Autogenerated Power Applications
- Software Coding Approaches for the Future

Rainer Bacher
Swiss Federal Institute of Technology (ETH), CH 8092 Zürich, Switzerland
E-mail: Rainer.Bacher@eeh.ee.ethz.ch; WWW: http://www.eus.ee.ethz.ch/~bacher

Abstract This tutorial describes two approaches for the autogeneration of parts of power applications:
One approach uses the concept of automatic code differentiation of standard Fortran or C code. This
allows the automatic source code generation of code parts for the computation of first and possibly
higher order derivatives. The other approach uses the concept of systematic software reuse by defining
a domain architecture. In this second approach an appropriate domain model formulates network and
connected entities with behavior leading to the formulation and solution of systems of simultaneous
nonlinear equations.

Keywords - Software Engineering, Power applications, Automatic code differentiation, Software
Reuse, Object orientation, Framework, Code Generation, Newton-Raphson Solution, Network Simulation Environment

1 Introduction: Two approaches of autogenerated power applications

Research, planning and operation permanently asks for new applications and simulation software for
different purposes. Software systems become larger and increasingly complex. As a consequence creat-
ing system software is more difficult to complete on time and within the constraints of budget. Using
traditional software development technology has shown that it is very difficult to create, maintain and
update these applications. Traditional coding means the specification of functional requirements and
the direct functional mapping of these requirements into a software language such as C or Fortran.
When finished, these traditional software coding approaches lead to software which is often not for-
malized at all. As a consequence the resulting code is difficult to understand, it is hard to maintain
and to modify for new requirements.

This tutorial focuses on two aspects of improving these problems:

- One complexity appears in almost all power applications: Code dependencies due to the use of a
Newton- or Newton-Raphson like solution procedure or an approximation of a set of functions.
These approaches need coded functions and also code for first and second order derivatives.
These code dependencies create the above mentioned problems: If one part of the functions
change, the code both for the functions and also the derivative code has to be updated. In this
process it is often hard to find coding errors.

In this tutorial an approach is shown which relieves the developer from these code dependency
problems: They are automatically taken over by an “automatic code differentiation tool”.

• The second code autogeneration approach focuses on the aspect of software reusability: Studies in computer science have shown that reusability can improve software development productivity and quality. Productivity increases as previously developed assets can be used in current applications, which saves new development time. Quality can be increased as frequently reused assets have been tested and corrected in different use cases.

In section 2, we begin with a discussion for the motivation of using autogenerated power applications. This section discusses the concept of generic frameworks versus specific problems and their realization by software. Also, the problems behind the conventional processes of software development for power applications are discussed.

Section 3 discusses a code generation concept which can take any Fortran or C code for any type of functions of any type of unknown variables. Using this code as input, a tool (“ADIFOR” for Fortran Code, “ADIC” for C-code) can automatically derive first (and in a near-future release second) order derivatives using the fact that the derivative matrices are inherently sparse. The Fortran or C code for functions can be either very specialized to a single problem or it can be formulated for a very structured problem area such as a power flow problem.

In section 4, the well-structured problem of the power flow is seen from a higher conceptual level. Starting from a conceptual, more generic level, we describe only types of the fundamental power flow structures and their parameters using a special, user-defined language. This data (i.e. parameters of the structures which includes algebraic methods fundamentally associated to these structures) is then taken to generate the Fortran, C, C++, Matlab or Java code for the functions and dependent first and higher order derivative code. The generated language depends on the tool-capabilities. Since this code generation tool is usually written by the developer familiar with the domain structures and the algorithms, he/she has complete control about the software language in which the problem solving code should be generated. In this tutorial the use of Maple V is proposed as the high level structural problem formulation language and environment. The goal language for the power flow code is C++.

Section 5 gives summaries and conclusions of the presented software methodologies.

In the appendix several information pieces related to the solution of non-linear power flow equations are given:

Appendix A describes the typical power flow equations. We use rectangular coordinates and a non-compact formulation which fits best to the proposed automatic code generation related software development steps. Emphasis if given to the description of the more complex power features such a remote control, shared MVar control of remote voltages, area interchanges and losses.

Appendices B and C show code pieces of Fortran 77 and Maple V code, respectively. They should give an impression of the complexity of the environments for each code generation approach.

2 Basics of power system code generation: From the specification of solution mechanisms of generic concepts to the specification and solution of concrete problems

2.1 Problems with traditional power application development processes

Some of the concepts of this tutorial are explained on a generic level, some are explained at the concrete example problem. These two terms, i.e. “generic” and “concrete” present two extremes. This distinction and the use of intermediate degrees of generic functionality are necessary to understand the presented approaches.
In the following text, an introduction to different levels of generic functionality or levels of concrete problems is given. Thoughts, philosophies and concepts behind autogenerated power application software are explained. The reader should, however, be aware, that at the end of this text, we do not really solve “new” problems. We do not even solve the “old” problems with new algorithmic solution approaches. With the approaches of this tutorial, however, we will be able to set up old (and new) power application problems using some new form of higher level problem specification. We will then apply code generation methods starting from this higher level specification to achieve parametrized code instances. These code instances form the problem solution code.

The text relies strongly on the example of the “well known” (among the group of power system engineers) power flow problem. In order to understand the concepts of automatic code generation one has first to understand the problem of the power flow. Details of a practically useful power flow equation set including many cases for handling operational limits are given in the appendix A of this tutorial.

If the reader approves all points mentioned below, he/she can assume to know the problems associated to the power flow problem to a very high degree. Many readers will only be able to approve some of the questions below. The more the reader has the feeling that he or she does not understand the problems associated with the statements, the harder will it be to grasp the advantages of the text of this tutorial and the underlying software methodology.

- I have programmed an “AC power flow code” using either C, Fortran or any another 3GL language such as Matlab.
- My code uses the Newton-Raphson approach for the solution of the non-linear power flow equations.
- My code computes and stores the Jacobian matrix related to the power flow equations in sparse form.
- My code solves the linear system using a high degree of sparsity.
- I think that my power flow code is not the fastest in the world, however, it is reasonably fast as compared with the best codes on the same computer.
- My power flow code can solve the power flow problem for a large number of electrical networks. My code remains unchanged for all of these different electrical networks. All that needs to be given are the parameters of the entities of the electrical network.
- I have tested my power flow code with many large size networks of which I do not really know all internal structures and network problem areas. Some of the power flow problems of the networks converged with my code, some not. I tried to solve the problematic cases. I think my code is correct and so the problem must be the network data.
- I wanted to test my code with “foreign” network data sets. Unfortunately these data sets had data organized in a different manner than my code assumed. Partially these data sets described objects for which I have already coded parts in my power flow program. Some parts are not yet coded, i.e. my code does not have the functional capability to handle this data. However, without any doubt, the “foreign” data set describes a valid power flow problem.
- After having a rather stable power flow code and after many months of not changing my code, I (or my colleague) needed to incorporate new models (or more refined models) for power system devices in my power flow code. Because I had done many other activities since the last time
when I had modified my power flow code, it was hard to find all the different places where I had to insert new code and to modify old code. After a too long time and many stupid errors (and I remembered that I had done some of these stupid errors already in previous code modification rounds) finally, the code worked with the new model and data.

- Because I wanted to promote my career I knew that this was the last time I ever touched a power flow code. However, I knew that as a software development manager I would have to tell my people how to do software engineering avoiding the problems I had encountered. My goal was to set up a software development process and an associated team to make the process of software engineering much better: I want a software development process,

  - which allows more transparency, i.e. a process which is easier to comprehend, to document and to talk about,
  - which can be modularized and can be adapted to the capabilities of my software development people,
  - where I as responsible person have a good feeling how long it would take to incorporate desired changes,
  - which has a very high chance of producing correct code,
  - which does not give up the high performance achieved with hand-coding,
  - which allows to use generic code pieces. These pieces can be developed by other developers outside my company.

Other statements can be imagined:

- I would like to avoid hand-coding the derivatives of functions of my algorithms. This code is completely defined by my (hand-coded) functions and there seems to be a possible automatic code generation mechanism behind it.

- I would like to switch from an existing Fortran power flow code to a C/C++ or Java code. Due to software technology advances I would like to adapt my power flow code to these new languages without rewriting them from scratch. I am ready to invest in this translation effort. However, the problem is, that in a few years I will have to redo this job again for all my Java code. I really would like to set up an environment today with makes me ready for a quick future change.

The content of this tutorial is a summary of the research by the author in this area [1, 2, 3, 4, 5]. Two concrete and realized code generation approaches are discussed: The first approach \textbf{A} takes “any” Fortran or C code representing functions. The tool (i.e. the code generator) then uses user-defined parameters (which includes the dependent (= functions) and independent (=derivative variables)) variables or symbols of your code to generate new code which includes both the functions and first order derivatives of these functions with respect to the user defined variables. In addition, sparsity of these first order derivative functions can be automatically taken into account.

The other approach \textbf{B} also generates derivatives of user-defined functions automatically. However, the process of computing derivatives is not based on a “free” user defined functional code: It is strongly coupled to a user-defined structural environment. These structures can best be understood as generic classes with certain fixed attribute classes. In addition these structures have also associated classes of methods depending on the final application which we want to solve with all objects. The structures are connected to each other using generic classes of connectivity.
2.2 From coding solutions of generic, parametrized concepts to coding solutions of concrete problems

The new software development concepts discussed in this tutorial lead to software approaches which generalize the power flow problem into domains such as “electric power network applications” or even into the domain of “network applications” where networks can be electric, gas, traffic or possibly data communications networks. Another way of generalization is the solution of power flow equations as a special case of an environment which can solve any set of non-linear equations by a Newton-Raphson solution procedure.

Using this philosophy of de-specialization or generalization of problem areas into higher-level domains is the background for autogenerated applications. In the following text we want to discuss these issues of generalized concepts versus concrete problems and associated software development and documentation issues.

The problem of generalization depends on the following subproblems:

1. Solving a non-linear set of equations $g(x) = 0$ where the number of functions in $g$ is equal to the number of unknowns $x$ and where we assume that a solution exists: As is generally known, we can solve this generic problem by using the Newton-Raphson solution process. In the documentation of this Newton-Raphson solution process, we will find a reference to the Jacobian which is part of the solution process. We note at this point, that methods for certain classes of generic mathematical problems can be described, formalized on paper. The generic class of a mathematical problem is here the solution of a well determined set of non-linear equations.

2. Solving a non-linear set of power flow equations $g_{PF}(x) = 0$. This generic problem is less generic than the above problem $g(x) = 0$: With power flow we imply a reference to a domain where we solve a set of non-linear equations (see above point 1). However, these equations represent a model for the computation of electrical voltages, currents and powers for electrical power system elements such as transformers, transmission lines, power generators and power consumers (loads). For the solution of this problem $g_{PF}(x) = 0$ we can again refer to many books, papers. The main question is: What is the main reason why we write documentation about this problem? Why are we not happy with the description of the generic problem of the above point 1? The reasons are the following:

Power flow equations are always attached to a certain structure: It is an electrical network with nodes and edges. Note that at this point we need to know the generic world of graphs with the inherent property of nodes and edges. Within the world of graphs, nodes and edges are quite concrete and can be described very precisely. Obviously, power flow equations and the solution associated to these power flow equations are based on structures which will not change for a long time (we will talk about the structures later). As a consequence our scientific society has decided many years ago, that it is worth to document these power flow equations.

Documentation makes most sense if the documents remain valid for a long period of time. Documentation has a high value if it describes a “standard” which is accepted and read by many people. Standards do not need to be rewritten in publications which talk about derivations or subclasses of these standards. In that sense, standards describe a certain level of generic functionality of a domain. However, in order to make a standard useful this generic functionality must be concrete enough so that enough people understand it. Understanding means in this context the ability to apply the standard to more concrete problems.
Why is this discussion important? Because the process of software development always ends up with a set of documents and this is the common aspect to standards, as discussed above.

For every software development process there exists always at least one document: It is the set of files containing some form of computer language source code.

Let us come back to the discussion of generic functionality, documentation and standards: When is a document often read, cited, reused: If it contains documented information, standards which have a long time validity. Two extremes can satisfy this point:

- Either we formulate a reasonable degree of generic functionality which is declared (de facto or de jure) as some form of standard. This level of generic functionality should not change for a quite long period of time. This means for example that a software associated to the content of this generic standard can be reused without risking a change. For example, the use of the language C is such a generic standard. We can see the language C as a generic standard, because you can program “everything” in C (However, note, that the language C itself, i.e. the generic standard, is very concrete and precisely defined).

- Or, standards can be formulated for details of a problem area which will also not change for a long period of time. Again, a software using these details in a hard-coded way can be reused without risking a change of their usage. For example, we all know what a resistance is (... do we really know this?). Using a resistance in a software code for power flow equations can be hard-coded.

Let us ask the question, should the software source code of a software development project be more generic, however, precisely defined at this generic level? Should we document, code software much more with the goal of writing code pieces,

- which are generic to a certain level,
- that will not need to be changed, modified for long period of time even if more concrete levels of the software change (examples will be given later).

Another point must be mentioned here: Software development in the very final stage ends up with source code which must be executed on a computer. Before execution the code must be compiled and then either be interpreted or be linked. In order to do these steps the source code must be concrete, it must include all source code statements leading to the final executable code. We must be aware that a software standard can impose an extremely high formalism on a developer. For software, any type of generic software standard can be rigorously checked by compilers.

2.3 Three coding approaches for the solution algorithm of \( g(x) = 0 \)

Let us ask the question if we can write code for the solution of the generic problem \( g(x) = 0 \) where \( g(x) \) can be “any” non-linear set of functions. It is assumed that this information about the concrete functions \( g(x) \) is defined as separate input to an executable program. The developer of this executable program has no information about the concrete algebraic expressions for the generic functions \( g(x) \) which will be fed to the program.

The answer to this problem is given here in three different contexts:

- **First solution approach**: By using the languages C or Fortran as programming languages we can only write non-compilable code skeletons for the solution of this problem assuming the use
of a Newton-Raphson solution algorithm. Without writing a complete C or Fortran interpreter we cannot generate executable code which can “read” the concrete functions \( g(x) \) and then solve them as long as the functions \( g(x) \) have no predefined structures.

- **Second solution approach:** We should, however, be aware that the use of high-level code interpretation leads to completely new viewpoints: Using for example Maple V [6] we can write code for the solution of \( g(x) = 0 \) without knowing the actual user defined functions for \( g(x) \). With an interpreted language such as Maple V generic algorithmic problems such as the solution of \( g(x) = 0 \) can be formalized and be solved by source code statements. We can be sure that such a Maple V code will not change for a long time. Of course, if the source code language (e.g. Maple V) changes the problem solver code has to be rewritten.

- **Third solution approach:** This approach is somewhere between interpreted and executable code: Assume that we hand-code the skeleton for the solution of \( g(x) = 0 \) using C or Fortran. This skeleton is perfectly compilable with the exception of one callable C or Fortran subroutine which will include C or Fortran source functions for \( g(x) \). The call to this subroutine must include information of the values of \( x \). The computation of the Jacobian elements is also assumed to be done in this user-defined C or Fortran routine. We assume that the Jacobian and right-hand-side values are transferred back to the generic program in some standardized form (as sparse structures). This concept allows the coding and compilation of a generic C or Fortran program which can, after compilation of and linking with the user-specified subroutine, solve “any” problem \( g(x) = 0 \).

To close the loop of thinking, software development should have the goal of producing source code and associated documents for high level generic problems, which will not change for a long period of time. The higher the solvable generic level is, the longer will the associated source code be reusable without any changes.

The degree of generic functionality is very important: In some sense every subroutine which is called at more than one place in a source code has a degree of generic functionality. However, as shown at the example of the solution of \( g(x) = 0 \), software development should try to go to higher-level software development solutions for abstractions or levels of generic functionality.

2.4 Coding of the power flow solution algorithm of \( g_{PF}(x) = 0 \)

2.4.1 Conceptual difference of the solution of \( g_{PF}(x) = 0 \) and \( g(x) = 0 \)

In the following the discussion is extended to the solution to the less generic set of power flow equations \( g_{PF} = 0 \) (less generic or more specific is meant in comparison with the solution of \( g(x) = 0 \)).

The state of the art in the area of power flow source codes is represented by generic codes which can handle many types of power flow problems of many types of electrical networks. However, in the following text we want to get a clearer picture about the difference in the software specification of generic solutions of power flow equations as compared to generic solutions of any-type of equations.

Also here, let us ask the question if we can write code for the solution of the generic power flow problem \( g_{PF}(x) = 0 \) where \( g_{PF}(x) \) can be any set of non-linear power flow functions (Note the formulation; It is purposely almost the same type of question as asked before for \( g(x) = 0 \))? It is assumed that this information about the concrete functions \( g_{PF}(x) \) is defined as separate input to a program which is executed. The developer of this executable program needs to know only a certain degree of information about the concrete algebraic expressions for the generic functions \( g_{PF}(x) \) which will be fed to the program.
We are aware that the solution of the “power flow” equations is a subproblem of the solution of “generic non-linear” equations. Thus all points related to the solution of \( g(x) = 0 \) can be referenced and have full validity. Is it still worth talking about software solutions of power flow equations and documenting these software solutions when there are already very good software solutions and documents available about the more generic super-problem \( g(x) = 0 \)?

For the near future, this is a very valid question and its answer is not totally clear. If “standard” computers become very fast, and flexible interpreted languages with symbolic (e.g. differentiation) and numeric (e.g. handling large sparse systems of equations) computational capabilities become available, the power flow problem could just be an example of a non-linear problem and would be solved with the generic program associated to the solution of \( g(x) = 0 \).

However, with the limited computing technology of 1997, it is still worth to specify another level of generic functionality for the solution of the power flow equations. It is mainly the “slower” performance of interpreted languages such as Maple V (and also in some sense Java) as compared to compiled and executed languages such as C, Fortran which leads to another level of coding solution for large network power flow problems.

Also, applying the previously mentioned third solution approach for \( g(x) = 0 \) has the drawback that the user has to hand-code all functions and the Jacobian related to this functions. If functions change, the Jacobian related code has to be changed. If the set of variables \( x \) changes, functions and associated Jacobian related code parts change (note, however, that this single problem can be solved by using the automatic code differentiation approach A, discussed later in this text). In addition, this third approach leaves the developer in the framework of C or Fortran programming without formally supporting a change to another computer language such as Java.

### 2.4.2 Defining, finding structures within \( g_{PF}(x) = 0 \)

We will now discuss an approach which allows

- to go beyond the coding in a single 3GL language, i.e. the final source code language is only a question of implementation of code generation tools,
- to achieve a software development environment which can be adapted to capabilities of software developers,
- to achieve very modularized software development on various generic levels,
- a natural documentation of software based on different generic levels for structures,
- to make good time estimates how long it will take to incorporate desired changes,
- to produce high-quality, correct power flow (power applications) software code,
- to keep the performance of hand-coding,
- to use more generic code pieces from developers outside my company,
- to avoid hand-coding of the derivatives of functions of my algorithms,
- to switch from an existing Fortran power flow code to a C/C++ or Java code using a highly structured, formalized process.
By specifying the power flow equations \( g_{PF}(x) = 0 \), the generic functions \( g(x) = 0 \) get a structure. This structure associated to power flow problems has the following generic characteristics:

From a **mathematical** point of view, the functions are “partially separable”, i.e. \( g_{PF,i}(x) = \sum_{j \in \text{type}} g_{ij}(x) \). This means that they are composed of sums of separable functions where each separable function is dependent on only a few of all variables of the problem. The sum goes over a set of entities of the same type. From a **power systems engineering** point of view, the structures are object types of the power system.

Parametrized instances of these object types form the power flow objects, i.e. transmission lines, phase shift transformers, PV-generators, etc.

Depending of how we formulate the power flow equations, usually these objects types are either generic node or generic branch types (node types: PV-node, PQ-node, etc.; branch types: transmission line, phase shift transformer, tap changing transformer, etc.). In this tutorial another approach is preferred: We can see these structures as Conducting Equipment objects which are connected to each other at Connectivity Nodes\(^1\) (Conducting Equipment type: AC-transmission line, DC-transmission line, Capacitor Bank, Energy Consumer, Synchronous Machine, etc.; Connectivity nodes: Connection of “terminals” of two or more Conducting Equipments).

This latter approach allows more flexibility in the context of the power systems applications. For example it allows the “clean” incorporation of three-winding transformer or the consideration of a multi-phase power flow on the same generic level.

In any case, for the power flow applications it is advantageous that **structures** are defined whose inherent generic characteristics and generic associated methods will not change for a long period of time. This fact helps to write code which will also not change for a long period of time.

### 2.4.3 A fourth software development approach for the solution of \( g_{PF}(x) = 0 \)

The reuse of generic structures and associated methods leads to a fourth solution approach. It seems to take the best properties of all three approaches together with the generic structural properties of power flow applications:

**Fourth solution approach**: We assume that the domain of interest is composed of objects of type Conducting Equipment which have terminals and the terminals are connected to each other via Connectivity Nodes. We assume that we want to solve a set of equations \( g_{PF}(x) = 0 \), where these functions \( g_{PF}(x) \) can be subdivided into two categories of functions:

- One category of functions, \( g_1(x_{\text{internal}}, x_{\text{external}}) \) is limited to the expression of functions of the Conducting Equipment. As mentioned above, a Conducting Equipment is an entity which has a certain number (e.g. 1, 2, 3 or more) of terminals which allow a connection to the terminals of other Conducting Equipment via Connectivity Nodes. The variable types within \( x_{\text{internal}} \) represent internal variables of each Conducting Equipment, the variables \( x_{\text{external}} \) represent external variables associated to the terminals of these Conducting Equipment.

- The second category of functions \( g_2(\text{“Parameters”}) \) is instantiated for every Connectivity Node. We limit the type of functions associated to every Connectivity Node to
  - a) Numerically given sums of external variables of the same variable name of all connected Conducting Equipment,
  - b) to equality relationships (=) between external variables of Conducting Equipment and a variable associated to the Connectivity Node with the same variable name.

\(^1\) The terms Conducting Equipment and Connectivity Node have been taken from the EPRI/API project No. 3654 [7]
The function of type a) can be seen as a flow law, the function of type b) can be seen as a potential law. Later, in section 4 of this tutorial, we will show that both functions can be defined as two special types of a generic sum-function.

These assumptions are the hard-coded constraints and describe the generic level at which power flow equations will be defined. The term “generic” is really applicable, because we can associate the notion of well documentable entities. Also, we have created entities whose characteristics will not change for a long period of time. Also, we can cover many application areas by these generic assumptions.

To understand the process of software development which we can associate to these types of structures, we have to go through some new thoughts:

Can a developer write software code which solves the generic set of non-linear power flow equations \( g_{PF}(x) = 0 \) where the equations follow the above mentioned generic rules?

Of course we can again apply the previously mentioned second approach, which interprets functions given by the user. However, with the generic structures defined above, we can even write C, C++, or Fortran code which solve the problem to a very high degree without knowing the types and associated functions of Conducting Equipment, the types of variables associated to the flow and potential laws related to all Connectivity Nodes, the types of internal variables of Conducting Equipment.

The main difference to the general problem \( g(x) = 0 \) is the fact, that we know how these power flow structures influence the place of first derivatives of the set of functions \( g_{PF}(x) \). Because of the fact that the functional behavior of Conducting Equipment is bound to this type of element, its first and higher order derivatives will also be bound to the variables and functions of an individual Conducting Equipment. Also, because flow and potential laws will only have derivatives \( \neq 0 \) with respect to the connected internal variables of Conducting Equipment, the possible non-zero areas in the Jacobian matrix are known as soon as the main topology of the network is given. Note, that we talk about non-zero number areas within the sparse Jacobian matrix, not of actual places. The places will only be known, when the actual function types, unknown variables and parameters and the connectivity of the network entities have been specified by the user.

As a consequence the skeleton for each non-zero area of the Jacobian can be coded in either C, C++, Fortran, Matlab, Java, etc. This means that our code skeleton can include loops over all Conducting Equipment instances (the number is not yet determined) and starts the computation of the associated Jacobian matrix part.

The exact coding principles for these skeleton parts depend strongly on the chosen source language and its capabilities. For example when using C++, each Conducting Equipment can be a class with associated methods. C++ allows the use of abstract methods. This corresponds to a method name specification without actually defining the method. This fits perfectly to the software development process where the actual methods and functions and even variables and parameters associated to concrete types of Conducting Equipment and Connectivity Nodes are assumed to be defined by the “user”.

When using Matlab we should profit from the fact that Matlab allows very efficient vector statements.

For those Jacobian matrix areas whose non-zero values actually depend on the user defined functional definition, the following software development concept is proposed. Due to the fact that the code will need first or higher order derivatives with respect to the unknown variables, we use Maple V as software environment. Maple V is the ideal language because it allows the following important development steps:

- Definition of lists, vectors, matrices, sets, i.e. it allows to set up structures in a very flexible way.
These lists allow to enter not only numeric data but also symbolic, algebraic expression of user-defined functionality.

Maple V allows to write code which can loop over all instances of functions of a list.

While doing this, Maple V can compute first or higher order derivatives of any function with respect to any unknown variable in symbolic form.

Maple V has built-in simplification procedures for the generated first or higher order functional, algebraic expressions.

Also, it allows to manipulate the resulting functions with typical string operations. This is necessary in order to generate strings which are syntactically correct within the goal language such as C, C++, Fortran, Java, etc.

2.4.4 Code generation for the solution of $g_{PF}(x) = 0$

As mentioned in the preceding subsection Maple V has the capability to manipulate strings. Even strings representing computed derivatives of functions can be manipulated. Also, every string can be written out to ASCII-files. This allows to generate syntactically correct pieces of C, Fortran, C++, Java, Matlab code. We call this goal language a 3GL language.

The final goal of the software development process is a code formulated with a 3GL language. This code should be syntactically correct, should be compilable and after linking it with system libraries, the final code should, when executed, solve power flow problems.

Solving a power flow problem implies, that the program reads first numeric values for all symbols (i.e. parameters and initial values for variables) of Conducting Equipment and Connectivity Node from a data file or a data base.

This code section obviously depends on the definitions of the functionality, variables, parameters, etc. of the types of Conducting Equipment and Connectivity Node. Thus we notice another place in the final power flow solution code (beside the derivative code) which depends on the definitions of the power flow structures.

To generate the complete source code for the power flow program, a Maple V program must be written which coordinates the code-parts: Those parts which are independent of the user defined functional behavior of Conducting Equipment and Connectivity Node must be coordinated with those code parts which are dependent on the user defined functions and the first and higher order derivatives of these functions.

There is quite a high development effort necessary to code this code generating and integrating program with Maple V. However, once the program has been written, a very flexible power flow program generator is available. New types of Conducting Equipment can easily be defined and the complete new power flow code with a completely new functionality related to Conducting Equipment is regenerated.

The only restrictions for the formulation of new functionality are the generic limits of the Conducting Equipment and Connectivity Node. Also, the user has to observe the fact that the number of unknows must in the end be equal to the number of equations (however, this problem could be automated by extending the domain of the Newton-Raphson for example to the domain of a minimum least-squares Newton-type optimization algorithms).

In this concept the reused software is the Maple V program which can generate the power flow codes. Reused software pieces are also those parts of the final power flow code which are independent on the user defined functionality.
2.5 Generic concepts and software development consequences

From the preceding subsection, the following points have become clear.

- Generic functionality must be so concrete that it can be described, formalized, documented. Documentation is often done with examples of less generic, i.e. more specific problems.

- Generic worlds must have a tendency not to change for a long period of time.

- In the domain of interest (power system applications), generic functionality can be found in various areas:
  - Mathematics: algorithmic solution approaches for generic mathematical problems (example: solution of non-linear set of equations using the Newton-Raphson, Jacobian matrix, Linear Programming (LP), Quadratic Programming (QP), Sparse linear system solvers),
  - Power system models: objects having the same type/class of behavior (example: Conducting Equipment with internal behavior between electrical current, voltage and power, Connectivity Nodes with behavior associated to Kirchhoff laws).

- In order to allow software reuse, software development solutions should first be found for the solution of these general characteristics. There is a high chance that software pieces related to generic problems can be reused.

---

**Figure 1: General principle of autogenerated applications**

Fig. 1 shows the general principles behind automatic code generation: A problem specification is formulated within the bounds of a generic problem. This specification is then taken as input to a specific problem code generator program. This specific problem code generation process must be coordinated with the parameter data access of the final specific problem solution code by a specification of this parameter access. The problem specific parameter data is assumed to be given in ASCII files or databases with standardized access. Finally the generated, compiled and linked specific problem solution code computes the solution of the associated problem specified in the parameter database for structures of the specific problem.
3 Code generation approach A: Automatic code differentiation

3.1 ADIFOR and ADIC: Tools for automatic code differentiation

ADIFOR/ADIC are tools to provide automatic code differentiation for Fortran77 and C programs [8]. The developer responsible for coding first or higher order derivatives of code sections describing functions only needs to supply the source code (standard Fortran77 or C code). For the Fortran77 environment the code can contain if statements, do loops, common blocks, implicit none and include statements) and to specify the variables that correspond to the dependent and independent variables. ADIFOR performs a dependency analysis and determines the active variables, i.e. the variables that have an associating derivative object[9].

3.2 Developer point of view: Differentiated code and code generation capabilities

Assume a function $f$ with an $n$—vector $x$ as independent and an $m$—vector $y$ as dependent variables. The ADIFOR-generated code will compute

$$g_y = \left( \frac{\partial y}{\partial x} \times g_x \right)^T = (J * S)^T \quad (1)$$

where $J$ is the Jacobian matrix and $S$ is the seed matrix. By proper initialization of $S$ all elements of the transposed Jacobian or the transpose of a Jacobian-vector product can be obtained. The auto-generated code is an augmentation of the original source code, which must be available in subroutine form (Fortran77). A call to the generated subroutine evaluates both the function value and the value of derivatives (original function related code is no longer needed).

Throughout the text $g_{<\text{var}>}$ is the derivative object associated with the active variable $<\text{var}>$.

3.3 Theory summary of automatic code differentiation (ACD)

Automatic Code Differentiation (ACD) produces code that computes derivatives accurate up to machine precision. Moreover it is applicable to codes of arbitrary length that can contain subroutine calls and constructs such as branches and loops, introducing one main advantage compared to symbolic differentiation techniques [1, 2, 3].

Assume function $f(s) = f(g(t))$ where $t$ is the independent variable and variable $s$ is an intermediate variable that derives from the elementary function $g$. By applying the chain rule (2) derivatives of $f$ are calculated:

$$\frac{\partial f(g(t))}{\partial t} \bigg|_{t=t_0} = \left( \frac{\partial f(s)}{\partial s} \bigg|_{s=g(t_0)} \right) \left( \frac{\partial g(t)}{\partial t} \bigg|_{t=t_0} \right) \quad (2)$$

The two main approaches to ACD that have been traditionally developed are the forward and the reverse mode [9, 10]. These two modes are mainly distinguished by the way the chain rule propagates derivatives. The forward mode computes derivatives of intermediate variables with respect to the independent variables while reverse mode computes derivatives of the final result $f$ with respect to intermediate variables. For advantages and drawbacks of the two modes see [9].

ADIFOR/ADIC employ a hybrid forward/reverse approach: After breaking down $f$ in elementary operations, derivatives of the left hand side with respect to right hand side (intermediate or independent) variables are accumulated and at the end the forward mode propagates derivatives with the chain rule. Of significant importance is the fact that only derivatives with respect to the right hand side variables are considered so only the non-zero derivatives are calculated. To give an impression, we show
the code generation at the following code segment which is a very small part out of a loop over all nodes of a user-defined network: If \( f = V_e^2 + V_f^2 - V_{fix}^2 \) \((V_e/V_f\) represent the real/imaginary part of voltage) then \( f = 0 \) fixes the voltage of a PV-generator to the desired magnitude \( |V_{fix}| \). The parts of the original and generated code that correspond to the voltage fix of generator \((i)\) are:

\[
\text{(in the original (input) routine:)}
\]

\[
\text{mismatch(indx+i) = } V_e(i)^2 + V_f(i)^2 - V2\text{-fix}(i)
\]

\[
\text{(in the generated routine:)}
\]

\[
\begin{align*}
\text{d2\_v} &= V_e(i) * V_e(i) \\
\text{d2\_p} &= 2.0d0 * V_e(i) \\
\text{d4\_v} &= V_f(i) * V_f(i) \\
\text{d1\_p} &= 2.0d0 * V_f(i) \\
call &\text{ dspg2q(g\text{-mismatch(indx+i)}, d1\_p, g\text{-Vf(i)}, d2\_p,} \\
&\quad \& g\text{-Ve(i))} \\
\text{mismatch(indx+i) = } d2\_v + d4\_v - V2\text{-fix}(i)
\end{align*}
\]

In a first stage the function statement has been broken down to the intermediate variables \(d2\_v\) and \(d4\_v\) and then derivatives of those variables are computed with respect to \(V_e\) and \(V_f\) of the right hand side. The function \text{dspg2q} is a SparsLinC routine (section 3.4) that computes

\[
\text{sparse\_object(g\text{-mismatch(indx+i)) =}}
\]

\[
\text{d1\_p} * \text{sparse\_object(g\text{-Vf(i))} +}
\]

\[
\text{d2\_p} * \text{sparse\_object(g\text{-Ve(i))}}
\]

where \text{sparse\_object(g\text{-<var>)}} is the sparse representation of the derivative with pointer \text{g\_<var>}.  

### 3.4 Developer point of view: Automatic sparsity handling

ADIFOR and ADIC allow code generation for full Jacobian and sparse Jacobian matrices. If the Jacobian structure instance of the problem functions are unchanged during the life of the program a clever, user-driven combination of a seed-matrix \(S\) (see equation (1)) and a generation of code for a full Jacobian matrix leads to highest code run-time speed.

For power flow problems the situation is different: Due to the fact even for a given electric power utility the network topology can vary widely (switches, various ways of incorporating network equivalents, various levels of modelling lower voltage level networks, various ways to summarize loads on voltage levels) the structure of the Jacobian cannot be assumed to be constant for a given power flow code. Also, this is not of interest to the power flow code developing community which wants to provide a generic power flow solution code for many types of different electric power networks.

For these cases, ADIFOR and ADIC provide the advanced feature of automatic code generation using inherent sparsity. This means that a Jacobian value which for any network instance will be zero should never be used in any part of the code for the computation of the Jacobian matrix.

To discuss the complexity and the automatic support to a power flow developer we have to be aware about the following: When hand-coding a sparse Jacobian the power flow program developer has to have deep knowledge about the mathematical characteristics of the equations which allows to conclude on the places in the Jacobian matrix with guaranteed zero value. This knowledge will lead to Jacobian code which avoids that at any time during the code execution these zero value elements are referred to by functions calls.

Efficient and correct hand-coding of these parts is very tricky, especially when applied to a practical power flow approach (see appendix A): There exist many code parts for every possibly limited
controlled quantity with an associated possibly added controlling unknown variable. Each of these code parts lead to Jacobian code parts. Clearly, in the same way as the Jacobian depends by the mathematical definition on the functions the code part related to the Jacobian depends on the code part related to the functions (i.e. power flow mismatches).

From a users point of view, ADIFOR/ADIC relieve the programmer 100% from the problem of finding out which parts in the Jacobian for any user defined function code part will be zero and should never be computed and be referred to. This problem is automatically solved by applying compiler technology to the user specified function code: dependencies of code structures and code variables are determined and the non-zeros in the Jacobian matrix are detected.

The automatic sparse code generation approach is supported by a C-language package called SparsLinC [8, 11]. The SparsLinC library exploits sparsity dynamically. Non-zero entries have been detected and calculated at automatic code derivation time. When invoked with the SparsLinC approach, ADIFOR allocates for every active variable an integer type variable that is a pointer to the sparse representation of the corresponding variable. Using C allows to dynamically use memory locations when generating the non-zero terms of the Jacobian.

### 3.5 Steps to apply ADIFOR to the power flow problem

#### 3.5.1 Step 1: Specifying Input for ADIFOR and generating derivative code

In order to apply ADIFOR for generating the code related to the sparse Jacobian matrix of user programmed mismatch function code and also for the combined and efficient computation of the power flow equation mismatches the user must provide ADIFOR with the following code pieces, data and parameter values:

1. The source (Fortran77) code (power_flow.f) that builds the vector of equations (mismatch). This is the name of the variable used in the subroutine code which after execution of this compiled and linked routine will include all numeric mismatch values. This code part includes all necessary case statements (if-then-else), all necessary do-enddo loops, all necessary mathematical functions such as \textit{sin}, \textit{cos}, etc. Using ADIFOR, the user is free to use almost all expression and language capabilities of the standard Fortran77 language.

2. a script file (ADIFOR\_CONTROL.adf) in which all ADIFOR options are listed. This file is the control file for the ADIFOR executable. The most important parameters are:

   ```
   AD_PROG   = ADIFOR\_Main.cmp
   AD_TOP     = power_flow
   AD_IVARS   = Ve_1p,Vf_1p,tap_e,tap_f,\n               Q_pv,Q_pq,...
   AD_OVARS   = mismatch
   AD_FLAVOR  = sparse
   ```

   The option \textit{AD\_IVARS} contains a list of all vector names that represent independent variables and \textit{AD\_OVARS} the dependent variable \textit{mismatch}. As the option \textit{AD\_FLAVOR} is set to \textit{sparse} calls to SparsLinC routines will be invoked and sparse derivative code will be generated.

3. a “dummy” main program (powerflow\_dummy.f) that declares all variables and simply calls the top level subroutine power_flow.f. This dummy program should be compilable without errors and executable without actually solving the problem.
4. A composition file (ADIFOR_Main.cmp) containing the names of the files of the source Fortran77 code to be processed (powerflow_dummy.f and power_flow.f).

With this options script file the derivative code of all code of the top-level subroutine (AD_TOP) and all desired routines called by this top-level routine is generated (g_power_flow.f).

3.5.2 Step 2: Incorporating the generated derivative code to the main power flow program

To clarify the difference of the main dummy program to the true main power flow program some explanations have to be given:

The dummy main program (powerflow_dummy.f) is only needed to tell ADIFOR about the calling conventions of the subroutine (power_flow.f). The calling convention of the generated subroutine (g_power_flow.f) has to be used by the “true” main power flow program which first reads the network parameters from a network data file, stores the parameter data into arrays, checks the parameter data for correctness using heuristic rules and initializes values of the unknown variables.

In this code generation concept A, the “true” main program still has to be hand-coded by the power flow developer. However, it consists mainly of independent code parts related to the used power flow model.

The term “generic” used in the following paragraph indicates that these code sections have no relation with the actual choice of the power flow equation types which are coded in power_flow.f or the autogenerated routine g_power_flow.f together with the derived Jacobian terms.

The main program includes the following parts which need to be hand-coded by the power flow developer:

1. The developer has to specify the order of variables in the Jacobian matrix. Note that this order is not necessarily the same as used in the sparsity package for the sparse linear solution within each Newton-Raphson step.

2. A small generic code piece must be written which calls the autogenerated subroutine (g_power_flow.f) to compute both the function values (mismatch) and all non-zero values of the Jacobian matrix. These values appear to the power flow programmer in a form similar to three arrays i, j, value of same length, where i indicates the row i, j indicates the column j and value the Jacobian matrix value at matrix place value(i, j).

3. The program developer then adds a small, generic code piece to the main power flow program which feeds these sparse Jacobian values into a generic linear system sparsity package (UMFPACK 2 [12] has been used in our implementation) together with the mismatch values.

4. After computing the solution of the sparse linear system of equations a small piece of hand-code is needed which updates the unknown variables.

These code pieces allow to create an executable version of a power flow program by looping over all steps discussed before.

To summarize, with the ADIFOR/SparsLinC concept, the power flow developer still has to code the main, generic steps of a Newton-Raphson which is used as the algorithmic approach. Also, he/she is responsible to set up and call the main step of the Newton-Raphson, i.e. the sparse linear system of equations. Also, the correct (and almost trivial) variable update code sections are not done automatically and these code sections must be written by the power flow developer.

However, the power flow developer is completely relieved from the coding of the formulas for the Jacobian and the fact that there exist non-zero Jacobian elements which must be coded and Jacobian
elements with a value of zero which should never be coded explicitly. This is completely taken over by the ADIFOR automatic code generation.

The power flow developer still has complete control about the power flow steps, the number of unknown variables, when to switch from what equation set to another, when to activate violated controlled quantities as explicit equations. Also, he/she is free to choose a polar or rectangular power flow formulation.

4 Code generation approach B: Power flow code generated from a generic structural domain specification

In section 2 of this tutorial, we have highlighted the main advantages and steps of programming the skeleton of a generic concept. For this generic concept we intend to write a Maple V code which realizes, instantiates the final power flow code from a set of parameters related to the structures of the power flow domain.

In this section, we show step by step what formal development steps have to be undertaken until we can obtain an executable power flow code.

4.1 Step 1: Domain analysis and generalization

This paragraph presents the result of domain analysis. This was done by analyzing the power flow problem which is a typical instance of the domain. A common data format definition of a solved power flow was used [13] to figure out requirements. The structure of data and flow of information within the domain are identified and described. It is shown how the different entities contribute to the overall nonlinear equation system \( \vec{g}(\vec{x}, \vec{p}) = 0 \) where \( \text{dim } [\vec{x}] = \text{dim } [\vec{g}] \), \( \vec{x} \) is the vector of unknown variables, \( \vec{p} \) is the vector of known parameters.

A network is a set of connected entities called Conducting Equipment and Connectivity Nodes. A Conducting Equipment has \( n \) terminals. Conceptually, terminals can be connected to each other via the Connectivity Nodes. A Conducting Equipment entity has exactly \( n \) terminals where \( n \) is an integer number greater than zero. Note, however, that a physical device which has 3 physical connections does not necessarily have 3 terminals. Its number “\( n \)” is identical with the number of real variables which are accessed by Connectivity Nodes.
4.1.1 Terminals

Connectivity Nodes access variables located in Conducting Equipment. The access path to variables is called a connection. Each connection is established by adding a terminal to a Conducting Equipment. Each terminal holds two data items: The appropriate identifier $k$ of the associated Conducting Equipment and an index $j$, which is part of a key to a variable located in a Conducting Equipment $k$.

4.1.2 Variable

A variable is an entity related to a component $x_m$ of unknowns $\bar{x}$ within the system of functions $\bar{g}(\bar{x}, \bar{p})$. It is represented on the computer as a real type value. One key idea of this model is, that the numerical value $x_m$ is accessible from two different view points. The first view point is associated with the order of variables used in the linear system solver of the Newton-Raphson solution of $\bar{g}(\bar{x}, \bar{p}) = 0$. Access to the variable value is made by the index $m$ which is stored on variables. The second view point is related to equations and variables local to Conducting Equipment and Connectivity Nodes. The function $V_k(basename, index)$ returns the numerical value of the variable identified by a string basename and an optional integer value index. Both arguments build the key to a variable in the naming space of the Conducting Equipment $k$.

4.1.3 Parameter

A parameter is an entity related to a known value stored in Conducting Equipment and Connectivity Nodes. It is a component of known values $\bar{p}$ within the system $\bar{g}(\bar{x}, \bar{p})$. It is represented on the computer as a real type value.

4.1.4 Conducting Equipment

Conducting Equipment is the entity typically associated with an electrical element such as a synchronous machine (generator), an energy consumer (load), AC transmission line etc. It can be defined as a finite state machine with a behavior $\mathcal{BN}_\sigma(V_\sigma, P_\sigma, E_\sigma, T_\sigma)$ for each state $\sigma$. Opposite to Connectivity Node the behavior of the Conducting Equipment is known at specification time. The $ng_\sigma$ equations ranging from $q$ to $q + ng_\sigma - 1$ are specified independently of the embedding network. They are formulated with a known set $\mathcal{V}$ of $nv_\sigma$ variables ranging from $s$ to $s + nv_\sigma - 1$ and parameters. The behavior $\mathcal{BN}_\sigma$ in a state $\sigma$ contains the following sets:

\[ V_\sigma = \{ x_3, x_5, \ldots, x_{s+nv_\sigma-1} \} \]
\[ P_\sigma = \{ p_1, p_2, \ldots, p_{ng_\sigma} \} \]
\[ \mathbb{E}_{\sigma} = \{ \quad g_q(V_{\sigma}^g, P_{\sigma}^g) = 0, \]
\[ g_{q+1}(V_{\sigma}^{g+1}, P_{\sigma}^{g+1}) = 0, \]
\[ \ldots, \quad g_{q+n_{g\sigma}-1}(V_{\sigma}^{g+n_{g\sigma}-1}, P_{\sigma}^{g+n_{g\sigma}-1}) = 0 \} \]
\[ \mathbb{T}_{\sigma} = \{ \quad v - p_u \geq 0 \Rightarrow \text{next state} = \sigma_1, \]
\[ p_l - v \geq 0 \Rightarrow \text{next state} = \sigma_2 \} \]

Parameter \( p_u, p_l \in \mathbb{P}_\sigma \), where \( p_u > p_l \) and variable \( v \in \mathbb{V}_\sigma \). For a state \( \sigma \) there are a maximum of two next states \( \sigma_1 \) and \( \sigma_2 \). If no condition for a transition is satisfied it is assumed that the state does not change. If a condition for a transition is satisfied, the Conducting Equipment propagates a change request to the owner object (the network). This object in turn enables the Conducting Equipment to go to the specified next state \( \sigma_i \). Then, the new behavior will be \( \mathcal{B}\mathcal{N}_{\sigma_i}(\mathbb{V}_\sigma, \mathbb{P}_\sigma, \mathbb{E}_{\sigma_i}, \mathbb{T}_{\sigma_i}) \). \( \mathbb{T}_{\sigma} = \{} \) designates a state without next state(s).

These state descriptions allow the formulation of changes of sets of equations for certain Conducting Equipment conditions. This is especially important in a practical power flow formulation where for example generators can change from PV- to PQ-type due to limited Q or where remotely controlled branch quantities such as the active power flow can become active, etc.

### 4.1.5 Connectivity Node

A Connectivity Node \( s \) accesses variables located in Conducting Equipment to create further subsets of equations \( \sum_{k=1}^{n_g} g_k(basename, \sigma_{k}, [par], \mathbb{A}) \) \( \sigma_{k}^g \)
\[ \sum_{k=1}^{n_g} \mathbb{V}_{k}(basename, j_i) + [lv] + [par] = 0 \]

2. **MultiRowSum(basename, A):**
\[ \mathbb{V}_{k_1}(basename, j_1) - \mathbb{V}_{k_2}(basename, j_2) = 0 \]
\[ \mathbb{V}_{k_2}(basename, j_2) - \mathbb{V}_{k_3}(basename, j_3) = 0 \]
\[ \ldots - \ldots = 0 \]
\[ \mathbb{V}_{k_{n-1}}(basename, j_{n-1}) - \mathbb{V}_{k_n}(basename, j_n) = 0 \]

The behavior \( \mathcal{B}\mathcal{S}(\mathbb{A}, \mathbb{E}) \) of a Connectivity Node has the following sets, whereby Sum stands for both types of sums:
\[ \mathbb{A} = \{ \text{terminal}_1, \text{terminal}_2, \ldots, \text{terminal}_n \} \]
\[ \mathbb{E} = \{ \quad g_q = \text{Sum}(basename_q, [lv_q], [par_q], \mathbb{A}), \]
\[ g_{q+1} = \text{Sum}(basename_{q+1}, [lv_{q+1}], \mathbb{A}), \]
\[ \ldots, \quad g_{q+n_{g\sigma}-1} = \text{Sum}(basename_{q+n_{g\sigma}-1}, [lv_{q+n_{g\sigma}-1}], \mathbb{A}) \} \]

In contrast to Conducting Equipment the behavior \( \mathcal{B}\mathcal{S} \) is not completely known at program specification time. The actual set of terminals \( \mathbb{A} \) is known only after the network data has been read.
4.1.6 Entities and system of equations

The overall system \( g(\bar{x}, \bar{p}) = 0 \) contains \( ng \) simultaneous nonlinear functions, depending on unknowns \( \bar{x} = \{x_1, x_2, \cdots x_{nv}\} \) and knowns \( \bar{p} \). Subsets of \( g \), \( \bar{x} \) and \( \bar{p} \) are stored and managed only in Conducting Equipment and Connectivity Nodes.

For the system the following assumptions are made:

1. All components of \( \bar{x} \) are continuous variables

2. The total number of variables \( nv \) is equal to the total number of functions \( ng \). This is a necessary but not a sufficient condition for the solution of a non-linear equation set.

3. The linearized components of \( \bar{g} \) are linearly independent functions.

4. \( \exists \frac{\delta g_i(\bar{x}, \bar{p})}{\delta x_j} \forall \{i, j\} \): All first partial derivatives of all equations with respect to all unknown variables exist. However, they can be zero.

4.2 Step 2: Formulation of domain architecture

This paragraph presents a generic, organizational structure for applications in the domain. It contains a design that satisfies the requirements specified in the domain model. Thus the domain model has impact on the elements of the architecture. The architecture can be adapted to create different applications within the domain. It defines a structure and a development process for configuring the object-oriented framework. An application means a set of specified behavior for each used type of Conducting Equipment and Connectivity Node. Further specifications define the correspondence to a selected network data interchange format [13].

The proposed architecture has three layers, see Fig. 4.

- Specification: In the first layer, the variability of applications are captured in specification data. This data controls the adaptation of generic parts of the framework.

- Adaptation: The second layer contains the commonality of applications in the domain. They are formulated by hard-coded classes within the object-oriented framework. The framework is C++ source code containing hard-coded and generated classes.

- Object-oriented framework: Within the third layer, a runtime system is built. An instance of the framework is created which reads user defined network data and outputs the result of one simulation step.
4.2.1 Specification

The behavior $\mathcal{BN}$ of each Conducting Equipment and $\mathcal{BS}$ of each Connectivity Node type is specified in a symbolical way by an appropriate Maple V [6] procedure, see also appendix C.

The procedure has the same name as the Conducting Equipment type or the Connectivity Node type together with the name part “Def”. This naming convention is used for later automatic Maple V code processing.

The procedure returns the behavior in two unified Maple V record formats: one for the $\mathcal{BN}$ and one for the $\mathcal{BS}$. These records define an intermediate format to the adaptation part of the architecture. Further parts of the specification are the set of all required types (Connectivity Node and Conducting Equipment) and information of how to read network data from a selected network data interchange format (for example the one of [13]).

4.2.2 Adaptation

A code-generating part customizes the generic parts of the framework. It is done by creating classes for each required Connectivity Node and Conducting Equipment type. Again Maple V is used to do this job and to generate Jacobian and function mismatch calculating C-source code. In addition, the constructor of the network class is built, depending on the selected network data interchange format. It reads network data from this interchange format and instantiates objects.

4.2.3 Framework

By applying object modeling techniques [14] the domain model was translated into ten hard-coded and two generic classes, AnyConductingEquipment and AnyConnectivityNode. Translation is complicated by two constraints:

- execution runtime efficiency
- ease of adaptation of generic parts: AnyConductingEquipment, AnyConnectivityNode and constructor of the network class

The strongly coupled classes demonstrated in Figs. 5 and 6 together build the key elements of the domain architecture.

On the top of Fig. 5 the most general classes are shown.

![Figure 5: Class hierarchy](image-url)
Lower positioned classes inherit data and behavior (methods) from the upper classes. Multiple inheritance was used to accumulate characteristics of parent classes (note: C++ has been chosen as the framework language).

Description of classes not mentioned in the model:

- **NetworkObj**: Provides a name for objects.

- **NewtonRaphson**: Virtual base class to declare the behavior of objects involved in the domain solution process. They all act on a solver object which encapsulates a linear sparse solver. The appropriate methods are:

  ```cpp
  virtual void EvalLinSysSize(int& size){}
  virtual void CountNonZeroes(int& nZ){}
  virtual void PutSolver(LinSolver& solver){}
  virtual void GetFromSolver(
            LinSolver& solver, double& maxMism ){;}
  virtual void SetFunStartLfNr(int& fnr){}
  virtual void PrintVarsAndPars(){}
  RVar* GetVar(char* name){...;}
  ```

- **Network**: is a container that stores objects derived from Conducting Equipment and Connectivity Node. There are methods to create and insert these instances and connections (terminals) between them. Further there is a method to start the calculation with a given mismatch tolerance and a maximum number of iterations. The constructor has to be customized to interpret a chosen interchange format of network data.

### 4.2.4 Mapping of locally defined behavior to the Newton-Raphson process

The Conducting Equipment and Connectivity Node assemble the overall system of simultaneous non-linear equations $\bar{g}(\bar{x}, \bar{p}) = 0$. The solution of this system is obtained by the well known Newton-Raphson process. The symbolic expressions of the linearized equation terms $A_{ij}$ and $b_i$ of a Newton-Raphson iteration are discussed in the following. Fig. 7 demonstrates the following assignments:
Figure 7: Structure of sparse linear equation system

**ConductingEquipment**

\[
A_{ij} = \begin{cases} 
  \frac{\delta g_i(V_k, P_k)}{\partial q_j} : q_k \leq i < q_k + n_g \\
  s_k \leq j < s_k + n_v \\
  0 : \text{else} 
\end{cases} 
\]

\[
b_i = \begin{cases} 
  g_i(V_k, P_k) : q_k \leq i < q_k + n_g 
\end{cases} 
\]

**ConnectivityNode**

\[
A_{ij} = \begin{cases} 
  1 : q_k \leq i < q_k + n_g \\
  x_j \text{ part of } OneRowSum_i(\ldots) \\
  \text{or } MultiRowSum_i(\ldots) \\
  -1 : q_k \leq i < q_k + n_g \\
  x_j \text{ part of } MultiRowSum_i(\ldots) \\
  0 : \text{else} 
\end{cases} 
\]

\[
b_i = \begin{cases} 
  OneRowSum_i(\ldots) \text{ or } \\
  MultiRowSum_i(\ldots) : q_k \leq i < q_k + n_g 
\end{cases} 
\]

*ui* is the negative correction value of the associated variable *xj*. Conducting Equipment and Connectivity Node update their maximum mismatch and variables.

C++ was chosen as main programming language. The sparse linear solver is taken from the Fortran77 “umf-package” [12]. Templates from the C++-LEDA library [15] are used for linked lists.

### 4.2.5 Development Environment

An executable file is generated within a UNIX environment. A UNIX “Makefile” has been used to coordinate the whole application generation process.

- First a library containing all Maple V procedures needed for specification and code generation is generated.
- The parts of the framework are customized accordingly.
All variable code parts are included in a default main program together with the hard-coded classes of the framework. The default main program simply creates an instance of a network and starts the methods Calculate and PrintVariables. Depending on the arguments the main program reads in the network data file of the selected network data interchange format.

4.3 Step 3: Specification of variable code pieces of power flow code

4.3.1 Specification of an example power flow code

A power flow application example is given in Fig. 8. It is an AC power flow problem for steady state conditions. The unknowns are formulated as complex numbers. The complex cartesian coordinate variables $e + j \cdot f$ (voltage), $ie + j \cdot if$ (current) and $P + j \cdot Q$ (power) are related to each electrical terminal of the elements SlackGen, SlackArea, PVGen, PQLoad, ImpLoad and Z (The names should be self-explanatory to people familiar with the power flow). These elements are mapped to types of Conducting Equipment. PVGen can change from PV- to PQ-type due to limited Q. Since $Q_{min} \leq Q \leq Q_{max}$ three states are necessary to cope with the different behavior.

Bus1 ... Bus5 are the interconnecting points of the network. The interchange area defines a region for which the sum of exported real power is a given value $P_{schedule}$. All entities called bus and power interchange area are modeled as types of Connectivity Nodes.

**Figure 8: Network example**

**Specification of Conducting Equipment types:**

**SlackGen (Network slack generator):**

\[
\begin{align*}
V &= \{ e_1, f_1, ie_1, if_1, P, Q \}, \ T = \{ \} \\
I &= \{ esl, fsl \} \\
E &= \{ e_1 - esl = 0, f_1 - fsl = 0, \\
&\quad P - esl ie_1 - fsl if_1 = 0, \\
&\quad Q + esl if_1 - fsl ie_1 = 0 \}
\]

**SlackArea (Area slack bus):**

\[
\begin{align*}
V &= \{ e_1, f_1, ie_1, if_1, P, Q \}, \ T = \{ \} \\
I &= \{ U \} \\
E &= \{ ie_1 - \frac{e_1 P + if_1 Q}{e_1^2 + if_1^2} = 0, if_1 - \frac{e_1 Q + P f_1}{e_1^2 + if_1^2} = 0, \\
&\quad U^2 - e_1^2 - f_1^2 = 0 \}
\]

**PQLoad (PQ-load):**

\[
\begin{align*}
V &= \{ e_1, f_1, ie_1, if_1 \}, \ T = \{ \} \\
I &= \{ P, Q \} \\
E &= \{ ie_1 - \frac{e_1 P + if_1 Q}{e_1^2 + if_1^2} = 0, if_1 - \frac{e_1 Q + P f_1}{e_1^2 + if_1^2} = 0 \}
\]

**PVGen (PV-generator):**
V_1 = \{ e_1, f_1, i e_1, i f_1, Q \} \\
P_1 = \{ P, U, Q_{\text{min}}, Q_{\text{max}} \} \\
E_1 = \{ i e_1 - \frac{e_1 P + f_1 Q}{e_1^2 + f_1^2} = 0, i f_1 - \frac{-e_1 Q + P f_1}{e_1^2 + f_1^2} = 0 \} \\
T_1 = \{ Q - Q_{\text{max}} \geq 0, \Rightarrow 2 \\
Q_{\text{min}} - Q \geq 0, \Rightarrow 3 \} \\
V_2 = \{ e_1, f_1, i e_1, i f_1 \}, T_2 = \{ \} \\
P_2 = \{ P, Q_{\text{max}} \} \\
E_2 = \{ i e_1 - \frac{e_1 P + f_1 Q_{\text{max}}}{e_1^2 + f_1^2} = 0, i f_1 - \frac{-e_1 Q_{\text{max}} + P f_1}{e_1^2 + f_1^2} = 0 \} \\
V_3 = \{ e_1, f_1, i e_1, i f_1 \}, T_3 = \{ \} \\
P_3 = \{ P, Q_{\text{min}} \} \\
E_3 = \{ i e_1 - \frac{e_1 P + f_1 Q_{\text{min}}}{e_1^2 + f_1^2} = 0, i f_1 - \frac{-e_1 Q_{\text{min}} + P f_1}{e_1^2 + f_1^2} = 0 \} \\
\textbf{Z (Impedance branch):} \\
V = \{ e_1, f_1, i e_1, i f_1, e_2, f_2, i e_2, i f_2, P_1, P_2 \} \\
P = \{ R, X \}, T = \{ \} \\
E = \{ i e_1 - \Re \left( \frac{e_1 + j f_1}{R + j X} - \frac{e_2 + j f_2}{R + j X} \right) = 0, \\
i f_1 - \Im \left( \frac{e_1 + j f_1}{R + j X} - \frac{e_2 + j f_2}{R + j X} \right) = 0, \\
i e_2 + i e_1 = 0, \\
i f_2 + i f_1 = 0, \\
P_1 - \Re ( (e_1 + j f_1) (ie_1 + j if_1)^*) = 0, \\
P_2 - \Re ( (e_2 + j f_2) (ie_2 + j if_2)^*) = 0 \} \\
\textbf{Specification of Connectivity Node types} \\
\textbf{Bus:} \\
S = \{ \text{OneRowSum(basename = i e)}, \ \\
\text{OneRowSum(basename = i f)}, \ \\
\text{MultiRowSum(basename = e)}, \ \\
\text{MultiRowSum(basename = f)} \} \\
\textbf{PowerInterchangeArea:} \\
S = \{ \text{OneRowSum( basename = P, par = P_{schedule})} \} \\
\text{Specification of network instantiation, parametrization and results are omitted for space reason.}

\textbf{4.3.2 Aspects of specification code and generated code pieces} \\

After implementation of a specific power flow model the following concrete information can be given: 

The specification of the variable parts of a power flow code uses 160 lines of Maple V code, the generated C++ code has a total length of about 1400 lines of which 570 lines are generated in the adaptive part of the Class "Network" (see Fig. 5), 250 lines of code in all classes of "AnyConnectivityNode" and 580 lines of C++ code in all classes of "AnyConductingEquipment". In contrast to these 1400 lines of adaptive C++ code, the total C++ code which includes also the hard-coded parts has 1900 lines. This C++ code does, however, not include the source code for the solution of the sparse linear system of equations and general purpose classes such as lists.
The Maple V code to generate the 1400 lines of adaptive C++ code has a total length of 700 lines including the 160 lines of specification. After compilation, the application creates 69 variables and functions. It converges after 4 Newton-Raphson iterations.

Note that Maple V allows the use of \( \Re \) (Real part) or \( \Im \) (Imaginary part) within the specification part.

Specifications of Conducting Equipment- and Connectivity Node types have been implemented for other power flow problems such as a polar coordinate power flow formulation.

The proposed domain architecture was tested in a power flow application by applying a 2500-bus network where the system has 5600 Conducting Equipment and creates about 45000 variables.

To start applying the proposed techniques the following tools and libraries are needed: An operating system supporting the make-tool, GNU g++-compiler/linker, a Fortran90-compiler, a Fortran77 to C converter (f2c) and Maple V. Further the publicly available libraries LEDA, UMFPACK, HARWELL and BLAS are needed. Both source-code of the framework, the elements of the architecture and the specifications of Conducting Equipment/Connectivity Node suited for a loadflow application are ASCII-files.

5 Conclusions

5.1 Automatic code differentiation

In this tutorial we have described a novel approach A for developing power flow software which should satisfy many industrial requirements. The key aspect is the application of program differentiation and code generation in power flow algorithms, in combination with the ADIFOR or ADIC tool.

It allows the power flow developer to specify standard Fortran77 or C code representing any set of power flow functions using any number of case statements, loops, etc. This set of functions is compiled/interpreted by ADIFOR / ADIC and a new Fortran or C code is generated which includes the efficient computation both of the user specified functions and associated non-zero Jacobian terms. The program developer can specify what the dependent and independent variables of the functions are for which derivative code should be computed. In addition to this feature, ADIFOR or ADIC include the capability of inherent sparsity detection of the Jacobian related code parts. It is the combination of Jacobian code generation and inherent sparsity consideration which allows to develop extremely powerful power flow codes.

From a developer point of view the main strength of the resulting power flow code lies in the fact that complex data dependencies between controlled and controlling quantities can be built into the power flow equation set without worrying about the Jacobian term computation.

The flexibility and efficiency in the maintenance and enhancement of the code is obvious: to add a new feature the user needs only to add the new equations in the power_flow.f Fortran77 subroutine, declare the new variables, append them in the script file ADIFOR_CONTROL.adf, and initialize them in the seed matrix related code part.

Tools using ACD techniques guarantee correct Jacobian related code parts and relieve the power system applications developer from code dependency and sparsity related problems which till now could only be solved by experienced and hard-to-find specialists.
5.2 Framework, domain software engineering with code generation as a key component

In this tutorial we propose a second power application software development approach B which uses an object-oriented framework as a key component of a reuse-oriented architecture for the domain.

Both the framework and the architecture have evolved in an iterative development process including steps of analysis, abstraction, model enhancement and update. It turned out that a network can be modeled by instances of the two generic types Conducting Equipment and Connectivity Node.

The adaptation of generic parts has been done in two steps. In a first step the symbolical behavior specification of Conducting Equipment and Connectivity Node types is captured using Maple V as environment and language. Then code-generating functions customize the generic parts of the framework. This is done by creating classes for each required Connectivity Node and Conducting Equipment type.

The second step adds the constant parts of the domain. This allocation of responsibilities simplifies the development of the code-generating part which is a task complicated by manifold aspects. The payoff of this approach is a high reduction in the effort to develop software within this complex domain.

Thereby a first advantage is the enhanced modeling flexibility of the proposed approach by Conducting Equipment and Connectivity Nodes:

- ease of modeling of entities with more than two terminals, such as three phase networks, substations, mutually coupled lines etc.
- ease of zero impedance modeling of closed switches

The main disadvantage of the modeling approach is the higher dimension of the linearized equation system.

- more equations and variables
- increased number of instantiated objects

Due to the object-oriented framework together with the proposed modeling approach we strongly believe that a second advantage of the proposed domain-architecture is that the specification and adaption part have become smaller and thus easier to use and to change by a reuser.

The framework is more than a class-library. It contains control flow and protocols, allowing to reuse the design of the solution for a class of problems. The framework itself contains much generic functionality, which has no longer to be provided by the adapting part. The proposed modeling approach is better suited for the object-oriented framework than the approach based on buses and branches. Due to decoupling the information of topology and behavior of entities the setup of Jacobian terms is much easier, simplifying the adaption part further.

5.3 Summary

This tutorial has presented two practical concepts of code generation to the area of power system applications. They differ, however, in many aspects and the effort to use them is quite different.

Common to both is the wish to provide better concepts and answers to the questions and problems raised in chapter 2 of this tutorial. The main points, however, are summarized: Code generation approaches in the area of power flow applications

- allow to keep sparsity,
allow to keep a high performance comparable to hand-coding,

allow to keep the power flow code as a generic power network problem solver,

allow easy and quick adaptation to new models,

allow to describe the power flow in a more compact way,

allow to concentrate on the parametric pieces of a power flow software,

lead to higher reuse of generic code pieces in other power application projects,

allow to parametrize generic code pieces and produce less generic or more specific code,

lead to more transparency, i.e. they lead to a process which is easier to comprehend, to document and to talk about,

allow modularization and adaptation to the capabilities of software development people,

allow better estimation of the time it takes to incorporate changes to existing power applications software,

allow to produce guaranteed correct derivative code,

allow the formalized change of an existing application to another, new language.

These advantages of code generation will be supported by technological advances in both software and hardware technology. The most promising development is the use of interpreted languages, where the present phases of compilation and linking of source code are not necessary any more. Parametrized versions of efficient numeric and flexible algebraic solution methods together with high level abstract software development architectures will be the future path for the development of power system applications.

References


Appendix

A Power flow equations

A.1 Introduction

This appendix describes the functional specification of a power flow equation set which we call an “explicit control power flow approach”. In this explicit power flow approach all controlled quantities which violate limit values are explicitly solved at the limit values during the power flow iterations. This leads to an explicit modelling of the associated controlling quantity as unknown variable. For example when a phase shifter transformer has the built-in capability to control the MW-flow through the transformer and this MW-flow violates a limit value, then the MW-flow at the controlled side is explicitly formulated as an equation and the associated phase shift is considered as a variable.

In this appendix the power flow equations of typical 1-port 1p and 2-port 2p network elements type are listed. The number of ports indicates the number of physical connections of the elements towards other elements. Note that the term port is not identical with the term “terminal” as used in the preceding section. Terminals indicate the number of access paths to variables of a network element.

To improve readability the index of variable vectors (e.g. \( V_{1p} \)) is omitted (\( V_{1p} \)). In figures (9,10) the symbols and the sign convention for currents are explained.

A.2 Power flow one-port elements (1p)

![One-port diagram]

\[
V^{1p} = V_{1p}^{1p} + jV_{1p}^{1p} \\
I^{1p} = I_{1p}^{1p} + jI_{1p}^{1p} \\
S^{1p} = P^{1p} + jQ^{1p}
\]

Figure 9: One-port

A.2.1 Generators

<table>
<thead>
<tr>
<th>Specified</th>
<th>( P_{G}^{(0)}, P_{L}^{(0)}, Q_{L}^{(0)}, V^{(0)} ) (or ( Q_{G}^{(0)} ))</th>
</tr>
</thead>
<tbody>
<tr>
<td>Variables</td>
<td>( V_{e}^{1p}, V_{f}^{1p}, I_{e}^{1p}, I_{f}^{1p}, Q^{1p} )</td>
</tr>
<tr>
<td>Equations :</td>
<td></td>
</tr>
<tr>
<td>1. Current mismatch equations</td>
<td></td>
</tr>
</tbody>
</table>

\[
\begin{align*}
I_{e}^{1p} \cdot \frac{V_{e}^{1p} V_{f}^{1p}}{(V_{e}^{1p})^2 + (V_{f}^{1p})^2} - G_{sh} \cdot V_{1p}^{1p} + B_{sh} \cdot I_{f}^{1p} &= 0 \\
I_{f}^{1p} \cdot \frac{V_{e}^{1p} V_{f}^{1p}}{(V_{e}^{1p})^2 + (V_{f}^{1p})^2} - B_{sh} \cdot V_{1p}^{1p} - G_{sh} \cdot I_{f}^{1p} &= 0
\end{align*}
\]

with: \( P^{1p} = -P_{G}^{(0)} + P_{L}^{(0)} = P_{1p}^{(0)} \) and \( Q^{1p} = -Q_{G} + Q_{L}^{(0)} \) and \( G_{sh} + j \cdot B_{sh} \) is the 1p shunt admittance.
2. an additional equation that fixes

- The voltage magnitude ($PV$—generators)
  \[(V^1_p)^2 + (V^1_f)^2 - (|V|^{(0)})^2 = 0 \quad (4)\]

- The reactive power of the bus ($PQ$—generators)
  \[Q^1_p + Q_G^{(0)} - Q_L^{(0)} = 0 \quad (5)\]

### A.2.2 Loads

<table>
<thead>
<tr>
<th>Specified</th>
<th>$P_L^{(0)}, Q_L^{(0)}$ ($S_G = 0$)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Variables</td>
<td>$V^1_p, V^1_f, I^1_p, I^1_f$</td>
</tr>
</tbody>
</table>

Equations:

1. Current mismatch (3) with $P^1_p = I_L^{(0)}, Q^1_p = Q_L^{(0)}$

### A.2.3 Slack

| Specified       | $P_L^{(0)}, Q_L^{(0)}, |V|_{\text{slack}}^{(0)}, \phi_{\text{slack}}^{(0)}$ |
|-----------------|---------------------------------------------------------------|
| Variables       | $V^1_p, V^1_f, I^1_p, I^1_f, P^1_p, Q^1_p$                    |

Equations:

1. Current mismatch (3) with $P^1_p = -P_{\text{slack}} + P_L^{(0)}, Q^1_p = -Q_{\text{slack}} + Q_L^{(0)}$ ($P_{\text{slack}}, Q_{\text{slack}}$ generation of slack)

2. Voltage magnitude ($|V|_{\text{slack}}$) and angle ($\phi_{\text{slack}}$) are specified:

\[
\begin{align*}
V^1_e - |V|_{\text{slack}}^{(0)} \cdot \cos(\phi_{\text{slack}}^{(0)}) & = 0 \\
V^1_f - |V|_{\text{slack}}^{(0)} \cdot \sin(\phi_{\text{slack}}^{(0)}) & = 0
\end{align*}
\]

### A.3 Power flow two-port elements (2p)

<table>
<thead>
<tr>
<th>Port</th>
<th>Description</th>
<th>Relationship</th>
</tr>
</thead>
<tbody>
<tr>
<td>$V^1_{p-1}$</td>
<td>$V^2_{p-1}$ + $j \cdot V^2_{p-1}$</td>
<td>$V^2_{p-2}$ = $V^2_{p-1}$ + $j \cdot V^2_{p-2}$</td>
</tr>
<tr>
<td>$I^1_{p-1}$</td>
<td>$I^2_{p-1}$ + $j \cdot I^2_{p-1}$</td>
<td>$I^2_{p-2}$ = $I^2_{p-1}$ + $j \cdot I^2_{p-2}$</td>
</tr>
<tr>
<td>$S^1_{p-1}$</td>
<td>$P^2_{p-1}$ + $j \cdot Q^2_{p-1}$</td>
<td>$S^2_{p-2}$ = $P^2_{p-1}$ + $j \cdot Q^2_{p-2}$</td>
</tr>
</tbody>
</table>

Figure 10: Two-port

Current mismatch equations:
A.3.1 Transmission Lines (AC)

| Variables | $I_e^{2p-1}$, $I_f^{2p-1}$, $I_e^{2p-1}$, $I_f^{2p-1}$, $I_e^{2p-2}$, $I_f^{2p-2}$, $I_e^{2p-2}$, $I_f^{2p-2}$ |

Equations:

1. Current mismatch equations

\[
\begin{align*}
I_e^{2p-1} - G_i \cdot V_e^{2p-1} + (B_i + B_c) \cdot V_f^{2p-1} + G_i \cdot V_e^{2p-2} - B_i \cdot V_f^{2p-2} &= 0 \\
I_f^{2p-1} - (B_i + B_c) \cdot V_e^{2p-1} - G_i \cdot V_f^{2p-1} + B_i \cdot V_e^{2p-2} + G_i \cdot V_f^{2p-2} &= 0 \\
I_e^{2p-2} - G_i \cdot V_e^{2p-1} - B_i \cdot V_f^{2p-1} - G_i \cdot V_e^{2p-2} + (B_i + B_c) \cdot V_f^{2p-2} &= 0 \\
I_f^{2p-2} + B_i \cdot V_e^{2p-1} + G_i \cdot V_f^{2p-1} - (B_i + B_c) \cdot V_e^{2p-2} - G_i \cdot V_f^{2p-2} &= 0
\end{align*}
\]  

(7)

where $G_i + j \cdot B_i$, $(B_i < 0)$ is the series admittance and $jB_c$, $(B_c > 0)$ the shunt capacitive admittance of the $2p$.

A.3.2 Transformers (Electrical model)

| Variables | $V_e^{2p-1}$, $V_f^{2p-1}$, $I_e^{2p-1}$, $I_f^{2p-1}$, $I_e^{2p-2}$, $I_f^{2p-2}$, $I_e^{2p-2}$, $I_f^{2p-2}$, $t_E$, $t_F$ |

1. Current mismatch equations (for IEEE common format transformer model [13])

\[
\begin{align*}
I_e^{2p-1} - \frac{G_i}{t_E} \cdot V_e^{2p-1} + (B_i + B_c) \cdot V_f^{2p-1} + \frac{G_i}{t_E} \cdot V_e^{2p-2} - B_i \cdot V_f^{2p-2} &= 0 \\
I_f^{2p-1} - (B_i + B_c) \cdot V_e^{2p-1} - \frac{G_i}{t_E} \cdot V_f^{2p-1} + (B_i + B_c) \cdot V_f^{2p-2} &= 0 \\
I_e^{2p-2} - (B_i + B_c) \cdot V_e^{2p-1} - \frac{G_i}{t_E} \cdot V_f^{2p-1} - (B_i + B_c) \cdot V_f^{2p-2} &= 0 \\
I_f^{2p-2} + (B_i + B_c) \cdot V_e^{2p-1} + \frac{G_i}{t_E} \cdot V_f^{2p-1} - (B_i + B_c) \cdot V_f^{2p-2} &= 0
\end{align*}
\]  

(8)

with $t_E = \cos(\phi_t)/|t|$, $t_F = \sin(\phi_t)/|t|$ and $\phi_t = \angle t$ (the use of $t_E, t_F$ as variables simplifies the equations compared to the ones using $t_r = Re\{t\}$, $t_im = Im\{t\}$)

2. additional equations that are activated when certain conditions (different per 2p type) occur. A transformer can be:

(a) Fixed tap transformer (type 1)
(b) Regulating transformer with voltage control (type 2)
(c) Regulating transformer with MVAR flow control (type 3)
(d) Regulating transformer with MW flow control (type 4)
**Fixed tap** (type 1) transformer has fixed voltage ratio $|t|$ and fixed phase angle (non-regulating) so the 'tap related' $t_E,t_F$ remain constant and equal to given values of the tap amplitude (voltage ratio) and phase angle. Although constant for type 1 transformers $t_E,t_F$ can be included in the set of variables with the cost of two more equations per type-1 transformer that 'fix' the tap to the specified value $|t|^{(0)} \angle \phi_t^{(0)}$:

$$
\begin{align*}
  t_E \cdot |t|^{(0)} - \cos(\phi_t^{(0)}) &= 0 \\
  t_F \cdot |t|^{(0)} - \sin(\phi_t^{(0)}) &= 0
\end{align*}
$$

The regulating transformers (types 2,3,4) have $t_E,t_F$ as variables. They are initialized to values based on the tap data using the equations (9). This set of equations keeps the tap (amplitude and angle) constant until the transformer changes its control mode. That means that the controlled quantity has violated a limit and has to be set to this violated limit.

(a) A **voltage control transformer** holds a bus voltage (remote or one of the transformer terminals) within specified limits by means of variable tap amplitude. For the case where the voltage of the controlled bus $i$ violates the limit $V_{lim}^i$ (maximum or minimum) the transformer switches to another 2p-type and instead of equations (8) and (9) the equations (10) are solved.

$$
(V_{1p}^i)^2 + (V_{1p}^f)^2 - (V_{lim}^i)^2 = 0
$$

$$
t_F - t_E \cdot \tan(\phi_t^{(0)}) = 0
$$

(b) In the same way (variable tap amplitude) a **MVAR flow controlling transformer** sets the MVAR flow ($Q_{2p}^p = V_{2p}^f \cdot I_{2p}^{e-p} - V_{2p}^e \cdot I_{2p}^f$ if measured at $\hat{m}, m = 1,2$) to the violated limit $Q_{lim}^{2p}$

$$
V_{2p}^f \cdot I_{2p}^{e-p} - V_{2p}^e \cdot I_{2p}^f - Q_{lim}^{2p} = 0
$$

and the angle remains fixed (second equation of (10)).

(c) A **MW flow regulating transformer** holds the MW flow of the branch within limits by means of variable phase angle (phase shifting). For the case where the MW flow measured at $\hat{m}$ hits a limit the new set of equations that replace (8) and (9) are equations (12) that fix the MW flow and the tap amplitude:

$$
V_{2p}^e \cdot I_{2p}^{e-p} + V_{2p}^f \cdot I_{2p}^{f-p} - P_{lim}^{2p} = 0
$$

$$
t_E^2 + t_F^2 - 1/|t|^{(0)}|^2 = 0
$$

A.4 Kirchhoff laws: Connect 1p, 2p

\[\text{Figure 11: Connection of 1p, 2p}\]
For every 2p the port voltages at each of the two ports are identical with the voltages of the two 1p that are also connected to it (see Fig. 11). So for the 2p i that is connected to 1p’s i1 and i2 the following equations are formulated:

\[
\begin{align*}
V_e^{2p-1}(i) - V_e^{1p}(i1) &= 0 \\
V_f^{2p-1}(i) - V_f^{1p}(i1) &= 0 \\
V_e^{2p-2}(i) - V_e^{1p}(i2) &= 0 \\
V_f^{2p-2}(i) - V_f^{1p}(i2) &= 0
\end{align*}
\]  

(13)

Furthermore the Kirchhoff law must be satisfied for every node (1p). For every 1p i the Kirchhoff equation ensure that the sum of the real and imaginary part of all currents is zero, where \( k1 \) are all 2p’s whose \( \hat{1} \) is i and \( k2 \) are all 2p’s whose \( \hat{2} \) is i.

\[
\begin{align*}
I_e^{1p}(i) + \sum_{k \in k1} I_e^{2p-1}(k) + \sum_{k \in k2} I_e^{2p-2}(k) &= 0 \\
I_f^{1p}(i) + \sum_{k \in k1} I_f^{2p-1}(k) + \sum_{k \in k2} I_f^{2p-2}(k) &= 0
\end{align*}
\]  

(14)

\section*{A.5 Remote Control}

The special feature of remote voltage control (RVC) can be incorporated in the power flow calculation. We consider the case where the voltage of a load 1p \( L1 \) is controlled by the generating units 1, 2, ..., \( N_{RVC} \). Each one of these generators loses control over its own voltage and only the MW generation \( P \) is specified, whereas the controlled 1p \( L1 \) has \( P, Q \), and \( |V| \) fixed. Two new 1p types are introduced: the P-type, and PVQ-type. For every voltage controlled 1p (PVQ-) there is a corresponding P 1p with specified \( P(0) = \sum_{i=1}^{N_{RVC}} P(i)(0) \). The voltage at \( L1 \) is regulated by means of the total \( Q \) injection \( \sum_{i=1}^{N_{RVC}} Q(i) \) of the regulating generators and this total reactive generating power forms a new variable \( Q_{genRVC} \). Since no explicit information is provided for the allocation of \( Q_{genRVC} \) we assume either proportionally to the active generation or to the Q-limit range. For the \( i \) generator \( (i = 1, ..., N_{RVC}) \) an equation of type (15) is formed.

\[
Q_{pv}(i) + Q_l(i) - d(i) \cdot Q_{genRVC} = 0
\]  

(15)

where \( Q_{pv}(i) \) is the (total) reactive power of the 1p, \( Q_l(i) \) the reactive load and \( d(i) \) the participating coefficient of generator \( i \) to the voltage regulation of \( L1 \) \( (\sum_{i=1}^{N_{RVC}} d(i) = 1) \).

For the voltage regulated 1p \( L1 \) an equation of type (16) fixes its voltage to the desired value \( V_{des}(0) \).

\[
(V_e^{1p})^2 + (V_f^{1p})^2 - (V_{des}(0))^2 = 0
\]  

(16)

In every iteration the equation (15) is formed for every generating unit which participates in the voltage control of \( L1 \) and which is still a PV-1p. This means that it has not violated a Q-limit and therefore the reactive power of the 1p \( (Q_{pv}) \) is a free variable.

If \( N_{qRVC} \) of the \( N_{RVC} \) generators \( (N_{qRVC} <= N_{RVC}) \) have violated a Q-limit and have been switched to PQ-type, then Q-allocating equation (15) will be activated only for the \( N_{RVC} - N_{qRVC} \) PV-generators. In the case \( N_{qRVC} = N_{RVC} \) none of the generating units has free reactive generation and so the equation (16) is omitted.

Generalizing the problem further we assume that \( N_{reg} \) 1p’s \( (L_1, L_2, ..., L_{N_{reg}}) \) are voltage regulated by (remote) generating units then the same procedure described above is applied. A two-dimensioned
matrix is built that contains the generators for the voltage regulation of \( \mathbf{1} \mathbf{p} \) \( L_i \) \( (i = 1, \ldots, N_{\text{reg}}) \) in the \( i \)th row.

### A.6 Area interchange

![Figure 12: Areas](image)

If the power flow study involves interconnected control areas the MW flow between areas is of great interest. There are cases where the MW exchange between areas should meet a prespecified net value, or should be held within limits. In this section power flowing out of an area is positive.

#### A.6.1 Calculation of area MW export

The MW export of an area \( A_i \) is equal to the sum of MW flows over the tie lines, the lines (2p’s) that connect this area with any other area \( A_j, j \neq i \). In the IEEE common format [13] TIE LINE data are defined where the metered side for every tie line (the side at which the tie flow is measured) is defined. This fact has to be considered while adding the tie flows. For every tie line that connects areas \( A_i, A_j \) (see Fig. (13)):

![Figure 13: Tie line](image)

if metered side \( \equiv 1 \) (area \( A_i \))

\[
\begin{align*}
\mathbf{P}_{\text{tie}}^{(1)} &= V_e^{2p-1} \cdot I_e^{2p-1} + V_f^{2p-1} \cdot I_f^{2p-1} \\
\mathbf{MW}_{\text{area}}^{\text{exp}}(A_i) &= \mathbf{MW}_{\text{area}}^{\text{exp}}(A_i) + \mathbf{P}_{\text{tie}}^{(1)} \\
\mathbf{MW}_{\text{area}}^{\text{exp}}(A_j) &= \mathbf{MW}_{\text{area}}^{\text{exp}}(A_j) - \mathbf{P}_{\text{tie}}^{(1)}
\end{align*}
\]

(17)
else if metered side $\equiv \overline{2}$ (area $A_j$)

$$P^{(2)}_{tie} = \frac{V^2_{e}I^2_{e}}{V^2_{f}I^2_{f}} + 2$$

$$MW^{\text{area}}(A_i) = MW^{\text{area}}(A_i) - P^{(2)}_{tie}$$

$$MW^{\text{area}}(A_j) = MW^{\text{area}}(A_j) + P^{(2)}_{tie}$$

Then for every area $A_k$ we solve the equation

$$MW^{\text{area}}(A_k) - P^{\text{free}}_{\text{exp}}(A_k) = 0, \quad k = 1, \ldots, N_{\text{area}}$$

A.6.2 Fixed area MW export

In the case where a fixed MW export of area $A_k (k = 1, \ldots, N_{\text{fix}})$ is desired the sum of all tie flows (taken at the correct tie line side and with the correct direction) for area $A_k$ is forced to the desired net power export value. The tie line flow summation for every area ($MW^{\text{area}}(A_k)$) is described in (17), (18) and equations (20) force the calculated sums to the desired export values $P^{\text{des}}_{\text{exp}}(A_k)^{(0)}$:

$$MW^{\text{area}}(A_k) - P^{\text{exp}}_{\text{exp}}(A_k)^{(0)} = 0$$

The $N_{\text{areas}}^{\text{fix}}$ equations (20) demand $N_{\text{areas}}^{\text{fix}}$ new unknown variables. These are the P generations of the area slacks (regulating generators) that are defined for every area. The area export control is therefore enabled by controlling the real power at one generator. Also, the fact must be taken into account that diverging cases are created by allowing two slacks (two free P generations) in one area.

The area slack is a new type of 1p: since the P is now a free variable a V- and a Q-1p type are introduced depending on whether area slack was a PV- or a PQ-generator respectively.

A practical power flow code handles the case where among $N_{\text{areas}}$ areas the export of $N_{\text{areas}}^{\text{fix}}$ ($N_{\text{areas}}^{\text{fix}} \leq N_{\text{areas}}$) is desired to be fixed.

1. $N_{\text{areas}}^{\text{fix}} = N_{\text{areas}}$: a set of only $N_{\text{areas}} - 1$ equations of type (20) will be built (the last one is linearly dependent since the prespecified MW export of all areas must be zero). In the code the MW fix equation for the area (area-SS) that contains the System Slack (SS) is omitted. The $N_{\text{areas}} - 1$ new variable P generations of the area slacks of all (but not area-SS) areas form the unknown variable vector $P^\text{area-slack}$.

2. $N_{\text{areas}}^{\text{fix}} < N_{\text{areas}}$: an equation of type (20) is solved for all $N_{\text{areas}}^{\text{fix}}$ areas including the area-SS if a fixed MW export in this area is desired. In that case the $P^\text{area-slack}$ variable vector will refer to the predefined (selected) area slack 1p. Note that the area of the SS contains a free P ($P$ of SS). The new variable (free P) must be the P of a generator belonging to an area which does not have a fixed desired export.

A.6.3 Losses allocation

For an area-oriented output of the power flow results, the branch MW losses must be assigned to the area to which the 2p belongs to. For that reason intermediate areas are introduced.

The intermediate area $A_iA_j$ contains the tie-lines that connect areas $A_i$ and $A_j, (j \neq i)$. For every 2p that connects areas $P^{\text{loss}}_{2p}(k)$ is allocated to the appropriate area:
\[ P_{\text{loss}}^{2p} = P^{2p-1} + P^{2p-2} \]

\[ SP_{\text{Loss}_{\text{area}}}(A_i) = \sum_{k \in A_i} P_{\text{Loss}}^{2p}(k) \]

\[ SP_{\text{Loss}_{\text{intm}}}(A_i A_j) = \sum_{k \in A_i A_j} P_{\text{Loss}}^{2p}(k) \] (21)
B  A Fortran77 code piece within the automatic code differentiation project

B.1 Input code: Code pieces of the power flow functions

    subroutine power_flow(mismatch)
    ....
    elseif (type_lp(i) .eq. SlGen) then
      c Slack lp
      c Computation the currents produced by the slack lp generator including the contribution of shunt elements
      c mismatch is the array for the numeric values of mismatches
      c ---------
      I_lp_e(i) = - ( - (e_lp(i) * P_slack + f_lp(i) * Q_slack) / (e_lp(i)**2 + f_lp(i)**2) -
                     G_b(i) * e_lp(i) + B_b(i) * f_lp(i))
      I_lp_f(i) = - ( - (f_lp(i) * P_slack - e_lp(i) * Q_slack) / (e_lp(i)**2 + f_lp(i)**2) -
                     B_b(i) * e_lp(i) - G_b(i) * f_lp(i))
      mismatch( k ) = e_lp(i) - e_lp_slack_0
      mismatch( k + 1 ) = f_lp(i) - f_lp_slack_0

B.2 Generated code: Code pieces of the power flow functions including code pieces for the sparse first order derivatives

    subroutine g_power_flow(mismatch, g_mismatch)
    ....
    else
      if (type_lp(i) .eq. slgen) then
        C Slack lp
        C Computation the currents produced by the slack lp generator including the contribution of shunt elements
        C mismatch is the array for the numeric values of mismatches
        C --------
        call dspg4q(g_d1_w, -f_lp(i), g_q_slack, -q_slack, g_f_1p(i), -e_lp(i), g_p_slack, -p_slack, g_e_1p(i))
        d1_w = -(e_lp(i) * p_slack + f_lp(i) * q_slack)
        d3_v = e_lp(i) * e_lp(i)
        d2_p = 2.0d0 * e_lp(i)
        d5_v = f_lp(i) * f_lp(i)
        d1_p = 2.0d0 * f_lp(i)
        d6_v = d3_v + d5_v
        d7_v = d1_w / d6_v
        d10_b = -(d7_v / d6_v)
        call dspg3q(g_i_1p_e(i), -b_b(i) + d10_b * d1_p, g_f_1p(i), g_b(i) + d10_b * d2_p, g_e_1p(i), -(1.0d0 / d6_v), g_d1_w)
        i_1p_e(i) = -(d7_v - g_b(i) * e_lp(i) + b_b(i) * f_lp(i))
      C--------
C

call dspg4q(g_d1_w, e_lp(i), g_q_slack, q_slack, g_e_lp(i) + i), -f_lp(i), g_p_slack, -p_slack, g_f_lp(i))

d1_w = -(f_lp(i) * p_slack - e_lp(i) * q_slack)

d3_v = e_lp(i) * e_lp(i)

d2_p = 2.0d0 * e_lp(i)

d5_v = f_lp(i) * f_lp(i)

d1_p = 2.0d0 * f_lp(i)

d6_v = d3_v + d5_v

d7_v = d1_w / d6_v

d10_b = -(-d7_v / d6_v)

call dspg3q(g_i_lp_f(i), g_b(i) + d10_b * d1_p, g_f_lp(i) + i), b_b(i) + d10_b * d2_p, g_e_lp(i), -(1.0d0 / d6_v), g_d1_w)

C--------
C

call dspglq(g_mismatch(k), 1.0d0, g_e_lp(i))

mismatch(k) = e_lp(i) - e_lp_slack_0

C--------

call dspglq(g_mismatch(k + 1), 1.0d0, g_f_lp(i))

mismatch(k + 1) = f_lp(i) - f_lp_slack_0

C--------
C

C  A Maple V framework definition code piece for the power flow domain

ConductivityEquipment[DefSlackGen] := proc(varsPerTerminal)
local funs, vars, pars, N, AnzFuns, AnzPars, typename;

typename := 'SlackGen';

N := 1;
NoFuns := 4; # Number of functions
NoPars := 2; # Number of internal parameters

vars := ExpandVector( AllTerminalVars( N, varsPerTerminal), [P, Q ]); # Expand the vector of
terminal variables

pars := vector(NoPars,[esl,fsl]); # The parameter symbols

tfunctions := vector(NoFuns); # Create function vector
funs[1] := ue1 - esl; # Define function 1
funs[2] := uf1 - fsl; # Define function 2
funs[3] := ie1 - (ue1\*P + uf1\*Q) / (ue1\*2 + uf1\*2); # Define function 3
funs[4] := if1 - (-ue1\*Q + uf1\*P) / (ue1\*2 + uf1\*2); # Define function 4

RETURN( vars, pars, funs, typename );
end:
D Acknowledgment

This tutorial is based on material to which the following ETH research assistants have contributed: Peter Bosshart, Tina Orfanogianni, Reto Hirt. The author wishes to acknowledge these persons for their discussion, documentation and software contributions within this code generation project. Parts of this project have been sponsored by the BBW (Bundesamt für Bildung und Wissenschaft), Bern, Switzerland (Project No. 95.0840) as part of the European community project “Electronet: High Voltage Electrical Network Information Exchange for Planning and Analysis” (Esprit No. 22297).

E Biography

Rainer Bacher received the Dipl.El.-Ing. degree in electrical engineering in 1982 and the Dr.sc.techn. degree in 1986, both from the Swiss Federal Institute of Technology (ETH) in Zürich, Switzerland. After his doctorate he joined the Energy Management Systems Division of Control Data Corporation in Minneapolis, U.S.A. In 1989 he joined Colenco Power Consulting AG, a Swiss consulting company. Also in 1989 he started as a lecturer and part-time senior researcher at the power system group at the ETH. In 1992 he became a full-time senior assistant and lecturer at the ETH. In 1993 he was appointed assistant professor of Energy Management Systems at the department of electrical engineering at the ETH Zürich. He is a member of technical committees for PSCC 93, PSCC 96 and PICA 93, PICA 95, PICA 97. He can be reached at Internet e-mail: Rainer.Bacher@eeh.ee.ethz.ch. His WWW home page is: http://www.eus.ee.ethz.ch/~bacher.