica*, vol. 74, pp. 135–150, Dec. 2016, doi: 10.1016/j.automatica.2016.07.036.

[101] N. Goudarzi and W. D. Zhu, "A review on the development of wind turbine generators across the world," *Int. J. Dynam. Control*, vol. 1, no. 2, pp. 192–202, Jun. 2013, doi: 10.1007/s40435-013-0016-y.

[102] H. W. Penrose, "Evaluation of DFIG Wind Turbine Generator and Transformer Conditions with Electrical Signature Analysis," in *2022 IEEE Electrical Insulation Conference (EIC)*, Jun. 2022, pp. 379–384. doi: 10.1109/EIC51169.2022.9833161.

[103] J. B. Ekanayake, L. Holdsworth, and N. Jenkins, "Comparison of 5th order and 3rd order machine models for doubly fed induction generator (DFIG) wind turbines," *Electric Power Systems Research*, vol. 67, no. 3, pp. 207–215, Dec. 2003, doi: 10.1016/S0378-7796(03)00109-3.

[104] M. Cañas Carretón, "Nuevas propuestas de resolución de modelos electromecánicos de aerogeneradores: aplicación a problemas de agregación," UPCT, Cartagena, España, 2013. Accessed: Mar. 26, 2020. [Online]. Available: https://repositorio.upct.es/handle/10317/3901

[105] L. Quéval and H. Ohsaki, "Back-to-back converter design and control for synchronous generator-based wind turbines," in *2012 International Conference on Renewable Energy Research and Applications (ICRERA)*, Nov. 2012, pp. 1–6. doi: 10.1109/ICRERA.2012.6477300.

[106] R. V. de Oliveira, J. A. Zamadei, M. A. Cardoso, and R. Zamodzki, "Control of wind generation units based on doubly-fed induction generator for small-signal stability enhancement," in *2012 IEEE Power and Energy Society General Meeting*, Jul. 2012, pp. 1–8. doi: 10.1109/PESGM.2012.6344855.

[107] A. Manyonge, R. Manyala, F. Onyango, and J. Shichika, "Mathematical Modelling of Wind Turbine in a Wind Energy Conversion System: Power Coefficient Analysis," *Applied Mathematical Sciences*, vol. 6, pp. 4527–4536, Jan. 2012.

[108] O. Alan and W. Alan, *Signals and Systems*. Upper Saddle River, N.J, 1996.

[109] A. Ishchenko, J. M. A. Myrzik, and W. L. Kling, "Linearization of Dynamic Model of Squirrel-Cage Induction Generator Wind Turbine," in *2007 IEEE Power Engineering Society General Meeting*, Jun. 2007, pp. 1–8. doi: 10.1109/PES.2007.386079.

[110] R. C. Ward, “Numerical Computation of the Matrix Exponential With Accuracy Estimate,” *SIAM Journal on Numerical Analysis*, vol. 14, no. 4, pp. 600–610, 1977.
[111] J. Guenther, “An Adaptive, Highly Accurate and Efficient, Parker-Sochacki Algorithm for Numerical Solutions to Initial Value Ordinary Differential Equation Systems,” *SIAM Undergraduate Research Online*, vol. 12, Jan. 2019, doi: 10.1137/19S019115.
[112] M. Hochbruck and A. Ostermann, “Exponential integrators,” *Acta Numerica*, vol. 19, pp. 209–286, May 2010, doi: 10.1017/S0962492910000048.
[113] L. Sigrist, E. Lobato, F. M. Echavarren, I. Egido, and L. Rouco, *Island Power Systems*, Edition 1. CRC Press, 2016.
[114] J. R. Dormand, M. E. A. El-Mikkawy, and P. J. Prince, “High-Order Embedded Runge-Kutta-Nystrom Formulae,” *IMA Journal of Numerical Analysis*, vol. 7, no. 4, pp. 423–430, Oct. 1987, doi: 10.1093/imanum/7.4.423.
[115] Lazard Ltd, “Lazard’s Levelized Cost Of Energy Analysis—Version 15.0,” Lazard, New Orleans, LA, USA, Version 15.0, Oct. 2021.
[116] K. McIntosh, P. Altermatt, and G. Heiser, “Depletion-region recombination in silicon solar cells: when does m_DR = 2 ?,” May 2000.
[117] A. S. Sedra, K. C. Smith, T. C. Carusone, and V. Gaudet, *Microelectronic Circuits*, 8th edition. New York ; Oxford: Oxford University Press, 2019.
[118] I. Subedi, T. J. Silverman, M. G. Deceglie, and N. J. Podraza, “Emissivity of solar cell cover glass calculated from infrared reflectance measurements,” *Solar Energy Materials and Solar Cells*, vol. 190, pp. 98–102, Feb. 2019, doi: 10.1016/j.solmat.2018.09.027.
[119] İ. Ceylan *et al.*, “Determination of the heat transfer coefficient of PV panels,” *Energy*, vol. 175, pp. 978–985, May 2019, doi: 10.1016/j.energy.2019.03.152.
[120] J. Dawe and P. Austin, “Statistical analysis of a LES shallow cumulus cloud ensemble using a cloud tracking algorithm,” *Atmospheric Chemistry and Physics Discussions*, vol. 11, pp. 23231–23273, Aug. 2011, doi: 10.5194/acpd-11-23231-2011.
[121] C. Tobar and A. Karina, “Large scale photovoltaic power plants: configuration, integration and control,” Ph.D. Thesis, Universitat Politècnica de Catalunya, 2018. [Online]. Available: http://www.tdx.cat/handle/10803/619800
[122] J. López Seguel, S. I. Jr, and L. M. F. Morais, “Comparison of the performance of MPPT methods applied in converters Buck and Buck-Boost for autonomous photovoltaic systems,” *Ingeniare. Revista chilena de ingeniería*, vol. 29, pp. 229–244, Jun. 2021, doi: 10.4067/S0718-33052021000200229.
[123] R. Yan, S. Roediger, and T. K. Saha, “Impact of photovoltaic power fluctuations by moving clouds on network voltage: A case study of an urban network,” in *AUPEC 2011*, Sep. 2011, pp. 1–6.
[124] Nuclear Energy Agency, “Robustness of Electrical Systems of NPPs in Light of the Fukushima Daiichi Accident: Appendix 2 (Cont’d) and Appendix 3,” Paris, France, NEA/CSNI/R(2015)4/ADD1, Mar. 2015. Accessed: Jan. 13, 2021. [Online]. Available: https://www.oecd-nea.org/jcms/pl_19602/robustness-of-electrical-systems-of-npps-in-light-of-the-fukushima-daiichi-accident-appendix-2-cont-d-and-appendix-3?details=true
[125] Ministerio de Industria, Energía y Turismo, *Real Decreto 337/2014, de 9 de mayo, por el que se aprueban el Reglamento sobre condiciones técnicas y garantías de seguridad en instalaciones eléctricas de alta tensión y sus Instrucciones Técnicas Complementarias ITC-RAT 01 a 23*, vol. BOE-A-2014-6084. 2014, pp. 43598–43728. Accessed: May 19, 2023. [Online]. Available: https://www.boe.es/eli/es/rd/2014/05/09/337
[126] N. Watson and J. Arrillaga, *Power Systems Electromagnetic Transients Simulation*. IET Digital Library, 2003. doi: 10.1049/PBPO039E.
[127] S. M. Yeo *et al.*, “A novel algorithm for fault classification in transmission lines using a combined adaptive network and fuzzy inference system,” *International Journal of Electrical Power & Energy Systems*, vol. 25, pp. 747–758, Nov. 2003, doi: 10.1016/S0142-0615(03)00029-2.
[128] J. R. Marti, “Accurate Modelling of Frequency-Dependent Transmission Lines in Electromagnetic Transient Simulations,” *IEEE Transactions on Power Apparatus and Systems*, vol. PAS-101, no. 1, pp. 147–157, Jan. 1982, doi: 10.1109/TPAS.1982.317332.

[129] Chung-Wen Ho, A. Ruehli, and P. Brennan, "The modified nodal approach to network analysis," *IEEE Transactions on Circuits and Systems*, vol. 22, no. 6, pp. 504–509, Jun. 1975, doi: 10.1109/TCS.1975.1084079.
[130] J. C. Hernández, "It Can Power a Small Nation. But This Wind Farm in China Is Mostly Idle.," *The New York Times*, Jan. 15, 2017. Accessed: May 25, 2023. [Online]. Available: https://www.nytimes.com/2017/01/15/world/asia/china-gansu-wind-farm.html
[131] W. L. Miranker and W. Liniger, "Parallel Methods for the Numerical Integration of Ordinary Differential Equations," *Mathematics of Computation*, vol. 21, no. 99, pp. 303–320, 1967, doi: 10.2307/2003233.
[132] H. C. Tang, "Parallelizing a fourth-order Runge-Kutta method:," *NIST*, Jan. 1997, Accessed: May 27, 2023. [Online]. Available: https://www.nist.gov/publications/parallelizing-fourth-order-runge-kutta-method-0
[133] *Waveform Relaxation: Theory and Practice*. Electronics Research Laboratory, College of Engineering, University of California, 1985.
[134] M. J. Gander, "50 Years of Time Parallel Time Integration," in *Multiple Shooting and Time Domain Decomposition Methods*, T. Carraro, M. Geiger, S. Körkel, and R. Rannacher, Eds., in Contributions in Mathematical and Computational Sciences. Cham: Springer International Publishing, 2015, pp. 69–113. doi: 10.1007/978-3-319-23321-5_3.
[135] J. Cortial and C. Farhat, "A time-parallel implicit method for accelerating the solution of non-linear structural dynamics problems," *International Journal for Numerical Methods in Engineering*, vol. 77, no. 4, pp. 451–470, 2009, doi: 10.1002/nme.2418.
[136] M. Emmett and M. Minion, "Toward an efficient parallel in time method for partial differential equations," *Communications in Applied Mathematics and Computational Science*, vol. 7, Mar. 2012, doi: 10.2140/camcos.2012.7.105.
[137] D. Goldberg, "What Every Computer Scientist Should Know About Floating-Point Arithmetic," *ACM Computing Surverys*, vol. 23, no. 1, pp. 5–48, Mar. 1991.
[138] Mark Harris, "Optimizing Parallel Reduction in CUDA," NVIDIA, 2008.
[139] S. Dutta and T. J. Overbye, "Optimal Wind Farm Collector System Topology Design Considering Total Trenching Length," *IEEE Transactions on Sustainable Energy*, vol. 3, no. 3, pp. 339–348, Jul. 2012, doi: 10.1109/TSTE.2012.2185817.
[140] J. Wang, X. Xie, and J. Cong, "Communication Optimization on GPU: A Case Study of Sequence Alignment Algorithms," in *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, May 2017, pp. 72–81. doi: 10.1109/IPDPS.2017.79.
[141] M. H. J. Bollen, E. Styvaktakis, and I. Y.-H. Gu, "Categorization and analysis of power system transients," *IEEE Transactions on Power Delivery*, vol. 20, no. 3, pp. 2298–2306, Jul. 2005, doi: 10.1109/TPWRD.2004.843386.
[142] X. Wang and H.-D. Chiang, "Quasi steady-state model for power system stability: Limitations, analysis and a remedy," in *2014 Power Systems Computation Conference, Wroc\law, Poland, August 18-22, 2014*, IEEE, 2014, pp. 1–7. doi: 10.1109/PSCC.2014.7038362.
[143] B. Peccerillo, M. Mannino, A. Mondelli, and S. Bartolini, "A survey on hardware accelerators: Taxonomy, trends, challenges, and perspectives," *Journal of Systems Architecture*, vol. 129, p. 102561, Aug. 2022, doi: 10.1016/j.sysarc.2022.102561.
[144] Ministerio de Industria, Turismo y Comercio, "Resolución de 4 de octubre de 2006, de la Secretaría General de Energía, por la que se aprueba el procedimiento de operación 12.3 Requisitos de respuesta frente a huecos de tensión de las instalaciones eólicas." BOE, Oct. 24, 2006. [Online]. Available: https://www.ree.es/sites/default/files/01_ACTIVIDADES/Documentos/ProcedimientosOperacion/PO_resol_12.3_Respuesta_huecos_eolica.pdf
[145] "Iberdrola Núñez de Balboa PV," European Investment Bank. Accessed: Oct. 18, 2022. [Online]. Available: https://www.eib.org/en/projects/all/20180584
[146] A. Luque and S. Hegedus, *Handbook of Photovoltaic Science and Engineering*, 2nd ed. John Wiley & Sons Inc, 2010.
[147] D. Matuszko, "Influence of the extent and genera of cloud cover on solar radiation intensity," *International Journal of Climatology*, vol. 32, no. 15, pp. 2403–2414, 2012, doi:

10.1002/joc.2432.
[148] M. Cañas-Carretón, E. Gomez-Lazaro, S. Amat-Plata, and Á. Molina-Garcia, "Simulation of DFIG wind turbines for transient studies: An alternative approach based on symbolic–numeric computations," *Journal of the Franklin Institute*, vol. 352, Jan. 2015, doi: 10.1016/j.jfranklin.2014.12.019.
[149] M.-A. Faedy and I. Scian, "WindSTAR - World's first large 33 and 66 kV offshore wind turbine transformer," *ABB Review*, pp. 54–59, Jan. 2018.
[150] M. Luo, "Multi-Physical Domain Modeling of a DFIG Wind Turbine System using PLECS®." 2014.
[151] T. Ackermann, *Wind Power in Power Systems*. Chichester, West Sussex ; Hoboken, N.J, 2012.
[152] GAMESA, "Gamesa G80 - 2,00 MW - Wind Turbine Datasheet." 2011.
[153] ABB, "Distribution Transformers Catalog." Accessed: Apr. 12, 2021. [Online]. Available: http://ocw.uniovi.es/pluginfile.php/5422/mod_resource/content/1/Cat%C3%A1logo%20transformadores%20ABB.pdf
[154] S. Pindado and J. Cubas, "Simple mathematical approach to solar cell/panel behavior based on datasheet information," *Renewable Energy*, vol. 103, pp. 729–738, Apr. 2017, doi: 10.1016/j.renene.2016.11.007.
[155] General Cable, "Cables para instalaciones de energía solar fotovoltaica." Prysmian Group. Accessed: May 23, 2023. [Online]. Available: https://www.aunadistribucion.com/mostrar-catalogo/1437

Annex A. List of Unified Equations

This annex includes the mathematical equations that define the unified DFIG wind model. The subsystems that were introduced throughout Chapter 3 were substituted one into another to find a single explicit system of differential equations, where the Left-Hand Side (LHS) only features the derivative of a state. These equations are extremely long and for that reason they have been written in this annex using common sub-expressions. The variables T_x are temporary values equal to an expression that is repeated more than once in the system of equations. The temporary values are exclusively used in each subsection, and their numbering is restarted for each one. After all temporary values are defined, the equation or matrix is expressed in terms of these temporary values at the end of each subsection of this annex. The mathematical expressions shown in this annex were obtained with the aid of the Python package `sympy`[20].

A.1. 3rd order Nonlinear DFIG Wind Turbine Model

The first section of this annex will detail the full Equations (240)-(268) of the 3rd order nonlinear DFIG wind model used in this thesis. These equations are deduced from the equations shown in Section 3.7 of this thesis.

$$T_1 = L_{ls} + L_m \tag{240}$$

$$T_2 = L_m^2 - T_1(L_{lr} + L_m) \tag{241}$$

$$\begin{aligned} T_3 = {} & L_{lr}^2 L_{ls}^2 \omega_s v_{sq} + 2L_{lr}^2 L_{ls} L_m \omega_s v_{sq} + L_{lr}^2 L_{ls} R_s v_{sd} + L_{lr}^2 L_m^2 \omega_s v_{sq} + L_{lr}^2 L_m R_s v_{sd} \\ & + 2L_{lr} L_{ls}^2 L_m \omega_s v_{sq} + 2L_{lr} L_{ls} L_m^2 \omega_s v_{sq} + L_{lr} L_{ls} L_m R_s \omega_s \phi_{rq} + 2L_{lr} L_{ls} L_m R_s v_{sd} \\ & + L_{lr} L_m^2 R_s \omega_s \phi_{rq} + L_{lr} L_m^2 R_s v_{sd} + L_{lr} L_m R_s^2 \phi_{rd} + L_{ls}^2 L_m^2 \omega_s v_{sq} + L_{ls} L_m^2 R_s \omega_s \phi_{rq} \\ & + L_{ls} L_m^2 R_s v_{sd} + L_m^2 R_s^2 \phi_{rd} \end{aligned} \tag{242}$$

$$\begin{aligned} T_4 = \frac{1}{} & L_{lr}^2 L_{ls}^2 \omega_s^2 + 2L_{lr}^2 L_{ls} L_m \omega_s^2 + L_{lr}^2 L_m^2 \omega_s^2 + \\ & L_{lr}^2 R_s^2 + 2L_{lr} L_{ls}^2 L_m \omega_s^2 + \\ & 2L_{lr} L_{ls} L_m^2 \omega_s^2 + 2L_{lr} L_m R_s^2 + L_{ls}^2 L_m^2 \omega_s^2 + L_m^2 R_s^2 \end{aligned} \tag{243}$$

$$T_5 = L_m T_3 T_4 - T_1 \phi_{rd} \tag{244}$$

$$T_6 = \omega_s \left(L_m T_5 v_{sq} + Q_{ref} T_1 T_2 \right) \tag{245}$$

$$T_7 = -T_2 v_{sq}^2 + T_6 \tag{246}$$

$$T_8 = \omega_r - \omega_s \tag{247}$$

[20] Sympy. `https://www.sympy.org`

$$T_9 = -L_{lr}^2 L_{ls}^2 \omega_s v_{sd} - 2L_{lr}^2 L_{ls} L_m \omega_s v_{sd} + L_{lr}^2 L_{ls} R_s v_{sq} - L_{lr}^2 L_m^2 \omega_s v_{sd} + L_{lr}^2 L_m R_s v_{sq} - 2L_{lr} L_{ls}^2 L_m \omega_s v_{sd} - 2L_{lr} L_{ls} L_m^2 \omega_s v_{sd} - L_{lr} L_{ls} L_m R_s \omega_s \phi_{rd} + 2L_{lr} L_{ls} L_m R_s v_{sq} - L_{lr} L_m^2 R_s \omega_s \phi_{rd} + L_{lr} L_m^2 R_s v_{sq} + L_{lr} L_m R_s^2 \phi_{rq} - L_{ls}^2 L_m^2 \omega_s v_{sd} - L_{ls} L_m^2 R_s \omega_s \phi_{rd} + L_{ls} L_m^2 R_s v_{sq} + L_m^2 R_s^2 \phi_{rq} \quad (248)$$

$$T_{10} = L_m T_4 T_9 - T_1 \phi_{rq} \quad (249)$$

$$T_{11} = L_m T_2 \omega_s v_{sq} (-K_i State_{reactive} T_1 + T_{10} T_8) \quad (250)$$

$$T_{12} = K_{opt} T_1 T_2 \omega_r^2 - L_m T_{10} v_{sq} \quad (251)$$

$$T_{13} = K_i L_m State_{active} T_2 v_{sq} + K_p T_{12} \quad (252)$$

$$T_{14} = K_{2mass} \theta_{shaft} \quad (253)$$

$$T_{15} = \frac{1}{2H_t} \quad (254)$$

$$T_{16} = \frac{1}{\omega_b} \quad (255)$$

$$T_{17} = \omega_t^2 \quad (256)$$

$$T_{18} = n_{poles} r_{gearbox} u_w \quad (257)$$

$$T_{19} = v_{sd}^2 + v_{sq}^2 \quad (258)$$

$$T_{20} = -T_5 \left(K_p T_1 \left(T_2 v_{sq}^2 - T_6 \right) - T_{11} \right) \quad (259)$$

$$\frac{d\phi_{rd}}{dt} = -\frac{K_p T_7 \omega_b}{L_m T_2 \omega_s v_{sq}} - \frac{R_r T_5 \omega_b}{T_2} - T_8 \omega_b \phi_{rq} - \frac{T_{11} \omega_b}{L_m T_1 T_2 \omega_s v_{sq}} \quad (260)$$

$$\frac{d\phi_{rq}}{dt} = -\frac{R_r T_{10} \omega_b}{T_2} + T_8 \omega_b \phi_{rd} + \frac{T_{13} \omega_b}{L_m T_2 v_{sq}} \quad (261)$$

$$\frac{d\omega_r}{dt} = \frac{L_m T_4 \left(T_3 \phi_{rq} - T_9 \phi_{rd} \right) - T_{14} T_2}{2H_r T_2} \quad (262)$$

$$\frac{d\omega_t}{dt} = \begin{cases} T_{14} T_{15} & \text{for } \omega_t \leq 0 \\ T_{15} T_{16} (P_b T_{14} T_{17} \omega_b e^{\frac{21 T_{16} T_{18}}{\omega_t r_{blades}}} + \pi r_{blades} u_w^3 \cdot (60.479 T_{18} - 4.724 \omega_b \omega_t r_{blades})) e^{-\frac{21 T_{16} T_{18}}{\omega_t r_{blades}}} & \text{otherwise} \\ \hline P_b T_{17} \end{cases} \quad (263)$$

$$\frac{d\theta_{shaft}}{dt} = \omega_b (\omega_r - \omega_t) \quad (264)$$

$$\frac{dState_{reactive}}{dt} = -\frac{T_7}{L_m T_2 \omega_s v_{sq}} \quad (265)$$

$$\frac{dState_{active}}{dt} = \frac{T_{12}}{L_m T_2 v_{sq}} \quad (266)$$

$$\frac{dGSC_d}{dt} = -GSC_d P + \frac{T_{10} T_{13} v_{sd}}{L_m T_{19} T_2^2 v_{sq}} - \frac{T_{20} v_{sd}}{L_m T_1 T_{19} T_2^2 \omega_s v_{sq}} \quad (267)$$

$$\frac{dGSC_q}{dt} = -GSC_q P + \frac{T_{10} T_{13}}{L_m T_{19} T_2^2} - \frac{T_{20}}{L_m T_1 T_{19} T_2^2 \omega_s} \quad (268)$$

A.2. 5th order Nonlinear DFIG Wind Turbine Model

As in the previous section, this section will introduce the system of equations that define the 5th order nonlinear DFIG wind turbine model.

$$T_1 = L_{lr} + L_m \tag{269}$$

$$T_2 = L_{ls} + L_m \tag{270}$$

$$T_3 = L_m^2 - T_1 T_2 \tag{271}$$

$$T_4 = L_m \phi_{sd} - T_2 \phi_{rd} \tag{272}$$

$$T_5 = \omega_s \left(L_m T_4 v_{sq} + Q_{ref} T_2 T_3 \right) \tag{273}$$

$$T_6 = -T_3 v_{sq}^2 + T_5 \tag{274}$$

$$T_7 = \omega_r - \omega_s \tag{275}$$

$$T_8 = L_m \phi_{sq} - T_2 \phi_{rq} \tag{276}$$

$$T_9 = L_m T_3 \omega_s v_{sq} (-K_i State_{reactive} T_2 + T_7 T_8) \tag{277}$$

$$T_{10} = K_{opt} T_2 T_3 \omega_r^2 - L_m T_8 v_{sq} \tag{278}$$

$$T_{11} = K_i L_m State_{active} T_3 v_{sq} + K_p T_{10} \tag{279}$$

$$T_{12} = K_{2mass} \theta_{shaft} \tag{280}$$

$$T_{13} = \frac{1}{2H_t} \tag{281}$$

$$T_{14} = \frac{1}{\omega_b} \tag{282}$$

$$T_{15} = \omega_t^2 \tag{283}$$

$$T_{16} = n_{poles} r_{gearbox} u_w \tag{284}$$

$$T_{17} = v_{sd}^2 + v_{sq}^2 \tag{285}$$

$$T_{18} = -T_4 \left(K_p T_2 \left(T_3 v_{sq}^2 - T_5 \right) - T_9 \right) \tag{286}$$

$$\frac{d\phi_{sd}}{dt} = -\frac{\omega_b \left(R_s (L_m \phi_{rd} - T_1 \phi_{sd}) - T_3 \left(\omega_s \phi_{sq} + v_{sd} \right) \right)}{T_3} \tag{287}$$

$$\frac{d\phi_{sq}}{dt} = -\frac{\omega_b \left(R_s \left(L_m \phi_{rq} - T_1 \phi_{sq} \right) + T_3 \left(\omega_s \phi_{sd} - v_{sq} \right) \right)}{T_3} \tag{288}$$

$$\frac{d\phi_{rd}}{dt} = -\frac{K_p T_6 \omega_b}{L_m T_3 \omega_s v_{sq}} - \frac{R_r T_4 \omega_b}{T_3} - T_7 \omega_b \phi_{rq} - \frac{T_9 \omega_b}{L_m T_2 T_3 \omega_s v_{sq}} \tag{289}$$

$$\frac{d\phi_{rq}}{dt} = -\frac{R_r T_8 \omega_b}{T_3} + T_7 \omega_b \phi_{rd} + \frac{T_{11} \omega_b}{L_m T_3 v_{sq}} \tag{290}$$

$$\frac{d\omega_r}{dt} = \frac{-L_m \left(\phi_{rd} \phi_{sq} - \phi_{rq} \phi_{sd} \right) - T_{12} T_3}{2 H_r T_3} \tag{291}$$

$$\frac{d\omega_t}{dt} = \begin{cases} T_{12} T_{13} & \text{for } \omega_t \leq 0 \\ T_{13} T_{14} (P_b T_{12} T_{15} \omega_b e^{\frac{21 T_{14} T_{16}}{\omega_t r_{blades}}} + \\ \pi r_{blades} u_w^3 \cdot (60.479 T_{16} - 4.724 \omega_b \omega_t r_{blades})) e^{-\frac{21 T_{14} T_{16}}{\omega_t r_{blades}}} & \text{otherwise} \end{cases} \tag{292}$$

$$\frac{d\theta_{shaft}}{dt} = \omega_b (\omega_r - \omega_t) \tag{293}$$

$$\frac{dState_{reactive}}{dt} = -\frac{T_6}{L_m T_3 \omega_s v_{sq}} \tag{294}$$

$$\frac{dState_{active}}{dt} = \frac{T_{10}}{L_m T_3 v_{sq}} \tag{295}$$

$$\frac{dGSC_d}{dt} = -GSC_d P + \frac{T_{11} T_8 v_{sd}}{L_m T_{17} T_3^2 v_{sq}} - \frac{T_{18} v_{sd}}{L_m T_{17} T_2 T_3^2 \omega_s v_{sq}} \tag{296}$$

$$\frac{dGSC_q}{dt} = -GSC_q P + \frac{T_{11} T_8}{L_m T_{17} T_3^2} - \frac{T_{18}}{L_m T_{17} T_2 T_3^2 \omega_s} \tag{297}$$

A.3. Equilibrium Point without the Rotor Speed

The set of Equations (298)-(368) define the equilibrium point of the DFIG wind turbine model without the rotor speeds ω_r and ω_t. These magnitudes have no analytical solution, and they are calculated using a hybrid Newton-Raphson method as presented in Section 3.10.

$$T_1 = L_{lr} + L_m \tag{298}$$

$$T_2 = L_{ls} + L_m \tag{299}$$

$$T_3 = L_m^2 - T_1 T_2 \tag{300}$$

$$T_4 = \frac{R_s T_1 \omega_b}{T_3} + \frac{T_3 \omega_b \omega_s^2}{R_s T_1} \tag{301}$$

$$T_5 = \frac{L_m \omega_b \left(-K_p - R_r\right)}{T_3} \tag{302}$$

$$T_6 = -\omega_s + \omega_t \tag{303}$$

$$T_7 = -\frac{L_m T_6 \omega_b}{T_2 T_4} - \frac{T_3 T_5 \omega_s}{R_s T_1 T_4} \tag{304}$$

$$T_8 = \frac{L_m T_3 (T_5 + T_7 \omega_b \omega_s) + T_1 T_2 \omega_b \left(K_p + R_r\right)}{T_1 T_3} \tag{305}$$

$$T_9 = \frac{L_m \omega_b \left(-K_p - R_r\right)}{T_3} \tag{306}$$

$$T_{10} = \frac{L_m T_9 \omega_b \omega_s}{T_1 T_4} + T_6 \omega_b \tag{307}$$

$$T_{11} = \frac{\omega_b \left(-L_m R_s T_{10} T_4 T_7 + L_m R_s T_8 T_9 + T_2 T_4 T_8 \left(K_p + R_r\right)\right)}{T_3 T_4 T_8} \tag{308}$$

$$T_{12} = -\frac{L_m^2}{T_1 T_3} + \frac{L_m^2 \omega_b \omega_s^2}{R_s T_1^2 T_4} + \frac{T_2}{T_3} \tag{309}$$

$$T_{13} = \frac{L_m \omega_b (L_m T_8 \omega_s - R_s T_1 T_{12} T_4 T_7)}{T_1 T_3 T_4 T_8} \tag{310}$$

$$T_{14} = \frac{K_i \omega_b (T_{10} T_{13} - T_{11} T_{12})}{T_{11} T_8} \tag{311}$$

$$T_{15} = \frac{L_m^3 R_s T_7 \omega_b^2 \omega_s}{T_1 T_3^2 T_4 T_8} - \frac{L_m^2 R_s \omega_b}{T_3^2 T_4} + \frac{T_2}{T_3} \tag{312}$$

$$T_{16} = \frac{K_i \omega_b (L_m^2 T_{11} \omega_b \omega_s + T_1 T_{10} T_{15} T_3 T_4)}{T_1 T_{11} T_{14} T_3 T_4 T_8} \tag{313}$$

$$T_{17} = \frac{K_i \omega_b (T_{13} T_{16} - T_{15})}{T_{11}} \tag{314}$$

$$T_{18} = \omega_b v_{sq} + \frac{T_3 \omega_b \omega_s v_{sd}}{R_s T_1} \tag{315}$$

$$T_{19} = -\frac{K_p Q_{ref} T_2 \omega_b}{L_m v_{sq}} + \frac{K_p \omega_b v_{sq}}{L_m \omega_s} - T_{18} T_7 - \frac{T_3 T_5 v_{sd}}{R_s T_1} \tag{316}$$

$$T_{20} = T_3 \left(Q_{ref} T_2 \omega_s - v_{sq}^2\right) \tag{317}$$

$$T_{21} = \frac{K_{opt} K_p T_2 \omega_b \omega_t^2}{L_m v_{sq}} - \frac{T_{10} T_{19}}{T_8} - \frac{T_{18} T_9}{T_4} \tag{318}$$

$$T_{22} = \frac{L_m T_{18}\omega_s}{R_s T_1 T_4} - \frac{L_m v_{sd}}{R_s T_1} + \frac{T_{12}T_{19}}{T_8} + \frac{T_{13}T_{21}}{T_{11}} + \frac{T_{20}}{L_m T_3 \omega_s v_{sq}} \tag{319}$$

$$T_{23} = \frac{K_{opt}T_2\omega_t^2}{L_m v_{sq}} + \frac{L_m^2 T_{19}\omega_b\omega_s}{T_1T_3T_4T_8} + \frac{L_m T_{18}}{T_3T_4} + T_{16}T_{22} - \frac{T_{15}T_{21}}{T_{11}} \tag{320}$$

$$T_{24} = -\frac{K_i T_{13}T_{23}\omega_b}{T_{11}} + T_{17}T_{22} \tag{321}$$

$$T_{25} = \frac{K_i T_{10}T_{24}\omega_b + T_{14}T_8(K_i T_{23}\omega_b - T_{17}T_{21})}{T_8} \tag{322}$$

$$T_{26} = \frac{L_m R_s T_{25}T_7\omega_b + T_{11}T_3(K_i T_{24}\omega_b + T_{14}T_{17}T_{19})}{T_3} \tag{323}$$

$$T_{27} = \frac{L_m R_s T_{25}T_8\omega_b}{T_3} - \frac{L_m T_{26}\omega_b\omega_s}{T_1} - T_{11}T_{14}T_{17}T_{18}T_8 \tag{324}$$

$$T_{28} = \frac{L_m^2(-R_s^2T_1^2\omega_b + R_sT_1T_3T_4 - T_3^2\omega_b\omega_s^2)}{2H_rR_sT_1^2T_3^2T_4T_8} \tag{325}$$

$$T_{29} = \frac{L_m\omega_b\left(-2H_rR_sT_1T_{28}T_3^2T_4T_7T_8^2 - L_m^3R_s^2T_7^2\omega_b^2\omega_s - L_mT_3^2T_8^2\omega_s\right)}{2H_rT_1T_{11}^2T_3^3T_4T_8^2} \tag{326}$$

$$T_{30} = -T_{19}T_{28} - \frac{L_m^3R_sT_{19}T_7\omega_b^2\omega_s}{H_rT_1T_3^2T_4T_8^2} - \frac{L_m^2R_sT_{18}T_7\omega_b}{2H_rT_3^2T_4T_8} + \frac{L_mT_{18}\omega_s}{2H_rR_sT_1T_4} - \frac{L_mv_{sd}}{2H_rR_sT_1} \tag{327}$$

$$T_{31} = \frac{1}{v_{sd}^2 + v_{sq}^2} \tag{328}$$

$$T_{32} = -\frac{K_pL_m^2T_{31}v_{sd}}{T_3^2} - \frac{K_pL_m^2T_{31}\omega_s^2v_{sd}}{R_s^2T_1^2} + \frac{L_m^2T_{31}T_6\omega_sv_{sd}}{R_sT_1T_2} \tag{329}$$

$$T_{33} = \frac{K_{opt}K_pT_2T_{31}\omega_t^2v_{sd}}{T_3v_{sq}} - \frac{2K_pL_m^2T_{31}\omega_sv_{sd}^2}{R_s^2T_1^2} + \frac{K_pT_{20}T_{31}v_{sd}}{R_sT_1T_3v_{sq}} + \frac{L_m^2T_{31}T_6v_{sd}^2}{R_sT_1T_2} \tag{330}$$

$$T_{34} = \frac{L_mT_{31}v_{sd}\left(2K_pL_m^2T_2\omega_s - 2K_pT_1T_2^2\omega_s - L_m^2R_sT_1T_6 + R_sT_1^2T_2T_6\right)}{R_sT_1^2T_2T_3} \tag{331}$$

$$T_{35} = -\frac{K_pL_m^4T_{31}v_{sd}}{T_1^2T_3^2} + \frac{2K_pL_m^2T_2T_{31}v_{sd}}{T_1T_3^2} - \frac{K_pT_2^2T_{31}v_{sd}}{T_3^2} + \frac{L_m^2T_{32}\omega_b^2\omega_s^2}{T_1^2T_4^2} + \frac{L_mT_{34}\omega_b\omega_s}{T_1T_4} \tag{332}$$

$$T_{36} = \frac{2K_pL_m^3T_{31}v_{sd}^2}{R_sT_1^2T_3} - \frac{K_pL_mT_{20}T_{31}v_{sd}}{T_1T_3^2\omega_sv_{sq}} - \frac{2K_pL_mT_2T_{31}v_{sd}^2}{R_sT_1T_3} + \frac{K_pT_2T_{20}T_{31}v_{sd}}{L_mT_3^2\omega_sv_{sq}} - \frac{2L_mT_{18}T_{32}\omega_b\omega_s}{T_1T_4^2} + \frac{L_mT_{33}\omega_b\omega_s}{T_1T_4} - \frac{T_{18}T_{34}}{T_4} \tag{333}$$

$$T_{37} = \frac{2K_pL_mT_2T_{31}v_{sd}}{T_3^2} - \frac{L_mT_{31}T_6\omega_sv_{sd}}{R_sT_1} \tag{334}$$

$$T_{38} = \frac{2L_m^2R_sT_{32}\omega_b^2\omega_s}{T_1T_3T_4^2} + \frac{L_m^2T_{31}T_6v_{sd}}{T_1T_3} + \frac{L_mR_sT_{34}\omega_b}{T_3T_4} + \frac{L_mT_{37}\omega_b\omega_s}{T_1T_4} - \frac{T_2T_{31}T_6v_{sd}}{T_3} \tag{335}$$

$$T_{39} = -\frac{K_p T_2^2 T_{31} v_{sd}}{T_3^2} + \frac{L_m^2 R_s^2 T_{32} \omega_b^2}{T_3^2 T_4^2} + \frac{L_m^2 R_s^2 T_{35} T_7^2 \omega_b^2}{T_3^2 T_8^2} + \frac{L_m R_s T_{37} \omega_b}{T_3 T_4} - \frac{L_m R_s T_{38} T_7 \omega_b}{T_3 T_8} \tag{336}$$

$$T_{40} = -\frac{K_{opt} K_p T_2^2 T_{31} \omega_t^2 v_{sd}}{L_m T_3 v_{sq}} - \frac{2 L_m R_s T_{18} T_{32} \omega_b}{T_3 T_4^2} + \frac{2 L_m R_s T_{19} T_{35} T_7 \omega_b}{T_3 T_8^2} + \frac{L_m R_s T_{33} \omega_b}{T_3 T_4} - \frac{L_m R_s T_{36} T_7 \omega_b}{T_3 T_8} - \frac{L_m T_{31} T_6 v_{sd}^2}{R_s T_1} - \frac{T_{18} T_{37}}{T_4} - \frac{T_{19} T_{38}}{T_8} \tag{337}$$

$$T_{41} = \frac{K_i T_{31} v_{sd} (L_m^2 R_s T_1 T_4 - L_m^2 T_3 \omega_b \omega_s^2 - R_s T_1^2 T_2 T_4)}{R_s T_1^2 T_3 T_4} \tag{338}$$

$$T_{42} = -\frac{K_i L_m^2 T_{31} \omega_b \omega_s v_{sd}}{T_1 T_3 T_4} + \frac{2 K_i L_m R_s T_{35} T_7 \omega_b^2}{T_3 T_8^2} - \frac{K_i T_{38} \omega_b}{T_8} - \frac{L_m R_s T_{41} T_7 \omega_b}{T_3 T_8} \tag{339}$$

$$T_{43} = \frac{K_i \omega_b (K_i T_{10}^2 T_{39} \omega_b + K_i T_{11}^2 T_{35} \omega_b + T_{10} T_{11} T_{42} T_8 - T_{11}^2 T_{41} T_8)}{T_{11}^2 T_8^2} \tag{340}$$

$$T_{44} = \frac{K_i L_m T_{18} T_{31} \omega_s v_{sd}}{R_s T_1 T_4} - \frac{K_i L_m T_{31} v_{sd}^2}{R_s T_1} + \frac{K_i T_{10} T_{40} \omega_b}{T_{11} T_8} - \frac{2 K_i T_{10} T_{21} T_{39} \omega_b}{T_{11}^2 T_8} + \frac{2 K_i T_{19} T_{35} \omega_b}{T_8^2} - \frac{K_i T_{36} \omega_b}{T_8} - \frac{T_{19} T_{41}}{T_8} - \frac{T_{21} T_{42}}{T_{11}} \tag{341}$$

$$T_{45} = \frac{K_i T_{31} v_{sd} \left(-L_m^3 R_s T_7 \omega_b^2 \omega_s + L_m^2 R_s T_1 T_8 \omega_b - T_1 T_2 T_3 T_4 T_8\right)}{T_1 T_3^2 T_4 T_8} \tag{342}$$

$$T_{46} = -\frac{K_i^2 L_m^2 T_{31} \omega_b^2 \omega_s v_{sd}}{T_1 T_3 T_4 T_8} - \frac{2 K_i^2 T_{10} T_{39} \omega_b^2}{T_{11}^2 T_8} + \frac{K_i T_{10} T_{45} \omega_b}{T_{11} T_8} - \frac{K_i T_{42} \omega_b}{T_{11}} \tag{343}$$

$$T_{47} = \frac{K_p T_{31} \left(-Q_{ref} T_2 \omega_s + v_{sq}^2\right)}{T_3 \omega_s} \tag{344}$$

$$T_{48} = -\frac{K_p L_m^2 T_{31} v_{sq}}{T_3^2} - \frac{K_p L_m^2 T_{31} \omega_s^2 v_{sq}}{R_s^2 T_1^2} + \frac{L_m^2 T_{31} T_6 \omega_s v_{sq}}{R_s T_1 T_2} \tag{345}$$

$$T_{49} = \frac{K_{opt} K_p T_2 T_{31} \omega_t^2}{T_3} - \frac{2 K_p L_m^2 T_{31} \omega_s v_{sd} v_{sq}}{R_s^2 T_1^2} + \frac{L_m^2 T_{31} T_6 v_{sd} v_{sq}}{R_s T_1 T_2} - \frac{T_3 T_{47} \omega_s}{R_s T_1} \tag{346}$$

$$T_{50} = \frac{L_m T_{31} v_{sq} \left(2 K_p L_m^2 T_2 \omega_s - 2 K_p T_1 T_2^2 \omega_s - L_m^2 R_s T_1 T_6 + R_s T_1^2 T_2 T_6\right)}{R_s T_1^2 T_2 T_3} \tag{347}$$

$$T_{51} = -\frac{K_p L_m^4 T_{31} v_{sq}}{T_1^2 T_3^2} + \frac{2 K_p L_m^2 T_2 T_{31} v_{sq}}{T_1 T_3^2} - \frac{K_p T_2^2 T_{31} v_{sq}}{T_3^2} + \frac{L_m^2 T_{48} \omega_b^2 \omega_s^2}{T_1^2 T_4^2} + \frac{L_m T_{50} \omega_b \omega_s}{T_1 T_4} \tag{348}$$

$$T_{52} = \frac{2 K_p L_m^3 T_{31} v_{sd} v_{sq}}{R_s T_1^2 T_3} - \frac{2 K_p L_m T_2 T_{31} v_{sd} v_{sq}}{R_s T_1 T_3} + \frac{K_p Q_{ref} T_2^2 T_{31}}{L_m T_3} - \frac{K_p T_2 T_{31} v_{sq}^2}{L_m T_3 \omega_s} - \frac{2 L_m T_{18} T_{48} \omega_b \omega_s}{T_1 T_4^2} + \frac{L_m T_{47}}{T_1} + \frac{L_m T_{49} \omega_b \omega_s}{T_1 T_4} - \frac{T_{18} T_{50}}{T_4} \tag{349}$$

$$T_{53} = \frac{2 K_p L_m T_2 T_{31} v_{sq}}{T_3^2} - \frac{L_m T_{31} T_6 \omega_s v_{sq}}{R_s T_1} \tag{350}$$

$$T_{54} = \frac{2L_m^2 R_s T_{48} \omega_b^2 \omega_s}{T_1 T_3 T_4^2} + \frac{L_m^2 T_{31} T_6 v_{sq}}{T_1 T_3} + \frac{L_m R_s T_{50} \omega_b}{T_3 T_4} + \frac{L_m T_{53} \omega_b \omega_s}{T_1 T_4} - \frac{T_2 T_{31} T_6 v_{sq}}{T_3} \tag{351}$$

$$T_{55} = -\frac{K_p T_2^2 T_{31} v_{sq}}{T_3^2} + \frac{L_m^2 R_s^2 T_{51} T_7^2 \omega_b^2}{T_3^2 T_8^2} + \frac{L_m^2 R_s^2 T_{48} \omega_b^2}{T_3^2 T_4^2} - \frac{L_m R_s T_{54} T_7 \omega_b}{T_3 T_8} + \frac{L_m R_s T_{53} \omega_b}{T_3 T_4} \tag{352}$$

$$T_{56} = -\frac{K_{opt} K_p T_2^2 T_{31} \omega_t^2}{L_m T_3} - \frac{2L_m R_s T_{18} T_{48} \omega_b}{T_3 T_4^2} + \frac{2L_m R_s T_{19} T_{51} T_7 \omega_b}{T_3 T_8^2} - \frac{L_m R_s T_{52} T_7 \omega_b}{T_3 T_8} + \frac{L_m R_s T_{49} \omega_b}{T_3 T_4} - \frac{L_m T_{31} T_6 v_{sd} v_{sq}}{R_s T_1} - \frac{T_{18} T_{53}}{T_4} - \frac{T_{19} T_{54}}{T_8} \tag{353}$$

$$T_{57} = \frac{K_i T_{31} v_{sq} (L_m^2 R_s T_1 T_4 - L_m^2 T_3 \omega_b \omega_s^2 - R_s T_1^2 T_2 T_4)}{R_s T_1^2 T_3 T_4} \tag{354}$$

$$T_{58} = -\frac{K_i L_m^2 T_{31} \omega_b \omega_s v_{sq}}{T_1 T_3 T_4} + \frac{2K_i L_m R_s T_{51} T_7 \omega_b^2}{T_3 T_8^2} - \frac{K_i T_{54} \omega_b}{T_8} - \frac{L_m R_s T_{57} T_7 \omega_b}{T_3 T_8} \tag{355}$$

$$T_{59} = \frac{K_i \omega_b (K_i T_{10}^2 T_{55} \omega_b + K_i T_{11}^2 T_{51} \omega_b + T_{10} T_{11} T_{58} T_8 - T_{11}^2 T_{57} T_8)}{T_{11}^2 T_8^2} \tag{356}$$

$$T_{60} = \frac{K_i L_m T_{18} T_{31} \omega_s v_{sq}}{R_s T_1 T_4} - \frac{K_i L_m T_{31} v_{sd} v_{sq}}{R_s T_1} + \frac{K_i T_{10} T_{56} \omega_b}{T_{11} T_8} - \frac{2K_i T_{10} T_{21} T_{55} \omega_b}{T_{11}^2 T_8} + \frac{2K_i T_{19} T_{51} \omega_b}{T_8^2} - \frac{K_i T_{52} \omega_b}{T_8} - \frac{T_{19} T_{57}}{T_8} - \frac{T_{21} T_{58}}{T_{11}} \tag{357}$$

$$T_{61} = \frac{K_i T_{31} v_{sq} \left(-L_m^3 R_s T_7 \omega_b^2 \omega_s + L_m^2 R_s T_1 T_8 \omega_b - T_1 T_2 T_3 T_4 T_8\right)}{T_1 T_3^2 T_4 T_8} \tag{358}$$

$$T_{62} = -\frac{K_i^2 L_m^2 T_{31} \omega_b^2 \omega_s v_{sq}}{T_1 T_3 T_4 T_8} - \frac{2K_i^2 T_{10} T_{55} \omega_b^2}{T_{11}^2 T_8} + \frac{K_i T_{10} T_{61} \omega_b}{T_{11} T_8} - \frac{K_i T_{58} \omega_b}{T_{11}} \tag{359}$$

$$\phi_{sd} = \frac{-L_m R_s T_{26} T_4 - T_3 (T_{11} T_{14} T_{17} T_4 T_8 v_{sd} + T_{27} \omega_s)}{R_s T_1 T_{11} T_{14} T_{17} T_4 T_8} \tag{360}$$

$$\phi_{sq} = \frac{T_{27}}{T_{11} T_{14} T_{17} T_4 T_8} \tag{361}$$

$$\phi_{rd} = -\frac{T_{26}}{T_{11} T_{14} T_{17} T_8} \tag{362}$$

$$\phi_{rq} = \frac{T_{25}}{T_{11} T_{14} T_{17}} \tag{363}$$

$$\begin{aligned}\theta_{shaft} = {} & \frac{1}{K_{2mass}T_1T_{11}T_{14}^2T_{17}^2T_3^2T_4T_8^3}\cdot(2H_rK_i^2T_1T_{11}T_{14}^2T_{23}^2T_{29}T_3^2T_4T_8^3\omega_b^2 \\ & +K_i^2T_{24}^2\omega_b^2\cdot\Big(2H_rT_1T_{10}^2T_{11}T_{29}T_3^2T_4T_8 - L_m^2T_{11}T_3T_8\omega_b\omega_s \\ & -2T_{10}\big(H_rT_1T_{28}T_3^2T_4T_8^2 + L_m^3R_sT_7\omega_b^2\omega_s\big)\Big) \\ & -K_iT_{14}T_{17}T_{24}T_8\omega_b(4H_rT_1T_{10}T_{11}T_{21}T_{29}T_3^2T_4T_8 - 2H_rT_1T_{10}T_3^2T_{30}T_4T_8 \\ & +2L_m^2T_{11}T_{19}T_3\omega_b\omega_s \\ & +L_mT_1T_{11}T_{18}T_3T_8 - 2T_{21}\big(H_rT_1T_{28}T_3^2T_4T_8^2 + L_m^3R_sT_7\omega_b^2\omega_s\big)) \\ & +2K_iT_{14}T_{23}T_8\omega_b(2H_rK_iT_1T_{10}T_{11}T_{24}T_{29}T_3^2T_4T_8\omega_b - H_rT_1T_{14}T_{17}T_3^2T_4T_8^2 \\ & \cdot(2T_{11}T_{21}T_{29} - T_{30}) \\ & -K_iT_{24}\omega_b\big(H_rT_1T_{28}T_3^2T_4T_8^2 + L_m^3R_sT_7\omega_b^2\omega_s\big) \\ & +T_{14}^2T_{17}^2T_3T_8\cdot(2H_rT_1T_{11}T_{21}^2T_{29}T_3T_4T_8^2 - 2H_rT_1T_{21}T_3T_{30}T_4T_8^2 \\ & -L_m^2T_{11}T_{19}^2\omega_b\omega_s - L_mT_1T_{11}T_{18}T_{19}T_8))\end{aligned} \tag{364}$$

$$State_{reactive} = \frac{T_{24}}{T_{14}T_{17}} \tag{365}$$

$$State_{active} = -\frac{T_{23}}{T_{17}} \tag{366}$$

$$\begin{aligned}GSC_d = {} & \frac{K_i^2T_{13}^2T_{23}^2T_{43}\omega_b^2}{PT_{11}^2T_{14}^2T_{17}^2} + \frac{K_i^2T_{23}^2T_{39}\omega_b^2}{PT_{11}^2T_{17}^2} + \frac{K_iL_m^2T_{19}T_{23}T_{31}\omega_b\omega_sv_{sd}}{PT_1T_{17}T_3T_4T_8} \\ & + \frac{K_iL_mT_{18}T_{23}T_{31}v_{sd}}{PT_{17}T_3T_4} - \frac{K_iT_{13}T_{23}T_{44}\omega_b}{PT_{11}T_{14}T_{17}} \\ & + \frac{K_iT_{13}T_{23}^2T_{46}\omega_b}{PT_{11}T_{14}T_{17}^2} - \frac{2K_iT_{13}T_{22}T_{23}T_{43}\omega_b}{PT_{11}T_{14}^2T_{17}} + \frac{K_iT_{23}T_{40}\omega_b}{PT_{11}T_{17}} \\ & - \frac{K_iT_{23}^2T_{45}\omega_b}{PT_{11}T_{17}^2} - \frac{2K_iT_{21}T_{23}T_{39}\omega_b}{PT_{11}^2T_{17}} - \frac{K_pL_m^2T_{31}v_{sd}^3}{PR_s^2T_1^2} + \frac{K_pT_{20}T_{31}v_{sd}^2}{PR_sT_1T_3\omega_sv_{sq}} \\ & + \frac{T_{18}^2T_{32}}{PT_4^2} - \frac{T_{18}T_{33}}{PT_4} + \frac{T_{19}^2T_{35}}{PT_8^2} - \frac{T_{19}T_{36}}{PT_8} + \frac{T_{22}T_{44}}{PT_{14}} - \frac{T_{22}T_{23}T_{46}}{PT_{14}T_{17}} \\ & + \frac{T_{22}^2T_{43}}{PT_{14}^2} - \frac{T_{21}T_{40}}{PT_{11}} + \frac{T_{21}T_{23}T_{45}}{PT_{11}T_{17}} + \frac{T_{21}^2T_{39}}{PT_{11}^2}\end{aligned} \tag{367}$$

$$\begin{aligned}GSC_q = {} & \frac{K_i^2T_{13}^2T_{23}^2T_{59}\omega_b^2}{PT_{11}^2T_{14}^2T_{17}^2} + \frac{K_i^2T_{23}^2T_{55}\omega_b^2}{PT_{11}^2T_{17}^2} + \frac{K_iL_m^2T_{19}T_{23}T_{31}\omega_b\omega_sv_{sq}}{PT_1T_{17}T_3T_4T_8} \\ & + \frac{K_iL_mT_{18}T_{23}T_{31}v_{sq}}{PT_{17}T_3T_4} - \frac{K_iT_{13}T_{23}T_{60}\omega_b}{PT_{11}T_{14}T_{17}} \\ & + \frac{K_iT_{13}T_{23}^2T_{62}\omega_b}{PT_{11}T_{14}T_{17}^2} - \frac{2K_iT_{13}T_{22}T_{23}T_{59}\omega_b}{PT_{11}T_{14}^2T_{17}} + \frac{K_iT_{23}T_{56}\omega_b}{PT_{11}T_{17}} \\ & - \frac{K_iT_{23}^2T_{61}\omega_b}{PT_{11}T_{17}^2} - \frac{2K_iT_{21}T_{23}T_{55}\omega_b}{PT_{11}^2T_{17}} - \frac{K_pL_m^2T_{31}v_{sd}^2v_{sq}}{PR_s^2T_1^2} + \frac{T_{18}^2T_{48}}{PT_4^2} \\ & - \frac{T_{18}T_{49}}{PT_4} + \frac{T_{19}^2T_{51}}{PT_8^2} - \frac{T_{19}T_{52}}{PT_8} + \frac{T_{22}T_{60}}{PT_{14}} - \frac{T_{22}T_{23}T_{62}}{PT_{14}T_{17}} + \frac{T_{22}^2T_{59}}{PT_{14}^2} \\ & - \frac{T_{21}T_{56}}{PT_{11}} + \frac{T_{21}T_{23}T_{61}}{PT_{11}T_{17}} + \frac{T_{21}^2T_{55}}{PT_{11}^2} - \frac{T_3T_{47}v_{sd}}{PR_sT_1}\end{aligned} \tag{368}$$

A.4. 3rd Order DFIG Model State-Space Matrices

This section introduces the Equations (369)-(493) of the linearized LTI state-space matrices for the 3rd order linearized model. The equations in this section were obtained by linearizing the system as presented in Section 3.8.

$$T_1 = L_{ls} + L_m \tag{369}$$

$$T_2 = L_m R_s^2 (L_{lr} + L_m) \tag{370}$$

$$T_3 = \frac{1}{\begin{matrix} L_{lr}^2 L_{ls}^2 \omega_s^2 + 2L_{lr}^2 L_{ls} L_m \omega_s^2 + L_{lr}^2 L_m^2 \omega_s^2 + L_{lr}^2 R_s^2 + 2L_{lr} L_{ls}^2 L_m \omega_s^2 \\ +2L_{lr} L_{ls} L_m^2 \omega_s^2 + 2L_{lr} L_m R_s^2 + L_{ls}^2 L_m^2 \omega_s^2 + L_m^2 R_s^2 \end{matrix}} \tag{371}$$

$$T_4 = L_m T_2 T_3 - T_1 \tag{372}$$

$$T_5 = L_m R_s \omega_s (L_{lr} L_{ls} + L_{lr} L_m + L_{ls} L_m) \tag{373}$$

$$T_6 = L_{lr} + L_m \tag{374}$$

$$T_7 = L_m^2 - T_1 T_6 \tag{375}$$

$$T_8 = \omega_r - \omega_s \tag{376}$$

$$T_9 = L_m \omega_s v_{sq} \left(K_p T_1 T_4 - L_m T_3 T_5 T_7 T_8\right) \tag{377}$$

$$T_{10} = L_m \omega_s v_{sq} \left(K_p L_m T_1 T_3 T_5 + T_4 T_7 T_8\right) \tag{378}$$

$$\begin{matrix} T_{11} = T_3(-L_{lr}^2 L_{ls}^2 \omega_s v_{sd} - 2L_{lr}^2 L_{ls} L_m \omega_s v_{sd} + L_{lr}^2 L_{ls} R_s v_{sq} - L_{lr}^2 L_m^2 \omega_s v_{sd} + L_{lr}^2 L_m R_s v_{sq} \\ - 2L_{lr} L_{ls}^2 L_m \omega_s v_{sd} - 2L_{lr} L_{ls} L_m^2 \omega_s v_{sd} \\ -L_{lr} L_{ls} L_m R_s \omega_s \phi_{rd} + 2L_{lr} L_{ls} L_m R_s v_{sq} \\ -L_{lr} L_m^2 R_s \omega_s \phi_{rd} + L_{lr} L_m^2 R_s v_{sq} + L_{lr} L_m R_s^2 \phi_{rq} - L_{ls}^2 L_m^2 \omega_s v_{sd} \\ -L_{ls} L_m^2 R_s \omega_s \phi_{rd} + L_{ls} L_m^2 R_s v_{sq} + L_m^2 R_s^2 \phi_{rq}) \end{matrix} \tag{379}$$

$$T_{12} = L_m T_{11} - T_1 \phi_{rq} \tag{380}$$

$$T_{13} = R_s \left(L_{lr}^2 L_{ls} + L_{lr}^2 L_m + 2L_{lr} L_{ls} L_m + L_{lr} L_m^2 + L_{ls} L_m^2\right) \tag{381}$$

$$T_{14} = \omega_s \left(L_{lr}^2 L_{ls}^2 + 2L_{lr}^2 L_{ls} L_m + L_{lr}^2 L_m^2 + 2L_{lr} L_{ls}^2 L_m + 2L_{lr} L_{ls} L_m^2 + L_{ls}^2 L_m^2\right) \tag{382}$$

$$T_{15} = L_m^2 T_3 \omega_s v_{sq} \left(K_p T_1 T_{13} - T_{14} T_7 T_8\right) \tag{383}$$

$$\begin{matrix} T_{16} = T_3(L_{lr}^2 L_{ls}^2 \omega_s v_{sq} + 2L_{lr}^2 L_{ls} L_m \omega_s v_{sq} + L_{lr}^2 L_{ls} R_s v_{sd} + L_{lr}^2 L_m^2 \omega_s v_{sq} + L_{lr}^2 L_m R_s v_{sd} \\ + 2L_{lr} L_{ls}^2 L_m \omega_s v_{sq} + 2L_{lr} L_{ls} L_m^2 \omega_s v_{sq} \\ +L_{lr} L_{ls} L_m R_s \omega_s \phi_{rq} + 2L_{lr} L_{ls} L_m R_s v_{sd} + L_{lr} L_m^2 R_s \omega_s \phi_{rq} \\ +L_{lr} L_m^2 R_s v_{sd} + L_{lr} L_m R_s^2 \phi_{rd} + L_{ls}^2 L_m^2 \omega_s v_{sq} \\ +L_{ls} L_m^2 R_s \omega_s \phi_{rq} + L_{ls} L_m^2 R_s v_{sd} + L_m^2 R_s^2 \phi_{rd}) \end{matrix} \tag{384}$$

$$T_{17} = L_m T_{16} - T_1 \phi_{rd} \tag{385}$$

$$T_{18} = \omega_s \left(L_m T_{17} v_{sq} + Q_{ref} T_1 T_7\right) \tag{386}$$

$$T_{19} = T_{18} - T_7 v_{sq}^2 \tag{387}$$

$$T_{20} = L_m T_7 \omega_s v_{sq} (-K_i State_{reactive} T_1 + T_{12} T_8) \tag{388}$$

$$T_{21} = L_m \omega_s \left(L_m T_{14} T_3 v_{sq} + T_{17}\right) \tag{389}$$

$$T_{22} = T_{21} - 2T_7 v_{sq} \tag{390}$$

$$T_{23} = L_m T_7 \omega_s \left(-K_i State_{reactive} T_1 + L_m T_{13} T_3 T_8 v_{sq} + T_{12} T_8\right) \tag{391}$$

$$T_{24} = K_{opt} T_1 T_7 \omega_r^2 - L_m T_{12} v_{sq} \tag{392}$$

$$T_{25} = K_i L_m State_{active} T_7 v_{sq} + K_p T_{24} \tag{393}$$

$$T_{26} = L_m\left(-L_m T_{13} T_3 v_{sq} - T_{12}\right) \quad (394)$$

$$T_{27} = K_i L_m State_{active} T_7 + K_p T_{26} \quad (395)$$

$$T_{28} = \frac{L_m}{2H_r T_7} \quad (396)$$

$$T_{29} = \omega_t \leq 0 \quad (397)$$

$$T_{30} = \omega_t^2 \quad (398)$$

$$T_{31} = K_{2mass} P_b \theta_{shaft} \quad (399)$$

$$T_{32} = n_{poles} r_{gearbox} \quad (400)$$

$$T_{33} = \frac{21}{\omega_b} \quad (401)$$

$$T_{34} = \frac{T_{33}}{\omega_t} \quad (402)$$

$$T_{35} = e^{-\frac{T_{32} T_{34} u_w}{r_{blades}}} \quad (403)$$

$$T_{36} = \pi T_{35} u_w^3 \quad (404)$$

$$T_{37} = \omega_b \omega_t \quad (405)$$

$$T_{38} = 60.479 T_{32} u_w - 4.724 T_{37} r_{blades} \quad (406)$$

$$T_{39} = T_{38} r_{blades} \quad (407)$$

$$T_{40} = \frac{1}{H_t P_b \omega_b} \quad (408)$$

$$T_{41} = \frac{1}{T_{30}} \quad (409)$$

$$T_{42} = \pi T_{35} \quad (410)$$

$$T_{43} = \frac{T_{40} T_{41}}{2} \quad (411)$$

$$T_{44} = T_{32} T_{36} \quad (412)$$

$$T_{45} = K_p T_1\left(T_{18} - T_7 v_{sq}^2\right) + T_{20} \quad (413)$$

$$T_{46} = -K_p L_m^2 T_1 T_{12} T_3 T_5 \omega_s v_{sq} + L_m T_1 T_{25} T_3 T_5 \omega_s + T_{17} T_9 + T_4 T_{45} \quad (414)$$

$$T_{47} = v_{sd}^2 + v_{sq}^2 \quad (415)$$

$$T_{48} = K_p L_m T_1 T_{12} T_4 \omega_s v_{sq} + L_m T_3 T_{45} T_5 - T_1 T_{25} T_4 \omega_s + T_{10} T_{17} \quad (416)$$

$$T_{49} = T_{12} T_7 \omega_s\left(2K_{opt} K_p T_1^2 \omega_r - L_m T_{17} v_{sq}\right) \quad (417)$$

$$T_{50} = T_1 T_{12} T_{25} \omega_s - T_{17} T_{45} \quad (418)$$

$$T_{51} = -GSC_d L_m P T_1 T_{47} T_7^2 \omega_s v_{sq} + T_{50} v_{sd} \quad (419)$$

$$T_{52} = -K_p L_m^2 T_1 T_{12} T_{14} T_3 \omega_s v_{sq} + L_m T_1 T_{14} T_{25} T_3 \omega_s + L_m T_{13} T_3 T_{45} + T_{15} T_{17} \quad (420)$$

$$T_{53} = -T_1 T_{12} T_{25} \omega_s + T_{17} T_{45} \quad (421)$$

$$T_{54} = T_{17}\left(K_p T_1\left(T_{21} - 2T_7 v_{sq}\right) + T_{23}\right) \quad (422)$$

$$T_{55} = L_m T_1 T_{13} T_{25} T_3 \omega_s - L_m T_{14} T_3 T_{45} + T_1 T_{12} T_{27} \omega_s - T_{54} \quad (423)$$

$$T_{56} = -GSC_q L_m P T_1 T_{47} T_7^2 \omega_s - T_{53} \quad (424)$$

$$T_{57} = L_m - T_2 T_3 T_6 \quad (425)$$

$$T_{58} = L_m \phi_{rd} - T_{16} T_6 \quad (426)$$

$$T_{59} = L_m \phi_{rq} - T_{11} T_6 \quad (427)$$

$$T_{60} = L_m T_1 T_7 \omega_s \left(T_{58} v_{sd} + T_{59} v_{sq}\right) \tag{428}$$

$$T_{61} = -\frac{R_r T_4 \omega_b}{T_7} - \frac{T_9 \omega_b}{L_m T_1 T_7 \omega_s v_{sq}} \tag{429}$$

$$T_{62} = -\frac{L_m R_r T_3 T_5 \omega_b}{T_7} - T_8 \omega_b - \frac{T_{10} \omega_b}{L_m T_1 T_7 \omega_s v_{sq}} \tag{430}$$

$$T_{63} = -\frac{\omega_b \left(T_1 \phi_{rq} + T_{12}\right)}{T_1} \tag{431}$$

$$T_{64} = -\frac{L_m R_r T_{13} T_3 \omega_b}{T_7} - \frac{T_{15} \omega_b}{L_m T_1 T_7 \omega_s v_{sq}} \tag{432}$$

$$T_{65} = \frac{\omega_b \left(K_p T_1 T_{19} - K_p T_1 T_{22} v_{sq} - L_m^2 R_r T_1 T_{14} T_3 \omega_s v_{sq}^2 + T_{20} - T_{23} v_{sq}\right)}{L_m T_1 T_7 \omega_s v_{sq}^2} \tag{433}$$

$$T_{66} = -\frac{K_p T_1 \omega_b}{L_m v_{sq}} \tag{434}$$

$$T_{67} = \frac{\omega_b \left(K_p L_m T_3 T_5 + L_m R_r T_3 T_5 + T_7 T_8\right)}{T_7} \tag{435}$$

$$T_{68} = -\frac{T_4 \omega_b \left(K_p + R_r\right)}{T_7} \tag{436}$$

$$T_{69} = \frac{2 K_{opt} K_p T_1 \omega_b \omega_r}{L_m v_{sq}} + \omega_b \phi_{rd} \tag{437}$$

$$T_{70} = \frac{L_m T_{14} T_3 \omega_b \left(K_p + R_r\right)}{T_7} \tag{438}$$

$$T_{71} = \frac{\omega_b \left(-L_m^2 R_r T_{13} T_3 v_{sq}^2 - T_{25} + T_{27} v_{sq}\right)}{L_m T_7 v_{sq}^2} \tag{439}$$

$$T_{72} = T_{28} \left(-T_{11} + T_2 T_3 \phi_{rq} + T_3 T_5 \phi_{rd}\right) \tag{440}$$

$$T_{73} = T_{28} \left(T_{16} - T_2 T_3 \phi_{rd} + T_3 T_5 \phi_{rq}\right) \tag{441}$$

$$T_{74} = -\frac{K_{2mass}}{2 H_r} \tag{442}$$

$$T_{75} = T_{28} T_3 \left(T_{13} \phi_{rq} + T_{14} \phi_{rd}\right) \tag{443}$$

$$T_{76} = T_{28} T_3 \left(-T_{13} \phi_{rd} + T_{14} \phi_{rq}\right) \tag{444}$$

$$T_{77} = \begin{cases} 0 & \text{for } T_{29} \\ -T_{40}(T_{30} T_{31} \omega_b + T_{36} T_{39}) & \\ +T_{43} \omega_t^3 \cdot \left(2 T_{31} T_{37} + T_{32} T_{33} T_{38} T_{41} T_{42} u_w^4 - 4.72 T_{36} \omega_b r_{blades}^2\right) & \text{otherwise} \\ \omega_t^3 & \end{cases} \tag{445}$$

$$T_{78} = \frac{K_{2mass}}{2 H_t} \tag{446}$$

$$T_{79} = \begin{cases} 0 & \text{for } T_{29} \\ T_{43}(-T_{34} T_{38} T_{44} + 3 T_{39} T_{42} u_w^2 + 60.479 T_{44} r_{blades}) & \text{otherwise} \end{cases} \tag{447}$$

$$T_{80} = -\frac{L_m T_3 T_5}{T_7} \tag{448}$$

$$T_{81} = -\frac{L_m T_{13} T_3}{T_7} \tag{449}$$

$$T_{82} = \frac{T_{19} - T_{22} v_{sq}}{L_m T_7 \omega_s v_{sq}^2} \tag{450}$$

$$T_{83} = -\frac{T_1}{L_m v_{sq}} \tag{451}$$

$$T_{84} = \frac{L_m T_3 T_5}{T_7} \tag{452}$$

$$T_{85} = \frac{2K_{opt} T_1 \omega_r}{L_m v_{sq}} \tag{453}$$

$$T_{86} = \frac{L_m T_{14} T_3}{T_7} \tag{454}$$

$$T_{87} = \frac{-T_{24} + T_{26} v_{sq}}{L_m T_7 v_{sq}^2} \tag{455}$$

$$T_{88} = -\frac{T_{46} v_{sd}}{L_m T_1 T_{47} T_7^2 \omega_s v_{sq}} \tag{456}$$

$$T_{89} = -\frac{T_{48} v_{sd}}{L_m T_1 T_{47} T_7^2 \omega_s v_{sq}} \tag{457}$$

$$T_{90} = \frac{T_{49} v_{sd}}{L_m T_1 T_{47} T_7^2 \omega_s v_{sq}} \tag{458}$$

$$T_{91} = \frac{K_i T_{17} v_{sd}}{T_{47} T_7} \tag{459}$$

$$T_{92} = \frac{K_i T_{12} v_{sd}}{T_{47} T_7} \tag{460}$$

$$T_{93} = \frac{T_{47}\left(-2GSC_d L_m P T_1 T_7^2 \omega_s v_{sd} v_{sq} - T_{52} v_{sd} - T_{53}\right) - 2T_{51} v_{sd}}{L_m T_1 T_{47}^2 T_7^2 \omega_s v_{sq}} \tag{461}$$

$$T_{94} = -\frac{GSC_d P}{v_{sq}} - \frac{2GSC_d P v_{sq}}{T_{47}} - \frac{T_{51}}{L_m T_1 T_{47} T_7^2 \omega_s v_{sq}^2} + \frac{T_{55} v_{sd}}{L_m T_1 T_{47} T_7^2 \omega_s v_{sq}} - \frac{2T_{51}}{L_m T_1 T_{47}^2 T_7^2 \omega_s} \tag{462}$$

$$T_{95} = -\frac{K_p T_1 T_{17} v_{sd}}{L_m T_{47} T_7 v_{sq}} \tag{463}$$

$$T_{96} = -\frac{T_{46}}{L_m T_1 T_{47} T_7^2 \omega_s} \tag{464}$$

$$T_{97} = -\frac{T_{48}}{L_m T_1 T_{47} T_7^2 \omega_s} \tag{465}$$

$$T_{98} = \frac{T_{49}}{L_m T_1 T_{47} T_7^2 \omega_s} \tag{466}$$

$$T_{99} = \frac{K_i T_{17} v_{sq}}{T_{47} T_7} \tag{467}$$

$$T_{100} = \frac{K_i T_{12} v_{sq}}{T_{47} T_7} \tag{468}$$

$$T_{101} = \frac{T_{47}\left(-2GSC_q L_m P T_1 T_7^2 \omega_s v_{sd} - T_{52}\right) - 2T_{56} v_{sd}}{L_m T_1 T_{47}^2 T_7^2 \omega_s} \tag{469}$$

$$T_{102} = \frac{T_{47}\begin{pmatrix} -2GSC_q L_m P T_1 T_7^2 \omega_s v_{sq} + L_m T_1 T_{13} T_{25} T_3 \omega_s \\ -L_m T_{14} T_3 T_{45} + T_1 T_{12} T_{27} \omega_s - T_{54} \end{pmatrix} - 2T_{56} v_{sq}}{L_m T_1 T_{47}^2 T_7^2 \omega_s} \tag{470}$$

$$T_{103} = -\frac{K_p T_1 T_{17}}{L_m T_{47} T_7} \tag{471}$$

$$T_{104} = -\frac{T_3 T_5 T_6}{T_7} \tag{472}$$

$$T_{105} = -\frac{T_{13} T_3 T_6}{T_7} \tag{473}$$

$$T_{106} = -\frac{T_{14} T_3 T_6}{T_7} \tag{474}$$

$$T_{107} = \frac{T_3 T_5 T_6}{T_7} \tag{475}$$

$$T_{108} = \frac{T_{14} T_3 T_6}{T_7} \tag{476}$$

$$T_{109} = -\frac{T_{13} T_3 T_6}{T_7} \tag{477}$$

$$T_{110} = \frac{T_3 T_5 T_6 v_{sq}}{T_7} + \frac{T_{57} v_{sd}}{T_7} - \frac{T_{46}}{L_m T_1 T_7^2 \omega_s v_{sq}} \tag{478}$$

$$T_{111} = -\frac{T_3 T_5 T_6 v_{sd}}{T_7} + \frac{T_{57} v_{sq}}{T_7} - \frac{T_{48}}{L_m T_1 T_7^2 \omega_s v_{sq}} \tag{479}$$

$$T_{112} = \frac{T_{49}}{L_m T_1 T_7^2 \omega_s v_{sq}} \tag{480}$$

$$T_{113} = \frac{K_i T_{17}}{T_7} \tag{481}$$

$$T_{114} = \frac{K_i T_{12}}{T_7} \tag{482}$$

$$T_{115} = \frac{L_m \phi_{rd}}{T_7} - \frac{T_{13} T_3 T_6 v_{sd}}{T_7} + \frac{T_{14} T_3 T_6 v_{sq}}{T_7} - \frac{T_{16} T_6}{T_7} - \frac{T_{52}}{L_m T_1 T_7^2 \omega_s v_{sq}} \tag{483}$$

$$T_{116} = \frac{\begin{matrix} -T_{50} - T_{60} v_{sq} + v_{sq}(-L_m T_1 T_7 \omega_s v_{sq}(-L_m \phi_{rq} + T_{11} T_6 \\ + T_{13} T_3 T_6 v_{sq} + T_{14} T_3 T_6 v_{sd}) + T_{55} + T_{60}) \end{matrix}}{L_m T_1 T_7^2 \omega_s v_{sq}^2} \tag{484}$$

$$T_{117} = -\frac{K_p T_1 T_{17}}{L_m T_7 v_{sq}} \tag{485}$$

$$T_{118} = \frac{-T_3 T_5 T_6 v_{sd} + T_{57} v_{sq}}{T_7} \tag{486}$$

$$T_{119} = \frac{-T_3 T_5 T_6 v_{sq} - T_{57} v_{sd}}{T_7} \tag{487}$$

$$T_{120} = \frac{-T_{13} T_3 T_6 v_{sq} - T_{14} T_3 T_6 v_{sd} - T_{59}}{T_7} \tag{488}$$

$$T_{121} = \frac{T_{13} T_3 T_6 v_{sd} - T_{14} T_3 T_6 v_{sq} + T_{58}}{T_7} \tag{489}$$

$$A = \begin{bmatrix} T_{61} & T_{62} & T_{63} & 0 & 0 & K_i\omega_b & 0 & 0 & 0 \\ T_{67} & T_{68} & T_{69} & 0 & 0 & 0 & K_i\omega_b & 0 & 0 \\ T_{72} & T_{73} & 0 & 0 & T_{74} & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & T_{77} & T_{78} & 0 & 0 & 0 & 0 \\ 0 & 0 & \omega_b & -\omega_b & 0 & 0 & 0 & 0 & 0 \\ -\frac{T_4}{T_7} & T_{80} & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ T_{84} & -\frac{T_4}{T_7} & T_{85} & 0 & 0 & 0 & 0 & 0 & 0 \\ T_{88} & T_{89} & T_{90} & 0 & 0 & T_{91} & T_{92} & -P & 0 \\ T_{96} & T_{97} & T_{98} & 0 & 0 & T_{99} & T_{100} & 0 & -P \end{bmatrix} \tag{490}$$

$$B = \begin{bmatrix} T_{64} & T_{65} & T_{66} & 0 \\ T_{70} & T_{71} & 0 & 0 \\ T_{75} & T_{76} & 0 & 0 \\ 0 & 0 & 0 & T_{79} \\ 0 & 0 & 0 & 0 \\ T_{81} & T_{82} & T_{83} & 0 \\ T_{86} & T_{87} & 0 & 0 \\ T_{93} & T_{94} & T_{95} & 0 \\ T_{101} & T_{102} & T_{103} & 0 \end{bmatrix} \tag{491}$$

$$C = \begin{bmatrix} \frac{T_{57}}{T_7} & T_{104} & 0 & 0 & 0 & 0 & 0 & P & 0 \\ T_{107} & \frac{T_{57}}{T_7} & 0 & 0 & 0 & 0 & 0 & 0 & P \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ T_{110} & T_{111} & T_{112} & 0 & 0 & T_{113} & T_{114} & 0 & 0 \\ T_{118} & T_{119} & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \tag{492}$$

$$D = \begin{bmatrix} T_{105} & T_{106} & 0 & 0 \\ T_{108} & T_{109} & 0 & 0 \\ 0 & 0 & 0 & 0 \\ T_{115} & T_{116} & T_{117} & 0 \\ T_{120} & T_{121} & 0 & 0 \end{bmatrix} \tag{493}$$

A.5. 5th Order DFIG Model State-Space Matrices

The previous section introduced the equations for the 3rd order state-space linearized DFIG wind turbine model. This section shows the equations for the 5th order model (494)-(600).

$$T_1 = L_{lr} + L_m \tag{494}$$
$$T_2 = L_{ls} + L_m \tag{495}$$
$$T_3 = L_m^2 - T_1 T_2 \tag{496}$$
$$T_4 = \omega_r - \omega_s \tag{497}$$
$$T_5 = L_m \phi_{sq} - T_2 \phi_{rq} \tag{498}$$
$$T_6 = L_m \phi_{sd} - T_2 \phi_{rd} \tag{499}$$
$$T_7 = L_m T_6 \omega_s - 2 T_3 v_{sq} \tag{500}$$
$$T_8 = L_m T_3 \omega_s (-K_i State_{reactive} T_2 + T_4 T_5) \tag{501}$$
$$T_9 = \omega_s \left(L_m T_6 v_{sq} + Q_{ref} T_2 T_3\right) \tag{502}$$
$$T_{10} = -T_3 v_{sq}^2 + T_9 \tag{503}$$
$$T_{11} = L_m T_3 \omega_s v_{sq} (-K_i State_{reactive} T_2 + T_4 T_5) \tag{504}$$
$$T_{12} = L_m \left(K_i State_{active} T_3 - K_p T_5\right) \tag{505}$$
$$T_{13} = K_{opt} T_2 T_3 \omega_r^2 - L_m T_5 v_{sq} \tag{506}$$
$$T_{14} = K_i L_m State_{active} T_3 v_{sq} + K_p T_{13} \tag{507}$$
$$T_{15} = \frac{L_m}{2 H_r T_3} \tag{508}$$
$$T_{16} = \frac{1}{2 H_r T_3} \tag{509}$$
$$T_{17} = \omega_t \leq 0 \tag{510}$$
$$T_{18} = \omega_t^2 \tag{511}$$
$$T_{19} = K_{2mass} P_b \theta_{shaft} \tag{512}$$
$$T_{20} = n_{poles} r_{gearbox} \tag{513}$$
$$T_{21} = \frac{21}{\omega_b} \tag{514}$$
$$T_{22} = \frac{T_{21}}{\omega_t} \tag{515}$$
$$T_{23} = e^{-\frac{T_{20} T_{22} u_w}{r_{blades}}} \tag{516}$$
$$T_{24} = \pi T_{23} u_w^3 \tag{517}$$
$$T_{25} = \omega_b \omega_t \tag{518}$$
$$T_{26} = 60.479 T_{20} u_w - 4.724 T_{25} r_{blades} \tag{519}$$
$$T_{27} = T_{26} r_{blades} \tag{520}$$
$$T_{28} = \frac{1}{H_t P_b \omega_b} \tag{521}$$
$$T_{29} = \frac{1}{T_{18}} \tag{522}$$
$$T_{30} = \pi T_{23} \tag{523}$$

$$T_{31} = \frac{T_{28}T_{29}}{2} \tag{524}$$

$$T_{32} = T_{20}T_{24} \tag{525}$$

$$T_{33} = -K_p T_2\left(T_3 v_{sq}^2 - T_9\right) + T_{11} \tag{526}$$

$$T_{34} = L_m\left(K_p L_m T_2 T_6 \omega_s v_{sq} + T_{33}\right) \tag{527}$$

$$T_{35} = v_{sd}^2 + v_{sq}^2 \tag{528}$$

$$T_{36} = L_m\omega_s\left(K_p L_m T_2 T_5 v_{sq} + L_m T_3 T_4 T_6 v_{sq} - T_{14}T_2\right) \tag{529}$$

$$T_{37} = T_2\left(-K_p L_m T_2 T_6 \omega_s v_{sq} - T_{33}\right) \tag{530}$$

$$T_{38} = T_2\omega_s\left(-K_p L_m T_2 T_5 v_{sq} - L_m T_3 T_4 T_6 v_{sq} + T_{14}T_2\right) \tag{531}$$

$$T_{39} = T_3 T_5 \omega_s\left(2K_{opt}K_p T_2^2 \omega_r - L_m T_6 v_{sq}\right) \tag{532}$$

$$T_{40} = -T_{14}T_2T_5\omega_s + T_{33}T_6 \tag{533}$$

$$T_{41} = T_{14}T_2T_5\omega_s - T_{33}T_6 \tag{534}$$

$$T_{42} = -GSC_d L_m P T_2 T_3^2 T_{35}\omega_s v_{sq} + T_{41}v_{sd} \tag{535}$$

$$T_{43} = T_6\left(K_p T_2\left(L_m T_6 \omega_s - 2T_3 v_{sq}\right) + T_8\right) \tag{536}$$

$$T_{44} = T_{12}T_2T_5\omega_s - T_{43} \tag{537}$$

$$T_{45} = -\frac{GSC_q P}{T_{35}} - \frac{T_{40}}{L_m T_2 T_3^2 T_{35}^2 \omega_s} \tag{538}$$

$$T_{46} = L_m\phi_{rd} - T_1\phi_{sd} \tag{539}$$

$$T_{47} = L_m\phi_{rq} - T_1\phi_{sq} \tag{540}$$

$$T_{48} = T_{46}v_{sd} + T_{47}v_{sq} \tag{541}$$

$$T_{49} = \frac{R_s T_1 \omega_b}{T_3} \tag{542}$$

$$T_{50} = -\frac{L_m R_s \omega_b}{T_3} \tag{543}$$

$$T_{51} = \frac{R_s T_1 \omega_b}{T_3} \tag{544}$$

$$T_{52} = -\frac{L_m R_s \omega_b}{T_3} \tag{545}$$

$$T_{53} = -\frac{L_m \omega_b\left(K_p + R_r\right)}{T_3} \tag{546}$$

$$T_{54} = -\frac{L_m T_4 \omega_b}{T_2} \tag{547}$$

$$T_{55} = \frac{T_2 \omega_b\left(K_p + R_r\right)}{T_3} \tag{548}$$

$$T_{56} = -\frac{\omega_b\left(T_2\phi_{rq} + T_5\right)}{T_2} \tag{549}$$

$$T_{57} = \frac{\omega_b\left(K_p T_{10} T_2 - K_p T_2 T_7 v_{sq} + T_{11} - T_8 v_{sq}\right)}{L_m T_2 T_3 \omega_s v_{sq}^2} \tag{550}$$

$$T_{58} = -\frac{K_p T_2 \omega_b}{L_m v_{sq}} \tag{551}$$

$$T_{59} = -\frac{L_m \omega_b \left(K_p + R_r\right)}{T_3} \tag{552}$$

$$T_{60} = \frac{T_2 \omega_b \left(K_p + R_r\right)}{T_3} \tag{553}$$

$$T_{61} = \frac{2K_{opt} K_p T_2 \omega_b \omega_r}{L_m v_{sq}} + \omega_b \phi_{rd} \tag{554}$$

$$T_{62} = \frac{\omega_b \left(T_{12} v_{sq} - T_{14}\right)}{L_m T_3 v_{sq}^2} \tag{555}$$

$$T_{63} = -\frac{K_{2mass}}{2H_r} \tag{556}$$

$$T_{64} = \begin{cases} 0 & \text{for } T_{17} \\ -T_{28}(T_{18}T_{19}\omega_b + T_{24}T_{27}) + T_{31}\omega_t^3 \cdot (2T_{19}T_{25} & \\ \quad +T_{20}T_{21}T_{26}T_{29}T_{30}u_w^4 - 4.724T_{24}\omega_b r_{blades}^2) & \\ \dfrac{}{\omega_t^3} & \text{otherwise} \end{cases} \tag{557}$$

$$T_{65} = \frac{K_{2mass}}{2H_t} \tag{558}$$

$$T_{66} = \begin{cases} 0 & \text{for } T_{17} \\ T_{31}(-T_{22}T_{26}T_{32} + 3T_{27}T_{30}u_w^2 + 60.47897778T_{32}r_{blades}) & \text{otherwise} \end{cases} \tag{559}$$

$$T_{67} = \frac{T_{10} - T_7 v_{sq}}{L_m T_3 \omega_s v_{sq}^2} \tag{560}$$

$$T_{68} = -\frac{T_2}{L_m v_{sq}} \tag{561}$$

$$T_{69} = \frac{2K_{opt} T_2 \omega_r}{L_m v_{sq}} \tag{562}$$

$$T_{70} = \frac{-L_m T_5 v_{sq} - T_{13}}{L_m T_3 v_{sq}^2} \tag{563}$$

$$T_{71} = -\frac{T_{34} v_{sd}}{L_m T_2 T_3^2 T_{35} \omega_s v_{sq}} \tag{564}$$

$$T_{72} = -\frac{T_{36} v_{sd}}{L_m T_2 T_3^2 T_{35} \omega_s v_{sq}} \tag{565}$$

$$T_{73} = -\frac{T_{37} v_{sd}}{L_m T_2 T_3^2 T_{35} \omega_s v_{sq}} \tag{566}$$

$$T_{74} = -\frac{T_{38} v_{sd}}{L_m T_2 T_3^2 T_{35} \omega_s v_{sq}} \tag{567}$$

$$T_{75} = \frac{T_{39} v_{sd}}{L_m T_2 T_3^2 T_{35} \omega_s v_{sq}} \tag{568}$$

$$T_{76} = \frac{K_i T_6 v_{sd}}{T_3 T_{35}} \tag{569}$$

$$T_{77} = \frac{K_i T_5 v_{sd}}{T_3 T_{35}} \tag{570}$$

$$T_{78} = \frac{T_{35}\left(-2GSC_d L_m P T_2 T_3^2 \omega_s v_{sd} v_{sq} - T_{40}\right) - 2T_{42} v_{sd}}{L_m T_2 T_3^2 T_{35}^2 \omega_s v_{sq}} \tag{571}$$

$$T_{79} = -\frac{GSC_d P}{v_{sq}} - \frac{2GSC_d P v_{sq}}{T_{35}} - \frac{T_{42}}{L_m T_2 T_3^2 T_{35} \omega_s v_{sq}^2} + \frac{T_{44} v_{sd}}{L_m T_2 T_3^2 T_{35} \omega_s v_{sq}} - \frac{2T_{42}}{L_m T_2 T_3^2 T_{35}^2 \omega_s} \tag{572}$$

$$T_{80} = -\frac{K_p T_2 T_6 v_{sd}}{L_m T_3 T_{35} v_{sq}} \tag{573}$$

$$T_{81} = -\frac{T_{34}}{L_m T_2 T_3^2 T_{35} \omega_s} \tag{574}$$

$$T_{82} = -\frac{T_{36}}{L_m T_2 T_3^2 T_{35} \omega_s} \tag{575}$$

$$T_{83} = -\frac{T_{37}}{L_m T_2 T_3^2 T_{35} \omega_s} \tag{576}$$

$$T_{84} = -\frac{T_{38}}{L_m T_2 T_3^2 T_{35} \omega_s} \tag{577}$$

$$T_{85} = \frac{T_{39}}{L_m T_2 T_3^2 T_{35} \omega_s} \tag{578}$$

$$T_{86} = \frac{K_i T_6 v_{sq}}{T_3 T_{35}} \tag{579}$$

$$T_{87} = \frac{K_i T_5 v_{sq}}{T_3 T_{35}} \tag{580}$$

$$T_{88} = \frac{2v_{sd}\left(-GSC_q P - T_{35} T_{45}\right)}{T_{35}} \tag{581}$$

$$T_{89} = -\frac{2GSC_q P v_{sq}}{T_{35}} - 2T_{45} v_{sq} + \frac{T_{12} T_5}{L_m T_3^2 T_{35}} - \frac{T_{43}}{L_m T_2 T_3^2 T_{35} \omega_s} \tag{582}$$

$$T_{90} = -\frac{K_p T_2 T_6}{L_m T_3 T_{35}} \tag{583}$$

$$T_{91} = -\frac{T_1 v_{sd}}{T_3} - \frac{T_{34}}{L_m T_2 T_3^2 \omega_s v_{sq}} \tag{584}$$

$$T_{92} = -\frac{T_1 v_{sq}}{T_3} - \frac{T_{36}}{L_m T_2 T_3^2 \omega_s v_{sq}} \tag{585}$$

$$T_{93} = \frac{L_m v_{sd}}{T_3} - \frac{T_{37}}{L_m T_2 T_3^2 \omega_s v_{sq}} \tag{586}$$

$$T_{94} = \frac{L_m v_{sq}}{T_3} - \frac{T_{38}}{L_m T_2 T_3^2 \omega_s v_{sq}} \tag{587}$$

$$T_{95} = \frac{T_{39}}{L_m T_2 T_3^2 \omega_s v_{sq}} \tag{588}$$

$$T_{96} = \frac{K_i T_6}{T_3} \tag{589}$$

$$T_{97} = \frac{K_i T_5}{T_3} \tag{590}$$

$$T_{98} = \frac{L_m T_2 T_3 T_{47} \omega_s v_{sq}^2 - T_{41} + T_{44} v_{sq}}{L_m T_2 T_3^2 \omega_s v_{sq}^2} \tag{591}$$

$$T_{99} = -\frac{K_p T_2 T_6}{L_m T_3 v_{sq}} \tag{592}$$

$$T_{100} = -\frac{T_1 v_{sq}}{T_3} \tag{593}$$

$$T_{101} = \frac{T_1 v_{sd}}{T_3} \tag{594}$$

$$T_{102} = \frac{L_m v_{sq}}{T_3} \tag{595}$$

$$T_{103} = -\frac{L_m v_{sd}}{T_3} \tag{596}$$

$$A = \begin{bmatrix} T_{49} & \omega_b\omega_s & T_{50} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ -\omega_b\omega_s & T_{51} & 0 & T_{52} & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ T_{53} & T_{54} & T_{55} & 0 & T_{56} & 0 & 0 & K_i\omega_b & 0 & 0 & 0 \\ 0 & T_{59} & T_4\omega_b & T_{60} & T_{61} & 0 & 0 & 0 & K_i\omega_b & 0 & 0 \\ T_{15}\phi_{rq} & -T_{15}\phi_{rd} & -L_m T_{16}\phi_{sq} & L_m T_{16}\phi_{sd} & 0 & 0 & T_{63} & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & T_{64} & T_{65} & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & \omega_b & -\omega_b & 0 & 0 & 0 & 0 & 0 \\ -\frac{L_m}{T_3} & 0 & \frac{T_2}{T_3} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & -\frac{L_m}{T_3} & 0 & \frac{T_2}{T_3} & T_{69} & 0 & 0 & 0 & 0 & 0 & 0 \\ T_{71} & T_{72} & T_{73} & T_{74} & T_{75} & 0 & 0 & T_{76} & T_{77} & -P & 0 \\ T_{81} & T_{82} & T_{83} & T_{84} & T_{85} & 0 & 0 & T_{86} & T_{87} & 0 & -P \end{bmatrix} \tag{597}$$

$$B = \begin{bmatrix} \omega_b & 0 & 0 & 0 \\ 0 & \omega_b & 0 & 0 \\ 0 & T_{57} & T_{58} & 0 \\ 0 & T_{62} & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & T_{66} \\ 0 & 0 & 0 & 0 \\ 0 & T_{67} & T_{68} & 0 \\ 0 & T_{70} & 0 & 0 \\ T_{78} & T_{79} & T_{80} & 0 \\ T_{88} & T_{89} & T_{90} & 0 \end{bmatrix} \tag{598}$$

$$C = \begin{bmatrix} -\frac{T_1}{T_3} & 0 & \frac{L_m}{T_3} & 0 & 0 & 0 & 0 & 0 & 0 & P & 0 \\ 0 & -\frac{T_1}{T_3} & 0 & \frac{L_m}{T_3} & 0 & 0 & 0 & 0 & 0 & 0 & P \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ T_{91} & T_{92} & T_{93} & T_{94} & T_{95} & 0 & 0 & T_{96} & T_{97} & 0 & 0 \\ T_{100} & T_{101} & T_{102} & T_{103} & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \tag{599}$$

$$D = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ \frac{T_{46}}{T_3} & T_{98} & T_{99} & 0 \\ -\frac{T_{47}}{T_3} & \frac{T_{46}}{T_3} & 0 & 0 \end{bmatrix} \tag{600}$$

Annex B.Parameters and Specifications

This annex details the value of all parameters of all models studied in this thesis and the main specifications of the machine where all simulations were carried out.

B.1. Wind Farm Parameters

Table XI shows the parameters of both the wind turbine and the wind farm as a whole. The whole list of parameters was gathered form diverse sources to suit the simulations carried out in this thesis.

Table XI. Wind turbine parameters.

Variable	Parameter	Value	Units	Reference	Notes
L_m	Magnetizing Inductance	3.34	$p.u.$	[148]	
L_{ls}	Leakage Stator Inductance	0.1110	$p.u.$	[148]	
L_{lr}	Leakage Rotor Inductance	0.1050	$p.u.$	[148]	
R_s	Stator Resistance	0.0130	$p.u.$	[148]	
R_r	Rotor Resistance	0.0094	$p.u.$	[148]	
ω_{mb}	Base Mechanical Speed	1686	rpm	[148]	
v_{b-lv}	Base Voltage (Wind Turbine)	$\frac{690\sqrt{2}}{\sqrt{3}}$	V	[148]	Set to match the nominal power of the proposed wind turbine. Converted to peak phase-to-ground value.
v_{b-lv}	Base Voltage (Transmission Lines)	$\frac{33\sqrt{2}}{\sqrt{3}}$	kV	[149]	Converted to peak phase-to-ground value.
r	Resistance per Unit Length	0.206	$\frac{\Omega}{km}$	[150]	
l	Inductance per Unit Length	0.363	$\frac{mH}{km}$	[150]	
c	Capacitance per Unit Length	0.25	$\frac{\mu F}{km}$	[150]	
g	Shunt Admittance per Unit Length	0	$\frac{S}{km}$	[150]	
ℓ	Cable Length	0.5	km	[151]	The distance between wind turbines is approximately 10 times the diameter of the blades if installed downstream.
F	Friction Factor	0.4	$\frac{p.u.}{rad}$	[106]	
ω_{eb}	Base Electrical Angular Speed	$2\pi 50$	$\frac{rad}{s}$		Standard angular speed of the European grid
ω_s	Stator Angular Speed	1.0	$p.u.$		
H_r	Wind Turbine Inertia Constant	0.3	s	[106]	

Variable	Parameter	Value	Units	Reference	Notes
H_g	Rotor Inertia Constant	3	s	[106]	
$n_{pairpoles}$	# of poles	2	-	[148]	
K_{opt}	Active Power Controller Constant	0.3376	-		Optimal value for the wind turbine. Found with an iterative value sweep.
r_{blades}	Blade Length	39	m	[152]	
$r_{multiplier}$	Gearbox Multiplier Ratio	100	-	[152]	
P_b	Base Power	2	MW	[152]	Set to match the nominal power of the GAMESA G80 Wind Turbine
GSC_{Filter}	GSC Filter	15	-	[105]	According to the reference, the transfer function of the inverter has its pole at $\frac{R_g}{L_g}$. The article proposes $R_g = 0.12\Omega$ and $L_g = 0.008H$. Thus $\frac{0.12}{0.008} = 15$.
$Z_{transformer}$	Impedance Elevating Transformer	6.35	%	[153]	
$P_{transformer}$	Nominal Power of Elevating Transformer	3.15	MW	[153]	

B.2. PVPP Parameters

Table XII shows the parameters of both the PV panel and the cables that are installed to transport energy in the PVPP. The original article had reference values for a 175W PV panel. A 400W PV panel was chosen and the original reference values were scaled to match the new specifications.

Table XII.PVPP wiring parameters.

Variable	Parameter	Value	Units	Reference	Notes
P_{panel}	PV Panel Nominal Power	400	W		
n_a	PV Panel Exponential Constant	48.548	-	[154]	
I_{01}	First PV Panel Exponential Multiplier	$1.1109 \cdot 10^{-9}$	A	[154]	Adapted Value
I_{02}	Second PV Panel Exponential Multiplier	$1.1109 \cdot 10^{-9}$	A	[154]	Adapted Value
R_s	PV Panel Series Resistance	0.1146	Ω	[154]	Adapted Value
R_{sh}	PV Panel Shunt Resistance	90.85	Ω	[154]	
I_{PV}	PV Panel Current at Short circuit	19.47	A	[154]	Adapted Value
K_0	Temperature Performance Loss Percentage	-0.45	%	[154]	
r	Cable Resistance per Unit Length	0.210	$\frac{\Omega}{km}$	[155]	
$\ell_{PVpanels}$	Distance Between PV Panels	2	m		
ℓ_{PV-AC}	Cable Length for AC Part	1	km		

B.3. Machine Specifications

This section details the most relevant specifications of the machine where all tests were carried out. Table XIII shows the operating system, CPU features, RAM size and speed of the machine, while Table XIV shows the main specification of the GPU used in this thesis, the *NVIDIA GeForce RTX 3090*. The values of Table XIII were obtained using the `lshw` utility, while the specifications of the graphics card were extracted using the `ncu` utility, which is part of the NVIDIA SDK.

Table XIII. CPU-Host specifications.

Description	Value
Linux Kernel	5.15.0-60
Linux Distribution	Ubuntu 22.04
CPU Name	Intel(R) Core (TM) i9-10900K CPU
CPU Frequency	3.70 GHz
CPU Architecture	x86_64
CPU Cores	10
L1 Cache	640 KiB
L2 Cache	2560 KiB
L3 Cache	20 MiB
RAM Memory Size	96GiB
RAM Interface	DDR4
RAM Speed	2400 MHz

Table XIV. GPU-Device specifications.

Description	Value
Graphics Card	NVIDIA GeForce RTX 3090
Graphics Card Architecture	NVIDIA Ampere
Graphics Card Chip	GA102
VRAM Size	24 GiB
GPU Compute Capability	8.6
GPU Frequency	1.695 GHz
# Warps per Multiprocessor	48
Maximum Blocks per Multiprocessor	16
# Multiprocessors	82
# Schedulers per Multiprocessor	4
Max Threads per Block	1024
Warp Size	32
L1 Cache	128 kB (per SM)
L2 Cache	6 MB
Display Driver Version	525.116.04
CUDA Version	12.0