Abstract

First order derivative computations (Jacobian matrices) are required by power flow studies. In this paper the computation of the sparse Jacobian matrix is implemented using an automatic code differentiation tool for FORTRAN 77 code. The user only programs the code for the computation of the numeric mismatch values of all equations. The automatic generation of the derivative code includes also the automatic consideration of inherent sparsity of the Jacobian of the power flow equations. This approach allows to program as explicit equations also complex features such as remote voltage and area inter-change control, controlled quantity limits and the dynamic change from one equation set to another during the power flow iterations. The resulting power flow code is easy to enhance and to maintain, shows good execution speed and has been successfully applied to network systems as large as 2550 nodes.

Keywords - Automatic code differentiation, advanced power flow features

1 Introduction

For the solution of the non-linear power flow equations the iterative Newton-Raphson method is employed, i.e. per iteration a sparse, linear system

\[ J \cdot \Delta x = -g \]

is solved, where \( x \) is the vector of variables, \( g \) represents the power flow equation mismatches valid at the current iteration and \( J = \frac{\partial g}{\partial x} \) is the sparse Jacobian matrix associated to these equations.

Power flow problems have been successfully solved for many years and do not present a real algorithmic challenge any more. In this area, however, two software engineering related aspects still present a challenge to the power system application developers:

1. The incorporation of practical features such as the limit consideration of reactive generator power and the control of possibly remote quantities: The problem lies for example in the correct coding and modelling of transformers which have inherent capability to control voltage magnitudes, Mvar- or MW-flows. Another example for a controlled quantity is the area import or export: Often the value of MW which leaves or enters a specified area must be hold to a certain value. The incorporation of this feature is of strong practical value. For certain networks a power flow program has no practical value at all without these modelling features.

As of today two algorithmic approaches are seen for the incorporation of these “advanced” power flow features:

In approach A (called the “implicit control power flow approach”) only the standard power flow equations are explicitly solved to a certain accuracy. These standard equations include the Kirchhoff-laws (power or current nodal mismatch equations) and the voltage magnitude equations for the PV-nodes (only for rectangular system coordinate formulation). Most of the above mentioned control features are handled outside the solution of these standard power flow equations: Assuming that the standard power flow equations have been solved to a certain tolerance \( \epsilon \), the controlled quantities can be outside of allowed limits. If this happens the error (usually computed as the distance to the nearest violated limit value) is computed and the controlling quantity which represents a power flow input parameter in this standard power flow formulation A is modified using sensitivities or heuristic rules. Then, one or more iterations of the standard power flow equations are executed again, etc. until a) the power flow equations are solved, b) the controlled quantities do not violate given upper and lower limits and c) the controlling quantities are practically feasible, i.e. also within defined upper and lower limits. Only the satisfaction of all three points together represents a power flow result which has a chance of being acceptable for a practical environment.

In approach B (called the “explicit control power flow approach”), beside the above mentioned standard power flow equations, all controlled quantities which violate limit values are explicitly solved at the limit values during the power flow iterations and lead to the 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,
Wrong coding of these parts will lead in most cases to non-converging cases even if the Jacobian parts are correctly coded. One can note, however, that for example the incorrect coding of mismatch equation parts related to transformer taps (for example coding tap instead of 1/tap in the primitive admittance matrix) often leads to “reasonable” results, i.e. nodal voltages can be within the expected p.u.-value ranges.

As a consequence of these described problems a new concept is needed which helps to solve these problems. In this paper we describe how automatic code differentiation (ACD) is used during various stages in the development process of a power flow problem. To demonstrate the merits of ACD, the software tool “ADIFOR” [1] is used. ADIFOR has its main strength in the automatic generation of the first derivative code of (almost) any user programmed FORTRAN 77 code with user definable sets of derivative variables. This set of “derivative variables” is easily defined on a high usage level. In addition, the fact that not all derivative terms will be non-zero is considered automatically, i.e. sparsity will be detected without power flow developer action. These two features, i.e. automatic code generation of Jacobian terms of a given function (i.e. power flow equation) related code and the inherent use of sparsity makes this approach very powerful when applied to the power flow approach B mentioned above.

This paper assumes knowledge of power flow equations structure, types and problems related to approach B, see [2, 3]. Power flow equations are only mentioned in combination with examples to clarify the application of the ACD concept to the power flow approach B.

The paper starts with a general overview description of the ADIFOR technology and the underlying ACD methodology. Next, we describe the view of a power flow developer: What is his/her responsibility, what is automatically taken over by ADIFOR. Finally the approach has been applied to a concrete power flow design which includes many features relevant to approach B. This paper ends with a summary of autogenerated power flow code characteristics and convergence results of the executed code with large networks up to 2550 nodes.

2 ADIFOR: Technology assessment

2.1 Users point of view: Code generation capabilities

ADIFOR is a tool to provide automatic code differentiation for FORTRAN77 programs [1]. The user needs only to supply the source code (standard FORTRAN77 code that 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[4].

Assume a function $f$ with an $n$-vector $x$ as independent and an $m$-vector $y$ as dependent vari-
The ADIFOR-generated code will compute $\mathbf{g} \cdot \mathbf{y} = (\frac{\partial f}{\partial \mathbf{x}} \times \mathbf{g} \cdot \mathbf{x})^T = (\mathbf{J} \ast \mathbf{S})^T$ where $\mathbf{J}$ is the Jacobian matrix and $\mathbf{S}$ is the seed matrix. By proper initialization of $\mathbf{S}$ all elements of the transposed Jacobian or the transpose of a Jacobian-vector product can be obtained. The autogenerated 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 $<\text{var}>$.

### 2.2 Theory: Summary of 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 comparing to symbolic differentiation techniques [2].

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:

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

(2)

The two main approaches to ACD that have been traditionally developed are the forward and the reverse mode ([4], [5]). 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 [4].

ADIFOR employs 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:

**(in the original (input) routine)**

```
mismatch(indx+i) = V2_e + V2_f - V2_fix
```

**(in the generated routine)**

```
d2_v = Ve(i) * Ve(i)
d2_p = 2.0d0 * Ve(i)
d4_v = Vf(i) * Vf(i)
d1_p = 2.0d0 * Vf(i)
call dspg2q(g_mismatch(indx+i), d1_p, g_Vf(i), d2_p, & g_Ve(i))
mismatch(indx+i) = d2_v + d4_v - V2_fix
```

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 $Ve$ and $Vf$ of the right hand side. The function dspg2q is a SparsLinC routine (section 2.3) that computes

```
sparse_object(g_mismatch(indx+i)) =
  d1_p * sparse_object(g_Vf(i)) +
  d2_p * sparse_object(g_Ve(i))
```

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

### 2.3 User point of view: Automatic sparsity handling

ADIFOR allows 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 $S$ 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 the 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 provides 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 function calls.

Efficient and correct coding of these parts is very tricky, especially when applied to the power flow approach B:
There exist many code parts for every possibly binding controlled quantity with an associated possibly added controlling unknown variable. Each of these code parts leads 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 relieves 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 [1, 6]. 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 type variable that is a pointer to the integer representation of the corresponding variable. Using C allows to dynamically use memory locations when generating the non-zero terms of the Jacobian.

3 ADIFOR applied to the power flow approach B

3.1 Specifying Input for ADIFOR

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:

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 sin, cos, etc. Using ADIFOR, the user is free to use almost all expression and language capabilities of the standard FORTRAN 77 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,
             Ve_2p,Vf_2p,tap_e,tap_f,
             Q_pq,...

   AD_OVARS = mismatch
   AD_FLAVOR = sparse

   The option AD_IVARS contains a list of all vector names that represent independent variables and AD_OVARS the dependent variable mismatch. As the option AD_FLAVOR is set to sparse calls to SparsLinC routines will be invoked and sparse derivative code will be generated.

3. a “dummy” main program (powerflow_dummy.f) that simply calls the top level subroutine power_flow.f.

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 and all desired routines called by this top-level routine is generated (g_power_flow.f).

3.2 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.

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 \( \text{mismatch} \) and all non-zero values of the Jacobian matrix. These values appear to the power flow programmer as 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 [7] has been used in our implementation) together with the mismatch values.

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

These code pieces allow to create an executable version of a power flow program of type B 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 Software engineering and simulation results

The concept as described in the previous chapters has been realized at a power flow with the following features: Kirchhoff laws respected, use of rectangular coordinate nodal voltage variables, use of current mismatch equations leading to reactive generator power as unknown variable, PQ-load, PQ-generator, PV-generator, upper and lower Q-limits at generator loads, 1 Slack node generator with fixed voltage angle, change of PV-generator to PQ when meeting Q-limits (in the very final power flow result, \( V \) is allowed to be outside limits, but never \( Q \)), P-line model, admittance shunts model, transformer types 1 (fixed tap), 2 (regulating transformer with voltage control; if controlled voltage limit is violated, tap position is freed, however, never beyond upper and lower tap limits), 3 (regulating transformer with Mvar flow control; if controlled Mvar flow limit is violated, tap position is freed, however, never beyond upper and lower tap limits), 4 (regulating transformer with MW flow control; if controlled MW flow limit is violated, phase shift related tap position is freed, however, never beyond upper and lower tap limits), single bus remote voltage control by reactive powers \( Q \) of several generators (only to the extent that the reactive powers \( Q \) of all participating generators do not hit individual generator reactive power \( Q \) limits), area interchange active power control of all or a subset of specified areas of the whole network by area slack generator active powers \( P \).

These features indicate the cases which must be coded to compute the mismatches based on the various states of controlling and controlled variables. One can imagine that a program can be hand-coded which computes the Jacobian terms for all these cases. However, the management of such a code is very difficult especially when it has to be expanded to include new model features.

The code for power flow mismatch computation including all features mentioned above has a length of 1107 lines including output purpose statements (excluding comments). The corresponding length of the autogenerated code which includes both the computation of the mismatches and the computation of all sparse Jacobian terms is 1577 lines.

The additional memory requirements of the data sections of the autogenerated subroutine is 872 kB static allocated memory (the code was compiled for networks up to 3000 nodes, 3500 lines, 2000 transformers) and 5140 kB dynamically allocated sparse Jacobian matrix elements related memory (for the case of the 2550 node network). The dynamically allocated memory is measured using the SparsLinC \texttt{xspmem} command. Note that this number includes all memory needed for the computation of temporary variables for the computation of the Jacobian terms. A small disadvantage of the ADIFOR/SparsLinC approach can be noted: Compared to hand-coded sparse Jacobian matrices of power flow equation approach B, it becomes clear that the data sections of the autogenerated differentiated code use more memory. This disadvantage is, however, offset by the tendency of the computer industry to offer increasing amounts of memory for less money. Even more important, this negative side-effect of more memory consumption is offset by an extremely flexible and fast (development speed related) handling of code dependencies.

The 2550 node network includes 3400 branches and 20 areas. Several sets of power flow variables (explicit currents, currents eliminated) and associated power flow equations have been programmed. All automatically generated Jacobian codes have performed without code generation er-
The computation of the automatically generated Jacobian terms and the mismatch parts for a system with 5392 variables and 38131 non-zero Jacobian elements (this run used a power flow code implementation with all currents eliminated) takes 0.8 seconds per iteration (Sun Sparc Server 4000, 128 MB Memory, Solaris 2, UltraSparc based CPU, CPU clock rate 168 MHz, memory clock rate 83 MHz).

To allow comparison to well known power flow solution parts, the solution of the sparse linear system of equations including sparsity setup, symbolic and numeric factorization, forward and backward solution takes in average 0.4 seconds for one iteration using UMFPACK [7] as general purpose sparse package.

5 Conclusions

In this paper we describe a novel approach 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 tool. It allows the power flow developer to specify standard FORTRAN 77 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 and a new 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 includes 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 FORTRAN 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.

References


Tina Orfanogianni received the Dipl.El.-Eng. degree in electrical engineering in 1995 from the National Technical University of Athens (NTUA), Greece. She is currently doing her PhD in the area of power system optimization using FACTS devices. She can be reached at Internet e-mail: Tina.Orfanogianni@eeh.ee.ethz.ch

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 Ltd., 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 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, 96 and PICA 93, 95, 97. He can be reached at Internet e-mail: Rainer.Bacher@eeh.ee.ethz.ch