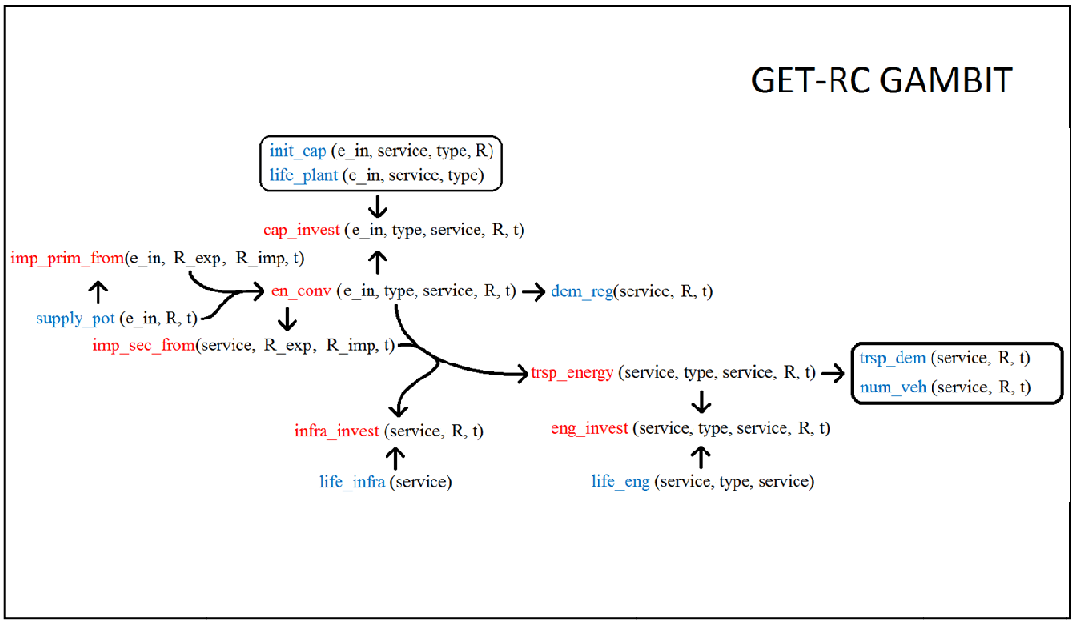
CHALMERS


GET-RC GAMBIT
init_cap (e_in, service, type, R)
life_plant (e_in, service, type)
cap_invest (e_in, type, service, R, t)
imp_prim_from(e_in, R_exp, R_imp, t)
en_conv (e_in, type, service, R, t)
dem_reg(service, R, t)
supply_pot (e_in, R, t)
imp_sec_from(service, R_exp, R_imp, t)
trsp_energy (service, type, service, R, t)
trsp_dem (service, R, t)
num_veh (service, R, t)
infra_invest (service, R, t)
eng_invest (service, type, service, R, t)
life_infra (service)
life_eng (service, type, service)

FURTHER PROGRAM DEVELOPMENT FOR THE COST MINIMIZING GLOBAL ENERGY SYSTEM MODEL GET-RC

Master of Science Thesis in the Industrial Ecology Programme, MPECO

MAGNUS ANDERSSON

Department of Energy and Environment
Division of Physical Resource Theory
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden, 2013
Report No. 2013:9

REPORT NO. 2013:9

CHALMERS UNIVERSITY
OF TECHNOLOGY

FURTHER PROGRAM DEVELOPMENT FOR THE COST MINIMIZING GLOBAL ENERGY SYSTEM MODEL GET-RC

MAGNUS C.M.K. ANDERSSON

Supervisor:
Maria Grahn

Examiner:
Sten Karlsson

Department of Energy and Environment
CHALMERS UNIVERSITY OF TECHNOLOGY
Göteborg, Sweden 2013

Further Program Development for the Cost Minimizing
Global Energy System Model GET-RC
MAGNUS C.M.K. ANDERSSON

Technical report no 2013:9
Department of Energy and Environment
Chalmers University of Technology
SE-412 96 Göteborg
Sweden
Telephone + 46 (0)31-772 1000

Cover:
The figure on the front cover is an overview of the new implementation of the GET-RC model (see page 37 chapter 5.7 for more information).

Chalmers Reproservice
Göteborg, Sweden 2013

Further Program Development for the Cost Minimizing Global Energy System Model GET-RC

MAGNUS C.M.K. ANDERSSON

Department of Energy and Environment
Chalmers University of Technology

SUMMARY

The linear programming Global Energy Transition (GET) model covers the global energy system and is designed to meet exogenously given energy demand levels, subject to a CO2 constraint, at the lowest system cost. The model can be used to better understand the role of different energy technologies in a future carbon constrained world and how the technologies fit into the larger global energy system, where different energy sectors compete for the same limited primary energy sources.

In this thesis a new optimization environment was put together for the cost minimizing global energy system model GET-RC. This new optimization environment was written in the functional programming language of SCHEME implementing the GAMBIT compiler. As a result of the reimplementation a mathematical interpretation or linear combination that mapped the GET-model was made.

The new model consists of only 7 variables but still can account for all functionality of the original model of about 45 variables. The model was then implemented into the new framework by a method based on six so called Simple models where the limited time frame of the project allowed for the first 3 models to be implemented in the new environment.

The solving time for the reimplemented simple model 3 was proven to be twice as fast as the original version of simple model 3.

The report is written in English.

Keywords: get-rc, cost, minimization, model, green, red, variables.

Acknowledgements

First I would first like to thank my supervisor Maria Grahn for helping me so much with this project, for her support, her interest in the small details of my work and her endless patience. I would also like to thank my friend Mikael More for invaluable help and advices regarding the C programming language and Scheme, especially in understanding functional programming compared to object based programming. Finally I would like to thank my examiner Sten Karlsson for his advices regarding how the report should be organised and his patience.

Contents

List of Figures

List of Acronyms

_P	Passenger Transport
_F	Freight Transport
0	Conventional Technology
AFR	Africa
Bio	Biomass
BTL	Bio-To-Liquid or bio-fuels
ccs	combustion with Carbon Capture and Storage
cg	combustion with co generation of heat
cg-ccs	A combined cg and ccs plant
CPA	Centrally Planned Asia — mainly China
CSP	Concentrated Solar Power
CTL-GTL	Coal-To-Liquid and Gas-To-Liquid
Elec	Electricity
EUR	Europe
FCV	Fuel Cell Vehicle
FSU	Former Soviet Union
HEV	Hybrid Electric Vehicle
ICEV	Internal Combustion Engine Vehicle
LAM	Latin America
MEA	Middle East
NAM	North America
NG	Natural Gas
PAO	OECD countries in the Pacific Ocean
PAS	Pacific Asia
PHEV	Plug-in Hybrid Electric Vehicle
SAS	South Asia — mainly India

1 Background and Introduction

The well and foundation of the industrial revolution was the use of energy coming from other source than physical labour of humans or animals, that is the use of fossil fuels (Ponting 2007). Some believe that this is the point in history where the human-race became too strong for nature to handle, the time when nature started to degrade due to our use.

Unfortunately this is not true. Degrading of the ecosystems is something humans have been doing possibly even back in the time of hunting and gathering (Ponting 2007). It is true however that the destructiveness precoded at unprecedented levels during the industrialisation (Grübler 1998).

It would not have been possible to reach where the world is today without the fossil fuels but as soon as the fossil fuels started to be used the human-raise had a mission. The resources of our planet are limited so the fossil fuels must run out one day as they are renewed at a much slower pace than they are consumed. Some resources are not renewed on Earth at all such as uranium. The mission is to find a solution as to what should be done when the fossil fuels are depleted. Today 80% of our energy use comes from fossil sources (IEA 2012). There are still however a lot of fossil fuels left.

Unfortunately another threat than the depletion of the fossil fuels are much more pressing. The human raise has been aware of this effect for hundreds of years due to astronomical studies but it was thought that it would not be a problem on Earth. Carbon dioxide is a gas that is transparent to the visible range of light but much less transparent to the longer wavelengths of heat radiation.

With carbon dioxide in the atmosphere of Earth this means that the energy radiating from the sun reaches the surface of Earth unobstructed by the atmosphere but that the resulting heat not as easily can leave. This is refereed to as the greenhouse effect. Gases which has this filtering effect on light is refereed to as greenhouse gases (de Pater 2001).

Earth has had greenhouse gases in the atmosphere for a long time, the most important are carbon dioxide, methane and water, which is the reason for why our surface temperature is as high as it is despite the long distance to the sun. An increase of the carbon dioxide content of Earth's atmosphere however will sooner or later result in an increase of Earth's surface temperature (de Pater 2001).

Another example where the greenhouse effect is perhaps more obvious is the example of our neighbour planets Mercury and Venus. Mercury has no atmosphere which gives it a very unstable surface temperature shifting between 90 and 700 kelvin degrees over a day. But no atmosphere also means no greenhouse effect. Venus on the other hand has an atmosphere consisting of 96% carbon dioxide. Although Venus is approximately twice as far away from the sun as mercury its surface temperature is still higher, about 730 Kelvin degrees.

It was initially thought that our carbon dioxide emissions would not result in an increase of the carbon dioxide content of the atmosphere. This mainly due to the large fluxes of carbon dioxide in and out of the oceans. But the oceans did not capture the carbon dioxide as quickly as expected so the concentration in the atmosphere increased.

In 1988 the World Meteorological Organisation (WMO) and the United Nations Environmental Programme (UNEP) established a scientific advisory body called the Intergovernmental Panel of Climate Change (IPCC). In 1990 IPCC presented their first report showing that the mean temperature of Earth has increased by 0.5 Celsius degrees in the last century and that it will rise further with 0.3 Celsius degrees per decade in the 21th century (Wayne 2009). Their fourth report presented in 2007 showed that this climate change with a 90% certainty was human caused (IPCC 2007).

1.1 Renewable Alternative Energy Systems

The end-use sectors in an Energy system can roughly be divided into two very distinct categories, stationary energy use and mobile energy use. The main difference is that energy carriers used in mobile applications need an on board storage system (e.g., a fuel tank or a battery). Energy carriers used in the stationary energy sector (e.g. electricity and heat) are however possible to use without energy storage facilities.

Both the stationary and the mobile energy sectors are today dependent on fossil fuels.

At a first glance the stationary energy sector seams the easiest to change, when fossil fuels need to be substituted, as many alternatives already exists. Some of them invented even before the development of the electricity net. Wind, hydro and in some regions even solar energy. These forms of energy harvest have been developed over many years and are now in large scale use. A problem concerning wind and solar power are the availability discussed in more detail in the next section.

Another alternative energy source is bio-energy. The problem with bio-energy is the limited scale of the production. The two main limitations to the scale are the overall limited amount of land and the need to use land for other purposes than bioenery, such as produce food and feed for animals as well as produce timber, paper and textile fibres.

A great concern regarding the population growth is the limited food supply, even without an increase in bio-energy use it will be a challenge to grow food for 9 or 10 billion people (Godfray 2010).

This poses a problem for the idea of using bio-energy. Many different assessments of this has been made with different focuses and different results. In this report a global maximum of 200 EJ for the bio-energy is assumed based on a previous assessment (Grahn M 2009), (Berndes et al, 2003). In that is included 100 EJ of primary harvesting and 100 EJ of residues.

When it comes to the mobile energy sector the challenge is twofold. Not only is a primary energy source needed, it is also important to find good energy carrier. The carries used today such as gasoline and diesel are mainly based on what can easily be extracted from raw oil. But ones raw oil is not used those conventional fuels are not necessarily the most efficient choice in, for example, an overall perspective of energy efficiency in production and use combined.

Research on mobile energy carries is therefore focused on both what energy source that should be used for the production and what fuel that should be produced. Here ones again the importance of the availability of the energy source is of interest which will be discussed in the next section.

1.2 Intermittent energy

Intermittent energy refers to an energy source that is not always continuously available due to some uncontrollable external factors. Most commonly it refers to energy sources effected by weather such as wind and solar power. As a rule of thumb it has been argued that as long as the usage of these sources are below 30 % the electricity demand it is not a problem (Grahn, 2012).

If however the use is to be greater than that some sort of storage of the energy is needed as a buffer for the times when the source is not available. One implementation discussed is for example to use wind power to pump water back up into the water magazines of hydro power plants.

When it comes to the mobile energy use however things look a little different. As described in the previous section mobile energy use is a lot about finding a good energy storage system. Therefore the use of intermittent sources in the production is not really a problem.

Fuels for transport produced with the help of intermittent energy sources can possibly make the production more costly but it does not have to. To get enough fuel the plants must be made large enough to produce the required amount of fuel when the energy is available. Thus leftover energy from the stationary energy system can also be used in the fuel production.

To summarize, it is challenging to find a suitable energy source, to replace fossil fuels, for electricity production as electricity can only be used instantly when it is produced and many renewable sources are intermittent. Few ways of storing energy is developed today to use with the intermittent sources and the renewable sources that are not intermittent are rather limitedly available (eg hydro power).

But for the transport sector and other mobile energy use sectors it should in principal be possible to make use of intermittent energy without this 30% limitation. This is because production of fuel when, it comes down to it, is storing of energy. The limitation then is now instead how easily the fuel production can be scaled up and down depending on the availability of the source. That question is not so easy to answer.

The use of the produced fuel however can then be set to the mean production over a year, thus evening out the uncertainty in the energy availability given that an adequate amount of fuel first is buffered.

Different alternatives are currently being discussed and tried out regarding the use of intermittent energy in fuel production. Some of the alternatives are described in more detail in the next chapter an in chapter 5.8.

1.3 Synthetic Gas Synthesization

In view of what has been mentioned above it is interesting to look closer on long term sustainable energy storage systems for the transport sectors energy carriers. For a definition of long term sustainable see (Holmberg, J. 1995).

The limited availability of oil and ethical problems of bio-energy can both be seen as symptoms from the same problem, the limited organic productivity. If wood is combusted for the sake of producing heat less than one percent of the energy from the previous incommode sunshine during the trees growth is preserved.

Compared to that a conventional solar panel can have an energy efficiency of about 10%. A solution to this fundamental problem of the plants would be to find a way to do what the plants do but more efficiently.

Replicate the photosynthesises has this far proven to be rather difficult but that is not what is needed here. Instead of producing sugar a process that can produce a flammable energy carrier is what is needed. One such is the Fischer-Tropsch process that has been used since the 50s to produce diesel in south Africa (see sasol). This process is however based on coal which makes it unsustainable.

The coal is used to produce a gas mix called synthetic-gas or syngas for short. It contains one part carbon monoxide and two parts hydrogen gas. The interesting part of the Fischer-Tropsch process is that it contains the key of how to produce diesel, gasoline, plain-fuel and more from syngas and it is already in large scale use.

What is needed is a long-term sustainable process on producing syngas. Shell recently finished a plant that uses natural gas in the Fischer-Tropsch process. This is better than coal because the carbon dioxide emissions in the production process are much smaller but it is still not long-term sustainable.

In the long run if carbon is to be a part of our transport fuel the syngas must be made from carbon dioxide from the air and water. This involves making hydrogen by splitting the bindings of water and making carbon monoxide by capturing carbon dioxide from the air and reform it to carbon monoxide.

These are both rather expensive processes that will need some development but in a first stage carbon dioxide can be captured from industrial emissions. This will be discussed more in the chapter of further work in this report.

Some studies has been conducted before regarding the possibility of turning CO_2 into CO for car fuels (Miller 2007). The process discussed here builds on these previous findings. Figure 1 shows a very schematic picture of the process. The idea is in short to reform carbon dioxide and water separately to carbon monoxide and hydrogen and then to mix them to syngas.

In figure 1 it can be seen that the process of reforming carbon dioxide to carbon monoxide requires hydrogen. Equal amounts of water is produced as a result of this when part of the oxygen leaves the carbon dioxide (Miller 2007). This can then be circled back to the hydrogen reactor.

The Hydrogen reactor produces hydrogen from water with heat in a sulphur-iodine cycle (Vitart 2006). Both reactors uses heat as their primary energy source. This is good from a thermodynamic perspective compared to more conventional hydrogen production by hydrolysis as the losses of producing electricity can be avoided.

The temperature needed for the processes are 800°C for the carbon monoxide process and 850°C for the hydrogen. Various energy sources has been discussed but the two foremost are a VHT nuclear reactor or a solar CSP power plant (Vitart 2006), (Huang 2005).

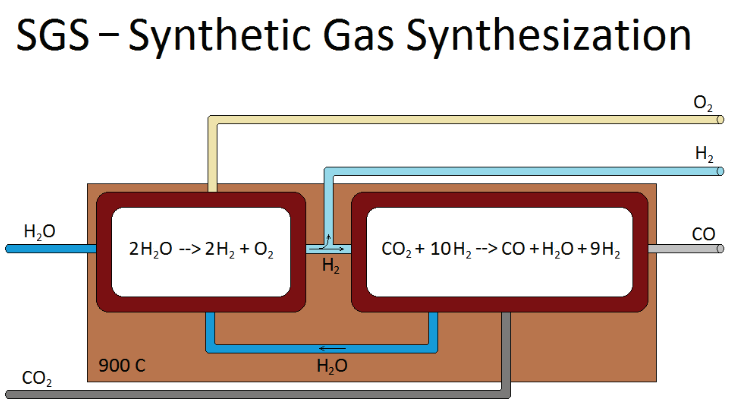

SGS – Synthetic Gas Synthesization
O_2
H_2
H_2O
$2H_2O \rightarrow 2H_2 + O_2$
H_2
$CO_2 + 10H_2 \rightarrow CO + H_2O + 9H_2$
CO
900 C
H_2O
CO_2

Figure 1: *A schematic picture of Synthetic Gas Synthesization*

A process producing methanol from industrial CO_2 is in use on Iceland (Carbon Recycling International 2012). The greatest uncertainty in the over all process discussed here is what it would cost to use CO_2 captured from the air. Most likely the production of synthetic fuels from carbon dioxide and water will be more expensive than producing fuels from fossil sources or from biomass.

Both fossil and bioenergy sources do however face long-term challenges since they both cause environmental problems and are limited sources. Limited sources will become more expensive when demand is higher than the production.

Instead of fossil fuels and biofuels it is possible to use hydrogen or electricity as energy carriers in the transportation sector. They do however also face long-term challenges since both batteries and fuel cells need improvements in e.g., capacity, life time and dependency on scarce metals. Electricity and hydrogen will also be difficult to use in the aviation and shipping sector.

One advantage for carbon-based synthetic fuels is that they can be used in current infrastructure and current vehicle fleet. It would be interesting to analyze under what circumstances synthetic fuels from carbon dioxide and water could be a cost-effective fuel choice in the transportation sector in a future carbon constrained world, see more in Section 5.8, Pre-requisites for Carbon Dioxide based Synthetic Fuels.

1.4 The GET-model

To get a good overview of the energy system, and an idea of how the fossil fuels can be replaced at lowest cost, the GET-model was written. GET stands for Global Energy Transition and is a linear global energy optimisation model developed at the department of Energy and Environment, Physical Resource Theory (PRT), at Chalmers see e.g., Azar et al (2003), Grahn et al, (2009) and Hedenus et al (2010).

It has been developed over the years and therefore exists in many different versions. The version used in this study is GET-RC 6.1. This version is regionalized and has a refined transportation sector, focussing on fuels and vehicle technologies for cars (Grahn et al, 2009).

The model is used to generate global long-term scenarios of the energy system under different conditions. It is also a tool for general analysis of the energy system. It's greatest strength is that the resources of different types can float between the different energy end-use sectors (in this study called services) of the energy system. Providing electricity is one example of a service and Heat, for first and foremost the process industry, is another.

The model then does not just say what resource that is most cost-effective for electricity and what's for heat. It takes all services into account at the same time. This means that the model also calculates for what service a specific resource should be used in each sector to reduce the CO_2 emissions as inexpensive as possible.

2 Purpose

This thesis describes some further development of the GET-model. The degrees of freedom needed by the model has grown very quickly with increased complexity partly due to the optimization environment used called GAMS but also because the model was constructed over a long time by many different people. Most quantities in the model are stored in atleast two variables in the GAMS implementation. One variable for each quantity would have been enough but some quantities are declared even in more than five places. This also called for many extra constraints.

This is no problem in it self as our computers of today are rather fast and the model is still relatively small. GAMS and the optimizer used called CPLEX however also have rather costly licenses. As the complexity of the model would increase if new fuel alternatives were included it was decided to first investigate whether rewriting the model could make it more efficient.

In the reimplementation of the model a different programming language was used. The alternative language implemented in this project is the functional programming language Scheme and the GAMBIT compiler. The optimizer used is GUROBI.

This report describes both the simplification of the GET models algorithms and their reimplementation in the new language. Pre-requisites for introducing carbon dioxide based synthetic fuels was identified. Finally at the end of the report some further work is proposed.

Short Summary of Purpose Here the purpose of the thesis is summarized in three points:

- Find the smallest linear combination (a mathematical interpretation) that captures the GET-RC 6.1 model by simplifying the GET-models algorithms.
- Try out this simplified GET-model in a functional programming language.
- Identify pre-requisites needed to introduce synthetic fuels made from carbon dioxide and water in the model.

3 Overview – Differences in programming languages

All versions of the GET-model developed by Physical Resource Theory (PRT) are written in an optimisation environment called GAMS. GAMS in short makes the computers more accessible for users by providing functionalities that can come in handy when you write optimization models.

A problem with such environments or programming languages however is that the capabilities of the computer are limited to what is provided within the environment. Sometimes it might be easier to solve a problem with other methods than the ones included in the environment. Performance and clarity are also easily lost on the way.

In GAMS a slight loss of clarity can be explained in that the user never really need to make a mathematical interpretation of the model. The tools GAMS provide makes it easier to explain the problem for the computer and then it is more up to the machine to interpret the model. The user can therefore not always see what the computer do as it tries to solve the problem in the background.

This of course made my rewriting of the model into a new language a little more complicated then was thought at first. In a way I did not only need to re-write the model it self but also re-write parts of the optimization environment. This is described in more detail later.

When rewriting the GET-model there are many different programming languages that can be used. To understand what the difference is between GAMS and the Scheme a little background in some central elements of programming is needed. Below therefore follows a brief introduction.

3.1 The Compiler, Assembler

In the early days of computing, programming consisted of flipping switches to load a program into a Computers memory. These sequences of ones and zeros were called machine code and told the computer what to do. Soon however it become apparent that this way of instructing the computer was rather inefficient.

Today however our methods for producing this machine code is a lot more efficient. A programming language can be made with which instructions for a computer can be written in a more efficient and concentrated form. The problem however is that the computer can not by it self understand and follow these new instructions.

The instructions needs to be translated to machine code for the computer to be able to read and follow them. The translation is done by a program or set of programs called a Compiler. This means that the final program will be just as complicated as if it was written in machine code from the beginning.

The difference is that the computer does not do as many mistakes as a human do when printing a long row of ones and zeros. Many things have to be repeated many times over so the difficult thing is not to figure out what to write but to get it right over and over again. This is a task very well suited for a computer. In this way the computer to some degree programs it self.

One early programming languages that is compiled to machine code is Assembler. It is possible to write assembler code by hand but it is not so common today. This is because it is still a very primitive language but it is better than writing the machine code by hand. For more about Assembler see (Salomon, D. 1993).

3.2 The C Programming language

As Assembler code were still a rather simple programming language, new languages particularly for making mathematical calculations and subroutines were made. These languages are often refereed to as low-level languages but the definitions here vary. What is to be called a low-level language or not depends on the perspective of how near machine code you must be to be on a low-level.

The meaning of what a compiler is changes a little with the introduction of these languages. The new Compilers for these low level languages often do not make machine code but produce Assembler code instead. A compiler can now therefore be a program that translates between two programming languages and not only to machine code.

Among these low level languages the C programming language is the one that is generally considered to produce the fastest programs. C has other advantages and some disadvantages such as some difficulties when allocating memory. It is however a very useful language for writing simple background programs that needs to be fast or programs that needs to be called on repeatedly. C is there fore often used in programming operating systems. For more about the C programming language see (Kernighan 1988).

3.3 Object-oriented Languages

As a final step to get where the programming languages are today the Compiler process needs to be repeated one last time. Compilers today mostly writes ether C code or assembler code. But the new languages are rather different from the ones before. Earlier the new languages were invented to increase the effectiveness of writing the old languages and to effectively map only what a computer can do was the priority.

The new so called object-oriented languages instead provide certain objects that programmers can find useful. These objects has made programming a lot easier than before as many things can be reused. But these objects are more made for what a programmer need than what a computer can do. It is therefore not always so easy to successfully implement these objects in the underlying C or assembler code. The objects are made to fit the programmer, more than the computer.

There are many object orientated languages. A few examples here are C+, C++ and C#. These three languages are all related to C but are larger and have many new functions that makes the programming more efficient.

Even though the objects make the programming more efficient they also introduce a loss in transparency and clarity. The implementation of certain functions or objects was something the programmer had to figure out before but it is now built into the language it self. The objects implementation can not be changed or even seen by any individual programmer unless the objects source code is available. This can introduce losses in efficiency of different magnitude depending on how well a object suits a certain use. This can be hard to judge considering the limited transparency. For more about Object-orientated languages see: (Nino 2008).

3.4 Functional programming Languages

Due to the limited transparency of the object based languages it is difficult to invent new languages and write new compilers to make the computer once again program it self on top of a object based language. This next step in the development of the programming languages therefore once again seeks to find more efficient ways of writing C or assembler code.

The difference here compared to the object based languages is that the focus now once again is back on the computer and what it is capable of. The new language like C to assembler seeks to find a more efficient way to instruct computer.

A functional programming language can be seen as a language to make the computer program it self. Instead of just repeating the compiler stage once more the functional languages are built to include this in the language it self. The programmer invents his or hers own language as needs arise and the components invented can be reused. To make sure that the programs are using the computers resources efficiently the languages have a mathematical nature. The programs are functions and these functions can efficiently be used by other functions.

This stacking of functions applied to other functions makes the code more manageable and transparent. The programs become shorter because many things does not need to be repeated. If the code becomes inefficient the language can be redefined so the computer can do more by it self. For more about the Functional Programming Language Scheme see: (More 2008).

3.5 GAMS

The GAMS optimisation environment can be seen as a object-oriented programming language. GAMS is then the compiler that provides the objects. Like many other compilers GAMS is written in the C language. GAMS also provide many useful objects. One such object is called a variable.

This variable is not an ordinary mathematical variable representing one numerical value or a dimension in space. It is in fact a table or system of many mathematical variables. The size or number of dimensions of this variable object is defined by another object called a set. A set can be for example the regions included in the model or the primary energy sources that are allowed.

These objects can then be combined to form the variable of all coal used in all regions for example. The variables in the GET model are sometimes depending on five different sets which makes the variable five-dimensional. By using these and other GAMS objects and input data the GET-models dynamics is defined for the computer.

GAMS then in the background creates a mathematical interpretation of the model by reading the objects. This mathematical implementation which basically is a matrix is then sent to a so called optimizer. The optimizer is then the program that does the actual optimization.

The optimizer is a separate program outside of GAMS and the looks of the matrix sent to it is defined from what the optimizer is expecting. So GAMS also translates the GET-model matrix to the right form before sending it to the optimizer. GAMS can handle many different optimizers but the one often used at Physical Resource Theory is called CPLEX. CPLEX then returns the solution to GAMS which interprets it and present the solution to the user. For more information on the GAMS optimisation environment, see e.g., Rosenthal (2012).

3.6 GAMBIT and a comparison with GAMS

GAMBIT is a compiler for the functional programming language Scheme. Just as GAMS GAMBIT is written in C but it is not an object-orientated optimization environment. GAMBIT only provides the basic functions of a computer but with many mathematical functions available. For example GAMBIT can from the start handle complex numbers that can be as large as you like as long as they can fit into the computers ram memory. With basic functions of a computer I refer to things such as allocating memory and to perform five basic binary Operations.

Apart from this GAMBIT also provide different kinds of lists and loops. In GAMS these elementary components are already combined into variables, sets and other things useful for optimization. In GAMBIT these things must be constructed manually.

On the up side though the language is not restricted to the objects available in GAMS but many more objects can be created. Objects can be created in other ways than in GAMS to suit the GET-models needs. The connection to the underlying machine is not broken but open and the functions created will be more transparent, not working in the background as before. The construction of this new optimization environment is explained in the next section. For more about the syntax of GAMBIT and Scheme see (Kelsey 1998), (More 2008).

4 Method

4.1 The New GAMBIT-GUROBI Environment

As explained earlier the first step in rewriting the GET-model for the GAMBIT compiler was to reconstruct the optimization environment. It however proved easier to introduce new functions as they were needed instead of reimplement the whole GAMS environment from the start. When an object was needed all specifications was also known.

The first thing to be reimplemented was the translation method of GAMS which translates the model to the interface of the optimizer. A so called Foreign Function Interface (FFI) was written for this purpose.

Instead of CPLEX a new optimizer called GUROBI was used. It has the same basic functions as CPLEX but with the difference that it is free of charge for academic use. GUROBI has many interfaces for gathering instructions but in this environment the C interface was used as it is the fastest and because it is very easy to produce C code in scheme. The FFI simply produce C coded instructions of the model in the way GUROBI is expecting. The FFI is published under a MIT open source license and can be found at (Andersson 2012).

The GET-model it self was implemented in smaller sections, one at a time. These sections are called simple model 1 through 6 and the sectioning was made by Maria Grahn in a previous project(Grahn et al, 2013). This method section is therefore separated into one section for each simple model that was implemented.

In between each new simple model sections are one so called functionalisation section where it is explained what new objects or rather functions that was introduced, as scheme is not object-oriented but functional. Due to the limited time frame of the project three out of these six simple models was reimplemented into the new optimization environment. This was judged enough to see if the model got any faster.

Below follows a table providing an overview of the differences between GAMS and the new GAMBIT environment.

Table 1: *An overview of the different components of the old and new optimization environment.*

Description	GAMS Version	GAMBIT Version
Programming Language:	.gms-code	Scheme
Compiler Name:	GAMS	GAMBIT
Optimizer Translation:	GAMS (Black box)	The GAMBIT-GUROBI FFI
Optimizer used here:	CPLEX	GUROBI

The simplification of the GET-model was made during the reimplementation and is mainly based on unit analysis. The GET-model is built up around starting values, conversion coefficients and expected demands. These values together form the static part of the model.

Between these values are the variables which together with the constraints defines the dynamics of the model. The model then measures the total cost of the specific paths taken through the dynamics of the constraints and variables.

That total cost is what is minimized.

What is meant by simplifying with unit analysis here can be a little unclear and it is hard to explain. The GAMS implementation has many so called green variables. These green variables are of great use when making a presentation of the model's results, and comparing them with real world data, but of no use for the computer.

In the underlying calculations the green variables mainly cause confusion as GAMS needs to sort out what is relevant and not. That kind of philosophical judgement is something the programmers do much better than computers. Therefore when simplifying the GET-model as many as possible of these philosophical green variables has been removed out of the optimization. Instead they can be calculated afterwords when asked for. An overview of the GET-model as it is implemented in GAMS is shown in Figure 2.

This report does not seek to give a full picture of the GET-model in GAMS but figure 2 is meant to at least give a feeling for the original implementation. The figure can however be compared to figure 13 in the results section.

When the simplification of the GET-model was made other names were used for the variables and the data was sorted in a different way. In this report however these other names and frameworks has been translated as good as possible to the names of the original implementation of the GET-model to make it easier to compare the implementations.

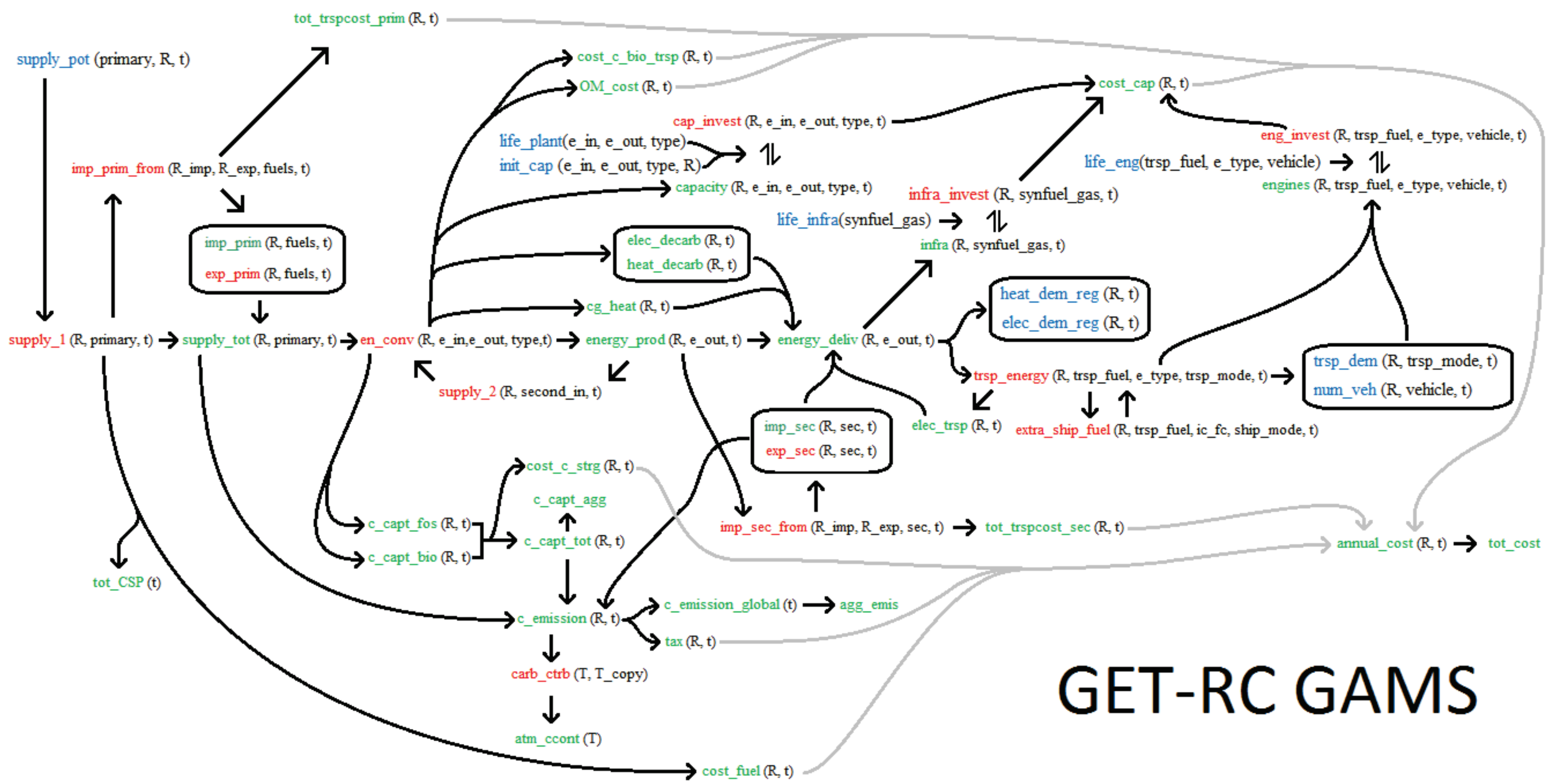

tot_trspcost_prim (R, t)
supply_pot (primary, R, t)
cost_c_bio_trsp (R, t)
OM_cost (R, t)
cost_cap (R, t)
cap_invest (R, e_in, e_out, type, t)
eng_invest (R, trsp_fuel, e_type, vehicle, t)
life_plant(e_in, e_out, type)
init_cap (e_in, e_out, type, R)
life_eng(trsp_fuel, e_type, vehicle)
imp_prim_from (R_imp, R_exp, fuels, t)
capacity (R, e_in, e_out, type, t)
infra_invest (R, synfuel_gas, t)
engines (R, trsp_fuel, e_type, vehicle, t)
life_infra(synfuel_gas)
imp_prim (R, fuels, t)
exp_prim (R, fuels, t)
elec_decarb (R, t)
heat_decarb (R, t)
infra (R, synfuel_gas, t)
heat_dem_reg (R, t)
elec_dem_reg (R, t)
cg_heat (R, t)
supply_1 (R, primary, t)
supply_tot (R, primary, t)
en_conv (R, e_in,e_out, type,t)
energy_prod (R, e_out, t)
energy_deliv (R, e_out, t)
trsp_energy (R, trsp_fuel, e_type, trsp_mode, t)
trsp_dem (R, trsp_mode, t)
num_veh (R, vehicle, t)
supply_2 (R, second_in, t)
imp_sec (R, sec, t)
exp_sec (R, sec, t)
elec_trsp (R, t)
extra_ship_fuel (R, trsp_fuel, ic_fc, ship_mode, t)
cost_c_strg (R, t)
c_capt_agg
c_capt_fos (R, t)
c_capt_tot (R, t)
c_capt_bio (R, t)
imp_sec_from (R_imp, R_exp, sec, t)
tot_trspcost_sec (R, t)
annual_cost (R, t)
tot_cost
tot_CSP (t)
c_emission (R, t)
c_emission_global (t)
agg_emis
tax (R, t)
carb_ctrb (T, T_copy)
atm_ccont (T)
cost_fuel (R, t)
GET-RC GAMS

Figure 2: *The GET-model with all its variables as it is implemented in the GAMS environment. The arrows in the figure represent the constraints and the red and green texts are the variables. Some more important static components are also showed in blue in the figure. The letters added in parentheses and separated by comma represents sets. The grey arrows are the final terms of the objective function. For description of sets variables and more see Appendix C.*

5 Results

To make it easier to see the results each new Simple-Model section below begins with a flowchart providing an overview of that specific simple model implemented in GAMS. Later in the same section follows another flowchart of the same simple model but now implemented in GAMBIT.

All new implementation are mathematically equivalent to the original models, that is all information that was provided by the original model can be composed from the output of the new calculations. The constraints are also equivalent but expressed slightly different.

As explained earlier this report does not seek to give a complete description of the GAMS implementation of the GET-model but explains only what is needed to understand the new implementation which is described in full.

5.1 Simple-model 1

The main purpose of simple model 1 is to find the optimal use of the primary energy sources. An overview of how this is implemented in GAMS is showed in figure 3 below.

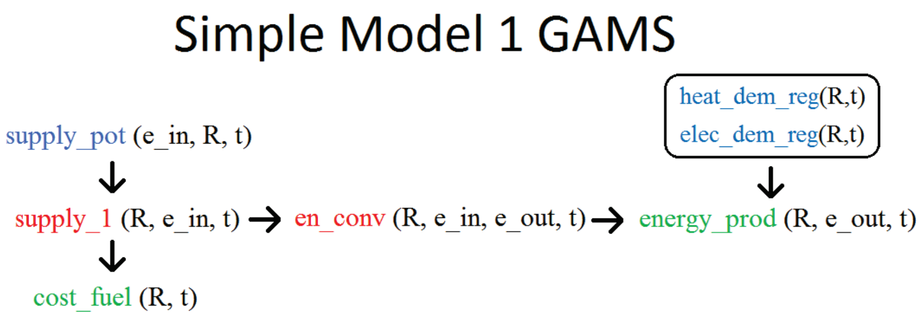


Figure 3: *The simple-model 1 implementation in GAMS. See List of Acronyms for explanation to the acronyms.*

supply_pot is the available primary energy sources given as a constraint to the model, supply_1 is the amount of primary energy resources that the model chooses to use, cost_fuel is the cost of the primary energy resources chosen in supply_1, en_conv keeps track on what the resources in supply_1 are used for and energy_prod is the amount of heat and electricity that is produced. heat_dem_reg and elec_dem_reg are constraints on the model making sure the energy_prod is big enough.

The set of energy source alternatives is called e_in, e_out is the set of secondary energies demanded, R is the region set and t is the set of time steps in the model. Table 2 describes the contents of these four sets in simple model 1. The sets are the same for the GAMS and GAMBIT implementation but the sorting within the sets are sometimes different, when so the GAMBIT sorting is what's shown.

Table 2: *The contents of the sets in simple model 1. See List of Acronyms for explanation to the acronyms.*

e_in	e_out	R	t
Coal	Heat	NAM	1990
Oil	Elec	EUR	2000
NG	-	-	2010
Bio	-	-	-

By applying the previous described unit analysis (see Method) to the model above it is found that the variables supply_1, cost_fuel and energy_prod are all subsets of en_conv. This is clear because all these variables contain information about the primary energy source resources and their use but en_conv has the greatest detail. In the sought smallest linear combination of the GET-model these variables are therefore removed. An overview of the new GAMBIT implementation of simple model 1 is shown in figure 4.

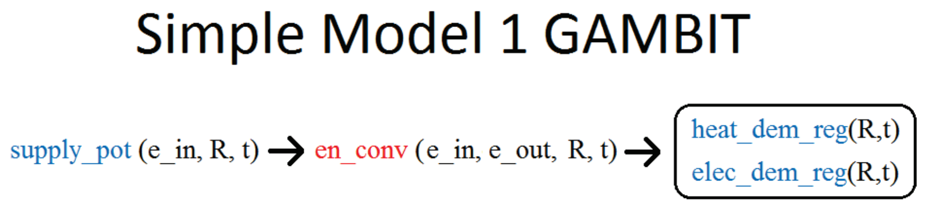


Figure 4: *The simple-model 1 implementation in GAMBIT. See List of Acronyms for explanation to the acronyms.*

In this new implementation all constraints are applied to en_conv directly. This grately reduces the complexity of the model and the number of constraints too as no borders needs to be set for the three variables that was removed. In the new model the first set declared in a variable defines the content of it. en_conv thus contains e_in material which is primary energy source resources.

The model now have two constraints, one that make sure en_conv uses less resources than what's available (Formula 1) and one that makes sure en_conv satisfy the demand on electricity and heat (Formula 2). effic is a table of constants representing each process' conversion efficiency, i.e. the share of the energy content in the primary energy source that remain in the energy carrier after energy losses in the conversion. These constants convert the e_in content of en_conv to the corresponding amount of e_out for the comparison.

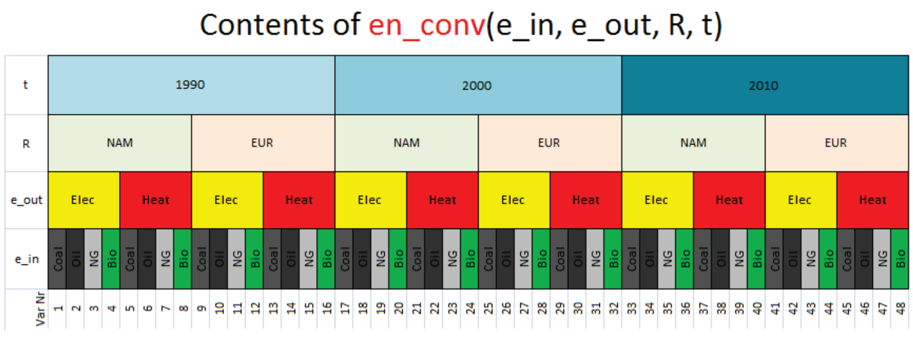
$$\sum_{e\_out} en\_conv(e\_in, e\_out, R, t) \leq supply\_pot(e\_in, R, t) \tag{1}$$

$$\sum_{e\_in} en\_conv(e\_in, e\_out, R, t) * effic(e\_in, e\_out, t) \geq dem\_reg(e\_out, R, t) \tag{2}$$

The cost to be minimized in the model is simply the cost of the primary energy resources allocated in en_conv (See Formula 3). price is a table of constants converting the e_in content of en_conv to the price of extracting those resources. t_step is the length of the time steps in the model that is 10 years.

$$min(t\_step * \sum_{\substack{e\_in,\\ e\_out,\\ R,t}} (en\_conv(e\_in, e\_out, R, t) * price(e\_in, R))) \tag{3}$$

To get an overview of en_conv when the GAMBIT implementation was made an excel table of en_conv was made (see figure 5). This table describes what the variable is suppose to contain if all constraints are set right. This was then verified towards the original model in GAMS. First when both the final cost and the content of en_conv matched between the two programming languages, the work moved on to the next simple model.



Contents of en_conv(e_in, e_out, R, t)

t	1990																2000																2010															
R	NAM								EUR								NAM								EUR								NAM								EUR							
e_out	Elec				Heat				Elec				Heat				Elec				Heat				Elec				Heat				Elec				Heat				Elec				Heat			
e_in	Coal	Oil	NG	Bio	Coal	Oil	NG	Bio	Coal	Oil	NG	Bio	Coal	Oil	NG	Bio	Coal	Oil	NG	Bio	Coal	Oil	NG	Bio	Coal	Oil	NG	Bio	Coal	Oil	NG	Bio	Coal	Oil	NG	Bio	Coal	Oil	NG	Bio	Coal	Oil	NG	Bio	Coal	Oil	NG	Bio
Var Nr	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48

Figure 5: *What the variable en_conv is suppose to contain if all constraints are correctly defined. See List of Acronyms for explanation to the acronyms.*

The content of the tables of constants and the reference values of the constraints can be found in Appendix B.

5.2 Simple-model 1 Functionalisation

As mentioned earlier these functionalisation sections mainly describes the work of building the new optimization environment. It become apparent what was missing when the environment was used and so the environment was developed during the implementation. After the introduction of simple model 1 the need of a better loop function was apparent.

The environment and the model needed structure for better oversight. Still this structure also needed to be built in such a way that it did not cause problems for the computer when calculating. What was introduced now in the GAMBIT environment to get this structure has no good comparison in GAMS. When GAMS however addresses the problem sets and variables are what is used so these names will be used here to explain the new structuring of the model.

In the current state of the environment a lot of code needed to be repeated many times over to get through all values of en_conv. A new loop and a so called list builder was therefore implemented into the GAMBIT environment. The new loop required only a start and ending value to run. It also provided its current value as a separate reference value for easy implementation in the construction of the model. It is these loops that forms the sets and with them the model could loop over the sets of a variable in what ever order preferred without missing any entries.

The new list builder could access the reference values provided by the loop and thereby know where in the model it was currently working. In this way this so called list builder could pick the right constants out of the constant tables (price and effic) and find the right reference values in dem_reg to build the constraints.

Afterwords when I compared the new environment with GAMS I found that these functions discussed here were very similar to the functions of sets in GAMS. The new loops are the sets and what they contain and the list-builder is similar to the background process of GAMS that reads the sets and build the model around them. Both these new functions were used repeatedly when making the next model simple-model 2.

During the implementation these new sets of GAMBIT were called fundamental variables because in some ways they are different from the sets of GAMS. The differences however between the sets and later also variables in the new environment and in GAMS will be described further in the simple-model 2 functionalisation section as it is not until then they are fully implemented.

5.3 Simple-model 2

The new thing introduced in simple-model 2 is transfer of primary energy resources between the regions. In simple one en_conv could be seen as both what was used and how much resources that was extracted domestically. This is not true for simple-model 2 as en_conv primarily describes what is used for domestic production and resources now can be extracted in one region and then transferred to another. Below follows the GAMS implementation of this in figure 6.

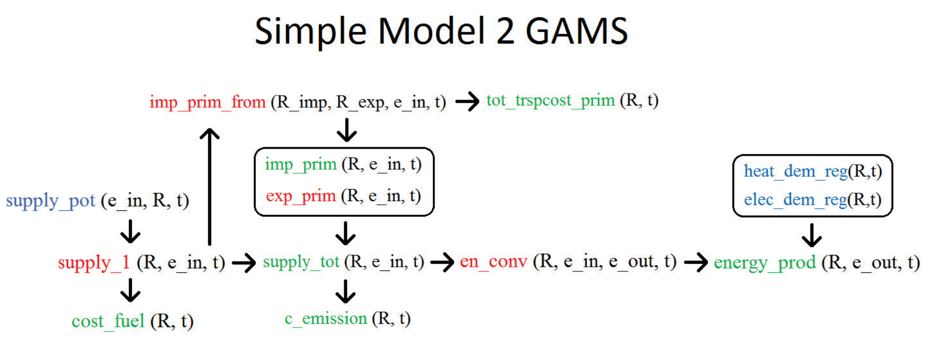


Figure 6: *The simple model 2 implementation in GAMS. See List of Acronyms for explanation to the acronyms.*

One new variables in this model is import_prim_from that contains all data of the resource transfer with where the goods come from, where it is going, what the goods are and when it was transferred. If the unit of import_prim_from is changed to GUSD and the variable is summarized to only R and t it becomes tot_trspcost_prim which therefore is a subset variable.

imp_prim and exp_prim are also subset variables to import_prim_from. imp_prim summarize the amount of goods that is imported to the regions and exp_prim how much that is exported from each region. These two variables are not enough to calculate tot_trspcost_prim as they neither contain where exported goods are going or where imported goods are coming from but together with supply_1 they form the new variable for the amount of available resources in a region called supply_tot which thereby also is a subset variable.

en_conv is then built upon supply_tot or rather, as no resources are extracted anywhere unless they are used, sypply_tot are based on en_conv and constrained by supply_1, imp_prim and exp_prim. Finally the variable c_emission is introduced but it is not part of any constraint on the model and therefore unnecessary for the optimization at this time. Also c_emission is a subset variable of supply_tot as all fuel there are to be combusted. Finally the sets have not changed but are identical to the sets of simple-model 1.

Just as before all subset variables are removed in the GAMBIT model. The only new variable that is not a subset variable is import_prim_from. This is therefore the only new variable that needs to be introduced. Figure 7 below summarizes simple-model 2 in GAMBIT.

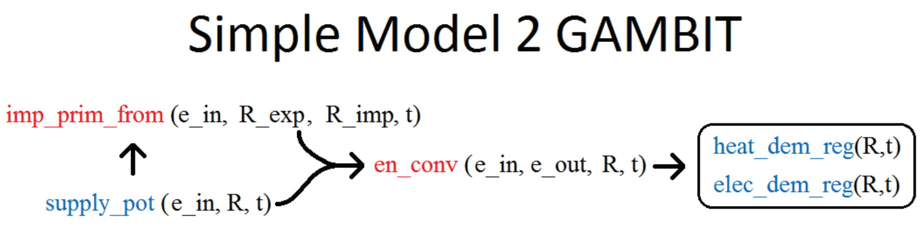


Figure 7: *The simple model 1 implementation in GAMS. See List of Acronyms for explanation to the acronyms.*

No new constraints needed to be added to this model but the first constraint was slightly rewritten. supply_pot is now not just a limit for the domestic use but for the domestic use and export. Further more as primary energy sources can be imported this also needs to be taken into account as the domestic supply_pot not always carry the whole burden. The new first constraint is shown in formula 4. The constraint in formula 5 is the same as in formula 2.

$$\sum_{e\_out} en\_conv(e\_in, e\_out, R, t) - \sum_{R\_exp} imp\_prim\_from(e\_in, R\_exp, R\_imp, t) + \sum_{R\_imp} imp\_prim\_from(e\_in, R\_imp, R\_exp, t) \leq supply\_pot(e\_in, R, t) \quad (4)$$

$$\sum_{e\_in} en\_conv(e\_in, e\_out, R, t) * effic(e\_in, e\_out, t) \geq dem\_reg(e\_out, R, t) \quad (5)$$

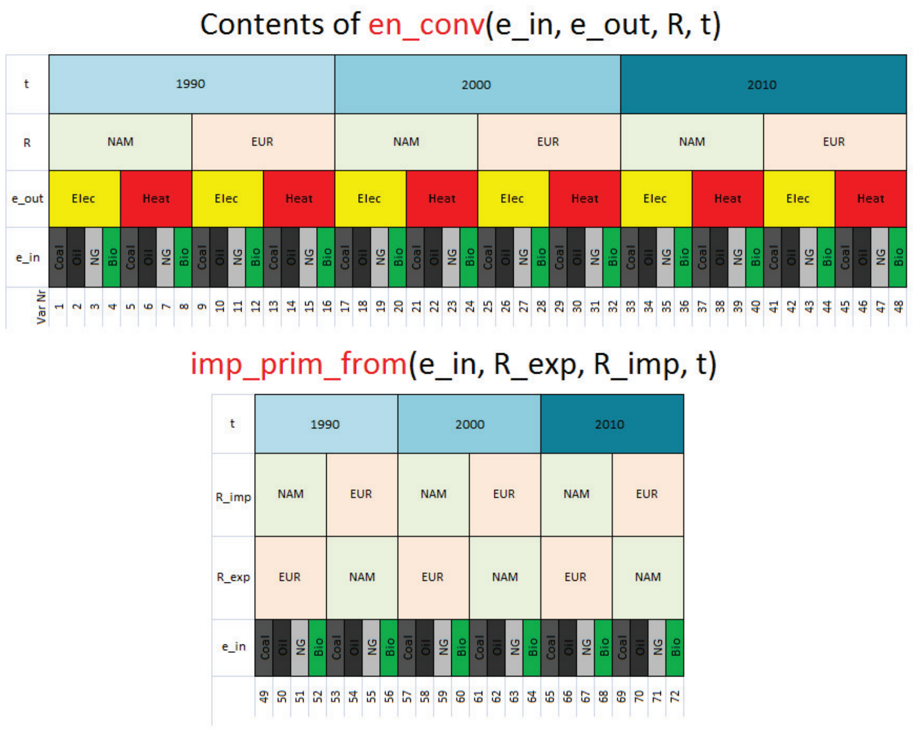
It should be pointed out that the different order of the sets of imp_prim_from in the two different sums of formula 4 is not accidental but deliberate and the reasons for this are described in more detail in the next chapter.

The objective function has changed in that the cost of transferring goods now needs to be taken into account. The price of extracting can also different in different regions. All this is handled in a new term in the objective function based on the new variable imp_prim_from. Formula 6 shows the new objective function in its entirety. The terms A, B and C just used to make the objective function clearer in this paper.

$$\begin{aligned} min(t\_step * \sum_{\substack{e\_in,\\ e\_out,\\ R,t}} (en\_conv(e\_in, e\_out, R, t) * price(e\_in, e\_out, t) \\ +(impC + impCL + priceDiff) * imp\_prim\_from(e\_in, R\_exp, R\_imp, t))) \\ where: \\ impC = imp\_cost(R\_exp, R\_imp) \\ impCL = imp\_cost\_lin(e\_in)) * distance(R\_exp, R\_imp) \\ priceDiff = price(e\_in, R\_exp) - price(e\_in, R\_imp) \end{aligned} \quad (6)$$

imp_cost, imp_cost_lin and distance are constant tables. imp_cost contains a fix cost and imp_cost_lin contains a linear cost dependent on the distance between the regions. distance contains a table with very approximative distances between the regions.

Finally for the simple-model 2 the Excel table used during the implementation was updated to cover the new variable imp_prim_from (see figure 8). This table as described before shows the supposed content of the variables in the model and acts as a reference for whether the constants were set right. The table of en_conv is unchanged.



Contents of en_conv(e_in, e_out, R, t)

t	1990															
R	NAM								EUR							
e_out	Elec				Heat				Elec				Heat			
e_in	Coal	Oil	NG	Bio	Coal	Oil	NG	Bio	Coal	Oil	NG	Bio	Coal	Oil	NG	Bio
Var Nr	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

t	2000															
R	NAM								EUR							
e_out	Elec				Heat				Elec				Heat			
e_in	Coal	Oil	NG	Bio	Coal	Oil	NG	Bio	Coal	Oil	NG	Bio	Coal	Oil	NG	Bio
Var Nr	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32

t	2010															
R	NAM								EUR							
e_out	Elec				Heat				Elec				Heat			
e_in	Coal	Oil	NG	Bio	Coal	Oil	NG	Bio	Coal	Oil	NG	Bio	Coal	Oil	NG	Bio
Var Nr	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48

imp_prim_from(e_in, R_exp, R_imp, t)

t	1990								2000								2010							
R_imp	NAM				EUR				NAM				EUR				NAM				EUR			
R_exp	EUR				NAM				EUR				NAM				EUR				NAM			
e_in	Coal	Oil	NG	Bio	Coal	Oil	NG	Bio	Coal	Oil	NG	Bio	Coal	Oil	NG	Bio	Coal	Oil	NG	Bio	Coal	Oil	NG	Bio
	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63	64	65	66	67	68	69	70	71	72

Figure 8: *What the variables en_conv and imp_prim_from are suppose to contain if all constraints are correctly defined. See List of Acronyms for explanation to the acronyms.*

5.4 Simple Model 2 Functionalisation

After the implementation of simple-model 2 there were mainly three things that needed to be changed in the optimization environment. The smallest of these was that the structuring system previously described as sets was expanded to include the input data.

The list builder, see simple-model 1 functionalisation, could now browse the input data tables and request data with the use of sets. The data tables got individual search engines which made it easier to find errors in the search engines as that could be tested individually. It also reduced the amount of code as the search engines only needed to be written once.

5.4.1 Variables

A new function called EPF-set! was introduced. This is where variables are introduced in the GAMBIT environment. In figure 8 above it can be seen that en_conv has 48 degrees of freedom or dimensions. The EPF of EPF-set! stands for Element Position Function. EPF-set! finds the index of a certain dimension of a variable provided the set configuration of this particular dimension. This information is then used to set a constraint.

As there are now more than one variable EPF-set! also require the index of the variable that the constraint is to be applied to. en_conv has index 1 as it was introduced first and imp_prim_from has index 2. It is this variable index that becomes the variables in the new environment.

The variable indexes are needed because all sets were already looped throw once in en_conv. As the sets or at least most of the sets now need to be looped through twice to capture the model an index is needed to separate time step 1990 of en_conv from time step 1990 of imp_prim_from.

It was mentioned earlier that the use of both imp_prim_from(e_in, R_exp, R_imp, t) and imp_prim_from(e_in, R_imp, R_exp, t) in the constraints was not accidental. As variables are now part of the environment a distinction could be made between two different types of variables. In the GAMS version of GET-RC they are referred to as red and green variables.

When these different variables are declared in GAMS no difference are made between them, both are declared in the computers memory and sent onwards to the optimizer. In the GAMBIT environment green variables are treated separately. imp_prim_from(e_in, R_imp, R_exp, t) is actually a third variable in simple-model 2 but it does not allocate any new memory, nor is it needed in the calculations.

The variables in the GAMBIT environment are a little different from the variables of GAMS. en_conv need 48 degrees of freedom or dimensions but these are not stored in the variable itself. The variable only contains directions as to where the dimensions can be found. Many variables can therefore be declared on the same dimensions. The third variable of simple-model 2 has another sorting of the contents of the second variable imp_prim_from. This new sorting is better suited for calculating on exports instead of imports as before.

In simple-model 2 this resorting of imp_prim_from does not make much of a difference as there are only two regions but simple-model 3, which has ten regions, the difference is significant. Furthermore if the third variable had been declared with its own memory allocated simple-model 3 would have had about twice as many dimensions to account for.

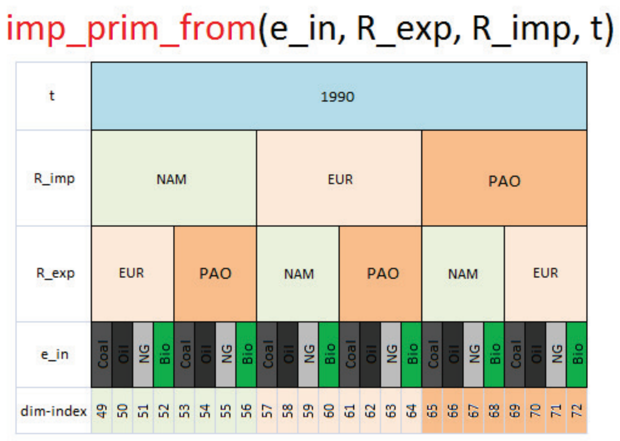
5.4.2 Sets

Something more should be said about the new sets as well. The biggest difference worth to be noticed is that the new sets can be subjected to some limitations and constraints. In GAMS for example the sets R_imp and R_exp are used to make sure a region does not import from it self. But in the GAMBIT environment there is only one set called R which is then subjected to the necessary constraints to fit the variable.

What this brings along is that the size of the sets can be changed easily as the new model can adapt to this automatically through the constraints on the sets. This makes it easier to search for errors as the model can be scaled down to only a few regions and time steps when needed. To do this in GAMS all sets related to each other needed to be modified and all data tables must be edited to fit the smaller sets. In the new environment only one index needs to be changed, that is the index that defines the size of a set.

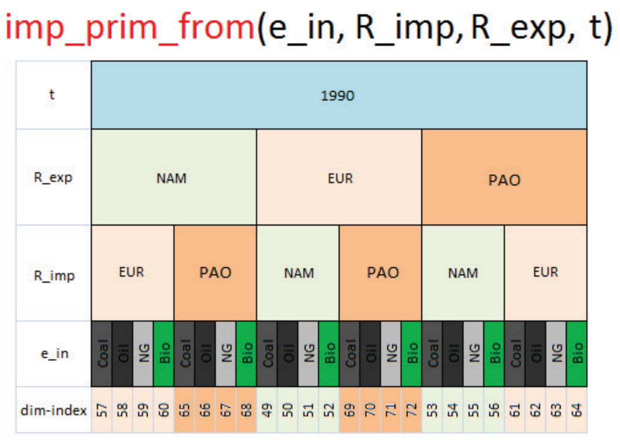
5.4.3 Example

Below in figure 9 is a Excel table of the first time-step of the second and third variable of simple-model 2. For clarity of the difference a third region has been added called PAO that is not present in simple-model 2. Look especially on how the variable indexes in the bottom row of the two tables are different.



imp_prim_from(e_in, R_exp, R_imp, t)

t	1990																							
R_imp	NAM								EUR								PAO							
R_exp	EUR				PAO				NAM				PAO				NAM				EUR			
e_in	Coal	Oil	NG	Bio	Coal	Oil	NG	Bio	Coal	Oil	NG	Bio	Coal	Oil	NG	Bio	Coal	Oil	NG	Bio	Coal	Oil	NG	Bio
dim-index	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63	64	65	66	67	68	69	70	71	72



imp_prim_from(e_in, R_imp, R_exp, t)

t	1990																							
R_exp	NAM								EUR								PAO							
R_imp	EUR				PAO				NAM				PAO				NAM				EUR			
e_in	Coal	Oil	NG	Bio	Coal	Oil	NG	Bio	Coal	Oil	NG	Bio	Coal	Oil	NG	Bio	Coal	Oil	NG	Bio	Coal	Oil	NG	Bio
dim-index	57	58	59	60	65	66	67	68	49	50	51	52	69	70	71	72	53	54	55	56	61	62	63	64

Figure 9: *The difference between the variables imp_prim_from(e_in, R_exp, R_imp, t) and imp_prim_from(e_in, R_imp, R_exp, t). See List of Acronyms for explanation to the acronyms.*

5.5 Simple Model 3

Simple-model 3 was the last model to be implemented in this study. However an overview of the final version of GET-RC with all variables and sets needed is provided in section 5.7.

The introduction of simple-model 3 lead to three major changes. Firstly plants was introduced which put a price on the process needed to transform primary energy sources to secondary for example coal power plants. The plants however still only produce heat or electricity. Figure 10 shows simple-model 3 as it is implemented in GAMS.

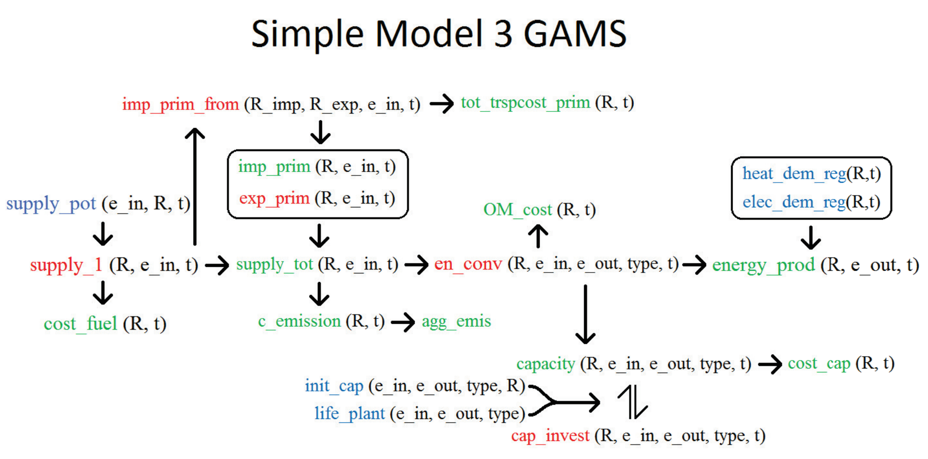


Figure 10: *The simple-model 3 implementation in GAMS. See List of Acronyms for explanation to the acronyms.*

A couple of new variables are introduced in the GAMS implementation. agg_emis is a summation of c_emission and neither are part of the cost minimization, since emission constraints are not yet included in simple-model 3. cap_invest contains the capacity (TW) of plants that is to be built. Capacity (TW) is a subset variable built from cap_invest, the constant tables init_cap and life_plant and is constrained by en_conv to contain a capacity of adequate size. cost_cap is a subset summation of capacity of the unit GUSD.

The double arrow in figure 10 refers to a process that builds the capacity and cap_invest variables one time-step at a time. These two variables are interconnected because every new investment is effected by the current capital and demand. Likewise the current capital is dependent on the investment in the last time step.

Therefore it is hard to say which variable here that is a foundation for the other as there is a time-step dependency involved and thus the arrow is a double arrow is used to describe this interaction.

Secondly the energy conversion plants could be of different types introducing a new set called type. This set monitors whether the power plant has carbon dioxide capture, co-generation of heat or both carbon capture and co generation. The plants can also be traditional power plant with neither of the two. Simple-model 3 include the traditional power plants only but the set type was judged easiest to introduce anyway to find the right values in the input data and more.

Thirdly the old sets e_in, R and t were expanded. A table of the five sets and what they contain is shown in table 3 below.

Table 3: *The contents of the sets in simple-model 3. See List of Acronyms for explanation to the acronyms.*

e_in	e_out	type	R	t
Coal	Heat	0	NAM	1990
Oil	Elec	-	EUR	2000
NG	-	-	PAO	2010
Bio	-	-	FSU	2020
Nuclear	-	-	AFR	2030
Wind	-	-	PAS	2040
Hydro	-	-	LAM	2050
Solar	-	-	MEA	2060
-	-	-	CPA	2070
-	-	-	SAS	2080
-	-	-	-	2090
-	-	-	-	2100
-	-	-	-	2110
-	-	-	-	2120
-	-	-	-	2130
-	-	-	-	2140

Just as before all the subset variables and variables outside the constraints are removed in the new implementation. cap_invest is therefore the only new variable in the Gambit implementation of simple-model 3 and any new constraints apply to this new variable. Figure 11 shows an overview of the new implementation of the model.

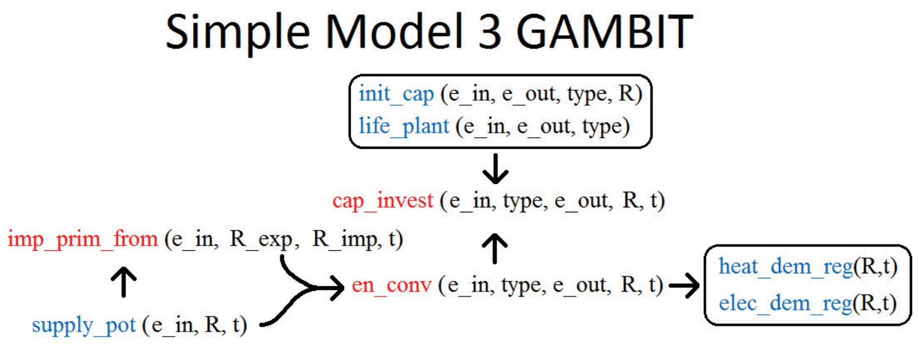

Simple Model 3 GAMBIT
init_cap (e_in, e_out, type, R)
life_plant (e_in, e_out, type)
cap_invest (e_in, type, e_out, R, t)
imp_prim_from (e_in, R_exp, R_imp, t)
en_conv (e_in, type, e_out, R, t)
supply_pot (e_in, R, t)
heat_dem_reg(R,t)
elec_dem_reg(R,t)

Figure 11: *The simple-model 3 implementation in GAMBIT. The variables, parameters and sets are described in Appendix C.*

The constraints are the same as in simple-model 2 (see formula 7 and 8) but one new constraint is added to make sure the cap_invest variable is large enough to cover the need of plants (see formula 9).

$$\sum_{e\_out} en\_conv(e\_in, e\_out, R, t) + \sum_{R\_exp} imp\_prim\_from(e\_in, R\_exp, R\_imp, t) - \sum_{R\_imp} imp\_prim\_from(e\_in, R\_imp, R\_exp, t) \leq supply\_pot(e\_in, R, t) \tag{7}$$

$$\sum_{e\_in} en\_conv(e\_in, e\_out, R, t) * effic(e\_in, e\_out, t) \geq dem\_reg(e\_out, R, t) \tag{8}$$

$$\begin{aligned} en\_conv(e\_in, e\_out, R, t) * effic(e\_in, e\_out, t) \\ -cLf * MPY * dep^{t-t\_erl} * cap\_invest(e\_in, e\_out, R, t) \\ \leq cLf * MPY * dep^{t-1} * init\_cap(e\_in, e\_out, type, R) \\ \\ where: \\ cLf = lf(e\_in, e\_out, type, R) \\ MPY = Msec\_per\_year = 3.6 \\ dep = e^{10*lg(1-(1/life\_plant(e\_in, e\_out, type))} \\ 1 \leq t\_erl \leq t \end{aligned} \tag{9}$$

In this formula 9 t_erl might need some more explanation. t_erl is apart of the set t but is looped over once for each time step t. The loop then goes from 1 to the current value of t. In the equations t refers to the index of the time step where 1990 is index 1 as it is the first. All this is needed to capture the old mechanism of the interaction between capacity and cap_invest in the GAMS implementation described previously.

The term lf in formula 9 is a constant table containing the so called load factors of the plants. A power plant's load factor is the fraction of it's maximum capacity over a year that can be utilized. This is due to many factors such as repairs and limitations on technologies them selves.

The term MPY is a constant containing an approximate value of the number of seconds on a year. The unit is Millions of Seconds per Year.

$$MPY = 60 * 60 * 24 * 365 \approx 3.6 Millionsofsecondsperyear \tag{10}$$

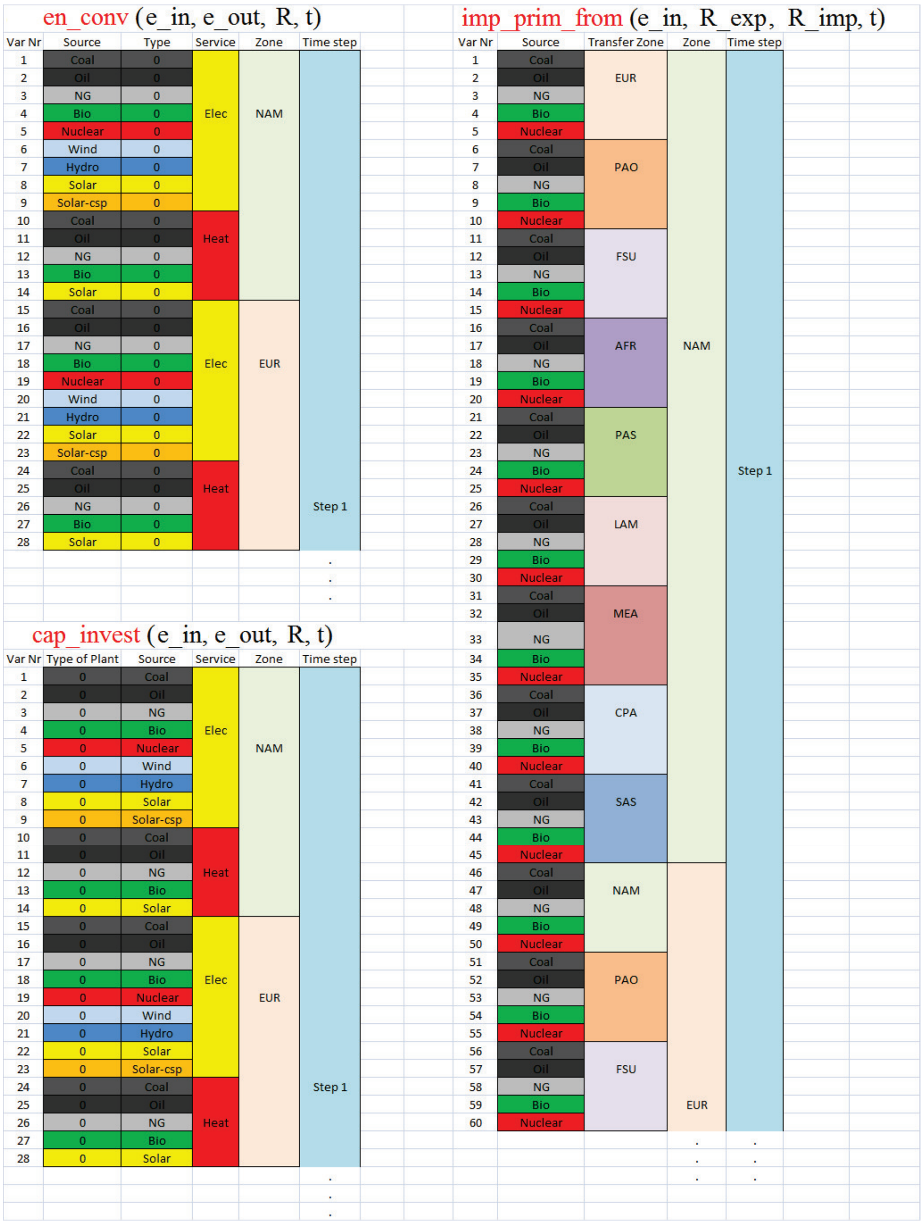
The objective function (see formula 11) is similar to the earlier version, but two things has changed. Firstly a coefficient called OM_cost is added to the first term of the objective function. It is the cost of using the plants and it is dependent on the amount of primary energy resources that is converted. It was therefore easiest to let the first term of the objective function handle this cost too as en_conv is there.

Secondly a new third term, cap_invest, is added to the objective function handling the cost of building new plants

$$\begin{aligned} min(t\_step * \sum_{\substack{e\_in,\\ e\_out,\\ R,t}} (en\_conv(e\_in, e\_out, R, t) * (D + E) \\ +(A + B + C) * imp\_prim\_from(e\_in, R\_exp, R\_imp, t) \\ +cost\_inv\_mod(e\_in, e\_out, type) * cap\_invest(e\_in, e\_out, R, t))) \\ \\ where: \\ A = imp\_cost(R\_exp, R\_imp) \\ B = imp\_cost\_lin(e\_in)) * distance(R\_exp, R\_imp) \\ C = price(e\_in, R\_exp) - price(e\_in, R\_imp) \\ D = price(e\_in, e\_out, t) \\ E = OM\_cost(e\_in, e\_out, type, t) \end{aligned} \tag{11}$$

cost_inv_mod is a constant table containing the cost of increasing the capacity of that particular kind of plant with one TW.

Below in figure 12 is the Excel version of the simple-model 3 variable layout used during the implementation. It shows as before what the variables are suppose to contain when the constraints are set right. Due to the expansion of the original sets in this model the table now does not show the complete variables but only a fraction of the first time step.



en_conv (e_in, e_out, R, t)

Var Nr	Source	Type	Service	Zone	Time step
1	Coal	0			
2	Oil	0			
3	NG	0			
4	Bio	0	Elec	NAM	
5	Nuclear	0			
6	Wind	0			
7	Hydro	0			
8	Solar	0			
9	Solar-csp	0			
10	Coal	0			
11	Oil	0	Heat		
12	NG	0			
13	Bio	0			
14	Solar	0			
15	Coal	0			
16	Oil	0			
17	NG	0			
18	Bio	0	Elec	EUR	
19	Nuclear	0			
20	Wind	0			
21	Hydro	0			
22	Solar	0			
23	Solar-csp	0			
24	Coal	0			
25	Oil	0	Heat		
26	NG	0			Step 1
27	Bio	0			
28	Solar	0			
					.
					.
					.

cap_invest (e_in, e_out, R, t)

Var Nr	Type of Plant	Source	Service	Zone	Time step
1	0	Coal			
2	0	Oil			
3	0	NG			
4	0	Bio	Elec		
5	0	Nuclear		NAM	
6	0	Wind			
7	0	Hydro			
8	0	Solar			
9	0	Solar-csp			
10	0	Coal			
11	0	Oil			
12	0	NG	Heat		
13	0	Bio			
14	0	Solar			
15	0	Coal			
16	0	Oil			
17	0	NG			
18	0	Bio	Elec		
19	0	Nuclear		EUR	
20	0	Wind			
21	0	Hydro			
22	0	Solar			
23	0	Solar-csp			
24	0	Coal			Step 1
25	0	Oil			
26	0	NG	Heat		
27	0	Bio			
28	0	Solar			
					.
					.
					.

imp_prim_from (e_in, R_exp, R_imp, t)

Var Nr	Source	Transfer Zone	Zone	Time step
1	Coal			
2	Oil	EUR		
3	NG			
4	Bio			
5	Nuclear			
6	Coal			
7	Oil	PAO		
8	NG			
9	Bio			
10	Nuclear			
11	Coal			
12	Oil	FSU		
13	NG			
14	Bio			
15	Nuclear			
16	Coal			
17	Oil	AFR	NAM	
18	NG			
19	Bio			
20	Nuclear			
21	Coal			
22	Oil	PAS		
23	NG			
24	Bio			Step 1
25	Nuclear			
26	Coal			
27	Oil	LAM		
28	NG			
29	Bio			
30	Nuclear			
31	Coal			
32	Oil	MEA		
33	NG			
34	Bio			
35	Nuclear			
36	Coal			
37	Oil	CPA		
38	NG			
39	Bio			
40	Nuclear			
41	Coal			
42	Oil	SAS		
43	NG			
44	Bio			
45	Nuclear			
46	Coal			
47	Oil	NAM		
48	NG			
49	Bio			
50	Nuclear			
51	Coal			
52	Oil	PAO		
53	NG			
54	Bio			
55	Nuclear			
56	Coal			
57	Oil	FSU		
58	NG			
59	Bio		EUR	
60	Nuclear			
			.	.
			.	.
			.	.

Figure 12: *What the variables en_conv, imp_prim_from and cap_invest are suppose to contain if all constraints are correctly defined. See List of Acronyms for explanation to the acronyms.*

5.6 Solving Times for the New and the Old Implementation

As simple model one, two and three are now all simplified and reimplemented into the new optimization environment it is time to see if this had any effect on the solving times of the model. Table 4 and 5 below show the solving-time of the original simple model 3 in the previous GAMS-CPLEX environment and the new simple model 3 in the new GAMBIT-GUROBI environment with two different solver methods. The numbers below are in CPU-time.

Table 4: *Solving Time of Simple Model 3 in CPLEX with the GAMS implementation*

Cores	1	2	4
Simplex	0.27	0.37	0.35
Dual Simplex	0.35	0.35	0.34

Table 5: *Solving Time of Simple Model 3 in GUROBI with the GAMBIT implementation*

Cores	1	2	4
Simplex	0.35	0.35	0.35
Dual Simplex	0.18	0.17	0.18

Below are some things worth noticing in the tables that will be discussed in more detail in the discussion.

From Table 4 it can be seen that when using one or two cores the solving times are similar between the Simplex and Dual Simplex method for CPLEX. But when 1 core is being used only, the GAMS-CPLEX environment solve simple model 3 faster with the Simplex method, even faster than with the Dual Simplex method and two cores.

From Table 5 it can be seen that regardless of how many cores that are being used GAMBIT-GUROBI can solve simple model 3 faster when using the Dual Simplex method, compared to the Simplex method.

Finally by comparing the two tables to each other it can be seen that the solving times of simple 3 in dual simplex can be halved by using the GAMBIT-GUROBI environment instead of GAMS-CPLEX.

Both CPLEX and GUROBI supports a third method called Barrier but it was very hard to extract reliable figures on those solving times. The barrier solving-time was approximately 0.4 seconds or 1.6 seconds for both models depending on if the CPU-time or the total elapsed time is measured.

5.7 The Smallest Linear Combination of the GET-model

Although reimplementing the whole model into scheme fell out of the time frame of this project it would give a more complete picture to show how the entire GET-RC 6.1 look like in this smallest linear combination. This new perspective of the GET-RC 6.1 model is found by analogically applying the same methods as explained earlier in the reimplementation of simple model 1 to 3.

Simple model 6 is identical to GET-RC 6.1, therefore the documentation on the simple models are used in this analysis too (Grahn et al, 2013). Between simple model 3 and 6 the model is enlarged in two steps. New in these simple model versions 4 and 5, are features as import and export of energy carriers, co-generation of electricity and heat, the option of carbon capture and storage, the carbon cycle module, the emission constraints, the discount factor, as well as the entire infrastructure and transport module. As before the green variables need to be identified and removed to simplify the calculations

The variable layout for simple model 5 and 6 are identical therefore only the constraints of simple model 5 are necessary to get the variable layout above. The new constraints introduced in simple model 6 only restricts the freedom of the variables that were already declared in simple model 5. These constraints should therefore be applied in another manner which is explained in more detail in the next section.

For a more complete and more mathematical explanation on how the green variables are identified and how the smallest linear combination of the GET-model is found, see Appendix A.

Figure 13 shows a flowchart of the final simplified or more dense GET-RC model. Below are also pictures of the final model's variable layout. These are Figures 14, 15, 16 and 17.

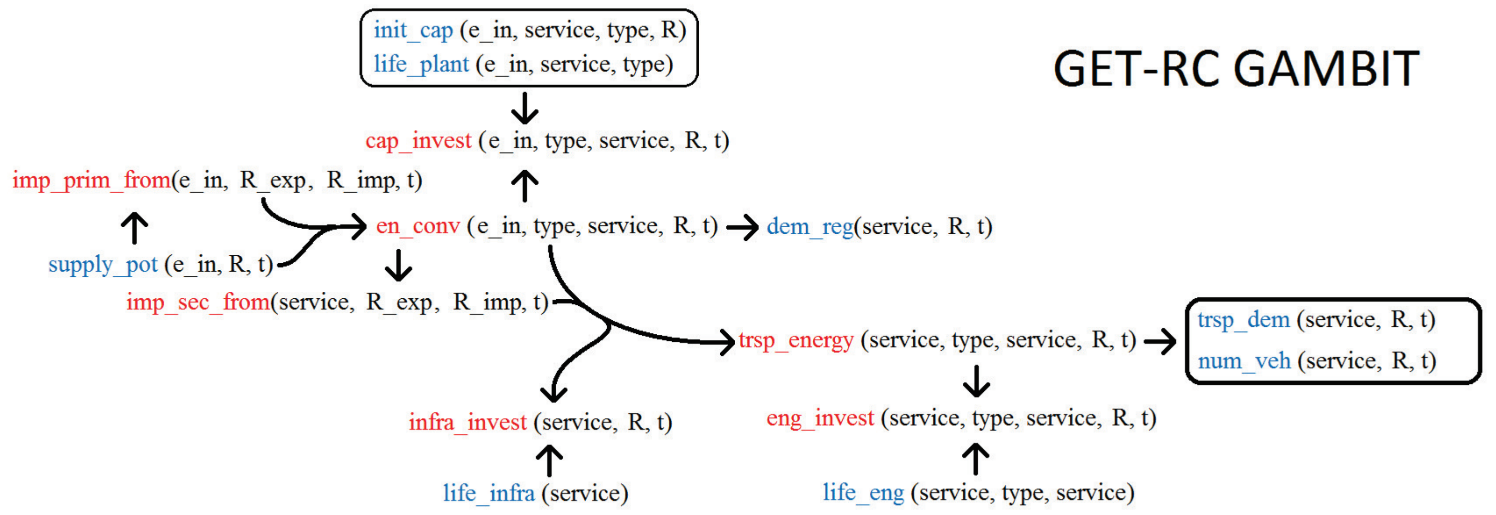
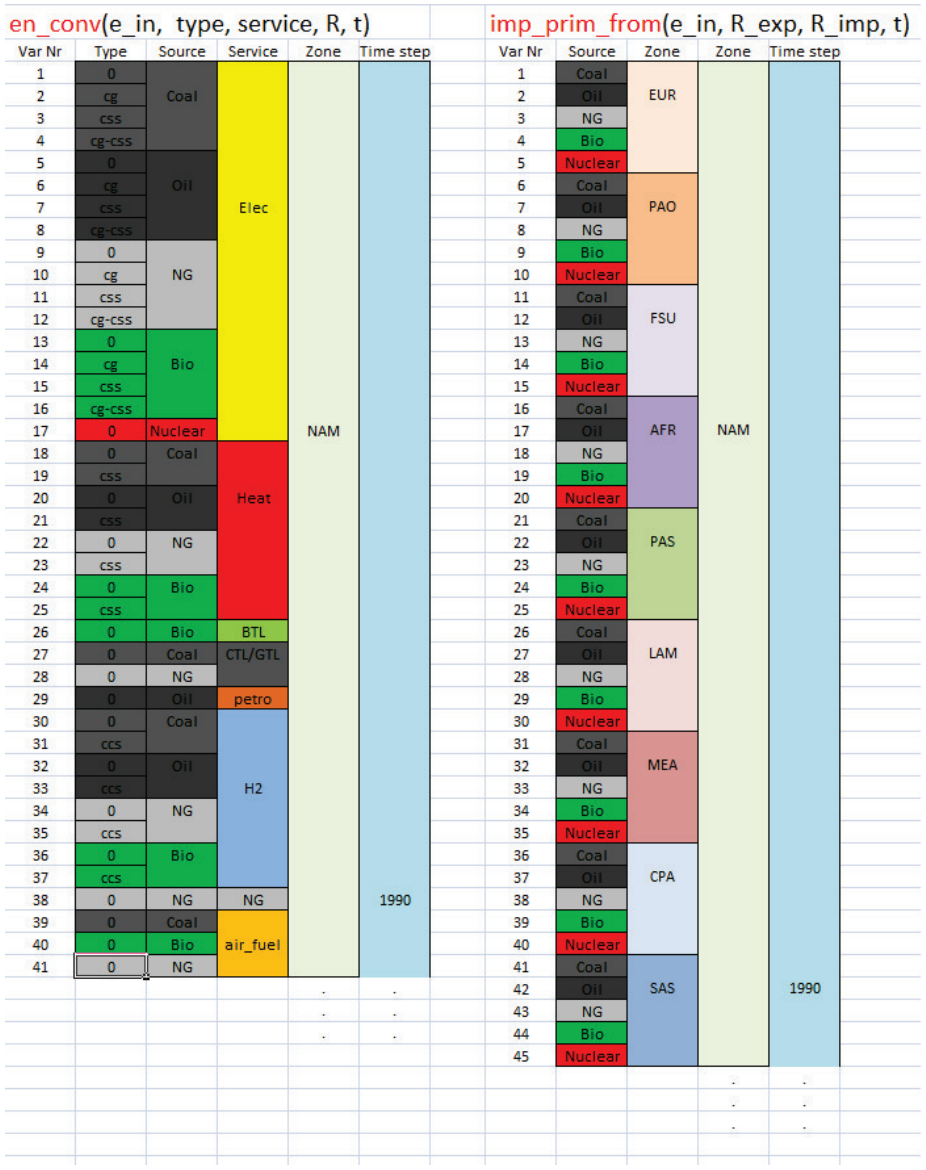

GET-RC GAMBIT
init_cap (e_in, service, type, R)
life_plant (e_in, service, type)
cap_invest (e_in, type, service, R, t)
imp_prim_from(e_in, R_exp, R_imp, t)
supply_pot (e_in, R, t)
en_conv (e_in, type, service, R, t)
dem_reg(service, R, t)
imp_sec_from(service, R_exp, R_imp, t)
trsp_energy (service, type, service, R, t)
trsp_dem (service, R, t)
num_veh (service, R, t)
infra_invest (service, R, t)
eng_invest (service, type, service, R, t)
life_infra (service)
life_eng (service, type, service)

Figure 13: *The GET-model as it is implemented in GAMBIT. The arrows in the figure represent the constraints and the red texts are the variables. Some more important constants are also showed in blue in the figure. The letters added in parentheses and separated by comma represents sets. All variables are part of the objective function. For description of constants see Appendix C. For description of sets and variables see Appendix A*



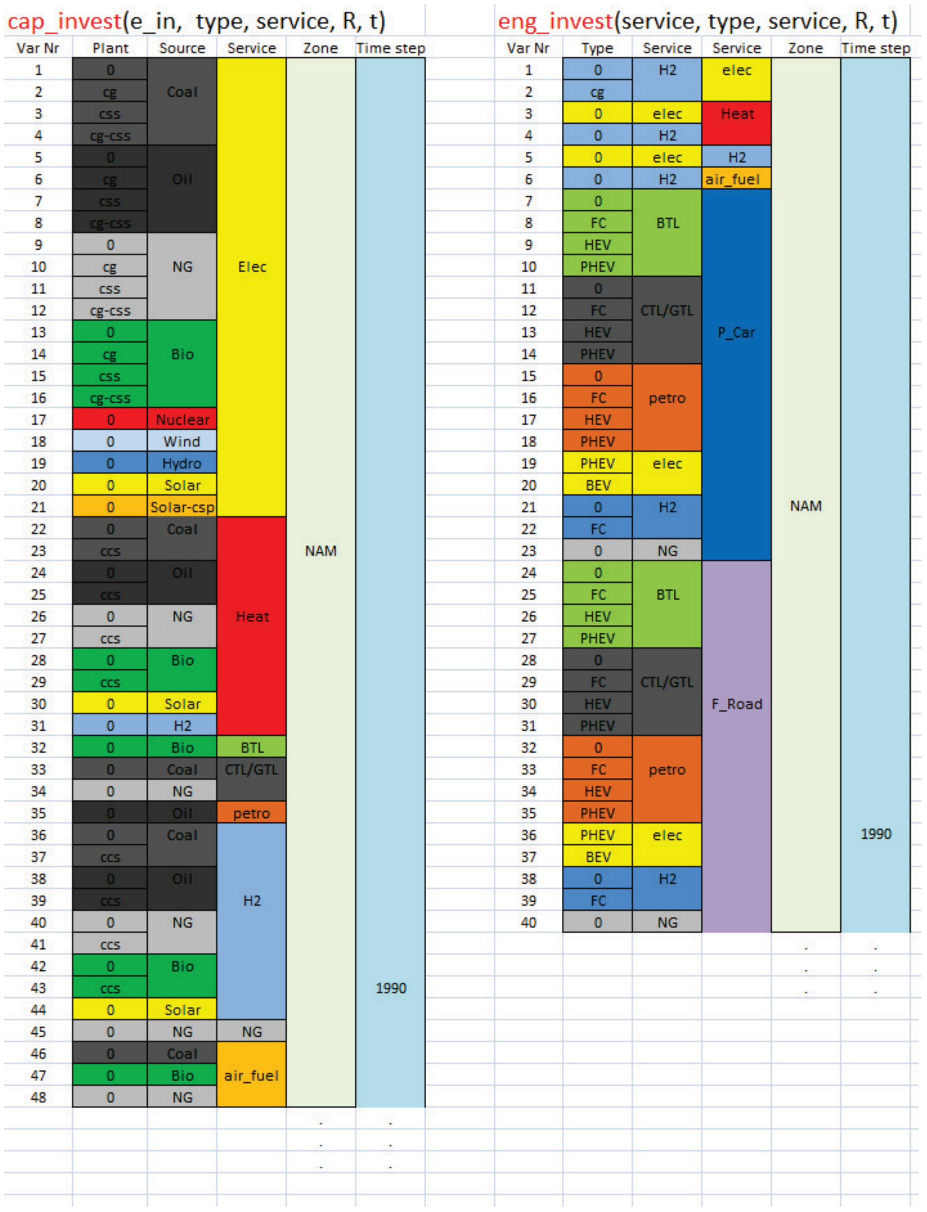
en_conv(e_in, type, service, R, t)

Var Nr	Type	Source	Service	Zone	Time step
1	0				
2	cg	Coal			
3	css				
4	cg-css				
5	0				
6	cg	Oil			
7	css		Elec		
8	cg-css				
9	0				
10	cg	NG			
11	css				
12	cg-css				
13	0				
14	cg	Bio			
15	css				
16	cg-css				
17	0	Nuclear		NAM	
18	0	Coal			
19	css				
20	0	Oil	Heat		
21	css				
22	0	NG			
23	css				
24	0	Bio			
25	css				
26	0	Bio	BTL		
27	0	Coal	CTL/GTL		
28	0	NG			
29	0	Oil	petro		
30	0	Coal			
31	ccs				
32	0	Oil			
33	ccs		H2		
34	0	NG			
35	ccs				
36	0	Bio			
37	ccs				
38	0	NG	NG		1990
39	0	Coal			
40	0	Bio	air_fuel		
41	0	NG			
				.	.
				.	.
				.	.

imp_prim_from(e_in, R_exp, R_imp, t)

Var Nr	Source	Zone	Zone	Time step
1	Coal			
2	Oil	EUR		
3	NG			
4	Bio			
5	Nuclear			
6	Coal			
7	Oil	PAO		
8	NG			
9	Bio			
10	Nuclear			
11	Coal			
12	Oil	FSU		
13	NG			
14	Bio			
15	Nuclear			
16	Coal			
17	Oil	AFR	NAM	
18	NG			
19	Bio			
20	Nuclear			
21	Coal			
22	Oil	PAS		
23	NG			
24	Bio			
25	Nuclear			
26	Coal			
27	Oil	LAM		
28	NG			
29	Bio			
30	Nuclear			
31	Coal			
32	Oil	MEA		
33	NG			
34	Bio			
35	Nuclear			
36	Coal			
37	Oil	CPA		
38	NG			
39	Bio			
40	Nuclear			
41	Coal			
42	Oil	SAS		1990
43	NG			
44	Bio			
45	Nuclear			
			.	.
			.	.
			.	.

Figure 14: *The contents of the variables en_conv and imp_prim_from in the simplified GET-RC 6.1 model. See List of Acronyms for explanation to the acronyms.*



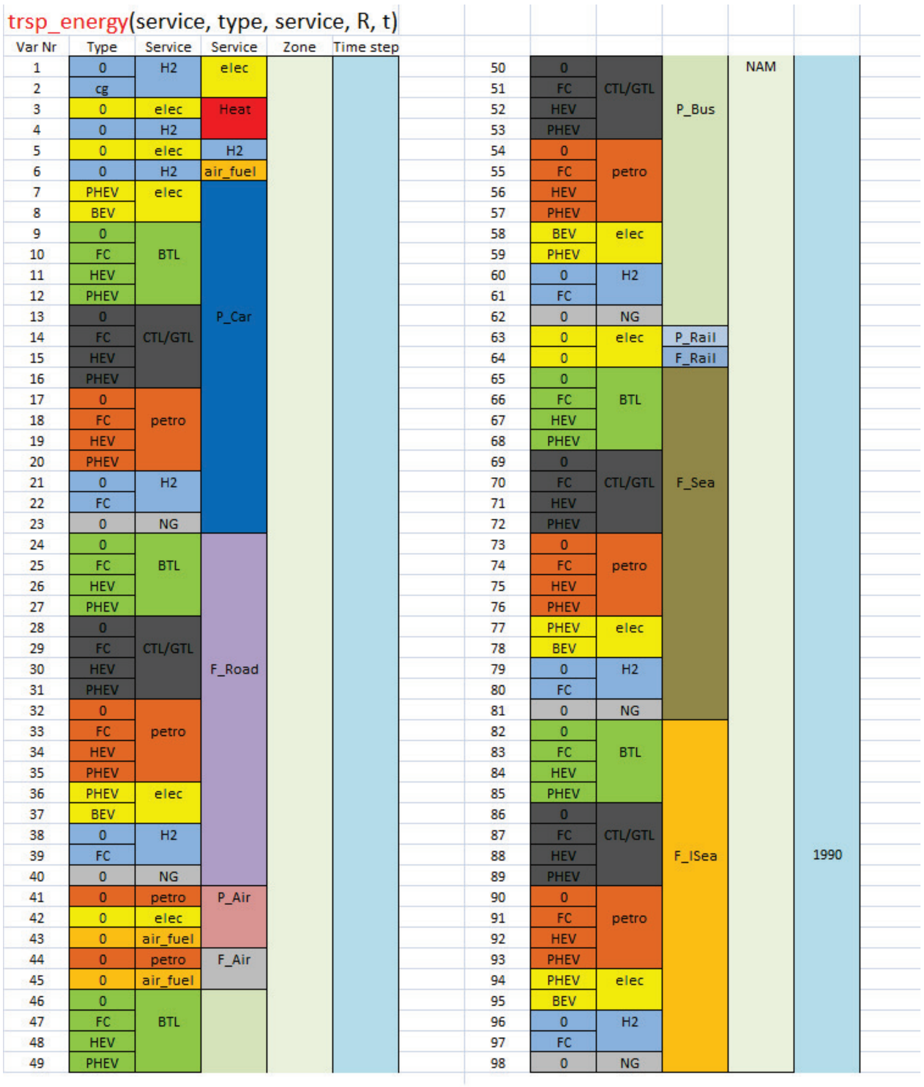
cap_invest(e_in, type, service, R, t)

Var Nr	Plant	Source	Service	Zone	Time step
1	0				
2	cg	Coal			
3	css				
4	cg-css				
5	0				
6	cg	Oil			
7	css				
8	cg-css				
9	0				
10	cg	NG	Elec		
11	css				
12	cg-css				
13	0				
14	cg	Bio			
15	css				
16	cg-css				
17	0	Nuclear			
18	0	Wind			
19	0	Hydro			
20	0	Solar			
21	0	Solar-csp			
22	0	Coal			
23	ccs			NAM	
24	0	Oil			
25	ccs				
26	0	NG	Heat		
27	ccs				
28	0	Bio			
29	ccs				
30	0	Solar			
31	0	H2			
32	0	Bio	BTL		
33	0	Coal	CTL/GTL		
34	0	NG			
35	0	Oil	petro		
36	0	Coal			
37	ccs				
38	0	Oil			
39	ccs		H2		
40	0	NG			
41	ccs				
42	0	Bio			
43	ccs				1990
44	0	Solar			
45	0	NG	NG		
46	0	Coal			
47	0	Bio	air_fuel		
48	0	NG			
				.	.
				.	.
				.	.

eng_invest(service, type, service, R, t)

Var Nr	Type	Service	Service	Zone	Time step
1	0	H2	elec		
2	cg				
3	0	elec	Heat		
4	0	H2			
5	0	elec	H2		
6	0	H2	air_fuel		
7	0				
8	FC	BTL			
9	HEV				
10	PHEV				
11	0				
12	FC	CTL/GTL			
13	HEV		P_Car		
14	PHEV				
15	0				
16	FC	petro			
17	HEV				
18	PHEV				
19	PHEV	elec			
20	BEV				
21	0	H2		NAM	
22	FC				
23	0	NG			
24	0				
25	FC	BTL			
26	HEV				
27	PHEV				
28	0				
29	FC	CTL/GTL			
30	HEV		F_Road		
31	PHEV				
32	0				
33	FC	petro			
34	HEV				
35	PHEV				
36	PHEV	elec			1990
37	BEV				
38	0	H2			
39	FC				
40	0	NG			
				.	.
				.	.
				.	.

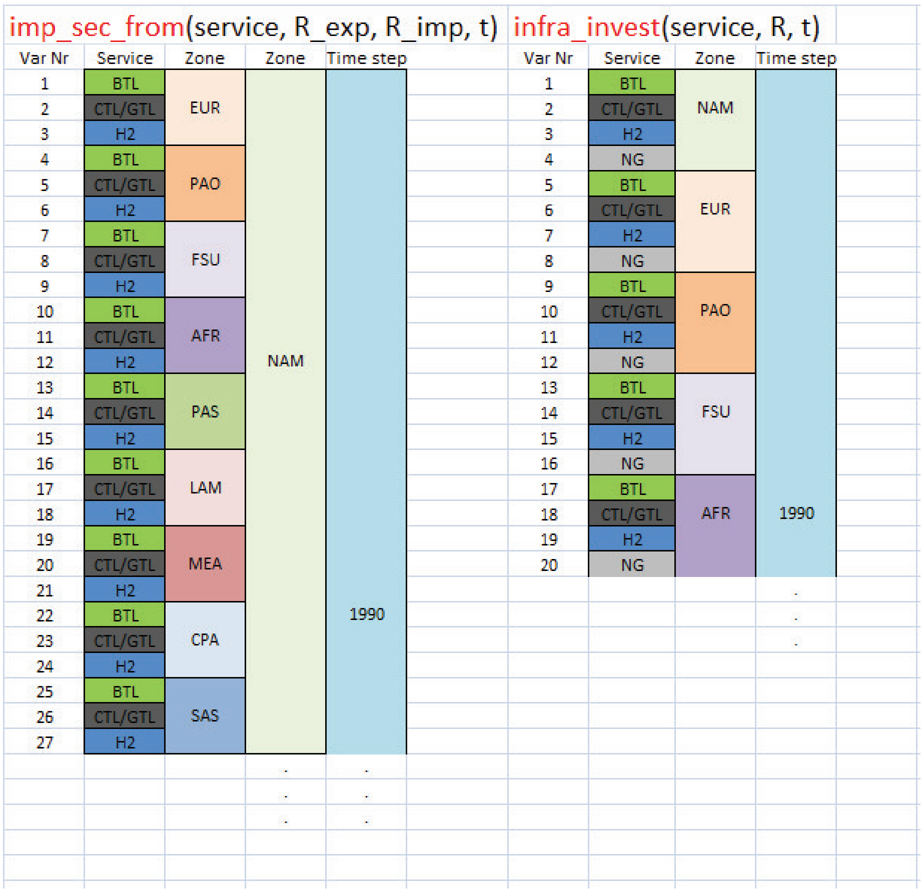
Figure 15: *The contents of the variables cap_invest and eng_invest in the simplified GET-RC 6.1 model. See List of Acronyms for explanation to the acronyms.*



trsp_energy(service, type, service, R, t)

Var Nr	Type	Service	Service	Zone	Time step
1	0	H2	elec		
2	cg				
3	0	elec	Heat		
4	0	H2			
5	0	elec	H2		
6	0	H2	air_fuel		
7	PHEV	elec	P_Car		
8	BEV				
9	0	BTL			
10	FC				
11	HEV				
12	PHEV				
13	0	CTL/GTL			
14	FC				
15	HEV				
16	PHEV				
17	0	petro			
18	FC				
19	HEV				
20	PHEV				
21	0	H2			
22	FC				
23	0	NG			
24	0	BTL	F_Road		
25	FC				
26	HEV				
27	PHEV				
28	0	CTL/GTL			
29	FC				
30	HEV				
31	PHEV				
32	0	petro			
33	FC				
34	HEV				
35	PHEV				
36	PHEV	elec			
37	BEV				
38	0	H2			
39	FC				
40	0	NG			
41	0	petro	P_Air		
42	0	elec			
43	0	air_fuel			
44	0	petro	F_Air		
45	0	air_fuel			
46	0	BTL			
47	FC				
48	HEV				
49	PHEV				
50	0	CTL/GTL	P_Bus	NAM	1990
51	FC				
52	HEV				
53	PHEV				
54	0	petro			
55	FC				
56	HEV				
57	PHEV				
58	BEV	elec			
59	PHEV				
60	0	H2			
61	FC				
62	0	NG			
63	0	elec	P_Rail		
64	0		F_Rail		
65	0	BTL	F_Sea		
66	FC				
67	HEV				
68	PHEV				
69	0	CTL/GTL			
70	FC				
71	HEV				
72	PHEV				
73	0	petro			
74	FC				
75	HEV				
76	PHEV				
77	PHEV	elec			
78	BEV				
79	0	H2			
80	FC				
81	0	NG			
82	0	BTL	F_ISea		
83	FC				
84	HEV				
85	PHEV				
86	0	CTL/GTL			
87	FC				
88	HEV				
89	PHEV				
90	0	petro			
91	FC				
92	HEV				
93	PHEV				
94	PHEV	elec			
95	BEV				
96	0	H2			
97	FC				
98	0	NG			

Figure 16: *The contents of the variable trsp_energy in the simplified GET-RC 6.1 model. See List of Acronyms for explanation to the acronyms.*



imp_sec_from(service, R_exp, R_imp, t)

Var Nr	Service	Zone	Zone	Time step
1	BTL			
2	CTL/GTL	EUR		
3	H2			
4	BTL			
5	CTL/GTL	PAO		
6	H2			
7	BTL			
8	CTL/GTL	FSU		
9	H2			
10	BTL			
11	CTL/GTL	AFR		
12	H2		NAM	
13	BTL			
14	CTL/GTL	PAS		
15	H2			
16	BTL			
17	CTL/GTL	LAM		
18	H2			
19	BTL			
20	CTL/GTL	MEA		
21	H2			
22	BTL			1990
23	CTL/GTL	CPA		
24	H2			
25	BTL			
26	CTL/GTL	SAS		
27	H2			
			.	.
			.	.
			.	.

infra_invest(service, R, t)

Var Nr	Service	Zone	Time step
1	BTL		
2	CTL/GTL	NAM	
3	H2		
4	NG		
5	BTL		
6	CTL/GTL	EUR	
7	H2		
8	NG		
9	BTL		
10	CTL/GTL	PAO	
11	H2		
12	NG		
13	BTL		
14	CTL/GTL	FSU	
15	H2		
16	NG		
17	BTL		
18	CTL/GTL	AFR	1990
19	H2		
20	NG		
			.
			.
			.

Figure 17: *The contents of the variables imp_sec_from and infra_invest in the simplified GET-RC 6.1 model. See List of Acronyms for explanation to the acronyms.*

5.7.1 The Constraints of Simple 6

Simple model 6 add a number of constraints on the model that shapes the model results to what actually was done in society the last couple of years. It has never been said that the GET model can tell us about how the future will be it simply tells us the least costly way to reduce the carbon dioxide emissions out of today's perspective under different technology and cost assumptions.

But as people do not always take the least costly way and as changes are not always made so fast something needed to be done to the starting conditions and constraints of the model to make it still valid today. These conditions made the original implementation go from taking a fraction of a second to a few minutes.

In the new implementation it is easier to split the GET-model into different sections based on fractions of the models sets. This is mainly due to the fact that the model can be made to loop over a subset of the original sets. By changing an index or so at the top of the program code the model can be made to start the optimization at 2010 instead of 1990. An unexpected strength of the new model is therefore that some of new constraints introduced in simple 6 will turn into boundary conditions instead of optimization constraints.

The rest of the constraints in simple 6 introduces a couple of upper and lower bounds on the model variables. These constraints do not fall out as boundary conditions still they should not be as devastating to the calculation time as in the original implementation as they will have to be rewritten to act more on the model core (the red variables) rather than the model crust (the green variables). The new implementation of simple model 6 will most likely be slower than the new simple model 5 but this time difference should be smaller than in the original implementations of simple 5 and 6.

5.7.2 Calculating the Values of the Green Variables

As described earlier the green variables were removed in the reimplementation. This mainly because they were expected to slow down the calculation process without adding any new degrees of freedom that were needed.

It should be pointed out however that, as the new implementation is mathematically equivalent to the previous, a simple script can be made to calculate the values of all the green variables from the values of the seven red variables.

That script should be fairly straight forward to make but for some directions see the explanations of how the green variables were removed in the descriptions of the reimplementations of simple 1 throughout 3. The report describing the original implementation could also be of help (Grahn et al 2013).

5.7.3 A new view of the GET-model

Throughout this report an attempt has been made at describing the model in the same terminology as it has been viewed in previous publication and reports by physical resource theory. This has sometimes had the consequence that certain points were hard to make and show to the reader. Approximations were needed to display the reimplemented model in the old fashion.

In Appendix A therefore can be found an attempt at describing the new implementation of the GET-model in a more strictly mathematical point of view, following the process described in the sections Method and Results in this report.

5.8 Pre-requisites for Carbon Dioxide based Synthetic Fuels

In the original GET-RC model version it is found that water splitting into hydrogen and oxygen using solar energy is a cost-effective solution in a future carbon constrained world (Grahn 2009). This is the model's way to be able to use a large amount of intermittent energy, since wind and direct solar electricity together are maximized to 30% of a region's electricity demand.

It would be interesting to see if the model might shift from storing intermittent energy in hydrogen to instead store the energy in carbon dioxide based synthetic fuels, and if so under what circumstances and in what regions this would be most used.

Since there is no commercial production of synthetic fuels from carbon dioxide and water today, it is not possible to know the exact data on for example conversion efficiencies and investment costs of the different conversion facilities. To analyze the role of these fuels it is necessary to make guesses and test how the results depend on changing these guesses in a range of sensitivity analyzes. For more about synthetic fuels from carbon dioxide and water see (Mohseni 2012) and (Graves et al 2011).

When developing the model in order to analyze the role of synthetic fuels from carbon dioxide and water we need to make assumptions on:

- The cost of extracting carbon dioxide from different sources.
- The cost of producing fuels from carbon dioxide and water which may include
  - pre-treatment of the carbon dioxide (remove unwanted molecules, e.g., sulphure) and if needed convert carbon dioxide to carbon monoxide
  - produce hydrogen from water (electrolysis and other)
  - a facility for mixing carbon dioxide and hydrogen or carbon monoxide and hydrogen, to produce synthetic fuels
  - eventual investments for co-generation of fuels and heat that can be sold for district heating.
- For all investments done we need data on
  - the capital cost of each investment,
  - the energy use for the process (conversion efficiency),
  - the life time of the investment,
  - a capacity factor, i.e., how many hours per year a facility may run on full capacity and
  - the operation and maintenance costs.

Firstly it should be relatively easy to find data on the cost of extracting carbon dioxide out of exhaust gases or pure carbon dioxide streams (from for example ethanol or biogas production facilities) but much harder to find reliable date on the cost of extracting carbon dioxide from the atmosphere. Both alternatives are interesting but out of the long term perspective only the later. To start the development however exhaust gases are interesting too in order to reduce the emissions during the phasing out of fossil fuels.

Splitting of carbon dioxide into carbon monoxide and oxygen is not a very well developed technology. Some more theoretical studies was found during this project but there is currently a large uncertainty regarding what cost that should be implemented in the model (Miller 2007; Stoots et al 2009). More research in order to develop the technology is needed.

The technology for producing hydrogen from water via electrolysis is well known, but more advanced technologies that have the potential of producing hydrogen a lower cost is still under development and therefore uncertain to estimate.

The investment cost of the facility that can convert the hydrogen and carbon monoxide mix (or the hydrogen and carbon dioxide mix) to synthetic fuels (e.g. methane, methanol, DME, gasoline or diesel) is uncertain but in (Mohseni 2012) estimated to approximately 1800 EUR/kW, including the electrolyzer, the sabatier tehnology for methane synthesis from hydrogen and carbon dioxide, gas storage and district heater connection.

One option to get this new production technology into the model might be to separate current gasification technology (that in the model leads to the production of synthetic fuels BTL, CTL or GTL) into two processes, one facility producing the syngas and one facility where the syngas is synthesized into the fuel. The stand alone FT process could then be fed by either the syngas produced from carbon dioxide or the syngas of the gasification processes.

6 Discussion

As is showed above the GAMS and GAMBIT systems are fairly similar in speed in the Simplex case. The reason for why CPLEX in the GAMS system has different speeds for the number of cores is hard to say. It is probably due to the time needed to set up the problem rather then the solving time.

In the Dual Simplex case however the new GUROBI-GAMBIT system with the smallest linear combination is about twice as fast. This makes sense as the new version of the model should be more adapted to the computer and thereby decrease the workload of solving the problem.

This more computer orientated implementation does not necessary mean that it is more difficult to work with the new environment than GAMS. Setting up the environment and making the optimizer available to the user is difficult but to use the new environment to further develop the GET-model is not as difficult.

It may require a more mathematical approach as scheme is a functional programming language but the whole point of functional programming is to make it easier for the user to command the computer do all the hard work. As that is already done all that is needed to further develop the model is to add variables missing from simple 3 to simple 5. This work should be of a more mechanical nature as the last variables can be added in a very similar way as the variables previously implemented in the model.

The use of the smallest linear combination drastically reduced the number of variables and is likely to have even greater effect as simple 4 throughout 6 are implemented. This is mainly because the largest variable of the mathematical interpretation of the GET-RC model is trsp_energy. It is more than twice as big as any other variable in the model. This means that very many green variables were declared to handle this function in the GAMS environment.

There are therefore good reasons for believing that the time-gap between the environments will increase slightly when more of the model is implemented, leading to that the new system might become more then twice as fast. As asked for the new environment is also built only on open-source apart from GUROBI which however is free for academic use.

Some more details should be given on why Barrier in the end was excluded from the results section. It would have been possible to extract some numbers of the Barrier time as well but much less reliable then those of Simplex and Dual Simplex as mentioned in the results. However the GET-model in its entreaty too is a small model. Even with Simple 6 it is possible that the Barrier optimization method would spend most time on setting up the calculations and not counting.

As dual simplex is a simpler solving method it uses more time to calculate and less on setting up the problem. Dual simplex should therefore say more of the environment setting up the problem rather then the optimizer. As this is supposed to be an evaluation of GAMS rather than CPLEX barrier was excluded as that data is both less reliable due to the problems of extracting data and less relevant.

7 Conclusions

As the new GUROBI-GAMBIT system written in Scheme was proven to be twice as fast as the original for simple model 3 it should be safe to conclude that a complete GET-RC 6.1 reimplementation atleast should not be slower than the original GAMS implementation. If any the new implementation should be faster even when the final constraints of simple 6 are implemented.

The fact that GAMBIT is open-source and GUROBI free for academic use should also be to the new systems advantage. The new GUROBI-GAMBIT environment is simply twice as fast and for free.

A mathematical interpretation of the GET-model has been found which can be implemented in the new environment. It is built up by 7 variables instead of the about 45 variables of the original GAMS implementation. Due to this less constraints were needed too.

Pre-requisites needed to implement fuels from carbon dioxide and water has been found. More research to get good data is needed as a commercial technology for making carbon monoxide from carbon dioxide was not found. The ones in existence were more on the experimental stage still.

However, model runs to get interesting insights can be made from testing a range of cost assumptions and shed some light of under what circumstances synthetic fuels from carbon dioxide and water can be a cost-effective solution for the transportation sector.

7.1 Conclusions in Short:

- The new version of the model is twice as fast as the original and for free.
- The model are most likely faster than the original when the remaining 5 variables of the mathematical interpretation are implemented too.
- A mathematical interpretation of the complete GET-RC 6.1 model has been found.
- Pre-requisites needed for implementing carbon dioxide based fuels has been identified.

8 Further Work

What can be done in the future is to complete the reimplementation of the model into the new GAMBIT-GUROBI optimization environment. If it would be necessary to use the solution method Barrier to solve Simple 6 a little faster the new Environment already supports this solution method. Some work can also be devoted to develop the new environment with perhaps a graphical user interface.

An earlier implementation in the environment of the sparse matrix format would be good too. As the first two simple models had 48 respectively 72 degrees of freedom it was not judged necessary to implement a system for virtual matrixes. Instead in the constraint matrix of simple model 2 for example 72 zeros are declared for every constraint and then 2 or 4 of these were changed to something else than zero in each row.

Simple model 6 has 48480 degrees of freedom so declaring 48480 zeros for each constraint in the constraint matrix is not an option. The environment is however built in many small sections interacting with each other at the beginning and end of a task but otherwise working independently. It should therefore be relatively easy to make changes to the construction of the constraint matrix so that one main vector of many sub vectors is declared once in the beginning instead of one giant matrix mainly containing zeros.

The main vector should contain 48480 sub vectors. The constraints are then added to these sub vectors as a set of sub sub vectors with two elements each. The system here is that the sub vectors defines the columns of the virtual matrix while the sub sub vectors defines the row number as one of there entries. The other entry in a sub sub vector should contain the desired value at that location in the constraint matrix.

This virtual vector can be searched through much faster to find the entries that are to be forwarded to the optimizer.

- Andersson M. (2012) On this page you can find the GUROBI FFI developed in this project. It is item 6 under the Math heading. `http://dynamo.iro.umontreal.ca/wiki/index.php/Dumping_Grounds`

- Azar C., Lindgren K. and Andersson B. A. (2003) *Global energy scenarios meeting stringent CO2 constraints – cost-effective fuel choices in the transportation sector* Energy Policy 31(10): 961–976.

- Berndes G., Hoogwijk M. and van den Broek R. (2003) *The contribution of biomass in the future global energy supply: a review of 17 studies.* Biomass & Bioenergy 25(1): 1-28.

- Börjesson P. (2007) *Produktionsförutsättningar för biobränslen inom svenskt jordbruk* [Production conditions of bioenergy in Swedish agriculture] and *Förädling och avsättning av jordbruksbaserade biobränslen* [Conversion and utilisation of biomass from Swedish agriculture]. Two reports (Lund reports No 61 and 62) included as Appendix to Bioenergi från jordbruket – en växande resurs [Appendix to Bioenergy from Swedish agriculture – a growing resource], Statens offentliga utredningar 2007:36. Jordbrukets roll som bioenergiproducent Jo 2005:05. Available at `http://www.regeringen.se/content/1/c6/08/19/74/5c250bb0.pdf`.

- Carbon Recycling International (2012) `http://www.carbonrecycling.is/`

- Godfray H.C.J. (2010) *Food Security: The Challenge of Feeding 9 Billion People.* SCIENCE, vol. 327, ss. 812-817.

- Grahn M. (2009) *Cost-effective fuel and technology choices in the transportation sector in a future carbon constrained world - results from the Global Energy Transition (GET) model. Thesis for the degree of Doctor of Philosophy in Energy and Environment.* Physical Resource Theory, Chalmers, Sweden.

- Grahn M., Azar C., Williander M.I., Anderson J.E., Mueller S.A., Wallington T.J. (2009) *Fuel and Vehicle Technology Choices for Passenger Vehicles in Achieving Stringent CO2 Targets: Connections between Transportation and Other Energy Sectors* Environmental Science and Technology (ES&T) 43(9) 3365-3371.

- Grahn M., Klampfl E., Whalen M.J., Wallington T.J., Lindgren K. (2013) *Model description of the linearly programmed long-term energy systems cost-minimizing model GET-RC 6.1.* Report. Physical Resource Theory, Chalmers, Sweden.

- Graves C., Ebbesen S.D., Mogensen M., Lackner K.S. (2011) *Sustainable hydrocarbon fuels by recycling CO2 and H2O with renewable or nuclear energy.* Renewable and Sustainable Energy Reviews 15: 1–23

- Grübler A. (1998) *Technology and Global Change.* Cambridge: Cambridge University Press.

- Hedenus et al (2010) *Cost-effective energy carriers for transport - the role of the energy supply system in a carbon-constrained world.* International Journal of Hydrogen Energy 35 (10) pp. 4638-4651.
- Holmberg J. (1995) *Socio-Ecological Principles and Indicators for Sustainability.* PhD Thesis, Physical Resources Theory, Chalmers University of Technology, Sweden.
- Huang C., T-Raissi A. (2005) *Analysis of sulfur-iodine thermochemical cycle for solar hydrogen production. Part I: decomposition of sulfuric acid Solar Energy,* vol. 78, ss. 632-646.
- IEA (2012) *International Energy Agency Statistics. Global primary energy supply.* Available at `http://www.iea.org/stats/pdf_graphs/29TPES.pdf`
- IPCC, Solomon S. D., Qin M., Manning Z., Chen M., Marquis K.B., Averyt M., Tignor and Miller H.L. (eds.) (2007) *Summary for policymakers. In: Climate Change 2007: The Physical Science Basis. Contribution of Working Group I to the Fourth Assessment Report of the Intergovernmental Panel on Climate Change* Cambridge University Press, Cambridge, United Kingdom and New York, NY, USA. Available at `http://www.ipcc.ch/pdf/assessment-report/ar4/wg1/ar4-wg1-spm.pdf`
- Kelsey R., Clinger W. & Rees J. (1998) *Revised^5 Report on the Algorithmic Language Scheme*
- Kernighan B. W., Ritchie D. M. (1988) *The C Programming Language Second Edition* Courier Stoughton in Stoughton, Massachusetts
- Miller J.E. (2007) *Initial Case for Splitting Carbon Dioxide to Carbon Monoxide and Oxygen* Springfield: U.S. Department of Commerce, National Technical Information Service.
- Mohseni F. (2012). *Power to gas – bridging renewable electricity to the transport sector.* Licentiate Thesis in Chemical Engineering and Technology, KTH, Stockholm.
- More M. (2008) *A Tour of Scheme in Gambit* Granted for free unlimited distribution presuming these original notices are maintained. Distributed in PDF, HTML and Microsoft Word formats.
- Nino J., Hosch F. A. *An Introduction to Programming and Object-Oriented Design Using Java Third Edition* John Wiley & Sons, inc.
- Nordling C., Sterman J. *Physics Handbook for Science and Engineering*
- de Pater I., Lissauer J.J. (2001) *Planetary Sciences.* Cambridge: Cambridge University Press.
- Ponting C. (2007) *A New Green History of the World* Vintage Books London

- Rosenthal R.E. (2012) *GAMS - A User's guide. Tutorial.* GAMS Development Corporation, Washington, DC, USA. December 2012. Available at: `http://www.gams.com/dd/docs/bigdocs/GAMSUsersGuide.pdf`
- Salomon D. (1993) *Assemblers and Loaders* Available at: `http://www.davidsalomon.name/assem.advertis/asl.pdf`
- Sasol, for synfuel production history see: `http://www.sasol.com/sasol_internet/frontend/navigation.jsp?navid=21300010&rootid=2` or main page at `http://www.sasol.com/`
- Stoots C.M., O'Brien J.E, Condie K.G., Hartvigsen J.J (2010) *High-temperature electrolysis for large-scale hydrogen production from nuclear energy - Experimental investigations.* SCIENCE, vol. 327, ss. 812-817.
- Vitart X., Le Duiguo, A., Carles, P. (2006) *Hydrogen production using the sulfur-iodine cycle coupled to a VHTR: An Overview Energy Conversion and Management,* vol. 47, ss. 2740-2747.
- Wayne V. (2009) *Landmarks for Sustainability.* Sheffield: Greenleaf Publishing Ltd.

9 Appendix A - The Reimplementation in Mathematical Terms

9.1 Introduction - What is an Optimization Environment

As explained earlier, when the GET-Model was reimplemented another structure and terminology was used to sort and calculate the data then what has been used in the description of the original GAMS implementation of the GET-model (see Grahn et al 2013; Grahn 2009). For better understanding the original terminology was used in the main parts of this report.

This however came with a price, although the reader might recognize things easier, the original terminology is not wide enough to capture some of the new concepts introduced in the reimplementation and can therefore be misleading when describing the new implementation. Below therefore follows an explanation of the new structure of the GET-model with the use of a slightly more algebraic orientated terminology.

But first, to be able to rewrite the GET-model to a functional programming language, or any programming language other than the GAMS language, a mathematical interpretation of the model was needed. More precisely the GAMS code of the GET-model does not contain the explicit structure of how the computer should execute the model.

Instead the original programming code is a set of instructions to GAMS defining what the model should describe. It is then up to the optimization environment, GAMS, to construct an executable program code out of these instructions.

Understanding human philosophy and translate this into a mathematical model and back again is not a computers strong suit. Therefore, to make this collaboration between man and machine work, the instructions about the GET-models functionality given to GAMS must be made in a certain way. Most of the philosophy and all the considerations must be left for the programmer to handle when the instructions to GAMS are made.

The computer can then however handle more low level programming functions such as repeating certain code over again or syntax needed for the model. GAMS can for example rewrite the GET-model to fit with many different optimizers. This turns GAMS into something of a compiler, described earlier in section 3.1, as GAMS handle the more low level programming for the user. GAMS however does a little than a mare compiler as it also presents the results for the user in a graphical interface and handle the relation with the optimizer or solver. Thus GAMS is not just a compiler but an optimization environment.

Because of this big work done by GAMS the first thing that was needed to rewritable the GET-model to a new language was a mathematical mapping or interpretation of the GET-model which a computer could execute without the help of an optimization environment.

Finding this mathematical interpretation of the GET-RC 6.1 is what has previously been described as ”simplifying the GET-model’s algorithms” or ”finding the smallest linear combination”. It is also this process that has lead forth to this new terminology for describing the GET-model which is introduced in this appendix. It is the mapping described here that has been the foundation for the final simplified model described in the results section 5.7.

The two biggest differences between the original terminology for describing the GET-model and the terminology of this appendix are explained in two separate sections below called ”Unit Vectors vs. Sets” (9.2.1) and ”Systems vs. Variables” (9.2.2). After that the process of identifying red and green variables are shown in section 9.3 and finally the new implementation is described with the new terminology in section 9.4.

9.2 The New Environment - Why the new implementation was faster

The new terminology or structure of the GET-model described in this appendix is also used in the new programming code. The new model code are showed in full in appendix B including simple model one, two and three. A difference is that the new model code contain both the functionality of an optimization environment, previously handled by GAMS, and the more explicit GET-model code.

In this way the mapping of the GET-model was also used to construct what can be called the new optimization environment. To make the model mapping and the new optimization environment fit together however certain changes were made compared to GAMS which is the reason for why section 9.3 and 9.4 in this appendix are needed. It was found that some of the work previously done by the optimization environment, GAMS, was more a good job for the user rather then something to leave to the environment.

This is the main reason for way the new implementation of the GET-model is running faster on a computer than the original GAMS implementation. The new optimization environment is written specially for the GET-model. It requires a little more of the user but in return gives more freedom for the composition of the model. This is described in more detail in the upcoming sections.

9.2.1 Unit Vectors vs. Sets - The GET-Model Room

By studying the model description of GET-RC 6.1 (Grahn 2009) it was found that the GET-model could be described as a linear combination of five (here in this Appendix called) unit vectors. In GAMS such unit vectors are called sets or rather a set is a collection of some or all entries available along one unit vector axis.

For example the GET-model have a set T containing all the time steps of the model but also a smaller set t containing a subset of all the time steps in the model. In the new environment only one unit vector is declared for time which is common for the whole model. How this effects the variables are described in the next section.

The term unit vectors refers to the point of view where the solution to the problem is a GET-model. In short to get an overview of what degrease of freedom that the GET-model needs, a room was defined within which all the variables of the simple models and GET-RC 6.1 could fit. The models can then be plotted as dots in this room. The room is five dimensional and spanned by the unit vectors.

This five dimensional point of view is a simplification that can only describe different GET-models and there variables, not explicit solutions to any of these models. In section 9.3 it will however be shown how this GET-model room can be used to identify red and green variables.

To capture a solutions to the complete GET-RC 6.1 model a room of approximately 48480 dimensions are needed as that is the approximate number of degrees of freedom in the final model. However only the first quadrant of such a 48480 dimensional room is needed as all degrees of freedom in the model are defined as equal to or larger than zero.

9.2.2 Systems vs. Variables

The five dimensional GET-model room described in the previous section can now be used to group the 48480 degrees of freedom of the GET-RC 6.1 model into smaller more manageable packages. In the original GAMS implementation as explained earlier these packages are refereed to as variables causing some confusion when one is to describe the new structure of the model.

Therefore the individual values or enteritises within the variables of GAMS are here in this appendix called elements. Out of the computers point of view these individual elements are the variables of the model but to not cause confusion here they will be referred to as elements anyway.

In the new environment, and in the programming code showed in appendix B, packages of elements are referred to as systems instead of variables. These systems have many similarities to variables but are not completely equivalent.

In GAMS variables are defined as dependent on sets. The systems of the new environment are defined as dependent on unit vectors instead of sets. In section 9.3 figure 19 shows all systems of the new implementation and which unit vectors that defines them.

For a system to be explicitly defined an interval on each unit vector axes of the variable needs to be defined. These intervals are declared separately for each variable and can easily be redefined or changed to a list of indexes where the values along the unit vector axes even can appear in a random order if needed.

This saves some time as constraints defining certain unnecessary entries to zero are not needed as these elements can be omitted from the systems. During the declaration of a system sorting and if statements can also be used.

9.3 Identification of Red and Green Variables

Although the GET-RC 6.1 model could be defined as a point in the five dimensional unit vector space, it is not unambiguously defined until every variable in the model is defined within this space. When identifying red vs green variables the method is to see if any of the two variable's elements are in the same point in the unit vector room. If the elements are in the same point they can either both be red variables or one of them is derived from the other.

Thus another variable or to be more precise system is defined in the new environment if there are two variables with the same unit vector properties that can not be derived from each other. In the original GAMS implementation many elements needed in the model were defined in more than one variable. Therefore constraints, often containing equal signs, were needed to make sure these different variables were in balance. These extra variables was refereed to in the main parts of this report as green variables.

In the scheme code double declaring elements in two different variables requires precisely twice as much work as declaring them once. As the constraints balancing for example extracted coal and used coal would remove one of these variables and thereby render the extra work completely needless, all elements were only declared once. This is the reason for why every simple model reimplementation step in the result section of this report begin by removing the green variables.

When viewing the GET-model from the GAMS code it is not always clear what variables that are green but it becomes clearer when the background work of GAMS is viewed at the same time as the model code is written.

9.4 Unit vectors and Systems of GET-RC

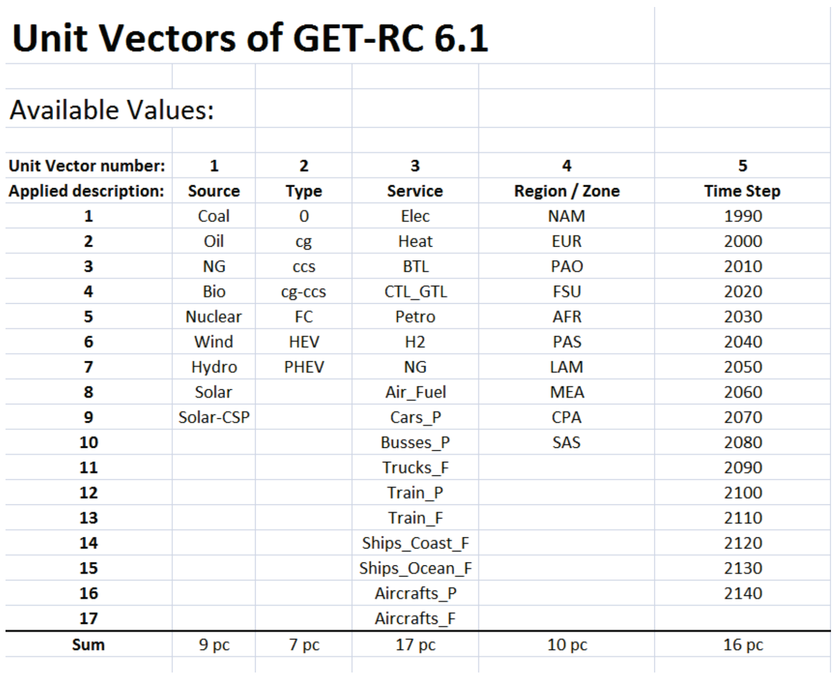
What names that are most descriptive of GET-RC unit vectors changes a little with the current perspective the programmer has of the model. Below are two examples on how to describe the unit vectors, one more general and one for a specific system of elements. These two examples are followed by figure 18 showing all the values along the unit vector axes.

Example 1 : Out of a more applied perspective of the GET-model the unit vectors can be described as follows:

1. The primary energy resources (Source). Containing coal, oil, ...
2. The combustion types (Type). Containing conventional combustion (0), carbon capture and storage (ccs), ...
3. Energy conversion or society services (Service). Electricity plant, BTL plant, Personal Car transports, ...
4. The regions of the model (Region). North America, Europe, ...
5. The time steps of the model (Time Step). 1990, 2000, ...

Example 2 : Out of a more philosophical point of view, describing what the unit vectors are governing for a specific system of elements, the same unit vectors need to be described a bit differently. These descriptions are dependent on what system the unit vectors are currently being used for. For system 1, described more later but comparable to the variable en_conv, the unit vectors can be described as follows:

1. The amount of primary energy resources the model plan to use.
2. How primary energy resources are to be combustion.
3. What the primary energy resources will be used for.
4. In what region the production will be.
5. At what time step the production will take place.



Unit Vectors of GET-RC 6.1

Available Values:

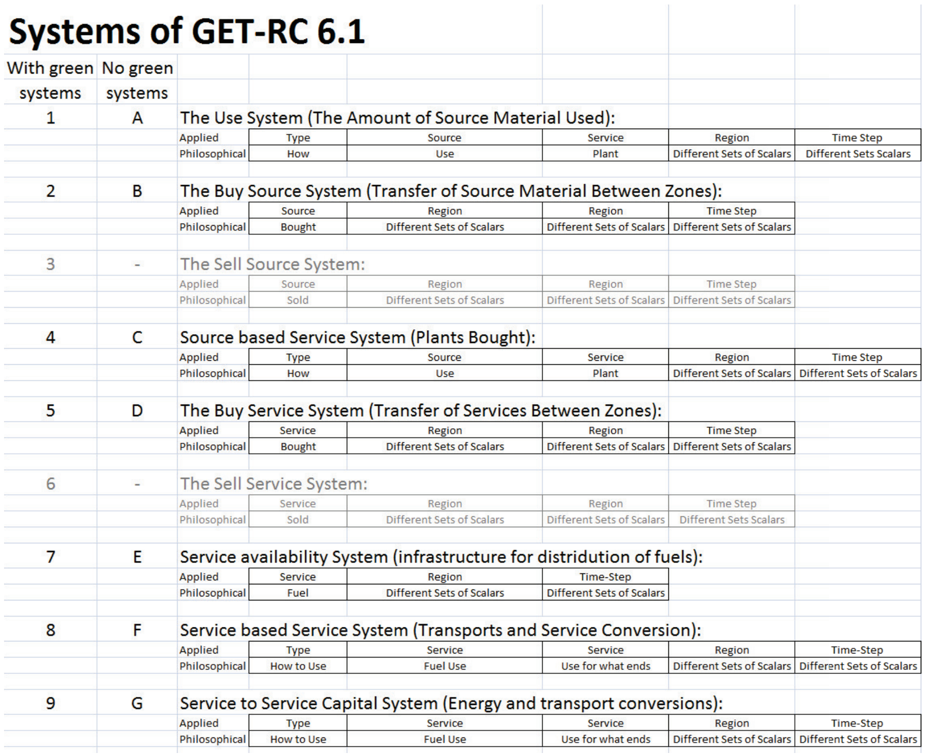
Unit Vector number:	**1**	**2**	**3**	**4**	**5**
Applied description:	**Source**	**Type**	**Service**	**Region / Zone**	**Time Step**
1	Coal	0	Elec	NAM	1990
2	Oil	cg	Heat	EUR	2000
3	NG	ccs	BTL	PAO	2010
4	Bio	cg-ccs	CTL_GTL	FSU	2020
5	Nuclear	FC	Petro	AFR	2030
6	Wind	HEV	H2	PAS	2040
7	Hydro	PHEV	NG	LAM	2050
8	Solar		Air_Fuel	MEA	2060
9	Solar-CSP		Cars_P	CPA	2070
10			Busses_P	SAS	2080
11			Trucks_F		2090
12			Train_P		2100
13			Train_F		2110
14			Ships_Coast_F		2120
15			Ships_Ocean_F		2130
16			Aircrafts_P		2140
17			Aircrafts_F		
Sum	9 pc	7 pc	17 pc	10 pc	16 pc

Figure 18: *The composition of the five unit vectors that are used to build the GET-RC model. See List of Acronyms or Appendix C for explanation to the acronyms.*

When it comes to the systems of GET-RC, as described before in the result section of the main report, the new optimization environment written during this project contains a function for declaring real ”green variables” or green systems. Instead of double declaring, the new environment can tie one element to many systems if needed. The number of systems in the new implementation can therefore be viewed a little differently depending on whether the green systems are counted as well or not.

Figure 19 shows all the Systems of the GET-RC model. In the first row describing system 1 it is shown that unit vector 2, Type, in this system builds on unit vector 1, Source. This means that all the available entries of the first unit vector must be looped through once to capture all possible ways of performing the first value of Type. This shows that the number of elements needed for each system grows very fast.

To see precisely what values each unit vector are aloud to take, in a given system, see figure 14, 15, 16 and 17 in the results section of the main report. As explained in section 9.2.2 not all available values for the a unit vector has to be allowed.



Systems of GET-RC 6.1

With green systems	No green systems						
1	A	The Use System (The Amount of Source Material Used):					
		Applied	Type	Source	Service	Region	Time Step
		Philosophical	How	Use	Plant	Different Sets of Scalars	Different Sets Scalars
2	B	The Buy Source System (Transfer of Source Material Between Zones):					
		Applied	Source	Region	Region	Time Step	
		Philosophical	Bought	Different Sets of Scalars	Different Sets of Scalars	Different Sets of Scalars	
3	-	The Sell Source System:					
		Applied	Source	Region	Region	Time Step	
		Philosophical	Sold	Different Sets of Scalars	Different Sets of Scalars	Different Sets of Scalars	
4	C	Source based Service System (Plants Bought):					
		Applied	Type	Source	Service	Region	Time Step
		Philosophical	How	Use	Plant	Different Sets of Scalars	Different Sets of Scalars
5	D	The Buy Service System (Transfer of Services Between Zones):					
		Applied	Service	Region	Region	Time Step	
		Philosophical	Bought	Different Sets of Scalars	Different Sets of Scalars	Different Sets of Scalars	
6	-	The Sell Service System:					
		Applied	Service	Region	Region	Time Step	
		Philosophical	Sold	Different Sets of Scalars	Different Sets of Scalars	Different Sets Scalars	
7	E	Service availability System (infrastructure for distridution of fuels):					
		Applied	Service	Region	Time-Step		
		Philosophical	Fuel	Different Sets of Scalars	Different Sets of Scalars		
8	F	Service based Service System (Transports and Service Conversion):					
		Applied	Type	Service	Service	Region	Time-Step
		Philosophical	How to Use	Fuel Use	Use for what ends	Different Sets of Scalars	Different Sets of Scalars
9	G	Service to Service Capital System (Energy and transport conversions):					
		Applied	Type	Service	Service	Region	Time-Step
		Philosophical	How to Use	Fuel Use	Use for what ends	Different Sets of Scalars	Different Sets of Scalars

Figure 19: *An overview of what unit vectors each system is dependent on. For a detailed description of the more explicit entries of each system see figure 14, 15, 16 and 17 in the results section of the main report.*

In the objective function all the systems of variables are added together multiplied with appropriate scalars. There are 9 Systems in total but the computer only need to declare 7 of these as the two selling systems are green systems. The systems that are declared are therefore numbered separately in figure 19 with the letters A to G. The tasks of the systems can be described as follows:

System A (compare to en_conv): System 1 is keeping track of how much Coal and the other sources that is being used for the different services. As no resources are allocated unless they are used this also represent how much the plant in the different zones are used.

System B and 3 (compare System B to imp_prim_from): System B keeps track of how much resources that the regions buy from each other. System 3 is a resorting of the elements of system B, sorted after how much each region is selling to another. The amount of locally extracted resources is the used (system 1) minus what was bought (System 2) plus what was sold (System 3).

System C (compare to cap_invest): System C makes sure that enough capital is invested in to fulfill the Service demand. In this demand the needs of System 7 is also included. This is the system for the source based services.

System D and 6 (compare System D to imp_sec_from): System D keeps track on the fuel market or the buying and selling (trading) of service products. Currently only Bio-To-Liquid (BTL), Coal-To-Liquid or Gas-To-Liquid (CTL-GTL), and hydrogen gas (H_2) can be traded. System 6 is just another sorting of System D (for more info see **System B and 3**).

System E (compare to infra_invest): The infrastructure of GET-RC requires a separate system because it is a kind of capital related to the use of liquid fuels but it is not compatible with the capital of transport engines. This is the smallest system of the new mathematical interpretation.

System F (compare to trsp_energy): System 7 keeps track of the service based services. Some services are not dependent on source materials directly but dependent on other services in between. This services are added to the model here. System C then takes these needs into account too when calculating the need of capital investments.

System G (compare to eng_invest): This system keeps track on the capital of transport engines. In a more philosophical perspective however it also keeps track on all capital needed for service to service conversions. So capital for energy conversions such as hydrogen to electricity is also watched over by this system.

10 Appendix B and C

Appendix B contains the model code of Simple 1, 2 and 3 and the final input data file with is common for all three implemented models. After this follows Appendix C which contains the set and variable descriptions of the original GET-RC Model implementation in GAMS.

Appendix B

Master thesis at Physical Resource Theory

ENMX60

Further program development for the cost minimizing global energy system model GET-RC

The model code of Simple Model 1

```
;Importing Grubi FFI mm
(load "gurobiffi.o17")
(include "verktyg.scm")

;Declaring matrixes of constants
(define t-step 10)

                      ;NAM  EUR
(define price  '#(#f64(1.0  1.0)   ;Coal
                  #f64(3.0  3.0)   ;Oil
                  #f64(2.0  2.0)   ;NG
                  #f64(4.0  4.0))) ;Bio


                           ;1990   2000   2010
(define supply-pot   '#(#f64(500.0  500.0  500.0)   ;Coal.NAM
                       #f64(500.0  500.0  500.0)   ;Coal.EUR
                       #f64(300.0  300.0  300.0)   ;Oil.NAM
                       #f64(300.0  300.0  300.0)   ;Oil.EUR
                       #f64(200.0  200.0  200.0)   ;NG.NAM
                       #f64(200.0  200.0  200.0)   ;NG.EUR
                       #f64(20.0   20.0   20.0)    ;Bio.NAM
                       #f64(20.0   20.0   20.0)))  ;Bio.


                        ;1990 2000 2010
(define demand       '#(#f64(100.0  100.0  100.0)   ;NAM.elec
                       #f64(200.0  200.0  200.0)   ;Nam.Heat
                       #f64(100.0  100.0  100.0)   ;EURO.elec
                       #f64(200.0  200.0  200.0))) ;EURO.heat
```

```
                           ;1990 2000 2010
(define effic       '#(#f64(0.5  0.5  0.5)   ;Coal.elec
                       #f64(0.9  0.9  0.9)   ;Coal.heat
                       #f64(0.5  0.5  0.5)   ;Oil.elec
                       #f64(0.9  0.9  0.9)   ;Oil.heat
                       #f64(0.6  0.6  0.6)   ;NG.elec
                       #f64(0.9  0.9  0.9)   ;NG.heat
                       #f64(0.5  0.5  0.5)   ;Bio.elec
                       #f64(0.9  0.9  0.9))) ;Bio.heat

(define nStep 3)
(define nZone 2)
(define nService 2)
(define nSource 4)
(define nVar (* nStep nZone nService nSource))

;Defining and Building the expresione to minimice obj
(define obj
  (list->f64vector
    (with-list-maker
      (for-interval cStep 1 nStep
        (for-interval cZone 1 nZone
          (for-interval cService 1 nService
            (for-interval cSource 1 nSource
              (cons! (* t-step (f64vector-ref (vector-ref price (- cSource 1)) (- cZone 1)))) )))))))


;Building Constraint Matrix
  ;Declaring Constants
    (define servicePeriod 4)
    (define zonPeriod 8)
    (define tStepPeriod 16)

  ;Building row of zeros
    (define ZerosList (list->f64vector (with-list-maker
      (for-interval coll 1 nVar
        (cons! 0.) ))))
```

```
;Adding up rows to form Constraint Matrix
  (define Constr (list->vector
    (with-list-maker
      ;Use Less then Supply
        ;Coal
          (for-interval row 1 6
            (let ((cRow (f64vector-copy ZerosList)))
              (f64vector-set! cRow (- (- (* 8 row) 1) 7) 1.)
              (f64vector-set! cRow (- (- (* 8 row) 1) 3) 1.)
              (cons! cRow) ))
        ;Oil
          (for-interval row 1 6
            (let ((cRow (f64vector-copy ZerosList)))
              (f64vector-set! cRow (- (- (* 8 row) 1) 6) 1.)
              (f64vector-set! cRow (- (- (* 8 row) 1) 2) 1.)
              (cons! cRow) ))
        ;NG
          (for-interval row 1 6
            (let ((cRow (f64vector-copy ZerosList)))
            (f64vector-set! cRow (- (- (* 8 row) 1) 5) 1.)
            (f64vector-set! cRow (- (- (* 8 row) 1) 1) 1.)
            (cons! cRow) ))
        ;Bio
          (for-interval row 1 6
            (let ((cRow (f64vector-copy ZerosList)))
              (f64vector-set! cRow (- (- (* 8 row) 1) 4) 1.)
              (f64vector-set! cRow (- (- (* 8 row) 1) 0) 1.)
              (cons! cRow) ))

      ;Produce more then Demanded
        (for-interval cStep 1 nStep
          (for-interval cZone 1 nZone
            (for-interval cService 1 nService
              (let ((cRow (f64vector-copy ZerosList))
                   ;Calculate ElementPositionConstant
                    (EPC (+ (* servicePeriod (- cService 1)) (* zonPeriod (- cZone 1)) (* tStepPeriod (- cStep 1)))))
                (f64vector-set! cRow (+ 0 EPC) (f64vector-ref (vector-ref effic (+ 0 (1- cService))) (1- cStep))) ;Coal
                (f64vector-set! cRow (+ 1 EPC) (f64vector-ref (vector-ref effic (+ 2 (1- cService))) (1- cStep))) ;Oil
                (f64vector-set! cRow (+ 2 EPC) (f64vector-ref (vector-ref effic (+ 4 (1- cService))) (1- cStep))) ;NG
                (f64vector-set! cRow (+ 3 EPC) (f64vector-ref (vector-ref effic (+ 6 (1- cService))) (1- cStep))) ;Bio
                (cons! cRow) )))))))
```

```
;Defining nConstr
(define nConstr (vector-length Constr))

;Defining procedure for building Constraint Vectors to Gurobi

  (define (constr->vbeg&vlen&vind&vval constr)
    (let ((n-constr (vector-length constr))(n-var (f64vector-length (vector-ref constr 0))))
        ;Declaring Constraint Vectors
          (with-list-maker
            (cons! (list->f64vector (with-list-maker5
              (cons! (list->s32vector (with-list-maker4
                (cons! (list->s32vector (with-list-maker3
                  (cons! (list->s32vector (with-list-maker2
                    (let ((i 0))
                      (for-interval coll 1 n-var
                        ;Defining sublists
                          (let ((clist (with-list-maker6
                            (cons!6 (with-list-maker7 (cons!6 (with-list-maker8
                              ;Looping through colum
                                (for-interval row 1 n-constr
                                  (let ((c-entry (f64vector-ref (vector-ref constr (1- row)) (1- coll))))
                                    (if (not (zero? c-entry)) (begin (cons!7 (1- row)) (cons!8 c-entry))) )))))))))

                            ;Building Constraint Vectors
                              (cons!2 (length (get-list5)))                               ;vbeg
                              (cons!3 (length (list-ref clist 0)))                        ;vlen
                              (for-each cons!4 (list-ref clist 1))                        ;vind
                              (for-each cons!5 (list-ref clist 0)) )))))))))))))))))) ;vval


;Building Constraint Vectors to Gurobi
(define vbeg&vlen&vind&vval (constr->vbeg&vlen&vind&vval Constr))


;Building string of more and less
(define moreOrLess (string-append (make-string 24 grb-less-equal)      ;Use Less then Supply
                                   (make-string 12 grb-greater-equal))) ;Produce more then Demanded
```

```
;Building Right Hand Side of Constraints
  (define zonPeriod 2)
  (define servicePeriod 2)

  (define rhs
    (list->f64vector
      (with-list-maker
        ;Use Less then Supply
          (for-interval cSource 1 nSource
            (for-interval cStep 1 nStep
              (for-interval cZone 1 nZone
                (cons! (f64vector-ref (vector-ref supply-pot (+ (* (- cSource 1) zonPeriod) (- cZone 1))) (- cStep 1))) )))

        ;Produce more then Demanded
          (for-interval cStep 1 nStep
            (for-interval cZone 1 nZone
              (for-interval cService 1 nService
                (cons! (f64vector-ref (vector-ref demand (+ (- cService 1) (* (- cZone 1) servicePeriod) )) (- cStep 1))) ))))))


;Building Variable Type Vector (vtype)
  (define vtype (make-string nVar grb-continuous))


;Calling on Gurobi
  (define model
    (let-args
     vbeg&vlen&vind&vval (vbeg vlen vind vval)
     (grb-load-model grb-env "example" nVar nConstr 1 0. obj moreOrLess
                     rhs vbeg vlen vind vval
                     #f #f #f #f vtype
                     #f '() ; = inga varnames
                     #f '() ; = inga constrnames
                     )))


;Tell Gurobi to optimice
  (grb-optimize grb-env model)

;Retreave results from Gurobi

  ;Retreav the minimum cost found
```

```
    (define Optimum (grb-get-dbl-attr* grb-env model "ObjVal"))

  ;Retreav the minimum point
    (define Point (list->f64vector (with-list-maker
      (for-interval cVar 1 nVar
        (cons! (grb-get-dbl-attr-element* grb-env model "X" (1- cVar))) ))))


;Process results
(define Time-Result-Matrix (list->vector (with-list-maker
  (for-interval cStep 1 nStep
    (cons! (list->f64vector (with-list-maker2
      (for-interval cVar 1 tStepPeriod
        (cons!2 (f64vector-ref Point (+ (1- cVar) (* tStepPeriod (1- cStep))))) ))))))))

(define Var-Time-Matrix (list->vector (with-list-maker
  (for-interval cVar 1 tStepPeriod
    (cons! (list->f64vector (with-list-maker2
      (for-interval cStep 1 nStep
        (cons!2 (f64vector-ref (vector-ref Time-Result-Matrix (1- cStep)) (1- cVar))) ))))))))

;Print out
Optimum
Var-Time-Matrix
```

The model code of Simple Model 2

```
;Importing Grubi FFI mm
(load "gurobiffi.o18")
(include "verktyg.scm")
(include "Data SimpleFive.scm")

;Declaring Constants
(define startStep 1)
(define nStep 3)
(define nZone 2)
(define nTrancferZone (1- nZone))
(define nService 2)
(define nSource 4)
(define nVar (* nStep nZone nService nSource))
(define businessNVar (* nStep nZone nTrancferZone nSource))
(define totNVar (+ nVar businessNVar))

;Defining and Building the expresione to minimice obj
(define obj
  (list->f64vector
    (with-list-maker
      ;Primery model (System 1)
        (for-interval cStep 1 nStep
          (for-interval cZone 1 nZone
            (for-interval cService 1 nService
              (for-interval cSource 1 nSource
                (cons! (* t-step (price-get cZone cSource))) ))))

      ;Buying (System 2)
        (for-interval cStep 1 nStep
          (for-interval cZone 1 nZone
              (for-interval cTrancferZone 1 nZone
                (if (not (zero? (- cTrancferZone cZone)))
                  (for-interval cSource 1 nSource
                    ;Fix transport cost + lin transport cost + extraction cost in export region cTrancferZone - extra
                    extraction cost from usevariables above
                      (let* ((fixCost (imp-cost-get cSource))
                             (linCost (imp-cost-lin-get cSource))
                             (dist (distance-get cZone cTrancferZone))
                             (aboveextractCost (price-get cZone cSource))
```

```
                              (cCost (+ fixCost (* linCost dist) (price-get cTrancferZone cSource) (- aboveextractCost))) )
                         (cons! (* t-step cCost)) )))))))))


;Building Constraints
  ;Declaring Constants
    (define servicePeriod nSource) ;4
    (define zonPeriod (* nSource nService)) ;8
    (define zonPeriodBusiness (* nSource nTrancferZone)) ;4
    (define trancferZonePeriod nSource) ;4
    (define tStepPeriod (* zonPeriod nZone)) ;16
    (define tStepPeriodBusiness (* nZone zonPeriodBusiness)) ;16

  ;Building row of zeros
    (define ZerosList (list->f64vector (with-list-maker
      (for-interval coll 1 totNVar
        (cons! 0.) ))))

  ;(EPF-set! cSystem cValue
    (define-macro (EPF-set! var-System . code)
     `(begin (if (= ,var-System 1)
        (let ((EPC (+ (* tStepPeriod (1- cStep)) (* zonPeriod (1- cZone)) (* servicePeriod (1- cService)) (1- cSource))))
                        (f64vector-set! cRow EPC ,@code) ))

      (if (= ,var-System 2)
        (let ((EPC (+ nVar (* tStepPeriodBusiness (1- cStep)) (* zonPeriodBusiness (1- cZone)) (* trancferZonePeriod (1-
cTrancferZone)) (1- cSource))))
                        (f64vector-set! cRow EPC ,@code) ))

      (if (= ,var-System 3)
        (let ((EPC (+ nVar (* tStepPeriodBusiness (1- cStep)) (* zonPeriodBusiness (1- cTrancferZone)) (*
trancferZonePeriod (1- cOurTrancferZoneIndex)) (1- cSource))) )
                                (f64vector-set! cRow EPC ,@code) ))))
```

```
;Adding up rows to form Constraint Matrix
   (define Constr (list->vector
     (with-list-maker
       ;Services - Buy + Sell less then Supply-pot
         (for-interval cSource 1 nSource
           (for-interval cStep 1 nStep
             (for-interval cZone 1 nZone
               (let ((cRow (f64vector-copy ZerosList)))
                 ;Services
                   (for-interval cService 1 nService
                     (EPF-set! 1 1.) )
                 ;Buy
                   (for-interval cTrancferZone 1 nTrancferZone
                     (EPF-set! 2 (- 1.)) )
                 ;Sell
                   (for-interval cTrancferZone 1 nZone
                     (if (not (zero? (- cTrancferZone cZone)))
                       ;cZone < cTrancferZone
                         (begin (if (< cZone cTrancferZone)
                           (let ((cOurTrancferZoneIndex cZone))
                             (EPF-set! 3 1.) ))
                       ;cZone > cTrancferZone
                          (if (> cZone cTrancferZone)
                            (let ((cOurTrancferZoneIndex (1- cZone)))
                              (EPF-set! 3 1.) )) )))
                 (cons! cRow) ) )))


       ;Produce more then Demanded
         (for-interval cStep 1 nStep
           (for-interval cZone 1 nZone
             (for-interval cService 1 nService
               (let ((cRow (f64vector-copy ZerosList)))
                 (for-interval cSource 1 nSource
                   (EPF-set! 1 (effic-get cStep cService cSource)) )
                 (cons! cRow) )))) )))


 ;Defining nConstr
   (define nConstr (vector-length Constr))
   (define nConstrLessThenSupply (* nSource nStep nZone))
   (define nConstrDemanded (* nService nZone nStep))
```

```
;Building string of more and less
  (define moreOrLess (string-append (make-string nConstrLessThenSupply grb-less-equal) ;Use Less then Supply
                                (make-string nConstrDemanded grb-greater-equal) )) ;Produce more then Demanded


;Building Right Hand Side of Constraints
  (define rhs
    (list->f64vector
      (with-list-maker
        ;Use Less then Supply
          (for-interval cSource 1 nSource
            (for-interval cStep 1 nStep
              (for-interval cZone 1 nZone
                (cons! (supply-pot-get cStep cZone cSource)) )))

        ;Produce more then Demanded
          (for-interval cStep 1 nStep
            (for-interval cZone 1 nZone
              (for-interval cService 1 nService
                (cons! (demand-get cStep cZone cService)) ))) )))


;Building Variable Type Vector (vtype)
  (define vtype (make-string totNVar grb-continuous))
```

```
;Defining procedure for building Constraint Vectors to Gurobi (Sparce Matrix Format)
  (define (constr->vbeg&vlen&vind&vval constr)
    (let ((n-constr (vector-length constr))(n-var (f64vector-length (vector-ref constr 0))))
        ;Declaring Constraint Vectors
          (with-list-maker
            (cons! (list->f64vector (with-list-maker5
              (cons! (list->s32vector (with-list-maker4
                (cons! (list->s32vector (with-list-maker3
                  (cons! (list->s32vector (with-list-maker2
                    (let ((i 0))
                      (for-interval coll 1 n-var
                        ;Defining sublists
                          (let ((clist (with-list-maker6
                            (cons!6 (with-list-maker7 (cons!6 (with-list-maker8
                              ;Looping through colum
                                (for-interval row 1 n-constr
                                  (let ((c-entry (f64vector-ref (vector-ref constr (1- row)) (1- coll))))
                                    (if (not (zero? c-entry)) (begin (cons!7 (1- row)) (cons!8 c-entry))) )))))))))

                            ;Building Constraint Vectors
                              (cons!2 (length (get-list5)))                           ;vbeg
                              (cons!3 (length (list-ref clist 0)))                    ;vlen
                              (for-each cons!4 (list-ref clist 1))                    ;vind
                              (for-each cons!5 (list-ref clist 0)) )))))))))))))))))) ;vval


;Building Constraint Vectors to Gurobi
  (define vbeg&vlen&vind&vval (constr->vbeg&vlen&vind&vval Constr))


;Calling on Gurobi
  (define model
    (let-args
     vbeg&vlen&vind&vval (vbeg vlen vind vval)
     (grb-load-model grb-env "example" totNVar nConstr 1 0. obj moreOrLess
                     rhs vbeg vlen vind vval
                     #f #f #f #f vtype
                     #f '() ; = inga varnames
                     #f '() ; = inga constrnames
                     )))
```

```
;Tell Gurobi to optimice
  (grb-optimize grb-env model)


;Retreave results from Gurobi
  ;Retreav the minimum cost found
    (define Optimum (grb-get-dbl-attr* grb-env model "ObjVal"))

  ;Retreav the minimum point
    (define Point (list->f64vector (with-list-maker
      (for-interval cVar 1 totNVar
        (cons! (grb-get-dbl-attr-element* grb-env model "X" (1- cVar))) ))))


;Process results
  (define Time-Result-Matrix (list->vector (with-list-maker
    (for-interval cStep 1 nStep
      (cons! (list->f64vector (with-list-maker2
        (for-interval cVar 1 tStepPeriod
          (cons!2 (f64vector-ref Point (+ (1- cVar) (* tStepPeriod (1- cStep))))) ))))))))

  (define Var-Time-Matrix (list->vector (with-list-maker
    (for-interval cVar 1 tStepPeriod
      (cons! (list->f64vector (with-list-maker2
        (for-interval cStep 1 nStep
          (cons!2 (f64vector-ref (vector-ref Time-Result-Matrix (1- cStep)) (1- cVar))) ))))))))



;Print out
  Optimum
  Var-Time-Matrix
  Point
```

The model code of Simple Model 3

```
;Importing Grubi FFI mm
(load "gurobiffi.o19")
(include "verktyg.scm")
(include "Data SimpleFive.scm")


;Defining Model Dimentiones
(define startStep 1)
(define nStep 16)
(define nZone 10)
(define nTrancferZone (1- nZone))
(define nService 2)
(define nServicePeriod 2)
(define nSource 9)
(define nSourceTrade 5)
(define nType 1)
(define nTypePeriod 4)
(define nVar (* nStep nZone 14))
(define businessNVar (* nStep nZone nTrancferZone nSourceTrade))
(define convNVar (* nStep nZone 14))
(define totNVar (+ nVar businessNVar convNVar)) ;11680
;(define totNVar (+ nVar businessNVar)) ;11680
```

```
;Defining and Building the expresione to minimice obj
(define obj
 (list->f64vector
  (with-list-maker
   ;Consumption Costs (Part of System 1)
    (for-interval cStep 1 nStep
     (for-interval cZone 1 nZone
      (for-interval cService 1 nService
       (for-interval cSource 1 nSource
        (for-interval cType 1 nType
         (if (< 0 (effic-get cStep cService cSource cType))
          (let ((cOpManfr (OM-cost-fr-get cService cSource cType))
                (cInvCost (cost-inv-mod-get cService cSource cType))
                (cExtrPrice (price-get cZone cSource))
                (cConvEffic (effic-get cStep cService cSource cType)))
          ;(cons! (/ (* cOpManfr cInvCost) (* Msec-per-year 0.7))) )))))))
          (cons! (* t-step (+ cExtrPrice (/ (* cOpManfr cInvCost cConvEffic) (* Msec-per-year 0.7))))) )))))))

   ;Resource Transfer (Part of System 2 and impicitly also System 3)
    ;To the coeficientes below a value called aboveextractCost is removed. This is because the extractione cost of
    ;the bought amount of a recource is payed for with the wrong price above. Here as transfer between zons are
    ;posible this payment is removed and replased with the prise from the right extraction none
    ;(see: (price-get cTrancferZone cSource)).
    (for-interval cStep 1 nStep
     (for-interval cZone 1 nZone
       (for-interval cTrancferZone 1 nZone
        (if (not (zero? (- cTrancferZone cZone)))
         (for-interval cSource 1 nSourceTrade
          (let* ((fixCost (imp-cost-get cSource))
                 (linCost (imp-cost-lin-get cSource))
                 (dist (distance-get cZone cTrancferZone))
                 (aboveextractCost (price-get cZone cSource))
                                                ;Adding right price                Removing wrong price from System 1
                (cCost (+ fixCost (* linCost dist) (price-get cTrancferZone cSource) (- aboveextractCost))) )
           (cons! (* t-step cCost)) ))))))
```

```
;Investment Costs (Part of System 4)
        (for-interval cStep 1 nStep
          (for-interval cZone 1 nZone
            (for-interval cService 1 nService
              (for-interval cSource 1 nSource
                (for-interval cType 1 nType
                  (if (< 0 (effic-get cStep cService cSource cType))
                    (let ((cInvCost (cost-inv-mod-get cService cSource cType)))
                      (cons! (* t-step cInvCost))) ))))))
                         )))



;Building Constraints
  ;Declaring Constants
                              ;Coal Oil NG Bio Nuclear Wind Hydro Solar Solar-csp
    (define typePeriod '#(#s32( 1    1  1   1     1     1    1     1     1)    ;Elec
                          #s32( 1    1  1   1     0     0    0     1     0 ) )) ;Heat

                               ;Elec ;Heat
    (define serviceRange '#s32( 0     9))

                               ;Elec ;Heat
    (define servicePeriod '#s32( 9     5))

    (define zonPeriod (+ (s32vector-ref servicePeriod 0) (s32vector-ref servicePeriod 1))) ;14
    (define zonPeriodBusiness (* nSourceTrade nTrancferZone)) ;45
    (define trancferZonePeriod nSourceTrade) ;5
    (define tStepPeriod (* zonPeriod nZone)) ;140
    (define tStepPeriodBusiness (* nZone zonPeriodBusiness)) ;450

  ;Building row of zeros
    (define ZerosList (list->f64vector (with-list-maker
      (for-interval coll 1 totNVar
        (cons! 0.) ))))
```

```
;Seystem declaration
    ;(EPF-set! cSystem cValue)
      (define-macro (EPF-set! var-System . code)
        ;System 1
        `(if (= ,var-System 1)
          (let* ((cServiceRange (s32vector-ref serviceRange (1- cService)))
                (EPC (+ (* tStepPeriod (1- cStep)) (* zonPeriod (1- cZone)) cServiceRange (1- cSource) (1- cType)))
                (EPC2 (+ (* tStepPeriod (1- cStep)) (* zonPeriod (1- cZone)) cServiceRange (1- 5) (1- cType))))
            (if (and (= cService 2) (= cSource 8))
                           ;True                             ;False
              (f64vector-set! cRow EPC2 ,@code) (f64vector-set! cRow EPC ,@code)))

          ;System 2
          (if (= ,var-System 2)
            (let ((EPC (+ nVar (* tStepPeriodBusiness (1- cStep)) (* zonPeriodBusiness (1- cZone)) (* trancferZonePeriod
            (1- cTrancferZone)) (1- cSource))))
              (f64vector-set! cRow EPC ,@code) )

            ;System 3
            (if (= ,var-System 3)
              (let ((EPC (+ nVar (* tStepPeriodBusiness (1- cStep)) (* zonPeriodBusiness (1- cTrancferZone))
              (* trancferZonePeriod (1- cOurTrancferZoneIndex)) (1- cSource))))
                (f64vector-set! cRow EPC ,@code) )

              ;System 4
              (if (= ,var-System 4)
                (let* ((cServiceRange (s32vector-ref serviceRange (1- cService)))
                       (EPC (+ nVar businessNVar (* tStepPeriod (1- cStep2)) (* zonPeriod (1- cZone)) cServiceRange (1- cSource) (1- cType)))
                       (EPC2 (+ nVar businessNVar (* tStepPeriod (1- cStep2)) (* zonPeriod (1- cZone)) cServiceRange (1- 5) (1- cType))))
                  (if (and (= cService 2) (= cSource 8))
                           ;True                             ;False
              (f64vector-set! cRow EPC2 ,@code) (f64vector-set! cRow EPC ,@code)) ))))))
```

```
;Adding up rows to form Constraint Matrix
    (define ConstrTot (with-list-maker3
      (cons!3 (list->f64vector (with-list-maker2
        (cons!3 (list->vector (with-list-maker

          ;Services - Buy + Sell less then Supply-pot
            (for-interval cSource 1 nSource
              ;Fossil
                (if (< cSource 4)
                  (for-interval cZone 1 nZone
                    (for-interval cStep2 1 nStep
                      (let ((cRow (f64vector-copy ZerosList)))
                        (for-interval cStep 1 cStep2
                          ;Services
                            (for-interval cService 1 nService
                              (for-interval cType 1 nType
                                (if (< 0 (effic-get cStep cService cSource cType))
                                  (EPF-set! 1 (* 1. t-step))
                                    )))
                          ;Buy
                            (for-interval cTrancferZone 1 nTrancferZone
                              (EPF-set! 2 (* (- 1.) t-step))
                                )
                          ;Sell
                            (for-interval cTrancferZone 1 nZone
                              (if (not (zero? (- cTrancferZone cZone)))
                                ;cZone < cTrancferZone
                                  (if (< cZone cTrancferZone)
                                    (let ((cOurTrancferZoneIndex cZone))
                                      (EPF-set! 3 (* 1. t-step))
                                        )
                                ;cZone > cTrancferZone
                                    (if (> cZone cTrancferZone)
                                      (let ((cOurTrancferZoneIndex (1- cZone)))
                                        (EPF-set! 3 (* 1. t-step))
                                          )) ))))
                        (cons! cRow)
                        (cons!2 (supply-pot-get cZone cSource)) ))))
    ;Renewable
      (if (> cSource 3)
        (for-interval cStep 1 nStep
          (for-interval cZone 1 nZone
            (let ((cRow (f64vector-copy ZerosList)))
              ;Services
                (for-interval cService 1 nService
                  (for-interval cType 1 nType
                    (if (< 0 (effic-get cStep cService cSource cType))
                      (EPF-set! 1 1.)
                        )))
                (if (< cSource 6)
                  (begin
                    ;Buy
                      (for-interval cTrancferZone 1 nTrancferZone
                        (EPF-set! 2 (- 1.))
                          )
                    ;Sell
                      (for-interval cTrancferZone 1 nZone
                        (if (not (zero? (- cTrancferZone cZone)))
                          ;cZone < cTrancferZone
                            (if (< cZone cTrancferZone)
                              (let ((cOurTrancferZoneIndex cZone))
                                (EPF-set! 3 1.)
                                  )
                              )
                          ;cZone > cTrancferZone
                              (if (> cZone cTrancferZone)
                                (let ((cOurTrancferZoneIndex (1- cZone)))
                                  (EPF-set! 3 1.)
                                    )) )))))

              (cons! cRow)
              (cons!2 (supply-pot-get cZone cSource)) )))))


;Produce more then Demanded
  (for-interval cStep 1 nStep
    (for-interval cZone 1 nZone
      (for-interval cService 1 nService
        (let ((cRow (f64vector-copy ZerosList)))
          (for-interval cSource 1 nSource
            (for-interval cType 1 nType
                      (if (< 0 (effic-get cStep cService cSource cType))
                        (EPF-set! 1 (effic-get cStep cService cSource cType)) )))
                  (cons! cRow) )
                  (cons!2 (demand-get cStep cZone cService)) )))


      ;Convertion capacity grater then Use for each type
        (for-interval cStep 1 nStep
          (for-interval cZone 1 nZone
            (for-interval cService 1 nService
              (for-interval cSource 1 nSource
                (for-interval cType 1 nType
                  (if (< 0 (effic-get cStep cService cSource cType))
                    (let* ((cRow (f64vector-copy ZerosList))
                          (cConvEffic (effic-get cStep cService cSource cType)) ;This is wrong but just as wrong as in
                          (cLf (lf-get cZone cService cSource cType))               GAMS. Efficiency is instantly
                          (cInitCap (init-Cap-get cZone cService cSource cType))    updated for all powerplants without
                          (cPlantLifeLength (life-plant-get cService cSource cType)) costt in every iteration.
                          (depreciation (exp (* 10 (log (- 1 (/ 1 cPlantLifeLength)))))) )
                      (EPF-set! 1 cConvEffic)
                      ;Adding upp preveus years capacity
                        (for-interval cStep2 1 cStep
                          (EPF-set! 4 (* (- cLf) Msec-per-year t-step (expt depreciation (- cStep cStep2)))) )
                      (cons! cRow)
                      (cons!2 (* cLf Msec-per-year cInitCap (expt depreciation (1- cStep))))
                      )))))))
                      ))))))))


;Separating constraint Matrix
  (define Constr (list-ref ConstrTot 0))
  (define rhs (list-ref ConstrTot 1))

;Defining nConstr
  (define nConstr (vector-length Constr))
  ;Building string of more and less
    ;Defining Constants
      (define nConstrLessThenSupply
        (let ((nFossilS 3)
              (nRenewS 6))
                  ;Fossil              ;Renewable
          (+ (* nFossilS nStep nZone) (* nRenewS nStep nZone)) ))
      (define nConstrDemanded (* nService nStep nZone))
      (define nConstrConv (- nConstr nConstrDemanded nConstrLessThenSupply))

    ;Building string
      (define moreOrLess (string-append (make-string nConstrLessThenSupply grb-less-equal) ;Use Less then Supply
                                        (make-string nConstrDemanded grb-greater-equal)    ;Produce more then Demanded
                                        (make-string nConstrConv grb-less-equal)           ;Convertion capacity grater
then Use of each type
                                        ))


  ;Building Variable Type Vector (vtype)
    (define vtype (make-string totNVar grb-continuous))


(load "toSparceMatrix.o2")


;Building Constraint Vectors to Gurobi
  (define vbeg&vlen&vind&vval (constr->vbeg&vlen&vind&vval Constr))


;Seting number of threds and calculation method
  (grb-set-number-of-threds grb-env 4)
  (grb-set-method grb-env 2)

;nThreads cMethod
```

```
;Calling on Gurobi
  (define model
    (let-args
     vbeg&vlen&vind&vval (vbeg vlen vind vval)
     (grb-load-model grb-env "example" totNVar nConstr 1 0. obj moreOrLess
                     rhs vbeg vlen vind vval
                     #f #f #f #f vtype
                     #f '() ; = inga varnames
                     #f '() ; = inga constrnames
                     )))


;Tell Gurobi to optimice
  (grb-optimize grb-env model)


;Retreave results from Gurobi
  ;Retreav the minimum cost found
    (define Optimum (grb-get-dbl-attr* grb-env model "ObjVal"))

  ;Retreav the minimum point
    (define Point (list->f64vector (with-list-maker
      (for-interval cVar 1 totNVar
        (cons! (grb-get-dbl-attr-element* grb-env model "X" (1- cVar))) ))))
```

```
;Process results
  ;System 1
    (define Time-Result-Matrix (list->vector (with-list-maker
      (for-interval cStep 1 nStep
        (cons! (list->f64vector (with-list-maker2
          (for-interval cVar 1 tStepPeriod
            (cons!2 (f64vector-ref Point (+ (1- cVar) (* tStepPeriod (1- cStep))))) ))))))))

    (define System1 (list->vector (with-list-maker
      (for-interval cVar 1 tStepPeriod
        (cons! (list->f64vector (with-list-maker2
          (for-interval cStep 1 nStep
            (cons!2 (f64vector-ref (vector-ref Time-Result-Matrix (1- cStep)) (1- cVar))) ))))))))

   ;System 2
    (define Time-Result-Matrix2 (list->vector (with-list-maker
      (for-interval cStep 1 nStep
        (cons! (list->f64vector (with-list-maker2
          (for-interval cVar 1 tStepPeriodBusiness
            (cons!2 (f64vector-ref Point (+ (1- cVar) (* tStepPeriodBusiness (1- cStep)) (* tStepPeriod nStep)))) ))))))))

    (define System2 (list->vector (with-list-maker
      (for-interval cVar 1 tStepPeriodBusiness
        (cons! (list->f64vector (with-list-maker2
          (for-interval cStep 1 nStep
            (cons!2 (f64vector-ref (vector-ref Time-Result-Matrix2 (1- cStep)) (1- cVar))) ))))))))

   ;System 4
    (define Time-Result-Matrix4 (list->vector (with-list-maker
      (for-interval cStep 1 nStep
        (cons! (list->f64vector (with-list-maker2
          (for-interval cVar 1 tStepPeriod
            (cons!2 (f64vector-ref Point (+ (1- cVar) (* tStepPeriod (1- cStep)) (* tStepPeriod nStep) (*
tStepPeriodBusiness nStep)))) ))))))))

    (define System4 (list->vector (with-list-maker
      (for-interval cVar 1 tStepPeriod
        (cons! (list->f64vector (with-list-maker2
          (for-interval cStep 1 nStep
            (cons!2 (f64vector-ref (vector-ref Time-Result-Matrix4 (1- cStep)) (1- cVar))) ))))))))
```

```
;Out
 Optimum
  ;System1
```

The Input Data of the Simple Models (called "Data SimpleFive.scm")

```
;Declaring Constants
;################################################################################################################################################################################
;PRIMARY MODELL (USE) System 1
;################################################################################################################################################################################

(define t-step 10)
(define Msec-per-year 31.6)

;(t-step)

;################################################################################################################################################################################

                      ;NAM    EUR   PAO   FSU   AFR   PAS   LAM   MEA   CPA   SAS
(define price  '#(#f64(1.0    1.0   1.0   1.0   1.0   1.0   1.0   1.0   1.0   1.0)  ;Coal
                  #f64(3.0    3.0   3.0   3.0   3.0   3.0   3.0   3.0   3.0   3.0)  ;Oil
                  #f64(2.5    2.5   2.5   2.5   2.5   2.5   2.5   2.5   2.5   2.5)  ;NG
                  #f64(3.0    4.0   2.0   4.0   2.0   2.0   2.0   2.0   2.0   2.0)  ;Bio
                  #f64(1.0    1.0   1.0   1.0   1.0   1.0   1.0   1.0   1.0   1.0)  ;Nuclear
                  #f64(0.     0.    0.    0.    0.    0.    0.    0.    0.    0.)   ;Wind
                  #f64(0.     0.    0.    0.    0.    0.    0.    0.    0.    0.)   ;Hydro
                  #f64(0.     0.    0.    0.    0.    0.    0.    0.    0.    0.)   ;Solar
                  #f64(0.     0.    0.    0.    0.    0.    0.    0.    0.    0.))) ;Solar_csp


;     1.0      1.0      1.0      1.0      1.0      1.0      1.0      1.0      1.0      1.0
;100000.0 100000.0 100000.0 100000.0 100000.0 100000.0 100000.0 100000.0 100000.0 100000.0

                      ;NAM      EUR      PAO      FSU      AFR      PAS      LAM      MEA      CPA      SAS
;(define price  '#(#f64(1.0      1.0      1.0      1.0      1.0      1.0      1.0      1.0      1.0      1.0)  ;Coal
 ;                  #f64(100000.0 100000.0 100000.0 100000.0 100000.0 100000.0 100000.0 100000.0 100000.0 100000.0)  ;Oil
  ;                 #f64(100000.0 100000.0 100000.0 100000.0 100000.0 100000.0 100000.0 100000.0 100000.0 100000.0)  ;NG
   ;                #f64(100000.0 100000.0 100000.0 100000.0 100000.0 100000.0 100000.0 100000.0 100000.0 100000.0)  ;Bio
    ;               #f64(100000.0 100000.0 100000.0 100000.0 100000.0 100000.0 100000.0 100000.0 100000.0 100000.0)  ;Nuclear
     ;              #f64(100000.0 100000.0 100000.0 100000.0 100000.0 100000.0 100000.0 100000.0 100000.0 100000.0)   ;Wind
      ;             #f64(100000.0 100000.0 100000.0 100000.0 100000.0 100000.0 100000.0 100000.0 100000.0 100000.0)   ;Hydro
       ;            #f64(100000.0 100000.0 100000.0 100000.0 100000.0 100000.0 100000.0 100000.0 100000.0 100000.0)   ;Solar
        ;           #f64(100000.0 100000.0 100000.0 100000.0 100000.0 100000.0 100000.0 100000.0 100000.0 100000.0))) ;Solar_csp


;(price-get cZone cSource)
(define-macro (price-get var-Zone var-Source)
  `(f64vector-ref (vector-ref price (1- ,var-Source)) (1- ,var-Zone)) )


;################################################################################################################################################################################

                           ;NAM     EUR     PAO     FSU      AFR     PAS     LAM     MEA     CPA     SAS
(define supply-pot  '#(#f64(37262.  17640.  19074.  121747.  6457.   380.    1983.   633.    54649.  3756.)    ;Coal
                       #f64(1550.   864.    65.     1115.    830.    96.     1144.   5362.   564.    254.)     ;Oil
                       #f64(1959.   1420.   155.    2348.    706.    165.    665.    2578.   315.    434.)     ;NG
                       #f64(18.     20.     2.      13.      44.     6.5     65.     0.25    20.     17.)      ;Bio
                       #f64(10000.  10000.  10000.  10000.   10000.  10000.  10000.  10000.  10000.  10000.)    ;Nuclear
                       #f64(1000.   1000.   1000.   1000.    1000.   1000.   1000.   1000.   1000.   1000.)    ;Wind
                       #f64(2.57    2.7     0.5     2.04     1.88    0.50    2.53    0.05    3.28    1.30)    ;Hydro
                       #f64(1000.   1000.   1000.   1000.    1000.   1000.   1000.   1000.   1000.   1000.)    ;Solar
                      #f64(1000.   1000.   1000.   0.       1000.   1000.   1000.   1000.   1000.   1000.) )) ;Solar-CSP

;(supply-pot-get cZone cSource)
(define-macro (supply-pot-get var-Zone var-Source)
  `(f64vector-ref (vector-ref supply-pot (1- ,var-Source)) (1- ,var-Zone)) )


;#################################################################################################################################################################################

                      ;1990    2000    2010    2020    2030    2040    2050    2060    2070    2080    2090    2100    2110    2120    2130    2140
(define demand  '#(#f64(12.67   16.32   16.93   17.29   17.48   17.15   16.84   15.61   14.41   13.53   12.66   11.78   11.19   10.63   10.10   10.10)    ;NAM.elec
                  #f64(29.61   33.29   31.69   28.70   23.81   20.11   16.42   13.38   10.35   8.86    7.36    5.86    5.75    5.63    5.52    5.52)     ;NAM.Heat
                  #f64(8.95    10.61   12.38   13.63   15.02   16.03   17.03   17.06   17.09   16.63   16.18   15.72   15.72   15.72   15.72   15.72)    ;EUR.elec
                  #f64(30.48   29.49   28.68   26.48   24.41   22.06   19.71   17.17   14.62   13.44   12.26   11.09   10.87   10.65   10.44   10.44)    ;EUR.heat
                  #f64(3.75    5.20    5.19    5.28    5.58    5.40    5.23    4.97    4.71    4.65    4.58    4.52    4.52    4.52    4.52    4.52)     ;PAO.elec
                  #f64(9.17    12.04   10.82   9.16    7.34    5.95    4.55    3.94    3.32    3.10    2.88    2.66    2.66    2.66    2.66    2.66)     ;PAO.heat
                  #f64(4.83    3.54    4.94    5.00    5.22    5.52    5.84    6.21    6.58    6.99    7.40    7.81    7.81    7.81    7.81    7.81)     ;FSU.elec
                  #f64(28.23   18.07   27.34   28.36   27.61   26.15   24.69   22.20   19.71   17.55   15.39   13.24   12.97   12.71   12.46   12.46)    ;FSU.heat
                  #f64(0.95    1.29    1.58    2.02    2.44    3.67    4.89    6.62    8.34    13.07   17.81   22.55   24.81   27.29   30.01   30.01)    ;AFR.elec
                  #f64(3.15    12.39   15.83   19.13   22.82   25.81   28.80   32.19   35.59   40.78   45.97   51.16   53.72   56.41   59.23   59.23)    ;AFR.heat
                  #f64(0.84    1.54    2.65    4.12    5.48    7.07    8.67    9.92    11.18   11.97   12.78   13.58   13.58   13.58   13.58   13.58)    ;PAS.elec
                  #f64(4.21    9.89    15.13   16.09   16.83   16.56   16.29   14.80   13.31   12.77   12.23   11.69   11.69   11.69   11.69   11.69)    ;PAS.heat
                  #f64(1.60    2.43    3.37    3.98    4.66    5.32    5.98    7.41    8.82    9.71    10.59   11.49   11.49   11.49   11.49   11.49)    ;LAM.elec
                  #f64(6.32    7.96    15.28   17.32   18.15   18.35   18.55   17.14   15.73   15.19   14.64   14.10   14.10   14.10   14.10   14.10)    ;LAM.heat
                  #f64(0.71    1.38    1.85    2.55    3.43    4.78    6.11    8.43    10.73   14.72   18.71   22.70   23.84   25.03   26.28   26.28)    ;MEA.elec
                  #f64(4.48    7.13    8.37    10.83   14.06   17.74   21.42   24.82   28.23   30.88   33.52   36.17   36.89   37.63   38.38   38.38)    ;MEA.heat
                  #f64(2.87    5.26    6.95    9.10    11.73   14.53   17.34   20.26   23.19   25.13   27.07   29.03   30.48   32.00   33.60   33.60)    ;CPA.elec
                  #f64(16.94   25.43   45.09   50.66   55.04   54.19   53.33   49.40   45.46   43.32   41.18   39.05   38.26   37.50   36.75   36.75)    ;CPA.heat
                  #f64(1.23    2.24    2.75    3.53    4.70    6.34    7.99    11.09   14.22   19.21   24.21   29.23   32.15   35.36   38.90   38.90)    ;SAS.elec
                  #f64(6.31    14.84   20.97   24.75   27.75   32.07   36.39   42.43   48.47   52.58   56.69   60.81   63.85   67.04   70.39   70.39) )) ;SAS.heat

;(demand-get cStep cZone cService)
(define-macro (demand-get var-Step var-Zone var-Service)
  `(f64vector-ref (vector-ref demand (+ (- ,var-Service 1) (* (- ,var-Zone 1) nServicePeriod) )) (- ,var-Step 1)) )


;#################################################################################################################################################################################

                       ;Elec  Heat
(define effic-0  '#(#f64(0.5   0.9)   ;coal-0
                   #f64(0.4   0.)    ;coal-cg
                   #f64(0.35  0.8)   ;coal-ccs
                   #f64(0.3   0.)    ;coal-cg-ccs
                   #f64(0.5   0.9)   ;oil-0
                   #f64(0.45  0.)    ;oil-cg
                   #f64(0.40  0.8)   ;oil-ccs
                   #f64(0.35  0.)    ;oil-cg-ccs
                   #f64(0.55  0.9)   ;NG-0
                   #f64(0.50  0.)    ;NG-cg
                   #f64(0.45  0.8)   ;NG-ccs
                   #f64(0.40  0.)    ;NG-cg-ccs
                   #f64(0.5   0.9)   ;bio-0
                   #f64(0.35  0.)    ;bio-cg
                   #f64(0.30  0.8)   ;bio-ccs
                   #f64(0.25  0.)    ;bio-cg-ccs
                   #f64(0.33  0.)    ;nuclear-0
                   #f64(0.    0.)    ;nuclear-cg
                   #f64(0.    0.)    ;nuclear-ccs
                   #f64(0.    0.)    ;nuclear-cg-ccs
                   #f64(1.    0.)    ;wind-0
```

```
                    #f64(0.    0.)    ;wind-cg
                    #f64(0.    0.)    ;wind-ccs
                    #f64(0.    0.)    ;wind-cg-ccs
                    #f64(1.    0.)    ;hydro-0
                    #f64(0.    0.)    ;hydro-cg
                    #f64(0.    0.)    ;hydro-ccs
                    #f64(0.    0.)    ;hydro-cg-ccs
                    #f64(1.    1.)    ;solar-0
                    #f64(0.    0.)    ;solar-cg
                    #f64(0.    0.)    ;solar-ccs
                    #f64(0.    0.)    ;solar-cg-ccs
                    #f64(1.    0.)    ;solar-csp-0
                    #f64(0.    0.)    ;solar-csp-cg
                    #f64(0.    0.)    ;solar-csp-ccs
                    #f64(0.    0.)    ;solar-csp-cg-ccs
                    #f64(0.55  0.9)   ;h2-0
                    #f64(0.50  0.)    ;h2-cg
                    #f64(0.    0.)    ;h2-ccs
                    #f64(0.    0.)    ;h2-cg-ccs
                    #f64(0.    0.95)  ;elec-0
                    #f64(0.    0.)    ;elec-cg
                    #f64(0.    0.)    ;elec-ccs
                    #f64(0.    0.)))  ;elec-cg-ccs

;(effic-0-get cService cSource cType)
(define-macro (effic-0-get var-Service var-Source var-Type)
  `(f64vector-ref (vector-ref effic-0 (+ (* (1- ,var-Source) nTypePeriod) (1- ,var-Type))) (1- ,var-Service)) )

;(effic-get cStep cService cSource cType)
(define-macro (effic-get var-Step var-Service var-Source var-Type)
  `(let ((eff (effic-0-get ,var-Service ,var-Source ,var-Type)))
     (if (= 0. eff)
         0.
         (if (< 0. eff)
             (let* ((t-tech-effic 30)
                    (effic-calc (+ (* (/ 0.1 t-tech-effic) (* (1- ,var-Step) t-step)) (- eff 0.1))))
               (min eff effic-calc))
             (error "eff mindre en noll" eff)))))


;############################################################################################################################################################################
;RESOURCE TRANSFER (BUYING) System 2 (and System 3 for selling implicitly)
;############################################################################################################################################################################

                          ;Cost
(define imp-cost   '#(#f64(1.0)    ;Coal
                      #f64(0.5)    ;Oil
                      #f64(1.5)    ;NG
                      #f64(2.0)    ;Bio
                      #f64(0.5))) ;Nuclear

;(imp-cost-get cSource)
(define-macro (imp-cost-get var-Source)
  `(f64vector-ref (vector-ref imp-cost (1- ,var-Source)) 0) )


;############################################################################################################################################################################

                          ;Cost
(define imp-cost-lin  '#(#f64(0.00001)    ;Coal
                         #f64(0.00001)    ;Oil
                         #f64(0.00002)    ;NG
                         #f64(0.00001)    ;Bio
                         #f64(0.00001))) ;Nuclear

;(imp-cost-lin-get cSource)
(define-macro (imp-cost-lin-get var-Source)
  `(f64vector-ref (vector-ref imp-cost-lin (1- ,var-Source)) 0) )


;##########################################################################################################################################################

                         ;NAM     EUR     PAO     FSU     AFR     PAS     LAM     MEA     CPA     SAS
(define distance  '#(#f64(0.      6800.   9600.   7900.   10400.  13300.  8100.   10000.  10100.  12800.) ;NAM
                     #f64(6800.   0.      9700.   2500.   4200.   9400.   8713.   3200.   8200.   7000.)  ;EUR
                     #f64(9600.   9700.   0.      7500.   12200.  4600.   17700.  9600.   2100.   6700.)  ;PAO
                     #f64(7900.   2500.   7500.   0.      5200.   7100.   11200.  2900.   5800.   5000.)  ;FSU
                     #f64(10400.  4200.   12200.  5200.   0.      9200.   7600.   2600.   10100.  6200.)  ;AFR
                     #f64(13300.  9400.   4600.   7100.   9200.   0.      16600.  7300.   3300.   3000.)  ;PAS
                     #f64(8100.   8713.   17700.  11200.  7600.   16600.  0.      9900.   16900.  13800.) ;LAM
                     #f64(10000.  3200.   9600.   2900.   2600.   7300.   9900.   0.      7600.   4400.)  ;MEA
                     #f64(10100.  8200.   2100.   5800.   10100.  3300.   16900.  7600.   0.      4800.)  ;CPA
                     #f64(12800.  7000.   6700.   5000.   6200.   3000.   13800.  4400.   4800.   0.)))   ;SAS

;(distance-get cZone cTrancferZone)
(define-macro (distance-get var-Zone var-TrancferZone)
  `(f64vector-ref (vector-ref distance (1- ,var-Zone)) (1- ,var-TrancferZone)) )


;#########################################################################################################################################################
;CONVERSION (PLANTS) System 4
;#########################################################################################################################################################

;(life-plant-get cService cSource cType)
(define-macro (life-plant-get var-Service var-Source var-Type)
  `(if (= ,var-Source 7)
     (if (and (= ,var-Service 1) (= ,var-Type 1))
        40. 25.)
     (if (= ,var-Source 9)
       (if (and (= ,var-Service 1) (= ,var-Type 1))
         30. 25. )
       25. )))


;#########################################################################################################################################################

;(OM-cost-fr-get cService cSource cType)
(define-macro (OM-cost-fr-get var-Service var-Source var-Type)
  `(if (and (= ,var-Source 9) (= ,var-Service 1))
       0.014 0.04))


;##########################################################################################################################################################

                              ;Elec    Heat
(define cost-inv-base   '#(#f64(1100.   300.)  ;coal-0
                           #f64(1200.   0.)    ;coal-cg
                           #f64(1500.   500.)  ;coal-ccs
                           #f64(1600.   0.)    ;coal-cg-ccs
                           #f64(600.    100.)  ;oil-0
```

```
                              #f64(700.   0.)    ;oil-cg
                              #f64(1000.  300.)  ;oil-ccs
                              #f64(1100.  0.)    ;oil-cg-ccs
                              #f64(500.   100.)  ;NG-0
                              #f64(600.   0.)    ;NG-cg
                              #f64(900.   300.)  ;NG-ccs
                              #f64(1000.  0.)    ;NG-cg-ccs
                              #f64(1200.  300.)  ;bio-0
                              #f64(1300.  0.)    ;bio-cg
                              #f64(1700.  500.)  ;bio-ccs
                              #f64(1800.  0.)    ;bio-cg-ccs
                              #f64(2000.  0.)    ;nuclear-0
                              #f64(0.     0.)    ;nuclear-cg
                              #f64(0.     0.)    ;nuclear-ccs
                              #f64(0.     0.)    ;nuclear-cg-ccs
                              #f64(600.   0.)    ;wind-0
                              #f64(0.     0.)    ;wind-cg
                              #f64(0.     0.)    ;wind-ccs
                              #f64(0.     0.)    ;wind-cg-ccs
                              #f64(1000.  0.)    ;hydro-0
                              #f64(0.     0.)    ;hydro-cg
                              #f64(0.     0.)    ;hydro-ccs
                              #f64(0.     0.)    ;hydro-cg-ccs
                              #f64(1200.  400.)  ;solar-0
                              #f64(0.     0.)    ;solar-cg
                              #f64(0.     0.)    ;solar-ccs
                              #f64(0.     0.)    ;solar-cg-ccs
                              #f64(3200.  0.)    ;solar-csp-0
                              #f64(0.     0.)    ;solar-csp-cg
                              #f64(0.     0.)    ;solar-csp-ccs
                              #f64(0.     0.)    ;solar-csp-cg-ccs
                              #f64(500.   100.)  ;h2-0
                              #f64(600.   0.)    ;h2-cg
                              #f64(0.     0.)    ;h2-ccs
                              #f64(0.     0.)    ;h2-cg-ccs
                              #f64(0.     100.)  ;elec-0
                              #f64(0.     0.)    ;elec-cg
                              #f64(0.     0.)    ;elec-ccs
                              #f64(0.     0.)))  ;elec-cg-ccs

;(cost-inv-get cService cSource cType)
(define-macro (cost-inv-get var-Service var-Source var-Type)
  `(f64vector-ref (vector-ref cost-inv-base (+ (* (1- ,var-Source) nTypePeriod) (1- ,var-Type))) (1- ,var-Service)) )

;(cost-inv-mod-get cService cSource cType)
(define-macro (cost-inv-mod-get var-Service var-Source var-Type)
  `(let ((dr_invest 5) (dr 5) (lifeLengthPlant (life-plant-get ,var-Service ,var-Source ,var-Type)))
     (* (cost-inv-get ,var-Service ,var-Source ,var-Type) (/ (+ dr_invest (/ 1 lifeLengthPlant)) (+ dr (/ 1 lifeLengthPlant))))) )


;#####################################################################################################################################################################

                              ;Elec  Heat
(define lf-global  '#(#f64(0.7   0.7)   ;coal-0
                      #f64(0.7   0.)    ;coal-cg
                      #f64(0.7   0.7)   ;coal-ccs
                      #f64(0.7   0.)    ;coal-cg-ccs
                      #f64(0.7   0.7)   ;oil-0
                      #f64(0.7   0.)    ;oil-cg
                      #f64(0.7   0.7)   ;oil-ccs
                      #f64(0.7   0.)    ;oil-cg-ccs
                      #f64(0.7   0.7)   ;NG-0
                      #f64(0.7   0.)    ;NG-cg
                      #f64(0.7   0.7)   ;NG-ccs
                      #f64(0.7   0.)    ;NG-cg-ccs
                      #f64(0.7   0.7)   ;bio-0
                      #f64(0.7   0.)    ;bio-cg
                      #f64(0.7   0.7)   ;bio-ccs
                      #f64(0.7   0.)    ;bio-cg-ccs
                      #f64(0.7   0.)    ;nuclear-0
                      #f64(0.    0.)    ;nuclear-cg
                      #f64(0.    0.)    ;nuclear-ccs
                      #f64(0.    0.)    ;nuclear-cg-ccs
                      #f64(0.25  0.)    ;wind-0
                      #f64(0.    0.)    ;wind-cg
                      #f64(0.    0.)    ;wind-ccs
                      #f64(0.    0.)    ;wind-cg-ccs
                      #f64(0.7   0.)    ;hydro-0
                      #f64(0.    0.)    ;hydro-cg
                      #f64(0.    0.)    ;hydro-ccs
                      #f64(0.    0.)    ;hydro-cg-ccs
                      #f64(0.25  0.25)  ;solar-0
                      #f64(0.    0.)    ;solar-cg
                      #f64(0.    0.)    ;solar-ccs
                      #f64(0.    0.)    ;solar-cg-ccs
                      #f64(0.6   0.)    ;solar-csp-0
                      #f64(0.    0.)    ;solar-csp-cg
                      #f64(0.    0.)    ;solar-csp-ccs
                      #f64(0.    0.)    ;solar-csp-cg-ccs
                      #f64(0.7   0.7)   ;h2-0
                      #f64(0.7   0.)    ;h2-cg
                      #f64(0.    0.)    ;h2-ccs
                      #f64(0.    0.)    ;h2-cg-ccs
                      #f64(0.    1.)    ;elec-0
                      #f64(0.    0.)    ;elec-cg
                      #f64(0.    0.)    ;elec-ccs
                      #f64(0.    0.)))  ;elec-cg-ccs

;(lf-global-get cService cSource cType)
(define-macro (lf-global-get var-Service var-Source var-Type)
  `(f64vector-ref (vector-ref lf-global (+ (* (1- ,var-Source) nTypePeriod) (1- ,var-Type))) (1- ,var-Service)) )

                                       ;NAM   EUR   PAO   FSU   AFR   PAS   LAM   MEA   CPA   SAS
(define lf-global-Zone-Solar-0  '#(#f64(0.27  0.25  0.25  0.25  0.28  0.28  0.28  0.28  0.28  0.28)    ;Elec
                                   #f64(0.27  0.25  0.25  0.25  0.28  0.28  0.28  0.28  0.28  0.28) )) ;Heat


;(lf-get cZone cService cSource cType)
(define-macro (lf-get var-Zone var-Service var-Source var-Type)
  `(if (= ,var-Source 8)
     (if (= ,var-Type 1)
       (f64vector-ref (vector-ref lf-global-Zone-Solar-0 (1- ,var-Service)) (1- ,var-Zone))
       (lf-global-get ,var-Service ,var-Source ,var-Type) )
     (lf-global-get ,var-Service ,var-Source ,var-Type) ))


;#############################################################################################################################################################################

                           ;NAM      EUR      PAO      FSU      AFR      PAS      LAM      MEA      CPA      SAS
(define init-Cap  '#(#f64(0.39000  0.24000  0.06000  0.08000  0.02850  0.03000  0.00500  0.00400  0.23500  0.05000)    ;coal.elec.0
                     #f64(0.08259  0.34336  0.00584  0.37903  0.15683  0.31040  0.27984  0.04344  0.26989  0.30000)    ;coal.heat.0
                     #f64(0.04000  0.05700  0.04000  0.04000  0.00300  0.03000  0.03000  0.04400  0.00800  0.00600)    ;oil.elec.0
                     #f64(0.72370  0.77939  0.16286  0.35523  0.24350  0.04772  0.00784  0.27326  0.15058  0.20000)    ;oil.heat.0
                     #f64(1.28615  0.75803  0.20182  0.35448  0.10533  0.14582  0.30928  0.22679  0.18356  0.14197)    ;oil.petro.0
                     #f64(0.08000  0.04700  0.05000  0.18000  0.00080  0.00800  0.02000  0.04000  0.00070  0.00500)    ;NG.elec.0
                     #f64(0.23354  0.11133  0.09835  0.63462  0.46688  0.00970  0.13464  0.01415  0.33810  0.40000)    ;NG.heat.0
                     #f64(0.01550  0.00530  0.02100  0.00384  0.00000  0.00096  0.00240  0.00043  0.00098  0.00096)    ;bio.elec.0
                     #f64(0.13521  0.12215  0.02391  0.03511  0.37960  0.16549  0.14642  0.00358  0.38700  0.34000)    ;bio.heat.0
                     #f64(0.13212  0.15489  0.03886  0.04098  0.00183  0.01650  0.00266  0.00000  0.00000  0.00107)    ;nuclear.elec.0
                     #f64(0.12510  0.10505  0.02777  0.05554  0.00932  0.00785  0.08633  0.00483  0.03575  0.02113) )) ;hydro.elec.0


;(init-Cap-get cZone cService cSource cType)
(define-macro (init-Cap-get var-cZone var-cService var-cSource var-cType)
  `(cond ((> ,var-cType 1) 0.) ;Remove all types other then .0
         ((and (= ,var-cService 7) (= ,var-cSource 2)) (f64vector-ref (vector-ref init-Cap 4) (1- ,var-cZone))) ;deliver petrolium row
         ((> ,var-cService 2) 0.) ;deliver 0 if Service is not elec or heat
         ((and (> ,var-cSource 5) (not (= ,var-cSource 7))) 0.) ;If source not in list deliver 0
         ((and (or (= ,var-cSource 5) (= ,var-cSource 7)) (not (= ,var-cService 1))) 0.) ;if heat from nuclear or hydro deliver 0
         (else (f64vector-ref (vector-ref init-Cap (cond ((= ,var-cSource 1) (+ 0 (1- ,var-cService)))
                                                                        ((= ,var-cSource 2) (+ 2 (1- ,var-cService)))
                                                                        ((= ,var-cSource 3) (+ 5 (1- ,var-cService)))
                                                                        ((= ,var-cSource 4) (+ 7 (1- ,var-cService)))
                                                                        ((= ,var-cSource 5) (+ 9 (1- ,var-cService)))
                                                                        ((= ,var-cSource 7) (+ 10 (1- ,var-cService))))) (1- ,var-cZone)))))


;###################################################################################################################################################################################
;EMISSION CONSTANTS
;###################################################################################################################################################################################

                          ;Cost
(define emis-fact  '#(#f64(24.7)   ;Coal
                      #f64(20.5)   ;Oil
                      #f64(15.4)   ;NG
                      #f64(32.0)   ;Bio
                      #f64( 0.0)   ;Nuclear
                      #f64(19.1))) ;Meoh

;(emis-fact-get cSource)
(define-macro (emis-fact-get var-Source)
  `(f64vector-ref (vector-ref emis-fact (1- ,var-Source)) 0) )
```

Two Dependency Libraries

As explained in the report scheme is a language that builds on the programmer being free to make changes to the language when needed. Some new things were introduced to the scheme code during this project.

Firstly an FFI used to communicate with the GUROBI optimizer from GAMBIT. The code of this API is included here and it is also published on the site GAMBIT Dumping Grounds (http://gambitscheme.org/wiki/index.php/Dumping_Grounds).

Secondly some macros were written to safely handle mutation in the code. This is perhaps the least interesting and least readable section of the code. It is included only for transparency and by principle. The file is called `"verktyg.scm"`.

GUROBI FFI

```
; To compile on Windows MSVC 32bit:
; Komandot i gambit för att kompilera denhär filen: (compile-file "gurobiffi.scm" cc-options: "/ERRORREPORT:PROMPT  -I 
c:\\gurobi451\\win32\\include" ld-options-prelude: "\\gurobi451\\win32\\lib\\gurobi45.lib 
\\gurobi451\\win32\\lib\\gurobi_c++mtd2010.lib")
; (load "gurobiffi.oX")

(declare (not interrupts-enabled))

(c-declare #<<c-declare-end

// #pragma comment(lib,"gurobi45.lib")
// #pragma comment(lib,"gurobi_c++mtd2010.lib")
#include <stdio.h>
// #include <float.h>
#include "gurobi_c.h"

// printf("Floating point values are %i bytes on this puter.\n",sizeof(double));

c-declare-end
           )


(define testa-c2 (c-lambda () void #<<KODEN
printf("C funkade!\n");
KODEN
                           ))
```

```
(c-declare #<<c-declare-end

___SCMOBJ release_grb_env_star(void* p) {
     GRBfreeenv((GRBenv*) p); // fprintf(stderr,"gurobiffi: release_grb_env_star: %i garbage collected.\n",(int) p);
     return ___FIX(___NO_ERR);
}

___SCMOBJ release_grb_model_star(void* p) {
     GRBfreemodel((GRBmodel*) p); // fprintf(stderr,"gurobiffi: release_grb_model_star: %i garbage collected.\n",(int) p);
     return ___FIX(___NO_ERR);
}

c-declare-end
           )

; (c-define-type grb-env "GRBenv")
(c-define-type grb-env*   (nonnull-pointer "GRBenv"   grb-env*   "release_grb_env_star"  ))
(c-define-type grb-model* (nonnull-pointer "GRBmodel" grb-model* "release_grb_model_star"))

; Import constants:
(define grb-less-equal    ((c-lambda () char "___result = GRB_LESS_EQUAL;"   )))
(define grb-greater-equal ((c-lambda () char "___result = GRB_GREATER_EQUAL;")))
(define grb-equal         ((c-lambda () char "___result = GRB_EQUAL;")))

(define grb-continuous ((c-lambda () char "___result = GRB_CONTINUOUS;")))
(define grb-binary     ((c-lambda () char "___result = GRB_BINARY;")))
(define grb-integer    ((c-lambda () char "___result = GRB_INTEGER;")))
(define grb-semicont   ((c-lambda () char "___result = GRB_SEMICONT;")))
(define grb-semiint    ((c-lambda () char "___result = GRB_SEMIINT;")))

(define grb-loaded           ((c-lambda () int "___result = GRB_LOADED;")))
(define grb-optimal          ((c-lambda () int "___result = GRB_OPTIMAL;")))
(define grb-infeasible       ((c-lambda () int "___result = GRB_INFEASIBLE;")))
(define grb-inf-or-unbd      ((c-lambda () int "___result = GRB_INF_OR_UNBD;")))
(define grb-unbounded        ((c-lambda () int "___result = GRB_UNBOUNDED;")))
(define grb-cutoff           ((c-lambda () int "___result = GRB_CUTOFF;")))
(define grb-iteration-limit ((c-lambda () int "___result = GRB_ITERATION_LIMIT;")))
(define grb-node-limit       ((c-lambda () int "___result = GRB_NODE_LIMIT;")))
(define grb-time-limit       ((c-lambda () int "___result = GRB_TIME_LIMIT;")))
(define grb-solution-limit   ((c-lambda () int "___result = GRB_SOLUTION_LIMIT;")))
(define grb-interrupted      ((c-lambda () int "___result = GRB_INTERRUPTED;")))
(define grb-numeric          ((c-lambda () int "___result = GRB_NUMERIC;")))
```

```
(define grb-suboptimal      ((c-lambda () int "___result = GRB_SUBOPTIMAL;")))

(define grb-optimize-status-codes (list->table `((,grb-loaded . loaded)
                                                 (,grb-optimal . optimal)
                                                 (,grb-infeasible . infeasible)
                                                 (,grb-inf-or-unbd . inf-or-unbd)
                                                 (,grb-unbounded . unbounded)
                                                 (,grb-cutoff . cutoff)
                                                 (,grb-iteration-limit . iteration-limit)
                                                 (,grb-node-limit . node-limit)
                                                 (,grb-time-limit . time-limit)
                                                 (,grb-solution-limit . solution-limit)
                                                 (,grb-interrupted . interrupted)
                                                 (,grb-numeric . numeric)
                                                 (,grb-suboptimal . suboptimal))
                                               test: eq?))

(define (import-grb-optimize-status-codes v) (table-ref grb-optimize-status-codes v))

(define grb-load-env (c-lambda (char-string) grb-env* #<<c-declare-end
GRBenv* envP;
const char* logfilename = ___arg1;
int r = GRBloadenv(&envP,logfilename);
if (r == 0) ___result_voidstar = envP; else ___result_voidstar = 0;
c-declare-end
))

(define grb-env (grb-load-env #f))

(define f64vectorref   (c-lambda (scheme-object int) double "double* d = &___F64VECTORREF(___arg1,0); ___result = d[___arg2];"))
; obsolete: (define f64vectorref+1/2 (c-lambda (scheme-object) double "double* d = &___F64VECTORREF(___arg1,1); ___result = *d;"))
; obsolete: (define f64vectorref+1/3 (c-lambda (scheme-object) double "double d = ___F64VECTORREF(___arg1,1); ___result = d;"))

; (define model (grb-load-model grb-env "example" 3 2 -1 0. '#f64(1. 1. 2.) (list->string (list grb-less-equal grb-greater-equal))
;                               '#f64(4. 1.) '#s32(0 2 4) '#s32(2 2 1) '#s32(0 1 0 1 0) '#f64(1. 1. 2. 1. 3.)
;                               #f #f #f #f (list->string (list grb-binary grb-binary grb-binary)) '() '()))
```

```
(define grb-load-model (c-lambda (grb-env* nonnull-char-string int int int double
                                  scheme-object nonnull-char-string scheme-object scheme-object scheme-object scheme-object
                                  scheme-object
                                  bool scheme-object ; = lb-enabler lb-content
                                  bool scheme-object ; = ub-enabler ub-content
                                  nonnull-char-string ; = vtype
                                  bool nonnull-char-string-list ; = varnames-enabler varnames-content
                                  bool nonnull-char-string-list ; = constrnames-enabler constrnames-content
                                  )
                                 grb-model*
                                    #<<c-declare-end
// printf("Into GRBloadmodel. scheme-object args are: %p %p %p %p %p %p %b %p %b
%p\n",___arg7,___arg9,___arg10,___arg11,___arg12,___arg13,___arg14,___arg15,___arg16,___arg17);
{
GRBenv* env = ___arg1;
GRBmodel *modelP;
char* Pname = ___arg2;
double* obj = &___F64VECTORREF(___arg7,0);
char* sense = ___arg8;
double* rhs = &___F64VECTORREF(___arg9,0);
int numvars = ___arg3; int numconsrs = ___arg4; int objsense = ___arg5; int objcon = ___arg6;
int* vbeg = ___CAST(int*,&___FETCH_S32(___BODY(___arg10),___INT(0)));
int* vlen = ___CAST(int*,&___FETCH_S32(___BODY(___arg11),___INT(0)));
int* vind = ___CAST(int*,&___FETCH_S32(___BODY(___arg12),___INT(0)));

double* vval = &___F64VECTORREF(___arg13,0); // ___CAST(double*,___FETCH_U8(___BODY(___arg13),___INT(0)));
// lb sätts såhär: Ifall argument 14 är #f, så används INGEN lb. Ifall argument inte är 14 (så t.ex. #t),
// så förväntas argument 15 vara en lista av strängar (dvs '("a" "b" "c") etc. ), och tas i användning.
double* lb   = ___arg14 ? &___F64VECTORREF(___arg15,0) : NULL; //
___CAST(double*,___FETCH_U8(___BODY(___arg15),___INT(0))) : NULL;
double* ub   = ___arg16 ? &___F64VECTORREF(___arg17,0) : NULL; //
___CAST(double*,___FETCH_U8(___BODY(___arg17),___INT(0))) : NULL;
char* vtype = ___arg18;
char** varnames = ___arg19 ? ___arg20 : NULL;
char** constrnames = ___arg21 ? ___arg22 : NULL;
{
// printf("Varnames: First is %s second is %s.\n",varnames[0],varnames[1]);
// double testval = 1.23; if(___arg1&&___arg3) {testval += 0.2;}
// printf("GRBloadmodel pointers are: %p %p %p %p %p %p %p %p\n",obj,rhs,vbeg,vlen,vind,vval,lb,ub);

// printf("GRBloadmodel is invoked with arguments: %p %p %s %i %i %I %i %p %s %p %p %p %p %p %p %p %s %p %p\n",
//
env,&modelP,Pname,numvars,numconsrs,objsense,objcon,obj,sense,rhs,vbeg,vlen,vind,vval,lb,ub,vtype,varnames,constrnames);
// printf("testval %e\n",testval); testval -= 0.01; printf("testval %f\n",testval);
// printf("obj els are: %i %i %i %i %i %i %i %i\n",obj[0],obj[1],rhs[0],rhs[1],vbeg[0],vbeg[1],vval[0],vval[1]);
// printf("picked directly obj els are: %i %i %i %i %i %i\n",___F64VECTORREF(___arg7,0),___F64VECTORREF(___arg7,1),
//        ___F64VECTORREF(___arg9,0),___F64VECTORREF(___arg9,1),
//        ___F64VECTORREF(___arg13,0),___F64VECTORREF(___arg13,1));
// printf("vbeg vlen vind has: %i %i %i %i %i %i %i %i
%i\n",vbeg[0],vbeg[1],vbeg[2],vlen[0],vlen[1],vlen[2],vind[0],vind[1],vind[2]);
{
int r = GRBloadmodel(env, // env
                     &modelP, // modeIP
                     Pname, // Pname
                     numvars,numconsrs,objsense,objcon, // numvars numconsrs objsense objcon
                     obj, // obj
                     sense, // sense
                     rhs, // rhs
                     vbeg, vlen, vind, // vbeg vlen vind
                     vval, lb, ub, // vval lb ub
                     vtype, varnames, constrnames); // vtype varnames constnames
// printf("GRBloadmodel returned! r: %i\n",r);
if (r == 0)

    ___result_voidstar = modelP;
else {
     ___result_voidstar = 0;
     printf("GRBloadmodel error: %s\n",GRBgeterrormsg(___arg1));
}
}}}
c-declare-end
))

(define grb-optimize (c-lambda (grb-env* grb-model*) bool #<<c-declare-end
GRBenv* env = ___arg1;
GRBmodel* model = ___arg2;
int r = GRBoptimize(model);
___result = r == 0;
if (r != 0) printf("GRBloadmodel error: %s\n",GRBgeterrormsg(env));
c-declare-end
))
```

```
(define grb-get-int-attr (c-lambda (grb-env* grb-model* nonnull-char-string scheme-object) bool #<<c-declare-end
GRBenv* env = ___arg1;
GRBmodel* model = ___arg2;
char* name = ___arg3;
int* i = ___CAST(int*,&___FETCH_S32(___BODY(___arg4),___INT(0)));
int r = GRBgetintattr(model,name,i);
if (r != 0) printf("GRBloadmodel error: %s\n",GRBgeterrormsg(env));
___result = r == 0;
c-declare-end
 ))

(define grb-get-int-attr*
  (let ((r-container (make-s32vector 1)))
    (lambda (env model param-name)
      (and (grb-get-int-attr env model param-name r-container) (s32vector-ref r-container 0)))))

(define (grb-model-status env model) (import-grb-optimize-status-codes (grb-get-int-attr* env model "Status")))

(define grb-get-dbl-attr (c-lambda (grb-env* grb-model* nonnull-char-string scheme-object) bool #<<c-declare-end
GRBenv* env = ___arg1;
GRBmodel* model = ___arg2;
char* name = ___arg3;
double* d = &___F64VECTORREF(___arg4,0);
int r = GRBgetdblattr(model,name,d);
if (r != 0) printf("GRBloadmodel error: %s\n",GRBgeterrormsg(env));
___result = r == 0;
c-declare-end
 ))

(define grb-get-dbl-attr*
  (let ((r-container (make-f64vector 1)))
    (lambda (env model param-name)
      (and (grb-get-dbl-attr env model param-name r-container) (f64vector-ref r-container 0)))))
```

```
(define grb-get-dbl-attr-element (c-lambda (grb-env* grb-model* nonnull-char-string int scheme-object) bool #<<c-declare-
end
GRBenv* env = ___arg1;
GRBmodel* model = ___arg2;
char* name = ___arg3;
int index = ___arg4;
double* d = &___F64VECTORREF(___arg5,0);
int r = GRBgetdblattrelement(model,name,index,d);
if (r != 0) printf("GRBloadmodel error: %s\n",GRBgeterrormsg(env));
___result = r == 0;
c-declare-end
 ))

(define grb-get-dbl-attr-element*
  (let ((r-container (make-f64vector 1)))
    (lambda (env model param-name index)
      (and (grb-get-dbl-attr-element env model param-name index r-container) (f64vector-ref r-container 0)))))

(define grb-get-str-attr-element (c-lambda (grb-env* grb-model* nonnull-char-string int) nonnull-char-string #<<c-declare-
end
GRBenv* env = ___arg1;
GRBmodel* model = ___arg2;
char* name = ___arg3;
int index = ___arg4;
char* s = NULL;
int r = GRBgetstrattrelement(model,name,index,&s);
if (r != 0) printf("GRBloadmodel error: %s\n",GRBgeterrormsg(env));
___result = r == 0 ? s : NULL;
c-declare-end
 ))

(define grb-get-dbl-param-info (c-lambda (grb-env* nonnull-char-string scheme-object) bool #<<c-declare-end
GRBenv* env = ___arg1;
char* s = ___arg2;
double* rvalues = &___F64VECTORREF(___arg3,0);
int r = GRBgetdblparaminfo(env,s,&rvalues[0],&rvalues[1],&rvalues[2],&rvalues[3]);
___result = r == 0;
if (r != 0) printf("GRBloadmodel error: %s\n",GRBgeterrormsg(___arg1));
c-declare-end
))
```

```
(define (grb-get-dbl-param-info* env param-name)
  (let ((v (make-f64vector 4)))
    (and (grb-get-dbl-param-info grb-env param-name v) v)))


(define grb-set-method (c-lambda (grb-env* int) bool #<<c-declare-end
GRBenv* env = ___arg1;
int i = ___arg2;
int r =  GRBsetintparam(env, "Method", i);
___result = r == 0;
if (r != 0) printf("GRBloadmodel error: %s\n", GRBgeterrormsg(___arg1));
c-declare-end
))


(define grb-set-number-of-threds (c-lambda (grb-env* int) bool #<<c-declare-end
GRBenv* env = ___arg1;
int i = ___arg2;
int r =  GRBsetintparam(env, "Threads", i);
___result = r == 0;
if (r != 0) printf("GRBloadmodel error: %s\n", GRBgeterrormsg(___arg1));
c-declare-end
))
```

Macros, "verktyg.scm"

```scheme
(define-macro (for-interval var-name start-value end-value . code)
  `(let loop ((,var-name ,start-value))
     ,@code
     (if (< ,var-name ,end-value) (loop (+ ,var-name 1)))))


(define-macro (** u v)
  `(if (= ,v 0) 1
     (let ((cPow ,1))
       (for-interval nPow 1 ,v
         (set! cPow (* cPow ,u)))
       cPow )))


; (with-list-maker (cons! 1) (cons! 2)) => '(1 2)
; code has access to procedures cons! , tail! and get-list .
; (Making this fully as a macro, in order to ensure that the code always will get max speed.)
(define-macro (with-list-maker . code) ; #!key return-thunk-rv? #!rest code) - bigloo can't do this.
  (let* ((return-thunk-rv?-set? (and (not (null? code))
                                     (eq? 'return-thunk-rv?: (car code))))
         (return-thunk-rv?      (and return-thunk-rv?-set?
                                     (cadr code)))
         (code                  (if return-thunk-rv?-set? (cddr code) code)))
    `(let* ((list-maker-thunk (lambda (cons! tail! get-list) ,@code))
            (list '())
            (last-element #f)
            (cons! (lambda (v) (let ((c (cons v '())))
                                 (if last-element
                                     (set-cdr! last-element c)
                                     (set! list c))
                                 (set! last-element c))))
            (tail! (lambda (tail) (if last-element (set-cdr! last-element tail) (set! list tail))
                                  (set! last-element 'terminated)))
            (get-list (lambda () list)))
       (list-maker-thunk cons! tail! get-list)
       ,@(if return-thunk-rv? '() '(list)))))
```

```scheme
; Same as with-list-maker but cons! is named cons!2 and get-list is named get-list2 .
(define-macro (with-list-maker2 . code) ; #!key return-thunk-rv? #!rest code) - bigloo can't do this.
  (let* ((return-thunk-rv?-set? (and (not (null? code))
                                     (eq? 'return-thunk-rv?: (car code))))
         (return-thunk-rv?      (and return-thunk-rv?-set?
                                     (cadr code)))
         (code                  (if return-thunk-rv?-set? (cddr code) code)))
    `(let* ((list-maker-thunk2 (lambda (cons!2 get-list2) ,@code))
            ; All the use of 2 from here and down is really not needed, maintained for
            ; extra code clarity only.
            (list2 '())
            (last-element2 #f)
            (cons!2 (lambda (v) (let ((c (cons v '())))
                                 (if last-element2
                                     (set-cdr! last-element2 c)
                                     (set! list2 c))
                                 (set! last-element2 c))))
            (get-list2 (lambda () list2)))
       (list-maker-thunk2 cons!2 get-list2)
       ,@(if return-thunk-rv? '() '(list2)))))

; Same as with-list-maker but cons! is named cons!3 and get-list is named get-list3 .
(define-macro (with-list-maker3 . code) ; #!key return-thunk-rv? #!rest code) - bigloo can't do this.
  (let* ((return-thunk-rv?-set? (and (not (null? code))
                                     (eq? 'return-thunk-rv?: (car code))))
         (return-thunk-rv?      (and return-thunk-rv?-set?
                                     (cadr code)))
         (code                  (if return-thunk-rv?-set? (cddr code) code)))
    `(let* ((list-maker-thunk3 (lambda (cons!3 get-list3) ,@code))
            ; All the use of 3 from here and down is really not needed, maintained for
            ; extra code clarity only.
            (list3 '())
            (last-element3 #f)
            (cons!3 (lambda (v) (let ((c (cons v '())))
                                 (if last-element3
                                     (set-cdr! last-element3 c)
                                     (set! list3 c))
                                 (set! last-element3 c))))
            (get-list3 (lambda () list3)))
       (list-maker-thunk3 cons!3 get-list3)
       ,@(if return-thunk-rv? '() '(list3)))))
```

```scheme
; Same as with-list-maker but cons! is named cons!4 and get-list is named get-list4 .
(define-macro (with-list-maker4 . code) ; #!key return-thunk-rv? #!rest code) - bigloo can't do this.
  (let* ((return-thunk-rv?-set? (and (not (null? code))
                                     (eq? 'return-thunk-rv?: (car code))))
         (return-thunk-rv?      (and return-thunk-rv?-set?
                                     (cadr code)))
         (code                  (if return-thunk-rv?-set? (cddr code) code)))
    `(let* ((list-maker-thunk4 (lambda (cons!4 get-list4) ,@code))
            ; All the use of 4 from here and down is really not needed, maintained for
            ; extra code clarity only.
            (list4 '())
            (last-element4 #f)
            (cons!4 (lambda (v) (let ((c (cons v '())))
                                 (if last-element4
                                     (set-cdr! last-element4 c)
                                     (set! list4 c))
                                 (set! last-element4 c))))
            (get-list4 (lambda () list4)))
       (list-maker-thunk4 cons!4 get-list4)
       ,@(if return-thunk-rv? '() '(list4)))))

; Same as with-list-maker but cons! is named cons!5 and get-list is named get-list5 .
(define-macro (with-list-maker5 . code) ; #!key return-thunk-rv? #!rest code) - bigloo can't do this.
  (let* ((return-thunk-rv?-set? (and (not (null? code))
                                     (eq? 'return-thunk-rv?: (car code))))
         (return-thunk-rv?      (and return-thunk-rv?-set?
                                     (cadr code)))
         (code                  (if return-thunk-rv?-set? (cddr code) code)))
    `(let* ((list-maker-thunk5 (lambda (cons!5 get-list5) ,@code))
            ; All the use of 5 from here and down is really not needed, maintained for
            ; extra code clarity only.
            (list5 '())
            (last-element5 #f)
            (cons!5 (lambda (v) (let ((c (cons v '())))
                                 (if last-element5
                                     (set-cdr! last-element5 c)
                                     (set! list5 c))
                                 (set! last-element5 c))))
            (get-list5 (lambda () list5)))
       (list-maker-thunk5 cons!5 get-list5)
       ,@(if return-thunk-rv? '() '(list5)))))
```

```
; Same as with-list-maker but cons! is named cons!6 and get-list is named get-list6 .
(define-macro (with-list-maker6 . code) ; #!key return-thunk-rv? #!rest code) - bigloo can't do this.
  (let* ((return-thunk-rv?-set? (and (not (null? code))
                                     (eq? 'return-thunk-rv?: (car code))))
         (return-thunk-rv?      (and return-thunk-rv?-set?
                                     (cadr code)))
         (code                  (if return-thunk-rv?-set? (cddr code) code)))
    `(let* ((list-maker-thunk6 (lambda (cons!6 get-list6) ,@code))
            ; All the use of 6 from here and down is really not needed, maintained for
            ; extra code clarity only.
            (list6 '())
            (last-element6 #f)
            (cons!6 (lambda (v) (let ((c (cons v '())))
                                 (if last-element6
                                     (set-cdr! last-element6 c)
                                     (set! list6 c))
                                 (set! last-element6 c))))
            (get-list6 (lambda () list6)))
       (list-maker-thunk6 cons!6 get-list6)
       ,@(if return-thunk-rv? '() '(list6)))))

; Same as with-list-maker but cons! is named cons!7 and get-list is named get-list7 .
(define-macro (with-list-maker7 . code) ; #!key return-thunk-rv? #!rest code) - bigloo can't do this.
  (let* ((return-thunk-rv?-set? (and (not (null? code))
                                     (eq? 'return-thunk-rv?: (car code))))
         (return-thunk-rv?      (and return-thunk-rv?-set?
                                     (cadr code)))
         (code                  (if return-thunk-rv?-set? (cddr code) code)))
    `(let* ((list-maker-thunk7 (lambda (cons!7 get-list7) ,@code))
            ; All the use of 7 from here and down is really not needed, maintained for
            ; extra code clarity only.
            (list7 '())
            (last-element7 #f)
            (cons!7 (lambda (v) (let ((c (cons v '())))
                                 (if last-element7
                                     (set-cdr! last-element7 c)
                                     (set! list7 c))
                                 (set! last-element7 c))))
            (get-list7 (lambda () list7)))
       (list-maker-thunk7 cons!7 get-list7)
       ,@(if return-thunk-rv? '() '(list7)))))
```

```
; Same as with-list-maker but cons! is named cons!8 and get-list is named get-list8 .
(define-macro (with-list-maker8 . code) ; #!key return-thunk-rv? #!rest code) - bigloo can't do this.
  (let* ((return-thunk-rv?-set? (and (not (null? code))
                                     (eq? 'return-thunk-rv?: (car code))))
         (return-thunk-rv?      (and return-thunk-rv?-set?
                                     (cadr code)))
         (code                  (if return-thunk-rv?-set? (cddr code) code)))
    `(let* ((list-maker-thunk8 (lambda (cons!8 get-list8) ,@code))
            ; All the use of 8 from here and down is really not needed, maintained for
            ; extra code clarity only.
            (list8 '())
            (last-element8 #f)
            (cons!8 (lambda (v) (let ((c (cons v '())))
                                 (if last-element8
                                     (set-cdr! last-element8 c)
                                     (set! list8 c))
                                 (set! last-element8 c))))
            (get-list8 (lambda () list8)))
       (list-maker-thunk8 cons!8 get-list8)
       ,@(if return-thunk-rv? '() '(list8)))))

; (with-reverse-list-maker (cons! 1) (cons! 2)) => '(2 1)
; (Making this fully as a macro, in order to ensure that the code always will get max speed.)
(define-macro (with-reverse-list-maker . code) ; #!key return-thunk-rv? #!rest code) - bigloo can't do this.
  (let* ((return-thunk-rv?-set? (and (not (null? code))
                                     (eq? 'return-thunk-rv?: (car code))))
         (return-thunk-rv?      (and return-thunk-rv?-set?
                                     (cadr code)))
         (code                  (if return-thunk-rv?-set? (cddr code) code)))
    `(let* ((list-maker-thunk (lambda (cons! get-list) ,@code))
            (list '())
            (cons! (lambda (v)
                     (set! list (cons v list))))
            (get-list (lambda () list)))
       (list-maker-thunk cons! get-list)
       ,@(if return-thunk-rv? '() '(list)))))


(define-macro (1+ v) `(+ ,v 1))
(define-macro (1- v) `(- ,v 1))
```

```scheme
(define-macro (let-args list args . code)
  ; (print "let-args invoked: args=") (write args) (print " list=") (pp list) (print " code=") (pp code)
  (let ((r
         ; Any of #!key #!rest in args?
         (if (let loop ((rest args))
               (cond ((or (null? rest) (symbol? rest))
                      #t) ; TODO: =ALWAYS TAKE THE PATH VIA (apply). SET TO #f WHEN BIGLOO SIGSEGV BUG IS FIXED.
                     ((eq? (car rest) #!key) ; (memq (car rest) '(#!key #!optional #!rest))
                      #t)
                     (else (loop (cdr rest)))))
             ; Yes. Use (apply) to deliver.
             `(apply (lambda ,args
                       ,@code)
                     ,list)
             ; No. Load list into local namespace, to deliver
             `(let* ,@(let ((orig-list-in-var '**let-args-temp-orig)) ; (gensym)))
                        (let loop ((result `((,orig-list-in-var ,list))) (rest-args args) (list-in-var orig-list-in-var)
                                   (expected-length 1) (in-optionals #f) (has-optionals #f))
                          (if (null? rest-args)
                              (let ((r (reverse result)))
                                (if list-in-var ; list-in-var is #f for args with #!rest / . .
                                    `(,r
                                  (if ,(if has-optionals
                                           ; If has-optionals, then list-in-var may either be '() (because all elements
                                           ; of the input list have been processed and now we're at '()), or #f, because
                                           ; more than all elements of the input list have been processed, per below.
                                               `(and ,list-in-var (not (null? ,list-in-var)))
                                               `(not (null? ,list-in-var)))
                                           (error
                                            ; To fit with Bigloo's (error) calling convention (proc msg obj):
                                            "Invalid length" #f
                                            (vector
                                             ,@(if has-optionals
                                                   '() ; If has optionals then skip reporting expected length.
                                                   ; Alternatively, we could report the interval of expected length.
                                                   `(,expected-length))
                                             ,orig-list-in-var
                                             ,list-in-var
                                             ; For really tough debugging the following could be applied:
                                             ',args ',list ',code
                                             )))
                                       (let () ,@code))
```

```
      `(,r ,@code)))


(if (symbol? rest-args) ; handle (arg0 arg arg2 . THIS)
    (loop (cons `(,rest-args ,list-in-var) result)
          '() #f #f #f has-optionals)
    (let ((c-arg (car rest-args)) (rest-args (cdr rest-args))
          (list-to-var '**let-args-temp)) ; (gensym)))
      (case c-arg
        ((#!optional)
         (loop result rest-args list-in-var expected-length #t #t))
        ((#!rest)
         (if (not (and (list? rest-args) (eq? 1 (length rest-args))))
             (error "Invalid #!rest - not only one more list element after it"
                    args list code))
         (loop (cons `(,(car rest-args) (or ,list-in-var '())) result)
               '() #f #f #f has-optionals))
        (else
         (loop
          (if in-optionals
              (cons `(,list-to-var (if **let-args-list-has-more?
                                       (cdr ; ** In Gambit this one can be replaced with ##cdr
                                        ,list-in-var)
                                             #f))
                    (cons `(,(if (pair? c-arg) (car c-arg) c-arg)
                             (if **let-args-list-has-more?
                                 (car ,list-in-var)
                                 ,(if (pair? c-arg)
                                      (begin
                          (if (not (eq? 2 (length c-arg)))
                             (error "Invalid #!optional argument default value" c-arg args list code))
                          (cadr c-arg))
                                      #f)))
                          (cons `(**let-args-list-has-more? (pair? ,list-in-var))
                                result)))
```

```
                                        (begin
                                      (if (not (symbol? c-arg)) (error "Invalid argument symbol" c-arg args list code))

                                      (cons `(,list-to-var (cdr ; ** In Gambit this one can be replaced with ##cdr
                                                                ,list-in-var))
                                                (cons `(,c-arg (car ,list-in-var))
                                                      ; Normally:
                                                      result
                                                      ; Extra type checking (valuable in bigloo?):
                                                      ; (cons `(**let-args-void (if (not (pair? ,list-in-var))
                                                      ;                             (error "Bad list passed" #f
                                                      ;                                    ,orig-list-in-var)))
                                                      ;       result)
                                                      ))))
                                         rest-args
                                         list-to-var
                                         (+ expected-length 1)
                                         in-optionals
                                         has-optionals)))))))))))
; (print "(let-args " ) (write list) (print " ") (write args) (print " ") (write code) (print ") returns: ") (pp r)
r))
```

END of Apendix B

Appendix C

Model Description

of the linearly programmed long-term energy systems cost-minimizing model

GET-RC 6.1

that generates the fuel and technology mix that meets the energy demand, subject to the constraints, at lowest global energy system cost.

Maria Grahn
Department of Energy and Environment, Physical Resource Theory, Chalmers University of Technology, 412 96 Göteborg, Sweden, Email: maria.grahn@chalmers.se

Erica Klampfl, Margaret J. Whalen, Timothy J. Wallington
Systems Analytics and Environmental Sciences Department, Ford Motor Company, Mail Drop RIC-2122, Dearborn, MI 48121-2053, USA.

Kristian Lindgren
Department of Energy and Environment, Physical Resource Theory, Chalmers University of Technology, 412 96 Göteborg, Sweden

November 2012

Keywords: GET model, Linear Programming, Sustainable mobility, Carbon emissions, Energy scenarios

2. Sets

In this section the different sets (indices) are presented. For a compact description of the sets in alphabetic order, see Appendix 2.

2.1 Time

The model's main time period is 1990-2140 divided in 10 year steps. Results are presented for the 2010-2100 period. The set "t_h" include historical time steps used for the carbon cycle calculations and for plotting a long-term figure of historical emissions combined with model results. In GAMS, it is sometimes necessary to have more than one name for the same set, e.g. when emissions one year affect emission concentration another year. Here the set "T_all_copy" includes all time step and is identical to "T_all". The timeset "t_2010_2140" is used when fixing a specific value for the result from year 2010 and beyond, e.g. the use of nuclear.

T_all	1800, 1810, 1820, 1830, 1840, 1850, 1860, 1870, 1880, 1890, 1900, 1910, 1920, 1930, 1940, 1950, 1960, 1970, 1980, 1990, 2000, 2010, 2020, 2030, 2040, 2050, 2060, 2070, 2080, 2090, 2100, 2110, 2120, 2130, 2140
T_all_copy $\subseteq$ T_all	1800, 1810, 1820, 1830, 1840, 1850, 1860, 1870, 1880, 1890, 1900, 1910, 1920, 1930, 1940, 1950, 1960, 1970, 1980, 1990, 2000, 2010, 2020, 2030, 2040, 2050, 2060, 2070, 2080, 2090, 2100, 2110, 2120, 2130, 2140
t_h $\subseteq$ T_all	1800, 1810, 1820, 1830, 1840, 1850, 1860, 1870, 1880, 1890, 1900, 1910, 1920, 1930, 1940, 1950, 1960, 1970, 1980
t $\subseteq$ T_all	1990, 2000, 2010, 2020, 2030, 2040, 2050, 2060, 2070, 2080, 2090, 2100, 2110, 2120, 2130, 2140
t_2010_2140 $\subseteq$ t	2010, 2020, 2030, 2040, 2050, 2060, 2070, 2080, 2090, 2100, 2110, 2120, 2130, 2140
init_year $\subseteq$ t	1990

2.2 Energy supply

Here are the sets for the model's energy options both primary energy sources, which contains the ingoing sources used in the energy conversion plants (e_in), and the secondary energy carriers, which are the final energy products, i.e., energy carriers coming out from the energy conversion plants (e_out), all listed in the set E.

The set of options that are allowed to enter the energy conversion module as incoming energy (e_in) has the following acronyms. Primary energy options are: natural gas (NG), oil (OIL), coal (COAL), nuclear (NUCLEAR), biomass (BIO), hydro power (HYDRO), wind power (WIND), concentrating solar power (SOLAR_CSP), and finally other solar energy technologies, i.e. solar-PV, solar-heat and solar-hydrogen (SOLAR). The energy carriers that can be converted a second time are: hydrogen (H2) and electricity (ELEC), i.e. electricity can be converted to heat or hydrogen and hydrogen can be converted to heat or electricity. In this model version, the technology CSP can be viewed as a new energy technology generating inexpensive electricity with low CO_2 emissions, and can therefore act as a proxy for any future inexpensive electricity with low CO_2 emissions, e.g., advanced fission, fusion, wave energy, geothermal energy.

The set of energy carriers, converted from primary energy sources, (e_out) has the following acronyms: all stationary use that not are electricity nor transportation fuels, e.g., industrial process heat, district heating and feedstock (HEAT), electricity (ELEC), biomass-to-liquid, which is biomass based synthetic fuels assuming cost assumptions from bio-based methanol via gasification (BTL), coal-to-liquid and gas-to-liquid, both using cost assumptions from methanol production (CTL/GTL), hydrogen (H2), natural gas (NG), petroleum based gasoline/diesel/kerosene (PETRO), synthetic fuels for aviation (AIR_FUEL).

E	bio, hydro, wind, solar, solar_CSP, NG, oil, coal, nuclear, BTL, CTL/GTL, H2, heat, elec, petro, air_fuel
e_in $\subseteq$ E	bio, hydro, wind, solar, solar_CSP, NG, oil, coal, nuclear, H2, elec
e_out $\subseteq$ E	heat, elec, BTL, CTL/GTL, H2, NG, petro, air_fuel

The following are different subsets, of the energy sources and carriers, used in some equations. Acronyms used here are co-generation of heat and electricity from the same conversion process (CG).

cg_e_in $\subseteq$ e_in	bio, NG, oil, coal, H2
cg_e_out $\subseteq$ e_out	elec, BTL, CTL/GTL, H2
primary $\subseteq$ e_in	bio, hydro, wind, solar, solar_CSP, NG, oil, coal, nuclear
second_in $\subseteq$ e_in	H2, elec
sec $\subseteq$ e_out	BTL, CTL/GTL, H2
nontrade_sec $\subseteq$ e_out	heat, elec, NG, petro, air_fuel
fuels $\subseteq$ primary	bio, coal, oil, NG, nuclear
nonfuels $\subseteq$ primary	hydro, wind, solar, solar_CSP
fossil $\subseteq$ fuels	coal, oil, NG

2.3 Type of conversion plant

The energy conversion plants can be of different types. In this model, energy conversion plants can use conventional technology (0) or co-generation plants where both electricity and heat are produced (cg) or plants where the plants have included carbon capture and storage technologies (CCS). The subset c_capt includes the two plant type options that can capture carbon, either with or without co-generation of electricity and heat. The subset CG_type includes the two plant type options that produce co-generated heat and electricity either with or without CCS.

type	0, cg, CCS, cg_CCS
c_capt $\subseteq$ type	CCS, cg_CCS
CG_type $\subseteq$ type	cg, cg_CCS

2.4 Fuels for transport

There are several fuel options that can be used in the transportation sector, i.e. biomass-based liquid fuels (BTL), coal to liquid (CTL), gas to liquid (GTL), petroleum-based fuels such as gasoline, diesel and kerosene (PETRO), electricity (ELEC), hydrogen (H2), natural gas (NG), and synthetic fuels for aviation (AIR_FUEL). The different subsets are used in equations only valid for some specific fuels.

trsp_fuel $\subseteq$ e_out	BTL, CTL/GTL, petro, elec, H2, NG, air_fuel
trsp_fuel_nonel $\subseteq$ trsp_fuel	BTL, CTL/GTL, petro, H2, NG, air_fuel
synfuel_gas $\subseteq$ trsp_fuel_nonel	BTL, CTL/GTL, H2, NG
road_fuel $\subseteq$ trsp_fuel	BTL, CTL/GTL, petro, elec, H2, NG
road_fuel_liquid $\subseteq$ road_fuel	BTL, CTL/GTL, petro

2.5 Vehicle technologies

There are five different vehicle technologies (e-type) available in the model, i.e., conventional internal combustion engine vehicles (0), fuel cell vehicles (FC), hybrid electric vehicles (HEV), plug-in hybrid electric vehicles (PHEV), and battery electric vehicles (BEV). The different subsets are declared to be used in equations only valid for some specific vehicle technologies.

e_type	0, FC, HEV, PHEV, BEV
ic_fc $\subseteq$ e_type	0, FC
hybrids $\subseteq$ e_type	HEV, PHEV
hev_phev_bev $\subseteq$ e_type	HEV, PHEV, BEV
non_phev $\subseteq$ e_type	0, FC, HEV, BEV

2.6 Transport modes

The energy demand in the transportation sector is divided between nine different transport modes (trsp_mode), i.e., light duty passenger vehicles (p_car), airplanes for passenger travel (p_air), buses (p_bus), passenger rail (p_rail), freight road, i.e. trucks (f_road), freight aviation (f_air), freight coastal shipping (f_sea), and freight international shipping (f_isea).

trsp_mode	p_car, p_air, p_bus, p_rail, f_road, f_air, f_sea, f_isea, f_rail
vehicle $\subseteq$ trsp_mode	p_car, f_road
ptrs_mode $\subseteq$ trsp_mode	p_car, p_air, p_bus, p_rail
frgt_mode $\subseteq$ trsp_mode	f_road, f_air, f_sea, f_isea, f_rail
ship_mode $\subseteq$ trsp_mode	f_sea, f_isea

2.7 Regions

In GET-RC 6.1, the world is treated as 10 distinct regions: North America (NAM), Europe (EUR), the Former Soviet Union (FSU), OECD countries in the Pacific Ocean (PAO), Latin America (LAM), the Middle East (MEA), Africa (AFR), Centrally Planned Asia – mainly China (CPA), South Asia – mainly India (SAS) and Pacific Asia (PAS). All regions can export and import and the costminimizing determine the trade. In GAMS, it is sometimes necessary to have more than one name for the same set, e.g. when energy carriers are traded between two regions. Here the set “R_exp” includes all regions that can export which is identical to “R_imp” which is the regions that can import.

R	NAM, EUR, PAO, FSU, AFR, PAS, LAM, MEA, CPA, SAS
R_exp $\subseteq$ R	NAM, EUR, PAO, FSU, AFR, PAS, LAM, MEA, CPA, SAS
R_imp $\subseteq$ R	NAM, EUR, PAO, FSU, AFR, PAS, LAM, MEA, CPA, SAS

3. Scalars and parameters

In this section we present the names of all used scalars and parameters (given data). The chosen data values can be found in Appendix 1.

3.1 Scalars

In this Section the name of the scalars are presented. We define scalars as a parameter with one specific value only. As soon as the parameter depends on one or more sets they are presented as a parameter, see Section Parameters. In this report we always present scalars and parameters in blue text.

3.1.1 Basic scalars

The scalars presented here are the amount of million seconds per year (Msec_per year), which together with a plant specific capacity factor, is used when converting from an energy conversion plant's effect expressed in kW into the amount of energy that comes out from the plant (GJ/yr). The discount rate (r) counts for the valuation of future costs and is in the base case set to 5%. Also the interest rate applied to investments (r_invest), is set to 5% in the base case runs.

Msec_per_year	= 31.6	Number of seconds per year expressed in millions	[Ms]
t_step	= 10	Number of year within each time step	[yr]
r	= 0.05	Discount rate	[-]
r_invest	= 0.05	Interest rate applied to investments	[-]
pre_ind_ccont	= 280	Pre-industrial atmospheric CO_2 concentration	[ppm]

3.1.2 Maximum growth and depreciation

Constraints have been added to the model to avoid solutions that are obviously unrealistic, primarily constraints on how fast changes can be made in the energy system. This includes constraints on the maximum expansion rates of new technologies (in general set so that it takes 50 years to change the entire energy system) as well as annual or total extraction limits on the different available energy sources. The growth is limited by both relative and absolute values. Here are first the relative values, where all limitations on growth from one timestep to another are maximized to 20% in base case runs. The depreciation, limitations the minimum fraction remaining in a timestep (compared to previous timestep) is set to 75% for a certain energy technology, 70% for a certain vehicle technology and minimum 20% for the phase out of oil.

cap_g_lim	= 0.2	maximum growth of capacity in energy conversion plants	[-]
supply_g_lim	= 0.2	maximum growth of primary energy extraction	[-]
infra_g_lim	= 0.2	maximum growth of infrastructure capacity	[-]
eng_g_lim	= 0.2	maximum growth of vehicle technologies	[-]
en_conv_decr_lim	= 0.75	minimum fraction of previous timestep's energy technology	[-]
mx_decay_frac	= 0.7	minimum fraction of previous timestep's vehicle technology	[-]
mx_decay_frac_oil	= 0.2	minimum fraction of previous timestep's oil use	[-]

The following scalars present the absolute values. Some scalars are first defined as a global static value and then regionalized and made dynamic in calculations included in the model, see Section 3.2.4 Growth and depreciation. The global values are presented here since they only consist of one value (scalars).

global_max_exp_p	= 2	Global maximum expansion of conversion plants	[TW/decade]
global_max_exp	= 60	Global maximum expansion for energy sources except biomass	[EJ/decade]
global_max_exp_b	= 32	Global maximum expansion for biomass production	[EJ/decade]
global_max_exp_i	=1	Global maximum expansion for infrastructure investments	[TW/decade]
global_init_i	= 0.05	"kick-start" value when a new infrastructure is introduced	[TW]
global_init_e	= 0.1	"kick-start" value when a new engine is introduced	[Gvehicles]
global_init_p	= 0.3	"kick-start" value when a new conversion plant is introduced	[TW]
global_init_s	= 0.3	"kick-start" value when a new energy source is introduced	[EJ]
global_en_conv_dis	= 5	The final "tail" of a certain energy conversion	[EJ]
global_mx_decay	= 6	The final "tail" of a certain transportation energy	[EJ]
global_mx_decay_oil	=10	The final "tail" of oil use in primary energy values	[EJ]
t_tech_plant	=50	Inertia. How fast new conversion technologies may totally change	[yr]
t_tech_eng	=50	Inertia. How fast new engine technologies may totally change	[yr]
t_tech_effic	=30	Inertia. How fast energy efficiency may totally change	[yr]

3.1.3 Energy conversion

Scalars used in energy conversion equations including intermittency limitations, co-generation and CCS.

fos_capt_effic	= 0.9	carbon capture efficiency from fossil CCS	[-]
bio_capt_effic	= 0.9	carbon capture efficiency from bioenergy CCS	[-]
c_capt_heat_fr	= 0.3	max fraction of heat sector using CCS	[-]
c_stor_maxgr	= 100	global annual growth limit on carbon storage capacity	[MtC/decade]
cogen_fr_e	= 0.2	max fraction of electricity demand from co-generation	[-]
cogen_fr_h	= 0.2	max fraction of heat demand that can come from co-generation	[-]
interm_fr	= 0.3	max fraction of intermittent electricity (wind + solar-elec)	[-]

3.1.4 Transport

Scalars used in the transportation module. We assume that maximum 20% of all trucks and 50% of all buses can run on electricity as plugin-hybrids (PHEVs) or as pure battery electric vehicles (BEVs).

frac_phev_trucks	= 0.2	share of trucks that can use PHEV	[-]
frac_phev_buses	= 0.5	share of buses that can use PHEV	[-]
frac_bev_trucks	= 0.2	share of trucks that can use BEV	[-]
frac_bev_buses	= 0.5	share of buses that can use BEV	[-]

3.1.5 Cost

Scalars used when calculating costs. The reason for that the storage cost differ between carbon from fossil fuels and carbon from bioenergy is that bioenergy conversion plants typically are smaller in size compared to fossil fuel conversion plants. The cost for distributing the carbon from larger conversion plants will benefit from the economics of scale.

cost_strg_fos	= 0.037	carbon storage cost from fossils (equivalent to 10 USD/t CO_2)	[GUSD/MtC]
cost_strg_bio	= 0.073	carbon storage cost from bioenergy (equivalent to 20 USD/t CO_2)	[GUSD/MtC]
c_bio_trspcost	= 0.5	additional transportation cost applied to bioenergy CCS	[GUSD/EJ]

3.2 Parameters

In this section all parameters (given data) used in the model are presented and in parenthesis it is shown what sets the parameter values depend on. Chosen data for the base case runs, in the model, is presented in Appendix 1.

3.2.1 Supply potential and energy demand

supply_pot (primary, R, t)	Annual upper limit on supply potential (non-fossil sources)	[EJ]
supply_pot_0 (primary, R)	Aggregated upper limit on fossil supply potential.	[EJ]
heat_dem_reg (R, t)	Heat demand	[EJ]
elec_dem_reg (R, t)	Electricity demand	[EJ]
ptrsp (R, ptrs_mode, t)	Energy demand for passenger transport	[EJ]
frgt (R, frgt_mode, t)	Energy demand for freight transport	[EJ]
trsp_dem (R, trsp_mode, t)	Energy demand for each transportation mode	[EJ]

The input data for energy demand for the transportation sector, parameter trsp_dem, is presented in two tables in Appendix 1, where Table ptrsp_all include demand for passenger transport modes and Table frgt_all include demand for the freight transport modes. The input data in earlier model versions assume an overall energy efficieny of 0.7% per year. In this model version we assume that energy savings of

0.3% per year can be achieved through improved rolling and air resitance as well as eco-diving, which is assumed to be equal for all types of road and sea based vehicle and ship technologies. The annual improvement on drivetrains are, however, assumed to be technology dependent, i.e., higher for internal combustion engines and fuel cell engines compared to electric vehicles. Therefore the input data in parameter trsp_dem on p_car, p_bus, f_road, f_sea, f_isea first need to be adjusted by a factor of 1.004, which in GAMS are made by the following code:

```
trsp_dem (R, "p_rail", t) = ptrsp_all(R, "p_rail", t);
trsp_dem (R, "p_air", t) = ptrsp_all(R, "p_air", t);
trsp_dem (R, "p_car", t) = ptrsp_all(R, "p_car", t)*1.004**(t_step*(ord(t)-1));
trsp_dem (R, "p_bus", t) = ptrsp_all(R, "p_bus", t)*1.004**(t_step*(ord(t)-1));
trsp_dem (R, "f_rail", t) = frgt_all(R, "f_rail", t);
trsp_dem (R, "f_air", t) = frgt_all(R, "f_air", t);
trsp_dem (R, "f_road", t) = frgt_all(R, "f_road", t)*1.004**(t_step*(ord(t)-1));
trsp_dem (R, "f_sea", t) = frgt_all(R, "f_sea", t)*1.004**(t_step*(ord(t)-1));
trsp_dem (R, "f_isea", t) = frgt_all(R, "f_isea", t)*1.004**(t_step*(ord(t)-1));
```

Note that “ORD” is a GAMS operator which generates integers from the position in the set, e.g. ORD(1990)=1 and ORD(2000)=2.

The following expression changes all static values on supply potential (presented in Table “supply_pot_0”) time dependent. That means all values presented in the table will be the upper limit for each time step. This will of course lead to unrealisticly high annual upper limits on fossil sources but equation (2) will correct for that.

supply_pot (primary, R, t) = supply_pot_0 (primary, R); [EJ]

3.2.2 Energy conversion plants, CCS and infrastructure

effic (e_in, type, e_out, t)	Time dependent energy conversion efficiency	[-]
effic_current (e_in, type, e_out)	Near term energy conversion efficiency	[-]
effic_0 (e_in, type, e_out)	Ideal conversion efficiency assumed available in 2020-2050	[-]
heat_effic (cg_e_in, cg_type, cg_e_out)	Heat efficiency when cogeneration of electricity and heat	[-]
lf (e_in, type, e_out, R)	Load factor (capacity factor) for energy conversion plants, i.e., the share of maximum capacity that is used per year	[-]
lf_infra (synfuel_gas)	Load factor infrastructure	[-]
life_plant (e_in, e_out, type)	Life time on energy conversion plants	[yr]
life_infra (synfuel_gas)	Life time for infrastructure	[yr]
dec_elec (e_in)	Electricity requirements when using CCS (fraction of en_conv)	[-]
init_cap (e_in, e_out, type, R)	Capacity in energy conversion plants for the initial year	[TW]

Calculation of some of the parameters listed above

Energy conversion efficiency in near term “effic_current” is assumed to be 0.1 lower than the ideal energy efficiency “effic_0”. It is of cource not possible to know exacly when in time the ideal efficiency can become reality. However we make an assumption that it can be fulfilled sometimes between 2020 and 2050.

effic_current (e_in, type, e_out) = effic_0 (e_in, type, e_out) – 0.1

In this model version we use time dependent energy efficiency. The model chooses the lowest value on “effic” from two linear functions, in the expression below, separated by comma. Values for effic will therefore be lower the first timesteps and increase during time until the two curves cross each other. From the timestep where they cross the values on effic will be stabilized at a so called mature level.

effic (e_in, type, e_out, t) =
min (effic_0 (e_in, type, e_out), (effic_0 (e_in, type, e_out) – effic_current (e_in, type, e_out))/
t_tech_effic (e_in, type, e_out)*((ord(t)-1)* t_step) + effic_current (e_in, type, e_out))

3.2.3 Transportation sector

num_veh (R, vehicle, t)	Number of vehicles	[Gvehicles]
life_eng (trsp_fuel, e_type, vehicle)	Life time on vehicle engines	[yr]
trsp_conv_st (trsp_fuel, e_type, trsp_mode)	A factor that relates the energy efficiency to conventional ICEV. Valid for aviation and rail.	[-]
trsp_conv (trsp_fuel, e_type, trsp_mode, t)	Time dependent trsp-conv	[-]
elec_frac_phev (vehicle)	Fraction of time that a PHEV operates in battery mode. We assume that BTL/CTL/GTL and Petro PHEVs have the same electricity fraction.	[-]
high_speed_train (R, t)	Fraction of aviation sector substituted with high speed trains run on electricity	[-]

3.2.4 Growth and depreciation

max_exp_p (e_in, e_out, type, R, t)	Growth limit on energy conversion plants	[TW/decade]
max_exp (R, t)	Primary fuel supply growth limit	[EJ/decade]
max_exp_bio (R, t)	Bio energy growth limit	[EJ/decade]
max_inv_infra (R, t)	Infrastructure growth limit	[TW/decade]
en_conv_dis (R, t)	The final "tail" of a certain energy conversion	[EJ]
mx_decay_abs (R, t)	The final "tail" of a certain transportation energy	[EJ]
mx_decay_abs_oil (R, t)	The final "tail" of oil use in primary energy values	[EJ]
init_infra (R)	“kick-start” value when a new infrastructure is introduced	[TW]
init_eng (R)	“kick-start” value when a new engine is introduced	[Gvehicles]
init_plant (R)	“kick-start” value when a new conversion plant is introduced	[TW]
init_supply (R)	“kick-start” value when a new energy source is introduced	[EJ]

Calculations of regionalized growth and depreciation parameters listed above
Calculation of regionalized parameters using the global scalars presented in Section Scalars. The scalars are transferred into dynamic parameters, i.e., changed into time dependent values depending on the regional energy demand in each time step. Note that the global maximum expansion values were presented in TW whereas the regionalized values are presented in EJ.

max_exp (R, t) =

$$\left(\left(\text{elec\_dem\_reg}(R,t) + \text{heat\_dem\_reg}(R,t) + \Sigma_{\text{trsp\_mode}}\left(\text{trsp\_dem}(R,\text{trsp\_mode},t)\right)\right) / \left(\text{Msec\_per\_year}*\text{t\_step}\right)\right)* \text{global\_max\_exp};$$

max_exp_p (e_in, e_out, type, R, t) =

$$\left(\left(\text{elec\_dem\_reg}(R,t) + \text{heat\_dem\_reg}(R,t) + \Sigma_{\text{trsp\_mode}}\left(\text{trsp\_dem}(R,\text{trsp\_mode},t)\right)\right) / \left(\text{Msec\_per\_year}*\text{t\_step}\right)\right)* \text{global\_max\_exp\_p};$$

max_exp_bio (R, t) =

$$\left(\left(\text{elec\_dem\_reg}(R,t) + \text{heat\_dem\_reg}(R,t) + \Sigma_{\text{trsp\_mode}}\left(\text{trsp\_dem}(R,\text{trsp\_mode},t)\right)\right) / \left(\text{Msec\_per\_year}*\text{t\_step}\right)\right)* \text{global\_max\_exp\_b};$$

max_inv_infra (R, t) =

$$\left(\left(\text{elec\_dem\_reg}(R,t) + \text{heat\_dem\_reg}(R,t) + \Sigma_{\text{trsp\_mode}}\left(\text{trsp\_dem}(R,\text{trsp\_mode},t)\right)\right) / \left(\text{Msec\_per\_year}*\text{t\_step}\right)\right)* \text{global\_max\_exp\_i};$$

init_eng (R) =

$$\left(\left(\text{elec\_dem\_reg}(R,\text{"2000"}) + \text{heat\_dem\_reg}(R,\text{"2000"}) + \Sigma_{\text{trsp\_mode}}\left(\text{trsp\_dem}(R,\text{trsp\_mode},\text{“2000”})\right)\right) / \left(\text{Msec\_per\_year}*\text{t\_step}\right)\right)* \text{global\_init\_e};$$

init_infra (R) =
$\Big(\big($elec_dem_reg (R,"2000") + heat_dem_reg (R,"2000") + $\Sigma_{\text{trsp\_mode}}$ $\big($trsp_dem (R, trsp_mode, ”2000”)$\big)\big)$ / $\big($Msec_per_year*t_step$\big)\Big)$* global_init_i;

init_plant (R) =
$\Big(\big($elec_dem_reg (R,"2000") + heat_dem_reg (R,"2000") + $\Sigma_{\text{trsp\_mode}}$ $\big($trsp_dem (R, trsp_mode, ”2000”)$\big)\big)$ / $\big($Msec_per_year*t_step$\big)\Big)$* global_init_p;

init_supply (R) =
$\Big(\big($elec_dem_reg (R,"2000")+ heat_dem_reg (R,"2000")+ $\Sigma_{\text{trsp\_mode}}$ $\big($trsp_dem (R, trsp_mode, ”2000”)$\big)\big)$ / $\big($Msec_per_year*t_step$\big)\Big)$* global_init_s;

As an alternative to the four latter calculations above, the values can instead be inserted as an input data table with the values presented in Table 1.

Table 1. Regional values for the four parameters acting as “kick-start” values when a new tecnology is introduced in model scenarios. These regional values can be used instead of the calculations where a regional value is generated from a global value.

	init_eng (R)	init_infra (R)	init_plant (R)	init_supply (R)
NAM	0.025	0.013	0.076	0.076
EUR	0.018	0.009	0.054	0.054
PAO	0.008	0.004	0.023	0.023
FSU	0.008	0.004	0.024	0.024
AFR	0.005	0.003	0.015	0.015
PAS	0.004	0.002	0.013	0.013
LAM	0.005	0.002	0.015	0.015
MEA	0.004	0.002	0.011	0.011
CPA	0.011	0.005	0.033	0.033
SAS	0.006	0.003	0.019	0.019

en_conv_dis (R, t) =
$\Big(\big($elec_dem_reg (R, t) + heat_dem_reg (R, t) + $\Sigma_{\text{trsp\_mode}}$ $\big($trsp_dem (R, trsp_mode, t)$\big)\big)$ / $\big($Msec_per_year*t_step)$\Big)$* global_en_conv_dis;

mx_decay_abs (R, t) =
$\Big(\big($elec_dem_reg (R, t)+ heat_dem_reg (R, t)+ $\Sigma_{\text{trsp\_mode}}$ $\big($trsp_dem (R, trsp_mode, t)$\big)\big)$ / $\big($Msec_per_year*t_step$\big)\Big)$* global_mx_decay;

mx_decay_abs_oil (R, t) =
$\Big(\big($elec_dem_reg (R, t)+ heat_dem_reg (R, t)+ $\Sigma_{\text{trsp\_mode}}$ $\big($trsp_dem (R, trsp_mode, t)$\big)\big)$ / $\big($Msec_per_year*t_step$\big)\Big)$* global_mx_decay_oil;

3.2.5 Emissions and carbon taxes

CO_2 emissions are registered whenever fossil primary energy sources are used in the model. The emissions differ between the fossil energy sources depending on the emission factor (emis_fact). An emission factor is also given for biomass since carbon can be stored by using carbon capture and storage technology also on biomass. Since it is possible to import and export fossil synthetic fuels the emissions from the production of CTL/GTL is registered on the region before exporting the fuel and the emissions from combusting the fuel is registered in the region using the fuel. A special emission factor is therefore needed for the combustion of synthetic fuels (emis_fact_syn).

To convert the CO_2 emissions into atmotsperic CO_2 concentration a carbon cycle module is included. Parameters used in the carbon cycle module are also presented here.

emis_fact (fuels)	Carbon dioxide emission factors	[MtC/EJ]
emis_fact_syn	Carbon dioxide emission factor from the use of CTL/GTL	[MtC/EJ]
c_tax (R, t)	Carbon tax	[GUSD/MtC]
hist_fos_emis (t_h)	Historical fossil emissions	[MtC]
fut_luc_emis (T_all)	Prognos for future emissions from land use change	[MtC]
fut_biota_sinks (T_all)	Estimation of the contribution of natural CO_2 sinks	[MtC]
IRfunc (T_all, T_all_copy)	Impulse Response function. Determines annual contribution to atmospheric carbon from each year's CO_2 emissions.	[-]

The impulse response function, IRfunc, is a declining function representing the annual contribution to atmospheric CO_2 from each emission impulse. The contribution is highest directly after the emission but will contribute to the atmospheric CO_2 for many decades after it is emitted. The impulse response function used in GET is taken from Maier-Reimer and Hasselmann (1987) and the function is named G(t) in their paper. Note that if the values of IRfunc is included to model as a table with data (and not calculated during the run) the set coeff as well as the parameters A (coeff) and tao (coeff) can be excluded. More details on IRfunc can be found in Appendix 3. The calculated values can for the paremeter IRfunc can be found in Appendix 1.

3.2.6 Prices and costs

Here we present parameter used when calculating costs in the model. Some costs are calculated in the model from a starting base cost. The cost is thereafter made time dependent and modified by interest rate and discount rate. A base cost for a theoretical standard vehicle run on petroleum based fuel (for cars a gasoline TDI) is first defined (vehicle_cost). For more advanced vehicle technology and fuel options we add an incremental cost relative to the conventional vehicle, see Appendix 1 for chosen data. The investment costs are then modified with an interest rate and a discount rate (cost_inv_mod).

price (fuels, R)	Basic fuel price without scarcity rent or carbon tax	[GUSD/EJ]
OM_cost_fr (e_in, e_out)	Operation & maintenance cost as fraction of capacity cost	[-]
cost_inv_base (e_in, type, e_out)	Investments cost for energy conversion plants	[GUSD/TW]
cost_inv (e_in, e_out, type, t)	Time is added to the plant investment cost	[GUSD/TW]
cost_inv_mod (e_in, e_out, type, t)	Plant investment cost adjusted if assuming different investment interest rate and discount rates	[GUSD/TW]
vehicle_cost (vehicle)	Basic cost for vehicle with gasoline IC engine	[GUSD/Gvehicle]
cost_eng_base (road_fuel, e_type, vehicle)	Additional cost above standard "petro-vehicle"	[GUSD/Gvehicle]
cost_eng (road_fuel, e_type, vehicle, t)	Time dependent vehicle investment cost	[GUSD/Gvehicle]
cost_eng_mod (road_fuel, e_type, vehicle, t)	Vehicle investment cost adjusted if assuming different investment interest rate and discount rates	[GUSD/Gvehicle]
cost_infra (synfuel_gas)	Investment cost for infrastructure	[GUSD/TW]
cost_infra_mod (synfuel_gas)	Investment cost for infrastructure adjusted if assuming different investments and discount rates	[GUSD/TW]
imp_cost (fuels)	Cost for transportation of primary energy sources when trading between regions, regardless the distance.	[GUSD/EJ]
imp_cost2 (sec)	Cost for transportation of secondary energy carriers when trading between regions, regardless the distance.	[GUSD/EJ]
imp_cost_lin (fuels)	Additional cost, dependent on distance, for transportation wen trading primary energy sources.	[GUSD/EJ]
imp_cost_lin2 (sec)	Additional cost, dependent on distance, for transportation when trading secondary energy carriers.	[GUSD/EJ]

Calculation of some parameters listed above

Time is added as a fourth dimension to the investment cost of energy conversion plants and vehicles. Originally we planned for introducing time dependent investment costs but in this model version we use identical investment costs for each time step.

cost_inv (e_in, e_out, type, t) = cost_inv_base (e_in, type, e_out);

cost_eng (road_fuel, e_type, vehicle, t) = cost_eng_base (road_fuel, e_type, vehicle) + vehicle_cost (vehicle);

Calculation of modified costs, if the discount rate (r) and investment interest rate (r_invest) are assumed different. In this model version we have set the r and r_invest to the same value, i.e. both set to 0.05. These three calculations below are therefore currently not contributing to the result but useful when making different tests on the effect of assuming that the discount rate and investment rates are not identical.

cost_inv_mod (e_in, e_out, type, t) =
cost_inv (e_in, e_out, type, t) * (r_invest + 1/life_plant (e_in, e_out, type))/(r+1/life_plant (e_in, e_out, type));

cost_eng_mod (road_fuel, e_type, vehicle, t) =
cost_eng (road_fuel, e_type, vehicle, t) * (r_invest + 1/life_eng (road_fuel, e_type, vehicle))/
(r+1/life_eng (road_fuel, e_type, vehicle));

cost_infra_mod (synfuel_gas) =
cost_infra (synfuel_gas) * (r_invest + 1/life_infra (synfuel_gas)) / (r + 1/life_infra (synfuel_gas));

3.2.7 Miscellaneous

distance (R_imp, R_exp)	Table of rough distances between regions	[km]
flow_matrix (e_in, type, e_out)	Table presenting which energy conversions that are allowed	[-]
population (R, t)	Population	[Gpeople]

The table flow_matrix is a matrix containing “0” or “1” depending on if an energy conversion, from a primary energy source to an energy carrier, is allowed or not.

4. Variables

In this section the variables are presented with a short explanation including units. Variables written in red indicate main variables, i.e. that the variables are not calculated from any other decision variables. All variables are of type continuous. For a compact description of the variables in alphabetic order, see Appendix 2.

4.1 Balancing energy flows

The energy flowes in the model are simplified in many ways, but captures the most important flows in a real global energy system, see Figure 1.

4.1.1 Energy conversion

The entire energy flow, from primary energy extraction until final energy use, is splitted up in different steps. The green variables are calculations of the red variables and can therefore theoretically be excluded from the model, but useful when analyzing model results, since they can be compared to real statistics and trends. From Figure 1 it can be seen that the following variables can be seen as nodes in the total energy flow.

supply_1 (R, primary, t)	The amount of primary energy extracted in each region	[EJ/yr]
supply_tot (R, primary, t)	The amount of primary energy used in a region after import/export	[EJ/yr]
en_conv (R, e_in,e_out, type,t)	The amount of energy converted from e_in to e_out expressed in primary energy terms.	[EJ/yr]
supply_2 (R, second_in, t)	The amount of energy that goes back into the energy conversion box to be converted a second time (H2 och ELEC).	[EJ/yr]
energy_prod (R, e_out, t)	The amount of energy carriers coming out of the conversion module in secondary energy (demand) terms, i.e. after energy losses.	[EJ/yr]
energy_deliv (R, e_out, t)	The amount of energy that meets the exogenously given energy demand (after import/export of H2, CTL/GTL and BTL).	[EJ/yr]
cg_heat (R, t)	The amount of heat produced using co-generation technologies	[EJ/yr]
heat_decarb (R, t)	Additional heat demand if the model chose to use CCS	[EJ/yr]
elec_decarb (R, t)	Additional electricity demand if the model chose to use CCS	[EJ/yr]
tot_CSP (t)	Total amount of energy from concentrating solar power (CSP)	[EJ/yr]

4.1.2 Import and export

There are two steps along the entire energy flow where regions can trade with each other. The primary energy sources "fuels" (bio, coal, oil, NG, and uranium for nuclear) can be traded before the primary energy sources are converted to secondary energy carriers "sec" (BTL, CTL/GTL, H2) between the variables supply_1 and supply_tot. The energy carriers can be traded between the variables energy_prod and energy_deliv, see Figure 1. The following variables are used to balance the energy flow that is imported and exported between regions.

imp_prim (R, fuels, t)	The amount of primary energy sources imported to a region	[EJ/yr]
imp_sec (R, sec, t)	The amount of secondary energy sources imported to a region	[EJ/yr]
exp_prim (R, fuels, t)	The amount of primary energy sources exported from a region	[EJ/yr]
exp_sec (R, sec, t)	The amount of secondary energy sources exported from a region	[EJ/yr]
imp_prim_from (R_imp, R_exp, fuels, t)	Primary energy trading flows	[EJ/yr]
imp_sec_from (R_imp, R_exp, sec, t)	Secondary energy trading flows	[EJ/yr]

4.1.3 Transportation

The main energy flow within the transportation sector is captured in variable trsp_energy. To keep track of the amount of electricity that is used in the transportation the variable elec_trsp is introduced. The amount of elec_trsp should then be added to the total electricity demand. Fuels for buses as well as fuels for ships are calculated from mirroring the shares of fuels and technologies used for trucks. Ships are, however, assumed to not be able to run on electricity, i.e., no HEVs and PHEVs are allowed in the shipping sector. If HEVs and PHEVs are used for trucks there will be a gap in the fuel use for ships. This gap is assumed to be filled with additional ship fuels by extending the shares equally much in variable extra_ship_fuel.

trsp_energy (R, trsp_fuel, e_type, trsp_mode, t)	The amount of energy used for transport	[EJ/yr]
elec_trsp (R, t)	The amount of electricity for the transportation sector	[EJ/yr]
extra_ship_fuel (R, trsp_fuel, ic_fc, ship_mode, t)	Additional ship fuel since ships cannot run on elec	[EJ/yr]

4.2 Calculating investments

cap_invest (R, e_in, e_out, type, t)	New investments in energy conversion technologies	[TW]
eng_invest (R, trsp_fuel, e_type, vehicle, t)	New investments in engines/vehicles	[Gvehicles]
infra_invest (R, trsp_fuel, t)	New investments in infrastructure	[TW]
capacity (R, e_in, e_out, type, t)	Aggregated capacity stock energy conversion techn	[TW]
engines (R, trsp_fuel, e_type, vehicle, t)	Aggregated capacity stock engines/vehicles	[Gvehicles]
infra (R, synfuel_gas, t)	Aggregated capacity infrastructure	[TW]

4.3 Calculating emissions and CCS

agg_emis	Total emissions	[MtC]
c_emission_global (t)	Annual global emissions	[MtC]
c_emission (R, t)	Annual emissions per region	[MtC]
c_capt_fos (R, t)	Annual amount of carbon captured from fossil fuels	[MtC]
c_capt_bio (R, t)	Annual amount of carbon captured from biomass	[MtC]
c_capt_tot (R, t)	Annual amount of carbon captured from fossil fuels and biomass	[MtC]
c_capt_agg	Total amount of captured carbon for all regions and time steps	[MtC]
carb_ctrb (T_all, T_all_copy)	Carbon contribution generated by the carbon cycle module	[MtC]
atm_ccont (T_all)	Atmospheric CO_2 concentration generated by the carbon cycle module	[ppm]

4.4 Calculating costs

cost_fuel (R, t)	Cost for extracting primary energy sources	[GUSD]
tot_trspcost_prim (R, t)	Cost for trading primary energy sources	[GUSD]
tot_trspcost_sec (R, t)	Cost for trading secondary energy carriers	[GUSD]
cost_cap (R, t)	Cost for investing in energy conversion technologies	[GUSD]
OM_cost (R, t)	Operation and maintenance cost	[GUSD]
cost_c_bio_trsp (R, t)	Additional transportation cost if applying CCS on bioenergy	[GUSD]
cost_c_strg (R, t)	Cost for storing carbon	[GUSD]
tax (R, t)	Cost if applying carbon taxes to the model	[GUSD]
annual_cost (R, t)	Sum of all annual costs in the model	[GUSD]
tot_cost	Total cost for the entire energy system	[GUSD]

5. Equations

In this section we describe the equations (constraints and the objective function). The equations drive the model to fulfill all energy balances, keep track of the energy sources and emissions made as well as calculating all costs. The major energy flows in GET-RC 6.1 are illustrated in Figure 1. Note that cost and emission variables are not included in Figure 1.See Figure 20 for a more complete model illustration.